

EE599: Computing and Software for Systems Engineers

University of Southern California

Spring 2020

Instructor: Arash Saifhashemi
Ari Saif

Who Is This Course For?

Preparing you for **design**, **analysis**, and
implementing a **complete** software **system**

Two 2 unit courses:

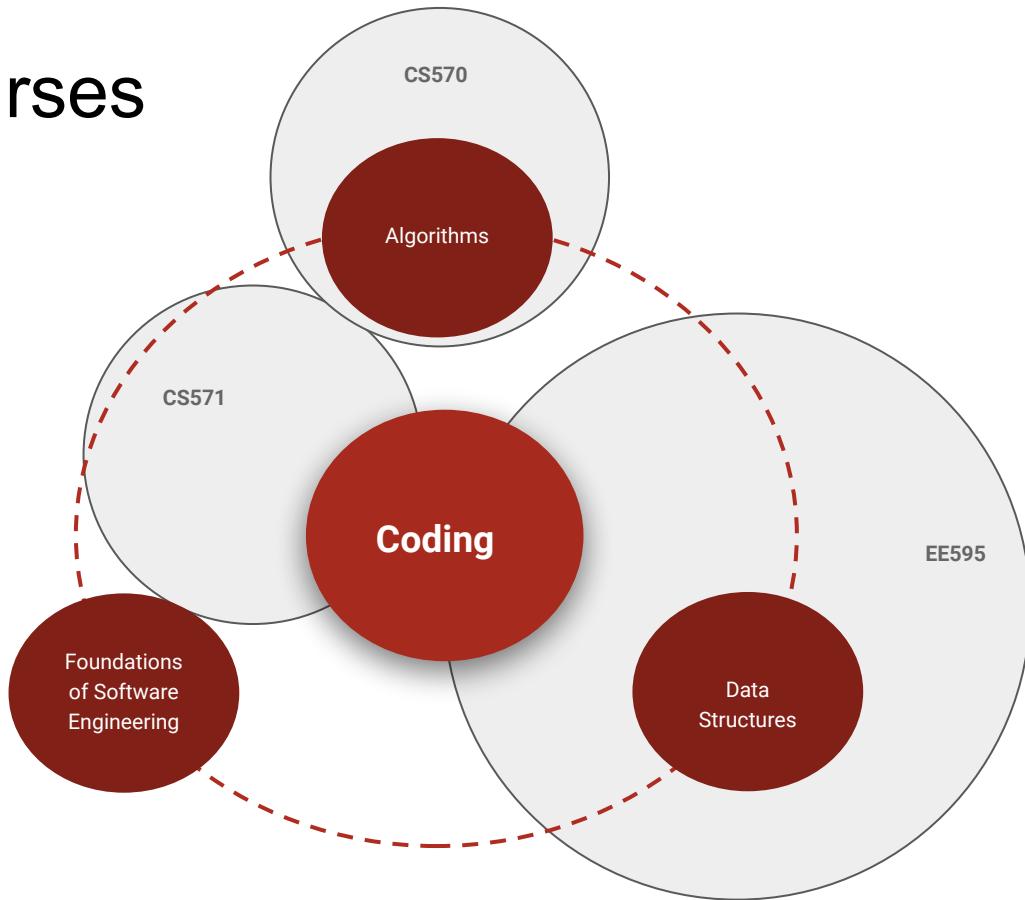
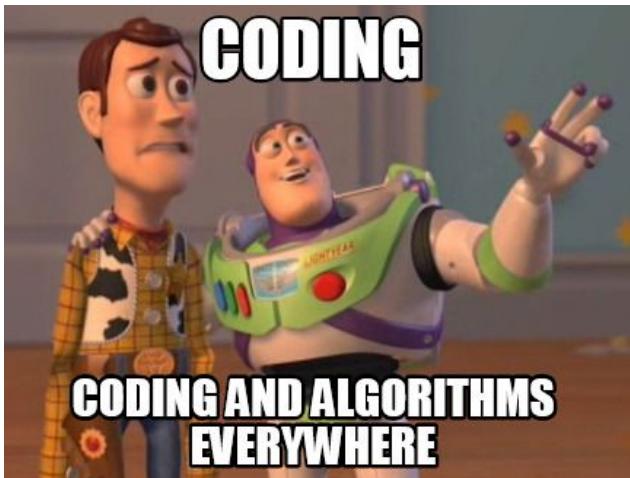
- Spring 2020
- Fall 2020

Introduction

- Coding
 - Basics of C++, HTML, JavaScript, Python
- Foundations of Software Engineering
 - Testing, Source Control, Shell scripts
 - Modular Programming
 - Object Oriented Programming
- Basics of Algorithms and Data Structure
 - Trees, linked-lists, hash tables, heaps, ...
 - Runtime analysis
 - Algorithm design and analysis
 - Greedy, recursive, dynamic programming, ...
- Cloud Computing
 - Virtualization, Databases (SQL, NoSQL), Security
 - AWS, Google Cloud

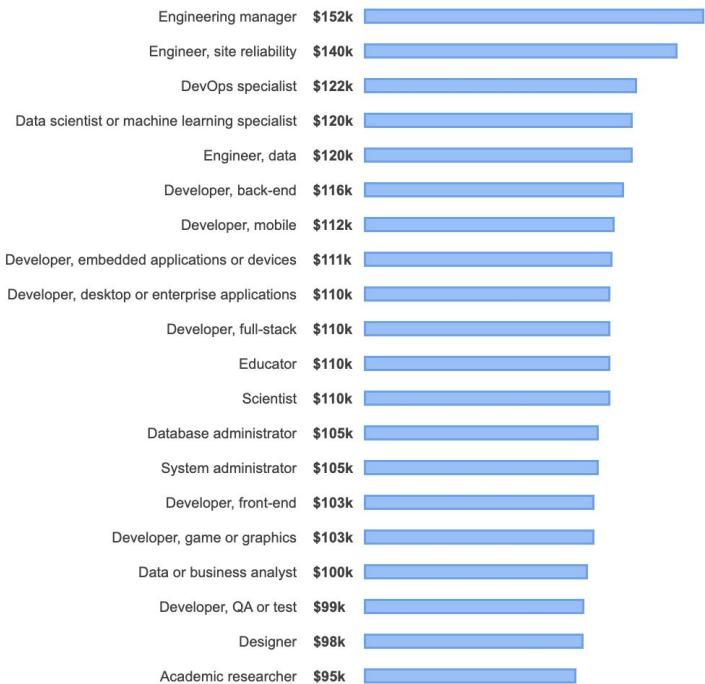
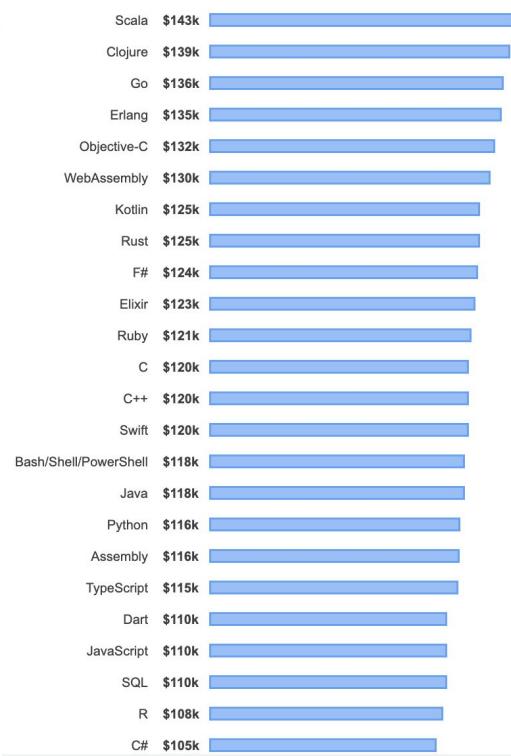
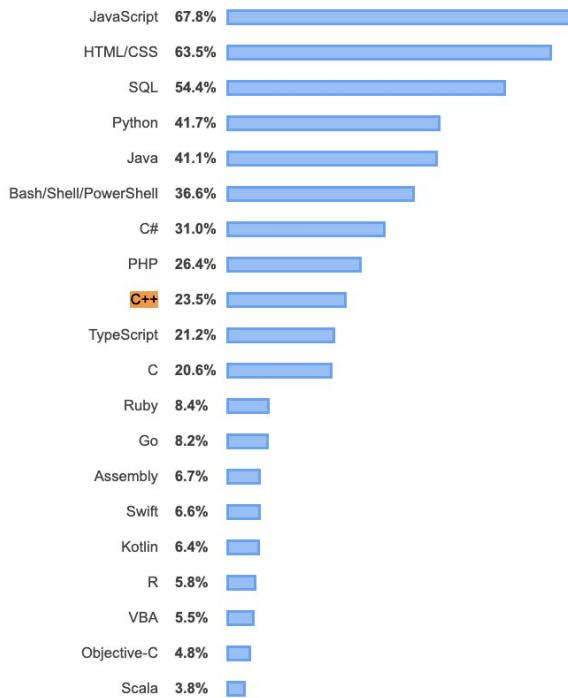
Overlap with Other Courses

- We are focused on coding:
 - Almost all algorithms discussed should be implemented and tested.



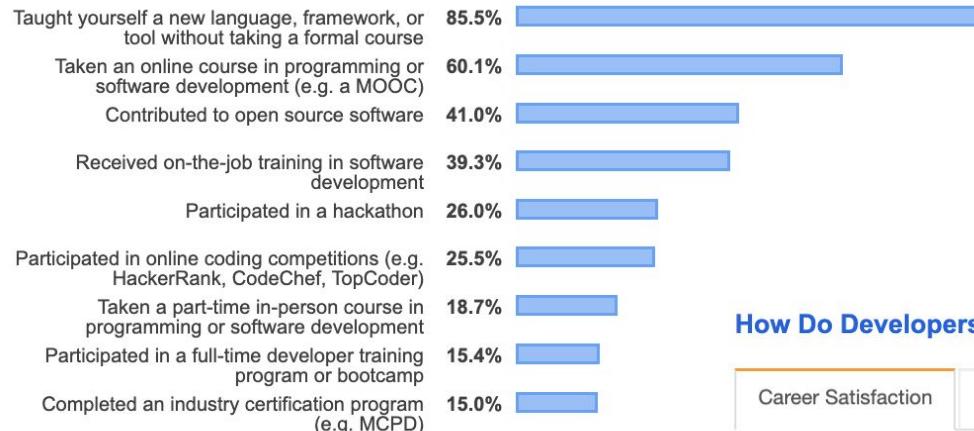
This is a coding-oriented course!

Some Programming Trends (2019)



[Source](#)

Some Programming Trends (2019)



84,260 responses; select all that apply

How Do Developers Feel About Their Careers and Jobs?



[Source](#)

Programmer Basic Tools

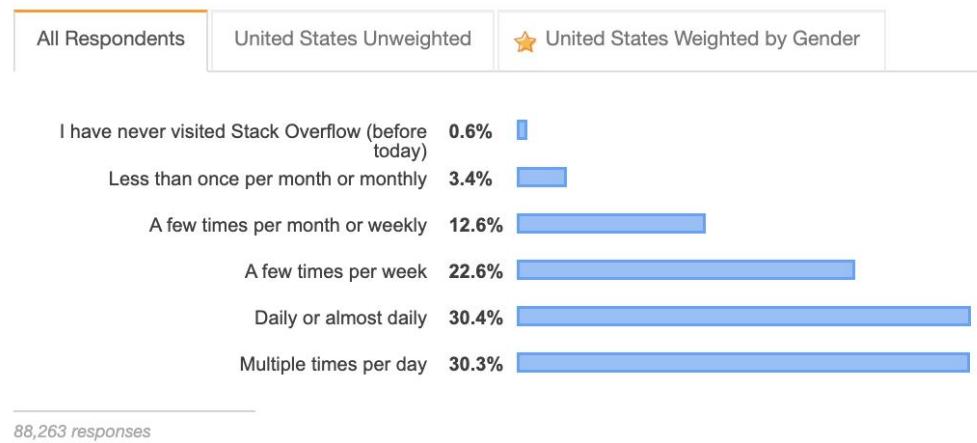
- We should all have:
 - Stackoverflow account
 - Github account
- Please make sure you install:
 - [Visual Studio Code](#)
 - [Git](#)
 - Linux-compatible terminal
 - C++ Toolchain
 - Mac: Xcode
 - Windows: [Cywin](#) or [MinGW](#)



Programmer Basic Tools

- Before asking any questions, consult:
 - Google
 - Stackoverflow
- Do not blindly post your homework assignments on Stackoverflow or Blackboard
 - Instead, start with something yourself and ask for help as you go

Visiting Stack Overflow



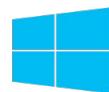
[How to ask a good question?](#)

Where is C++ Used?



Adobe
Illustrator

Google



Windows



Mac OS



mozilla
Firefox®

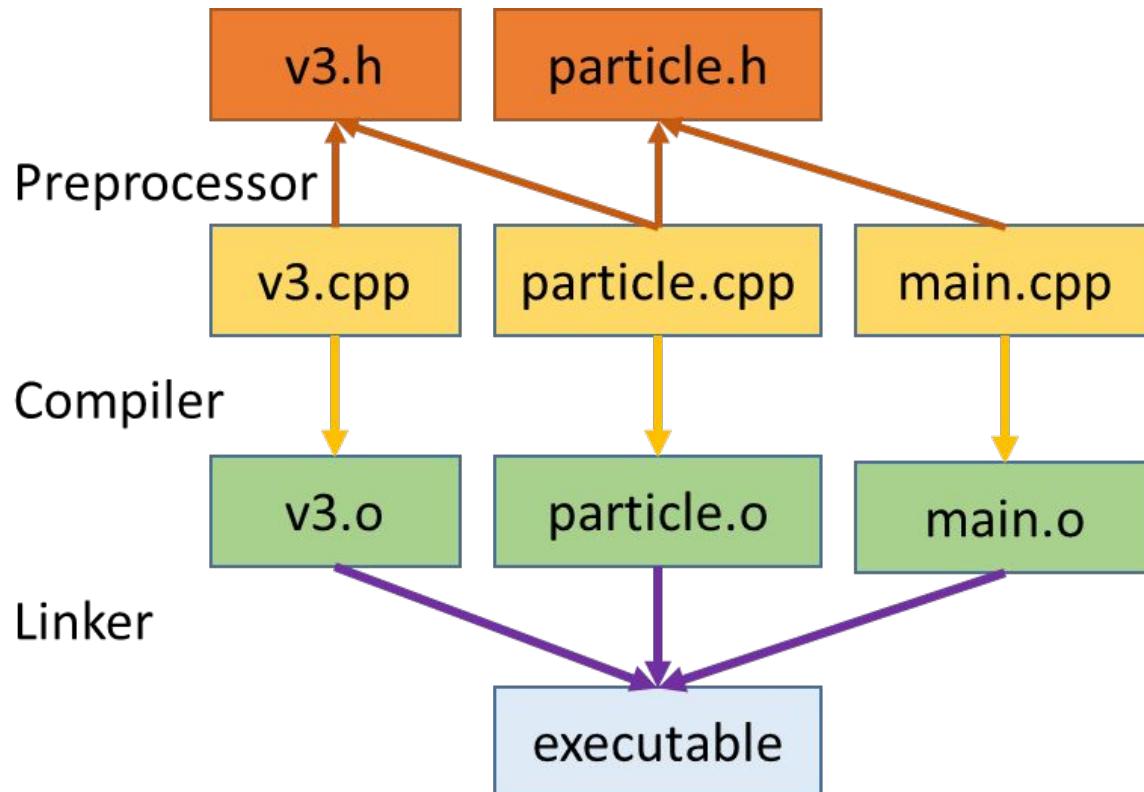


MySQL®



Office

C++ Needs a Compiler



Example

Find the maximum value in an array of integers

- **Step 1:** Clearly define the input and output of the problem

```
int FindMax(std::vector<int> &inputs);
```

Example

Find the maximum value in an array of integers

- **Step 2:** What are some example input/outputs?
 - Find corner cases

inputs = {1,2,3,4}, output = 4

inputs = {1}, output= 1

inputs = { }, output= ?

inputs = {1,1,1,1}, output = ?

Example

Find the maximum value in an array of integers

- **Step 3:** Propose an algorithm
 - We will learn various techniques in this course

```
int FindMax(std::vector<int> &inputs)
{
    int result = INT32_MIN;
    for (auto n : inputs) {
        if (n > result) {
            result = n;
        }
    }
    return result;
}
```

Example

Find the maximum value in an array of integers

- Step 4: Test your algorithm

inputs = {1, 2, 3, 4}, output = ?

inputs = {1}, output= ?

inputs = {}, output= ?

inputs = {1, 1, 1, 1}, output = ?

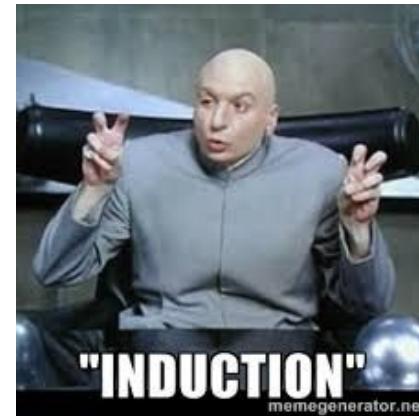
```
int FindMax(std::vector<int> &inputs) {
    int result = INT32_MIN;
    for (auto n : inputs) {
        if (n > result) {
            result = n;
        }
    }
    return result;
}
```

Example

Find the maximum value in an array of integers

- **Step 5:** Prove its correctness

- Induction
- Contradiction
- Other techniques



```
int FindMax(std::vector<int> &inputs) {  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

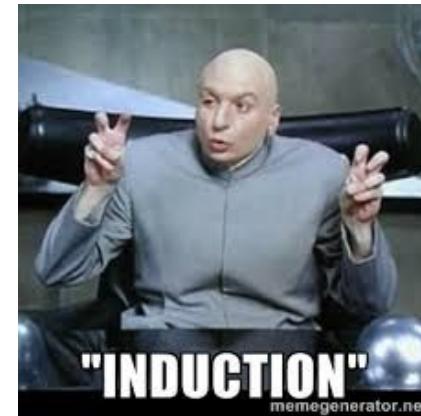
Example

Find the maximum value in an array of integers

- **Step 5:** Prove its correctness

- Induction
- Contradiction
- Other techniques

$I = 0$



```
int FindMax(std::vector<int> &inputs) {  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

Proof By Induction

- (Base Case) Show the statement is true for $k=1$.
- (Inductive Step) Show that if the statement is true for k , this implies the statement is true for $k+1$.



Example

Find the maximum value in an array of integers

- Step 5:
 - Proof by Induction
 - Proof by Contradiction

```
int FindMax(std::vector<int> &inputs) {  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

Proof by Induction: We prove value of *result* at step *i* is the max of elements 0 to *i*.

- Base case: $i = 0$
- Inductive step:
 - *result* is max of $[0, \dots, i]$, can we say *result* will be updated to max of $[0, \dots, i+1]$?

Example

Find the maximum value in an array of integers

- **Step 6:** Systematic Testing

```
int FindMax(std::vector<int> &inputs) {  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

Why ?

- Even with mathematical proof, there might be **implementation bugs**
- In practice, we may not be able to mathematically prove the correctness

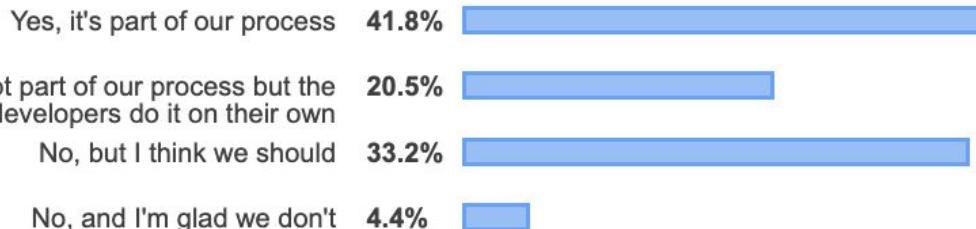
Unit Tests



Unit Tests

Does Your Company Employ Unit Tests?

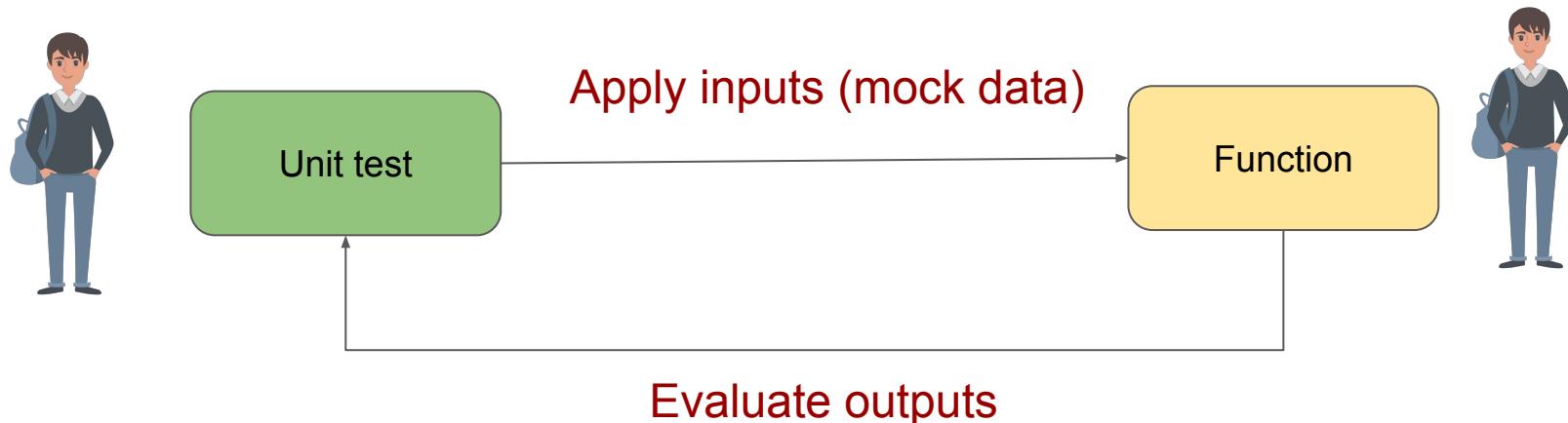
How are Job Satisfaction and Unit Tests Related?



62,668 responses

Unit Tests

- A unit test is a **piece of code** that tests a **function** or a **class**
- Unit tests are written by the **developer!**



Advantages?

Disadvantages?

Google Test Platform

- A testing framework for C++ code
- Automates various tasks:
 - Creates a main function
 - Calls our function under test
 - Applies inputs
 - Provides various functions for testing

```
// Tests factorial of 0.  
TEST(FactorialTest, HandlesZeroInput) {  
    EXPECT_EQ(Factorial(0), 1);  
}  
  
// Tests factorial of positive numbers.  
TEST(FactorialTest, HandlesPositiveInput) {  
    EXPECT_EQ(Factorial(1), 1);  
    EXPECT_EQ(Factorial(2), 2);  
    EXPECT_EQ(Factorial(3), 6);  
    EXPECT_EQ(Factorial(8), 40320);  
}
```

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_TRUE(condition);	EXPECT_TRUE(condition);	condition is true
ASSERT_FALSE(condition);	EXPECT_FALSE(condition);	condition is false

- **ASSERT_*** yields a fatal failure and returns from the current function.
- **EXPECT_*** yields a nonfatal failure, allowing the function to continue running.

Example

Find the maximum value in an array of integers

- Step 6: Using Google Test

```
int FindMax(std::vector<int> &inputs) {  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

```
TEST(FindMaxTest, HandlesConsecutiveNumbers) {  
    Solution solution;  
    std::vector<int> inputs = {1, 2, 3, 4};  
    EXPECT_EQ(solution.FindMax(inputs), 4);  
}
```

```
TEST(FindMaxTest, HandlesSizeOne) {  
    Solution solution;  
    std::vector<int> inputs = {2};  
    EXPECT_EQ(solution.FindMax(inputs), 2);  
}
```

```
TEST(FindMaxTest, HandlesEmptyVector) {  
    Solution solution;  
    std::vector<int> inputs = {};  
    EXPECT_EQ(solution.FindMax(inputs), -1);  
}
```

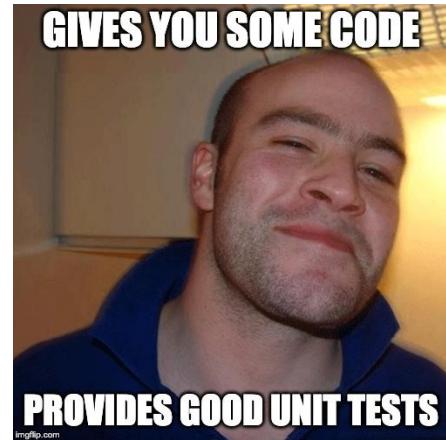
Google Test Platform

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(val1, val2);	EXPECT_EQ(val1, val2);	val1 == val2
ASSERT_NE(val1, val2);	EXPECT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPECT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPECT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPECT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

<https://github.com/google/googletest>

Unit Tests

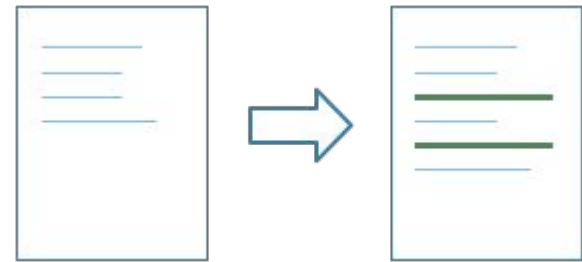
- Test should be **independent** and **repeatable**.
 - A test should not succeed or fail as a result of other tests.
- Tests should be **portable** and **reusable**.
 - They should work on different platforms
- Tests should be **fast**.
- Test should provide as much information about the problem as possible.



```
TEST(FindMaxTest, HandlesEmptyVector) {
    Solution solution;
    std::vector<int> inputs = {};
    EXPECT_EQ(solution.FindMax(inputs), 1)
        << "ERROR: The result of an empty vector was not -1";
}
```

Version Control

- Who made the change?
 - So you know whom to blame
- What has changed (added, removed, moved)?
 - Changes within a file
 - Addition, removal, or moving of files/directories
- Where is the change applied?
 - Not just which file, but which version or branch
- When was the change made?
 - Timestamp
- Why was the change made?
 - Commit messages



Tracking file changes

Git

- Started by Linus Torvalds – 2005
- Efficient for large projects
 - E.g. Linux
 - Much faster than other alternatives

```
GIT(1)                               Git Manual                               GIT(1)

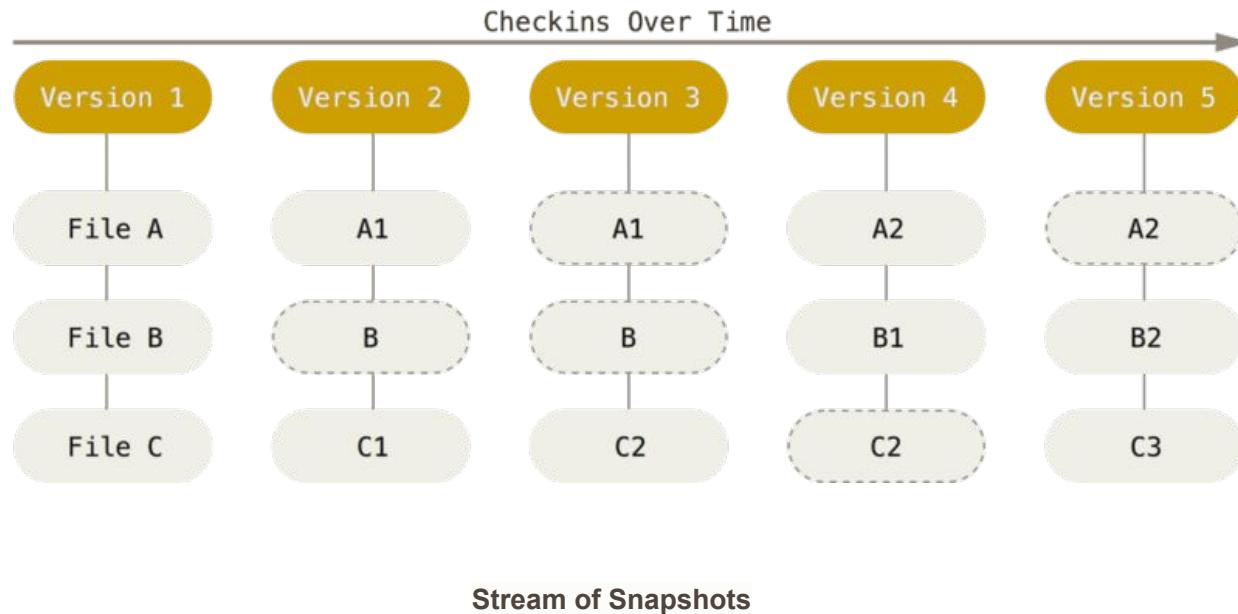
NAME
    git - the stupid content tracker

SYNOPSIS
    git [--version] [--help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        [--super-prefix=<path>]
        <command> [<args>]

DESCRIPTION
    Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full
    access to internals.

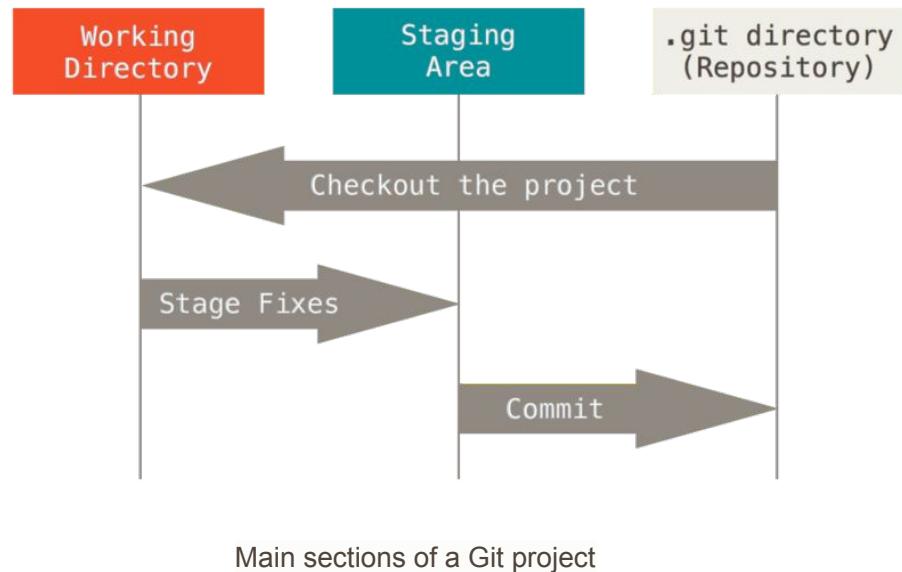
    See gittutorial\(7\) to get started, then see giteveryday\(7\) for a useful minimum set of commands. The Git User's Manual\[1\] has a more in-depth
    introduction.
```

Git



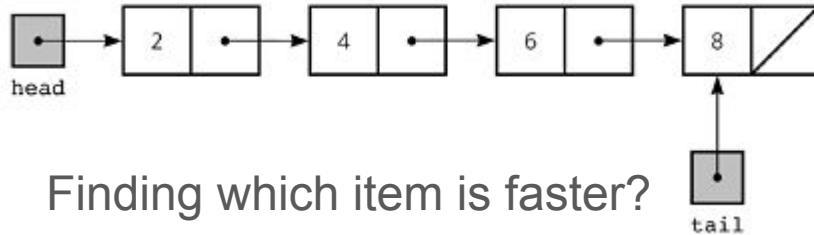
Git

- **Modified**
 - You have changed the file but have not committed it to your database yet.
- **Staged**
 - You have marked a modified file in its current version to go into your next commit snapshot.
- **Committed**
 - the data is safely stored in your local database.



Runtime Analysis

- How long does it take for our algorithm to finish?
- Depends on both **Algorithm** and **Input**



Finding which item is faster?

```
int FindMax(std::vector<int> &inputs) {  
    if (inputs.size() == 0) {  
        return -1;  
    }  
    int result = INT32_MIN;  
    for (auto n : inputs) {  
        if (n > result) {  
            result = n;  
        }  
    }  
    return result;  
}
```

The runtime grows with the size of the array

Runtime Analysis

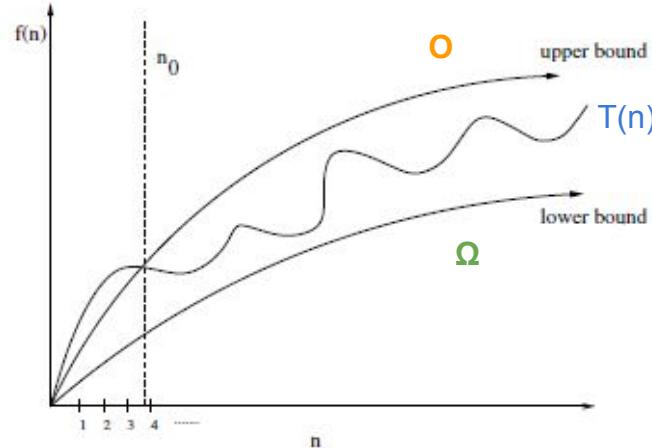
- It is hard to compare run times
 - Depends on the things like hardware, OS, ...
- Count number of operations
 - What is an operation?
 - $i++$
 - $i = i / 2$
 - $i = i + 2$
- For input I of size n : $R = F(I, n)$
- $T(n)$: worst case
 - Usually we care about the worst case input

```
int FindMax(std::vector<int> &inputs) {  
    if (inputs.size() == 0) { 1  
        return -1; 1  
    }  
    int result = INT32_MIN; 1  
    for (auto n : inputs) { 3  
        if (n > result) { 1  
            result = n; 1  
        }  
    }  
    return result; 1  
}
```

- Number of operations:
 - If $n=0$: 2
 - If $n>0$: $1 + 1 + 1 + n(3 + 1 + k) + 1$
 - K is between 0 and 1
 - Worst case: **$5n + 4$**

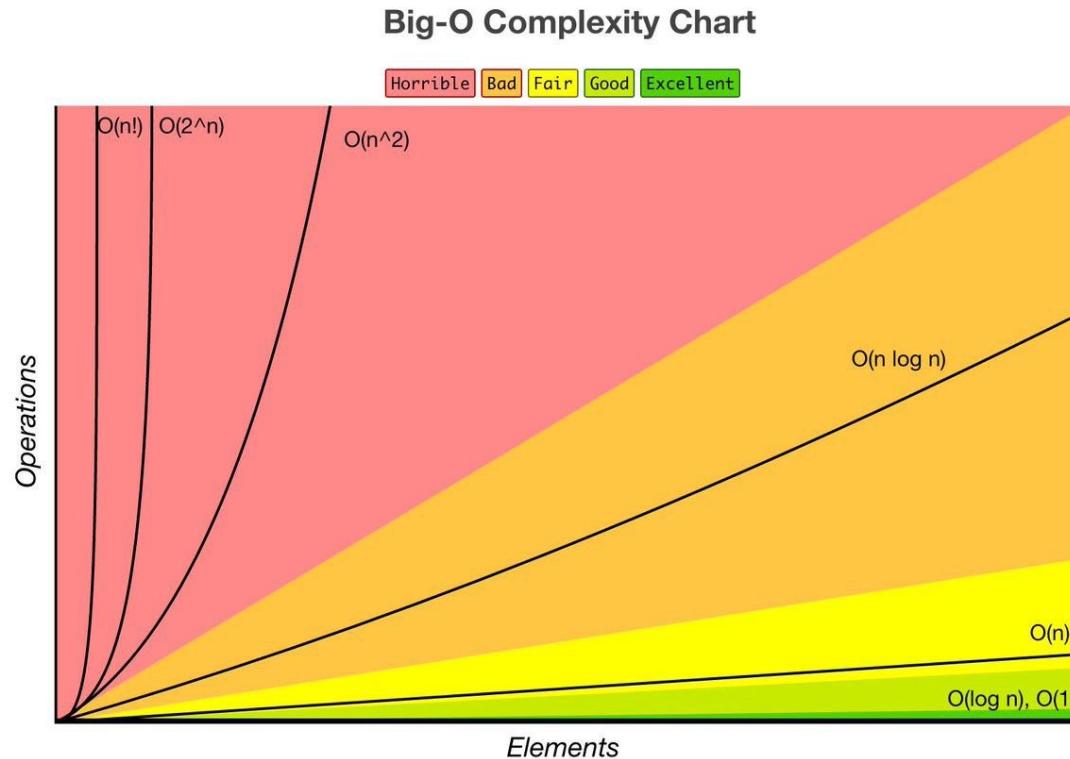
Big O, Ω , Θ

- Let a and n_0 be constants
- $T(n)$ is $O(f(n))$ if...
 - $T(n) < a \cdot f(n)$ for some $n > n_0$
 - where a and n_0 are constants
 - Essentially an upper-bound
- $T(n)$ is said $\Omega(f(n))$ if...
 - $T(n) > a \cdot f(n)$ for some $n > n_0$
 - Essentially a lower-bound
- $T(n)$ is said to be $\Theta(f(n))$ if...
 - $T(n)$ is both $O(f(n))$ AND $\Omega(f(n))$



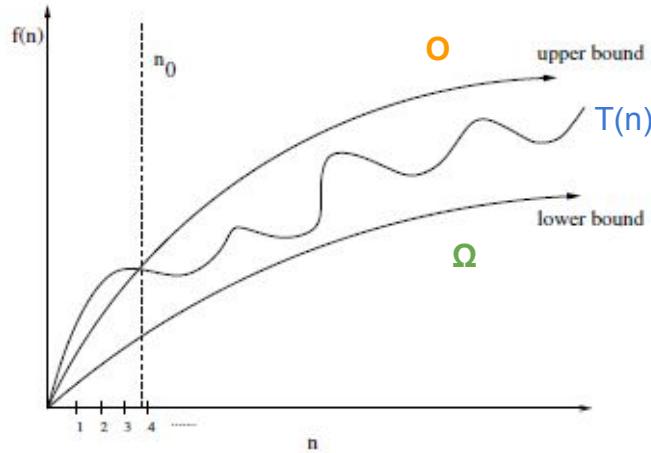
- For input I of size n : $R = F(I, n)$
- $T(n)$: worst case
 - Usually we care about the worst case input

Big O, Ω, Θ



Some examples

- $T(n) = f(n) + c = O(f(n))$
- $T(n) = c.f(n) = O(f(n))$
- $T(n) = n^2 + n = O(n^2)$
- $T(n) = n^2 + \log(n) = O(n^2)$
- $T(n) = n + n\log(n) = O(n\log(n))$



Runtime Analysis

```
int FindMax(std::vector<int> &inputs) {
    if (inputs.size() == 0) {           1
        return -1;                   1
    }
    int result = INT32_MIN;          1
    for (auto n : inputs) {          2
        if (n > result) {           1
            result = n;             1
        }
    }
    return result;                  1
}
```

- $T(n)$:

- If $n=0$: 2
- If $n>0$: $1 + 1 + 1 + n(2 + 1 + k) + 1$
 - K is between 0 and 1
 - Worst case: $4n + 4 = O(n)$

Runtime Analysis

```
void MatrixInitialization (std::vector<std::vector<int>> &matrix) {  
    int n = matrix.size();  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            matrix[i][j] = 1;  
        }  
    }  
}
```

$$T(n) = 1 + n(2 + n(2 + 1)) = 1 + n(2 + 3n) = 3n^2 + 2n + 1 = O(n^2)$$

$$\sum_{1}^n \sum_{1}^n 1 = \sum_{1}^n n = n \cdot n = n^2$$

Runtime Analysis

```
oid Solution::MatrixInitialization(std::vector<std::vector<int>> &matrix1,
                                    std::vector<std::vector<int>> &matrix2) {
    int n = matrix1.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix1[i][j] = 1;
        }
    }
    n = matrix2.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix2[i][j] = 1;
        }
    }
}
```

$T(n) = O(n^2) + O(n^2) = O(n^2)$

Runtime Analysis

- **Careful about nested loops**
 - We can't always look at the number of nested loops and raise n to that power!
- Carefully count the operations
 - Outer loop increments by 1 each time
 - Inner loop updates by dividing x in half each iteration
 - After 1st iteration => $x=n/2$
 - After 2nd iteration => $x=n/4$
 - After 3rd iteration => $x=n/8$
 - After k th iteration is last => $x = n/2^k = 1$. Solve for k: $k = \log_2(n)$ iterations
 - $O(n \cdot \log(n))$

```
void DoubleLoops(int n) {  
    for (int i = 0; i < n; i++) {  
        int y = 0;  
        for (int x = n; x > 1; x = x / 2) {  
            y++;  
        }  
        cout << y << endl;  
    }  
    return 0;  
}
```

Some Base Sums

- Arithmetic Series

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

- Geometric Series

$$\sum_{i=0}^n c^i = \frac{c^{n+1}-1}{c-1} = \Theta(c^n)$$

- Harmonic Series

$$\sum_{i=1}^n 1/i = \Theta(\log n)$$

Runtime Analysis

- What is the runtime of this function?

