Heaps

▼ *Priority Queues*

Priority Queues

A priority queue (heap) is an abstract data type that maintains a collection of items which has an associated *priority* value and can get (and remove) the item with highest priority efficiently. New items with arbitrary priorities can be added at any time.

We will use the names heap and priority queue interchangeably, although often *heap* is reserved for particular implementations of the priority queue ADT.

- ***Just as a reminder ADT stands for Abstract Data Type***

Heap ADT

We will use the names heap and priority queue interchangeably, although often *heap* is reserved for particular implementations of the priority queue ADT.

A heap is an abstract data structure consisting of a set of items, each with a `key`, with the following basic operations:

`EmptyHeap` Return a new, empty heap.

`heap.insert(value)` Insert a new value with predefined key into the heap.

`heap.get_max()` Return an item of maximum key in the heap. This operation does not change the heap.

`heap.delete_max()` Delete an item of maximum key from the heap and return its value.

Priority Queue

There are a number of obvious implementation strategies we could use:

- Unsorted Array: Inserting an item is $O(1)$, `get_max` is $O(n)$
- Sorted Array: Insert is $O(n)$, `get_max` is $O(1)$
- ...

We can do better than that.

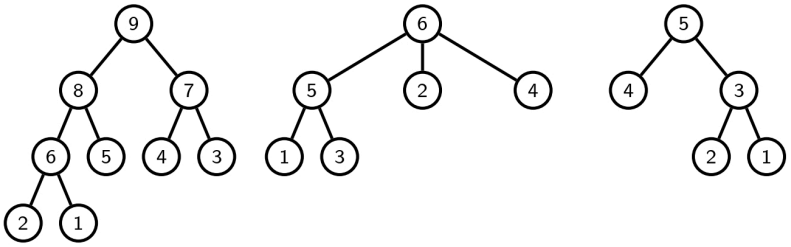
▼ *Binary Heaps*

Heap trees

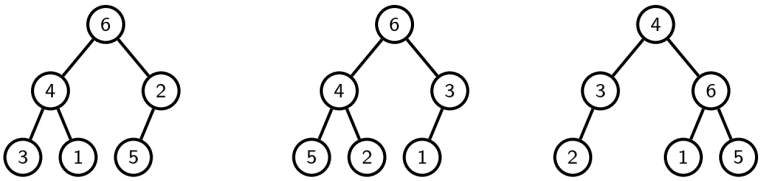
A tree is a heap tree if it satisfies the following condition:

The key of each node is bigger than, or equal to, the key of each of is children.

Examples:



Non-examples:

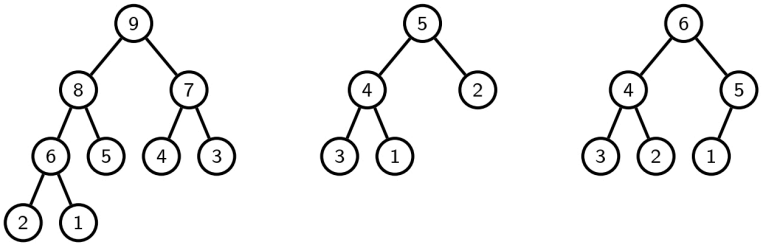


4

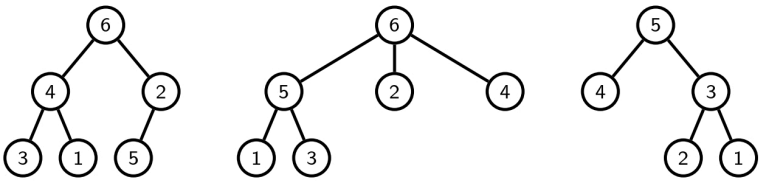
Binary Heap

Definition. A binary heap tree is a complete binary heap tree.

Examples:



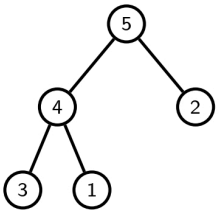
Non-examples:



5

Binary heap in an array

Binary heaps are easy to store in an array. We can simply allocate large-enough array, and keep track of the size of the heap.



