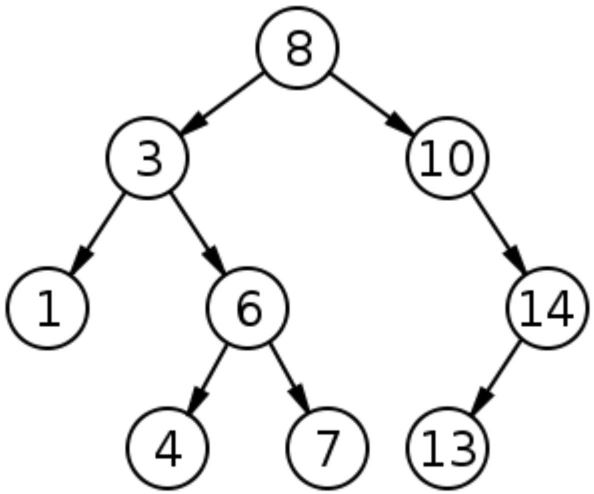
HKN CS61B Midterm 2 Review

Alec Mouri
Evan Ye
Jene Li
Manuk Hovanesian
Riyaz Faizullabhoy

Binary Tree

Binary Tree:

- Each node has max 2 children
- N nodes, h height
  - worst case h is O(N)
  - average case h is O(log N)



Binary **Search** Tree:

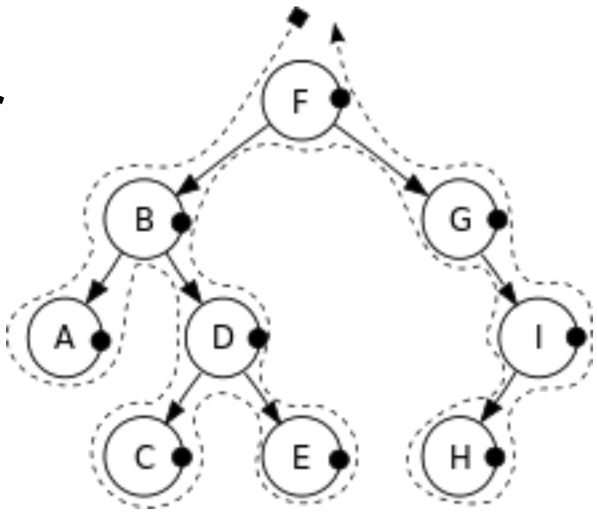
- All nodes in LEFT subtree < root
- All nodes in RIGHT subtree > root

Post Order

For each node n

- recursively traverse n.left child
- recursively traverse n.right child
- traverse n

We traverse a node only after traversing its children



A, C, E, D, B, H, I, G, F

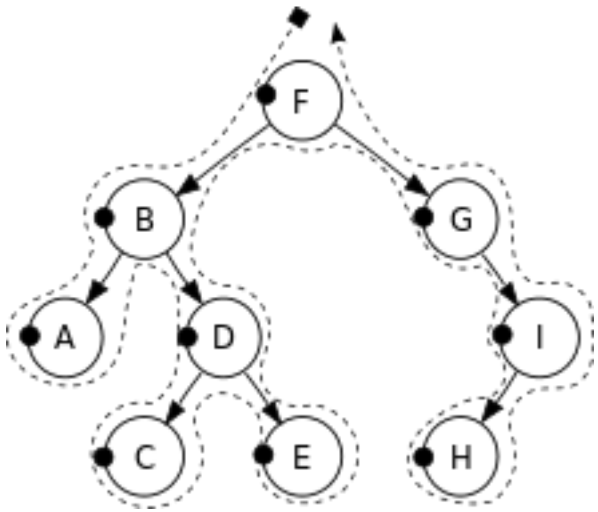
Pre Order

For each node n

- traverse n
- recursively traverse n.left child
- recursively traverse n.right child

We traverse a node before traversing either child

F, B, A, D, C, E, G, I, H



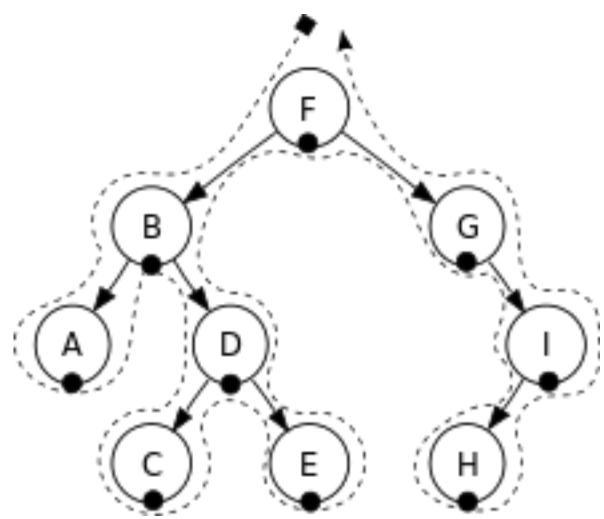
In Order

For each node n

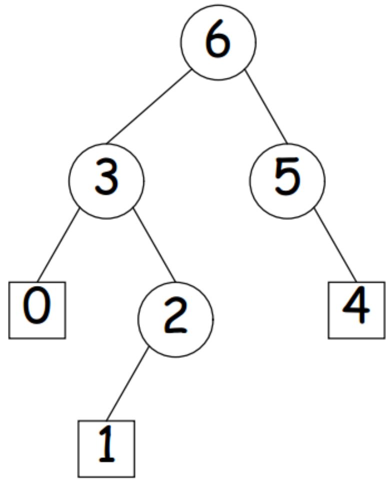
- recursively traverse n.left child
- traverse n
- recursively traverse n.right child

We traverse a node after the left child, but before the right

A, B, C, D, E, F, G, H, I



Traversing BST



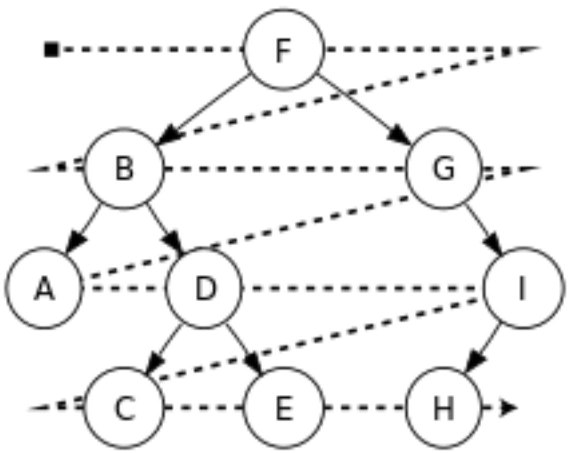
Postorder *Preorder* *inorder*

Breadth First Traversal

Pre/Post/In -order traversals are Depth First

We also have Breadth First Traversal.

F, B, G, A, D, I, C, E, H



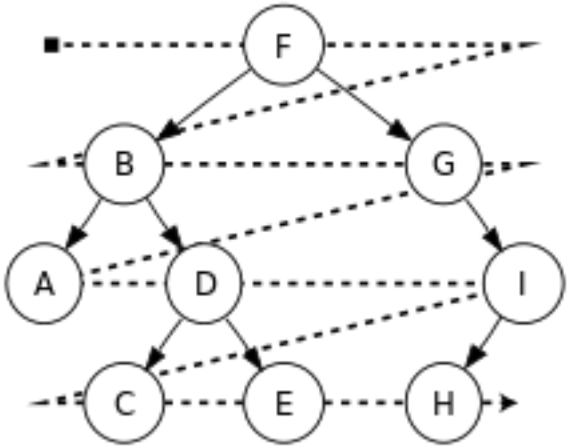
BFS Traversal

For each node n

- Traverse node n
- add n.left and n.right children to queue
- traverse and remove the next node from the queue

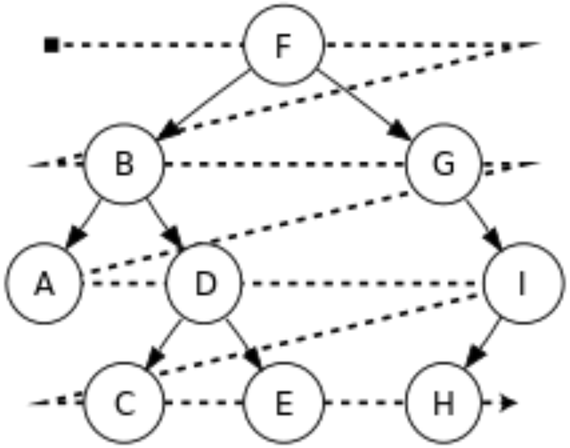
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G



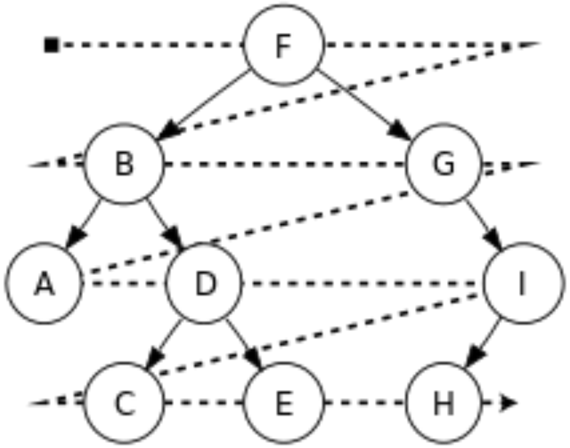
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D



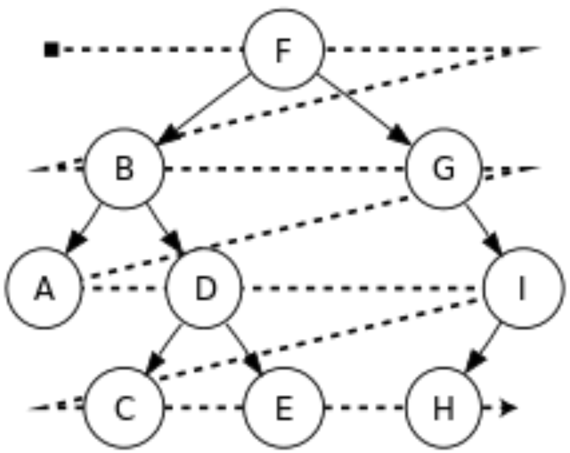
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D
- Remove and traverse G
  - Queue: A, D, I



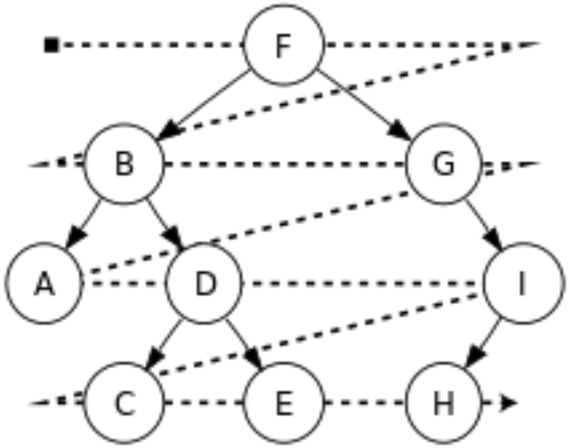
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D
- Remove and traverse G
  - Queue: A, D, I
- Remove and traverse A
  - Queue: D, I



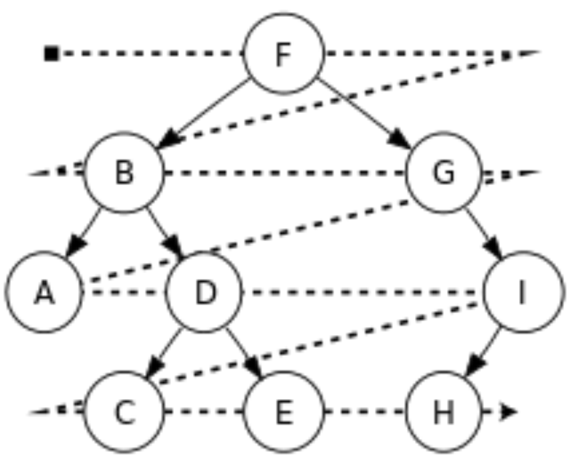
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D
- Remove and traverse G
  - Queue: A, D, I
- Remove and traverse A
  - Queue: D, I
- Remove and traverse D
  - Queue: I, C, E



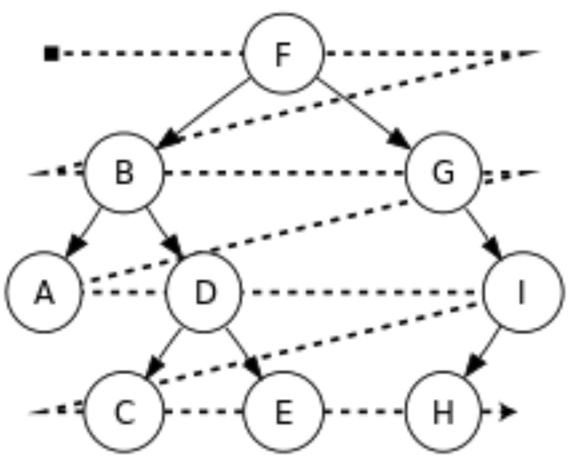
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D
- Remove and traverse G
  - Queue: A, D, I
- Remove and traverse A
  - Queue: D, I
- Remove and traverse D
  - Queue: I, C, E
- Remove and traverse I
  - Queue: C, E, H



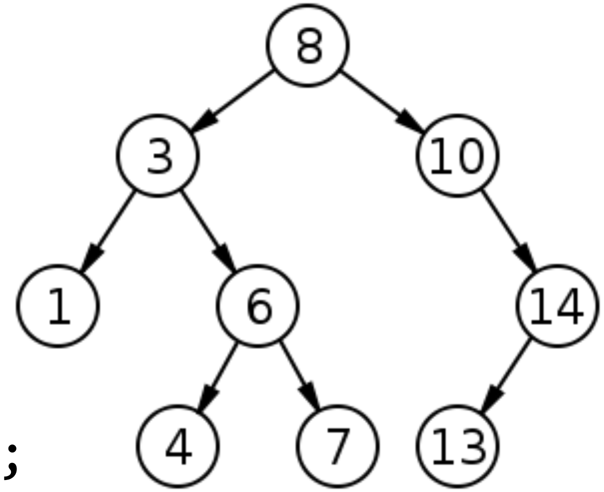
BFS Traversal (Example)

- Begin with F: traverse F and queue its children
  - Queue: B, G
- Remove B from queue and traverse B
  - Queue: G, A, D
- Remove and traverse G
  - Queue: A, D, I
- Remove and traverse A
  - Queue: D, I
- Remove and traverse D
  - Queue: I, C, E
- Remove and traverse I
  - Queue: C, E, H
- Remove and traverse C, E, and H



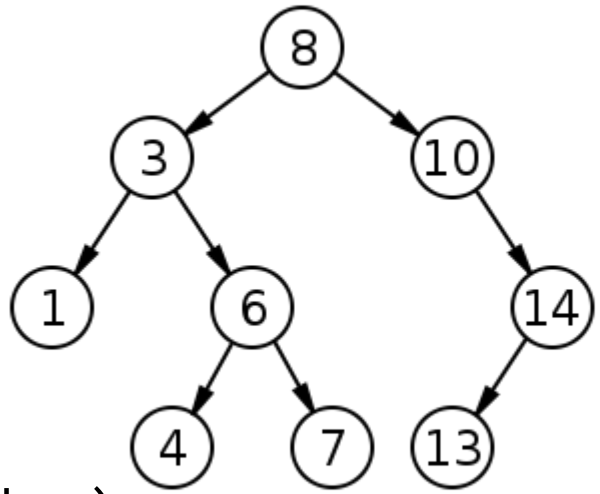
BST Find

```
bst_find(tree, key):
    if (tree == null)
        return false;

    if (key == tree.value)
        return true;

    if (key < tree.value)
        return bst_find(tree.left, key);
    else
        return bst_find(tree.right, key);
```

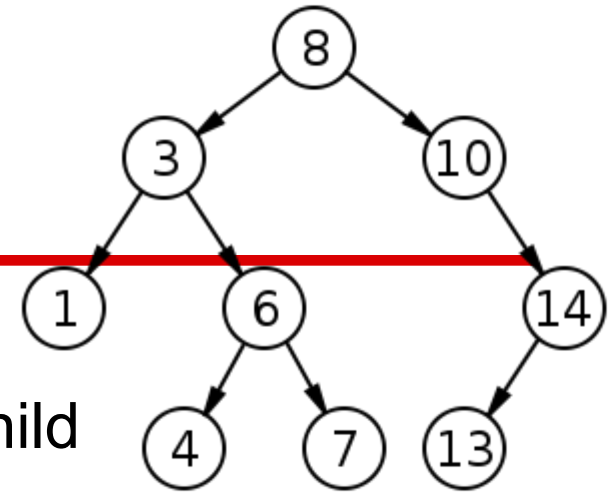


BST Insert (Unique Keys)

```
bst_insert(tree, key):
    if (tree == null)
        return new Tree(key);

    if (key == tree.value)
        return tree;

    if (key < tree.value)
        tree.left = bst_insert(tree.left, key);
    else
        tree.right = bst_insert(tree.right, key);

    return tree;
```



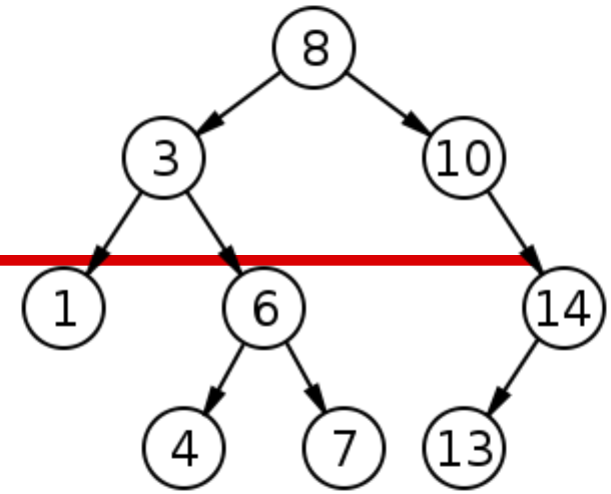
BST Remove?



