

# Data Structures and Algorithms

C++ for Developers

# Content

- Introduction
- Arrays
- Linked List
- Stack
- Queue

# Introduction

C++ for Developers

# Code without data structures

No good way of accessing data in an effective way.

```
// without data structures
char char1 = 'C';
char char2 = 'a';
char char3 = 'r';
char char4 = 'I';

std::cout << char1 << char2 << char3 << char4 << '\n';
```

# What is a data structure?

A data structure is a way to store and organize related data in a consistent and efficient manner so that it can be accessed and manipulated effectively.

- Arrays
- Vectors

# What does a data structure contain?

## Structure

- A defined way of storing data in memory
- Determines how elements are arranged, linked, and accessed

## Operations

- A set of functions or procedures to interact with the structure
- Define how we insert, delete and traverse / search

# Why are data structures important?

Data structures give us the possibility to manage large amounts of data efficiently in a defined manner.

# What is an algorithm?

A step-by-step sequence of well-defined instructions that solves a specific problem or accomplishes a task.

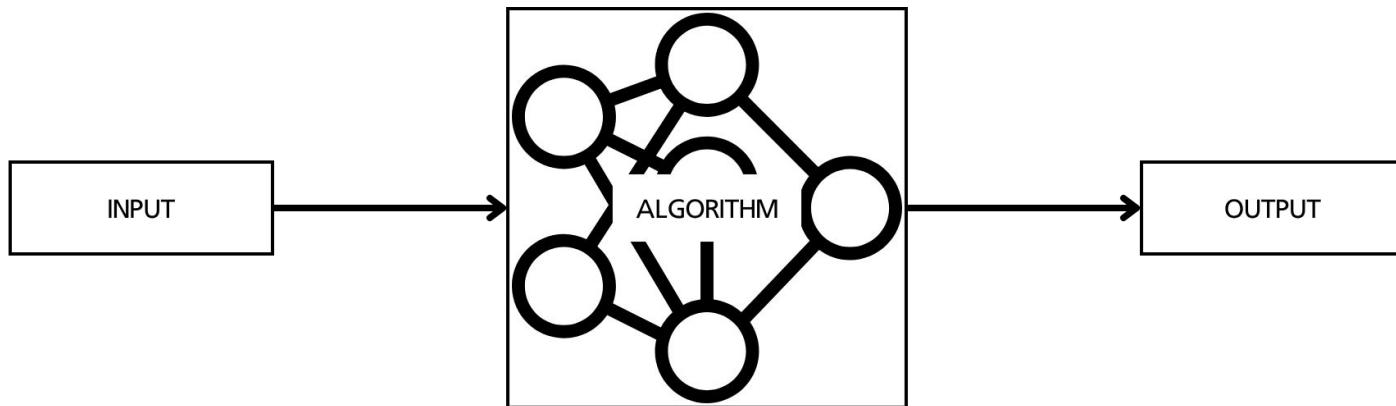
- Procedure
- Deterministic
- Unambiguous

# Difference between an algorithm and a function?

Algorithms are the abstract idea of how to solve the problem

Functions are the implementation of that algorithm

# Power of abstraction



# Arrays

C++ for Developers

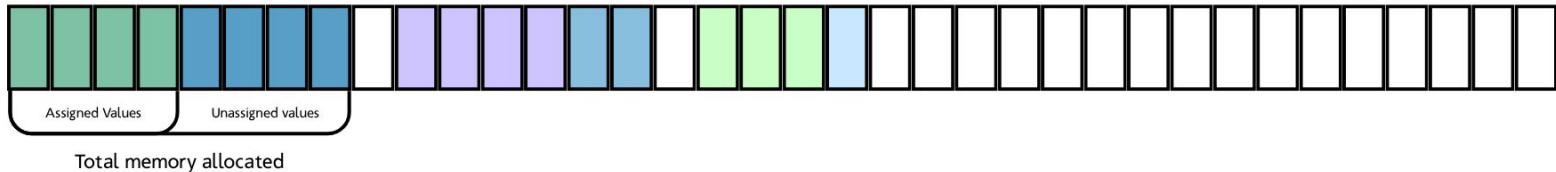
# Arrays

Stored contiguously in memory - one after another. Need to specify size when declared.

Accessed through [i] or \*(ptr + i). Can be accessed randomly.

