CS106X Final Review

Rachel Gardner and Jared Bitz

slides adapted from Ashley Taylor, Anton Apostolatos, Nolan Handali, and Zachary Birnholz

Final Review Session Overview

- Logistics
- Pointers and Dynamic Memory
- The Stack and the Heap
- Linked Lists
- Hashing
- Trees
- Recursion
- Graphs
- Inheritance, polymorphism and OOP (Object oriented programming)

Logistics

Final Logistics

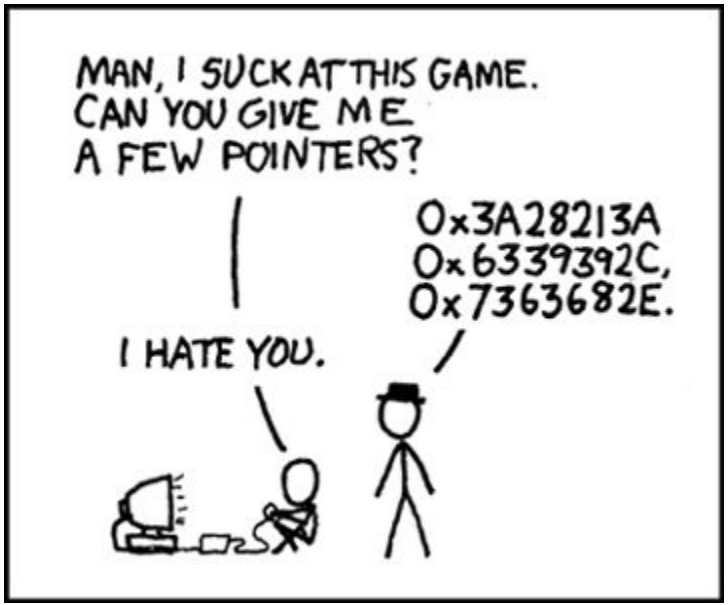
- Monday, December 10, 8:30-11:30am
- Room 420-041
- Open textbook, closed notes
- Exam taken on BlueBook (bring + charge your laptops!)
- Also bring a two-step authentication device
- Same rules as midterm

What’s not on the final

- Sorting
- Standard C++
- Overflow only material

Note: will be weighted to second half of the course

Linked Lists



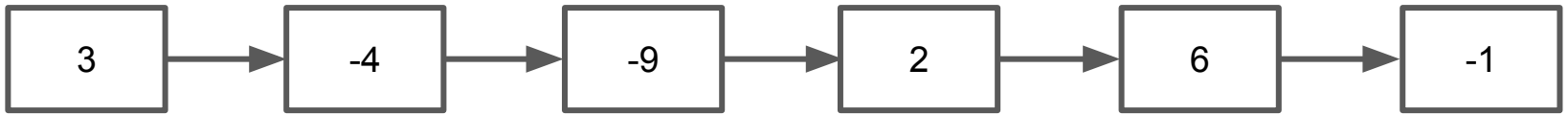
LinkedList Tips

- 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; try adding/deleting from the beginning, middle, and end

