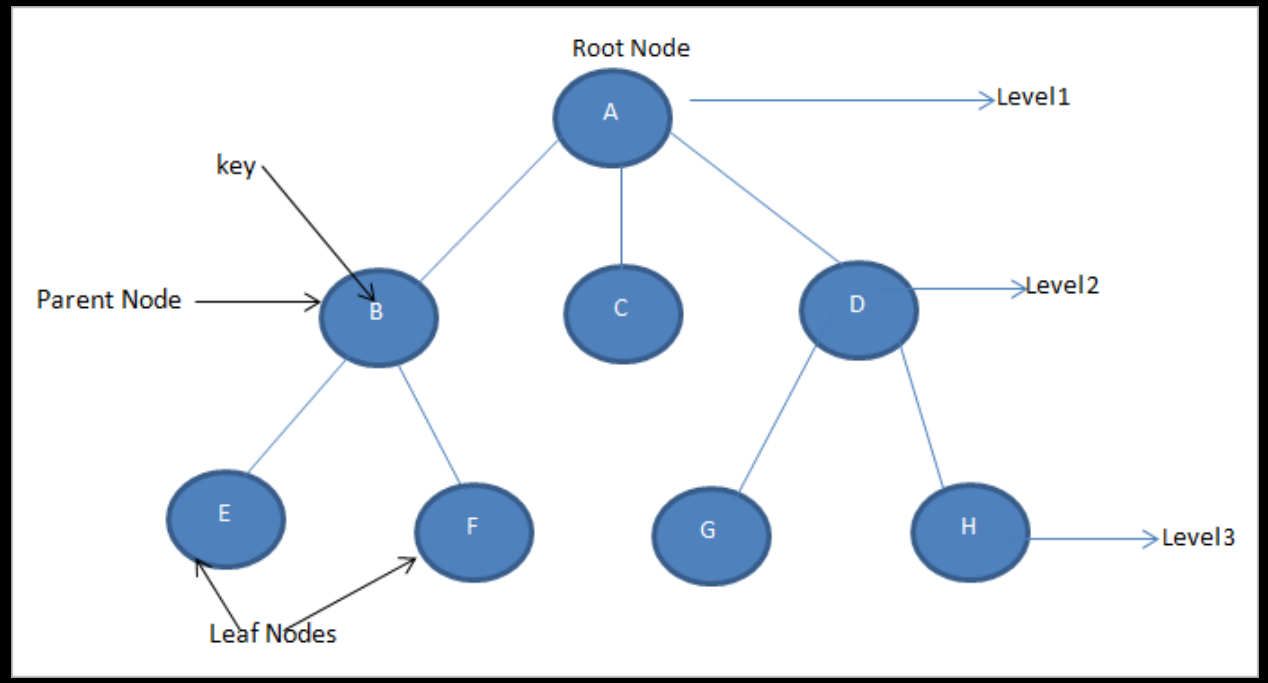
Getting Ready for Canadian Computing Competition

Class 7
January 11, 2020

Outline

Trees

Tree Terminology



Node

Edge – connecting two nodes

Key

Level

Root Node - topmost node

Leaf node - Bottom most nodes

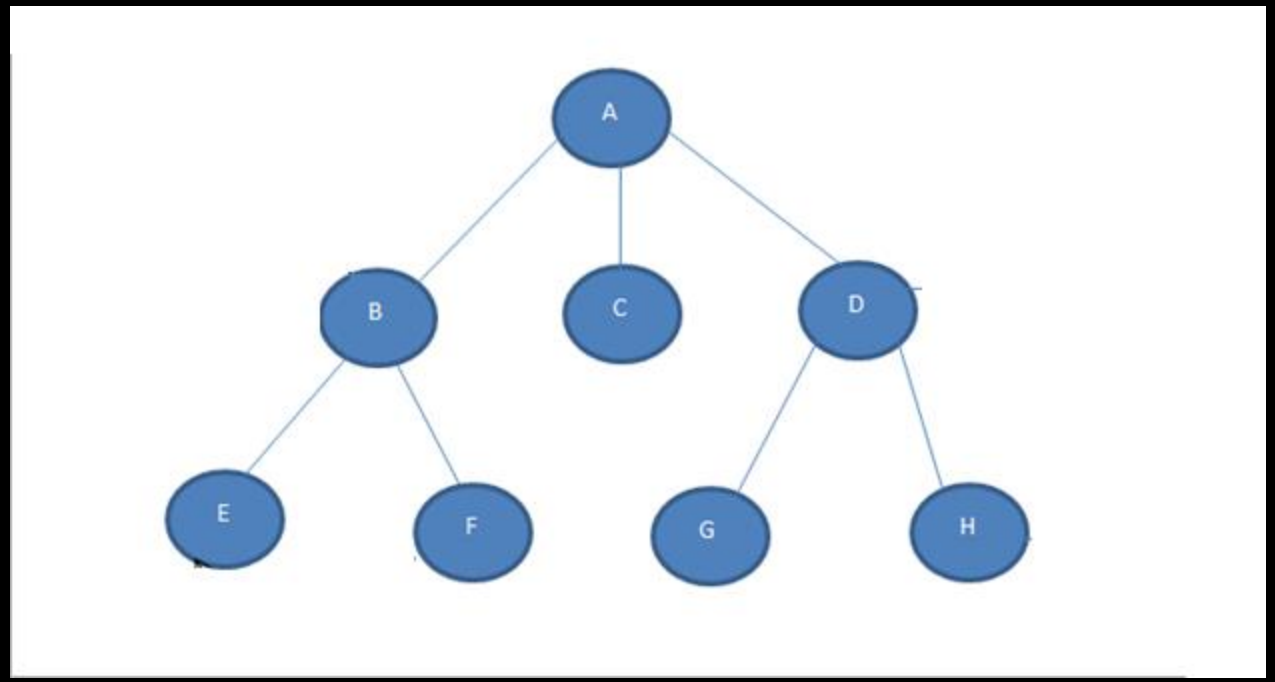
Sibling nodes – two nodes on the same level