`heap` $= [5, 4, 2, 3, 1, \times, \times]$

`size` $= 5$

```
1  class BinaryHeap {
2    int[] heap;
3    int size;
4
5    public EmptyHeap() {
6      heap = new int[MAXSIZE];
7      size = 0;
8    }
9
10   public bool isEmpty() {
11     return (size == 0);
12   }
13
14   int parent(i) return (i - 1)/2;
15   int left(i) return 2*i + 1;
16   int right(i) return 2*i + 2;
17 }
```

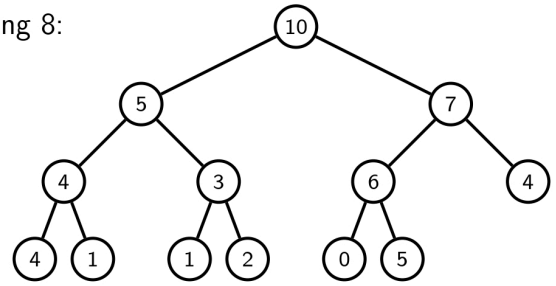
6

▼ *Inserting to anywhere and Deleting from end*

Insertion

Idea: Insert the value at the end of the last level and then keep bubbling it up as long as it is larger than its parent.

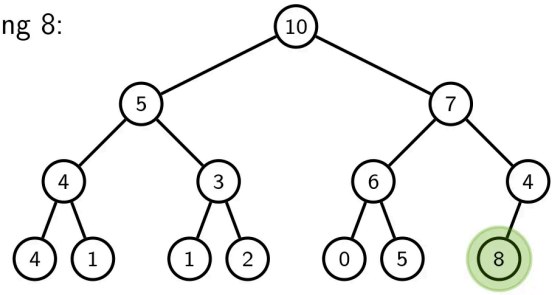
Inserting 8:



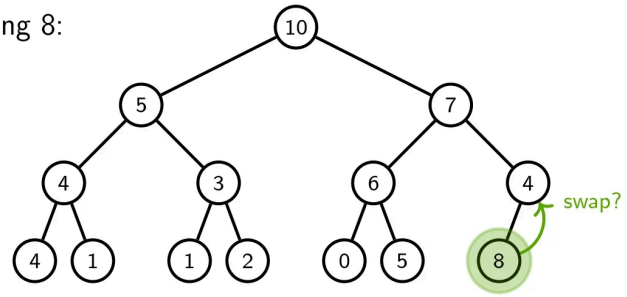
As we bubble up, when we swap the value of a node i with that of its parent, we don't have to compare i with its sibling, because if the value of i is greater than that of its parent, then it must be greater than that of its sibling because of the Binary Heap Tree property.

7

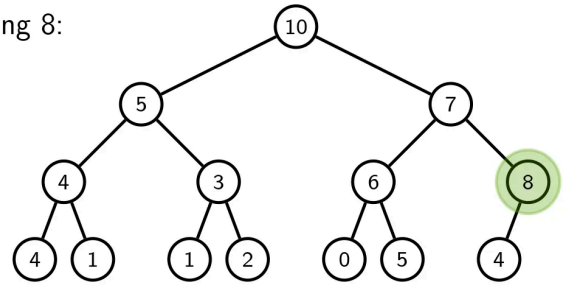
Inserting 8:



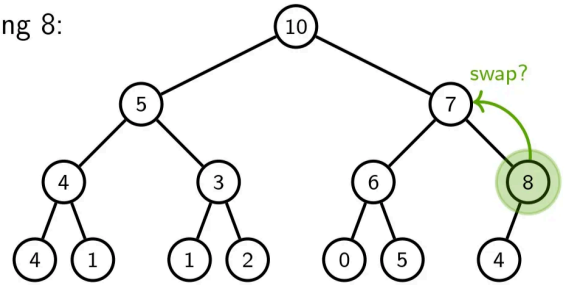
Inserting 8:



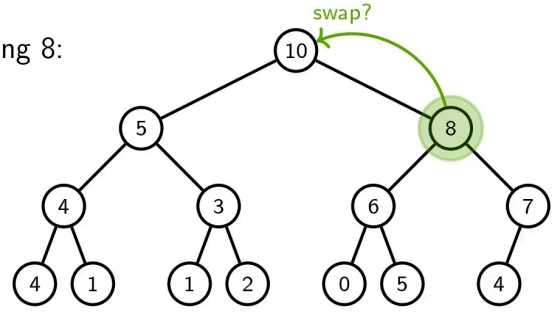
Inserting 8:

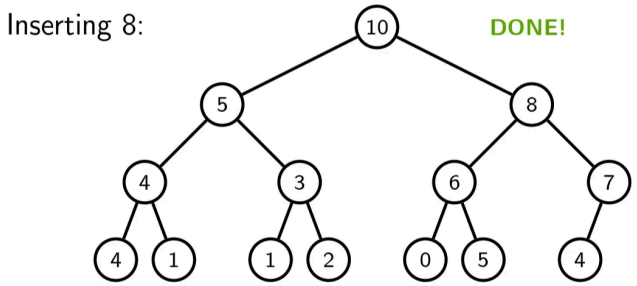


Inserting 8:



Inserting 8:





Insert and sift_up code

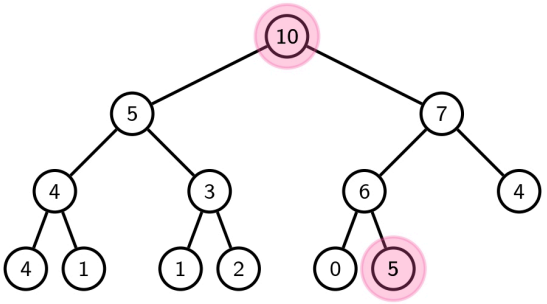
```
1  public void insert(int key) {
2    if (size == MAXSIZE) throw HeapFullException;
3
4    heap[size] = key;  // insert the new value as the last element
5    bubble_up(size);     // and bubble it up
6    size++;            // increase the size
7  }
8
9  void bubble_up(int index) {
10   int i = index;
11   while (i > 0 && heap[i] > heap[parent(i)]) {
12     // swap heap[i] and heap[parent(i)]
13     int tmp = heap[i]; heap[i] = heap[parent(i)]; heap[parent(i)] = tmp;
14     // and continue as long as necessary
15     i = parent(i);
16   }
17 }
```

8

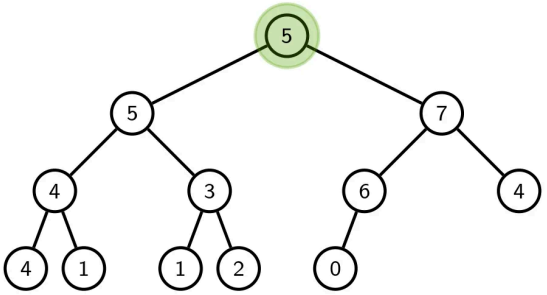
Delete

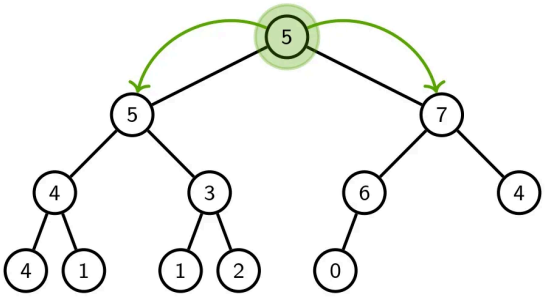
Idea!

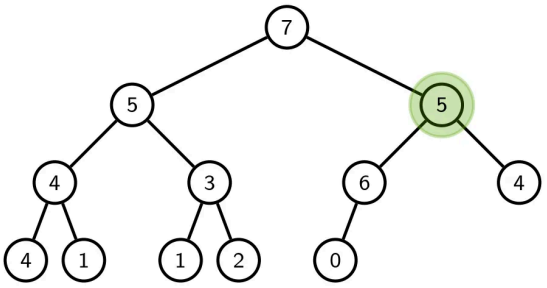
1. Remove the last node and use it to replace the root
2. Then bubble down: keep swapping it with the higher priority child as long as any of its children has a higher priority.

