

# IA Algorithms

Damon Wischik, Computer Laboratory, Cambridge University. Lent Term 2020

## Contents

<b>5</b>	<b>Graphs</b>	<b>1</b>
5.1	Notation and representation . . . . .	1
5.2	Depth-first search . . . . .	4
5.3	Breadth-first search . . . . .	6
5.4	Dijkstra's algorithm . . . . .	9
5.5	Algorithms and proofs . . . . .	13
5.6	Bellman-Ford . . . . .	15
5.7	Dynamic programming . . . . .	18
5.8	Johnson's algorithm . . . . .	20
5.9	Graphs and big data* . . . . .	22
<b>6</b>	<b>Flow and subgraph algorithms</b>	<b>25</b>
6.1	Max-flow min-cut theorem . . . . .	26
6.2	Ford-Fulkerson algorithm . . . . .	29
6.3	Matchings . . . . .	33
6.4	Prim's algorithm . . . . .	36
6.5	Kruskal's algorithm . . . . .	39
6.6	Topological sort . . . . .	41
<b>7</b>	<b>Advanced data structures</b>	<b>45</b>
7.1	Amortized costs . . . . .	46
7.2	Potential functions . . . . .	48
7.3	Heaps* . . . . .	50
7.4	Fibonacci heap . . . . .	52
7.5	Analysis of Fibonacci heap . . . . .	56
7.6	Implementing the Fibonacci heap* . . . . .	58
7.7	Disjoint sets . . . . .	61
<b>8</b>	<b>Geometrical algorithms</b>	<b>63</b>
8.1	Segment intersection . . . . .	63
8.2	Jarvis's march . . . . .	65
8.3	Graham's scan . . . . .	68



## 7. Advanced data structures

Some data structures are designed so that operations on them are usually very fast, for example by being lazy and letting ‘mess’ accumulate, the price being that every now and then an operation needs expensive internal ‘housekeeping’.

- A hash table is typically stored as an array, and we increase the capacity of the array and rehash all the existing items whenever the occupancy exceeds a certain threshold. Java’s `HashMap`, for example, rehashes once occupancy reaches 75%. Most insertions are very fast, but every so often an insertion will trigger a slow rehash.
- Dijkstra’s algorithm uses a priority queue. Running the algorithm involves  $O(V)$  calls to `popmin` and  $O(E)$  calls to `push` or `decreasekey`, where  $V$  is the number of vertices and  $E$  the number of edges. As far as running time is concerned, what matters is the aggregate of all of these operations. We want a priority queue implementation that makes the aggregate running time as fast as possible, even if some of the individual operations are slow.

reshaping a hash  
table: section 4.7

running time  
analysis of  
Dijkstra’s algorithm:  
section 5.4 page 11

The hope of doing housekeeping in batches is that we might be able to get efficiency gains, when we look in aggregate at a sequence of operations. Data structures that achieve this can be described as advanced. More generally, the art of advanced data structures is inventing cunning tricks so that work done by one operation in the sequence, such as housekeeping, can benefit other operations.

The study of advanced data structures is therefore centered around analysing the worst-case computational complexity of *sequences* of operations. This is known as *aggregate analysis*.

**Example 7.1 (Aggregate analysis).** When we analysed the running time of `dfs_recurse` on page 4, we argued<sup>15</sup> “Each vertex is visited at most once; when a vertex is visited then the algorithm iterates over the edges out of that vertex; therefore the algorithm looks at each vertex and each edge at most once; therefore the running time is  $O(V + E)$ .”

\* \* \*

When we analyze computational complexity, it’s good practice to seek matching upper and lower bounds on the worst-case performance. Likewise, when we’re analysing advanced data structures, we want to find tight bounds for sequences of operations.

**Example 7.2 (Tight bounds).** Consider a binomial heap. The worst case complexity of insertion is  $O(\log n)$ , where  $n$  is the number of items in the heap. It’s also  $O(n!)$ , and  $O(n^{n^n})$  ... these are technically true big- $O$  statements, but they’re pretty useless! However, for arbitrarily large  $n$ , we can adduce a concrete example where the complexity is  $\Omega(\log n)$  — namely, inserting into a heap that contains  $n = 2^k - 1$  items, for any  $k$ . This gives us matching  $O$  and  $\Omega$  bounds, thus the bounds are tight: the worst case complexity is  $\Theta(\log n)$ .

binomial heap:  
section 4.8.2

Now consider repeated insertions into a binomial heap. Some detailed thought tells us that if we perform two insertions in a row, either the first or the second must be  $O(1)$  — it’s impossible to hit the  $\log n$  case twice in a row. What is the worst case we can actually achieve, for  $m$  insertions? See the example sheet!

<sup>15</sup>Notice the subtle trick: the analysis looks at an individual ‘location’ in the data structure (e.g. an edge of the graph), asks “how much work is done *on this location*, over the entire sequence of operations?”, and sums up over all locations. This trick, when it works, is much easier than trying to find a formula for ‘amount of work done in operation number  $i$ ’ and summing over  $i$ .

## 7.1. Amortized costs

“Amortize a loan” means “pay it off gradually by making regular repayments before the debt finally becomes due”. We can use the same idea to think about the cost of an expensive ‘housekeeping’ operation in an advanced data structure. Here’s a simple illustration. Suppose we want to store a list of items, and we have to support four operations:

```
class MinList<T>:
    append(T v) # add a new item v to the list
    flush()    # empty the list
    foreach(f) # do f(x) for each item in the list
    T min()    # get the minimum of all items in the list
```

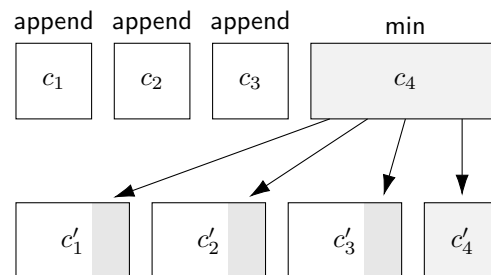
The first three operations are straightforward; a simple linked list will do. To implement `min`, here are four stages of enlightenment.

**Stage 0.** Simply iterate through the entire list every time we want to compute `min`. This takes time  $\Theta(n)$ , where  $n$  is the number of items in the list.

**Stage 1.** It’s a waste to redo the work involved in computing `min`. Instead, we should remember the result of the last call to `min`, and keep a pointer to the tail of the list at the time of that last call. Next time we need `min` we only need to iterate through the items added after the last `min` or `flush`. It’s faster—but the worst case is still  $\Theta(n)$ .

**Stage 2.** We could store the minimum of the entire list, and update it every time a new item is added. This way, `append` and `min` are both  $O(1)$ , and obviously we can’t do any better than this! We’d describe this as “amortizing the computation of `min`”, meaning we split the work up into small pieces done along the way.

**Stage 3.** If we count up the total amount of work for the Stage 2 implementation, it’s no better than that for Stage 1. (It could even be worse, if `flush` is called before `min`.) Morally, it’s unfair to say that Stage 2 has better running-time complexity. Instead, we should just stick with the Stage 1 implementation and ‘pay off’ the running time of `min` in early repayments, by *ascribing* its cost to each of the `append` calls that preceded it. We’ll say that those `append` calls have amortized costs  $c'_1$ ,  $c'_2$ , and  $c'_3$ , all  $O(1)$ , and that the final `min` call has amortized cost  $c'_4$ , also  $O(1)$ .



### DEFINITION

Here’s how amortized costs are referred to in practice. We might read for example that a certain data structure “supports `push` at amortized cost  $O(1)$ , and `popmin` at amortized cost  $O(\log N)$ , where  $N$  is the number of elements stored.” This is just a way of writing a statement about aggregate costs: it says that for any sequence comprising  $m_1$  `push` operations and  $m_2$  `popmin` operations,

$$\text{worst-case aggregate cost} \leq m_1 O(1) + m_2 O(\log N) = O(m_1 + m_2 \log N).$$

**Formalities.** Here is a formal definition. Let there be a sequence of  $k$  operations, whose true costs are  $c_1, c_2, \dots, c_k$ . Suppose we invent  $c'_1, c'_2, \dots, c'_k$  such that

$$c_1 + \dots + c_j \leq c'_1 + \dots + c'_j \quad \text{for all } j \leq k; \quad (3)$$

then we call these *amortized costs*. In words,

$$\text{aggregate true cost of a sequence of operations} \leq \text{aggregate amortized cost of those operations.}$$

Note that this inequality applies to sequences of any length  $j \leq k$ . It is not an asymptotic (big- $O$ ) statement.

