Recursive Function

Basic Rules for a recursive function

1) Base Case: must always have a base case in order to make a recursive call 2) Must stop at some point

Examples

Fibonacci

$$fib_n = fib_{n-1} + fib_{n-2}$$

```
int sum(A,n)    // finds the sum of n elements

// Recursive Example
int sum(A,n){
        if(n==0) return 0;
        return sum(A, n-1) + A[n-1];
}
```

#cpp #code #recursion #functions

CPP_Review

Classes and Objects

- Classes & Objects
- *Classes define abstract characteristics of a type*
- *Members can be*
  - Data variables
  - Functions & Constructors
- *Object*
  - Instance of a class
  - Classes and Objects II
- *Information Hiding Labels*
  - Private
  - Public
  - Protected
- *Default Parameters*
  - Parameter to the constructor is optional
- *Explicit Constructor*
  - Avoids automatic conversion
  - Functions & Constructors
  - Multi-File Compilation
  - Composition and aggregation
- *Constant Member functions*
  - Examines but does not change the state of the object
  - Called accessors
- *Interface is defined through the .h files*
  - `#include` in the cpp file
  - Also referred to as declaration
- *Preprocessor commands*
  - Guards against multiple inclusions of .h files

```
// Preprocessor Statements
#ifndef _NAME_OF_FILE_
#define _NAME_OF_FILE_
```

```
/*
        Code Goes Here
*/

#endif
```

- *Scope-resolution operators*
    - symbolized by the `::`
    - To identify the corresponding class to each function
    - Function signatures must match in the definition and the implementation file

```
// dog.h *Implementation File*

class dog{
public:
        dog();
        void setName(string n);
private:
        string name;
};
```

```
// dog.cpp *Definition File*

dog::dog(){
        name = "Jerry"; // Default Name
}
void dog::setName(string n){
        name = n;
}
```

- *Objects are declared like primitive data types*
- *Standard Vector Class*
    - Gives `size()` function
    - Can be assigned using =
- *Standard String Class*
    - Compares with `==, <, etc`
- *Keyword auto*
    - You do not need to specify the type

```
auto i = 20
auto itr = vect1.begin();
```

- auto may not be used in some cases
- *Pointer variable*
    - stores the address of another obj in memory
    - before the variable name indicates a pointer declaration
    - `type * variableName;`
    - Pointers are uninitialized when declared, this may result in bugs

#cpp #review #classes #objects #code

Recitation_1

Tar File

- To tar a directory:
  - `tar cvf file_name.tar directory_name/`
  - `c` : create a tar file
  - `v` : show progress
  - `f` : specify the name of the file
  - `tf` : view the contents of a tar
  - `tar xvf file_name.tar` will untar a file

Make File

```
CXX=g++
        hello_make:
                $(CXX) fileName.cpp -o fileName.x
```

Reading Char

```
char c;
while ((c = std::cin.get()) != EOF) {
        std::cout << c;
}
```

Reading Lines

```
string line;
ifstream myfile("file.txt");
if (myfile.is_open()){
        while(geline(myfile, line)){
                cout << line << "\n" << endl;
        }
        myfile.close();
}
else { cout << "File could not be opened" << endl; }
```

Variable Scope and Program

- L value - Associated with non-temporary objects
- `string str = "hello";`
- `str` - L Value
- `"Hello"` - R Value, Temporary

```
string x= findMax(a);
string & y = x;
cout << y << endl;
```

- R values can be moved (we do not need the value)
- L values can be copied

```
// Refrence to an R Value
string && str2 = "Hello";
```

Parameter Passing

Call by Value

- Copies the value of the parameter being passed
- Called function can modify the parameter but not the initial

Pass by Reference

- Can modify the original value
- Faster because you dont need to make a copy

Pass by Reference

- Cannot modify the value
- Should be used for large values

Call by rvalue refrence

- Move rvalue instead of copy
- Normally more efficient

```
vector <string> v("hello", "world");
cout << randomItem(v) << endl;  // L Value
cout << randomItem({"Hello", "World"}) << endl; // R Value
```

Given File Example

```
#include <iostream>
#include <vector>
#include <string.h>

using namespace std;


double ave( const vector<int> & arr, int n, bool & errorFlag)
{

    int sum = 0;
        for( int i = 0; i < arr.size( ); ++i )
          sum += arr[i];

        n = 100;
        errorFlag = true;
        return ((double)sum)/arr.size();
}


int main( )
{
  int nn = 5;
  bool err = false;

  vector<int> myArray {1, 2, 3, 4, 5};
  double average = 0.0;

  cout << "Before: average = " << average << ", nn = " << nn << ", err = "
<< err << endl;

  average = ave(myArray, nn, err);

  cout << " After: average = " << average << ", nn = " << nn << ", err = "
<< err << endl;

  return 0;
}
```

Return Passing

Return by Value

- makes a copy of a variable returned

Return by Reference

- Return a reference of the variable returned

Return by Const Reference

- Return the reference of the variable returned
- Return value cannot be modified by caller

Lifeline extended beyond function call for by const reference and reference.

Big Five in C++

- Five special functions provided in all c++ classes
    - Destructor constructor
    - Copy constructor
    - Move constructor
    - Copy assignment operator =
    - Move assignment operator =

A constructor is called whenever..