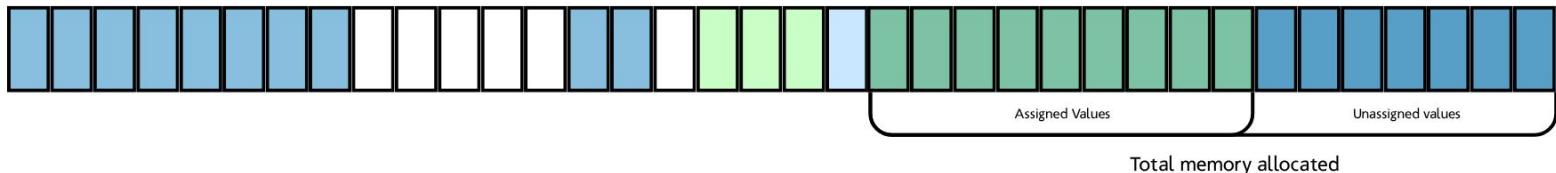
Raw data structure without operations, a derived data type

```
std::vector<int> values = {3,9,2,1};
```



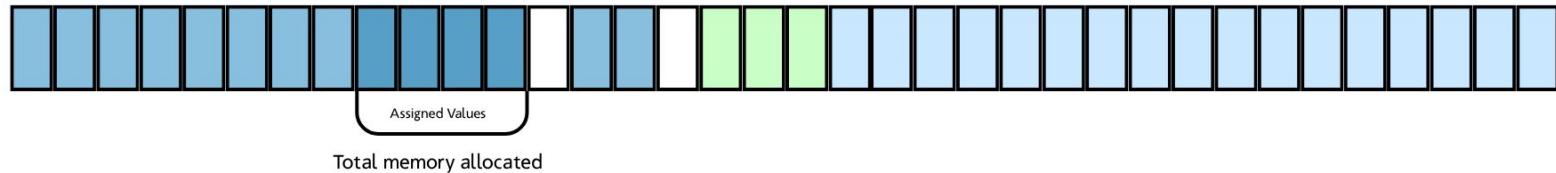
```
std::vector<int> values2 = {13, 24, 7, 1, 2};  
values.insert(values.end(), values2.begin(), values2.end());
```

**Reallocated**



```
values.erase(values.begin() + 4, values.end());
```

**Reallocated**



# **std::vector - Dynamic Array**

Dynamically allocated array on the heap. With defined member functions like:

- size()
- push\_back()
- pop\_back()

# Let's look at how std::vector behaves under the hood

[https://github.com/lafftale1999/cpp\\_for\\_developers/blob/main/week\\_4/1\\_data\\_structures\\_and\\_algorithms/3\\_dynamic\\_array/main.cpp](https://github.com/lafftale1999/cpp_for_developers/blob/main/week_4/1_data_structures_and_algorithms/3_dynamic_array/main.cpp)

# We can see that

A vector:

- Keeps track on its size during shrink at grow.
- It needs to be reallocated during runtime

Incredibly fast random access through indexing - but heavy on reallocation.

