Priority Queues & Heaps

Priority Queues

Stores elements using a partial ordering based on priority such that the element of the highest priority is at the head queue.

Heaps

Heaps are data structures for implementing priority queues. Heaps support following two basic operations:

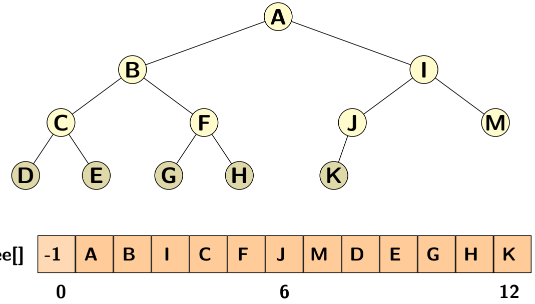
1. **insert()**: Is equivalent to enqueue. Enqueues elements according to their priorities maintaining the queue order.
2. **deleteMin()**: Is equivalent to dequeue. Finds minimum element (of the highest priority) and deletes it from the queue. The priority order of queue is restored by placing the next smallest at the head.

Implementing Priority Queues

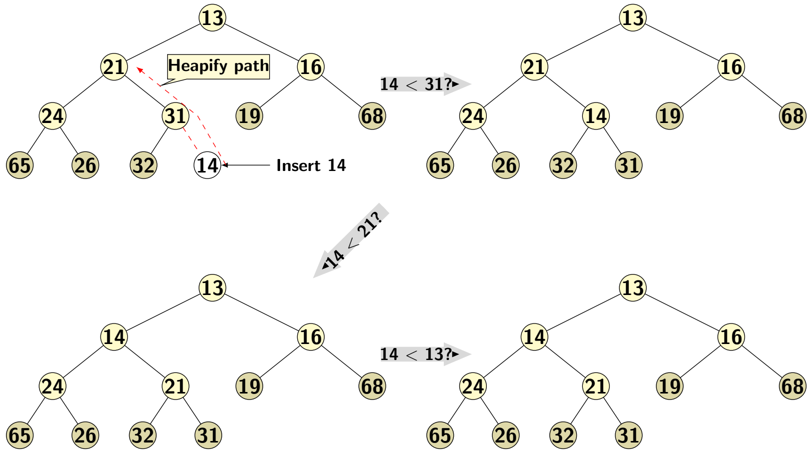
- Using unsorted linked lists
  - O(1) **insert**: always insert at the beginning
  - O(n) **deleteMin**: traverse list to find the minimum.
- Using sorted linked list
  - O(n) **insert**: inserts at the correct postion in the list.
  - O(1) **deleteMin**: gets the first element of the list.
- Using sorted array
  - O($n + \log n$) **insert**: inserts at the correct postion in the array then readjusts the array.
  - O(n) **deleteMin**: deletes the minimum, then r eadjusts the array.

Binary Heap

- The elements are organized in the form of a complete binary tree.
- For storing a complete binary tree use an array.
- The parent of a node at i at stored at $\lfloor i/2 \rfloor$.
- The tree also should satisfy heap order property.
  - For every node X, the element stored at X should be smaller or equal to the element stored at the parent of X.



Insertion into a Heap

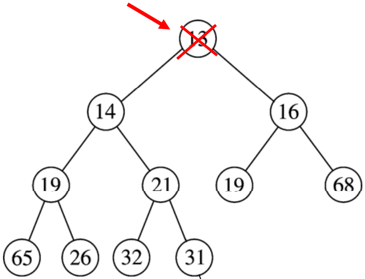


Insertion Algorithm

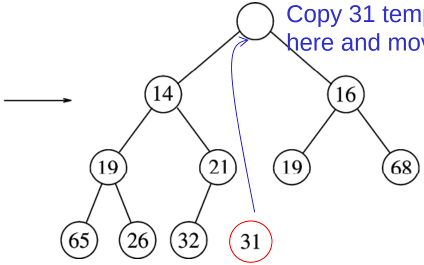
```
InsertToHeap(int x) {
        H[++n] = x;
        for(int k = n; k > 1; k /= 2) {
             if(H[k] > H[k / 2])
                  swap(H[k], H[k / 2]);
             else
                  break;
        }
}
```

- Since tree is a complete binary tree $h = O(\log n)$.
- Heapify procedure takes O($\log n$) time.

