B16 Software Engineering

Algorithms and Data Structures 1

Lecture 1 of 4: Recap on complexity, quasilinear and linear sort, elementary data structures (arrays, stacks, queues, linked lists)

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Module content & resources

Learning objectives

- Elementary data structures: arrays, stacks, queues, linked lists
- Binary Trees
- Binary Search Trees
- Heaps
- Priority Queues
- Hashing
- Graphs
- Shortest paths

Materials

Slides, Notes, and Examples

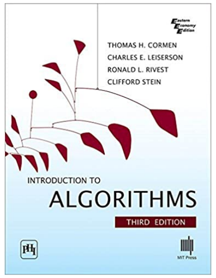
- https://www.robots.ox.ac.uk/~vedaldi/teach.html

Source code for the Examples

- https://github.com/vedaldi/b16-code

Feedback Form

Reference text



Introduction to Algorithms, 3rd Edition. Cormen, Leiserson, Rivest, Stein. McGraw-Hill, 1990.

B16 Software Engineering

Algorithms and Data Structures 1

Part 1 of 4: Recap on complexity, quasilinear and linear sort, elementary data structures (arrays, stacks, queues, linked lists)

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Problem

A **problem** is a description of the input data, the output data, and the relationship between them.

Algorithm

An **algorithm** is a description of certain computational steps that generate the output data from the input data, thus solving the problem.

Sorting problem [revision]

Problem definition

- **Input**: A sequence $A = (A_0, A_1, \ldots, A_{n-1})$
- **Output**: The same sequence, but permuted so that

$$A_{i-1} \le A_i \quad \text{for} \quad i = 1,\ldots,n-1$$

Problem instance

- **Input**: $A = (5,4,3,2,1)$
- **Output**: $A = (1,2,3,4,5)$

Merge Sort [revision]

MergeSort(A):

- **Precondition**: A is an array
- **Postcondition**: A has the same element as before, but permuted in non-decreasing order

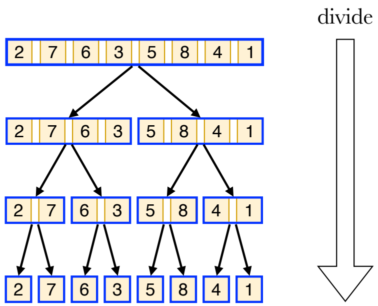
1. If $|A| = 1$, return
2. Let $i \leftarrow \lfloor |A|/2 \rfloor$
3. Let $B \leftarrow (A_0, \ldots, A_{i-1})$
4. Let $C \leftarrow (A_i, \ldots, A_{|A|-1})$
5. Call MergeSort(B)
6. Call MergeSort(C)
7. Set $A \leftarrow$ Merge(B, C)

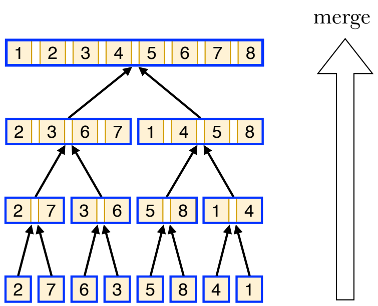
Merge(B, C):

- **Precondition**: arrays B and C are sorted
- **Postcondition**: return an array A which is the non-decreasing union of arrays B and C

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 $j < |C|$:
   3.1. If $B_i \le C_j$:
      3.1.1. Set $A_{i+j} \leftarrow B_i$ and $i \leftarrow i + 1$
   3.2. Else:
      3.2.1. Set $A_{i+j} \leftarrow C_j$ and $j \leftarrow j + 1$
4. While $i < |B|$:
   4.1. Set $A_{i+j} \leftarrow B_i$ and $i \leftarrow i + 1$
5. While $j < |C|$:
   5.1. Set $A_{i+j} \leftarrow C_j$ and $j \leftarrow j + 1$
6. Return A

Merge Sort: example [revision]





Complexity [revision]

The goal of complexity is to analyse the speed of an algorithm

Let n be a parameter characterising the **size of the input**

We study the **number of computational steps** $f(n)$ that an algorithm requires to solve the problem

Worst-case complexity

$f(n)$ is the largest possible number of steps to solve any problem instance of size n

Average-case complexity

$f(n)$ is the average possible number of steps to solve "random" problem instances of size n

This requires defining a probability distribution over problem instances

Complexity [revision]

Big-O notation

We say that $f(n)$ is **Big-O** of $g(n)$ iff there are constant n_0, a such that

$\forall n \geq n_0 : f(n) \leq ag(n)$

Big-Ω notation

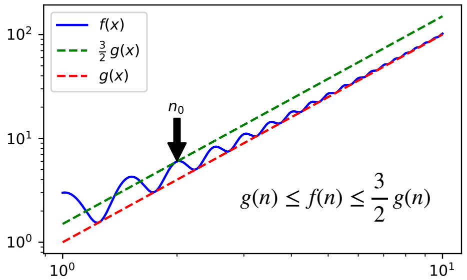
We say that $f(n)$ is **Big-Ω** of $g(n)$ iff there are constant n_0, a such that

$\forall n \geq n_0 : f(n) \geq ag(n)$

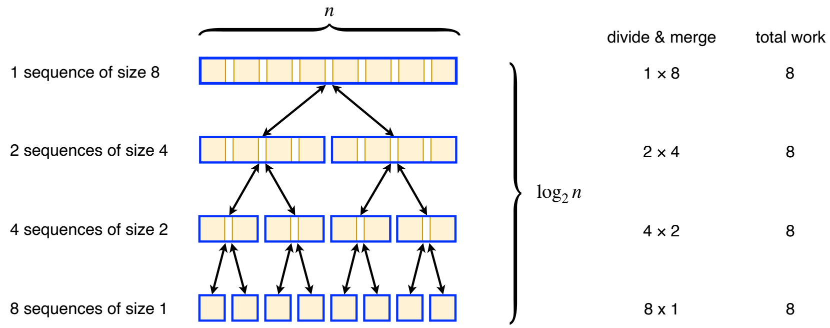
Big-Θ notation

We say that $f(n)$ if **Big-Θ** of $g(n)$ iff it is simultaneously Big-O and Big-Ω of $g(n)$

$f(n) = n^2 + \cos(4\pi n) + 1 \qquad g(n) = n^2$



Merge Sort: work done [revision]



$O(n \log n)$

Merge Sort: complexity [revision]

Recurrence relation

Merge Sort called on a sequence of length $n = |A|$:

- Calls itself recursively on sequences of size $n/2$
- Merges the resulting sorted subsequences in n steps

The total number of steps is thus given by the following recurrence relation:

- $f(n) = 2f(n/2) + n$
- $f(1) = 1$

Solution of the recurrence relation

The solution of of the recurrence equations is

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

(homework: verify this expression)

Conclusion: Merge Sort is $O(n \log n)$

How fast can you sort?

Sorting using comparisons

Algorithm $\mathcal{S}(A)$ only observes the input sequence A by the results of **pairwise comparisons** $A_i < A_j$

It then outputs a **permutation** of the sequence A which sorts it

A counting argument

There are $n!$ possible permutations A of the sequence $(1, 2, \ldots, n)$

As A varies, the algorithm $\mathcal{S}(A)$ must eventually output $n!$ different permutations

If $\mathcal{S}(A)$ performs only t comparisons, it can only output 2^t possible permutations

Hence, we must have $2^t \geq n!$

How fast can you sort?

A counting argument (/ctd)

We thus have the following bound:

$$2^{f(n)} \geq n! = \underbrace{n(n-1)\cdots(n/2)}_{n/2 \text{ terms}}(n/2-1)\cdots 2 \cdot 1 \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Hence:

$$f(n) \geq \frac{n}{2}\log_2\frac{n}{2} \quad \Rightarrow \quad f(n) \in \Omega(n \log n)$$

Lower bound on complexity

No sorting algorithm based on pairwise comparisons can be faster than $\Omega(n \log n)$

Sorting faster than $n \log n$

Sorting faster is possible under **additional assumptions**. For example:

Assumption: the input sequence A consists of natural numbers A_i in the range 0 to $k-1$

CountingSort(A, k):

1. Allocate an array C with k elements initialised to 0 — k steps
2. For $i = 0, \ldots, |A|-1$: — n steps
   - 2.1. Set $C_{A_i} \leftarrow C_{A_i} + 1$
3. Let $i \leftarrow 0$ and $j \leftarrow 0$
4. While $j < k$:
   - 4.1. If $C_j = 0$, then set $j \leftarrow j+1$ and continue with line 4 — at most k times
   - 4.2. Set $A_i \leftarrow j$
   - 4.3. Set $C_j \leftarrow C_j - 1$
   - 4.4. Set $i \leftarrow i + 1$ — at most n times

Complexity: $\Theta(n+k)$

Data structures

A **data structure** is a container that arranges data in such a way that certain operations can be implemented efficiently

Today we will look at:

- Arrays
- Stacks
- Queues
- Linked lists

In the rest off the course we will look at:

- Binary trees
- Heaps
- Priority queues
- Hashes
- Graphs

Arrays

An **array** A is a map from indices $0, \ldots, n-1$ to elements $A_0, \ldots, A_{n-1}$ that allows fast access to any of the elements

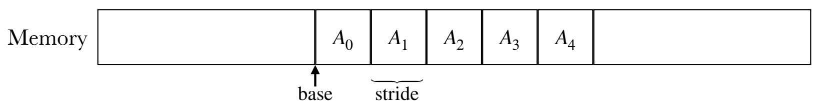
This means that reading or writing any element A_i is a $\Theta(1)$ operation

Typical implementation of an array

An array is implemented by storing elements at equally-spaced memory locations

Then the address of element A_i is computed in $\Theta(1)$ time as `base` $+ i$ `stride` for any value of the index i

In a RAM machine, accessing an element by its address is a $\Theta(1)$ operation



Array insert

While random access with an array is fast, other operations such as inserting a new element at an arbitrary position are *not*

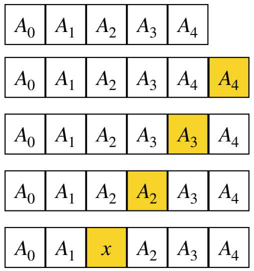
ArrayInsert(A, i, x):

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

1. For $j = n, \ldots, i+1$:
   1.1. Set $A_j \leftarrow A_{j-1}$
2. Set $A_i \leftarrow x$

The complexity is $O(n)$ (why?)