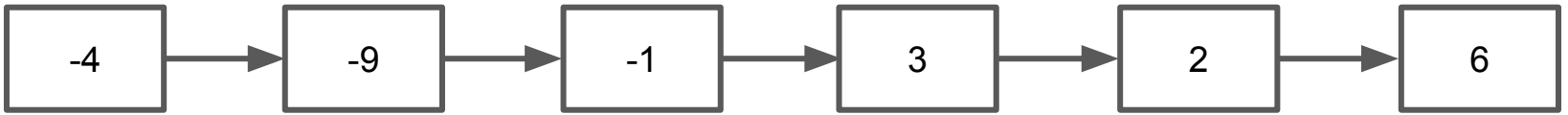
LinkedList - Split

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



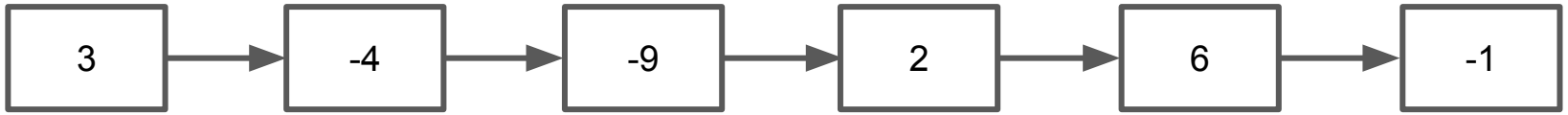
After



LinkedList - Split

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

Before



After

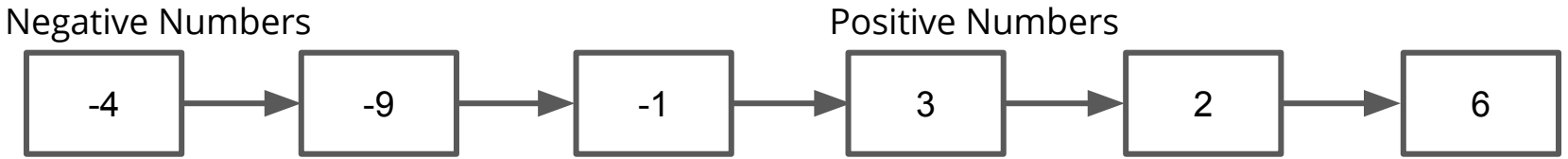
Negative Numbers

Positive Numbers

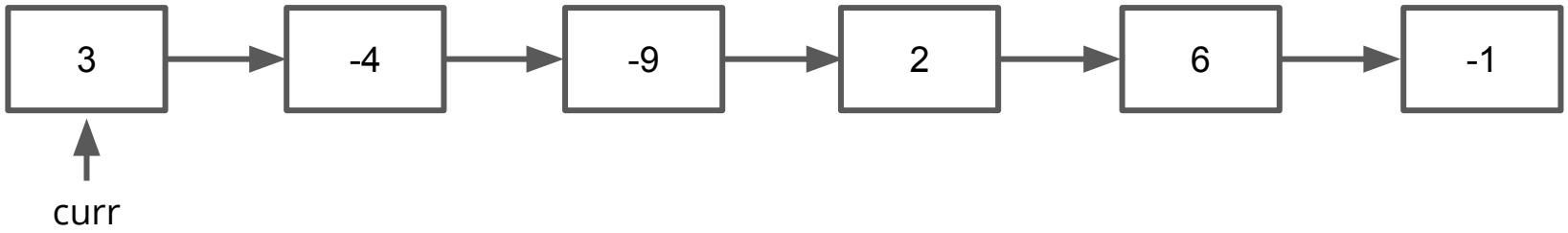
LinkedList - Split

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



LinkedList - Split



Negative Numbers

negStart = NULL

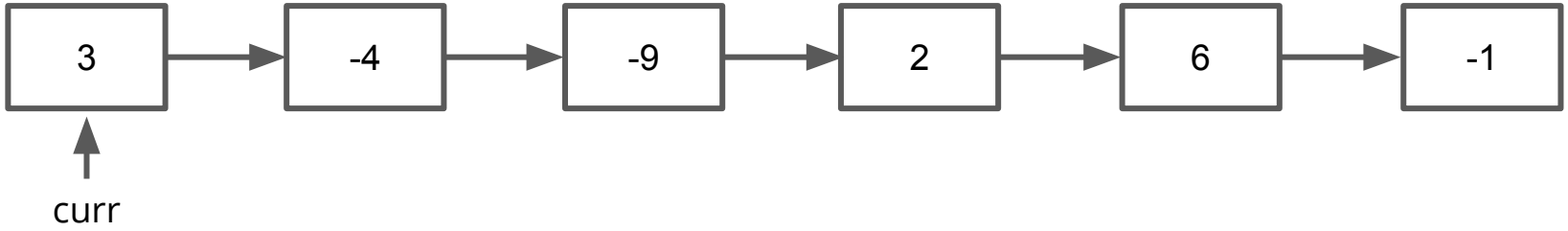
negEnd = NULL

Positive Numbers

posStart = NULL

posEnd = NULL

LinkedList - Split

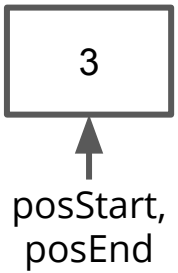


Negative Numbers

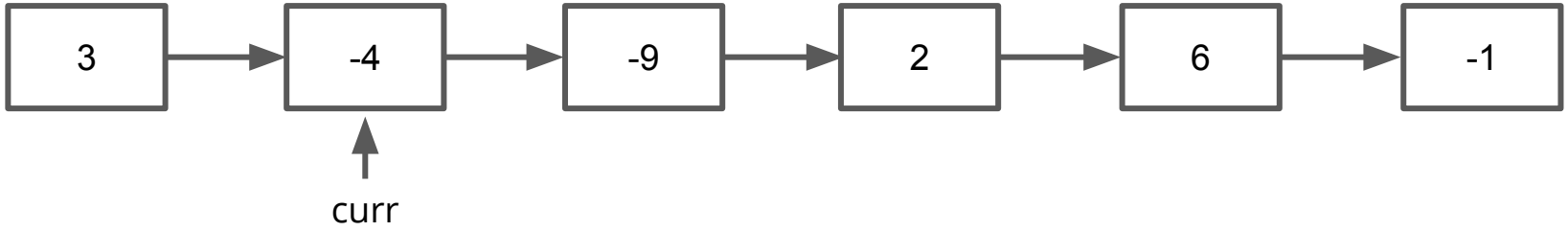
negStart = NULL

negEnd = NULL

Positive Numbers



LinkedList - Split

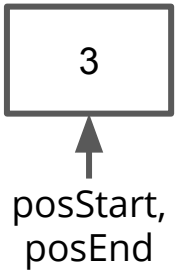


Negative Numbers

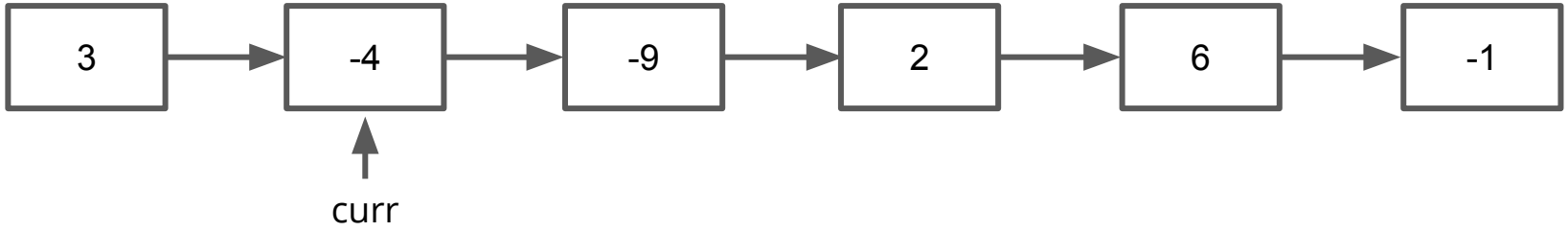
negStart = NULL

negEnd = NULL

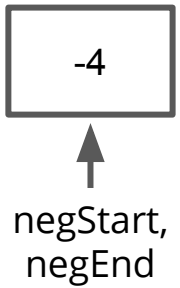
Positive Numbers



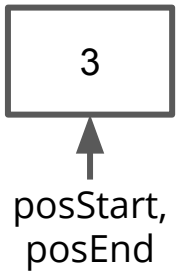
LinkedList - Split



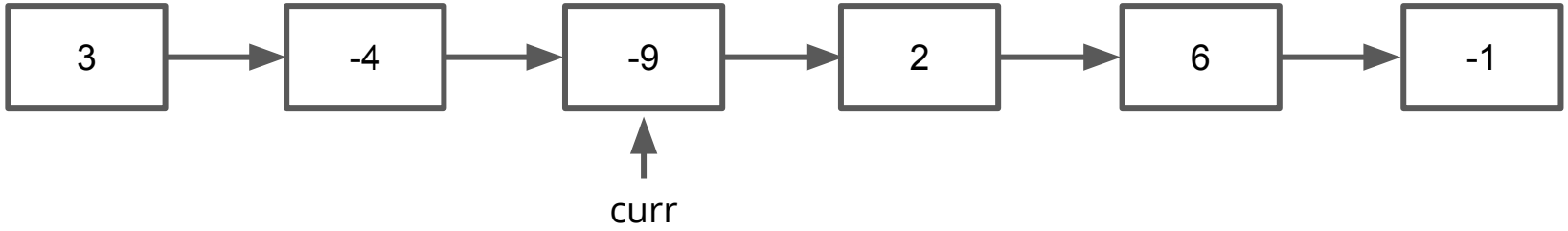
Negative Numbers



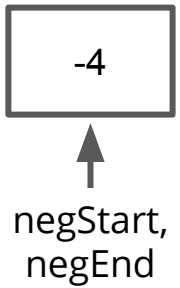
Positive Numbers



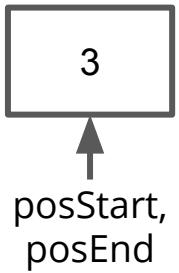
LinkedList - Split



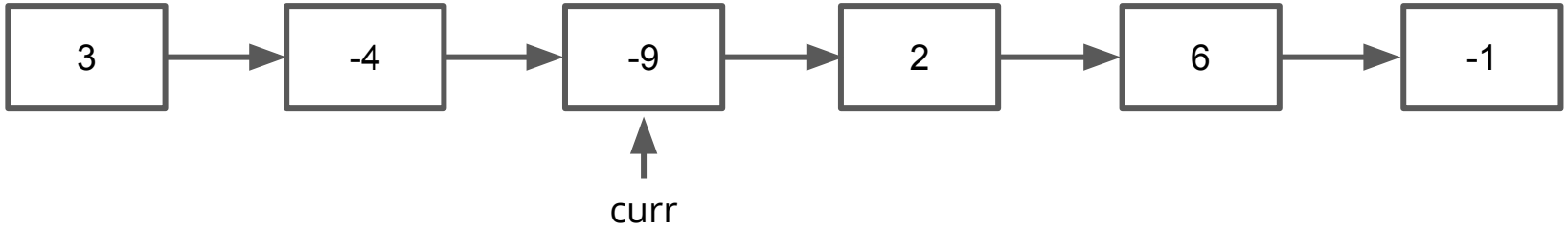
Negative Numbers



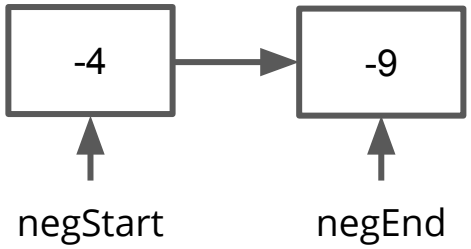
Positive Numbers



