CSE 373

OCTOBER 30TH – PRIORITY QUEUES

ADMINISTRIVIA

- **Practice exam out by tomorrow**
- **P1 EC graded by tonight**
- **P2 graded on Wednesday**
- **HW regrades tonight**
  - Please fill out the form if you have regrade questions.

MIDTERM EXAM

- Friday, November 3rd, 2:30-3:20
- No note sheets or calculators
- Exam review in class on Wednesday
- Covers everything through the end of today's lecture

PRIORITY QUEUE

- **New ADT**
- **Objects in the priority queue have:**
  - Data
  - Priority
- **Conditions**
  - Lower priority items should dequeue first
  - Should be able to change priority of an item
  - FIFO for equal priority?

PRIORITY QUEUE

- **insert(K key, int priority)**
  - Insert the key into the PQ with given priority
- **findMin()**
  - Return the key that currently has lowest priority in the PQ (min-heap)
- **deleteMin()**
  - Return and remove the key with lowest priority
- **changePriority(K key, int newPri)**
  - Assign a new priority to the object key

PRIORITY QUEUE

- **How to implement?**
  - Keep data sorted (somehow)
- **Array?**
  - Inserting into the middle is costly
    (must move other items)
- **Linked list?**
  - Must iterate through entire list to find place
  - Cannot move backward if priority changes

PRIORITY QUEUE

- These data structures will all give us the behavior we want as far as the ADT, but they may be poor design decisions
- Any other data structures to try?

PRIORITY QUEUE

- Want the speed of trees (but not BST)
- Priority Queue has unique demands
- Other types of trees?
- Review BST first

PROPERTIES (BST)

- **Tree (Binary)**
  - Root
  - (Two) Children
  - No cycles
- **Search**
  - Comparable data
  - Left child data < parent data
  - Smallest child is at the left most node

PROPERTIES (BST)

- **Binary tree may be useful**
- **Search property doesn't help**
  - Always deleting min
  - Put min on top!

HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children

HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children
- How to implement?
- Insert and delete are different than BST

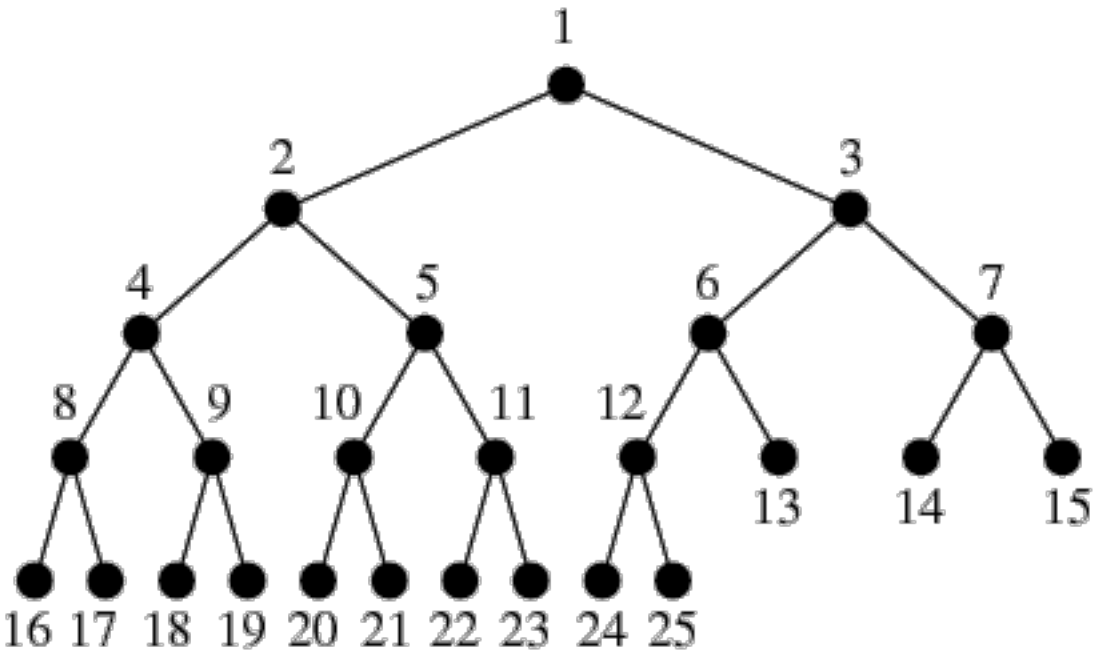
HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children
- How to implement?
- Insert and delete are different than BST