When we read an asymptotic statement about amortized costs, for example

$$\text{worst case aggregate cost} = O(m_1 + m_2 \log N),$$

then the big- $O$  bound is asymptotic in  $N$  (the size of the data structure), and the statement holds for every  $m_1$  and  $m_2$ . Typically we're interested in cases where  $m_1$  and  $m_2$  grow with  $N$ . Formally, the asymptotic statement says: there exist  $N_0$  and  $k > 0$  such that for all  $N \geq N_0$ , and for all  $m_1$  and  $m_2$ ,

$$\text{worst case aggregate cost} \leq k(m_1 + m_2 \log N).$$

In order for amortized costs to be useful, we must restrict attention to sequences of operations that begin with an initially empty data structure. To see why this is needed, consider for example a hash table that starts with  $N$  items, and is just on the brink of needing a rehash. Then the next insertion will cost  $\Omega(N)$ , so we'd be forced to say that insertion is  $\Omega(N)$ . This is not helpful! This is why formal statements about amortized costs generally have awkward phrasing like "consider any sequence of  $m \geq N$  operations on a data structure, initially empty, where  $N$  of those operations are insertions".

#### EXAMPLE: DYNAMIC ARRAY

Here is a more involved example. Consider a dynamically-sized array with initial capacity 1, and which doubles its capacity whenever it becomes full. (To be precise: we maintain a fixed-length array; when it becomes full then we allocate a new fixed-length array of double the length, copy everything across from the old array, and deallocate the old array. We'll only consider appending, not deleting.)

initially empty  
 $\square$   
 append  
 $\bullet$   
 append, requires doubling capacity  
 $\bullet \bullet$   
 append, requires doubling capacity  
 $\bullet \bullet \bullet \square$   
 append  
 $\bullet \bullet \bullet \bullet$   
 append, requires doubling capacity  
 $\bullet \bullet \bullet \bullet \square \square \square$

**Aggregate analysis.** Suppose that the cost of doubling capacity from  $m$  to  $2m$  and copying everything across is  $\leq \kappa m$  for some constant  $\kappa > 0$ . After adding  $n$  elements, the total cost from all of the doubling is

$$\leq \kappa(1 + 2 + \dots + 2^{\lceil \log_2(n-1) \rceil})$$

which is  $\leq \kappa(2n - 3)$ . The cost of writing in the  $n$  values is  $n$ . Thus, the total cost of  $n$  calls to **append** is  $\leq \kappa(2n - 3) + n = O(n)$ .

Standard formula:  
 $1 + r + \dots + r^{n-1}$  is  
 equal to  
 $(r^n - 1)/(r - 1)$ .

**Amortized costs.** Let's ascribe a cost  $c' = 2\kappa + 1$  to each **append** operation. Then, for  $n$  calls to **append**,

$$\begin{aligned} \text{aggregate true cost} &\leq n(2\kappa + 1) - 3\kappa, \\ \text{aggregate amortized cost} &= n(2\kappa + 1) \end{aligned}$$

so the fundamental inequality "aggregate true cost  $\leq$  aggregate amortized cost" is satisfied, i.e. these are valid amortized costs. We'd write this as "the amortized cost of **append** is  $O(1)$ ."

## 7.2. Potential functions

How do we come up with amortized costs? There's a systematic approach that sometimes gives useful answers. Suppose there's a function  $\Phi$ , called a *potential function*, that maps possible states of the data structure to real numbers  $\geq 0$ , and for which  $\Phi$  applied to the empty initial state is equal to 0. For an operation with true cost  $c$ , for which the state just beforehand was  $\mathcal{S}_{\text{ante}}$  and the state just after is  $\mathcal{S}_{\text{post}}$ , define the amortized cost of that operation to be

$$c' = c + \Phi(\mathcal{S}_{\text{post}}) - \Phi(\mathcal{S}_{\text{ante}}).$$

Notation:  $[x]^+$  means  $\max(0, x)$ .

**Example 7.3 (Dynamic array).** Consider the dynamic array from page 47. Define the potential function

$$\Phi = \left[ 2 \left( \begin{array}{l} \text{num. items} \\ \text{in array} \end{array} \right) - \begin{array}{l} \text{capacity} \\ \text{of array} \end{array} \right]^+ \quad (4)$$

This potential function is  $\geq 0$ , and for an empty data structure it is  $= 0$ . Now, consider the two ways that an **append** operation could play out:

- We could add the new item without needing to double the capacity. The true cost is  $c = O(1)$  and the change in potential is  $\Delta\Phi \leq 2$ , so the amortized cost is  $c + \Delta\Phi = O(1)$ .
- Alternatively we need to double the capacity. Let  $n$  be the number of items just before. The true cost is  $c = O(n)$ , to create a new array and copy  $n$  items and then write in the new value. The potential before is  $n$ , and the potential after is 2, so  $\Delta\Phi = 2 - n$ , thus the amortized cost is  $c + \Delta\Phi = O(n) + 2 - n = O(1)$ .

In both cases the amortized cost of an **append** is  $O(1)$ .

We used sloppy notation in the second case, when we wrote  $O(n) + 2 - n = O(1)$ . What we really mean is this: The true cost is  $O(n)$ , i.e. there is some constant  $\kappa$  such that the true cost is  $\leq \kappa n$  for  $n$  sufficiently large. Let the potential be  $\kappa$  times the expression in equation (4); then the amortized cost is

$$c + \Delta\Phi \leq \kappa n + \kappa(2 - n) = 2\kappa = O(1).$$

Keep this hidden constant in mind when you're reasoning about potential functions—but it's fine to write out answers using the sloppy notation, as long as you know what you're doing!

### JUSTIFICATION

We're not allowed to just write down an arbitrary formula and call it 'amortized cost'. For it to be valid, it must satisfy the fundamental inequality of amortized analysis, "aggregate true cost  $\leq$  aggregate amortized cost". We need to actually prove that  $c'$  defined by

$$c' = c + \Phi(\mathcal{S}_{\text{post}}) - \Phi(\mathcal{S}_{\text{ante}})$$

is a valid amortized cost, i.e. that inequality (3) on page 46 is satisfied. Consider a sequence of operations starting from an initially empty state,

$$\mathcal{S}_0 \xrightarrow{c_1} \mathcal{S}_1 \xrightarrow{c_2} \mathcal{S}_2 \xrightarrow{c_3} \dots \xrightarrow{c_k} \mathcal{S}_k$$

where the true costs are  $c_1, c_2, \dots, c_k$ . Then

$$\begin{aligned} & \text{aggregate amortized cost} \\ &= \left\{ -\Phi(\mathcal{S}_0) + c_1 + \Phi(\mathcal{S}_1) \right\} + \left\{ -\Phi(\mathcal{S}_1) + c_2 + \Phi(\mathcal{S}_2) \right\} \\ & \quad + \dots + \left\{ -\Phi(\mathcal{S}_{k-1}) + c_k + \Phi(\mathcal{S}_k) \right\} \\ &= c_1 + \dots + c_k - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k) \\ &= \text{aggregate true cost} - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k) \end{aligned}$$

hence

$$\begin{aligned} & \text{aggregate true cost} \\ &= \text{aggregate amortized cost} + \Phi(\mathcal{S}_0) - \Phi(\mathcal{S}_k) \\ &\leq \text{aggregate amortized cost} + \Phi(\mathcal{S}_0) \quad (\text{since } \Phi \geq 0) \\ &= \text{aggregate amortized cost} \quad (\text{since } \Phi = 0 \text{ for the initial empty state } \mathcal{S}_0). \end{aligned}$$

This 'telescoping sum' trick also appeared in the analysis of Johnson's algorithm, section 5.8 page 21

Thus the costs  $c'$  that we defined are indeed valid amortized costs.

### DESIGNING POTENTIAL FUNCTIONS

Where do potential functions come from? We can invent whatever potential function we like, and different choices might lead to different amortized costs of operation. If we are cunning in our choice of potential function, we'll end up with informative amortized costs. There is no universal recipe. Here are some suggestions:

- Sometimes we want to say “this operation may technically have worst case  $O(n)$  but morally speaking it should be  $O(1)$ ”, as with `MinList.min` on page 46 and with `DynamicArray.append`. To get amortized cost  $O(1)$ , the potential has to build up to  $n$  before the expensive operation, and drop to 0 after—or, to be precise, build up to  $\Omega(n)$  and drop to  $O(1)$ . Think of  $\Phi$  as storing up credit for the cleanup we're going to have to do.
- More generally, think of  $\Phi$  as measuring the amount of stored-up mess. If an operation increases mess ( $\Delta\Phi > 0$ ) then this mess will have to be cleaned up eventually, so we set the amortized cost to be larger than the true cost, to store up credit. An operation that does cleanup will decrease mess ( $\Delta\Phi < 0$ ), which can cancel out the true cost of the cleanup operation.
- When you invent a potential function, make sure that  $\Phi \geq 0$  and that  $\Phi = 0$  for the initial empty data structure. Otherwise you'll end up with spurious amortized costs. (See the example sheet.)

The goal in designing a potential function is to obtain useful amortized costs, to give us tight bounds on the worst case performance of sequences of operations.