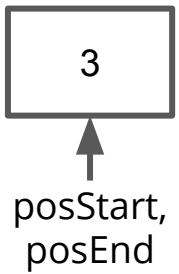
LinkedList - Split



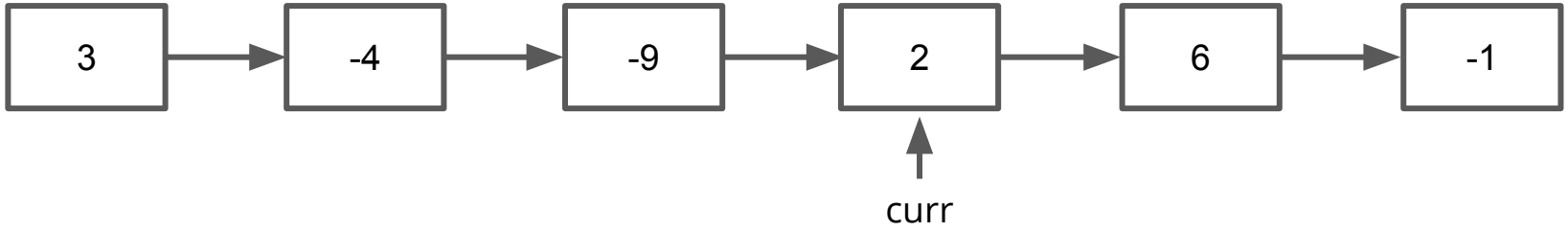
Negative Numbers



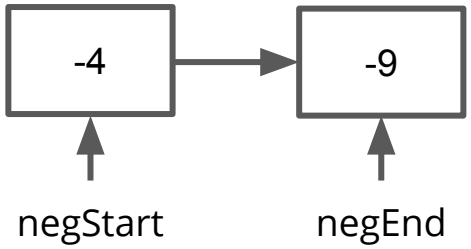
Positive Numbers



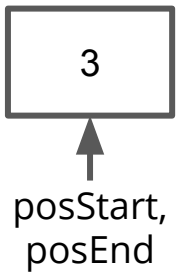
LinkedList - Split



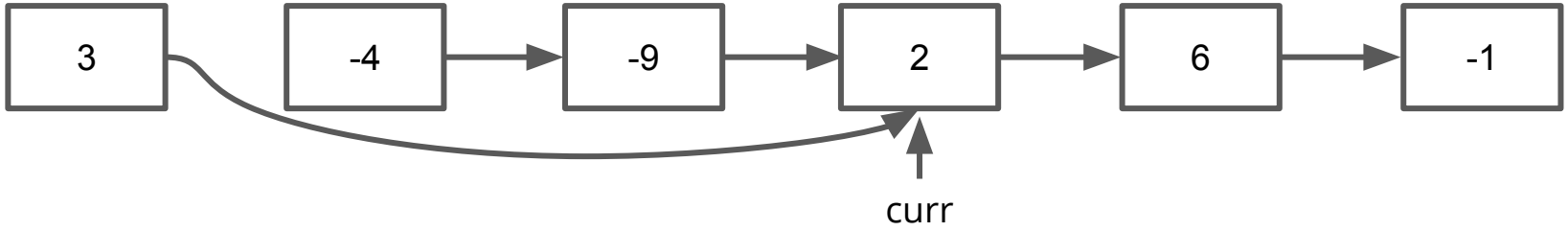
Negative Numbers



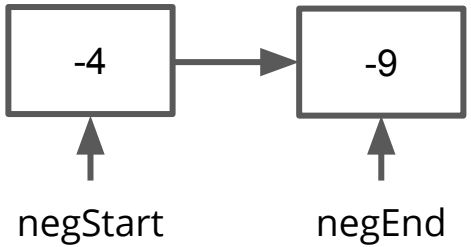
Positive Numbers



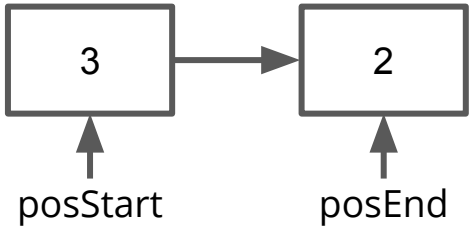
LinkedList - Split



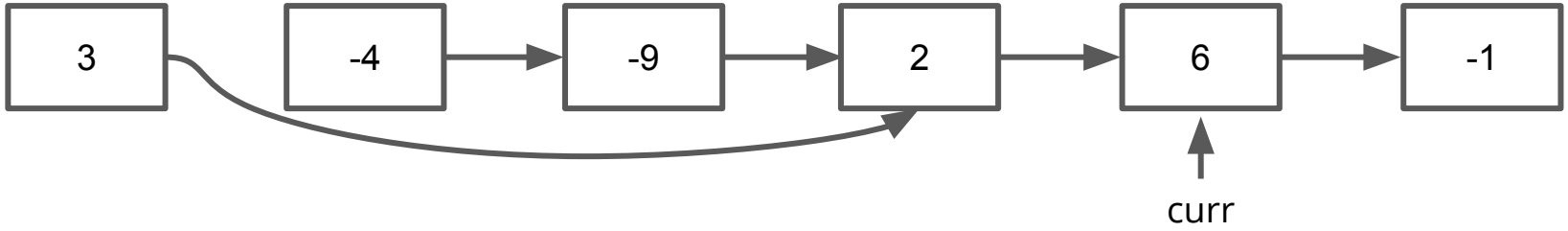
Negative Numbers



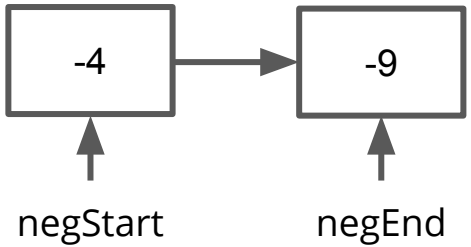
Positive Numbers



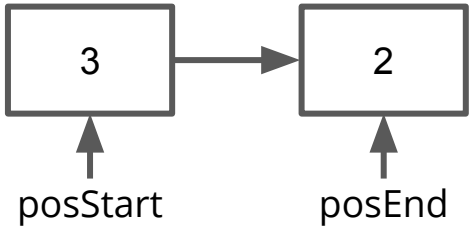
LinkedList - Split



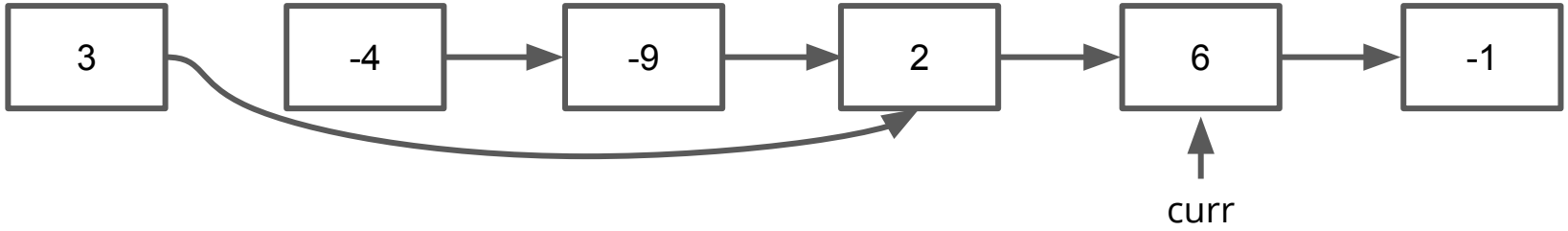
Negative Numbers



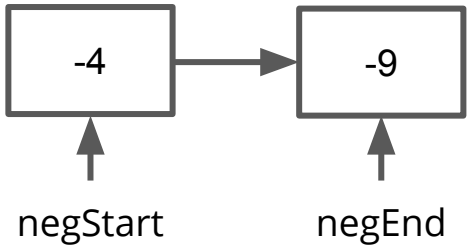
Positive Numbers



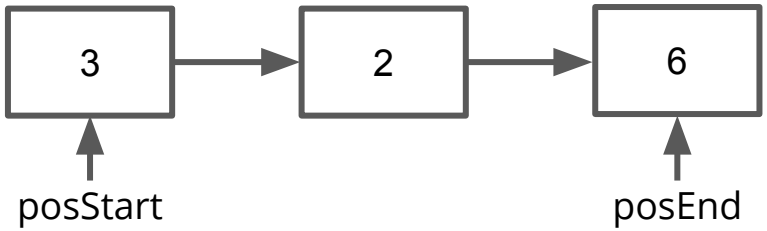
LinkedList - Split



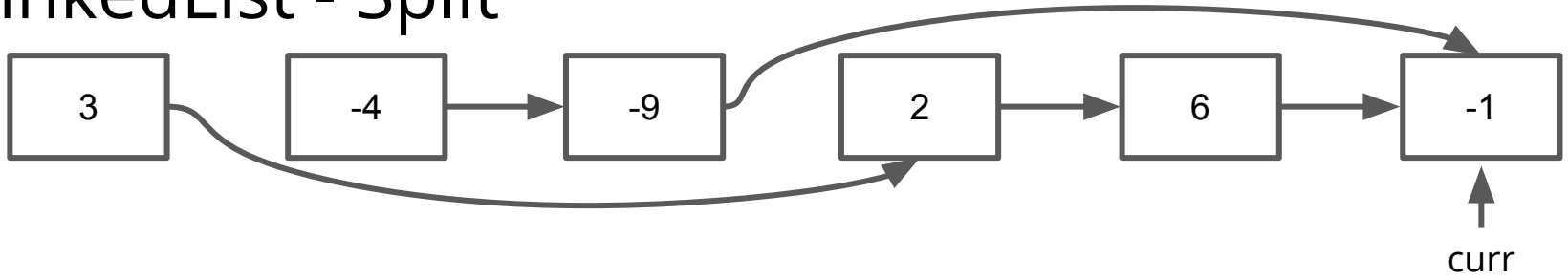
Negative Numbers



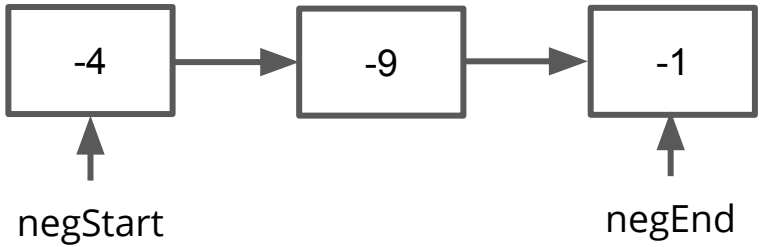
Positive Numbers



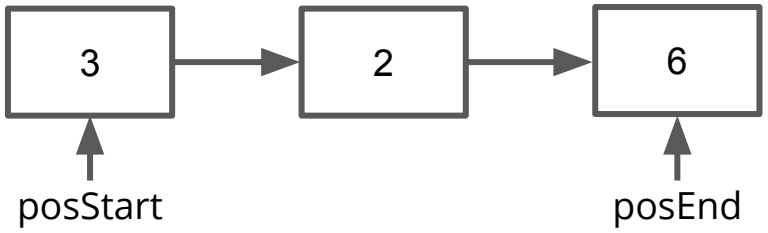
LinkedList - Split



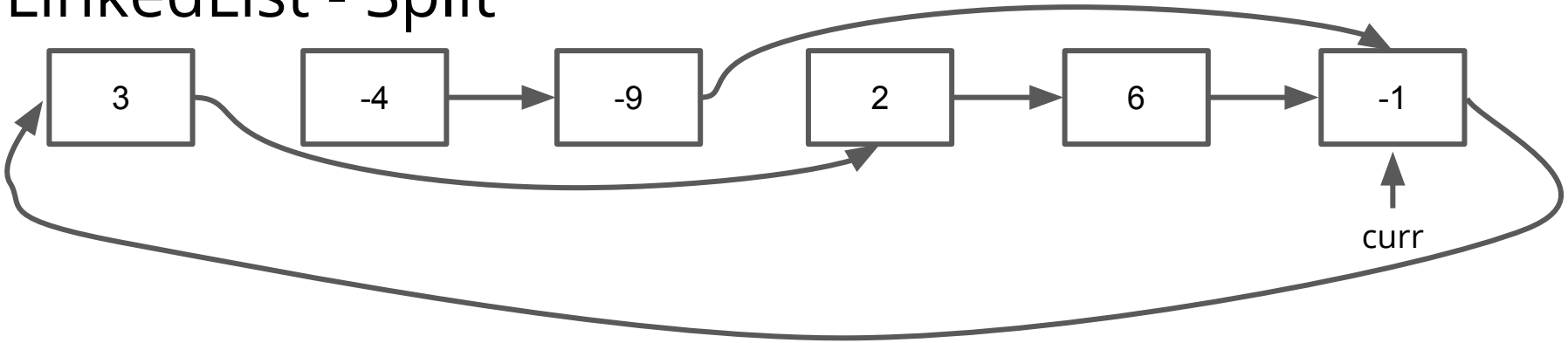
Negative Numbers



Positive Numbers

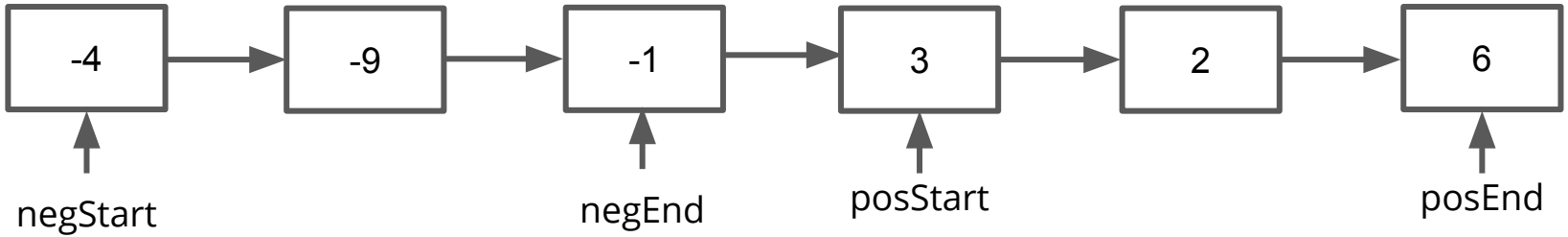


LinkedList - Split

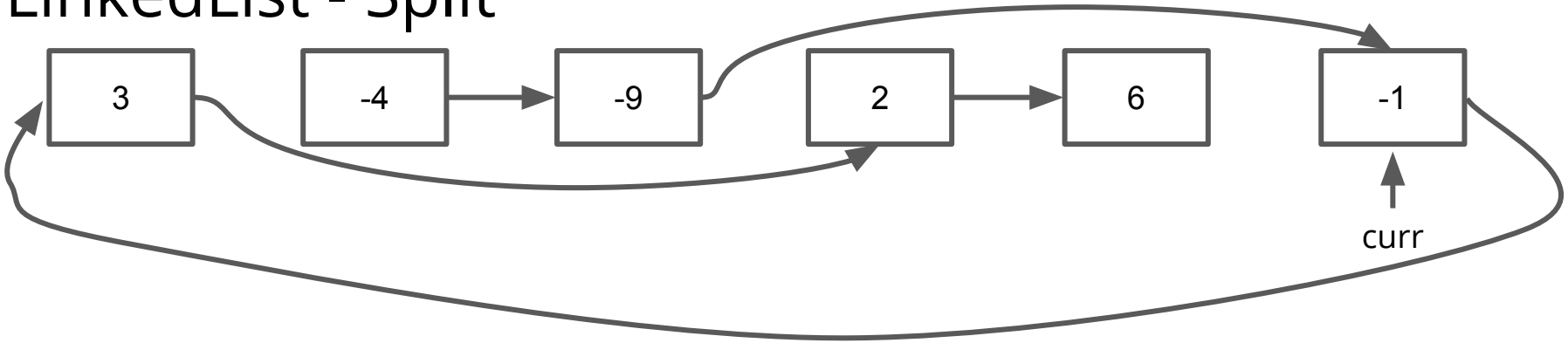


Negative Numbers

Positive Numbers



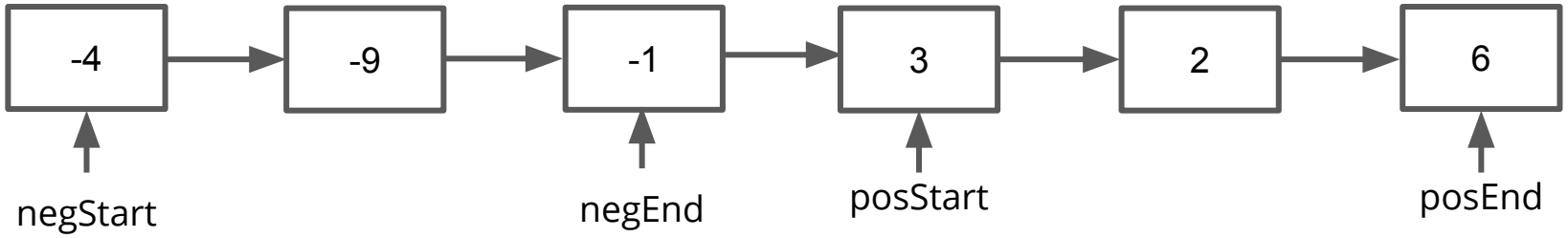
LinkedList - Split



Make sure that posEnd points to NULL

Negative Numbers

Positive Numbers



LinkedList - Split Solution (Part 1)