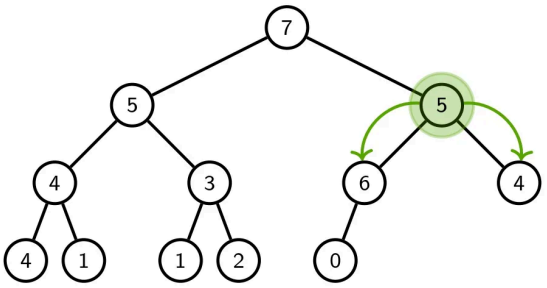


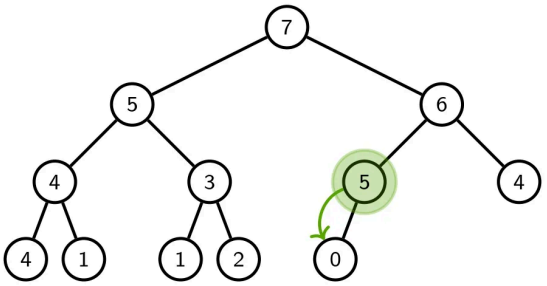
9

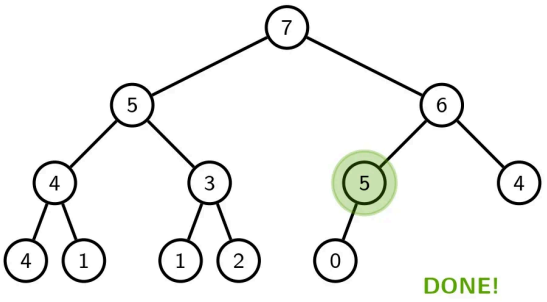

5
5
7
4
3
6
4
4
1
1
2
0


5
5
7
4
3
6
4
4
1
1
2
0


7
5
5
4
3
6
4
4
1
1
2
0


7
5
5
4
3
6
4
4
1
1
2
0


7
5
6
4
3
5
4
4
1
1
2
0



Delete (code)

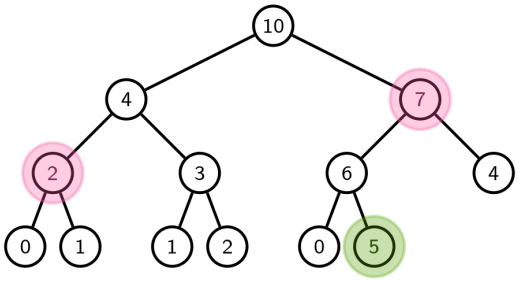
```
1  void delete_max() {
2    if (isEmpty()) throw EmptyHeapException;
3
4    heap[0] = heap[size-1];
5    size--;
6    bubble_down(0);
7  }
```

▼ *Deleting arbitrarily*

Exercise: Delete an arbitrary node

Deleting an arbitrary node means the node might be anywhere in the tree.

1. Remove the last node and use it to replace the node to be deleted
2. This node may be smaller than its children or larger than its parent, so bubble up or bubble down as necessary



13

▼ *Bubbling Downwards*

Bubble downwards

The `bubble_down` method needs to deal with 5 cases, depending on the current node that is being bubbled down:

1. it is a leaf node: nothing to do, `bubble_down` is complete;
2. otherwise, if it is greater than one of its children, swap it with the largest of its children. We still need to consider three cases:
   - 2.1 it has a left child, but no right child: if left child is greater than it, swap left child with current and `bubble_down` is complete (no right child and this is a complete tree means the left child cannot have any children);
   - 2.2 it has two children, the left child is greater of the two and is greater than the current node: swap left child and current and continue recursively bubbling down the left child;
   - 2.3 as the previous case but the right child is the greater: continue as above but on the right child.

11

Bubble downwards (code)

```
1  void bubble_down(int index) {
2    int i = index;
3    while (left(i) < size) {      // while the node has children
4      if (right(i) >= size) {     // only left child
5        if ( heap[i] < heap[left(i)] )
6          swap heap[i] and heap[left(i)]
7          i = left(i);            // will terminate on the next loop
8      } else {                    // two children
9        // pick the bigger_child
10       int next_i = (heap[left(i)] > heap[right(i)]) ? left(i) : right(i);
11       if (heap[i] < heap[next_i]) {
12           swap heap[i] and heap[next_i]
13           i = next_i;           // continue
14       } else return;            // done!
15     }
16   }
17 }
```

12

▼ *Heapsort algorithms*

Heapsort

What can we use heaps for?

Recall a sorting strategy: Pick the *maximal element* and put it at the end of the array. The naive implementation of this algorithm, known as Selection sort, runs in $O(n^2)$ time.

Idea! But picking the maximal element from a heap is $O(\log n)$ instead of $O(n)$ in case of an array!

Algorithm (Heapsort) Input: an array `a`.

1. build a heap out of `a`;
2. keep a pointer `i` into `a`, and set it to start with the last element;
3. while `heap` is not empty:
   3.1 take the top out of the heap, and put it at the end of `a`;
   3.2 decrement `i`;

14

It's $O\log(n)$ because you cause use binary search to find the max element. Simply assign the current value to your first value and every time one value is bigger than your current value eliminate that half of the list

Building a Binary Heap = Heapify

There are two ways of turning an array to a binary heap:

```
1 void heapify_up(int[] array) {
2   heap = array;
3   size = array.length;
4   for (int i=0; i<array.length; i++)
5       bubble_up(i);
6 }
```

```
1 void heapify_down(int[] array) {
2   heap = array;
3   size = array.length;
4   for (int i=array.length-1; i>=0; i--)
5       bubble_down(i);
6 }
```

Note. This is the same as n insertions.

Which one is better and why?

15

Inserting a set of n items into an empty Binary Heap Tree is n inserts of complexity $O(\log n)$ giving a total complexity of $O(n \log n)$.