Suppose we prove, using our potential function, that “operation `popmin` has amortized cost  $O(\log N)$  for a data structure containing  $\leq N$  items”. In other words, the aggregate cost of  $m$  calls to `popmin` is<sup>16</sup>  $O(m \log N)$ .

We should also look for a matching lower bound. In other words, given  $N$ , we should adduce a concrete example of a sequence of  $m$  operations that has cost  $\Omega(m \log N)$ . Typically,  $m$  will grow with  $N$ . If we can find such a lower bound, we know that the amortized costs are tight. If on the other hand we had chosen a daft potential function, we'd end up with a  $\Omega$ - $O$  gap.

Conversely, suppose we have come up with concrete examples with cost  $\Omega(m \log N)$ . If someone tells us they have devised a potential function that gives amortized cost  $O(1)$ , they must have made a mistake.

<sup>16</sup>This is an asymptotic statement in  $N$ , and  $m$  is allowed to depend on  $N$ . Remember that the fundamental inequality of amortized analysis, the inequality on page 46, is required to hold for *every* sequence of operations.

## 7.3. Heaps\*

This is a refresher of Section 4.8.

Recall the abstract data type PriorityQueue:

ADT PriorityQueue:

```
# Holds a dynamic collection of items.
# Each item has a value/payload v, and a key/priority k.

# Extract the item with the smallest key
Pair<Key, Value> popmin()

# Add v to the queue, and give it key k
push(Value v, Key k)

# For a value already in the queue, give it a new (lower) key
decreasekey(Value v, Key newk)

# Sometimes we also include methods for:
# merge two priority queues
# delete a value
# peek at the item with smallest key, without removing it
```

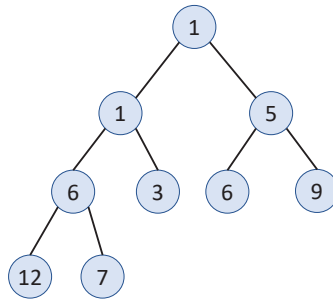
In this section we'll look at two simple implementations of a priority queue, and then introduce a third, the binomial heap.

### TWO SIMPLE PRIORITY QUEUES

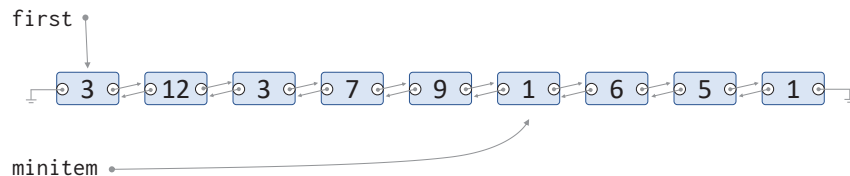
Notation:  $\lfloor x \rfloor$  is the floor of  $x$ , i.e.  $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$ .

What's the most efficient way you can think of to merge two binary heaps?

A *binary heap* is an almost-full binary tree (i.e. every level except the bottom is full), so its height is  $\lfloor \log_2 n \rfloor$  where  $n$  is the number of elements. It satisfies the heap property (each node's key is  $\leq$  its children), so it's easy to find the minimum of the entire heap. When the heap is altered by `push` or `popmin` or `decreasekey`, the change needs to be bubbled up or down the tree, which takes time  $O(\log n)$ .



Why not something simpler, a *doubly linked list*? It would make `push` and `decreasekey` very fast.



`push(v, k)` is  $O(1)$ :

just attach the new item to the front of the list, and if  $k < \text{minitem.key}$  then update `minitem`

`decreasekey(v, newk)` is also  $O(1)$ :

update  $v$ 's key, and if  $\text{newk} < \text{minitem.key}$  then update `minitem`

`popmin()` is  $O(n)$ :

we can remove `minitem` in  $O(1)$  time, but to find the new `minitem` we have to traverse the entire list.

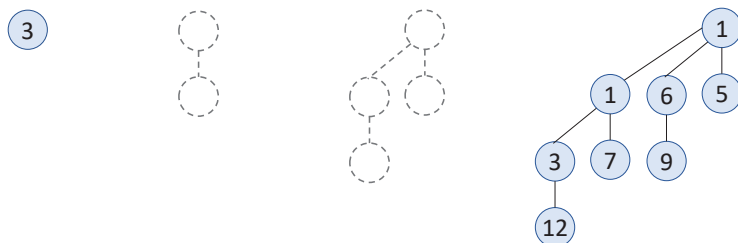


## BINOMIAL HEAP

A binomial heap is a compromise between the linked list and the binary heap. It maintains a list rather than obsessively tidying everything into a single heap, so that `push` is fast; but it still keeps some heap structure so that `popmin` is fast. It is defined as follows:

A *binomial tree of order 0* is a single node. A *binomial tree of order  $k$*  is a tree obtained by combining two binomial trees of order  $k - 1$ , by appending one of the trees to the root of the other.

A *binomial heap* is a collection of binomial trees, at most one for each tree order, each obeying the heap property i.e. each node's key is  $\leq$  those of its children. Here is a binomial heap consisting of one binomial tree of order 0, and one of order 3. (The dotted parts in the middle indicate 'there is no tree of order 1 or 2'.)



Here are some basic properties of binomial trees and heaps.

1. A binomial tree of order  $k$  has  $2^k$  nodes
2. A binomial tree of order  $k$  has height  $k$
3. In a binomial tree of order  $k$ , the root node has  $k$  children. Another word for this: the *degree* of the root is  $k$ .
4. In a binomial tree of order  $k$ , the root node's  $k$  children are binomial trees of all the orders  $k - 1, k - 2, \dots, 0$ .
5. In a binomial heap with  $n$  nodes, the 1s in the binary expansion of the number  $n$  correspond to the orders of trees contained in the heap. For example, a heap with 9 nodes (binary  $1001 = 2^3 + 2^0$ ) has one tree of order 3 and one tree of order 0.
6. If a binomial heap contains  $n$  nodes, it contains  $O(\log n)$  binomial trees, and the largest of those trees has degree  $O(\log n)$ .

The operations on binomial heaps end up resembling binary arithmetic, thanks to property 5.

`push(v, k)` is  $O(\log n)$ :

Treat the new item as a binomial heap with only one node, and merge it as described below, at cost  $O(\log n)$ , where  $n$  is the total number of nodes.

`decreasekey(v, newk)` is  $O(\log n)$ :

Proceed as with a normal binary heap, applied to the tree to which  $v$  belongs. The entire heap has  $O(n)$  nodes, so this tree has  $O(n)$  nodes and height  $O(\log n)$ , so the cost of `decreasekey` is  $O(\log n)$ .

`popmin()` is  $O(\log n)$ :

First scan the roots of all the trees in the heap, at cost  $O(\log n)$  since there are that many trees, to find which root to remove. Cut it out from its tree. Its children form a binomial heap, by property 4. Merge this heap with what remains of the original one, as described below, at cost  $O(\log n)$ .

`merge(h1, h2)` is  $O(\log n)$ :

To merge two binomial heaps, start from order 0 and go up, as if doing binary addition, but instead of adding digits in place  $k$  we merge binomial trees of order  $k$ , keeping the tree with smaller root on top. If  $n$  is the total number of nodes in both heaps together, then there are  $O(\log n)$  trees in each heap, and  $O(\log n)$  operations in total.

\* \* \*

What's the advantage of the binomial heap, over the binary heap? A simple answer is that it supports  $O(\log n)$  `merge`, whereas the natural implementation for binary heaps — concatenating the two underlying arrays and then heapifying the result — takes  $O(n)$ .

A deeper answer comes from an amortized analysis. It can be shown that even though the worst-case cost of `push` is  $O(\log n)$ , the amortized cost is  $O(1)$ . (See Example Sheet 7/8.)

## 7.4. Fibonacci heap

The Fibonacci heap is a fast priority queue, a development from the binomial heap. It was developed by Fredman and Tarjan in 1984, specifically to speed up Dijkstra’s algorithm.

### GENERAL IDEA

Here is an outline of the thinking that led to the Fibonacci heap.

1. On a graph with  $V$  vertices and  $E$  edges, Dijkstra’s algorithm might make  $V$  calls to `popmin`, and  $E$  calls to `push` and/or `decreasekey`. Since  $E$  might be as big as  $\Omega(V^2)$ , `push` and `decreasekey` are the common operations. We’d like them to be  $O(1)$ .
2. To make `push` and `decreasekey` be  $O(1)$ , they have to be lazy, and they should only ‘touch’ a small part of the data structure. The linked list implementation tells us how we can make `push` be lazy — just dump new nodes into a list, with no further tidying. Nor can `decreasekey` do any tidying — if decreasing a key leads to a heap violation, then the offending node should just be dumped into a list, with no further tidying.