- An object goes out of scope
- Delete is called
  Invoked during
- Declaration
- Call by value, and return by value
- Not in Assignment operator

Problem with Defaults

- Usually dont work when data member is a pointer type
- If a class contains pointers as member variables and you want two copies of objects pointed at

Variable Scope

- If your program sometimes doesn't work. Then...
  You Have Bugs

- a local variable only exists within its scope

Templates

- Type independent patterns
- Allows for reusable code, and generic programming
- The template declaration indicates that Comparable is the template argument it can be replaced by any type to produce a real function.

```
// Return the maximum item in the array a
template<typename Comparable>
const Comparable& findMax(const vector<Comparable>& a){
        int maxIndex = 0;
        for(int i= 1; i < a.size(); i++){
                if(a[maxIndex] < a[i]){
                        maxIndex = i;
                }
        }
        return a[maxIndex];
}
```

For Example

- If a user needs to use the same function to hold a string, as well as integers. A Code example can be seen above.
- Also covered in COP3330 notes

Function Objects

- Objects whose primary purpose is to define a function
- Using operator overloading : `operator ()`

Memory Cell

- Can be used for any type object

#code #functions #pass-by-ref #assignment #l-value #r-value #defaults #memberFunctions #memberVariables #declaration #cpp

Math Review

Exponent

$$X, X^2, = X * X, X^3 = X * X * X$$

```
// O(N) Time Complexity
for(int i=0; i< N, i++){
        A[i] = 0;
}
```

$X^A = B$ if and only $log_x B = A$

Theorom 1.1

$$log_a B = (log_c B)/(log_c A)$$

$log(x)$ refers to base 2 in our case. *For Binary*

Algorithm Analysis

Complexity Analysis

- Establishing the relationship between the input size and the algorithm/ program time (and/or space) requirement.
    - Estimate the time and space requirement for a given input size
    - Compare algorithms
- Time Complexity: counting operations
    - Count the number of operations that an algorithm will perform
- Asymptomatic analysis
    - The Big O notation
    - How fast time/space requirements increase as the input size approaches infinity

```
// number of outputs
// t(n) = n
for(i = 0; i < n, i++){
        cout << A[i] << endl;
}

// number of comparisons
// t(n) = n-1
template<class T>
bool IsSorted(T *A, int n){
        bool sorted = true;
        for(int i=0; i<n-1; i++){
                if(A[i]) > A[i+1]){
                        sorted = false;
                }
        }
        return sorted;
}
```

Algorithm analysis covers the worst case (most of the time).
The average case is more useful, however, it is more difficult to calculate.
The complexity of a function is its input size. For instance...
$t(n) = 1000n$ vs. $t(n) = 2n^2$
if n doubled...
$t(n) = 1000n...1000 * 2n/1000n = 2$

- Time Will Double
  $t(n) = 2n^2 ... 2(2n^2) = 4n^3$
- Time increases by 4x

Scaling Analysis

- The constant factor does not change the growth rate and can be ignored
- We can ignore the slower-growing terms.
  - Ex. $n^2 + n + 1$ ... n^2
  - capturing $O(n^2)$

Example Problem

Algorithm 1: $t_1(n) = 100n + n^2$

- insert - n, delete - $log(n)$, lookup - 1
  Algorithm 2: $t_2(n) = 10n^2$
- insert - $log(n)$, delete - n, lookup - $log(n)$
  Which is faster if an application has as many inserts but few deletes and lookups?

Asymptotic Complexity Analysis

- Compares the growth rate of two functions
- Variables & Values - Nonnegative integers
- Dependent on eventual (asymptotic) behavior
- Independent of constant multipliers, and lower-order effects

Big "O" Notation

$f(n) = O(g(n))$
$iff \exists c, n_0 > 0 | 0 < f(n) < cg(n) \forall n >= n_0$

- if there exists two positive constants $c > 0$ & $n_0 > 0$ such that $f(n) \leq cgn(n)$ for all $n \geq n_0$

Big "Omega" Notation

$f(n) = \Omega(g(n))$
$iff \exists c, n_0 > 0 | 0 < cg(n) < f(n) \forall n >= n_0$

- $f(n)$ is asymptotically lower bounded by $g(n)$

Big "Theta" Notation

$f(n) = \theta(g(n))$
$iff \exists c_1, c_2, n_0 > 0 | 0 < c_1 g(n) < f(n) < c_2 g(n) \forall n >= n_0$

- $f(n)$ has the same long-term growth as $g(n)$

Examples

$f(n) = 3n^2 + 17$

$\Omega(1), \Omega(n), \Omega(n^2)$ -> lower bounds

- Chose $\Omega(n^2)$ because it is the closest to the lower bound
- Set $\exists$ to 3. So... $3(n^2)$
  $O(n^2), O(n^3)$ -> upper bounds
- Chose $O(n^3)$ because it is the closest to the upper-bound
  $\theta(n^2)$ -> exact bound
  Why f(n) != O(n)?

