CS 61B Midterm 2 Review

Lu Cheng, Harrison Wallace, Riyaz Faizullabhoy, John Du

Eta Kappa Nu, Mu Chapter
University of California, Berkeley

November 3 2012

True/False

Determine the truth value of the following statement:

$$n^2 \in \Omega(n!)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$n^2 \in \Omega(n!)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

An **inorder** traversal of a BST will visit the nodes in ascending order.

Algorithmic Analysis

What are the bounds of the following algorithm?

```
1 public ArrayList<String> convertToArrayList(String[] toAdd) {
2     int N = toAdd.length;
3     ArrayList<String> result = new ArrayList<String>(0);
4     for (String x : toAdd) {
5         result.add(x);
6     }
7     return result;
8 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

$O(n)$, where n is the size of `toAdd`.

```
1 | public ArrayList<String> convertToArrayList(String[] toAdd) {
2 |     int N = toAdd.length;
3 |     ArrayList<String> result = new ArrayList<String>(0);
4 |     for (String x : toAdd) {
5 |         result.add(x);
6 |     }
7 |     return result;
8 | }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

```
1 public void reverse(LinkedList<String> A) {
2     int N = A.size();
3     for (int i = 0; i < N; i++) {
4         String last = A.removeLast();
5         A.addFirst(last);
6     }
7 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

$O(n)$, where n is the size of `A`.

```
1 public void reverse(LinkedList<String> A) {
2     int N = A.size();
3     for (int i = 0; i < N; i++) {
4         String last = A.removeLast();
5         A.addFirst(last);
6     }
7 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

```
1 public void reverse(ArrayList<String> A){
2     int N = A.size();
3     for(int i = 0; i < N; i++){
4         String last = A.remove(N - 1);
5         A.add(0, last);
6     }
7 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

$O(n^2)$, where n is the size of `A`.

```
1 public void reverse(ArrayList<String> A){
2     int N = A.size();
3     for(int i = 0; i < N; i++){
4         String last = A.remove(N - 1);
5         A.add(0, last);
6     }
7 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

```
1  public ArrayList<Integer> bucketNums(ArrayList<Integer> toSort){
2      int N = toSort.size();
3      HashMap<Integer, Integer> occurences
4          = new HashMap<Integer, Integer>();
5      for(int x : toSort){
6          if(!occurrences.containsKey(x)){
7              occurrences.put(x, 1);
8          } else {
9              occurrences.put(x, occurrences.get(x) + 1;
10         }
11     }
12     ArrayList<Integer> sorted = new ArrayList<Integer>();
13     for(int key : occurrences.keySet()){
14         for (int i = 0; i < occurences.get(key); i++){
15             sorted.add(key);
16         }
17     }
18     return sorted;
19 }
```

Algorithmic Analysis

What are the bounds of the following algorithm?

$O(n)$, where n is the size of `toSort`.

```
1  public ArrayList<Integer> bucketNums(ArrayList<Integer> toSort){
2      int N = toSort.size();
3      HashMap<Integer, Integer> occurences
4          = new HashMap<Integer, Integer>();
5      for(int x : toSort){
6          if(!occurrences.containsKey(x)){
7              occurrences.put(x, 1);
8          } else {
9              occurrences.put(x, occurrences.get(x) + 1;
10         }
11     }
12     ArrayList<Integer> sorted = new ArrayList<Integer>();
13     for(int key : occurrences.keySet()){
14         for (int i = 0; i < occurences.get(key); i++){
15             sorted.add(key);
16         }
17     }
18     return sorted;
19 }
```

Binary Search Trees: Review

A binary search tree is a tree with the following properties:

- The tree is a binary tree (each node has at most two children).
- The left subtree of any node contains only keys less than that node's key
- The right subtree of any node contains only keys greater than that node's key

Binary Search Trees: Review

A binary search tree is a tree with the following properties:

- The tree is a binary tree (each node has at most two children).
- The left subtree of any node contains only keys less than that node's key
- The right subtree of any node contains only keys greater than that node's key

To `find` an element e:

1. Start at the root. If the tree is empty, then the key is not in the tree.
2. If the root's key is e, then return the value. Otherwise:
   1. If the root's key is greater than e, then run `find()` on its left child.
   2. Otherwise, run `find()` on its right child.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

To `remove()` an element:

1. Search for the item using `find()`.
   1. If it has 0 children, remove the node from the tree.
   2. If it has 1 child, replace the node with its child.
   3. If it has 2 children, replace the label of the node with the label of its in-order successor and remove that node.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

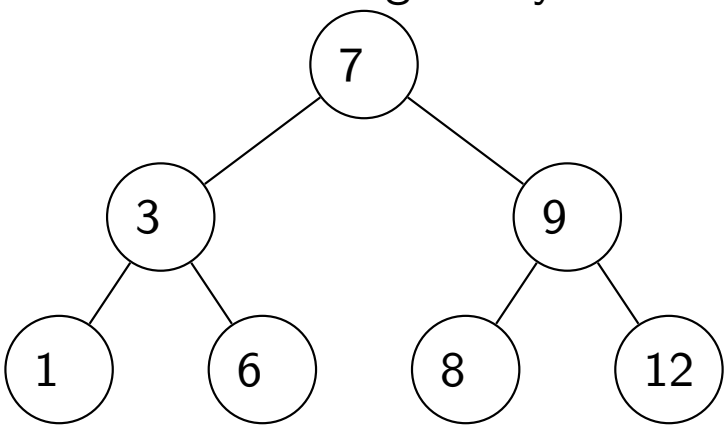
To `remove()` an element:

1. Search for the item using `find()`.
    1. If it has 0 children, remove the node from the tree.
    2. If it has 1 child, replace the node with its child.
    3. If it has 2 children, replace the label of the node with the label of its in-order successor and remove that node.

The **in-order successor** of a node is the node that is visited after the first node in an in-order traversal of the tree. In a binary search tree, the label of the in-order successor is the smallest value that is greater than the node's label. The in-order successor of a node is the bottom leftmost child in its right subtree.

