B16 Algorithms and Data Structures 1 - Notes

Andrea Vedaldi

Academic Year 2023-24 (version 2.0)

Contents

Introduction **5**

1 Problems and algorithms [revision] **7**
1.1 Correctness . . . 8
1.2 Complexity . . . 11

2 Sorting in quasilinear and linear time **15**
2.1 Merge sort [revision] . . . 15
2.2 A lower bound on the complexity of sorting . . . 19
2.3 Sorting in linear time . . . 20

3 Elementary data structures **23**
3.1 Passing data by value or reference . . . 23
3.2 Arrays . . . 24
3.3 Stacks . . . 26
3.4 Queues . . . 29
3.5 Linked lists . . . 32

4 Binary trees **37**
4.1 Size and height of a binary tree . . . 38
4.2 Elementary representation . . . 40
4.3 Traversing a tree: depth and breadth first . . . 42
4.4 Binary search trees . . . 45
4.5 An alternative representation of complete trees . . . 48

5 Heaps **53**
5.1 Restoring the heap property . . . 53
5.2 Building a heap . . . 56
5.3 HeapSort . . . 58
5.4 Priority queues . . . 61

6 Hashing **65**
6.1 Hashing via chaining . . . 65
6.2 Average cost analysis of hashing . . . 66
6.3 C++ implementation of hash tables . . . 67
6.4 Designing hash functions . . . 70
6.5 Proof of theorem H2 (optional) . . . 71

7 Graphs **73**
7.1 Formal definition of graph . . . 73
7.2 Representing graphs using adjacency matrices and lists . . . 74
7.3 Weighted graphs . . . 74

7.4 Paths . . . 75
7.5 Cycles . . . 75
7.6 C++ implementation . . . 76

8 Shortest paths **81**
8.1 Definition of shortest path . . . 81
8.2 Existence of shortest paths . . . 82
8.3 Optimal substructure of shortest paths . . . 83
8.4 A compact representation of shortest paths . . . 83
8.5 Versions of the shortest paths problem . . . 84
8.6 Bellman-Ford's algorithm for the SSSP . . . 84
8.7 Floyd-Warshall's algorithm for the APSP . . . 87
8.8 Dijkstra's algorithm for the SSSP . . . 90
8.9 Using a priority queue . . . 91

A Appendix (optional) **95**
A.1 Move semantics . . . 95
A.2 Universal references . . . 98

B Locality-sensitive hashing (optional) **103**
B.1 Correctness analysis . . . 104
B.2 Proof of the LSH retrieval theorem (optional) . . . 106
B.3 C++ implementation of an LSH table (optional) . . . 107

Introduction

These notes cover the material for *B16 Part 3 - Algorithms and Data Structures 1*. This part of the course focuses on the study of *algorithms* and their theory. It also further explores the practical implementation of algorithms using the C++ language and its Standard Template Library (STL), which are the focus of Parts 1 and 2 of the course.

A C++ implementation of each algorithm in these notes is provided and discussed, further exploring the features of C++ and the STL (one or two of these features requires the use of C++14 or above). In the same spirit, the example sheet (example sheet 3) asks you to reflect on theoretical questions about algorithms, but also to implement them, completing provided C++ code and running it.

All the material is examinable *except* the parts marked as **Revision** or **Optional**. The parts marked as **Revision** review content already covered in A2 in the previous year. This content is not directly examined, but it is propaedeutic for this course, and you will use it implicitly.

The source code for the exercises is available on GitHub. The code repository is designed to be easily used in a Codespace, a virtual machine that allows you to edit, compile and run C++ programs in a browser (if you prefer, you can run this code on your personal machine too). Instructions on how to use a Codespace are given in the GitHub repository.

These notes and the example sheet are available in PDF and HTML format. All the course materials are available at this page.

Chapter 1

Problems and algorithms [revision]

Informally, an *algorithm* is a *computational procedure* that solves a certain *computational problem.*

A computational problem is a specification of:

1. a type of input;
2. a type of output;
3. and a relationship between the two.

For example, the *integer sorting* problem can be described as follows:

Integer sorting problem (definition):

- **Input:** A sequence $A = (A_0, \dots, A_{n-1})$ of n integers A_i.
- **Output:** The same sequence as A, but with the elements permuted such that $A_{i-1} \leq A_i$ for all $i = 1, \dots, n-1$.

A problem *instance* is obtained by choosing a specific *value* for the input. For example, an instance of the integer sorting problem is obtained choosing A to be a specific sequence of numbers, such as $A = (4, 2, 2, 3)$.

An algorithm is a procedure that can solve *any* instance of the problem. For example, the following algorithm `InsertionSort` sorts any integer sequence passed as input, thus solving the integer sorting problem:

InsertionSort(A)**:**

1. For $i = 1, 2, \dots, |A| - 1$:
   1. For $j = i, i-1, \dots, 1$:
      1. If $A_{j-1} \leq A_j$, break this loop and continue with the outer loop.
      2. Otherwise, swap A_{j-1} and A_j.

Usually we do not specify an algorithm using a specific programming language. Rather, we provide an abstract or mathematical description of the computational steps carried out by the algorithm, which can then be implemented using (almost) any language. Often this description is given in *pseudo-code*, as in the `InsertionSort` example above.

The importance of algorithms is that they allow us to study different computational strategies for solving a problem regardless of the implementation details such a the language or the computer system used. In particular, it allows us to focus on studying two key aspects, correctness and complexity, discussed next.

We are often interested in making algorithms as generally-applicable as possible. For instance, we may be interested in sorting not just integers, but also real numbers, strings, or any other type of objects that can be ordered. This motivates introducing the more abstract *sorting* problem:

Sorting problem (definition):

- **Input:** A sequence $A = (A_0, \ldots, A_{n-1})$ of n objects endowed with an order relation $\leq$.
- **Output:** The same sequence as A, but with the elements permuted such that $A_{i-1} \leq A_i$ for all $i = 1, \ldots, n-1$.

This version of the problem only requires that the symbol $\leq$ (the order relation) is defined. For example, $A = (\text{Dijkstra}, \text{von Neumann}, \text{Minsky}, \text{Knuth})$ is also a valid instantiation of the sorting problem if we define $A_i \leq A_j$ to be the the *lexicographical order* between strings.

If you inspect `InsertionSort`, you will notice that the algorithm never uses the fact that the input objects A_i are integers, but only requires the order relation $\leq$ to be defined. This means that `InsertionSort` works for integer sorting, but also fro the more general sorting problem without modifications.

The benefit of more general algorithms is that they can applied in a wider variety of practical situations. However, this does not mean that the most general algorithm is necessarily the best choice for a specific application. For example, if in our application we need to sort integers specifically, we may be able to use a more specific algorithm than `InsertionSort`. While the result is the same (i.e., to sort the elements), the more specific algorithm may have other benefits, such as running faster or with less resources. We will give later the example of `CountingSort`, a fast sorting algorithm that only works by considering a less-general version of the sorting problem (i.e., by making more assumptions about the data).

1.1 Correctness

The most important property of any algorithm is to be *correct*. This means that, for any problem instance (e.g., any specific sequence of numbers in the integer sorting problem), the algorithm must eventually *terminate* producing an output that solves the problem (e.g., a sorted permutation of the sequence).

Because the problem and the algorithm are mathematical descriptions, proving correctness amounts to providing a mathematical proof.

In most cases, writing such a proof in full requires including many details that distract from the main idea that makes the algorithm work. Thus, we will usually provide abridged versions of such proofs, focusing on the key concepts without discussing all trivial details.

Proving the correctness of algorithms generally uses two tools: *invariants* and *mathematical induction*.

As the program runs, its state (roughly speaking, the content of the memory of the computer and the program counter) keeps changing. An invariant is a property P of the *state* which is always satisfied while the algorithm executes.

An invariant $P(i)$ is often parameterized with an integer $i = 1, 2, \ldots$ corresponding to an iteration counter in the algorithm.

For example, an invariant applicable to line 1.1 of `InsertionSort` is:

> **Sorted suffix invariant $P(i)$ (definition)**
>
> The sequence A is the same as the original, but with the first i elements permuted in sorted order, such that $A_{j-1} \leq A_j$ for all $j = 1, \ldots, i-1$.

Correctness is proved by showing that the invariant is *maintained* during the execution of the program. Such a proof is done by induction. First, one proves the base case $P(1)$ (often corresponding to the fact that the invariant is trivially true before the first iteration of the algorithm, i.e., right before the algorithm does anything at all). Then, one proves the inductive step: if $P(i)$ is true at a certain iteration of the algorithm, then $P(i+1)$ becomes true at the next iteration, due to the computations performed. By induction, this shows that, iteration after iteration, the algorithm satisfies properties $P(1), P(2), \ldots$.

Eventually, the algorithm satisfies a stopping condition and terminates. For instance, `InsertionSort` can stop when $P(n)$ becomes true, where $n = |A|$ is the number of elements in the input sequence. This is because $P(n)$ states that the entire sequence is sorted, thus solving the problem.

In this example, invariant $P(1)$ must be true before the algorithm has a chance to do anything, so it must be a property of the input. Input properties amount to assumptions: they are called **preconditions** and they are part of the problem specification. The stopping case $P(n)$ matches instead the specification of the output, and is thus called a **postcondition**.

For example, in `InsertionSort` the sorted suffix property $P(1)$ is trivially true for $n = 1$ for any input sequence (because a sequence of just one element is always sorted). When $P(n)$ is true, the entire sequence is sorted, satisfying the postcondition.

In general, algorithms are complex and proving correctness may require applying mathematical induction to several parts of them. Often, this can be done by breaking down the algorithm into sub-algorithms, analysing those individually, and then combining the results of the individual analyses into a conclusion about the algorithm as a whole. For example, we can rewrite `InsertionSort` as follows:

> InsertionSort(A)**:**
>
> **Precondition** and **postcondition** as in the sorting problem.
>
> 1. For $i = 1, 2, \ldots, n-1$:
>   1. Call Insert(A, i)

where most of the pseudo-code has been moved to the algorithm `Insert`:

> Insert(A, i)**:**
>
> - **Precondition:** $P(i)$, meaning that the first i elements of the sequence A are sorted.
> - **Postcondition:** $P(i+1)$, meaning that the first $i+1$ elements of the sequence A are sorted.
>
> 1. For $j = i, i-1, \ldots, 1$:
>   1. If $A_{j-1} \leq A_j$, stop
>   2. Otherwise, swap A_{j-1} and A_j

We have thus extracted the *inner loop* of `InsertionSort` as a separate algorithm `Insert` . With this, we can prove that `InsertionSort` is correct *assuming* that `Insert` is correct, as follows.

We have already shown that $P(1)$ is correct at line 1.1 of `InsertionSort` for the first iteration $i = 1$. Assume that $P(i)$ is true at the beginning of the i-th iteration. This means that the first i elements of the sequence are sorted, which matches the precondition of `Insert`. Hence, calling `Insert` on step 1.1 (and assuming the correctness of the latter) results in the first $i+1$ element be sorting, thus making $P(i+1)$ true and proving the inductive step.

> **Remark.** While this proof may seem fairly verbose for what it shows, it is still an abridged version of a rigorous proof. For instance, we did not explicitly list preconditions on the index i in `Insert`; instead, we have implicitly assumed that a valid index i is provided. What would be this precondition?

To conclude this section, we should also prove the correctness of `Insert`. For that, we need another invariant:

> **Partially sorted prefix invariant** $Q(i, j)$ **(definition)**
>
> The sequence A is the same as the original, but with the first $i+1$ elements permuted such that:
>
> - $A_0 \leq \cdots \leq A_{j-1} \leq A_{j+1} \leq \cdots \leq A_i$;
> - $A_j \leq A_{j+1}$.

Note that $Q(i, j)$ is nearly the same as $P(i+1)$: The first $i+1$ elements are sorted, except that the specific inequality $A_{j-1} \leq A_j$ may not hold.

We can now prove the correctness of `Insert`. Note that $Q(i, i) = P(i+1)$ (why?). Hence, when the loop starts at step 1.1 of `Insert`, one has $j = i$ and $Q(i, j)$ is true (base case). Furthermore, if $Q(i, j)$ is true at the beginning of step 1.1, then:

- At step 1.1, The algorithm checks if the "missing" inequality $A_{j-1} \leq A_j$ is already satisfied. In this case $Q(i, j) = P(i + 1)$ and `Insert` stops because its postcondition is satisfied.
- Otherwise, at step 1.2 `Insert` swaps $A_{j-1} \leq A_j$. This makes $Q(i, j - 1)$ true, proving the inductive step.

Once again, this analysis may seem overly-detailed, and yet we have glossed over a few details. For instance, we should guarantee that the algorithm eventually stops. This is obvious since the for loop in `Insert` cannot run for more iterations than the length of the input sequence A.

1.1.1 C++ implementation of insertion sort

We now look at a possible implementation of insertion sort in C++. We use a `std::vector` as a container for the sequence of elements to be sorted. We implement this as a template to allow using an arbitrary element type `T` for the elements of the vector. Because we use templates, most of the code is defined in an header file (`.hpp`) rather than in an implementation file (`.cpp`).

File `insertion_sort.hpp`:

```
1  #ifndef __insertion_sort__
2  #define __insertion_sort__
3
4  #include <cassert> // for assert
5  #include <cstddef> // for std::size_t
6  #include <utility> // for std::swap
7  #include <vector>  // for std::vector
8
9  template <typename T> void insert(std::vector<T> &A, std::size_t i)
10 {
11     assert(i < A.size());
12     for (std::size_t j = i; j >= 1; --j) {
13         if (A[j - 1] <= A[j]) return;
14         std::swap(A[j - 1], A[j]);
15     }
16 }
17
18 template <typename T> void insertion_sort(std::vector<T> &A)
19 {
20     for (size_t i = 1; i < A.size(); ++i) {
21         insert(A, i);
22     }
23 }
24
25 #endif // __insertion_sort__
```

The functions `insert` and `insertion_sort` take as input a *reference* `std::vector<T>&` to a STL vector. This is because the algorithm reorders the vector *in-place* (i.e., it modifies the vector passed as input instead of returning a new sorted vector, which would entail duplicating the elements).

The function `assert` is used only for debugging purposes. In this case, it immediately raises an exception and terminates the program if the value of the index i is not valid for the input sequence. This is an illegal condition that the program should never encounter and denotes a *programming error* (instead a *runtime error* which is caused by invalid data being passed to the algorithm, or other error conditions such as running out of memory). Assertions allow to explicitly state these conditions in code (i.e., similar to a comment, they inform the programmer reading the code of what to expect) and allow to detect illegal states early in the execution of the program, which makes it easier to debug the program.

The following test driver (we use this name to denote a test program that executes or 'drives' a specific

function to be debugged) tests our implementation by sorting a few numbers.

File `insertion_sort_driver.hpp`:

```
#include "insertion_sort.hpp"
#include "utils.hpp" // for print

int main(int argc, char **argv)
{
    auto v = std::vector<float>{3, 1, 0, 18, 7};
    print(v, "Before sorting: ");
    insertion_sort(v);
    print(v, "After sorting: ");
}
```

```
Before sorting: [3, 1, 0, 18, 7]
After sorting: [0, 1, 3, 7, 18]
```

Note that there is no need for the sequence to contain numbers: we can sort sequences of arbitrary objects `T` as long as an ordering relation $\leq$ is defined between them. For example, because the C++ STL defines `operator<=` for strings, we can use the code above to also sort strings:

```
#include "insertion_sort.hpp"
#include "utils.hpp"

int main(int argc, char** argv)
{
    auto v = std::vector<std::string>{
        "Perlis", "Wilkes", "Hamming", "Minsky", "Wilkinson",
        "McCarthy", "Dijkstra", "Bachman", "Knuth",
    };
    print(v, "Before sorting: ");
    insertion_sort(v);
    print(v, "After sorting: ");
}
```

```
Before sorting: [Perlis, Wilkes, Hamming, Minsky, Wilkinson, McCarthy, Dijkstra, Bachman, Knuth]
After sorting: [Bachman, Dijkstra, Hamming, Knuth, McCarthy, Minsky, Perlis, Wilkes, Wilkinson]
```

This flexibility is an example of the power of templates. There is a limitation however: while the sorted elements can be of an arbitrary type `T`, the container *must* be a `std::vector<T>`. The STL already provides a number of algorithms, including sorting, implemented in an even more generic manner via a more advanced usage of templates. Specifically, they support different types of containers by accessing the elements via *iterators*. We will give an (optional) example of this technique as we look at `MergeSort` in the next chapter.

1.2 Complexity

The complexity of an algorithm is a measure of the amount of time or space (memory) required to run it. In practice, the exact time and space complexity depend on implementation details such as the hardware and the programming language used. Even so, the dominant factor in determining the cost is the nature of the algorithm and the size of the problem instance being processed. Namely, if n is the size of the problem and if $f(n)$ is the actual time in seconds that a specific implementation of an algorithm requires to solve a specific problem instance of that size, we can generally find a bound $f(n) \leq ag(n)$ where $a > 0$ is a constant (valid for a given real-world implementation of the algorithm) that "hides" the dependency on the implementation details and $g(n)$ is a fixed function, valid for all large n regardless of implementation details.

Formally, we characterize the complexity by saying that the time (or space) cost function $f(n)$ is Big-O of

$g(n)$, meaning that:

> **Big-O notation (definition)**
>
> Given two non-negative functions f and g defined on the natural numbers $\mathbb{N}$, we say that f is Big-O of g, and write $f(n) \in O(g(n))$, if, and only if, there exist constants $a > 0$ and n_0 such that
>
> $$\forall n \geq n_0 : \quad f(n) \leq ag(n).$$

We can estimate the Big-O complexity of an algorithm by deriving an upper bound on the number of steps required to solve any problem instance via a mathematical analysis. This assumes that individual steps in the algorithm map to steps of a fixed cost when run on an actual CPU. For sorting algorithms, we can for instance estimate the complexity by counting the number of swap or comparison operations performed. This assumes that the sequence to be sorted is represented by a data structure such as an array that supports fast random access, so that reading, comparing and swapping arbitrary elements in the array is fast (specifically, constant time $O(1)$ independent of the size of the array or the specific elements).

When estimating the complexity, it is important to consider the *worst case* scenario. Often the speed of an algorithm depends on the specific problem instance inputted. Since the bound must be universally valid, we must consider the worst input that we can supply to the algorithm.

For instance, the procedure `Insert` above may stop the first time step 1.1 is performed; this happens if the array is already sorted. Instead, in the worst case, the inserted element is the smallest in the array, and thus the loop iterates $i - 1$ times. Thus the complexity is $O(i)$ and we say that `Insert` has *linear complexity*. Furthermore, since `InsertionSort` calls `Insert` with $i = 1, 2, \ldots, n-1$, the overall complexity is $\sum_{i=1}^{n-1} i = n(n-1)/2 \in O(n^2)$. We thus say that `InsertionSort` has *quadratic complexity*.

The space complexity of insertion sort is $O(n)$, as we need an amount of space proportional to the size of the array. A nice property of insertion sort is that it is an *in place* algorithm, meaning that no additional memory is required besides that used to store the input (or, more precisely, it requires a constant amount of additional memory independent on the input size). Other legitimate sorting algorithms (e.g., merge sort) do require additional memory (although they generally are all $O(n)$ as far as the *order* of the space complexity is concerned).

Sometimes we are interested in characterizing lower bounds to the complexity of algorithms instead of upper bounds. For this, we use the Big-Ω notation:

> **Big-Ω notation:**
>
> Given two non-negative functions f and g defined on the natural numbers $\mathbb{N}$, we say that f is Big-Ω of g, and write $f(n) \in \Omega(g(n))$, if, and only if, there exist constant $a > 0$ and n_0 such that:
>
> $$\forall n \geq n_0 : \quad f(n) \geq ag(n).$$

If g is both a lower bound and an upper bound we use the Big-Θ notation instead:

> **Big-Θ notation:**
>
> We say that f is Big-Θ of g, and write $f(n) \in \Theta(g(n))$, if, and only if, f is simultaneously Big-O and Big-Ω of g.

As an example of these concepts, consider the function $f(n) = n^2 + \cos(4\pi n) + 1$, as in 1.1. This function is $O(n^2)$ because $f(n) \leq \frac{3}{2}g(n)$ for all $n \geq 2$. It is also $\Omega(n^2)$ because $f(n) \geq g(n)$ for all $n \geq 0$. Hence, it is $\Theta(n^2)$ as well.

For example, as we have seen above, there are worst-case sequences for which insertion sort requires n^2 operations to complete. This means that its complexity is not only $O(n^2)$, but also $\Omega(n^2)$ and thus $\Theta(n^2)$.

When we characterize the complexity of an algorithm, we seek for an upper bound to its complexity (Big-O), but we are we are generally interested in giving the *tightest possible bound*. For instance, a function f which is

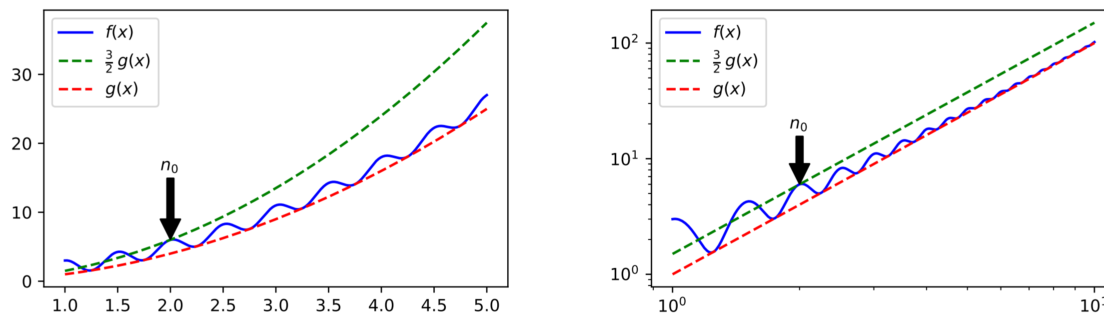


Figure 1.1: Example of Big-O and Big-Omega bounds (right in log-log space).

$O(n)$ is *also* $O(n^2)$, but we prefer to characterize f as $O(n)$ because the latter bound is stricter and thus more informative. If we can find a complexity order which is simultaneously a lower and an upper bound (Big-Θ), this is the tightest possible and thus the best (asymptotic) characterization of the algorithm complexity.

Note that the fact that the complexity of insertion sort is $\Theta(n^2)$ *does not* imply that it *always* takes in the order of n^2 operations to complete; on the contrary, in the best case that the input sequences are already sorted, insertion sort finishes in $O(n)$ time (is it possible to do better than that with any algorithm?). Unless otherwise specified, the complexity always refers to the worst case: saying that the complexity of insertion sort is $\Theta(n^2)$ means that the algorithm requires at most in the order of n^2 operations for *all* input sequences, i.e., at least in the order of n^2 operations for at least *one* worst-case input sequences.

Chapter 2

Sorting in quasilinear and linear time

In the previous chapter, we have revised the notion of algorithmic correctness and complexity. In this chapter, we further examine these ideas using as examples other sorting algorithms. We revisit merge sort (covered last year in A2), an algorithm with complexity $\Theta(n \log n)$, much better than insertion sort from the last chapter, which has complexity $\Theta(n^2)$.

We then prove a *lower bound* on the complexity of a large class of sorting algorithms, from which we can conclude that $O(n \log n)$ is the *best possible* worst-case sorting complexity.

Finally, we show that faster sorting algorithms are possible if more assumptions are made on the data to be sorted.

2.1 Merge sort [revision]

You should already be familiar with the concept of **merge sort** from A2. Recall that this algorithm works by splitting a sequence A into two (roughly) equally-sized subsequences A_1 and A_2, sorting them recursively via merge sort, and finally merging the results. This recursive algorithm is given in pseudo-code as follows:

MergeSort(A) :

Preconditions and **postconditions** as in the sorting problem.

1. If $|A| = 1$, return.
2. Let $i \leftarrow \lfloor |A|/2 \rfloor$.
3. Let $B \leftarrow (A_0, \ldots, A_{i-1})$.
4. Let $C \leftarrow (A_i, \ldots, A_{|A|-1})$.
5. Call MergeSort(B).
6. Call MergeSort(C).
7. Set $A \leftarrow$ Merge(B, C).

The `MergeSort` algorithms uses internally the `Merge` algorithm to merge the two subsequences B and C after sorting them. This is given in pseudo-code as follows:

Merge(B, C) :

Precondition: B and C are sorted arrays.

Postcondition: Returns an array A which contains a copy of the elements of B and C in sorted order.

1. Let $i \leftarrow 0$ and $j \leftarrow 0$.
2. Reserve space for a sequence A of $|B| + |C|$ elements.
3. While $i < |B|$ and $j < |C|$:

        1. If $B_i \leq C_j$:
            1. Set $A_{i+j} \leftarrow B_i$.
            2. Set $i \leftarrow i + 1$.
        2. Else:
            1. Set $A_{i+j} \leftarrow C_j$.
            2. Set $j \leftarrow j + 1$.
4. While $i < |B|$:
    1. Set $A_{i+j} \leftarrow B_i$.
    2. Set $i \leftarrow i + 1$.
5. While $j < |C|$:
    1. Set $A_{i+j} \leftarrow C_j$.
    2. Set $j \leftarrow j + 1$.
6. Return A.

The complexity of `MergeSort` can be analysed as follows. Let $f(n)$ be the cost of running `MergeSort` on an array of size n and assume for simplicity that n is a power of two. The cost is proportional to n (due to the call of `Merge`, which is clearly a linear time algorithm) plus the cost $2f(n/2)$ of calling `MergeSort` on two subsequences of size $n/2$. This gives us a *recurrence relation* defining the function $f(n)$:

$$f(1) = 1, \qquad f(n) = n + 2f(n/2).$$

We show that the formula $f(n) = n(\log_2 n + 1)$ solves the recurrence relation. We can verify by substitution that the formula is correct for $n = 1$. Consider a value $n > 1$, also a power of two, and assume that the formula is correct for $n/2$. Then we have

$$f(n) = n + 2f(n/2) = n + 2(n/2)(\log_2 n/2 + 1) = n(\log_2 n + 1)$$

which verifies the formula for n.

We thus conclude that the complexity of merge sort is $O(n \log n)$ (where we use log instead of $\log_2$ as they are equal up to a constant factor). This is known as **quasilinear** or **log-linear complexity**.

2.1.1 C++ implementation of merge sort [optional]

This example implementation is optional as it showcases slightly more advanced features of the STL, namely using iterators and move semantics.

We can implement merge sort in C++ as follows.

File `merge_sort.hpp`:

```
#ifndef __merge_sort__
#define __merge_sort__

#include <iterator>
#include <vector>

template <typename I>
std::vector<typename std::iterator_traits<I>::value_type>
merge(const I &begin, const I &middle, const I &end)
{
    using T = typename std::iterator_traits<I>::value_type;
    auto merged = std::vector<T>{};
    merged.reserve(end - begin);
    auto i = begin;
    auto j = middle;
```

```
16      while (i != middle && j != end) {
17          if (*i <= *j) {
18              merged.push_back(*i++);
19          } else {
20              merged.push_back(*j++);
21          }
22      }
23      while (i != middle) merged.push_back(*i++);
24      while (j != end) merged.push_back(*j++);
25      return merged;
26  }
27
28  template <typename I> void merge_sort(const I &begin, const I &end)
29  {
30      if (end - begin <= 1) { return; }
31      auto middle = std::distance(begin, end) / 2 + begin;
32      merge_sort(begin, middle);
33      merge_sort(middle, end);
34      auto sorted = merge(begin, middle, end);
35      std::move(sorted.begin(), sorted.end(), begin);
36  }
37
38  #endif // __merge_sort__
```

The implementation closely matches the pseudo-code. It uses STL *iterators* to represent generic sequences. An **iterator** is a STL object that represents a position in a STL container. For example, if `A` is a `std::vector<int>`, then `auto i = A.begin()`, or `auto i = begin(A)`, creates an iterator to the first element of the vector. You can think of `i` as a fancy pointer: the pointed value is accessed by *dereferencing* the iterator via the syntax `*i`. You can also increment an iterator (`i++` or `++i`) to access the following element, just as if it was a pointer.

The iterator `A.end()`, or `end(A)`, points to a virtual element[1], sometimes called a *sentinel*, which is one position after the end of the container. One checks if `i == A.end()` in order to determine if the end of the sequence has been reached

Iterators also have their own version of the "pointer math". The difference between iterators is the number of times an iterator has to be incremented (or decremented, if the difference is negative) to reach the other; for instance, for a vector `A`, we have `end(A) - begin(A) == A.size()` because `begin(A)` must be incremented `A.size()` times to reach `end(A)`.

The implementation uses templates in order to work with any type of iterator. Note that `I` in the code is the type of the iterator rather than the type of the iterated value. The iterated value type is obtained by accessing the `value_type` member of `I` using the syntax `typename std::iterator_traits<I>::value_type`. For example, if `I` is the type of the iterator returned by `std::vector<int>{}.begin()`, then `using T = typename std::iterator_traits<I>::value_type` defines `T` to be `int`. This explains the syntax used in defining the return type of the `merge` function: it is a vector whose element type is the same as the iterated type (e.g., `int`).