- Removing leaf nodes is easy.
- Nodes with one child : 'promote' the child
- Internal nodes (2 children)
  - Find the 'successor' (min node greater than deleted node)
  - Replace deleted node with 'successor'
  - Remove successor
    - What about children?
    - Max 1 child -- promote it.

- Think about why above operations preserve BST property.

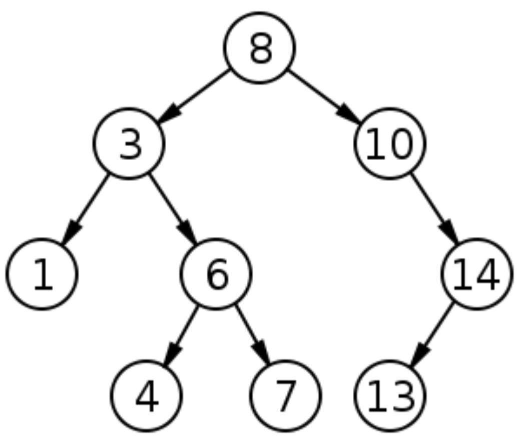
BST Successor




```
bst_next(node):
    if (node.right)
        return min(node.right);
    return first_right_parent(node);

first_right_parent(node):
    while (node.parent != null && node.parent.right == node) {
        node = node.parent;
    }
    return node.parent;

min(tree):
    //try this yourself
    //hint: are smaller nodes on the left or right?
```

Some BST Runtimes

Operation	Average Case	Worst Case
Find	O(log n)	O(n)
Insert	O(log n)	O(n)
Remove	O(log n)	O(n)
Construct	O(nlog n)	

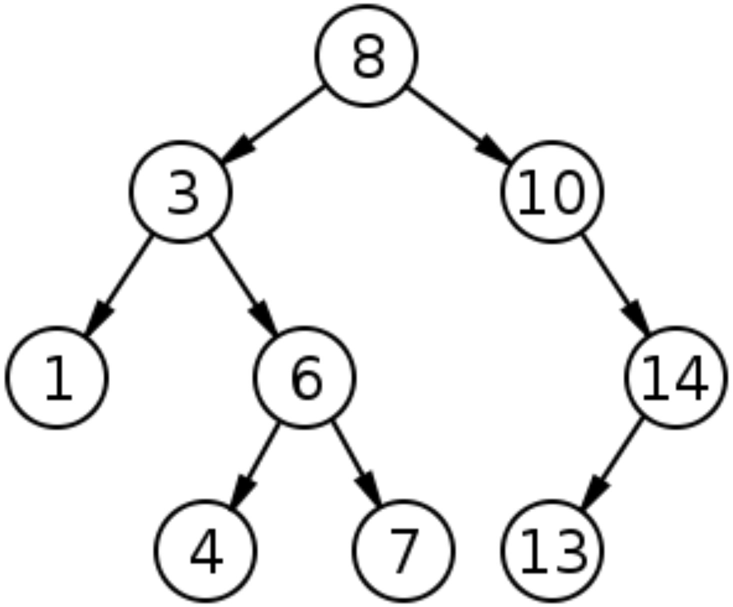


Everything is O (n) worst case!

Why?

BST Quiz Questions

- Write a function for
  Checking if a tree is a BST



Checking BST Property: First Try

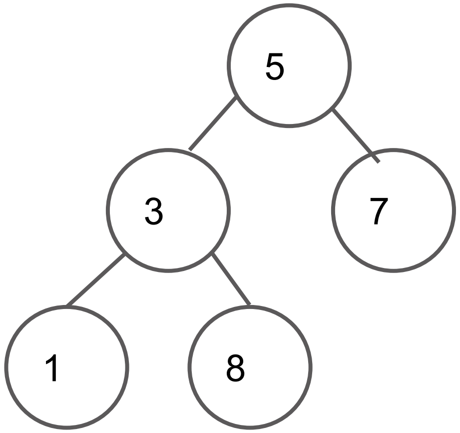
```
is_bst(tree) {
  if(!tree) return true;
  return tree.left.value < tree.value
    && tree.right.value > tree.value
    && is_bst(tree.left)
    && is_bst(tree.right)
}
```

What's wrong with this?

Checking BST Property: First Try

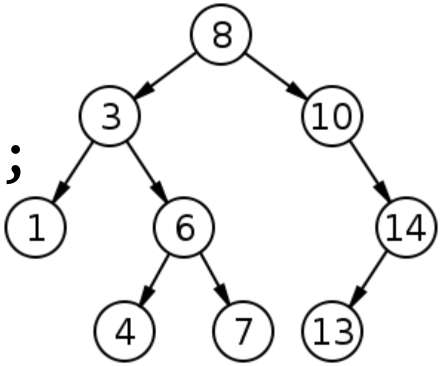
```
is_bst(tree) {
   if(!tree) return true;
   return tree.left.value < tree.value
      && tree.right.value > tree.value
      && is_bst(tree.left)
      && is_bst(tree.right)
}
```

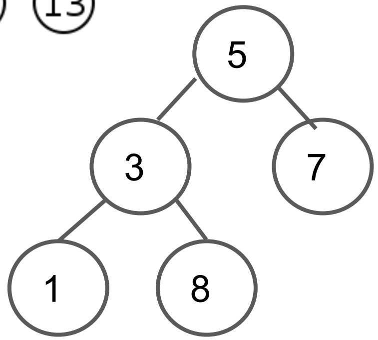
Fails for trees like:



Checking BST Property: Correct

```
is_bst(tree) {
    return is_bst(tree, -inf, +inf);
}

is_bst(tree, min, max) {
    if (!tree) return true;
    return     tree.value > min
            && tree.value < max
            && is_bst(tree.left, min, tree.value)
            && is_bst(tree.right, tree.value, max);
}
```



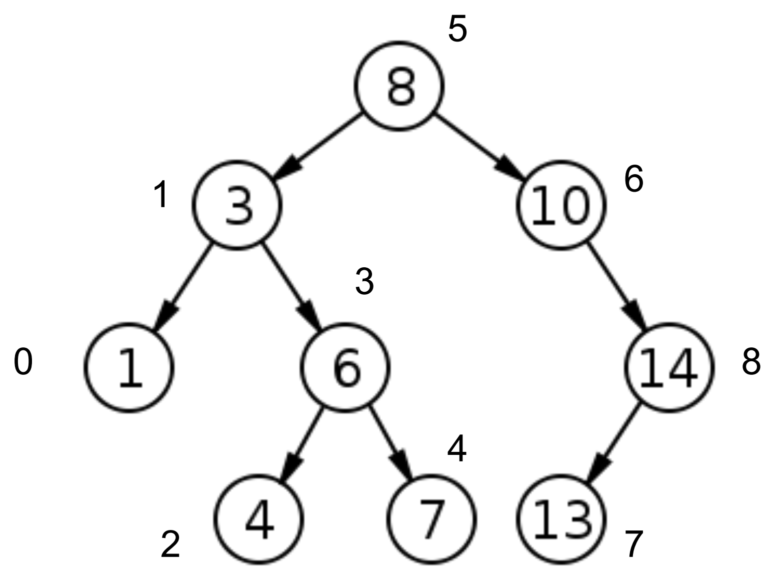


BST Intermediate Questions

- Given a BST annotated by its in-order traversal sequence, find the median element. (Best/Avg/Worst case?)

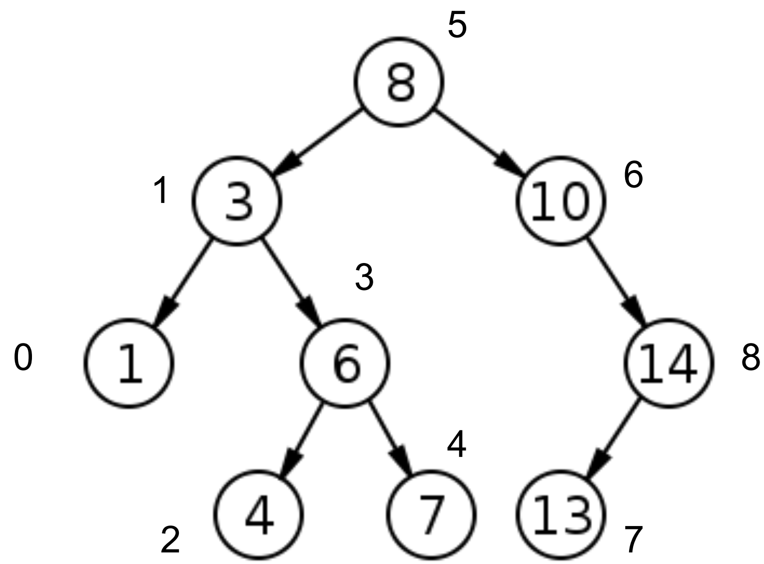
BST Intermediate Questions

- Given a BST annotated by its in-order traversal sequence, find the median element. (Best/Avg/Worst case?)
- Example:



BST Intermediate Questions

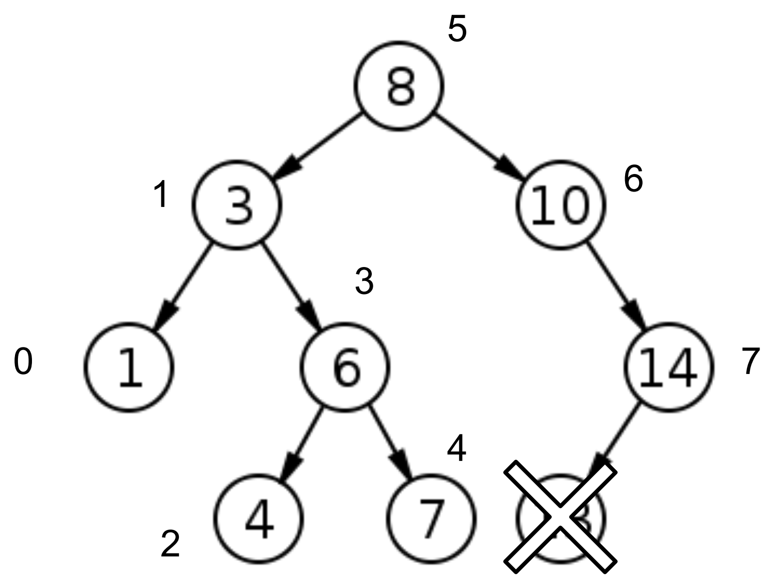
With n (odd) nodes, the median will be the n/2 'th node in sorted order. Remember that traversing in-order goes through the nodes in sorted order.



BST Intermediate Questions

What if n is even?

Average the n/2 'th and n/2+1 'th nodes.

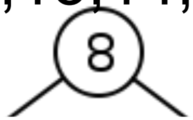


BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8

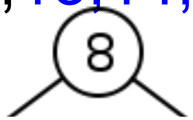
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8



BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8

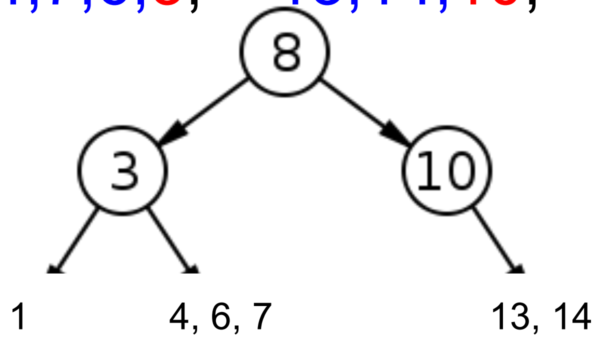


1, 3, 4, 6, 7

10, 13, 14

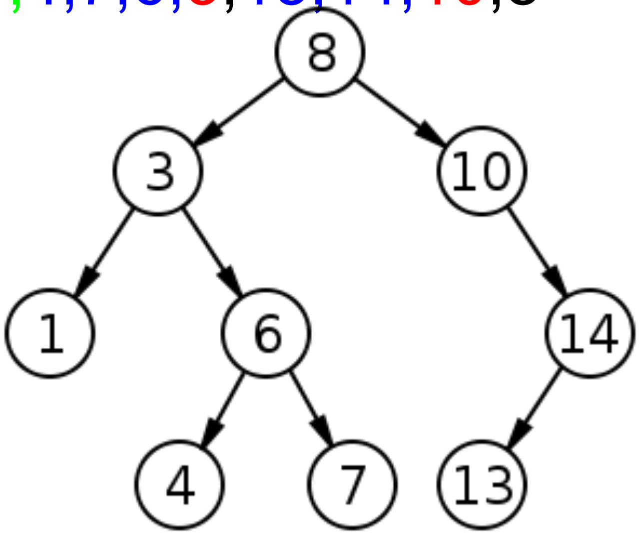
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
    - In-order: 1,3,4,6,7, 8, 10,13,14
    - Post-order: 1,4,7,6,3, 13,14,10, 8



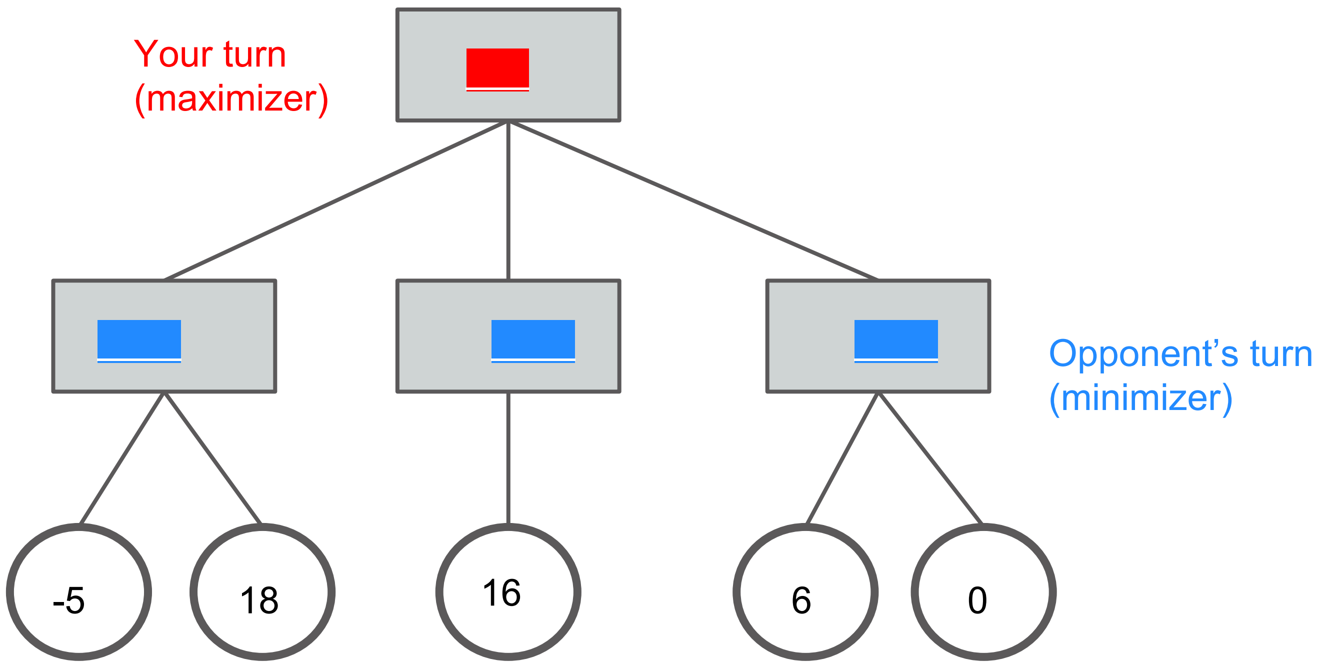
BST Intermediate Questions

- Reconstruct a BST from two of the three: {pre-order, post-order, in-order} traversals.
- Example:
  - In-order: 1,3,4,6,7,8,10,13,14
  - Post-order: 1,4,7,6,3,13,14,10,8



