

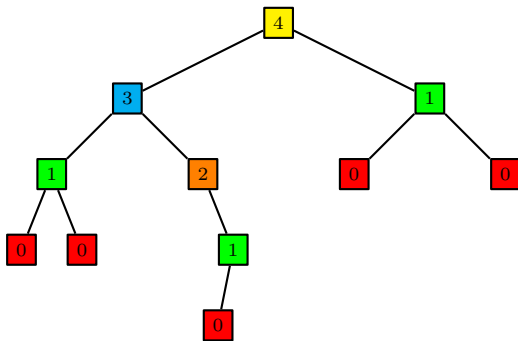
CS624 - Analysis of Algorithms

Heaps

September 17, 2025

Heaps and Heapsort – Introduction to Heaps

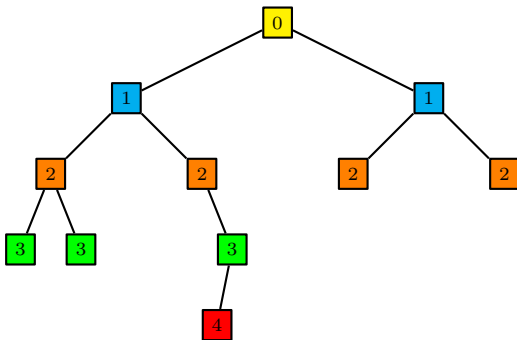
Define the height of a node in a tree as the number of edges on the longest path from that node down to a leaf



The height of the tree, H , is the height of its root.

Heaps and Heapsort – Introduction to Heaps

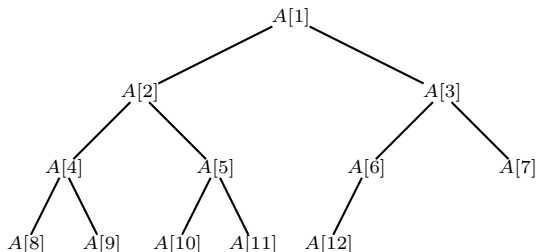
The *level* of the root is 0, the children of the root are at level 1. In general, the children of a node of level k are at level $k + 1$.



Heaps and Pre-Heaps

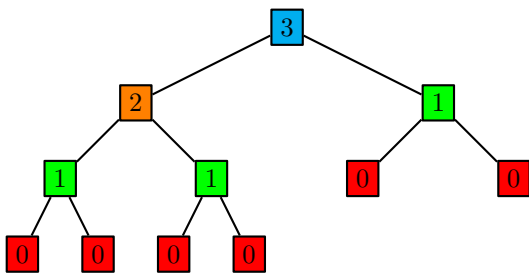
- In a binary tree, there are at most 2^k nodes at level k .
- If the highest level is completely filled in, that level contains 2^H nodes, and the tree contains $1 + 2 + 4 + \dots + 2^H = 2^{H+1} - 1$ nodes.
- A *heap* is a special kind of a binary tree (do not confuse with the CS term related to memory allocation!).
- Let us define a *pre-heap* as follows:
 - All leaves are on at most two adjacent levels.
 - Except maybe the lowest level, all the levels are completely filled. The leaves on the lowest level are filled in, without gaps, from the left.

Pre-Heap – Example



- Notice that it can be represented as a simple array.
- The children of the node holding $A[n]$ are the nodes holding $A[2n]$ and $A[2n + 1]$, and the parent of the the node holding $A[n]$ is $A[\lfloor n/2 \rfloor]$.

Pre-Heap – Another Example



- The nodes are tagged by their height.
- All the levels less than 3 are completely filled in, and there are a total of $2^3 - 1$ nodes at those levels.

Pre-Heap Properties

- If we have a pre-heap with n nodes, denote its height by H .
- As seen above, we must have $2^H \leq n \leq 2^{H+1} - 1 < 2^{H+1}$.
Equivalently, $H = \lfloor \log_2 n \rfloor$

Lemma

In a pre-heap with n elements, there are $\lceil \frac{n}{2} \rceil$ leaves.

Proof.