Transitivity

$f(n) = O(g(n))$ -> $(a <= b)$
$f(n) = \Omega(g(n))$ -> $(a >= b)$
$f(n) = \theta(g(n))$ -> $(a = b)$

Additive property

- If $e(n) = O(g(n))$ and $f(n)$ = $O(h(n))$
- Then...
  - $e(n) + f(n) = O(g(n)) + O(h(n))$

Function	Name
c	Constant
$log(N)$	Logarithmic
$log^2(N)$	Log-squared
N	Linear
$NlogN$	
N^2	Quadratic
N^3	Cubic
2^n	Exponential

Running time Calculations - Loops

```
for(j=0; j < n; ++j){
// 3 Atomics
}
```

- Each iteration has 3 atomics so $3n$

- Cost of the iteration itself ($c * n$, c is a constant)
- Complexity $\theta(3n + c * n) = \theta(n)$

Running time Calculations - Loops with a break

```
for(j=0; j < n; ++j){
// 3 Atomics
        if(condition) break;
}
```

- Upper bound - $O(4n) = (n)$
- Lower bound - $\Omega(4) = \Omega(1)$
- Complexity - $O(n)$

Complexity Analysis

- Find n = input size
- Find atomic activities count
- Find $f(n)$ = the number of atomic activities done by an input

Sequential Search

```
for(size_t i= 0; i < a.size() ;i++){
        if(a[i] == x){return}
}
// θ(n) time complexity
```

If then for loop

```
if(condition) i=0;
else
        for(j =0; j < n; j++)
                a[j] = j;
// θ(n) time complexity
```

Nested Loop Search

```
for(j =0; j < n; j++){
        // 2 atomics
        for(k =0; k<n; k++){
                // 3 atomics
        }
}
//θ(n^2) time complexity
```

Algorithm Analysis 2

Recursion

```
long factorial( int n ){
        if(n <= 1)
                return 1;
        else
                return n * factorial(n - 1);
// f(n) = 1 + t(n-1)
}
```

Expanding recurrence form results in a simulation of the function.

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

Which leads to...

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

$$T(n) > 2^{n-1/n} * T(1), T(n) = \Omega(2^{n-1/n})$$

Binary Search

```
int binary_search(const vector<int> &a, int X){
        unsigned int low=0, high = s.size()-1;
        while (low <= high){
                int mid = (low + high)/2;
                if(a[mid] < x)
                        low = mid + 1;
                else
                        return mid;
        }
        return NOT_FOUND
}
// log(n) time complexity
```

Euclid's Algorithm

- Find the GCD (greatest common divisor) between n and m
    - Given M >= N
- Why is it O(logN)

```
long gcd(long m, long n){
        while(n!=0){
                long rem = m %n;
                m = n;
                n = rem;
        }
        return m;
}
// LogN time complexity
```

Exponential

- Calculate x^0
- Complexity O(logN)

```
long pow(long x, int n){
        if(n == 0){
                return 1;
        }
        if(n == 1){
                return x;
        }
        if (isEven(n)){
                return pow(x * x, n/2);
        }
        else
                return pow(x * x, n/2) * x;
}
// O(logN)
```

Abstract Data Types

- mathematical abstractions of data types
- An ADT specifies
    - A set of objects
    - A set of operations on the data or subsets

> *Does not specify how the operations should be implemented*

Lists

- $A_0, A_1, A_2, \ldots A_{n-1}$
- The size of the list is N

Iterators

- Need a way to navigate through the items in a container
- A doubly linked list would need a different form than a simple linked list

```
// iterates through the array
for(int i=0; i != v.size(); i++){
        cout << v[i] << endl;
}
```

- A generalized type that helps in navigating any container
    - A way to initialize front and back
    - A way to move to the next
    - A way to detect the end

Getting an Iterator

- tells you the location of the objects
- can be written as

```
for(vector<int>::iterator itr=v.begin(); itr = v.end(); itr++)
        cout << *itr << endl; // must dereference

// use auto as well, C++11
auto itr = v.begin();
```

Adding / Removing

```
iterator insert(iteratorPos...)
iterator remove(iteratorPos...)

// Removing every other element from a list
auto itr = lst.begin();
while( itr != lst.end()){
        itr = lst.erase(itr);
        if(itr != lst.end())
                itr++;
}

// Before c++11
typename Container::iterator itr = lst.begin();
```

Vector in C++ STL

```
int size() // num of elements
void clear() // removes all elements
bool empty() // t or f if empty
void push_back() // put in back of vector
void pop_back() // remove from vector {size--}

// Operators
Object& operator[](index) // return obj index
Object& at (int index) // object at location
int capacity() // internal capacity
void reserve() // set new capacity
void resize() // change the size of a vector (need to copy)
```

Vector Class Template

- Can be copied
- The memory it uses automatically reclaimed
- Maintains primitive array

Public_Interface

O(N) Time Complexity

- Copy Constructor
- Copy assignment
- Destructor
- Clear()
- Erase(SI, EI)

O(1)

- Default constructor
- Move constructor
- Move assignment operator=
- push_front(t), push_back(t), insert(I, t)
- pop_front(), pop_back(), erase(I)
- begin(), end();
- front(), back();
- empty();

Read Only

```
// Read Only
bool operator== (const iterator & rhs) const;

bool operator!= (const iterator & rhs) const;

Object & operator* ( ) const;
// return a reference to the current value
```

Write Only

```
// Write only
iterator & operator++ ( ); // prefix

iterator operator++ ( int ); // postfix

iterator& operator-- ( ); // prefix
```

```
iterator operator-- ( int ); // postfix
```

List Implementation

- A doubly linked list with header and tail nodes as markers
- Linked Lists

Constructors and big-five

- Copy/move constructor
- Copy/move assignment operator
- Destructor

```
// Constructors
List();

List(const List &rhs);

List(List &&rhs);

List & operator=(const List &rhs);

List & operator=(List && rhs);

~List();

// Read-only accessor functions
int size() const;
bool empty() const;
```

