B16 Algorithms and Data Structures 1 - Example Sheet

Andrea Vedaldi

Academic Year 2023-24 (version 2.0)

Contents

Introduction

This is the example sheet for B16 Part 3 Algorithms and Data Structures 1. The sheet involves completing and running several C++ programs, for which you should start from the code made available here.

1 Linear-time sorting

1.1 Radix sort

Consider non-negative integers of k binary digits each. We can represent each integer as a binary sequence $\langle c_{k-1} c_{k-2} \cdots c_0 \rangle$, where $c_i \in \{0, 1\}$ where c_{k-1} is the most significant digit.

Informally describe an algorithm that sort a sequence A of n such integers one digit per time, with an overall cost of $O(kn)$.

Optionally, write a C++ implementation `radix_sort.hpp` of this algorithm with an interface similar to `merge_sort.hpp` and test it using the following C++ driver program. Do this only if you have time as it is slightly tricky to get right.

File `radix_sort_driver.cpp`:

```cpp
#include "radix_sort.hpp"
#include "utils.hpp"

int main(int argc, char **argv)
{
    auto A = std::vector<int>{1,  19, 2,  9,  12, 18, 4, 8, 5,  6,
                              17, 10, 11, 14, 16, 15, 7, 3, 13, 20};
    print(A, "Before sorting: ");
    radix_sort(begin(A), end(A));
    print(A, "After sorting: ");
}
```

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

2 Arrays

2.1 Complexity of `ArrayInsert`

What is the Big-O complexity of `ArrayInsert`?

2.2 Writing `ArrayDelete`

Design an algorithm `ArrayDelete` to delete the element at position i in an array A. What is the worst-case complexity of this algorithm?

Implement the algorithm as a C++ template function `void array_delete(std::vector<T>&, size_t)` by completing the file `array_delete.hpp`:

```cpp
#ifndef __array_delete__
#define __array_delete__

#include <cstddef>
```

```cpp
#include <utility>
#include <vector>

template <typename T>
void array_delete(std::vector<T> &A, std::size_t index)
{
    // WRITE YOUR CODE HERE
}

#endif // __array_delete__
```

Use the following test driver `array_delete_driver.cpp` to test your code:

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

int main(int argc, char **argv)
{
    auto A = std::vector<float>{0, 1, 2, 3, 4};
    print(A, "Initially A = ");
    while (!A.empty()) {
        array_delete(A, 0);
        print(A, "After deleting the element at position 0: A = ");
    }
    return 0;
}
```

```
Initially A = [0, 1, 2, 3, 4]
After deleting the element at position 0: A = [1, 2, 3, 4]
After deleting the element at position 0: A = [2, 3, 4]
After deleting the element at position 0: A = [3, 4]
After deleting the element at position 0: A = [4]
After deleting the element at position 0: A = []
```

3 Stacks

3.1 Enhanced stack class

Starting from the class `Stack` given in the notes, create a new stack class `StackEnhanced` with a member function `clear` to remove all elements from the stack. Also write an overload of `operator<<` so that elements can be pushed onto the stack via a notation like `stack << value1 << value2`. Write this into the `stack_enhanced.hpp` file, by completing as needed:

```cpp
#ifndef __stack_enhanced___
#define __stack_enhanced___

#include "stack.hpp"

template <typename T> class StackEnhanced : public Stack<T>
{
  public:
    // Inherit the Stack<T> constructors as they are
    // https://en.cppreference.com/w/cpp/language/using_declaration
    using Stack<T>::Stack;

    void clear()
    {
        // WRITE YOUR CODE HERE
    }
};

template <typename T>
StackEnhanced<T> &operator<<(StackEnhanced<T> &stack, const T &value)
{
    // WRITE YOUR CODE HERE
}

#endif // __stack_enhanced___
```

Use the following test driver to test your code.

File `stack_enhanced_driver.cpp`:

```cpp
#include "stack.hpp"
#include "stack_enhanced.hpp" // Put your code in this file
#include <iostream>

int main(int argc, char **argv)
{
    auto stack = StackEnhanced<int>(100);

    stack << 1 << 2 << 3;
    stack.clear();
    stack << 4 << 5 << 6;

