Data Structures and Algorithms
Priority Queues

Chris Brooks

Department of Computer Science
University of San Francisco
Often, it’s useful to be able to enqueue items according to *priority*.

- States to search
- Tasks to perform
- Calls to answer

We always want to be able to quickly dequeue the item with the highest (lowest) priority.
9-1: Priority Queue ADT

Operations

- Add an element with a given priority
- Return (and remove) element with highest priority

Implementation:

- Sorted Array
  - Add Element
  - Remove Highest Priority
9-2: Priority Queue ADT

Operations

- Add an element / priority pair
- Return (and remove) element with highest priority

Implementation:

- Sorted Array (this is part 4 of your project)
  - Add Element $O(n)$
  - Remove Higest Priority $O(1)$
9-3: Priority Queue ADT

Operations

- Add an element / priority pair
- Return (and remove) element with highest priority

Implementation:

- Binary Search Tree
  - Add Element
  - Remove Higest Priority
9-4: *Priority Queue ADT*

**Operations**

- Add an element / priority pair
- Return (and remove) element with highest priority

**Implementation:**

- Binary Search Tree
  - Add Element \( O(\lg n) \)
  - Remove Highest Priority \( O(\lg n) \)

*If the tree is balanced*
9-5: Priority Queue ADT

Operations

- Add an element / priority pair
- Return (and remove) element with highest priority

Implementation:

- Binary Search Tree
  - Add Element \( O(n) \)
  - Remove Highest Priority \( O(n) \)

In the worst case - tree is unbalanced.
9-6: Priority Queue ADT

Operations

- Add an element / priority pair
- Return (and remove) element with highest priority

Implementation:

- Heap
  - Add Element
  - Remove Highest Priority
Operations

- Add an element / priority pair
- Return (and remove) element with highest priority

Implementation:

- Heap
  - Add Element \(O(\log n)\)
  - Remove Highest Priority \(O(\log n)\)

So what the heck is a heap?
9-8: Heap Definition

- Complete Binary Tree (level \( n \) is filled in before level \( n + 1 \) starts)

- Heap Property
  - For every subtree in a tree, each value in the subtree is \( \leq \) value stored at the root of the subtree
9-9: Heap Examples

Valid Heap
9-10: Heap Examples

Invalid Heap
There is only one place we can insert an element into a heap, so that the heap remains a complete binary tree.

Inserting an element at the “end” of the heap might break the heap property.
There is only one place we can insert an element into a heap, so that the heap remains a complete binary tree.

Inserting an element at the “end” of the heap might break the heap property:
- Swap the inserted value with its parent
- Keep doing this while child is greater than parent.

How many swaps will we need to do?
Removing the Root of the heap is hard
Removing the element at the “end” of the heap is easy
Removing the Root of the heap is hard

Removing the element at the “end” of the heap is easy
  - Swap the last element with the root
  - May break the heap property
9-15: Heap Remove Largest

- Removing the Root of the heap is hard
- Removing the element at the “end” of the heap is easy
  - Move last element into root
  - Swap the root with the largest child, until heap property is satisfied
We can represent heaps using pointers, much like BSTs
  △ Need to add parent pointers for insert to work correctly
  △ Space needed to store pointers – 3 per node – could be greater than the space need to store the data in the heap!
  △ Memory allocation and deallocation is slow

There is a better way!
A Complete Binary Tree can be stored in an array:
The root is stored at index 0

For the node stored at index $i$:

- Left child is stored at index $2 \cdot i + 1$
- Right child is stored at index $2 \cdot i + 2$
- Parent is stored at index $\left\lfloor \frac{(i - 1)}{2} \right\rfloor$
Finding the parent of a node
int parent(int n) {
    return (n - 1) / 2;
}

Finding the left child of a node
int leftchild(int n) {
    return 2 * n + 1;
}

Finding the right child of a node
int rightchild(int n) {
    return 2 * n + 2;
}
9-20: Building a Heap

Build a heap out of $n$ elements
Build a heap out of \( n \) elements

- Start with an empty heap
- Do \( n \) insertions into the heap

```java
MaxHeap H = new MaxHeap();
for (i=0 < i<A.size(); i++)
    H.insert(A[i]);
```

Running time?
Build a heap out of $n$ elements

- Start with an empty heap
- Do $n$ insertions into the heap

MaxHeap $H = \text{new MaxHeap}();$
for ($i=0 < i<A\.size(); i++)
  H.insert(A[i]);

Running time? $O(n \lg n)$ – is this bound tight?
9-23: Building a Heap

Total time: \( c_1 + \sum_{i=1}^{n} c_2 \log i \)

\[
c_1 + \sum_{i=1}^{n} c_2 \log i \geq \sum_{i=n/2}^{n} c_2 \log i
\]

\[
\geq \sum_{i=n/2}^{n} c_2 \log(n/2)
\]

\[
= (n/2) c_2 \log(n/2)
\]

\[
= (n/2) c_2 ((\log n) - 1)
\]

\( \in \Omega(n \log n) \)

Running Time: \( \Theta(n \log n) \)
Build a heap from the bottom up

- Place elements into a heap array
- Each leaf is a legal heap
- First potential problem is at location $\lfloor i/2 \rfloor$
9-25: Building a Heap

Build a heap from the bottom up

- Place elements into a heap array
- Each leaf is a legal heap
- First potential problem is at location \([i/2]\)

```c
for(i=n/2; i>=0; i--)
    siftdown(A, i);
```
public void siftdown(int[] A, int index) {
    int l1 = left(index);
    int r1 = right(index);
    int largest;

    if (l1 < size() && A[l1] > A[index]) {
        largest = l1;
    } else {
        largest = index;
    }
    if (r1 < size() && A[r1] > A[index]) {
        largest = r1;
    } else if (largest != index)
        swap(A[index], A[largest])
    siftdown(A, largest);
}
How many swaps, worst case? If every siftdown has to swap all the way to a leaf:

- $n/4$ elements 1 swap
- $n/8$ elements 2 swaps
- $n/16$ elements 3 swaps
- $n/32$ elements 4 swaps

... 

Total # of swaps:

$$n/4 + 2n/8 + 3n/16 + 4n/32 + \ldots + (\lg n)n/n$$