Tree Terminology

- **Subtree** – descendants of a node from another node
- **Parent node**
- **Ancestor node** – predecessor node on path from root
  - The root node has no ancestors and no parent
- **Path** – sequence of consecutive edges
- **Degree** – number of children that a node has

Exercise

Fill in the blank with the most accurate term

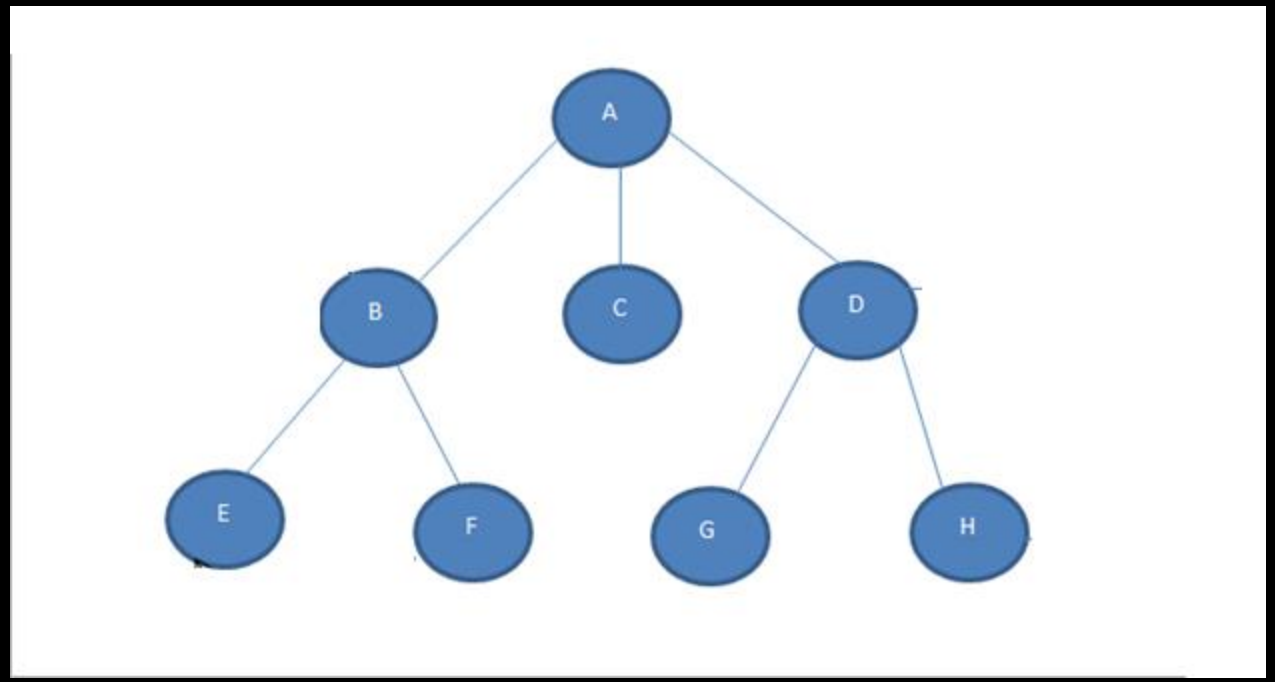


B is the ____ of E

A. Parent
B. Ancestor
C. Child
D. Root node
E. Path

Exercise

Fill in the blank with the most accurate term

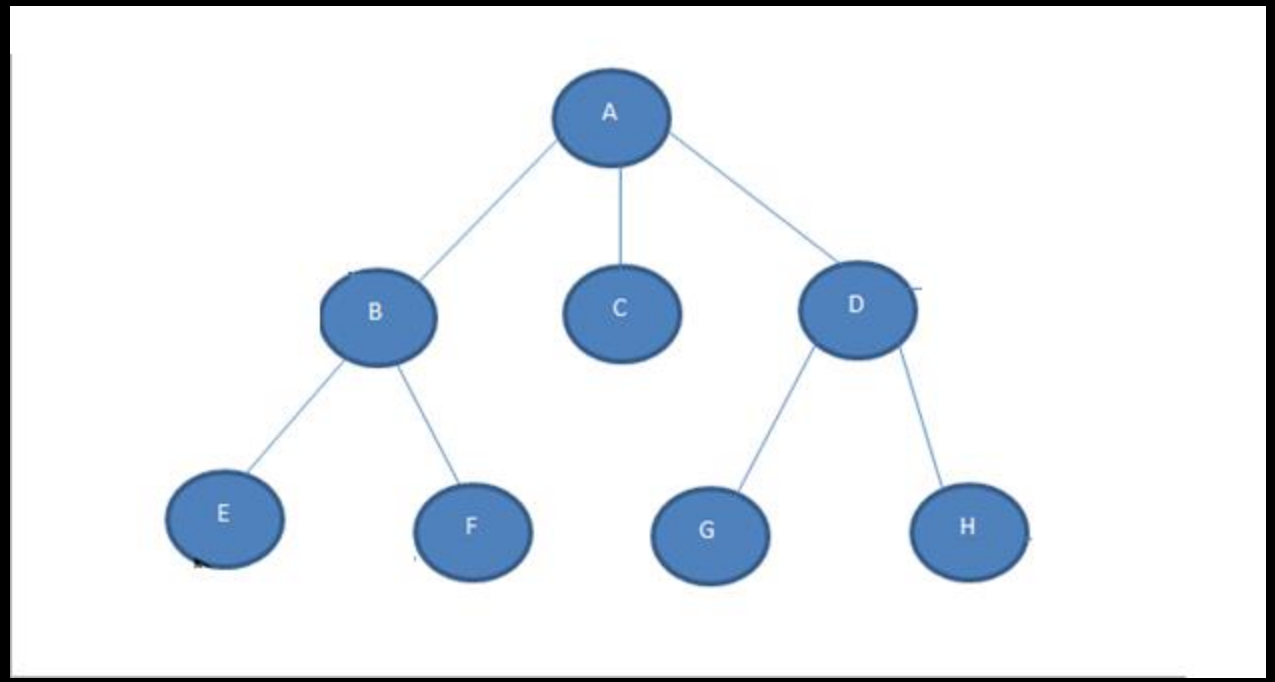


A is the ____ of G

A. Parent
B. Ancestor
C. Child
D. Root node
E. Path

Exercise

Fill in the blank with the most accurate term

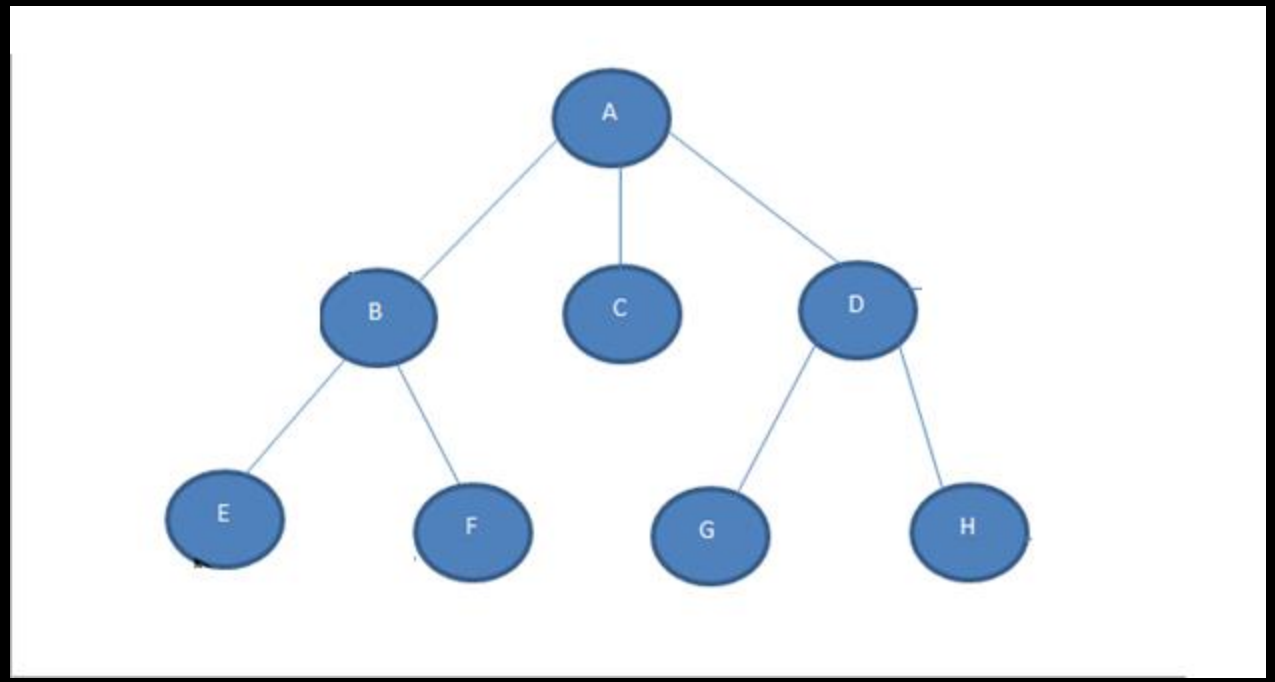


A is the _____

A. Parent
B. Ancestor
C. Child
D. Root node
E. Path

Exercise

Fill in the blank with the most accurate term

