void **buildTree**(Node \*\*root, int data)

{ Node \*newptr;

if(\*root==NULL){

newptr = (Node \*)malloc(sizeof(Node));

newptr->data = data;

newptr->left = NULL;

newptr->right = NULL;

\*root = newptr;

}else{

if(data < (\*root)->data) buildTree(&((\*root)->left),data);

else if(data > (\*root)->data) buildTree(&((\*root)->right),data);

}

}

int **getMax**(Node \*root)

{

int left\_height;

int right\_height;

if(root==NULL) return 0;

else{

left\_height = getMax(root->left);

right\_height = getMax(root->right);

if(left\_height > right\_height) return left\_height+1;

else return right\_height+1;

}

}

void **printInorder**(Node \*root)

{

Node \*tmp = root;

if(tmp!=NULL){

printInorder(tmp->left);

printf("%d ",tmp->data);

printInorder(tmp->right);

}

}

void **SumLevel**(Node \*root, int level)

{

if(root!=NULL && level==1){

count++;

sum += root->data;

}else if(root!=NULL && level>1){

level--;

SumLevel(root->left,level);

SumLevel(root->right,level);

}

}

void **freeTree**(Node \*root)

{ if(root!=NULL){

freeTree(root->left);

freeTree(root->right);

free(root);

}

}

Node\* **buildTree(int\* inorder, int\* preorder**, int inorder\_start, int inorder\_end)

{

Node \*root;

int N;

int count=0;

N = inorder\_end - inorder\_start + 1;

while(count<N && inorder[count]!=preorder[0]){

count++;

}

if(count==N) return NULL;

root = (Node \*)malloc(sizeof(Node));

root->data = preorder[0];

root->left = buildTree(inorder,preorder+1,inorder\_start,inorder\_start+count);

root->right = buildTree(inorder+count+1,preorder+count+1,inorder\_start+count+1,inorder\_end);

return root;

}

typedef **struct** treeNode

{ int data;

struct treeNode \*left;

struct treeNode \*right;

} Node;

void **caculate**(Node \*root)

{ if (root == NULL) return; if(root->left==NULL && root->right==NULL){

ans += root->data;

}

caculate(root->left);

caculate(root->right);

}

void **caculateLeafNodesSum**(Node\* root)

{ caculate(root);

printf("%d\n",ans);

}

void **Array\_BST::insert**(const int &data)

{ int i = 1;

int h = 1;

while(array[i] != 0){

if(data >array[i]) i = 2\*i + 1;

else if(data < array[i]) i = 2\*i;

else if(data == array[i]) return;

h++;

}

array[i] = data;

if(h > Height) Height = h;

}

void **List\_BST::insert**(const int &data)

{ ListNode \*parrent;

ListNode \*tmp = root;

int h = 1;

int dir;

if(root == NULL){

root = createLeaf(data);

Height = 1;

return;

}else{

while(tmp != NULL){

if(data > tmp->key){

parrent = tmp;

tmp = tmp->right;

h++;

dir = 1;

}else if(data < tmp->key){

parrent = tmp;

tmp = tmp->left;

h++;

dir = 0;

}else return;

}

if(dir == 1) parrent->right = createLeaf(data);

else if(dir == 0) parrent->left = createLeaf(data);

if(h > Height) Height = h;

}

}

bool **List\_BST::search**(const int &data) const

{ ListNode \*tmp = root;

while(tmp != NULL){

if(data > tmp->key) tmp = tmp->right;

else if(data < tmp->key) tmp = tmp->left;

else return true;

}

return false;

}

Node \***construct**()

{ Node \*root = NULL;

while(1){

if(str[ind] == ')'){

return NULL;

}else if(str[ind]=='('){

ind++;

}else if(str[ind]=='-'){

ind++;

flag = 1;

}else if(isdigit(str[ind])){

data2 = str[ind] - '0';

ind++;

while(isdigit(str[ind])){

data2 = data2\*10;

data2 += str[ind] - '0';

ind++;

}

if(flag==1){

data2 = (-1)\*data2;

root = newNode(data2);

flag = 0;

}else{

root = newNode(data2);

}

break;

}

}

root->left = construct();

ind++;

root->right = construct();

ind++;

if(str[ind]==')') return root;

}

int **countNode**(Node \*root)

{

if(root==NULL) return 0;

return ( 1 + countNode(root->left) + countNode(root->right) );

}

bool **complete**(Node \*root, int index2, int total)

{ if(root==NULL) return true;

if(index2 >= total) return false;

return ( complete(root->left, 2\*index2 + 1, total) && complete(root->right, 2\*index2 + 2, total) );

}

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

{ if (node == NULL) return;

else {

Node \*tmp;

mirror(node->left);

mirror(node->right);

tmp = node->left;

node->left = node->right;

node->right = tmp;

}

}

bool **isStructSame**(Node \*a, Node \*b)

{ if(a == NULL && b == NULL) return true;

if(a != NULL && b != NULL && isStructSame(a->left, b->left) && isStructSame(a->right, b->right)) return true;

return false;

}

bool **foldable**(Node \*root)

{ int result;

if(root==NULL) return true;

mirror(root->left);

result = isStructSame(root->left, root->right);

mirror(root->left);

return result;

}

long long int **helper**(Node\* node, long long int& l, long long int& r) {

if (!node) return 0;

long long int ll = 0, lr = 0, rl = 0, rr = 0;

l = helper(node->left, ll, lr);

r = helper(node->right, rl, rr);

return max(node->data + ll + lr + rl + rr, l + r);

