

Final Review

Amrita Kaur and Elyse Cornwall

August 15, 2023

Announcements

- Course evals are now open - find these on Canvas
 - This is a chance for you to provide feedback on the instructors and CS106B in general (not your SL)
- We've got our final lecture tomorrow, "Life After CS106B"

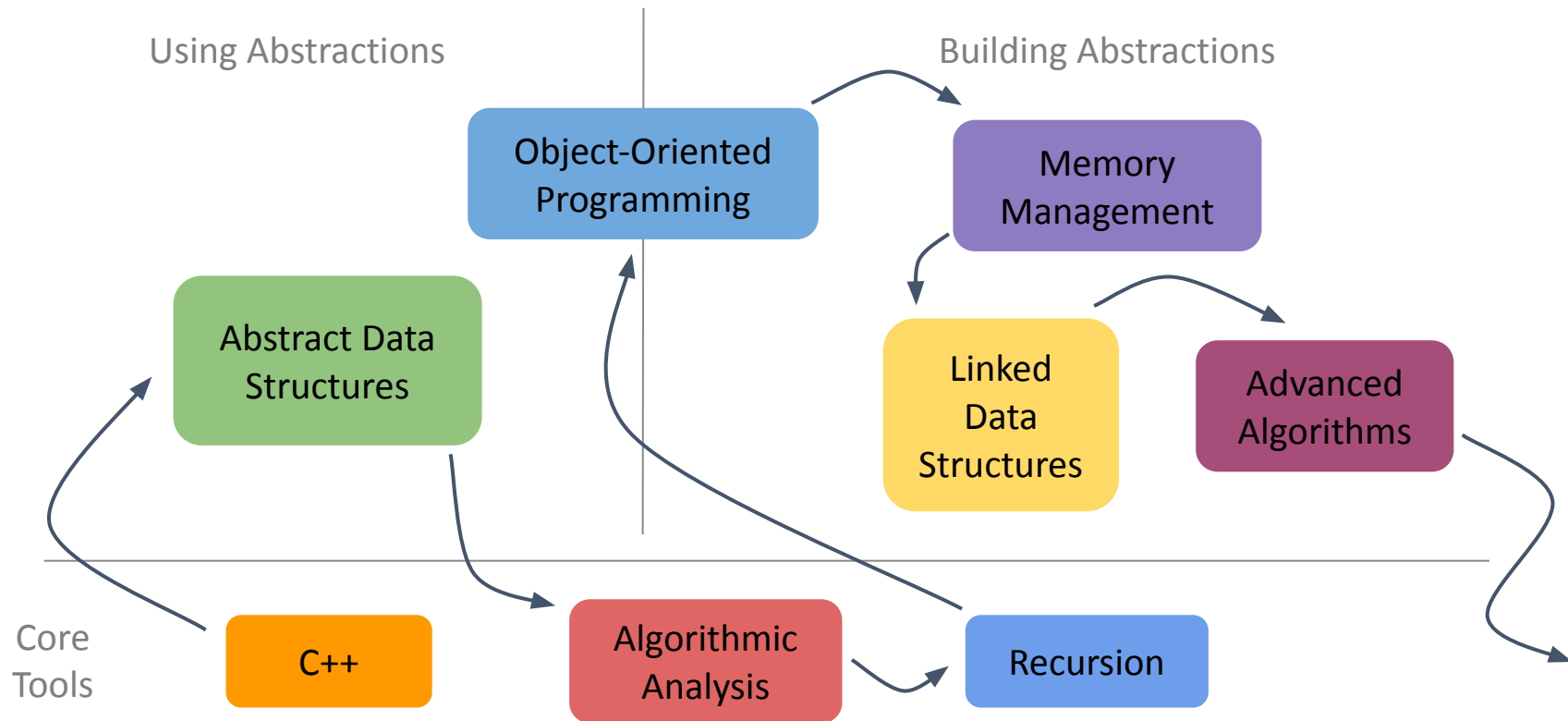
Final Exam Logistics

- 8/18 from 3:30-6:30pm in Hewlett Teaching Center, Room 200
 - Students with exam accommodations have already been contacted
- Same logistics as midterm
 - On paper, using pen/pencil
 - Closed-book and closed-device
 - [Reference sheet](#) on Stanford library functions
 - Notes sheet (one page, front and back, 8-1/2" x 11")
- All information is [here](#)

Final Exam Logistics

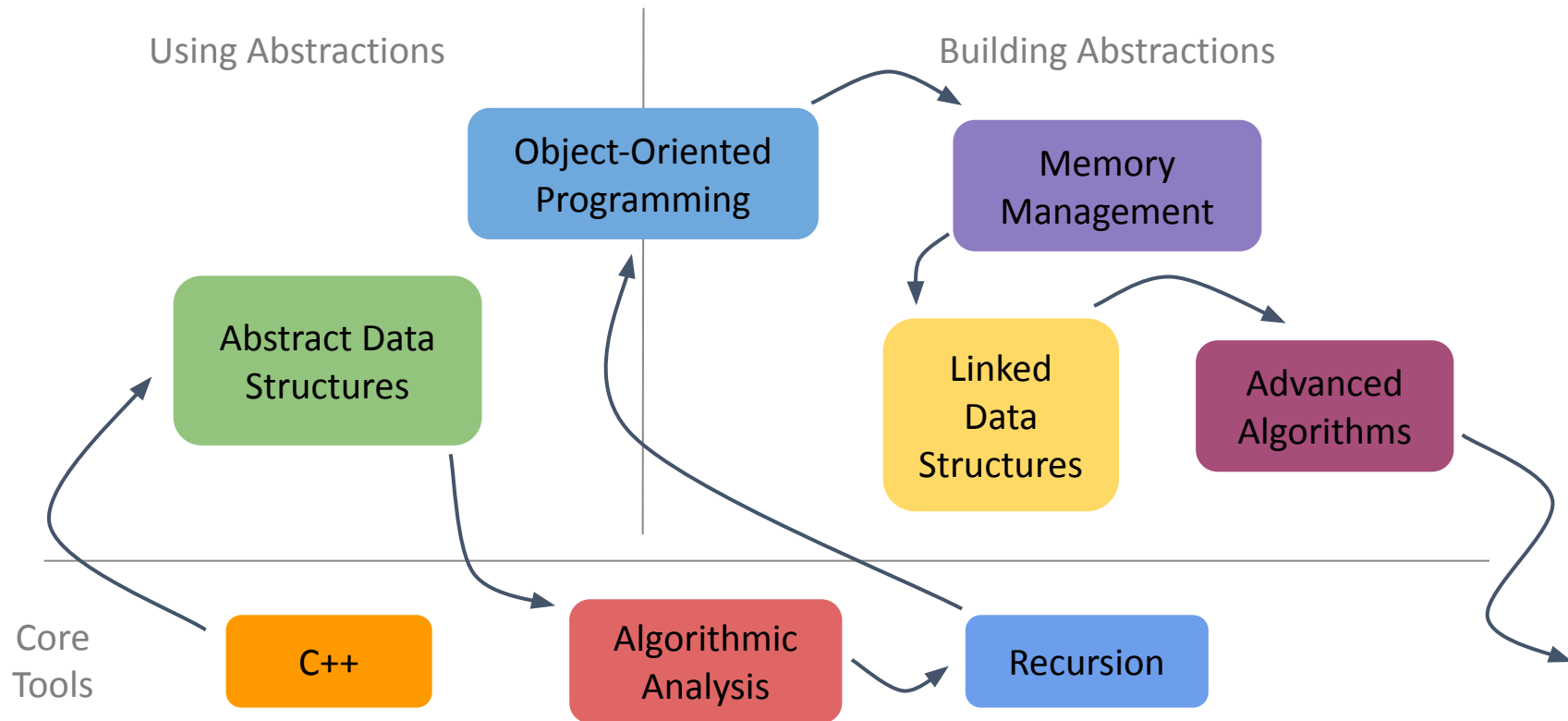
- Evaluate your problem-solving skills and conceptual understanding of the material, not your ability to use perfect syntax
 - Most points awarded for valid approach to solving the problem, fewer points for the minute details of executing your plan
- Mix of different problem types (see practice exams for examples)
- Not taking off points for
 - Missing braces around clearly indented blocks of code
 - Missing semicolons
 - Missing `#include`

Roadmap - Final Coverage



Not covered:
week 8 material

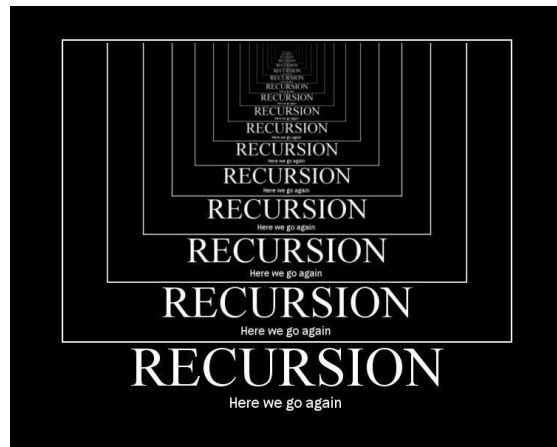
Roadmap - Final Coverage



Recursion and Recursive Backtracking

What is recursion?

- A problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**
- The function calls itself and every time, the problem becomes a little smaller



Two main components

- Base case
 - The simplest version of your problem that all other cases reduce to
 - The point where we've reached our answer
- Recursive case
 - More complex version of the problem that cannot be directly answered
 - Break down the task into smaller occurrences
 - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

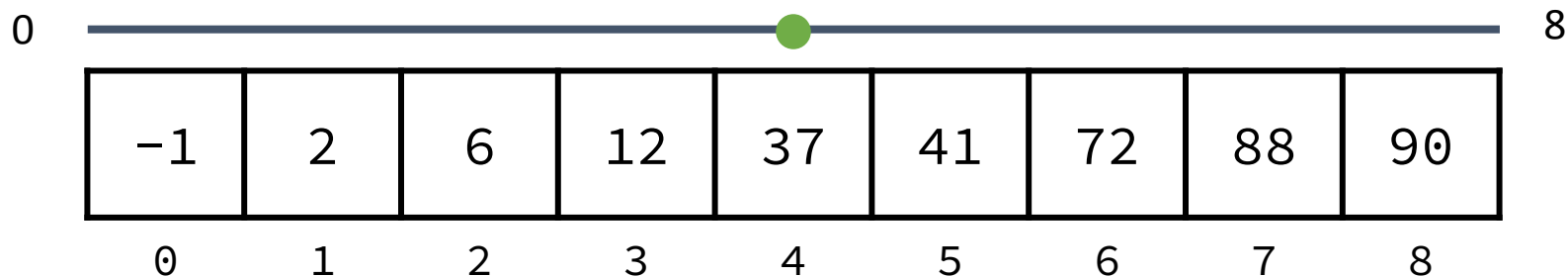
Three “Musts” of Recursion

1. Your code must have a case for all valid inputs
2. You must have a base case that does not make recursive calls
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case

An *efficient* solution: Binary Search

Binary Search

- Let's say we have a sorted Vector of integers

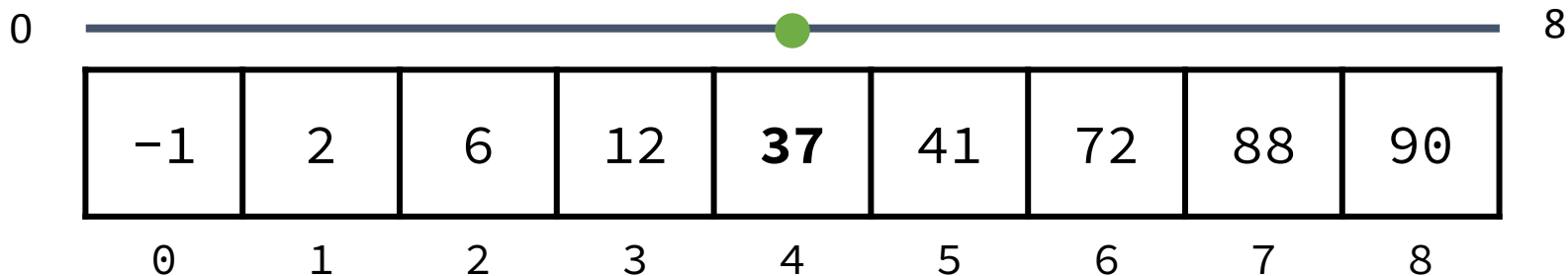


Let's try to find the number 6 in our Vector

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers

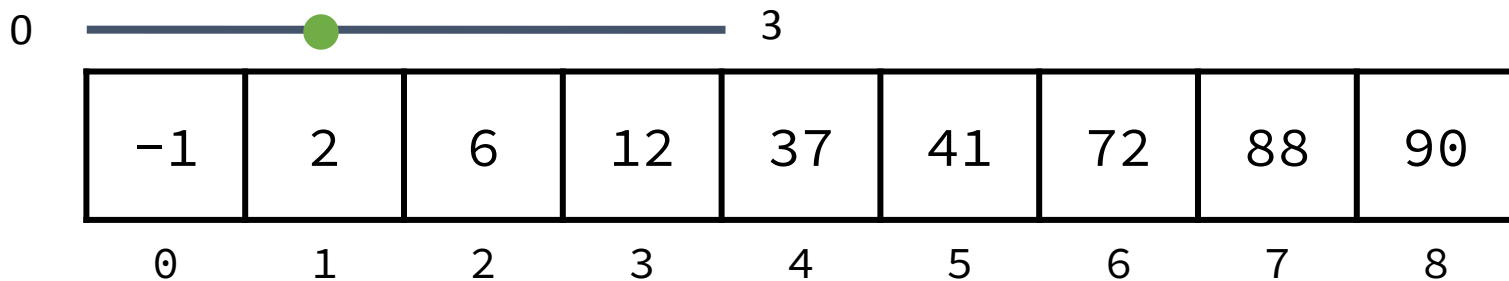


Too big, look left

Binary Search

Let's try to find the number 6

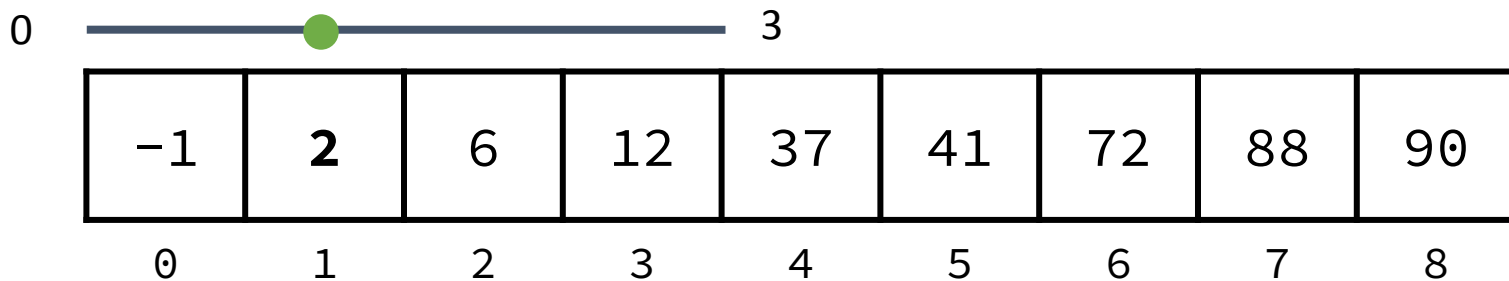
- Let's say we have a sorted Vector of integers



Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers

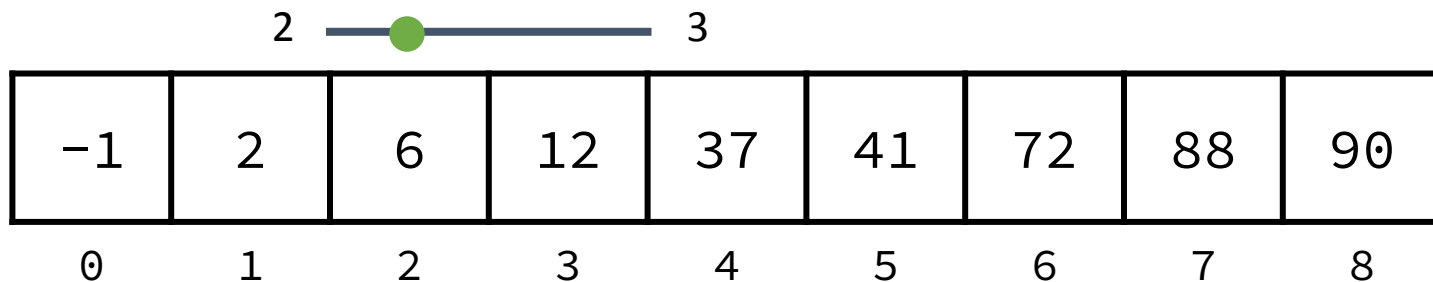


Too small, look right

Binary Search

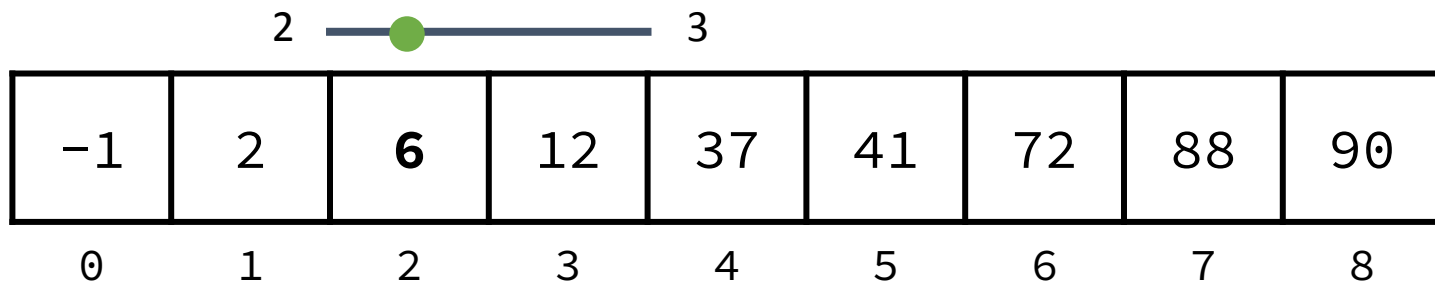
Let's try to find the number 6

- Let's say we have a sorted Vector of integers



Binary Search

- Let's say we have a sorted Vector of integers



Found it! 🎉🎉🎉

Binary Search as a Recursive Process

Binary search over some range of sorted elements:

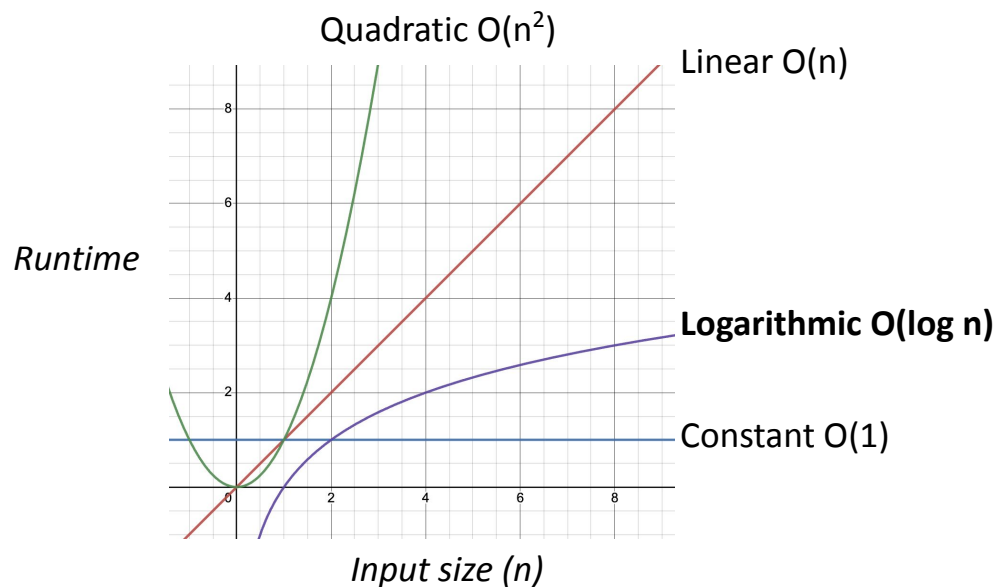
1. Choose element in the middle of the range
2. If this element is our target, success!
3. If element is less than our target, do **binary search** to the right
4. If element is greater than our target, do **binary search** to the left

Binary Search Code

```
int binarySearchHelper(Vector<int>& v, int target, int start, int end) {  
    if (start > end) return -1;    // base case 1: element not in vector  
    int mid = (start + end) / 2;  
    int elem = v[mid];  
    if (elem == target) {          // base case 2: found element  
        return mid;  
    } else if (elem < target) {  
        return binarySearchHelper(v, target, mid + 1, end);  
    } else {  
        return binarySearchHelper(v, target, start, mid - 1);  
    }  
}
```

Runtime of Binary Search

- Binary search has runtime $O(\log n)$
 - Common runtime for algorithms that halve search space at every step

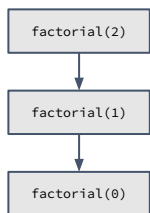


A dynamic solution:
Recursive Backtracking

Two Types of Recursion

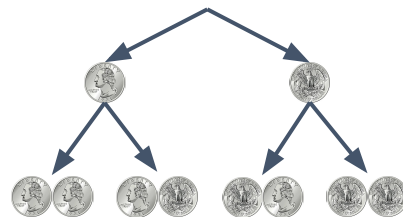
Basic recursion

- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to the recursive function produces the final solution



Backtracking recursion

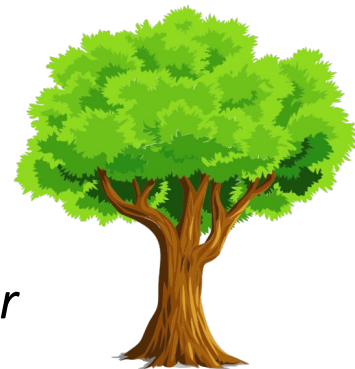
- Build up many possible solutions through multiple recursive calls at each step
- Seed the initial recursive call with an “empty” solution
- At each base case, you have a potential solution



3 Problems to Solve with Backtracking

1. Generate all solutions to a problem or count number of solutions
2. Find one specific solution or prove that one exists
3. Find the best possible solution to a problem

All of these involve exploring many possible solutions, rather than proceeding down a linear path towards one solution.



Solving Recursive Backtracking

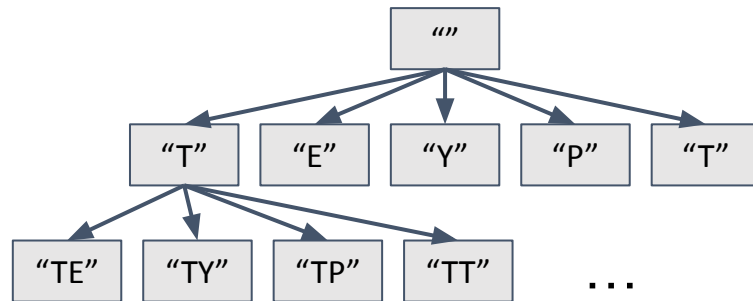
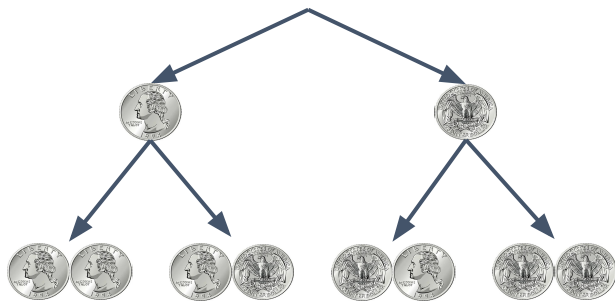
- Which of our three use cases does our problem fall into?
(generate/count all solutions, find one solution/prove its existence,
or pick one best solution)

Solving Recursive Backtracking

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, or pick one best solution)
- What's the provided function prototype and requirements? Do we need a helper function?
 - What are we returning as our solution?
 - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

Patterns

- "Choose / explore / unchoose" pattern in backtracking
- It is important to keep track of the decisions we've made so far and the decisions we have left to make
- Backtracking recursion can have variable branching factors at each level



Word Jumble

- We'd like to print every ordering of "TEYPT" to solve the puzzle
- This is much like coin sequences, but instead of choosing H or T, we are choosing a letter at each step



string sequence

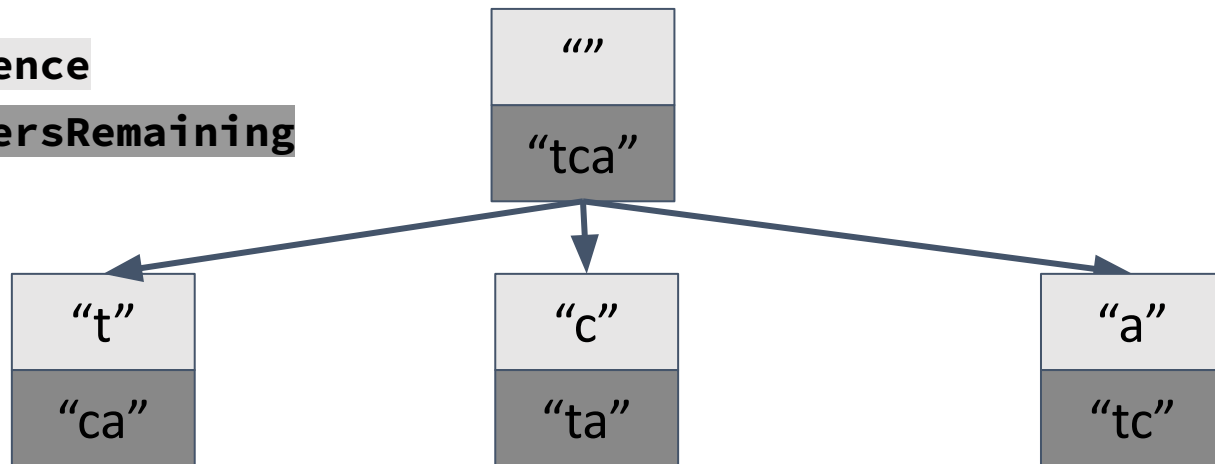
string lettersRemaining

""

"tca"

