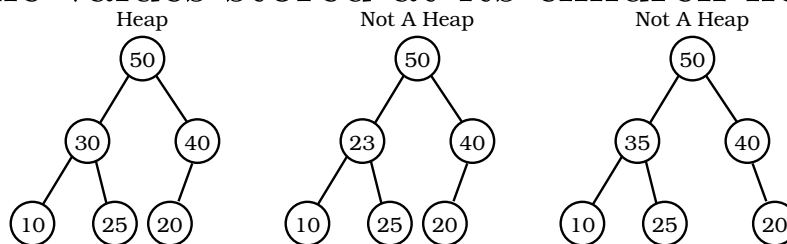
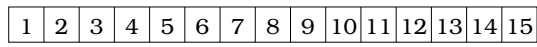
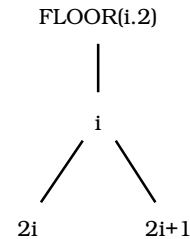
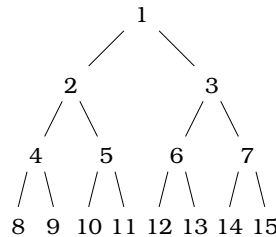


Max Heaps

- Each node stores one value, but the values may be repeated (i.e., it is a multiset, rather than a set).
- Complete binary tree (All possible nodes at each level are present except possibly for the last level. All the missing nodes in the last level, if any, are located to the right of the nodes that are present.)
- The value stored at node v is at least as large as the values stored at its children nodes.



Representation Without Pointers



```

template<class T>
class MaxHeap {
public:
    MaxHeap(int MaxHeapSize = 10);
    ~MaxHeap() {delete [] heap;}
private:
    int CurrentSize, MaxSize;
    T *heap; // element array };
template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{ MaxSize = MaxHeapSize;
  heap = new T[MaxSize+1];
  CurrentSize = 0; }
  
```

Height of Heap with n nodes

- Full Binary Tree (no missing nodes).
 - Number of nodes at level 1 is 2^0 , at level 2 is 2^1 , at level 3 is 2^2 , ..., at level h is 2^{h-1} .
 - Therefore, $n = \sum_{i=0}^{h-1} 2^i$
 - So, $n = 2^h - 1$, and $n + 1 = 2^h$.
 - So, $h = \log_2(n + 1)$.
 - Or h is $O(\log n)$.
- Not Full Binary Tree (missing nodes).
 - Fill it up. Number of nodes is $m < 2n$.
 - Therefore, $h = \log_2(m + 1) < \log_2(2n + 1)$.
 - Or h is $O(\log n)$.

Insert x

- Assign x to the next available position.
- If x is greater than the value of its parent then swap them.
- repeat the above operation till you reach the root or it does not hold.

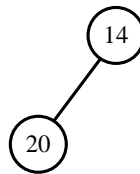
Example for Insert

EMPTY HEAP

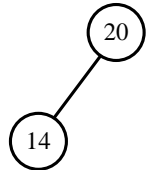
Insert 14



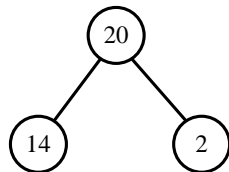
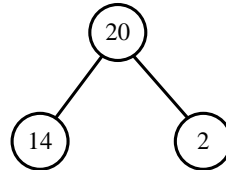
Insert 20



