ACM Programming Contest

• Do you want a free lunch and dinner on Saturday?

• Do you want a free t-shirt?

• Are you free on Saturday from 11:15am - 6:00pm

• Volunteer to help

Queues and Stacks

• Priority queues

• Heaps
Queue vs. Stack

public static void main(String[] args) {
    Queue aQueue = new LinkedList();
    Stack aStack = new Stack();
    String[] wordsToAdd = {"compsci", "201", "is", "great"};

    for (String s: wordsToAdd){
        aQueue.add(s); //enqueue
        aStack.push(s);
    }

    while (!aQueue.isEmpty())
        System.out.print(aQueue.remove() + " "); //dequeue

    System.out.println();
    while (!aStack.isEmpty())
        System.out.print(aStack.pop() + " ");
}

Priority Queue

- What about those people in first class?
public static void main(String[] args) {

    //Queue aQueue = new LinkedList();
    PriorityQueue<String> aQueue = new PriorityQueue<String>();

    String[] wordsToAdd = {"compsci", "201", "is", "great"};

    for(String s: wordsToAdd) {
        aQueue.add(s);
        aStack.push(s);
    }

    while(!aQueue.isEmpty())
        System.out.print(aQueue.remove() + " ");
}

1. compsci 201 is great
2. great is compsci 201
3. is great compsci 201
4. 201 compsci great is

• What is the output?

PriorityQueue<Integer> ex = new PriorityQueue<Integer>();
ex.add(2);
ex.add(13);
ex.add(9);
ex.add(75);
ex.add(4);
while(!ex.isEmpty()) {
    System.out.println(ex.remove());
}

• add in any order
• remove smallest first
Code practice

- Snarf the code
- The class BestPrice tracks all the prices of an item.
- Every time an item is found the price is added to an instance variable
- buyCheapest(n) returns the total price of the n cheapest items
  - and removes them because they have now been purchased.

- Hint: make a priority queue instance variable

Heaps

- A (common) implementation of a priority queue
- A tree-like structure
- Almost completely filled
  - all nodes filled except last level
- Max-Heap - Descendants have values <= to its parent
- Min-Heap - Descendants have values >= to its parent
Min Heap

- The root is the minimum value!

- Why would a priority queue be implemented with a heap?

Heaps as Arrays

- Heaps can be implemented as arrays.
- The root is at index 0.
- Each parent node is at index i and its children are at indices 2i+1 and 2i+2.

```
  20
 /   \
75    43
 / \
84  90
 / \
96 91
```

Layer 1 | Layer 2 | Layer 3 | Layer 4
---------|---------|---------|---------
 20      | 75      | 43      | 84  90
 75      | 43      | 84      | 90  57
 84      | 90      | 57      | 71  71
 96      | 91      | 71      | 96  91
 93      |         |         | 93     
```
Add

- Add 55 to the heap
- Add node in first open slot

Heap insert

- Add 55 to the heap
- Add node in first open slot
Heap insert

- Add 55 to the heap
- If parent is larger, swap
Heap insert

- Add 55 to the heap
- If parent is larger, swap

```
public void add(double d){
    mySize++;
    myMinHeap[mySize] = d;

    int index = mySize;
    int parentIndex = index/2;
    while((myMinHeap[parentIndex] > myMinHeap[index]) & parentIndex != 0){
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = index/2;
    }
}

private void swap(int i, int j){
    double temp = myMinHeap[i];
    myMinHeap[i] = myMinHeap[j];
    myMinHeap[j] = temp;
}
```

As an array

```
  20  75  43  84  90  57  71  96  91  93  55
```

```
  1  2  3  4  5  6  7  8  9 10 11
```
As an array

public void add(double d){
    mySize++;
    myMinHeap[mySize] = d;

    int index = mySize;
    int parentIndex = index/2;
    while((myMinHeap[parentIndex] > myMinHeap[index]) & parentIndex != 0){
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = index/2;
    }
}

private void swap(int i, int j){
    double temp = myMinHeap[i];
    myMinHeap[i] = myMinHeap[j];
    myMinHeap[j] = temp;
}

20  75  43  84  90  57  71  96  91  93  55
1    2    3    4    5    6    7    8    9    10    11

As an array

public void add(double d){
    mySize++;
    myMinHeap[mySize] = d;

    int index = mySize;
    int parentIndex = index/2;
    while((myMinHeap[parentIndex] > myMinHeap[index]) & parentIndex != 0){
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = index/2;
    }
}

private void swap(int i, int j){
    double temp = myMinHeap[i];
    myMinHeap[i] = myMinHeap[j];
    myMinHeap[j] = temp;
}

20  75  43  84  55  57  71  96  91  93  90
1    2    3    4    5    6    7    8    9    10    11
As an array

```java
public void add(double d){
    mySize++;
    myMinHeap[mySize] = d;

    int index = mySize;
    int parentIndex = index/2;
    while((myMinHeap[parentIndex] > myMinHeap[index]) & parentIndex != 0){
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = index/2;
    }
}

private void swap(int i, int j){
    double temp = myMinHeap[i];
    myMinHeap[i] = myMinHeap[j];
    myMinHeap[j] = temp;
}
```

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>75</td>
<td>43</td>
<td>84</td>
<td>55</td>
<td>57</td>
<td>71</td>
<td>96</td>
<td>91</td>
<td>93</td>
<td>90</td>
</tr>
</tbody>
</table>
As an array

```java
public void add(double d){
    mySize++;
    myMinHeap[mySize] = d;

    int index = mySize;
    int parentIndex = index/2;
    while((myMinHeap[parentIndex] > myMinHeap[index]) & parentIndex != 0){
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = index/2;
    }
}

private void swap(int i, int j){
    double temp = myMinHeap[i];
    myMinHeap[i] = myMinHeap[j];
    myMinHeap[j] = temp;
}
```

![Array elements](image)

Remove

- Remove the root
- Move last value into root
- If a child is smaller than root
  - promote the smallest child

![Array elements](image)

- What would the array look like if I called `remove()`?
- And then `remove()`?
Remove

- Remove the root
- Move last value into root
- If a child is smaller than root
  - promote the smallest child

20 55 43 84 75 57 71 96 91 93 90

43 55 57 84 75 90 71 96 91 93

55 75 57 84 93 90 71 96 91

Practice

- Imagine a binary tree where each node gives a score. Your goal is to get a high score by selecting nodes. But you’re only allowed to select a node if you’ve already selected its parent. (so to get the 9 in the lower left, you must also select 2, 0, and 6 first)

- One algorithm is to always select the node that has the highest value of your current possible nodes. First you would select 2. Then you would select 5 (best of [0,5]). Then you would select 3 [best of [0,3,2]].

- Write the function greedyTreeScore that computes your overall score using this approach, given a tree and the number of nodes you are allowed to select.

- If you finish early write a function that computes the best possible score given a tree and a number of nodes you are allowed to select.