Trees - Outline

Trees

Trees in General
Recursion
Using struct
Binary Trees
C++ review
Binary Search Trees
Exercises
Data structures that aren't trees

Sequential data structure

• Array
• Linked list
• Stack

What are some limitations of sequential data structure?

Fast or slow?

• Insertion
• Deletion
• Searches
QUESTION

How can we organize hierarchical data?
Tree data structure

Some real life examples of trees.
**Tree Terminology**

Node  A structure that may be connected to other similar node structures, each of which may contain some value. (Sometimes called a vertex)

Edge  An edge, \((u, v)\) is a link between two nodes, \(u\), and \(v\). In general, an edge can be directed or undirected, but in a tree, they are directed, with \(u\) being \(v\)’s parent.

Tree  A data structure made up of nodes, each of which may be linked to at most one other node as its parent, and an unlimited number of nodes as children (though none are required). No cycles are permitted.

Root  The topmost node in a tree, from which all other nodes descend.
**Tree Terminology, cont.**

**Path** A sequence of edges that describes a way of getting from one node to another. The length of a path is how many edges must be traversed. In a tree, there is a unique path from the root to every other node.

**Cycle** A cycle is a path of non-zero length that begins at a node and ends up at the same node.

**Parent** A node that has a child is called the child’s parent node. Each node has one parent, except for the root, which has none.

**Child** A node extending from another node.

**Sibling** Two nodes are siblings if they have same parent.

**Ancestor** A node reachable by repeated proceeding from child to parent. (Nodes on the path from the node to the root).

**Descendant** A node reachable by repeated proceeding from parent to child.
**Tree terminology, cont.**

- **Leaf** A node with no children (external node).
- **Level** The level of a node is the number of edges between the node and the root.
- **Node Height** The height of a node is the number of edges on the longest path from the node to a leaf. The height of a leaf node is zero.
- **Tree Height** The number of edges on the longest path from the root to a leaf (height of the root or maximum depth of any node). The height of an empty tree is -1.
- **Node Depth** The number of edges from the node to the root (number of ancestors). The depth of the root node is zero.
- **Internal Node** A node with at least one child.
**Labeled Tree Example**

An example tree with parts labeled.

- **Level 0**: Root node A.
- **Level 1**: Nodes B and C.
- **Level 2**: Nodes D, E, F, and G.
- **Level 3**: Node H.

- **A's left subtree**: D → H
- **A's right subtree**: C → F → G

- **Siblings**: B and C are siblings.
- **Edges**: A → B and A → C.
- **Depth**: A (root) is depth 0.
- **Height**: D (leaf) is height 1.

- **Label**: A's right subtree is labeled as F → G.
Recursion see Recursion

Something that is recursive is defined in terms of itself.

A recursive function calls itself.

Recursion can be useful as a simple way of defining a problem. It will be used a lot with trees, because a lot of things that are done with trees can be broken down by having the function call deal with smaller and smaller subtrees, until the subtree is a single node, and we can stop the recursion.

It is not always the most efficient implementation, but is often simplest to express.
Recursion example: Countdown 1

```cpp
void countDown(int count) {
    cout << "push " << count << '\n';
    countDown(count - 1);
}

int main() {
    countDown(5);
    return 0;
}

// What is the output?
```
Recursion example: Countdown 2

```cpp
void countDown(int count) {
    cout << "push " << count << '
';
    if(count > 1) // termination condition
        countDown(count - 1);
    cout << "pop " << count << '
';
}

int main() {
    countDown(5); return 0; }

// push 5
// push 4
// push 3
// push 2
// push 1
// pop 1
// pop 2
// pop 3
// pop 4
// pop 5
```
An example of recursion in the world of mathematics is the factorial operation.

\[ n! = n \cdot (n - 1)!, \quad n > 1 \]

\[ n! = 1, \quad n \leq 1 \]

```cpp
long factorial(long n) {
    if (n <= 1) return 1;
    else return n * factorial(n-1);
}

int main() {
    cout << factorial(5);
    return 0;
}
```
Recursion examples: Fibonacci numbers

Fibonacci numbers are another example of recursion in mathematics.

