National University of Computer and Emerging Sciences, Lahore Campus

Course:	**Data Structures**	**Course Code:**	**CS-218**
Program:	**BS (Computer Science)**	**Semester:**	**Spring 2020**
Duration:	**5 hrs**	**Total Marks:**	**50**
Paper Date:	**29th June-2020**	**Page(s):**	**7**
Section:	**All sections**	**Roll No.:**	
Exam:	**Final Term**		

Instruction/Notes:

- Understanding question is a part of your exam. In case of ambiguity, write your assumptions and answer accordingly
- Time management is the key to success
- Upload codes with your final answer sheets. Failing in doing so will earn you **NO** credit
- All the submitted codes will be checked for plagiarism
- Question # 2 and Question # 6 must be submitted as a '.cpp' file. Remaining questions are required to be solved on answer sheets.

Q1: **[2+3=5]**

Consider a function *findTopPlayers* which receives a list of N players with in a tournament and returns top 3 players based on their scores.

```
void sort(Player** playersList, int N) {

      int largest;
      for (int j = 0; j < N - 1; j++) {
            largest = j;
            for (int i = j + 1; i < N; i++) {
            if (playersList[i]->getScore() > playersList[largest]->getScore())
              largest = i;
            }
            swap(playersList[j], playersList[largest]);
      }
}

void findTopPlayers(Player** playersList, int N, Player** topPlayersList) {

      sort(playersList, N);

      for (int i = 0; i < 3; ++i) {
            topPlayersList[i] = playersList[i];
      }

}
```

(a) Do the time complexity analysis of the function *findTopPlayers*.

Solution:

Statement	No. of times executed
`for (int j = 0; j < N - 1; j++) {`	N
`largest = j;`	N
`for (int i = j + 1; i < N; i++) {`	$\frac{N(N+1)}{2}$
`if (playersList[i]->getScore() > playersList[largest]->getScore())`	$\frac{N(N+1)}{2}$
`largest = i;`	$\frac{N(N+1)}{2}$
`swap(playersList[j], playersList[largest]);`	N
`for (int i = 0; i < 3; ++i) {`	3
`topPlayersList[i] = playersList[i];`	3

$$T(N) = N + N + \frac{N(N+1)}{2} + \frac{N(N+1)}{2} + \frac{N(N+1)}{2} + N + 3 + 3$$

$$T(N) = \frac{3N(N+1)}{2} + 3N + 6$$

$$T(N) = O(N^2)$$

(b) Devise a better solution for this problem (and write its pseudocode) to improve the running time of this algorithm. What will be the time complexity of your improved algorithm?

Solution:

As the function returns top 3 players, so we assume that the *playersList* contains at least 3 players.

```
void findTopPlayers(Player** playersList, int N, Player** topPlayersList) {
	topPlayersList[0] = NULL;
	topPlayersList[1] = NULL;
	topPlayersList[2] = NULL;

	for (int i = 0; i < N; ++i) {
		if (topPlayersList[0] == NULL ||
		    playersList[i]->getScore() > topPlayersList[0]->getScore()) {
			topPlayersList[2] = topPlayersList[1];
			topPlayersList[1] = topPlayersList[0];
			topPlayersList[0] = playersList[i];
		}
		else if (topPlayersList[1] == NULL ||
			    playersList[i]->getScore() > topPlayersList[1]->getScore()) {
			topPlayersList[2] = topPlayersList[1];
			topPlayersList[1] = playersList[i];
		}
		else if (topPlayersList[2] == NULL ||
			    playersList[i]->getScore() > topPlayersList[2]->getScore()) {
			topPlayersList[2] = playersList[i];
		}
	}
}
```

Another solution:

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

void findTopPlayers(Player** playersList, int N, Player** topPlayersList) {

	sort(playersList, N, 3);