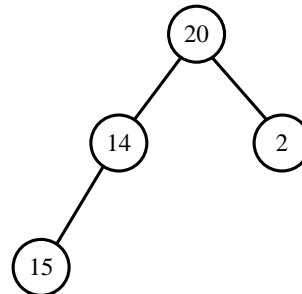
Move Up



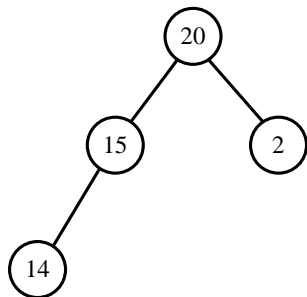
Insert 2



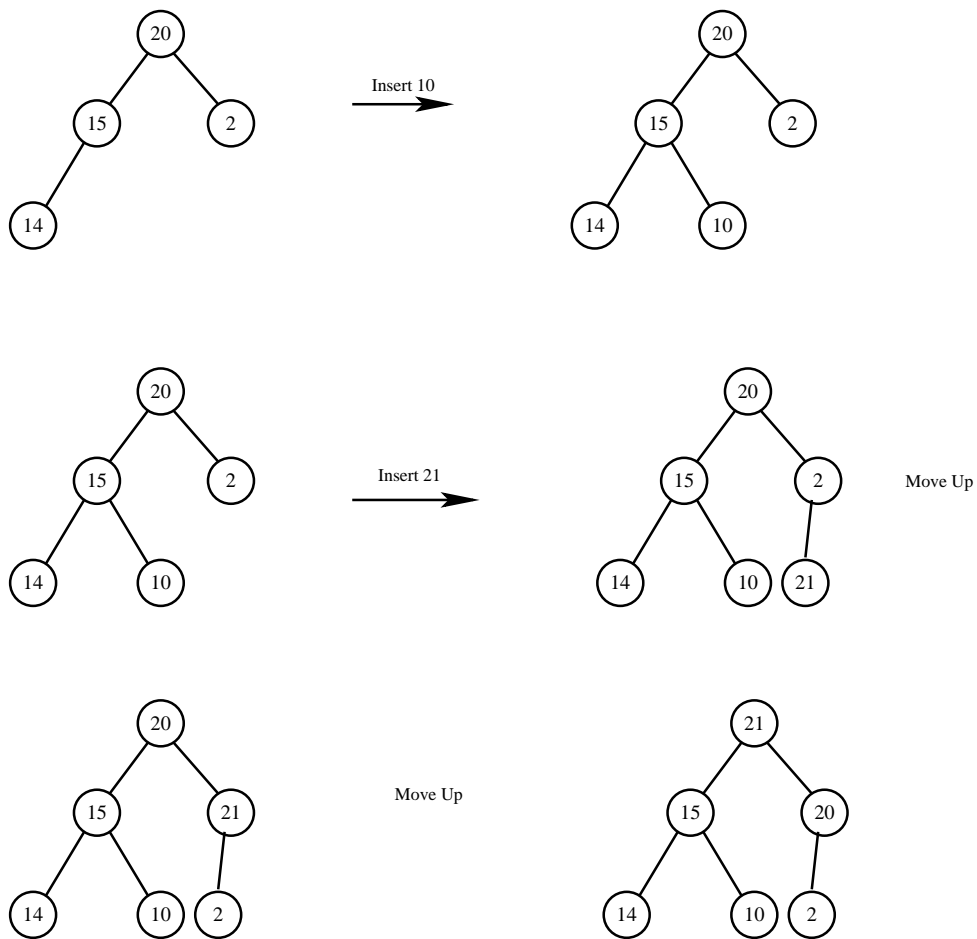
Insert 15



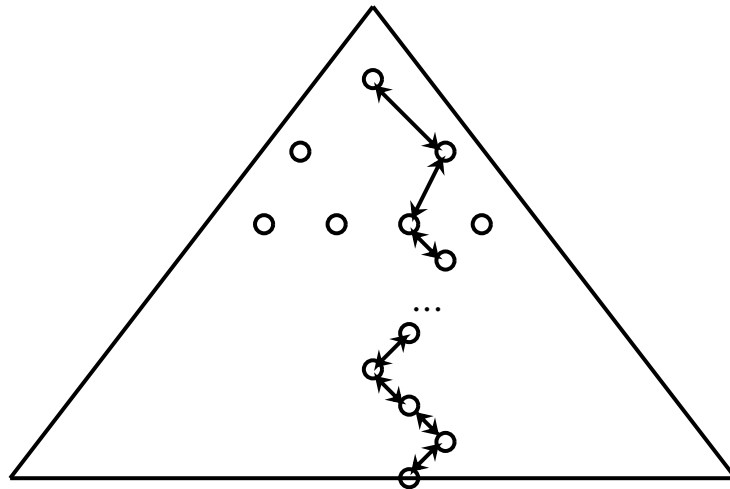
Move Up



Example for Insert



General Idea



Just move them down (instead of swap) and store x in the appropriate place.

```
template<class T>
MaxHeap<T>& MaxHeap<T>::Insert(const T& x)
{ // Insert x into the max heap.
    if (CurrentSize == MaxSize)
        throw NoMem(); // no space

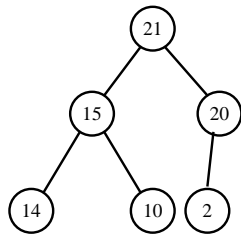
    // find place for x
    // i starts at new leaf and moves up tree
    int i = ++CurrentSize;
    while (i != 1 && x > heap[i/2]) {
        // cannot put x in heap[i]
        heap[i] = heap[i/2]; // move down
        i /= 2;} // move to parent

    heap[i] = x;
}
```

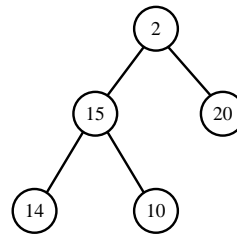
Time Complexity is $O(\log n)$.