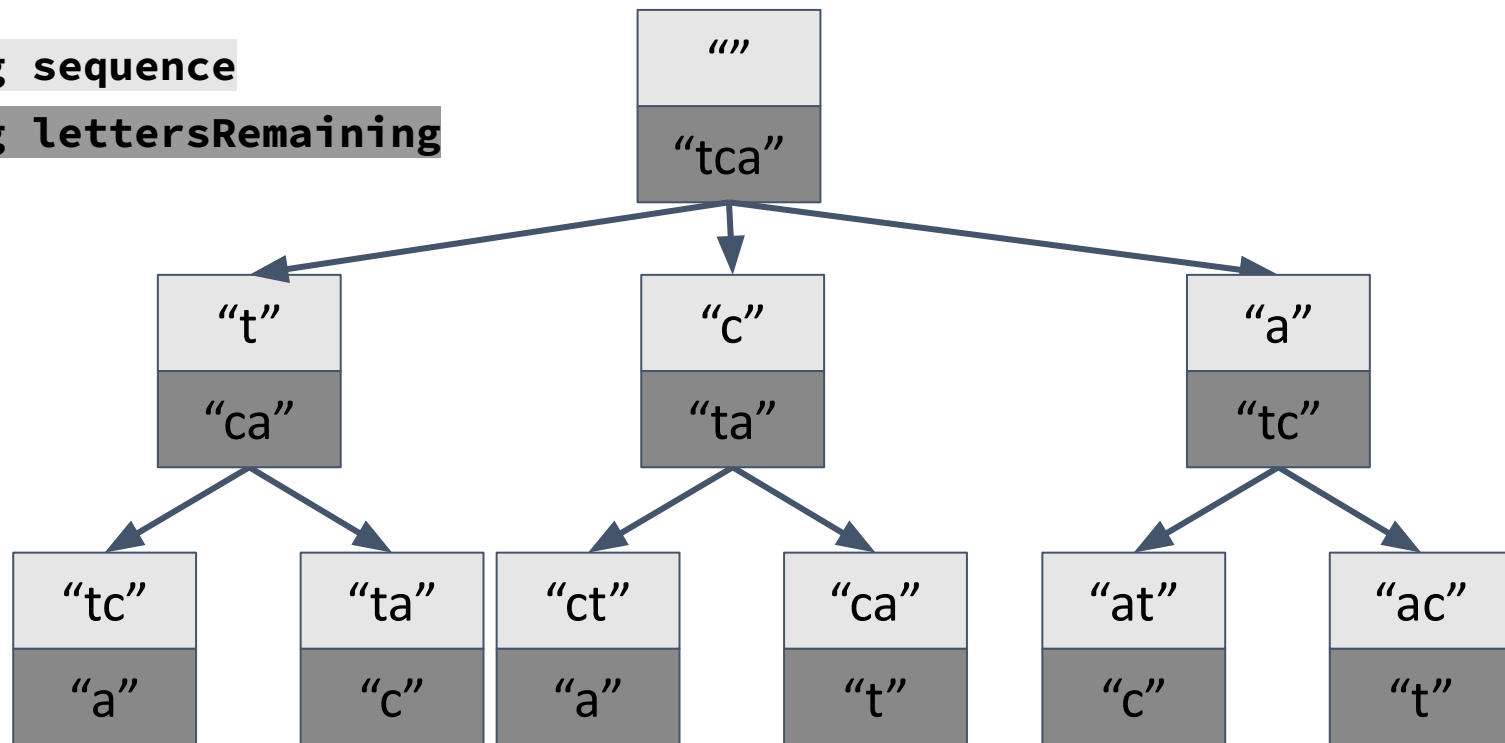
string sequence

string lettersRemaining



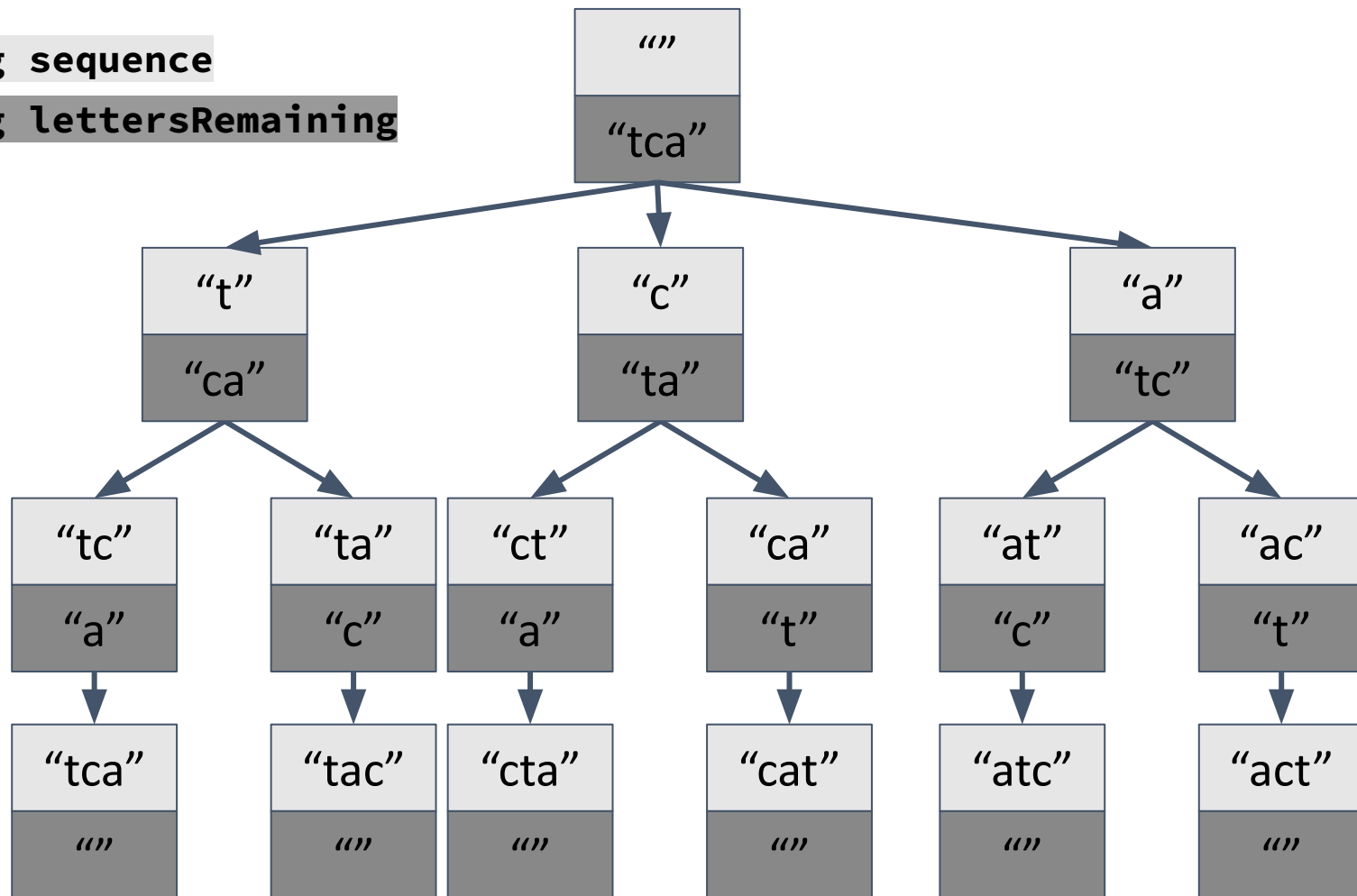
string sequence

string lettersRemaining



string sequence

string lettersRemaining



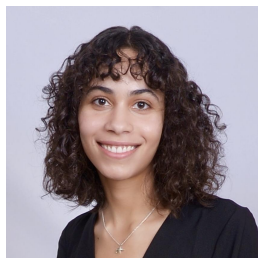
Permutations Solution Code

```
void generatePermutationsHelper(string lettersRemaining, string sequence) {
    // Base case: lettersRemaining = 0, no more letters to choose from
    if (lettersRemaining.length() == 0) {
        cout << sequence << endl;
    } else {
        // Many recursive cases (when lettersRemaining > 0)
        for (int i = 0; i < lettersRemaining.length(); i++) {
            char letter = lettersRemaining[i]; // choose one of our remaining letters to build on sequence
            generatePermutationsHelper(lettersRemaining.substr(0, i) + lettersRemaining.substr(i + 1),
                                   sequence + letter);
        }
    }
}

void generatePermutations(string word) {
    generatePermutationsHelper(word, "");
}
```

Subsets

Given a group of people, generate all possible teams, or subsets, of these people:



`{}`

`{"Amrita"}`

`{"Elyse"}`

`{"Taylor"}`

`{"Amrita", "Elyse"}`

`{"Amrita", "Taylor"}`

`{"Elyse", "Taylor"}`

`{"Amrita", "Elyse", "Taylor"}`

Making a Decision Tree

- Decision at each step (each level of the tree)
 - Are we going to include a given element in our subset?
- Options at each decision (branches from each node)
 - Include the element
 - Don't Include the element
- Information you need to store along the way
 - Set you've built so far
 - Remaining elements in original set

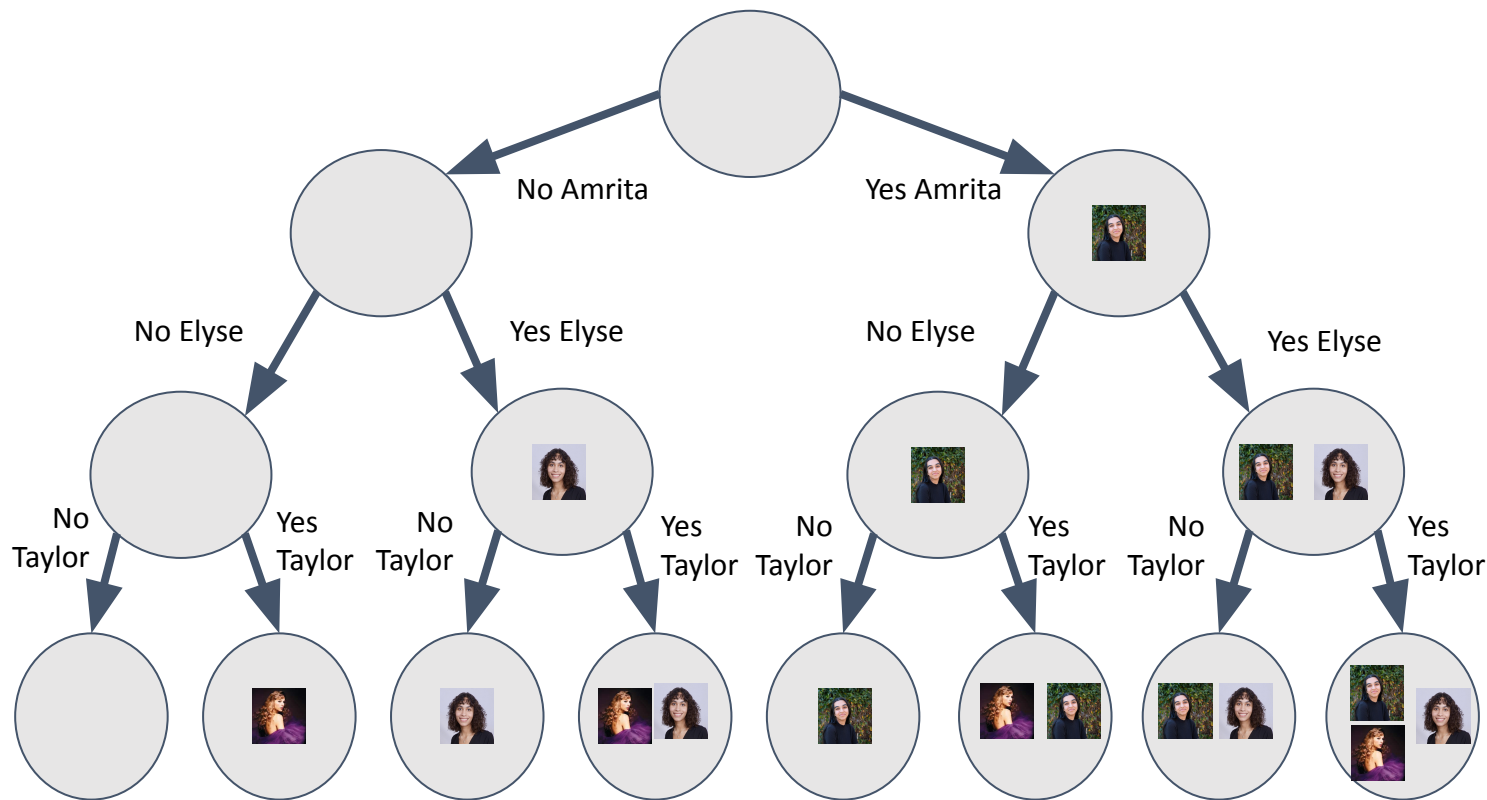
Remaining Elements:

{“Amrita”,
“Elyse”,
“Taylor”}

{“Elyse”,
“Taylor”}

{“Taylor”}

{ }



Subsets Solution Code

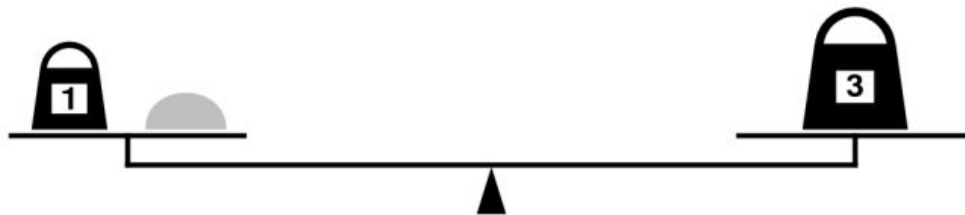
```
void listSubsetsHelper(Set<string>& remaining, Set<string>& chosen) {  
    // (base case omitted for space)  
    // choose  
    string elem = remaining.first();  
    remaining = remaining - elem;  
    // explore  
    listSubsetHelper(remaining, chosen);  
    chosen = chosen + elem  
    listSubsetHelper(remaining, chosen);  
    // unchoose by adding it back to possible choices  
    chosen = chosen - elem;  
    remaining = remaining + elem;  
}
```

Choose / explore / unchoose

- Implicit “unchoose” step
 - Pass by value; usually when memory constraints aren’t an issue
 - Works because you’re making edits to a copy
 - E.g. Building up a string over time
- Explicit “unchoose” step
 - Uses pass by reference; usually with large data structures
 - “Undoing” prior modifications to structure
 - E.g. Generating subsets (one set passed around by reference to track subsets)

Practice Problem: Weights

Problem 6 from Section 4 ([see description](#))



```
bool isMeasurable(int target, Vector<int>& weights) {  
  
}
```

Practice Problem: Weights (Solution)

```
bool isMeasurable(int target, Vector<int>& weights) {  
    if (weights.isEmpty()) {  
        return target == 0; // base case; no weights left to place  
    } else {  
        // choose  
        int last = weights[weights.size() - 1]; // just because removing last index is faster  
        weights.remove(weights.size() - 1);  
        // explore  
        bool result = isMeasurable(target + last, weights) || isMeasurable(target - last, weights)  
                     || isMeasurable(target, weights);  
        // un-choose  
        weights.add(last);  
        return result;  
    }  
}
```

Classes / OOP

Class

- Defines a new data type for our program to use
- Help us create types of objects
 - Which is why we call this object-oriented programming!

What is a class?

- The main difference between structs and classes are the encapsulation defaults
 - Struct defaults to **public** members (accessible outside the struct itself).
 - Class defaults to **private** members (accessible only inside the class implementation).

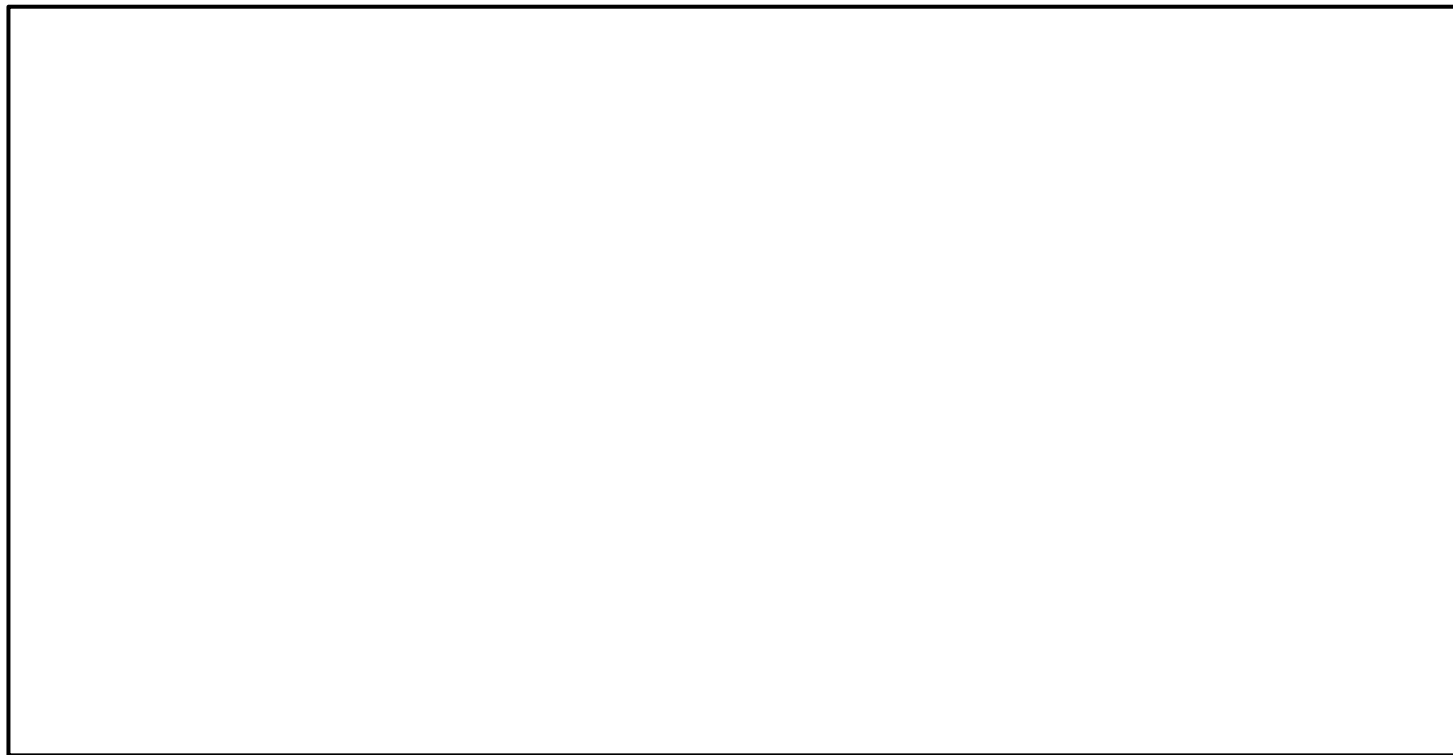
Creating C++ Class

- Defining a class in C++ (typically) requires two steps:
 - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
 - Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class.
- Clients of the class can then include (using the `#include` directive) the header file to use the class.

Three Main Parts

- Member variables (*What subvariables make up this new variable type?*)
 - These are the variables stored within the class
 - Usually not accessible outside the class implementation
- Member functions (*What functions can you call on a variable of this type?*)
 - Functions you can call on the object
 - Known as methods
- Constructor (*What happens when you make a new instance of this type?*)
 - Gets called when you create the object
 - Sets the initial state of each new object

What is in a header file?



What is in a header file?

```
#pragma once
```

```
class RandomBag {
```

```
};
```

This is a **class definition**. We're creating a new class called RandomBag. Like a struct, this defines the name of a new type that we can use in our programs.

When naming classes, use UpperCamelCase.

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:
```

```
private:
```

```
};
```

Interface
(What it looks like)

Implementation
(How it works)

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:
```

```
private:
```

```
};
```

The **public interface** specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the Vector
.add() function or the string's
.find().

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:
```

```
private:
```

```
};
```

The **public interface** specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the Vector
.add() function or the string's
.find().

The **private implementation** contains information that objects of this class type will need in order to do their job properly. This is invisible to people using the class.

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    int size();  
};
```

These are **member functions** of the RandomBag class. They're functions you can call on objects of type RandomBag.

All member functions must be defined in the class definition. We'll implement these functions in the C++ file.

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    int size();  
};
```

Member functions of a class can be **public** or **private**, depending on if you want a client to be able to access the functionality.

What is in a header file?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    bool isEmpty();
private:
    Vector<int> elems;
    int size();
};
```

This is a **member variable** of the class. This tells us how the class is implemented. Internally, we're going to store a `Vector<int>` holding all the elements. The only code that can access or touch this `Vector` is the `RandomBag` implementation

What is in a header file?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    bool isEmpty();
private:
    Vector<int> elems;
    int size();
};
```

Member variables of a class can be **public** or **private**. You should default towards member variables being private if possible.

What is in a header file?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    bool isEmpty();
private:
    Vector<int> elems;
    int size();
};
```

```
#include "RandomBag.h"
```

If we're going to implement the RandomBag type, the .cpp file needs to have the class definition available. All implementation files need to include the relevant headers.

```
#include "RandomBag.h"
```

```
void RandomBag::add(int value){  
    elems.add(value);  
}
```

```
#pragma once  
#include "vector.h"  
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    Vector<int> elems;  
    int size();  
};
```

```
#include "RandomBag.h"
```

```
void RandomBag::add(int value){  
    elems.add(value);  
}
```

The syntax **RandomBag::add** means “the add function defined inside of RandomBag.” The **::** operator is called the **scope resolution operator** in C++ and is used to say where to look for things.

```
#pragma once  
#include "vector.h"  
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    Vector<int> elems;  
    int size();  
};
```

```
#include "RandomBag.h"
```

```
void RandomBag::add(int value){  
    elems.add(value);  
}
```

If we had written something like this instead, then the compiler would think we were just making a free function named add that has nothing to do with RandomBag's version of add. That's an easy mistake to make!

```
#pragma once  
#include "vector.h"  
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    Vector<int> elems;  
    int size();  
};
```

```
#include "RandomBag.h"
```

```
void RandomBag::add(int value){  
    elems.add(value);  
}
```

We don't need to specify where `elems` is. The compiler knows that we're inside `RandomBag`, and so it knows that this means "the current `RandomBag`'s collection of elements."

Using the scope resolution operator is like passing in an invisible parameter to the function to indicate what the current instance is.

```
#pragma once  
#include "vector.h"  
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
    bool isEmpty();  
private:  
    Vector<int> elems;  
    int size();  
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    bool isEmpty();
private:
    Vector<int> elems;
    int size();
};
```

```

#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}

```

This code calls our own `size()` function. The class implementation can use the public or private interface.

```

#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    bool isEmpty();
private:
    Vector<int> elems;
    int size();
};

```

Constructor

- Specially defined method for classes that initializes the state of new objects as they are created
 - Often accepts parameters for the initial state of the fields.
 - Special naming convention defined as `ClassName()`
 - You can never directly call a constructor, but one will always be called when declaring a new instance of an object

```
// MyClass.h
class MyClass {
public:
    MyClass();

    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.h
class MyClass {
public:
    MyClass();

    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

...
```

```
// MyClass.h
class MyClass {
public:
    MyClass();

    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

...
```

```
// main.cpp
int main() {
    MyClass firstInstance;
}
```

```
// MyClass.h
class MyClass {
public:
    MyClass();
    MyClass(int var1, int var2);
    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

...
```

```
// main.cpp
int main() {
    MyClass firstInstance;
}
```

```
// MyClass.h
class MyClass {
public:
    MyClass();
    MyClass(int var1, int var2);
    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

MyClass::MyClass(int var1, int var2) {
    this->var1 = var1;
    this->var2 = var2;
}
...
```

```
// main.cpp
int main() {
    MyClass firstInstance;
}
```

```
// MyClass.h
class MyClass {
public:
    MyClass();
    MyClass(int var1, int var2);
    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);

private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

MyClass::MyClass(int var1, int var2) {
    this->var1 = var1;
    this->var2 = var2;
}
...
```

```
// main.cpp
int main() {
    MyClass firstInstance;
    MyClass secInstance(3, 4);

}
```

Destructor

- Specially defined method for classes
 - Special naming convention defined as `~ClassName()`
- Does not take in parameters and does not return anything
- Automatically called when the object's lifetime ends (for example, if it's a local variable that goes out of scope)
- Responsible for cleaning up an object's memory

```
// MyClass.h
class MyClass {
public:
    MyClass();
    MyClass(int var1, int var2);
    ~MyClass();
    returnType func1(parameters);
    returnType func2(parameters);
    returnType func3(parameters);
private:
    int var1;
    int var2;
    type func4();
};
```

```
// MyClass.cpp
MyClass::MyClass() {
    var1 = 1;
    var2 = 1;
}

MyClass::MyClass(int var1, int var2) {
    this->var1 = var1;
    this->var2 = var2;
}
...
```

```
// main.cpp
int main() {
    MyClass firstInstance;
    MyClass secInstance(3, 4);

}
```

Memory and Pointers

How is computer memory organized?

- Memory in your computer is just a giant array!
 - Can think of it as a long row of boxes, with each box having a value in it and an associated index



- How can we communicate with the computer to find exactly which box we want to access/store information in?
 - We'll give each box an associated numerical location, called a memory address

Memory on Stack vs Heap

```
Vector<string> varOnStack;
```

- Before 106B, all variables we've created get defined on the **stack**
- This is static memory allocation
- Variables on the stack are stored directly to the memory and access to this memory is very fast
- We don't have to worry about memory management

Memory on Stack vs Heap

```
Vector<string> varOnStack;
```

- Before 106B, all variables we've created get defined on the **stack**
- This is static memory allocation
- Variables on the stack are stored directly to the memory and access to this memory is very fast
- We don't have to worry about memory management

```
string* arr = new string[numValues];
```

- We can now request memory from the **heap**
- This is dynamic memory allocation
- We have more control over variables on the heap
- But this means that we also have to handle the memory we're using carefully and properly clean it up when done

Dynamic Memory Allocation: new

- To request memory from the heap to allocate one element:

```
type* variable = new type;
```

- To allocate multiple (n) elements on the heap:

```
type* variable = new type[n];
```

Dynamic Memory Allocation: new

```
type* variable = new type;
```

The diagram illustrates the components of the `new` statement. A bracket under `type*` and `variable` is linked by an arrow to the text 'Declaring a variable that will point at our newly-allocated memory'. Another bracket under `new` and `type` is linked by an arrow to the text 'Allocating heap memory with the new keyword'. A third arrow points from the text 'Assigning the pointer to point to the heap memory' to the equals sign between the variable and the allocation.

Declaring a variable that will point at our newly-allocated memory

- Name is `variable`
- Type is `type*` (match the type of the element)

Allocating heap memory with the new keyword

Assigning the pointer to point to the heap memory

Pointer

- Data type that allows us to work directly with computer memory addresses
- Just like all other data types, pointers take up space in memory and store specific values
- Always stores a **memory address**, telling us where in the computer to look for a certain value
- They quite literally "point" to another location on your computer

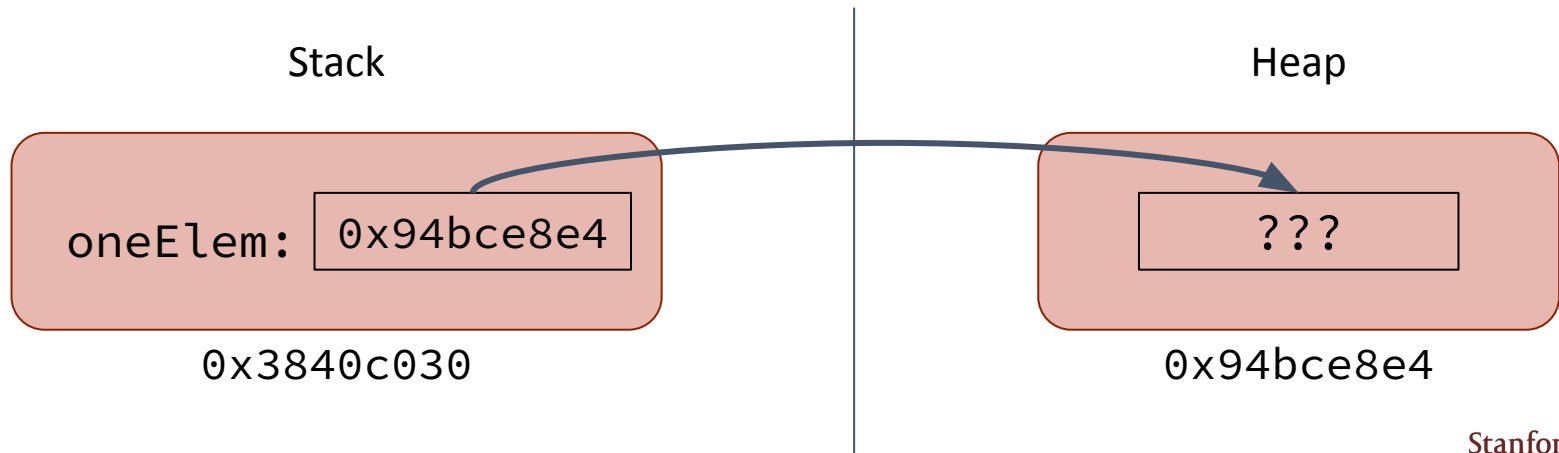
What is a pointer?

