Lecture 21: Priority Queues & Heaps

PIC 10B
Todd Wittman

Priority Queues

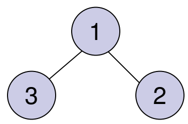
- Suppose we are keeping track of a line of patients entering a hospital emergency room.
- In a queue, the first person to enter the emergency room is the first person to see the doctor (FIFO).
- But what if a patient comes in who needs immediate medical attention (priority 1)?
- In a priority queue, each element is assigned a unique priority.
- Elements with the highest priority should be the first to leave the queue (HPFO?).

    3, 2, 8 1, 5, 6, 7   The first one to leave is the 1.

- There are several ways to implement a priority queue. We will use the data structure called a heap.

The Heap Data Structure

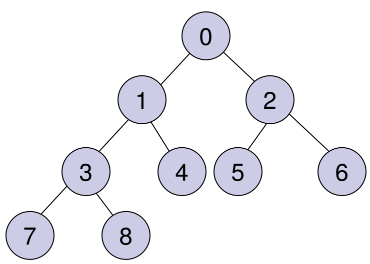
- A <u>heap</u> *(or min-heap)* is a data structure that is a cross between a queue and a binary search tree. Essentially it has the shape of a binary tree and the ordering of a queue.
- Each node of the heap stores a value and has two children, just like in a binary tree.

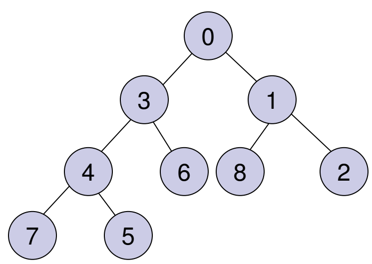


- **<u>The Heap Property</u>**: Every node has a value smaller than both its children.
- **<u>The Shape Property</u>**: Every level of the heap is full, except for possibly the last level which is filled left to right.
- **<u>The Queue Property:</u>** Elements can be removed or accessed only at the root node. (No traversals.)

The Heap Concept

- Here's an example of 2 possible heaps on the numbers 0-8.

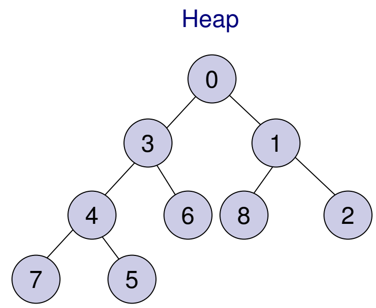


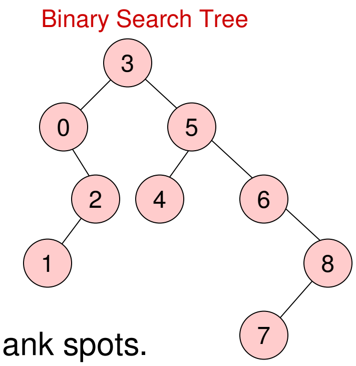


- Note that the Heap Property implies that the minimum value is always at the root.
- So the min will be the first value to leave the heap.
- The Shape Property implies that tree is always perfectly balanced.
- Every heap with same number of nodes has the same shape.

Heaps vs. Binary Search Trees

- They may look similar, but the ordering is very different.

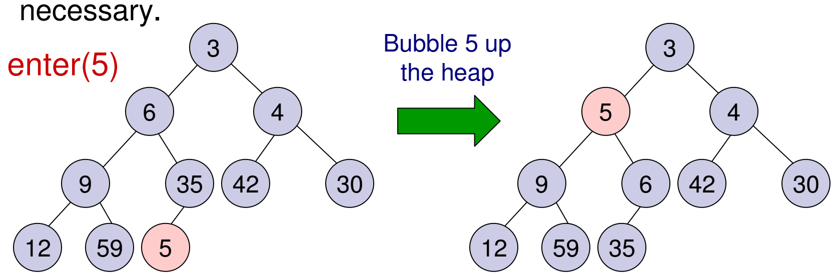




- The BST may contain some blank spots.
- For a BST, the height h may vary: $\lfloor \log_2 N \rfloor \le h \le N-1$
- For a heap, the height h is always: $h = \lfloor \log_2 N \rfloor$
- You can traverse all the nodes of a tree. In a heap, you can only look at the root.