    // Dump the stack content
    std::cout << "Stack content:";
    while (!stack.empty()) {
        std::cout << ' ' << stack.top();
        stack.pop();
    }
    std::cout << '\n';

    return 0;
}
```

```
Stack content: 6 5 4
```

3.2 Reverse Polish calculator

The Reverse Polish Notation (RPN) for arithmetic expression lists first the operands and then the operation. For example, the expression $2 \times (2 + 3)$ is written $2\ 2\ 3 + \times$. In order to evaluate the expression, one reads it from left to right and immediately replaces any operation and its operands with the corresponding result. For instance, the expression above evaluates as:

$$\begin{aligned}&2\ (2\ 3\ +)\ \times\\&2\ 5\ \times\\&10\end{aligned}$$

A stack is the ideal data structure to store the intermediate results as an RPN expression is evaluated, because each operation always involves the last n values computed (i.e., the ones at the top of the stack).

Write functions `plus`, `minus`, `multiplies`, `divides` and `negate` to implement a basic RPN calculator. Each function takes as input a stack containing the state of the calculation and immediately replaces the top one or two elements with the operation result. Use the following test code to check that your code works.

File `stack_rpn_driver.cpp`:

```cpp
#include <iostream>

#include "stack.hpp"
#include "stack_rpn.hpp" // Put your code in this file

int main(int argc, char **argv)
{
    // Basic interface
    auto stack = Stack<int>(100);
    stack.push(2);
    stack.push(2);
    stack.push(3);
    plus(stack);
    multiplies(stack);
    std::cout << "2 2 2 + * = " << stack.top() << '\n';

    // Advanced interface (optional)
    stack << 2 << 2 << 3 << plus << multiplies;
    std::cout << "2 2 2 + * = " << stack.top() << '\n';

    return 0;
}
```

```
2 2 2 + * = 10
2 2 2 + * = 10
```

Optionally, also implement the "advanced interface" used in the driver above, where operand and operations can be entered by using the `<<` symbol (hint: overload `operator<<`).

4 Queues

4.1 Enhanced queue class

Starting from the class `Queue` given in the notes, create a new stack class `Dequeue` (double-ended queue) with:

1. A member function `clear` to remove all elements from the queue.
2. A member function `enqueue_front` to enqueue elements at the front of the queue.
3. A member function `dequeue_back` to remove elements from the front of the queue.
4. A member function `back` to access the first element of the queue.

You can start from the following `queue_enhanced.hpp` file:

```cpp
#ifndef __queue_enhanced__
#define __queue_enhanced__

#include <cassert>
#include <vector>

```

```cpp
#include "queue.hpp"

template <typename T> class Dequeue : public Queue<T>
{
  public:
    // Inherit the constructors of Queue<T>
    using Queue<T>::Queue;

    // Access the element at the back of the queue
    T &back()
    {
        // WRITE YOUR CODE HERE
    }

    // Const-access the element at the back of the queue
    const T &back() const
    {
        // WRITE YOUR CODE HERE
    }

    // Add a new element to the front of the queue by copying
    void enqueue_front(const T &value)
    {
        // WRITE YOUR CODE HERE
    }

    // Remove the element at the back of the queue
    void dequeue_back()
    {
        // WRITE YOUR CODE HERE
    }

    // Remove all elements from the queue
    void clear() {
        // WRITE YOUR CODE HERE
    }

  protected:
    // Return the index of the element at the back of the queue
    size_t _tail() const
    {
        // WRITE YOUR CODE HERE
    }
};

#endif // __queue_enhanced__
```

Use the following test driver to test your code.

File `queue_enhanced_driver.cpp`:

```cpp
#include "queue_enhanced.hpp"
#include "utils.hpp"