Finally, note that, at the beginning of `merge`, the function allocates a *new* vector and reserves sufficient space to store all the elements in the merged sequence. This is because merge sort is *not* an in-place algorithm: it requires to use a temporary buffer as large as the original sequence to do its job. This is sometimes a

[1] This is a slight approximation: in the construction above, keys are assumed to be all distinct — if two identical keys are sampled, we need to discard the second occurrence and sample another one, meaning that keys cannot be assumed to be exactly i.i.d. This in turn makes certain sequences of slots to be slightly more probable than others. We ignore this effect as, in practice, the space of keys is usually very large, so the probability of sampling two identical keys is very small.

disadvantage compared to other algorithms that are in place (for example, insertion sort and the heap sort algorithm we will study later).

Note that, while `merge` returns a new vector as output, `merge_sort` modifies the input sequence in place. This is not strictly needed: `merge_sort` could just return a new vector; it is done here for compatibility with the interface to the other sorting algorithms we have implemented, which change the input sequence in place. In this manner, `merge_sort` can be used as a "drop-in" replacement for any of these algorithms.

Also, note in the code of `merge_sort` that the program "moves" the elements of the merged vector on top of the elements of the input vector using `std::move`. You could also use `std::copy` instead of `std::move` to do so, but moving can be more efficient by avoiding redundant copies.

The following test driver tests `MergeSort`.

File `merge_sort_driver.cpp`:

```
#include "merge_sort.hpp"
#include "utils.hpp"

int main(int argc, const char *argv[])
{
    auto v = std::vector<float>{1,  19, 2,  9,  12, 18, 4, 8, 5,  6,
                                17, 10, 11, 14, 16, 15, 7, 3, 13, 20};
    print(v, "Before merge sort: ");
    merge_sort(v.begin(), v.end());
    print(v, "After merge sort: ");
    return 0;
}
```

```
Before merge sort: [1, 19, 2, 9, 12, 18, 4, 8, 5, 6, 17, 10, 11, 14, 16, 15, 7, 3, 13, 20]
After merge sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Because the code is generic, it can also sort strings:

```
#include "merge_sort.hpp"
#include "utils.hpp"
#include <vector>

int main(int argc, char** argv)
{
    auto v = std::vector<std::string>{
        "Perlis", "Wilkes", "Hamming", "Minsky", "Wilkinson",
        "McCarthy", "Dijkstra", "Bachman", "Knuth",
    };
    print(v, "Before sorting: ");
    merge_sort(begin(v), end(v));
    print(v, "After sorting: ");
}
```

```
Before sorting: [Perlis, Wilkes, Hamming, Minsky, Wilkinson, McCarthy, Dijkstra, Bachman, Knuth]
After sorting: [Bachman, Dijkstra, Hamming, Knuth, McCarthy, Minsky, Perlis, Wilkes, Wilkinson]
```

In fact, the code works not only with arbitrary element types, but also with different container types. For instance, it can use a `std::array` instead of a `std::vector`:

```
#include "merge_sort.hpp"
#include "utils.hpp"
#include <array>

```

```
5  int main(int argc, char** argv)
6  {
7      auto v = std::array<std::string, 9>{
8          "Perlis", "Wilkes", "Hamming", "Minsky", "Wilkinson",
9          "McCarthy", "Dijkstra", "Bachman", "Knuth",
10     };
11     print(v, "Before sorting: ");
12     merge_sort(begin(v), end(v));
13     print(v, "After sorting: ");
14 }
```

```
Before sorting: [Perlis, Wilkes, Hamming, Minsky, Wilkinson, McCarthy, Dijkstra, Bachman, Knuth]
After sorting: [Bachman, Dijkstra, Hamming, Knuth, McCarthy, Minsky, Perlis, Wilkes, Wilkinson]
```

We can even just use a vanilla C array of objects (which is *not* a STL container)! In this case, `begin(v)` is just a (vanilla) pointer to the first element of the array, and `end(v)` is a pointer to one element past the last.

```
1  #include "merge_sort.hpp"
2  #include "utils.hpp"
3
4  int main(int argc, char** argv)
5  {
6      std::string v [9] = {
7          "Perlis", "Wilkes", "Hamming", "Minsky", "Wilkinson",
8          "McCarthy", "Dijkstra", "Bachman", "Knuth",
9      };
10     print(v, "Before sorting: ");
11     merge_sort(begin(v), end(v));
12     print(v, "After sorting: ");
13 }
```

```
Before sorting: [Perlis, Wilkes, Hamming, Minsky, Wilkinson, McCarthy, Dijkstra, Bachman, Knuth]
After sorting: [Bachman, Dijkstra, Hamming, Knuth, McCarthy, Minsky, Perlis, Wilkes, Wilkinson]
```

2.2 A lower bound on the complexity of sorting

So far, we have given sorting algorithms with a complexity of $O(n^2)$ and $O(n \log n)$. It is natural to ask what is the best possible speed achievable by any sorting algorithm.

For example, it is clear that any sorting algorithm is at least $\Omega(n)$ as it has to check at least once every element in the input sequence. However, there is a non-trivial gap between bounds $\Omega(n)$ and $O(n \log n)$.

This is a typical situation. We often have an estimate of the minimum amount of work that any algorithm requires to carry out to solve a given problem and we also have a specific algorithm that does more work than the minimum we could prove. In this situation, it is often very difficult to show whether the algorithm can be improved further or not (after all, our estimate of the minimum amount of work required is only that, an estimate, and the algorithm may already be optimal). However, for the sorting problem we *can* prove the following very remarkable result:

Theorem 2.1 (Lower bound on sorting complexity)**.** *Let $\mathcal{S}$ be an algorithm that solves the* sorting problem *by computing a permutation $\mathcal{S}(A)$ that sorts a sequence A. Furthermore, assume that $\mathcal{S}$ does not look a the element values (e.g., A_i), but only at pairwise comparisons between its elements (e.g., $A_i \leq A_j$). Then, the worst case time complexity of the algorithm $\mathcal{S}$ is at least $\Omega(n \log n)$. Hence, any sorting algorithm with complexity $O(n \log n)$ is asymptotically optimal.*

Stated in another way, there is no way of sorting a sequence faster than in $O(n \log n)$ steps if one is only allowed to perform pairwise element comparisons.

Proof. The proof is simple and yet insightful. There are $n!$ possible permutations A of the sequence $(1, 2, \dots, n)$ of n elements. Hence, as A varies in the set of all such permutations, applications of $\mathcal{S}(A)$ must produce $n!$ different permutations to "undo" them and sort these sequences. However, the number of different outputs that $\mathcal{S}$ can produce as different inputs are provided to it is limited by the number of comparisons performed. For instance, if $\mathcal{S}$ terminates after performing *one* pairwise comparison, it can generate at most two different outputs/permutations (regardless of the specific choice of input sequences A). This is because the algorithm is deterministic, and which output it produces only depends only on the result of the one comparison performed, which is either true or false; thus, there can be only two different outputs. Similarly, if $\mathcal{S}$ always terminates after at most two comparisons, it can generate at most 4 different outputs. After t comparisons, it can generate at most 2^t different outputs.

Since the algorithm must be able to generate all $n!$ possible permutations in order to sort all sequences A, for at least some sequences it must perform a number of comparisons t such that:

$$2^t \geq n!$$

In order to rewrite this in terms of a complexity order, note that:

$$\begin{aligned} 2^t \geq n! &= \underbrace{n \cdot (n-1) \cdots (n/2)}_{n/2 \text{ times}} \underbrace{(n/2-1) \cdots 2 \cdot 1}_{n/2 \text{ times}} \\ &\geq \underbrace{n \cdot (n-1) \cdots (n/2)}_{n/2 \text{ times}} \geq (n/2)^{n/2} = 2^{\frac{n}{2} \log \frac{n}{2}} \end{aligned}$$

Hence, we conclude that $t \geq \frac{n}{2} \log \frac{n}{2}$, which means that the complexity is at least $\Omega(n \log n)$. □

The remarkable thing about this result is its universality: no matter how smart you are in designing a sorting algorithm of this type, you can *never* sort faster than this (because of the need to perform a minimum number of pairwise comparisons). Hence, algorithms such as merge sort and heap sort are, in this sense, optimal.

There are some limitations of this result, however. First, this is an asymptotic argument: it is still possible to obtain practical gains by optimizing a given algorithm for a given problem size. Second, sometimes we are interested in the *average* (or *expected*) cost of an algorithm, not in its worst case. Thirdly, as we show next, if one is allowed to make further assumptions on the data and the algorithm, it is possible to improve the asymptotic order too.

2.3 Sorting in linear time

We now show that sorting algorithms faster than $\Omega(n \log n)$ are possible, but this requires to leverage more assumptions on the sequence to be sorted. For example, assume that the elements A_i are integers in the range 0 to $k-1$. Then, we can sort faster by *counting*:

CountingSort(A, k)**:**

Precondition. A sequence A whose elements are integers in the range 0 to $k-1$.

Postcondition. The sequence A has the same integer elements as before, but sorted in non-decreasing order.

1. Allocate a sequence C with k elements initialized to 0.
2. For $i = 0, \dots, |A| - 1$:
    1. Set $C_{A_i} \leftarrow C_{A_i} + 1$.
3. Let $i \leftarrow 0$ and $j \leftarrow 0$.
4. While $j < k$:
    1. If $C_j = 0$, then set $j \leftarrow j + 1$ and continue with line 4.
    2. Set $A_i \leftarrow j$.

3. Set $C_j \leftarrow C_j - 1$.
4. Set $i \leftarrow i + 1$.

This algorithm works by counting how many occurrences of each number in the range 0 to $k-1$ exist in the input sequence. It then reconstructs the sequence by emitting that number of occurrences of each number, in order.

The complexity of this algorithm is driven by the for loop at line 4. Since j is only updated occasionally during the loop, it is not immediately obvious how many time this loop runs. First, note that the "then" part of line 4.1 is executed at most k times, because each time j is incremented by one and its maximum value is $k-1$. Then, note that lines 4.2-4.4 are executed at most n times because each time i is incremented by one and its maximum value is $n-1$ (otherwise the algorithm would output more than the n input elements). Hence, the while loop at line 4 costs at most $O(n+k)$ operations. The cost of initializing the sequence at line 1 and 2 is also at most $O(n+k)$ operations. In short, the complexity of `CountingSort` is $O(n+k)$.

The following C++ implementation demonstrates this algorithm.

File `counting_sort_driver.cpp`:

```cpp
#include "utils.hpp"
#include <vector>

void counting_sort(std::vector<int> &A, size_t k)
{
    auto counts = std::vector<size_t>(k, 0);
    size_t i = 0;
    size_t j = 0;
    for (auto j : A) ++counts[j];
    while (j < k) {
        if (counts[j] == 0) {
            ++j;
            continue;
        }
        A[i++] = static_cast<int>(j);
        --counts[j];
    }
}

int main(int argc, char **argv)
{
    auto A = std::vector<int>{5, 3, 0, 1, 5, 3};
    print(A, "Before sorting: ");
    counting_sort(A, 6);
    print(A, "After sorting: ");
    return 0;
}
```

```
Before sorting: [5, 3, 0, 1, 5, 3]
After sorting: [0, 1, 3, 3, 5, 5]
```

Chapter 3

Elementary data structures

Data structures such as arrays, stacks, queues and linked lists are data containers that support certain operations efficiently, such adding a new element to the container or searching for one based on some attribute. Different data structures support different operations with different efficiency, and are thus appropriate in different scenarios.

In this chapter, we will look at basic data structures and corresponding algorithms. We will also provide example C++ implementations of them. However, all such structures and algorithms are *already* implemented in the C++ STL. Therefore, in "real life" you would never write such implementations by yourself, but use the standard ones.

Before discussing data structures and their algorithms, we need first to clarify our notation for references and values in the pseudo-code we use, which we do in the next section.

3.1 Passing data by value or reference

When passing an argument to a function, it is important to clarify if the argument is passed by value or by reference.

In C, you have the concepts of *value* type (`T`) and *pointer* type (`T*`). C++ adds the concept of *reference* type (`T&`), which can be thought of as a pointer which looks like a value . C++11 further adds the concept of rvalue reference (`T&&`); this is an advanced optional topic, briefly discussed below and in the Appendix.

In pseudo-code, we do not require such level of detail; instead, we borrow the simple implicit convention used in programming languages such as Python:

> **Convention**: passing arguments to functions in pseudo-code:
>
> 1. All the elementary types (numbers and strings) are passed by value.
> 2. All other types, including data structures, are passed by reference.

For example, if A is an array, i is an integer, and we define in pseudo-code the function `DoSomething` as follows:

> DoSomething(A, i) :
>
> 1. Set $A_0 \leftarrow 1$.
> 2. Set $i \leftarrow 1$.

then the following chunk of pseudo-code:

> TestDriver :
>
> 1. Set $A_0 \leftarrow 0$.

2. Set $i \leftarrow 0$.
3. Call DoSomething(A, i).
4. Print A_0 and i.

prints the values 1 and 0 because A, which is a data structure, is passed by reference and i, which is a number, is passed by value.

In C++, references must be denoted explicitly. In order to implement the example above in C++, we can implement the array `A` using a `std::vector<float>`, so the function `do_something` takes a `std::vector<float>&` as argument (note the `&` symbol, meaning that the argument is a reference to a vector), as in the following example:

```
#include <iostream>
#include <vector>

void do_something(std::vector<float>& A, int i)
{
    A[0] = 1;
    i = 1;
}

int main(int argc, char** argv)
{
    auto A = std::vector<float>{0};
    int i = 0;
    std::cout << "Before calling do_something(): "
      << A[0] << ", " << i << '\n';
    do_something(A, i);
    std::cout << "After calling do_something(): "
      << A[0] << ", " << i << '\n';
    return 0;
}
```

```
Before calling do_something(): 0, 0
After calling do_something(): 1, 0
```

3.2 Arrays

An **array** is a data structure storing a sequence of elements $A = (A_0, \ldots, A_{n-1})$ with efficient operations for reading and writing any element A_i in constant $O(1)$ time, independently of the index i and size n of the array. This is also called *random access* to differentiate it from other access patterns such as sequential (sequential access is in particular often easier to support efficiently in data structures, including non-arrays).

An array is usually implemented by mapping consecutive array elements to consecutive equally-sized memory blocks in the memory of the computer. Then, index i can immediately be mapped to the corresponding memory address by multiplying i by the block size and adding the result to the *base address*, i.e., the address of the first array element. In this manner, the CPU can access the relevant record in one step.

> **Remark.** The latter assumes that the hardware implementing the memory supports random access too (e.g., RAM or ROM). This may not be the case for all memory types: for instance, the array could be stored on a tape, in which case the CPU would have to wait for the tape to unroll to the correct position before being able to read or write an element! (While the idea of using a tape may seem anachronistic, tapes are still in common use today for archival and backup).

While accessing individual elements in an array is fast, *inserting* and *deleting* them is not. This is because existing elements must be shifted in memory in order to create a space to add a new element, or to fill the hole left by removing an element.

For instance, the following algorithm inserts an element x at position i in the array A:

ArrayInsert(A, i, x):

- **Precondition:** An array $A = (A_0, \ldots, A_{n-1})$, an index $0 \leq i \leq n$, and a new value x.
- **Postcondition:** The array A is $(A_0, \ldots, A_{i-1}, x, A_i, \ldots, A_{n-1})$.

1. Set $A \leftarrow (A_0, \ldots, A_{n-1}, *)$ // Extend the array by one element
2. For $j = n, \ldots, i+1$: // Iterate backward
    1. Set $A_j \leftarrow A_{j-1}$.
3. Set $A_i \leftarrow x$.

Intuitively, `ArrayInsert` works by replacing element A_n with A_{n-1}, A_{n-1} with A_{n-2} and so in order to make space for the new element x at position i.

In order to prove the correctness of `ArrayInsert`, we consider the invariant:

Array insertion invariant $P(j)$:

The current array is the same as the input A, except for an additional element at position j. In other words, the array is of the type:

$$(A_0, \ldots, A_{j-1}, *, A_j, \ldots, A_{n-1})$$

where $*$ denotes some arbitrary value.

At the beginning of the loop, $j = n$ and $P(n)$ simply states that the sequence is $(A_0, \ldots, A_{n-1}, *)$, which is the same as the input A plus one "free space" at the end. Assuming that at the beginning of each iteration of the for loop the array the invariant $P(j)$ is satisfied, we see that line 1.1 satisfies $P(j-1)$:

$(A_0, \ldots, A_{j-2},$	$A_{j-1},$	$*,$	$A_j, \ldots, A_{n-1})$	assuming $P(j)$
$(A_0, \ldots, A_{j-2},$	$A_{j-1},$	$A_{j-1},$	$A_j, \ldots, A_{n-1})$	after the copy operation at line 1.1
$(A_0, \ldots, A_{j-2},$	$*,$	$A_{j-1},$	$A_j, \ldots, A_{n-1})$	interpreted as $P(j-1)$

At the *end* of the last iteration of the loop, in particular, property $P(i)$ is satisfied. Line 2 then replaces $*$ at position j with the element x to be inserted in the array.

The cost of insertion is evidently $O(n)$ as, in the worst case, inserting at $i = 0$ requires shifting n elements. Also note that at line 1 of the algorithm we recreate the array with a placeholder for the new element at the end. Depending on implementation details, this may cost $O(1)$, if the memory buffer backing the array was pre-allocated for more than n element, or $O(n)$, if a new, larger memory buffer needs to be allocated from scratch, which usually involves copying all the old elements into the new buffer.

3.2.1 C++ implementation of arrays

C++ already implements arrays as `std::vector`. For demonstration purposes, we show how `ArrayInsert` can be implemented from scratch.

File `array.hpp`:

```
1  #ifndef __array__
2  #define __array__
3
4  #include <cassert>
5  #include <cstddef>
6  #include <vector>
7
8  template <typename T>
9  void array_insert(std::vector<T> &A, std::size_t index, const T &x)
10 {
```

```
    assert(index <= A.size());
    if (index == A.size()) {
        A.push_back(x);
    } else {
        auto i = A.size();
        A.push_back(A[i - 1]);
        for (--i; i > index; --i) { A[i] = A[i - 1]; }
        A[index] = x;
    }
}

#endif // __array__
```

Note that, due to the `std::vector` API, we need a special case for inserting the element at the very end of the array. This is because the member function `push_back` both extends the array by one element and sets the element value.

The following test driver runs this code, inserting a few numbers at the beginning of an array. The code uses the `utils.hpp` that defines a `print` function, which we use for conveniently print the content of containers.

```
#include "array.hpp"
#include "utils.hpp"

int main(int argc, char **argv)
{
    auto A = std::vector<float>{};
    print(A, "Array before inserting any elment = ");
    for (size_t i = 0; i < 5; ++i) {
        array_insert(A, 0, static_cast<float>(i));
        print(A, "Array after inserting " + std::to_string(i) +
                 " at position 0 = ");
    }
    return 0;
}
```

```
Array before inserting any elment = []
Array after inserting 0 at position 0 = [0]
Array after inserting 1 at position 0 = [1, 0]
Array after inserting 2 at position 0 = [2, 1, 0]
Array after inserting 3 at position 0 = [3, 2, 1, 0]
Array after inserting 4 at position 0 = [4, 3, 2, 1, 0]
```

3.3 Stacks

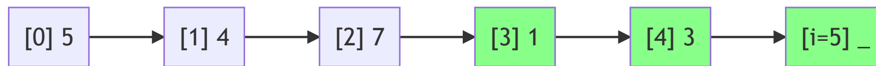
A **stack** is a container with two efficient operations: adding and removing an element from the top of the stack. This is also called a LIFO (last-in first-out) data structure.

We can represent a stack S with an array A plus an index i pointing at the top of the stack. Conventionally, i points to the first available unused space at the top, so that, when the stack is empty, $i = 0$.

For example, this is a stack with 3 elements (5, 4, 7) and space for six in total:

We can then push and pop element to and from the stack in time $O(1)$ by manipulating the index i. For

instance, pushing 1 and 3 on the stack results in the data structure:



Pushing on the stack amounts to writing a new value in A_i and then incrementing i:

StackPush(S, x) :

1. Set $S.A_{S.i} \leftarrow x$.
2. Set $S.i \leftarrow S.i + 1$.

Popping an element from the stack does the reverse:

StackPop(S) :

1. Set $S.i \leftarrow S.i - 1$.
2. Return $S.A_{S.i}$.

3.3.1 C++ implementation of stacks

We implement a stack as a C++ class with the following member functions: `push`, `pop`, `top`, `empty` and `full` (the last two to tell if the stack is empty or full). The stack is backed by a `std::vector` to represent the array `A`; the latter must be pre-allocated with sufficient space to store the stack at its fullest; in other words, pushing more than $|A|$ elements on the stack would cause an error in this basic implementation.

File `stack.hpp`:

```
#ifndef __stack__
#define __stack__

#include <cassert>
#include <cstddef>
#include <vector>

template <typename T> class Stack
{
  protected:
    std::vector<T> _storage;
    std::size_t _head;

  public:
    // Initialize a stack with the specified capacity
    Stack(std::size_t capacity) : _storage(capacity), _head{0} {}

    // Access the value at the top of the stack
    T &top()
    {
        assert(_head > 0);
        return _storage[_head - 1];
    }

    // Const-access the value at the top of the stack
    const T &top() const
    {
        assert(_head > 0);
        return _storage[_head - 1];
    }
```

```
    // Pop the value at the top of the stack
    void pop()
    {
        assert(_head >= 1);
        --_head;
    }

    // Copy a value to the top of the stack
    void push(const T &x)
    {
        assert(_head < _storage.size());
        _storage[_head++] = x;
    }

    // Check if the stack is empty
    bool empty() const { return _head == 0; }

    // Check if the stack is full
    bool full() const { return _head == _storage.size(); }
};

#endif // __stack__
```

Members `_storage` and `_head` are protected (we use the convention of prefixing private and protected members of a class with an underscore `_`). In order to preserve encapsulation and modularity, the user of the `Stack` class should not be able to access such implementation details.

The class provides two versions of the `top` function, one const and one non-const. They both return a reference to the top element in the stack, with the only difference that the reference is respectively const or non-const. When the stack itself is const, only const member functions can be invoked. In this case, the const accessor `top` still allows reading the top element on the stack.

You may also wonder why the `pop` function returns `void` instead of the popped element. This is because this would require to always copy this element back to the caller, even when the caller does not need it. Using `top` to access the top element and `pop` to remove it allows to fine-tune which operations are applied.

Finally, the `push` function copies the specified value to the top of the stack. A real-world implementation would also include an implementation of `push` that supports moving instead of copying the value, which can in some cases be much more efficient(see the optional appendix on move semantics).

The following test driver pushes and then pops a few numbers from a stack.

File `stack_driver.cpp`:

```
#include "stack.hpp"
#include <iostream>

int main(int argc, char **argv)
{
    // Create a stack with space for 10 elements
    auto stack = Stack<float>(10);

    // Push some numbers on the stack
    std::cout << "Pushing";
    for (int i = 0; i < 5; ++i) {
        stack.push(i);
```

```
13            std::cout << ' ' << i;
14        }
15        std::cout << '\n';
16
17        // Pop the numbers from the stack
18        std::cout << "Popping";
19        while (!stack.empty()) {
20            std::cout << ' ' << stack.top();
21            stack.pop();
22        }
23        std::cout << '\n';
24
25        return 0;
26    }
```

```
Pushing 0 1 2 3 4
Popping 4 3 2 1 0
```

3.4 Queues

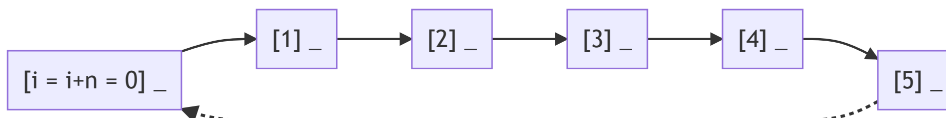
A **queue** is a container with two efficient operations: adding an element at the back of the queue and removing an element from the front of the queue. This is also called a FIFO (first in first out) data structure.

We can implement a queue in a manner similar to the stack, using an array A and and index i to keep track of the head of the queue. However, a complication with queues is that, as we keep adding and removing elements, eventually the index will fall off the boundary of the array even if not all elements in the array are used up by the queue. In other words, we may have $i > |A|$ even though the queue does not contain more elements than the array size.

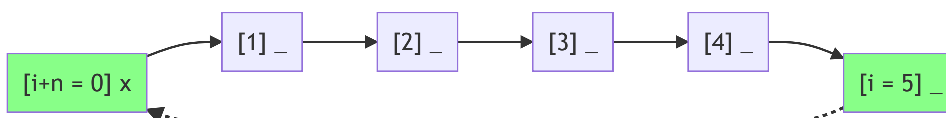
We solve this issue by allowing indices to "wrap" around the array in a circular fashion.

In order to do so, we represent a queue with an array A plus and index i pointing at the first non used element at the back of the queue and an integer n denoting the number of elements in the queue. Initially, with an empty queue, we have $i = 0$ and $n = 0$.

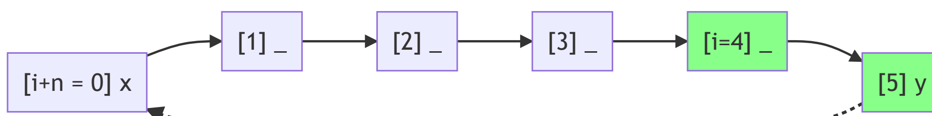
For example, an array A storing an empty queue with space for six elements in total looks like the following structure:



After enqueuing the value x, this looks like:



Note that the index i is now set to the value 5: this is the result of subtracting 1 from it, obtaining -1 and then "wrapping around" the array. n is set to 1 to indicate that there is one element in the queue, wrapping around (i.e., $5 + 1 = 6$ becomes 0). After enqueuing the value y, this becomes:



In pseudo-code, the `Enqueue` algorithm is given by:

Enqueue(Q, x) :

1. Set $Q.A_{Q.i} \leftarrow x$.
2. Set $Q.n \leftarrow Q.n + 1$.
3. Set $Q.i \leftarrow Q.i - 1$.
4. If $Q.i = -1$:
    1. Set $Q.i \leftarrow |Q.A| - 1$.

Line 1 stores x at the first empty space at back of the queue, pointed at by $Q.i$. Line 2 increases the size of the queue by one. Line 3 decreases $Q.i$ by one, so that $Q.i$ points to the new empty element at the back of the queue. Lines 4 and 4.1 check if $Q.i$ is -1, which is beyond the array boundaries. If so, they reset Q_i to $|Q.A| - 1$, pointing to the last element of the array (in other words, position -1 is remapped to position $|Q.A| - 1$).

The `Dequeue` algorithm is given by:

Dequeue(Q) :

1. Let $j \leftarrow Q.i + Q.n$
2. If $j \geq |Q.A|$:
    1. Set $j \leftarrow j - |Q.A|$.
3. Set $Q.n \leftarrow Q.n - 1$.
4. Return $Q.A_j$.

The main idea of this algorithm is the calculation of the index j pointing at the head of the queue. Normally, this is obtained by summing to the index $Q.i$, pointing at the back of the queue, the number of elements in the queue $Q.n$. However, if the resulting index exceeds the array boundaries, we "wrap around" by subtracting $|Q.A|$ from it (line 2.1).

3.4.1 C++ implementation of queues

We now introduce a simple C++ implementation of a `Queue` class.

File `queue.hpp`:

```
#ifndef __queue__
#define __queue__

#include <cassert>
#include <cstddef>
#include <vector>

template <typename T> class Queue
{
  protected:
    std::vector<T> _storage;
    size_t _position;
    size_t _size;

  public:
    // Create a queue with the specified capacity
    Queue(size_t capacity) : _storage(capacity), _position{0}, _size{0}
    {
        assert(capacity > 0);
    }

    // Access the element at the front of the queue
```

```
23      T &front() { return _storage[_head()]; }
24
25      // Const-access the element at the front of the queue
26      const T &front() const { return _storage[_head()]; }
27
28      // Add a new element to the back of the queue by copying
29      void enqueue(const T &value)
30      {
31          assert(_size < _storage.size());
32          _storage[_position] = value;
33          _size++;
34          if (_position == 0) {
35              _position = _storage.size() - 1;
36          } else {
37              _position--;
38          }
39      }
40
41      // Remove the element at the front of the queue
42      void dequeue()
43      {
44          assert(_size >= 1);
45          _size--;
46      }
47
48      // Check if the queue is empty
49      bool empty() const { return _size == 0; }
50
51      // Check if the queue is full
52      bool full() const { return _size == _storage.size(); }
53
54    protected:
55      // Return the index of the element at the front of the queue.
56      size_t _head() const
57      {
58          assert(_size >= 1);
59          auto index = _position + _size;
60          if (index >= _storage.size()) { index -= _storage.size(); }
61          return index;
62      }
63  };
64
65  #endif // __queue__
```

The class is similar to `Stack` and contains a direct implementation of the two algorithms above. Just as for `Stack`, the dequeue operation is broken down into `front()` (to access the element at the front of the queue) and `dequeue()` for removing the element. The function `enqueue()` adds an element to the front of the queue, initializing it with a copy of the value specified as argument.

The protected member function `_head()` returns the index of the front element and it is separated out as a function because this calculation needs to be performed in different parts of the code.

The following test driver uses this code.

File `queue_driver.cpp`:

```cpp
#include <iostream>

#include "queue.hpp"
#include "utils.hpp"

int main(int argc, char **argv)
{
    // Create a queue with space for a few elements
    auto queue = Queue<float>(5);

    // Keep pushing and popping elements from the queue for a while
    // wrapping around the ring buffer
    for (int repetition = 0; repetition < 3; ++repetition) {
        std::cout << "Enqueued";
        for (int i = 0; i < 3; ++i) {
            queue.enqueue(i);
            std::cout << ' ' << i;
        }
        std::cout << '\n';

        std::cout << "Dequeued";
        for (int i = 0; i < 3; ++i) {
            std::cout << ' ' << queue.front();
            queue.dequeue();
        }
        std::cout << '\n';
    }

    return 0;
}
```

```
Enqueued 0 1 2
Dequeued 0 1 2
Enqueued 0 1 2
Dequeued 0 1 2
Enqueued 0 1 2
Dequeued 0 1 2
```

3.5 Linked lists

Linked lists, similar to arrays, represent a *sequence* of elements $(A_0, \ldots, A_{n-1})$. However, differently from arrays, they support efficient insertion and deletion, but slow random access (sequential access is fast).

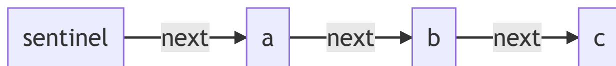
The idea of linked lists is to break the association, used in arrays, of the index of an element in the sequence with a particular memory address. In this manner, random access becomes slower as there is no one-step calculation that gives the memory address of an element, but insertions and deletions become fast because the memory locations of the elements already in the list need not to be changed when elements are added or taken away from the list.

Formally, a **singly-linked list** is a chain of **nodes**. We can represent a list node N as a structure with two fields:

- N.next is a pointer to the next node in the list. This pointer can be NIL if there is no next node (denoting the end of the list).
- N.value is the data associated to the node (e.g., a number or a string).

We represent the list as a whole using a pseudo node called a *sentinel*. The sentinel does not contain useful data; its only purpose is to point to the first actual node in the list. This is convenient because we can use a single object type to represent both the list and the individual nodes.

For example, a list containing the values “a”, “b”, “c” may look like:



Linked lists are particularly useful because they allow to quickly add new nodes at any point in the list. For instance, if Q is an existing node (possibly the sentinel), the following algorithm inserts a new node/value *after* Q:

$\text{ListInsertAfter}(Q, x)$:

1. Create a new node N.
2. Set $N.\text{next} \leftarrow Q.\text{next}$.
3. Set $N.\text{value} \leftarrow x$.
4. Set $Q.\text{next} \leftarrow N$.

The complexity of insertion is thus $O(1)$ vs $O(n)$ of an array.

Due to the chain-like structure, list elements must be accessed sequentially. For example, the following algorithm finds the *predecessor* of the first node N in list Q whose value matches a given query x:

$\text{ListFindPredecessor}(Q, x)$:

1. While Q and $Q.\text{next}$ are not `NIL`:
   1. If $Q.\text{next.value} = x$ return Q.
   2. Set $Q \leftarrow Q.\text{next}$.
2. Return `NIL`.

Finding the predecessor rather than the node itself can be useful to use routines such as `ListInsertAfter` that affect the node following the one specified.

3.5.1 C++ implementation of linked lists

A possible implementation of the node object and of `ListInsertAfter` in C++ is as follows.

File `list.hpp`:

```
#ifndef __list__
#define __list__

#include <cassert>
#include <memory>
#include <vector>

template <typename T> struct Node {
    T value;
    std::unique_ptr<Node<T>> next;

    Node() : value{}, next{nullptr} {}
    Node(const T &value, std::unique_ptr<Node<T>> next)
        : value{value}, next{std::move(next)}
    {
    }
};

template <typename T>
```

```
Node<T> *list_insert_after(Node<T> *node, const T &value)
{
    node->next =
        std::make_unique<Node<T>>(value, std::move(node->next));
    return node->next.get();
}

template <typename T, typename F>
Node<T> *list_find_predecessor(Node<T> *node, F predicate)
{
    for (; node && node->next; node = node->next.get()) {
        if (predicate(node->next->value)) return node;
    }
    return nullptr;
}

template <typename T>
std::vector<T> list_to_vector(const Node<T> &node)
{
    std::vector<T> v;
    for (Node<T> *current = node.next.get(); current;
         current = current->next.get()) {
        v.push_back(current->value);
    }
    return v;
}

#endif // __list__
```

We use a template to allow customizing the type of data `T` held by a node. We also use `unique_ptr` to manage ownership of the node objects.

The smart pointer `unique_ptr` was briefly mentioned in Part 1 and 2. Here, we look at it in more detail.

Differently form a regular C/C++ pointer, which has an unspecified ownership relation with the pointed object, an instance of `unique_ptr` is meant to own this object. This means that, when the `unique_ptr` object is deleted, the pointed object is deleted as well.

The concept of **object ownership** is not enforced at the level of the C++ language. Rather, it is a programming idiom where the owner of an object is the entity (usually another object) responsible for managing its lifetime.

In the most basic case, there is a single entity that owns an object (the STL also supports shared ownership via `shared_ptr`). This means that:

1. When the owner entity is destroyed, it can safely destroy all the objects it owns, because no other entity in the program will attempt to also destroy them (this is good because it is an error to attempt to destroy the same object twice);

2. Other parts of the program that want to use the owned object must ensure that the owner is still alive while they use it. This is because, when the owner is destroyed, then the owned object is destroyed as well and thus becomes invalid.

The name *unique* in `unique_ptr` means that this is the *only* pointer meant to own the object (not necessarily the only pointer to the object that exists in the program, though!). By having nodes in the list own the successive nodes via `unique_ptr`, we do not need to worry about calling `delete` to destroy the nodes when the list is destroyed. Destroying the root node causes a “chain reaction” that automatically destroys all the

other nodes in the list. This is also an example of the RAII paradigm.

Without smart pointers, you would normally use the `Node my_new_node = new Node{...}` syntax to dynamically create a new `Node` instance. With `unique_ptr`, you would use instead `std::unique_ptr<Node> my_new_node = std::make_unique<Node>(...)`, which does the same, but wraps the vanilla pointer in a `unique_ptr`, to denote ownership.

An interesting aspect of `unique_ptr` is the fact it supports transferring ownership, but this: (1) requires to use the `std::move` operator in the program, which makes the programmer's intent very explicit, and (2) still guarantees that a single owning pointer remains.

In the code above, this feature can be observed in the constructor that creates a new `Node` pointing to an existing node. Suppose that your code has a variable `my_old_node`, a `unique_ptr` to a `Node`. To use this constructor, you must use a syntax similar to `std::unique_ptr<Node> my_new_node = std::make_unique<Node>(value, std::move(my_old_node))`. This is a mouthful, but it is actually very explicit code. It means: create a new Node, wrapping it in a `unique_ptr`, and transfer ownership of the object pointed by `my_old_node` to it. After this operation is complete, `my_old_node` is invalid (i.e., it becomes a null pointer), to denote the fact that ownership has been transferred.

We also include a convenience function `list_to_vector` to copy the data in the list to a vector, which we use for printing. This function scans the list, one `Node` at a time, and prints the corresponding values. It is rather straightforward, but for the use of the `get` member function on the `next` data member (i.e., the call `next.get()`). This operation returns the vanilla pointer contained in `unique_ptr`, in a certain sense "unwrapping it".

> Why is `get` used by this code? Is it safe to do so and why?

The following test driver shows how to create an empty list. It then uses `list_insert_after` to insert ten numbers in the list. Note that we pass the sentinel node to the function: this has the effect of inserting these numbers at the beginning of the list. The code also shows how the list can be traversed by maintaining a pointer to the current node, and using `next` to update the latter to the following node.

File `list_driver.cpp`:

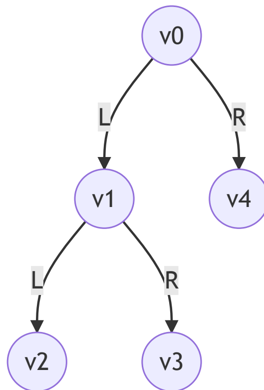
```
1  #include "list.hpp"
2  #include "utils.hpp"
3
4  int main(int argc, char **argv)
5  {
6      auto list = Node<float>{};
7
8      // Insert some numbers in the list
9      for (int i = 0; i < 10; ++i) {
10         list_insert_after(&list, static_cast<float>(i));
11     }
12
13     // Print the list
14     print(list_to_vector(list));
15
16     return 0;
17 }
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Chapter 4

Binary trees

Binary trees are ubiquitous and very useful data structures. A binary tree is similar to a linked list from the previous chapter, but each `Node` can have up to two successors, a *left child* and a *right child* (so the node is called the *parent* of its successors), as in the following diagram:



For this to be a valid tree, the diagram can never loop back: it is not possible for an ancestor of a node to be also one of its descendent.

We also need a mathematical definition of binary tree. A binary tree can be defined recursively as follows:

Definition 4.1 (Binary tree)**.** A **binary tree** is a *finite set* T such that:

1. $T = \{\}$ is empty or
2. $T = \{r\} \cup L \cup R$ is the union of three disjoint sets, where r is the *root* of the tree and the *left child* L and *right child* R are binary trees (sometimes called *subtrees*).

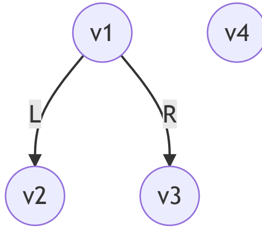
As for the linked list before, the elements v of a binary tree T are called *nodes*. Each node v is the root of a different subtree and identifies it.[1]

Using this definition, the binary tree in the diagram above is given by the expression:

$$\{v_0\} \cup (\{v_1\} \cup (\{v_2\} \cup \{\} \cup \{\}) \cup (\{v_3\} \cup \{\} \cup \{\})) \cup (\{v_4\} \cup \{\} \cup \{\})$$

To understand why, note that the diagram shows the tree $T = \{v_0\} \cup L \cup R$ where the left and right subtrees $L = \{v_1\} \cup L' \cup R'$ and $R = \{v_4\} \cup \{\} \cup \{\}$ have diagrams:

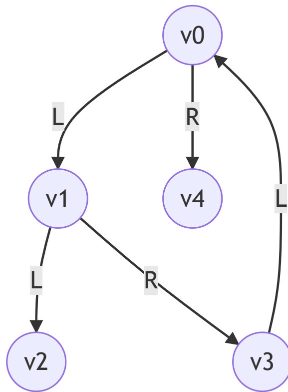
[1] Because nodes and subtrees are in one-to-one correspondence (with the exception of the empty subtree), the terms "node" and "subtree" are sometimes used interchangeably, but mathematically they are distinct.



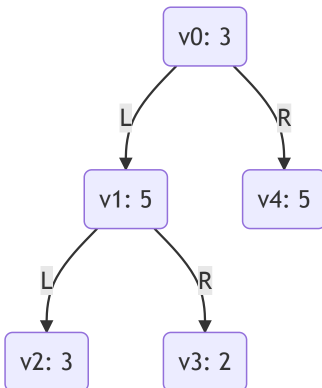
The right subtree $R = \{v_4\} \cup \{\} \cup \{\}$ is called a **leaf** because both of its children are empty. On the other hand, the L subtree further decomposes in two leaf trees $\{v_2\} \cup \{\} \cup \{\}$ and $\{v_3\} \cup \{\} \cup \{\}$, whose diagrams are:

The **depth** of a node is the number of steps required to reach it from the root. For instance, in the tree above, v_0 has depth 0, v_1 and v_4 have depth 1 and v_2 and v_3 have depth 2.

Because we have defined the tree as a set, each node can can appear only once in it (sets cannot contain repeated elements). Visually, the diagram of a tree cannot contain connections looping backwards. For instance, the following diagram does *not* represent a tree:



We often associate **values** to the nodes of a tree; for example, the following tree has integers associated to the nodes:



4.1 Size and height of a binary tree

The **size** of a binary tree T is the number of nodes, denoted as the caridnality $|T|$ of the set.

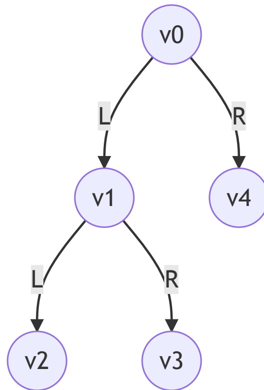
The *height* of a binary tree T is the maximum number of steps required to go from the root node to any of the leaf nodes. Formally:

Definition 4.2 (Height of a binary tree)**.** The *height* $h(T)$ of a binary tree T is given by

$$h(T) = \begin{cases} 1 + \max\{h(L), h(R)\}, & \text{if } T = \{r\} \cup L \cup R, \\ -1, & \text{if } T \text{ is empty.} \end{cases}$$

An empty tree $T = \{\}$ has thus height of -1. This is an arbitrary but useful value because, used in the formula above, assigns height 0 to a tree $T = \{r\}$ that contains a single node.

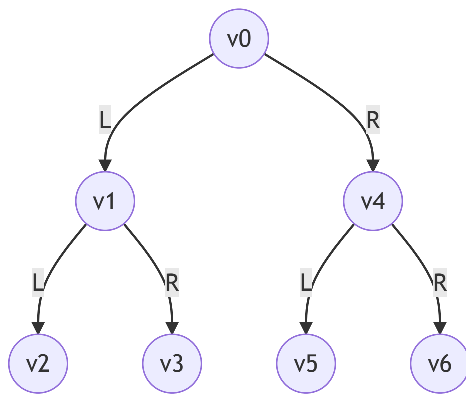
As an additional example, the following tree has size 5 and height 2:



If the tree is not empty, its height is also the same as the largest depth of any of its nodes.

Since many algorithms have a complexity proportional to the height of a tree, we are interested in packing as many nodes as possible for a given height. A binary tree is **perfect** if it contains the maximum number of nodes for its height.

For example, the tree in the diagram above is *not* perfect, but the following one is:



The following theorem establishes a few facts about perfect trees:

Theorem 4.1 (Maximum size of a binary tree)**.**

1. *The number of nodes in a binary tree T is at most $2^{h(T)+1} - 1$.*
2. *This bound is tight when the binary tree is perfect.*
3. *The binary tree T is perfect if, and only if, T is empty or if $T = \{r\} \cup L \cup R$ and the subtrees L and R are perfect and have equal height.*

Proof. This proof is optional. We can prove fact (1) by induction. For a tree of height -1 (empty tree), the formula returns 0 as maximum number of nodes, which is correct. Let T be a tree of height larger than -1;

then the tree is not empty, so it can be written as $T = \{r\} \cup L \cup R$. The height of subtrees L and R is at most $h(L) = h(R) = h(T) - 1$ (because of the definition of height of a tree). Hence, assuming inductively that the formula is correct for height less than $h(T)$, the maximum number of nodes in the tree T is upper-bounded by:

$$\begin{aligned} |T| &= |\{r\}| + |L| + |R| \\ &\leq 1 + 2^{h(L)+1} - 1 + 2^{h(R)+1} - 1 \\ &\leq 2 \cdot 2^{h(T)} - 1 = 2^{h(T)+1} - 1. \end{aligned}$$

This proves fact (1).

Fact (2) is due to the definition of perfect tree.

Fact (3) can be proven by showing that a tree constructed in this manner has indeed the maximum number of nodes, and that no other trees of equal height can have more nodes – this can also be shown by induction in a similar manner as (1). □

Below we introduce two different representations of binary trees. From the viewpoint of some algorithms operating on trees, they are largely equivalent. For such algorithms, the only requirement is that the following four operations can be implemented efficiently ($O(1)$ time complexity):

- left(T) is the left child of tree T.
- right(T) is the right child of tree T.
- empty(T) tells whether the tree T is empty or not.
- value(T) is the value (data) associated to the root of tree T.

4.2 Elementary representation

The conventional way of representing a binary tree T is to use an object N which is either:

- A structure with:
  - a left child object N.left,
  - a right child object N.right,
  - a value N.value;

  or
- The null object `NIL` (to represent an empty tree).

With this, we can simply define:

- left(N) = N.left
- right(N) = N.right
- empty(N) = $\delta_{\{N=\text{NIL}\}}$
- value(N) = N.value

4.2.1 C++ implementation of an elementary binary tree

We now give an example implementation in C++ of the elementary representation of a binary tree.

File `binary_tree.hpp`:

```
#ifndef __binary_tree__
#define __binary_tree__

#include <memory>
#include <utility>

// A class representing a binary tree
template <typename V> struct BinaryTree {
```

```
9	    V _value;
10	    std::unique_ptr<BinaryTree<V>> _left;
11	    std::unique_ptr<BinaryTree<V>> _right;
12	
13	    // These are global functions because of `friend`
14	    friend V &value(BinaryTree *t) { return t->_value; }
15	    friend const V &value(const BinaryTree *t) { return t->_value; }
16	    friend BinaryTree *left(BinaryTree *t) { return t->_left.get(); }
17	    friend BinaryTree *right(BinaryTree *t)
18	    {
19	        return t->_right.get();
20	    }
21	};
22	
23	// A helper function to build a binary tree
24	template <typename V>
25	std::unique_ptr<BinaryTree<V>>
26	make_binary_tree(const V &value, std::unique_ptr<BinaryTree<V>> left,
27	                 std::unique_ptr<BinaryTree<V>> right)
28	{
29	    return std::unique_ptr<BinaryTree<V>>{
30	        new BinaryTree<V>{value, std::move(left), std::move(right)}};
31	}
32	
33	#endif // __binary_tree__
```

There are a few points of note in this particular implementation. First, just like the linked list before, a tree owns its subtrees via `unique_ptr`. This means that, when a tree object instance is deleted, so are all the subtrees automatically.

The functions `left` and `right` are defined as friends of `BinaryTree` for notational convenience (`friend` has no effect since `BinaryTree` has no private members). Specifically, `friend` is used here to define non-member functions directly in the class body.

`left` and `right` return a vanilla pointer to the left and right trees (using the `.get()` member function of `unique_ptr`). This means that a tree is represented by a *pointer* to a `BinaryTree<V>` structure, not by a *value* of type `BinaryTree<V>`. This is a subtle but important difference as pointers can take value `nullptr`, which we use here to denote an empty tree. Note in particular that we did not implement the `empty` function. This is implicitly given by testing whether the pointer is `nullptr` or not. Because in C++ a pointer implicitly converts to a boolean `true` or `false` depending on whether it is different from `nullptr` or not, in code we can simply use the syntax `if (tree) { /* empty */}` or `if (!tree) { /* not empty */ }` to act based on whether a tree is empty or not.

Note also a tree is represented by a *vanilla* pointer, not by a `unique_ptr`. This is because we do no wish to transfer ownership of the subtrees to a function that calls `left` or `right` merely for the purpose of visiting the tree. Ownership is instead kept by the part of the program that manages the lifetime of the tree structure.

Finally, `make_binary_tree` is a convenience function that makes the creation of new trees slighter more compact.

In the code distribution, we also provide a file `binary_tree_print.hpp` defining the function `print_binary_tree` to print a binary tree on screen diagrammatically.

4.3 Traversing a tree: depth and breadth first

Traversing a tree means visiting each of its nodes starting from the root. There are two main types of traversals: depth first and breadth first.

A **depth-first traversal** (DFT) descends the tree from the root to the left-most leaf and then gradually backtracks and descends the other branches. This is very easy to implement with a recursive algorithm:

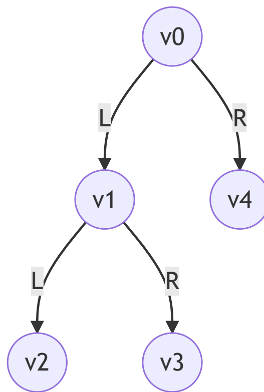
DFTraversal(T) :

Precondition: T is a binary tree.

1. If empty(T), return.
2. Optionally, process value(T). // Pre-order visit
3. Call DFTraversal(left(T)).
4. Optionally, process value(T). // In-order visit
5. Call DFTraversal(right(T)).
6. Optionally, process value(T). // Post-order visit

When a subtree T is visited during a traversal, one usually performs an operation on its value (such as printing the value on screen). This operation does not necessarily occur immediately when the subtree is visited; instead, it can be performed before the recursive calls (pre-oder), in between (in-order) or after (post-order).

For example, `DFTraversal` called on the following tree:



will *visit* the nodes in order v_0, v_1, v_2, v_3, v_4 but will *process* their values as follows:

- pre-order depth-first processing order: v_0, v_1, v_2, v_3, v_4 (same as the visit order);
- in-order depth-first processing order: v_2, v_1, v_3, v_0, v_4;
- post-order depth-first processing order: v_2, v_3, v_1, v_4, v_0.

Alternatively, a **breadth-first traversal** (BFT) visits first all the nodes at depth 0, then all the nodes at depth 1 and so on. This is done with the help of a queue Q, pushing the child of each visited node and poppin git later to vist its children:

BFTraversal(Q) :

Precondition: The queue $Q = (T)$ contains the tree as sole element.

1. While empty(Q) is false, repeat:
    1. Let $T \leftarrow$ Dequeue(Q).
    2. If empty(T), continue with the next iteration of loop 1.
    3. Optionally, process value(T).
    4. Call Enqueue(Q, left(T)).
    5. Call Enqueue(Q, right(T)).

Note that, in the pseudo-code, the function `BFTraversal` should be called with a queue Q that initially contains the root of the tree.

For example, `BFTraversal` applied to the tree above will visit and process the values of nodes in order v_0, v_1, v_4, v_2 and v_3.

Traversing a tree is sometimes called "searching" the tree, so the two algorithms above are also called depth-first search (DFS) and breadth-first search (BFS). We prefer not to use this terminology to avoid confusion with algorithms that actually search trees for specific elements.

We can implement these two functions, as well as a function to compute the height of a tree, as follows.

File `binary_tree_traversal.hpp`:

```
#ifndef __binary_tree_traversal__
#define __binary_tree_traversal__

#include <queue>

// Compute the height of a tree
template <typename T> int height(const T &tree)
{
    if (empty(tree)) return -1;
    return 1 + std::max(height(left(tree)), height(right(tree)));
}

// Depth first traversal (in order)
template <class T, typename F>
void df_traversal(const T &tree, F action)
{
    if (!tree) return;
    df_traversal(left(tree), action);
    action(tree);
    df_traversal(right(tree), action);
}

// Breadth first traversal
template <class T, typename F>
void bf_traversal(const T &tree, F action)
{
    auto queue = std::queue<T>{};
    queue.push(tree);
    while (!queue.empty()) {
        auto &current = queue.front();
        if (current) {
            action(current);
            queue.push(left(current));  // enqueue
            queue.push(right(current)); // enqueue
        }
        queue.pop(); // dequeue
    }
}

#endif // __binary_tree_traversal__
```

The functions take as input an object `action`. This is a callable object (e.g., another function) which is executed for each subtree. For example, this can be used to print the value associated to each subtree in the tree, as shown in the following test driver:

```cpp
#include "binary_tree.hpp"
#include "binary_tree_traversal.hpp"
#include "binary_tree_print.hpp"

int main(int argc, char** argv)
{
    auto bt = make_binary_tree(1.0f,
      make_binary_tree(2.0f,
        make_binary_tree(4.0f, {}, {}),
        make_binary_tree(5.0f, {},
          make_binary_tree(8.0f, {}, {})
        )
      ),
      make_binary_tree(3.0f,
        make_binary_tree(6.0f, {}, {}),
        make_binary_tree(7.0f, {}, {})
      ));

    std::cout << "Tree:\n";
    print_binary_tree(bt.get());

    auto action = [](const auto& tree) {
        std::cout << "Visited subtree: " << value(tree) << '\n';
    };

    std::cout << "\nDepth-first traversal (DFT)\n";
    df_traversal(bt.get(), action);

    std::cout << "\nBreadth-first traversal (BFT)\n";
    bf_traversal(bt.get(), action);

    return 0;
}
```

```
Tree:
1 -------v
2 -v     3 -v
4  5 -v  6  7
      8

Depth-first traversal (DFT)
Visited subtree: 4
Visited subtree: 2
Visited subtree: 5
Visited subtree: 8
Visited subtree: 1
Visited subtree: 6
Visited subtree: 3
Visited subtree: 7

Breadth-first traversal (BFT)
Visited subtree: 1
Visited subtree: 2
Visited subtree: 3
```

```
Visited subtree: 4
Visited subtree: 5
Visited subtree: 6
Visited subtree: 7
Visited subtree: 8
```

Note that `print_binary_tree` is an utility function to print a tree graphically. It is provided in the header `binary_tree_print.hpp`.

4.4 Binary search trees

An important application of trees is to store elements so that they can be quickly searched by value. This requires a special kind of tree called a binary search tree:

Binary search tree (definition)

A binary tree T is a *binary search tree* (BST) if, and only if, T is empty or $T = \{r\} \cup L \cup R$ and

- for all subtrees $S \subset L$ one has $\text{value}(S) \leq \text{value}(T)$ and
- for all subtrees $S \subset R$ one has $\text{value}(S) > \text{value}(T)$ and
- L and R are also BSTs.

Note that the definition implies not only that $\text{value}(L) \leq \text{value}(T)$ for the left child, but that this is also true for *all* left descendant subtrees $S \subset L$ (and a symmetric condition applies to right descendants).

4.4.1 Searching a BST

Searching for an element x means returning a subtree that has x as the value of its root, if any can be found. We can in fact do something slightly more general and return the maximal subtree whose value does not exceed x. In pseudo-code, this is done as follows:

$\text{BSTSearch}(T, x)$:

- **Precondition:** T is a BST.
- **Postcondition:** Returns the subtree $S \subset T$ with the largest value not greater than x. If no such subtree exists, returns the empty tree.

1. If $\text{empty}(T)$ or $\text{value}(T) = x$, then return T.
2. Let $T = \{r\} \cup L \cup R$.
3. If $x < \text{value}(T)$, then return $\text{BSTSearch}(L, x)$.
4. Otherwise:
   1. Let $S \leftarrow \text{BSTSearch}(R, x)$.
   2. If S is empty, return T.
   3. Otherwise, return S.

The algorithm works as follows:

1. Line 1 checks if the current tree T is empty, in which case it gives up, or if the value of T is x, in which case it stops and returns T as the solution.

2. Otherwise, line 3 checks if the value of $\text{value}(T)$ is greater than x. In this case, T cannot be the answer as we are looking for a tree whose value is *not* greater than x. The same is true for all right subtrees, as their values are even larger than T. Hence, the algorithm calls `BSTSearch` recursively to the left in search for the subtree that satisfies the required property. Note that. if the recursive function fails to find a suitable tree, then there is none.

3. Otherwise, $\text{value}(T)$ is (strictly) *less* than x ($\text{value}(T) < x$) and line 4 is executed. Since the value of T is less than x, it is also not greater than x, so T *could* be the solution to the problem according to the definition. However, T may not be maximal among the trees whose value is not greater than x. Hence,

the algorithm calls `BSTSearch` recursively to the right to check if a better subtree S can be found. If such a tree is found, then S is the solution to the problem because $\text{value}(T) < \text{value}(S) \leq x$. Otherwise, T is the solution, because $\text{value}(T) < x$ as we have already established, and there is no better subtree that satisfies the property.

4.4.2 Building a BST

We can easily build a BST from scratch by inserting one value at a time: if the tree is empty, we just create a singleton tree with that value. Otherwise, we insert the value into the left or right subtrees, depending on whether it is smaller or greater than the value of the root, in a recursive fashion.

Because we need to modify the tree in order to insert elements into it, it is not enough to use the operations `left`, `right`, `value`, etc. we have introduced above because they do not allow to *modify* the tree. To apply such modifications, we make use of the specific elementary tree representation N introduced above.

With this, inserting an element in the tree is obtained via the algorithm:

$\text{BSTInsert}(N, x)$:

- **Precondition:** N is the elementary representation of a BST.
- **Postcondition:** Returns a BST that contains the same nodes as N plus a new node whose value is x.

1. If N is `NIL` then return $\{x, \text{NIL}, \text{NIL}\}$.
2. If $x \leq N.\text{value}$ then:
    1. Set $N.\text{left} \leftarrow \text{BSTInsert}(N.\text{left}, x)$
3. Otherwise:
    1. Set $N.\text{right} \leftarrow \text{BSTInsert}(N.\text{right}, x)$
4. Return N.

4.4.3 C++ implementation of a BST

The following example implementation demonstrates building and searching a BST in C++.