```
int main() {  
    for (int i = 0; i < n; i++) {  
        a[i] = 0;  
        for (int j = 0; j < i; j++) {  
            a[i] += j;  
        }  
    }  
    return 0;  
}
```

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = ?$$

What about recursion?

- What is the runtime of this function?

$$T(n) = 1 + T(n-1)$$

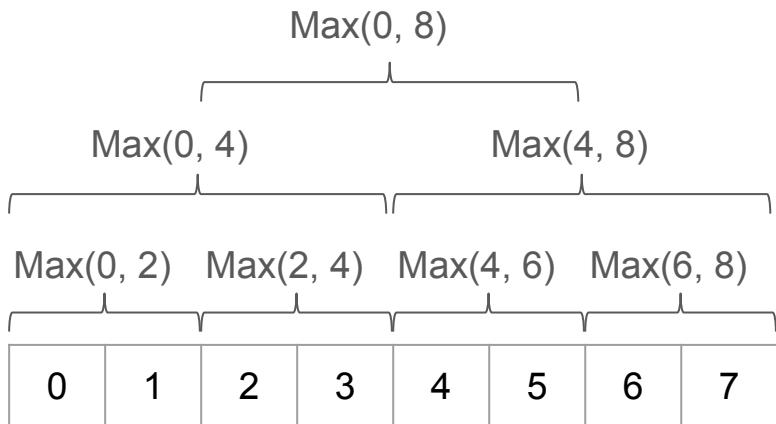
$$= 1 + 1 + T(n-2)$$

$$= 1 + 1 + 1 + T(n-3)$$

$$= O(n)$$

```
void print(Item *head) {  
    if (head == NULL)  
        return;  
    else {  
        std::cout << head->val <<  
        std::endl;  
        print(head->next);  
    }  
}
```

Recursive FindMax



```
int Solution::FindMax(std::vector<int> &inputs,
                      int left, int right) {
    if (right == left + 1) {
        return inputs[left];
    }
    int mid = (right + left) / 2;
    return std::max(
        FindMaxRecursiveAux(inputs, left, mid),
        FindMaxRecursiveAux(inputs, mid, right)
    );
}
```

- We visit every element only once, so $T(n)$ should be $O(n)$
- $T(n) = 2T(n/2) + \Theta(1) = O(n)$
 - We don't provide the proof now

C++ Struct

- Struct is for bundling data
- By default variables are public
- In C++, struct can have methods too (Different than C)
 - However, typically by convention we only use struct for types that don't have any methods

```
struct Person {  
    Person() { name = "UNKNOWN"; }  
    std::string name;  
    int age;  
};  
  
Person Solution::MakeDefaultPerson() {  
    Person p;  
    p.name = "Tommy";  
    p.age = 10;  
    return p;  
}
```

C++ Class

- **Encapsulation**

- Place data and operations in single place
- Keep state hidden/separate from users via private data, public member functions

- **Abstraction**

- Depend only on an interface!
- Ex. a microwave...Do you know how it works? But can you use it?
- Hide implementation details to create low degree of coupling between different components

- **Inheritance**

- “Is A” relationship

```
class Person {  
  
public:  
    Person () { _name = "UNKNOWN"; }  
  
    std::string GetSSN () {  
        return "***-**-*****";  
    }  
  
    void SetSSN (const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Main Parts of a Class

- Member variables
 - What data must be stored?
- Constructor(s)
 - How do you build an instance?
- Member functions
 - How does the user need to interact with the stored data?
- Destructor
 - How do you clean up after an instance?

```
class Person {  
  
public:  
  
    Person () { _name = "UNKNOWN"; }  
  
    std::string GetSSN () {  
        return "***-**-*****";  
    }  
  
    void SetSSN (const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Main Parts of a Class

- **Public or private**
 - Defaults is private (only class methods can access)
 - Must explicitly declare something public
- Most common C++ **operators** will not work by default
 - (e.g. ==, +, <<, >>, etc.)
- May be used just like other data types
 - Get pointers/references to them
 - Pass them to functions (by copy, reference or pointer)
 - Dynamically allocate them
 - Return them from functions

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "***-**-****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

EE599: Computing and Software for Systems Engineers

Lecture 2: A Tour of the C++ Language

University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Hello World

```
#include <iostream>

// My first C++ program
/* Prints "Hello World"*/
int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```



Functions

Return type

Name

Argument List

```
#include <iostream>

// Prints a string and adds a new line at the end.

int PrintLine(std::string text) {
    std::cout << text << std::endl;

    return 0;
}

int main() {
    std::string text = "Hello world!";
    PrintLine(text);
    return 0;
}
```

Functions

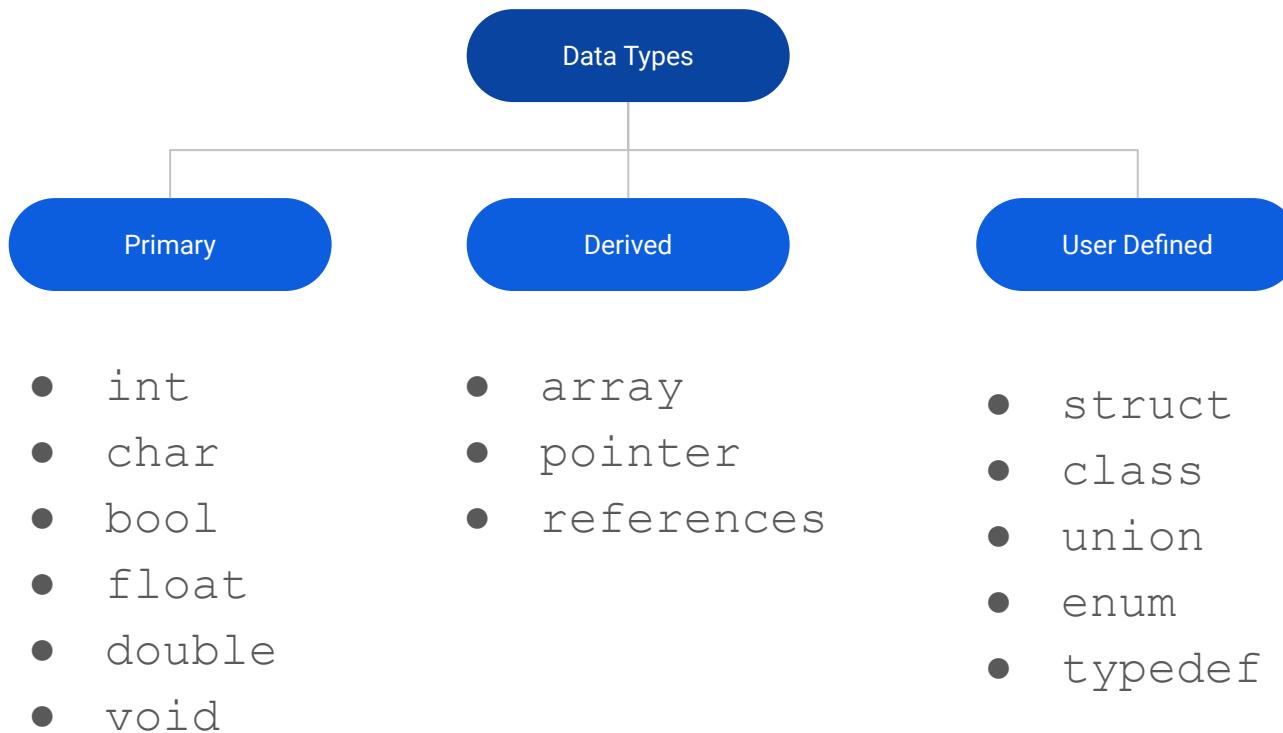
- Can be overloaded
 - Same name
 - Different parameters

```
void PrintLine(int input) { std::cout << input << std::endl; }  
void PrintLine(char input) { std::cout << input << std::endl; }  
void PrintLine(float input) { std::cout << input << std::endl; }  
void PrintLine(double input) { std::cout << input << std::endl; }
```

Operators

Operator	Type
<code>++, --</code>	Unary operator. E.g. <code>i++</code>
<code>+, -, *, /, %</code>	Arithmetic operators. E.g. <code>a = b + c</code>
<code><, <=, >, >=, ==, !=</code>	Relational operators. E.g. <code>a == b</code>
<code>&&, , !</code>	Logical operators E.g. <code>if (a == b && c == d)</code>
<code>&, , <<, >>, ~, ^</code>	Bitwise operators E.g. <code>a << 2, C = a & b</code>
<code>=, +=, *=, /=, %=</code>	Assignment operators. E.g. <code>a += 2;</code>
<code>?:</code>	Ternary conditional operator. E.g. <code>c == d ? 1 : 2;</code>

Data Types



Variables

```
int main() {  
  
    std::string hello = "Hello";  
    std::string world = " world";  
    int year = 2020;  
  
    std::string hello_world = hello + world +  
        " " + std::to_string(year);  
  
    PrintLine(hello_world);  
    return 0;  
}
```

Variables and Modifiers

- signed
- unsigned
- long
- short

Type specifier	Equivalent type	Width in C++ standard	
short	short int	at least 16	
short int			
signed short			
signed short int			
unsigned short			
unsigned short int			
int			
signed			
signed int			
unsigned			
unsigned int	unsigned int	at least 16	
long			
long int			
signed long			
signed long int			
unsigned long	long int	at least 32	
unsigned long int			
long long			
long long int	long long int (C++11)		
signed long long			
signed long long int			
unsigned long long			
unsigned long long int	unsigned long long int (C++11)	at least 64	
unsigned long long int			

Machine Independent

- Fixed width integer types
 - Defined in header <cstdint>
 - `int8_t`
 - `int16_t`
 - `int32_t`
 - `Int64_t`
 - `uint8_t`
 - `uint16_t`
 - `uint32_t`
 - `Uint64_t`
- Some defined constants
 - Defined in header <cstdint>
 - `INT8_MIN` and `INT8_MAX`
 - `INT16_MIN` and `INT16_MAX`
 - `INT32_MIN` and `INT32_MAX`
 - `INT64_MIN` and `INT64_MAX`
 - `UINT8_MAX`
 - `UINT16_MAX`
 - `UINT32_MAX`
 - `UINT64_MAX`

Float and Double

- Two primary types
 - **float** (4 byte)
 - **double** (8 byte)
 - **long double** (larger)
- No signed and unsigned variants.
 - They can represent negative numbers by default.

- Size

- **float**: is a **32 bit** IEEE 754 single precision Floating Point Number
 - 1 bit for the sign, (8 bits for the exponent, and 23 for the value)
 - float has **7 decimal digits** of precision.
- **double**: is a **64 bit** IEEE 754 double precision Floating Point Number:
 - (1 bit for the sign, 11 bits for the exponent, and 52* bits for the value)
 - double has **15 decimal digits** of precision.

Enum Class

- Scoped enumeration
 - Strongly typed
 - Strongly scoped
- Use enum class instead of type!

```
// Enum type in C:  
  
enum ColorPallet1 { Red, Green, Blue };  
  
enum ColorPallet2 { Yellow, Orange, Red };  
  
  
// Enum Class in C++  
  
// Declaration  
  
enum class ColorPalletClass1 { Red, Green, Blue };  
  
enum class ColorPalletClass2 { Yellow, Orange, Red };  
  
  
// Assignment  
  
ColorPalletClass1 col1 = ColorPalletClass1::Red;  
  
ColorPalletClass2 col2 = ColorPalletClass2::Red;
```

Assignment

- Effectively copying

```
int i = 10;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

```
int j = i;
```

Address	Value
0x7000	55
0x8004	10
0x8008	45

Pointers

- Hold address of a variable

```
int *p = &i;
```

Can act as an alias to variables

What is the result of:

- `(*p)++;`
- `p++;`

```
int i = 10;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

References

- Hold address of a variable

```
int &j = i;
```

An alias for a variable

What is the result of:

- `j++;`

```
int i = 10;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

Passing Parameters

```
// Acts like int j = i  
void PassByValue (int j) { j++; }  
  
// Acts like int *j = &i;  
void PassByReferenceUsingPointer (int *j) { (*j)++; }  
  
// Acts like int &j = i;  
void PassByReferenceUsingReference (int &j) { j++; }
```

Why Using References (or Pointers)

- When is pass-by-value useful?
- Pass by Reference
 - Modify a variable in a function
 - Return multiple variables
 - Avoiding copying
 - Loops
 - Modification
 - Avoiding modification

```
// Acts like int j = i
void PassByValue (int j) {
    j++;
}

// Acts like int *j = &i;
void PassByReferenceUsingPointer (int *j) {
    (*j)++;
}

// Acts like int &j = i;
void PassByReferenceUsingReference (int
&j) {
    j++;
}
```

References in Loops

```
int main() {  
    std::vector<int> my_vector = { 1, 2, 3, 4, 5, 6, 7, 8 };  
    for (auto n : my_vector) {  
        n++;  
    }  
    for (auto &n : my_vector) {  
        n *= 10;  
    }  
    return 0;  
}
```

Pointers: new and delete

- Pointers can use **dynamic memory** AKA **heap**
 - **new**
 - **delete**
- Depending on the inputs, the memory footprint changes

```
int main () {  
  
    int *p = NULL;  
  
    if (4 > 2) {  
  
        p = new int;  
  
        *p = 5;  
  
        std::cout << "*p: " <<  
            *p << std::endl;  
    }  
  
    if (p != NULL) {  
  
        delete p;  
  
    }  
}
```

References v.s Pointers

- References:
 - Cannot be **reassigned**.
 - Must be **initialized** once defined.
 - Cannot be **NULL**.
 - References can become invalid
 - Less common
- Pointers:
 - Need to be **dereferenced**.
 - * or ->
 - Limited **arithmetic** operations
 - Pointers use **new** and **delete** to store values in dynamic memory
 - Are generally less **safe**

```
Person person;  
  
Person *person_ptr = &person;  
  
(*person_ptr).first_name = "Tommy";  
  
person_ptr->last_name = "Trojan";
```



- Pointers more likely to be misused, and they can be very dangerous.
- References can be misused too, but less likely

Arrays

```
int arr[8]
```



- Features
 - Collection of items
 - Contiguous memory locations
 - Can be indexed
- What you need to know:
 - Index starts from 0
 - Array size is predefined
 - Unless it is a dynamic array
 - It is really a pointer
 - It can be dangerous!
 - It is passed by reference

```
int main () {  
    int arr[8];  
    arr[0] = 5;  
    arr[2] = -10;  
    print_array(arr, 8);  
  
    int x = arr[0];  
    std::cout << "x: " << x << std::endl;  
  
    return 0;  
}
```

Dynamic Arrays

```
arr = new int[size];  
delete [] arr;
```

- We use `new` and `delete`
 - The size of dynamic array doesn't have to be known at compile time
 - But it still cannot be resized!

Why Vectors?

- Std::vector is a user defined type provided by C++ STL.

Data type	Feature
Array	<ul style="list-style-type: none">Size is fixed at compile timeCannot be resizedThere is not a reliable way to find the array size<ul style="list-style-type: none">Homework: how do we find the size of an array?
Dynamic Array	<ul style="list-style-type: none">Size doesn't have to be known at compile timeCannot be resizedThere is no way to find the size.
Vectors	<ul style="list-style-type: none">Size doesn't have to be known at compile timeCan be resized automaticallyThe size is always kept updated

std::vectors

- Important methods to know
 - push_back()
 - pop_back()
 - insert()
 - erase()
- Homework:
 - What is the time complexity of each of the above functions?

Const

- Indicates no change
 - Variables
 - Pointers
 - Function arguments and return types
 - Class Data members
 - Class Member functions
 - Objects

```
int main() {  
  
    const int i = 1;  
  
    const int j = i + 1; // Initializing is ok  
    i++;                // Don't change the const!  
  
    return 0;  
}
```

Const in Function Parameters

Use: const references for input parameters to:

1. Avoid copying
2. Avoid modification

You **CAN** pass a **non-const** to **const**

You **CANNOT** pass a **const** to **non-const**

```
int CalculateTax(int income) {  
    income = income - 20; // It changes the copy  
    return income * 0.3;  
}  
  
int CalculateTaxRef(int &income) {  
    income = income - 20; // It changes original!  
    return income * 0.3;  
}  
  
int CalculateTax(const int &income) {  
    income = income - 20; // Don't touch my income!  
    return income * 0.3;  
}
```

Const in Classes

1. Member variables:
 - a. They should **be initialized** by constructor
2. Member Functions:
 - a. They cannot change the member variables
3. Const objects:
 - a. Their member variables cannot change
 - b. Should be initialized by constructor

```
// Const object  
  
const Person q(/*_ssn=*/354545454);  
// q._age = 21; // Error!  
  
// Initializing ok  
const Person r(354545454, 21);
```

const objects

Flow control

- Conditionals
 - if
 - switch
- Loops
 - while
 - do-while
 - for

Flow control

```
while (i < 10) {  
    if (i == 2) {  
        continue;  
    }  
  
    my_vector.push_back(i);  
  
    i++;  
}
```



```
bool b = false;  
  
if (b = true) {  
  
    std::cout << "b is true" << std::endl;  
  
}
```

- Beware of corner case conditions
- Beware of infinite loops

std::string

- Important methods to know
 - push_back()
 - pop_back()
 - getline()
 - concat
 - insert()
- Homework:
 - What is the time complexity of each of the above functions?

Main Parts of a Class

- Member variables
 - What data must be stored?
- Constructor(s)
 - How do you build an instance?
- Member functions
 - How does the user need to interact with the stored data?
- Destructor
 - How do you clean up after an instance?

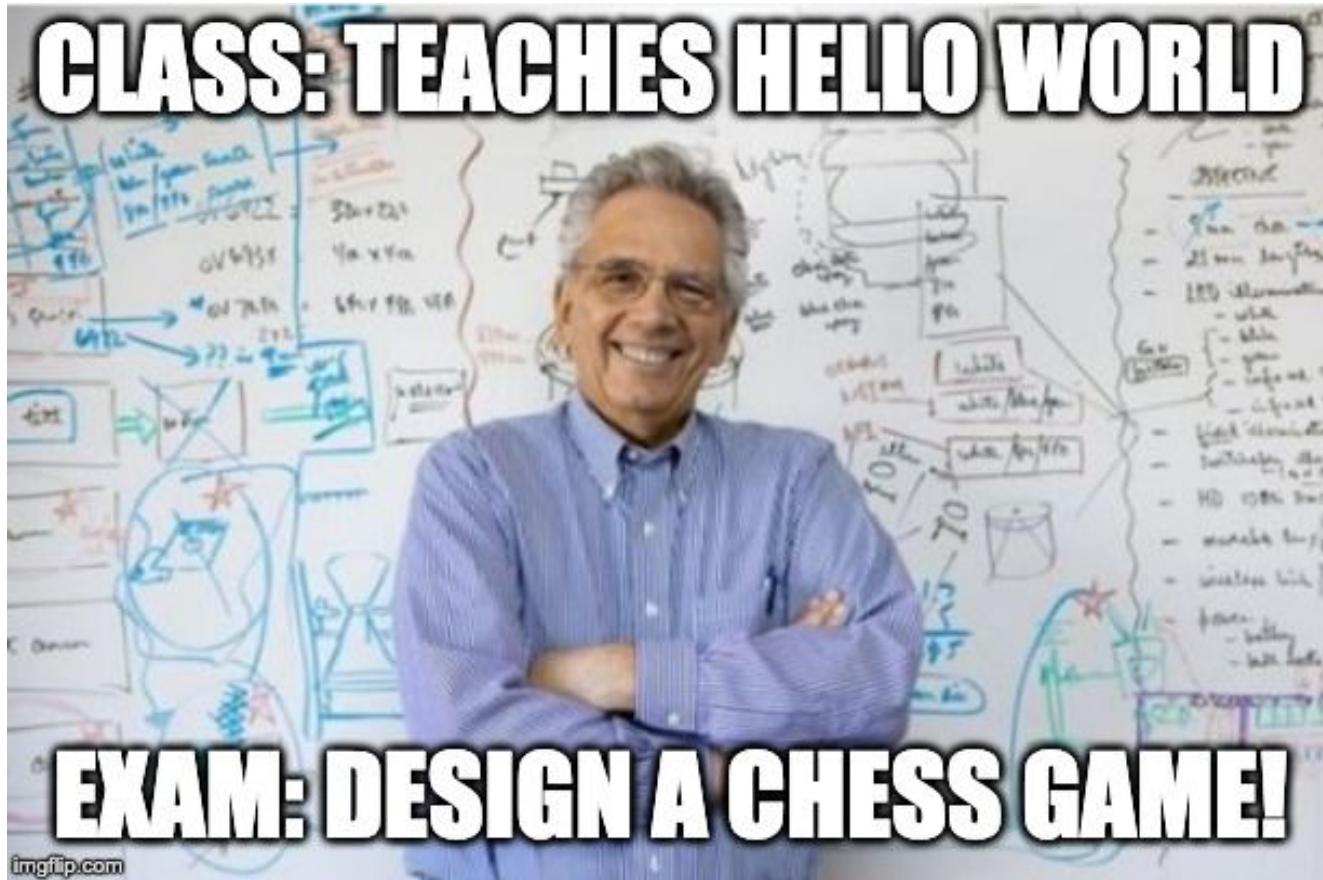
```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN(){  
        return "***-**-*****";  
    }  
  
    void SetSSN(const std::string& ssn){  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Main Parts of a Class

- **Public or private**
 - Defaults is private (only class methods can access)
 - Must explicitly declare something public
- Most common C++ **operators** will not work by default
 - (e.g. ==, +, <<, >>, etc.)
- May be used just like other data types
 - Get pointers/references to them
 - Pass them to functions (by copy, reference or pointer)
 - Dynamically allocate them
 - Return them from functions

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "***-**-****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

CLASS: TEACHES HELLO WORLD



EXAM: DESIGN A CHESS GAME!

imgflip.com

EE599: Computing and Software for Systems Engineers

Lecture 3: A Tour of the C++ Language Part 2

University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Namespaces

```
namespace ns1 {  
    int x = 1;  
  
    void Print() { std::cout << "Printing example 1." << std::endl; }  
}  
// namespace ns1  
  
namespace ns2 {  
    int x = 2;  
  
    void Print() { std::cout << "Printing example 2." << std::endl; }  
}  
// namespace ns2  
  
  
int main() {  
    std::cout << "ns1::x: " << ns1::x << std::endl;  
    std::cout << "ns2::x: " << ns2::x << std::endl;  
    ns1::Print();  
    ns2::Print();  
}
```

Variable Scope

- Variable scope
 - Global
 - Local
 - i. Inside functions
 - ii. Inside blocks (curly braces)

Arrays

```
int arr[8]
```

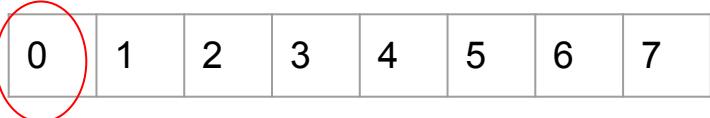


- Features
 - Collection of items
 - Contiguous memory locations
 - Can be indexed
- What you need to know:
 - Index starts from 0!
 - Array size is predefined
 - Unless it is a dynamic array

```
int main() {  
    int arr[8];  
    arr[0] = 5;  
    arr[2] = -10;  
    print_array(arr, 8);  
  
    int x = arr[0];  
    std::cout << "x: " << x << std::endl;  
  
    return 0;  
}
```

Arrays

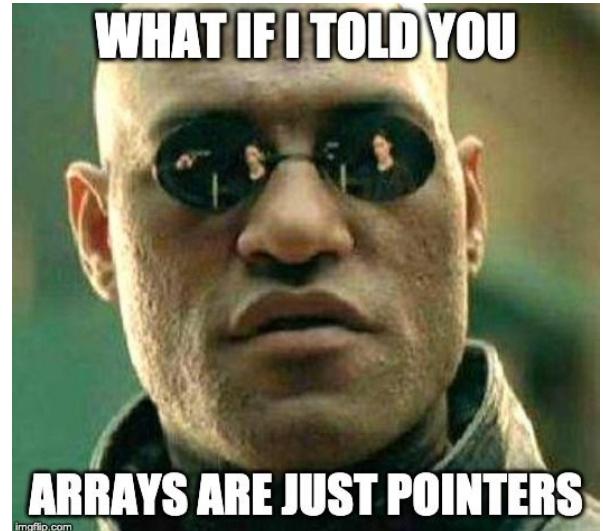
```
int arr[8]
```



```
53 int my_arr[8];
54 InitializeArray(my_arr, 8);
55 PrintArray(my_arr, 8);
56 int *p = &my_arr[0];
57 std::cout << "arr: " << my_arr << std::endl;
58 std::cout << "p: " << p << std::endl;
59 std::cout << "*(p): " << *(p) << std::endl;
60 std::cout << "*(p+1): " << *(p + 1) << std::endl;
61 std::cout << "*(p+2): " << *(p + 2) << std::endl;
62
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

```
{ 0, 1, 2, 3, 4, 5, 6, 7 }
arr: 0x7ffee529a6d0
p: 0x7ffee529a6d0
*(p): 0
*(p+1): 1
*(p+2): 2
```



Dynamic Arrays

```
arr = new int[size];  
  
delete [] arr;
```

- We use `new` and `delete`
 - The size of dynamic array doesn't have to be known at compile time
 - But it still cannot be resized!

Array Misuse

- Index out of bound
 - Undefined behavior
 - Always check to see if there is any chance you index array out of bound

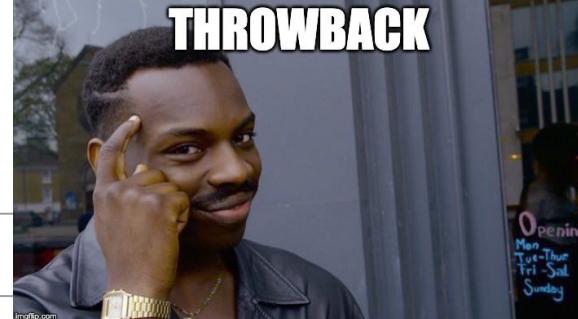
```
int arr[10];  
  
for (int i = 0; i <= 10; i++) {  
  
    arr[i] = 0;  
  
}
```



imgflip.com

Why Vectors?

Data type	Feature
Array	<ul style="list-style-type: none"> ● Size is fixed at compile time ● Cannot be resized ● There is not a reliable way to find the array size <ul style="list-style-type: none"> ○ Homework: how do we find the size of an array? ● Can be misused (out of bound)
Dynamic Array	<ul style="list-style-type: none"> ● Size doesn't have to be known at compile time ● Cannot be resized ● There is no way to find the size. ● Can be misused (out of bound)
Vectors	<ul style="list-style-type: none"> ● Size doesn't have to be known at compile time ● Can be resized automatically ● The size is always kept updated ● Can be misused (out of bound) with [] <ul style="list-style-type: none"> ○ Misuse can be controlled using at() method (homework)



Const

- Indicates no change
 - Variables
 - Pointers
 - Function arguments and return types
 - Class Data members
 - Class Member functions
 - Objects

```
int main() {  
  
    const int i = 1;  
  
    const int j = i + 1; // Initializing is ok  
  
    i++; // Don't change the const!  
  
}
```

Const in Function Parameters

Use: const references for input parameters to:

1. Avoid copying
2. Avoid modification

You **CAN** pass a **non-const** to **const**

You **CANNOT** pass a **const** to **non-const**

```
int CalculateTax(int income) {  
    income = income - 20; // It changes the copy  
    return income * 0.3;  
}  
  
int CalculateTaxRef(int &income) {  
    income = income - 20; // It changes original!  
    return income * 0.3;  
}  
  
int CalculateTax(const int &income) {  
    income = income - 20; // Don't touch my income!  
    return income * 0.3;  
}
```

Const in Function Parameters

Data type	Feature
Pass by value	<ul style="list-style-type: none">• Copying, so original is protected• But, copying can have high cost
Pass by reference	<ul style="list-style-type: none">• No copying<ul style="list-style-type: none">◦ Original might be changed (can be good or bad)• No copy overhead
Pass by const reference	<ul style="list-style-type: none">• No copying• Original cannot be changed

Const in Classes

1. Member variables:
 - a. They should **be initialized** by constructor
2. Member Functions:
 - a. They cannot change the member variables
3. Const objects:
 - a. Their member variables cannot change
 - b. Should be initialized by constructor

```
// Const object  
  
const Person q(/*_ssn=*/354545454);  
// q._age = 21; // Error!  
  
