18L-1223

Ikram ul Hassan

Data Structures

Q1) a) Time Complexity analysis:

```
void findTopPlayers(Player** playerlist, int N, Player** topPlayerList)
{
    sort(playerlist, N);          → N^2
    for(int i=0; i<3; i++)        → 3
    {
        topPlayerlist[i] = playerlist[i];
    }
}

void sort(Player** playerlist, int N)
{
    int largest;                  → c
    for(int j=0; j<N-1; j++)      → (N-1)
    {
        largest = j;              → c
        for(int i=j+1; i<N; i++)  → (N-1)
        {
            if(playerlist[i]->getScore() > playerlist[i]->getScore())
            {
                largest = i;      → c
            }
        }
        swap(playerlist[j], playerlist[largest]);  → c
    }
}
```

Time complexity of Sort function is:

$= c + (N-1+c+c)(N-1+c)$

$\{ c + (N-1+2c)(N-1+c)$

* Ignoring constants

$= (N+1)(N)$

$= N^2$

$= O(N^2)$

Time Complexity of Find Top Players is:

$= N^2+3$

Ignoring constants

$= N^2$

$= O(N^2)$

So, the time complexity of FindTop Players is $O(N^2)$

b) The better solution will be done by using min-heap and then heap Sort for sorting.

```
void heapSort (Player** Players, int n)
 {
     for(int i= n/2 -1 ; i>=0 ;i--)
        call heapify Players, n, i) //For building heap

     for (int i=n-1 ; i>=0 ; i--)
      {
         Swap the first and last element
      }

     Call heapify again
        heapify (Players, i, 0);
}

void heapify (Player** Players, int n, int i)
    {
     int Smallest = i; //As root
     int l= left child of smallest
     int r= right child of smallest
     if (l<n && Players[l] < Players[smallest])
         Smallest = l;
     if (r<n && Players[r] < Players[smallest])
         Smallest=r;
      if (smallest != i){
        Swap Players(i) and Players[smallest]
        heapify (Players, n, smallest);
    }
```

```
void findTopPlayers(Player* TopPlayerslist, int n, Player** playerslist)
{
    heapsort(Playerslist, n);   → n log n
    for(int i=0; i<3; i++)      → 3
        TopPlayerslist[i] = Playerslist[i];
}
```

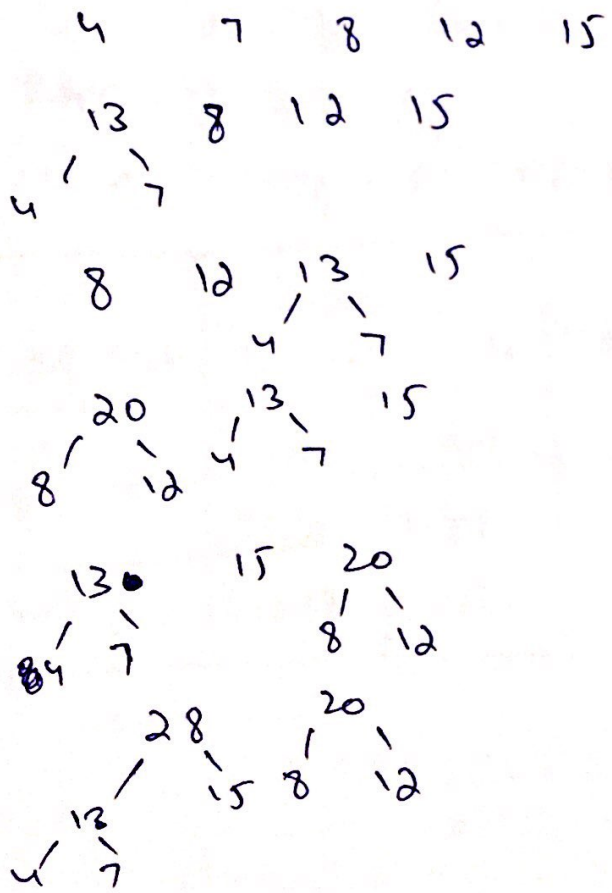
The time complexity of above solution will be:

