7. Advanced data structures

7.1. Amortized analysis

For some data structures, looking at the worst-case run time per operation may be unduly pessimistic. This is especially true in two situations:

- Some data structures are designed so that most operations are cheap, but some of them occasion expensive internal 'housekeeping'. For example, Java's ArrayList is a dynamically-sized array which increases its capacity by 50% when it becomes full; if the array has n elements then adding a new element might be $O(1)$, or it might be $O(n)$ if a capacity increase is needed.
- Sometimes we want to know the aggregate cost of a sequence of operations, not the individual costs of each operation. For example, in analysing the running time of Dijkstra's algorithm in Section 5.4, we said there are $O(V)$ calls to popmin and $O(E)$ calls to either push or decreasekey; as far as this analysis is concerned all that matters is total running time for all these $O(V+E)$ operations, not the worst case for each.

Amortized analysis is a tool that makes it simpler to analyse aggregate costs, by using an accounting trick for attributing the cost of expensive housekeeping operations to all the other operations. First though, we'll remind ourselves about aggregate costs.

AGGREGATE ANALYSIS

Aggregate analysis just means 'work out the worst-case total cost of a sequence of operations'.

Example 7.1. When we analysed the running time of the breadth-first search code snippet on page 5, we argued "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 $O(V+E)$."

This is an example of aggregate analysis. To be more formal about it, we might say that a run of the algorithm is a sequence comprising three basic types of operations, init_vertex on line 4, popright on line 9, and process_edge on lines 11–13. The sequence comprises V cases of init_vertex, followed by $\leq V$ cases of popright mixed with $\leq E$ cases of process_edge. Each of these three basic operation types is $O(1)$, and so the aggregate cost of the sequence is $O(V+E)$.

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 place*, 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.

Example 7.2 (Dynamic array). 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.)

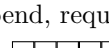
initially empty

append

append, requires doubling capacity

append, requires doubling capacity

append



append, requires doubling capacity

Let's assume that the cost of doubling capacity from m to $2m$ is m. After adding n elements, the total cost from all of the doubling is

$$1 + 2 + \cdots + 2^{\lfloor \log_2(n-1) \rfloor}$$

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

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

AMORTIZED COST

When we look at the aggregate cost of a sequence of operations, we can (if it's convenient for the analysis) reassign costs. Suppose there is a sequence of k operations, whose true costs are $c_1, c_2, \ldots, c_k$. Suppose we invent $c'_1, c'_2, \ldots, c'_k$ such that

$$c_1 + \cdots + c_j \;\leq\; c'_1 + \cdots + c'_j \quad \text{for all } j \leq k\,;$$

We're not changing the actual cost of an operation. We're just using an accounting trick to *ascribe* the total cost differently.

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

$$\begin{array}{l}\text{aggregate true cost of a}\\ \text{sequence of operations}\end{array} \;\leq\; \begin{array}{l}\text{aggregate amortized cost}\\ \text{of those operations.}\end{array} \tag{3}$$

Note that this inequality must apply to any sequence of operations, no matter how short or long. It is not an asymptotic (big-O) statement.

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 tells us: for any sequence comprising n_1 push operations and n_2 popmin operations,

$$\text{worst-case aggregate true cost} \;\leq\; n_1 O(1) + n_2 O(\log N) \;=\; O(n_1 + n_2 \log N).$$

Example 7.3 (Dynamic array). Consider again the dynamically-sized array from Example 7.2. We argued that n calls to append cost $\leq 3n - 3$. If we assign a cost $c' = 3$ to each append operation, we ensure equation (3). Another way of putting this: the amortized cost of append is $O(1)$.

Example 7.4 (Accounting costs). Some items, like fridges, are expensive to dispose of. If it costs £x to buy the item, and £y to dispose of it, I could (for accounting purposes) distribute the total cost £$(x + y)$ in various ways:

- pay £x when I acquire the fridge, pay £y when I dispose of it
- pay £$(x + y)$ when I acquire the fridge, £x to the shop and £y into my bank account, and pay nothing more when I dispose of it
- pay £$(x + y/2)$ when I acquire the fridge, pay £$y/2$ more when I dispose of it

Indeed, garbage collection is a perfect example of amortized analysis. The cost of a single large expensive storage reorganization can be ascribed to each of the elementary allocations that made it necessary.

but of course I can't get away with paying less than £x when I acquire it, since the shop won't let me get away with it! This requirement, that I've paid enough to cover the total true costs *at each stage of the sequence*, is the interpretation of 'for all $j \leq k$' in the definition of amortized cost.

