Jenish Patel

Professor Ken

CS350

11 November 2020

Week10

1. Bubble sort uses (N^2) notation so that’s why they compare all and each node.

void bubbleSort(int array[], int size) {

for (int step = 0; step < size - 1; ++step) {

Int count =0

for (int i = 0; i < size - step - 1; ++i) {

if (array[i] > array[i + 1]) {

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

count=1;

}

}

if(count==0){

brake;}

}

}

2. If the given **array** has to be **sorted** in ascending order, then **bubble sort** will start by comparing the first element of the **array** with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

With a Best case scenario, the bubble sort perform N notation

1. In insertion sort the arrey will be virtually split into two-part the backside part will be sorted and the front side part will unsorted.

In the best case, the insertion sort performs (O(n))

1. In Merge sort the array Divide the unsorted list into n sublists, each containing one element. Repeatedly **merge** sublists to produce new **sorted** sublists until there is only one sublist remaining.

With a merge-sort when the array will be sorted in the reverse order the sortation will perform (Ologn) and (O(N)) Notation

1. In the Quicksort the **array** and partitioning the other elements into two sub-**arrays**, according to whether they are less than or greater than the pivot. The sub-**arrays** are then **sorted** recursively.

For the best-case, Quicksort hasn’t any impact because he just dived the node and seems the left side is larger than the right side the sortition will perform withing only 2 steps

The quicksort perform N\*(Logn)

3. void Binarytnc(add, int num){

Node \*temp, \*dad;

if (root==NULL) {

root = new Node;

root->data=num;

root->left=0;

root->right=0;

temp=root;

printf("Number is main root \n");

} else {

temp=root;

while (temp!=NULL) {

if (num <= temp->data){

dad = temp;

temp=temp->left;

//root =temp;

printf("value is to the left \n");

} else {

dad =temp;

temp=temp->right;

//root=temp;

printf("value is to the right \n");

}

}

Node \*newNode = new Node;

newNode->data = num;

newNode->left = NULL;

newNode->right = NULL;

if(num <= dad->data)

dad->left = newNode;

else

dad->right = newNode;

}

}

4. void BST (Nodetype \* Insert, Nodetype \* tree, int key, char \* name);

{

if(ptr == NULL) {

ptr = new node;

ptr->info = num;

ptr->lchild = NULL;

ptr->rchild = NULL;

if(start == NULL)

start = ptr;

return;

}

else if(ptr->info){

insert(ptr->lchild);

Return num;

}

else if( ptr->info) {

insert(ptr->rchild);

}

else {

printf "Duplicate element \n";

return;

}

}

5. #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

int isBSTUtil(struct node\* node, int min, int max);

int isBST(struct node\* node)

{

return(isBSTUtil(node, INT\_MIN, INT\_MAX));

}

int isBSTUtil(struct node\* node, int min, int max)

{

/\* an empty tree is BST \*/

if (node==NULL)

return 1;

if (node->data < min || node->data > max)

return 0;

return

isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values

isBSTUtil(node->right, node->data+1, max); // Allow only distinct values

}

struct node\* newNode(int data)

{

struct node\* node = (struct node\*)

malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return(node);

}

int main()

{

struct node \*root = newNode(4);

root->left = newNode(2);

root->right = newNode(5);

root->left->left = newNode(1);

root->left->right = newNode(3);

if(isBST(root))

printf("Is BST");

else

printf("Not a BST");

getchar();

return 0;

}

6. #include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

struct node\* newNode(int data)

{

struct node\* node = (struct node\*)

malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return(node);

}

int identicalTrees(struct node\* a, struct node\* b)

{

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

return 1;

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

{

return

(

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

identicalTrees(a->left, b->left) &&

identicalTrees(a->right, b->right)

);

}

return 0;

}

int main()

{

struct node \*root1 = newNode(1);

struct node \*root2 = newNode(1);

root1->left = newNode(2);

root1->right = newNode(3);

root1->left->left = newNode(4);

root1->left->right = newNode(5);

root2->left = newNode(2);

root2->right = newNode(3);

root2->left->left = newNode(4);

root2->left->right = newNode(5);

if(identicalTrees(root1, root2))

printf("Both tree are identical.");

else

printf("Trees are not identical.");

getchar();

return 0;

}