Linked Lists

Singly Linked List

- push_front- Make the new node the head pointer, and make it point to the previous head node
- push_back- Make the tail node point to the new back node, and the back node point to null
- pop_front- Make the next node, following the head node, the new head node
- pop_back- Make the node before the tail node point to null, rather than the tail node

> *Efficiently = O(1) time complexity*

Doubly Linked List

- Contains a reference to the next element, as well as a reference to the previous element

```
// Insertion
auto I = Cities.begin();

for (; I != Cities.end(); ++I) {

        if (“Miami” == *I) {
        break;
        }
}

//Insert the new string

Cities.insert(I, “Orlando”);

// “Jacksonville”, “Tallahassee”, “Gainesville”, “Orlando”, “Miami”

// Remove Orlando
List<string>::iterator I = Cities.begin();

// auto I = Cities.begin();  // c++11
while( I != Cities.end()) {
if (“Orlando” == *I) {
        I = Cities.erase(I);
} else {
        I++;
}}
```

Node -

- Data Value
- Pointers to the previous and next element
- Defined within the List class, with limited scope

Creating A List

```
template <typename Object>

class List

{
  private:
    struct Node
    {
        Object  data;
        Node   *prev; // Points to previous Node
        Node   *next; // Points to next Node

        Node( const Object & d = Object{ }, Node * p = nullptr, Node * n = 
nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };
```

Insertion within List

```
iterator insert( iterator itr, const Object & x )

    {

        Node *p = itr.current;

        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ x, p->prev, p
} );

    }



  iterator insert( iterator itr, Object && x )

    {

        Node *p = itr.current;

        ++theSize;

        return iterator( p->prev = p->prev->next = new Node{ std::move( x
), p->prev, p } );

    }
```

Empty List

```
  private:
    int   theSize;
    Node *head;
    Node *tail;
    void init( )
        // Doubly Linked List Init
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail; // Head -> &Tail
        tail->prev = head; // Tail (previos) -> &Head
    }

};
```

Erase Node

```

iterator erase( iterator itr )
```

```
    {

        Node *p = itr.current;

        iterator retVal( p->next );

        p->prev->next = p->next;

        p->next->prev = p->prev;

        delete p;

        --theSize;

        return retVal;
    }

iterator erase( iterator from, iterator to )

    {
        for( iterator itr = from; itr != to; )
            itr = erase( itr );
        return to;
    }
```

Stacks

Stack

- Last in First Out

```
#include <stack>
// Stack operations
stack<T> stackExample;
stackExample.push();
stackExample.pop();
stackExample.top();
stackExample.empty();
stackExample.size();
// with constructor & destructor
```

Stack Model - LIFO

- The top allows access to the top of the "Stack"
- Any list implementation could be used to make a stack
  - Operating on one end
- Vector/List ADTs
  - push_front()/pop_front()
  - push_back()/pop_back()

Stack Uses

- Depth-first search/backtracking
- Evaluating postfix expressions
- Converting infix to postfix
- Function calls (runtime stack)
- Recursion

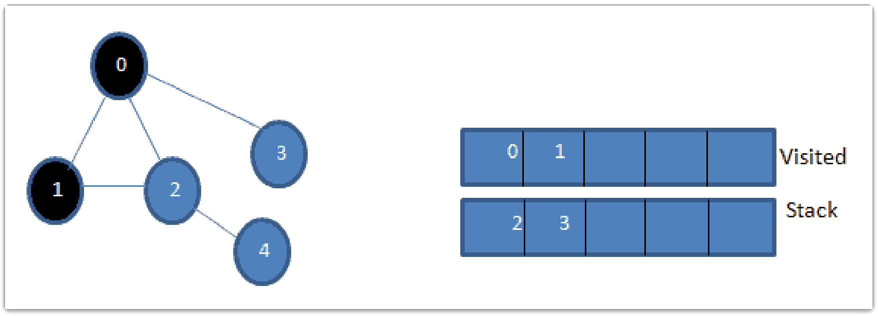
Runtime Stack

- Static
  - Executable code
  - Global variables
- Stack
  - Push for each function call
  - Pop for each function return

    - Local variables
- Heap
    - Dynamically allocated memories
    - new and delete

Depth First Expanded

- If there is an unlisted neighbor go there
- Retreat along the path to find unlisted neighbor, it cannot go deeper
- If there is a path from start to goal, DFS finds one such path
- Discover a path from **start** to the goal
    - Start from Node start stop if Node reaches goal



> *Keep Going Deeper*

```
// Depth First Search
DFS() {
stack<location> S;
//Mark the start location as visited
        S.push(start);
        while (!S.empty()) {
                t = S.top();
                if (t == goal) Success(S);
                if (// t has unvisited neighbors) {
                        //Choose an unvisited neighbor n
                        // mark n visited;
                        S.push(n);
                } else {
                        BackTrack(S);
                }
        }
        Failure(S);
}

/*
```

```
                              Another Implementation Of DFS
   ____________________________________________________
*/

BackTrack(S) {
        while (!S.empty() && S.top() has no unvisited neighbors) {
                S.pop();
        }
}

Success(S) {
        // print success
        while (!S.empty()) {
                output(S.top());
                S.pop();
        }
}

Failure(S) {
        // print failure
        while (!S.empty()) {
                S.pop();
        }
}
```