Entering a Heap

- To enter a heap, we put the new value in the next position at the bottom of the heap.
- To maintain the Heap Property, we then "bubble" or "sift" the value up the heap by swapping the value with its parent if necessary.

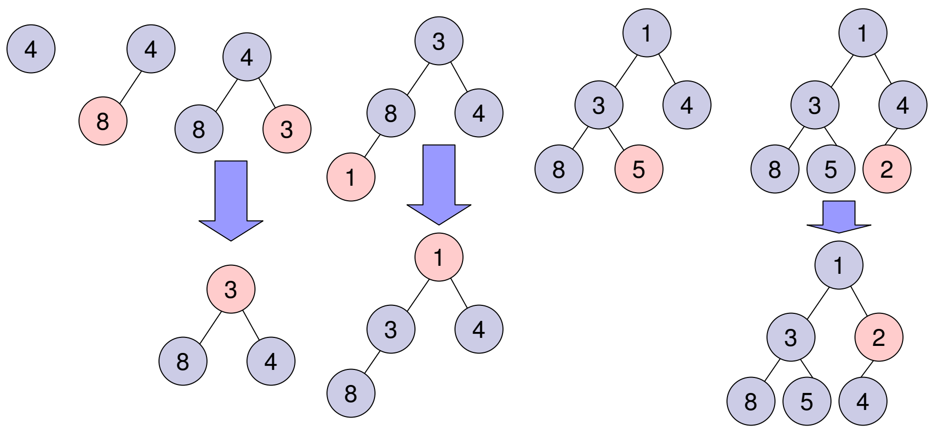


- Note we have to swap at most *h* nodes, so inserting is O(logN).

Building A Heap

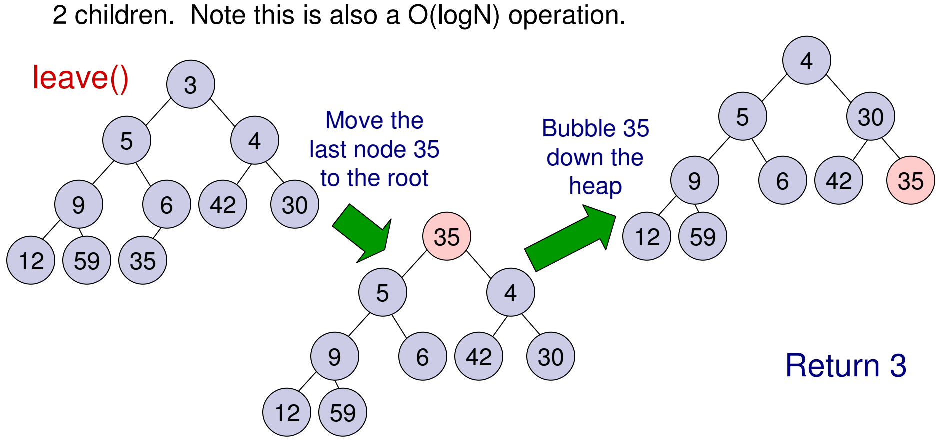
- To create a heap, we repeatedly insert the new values.
- <u>Ex</u> Build a heap by inserting the following values in order:

4, 8, 3, 1, 5, 2



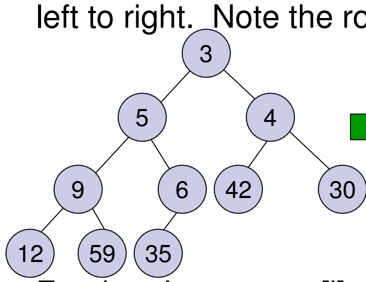
Leaving a Heap

- Only the root node can be removed from a heap.
- We swap the root value with the value in the last position of the heap and delete the last node.
- We then bubble this value down the tree, swapping with the smaller of its 2 children. Note this is also a O(logN) operation.