// Initializing ok  
const Person r(354545454, 21);
```

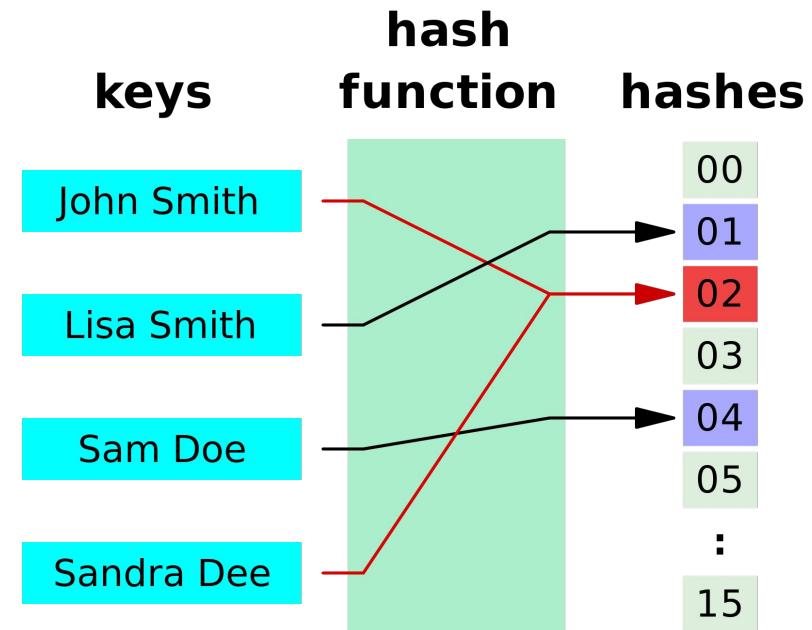
const objects

std::string

- TLDR:
 - Strings are optimized to only contain character primitives
 - Vectors can contain primitives or objects
- Important methods to know
 - push_back()
 - pop_back()
 - getline()
 - concat
 - insert()
 - c_str()
- Homework:
 - What is the time complexity of each of the above functions?

Hash Function

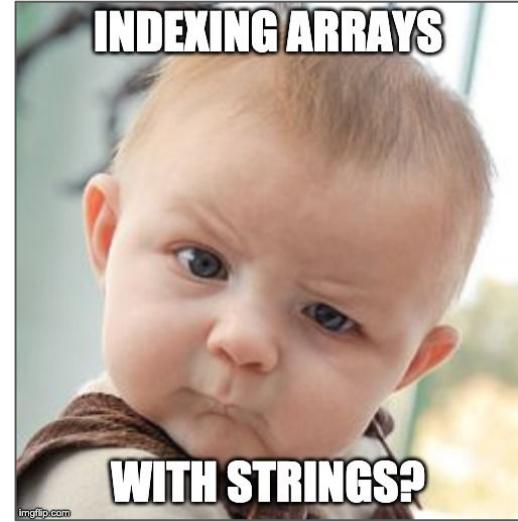
- Suppose we want to:
 - hold information of **Persons** in a vector
 - index it with their **names** rather than numbers



Hash function: $f(\text{keys}) \rightarrow \text{indices}$

std::map

- Store elements in a mapped fashion
 - AKA **Hash map** OR **Associated Array**
- Important methods to know
 - size()
 - insert()
 - count()
- Things to know about std::map:
 - Internally it is **sorted** based on keys
 - **Access, Insert, and find** complexity is **O(log(n))**
 - Map is really a collection of pairs
 - Accessing a non-existent key using [], creates that key
 - No duplicate keys
- std::pair<T1, T2>
 - Conceptually, it's a map of size 1
 - Couples together a pair of things

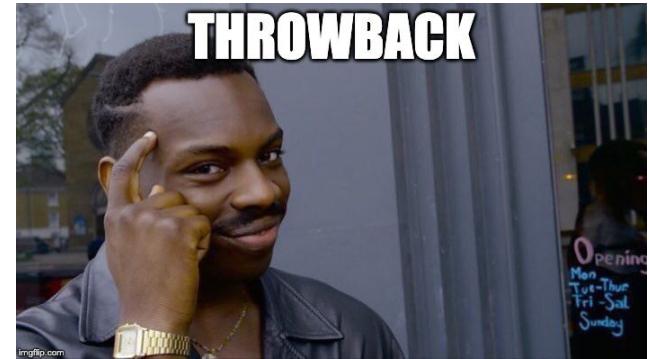


std::set

- Store a set of elements
 - AKA **Hash Set**
- Important methods to know
 - size()
 - insert()
 - count()
 - find()
- Things to know about std::set:
 - Internally it is **sorted** based on keys
 - Access, Insert, and find complexity is
O(log(n))
 - Reinserting the same key will just update the data, there is no duplicate keys

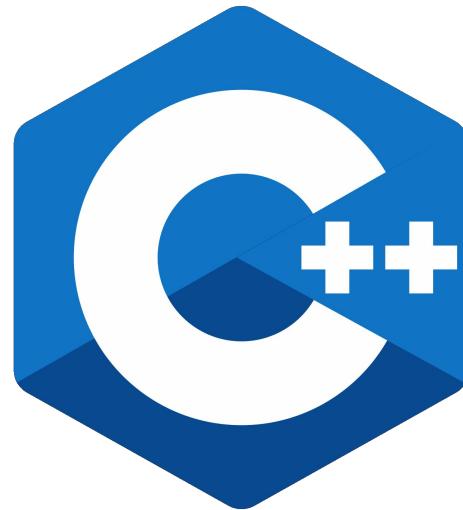
Example: Find union of two vectors

- Throwback to Lecture 1:
 - **Step 1:** Clearly define the input and output of the problem
 - **Step 2:** Find some example input/outputs
 - **Step 3:** Propose an algorithm
 - **Step 4:** Test its correctness
 - **Step 5:** Prove its correctness
 - **Step 6:** Find runtime
- Homework:
 - Find union of two vectors



STL

- A set of C++ template classes and functions
- Four components
 - Algorithms
 - Containers
 - Functions
 - Iterators
- So far we have seen:
 - Vectors, sets, strings, and maps



Iterators

- Used for iteration of STL objects
 - An internal variable (similar to a **pointer**) of the class that moves one step in the collection as we iterate

```
std::vector<int>::iterator it;
```

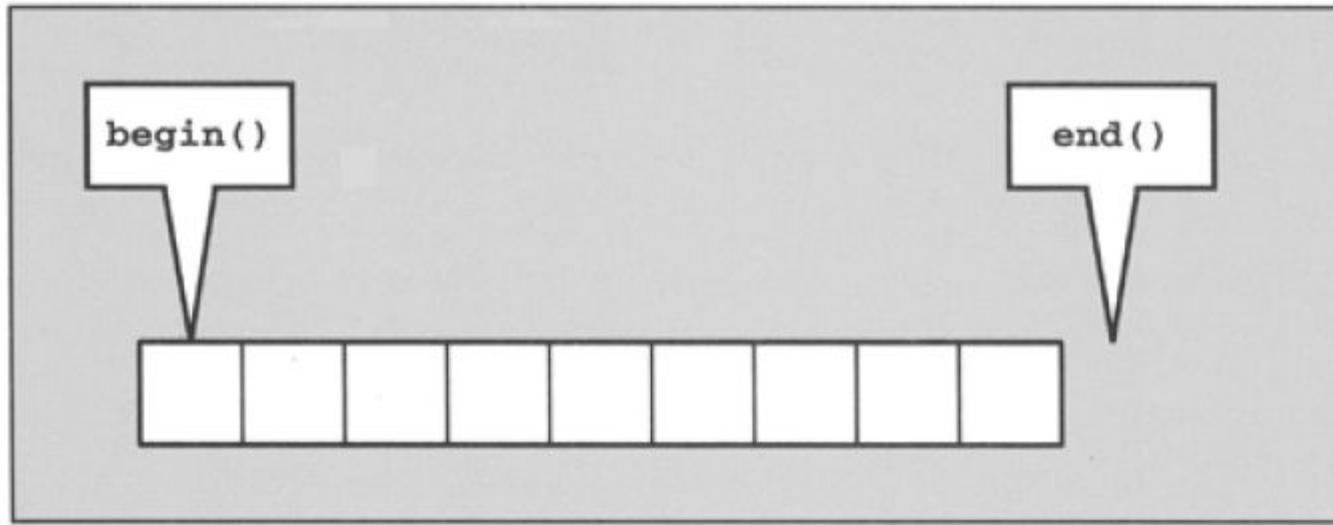
Definition

```
int n = *it;
```

Dereferencing

```
std::vector<int> v = {1, 2, 3, 4, 5};  
// An easy way of iteration  
  
for (int n : v) {  
  
    std::cout << "n: " << n << std::endl;  
}  
  
// General way of iteration  
  
std::vector<int>::iterator it;  
  
for (it = v.begin(); it != v.end(); ++it) {  
  
    int n = *it;  
  
    std::cout << "n: " << n << std::endl;  
}
```

Iterators



What do we need to know about iterators?

- Think of an iterator as a pointer
 - Initially it points to nothing
 - **begin()** : address of the **first** item
 - **end()**: address **AFTER the last item** or **NULL**
 - As long as the iterator is less than or not equal end(), you are safe
 - You can perform ++ and -- on them
 - Each time check against **begin()** and **end()**

```
// General way of iteration
std::vector<int>::iterator it;

for (it = v.begin(); it != v.end(); ++it) {
    int n = *it;
    std::cout << "n: " << n << std::endl;
}
```

Definition, initialization, check for end(), increment, and dereferencing

Why Iterators?

- Iterating the container
 - Duh!
- Reuse code
- Container manipulation

```
std::vector<int> v = {1, 2, 3, 4, 5};

std::vector<int>::iterator v_it;

for (it = v.begin(); it != v.end(); ++it) {

    int n = *it;

    std::cout << "n: " << n << std::endl;

}

std::set<int> s = {1, 2, 3, 4, 5};

std::set<int>::iterator s_it;

for (s_it = s.begin(); s_it != s.end(); ++s_it) {

    int n = *s_it;

    std::cout << "n: " << n << std::endl;

}
```

A More Modern Way of Using Iterators

- Using auto
 - The compiler deduces the type for us
 - Use a const reference if you don't want to modify the items and prevent copy

```
// using auto

std::set<int> s = {1, 2, 3, 4, 5};

for (auto it = s.begin(); it != s.end(); ++it) {

    const int &n = *it;

    std::cout << "n: " << n << std::endl;
}
```

Insert and Delete

- Iterators can tell us where to insert or delete
 - Irrespective of the type of the container

```
// Insert a number before 3

std::vector<int> v = {1, 2, 3, 4, 5};

for (auto it = v.begin(); it != v.end(); ++it) {

    const int &n = *it;

    if (n == 3) {

        v.insert(it, 12);

        break;

    }

}
```

STL Algorithms

- Iterators are used to generalize them
 - Sort
 - Find
 - Reverse
 - Max_element
 - Min_element
 - Accumulate
 - Count
 - Count_if

```
std::vector<int> v = {12, -2, 0, 13, 3, 5};

auto it = std::find(v.begin(), v.end(), 4);

if (it != v.end()) {
    const auto &n = *it;
    std::cout << "Found n: " << n << std::endl;
} else {
    std::cout << "Didn't find." << std::endl;
}
```

Main Parts of a Class

- Member variables
 - What data must be stored?
- Constructor(s)
 - How do you build an instance?
- Member functions
 - How does the user need to interact with the stored data?
- Destructor
 - How do you clean up after an instance?

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "*****-*****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Main Parts of a Class

- **Public or private**
 - Defaults is private (only class methods can access)
 - Must explicitly declare something public
- Most common C++ **operators** will not work by default
 - (e.g. ==, +, <<, >>, etc.)
- May be used just like other data types
 - Get pointers/references to them
 - Pass them to functions (by copy, reference or pointer)
 - Dynamically allocate them
 - Return them from functions

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "***-**-****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

EE599: Computing and Software for Systems Engineers

Lecture 4: A Tour of the C++ Language

University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

std::map vs std::unordered_map

- AKA **Associative Arrays** or **Dictionary**
- Used for mapping **unique Keys** to **Values**.
 - Example: Mapping **SSN** to Person
 - Example: Count the number of each word in a book: Map of words to numbers.
- Both provide similar APIs

std::map	std::unordered_map
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, removal, and insertion operations have logarithmic complexity: O(log n)	Search, insertion, and removal of elements have average time of O(1) , but worst case can be O(n)

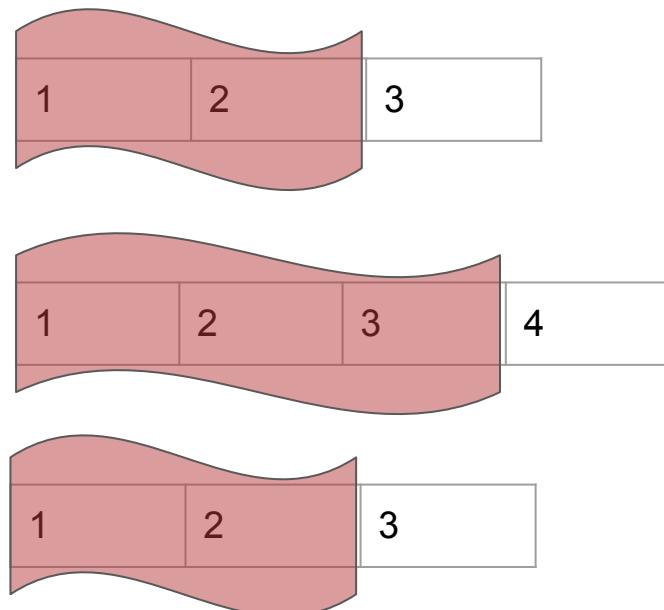
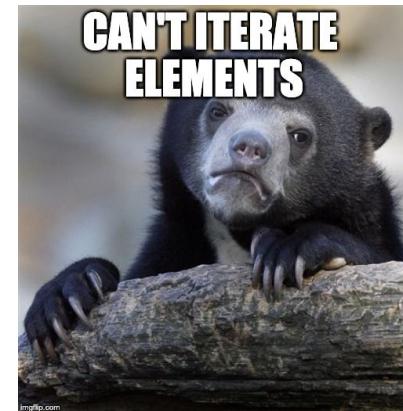
std::set vs std::unordered_set

- Used for keeping a list of **unique** items
 - A set is really the list of keys in a map
- Both provide similar APIs

std::map	std::unordered_map
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, removal, and insertion operations have logarithmic complexity: O(log n)	Search, insertion, and removal of elements have average time of O(log 1) , but worst case can be O(n)

std::stack

- Conceptually, stack is like a vector, but we can always only access its last element.
 - LIFO ordering
 - We can't iterate all of its elements.
 - That means there is no **begin** and **end**!
- It provides these methods:
 - `empty()`
 - `size()`
 - `top()`
 - `push()`
 - `pop()`
- Homework:
 - Find time complexity of the above functions
 - Write a function to print the stack



std::stack

- What should you be worried about?
 - Don't top() or pop() when the stack is empty
 - Don't push when the stack is full

```
std::stack<int> s;

int r = s.top();    // Seriously?
s.pop();           // Don't do this either!

// Do this instead

if(!s.empty()){

    s.pop();

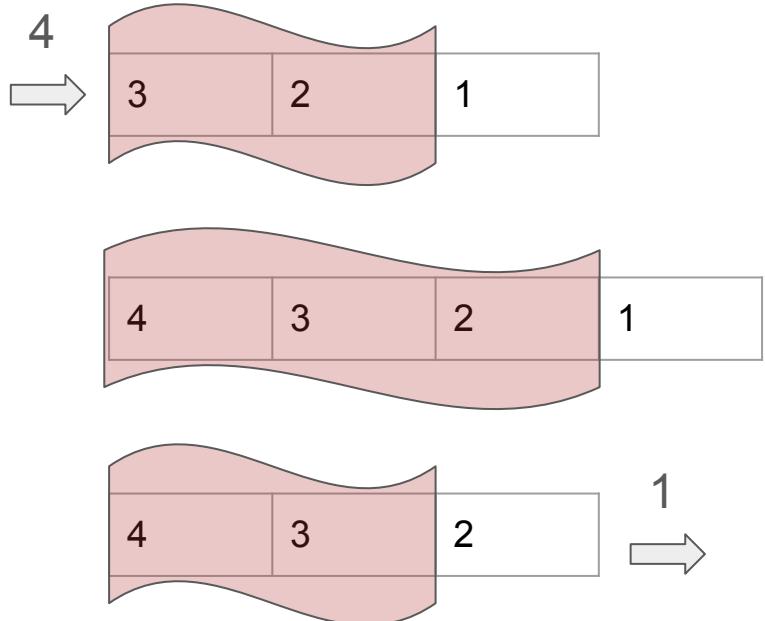
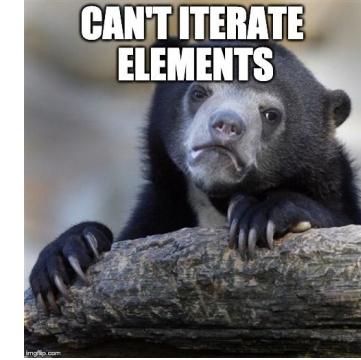
}
```



CAREFUL WITH PUSH, POP,
AND TOP

std::queue

- Conceptually, queue is like a vector, but we can always only access its **first(front)** and **last(rear)** elements.
 - We can't iterate all of its elements.
 - That means there is no **begin** and **end!**
- It provides these methods:
 - `empty()`
 - `size()`
 - `front():` read front
 - `push():` push into rear
 - `pop():` pop from front
- Homework:
 - Write a function to print a queue



std::list

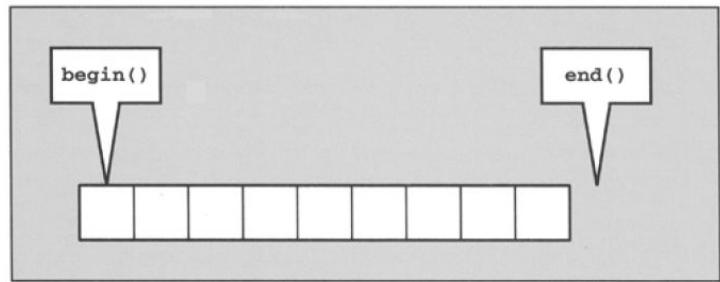
- Like vector, list is a collection of items, but:
 - No contiguous memory locations, therefore
 - No indexing operator
 - Slow indexing
 - Fast insert/delete (after indexing)
- It **CAN** be iterated
- It provides these methods:
 - empty()
 - size()
 - insert(), erase()
 - front(), back()
 - push_front(), pop_front()
 - push_back(), pop_back()



1	2	3	4
---	---	---	---

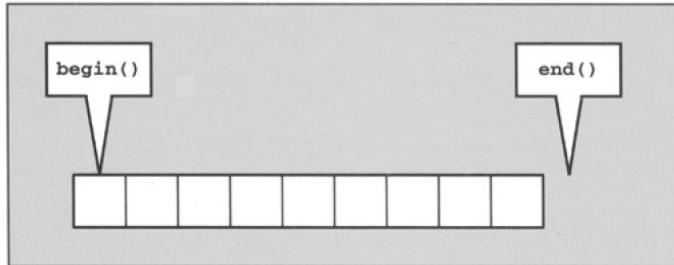
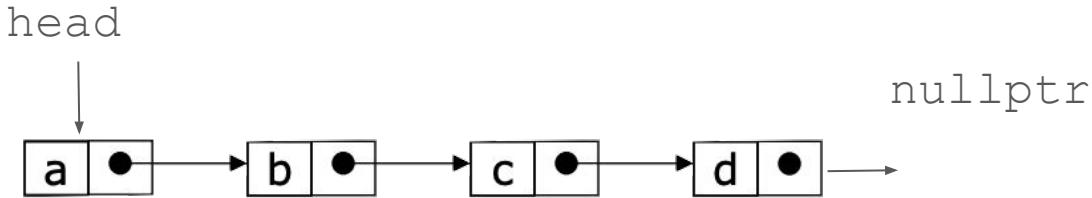
std::list indexing

- Using iterators, we only have access to the **front** and **back** of the list
 - Front: **begin()**
 - One after back: **end()**
- So index *i* would be:
 - $it = begin(); it++ \text{ (i times)}$
- We can't directly add to iterators
 - STL provides these functions:
 - std::advance**
 - std::next**



Linked Lists

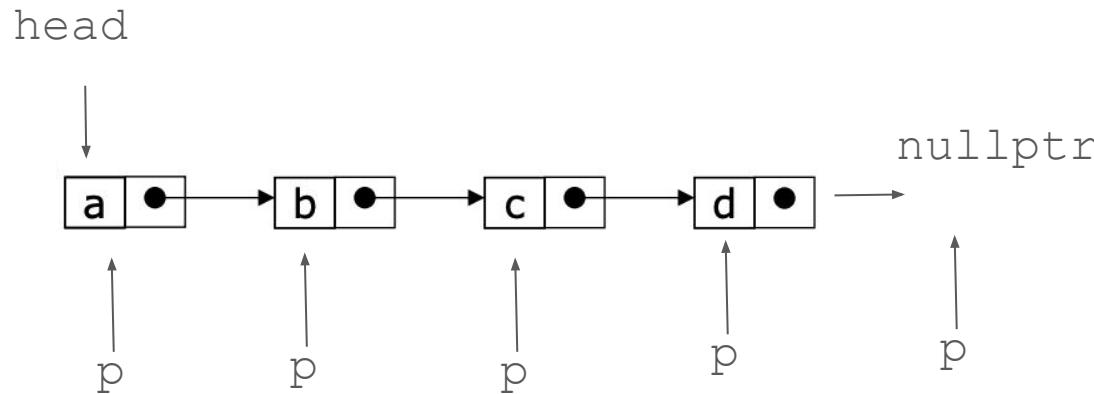
- A sequence of **Nodes**
- Each node has:
 - Value
 - Next pointer
- Each node points to the next one
- Last node points to nothing (**nullptr**)
- First node is in **head**



```
struct ListNode {  
    int val;  
    ListNode *next;  
    ListNode(int x) : val(x), next(nullptr) {}  
};
```

Iterating a List

- Usually, all we have is the **head** pointer
 - How to find the last node?

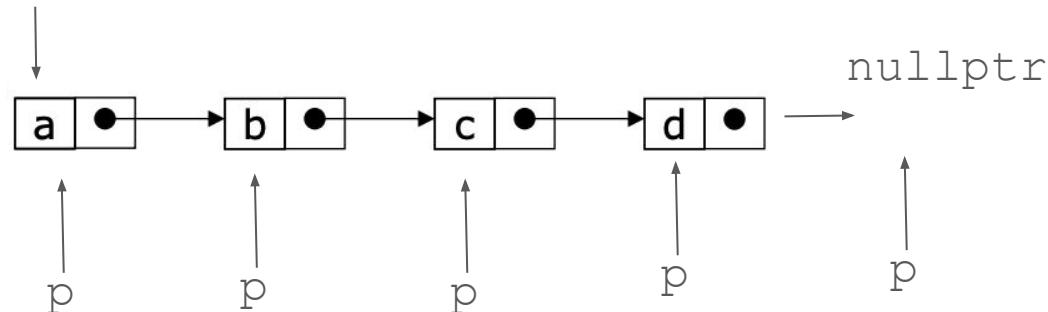


```
while (p != nullptr) {  
    p = p->next;  
}
```

Iterating a List

- Usually, all we have is the **head** pointer
 - How to find the last node?

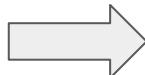
head



```
while (p != nullptr) {  
    p_prev = p;  
    p = p->next;  
}
```

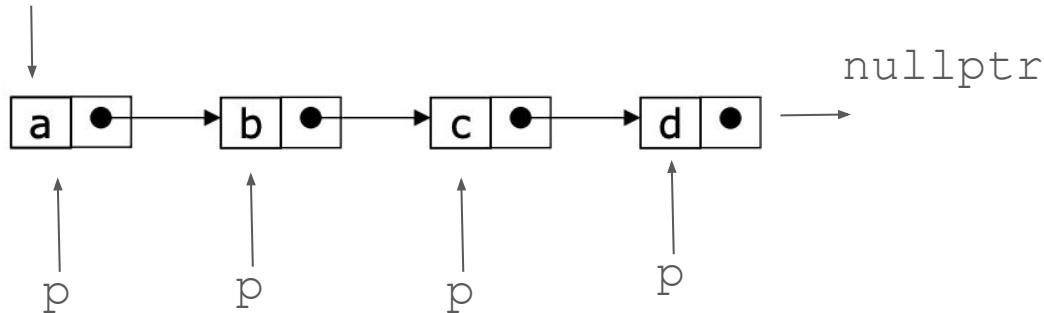
Getting the Tail Pointer (Optimized)

```
while (p != nullptr) {  
    p_prev = p;  
    p = p->next;  
}
```



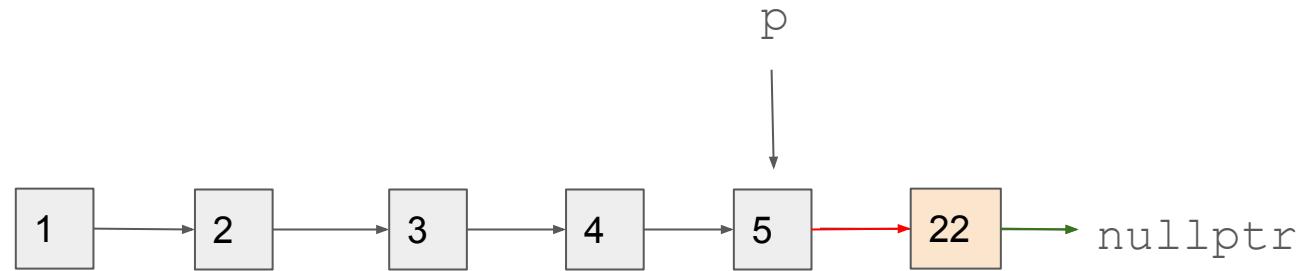
```
while (p->next != nullptr) {  
    p = p->next;  
}
```

head



push_back

push_back(22)



```
p = GetBackPointer();
```

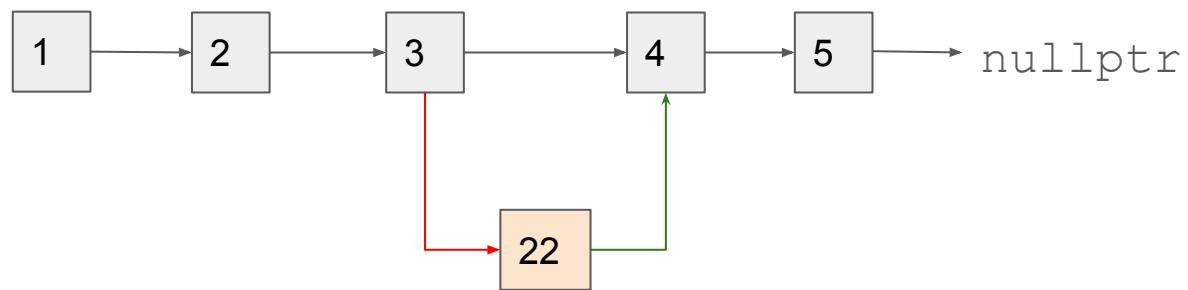
```
ListNode* newNode = new ListNode;
```

```
P -> next = newNode;
```

insert_after

```
insert_after(p, 22)
```

p



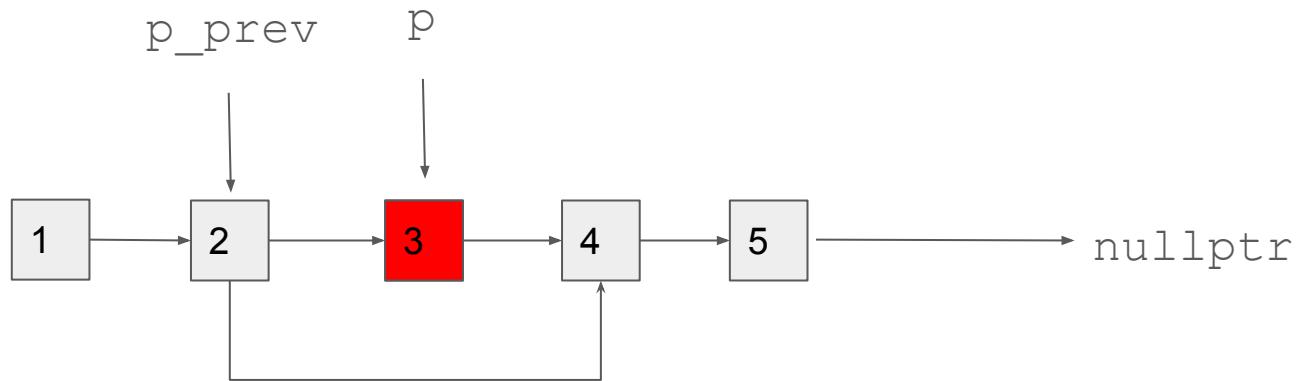
```
ListNode* newNode = new ListNode;
```

```
newNode -> next = p -> next;
```

```
P -> next = newNode;
```

erase

erase (p)

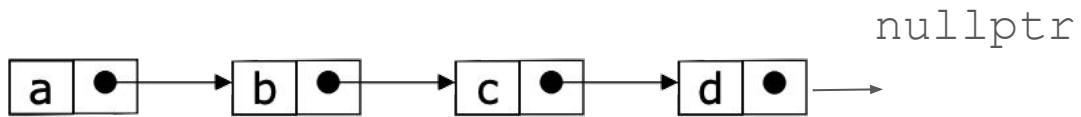


```
p_prev -> next = p->next;
```

```
delete p;
```

There is a famous interview question that requires you to do this without `p_prev`. Homework :)

Activity



```
class SinglyLinkedList {  
public:  
    SinglyLinkedList() { }  
    ~SinglyLinkedList() { }  
    ListNode *head_;  
    bool empty();  
    int size();  
    void push_back(int i);  
    void pop_back();  
    int back();  
    ListNode *GetBackPointer();  
    ListNode *GetIthPointer(int i);  
    void print();  
};
```

STL Algorithms

- Iterators are used to generalize them
 - Sort
 - Find
 - Reverse
 - Max_element
 - Min_element
 - Accumulate
 - Count
 - Count_if

```
std::vector<int> v = {12, -2, 0, 13, 3, 5};

auto it = std::find(v.begin(), v.end(), 4);

if (it != v.end()) {
    const auto &n = *it;
    std::cout << "Found n: " << n << std::endl;
} else {
    std::cout << "Didn't find." << std::endl;
}
```

Main Parts of a Class

- Member variables
 - What data must be stored?
- Constructor(s)
 - How do you build an instance?
- Member functions
 - How does the user need to interact with the stored data?
- Destructor
 - How do you clean up after an instance?

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "*****-*****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Main Parts of a Class

- **Public or private**
 - Defaults is private (only class methods can access)
 - Must explicitly declare something public
- Most common C++ **operators** will not work by default
 - (e.g. ==, +, <<, >>, etc.)
- May be used just like other data types
 - Get pointers/references to them
 - Pass them to functions (by copy, reference or pointer)
 - Dynamically allocate them
 - Return them from functions

```
class Person {  
  
public:  
  
    Person() { _name = "UNKNOWN"; }  
  
    std::string GetSSN() {  
        return "***-**-****";  
    }  
  
    void SetSSN(const std::string& ssn) {  
        _social_security_number = encrypt(ssn);  
    }  
  
private:  
    std::string _name;  
    int _age;  
    std::string _social_security_number;  
};
```

Constructor



`Point()`

`Point(int i, int j)`

`Point(const Point& r)`

`Point p;`

`Point p(1, 3)`

`Point p1 = p2;`

- **No parameterized and copy constructor** will be provided by compiler if we don't define ANY constructor
 - We call them **default constructor** and **default copy constructor**.
 - If you write ANY constructor, the default constructor is not created.
 - Default constructor does not necessarily initialize member variables.

Constructor

Point()

```
class Point {  
  
public:  
  
    Point() {  
  
        i_ = 5;  
  
        j_ = 5;  
  
        std::cout << "NO PARAMETERIZED constructor." << std::endl;  
  
    }  
  
private:  
  
    int i_;  
  
    int j_;  
};
```

Copy Constructor

Point(const Point &p2)

- What you need to know
 - Gets executed when the object is **copied**.
 - Performs **shallow** copy (i.e. non all members on non-dynamic memory)
 - If you have dynamic memory, you should provide **deep copy**
 - STL containers: copy constructor copies all items
 - i. E.g. Copying a vector might have a huge cost

```
Point(const Point &p2) {  
    std::cout << "COPY constructor." << std::endl;  
    i_ = p2.GetI();  
    j_ = p2.GetJ();  
}
```

Destructor

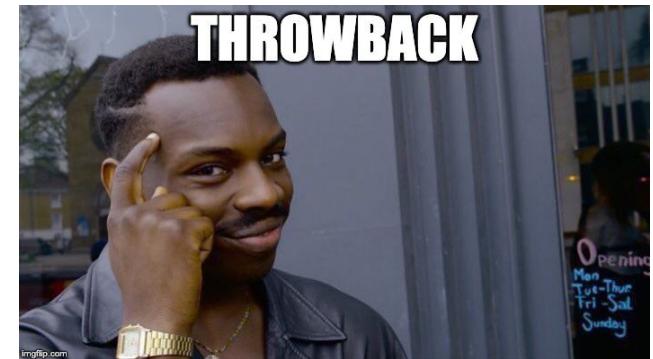
`~Point()`

- What you need to know
 - Gets executed when the object is **destroyed**.
 - If you don't write one, the compiler generates a default one.
 - Very common for dynamic memory allocation
 - i. Use **delete** in destructor

```
~Point() { std::cout << "DESTRUCTOR." << std::endl; }
```

When does a variable get destructed?

- Destruction of an object
 - End of program
 - End of function
 - Variable goes **out of scope**
 - **Delete** operator



Throwback

- When you see a pointer, check for misuse:
 - Is it **deleted** correctly?
 - i. Memory leak
 - Is it **initialized**?
 - i. Can crash
 - Is the pointer value itself **modified**?
 - i. Can crash
- What if the **pointer goes out of scope**?
- What if the variable that the pointer is pointing to **goes out of scope or deleted**?



Operator overloading

- Most operators can be overloaded
 - Examples: +, -, =, ++, ...

```
Point operator+(const Point &rhs) {  
    Point res;  
  
    res.SetI(i_ + rhs.GetI());  
  
    res.SetJ(j_ + rhs.GetJ());  
  
    return res;  
}
```

Overloading pre and postfix unary operators(++, --)

```
// Prefix overload  
  
// ++p;  
  
Point operator++() {  
  
    i_++;  
  
    j_++;  
  
    return *this;  
}
```

```
// Postfix overload  
  
// p++;  
  
Point operator++(int) {  
  
    Point temp = *this;  
  
    i_++;  
  
    j_++;  
  
    return temp;  
}
```



Overloading << and >>

```
std::ostream &operator<<(std::ostream &os, const Point &m) {
    return os << " (" << m.GetI() << ", " << m.GetJ() << " )";
}

std::istream &operator>>(std::istream &is, Point &p) {
    std::cout << "Enter i ";
    is >> p.i_;
    std::cout << "Enter j ";
    is >> p.j_;
    return is;
}
```

EE599: Computing and Software for Systems Engineers

Lecture 5: Functions as Parameters, Trees, and Heaps

University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Passing Functions As Parameters

Passing Function as A Parameter

- Old C way: function pointers

- `void func (void (*f)(int));`

- Using `std::function`

```
int BinaryOperation(int a, int b, std::function<int(int, int)> func) {
    return func(a, b);
}

int Add(int a, int b) { return a + b; }

int Mult(int a, int b) { return a * b; }

int main() {
    int result1 = BinaryOperation(10, 20, Add);
    int result2 = BinaryOperation(10, 20, Mult);
}
```

STL Algorithms

- Iterators are used to generalize them
 - Sort
 - Find
 - Reverse
 - Max_element
 - Min_element
 - Accumulate
 - Count
 - Count_if

```
std::vector<int> v = {12, -2, 0, 13, 3, 5};

auto it = std::find(v.begin(), v.end(), 4);

if (it != v.end()) {
    const auto &n = *it;
    std::cout << "Found n: " << n << std::endl;
} else {
    std::cout << "Didn't find." << std::endl;
}
```

STL Algorithms

- We can pass functions to them

```
bool IsOdd(int i) { return ((i % 2) == 1); }

bool IsEven(int i) { return ((i % 2) == 0); }

int main() {
    std::vector<int> v = {12, -2, 0, 0, 1, 12, 5, 3, 13, 3, 5};

    auto count_odd = std::count_if(v.begin(), v.end(), IsOdd);
    auto count_even = std::count_if(v.begin(), v.end(), IsEven);

    return 0;
}
```

Transform (AKA Map)

- Map each x in the container to $f(x)$

```
int IncrementByTen(int a) { return a + 10; }

int main() {
    std::vector<int> inputs = {0, 1, 2, 3, 4, 5, 6, 7, 8};
    std::vector<int> outputs(inputs.size());
    // Increment all of them
    std::transform(inputs.begin(), inputs.end(),
                  outputs.begin(), IncrementByTen);
}
```

Copy_if (AKA Filter)

- Keep item x of the container if $f(x) == \text{true}$

