**Section (I): Tracing Questions (Total: 10 marks)**

**(Q1) (3 mark)**

* Draw the binary expression tree for the following prefix expression.
* Also, write the corresponding postfix, and infix expression.
* **+ - A \* B C - K \* \* \* W G H M.**

**(Q2) (5 mark)**

**For exercises (i - iii) use the following values, and draw the hash table for each of them:**

**4 , 7 , 6 , 14 , 21 , 12 , 31 , 24**

1. Store the values in a hash table with size 8 cells.
2. Store the values in a hash table with 3 buckets, each bucket contains 3 cells.
3. Store the values in a hash chain table with function (Key % 3).
4. Fill in the following table, showing the number of comparisons needed to find each value:-

|  |  |  |  |
| --- | --- | --- | --- |
|  | **(i)** | **(ii)** | **(iii)** |
| **14** |  |  |  |
| **31** |  |  |  |
| **24** |  |  |  |

**Section (II): Algorithm Questions (Total: 14 marks)**

**Algorithm 1: (14 marks)**

You are asked to write a function called **Display\_PathOf\_2nd\_BasedON\_the\_1st (int v1 , int v2 )**

* Your function will find the value **(V1).**
  + Then starting from **(V2)**, Display the nodes in the backward of the path of **(V1)**.
* e.g. V1 = 275 & V2 = 1000

**output : 1000, 990, 980, 1100, 960, 1200**

**Note:** in case (V2) failed to make backward path like (V1) 🡪 then don’t display anything.

**275**

**990**

**1000**

1020

995

**500**

850

90

910

800

73

75

85

84

72

86

70

900

**250**

**280**

**200**

**260**

276

277

1300

**980**

**1100**

**960**

**1200**

950

void Display\_PathOf\_2nd\_BasedON\_the\_1st(int v1, int v2)

{

CBST CTree;

CTNode\* pT = CTree.pRoot;

while (pT->info != v1)

{

if (v1 < pT->info)

{

cout << pT->info << endl;

pT = pT->pLeft;

}

else

{

cout << pT->info << endl;

pT = pT->pRight;

}

}

int check = 0;

pT = CTree.pRoot;

CTNode\* pT2 = CTree.pRoot;

CTNode\* pCatch = CTree.pRoot;

while (pT->info != v1 && pT2->info != v2)

{

if (v1 < pT->info && v2 < pT2->info)

{

pT = pT->pLeft;

pT2 = pT2->pLeft;

check = 1;

}

else if (v1 > pT->info && v2 > pT2->info)

{

pT = pT->pRight;

pT2 = pT2->pRight;

check = 1;

}

else

{

if (v2 < pT->info)

{

pT2 = pCatch;

pT = CTree.pRoot;

pT2 = pT2->pLeft;

pCatch = pT2;

check = 0;

}

else

{

pT2 = pCatch;

pT = CTree.pRoot;

pT2 = pT2->pRight;

pCatch = pT2;

check = 0;

}

}

}

while (pCatch->info != v2 && check==1)

{

if (v1 < pCatch->info)

{

cout << pCatch->info << endl;

pCatch = pCatch->pLeft;

}

else

{

cout << pCatch->info << endl;

pCatch = pCatch->pRight;

}

}

}

**DR WAY**

void DoIt(int v1, int v2)

{

CTNode\* pT = pRoot;

int ct = 0;

int ct2 = 0;

while (pT != NULL)

{

if (pT->info == v1)

{

return;

}

if (v1 > pT->info)

{

pT = pT->pRight;

}

else

{

pT = pT->pLeft;

}

ct++;

}

pT = pRoot;

while (pT != NULL)

{

if (pT->info == v2)

{

return;

}

if (v2 > pT->info)

{

pT = pT->pRight;

}

else

{

pT = pT->pLeft;

}

ct2++;

}

int diff = ct2 - ct;

CTNode\* pR = pRoot;

int ct3 = 0;

while (ct3 < diff)//goes till 1200

{

if (pR->info == v2)

{

return;

}

if (v2 > pR->info)

{

pR = pR->pRight;

}

else

{

pR = pR->pLeft;

}

ct3++;

}

pT = pRoot;

Disp(pT, pR, v1);

}

void Disp(CTNode\* pT, CTNode\* pR, int v1)

{

if (pT->info == v1)

{

cout << pR->info;

return;

}

if (v1 > pT->info)

{

pR = pR->pRight;

pT = pT->pRight;

}

else

{

pR = pR->pLeft;

pT = pT->pLeft;

}

Disp(pT, pR, v1);

cout << pR->info;

}

**Section (III): Problem Solving (Total: 36 marks)**

**Problem 1: (10 marks)**

*In the this question use the following definition for the nodes in the List:*

**class CListNode**

**{**

**public:**

info

pNext

**int info;**

**CListNode \*pNext;**

**};**

**Note:**

* in this problem you have to write just a single function

(no permission to write more than 1 function).

* no permission to use loop inside the function.

Write a **recursive** function to do the following:

The function should will copy the **(N)** nodes and paste them at the end of the list.

