Data Structures – List

1. Array – data type in which data are stored in structured ordered , array can be of 2 types

First can be predefined size or non predefined size

Predefined Size Can Be Passed Using - #define max\_size in c++ or if size is given it can be passed directly

Or else new , malloc can be used to call the array which return void pointer

The Allocation can be – (i\*sizeof(int))

The Data Stored Is on the Stack Data Members , if malloc , new is passed the array are maintained on the heap memory

Before that all the variables declared are in the activation record . while using for loop if u pass i<= where I is any integer , this might effect the activation record of some other variable which is present at that data structures

Program For Various Function That Can Be Performed In Array -

void searching (int ar , int data, int size)

{

for(int i=0;i<size;i++)

{

if(ar[i]==data)

cout<<”found in an index”<<i;

else

cout<<”not present”;

}

}

1. Linked List : When people find the difficulty of array they switched to linked list in which each segment contains two data’s that are next and data

Where data contains the item and next contains the address to another data set

In linked list the elements are organized in a particular order and removal and insertion at any point is very easy

Struct Node

{

Int data;

Node \* next

}

#adding the element at the end u have to create a new node using new operator and then traverse to the end of the list and then add the elements over there

Remember while creating the node pass node.next = NULL so that the last element should have the Null pointer

PS : same concept is used in the stack (LIFO – last in first out )

#adding the element to first of the list

* While adding of the first new element remember to put null pointer and then every time u add the element u have to pass the head pointer a new location and the new element should point to previous head pointer
* While deleting the element remember to free that particular node which is being removed

1. Priority Queue : Each element has a particular priority , so taking up the priority queue from linked list or the array is time consuming task hence a new data structures are used know as Heap

The Heap data structures id used to stored of trees and there is a validity of a heap that a heap variant should be verified every time the item is inserted on the heap data structures

The heap data structures are in form of an array – where parent element can be assessed using ---

Parent – (i-1)/2;

Left -- (2\*i+1);

Right – (2\*i+2);

There are two typed of heaps

1. Max Heap – in which the parent element is the highest element
2. Min Heap – in which the parent element is the lowest element

Pseudo Code for Heap Insertion.

class Heap

{

public:

int cap , \*harr,arr\_size;

Heap(int size)

{

int cap=size;

harr = new int[size];

}

void insert(int key)

{

if(cap == arr\_size)

{

cout<<”overflow”;

}

arr\_size ++;

int i = arr\_size-1;

arr[i] = key;

while(i!=0 && ar[i]>arr[parent(i)])

{

swap(&ar[i],&ar[parent]);

i=parent(i);

}

}

Binary search Tree:

In a binary search tree the elements are sorted on the bases of the parent node example –

* If element to be inserted is less than the root element then the data will be stored in the left side of the root element
* else it will be stored to the right side of the root element

Traversal to the Binary Search Tree

1. In-order : - traversal to the left and then the root element and then the right element – this will generate the sorted array
2. Pre-order :- traversal to the root then to the left and then to the right
3. Post-order :- traversal to the left , then right and then the root element

AVL trees :- Sometimes there may be a possibility that binary tree get result into the straight line so to convert to have a binary tree in-variant true we use the concept of AVL trees in which the height is being specified to the AVL trees , which results into the consideration of left shift and right shift of the segment to make proper tree/binary tree

Providing the heights to the AVL trees

def height(node):

if(node == NULL)

return 0;

else

return max((height.(node->left) , height.(node->right))+1)

Backtracking Algorithms

**Backtracking**works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In context of Knight’s tour problem, an item is a Knight’s move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn’t violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

LeetCode Problems Revision

1. Maximum Depth Of Binary Tree

* Traverse to the left and right side of the tree and with every traversal increment the left and right side height int variables
* initially (root==NULL) return 0 else
* int lefth = traverse(root->left) ;
* int righth = traverse(root->right)
* if(lefth<righth) return righth+1;
* else return lefth+1

concept used – recursive function operation

1. Reverse the linked list

Concept used – we will be using 3 pointer where 2 pointers point to null and one pointer points to first index element of linked list

now , one pointer pointing to NULL shift it to the first element it is pointing and shift the first element pointer to second element

1st pointer pointing to Null

2nd pointer pointing to First Element of Linked List

3rd pointer pointing to Second Element of Linked List

now , 2nd pointer next point it to 1st pointer i.e NULL and then increment 1st pointer , 2nd pointer and 3rd pointer

now 2nd pointer next point it to 1st pointer i.e first element of linked list increment 1st , 2nd ,3rd pointer

incrementing is just equivalent that current->next value

***Program***

temp = prev = NULL

current = head;

prev = current->next;

current->next=temp;

prev=current ;

current=temp;

1. Single Number – Non duplicate Number – XOR Concept

Concept used – take each element and xor it with its previous element example

int a;

for i in nums where nums is an array

a^=i;

return i

In this method the number having non repeated printed into it

1. Invert Binary Tree

concept used is that point the left side to right and right side to left for the inversion

if(root==NULL)

return root;

right = traverse to the right

left = traverse to the left

root->right = left;

root->left=right;

return root;

1. Move Zeros To The End

concept – whenever you are not encountering with the zeros put the values of ar[i] where ar is array I is index into ar[last++] where last is just an counter

when encountered with zero do nothing at the end start from last and append zeros into the ar[] array starting with the index of last

for(int i =0 ; i <nums.length() ; i++)

{

if(ar[i]!=0)

{

ar[last++] = ar[i];

}

}

// start from last and append zeros to the array till you reach the array end i.e nums.length

//program is based on the vector concept

1. Merging Two Binary Tree –

concept – traverse to the last of the binary tree 1 and then add the its next should be point to the next binary tree

Node\* merge(Root\*N1, Root\*N2)

while(N1!=NULL)

{

N1=N1->next;

}

N1->next=N2;

return (N1)

1. Valid Anagram – Two Concept can be used a) Sort the array and then compare b) or compute the net total of all the elements and then compare them c) same concept as b create an array of 26 words one increment in array 1 and decrement in array 2

compare them and then you will get the result

1. Circular Linked List - concept used was using a two pointer in which we can we check slow and fast pointer coincide at a particular point

while(fast&&fast->next)

where slow=head->next;

fast = head->next->next

if(slow==fast) return true

1. Intersection of linked list – detection that there is a intersection in an linked list --- there are two linked lists so , A and B

combine A with B and then B with A and then which value is same in both and where it matches both return that one

1. Longest Substring in An Linked List – Dynamic Programming – concept used was sliding window – ByteByByte Study
2. Inorder Traversal Of Linked List – go to the left most root , cout<<root->data , go the right most root