A memory address!!

Pointer Syntax

- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap)

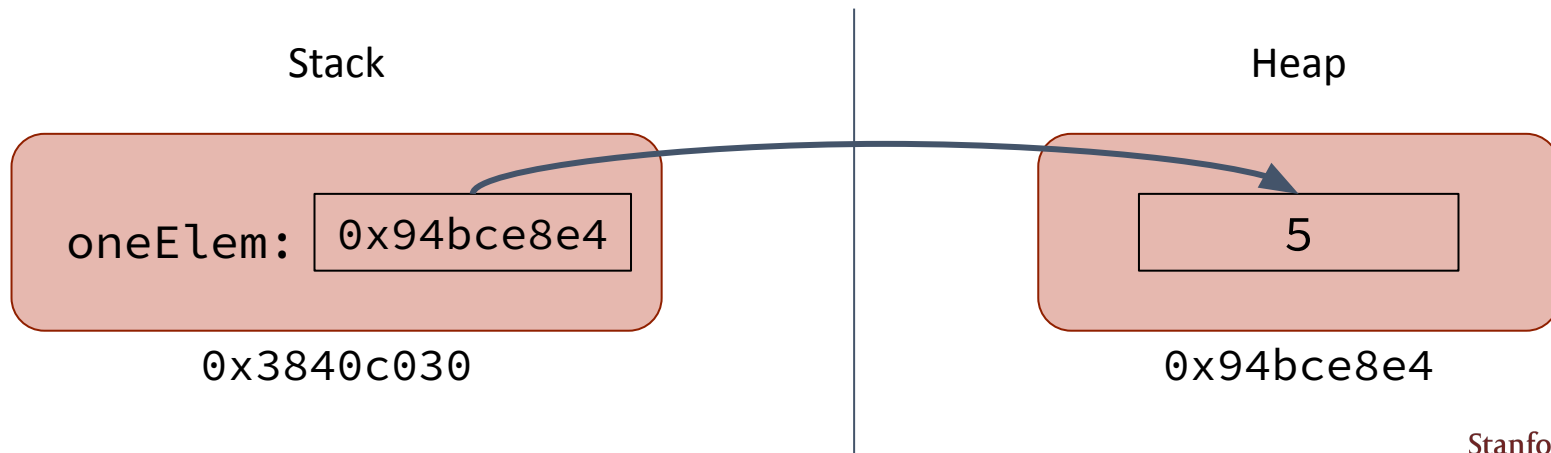
```
int* oneElem = new int;
```



Pointer Syntax

- To read or modify the variable that a pointer points to, we use the `*` (asterisk) operator (in a different way than before!)
- Known as **dereferencing the pointer**
- Follow the arrow to the memory location

`*oneElem = 5;`



nullptr

- When we declare/initialize a pointer but don't have anything to point it at yet, that can be dangerous and unpredictable
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `nullptr`, which means "no valid address"

```
string* showPtr = nullptr;
```

showPtr:



0x35efcdf8

nullptr

- How can we tell if a pointer is safe to use (dereference)?
- If you are unsure if your pointer holds a valid address, you should check for `nullptr`!

```
void printShowName(string* showPtr) {  
    if (showPtr != nullptr) {  
        cout << *showPtr << endl; // prints out the value pointed to by showPtr  
        // if it is not nullptr  
    } else {  
        cout << "showPtr is not valid!" << endl;  
    }  
}
```






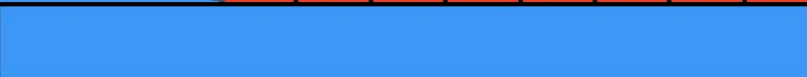
Under the Hood

```
int* tenInts = new int[10];
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X	X	X	X				X	X	X	X	X	X	X	X	X
1			X	X	X	X	X	X	X	X	X	X				
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X												
4	X								X	X	X	X	X	X	X	X
5	X	X	X	X	X											

Under the Hood

```
int* tenInts = new int[10];
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X	X	X	X				X	X	X	X	X	X	X	X	X
1				X	X	X	X	X	X	X	X	X				
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X												
4	X								X	X	X	X	X	X	X	X
5	X	X	X	X	X											

Under the Hood

```
int* tenInts = new int[10];
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Pitfalls and Dangers

- The array you get from `new[]` is **fixed-size**: it can neither grow nor shrink once it's created
- The array you get from `new[]` has **no bounds-checking**: accessing anything past the beginning or end of an array triggers undefined behavior

Cleaning Up

- When declaring local variables or parameters, C++ automatically handles memory allocation and deallocation for you
- When using `new`, you are responsible for deallocating the memory you allocate
- If you don't, you get a memory leak
 - Your program will never be able to use that memory again
 - Too many leaks can cause a program to crash – it's important to not leak memory!

Cleaning Up: delete

- You can deallocate (free) memory with the `delete` keyword
- To deallocate a single element:

```
delete var;
```

- To deallocate an array of elements:

```
delete[] arr;
```

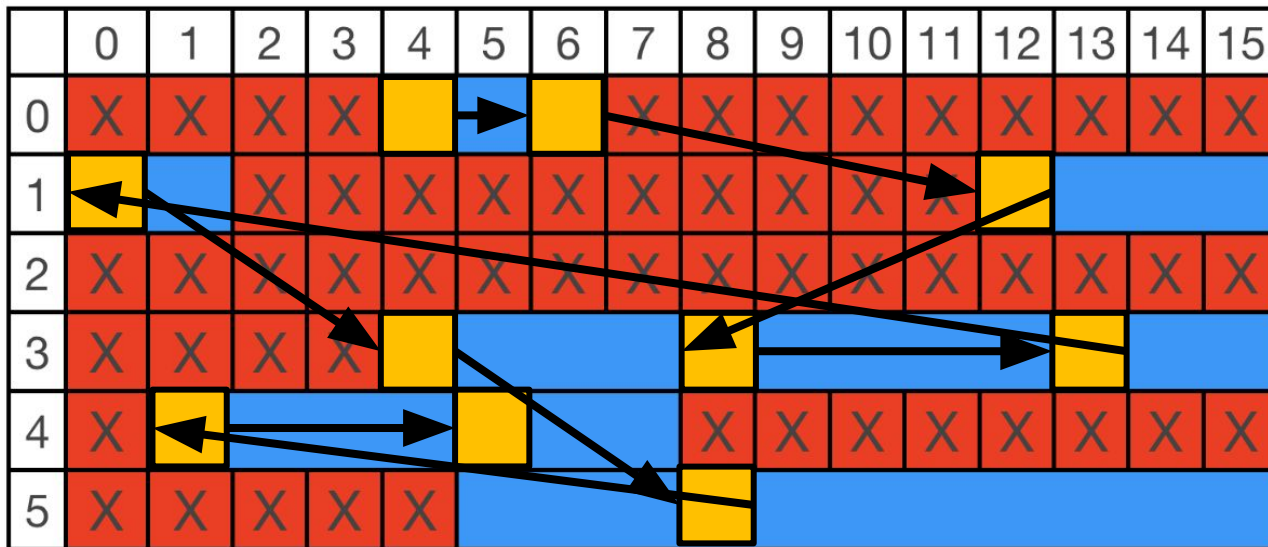
Cleaning Up: delete

- This destroys the array pointed to by the given pointer, not the pointer itself
 - You can think of this operation as relinquishing control over the memory back to the computer
- Once you've deleted the memory pointed at by a pointer, you have a dangling pointer and shouldn't read or write from it

Linked Lists

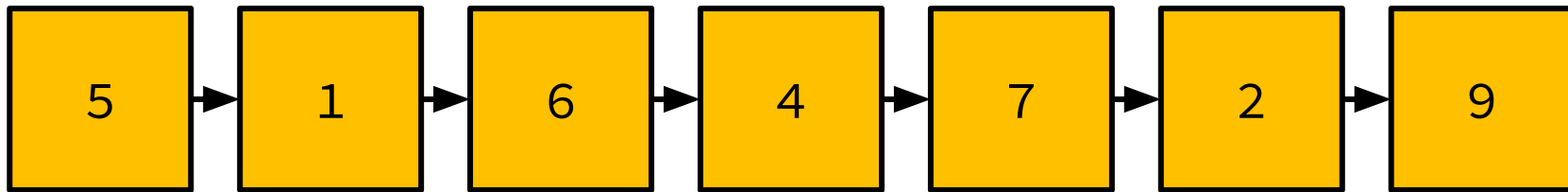
Linked Lists

- Unlike arrays, linked lists allow us to store our data in non-contiguous memory on the heap



Benefits of Linked Lists

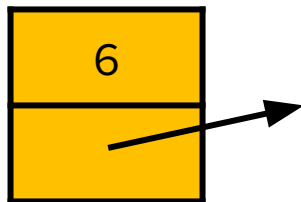
- Easily resizable
- Efficient to insert elements at the beginning



Okay, but what are these little boxes?

Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node is a struct that contains:
 - A piece of data (like an int, or string)
 - A pointer to the next node



```
struct Node {  
    int data;  
    Node* next;  
};
```

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;  
list->data = 6;  
list->next = nullptr;
```

Dereference AND access the field for struct pointers using ->

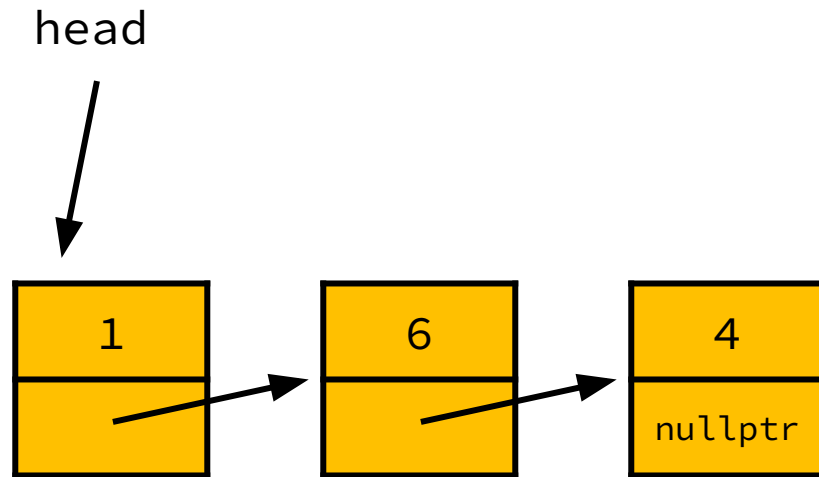
list: 0xfca20b00



Lives at 0xfca20b00 on the heap

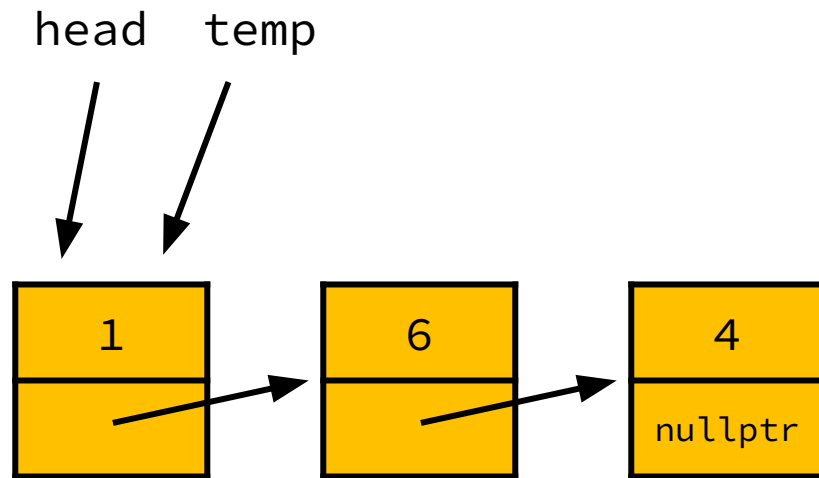
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



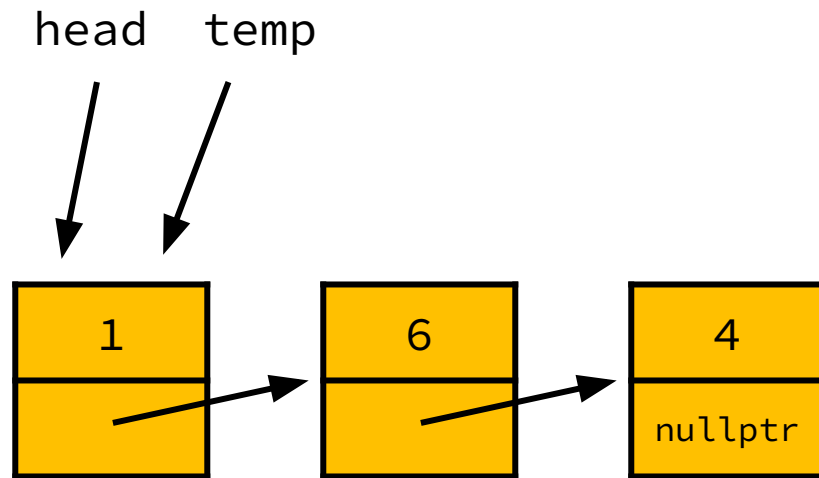
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



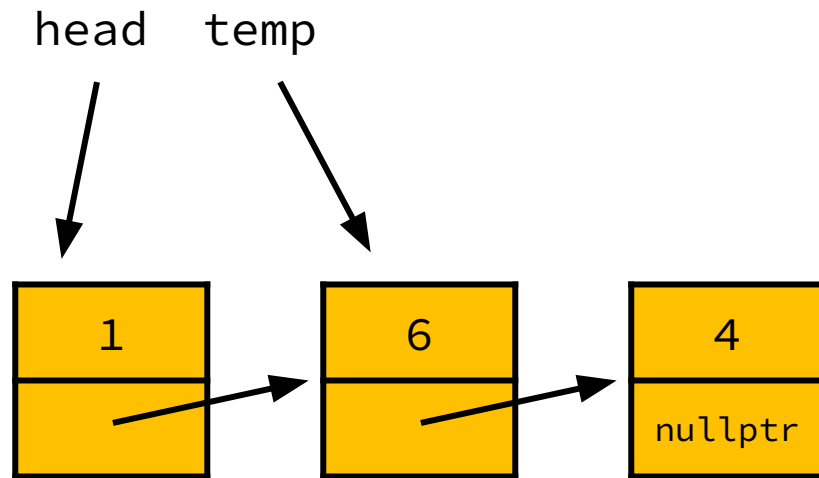
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



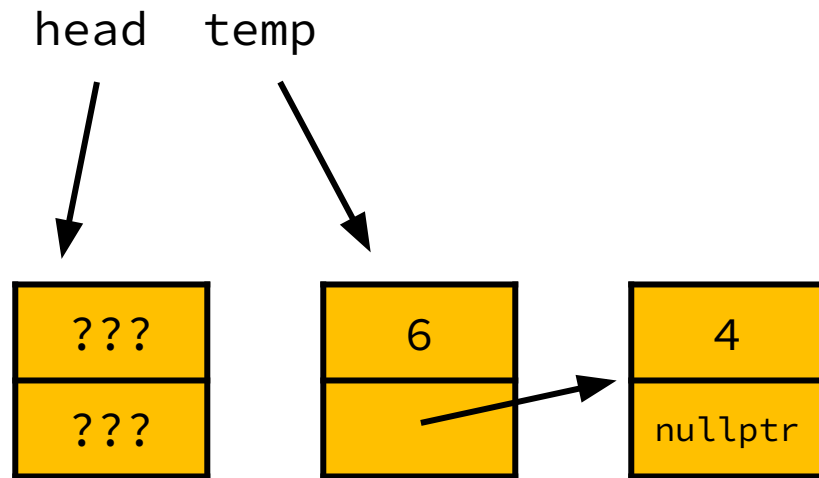
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



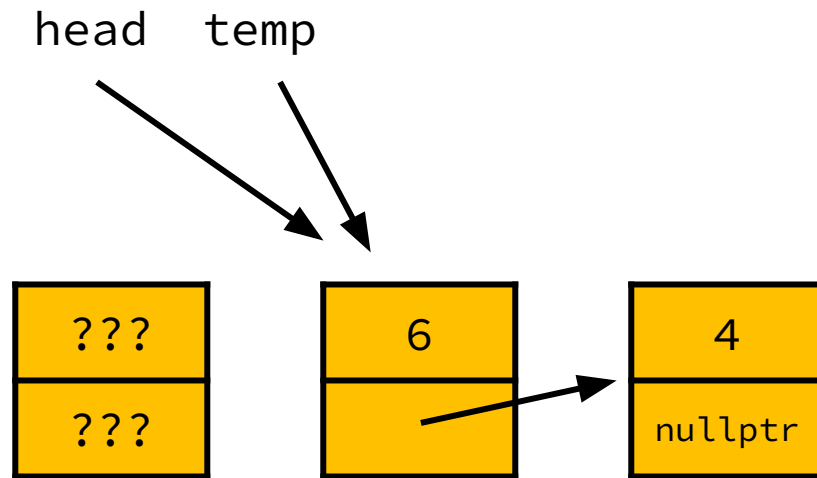
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



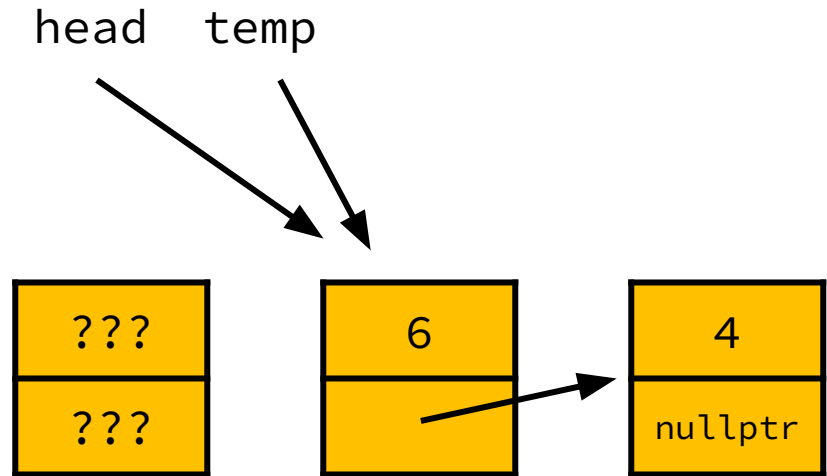
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



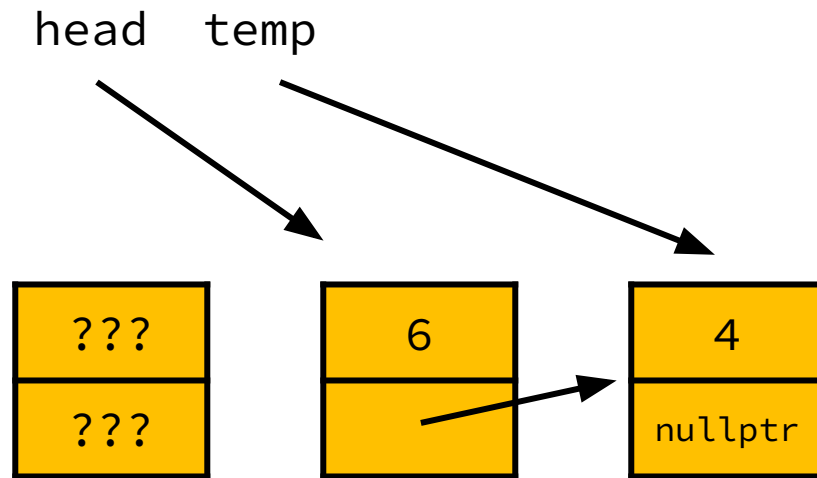
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



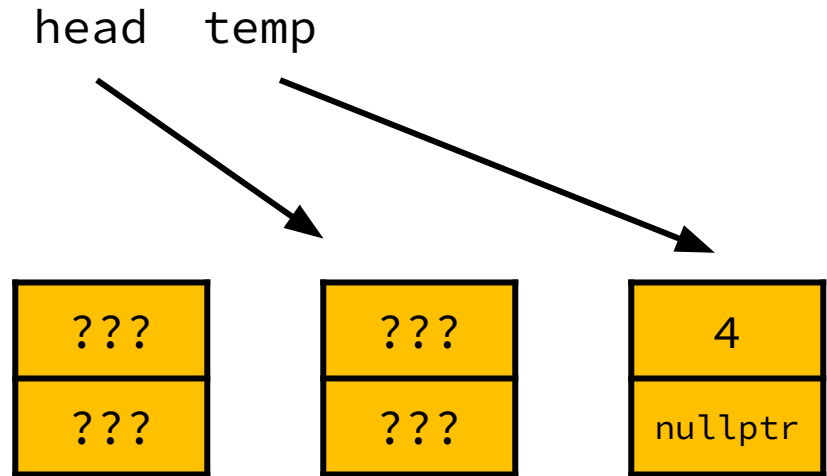
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



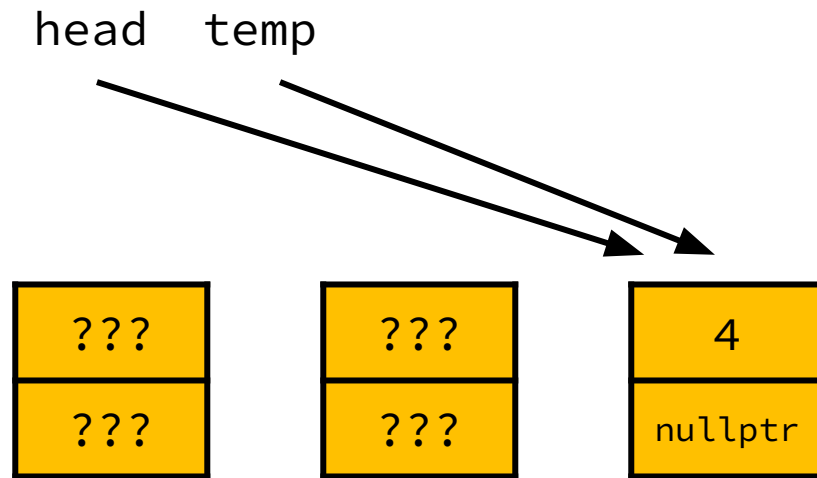
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



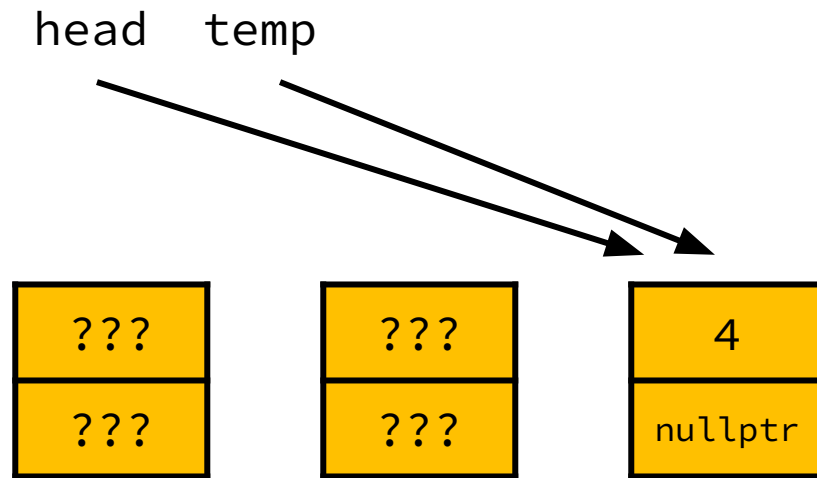
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



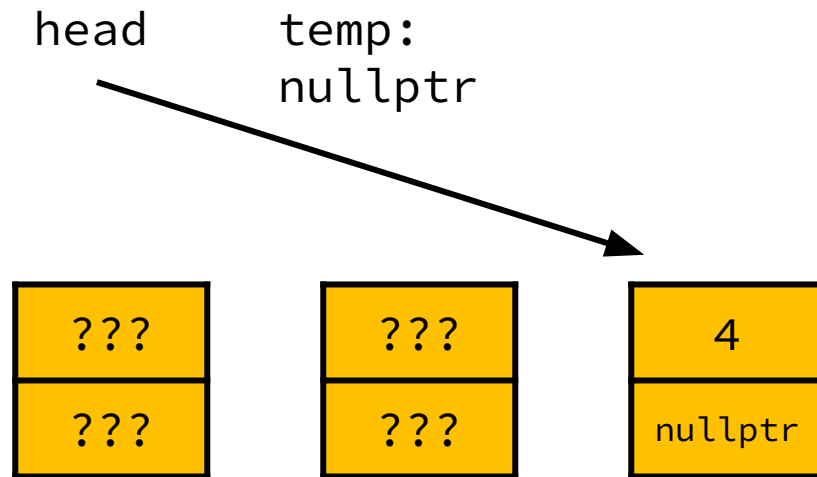
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



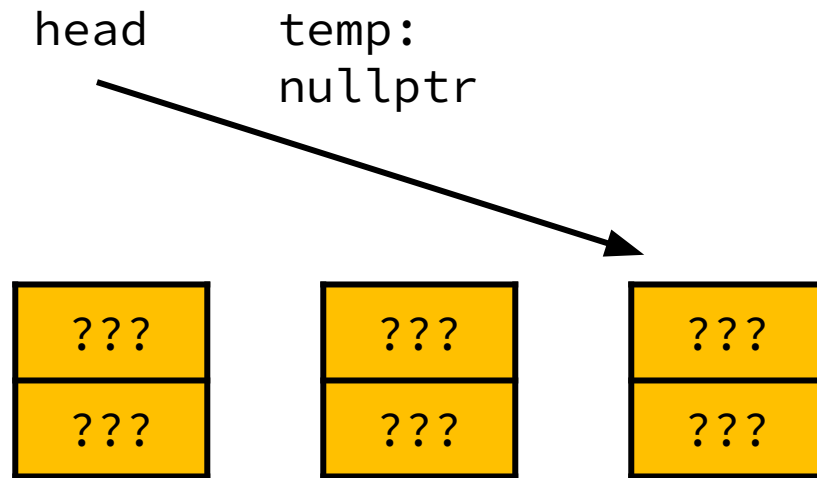
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



Code Trace: Free Linked List

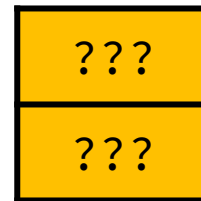
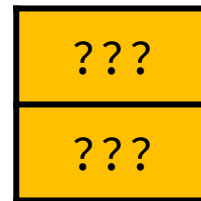
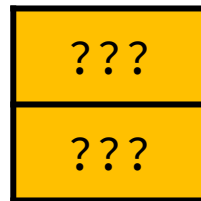
```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```



Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```

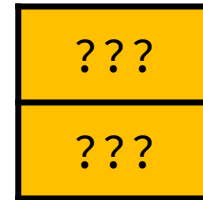
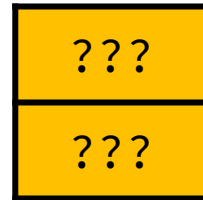
head: temp:
nullptr nullptr



Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```

head: temp:
nullptr nullptr



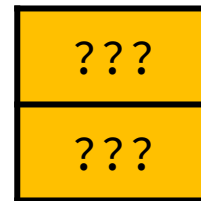
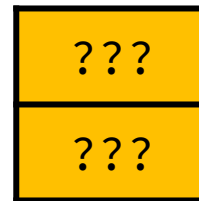
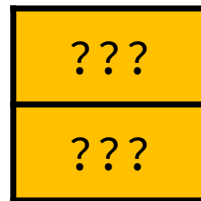
Code Trace: Free Linked List