#include <iostream>
```

```cpp

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

    // Keep pushing and popping elements from the dequeue for a while
    for (int repetition = 0; repetition < 3; ++repetition) {
        std::cout << "Enqueued front";
        for (int i = 0; i < 3; ++i) {
            queue.enqueue_front(i);
            std::cout << ' ' << i;
        }
        std::cout << "\nDequeued front";
        for (int i = 0; i < 3; ++i) {
            std::cout << ' ' << queue.front();
            queue.dequeue();
        }
        std::cout << "\nEnqueued back";
        for (int i = 0; i < 3; ++i) {
            queue.enqueue(i);
            std::cout << ' ' << i;
        }
        std::cout << "\nDequeued back";
        for (int i = 0; i < 3; ++i) {
            std::cout << ' ' << queue.back();
            queue.dequeue_back();
        }
        std::cout << '\n';
    }

    return 0;
}
```

```
Enqueued front 0 1 2
Dequeued front 2 1 0
Enqueued back 0 1 2
Dequeued back 2 1 0
Enqueued front 0 1 2
Dequeued front 2 1 0
Enqueued back 0 1 2
Dequeued back 2 1 0
Enqueued front 0 1 2
Dequeued front 2 1 0
Enqueued back 0 1 2
Dequeued back 2 1 0
```

5 Lists

5.1 Deleting elements from a list

Starting from the module `list.hpp` given in the notes, write a `list_delete_after` function adding it to a new `list_enhanced.hpp` module, completing the code below:

```cpp

#ifndef __list_enhanced__
#define __list_enhanced__

#include "list.hpp"
#include <iostream>

template <typename T> void list_delete_after(Node<T> *node)
{
    // WRITE YOUR CODE HERE
}

#endif // __list_enhanced__
```

Then test it using the following driver.

File `list_enhanced_driver.cpp`:

```cpp
#include "list_enhanced.hpp"
#include "utils.hpp"

int main(int argc, char **argv)
{
    auto list = Node<float>{};

    // Insert some numbers to the front of the list.
    auto last = &list;
    for (int i = 0; i < 10; ++i) {
        last = list_insert_after(last, static_cast<float>(i));
        print(list_to_vector(list));
    }

    // Remove the elements from the beginning of the list.
    for (int i = 0; i < 10; ++i) {
        list_delete_after(&list);
        print(list_to_vector(list));
    }
}
```

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

```
[7, 8, 9]
[8, 9]
[9]
[]
```

What is the worst-case complexity of this algorithm?

6 Binary trees

6.1 Enhanced `BinaryTree` class

Similar to the `BinaryTree` class given in the notes, write an enhanced binary tree class with a member function `parent()` returning the parent of a node. You can start from the file `binary_tree_enhanced.hpp` file:

```cpp
#ifndef __binary_tree_enhanced__
#define __binary_tree_enhanced__

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

// A class representing a binary tree
template <typename V> struct BinaryTreeEnhanced {
    V _value;
    std::unique_ptr<BinaryTreeEnhanced<V>> _left;
    std::unique_ptr<BinaryTreeEnhanced<V>> _right;

    // WRITE YOUR CODE HERE

    friend V &value(BinaryTreeEnhanced *t) { return t->_value; }
    friend const V &value(const BinaryTreeEnhanced *t)
    {
        return t->_left_value;
    }
    friend BinaryTreeEnhanced *left(const BinaryTreeEnhanced *t)
    {
        return t->_left.get();
    }
    friend BinaryTreeEnhanced *right(const BinaryTreeEnhanced *t)
    {
        return t->_right.get();
    }
    friend BinaryTreeEnhanced *parent(const BinaryTreeEnhanced *t)
    {
        // WRITE YOUR CODE HERE
    }
};

// A helper function to build an enhanced binary tree
template <typename V>
std::unique_ptr<BinaryTreeEnhanced<V>>
make_binary_tree_enhanced(const V &value,
                          std::unique_ptr<BinaryTreeEnhanced<V>> l,
                          std::unique_ptr<BinaryTreeEnhanced<V>> r)
```

```cpp
{
    // WRITE YOUR CODE HERE
}

#endif // __binary_tree_enhanced__
```

Use the following test driver to test it.

File `binary_tree_enhanced_driver.cpp`:

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

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

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

    auto action = [](const auto &tree) {
        auto p = parent(tree);
        if (p) {
            std::cout << "The parent of " << value(tree) << " is "
                      << value(p) << '\n';
        } else {
            std::cout << "Node " << value(tree) << " has no parent\n";
        }
    };

    df_traversal(bt.get(), action);

    return 0;
}
```

```
Tree:
1 -------v
2 -v     3 -v
4  5 -v  6  7
      8
The parent of 4 is 2
The parent of 2 is 1
The parent of 5 is 2
The parent of 8 is 5
Node 1 has no parent
The parent of 6 is 3
The parent of 3 is 1
The parent of 7 is 3
```