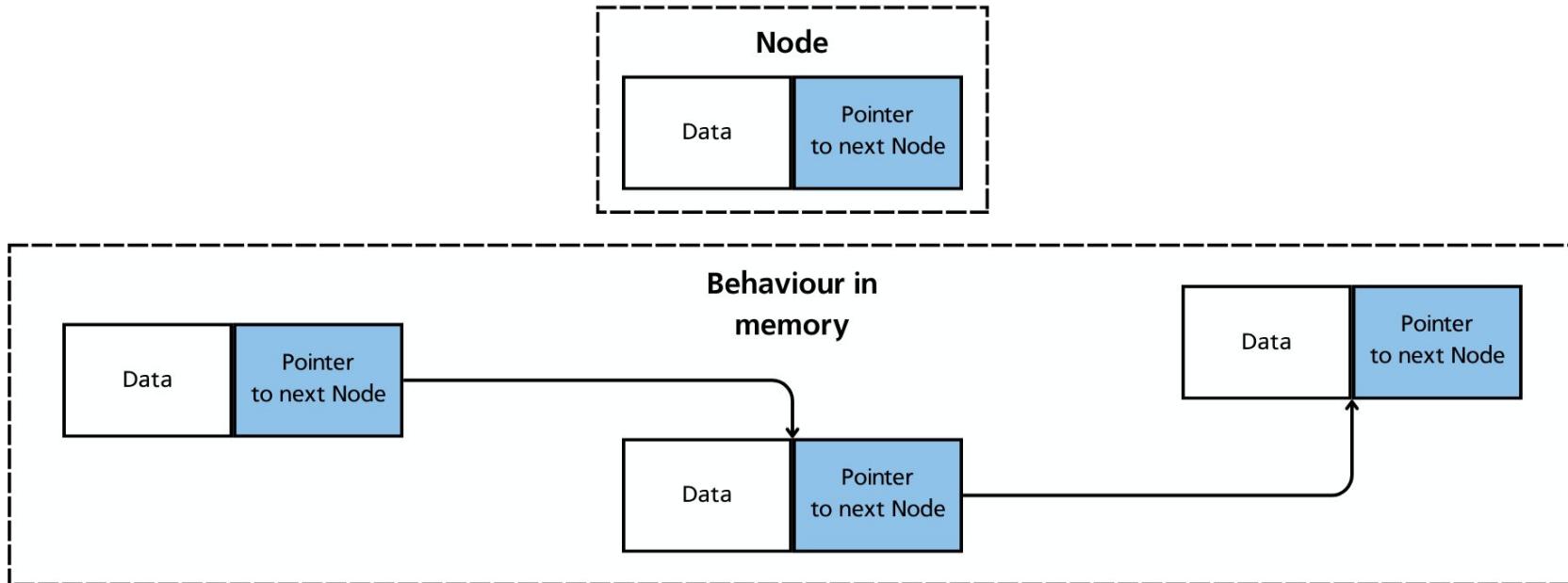
# Linked List

C++ for Developers

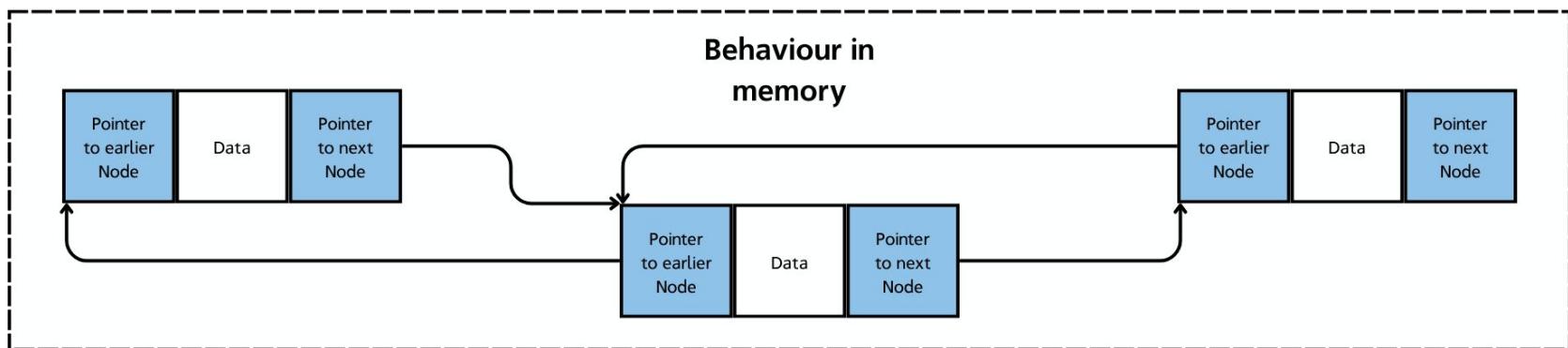
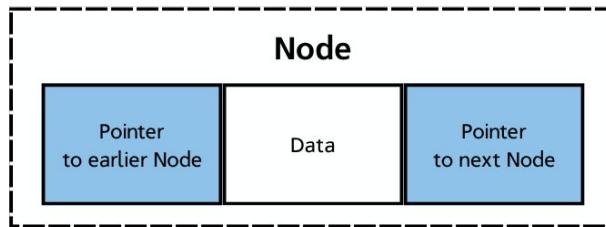
# Linked List

Collections of nodes containing data and the address of next node.

# Single Linked List



# Double Linked List



# Linked List

- Great for quick inserts at start or end
- Does not require reallocation
- No random access, need to traverse

# Let's look at how an linked list behaves!

[https://github.com/lafftale1999/cpp\\_for\\_developers/blob/main/week\\_4/1\\_data\\_structures\\_and\\_algorithms/4\\_linked\\_list/main.cpp](https://github.com/lafftale1999/cpp_for_developers/blob/main/week_4/1_data_structures_and_algorithms/4_linked_list/main.cpp)

# We can see that

A linked list:

- Keeps track on its length during insert and deletion
- It only keep track on its head (and tail if double-linked)
- Does not need reallocation
- Can't be randomly accessed

# Tree Structure

C++ for Developers

# Tree Structure

Built with nodes, where each node contains data and pointer to the next ones in the tree.