Implementing a Heap with a Vector

- Because the heap is full, we can implement a heap using a vector rather than using a tree and pointers.
- Read the heap in a level-order traversal, reading each row left to right. Note the root is at v[0].



3	5	4	9	6	42	30	12	59	35
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

- For the element at v[i] we have:
  - Left child: v[2i+1]
  - Right child: v[2i+2]
  - Parent: v[$\lfloor$(i-1)/2$\rfloor$]

The Heap Class

- The Heap class declaration looks a lot like the Queue class.

```
template <typename T>
class Heap {
   public:
         Heap();
         bool isEmpty() const;
         T peek() const;
         void enter(T value);
         T leave();
   private:
         vector<T> v;
         void swap(int i, int j);
   };
```

- Do we need the Big 4?

Basic Heap Functions

```
template <typename T>
Heap<T>::Heap() {
    v.resize(0);
}

template <typename T>
T Heap<T>::peek() const {
    return v[0];
}
```

```
template <typename T>
bool Heap<T>::isEmpty() const {
    return v.size()==0;
}

template <typename T>
void Heap<T>::swap(int i, int j) {
    T temp = v[i];
    v[i] = v[j];
    v[j] = temp;
    return;
}
```

Entering a Heap

```
template <typename T>
void Heap<T>::enter(T value) {
    v.push_back(value);
    int pos = v.size()-1;
    int parent = (int) (pos-1)/2;
    while (parent>=0 && v[pos]<v[parent]) {
        swap(parent,pos);
        pos = parent;
        parent = (int) (pos-1)/2;
    }
    return;
}
```

Add new value to the bottom of the heap.

Bubble the new value up by swapping with its parent.

Leaving a Heap

```
template <typename T>
T Heap<T>::leave() {
    T top = v[0];
    v[0] = v[v.size()-1];
    v.pop_back();
    int pos = 0;
    bool continueBubbleDown = (2*pos+1 < v.size());
    while (continueBubbleDown) {
        if (2*pos+1 >= v.size())
            continueBubbleDown = false;
        else if (2*pos+2 >= v.size())  {
            if (v[pos] > v[2*pos+1]) {
                swap(pos,2*pos+1);
                pos = 2*pos+1;
            }
            continueBubbleDown = false;
        }
```

Store the value at the root. We return this at the end.

Copy value at bottom into root and delete last node.

Case 1: No children means stop.

Case 2: Only left child. We have reached bottom of heap, so perform at most 1 more swap.

More....

Leaving a Heap

```
    else
        if (v[pos]>v[2*pos+1] && v[2*pos+1]<=v[2*pos+2]) {
            swap(pos,2*pos+1);
            pos = 2*pos+1;
        }
        else if (v[pos]>v[2*pos+2]) {
            swap(pos,2*pos+2);
            pos = 2*pos+2;
        }
        else
            continueBubbleDown = false;
    }
    return top;
}
```

Case 3: 2 children

Left child is smaller.

Right child is smaller.

If both children are bigger, stop.

Return the value that was at the root.

Example Program

- Put 25 random 4-letter words into a heap and then remove them one at a time.

```
int main() {
    Heap<string> H;
    string word = "aaaa";
    for (int i=0; i<25; i++) {
        for (int j=0; j<4; j++)
            word[j] = (char) (rand()%26+(int)'a');
        H.enter(word);
    }
    while (!H.isEmpty())
        cout << H.leave() << "\n";
    return 0;
}
```

- Note this prints the words in sorted order from smallest to largest.

Heap Sort

- We can sort the elements in a vector by placing the values into a heap and removing them one at a time.
- Each enter/leave is O(logN), so doing N inserts/leaves takes O(NlogN) time.
- So we can heap sort a list in O(NlogN) time.
- But this requires an additional O(N) memory for the heap vector.
- There is a special way to "heapify" a given vector, so that the sorting is done *in place*.
- You will implement heap sort using this trick for this week's homework. Do <u>not</u> use the Heap class for heap sort.