6.2 Minimum and maximum element in a binary search tree

Write a module `binary_search_tree_enhanced.hpp` defining functions `bst_min` and `bst_max` computing the minimum and maximum element in a BST so that the following test driver works.

You can start from the interface file:

```cpp
#ifndef __binary_saerch_tree_enhanced__
#define __binary_saerch_tree_enhanced__

#include "binary_search_tree.hpp"

template <typename T> T bst_min(const T &tree)
{
    // WRITE YOUR CODE HERE
}

template <typename T> T bst_max(const T &tree)
{
    // WRITE YOUR CODE HERE
}

#endif // __binary_saerch_tree_enhanced__
```

Test it using the following driver.

File `binary_search_tree_enhanced_driver.cpp`:

```cpp
#include <iostream>

#include "binary_search_tree_enhanced.hpp"
#include "binary_tree_print.hpp"

int main(int argc, char **argv)
{
    std::unique_ptr<BinaryTree<int>> bt;

    for (int x : {12, 5, 18, 2, 9, 15, 19, 13, 17}) {
        bt = bst_insert(std::move(bt), x);
    }

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

    std::cout << "The smallest element is "
              << value(bst_min(bt.get())) << '\n';

    std::cout << "The largest element is " << value(bst_max(bt.get()))
              << '\n';

    return 0;
}
```

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

The smallest element is 2
The largest element is 19
```

6.3 Traversing the nodes of a BST by non-decreasing value

Design an algorithm to traverse the nodes of a BST by non-decreasing value. You can omit the C++ implementation.

7 Heaps

7.1 Building a heap

In the notes, we have discussed the `BuildHeap` algorithm, for building a heap given an initial vector A of values:

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

This algorithm builds the heap starting from the leaves of the binary tree and working its way towards the root. Alternatively, we can build a heap starting from the root and moving towards the leaves. The idea is to progressively include elements A_0, A_1 etc in the heap, which amounts to adding one more leaf to the corresponding binary tree. Each time this is done, one calls `SiftUp` on the last element added to restore the heap property. In pseudo-code:

```
BuildHeapAlt($A$):
  1. For $i = 1, \dots, |A| - 1$:
     1. Interpret the subarray $(A_0, \dots, A_i)$ as a complete binary tree $T$ and let $S$ be the subtree
        rooted at $A_i$.
     2. Call SiftUp($S$).
```

Discuss the relation between `BuildHeapAlt` and `PriorityEnqueue` and compare the complexity of `BuildHeap` and `BuildHeapAlt`.

7.2 Building a heap vs sorting

Discuss whether sorting an array A can be an alternative implementation of `BuildHeap`.

7.3 Updating the priority of a queued element

Discuss how you could use `SiftUp` and `SiftDown` to update the priority of an element in a heap, used as a priority queue.

8 Hashing

8.1 Worst case complexity for multiple chaining

Consider n distinct keys $k_1, \dots, k_n$ drawn from a finite set $\mathcal{K}$, called the *universe*. We wish to store the keys in a hash table that uses m chains. Assume that the universe is large, meaning that $|\mathcal{K}| > m \cdot n$, and show that it is possible to choose the n keys so that they all hash to the same chain. Discuss what is the implication on the worst case complexity of retrieving the keys from the resulting hash table.

8.2 Other usages of hash functions

Consider a file such as a photo or a video. Explain how a hash function can be used to create a compact "signature" of the file to verify its integrity. Specifically, explain how knowledge of the signature can be used to tell with high probability whether the file has been modified without requiring knowledge of the original file. Explain how you would choose the size m of the hash space for this application.

8.3 Division method for large keys

Consider the problem of designing a hash function $h(k)$. The division method consists in computing the remainder $h(k) = k \mod m$ of dividing the key k, regarded as a large natural number, by the number m of chains. This is convenient because $h(k) \in [0, \dots, m-1]$ can be used directly to index a chain.

Suppose that the key is a string $\langle c_{q-1}c_{q-2}\cdots c_0\rangle$ where $c_i \in [0, 255]$ is a character or byte. Except for very short strings ($q \leq 8$), the corresponding number

$$\sum_{i=0}^{q-1} c_{q-1-i} \cdot 256^i$$

does *not* fit in any of the standard atomic data types such as `char`, `short`, `int`, etc. The size of these data types is bounded by the size of the underlying CPU registers, which are thus usually up to 64 bit long. The built-in C++ operator `%` can be used to compute the remainder for atomic data types, this cannot be used directly for numbers wider than 64 bits.

Write a function `uint32_t hash(const std::string& str, const uint32_t m)` that takes as input an arbitrary long string `str` and a divisor `m` and computes

$$h(c) = \left(\sum_{i=0}^{q-1} c_{q-1-i} \cdot 256^i\right) \mod m$$

where c denotes the string `str` and q is the number of characters in the string.

Hint. Make use of the properties of modular arithmetic:

- $a + b \mod m = ((a \mod m) + (b \mod m)) \mod m$
- $ab \mod m = ((a \mod m) \cdot (b \mod m)) \mod m$

8.4 Permutation invariance of the division method

Show that choosing $m = 255$ in the previous question results in hashes that are invariant to permutation of the characters in the string (meaning for example that $h(\text{ciao}) = h(\text{oaic})$). Discuss whether this is a desirable property or not.

9 Graphs

9.1 Shortest paths using the adjacency list representation

Starting from the code given in the notes, write a version of the Bellman-Ford and Dijkstra algorithms that use the adjacency list representation of a graph instead of the adjacency matrix representation, as given in the notes. In particular, implement the following functions.

File `shortest_paths_sparse.hpp`:

```cpp
#ifndef __shortest_paths_sparse__
#define __shortest_paths_sparse__

#include "graph.hpp"

std::vector<hop_t> bellman_ford(const SparseGraph &graph, const int source,
                                bool &has_negative_cycle);

std::vector<hop_t> dijkstra(const SparseGraph &graph, const int source);

#endif // __shortest_paths_sparse__
```

You can start from the following `shortest_paths_sparse.cpp` template:

```cpp
#include "shortest_paths_sparse.hpp"
#include <priority_queue.hpp>

#include <cmath>

std::vector<hop_t> bellman_ford(const SparseGraph &graph, const int source,
                                bool &has_negative_cycle)
{
    const int V = static_cast<int>(graph.size());
    auto DP = std::vector<hop_t>(V, {inf, -1});

    // WRITE YOUR CODE HERE

    return DP;
}

struct triplet_t {
    float d;
    int r;
    int v;
};

std::vector<hop_t> dijkstra(const SparseGraph &graph, const int source)
{
    assert(source >= 0);
    assert(source < (signed)graph.size());

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

    // WRITE YOUR CODE HERE

    return DP;
}
```

Use the following driver to test them.

File `shortest_paths_sparse_driver.cpp`:

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

int main(int argc, const char *argv[])
```

```cpp
{
    auto graph = sparse_test_graph;
    print_graph(graph);

    {
        int source = 2;
        bool has_negative_cycle;
        std::cout << "Bellman-Ford from source " << source
                  << std::endl;
        auto DP = bellman_ford(graph, source, has_negative_cycle);
        print(DP);
        std::cout << std::endl;
    }

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

    return 0;
}
```

Are these implementations better for sparse graph?

9.2 Decoding shortest paths

Our shortest paths algorithms encode the shortest paths as a pair of vectors (or matrices) D and P, both coded in a single vector `DP` of type `std::vector<hop_t>`, where each `hop_t` element contains a pair (D_v, P_v).

