

# CS106B Final Review

Ashley Taylor

some slides adapted from Anton Apostolatos

# Final Review Session Overview

- Logistics
- Sorting
- Linked Lists
- Hashing
- Binary Heap
- Trees
- Binary Search Trees
- Graphs
- Inheritance

# Logistics

# Final Logistics

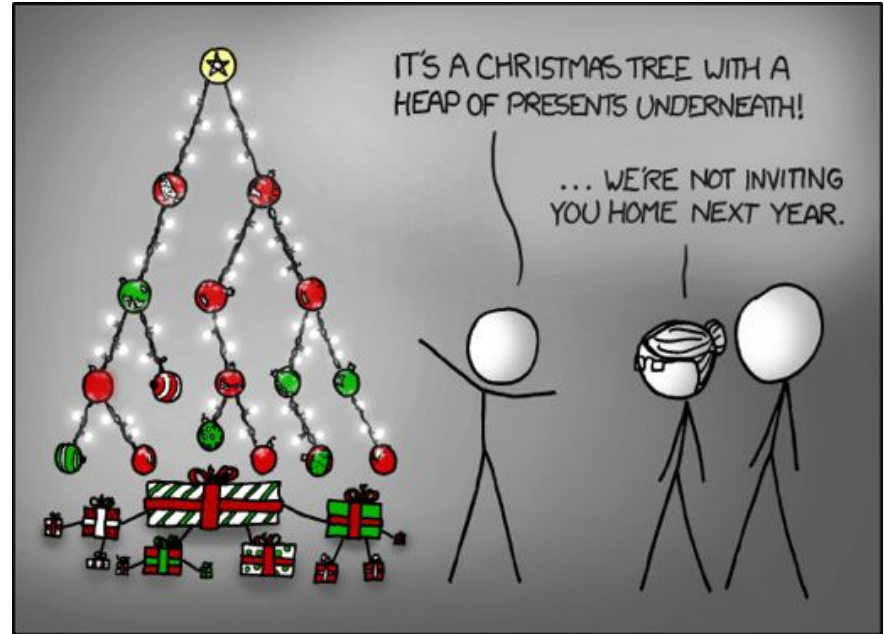
- Monday, December 11, 8:30-11:30am
- Memorial Auditorium
- Open books, one two-sided page of notes
- Pencils and pens accepted (please write darkly and clearly)

# Basic exam tips

- Practice by **writing** answers to the practice exam/CSBS questions
- If your answer is outside the designated answer area, please indicate where to look (in the designated answer space) so we don't miss it

# What's on the final

- Everything is fair game!
- Will be weighted to second half of the course



# Sorting

# Sorting: Tips

Make sure you understand the differences between the algorithms (especially in terms of run-time)

Make sure you can recognize the code when you see it

Practice tracing through how each algorithm will sort a list (when given the code)



# Sorting: Insertion Sort

Works by inserting one element at a time into the sorted list

Sorted list starts as a list of size 1 with the first element in the list, then for all the other elements in the list, we insert each into its proper place in the sorted list

Runtime:  $O(N^2)$

# Sorting: Selection Sort

Find the smallest element in the list, and swap it with the leftmost element in the list

Continue by looking at the remaining (unsorted elements)

Runtime:  $O(N^2)$

# Sorting: Heap Sort

Load all the elements into a heap priority queue, then dequeue one-by-one

Runtime:  $O(N \log N)$

# Sorting: Merge Sort

Split the array into two smaller arrays

Base case: an array of size 1 is trivially sorted

Recursive step: split into two arrays of size  $(N/2)$  that we call mergeSort on, then merge the two sorted arrays together

Runtime:  $O(N \log N)$

# Sorting: QuickSort

Choose an element as a “pivot element”

For all the other elements in the array, split them to the left of the pivot (if they're smaller than the pivot) or right (if they're greater). The pivot is now in the correct spot

Recurse on the two arrays on either side of the pivot

Expected:  $O(N \log N)$

Worst case: (e.g. picking the smallest element each time)  $O(N^2)$

# LinkedLists

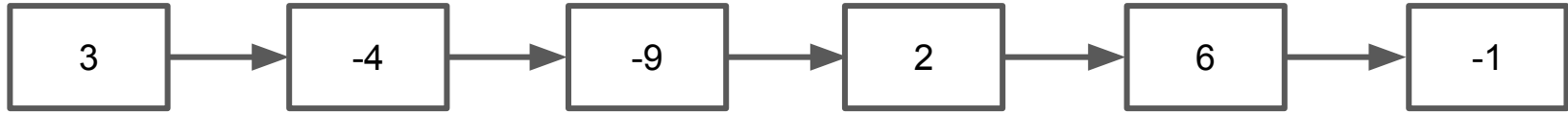
# LinkedList Tips

- Draw lots of pictures! Make sure you know exactly where you want things to point, and draw out every step (you want to always have a pointer to everything you want to access)
- Make sure you delete a node when you don't need it anymore (but after you saved its next)
- Good test cases for your list: empty list, list of size 1, list of size 2, list of size 3; try adding/deleting from the beginning, middle, and end

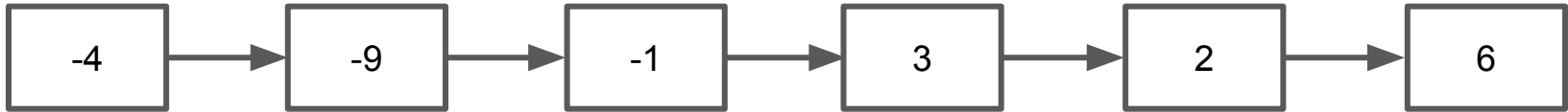
# LinkedList - Split

Write a function that given a LinkedList of integers, reorders the list so that all the negative numbers are at the front, and all the positive numbers are at the end.

Before



After

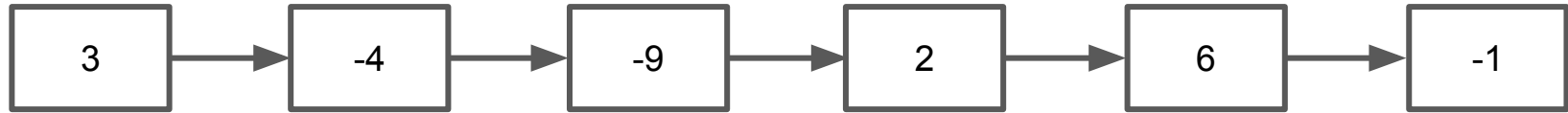




# LinkedList - Split

Algorithm: Separate the list into a list of positive numbers and negative numbers

Before



After

Negative Numbers



Positive Numbers

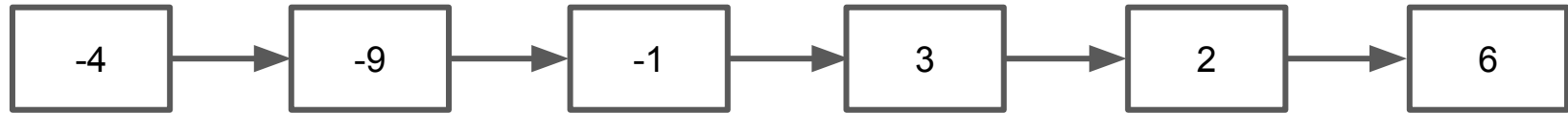


# LinkedList - Split

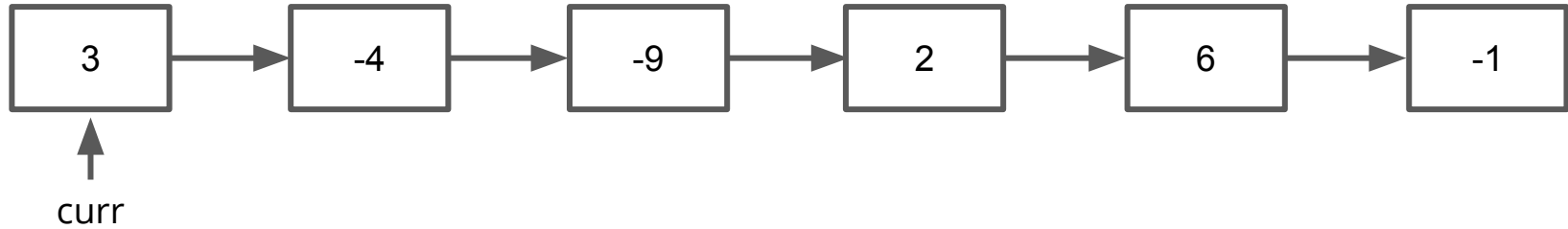
Algorithm: Separate the list into a list of positive numbers and negative numbers

Add the positive numbers to the end of the negative numbers, and return the start of the negative numbers list