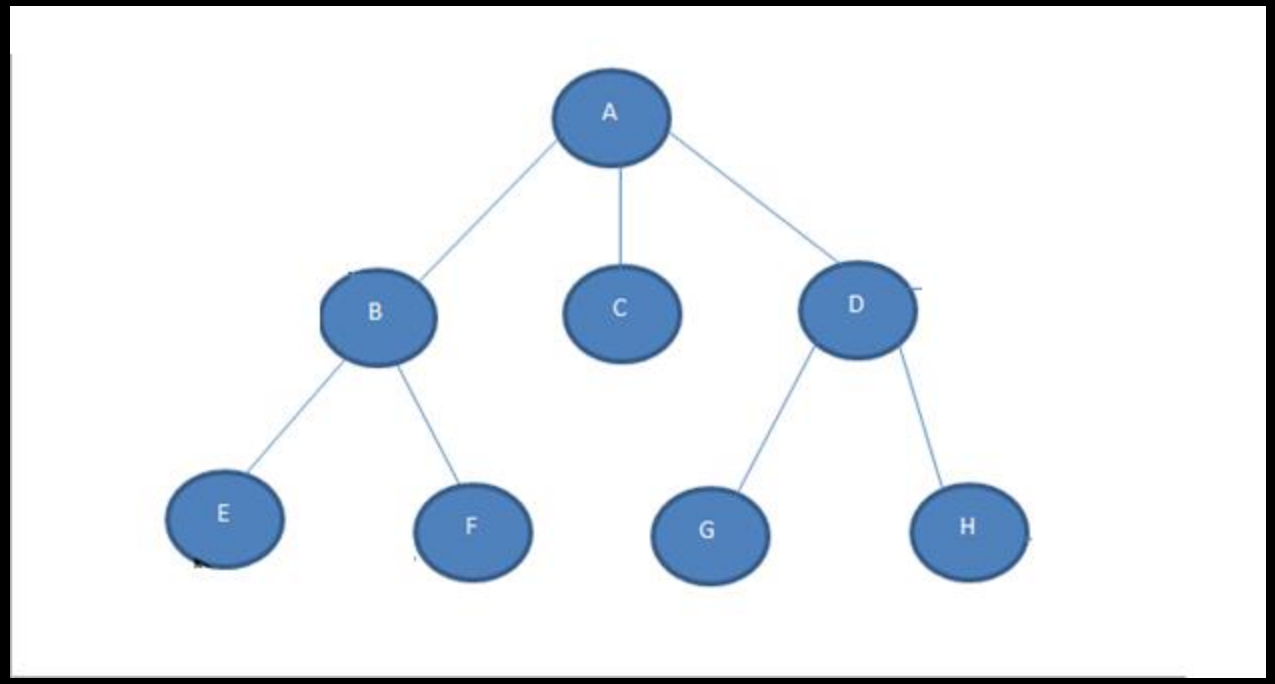


C to G is a ____

A. Parent
B. Ancestor
C. Child
D. Root node
E. Path

Exercise

Fill in the blank with the most accurate term

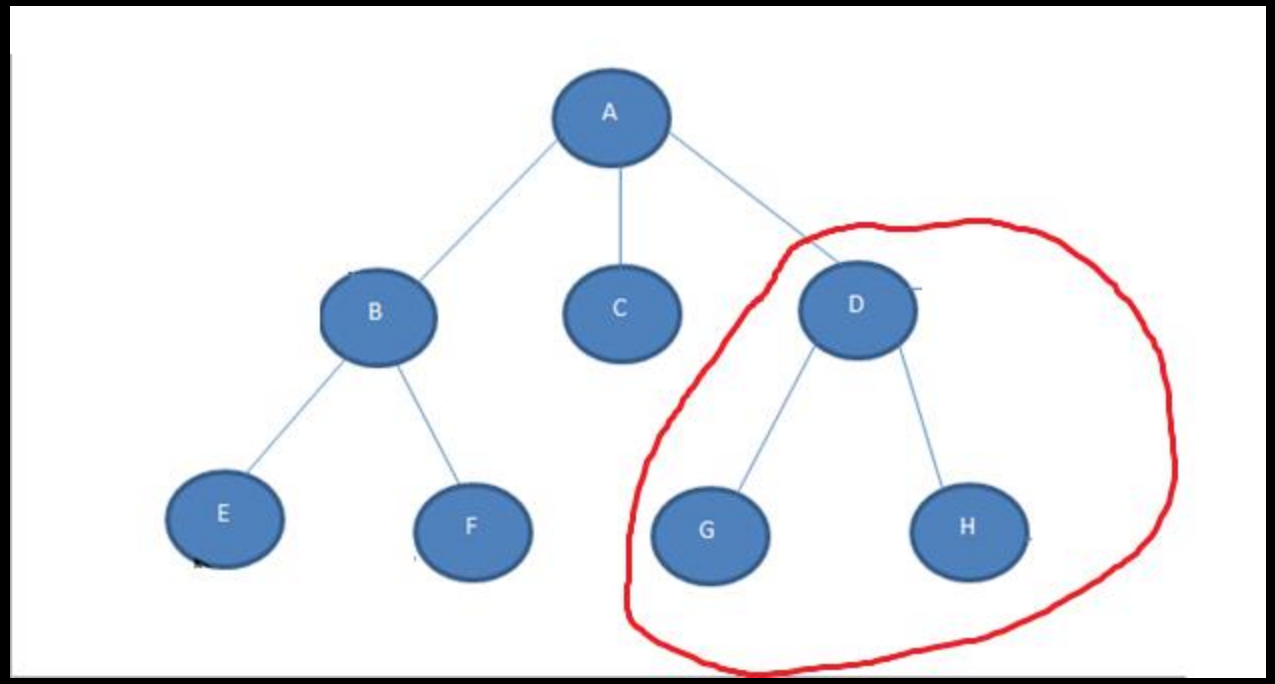


AD is a(n)

A. Sibling
B. Edge
C. Node
D. Path
E. Subtree

Exercise

Fill in the blank with the most accurate term



The circled part is a

A. Parent
B. Ancestor
C. Child
D. Root node
E. Subtree

Types of Trees

- General Tree
  - Nodes can have any number of children
  - Children are usually stored in an array/vector/dynamically allocated memory for easier access and modification

- Binary Tree
  - Every node has exactly two children
  - Children can be stored in variables (ex. Left, right) instead of in complicated data (vector, array, etc.)

Representing a Binary Tree

- We will need a structure to represent each node