HEAPS

- The Priority Queue is the ADT
- The Heap is the Data Structure

COMPLETENESS



Filling left to right and top to bottom is another property - completeness

HEAP EXAMPLE

HEAPS

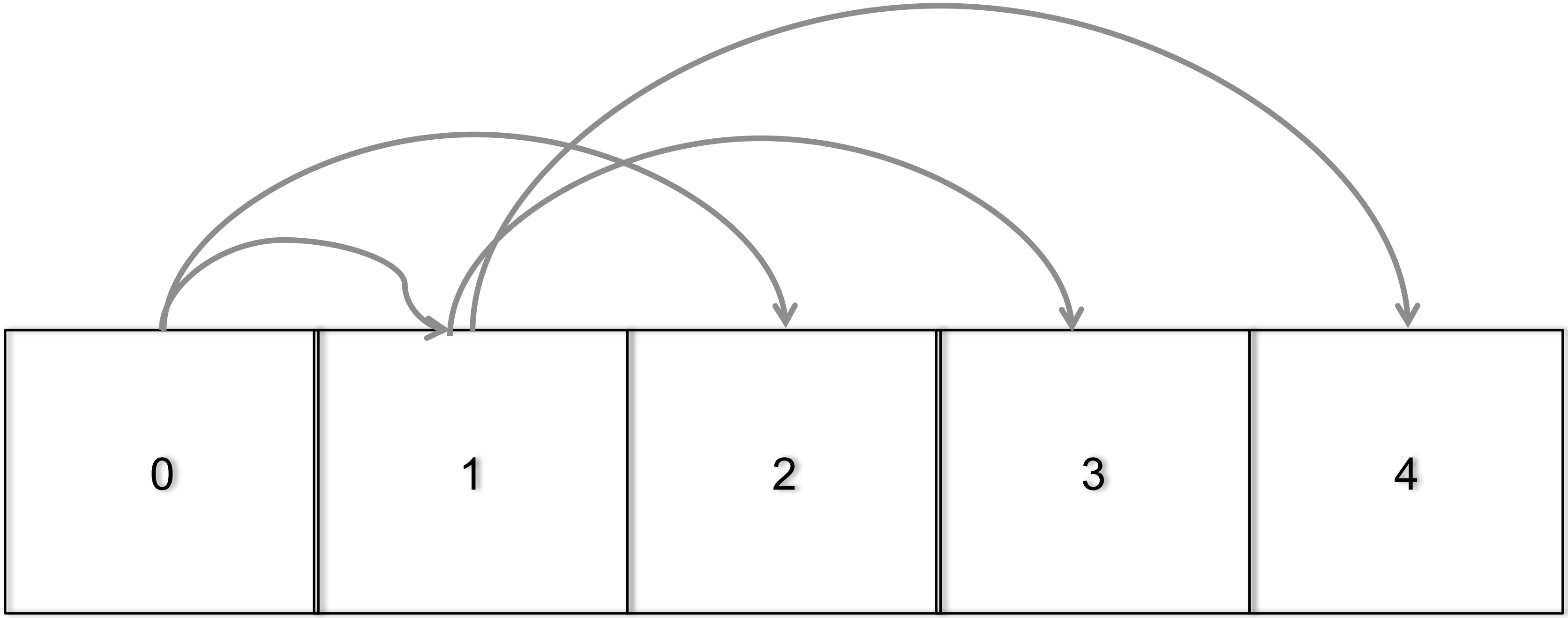
- Heap property (parents < children)
- Complete tree property (left to right, bottom to top)