```
void freeList(Node* head) {  
    Node* temp = head;  
    while (head != nullptr) {  
        temp = temp->next;  
        delete head;  
        head = temp;  
    }  
}
```

HAPPY TIMES



head: temp:
nullptr nullptr



A Few Applications of Traversal

```
void printList(Node* list) {  
    while (list != nullptr) {  
        cout << list->data << endl;  
        list = list->next;  
    }  
}
```

```
int measureList(Node* list) {  
    int count = 0;  
    while (list != nullptr) {  
        count++;  
        list = list->next;  
    }  
    return count;  
}
```

```
void freeList(Node* list) {  
    while (list != nullptr) {  
        Node* temp = list->next;  
        delete list;  
        list = temp;  
    }  
}
```

Pitfalls of Recursive List Traversal

- Recursive solutions to list traversal look elegant, but they generate a recursive call for every element in the list - a linked list with n elements would require n stack frames
- For most computers, the stack frame limit is somewhere in the range of 16-64K - we can't traverse lists with more than 64K elements recursively!

Linked Lists vs. Arrays

Linked Lists

- Chain of nodes, not contiguous in heap memory
- Access nodes starting at head, following the -> next pointer
- Good for implementing other data structures
- Has no member functions like `.size()` or `.add()`

Arrays

- Contiguous chunk of memory on the heap
- Access elements by index
- Same!
- Same!

Linked Lists vs. Arrays, Big-O

Linked Lists

- Prepend - $O(1)$
- Append - $O(n)$
- Insert - $O(n)$
- Delete - $O(n)$
- Traverse - $O(n)$

Arrays

- Prepend - $O(n)$
- Append - $O(1)$
- Insert - $O(n)$
- Delete - $O(n)$
- Traverse - $O(n)$

Passing Pointers by Value

- Unless specified otherwise, parameters in C++ are passed by value
 - this includes pointers!
- When passed by value, callee function gets a copy of the pointer;
it cannot change where the original pointer points

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points
- When you want a helper function to modify the address a pointer points to, you should pass it by reference

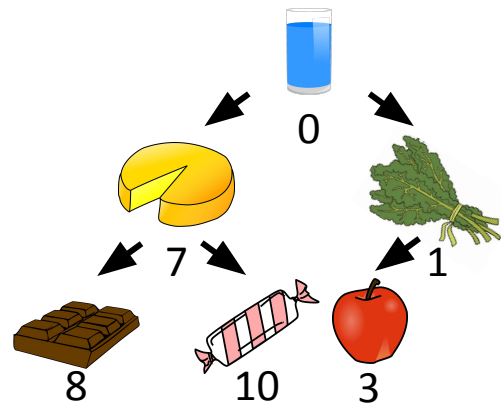
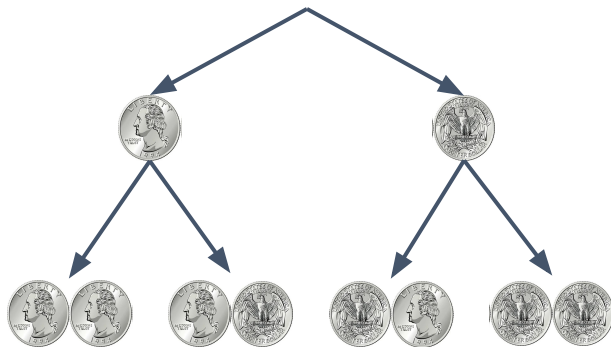
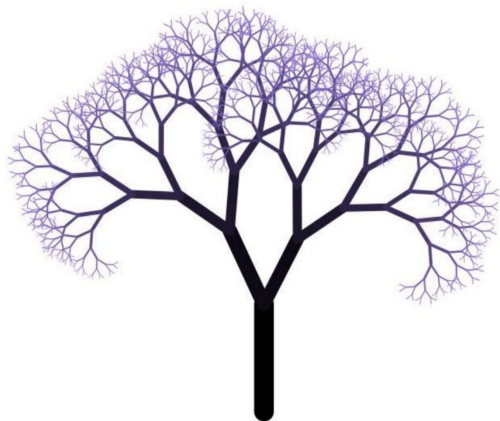
Practice Problem: Reverse List (Solution)

```
void reverse(ListNode*& head) {  
    ListNode* prev = nullptr;  
    ListNode* cur = head;  
    ListNode* next = nullptr;  
  
    while (cur != nullptr) {  
        next = cur->next;  
        cur->next = prev;  
        prev = cur;  
        cur = next;  
    }  
    head = prev;  
}
```

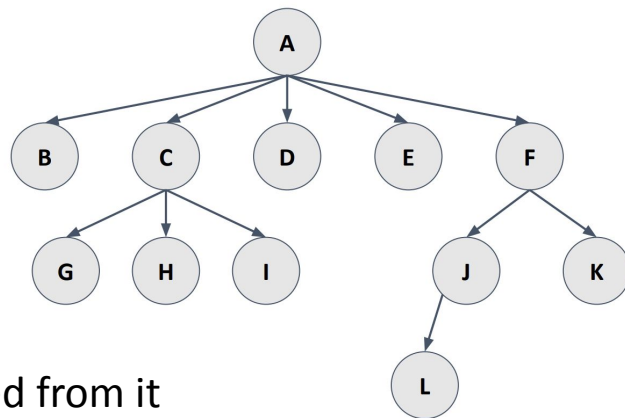
Trees

Throwback

- We've seen trees a ton in this class!



Tree Terminology



Types of nodes

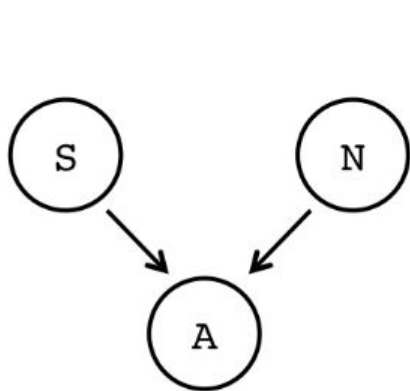
- The **root** node defines the "top" of the tree
- Every node has 0 or more **children** nodes descended from it
- Nodes with no children are called **leaf** nodes
- Every node in a tree has exactly one **parent** node (except for the root node)

Terminology for quantifying trees

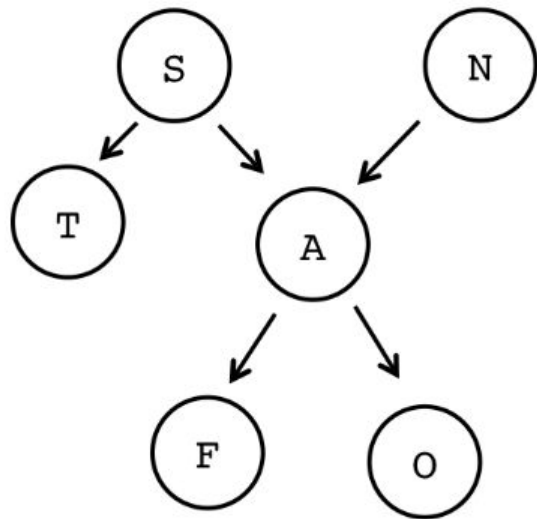
- The **length** of a path between two nodes is the number of edges between them
- The **depth** of a node is the length of the path from the root to that node
- The **height** of a tree is the number of nodes in the longest path through the tree (i.e. the number of levels in the tree)

Tree Properties

- Any node in a tree can only have one parent

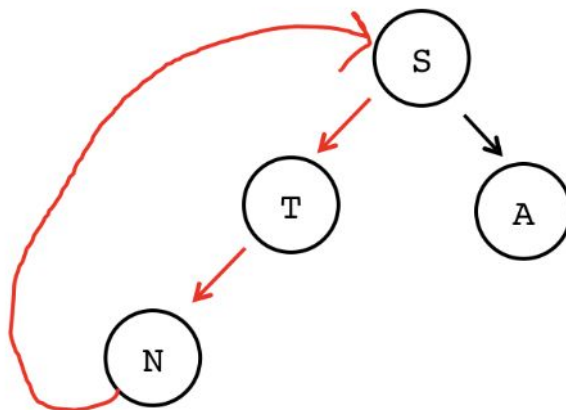


Not trees!



Tree Properties

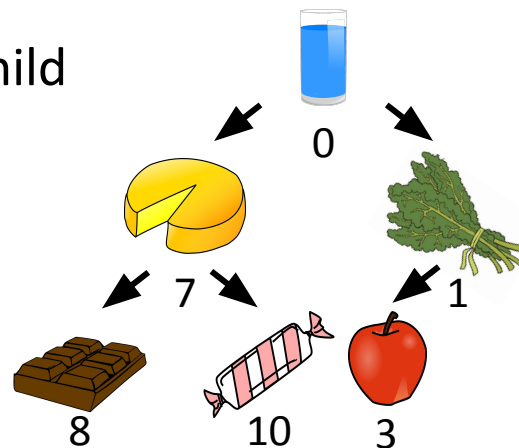
- Any node in a tree can only have one parent
- A tree cannot have cycles or loops



Not a tree!

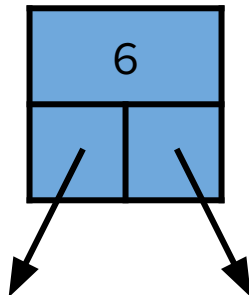
Binary Trees

- Most common trees in CS
 - We've seen these before, Binary Heaps!
- Every node has either 0, 1, or 2 children
- Children are referred to as left child and right child



Building Binary Trees

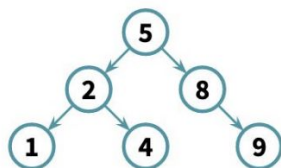
- A binary tree is composed of nodes
- Each node is a struct that contains:
 - A piece of data (like an int, or string)
 - A pointer to the left child
 - A pointer to the right child



```
struct TreeNode {  
    int data;  
    TreeNode* left;  
    TreeNode* right;  
};
```

Tree Traversal Recap

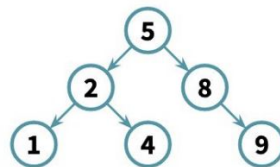
Pre-order



do something (aka cout)
traverse left subtree
traverse right subtree

5 2 1 4 8 9

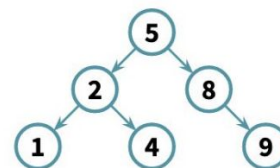
In-order



traverse left subtree
do something (aka cout)
traverse right subtree

1 2 4 5 8 9

Post-order



traverse left subtree
traverse right subtree
do something (aka cout)

1 4 2 9 8 5

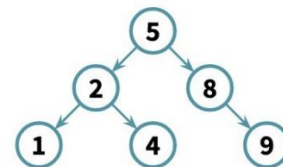
Post-order Traversal - Freeing a Tree

```
void freeTree(TreeNode* node) {  
    if (node == nullptr) {  
        return;  
    }  
    freeTree(node->left);  
    freeTree(node->right);  
    delete node;  
}
```

Post-order Traversal - Freeing a Tree