Negative Numbers



# LinkedList - Split



Negative Numbers

negStart = NULL

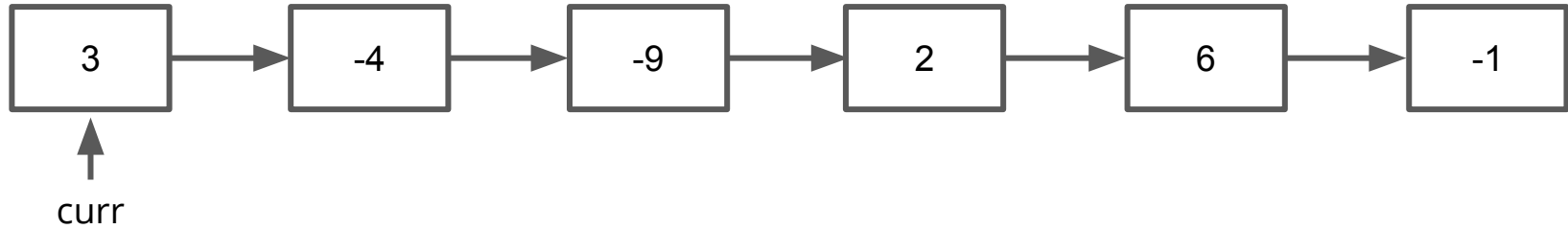
negEnd = NULL

Positive Numbers

posStart = NULL

posEnd = NULL

# LinkedList - Split



Negative Numbers

negStart = NULL

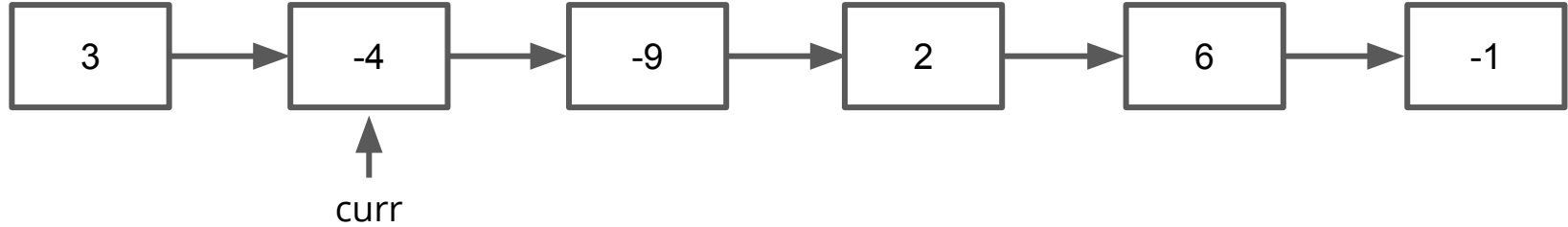
negEnd = NULL

Positive Numbers



posStart,  
posEnd

# LinkedList - Split



Negative Numbers

negStart = NULL

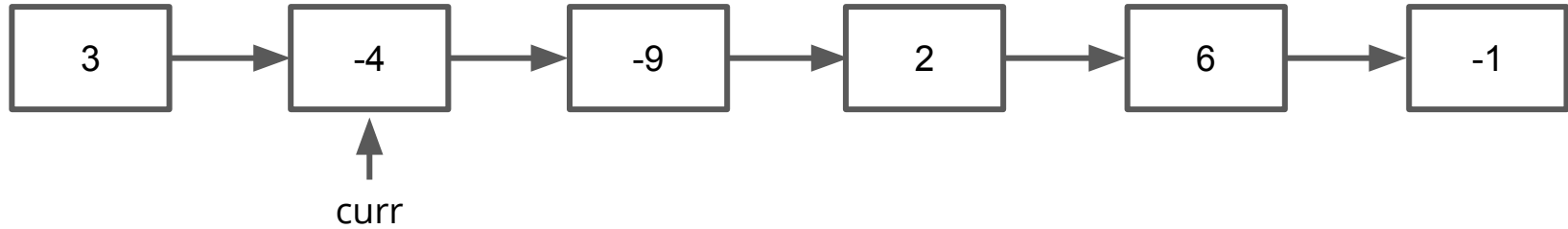
negEnd = NULL

Positive Numbers

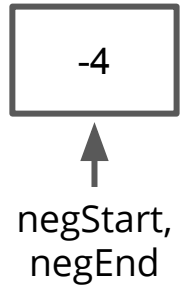


posStart,  
posEnd

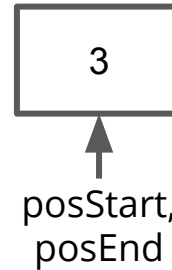
# LinkedList - Split



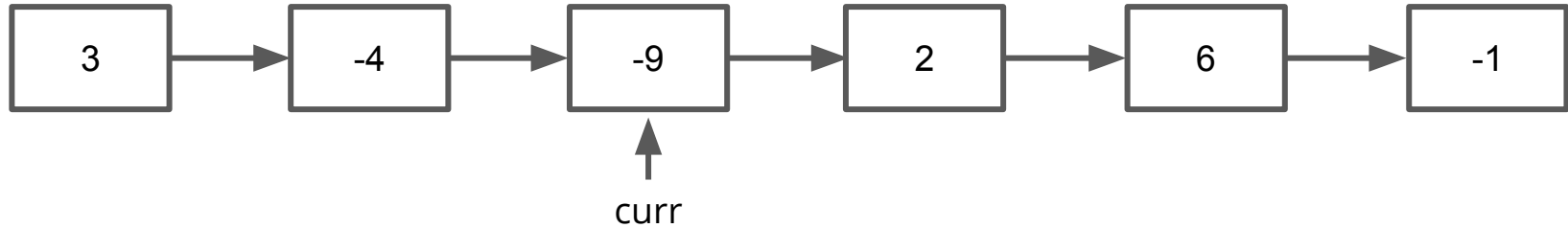
Negative Numbers



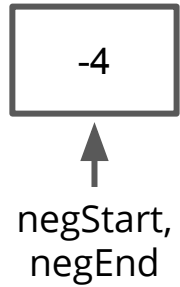
Positive Numbers



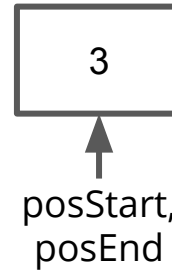
# LinkedList - Split



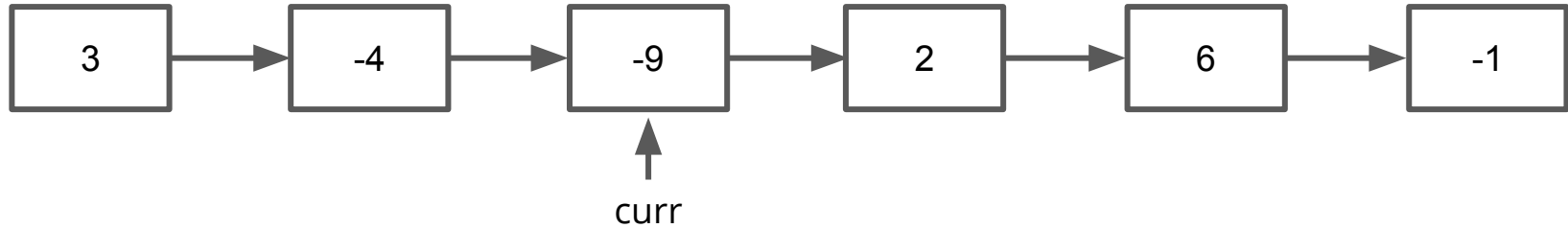
Negative Numbers



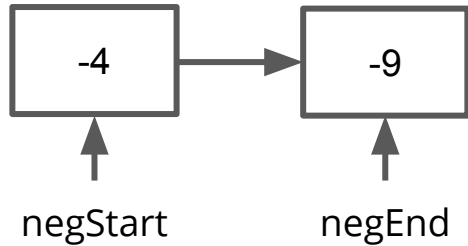
Positive Numbers



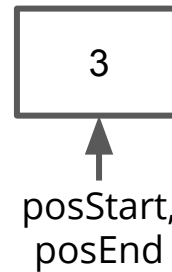
# LinkedList - Split



Negative Numbers

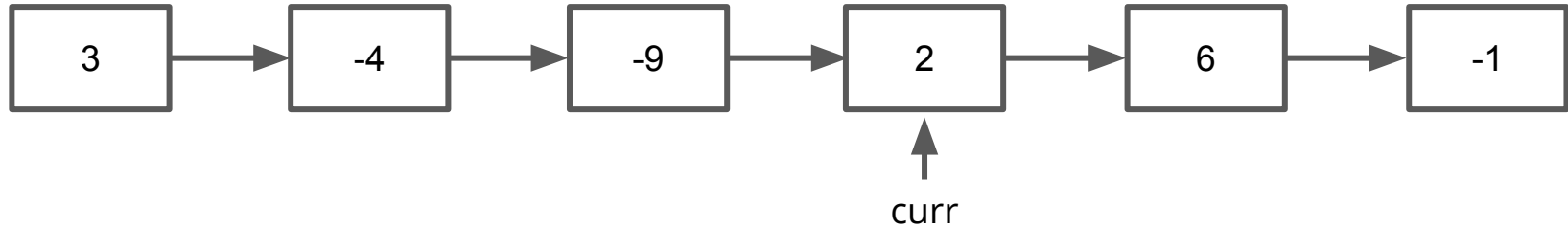


Positive Numbers

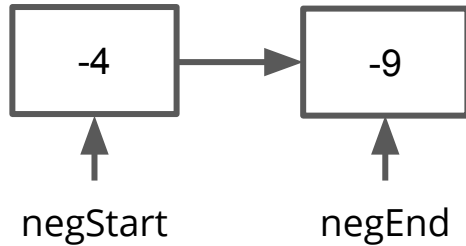