Example: ArrayInsert$(A, x, 2)$



Array Insert: C++ implementation


```
#ifndef __array__
#define __array__

#include <vector>

template <typename T>
void array_insert(std::vector<T>& A, size_t index, const T& x)
{
    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__
```

template allows generic type T for the elements (int, string, …)

array implemented as a `std::vector`

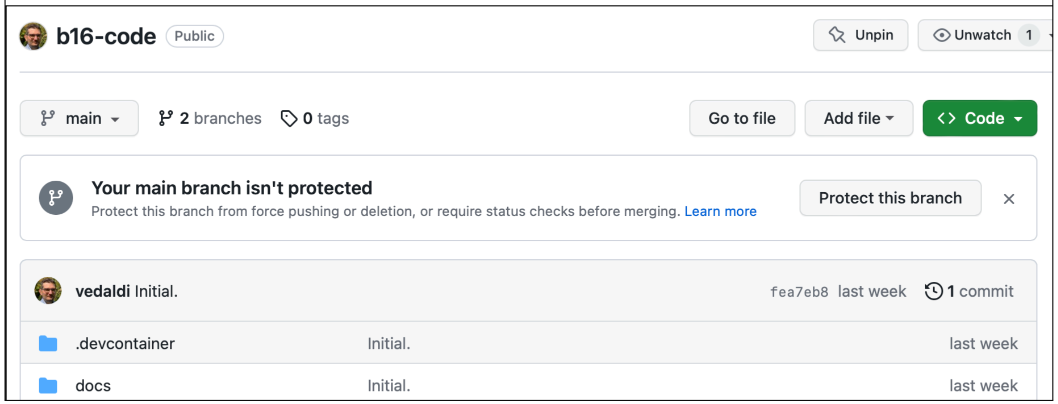
for debugging: raise an error if called with an illegal index

special case: insert the element as last

Try the code for yourself!

The course source code for the lectures and examples is available here

https://github.com/vedaldi/b16-code

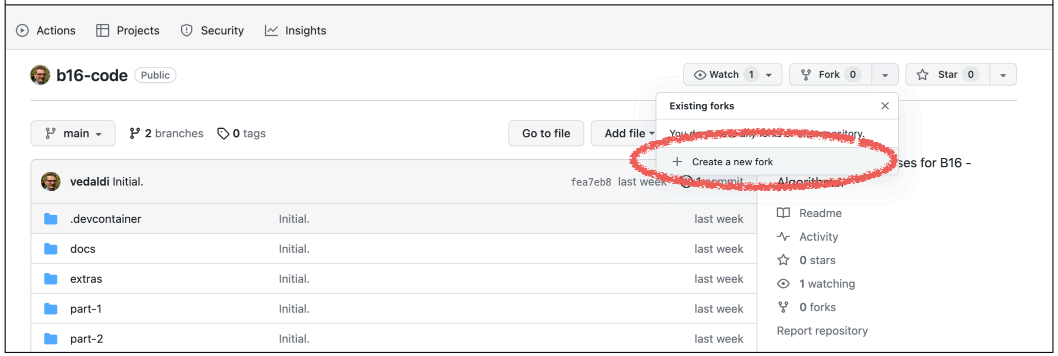


First, fork the B16 code repository

Create a GitHub user (optionally enrol in GitHub Education) and log in

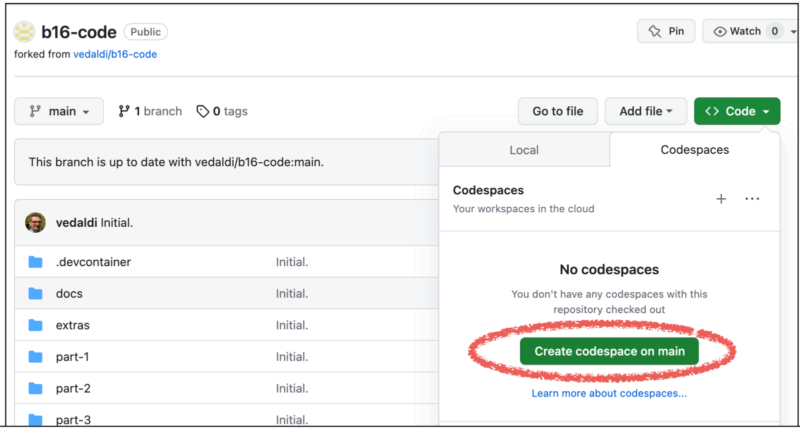
Go to https://github.com/vedaldi/b16-code

Select `Fork` > `+ Create a new fork`



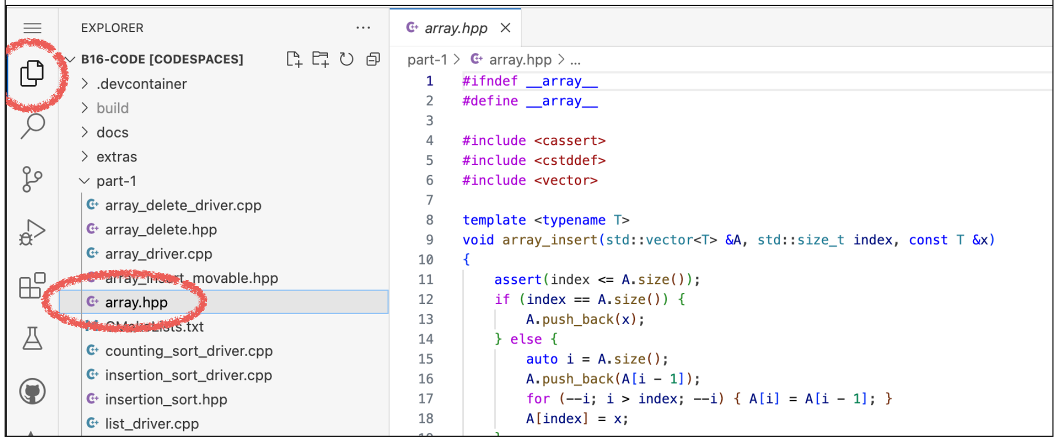
Second, start a GitHub Codespace

Select `Code` > `Create codespace on main`



Edit the code using VS Code in the virtual machine

Select `B16-Code` > `part-1` > `array.hpp`



Build any of the provided programs (but the exercises are incomplete)

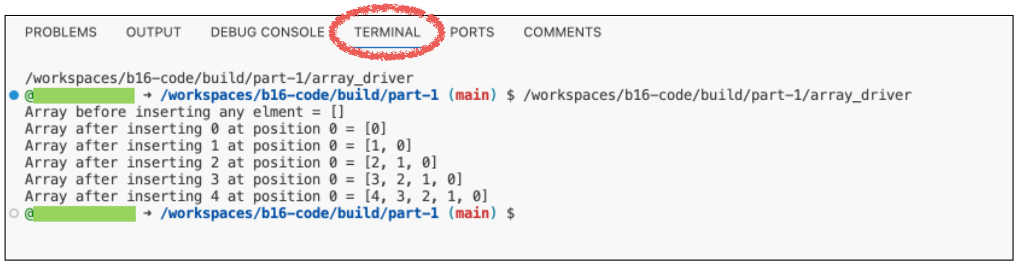
Press `[All]` next to `Build` at the bottom of the screen and select `[array_driver]`

Press `Build`

You can now execute the program

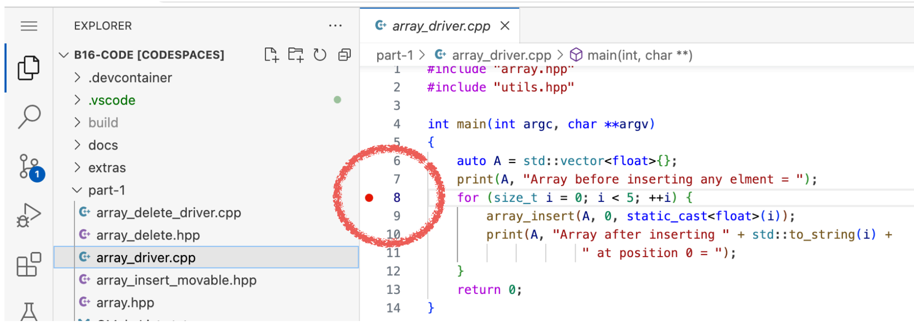
Press the ▶ button and select `[array_driver]`

This will run the code in a terminal, which allows you to see the output



You can debug the program

Add a breakpoint to the code by clicking to the left of any line number

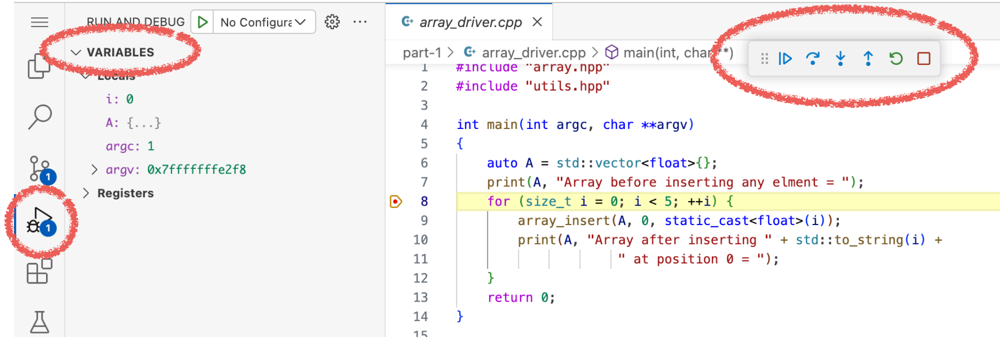


Press the debug button in the bottom bar

You can step through the code and observe the variables

Use the `Variables` watch to observe the variables

Use the stepping controls to execute one line of the program at a time

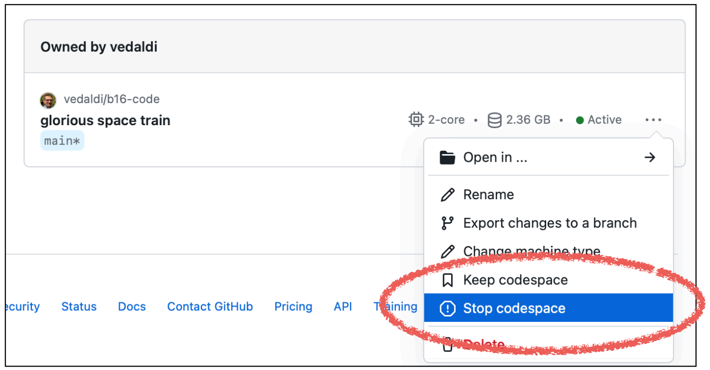


Once you are done, do not forget to stop the codespace

Codespace can only be used for 60 hours per month (90 with the Education account)