THE POTENTIAL METHOD

How do we come up with amortized costs? For some data structures, there's a systematic approach.

Suppose there's a function Φ, called a *potential function*, that maps possible states of the data structure to real numbers ≥ 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

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

Think of Φ as measuring the amount of stored-up mess. If the operation increases mess then $c' > c$, and if it decreases mess then $c' < c$.

We need to actually prove that c' is a valid amortized cost, i.e. that inequality (3) is satisfied. Consider a sequence of operations

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

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

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

It's convenient to set $\Phi = 0$ for an empty data structure. Then, for any sequence of operations starting from empty,

$$\text{aggregate true cost} \;\leq\; \text{aggregate amortized cost}$$

i.e. the costs c' that we defined are indeed valid amortized costs.

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

Example 7.5 (Dynamic array). Define the potential function

$$\Phi = \left[2\begin{pmatrix}\text{num. items}\\ \text{in array}\end{pmatrix} - \begin{matrix}\text{capacity}\\ \text{of array}\end{matrix}\right]^+ \tag{4}$$

This potential function is ≥ 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 $O(1) + 2 = O(1)$.
- 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 $O(n) + 2 - n = O(1)$.

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

We used sloppy notation in this argument, to say that $\Delta\Phi = 2 - n$ cancels out $O(n)$ true cost. What we really mean is this: The true cost is $c = O(n)$, i.e. there is some constant κ such that the true cost is $c \leq \kappa n$ for n sufficiently large. Let the potential be κ times the expression in equation (4); then the change in potential genuinely does cancel out $O(n)$ for n sufficiently large.

UNIQUENESS AND TIGHTNESS

We can invent whatever potential function we like, and different choices might lead to different amortized costs of operation. If we're lucky we can find a matching big-Ω lower bound, and then we know we can't do better. See the example sheet.

7.2. Binomial heap

Recall the abstract data type PriorityQueue[11]:

```
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
```

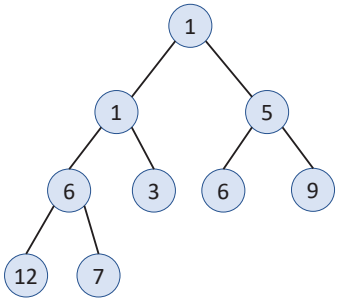
In this section we'll look at two simple implementations of a priority queue, and then introduce a third, the binomial heap.

TWO SIMPLE PRIORITY QUEUES

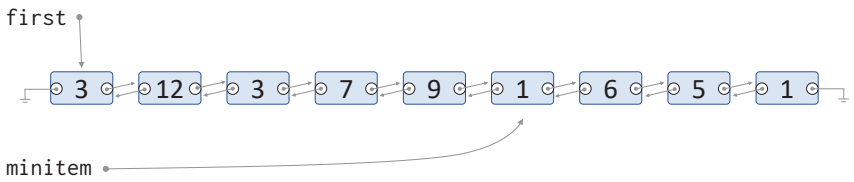
A *binary heap* is an almost-full binary tree (i.e. every level except the bottom is full), so its height is $\lfloor \log_2 n \rfloor$ where n is the number of elements. It satisfies the heap property (each node's key is $\leq$ its children), so it's easy to find the minimum of the entire heap. When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

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?



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 < minitem.key then update minitem

decreasekey(v, newk) is also $O(1)$:
: update v's key, and if newk < minitem.key then update minitem

[11] The code for Dijkstra's algorithm in Section 5.4 also needs to test whether a value is already in the PriorityQueue. We'll assume that this piece of information is stored with each value, so it can be accessed in $O(1)$ time. We won't include this test in the ADT.

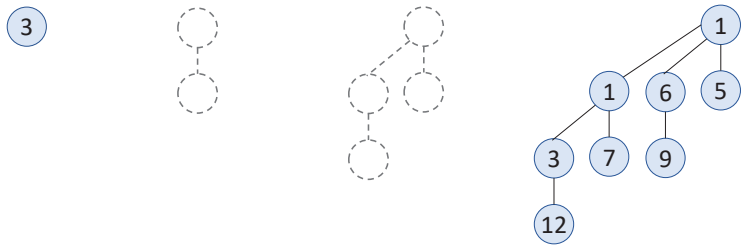
popmin() is $O(n)$:
: we can remove minitem in $O(1)$ time, but to find the new minitem we have to traverse the entire list.

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 $k-1$, $k-2$, ..., 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.)

7.3. Fibonacci heap

The Fibonacci heap is a fast priority queue. It was developed by Fredman and Tarjan in 1984, specifically to speed up Dijkstra's algorithm. 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 $\Omega(V^2)$, we want an implementation of PriorityQueue with very fast push and decreasekey.