HEAPS

- **Heap property (parents < children)**
- **Complete tree property (left to right, bottom to top)**
- **How does this help?**
  - Array implementation

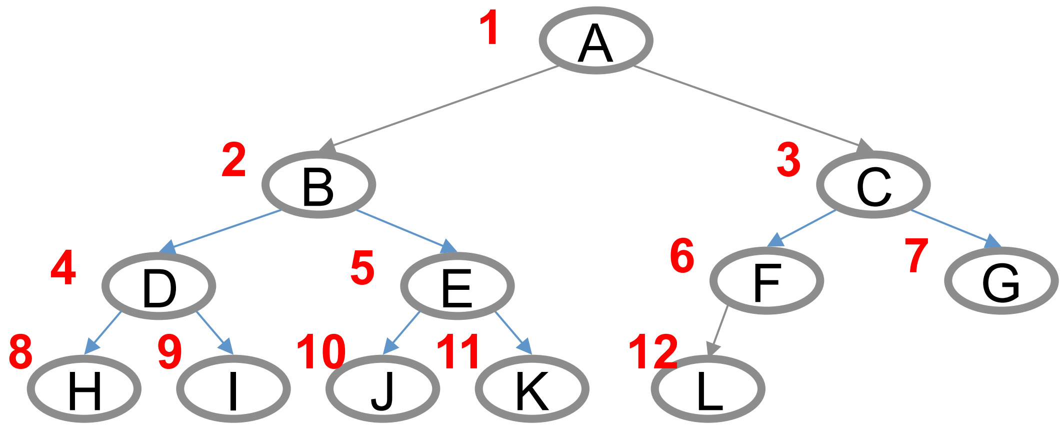
HEAPS

- Insert into array from left to right
- For any parent at index i,
  children at 2*i+1 and 2*i+2



REVIEW

- **Array property**



	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

HEAPS

- **How to maintain heap property then?**
  - Parent must be higher priority than children
- **Two functions – percolate up and percolate down**

HEAP FUNCTIONS

- **Percolate up**
  - When a new item is inserted:
    - Place the item at the next position to preserve completeness
    - Swap the item up the tree until it is larger than its parent

HEAP FUNCTIONS

- **Percolate down**
  - When an item is deleted:
    - Remove the root of the tree (to be returned)
    - Move the last object in the tree to the root
    - Swap the moved piece down while it is larger than it's smallest child
    - Only swap with the smallest child

HEAPS AS ARRAYS

- **Because heaps are complete, they can be represented as arrays without any gaps in them.**
- **Naïve implementation:**
  - Left child: 2*i+1
  - Right child: 2*i + 2
  - Parent: (i-1)/2

HEAPS AS ARRAYS

- **Alternate (common) implementation:**
  - Put the root of the array at index 1
  - Leave index 0 blank
  - Calculating children/parent becomes:
    - Left child: 2*i
    - Right child: 2*i + 1
    - Parent: i/2

HEAPS AS ARRAYS

- **Why do an array at all?**
  - + Memory efficiency
  - + Fast accesses to data
  - + Forces log n depth
  - - Needs to resize
  - - Can waste space
- **Almost always done through an array**

ANALYSIS

- Let's find an interesting algorithm to analyze

ANALYSIS

- Let’s find an interesting algorithm to analyze
- Given an array of length n, how do we make that array into a heap?

ANALYSIS

- **Let's find an interesting algorithm to analyze**
- **Given an array of length n, how do we make that array into a heap?**
- **Naïve approach?**
  - Make a new heap and add each element of the array into the heap

ANALYSIS

- **Let’s find an interesting algorithm to analyze**
- **Given an array of length n, how do we make that array into a heap?**
- **Naïve approach?**
  - Make a new heap and add each element of the array into the heap
  - How long to finish?

FUN FACTS!

- Is it really O(n log n)?

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

  1+2+3+4+5…

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

1+2+3+4+5…

FUN FACTS!

- **Is it really O(n log n)?**
  - Early insertions are into empty trees **O(1)**!
  - Consider a simpler example, creating a sorted linked list.
  - Adding at the end of a linked list with k items takes O(k) operations.