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

LinkedList - Split Solution (Part 2)

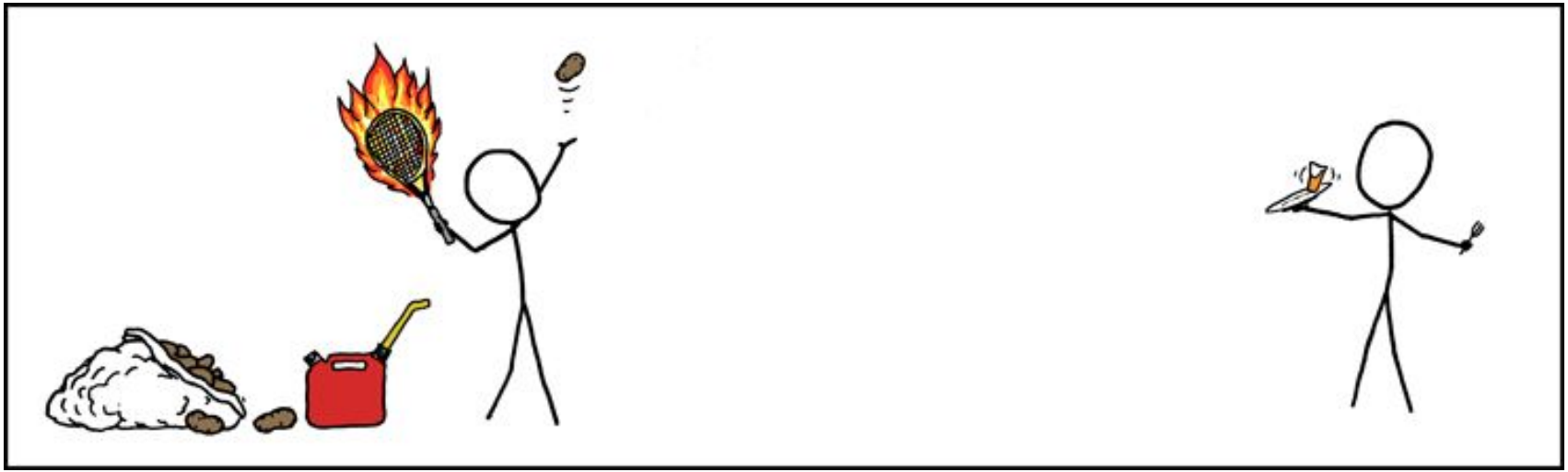
```
    // if posEnd is NULL,
    // then the end of our list will already point to NULL
    if (posEnd != nullptr) {
        posEnd->next = nullptr;
    }

    // if there aren't any negative numbers,
    // just return the positive list
    if (negEnd != nullptr) {
        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 (sorted) LinkedList
- Remove an element from a LinkedList and delete it
- 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, like assigning that something an ID 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 (few collisions)

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

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

```
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;