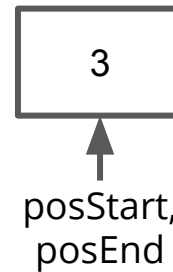
# LinkedList - Split



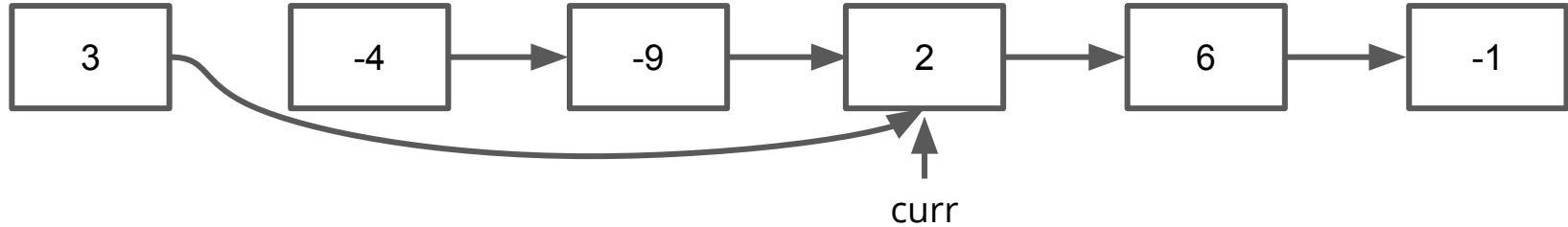
Negative Numbers



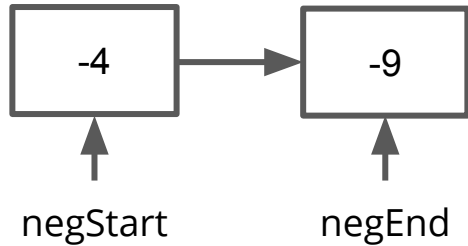
Positive Numbers



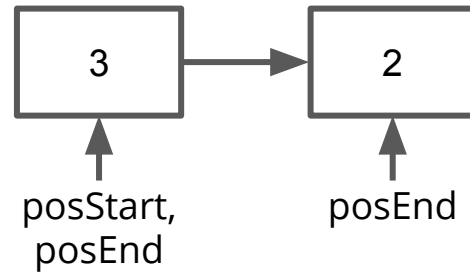
# LinkedList - Split



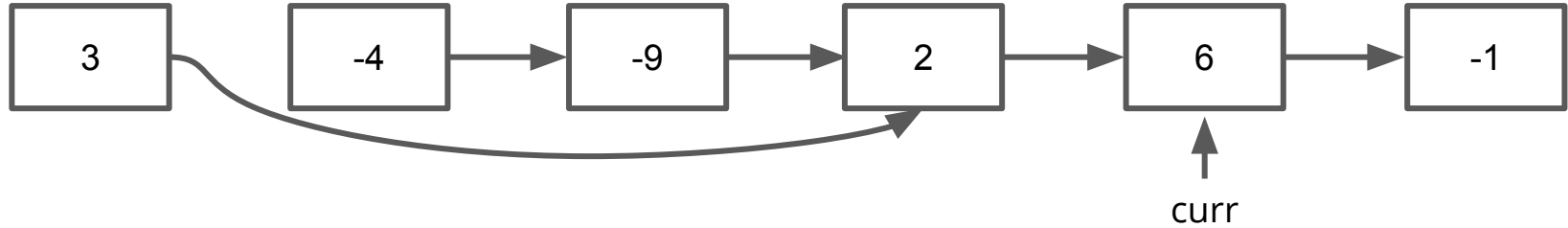
Negative Numbers



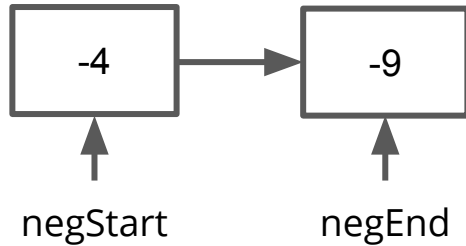
Positive Numbers



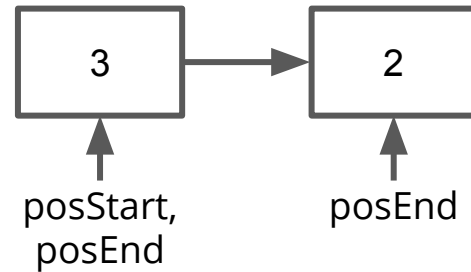
# LinkedList - Split



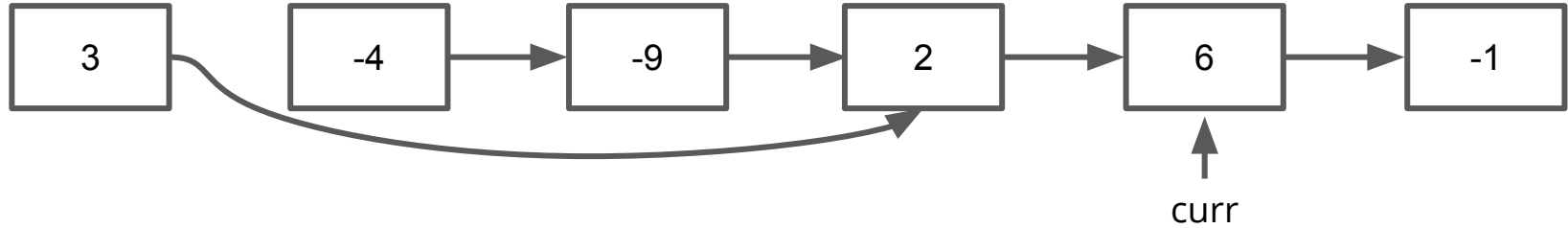
Negative Numbers



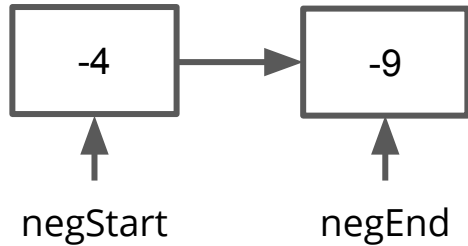
Positive Numbers



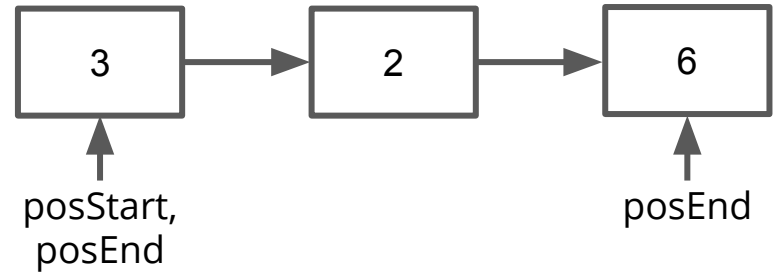
# LinkedList - Split



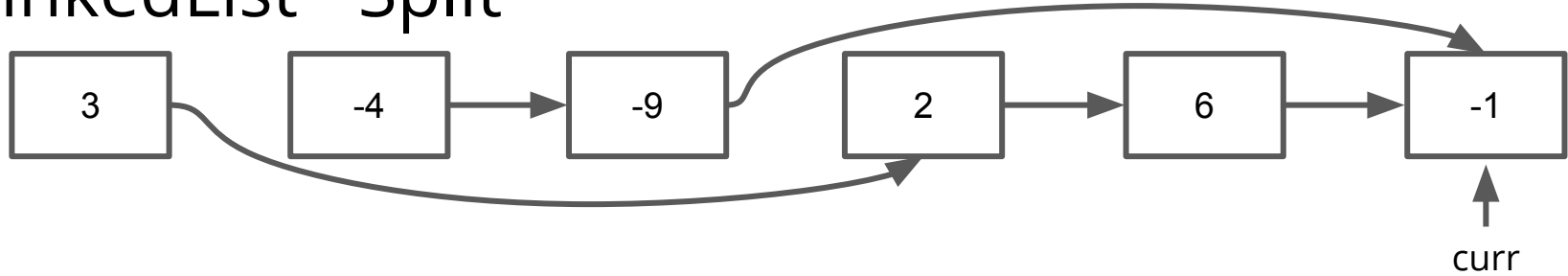
Negative Numbers



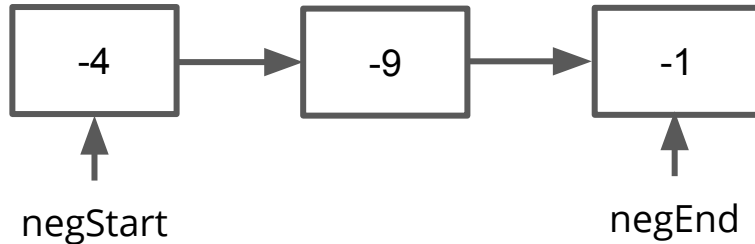
Positive Numbers



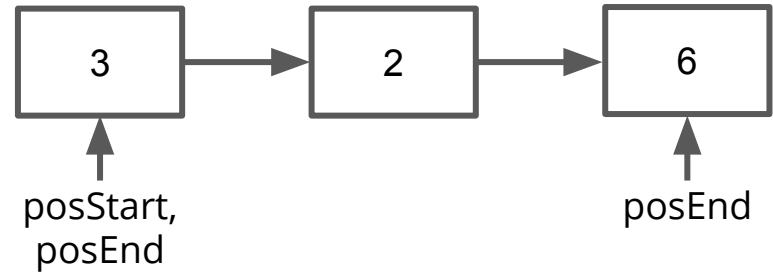
# LinkedList - Split



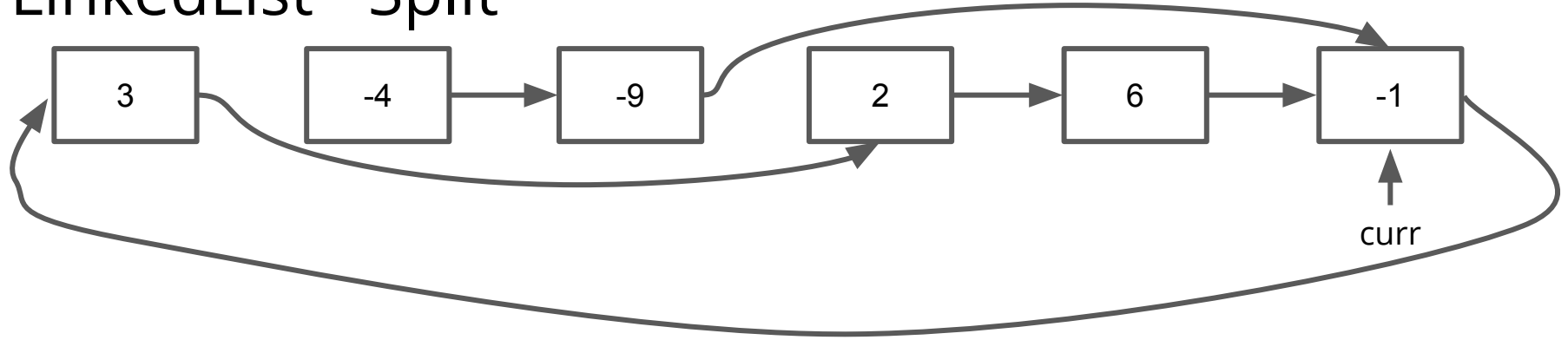
## Negative Numbers



## Positive Numbers

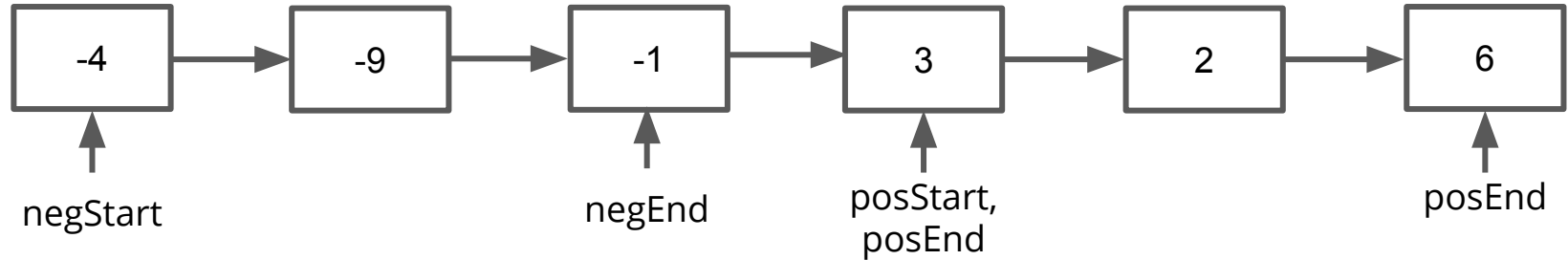


# LinkedList - Split

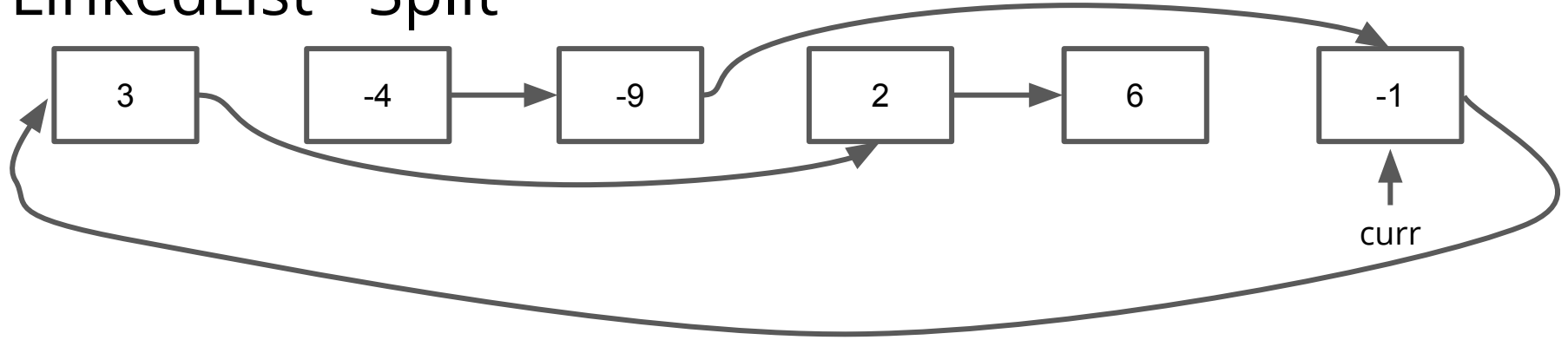


Negative Numbers

Positive Numbers



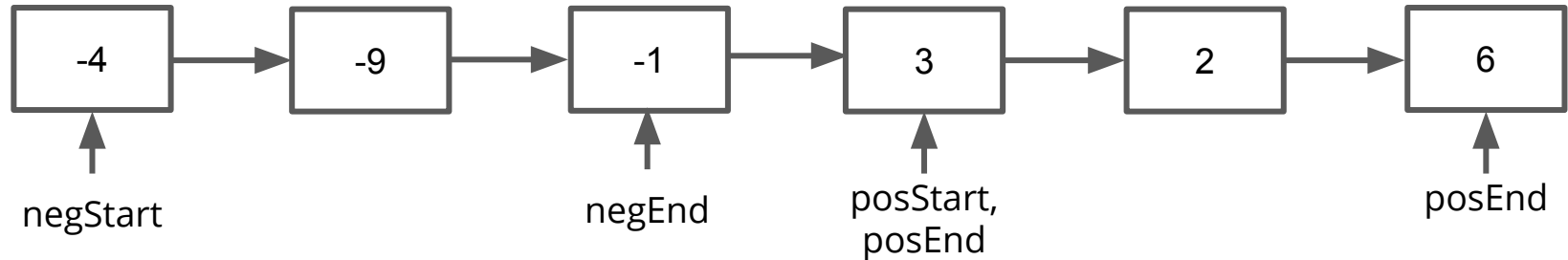
# LinkedList - Split



Negative Numbers

Positive Numbers

Make sure that  
posEnd points  
to NULL



# LinkedList - Split Solution (Part 1)

```
ListNode *split(ListNode *head) {
    ListNode *curr = head, *negStart = NULL, *negEnd = NULL, *posStart = NULL, *posEnd = NULL;
    while (curr != NULL) {
        if (curr->data < 0) {
            if (negStart != NULL) {
                negEnd->next = curr;
                negEnd = curr;
            } else {
                negStart = negEnd = curr;
            }
        } else {
            if (posStart != NULL) {
                posEnd->next = curr;
                posEnd = curr;
            } else {
                posStart = posEnd = curr;
            }
        }
        curr = curr->next;
    }
}
```



# LinkedList - Split Solution (Part 2)

```
// if posEnd is NULL,  
// then the end of our list will already point to NULL  
if (posEnd != NULL) {  
    posEnd->next = NULL;  
}  
  
// if there aren't any negative numbers,  
// just return the positive list  
if (negEnd != NULL) {  
    negEnd->next = posStart;  
    return negStart;  
} else {  
    return posStart;  
}  
}
```

