Priority Queues
What is a Priority Queue?

1) Stores prioritized key-value pairs
2) Implements insertion
   • No notion of storing at particular position
3) Returns elements in priority order
   • Order determined by $key$
What’s so different?

Stacks and Queues
• Removal order determined by order of inserting

Sequences
• User chooses exact placement when inserting and explicitly chooses removal order

Priority Queue
• Order determined by key
• Key may be part of element data or separate
Order of returned elements is not FIFO or LIFO (as in queue or stack)

Random access not necessary (as in sequence) or desirable

Examples

• Plane landings managed by air traffic control
• Processes scheduled by CPU
• College admissions process for students

—What are some of the criteria?
College Admissions Key

Student submits:

- Personal data (geography, is parent alum?, activities?)
- Transcript
- Essays
- Standardized test scores
- Recommendations

Admissions agent:

- Each datum converted to number
- Formula converts to single numeric key
Student selection process

Simple scheme

• Collect applications until due date
• Sort by keys
• Take top $k$ students

More realistic

• Prioritize applications as they come in
• Accept some top students ASAP
• Maybe even change data/key as you go
An **entry** in a priority queue is simply a key-value pair.

Priority queues store entries to allow for efficient insertion and removal based on keys.

Methods:

- **getkey()**: returns the key for this entry
- **getvalue()**: returns the value associated with this entry

As a Java interface:

```java
/**
 * Interface for a key-value pair entry
 */

public interface Entry <K,V> {
    public K getkey();
    public V getvalue();
}
```
public interface PriorityQueue<K extends Comparable<? super AnyType>, V>
{
    public int size();
    public boolean isEmpty();
    public Entry<K, V> min() throws EmptyPriorityQueueException;
    public Entry<K, V> insert(K key, V value) throws InvalidKeyException;
    public Entry<K, V> removeMin() throws EmptyPriorityQueueException;
}

Implementing PQ with Unsorted Sequence

Each call to insertItem(k, e) uses insertLast() to store in Sequence

- $O(1)$ time

Each call to extractMin( ) traverses the entire sequence to find the minimum, then removes element

- $O(n)$ time
Implementing PQ with Sorted Sequence

Each call to insertItem(k, e) traverses sorted sequence to find correct position, then does insert

- $O(n)$ worst case

Each call to extractMin( ) does removeFirst( )

- $O(1)$ time
Implementing PQ with a BST

Each call to insertItem(k, e) does tree insert

- $O(\log(n))$ worst case

Each call to extractMin() does delete()

- $O(\log(n))$ time
A heap is a binary tree storing keys at its nodes and satisfying the following properties:

- **Heap-Order**: for every internal node \( v \) other than the root, \( \text{key}(v) \geq \text{key}(\text{parent}(v)) \)
- **Complete Binary Tree**: let \( h \) be the height of the heap
  - for \( i = 0, \ldots, h - 1 \), there are \( 2^i \) nodes of depth \( i \)
  - at depth \( h - 1 \), the internal nodes are to the left of the external nodes

The last node of a heap is the rightmost node of depth \( h \)
Height of a Heap

**Theorem:** A heap storing $n$ keys has height $O(\log n)$

**Proof:** (we apply the complete binary tree property)

- Let $h$ be the height of a heap storing $n$ keys
- Since there are $2^i$ keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth $h$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$
Heap

Binary tree-based data structure

- *Complete* in the sense that it fills up levels as completely as possible
- Height of tree is $O(\log n)$

Can be stored using the array representation (just add at the end of the array)

Use extendable arrays to expand and shrink as needed
Heap Example
PQ Quiz Show!

Heap, or Not A Heap?