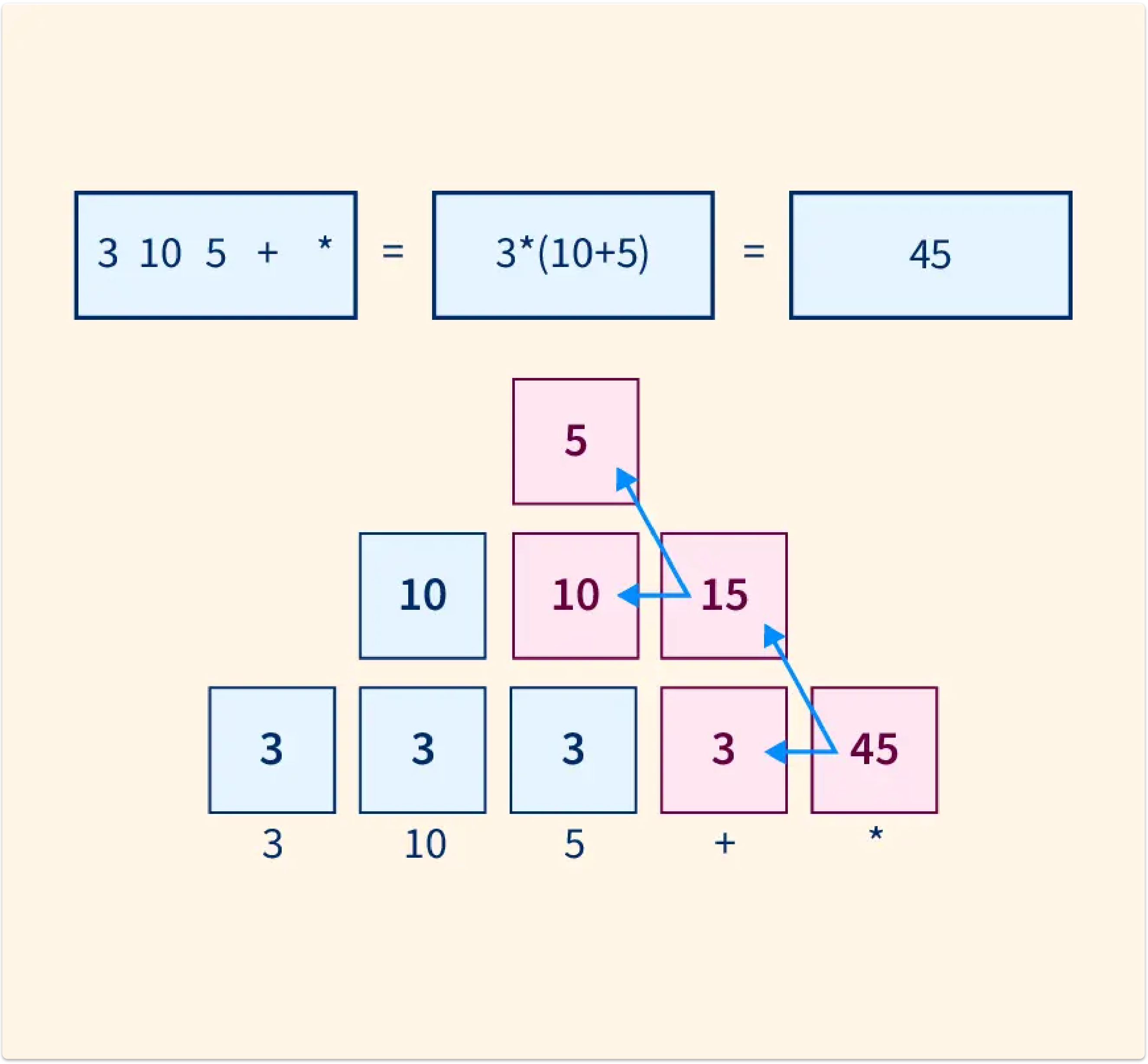
Postfix Expressions

- Use a stack of tokens
- Repeat
    - If operand, push onto the stack
    - If operator
    - pop operands off the stack
    - evaluate operator on operands
    - push the result onto the stack
    - Until expression is read
    - Return top of the stack

```
Evaluate(postfix expression) {

        // use stack of tokens;
        while(// expression is not empty) {
                t = next token;
                if (t is operand) {
                        // push onto the stack
                } else {
                        // pop operands for t off stack
                        // evaluate t on these operands
                        // push the result onto the stack
                }
        }
        // return top of stack
}
```

Postfix Visualized



Queue

Queue ADT - FIFO

- Elements of some proper type of T
- Operations
  - Feature: First In, First Out
  - void push(T t)
  - void pop()
  - T front()
  - bool empty()
  - unsigned int size()
  - Constructors and destructors

Operation Number	Command	Stack
1	Q.push(Ant)	Ant
2	Q.push(Bee)	Bee
3	Q.push(Cat)	Cat
4	Q.push(Dog)	Dog

> *What if we were to do Q.pop()?*

Operation Number	Command	Stack
1	Q.push(Bee)	Bee
2	Q.push(Cat)	Cat
3	Q.push(Dog)	Dog