\[ F_n = F_{n-1} + F_{n-2} \]

\[ F_0 = 0 \quad F_1 = 1 \]

```java
long fibonacci(long n) {
    if (n < 2) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    cout << "Fibonacci of 5 = " << fibonacci(5);
    return 0;
}
```
Write a recursive function, `print_backwards`, that receives a `vector` as a parameter (e.g., 2,4,7) and prints its contents in reverse order (e.g., 7,4,2).

```cpp
void print_backwards(vector<int> v)
{
    // what goes here?
}

int main()
{
    vector<int> v = { 2, 4, 7 };
    print_backwards(v);
    return 0; }
```
```cpp
void print_backwards(vector <int> v) {
    if(v.size() > 0) {
        cout << v.back();
        v.pop_back();
        print_backwards(v); } }

int main() {
    vector <int> v = { 2, 4, 7 };    
    print_backwards(v);
    return 0; } 
```

- Why is `v` passed by value and not by reference?
- Find an expression for how much memory, in `int`s, this implementation would take, with a vector of size $N$. 

Recursive vector sum

Write a recursive function to sum all the numbers in a vector

```cpp
int findSum(vector<int> v) {
    if(v.empty()) return 0;
    else {
        int last = v.back();
        v.pop_back();
        return last + findSum(v); }
}

int main() {
    vector<int> v = {3, 7, 9};
    cout << findSum(v);
    return 0; }
```
**DATA STRUCTURES WITH struct**

A group of data elements (members) grouped together under one name.

To access the members of an object we simply insert a dot (.) between the object name and the member name.

```c
struct product
{
    int quantity;
    double price;
};
```

```c
product apple, orange; // Each object of this type has all members.
apple.quantity = 4;    // Access member of a struct
product *x = &apple;  // Pointers point to the struct
x->price *= 2.0;       // Members of pointers to struct
```
POINTERS TO STRUCTURES

To access the members of a pointer to structure

• Use the arrow operator - > (minus sign and greater than sign with no whitespace)
  • PointerToStruct - > member
• Which is equivalent to:
  • (*PointerToStruct). member

```c
product apple, *ptr;
ptr = &apple;
ptr->quantity = 5
// OR
// (*ptr).quantity = 5;
```

More details
**Example using struct**

```cpp
struct Time {
    int hour;
    int minutes;
    int seconds;
}; // <-- ends with ; just like class
void main() {
    Time now, *ptr;
    now.hour = 11; now.minutes = 2; now.seconds = 30;
    cout << "The time now is " <<
        now.hour << ":" << now.minutes << ":" << now.seconds;
    ptr = &now;
    ptr->hour++;
    cout << "\nThe time after 1 hour = " <<
        now.hour << ":" << now.minutes << ":" << now.seconds; }
// The time now = 11:2:30
// The time after 1 hour = 12:2:30
// What would we add to make the minute display properly?
```
Binary Trees

Binary Tree  A binary tree is a tree in which every node has at most two children.

Applications:

- Arithmetic expressions like \((2 \cdot (x - 1) + (3 \cdot y))\)
- Yes/no decision processes
- Searching
Properties of Binary Trees

A binary tree has the following properties:

\[ b + 1 \leq n \leq 2^{b+1} - 1 \]

\[ 1 \leq n_e \leq 2^b \]

\[ b \leq n_i \leq 2^b - 1 \]

\[ \log_2(n + 1) - 1 \leq b \leq n - 1 \]

---

\( n \) the number of nodes in the tree
\( n_e \) the number of external nodes in the tree
\( n_i \) the number of internal nodes in the tree
\( b \) the height of the tree
Struct for Binary Tree nodes

Each node needs to have:

- The data the node contains – `data`
- A pointer to the left child – `left`
- A pointer to the right child – `right`

```c
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

Pointer to the root node is stored outside the node structures. This root pointer defines an entry point into the binary tree. If a child is not present, the corresponding pointer will be `nullptr`. 
**Example setting up nodes**

```c
struct Node {
    int data;
    Node *left;
    Node *right;
};

/* Helper function that allocates a new node with the given data and NULL left and right pointers.*/
struct Node * newNode(int data) {
    Node * n = new Node; // dynamically allocate new objects of type Node
    n->data = data;
    n->left = nullptr;
    n->right = nullptr;
    return (n);
}
```
How `maxDepth` will be used

Write a C++ program to find the maximum depth of a tree