Go to https://github.com/codespaces

Select … > `Stop codespace`



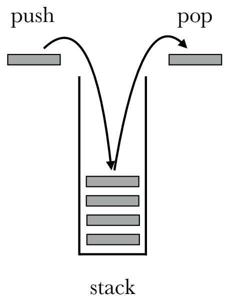
Stacks

A **stack** S is a sequence of elements that allow fast storage and retrieval at one end

Also known as a LIFO (last in, first out) data structure

This means that there are two efficient $\Theta(1)$ operations:

1. **Pushing** a new element x on the "top" of S
2. **Popping** the element at the "top" of S



Stack push and pop

We implement a stack via a **structure** S with fields:

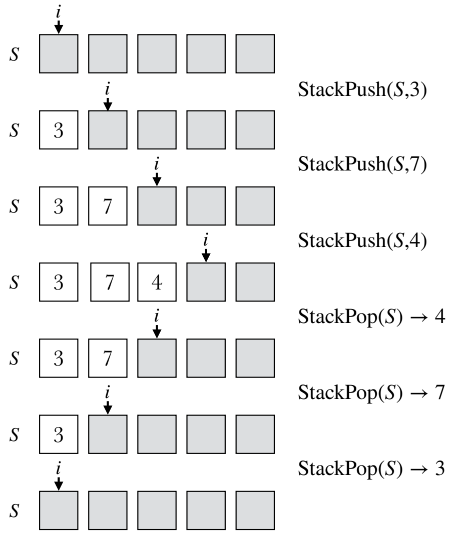
- $S.A$ a pre-allocated array with space for n elements
- $S.i$ the index pointing to the **head** of the stack

StackPush(S, x):

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

StackPop(S):

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



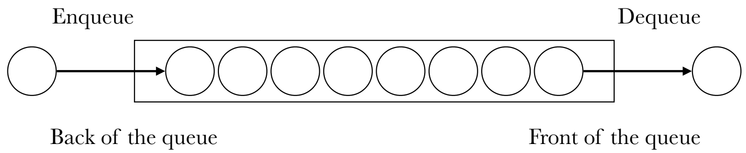
Queues

A **queue** Q is a sequence of elements that allows quickly adding elements from one end and removing them from the other

A queue is also known as a FIFO (first in, first out) data structure

This means that there are two efficient $\Theta(1)$ operations:

1. **Enqueuing** a new element x at the back of Q
2. **Dequeuing** the element at the front of Q



Enqueue and dequeue

We implement a queue via a *structure* Q with fields:

- $Q.A$ a pre-allocated array
- $Q.i$ index of predecessor of the queue back
- $Q.n$ number of enqueued elements

We arrange the array A in a *ring buffer,* storing elements in a "circular" manner

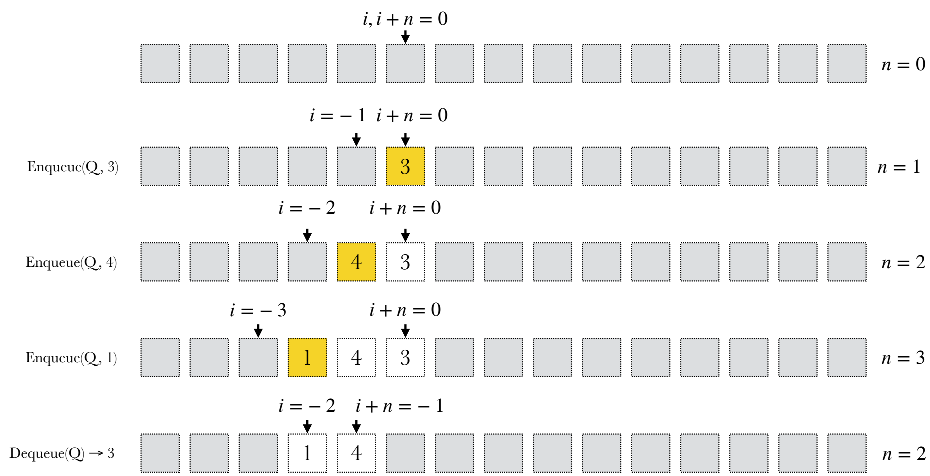
Enqueue(Q, x):

1. $Q.A_i \leftarrow x$
2. $Q.n \leftarrow Q.n + 1$
3. $Q.i \leftarrow |A| - 1$
4. If $Q.i = 0$:
   4.1. $Q.i \leftarrow Q.i - 1$

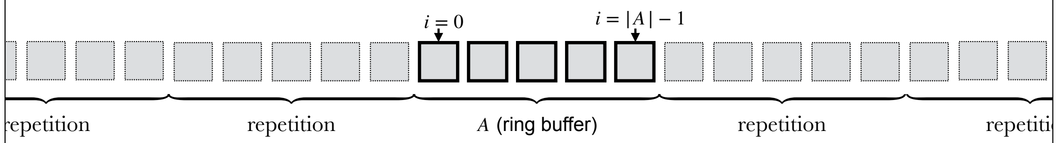
Dequeue(Q):

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

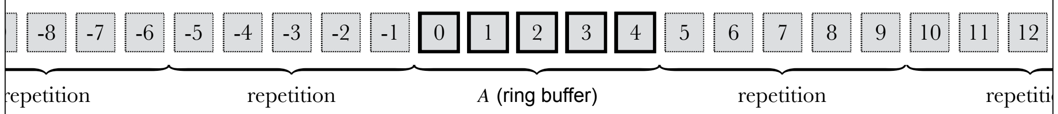
Queue: logical implementation using an infinite buffer



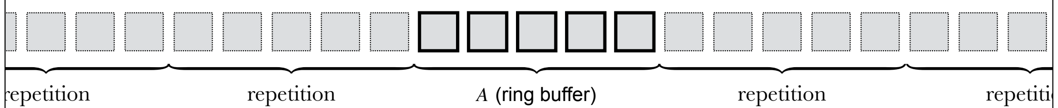
Queue: "physical" implementation using a ring buffer



Ring buffer

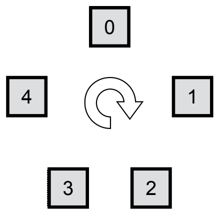


Ring buffer

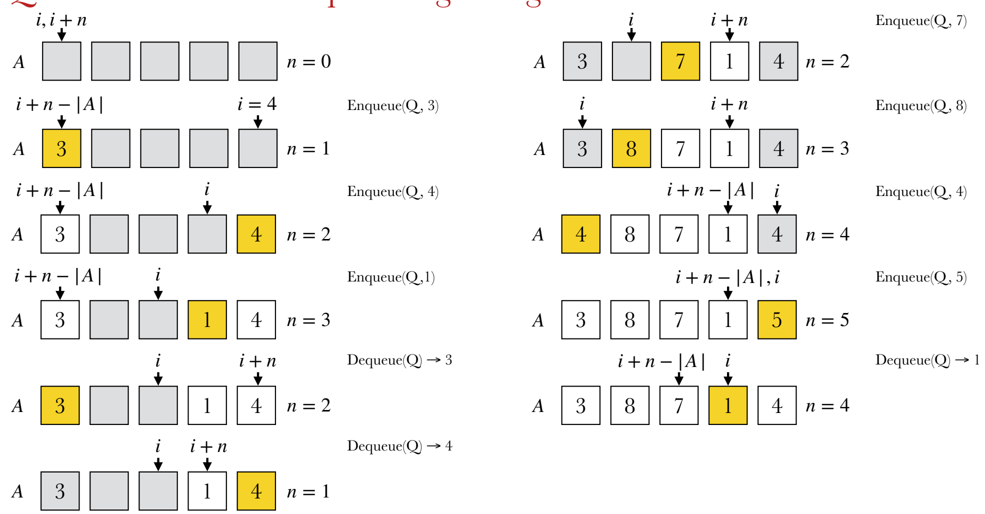


The ring buffer views a finite array A as an infinite one

This works correctly as long as the part of the infinite array which is utilised is contiguous and of size at most $|A|$



Queue: extended example using a ring buffer



Linked lists

A **linked list** L represents a sequence of elements, similarly to an array

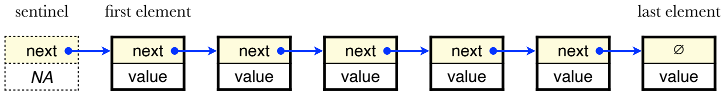
Differently from an array, a linked list does not support fast random access to its element, but can significantly accelerate other operations such as insertion

The linked list is given by a chain of **nodes** N

Each node N is a structure with fields:

- N . value value associated to the node
- N . next next node in the chain

We use a fake **sentinel node** as a “pointer” to the first element in the list

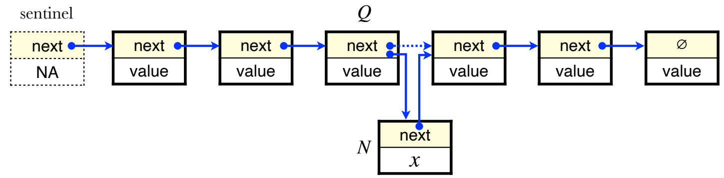


Linked lists: insertion

Inserting a new node in a linked list is done in time Θ(1) via simple pointer operations

