Tree Good Questions

Zigzag level order traversal -

Code:

```
    vvi answer;
    ~~queue<pair<int,TreeNode*>> q;~~

    if(root == NULL)
        return answer;

stack <pair<int, TreeNode*>> s1, s2;
  s1.push({0, root});
while(!s1.empty() || !s2.empty()){

  while(!s1.empty()){
        auto s = s1.top(); s1.pop();
    int lvl = s.first; TreeNode* p = s.second;
    if(answer.size() == lvl)
          answer.push_back(vector<int>());
     answer[lvl].push_back(p->val);
    if(p->left) s2.push({lvl+1, p->left});
    if(p->right) s2.push({lvl+1, p->right});
}
```

```
while(!s2.empty()){
    auto s = s2.top(); s2.pop();
    int lvl = s.first; TreeNode* p = s.second;

    if(answer.size == lvl)
        answer.pb(vector<int>());
    answer[lvl].pb(p->val);
    if(p->right) s1.push({lvl+1, p->right});
    if(p->left) s1.push({lvl+1, p->left});

}
}
return answer;
```

Other approach → Level order traversal
then reverse vectors.

Construct Binary Tree from preorder and inorder

- Node that comes first in preorder is parent of successors.

- Left in ~~mod~~ inorder are to the left and right in inorder are to the right.

Create map of preorder for fast search of root node

Code:-

```
TreeNode* solve(vi inorder, int s, int e){
    if(s>e) return NULL;
    if(s==e) return new TreeNode(inorder[s]);
    int m=-1, mn= INT_MAX;
    for(int i=s; i<=e; i++){
        if(mp[inorder[i]] < mn) {
            mn= mp[inorder[i]]; m=j;}
    }
    TreeNode* root = new TreeNode(inorder[m]);
    root->left = solve(inorder, s, m-1);
    root->right = solve(inorder, m+1, e);
    return root;
}
```

Populating Next Right Pointers in Each Node

Given perfect binary tree with struct

```
struct Node{
        int val;
      Node* left ;
      Node* right;
      Node* next;
}
```

Approach 1 :- BFS

```
  if(root==NULL) return root;
   Node* larg = NULL; int lastlvl= -1;
queue <pair<int, Node*>> que;

~~Node*~~ que.push ({0, root});
```

```
while(!que.empty()){
    auto s = que.front(); que.pop();

  Node* p = s.second; int lvl = s.first;

   if(lazy!=NULL){
        if(lastlvl == lvl){
             lazy->next = p;
        }
   }
   lazy=p;
   lastlvl = lvl;
   if(p->left)
       que.push({lvl+1, p->left});
   if(p->right)
       que.push({lvl+1, p->right});
}
return root;
```

Approach 2 (Constant Space)

```
if(root==NULL) return;

~~T~~ Node* pre = root;
~~T~~ Node* curr = NULL;

while(pre->left){
        curr = pre
        while(curr){
            curr->left->next = curr->right;

  if(curr->next) curr->right->next = curr->next
                                        ->left

      curr = curr->next;
  }

pre= pre->left
}
```

1st Choice

Page No. Date

Approach 3 Recursive

```
dfs (Node* curr, Node* next){
    if( curr == null) return;
    curr->next = next;
    dfs( curr->left, curr->right);
    dfs( curr->right, curr->next == null ? null :
                                   curr->next->left)
}
```

Binary Tree Maximum path sum

```
int mx;
int solve( TreeNode* root) {
    if (root == null){ mx = max(mx, 0); return 0;}
    int l = solve ( root->left);
    int r = solve(root->right);
    int sum = max( root->val, max(root->val+l, root->val
                                   +r ))
mx = max( mx, max(sum, root->val + l+r))
return sum;
}
```

Kth smallest in BST

```
int ans;

int solve(TreeNode* root, int k, int small=0){
    if(root==NULL)
        return 0;
    int left = solve(root->left, k, small);
    if(small+left ==k-1)
        ans= root->val
    int right= solve(root->right, k, small
                                +left+1);
    return left+right+1;
```