```cpp
int main() {
    Node *root = newNode (1);
    root->left = newNode (2);
    root->right = newNode (3);
    root->left->left = newNode (4);
    root->left->right = newNode (5);
    cout << "The maximum depth is " << maxDepth(root);
    return 0; }
```
**Implementation of maxDepth**

/* Compute the maximum depth of a tree -- the number of edges along the longest path from the root node down to the lowest leaf node. */

int maxDepth(Node* n) {
    if (n == nullptr) // why?
        return -1; // depth of an empty tree
    else {
        // compute the depth of each subtree
        int lDepth = maxDepth(n->left);
        int rDepth = maxDepth(n->right);
        // use the larger one
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
        // OR, equivalently
        // return 1 + max(maxDepth(n->left), maxDepth(n->right));
Binary Tree Traversal

Unlike the list or the vector, trees don’t have an inherent order. We couldn’t just use our standard iterator loop to visit everything inside.

Instead, we have some options on how to proceed. They all start with the root. We will eventually handle everything, but at each node, we can decide which to handle first

- the current node
- one or both of its subtrees
- its siblings

These choices lead to different recursive traversal algorithms

- Depth-first traversal
  - Inorder
  - Preorder
  - Postorder
- Breadth-first traversal - level order
**Inorder traversal (LNR)**

- Traverse the left subtree
- Visit the node (We cannot do step 2 until we have finished step 1)
- Traverse the right subtree

```c++
void inorder(Node * p) {
    if (p != nullptr) {
        inorder(p->left);
        cout << p->data;
        inorder(p->right);
    }
}
```

Nodes labeled based on position in inorder traversal
Preorder traversal (NLR)

- Visit the node
- Traverse the left subtree
- Traverse the right subtree

```cpp
void preorder(Node * p) {
    if (p != nullptr) {
        cout << p->data;
        preorder( p->left);
        preorder( p->right); }
}
```
**Postorder traversal (LRN)**

- Traverse the left subtree
- Traverse the right subtree
- Visit the node

**Application:** on non-binary trees, this idea can be used to calculate space used by files in a directory and its subdirectories

```c
void postorder(Node * p) {
    if (p != nullptr) {
        postorder(p->left);
        postorder(p->right);
        cout << p->data;
    }
}
```
Breadth-first traversal (level order)

Starting at the root node:

- start at the leftmost node of the current level
- visit it and all of its siblings from left to right
- move to the next level

```cpp
// Notice that this one is NOT recursive
void levelorder(Node * p) {
    // Define q here. What type does it need to be?
    q.push(p); // Push the root
    while(!q.empty()) {
        Node *v = q.front();
        cout << v->data;
        if(v->left != nullptr) q.push(v->left);
        if(v->right != nullptr) q.push(v->right);
        q.pop();
    }
}
```
Traversals Example 1

Exercise: Traverse the following binary tree:

(a) in preorder; (b) in inorder ; (c) in postorder

```
A
/   \
B     C
/   \
D     
```

Binary tree to traverse

<table>
<thead>
<tr>
<th></th>
<th>Preorder</th>
<th>A B D C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Inorder</td>
<td>B D A C</td>
</tr>
<tr>
<td>(b)</td>
<td>Postorder</td>
<td>D B C A</td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td>Level order A B C D</td>
</tr>
</tbody>
</table>
Traversing Example 2

```
A
B C
D E F
```

Tree
**Traversals Example 2: Preorder**

```
A,1
B,2
C,4
D,3
E,5
F,6
```

Preorder: A B D C E F
Traversals Example 2: Inorder

Inorder: B D A E C F
Traversals Example 2 Postorder

Postorder: D B E F C A
C++ REVIEW

Classes (I) http://www.cplusplus.com/doc/tutorial/classes/

Friendship and inheritance http://www.cplusplus.com/doc/tutorial/inheritance/

- Private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to “friends”.

Include guard: Used to avoid the problem of double inclusion

```cpp
#ifndef HEADERFILE
#define HEADERFILE

// code

#endif
```

# endif
C++ Enumeration

An enumeration is a user-defined type whose value is restricted to one of several explicitly named constants (enumerators). Enumerations are defined using the keyword enum.
Example of enum