(no paper necessary)
Heap, or Not a Heap?

```
a
g   h   i

p   n   q   w

f k e

c
```
Heap, or Not a Heap?
Heap, or Not a Heap?

```
1
3
20 32
5
6
8
```
Inserting into Heap

Create new node as “last” element

Insert key/element into new node

Bubble node upward until heap property is satisfied

while (!isRoot(node) &&
    (node.key < node.parent.key))
    swap(node, parent)

(can we do even better?)
Heap Insert Example
Bubble Upward

10

15

19

17

7

20

30

22

21

35

31

32

25

21

22

20
Bubble Upward
Bubble Upward

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
Heap Insert Analysis

New node always inserted at lowest level

Node bubbles upward

• up to root in worst case

• path length to root is $O(\log n)$

Total time for insert is $O(\log n)$
Extracting from Heap

Copy element from root node

Copy element/key from last node to root node

Delete last node

Bubble root node downward until heap property satisfied

```java
while (!isExternal(node) &&
    (node.key > node.smallestChild.key))
    swap(node, node.smallestChild)
```
Heap Extract Example
Move last node to root

7

25

15

19

17

20

10

31

32

22

21

35

30

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
Bubble Downward
All done
Heap Extract Analysis

Again, each swap takes constant time

Maximum swaps is path length from root to leaf

→ Total work is $\log n \times O(1) = O(\log n)$
Sort Analysis

foreach element, $E_i$, in $S$

```plaintext
PQ.insert($E_i$)
```

while !PQ.empty()

```plaintext
PQ.extractMin()
```

$\mathcal{O}(n)$

Heap Sort

$\mathcal{O}(n)$

$\sum_{i=0}^{n-1} \mathcal{O}(\log i)$

$\mathcal{O}(n)$

$\sum_{i=0}^{n-1} \mathcal{O}(\log i)$

$\mathcal{O}(n) + 2 \sum_{i=0}^{n-1} \mathcal{O}(\log i) < \mathcal{O}(n) + 2 \sum_{i=0}^{n-1} \mathcal{O}(\log n)$

$= \mathcal{O}(n) + 2n*\mathcal{O}(\log n) = \mathcal{O}(n\log n)$

(showing $\theta(n\log n)$ is a bit harder)
Merging Two Heaps

- We are given two heaps and a key \( k \)
- We create a new heap with the root node storing \( k \) and with the two heaps as subtrees
- We perform downheap to restore the heap-order property
Bottom-up Heap Construction

- We can construct a heap storing $n$ given keys in using a bottom-up construction with $\log n$ phases.
- In phase $i$, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys.
Example

© 2004 Goodrich, Tamassia
Example (contd.)
Example (contd.)
Example (end)
Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path).
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$.
- Thus, bottom-up heap construction runs in $O(n)$ time.
- Bottom-up heap construction is faster than $n$ successive insertions and speeds up the first phase of heap-sort.
In-class Exercise

What does the heap look like after the following sequence of insertions:

5   30   2   15   7   45   20   6   18
Just like binary trees are not always optimal, binary heaps are not

D-Heaps are heaps that have D children at each node

Speeds insert/delete

Does mean more computation per node ..
Text
- Read Weiss, §6.8

Binomial Queue
- Definition of binomial queue
- Definition of binary addition

Building a Binomial Queue
- Sequence of inserts
- What in the world does binary addition have in common with binomial queues?
A binary heap provides $O(\log n)$ inserts and $O(\log n)$ deletes but suffers from $O(\log n)$ merges.

A binomial queue offers $O(\log n)$ inserts and $O(\log n)$ deletes and $O(\log n)$ merges.

Note, however, binomial queue inserts and deletes are more expensive than binary heap inserts and deletes worst case (but constant time in average case).
A Binomial Queue is a collection of heap-ordered trees known as a forest. Each tree is a binomial tree. A recursive definition is:

1. A binomial tree of height 0 is a one-node tree.
2. A binomial tree, $B_k$, of height $k$ is formed by attaching a binomial tree $B_{k-1}$ to the root of another binomial tree $B_{k-1}$.
Examples

- $B_0$: 4
- $B_1$: 4, 8
- $B_2$: 4, 8, 5, 12
- $B_3$: 4, 8, 5, 7, 12, 8, 10, 11
- $B_4$: 4, 8, 5, 7, 12, 8, 10, 9, 15, 24, 10, 22
Questions

1. How many nodes does the binomial tree $B_k$ have?

2. How many children does the root of $B_k$ have?

3. What types of binomial trees are the children of the root of $B_k$?

4. Is there a binomial queue with one node? With two nodes? With three nodes? … With $n$ nodes for any positive integer $n$?
Consider binary numbers

Positional notation:

\[
\begin{array}{cccccccc}
2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & + & 16 & + & 0 & + & 4 & + & 2 & + & 0 & = & 22 \\
\Rightarrow & 010110_2 & = & 22_{10}
\end{array}
\]

What is the decimal value of these binary numbers?