Examples of where tree data structure is used:

- File systems
- Maps and Sets in C++

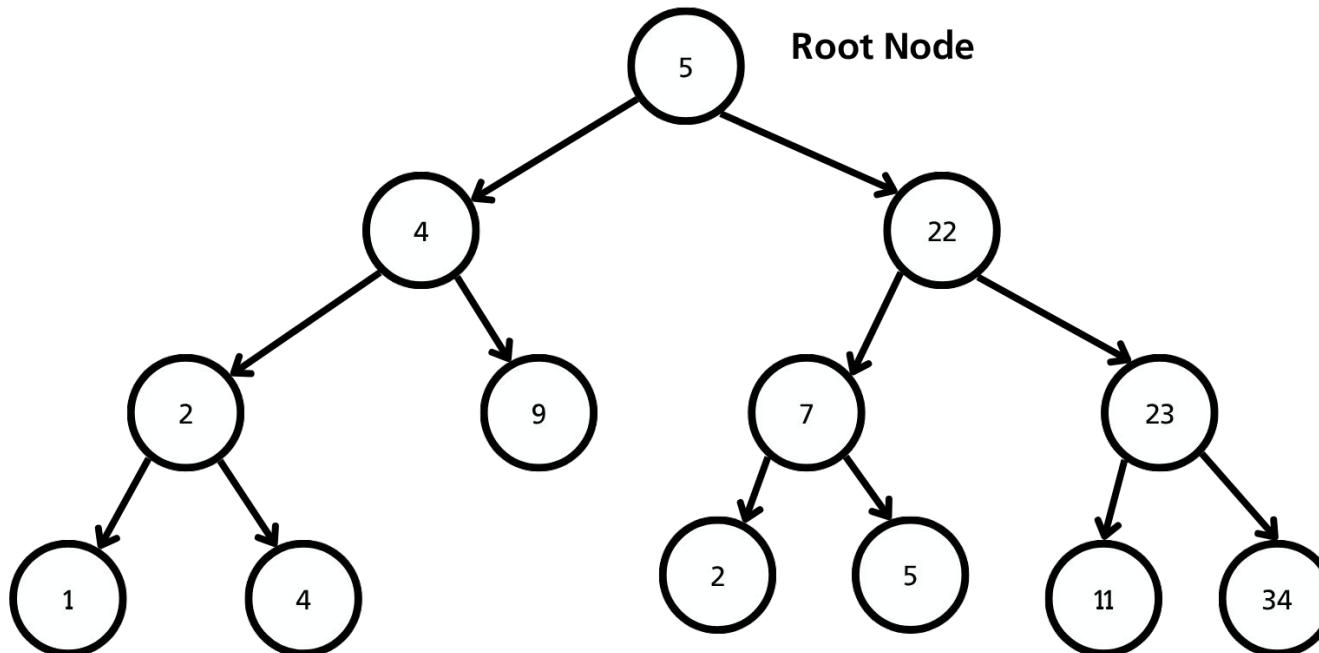
# Tree Structure

There are different types of trees. For example:

- Binary Tree - Each node can have at most two children
- Ternary Tree - Each node can have at most three children
- N-ary Tree - Each node can have at most N children