Stack Uses

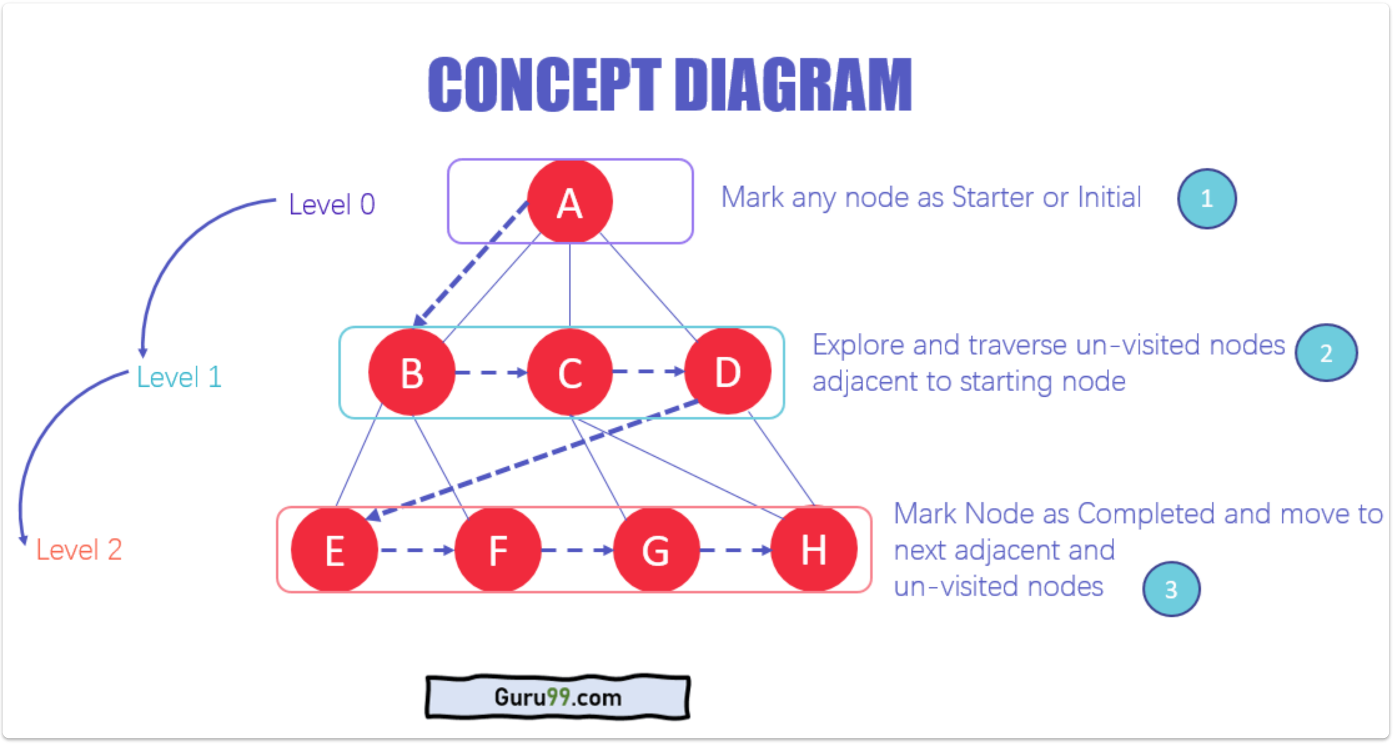
- Buffers
- Breadth first search
- Simulations
- Producer-Consumer Problems

Breadth first search Expanded

- Used to find the shortest path to the start to the goal
- Starting from Node start
- Visit all neighbors of the node

- Stop
    - if the neighbor is the goal
- Otherwise
    - Visit neighbors two hops away
- Repeat until visiting all neighbors

Breadth First Visualized


CONCEPT DIAGRAM
Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3
Guru99.com

Trees 🌲

Why We Need To Know

We need to know which data structure to utilize within certain use cases so that we may optimize our program's functionality as well as usability. We need to know when to use these structures and how to implement them.

Time Complexity Of Data Structures

Operation	Vector	Linked List	Deque	Tree (Unordered)	Hashtable (Unordered)
Insert front	O(n)	O(1)	O(1)	O(log(n))	NA
Insert Back	O(1)	O(1)	O(1)	O(log(n))	NA
Insert Middle	O(n)	O(1)	O(N)	O(log(n))	O(1)*
Remove Front	O(n)	O(1)	O(1)	O(log(n))	NA
Remove back	O(1)	O(1)	O(1)	O(log(n))	NA
Remove Middle	O(n)	O(1)	O(n)	O(log(n))	O(1)*
Random Access	O(1)	O(n)	O(1)	O(log(n))	NA
Search	O(n)	O(n)	O(n)	O(log(n))	O(1)*

- star is the Average Complexity
- C++ STL ordered map and set: balanced tree
- C++ STL unordered map and set: Hash table

