Priority Queues and Heaps

Goodrich chapter 7

Priority Queue Implementations

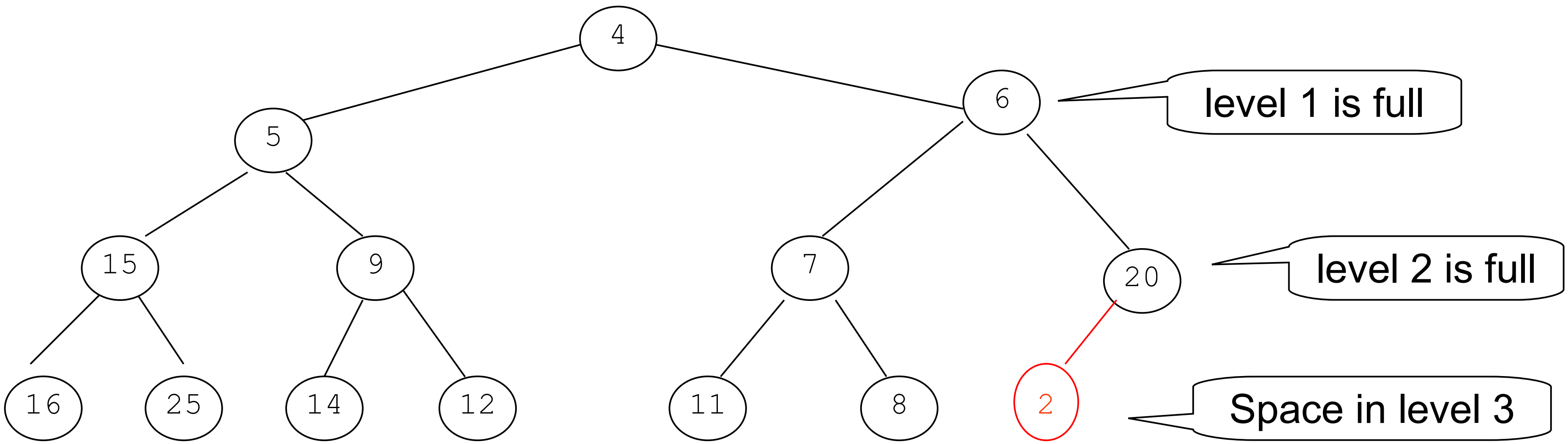
- Any container of items which supports ordering, can be used to implement a priority queue
  - It could be an array or a doubly-linked list
- If the array or list *is not sorted*
  - An item can always be added to the priority queue in constant time
  - To remove highest priority will involve searching the whole collection to find it – so O(n), i.e. linear time
- If we maintain the array or list *in sorted order*
  - It will take longer to insert an item
  - But maximum can be removed in constant time

A Better Priority Queue Implementation - The Heap

- One very good data structure for implementing a Priority Queue is the Heap, which is another type of binary tree data structure.
  - It is not a binary search tree, but it is sorted in such a way that we can quickly remove the elements on the heap in sorted order, so it is a priority queue.
- The definition of a heap is recursive:
  - A binary tree in which the root node holds the smallest (or largest) item, and every sub-tree is also a heap.
- Any binary tree can be organised in this way: it is referred to as sifting the heap.
  - Once a binary tree is sifted, items can quickly be removed in order
  - Sifting the heap can be done in O(nlogn) time, so it is a more efficient sorting algorithm than bubble sort or selection sort which were both $O(n^2)$.
- Whenever the smallest item is removed (i.e. the root node), what is left is re-organised so that it is still a heap
- Whenever an item is added to the heap, re-organisation ensures that it is still a heap.