The binomial heap actually uses a ‘binary counter’ structure so that, most of the time, `push` only needs to touch a few small trees, and this gives amortized cost  $O(1)$ . The Fibonacci heap uses the same trick, but inside `popmin` rather than `push`.

3. In a heap, a call to `popmin` extracts the root of a tree, and so all of its children need to be processed. Thus `popmin` has complexity  $\Omega(d)$  where  $d$  is the number of children, also called the *degree*.
4. We don’t want our heap to have any wide shallow trees — they would require `popmin` to do a lot of work, just as bad as the simple linked list implementation. We need to limit the degree of a node in the heap. We need a mechanism to ensure that the trees are deep and bushy, as they are in a binomial tree.
5. We shall from time to time do expensive housekeeping. If we’re accumulating mess that will have to be cleaned up anyway, why not just clean up as we go? The heart of the answer lies in our analysis of `heapsort` in Section 2.12. We saw that it takes time  $O(n \log n)$  to add  $n$  items to a binary heap one by one, but only  $O(n)$  to heapify them in a batch.

Doing housekeeping in batches is the big idea behind the Fibonacci heap. The housekeeping is split between `popmin` and `decreasekey`. We will invent a potential function to help us reason about the amount of housekeeping.

### HOW TO PUSH AND POPMIN

```

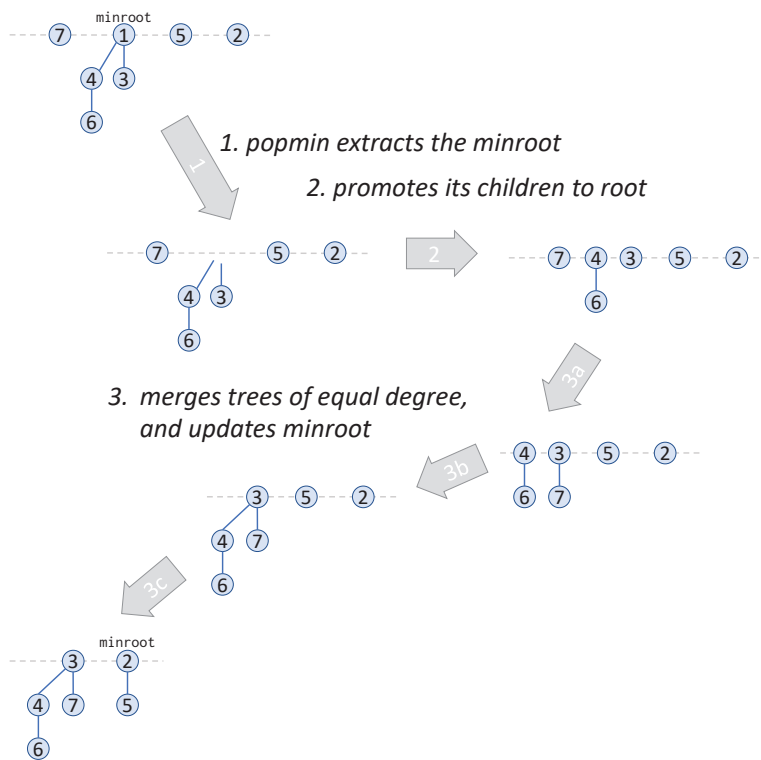
1  # Maintain a list of heaps (i.e. store a pointer to the root of each heap)
2  roots = []
3
4  # Maintain a pointer to the smallest root
5  minroot = None
6
7  def push(v, k):
8      create a new heap h consisting of a single item (v, k)
9      add h to the list of roots
10     update minroot if k < minroot.key
11
12  def popmin():
13     take note of minroot.value and minroot.key
14     delete the minroot node, and promote its children to be roots
15     # cleanup the roots
16     while there are two roots with the same degree:

```

```

17     merge those two roots, by making the larger root a child of the smaller
18     update minroot to point to the smallest root
19     return the value and key from line 13

```



In this simple version, with only `push` and `popmin`, one can show by induction that the Fibonacci heap consists at all times of a collection of binomial trees, and that after the cleanup in lines 16–17 it is a binomial heap. (See the example sheet.)

It doesn't matter how the cleanup is implemented, as long as it is done efficiently. Here is an example implementation.

```

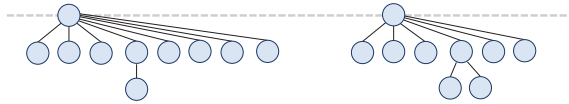
20 def cleanup(roots):
21     root_array = [None, None, ...] # empty array
22     for each tree t in roots:
23         x = t
24         while root_array[x.degree] is not None:
25             u = root_array[x.degree]
26             root_array[x.degree] = None
27             x = merge(x, u)
28         root_array[x.degree] = x
29     return list of non-None values in root_array

```

## HOW TO DECREASEKEY

If we can decrease the key of an item in-place (i.e. if its parent is still  $\leq$  the new key), then that's all that `decreasekey` needs to do. If however the node's new key is smaller than its parent, we need to do something to maintain the heap. We've already discussed why it's a reasonable idea to be lazy—to just cut such a node out of its tree and dump it into the root list, to be cleaned up in the next call to `popmin`.

There is however one extra twist. If we just cut out nodes and dump them in the root list, we might end up with trees that are shallow and wide, even as big as  $\Omega(n)$ , where  $n$  is the number of items in the heap. This would make `popmin` very costly, since it has to iterate through all `minroot`'s children.



To make `popmin` reasonably fast, we need to keep the maximum degree small. The Fibonacci heap achieves this via two rules:

1. Lose one child, and you get marked as a ‘loser’ node.
2. Lose two children, and you get dumped into the root list (and your mark is removed).

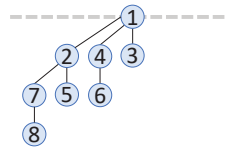
This ensures that the trees end up with a good number of descendants. Formally, we’ll show in Section 7.5 that a tree with degree  $d$  contains  $\geq 1.618^d$  nodes, and hence that the maximum degree in a heap of  $n$  items is  $O(\log n)$ .

Similarly, a binomial tree of degree  $k$  has  $2^k$  nodes, which implies a binomial heap of  $n$  items has maximum degree  $O(\log n)$ .

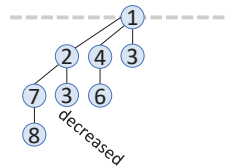
```

30 # Every node will store a flag, p.loser = True / False
31
32 def decreasekey(v, k'):
33     let n be the node where this value is stored
34     n.key = k'
35     if n violates the heap condition:
36         repeat:
37             p = n.parent
38             remove n from p.children
39             insert n into the list of roots, updating minroot if necessary
40             n.loser = False
41             n = p
42         until p.loser == False
43         if p is not a root:
44             p.loser = True
45
46 def popmin():
47     mark all of minroot's children as loser = False
48     then do the same as in the simple version, lines 13–19

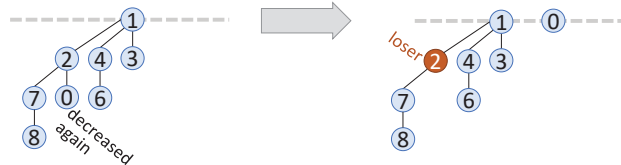
```



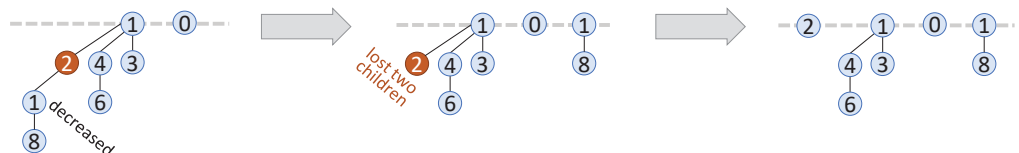
*decreasekey from 5 to 3*



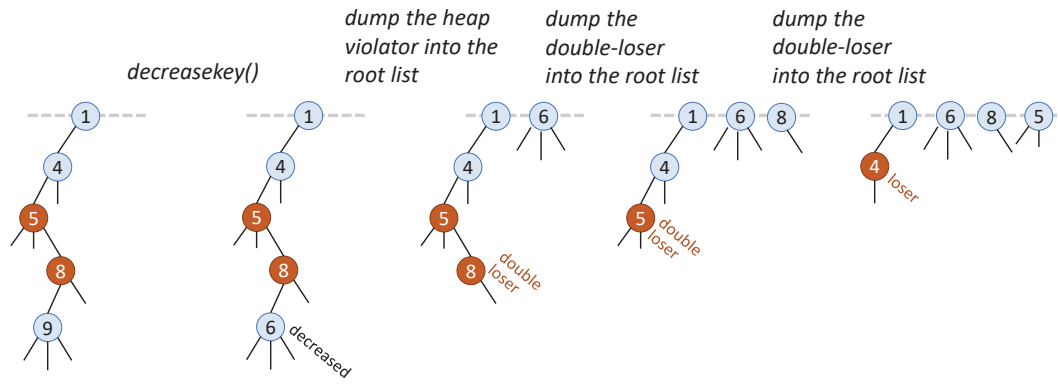
*decreasekey again to 0 — move 0 to maintain the heap*



*decreasekey from 7 to 1 — move 1 to maintain the heap — move the double-loser to root*



Here is another example of the operation of `decreasekey`, this time highlighting what happens if there are multiple loser ancestors.