}

long long int **QQ**(Node\* root) {

long long int l = 0, r = 0;

return helper(root, l, r);

}

Node \*deleteLeaf(Node \*root)

{ if(root == NULL) return NULL;

if(root->left==NULL && root->right==NULL){

delete root;

return NULL;

}

root->left = deleteLeaf(root->left);

root->right = deleteLeaf(root->right);

return root;

}

void **printLevelOrder**(node\* root)

{

int h = getMax(root);

int i;

for (i = 1; i <= h; i++)

printGivenLevel(root, i);

}

void **printGivenLevel**(node\* root, int level)

{

if (root == NULL)

return;

if (level == 1)

cout << root->data << " ";

else if (level > 1)

{ printGivenLevel(root->left, level-1);

printGivenLevel(root->right, level-1);

}

}

//flattening ( inorder)

int main(void)

{

int i;

scanf("%d",&total);

for(i=1;i<=N;i++){ scanf("%d",&A[i]);

}

find\_seq(1);

return 0;

}

void **find\_seq**(int k)

{ if(k < N){

find\_seq(2\*k);

printf("%d ",A[k]);

find\_seq(2\*k+1);

}

if (k==N){

printf("%d\n",A[k]);

}

}

**// flattening (preorder)**

void flatten(int i);

int A[1000];

int n;

int main(void)

{ int i;

scanf("%d",&n);

for(i=1;i<=n;i++){scanf("%d",&A[i]);}

flatten(1);

return 0;

}

void flatten(int k)

{ if(k==n){

printf("%d\n",A[n]);

}else if(k<n){

printf("%d ",A[k]);

flatten(2\*k);

flatten(2\*k+1);

}

}

**// flattening (preorder)(version 2)**

void flatten(struct Node\* root)

{ if (root == NULL || root->left == NULL && root->right == NULL) { return;

}

// if root->left exists then we have

// to make it root->right

if (root->left != NULL) {

flatten(root->left);

struct Node\* tmpRight = root->right;

root->right = root->left;

root->left = NULL;

struct Node\* t = root->right;

while (t->right != NULL) {

t = t->right;

}

t->right = tmpRight;

}

flatten(root->right);

}再用inorder traversal去printf

//**level order to BST**

Node\* getNode(int data)

{ Node \*newNode =

(Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

Node \*LevelOrder(Node \*root , int data)

{ if(root==NULL){

root = getNode(data);

return root;

}

if(data <= root->data)

root->left = LevelOrder(root->left, data);

else

root->right = LevelOrder(root->right, data);

return root;

}

Node\* constructBst(int arr[], int n)

{ if(n==0)return NULL;

Node \*root =NULL;

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

root = LevelOrder(root , arr[i]);

return root;

}

Main : Node \*root = constructBst(arr, n);

**// delete node in BST**

TreeNode\* deleteNode(TreeNode\* root, int key) {

if (!root) return NULL;

if (root->val > key) {

root->left = deleteNode(root->left, key);

} else if (root->val < key) {

root->right = deleteNode(root->right, key);

} else {

if (!root->left || !root->right) {

root = (root->left) ? root->left : root->right;

} else {

TreeNode \*cur = root->right;

while (cur->left) cur = cur->left;

root->val = cur->val;

root->right = deleteNode(root->right, cur->val);

}

}

return root;

}

**// MAX heap**

void **swap**(int &p1, int &p2){

int temp = p1;

p1 = p2;

p2 = temp;

}

void **MaxHeapify**(std::vector<int> &array, int root, int length){

int left = 2\*root,right = 2\*root + 1, largest;

if (left <= length && array[left] > array[root])

largest = left;

else largest = root;

if (right <= length && array[right] > array[largest])

largest = right;

if(largest !=root){ swap(array[largest], array[root]);

MaxHeapify(array, largest, length);

}

}

void **BuildMaxHeap**(std::vector<int> &array){

for (int i = (int)array.size()/2; i >= 1 ; i--) {

MaxHeapify(array, i, (int)array.size() - 1);

}

}

void **HeapSort**(std::vector<int> &array){

array.insert(array.begin(), 0);

**BuildMaxHeap**(array);

int size = (int)array.size() -1;

for (int i = (int)array.size() -1; i >= 2; i--) {

swap(array[1], array[i]);

size--;

**MaxHeapify**(array, 1, size)

; }

array.erase(array.begin()); }

//**MAX heap(2)**

void **heapify**(int arr[], int n, int i)

{ int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i) {

swap(arr[i], arr[largest]);

**heapify**(arr, n, largest);

}

}

// Function to build a Max-Heap from the given array

void **buildHeap**(int arr[], int n)

{ int startIdx = (n / 2) - 1;

for (int i = startIdx; i >= 0; i--) {

heapify(arr, n, i);

}

}

void **deleteRoot**(int arr[], int& n)

{ int lastElement = arr[n - 1];

arr[0] = lastElement;

n = n - 1;

heapify(arr, n, 0);

}

**// insert**

void **heapify**(int arr[], int n, int i)

{ int parent = (i - 1) / 2;

if (arr[parent] > 0) {

if (arr[i] > arr[parent]) {

swap(arr[i], arr[parent]);

heapify(arr, n, parent);

}

}

}

// Function to insert a new node to the Heap

void insertNode(int arr[], int& n, int **Key**)

{ n = n + 1;

arr[n - 1] = Key;

heapify(arr, n, n - 1);

}