Deletion

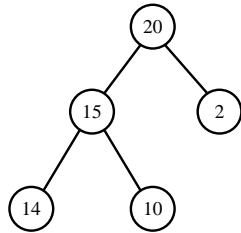
- Copy the value in the root and that is what will be returned.
- Move the last element to the root.
- If one of its children has a larger value, then move it to the child with largest value.
- Repeat the above until you reach a leaf or the above condition does not hold.



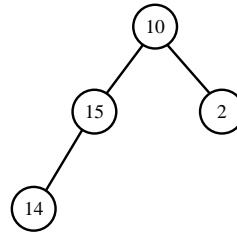
DeleteMax
→



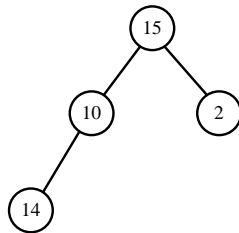
Move Down



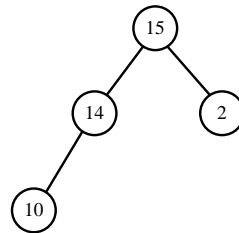
DeleteMax
→



Move Down



Move Down

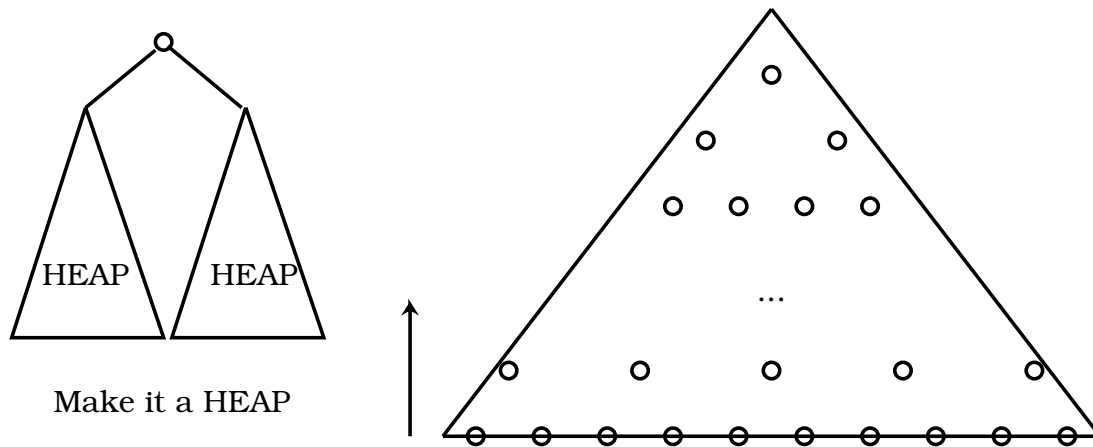



```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{ if (CurrentSize == 0)
    throw OutOfBounds(); // empty
  x = heap[1]; // max element
  T y = heap[CurrentSize--]; // last element
  int i = 1, // current node of heap
      ci = 2; // child of i
  while (ci <= CurrentSize) {
    if (ci < CurrentSize &&
        heap[ci] < heap[ci+1]) ci++;
    // can we put y in heap[ci]?
    if (y >= heap[ci]) break; // yes
    // no
    heap[i] = heap[ci]; // move child up
    i = ci; // move down a level
    ci *= 2; }
  heap[i] = y;
}
```

Time Complexity is $O(\log n)$.

n Sequential Inserts

- By the above procedure it takes $O(n \log n)$ time (actually Ω too).
- New procedure takes $O(n)$ Time.