```cpp
enum seasons { spring, summer, autumn, winter };  
int main() {  
  seasons s;  
  s = summer;  
  cout << "summer = " << s << endl;  
  s = autumn;  
  cout << "autumn = " << s << endl;  
  return 0; }  
// summer = 1  
// autumn = 2
```
Templates are the foundation of generic programming, which is a way of writing code that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or function. The STL uses templates heavily, and you should be able to use them in similar ways.

There is a single implementation for each container, algorithm, etc... For instance, `vector` has a single definition, but it has a templated type, so we can declare many different kinds of `vector` without reimplementation. (`vector<int>`, `vector<string>`, etc.).
Function Template

The general form of a template function definition is shown here:

```cpp
template <class type>
returnType functionName(parameter list) {
    type a;
    // body of function
}
```

Here, `type` is a placeholder name for a data type the function uses. When the function is instantiated (called) later, the compiler generates machine code for the function with the type used. This name can be used within the function definition.

The `typename` and `class` keywords can be used interchangeably to state that a template parameter is a type variable.
```cpp
// Example of function template

template <typename T>
T Max(T a, T b) {
    return a < b ? b : a;
}

int main() {
    int i = 39;
    int j = 20;
    cout << "Max( i , j): " << Max( i , j) << endl;
    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;
    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
    return 0; }
```

// Max( i , j): 39
// Max(f1, f2): 20.7
// Max(s1, s2): World
**Multiple type parameters**

```cpp
// 2 type parameters:
template<class T1, class T2>
void someFunc(T1 var1, T2 var2) {
    // some code in here...
}
```
CLASS TEMPLATES

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```cpp
template <class type>
class class-name {
    //implement class here using type as placeholder
};
```

Here, `type` is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.
**Example: Stack**

```cpp
template <class T>
class Stack {
private:
    vector <T> elems; // elements
public:
    void push(T); // push element
    void pop(); // pop element
    T & top(); // return top element
};
```

**Review from C++:** Is this a definition or a declaration?
Method definitions

```cpp
template <class T>
void Stack <T>::push(T elem) {
    elems.push_back(elem); }

template <class T>
void Stack <T>::pop() {
    elems.pop_back(); }

template <class T>
T & Stack <T>::top() {
    return elems.back(); }
```
Example instantiating class template

```cpp
int main() {
    Stack<int> intStack; // stack of ints
    Stack<string> stringStack; // stack of strings
    // manipulate int stack
    intStack.push(7); intStack.push(5);
    cout << intStack.top() << endl;
    // manipulate string stack
    stringStack.push("hello");
    stringStack.push("world");
    cout << stringStack.top() << endl;
    stringStack.pop();
    cout << stringStack.top();
    return 0; }
```

// 5
// world
// hello
The virtual keyword

In cases where you have a base class and a derived class that have different implementations of members with the same name, the `virtual` keyword allows control over which version is called.

This is a form of something called *polymorphism*, which is an important feature of *object oriented programming* (OOP).

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

A virtual method is a method that can be redefined in a derived class, allowing the derived class version of the method to be called, even when treating it as an instance of the base class.

If this is desired, the keyword `virtual` should be added before the return type on the virtual method’s declaration in the base class.
Base class with virtual method

```cpp
class Polygon { // base class
protected:
    int width, height;
public:
    void set_values(int a, int b) {
        width = a; height = b; }
    virtual int area() {
        return 0; }
};
```
Derived classes implementing virtual method

```cpp
class Rectangle : public Polygon {
public:
    int area() { // was virtual in base class, not here
        return width * height;
    }
};
class Triangle : public Polygon {
public:
    int area() {
        return (width * height / 2);
    }
};
```
```cpp
int main() {
    Rectangle rect;
    Triangle trgl;
    Polygon poly;
    Polygon *ppoly1 = &rect,
           *ppoly2 = &trgl,
           *ppoly3 = &poly;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    ppoly3->set_values(4, 5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
    return 0; }
```

Using polymorphism with virtual methods
The member function, `area`, has been declared as `virtual` in the base class because it is later redefined in each of the derived classes.