- How about this?

```
struct Node {
    int val;
    Node left, right;
};
```

A. Sure!

B. This looks wrong…

Representing a Binary Tree

- We will need a structure to represent each node
- How about this?

```
struct Node {
    int val;
    Node left, right;
};
```

A. Sure!

B. This looks wrong…

How big is the struct?
How do I allocate memory?

Representing a Binary Tree

- A structure can never contain a field of itself

Representing a Binary Tree

- A structure can never contain a field of itself
- But we can contain a pointer! Pointers are always 4 bytes

```
struct Node {
    int val;
    Node *left, *right;
};
```

Processing Trees

- Trees contain subtrees, which can be treated as a new trees => Recursion
- For example, creating a tree will require recursive calls on a node's children
- Problem:
  - Recursion needs a separate function
  - We cannot pass a tree by reference because the tree creation function will be popped from the stack
  - But the structure itself has pointers, even if we try to pass by value, it is still a shallow copy

Creating a Tree

We will need two elements to solve this problem:

1. Constructor
2. Heap

Constructor

- Terminology:
  - When a struct is defined, no memory is allocated.
  - When a structure is declared, memory is allocated. We call this an **object** or **instance of the class**
- When an object is created, the constructor is called
- If we do not write a constructor, the C++ compiler generates two default constructors for us
- Once we write our own, we lose the default constructors

Constructor

- A constructor has same name as the struct itself
- Constructors do not have return types