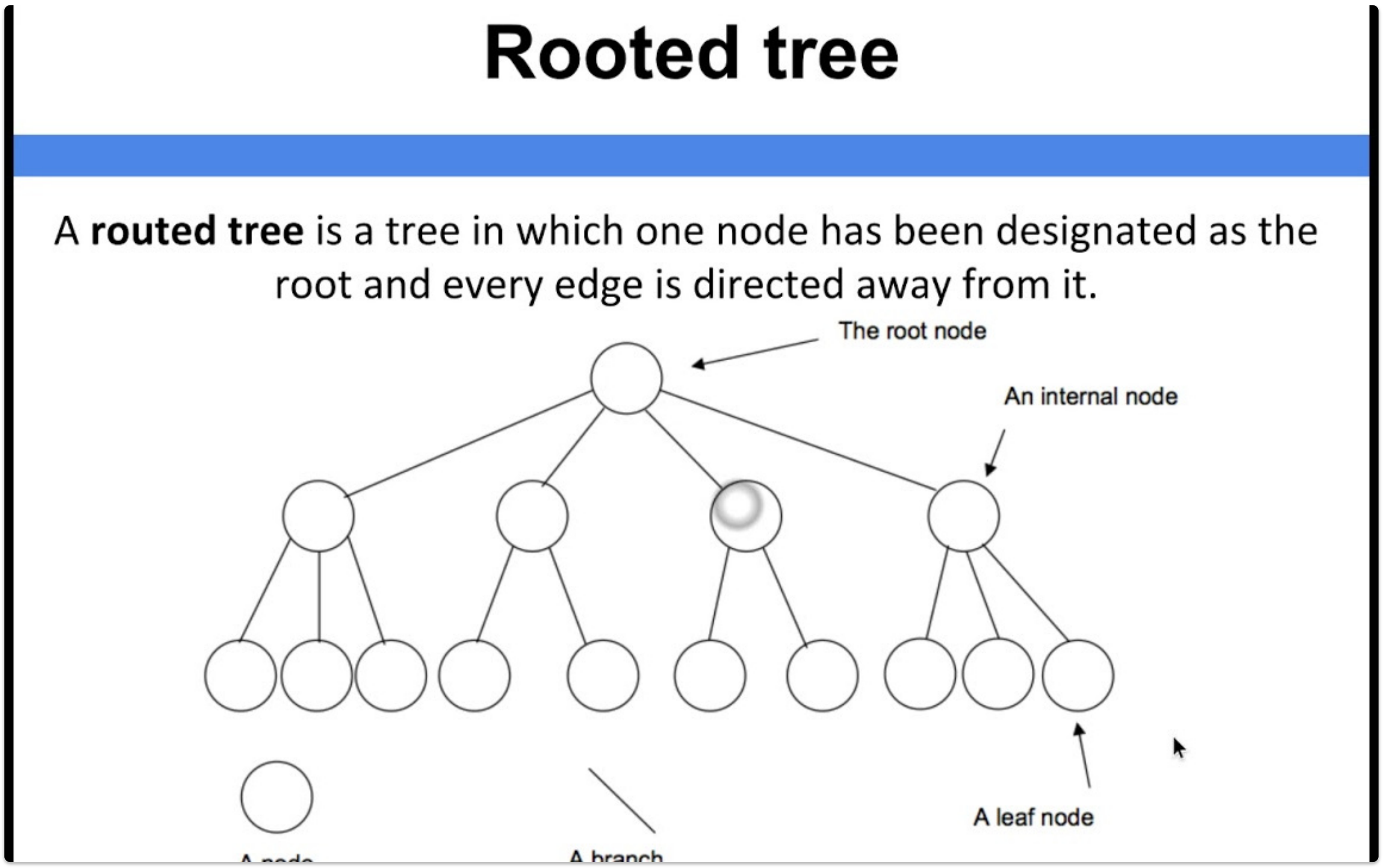
Theory and Terminology

Tree

- The tree is a connected graph with no cycles (no circles) - Consequences: - Between any two vertices, there is exactly one unique path

Rooted Tree

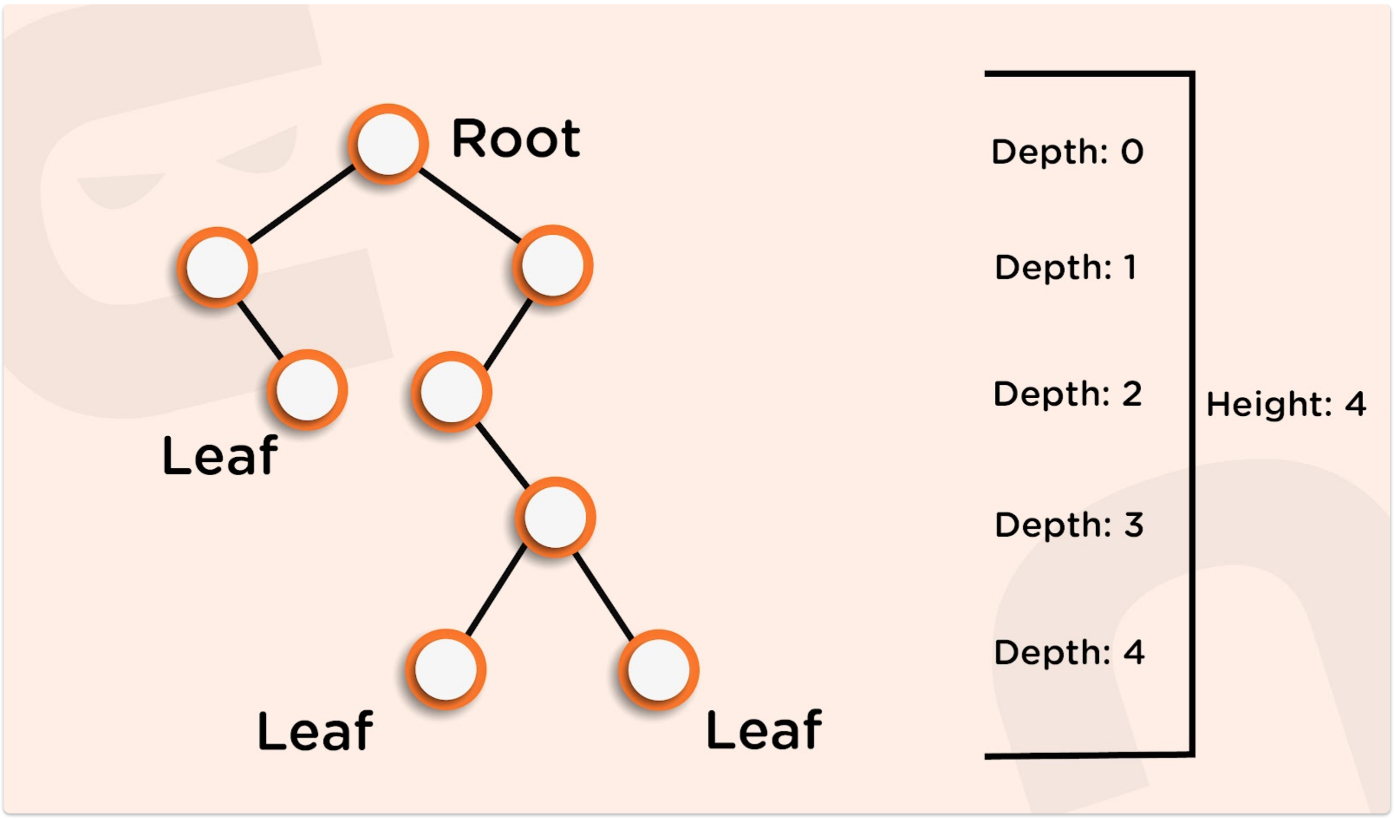
- A rooted tree - is connected - has no cycles - has exactly one vertex called the root of the tree - Consequences - Can be arranged so that the root is at the top - parent vs. child nodes and edges - sibling nodes - Nodes of the same parent nodes - leaf nodes - Nodes without children nodes



