CSCI 241
Lecture 15:
Priority Queues
Heaps

There will be Socratic today!
Announcements

• Quiz today: the usual

• A2 is due Monday night
Goals

• Understand the purpose and interface of the Priority Queue ADT.

• Know the definition and properties of a heap.

• Know how heaps are stored in practice.

• Know how to perform (on paper) and implement (in code) add, peek, and poll.
Abstract Data Types

- **interface** `List` defines an “abstract data type”
- It has public methods: `add`, `get`, `remove`, ...
- Various classes **implement** `List`:

<table>
<thead>
<tr>
<th>Class:</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backing storage:</strong></td>
<td>array</td>
<td>chained nodes</td>
</tr>
<tr>
<td><code>add(i, val)</code></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>add(0, val)</code></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>add(n, val)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>get(i)</code></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>get(0)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>get(n)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Our next two topics (and the subject of A3):

- **Priority Queues**
- Hashing, HashSets, **HashMaps**
Priority Queues
Queue vs Priority Queue

add (enqueue): inserts an item into the queue
remove (dequeue): removes the first item to be inserted (FIFO)

add (enqueue): inserts an item into the queue
remove (poll): remove the highest-priority item from the queue
Uses for Priority Queues

- Computer Graphics: mesh simplification
- Graph algorithms: shortest paths, spanning trees
- Statistics: maintain largest M values in a sequence
- Graphics and simulation: "next time of contact" for colliding bodies
- AI Path Planning: A* search (e.g., Map directions)
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
Priority Queues

Like a Queue, but:

• Each item in the queue has an associated priority which is some type that implements Comparable

• `remove()` returns item with the “highest priority”
  • or, the element with the “smallest” associated priority value

• Ties are broken arbitrarily
interface PriorityQueue<E> {
    boolean add(E e); // insert e
    E peek(); // return min element
    E poll(); // remove/return min element
    void clear();
    boolean contains(E e);
    boolean remove(E e);
    int size();
    Iterator<E> iterator();
}
Priority Queue: LinkedList implementation

An unordered list:
• **add()** - new element at front of list - $O(1)$
• **poll()** - requires searching the list -
• **peek()** - requires searching the list -

An ordered list:
• **add()** - requires searching the list - $O(n)$
• **poll()** - min element is kept at front -
• **peek()** - min element is kept at front -

Exercise: fill in all the runtimes.
Priority Queue: LinkedList implementation

An unordered list:

- **add()** - new element at front of list - O(1)
- **poll()** - requires searching the list - O(n)
- **peek()** - requires searching the list - O(n)

An ordered list:

- **add()** - requires searching the list - O(n)
- **poll()** - min element is kept at front - O(1)
- **peek()** - min element is kept at front - O(1)
Question to ponder:

What would be the runtime of add, peek, and poll if you implement a Priority Queue using a BST?

What about an AVL tree?
Priority Queue: heap implementation

- A **heap** is a **concrete** data structure that can be used to **implement** a Priority Queue

- Better runtime complexity than either list implementation:
  - **peek()** is $O(1)$
  - **poll()** is $O(\log n)$
  - **add()** is $O(\log n)$

- Not to be confused with **heap memory**, where the Java virtual machine allocates space for objects – different usage of the word heap.
A heap is a special binary tree with two additional properties.
A heap is a special binary tree.

1. **Heap Order Invariant:**
   Each element $\geq$ its parent.
A heap is a special binary tree.

1. **Heap Order Invariant:** Each element ≥ its parent.
A heap is a special binary tree.

1. **Heap Order Invariant:**
   Each element \( \geq \) its parent.
A heap is a special binary tree.

2. **Complete**: no holes!
- All levels except the last are **full**.
- Nodes in last level are as far left as possible.
A heap is a special binary tree.

2. **Complete**: no holes!
   - All levels except the last are full.
   - Nodes in last level are as far left as possible.
A heap is a special binary tree.

2. **Complete**: no holes!
- All levels except the last are full.
- Nodes in last level are as far left as possible.

```
          4
         /   \
        6     14
       / \
      21   8
     /  \
    22   38
   / \
  55   10
 /   \
20   19  35
```

→ as far left as possible
Which of these are valid heaps?

1. Each element $\geq$ its parent.
2. The tree is complete.
Heap it real.

Which of these are valid heaps?

1. Each element >= its parent.
2. The tree is complete.
Heap it real.

Which of these are valid heaps?