Binary Search Trees

Given the following binary search tree:



Draw what it looks like after each of the following consecutive method calls:

- `insert(5)`
- `remove(3)`
- `insert(3)`
- `remove(9)`

Binary Search Trees

After `insert(5)`:

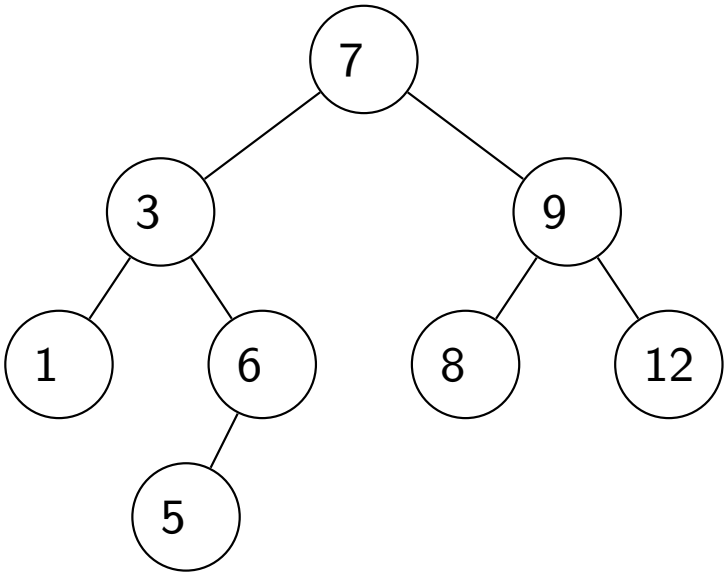
After `insert(3)`:

After `remove(9)`:

After `remove(3)`:

Binary Search Trees

After `insert(5)`:



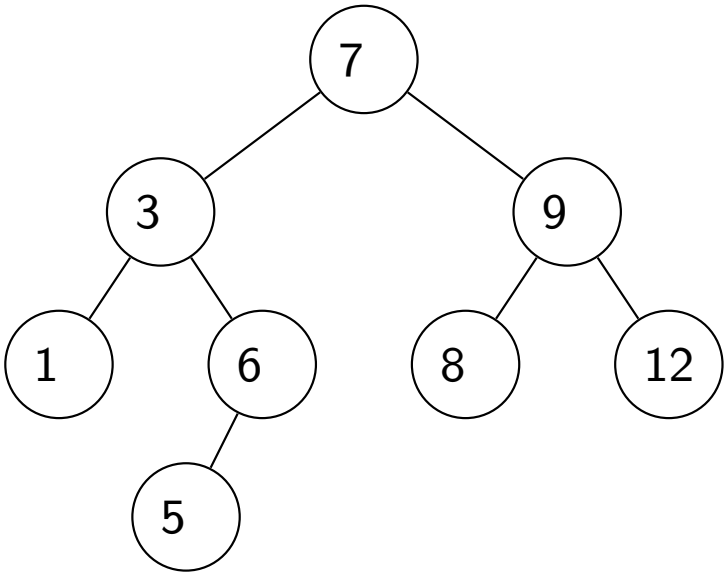
After `insert(3)`:

After `remove(3)`:

After `remove(9)`:

Binary Search Trees

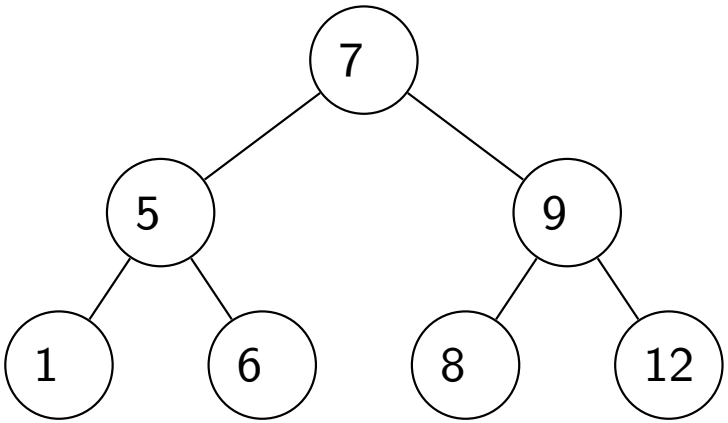
After `insert(5)`:



After `insert(3)`:

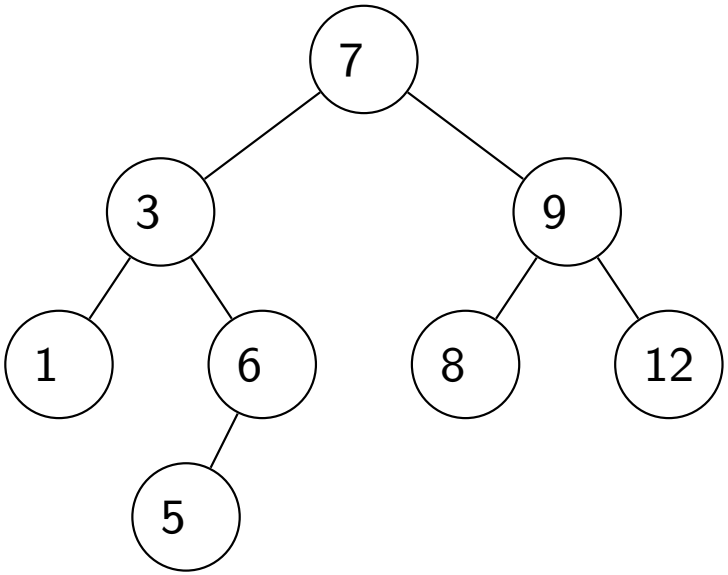
After `remove(9)`:

After `remove(3)`:

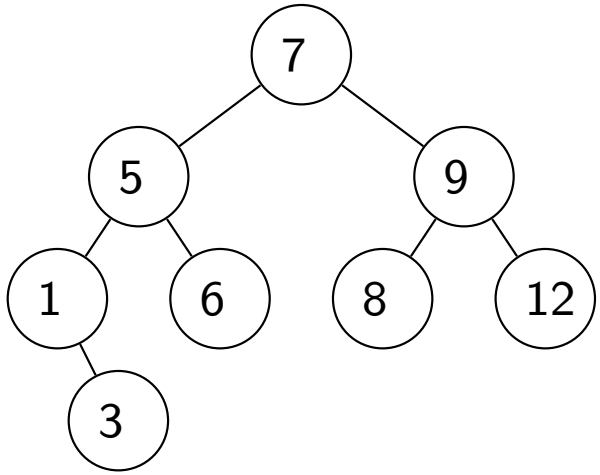


Binary Search Trees

After `insert(5)`:

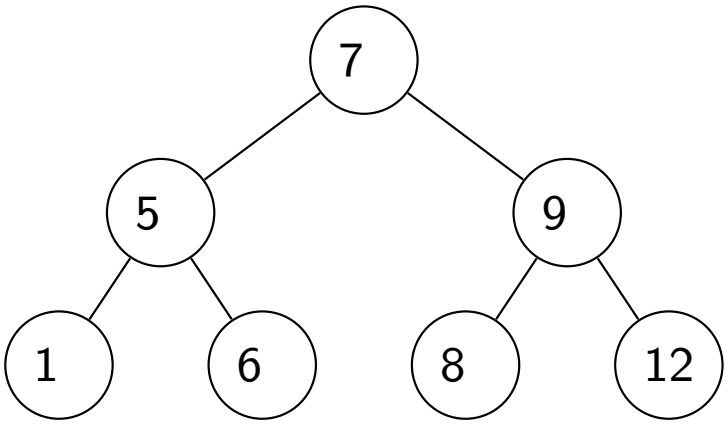


After `insert(3)`:



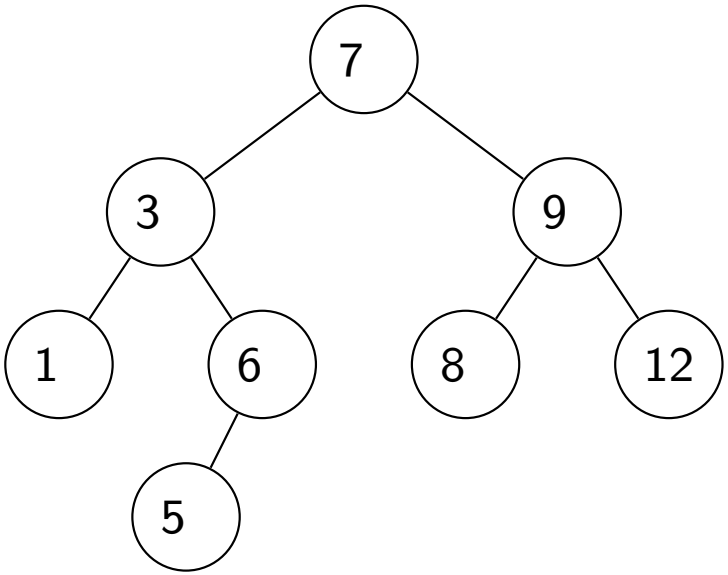
After `remove(9)`:

After `remove(3)`:

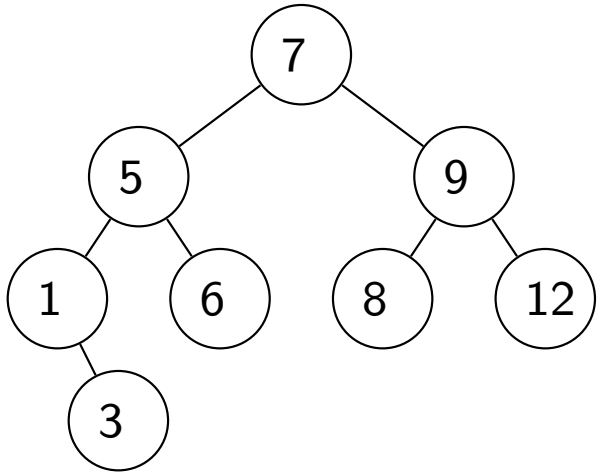


Binary Search Trees

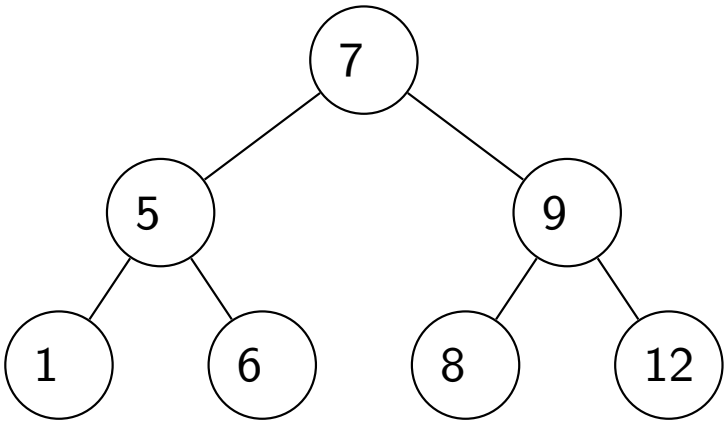
After `insert(5)`:



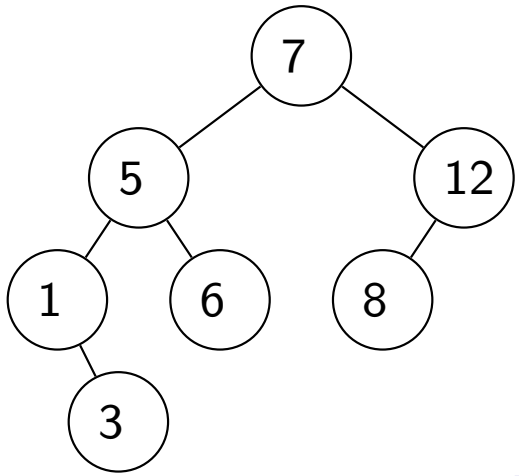
After `insert(3)`:



After `remove(3)`:



After `remove(9)`:



Binary Search Trees

Given:

```
1 public class Node{
2     Node left, right, parent;
3     int value;
4 }
5
6 public boolean isLeftChild(Node N){
7     //Fill out here
8 }
```

Write a method `isLeftChild` that determines if the `Node N` is the left child of its parent.

Binary Search Trees

Solution:

```
1  public class Node{
2      Node left, right, parent;
3      int value;
4  }
5
6  public boolean isLeftChild(Node N){
7      return (N.parent != null) && (N.parent.left == N);
8  }
```

Binary Search Trees

Given:

```
1  public class Node{
2      Node left, right, parent;
3      int value;
4  }
5
6  public boolean isParent(Node N){
7      //Fill out here
8  }
```

Write a method `isParent` that determines if the `Node N` is a parent.

Binary Search Trees

Solution:

```
1  public class Node{
2      Node left, right, parent;
3      int value;
4  }
5
6  public boolean isParent(Node N){
7      return (N.left != null) || (N.right != null);
8  }
```

Binary Search Trees

Given:

```
1  public class Node{
2      Node left, right, parent;
3      int value;
4  }
5
6  public boolean isMinOfTree(Node N){
7      //Fill out here. Hint: you may need a helper method.
8  }
```

Write a method `isMinOfTree` that determines if the `Node N` is the minimum element of the binary search tree that contains it.