GENERAL IDEA

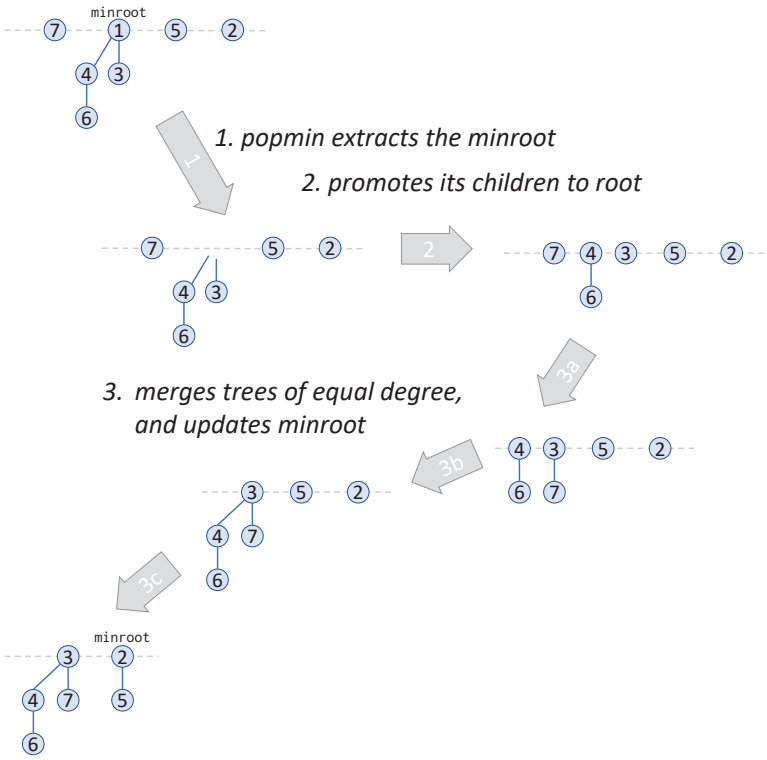
The general idea behind the Fibonacci heap is that we should be lazy while we're doing push and decreasekey, only doing $O(1)$ work, and just accumulating a collection of unsorted items, rather like the linked list implementation from Section 7.2. We'll only do cleanup (into something resembling a binomial heap) on calls to popmin.

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 2.13. 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 cleanup in batches is the first big idea behind the Fibonacci heap.

The second big idea is that for decreasekey to be $O(1)$, it needs to be able to get away with only touching a small part of the data structure. In the binomial heap, we used a 'binary counter' structure so that push only needs to touch a few small trees (most of the time), and this gave us amortized cost $O(1)$. The Fibonacci heap uses a clever trick so that decreasekey can do the same.

7.3.1. IMPLEMENTATING 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 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.

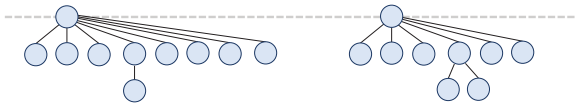
It doesn't matter how the cleanup is implemented, as long as it is done efficiently. Here is an example implementation.

```
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
```

7.3.2. IMPLEMENTING 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.



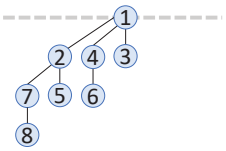
To make popmin reasonably fast, we need to keep the maximum degree small. The Fibonacci heap achieves this via two rules:

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).

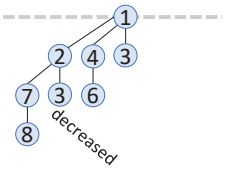
This ensures that the trees end up with a good number of descendants. Formally, we'll show in Section 7.4 that a tree with degree d contains $\geq 1.618^d$ nodes, and hence that the maximum degree in a heap of n items is $O(\log n)$.

