Root – top node in a BST tree  
Asymptotic time – (1) constant -> (n!) factorial time  
Inorder – LVR || Preorder – VLR || Postorder – LRV  
When a graph is dense(lots of connections) or weighted use matrix. Else use list  
BST: Pros – Fast to search (log(n)), sorted upon insertion || cons – balance overhead, insert scenario is (log{n))  
Best Data Structure for Redo/Undo buttons – stack; pushed/popped in reverse order  
LIFO/FIFO – Stack = LIFO, Queue = FIFO  
Stack is recursive data structure  
Arrays Pros: Fast access: (1), less memory   
Linked List Pros: fast add/remove (1), dynamic size  
Negative correlation between time/memory: more code == complexity   
Null pointer exception when you try to pop empty value  
Going Stack -> Queue: Order maintained Queue -> Stack: Order reversed  
BFSearch rules: smallest value first, in relative nodes (horizontal)  
DFSearch rules: same as BFS but (vertical)  
Cycles in graph are loops: 1 -> 2, 2-> 3, 3 -> 1  
DAG – Directed Acyclic Graph  
Undirected graphs are symmetric.  
Be able to look at function method and say what it does

Use Java Implementation to setup a Stack/Queue -Declare, Initialize, Add, Remove, Peek, Clear, Traverse

Stack: import java.util.Stack; // This references the Stack class of the Java API

Stack<String> st = new Stack<String>(); // Declared and initializes empty string stack named “st”

st.push(“String #1”); // Add first string to stack

st.push(“String #2”); // Add second string to stack

String s = st.pop(); // Capture removed item into string “s”

String s2 = st.peek(); // Save string at top of stack into string “s2”

boolean b = st.isEmpty(); // Saves whether stack is empty (it isn’t currently so this is false)

int n = st.size(); // Save number of items in stack in “n”

st.clear(); // Resets stack to empty (size() == 0; isEmpty() == true)

Queue: import java.util.Queue; import java.util.LinkedList;

Queue<String> q = new LinkedList<String>();

q.add(“String #1”); // Java API version uses “add” instead of “enqueue”

q.add(“String #2”); // Add the second string to the queue

String s = q.remove(); // Java API uses “remove” instead of “dequeue”

String s2 = q.peek(); // Head of queue item placed into s2

boolean b = q.isEmpty();// Works just like a Stack here

int n = q.size(); // Save number of items left in Queue

q.clear(); // Resets queue to empty (size() == 0; isEmpty() == true)

18. How an arbitrary linked list adds/removes node? - Add (5) where to put new node?

**public** **void** add(**int** index, String item){

// Create node to add Node X = **new** Node(); X.item = item; X.next = **null**;

/ Check for strange use cases... **if**(head == **null**){**if**(index != 0) **return**; // Not a valid index relative to current list

head = X; // Set our new node as head n++; **return**;} **else** **if** (head != **null** && index == 0){X.next = head; head = X; n++; **return**; }

// Otherwise, traverse the linked list for index position

**if**(index > n) **return**; // Cannot insert at an index larger than n

Node current = head;

Node previous = **null**;

**int** i = 0;

**while** (i < index) {

previous = current;

current = current.next;

**if** (current == **null**) **break**;

i++;

}

X.next = current;

previous.next = X;

n++;

}

19. How to implement Stack/Q using Arbitrary linked list? -How to push/pop? Adding to Q -> index = size()

**public** **class** Fstack{

// Fields **private** **int**[] a; **private** **int** cursor;

// Constructor **public** Fstack(){ a = **new** **int**[256]; cursor = 0;}

// Methods **public** **void** push(**int** item){ **if**(cursor >= a.length) // Prevent buffer overrun **return**; a[cursor] = item; cursor++; }

**public** **int** pop(){ **int** ret = -1;// Set to error code **if**(cursor > 0){cursor--; ret = a[cursor];} **return** ret; }

**public** **int** peek(){**int** ret = -1; **if**(cursor > 0){ret = a[cursor-1];}**return** ret;}

**public** **boolean** isEmpty(){ **return** cursor == 0;}

**public** **int** size(){**return** cursor;}

**public** **void** clear(){cursor = 0; // Be lazy and save cycles}}

///=>>Linked list implementation. **public** **class** Lstack{

// Fields **private** Node top; // Top of stack **private** **int** n; // How many items

**private** **class** Node{**int** item;// Holds the discrete item in node Node next; // Reference to next node in list}

// Constructor **public** Lstack(){top = **null**; n = 0;}

// Methods **public** **void** push(**int** item){Node oldTop = top; top = **new** Node();top.item = item; top.next = oldTop; n++;}

**public** **int** pop(){**if**(n == 0) **return** -1; **int** ret = top.item; top = top.next; n--; **return** ret;}

**public** **int** peek(){ **if**(n == 0) **return** -1; **return** top.item; }

**public** **boolean** isEmpty(){ **return** top == **null**; }

**public** **int** size(){ **return** n; }

**public** **void** clear(){ top = **null**; n = 0;}

//Linked list implementation **public** **class** Lqueue{

// Fields **private** Node head; **private** Node tail; **private** **int** n;

**private** **class** Node{**int** item; Node next;}

// Constructor **public** Lqueue(){ head = **null**; tail = **null**; n = 0;}

// Methods**public** **void** enqueue(**int** item){Node oldTail = tail; tail = **new** Node(); tail.item = item; tail.next = **null**;

**if**(isEmpty()) head = tail; **else** oldTail.next = tail; n++;}

**public** **int** dequeue(){ **if**(n == 0) **return** -1; **int** item = head.item; head = head.next;

**if**(isEmpty()) tail = **null**; n--; **return** item;}

**public** **int** peek(){**if**(n == 0) **return** -1; **return** head.item;}

**public** **boolean** isEmpty(){**return** head == **null**;}

**public** **int** size(){**return** n;}

**public** **void** clear(){head = **null**; tail = **null**; n = 0;} }