Binary Search Trees

Solution:

```
1  public class Node{
2      Node left, right, parent;
3      int value;
4  }
5
6  public boolean isMinOfTree(Node N){
7      return N.left == null && isLeftChild(N)
8          && isMinOfTreeHelper(N.parent);
9  }
10
11 public boolean isMinOfTreeHelper(Node N){
12     return (N == null)
13         || (isLeftChild(N) && isMinOfTreeHelper(N.parent));
14 }
```

Heaps: Review

A heap is a binary tree with the following properties:

- The tree is complete. That is, every level is filled except possibly the last, which is filled from left to right.
- The *heap property* or *heap invariant* holds for all nodes of the tree: If B is a descendant of A, then the key of B is greater than or equal to that of A (for a min heap).

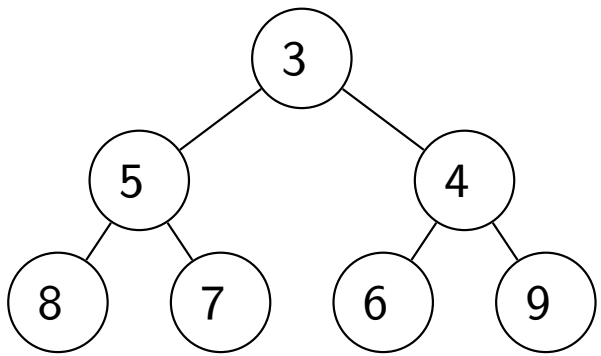
Heaps: Review

A heap is a binary tree with the following properties:

- The tree is complete. That is, every level is filled except possibly the last, which is filled from left to right.
- The *heap property* or *heap invariant* holds for all nodes of the tree: If B is a descendant of A, then the key of B is greater than or equal to that of A (for a min heap).

Heaps are usually implemented as arrays:

×	3	5	4	8	7	6	9



Heaps: Review

To `insert()` an element:

1. Insert the item at the end of the array.
2. Bubble up by repeatedly swapping with parents until the heap property is satisfied.

Heaps: Review

To `insert()` an element:

1. Insert the item at the end of the array.
2. Bubble up by repeatedly swapping with parents until the heap property is satisfied.

To `removeMin()`:

1. Swap the first and last elements of the array.
2. Remove the last element and return it.
3. Bubble the root down by repeatedly comparing with both of its children and swapping until the heap property is satisfied.

Heaps: Review

Running Times			
	Binary Heap	Sorted List/Array	Unsorted List/Array
min()	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
insert() (worst case)	$\Theta(logn)^*$	$\Theta(n)$	$\Theta(1)^*$
insert() (best case)	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$
removeMin() (worst case)	$\Theta(logn)$	$\Theta(1)$	$\Theta(n)$
removeMin() (best case)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

* If you are using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take $\Theta(n)$ time to allocate a larger array and copy the entries into it. However, if you double the array size each time, the average running time will still be as indicated.

Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

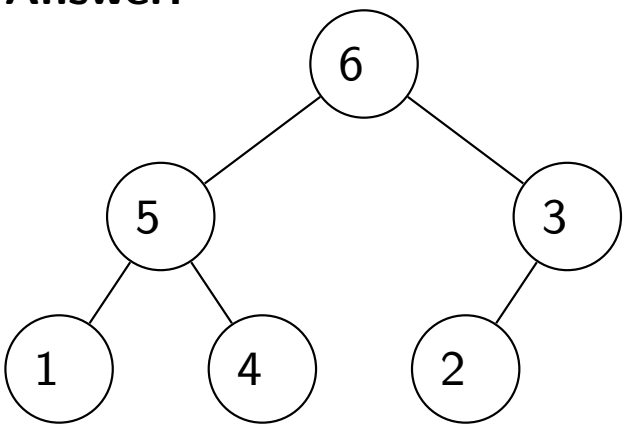
Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

Answer:



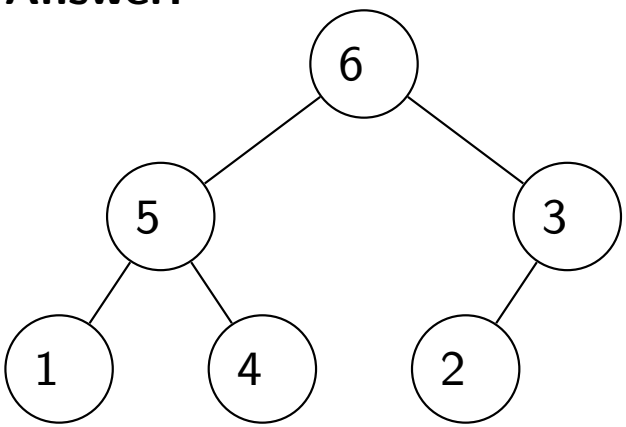
Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

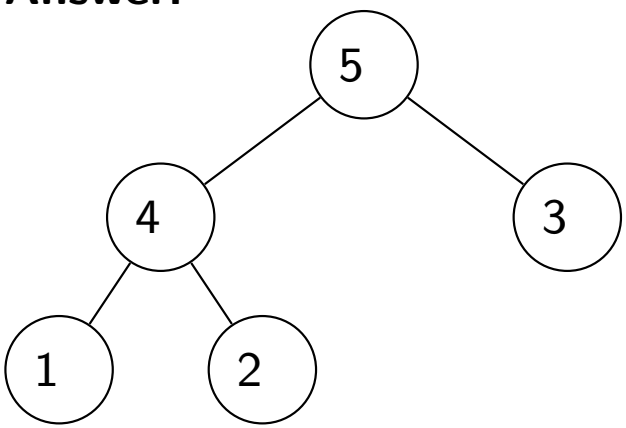
Answer:



What does the heap look like after calling `removeMax()`?