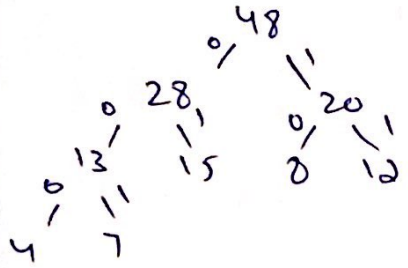
$= (n \log n) + 3$

$= n \log n$

$= O(n \log n)$

Q3) All codes are wrong





Find Codes are

a	01
h	000
m	10
t	001
Space	11

So, all the options are wrong.

Q5)

a) CPU - Capacity = 1.60GHz

RAM = 4GB

Virtual Memory Size: 2048MB

b)

Input Size	Recursive			Iterative		
	Inorder	Preorder	Postorder	Inorder	Preorder	Postorder
10	3200	22401	3840	906907	1959099	271369
50	7680	12800	48642	563857	3136734	2772584
500	71643	391652	84402	11595 1008	15579988	2255237

c)

	Recursive	Iterative
Post Order	O(n)	O(n)
Pre-Order	O(n)	O(n)
Inorder	O(n)	O(n)

---

d) ~~Recursive~~ ~~Iterative~~ version grows faster i.e take less time. The difference in computational time is due to the memory. Since, iterative solution uses stack and check for its right and left child and then do process, so it ~~but~~ takes more time in terms of recursion.

Q6) c) In worst case:

The time complexity of insertion will be O(n).

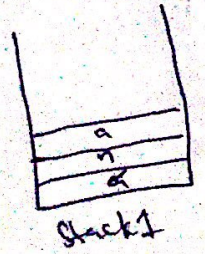
The time complexity of searching will be O(n).

Q2) b) The time complexity will be $O(n^2)$.

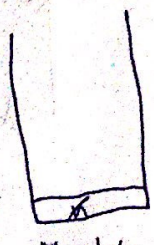
c) S= "Iteza2 w Hassan" len=16 flag= false b= null

Dry Run:

i) a='h'



ii) a='a'
flag=false
while(Stack1 is not empty)
b='h'
while(Stack2 is not empty)
b2='n'
if a!=b and flag=true



iii) a = 's'

while (Stack1 is not empty)

b = " a "

b = " n "

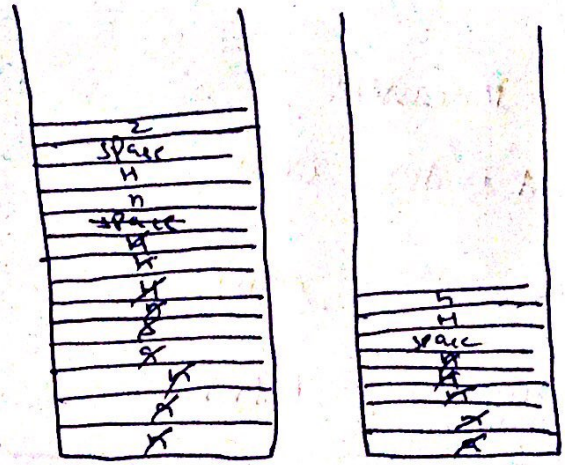
while (Stack 2 is not empty)

b = 'n'

b = 'a'

if (a != b && flag != true)

Stack2.push(a)



iv) a = 's'

while (Stack1 is not empty)

b = 's'

if (a == b)

flg = true

break;

v) a = 'a'

while (Stack1 is not empty)

b = 'a'

if (a == b)

flg = true

break;

vi) a = 'H' flag = false

while (Stack1 is not empty)

b = 'n'

while (Stack2 is not empty)

b = 'n'

if (a != b && flg = true)

Stack1.push(a)

vii) a = "n"

while (Stack1 is not empty)

b = 'H'

b = 'n'

while (Stack2 is not empty)

b = 'n'

b = 'H'

if (a != b && flg = true)

