Cosc 201
Algorithms and Data Structures
Lecture 9 (24/3/2025)
Heaps

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

The idea of the heap

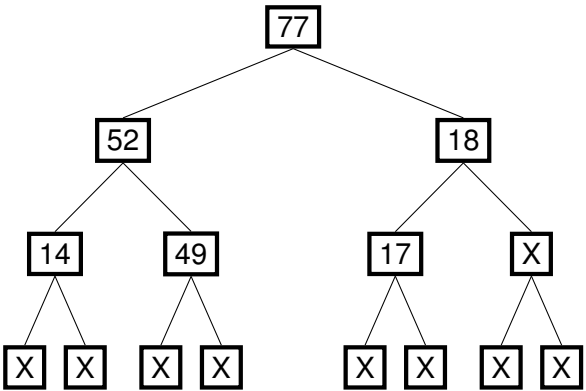
When we are managing a priority queue, what do we want to be able to do?

- Access and remove the element of greatest priority.
- Add new elements.

The difficulty is to ensure that both operations can be performed without having to move $\Theta(n)$ items (the problem with the sorted version) or examine $\Theta(n)$ items (the problem with the unsorted version).

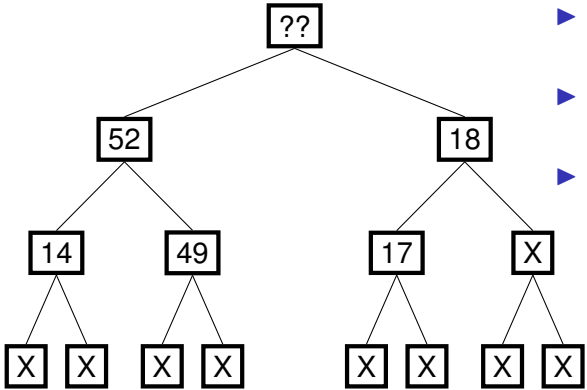
In the third union-find implementation we saw the idea of using a ranked data structure to store at least 2^k things, while keeping a rank bounded by k. Perhaps that can help?

Pictures help



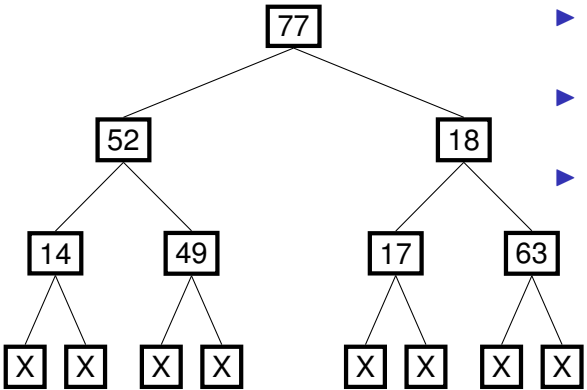
- Think of the squares as *places to put things*
- There's room for $15 = 2^3 + 2^2 + 2^1 + 1$ things
- But the maximum distance from bottom to top is three.
- What's the guiding principle of the organisation of data?
- Will it allow us to do what we want it to do?

Removing the maximum



- Every element should be greater than its occupied children.
- The structure should be filled top to bottom and left to right.
- How do we fix things?

Adding a new element

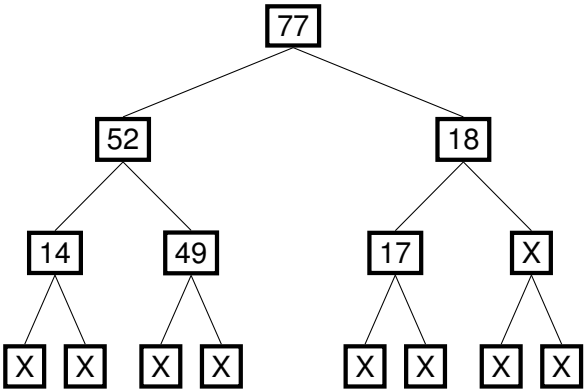


- Every element should be greater than its occupied children.
- The structure should be filled top to bottom and left to right.
- How do we fix things?

How deep is a heap?

- Each layer of the heap is twice as big as the preceding one.
- So layer k (counting from the root being layer 0) can hold up to 2^k elements.
- If we have n elements to store, then certainly we can use k layers where $k = \log n$.
- In fact a heap of total depth k can store up to $2^{k+1} - 1$ elements – a little more than half the elements belong to the final layer.
- So certainly we need $\Theta(\log n)$ layers.
- A consequence is that any algorithm that 'walks along a branch' in whole or in part will have $O(\log n)$ complexity (assuming constant-time work at each spot on the branch).

The idea of the heap



- **A**: Every element should be greater than its occupied children.
- **B**: The structure should be filled top to bottom and left to right.
- To remove the maximum, replace it with the item from the last full spot and let that sink to ensure **A**.
- To add an element put it in the next vacant spot, then let it float up as far as needed to ensure **A**.