Time Complexity

At level j there are 2^{j-1} vertices and making that subtree a heap takes $h - j$ operations (assume complete tree).

$$T(n) = \sum_{j=1}^h 2^{j-1} (h - j)$$

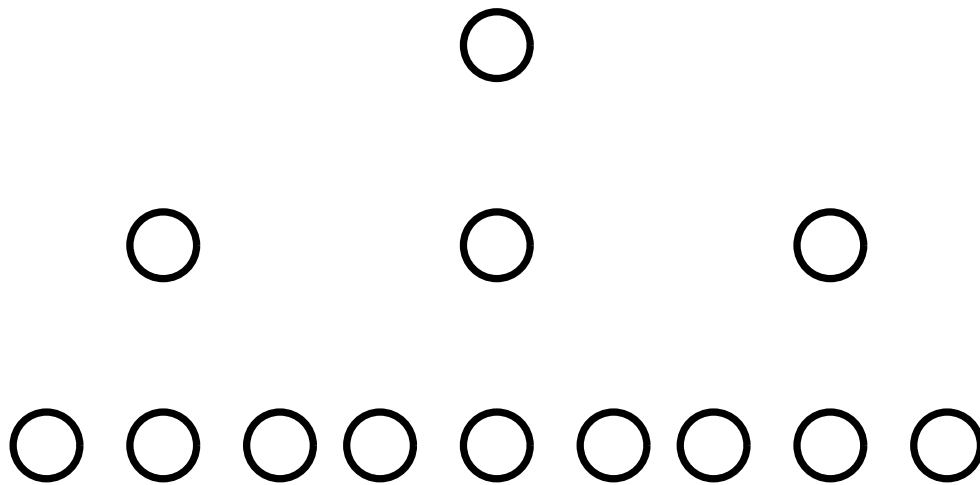
$$\begin{aligned}T(n) &= \sum_{j=1}^h 2^{j-1}(h-j) \\&= \sum_{i=0}^{h-1} 2^{h-1-i}(i) \\&= 2^{h-1} \sum_{i=0}^{h-1} \frac{i}{2^i} \\&= 2^{h-1} \cdot \frac{2^h - 1 - h}{2^{h-1}} \\&= 2^h - h - 1 \\&= O(n)\end{aligned}$$

```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size,
                           int ArraySize)
{
    delete [] heap;
    heap = a;
    CurrentSize = size;
    MaxSize = ArraySize;
    for (int i = CurrentSize/2; i >= 1; i--) {
        T y = heap[i]; // root of subtree
        int ci = 2*i; // parent of c is target
                    // location for y
        while (ci <= CurrentSize) {
            // heap[ci] should be larger sibling
            if (ci < CurrentSize &&
                heap[ci] < heap[ci+1]) ci++;
            // can we put y in heap[ci]?
            if (y >= heap[ci]) break; // yes
            // no
            heap[ci/2] = heap[ci]; // move child up
            ci *= 2; // move down a level }
        heap[ci/2] = y;} }
```

Other Operations

- Decrease Value $O(\log n)$.
- Increase Value $O(\log n)$.
- Delete element (if you know its position) $O(\log n)$.
- Delete element (if you do NOT know its position) $O(n)$.

Ternary Heaps (d-heaps with $d=3$)



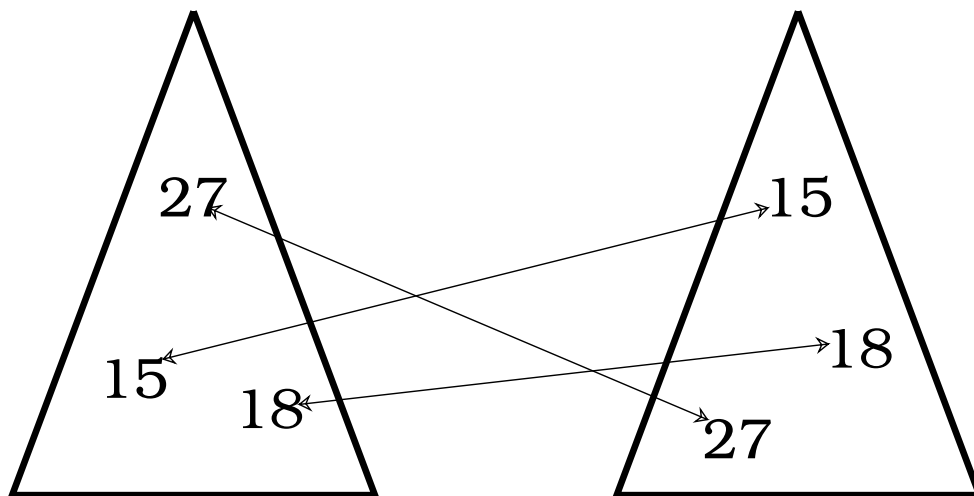
...