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

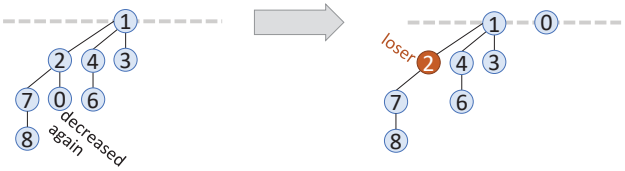
```
30 # Every node will store a flag, p.loser = True / False
31
32 def decreasekey(v, k'):
33     let n be the node where this value is stored
34     n.key = k'
35     if n violates the heap condition:
36         repeat:
37             p = n.parent
38             remove n from p.children
39             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
```



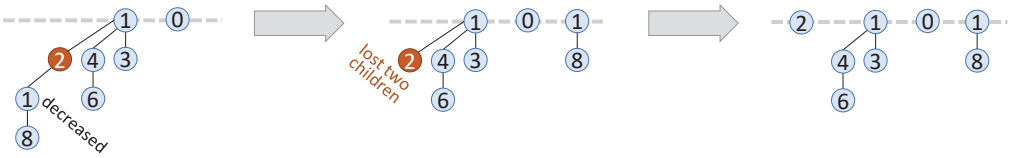
decreasekey from 5 to 3



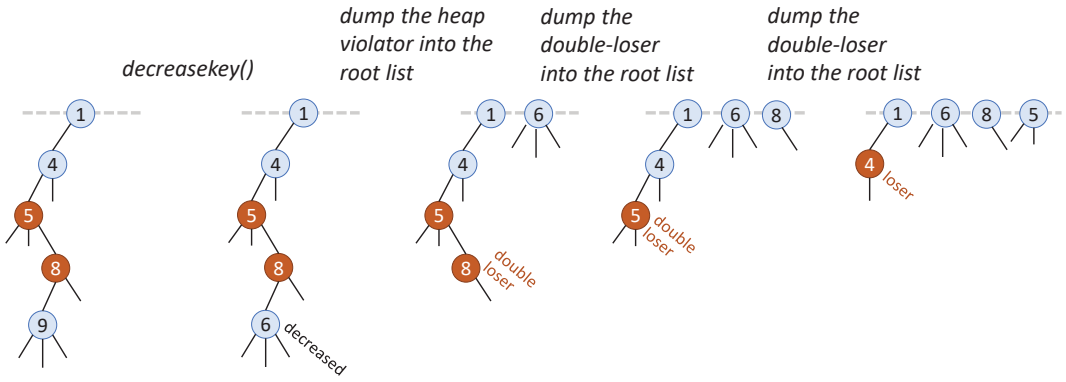
decreasekey again to 0 — move 0 to maintain the heap



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

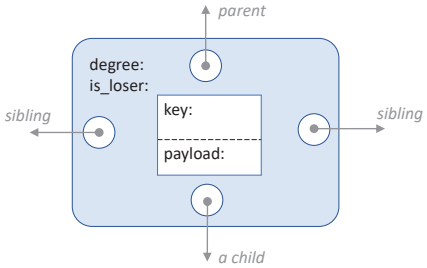


Here is another example of the operation of decreasekey, this time highlighting what happens if there are multiple loser ancestors.

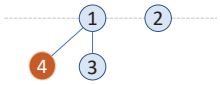


7.3.3. NUTS AND BOLTS OF THE IMPLEMENTATION

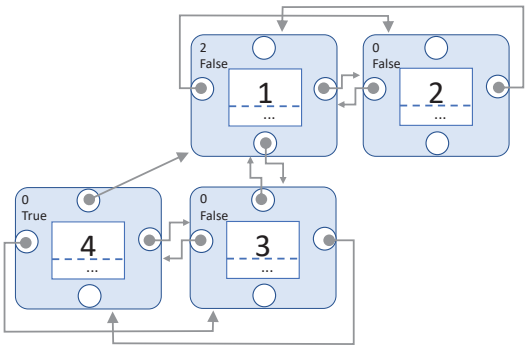
All problems in computer science can be solved by adding a layer of indirection. All the steps we've described (cutting nodes out of their tree, promoting nodes to the root list, merging trees) can be achieved in $O(1)$ time by keeping enough pointers around. We can use a circular doubly-linked list 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



7.4. Analysis of Fibonacci heap

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

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

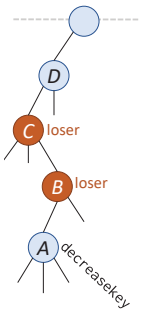
Let n be the number of items in the heap, and let $d_{\max}$ be an upper bound on the degree of any node in the heap (we'll see soon that $d_{\max} = O(\log n)$ is suitable).

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 $\Delta\Phi \le d_{\max}$. So the amortized cost for this first part is $O(d_{\max})$.

: The second part is running cleanup. Line 21 initializes an array, and size $d_{\max}+1$ will do, so this is $O(d_{\max})$. Suppose that cleanup does t merges and ends up with d trees left in root_array; there must have been $t+d$ trees to start with, so the total work in lines 22–28 is $t+d$, and Φ decreases by t; so the total amortized cost is $O(d)$, and $d \le d_{\max}$.