## 7.5. Analysis of Fibonacci heap

We'll now compute the amortized costs of the various operations on the Fibonacci heap, using the potential function

$$\Phi = \text{number of roots} + 2(\text{number of loser nodes}).$$

Let  $n$  be the number of items in the heap, and let  $d_{\max}$  be an upper bound on the degree of any node in the heap (we'll see soon that  $d_{\max} = O(\log n)$  is suitable).

**push()** : amortized cost  $O(1)$

This just adds a new node to the root list, so the true cost is  $O(1)$ . The change in potential is  $\Delta\Phi = 1$ , so the amortized cost is  $O(1)$ .

**popmin()** : amortized cost  $O(d_{\max})$

Let's split this into two parts. First, cut out **minroot** and promote its children to the root list. There are at most  $d_{\max}$  children to promote, so the true cost is  $O(d_{\max})$ . These children get promoted to root, and maybe some of them lose the loser mark, so  $\Delta\Phi \leq d_{\max}$ . So the amortized cost for this first part is  $O(d_{\max})$ .

The second part is running cleanup. Line 21 initializes an array, in which we will store a tree of degree  $d$  in `root_array[d]`. Since  $0 \leq d \leq d_{\max}$ , an array of size  $d_{\max} + 1$  is sufficient, so it takes  $O(d_{\max})$  to initialize the array.

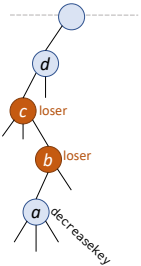
Suppose that cleanup starts with  $x$  trees, it does  $M$  merges, and ends up with  $y$  trees. Note that  $y = x - M$ , since every merge decreases the number of trees by one, and  $y \leq d_{\max} + 1$  since we end up with at most one tree in every cell of `root_array`. The true cost is

- $O(x)$  for processing each tree, lines 22–23 & 28; this is  $O(M + y)$  since  $y = x - M$
- $O(M)$  for the merges, lines 24–27
- $O(y)$  for the final copy out of `root_array`, line 29

So the total true cost is  $O(M + y)$ . And  $\Delta\Phi = -M$  since the number of roots decreases by  $M$ . Thus the amortized cost is  $O(y) = O(d_{\max})$ .

**decreasekey()** : amortized cost  $O(1)$

It takes  $O(1)$  elementary operations to decrease the key. If the node doesn't have to move, then  $\Phi$  doesn't change, so amortized cost = true cost =  $O(1)$ . If the node does have to move, the following happens:



- We move the node, call it  $a$ , to the root list. The true cost is  $O(1)$ , and  $\Phi$  increases by  $\leq 1$ : it increases by 1 if  $a$  wasn't a loser, and decreases by 1 if it was.
- Some of  $a$ 's loser ancestors  $b$  and  $c$  are moved to the root list. Say there are  $L$  of these. The true cost of moving is  $O(L)$ . And  $\Delta\Phi = L - 2L$ : the  $L$  is because of the new roots, and the  $-2L$  is because their loser marks are erased. Thus the amortized cost of this step is zero, regardless of  $L$ .
- One ancestor  $d$  might have to be marked as a loser. The true cost is  $O(1)$ , and  $\Phi$  increases by 2, so the amortized cost is  $O(1)$ .

We can see now why the potential function was chosen just so. The two operations **popmin** and **decreasekey** both include an uncontrolled number of steps,  $M$  for **popmin** and  $L$  for **decreasekey**. But these steps have already been 'paid for', by the operations that increased  $\Phi$ , so they contribute zero to the amortized cost.

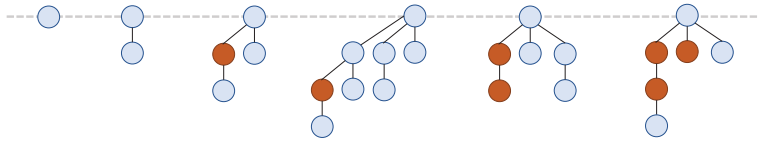
### BOUNDING THE SHAPE OF THE TREES

The amortized cost of **popmin** is  $O(d_{\max})$ , where  $d_{\max}$  is the maximum number of children of any of the nodes in the heap. The peculiar mechanism of **decreasekey** was designed to keep  $d_{\max}$  small. How small?

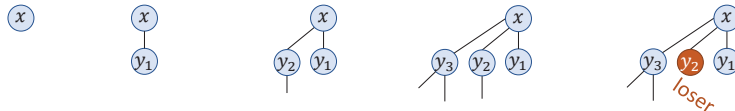
**Theorem.** *If a node in a Fibonacci heap has  $d$  children, then the subtree rooted at that node consists of  $\geq 1.618^d$  nodes. More precisely, it has  $\geq F_{d+2}$  nodes, the  $(d + 2)$ nd Fibonacci number, and  $F_{d+2} \geq \phi^d$  where  $\phi = (1 + \sqrt{5})/2 \approx 1.618$  is the golden ratio.*

$F_1 = 1, F_2 = 1,$   
 $F_3 = 2, F_4 = 3,$   
 $F_5 = 5, \dots$  The  
 general formula for  
 $F_n$  is  
 $(\phi^n - (-\phi)^{-n})/\sqrt{5}.$

It's a simple exercise to deduce from this theorem that  $d_{\max} = O(\log n)$ .



*Proof.* Consider an arbitrary node  $x$  in a Fibonacci heap, at some point in execution, and suppose it has  $d$  children, call them  $y_1, \dots, y_d$  in the order of when they last became children of  $x$ . (There may be other children that  $x$  acquired then lost in the meantime, but we're not including those.)



When  $x$  acquired  $y_2$ ,  $x$  already had  $y_1$  as a child, so  $y_2$  must have had  $\geq 1$  child seeing as it got merged into  $x$ . Similarly, when  $x$  acquired  $y_3$ ,  $y_3$  must have had  $\geq 2$  children, and so on. After  $x$  acquired a child  $y_i$ , that child might have lost a child, but it can't have lost more because of the rules of `decreasekey`. Thus, at the point of execution at which we're inspecting  $x$ ,

- $y_1$  has  $\geq 0$  children
- $y_2$  has  $\geq 0$  children
- $y_3$  has  $\geq 1$  child, ...
- $y_d$  has  $\geq d - 2$  children.

Now for some pure maths. Consider an arbitrary tree all of whose nodes obey the grandchild rule "a node with children  $i = 1, \dots, d$  has at least  $i - 2$  grandchildren via child  $i$ ". Let  $N_d$  be the smallest possible number of nodes in a subtree whose root has  $d$  children. Then

$$N_d = \underbrace{N_{d-2}}_{\text{child } d} + \underbrace{N_{d-3}}_{\text{child } d-1} + \dots + \underbrace{N_0}_{\text{child } 2} + \underbrace{N_0}_{\text{child } 1} + \underbrace{1}_{\text{the root.}}$$

Substituting in  $N_{d-1}$ , we get  $N_d = N_{d-2} + N_{d-1}$ , the defining equation for the Fibonacci sequence, hence  $N_d = F_{d+2}$ .

We've shown that the nodes in a Fibonacci heap obey the grandchild rule, therefore the number of nodes in the subtree rooted at  $x$  is  $\geq F_{d+2}$  where  $d$  is the number of children of  $x$ . □

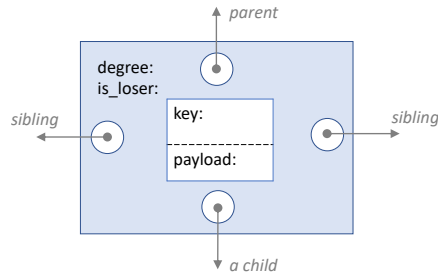
## 7.6. Implementing the Fibonacci heap\*

**This section of notes is not examinable.**

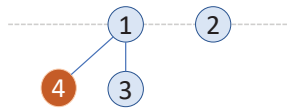
All problems in computer science can be solved by adding a layer of indirection. The steps we need for implementing a Fibonacci heap,

- slice a node out of a tree in  $O(1)$
- add a node to the root list in  $O(1)$
- merge two trees in  $O(1)$
- iterate through a root's children in  $O(\text{num. children})$

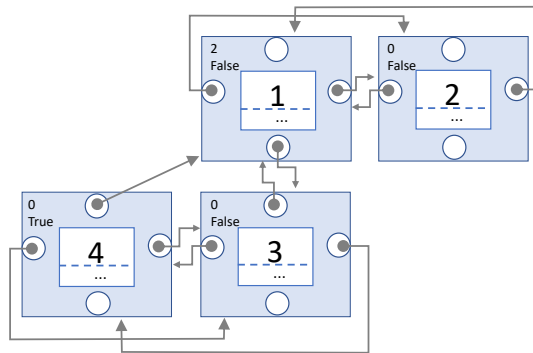
can all be achieved by keeping enough pointers around. We can use a circular doubly-linked list for the root list; and the same for a sibling list; and we'll let each node point to its parent; and each parent will point to one of its children.