Example:

```
struct Student {
    string name;
    int student_num, grade;
};
int main() {
    Student s1; // default constructor
    Student s2{"Jasmine", 1, 99}; // default constructor
}
```

Example

- Every object has a built-in pointer called **this** which points to itself
- Example:

Assume that we want to automatically assign a student number to every student. We also want to set grade = 0 for beginning of term.

```
int counter = 0;
struct Student {
    string name;
    int student_num, grade;
    Student (string name) { // same name as struct
        this->name = name;
        this->grade = 0;
        this->student_num = counter;
        ++counter;
    }
};
```

Calling a Constructor

The corresponding main function can be:

```
int main() {

    Student s1 {"A"};

    cout << s1.name << " " << s1.student_num << " " << s1.grade
<< endl; // A 0 0

    Student s2 {"B"};

    cout << s2.name << " " << s2.student_num << " " << s2.grade
<< endl; // B 1 0

}
```

Exercise

What would this line of code in main produce?

```
Student s3;
```

A. An uninitialized student
B. Error
C. Student {“C”, 2, 0}
D. A pointer

Exercise

What would this line of code in main produce?

Student s3 {“C”, 2, 0};

A. An uninitialized student
B. Error
C. Student {“C”, 2, 0}
D. A pointer

Simplifying the Constructor

```
int counter = 0;
struct Student {
    string name;
    int student_num, grade;
    Student (string name) { // same name as struct
        this->name = name;
        grade = 0;
        student_num = counter;
        ++counter;
    }
};
```

C++ is smart enough to know grade means this->grade

Simplifying the Constructor - MIL

- Member Initialization List
- Syntax: Constructor (parameters) : field_name1 {value from parameter}, field_name2 {value2} ,… field_namen {valuen} { body }
- You can leave out any field name and it will be filled with default values

```
int counter = 0;
struct Student {
    string name;
    int student_num, grade;
    Student (string name) : name {name}, grade{0}, student_num {counter} {
        ++counter;
        }
};
```

Constructors

- You can define multiple constructors because C++ allows function overloading
- You can also declare constructors in the struct and define it outside. You must use StructName::ConstructorName instead of just ConstructorName

Exercise

1. Complete the definition of the constructor by setting x and y to 0.

```
struct Posn {
    int x, y;
    Posn();
};
```

2. Add another constructor that takes in two numbers and sets x and y accordingly

3. Write a main function that calls these two constructors

4. What happens if I write Posn {1};

Constructor Calling

- Always use {} instead of ()
- Example:

```
struct Posn {
    int x, y;
};

int main() {
    Posn a(1,2); // error
    Posn b{1,2};
}
```

Practice - Node Constructor

Write a constructor for the Node struct:

```
struct Node {
    int val;
    Node *left, *right;
};
```

Practice - Node Constructor

Write a constructor for the Node struct:

```
Node (int val) : val{val}, left {nullptr}, right{nullptr} {}
```

Heap

A program is divided into a few segments:

- Code (read-only)
- Global variables
- Data
- Call stack
- Heap

Heap

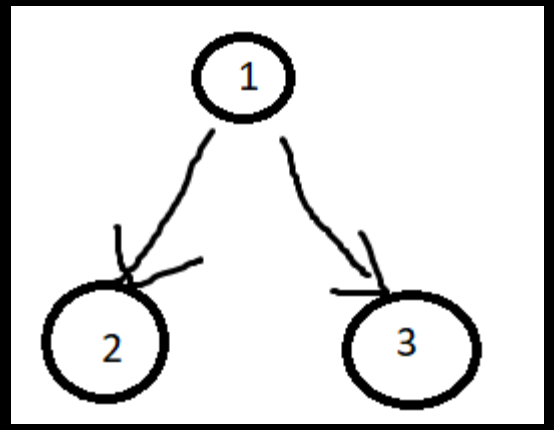
- A pool of memory for dynamic memory allocation
- You should return all burrowed memory at the end of execution
  - Otherwise it will cause a memory leak
  - Good news …
- The “new” keyword allocates memory from the heap
- Syntax: new data_type (or data_type {…} which calls its constructor)
- Returns a pointer to the allocated memory

Exercise

Given the Node struct and its constructor, allocate memory from the heap to create a root node with value 1 (in main).

Exercise

Given the Node struct and its constructor, create the following tree on the heap (in main)



Stack vs Heap

- Stack: Objects are destroyed once they go out of scope

- Heap: Objects are only destroyed when you manually delete it

Creating a Binary Tree

- Recall our node structure:

```
struct Node {
    int val;
    Node *left, *right;
};
```

- Can we use lvalue references for left and right?