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



: If the node does have to move, the following happens. (i) We move the node, call it A, to the root list. The true cost is $O(1)$, and Φ increases by ≤ 1: it increases by 1 if A wasn't a loser, and decreases by 1 if it was. (ii) Some of A's loser ancestors B and C are moved to the root list. For each of these, the true cost of moving is $O(1)$, and Φ increases by 1 because there's a new root, and decreases by 2 because the node gets marked as not as loser. Thus the amortized cost of this part is zero, regardless of the nuber of loser ancestors. (iii) One ancestor D might have to be marked as a loser. The true cost is $O(1)$, and Φ increases by 2. In total the amortized cost is $O(1)$.

We can see now why the potential function was chosen just so: popmin and decreasekey both include a variable number of steps, but each step has already been 'paid for', so they contribute zero to the amortized cost.

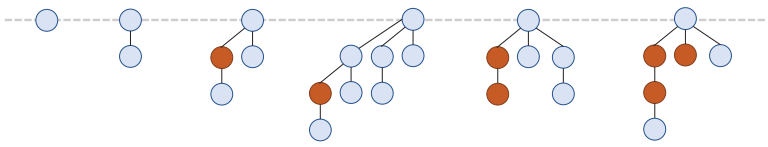
BOUNDING THE SHAPE OF THE TREES

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?

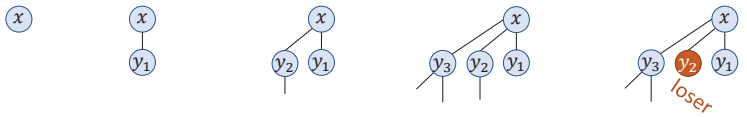
Theorem. *If a node in a Fibonacci heap has d children, then the subtree rooted at that node consists of $\ge 1.618^d$ nodes. More precisely, it has $\ge F_{d+2}$ nodes, the $(d+2)$nd Fibonacci number, and $F_{d+2} \ge \phi^d$ where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio.*

$F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, ... 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 $y_1, \ldots, y_d$ 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 ≥ 1 child seeing as it got merged into x. Similarly, when x acquired y_3, y_3 must have had ≥ 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,

$$\begin{aligned} y_1 &\text{ has } \geq 0 \text{ children} \\ y_2 &\text{ has } \geq 0 \text{ children} \\ y_3 &\text{ has } \geq 1 \text{ child}, \ldots \\ y_d &\text{ has } \geq d-2 \text{ children.} \end{aligned}$$

Now for some pure maths. Consider an arbitrary tree all of whose nodes obey the grandchild rule "a node with children $i = 1, \ldots, d$ 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

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

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. □

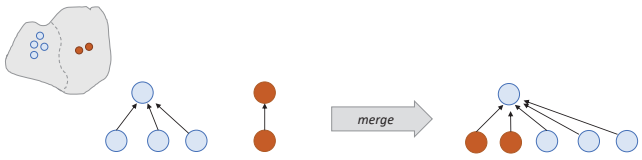
7.5. Disjoint sets

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

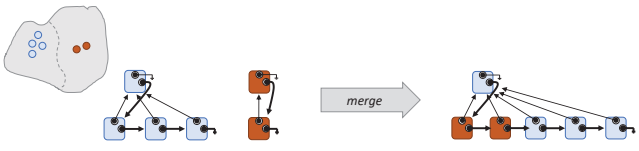


To make get_set_with fast, we could make each item point to its set's handle.

get_set_with() is just a single lookup.

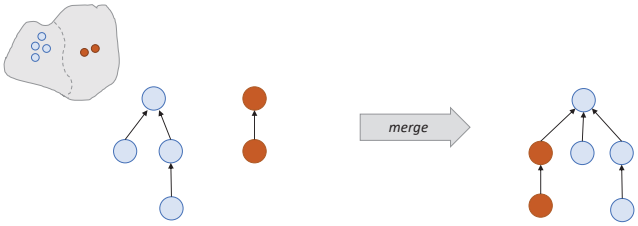
merge() needs to iterate through each item in one or other set, and update its pointer. This takes $O(n)$ time, where n is the number of items in the DisjointSet.

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 n elements (i.e. m operations of which n are add_singleton) is $O(m + n \log n)$.

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.