Algorithm for inserting into a heap

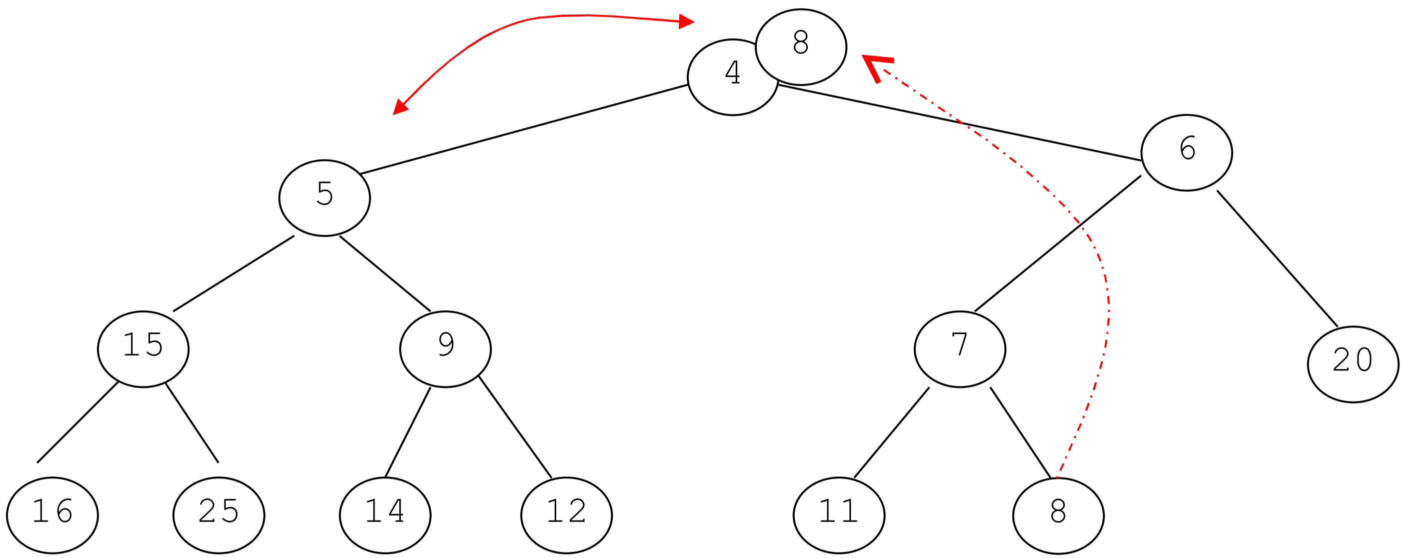
- The heap is maintained as a complete binary tree. That means that every level except the last one is full, and the bottom level is filled from left to right



- Unlike with a binary search tree, when we insert into a heap, we always maintain a complete binary tree.
- The node to be inserted is placed in the next vacant position, and is then bubbled up by repeatedly swapping with its parent if it is smaller than the parent.
- Example, to insert 2 in this heap, place it as left child of 20. swap with 20, then swap with 6, then swap with 4