- Some leaves are at level H , and some are at level $H - 1$.
- Since the number of nodes at level $H - 1$ or less is $2^H - 1$, the number of leaves at level H is $n - (2^H - 1)$.
- The parent of node n is node $\lfloor \frac{n}{2} \rfloor$, and that node is the last node of height 1.



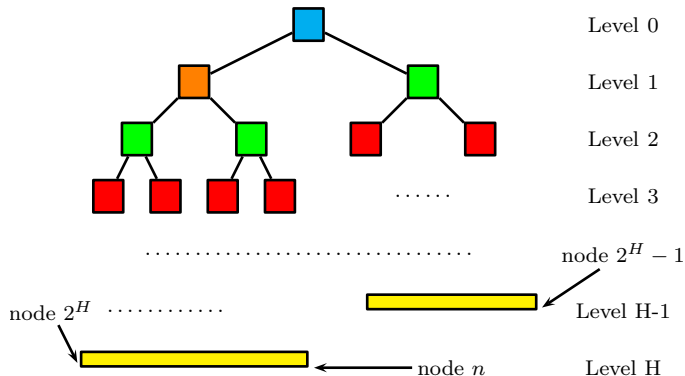
Proof (Cont.)

- So all the rest of the nodes at level $H - 1$ are of height 0, i.e., are leaf nodes.
- Therefore the number of leaves at level $H - 1$ is $(2^H - 1) - \lfloor \frac{n}{2} \rfloor$.
- Hence the total number of leaves is

$$n - (2^H - 1) + (2^H - 1) - \lfloor \frac{n}{2} \rfloor = n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$$



Pre-Heap Properties



Pre-Heap Properties

Corollary

In a pre-heap with height H , there are at most 2^H leaves.

Proof.

If n is the number of elements in the pre-heap, we know that $2^H \leq n \leq 2^{H+1} - 1 < 2^{H+1}$. Then by the Lemma, the number of leaves is

$$\left\lceil \frac{n}{2} \right\rceil \leq \frac{2^{H+1}}{2} = 2^H$$



Pre-Heap Properties

Theorem

In a pre-heap with n elements, there are at most $\frac{n}{2^h}$ nodes at height h .

Proof.

- We have just seen that there are at most 2^H leaves in such a tree, and the leaves are just the nodes at height 0.
- If we take away the leaves, we have a smaller pre-heap with at most 2^{H-1} leaves, and these leaves are exactly the nodes at height 1 in the original tree.
- Continuing, we see that there are at most 2^{H-h} nodes at height h in the original tree, therefore $2^{H-h} = \frac{2^H}{2^h} \leq \frac{n}{2^h}$

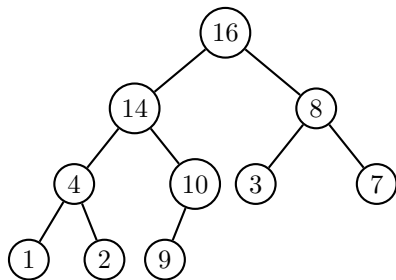
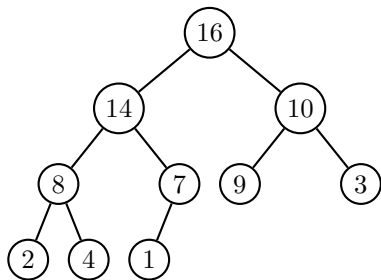


Definition

- A *heap* is a binary tree with a key in each node.
- The keys must be comparable.
- Additionally, the heap must have the following properties:
 - All leaves are on at most two adjacent levels.
 - With the possible exception of the lowest level, all the levels are completely filled. The leaves on the lowest level are filled in, without gaps, from the left.
 - The key at each node is greater than or equal to the key in any descendant of that node.

- Note that another way of phrasing the third condition would be:
 - The key in the root is greater than or equal to that of its children, and its left and right subtrees are again heaps.
- Thus every heap is a pre-heap.
- Even though the shape of a heap containing n elements is uniquely determined (since it is a pre-heap), the arrangement of those n elements is not.