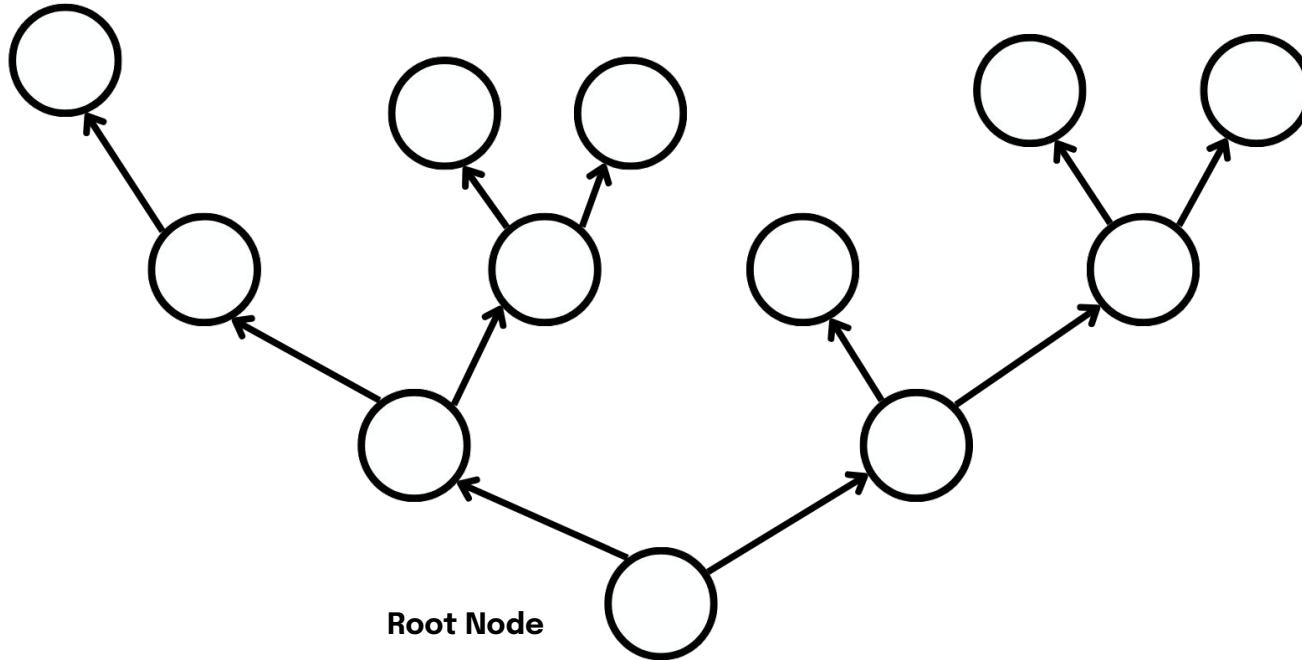
We will look at a Binary Tree in this course

# Tree Structure

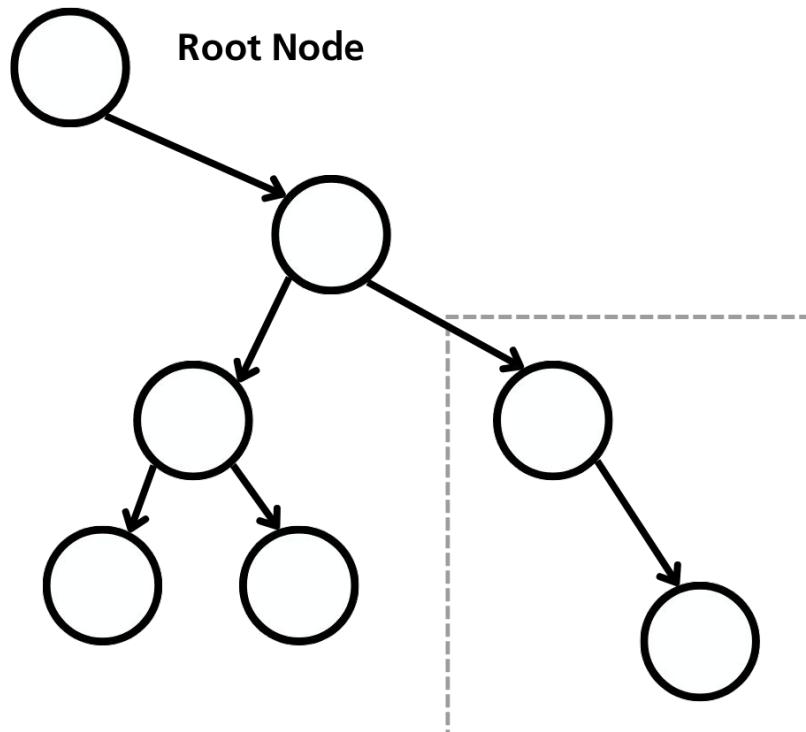


Nodes at end of tree is called *Leaves*

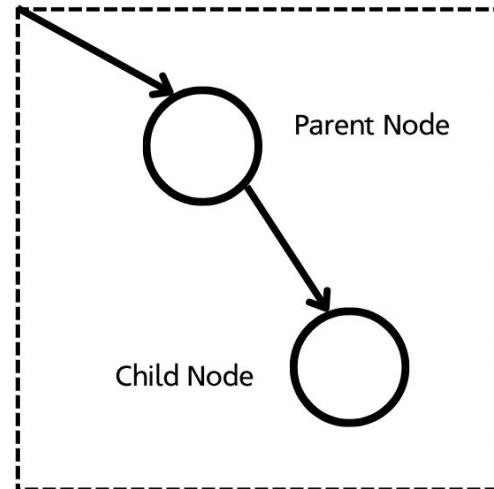
# Might make more sense?



