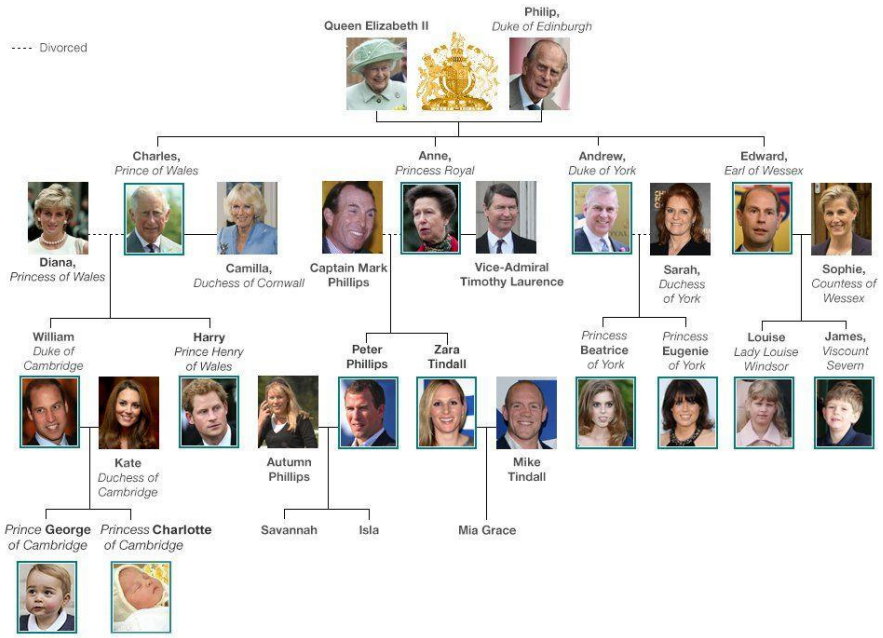
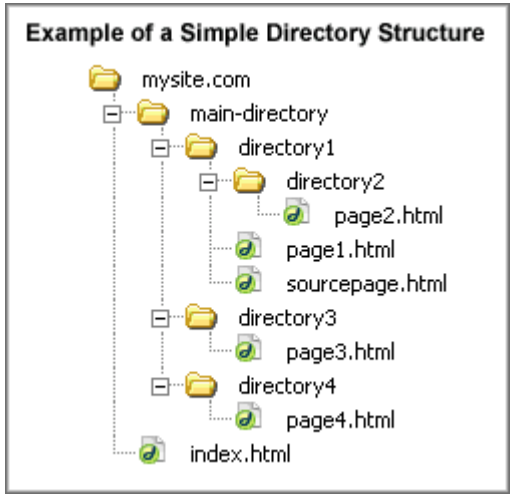
- **Sequential** data structure
  - Array
  - Linked list
  - Stack
- What are some limitations of sequential data structure?
- Fast or slow?
  - Insertion
  - Deletion
  - Searches

How can we organize hierarchical data?





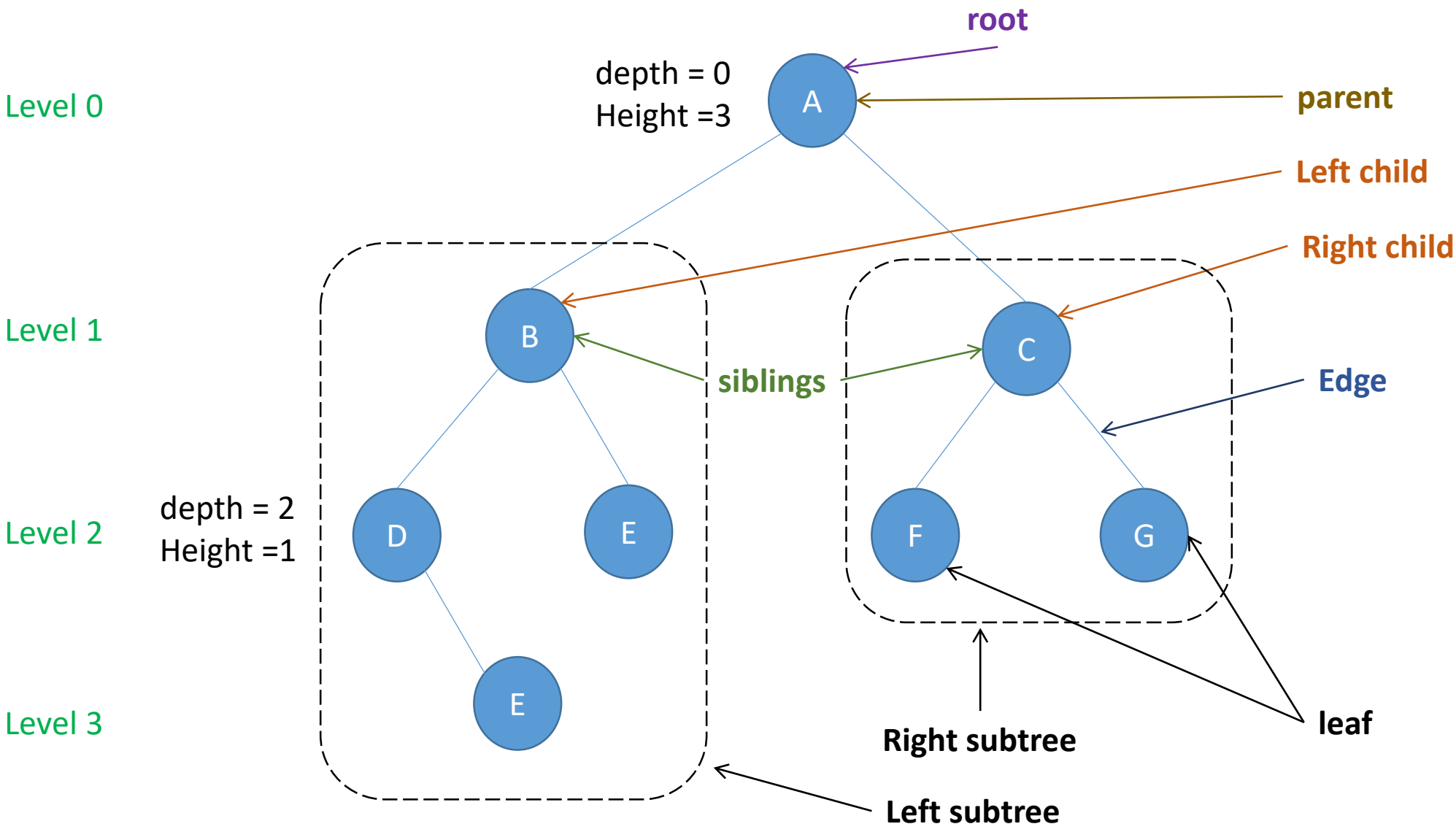
Tree data structure

- Productivity experts say that breakthroughs come by thinking “**nonlinearly**.”
- "Sequentiality is an illusion" *Kevin Skadron* http://www.cs.virginia.edu/~skadron//

Tree Terminology

- **Node** is a structure which may contain a value or condition, or represent a separate data structure. Each node of the tree is represented as a **circle.** Each node in a tree has zero or more child nodes.
- **Tree:** is a data structure made up of nodes or vertices and edges **without having any cycle** and stores elements hierarchically.
- **Parent:** A node that has a child is called the child's parent node. Each node has one parent with the exception of the root.
- **Child:** a node extending from another node.
- **Root:** The topmost node in a tree.
- **Sibling:** Two nodes are siblings if they have same parent.
- **Leaf:** a node with no children (external node).
- **Internal node**: a node with at least one child.
- **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.
- **Edge:** a pair of nodes (u,v) such that u is directly connected to v. Children are connected to the parent by an arrow from the parent to the child. An arrow is usually called a **directed edge** or a directed branch.
- **Path**: sequence of nodes. There is a **unique** path from the root to every node in the binary tree.
- **Length of a path:** # of edges in path.
- **Level**: The level of a node is defined by the number of connections between the node and the root.
- **Height of a node:** Number of edges on the longest path from the node to a **leaf.** Height of leaf nodes = 0.
- **Height of a binary tree:** Number of edges on the longest path from the root to a leaf (height of the root or maximum depth of any node). Height of an empty tree is -1.
- **Depth of a node** is the number of edges from the node to the **root** (number of ancestors). Depth of root = 0.