Example – Two Heaps With the Same Set of Keys



The Heapify Procedure

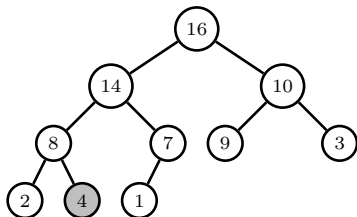
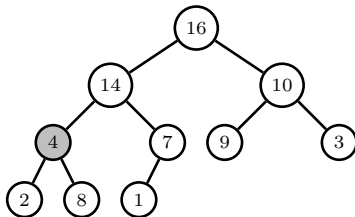
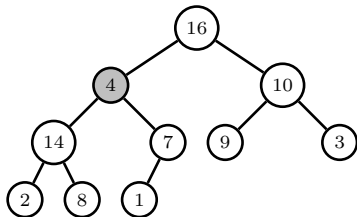
- The fundamental procedure to build a heap.
- We have a binary tree in the shape of a heap (but perhaps not actually a heap).
- We represent the tree as an array $A[1..n]$, where n is the size of the heap.
- We look at node i (holding the value $A[i]$). We assume that:
 - The tree rooted at $l = \text{Left}(i)$ is a heap.
 - The tree rooted at $r = \text{Right}(i)$ is a heap.
- However, we do not assume that the tree rooted at i is a heap.
- Heapify works by letting the value $A[i]$ “float down” to its proper position in the heap.

The Heapify Procedure

Algorithm 1 Heapify(A, i)

```
1:  $l \leftarrow \text{Left}(i)$ 
2:  $r \leftarrow \text{Right}(i)$ 
3: if  $l \leq \text{Heapsize}[A]$  and  $A[l] > A[i]$  then
4:    $\text{largest} \leftarrow l$ 
5: else
6:    $\text{largest} \leftarrow i$ 
7: end if
8: if  $r \leq \text{heapsize}[A]$  and  $A[r] > A[\text{largest}]$  then
9:    $\text{largest} \leftarrow r$ 
10: end if
11: if  $\text{largest} \neq i$  then
12:    $\text{exchange} A[i] \leftrightarrow A[\text{largest}]$ 
13:   Heapify( $A, \text{largest}$ )
14: end if
```

Heapify – Example



Running Time of Heapify

The time needed to run Heapify on a subtree of size n rooted at a given node i is

- time $\Theta(1)$ to fix up the relationships among the elements $A[i]$, $A[\text{Left}(i)]$, and $A[\text{Right}(i)]$
- time to run Heapify on a subtree rooted at one of the children of node i .
- That subtree has size at most $2n/3$ – the worst case occurs when the last row of the tree is exactly half full.

So the running time $T(n)$ can be characterized by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

This falls into case 2 of the “master theorem”, and so we must have $T(n) = O(\log n)$

Building a Heap

The heap is built from the bottom up, starting at the first non-leaf node.

Algorithm 2 BuildHeap(A)

```
1:  $heapsize[A] \leftarrow length[A]$ 
2: for  $i \leftarrow \lfloor length[A]/2 \rfloor$  to 1 do
3:    $Heapify(A, i)$ 
4: end for
```

To prove that this is correct We use the following loop invariant:

Lemma

At the start of each iteration of the for loop, each node $i + 1, i + 2, \dots, n$ is the root of a heap.

Proof of Correctness

Proof.

- On the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial heap.
- Inductive step – going from $i + 1$ to i , we assume that each element $i + 1, i + 2, \dots, n$ is the root of a heap.
- Therefore each of the two children of node i (i.e., nodes $2i$ and $2i + 1$) is the root of a heap.
- Therefore the call to *Heapify*(A, i) makes i the root of a heap.
- Further, all nodes which are not descendants of i are untouched by the call to *Heapify*(A, i), (Do you see why?) and so we can conclude that each node $i, i + 1, \dots, n$ is now the root of a heap.