```
void freeTree(TreeNode* node) {  
    if (node == nullptr) {  
        return;  
    }  
    freeTree(node->left);  
    freeTree(node->right);  
    delete node;  
}
```

Post-order

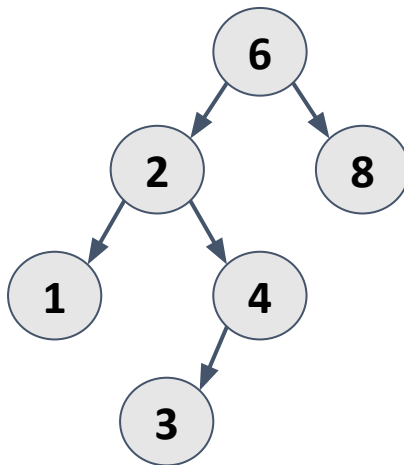


traverse left subtree
traverse right subtree
do something (aka cout)

1 4 2 9 8 5

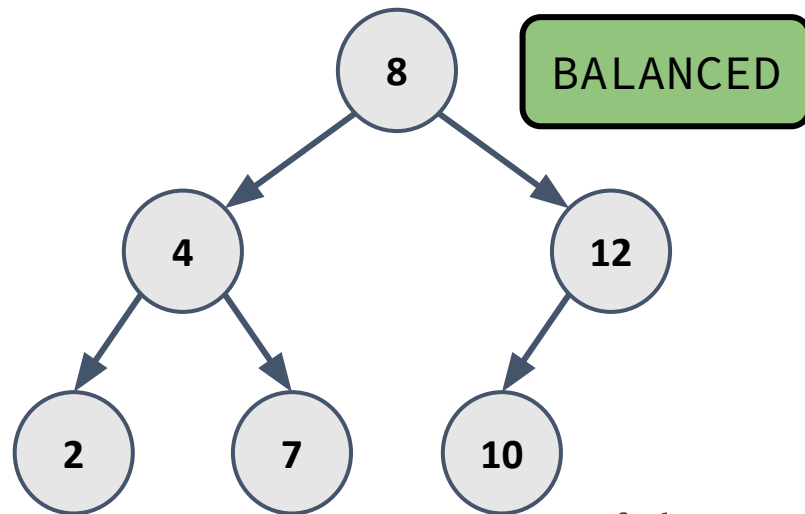
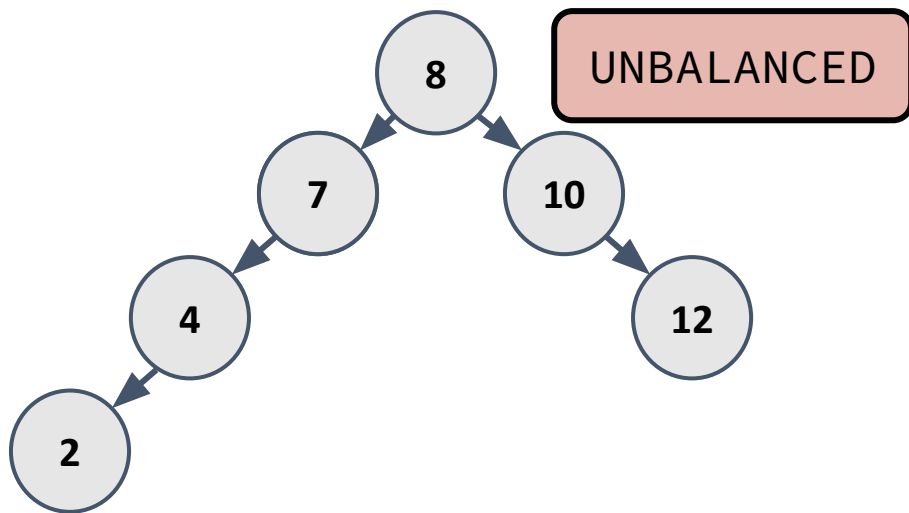
Binary Search Trees (BSTs)

1. Binary tree (each node has 0, 1, or 2 children)
2. For a node with value X:
 - a. All nodes in its left subtree must be less than X
 - b. All nodes in its right subtree must be greater than X



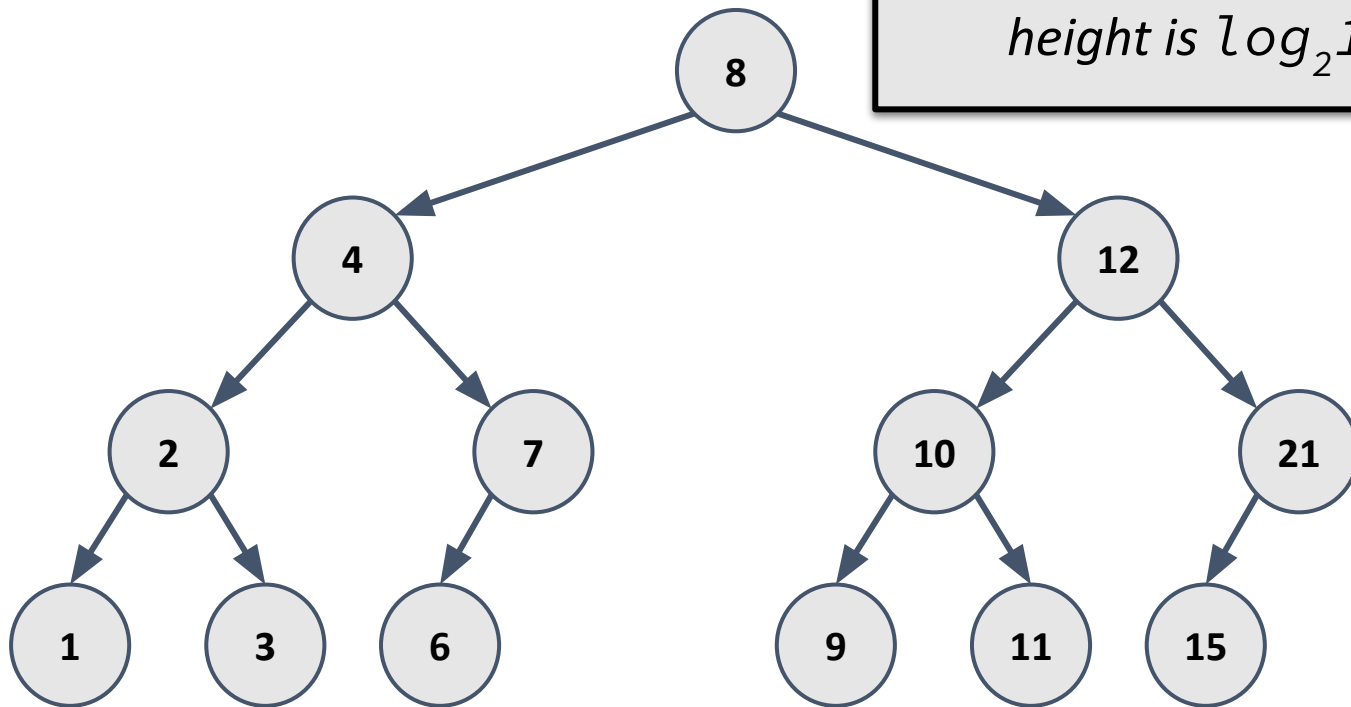
Balanced BSTs

- A BST is **balanced** if its height is $O(\log n)$, where n is the number of nodes in the tree
 - This means left/right subtrees don't differ in height by more than 1



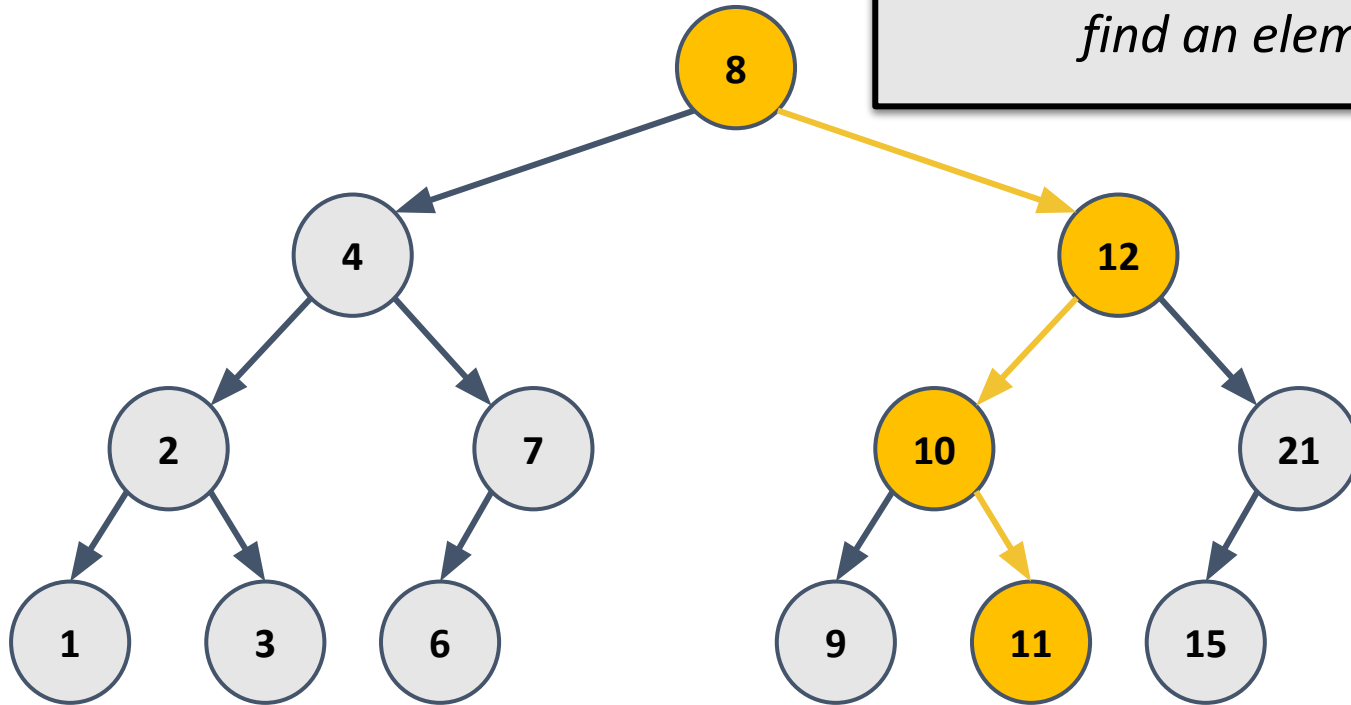
Binary Search Trees (BSTs)

This tree is balanced; we've got 13 nodes in this tree, and its height is $\log_2 13 \approx 4$.



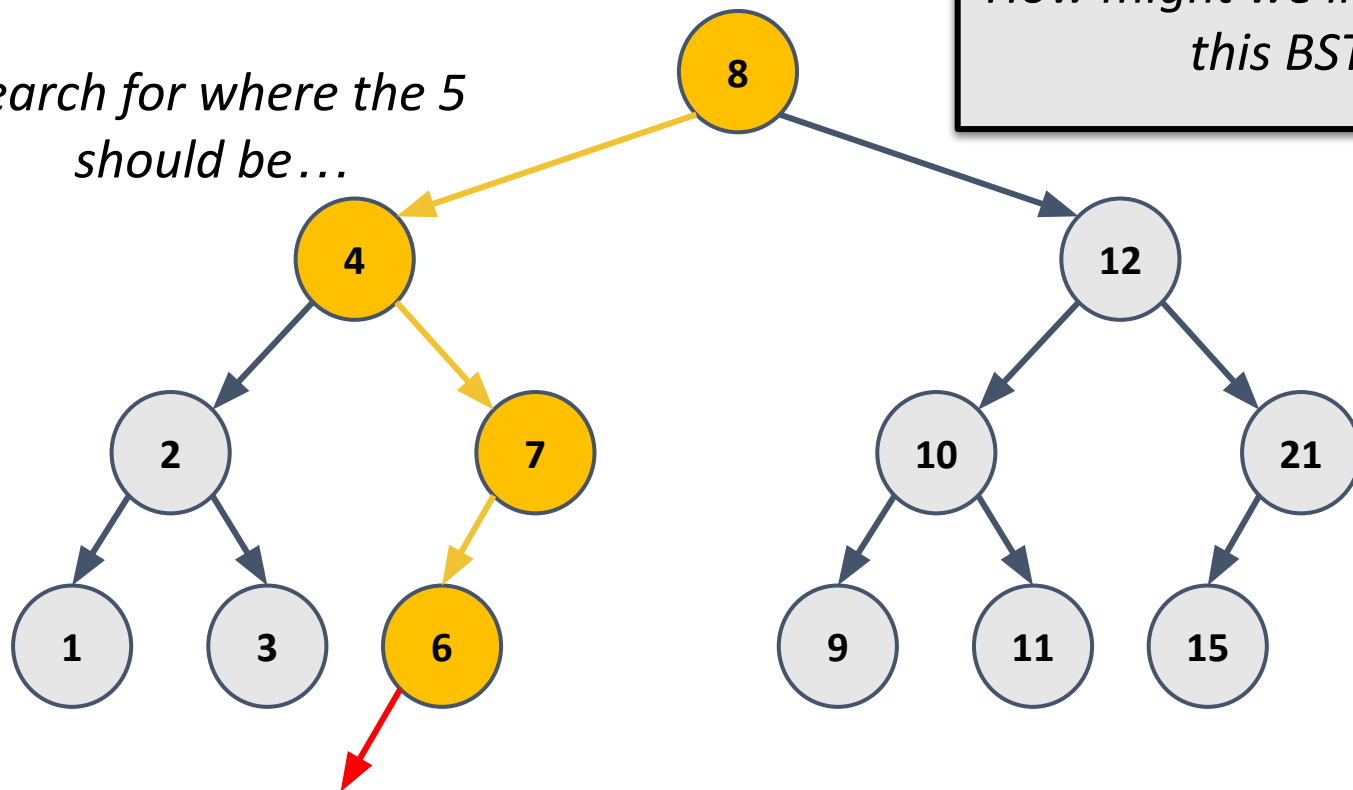
BST Lookups

Worst case, we have to take $O(\log n)$ steps in the tree to find an element.



BST Insertion

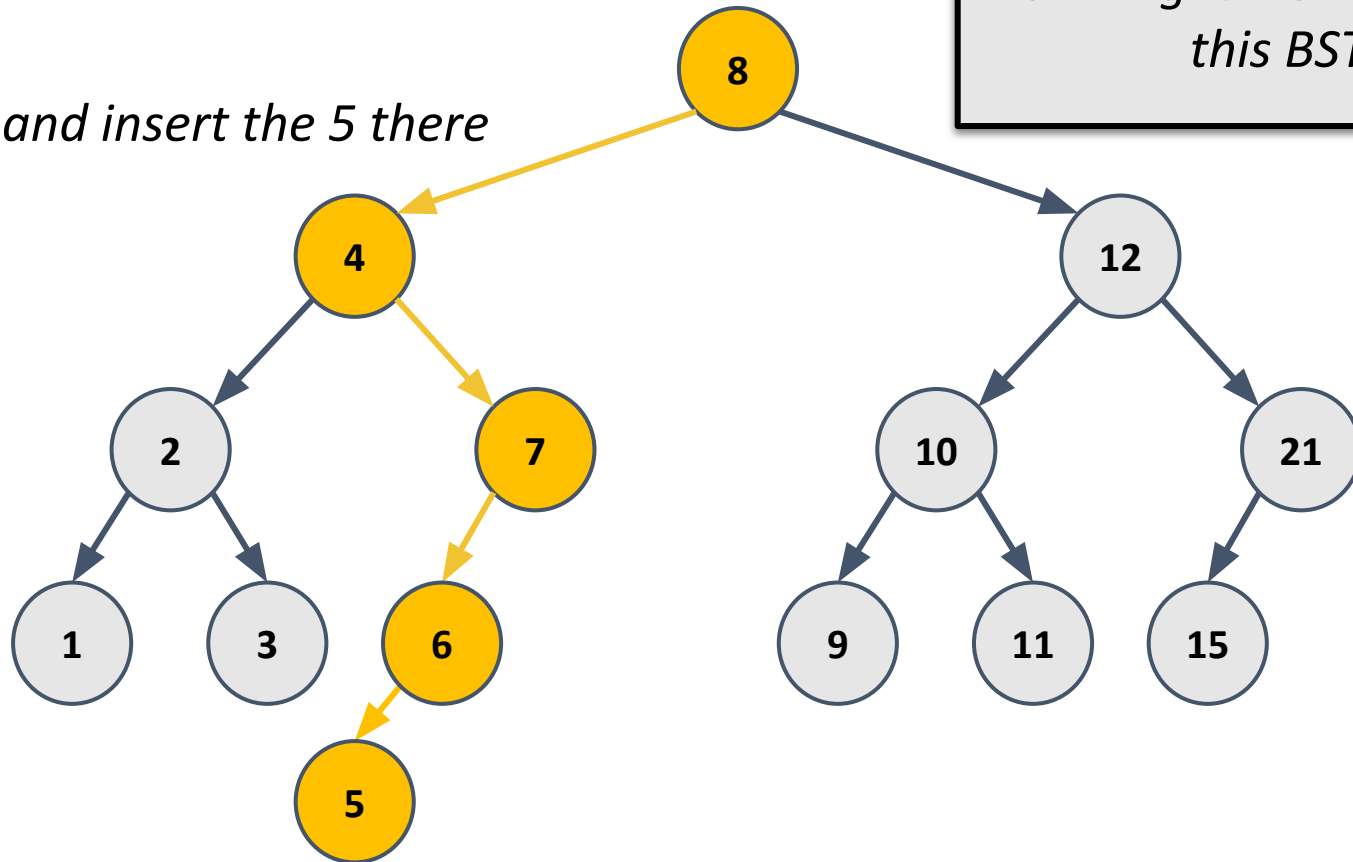
*Search for where the 5
should be...*



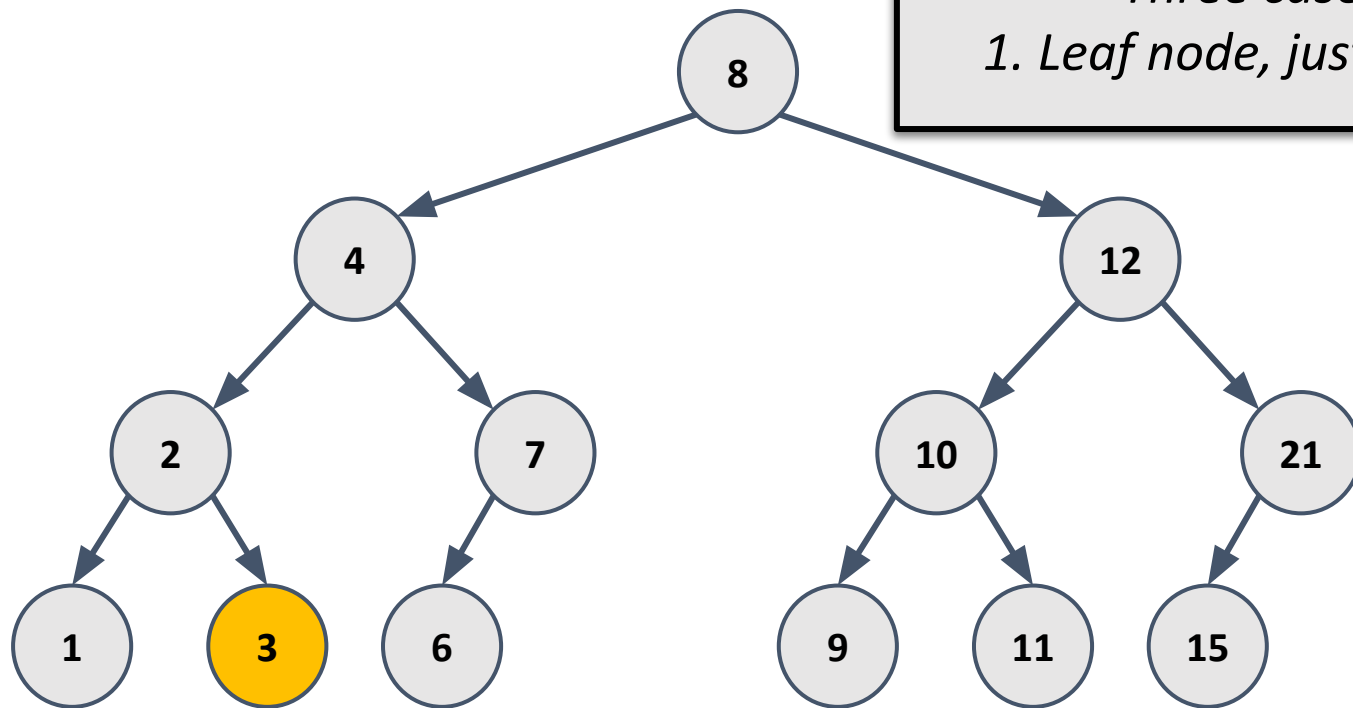
*How might we insert 5 into
this BST?*

BST Insertion

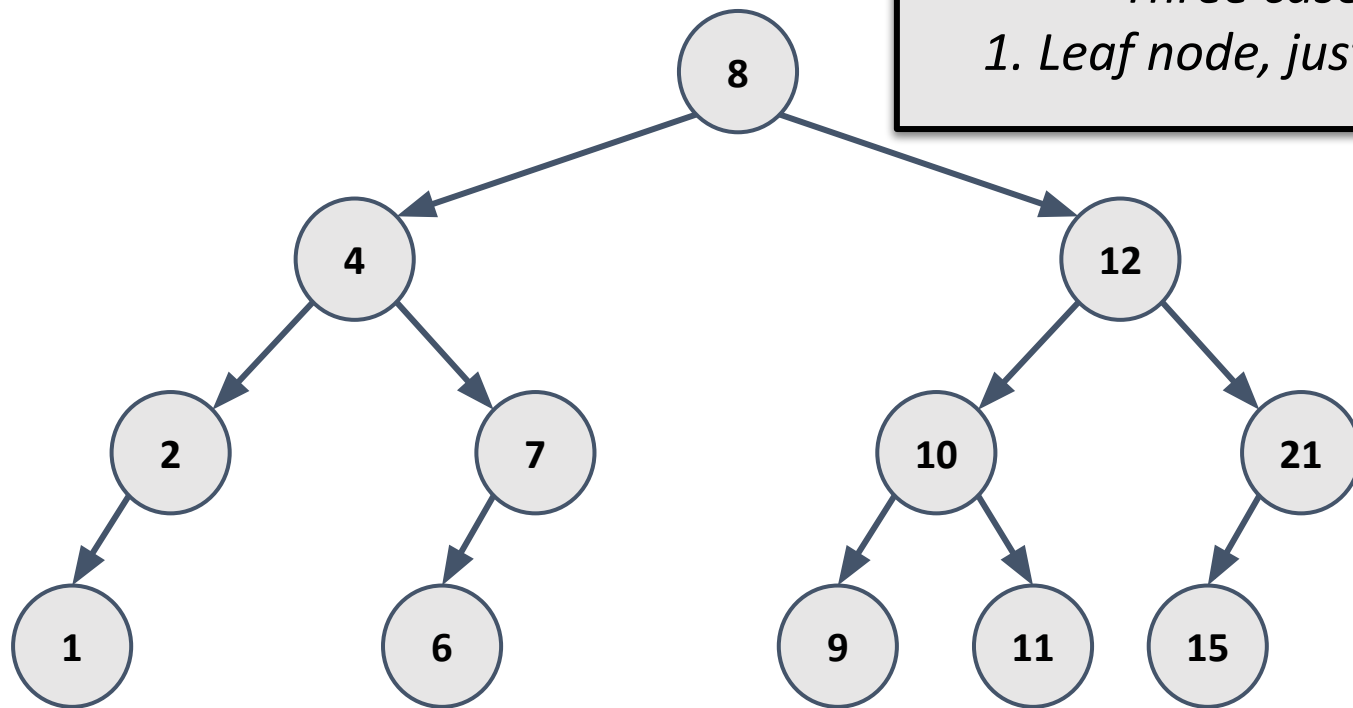
... and insert the 5 there



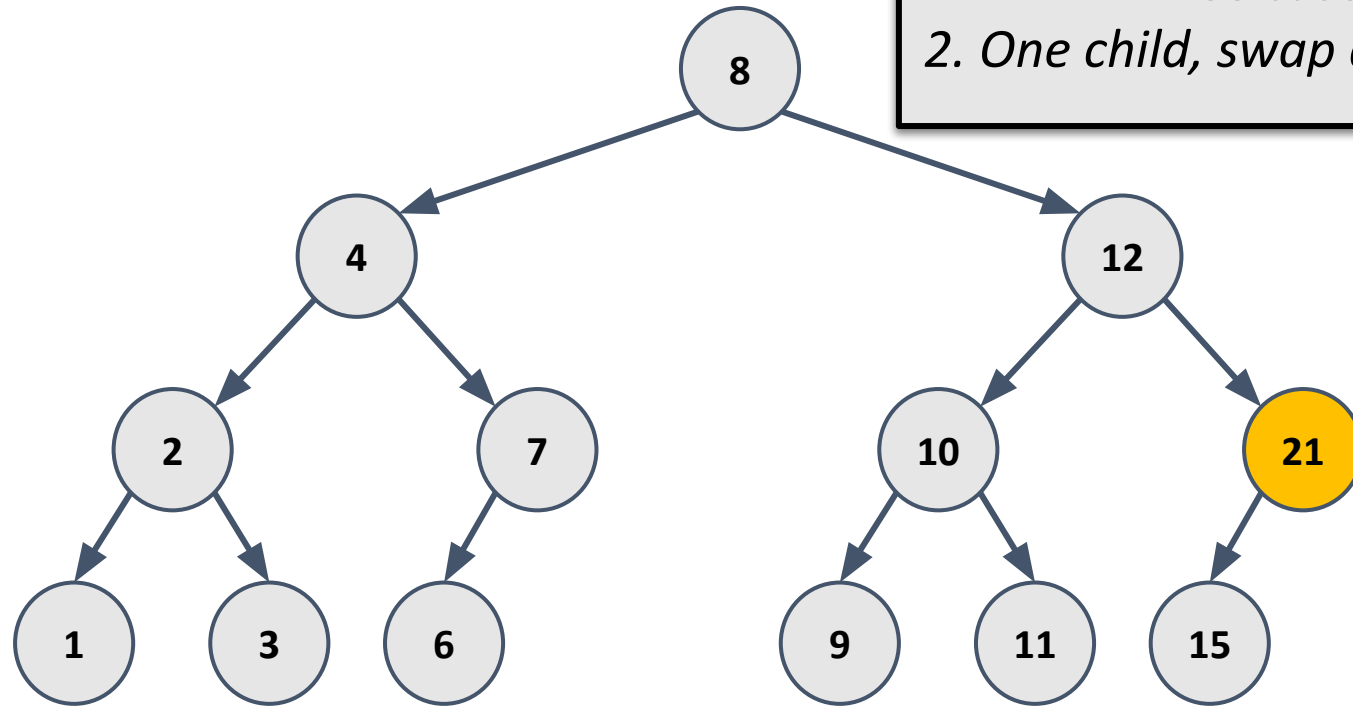
BST Deletion



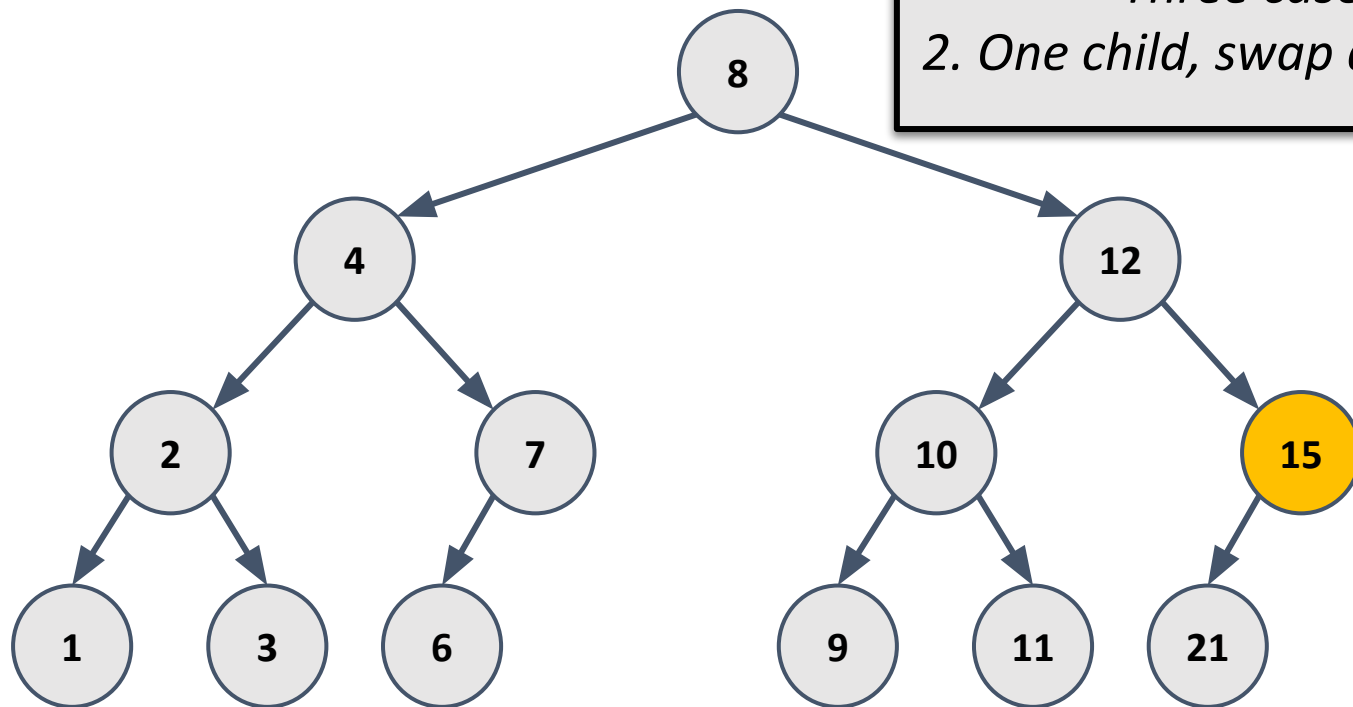
BST Deletion



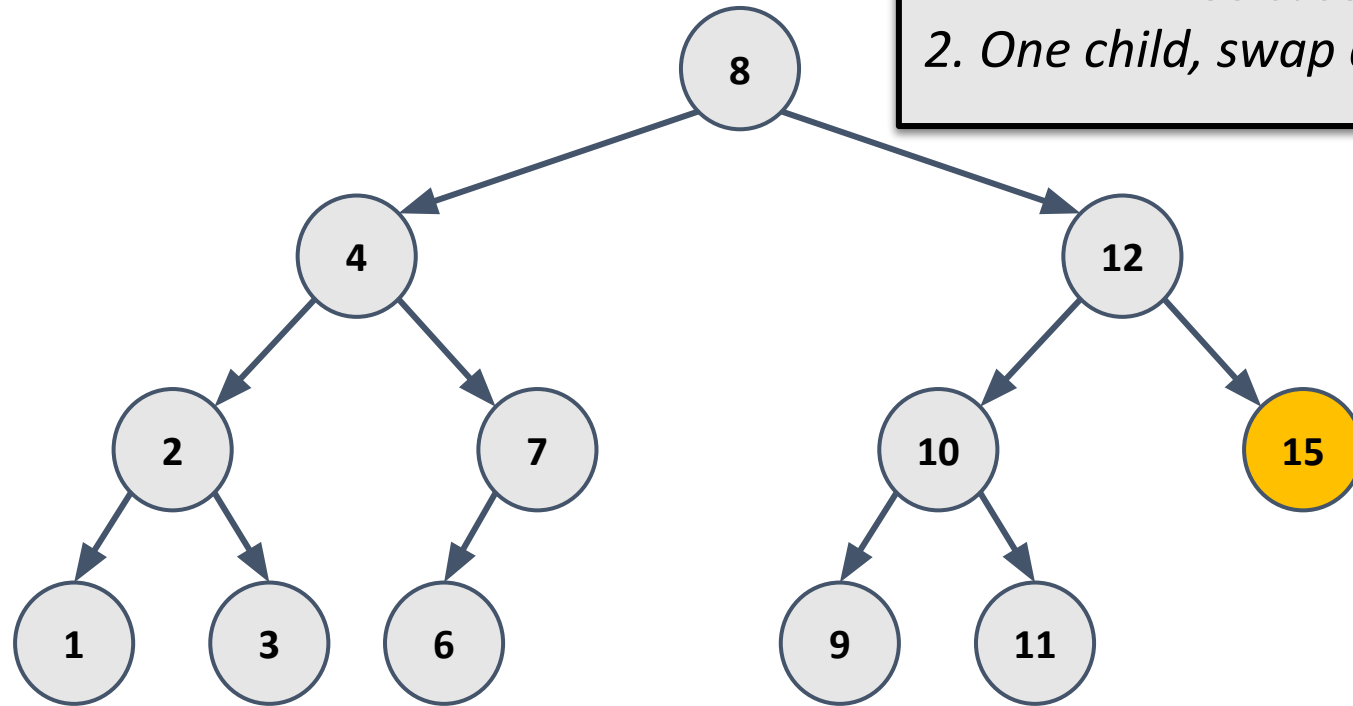
BST Deletion



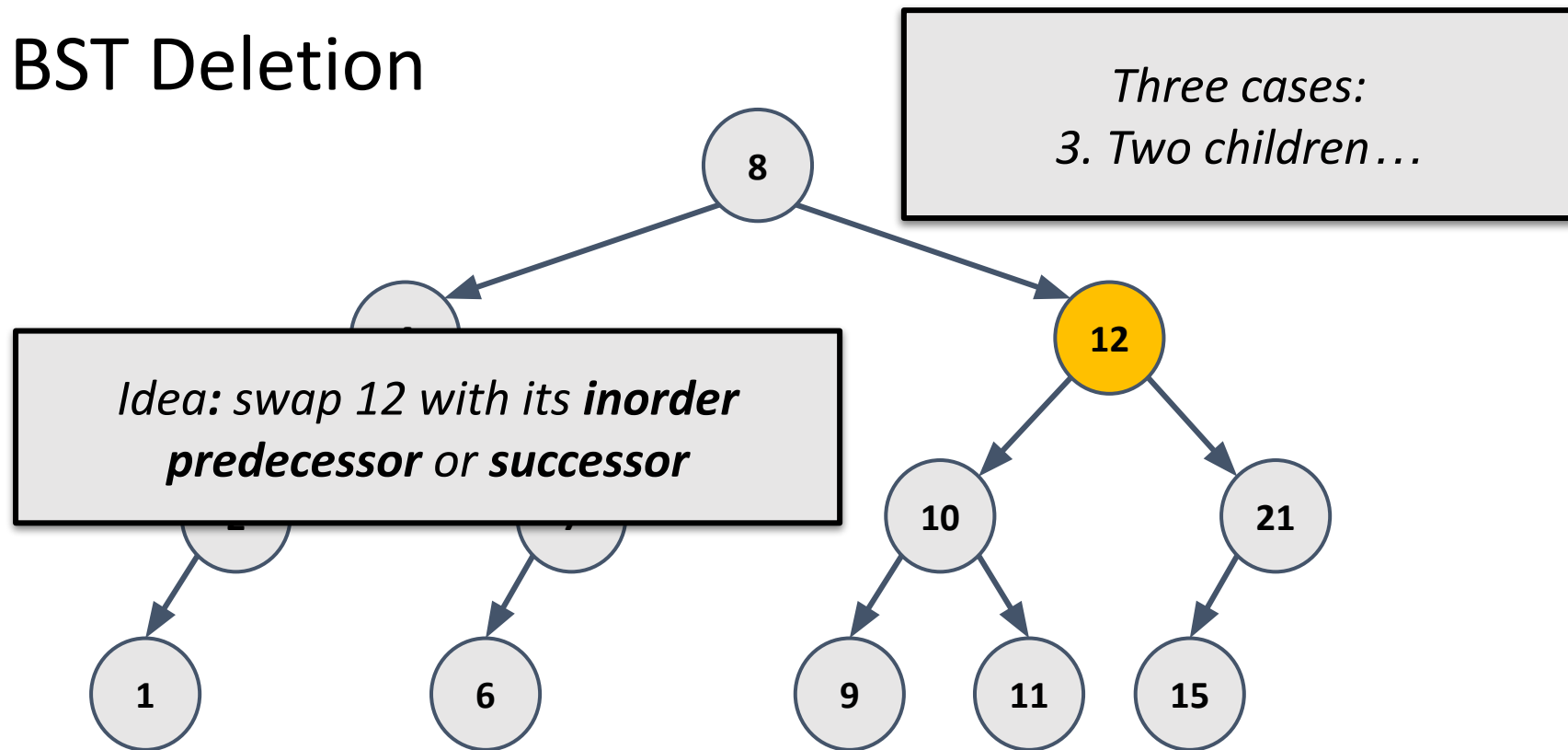
BST Deletion



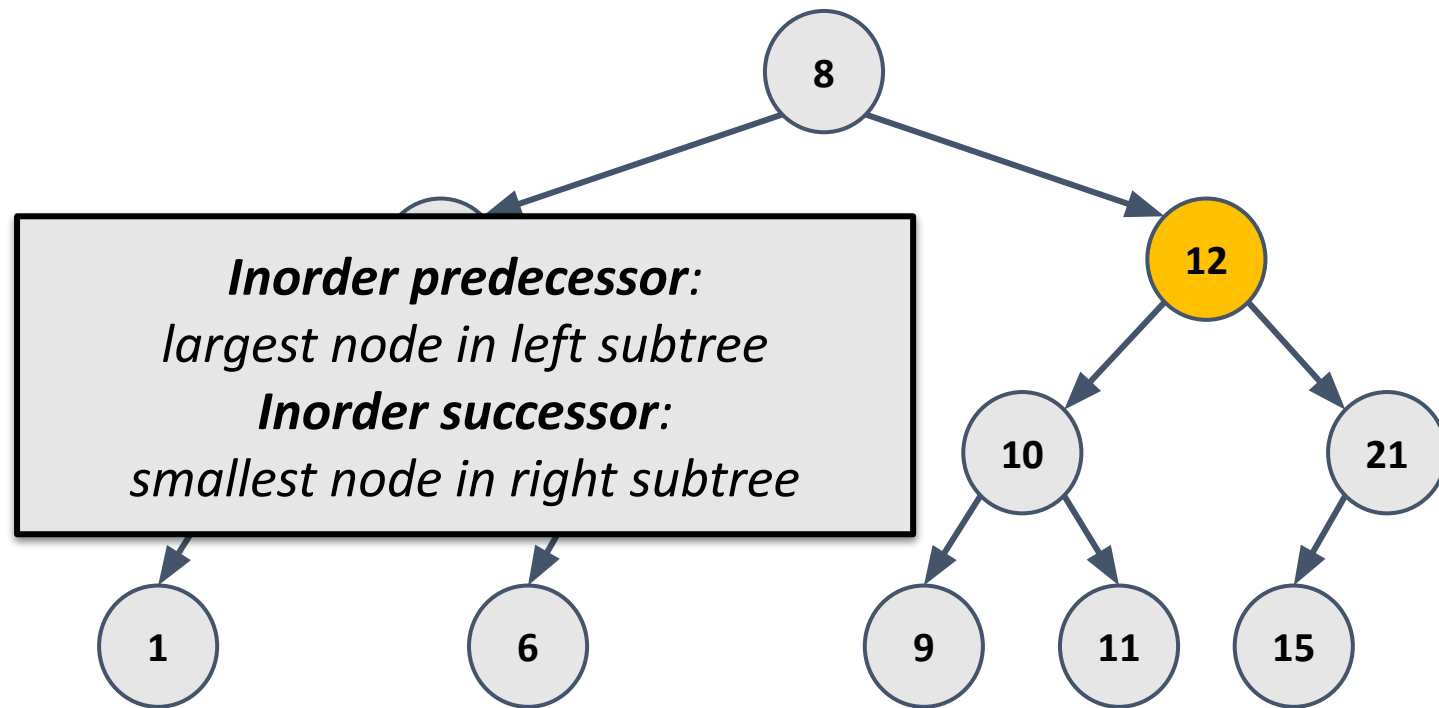
BST Deletion



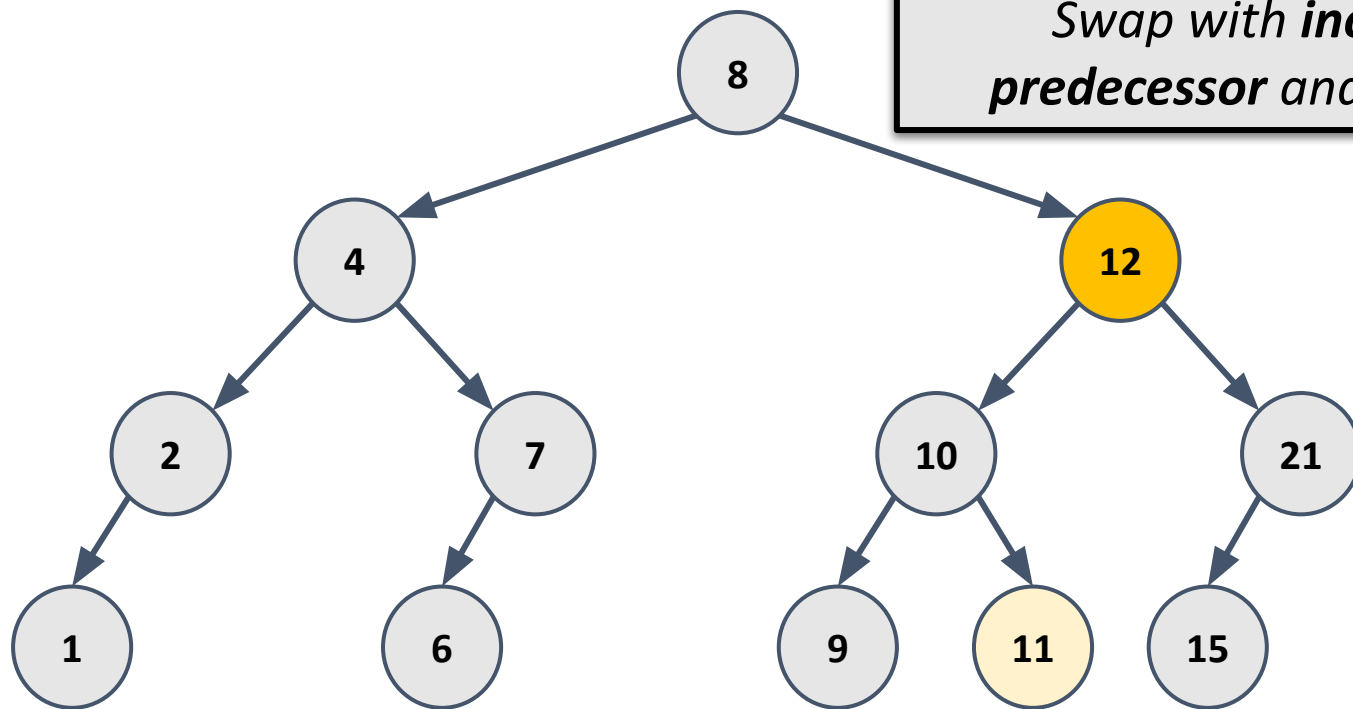
BST Deletion



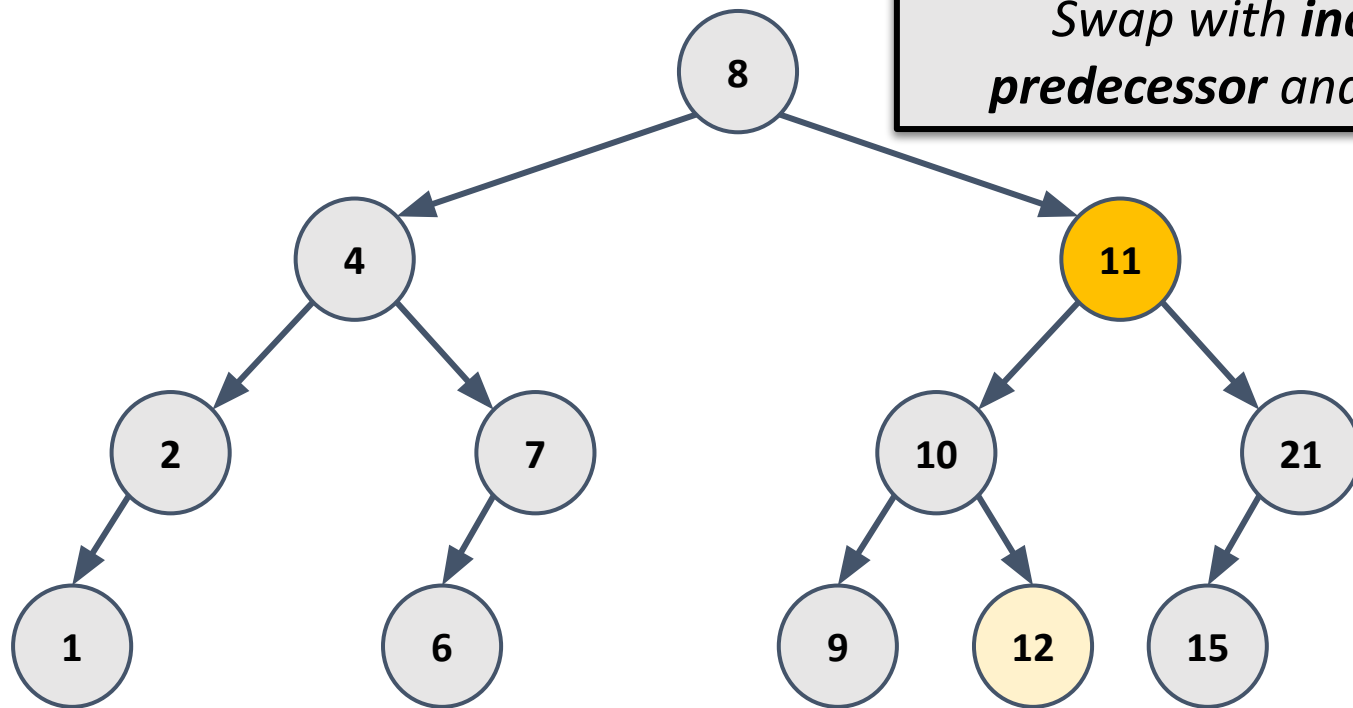
BST Deletion



BST Deletion

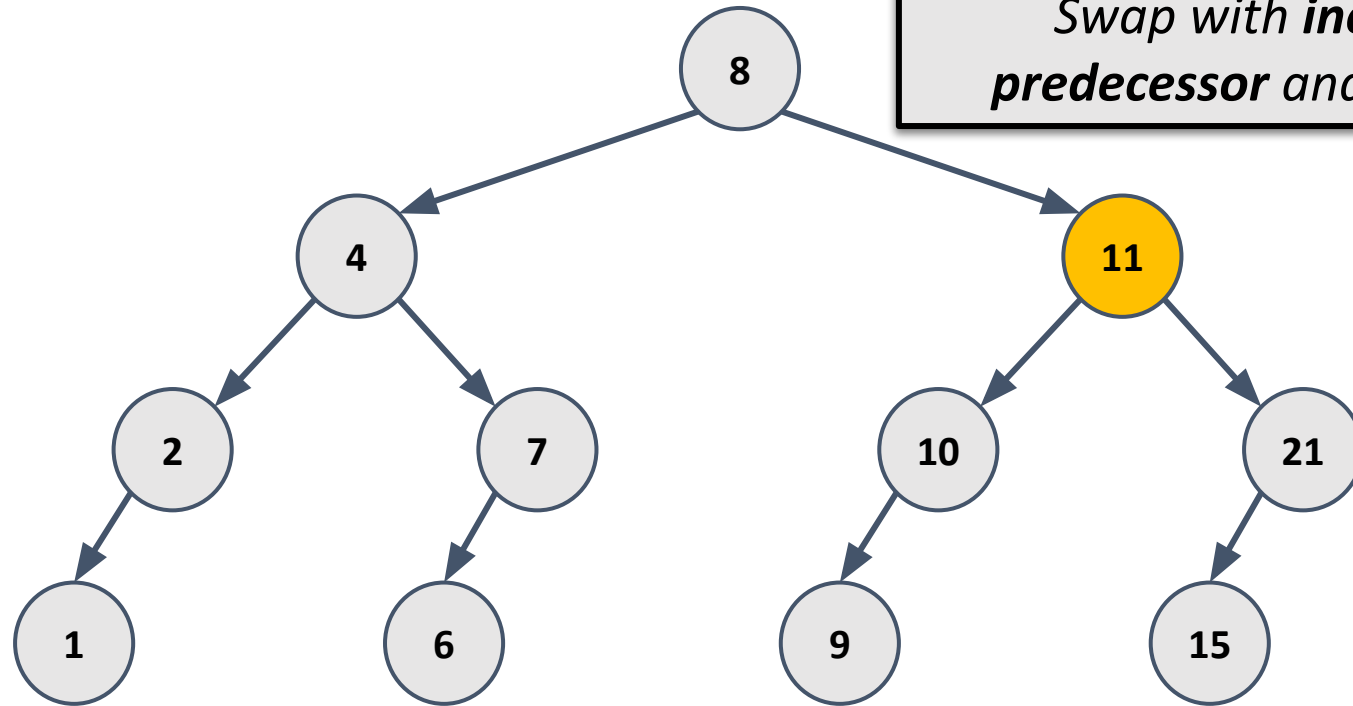


BST Deletion



Three cases:
Swap with **inorder**
predecessor and delete.

BST Deletion



Three cases:
Swap with **inorder**
predecessor and delete.

Big-O of ADT Operations

Vectors

- `.size()` - $O(1)$
- `.add()` - $O(1)$
- `v[i]`
- `.insert()`
- `.remove()`
- `.subarray()`
- traversal

Queues

- `.size()` - $O(1)$
- `.peek()` - $O(1)$

Sets

- `.size()` - $O(1)$
- `.isEmpty()` - $O(1)$
- `.add()` - $O(\log n)$
- `.remove()` - $O(\log n)$
- `.contains()` - $O(\log n)$
- traversal - $O(n)$

Grids

- `.numCells()`
- `.numCols()` - $O(1)$
- `grid[i][j]` - $O(1)$
- `.inBounds()` - $O(1)$
- traversal - $O(n^2)$

- `.peek()` - $O(1)$
- `.push()` - $O(1)$
- `.pop()` - $O(1)$
- `.isEmpty()` - $O(1)$
- traversal - $O(n)$

Maps

- `.size()` - $O(1)$
- `.isEmpty()` - $O(1)$
- `m[key]` - $O(\log n)$
- `.contains()` - $O(\log n)$
- traversal - $O(n)$

Sets and Maps have $O(\log n)$ lookups, insertion, and deletion because they use BSTs behind the scenes to store data!

Practice Problem: Copy Tree (Solution)

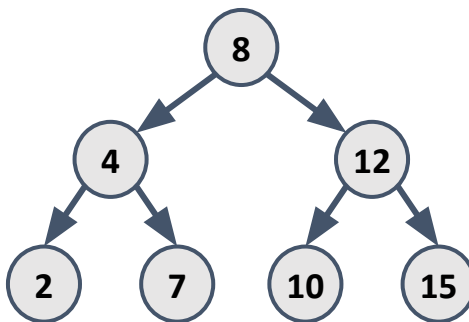
```
TreeNode* copyTree(TreeNode* root) {  
    if (root == nullptr) return nullptr;  
  