DeleteMIN from a Heap



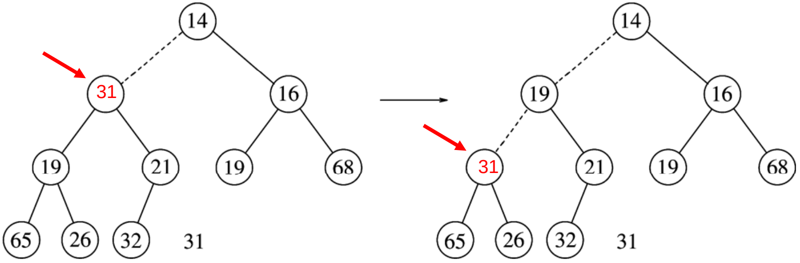
Make this position empty



Copy 31 temporarily here and move it dow

Is 31 > min(14,16)?
•Yes - swap 31 with min(14,16)

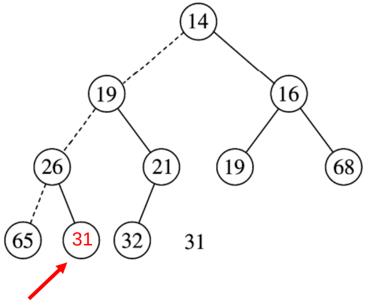
DeleteMIN from a Heap



Is 31 > min(19,21)?
•Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
•Yes - swap 31 with min(65,26)

DeleteMIN from a Heap



DeleteMIN Algorithm

```
DeleteMIN(HEAP H) {
    Last = H[n]; // Copy the last in temporary
    n = n-1;     // Update the last index
    i = 1;       // Starting index for heapify
    left = 2;    // Left child index
    right = 3;   // Right child index
    H[i] = Last; // Store last value into the root

    // Heapify in next slide
}
```

Heapify Algorithm

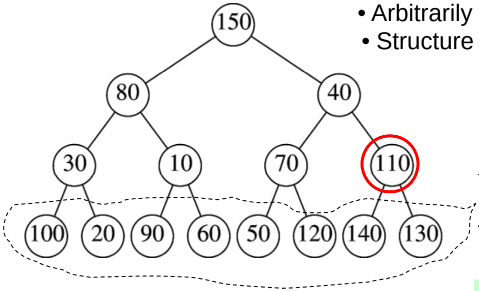
```
// Heapify operation
while (left ≤ n) {
    if (H[i] ≤ H[left] && H[i] ≤ H[right])
        return; // Heap property restored
    if (H[right] ≤ H[left]) {// swap with left
        swap(H[i], H[left]);
        i = left; // Heapify left subtree
    } else {// swap with right
        swap(H[i], H[right]);
        i = right; // Heapify right subtree
    }
    left = 2*i;
    right = left+1;
}
```

Building Heap

- A heap of n elements formed placing elements randomly into a binary tree.
- At leaf level (single element) heap property trivially holds.
- But heap property should be preserved at all levels.
- Pair up the heaps bottom up, by examining heap property for each internal node.
- Move the elements down from upper levels as needed.

Building Heap: Initial Configuration

Input: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }



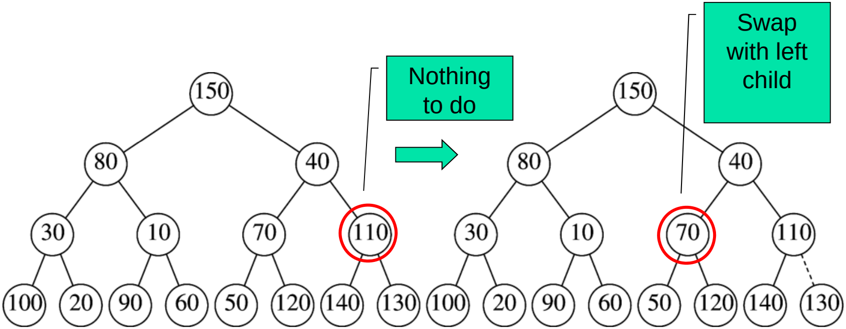
- Arbitrarily assign elements to heap nodes
- Structure property satisfied

Leaves are all valid heaps (implicitly)

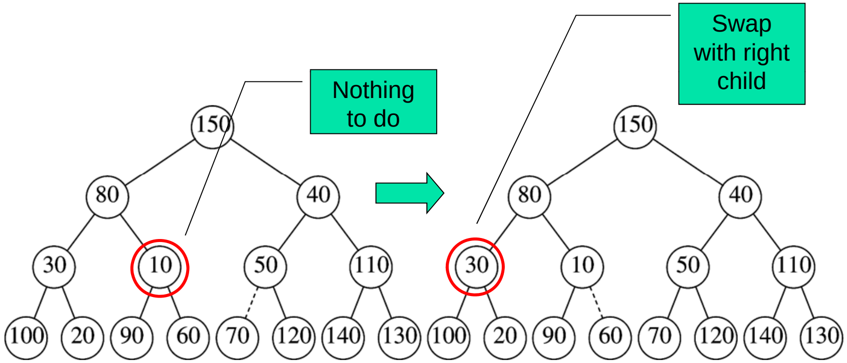
- Heap order property violated
- Leaves are all valid heaps (implicit)