# Extra Practice

- Traverse (i.e. read every element in a LinkedList) without notes
- Add an element to a LinkedList (you can use the add function of your PQueue for reference)
- Delete an element from a LinkedList (use changePriority of PQueue for reference)
- Check out CSBS for lots of practice

# Hashing

# Hash Functions

Basic definition: a hash function maps something (like an int or string) to a number

A **valid** hash function will always return the same number given two inputs that are considered equal

A **good** hash function distributes the values uniformly over all the numbers

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return randomInteger(0, 100);  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

INVALID - given the same bank account, might return any random number

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return randomInteger(0, 100);  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return account.routingNumber % 2;  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

VALID but Not Good - will generate the same result for two accounts that are considered the same, but not uniformly spread over all the integers

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return account.routingNumber % 2;  
}
```



# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return account.amount % 100;  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

INVALID - The bank account amount might change, leading to accounts with the same routing number being put in different buckets

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return account.amount % 100;  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return (account.routingNumber * 265443761L) %  
    INT_MAX;  
}
```

# Hash Functions: Good or Bad

```
struct BankAccount {  
    int routingNumber;  
    int amount;  
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {  
    return (account.routingNumber * 265443761L) %  
    INT_MAX;  
}
```

VALID and GOOD - given the same routing number, this will always return the same number, and it's spread relatively evenly over all possible (positive) integers

\*Taken from StackOverflow

# Using Hash Functions

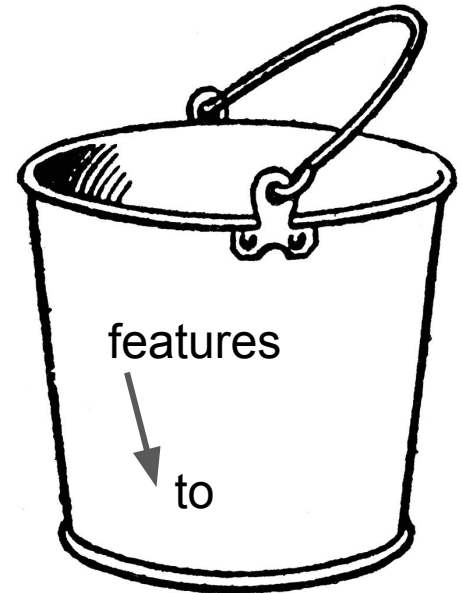
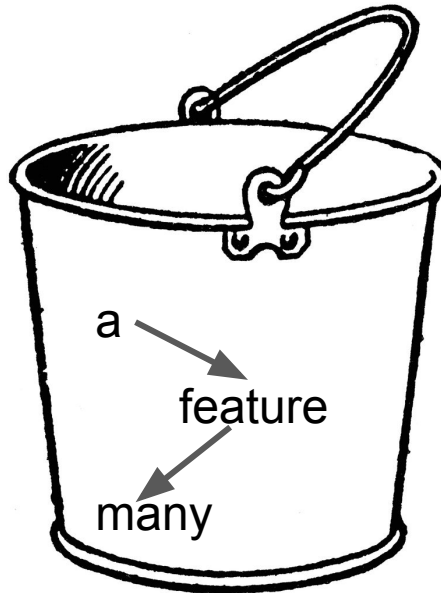
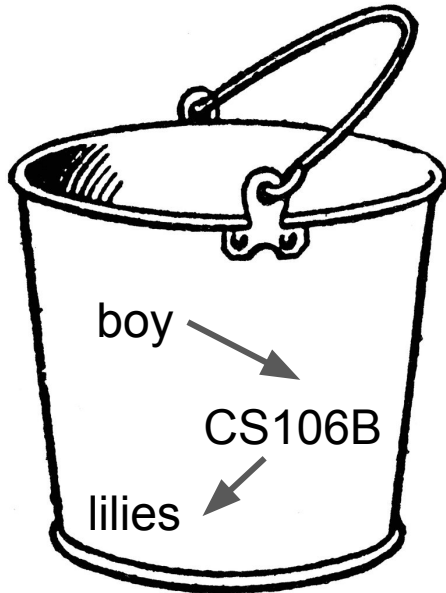
Hash Functions are used to assign elements to buckets for a HashSet or HashMap.

```
int bucket(elem) {  
    return hash(elem) % numBuckets;  
}
```

# Using Hash Functions

Let's say our hash function returned the length of the string we're hashing and we have 3 buckets. Our buckets may look like:

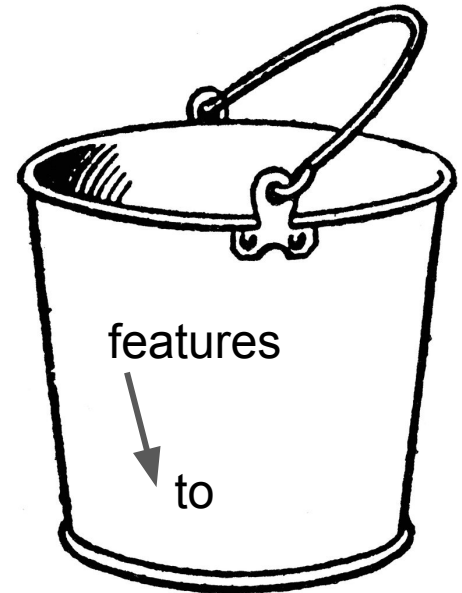
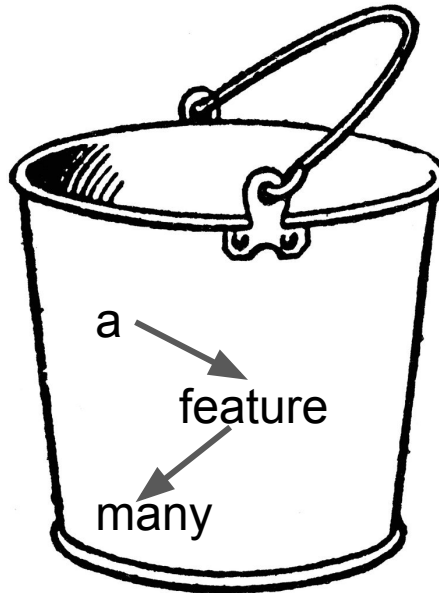
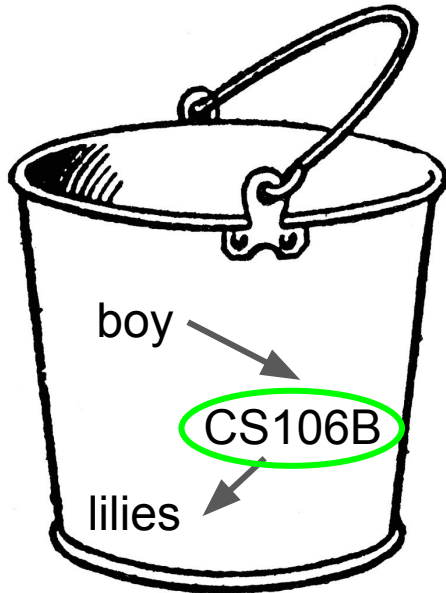
Inside each bucket, we have a LinkedList of elements



# Using Hash Functions

When we search for an element, we look in it's bucket

Search CS106B: look in bucket 0 and look through the whole  
LinkedList



# Heaps

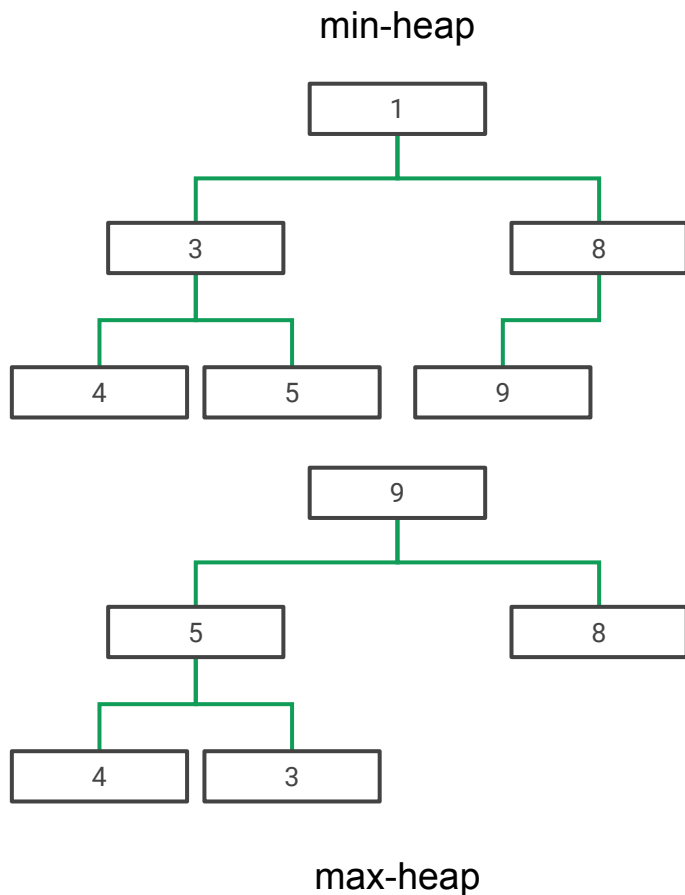


# Heaps: Overview

A heap is a type of **complete binary tree** that obeys some ordering property.  
(complete = each level is filled left to right before moving to the next level)