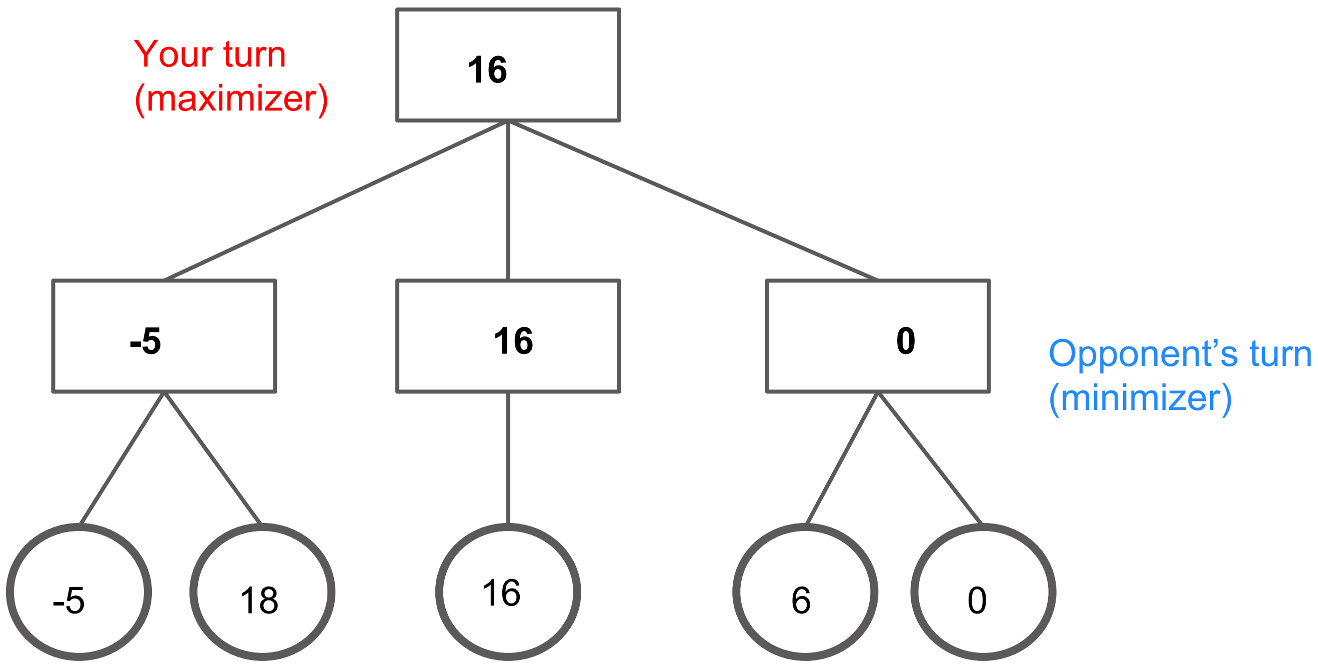
Minimax

Complete minimax on the tree below:



Minimax

Complete minimax on the tree below:



Alpha Beta Pruning

Idea: make minimax faster!

α - best (highest) guaranteed score seen so far for a maximizer node.

-Starts at negative infinity

β - the best (lowest) guaranteed score seen so far for a minimizer node

-Starts at positive infinity

Prune Cases

- prune at a minimizer node whose β value is less than or equal to the α value of any of its maximizer node ancestors

- prune at a maximizer node whose α value is greater than or equal to the β value of any of its minimizer node ancestors

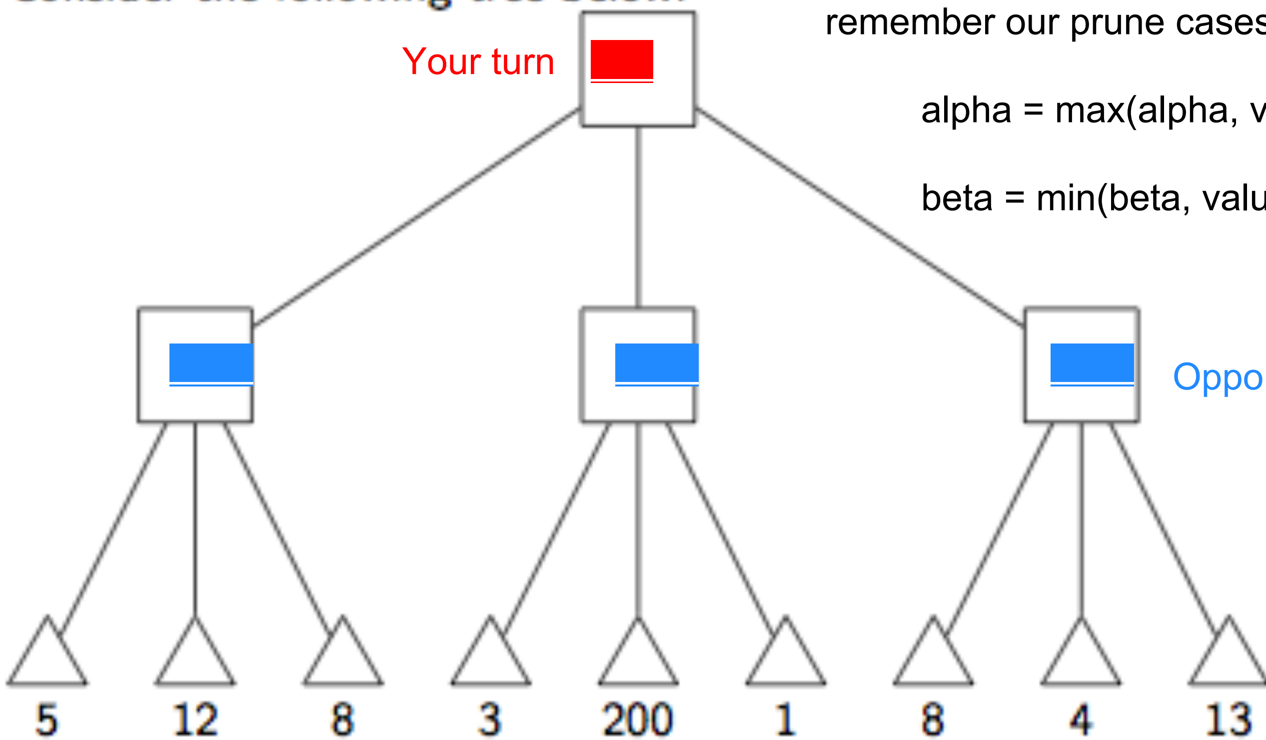
Alpha Beta Pruning

Consider the following tree below:

remember our prune cases, and:

alpha = max(alpha, value_below_max_node)

beta = min(beta, value_below_min_node)



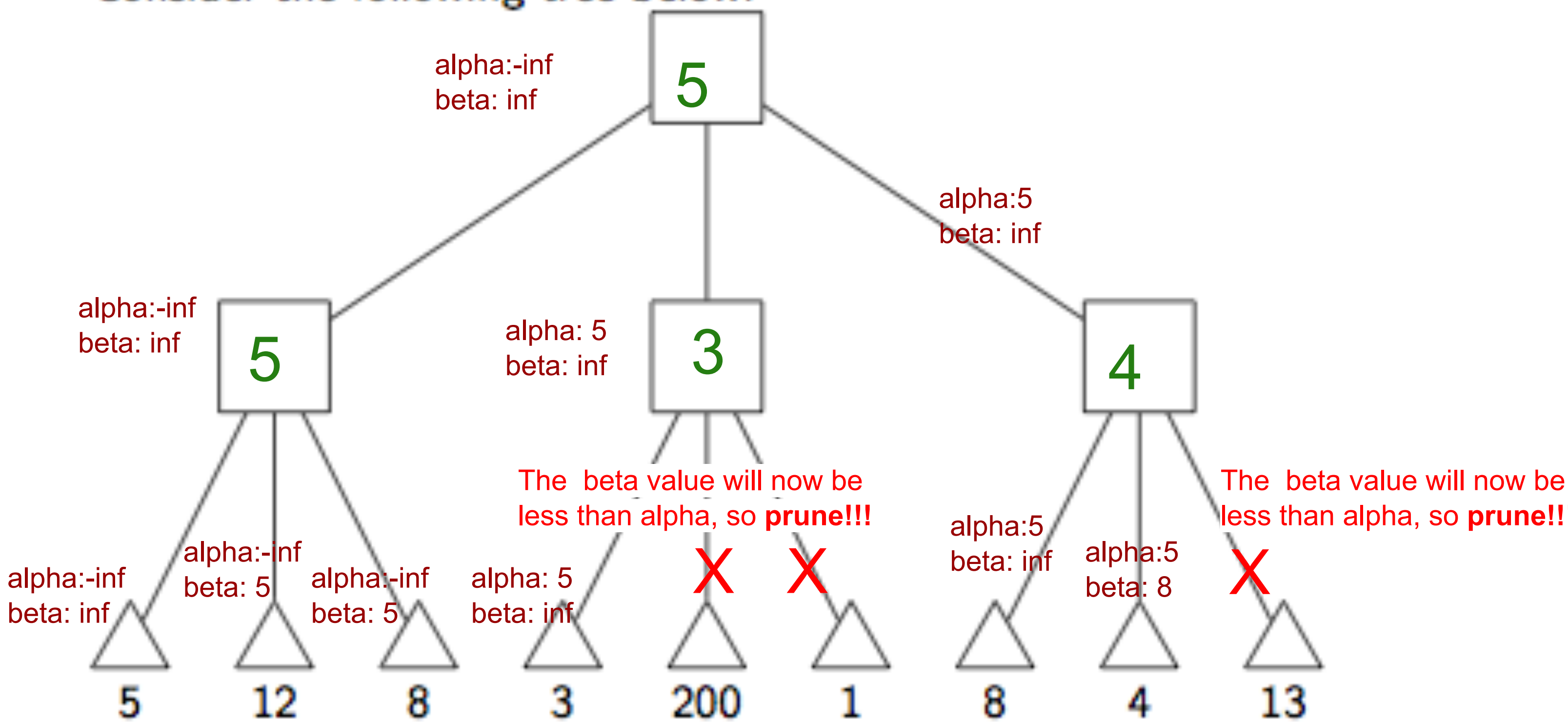
Opponent's turn

Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

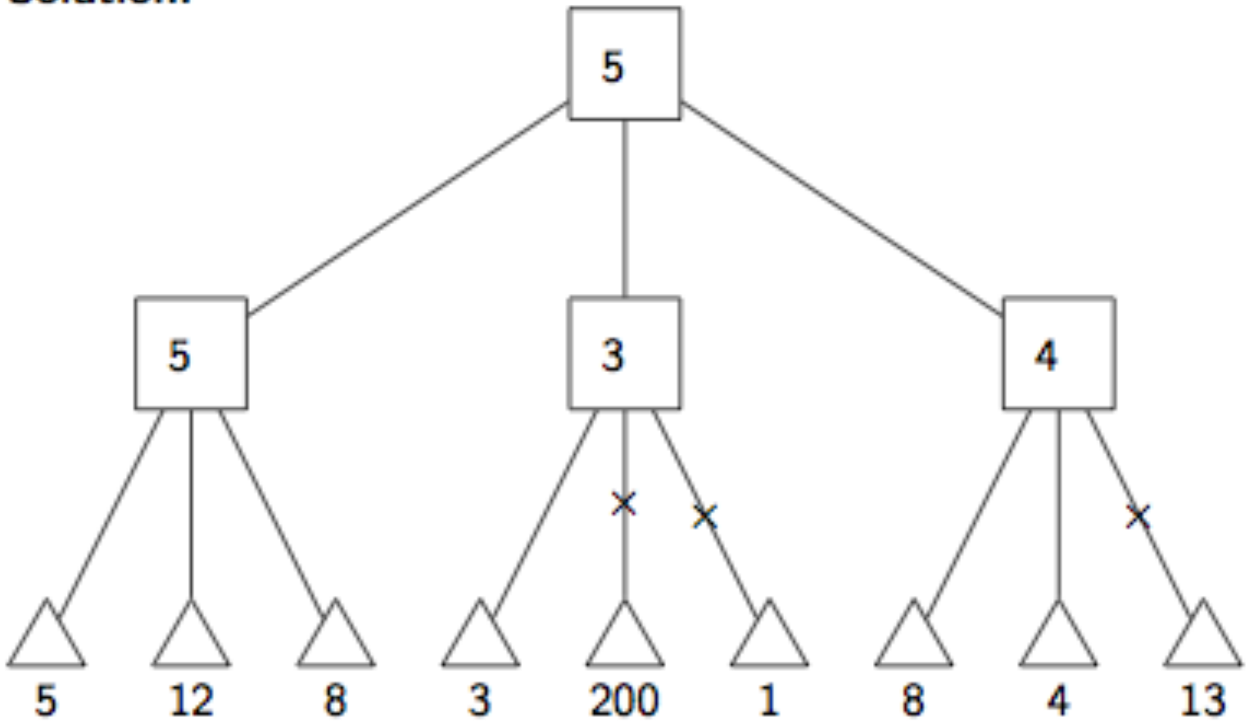


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Solution:



Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

   True -- try doing minimax on our example

2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

   False -- try doing minimax on our example

3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

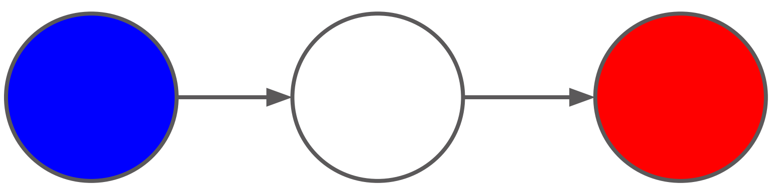
   True -- what would have happened if we started with the middle minimizer's children first?

Searches

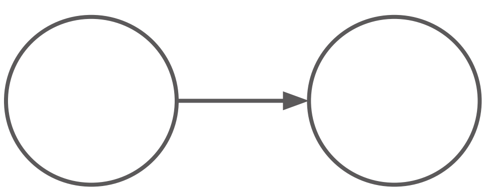
You have a singly linked list of `Nodes`, which have `next` pointers and String value `color`. The possible colors are "red," "blue," or "green."
(ex: `Node n = Node.next, String color = Node.color`)

Some nodes may be pre-colored, others simply have `null` color.
Now, given a pointer to the head of the linked list, determine how many colorings of the list <u>**DO NOT**</u> result in consecutive nodes of the same color

EX:



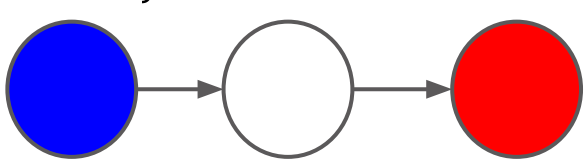
Ans: 1 coloring - middle node can ONLY be green



Ans: 6 colorings: RB, RG, BR, BG, GR, GB

Searches: Use Backtracking!

```
boolean find_path(curr, past_color, path) {
     if (curr.color.equals(past_color)) // if the node is pre-colored
“incorrectly”
          return false;
     else {
          for (color in COLORS) {
               if (!color.equals(past_color)) {
                    if (!find_path(curr.next, color, path+[color]) {
                         path.remove(path.size() - 1); // bad path, remove it
                    } else {
                         return true;
                    }
               }
          }
          return false;
     }
}
```



EX: Call: `path = []`

`find_path(head, null, path)`

Heaps

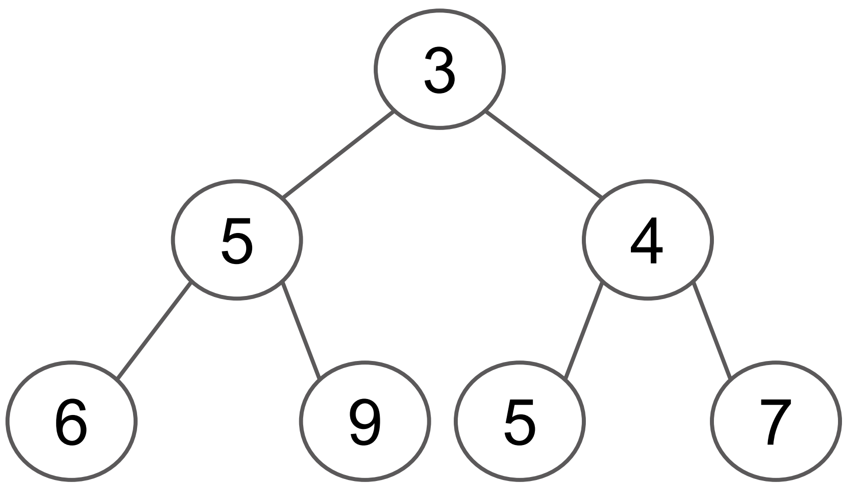
A heap is a binary tree with extra properties:

- The tree is complete
  - Recall: complete means that every level is filled, except possibly the last, which is filled left to right.