root
depth = 0
Height =3
A
parent
Level 0
Left child
Right child
Level 1
B
C
siblings
Edge
depth = 2
Height =1
Level 2
D
E
F
G
E
Level 3
Right subtree
leaf
Left subtree

SECTIONS SEARCH Chicago Tribune SUBSCRIBE

TUESDAY SEP. 29, 2015 TRENDING SPORTS SUBURBS BREAKING BUSINESS OPINION BEARS ADVERTISING

Habla espanol? Local employers say it's increasingly important

Employers are increasingly looking to hire applicants who can communicate effectively in more than one language. (Jos
Pelaez Inc / Getty Images/Blend Images)

Alexia Elejalde-Ruiz · Contact Reporter
Chicago Tribune

In a survey conducted in Summer 2015 by Northern Illinois University's Center for Government Studies, a **third** of employers said it is important to hire a recent college graduate who can communicate effectively in more than one language, and half said it will be important **five years** from now.

http://www.chicagotribune.com/business/ct-bilingual-employees-niu-0929-biz-20150928-story.html

Recursion

- A function that calls itself
- The function actually knows how to solve only the simplest case(s) (base case(s)).

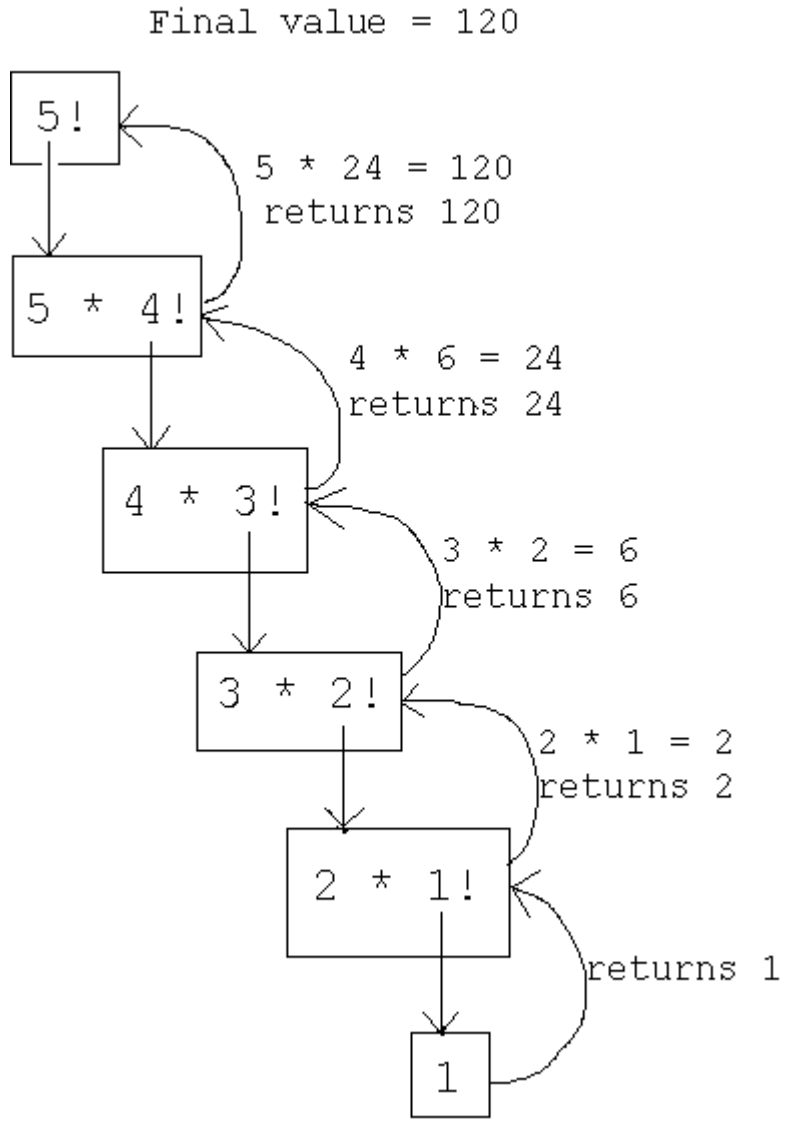
```
#include <iostream>

void countDown(int count)
{
 std::cout << "push " << count;
 countDown(count - 1);
}

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

What is the output?

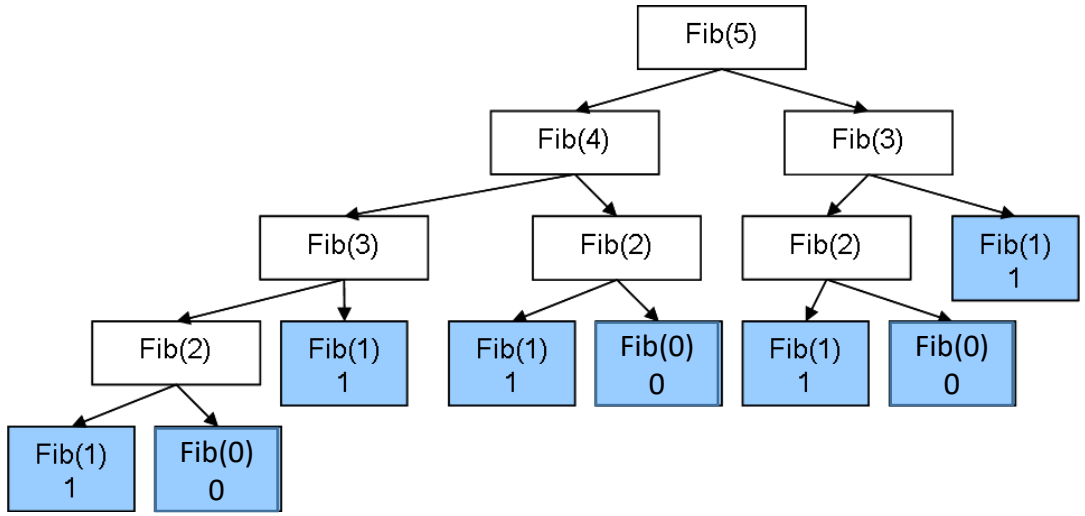
```cpp
#include <iostream>

void countDown(int count)
{
 std::cout << "push " << count << '\n';

if (count > 1) // termination condition
    countDown(count - 1);

 std::cout << "pop " << count << '\n';
}

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
```

```cpp
#include<iostream>
using namespace ::std;
long factorial(long number)
{
 if (number <= 1)
     return 1;
else
     return (number * factorial(number - 1));
}
int main()
{
  cout << factorial(5);
  return 0;
}
```



```
#include<iostream>
using namespace ::std;

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;
}
```



Fibonacci of 5 = 5

Write a recursive function print_backwards, that receives a vector (e.g., 2,4,7) and prints it in reverse order (e.g., 7,4,2).

```
#include<iostream>
#include<vector>
using namespace std;
void print_backwards(vector<int> v)
{









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

Write a recursive function print_backwards(), that receives a vector (e.g., 2,7,4) and prints it backwards on the screen (e.g., 4,7,2).

```
#include<iostream>
#include<vector>
using namespace std;
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;}
```

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

```
#include<iostream>
#include<vector>
using namespace std;
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**

- 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.