# Node Relationships



Relationship between nodes

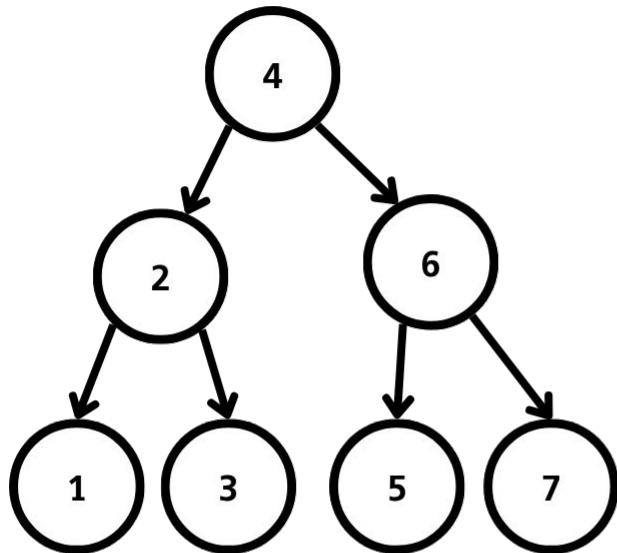


# Traversing a Tree

When traversing trees we use two different types of algorithms:

- DFT (Depth-First Traversal)  
Deepest node (leaf) before going up.
- BFT (Breadth-First Traversal)  
Level by level - Root to leaves.

# DFT: Inorder



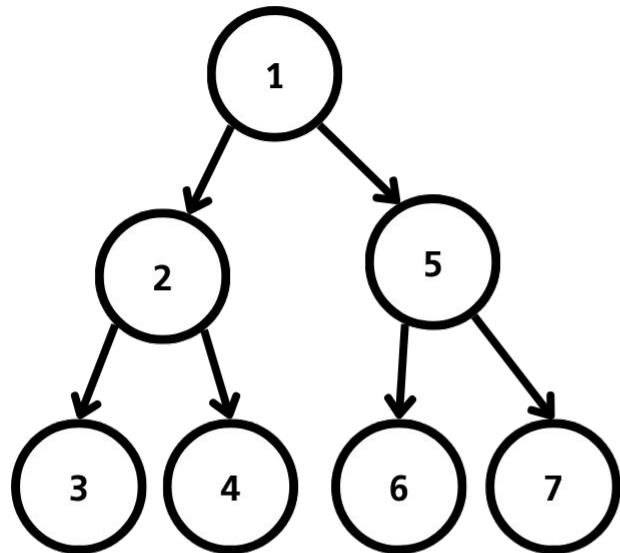
## Traversal:

1. Deepest of the left node.
2. Parent node
3. Right child node

## Areas of use:

- Traversing an ordered binary search tree

# DFT: Preorder



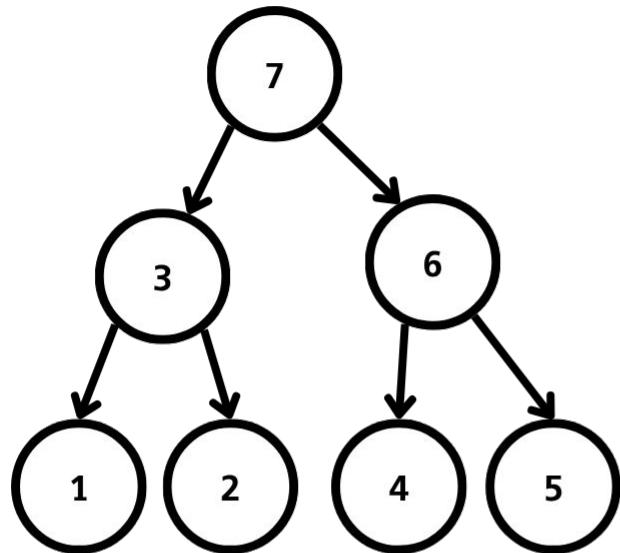
**Traversal:**

1. Root
2. Traverse the left subtree
3. Traverse the right subtree

**Areas of use:**

- Good for copying a tree

# DFT: Postorder



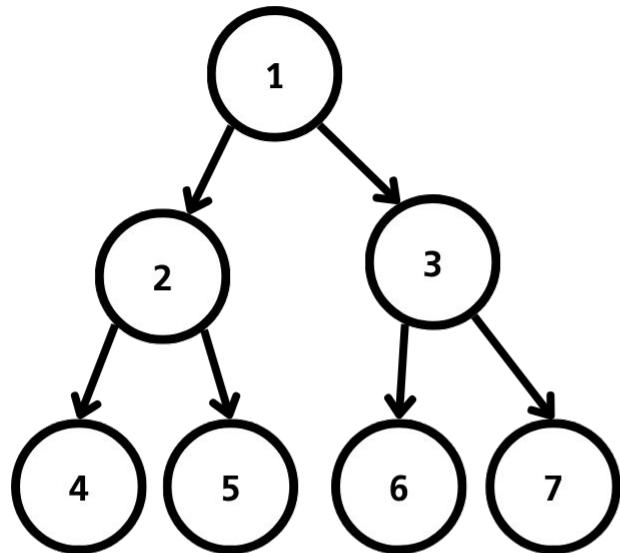
## Traversal:

1. Left-most node
2. Right node
3. Root

## Areas of use:

- Deleting a tree

# BFT



## Traversal:

Visit each level of the tree

## Areas of use:

- Looking at each level
- Measuring amount nodes on each level

# Let's look at how a binary search tree behaves!

[https://github.com/lafftale1999/cpp\\_for\\_developers/blob/main/week\\_4/1\\_data\\_structures\\_and\\_algorithms/5\\_binary\\_search\\_tree/main.cpp](https://github.com/lafftale1999/cpp_for_developers/blob/main/week_4/1_data_structures_and_algorithms/5_binary_search_tree/main.cpp)

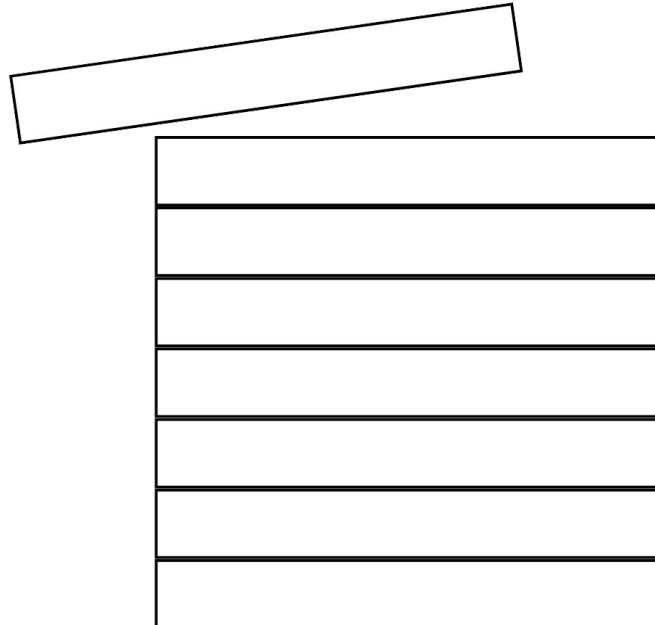
# Stack

C++ for Developers

# Stack

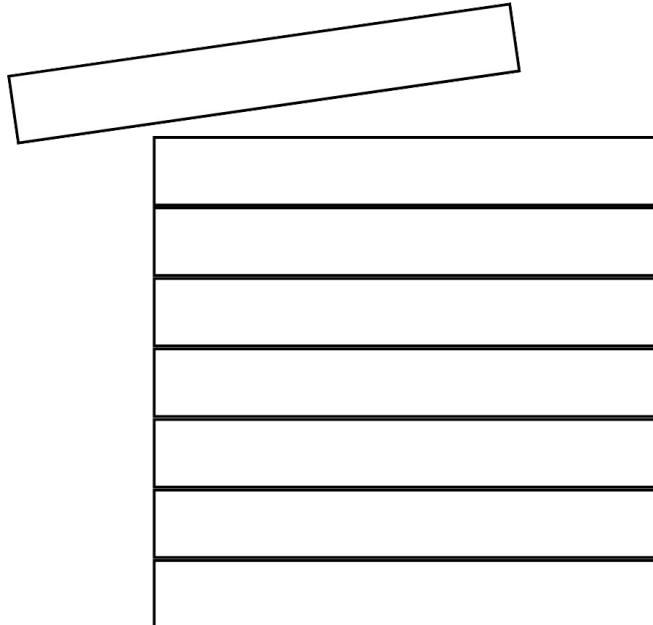
LIFO - last in first out

We can only place and remove on top of the stack



# Usual methods

- `pop()` - remove element from top of stack
- `push()` - push element on top of stack
- `top()` - look at element on top of stack



# Areas of use

- Text editors - ctrl z uses stack
- Reverse arrays / lists / strings
- Program memory
- Keep track of nodes in a depth first search algorithm