- 011 =
- 101 =
- 10110 =
- 1001011 =
Binary Numbers

• Consider binary numbers
• Positional notation:
  - $2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$
  - $0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0$
  - $0 + 16 + 0 + 4 + 2 + 0 = 22$
  - $010110_2 = 22_{10}$

• What is the decimal value of these binary numbers?
  - 011 = 3
  - 101 = 5
  - 10110 = 22
  - 1001011 = 75
Binary Addition

(carry) 1 1

1 0 1 1 0 1 0
+ 0 0 1 1 1 0 0

-----------------

1 1 1 0 1 1 0
Consider two binomial queues, $H_1$ and $H_2$.
Merging Binomial Queues

Merge two $B_0$ trees forming new $B_1$ tree

$H_1$

$H_2$
Merging Binomial Queues

Merge two $B_1$ trees forming new $B_2$ tree

$H_1$

$H_2$
Merging Binomial Queues

Merge two $B_2$ trees forming new $B_3$ tree (but which two $B_2$ trees?)

$H_1$

$H_2$

Diagram of Binary Trees
Which two $B_2$ trees? Arbitrary decision: merge two original $B_2$ trees

$H_1$

$H_2$
Merging Binomial Queues

Which two $B_2$ trees? Arbitrary decision: merge two original $B_2$ trees
Which root becomes root of merged tree?
Arbitrary decision: in case of a tie, make the root of $H_1$ be the root of the merged tree.

$H_1$

$H_2$
Merging Binomial Queues

Merge two $B_2$ trees forming new $B_3$ tree

$H_1$

$H_2$
Merging Binomial Queues

Merge two $B_3$ trees forming a new $B_4$ tree

$H_1$

$H_2$
Merging Binomial Queues

Call new binomial queue $H_3$
Reconsider the two original binomial queues, $H_1$ and $H_2$ and identify types of trees.
Represent each binomial queue by a binary number

**H₁**

<table>
<thead>
<tr>
<th>B₄</th>
<th>B₃</th>
<th>B₂</th>
<th>B₁</th>
<th>B₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**H₂**

<table>
<thead>
<tr>
<th>B₄</th>
<th>B₃</th>
<th>B₂</th>
<th>B₁</th>
<th>B₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**B₃**

- 4
- 8
- 5
- 7
- 12
- 10
- 11

**B₂**

- 3
- 6
- 2
- 15
- 13
- 27

**B₁**

- 2
- 16

**B₀**

- 1
- 6
Merging Binomial Queues

Note that the merged binomial queue can be represented by the binary sum:

\[
\begin{align*}
H_1 &= B_4 \ B_3 \ B_2 \ B_1 \ B_0 \\
     &= 0 \ 1 \ 1 \ 0 \ 1 \\
H_2 &= B_4 \ B_3 \ B_2 \ B_1 \ B_0 \\
     &= 0 \ 0 \ 1 \ 1 \ 1 \\
H_3 &= B_4 \ B_3 \ B_2 \ B_1 \ B_0 \\
     &= 1 \ 0 \ 1 \ 0 \ 0
\end{align*}
\]
This suggests a way to implement binomial queues:

1. Use a *k-ary* tree to represent each binomial tree – sibling and child pointers

2. Use a Vector to hold references to the root node of each binomial tree

3. Keep a reference to smallest root for past find min (e.g. a Heap on positions).
Implementing Binomial Queues

Use a k-ary tree to represent each binomial tree. Use an array to hold references to root nodes of each binomial tree.
Questions

We now know how to merge two binomial queues. How do you perform an insert?

How do you perform a delete?

What is the order of complexity of a merge? an insert? a delete?
Carefully study Java code in Weiss, Figure 6.52 – 6.56
PQ summary

What is a PQ (what ops)
What are heap properties
How do heaps work
Binomial queues as a second example