A. Yes
B. No

Exercise

- Which of the following is a leaf node? (Assuming all variables have been defined)

A. `Node {1, &temp, nullptr}`

B. `Node {2, &leftNode, &rightNode}`

C. `Node {0, &nullNode, nullptr}`

D. `Node {10, nullptr, nullptr}`

Checking for Leaf Nodes

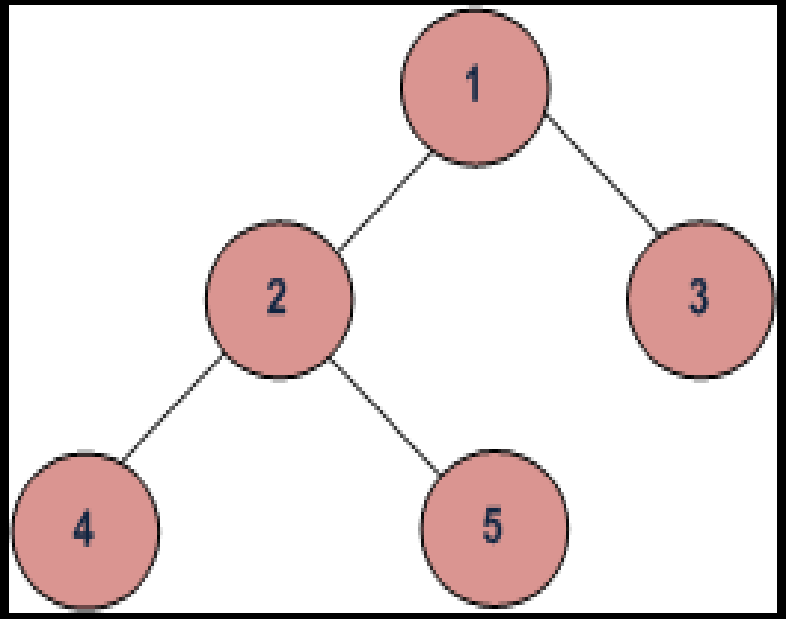
- We can use pointers as bools to determine if they are null pointers
- Example:

```
Node *p = nullptr;
if (p) {
    cout << “not null”<< endl;
} else {
    cout << “null” << endl;
}
// prints null
```

Tree Traversal

- Traversing a tree means visiting the nodes of a tree
- There are many ways to visit trees:
  - Breadth first / level order
  - Depth first (next class)
    - Inorder
    - Preorder
    - postorder

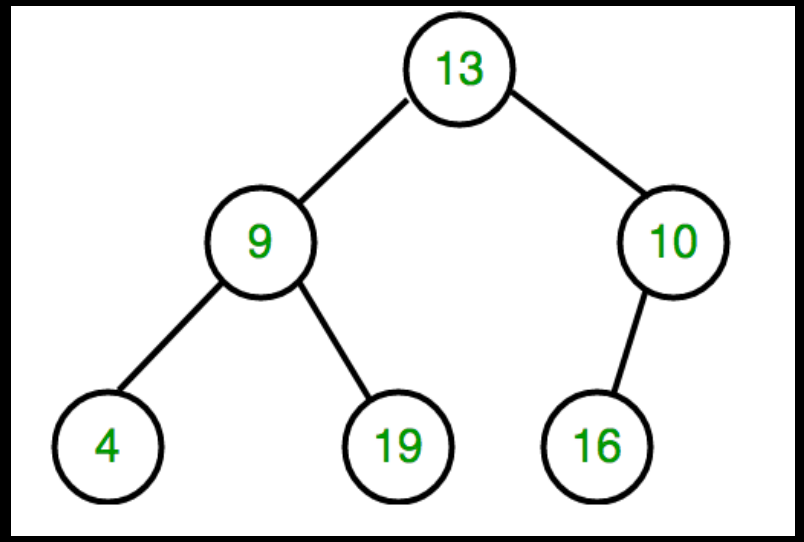
Level Order Traversal



- Left to right
- Visit all sibling nodes at the present depth before moving on to the next left
- Example: 1 2 3 4 5