```
bool IsOdd(int i) { return ((i % 2) == 1); }

int main() {
    std::vector<int> inputs = {0, 1, 2, 3, 4, 5, 6, 7, 8};
    std::vector<int> outputs(inputs.size());
    // Increment all of them
    std::copy_if(inputs.begin(), inputs.end(),
                 outputs.begin(), IsOdd);
}
```

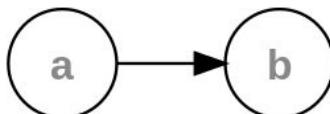
Accumulate (AKA Reduce)

- Homework: implement using std::accumulate

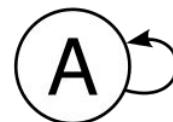
Tree Data Structure

Tree

- A **connected, acyclic**, graph
- collection of nodes (starting at a **root**)
- Each node points to other nodes (the "**children**")
 - **No duplicated** children
 - Nothing points to the **root**
 - **Leaves** point to nothing
 - Each node has at most **one** parent

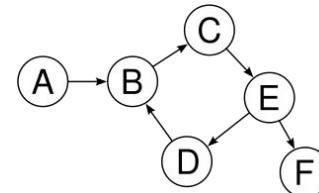


Tree



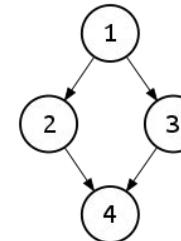
Not Tree

Root cannot
have parent



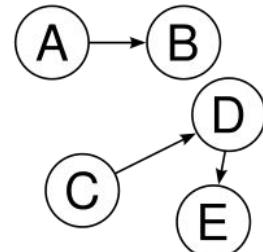
Not Tree

Cycle



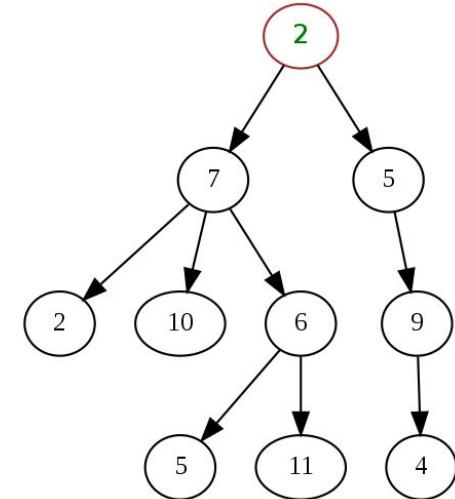
Not Tree

Multiple parents

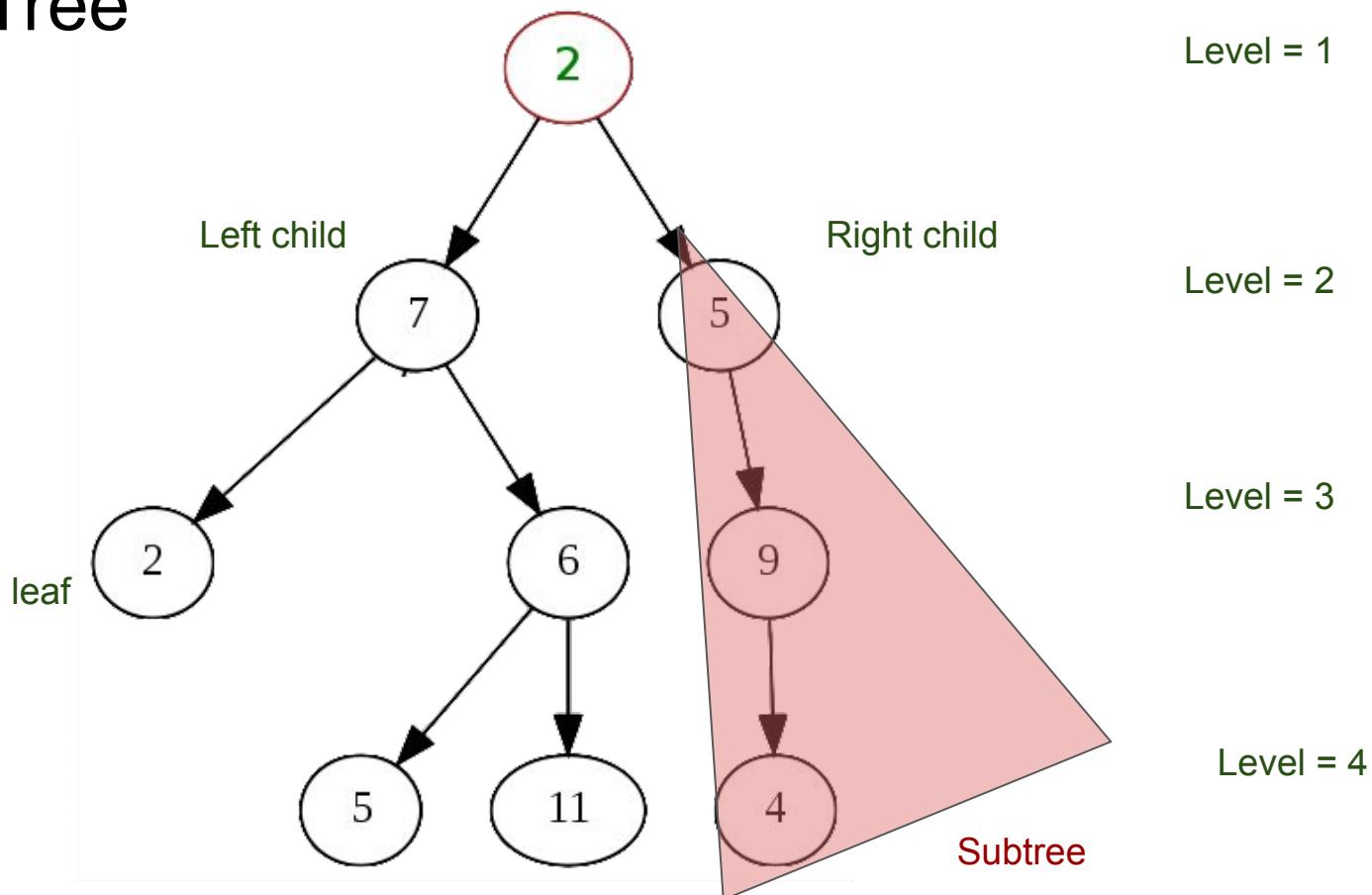


Not Tree

Unconnected

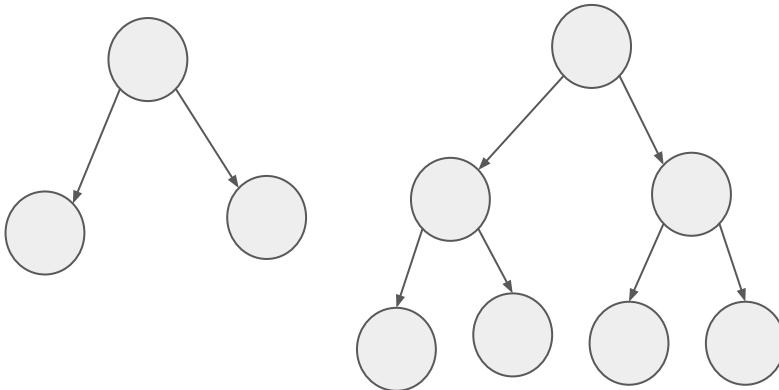


Binary Tree

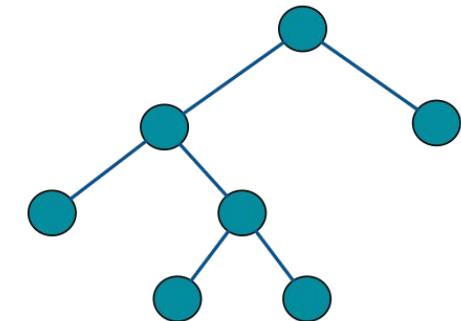


Binary Tree

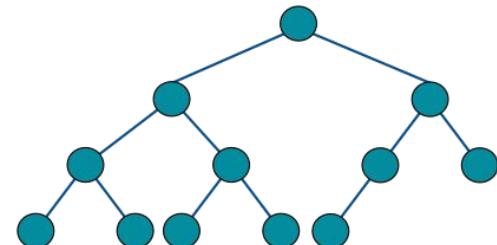
- Full Binary Tree
 - Each node has 0 or 2 children (no 1 child node)
- Complete Binary Tree
 - All levels are full, except the last one
 - Last level is filled from left to right
- Perfect Binary Tree
 - Last level is completely full



Perfect



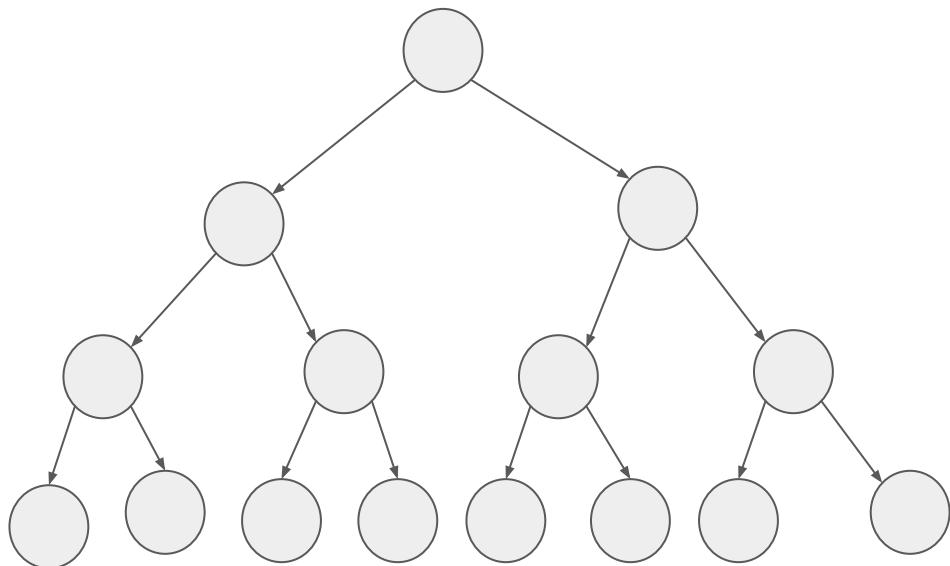
Full Binary Tree



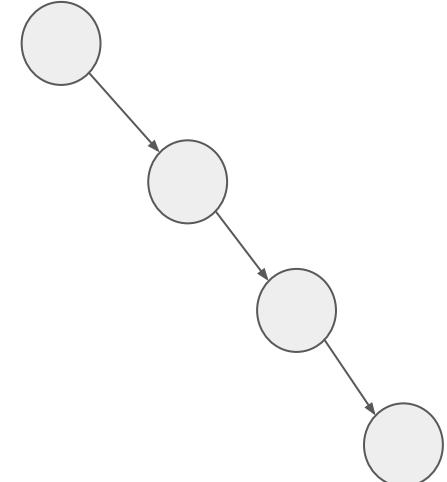
Complete, not full

Tree Height

- $\log_2(n+1) \leq \text{height} \leq n$
- $\log_2(n)$ appears in runtime of a lot of tree-based algorithms



15 nodes, height =4



4 nodes, height =4

Iterating Nodes in Tree (Traversal)

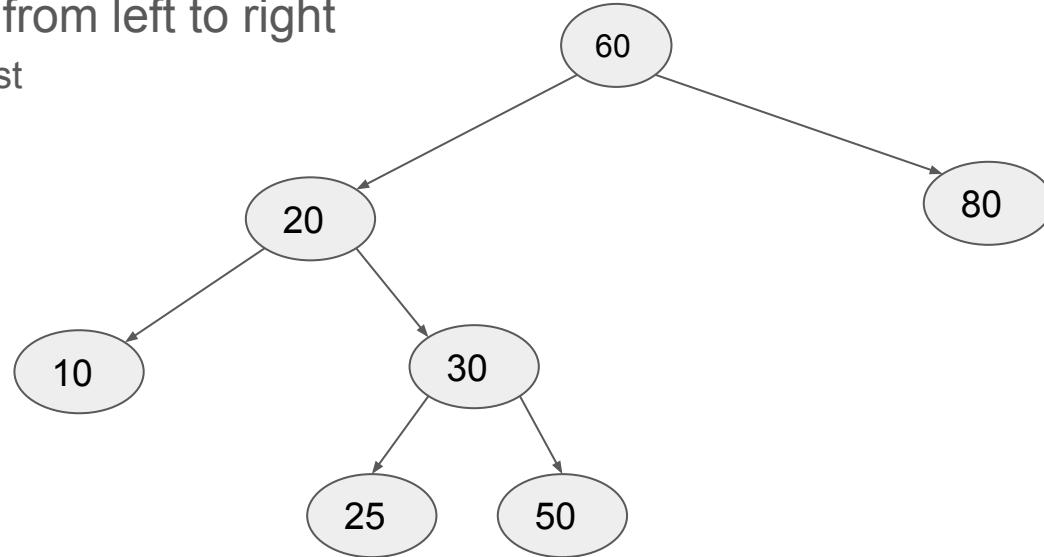
- Preorder (**NLR**): Node, Left, Right.
 - AKA Depth First
- Inorder (**LNR**): Left, Node, Right
- Postorder (**LRN**): Left, Right, Node
- Breadth first: Levels in order from left to right
 - AKA Level order or Breadth First

Pre: **60, 20, 10, 30, 25, 50, 80**

In: **10, 20, 25, 30, 50, 60, 80**

Post: **10, 25, 50, 30, 20, 80, 60**

Level: **60, 20, 80, 10, 30, 25, 50**



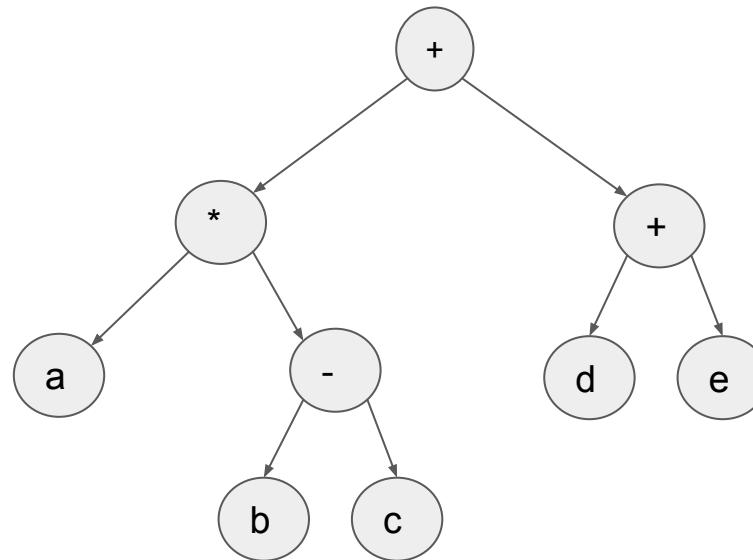
Iterating Nodes in Tree (Traversal)

- Representing expressions of binary operations

In: $a * (b - c) + (d + e)$

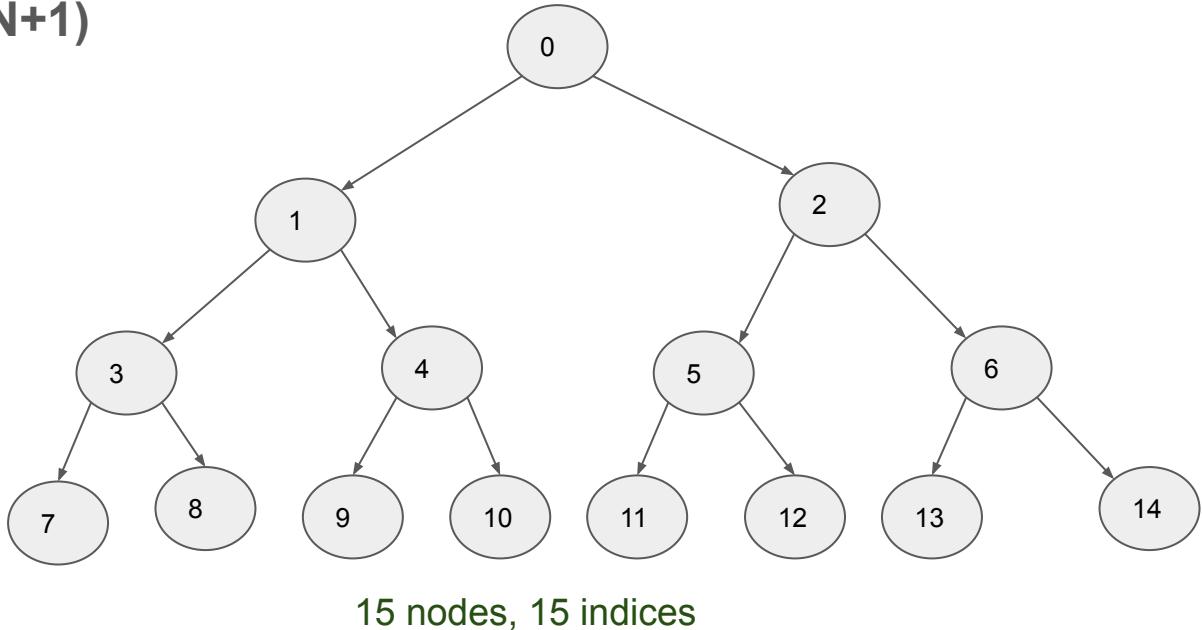
Pre: $+ * a - b c + d e$

Post: $a b c - * d e + +$



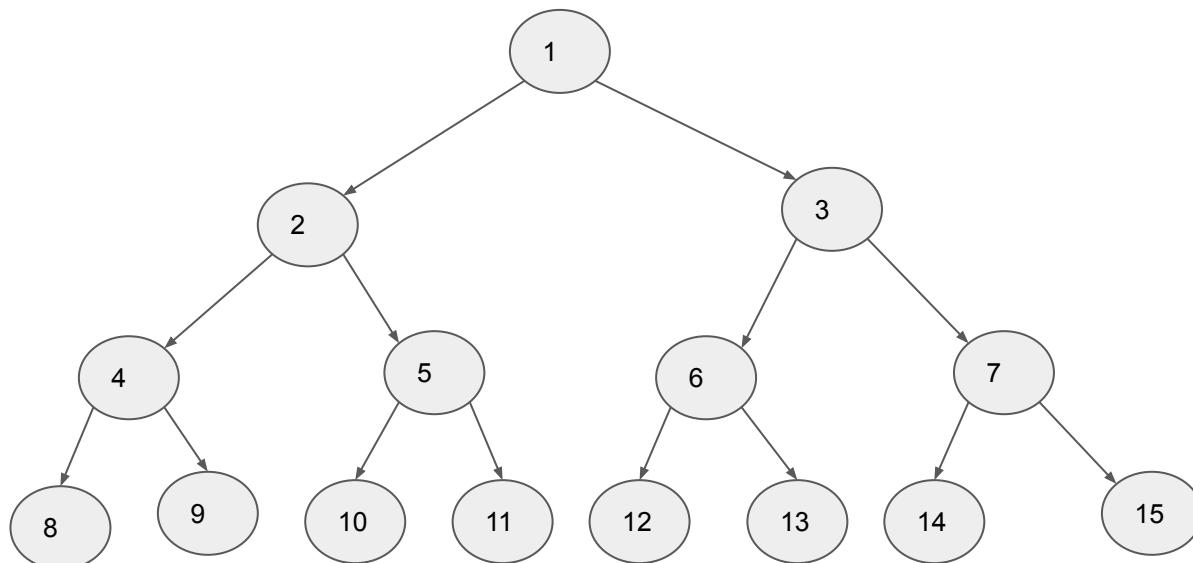
Some Properties of Binary Trees

- The **maximum** number of nodes at level 'l': 2^{l-1}
- Maximum number of nodes in tree of height 'h': $2^h - 1$
- What is the minimum height of a tree of N nodes? **Log(N+1)**



Binary Tree Indexing Nodes Starting from 1

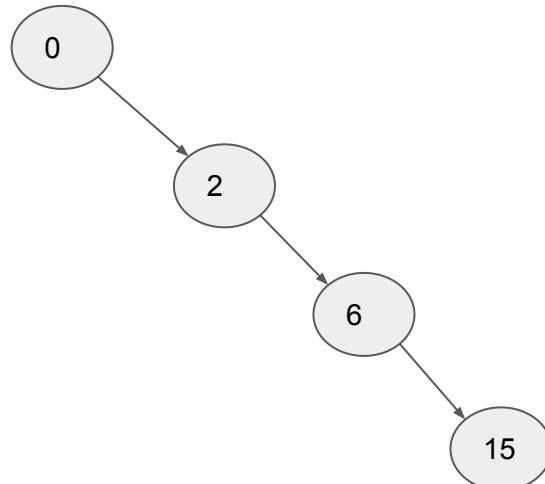
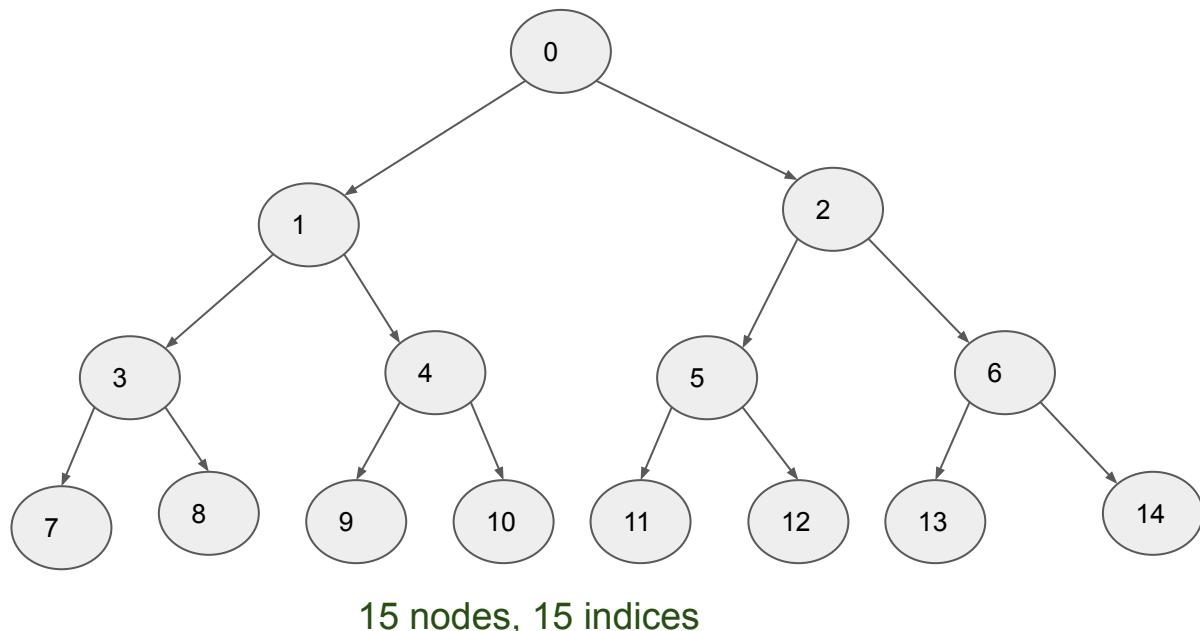
- $\text{Parent}(i) = i / 2$
- $\text{LeftChild}(i) = i * 2$
- $\text{RightChild}(i) = i * 2 + 1$



15 nodes, 15 indices

Binary Tree Indexing Nodes Starting from 0

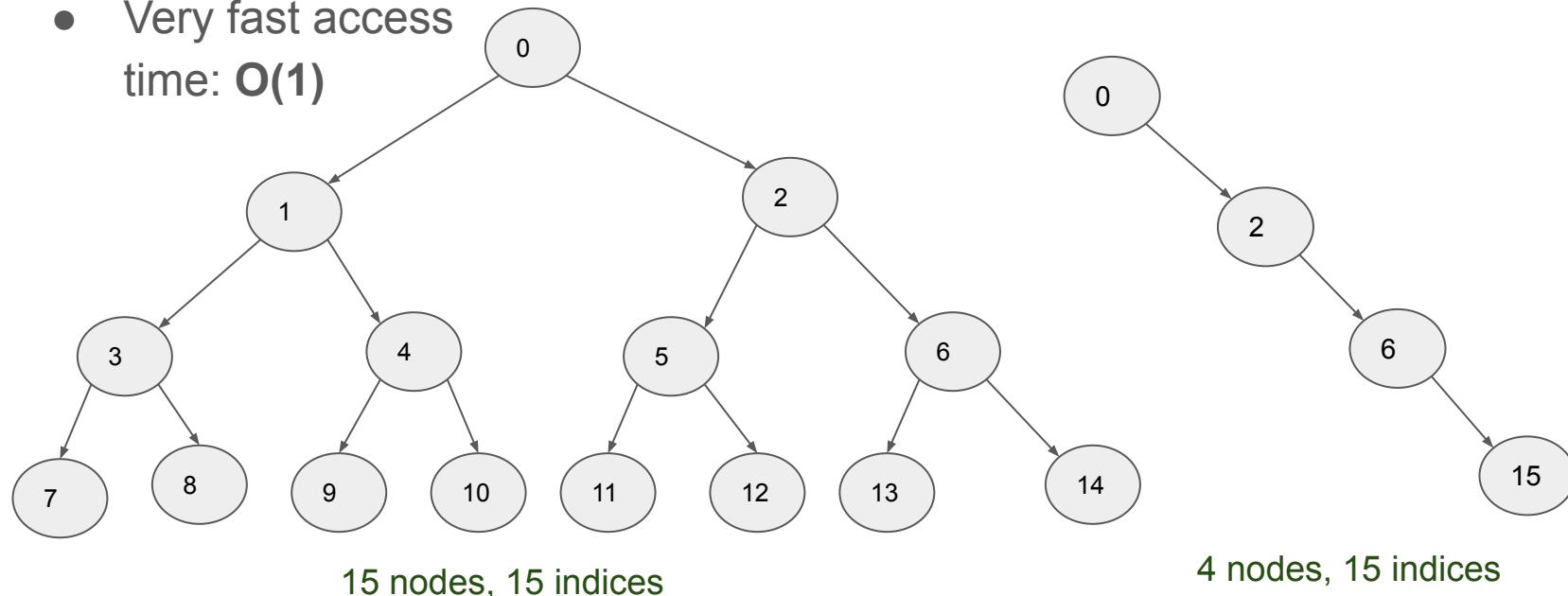
- $\text{Parent}(i) = (i-1) / 2$
- $\text{LeftChild}(i) = i*2 + 1$
- $\text{RightChild}(i) = i*2 + 2$



Storing Binary Trees in Vectors (or Arrays)

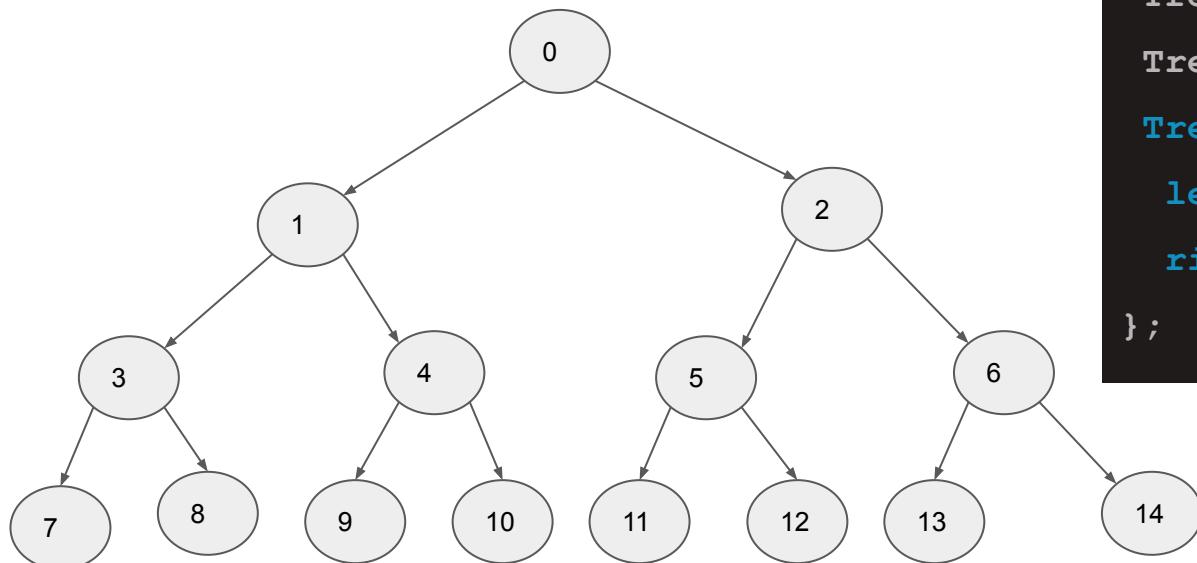
- Good for **complete** trees
- Wasteful for **non-complete** trees
- Very fast access time: $O(1)$

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



Storing Binary Trees Using Pointers

- Less memory overhead
- Acceptable access time: $O(\log n)$



15 nodes, 15 indices

```
struct TreeNode {  
    int val;  
    TreeNode *left;  
    TreeNode *right;  
    TreeNode(int x) : val(x),  
        left(NULL),  
        right(NULL) {}  
};
```

How about Using a Map?

- Very common in Javascript and Python
- In c++ (before c++17), a map can only have **one** key type
 - E.g.: std::map<std::string, int>
- We will revisit this again

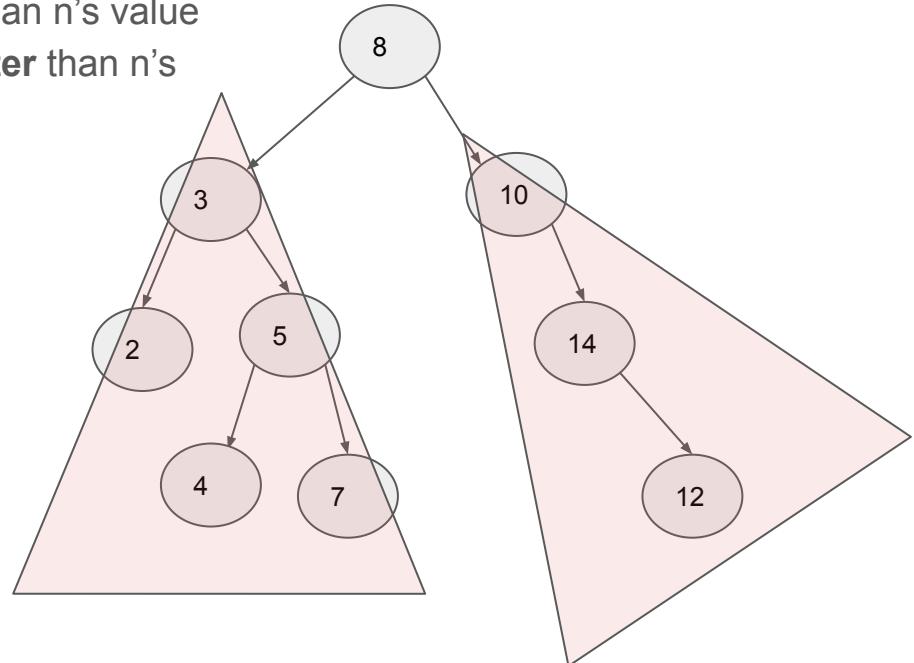
```
{  
    'val': 'A',  
    'left': {  
        'val': 'B',  
        'left': {'val': 'D'},  
        'right': {'val': 'E'}  
    },  
    'right': {  
        'val': 'C',  
        'right': {'val': 'F'}  
    }  
}
```

Example Javascript
object representing a
tree

Binary Search Tree

Binary Search Tree

- A Binary Tree. For each node n:
 - Any node in left subtree has value **less** than n's value
 - Any node in right subtree has value **greater** than n's value
 - The **left** and **right** subtree are **BSTs**
 - There must be no duplicate nodes

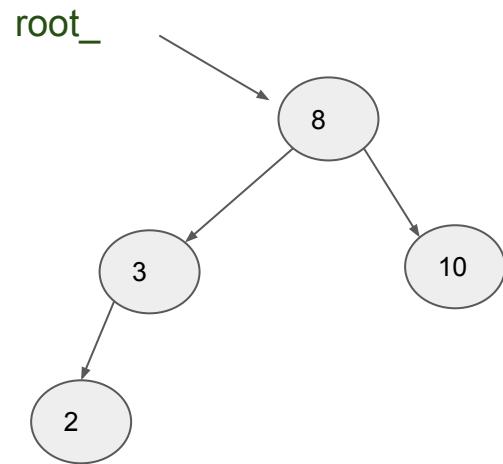


Insert in BST

```
void BST::insert(TreeNode *&root, int v) {  
    if (root == nullptr) {  
        root = new TreeNode(v);  
    } else if (v < root->val) {  
        insert(root->left, v);  
    } else if (v > root->val) {  
        insert(root->right, v);  
    }  
}  
  
t.insert(t.root_, 8);  
t.insert(t.root_, 3);  
t.insert(t.root_, 10);  
t.insert(t.root_, 2);
```

Worst case Runtime is $O(h)$:

$O(\log n) \leq T(n) \leq O(n)$

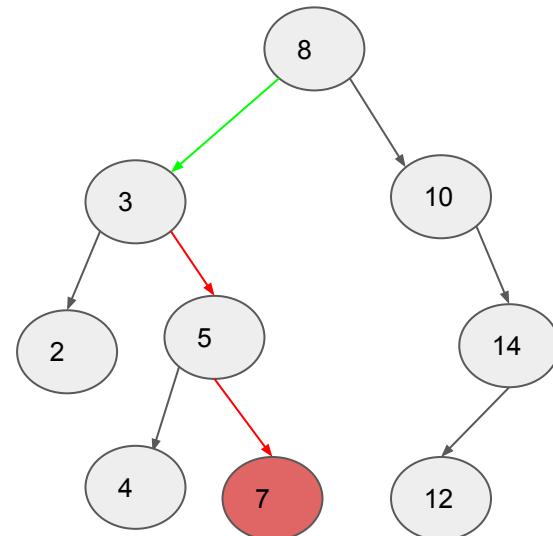


Search in BST

```
TreeNode *BST::search(TreeNode *root, int v) {  
    if (root == nullptr) {  
        return root;  
    }  
    if (root->val == v) {  
        return root;  
    }  
    if (v < root->val) {  
        return search(root->left, v);  
    } else // v > root->val  
    {  
        return search(root->right, v);  
    }  
}
```

Worst case Runtime is $O(h)$:

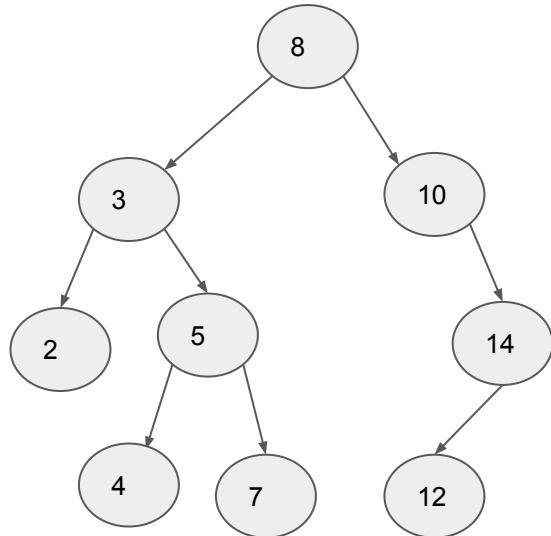
$O(\log n) \leq T(n) \leq O(n)$



t.search(t.root_, 7)

Binary Search Tree

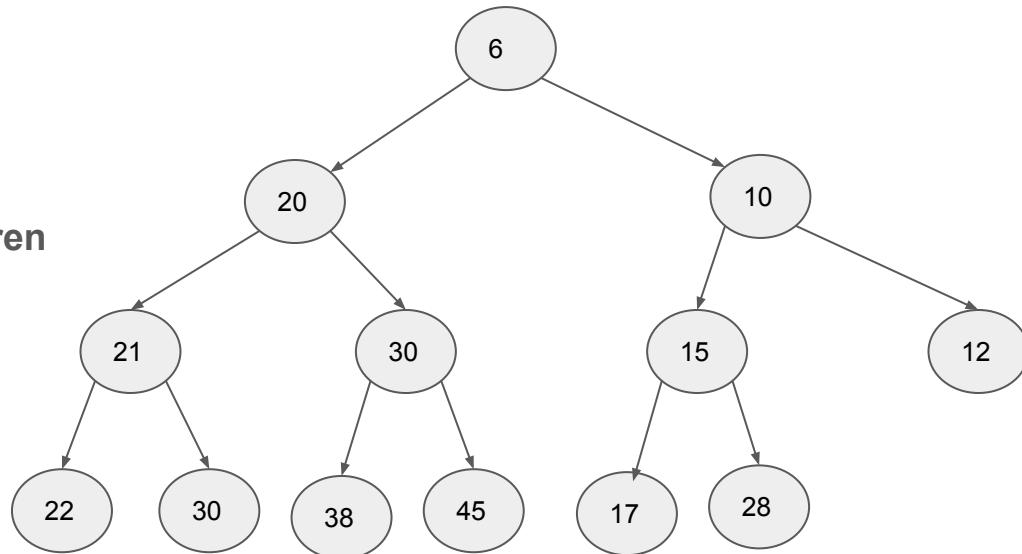
- Fundamental data structure for **Map** and **Set**
- Can reduce the search and insert time from $O(n)$ to $O(\log n)$



Heap Data Structure

Min Heap

- **Complete binary tree**
 - Parent is **less-than both children**
 - Therefore: the **min** is the root.
- **Functions:**
 - push(key)
 - pop()
 - Key = top()



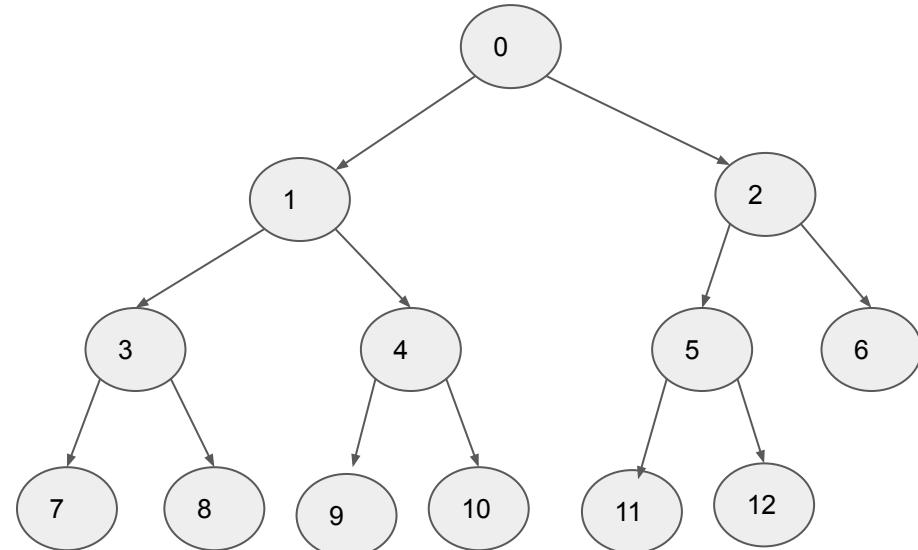
Good news: Heaps are complete binary trees. Therefore, using a **vector** is not that wasteful

```
class Heap {  
  
public:  
  
    int GetParentIndex(int i);  
  
    int GetLeftIndex(int i);  
  
    int GetRightIndex(int i);  
  
    int GetSmallestChildIndex(int i);  
  
    int top();  
  
    void push(int v);  
  
    void pop();  
  
    void TrickleUp(int i);  
  
    void TrickleDown(int i);  
  
private:  
  
    std::vector<int> data_;  
  
};
```

Some Utility Functions

```
int GetLeftIndex(int i) {  
  
    if ((2 * i) + 1 >= data_.size()) {  
  
        return -1;  
  
    }  
  
    return (2 * i) + 1;  
}  
  
int GetRightIndex(int i) {  
  
    if ((2 * i) + 2 >= data_.size()) {  
  
        return -1;  
  
    }  
  
    return (2 * i) + 2;  
}
```

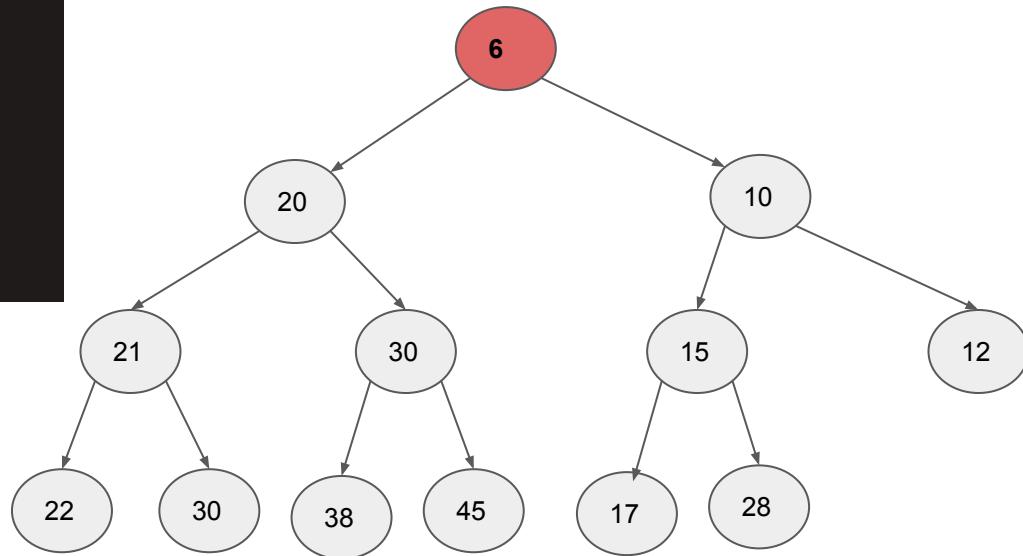
```
int GetParentIndex(int i) {  
  
    if (i == 0) {  
  
        return -1;  
  