	for (int i = 0; i < 3; ++i) {
		topPlayersList[i] = playersList[i];
	}
}
```

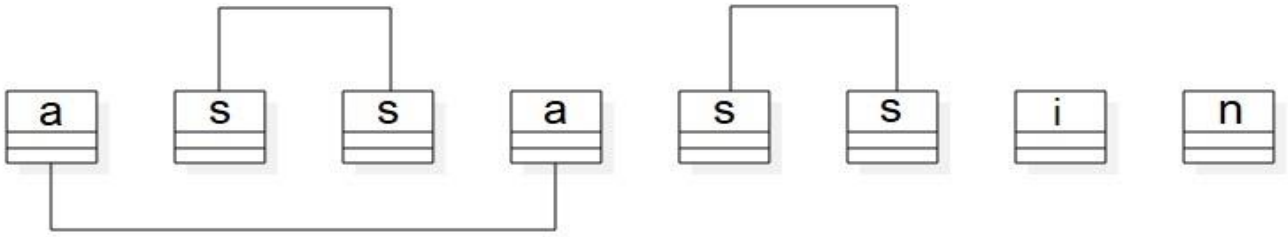
The time complexity of both solutions is $O(N)$.

Q2: **[6+2+2=10]**

Given a string containing only lowercase letters and spaces, remove pairs of matching characters. This matching starts from the end of the string. For example,

Input: "assassin"
Output: "in"
Input: "pacific ocean"
Output: "pcf oen"
Input: "an assassin sins"
Output: "an"

The first case has been elaborated below:



Remember that the matching starts from the end of the string. So, for input string "sadness", the output must be "sadne" and not "adnes".

a) Write a C++ implementation for the function `removePairs` to solve the above mentioned problem. You're allowed to use only **two stacks and a constant number of variables**. The minimum program structure is shown below

Solution:

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

string removePairs(string input) {
	stack <char> stack1;
	stack <char> stack2;

	for (int i = 0; i < input.length(); ++i)
		stack1.push(input.at(i));

	char topOfStack2 = '\0';
	while (!stack1.empty()) {
		char C = stack1.top();	stack1.pop();

		while (!stack1.empty() && stack1.top() != C) {
			stack2.push(stack1.top());
			stack1.pop();
```

```
            }

            bool matched = false;
            if (!stack1.empty()) {
                  matched = true;
                  stack1.pop();
            }

            while (!stack2.empty() && stack2.top() != topOfStack2) {
                  stack1.push(stack2.top());
                  stack2.pop();
            }

            if (!matched) {
                  stack2.push(C);
                  topOfStack2 = C;
            }
      }

      string result;

      while (!stack2.empty()) {
            char C = stack2.top();    stack2.pop();
            result.push_back(C);
      }
      return result;

}
int main() {

      cout << removePairs("assassin") << endl;
      cout << removePairs("an assassin sins") << endl;
      cout << removePairs("pacific ocean") << endl;

      return 0;
}
```

b) What is the overall computational complexity of your program in terms of Big-Oh?

Solution: $O(N^2)$

c) Show the dry run of your algorithm with your name as the input string and write the final output of your algorithm. (Use your complete official name registered in FLEX at FAST-NU)

Q3: **[2]**

Three agents were working on a mission when they came across a secret message. They decided to deliver this message to their boss but they can't deliver the message as it is because it's very sensitive information. They decided to encrypt it and then send it to their boss. All of the three agents tried to encode it and came up with the following Huffman codes:

Letter	Frequency	Agent 1	Agent 2	Agent 3
a	15	00	11	01
h	4	111	000	110
m	8	10	01	10
t	7	110	001	111
space	12	01	10	00

Choose one of the following options and Justify your answer:

i. All codes are wrong
ii. All codes are correct
iii. Only one of them is correct (mention that agent)

Solution: All codes are correct.
Huffman codes can be different because different initial arrangements of the characters will yield different outputs. However, in any case, the following two conditions are always fulfilled:

1. These are prefix codes i.e. no code is a prefix of any other code
2. If different Huffman codes are possible, codes of the same letter are always of the same length

Q4: **[4+1+(2+3)=10]**

a. It is possible to determine the shortest distance (or the least effort / lowest cost) between a start node and any other node in a graph. The idea of the algorithm is to continuously calculate the shortest distance beginning from a starting point, and to exclude longer distances when making an update. FINDPATH function shown below calculates the shortest path from a given source vertex towards the destination.

```
1: function FINDPATH(Graph, source):
2:     for each vertex v in Graph:                // Initialization
3:         dist[v] := infinity                    // initial distance from source to vertex v is set to infinite
4:         previous[v] := undefined               // Previous node in optimal path from source
5:     dist[source] := 0          // Distance from source to source
6:     Q := the set of all nodes in Graph         // all nodes in the graph are unoptimized - thus are in Q
7:     while Q is not empty:                      // main loop
8:         u := node in Q with smallest dist[ ]
9:         remove u from Q
10:        for each neighbor v of u:              // where v has not yet been removed from Q.
11:            alt := dist[u] + dist_between(u, v)
12:            if alt < dist[v]                   // Relax (u,v)
13:                dist[v] := alt
14:                previous[v] := u
15:    return previous[ ]
```

Given the pseudocode shown above, and the graph shown in Fig.1, perform the following tasks

i. Take vertex ‘A’ an input, show every transition state of Q along with the updated values of cost for each vertex.