Exercise

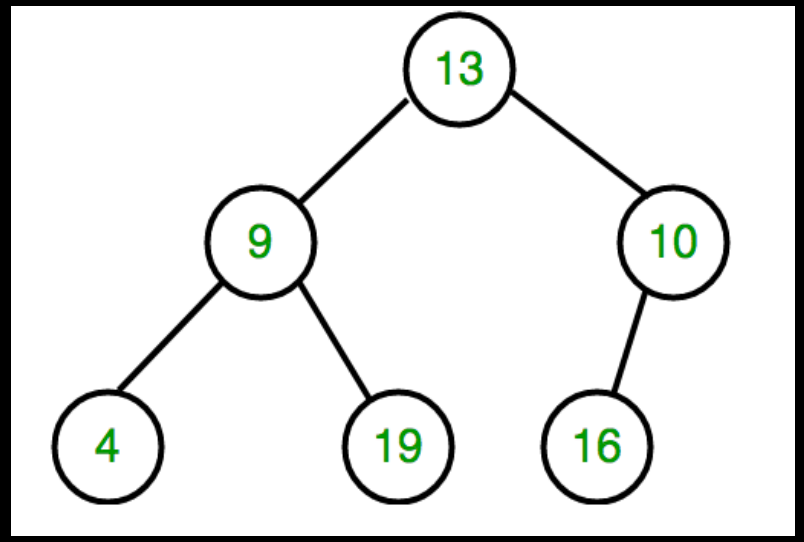
What doe printing this tree in level order traversal produce?



A. 13 9 4 19 10 16
B. 13 9 10 4 19 16
C. 16 19 4 10 9 13
D. 13 10 9 16 19 4

Example

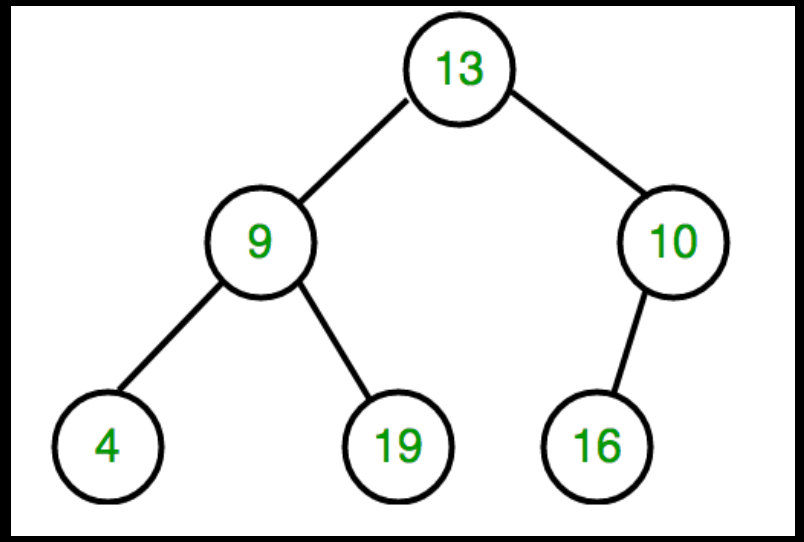
Given a list of integers that represent a binary tree in level order traversal, create the binary tree.



Given 13, 9, 10, 4, 19, 16

Example

Given a list of integers that represent a binary tree in level order traversal, create the binary tree.



Given 13, 9, 10, 4, 19, 16
Index 0 1 2 3 4 5

Left: 2 * loc + 1
Right: 2 * loc + 2

We can write it both iteratively and recursively

Iterative Approach

hw

Iterative Approach Main Function

hw

Recursive Approach

```
Node * makeTree (const vector <int> &tempVal, int n, int i = 0) {
    if (i < n) {
        Node *temp = new Node{tempVal[i]};
        temp->left = makeTree(tempVal, n, 2 * i + 1);
        temp->right = makeTree(tempVal, n, 2 * i + 2);
        return temp;
    }
    else {
        return nullptr;
    }
}
```

Recursive Approach Main Function

```
int main() {
    int n, input;
    cin >> n;
    vector <int> tempVal;
    for (int i = 0; i < n; ++i) {
        cin >> input;
        tempVal.emplace_back (input);
    }
    Node *root = makeTree(tempVal, n);
    printTree(root);
    return 0;
}
```

Exercise

What will the output of this input be if I add a line to makeTree?

```
Node * makeTree (const vector <int> &tempVal, int n, int i = 0) {
    if (i < n) {
        Node *temp = new Node{tempVal[i]};
        cout << "adding: " << tempVal[i] << endl;
        temp->left = makeTree(tempVal, n, 2 * i + 1);
        temp->right = makeTree(tempVal, n, 2 * i + 2);
        return temp;
    }
    else {
        return nullptr;
    }
}
```

Input: 5 1 2 3 4 5

Trace of Recursive makeTree

```
Node * makeTree (const vector <int> &tempVal, int n, int i = 0)
{
    if (i < n) {
        Node *temp = new Node{tempVal[i]};
        temp->left = makeTree(tempVal, n, 2 * i + 1);
        temp->right = makeTree(tempVal, n, 2 * i + 2);
        return temp;
    }
    else {
        return nullptr;
    }
}
```

Input: 5 1 2 3 4 5

tempVal:
Value: 1 2 3 4 5
Index: 0 1 2 3 4