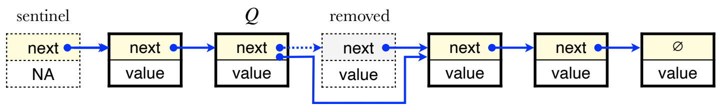
ListInsertAfter(Q, x) :

1. Create a new node N
2. Set N . next ← Q . next
3. Set N . value ← x
4. Set Q . next ← N



Linked lists: removal

ListRemoveAfter(Q) is similar to ListInsertAfter(Q), and is left as an exercise



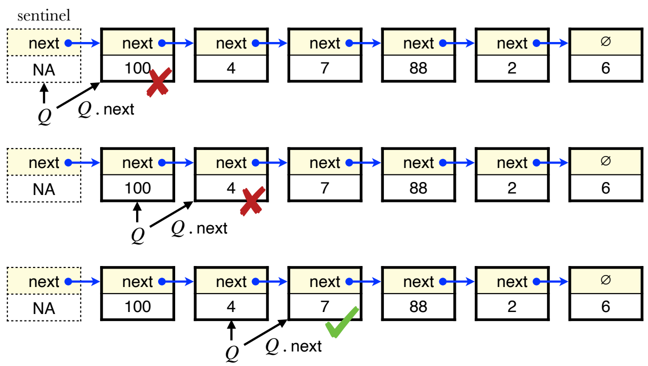
Linked lists: value-based search

Searching for a node with a given value requires scanning the list in $O(n)$ time

ListFindPredecessor(Q, x) :

1. While Q and Q . next are not NIL:
   1.1. If Q . next . value = x return Q
   1.2. Set Q ← Q . next
2. Return NIL

ListFindPredecessor(Q,7)



B16 Software Engineering

Algorithms and Data Structures 1

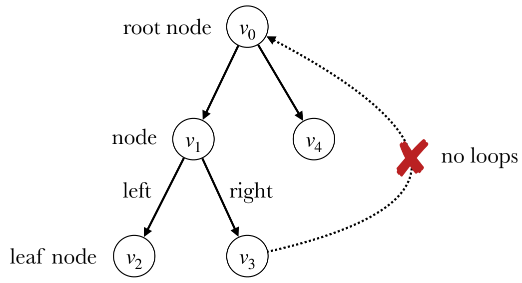
Part 2 of 4: Binary tree and heaps

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Binary trees

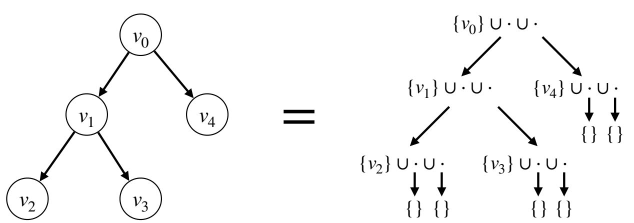
Informally, a **binary tree** is a collection of nodes,
each of which can have a left child and a right child, without loops



Binary trees: formal definition

A **binary tree** T is a finite set such that:

- $T = \{\}$ is the empty set, *or*
- $T = \{r\} \cup L \cup R$ is the union of three disjoint sets:
  - the **root** $\{r\}$
  - the **left child** L, which is also a binary tree
  - the **right child** R, which is also a binary tree



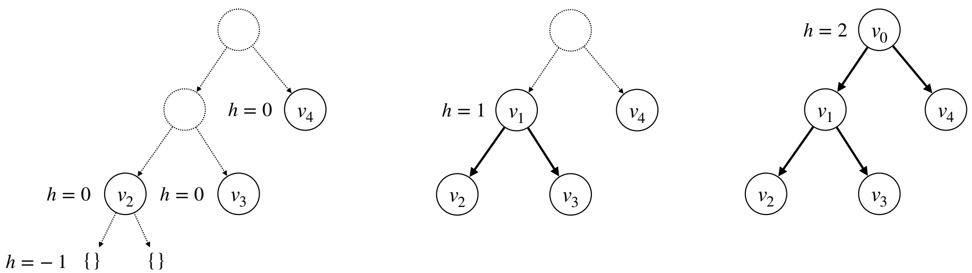
$$T = \{v_0\} \cup (\{v_1\} \cup (\{v_2\} \cup \{\} \cup \{\}) \cup (\{v_3\} \cup \{\} \cup \{\})) \cup (\{v_4\} \cup \{\} \cup \{\}))$$

Height of a binary tree

The **height** $h(T)$ of a binary tree is the number of links from the root to the deepest leaf

Formally:

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

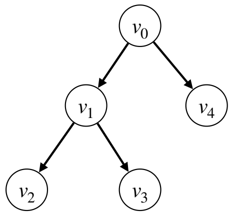


Perfect binary tree

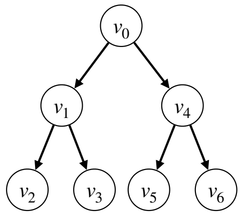
A binary tree is **perfect** if *any* of the following two equivalent conditions is satisfied:

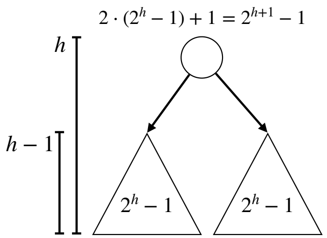
1. It has a maximal number of nodes for its height h
2. It has $2^{h+1} - 1$ nodes

Not perfect



Perfect





Implementing a binary tree

Operations

If T is a binary tree, the following operations are defined:

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

We can express many algorithm based only on these four operations!

Canonical representation

A binary tree can be represented by an object N which is either:

- The null object `NIL` (to represent an empty tree)
- A data structure with fields:
  - N.left — the left child object
  - N.right — the right child object
  - N.value — the node's value

In this case, the four operations are simply:

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

Computing the height of a binary tree

The formula for the height of a binary tree

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

translates directly into a recursive algorithm:

BinaryTreeHeight(T):

1. If empty(T):
   1.1. Return the value -1
2. Let $L \leftarrow$ left(T)
3. Let $R \leftarrow$ right(T)
4. Let $h_L \leftarrow$ BinaryTreeHeight(L)
5. Let $h_R \leftarrow$ BinaryTreeHeight(R)
6. Return $1 + \max\{h_L, h_R\}$

The complexity is $O(n)$, because the algorithm visits each node once

A note on encapsulation:

- This algorithm is agnostic on the choice of a representation for the binary tree
- Instead, it only requires the functions empty, left and right to be defined

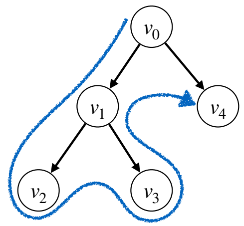
Depth-first traversal of a binary tree

Traversing a tree means visiting and processing all the nodes once in a certain order

Depth-first traversal starts from the root and visits recursively the left and right children

DFTraversal(T):

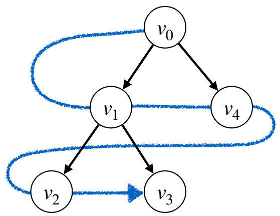
1. If empty(T):
   1.1. Return
2. Process value(T) // pre-order processing
3. Let $L \leftarrow$ left(T)
4. Let $R \leftarrow$ right(T)
5. Let DFTraversal(L)
6. Process value(T) // in-order processing
7. Let DFTraversal(R)
8. Process value(T) // post-order processing



Depth-first	**visit** order	v_0, v_1, v_2, v_3, v_4
Pre-order	**processing** order	v_0, v_1, v_2, v_3, v_4
In-order	**processing** order	v_2, v_1, v_3, v_0, v_4
Post-order	**processing** order	v_2, v_3, v_1, v_4, v_0

Breadth-first traversal of a binary tree

Breadth-first traversal visits the nodes layer by layer, using a queue to remember which subtree to visit next



Breadth-first visit/process order:
v_0, v_1, v_4, v_2, v_3

BFTraversal(Q):

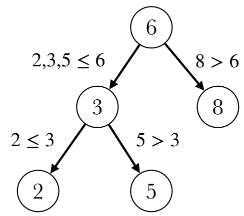
- **Precondition**: the queue $Q = \{T\}$ contains the tree as sole element

1. While Q is not empty:
   1.1. Let $T \leftarrow \text{Dequeue}(Q)$
   1.2. Process value(T)
   1.3. Let $L \leftarrow \text{left}(T)$
   1.4. Let $R \leftarrow \text{right}(T)$
   1.5. If not empty(L):
      1.5.1. Enqueue(Q, L)
   1.6. If not empty(R):
      1.6.1. Enqueue(Q, R)

Binary search tree

A binary tree T is a **binary search tree** (BST) iff

- it is empty (i.e., $T = \{\}$), *or*
- it is given by $T = \{r\} \cup L \cup R$, where
  - for all subtrees $S \subset L$, value(S) $\leq$ value(T) and
  - for all subtrees $S \subset R$, value(S) $>$ value(T) and
  - L and R are also BSTs



Note: this diagram shows the value of the nodes instead of the node indices

Searching a BST

Searching a BST T for a value x is done by descending from the root to a leaf, "turning" left or right depending on value comparisons

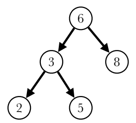
BSTSearch(T, x) :

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

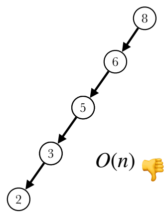
BSTSearch complexity is $O(h)$ as a function of the three height h

For a perfect (or sufficiently balanced) tree, $n \propto 2^h$ so the complexity is $O(\log n)$ as a function of the tree size n

However, for a degenerate tree (a chain), $n = h + 1$, so the complexity is $O(n)$



$O(\log n)$ 👍



$O(n)$ 👎

BST search: example

Searching for the value 5

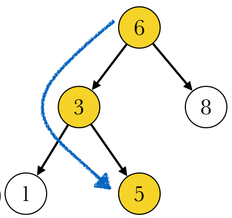
Steps:

1. 5 is less than 6, so search left
2. 5 is larger than 3, so search right
3. 5 is found

BSTSearch(T, 5)

BSTSearch(T.L, 5)

BSTSearch(T.L.R, 5)