Height is $\log_3 n$

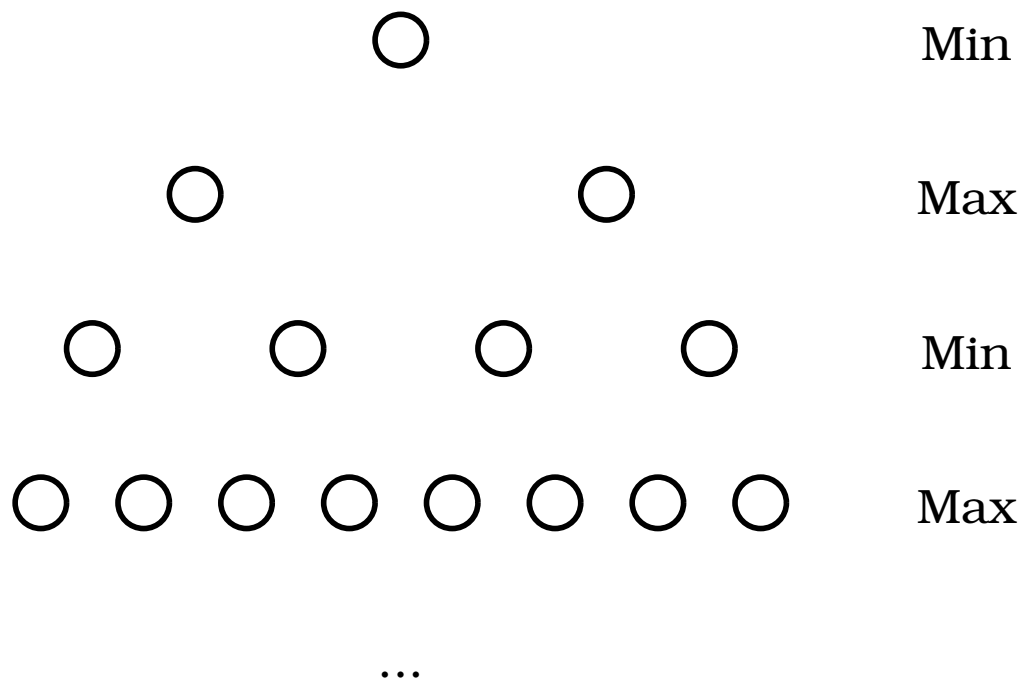
Matching Heaps

Max

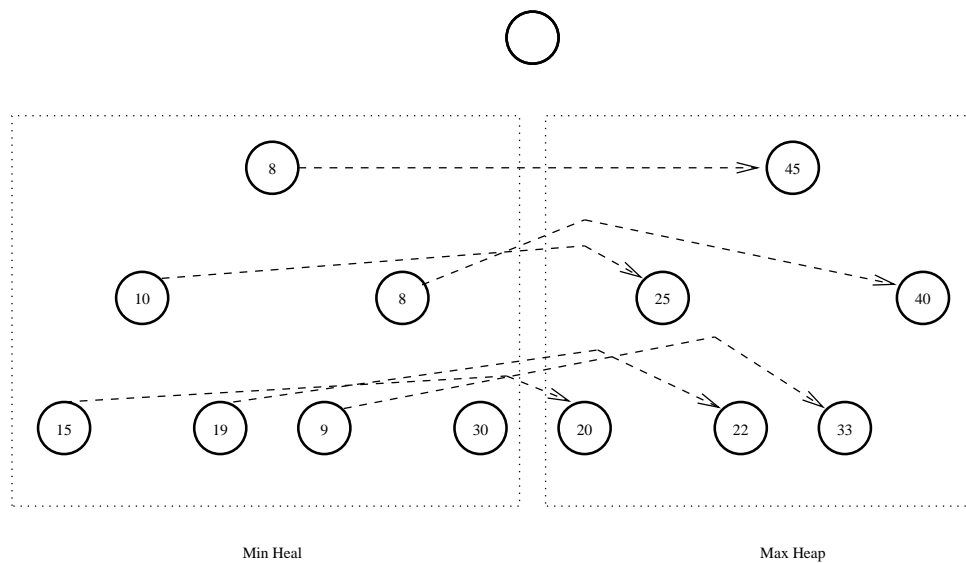
Min



Min-Max Heaps



Deaps



Dotted arrow $a \rightarrow b$ means that $a \leq b$.