    // pre-order traversal, not the only order that would work  
    TreeNode* leftSubtree = copyTree(root->left);  
    TreeNode* rightSubtree = copyTree(root->right);  
  
    TreeNode* currentNode = new TreeNode();  
    currentNode->data = root->data;  
    currentNode->left = leftSubtree;  
    currentNode->right = rightSubtree;  
    return currentNode;  
}
```

Hashing

Binary Search Tree (and Set)

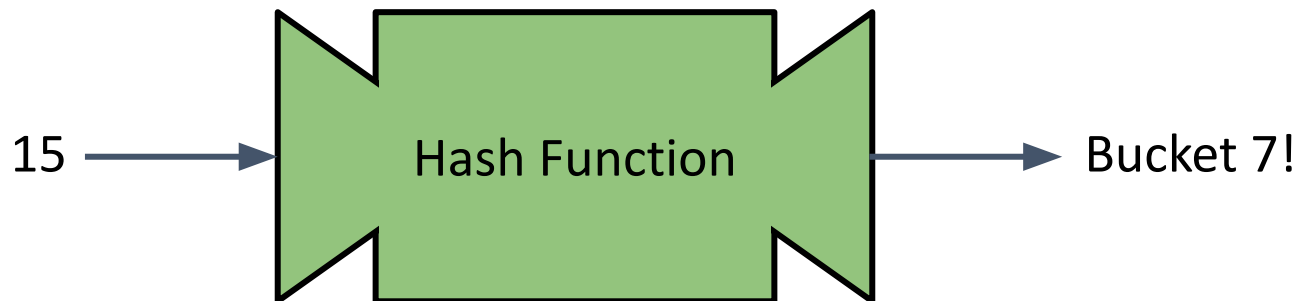
Operation	Runtime
Contains	$O(\log n)$
Insert	$O(\log n)$
Remove	$O(\log n)$

*Motivating question:
CAN WE DO BETTER?*



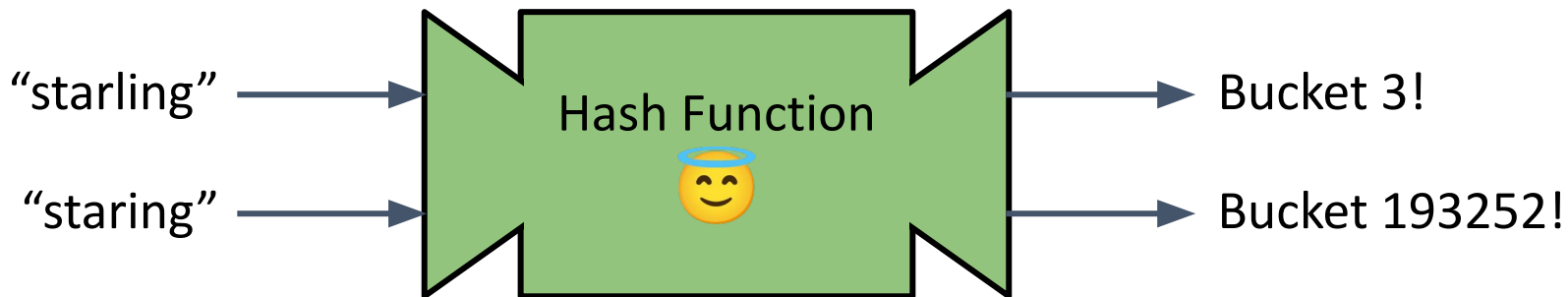
Hash Functions

- A **hash function** is a function that assigns elements to buckets



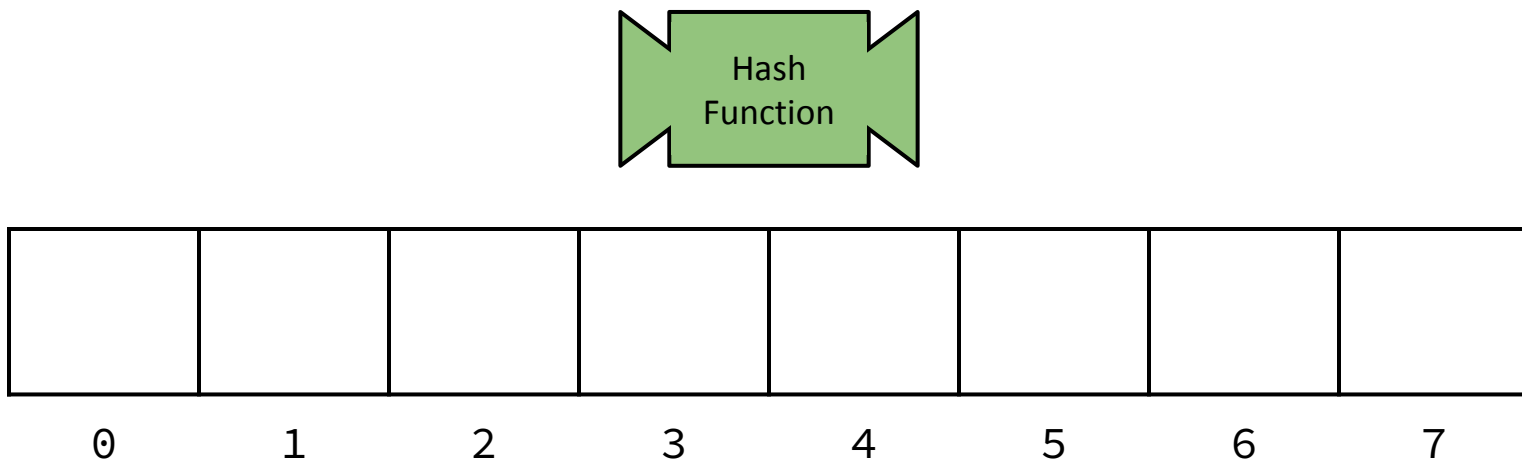
Good Hash Functions

- A **good hash function** distributes elements evenly across buckets
 - This way, no bucket contains too many elements
- Similar inputs will not necessarily have similar hash codes



Chaining Hash Table

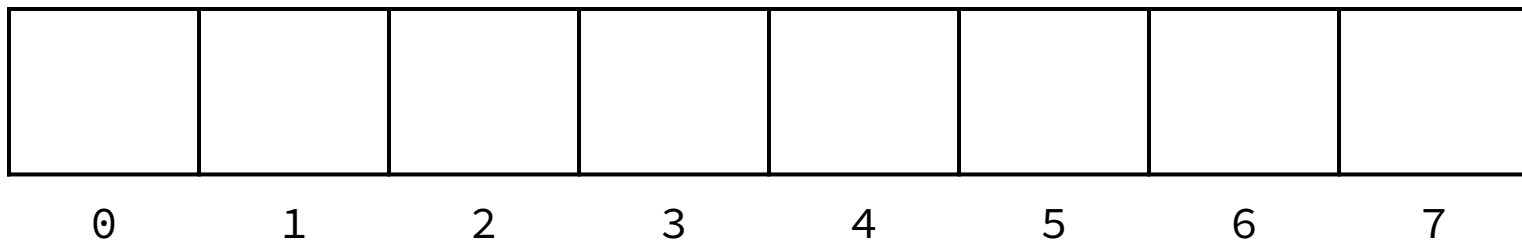
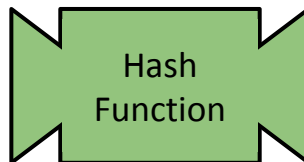
- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

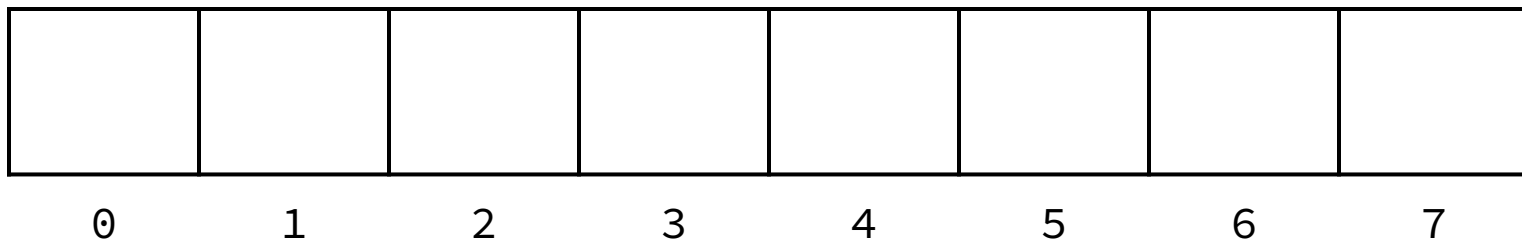
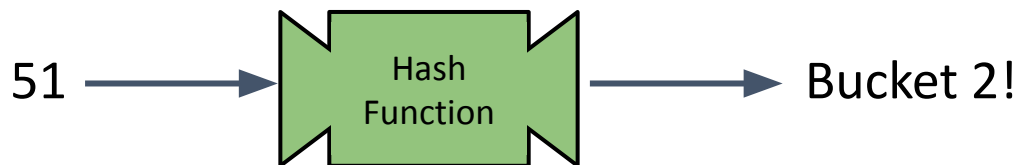
Add 51



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

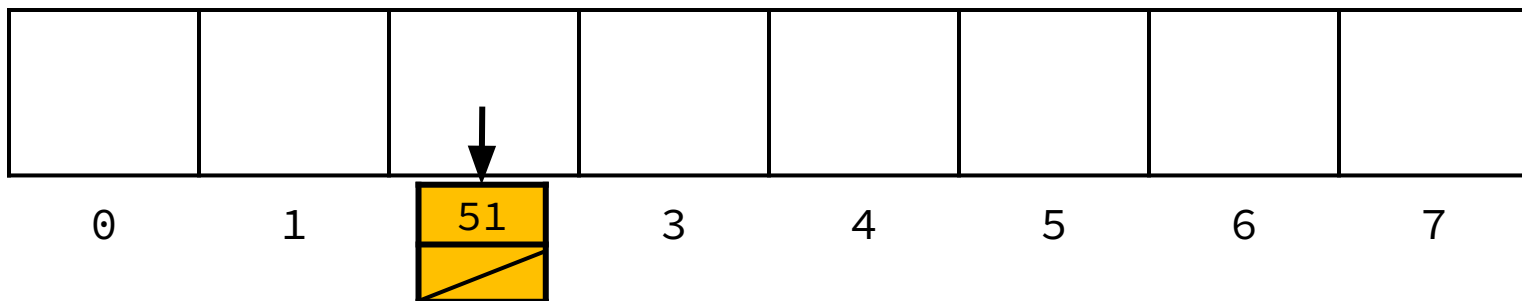
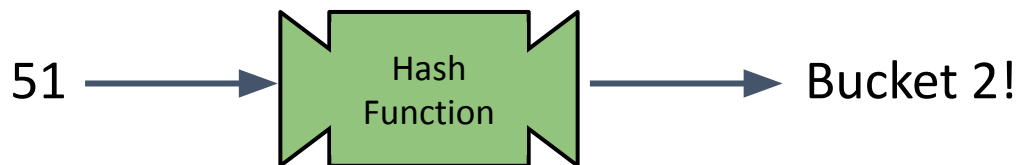
Add 51



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

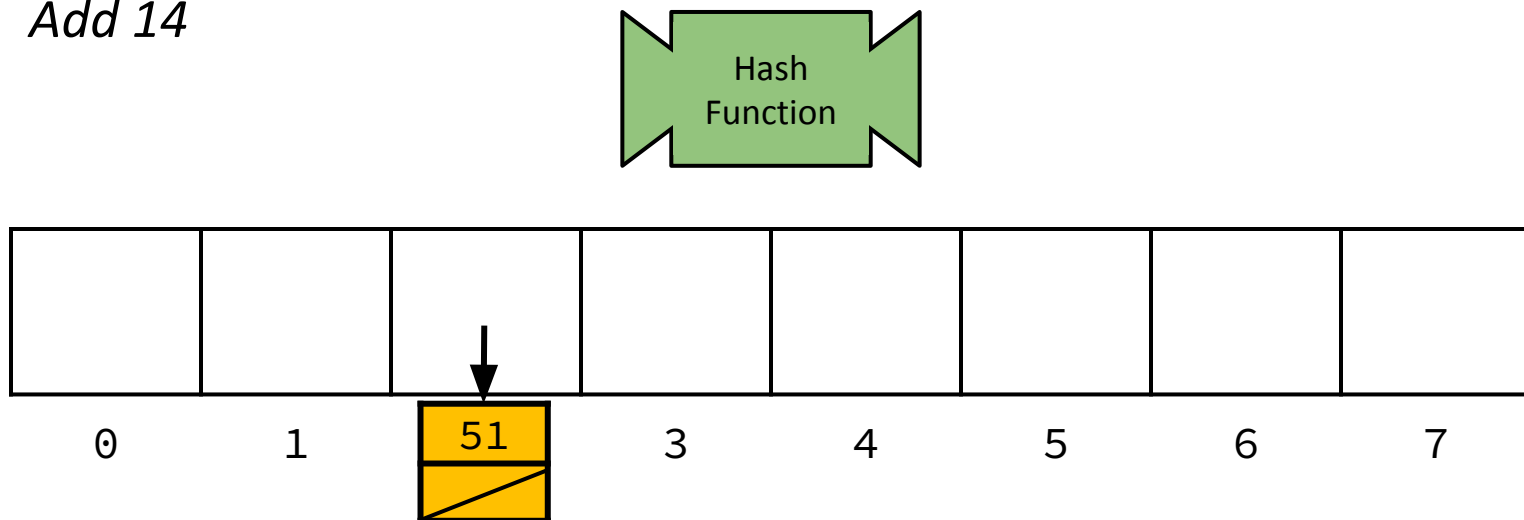
Add 51



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

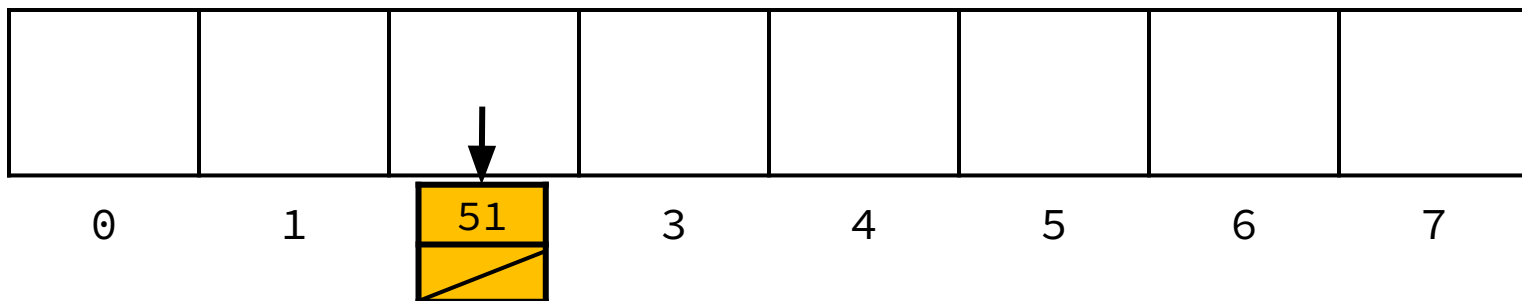
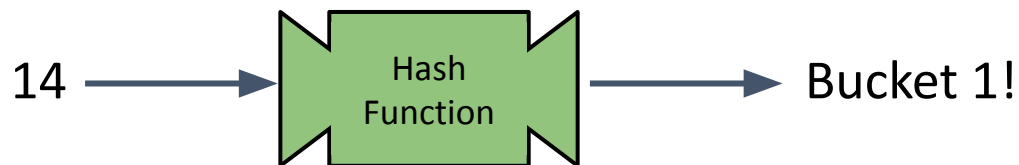
Add 14



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

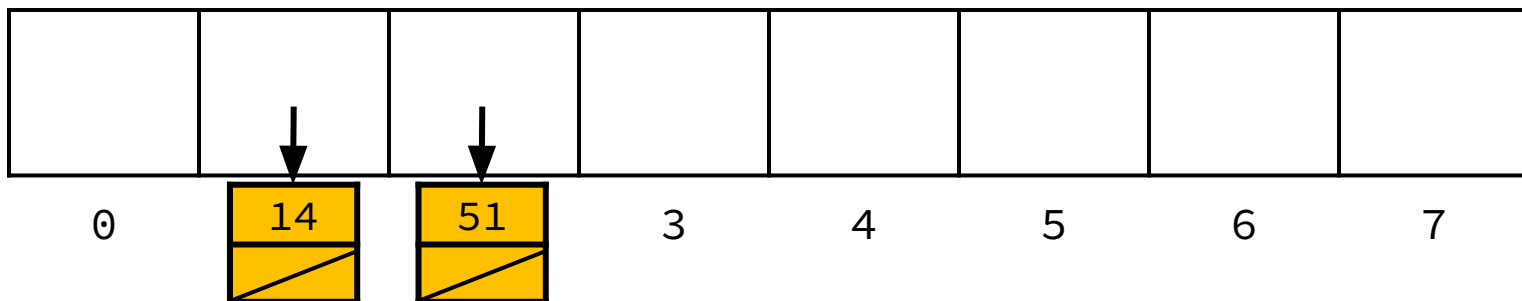
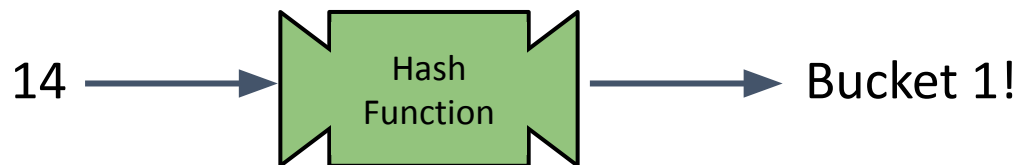
Add 14



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

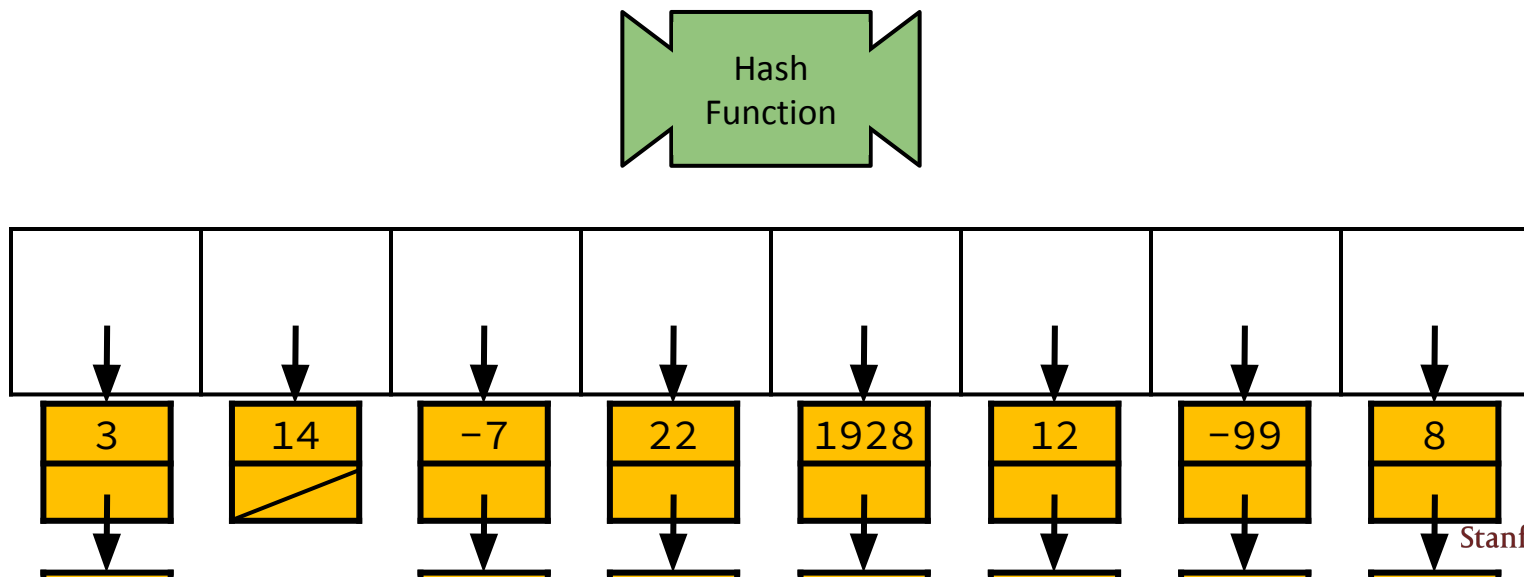
Add 14





Chaining Hash Table

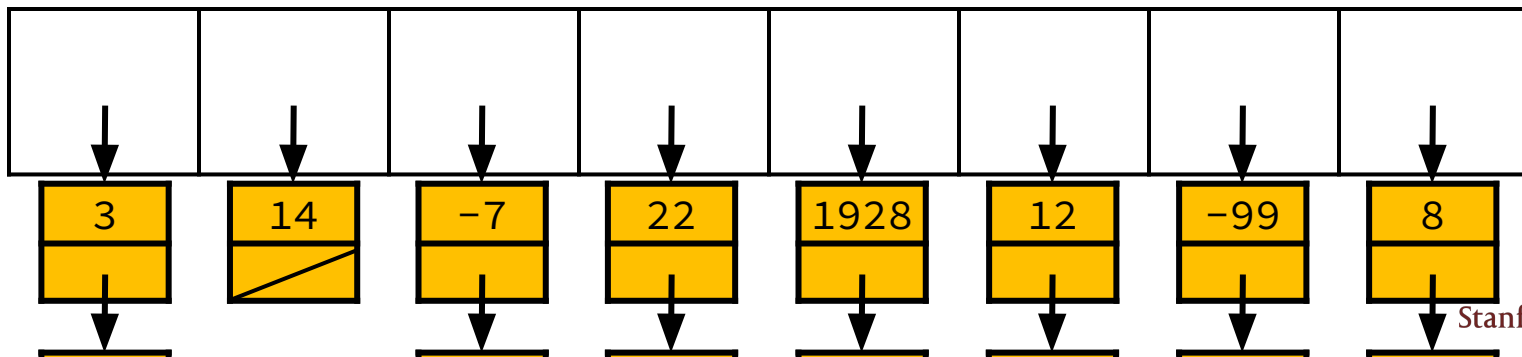
- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$



Chaining Hash Table

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket $hash(num)$

*If we've got a **good** hash function, and we've hashed n elements into b buckets, what's our average bucket size?*



Load Factor: n/b

- The average number of elements in each bucket
 - If the load factor is low: lots of empty buckets, wasted space
 - If the load factor is high: very full buckets, slow operations
- This means we'll have to look through $O(n/b)$ elements for `contains` and `remove`... is this better than $O(n)$?

Load Factor: n/b

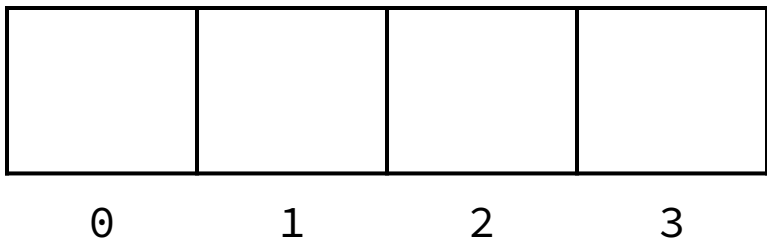
- The average number of elements in each bucket
 - If the load factor is low: lots of empty buckets, wasted space
 - If the load factor is high: very full buckets, slow operations
- This means we'll have to look through $O(n/b)$ elements for `contains` and `remove`... is this better than $O(n)$?

Big idea: if we choose b (# of buckets) to be a number close to n , then n/b will be constant.

Hashing Walkthrough

Let's walk through the operations of a Chaining Hash Table. This works much like a Chaining Hash Set, but we'll allow duplicates.

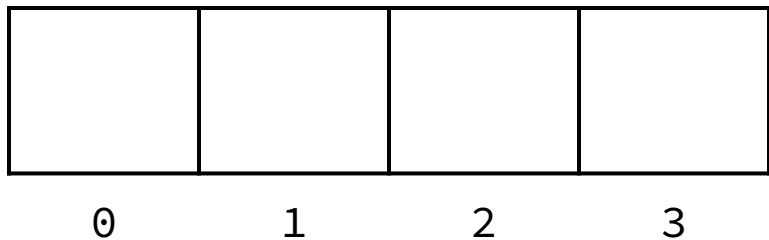
We'll begin with 4 buckets. We'll keep a load factor (n/b) of $\frac{3}{4}$ or less. This means that our ratio of elements to buckets cannot exceed $\frac{3}{4}$.



Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`



`numElements = 0`
`numBuckets = 4`

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

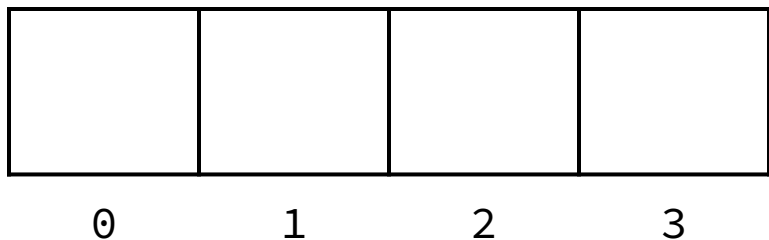
Add -4

Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`

$$(3 - 1) \% 4 = 2$$



`numElements = 0`
`numBuckets = 4`

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

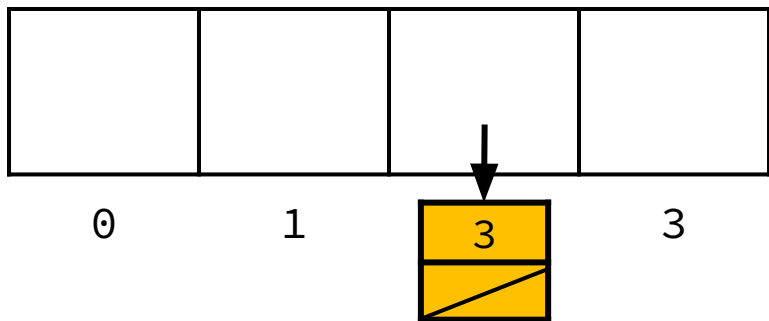
Add -4

Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(3 - 1) \% 4 = 2$$



numElements = 1
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

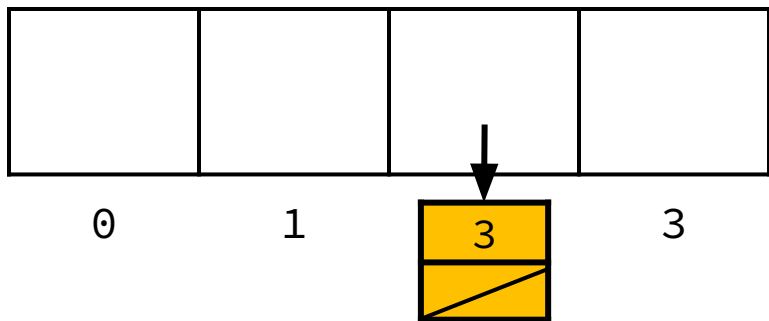
Add -4

Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(2 - 1) \% 4 = 1$$



numElements = 1