- Heap-order property: If node B is a descendant of node A:
  - Min heap: key of B >= key of A
  - Max heap: key of B <= key of A

Heaps

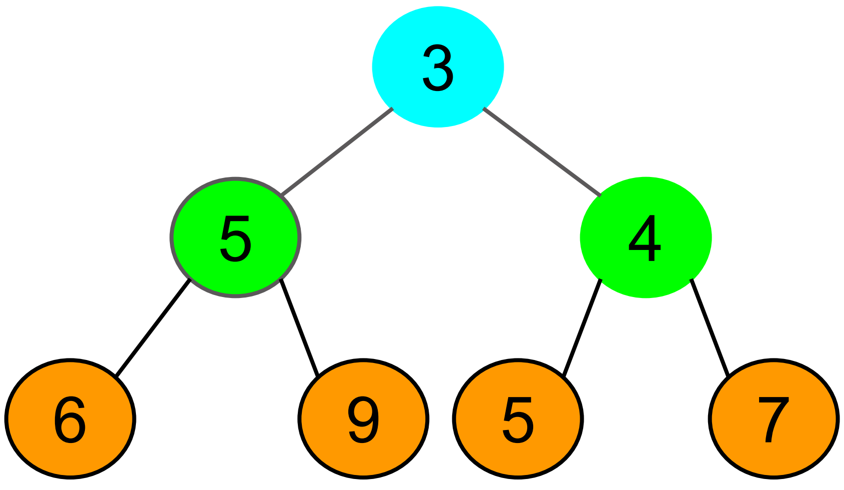
X	3	5	4	6	9	5	7



Heaps

X	3	5	4	6	9	5	7

Why do we X out the zero-th index?



Heaps, Array Representation

You are at a node with index **i**

Parent is at **floor(i/2)**

Children are at indices **2i** and **2i+1**

Ex: Key at index 3.

Parent: index 1.

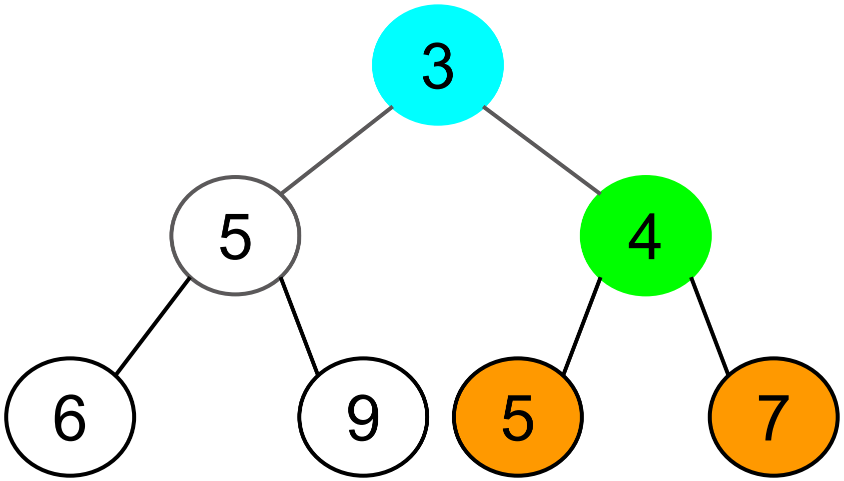
Children: indices 6 and 7.

Heaps

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

Let i = 3

parent: floor(i/2) = 1
children: 2i, 2i+1 = 6, 7



Heaps, a possibly tricksy question

True/False:

In a min heap:

Key $\mathbf{k_1}$ is at level $\mathbf{l_1}$ and key $\mathbf{k_2}$ is at level $\mathbf{l_2}$.

If $\mathbf{k_1 < k_2}$, then $\mathbf{l_1 <= l_2}$?

What if instead, we are given $\mathbf{l_1 < l_2}$.

Is $\mathbf{k_1 <= k_2}$?

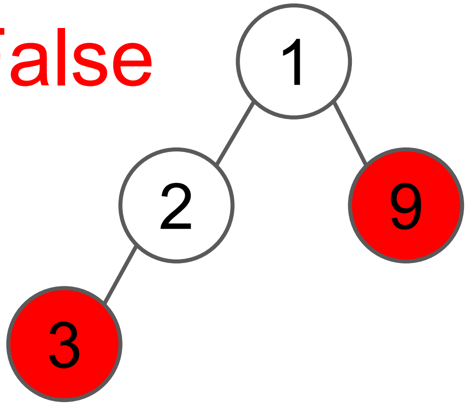
Heaps, a possibly tricksy question

True/False: In a min heap:

Key k_1 is at level l_1 and key k_2 at level l_2.
If $k_1 < k_2$, then $l_1 <= l_2$?
Or if $l_1 < l_2$, does $k_1 <= k_2$?

Ans: Both False

Ex:



k_1: 3
k_2: 9
l_1: 3
l_2: 2

Heaps, insert and removeMin

To `insert()` in a heap:

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

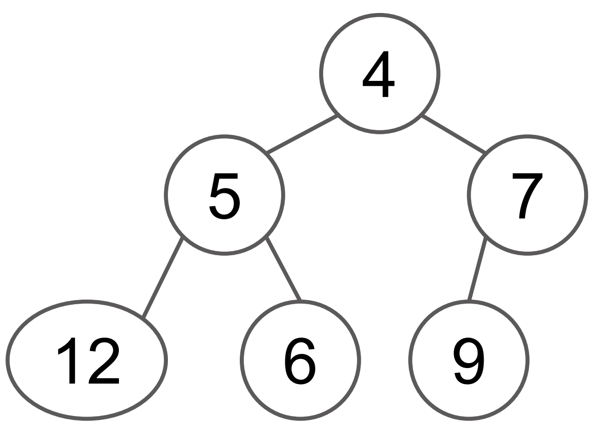
To `removeMin()`:

- Replace first element (root) with last element, and remove last node.
- Bubble down new root by repeatedly swapping with smaller of two children until heap-order property is satisfied.

Inserting into Heaps

Let's insert **-1**

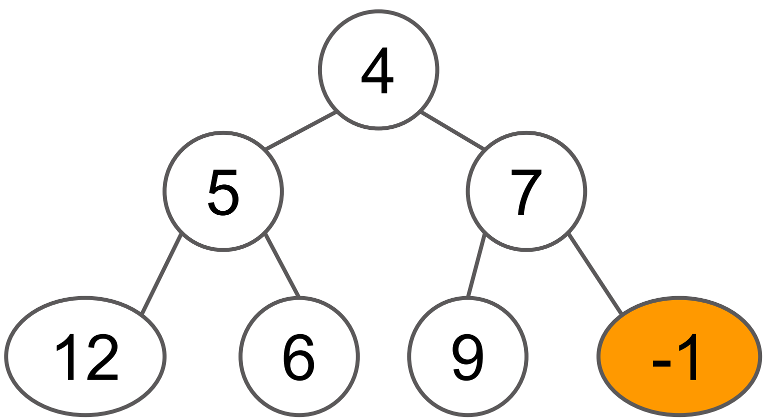
X	4	5	7	12	6	9	



Inserting into Heaps

put it at the beginning

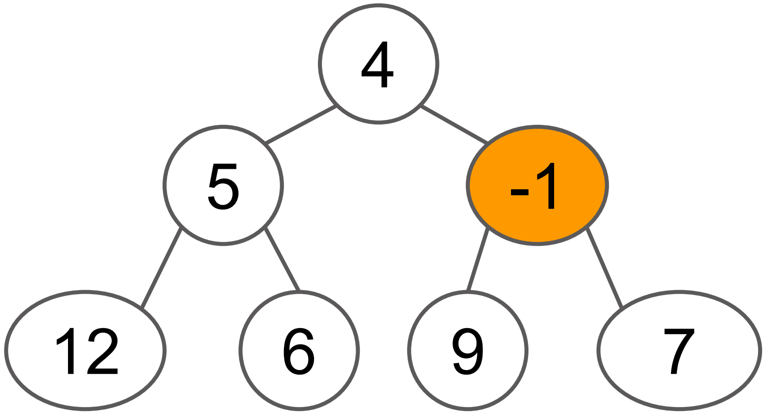
X	4	5	7	12	6	9	-1



Inserting into Heaps

bubble up...

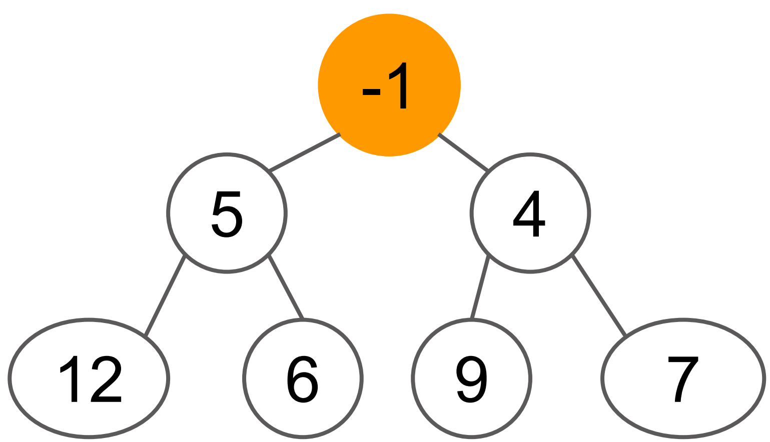
X	4	5	-1	12	6	9	7



Inserting into Heaps

bubble up.. and done!

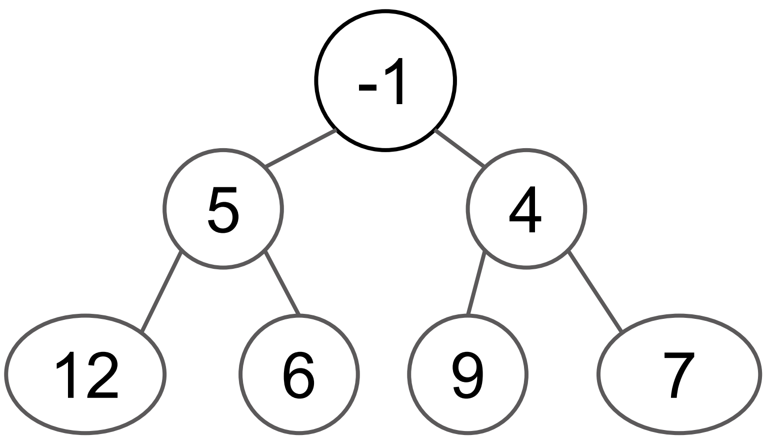
X	-1	5	4	12	6	9	7



Removing from Heaps

(a) `removeMin()`

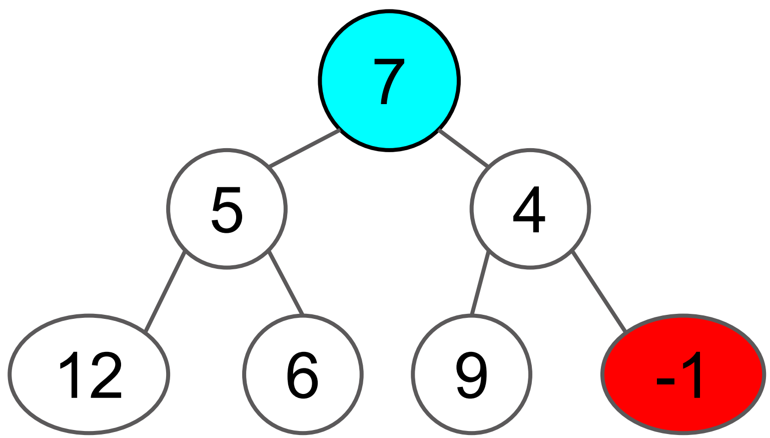
X	-1	5	4	12	6	9	7



Removing from Heaps

Switch first and last elements

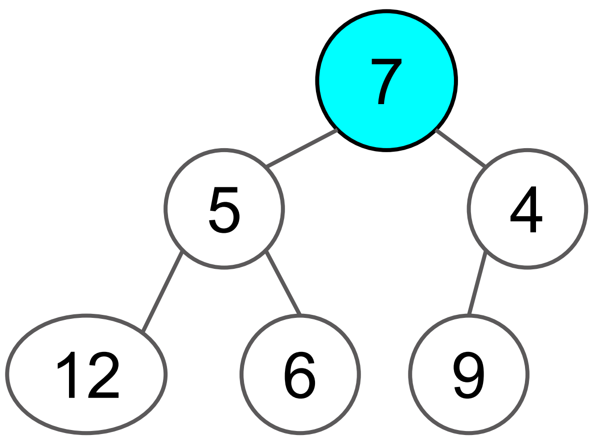
X	7	5	4	12	6	9	-1



Removing from Heaps

pop the last element

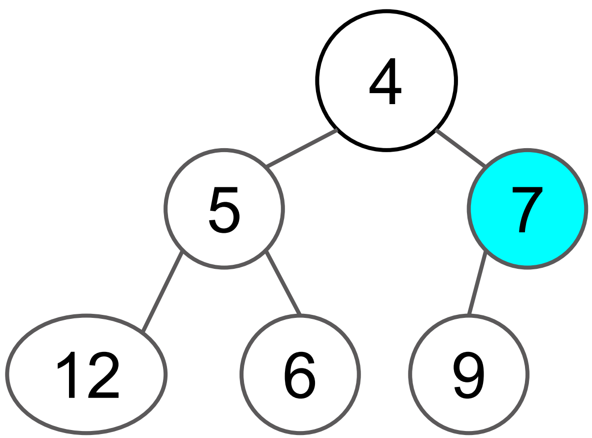
X	7	5	4	12	6	9	



Removing from Heaps

bubble down… and done!

X	4	5	7	12	6	9	



How do we know which way to bubble down?

Heap Complexities

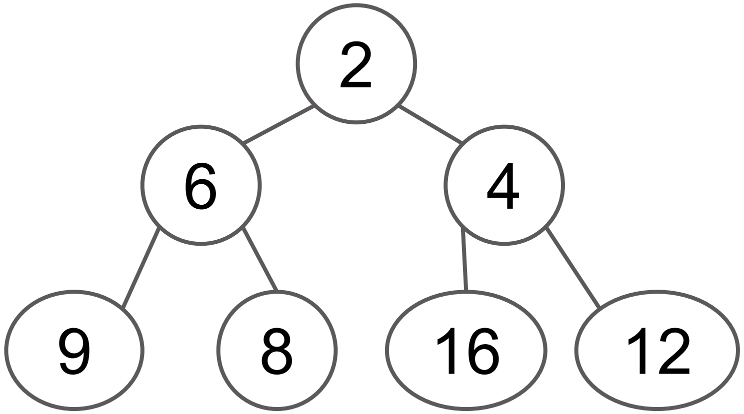
Running Times			
	Binary Heap	Sorted List/Array	Unsorted List/Array
min()	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
insert() (worst case)	$\Theta(logn)^*$	$\Theta(n)$	$\Theta(1)^*$
insert() (best case)	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$
removeMin() (worst case)	$\Theta(logn)$	$\Theta(1)$	$\Theta(n)$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

Heap Practice

Given the following min heap, draw what it looks like after each following consecutive method calls:

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



Heap Practice

After insert(10)

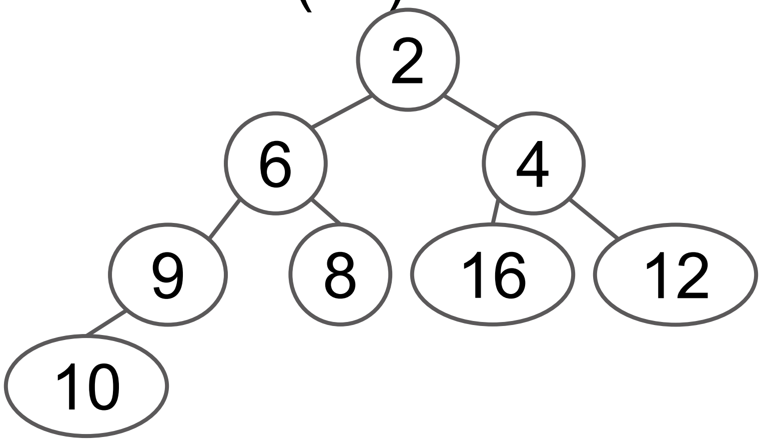
After insert(3)

After removeMin()

After removeMin()

Heap Practice

After insert(10)