Thus, this little Fibonacci heap

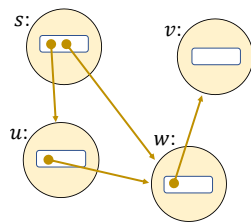


would be represented as

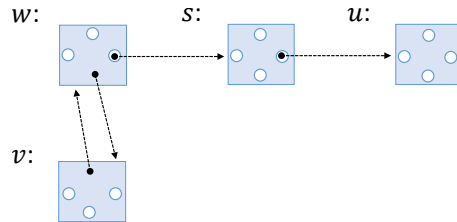


### USING A FIBONACCI HEAP IN ANOTHER ALGORITHM

When we use a Fibonacci heap as the priority queue in Dijkstra's algorithm, each vertex is doing double duty: it is simultaneously a vertex in the graph and also a node in the Fibonacci heap. How should we implement this?



vertices in a graph;  
arrows show  
graph-neighbours



the same vertices;  
arrows show  
heap-relationships

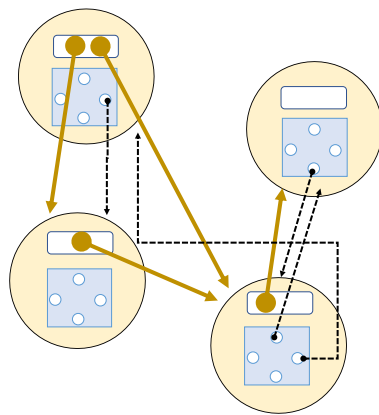
More concretely, what are the data types in the code snippet below? When we iterate through  $v$ 's neighbours and get a vertex  $w$ , how do we figure out where  $w$  is in the heap so that we can call `decreasekey` on it?



```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)
    while not toexplore.is_empty():
        v = toexplore.popmin()           # get a vertex from the FibHeap
        for (w,c) in v.neighbours:     # look up a vertex's neighbours
            dist_w = v.distance + c
            ...
            toexplore.decreasekey(w, dist_w) # decrease FibHeap priority of w
```

### CLASS-ORIENTED ARCHITECTURE

If `Vertex` represents a vertex in the graph, and `Node` represents a node in a Fibonacci heap, we can declare that each `Vertex` *uses* a `Node` to store its heap-related pointers. These pointers should be to `Vertex` objects (so that `popmin` can return a `Vertex`).



Here is a pseudocode outline of the classes involved. As well as `Vertex` and `Node`, there is an interface `AsFibNode` so that the Fibonacci heap routines can ‘see through’ a `Vertex` to get to its `Node` object, without needing to know anything about how `Vertex` is implemented.

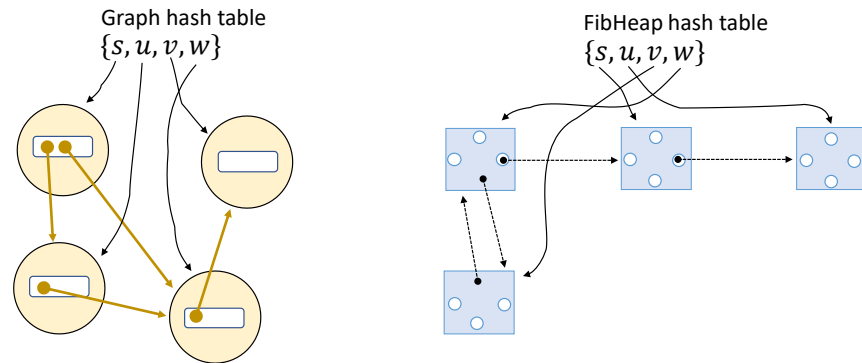
```
class Graph:
    class Vertex implements FibHeap.AsFibNode<Vertex>:
        List<Pair<Vertex, float>> neighbours
        float distance
        Vertex? come_from
        FibHeap.Node<Vertex> pqn
        FibHeap.Node<Vertex> as_fib_node(): return pqn
    def compute_shortest_paths_from(Vertex s):
        ...

class FibHeap<T extends FibHeap.AsFibNode<T>>:
    class Node<T>:
        float key
        int degree
        T parent, child, prev_sibling, next_sibling
    interface AsFibNode<T>:
        Node<T> as_fib_node()

    List<T> roots
    def T popmin(): ...
    def push(T value, float key): ...
    def decrease_key(T value, float newkey): ...
```

## DYNAMIC ARCHITECTURE

Another approach is to label each vertex by an id, and to use two hash tables: one to look up a vertex given its id, and another to look up a heap node given the id. Often there is a natural id to use for each vertex, for example, the node id in an OpenStreetMap graph, or the primary key of the graph as stored in a database.



The advantage of this architecture is that it keeps the code for graph-traversal entirely separate from that for the priority queue. If we want to run several different algorithms on a graph, each with its own storage requirements, it doesn't make sense to have a `Vertex` class that embodies a particular storage.

```
class Graph:
    HashMap<Id, Vertex> vertices
    class Vertex:
        Id id
        List<Pair<Vertex, float>> neighbours
        float distance
        Vertex? come_from
    def compute_shortest_paths_from(Id s):
        ...

class FibHeap:
    HashMap<Id, FibHeapNode> nodes
    class FibHeapNode:
        Id id
        float key
        int degree
        FibHeapNode parent, child, prev_sibling, next_sibling

    List<FibHeapNode> roots
    def Id popmin(): ...
    def push(Id node, float key): ...
    def decrease_key(Id node, float newkey): ...
```

## 7.7. Disjoint sets



The DisjointSet data structure (also known as union-find or merge-find) is used to keep track of a dynamic collection of items in disjoint sets. We used it in Kruskal's algorithm for finding a minimum spanning tree: the items corresponded to vertices of a graph, and we used sets to track which vertices we had joined together into forest fragments.

## ADT DisjointSet:

```
# Holds a dynamic collection of disjoint sets

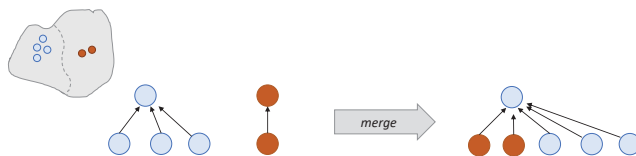
# Return a handle to the set containing an item.
# The handle must be stable, as long as the DisjointSet is not modified.
Handle get_set_with(Item x)

# Add a new set consisting of a single item (assuming it's not been added already)
add_singleton(Item x)

# Merge two sets into one
merge(Handle x, Handle y)
```

The specification doesn't say what a handle *is*, only that handles don't change unless the DisjointSet is modified (by either `add_singleton` or `merge`). In practice, we might use a representative element from each set as the set's handle.

## IMPLEMENTATION 1: FLAT FOREST

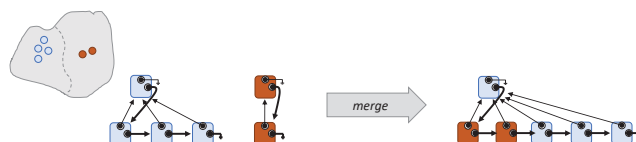


To make `get_set_with` fast, we could make each item point to its set's handle.

`get_set_with()` is just a single lookup.

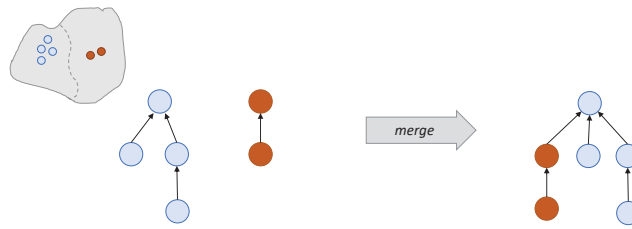
`merge()` needs to iterate through each item in one or other set, and update its pointer. This takes  $O(n)$  time, where  $n$  is the number of items in the DisjointSet.

To be able to iterate through the items, we could store each set as a linked list:



A smarter way to merge is to keep track of the size of each set, and pick the smaller set to update. This is called the *weighted union* heuristic. In the Example Sheet you'll show that the aggregate cost of any sequence of  $m$  operations on  $\leq N$  elements (i.e.  $m$  operations of which  $\leq N$  are `add_singleton`) is  $O(m + N \log N)$ , asymptotic in  $N$ .