```
struct product
{
   int quantity;
   double price;
};
product apple, orange;  // Many objects can be declared from a single structure type.
apple.quantity = 4;     // Access member of a structure.
```

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
```

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

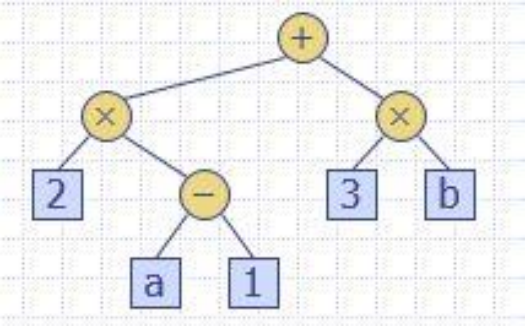
- More details http://www.cplusplus.com/doc/tutorial/structures/

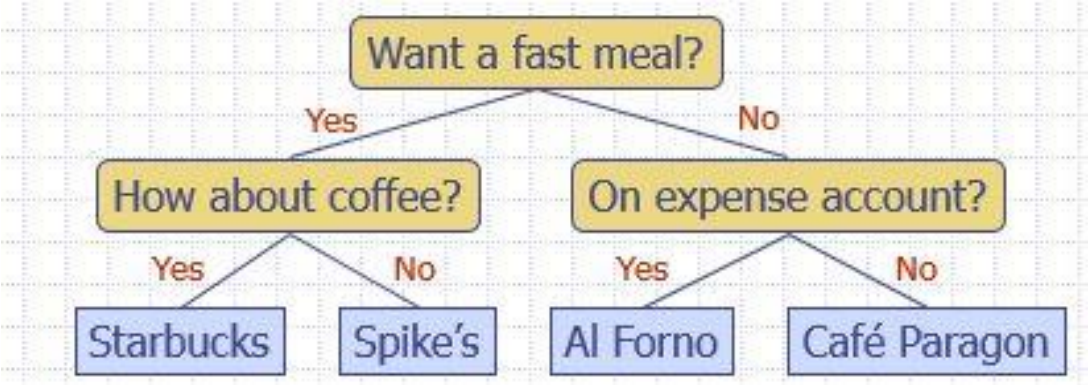
```
#include <iostream>
using namespace std;
struct Time {
int hour;
int minutes;
int seconds;
};
void main()
{
 Time current, *ptr;
 current.hour = 11;
 current.minutes = 2;
 current.seconds = 30;


cout << " The time now = " <<current.hour << ":"<< current.minutes <<":"<<
current.seconds;


 ptr = &current;
 ptr->hour++;
 cout << "\n The time after 1 hour = " << current.hour << ":" << current.minutes <<
":" <<       current.seconds;
}
```

```
The time now = 11:2:30
The time after 1 hour = 12:2:30
```

Binary Tree

- Every node in a **binary tree** has **at most two children**.
- Applications:
  - arithmetic expressions (e.g., (2 × (a - 1) + (3 × b)))
  - decision processes (e.g., dining decision)
  - searching



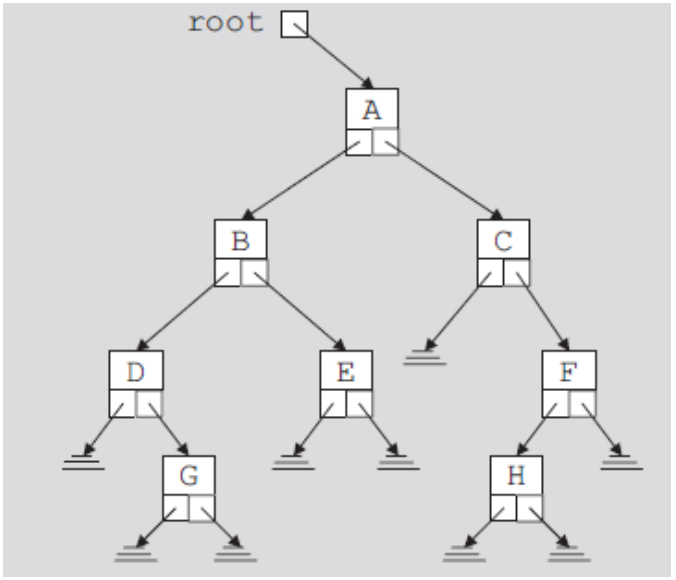


Properties of Binary Trees

- n = number of nodes
- ne = number of external nodes
- ni = number of internal nodes
- h = height of a binary tree
- Then the binary tree has the following properties:
  - $h+1 \leq n \leq 2^{h+1} - 1$
  - $1 \leq ne \leq 2^{h}$
  - $h \leq ni \leq 2^{h} - 1$
  - $\log(n+1) - 1 \leq h \leq n-1$

- For each node
  - The data stored in **`data`**
  - A pointer to the left child stored in **`left`**
  - A pointer to the right child stored in **`right`**

```
class Node
{
 int data;
 Node *left;
 Node *right;
};
```



- Pointer to **root** node is stored outside the binary tree. The root node defines an entry point into the binary tree.

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

```
#include<iostream>
using namespace std;
class Node
{
public:
 int data;
 Node* left;
 Node* right;
};
/* Helper function that allocates a new node with the given data and NULL left and right
pointers.*/
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);
}
```

Replace with constructor

```
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;
}
```

Replace with insert function

```
/* Compute the "maxDepth" of a tree -- the number of edges along the longest path
from the root node down to the farthest leaf node.*/
int maxDepth(Node* n)
{
 if (n == nullptr)
     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 just
  // return 1 + max(maxDepth(n->left), maxDepth(n->right));
} }
```

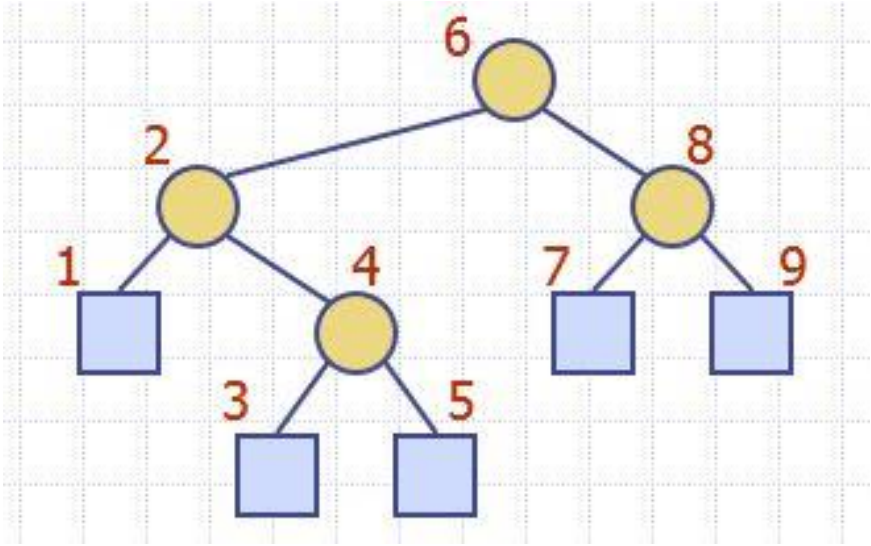
Binary Tree Traversal

- The item insertion, deletion, and lookup operations require that the binary tree be traversed or **visit** each node of the binary tree.
- Must **start** at the **root**, and then we can first visit the
  a) node *or*
  b) subtrees
- These choices lead to different recursive traversal algorithms
  1) Depth-first traversal
     1) Inorder
     2) Preorder
     3) Postorder
  2) Breadth-first traversal

1) 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

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



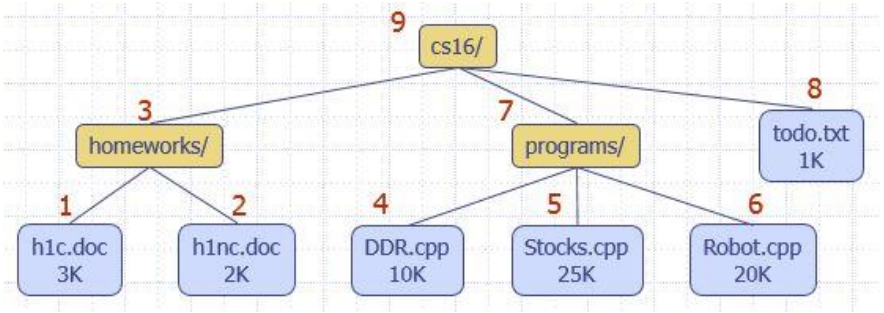
2) Preorder traversal (NLR)