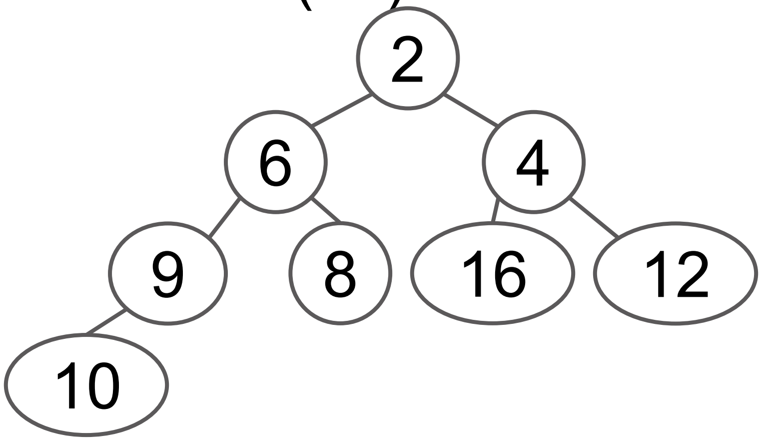
After insert(3)

After removeMin()

After removeMin()

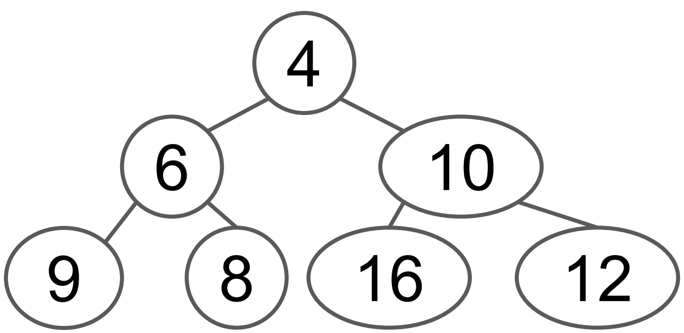
Heap Practice

After insert(10)



After insert(3)

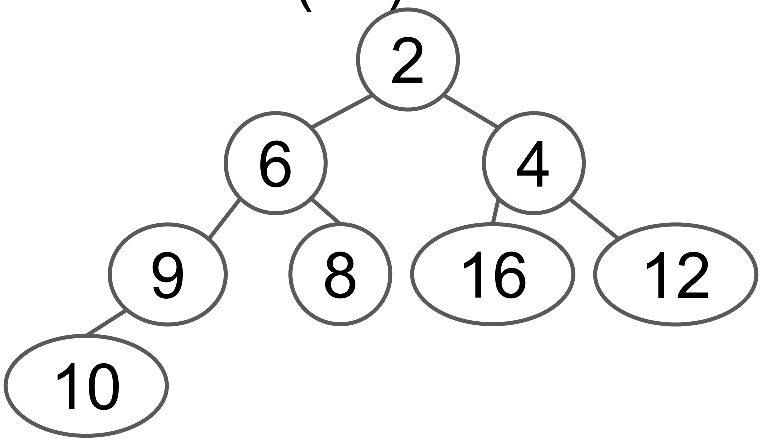
After removeMin()



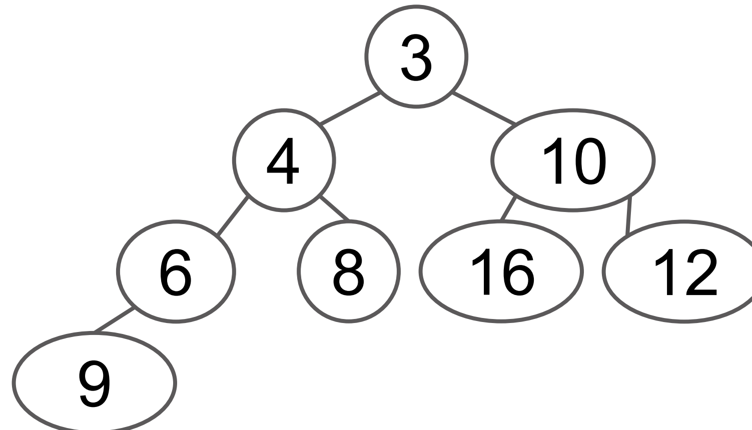
After removeMin()

Heap Practice

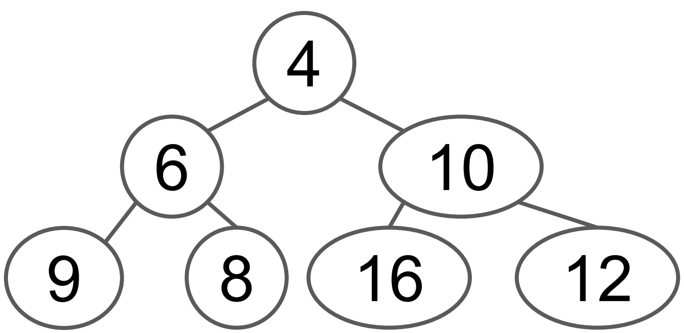
After insert(10)



After insert(3)



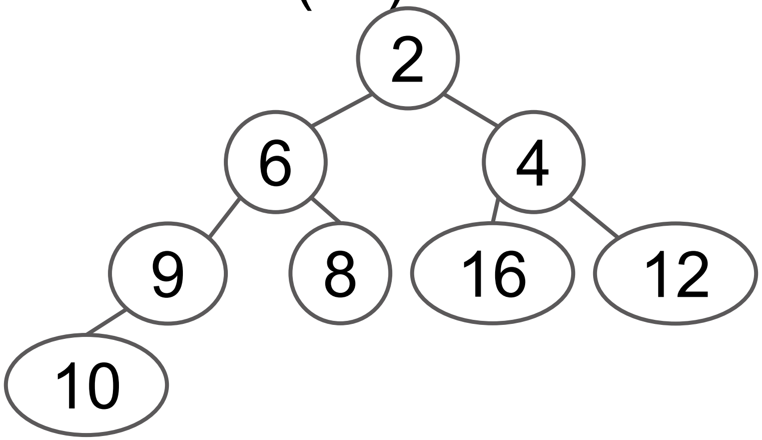
After removeMin()



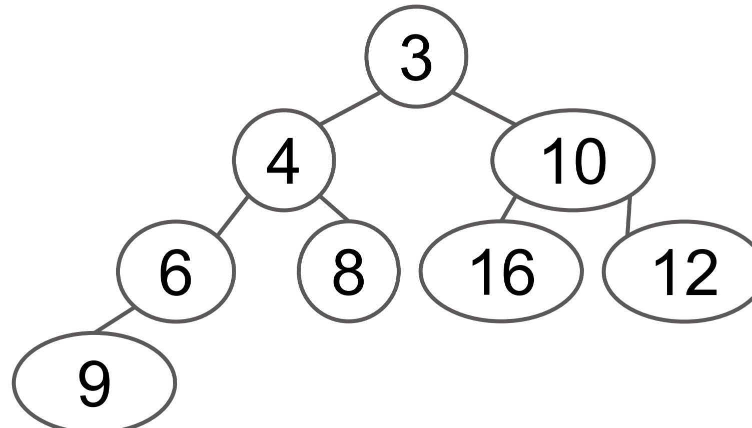
After removeMin()

Heap Practice

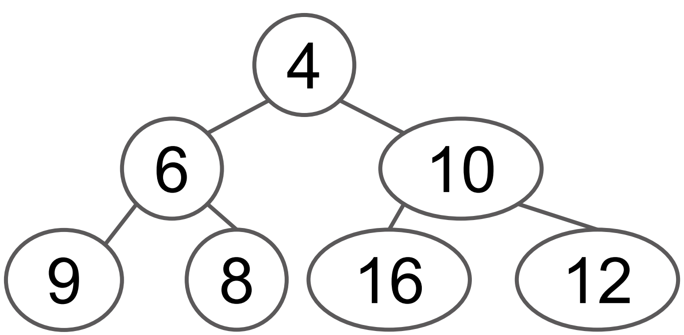
After insert(10)



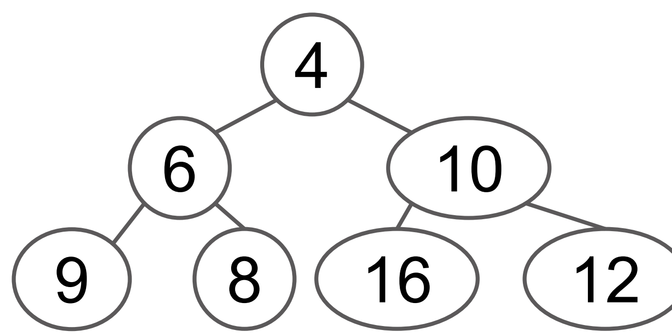
After insert(3)



After removeMin()



After removeMin()



Heaps, a silly question

Write a function **int[] kLargest(int[] a, int k)** that takes an **unsorted integer array** of size **N** and finds the **k** largest elements.

You also know that **k << N**

Heaps, a silly answer

Ans:

1. Build a **min** heap of size **k** using the first k elements.
2. For each element **x** of **a**:
   a. Compare heap.peekMin() and **x**
   b. If x is larger, heap.removeMin(), and heap.insert(x)
3. Output the elements of the heap

Running time: **O(nlogk)**

Heaps, a silly answer

```
int[] kLargest(int[] a, int k){
  Heap bestOfTheBest = new MinHeap();
  for(int i = 0; i < k; i++)
    bestOfTheBest.insert(a[i]);

  for(int i = k; i < a.length; i++){
    if(bestOfTheBest.peekMin() < a[i]){
      bestOfTheBest.removeMin();
      bestOfTheBest.insert(a[i]);
    }
  }

  int[] largest = new int[k];
  for(int i = 0; i < k; i++){
    largest[i] = bestOfTheBest.removeMin();
  }
  return largest;
}
```

Heaps, a better question

Design an efficient data structure that supports the following method calls:

- void insert(int n)
- int getMedian()

Heaps, a better question

Example input = {42, 0, 100}

>>> insert(42);

>>> getMedian();

42

>>> insert(0);

>>> getMedian();

21

>>> insert(100); getMedian();

42

For heaps, you are given the following methods:

- int size()
- int peakMin() or peakMax()
- int removeMin() or removeMax()
- int insert(int num)

Note: an **O(n)** insert is too slow!

Heaps, a better solution

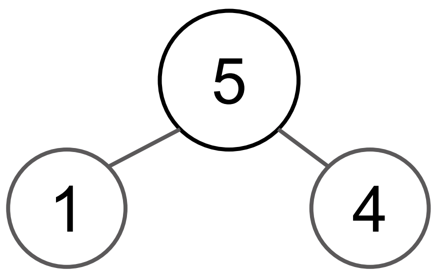
The Idea: split data into two halves

1. Use two heaps, **upper** and **lower.**
   a. **upper** is a **min-heap**, **lower** is a **max-heap**
2. getMedian()
   a. check the sizes of **upper, lower**
   b. If **upper** is larger, return **upper.peekMin()**
   c. If **lower** is larger, return **lower.peekMax()**
   d. If they are the same size, return the average of **upper.peekMin()** and **lower.peekMax()**
3. insert(int n)
   a. If **n** is smaller than **upper.peekMin()**, insert into **lower**
   b. Otherwise, insert into **upper**
   c. rebalance **lower** and **upper**

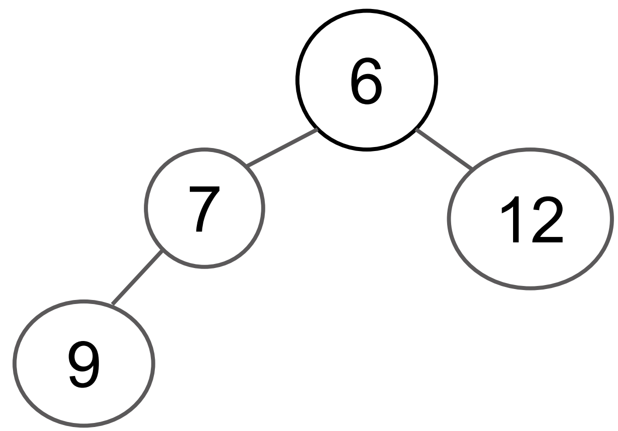
Heaps, a better solution

Data							
	1	4	5	7	12	6	9

lower, max heap



upper, min heap



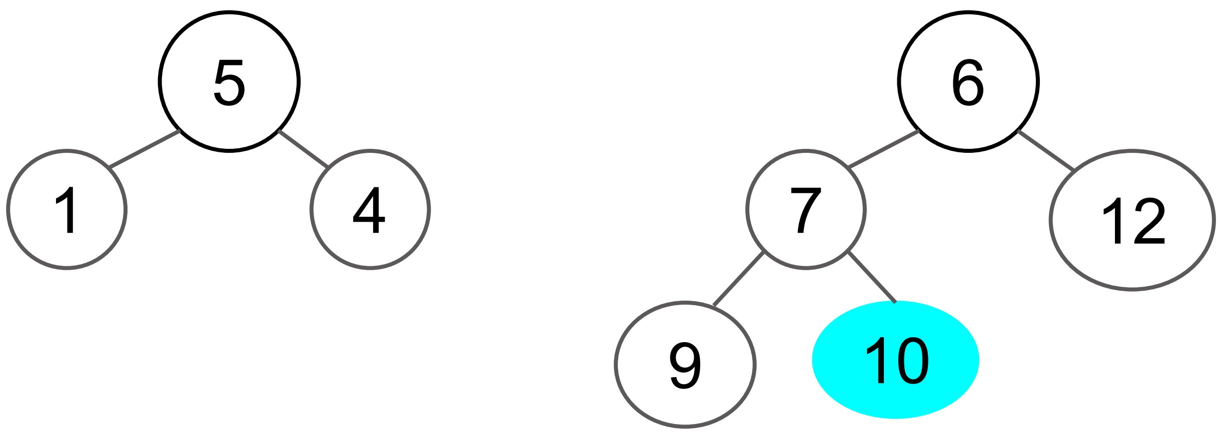
getMedian() returns **6**, the min of the upper heap

Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap

upper, min heap

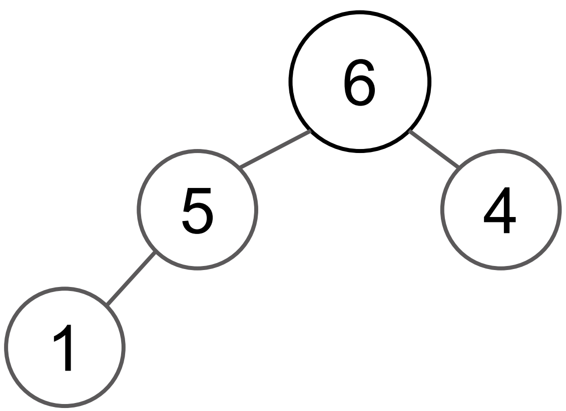


needs rebalancing!