File `binary_search_tree.hpp`:

```
#ifndef __binary_search_tree__
#define __binary_search_tree__

#include "binary_tree.hpp"

template <class T, typename V> T bst_search(const T &tree, const V &v)
{
    if (!tree || v == value(tree)) return tree;
    if (v < value(tree)) return bst_search(left(tree), v);
    auto other = bst_search(right(tree), v);
    return other ? other : tree;
}

template <typename V>
std::unique_ptr<BinaryTree<V>>
bst_insert(std::unique_ptr<BinaryTree<V>> tree, const V &v)
{
    if (!tree) return make_binary_tree(v, {}, {});
    if (v <= value(tree.get())) {
        tree->_left = bst_insert(std::move(tree->_left), v);
    } else {
        tree->_right = bst_insert(std::move(tree->_right), v);
```

```
23      }
24      return tree;
25  }
26
27  #endif // __binary_search_tree__
```

Note that `bst_search` only use the basic interface to a binary tree (i.e., the functions `left`, `right`, and `value`. It therefore works for BST backed by any binary tree type (for example, we introduce below a `CompleteBT` class that is also be compatible with this interface). On the other hand, `bst_insert` assumes that the tree is of type `BinaryTree`.

File `binary_search_tree_driver.hpp`:

```
1   #include "binary_search_tree.hpp"
2   #include "binary_tree_print.hpp"
3
4   #include <iostream>
5
6   int main(int argc, char **argv)
7   {
8       std::unique_ptr<BinaryTree<int>> bt;
9
10      for (int x : {12, 5, 18, 2, 9, 15, 19, 13, 17}) {
11          bt = bst_insert(std::move(bt), x);
12          std::cout << "Tree after inserting " << x << ":\n";
13          print_binary_tree(bt.get());
14          std::cout << "\n";
15      }
16
17      for (int x : {0, 5, 6, 18, 19, 20}) {
18          BinaryTree<int> *result = bst_search(bt.get(), x);
19          std::cout << "The largest element not exceeding " << x
20                    << " is ";
21          std::cout << (result ? std::to_string(value(result))
22                               : "none");
23          std::cout << "\n";
24      }
25
26      return 0;
27  }
```

```
Tree after inserting 12:
12

Tree after inserting 5:
12
5

Tree after inserting 18:
12 -v
5   18

Tree after inserting 2:
12 -v
5   18
```

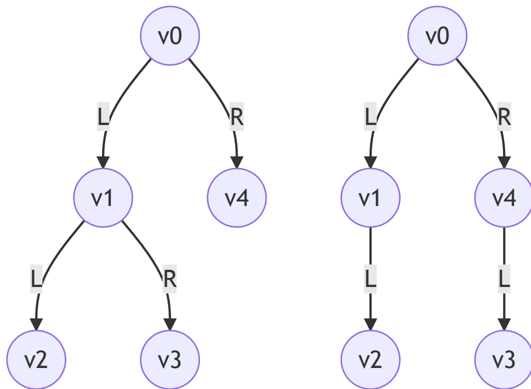
```
2

Tree after inserting 9:
12 ---v
5 -v  18
2  9

Tree after inserting 15:
12 ---v
5 -v  18
2  9  15

Tree after inserting 19:
12 ---v
5 -v  18 -v
2  9  15  19

Tree after inserting 13:
12 ---v
5 -v  18 -v
2  9  15  19
      13

Tree after inserting 17:
12 ---v
5 -v  18 -----v
2  9  15 -v   19
      13  17

The largest element not exceeding 0 is none
The largest element not exceeding 5 is 5
The largest element not exceeding 6 is 5
The largest element not exceeding 18 is 18
The largest element not exceeding 19 is 19
The largest element not exceeding 20 is 19
```

4.5 An alternative representation of complete trees

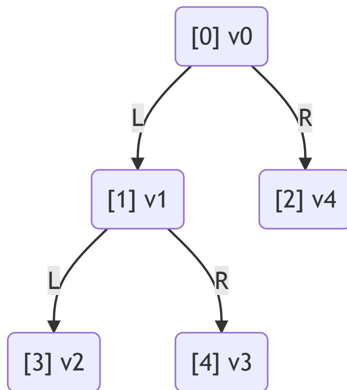
We now give a different representation that is only applicable to a subset of all binary trees known as "complete trees".

Definition 4.3 (Complete binary tree)**.** A binary tree T is **complete** if, and only if, all the levels in the tree are full except the last one, which is partially filled from left to right.

For example, the first of these two trees is complete, but the second one is not:



The nodes in a complete binary tree can be enumerated from left to right and from top to bottom, in the same manner as a breadth first traversal. For example, we can enumerate the nodes in the complete binary tree above as follows:



We can then represent the tree by array A of n elements, where n is the total number of nodes in the tree. We do so by associating the i-th node in the tree with element A_i in the array and storing in the array the node's value. For example, the complete binary tree above maps to the array

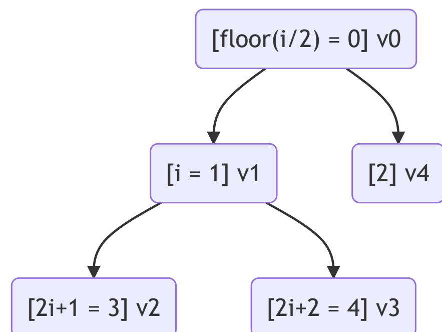
$$A = [\text{value}(v_0), \text{value}(v_1), \text{value}(v_4), \text{value}(v_2), \text{value}(v_3)].$$

This representation is very space efficient because it does not require storing "metadata" such as pointers to nodes: only the actual node values are stored in the array. Furthermore, the usual operations are also very efficient. Specifically, given the index i of a node, we have:

- $\text{left}(i) = 2i + 1$
- $\text{right}(i) = 2i + 2$
- $\text{parent}(i) = \lfloor (i-1)/2 \rfloor$
- $\text{empty}(i) = \delta_{\{i \geq |A|\}}$
- $\text{value}(i) = A_i$

Note that we also defined a function parent which computes the index of the parent of a given subtree. This comes "for free" with this representation, but would require some modifications and increased space utilization to implement in the elementary binary tree representation discussed before.

The following example shows how the formulas apply to node v_1 in the tree above:



Proof. This proof is optional. The correctness of the formulas above is not immediately obvious. We now sketch how the expression $\text{left}(i)$ can be proved. By induction, if $i = 0$, then the index of the left child is $1 = 2 \cdot 0 + 1 = \text{left}(0)$. Furthermore, suppose that the left child of node $i > 0$ is $\text{left}(i) = 2i + 1$. The following node $2i + 2$ is thus the the right child of i, and the node after that $2i + 3 = 2(i + 1) + 1 = \text{left}(i + 1)$ is the left child of node $i + 1$. *Q.E.D.* □

4.5.1 C++ implementation of a complete binary tree

Next, we show an implementation of a complete binary tree in C++. This implementation "wraps" a given `vector`. Wrappers are classes that provide a view of a piece of data (in this case a vector) as if it was of a different type (in this case a complete binary tree).

File `binary_tree_complete.hpp`:

```
#ifndef __binary_tree_complete__
#define __binary_tree_complete__

#include <cstddef>
#include <limits>
#include <vector>

// Wraps a vector in a complete binary tree
template <typename V> struct CompleteBT {
    std::vector<V> *_storage;
    std::size_t _root;
    std::size_t _size;

    CompleteBT<V>(std::vector<V> *storage)
        : _storage{storage}, _root{0}, _size{storage->size()}
    {
    }

    CompleteBT<V>(std::vector<V> *storage, std::size_t root,
                  std::size_t size)
        : _storage{storage}, _root{root}, _size{size}
    {
    }

    CompleteBT<V> subtree(std::size_t root) const
    {
        return CompleteBT<V>{_storage, root, _size};
    }

    friend V &value(CompleteBT &t) { return (*t._storage)[t._root]; }
```

```
    friend const V &value(const CompleteBT &t)
    {
        return (*t._storage)[t._root];
    }
    friend CompleteBT parent(const CompleteBT &t)
    {
        if (t._root == 0)
            return t.subtree(std::numeric_limits<std::size_t>::max());
        return t.subtree((t._root - 1) / 2);
    }
    friend CompleteBT left(const CompleteBT &t)
    {
        return t.subtree(2 * t._root + 1);
    }
    friend CompleteBT right(const CompleteBT &t)
    {
        return t.subtree(2 * t._root + 2);
    }

    explicit operator bool() const
    {
        return _root < _size;
    } // not empty
};

#endif // __binary_tree_complete__
```

The functions `left`, `right` and `parent` return another instance of `CompleteBT`. Note that, differently from `BinaryTree`, they *do not* return a pointer to a tree object, but a `CompleteBT` object by value. `CompleteBT` internally contains a pointer to the vector backing the tree — in a certain sense, `CompleteBT` is just a "fancy pointer".

`operator bool` allows a syntax such as `if (tree) { ... }` to work by testing if the `CompleteBT` tree is empty or not. For `BinaryTree*` this is implicitly obtained by testing whether the pointer is null or not.

In a full implementation, we may need to also implement `operator*` (dereferencing), but we do not use it in our examples.

Note that we have also implemented the function `parent`, which was not available for `BinaryTree` (it is an exercise to add `parent()` to `BinaryTree`).

Because of the common interface implemented by functions `left`, `right`, etc., we can use the same generic algorithms we have shown before, as in the following example.

File `binary_tree_complete_driver.cpp`:

```
#include "binary_tree_complete.hpp"
#include "binary_tree_print.hpp"
#include "binary_tree_traversal.hpp"
#include <utils.hpp>

#include <vector>

int main(int argc, char **argv)
{
    auto array = std::vector<float>{1, 2, 3, 4, 5, 6, 7, 8};
    auto bt = CompleteBT<float>{&array};
```

```
12
13      std::cout << "Tree:\n";
14      print_binary_tree(bt);
15
16      auto action = [](const auto &tree) {
17          std::cout << "Visited subtree: " << value(tree) << '\n';
18      };
19
20      std::cout << "\nDepth-first traversal (DFT)\n";
21      df_traversal(bt, action);
22
23      std::cout << "\nBreadth-first traversal (BFT)\n";
24      bf_traversal(bt, action);
25
26      return 0;
27  }
```

```
Tree:
1 ----v
2 -v  3 -v
4  5  6  7
8

Depth-first traversal (DFT)
Visited subtree: 8
Visited subtree: 4
Visited subtree: 2
Visited subtree: 5
Visited subtree: 1
Visited subtree: 6
Visited subtree: 3
Visited subtree: 7

Breadth-first traversal (BFT)
Visited subtree: 1
Visited subtree: 2
Visited subtree: 3
Visited subtree: 4
Visited subtree: 5
Visited subtree: 6
Visited subtree: 7
Visited subtree: 8
```

Chapter 5

Heaps

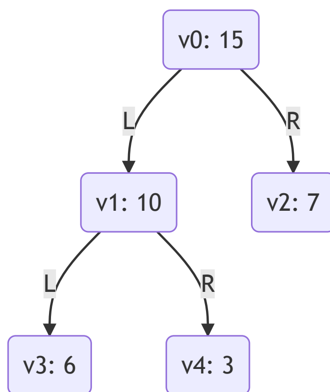
Heaps are efficient data structures used to implement sorting algorithms, priority queues, and other useful tools.

A *heap* is a binary tree with the following property:

Definition 5.1 (Heap property)**.** A binary tree T has the *heap property* if, and only if, for any subtree $S \subset T$:

$$\text{value}(S) \leq \text{value}(\text{parent}(S))$$

Here's an example of a heap with five nodes:

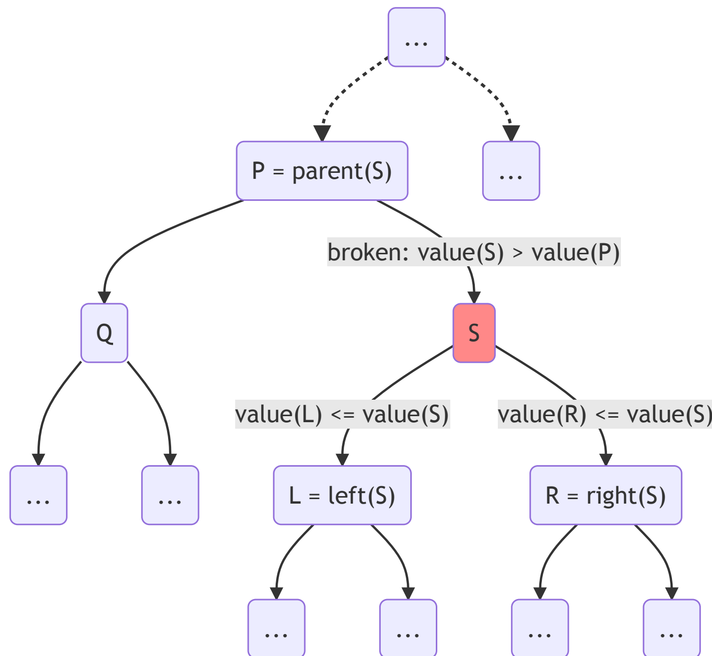


Note that the value of the root tree T mus be the maximum overall. This is also called a *max heap* to distinguish it from a *min heap*, defined in the same way but with symmetric inequalities.

5.1 Restoring the heap property

Often we are given a binary tree T where the heap property is satisfied everywhere *except* for the value associated to a specific subtree $S \subset T$. This means that we can turn T into a proper heap by changing value(S). In such a case, the heap property can also be restored by rearranging the values already stored in the tree instead of changing them.

Consider first the case in which the value(S) of the subtree S is "too large", as in the following example:



Assume as precondition that the heap property can be restored by reducing $y = \text{value}(S)$ to a certain value $x \leq y$. The heap property contains the following inequalities involving S:

$$\text{value}(D) \leq \text{value}(S) \leq \text{value}(A)$$

for all descendants $D : D \subset S$ and ancestors $A : S \subset A$ of the subtree S. The precondition means that there is a value x such that:

$$\text{value}(D) \leq x \leq \text{value}(A).$$

We can in particular choose $x = \text{value}(P)$ where P is the parent of S because, also due to the heap property, $\text{value}(P) \leq \text{value}(A)$ for all ancestors A of S.

We conclude that, after replacing $\text{value}(S)$ with $x = \text{value}(P)$, the heap property is restored. Unfortunately, this also means that the original value y of S is overwritten. To keep y, we further increase $\text{value}(P)$ to be y. The net effect is to *swap* x and y, thus keeping all the original values in the tree. However, increasing $\text{value}(P)$ can break the heap property again, this time at P. The key advantage is that the defect is either eliminated or moved one step closer to the root of the tree. We can then repeat the process recursively until the defect is eventually eliminated (this certainly occurs when $S = T$ is the root as there are no more parents).

This idea is implemented by the `SiftUp` algorithm:

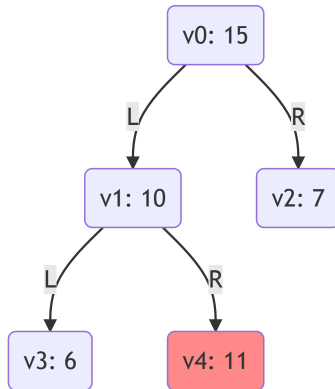
SiftUp(S)**:**

- **Precondition:** S is a subtree of a binary tree T which is a heap up to *reducing* $\text{value}(S)$.
- **Postcondition:** T is a heap with a permutation of the input values.

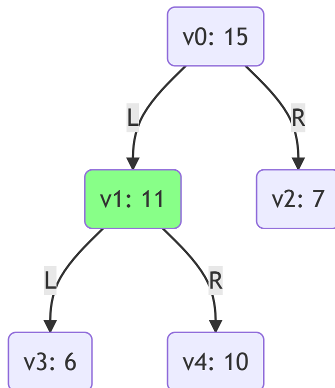
1. If $\text{empty}(\text{parent}(S))$ stop.
2. if $\text{value}(\text{parent}(S)) \geq \text{value}(S)$ stop.
3. Swap the values of S and $\text{parent}(S)$.
4. Call recursively $\text{SiftUp}(\text{parent}(S))$.

Line 1 checks if S is already the root tree, in which case there is nothing to do: the binary tree is already a heap. Line 2 checks if the value of the parent P is already larger than S. In this case, there is no need to reduce S because the tree is already a heap. Line 3 swaps the values of S and P, as explained. Finally, line 4 calls `SiftUp` recursively on the parent P.

For example, consider the following binary tree:



The tree has the heap property except for v_4, which is larger than its parent. Hence, `SiftUp` swaps the values of v_1 and v_4, producing the tree



Note that the value of v_1 has increased from 10 to 11 due to this exchange. This could in principle break the heap property for this node, so `SiftUp` checks the parent v_0. In this case the parent has value 15 which is larger, so the algorithm stops.

The next algorithm does the opposite: if a value in the tree is too small, it sifts it down towards the leaves until the heap property is restored:

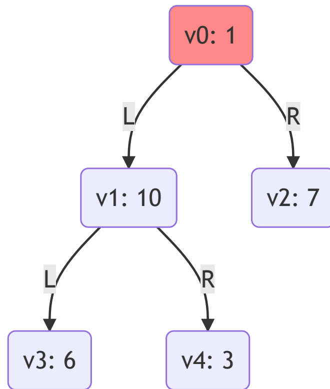
SiftDown(S)**:**

- **Precondition:** S is a subtree of a binary tree T which is a heap up to *increasing* value(S).
- **Postcondition:** T is a heap with the same values as the input tree up to permutation.

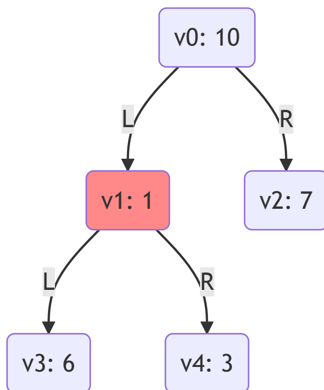
1. Let $C \leftarrow$ left(S).
2. Let $O \leftarrow$ right(S).
3. If empty(C) or if not empty(O) and value(O) $\geq$ value(C):
    1. Set $C \leftarrow O$.
4. If empty(C), stop *// both children empty*
5. If value(S) $\geq$ value(C), stop.
6. Swap the values of S and C.
7. Call recursively SiftDown(C).

The algorithm works like SiftUp, but in reverse. The only point of note are lines 1-5: here the algorithm checks which ones of the two children of S is the largest, and chooses that for a potential value swap with S. The code is slightly complex due to the fact that none, one or both of the children can be empty.

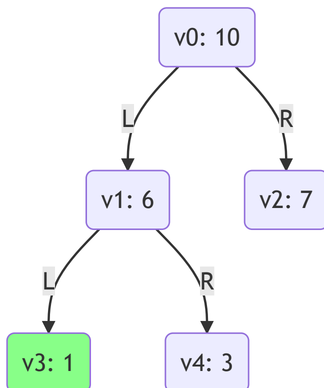
For example, in the following binary tree node v_0 has a value smaller than both children v_1 and v_2:



In this case, `SiftDown` swaps v_0 with v_1 because v_1 is the largest of the two children, resulting in the modified tree:



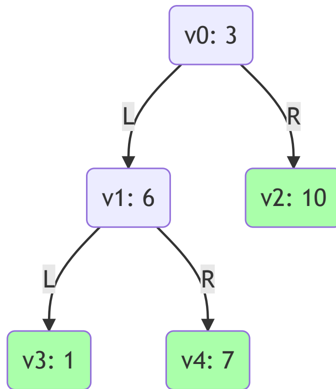
However, now the value of v_1 has been reduced from 10 to 1, so `SiftDown` checks this node. It is indeed smaller than both children, so the operation is repeated, swapping the values of v_1 and v_3 (the largest children):



Now the heap property is fully restored.

5.2 Building a heap

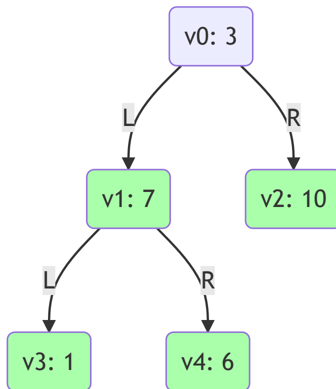
Next, we describe an algorithm for building a heap given an initial set of values. The idea is to start by arranging the values arbitrarily in a binary tree. For example, given values $A = [3, 6, 10, 1, 7]$, we can arrange them in the tree



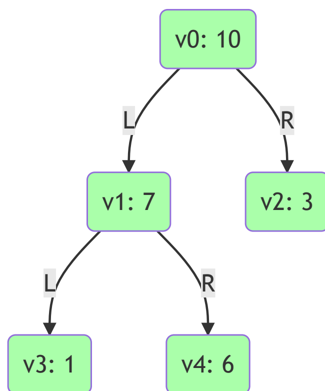
The leaves ($\{v_3\}$, $\{v_4\}$ and $\{v_2\}$) already form valid heaps of a single element (so they are coloured in green). Then, one considers intermediate subtrees S such that their left and right children are already heaps and calls SiftDown(S) to restore the heap property for the entire subtree.

Remark. The precondition for calling `SiftDown` is satisfied by such trees S. Can you see why?

In the example, one calls SiftDown first on the tree rooted at v_1, which results in the structure:



Then one calls SiftDown on the tree rooted at v_0 (the whole tree), obtaining:



at which point the entire binary tree is a heap.

A particularly elegant and efficient version of this algorithm is obtained by using the complete binary tree representation introduced in the previous chapter. Recall that the array A can be interpreted as the representation of a complete binary tree.

To get the algorithm started, we need to find out which elements in the array forms the leaves of the tree. Recall that the left child of node i has index $2i + 1$. If the array has $n = |A|$ elements, then the largest

possible index i that still has a left child must satisfy inequality $2i+1 \leq n-1$, or $i = \lfloor n/2 \rfloor - 1$. In the example above, $n = 5$, so the latest non-leaf node is $i = 1$, meaning that $i = 2, 3, 4$ are all leaf nodes.

With this information, one simply calls `SiftDown` on the trees rooted at $i, i-1, \ldots, 0$ in this order to build the heap:

BuildHeap(A):

- **Precondition:** An array A.
- **Postcondition:** An array A that, interpreted as a complete binary tree, has the heap property.

1. For $i = \lfloor |A|/2 \rfloor - 1, \ldots, 0$:
   1. Interpret the subarray $(A_i, \ldots, A_{|A|-1})$ as a complete binary tree S.
   2. Call SiftDown(S).

5.3 HeapSort

`BuildHeap` can be used to sort an array by removing elements from the root of the heap. Since the largest element is found at the root, these extracted element are added to the sorted arrow backward, from the end to the beginning. This is done very efficiently by swapping element A_0 with element A_i and then calling `SiftDown` on the binary tree given by the subarray $(A_0, \ldots, A_{i-1})$. This idea is codified by the `HeapSort` algorithm:

HeapSort(A):

Precondition: An array A.

Postcondition: The array A has the same elements as before, but permuted in non-decreasing order.

1. Call BuildHeap(A)
2. For $i = |A| - 1, \ldots, 1$:
   1. Swap elements A_0 and A_i.
   2. Interpret the subarray $(A_0, \ldots, A_{i-1})$ as a complete binary tree T.
   3. Call SiftDown(T).

5.3.1 Complexity analysis

Next, we analyze the worst case complexity of these algorithms. `SiftUp` and `SiftDown` make work proportional to the height h of the input tree, i.e. $\Theta(h)$. If the input tree is complete, then its height is proportional to the log of the number of nodes n, so the complexity is $\Theta(\log n)$.

`BuildHeap` calls `SiftDown` on every subtree; since there are $\lfloor n/2 \rfloor$ of those and each has at most n elements, the complexity is bounded by $O(n \log n)$. However, this complexity is actually a *pessimistic* estimate. This is because most of the intermediate trees are short. Specifically, consider a full tree of height h (thus with $n = 2^{h+1} - 1$ nodes). There are 2^h subtrees of height 0, 2^{h-1} of height 1, and so on, so the total cost of `BuildHeap` is:

$$0 \cdot 2^h + 1 \cdot 2^{h-1} + \cdots + h \cdot 1 = \sum_{i=0}^{h} i 2^{h-i} = 2^{h+1} - h - 2$$

The last formula can be derived using Z-transform tricks, or simply verified by induction. For the latter, the formula gives the correct result 2 for $h = 1$; furthermore, if it is correct for h, it is also correct for $h + 1$ because:

$$\sum_{i=0}^{h+1} i 2^{h+1-i} = 2 \sum_{i=0}^{h} i 2^{h-i} + h + 1 = 2(2^{h+1} - h - 2) + h + 1 = 2^{h+2} - (h+1) - 2.$$

Because h is in the order of $\log n$, the complexity of `BuildHeap` is in the order of $2n - \log_2 n - 2$, i.e., $O(n)$, which is better than the previous estimate $O(n \log n)$.

The cost of `HeapSort` is given by the cost of `BuildHeap` $O(n)$ followed by the cost of extracting one element from the heap at a time. Each time this is done, `SiftDown` is called from the root of the tree. Starting from a full binary tree of height h, it means that extracting the first 2^h elements incurs cost h, the following 2^{h-1} elements cost $h-1$, etc, so the cost is:

$$h \cdot 2^h + (h-1) \cdot 2^{h-1} + \cdots + 2 = \sum_{i=1}^{h} i2^i = 2(h-1)2^h + 2$$

Hence the cost of `HeapSort` is $\Theta(n \log n)$. This is unsurprising given that we already know that this is the lower bound for this class of sorting algorithms.

5.3.2 C++ implementation of heaps

The following C++ implementation demonstrates these concepts.

File `heap.hpp`:

```
#ifndef __heap__
#define __heap__

#include <cassert>
#include <cstdint>
#include <functional>
#include <utility>
#include <vector>

#include "binary_tree_complete.hpp"

template <typename V> using default_comparison_t = std::greater<V>;

template <typename T, typename C> void heap_sift_up(T &&tree, C compare)
{
    auto p = parent(tree);
    if (!p) return;
    if (compare(value(tree), value(p))) std::swap(value(tree), value(p));
    heap_sift_up(p, compare);
}

template <typename T, typename C> void heap_sift_down(T &&tree, C compare)
{
    auto child = left(tree);
    auto other = right(tree);
    if (!child || (other && compare(value(other), value(child)))) {
        child = other;
    }
    if (!child) return;
    if (compare(value(child), value(tree))) {
        std::swap(value(child), value(tree));
    }
    heap_sift_down(child, compare);
}

template <typename V, typename C = default_comparison_t<V>>
```

```
37  void build_heap(std::vector<V> &storage, C compare = C{})
38  {
39      for (std::intmax_t i = (signed)storage.size() / 2 - 1; i >= 0; --i) {
40          heap_sift_down(CompleteBT<V>{&storage, (size_t)i, storage.size()},
41                         compare);
42      }
43  }
44
45  template <typename V, typename C = default_comparison_t<V>>
46  void heap_sort(std::vector<V> &storage, C compare = C{})
47  {
48      build_heap(storage, compare);
49      for (size_t back = storage.size() - 1; back >= 1; --back) {
50          std::swap(storage[0], storage[back]);
51          heap_sift_down(CompleteBT<V>{&storage, 0, back}, compare);
52      }
53  }
54
55  #endif /* __heap__ */
```

The code closely matches the pseudo-code we have seen above. The various function take a *comparison* operator `compare` as input, which defaults to an instance of `std::greater<V>`. This operator is used to compare values in the heap. With the default choice, the functions implement a max heap. Other choices are possible, for instance to implement a min heap or to make the heap work for custom objects of a type `V` for which the default choice `std::greater<V>` is undefined.

The implementation of `heap_sift_up` and `heap_sift_down` are generic, and work for complete or incomplete binary trees. As for the other binary tree algorithms, the only requirement is that operations `parent`, `left`, `right`, `value` and `empty` are defined. In practice, `heap_sift_down` can be slightly optimized if one knows that the tree is complete (which is a very common case for a heap).

`build_heap` and `heap_sort` are more specific: they use the `CompleteBT` class to wrap an input array and interpret is as a complete binary tree.

The following driver tests the code above.