    }  
  
    return (i - 1) / 2;  
}
```



Only indices are shown

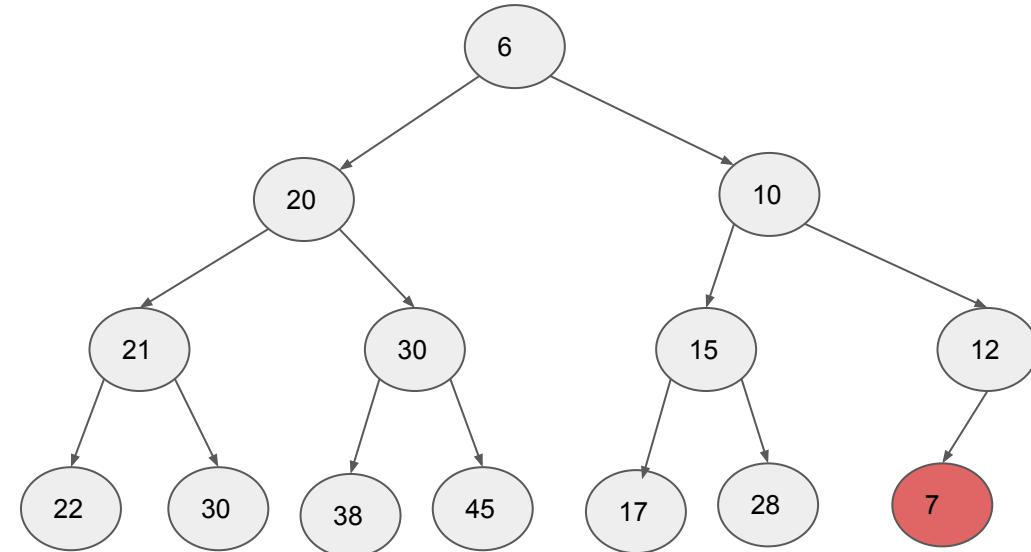
top()

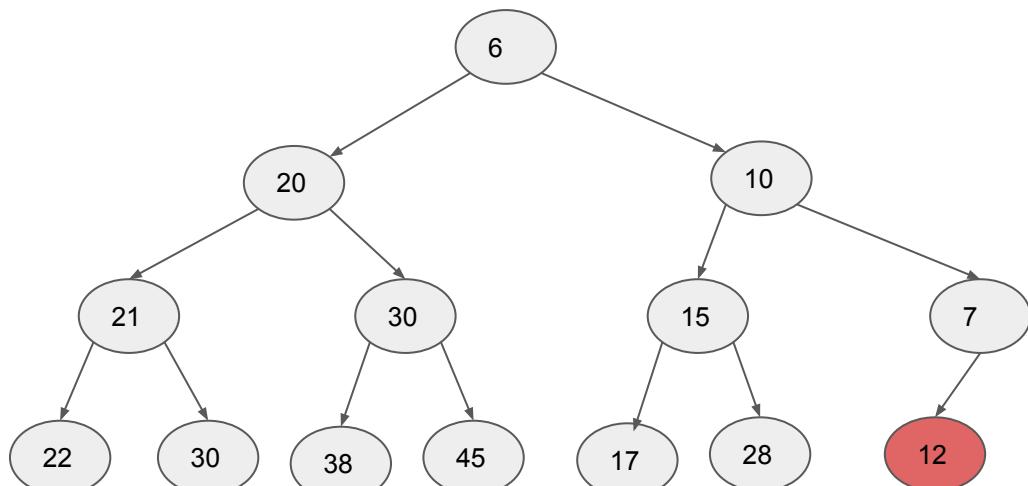
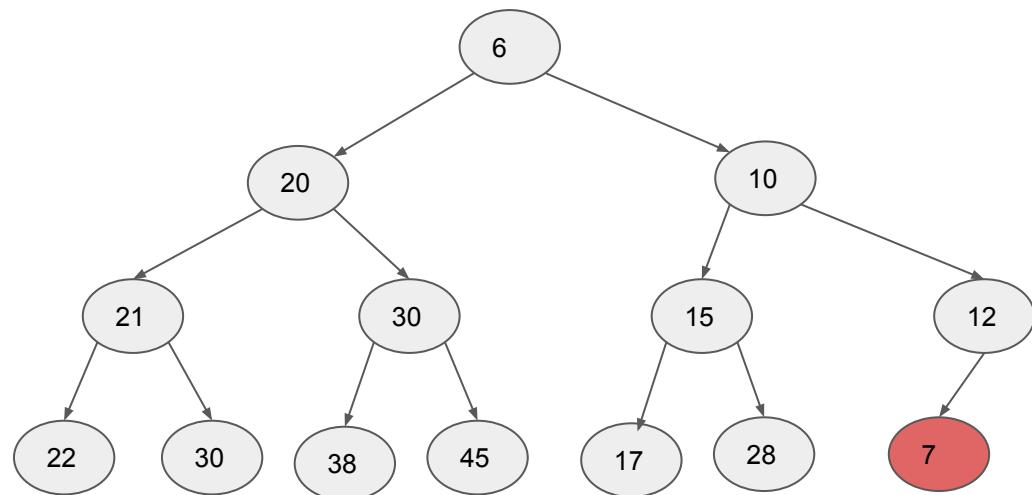
```
int Heap::top() {  
    if (data_.size() == 0) {  
        return INT_MAX;  
    } else {  
        return data_[0];  
    }  
}
```

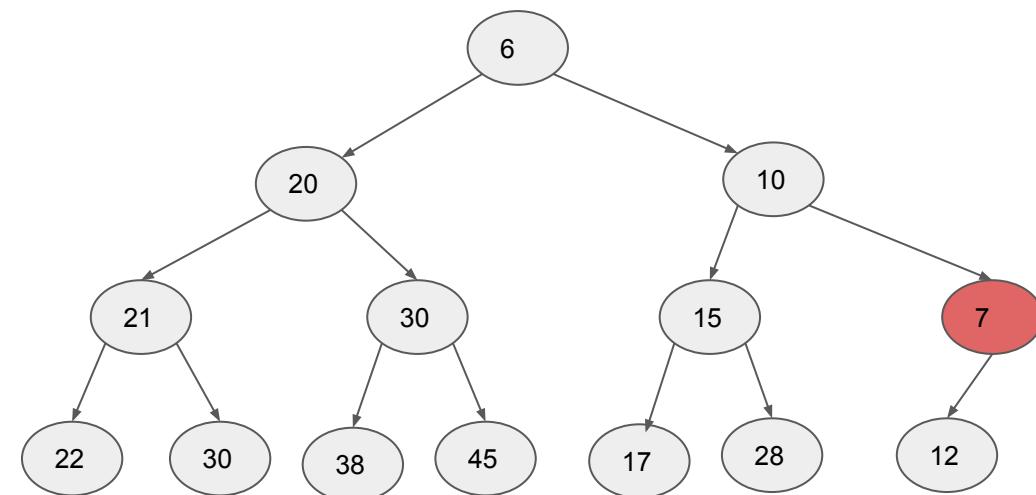


push()

- Add to the end
- Trickle up to preserve heap property
 - also known as bubble-up, percolate-up, sift-up, heapify-up, or cascade-up

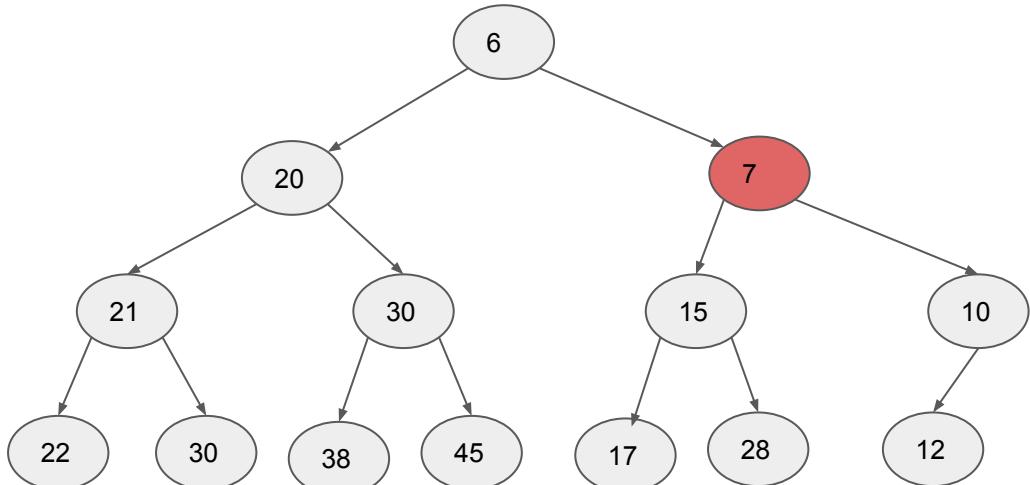






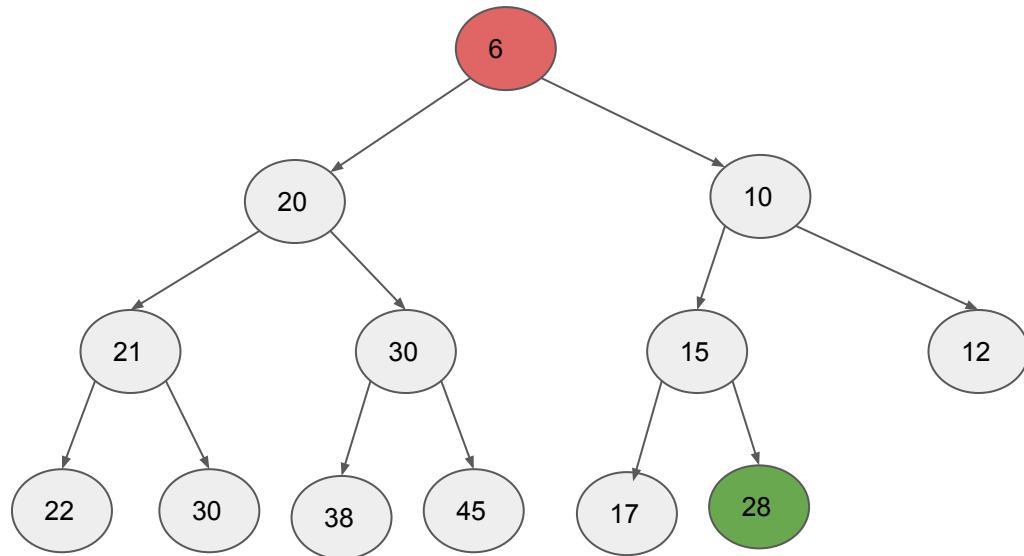
```

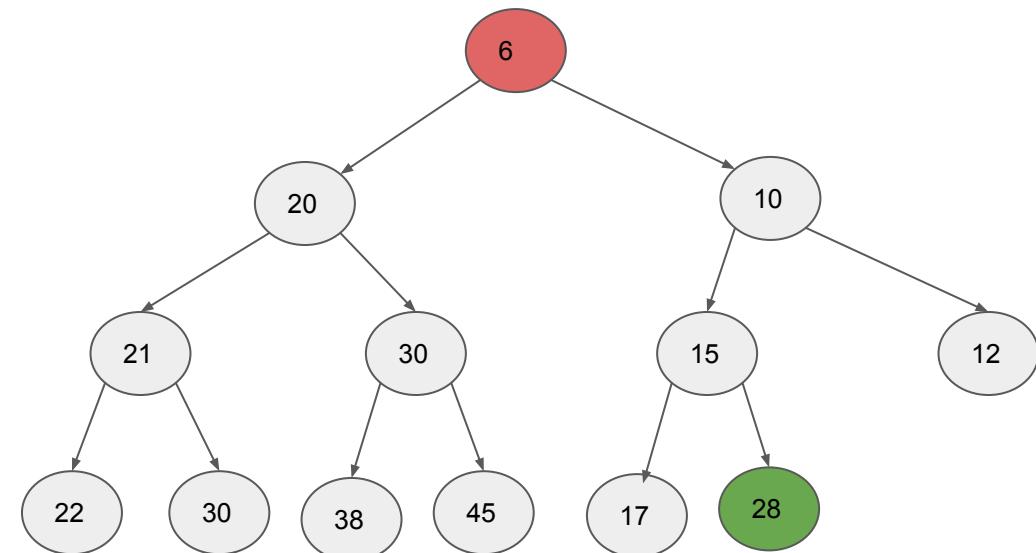
void Heap::TrickleUp(int i) {
    // Fix the min heap property if it is
    // violated
    while (i != 0 && GetParent(i) > data_[i]) {
        Swap(data_[i], data_[GetParentIndex(i)]);
        i = GetParentIndex(i);
    }
}
  
```



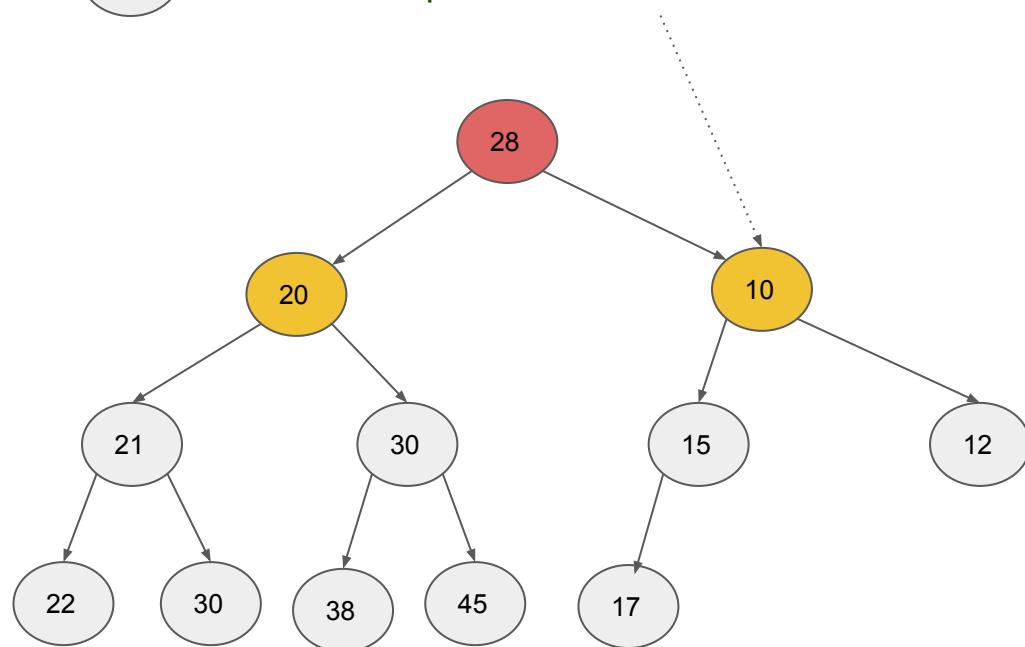
pop()

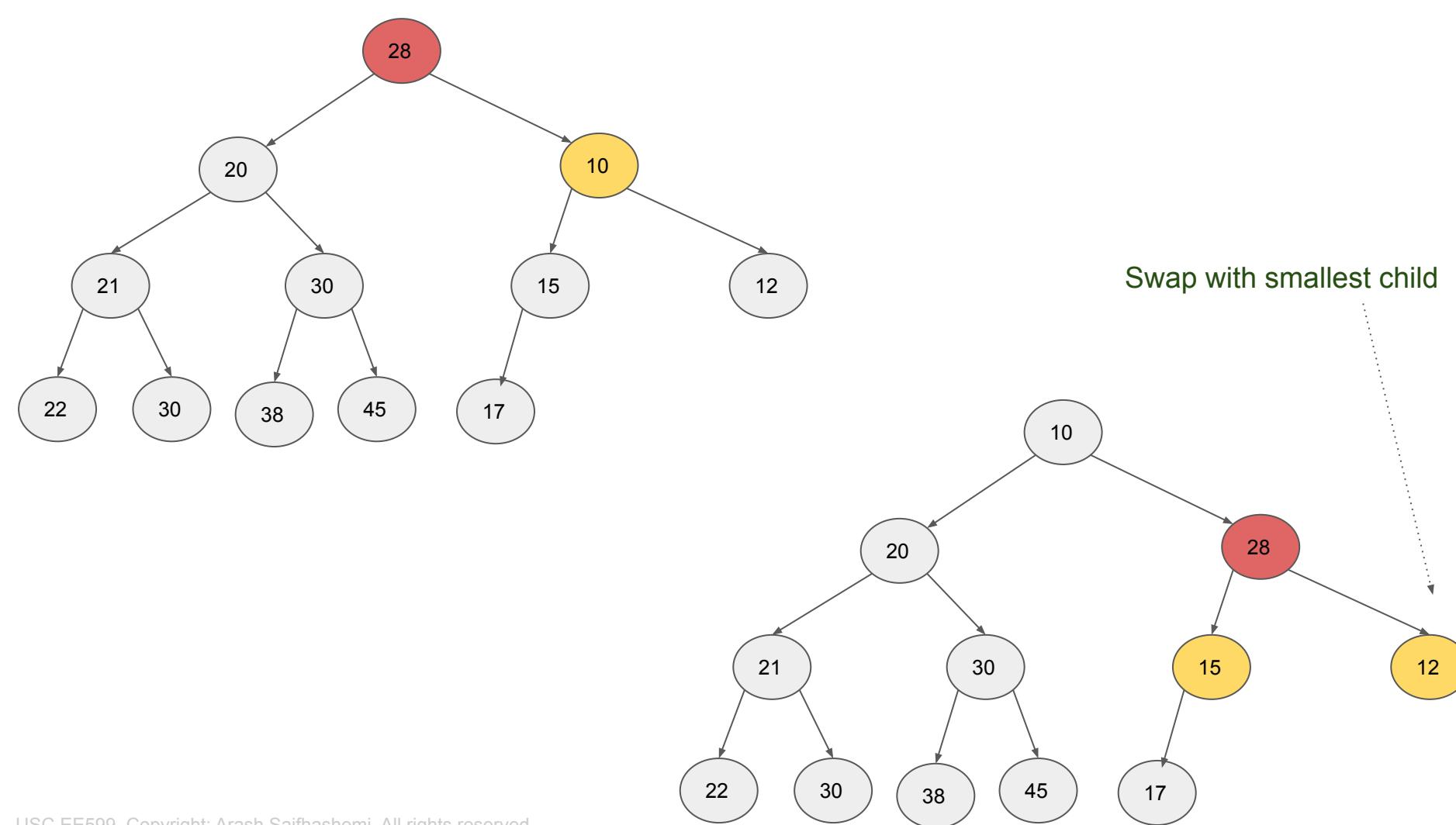
- Remove root and replace it with the last item
- Trickle down to preserve heap
 - also known as bubble-down, percolate-down, sift-down, sink-down, heapify-down, cascade-down, and extract-min

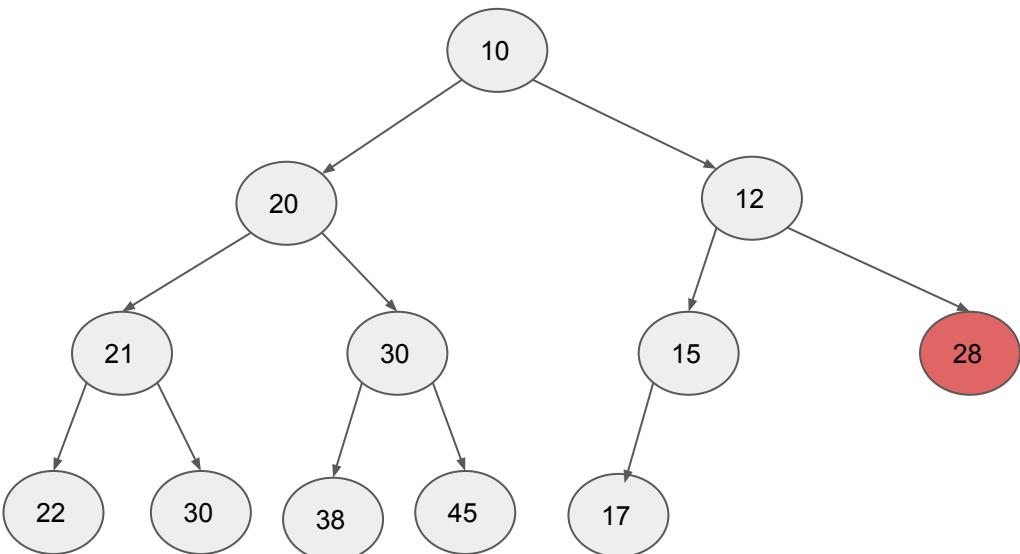
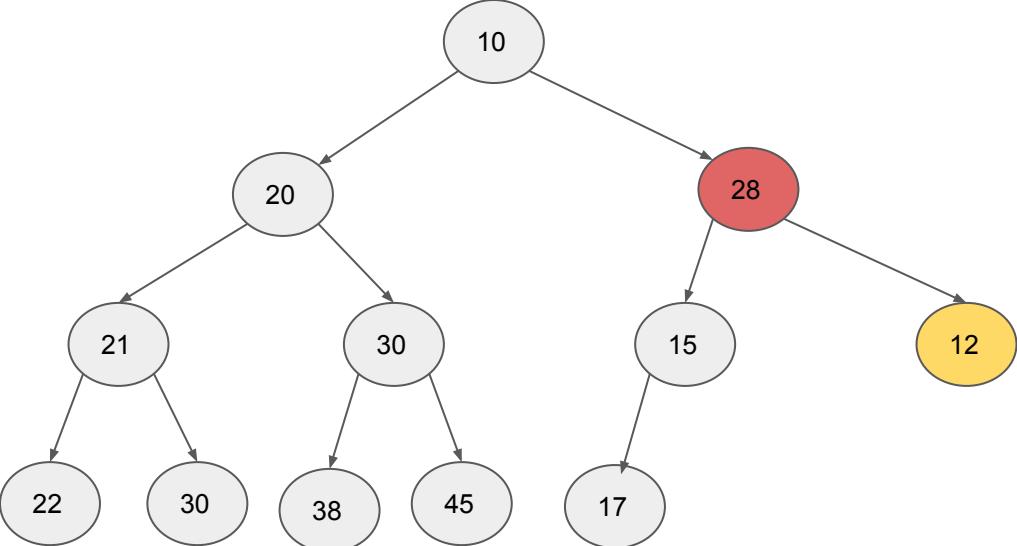




Swap with smallest child

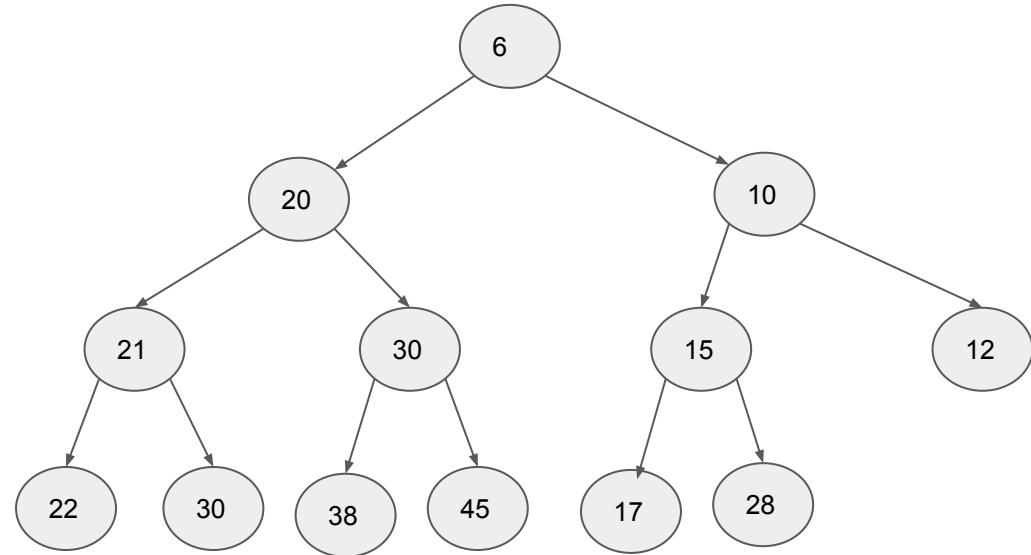






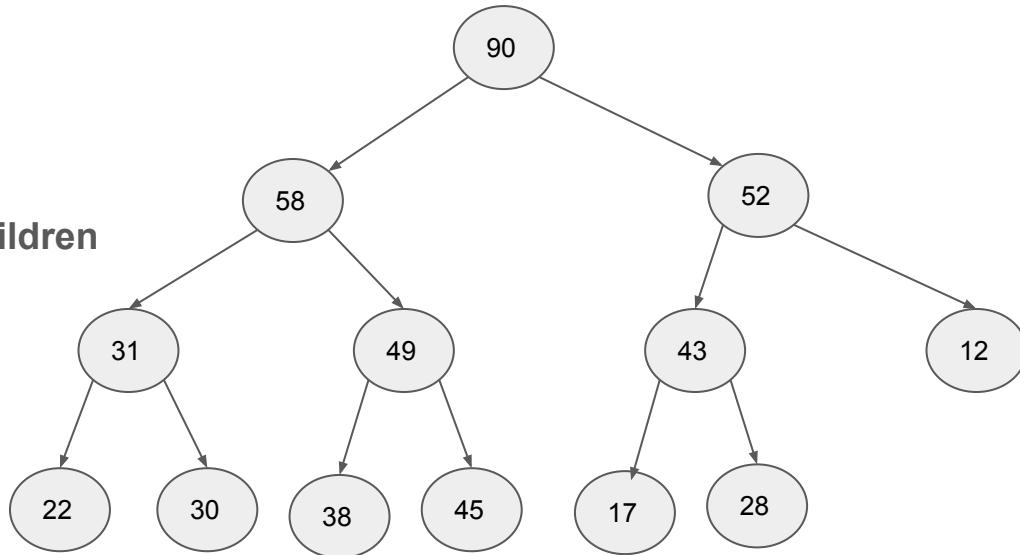
Why did we do this?

- What is the runtime?
- Functions:
 - `push(key)`: $O(\log n)$
 - `pop()`: $O(\log n)$
 - `Key = top()`: $O(1)$



Max Heap

- Complete binary tree
 - Parent is **greater-than both children**
 - Therefore: the **max** is the root.
- Functions:
 - push(key)
 - pop()
 - Key = top()



EE599: Computing and Software for Systems Engineers

Lecture 6: Sorting - Graph Algorithms

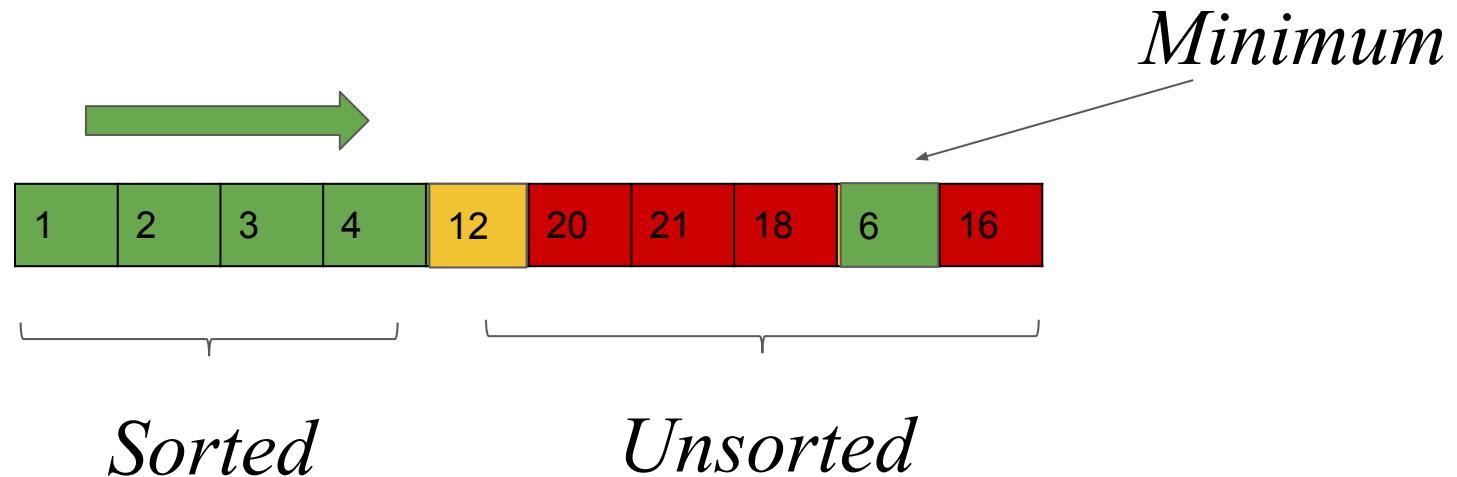
University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Announcements

- Midterm:
 - **March 11 from 6:45pm to 8:45pm**
 - Location: Same as our lecture **SOS-B46**
 - Questions may include **everything** that we covered in the **lectures** until that date.
 - The exam is **closed book, closed laptop**, and **no cell phone usage** is allowed.
 - You can bring a **single one sided letter size page** with your own notes and use it during the exam.
 - You will be asked to write reasonable code in C++. Minor syntax errors would be OK.
 - The complexity of the questions will be similar to your homework assignments.
- Academic Conduct
 - **Reminder:** Students are encouraged to collaborate on general solution strategies for homework. The writeup, however, must be your own - you may not copy someone else's solution.
 - In addition, your homework should list all the fellow students with whom you discussed the solutions.

Selection Sort

- Divide the array into two parts:
 - Sorted
 - Unsorted
- Grow the sorted section by finding the minimum value in unsorted and add it to the sorted section.



Sorted

Unsorted

SelectionSort

- Use **FindMinIndex** to get the index of the min value in the unsorted section

```
void Sort::SelectionSort(std::vector<int> &input) {  
    for (int i = 0; i < int(input.size() - 1); i++) {  
        int min_index = FindMinIndex(input, i);  
        Swap(input[i], input[min_index]);  
    }  
}
```

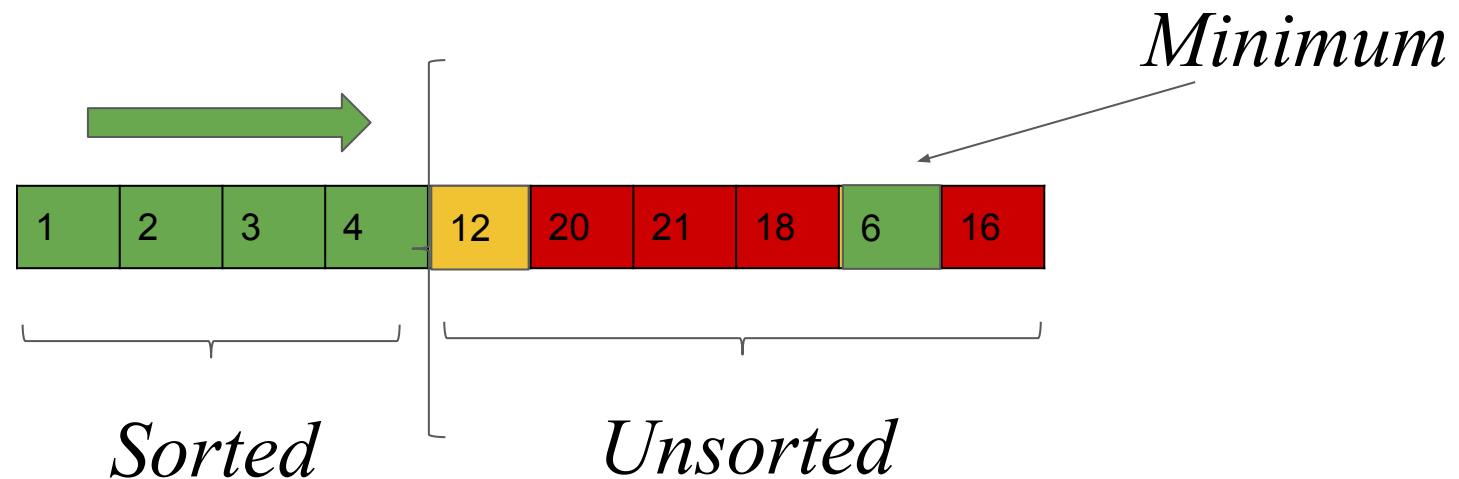
FindMinIndex

- Start from **start_index** and keep the **cur_min** value and **min_index**

```
int Sort::FindMinIndex(const std::vector<int> &input, int start_index) {  
    int min_index = start_index;  
    int cur_min = input[start_index];  
    for (int i = start_index; i < input.size(); i++) {  
        if (input[i] < cur_min) {  
            cur_min = input[i];  
            min_index = i;  
        }  
    }  
    return min_index;  
}
```

Selection Sort

- What do we need to know?
 - Worst case runtime: $O(n^2)$
 - Memory: $O(1)$



Bubble Sort

- Iterate the items from left to right
- Swap neighbors if out of order
- Repeat until there is no more swap possible

1	2	16	3	4	20	21	18	12	15
---	---	----	---	---	----	----	----	----	----



Swap

1	2	3	16	4	20	21	18	12	15
---	---	---	----	---	----	----	----	----	----



Swap

BubbleSort

- Keep iterating until there is no more swaps

```
void Sort::BubbleSort(std::vector<int> &input) {  
    bool go;  
    do {  
        go = false;  
        for (int i = 0; i < int(input.size() - 1); i++) {  
            if (input[i] > input[i + 1]) {  
                Swap(input[i], input[i + 1]);  
                go = true;  
            }  
        }  
    } while (go);  
}
```

Bubble Sort

- After pass 1, the last item is the largest
- After pass i , the last i items are sorted
- How many passes do we need in the worst case?

Beginning
of pass 1

21	2	3	16	4	20	9	18	12	15
----	---	---	----	---	----	---	----	----	----

End of
pass 1

1	2	3	4	16	20	18	12	15	21
---	---	---	---	----	----	----	----	----	----

Bubble Sort Improved

- Idea:

- $n-1$ passes (at most)
 - In pass i , the last i items are already sorted

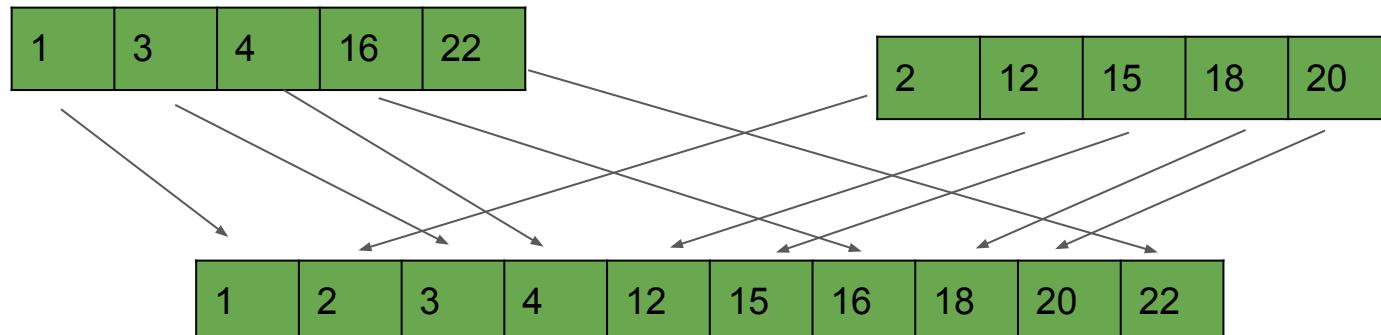
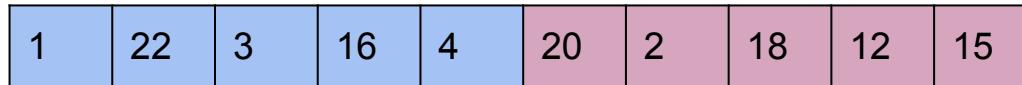
```
void Sort::BubbleSortImproved(std::vector<int> &input) {  
    int n = input.size();  
  
    for (int i = 0; i < n - 1; i++) {  
  
        bool swapped = false;  
  
        for (int j = 0; j < n - 1 - i; j++) {  
  
            if (input[j] > input[j + 1]) {  
  
                Swap(input[j], input[j + 1]);  
  
                swapped = true;  
            }  
        }  
  
        if (!swapped) {  
  
            break;  
        }  
    }  
}
```

Bubble Sort

- Worst case runtime: $O(n^2)$
- Memory: $O(1)$

Merge Sort

- Idea (Divide and Conquer):
 - Sort each half
 - Merge the sorted results
- Merge Sort:
 - Do this recursively until the size of items is 1 (Vector of size 1 is already sorted!)



MergeSort

```
// l: first index, r: last index (included)

void Sort::MergeSortHelp(std::vector<int> &input, int l, int r) {

    if (l < r) {

        int m = (l + r) / 2;

        // Sort first and second halves

        MergeSortHelp(input, l, m);

        MergeSortHelp(input, m + 1, r);

        Merge(input, l, m, r);

    }

}

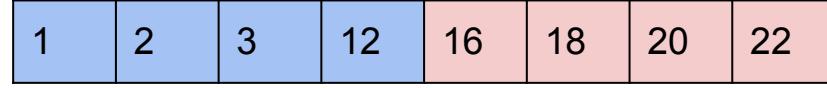
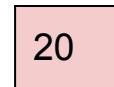
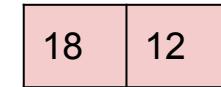
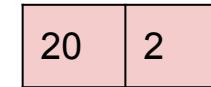
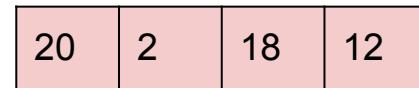
// Calls MergeSortHelp with l=0, r = input.size() - 1

void Sort::MergeSort(std::vector<int> &input) {

    MergeSortHelp(input, 0, int(input.size() - 1));