## IMPLEMENTATION 2: DEEP FOREST



To make `merge` faster, we could skip all the work of updating the items in a set, and just build a deeper tree.

`merge()` attaches one root to the other, which only requires updating a single pointer.

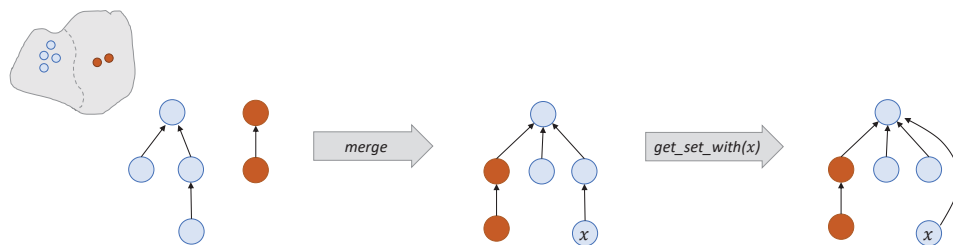
`get_set_with()` needs to walk up the tree to find the root. This takes  $O(h)$  time, where  $h$  is the height of the tree.

To keep  $h$  small, we can use the same idea as for the flat forest: keep track of the rank of each root (i.e. the height of its tree), and always attach the lower-rank root to the higher-rank. If the two roots had ranks  $r_1$  and  $r_2$  then the resulting rank is  $\max(r_1, r_2)$  if  $r_1 \neq r_2$ , and  $r_1 + 1$  if  $r_1 = r_2$ . This is called the *union by rank* heuristic. It can be shown that the aggregate cost of any sequence of  $m$  operations on  $\leq N$  elements is  $O(m \log N)$ , asymptotic in  $N$ .

Aggregate cost: see  
CLSR exercises  
21.3-3 and 21.4-4

## IMPLEMENTATION 3: LAZY FOREST

We'd like the forest to be flat so that `get_set_with` is fast, but we'd like to let it get deep so that `merge` can be fast. Here's a way to get the best of both worlds, inspired by the Fibonacci heap—defer cleanup until you actually need the answer.



`merge()` is as for the deep forest.

`get_set_with(x)` does some cleanup. It walks up the tree once to find the root, and then it walks up the tree a second time to make  $x$  and all the intermediate nodes be direct children of the root.

This method is called the *path compression* heuristic. We won't adjust the stored ranks during path compression, and so rank won't be the exact height of the tree, just an upper bound on the height. (If we wanted to know the actual tree height we'd have to compute it, which would be too much work.)

It can be shown that with the lazy forest the cost of  $m$  operations on  $\leq N$  items is  $O(m\alpha_N)$  where  $\alpha_N$  is an integer-valued monotonically increasing sequence, related to the Ackerman function, which grows extremely slowly:

$$\alpha_N = 0 \quad \text{for } N = 0, 1, 2$$

$$\alpha_N = 1 \quad \text{for } N = 3$$

$$\alpha_N = 2 \quad \text{for } N = 4, 5, 6, 7$$

$$\alpha_N = 3 \quad \text{for } 8 \leq N \leq 2047$$

$$\alpha_N = 4 \quad \text{for } 2048 \leq N \leq 10^{80}, \text{ more than there are atoms in the observable universe.}$$

For practical purposes,  $\alpha_N$  may be ignored in the  $O$  notation, and therefore the amortized cost per operation is  $O(1)$ .

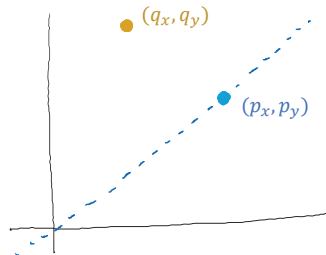
## 8. Geometrical algorithms

### 8.1. Segment intersection

Do two line segments intersect? This is a simple question, and a good starting point for many more interesting questions in computational geometry.



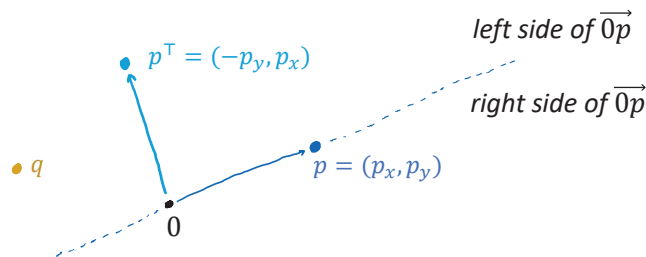
Let's start with a simpler problem. Is the point  $q$  above or below the dotted line? The answer doesn't need anything more than basic school maths: if  $q_y > (p_y/p_x)q_x$  then it's above.



But it's easy to get tangled up thinking through all the cases (e.g. if  $p_x > 0$  and  $p_y < 0$  do I need to flip the sign?) We can use slightly cleverer maths, namely dot products, to get a cleaner answer:

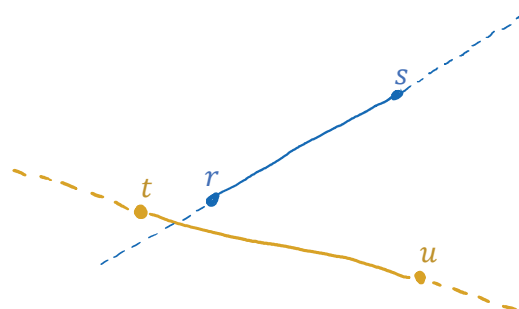
Let  $p^\top = (-p_y, p_x)$ . If we rotate the vector  $\vec{0p}$  by  $90^\circ$  anticlockwise, we get  $\vec{0p}^\top$ . Now, the sign of  $p^\top \cdot q$ , i.e. of  $-p_y q_x + p_x q_y$ , tells us which side of the dotted line  $q$  is on. The dotted line is called the *extension* of  $\vec{0p}$ .

- $p^\top \cdot q > 0$  :  $q$  is on the left, as you travel along the dotted line in direction  $\vec{0p}$
- $p^\top \cdot q = 0$  :  $q$  is on the line itself
- $p^\top \cdot q < 0$  :  $q$  is on the right



This gives us all the tests we need to decide if two line segments  $\overline{rs}$  and  $\overline{tu}$  intersect:

1. If  $t$  and  $u$  are both on the same side of the extension of  $\overline{rs}$ , i.e. if  $(s-r)^\top \cdot (t-r)$  and  $(s-r)^\top \cdot (u-r)$  have the same sign, then the two line segments don't intersect.
2. Otherwise, if  $r$  and  $s$  are both on the same side of the extension of  $\overline{tu}$ , then the two line segments don't intersect.
3. Otherwise, they do intersect.



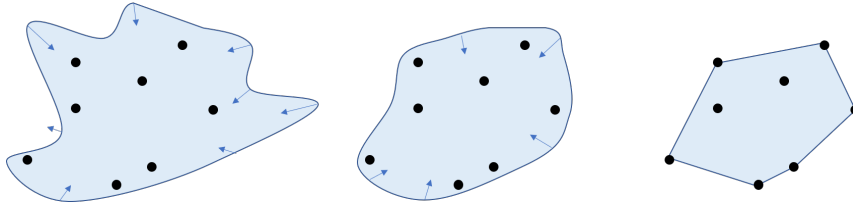
Well-written code should test all the boundary cases, e.g. when  $r = s$  or when  $t$  or  $u$  lie on the extension of  $\vec{rs}$ . It is however a venial sin to test equality of floating point numbers<sup>17</sup>, because of the vagaries of finite-precision arithmetic, and so the question “How should my segment-intersection code deal with boundary cases?” depends on “What do I know about my dataset and what will my segment-intersection code be used for?” The example sheet asks you to consider a case in detail.

---

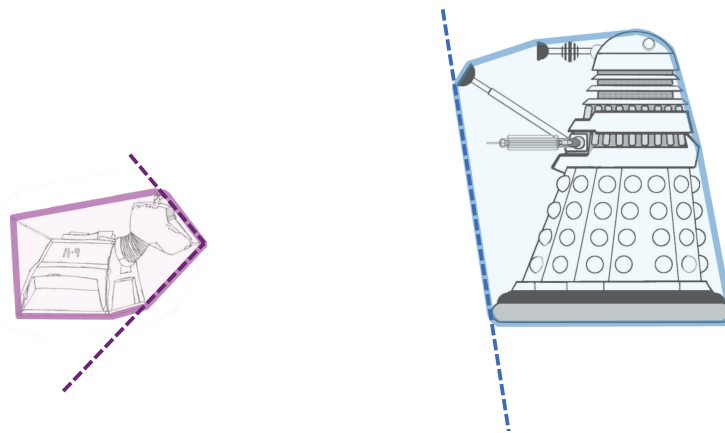
<sup>17</sup>In Python,  $0.1 + 0.1 + 0.1 - 0.3$  gives  $5.55 \times 10^{-17}$ , which is not equal to 0

## 8.2. Jarvis's march

Given a collection of points, a *convex hull* is what you get if you throw a lasso around all the points, then pull it tight.