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

```
void preorder(Node *p)
{
 if (p != NULL)
    {
     cout << p->data;
     preorder(p->left);
     preorder(p->right);
    }
}
```

3) Postorder traversal (LRN)

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



- **Application:** compute space used by files in a directory and its subdirectories

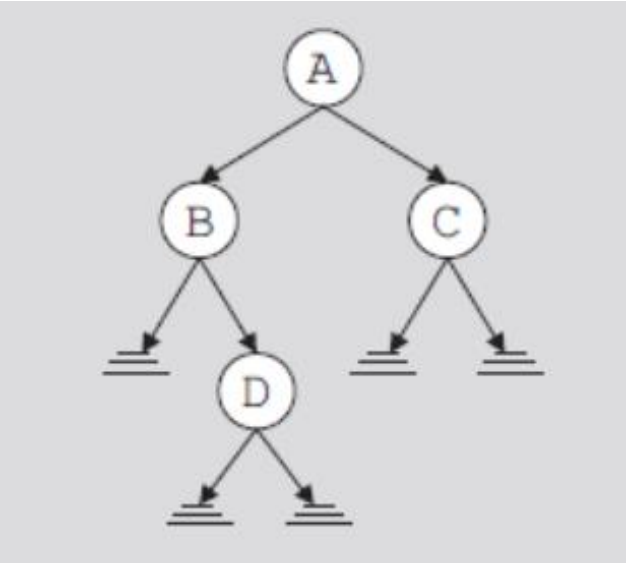
```
void postorder(Node *p)
{
 if (p != NULL)
    {
     postorder(p->left);
     postorder(p->right);
     cout << p->data;
    }
}
```

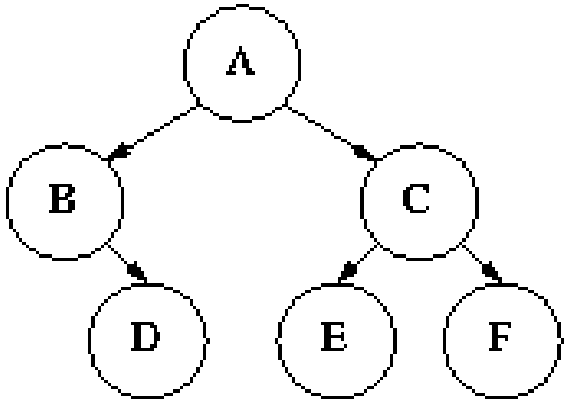
4) Breadth-first traversal (level-order): the root is processed first, all its children are processed next, then all of their children, etc. down to the bottom level.

```
void levelOrder(Node *p) {
q.push(p); // Push the root

while (!q.empty())
{
 Node *v = q.front();
 cout << v->data;

 if (v->left != NULL)
     q.push(v->left);

 if (v->right != NULL)
    q.push(v->right);

    q.pop();
}}
```

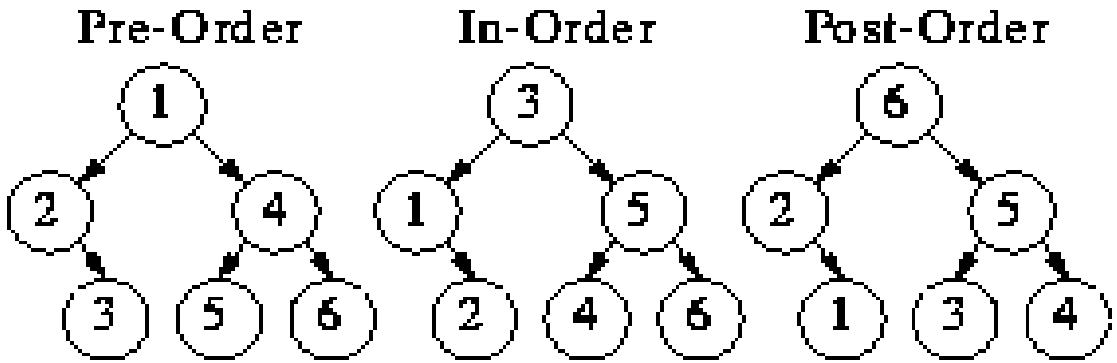
- Exercise: Traverse the following binary tree: (a) in inorder; (b) in preorder; (c) in postorder. Show the contents of the traversal as the algorithm progresses.

Inorder sequence: B D A C
Preorder sequence: A B D C
Postorder sequence: D B C A
level-order sequence: A B C D





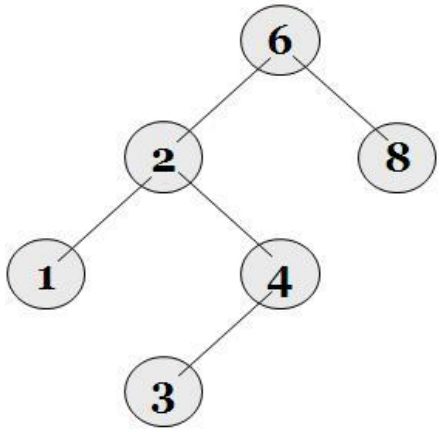
- Pre-Order: A-B-D-C-E-F
- In-Order: B-D-A-E-C-F
- Post-Order: D-B-E-F-C-A

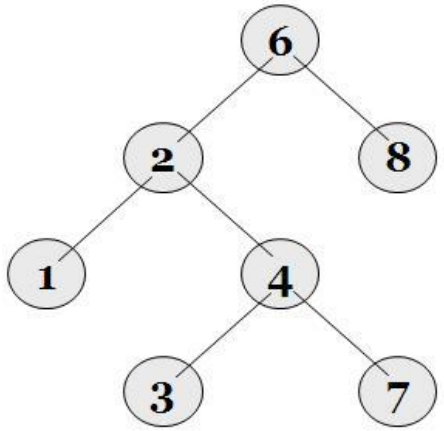


Binary Search Tree (BST)

- The binary search tree is a binary tree with the following properties:
  - The **key in each node** must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree.
  - Each subtree is, itself, a binary search tree.

A *binary search tree*





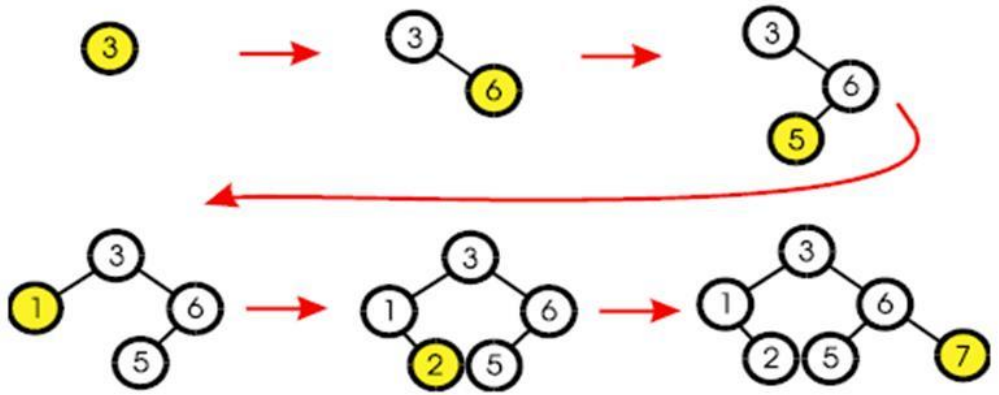
Not a *binary search tree*,
but a *binary tree*

Operations on Binary Search Trees

- Insertion
- Search
- Traversal
- Deletion
- Find minimum
- Find maximum
- Successor: finds the **next** item in the tree that is greater than the current node
- Predecessor: finds the **previous** item in the tree that is smaller than the current node

Inserting an item in BST

- The **first** value inserted goes at the **root**.
- Every node inserted becomes a **leaf**.
- Insert **left** or **right** depending upon **value**.
- **Duplicate** values:
  1. Can be stored in another data structure (e.g., **list**).
  2. Can **all** be kept in the **left** subtree, or all in the **right** subtree (the choice must be same for the whole implementation).
  3. Need **extra operations** (e.g., getAll and removeAll)



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

The ptr parameter is not simply a pointer to a Node, but a **reference to a pointer** to a Node. This means that any action performed on ptr is actually performed on the argument that was passed into ptr.