1+2+3+4+5…

What is this summation?

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n + 1)$$

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$$

- **What does this mean?**

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$$

- What does this mean?
- Summing k from 1 to n is still $O(n^2)$

FUN FACTS!

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n + 1)$$

- What does this mean?
- Summing k from 1 to n is still `O(n`^{2}`)`
- Similarly, summing `log(k)` from 1 to n is `O(n log n)`

ANALYSIS

- **Naïve approach:**
  - Must add n items

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long?

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(n log(n))`

ANALYSIS

- **Naïve approach:**
  - Must add n items
  - Each add takes how long? `log(n)`
  - Whole operation is `O(n log(n))`
  - Can we do better?
    - What is better? O(n)

HEAPS

- **Facts of binary trees**

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes
  - Therefore, a complete binary tree has half of its nodes in the leaves

HEAPS

- **Facts of binary trees**
  - Increasing the height by one doubles the number of possible nodes
  - Therefore, a complete binary tree has half of its nodes in the leaves
  - A new piece of data is much more likely to have to percolate down to the bottom than be the smallest item in the heap

BUILDHEAP

- So a naïve buildheap takes O(n log n)

BUILDHEAP

- **So a naïve buildheap takes O(n log n)**
  - Why implement at all?

BUILDHEAP

- **So a naïve buildheap takes O(n log n)**
  - Why implement at all?
  - If we can get it `O(n)`!

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find?

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`

FLOYD'S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`
- **Percolate down each node as necessary**

FLOYD’S METHOD

- **Traverse the tree from bottom to top**
  - Reverse order in the array
- **Start with the last node that has children.**
  - How to find? `Size / 2`
- **Percolate down each node as necessary**
  - Wait! Percolate down is O(log n)!
  - This is an O(n log n) approach!

FLOYD'S METHOD

- It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!

FLOYD'S METHOD

- It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!
- How far does each node travel (at worst)

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - Leaves don't move at all: Height = 0

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - Leaves don't move at all: Height = 0
    - This is half the nodes in the tree

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - 1/2 of the nodes don't move:
    - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two

FLOYD'S METHOD

- **It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!**
- **How far does each node travel (at worst)**
  - 1/2 of the nodes don't move:
    - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two …

FLOYD'S METHOD

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} =$$

FLOYD'S METHOD

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} = 2^{-n-1}\left(-n + 2^{n+1} - 2\right)$$

- **Thanks Wolfram Alpha!**

FLOYD'S METHOD

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} = 2^{-n-1}\left(-n + 2^{n+1} - 2\right)$$

- Thanks Wolfram Alpha!
- Does this look like an easier summation?

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- This is a must know summation!

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- **This is a must know summation!**
- **1/2 + 1/4 + 1/8 + … = 1**

FLOYD'S METHOD

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- This is a must know summation!
- 1/2 + 1/4 + 1/8 + … = 1
- How do we use this to prove our complicated summation?

FLOYD'S METHOD

$1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n \leq 1$

FLOYD'S METHOD

$$1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n \leq 1$$

$$1/4 + 1/8 \ldots \quad \ldots + 1/2^n \leq 1/2$$

$$1/8 \ldots \quad \ldots + 1/2^n \leq 1/4$$

FLOYD'S METHOD

$$
\begin{array}{l}
1/2 + 1/4 + 1/8 \ldots \quad \ldots + 1/2^n \leq 1 \\
\phantom{1/2 + {}} 1/4 + 1/8 \ldots \quad \ldots + 1/2^n \leq 1/2 \\
\phantom{1/2 + 1/4 + {}} 1/8 \ldots \quad \ldots + 1/2^n \leq 1/4
\end{array}
$$

- **Vertical columns sum to:**
  `i/2^i`, which is what we want
- **What is the right summation?**
  - Our original summation plus 1

FLOYD'S METHOD

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

FLOYD'S METHOD

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

- This means that the number of swaps we perform in Floyd's method is 2 times the size… So Floyd's method is `O(n)`