**Min-heap:** each parent is smaller than its children

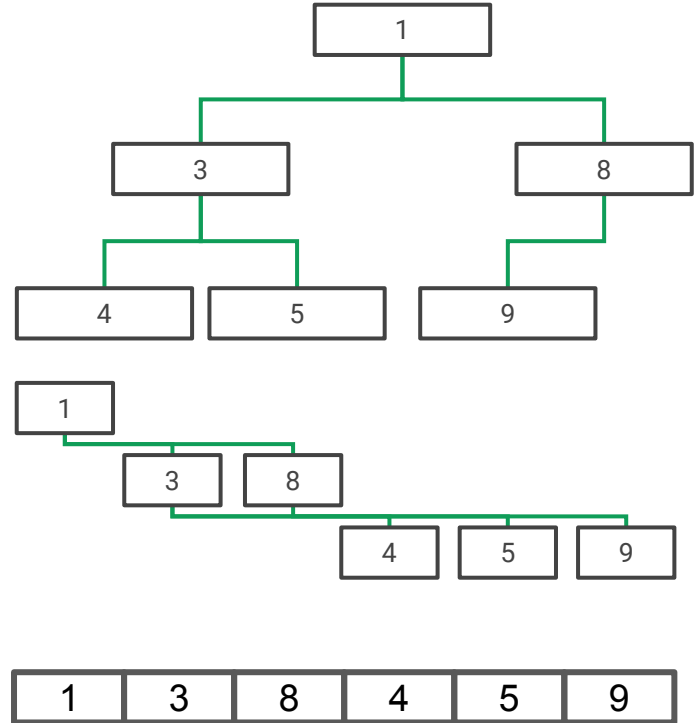
**Max-heap:** each parent is larger than its children



# Heaps: Stored as an array

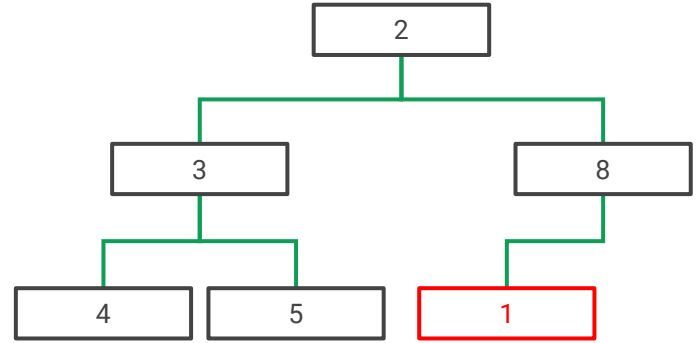
Usually stored in an array instead of as a tree

Written level by level (so that for a node at index  $n$ , its parent is at  $n/2$  and its children are at  $2n$  and  $2n+1$  if we one-index the array)



# Inserting into a heap: bubble-up (inserting 1)

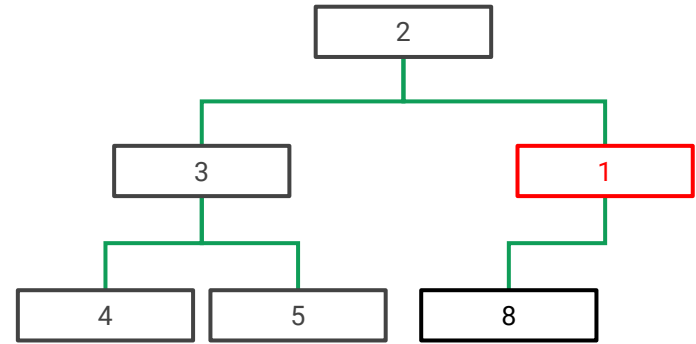
Add the element as a leaf node  
to the heap



# Inserting into a heap: bubble-up (inserting 1)

Add the element as a leaf node to the heap

If the element is less than its parent (if it's a min-heap), swap it with its parent

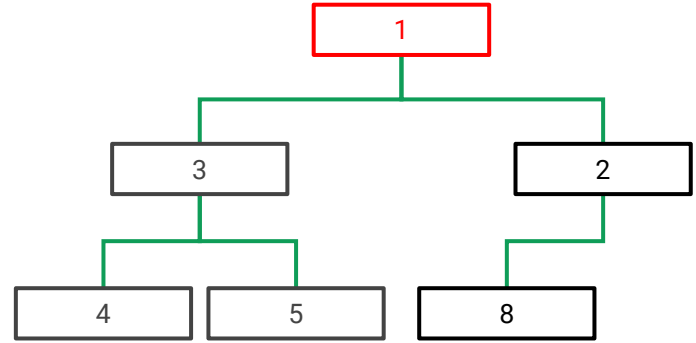


# Inserting into a heap: bubble-up (inserting 1)

Add the element as a leaf node to the heap

If the element is less than its parent (if it's a min-heap), swap it with its parent

Recursively continue swapping until you reach the root or the element's parent is smaller than it

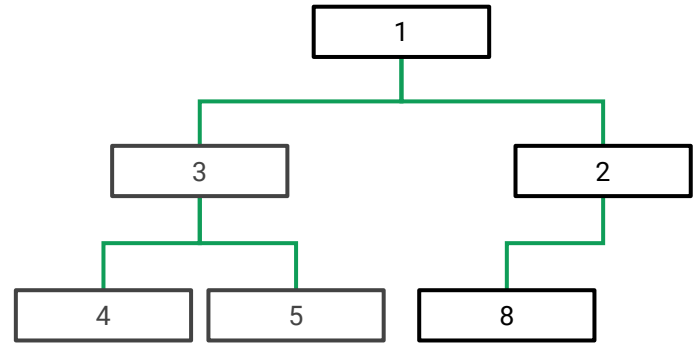


# Inserting into a heap: bubble-up (inserting 1)

Add the element as a leaf node to the heap

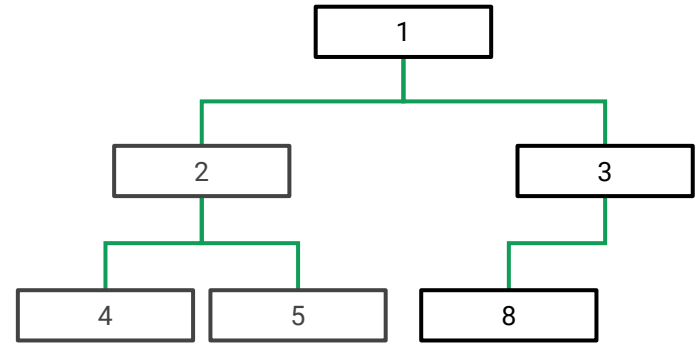
If the element is less than its parent (if it's a min-heap), swap it with its parent

Recursively continue swapping until you reach the root or the element's parent is smaller than it



# Deleting from the heap

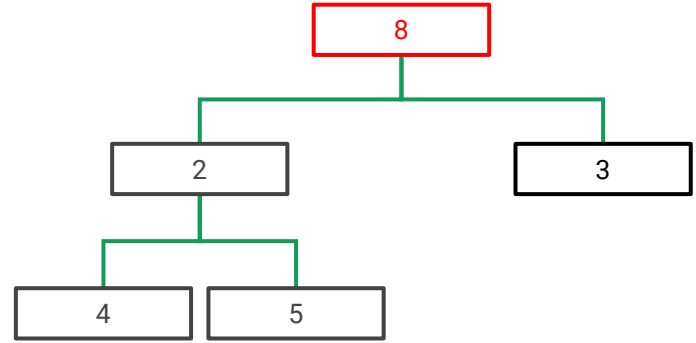
Remove the top element, and put the last leaf as the new root



# Deleting from the heap

Remove the top element, and put the last leaf as the new root

While the node is larger than its children (min-heap), swap it with its **smaller** child