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

```
#ifndef HEADERFILE
#define HEADERFILE
// code
#endif
```

- Example https://en.wikipedia.org/wiki/Include_guard

```
#include <iostream>
using namespace std;
class Point
{
 public:
 int a, b;

Point() // Default Constructor
{
 a = 10;
 b = 20;
}};

int main()
{
 // Default constructor called automatically when the object is created
 Point c;
 cout << "a: " << c.a << endl << "b: " << c.b;
 return 0;
}
```

```cpp
#include<iostream> // A program to illustrate parameterized constructors
using namespace std;
class Point
{
 private:
  int x, y;

public:
 Point(int x1, int y1) // Parameterized Constructor
 {  x = x1;
   y = y1;}
 int getX() { return x; }
 int getY() { return y;}
};

int main()
{
 Point p1(10, 15); // Constructor called
 // Access values assigned by constructor
 cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
return 0;
}
```

C++ Enumeration

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

```cpp
#include <iostream>
using namespace std;

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
```

C++ Templates

- Templates are the foundation of **generic programming**, which involves writing code in a way that is independent of any particular type.

- A template is a blueprint or formula for creating a generic class or a function. The library containers like **iterators** and **algorithms** are examples of generic programming and have been developed using template concept.

- There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

Function Template

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

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

- Here, type is a placeholder name for a data type used by the function. 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.
- When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). *Next Slide*

```cpp
#include <iostream>
#include <string>
using namespace std;
template <class 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
```

Function template with more than one type parameter

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

Class Template

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

```
template <class type> class class-name {
.
.
}
```

- 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
#include <iostream>
#include <vector>
#include <string>

using namespace std;

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
};
```

```
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();
}
```

```
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
```

Exercise

- Write a function (mySwap) to swap two variables of generic type passed-by-reference.
- Call the function 3 times using:
  - mySwap (int,int)
  - mySwap (char,char)
  - mySwap (double,double)
- .. and print the new values after each call.

```cpp
#include <iostream>
using namespace std;

template <typename T>
void mySwap(T &a, T &b) {
 T temp;
 temp = a;
 a = b;
 b = temp;
}

int main() {
 int i1 = 1, i2 = 2;
 mySwap(i1, i2);   // Compiler generates mySwap(int &, int &)
 cout << "i1 is " << i1 << ", i2 is " << i2 << endl;

 char c1 = 'a', c2 = 'b';
 mySwap(c1, c2);   // Compiler generates mySwap(char &, char &)
 cout << "c1 is " << c1 << ", c2 is " << c2 << endl;

 double d1 = 1.1, d2 = 2.2;
 mySwap(d1, d2);   // Compiler generates mySwap(double &, double &)
 cout << "d1 is " << d1 << ", d2 is " << d2 << endl;
}
```

Virtual

- If there are member functions with **same name in base class and derived class**, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as **polymorphism** which is one of the important features of 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 member is a member function that can be redefined in a derived class**, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the **virtual** keyword.

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
protected:
int width, height;
public:
void set_values(int a, int b)
{
 width = a; height = b;
}
virtual int area()
{
 return 0;
}
};
```

```
class Rectangle : public Polygon {
public:
int area()
{
 return width * height;
}
};

class Triangle : public Polygon {
public:
int area()
{
 return (width * height / 2);
}
};
```

```
int main() {
Rectangle rect;
Triangle trgl;
Polygon poly;
Polygon * ppoly1 = &rect;
Polygon * ppoly2 = &trgl;
Polygon * ppoly3 = &poly;
ppoly1->set_values(4, 5);
ppoly2->set_values(4, 5);
ppoly3->set_values(4, 5);
cout << ppoly1->area() << '\n';
cout << ppoly2->area() << '\n';
cout << ppoly3->area() << '\n';
return 0;
}
```

```
20
10
0
```

- 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** members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if **virtual** is removed from the declaration of area in the example above, all three calls to **area** would return zero, because in all cases, the version of the base class would have been called instead.
- A class that declares or inherits a virtual function is called a polymorphic class.

Copy Tree

- Shallow copy of the data
  - Obtained when value of the pointer of the root node used to make a copy of a binary tree
- Identical (deep) copy of a binary tree
  - Need to create as many nodes as there are in the binary tree to be copied
  - Nodes must appear in the same order as in the original binary tree
- Function `copyTree`
  - Given a pointer to the root node, makes a copy of a given binary tree.

```
void copyTree(Node* &copiedTreeRoot, Node* otherTreeRoot)
{
if (otherTreeRoot == NULL)
    copiedTreeRoot = NULL;
else
    {
     copiedTreeRoot = new Node (otherTreeRoot->data);
     copyTree(copiedTreeRoot->left, otherTreeRoot->left);
     copyTree(copiedTreeRoot->right, otherTreeRoot->right);
    }
}
```

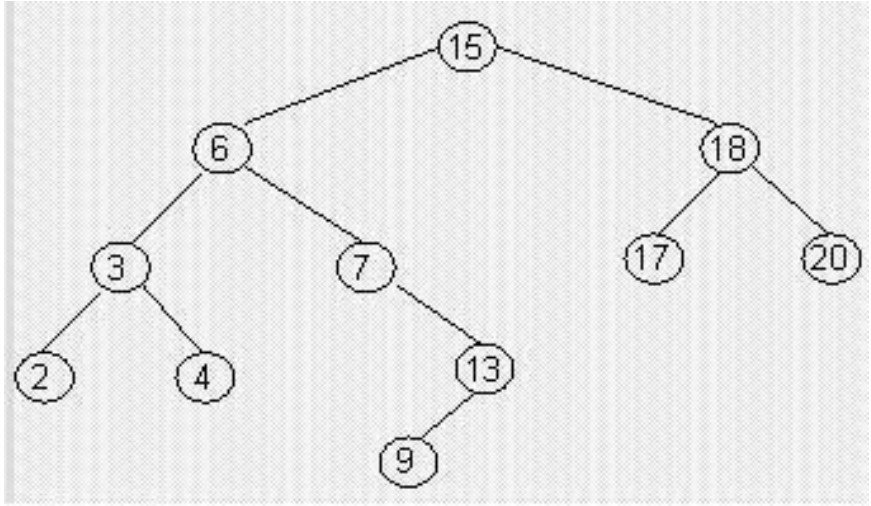
Successor

- 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 successor(x) is the **left most** element in that **right sub tree**.

(2) If x has no right sub tree, then successor(x) is the **lowest ancestor** of x (above x on the path to the root) that has x in its **left sub tree**.

- Successor(15) = 17
- Successor (7) = 9
- Successor (13)= 15



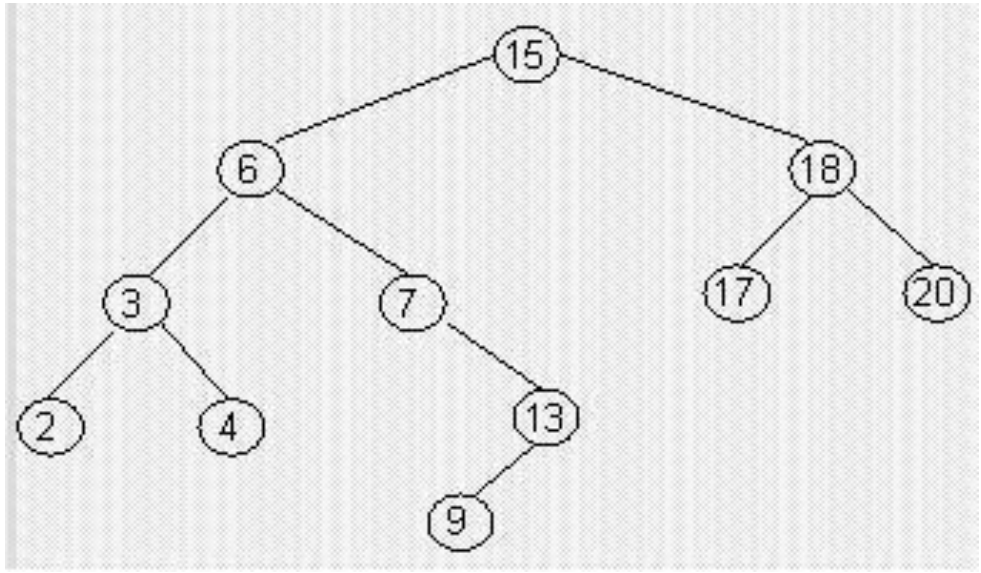
Predecessor

- The predecessor is the node that has the largest key smaller than that of x.