Stack1.push(a)

viii) a = 'z'

while (Stack1 is not empty)

b = " "

b = 'H'

b = 'n'

while (Stack2 is not empty)

b = 'n'

b = 'H'

b = 'space'

if (a != b && !flg)

Stack1.push(a)

iv) a = 'q'

while (Stack1 is not empty)
b = 'e'
b = 'space'
b = 'H'
b = 'n'
while (Stack2 is not empty)
b = 'n'
b = 'H'
b = 'space'
b = 'a'
if (a != b && !flag)
Stack1.push(a)

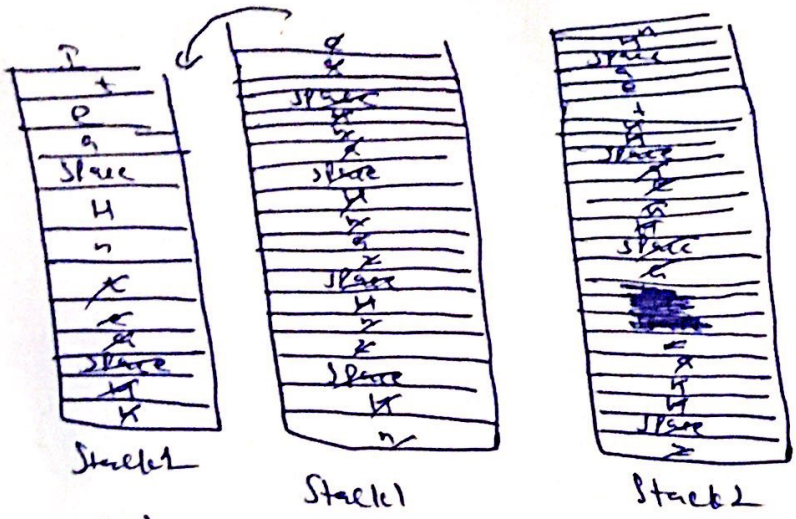
v) a = 'z'

while (Stack1 is not empty)
b = 'e'
b = 'z'
b = 'space' a == b
flag = true
break)
b = 'n'

while Stack2 is not empty
b = 'n'
b = 'H'
b = 'space'
b = 'a'

vi) a = 'e'

while (Stack1 is not empty)
b = 'a'
b = 'space'
b = 'H'
b = 'n'
while (Stack2 is not empty)
b = 'n'
b = 'H'
b = 'space'
b = 'a'
Stack1.push(a)



vii) a = 'I'

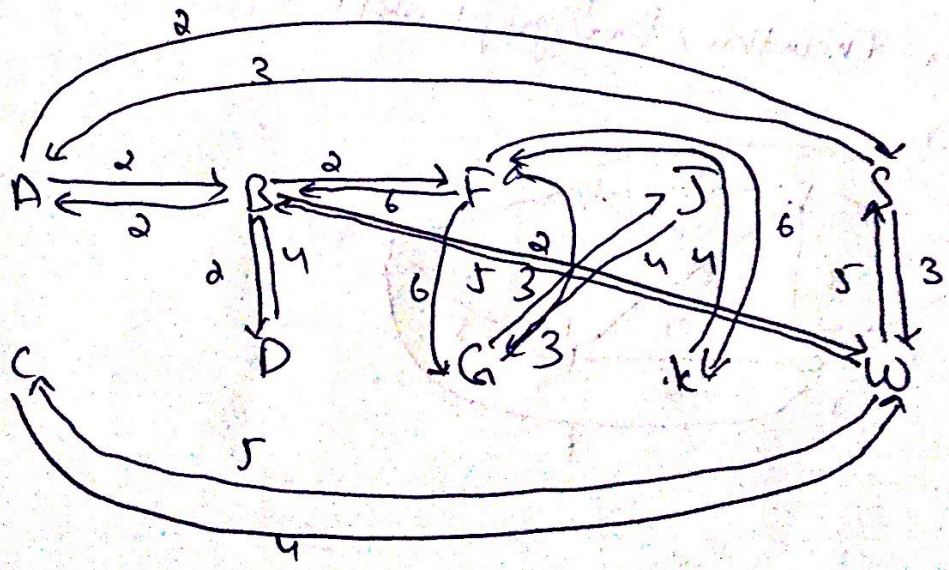
while (Stack1 is not empty)
b = 'I'
b = 'e'
b = 'a'
b = 'space'
b = 'H'
b = 'n'
while (Stack2 is not empty)
b = 'n'
b = 'H'
b = 'space'
b = 'a'
b = 'e'
b = 'I'
Stack1.push(a)
Output = Iteza ul Hn