File `heap_driver.hpp`:

```
1   #include "binary_tree.hpp"
2   #include "binary_tree_print.hpp"
3   #include "heap.hpp"
4   #include <utils.hpp>
5
6   #include <iostream>
7   #include <vector>
8
9   int main(int argc, const char *argv[])
10  {
11      auto array =
12          std::vector<float>{1,  19, 2,  9,  12, 18, 4, 8, 5,  6,
13                             17, 10, 11, 14, 16, 15, 7, 3, 13, 20};
14
15      std::cout << "Before building the heap: ";
16      print(array);
17      print_binary_tree(CompleteBT<float>{&array});
18
```

```
19      build_heap(array);
20
21      std::cout << "\nAfter building the heap: ";
22      print(array);
23      print_binary_tree(CompleteBT<float>{&array});
24
25      heap_sort(array);
26      print(array, "\nArray after heapsort: ");
27
28      return 0;
29  }
```

```
Before building the heap: [1, 19, 2, 9, 12, 18, 4, 8, 5, 6, 17, 10, 11, 14, 16, 15, 7, 3, 13, 20]
1 --------------------v
19 -----------v       2 ------v
9 -----v      12 -v   18 -v   4 --v
8 --v  5 -v   6   17  10  11  14  16
15  7  3  13  20

After building the heap: [20, 19, 18, 15, 17, 11, 16, 9, 13, 12, 1, 10, 2, 14, 4, 8, 7, 3, 5, 6]
20 -----------------v
19 ----------v      18 ----v
15 ---v      17 -v  11 -v  16 -v
9 -v  13 -v  12  1  10  2  14  4
8  7  3   5  6

Array after heapsort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

5.4 Priority queues

Heaps are often used to implement *priority queues*. A priority queue is a container whose elements have an associated *priority*. The queue implements two operations with high efficiency: adding an element to the queue (regardless of its priority) and removing the highest priority element from the queue.

We implement a priority queue as a structure Q with fields:

- $Q.A$ is an array with storage for the queue elements.
- $Q.\text{size}$ is the number of elements currently in the queue.

Enqueuing elements is similar to the `BuildHeap` algorithm above:

PriorityEnqueue(Q, x):

1. Let $i \leftarrow Q.\text{size}$
2. Set $Q.A_i \leftarrow x$
3. Interpret $(Q.A_0, \ldots, Q.A_i)$ as a complete binary tree T and let S be the subtree rooted at A_i.
4. Call SiftUp(S).
5. Set $Q.\text{size} \leftarrow i + 1$.

Dequeuing elements is instead similar to `HeapSort`:

PriorityDequeue(Q, x):

1. Let $i \leftarrow Q.\text{size}$
2. Swap A_0 and A_i.
3. Interpret $(Q.A_0, \ldots, Q.A_{i-1})$ as a complete binary tree T.
4. Call SiftDown(T).

5. Set Q.size $\leftarrow i - 1$.
6. Return A_i.

5.4.1 C++ implementation of a priority queue

The following `priority_queue.hpp` file defines the priority queue operations using a `vector` for backing storage and the heap functions defined above.

File `priority_queue.hpp`:

```
#ifndef __priority_queue__
#define __priority_queue__

#include "heap.hpp"
#include <vector>

template <typename V, typename C = default_comparison_t<V>>
void priority_dequeue(std::vector<V> &storage, C compare = C{})
{
    assert(storage.size() > 0);
    std::iter_swap(begin(storage), end(storage) - 1);
    storage.pop_back();
    heap_sift_down(CompleteBT<V>{&storage, 0, storage.size()},
                   compare);
}

template <typename V, typename C = default_comparison_t<V>>
void priority_enqueue(std::vector<V> &storage, V value,
                      C compare = C{})
{
    storage.push_back(std::move(value));
    heap_sift_up(
        CompleteBT<V>{&storage, storage.size() - 1, storage.size()},
        compare);
}

#endif /* __priority_queue__ */
```

The following test driver tests the priority queue.

File `priority_queue_driver.cpp`:

```
#include "priority_queue.hpp"
#include <utils.hpp>

#include <iostream>

int main(int argc, char **argv)
{
    std::vector<int> queue;

    for (int x : {15, 9, 3, 23}) {
        std::cout << "Enqueued " << x << ' ';
        priority_enqueue(queue, x);
        print(queue);
    }
```

```
15
16      std::cout << "Dequeued " << queue[0] << ' ';
17      priority_dequeue(queue);
18      print(queue);
19
20      std::cout << "Dequeued " << queue[0] << ' ';
21      priority_dequeue(queue);
22      print(queue);
23
24      for (int x : {2, 1}) {
25          std::cout << "Enqueued " << x << ' ';
26          priority_enqueue(queue, x);
27          print(queue);
28      }
29
30      while (!queue.empty()) {
31          std::cout << "Dequeued " << queue[0] << ' ';
32          priority_dequeue(queue);
33          print(queue);
34      }
35
36      return 0;
37  }
```

```
Enqueued 15 [15]
Enqueued 9 [15, 9]
Enqueued 3 [15, 9, 3]
Enqueued 23 [23, 15, 3, 9]
Dequeued 23 [15, 9, 3]
Dequeued 15 [9, 3]
Enqueued 2 [9, 3, 2]
Enqueued 1 [9, 3, 2, 1]
Dequeued 9 [3, 1, 2]
Dequeued 3 [2, 1]
Dequeued 2 [1]
Dequeued 1 []
```

Chapter 6

Hashing

So far we have focused on data structures that allow to store and retrieve elements in a specified order, or by priority. Often, however, one needs to access elements in a random order, based on identifiers.

Arrays are an example of a random access data structure because they allow to access any element in constant time $O(1)$ based on their index. The data identifiers in arrays are the integers from 0 to the array size minus 1. In this chapter, we will introduce data structures that allow to access data in a similar manner, but based on identifiers of arbitrary types (e.g., character strings).

The key concept is the one of *hash table*. A hash table is a container H that allows inserting and retrieving elements in arbitrary order based on generic identifiers or *keys*. Specifically, H has two operations:

- $\text{insert}(H, k, v)$ stores value v in the hash table H for key k.
- $\text{retrieve}(H, k)$ returns the value v associated to key k if any such association exists, or NIL otherwise.

Next, we will look at how a hash table can be implemented via a particular technique called chaining and how the latter can be significantly accelerated by using hash functions.

6.1 Hashing via chaining

Chaining implements a hash table H as a linked list. Each node of the list stores a key-value pair $\langle k, v\rangle$. To insert a new element $\langle k, v\rangle$, chaining first checks if the list already contains a node $\langle k, \star\rangle$, where $\star$ is a placeholder that stands for "any value". If so, the algorithm updates $\star$ to be v; otherwise, the algorithm inserts a new node storing $\langle k, v\rangle$ at the beginning of the list. We call this procedure `ChainInsert`:

$\text{ChainInsert}(L, k, v):$

1. $N \leftarrow \text{ListFindPredecessor}(L, \langle k, \star\rangle)$
2. If $N = \text{NIL}$, then:
    1. Call $\text{ListInsertAfer}(L, \langle k, v\rangle)$.
3. Else:
    1. Set $N.\text{next.value} \leftarrow \langle k, v\rangle$.

To retrieve the value of key k from the hash table, chaining searches the list for a corresponding node $\langle k, v\rangle$, and returns v if such a node is found, or `NIL` otherwise. We call this procedure `ChainRetrieve`:

$\text{ChainRetrieve}(L, k):$

1. $N \leftarrow \text{ListFindPredecessor}(L, \langle k, \star\rangle)$
2. If $N = \text{NIL}$, then:
    1. Return NIL.
3. Else:
    1. Let $\langle k, v\rangle \leftarrow N.\text{next.value}$.

2. Return v.

The problem with chaining is that inserting and retrieving elements has worst case complexity $\Theta(n)$, where n is the number of elements stored in the table. This is because, in the worst case, the list must be scanned from start to finish for both insertion and retrieval of keys. By comparison, arrays allow insertion and retrieval in $\Theta(1)$ time!

Chaining can be accelerated substantially by using a large number of short lists, each responsible for storing a given subset of possible keys (this technique is known as "separate chaining"). Each chain corresponds to a *slot* i in the table, and there are m such slots, numbered from 0 to $m-1$.

Given a key k, the algorithm needs to decide *which* slot is responsible for storing it, and must to do so quickly, in constant $\Theta(1)$ time. This is usually achieved by means of a *hash function* h. The hash function takes as input the key k and returns as output the index $s = h(k)$ of the slot to use.

Given the hash function h, the hash table H can be implemented as a data structure with a field $H.A$, which is simply an array containing the m lists. Insertion uses the `HashInsert` algorithm:

HashInsert(H, k, v) :

1. Let $s \leftarrow h(k)$.
2. Let $L \leftarrow H.A[s]$.
3. Call ChainInsert(L, k, v).

Retrieving an element uses the `HashRetrieve` algorithm:

HashRetrieve(L, k) :

1. Let $s \leftarrow h(k)$.
2. Let $L \leftarrow H.A[s]$.
3. Return ChainRetrieve(L, k).

A good hash function distributes keys to slots as uniformly as possible. In this manner, if there are m slots and n elements in the table, we can expect that, on average, each list will contain n/m elements, therefore accelerating insertion and retrieval by a factor of m compared to using a single list.

However, in the *worst case* it may occur that *all* the keys we wish to store in the table are hashed by h to the *same* slot. Hence, the worst case complexity is still $\Theta(n)$. Next, we show that the *average* the cost is much closer to the ideal rather than the worst case.

6.2 Average cost analysis of hashing

In this part, we analyse the *average* cost of *retrieving* elements from a hash table H. This requires making a statistical assumption on how the data is inserted into the table before it is retrieved from it:

Simple uniform hashing assumption (SUHA) (definition)

Let $h : \mathcal{K} \rightarrow \{0, \ldots, m-1\}$ be a hash function mapping keys $k \in \mathcal{K}$ to m slots. We assume that:

1. Each time a key k is inserted in the table, it is selected at random from a fixed distribution $p(k)$, independently of previously-inserted keys.
2. The probability that a key is hashed to any of the slots is uniform, i.e., $P[h(k) = s] = 1/m$ for all slots $s = 0, \ldots, m-1$.

Remark: SUHA states that the slot distribution $P[h(k) = s]$ is uniform, not that the key distribution $p(k)$ is.

We show below that the cost of retrieving elements from a hash table is characterized by its load factor:

Load factor (definition)

The *load factor* of a hash table H containing n elements and m slots is $\alpha = n/m$.

To analyze the cost of retrieving a key from the hash table, we need to distinguish two cases: whether the retrieved key is contained in the table or not.

First, we analyse the cost of retrieving a key k that is *not* contained in the hash table. Intuitively, the cost incurred is proportional to the average length of the chains in the table, i.e. the load factor. This is captured by the following theorem:

> **Expected cost of retrieving a missing key (theorem H1)**
>
> Under the SUHA, averaging over possible hash tables H and retrieved keys k, the expected number of list nodes visited by $\text{HashRetrieve}(H, k)$ for a key k *not* in the table is $1 + \alpha$, where α is the load factor.

Proof. Executing $\text{HashRetrieve}(H, k)$ visits $1 + n_i$ nodes, where $i = h(k)$ is the slot selected by key k and n_i is the length of the list stored at that slot. The plus one accounts for visiting the sentinel node at the beginning of the list.

Suppose that the hash table contains n keys in total and let us find the expected value $E[n_i|n]$ of the length of the i-th list given that the total number of keys is k. Due to the fact that the sum of the list lengths must be n and due to the linearity of expectation, we have:

$$n = E[n_0 + n_1 + \cdots + n_{m-1}|n] = E[n_0|n] + \cdots + E[n_{m-1}|n].$$

Due to the SUHA, all slots are used equally likely, which means that the marginal expectations $E[n_i|n]$ must be the same for all slots i. We thus conclude that $mE[n_i|n] = n$, so that $E[1 + n_i|n] = 1 + n/m = 1 + \alpha$. *Q.E.D.*

Next, we look at the cost of retrieving a key that is actually contained in the table. This case is slightly different from the one above because, on average, the key will be found by visiting only half of the corresponding list; therefore the number of nodes visited should be slightly less than before.

> **Expected average cost of retrieving an existing key (theorem H2)**
>
> Under the SUHA, the expected number of list nodes visited by $\text{HashRetrieve}(H, k)$ averaged overall all possible hash tables H of size n and keys k contained in them is $1 + \alpha/2 - \alpha/2n$.

The proof is a little complex and is given as optional material at the end of the chapter.

The message of these two theorems is that, as long as we keep the load α small, we can expect the table operations to run efficiently, in time $\Theta(1+\alpha)$. This depends on the table being used according to an "average" pattern, captured by the SUHA and, for the case of retrieving a key that exists in the table, by the fact the we are equally likely to retrieve any of the existing keys.

6.3 C++ implementation of hash tables

Next, we look at a possible implementation of the hash table in C++.

File `hash.hpp`:

```
#ifndef __hash__
#define __hash__

#include <cassert>
#include <iostream>
#include <limits>
#include <vector>

#include "../part-1/list.hpp"

template <typename K, typename V, typename H> class HashTable
```

```
{
  public:
    HashTable(size_t num_chains) : _table{num_chains} {}

    void insert(const K &key, const V &value)
    {
        size_t slot = _get_slot(key);
        Node<KeyValuePair> *node = _find_key(slot, key);
        if (!node) {
            list_insert_after(&_table[slot], {key, value});
        } else {
            node->next->value.value = value;
        }
    }

    V *get(const K &key)
    {
        size_t slot = _get_slot(key);
        Node<KeyValuePair> *node = _find_key(slot, key);
        if (!node) { return nullptr; }
        return &node->next->value.value;
    }

    void print(bool details = false) const
    {
        size_t min = std::numeric_limits<size_t>::max(), max = 0, average = 0;
        for (size_t slot = 0; slot < _table.size(); ++slot) {
            if (details) { std::cout << "Slot " << slot << " contains"; }
            size_t count = 0;
            for (Node<KeyValuePair> *node = _table[slot].next.get(); node;
                 node = node->next.get()) {
                if (details) { std::cout << " '" << node->value.key << '\''; }
                count++;
            }
            if (details) { std::cout << " (" << count << ")\n"; }
            max = std::max(count, max);
            min = std::min(count, min);
            average += count;
        }
        std::cout << "Slot sizes: min: " << min;
        std::cout << ", max: " << max;
        std::cout << ", average: " << float(average) / _table.size() << '\n';
    }

  private:
    struct KeyValuePair {
        K key;
        V value;
    };

    std::vector<Node<KeyValuePair>> _table;

    size_t _get_slot(const K &key) const { return H{}(key) % _table.size(); }
```

```
    Node<KeyValuePair> *_find_key(size_t slot, const K &key)
    {
        auto match = [&](const KeyValuePair &pair) { return pair.key == key; };
        return list_find_predecessor(&_table[slot], match);
    }
};

#endif // __hash__
```

The HastTable template is parameterized in the key type K, the value type V, the number of slots m, and the hash function policy H. The key-value pairs stored in the lists are represented by the structure HashTable::Entry. The hash function policy H is an object type that specifies the operation computed to hash keys and is further discussed below. The policy object is used in the Hash<K,V,m,H>::get_slot member function to compute the slot for a given key.

Chaining uses the list.hpp modules we have introduced early in the course. list_find_predecessor is used to find a key-value pair in the list. Nodes in the list are of type Node<Entry> where Entry is a structure storing a key-value pair. list_find_predecessor is called with a suitable comparison function match which check for key equality. match is defined as a lambda function inside of Hash<K,V,m,H>::find_entry.

The following driver demonstrates using this class.

File hash_driver.cpp:

```
#include "hash.hpp"
#include <iostream>
#include <string>

constexpr size_t num_chains = 31;

struct HashFunction {
    size_t operator()(const std::string &str)
    {
        size_t value = 0;
        for (auto c : str) { value += c; }
        return value % num_chains;
    }
};

int main(int agrc, char **argv)
{
    HashTable<std::string, int, HashFunction> hash_table{num_chains};

    int value = 0;
    for (auto key : {"Apple", "Apricots", "Avocado", "Banana", "Blackberries",
                     "Blackcurrant", "Blueberries", "Breadfruit", "Cantaloupe",
                     "Carambola", "Cherimoya", "Cherries", "Clementine"}) {
        hash_table.insert(key, value++);
    }

    hash_table.print();

    std::cout << "'Carambola' is the " << *hash_table.get("Carambola")
              << "-th fruit\n";

```

```
32     std::cout << "Retrieving 'Beans' results in the pointer value "
33               << hash_table.get("Beans") << "\n";
34     return 0;
35 }
```

```
Slot sizes: min: 0, max: 2, average: 0.419355
'Carambola' is the 9-th fruit
Retrieving 'Beans' results in the pointer value 0x0
```

The `HashFunction` policy object is just a structure with no data member whose only purpose is to implement the hash function as its `operator()`. Given a key `key`, the hash function can then be called by instantiating the object and calling it, as in `HashFunction{}(k)`. These "policy" objects are a standard C++ idiom used to parameterize the behaviour of template classes.

6.4 Designing hash functions

We now consider the design of the hash functions h. The goal of a hash function is to spread keys to slots in the most uniform manner possible.

While an optimal design depends on the nature and distribution of the keys k, we can often resort to simple heuristics.

A common strategy starts by interpreting the keys k as a natural number. A way to do so is to take the binary representation of k (i.e., the sequence of bytes that are used to store k in the memory of the computer) and read that as a (usually large) natural number in binary notation. If k is a string $c_0 c_1 \dots c_{n-1}$ of ASCII characters, this number is given by $k = \sum_{i=0}^{n-1} 256^i c_i$. In the following, we thus assume that keys are (potentially large) natural numbers.

While the input key k can ber an arbitrary number, the hash function must return a slot index in the range $[0, m-1]$. A straightforward way of achieving this result is to define:

$$h(k) = k \bmod m.$$

Recall that the $\bmod m$ operator computes the reminder of the division by m, which can be any number in the range 0 to $m-1$.

This construction, however, does not work well for certain values of m. For instance, if we choose $m = 2^8 = 256$ in the example above, we obtain $h(c_0 c_1 \dots c_{n-1}) = c_0$, meaning that all strings that begin with the same character hash to the same slot. Unless all characters are equally likely, and this is almost never the case, then this hash function is not going to work well.

In general, we would like the result of the hash function to at least depends on *all* the bits, or binary digits, of the integer k. One way to achieve this is to choose m to be a prime number other than 2. To see why, consider a pair of keys $k' = k$ that differ only in the i-th bit. Without loss of generality, we can thus assume that $k' = k + 2^i$. If we use the definition above for the hash function, we must have

$$k' = q'm + h(k') = k + 2^i = qm + h(k) + 2^i$$

If the hashes $h(k) = h(k')$ are the same, then we must have:

$$q'm = qm + 2^i \quad \Rightarrow \quad (q' - q)m = 2^i \quad \Rightarrow \quad m | 2^i$$

where the last symbol means that "m divides 2^i". The latter is of course impossible since m is a prime number other than 2 and 2^i has 2 as the sole prime factor. This means that flipping *any* bit in the key k will cause the hash value to change.

Generally, a good choice for m is to pick a prime number not too close to a power of two.

If changing m is not an option, we can consider other kinds of hash functions, such as the multiplication method. However, discussion those is beyond the scope of these notes.

6.5 Proof of theorem H2 (optional)

We prove theorem H2. The proof is a slight modification of the original proof provided by Donald Knuth.

Consider the cost $c(k_1, \ldots, k_n, i)$ of retrieving the i-th key k_i from the hash table H obtained by inserting keys $(k_1, \ldots, k_n)$ in this order. We need to compute the expectation of this cost over all possible key sequences and which one of them is retrieved:

$$C = \frac{1}{n} \sum_{i=1}^{n} \int c(k_1, \ldots, k_n, i) p(k_1) \cdots p(k_n) dk_1 \cdots dk_n$$

In order to simplify this calculation, we note that the specific values of the keys do not really matter. In fact, the cost c is the same as long as the sequence of slots $s_i = h(k_i)$ activated by the keys is the same:

$$(s_1, \ldots, s_n) = (s'_1, \ldots, s'_n) \quad \Rightarrow \quad c(k_1, \ldots, k_n, i) = c(k'_1, \ldots, k'_n, i)$$

Hence, instead of averaging over key values, we can average over possible slot sequences. There are m^n possible sequences of slots $(s_1, s_2, \ldots, s_n) \in [0, m-1]^n$ and, because of the SUHA, they are all equally probable[1]. We conclude that the cost can be rewritten as an average over such sequences:

$$C = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{m^n} \sum_{(s_1, s_2, \ldots, s_n) \in [0, m-1]^n} c(s_1, \ldots, s_n, i).$$

The cost of retrieving key k_i is given by

$$c(s_1, \ldots, s_n, i) = 1 + r$$

where r is the number of keys that are found in the chain in slot s_i *before* the retrieved key k_i itself. Noting that `ChainInsert` always adds new keys at the beginning of a chain, this is the number of keys added to the table *after* k_i that hash to the same slot; namely, r is the number of slots $(s_{i+1}, \ldots, s_n)$ that are equal to s_i.

Hence, for a given value of i and r, we must count for how many of the m^n slot sequences $(s_1, \ldots, s_n)$ we have $c(s_1, \ldots, s_n, i) = 1 + r$. In order to obtain a sequence with this particular value for the cost, we can choose the first i slots $s_1, \ldots, s_i$ arbitrarily, but for the remaining slots $s_j, j = i+1, \ldots, n$, exactly r of them must be the same as s_i (i.e., $s_j = s_i$) and the other $n - i - r$ ones can only take $m - 1$ different values $s_j \neq s_i$. Schematically, we have the following options for choosing the slot sequence:

$$\underbrace{s_1, \ldots, s_i,}_{m^i \text{ options}} \quad \underbrace{s_{i+1}, \ldots, s_n}_{\binom{n-i}{r}(m-1)^{n-i-r} \text{ options}}$$

Because all such sequences have the same cost $1 + r$, we can rearrange the sum above and sum over all possible values of r obtaining the expression:

$$C = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{m^n} \sum_{r=0}^{n-i} m^i \binom{n-i}{r} (m-1)^{n-i-r} (1 + r).$$

[1] This is a slight approximation: in the construction above, keys are assumed to be all distinct — if two identical keys are sampled, we need to discard the second occurrence and sample another one, meaning that keys cannot be assumed to be exactly i.i.d. This in turn makes certain sequences of slots to be slightly more probable than others. We ignore this effect as, in practice, the space of keys is usually very large, so the probability of sampling two identical keys is very small.

Computing this sum requires a little work. First, we consider the constant 1 in the factor $1+r$:

$$\begin{aligned}
C_1 &= \frac{1}{n}\sum_{i=1}^{n}\frac{1}{m^n}\sum_{r=0}^{n-i} m^i \binom{n-i}{r}(m-1)^{n-i-r} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}\sum_{r=0}^{n-i} \binom{n-i}{r}1^r(m-1)^{n-i-r} \quad \text{(binomial expansion)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}(1+m-1)^{n-i} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}m^{n-i} = 1.
\end{aligned}$$

The fact that this sum is equal to 1 is not a chance: it is the result of averaging the constant 1 over all sequences. The fact that the result is 1 provides a verification of the correctness of the formula above.

Next, we compute the r part of $1+r$. For this, we use a few Z-transform tricks that should be familiar to you:

$$\begin{aligned}
C_2 &= \frac{1}{n}\sum_{i=1}^{n}\frac{1}{m^n}\sum_{r=0}^{n-i} m^i \binom{n-i}{r}(m-1)^{n-i-r}r \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}\sum_{r=0}^{n-i} \binom{n-i}{r} z^{n-i+1} r z^{-r-1} \quad \text{for } z = m-1 \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}z^{n-i+1}\sum_{r=0}^{n-i} \binom{n-i}{r} \frac{d}{dz}\left[-z^{-r}\right] \quad \text{(derivative trick)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}z^{n-i+1}\frac{d}{dz}\left[-\sum_{r=0}^{n-i} \binom{n-i}{r} z^{-r}\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}z^{n-i+1}\frac{d}{dz}\left[-\sum_{r=0}^{n-i} \binom{n-i}{r} 1^{n-i-r}(z^{-1})^r\right] \quad \text{(binomial expansion)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}z^{n-i+1}\frac{d}{dz}[-(1+z^{-1})^{n-i}] \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}(n-i)z^{n-i+1}(1+z^{-1})^{n-i-1}z^{-2} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{m^i}{m^n}(n-i)(z+1)^{n-i-1} = \frac{1}{n}\sum_{i=1}^{n}\frac{m^n}{m^n}(n-i)m^{-1} \\
&= \frac{n}{m} - \frac{1}{mn}\sum_{i=1}^{n} i = \frac{n}{m} - \frac{1}{mn}\frac{n(n+1)}{2} \\
&= \frac{\alpha}{2} - \frac{\alpha}{2n}.
\end{aligned}$$

We conclude the proof by noting that the total average cost is $C = C_1 + C_2$. *Q.E.D.*

Chapter 7

Graphs

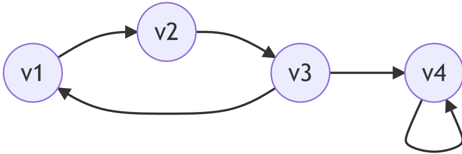
This chapter introduces the concept of *graph*, one of the most important data structure in computer science. A graph describes the connections (edges) between entities (vertices). As an example, the vertices can be geographic locations, and the edges roads between them.

In this chapter we introduce basic definitions and notations for graphs. In the next one, we will look at shortest paths algorithms, which are used in a wide spectrum of applications.

7.1 Formal definition of graph

A **directed graph** $G = (V, E)$, also called a *digraph* or simply a *graph*, is a finite set of *vertices* V together with a set of *edges* $E \subset V \times V$. An edge $e = (u, v)$, which is a pair of vertices, is visualized as arrow $u \mapsto v$ connecting vertex u to v. By analogy with an arrow, the first vertex u is called the *tail* of the edge the second vertex v is called its *head*. It is possible for the tail and head to coincide, i.e., (u, u) is a valid edge in a directed graph.

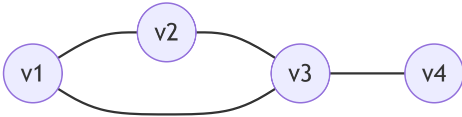
For example, the following figure shows a graph with four vertices $V = \{v_1, v_2, v_3, v_4\}$:



The set of edges of this graph is $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_3, v_4), (v_4, v_4)\}$.

An **undirected graph** is a graph where the edges E do not have an orientation. In this case, edges are represented as *set* of two vertices $\{v_1, v_2\}$ instead of as a pair. This is because the order of the elements in a set is irrelevant, so that $\{v_1, v_2\} = \{v_2, v_1\}$. Edges whose head and tail coincide are *not* allowed in an undirected graph.

The following figure shows an undirected graph:



In the rest of the notes, we will mainly focus on directed graphs.

7.2 Representing graphs using adjacency matrices and lists

A simple representation of a graph G that can be used in an algorithm is its **adjacency matrix** A. To build the adjacency matrix, one enumerates the vertices of the graph, so that each vertex $v \in V$ is identified with a number in the range 1 to V. Then, the entry A_{uv} of the adjacency matrix is set to 1 if there is an edge $(u, v) \in E$ and to 0 otherwise, resulting in a $|V| \times |V|$ binary matrix.

For example, the adjacency matrix for the directed graph given above is given by:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

An undirected graph can be represented by a symmetric adjacency matrix A (i.e., $A = A^\top$). Since self-edges are not allowed, there are zeros along the diagonal. For example, the adjacency matrix for the undirected graph above is given by:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

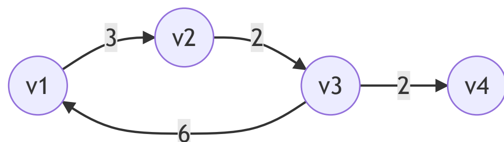
Adjacency matrices are inefficient if the graph is sparse. A graph is considered **sparse** if the number of edges $|E|$ is a small fraction of the theoretical maximum $|V|^2$ (in the analysis of an algorithm, we would characterise a sparse graph by stating that the number of edge is linear in the number of vertices, i.e., $|E| = O(|V|)$). For such a graph, a more efficient representation is given by the **adjacency list**: a list of lists, each specifying the edges outgoing from a given vertex. For example, the adjacency list for the directed graph above is:

$$L = [[2],\ [3],\ [1, 4],\ [4]]$$

This states that the first vertex v_1 has an edge connecting it to v_2, the second vertex v_2 has an edge connecting it to v_3, the third vertex v_3 has an edge connecting it to v_1 and another connecting it to v_4, and the fourt vertex v_4 has an edge connecting it to itself.

7.3 Weighted graphs

A **weighted graph** (G, w) is a graph with **weights** $w(e), e \in E$ associated to its edges, as in the following example:



We can use a variant of the adjacency matrix A to represent a weighted graph. Specifically, we introduce a **weight matrix** W such that $W_{uv} = w(u, v)$ with the convention that $w(u, v) = +\infty$ if there is no edge connecting u to v. In the example above, the weight matrix is given by:

$$W = \begin{bmatrix} \infty & 3 & 6 & \infty \\ \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty \end{bmatrix}.$$

The adjacency list can also be modified to represent a weighted graph. In this case, each element in the list stores the head of an edge as well as its weight. For the graph above, the adjacency list is:

$$L = [[(2, 3)],\ [(3, 2)],\ [(1, 6), (4, 2)],\ []]$$

7.4 Paths

A **path** in a *directed* graph $G = (V, E)$ is a sequence of vertices $p = (v_1, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ is an edge for each $i = 1, \ldots, n-1$.

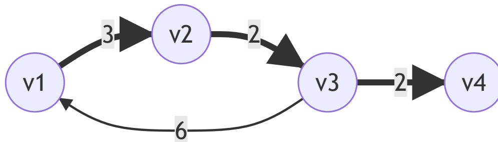
The **length** of the path is the number of edges in the path and is thus equal to $n - 1$ (the number of vertices minus one). A path of length zero is a sequence (v) comprising a single vertex.

The first vertex v_1 of a path is called the **source** and the last vertex v_n is called the **destination**. We say that the path **connects** the source v_1 to the destination v_n.

The **weight** of a path $p = (v_1, \ldots, v_n)$ in a weighted graph (G, w) is the sum of the weights of the edges appearing in the path:

$$w(p) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}).$$

For example, in the figure below the path $p = (v_1, v_3, v_4)$ connects v_1 to v_4 and its weight is 7.



Note that, if an edge is included more than once in a path, its weight is summed a corresponding number of times.

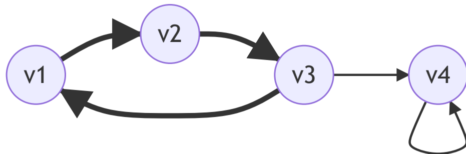
We say that a path is **simple** if no vertex is repeated. I.e., a simple path never visits the same vertex more than once.

If the destination of a path p' coincides with the source of a path p'', we can **concatenate** p'' to p', obtaining a composite path p. We denote this operation by using the circled plus symbol $p = p' \oplus p''$. For example, $(v_1, v_2, v_3, v_4) = (v_1, v_2, v_3) \oplus (v_3, v_4)$.

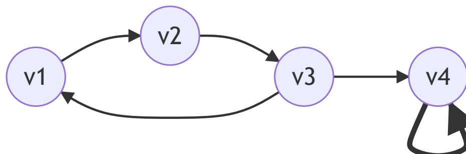
If $p = p' \oplus p'' \oplus p''$, then we say that paths p', p'' and p''' are **subpaths** of p.

7.5 Cycles

In a *directed* graph, a **cycle** is a path of length greater than zero where the source and destination coincide. For example, the path (v_1, v_2, v_3, v_1) in the following graph is a cycle:



Another cycle is (v_4, v_4):



A cycle is **simple** if only the first and last vertex repeat. Hence, the two previous examples of cycles are simple.

Note that the path (v_4, v_4) is not the same as the path (v_4). The first one is a cycle, but the latter is not, despite the fact that source and destination coincide in both cases. The reason is that path (v_4) has a length of zero.

7.6 C++ implementation

We consider two possible implementation of weighted graphs: via an weighed adjacency matrix and as a weighted adjacency list.

The weighted adjacency matrix `Graph` is implemented as a vector of vectors of floats, each specifying a row of the matrix.

The weighted adjacency list `SparseGraph` is also a vector of vectors, but each inner vector specifies a list of edges connecting the corresponding vertex to other vertices in the graph, each given by a structure `hop_t`, specifying a destination `vertex` and a `weight`.

These ideas are captured by the following example implementation:

```
#ifndef __graph_hpp__
#define __graph_hpp__

#include <iostream>
#include <limits>
#include <vector>

// Weighted adjacency matrix representation
using Graph = std::vector<std::vector<float>>;

void print_graph(const Graph &graph, bool as_url = false);

// Weighted adjacency list representation
struct hop_t {
    float weight;
    int vertex;
};

inline bool operator<(const hop_t &lhs, const hop_t &rhs)
{
    return lhs.weight < rhs.weight;
};

using SparseGraph = std::vector<std::vector<hop_t>>;

void print_graph(const SparseGraph &graph, bool as_url = false);

inline std::ostream &operator<<(std::ostream &os, const hop_t &hop)
{
    return os << "(" << hop.weight << "," << hop.vertex << ')';
}

constexpr auto inf = std::numeric_limits<float>::infinity();

// Test graphs
extern const Graph test_graph;
extern const SparseGraph sparse_test_graph;
```

```
38
39  #endif //__graph_hpp__
```

The code also defines the shorthand `inf` to denote the float value infinity.

The code provides two overloads of the function `print_graph` for printing `Graph` and `SparseGraph`, respectively. `print_graph` converts the input graph into the Dot format, which can be used to visualize the graph using the Graphviz software. Passing `true` as second argument of `print_graph` outputs an URL that can be pasted in a Internet browser to visualize the graph. Finally, the module `graph.hpp` also declares two functions to test the code provided: `test_graph` and `sparse_test_graph`.

The following test driver tests this code to display the test graphs:

```
1  #include "graph.hpp"
2
3  int main(int argc, const char *argv[])
4  {
5      print_graph(test_graph);
6      print_graph(sparse_test_graph);
7      return 0;
8  }
```

```
digraph G {
    0 -> 1 [label= 4];
    0 -> 7 [label= 8];
    1 -> 7 [label= 11];
    2 -> 5 [label= 4];
    2 -> 8 [label= 2];
    3 -> 4 [label= 9];
    3 -> 5 [label= 14];
    4 -> 5 [label= 10];
    5 -> 6 [label= 2];
    6 -> 3 [label= 3];
    6 -> 7 [label= 1];
    6 -> 8 [label= 6];
    7 -> 8 [label= 7];
}

digraph G {
    0 -> 1 [label= 4];
    0 -> 7 [label= 8];
    1 -> 7 [label= 11];
    2 -> 5 [label= 4];
    2 -> 8 [label= 2];
    3 -> 4 [label= 9];
    3 -> 5 [label= 14];
    4 -> 5 [label= 10];
    5 -> 6 [label= 2];
    6 -> 3 [label= 3];
    6 -> 7 [label= 1];
    6 -> 8 [label= 6];
    7 -> 8 [label= 7];
}
```

The output can be copied and pasted into this Graphviz online for visualization (you can also try to use this link which copies this example graph into Graphviz online and shows it to you).

If you are curious, the definition file **graph.cpp** is as follows:

```
#include "graph.hpp"

#include <cmath>
#include <iomanip>
#include <iostream>
#include <sstream>

const Graph test_graph =
    Graph{{inf , 4  , inf, inf , inf, inf , inf, 8  , inf}  ,
          {inf , inf, inf, inf , inf, inf , inf, 11 , inf}  ,
          {inf , inf, inf, inf , inf, 4   , inf, inf, 2}    ,
          {inf , inf, inf, inf , 9  , 14  , inf, inf, inf}  ,
          {inf , inf, inf, inf , inf, 10  , inf, inf, inf}  ,
          {inf , inf, inf, inf , inf, inf , 2  , inf, inf}  ,
          {inf , inf, inf, 3   , inf, inf , inf, 1  , 6}    ,
          {inf , inf, inf, inf , inf, inf , inf, inf, 7}    ,
          {inf , inf, inf, inf , inf, inf , inf, inf, inf}};

const SparseGraph sparse_test_graph =
    SparseGraph{
         {{4,1}, {8,7},},
         {{11,7},},
         {{4,5}, {2,8},},
         {{9,4}, {14,5},},
         {{10,5},},
         {{2,6},},
         {{3,3}, {1,7}, {6,8},},
         {{7,8},},
         {}};

static SparseGraph _graph_to_sparse(const Graph &graph)
{
    SparseGraph sp;
    for (const auto &row : graph) {
        std::vector<hop_t> outgoing;
        for (size_t v = 0; v < row.size(); ++v) {
            if (std::isfinite(row[v])) { outgoing.push_back({row[v], (int)v}); }
        }
        sp.push_back(outgoing);
    }
    return sp;
}

static std::string _graph_to_dot(const SparseGraph &sp)
{
    std::ostringstream oss;
    oss << "digraph G {" << std::endl;
    for (int v = 0; v < (int)sp.size(); ++v) {
        for (auto const &hop : sp[v]) {
            oss << "    " << v << " -> " << hop.vertex
                << " [label= " << hop.weight << "];" << std::endl;
        }
```

```
53     }
54     oss << "}" << std::endl;
55     return oss.str();
56 }
57
58 void print_graph(const SparseGraph &graph, bool as_url)
59 {
60     std::string str = _graph_to_dot(graph);
61     if (as_url) {
62         std::ostream os{std::cout.rdbuf()}; // to save the iomanip state
63         os << "https://dreampuf.github.io/GraphvizOnline/#";
64         std::string str = _graph_to_dot(graph);
65         for (const auto &c : str) {
66             os << '%' << std::hex << std::setfill('0') << std::setw(2)
67                << static_cast<int>(c);
68         }
69         os << std::endl;
70     } else {
71         std::cout << str << std::endl;
72     }
73 }
74
75 void print_graph(const Graph &graph, bool as_url)
76 {
77     print_graph(_graph_to_sparse(graph), as_url);
78 }
```

Chapter 8

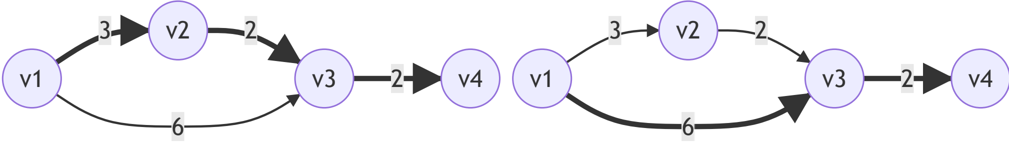
Shortest paths

This chapter looks at the problem of establishing shortest paths in a weighted directed graph. Finding shortest paths itself a key problem in discrete optimization and has countless applications. For example, a graph can be used to represent geographic locations connected by roads, and finding shortest paths between locations is the basis of automated navigation systems, for personal use as well as for optimizing the logistics of public transportation, emergency vehicles, supply chain management, deliveries, etc. Some Internet protocols use shortest paths to establish optimal routes between computers. Shortest path algorithms are also used to program the movement of non-player characters in video games, to plan the motion of robots, in the statistical analysis of social networks, in controlling elevators, in the design of complex integrated circuit, etc. Shortest paths are also often used in computational geometry.

8.1 Definition of shortest path

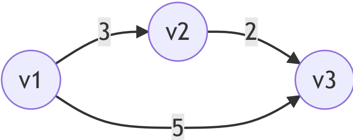
Given a weighted directed graph (G, w), we are interested in finding shortest paths p between some of all pairs of vertices u and v. A path p that connects u to v is *shortest* if it has minimum weight $w(p)$ among all the paths that also connect u to v. In other words, for any path q that connects u to v, p is shortest iif $w(p) \leq w(q)$.

For example, there are only two paths $p_1 = (v_1, v_2, v_3, v_4)$ and $p_2 = (v_1, v_3, v_4)$ connecting v_1 and v_4 in the following graph:



The paths have weight $w(p_1) = 7$ and $w(p_2) = 8$. Hence, p_1 is a shortest path, whereas p_2 is not.

There can be several paths of minimal length connecting two given vertices. Generally, we are interested in finding only one of them. For example, in the graph



both (v_1, v_2, v_3) and (v_1, v_3) are shortest paths connecting v_1 to v_3, because they both have weight 5. Thus, we consider these two paths to be equally good.

Remark: Note that "shortest" does not refer to the number of edges in the path (which is its length) but to its weight. Hence, it is more accurate to call such a path "lightest", but this is a less common terminology. The length and weight of a path coincide only if all edges have a weight of one.

8.2 Existence of shortest paths

Before considering the problem of computing shortest paths, we should ask whether they exist or not. We can formalize this problem as follows. Denote by $\mathbf{P}_{uv}$ the set of all paths connecting u to v. A shortest path from u to v is a lightest path among all of these:

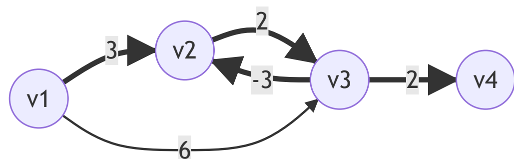
$$p^* \in \operatorname{argmin}_{p \in \mathbf{P}_{uv}} w(p).$$

We use the $\in$ symbol to clarify that p^* is potentially just one of several equivalently good shortest paths. The question if whether the set of shortest paths is empty or not.

For instance, if there are no paths connecting u to v, then $\mathbf{P}_{uv} = \{\}$ is empty and there are no shortest paths either. However, even if $\mathbf{P}_{uv} \neq \{\}$ is not empty, meaning that there is at least one path connecting u to v, it is still possible that there is no *shortest* one. The problem is that there can be a negative cycle.

We say that a directed weighed graph (G, w) contains a **negative cycle** if there is a cycle p with negative weight $w(q) < 0$. If a path p connecting u to v contains a negative cycle q, then one can create new paths that loop around the cycle q an arbitrary number of times, lowering the cost of the path arbitrarily. In this case, while there are infinite paths connecting u to v, there is no shortest path, as one can always find an even shorter one.

For example, in the following graph, one can reach v_4 from v_1 through paths with weights $w(v_1, v_2, v_3, v_4) = 7$, $w(v_1, v_2, v_3, v_2, v_3, v_4) = 6$, $w(v_1, v_2, v_3, v_2, v_3, v_2, v_3, v_4) = 5$ and so on, using the negative cycle (v_2, v_3, v_2) any number of times:



Formally, the problem is that the set $\mathbf{P}_{uv}$ contains an infinite number of paths with arbitrarily small weight, so there is no path which is shortest.

We can fix this problem by requiring the graph to have no negative cycles. In this case, if there is a path p connecting u to v, then there is also a shortest path.

To show this fact, note that, if there is a path p that connects u to v and there are no negative cycles, then we can find a *simple* path p' that also connects u to v which is at least as good as p, i.e., $w(p') \leq w(p)$. In fact, if p is *not* simple, then there is an intermediate vertex r that repeats, meaning that the path is of the form $p = (u, \dots, r, \dots, r, \dots, v)$ (it is possible that $u = r$ or $v = r$, and the argument still works). Because the weight $w(r, \dots, r) \geq 0$ of the cycle is non-negative, we can cut it out of p and obtain a new path p' with the same or smaller weight still connecting u to v. By repeating this construction as needed, we can eventually remove all cycles from p until we are left with a simple path p' as good as p.[1]

The importance of what we have just shown is that we can restrict the search of shortest paths to *simple* paths only. Specifically, let $\mathbf{P}'_{uv} \subset \mathbf{P}_{uv}$ be the subset of paths connecting u to v that are also simple. Because it is not possible for non-simple paths to be shorter than these (assuming that there are no negative cycles), we have

$$p^* \in \operatorname{argmin}_{p \in \mathbf{P}_{uv}} w(p) = \operatorname{argmin}_{p \in \mathbf{P}'_{uv}} w(p).$$

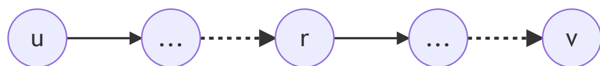
[1] This process of elimination must terminate because p has finite length and each time we remove at least one vertex from it.

The key is that the umber of simple path in a graph is *finite* (because there is a finite number of vertices, these paths must have finite length). Hence, differently from $\mathbf{P}_{uv}$, the set $\mathbf{P}'_{uv}$ is finite. This is sufficient to guarantee that there is indeed a path p^* of minimum weight.

8.3 Optimal substructure of shortest paths

Efficient shortest paths algorithms are based on the fact that all subpaths of a shortest path must be shortest paths as well.

To see this, let $p = (u, \ldots, r, \ldots, v)$ be a path connecting vertex u to v through the intermediate vertex r:



Let $p' = (u, \ldots, r)$ and $p'' = (r, \ldots, v)$ be the two subpaths obtained by splitting p at r. If p is a shortest path from u to v, then p' and p'' must be shortest paths connecting u to r and r to v, respectively. This is because the cost $w(p)$ of the full path p is the sum $w(p') + w(p'')$ of the costs of the subpaths. Hence, if we can improve either subpaths p' or p'', then we can also improve p, which means that p cannot be shortest.

This is an example of an *optimal substructure property*: the fact that the object p is optimal implies that its component objects p' and p'' are optimal as well. This is useful in the computation of shortest paths because it implies that shortest paths are composed of other, smaller, shortest paths.

Note that the optimal substructure property is a *necessary* but not a *sufficient* condition. It is not enough to concatenate two shortest paths to obtain a new shortest path. It only means that the two component paths must be shortest in order for their composition to *possibly* be shortest.

8.4 A compact representation of shortest paths

A first interesting application of the optimal substructure problem is to provide a compact representation of a large number of shortest paths.

Specifically, suppose that we have identified all shortest path p_{uv} from any vertex $u \in V$ in the graph to any other vertex $v \in V$. Let r be the penultimate vertex in the path $p_{uv} = (u, \ldots, r, v)$, i.e.,

$$p_{uv} = p_{ur} \oplus (r, v).$$

Due to the optimal substructure property, $p_{ur} = (u, \ldots, r)$ is a shortest path from u to r and (r, v) is a shortest path from r to v (consisting of a single edge). Furthermore, note that in this decomposition it does not matter *which* particular shortest path p_{ur} we consider. If there is more than one choice for p_{ur}, any of them will result in an overall path $p_{uv} = p_{ur} \oplus (r, v)$ of the same quality. The only thing that matters is the identity of the penultimate vertex r.

Based on this observation, we can encode a full set of shortest path p_{uv} by only remembering the *predecessor* $r = P_{uv} \in V$ of the last vertex v in each of these paths.

In more detail, we introduce the *predecessor matrix* $P \in (V \cup \{-1\})^{V \times V}$. This notation means that the matrix is indexed by a pair of vertices $(u, v) \in V \times V$ and has values $P_{uv} \in V \cup \{-1\}$, where -1 conventionally denotes the case where there is no predecessor. The latter occurs if there is no path at all from u to v, or $u = v$ and the path is (u).

We also usually store a corresponding *distance matrix* $D_{uv} \in (\mathbb{R} \cup \{\infty\})^{V \times V}$, such that $D_{uv} = w(p_{uv}) < \infty$ if there is a shortest path from u to v, and $D_{uv} = \infty$ otherwise.

The pair of matrices (P, D) thus encodes a full set of shortest paths using $O(|V|^2)$ storage only. Considering that each shortest path can contain up to $|V|$ vertices, this is much better than listing all shortest paths explicitly, which would require $O(|V|^3)$ storage.

One of the exercises asks you to write an algorithm to extract all paths encoded by the predecessor matrix P.

8.5 Versions of the shortest paths problem

We will consider two versions of shortest paths problem. The first version seeks for all shortest paths connecting any pair of vertices u and v in the graph:

All-paris shortest paths problem (APSP):

- **Input:** A weighted graph $(G, w) = ((V, E), w)$ with no negative cycles.
- **Output:** The predecessor matrix $P \in (V \cup \{-1\})^{V \times V}$ and the distance matrix $D \in (\mathbb{R} \cup \{\infty\})^{V \times V}$ encoding all shortest paths p_{uv}, using the format described above.

The second version is similar but only seeks for paths connecting a specific source u to all other vertices v in the graph:

Single-source shortest paths problem (SSSP):

- **Input:** A graph $(G, w) = ((V, E), w)$ with no negative cycles and a source vertex $u \in V$. The graph is represented by a weighted adjacency matrix or by a weighted adjacency list.
- **Output:** The predecessor vector $P \in (V \cup \{-1\})^V$ and the distance vector $D \in (\mathbb{R} \cup \{\infty\})^V$ encoding all shortest paths p_v from the same source u.

Note that the vectors P and D in the SSSP problem are corresponding rows of the matrices P and D in the APSP problem.

8.6 Bellman-Ford's algorithm for the SSSP

The Bellman-Ford's algorithm solves the SSSP problem. As the other algorithms described in this chapter, Bellman-Ford maintains a set of putative shortest paths p_v from the source u to all the other vertices v, encoded by predecessor and distance vectors P and D. It then iteratively improvements these paths until they are actual shortest paths.

Initially, there is only one non-degenerate path defined, namely $p_u = (u)$, which has length and weight of zero (this is a shortest path from u to itself because there are no negative cycles). We "pretend" that some paths p_v for all the other vertices $v \neq u$ are also defined, but have infinite weight. This initial state is encoded by setting $D_v \leftarrow \infty$ and $P_v \leftarrow -1$ for all $v \in V$, except for the special case $D_u \leftarrow 0$. Note that $P_u = -1$ because u is *not* a predecessor of itself in the path (u).

The Bellman-Ford's algorithm progressively improves these paths by applying the following *relaxation* procedure repeatedly:

Relax(D, P, w, r, v) :

1. If $D_r + w(r, v) < D_v$:
   1. Set $D_v \leftarrow D_r + w(r, v)$.
   2. Set $P_v \leftarrow r$.
   3. Return true.
2. Return false.

The relax procedure tries to improve path p_v by considering the alternative path $p'_v = p_r \oplus (r, v)$ which uses r as penultimate vertex. If this alternative path is shorter, i.e., $w(p'_v) < w(p_v)$, then P and D are update to replace p_v with p'_v. Additionally, the procedure returns true if the relaxation has a non-trivial effect and false otherwise − this feature allows to slightly simplify the code for some algorithms.

The Bellman-Ford algorithm works by relaxing all edges $(r, v) \in E$ in the graph $|V| - 1$ times:

BellmanFord(V, E, w, u) :

1. For all $v = 1, \ldots, |V|$:
   1. Let $D_v \leftarrow 0$ if $v = u$ or ∞ otherwise.

        2. Let $P_v \leftarrow -1$.
    2. Repeat $|V| - 1$ times:
        1. For all $(r, v) \in E$:
            1. Call Relax(D, P, w, r, v)
    3. Return D and P.

By inspection, the complexity of the algorithm is $\Theta(|V| \cdot |E|)$, i.e., proportional to the product of vertices and edges. For sufficiently dense graphs, the number of edges is in the order of the number of vertices squared, so that the complexity is $\Theta(|V|^3)$.

8.6.1 Proof of correctness

We show that the `BellmanFord` algorithm correctly solves the SSSP. We do so by induction on the length (number of edges) of the shortest paths. After initialization, all shortest paths of length at most 0 are established; after one iteration, all shortest paths of length at most 1 are established; and so on.

This invariant is true after initialization because there is only one shortest path of length zero, i.e., the path (u).

Suppose that the invariant is true for k iterations of the loop at line 2 and let us show that it is true for iteration $k + 1$ as well. Let p_v be one of the shortest paths of length $k + 1$. The key insight is to note that, due to the optimal substructure property, this path can be written as $p_v = (u, \ldots, r, v) = p_r \oplus (r, v)$ where r is the penultimate vertex in the path and p_r is also a shortest path, obviously of length k. By the inductive hypothesis, this means that p_r (or an equally good path) has *already* been found by the algorithm in a prior iteration. After edge (r, v) is relaxed during iteration $k + 1$, p_v is thus set to this path (or to another equally good path).

In this manner, we can conclude that, after k iterations, the algorithm has established all shortest paths of length at most k. In order to determine how many iterations are necessary to find all shortest path, we should thus ask what is their maximum length. We have shown above (see here) that, when there are no negative cycle, the shortest paths can be assumed to be simple. Hence, the maximum length of a shortest path is $|V| - 1$. We conclude that the algorithm has necessarily found all shortest path afters at most $|V| - 1$ iterations.

8.6.2 Detection of negative cycles (optional)

In our definition of the SSSP problem, the graph is required to be free of negative cycles, so that shortest paths are well defined. However, the Bellman-Ford algorithm can also be used to detect if a graph has negative cycles in the first place. The detection is performed by checking if it is *still* possible to lower the cost of any path after $|V| - 1$ iterations of the algorithm. If the graph has no negative cycles, we know that all shortest paths are established in $|V| - 1$ iterations, so no further improvements are possible. However, if there is a negative cycle, one can prove that the algorithm keeps finding shorter paths indefinitely, which can be thus used for detecting such cycles.

8.6.3 C++ implementation of Bellman-Ford's algorithm

The following C++ module implements the `bellman_ford` algorithm:

```
1  #ifndef __shortest_paths_bf__
2  #define __shortest_paths_bf__
3
4  #include "graph.hpp"
5
6  std::vector<hop_t> bellman_ford(const Graph &graph, const int source,
7                                  bool &has_negative_cycle);
8
9  #endif // __shortest_paths_bf__
```

For convenience, vectors D and P are combined in a single `vector<hop_t>` where `hop_t` is a structure, introduced above, that contains a weight (for the D part) and a vertex index (for the P part).

The implementation closely follows the pseudo-code above:

```
#include "shortest_paths_bf.hpp"
#include "shortest_paths_relax.hpp"

#include <cassert>

std::vector<hop_t> bellman_ford(const Graph &graph, const int source,
                                bool &has_negative_cycle)
{
    const int V = static_cast<int>(graph.size());
    assert(0 <= source && source < V);

    auto DP = std::vector<hop_t>(V, {inf, -1});
    DP[source].weight = 0;

    for (int iter = 0; iter < V - 1; ++iter) {
        has_negative_cycle = false;
        for (int r = 0; r < V; ++r) {
            for (int v = 0; v < V; ++v) {
                has_negative_cycle |= relax(graph, DP, r, v);
            }
        }
    }

    return DP;
}
```

Differently from the pseudo-code, this particular implementation visits all possible pairs (r, v) whether they are actual edges or not; this works because, by definition of weighted adjacency matrix, $W_{rv} = \infty$ if there is no edge (i.e., a missing edge is interpreted as an edge with infinite weight).

The following test driver tests the algorithm on the `test_graph`:

```
#include "shortest_paths_bf.hpp"
#include <iostream>
#include <utils.hpp>

int main(int argc, const char *argv[])
{
    Graph graph = test_graph;
    print_graph(graph);

    int source = 2;
    std::cout << "Bellman-Ford SSSP from source " << source << std::endl;

    bool has_negative_cycle;
    auto DP = bellman_ford(graph, source, has_negative_cycle);
    if (has_negative_cycle) {
        std::cout << "The graph has a negative cycle." << std::endl;
    } else {
        print(DP);
    }
```

```
20      std::cout << std::endl;
21      return 0;
22  }
```

```
digraph G {
    0 -> 1 [label= 4];
    0 -> 7 [label= 8];
    1 -> 7 [label= 11];
    2 -> 5 [label= 4];
    2 -> 8 [label= 2];
    3 -> 4 [label= 9];
    3 -> 5 [label= 14];
    4 -> 5 [label= 10];
    5 -> 6 [label= 2];
    6 -> 3 [label= 3];
    6 -> 7 [label= 1];
    6 -> 8 [label= 6];
    7 -> 8 [label= 7];
}

Bellman-Ford SSSP from source 2
[(inf,-1), (inf,-1), (0,-1), (9,6), (18,3), (4,2), (6,5), (7,6), (2,2)]
```

8.7 Floyd-Warshall's algorithm for the APSP

The Floyd-Warshall algorithm solves the APSP problem, computing the shortest paths between *all* pairs of vertices u and v in the graph.

Similarly to the Bellman-Ford's algorithm, the Floyd-Warshall's algorithm works by progressively shortening a set of paths p_{uv} between all pair of vertices, encoded by provisional matrices D and P.

Initially, the only non-degenerate paths are the ones of length zero, i.e., of the type (u) where $u \in V$, and of length one, i.e., of the type (u, v) where $(u, v) \in E$ is an edge. All other paths are initialized to have infinite weight. This situation is encoded by setting $D_{uv} \leftarrow \infty$ and $P_{uv} \leftarrow -1$ except for the cases $D_{uu} \leftarrow 0$ (paths of length zero) and $D_{uv} \leftarrow w(u, v)$ and $P_{uv} \leftarrow u$ when $(u, v) \in E$ is and edge (paths of length one).

The algorithm progressively *relaxes* these paths using the procedure:

$\text{Relax}(D, P, w, r, u, v)$:

1. If $D_{ur} + D_{rv} < D_{uv}$:
    1. Set $D_{uv} \leftarrow D_{ur} + D_{rv}$.
    2. Set $P_{uv} \leftarrow P_{rv}$.

This procedure replaces the current path p_{uv} from u to v with the path $p_{ur} \oplus p_{rv}$ using r as intermediate vertex provided that this new path is in fact *shorter* than the current one.

The Floyd-Warshall's algorithm relaxes all paths p_{uv} by considering each vertex $r \in V$ as a possible intermediate:

$\text{FloydWarshall}(V, E, w)$:

1. For all $u, v \in V$:
    1. Let $D_{uv} \leftarrow w(u, v)$ if $(u, v) \in E$, otherwise ∞.
    2. Let $P_{uv} \leftarrow u$ if $(u, v) \in E$, otherwise -1.
2. For all r in V:
    1. For all u in V:
        1. For all v in V:

1. Call Relax(D, P, w, r, u, v)

3. Return D and P.

The complexity of this algorithm is, clearly, $O(|V|^3)$.

An exercise asks you to prove the correctness of this algorithm, namely to show that it does terminates with all shortest paths correctly computed.

The key insight is that, after r iterations, the algorithm has found all shortest path p^r_{uv} with the additional constraint that only vertices $\{1, 2, \ldots, r\}$ can be used as *intermediate vertices* in those paths.

When (u, v) is relaxed at iteration $r+1$, the shortest path p^{r+1}_{uv} is established. In fact, this can be assumed to be of one of two types: either it uses vertex $r+1$ or it does not. In the latter case, it must be $p^{r+1}_{uv} = p^r_{uv}$. In the former case, we can assume $p^{r+1}_{uv} = p^r_{u,r+1} \oplus p^r_{r+1,v}$ where $p^r_{u,r+1}$ and $p^r_{r+1,v}$ are shortest paths that *terminate* and *begin* at $r+1$ and that only use vertices $\{1, \ldots, r\}$ as *intermediate*. By the inductive hypothesis, these have already been established in a previous iteration of the algorithm, so calling Relax$(D, P, w, r+1, u, v)$ establishes p^{r+1}_{uv} (or an equally good path).

8.7.1 C++ implementation of Floyd-Warshall's algorithm

We provide an example C++ implementation of the Floyd-Warshall algorithm. The algorithm is implemented as a single function:

```
#ifndef __shortest_paths_fw__
#define __shortest_paths_fw__

#include "graph.hpp"

std::vector<std::vector<hop_t>> floyd_warshall(const Graph &graph);

#endif // __shortest_paths_fw__
```

The implementation follows the pseudo-code directly:

```
#include "shortest_paths_fw.hpp"
#include <cmath>

std::vector<std::vector<hop_t>> floyd_warshall(const Graph &graph)
{
    const auto V = static_cast<int>(graph.size());

    auto DP =
        std::vector<std::vector<hop_t>>(V, std::vector<hop_t>(V, {inf, -1}));

    for (int u = 0; u < V; ++u) {
        for (int v = 0; v < V; ++v) {
            if (u == v) {
                DP[u][v].weight = 0;
                DP[u][v].vertex = -1;
            } else if (std::isfinite(graph[u][v])) {
                DP[u][v].weight = graph[u][v];
                DP[u][v].vertex = u;
            }
        }
    }

    for (int r = 0; r < V; ++r) {
```

```
        for (int u = 0; u < V; ++u) {
            for (int v = 0; v < V; ++v) {
                auto duv = DP[u][v].weight;
                auto dur = DP[u][r].weight;
                auto drv = DP[r][v].weight;
                if (dur + drv < duv) {
                    DP[u][v].weight = dur + drv;
                    DP[u][v].vertex = DP[r][v].vertex;
                }
            }
        }
    }

    return DP;
}
```

The following test driver tests the algorithm:

```
#include "shortest_paths_fw.hpp"
#include <iostream>
#include <utils.hpp>

int main(int argc, const char *argv[])
{
    Graph graph = test_graph;
    print_graph(graph);

    std::cout << "Floyd-Warshall all pairs" << std::endl;
    auto paths = floyd_warshall(graph);
    for (const auto &row : paths) { print(row); }
    std::cout << std::endl;

    return 0;
}
```

```
digraph G {
    0 -> 1 [label= 4];
    0 -> 7 [label= 8];
    1 -> 7 [label= 11];
    2 -> 5 [label= 4];
    2 -> 8 [label= 2];
    3 -> 4 [label= 9];
    3 -> 5 [label= 14];
    4 -> 5 [label= 10];
    5 -> 6 [label= 2];
    6 -> 3 [label= 3];
    6 -> 7 [label= 1];
    6 -> 8 [label= 6];
    7 -> 8 [label= 7];
}

Floyd-Warshall all pairs
[(0,-1), (4,0), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (8,0), (15,7)]
[(inf,-1), (0,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (11,1), (18,7)]
[(inf,-1), (inf,-1), (0,-1), (9,6), (18,3), (4,2), (6,5), (7,6), (2,2)]
```

```
[(inf,-1), (inf,-1), (inf,-1), (0,-1), (9,3), (14,3), (16,5), (17,6), (22,6)]
[(inf,-1), (inf,-1), (inf,-1), (15,6), (0,-1), (10,4), (12,5), (13,6), (18,6)]
[(inf,-1), (inf,-1), (inf,-1), (5,6), (14,3), (0,-1), (2,5), (3,6), (8,6)]
[(inf,-1), (inf,-1), (inf,-1), (3,6), (12,3), (17,3), (0,-1), (1,6), (6,6)]
[(inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (0,-1), (7,7)]
[(inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (inf,-1), (0,-1)]
```

8.8 Dijkstra's algorithm for the SSSP

Dijkstra's algorithm solves the SSSP problem. The algorithm is faster than Bellman-Ford's one for sparse graphs, which makes it a better choice for several applications. However, it requires all weights to be non-negative, which is more restrictive than assuming the absence of negative cycles as in the Bellman-Ford's algorithm.

Dijkstra's algorithm works by finding shortest paths from the source vertex u to an expanding set of other vertices. Specifically, the algorithm maintains a collection of paths p_v from the source u to all vertices v, represented as before by the predecessor vector P and the distance vector D. The graph vertices are divided into open ones $Q \subset V$ and closed ones $V - Q$. The paths p_v are shortest for all closed vertices $v \in V - Q$. At every iteration, the algorithm closes one more vertex, thus finding one more shortest path, until all are found.

In order to do so, the algorithm maintains the following two-parts invariant:

- (P1) For all closed vertices $v \in V - Q$, the paths p_v are shortest.
- (P2) For all open vertices $v \in Q$, the weights of the paths p_v are given by:

$$D_v = \min_{r \in V-Q} D_r + w(r, v). \tag{8.1}$$

  In this expression, we take $w(r, v) = \infty$ if there is no edge $(r, v) \in E$.

Initially, all vertices are open, so the algorithm starts by initialising $Q \leftarrow V$ and $D_v \leftarrow \infty$ and $P_v \leftarrow -1$ for all $v \in V$ except for $D_u \leftarrow 0$ (this is the same initialisation as in the Bellman-Ford algorithm).

At every iteration, Dijkstra's algorithm finds the open node v^* with minimal weight D_v, i.e.,

$$v^* = \operatorname{argmin}_{v \in Q} D_v. \tag{8.2}$$

It then "declares" p_{v^*} to be a shortest path by removing v^* from the set Q of open nodes, and then uses relaxation to update D and P in order to preserve the invariant. In pseudo-code, the algorithm is:

Dijkstra(V, E, w, u) :

1. For all $v \in V$:
    1. Let $D_v \leftarrow 0$ if $v = u$ or ∞ otherwise.
    2. Let $P_v \leftarrow -1$.
2. Let $Q \leftarrow V$.
3. While Q is not empty:
    1. Let $v^* \leftarrow \operatorname{argmin}_{v \in Q} D_v$
    2. Remove v^* from Q.
    3. For all $v \in Q$ such that $(v^*, v) \in E$:
        1. Call Relax(D, P, w, v^*, v).
4. Return D and P.

As for the complexity of this algorithm, line 3.3.1 is executed once for each edge in the graph. Line 3.1 is executed once for each vertex and it has cost $O(|V|)$. Hence, the overall cost of `Dijkstra` is $O(|V|^2 + |E|) = O(|V|^2)$.

8.8.1 Proof of correctness

At every iteration of line 3.1, Dijkstra's algorithm closes vertex v^* because path p_{v^*} is a shortest path. To show this fact, note that, by line 3.1 and invariants P1 and P2, this path is of the form $p_{v^*} = p_{r^*} \oplus (r^*, v^*)$ where

$$(r^*, v^*) = \operatorname{argmin}_{r \in V-Q, v \in Q} w(p_{r^*}) + w(r^*, v^*) \tag{8.3}$$

Any other path q from u to v can be written as

$$q = (u, \ldots, r) \oplus (r, v) \oplus (v, \ldots, v^*) \tag{8.4}$$

where r is closed and v is open. None of these paths can be better than p_{v^*} because:

$$w(q) = w(u, \ldots, r) + w(r, v) + w(v, \ldots, v^*) \geq w(p_r) + w(r, v) \geq w(p_{r^*}) + w(r^*, v^*) = w(p_{v^*}).$$

Here, the first inequality uses the fact that $w(v, \ldots, v^*) \geq 0$ because there are no negative weights, and the fact that $w(u, \ldots, r) \geq p_r$ because r is closed and thus p_r is already a shortest path from u to r. The second inequality is trivially satisfied by the definition of (r^*, v^*) from (8.3).

Having thus concluded that p_{v^*} is indeed a shortest path, the next step is to show that lines 3.2 and 3.3 restore invariants P1 and P2 for the next iteration. Line 3.2 removes v^* form Q, thus marking it as closed. Because p_{v^*} is shortest, this preserves property P1, but not necessarily property P2. The latter is restored by line 3.3 by updating vectors D and P. The invariant P2 from equation (8.1) states that, for all open nodes D_v, it must be:

$$D_v = \min_{r \in (V - Q - \{v^*\}) \cup \{v^*\}} D_r + w(r, v).$$

We have rewritten the expression to single out vertex v^* as this was just closed. Because of the invariant P2, before the loop at lines 3.3, D_v is already the minimum over $r \in V - Q - \{v^*\}$ for every open vertex $v \in Q$. Invoking the `Relax` algorithm at line 3.3.1 further extends this minimization to account for the newly closed vertex v^*, thus restoring property P2.

8.9 Using a priority queue

The slowest part of the `Dijkstra` algorithm is searching for the next node to close at line 3.1. This step has cost $O(|V|)$ and is executed $|V|$ times, which makes the bulk of the algorithm's $O(|V|^2)$ complexity. We show now that a *priority queue* can be used to lower this cost substantially.

Specifically, we replace Q with a min-priority queue, storing in it the pairs:

$$(D_r + w(r, v), v), \quad r \in V - Q, \quad v \in Q.$$

In this way, line 3.1 in `Dijkstra` can be replaced by extracting the highest-priority element from the queue. This is in fact equivalent to the combined minimisations (8.1) and (8.2). If we use a min-heap for implementing the queue, this has cost $O(\log |Q|)$ instead of $O(|Q|)$ of a linear search, which is much better.

The resulting algorithm is:

DijkstraPriority(V, E, w, u) :

1. For all $v \in V$:
    1. Let $D_v \leftarrow 0$ if $v = u$ or ∞ otherwise.
    2. Let $P_v \leftarrow -1$.
2. Let $Q \leftarrow \{(0, u)\}$ be a min-priority queue.
3. While Q is not empty:
    1. Let $(d^*, v^*) \leftarrow$ PriorityDequeue(Q).
    2. For all $v \in Q$ such that $(v^*, v) \in E$:
        1. If calling Relax(D, P, w, v^*, v) returns true:
            1. Call PriorityEnqueue$(Q, (D_{v^*} + w(v^*, v), v))$.

4. Return D and P.

After vertex v^* is closed, we call `Relax` as before for all open nodes $v \in Q$. Whenever a relaxation has a non-trivial effect, meaning that it shortens a path, we add the pair $(D_{v^*} + w(r, v^*), v)$ to the queue.

We should also remove all pairs from the queue for the newly closed vertex v^* as this should not be considered any longer; this is possible, but tricky to implement with simple priority queues. Instead, we leave such pairs in the queue. When they are extracted later, the relaxation at line 3.2.1 has no effect for them (as these correspond to paths that are already as short as possible); furthermore, in this case, `Relax` returns false, which skips line 3.2.1.1, so nothing happens (except wasting a bit of time) and it is as if we had removed these vertices from the queue.

As for the complexity, PriorityEnqueue at line 3.1 is called at most once for each edge, so the queue Q never contains more than $|E|$ elements. Thus the cost of `DijkstraPriority` is $O((|V| + |E|) \log |E|) = O((|E| + |V|) \log |V|)$. For a dense graph, the complexity is thus $O(|V|^2 \log |V|)$, which is slightly worse than $O(|V|^2)$ of `Dijkstra`. However, for a sparse graph, the cost is just $O(|V| \log |V|)$, which is much better than $O(|V|^2)$.

8.9.1 C++ implementation of Dijkstra's shortest paths algorithm

Next, we discuss a C++ implementation of `DijkstraPriority`.

The declaration is as simple as for the other algorithms so far:

```
1  #ifndef __shortest_paths_dijkstra__
2  #define __shortest_paths_dijkstra__
3
4  #include "graph.hpp"
5
6  std::vector<hop_t> dijkstra(const Graph &graph, const int source);
7  std::vector<hop_t> dijkstra_priority(const Graph &graph, const int source);
8
9  #endif // __shortest_paths_dijkstra__
```

The C++ implementation is close to the pseudo-code above. Here is the one for `Dijkstra`:

```
1  #include "shortest_paths_dijkstra.hpp"
2  #include "shortest_paths_relax.hpp"
3  #include <priority_queue.hpp>
4
5  #include <cmath>
6
7  std::vector<hop_t> dijkstra(const Graph &graph, const int source)
8  {
9      const int V = static_cast<int>(graph.size());
10     assert(0 <= source && source < V);
11
12     auto DP = std::vector<hop_t>(V, {inf, -1});
13     DP[source].weight = 0;
14
15     std::vector<bool> is_open(V, true);
16
17     while (true) {
18         float D_star = inf;
19         int v_star = -1;
20         for (int v = 0; v < V; ++v) {
21             if (is_open[v] && DP[v].weight < D_star) {
```

```
22                  D_star = DP[v].weight;
23                  v_star = v;
24              }
25          }
26
27          if (v_star < 0) {
28              break; // all closed, stop
29          }
30
31          is_open[v_star] = false;
32
33          for (int v = 0; v < V; ++v) {
34              if (is_open[v] && std::isfinite(graph[v_star][v])) {
35                  relax(graph, DP, v_star, v);
36              }
37          }
38      }
39
40      return DP;
41  }
```

And here is the one for `DijkstraPriority`:

```
1   #include "shortest_paths_dijkstra.hpp"
2   #include "shortest_paths_relax.hpp"
3   #include <priority_queue.hpp>
4
5   #include <cmath>
6
7   std::vector<hop_t> dijkstra_priority(const Graph &graph, const int source)
8   {
9       const int V = static_cast<int>(graph.size());
10      assert(0 <= source && source < V);
11
12      auto DP = std::vector<hop_t>(V, {inf, -1});
13      DP[source].weight = 0;
14
15      auto queue = std::vector<hop_t>{};
16      auto comparison = std::less<hop_t>{}; // for a min-priority queue
17      priority_enqueue(queue, {0, source}, comparison);
18
19      while (!queue.empty()) {
20          auto v_star = queue[0].vertex;
21          priority_dequeue(queue, comparison);
22
23          for (int v = 0; v < V; ++v) {
24              if (std::isfinite(graph[v_star][v])) {
25                  if (relax(graph, DP, v_star, v)) {
26                      priority_enqueue(queue, {DP[v].weight, v}, comparison);
27                  }
28              }
29          }
30      }
31
```

```
    return DP;
}
```

The following test driver tests the algorithm on the usual `test_graph`:

```
#include "shortest_paths_dijkstra.hpp"
#include <utils.hpp>

int main(int argc, const char *argv[])
{
    auto graph = test_graph;
    print_graph(graph);

    {
        int source = 2;
        std::cout << "Dijkstra from source " << source << std::endl;
        auto parents = dijkstra(graph, source);
        print(parents);
        std::cout << std::endl;
    }

    {
        int source = 2;
        std::cout << "Dijkstra priority from source " << source << std::endl;
        auto parents = dijkstra_priority(graph, source);
        print(parents);
        std::cout << std::endl;
    }

    return 0;
}
```

```
digraph G {
    0 -> 1 [label= 4];
    0 -> 7 [label= 8];
    1 -> 7 [label= 11];
    2 -> 5 [label= 4];
    2 -> 8 [label= 2];
    3 -> 4 [label= 9];
    3 -> 5 [label= 14];
    4 -> 5 [label= 10];
    5 -> 6 [label= 2];
    6 -> 3 [label= 3];
    6 -> 7 [label= 1];
    6 -> 8 [label= 6];
    7 -> 8 [label= 7];
}

Dijkstra from source 2
[(inf,-1), (inf,-1), (0,-1), (9,6), (18,3), (4,2), (6,5), (7,6), (2,2)]

Dijkstra priority from source 2
[(inf,-1), (inf,-1), (0,-1), (9,6), (18,3), (4,2), (6,5), (7,6), (2,2)]
```

Appendix A

Appendix (optional)

A.1 Move semantics

C++11 introduced the concept of "move semantics" to optimize away certain redundant copy operations. It also allows smart pointers such as `unique_ptr` to work by making the concept of object ownership more explicitly.

Moving an object onto another is similar to copying it, except that the resources of the source object can be "stolen" and transferred to the destination object. Consider for example the following code:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv) {
  std::vector<float> a {1,2,3,4};
  std::vector<float> b {a}; // copy `a`
  std::vector<float> c {std::move(a)}; // move `a`
  std::cout << "a.size() = " << a.size()
            << ", b.size() = " << b.size()
            << ", c.size() = " << c.size() << '\n';
}
```

```
a.size() = 0, b.size() = 4, c.size() = 4
```

The code creates a vector `a` of four elements, then a vector `b` which is a copy of `a`, and finally a vector `c` by moving (rather than copying) the content of `a` onto `c`. As a result, the final size of `a` is zero and the size of `b` and `c` is four. Note that object `a` still exists and is a valid object: moving does not destroy an object, but `a` is now an empty vector because its content has been moved to `c`, which can be done quickly by changing a data pointer buried in the implementation of the `std::vector` class.

Under the hood, the move behaviour is obtained by calling a *move constructor* instead of a *copy constructor* when the "target" object is constructed. For an object of type `T`, a copy constructor has signature `T(const T&)` and a move constructor has signature `T(T&&)`. Here the important difference is that `T&` is a lvalue reference and `T&&` an rvalue reference. This is usually explained as follows:

> An lvalue refers to an expression that can appear to both sides of the `=` assignment symbol, whereas an rvalue can only appear to the right of it.

A more useful way of thinking about it is that rvalues are temporary objects, created for instance as intermediate values when evaluating an expression like `(a+b)*c`. Here the result of the sum `a+b` is a temporary object that is destroyed immediately after the expression finishes computing. Lvalues are instead objects that last beyond the expression they appear in (for example variables `a`, `b`, and `c` in the expression).

The difference is important because the C++ compiler is allowed to discard rvalue objects as soon as their use in an expression finishes. This gives it an opportunity for optimisation: rather than copying the temporary rvalue to assign it to a target lvalue, it can move it.

The `std::move` function casts an lvalue `T&` to an rvalue `T&&`, therefore signalling to the C++ compiler that the object can be moved from; in practice, this allows the logic of the program to dispose of the current state of the object. Ultimately, what happens depends on the definition of the move constructor `T(T&&)`; in fact, this constructor may not even be declared, in which case the compiler falls back to using the copy constructor `T(const T&)` (because `T&&` is implicitly convertible to `const T&`). So an expression such as `std:move(a)` is best interpreted as "move if you can".

Note that `std::move` does not really "move" anything; it simply casts the object in a way that allows using a move operation on it. For example, `T y(x)` initializes an object `y` by invoking the copy constructor `T(const T&)`, whereas `T y(std::move(x))` invokes the move constructor `T(T&&)`. Temporary objects in expressions are automatically rvalues, so `T z(x+y)` invokes the move constructor `T&&` on the temporary object `x+y` obtained by summing `x` and `y`.

Note that the object moved from is not destroyed or disposed of. Only its state is changed, generally to signify an "empty" state. For example, vector `a` above is still a valid vector after moving, just an empty one. This means that the state must still be valid after moving. In particular, when the destructor `~T()` is eventually called to dispose of the object, this must not crash the program.

A.1.1 Ownership and `unique_ptr`

A `unique_ptr` is a pointer with unique ownership of an object: when the `unique_ptr` is deleted, so is the object. You cannot copy a `unique_ptr` because this would duplicate ownership. However, you can move it:

```
1  #include <utility>
2  #include <memory>
3  #include <iostream>
4
5  int main(int argc, char** argv) {
6    // `a` owns a float object allocated on the heap
7    auto a = std::make_unique<float>(1);
8    std::cout << "a = " << a.get() << '\n';
9    // transfers ownership of the float object to `b`
10   std::unique_ptr<float> b {std::move(a)};
11   std::cout << "a = " << a.get() << ", b = " << b.get() << '\n';
12 }
```

```
a = 0x7fe52d405830
a = 0x0, b = 0x7fe52d405830
```

If you want to support move operations in a class, you must either rely on a move constructor implicitly generated by the compiler, or write one yourself. For example:

```
1  #include <utility>
2  #include <memory>
3  #include <iostream>
4
5  struct T {
6    // Explicitly defined default constructor
7    T() : storage{new float(0)} {}
8    // Explicitly defined move constructor
9    T(T&& t) : storage{t.storage} { t.storage = nullptr; }
10   // Explicitly defined destructor
11   ~T() { if (storage) { delete storage; } }
```

```
  float* storage;
};

struct Q {
  // The compiler implicitly defines:
  // - The copy constructor Q(const Q&)
  // - The move constructor Q(Q&&)
  // - The destructor
  // etc.
  Q() : storage{new float(0)} {}
  std::unique_ptr<float> storage;
};

int main(int argc, char** argv) {
  T a1; // `a` owns a pointer to float
  T b1{std::move(a1)}; // now `b` owns the pointer
  std::cout << "a1.storage = " << a1.storage
            << ", b1.storage = " << b1.storage << '\n';

  Q a2;
  Q b2{std::move(a2)};
  std::cout << "a2.storage = " << a2.storage.get()
            << ", b2.storage = " << b2.storage.get() << '\n';
}
```

```
a1.storage = 0x0, b1.storage = 0x7fce1b405830
a2.storage = 0x0, b2.storage = 0x7fce1b405840
```

> Note tha the move constructor `T(T&& t)` *modifies* the object we are moving from by setting the `t.storage` to `nullptr`. Why?

There are several rules for when C++ implicitly defines constructors for a class. See for example the rule of three.

Utilities such as `std::swap` require input arguments to be move constructible. This means that, when possible, move constructors are used to swap objects more efficiently (instead of creating copies). However, this does not mean that an object *must* have a move constructor for `std::swap` to be usable (see below).

A.1.2 Move constructible and assignable

An object `T x` is *move constructible* if it can be constructed from an rvalue, i.e. if `T y(std::move(x))` compiles without error.

This obviously works if type `T` defines a move constructor `T(T&&)`. However, it also works if it only defines a copy constructor `T(const T&)` (and does not declare a move constructor) because the latter also accepts an rvalue `T&&` as input. However, an object is not move constructible if the move constructor is declared as deleted.

An object is *move assignable* if `T x, y; y=std:move(x);` works, with similar considerations.

Sometimes determining whether an object is move constructible or assignable requires understanding how implicit constructor definitions work. For instance:

```
#include <utility>

struct T1 {
  int x;
```

```
  // The compiler defines implicitly:
  // - the default constructor T1()
  // - the copy constructor T1(const T1&)
  // - the move constructor T1(T1&&)
  // etc.
};

struct T2 {
  int x;
  T2() : x{0} {}
  T2(const T2& t) : x{t.x} {}
  // Because there is a user-defined copy constructor
  // the compiler does not implicitly generate other constructors.
  // In particular, the move constructor T2(T2&&) is
  // not only not defined, but also *not declared*.
};

struct T3 {
  int x;
  T3() : x{0} {}
  T3(const T3& t) : x{t.x} {}
  T3(T3&&) = delete; // Declares the move constructor as deleted
};

int main(int argc, char** argv) {
  T1 a1; // OK: implicit default constructor
  auto b1 = T1(std::move(a1)); // OK: implicit move constructor
  std::swap(a1, b1); // OK: T1 is move constructible

  T2 a2; // OK: default constructor
  auto b2 = T2(std::move(a2)); // OK: copy constructor
  std::swap(a2, b2); // OK: T2 is move constructible

  T3 a3; // OK: default constructors
  // auto b3 = T3(std::move(a3)); // Error: move constructor deleted
  // std::swap(a3, b3); // Error: T3 is not move constructible, required by swap

  return 0;
}
```

A.2 Universal references

Universal references are relevant when move semantics meets templates.

Consider the problem of defining a template function that assigns an object `source` to a target object `target`. We may define the function like this:

```
template <typename T, typename E> void assign(T& target, E&& source) {
  target = std::forward<E>(source);
}
```

Both `T` and `E` are generic types in the template. However, `T&` is declared to be a reference type, whereas `E` is a **universal reference**. In a template context where `E` is a generic type, `E&&` does not mean rvalue reference, but rather is a placeholder for either lvalue or rvalue reference, depending on “what is available” when the

template is instantiated.

The `std::fowrard<E>(source)` syntax inside the template can be though as "move if possible". Namely, `std::forward<E>(source)` is the same as `std::move(source)` if we can move from `source`, or it is the same as `source` otherwise. Depending on the outcome of this step, then either the move `operator=(E&&)` or copy `operator=(const E&)` assignment operator of `T` is used to carry out the assignment.

In short, `assign(target, source)` will result in a copy-assignment operator and `assign(target, std::move(source))` will result in a move-assignment operator (provided that `T` supports it).

This is demonstrated by the following driver:

```
#include <utility>
#include <memory>
#include <iostream>
#include <probes.hpp>

// E&& is a universal reference
template <typename T, typename E> void assign(T& target, E&& source) {
  target = std::forward<E>(source);
}

int main(int argc, char** argv) {
  {
    Movable a{100} ;
    Movable b ;
    // `a` supports move semantics and is cast to an rvalue by std::move
    assign(b, std::move(a));
    std::cout << "a = " << a
              << ", b = " << b << '\n';
  }

  std::cout << '\n';

  {
    Movable b ;
    // `a` supports move semantics and is an rvalue (temporary)
    assign(b, Movable{100});
    std::cout << "b = " << b << '\n';
  }

  std::cout << '\n';

  {
    Movable a{100} ;
    Movable b ;
    // `a` supports move semantics, but std::move is not used, so
    // this is not engaged (`a` is a lvalue).
    assign(b, a);
    std::cout << "a = " << a
              << ", b = " << b << '\n';
  }

  std::cout << '\n';

  {
```

```
    Copyable a{100} ;
    Copyable b ;
    // `a` does not support move semantics: move can still be used
    // to cast `a` to an rvalue but the rvalue is converted back to
    // an lvalue within `assign` as there is no `operator=(Copyable&&)`
    // defined (instead `operator=(const Copyable&)` is used)
    assign(b, std::move(a));
    std::cout << "a = " << a
              << ", b = " << b << '\n';
  }

  return 0;
}
```

```
Movable: Constructed from "100"
Movable: Default-constructed
Movable: Move-assigned
a = 0, b = 100
Movable: Destructed
Movable: Destructed

Movable: Default-constructed
Movable: Constructed from "100"
Movable: Move-assigned
Movable: Destructed
b = 100
Movable: Destructed

Movable: Constructed from "100"
Movable: Default-constructed
Movable: Copy-assigned
a = 100, b = 100
Movable: Destructed
Movable: Destructed

Copyable: Constructed from "100"
Copyable: Default-constructed
Copyable: Copy-assigned
a = 100, b = 100
Copyable: Destructed
Copyable: Destructed
```

Next, we discuss in more detail the universal reference `E&&` in the function arguments; this is *not* a rvalue reference as the use of `&&` might imply. When we call `assign(target, expr)`:

- if `expr` is an rvalue reference of type `U&&`, then `E=U` and the argument `source` of the function `assign` is of type `U&&` (an rvalue reference);
- if `expr` is an lvalue reference of type `U&`, then `E=U&` and the argument `source` of the function `assign` is of type `U& && = U&` (an lvalue reference).

The latter condition `U& && = U&`is due to so called "reference collapsing rules": `& && -> &`, `&& & -> &` and `&& && -> &&`.

Universal references are often used with `std::forward<E>(x)`. All this does is to recast `x` to `E&&`. Hence:

- if `E=U`, then `std:forward<E>(x)` casts `x` to type `E&&`; hence, this is similar to using `std:move(x)`;

- if `E=U&`, then `std:forward<E>(x)` casts `x` to type `E& && = E&` (due to reference collapsing); hence, this is similar to using `x` directly (without moving).

You might wonder why, in the first case, we need to cast `x` to type `U&&` since `x` is of type `U&&` to start with. The reason is that `x` is a named entity (i.e. an object with a name in the code) and named entities always *evaluate* as lvalues even if it they are declared as rvalues.

We can now explain in more detail what happens in this code:

```
1 Movable a{100};
2 Movable b;
3 assign(b, a);
```

`a` evaluates as an lvalue reference to an object of type `Movable` (because `a` is a named entity). In this case, `E=Movable&` and the statement `target = std::forward<E>(source)` in the `assign` function is the same a `target = source`, which invokes the **copy assignment operator** of `Movable`. This is the expected behaviour because we did not authorize the program to move `x`. On the other hand:

```
1 Movable a{100};
2 Movable b;
3 assign(b, std::move(a));
```

converts `x` into a rvalue reference (via `std::move`). In this case, `E=Movable` (rvalue reference) and the statement `target = std::forward<E>(source)` in the `assign` function the same a `target = std::move(source)`, which invokes the **move assignment operator** of `Movable`.

See also:

- Scott Meyer on universal references

Appendix B

Locality-sensitive hashing (optional)

The hash tables introduced in the previous chapter index elements based on an exact match of keys: given a query key q, the hash table H returns the element $\langle k, v\rangle \in H$ such that $k = q$, or `NIL` if no such element can be found.

In many cases, however, we are interested in retrieving keys based on a degree of similarity to a query rather than equality. Namely, given a query q, we would like to find the elements $\langle k, v\rangle$ in the container such that a certain distance function $d(q, k)$ is less or equal to a given threshold τ. There are many applications of LSH, such as retrieving data by using imperfect keys (e.g., strings containing spelling mistakes) to retrieving continuous data such as 3D points in computational geometry.

The hash tables introduced in the previous chapter could be used for this purpose, up to minor modifications, if we could find a hash function h such that $h(k) = h(k')$ whenever $d(k, k') \leq \tau$. Unfortunately, it is clear that this is generally not *not* possible. A hash function divides the space of keys into regions corresponding to different slots, and two keys k and k' may end up across the boundary of two such regions while being very close to each other. In fact, when the keys are high dimensional objects, such as large vectors, this is quite likely to happen. To visualize this, draw random 2D points on a sheet of paper and divide them by drawing a grid: some points will be close to each other while still falling across grid boundaries.