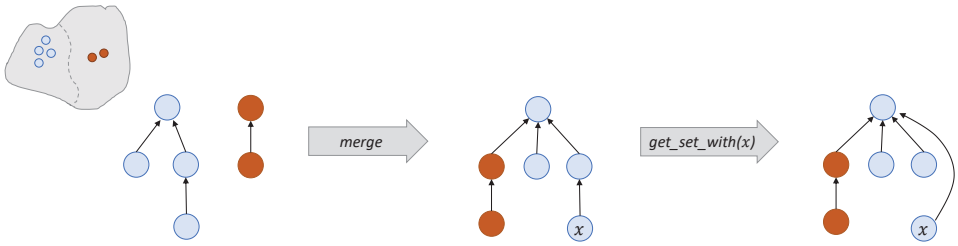
merge() attaches one root to the other, which only requires updating a single pointer.

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.

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 n items is $O(m\alpha_n)$ where α_n is an integer-valued monotonically increasing sequence, related to the Ackerman function, which grows extremely slowly:

$\alpha_n = 0$ for $n = 0, 1, 2$
$\alpha_n = 1$ for $n = 3$
$\alpha_n = 2$ for $n = 4, 5, 6, 7$
$\alpha_n = 3$ for $8 \leq n \leq 2047$
$\alpha_n = 4$ for $2048 \leq n \leq 10^{80}$, more than there are atoms in the observable universe.

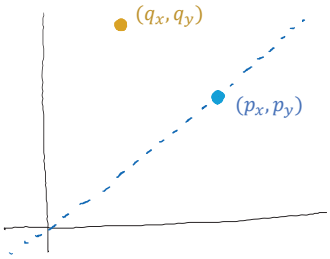
For practical purposes, α_n may be ignored in the O notation, and therefore the amortized cost per operation is $O(1)$.

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.

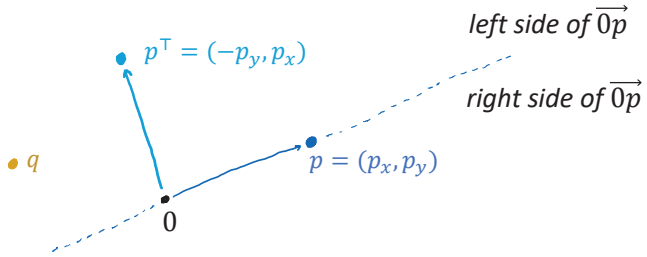
Let's start with a simpler problem. Is the point q above or below the dotted line? The answer doesn't need anything more than basic school maths: if $q_y > \left(p_y/p_x\right)q_x$ then it's above.



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:

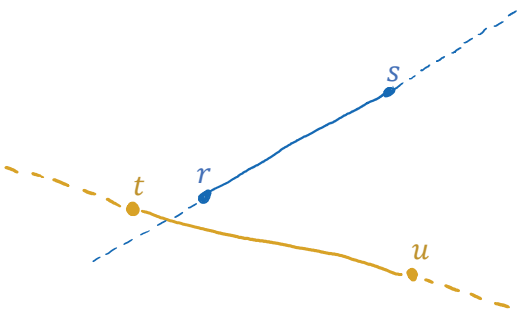
Let $p^\intercal = (-p_y, p_x)$. If we rotate the vector $\overrightarrow{0p}$ by 90° anticlockwise, we get $\overrightarrow{0p^\intercal}$. Now, the sign of $p^\intercal \cdot q$, i.e. of $-p_y q_x + p_x q_y$, tells us which side of the dotted line q is on. The dotted line is called the *extension*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

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



This gives us all the tests we need to decide if two line segments $\overline{r\,s}$ and $\overline{t\,u}$ intersect.

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



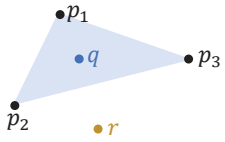
Well-written code should test all the boundary cases, e.g. when $r = s$ or when t or u lie on the extension of $\overrightarrow{r\,s}$. It is however a venial sin to test equality of floating point numbers, because of the vagaries of finite-precision arithmetic, and so the question "How should my segment-intersection 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.

8.2. Jarvis's march

Given a collection of points $p_1, \ldots, p_n$, a *convex combination* is any vector

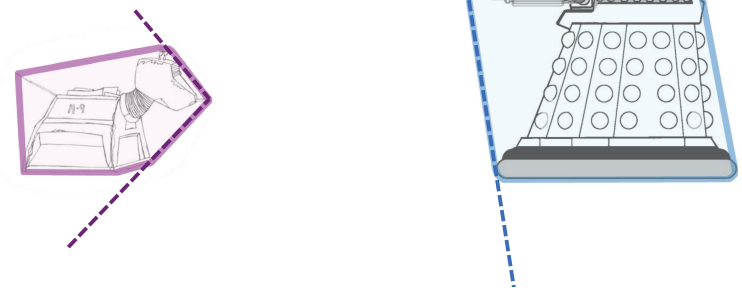
$$q = \alpha_1 p_1 + \cdots + \alpha_n p_n \quad \text{where } \alpha_i \geq 0 \text{ for all } i, \text{ and } \sum_{i=1}^{n} \alpha_i = 1.$$