So, let us look at each internal node,
from bottom to top, and fix it

Building Heap: Intermediate Configuration

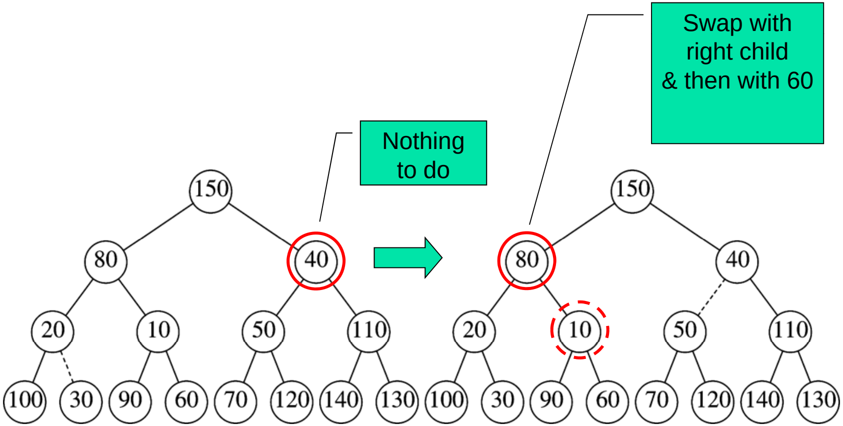


Building Heap: Intermediate Configuration



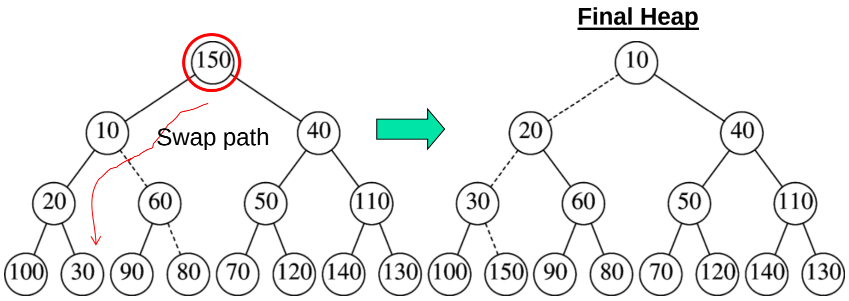
Dotted lines show path of percolating down

Building Heap: Intermediate Configuration



Dotted lines show path of percolating down

Building Heap: Final Configuration



Dotted lines show path of percolating down

Building Heap: Analysis

Time for Build Heap

Build heap takes time O(Sum of heights of all nodes).

Proof

- A node can go down from its own level along a tree path until it reaches a height h.
- At level ℓ there are 2^ℓ nodes, $0 \leq \ell \leq h$.

Therefore,

$$\begin{aligned}\sum_{\ell=o}^{h} 2^\ell \times (h-\ell) &= \sum_{\ell=o}^{h} 2^\ell \times h - \sum_{\ell=0}^{h} \ell \times 2^\ell \\ &= h(2^{h+1}-1) - S,\end{aligned}$$

where, $S = 1.2 + 2.2^2 + \ldots + h.2^h$.

Building Heap: Analysis

Evaluating S

$S = (h-1)2^{h+1} - 2$

Proof

$$\begin{array}{rcl} 2S & = & 1.2^2 + 2.2^3 + 3.2^4 + \ldots + (h-1)2^h) + h.2^{h+1} \\ 2S - S & = & -(2 + 2^2 + 2^3 + \ldots + 2^h) + h.2^{h+1} \\ S & = & -(2^{h+1} - 1 - 1) + h.2^{h+1} = (h-1)2^{h+1} - 2 \end{array}$$

Running time

Therefore, running time for building a heap is

$$h.(2^{h+1} - 1) - (h-1)2^{h+1} + 2 = 2^{h+1} - h + 2.$$

Any Other Operations?

- Sometimes **increaseKey()** and **decreaseKey()** are also considered as basic operations on heaps.
- However, increase and decrease key operations require position of a key to be accessible in heap.
- Exposing the position of each key (i.e. internals of implementation) is equivalent to cheating in ADT approach,