Write a module `shortest_path_decode.hpp` defining a function:

```cpp
std::vector<int> decode(const std::vector<hop_t> &DP, int v);
```

You can start from the following `shortest_paths_decode.hpp` template:

```cpp
#ifndef __shortest_paths_decode__
#define __shortest_paths_decode__

#include "graph.hpp"
#include <algorithm>
#include <vector>

inline std::vector<int> decode(const std::vector<hop_t> &DP, int v)
{
    // WRITE YOUR CODE HERE
}

#endif // __shortest_paths_decode__
```

that takes `DP` and a destination vertex `v` index and returns the path from the source to the destination as a vector of vertex indices. Use the following test driver to test it.

File `shortest_paths_fw_decode_driver.cpp`:

```cpp

#include <algorithm>
#include <iostream>
#include <vector>

#include "shortest_paths_decode.hpp"
#include "shortest_paths_fw.hpp"
#include <utils.hpp>

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

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

    for (int u = 0; u < (signed)graph.size(); ++u) {
        for (int v = 0; v < (signed)graph.size(); ++v) {
            auto path = decode(DP[u], v);
            if (path.size()) {
                std::cout << "Shortest path " << u << " ~~> " << v
                          << " (weight " << DP[u][v].weight << "): ";
                print(path);
            }
        }
    };

    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 ASPS
[(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)]

Shortest path 0 ~~> 1 (weight 4): [0, 1]
Shortest path 0 ~~> 7 (weight 8): [0, 7]
Shortest path 0 ~~> 8 (weight 15): [0, 7, 8]
Shortest path 1 ~~> 7 (weight 11): [1, 7]
Shortest path 1 ~~> 8 (weight 18): [1, 7, 8]
Shortest path 2 ~~> 3 (weight 9): [2, 5, 6, 3]
Shortest path 2 ~~> 4 (weight 18): [2, 5, 6, 3, 4]
Shortest path 2 ~~> 5 (weight 4): [2, 5]
Shortest path 2 ~~> 6 (weight 6): [2, 5, 6]
Shortest path 2 ~~> 7 (weight 7): [2, 5, 6, 7]
Shortest path 2 ~~> 8 (weight 2): [2, 8]
Shortest path 3 ~~> 4 (weight 9): [3, 4]
Shortest path 3 ~~> 5 (weight 14): [3, 5]
Shortest path 3 ~~> 6 (weight 16): [3, 5, 6]
Shortest path 3 ~~> 7 (weight 17): [3, 5, 6, 7]
Shortest path 3 ~~> 8 (weight 22): [3, 5, 6, 8]
Shortest path 4 ~~> 3 (weight 15): [4, 5, 6, 3]
Shortest path 4 ~~> 5 (weight 10): [4, 5]
Shortest path 4 ~~> 6 (weight 12): [4, 5, 6]
Shortest path 4 ~~> 7 (weight 13): [4, 5, 6, 7]
Shortest path 4 ~~> 8 (weight 18): [4, 5, 6, 8]
Shortest path 5 ~~> 3 (weight 5): [5, 6, 3]
Shortest path 5 ~~> 4 (weight 14): [5, 6, 3, 4]
Shortest path 5 ~~> 6 (weight 2): [5, 6]
Shortest path 5 ~~> 7 (weight 3): [5, 6, 7]
Shortest path 5 ~~> 8 (weight 8): [5, 6, 8]
Shortest path 6 ~~> 3 (weight 3): [6, 3]
Shortest path 6 ~~> 4 (weight 12): [6, 3, 4]
Shortest path 6 ~~> 5 (weight 17): [6, 3, 5]
Shortest path 6 ~~> 7 (weight 1): [6, 7]
Shortest path 6 ~~> 8 (weight 6): [6, 8]
Shortest path 7 ~~> 8 (weight 7): [7, 8]
```

10 Further examples (optional)

All these examples are **optional**. Do them if you want to develop a better understanding of certain advanced features of the C++ language.

10.1 Inserting movable objects in an array

The implementation of `array_insert` given in the notes is quite inefficient as it requires copying elements in the array. While the complexity of the algorithm cannot be improved without switching to a different data structure, we can at least make the implementation faster by moving instead of copying the array elements.

Write a function

```
template <typename T>
void array_insert_movable(std::vector<T>& A, size_t index, T&& x);
```

that uses move operations instead of copy operations whenever possible.