ii)

Nodes	Final Cost
A	0
B	3
C	5
D	7
E	9
F	9

b) No, we can not do that, because Spanning tree is built by maximum number of edges irrespective to its cost. In any Spanning tree, maximum number of edges are used always. So, if we add another edge, it will increase its cost in man. Spanning tree but then it will not be a Spanning tree. So, In conclusion we can not do that.

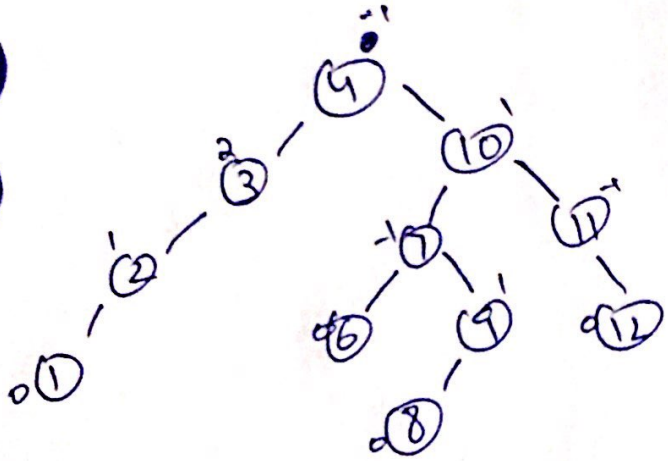
c) i)



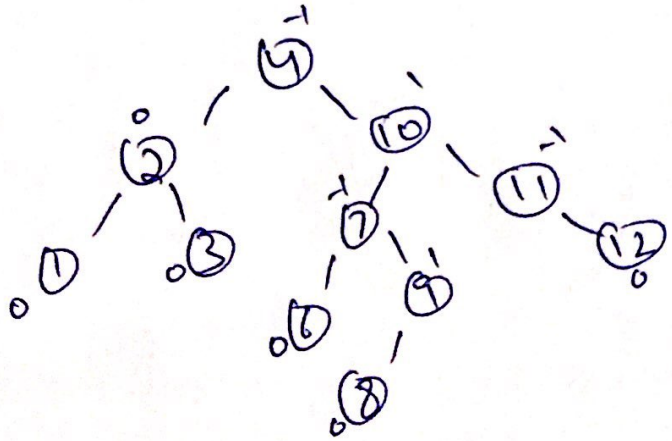
ii) Yes, it is possible, using Prims algorithm.

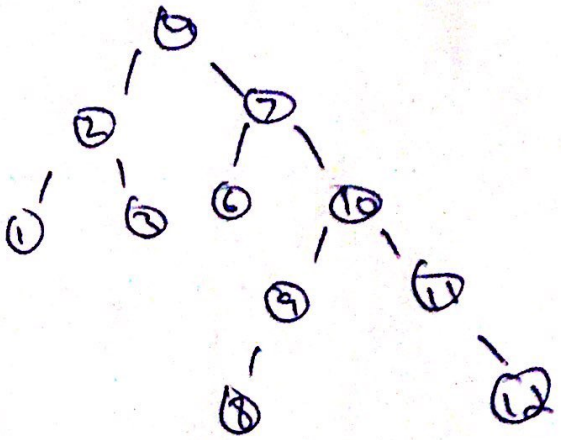
Q7)

a)



b)





Final :