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

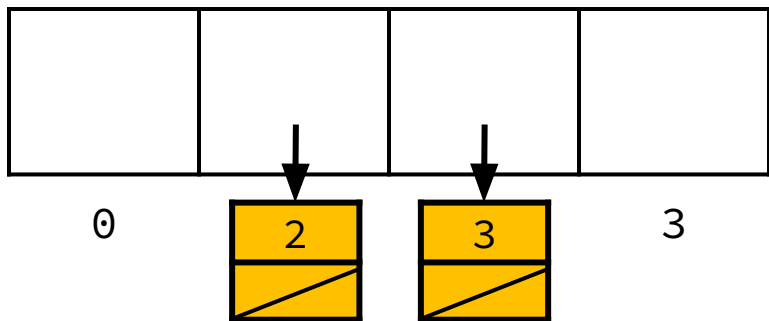
Add -4

Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`

$$(2 - 1) \% 4 = 1$$



`numElements = 2`
`numBuckets = 4`

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

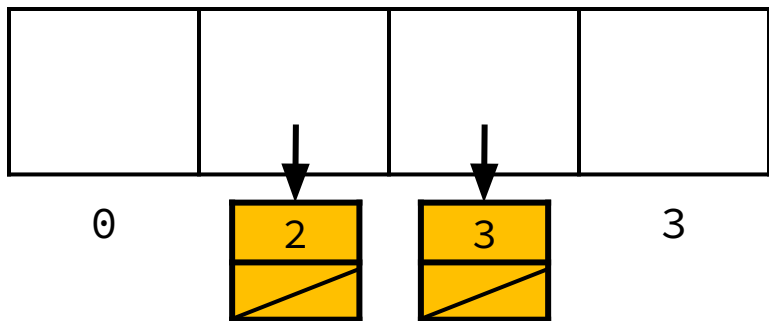
Add -4

Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(1 - 1) \% 4 = 0$$



numElements = 2
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

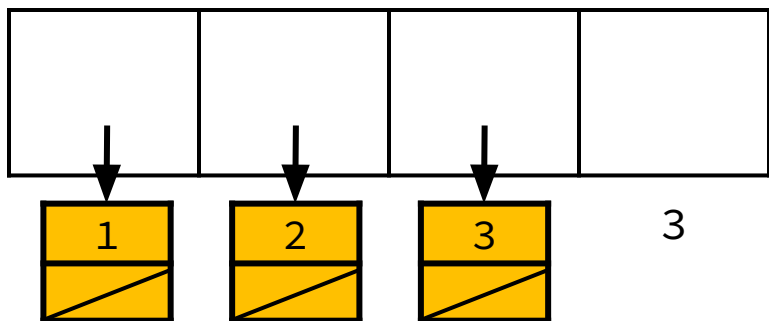
Add -4

Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`

$$(1 - 1) \% 4 = 0$$



`numElements = 3`
`numBuckets = 4`

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

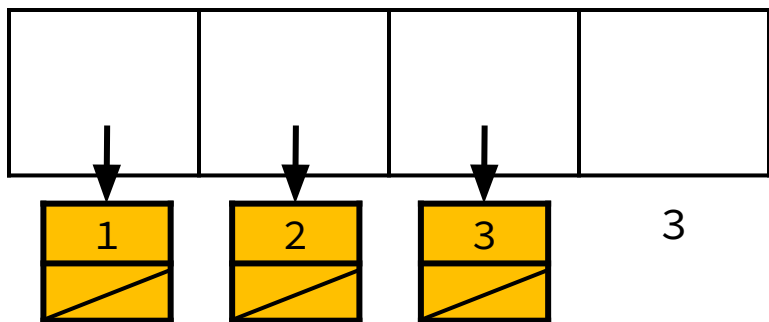
Add -4

Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(5 - 1) \% 4 = 0$$



numElements = 3
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

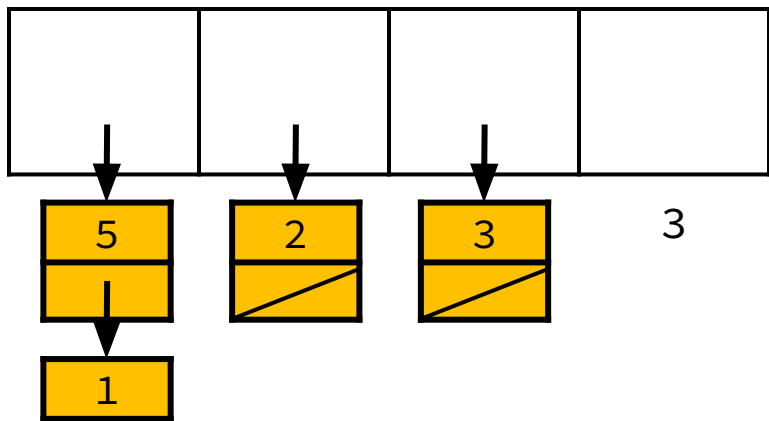
Add -4

Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(5 - 1) \% 4 = 0$$



numElements = 4
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

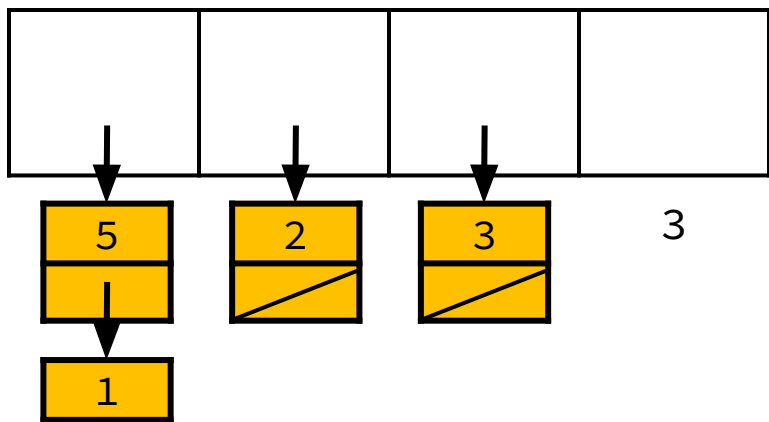
Add 1

Add -4

Hashing Walkthrough

Our load factor, $\frac{3}{4}$, has been exceeded! To reduce it, we double the number of buckets.

$$(5 - 1) \% 4 = 0$$



numElements = 4
numBuckets = 4

Add 3

Add 2

Add 1

Add 5

Remove 2

Add 1

Add -4

Hashing Walkthrough

Our load factor, $\frac{3}{4}$, has been exceeded! To reduce it, we double the number of buckets.

$$(5 - 1) \% 4 = 0$$

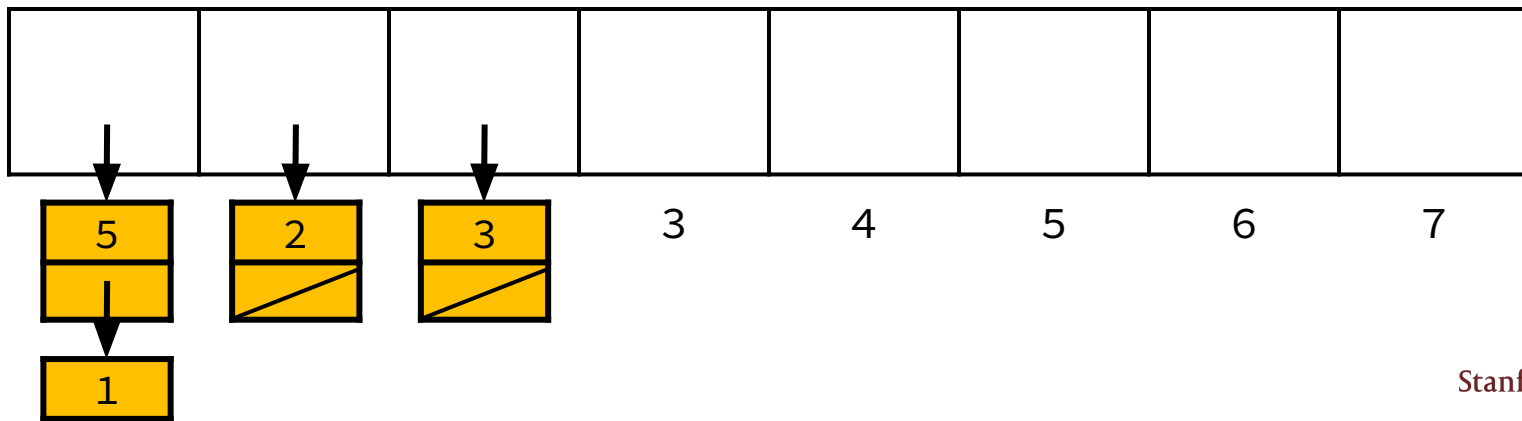
numElements = 4
numBuckets = 8

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

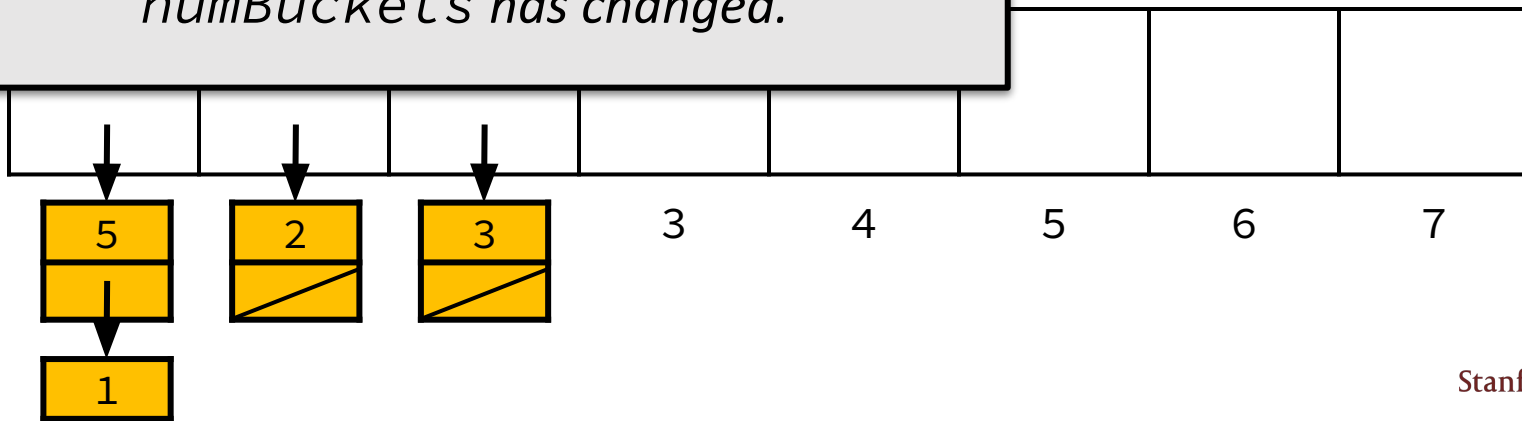
Add 5

Remove 2

Add 1

Add -4

*Now, we must **rehash** our elements, since our numBuckets has changed.*



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

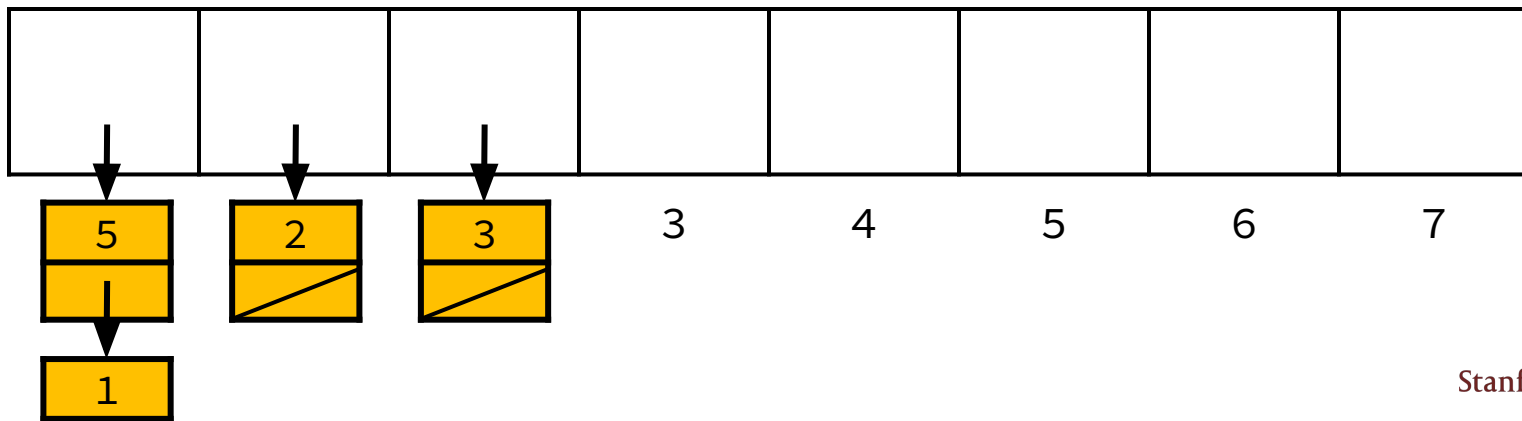
$$(5 - 1) \% 4 = 0$$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

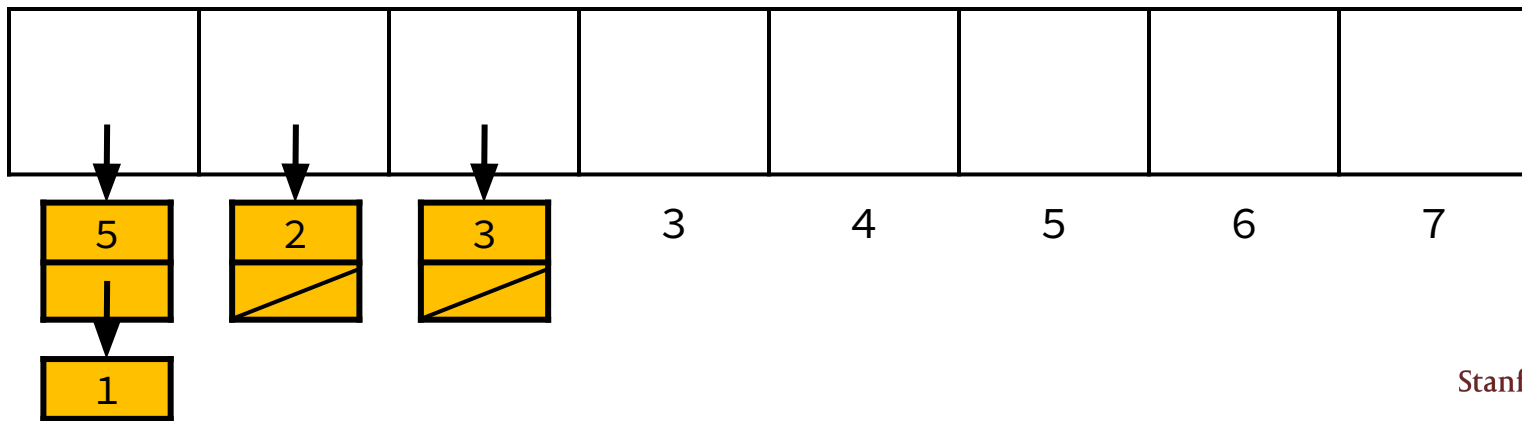
$$(5 - 1) \% 8 = 4$$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

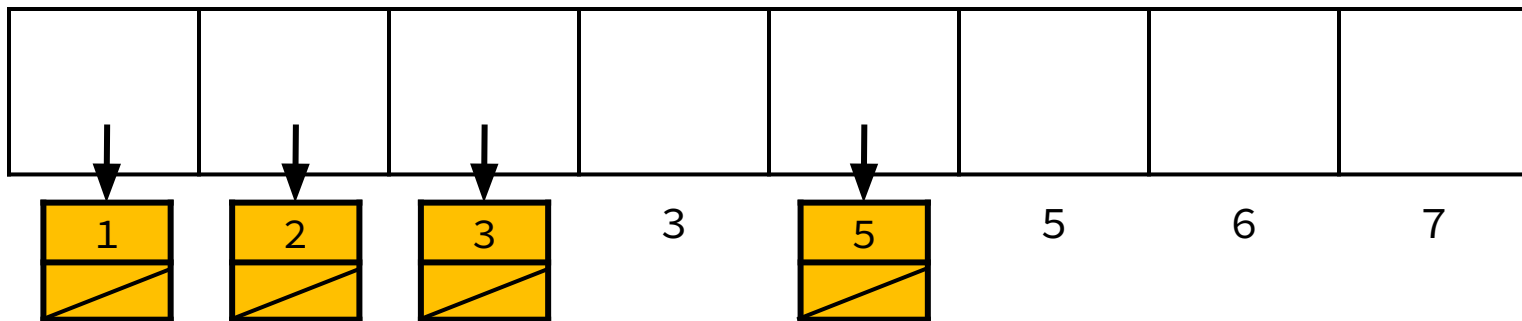
$$(5 - 1) \% 8 = 4$$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

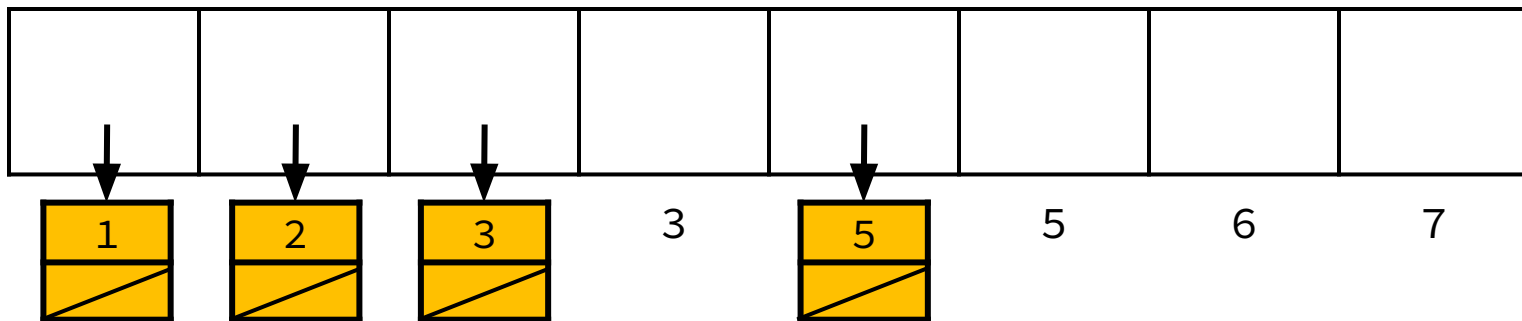
$(1 - 1) \% 8 = 0$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

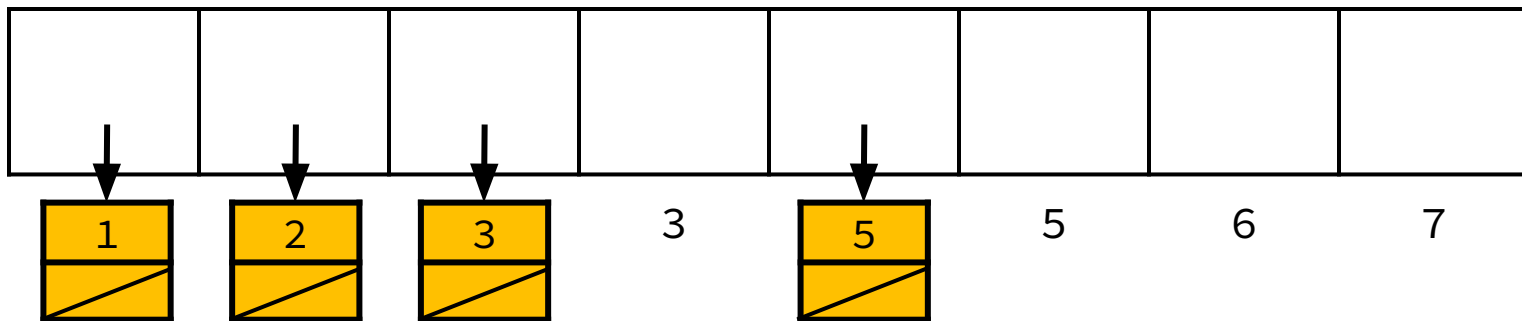
$(2 - 1) \% 8 = 1$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

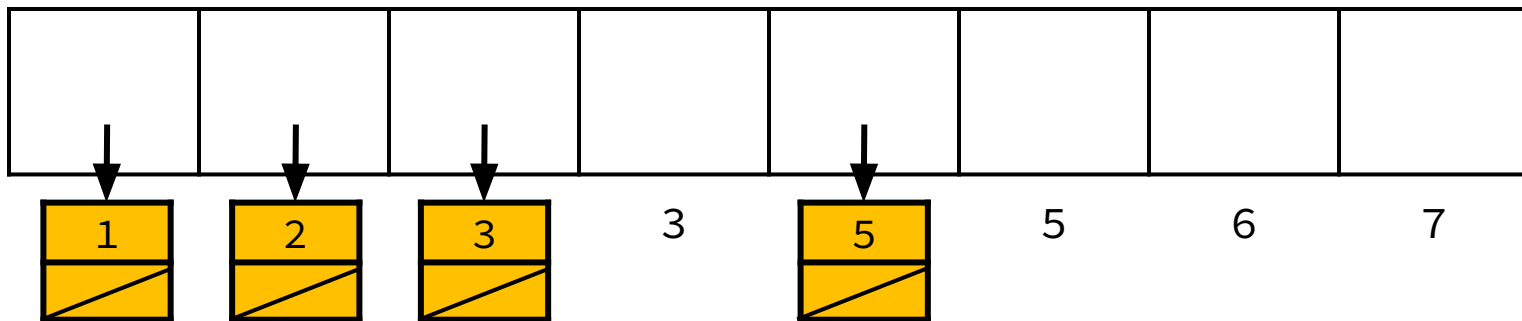
$$(3 - 1) \% 8 = 2$$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`

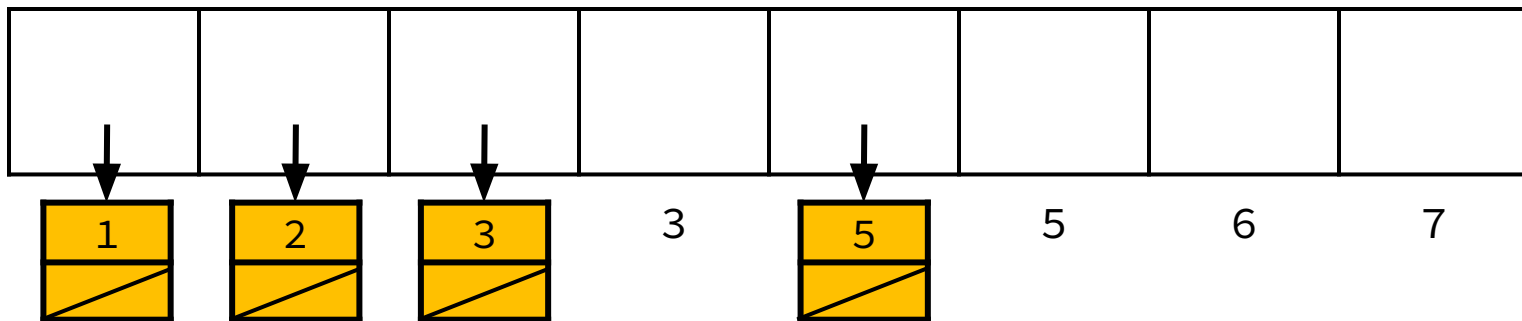
`numElements = 4`
`numBuckets = 8`

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

`hash(elem):`

`(elem - 1) % numBuckets`

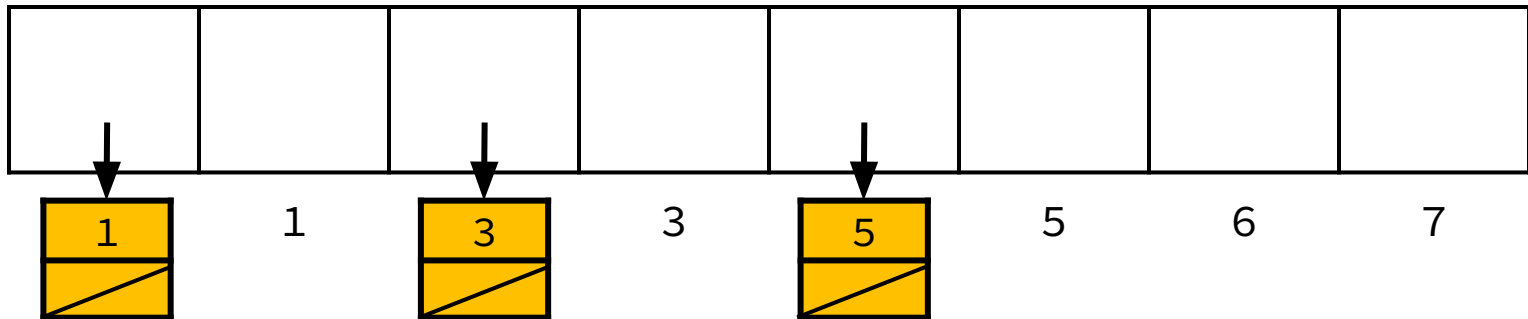
`numElements = 3`
`numBuckets = 8`

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

$$(1 - 1) \% 8 = 0$$

numElements = 3

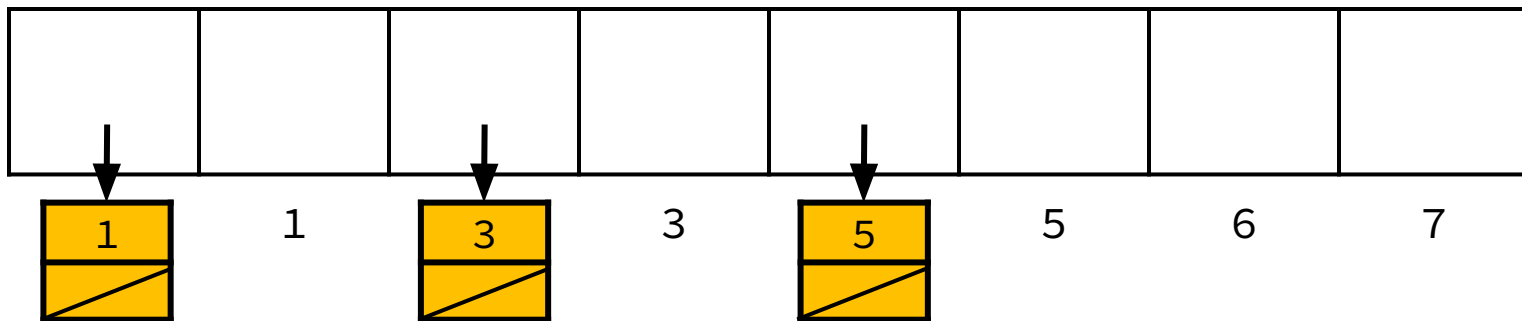
numBuckets = 8

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

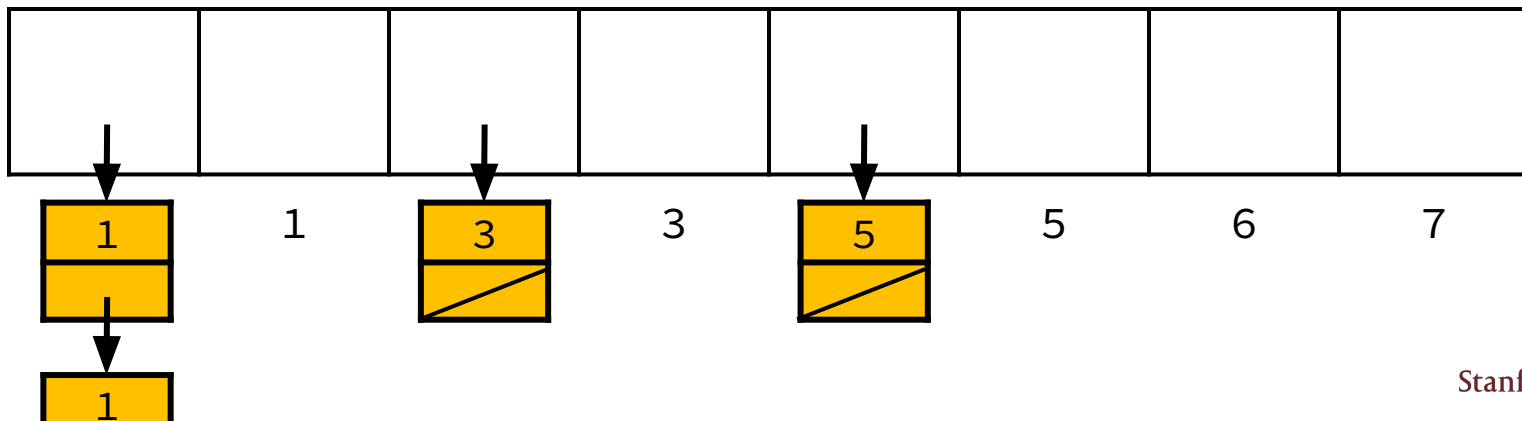
$(1 - 1) \% 8 = 0$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 4
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

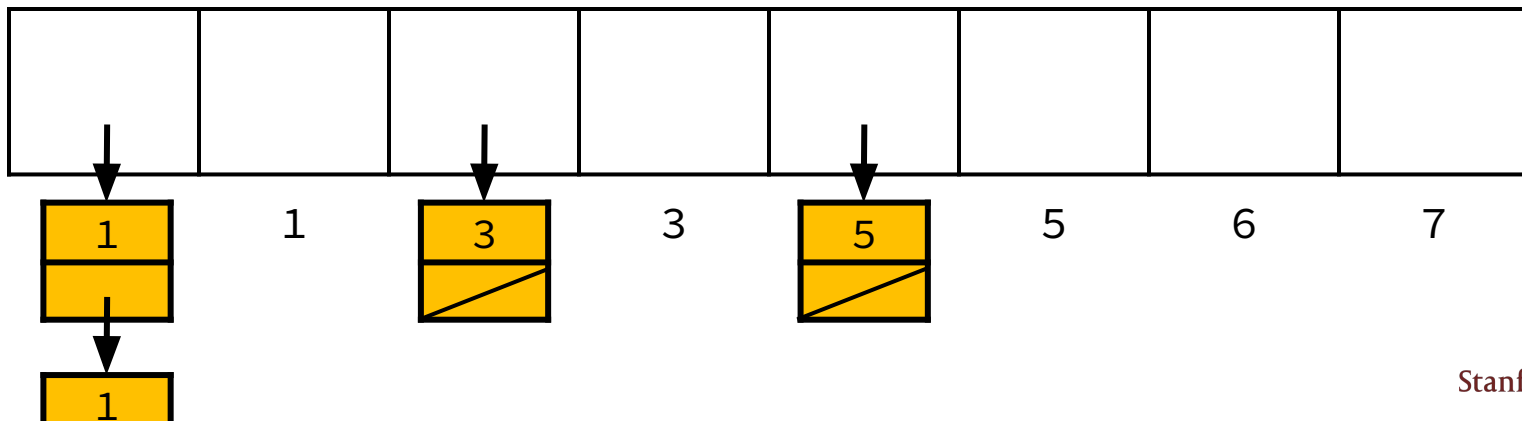
$$(-4 - 1) \% 8 = 3$$

Add 5

Remove 2

Add 1

Add -4



Hashing Walkthrough

numElements = 5
numBuckets = 8

hash(elem):

$(\text{elem} - 1) \% \text{numBuckets}$

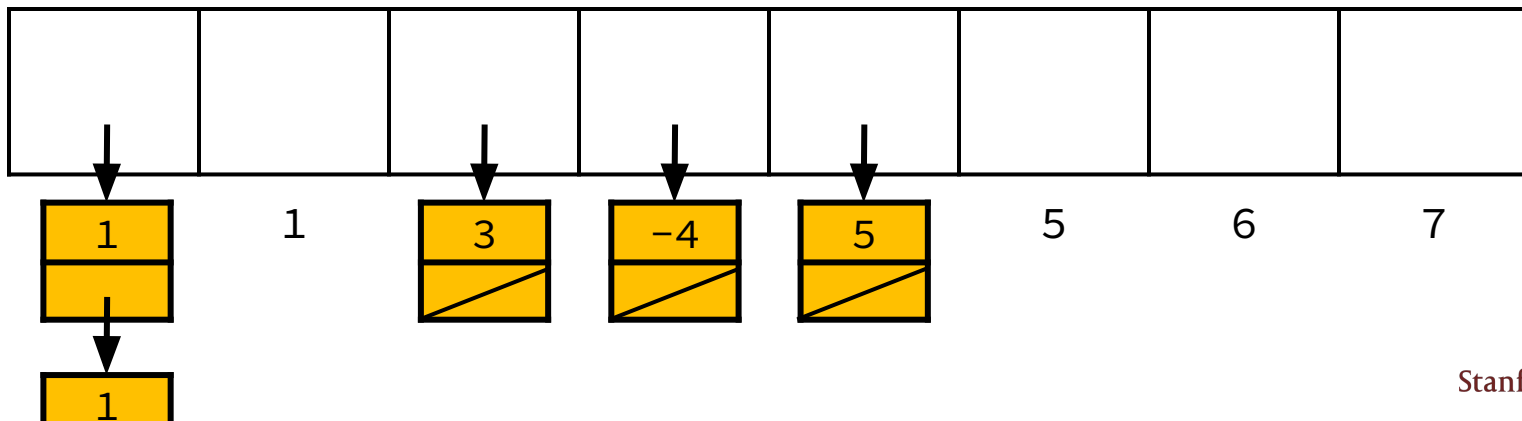
$$(-4 - 1) \% 8 = 3$$

Add 5

Remove 2

Add 1

Add -4



Practice Problem: Multiple Choice

Q1. What happens to our load factor when we increase the number of buckets in our hash table?

Practice Problem: Multiple Choice (Solutions)

Q1. The load factor is n/b (# elems / # buckets), so increasing b will make the load factor **smaller**. More intuitively, spreading the elements across more buckets means there will be fewer elements in each bucket.

Practice Problem: Multiple Choice

Q2. Let's say we have a hash function `hash` that we're using to determine which bucket to place strings into:

```
bucket = hash(input);
```

Suppose we compute the following three hash values:

```
A = hash("desert");
```

```
B = hash("dessert");
```

```
C = hash("brownie");
```

Which of the following are guaranteed to be true about these values (select all that apply):

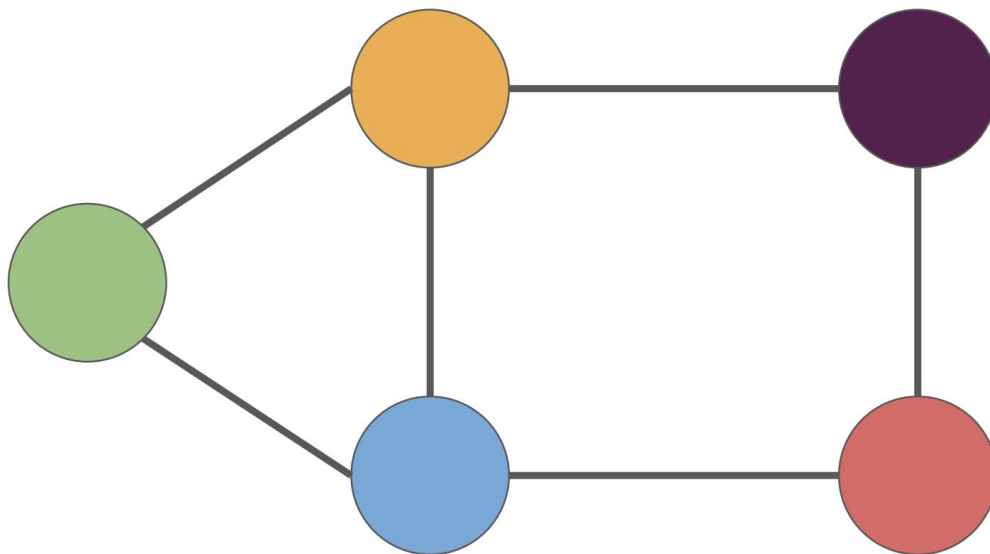
1. The values A and B will be closer together than A and C since "desert" and "dessert" are more similar.
2. The values B and C will be closer together than A and C since "dessert" and "brownie" are the same length.
3. The values A, B, and C are not equal.
4. None of the above.

Practice Problem: Multiple Choice (Solutions)

Q2. 4. Without knowing anything about our hash function, we have no guarantees on how the input string will be related to its output hash value - this is a property of good hash functions! Although good hash functions will spread elements evenly between buckets, we also aren't guaranteed that two (or even three) different elements won't hash to the same bucket.

Graphs

Graph Terminology



graph

a structured way to represent relationships between different entities

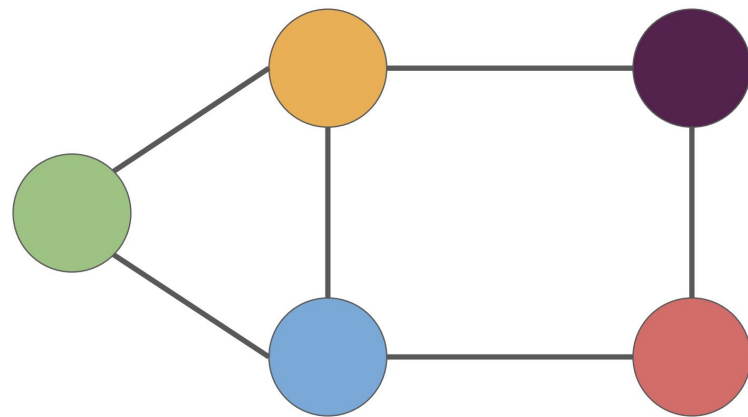
A graph consists of
a set of **nodes**
connected by **edges**.

Linked Data Structures

- We've already seen nodes connected by edges before when discussing linked lists and trees
- What differentiates these linked data structures?
 - **Linked lists:** Linear structure, each node connected to at most one other node
 - **Trees:** Nodes can connect to multiple other nodes, no cycles, parent/child relationship and a single, special root node.
 - **Graphs:** No restrictions. It's the wild, wild west of the node-based world!

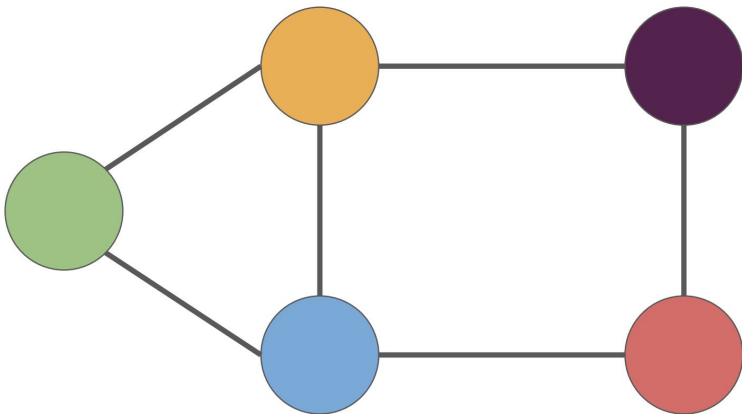
Wild World of Graphs

- Can have cycles
- No notion of a parent-child relationship between nodes
- No root node
- Most powerful, flexible, and expressive abstraction that we can use to model relationships between different distributed entities




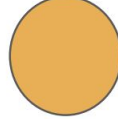
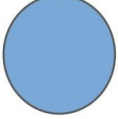





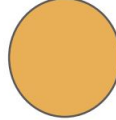

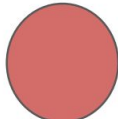
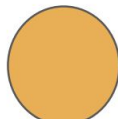
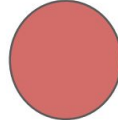

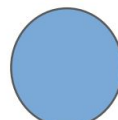


Approach 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.

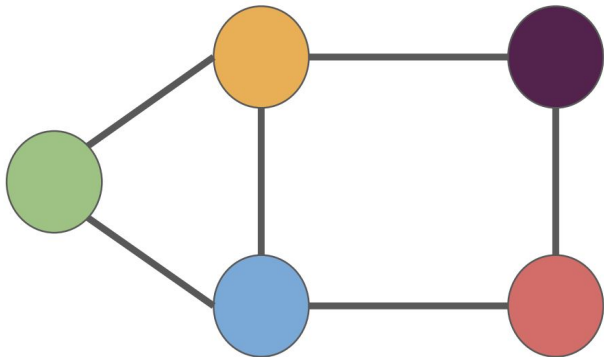







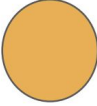


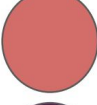

Map<Node, Set<Node>>

Node	Set<Node>>
Node	Adjacent to
	 
	  
	  
	 
	 

Approach 2: Adjacency Matrix

- We can also use a two-dimensional matrix to represent the relationships in a graph.



					
		1	1	0	1
	1		1		
	1	1		1	
	0		1		1
	1			1	

DFS Algorithm

```
dfs-from(node v) {  
    make a stack of nodes, initially seeded with v.  
    while the stack isn't empty:  
        pop a node curr.  
        process the node curr.  
        for each node adjacent to curr:  
            if that node has never been pushed:  
                push that node.  
}
```

BFS Algorithm

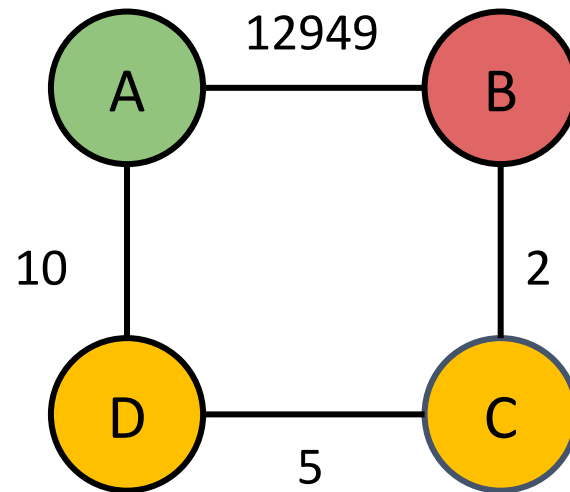
```
bfs-from(node v) {  
    make a queue of nodes, initially seeded with v.  
    while the queue isn't empty:  
        dequeue a node curr.  
        process the node curr.  
        for each node adjacent to curr:  
            if that node has never been enqueued:  
                enqueue that node.  
}
```

BFS vs DFS

- Running BFS or DFS from a node in a graph will visit the same set of nodes, but probably in a different order
- BFS will visit nodes in increasing order of distance
 - Will give you the **shortest path**
- DFS does visit nodes in some interesting order, but not order of distance
 - Take CS161 for more info

Shortest Weighted Path

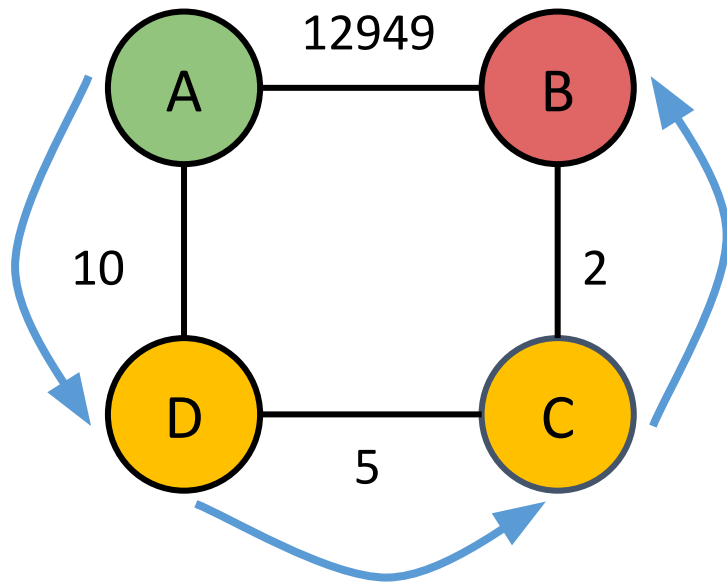
What is the shortest **weighted** path from A to B?



Shortest Weighted Path

What is the shortest **weighted** path from A to B?

- BFS doesn't work here



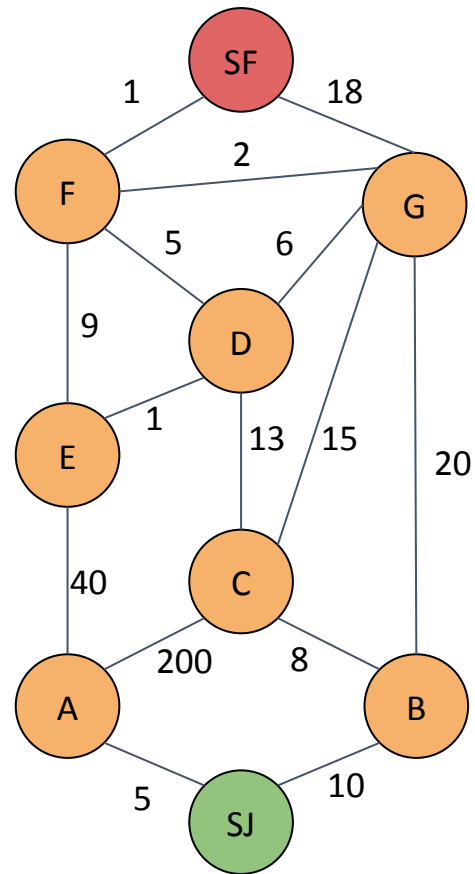
Dijkstra's Algorithm

- Finds the shortest weighted path from one node to another
- Greedy algorithm
 - Prioritizes finding a solution by what is "best right now"
 - Looks at its options and always chooses whatever gets it closer to a solution in the best possible way given the current situation
 - Ex: Change We Can Believe In (Section 4, Problem 2)
- Many different ways to model this
 - Can use a priority queue, where weights become priorities
 - Can use a table of nodes
- Real world uses: shortest paths on maps (Ethiopia), tracks of electricity lines and oil pipelines, network routing protocols

Dijkstra's Algorithm

Algorithm:

1. Of the unseen nodes, find the node that currently has the shortest distance from the start
2. Look at this node's neighbors, and update the total distance to the neighbors based on their distance and the distance already to this node.
3. If the node visited is the destination, stop
4. Repeat from step 1



A* Algorithm

- Finds the shortest weighted path from one node to another
- Uses external information about the graph
- Heuristic: estimates the cost of the cheapest path to the goal
 - Should always underestimate the distance to the goal, because if it overestimates, it could find a non-optimal solution
- If the distance to the destination is closer, weight the nodes in that direction to be preferable
 - $\text{priority}(u) = \text{weight}(s, u) + \text{heuristic}(u, d)$

Recap

- Graphs are a linked data structure with almost no rules
 - Represent in code with either an adjacency list or matrix
- Depth-First Search: does not always return the shortest path, though it may be faster in some cases
- Breadth-First Search: returns the shortest path, but it only works on unweighted graphs
- Dijkstra's Algorithm: returns the shortest weighted path, but not necessarily the most efficient
- A* Algorithm: returns the shortest weighted path using heuristics, and is often thought of as gold standard

Good luck on the final!
You're almost done with CS106B!!