Searching for the value 2

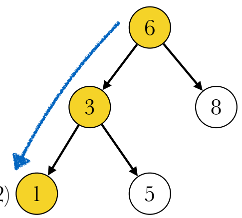
Steps:

1. 2 is less than 6, so search left
2. 2 is less than 3, so search left again
3. 2 is larger than 1, but there is no right child: stop

BSTSearch(T, 2)

BSTSearch(T.L, 2)

BSTSearch(T.L.L, 2)



Building a BST

We can trivially build a BST T by adding a new element x a time

The process is similar to searching a BST, except that a new leaf node is added to the tree to contain the new value

However, this process is **not** guaranteed to return a tree which is perfect or even reasonably balanced

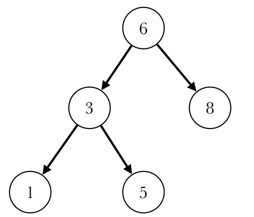
BSTInsert(N, x) :

- **Precondition**: N is a BST
- **Postcondition**: Returns the same BST N, extended with the new value x

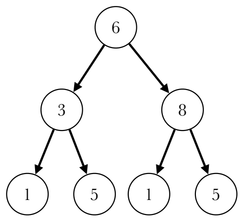
1. If N is NIL then return $\{x, \mathtt{NIL}, \mathtt{NIL}\}$
2. If $x \leq N.\text{value}$ then:
   2.1. Set $N.\text{left} \leftarrow \text{BSTInsert}(N.\text{left}, x)$
3. Else:
   3.1. Set $N.\text{right} \leftarrow \text{BSTInsert}(N.\text{right}, x)$
4. Return N

Complete binary trees

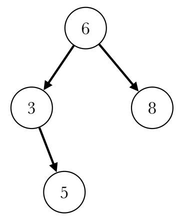
A binary tree is **complete** if all levels are full, except the last one which is partially filled from left to right



Complete



Perfect



Neither

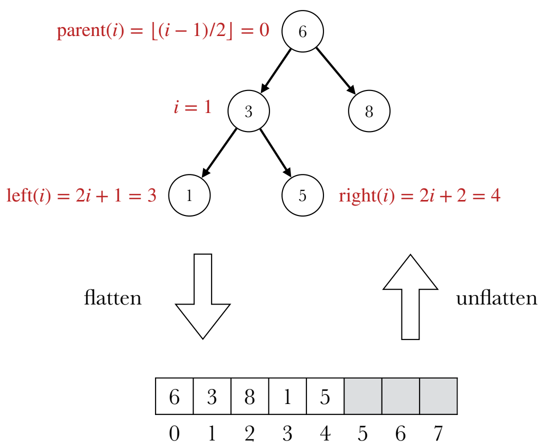
Representing a *complete* binary tree as an array

We can enumerate the elements of a complete tree from left to right and top to bottom, placing them in an array

The process can be inverted to reconstruct the complete tree unambiguously

Let i be the index of a given node in the array. Then:

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



Heaps

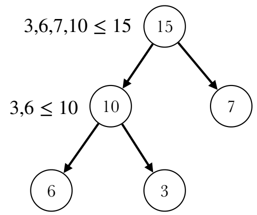
A binary tree T is a **max heap** iff:

- T is empty, *or*
- for all subtrees $S \subset T$, $\text{value}(S) \leq \text{value}(T)$

Note: the definition may look similar to a BST, but it is very different; in particular, we do not distinguish between left and right children

By construction, the heap's root is always the node in the tree with largest value

A **min heap** is similar, but with smaller instead of larger elements towards the top



Maintaining the heap property: SiftUp & SiftDown

We can "fix" a tree T which is a heap except for the value of subtree S, which is "defective"

SiftUp(S) is used to fix the tree if the value of S is too small

- It works by swapping the value of S with its parent until a suitable place in the tree is found

SiftDown(S) is used to fix the tree if the value of S is too large

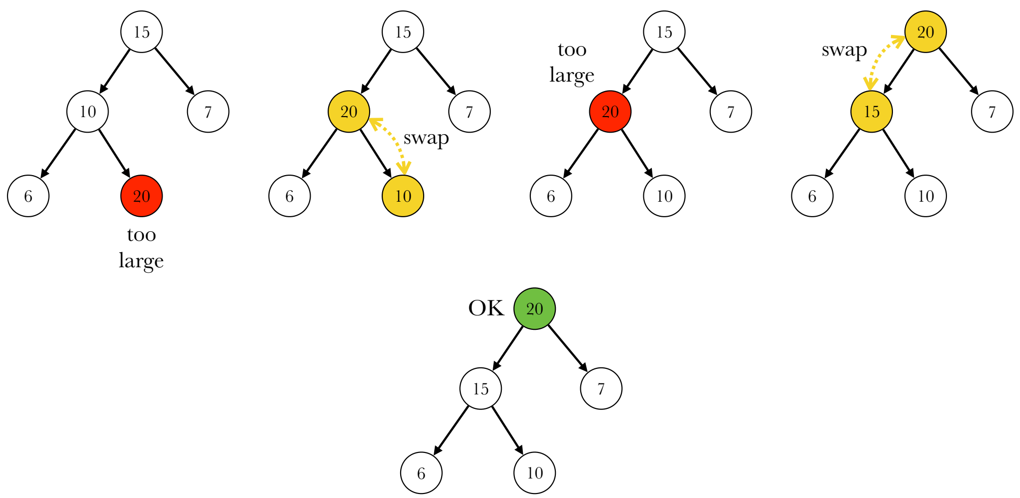
- It works by swapping the value of S with the "largest" child until a suitable place in the tree is found

SiftUp(S):

- **Precondition**: S is a subtree of a binary tree T which already has the heap property, or the latter can be restored by reducing value(S)
- **Postcondition**: The tree T is the same as before, except that the subtree values have been permuted to satisfy the heap property

1. If empty(parent(S)) return
2. If value(parent(S)) $\geq$ value(S) return
3. Swap the values of S and parent(S)
4. Call recursively SiftUp(parent(S))

SiftUp: example



Building a heap

Given an array A, the goal is to transform it into a valid heap by swapping its elements

We build a heap from the bottom up:

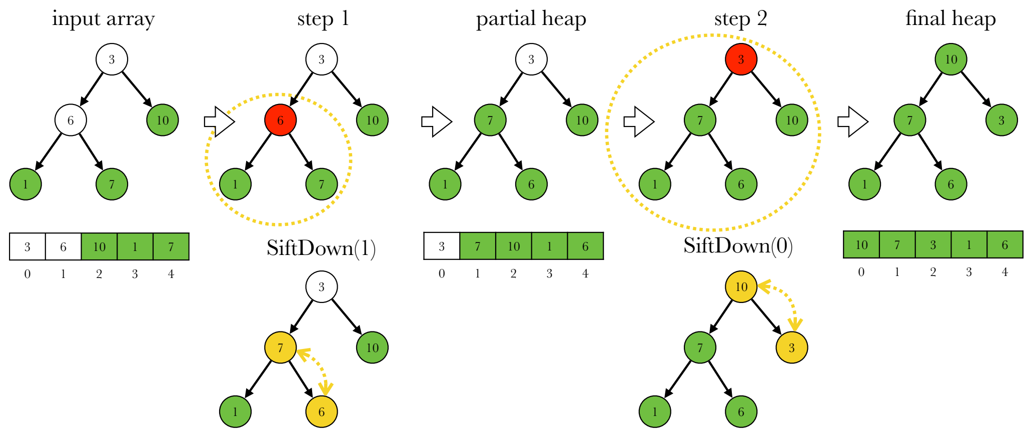
- The leaves are heaps of one element
- Moving one level up, we merge pairs of subtrees by adding a new root element to link them
- Because the new root can be "defective", we call SiftDown on it to "fix" it

BuildHeap(A):

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

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

Building a heap: example

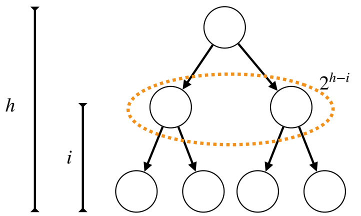


BuildHeap: complexity

Each call to SiftDown(S) is $O(i)$, where i is the height of the subtree S

If h is the height of the tree, there are 2^{h-i} subtrees of height i

The cost of calling SiftDown for level i is thus $O(i \cdot 2^{h-i})$



The total cost of BuildHeap is obtained by summing over all levels:

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

Recall that $h \propto \log n$

Hence, BuildHeap complexity is $O(n)$

Heap sort

A heap can be used to sort an array

First, the array is transformed into a heap using BuildHeap

Then, the top (maximum) element is extracted and the heap property is restored calling SiftDown

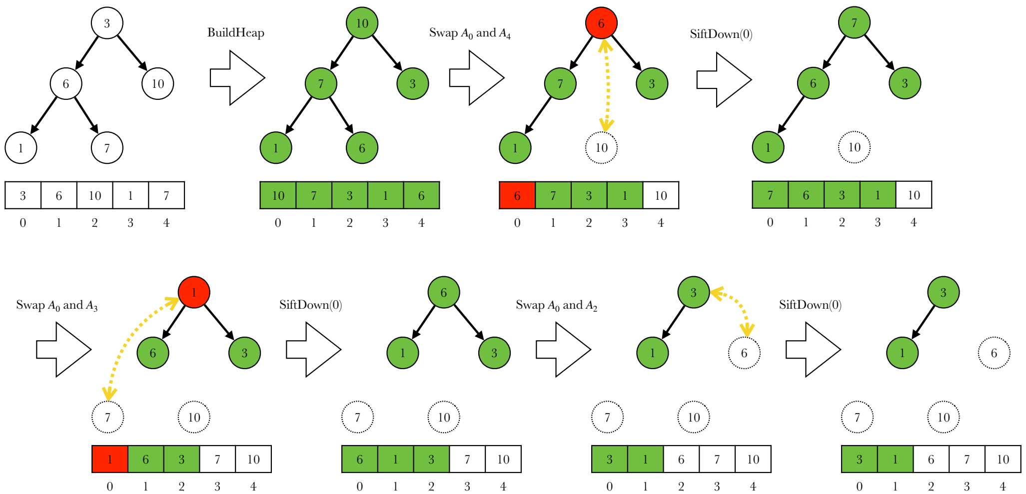
Then, the top (now second largest) element is extracted, the heap property is restored, and so on

The cost is $O(n \log n)$, same as for MergeSort (could have it been better?)