}
```

Using Hash Functions

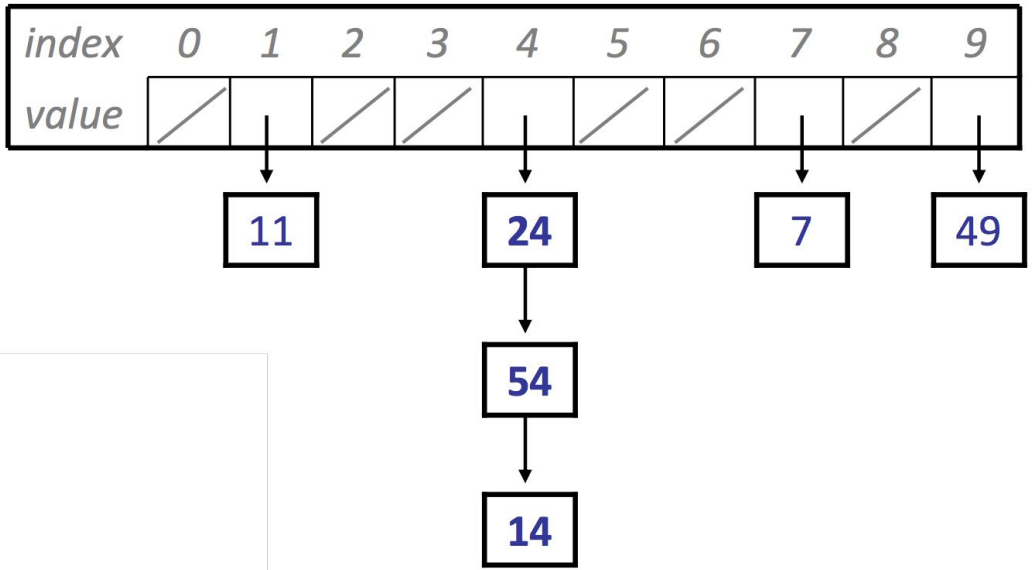
Hash Functions are used to assign elements to buckets for a HashSet or HashMap.

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

Using Hash Functions

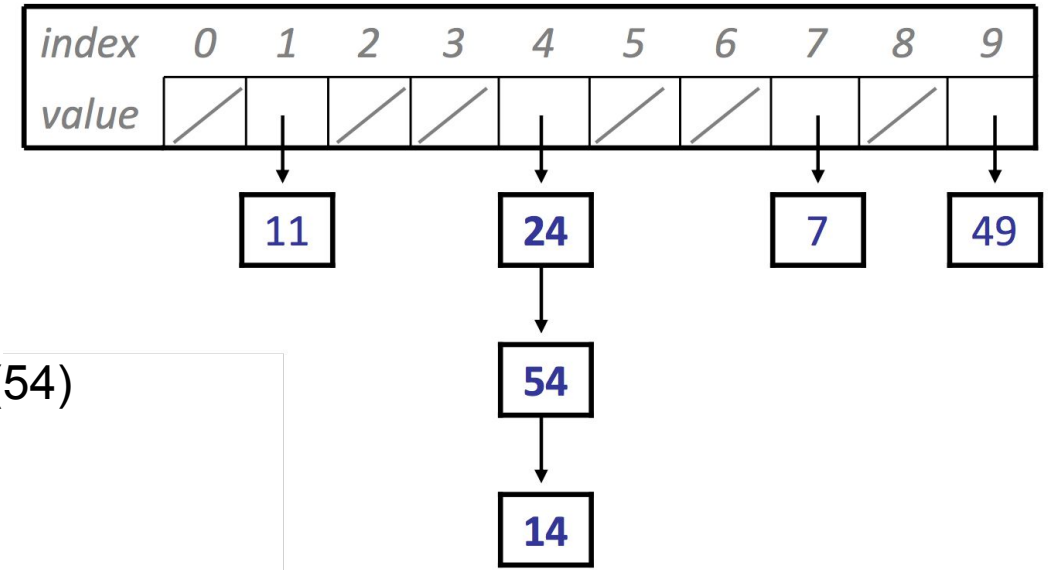
Inside each bucket, we have a LinkedList of elements.

In a HashSet of ints, our buckets may look like:



.contains in a HashSet

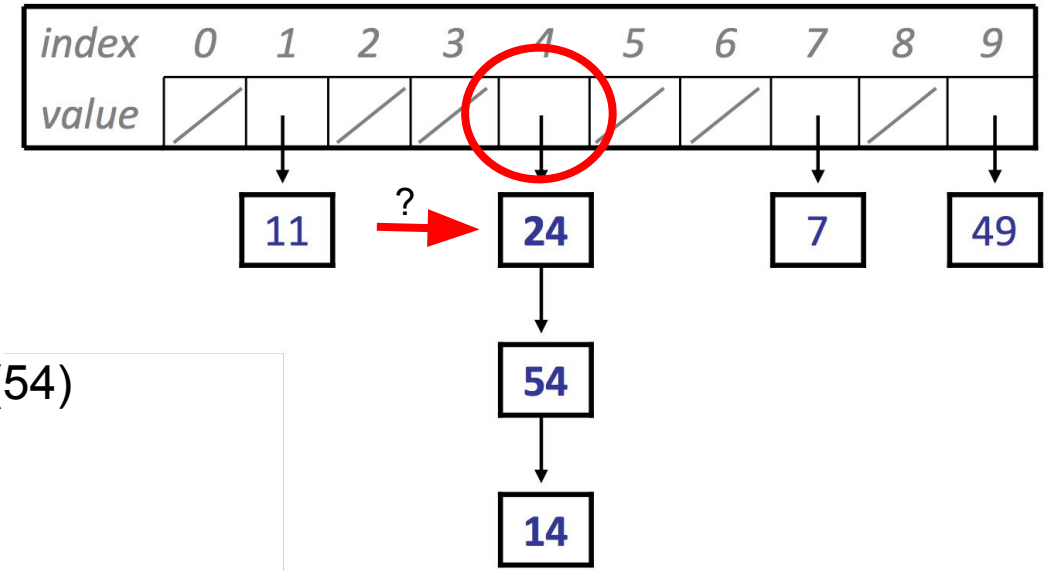
Given some element, loop over the LinkedList at the appropriate bucket and check if that element is in that LinkedList.



set.contains(54)

.contains in a HashSet

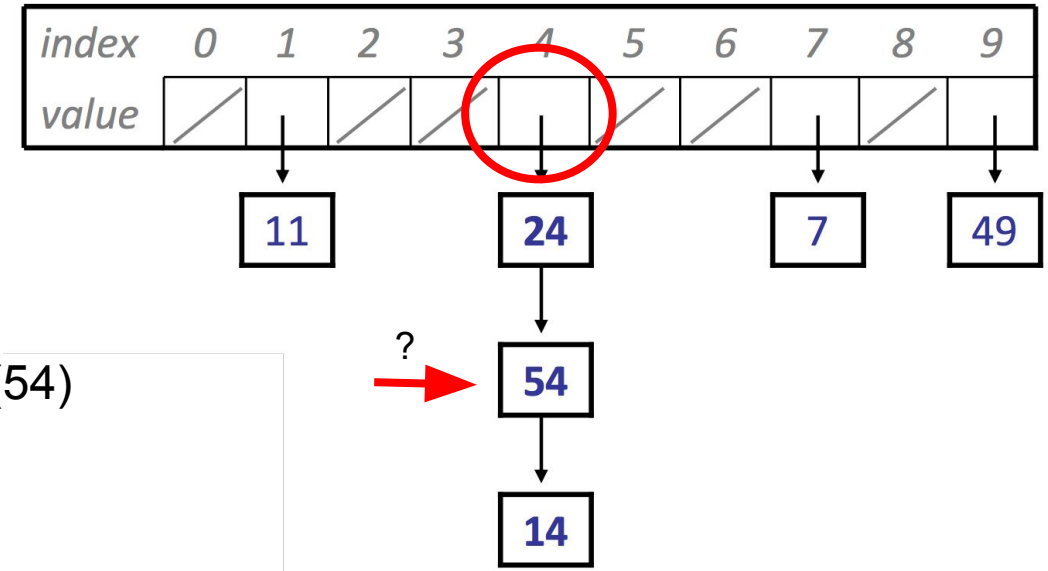
Given some element, loop over the LinkedList at the appropriate bucket and check if that element is in that LinkedList.



set.contains(54)

.contains in a HashSet

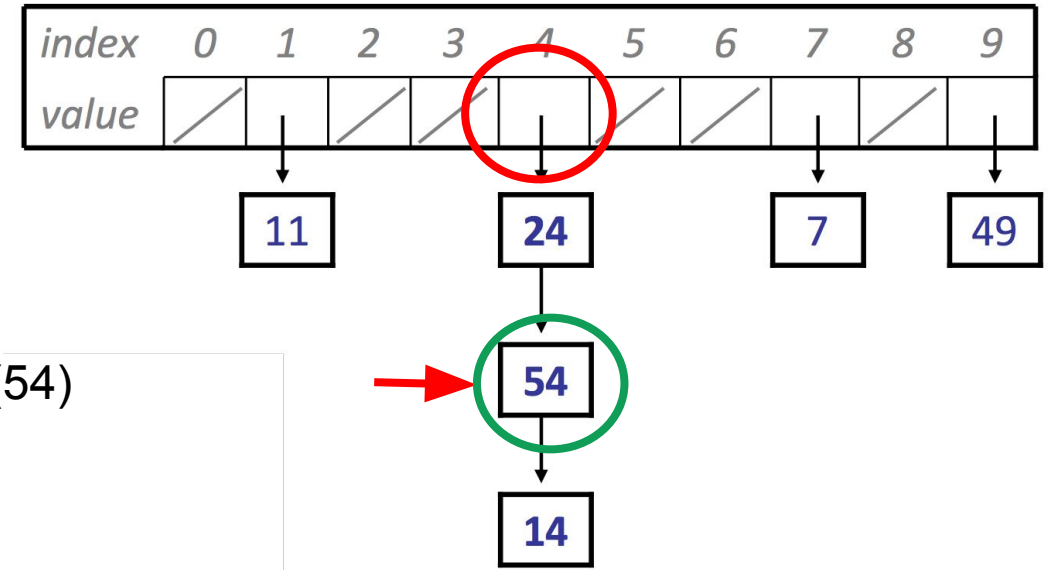
Given some element, loop over the LinkedList at the appropriate bucket and check if that element is in that LinkedList.



set.contains(54)

.contains in a HashSet

Given some element, loop over the LinkedList at the appropriate bucket and check if that element is in that LinkedList.



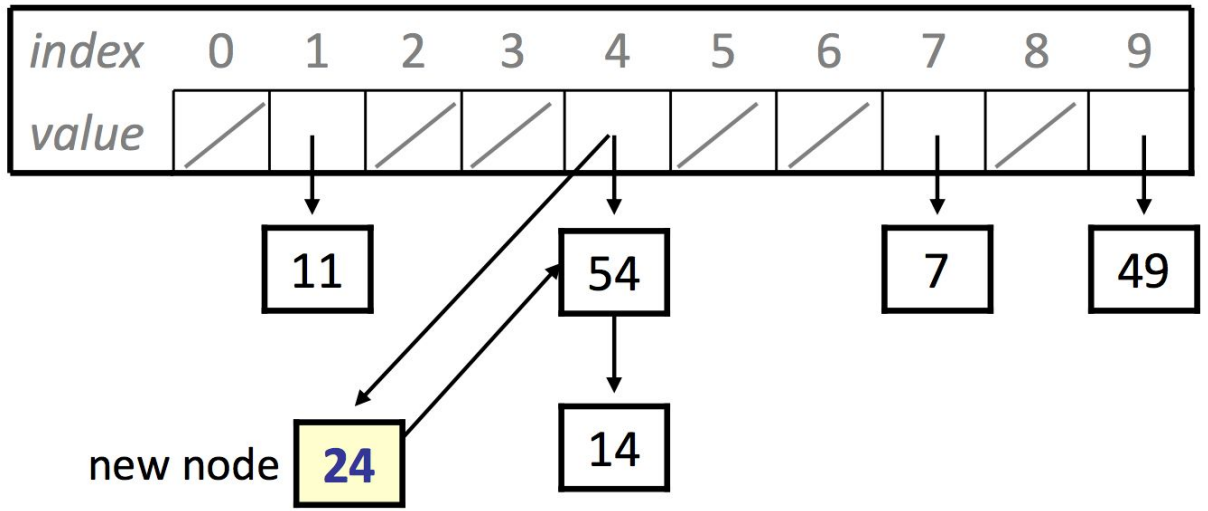
set.contains(54)

True!

Adding to a HashSet

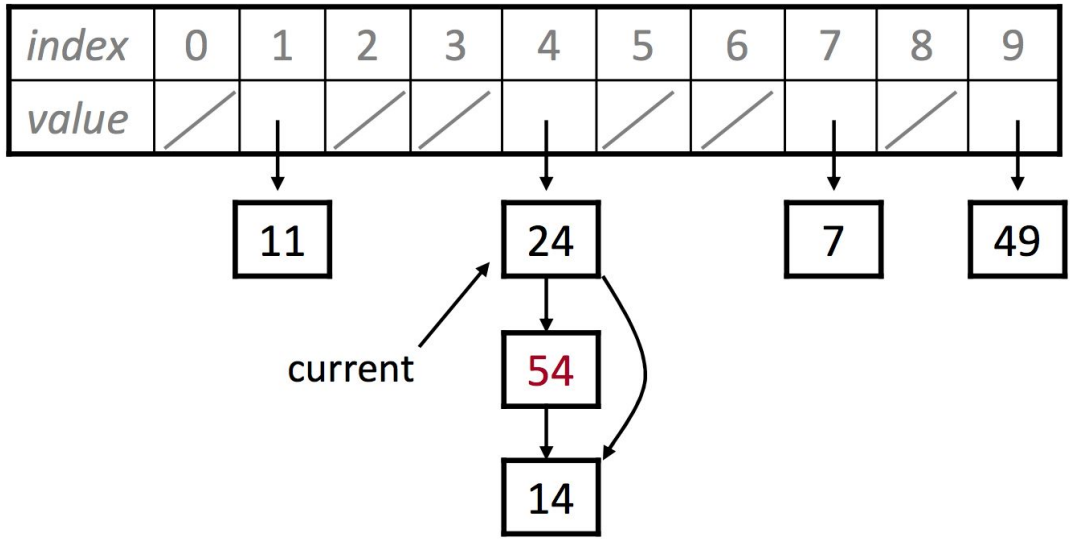
Go to the new element's bucket, and add it to the front of the LinkedList at that index if the element isn't already in that list.

```
set.add(24);
```



Removing from a HashSet

To remove an element, go to its bucket, iterate through the linked list at that bucket, and remove if it exists.