(1) If x has a left sub tree, then the predecessor is the **right most** element of that **left sub tree.**

(2) If x has no left sub tree, then predecessor (x) is the **lowest ancestor** of x (above x on the path to the root) that has x in its **right sub tree**.

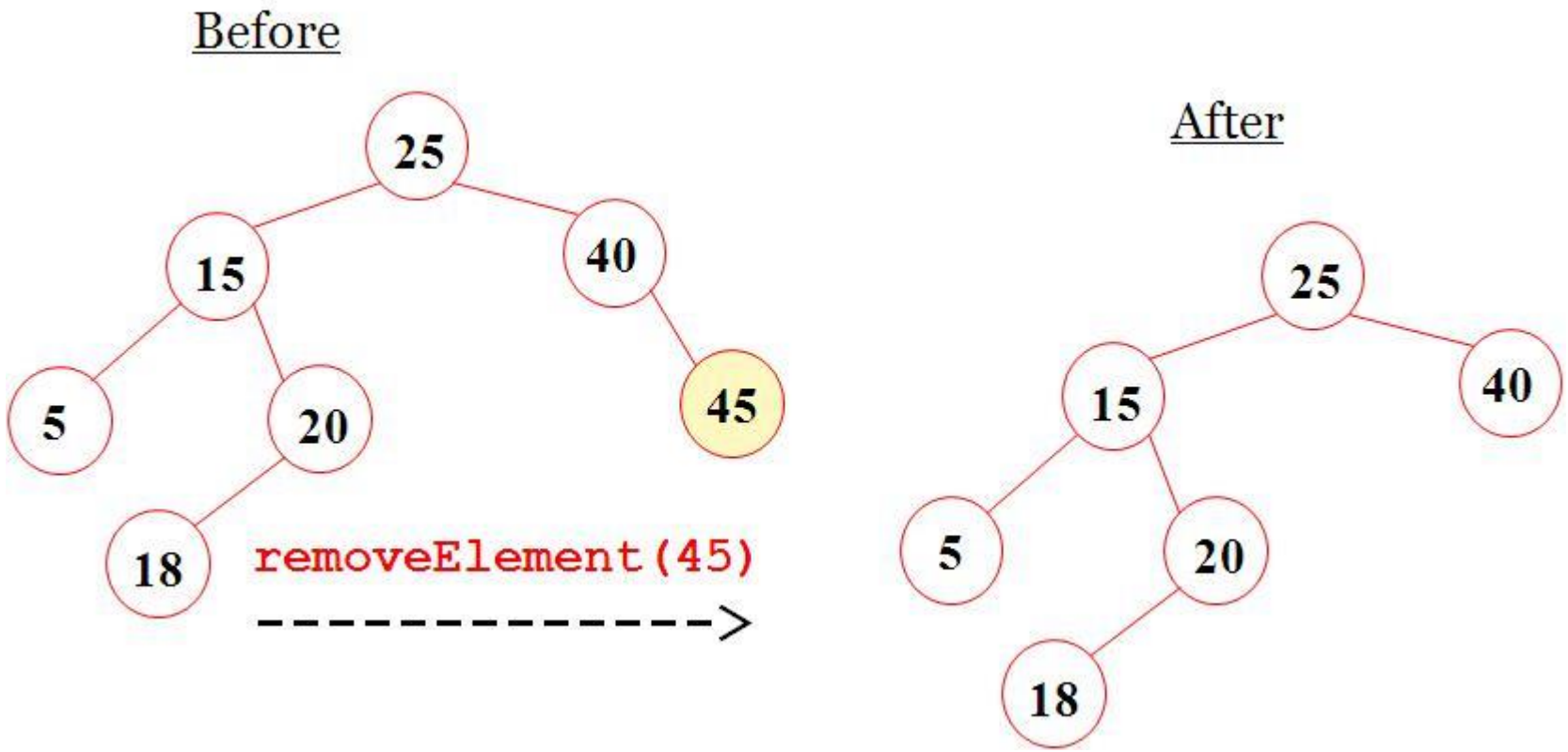
- Predecessor(6) = 4
- Predecessor(15) = 13
- Predecessor(17) = 15