}
```

Notice: we convert to int because size() returns unsigned value



Merge

- We have two sub-vectors:
 - Left: starts from **l**, ends at **m**. We use **left_index** to iterate through it.
 - Right: starts from **m+1**, ends at **r**. We use **right_index** to iterate through it.

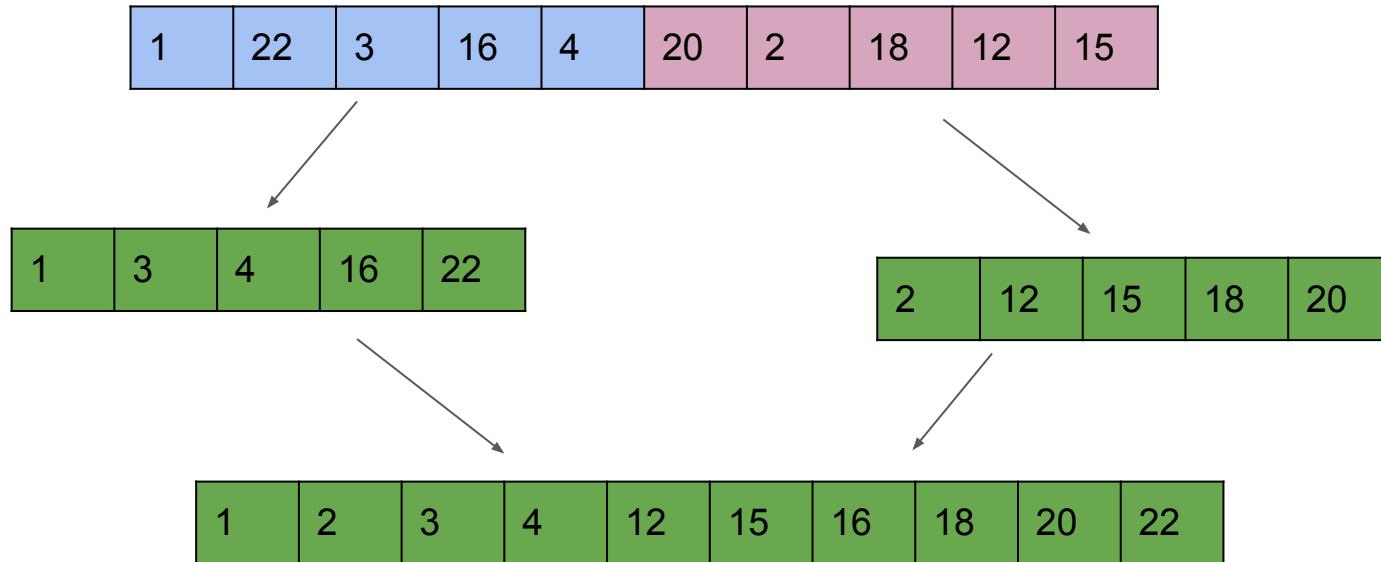
```
void Sort::Merge(std::vector<int> &input, int l, int m, int r) {  
    // We only copy from l to r (including r)  
  
    std::vector<int> input_copy(input.begin() + l, input.begin() + r + 1);  
  
    int left_index = 0;  
  
    int right_index = m + 1 - 1; // adjust m+1 by subtracting 1 from it  
  
    int left_max_index = m - 1; // Last index of left half  
  
    int right_max_index = r - 1; // Last index of right half  
  
    for (int i = l; i <= r; i++) {  
  
        input[i] = GetMinValueAndIncrementItsIndex(  
            input_copy, left_index, right_index, left_max_index, right_max_index);  
    }  
}
```

Get Min Value And Increment Its Index

```
int Sort::GetMinValueAndIncrementItsIndex(std::vector<int> &input,
                                         int &left_index, int &right_index,
                                         const int left_max_index,
                                         const int right_max_index) {
    if (left_index > left_max_index) {
        return input[right_index++];
    }
    if (right_index > right_max_index) {
        return input[left_index++];
    }
    if (input[left_index] <= input[right_index]) {
        return input[left_index++];
    } else {
        return input[right_index++];
    }
}
```

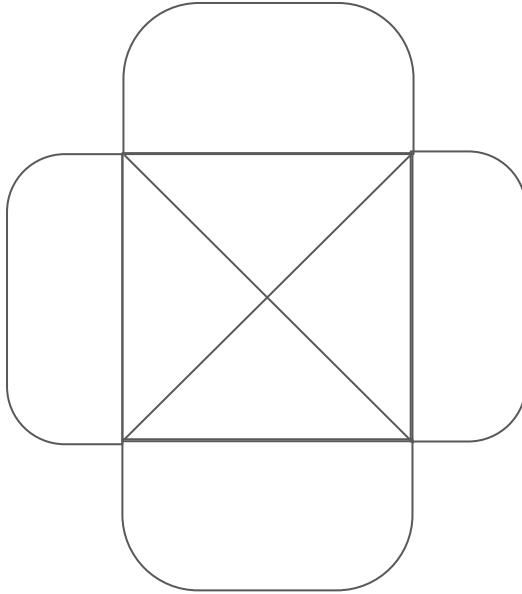
Merge Sort

- Worst case runtime: $O(n \log n)$
 - $T(n) = 2T(n/2) + n$
- Memory: $O(n)$



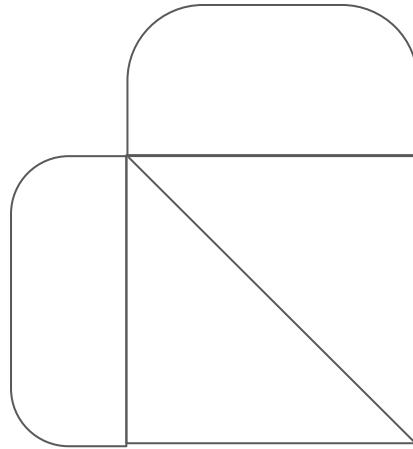
What is a Graph?

History



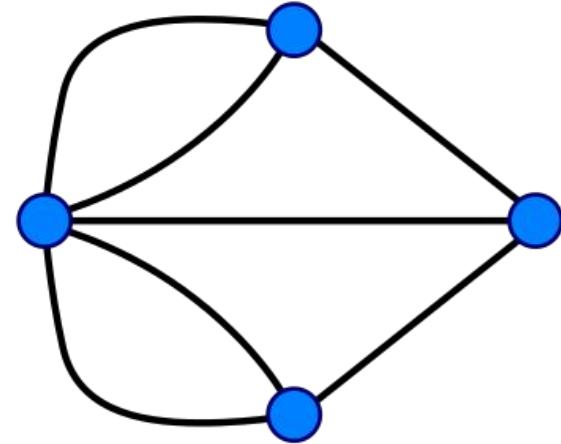
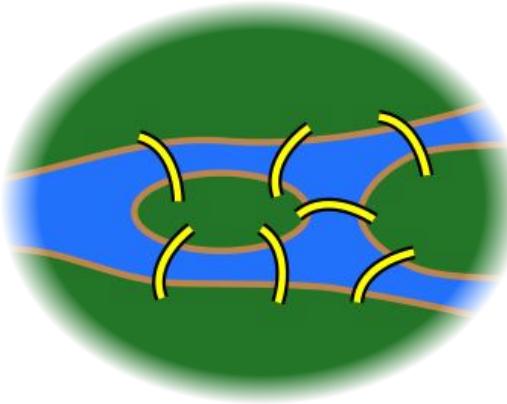
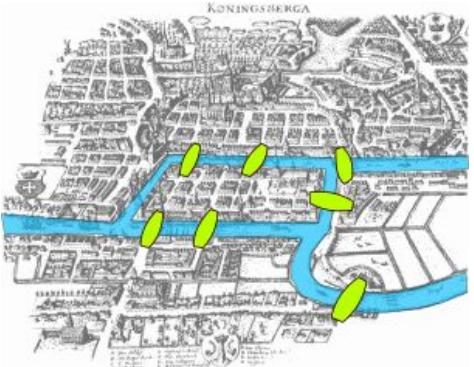
Can you draw this continuously without disconnecting the pen and paper and never going backwards?

History



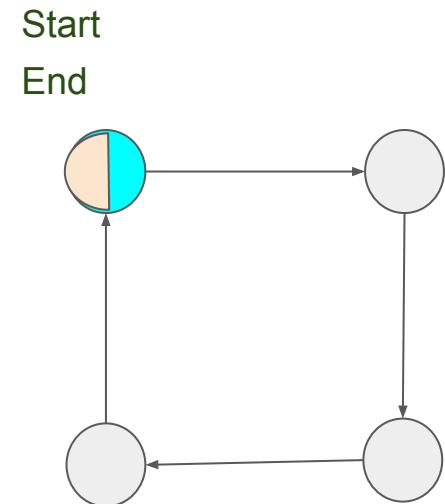
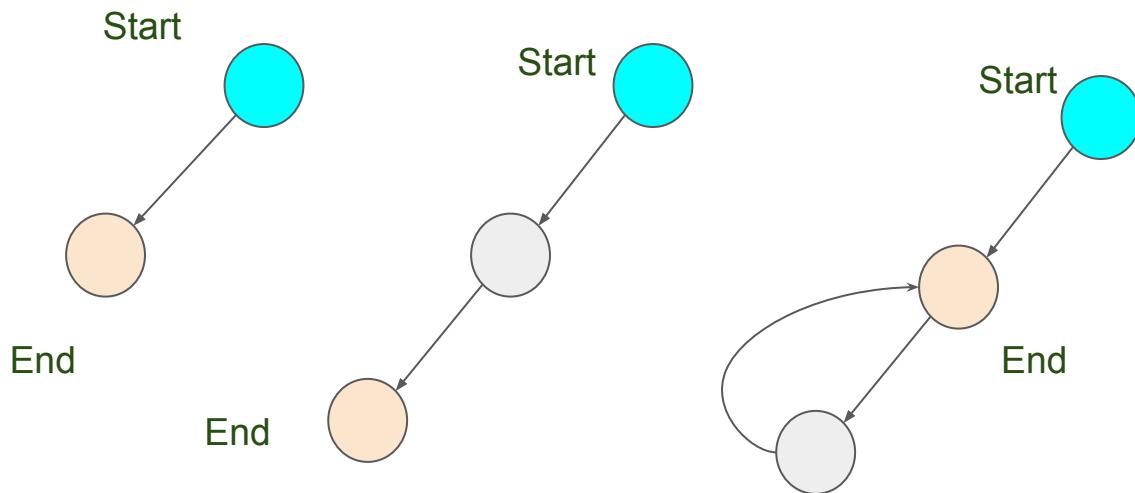
How about this simpler one?

The Seven Bridges of Königsberg



The city **bridges** and **land masses** can be **abstracted** to a Graph

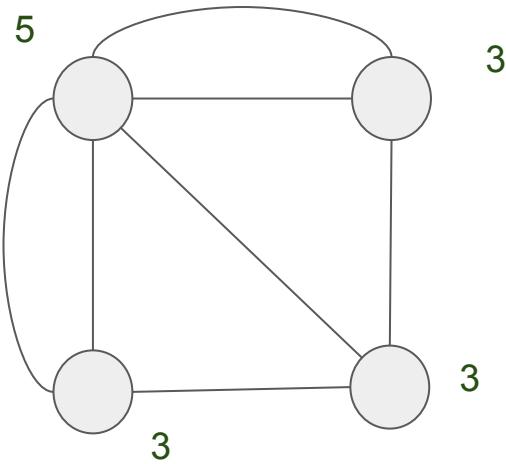
Euler's Observation



Observation:

- If we enter a vertex by a bridge, we should leave the vertex (except **may be** the **Start** and **End** vertices)
- Therefore: At most **two** vertices can have **odd** degree

Euler's Proof by Contradiction



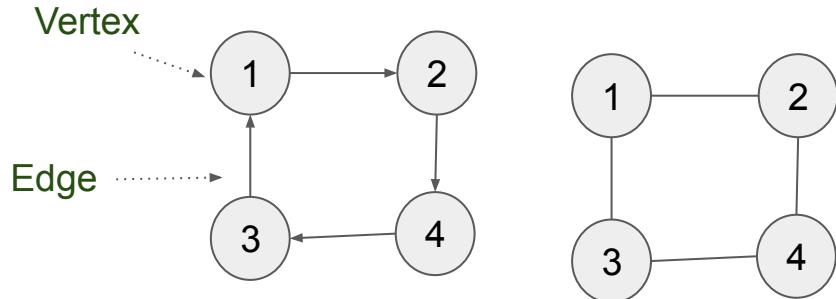
- At most **two** vertices can have **odd** degree
- We have **four vertices** with odd degree

⊥ (Contradiction)

Graph Representation

Graph Definition

- $G=(V, E)$
 - V : Set of **vertices** (AKA Nodes)
 - E : Set of edges
 - i. **Undirected Graph: Two-sets** of vertices
 - 1. Two-Set is a set of size 2
 - ii. **Directed Graph: Pairs** of vertices
- Graph Representation
 - It really comes down to representing **sets** and **two-sets/pairs!**



$V=\{1, 2, 3, 4\}$

Undirected:

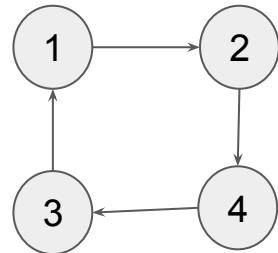
$E=\{\{1,2\}, \{2,4\}, \{4,3\}, \{3,1\}\}$

Directed:

$E=\{(1,2), (2,4), (4,3), (3,1)\}$

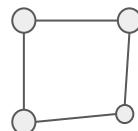
Some quick definitions

- **Simple graph:** no self-loops
 - Usually we limit ourselves to simple graphs
- Size
 - $|V|$: Size of V, usually denoted as n
 - $|E|$: Size of E, usually denoted as m
 - What is the max value of m?
- If $e = (a,b) \in E$, then **a** and **b** are **adjacent** (AKA **neighbors**)
 - In a directed graph **a** is **predecessor (or parent)** and **b** is **successor (or child)**.

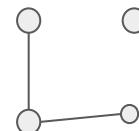


Connectivity

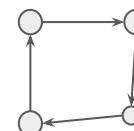
- **Undirected Graph:**
 - Connected if there is a path between each two nodes
- **Directed Graph:**
 - **Weakly connected:** If replacing all of its directed edges with undirected edges produces a connected (undirected) graph.
 - **Unilaterally connected:** if it contains a path from u to v OR from v to u for every u, v .
 - **Strongly connected:** if it contains a path from u to v AND from v to u for all u, v .



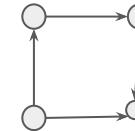
Connected



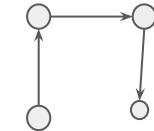
Disconnected



**Strongly
Connected**



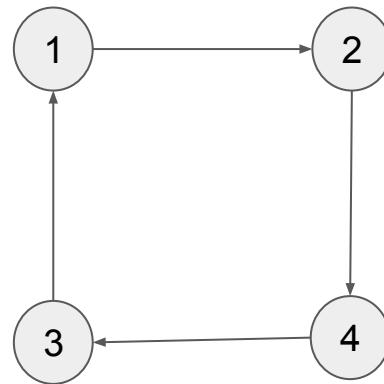
**Weakly
Connected**



**Unilaterally
Connected**

Graph Representation

- **V:** Set/List/Vector of vertices
E: Set/List/Vector of Two-sets/Pairs
 - When using List/Vector instead of Set, we implicitly mean there is no duplicates
- **V:** Set/List/Vector of vertices
E: For each vertex: Set/Map/List/Vector of all adjacent vertices
- **V,E:** Adjacency matrix



v={1, 2, 3, 4}

E={(1,2), (2,4), (4,3), (3,1)}

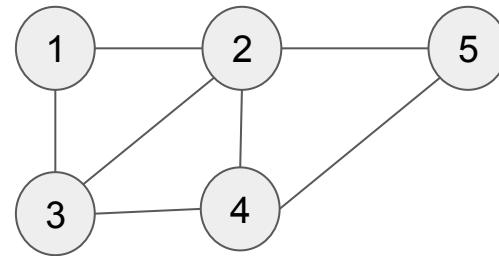
V: Set of Vertices, E: Set of Pairs

V: Set of Vertices, E: Vector of Pairs

```
class Graph {  
public:  
    Graph(std::vector<int> v, std::vector<std::pair<int, int>> e)  
        : v_(v), e_(e) {}  
    std::vector<int> v_;  
    std::vector<std::pair<int, int>> e_;  
};  
  
int main() {  
    std::vector<int> v = {1, 2, 3, 4};  
    std::vector<std::pair<int, int>> e = {{1, 2}, {2, 4}, {4, 3}, {3, 1}};  
    Graph g(v, e);  
}
```

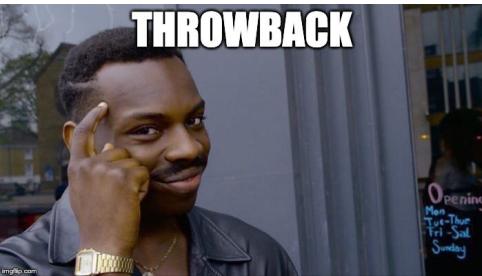
V : Set of Vertices, E : Adjacency List

1	2, 3
2	1, 3, 4, 5
3	1, 2, 4
4	2, 3, 4



What data structure can we use to represent this?

1. Set/Vector/List of Set/Vector/lists
2. Map of vertices to Set/Vector/List of adjacent vertices



Throwback

- Set
 - Automatically sorted
 - Insert/Delete/Find: $\Theta(\log n)$
 - Set of user defined objects requires defining the sort function (we haven't covered this yet)
- Unordered Set
 - Not sorted
 - Insert/Delete/Find: $O(1)$ (amortized)
 - Unordered Set of user defined objects requires defining the hash function (we haven't covered this yet)

- Vector
 - Not Sorted
 - Insert: $O(1)$, Delete: $O(n)$
- Sorted Vector
 - Insert: $O(n)$, Delete: $O(n)$
- List
 - Not Sorted
 - Insert/Delete/Find: $O(n)$,
Insert/Delete once found: $O(1)$ (unlike vector)

V: Vector of Vertices, E: Adjacency List (Vector)

```
struct Vertex {  
    Vertex(int v, std::set<int> a) : vertex_number(v), adjacents(a) {}  
    int vertex_number;  
    std::set<int> adjacents;  
};  
  
class Graph {  
public:  
    Graph(std::vector<Vertex> v) : v_(v) {}  
    std::vector<Vertex> v_;  
};  
  
int main() {  
    Graph g1({Vertex(1, {2, 3}),  
              Vertex(2, {1, 3, 4, 5}), Vertex(3, {1, 2, 4}),  
              Vertex(4, {2, 4, 4})});  
}
```

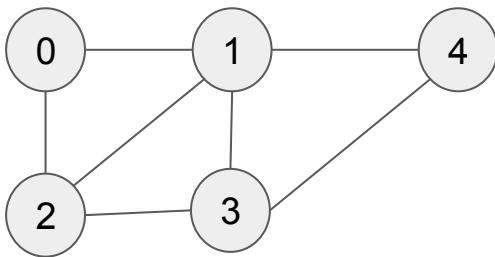
1	2, 3
2	1, 3, 4, 5
3	1, 2, 4
4	2, 3, 4

Using a Map and Set

```
class Graph {  
  
public:  
  
    Graph(std::map<int, std::set<int>> &vertices) : v_(vertices) {}  
  
    std::map<int, std::set<int>> v_;  
  
};  
  
int main() {  
  
    std::map<int, std::set<int>> vertices{  
        {1, {2, 3}},  
        {2, {1, 3, 4, 5}},  
        {3, {1, 2, 4}},  
        {4, {2, 4}}  
    };  
  
    Graph g(vertices);  
  
}
```

1	2, 3
2	1, 3, 4, 5
3	1, 2, 4
4	2, 3, 4

Adjacency Matrix

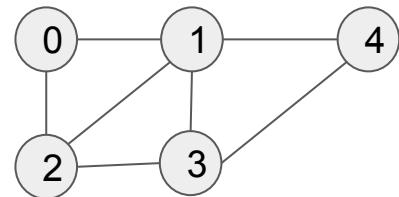


- Directed:
 - $M[i,j] = 1$ If $(i,j) \in E$
 - $M[i,j] = 0$ otherwise
- For undirected graphs, edges are considered to be bidirectional
 - $M[i,j] = M[j,i] = 1$ If $\{i,j\} \in E$
 - $M[i,j] = 0$ otherwise

What is the memory cost?

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	1	0
3	0	1	1	0	1
4	0	1	0	1	0

Adjacency Matrix: Vector of Vector

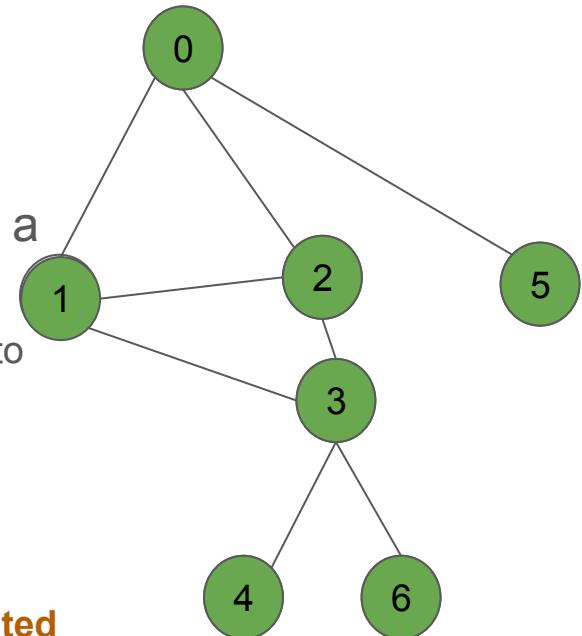


```
class Graph {  
public:  
    Graph(std::vector<std::vector<int>> & adjacency) : adjacency_(adjacency) {}  
    std::vector<std::vector<int>> adjacency_;  
};  
int main() {  
    std::vector<std::vector<int>> adjacency = {{0, 1, 1, 0, 0},  
                                                {1, 0, 1, 1, 1},  
                                                {1, 1, 0, 1, 0},  
                                                {0, 1, 1, 0, 1},  
                                                {0, 1, 0, 1, 0}};  
    Graph g(adjacency);  
}
```

Graph Algorithms

Depth First Search

- An algorithm to **iterate** or **search** through vertices in a directed graph
 - We already know that an undirected graph can be converted to bidirectional directed graph
- Algorithm Overview:
 - Start from an arbitrary vertex v (call it root)
 - **Mark v as visited**
 - **Recursively visit each neighbor of v that is not marked visited**



Observations:

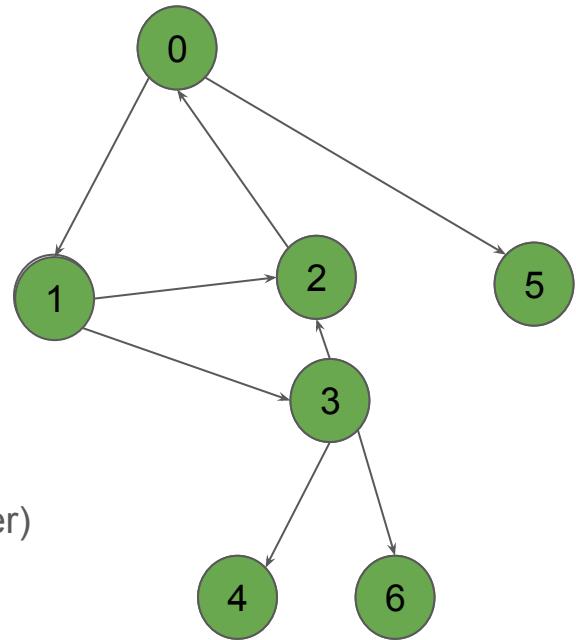
- We start from an empty set of marked nodes and keep adding to it.
- We visit **grandchildren** of the **first child** of v earlier than other v 's children, hence the name!
- We **mark** each node only **once**. We **use each edge only once**.

DFS

```
class MapSetGraph {  
  
public:  
  
    void DFS_helper(int root, std::map<int, int> &marks);  
  
    std::map<int, std::set<int>> edge_map_;  
};  
  
  
void MapSetGraph::DFS_helper(int root, std::map<int, int> &marks) {  
    marks[root] = 1;  
  
    std::cout << "visited: " << root << std::endl;  
  
    for (const int child : edge_map_[root]) {  
        if (marks[child] != 1) {  
            DFS_helper(child, marks);  
        }  
    }  
}
```

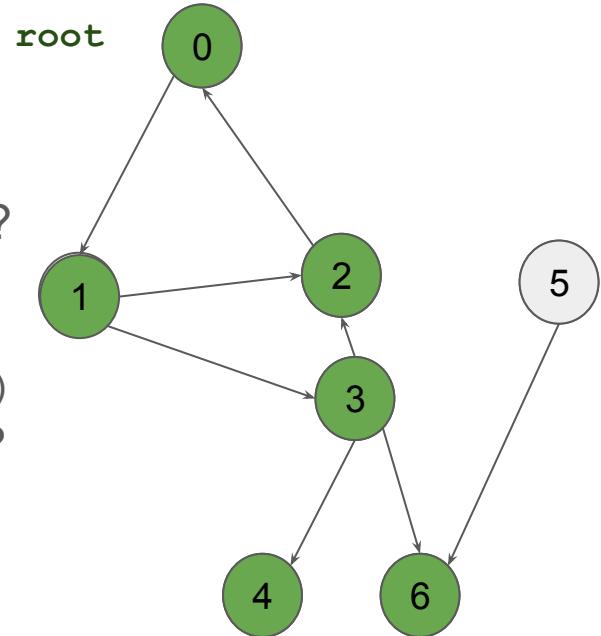
DFS, More Observations

- Runtime is: $m + n$
- Can we get stuck in a cycle?
 - Marking saves us
- It's a recursive algorithm
 - Can be converted to non-recursive (homework!)
 - Each time we recurse, is AKA **backtracking** (more on this later)
- Applications:
 - Simply iterating the Graph
 - Find if a node is in the graph
 - Find if there is a path between two vertices
 - Find if there is a cycle in the graph
 - It doesn't find all cycles. It can only detect if there is at least one cycle



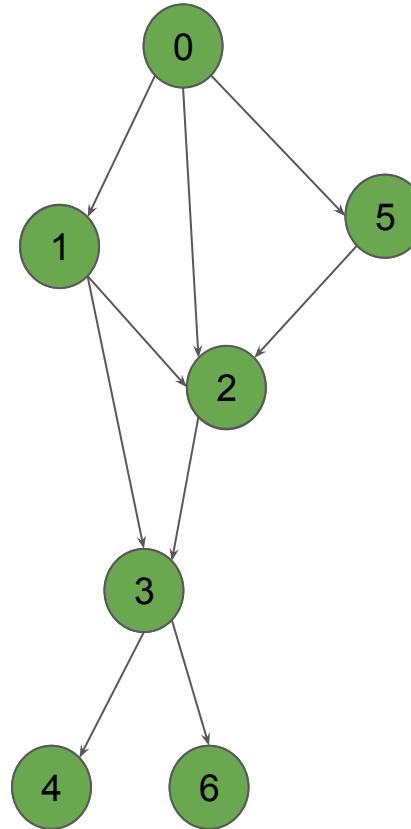
DFS, More Observations

- What if there was no path from root to some nodes?
 - Directed graphs: unconnected
 - Undirected graphs: non-strongly connected
 - Our code can slightly be modified to handle this (homework!)
- Question: what is the maximum stack size for DFS?



DFS for Topological Sort

- Defined In Directed Acyclic Graphs (DAG)
- Usually used to denote dependency of something to a set of other things
 - 0 depends on nothing
 - 1, 2, and 5 depend on 0
 - 3 depends on 1 and 2
 - 4 depends on 3
 - 6 depends on 3



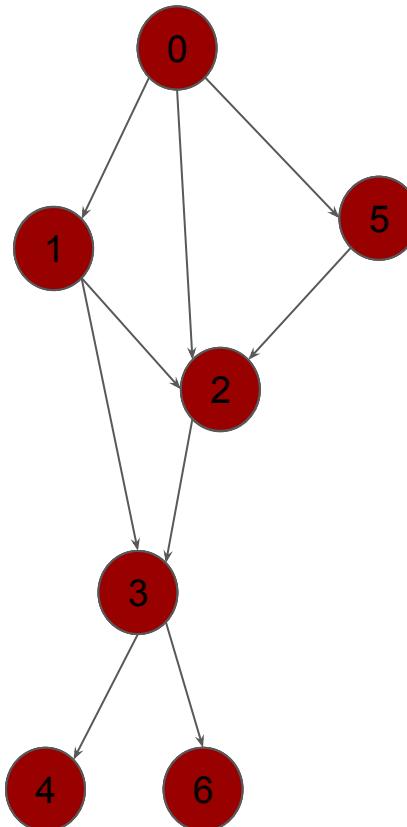
DFS for Topological Sort

- Add vertex to list when we recurse
- The reversed list is topological sort
- We might have multiple valid orders

4 6 3 2 1 5 0

Topological order is: 0, 5, 1, 2, 3, 6, 4

Another topological order is: 0, 1, 2, 5, 3, 6, 4



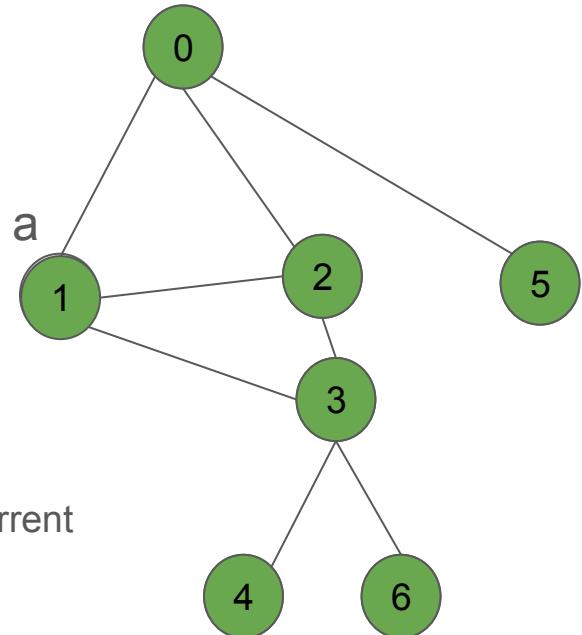
```
void MapSetGraph::DFS_helper_with_topo(int root, std::map<int, int> &marks,
                                         std::vector<int> &topo_list) {
    marks[root] = 1;
    for (const int child : edge_map_[root]) {
        if (marks[child] != 1) {
            DFS_helper_with_topo(child, marks, topo_list);
        }
    }
    topo_list.push_back(root);
}

std::vector<int> MapSetGraph::TopologicalSort(int root) {
    std::vector<int> topo_list;
    std::map<int, int> marks;
    DFS_helper_with_topo(root, marks, topo_list);
    return topo_list;
}
```

Breadth First Search

Breadth First Search

- An algorithm to **iterate** or **search** through vertices in a directed graph
- Algorithm Overview:
 - Start from an arbitrary vertex v (call it root)
 - **Mark v as visited**
 - Visit **each neighbor** of v that is **not marked visited** at the current depth, **before** moving to the next depth.
 - It is different than topological sort!

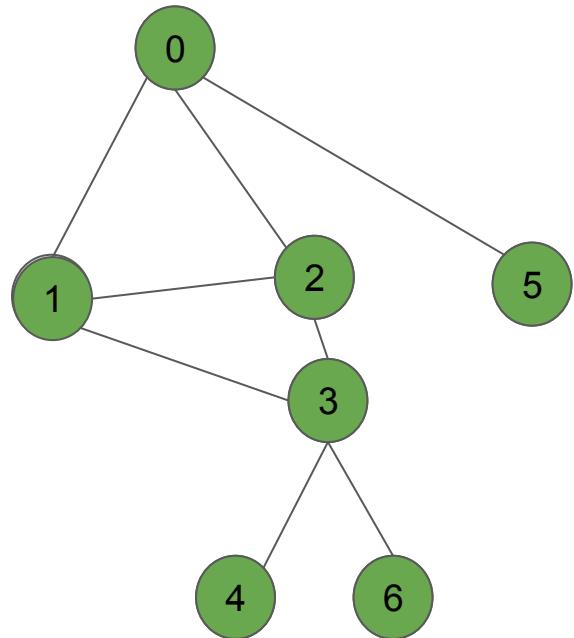


Observations:

- We start from an empty set of marked nodes and keep adding to it.
- We visit **all children** of v **before** moving to v 's **grandchildren**, hence the name! (Different than DFS)
- We **mark** each node only **once**. We **use each edge** only **once**.

BFS, More Observations

- Runtime is: $m + n$
- Can we get stuck in a cycle?
 - Marking saves us (Similar to DFS)
- It's an iterative algorithm
- We find closer vertices first
- What is the maximum size of the queue?
- Applications:
 - Simply iterating the Graph
 - Find if a node is in the graph
 - Find if there is a path between two vertices
 - Find if there is a cycle in the graph
 - It doesn't find all cycles. It can only detect if there is at least one cycle
 - Finds shortest path in unweighted graphs (or when weights are all equal)



EE599: Computing and Software for Systems Engineers

Lecture 7: Graph Search and Shortest Distance Algorithms

University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Announcements

- Midterm:
 - **March 11 from 6:45pm to 8:45pm**
 - Location: Same as our lecture **SOS-B46**
 - Questions may include **everything** that we covered in the **lectures** until that date.
 - The exam is **closed book, closed laptop**, and **no cell phone usage** is allowed.
 - You can bring a **single one sided letter size page** with your own notes and use it during the exam.
 - You will be asked to write reasonable code in C++. Minor syntax errors would be OK.
 - The complexity of the questions will be similar to your homework assignments.
- Academic Conduct (**Please use your own work. It is not worth it!**)
 - **Reminder:** Students are encouraged to collaborate on general solution strategies for homework. The writeup, however, must be your own - you may not copy someone else's solution.
 - In addition, your homework should list all the fellow students with whom you discussed the solutions.

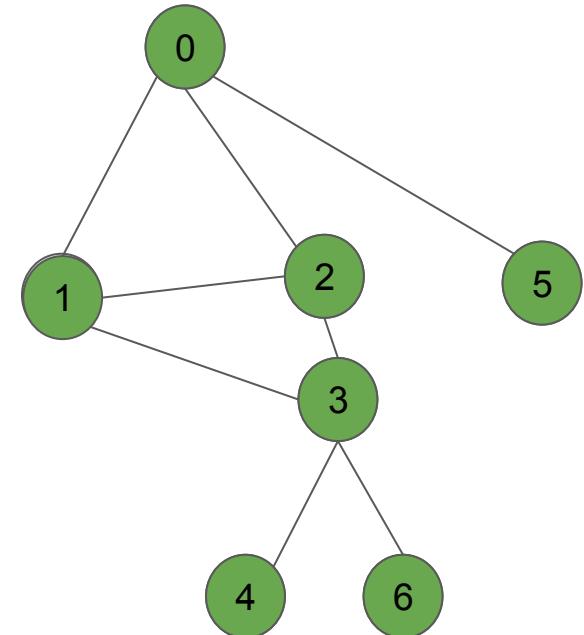
Announcements 2

- Topics that you are expected to know for midterm:
 - References, pointers
 - Pass by reference/value
 - Object oriented design (C++ class, methods, member variables, public/private)
 - STL Containers, their applications and runtime considerations
 - i. E.g. vector, list, map, set, stack, queue, string
 - ii. Iterators and their applications
 - Sort algorithms
 - i. Bubble sort, Selection sort, Merge sort, Heap sort, Quick sort*
 - Data structures:
 - i. Stack, queue, tree, graph, linked list
 - Tree Algorithms:
 - i. Iteration algorithms
 - ii. Binary search tree
 - Graph Algorithms:
 - i. Search and iteration: DFS, BFS, Topological Sort
 - ii. Shortest distance algorithms:
 1. Dijkstra, Bellman-Ford, Floyd-Warshall

Graph Algorithms

Depth First Search

- An algorithm to **iterate** or **search** through vertices in a directed graph
 - We already know that an undirected graph can be converted to bidirectional directed graph
- Algorithm Overview:
 - Start from an arbitrary vertex v (call it root)
 - **Mark v as visited**
 - **Recursively visit each neighbor of v that is not marked visited**



Observations:

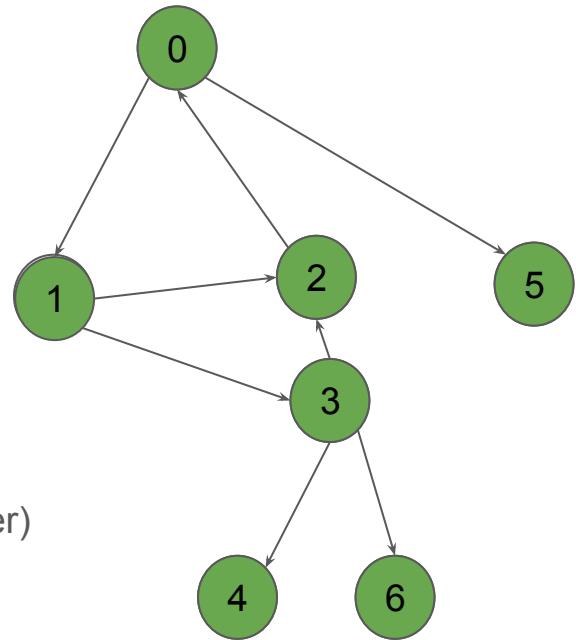
- We start from an empty set of marked nodes and keep adding to it.
- We visit **grandchildren** of the **first child** of v earlier than other v 's children, hence the name!
- We **mark** each node only **once**. We **use each edge** only **once**.

DFS

```
class MapSetGraph {  
  
public:  
  
    void DFS_helper(int root, std::map<int, int> &marks);  
  
    std::map<int, std::set<int>> edge_map_;  
};  
  