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



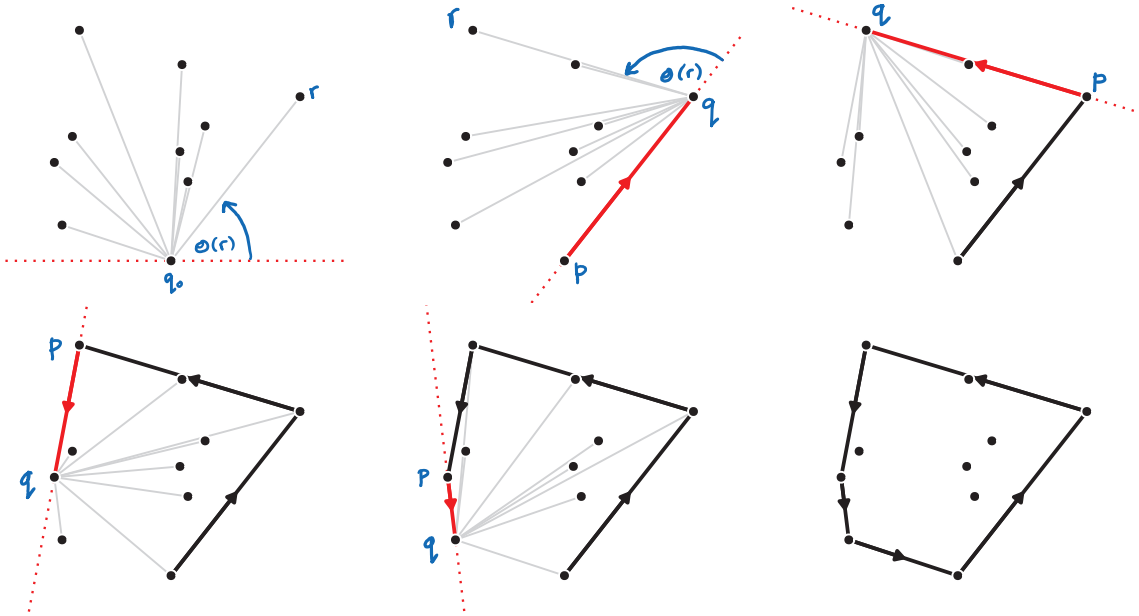
$q = 0.6p_1 + 0.16p_2 + 0.24p_3$: is a convex combination
$r = -0.5p_1 + 0.9p_2 + 0.6p_3$: not a convex combination

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.



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.)



```
let q_0 be the point with lowest y–coordinate
(in case of a tie, pick the one with the largest x–coordinate)

draw a horizontal (left→right) line through q_0
for all other points r ∈ P:
    find the angle θ(r) from the horizontal line to q_0 r, measured ↺
let q_1 be the point with the smallest angle
(in case of a tie, pick the one furthest from q_0)

h = [q_0, q_1]
repeatedly:
    let p and q be the last two points added to h respectively
    for all other points r ∈ P:
        find the angle θ(r) from the extended p q line to q r, measured ↺
    pick the point with the smallest angle, and append it to h
    (in case of a tie, pick the one furthest from q)
    stop when we return to q_0
```

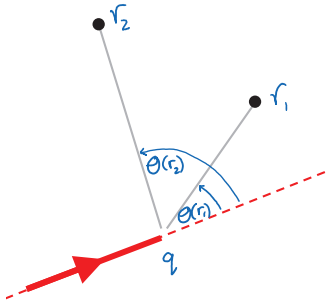
ANALYSIS

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 θ — 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

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

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

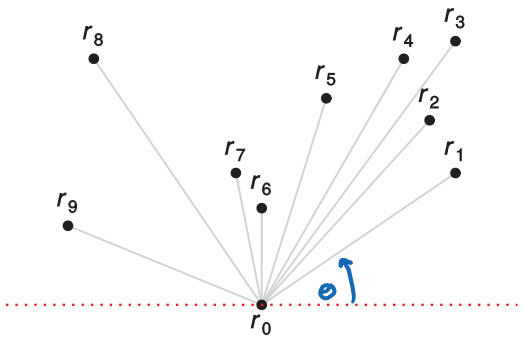


* * *

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.