Pseudocode for adding an element

Assumptions: We have a valid heap, H, and a way to:

- access the first vacant position,
- set (or find) the value $H.q$ stored in any (occupied) position q
- access the 'parent' of any given position,
- identify when we're at the root,

(all in constant time). We have an item, x to add.

Outcome: Change H by adding x to it, while maintaining the heap conditions.

1: $p \leftarrow$ first vacancy, $H.p \leftarrow x$
2: **while** p is not the root and $H.parent(p) < H.p$ **do**
3: Exchange $H.parent(p)$ and $H.p$
4: $p \leftarrow parent(p)$.
5: **end while**

Pseudocode for removing the maximum

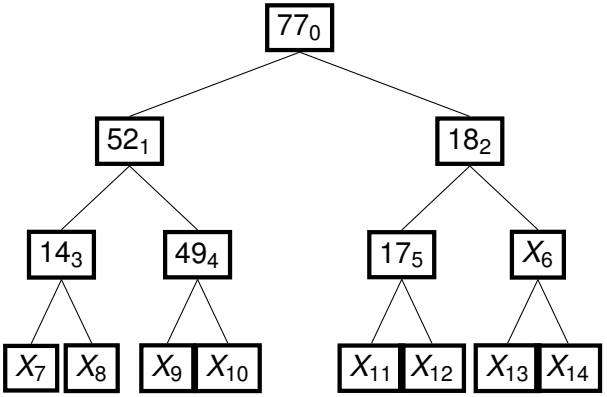
Outcome: Change H by removing its maximum (i.e., root) value while maintaining the heap conditions.

```
1:  v ← H.root
2:  Set H.root to be the value stored in the last occupied position.
3:  p ← root
4:  while p has children do
5:      if the largest value, H.c of a child of p is greater than H.p then
6:          Exchange H.c and H.p, p ← c.
7:      else
8:          Break
9:      end if
10: end while
11: return v
```

Complexity analysis

- According to the pseudocode above, what is the cost of the two heap operations?
- Both do a constant amount of work outside a single loop.
- In addition, we move along the branch from an added element up to the root, fixing any violations we find.
- Similarly in removing an element we move from the root down through some branch until all violations are fixed. Note that a violation can occur only if a node has children.
- So both loops traverse at most one branch of the heap and do $O(\log n)$ work.
- That's the balance we wanted in using a heap for a priority queue!

How do we store a heap?



- We could use an array, or a linked structure.
- A linked structure is a bit simpler, but an array is more efficient.
- We will use an array.
- We can store the root at position 0 and its (potential) children at indices 1 and 2.
- The children of node 1 go in indices 3 and 4, and those of node 2 in indices 5 and 6.
- Can we do all the things we need to do?

Things we need to do

Assumptions: We are using an array `heap` to store a heap H that (currently) contains n elements (assume `heap.length > n`). A "position" is then just an index of `heap`.

Access the first vacant position	`heap[n]`
Set or find the value at position q	`heap[q] ...`
Access the parent of any position	`parent(q) = (q-1)/2`
The children of q	`2*q + 1, 2*q+2` **if** less than `n`
Identify if q is the root	`q == 0`

Writing the code?

- It gets a bit messy (but see code sample for instance).
- Just use `java.util.PriorityQueue`
- But note that the items you add need to have an associated *comparator*
- The Java implementation is of a “min-queue” where the *least* element is returned first.
- In the heap context that would mean we’d want the *least* element at the root and the children of any element to be greater than (or equal to) the element.
- So why did I use max-heap? Because . . .

Heap sort

- A max-heap allows us to present the `HeapSort` algorithm, a $\Theta(n \log n)$ sorting algorithm that operates in place.
- Basic idea:
  - Start with an arbitrary array
  - Using itself as a heap, add the elements one at a time until all have been added
  - Then remove them one at a time - the largest element gets removed first and the place where it needs to be put gets freed from the heap.
- It's just magic.
- Watch the 'movie' version in the lecture video or google "heap sort animation"

To heap or to merge?

HeapSort (HS) An in-place sorting algorithm with $\Theta(n \log n)$ time complexity.

MergeSort (MS) A sorting algorithm that can require extra storage of size $n/2$ and has $O(n \log n)$ complexity.

- MS is generally preferred to HS. Why? There's extra storage cost and no apparent gain.
- It comes down to engineering or the fine print:
  - HS can't take advantage of "partially sorted" data, while MS does so automatically (hence the Θ for HS as opposed to the O for MS)
  - Many of MS's memory accesses are in sequential locations controlled by an incrementing counter – this is ideal for using fast memory.
  - On the other hand HS jumps around through the array.
  - HS does lots of exchanges, MS does overwrites. The latter can frequently allow for optimisations (e.g., block copies).
- Bottom line is that, MS is just going to run more quickly so unless memory use is a key bottleneck there's no reason to use HS.