1. Each element $\geq$ its parent.
2. The tree is complete.
Which of these are valid heaps?

1. Each element >= its parent.
2. The tree is complete.
Heap it real.

Which of these are valid heaps?

1. Each element >= its parent.
2. The tree is complete
Heap it real.

Which of these are valid heaps?

1. Each element >= its parent.
2. The tree is complete.
Heap operations

interface PriorityQueue<E> {
    boolean add(E e); // insert e
    E peek(); // return min element
    E poll(); // remove/return min element
    void clear();
    boolean contains(E e);
    boolean remove(E e);
    int size();
    Iterator<E> iterator();
}
void add(E e);

Algorithm:
• Add e in the wrong place
• While e is in the wrong place
  • move e towards the right place
```c
void add(E e);
```
void add(E e);
void add(E e);
void add(E e);
void add(E e);
`void add(E e);`

**Algorithm:**
- Add `e` in the wrong place *(the leftmost empty leaf)*
- While `e` is in the wrong place *(it is less than its parent)*
  - move `e` towards the right place *(swap with parent)*

The heap invariant is maintained!
If $k$ is less than $h$, the height of the tree, how many nodes are at depth $k$?

A. We can't know for sure
B. $2^k$
C. $2^{k-1}$
D. $2^k - 1$
If \( k \) is less than \( h \), the height of the tree, how many nodes are at depth \( k \)?

A. We can't know for sure
B. \( 2^k \)
C. \( 2^{k-1} \)
D. \( 2^k - 1 \)
If $k$ is less than $h$, the height of the tree, how many nodes are at depth $k$?

A. We can't know for sure
B. $2^k$
C. $2^{k-1}$
D. $2^k - 1$
So... runtime?

\[ O(\text{swaps}) \cdot \text{runtime of swap} \]

\[ O(h) \quad O(1) \]

\[ O(h) \]

\[ h = O(\log n) \]

\[ \text{add(e)} \quad \in \quad O(\log n)! \]
Runtime.
Runtime.

- \( O(\text{number of swap/bubble operations}) = O(\text{height}) \)
Runtime.

- $O(\text{number of swap/bubble operations}) = O(\text{height})$
- Complete $\Rightarrow$ balanced $\Rightarrow h$ is $O(\log n)$
Runtime.

- $O(\text{number of swap/bubble operations}) = O(\text{height})$
- Complete $\Rightarrow$ balanced $\Rightarrow$ $h$ is $O(\log n)$
- Maximum number of swaps is $O(\log n)$
add(e)

**Algorithm:**
- Add e in the wrong place *(the leftmost empty leaf)*
- While e is in the wrong place *(it is less than its parent)*
  - move e towards the right place *(swap with parent)*

The heap invariant is maintained!
Implementing Heaps
Implementing Heaps

```java
public class HeapNode {
    private int value;
    private HeapNode left;
    private HeapNode right;
    ...
}

public class Heap {
    HeapNode root;
    ...
}
```
Implementing Heaps

```java
public class Heap
{
    private int value;
    private Heap left;
    private Heap right;
    ...
}
```
public class Heap {  
    private int value;
    private Heap left;
    private Heap right;
    ...
}
A heap is a special binary tree.

2. **Complete:** no holes!

Full:
- 4
- 6
- 21
- 8
- 14
- 19
- 35
- 22
- 38
- 55
- 10
- 20

← as far left as possible
Numbering Nodes

Level-order traversal:
Numbering Nodes

Level-order traversal:

2. Complete: no holes!
Numbering Nodes

node $k$’s parent is $\frac{(k-1)}{2}$
node $k$’s children are nodes $2k+1$ and $2k+2$
Numbering Nodes

node $k$’s parent is $(k - 1)/2$
node $k$’s children are nodes and
Numbering Nodes

Node $k$’s parent is $(k - 1)/2$

Node $k$’s children are nodes $2k + 1$ and $2k + 2$
Implementing Heaps

public class Heap<E> {
    private E[] heap;
    private int size;
    ...
}

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
4 6 14 21 8 19 35 22 38 55 10 20
Implicit Tree Structure

2. Complete: no holes!
Heap it real, part 2.

Here's a heap, stored in an array:

\[1 5 7 6 7 10]\n
Write the array after execution of \texttt{add(4)}. Assume the array is large enough to store the additional element.

A. \[1 5 7 6 7 10 4]\nB. \[1 4 5 6 7 10 7]\nC. \[1 5 4 6 7 10 7]\nD. \[1 4 56 7 6 7 10\]