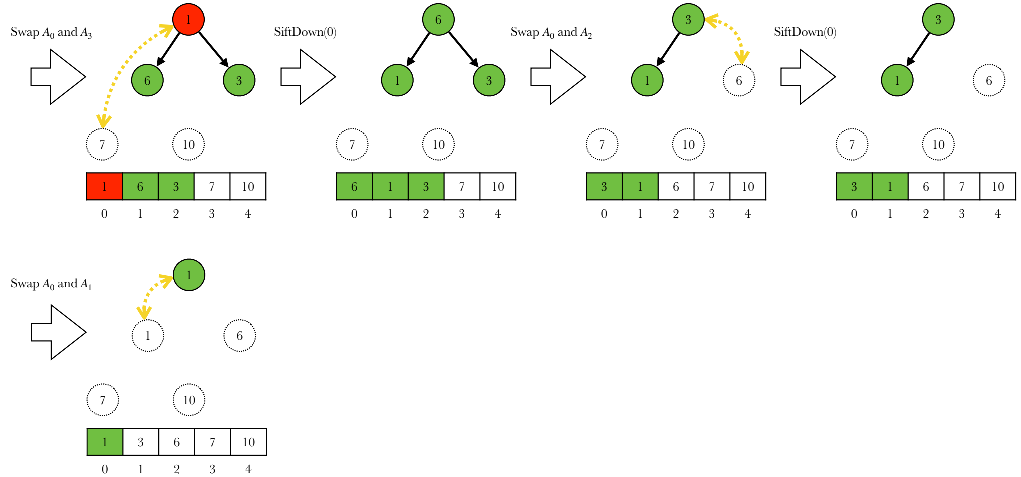
HeapSort(A):

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

Heap Sort: example



Heap Sort: example



Priority queues

We can use a heap to implement a **priority queue** with two operations:

- PriorityEnqueue(Q, x) to add an element x to the queue
- PriorityDequeue(Q) to extract the "highest priority" (largest) element from the queue

The queue Q is a data structure with fields

- $Q.A$ preallocated array for storing elements
- $Q.\text{size}$ number of elements in the queue

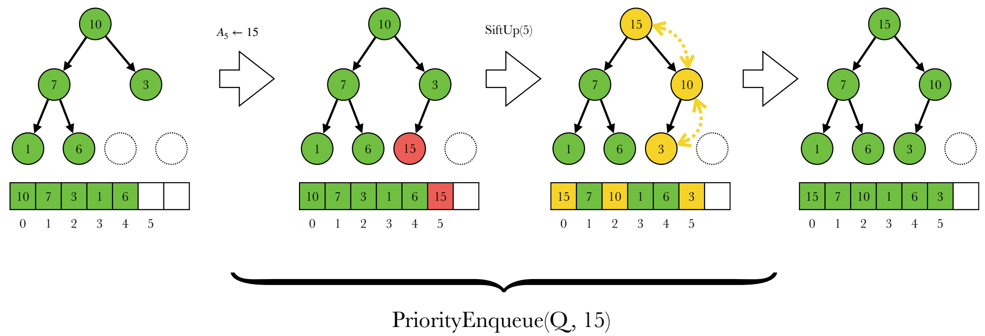
PriorityEnqueue(Q, x):

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 $Q.\text{size} \leftarrow i + 1$

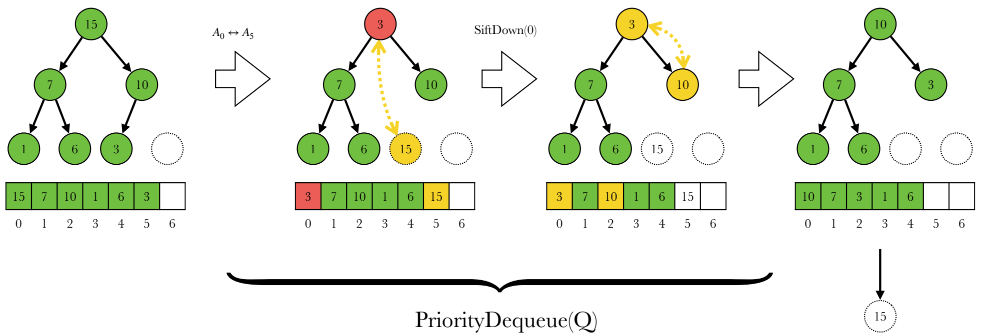
PriorityDequeue(Q, x):

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 $Q.\text{size} \leftarrow i - 1$
6. Return A_i

PriorityEnqueue: example



PriorityDequeue: example



B16 Software Engineering

Algorithms and Data Structures 1

Part 3 of 4: Hashing

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Hash tables as a generalisation of arrays

Arrays

- Map indices $\{0, 1, \ldots, n-1\}$ to values $i \mapsto A_i$
- Allow fast $\Theta(1)$ access to any of the indices

However, we often wish to index data based on different types of indices

For example, in a dictionary we would index entries based on words, which are strings, not integers

Hash tables

- Map keys $\mathscr{K}$ (e.g., ints, strings) to values $k \mapsto A_k$
- Allow fast $\Theta(1)$ access on average

Hence, a hash table generalises an array to keys other than consecutive integers

Hash tables via chaining

The simplest implementation of a **hash table** is a a *linked list* L containing a chain of key-value pairs $\langle k, v\rangle$

Complexity:

- Retrieving a key k requires scanning the entire list for a match, with worst case cost $\Theta(n)$
- Inserting a *new* element $\langle k, v\rangle$ is $\Theta(1)$: just call ListInsertAfter(L, k, v)
- But, if the inserted key k *can* already exist, one needs to check first if the key is already present to avoid duplicates, with cost $\Theta(n)$

This is also the *average* case cost, as on average key k is found half-way through the list

ChainInsert(L, k, v) :

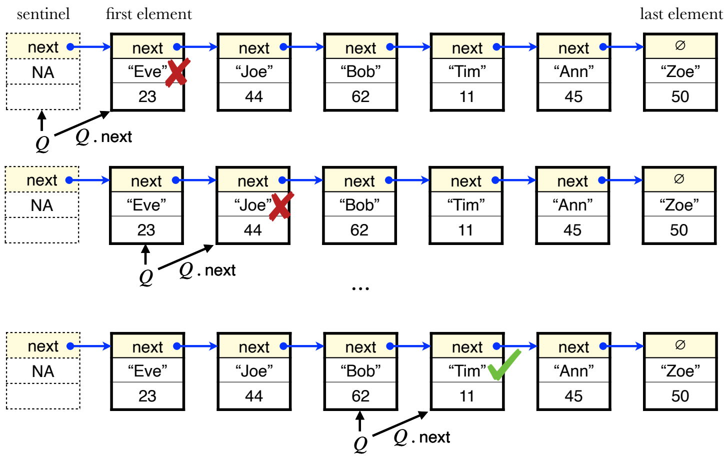
1. $N \leftarrow$ ListFindPredecessor$(L, \langle k, \star\rangle)$
2. If $N =$ NIL then:
   2.1. Call ListInsertAfer$(L, \langle k, v\rangle)$
3. Else:
   3.1. set N . next . value $\leftarrow \langle k, v\rangle$

ChainRetrieve(L, k) :

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

Hash table via chaining: example

ListFindPredecessor$(Q, \text{"Tim"})$



Multiple chains

We can significantly speed up access by using *multiple*, short chains

Each chain is tasked with storing a subset of keys

The **hash table** is a structure H with a single field:

- $H.A$ an array of m chains $L_0, \ldots, L_{m-1}$

The **load factor** α is the average number of elements per chain

$$\alpha = \frac{n}{m}$$

We also require a **hash function** h mapping keys k to chains $s = h(k)$

$h : \mathscr{K} \to \{0, 1, \ldots, m-1\}$

The cost of the hash function is independent of n and m ($\Theta(1)$ complexity)

Intuition

- We expect the cost of accessing an element in the hash table to be $O(\alpha)$ on average
- If so, and if the number of chains $m = \Omega(n)$ is proportional to the number of elements n added to the hash table, then the access cost is $O(1)$, the same as for an array

Multiple chains: insert and retrieve

HashInsert(H, k, v) :

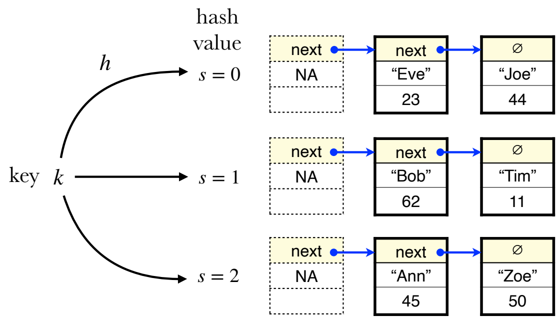
1. Let $s \leftarrow h(k)$
2. Let $L \leftarrow H.A[s]$
3. Call ChainInsert(L, k, v)

HashRetrieve(L, k) :

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

Example hash function h

$h(\text{Eve}) = 0$
$h(\text{Joe}) = 0$
$h(\text{Bob}) = 1$
$h(\text{Tim}) = 1$
$h(\text{Ann}) = 2$
$h(\text{Zoe}) = 2$

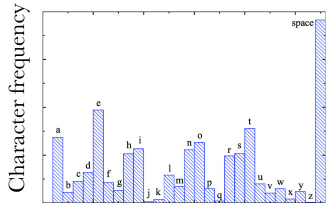


Hash functions

Hash functions goals

The goals of a hash function h are:

- To map keys k to one of m slots
- To do so quickly ($\Theta(1)$ complexity for all keys)
- To do so uniformly, meaning that different keys can be expected to spread equally in different slots



Example for string keys

- k is a string encoded in ASCII
- Set $m = 128$
- Set $h(k)$ to be the ASCII value of the first character

This satisfies some of the goals:

✓ Maps strings to $m = 128$ slots

✓ Does so quickly (just read the first character)

✗ But the key distribution is generally *not* uniform because certain characters are much more frequent than others

Building hash functions

Keys as integers

Any key k can always be thought of as a (large) natural number:

- Take the C bytes c_i used to represent the key in memory
- Interpret the key as the natural number:

$$\sum_{i=0}^{C-1} c_i \cdot 256^i$$

The division method

Define:

$h(k) = k \mod m$

Thus $h(k)$ is the *remainder* of dividing k by m

- The remainder is always in the range 0 to $m - 1$
- The remainder is relatively quick to compute
- Is the reminder uniformly distributed, and thus a good hash function?