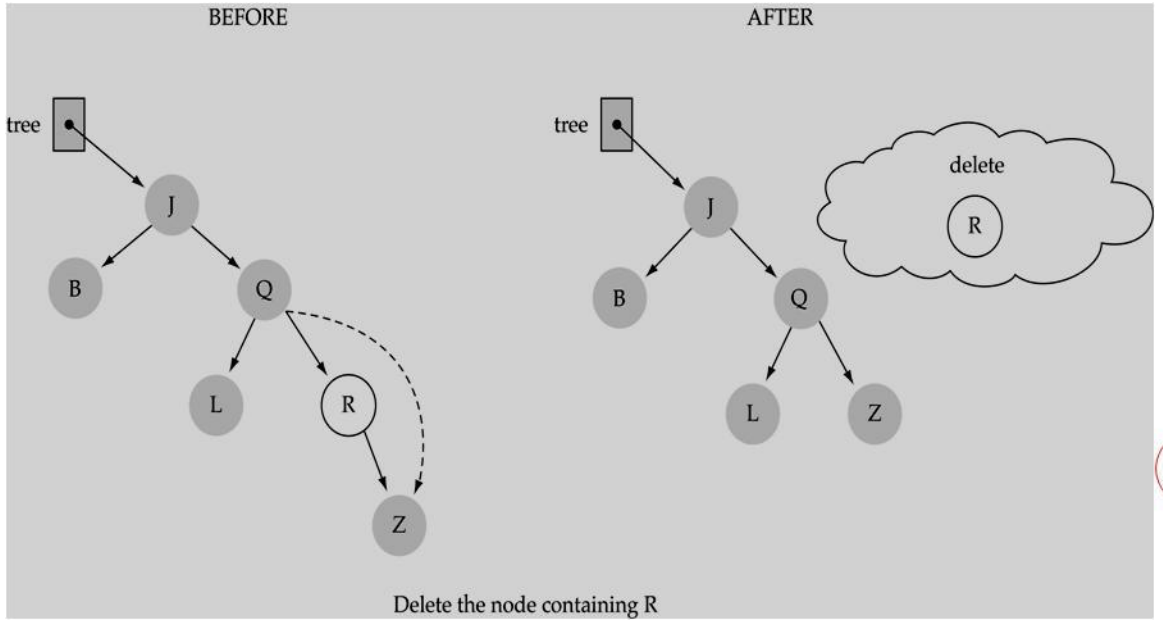
Deleting from BST

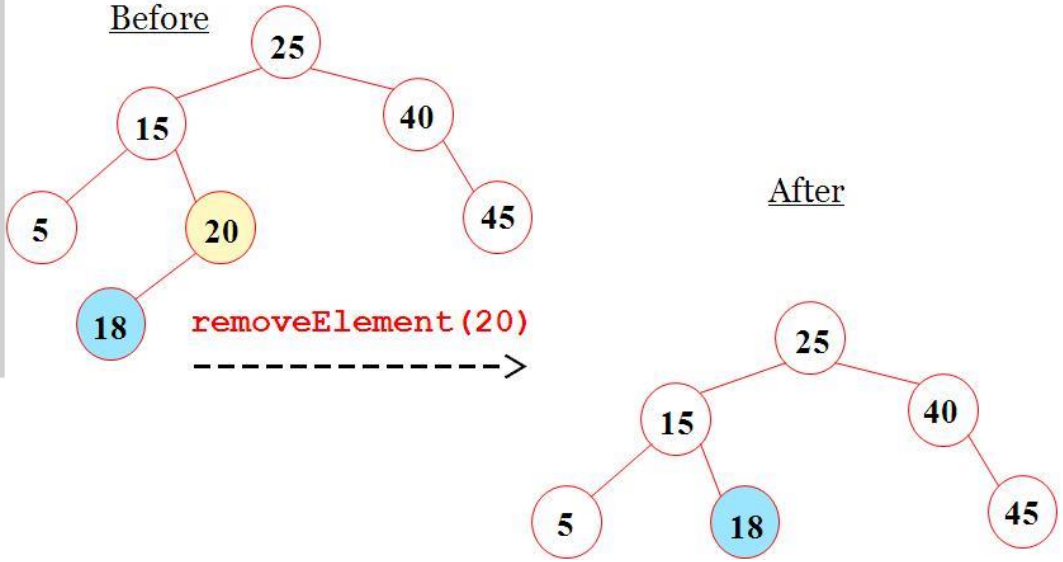
- First, find the item; then, delete it.
- Binary search tree property must be preserved.
- We need to consider three different cases:
  1. Deleting a leaf
  2. Deleting a node with only one child
  3. Deleting a node with two children