Convex hulls are used, among other things, for collision detection. Given a straight line segment on the convex hull of a complex object, all points in that object must lie on the same side of the line. If we can get away with checking the sides of just a few lines, we can drastically speed up checks for collision.

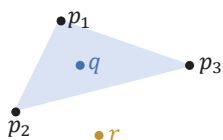


### MATHEMATICAL DEFINITION

It's worthwhile to define convex hull properly, even though for this section of the course we'll be using the informal definition (and in fact to turn the informal idea of a lasso pulled tight into precise symbolic logic is rather hard!) Formally speaking, given a collection of points  $p_1, \dots, p_n$ , a *convex combination* is any vector

$$q = \alpha_1 p_1 + \dots + \alpha_n p_n \quad \text{where } \alpha_i \geq 0 \text{ for all } i, \text{ and } \sum_{i=1}^n \alpha_i = 1.$$

The *convex hull* of a collection of points is the set of all convex combinations.

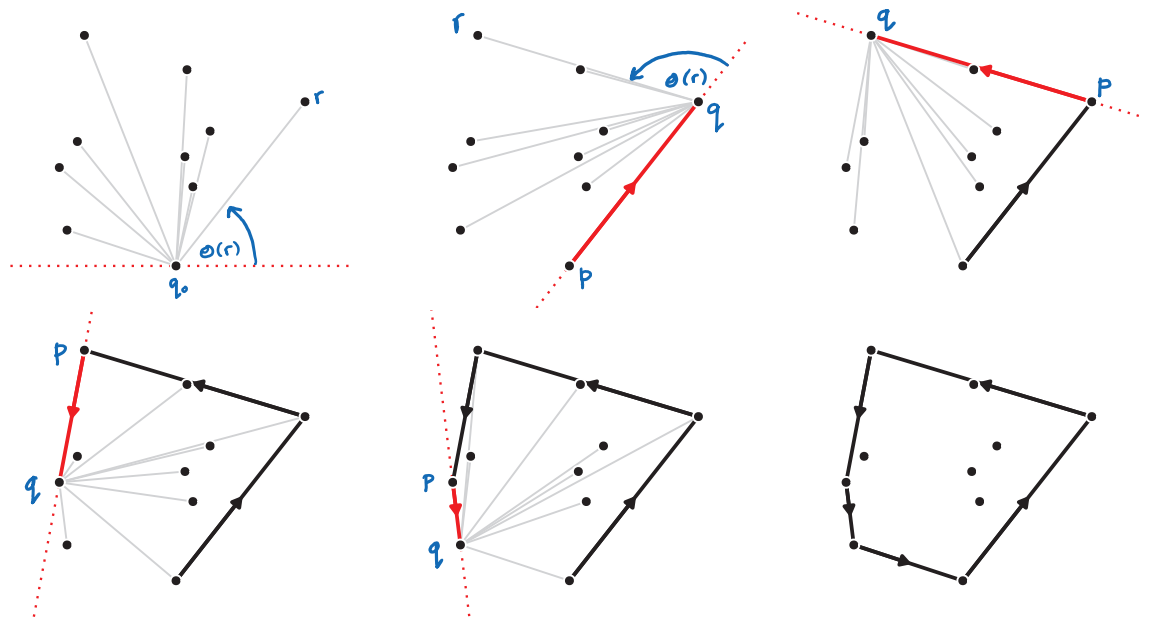


$$q = 0.6p_1 + 0.16p_2 + 0.24p_3: \text{ is a convex combination}$$

$$r = -0.5p_1 + 0.9p_2 + 0.6p_3: \text{ not a convex combination}$$

### THE ALGORITHM

Here is an algorithm to compute the convex hull of a collection of points  $P$ . It is due to Jarvis (1973), and was discovered independently by Chand and Kapur (1970). (Formally, this algorithm finds the corner points of the convex hull, i.e. the points  $p \in P$  such that  $p \notin \text{convexhull}(P \setminus \{p\})$ . But in this part of the course we shall use intuition rather than definitions.)



```

1  let  $q_0$  be the point with lowest  $y$ -coordinate
2  (in case of a tie, pick the one with the largest  $x$ -coordinate)
3
4  draw a horizontal (left→right) line through  $q_0$ 
5  for all other points  $r \in P$ :
6      find the angle  $\theta(r)$  from the horizontal line to  $\overrightarrow{q_0 r}$ , measured  $\odot$ 
7  let  $q_1$  be the point with the smallest angle
8  (in case of a tie, pick the one furthest from  $q_0$ )
9
10  $h = [q_0, q_1]$ 
11 repeatedly:
12     let  $p$  and  $q$  be the last two points added to  $h$  respectively
13     for all other points  $r \in P$ :
14         find the angle  $\theta(r)$  from the extended  $\overrightarrow{pq}$  line to  $\overrightarrow{qr}$ , measured  $\odot$ 
15         pick the point with the smallest angle, and append it to  $h$ 
16         (in case of a tie, pick the one furthest from  $q$ )
17     stop when we return to  $q_0$ 

```

### ANALYSIS

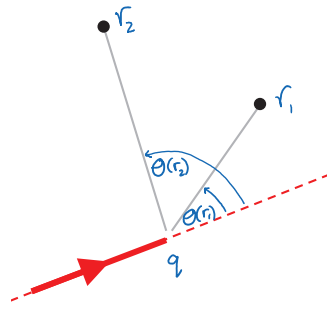
At each step of the iteration, we search for the point  $r \in P$  with the smallest angle  $\theta(r)$ , thus the algorithm takes  $O(nH)$  where  $n$  is the number of points in  $P$  and  $H$  is the number of points in the convex hull. (As with Ford-Fulkerson, running time depends on the content of the data, not just the size.)

**Performance note.** The algorithm says “find the point  $r$  with the smallest angle  $\theta(r)$ ”. We could use trigonometry to compute  $\theta$  — but there is a trick to make this faster. (Faster in the sense of ‘games get more frames per second’, but no difference in the big- $O$  sense.) If all the points we’re comparing are on the same side of dotted line, as they are at all steps of Jarvis’s march, then

$$\theta(r_1) < \theta(r_2) \iff r_2 \text{ is on the left of the extended line } \overrightarrow{qr_1}$$

and we’ve seen how to compute this true/false value with just some multiplications and additions.



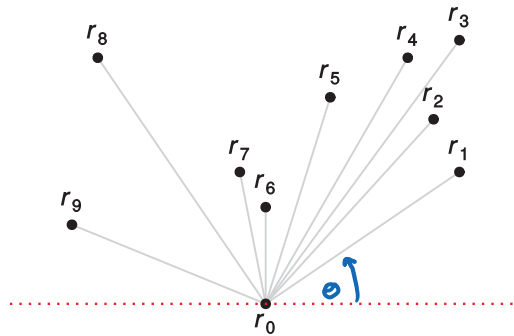


\* \* \*

Jarvis's march is very much like selection sort: repeatedly find the next item that goes into the next slot. In fact, most convex hull algorithms resemble some sorting algorithm.

### 8.3. Graham's scan

Here is another algorithm for computing the convex hull, due to Ronald Graham (1972). It builds up the convex hull by scanning through all the points in a fan, backtracking when necessary.



The code given here doesn't deal correctly with some boundary cases. The example sheet asks you to spot the problems.

```

1 let  $r_0$  be the point with lowest  $y$ -coordinate
2 (in case of a tie, pick the one with the largest  $x$ -coordinate)
3
4 draw a horizontal (left→right) line through  $r_0$ 
5 for all other points  $r$ :
6     find the angle from the horizontal line to  $\overrightarrow{r_0 r}$ , measured  $\odot$ 
7 let  $r_1, \dots, r_{n-1}$  be the sorted list of points, lowest angle to highest
8
9  $h = [r_0, r_1]$ 
10 for each  $r_i$  in the sorted list of points,  $i \geq 2$ :
11     if  $r_i$  isn't on the left of the extension of the final segment of  $h$ :
12         # backtrack
13         repeatedly delete points from the end of  $h$  until  $r_i$  is
14         append  $r_i$  to  $h$ 

```

The diagram on the next page shows how the algorithm proceeds. Each row in the diagram shows it working on a new  $r_i$ , and the side-by-side panels show steps in backtracking.

#### ANALYSIS

The initial sort takes time  $O(n \log n)$ , where  $n$  is the number of points in the set. During the scan, each point  $r_i$  is added to the list once, and it can be removed at most once, so the loop is  $O(n)$ .

To save some trigonometrical calculations, the same trick as in Section 8.2 works.