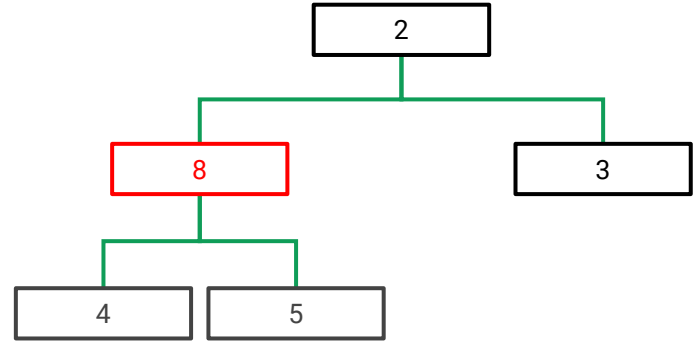




# Deleting from the heap

Remove the top element, and put the last leaf as the new root

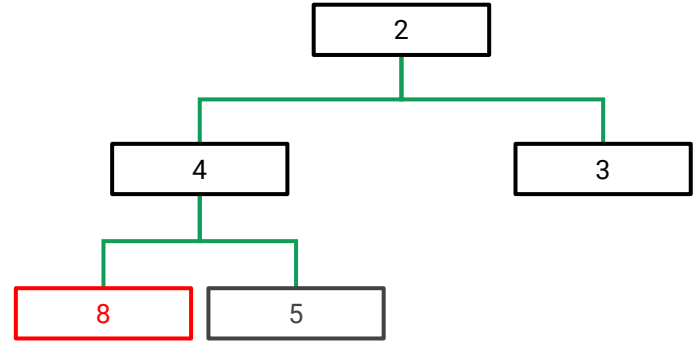
While the node is larger than its children (min-heap), swap it with its **smaller** child



# Deleting from the heap

Remove the top element, and put the last leaf as the new root

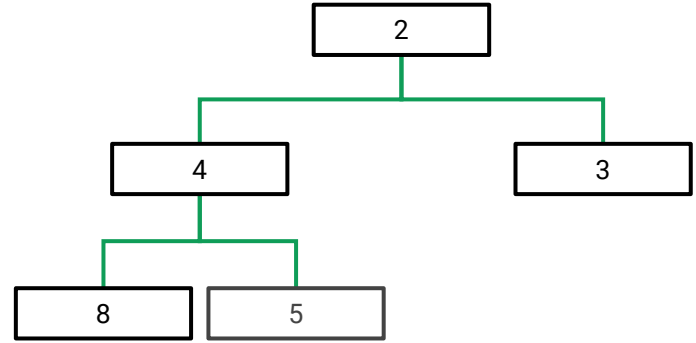
While the node is larger than at least one child (min-heap), swap it with its **smaller** child



# Deleting from the heap

Remove the top element, and put the last leaf as the new root

While the node is larger than at least one child (min-heap), swap it with its **smaller** child



# Binary Trees

# Binary Trees

Binary trees always have two children, though other kinds of trees can have more (see: Trie)

Another way to think of trees: an acyclic graph

# Trees - Traversals

General tree strategy: choose a traversal and implement the code that way

- Pre-order traversal: handle the root, then recurse to the two children
- In-order traversal: recurse to the left child, then handle the root, then recurse to the right child
- Post-order traversal: recurse to the children, then handle the root (e.g. deleting a tree)

Base case is usually that you've reached NULL

# Trees - Is it a Heap?

Given a pointer to a tree, is the tree a valid min-heap?

# Trees - Is it a Heap?

Given a pointer to a tree, is the tree a valid min-heap?

Recall: to be a min-heap, every parent must be smaller than both its children

Recall: every heap must be a **complete** binary tree

We can create two functions to check for these properties.



# Trees - followsMinProperty

Want to determine whether each parent is larger than both its children

Let's use preorder traversal (could write code with any traversal)

# Trees - followsMinProperty

```
bool followsMinProperty(TreeNode *root) {  
    if (root == NULL) {  
        return true;  
    }  
    if (root->left != NULL && root->left->data < root->data) {  
        return false;  
    }  
    if (root->right != NULL && root->right->data < root->data) {  
        return false;  
    }  
    return followsMinProperty(root->left) && followsMinProperty(root->right);  
}
```

# Trees - isCompleteTree

We can use a modified version of height

Recall: height of a tree is the number of nodes from the root to a NULL

How could we create a modified definition of height to determine whether a tree is **complete** (fills each level of the tree from left to right completely before going to the next row)?

# Trees - isCompleteTree

Idea: keep track of the closest and furthest away NULLs.

Idea: at each node, if the right subtree has height  $h$ , the left subtree can have either height  $h$  or  $h + 1$

# Trees - isCompleteTree

Idea: at each node, if the right subtree has modified height  $h$ , the left subtree can have either modified height  $h$  or  $h + 1$

Need to keep track of a modified height parameter

Post-order traversal - only after finding the modified height of the left and right subtrees can we determine whether our current subtree is complete

# Trees - isCompleteTree

```
bool isCompleteTree(TreeNode *head, int & minHeight, int & maxHeight) {
    if (head == NULL) {
        minHeight = 0;
        maxHeight = 0;
        return true;
    }
    int leftMinHeight, rightMinHeight, leftMaxHeight, rightMaxHeight;
    if (!isCompleteTree(head->left, leftMinHeight, leftMaxHeight) ||
        !isCompleteTree(head->right, rightMinHeight, rightMaxHeight)) {
        return false;
    }
    if (leftMinHeight < rightMaxHeight || leftMaxHeight > rightMinHeight + 1) {
        return false;
    }
    maxHeight = leftMaxHeight + 1;
    minHeight = rightMinHeight + 1;
    return true;
}
```

# Trees - isMinHeap

```
bool isMinHeap(TreeNode *root) {  
    int minHeight, maxHeight;  
    return followsMinProperty(root) &&  
        isCompleteTree(root, minHeight, maxHeight);  
}
```

# Binary Search Trees (BSTs)

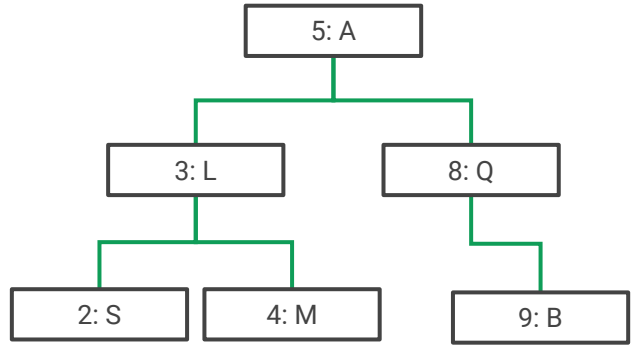


# BSTs: Overview

Every node has two children

Used to implement a Map: stores a **key** (which is how the tree is ordered) and a **value**

For every node in the tree, its key is greater than every key in the **left** subtree and less than every key in the **right** subtree (no ordering on values)



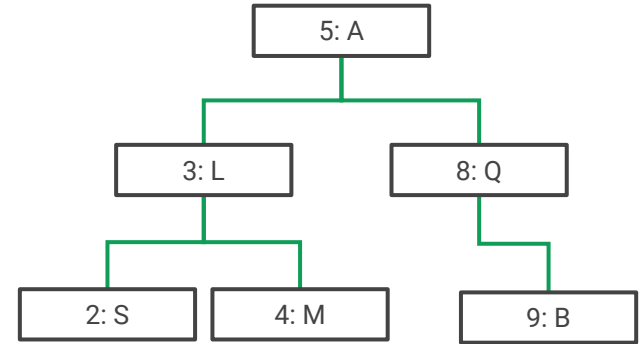
# BST Search (e.g. search for 4)

Start at the root

If the node's key is equal to the target,  
return the value

If the node's key is **greater** than the  
target, go **left**

If the node's key is **less** than the target,  
go **right**



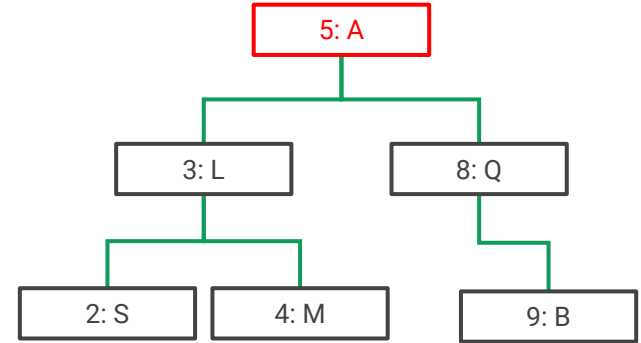
# BST Search (e.g. search for 4)

Start at the root

If the node's key is equal to the target,  
return the value

If the node's key is **greater** than the  
target, go **left**

If the node's key is **less** than the target,  
go **right**



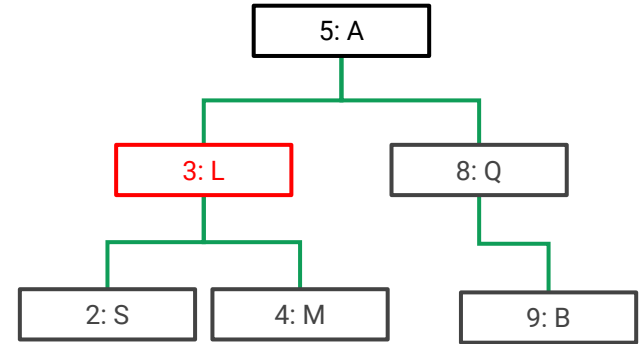
# BST Search (e.g. search for 4)