Remainder method: choosing m

Criterion: we would like $h(k)$ to depend on all the bits of the binary representation of the number k

Choosing m to be a prime number achieves this

To show this, assume that k and k' differ only by bit i, so that $k' = k + 2^i$

Then:

$$\begin{aligned} h(k') - h(k) &= (k' \mod m) - (k \mod m) \\ &= k' - k \mod m \\ &= 2^i \mod m \\ &\neq 0 \end{aligned}$$

This shows that two keys that differ by a single bit have different hash values

Average cost analysis

In the *worst case*, all keys are hashed to the same slot and insertion and retrieval of keys is $\Omega(n)$

Under suitable statistical assumptions, the *average case* is much better

Simple Uniform Hashing Assumption (SUHA)

- The keys k added to the hash table are selected i.i.d. at random
- All *hash values* are equally probable:

  $P[h(k) = s] = 1/m$

  for all $s \in [0, m - 1]$

Average key retrieval cost

Theorem: missing key cost

Under the SUHA, the number of list elements visited in attempting to retrieve a key k that is *not* contained in a hash table H, averaged over all possible keys and tables, is $1 + \alpha$

Proof (sketch)

This is because the average length of chains is $\alpha = n/m$ if the n elements in the table spread uniformly to the m chains

Since the key is missing, the entire chain must be visited before giving up

Theorem: existing key cost

Under the SUHA, the number of list elements visited by retrieving a key k that *is* contained in a hash table H, averaged over all possible keys and tables, is $1 + \alpha/2 - \alpha/2n$

Proof (sketch)

This proof, due to D. Knuth, is difficult and optional

Intuitively, if the key *is* present in the hash table, on average we need to visit only half a chain before finding it

B16 Software Engineering

Algorithms and Data Structures 1

Part 4 of 4: Graphs

Dr Andrea Vedaldi
4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see
http://www.robots.ox.ac.uk/~vedaldi/teach.html

Directed graphs

A **directed graph** $G = (V, E)$ is given by

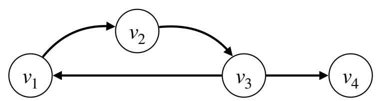
- a set ot **vertices** $V = \{v_1, \ldots, v_{|V|}\}$ and
- a set of **edges** $E \subset V \times V$

An edge $(v_i, v_j) \in E$ is drawn as an *arrow* $v_i \to v_j$

A directed graph can be represented by an **adjacency matrix** A such that

- $A \in \{0,1\}^{|V|\times|V|}$
- $A_{ij} = 1$ iff $(v_i, v_j) \in E$

Example



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

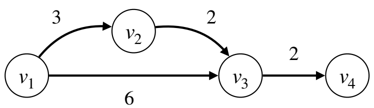
Weighted graph

A **weighted graph** (G, w) has weights $w(e) \in \mathbb{R}$ associated to the edges

It can be represented by a **weighted adjacency matrix** W where

- $W_{ij} = w(v_i, v_j)$ if $(v_i, v_j) \in E$
- $W_{ij} = \infty$ otherwise

Example



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

Paths

A **path** in an directed graph is a sequence of vertices $p = (v_1, v_2, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$

The path **connects** the **source** v_1 to the **destination** v_n

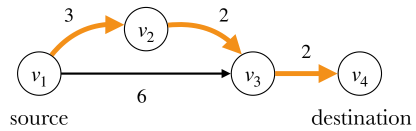
The **length** of the path is the number of edges in it (i.e., the number of vertices minus 1)

In a weighed directed graph, the **weight of a path** is the sum of the edge weights:

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

Example

- $p = (v_1, v_2, v_3, v_4)$
- $\text{length}(p) = 3$
- $w(p) = 7$



Path composition and subpaths

We can compose paths by **concatenating** them. Let:

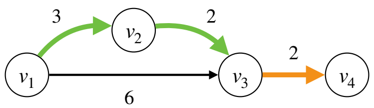
- $p' = (v_1, \ldots, v_2)$ connects v_1 to v_2
- $p'' = (v_2, \ldots, v_3)$ connects v_2 to v_3

Note that the destination of p' is the source of p''

Then $p = p' \oplus p'' = (v_1, \ldots, v_2, \ldots, v_3)$ connects v_1 to v_3

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

Example



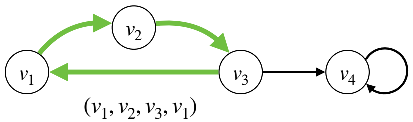
$$p = (v_1, v_2, v_3, v_4) = (v_1, v_2, v_3) \oplus (v_3, v_4)$$

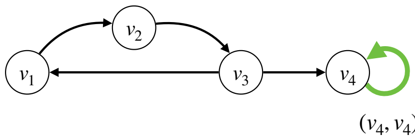
Cycles

A **cycle** is a path $p = (v_1, \ldots, v_1)$ where source and destination coincide

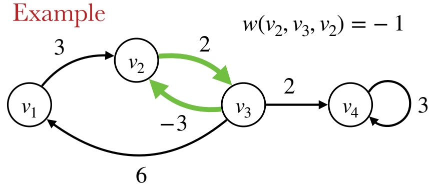
A **negative cycle** is a cycle whose weight is negative

Examples



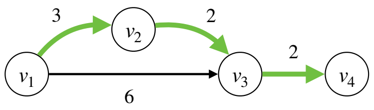


Example

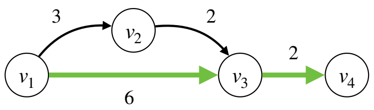


Shortest paths

A path p connecting u to v is **shortest** if no path with a smaller weight also connects u to v

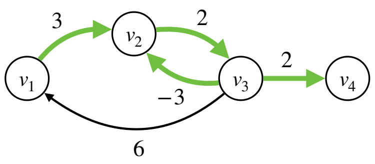


$w(v_1, v_2, v_3, v_4) = 7$ (shortest)



$w(v_1, v_3, v_4) = 8$ (not shortest)

If there is a negative cycle, then a shortest path may not be defined

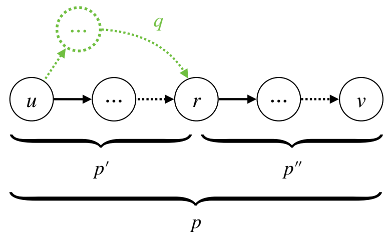


$w(v_1, v_2, v_3, v_2, v_3, \ldots, v_3, v_4)$ (arbitrarily short)

Optimal substructure of shortest paths

Theorem

If p is a shortest path and $p' \oplus p'' = p$ are two subpaths, then p' and p'' are also shortest paths



Proof

- Let $p' = (u, \ldots, r)$ and $p'' = (r, \ldots, v)$
- By definition $w(p) = w(p') + w(p'')$
- If p' is *not* shortest, then we can find a path q from u to r such that $w(q) < w(p')$
- Hence $q \oplus p''$ connects the same vertices as p and has smaller weight $w(q \oplus p'') < w(p)$
- Hence p is not a shortest path

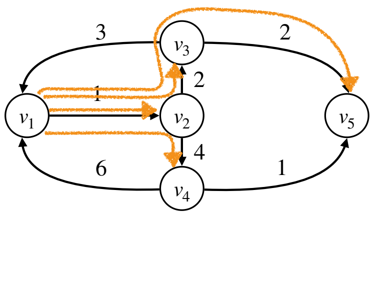
Shortest paths problems

Single-source shortest paths (SSSP)

Given an oriented weighted graph (G, w) and a source vertex u, find shortest paths to all vertices $v \in V$

All-pairs shortest paths (APSP)

Given an oriented weighted graph (G, w), find shortest paths between all pairs of vertices $u, v \in V$



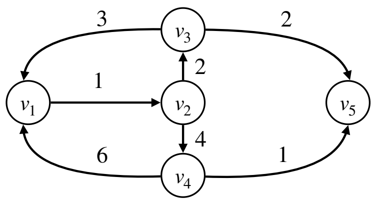
Representing shortest paths

The shortest paths p_{uv} connecting all pair of vertices u to v can be encoded by using a **predecessor matrix** $P \in V^{|V|\times|V|}$ and a **distance matrix** $D \in (\mathbb{R}_+ \cup \{\infty\})^{|V|\times|V|}$ such that:

- $r = P_{uv}$ is the node before v in the path p_{uv}
- $D_{uv} = w(p_{uv})$

To reconstruct the shortest paths, backtrack:

- Start with u and v so the path is $(u, \ldots, v)$
- Let $r = P_{uv}$ so the path is $(u, \ldots, r, v)$
- Let $t = P_{ur}$ so the path is $(u, \ldots, t, r, v)$
- etc.



$$P = \begin{bmatrix} 1 & 1 & 2 & 2 & 3 \\ 3 & 2 & 2 & 2 & 3 \\ 3 & 1 & 3 & 2 & 3 \\ 4 & 1 & 2 & 4 & 4 \\ \cdot & \cdot & \cdot & \cdot & 5 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 1 & 3 & 5 & 5 \\ 5 & 0 & 2 & 4 & 4 \\ 3 & 4 & 0 & 8 & 2 \\ 6 & 7 & 9 & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

for SSSP, we only need one row

Bellman-Ford SSSP

The **Bellman-Ford algorithm** computes the shortest paths p_v from a fixed source u to all vertices v

It works incrementally, by establishing all shortest paths of length 1, then of length 2 and so on

Note: We assume for simplicity that there are no negative cycles, but the algorithm can be modified to detect such cycles

Complexity: $O(|V| \cdot |E|)$ or $O(|V|^3)$ for dense graphs

BellmanFord(V, E, w, u) :