Algorithm to remove the min element from the heap

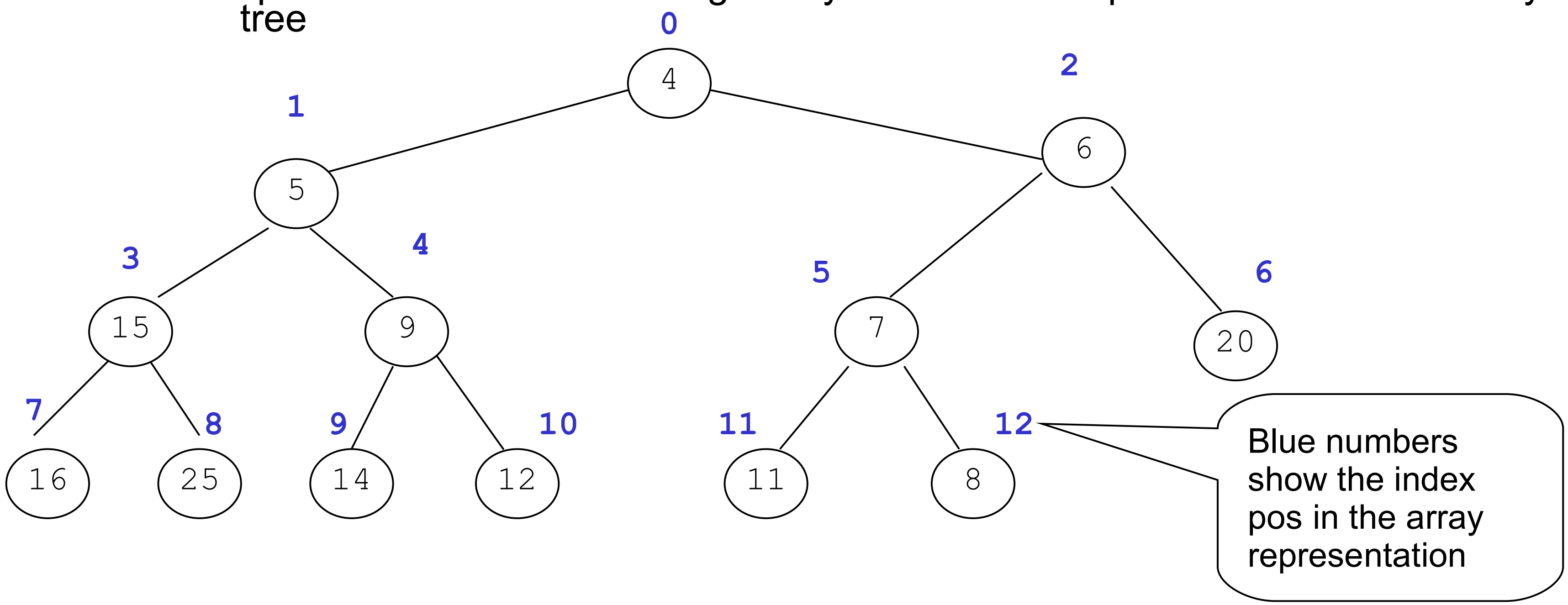
- This node will always be at the top of the heap
- We cant just take it away – we wouldn’t have a tree anymore
- So instead, we replace it with the last item in the tree, and then sift that item down to maintain the heap property.
- We will have to see which is the min of its right and left children, and swap with that, then work down the tree until it comes to the correct place



Representing a heap

A heap is usually represented as an array when implementing the heap algorithms.

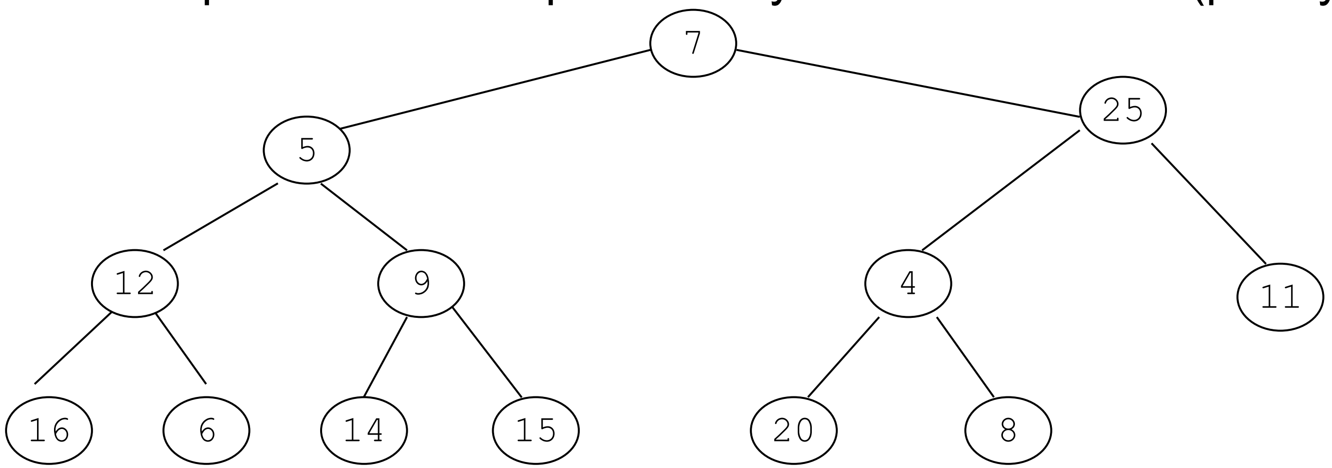
It is possible to do this unambiguously because a heap is a COMPLETE binary tree