The overall result is a hashing algorithm that occasionally fails to retrieve matching keys even when they are present in the table, but in other cases works correctly and efficiently – we say that the algorithm is correct only with a certain probability.

Locality-sensitive hashing (LSH) is a technique that can improve the probability that such a hashing algorithm is correct by combining several hash tables. To do this, consider a new data structure L with an array $L.H[i]$ of ℓ hash tables, one for each index $i = 1, \ldots, \ell$. We make use of two types of hash functions for each table:

- The functions g_i mapping the (continuous) keys k to discrete quantities $g_i : k \mapsto \hat{k} \in \hat{\mathcal{K}}$ that are suitable for insertion in a standard hash table from the previous chapter;
- And the hash function $h : \hat{\mathcal{K}} \to \{0, \ldots, m-1\}$ used by the standard hash tables $L.H[i]$ as in the previous chapter (we share one such function for all tables).

Fundamental to the analysis of correctness of the LSH are the functions g_i that convert continuous keys to discrete ones because, after discretization, the hash tables $L.H[i]$ work deterministically (i.e., always correctly) as in the previous chapter.

The `LSHInsert` algorithm inserts a new key k into *all* of the ℓ hash tables, each using a different function g_i for discretization. Because two different keys k and k' can be mapped to the same discretized key $\hat{k} = g_i(k) = g_i(k')$, the hash table $L.H[i]$ contains pairs $\langle \hat{k}, S\rangle$ where S is the *set of keys* added to the table that share the same discretization. This explains lines 2-5 in the following pseudo-code for `LSHInsert`:

LSHInsert(L, k) :

1. For $i = 1, \dots, \ell$:
   1. Let $\hat{k} \leftarrow g_i(k)$.
   2. Let $S \leftarrow \text{HashRetrieve}(L.H[i], \hat{k})$.
   3. If $S = \texttt{NIL}$ then set $S \leftarrow \{\}$ (empty set).
   4. Set $S \leftarrow S \cup \{k\}$.
   5. Call $\text{HashInsert}(L.H[i], \hat{k}, S)$.

Given a query q, we would like to retrieve a key k in the container that matches q up to a threshold τ, i.e., $d(q, k) \leq \tau$. In practice, we *relax* the threshold by multiplying it by a factor $c \geq 1$ and seek for a looser match $d(q, k) \leq c\tau$, which will be useful later to characterize the correctness of the algorithm.

The retrieval function `LSHRetrieve` scans the hashed keys in a manner similar to `LSHInsert`. It has two parameters: the relaxed threshold $c\tau$ and a maximum number of distance evaluations m to perform. It compares the query q to the keys until either a key within distance $c\tau$ to the query is found or until m distance evaluations have been performed, after which it gives up:

$\text{LSHRetrieve}(L, q, c\tau, m)$:

1. For $i = 1, \dots, \ell$:
   1. $\hat{q} \leftarrow g_i(q)$.
   2. $S \leftarrow \text{HashRetrieve}(L.H[i], q)$.
   3. Check the keys in $k \in S$ (in any order) and immediately stop and return k if $d(q, k) \leq c\tau$; otherwise, stop and return `NIL` as soon as m comparisons have been performed overall.
2. Return `NIL`.

Assuming that `HashRetrieve` has constant cost $\Theta(1)$, the cost of `LSHRetrieve` is $O(\ell + m)$, as at most ℓ hash tables are visited and at most m distance evaluations are performed.

B.1 Correctness analysis

The question is whether `LSHRetrieve` successfully retrieves the desired key. Differently from the algorithms seen so far, the answer is "often". Namely, we do not expect the algorithm to always work, but to work with a high probability.

Answering the question of correctness requires then making suitable statistical assumptions on the functions g_i:

Locality Sensitive Hashing (LSH) family

A distribution $g \sim G$ over functions forms a LSH family if, and only if: 1. For all **close** keys $d(k, k') \leq \tau$ the collision probability $P[g(k) = g(k')] \geq p_1$ is large. 2. For all **far** keys $d(k, k') > c\tau$ the collision probability $P[g(k) = g(k')] \leq p_2$ is small.

The LSH family is thus characterized by parameters (p_1, p_2, τ, c) where $p_1 > p_2$ are two probabilities, $\tau > 0$ is the distance threshold, and $c \geq 1$ is the relaxation factor for the threshold.

A key k can thus be in three relationships with respect to a given query q: * The key k is **close** to the query q if $d(q, k) \leq \tau$; * The key k is **not far** from the query q if $d(q, k) \leq c\tau$; * The key k is **far** from the query q if $d(q, k) > c\tau$.

Consider now a set of keys $(k_1, \dots, k_n)$ and a query q. The container L is obtained by first sampling ℓ functions from the LSH family G and then using `LSHInsert` to add the n keys to L. We can obtain different versions of the container L by repeating this procedure. Next, we analyze the behavior of the `LSHRetrieve` in expectation over all the containers L obtained in this manner.

LSH retrieval (theorem)

Let G be a LSH family with parameters (p_1, p_2, τ, c), let $k_1, \dots, k_n$ be a fixed set of keys and let q be a fixed query. Futhermore, let L be the container obtained by hashing the keys using a particular sample $g_1, \dots, g_\ell \sim G$ of functions. Then:

1. Calling LSHRetrieve$(L, q, c\tau, m)$ always terminates after at most m distance computations.
2. If all keys k_i are far from the query q (i.e., $d(q, k_i) > c\tau$), then this call always return `NIL`.
3. If there is a key k_i close to the query (i.e., $d(q, k_i) \leq \tau$), then this call returns a key k_j not far from the query (i.e., $d(q, k_j) \leq c\tau$) with probability at least

$$P_{\text{success}} \geq 1 - (1 - p_1)^{\ell} - \frac{n\ell}{m} p_2.$$

Otherwise, the call returns `NIL`.

Given the theorem above, the trick is to choose parameters such that P_{success} is large enough. For example, the following choice for the parameters results in a probability of success of at least 1/3:

1. $\ell = n^{\frac{\log p_1}{\log p_2}}$
2. $m = 4\ell$
3. $p_2 = \frac{1}{n}$

To see why this choice works, note that:

$$p_1 = n^{\log_n p_1} = n^{\frac{\log p_1}{\log n}} = n^{-\frac{\log p_1}{\log p_2}} = \frac{1}{n^{\frac{\log p_1}{\log p_2}}}.$$

Hence, plugging back this expression in the one for the probability of success, we obtain:

$$P_{\text{success}} \geq 1 - \left(1 - \frac{1}{n^{\frac{\log p_1}{\log p_2}}}\right)^{n^{\frac{\log p_1}{\log p_2}}} - \frac{n\ell}{m} p_2 \geq 1 - \frac{1}{e} - \frac{1}{4} \geq \frac{1}{3}$$

where we used the fact that $(1 - 1/a)^a \leq 1/e$ for all $a \geq 1$.

In the construction above, we are free to choose parameters ℓ and m as we wish. However, parameters (p_1, p_2, c, τ) depend on the family G of LSH functions, on which we have less control. In particular, it is not obvious that we can assume $p_2 = 1/n$, as n is the number of keys stored in L which is independent of G.

The following **amplification** trick can be used to modify a given LSH family G until the desired probability value p_2 is obtained. Suppose that we are given a LSH family G with fixed parameters (p_1, p_2, c, τ). Then, we can construct a *new* family G' where each function $g' = (g_1, \ldots, g_k)$ is obtained by stacking the output of k functions sampled from the original family G. This new family has parameters (p_1^k, p_2^k, c, τ) because the probability of a collision now requires k original functions to collide simultaneously.

By using amplification, we can start from an arbitrary LSH family G and obtain a new family for which the probabilities are as low as we like. Specifically, we can pick k large enough to satisfy the conditions above by setting:

$$p_2^k \leq \frac{1}{n} \quad \Rightarrow \quad k = \left\lceil -\frac{\log n}{\log p_2} \right\rceil.$$

The final piece of the puzzle is how to construct an LSH family G. In general, the construction depends on which distance measure $d(k, k')$ one wants to use for retrieval. For example, assume that the keys $k, k' \in \mathbb{R}^D$ are vectors and consider the *cosine distance*:

$$d(k, k') = 1 - \frac{\langle k, k' \rangle}{|k| \cdot |k'|}.$$

Thus the distance between k and k' is zero if the two vectors are aligned. This measure is often used in applications such as text or image retrieval. We can construct an LSH family G for the cosine distance by defining:

$$g_i(k) = \text{sign}\langle v_i, k \rangle$$

where $v_i \in \mathbb{R}^d$ is a uniformly-sampled unit vector. It can be shown[1] that the probability of a collision is inversely proportional to the cosine distance via the formula:

$$P[g_i(k) = g_i(k')] = 1 - \frac{1}{\pi} \arccos(1 - d(k, k')).$$

By plugging values τ and $c\tau$ for the value fo the distance into this formula, wes find that this is a LSH family with parameters (p_1, p_2, c, τ) where

$$p_1 = 1 - \frac{1}{\pi} \arccos(1 - \tau), \quad p_2 = 1 - \frac{1}{\pi} \arccos(1 - c\tau).$$

B.2 Proof of the LSH retrieval theorem (optional)

Parts (1) and (2) are obvious. For case (3), the two following conditions are *together* sufficient for `LSHRetrieve` to return a relaxed match:

1. The first condition is that there is a collision $g_i(q) = g_i(k^*)$ between the query q and the close key k^* for at least one of the functions g_i.

   The probability that a specific g_i results in a collision is at least p_1 by assumption. Thus, the probability that none of the g_i results in a collision is at most $(1 - p_1)^\ell$. Hence, the probability that at least one collision occurs between $g_1, \ldots, g_\ell$ is:
   $$P_1 \geq 1 - (1 - p_1)^\ell.$$

   In this case, `LSHRetrieve` will find and return a key not far from the query provided that it is allowed to check enough collisions. This is because it will eventually check the hash table where the close key k^* is stored. However, the algorithm stops after checking at most m collisions, which may be too soon. The next condition ensures that this is not the case.

2. The second condition is that strictly less than m of the key comparisons that the algorithm can perform are with keys *far* from the query. If this is the case, the algorithm must find and return a key *not far* from the query before hitting the limit of m comparisons, provided that one exits (which is given by the first condition).

   The total number of comparisons with far keys is given by:

   $$M = \sum_{k:d(q,k) \geq c\tau} \sum_{i=1}^{\ell} \mathbf{1}[g_i(q) = g_i(k)]$$

   The expected value of M over the choice of functions g_i is bounded by:

   $$E[M] = \sum_{k:d(q,k) \geq c\tau} \sum_{i=1}^{\ell} E[\mathbf{1}[g_i(q) = g_i(k)] \leq \sum_{k:d(q,k) \geq c\tau} \ell p_2 \leq n\ell p_2$$

   Using Markov inequality $P[X \geq \alpha] \leq E[X]/\alpha$, we can use this expected value to bound the probability that M is at least as large as m:
   $$P[M \geq m] \leq \frac{n\ell p_2}{m}.$$

   Hence, the probability that $M < m$ is:

   $$P_2 = P[M < m] = 1 - P[M \geq m] \geq 1 - \frac{n\ell p_2}{m}.$$

The probability that `LSHRetrieve` fails to find a match is is at most the sum of probabilities that the individual conditions above fail separately, i.e. $P_{\text{fail}} \leq (1 - P_1) + (1 - P_2)$. The probability of success is thus at least

$$P_{\text{success}} \geq 1 - ((1 - P_1) + (1 - P_2)) = 1 - (1 - p_1)^\ell - \frac{n\ell}{m} p_2$$

Q.E.D.

[1] http://www-math.mit.edu/~goemans/PAPERS/maxcut-jacm.pdf

B.3 C++ implementation of an LSH table (optional)

For completeness, we provide an example implementation of an LSH table in C++.

The implementation assumes that keys and queries (of type `data_t`) are float vectors of a given dimension `data_dimension`. The distance function is set to the cosine distance introduced above; likewise, we use the corresponding LSH family, using amplification in order to transform binary hashes into hashes of r bits, encoded as integers of type `code_t`.

The `LSHTable` class builds oon the `HashTable` class from the previous chapter, which take as input codes of type `code_t`. The LSH functions map input vectors `data_t` to corresponding discrete codes `code_t`.

The code benchmarks the implementation by setting the retrieval threshold τ to 0. In this case, the LSH table returns the closest key to a given test query within a set number m of comparison. The idea is to compare the LSH table against the baseline linear search in terms of number of comparisons performed (which provides an upper bound on the speed up) and distance of the retrieved key (which is generally worse than the distance obtained by an exhaustive search).

File `lsh_driver.cpp`:

```
#include <hash.hpp>
#include <utils.hpp>

#include <array>
#include <cmath>
#include <cstdint>
#include <functional>
#include <iomanip>
#include <iostream>
#include <limits>
#include <random>
#include <type_traits>
#include <vector>

constexpr size_t data_dim = 3;
constexpr size_t dataset_size = 10'000;
constexpr size_t queryset_size = 1'000;

using data_t = std::array<double, data_dim>; // keys and queries
using code_t = uint32_t; // codes for the keys, output by the LSH functions

constexpr auto inf =
    std::numeric_limits<data_t::value_type>::infinity(); // shorthand
constexpr auto random_seed = 0;                            // for reproducibility

// Sample a vector on the unit hypersphere.
data_t sample_unit_vector()
{
    static std::mt19937 gen{random_seed};
    static std::normal_distribution<> normal{0, 1};
    data_t vec{};
    data_t::value_type norm2 = 0;
    for (auto &x : vec) {
        x = normal(gen);
        norm2 += x * x;
    }
```

```
37      data_t::value_type norm = sqrt(norm2);
38      for (auto &x : vec) {
39          x /= norm;
40      }
41      return vec;
42  }
43
44  // Sample a collection of unit vectors.
45  std::vector<data_t> sample_dataset(size_t num_data)
46  {
47      std::vector<data_t> dataset;
48      dataset.reserve(num_data);
49      while (num_data--) {
50          dataset.emplace_back(sample_unit_vector());
51      }
52      return dataset;
53  }
54
55  // Cosine distance function between vectors (must be non-zero).
56  data_t::value_type cosine_distance(const data_t &vec1, const data_t &vec2)
57  {
58      data_t::value_type xnorm2 = 0;
59      data_t::value_type ynorm2 = 0;
60      data_t::value_type dot = 0;
61      auto xi = std::begin(vec1);
62      auto yi = std::begin(vec2);
63      while (xi != std::end(vec1)) {
64          auto x = *xi++;
65          auto y = *yi++;
66          xnorm2 += x * x;
67          ynorm2 += y * y;
68          dot += x * y;
69      }
70      auto den = sqrt(xnorm2 * ynorm2);
71      return 1 - dot / den;
72  }
73
74  // A LSH function maps data_t (keys) to code_t (codes)
75  using lsh_function_t = std::function<code_t(const data_t &)>;
76
77  // Sample a binary LSH function of the type f(x) = sign(<w, x>).
78  lsh_function_t sample_lsh_function()
79  {
80      auto weights = sample_unit_vector();
81      return [=](const data_t &vec) -> code_t {
82          data_t::value_type dot = 0;
83          auto wi = std::begin(weights);
84          auto xi = std::begin(vec);
85          while (wi != std::end(weights)) {
86              dot += (*wi++) * (*xi++);
87          }
88          return dot >= 0 ? 1 : 0;
89      };
```

```
}

// Sample a LSH function of the type F(x) = [f1(x), f2(x), ...,
// fr(x)]. The pattern must fit in an integer of type `code_t`.
lsh_function_t sample_amplified_lsh_function(size_t r)
{
    assert(1 <= r && r <= std::numeric_limits<code_t>::digits);
    std::vector<lsh_function_t> functions;
    while (r--) {
        functions.push_back(sample_lsh_function());
    }

    return [=](const data_t &vec) -> code_t {
        code_t code = 0;
        for (const auto &function : functions) {
            code = (code << 1) | function(vec);
        }
        return code;
    };
}

// A class representinig a LSH family.
struct lsh_family_t {
    size_t amplification;

    // Return a function sampled from the LSH faimly.
    lsh_function_t operator()() const
    {
        return sample_amplified_lsh_function(amplification);
    }
};

// A policy specifynig a hash function mappnig codes to slots.
struct hash_policy_t {
    int operator()(code_t code) const { return code; }
};

// A policy specifynig a distance function between data points.
struct distance_policy_t {
    using value_type = data_t::value_type;
    value_type operator()(const data_t &vec1, const data_t &vec2)
    {
        return cosine_distance(vec1, vec2);
    }
};

// The LSH table class.
template <class K, class D, class G, class H> class LSHTable
{
  public:
    // A retrieval result: distance, key and number of key-query comparisons
    // performed.
    struct result_t {
```

```
143         typename D::value_type distance;
144         const K *key;
145         size_t num_comparisons;
146     };
147
148     // Create a LSH table using `num_tables` hash tables, each using
149     // `num_chains` chains. Data are addedd to the hash table encooded via
150     // functions sampled from the `lsh_family`.
151     template <class GF>
152     LSHTable(size_t num_chains, size_t num_tables, GF lsh_family)
153     {
154         while (num_tables--) {
155             tables.push_back(H{num_chains});
156             functions.push_back(lsh_family());
157         }
158     }
159
160     // Add a key to the LSH table.
161     void insert(const K &key)
162     {
163         for (size_t i = 0; i < tables.size(); ++i) {
164             auto code = functions[i](key);
165             auto *retrieved = tables[i].get(code);
166             if (retrieved == nullptr) {
167                 tables[i].insert(code, {key});
168             } else {
169                 retrieved->push_back(key);
170             }
171         }
172     }
173
174     // Search for a key within a distance `tau` of the `query`.
175     // At most `m` key-query comparisons are performed, after which the serach
176     // stops, returning the closetst match found even if the distance is still
177     // larger than `tau`. If no matchnig key can be found
178     // at all, the search stops with a distance equal to infinity.
179     // Setting `tau` to zero can be used to find the closest key to the query
180     // found in the alloted number of comparisons.
181     result_t get(const K &query, int m = 1, data_t::value_type tau = 0)
182     {
183         auto result = result_t{inf, nullptr, 0};
184         auto distance_function = D{};
185         for (size_t i = 0; i < tables.size(); ++i) {
186             auto code = functions[i](query);
187             auto *retrieved = tables[i].get(code);
188             if (retrieved == nullptr) {
189                 continue;
190             }
191             for (const auto &key : *retrieved) {
192                 auto distance = distance_function(key, query);
193                 if (distance < result.distance) {
194                     result.distance = distance;
195                     result.key = &key;
```

```
196                 }
197                 if (++result.num_comparisons >= (unsigned)m) {
198                     goto give_up;
199                 }
200                 if (result.distance <= tau) {
201                     break;
202                 }
203             }
204         }
205     give_up:
206         return result;
207     }
208 
209   private:
210     std::vector<H> tables;
211     std::vector<G> functions;
212 };
213 
214 // Concrete instantiation of a LSH table and corresponding hash table.
215 using hash_table_t = HashTable<code_t, std::vector<data_t>, hash_policy_t>;
216 
217 using lsh_table_t =
218     LSHTable<data_t, distance_policy_t, lsh_function_t, hash_table_t>;
219 
220 using result_t = lsh_table_t::result_t;
221 
222 // Linear search through the dataset.
223 template <class D>
224 result_t naive_retrieve(const std::vector<data_t> &dataset, const data_t &query)
225 {
226     auto result = result_t{inf, nullptr, 1};
227     D distance_function{};
228     for (const auto &key : dataset) {
229         data_t::value_type distance = distance_function(key, query);
230         result.num_comparisons++;
231         if (distance < result.distance) {
232             result.distance = distance;
233             result.key = &key;
234         }
235     }
236     return result;
237 }
238 
239 // Benchmkar a retrieval function `get`.
240 template <typename F>
241 std::tuple<double, double, double> benchmark(const std::vector<data_t> &queries,
242                                              F get)
243 {
244     double d = 0;
245     double d2 = 0;
246     size_t n = 0;
247     size_t nok = 0;
248     for (const auto &query : queries) {
```

```
249         auto distance = get(query);
250         if (std::isfinite(distance)) {
251             d += distance;
252             d2 += distance * distance;
253             nok += 1;
254         }
255         n += 1;
256     }
257     double mean = d / nok;
258     double variance = d2 / nok - mean * mean;
259     return {mean, variance, (double)nok / n};
260 }
261 
262 int main(int argc, char **argv)
263 {
264     auto dataset = sample_dataset(dataset_size);
265     auto queries = sample_dataset(queryset_size);
266 
267 // clang-format off
268 #define print_result(nt, nc, r, sp, mean, stddev, rate, rel) \
269     std::cout \
270     << "| "    << std::setw(7)  << std::left << nt \
271     << " | "   << std::setw(7)  << std::left << nc \
272     << " | "   << std::setw(7)  << std::left << r \
273     << " | "   << std::setw(10) << std::setprecision(3) << std::left << std::defaultfloat << mean \
274     << " "     << std::setw(10) << std::setprecision(3) << std::left << std::defaultfloat << stddev \
275     << " | "   << std::setw(7)  << std::setprecision(1) << std::left << std::defaultfloat << sp \
276     << " | "   << std::setw(6)  << std::setprecision(1) << std::fixed << std::right << rate \
277     << " | "   << std::setw(6)  << std::setprecision(1) << std::fixed << std::right << rel \
278     << "|"     << std::endl;
279 // clang-format oon
280 
281     print_result("#tables", "#comp.", "amplif.", "speedup", "distance", "(stddev)", "succ.", "rel d.");
282     print_result("-", "-", "-", "-", "-", "", "-", "-");
283 
284     // Baseline: scan all keys.
285     double best;
286     {
287         auto result = benchmark(queries, [&](const data_t &query) -> double {
288             return naive_retrieve<distance_policy_t>(dataset, query).distance;
289         });
290         print_result("-", dataset_size, "-", 1, std::get<0>(result),
291                      std::get<1>(result) / sqrt(queryset_size),
292                      std::get<2>(result), 1);
293         best = std::get<0>(result);
294     }
295 
296     // LSH tables.
297     for (size_t num_comparisons : {1, 10, 100, 1000}) {
298         for (size_t num_tables : {1, 2, 3}) {
299             for (size_t amplification : {1, 4, 8, 16, 32}) {
300 
301                 // Build the LSH table.
```

```cpp
                size_t num_chains = std::min(1UL << amplification, 256UL);
                lsh_table_t lsh_table{num_chains, num_tables,
                                      lsh_family_t{amplification}};
                for (const auto &key : dataset) {
                    lsh_table.insert(key);
                }

                // Query the LSH table.
                auto result =
                    benchmark(queries, [&](const data_t &query) -> double {
                        return lsh_table.get(query, num_comparisons, 0)
                            .distance;
                    });

                print_result(
                    num_tables, num_comparisons, amplification,
                    (double)dataset_size / num_comparisons, std::get<0>(result),
                    std::get<1>(result) / sqrt(queryset_size),
                    std::get<2>(result) * 100, std::get<0>(result) / best);
            }
        }
    }

    return 0;
}
```

#tables	#comp.	amplif.	distance (stddev)		speedup	succ.	rel d.
-	-	-	-		-	-	-
-	10000	-	0.000199	1.25e-09	1	1.0	1
1	1	1	0.757	0.00847	1e+04	100.0	3810.7
1	1	4	0.375	0.00368	1e+04	100.0	1888.0
1	1	8	0.13	0.000814	1e+04	100.0	655.7
1	1	16	0.0417	0.000107	1e+04	99.9	210.0
1	1	32	0.0116	8.21e-06	1e+04	99.5	58.2
2	1	1	0.726	0.00793	1e+04	100.0	3654.1
2	1	4	0.374	0.0048	1e+04	100.0	1880.9
2	1	8	0.152	0.00172	1e+04	100.0	765.1
2	1	16	0.0655	0.000343	1e+04	100.0	329.7
2	1	32	0.0132	1.07e-05	1e+04	100.0	66.2
3	1	1	0.626	0.00625	1e+04	100.0	3150.6
3	1	4	0.342	0.00372	1e+04	100.0	1720.2
3	1	8	0.202	0.00132	1e+04	100.0	1014.4
3	1	16	0.0384	0.00011	1e+04	100.0	193.4
3	1	32	0.0106	7.84e-06	1e+04	100.0	53.4
1	10	1	0.123	0.000595	1e+03	100.0	620.1
1	10	4	0.0377	0.00012	1e+03	100.0	189.7
1	10	8	0.0115	8.68e-06	1e+03	100.0	58.0
1	10	16	0.0045	1.44e-06	1e+03	99.6	22.7
1	10	32	0.00145	4.04e-07	1e+03	98.6	7.3
2	10	1	0.0915	0.000221	1e+03	100.0	460.8
2	10	4	0.036	5.16e-05	1e+03	100.0	181.4
2	10	8	0.0197	2.79e-05	1e+03	100.0	99.4
2	10	16	0.00368	7.68e-07	1e+03	100.0	18.5
2	10	32	0.00122	1.52e-07	1e+03	99.9	6.2

```
| 3        | 10       | 1        | 0.111      0.000365   | 1e+03    |   100.0 |   558.4|
| 3        | 10       | 4        | 0.0319     4.99e-05   | 1e+03    |   100.0 |   160.6|
| 3        | 10       | 8        | 0.00973    5.79e-06   | 1e+03    |   100.0 |    49.0|
| 3        | 10       | 16       | 0.00629    5.33e-06   | 1e+03    |   100.0 |    31.6|
| 3        | 10       | 32       | 0.00142    3.85e-07   | 1e+03    |   100.0 |     7.1|
| 1        | 100      | 1        | 0.0104     3.24e-06   | 1e+02    |   100.0 |    52.3|
| 1        | 100      | 4        | 0.0028     3.51e-07   | 1e+02    |   100.0 |    14.1|
| 1        | 100      | 8        | 0.00115    7.7e-08    | 1e+02    |   100.0 |     5.8|
| 1        | 100      | 16       | 0.000353   8.92e-09   | 1e+02    |    99.9 |     1.8|
| 1        | 100      | 32       | 0.000279   7.04e-09   | 1e+02    |    98.7 |     1.4|
| 2        | 100      | 1        | 0.0108     3.8e-06    | 1e+02    |   100.0 |    54.1|
| 2        | 100      | 4        | 0.00261    3.08e-07   | 1e+02    |   100.0 |    13.1|
| 2        | 100      | 8        | 0.00156    1.51e-07   | 1e+02    |   100.0 |     7.8|
| 2        | 100      | 16       | 0.000417   9.95e-09   | 1e+02    |   100.0 |     2.1|
| 2        | 100      | 32       | 0.000216   1.93e-09   | 1e+02    |    99.8 |     1.1|
| 3        | 100      | 1        | 0.0108     4.08e-06   | 1e+02    |   100.0 |    54.3|
| 3        | 100      | 4        | 0.00215    1.57e-07   | 1e+02    |   100.0 |    10.8|
| 3        | 100      | 8        | 0.00116    7.19e-08   | 1e+02    |   100.0 |     5.8|
| 3        | 100      | 16       | 0.000359   7.85e-09   | 1e+02    |   100.0 |     1.8|
| 3        | 100      | 32       | 0.00023    2.06e-09   | 1e+02    |   100.0 |     1.2|
| 1        | 1000     | 1        | 0.00106    3.51e-08   | 1e+01    |   100.0 |     5.3|
| 1        | 1000     | 4        | 0.00031    3.33e-09   | 1e+01    |   100.0 |     1.6|
| 1        | 1000     | 8        | 0.000212   1.61e-09   | 1e+01    |   100.0 |     1.1|
| 1        | 1000     | 16       | 0.000239   2.96e-09   | 1e+01    |   100.0 |     1.2|
| 1        | 1000     | 32       | 0.000275   5.64e-09   | 1e+01    |    98.6 |     1.4|
| 2        | 1000     | 1        | 0.00108    3.77e-08   | 1e+01    |   100.0 |     5.4|
| 2        | 1000     | 4        | 0.000334   3.99e-09   | 1e+01    |   100.0 |     1.7|
| 2        | 1000     | 8        | 0.000221   1.79e-09   | 1e+01    |   100.0 |     1.1|
| 2        | 1000     | 16       | 0.000203   1.39e-09   | 1e+01    |   100.0 |     1.0|
| 2        | 1000     | 32       | 0.000226   4.36e-09   | 1e+01    |   100.0 |     1.1|
| 3        | 1000     | 1        | 0.00108    3.63e-08   | 1e+01    |   100.0 |     5.4|
| 3        | 1000     | 4        | 0.000325   3.98e-09   | 1e+01    |   100.0 |     1.6|
| 3        | 1000     | 8        | 0.0002     1.33e-09   | 1e+01    |   100.0 |     1.0|
| 3        | 1000     | 16       | 0.000199   1.25e-09   | 1e+01    |   100.0 |     1.0|
| 3        | 1000     | 32       | 0.0002     1.3e-09    | 1e+01    |   100.0 |     1.0|
```