1. For all v in V:
   1.1. Let $D_v \leftarrow 0$ if $v = u$ or ∞ otherwise
   1.2. Let $P_v \leftarrow u$ if $v = u$ or -1 otherwise
2. Repeat $|V| - 1$ times:
   2.1. For all $(r, v) \in E$
      2.1.1. Call Relax(D, P, w, r, v)
3. Return D and P

Path relaxation

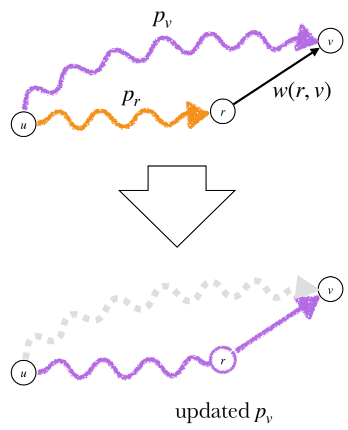
Let p_v the *current* path (not necessarily shortest) from u to v

Let $(r, v) \in E$ be an edge with head v and let p_r be the current path from u to r

The Relax routine *replaces* p_v with $p_r \oplus (r, v)$ if the latter is *shorter*:

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

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

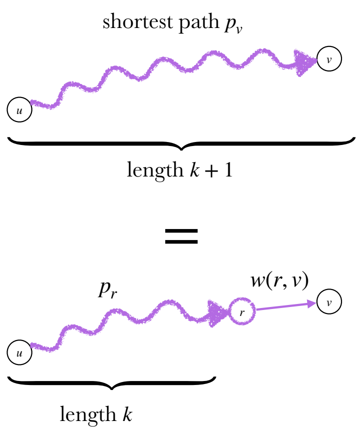


Bellman-Ford: correctness

Theorem: After k iterations, the Bellman-Ford algorithm has established all shortest paths of length at most k (and so all shortest paths after $|V| - 1$ iterations).

Proof (by induction)

- Suppose that the theorem is true for k iterations
- A shortest path p of length $k + 1$ can bee written as $p = (u, \ldots, r, v)$ where, due to the optimal substructure, $p' = (u, \ldots, r)$ is a shortest path of length k, hence already established
- When (r, v) is relaxed during iteration $k + 1$, a path p_{uv} from u to v at least as good as p is established



Floyd-Warshall APSP

The **Floyd-Warshall algorithm** computes paths p_{uv} between all pairs of vertices u and v

It does so incrementally, by establishing all shortest paths with no intermediate nodes (direct edges), then all shortest paths with intermediate notes in the set $\{1\}$, then in the set $\{1,2\}$ and so on

Complexity: $O(|V|^3)$ for sparse or dense graphs

FloydWarshall(V, E, w) :

1. For all u, v in V:
   1.1. Let $D_{uv} \leftarrow w(u, v)$ if $(u, v) \in E$ or ∞ otherwise
   1.2. Let $P_{uv} \leftarrow u$ if $(u, v) \in E$ or -1 otherwise
2. For all r in V:
   2.1. For all u in V:
      2.1.1. For all v in V:
         2.1.1.1. Call RelaxFW(D, P, r, u, v)
3. Return D and P

Path relaxation (Floyd-Warshall variant)

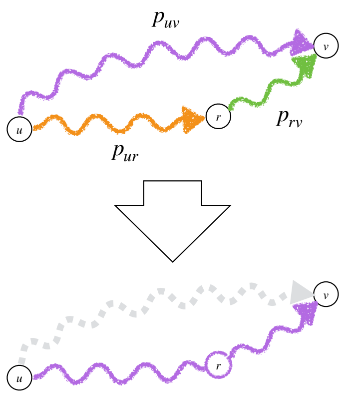
Let p_{uv} the *current* path (not necessarily shortest) from u to v

Let $r \in V$ be an intermediate vertex and let p_{ur} and p_{rv} be the current paths from u to r and from r to v

The RelaxFW routine *replaces* p_{uv} with $p_{ur} \oplus p_{rv}$ if the latter is *shorter*:

RelaxFW(D, P, r, u, v) :

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



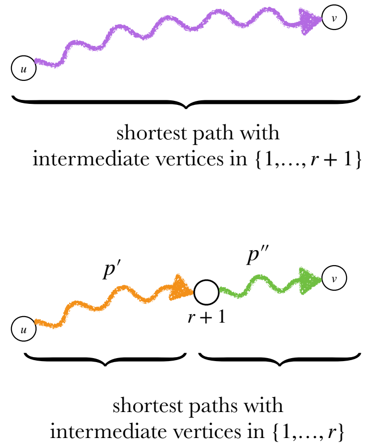
updated p_{uv} (if shorter)

Floyd-Warshall: correctness

Theorem: After r iterations, the Floyd-Warshall algorithm has established all shortest paths whose intermediate vertices are within the set $\{1,\ldots,r\}$ (and so all shortest paths in $|V|$ iterations).

Proof (by induction)

- Suppose that the theorem is true for r iterations
- A (simple) shortest path p whose intermediate vertices are within $\{1,\ldots,r+1\}$ and that contains vertex $r+1$ can be written as $p = (u,\ldots,r+1,\ldots,v)$
- The intermediate vertices of shortest paths $p' = (u,\ldots,r+1)$ and $p'' = (r+1,\ldots,v)$ are in $\{1,\ldots,r\}$, so p' and p'' have already been established
- When $(r+1,u,v)$ is relaxed during iteration $r+1$, a path p_{uv} from u to v at least as good as the shortest path p is established



Dijkstra's SSSP algorithm

The **Dijkstra algorithm** solves the SSSP problem under the assumptions that there are *no negative weights*

It establishes the shortest paths p_v from a source u in order of non-decreasing weight

To do so, it maintains a set Q of "open" vertices for which a shortest path has not yet been established, closing one more vertex at each iteration

Complexity: The naïve implementation of this algorithm shown to the right is $O(|V|^3)$

Dijkstra(V, E, w, u):

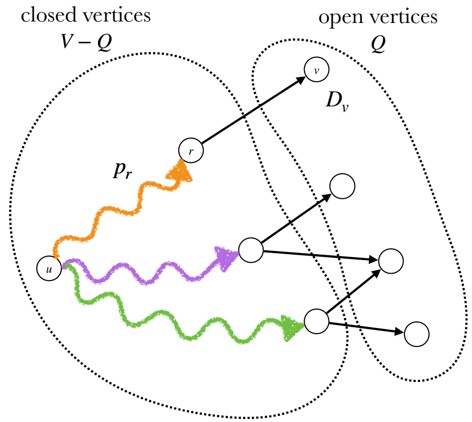
1. For all v in V:
   1.1. Let $D_v \leftarrow 0$ if $v = u$ or ∞ otherwise
   1.2. Let $P_v \leftarrow -1$
2. Set $Q \leftarrow V$
3. Repeat until Q is not empty:
   3.1. Let $v^* \leftarrow \operatorname{argmin}_{v \in Q} D_v$
   3.2. Remove v^* from Q
   3.3. For all $v \in Q$ such that $(v^*, v)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

Dijkstra's SSSP algorithm: the invariants

The algorithm maintains the following invariant:

(P1) For all closed vertices $r \in V - Q$, p_r is a shortest path

(P2) For all open vertices $v \in Q$, the vector D is given by $D_v = \min_{r \in Q - V} D_r + w(r, v)$



Why the algorithm finds a shortest path

We have:

- $D_v = \min_{r \in Q-V} D_r + w(r, v)$ (invariant (P2))
- $v^* \leftarrow \operatorname{argmin}_{v \in Q} D_v$ (calculation of v^*)
- $D_v = w(p_v)$ (definition of D)

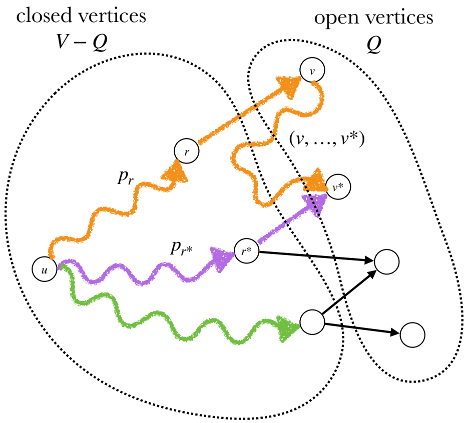
By composing argmin and min, the newly determined path is $p_{v^*} = p_{r^*} \oplus (r^*, v^*)$, where

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

Any other path from u to v^* is of the form $q = (u, \ldots, r) \oplus (r, v) \oplus (v, \ldots, v^*)$ where r is closed and v is open.

Hence, p_{v^*} is indeed shortest:

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



Dijkstra's algorithm with a priority queue

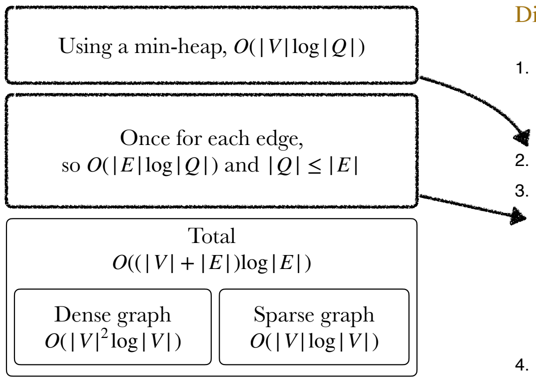
Dijkstra(V, E, w, u):

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

DijkstraPriority(V, E, w, u):

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

Dijkstra's algorithm with a priority queue



DijkstraPriority(V, E, w, u):

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

Conclusions

Key concepts

We have covered:

- Recap of problems, algorithms, complexity
- Lower bound on sorting complexity
- Array, stacks, queues, linked lists
- Binary trees, binary search trees
- Heaps, priority queues
- Hash functions and hashing
- Graphs and shortest paths

Hints

Practice: try implementing and testing algorithms "for real". Use C++ for this course, or any other programming language in general (e.g., Python)

The exercises mostly ask you to write and test algorithms in C++

Use the provided example code, especially for the exercises

Use the notes as needed: they contain several details that can help you to understand the content more firmly