Warm Up!

• What is the maximum number of total nodes in a binary tree of height $h$?
  • Height: The number of edges in the path from root to the deepest leaf
  • $2^{h+1} - 1$
  • 3, 7, 15, 31
  • $\Theta(2^h)$

• If I have $n$ nodes in a binary tree, what is the its minimum height?
  • The height of the tree must be high enough that $n$ nodes is possible
  • $2^{h+1} - 1 \geq n$
  • $2^{h+1} \geq n + 1$
  • $\log 2^{h+1} \geq \log(n + 1)$
  • $h \geq \log(n + 1) - 1$
  • $\Theta(\log n)$
Trees for Heaps

- Binary Trees:
  - The branching factor is 2
  - Every node has $\leq 2$ children

- Complete Tree:
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right
What is it?
• A collection of items and their “priorities”
• Allows quick access/removal to the “top priority” thing

What Operations do we need?
• insert(item, priority)
  • Add a new item to the PQ with indicated priority
  • Usually, smaller priority value means more important
• deleteMin
  • Remove and return the “top priority” item from the queue
• Is_empty
  • Indicate whether or not there are items still on the queue

Note: the “priority” value can be any type/class so long as it’s comparable (i.e. you can use “<” or “compareTo” with it)
Thinking through implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case time to insert</th>
<th>Worst case time to deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Circular Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

Note: Assume we know the maximum size of the PQ in advance
Heap – Priority Queue Data Structure

- Idea: We need to keep some ordering, but it doesn’t need to be perfectly sorted
- $\Theta(\log n)$ worst case for deleteMin and insert

Heap property
Heap – Priority Queue Data Structure

• Idea: We need to keep some ordering, but it doesn’t need to be perfectly sorted
• $\Theta(\log n)$ worst case for deleteMin and insert
Challenge!

• What is the maximum number of total nodes in a binary tree of height $h$?
  • $2^{h+1} - 1$
  • $\Theta(2^h)$

• If I have $n$ nodes in a binary tree, what is its minimum height?
  • $\Theta(\log n)$

• Heap Idea:
  • If $n$ values are inserted into a complete tree, the height will be roughly $\log n$
  • Ensure each insert and deleteMin requires just one “trip” from root to leaf
Heap Data Structure

• Keep items in a complete binary tree
• Maintain the “Heap Property” of the tree
  • Every node’s priority is \( \leq \) its children’s priority
• Where is the min? root
• How do I insert?
• How do I deleteMin?
• How to do it in Java?
Heap Insert

insert(item){
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority) {
        swap item with parent
    }
}

1.5
Heap Insert

insert(item){
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
Heap Insert

insert(item){
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority) {
        swap item with parent
    }
}

Percolate Up
Heap Insert

$$\log n$$

```plaintext
insert(item){
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Percolate Up
Heap Insert

```
insert(item)
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority)
        swap item with parent
}
```
Heap deleteMin

deleteMin()
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
Heap deleteMin

deleMin()
    min = root
    br = bottom-right item
    move br to the root
    while (br > either of its children)
        swap br with its smallest child
    
    return min
Heap deleteMin

deleteMin()
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
Heap deleteMin $\Theta(\log n)$

```plaintext
deleteMin()
    min = root
    br = bottom-right item
    move br to the root
    while (br > either of its children){
        swap br with its smallest child
    }
    return min
```
Heap deleteMin

deleteMin()
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
Percolate Up and Down

• Goal: restore the “Heap Property”

• Percolate Up:
  • Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent

• Percolate Down:
  • Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger

• Worst case running time of each:
  • $\Theta(\log n)$
Representing a Heap

- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node \( i \): \( \left\lfloor \frac{i}{2} \right\rfloor \)
- Left child of node \( i \): \( 2 \cdot i \)
- Right child of node \( i \): \( 2 \cdot i + 1 \)
- Location of the leaves: last half
insert(item){
    if(size == arr.length – 1){resize();}
    size++;
    arr[size] = item;
    percolateUp(size)
}
Percolate Up

\[
\text{percolateUp}(i)\{
\text{parent} = i/2; \quad \text{// index of parent}
\text{val} = \text{arr}[i]; \quad \text{// value at location}
\text{while}(i > 1 \text{ } \&\& \text{ } \text{arr}[i] < \text{arr}[\text{parent}]){ \quad \text{// until location is root or heap property holds}
\text{arr}[i] = \text{arr}[\text{parent}]; \quad \text{// move parent value to this location}
\text{arr}[\text{parent}] = \text{val}; \quad \text{// put current value into parent’s location}
\text{i} = \text{parent}; \quad \text{// make current location the parent}
\text{parent} = i/2; \quad \text{// update new parent}
\}
\]
DeleteMin Psuedocode

deleteMin()

    theMin = arr[1];
    arr[1] = arr[size];
    size--;
    percolateDown(1);
    return theMin;

}
Percolate Down

\[ \text{PercolateDown}(i) \{
\]

\[ \text{left} = \text{index of left child} = i \times 2; \]

\[ \text{right} = \text{index of right child} = i \times 2 + 1; \]

\[ \text{val} = \text{arr}[i]; \]

\[ \text{while (left <= size)} \{ \text{until location is leaf} \]

\[ \text{toSwap} = \text{right}; \]

\[ \text{if (right > size || arr[left] < arr[right])} \{ \text{if there is no right child or if left child is smaller} \]

\[ \text{toSwap} = \text{left; swap with left} \]

\[ \} \text{now toSwap has the smaller of left/right, or left if right does not exist} \]

\[ \text{if (arr[toSwap] < val)} \{ \text{if the smaller child is less than the current value} \]

\[ \text{arr}[i] = \text{arr[toSwap];} \]

\[ \text{arr[toSwap]} = \text{val; swap parent with smaller child} \]

\[ \text{i} = \text{toSwap; update current node to be smaller child} \]

\[ \text{left} = \text{i} \times 2; \]

\[ \text{right} = \text{i} \times 2 + 1; \]

\[ \} \text{else break;} \] \text{if we don’t swap, then heal property holds} \]

\[ \}
\]
Other Operations

• Increase Key
  • Given the index of an item in the PQ, subtract from its priority value

• Decrease Key
  • Given the index of an item in the PQ, add to its priority value

• Remove
  • Given the item at the given index from the PQ
Aside: Expected Running time of Insert
Building a Heap From “Scratch”

• Suppose we had $n$ items and wanted to “heapify” them

Violate Heap Property!

Two ways for “fix” the heap:
1) Percolate Up
2) Percolate Down
Floyd’s buildHeap method

• Working towards the root, one row at a time, percolate down

buildHeap()
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--)
    {
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
5
/  \
6   1
/ \  /
2  3 15

8  6
/ \  /
2  3 14
```

```
5  6 10  3 15  8  7 14  2  1
0  1  2  3  4  5  6  7  8  9  10

Violate Heap Property!

buildHeap()
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
```
Floyd’s buildHeap method

- Suppose we had $n$ items and wanted to “heapify” them

```java
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

![Heap Property Violation Diagram]
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
```
How long did this take?

• Worst case running time of buildHeap:
  • No node can percolate down more than the height of its subtree
    • When i is a leaf:
    • When i is second-from-last level:
    • When i is third-from-last level:

• Overall Running time:

```java
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```