  
  
void MapSetGraph::DFS_helper(int root, std::map<int, int> &marks) {  
    marks[root] = 1;  
  
    std::cout << "visited: " << root << std::endl;  
  
    for (const int child : edge_map_[root]) {  
        if (marks[child] != 1) {  
            DFS_helper(child, marks);  
        }  
    }  
}
```

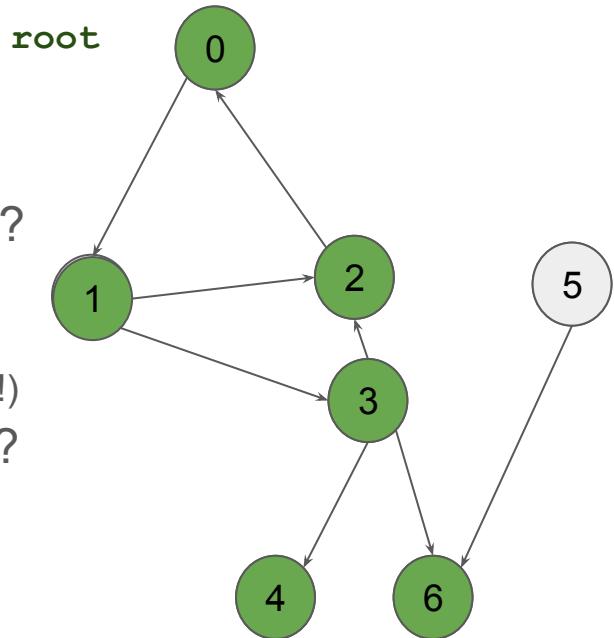
DFS, More Observations

- Runtime is: $m + n$
- Can we get stuck in a cycle?
 - Marking saves us
- It's a recursive algorithm
 - Can be converted to non-recursive (homework!)
 - Each time we recurse, is AKA **backtracking** (more on this later)
- Applications:
 - Simply iterating the Graph
 - Find if a node is in the graph
 - Find if there is a path between two vertices
 - Find if there is a cycle in the graph
 - It doesn't find all cycles. It can only detect if there is at least one cycle



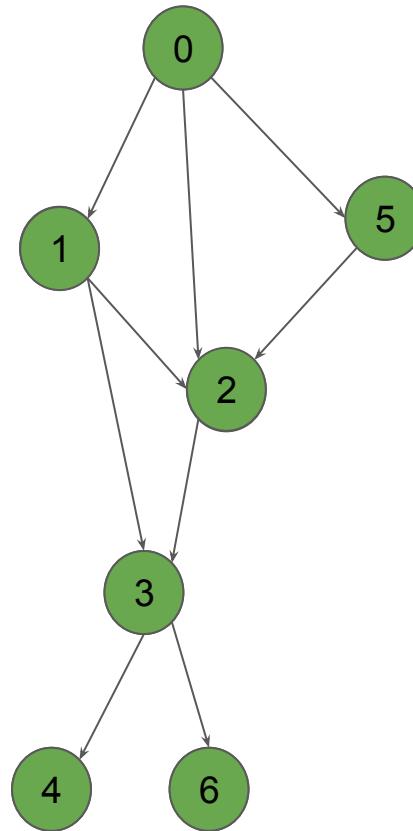
DFS, More Observations

- What if there was no path from root to some nodes?
 - Directed graphs: unconnected
 - Undirected graphs: non-strongly connected
 - Our code can slightly be modified to handle this (homework!)
- Question: what is the maximum stack size for DFS?



DFS for Topological Sort

- Defined In Directed Acyclic Graphs (DAG)
- Usually used to denote dependency of something to a set of other things
 - 0 depends on nothing
 - 1, 5, depend on 0
 - 2 depends on 0, 1, 5
 - 3 depends on 1 and 2
 - 4 depends on 3
 - 6 depends on 3



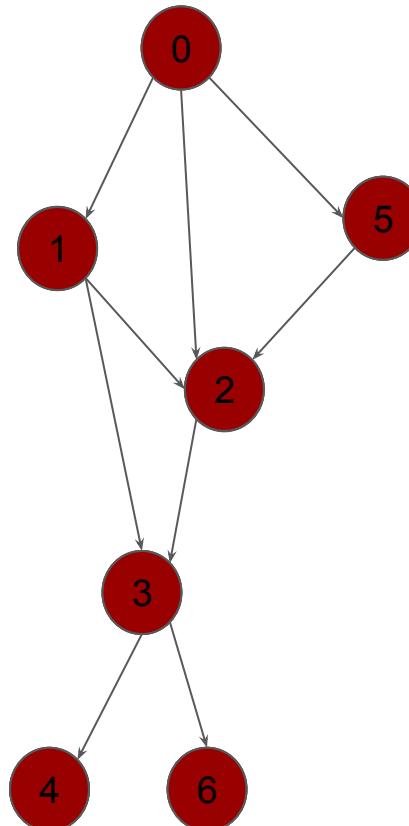
DFS for Topological Sort

- Add vertex to list when we recurse
- The reversed list is topological sort
- We might have multiple valid orders

4 6 3 2 1 5 0

Topological order is: 0, 5, 1, 2, 3, 6, 4

Another topological order is: 0, 1, 2, 5, 3, 6, 4



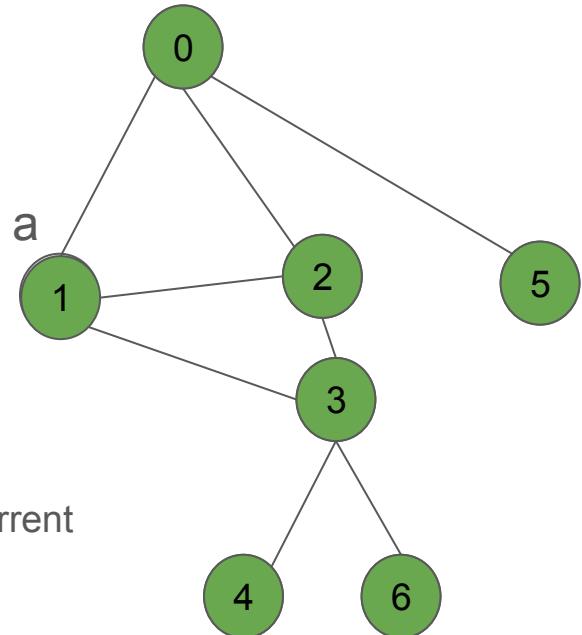
```
void MapSetGraph::DFS_helper_with_topo(int root, std::map<int, int> &marks,
                                         std::vector<int> &topo_list) {
    marks[root] = 1;
    for (const int child : edge_map_[root]) {
        if (marks[child] != 1) {
            DFS_helper_with_topo(child, marks, topo_list);
        }
    }
    topo_list.push_back(root);
}

std::vector<int> MapSetGraph::TopologicalSort(int root) {
    std::vector<int> topo_list;
    std::map<int, int> marks;
    DFS_helper_with_topo(root, marks, topo_list);
    return topo_list;
}
```

Breadth First Search

Breadth First Search

- An algorithm to **iterate** or **search** through vertices in a directed graph
- Algorithm Overview:
 - Start from an arbitrary vertex v (call it root)
 - **Mark v as visited**
 - Visit **each neighbor** of v that is **not marked visited** at the current depth, **before** moving to the next depth.
 - It is different than topological sort!



Observations:

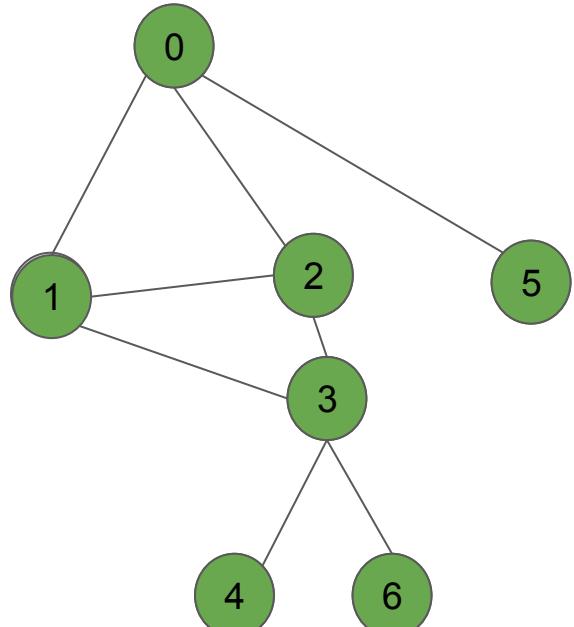
- We start from an empty set of marked nodes and keep adding to it.
- We visit **all children** of v **before** moving to v 's **grandchildren**, hence the name! (Different than DFS)
- We **mark** each node only **once**. We **use each edge** only **once**.

BFS

- Algorithm Overview:
 - Start from an arbitrary vertex v (call it root)
 - Mark v as visited
 - Visit each neighbor of v that is not marked visited at the current depth, before moving to the next depth.
 - It is different than topological sort!
- Use a Queue
 - Track which child should visit first

```
void MapSetGraph::BFS(int root) {  
  
    std::map<int, int> marks;  
  
    std::queue<int> q;  
  
    q.push(root);  
  
    marks[root] = 1;  
  
    while (!q.empty()) {  
  
        int cur = q.front();  
  
        q.pop();  
  
        for (auto &n : edge_map_[cur]) {  
  
            if (!marks[n]) {  
  
                marks[n] = 1;  
  
                q.push(n);  
            }  
        }  
    }  
}
```

Breadth First Search



Marked



0	1	2	5
---	---	---	---

0	1	2	5	3
---	---	---	---	---

0	1	2	5	3
---	---	---	---	---

0	1	2	5	3
---	---	---	---	---

0 1 2 5 3 4 6

0	1	2	5	3	4	6
---	---	---	---	---	---	---

Queue



5 2 1

3 5 2

3 5

3

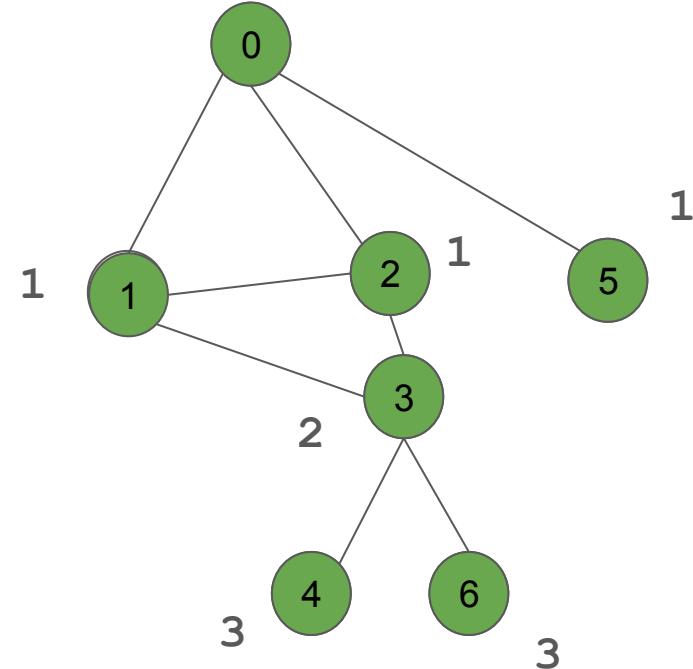
6 4

6

Breadth First Search

Observations:

- When we visit **root's children**, we are discovering all nodes at **distance 1**.
- When we visit children of a node at **distance i**, we discover all nodes at **distance i+1**.



In a graph where all edges have **equal weights**, BFS finds the **shortest distance!**

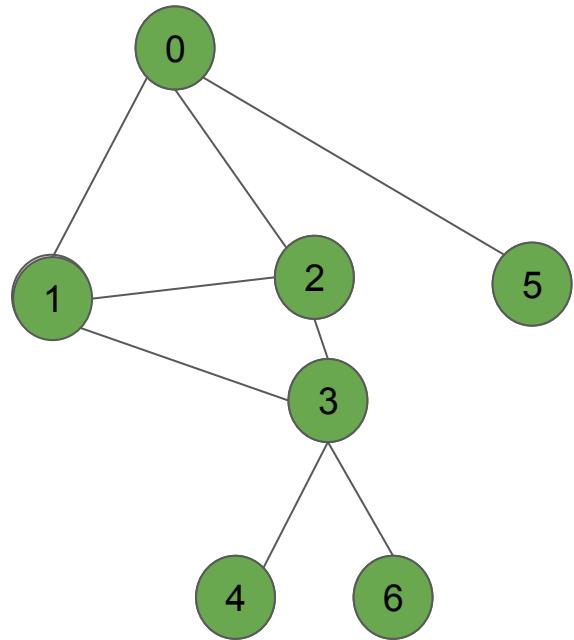
Homework

- 1. Enhance this code to calculate the **shortest distance** for each node
- 2. Enhance this code to find the actual **shortest path** from root to each node.

```
void MapSetGraph::BFS(int root) {  
    std::map<int, int> marks;  
    std::queue<int> q;  
    q.push(root);  
    marks[root] = 1;  
    while (!q.empty()) {  
        int cur = q.front();  
        q.pop();  
        for (auto &n : edge_map_[cur]) {  
            if (!marks[n]) {  
                marks[n] = 1;  
                q.push(n);  
            }  
        }  
    }  
}
```

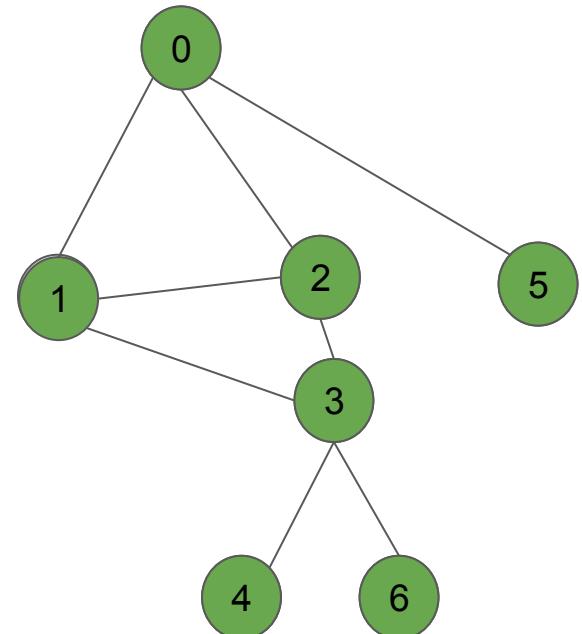
BFS, More Observations

- Runtime is: $m + n$
- Can we get stuck in a cycle?
 - Marking saves us (Similar to DFS)
- It's an iterative algorithm
- We find closer vertices first
- What is the maximum size of the queue?
- Applications:
 - Simply iterating the Graph
 - Find if a node is in the graph
 - Find if there is a path between two vertices
 - Finds shortest path in unweighted graphs (or when weights are all equal)



DFS VS BFS

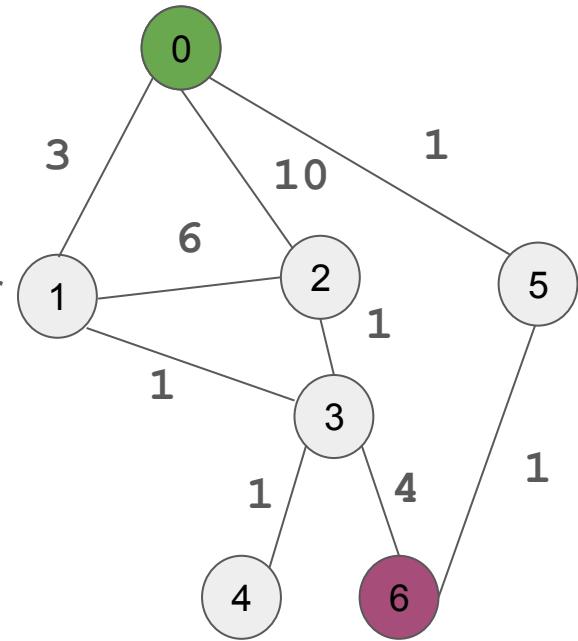
	BFS	DFS
Data Structure	Queue	Stack
Runtime	$O(m+n)$	$O(m+n)$
General usage	Can be used for search and iteration	Can be used for search and iteration
Special Application	Can find single source shortest path	Can be used for topological sort and detecting cycles
Memory size	Max memory: queue size, i.e. max numbers in a level	Max memory size: stack size, i.e. the maximum distance between root and other nodes



Shortest Path Algorithms

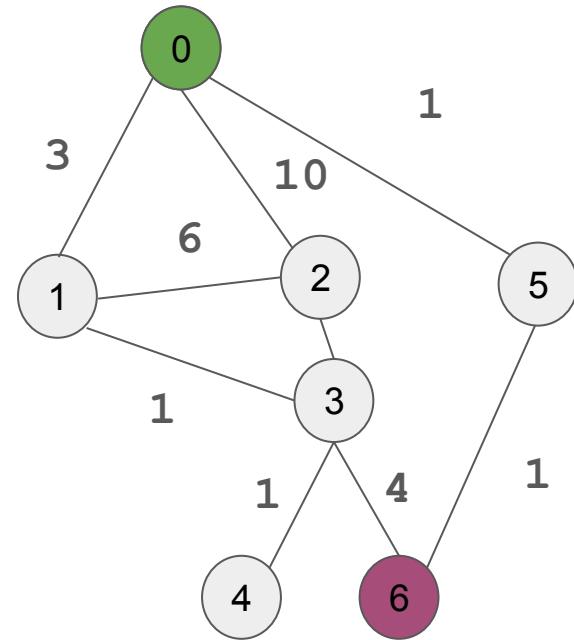
Weighted Graph

- **Weighted Graph:**
 - For each edge e in E we associated a real number $w(e)$, called its weight.
- **Unweighted Graph:**
 - Can be converted to a weighted graph with all weights equal to 1.



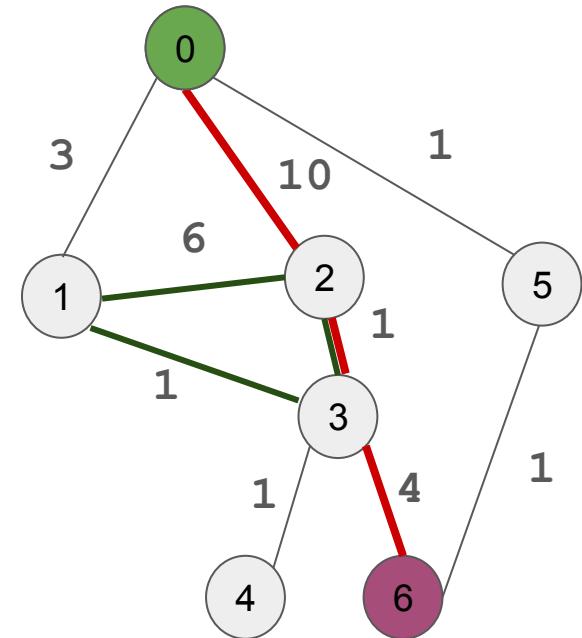
Shortest Path/Distance Algorithms

- **The Shortest Distance problem (SDP):** find the shortest **distance** between nodes.
- **The Shortest Path Problem (SPP):** find the actual shortest **path** between nodes.
 - Usually we can find path with minor modifications in SDP



Path and Cycles

- A Path:
 - A sequence of adjacent vertices
 - $P=(v_1, v_2, \dots, v_n)$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$.
- A Cycle: A path where $v_1 = v_n$
- A **Simple** Path / Cycle: A Path with no repeated vertices
 - Except for $v_1 = v_n$ in a simple cycle

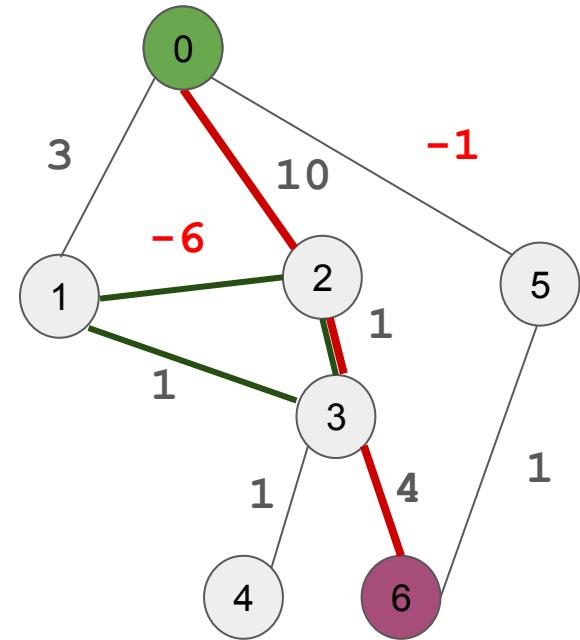


Remember:

- Shortest Distance/Path problem is defined for a graph that has both **simple AND non-simple**

Negative Edge and Cycles

- Shortest path is **meaningless** if there is a negative **cycle**
 - More precisely, a negative cycle is reachable from source
- We also ignore the case where there is a zero cycle
- But we can still define shortest path if there are negative **edges**

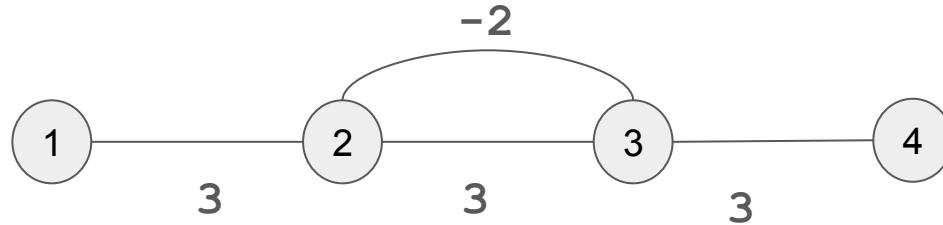


Remember:

- Negative **edges** (mostly) OK, negative **cycles**, not OK!

Negative Edge and Cycles

- If there is no negative cycles:
 - Shortest path problem is really shortest **simple** path problem
- Max length of a simple path in a graph = $n - 1$



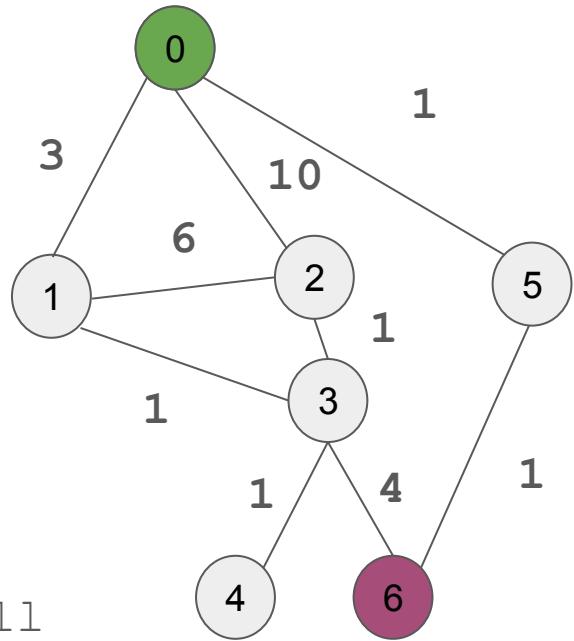
Remember:

- If there is no negative cycles, there is no point to include a cycle in the shortest path.

Algorithms We Cover



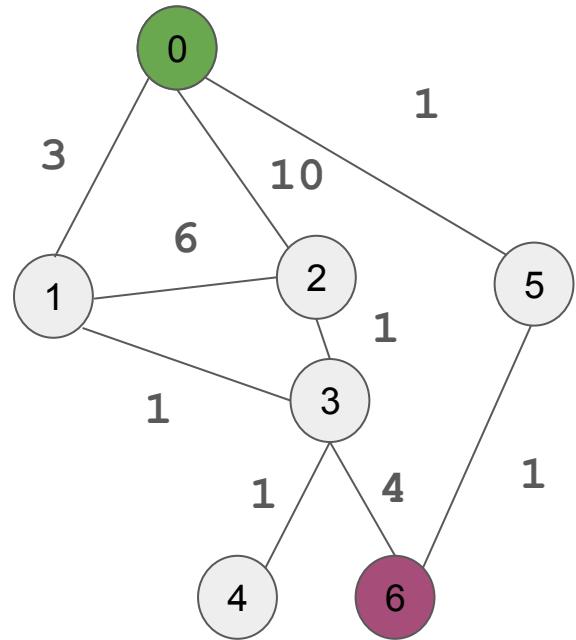
- Dijkstra
- Bellman-Ford
- Floyd-Warshall



Floyd-Warshall

- Given we are doing **All Pair** analysis, what is a good data structure?

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	∞	∞
2	10	6	0	1	∞	∞	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	∞	∞	∞	∞	0	1
6	∞	∞	∞	4	∞	1	0



We set the distance of each node
to itself to 0

Floyd-Warshall

- Intuition:
 - Find shortest path from node i to j using **first k nodes**.
 - Recursively update

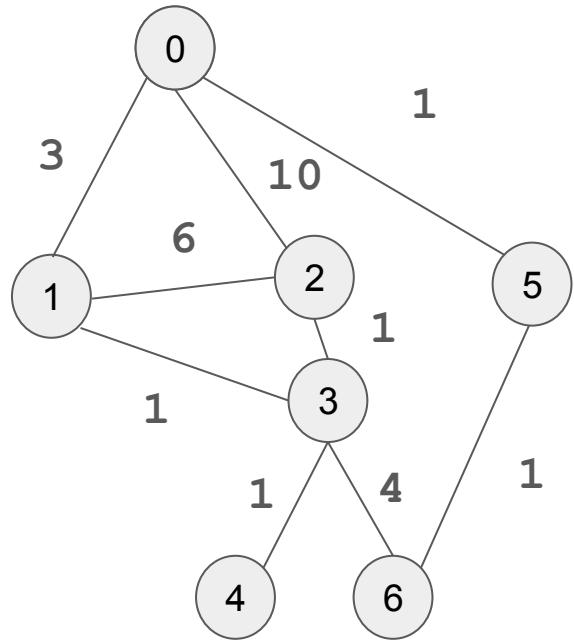
Base Case: $\text{shortestPath}(i, j, 0) = w(i, j)$

Recursive Case: $\text{shortestPath}(i, j, k) =$

$\text{Min}(\text{shortestPath}(i, j, k-1),$

$\text{shortestPath}(i, k, k-1) + \text{shortestPath}(k, j, k-1)$

)



Floyd-Warshall

Base Case: $\text{shortestPath}(i, j, 0) = w(i, j)$

Recursive Case: $\text{shortestPath}(i, j, k) =$

Min($\text{shortestPath}(i, j, k-1)$,

$\text{shortestPath}(i, k, k-1) + \text{shortestPath}(k, j, k-1)$

)

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	∞	∞
2	10	6	0	1	∞	∞	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	∞	∞	∞	∞	0	1
6	∞	∞	∞	4	∞	1	0

$k=0$

(use no nodes)

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	4	∞
2	10	6	0	1	∞	11	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	∞	4	11	∞	∞	1
6	∞	∞	∞	4	∞	1	0

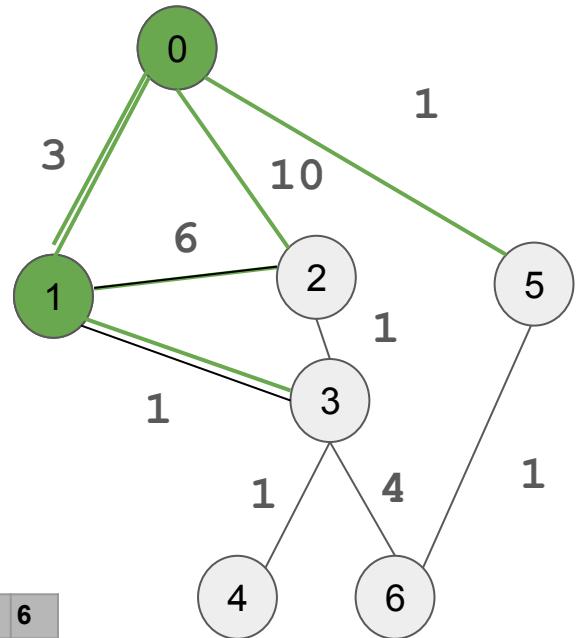
$k=1$

(use node 0)

	0	1	2	3	4	5	6
0	0	3	9	4	∞	1	∞
1	3	0	6	1	∞	4	∞
2	9	6	0	1	∞	10	∞
3	4	1	1	0	1	5	4
4	∞	∞	∞	1	0	∞	∞
5	1	4	10	5	∞	0	1
6	∞	∞	∞	4	∞	1	0

$k=2$

(use nodes {0, 1})



Floyd-Warshall

```
// Calculates shortest distance between i,j using nodes 0 to k

long DistMatrixGraph::FloydWarshallRecursiveHelper(int i, int j, int k) {

    long distance;

    if (k < 0) {

        return weight_[i][j];

    } else {

        return std::min(FloydWarshallRecursiveHelper(i, j, k - 1) ,
                        FloydWarshallRecursiveHelper(i, k, k - 1) +
                        FloydWarshallRecursiveHelper(k, j, k - 1));
    }
}
```

Floyd-Warshall (Recursive)

```
std::vector<std::vector<long>> DistMatrixGraph::FloydWarshallRecursive() {
    std::vector<std::vector<long>> d(weight_.size(),
                                         std::vector<long>(weight_.size()));

    // Find all pair distances
    for (int i = 0; i < weight_.size(); ++i) {
        for (int j = 0; j < weight_.size(); ++j) {
            d[i][j] = FloydWarshallRecursiveHelper(i, j, weight_.size() - 1);
        }
    }

    return d;
}
```

Floyd-Warshall (Non-Recursive)

- Non-Recursive Version of F.W. is an instance of
 - **Dynamic Programming (Tabulation Method)**

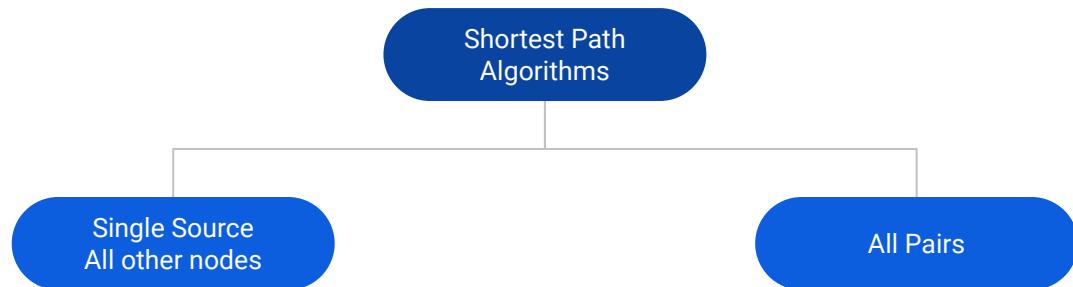
```
// To cover the base case (using no nodes)
// d is previously initialized to weights_
for (int k = 0; k < weight_.size(); k++) {
    for (int i = 0; i < weight_.size(); ++i) {
        for (int j = 0; j < weight_.size(); ++j) {
            d[i][j] = std::min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

Floyd-Warshall (Negative Cycles)

- Remember that we set the distance of each node to itself to 0.
- As we iterate through k, what would happen to the distance of a node to itself if there exists a negative cycle? (homework)

	0	1	2	3	4	5	6
0	0	3	10	∞	∞	1	∞
1	3	0	6	1	∞	∞	∞
2	10	6	0	1	∞	∞	∞
3	∞	1	1	0	1	∞	4
4	∞	∞	∞	1	0	∞	∞
5	1	∞	∞	∞	∞	0	1
6	∞	∞	∞	4	∞	1	0

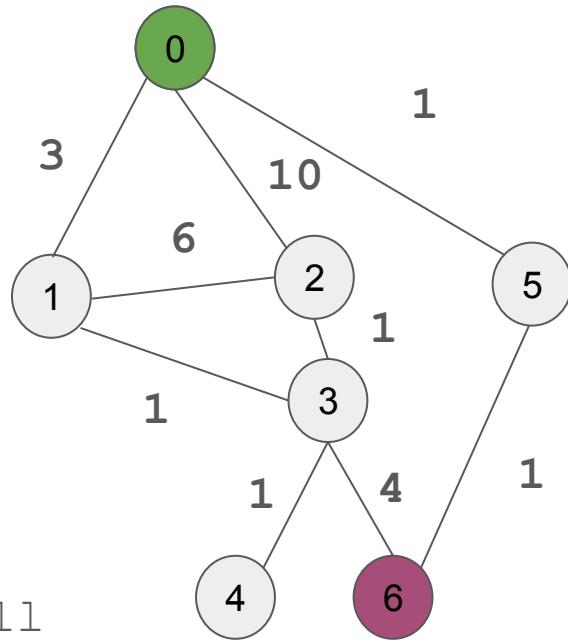
More About Floyd-Warshall



- Dijkstra
- Bellman-Ford

- Floyd-Warshall

CAN handle negative **edges**
CAN report negative **cycles**
Runtime: $O(n^3)$

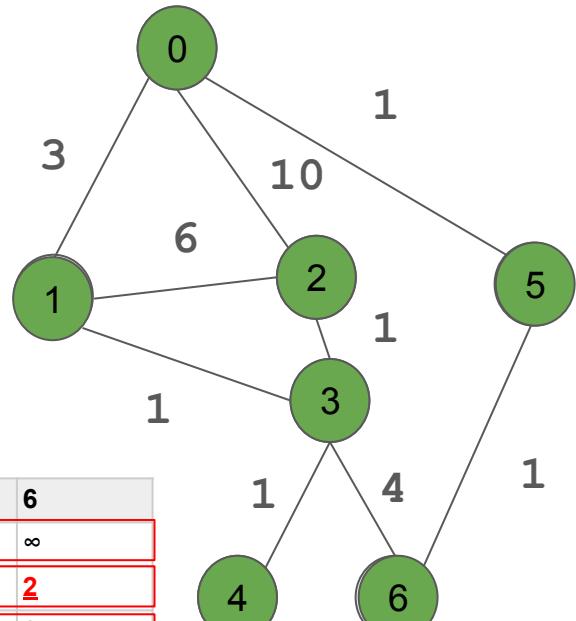


Dijkstra

- Intuition: Grow a list of visited nodes:

- Find the unvisited node u with shortest distance using only visited nodes
- Relax the distance of neighbors of u
- Add u to the visited list and repeat

	0	1	2	3	4	5	6
u=5 Visited={0}	0	3	10	∞	∞	1	∞
u=6 Visited={0,5}	0	3	10	∞	∞	4	2
u=1 Visited={0,5,6}	0	3	10	6	∞	4	2
u=3 Visited={0,5,6,1}	0	3	9	4	∞	4	2
u=2 Visited={0,5,6,1,3}	0	3	5	4	5	4	2
u=4 Visited={0,5,6,1,3,2}	0	3	5	4	5	4	2
Visited={0,5,6,1,3,2,4}	0	3	5	4	5	4	2



Dijkstra

```
std::vector<long> DistMatrixGraph::Dijkstra(int source) {
    std::unordered_set<int> visited;
    std::vector<long> d(weight_.size());
    for (int i = 0; i < weight_.size(); i++) {
        d[i] = weight_[source][i];
    }
    visited.insert(source);
    while (visited.size() < weight_.size()) {
        int u = FindMinInDButNotInVisited(d, visited);
        visited.insert(u);
        for (int i = 0; i < weight_.size(); i++) {
            d[i] = std::min(d[i], d[u] + weight_[u][i]);
        }
    }
    return d;
}
```

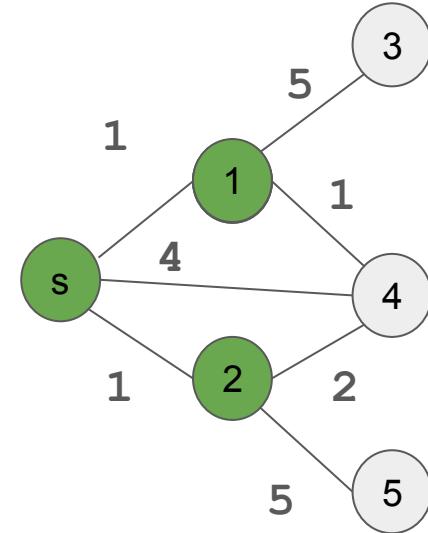
Dijkstra: Another Example

- **Intuition:** Grow a list of visited nodes:
 - Find the unvisited node u with shortest distance
 - Relax the distance of neighbors of u
 - Add u to the visited list and repeat

```
int u = FindMinInDButNotInVisited(d, visited);
```

At this moment in time,
which node do we select for u ?

		s	1	2	3	4	5
	
u=?	Visited={0, 2}	θ	1	4	∞	3	6
	Visited={0, 2, 1}	0	1	1	6	2	6



Dijkstra: Negative edges

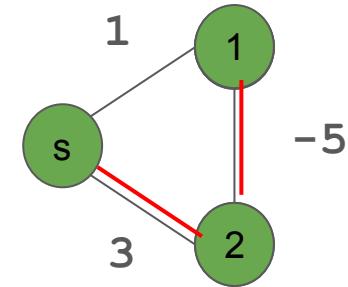
- **Greedy Approach:**

- Dijkstra visits the closest node and then locks it inside visited nodes.

```
int u = FindMinInDButNotInVisited(d, visited);
```

At this moment in time,
which node do we select for u?