There is a more efficient way: If we have the items in an array in random order (starting at index position 1), then we already have them in Complete Binary Tree form, but not in Binary Heap Tree form. At this point, all the leaf nodes satisfy the heap tree properties, but the internal nodes do not.

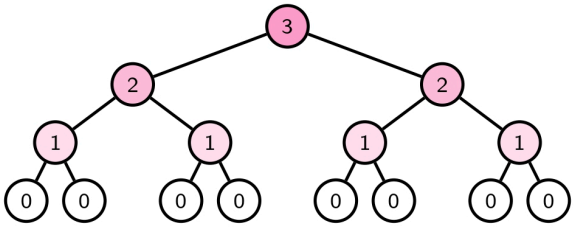
We can therefore iterate over the internal nodes, starting with the last internal node and working up to the first, calling `bubble_down` on each in turn. Each time we do, we ensure that the subtree based on that node becomes a valid Binary Heap Tree, so that in the end, the whole tree is a valid Binary Heap Tree.

Again because searching for the right spot to insert is $O \log n$ is the average case scenario

The heap tree property is all the children's value must be smaller than the parent

Complexity of heapify_down

Assume that we have a tree of size $n = 2^{h+1} - 1$, i.e., a full binary tree of height h. Note that $h \sim \log n$. Then the number of swaps needed for each node are:



The total number of swaps is:

$$2^h \cdot 0 + 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + \cdots + 1 \cdot h$$

$$= \sum_{i=0}^{h} 2^{h-i} \cdot i \leq \sum_{i=0}^{h} \frac{n}{2^{i+1}} \cdot i \leq n \sum_{i=0}^{\infty} \frac{i}{2^{i+1}} \leq Cn = O(n)$$

The worst-case time complexity is $O(n)$!

16

Since $n \approx 2^h$

$$2^{h-i} \approx \frac{n}{2^{i+1}} = n * 2^{-(i+1)} \approx 2^h * 2^{-(i+1)} = 2^h * 2^{-i-1} = 2^{h-i-1} \approx 2^{h-i}$$

Therefore

$$2^{h-i} * i \approx \frac{n}{2^{i+1}} * i$$

Heapsort conclusions

Algorithm (Heapsort)

Input: an array `a`.

1. build a heap out of `a`;
2. keep a pointer `i` into `a`, and set it to start with the last element;
3. while `heap` is not empty:
   - 3.1 take the top out of the heap, and put it at the end of `a`;
   - 3.2 decrement `i`;

```
1  void heapsort(int[] a) {
2    // heapify
3    for (int i=a.length-1; i >= 0; i--)
4        bubble_down(i);
5
6    // dismantle the heap
7    for (int i=a.length-1; i > 0; i--) {
8      int tmp = a[i];
9      a[i] = a[0];
10     a[0] = tmp;
11     bubble_down(0, size=i);
12   }
13 }
```

Time complexity:	Space complexity:	Stability:
$O(n) + n \cdot O(\log n) = O(n \log n)$	$O(1)$ (in place)	Not stable!

17

Not stable because if two elements are the same value they won't necessarily maintain the same positions in the list

Fibonacci Heaps

Fibonacci Heaps are a more complex implementation of priority queues as a heap trees. They are not necessarily binary, allow empty nodes, and make use of lazy modifications to keep themselves organised.

They are much more efficient than binary heaps: They achieve complexity of $O(1)$ for insert. They also allow efficient implementation of merging two heaps in $O(1)$, and updating the priority of a node in amortised complexity $O(1)$.

Operation	Binary Heaps	Fibonacci Heaps
Insert	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)^\star$
Heapify	$O(n)$	$O(n)$
Update	$O(\log n)$	$O(1)^\star$
Merge	$O(n)$	$O(1)$

* = amortised complexity.

18

Priority Queue applications

In general, priority queues are useful whenever we repeatedly need the maximum (or *minimum*)[1] of a changing collection.

For example, if do a search on the web, the underlying search algorithm will find potential matches, each with a score that estimates how good a match it is. It may put those matches into a priority queue, and then extract the 10 best matches from the queue. In the background, further matches might be searched for and added to the queue. When the go to the next page of results, then the next 10 best matches are extracted and displayed.

Similar examples can be found in finding the best next move in a computer game, online flight search websites, hotel booking systems, comparative pricing websites

[1] Heaps (as in our case) whose the top element is the maximum are usually called **max-heaps**. The other kind of heaps, **min-heaps** satisfy that the children of each node have *bigger* value.