**e.g. 🡪 N = 3**

L

Head

**The list after calling of your function will be:**

L

Head

void CopyPaste(CNode\* pT, CList& L, int i, int N) //assuming tail is present

{

if (i == N)

{

return;

}

CNode\* pnn = new CNode;

pnn->info = pT->info;

pnn->pNext = NULL;

L.pTail->pNext = pnn;

L.pTail = pnn;

CopyPaste(pT->pNext, L, i + 1, N);

}

void CopyPaste(CNode\* pT, CNode\* pKing, int& i, int N) //assuming no tail is present

{

if (i==N)

{

return;

}

if (pT->pNext == NULL)

{

i++;

CNode\* pnn = new CNode;

pnn->info = pKing->info;

pKing=pKing->pNext;

pnn->pNext = NULL;

pT->pNext = pnn;

}

CopyPaste(pT->pNext,pKing,i, N);

}

**Problem 2: (26 marks)**

Write a main function to do the following:

1. Read a Binary Search Tree.
2. Find the Largest Node (call it **pLargest**)

e.g. Largest 🡪 15000

Root

**pLargest**

10250

10600

10400

10200

920

15000

1200

950

970

1180

980

1100

1000

1700

10000

84

81

83

85

80

900

850

250

280

90

200

910

800

500

1. Calculate the total of the sub-tree of (**pLargest)**

**in case the total ≠ largest value**

**then cut the pLargest from your tree & paste it in a linked list of trees (LL).**

**in our example: total = 41450**

**41450 ≠ 15000 🡪 so we will cut & past it to (LL).**

10250

10600

10400

15000

**pLargest**

84

81

83

85

80

10000

970

1180

980

1100

1000

10200

1700

1200

950

900

850

250

280

90

200

910

800

500

Root

920

**LL**

**pHead**

10250

10600

10400

15000

10200

Your tree after cutting pLargest

970

980

1100

1000

1700

1200

950

900

850

250

280

90

200

910

800

500

Root

920

84

81

83

85

80

10000

1180

1. Repeat steps (2) & (3), till reach pLargest that has Sub-tree with total = pLargest.

In our example : **total = 10000**

**10000 = pLargest (10000) 🡪**

so we will generate many trees as a **copy** from pLargest without 1 leaf each time & **past** it to (**LL**).

1180

920

84

81

83

85

80

10000

970

980

1100

1000

1700

1200

950

900

850

250

280

90

200

910

800

500

Root

Copy of pLargest’s tree

But without Leaf (970)

Copy of pLargest’s tree

But without Leaf (920)

**LL**

**pHead**

1000

1700

1200

950

1180

920

10000

980

1100

980

1100

1000

1700

1200

950

10000

1180

970

10250

10600

10400

15000

10200

Copy of pLargest’s tree

But without Leaf (1180)

Copy of pLargest’s tree

But without Leaf (1700)

1180

920

10000

970

980

1100

1000

1200

950

1100

1000

1700

1200

950

920

10000

970

980

void CalcTot(CTNode\* pT, int& tot)

{

if (pT == NULL)

{

return;

}

CalcTot(pT->pLeft, tot);

tot += pT->info;

CalcTot(pT->pRight, tot);

}

void CountLeafs(CTNode\* pT, int& ctleafs)

{

if (pT == NULL)

{

return;

}

CountLeafs(pT->pLeft, ctleafs);

if (pT->pLeft == NULL && pT->pRight == NULL)

{

ctleafs++;

}

CountLeafs(pT->pRight, ctleafs);

}

void CopyTree(CTNode\*pT,int&k, int i,CList&LL,CBST T)

{

if (pT == NULL)

{

return;

}

CopyTree(pT->pLeft, k, i,LL,T);

if (pT->pLeft == NULL && pT->pRight == NULL)

{

if (i != k)

{

CTNode\* pnT = new CTNode;

pnT->info = pT->info;

pnT->pLeft = pnT->pRight = NULL;

T.Insert(pnT);

}

k++;

}

else

{

CTNode\* pnT = new CTNode;

pnT->info = pT->info;

pnT->pLeft = pnT->pRight = NULL;

T.Insert(pnT);

}

CopyTree(pT->pRight, k, i, LL, T);

}

void main()

{

CBST CTree;

CBST T;

CList LL;

int N;

cin >> N;

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

{

CTNode\* pnn = new CTNode;

cin >> pnn->info;

pnn->pLeft = pnn->pRight = NULL;

CTree.Insert(pnn);

}

int tot = 0;

CTNode\* pLargest = CTree.pRoot;

CTNode\* pB = NULL;

while (1)

{

tot = 0;

pLargest = CTree.pRoot;

pB = NULL;

while (pLargest->pRight != NULL)

{

pB = pLargest;

pLargest = pLargest->pRight;

}

CalcTot(pLargest, tot);

if (tot != pLargest->info)

{

CNode\* pnn = new CNode;

pnn->pDown = pLargest;

pnn->pNext = NULL;

pB->pRight = NULL;

LL.Attach(pnn);

}

else

{

break;

}

}

int ctleafs = 0,k=0;

CountLeafs(pLargest, ctleafs);

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

{

CNode\* pnn = new CNode;

pnn->pDown = T.pRoot;

pnn->pNext = NULL;

k = 0;

CopyTree(T.pRoot,k,i,LL,T);

LL.Attach(pnn);

T.pRoot = NULL;

}

}