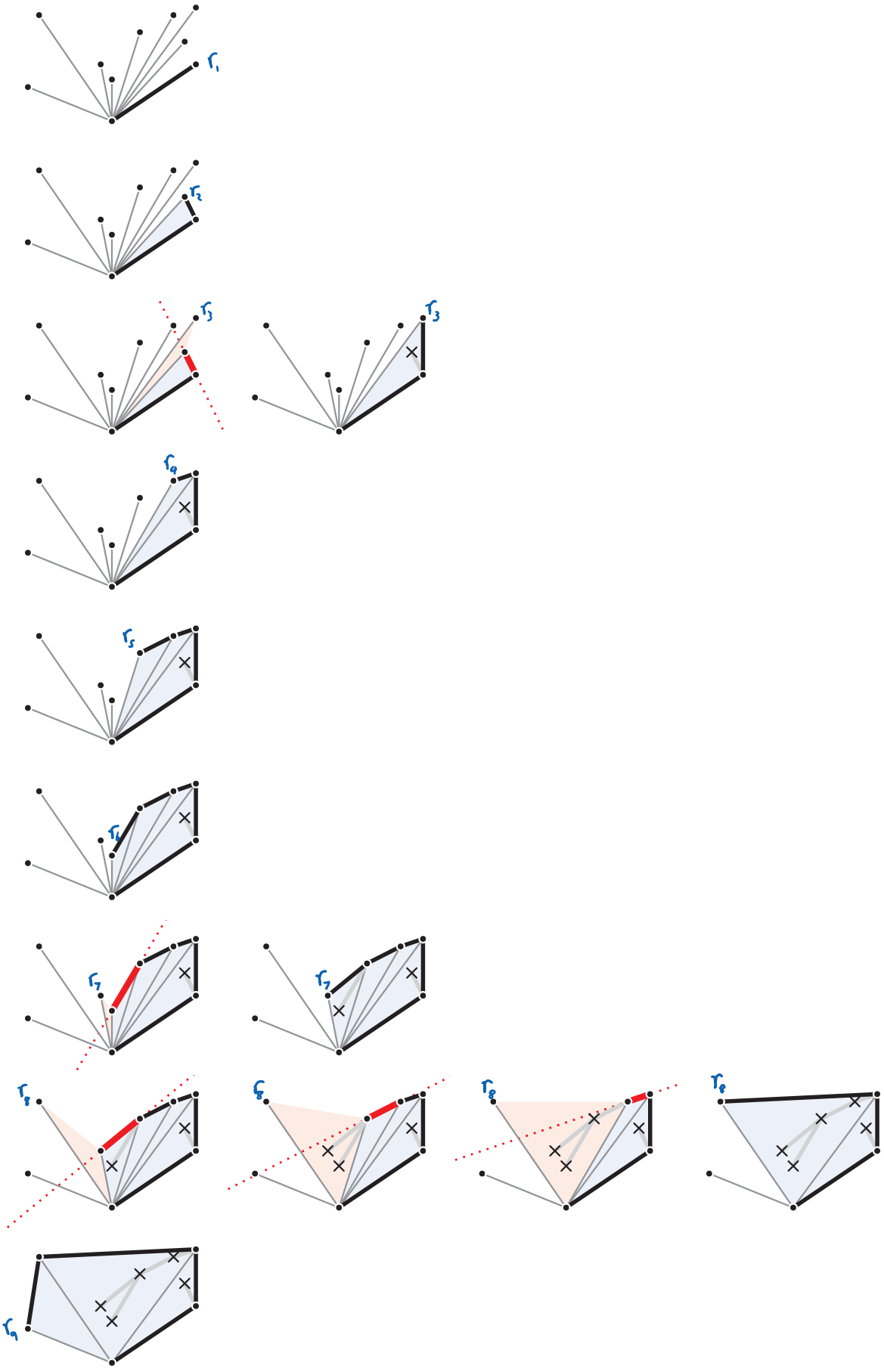
```
let r_0 be the point with lowest y–coordinate
(in case of a tie, pick the one with the largest x–coordinate)

draw a horizontal (left→right) line through r_0
for all other points r:
    find the angle from the horizontal line to r_0 r, measured ↺
let r_1, ..., r_{n-1} be the sorted list of points, lowest angle to highest

h = [r_0, r_1]
for each r_i in the sorted list of points, i ≥ 2:
    if r_i isn't on the left of the extension of the final segment of h:
        # backtrack
        repeatedly delete points from the end of h until r_i is
    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.


r1
r2
r3
r3
r4
r5
r6
r7
r7
r8
r8
r8
r9