(1) Deleting a leaf

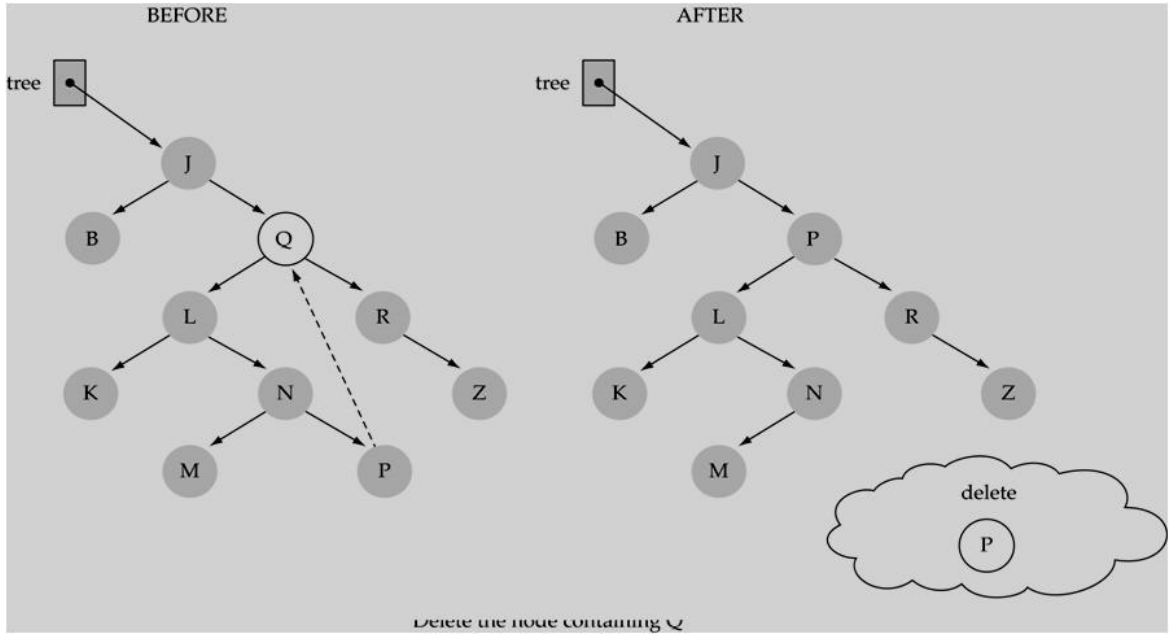


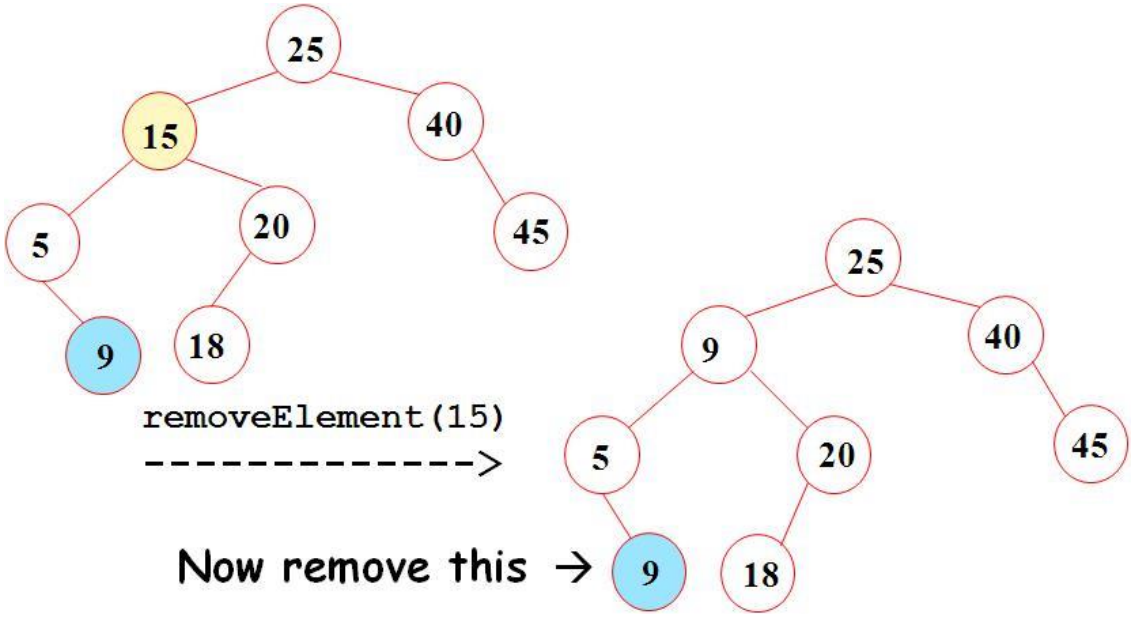
(2) Deleting a node with only one child

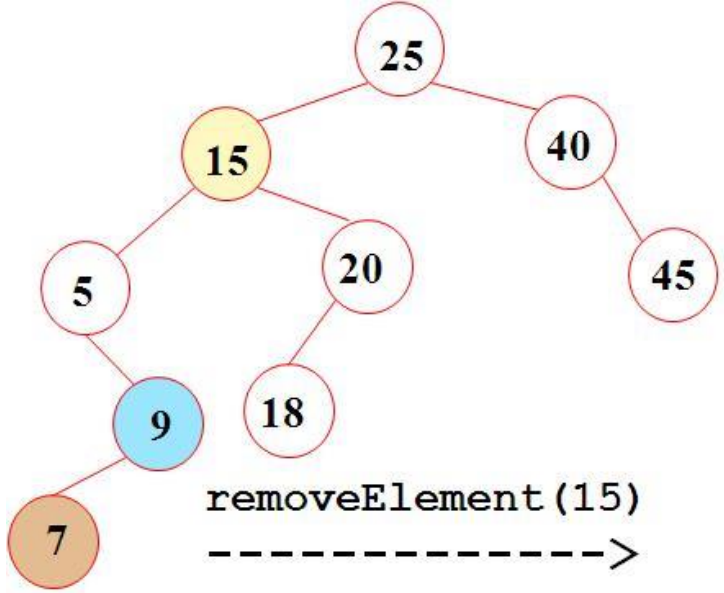




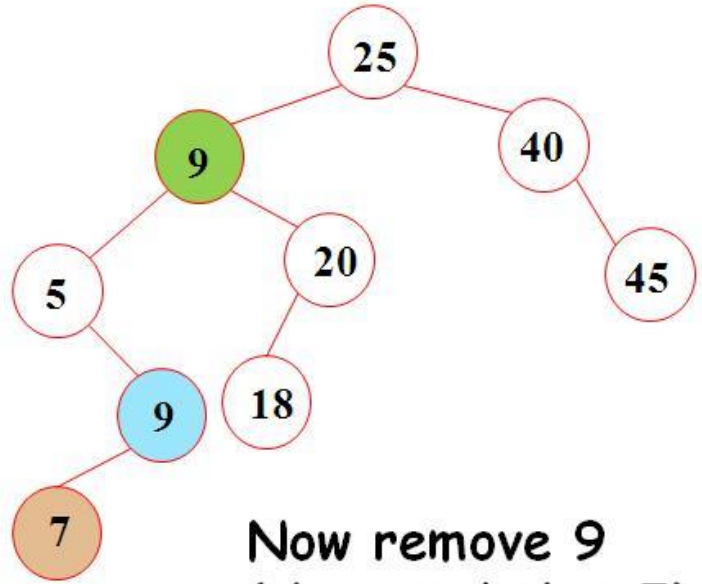
(3) Deleting a node with two children



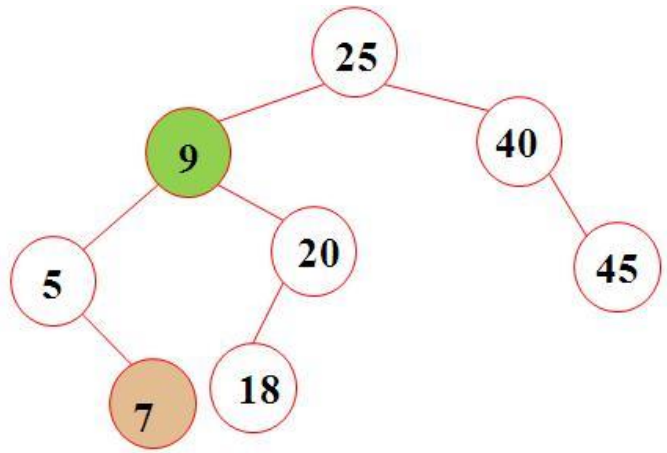




removeElement(15)
---------->



Now remove 9
(the one below 5)



After deleting 15, and then 9; the tree becomes

```
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 that ptr argument is a **reference to pointer**. Like the insert function, the deleteNode function must have access to an actual pointer in the tree, to the node that is being deleted (not just a copy of the pointer).

```
void deleteNode(Node  *&ptr)
{
 int  data;
 Node *tempPtr = ptr;
if (ptr->left == NULL) {
   // Reattach the right child or NULL
    ptr = ptr->right;
    delete tempPtr;
}
 else if (ptr->right == NULL) {
    ptr = ptr->left;
    delete tempPtr;
}
 else {
    getPredecessor(ptr->left, data);
    ptr->data = data;
    findNodeToDelete(ptr->left, data);
 }
}
```

```
void getPredecessor(Node  * pt, int  &data)
{
  while (pt->right != NULL)
         pt = pt->right;

  data = pt->data;
}
```

Exercises

(1) Given a sequence of numbers: 11, 6, 8, 19, 4, 13, 5, 17, 43, 49, 16, 31, 32

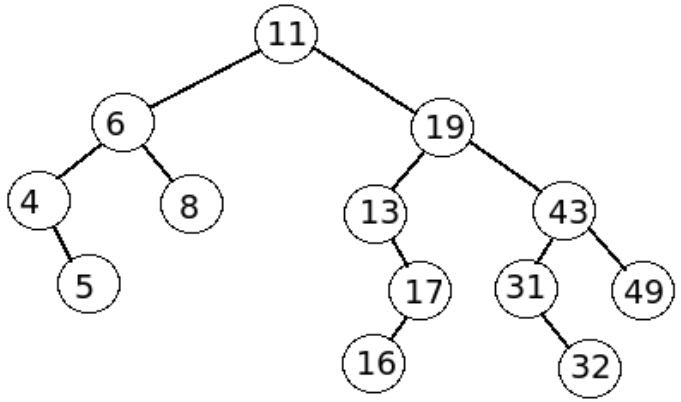
a) Draw a binary search tree by inserting the above numbers from left to right

b) What is the height of the above tree?

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

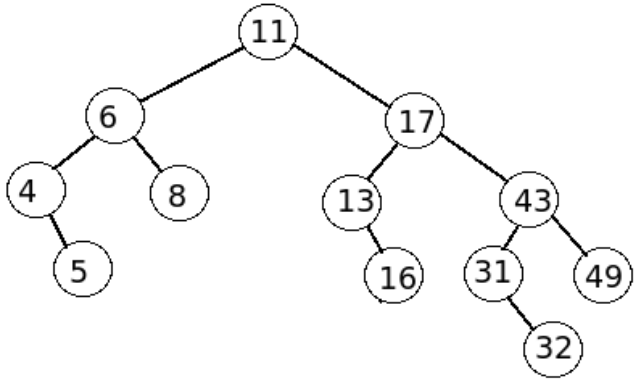
Answers

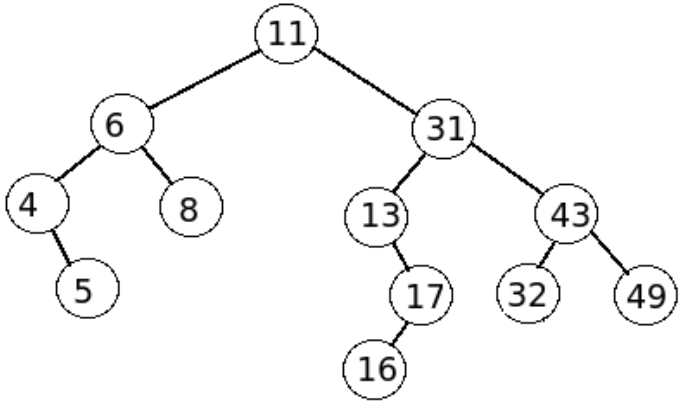
(a)



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

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





- Which of the following traversals always gives the sorted sequence of the elements in a BST?

a) Preorder

b) Inorder

c) Postorder

d) depends on how the elements are inserted

- In a Binary Search Tree, the largest element must

a) be the root.

b) be a leaf.

c) have at least one child.

d) have at most one child.

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

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.
  - Google http://www.glassdoor.com/Interview/Define-binary-search-tree-Develop-a-procedure-to-verify-a-binary-search-tree-QTN_36765.htm
  - Amazon http://www.glassdoor.com/Interview/First-explain-what-a-tree-then-binary-tree-then-a-binary-search-tree-is-Now-implement-a-function-that-verifies-whether-a-QTN_228019.htm
  - http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/
- 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
  - http://www.geeksforgeeks.org/find-minimum-depth-of-a-binary-tree/
- 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
  - http://www.geeksforgeeks.org/diameter-of-a-binary-tree/

- Using recursion to traverse a binary tree
  - Microsoft http://www.glassdoor.com/Interview/using-recursion-to-traverse-a-binary-tree-QTN_7009.htm
- Print the nodes on a tree level
  - Bloomberg L.P. http://www.glassdoor.com/Interview/gave-me-a-tree-of-3-level-and-provided-me-a-number-that-contains-the-level-number-and-asked-me-to-code-a-program-that-would-QTN_609076.htm
- 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 http://www.glassdoor.com/Interview/Write-a-function-that-takes-2-arguments-a-binary-tree-and-an-integer-n-it-should-return-the-n-th-element-in-the-inorder-t-QTN_325122.htm
- find lowest common ancestor of 2 nodes in a binary tree
  - Symantec http://www.glassdoor.com/Interview/find-lowest-common-ancestor-of-2-nodes-in-a-binary-tree-QTN_174437.htm
  - http://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/

Useful resources

- Binary Search Trees (BSTs) - Insert and Remove Explained https://www.youtube.com/watch?v=wcIRPqTR3Kc
- 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/