In order to test this function, you can use the `Movable` type provided in `probes.hpp`. This class prints messages whenever different constructors are invoked, which allows you to track copy and move operation explicitly, as follows.

File `probes.hpp`:

```
#ifndef __probes_hpp__
#define __probes_hpp__

#include <iostream>

// A test class with copy and move constructor and assignment
struct Movable {
    float x;

    Movable() : x{0}
    {
        std::cout << "Movable: Default-constructed" << '\n';
    }

    Movable(float x) : x{x}
    {
        std::cout << "Movable: Constructed from \"" << x << "\"\n";
    }

    Movable(const Movable& m) : x{m.x}
    {
        std::cout << "Movable: Copy-constructed" << '\n';
    }

    Movable(Movable&& m) noexcept : x{m.x}
    {
        m.x = 0;
        std::cout << "Movable: Move-constructed\n";
    }

    Movable& operator=(const Movable& m)
    {
        x = m.x;
        std::cout << "Movable: Copy-assigned\n";
        return *this;
    }

    Movable& operator=(Movable&& m)
    {
        x = m.x;
        m.x = 0;
        std::cout << "Movable: Move-assigned\n";
        return *this;
    }
```

```cpp

    ~Movable() { std::cout << "Movable: Destructed\n"; }
};

// A test class with copy constructor and assignment only
struct Copyable {
    float x;

    Copyable() : x{0}
    {
        std::cout << "Copyable: Default-constructed\n";
    }

    Copyable(float x) : x{x}
    {
        std::cout << "Copyable: Constructed from \"" << x << "\"\n";
    }

    Copyable(const Copyable& m) : x{m.x}
    {
        std::cout << "Copyable: Copy-constructed\n";
    }

    Copyable& operator=(const Copyable& m)
    {
        x = m.x;
        std::cout << "Copyable: Copy-assigned\n";
        return *this;
    }

    ~Copyable() { std::cout << "Copyable: Destructed\n"; }
};

inline std::ostream& operator<<(std::ostream& os, const Movable& m)
{
    return os << m.x;
}

inline std::ostream& operator<<(std::ostream& os, const Copyable& c)
{
    return os << c.x;
}

#endif  // __probes_hpp__
```

Test `array_insert_movable` with the following code:

```cpp
#include "array.hpp"
#include "array_insert_movable.hpp"
#include "utils.hpp"
#include "probes.hpp"

std::vector<Movable> make_test_array()
{
    std::cout << "Constructing an array of Movable objects\n";
    auto A = std::vector<Movable>{};
    A.reserve(10);
    A.push_back(Movable{1});
    A.push_back(Movable{2});
    A.push_back(Movable{3});
    print(A, "Constructed array: ");
    std::cout << "\n";
    return A;
}

int main(int argc, char** argv)
{
    {
        auto A = make_test_array();
        auto x = Movable{-1};
        std::cout << "Insertinig object by copy:" << '\n';
        array_insert(A, 0, x);
        print(A, "Array content: ");
    }

    std::cout << "\n";

    {
        auto A = make_test_array();
        auto x = Movable{-1};
        std::cout << "Insertinig object by copying and moving:" << '\n';
        array_insert_movable(A, 0, x);
        print(A, "Array content: ");
    }

    std::cout << "\n";

    {
        auto A = make_test_array();
        auto x = Movable{-1};
        std::cout << "Insertinig object by moving:" << '\n';
        array_insert_movable(A, 0, std::move(x));
        print(A, "Array content: ");
    }

    return 0;
}
```

```
Constructing an array of Movable objects
Movable: Constructed from "1"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "2"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "3"
Movable: Move-constructed
Movable: Destructed
```

```
Constructed array: [1, 2, 3]

Movable: Constructed from "-1"
Insertinig object by copy:
Movable: Copy-constructed
Movable: Copy-assigned
Movable: Copy-assigned
Movable: Copy-assigned
Array content: [-1, 1, 2, 3]
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed

Constructing an array of Movable objects
Movable: Constructed from "1"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "2"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "3"
Movable: Move-constructed
Movable: Destructed
Constructed array: [1, 2, 3]

Movable: Constructed from "-1"
Insertinig object by copying and moving:
Movable: Move-constructed
Movable: Move-assigned
Movable: Move-assigned
Movable: Copy-assigned
Array content: [-1, 1, 2, 3]
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed

Constructing an array of Movable objects
Movable: Constructed from "1"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "2"
Movable: Move-constructed
Movable: Destructed
Movable: Constructed from "3"
Movable: Move-constructed
Movable: Destructed
Constructed array: [1, 2, 3]

Movable: Constructed from "-1"
Insertinig object by moving:
```

```
Movable: Move-constructed
Movable: Move-assigned
Movable: Move-assigned
Movable: Move-assigned
Array content: [-1, 1, 2, 3]
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed
Movable: Destructed
```