Non-virtual methods can also be redefined in derived classes, but non-virtual methods of derived classes cannot be accessed through a reference of the base class: i.e., if `virtual` were removed from the declaration of `area` in the example above, all three calls to `area` would return zero, because in all cases, the base class’ version would have been called instead.

A class that declares or inherits a virtual function is called a *polymorphic class*. 
COPYING TREES

Shallow copy of the data

• Accomplished when value of the pointer of the root node used to make a copy of a binary tree
• Changes made to original will be reflected on the shallow copies

Deep copy of a binary tree

• Need to create as many nodes as there are in the binary tree to be copied
• The new nodes must be connected in the same way as their counterparts in the original
• Once done, the new copy is independent
**FUNCTION** `copyTree`

Given a pointer to the root node, makes a copy of a given binary tree.

```cpp
void copyTree(Node* &copiedTreeRoot, Node* otherTreeRoot) {
    if (otherTreeRoot == nullptr) copiedTreeRoot = nullptr;
    else {
        copiedTreeRoot = new Node(otherTreeRoot->data);
        copyTree(copiedTreeRoot->left, otherTreeRoot->left);
        copyTree(copiedTreeRoot->right, otherTreeRoot->right);
    }
}
```
**Binary Search Tree (BST)**

A *binary search tree* is a *binary tree* that additionally holds the following properties:

- The *key* in each node must be greater than all keys found in the left subtree, and less than all keys in the right subtree.
- All subtrees must meet the same key criterion as above.

Left tree is BST, right tree does not meet key criteria.
# Operations on Binary Search Trees

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Deletion</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Find minimum</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Find maximum</td>
<td>$O(\log_2 n)$</td>
</tr>
</tbody>
</table>
SUCCESSOR AND PREDECESSOR

Successor  The node in the tree that would be visited immediately after the current node when traversing the tree inorder.

Predecessor The node in the tree that was visited immediately before the current node when traversing the tree inorder.
**Inserting an Item in BST**

The *first* value inserted becomes the root node.

Every node is inserted as a new leaf in first available location, which is located by going left or right based on key value.

BST insertion of 3 6 5 1 2 7
**Handling duplicate keys**

Duplicate key values:

- Will be generally disallowed for simplicity’s sake.
- If the key is the only data for the node, a count of the number of times the key is present will be sufficient.
- Could be stored in another data structure within a node (e.g., make node value a list).
- Choose a side, left or right, and make sure all duplicate keys are inserted in that direction.
- Would require extra operations (e.g., getAll and removeAll)
Example BST insertion with recursion

```c
void insert(Node * &ptr, int val) {
    if (ptr == nullptr)
        ptr = new Node(val);
    else if (val < ptr->data)
        insert(ptr->left, val);
    else insert(ptr->right, val);
}
```

Notice that the `ptr` parameter is not simply a pointer to a `Node`, but a reference to a pointer to a `Node`. This allows us to change the value of the original in the scope that called our function.
**Successor**

- The successor of a node \( x \), is node \( y \), that has the smallest key greater than that of \( x \).
  1. If \( x \) has a right subtree, then \( \text{Successor}(x) \) is the leftmost element in that right subtree.
  2. If \( x \) has no right subtree, then \( \text{Successor}(x) \) is the lowest ancestor of \( x \) (above \( x \) on the path to the root) that has \( x \) in its left subtree.

\[
\begin{align*}
\text{Successor}(15) &= 17, \\
\text{Successor}(7) &= 9, \\
\text{Successor}(13) &= 15
\end{align*}
\]
**Predecessor**

- The predecessor is the node that has the largest key smaller than that of x.
  1. If x has a left subtree, then the predecessor is the rightmost element of the left subtree.
  2. If x has no left subtree, then Predecessor(x) is the lowest ancestor of x (above x on the path to the root) that has x in its right subtree.

Predecessor(6) = 4, Predecessor(15) = 13, Predecessor(17) = 15
Deleting from BST

First, locate the item; then, delete it.

Binary search tree property must be preserved.