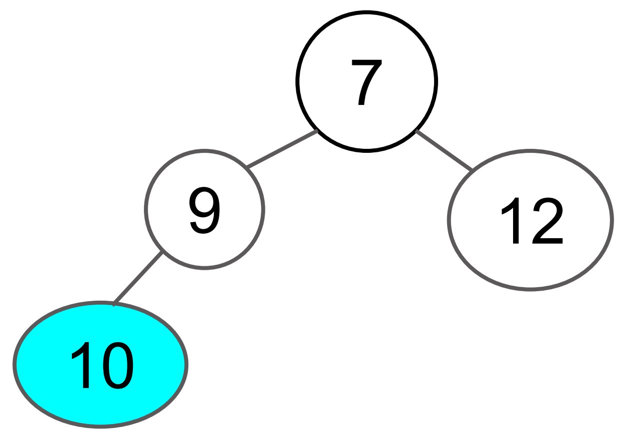
Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap



upper, min heap

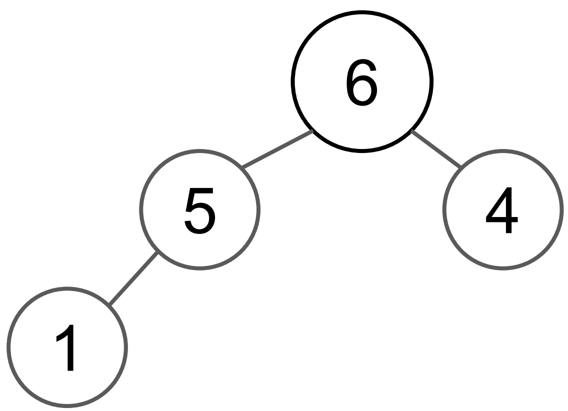


we pop **upper’s** min, and insert it into **lower**

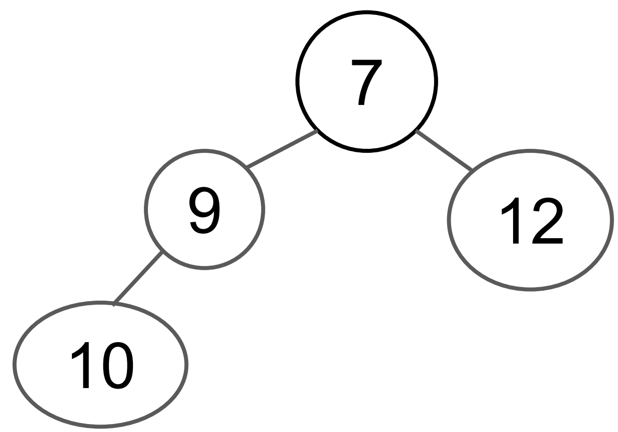
Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap



upper, min heap



getMedian() returns the average between **6, 7 = 6.5**

Heaps, a better solution

```java
public class SexyHeaps{
  private Heap lower, upper;
  public SexyHeaps(){
    lower = new MaxHeap();
    upper = new MinHeap();
  }

  public double getMedian(){
    if(upper.size() > lower.size())
      return upper.peekMin();
    else if(lower.size() > upper.size())
      return lower.peekMax();
    else
      return (upper.peekMin() + lower.peekMax()) / 2.0;
  }
```

Heaps, a better solution

```java

    public void insert(int n){
        if(n < upper.peekMin())
            lower.insert(n);
        else
            upper.insert(n);
        //rebalance
        if(lower.size() >= upper.size() + 2)
            upper.insert(lower.removeMax());
        else if(upper.size() >= lower.size() + 2)
            lower.insert(upper.removeMin());
    }
}

```

Heaps, a better solution

Arrays	Complexity	Total time over **n** inserts/getMedians
insert	O(1)	O(n)
getMedian	O(nlogn) for sorting more sophisticated algorithms can yield O(n)	O(n^2)

SexyHeaps	Complexity	Total time over **n** inserts/getMedians
insert	O(logn)	O(nlogn)
getMedian	O(logn)	O(nlogn)

Sorting

Insertion Sort

Given an empty sequence of outputs S and an unsorted sequence of inputs L:

for (each item x in L):

    add x into S such that S is sorted

Running time is $O(N^2)$

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

{3, 5, 8, 1, 4, 2, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

{3, 5, 8, 1, 4, 2, 0}

{1, 3, 5, 8, 4, 2, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

{3, 5, 8, 1, 4, 2, 0}

{1, 3, 5, 8, 4, 2, 0}

{1, 3, 4, 5, 8, 2, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

{3, 5, 8, 1, 4, 2, 0}

{1, 3, 5, 8, 4, 2, 0}

{1, 3, 4, 5, 8, 2, 0}

{1, 2, 3, 4, 5, 8, 0}

Insertion Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1, 4, 2, 0}

{3, 5, 8, 1, 4, 2, 0}

{1, 3, 5, 8, 4, 2, 0}

{1, 3, 4, 5, 8, 2, 0}

{1, 2, 3, 4, 5, 8, 0}

{0, 1, 2, 3, 4, 5, 8}

Shell's Sort

Improvement on insertion sort

Decreases inversions

Sort subsequences with elements $2^k - 1$ apart

Sort subsequences with elements $2^{k-1}-1$ apart

…

Sort subsequences with elements 1 apart

Running time for this spacing is $\Theta(N^{1.5})$

Shell’s Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell’s sort on arr, with k = 2:

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

Sort 2^k-1 = 3 away:

{0, 8, 5, 1, 4, 2, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

Sort 2^k-1 = 3 away:

{0, 8, 5, 1, 4, 2, 3}

{0, 4, 5, 1, 8, 2, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

Sort 2^k-1 = 3 away:

{0, 8, 5, 1, 4, 2, 3}

{0, 4, 5, 1, 8, 2, 3}

{0, 4, 2, 1, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort $2^{k-1}-1 = 1$ away (ordinary insertion sort)

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort $2^{k-1}-1 = 1$ away (ordinary insertion sort)

{0, 4, 2, 1, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort 2^{k-1}-1 = 1 away (ordinary insertion sort)

  {0, 4, 2, 1, 8, 5, 3}

  {0, 4, 2, 1, 8, 5, 3}

Shell’s Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell’s sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort 2^{k-1}-1 = 1 away (ordinary insertion sort)

- {0, 4, 2, 1, 8, 5, 3}
- {0, 4, 2, 1, 8, 5, 3}
- {0, 2, 4, 1, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort 2^{k-1}-1 = 1 away (ordinary insertion sort)

{0, 4, 2, 1, 8, 5, 3}

{0, 4, 2, 1, 8, 5, 3}

{0, 2, 4, 1, 8, 5, 3}

{0, 1, 2, 4, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort 2^{k-1}-1 = 1 away (ordinary insertion sort)

{0, 4, 2, 1, 8, 5, 3}

{0, 4, 2, 1, 8, 5, 3}

{0, 2, 4, 1, 8, 5, 3}

{0, 1, 2, 4, 8, 5, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort $2^{k-1}-1 = 1$ away (ordinary insertion sort)

- {0, 4, 2, 1, 8, 5, 3}
- {0, 4, 2, 1, 8, 5, 3}
- {0, 2, 4, 1, 8, 5, 3}
- {0, 1, 2, 4, 8, 5, 3}
- {0, 1, 2, 4, 5, 8, 3}

Shell's Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a shell's sort on arr, with k = 2:

{0, 4, 2, 1, 8, 5, 3}

Sort $2^{k-1}-1$ = 1 away (ordinary insertion sort)

{0, 4, 2, 1, 8, 5, 3}

{0, 4, 2, 1, 8, 5, 3}

{0, 2, 4, 1, 8, 5, 3}

{0, 1, 2, 4, 8, 5, 3}

{0, 1, 2, 4, 5, 8, 3} → {0, 1, 2, 3, 4, 5, 8}

Selection Sort

Given an unsorted list L and an empty list S…

while L is not empty:
    Move the smallest element of L to S's end

Running time is O(N^2)

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place selection sort on arr:

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

{0, 1, 2, 3, 8, 5, 4}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 8, 5, 4}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 4, 8, 5}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 4, 8, 5}

{0, 1, 2, 3, 4, 5, 8}

Selection Sort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform an in-place sort on arr:

{0, 3, 8, 5, 1, 4, 2}

{0, 1, 3, 8, 5, 4, 2}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 8, 5, 4}

{0, 1, 2, 3, 4, 8, 5}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 4, 5, 8}

Heapsort

Similar idea as selection sort, but convert the unsorted list L as a heap H…

while H is not empty:

    x = H.remove_first()

    append x to S

N remove_first() operations on H…

    Running Time: O(*N* log*N*)

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

First heapify arr (in this example a min-heap):

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

First heapify arr (in this example a min-heap):

{3, 8, 0, 1, 4, 2, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

First heapify arr (in this example a min-heap):

{3, 8, 0, 1, 4, 2, 5}

{3, 1, 0, 8, 4, 2, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

First heapify arr (in this example a min-heap):

{3, 8, 0, 1, 4, 2, 5}

{3, 1, 0, 8, 4, 2, 5}

{0, 1, 3, 8, 4, 2, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

First heapify arr (in this example a min-heap):

{3, 8, 0, 1, 4, 2, 5}

{3, 1, 0, 8, 4, 2, 5}

{0, 1, 3, 8, 4, 2, 5}

{0, 1, 2, 8, 4, 3, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 8, 4, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 8, 4, 5}

{0, 1, 2, 3, 4, 8, 5}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 8, 4, 5}

{0, 1, 2, 3, 4, 8, 5}

{0, 1, 2, 3, 4, 5, 8}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 8, 4, 5}

{0, 1, 2, 3, 4, 8, 5}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 4, 5, 8}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}

{0, 1, 4, 2, 8, 5, 3}

{0, 1, 3, 4, 2, 8, 5}

{0, 1, 2, 4, 3, 8, 5}

{0, 1, 2, 5, 4, 3, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 8, 4, 5}

{0, 1, 2, 3, 4, 8, 5}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 4, 5, 8}

{0, 1, 2, 3, 4, 5, 8}

Heapsort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a heapsort on arr:

After heapifying: {0, 1, 2, 8, 4, 3, 5}

{0, 5, 1, 2, 8, 4, 3}	{0, 1, 2, 3, 8, 4, 5}
{0, 1, 4, 2, 8, 5, 3}	{0, 1, 2, 3, 4, 8, 5}
{0, 1, 3, 4, 2, 8, 5}	{0, 1, 2, 3, 4, 5, 8}
{0, 1, 2, 4, 3, 8, 5}	{0, 1, 2, 3, 4, 5, 8}
{0, 1, 2, 5, 4, 3, 8}	{0, 1, 2, 3, 4, 5, 8}
{0, 1, 2, 3, 4, 5, 8}	{0, 1, 2, 3, 4, 5, 8}

Mergesort

Begin with an unsorted list S of N items.

Divide into two lists S_1, S_2 with N/2 items.

Sort S_1 and S_2 recursively to yield sorted lists
A_1 and A_2

Merge A_1 and A_2

Running time is $O(N \log N)$

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1}

{4, 2, 0}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

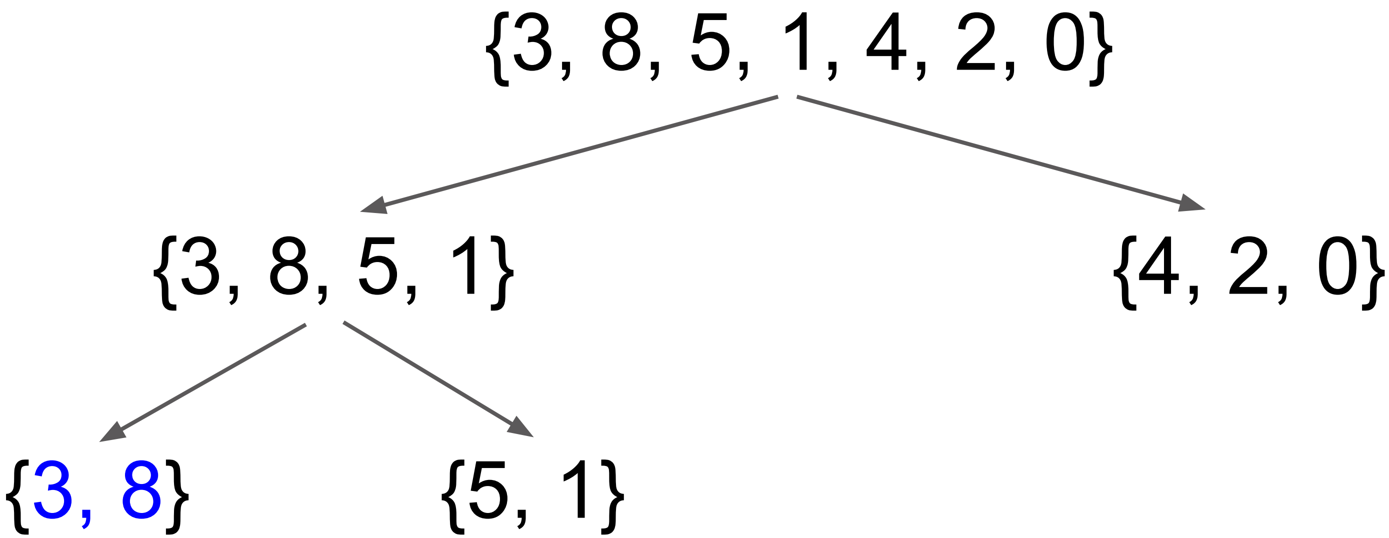
{3, 8, 5, 1}　　　　{4, 2, 0}

{3, 8}　　{5, 1}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:



Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

{3, 8, 5, 1}　　{4, 2, 0}

{3, 8}　　{1, 5}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

{1, 3, 5, 8}   {4, 2, 0}

{3, 8}   {1, 5}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

{1, 3, 5, 8} {4, 2, 0}

{3, 8} {1, 5} {4, 2} {0}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{3, 8, 5, 1, 4, 2, 0}

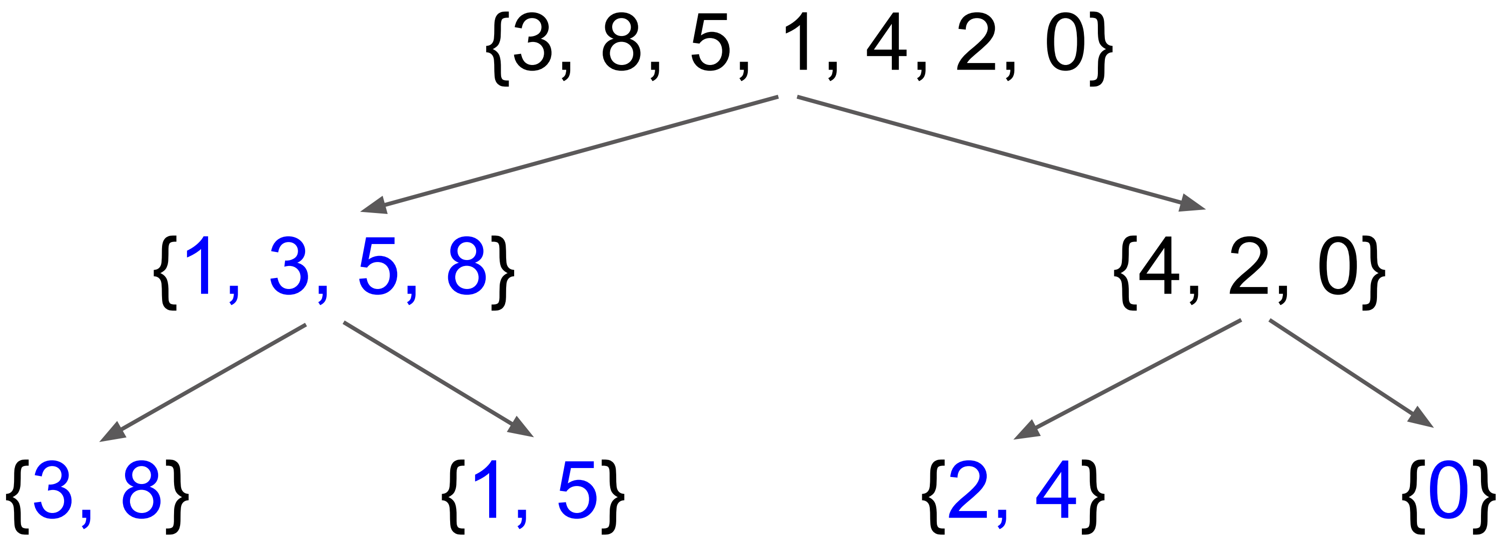
{1, 3, 5, 8} {4, 2, 0}

{3, 8} {1, 5} {2, 4} {0}

Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

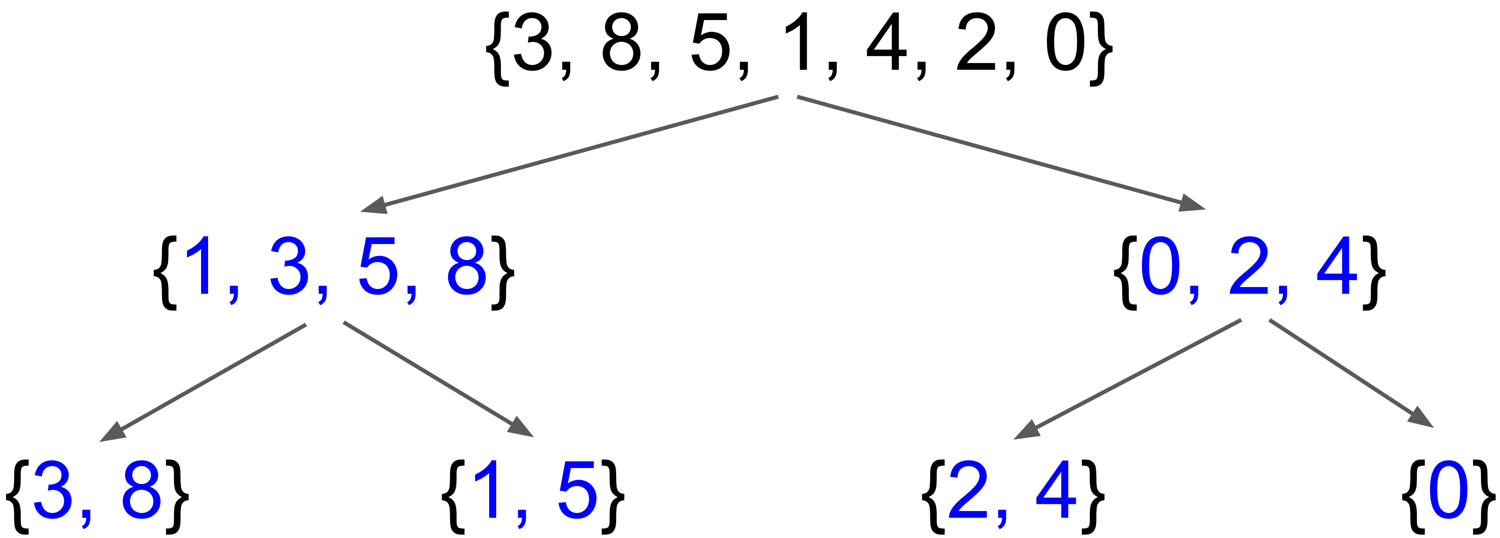
Perform a mergesort on arr:



Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:



Mergesort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a mergesort on arr:

{0, 1, 2, 3, 4, 5, 8}

{1, 3, 5, 8} {0, 2, 4}

{3, 8} {1, 5} {2, 4} {0}

Quicksort

Begin with an unsorted list L of N items

Choose a pivot v

Divide L into lists L_1 and L_2 not containing v...