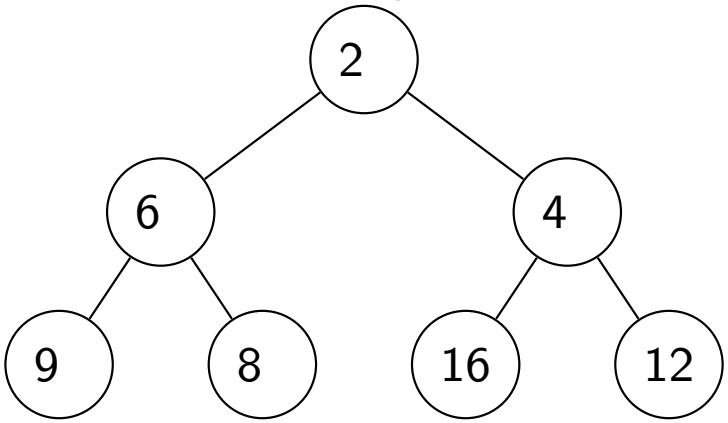
Heaps

Answer:



Heaps

Given the following **min heap**:



Draw what it looks like after each of the following consecutive method calls:

- `insert(10)`
- `removeMin()`
- `insert(3)`
- `removeMin()`

Heaps

After `insert(10)`:

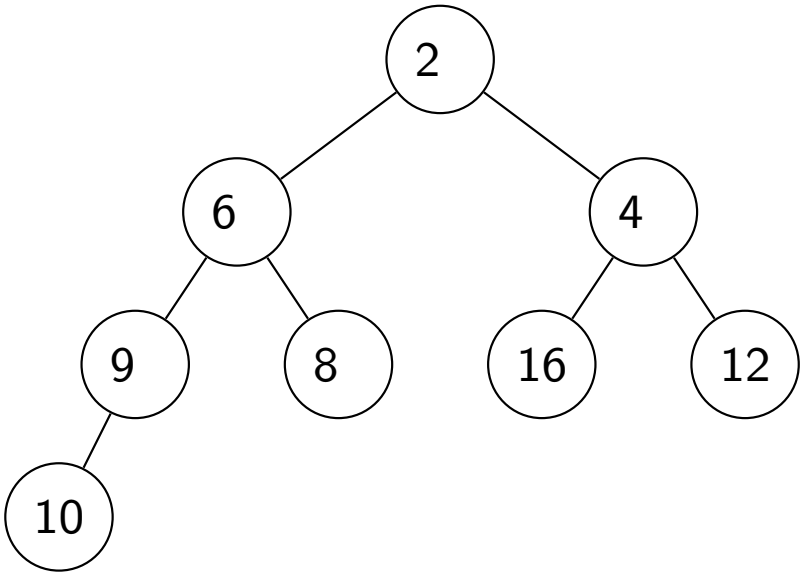
After `insert(3)`:

After `removeMin()`:

After `removeMin()`:

Heaps

After `insert(10)`:



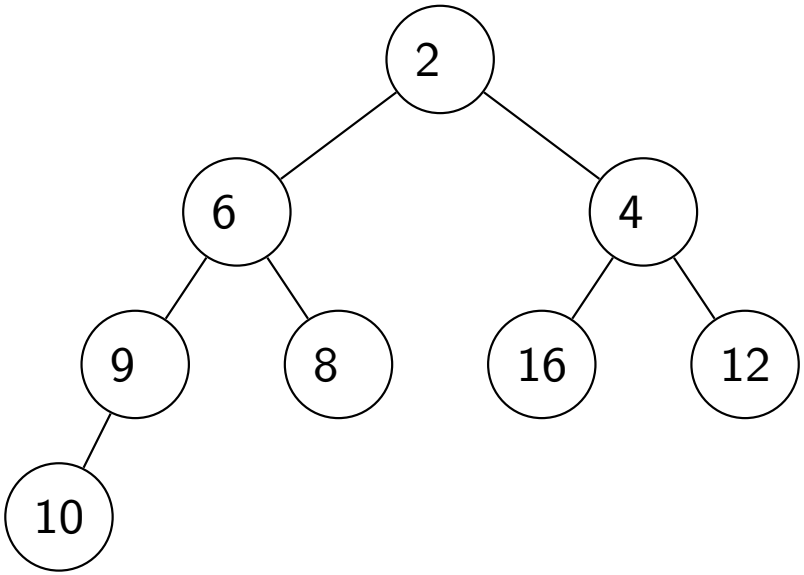
After `insert(3)`:

After `removeMin()`:

After `removeMin()`:

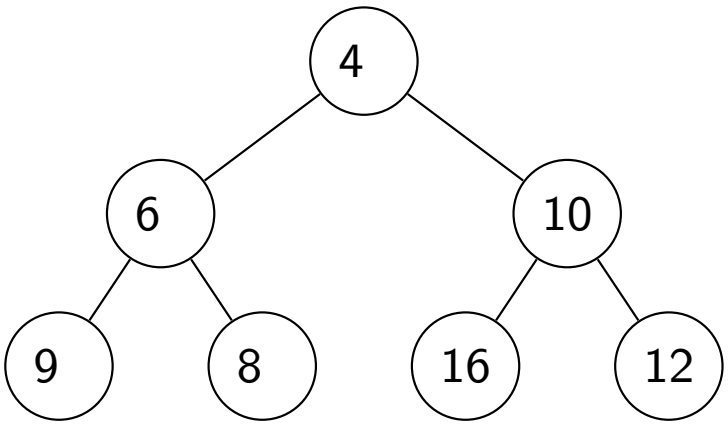
Heaps

After `insert(10)`:



After `insert(3)`:

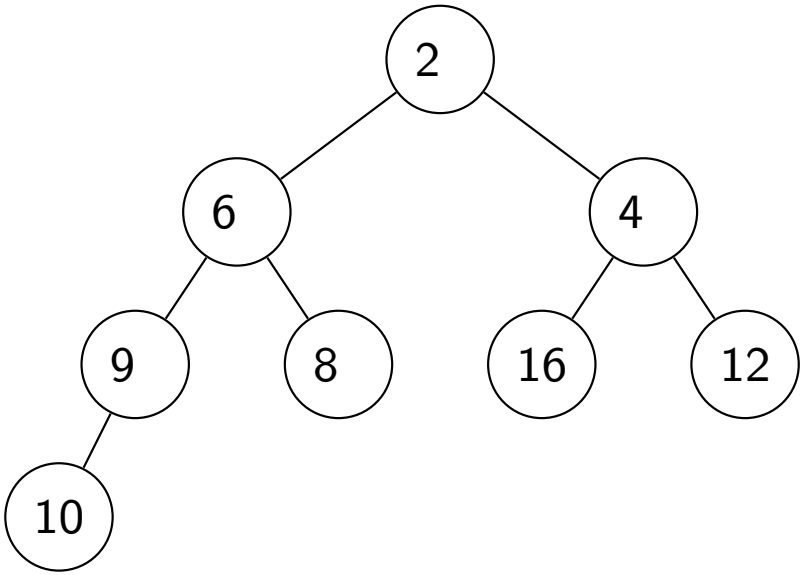
After `removeMin()`:



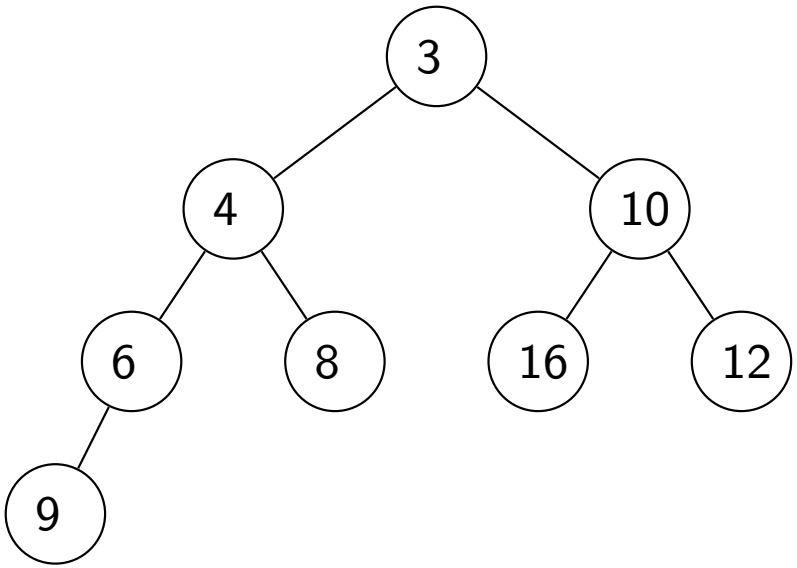
After `removeMin()`:

Heaps

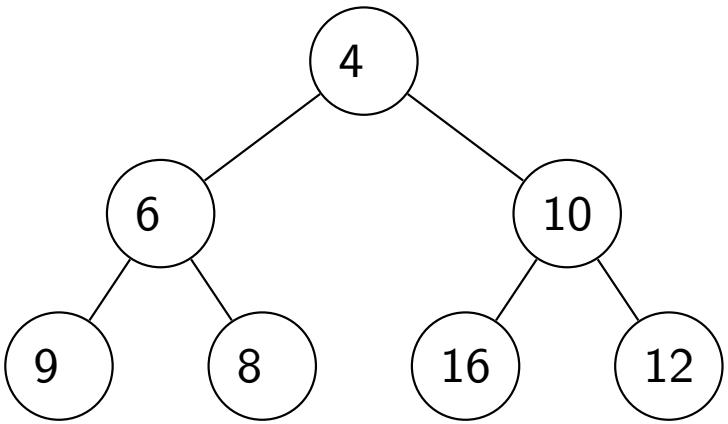
After `insert(10)`:



After `insert(3)`:



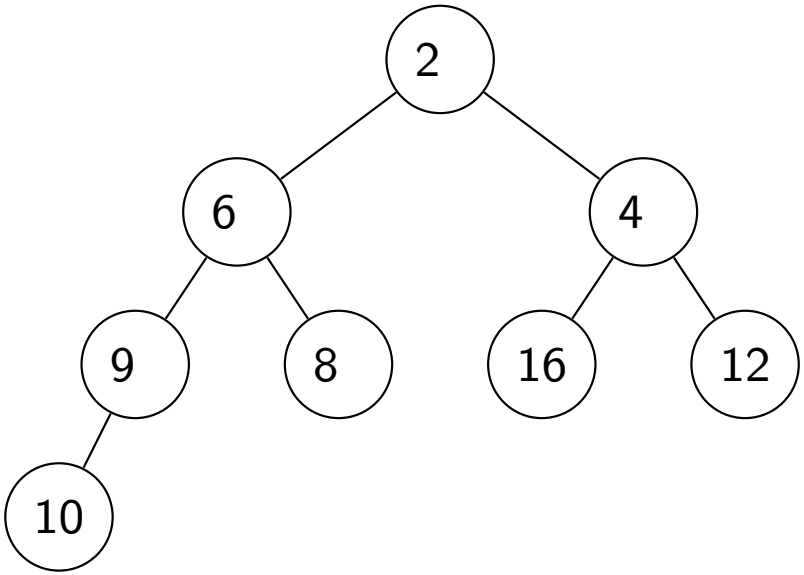
After `removeMin()`:



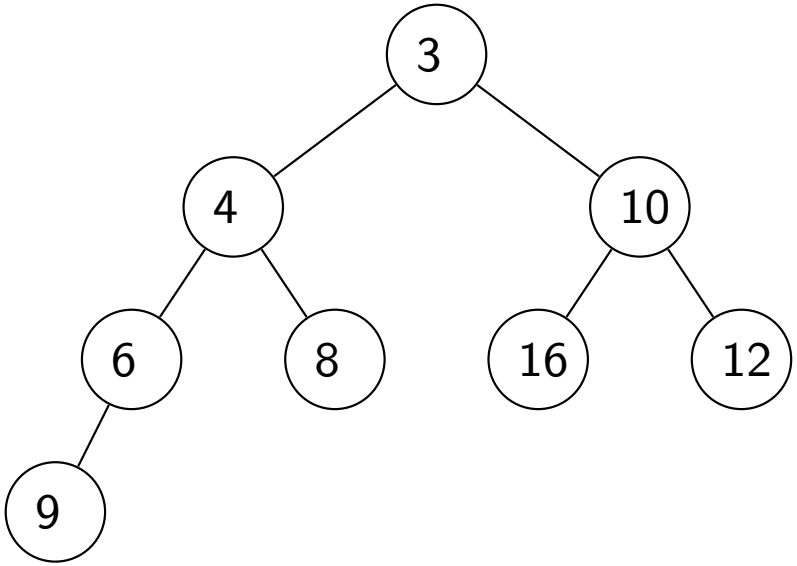
After `removeMin()`:

Heaps

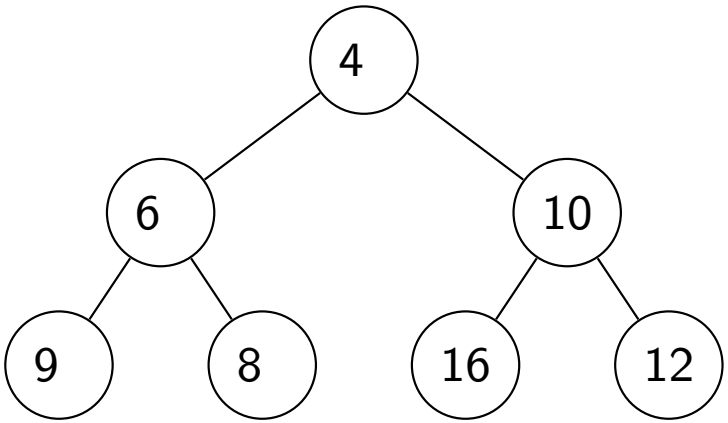
After `insert(10)`:



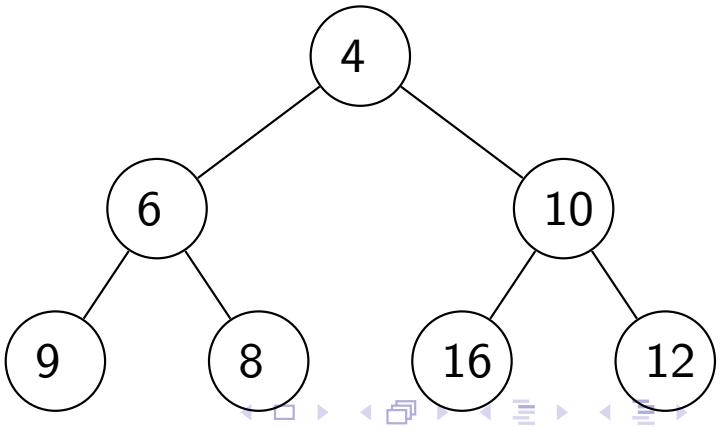
After `insert(3)`:



After `removeMin()`:



After `removeMin()`:



Heaps

Describe how you can implement a method `removeKthMin()` that, assuming there are $n > k$ nodes in the heap, removes the kth smallest item in $O(k \log n)$ time.

Heaps

Describe how you can implement a method `removeKthMin()` that, assuming there are $n > k$ nodes in the heap, removes the kth smallest item in $O(k \log n)$ time.

Answer: Call `removeMin()` k times and store the k smallest values. Insert all of them back in except for the kth smallest, and return it.
This takes $O(2k \log n) = O(k \log n)$ time.

Collections!

ArrayLists vs. LinkedLists

Collections!

SHOWDOWN:

You want to look at the m^{th} element in a sequence of length n (assume $m \leq n$)

ArrayList or LinkedList

Collections!

SHOWDOWN:

You want to look at the m^{th} element in a sequence of length n (assume $m \leq n$)

ArrayList or LinkedList

$O(1)$ vs. $O(n)$

Collections!

SHOWDOWN:

You want to add a bunch of elements, one at a time, to the beginning of the sequence. Assume there are a lot of elements you want to add.

ArrayList or LinkedList

Collections!

SHOWDOWN:

You want to add a bunch of elements, one at a time, to the beginning of the sequence. Assume there are a lot of elements you want to add.

ArrayList or LinkedList

$O(n)$ vs. $O(1)$

Collections!

SHOWDOWN:

You want to run binary search.

ArrayList or LinkedList

Collections!

SHOWDOWN:

You want to run binary search.

ArrayList or LinkedList

Remember, you can't index a LinkedList!

Collections!

ArrayList or LinkedList?

LinkedLists are best for quick pointer manipulations, whereas ArrayLists are best in cases that benefit from constant-time indexing.

Stacks and Queues

STACKS:

- First in, last out
- Depth-first recursion (maze solver, depth-first traversal, tree traversal)
- Possible implementation: linked list, add to head, remove from head

QUEUES:

- First in, first out
- Breadth-first recursion
- Possible implementation: linked list, add to head, remove from tail

Stacks on Stacks (on Stacks?)

Implement "enqueue" (add to front, like push) and "dequeue" (remove from end, like pop) methods for a **queue** of Objects using **stacks**. Use `java.util.Stack`, and the following methods:

- `boolean empty()` - returns True if empty
- `Object pop()` - returns the top of the stack
- `void push(Object o)` - push an object to the top of the stack

```
1 public void queue(Object o){
2     // Fill in here
3 }
4 public Object dequeue(){
5     // Fill in here
6 }
```

(Hints: is one stack really enough? You may have variables initialized outside of the methods. Also, efficiency is not a concern here.)

Stacks on Stacks (on Stacks?)

Implement "enqueue" (add to front, like push) and "dequeue" (remove from end, like pop) methods for a **queue** of Objects using **stacks**. Solution 1:

```
1  private Stack<Object> inValues = new Stack<Object>();
2  private Stack<Object> outValues = new Stack<Object>();
3
4  public void queue(Object o){
5      inValues.push(o);
6  }
7
8  public Object dequeue(){
9      if(outValues.empty()){
10         while(!inValues.empty()){
11             outValues.push(inValues.pop());
12         }
13     }
14     return outValues.pop();
15 }
```

Stacks on Stacks (on Stacks?)

Implement "enqueue" (add to front, like push) and "dequeue" (remove from end, like pop) methods for a **queue** of Objects using **stacks**. Solution 2:

```
1  private Stack<Object> values = new Stack<Object>();
2
3  public void queue(Object o) {
4      if (values.empty()) {
5          values.push(o);
6          return;
7      } else {
8          Object oldTop = values.pop();
9          queue(o);
10         stack.push(oldTop);
11     }
12 }
13
14 public Object dequeue() {
15     return values.pop();
16 }
```

More Collections!

TreeSet:

- *Ordered* - can quickly find minimum and maximum
- Similar to a BST - guaranteed $O(\log n)$ time for certain operations

HashSet:

- Best for checking membership
- Constant time lookup
- *Unordered*

More Collections!

You have an array of integers of arbitrary length $n > 0$.

You know nothing about these integers, except that the array has only one unique element, and the rest are duplicates.

Write a method to return the unique integer:

```
1 int returnUnique(int[] ary){
2     // Fill in here
3 }
```

More Collections!

Solution:

```
1  int returnUnique(int[] ary){
2      HashSet duplicates = new HashSet();
3      for(int num : ary){
4          if (duplicates.contains(num)){
5              duplicates.remove(num);
6          } else {
7              duplicates.add(num);
8          }
9      }
10     Iterator unique = duplicates.iterator();
11     return unique.next();
12 }
```

What is Insertion Sort?

Start with an empty list S and the unsorted list I of n input items.

Pseudocode:
for (each item x in I):
    insert x into the list S, positioned so that S remains in sorted order

This is usually done in-place.