```
set.remove(54);
```

Rehash

To rehash:

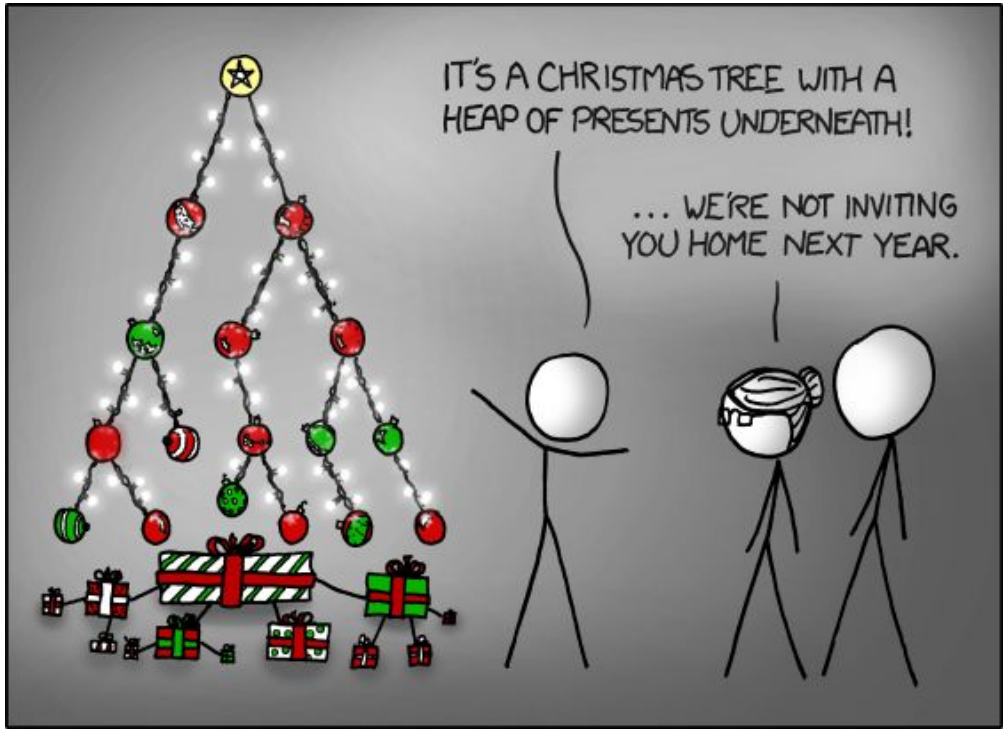
Create a new array with double the capacity of the existing array.

Loop over all elements in existing array and add them to their appropriate buckets in the new array, chaining as necessary.

O(N) to rehash

Generally rehash when the load factor (num elems / capacity) exceeds some threshold.

Trees


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

Binary Trees

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

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)

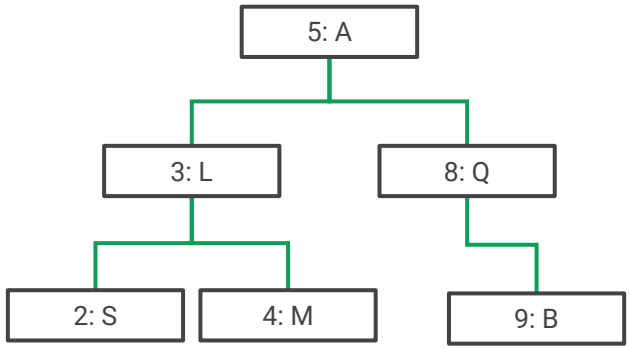
Base case is usually that you've reached NULL

BSTs: Overview

Every node has two children

Every child is the root of a smaller BST!

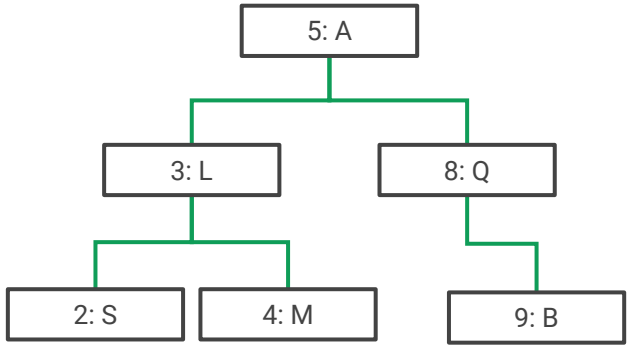
For every node in the tree, its key is greater than every key in the **left** subtree and less than every key in the **right** subtree (no ordering on values)



BSTs: Overview

To implement a Set: orders the tree by the natural ordering of its elements

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



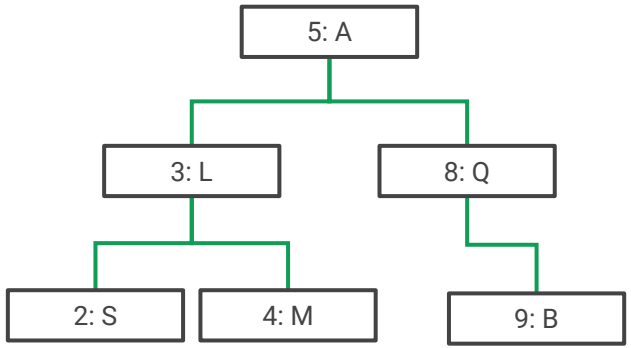
BST Search (e.g. search for 4)

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**



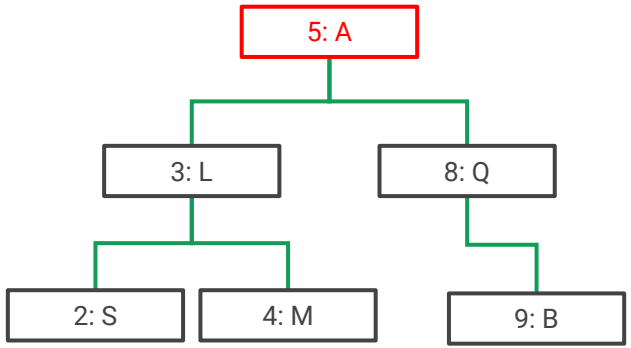
BST Search (e.g. search for 4)

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**



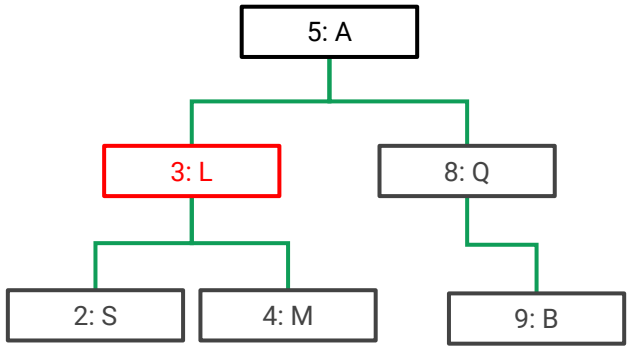
BST Search (e.g. search for 4)

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**



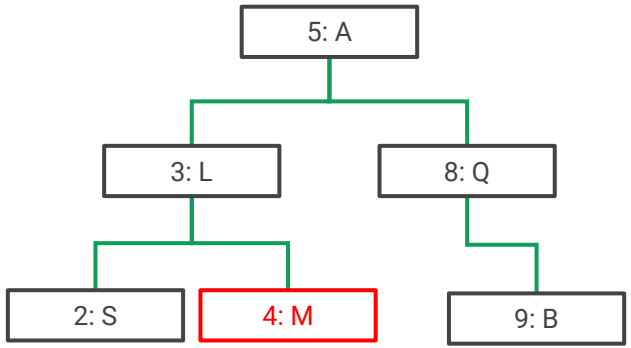
BST Search (e.g. search for 4)

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**

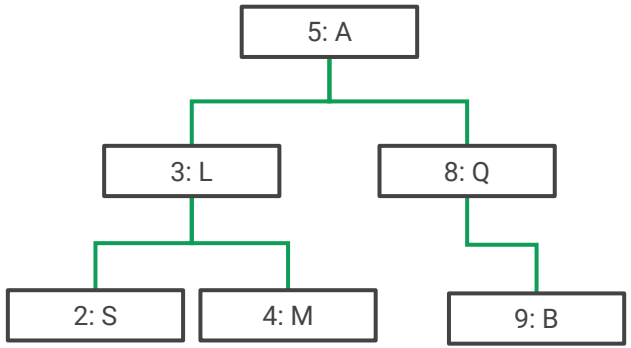


BST Add

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

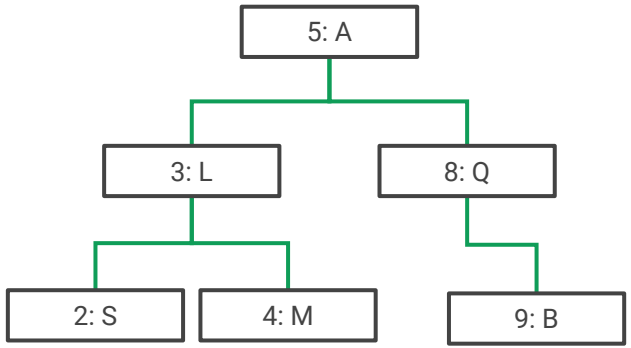


BST Add - (4, P)

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

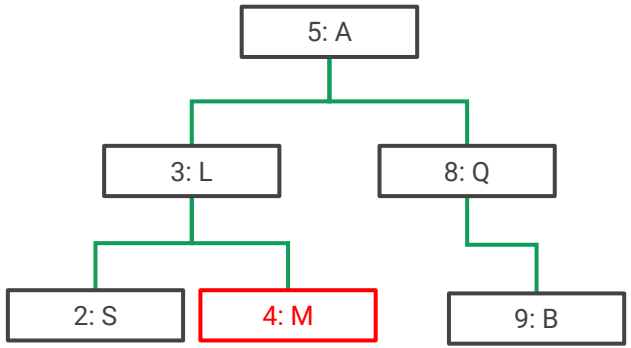


BST Add - (4, P)

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

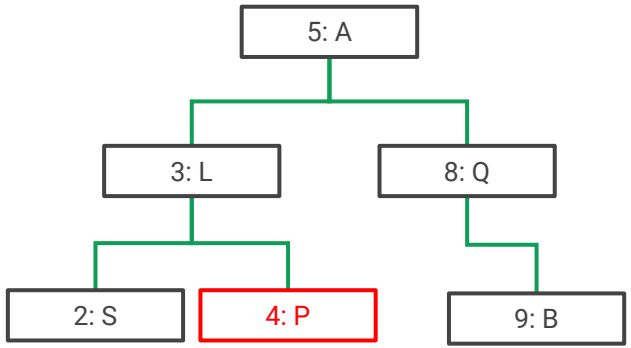


BST Add - (4, P)

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

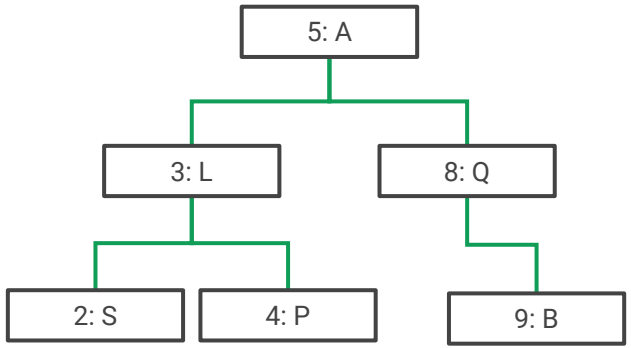


BST Add - (1, P)

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

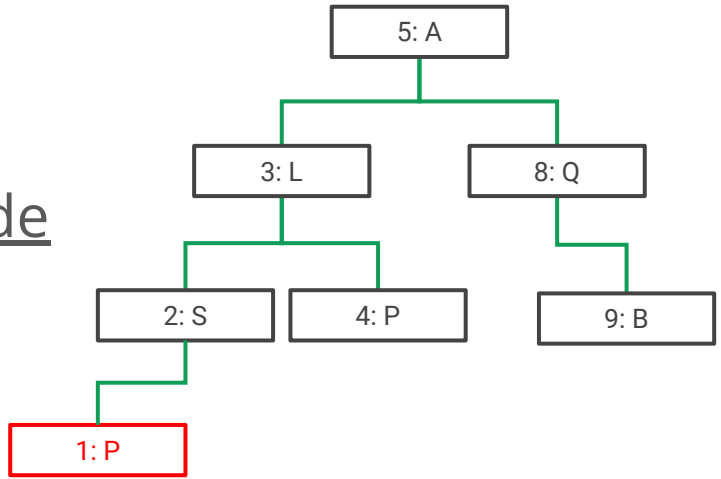


BST Add - (1, P)

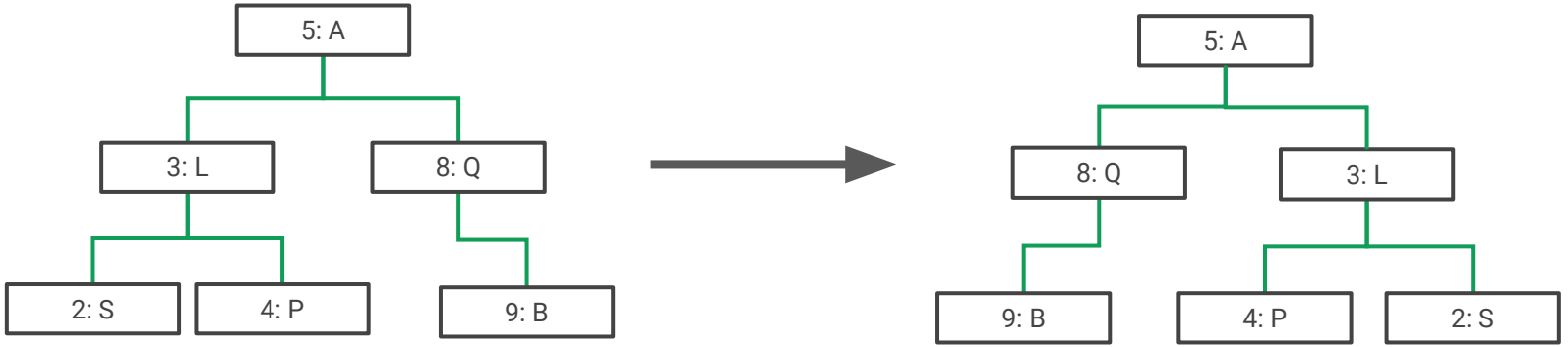
Search for the key

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



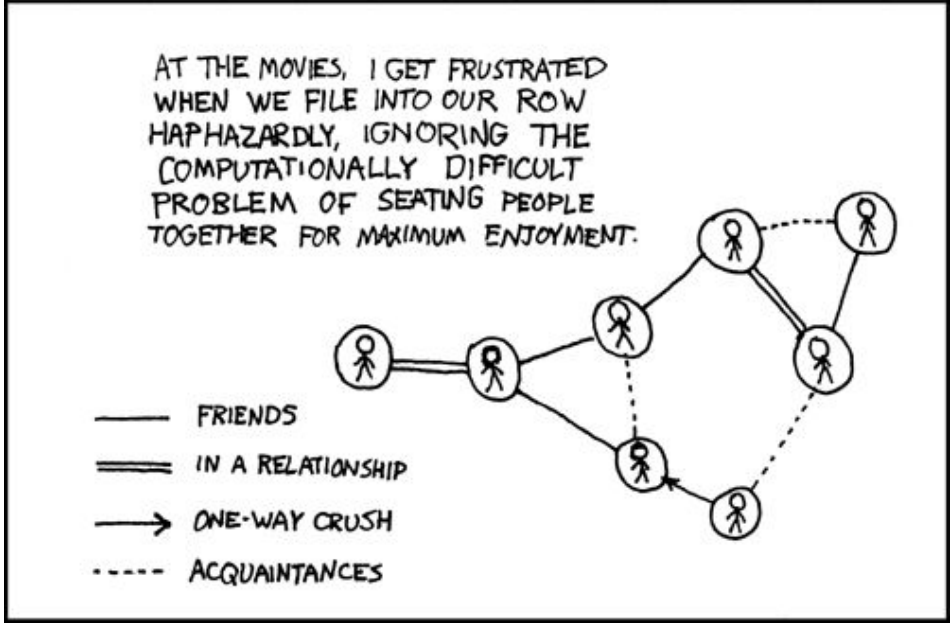
Mirror A BST

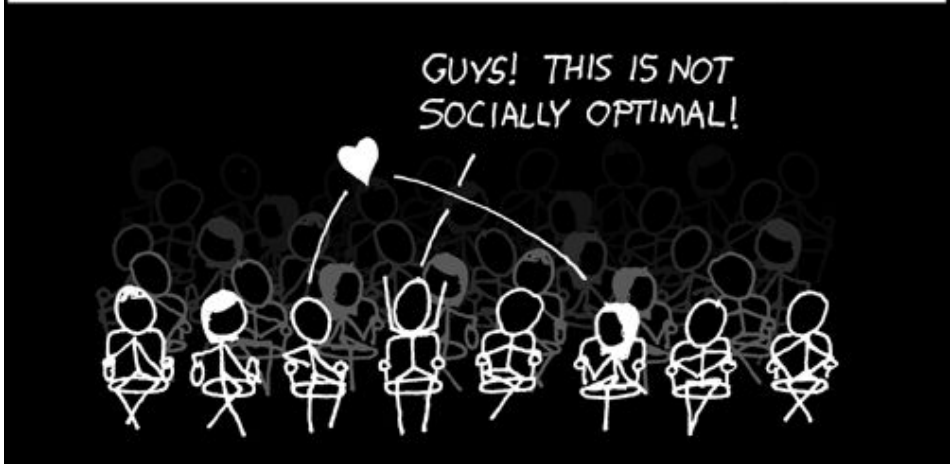


Example: Mirror a BST

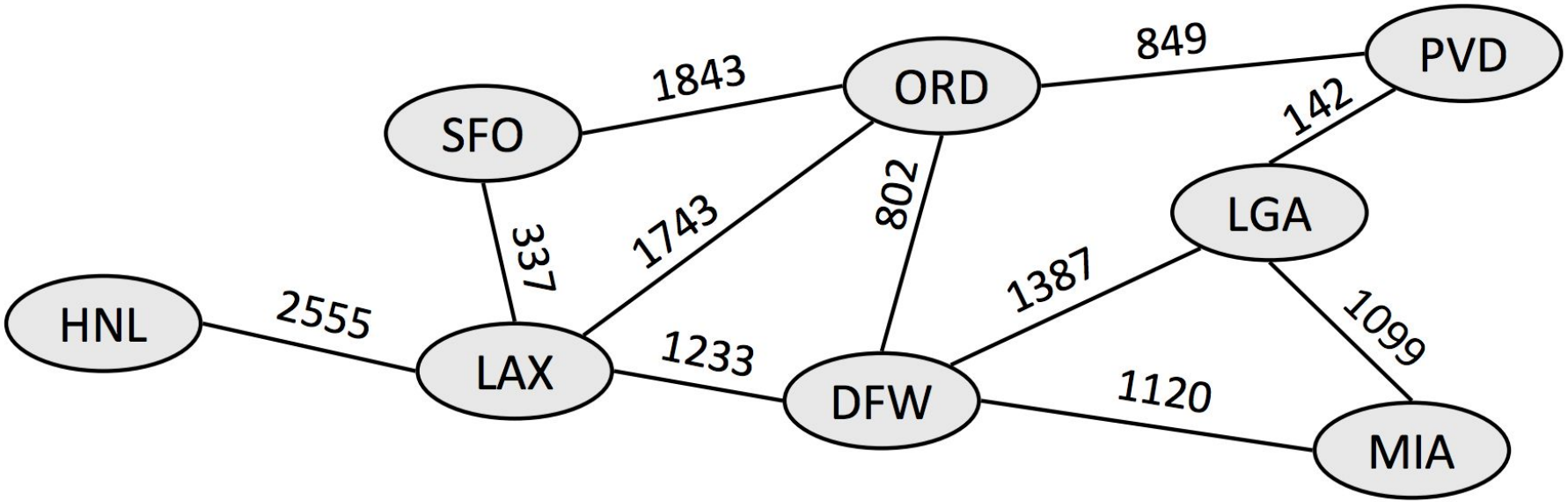
```
void mirror(TreeNode *root) {
   if (root == NULL) {
       return;
   }
   mirror(root->left);
   mirror(root->right);
   TreeNode * temp = root->left;
   root->left = root->right;
   root->right = temp;
}
```

Graphs


AT THE MOVIES, I GET FRUSTRATED WHEN WE FILE INTO OUR ROW HAPHAZARDLY, IGNORING THE COMPUTATIONALLY DIFFICULT PROBLEM OF SEATING PEOPLE TOGETHER FOR MAXIMUM ENJOYMENT.
FRIENDS
IN A RELATIONSHIP
ONE-WAY CRUSH
ACQUAINTANCES


GUYS! THIS IS NOT SOCIALLY OPTIMAL!

Example Graph - Airline Flights



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)
  - Assumes non-negative edge cost
- A*: a modified version of Dijkstra's that uses a heuristic

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)) {
            result++;
            findConnectedComponent(graph, node, visited);
        }
    }
    return result;
}
```

