Kruskal Sort decreasing,

Prim sort

**ListNode\* reverseList(ListNode\* head) {**

ListNode \*prev=NULL,\*curr=head,\*next;

while(curr){

next=curr->next;

curr->next=prev;

prev=curr;

curr=next;

}

return prev;

}

**void reorderList(ListNode\* head) {**

if(!head || !head->next) return;

ListNode\* temp = head->next;

queue<ListNode\*> fwd;

stack<ListNode\*> bwd;

while(temp){

bwd.push(temp);

fwd.push(temp);

temp = temp->next;

}

temp = head;

ListNode \* dummy = nullptr;

while(true){

cout<<bwd.top()->val<<" "<<fwd.front()->val<<endl;

if(bwd.top() == fwd.front()){

temp ->next = fwd.front();

temp->next->next = nullptr;

break;

}

temp->next = bwd.top();

bwd.pop();

temp = temp -> next;

temp -> next = fwd.front();

fwd.pop();

dummy = temp->next;

temp = temp -> next;

if(dummy==bwd.top()) {

cout<<"ended : "<<dummy->val<<endl;

temp -> next = nullptr;

break;

}

}

}

};

**int isBST(struct node\* node)**

{

if (node == NULL)

return 1;

/\* false if the max of the left is > than us \*/

if (node->left!=NULL && maxValue(node->left) > node->data)

return 0;

/\* false if the min of the right is <= than us \*/

if (node->right!=NULL && minValue(node->right) < node->data)

return 0;

/\* false if, recursively, the left or right is not a BST \*/

if (!isBST(node->left) || !isBST(node->right))

return 0;

/\* passing all that, it's a BST \*/

return 1;

}

**struct node\* search(struct node\* root, int key)**

{

// Base Cases: root is null or key is present at root

if (root == NULL || root->key == key)

return root;

// Key is greater than root's key

if (root->key < key)

return search(root->right, key);

// Key is smaller than root's key

return search(root->left, key);

}

Mirror

**void mirror(struct Node\* node)**

{

if (node == NULL)

return;

else

{

struct Node\* temp;

mirror(node->left);

mirror(node->right);

temp= node->left;

node->left = node->right;

node->right = temp;

}

}

Compare mirror

**int areMirror(Node\* a, Node\* b)**

{

if (a==NULL && b==NULL)

return true;

if (a==NULL || b == NULL)

return false;

Note that in recursive calls, we pass left

of one tree and right of other tree \*/

return a->data == b->data &&

areMirror(a->left, b->right) &&

areMirror(a->right, b->left);

}

Middle of node

**ListNode\* middleNode(ListNode\* head) {**

ListNode\* temp=head;

int l=0;

while(temp){

l++;

temp=temp->next;

}

cout<<l;

int mid=l/2 ;

temp=head;

while(mid--) temp=temp->next;

return temp;

}

Is list a cycle?

**bool hasCycle(ListNode \*head) {**

if(!head || !head->next)return false;

vector<ListNode\*> a;

ListNode \*temp = head;

while(temp->next){

a.push\_back(temp);

temp = temp->next;

for(int i=0; i<a.size();i++){

if(a[i]==temp)return true;

}

}

return false;

}

**BFS**

void BFS(Graph const &graph, int v, vector<bool> &discovered)

{

queue<int> q;

discovered[v] = true;

q.push(v);

while (!q.empty())

{

v = q.front();

q.pop();

cout << v << " ";

for (int u : graph.adjList[v])

if (!discovered[u])

{

discovered[u] = true;

q.push(u);

}

}

}

**RTF**

vector<int> rightSideView(TreeNode\* root)

{

vector<int> visable;

if (root == nullptr)

{

return visable;

}

queue<TreeNode\*> q;

q.push(root);

while (!q.empty())

{

int size = q.size();

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

{

TreeNode\* current = q.front();

q.pop();

if (i == size - 1)

{

visable.push\_back(current->val);

}

if (current->left)

{

q.push(current->left);

}

if (current->right)

{

q.push(current->right);

}

}

}

return visable;

}

**int CountNodes(node\*root)**

{

**if**(root==NULL)

**return** 0;

**if**(root->left!=NULL)

{

n=n+1;

n=CountNodes(root->left);

}

**if**(root->right!=NULL)

{

n=n+1;

n=CountNodes(root->right);

}

**return** n;

}

**void Graph::bfs(int start)**

{

map<int, bool> visited;

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty())

{

int currentNode = q.front();

q.pop();

cout << currentNode << " ";

for (auto child : adjlist[currentNode])

{

if (!visited[child])

{

q.push(child);

visited[child] = true;

}

}

}

}

**Double hashing**

void insertHash(int key)

{

if (isFull())

return;

int index = hash1(key);

if (hashTable[index] != -1) {

int index2 = hash2(key);

int i = 1;

while (1) {

int newIndex = (index + i \* index2) % TABLE\_SIZE;

if (hashTable[newIndex] == -1) {

hashTable[newIndex] = key;

break;

}

i++;

}

}

else

hashTable[index] = key;

curr\_size++;

}

**struct node \* rrotation(struct node \*n){**

struct node \*p;

struct node \*tp;

p = n;

tp = p->left;

p->left = tp->right;

tp->right = p;

return tp;

}

**struct node \* lrotation(struct node \*n){**

struct node \*p;

struct node \*tp;

p = n;

tp = p->right;

p->right = tp->left;

tp->left = p;

return tp;

}

**void levelOrder(Node\* root)**

{

if (root == nullptr)

{

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty())

{

cout << q.front()->val << " ";

if (q.front()->left != nullptr)

{

q.push(q.front()->left);

}

if (q.front()->right != nullptr)

{

q.push(q.front()->right);

}

q.pop();

}

}

**struct node \* rlrmoreheightrotation(struct node \*n){**

struct node \*p;

struct node \*tp;

struct node \*tp2;

p = n;

tp = p->right;

tp2 =p->right->left;

p -> right = tp2->left;

tp ->left = tp2->right;

tp2 ->left = p;

tp2->right = tp;

return tp2;

}

**struct node \* lrlmoreheightrotation(struct node \*n){**

struct node \*p;

struct node \*tp;

struct node \*tp2;

p = n;

tp = p->left;

tp2 =p->left->right;

p -> left = tp2->right;

tp ->right = tp2->left;

tp2 ->right = p;

tp2->left = tp;

return tp2;

}

Bst insert

**Node\* insert(Node\* root, int val)**

{

if (root == nullptr)

{

Node\* newNode = new Node(val);

root = newNode;

}

else if (val < root->val)

{

root->left = insert(root->left, val);

}

else if (val > root->val)

{

root->right = insert(root->right, val);

}

return root;

}