Insertion Sort

```
int[] arr = {2, 9, 7, 1, 9, 3};
```

How would this array be sorted using in-place insertion sort?

Insertion Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}

Insertion Sort

```
int[] arr = {2, 9, 7, 1, 9, 3};
```

How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}
{2, 9, 7, 1, 9, 3}

Insertion Sort

```
int[] arr = {2, 9, 7, 1, 9, 3};
```

How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}
{2, 9, 7, 1, 9, 3}
{2, 7, 9, 1, 9, 3}

Insertion Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}
{2, 9, 7, 1, 9, 3}
{2, 7, 9, 1, 9, 3}
{1, 2, 7, 9, 9, 3}

Insertion Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}
{2, 9, 7, 1, 9, 3}
{2, 7, 9, 1, 9, 3}
{1, 2, 7, 9, 9, 3}
{1, 2, 7, 9, 9, 3}

Insertion Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place insertion sort?

{2, 9, 7, 1, 9, 3}
{2, 9, 7, 1, 9, 3}
{2, 7, 9, 1, 9, 3}
{1, 2, 7, 9, 9, 3}
{1, 2, 7, 9, 9, 3}
{1, 2, 3, 7, 9, 9}

What is Selection Sort?

Start with an empty list S and the unsorted list I of n input items.

Pseudocode:
for $(i = 0; i < n; i++)$:
    Find item x in I that has the smallest key.
    Remove x from I
    Append x to the end of S

Again, this is usually done in-place.

Selection Sort

```
int[] arr = {2, 9, 7, 1, 9, 3};
```

How would this array be sorted using in-place selection sort?

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}
{1, 2, 9, 7, 9, 3}

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}
{1, 2, 9, 7, 9, 3}
{1, 2, 3, 9, 7, 9}

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}
{1, 2, 9, 7, 9, 3}
{1, 2, 3, 9, 7, 9}
{1, 2, 3, 7, 9, 9}

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}
{1, 2, 9, 7, 9, 3}
{1, 2, 3, 9, 7, 9}
{1, 2, 3, 7, 9, 9}
{1, 2, 3, 7, 9, 9}

Selection Sort

`int[] arr = {2, 9, 7, 1, 9, 3};`
How would this array be sorted using in-place selection sort?

{1, 2, 9, 7, 9, 3}
{1, 2, 9, 7, 9, 3}
{1, 2, 3, 9, 7, 9}
{1, 2, 3, 7, 9, 9}
{1, 2, 3, 7, 9, 9}
{1, 2, 3, 7, 9, 9}

What is Mergesort?

- Start with the unsorted list A of n input items.
- Break A into two halves A_1 and A_2, having $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ items.
- Sort A_1 recursively, yielding the sorted list S_1.
- Sort A_2 recursively, yielding the sorted list S_2.
- Merge S_1 and S_2 into a sorted list S.

```
int[] arr = {2, 9, 2, 4, 1, 1, 8, 7, 5, 9};
```

How would this array be sorted using mergesort?

What is Quicksort?

- Start with the unsorted list A of n input items.
- Choose a pivot item v from A.
- Partition A into two unsorted lists A_1 and A_2.
  - A_1 contains all items whose keys are smaller than v's key.
  - A_2 contains all items whose keys are larger than v's key.
  - Items with the same key as v can go into either list.
  - The pivot v, however, does not go into either list.
- Sort A_1 recursively, yielding the sorted list S_1.
- Sort A_2 recursively, yielding the sorted list S_2.
- Concatenate S_1, v, and S_2 together, yielding the sorted list S.

`int[] arr =` {`2, 9, 2, 4, 1, 1, 8, 7, 5, 9`}
How would this array be sorted using quicksort?

What is Counting Sort?

- Iterate through the array A to find the "counts" of each key.
- Create an array of running sums of the number of keys LESS THAN the current value (counts[i] contains the number of keys less than i).
- Reconstruct the array by iterating through A and placing each key in the appropriate location based on counts.

Counting Sort Example

```
int[] arr = {5, 5, 4, 5, 9, 8, 3, 9, 3, 1, 5, 1, 6, 5, 8, 8, 4,
7, 5, 1, 4, 4, 3, 9, 5};
```

1. Construct counts array:

```
1 for (i = 0; i < arr.length; i += 1){
2     counts[arr[i]] += 1;
3 }
```

0	1	2	3	4	5	6	7	8	9
0	3	0	3	4	7	1	1	3	3

Counting Sort Example

0	1	2	3	4	5	6	7	8	9
0	3	0	3	4	7	1	1	3	3

2. Scan the `counts` array so that `counts[i]` contains the number of keys less than `i`.

```
1 int total = 0;
2 int c;
3 for (int j = 0; j < counts.length; j++){
4     c = counts[j];
5     counts[j] = total;
6     total = total + c;
7 }
```

0	1	2	3	4	5	6	7	8	9
0	0	3	3	6	6	10	17	19	22

Counting Sort Example

0	1	2	3	4	5	6	7	8	9
0	0	3	3	6	6	10	17	19	22

3. Let s be the sorted output array. Walk through array x and copy each item to its final position in y. When you copy key k, you must increment `counts[k]` to make sure that the next item with key k goes into the next slot.

```
1 for (int i = 0; i < arr.length; i += 1){
2     y[counts[arr[i]]] = x[i];
3     counts[arr[i]] += 1;
4 }
```

```
int[] sorted = {1, 1, 1, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 6, 7, 8,
8, 9, 9, 9}
```

What is Radix Sort?

- Sort number of buckets q (a.k.a. radix) at one time, from least to most significant.
- After the most significant radix is sorted, the numbers are completely sorted.
- This works because counting sort is stable.

```
int[] arr = {134, 63, 874, 907, 975, 191, 575, 758, 624, 8, 290,
923, 907, 199, 898, 390, 530, 355, 611, 299};
```

How would this array be sorted using radix sort with $q = 10$ buckets?

Hash Tables

- Stores a set of keys, values
- Useful because `find` and `add` are constant amortized time
- Hashcode: used to map a specific key to a bucket in the hashtable

Hash Tables

```
int[] arr = {15, 32, 75, 41, 75, 6, 76, 51, 82, 66};
```

```
1 int n = 10;
2 private int hashCode(int i){
3     return i % n;
4 }
```

What would the has table look like with:

- external chaining?
- open addressing?

That's it!

Good luck on your midterm and thanks for coming!
Please fill out a feedback form before you leave!