We need to consider three different cases:

1) Deleting a leaf (no children)
2) Deleting a node with only one child
3) Deleting a node with two children
Deleting a leaf

Deleting the external node with 45 as its key.
Deleting a node with only one child

Deleting the node with 20 as its key.
Deleting a node with two children

To delete a node with two children, first swap it with either its successor or its predecessor, then delete from its new position. Deleting 15.
Deleting a node with two children, result

Result of deleting 15 from previous BST
BST deletion: findNodeToDelete

```c
void findNodeToDelete(Node *&p, int item) {
    if (item < p->data)
        findNodeToDelete(p->left, item);
    else if (item > p->data)
        findNodeToDelete(p->right, item);
    else deleteNode(p);
}
```

Notice once again that the ptr argument is a reference to pointer. This allows us to set the left or right pointer in the parent when it needs to change.
**BST deletion: deleteNode**

```c
void deleteNode(Node *& ptr) {
    int data;
    Node * tempPtr = ptr;
    if (ptr->left == nullptr) {
        // Reattach the right child or nullptr
        ptr = ptr->right;
        delete tempPtr;
    } else if (ptr->right == nullptr) {
        ptr = ptr->left;
        delete tempPtr;
    } else {
        getPredecessor(ptr->left, data);
        ptr->data = data;
        findNodeToDelete(ptr->left, data);
    }
}
```
(i) Given a sequence of numbers: 11, 6, 8, 19, 4, 13, 5, 17, 43, 49, 16, 31, 32

(a) Draw a binary search tree by inserting the above numbers from left to right
(b) What is the height of the above tree?
(c) Show the two trees that can be resulted after the removal of 19.
Exercise Answers (a), (b)

Binary Search Tree from (a)

(b) What is the height of the above tree? 4
Exercise Answers (c)

(c) Show the two trees that can be resulted after the removal of 19.
More exercises

(2) Which of the following traversals always visits the nodes of a BST in sorted order?

[ ?] Preorder
[ ?] Inorder
[ ?] Postorder

(3) In a Binary Search Tree, the largest element must:

[ ?] be the root
[ ?] be a leaf
[ ?] have at least one child
[ ?] have at most one child

(4) Given an array of comparable data. How would you sort it using a BST?
MORE EXERCISES: ANSWERS

(2) Which of the following traversals always visits the nodes of a BST in sorted order?

[ ] Preorder
[ ] Inorder
[ ] Postorder

(3) In a Binary Search Tree, the largest element must:

[ ] be the root
[ ] be a leaf
[ ] have at least one child
[ ] have at most one child

(4) Given an array of comparable data. How would you sort it using a BST?

• insert it into a BST
• traverse the BST inorder
**Binary tree interview questions**

Define tree, binary tree and binary search tree. Now implement a function that verifies whether a binary tree is a valid binary search tree.


- [http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/](http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/)
**Binary Tree Interview Questions**

Find the minimum depth of binary search tree

- [Facebook](http://www.glassdoor.com/Interview/Find-the-minimum-depth-of-binary-search-tree-QTN_127018.htm)

Find the longest path within a binary tree

- [Amazon](http://www.glassdoor.com/Interview/Hardest-Q-was-Here-s-a-binary-tree-find-the-longest-path-within-it-So-find-a-path-between-any-two-leaf-nodes-where-the-QTN_465987.htm)
MORE INTERVIEW QUESTIONS

Using recursion to traverse a binary tree


Print the nodes on a tree level

More interview questions

Write a function that takes 2 arguments: a binary tree and an integer n, it should return the n-th element in the inorder traversal of the binary tree.

- Facebook

Find lowest common ancestor of 2 nodes in a binary tree

- Symantec

Useful resources

Binary Search Trees (BSTs) - Insert and Remove Explained
https://www.youtube.com/watch?v=wc1RPqTR3Kc

Print Ancestors of a given node in Binary Tree
http://www.geeksforgeeks.org/print-ancestors-of-a-given-node-in-binary-tree/

Find sum of all left leaves in a given Binary Tree
http://www.geeksforgeeks.org/find-sum-left-leaves-given-binary-tree/

How to handle duplicates in Binary Search Tree?
http://www.geeksforgeeks.org/how-to-handle-duplicates-in-binary-search-tree/
ACKNOWLEDGEMENTS

Starting Out with C++ from Control Structures to Objects, 8th, Gaddis, 2014. Chapter 20 – Binary tree.


http://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/


http://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html