Rooted tree

A **routed tree** is a tree in which one node has been designated as the root and every edge is directed away from it.

- A unique path from the root to any vertex is a descending path
- Depth of vertex
  - Length of the unique descending path from the root to v
  - the root is at a depth 0
- The height of a vertex v is the length of the longest path from v to one of its leaves
- The height of a tree is the height of the root
  - Equal to the max depth



Rooted Tree: Recursive Definition

- A graph with N nodes and N-1 edges
- Graph has...
  - one root r
  - Zero or more non-empty subtrees

```
// Tree Node
struct TreeNode{
        Object element;
        TreeNode *firstChild;
        TreeNode *nextSibling;
}
```

Tree Traversal

- Often defined recursively
- Each kind corresponds to an iterator type
- Iterators are implemented non-recursively

Step	Description
1	Go to the root
2	Visit child subtrees

- Depth First Search
    - Begin at root
    - Visit vertex on arrival
- An implementation may be a recursive, stack-based, or nested loop
- For more information on depth-first search visit Stacks notes

Postorder Transversal

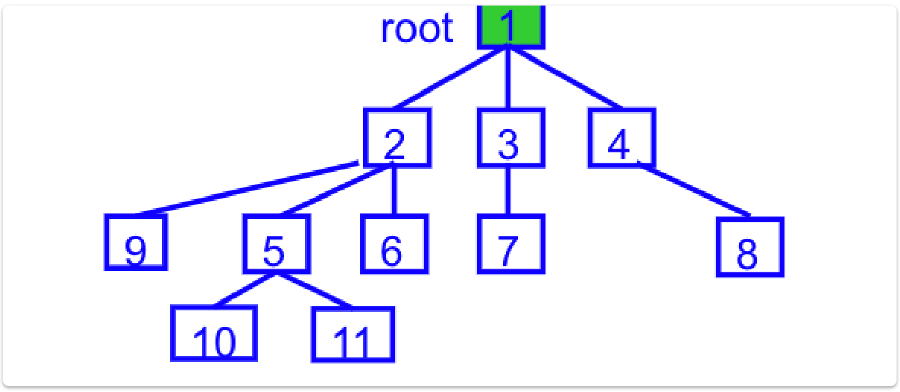
- The left subtree is traversed first
- Then the right subtree is traversed
- Finally, the root node of the subtree is traversed

Inorder Transversal

- The left subtree is traversed first
- Then the root node for that subtree is traversed
- Finally, the right subtree is traversed

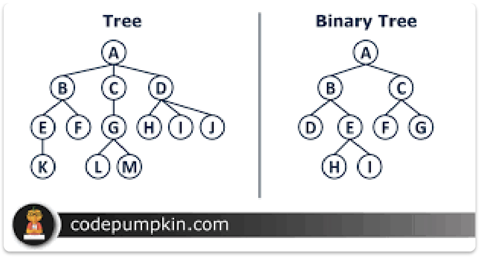
Preorder Transversal

- The root node of the subtree is visited first.
- Then the left subtree is traversed.
- At last, the right subtree is traversed.



Binary Tree

- Each node has at most two children, referred to as the left child and the right child.
- Every layer except maybe the bottom is fully populated with vertices.
- All nodes at the bottom level must occupy the leftmost spots consecutively



A complete binary tree with n vertices and h height satisfies

- $2^H \leq n < 2^{H+1}$
- $H \leq log(n) < H + 1$
- $H = floor(log(n))$