Start at the root

If the node's key is equal to the target,  
return the value

If the node's key is **greater** than the  
target, go **left**

If the node's key is **less** than the target,  
go **right**



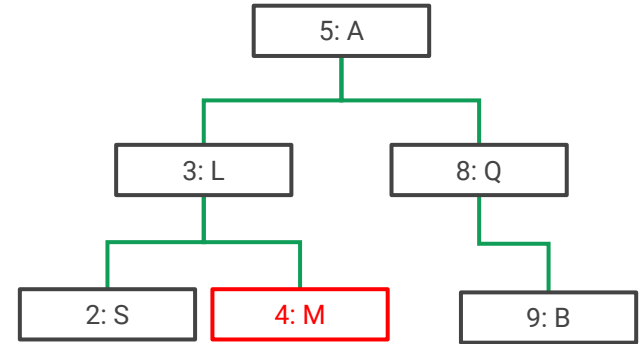
# BST Search (e.g. search for 4)

Start at the root

If the node's key is equal to the target,  
return the value

If the node's key is **greater** than the target, go **left**

If the node's key is **less** than the target, go **right**

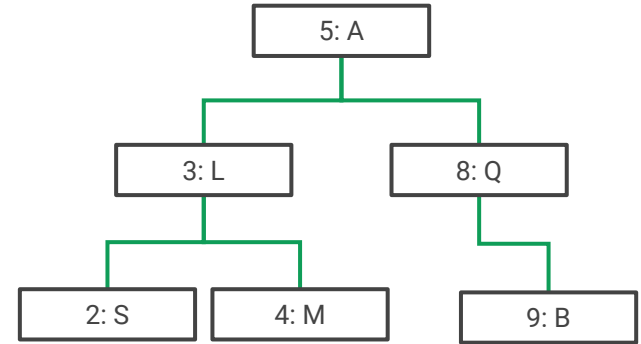


# BST Add

Search for the key

If you find it, change the value (Maps can't have duplicate keys)

If you don't, add the node as a leaf node to the last node you searched

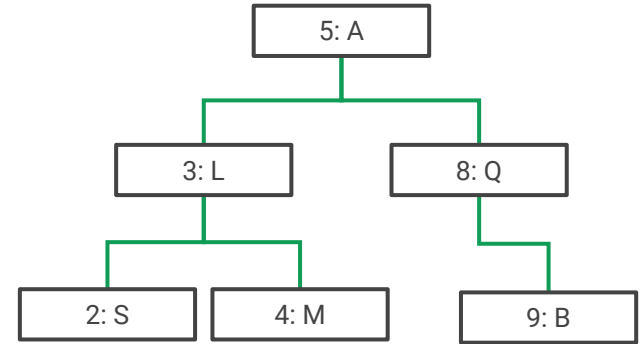


# BST Add - (4, P)

Search for the key

If you find it, change the value (Maps can't have duplicate keys)

If you don't, add the node as a leaf node to the last node you searched

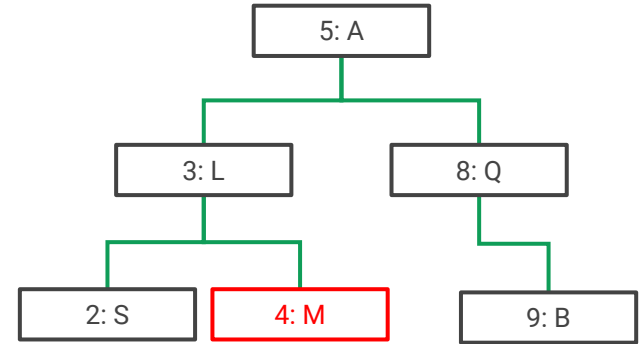


# BST Add - (4, P)

Search for the key

If you find it, change the value (Maps can't have duplicate keys)

If you don't, add the node as a leaf node to the last node you searched



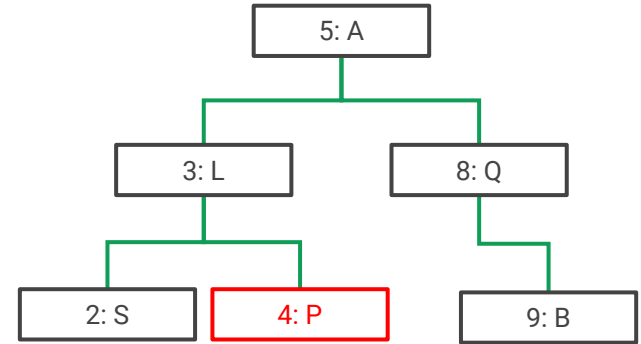


# BST Add - (4, P)

Search for the key

If you find it, change the value (Maps  
can't have duplicate keys)

If you don't, add the node as a leaf node  
to the last node you searched

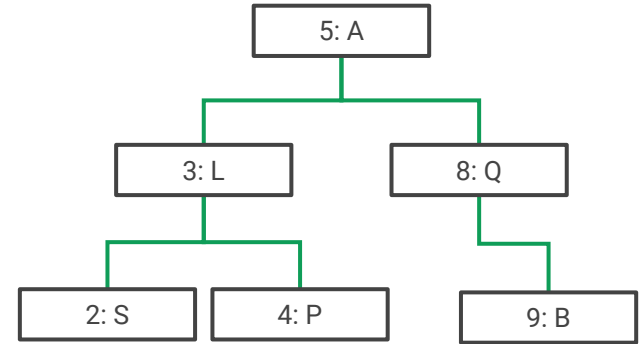


# BST Add - (1, P)

Search for the key

If you find it, change the value (Maps can't have duplicate keys)

If you don't, add the node as a leaf node to the last node you searched

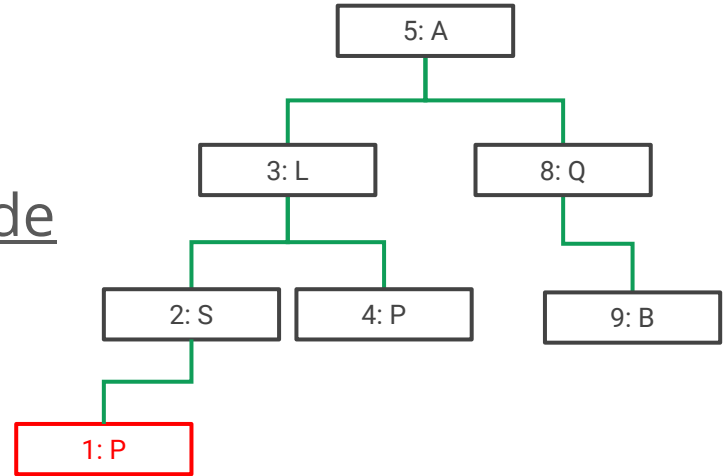


# BST Add - (1, P)

Search for the key

If you find it, change the value (Maps  
can't have duplicate keys)

If you don't, add the node as a leaf node  
to the last node you searched



# Graphs

# Graph Terminology

- Vertices and Edges
- Path: a sequence of edges that connect two nodes
- Cyclic vs. acyclic (is there a path from a vertex back to itself?)
- Directed vs. undirected (do the edges have a direction?)
- Connected: there is a path from each node to every other node
- Complete: there is an edge between every pair of nodes
- Weighted vs. unweighted (do the edges have a cost?)

# Important Graph Algorithms

- Depth-First Search: Good at determining if a path exists between two nodes
- Breadth-First Search: Finds the shortest path *in terms of number of edges* between two nodes
- Dijkstra's Algorithm: Finds the least cost path between two paths (same as BFS on unweighted graphs)
- A\*: a modified version of Dijkstra's that uses a heuristic
- Kruskal's Algorithm: Finds a minimum spanning tree

# Graphs - numConnectedComponents

Given a BasicGraph, find the number of connected components

What algorithm can we use?

# Graphs - numConnectedComponents

Given a BasicGraph, find the number of connected components

What algorithm can we use? Any of the important algorithms



# Graphs - numConnectedComponents

Given a BasicGraph, find the number of connected components

What algorithm should we use? DFS and BFS have better Big Oh runtimes.

# Graphs - numConnectedComponents

```
int numConnectedComponents(BasicGraph & graph) {  
    int result = 0;  
    Set<Vertex *> visited;  
    for (Vertex *node : graph.getVertexSet()) {  
        if (!visited.contains(node)) {  
            visited.add(node);  
            result++;  
            findConnectedComponent(graph, node, visited);  
        }  
    }  
    return result;  
}
```

# Graphs - numConnectedComponents

```
void findConnectedComponent(BasicGraph & graph, Vertex *node,
Set<Vertex *> &visited) {
    Set<Vertex *> neighbors;
    for (Vertex *neighbor : graph.getNeighbors(node)) {
        if (!visited.contains(neighbor)) {
            visited.add(neighbor);
            neighbors.add(neighbor);
        }
    }
    for (Vertex *neighbor : neighbors) {
        findConnectedComponent(graph, neighbor, visited);
    }
}
```

# Graphs Tips

Be able to trace the different algorithms and know the tradeoffs between them

Be able to iterate through both vertices, neighbors of a vertex, and edges

# Inheritance

# Inheritance - Flow Chart