		s	1	2
	
u=?	Visited={0}	0	1	3
	Visited={0, 1}	0	-2	-4



Shortest path to 1 is through 2, not 1

More About Dijkstra

Shortest Path
Algorithms

Single Source
All other nodes

- Dijkstra

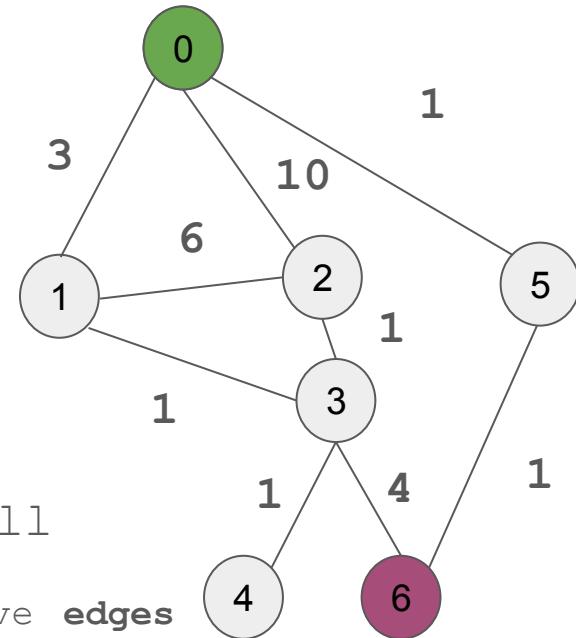
CANNOT handle negative **edges**

CANNOT handle negative **cycles**

$O(n^2)$ OR $O(m + n \log(n))$ with heap

- Bellman-Ford

All Pairs



- Floyd-Warshall

CAN handle negative **edges**

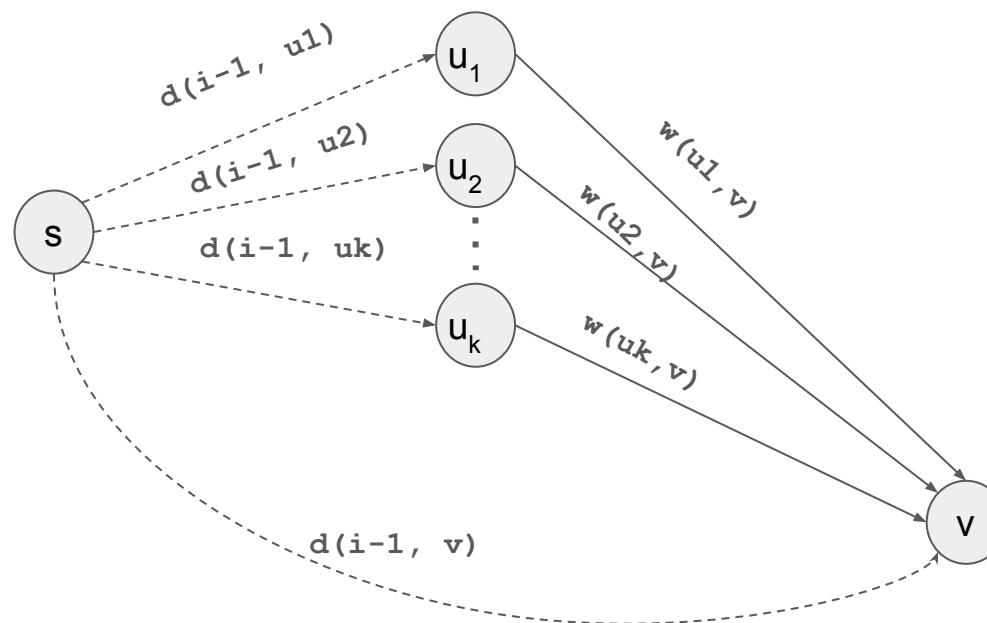
CAN report negative **cycles**

Runtime: $O(n^3)$

Bellman-Ford (Observation)

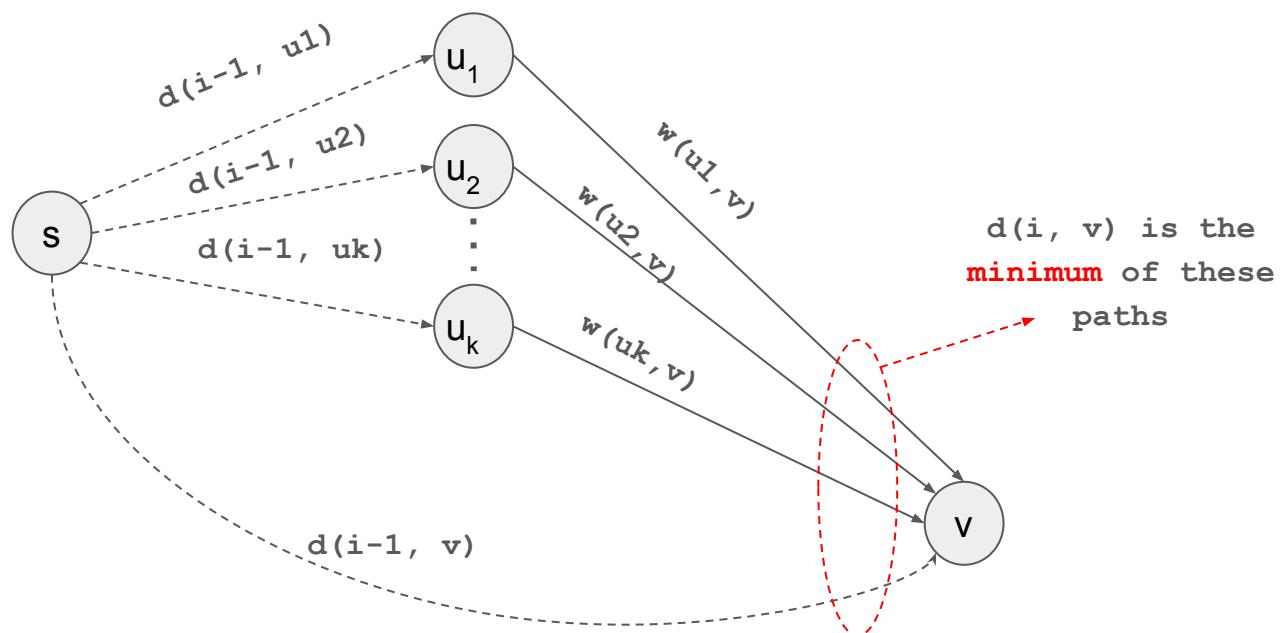
- Intuition:

- Let $d(i, v)$ be the length of the shortest s to v path whose length is at most i
- Let's write $d(i, v)$ recursively

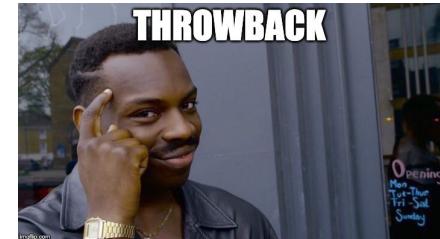


- **Intuition:**

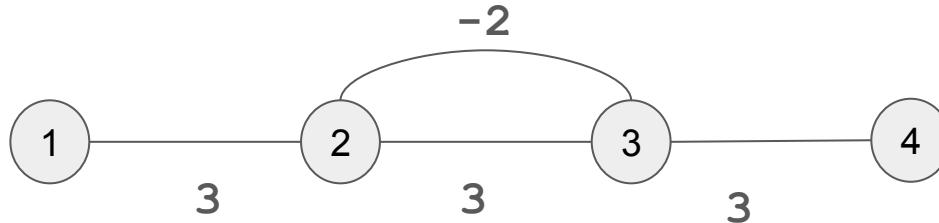
- Let $d(v, i)$ be the length of the shortest s to v path whose length is at most i
- Shortest distance from u_1 to v: $d(i - 1, u_1) + w(v, u_1)$
- Shortest distance from s to v using $i - 1$ edges: $d(i - 1, v)$



Negative Edge and Cycles



- If there is no negative cycles:
 - Shortest path problem is really shortest **simple** path problem
- Max length of a simple path in a graph = $n - 1$



Remember:

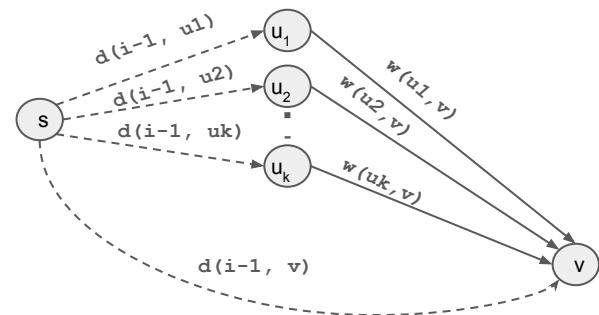
- If there is no negative cycles, shortest path is at most of size $n - 1$

Bellman-Ford (Recursive)

- **Intuition:**

- Let $d(v, i)$ be the length of the shortest s to v path whose length is at most i
- We want to find $d(v, n-1)$

$$d(i, v) = \begin{cases} \infty ; & \text{if } i=0, v=s \\ 0 ; & \text{(i.e. } s \text{ to itself)} \\ \min(d(i-1, v), \\ \min(d(i-1, u) + w(u, v)) ; & \text{if } i=0, v \neq s \\ (\text{For } (u, v) \text{ in } E) & \text{(i.e. } s \text{ to other nodes with} \\ & \text{path of size 0)} \\ & \text{Otherwise} \end{cases}$$

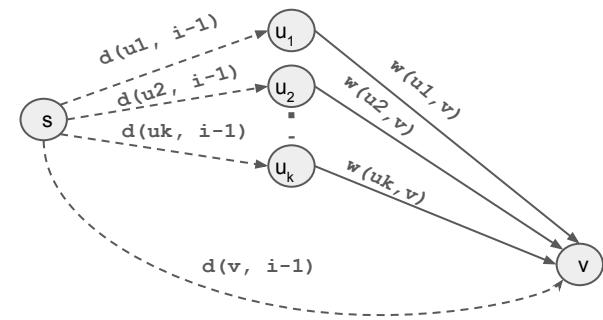


Bellman-Ford (Recursive)

```
long DistMatrixGraph::BellmanFordRecursiveHelper(int s, int i, int v) {  
    if (i == 0 && v == s) {  
        return 0;  
    } else if (i == 0 && v != s) {  
        return INT_MAX;  
    } else {  
        long d = INT_MAX;  
        for (int u = 0; u < weight_.size(); u++) {  
            d = std::min(d, BellmanFordRecursiveHelper(s, i - 1, u) + weight_[u][v]);  
        }  
        return std::min(BellmanFordRecursiveHelper(s, i - 1, v), d);  
    }  
}
```

Bellman-Ford (Iterative)

- **Intuition:**
 - Let $d(v, i)$ be the length of the shortest s to v path whose length is at most i
 - We want to find $d(v, n-1)$



Idea:

- What if we create a 2D table d , for all $v \in V$ and $i = 0$ to $n-1$

$$d(v, i) = \min(d(v, i-1), \min(d(u, i-1) + w(u, v)) ;$$

(For (u, v) in E)

Bellman-Ford (2D non-Recursive)

```
std::vector<long> DistMatrixGraph::BellmanFord2D(int source) {
    std::vector<std::vector<long>> d(weight_.size() - 1,
                                         std::vector<long>(weight_.size()));
    for (int i = 0; i < weight_.size() - 1; i++) {
        for (int v = 0; v < weight_.size(); v++) {
            // Base case not shown
            long e = INT_MAX;
            for (int u = 0; u < weight_.size(); u++) {
                e = std::min(e, d[i - 1][u] + weight_[u][v]);
            }
            d[i][v] = std::min(d[i - 1][v], e);
        }
    }
    return d[weight_.size() - 2];
}
```

Bellman-Ford (Runtime)

- 2 for loops: n^2 * (some work in the third one)
- In the 2nd and 3rd loop we are only checking incoming edges
 - Therefore: $n * (\text{Sum(all incoming edges)}) = n * m$

```
for (int i = 0; i < weight_.size() - 1; i++) {  
    for (int v = 0; v < weight_.size(); v++) {  
        // Base case not shown  
        long e = INT_MAX;  
        for (int u = 0; u < weight_.size(); u++) {  
            e = std::min(e, d[i - 1][u] + weight_[u][v]);  
        }  
        d[i][v] = std::min(d[i - 1][v], e);  
    }  
}
```

Bellman-Ford

Shortest Path
Algorithms

Single Source
All other nodes

- Dijkstra

CANNOT handle negative edges

CANNOT handle negative cycles

$O(n^2)$ OR $O(m + n \log(n))$ with heap

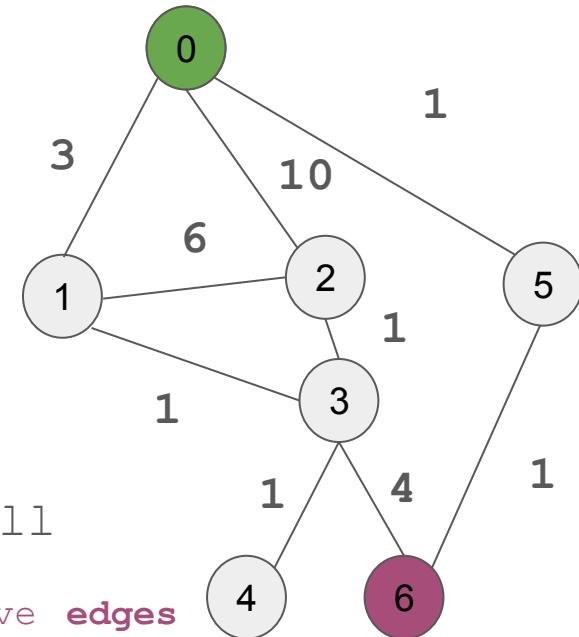
- Bellman-Ford

CAN handle negative edges

CAN report negative cycles

Runtime: $O(m * n)$

All Pairs



- Floyd-Warshall

CAN handle negative edges

CAN report negative cycles

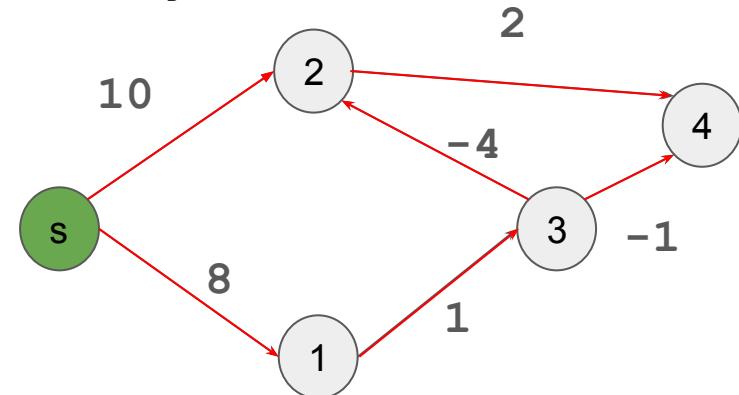
Runtime: $O(n^3)$

Bellman-Ford Optimization to 1D Table

Bellman-Ford (Iterative)

- Intuition:
 - Initialize all distances to ∞
 - Iterate $e = (u,v)$ on all edges:
 - Relax the distances:
 - $d[v] = \min(d[v], \min(d[u] + \text{weight}[u][v]))$;

	s	1	2	3	4
Using {}	0	∞	∞	∞	∞
Using {(2,4)}	0	∞	∞	∞	∞
Using {(2,4), (3,4)}	0	∞	∞	∞	∞
Using {(2,4), (3,4), (3,2)}	0	∞	∞	∞	∞
Using {(2,4), (3,4), (3,2), (1,3)}	0	∞	∞	∞	∞
Using {(2,4), (3,4), (3,2), (1,3), (s,1)}	0	8	∞	∞	∞
Using {(2,4), (3,4), (3,2), (1,3), (s,1), (s,2)}	0	∞	10	∞	∞



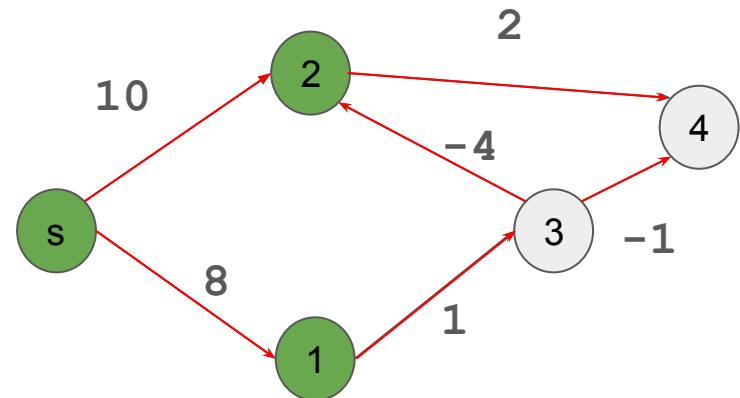
It's now guaranteed that all shortest distances of length **1** were found

Bellman-Ford (Iteration 2)

- **Intuition:**

- Initialize all distances to ∞
- Find all shortest paths of length i:
 - Iterate $e = (u,v)$ on all edges:
 - Relax the distances:
 - $d[v] = \min(d[v], d[u] + \text{weight}[u][v]);$

	s	1	2	3	4
Using {}	0	8	10	∞	∞
Using {(2,4)}	0	8	10	∞	12
Using {(2,4), (3,4)}	0	8	10	∞	12
Using {(2,4), (3,4), (3,2)}	0	8	10	∞	12
Using {(2,4), (3,4), (3,2), (1,3)}	0	8	10	9	12
Using {(2,4), (3,4), (3,2), (1,3), (s,1)}	0	8	10	9	12
Using {(2,4), (3,4), (3,2), (1,3), (s,1), (s,2)}	0	8	10	9	12

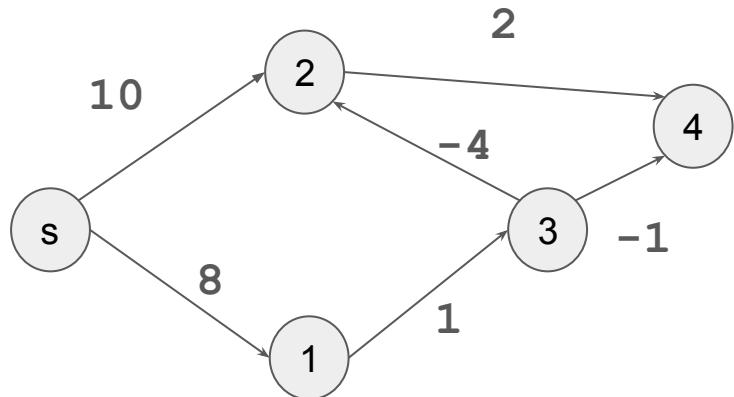


It's now guaranteed that all shortest
distances of length 2 were found

Bellman-Ford (Iteration n)

- **Intuition:**
 - Initialize all distances to ∞
 - **Repeat $n-1$ times:**
 - For $e = (u,v)$ on all edges:
 - Relax the distances:
 - $d[v] = \min(d[v], d[u] + \text{weight}[u][v]);$

Each shortest path is at most **size n**,
therefore, we repeat **n** times

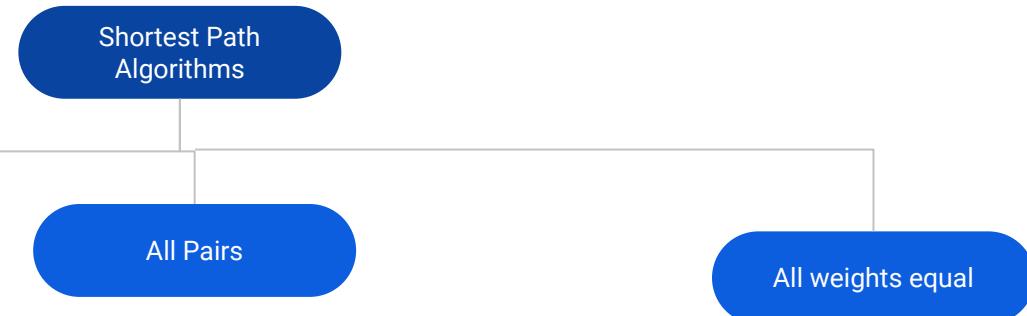


Bellman-Ford

- First loop: Repeat $n - 1$ times
- 2nd and 3rd loops: iterating edges

```
std::vector<long> DistMatrixGraph::BellmanFord(int source) {  
    std::vector<long> d(weight_.size(), INT_MAX);  
  
    d[source] = 0;  
  
    for (int i = 0; i < weight_.size() - 1; i++) {  
  
        for (int u = 0; u < weight_.size(); u++) {  
  
            for (int v = 0; v < weight_.size(); v++) {  
  
                d[v] = std::min(d[v], d[u] + weight_[u][v]);  
            }  
        }  
    }  
  
    return d;
```

Summary



- Dijkstra

CANNOT handle negative **edges**

CANNOT handle negative **cycles**

$O(n^2)$ OR $O((m+n)\log(n))$ with heap

- Bellman-Ford

CAN handle negative **edges**

CAN report negative **cycles**

Runtime: $O(m*n)$

- Floyd-Warshall

CAN handle negative **edges**

CAN report negative **cycles**

Runtime: $O(n^3)$

- BFS

CANNOT handle negative **edges**

CANNOT report negative **cycles**

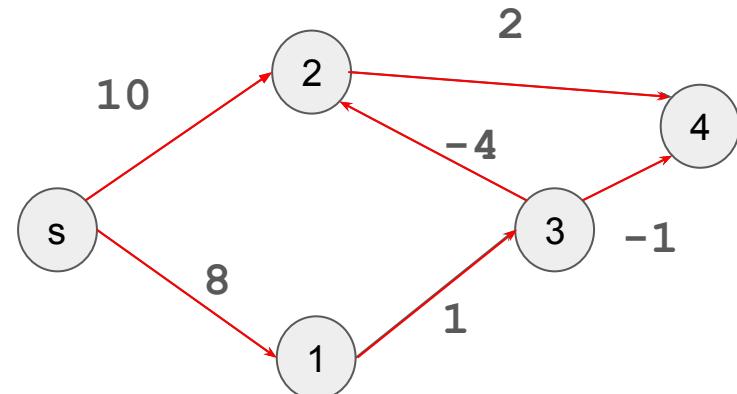
Runtime: $O(m+n)$

Bellman-Ford

- **Intuition:**

- Initialize all distances to ∞
- Iterate $e = (u,v)$ edges:
 - Relax the distances:
 - $d[v] = \min(d[v], d[u] + \text{weight}[u][v]);$

	s	1	2	3	4
Using {}	0	∞	∞	∞	∞
Using {(s,1)}	0	8	∞	∞	∞
Using {(s,1), (s,2)}	0	8	10	∞	∞
Using {(s,1), (s,2), (3,2)}	0	8	10	∞	∞
Using {(s,1), (s,2), (3,2), (3,4)}	0	8	10	∞	∞
Using {(s,1), (s,2), (3,2), (3,4), (2,4)}	0	8	10	∞	12
Using {(s,1), (s,2), (3,2), (3,4), (2,4),(1,3)}	0	8	10	9	12



EE599: Computing and Software for Systems Engineers

Quarantine Edition!

Lecture 9: NodeJS, a Closer Look

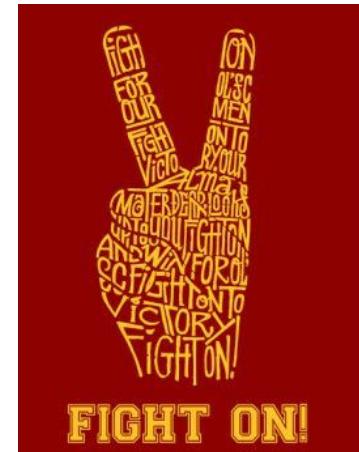
University of Southern California
Spring 2020
Instructor: Arash Saifhashemi

Welcome back!

- **Fight On, Literally!**
- **Office Hours:**
 - If you need help for your project use our office hours. You can set virtual meeting with me almost any time now!
- Our lectures are now very interactive:
 - Please have VSC ready



Corona-Virus

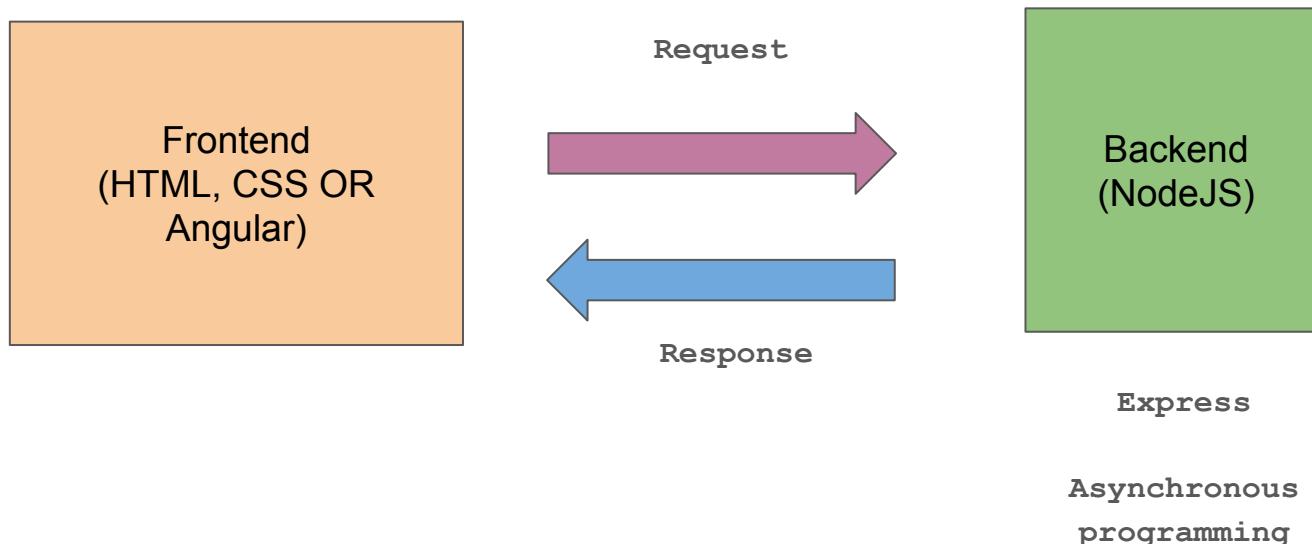


What Are we Covering Today?

- Project
- Introduction to Dynamic Programming
- Review of NodeJS
 - Unit tests
 - Asynchronous programming

Project (NodeJS)

- Simple client/server model



- Example: Algorithm submission tool

- Frontend: accepts the solution
- Backend: runs unit tests and send back the result

1. Two Sum

Easy 14071 513 Add to List Share

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have **exactly** one solution, and you may not use the same element twice.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,  
Because nums[0] + nums[1] = 2 + 7 = 9,  
return [0, 1].
```

Accepted 2,699,419 | Submissions 5,974,406

Seen this question in a real interview before?

Contributor

Companies 

Related Topics

Similar Questions

Show Hint 1

1/34

```
1 class Solution {  
2     public:  
3         vector<int> twoSum(vector<int> &nums, int target) {  
4             vector<int> r;  
5             unordered_map<int, int> m;  
6  
7                 for (int i=0; i < nums.size(); i++) {  
8                     m[nums[i]] = i + 1;  
9                 }  
10  
11                 for (int i=0; i < nums.size(); i++) {  
12                     if (m[target - nums[i]] && i != m[target - nums[i]] - 1) {  
13                         return { i, m[target - nums[i]] - 1 };  
14                     }  
15                 }  
16             return r;  
17         }  
18     };  
19 };
```

Your previous code was restored from your local storage. [Reset to default](#)

JavaScript/NodeJS Basics

Dynamic Types

- Unlike C++, variables in JS (and Python, Ruby, etc) are **dynamically typed**.
 - The type of a variable may not be known until runtime.

```
function main() {  
    let myVar = 1;  
    console.log('myVar: ', myVar);  
    myVar = "This is a string now."  
    console.log('myVar: ', myVar);  
    myVar = true;  
    console.log('myVar: ', myVar);  
}
```

Defining Variables

- **const:**
 - Has block level scope
- **let:**
 - Has block level scope
- **var:**
 - Has function level scope
 - Similar to function parameters
 - It's not recommended to use it

```
function main(myParameter) {  
  if (true) {  
    const myConst = 1;  
  
    myConst++; // This is obviously an error!  
  
    let myLet = 1;  
  
    var myVar = 1;  
  }  
  
  // myVar DOES have scope here!  
  
  console.log("myVar: ", myVar);  
  
  
  // myLet doesn't have scope here!  
  
  console.log("myLet: ", myLet);  
  
  
  // myConst doesn't have scope here!  
  
  console.log("myConst: ", myConst);  
}  
  
// myParameter doesn't have scope here!  
  
console.log("myParameter: ", myParameter);
```

Strings

- Loosely similar functionality to std::string in C++
- We can use single quote OR double quote OR backticks
 - With backticks, you can interpret variables inline

```
let thisYear = 2020;

let myString = "Hello world " + thisYear.toString();

let myString2 = 'Hello world '+ thisYear;

let myString3 = `Hello world ${thisYear}`;

console.log('myString: ', myString);
console.log('myString2: ', myString2);
console.log('myString3: ', myString3);

}
```

Strings

- Some useful methods/properties:
 - **+** : concatenation
 - **charAt(i)**: get the i(th) character
 - **length**: size of the string
 - **toUpperCase()** and **toLowerCase()**
 - **slice(start, end)**: get substring start to end
 - **match(regExp)**: matches a regular expression

Arrays

- Similar to C++ arrays, but we initialize them using []
 - **push(e)**: (add e to the end)
 - **pop()**: (remove from the end)
 - **length**: (returns the size. It's a property, not a function!)
 - **shift()**: removes from the front
 - **unshift(e)**: add e to the front
 - **indexOf(e)**: returns the index of e in the array
 - **splice(...)**: remove and insert
- Items in an array don't have to be of the same type!

```
let myArray = [];  
for(let i=0 ; i < 10 ; i++) {  
    myArray.push(i);  
}  
  
let myArrayMix = [1, 2, "a"];
```

Objects

Objects

- Loosely speaking:
 - A collection of (key,value), where keys are strings, and values can be anything (including objects, arrays, or even functions.)
 - Similar to C++ maps, but again, the types can be different, some values can be a map themselves.
 - Essentially, can create a tree
- Working with objects

```
let cup = {  
  color: "red", // String  
  bought_year: 2002, //Number  
  owners: ["Ari", "Jess", "Tom"] // Array  
};  
  
console.log('cup: ', JSON.stringify(cup, null, 2));
```

JSON

- A syntax for **serializing** objects, arrays, numbers, strings, booleans, and null.
- It is based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON.
- Widely used for passing data over the Internet

```
{  
  "color": "red",  
  "bought_year": 2002,  
  "owners": ["Ari", "Jess", "Tom"],  
  "age": 12  
}
```

JSON Object

**Array inside
object**

Object inside object

```
{  
  "firstName": "John",  
  "isAlive": true,  
  "age": 27,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "office",  
      "number": "646 555-4567"  
    }  
  ],  
  "children": [],  
  "spouse": null
```

Functions

Function Definitions

- Use function keyword
- Parameters don't take types
 - However, please always use [JSDOC](#)

```
function add(a, b) {  
    return a + b;  
}
```

```
/**  
 * Adds two variables  
 * @param {Number | String} a  
 * @param {Number | String} b  
 * @returns {Number | String}  
 */  
function add(a, b) {  
    return a + b;  
}
```

Assigning a Function to Variable

- Functions can be assigned to variables.
- The new variable acts as an alias to the function.

```
function add(a, b) {  
    return a + b;  
}  
  
let myAdd = add;  
let r = myAdd(1,2);  
console.log("r: ", r);
```

Assigning a Function to Variable

- We can also define an anonymous function and assign it to a variable
- This will not be hoisted.

```
// Valid, but won't be hoisted
let myAdd = function(a,b) {
    return a + b;
}
```

Functions Passed as Parameters

- This is similar to pointer to functions in C++.

```
function add(a, b) {  
    return a + b;  
}  
  
function operate(a, b, f) {  
    return f(a,b);  
}  
  
let r = operate(5,6, add);  
console.log('r: ', r);
```

Arrow Functions

- Very common for passing as parameters
- In other situations, use function keyword

```
let r = operate(5, 6, add);  
  
let r2 = operate(5, 6, (a, b) => a + b);  
console.log("r2: ", r2);
```

Pass by Value/Reference

For Primitives

- For **Primitives**: Always pass by value (Different than C++!)
- Sorry, no pass by reference
- Sorry, no pointers either (or maybe we should be happy!)

```
function increment(i) {  
    i++;  
}  
  
function main() {  
    let i = 0;  
    increment(i);  
    console.log('i: ', i);  
}
```

For Arrays

- Always pass by reference

```
function arrayPush(array, i) {  
    array.push(i)  
}  
  
function main() {  
    let array = [];  
    arrayPush(array, 10)  
    console.log('array: ', array);  
}
```

Objects

- Always pass by reference
 - Functions can modify the object properties

```
function addDefaultName(person) {  
    person.firstName = "Tommy";  
    person.lastName = "Trojan";  
}  
  
function main() {  
    let p = {};  
    addDefaultName(p);  
    console.log('p: ', p);  
}
```

Objects

- If the parameter object is reassigned, the original doesn't change
 - Kinda like a C++ pointer

```
function addDefaultName(person) {  
  person.firstName = "Tommy";  
  person.lastName = "Trojan";  
  p2 = {  
    firstName: "Ari",  
    lastName: "jones"  
  };  
  person = p2;  
}  
  
function main() {  
  let p = {};  
  addDefaultName(p);  
  console.log("p: ", p);  
}
```

Pass by Reference or Value?

- For **Primitives**: Always **pass by value**
 - No reference, No pointer
- For **Objects (and arrays)**
 - The "value" is a reference to the object (effectively call by reference)
- Assignment to a variable never changes the underlying primitive or object, it just points the variable to a new primitive or object.
 - Loosely similar to pointers in C++: when a pointer points to a new object, the previous object doesn't change.
- Changing a property of an object referenced by a variable does change the underlying object.
 - Loosely similar to pointers in C++: When we modify an object through a pointer, the value of the object changes.

Exceptions

Exceptions and Error Handling

- Responding to the occurrence anomalous or exceptional conditions requiring special processing
- What happens when we execute this code?

```
function makeFullName(firstName) {  
  let fullName = firstName + lastName;  
  console.log('fullName: ', fullName);  
}  
  
function main() {  
  makeFullName("Ari");  
  console.log("End of main");  
}
```

Does this line get
executed?

Exceptions and Error Handling

- Exception handling allows us to deal with error and **continue execution.**

```
function makeFullName(firstName) {  
  try {  
    let fullName = firstName + lastName;  
    console.log("fullName: ", fullName);  
  } catch (error) {  
    console.log("error: ", error);  
  }  
}  
  
function main() {  
  makeFullName("Ari");  
  console.log("End of main");  
}
```

Does this line get
executed?

Errors and Names

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

```
let quarantine = true;

function goToTheBeach() {
    if (quarantine) {
        throw "You should be at home now!";
    }
}

function main() {
    try {
        goToTheBeach();
    } catch (error) {
        console.log("Error: ", error);
        console.log("Let's learn NodeJS instead!");
    }
}

console.log("Stay safe!");
}
```

Does this line get
executed?

What Do You Need to Take Away?

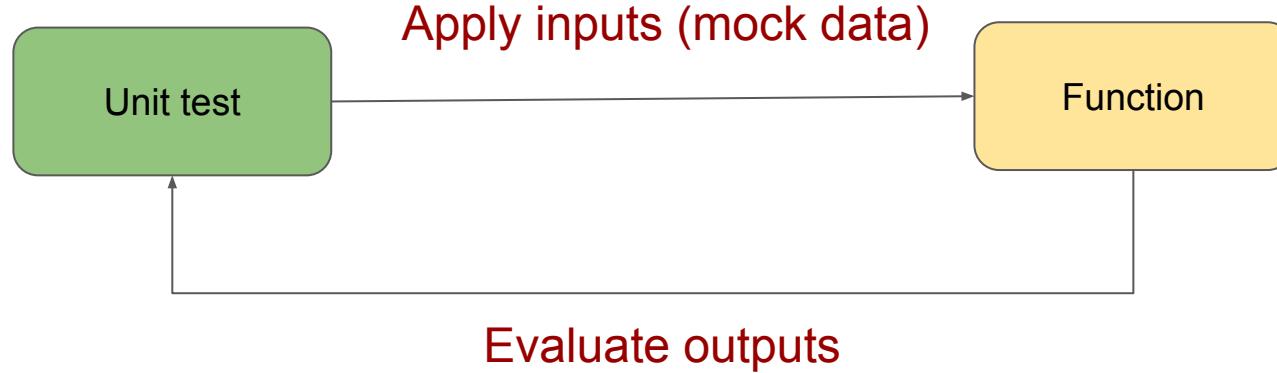
- `try...catch` is your friend!
 - It allows your program to keep running despite bad conditions
 - However, make sure you handle the error reasonably
- Anything that **can** go wrong, should be in `try...catch`
 - A server can be down
 - A user value is not provided
 - The parameter passed to the function is not as expected
 - ...
- The more error predictions you have, the friendlier your program.

Unit Tests



Unit Tests

- A unit test is a **piece of code** that tests a **function** or a **class**
- Unit tests are written by the **developer!**



Mocha

- Open Source testing framework for Javascript
- Automates various tasks:
 - Creates a main function
 - Calls our function under test
 - Applies inputs
 - Provides various functions for testing
- Has BDD Interface

Anything that throws

```
describe("FindMax Test with Mocha", () => {  
  it("should return max", () => {  
    let inputs = [1, 5, -9, 150, 3, 55, 67];  
    let myMax = myLib.FindMax(inputs);  
    assert.equal(myMax, 151);  
  });  
});
```

```
describe("An always failing test.", () => {  
  it("should should fail", () => {  
    throw new Error("Bad test!");  
  });  
});
```

Chai

- An expectation library
- A more friendly way of asserting expectations

```
describe("Should provide demo", () => {
  it("should test equality", () => {
    expect(1).to.be.equal(1);
    expect(2).to.not.be.equal(1);
    expect(2).to.greaterThan(1);
    expect(2).to.be.a("Number");
    expect([]).to.be.an("array").that.is.empty;
    expect([3, 2, 1]).to.be.an("array")
      .that.has.members([1, 2, 3]);
    expect([3, 2, 1]).to.be.an("array")
      .that.has.not.any.members([4, 5]);
    expect({ c: 3 }).to.not.have.any.keys("a", "b");
  });
});
```