The code test three cases: using `array_insert` from the notes, using `array_insert_movable` without actually moving the new element `x` (which causes it to be copied), and finally using `array_insert_movable` to move `x` into the array.

If you implemented `array_insert_movable`, from the printout above you should notice that only the move constructors/assignments of `Movable` are used in the last case.

10.2 Stack with move semantics

In the notes, the `Stack` class has two implementations of the `push` method: one version takes as input a lvalue reference to the value to be pushed, which internally creates a copy of the element. The other version takes a rvalue reference, which supports moving rather than copying. Besides being potentially faster, moving is required if you use automatic memory management using `unique_ptr`: by moving, you transfer ownership of objects to and from the stack.

This is demonstrated in the following driver.

File `stack_move_driver.cpp`:

```cpp
#include <iostream>
#include <memory>

#include <stack.hpp>

template <typename T> class StackWithMove : public Stack<T>
{
    using Stack<T>::Stack;

  public:
    // Move a value to the top of the stack (optional)
    void push(T &&x)
    {
        assert(this->_head < this->_storage.size());
        this->_storage[this->_head++] = std::move(x);
    }
};

int main(int argc, char** argv)
{
    auto stack = StackWithMove<std::unique_ptr<float>>(10);

    // Create a new `float` object
    auto data = std::make_unique<float>(1);
    std::cout << "`data` points to " << data.get() << std::endl;

    // Push the oobject on the stack by transferring ownership to it
    stack.push(std::move(data));
    std::cout << "`data` points to " << data.get() << std::endl;

    // Retrieve the object and its ownership
    auto popped = std::move(stack.top());
    stack.pop();
    std::cout << "`popped` points to " << popped.get() << std::endl;

    // The object will be automaticallly deleted
    // by exting the function as the variable `retrieved`, which owns
    // it, falls out of scope (RAII idiom)
    return 0;
}
```

```
`data` points to 0x7f87d8405880
`data` points to 0x0
`popped` points to 0x7f87d8405880
```

Explain the output of `stack_move_driver`.

10.3 List iterators

In the examples so far we have "manually" iterated over a linked list by using pointers. Iterators are a concept used by the C++ standard library to abstract operations involving such pointers.

By implementing iterators, we can iterate over the list in a much nicer way. The following test driver shows how this can be done using iterators directly (`cbeing(list)`) and using a range-based for loop (`x : list`). It also shows the power of template by passing the list to our own `print` helper function defined in `utils.hpp`: the latter works because, internally, it is based on a range-based for loop that our list container now supports.

```cpp
#include <iostream>

#include "list.hpp"
#include "list_iterator.hpp"
#include "utils.hpp"

int main(int argc, char** argv)
{
    auto list = Node<float>{};

    // Insert some numbers in the list
    for (int i = 0; i < 10; ++i) {
        list_insert_after(&list, static_cast<float>(i));
    }

    // Traverse the list using a const iterator
    for (auto iter = cbegin(list); iter != cend(list); ++iter) {
        std::cout << *iter << ' ';
    }
    std::cout << std::endl;

    // Traverse the list using a range for loop
    for (const auto& x : list) {
        std::cout << x << ' ';
    }
    std::cout << std::endl;

    // Traverse the list using our own `print` helper function from
    // `utils.hpp`
    print(list, "List content: ");

    return 0;
}
```

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

Write a STL-compatible iterator for the `Node<T>` list class.