- Working across each level in turn, this heap can be represented by the array: {4, 5, 6, 15, 9, 7, 20, 16, 25, 14, 12, 11,8}
- Index positions into the array can be used to access the children and parent of any node
  - The left child of 'node' at index pos i is at 2i + 1 and the right child is at 2i + 2
  - The parent of node at i is the node at ((i-1) / 2) (using integer arithmetic)

Using an array representation to sift a heap

- Start with an unsorted array:
- For example, the array { 7, 5, 25, 12, 9, 4, 11, 16, 6, 14, 15, 20,8 } which would represent the complete binary tree shown below (plainly not a heap)



- Sort the array in-place, by using the front of the array to store the portion sorted so far (shown here in blue) *This is a good way to do it as there is no memory overhead to maintain the sorted part of the heap.*
- {7, 5, 25, 12, 9, 4, 11, 16, 6, 14, 15, 20,8} - one item is a heap
- {5, 7, 25, 12, 9, 4, 11, 16, 6, 14, 15, 20,8} - add the 5 and bubble it up the heap (index pos 1, compared with parent at index pos (1-1) / 2 = 0)
- {5, 7, 25, 12, 9, 4, 11, 16, 6, 14, 15, 20,8} - add the 25 – no bubbling needed
- {5, 7, 25, 12, 9, 4, 11, 16, 6, 14, 15, 20,8} – add the 12 – no bubbling needed (12 is at index pos 3, so compare it with parent at index pos (3-1)/2 = 1, i.e. the 7
- etc

The heap-sort algorithm

- The heap data structure is also the basis for an efficient sort algorithm.
- Heap-Sort consists of taking an array of items, and inserting them into a heap so that they can be extracted in sorted order.
- Heap-Sort is an O (nlogn) algorithm
  - Both the insertion and removal of items from the heap takes no more than d comparisons, where d is the depth of the heap.
  - For a heap holding n items, the depth of the tree is considerably less than n, in fact it is proportional to log(n), so insertion and deletion are both done in log(n) time.
  - To sort a list of n items, we need n insertions into and n extractions from the heap, so total time to sort n items using heap sort is proportional to n × log(n) (hence O(nlog(n))
- The other sorting algorithms we have seen so far, bubble sort and selection sort, have taken $O(n^2)$ comparisons.
- We will study some other algorithms later – Merge Sort, which is also O(nlogn) and Quick Sort, which is O(nlogn) most of the time, but in the *worst case scenario*, it is still $O(n^2)$
  - And consider some pros and cons of each algorithm.

Heaps and the STL

- There are algorithms based on a heap in the STL, but no heap template class.
- The heap in the STL is sorted so that the item with the highest priority is at the top of the heap.
- There are 4 algorithms to handle a heap:
  - `make_heap()` converts a range of elements into a heap
  - `push_heap()` remakes the heap *after* one element has been added to the end
  - `pop_heap()` moves the highest priority item in the heap to the end, and re-makes the heap with the remaining items
  - `sort_heap()` converts a heap into a sorted collection in ***ascending*** order of priority (after this it is no longer a heap)
- Each of the algorithms has arguments as for the sort algorithm
  - A random-access container and two arguments which are iterators to the beginning and end of a range in the container
  - Optionally, a fourth argument which is the binary predicate used for the sort criteria (ie a ‘less than’ type function)
- The heap algorithms are used internally by the `priority_queue`

Heap analysis

- The heap data structure is very efficient for retrieving the highest priority item in a collection
  - Top item can be retrieved in constant time
- Inserting an item requires bubbling up the tree, so comparing with parent node at each level
  - The height of a complete tree is O(log(n))
  - Items can be inserted in O(logn) time

Deleting the top item requires sifting a replacement item down to its correct place – possible all the way to the bottom of the tree
  - The time taken to do this is again proportional to the height of the tree
  - Top Item can be deleted in O(logn) time
- But it would be next to useless for searching for a particular item when this wasn't the highest
  - No assumptions can be made about the position of any item
  - A recursive search of the tree would be required
  - The best we could do is use a search method which visits each node of the tree exactly once such as a depth first search
  - Search can be done in O(n) time

Use a priority queue for what its good at, not for general searching!