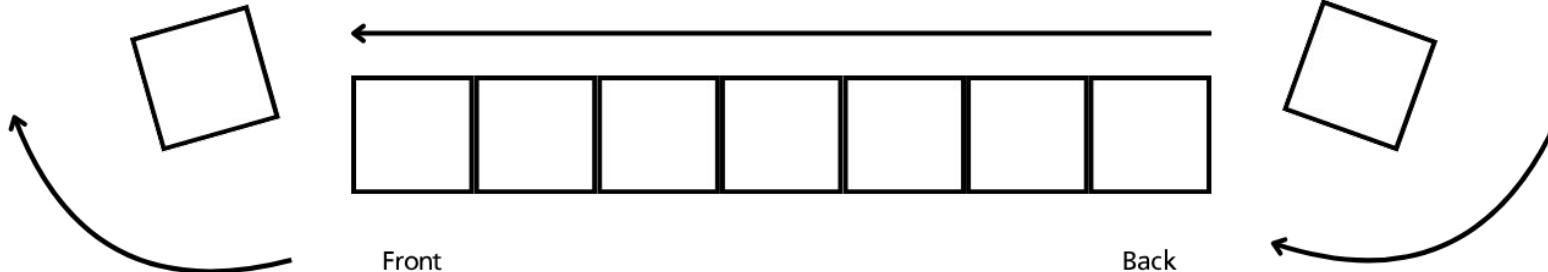
# Queue

C++ for Developers

# Queue

Fifo - First in first out

We can only remove from front and push to back.



# Common methods

- `pop()` - remove element at front of queue
- `push()` - add element to back of queue
- `front()` - look at element at front of queue
- `empty()` - return boolean value if its empty

# Areas of use

- Scheduling tasks
- Producer-Consumer
- BFT algorithms
- Buffers - input streams
- Cache

# Generic Code

C++ for Developers

# Imagine this...

You want to create a data structure that can store different data types.  
How would you go about this?

Overloading it with several different data types? A lot of repeating code!

# Templates

Enables us to write generic code. Included in the STL (Standard Template Library).

This is used to make data types used generic. By defining the template before the scope we use it in, we can create generic code.

# Defining Templates: Functions

```
template <typename T>
T addTwoNumbers(T x, T y) {
    return x + y;
}
```

Add the syntax:

```
template<typename T>
```

Before the scope you want  
to define your template.

Example from:

[https://github.com/lafftale1999/cpp\\_for\\_developers/blob/main/week\\_4/2\\_templates/1\\_basic\\_example.cpp](https://github.com/lafftale1999/cpp_for_developers/blob/main/week_4/2_templates/1_basic_example.cpp)

```
int main(void) {
    std::cout << addTwoNumbers(5, 10) << '\n';
    std::cout << addTwoNumbers(5.3, 10.2) << '\n';
}
```

Enables us to call the function using different data types!

# Defining Templates: Classes

```
template <typename T>
class TemperedArray {
private:
    std::unique_ptr<T[]> elements;
    size_t size;
    size_t capacity;
```

Add the syntax:  
template<typename T>

Before the scope you want  
to define your template.

Example from:

[https://github.com/lafftale1999/cpp\\_for\\_developers/blob/main/week\\_4/2\\_templates/2\\_template\\_array.cpp](https://github.com/lafftale1999/cpp_for_developers/blob/main/week_4/2_templates/2_template_array.cpp)

Enables us to  
create arrays  
with different  
data types.

```
TemperedArray() = default;
TemperedArray(size_t c)
: elements(nullptr), size(0), capacity(c) {
    elements = std::make_unique<T[]>(c);
    if (!elements) {
        throw std::runtime_error("Unable to allocate array");
    }
}

TemperedArray(std::initializer_list<T> e)
: elements(nullptr), size(e.size()), capacity(0) {
    capacity = size == 0 ? 10 : size * 2;
    elements = std::make_unique<T[]>(capacity);
    if (!elements) {
        throw std::runtime_error("Unable to allocate array");
    }

    std::copy(e.begin(), e.end(), elements.get());
}
```

```
TemperedArray<std::string> names = {"carl", "marl", "squarl", "harl"};

TemperedArray<int> values = {3, 9, 23, 19, 8};

TemperedArray<double> values = {3.9, 2.4, 9.82};
```

# What happens?

Compiler generates code that corresponds to the data type used.

- Templatized functions generate functions based on the caller
- Templatized classes generate classes based on the caller