Subset Generation

Backtracking

```
void subsets( arr, subset, index, res ) {
    res.push-back( subset)
    for(int i = index; i < arr.size ; i++) {
        subset.push-back (arr[i]);
        Subsets (arr, subset, i+1, res);
        subset.pop-back()
    }
}
```

Alternative IP = arr OP = empty

```
solve( IP, op) {  if ( IP.size == 0)
solve(IP, index+1, op)  res.push-back(OP);
solve(IP, index+1, op+ input[0]) ;
return; }
```

Iterative

Size of power set = 2^n

count from 0 to $2^n - 1$.

if (ith bit is set) then print ith element.

Heap Functions

Page No.
Date

```
max_heapify_down(vector<int>, int i){
    int left = 2i+1;
    int right = 2i+2;
    int mx = i;
    if( left < vec.size() && vec[left] > vec[mx])
            mx = left;
  if( right < vec.size() && vec[right] > vec[mx])
          mx = right;
  if(mx != i){
      swap(vec[i], vec[mx]);
    heapify_down(mx, vec);
  }
}
```

```
heapify~~down~~up( int i){
    if(i==0)
      return
int parent =(i-1)/2 ;
    if( A[i] > A[parent]) {
       swap( A[i], A[parent]);
       heapify-up( parent);
    }
}

int top() {
    if(size() ==0)
          return -1
    else
     return A[0];
}
```

```
void push (int ele){
    A.push-back(ele);
    int idx = A.size()-1;
  heapify-up(idx);
}

void pop(){
  if(size()==0)
      return;

  else{
     A[0] = A[A.size()-1];
      A.pop-back();
     heapify-down(0);
  }
}
```

$N/2 \rightarrow N-1$ are leaf nodes

Call heapify down from $N/2 - 1$ to 0. To Build heap in $O(n)$

Bit about Sorting Algorithms

Selection Sort — Comparison Based

$O(n^2)$, In place, Space $O(1)$

Takes atmost $O(N)$ swaps

Default implementation is not stable.

Bubble Sort — Comparison Based

$O(1)$ space

$O(n^2)$, Inplace, $\Omega(n)$ minimum time, when sorted already

Stable,

max $\frac{n(n-1)}{2}$ swaps

Insertion Sort Comparison Based

$O(n^2)$, Inplace, $O(1)$ space, $O(n)$ minimum time

Stable, Algorithm paradigm: Incremental Approach. $\frac{n(n-1)}{2}$ worst case swaps

When the array is nearly sorted

Comparison Based

Merge Sort (Divide and Conquer)

$T(n) = 2T(n/2) + \Theta(n)$

$\Theta(n\log n)$ in all cases best, worst, and avg

Auxiliary Space :- $O(n)$

Not in Place. Stable

Application, - Sort linked list in $O(n\log n)$

- Slower for smaller tasks.
- Extra space
- Goes through whole process even if array is already sorted.

Heapsort

$O(n\log n)$, in-place, not stable

Buildheap is $O(n)$ and overall $O(n\log n)$

☆ Quick Sort ☆ (Divide and Conquer)

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

Worst case:- When the process always picks the greatest or smallest elements as pivot.

When pivot is last element, worst case ~~of~~ ~~of~~ becomes when array is already sorted in increasing or decreasing order.

Not Stable, In place.

1st Choice

Page No.
Date

Bucket

$O(n)$, Not in Place, Stable

Counting Sort

Same

$O((n+k))$, Not in Place, Stable

Normalisation Required for the numbers.

~~$O(k)$~~ Space
$O(n+k)$

– Works when range is feasible
constant overhead, extra space

Radix Sort

$O(d*(n+k))$, not in place, Stable.

~~$O(n+k)$~~ Space
$O(n+k)$

– Constant overhead, extra space

☆ Quick Sort performs better with caches.

Nth highest sql query

```
Select * from emp e1 where N-1
    = (Select count(distinct(salary))
       from emp e2 where
         e2.salary > e1.salary)
```