Running Time of BuildHeap

- The number of elements of the heap at height h is $\leq \frac{n}{2^h}$, and the cost of running Heapify on a node of height h is $O(h)$.
- The root of a heap of n elements has height $\lfloor \log_2 n \rfloor$.
- Therefore the worst-case cost of running BuildHeap on a heap of n elements is bounded by

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O(n)$$

since the sum converges, so we don't care what the upper bound of the summation is.

Heap Properties

- Heaps give us partial information about the order of elements in a set.
- We can tell immediately what the largest element is.
- They are really cheap to build and can be stored in a simple array.
- This makes them very useful for various applications.

Algorithm 3 Heapsort(A)

```
1: BuildHeap( $A$ )
2: for  $i \leftarrow \text{length}[A]$  to 2 do
3:   exchange  $A[1] \leftrightarrow A[i]$ 
4:    $\text{heapsize}[A] \leftarrow \text{heapsize}[A] - 1$ 
5:   Heapify( $A, 1$ )
6: end for
```

The call to *BuildHeap* takes time $O(n)$. Each of the $n - 1$ calls to *Heapify* takes time $O(\log n)$. Hence the total running time is (in the worst case) $O(n \log n)$.

Priority Queues

Definition

A *priority queue* is a data structure that maintains a set S of elements, each with an associated value called a *key*. (As usual, the keys must be comparable.) The priority queue supports the following operations:

Insert(S, x) inserts the element x into the set S .

Maximum(S) returns the element of S with the largest key.

ExtractMax(S) removes and returns the element of S with the largest key.

IncreaseKey(S, x, k) increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

A priority queue can be implemented using a heap.

Priority Queue Operations

Algorithm 4 HeapMaximum(A)

1: **return** $A[1]$

Obviously, the run time is $O(1)$.

Algorithm 5 HeapExtractMax(A)

1: **if** $\text{heapsize}[A] < 1$ **then**
2: *ERROR – heap underflow*
3: **end if**
4: $\text{maxx} \leftarrow A[1]$
5: $A[1] \leftarrow A[\text{heapsize}[A]]$
6: $\text{heapsize}[A] \leftarrow \text{heapsize}[A] - 1$
7: $\text{Heapify}(A, 1)$
8: **return** maxx

Here the running time is dominated by the call to Heapify, so it is $O(\log n)$.

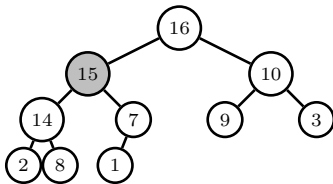
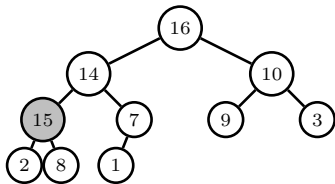
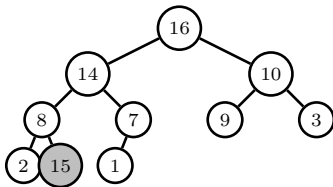
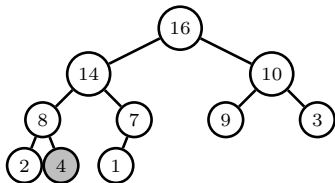
Priority Queue Operations

Algorithm 6 HeapIncreaseKey(A, i, key)

```
1: if  $\text{key} < A[i]$  then  
2:   ERROR – new key is smaller than current key  
3: end if  
4:  $A[i] \leftarrow \text{key}$   
5: while  $i > 1$  and  $A[\text{Parent}(i)] < A[i]$  do  
6:    $\text{exchange } A[i] \leftrightarrow A[\text{Parent}(i)]$   
7:    $i \leftarrow \text{Parent}(i)$   
8: end while
```

We just increase the key of $A[i]$, and then let that node “float up” to its proper position.

HeapIncreaseKey – Example



Algorithm 7 HeapInsert(A , key)

- 1: $heapsize[A] \leftarrow heapsize[A] + 1$
 - 2: $A[heapsize[A]] \leftarrow -\infty$
 - 3: $HeapIncreaseKey(A, heapsize[A], key)$
-

The running time here is again $O(\log_2 n)$.

Thus, a heap supports any priority queue operation on a set of size n in $O(\log n)$ time.