Graphs - numConnectedComponents (DFS)

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

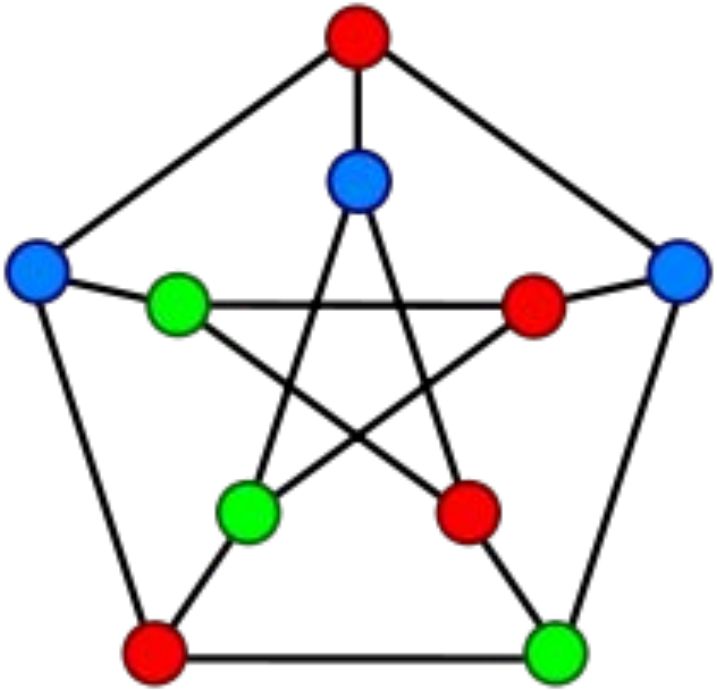
Graphs Tips

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

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

Graphs - Graph Coloring

Given a `BasicGraph`, can we assign each node a color (represented by an `int`) such that no two neighboring vertices share the same color?