$L_1 = \{x \text{ in } L \mid x \leq v\}$, and $L_2 = \{x \text{ in } L \mid x > v\}$

Sort L_1, L_2 recursively to yield S_1, S_2

Merge S_1, S_2 to yield sorted list S

Runtime: with a bad pivot $\Theta(N^2)$

with a good pivot $\Theta(N \log N)$

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

Choose a pivot!

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

Choose a pivot!

Minimum?

Maximum?

Random?

Median of the first, last, and middle items?

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

Choose a pivot!

Minimum? $\Theta(N^2)$

Maximum? $\Theta(N^2)$

Random? $\Theta(N \log N)$

Median of the first, last, and middle items?

$\Theta(N \log N)$

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

In this example, we will use median of the first, last and middle items.

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{3, 8, 5, 1, 4, 2, 0}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{3, 8, 5, 1, 4, 2, 0}

{0, 1, 3, 8, 5, 4, 2}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{3, 8, 5, 4, 2}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{3, 8, 5, 4, 2}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{3, 8, 5, 4, 2}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

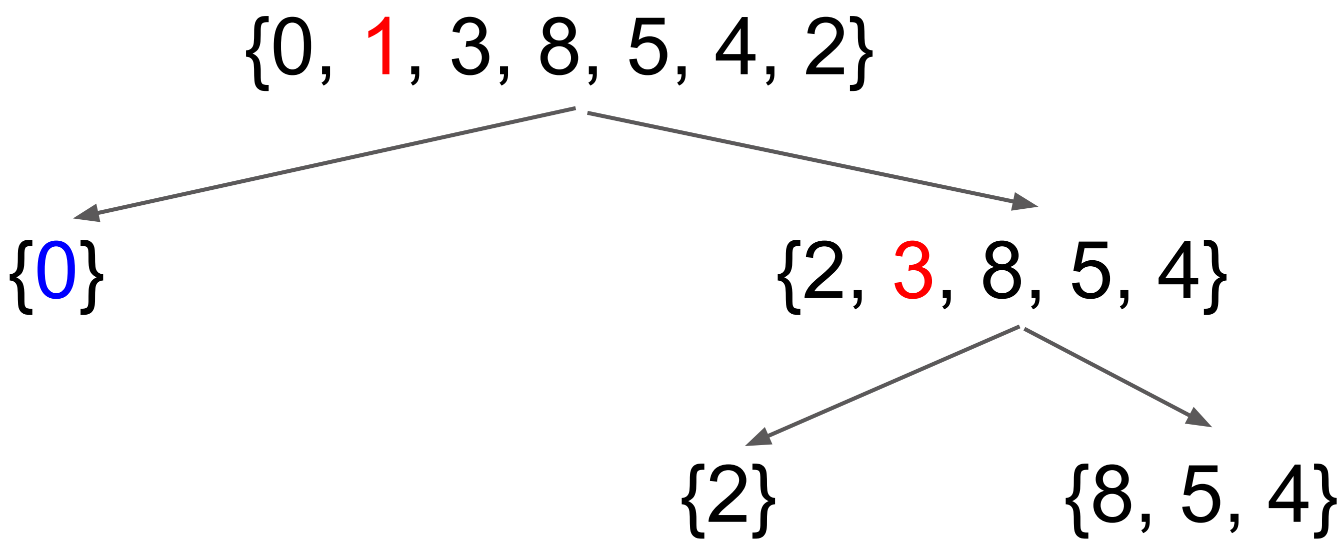
{3, 8, 5, 4, 2}

{2, 3, 8, 5, 4}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

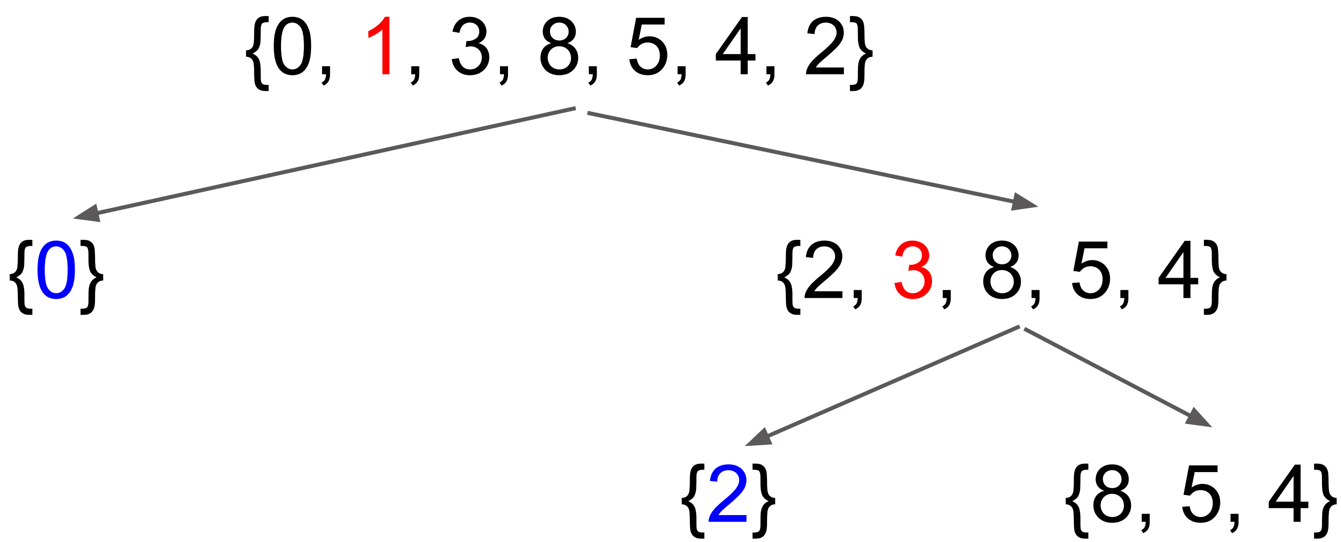
Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{2, 3, 8, 5, 4}

{2}

{8, 5, 4}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{2, 3, 8, 5, 4}

{2}

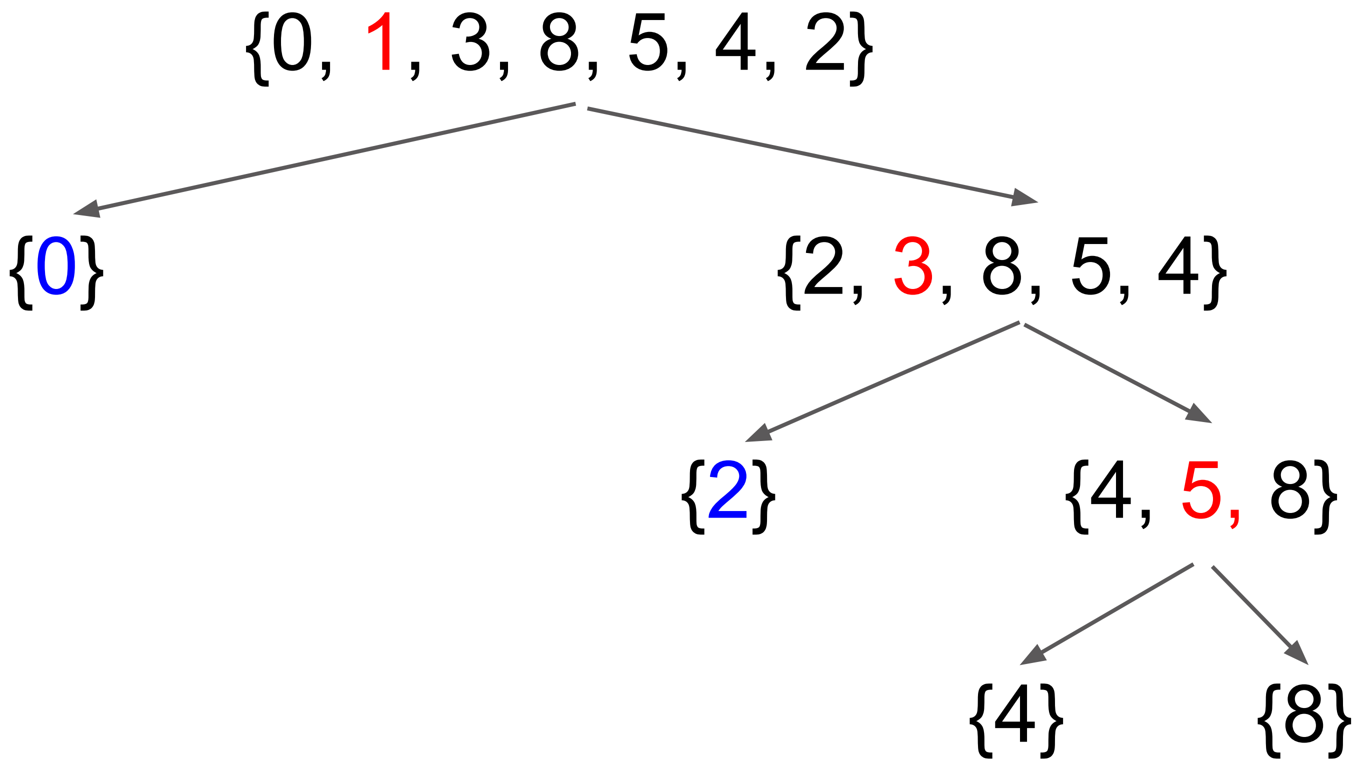
{8, 5, 4}

{4, 5, 8}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

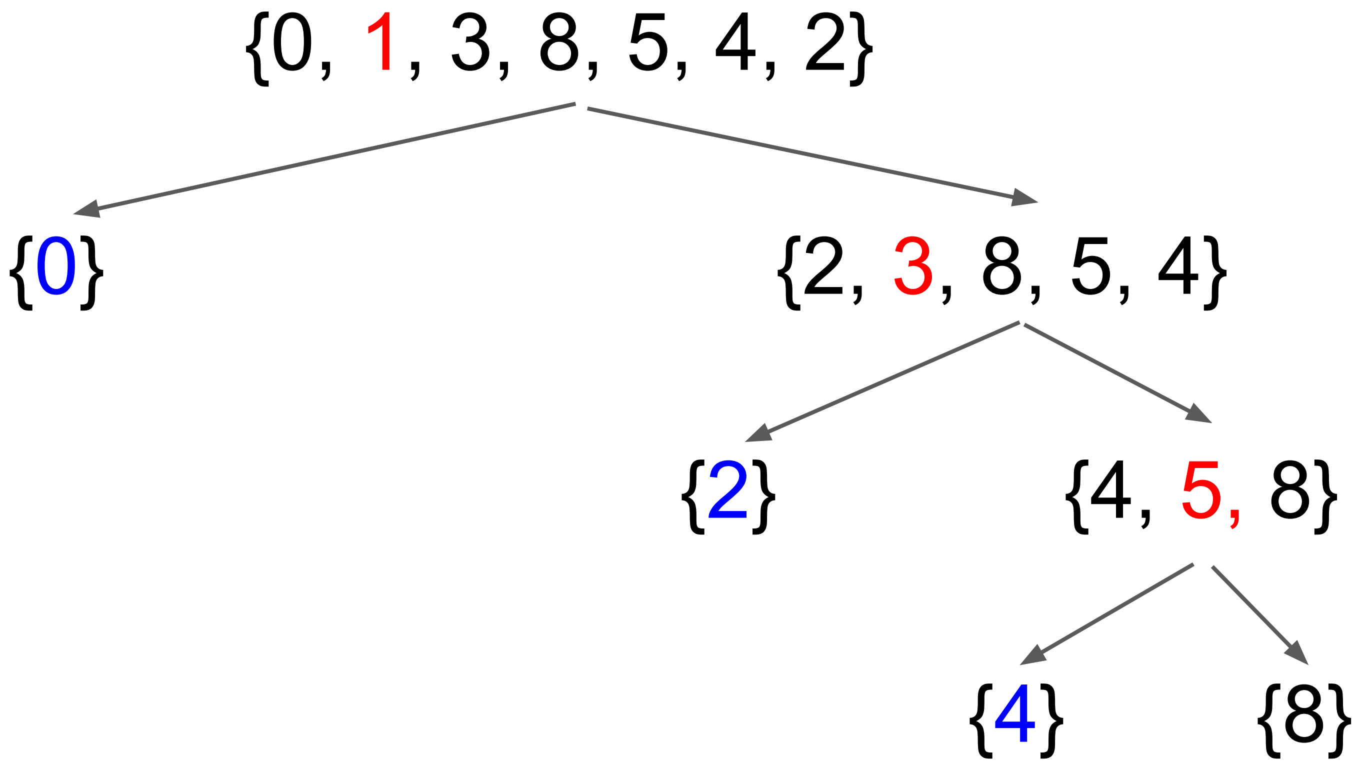
Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

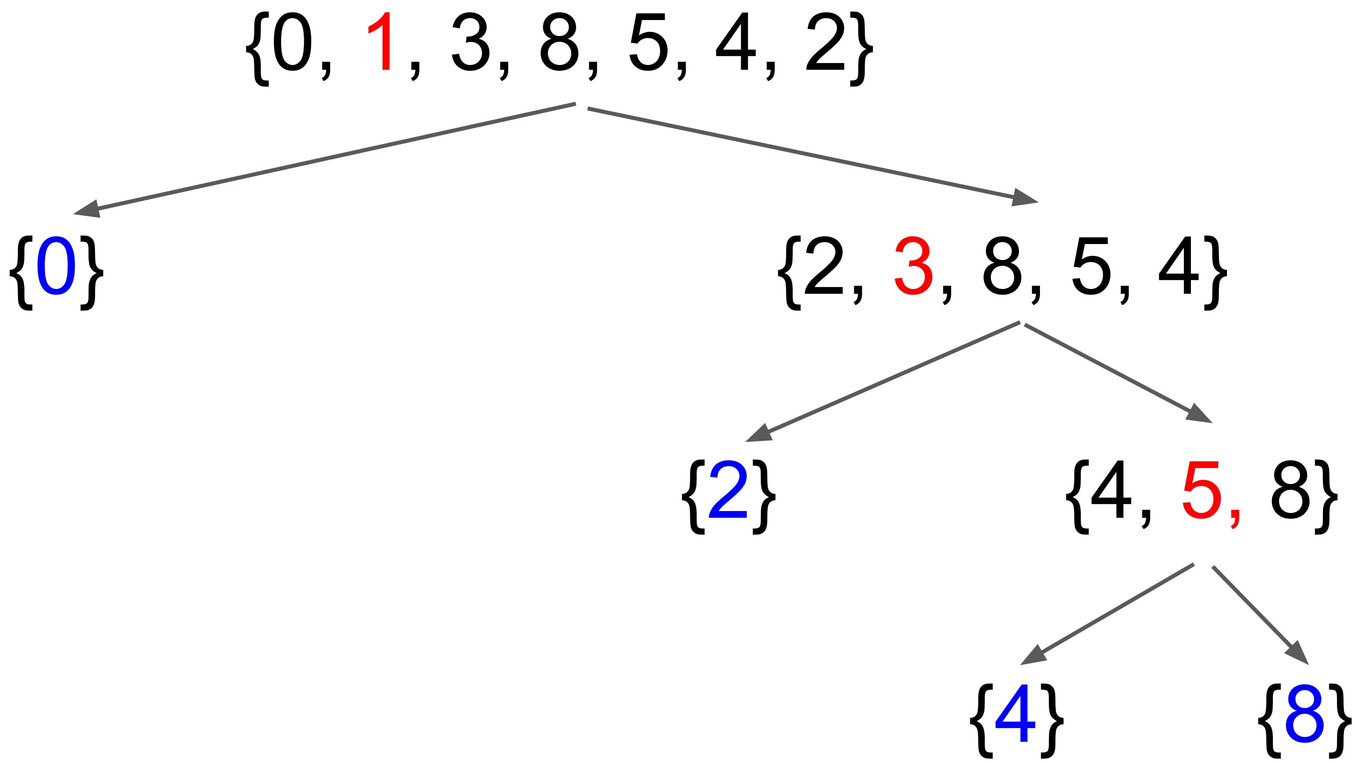
Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:

{0, 1, 3, 8, 5, 4, 2}

{0}

{2, 3, 8, 5, 4}

{2}

{4, 5, 8}

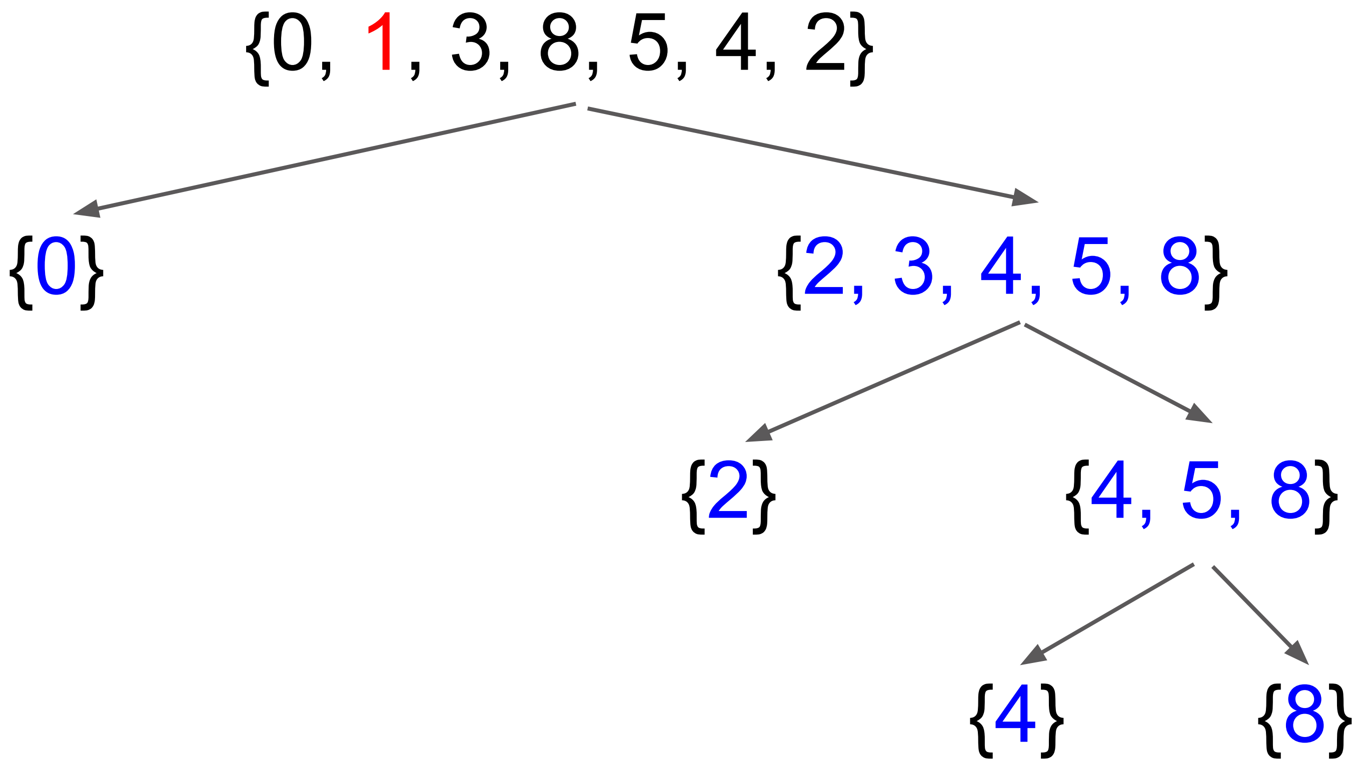
{4}

{8}

Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

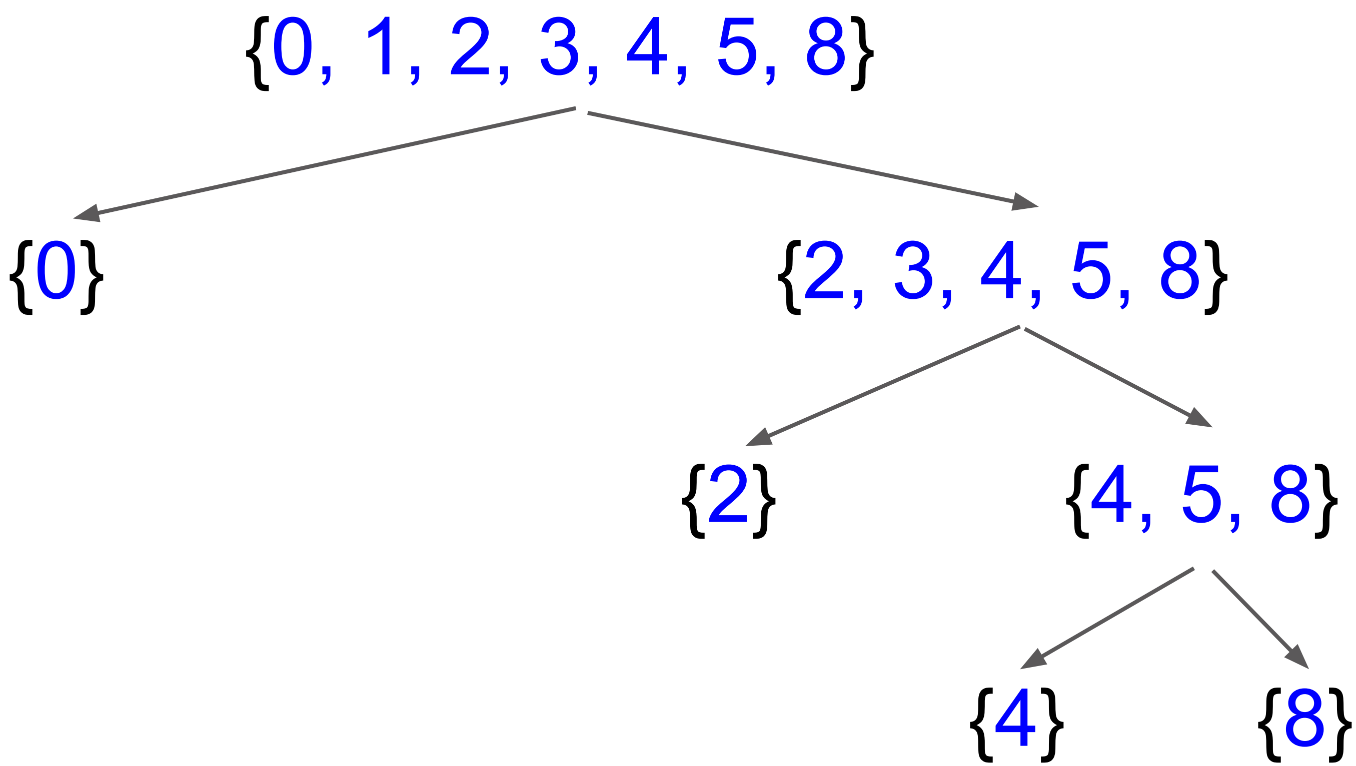
Perform a quicksort on arr:



Quicksort (example)

int[] arr = {3, 8, 5, 1, 4, 2, 0}

Perform a quicksort on arr:



Counting Sort

Given an array L of integers

Iterate through L to find the counts of each integer in L

Scan the counts array to produce an array A of running sums LESS THAN the current value

Reconstruct the sorted array by iterating through A and adding keys to S in sorted order.

Running Time: Linear

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}

Perform a counting sort on arr:

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}

Perform a counting sort on arr:

Construct the counts array...

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}

Perform a counting sort on arr:

Construct the counts array…

{0, 3, 0, 3, 4, 7, 1, 1, 3, 3}

0 1 2 3 4 5 6 7 8 9

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}

Perform a counting sort on arr:

{0, 3, 0, 3, 4, 7, 1, 1, 3, 3}

0 1 2 3 4 5 6 7 8 9

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

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}

Perform a counting sort on arr:

{0,	3,	0,	3,	4,	7,	1,	1,	3,	3}
0	1	2	3	4	5	6	7	8	9

Scan counts array so that counts[i] contains number of keys less than i…

{0,	0,	3,	3,	6,	10,	17,	18,	19,	22}
0	1	2	3	4	5	6	7	8	9

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}

Perform a counting sort on arr:

{0, 0, 3, 3, 6, 10, 17, 18, 19, 22}

0 1 2 3 4 5 6 7 8 9

Construct the sorted array S by walking through the counts and copying each item into S...

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}

Perform a counting sort on arr:

{0,	0,	3,	3,	6,	10,	17,	18,	19,	22}
0	1	2	3	4	5	6	7	8	9

Construct the sorted array S by walking through the counts and copying each item into S…

{1, 1, 1, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6, 7, 8, 8, 8, 9, 9, 9}

Radix Sort

Idea: Sort one radix at a time, from least significant to most significant

Uses counting sort

The stability of counting sort guarantees the correctness of radix sort.

Running time: Linear

Radix Sort (example)

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

Perform a radix sort on arr with q = 10 buckets.

Radix Sort (example)

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

Perform a radix sort on arr with q = 10 buckets.

Sort by Least significant bit…

Radix Sort (example)

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

Perform a radix sort on arr with q = 10 buckets.

Sort by least significant bit…

				624	355			898	
390	611		923	874	575			8	
299									
290	191		63	134	975		907	758	199
0	1	2	3	4	5	6	7	8	9

Radix Sort (example)

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

Perform a radix sort on arr with q = 10 buckets.

Sort by least significant bit…

{290, 390, 611, 191, 63, 923, 134, 874, 624, 975, 575, 355, 907, 758, 8, 898, 199, 299}

Radix Sort (example)

{290, 390, 611, 191, 63, 923, 134, 874, 624, 975, 575, 355, 907, 758, 8, 898, 199, 299}

									299
									199
									898
							575		191
8		624			758		975		390
907	611	923	134		355	63	874		290
0	1	2	3	4	5	6	7	8	9

Radix Sort (example)

{907, 8, 611, 923, 624, 134, 355, 758, 63, 874, 975, 575, 290, 390, 191, 898, 199, 299}

									299
									199
									898
							575		191
8		624			758		975		390
907	611	923	134		355	63	874		290
0	1	2	3	4	5	6	7	8	9

Radix Sort (example)

{907, 8, 611, 923, 624, 134, 355, 758, 63, 874, 975, 575, 290, 390, 191, 898, 199, 299}

	199								975
63	191	299	390			624		898	923
8	134	290	355		575	611	758	874	
907									
0	1	2	3	4	5	6	7	8	9

Radix Sort (example)

{907, 8, 611, 923, 624, 134, 355, 758, 63, 874, 975, 575, 290, 390, 191, 898, 199, 299}

	199								975
63	191	299	390			624		898	923
8	134	290	355		575	611	758	874	
907									
0	1	2	3	4	5	6	7	8	9

{8, 63, 134, 191, 199, 290, 299, 355, 390, 575, 611, 624, 758, 874, 898, 907, 923, 975}

Sorting questions

T/F: The tightest possible lower bound for comparison based sorting on an array A of length N is O(*N*)

Sorting questions

T/F: The tightest possible lower bound for comparison based sorting on an array A of length N is O(N)

Answer: ***False***

There are N! possible ways A can be scrambled. So if there are k comparisons, then

$$2^k > N!$$

Thus there are $\Omega(\log N!) = \Omega(N \log N)$ comparisons

Sorting questions

What is the best way to sort a million 32-bit integers?

Sorting questions

What is the best way to sort a million 32-bit integers?

Answer: Radix sort (Counting sort also fine).

Sorting questions

What is the best way to sort a million 32-bit integers?

Answer: Radix sort (Counting sort also fine).

NOT bubble sort (Thanks Obama!)

Sorting questions

T/F: Is the bound for the running time of Shell's Sort always $\Theta(N^{1.5})$?

Sorting questions

T/F: Is the bound for the running time of Shell's Sort always $\Theta(N^{1.5})$?

Answer: ***False***

Depends on the gap sequence used!

For example, with gap sequence consisting of successive numbers of the form $2^p 3^q$, Shell's Sort has a bound of $\Theta(N \log^2 N)$

Sorting questions

Suppose we are given a mostly sorted list L of length N, where exactly 7 unknown items are swapped. What is the most efficient method to sort L?

Sorting questions

Suppose we are given a mostly sorted list L of length N, where exactly 7 unknown items are swapped. What is the most efficient method to sort L?

Answer: There are only 7 inversions, so an insertion sort would run in $\Theta(N)$.

Sorting questions

Adapted from Spring 2005 final

Suppose you want to radix sort an array *A* of Java (32-bit) signed integers. Why would you never choose to use exactly q = 512 buckets?

Sorting questions

Suppose you want to radix sort an array *A* of Java (32-bit) signed integers. Why would you never choose to use exactly q = 512 buckets?

Answer: Integer values can be at most 2^{31}-1, and q = 512 = 2^9. So we need to make 4 passes to completely sort *A*.

But if q = 256 = 2^8, we still only need to make 4 passes to completely sort *A*. So q = 512 buckets needs double the memory for the same speed

Sorting questions

Is mergesort a stable sort?

Sorting questions

Is mergesort a stable sort?

Answer: Yes.

Sorting questions

Is quicksort a stable sort?

Sorting questions

Is quicksort a stable sort?

Answer: In general quicksort is not stable.

Stable implementations do exist at the cost of suboptimal memory and speed.

Sorting questions

Can mergesort be implemented in place?

Sorting questions

Can mergesort be implemented in place?

Answer: Kind of.

In-place mergesort exists, but the merging step is complicated and has to be done carefully (sloppy implementation can create a quadratic sort).

Sorting questions

Can quicksort be implemented in place?

Sorting questions

Can quicksort be implemented in place?

Answer: Yes, with no cost to speed.

Sorting questions

T/F: Mergesort uses more memory than quicksort.

Sorting questions

T/F: Mergesort uses more memory than quicksort.

Answer: ***True***

Mergesort uses O(N) memory, while quicksort uses O(log N) memory.

Hashing

Hashing

Why do we need chaining?

Hashing

Why do we need chaining?

We need chaining to deal with collisions

Hashing

Why do we need chaining?

We need chaining to deal with collisions

What is a good hash function?

Hashing

Why do we need chaining?

We need chaining to deal with collisions

What is a good hash function?

One that distributes objects into buckets uniformly and randomly and computes quickly

Hashing

Why do we need chaining?

We need chaining to deal with collisions

What is a good hash function?

One that distributes objects into buckets uniformly and randomly and computes quickly

Suppose we have two objects, A and B. If A.hashCode() == B.hashCode(), does that mean A == B?

Hashing

Why do we need chaining?

We need chaining to deal with collisions

What is a good hash function?

One that distributes objects into buckets uniformly and randomly and computes quickly

Suppose we have two objects, A and B. If A.hashCode() == B.hashCode(), does that mean A == B?

No, A and B can be different objects. This is how a collision happens. Remember A.equals(B) → A.hashCode() == B.hashCode() but not the other way around.

Hashing

If two objects have different hashCodes, does that mean they will be in different buckets?

Hashing

If two objects have different hashCodes, does that mean they will be in different buckets?

No, in Java, the process for hashing works as follows:
Object -> hashCode() -> Compression Function -> Bucket
This means that even though our hashCodes may be different
we have a limited number of buckets so the compression function
will put several hashCodes into the same bucket.

Hashing

What is a good hash function for an array of integers with a string representation already implemented?

Hashing

What is a good hash function for an array of integers with a string representation already implemented?

Use the hashCode of the string representation. Since the String's hashcode that comes with Java is known to be good, we can save ourselves extra work by using it.

Hashing

What is a good hash function for an array of integers with a string representation already implemented?

Use the hashCode of the string representation. Since the String's hashcode that comes with Java is known to be good, we can save ourselves extra work by using it.

If we change the hashCode of an object stored in a HashSet, is it still reachable?

Hashing

What is a good hash function for an array of integers with a string representation already implemented?

Use the hashCode of the string representation. Since the String's hashcode that comes with Java is known to be good, we can save ourselves extra work by using it.

If we change the hashCode of an object stored in a HashSet, is it still reachable?

It depends. If the hashCode still hashes into the same bucket, the object may still be reachable.

Hashing

What happens if we had the following hashCode?

```
public int hashCode() {
    return 0;
}
```

Hashing

What happens if we had the following hashCode?

```
public int hashCode() {
    return 0;
}
```

All our objects will hash into the same bucket. This would mean that our runtime to put and get items is O(n), where n is the number of items already added.

Hashing

What happens if we had the following hashCode?

```
public int hashCode() {
    return 0;
}
```

All our objects will hash into the same bucket. This would mean that our runtime to put and get items is O(n), where n is the number of items already added.

Hashing

What happens if we had the following equals method? Assume in this case, the hashCode works properly and distributes our objects such that collisions are minimized.

```
public boolean equals(Object o) {
   return true;
}
```

Hashing

What happens if we had the following equals method? Assume in this case, the hashCode works properly and distributes our objects such that collisions are minimized.

```
public boolean equals(Object o) {
    return true;
}
```

Our hash table would only contain one object in every bucket since the put method checks if a key already exists in the bucket and overwrites the entry if it does.