Graphs - Graph Coloring

Key realization: For each node, we have a finite number of choices for coloring.

So, try one and see if it works - if it doesn’t, try a different one.

Graphs - Graph Coloring

Key realization: For each node, we have a finite number of choices for coloring.

So, try one and see if it works - if it doesn't, try a different one.

Sounds like backtracking!

Graphs - Graph Coloring

Pseudocode:

```
if all vertices have been assigned a color:
    return true
else:
    choose an un-assigned vertex v
    for each valid color i:
        assign vertex v to color i
        if we can assign the rest of the vertices valid colors:
            return true
        unassign vertex v
    return false
```

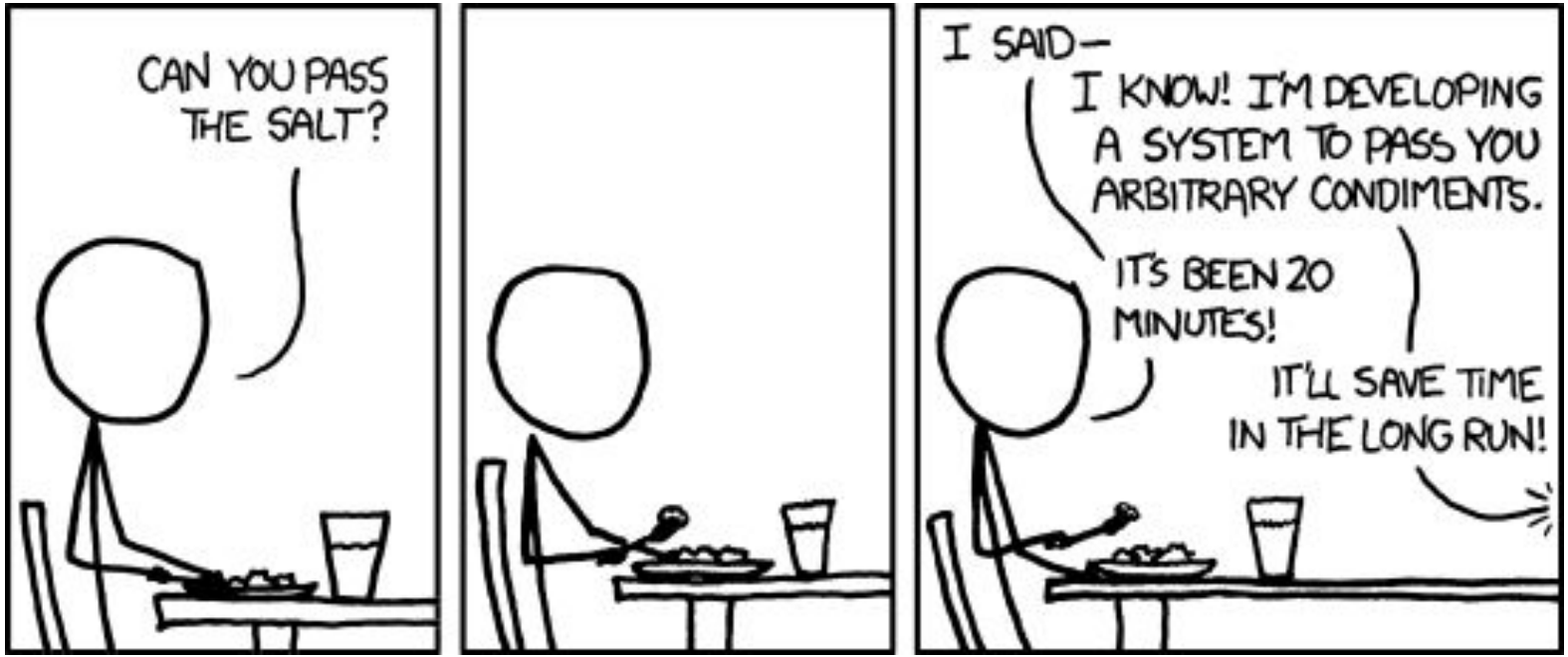
Graphs - Graph Coloring (solution)

```
bool isColorable(const BasicGraph& g, HashMap<Vertex*, int>& colors, int nColors) {
     if(colors.size() == g.getVertexSet().size()) return true;
     Set<Vertex*> remaining = g.getVertexSet() - colors.keys();
     Vertex* cur = remaining.first();
     for(int i = 0; i < nColors; i++) {
          bool valid = true;
          for(Vertex* v : cur->getNeighbors()) {
               if(colors.contains(v) && colors[v] == i) valid = false;
          }
          if(!valid) continue;
          colors[v] = i;
          if(isColorable(g, colors, nColors)) return true;
          colors.remove(v);
     }
     return false;
}
```

Graphs - Graph Coloring (solution cont.)

```
bool isColorable(const BasicGraph& g, int nColors) {
    HashMap<Vertex*, int> colors;
    return isColorable(g, colors, nColors);
}
```

Inheritance



Inheritance/Polymorphism: declaring vs. initializing

```
DeclaredType* var = new InitializedType();
```

What the compiler sees

What var actually is at runtime

Inheritance/Polymorphism: type casting

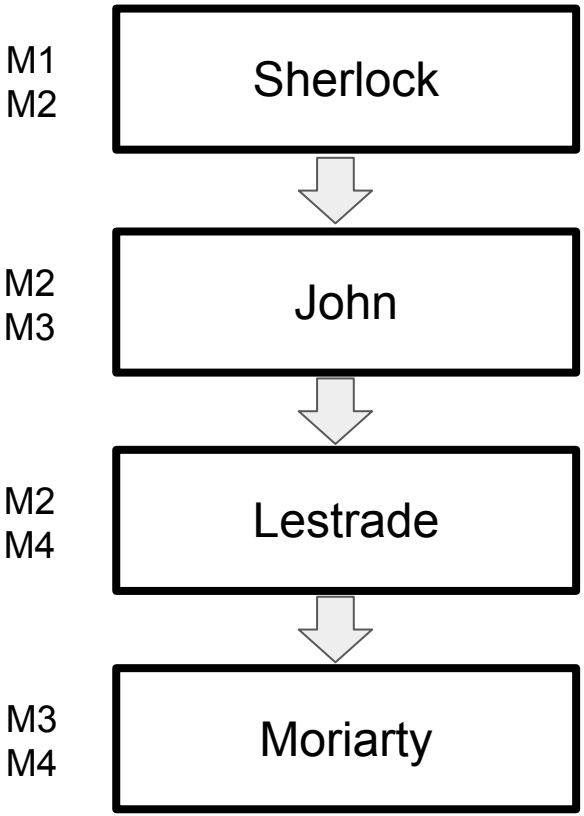
```
DeclaredType* var = new InitializedType();
```

```
((CastType*) var) -> someFunction();
```

Changes what the compiler thinks var is **for this line only**

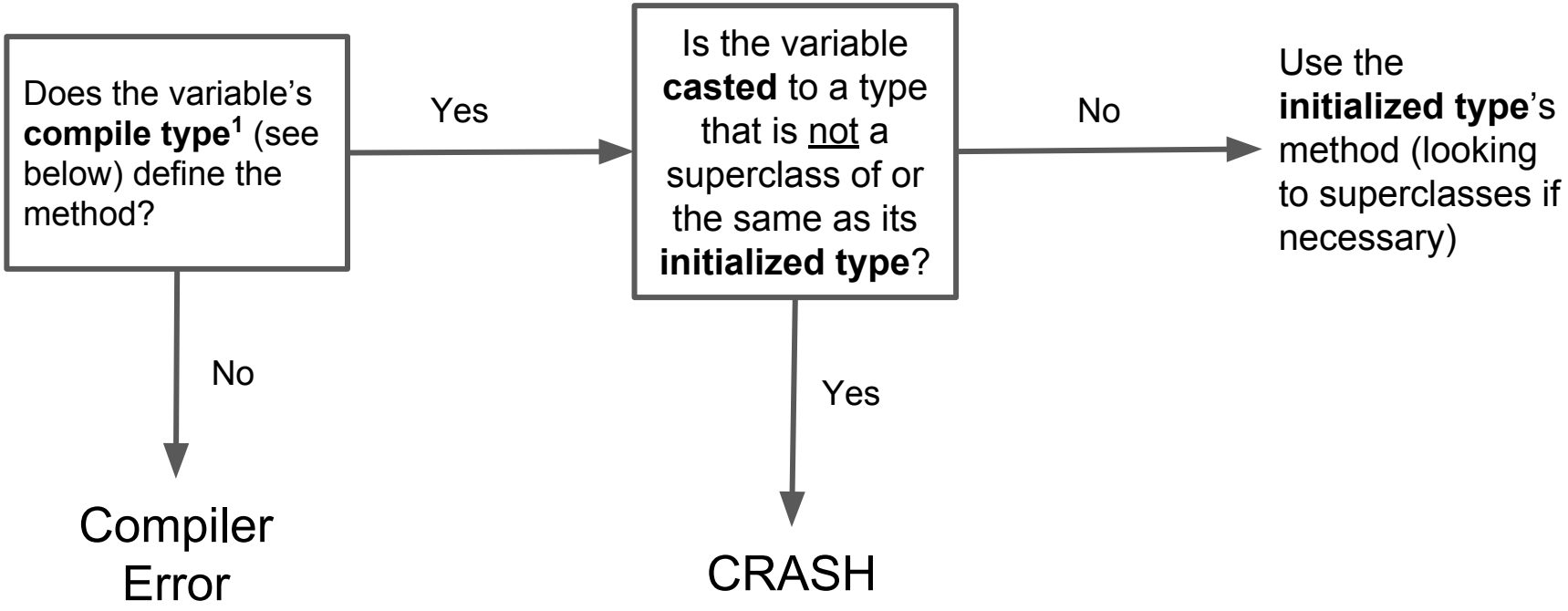
Compiles if CastType has this function; crashes if InitializedType doesn't

Inheritance/Polymorphism - Flow Chart




```
Sherlock* var1 = new John();
var1->m2(); // John's m2
var1->m3(); //COMPILE ERROR
((John*)var1)->m3(); // J's m3
((Lestrade*)var1)->m4();//Crash
Moriarty* var2 = new Moriarty();
((John*)var2)->m3(); // M's m3
```

Inheritance/Polymorphism - Flow Chart



[1] **Compile type** = **declared type** if the variable is not casted
= **cast type** if the variable is casted on that line