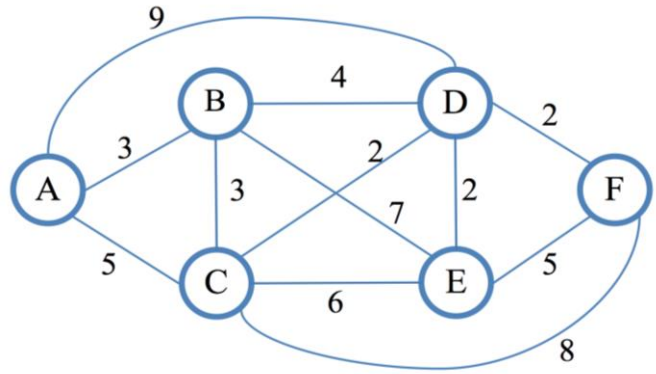


Fig. 1: Input Graph

Solution:

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	1	1	0
C	1	1	0	1	1	1
D	1	1	1	0	1	1
E	0	1	1	1	0	1
F	0	0	1	1	1	0

After execution of lines 2 – 6, the State of the Q is

Cost	0	∞	∞	∞	∞	∞
Q	A	B	C	D	E	F

1. After execution of line 7, the condition is true, so the Vertex ‘A’ with smallest distance value is removed from the Q. All the adjacent vertices of A are explored one by one and checked for the smallest distance available. So, after exploring all the adjacent vertices of vertex A, the overall values of smallest distance, and the previous vertex to be visited are as follows

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	A	9
E	Undefined	∞
F	Undefined	∞

And the status of the **Q** is

Cost	3	5	9	∞	∞
Q	B	C	D	E	F

2. Vertex B will be removed from the Q as it has the smallest distance

The adjacent vertices of B still not removed from the Q are C, D and E. The cost of moving from B to C is 6, the current cost of C is 5. Therefore, the condition at line 12 is false. The cost of moving from B to D is 7 which is less than the previous calculated distance. Therefore, the cost of the path is updated. The cots of moving from B to E is 10 which is smaller than ∞. Therefore, the cost of the path for vertex E is updated

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	B	7
E	B	10
F	Undefined	∞

And the current status of Q is

Cost	5	7	10	∞
Q	C	D	E	F

3. Vertex C will be removed from the Q as it has the smallest distance value. The neighboring vertices of C are D, E, and F. The updated table and the status if Q is as follows

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	B	7
E	B	10
F	C	13

Cost	7	10	13
Q	D	E	F

4. Now, D will be removed from the Q and explored for its adjacent vertices that are still in Q. The resulting table and the status of Q is as follows

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	B	7
E	D	9
F	D	9

Cost	9	9
Q	E	F

5. Vertex E will be removed from the Q and the only adjacent neighbor still part of the Q is F. No value will be updated so the resulting Q and table is shown below

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	B	7
E	D	9
F	D	9

Cost	9
Q	F

6. Vertex F will be removed from the Q and the resulting table and empty Q is shown

Current Node	Previous	Distance
A	NIL	0
B	A	3
C	A	5
D	B	7
E	D	9
F	D	9

Cost
Q

ii. Fill the following table, with NODES and total final cost

Solution:

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

b. If we add another edge into a Minimum Spanning Tree, it doesn't remain minimum anymore. But for a Maximum Spanning Tree, it makes sense because adding one or more edges to it will further increase its overall weight/cost. So, can we really do that? Justify your answer.

Solution: No, we can't do that because if add one more edge in a Maximum Spanning Tree, it will not remain a tree anymore.

c. Mr. X has just shifted in his brand new house. The floor plan for the ground floor is shown in Fig 2. Mr. X wants to give a tour of his new home to an old friend. The doors within the rooms are represented by the alphabets for simplicity. And the integers in the rooms are cost of moving from one door to another door. This means, in room 2 the cost of moving from door B to door W is 2, from door B to door D is 2, and from door B to door F is also 2. Similarly, in room 3 the cost of moving from door F to door G is 6, from door G to door K is 6 and from door F to door K is also 6.

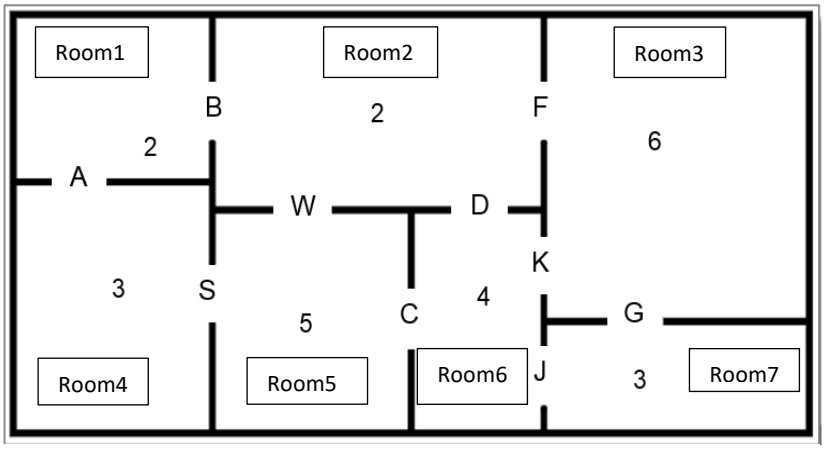
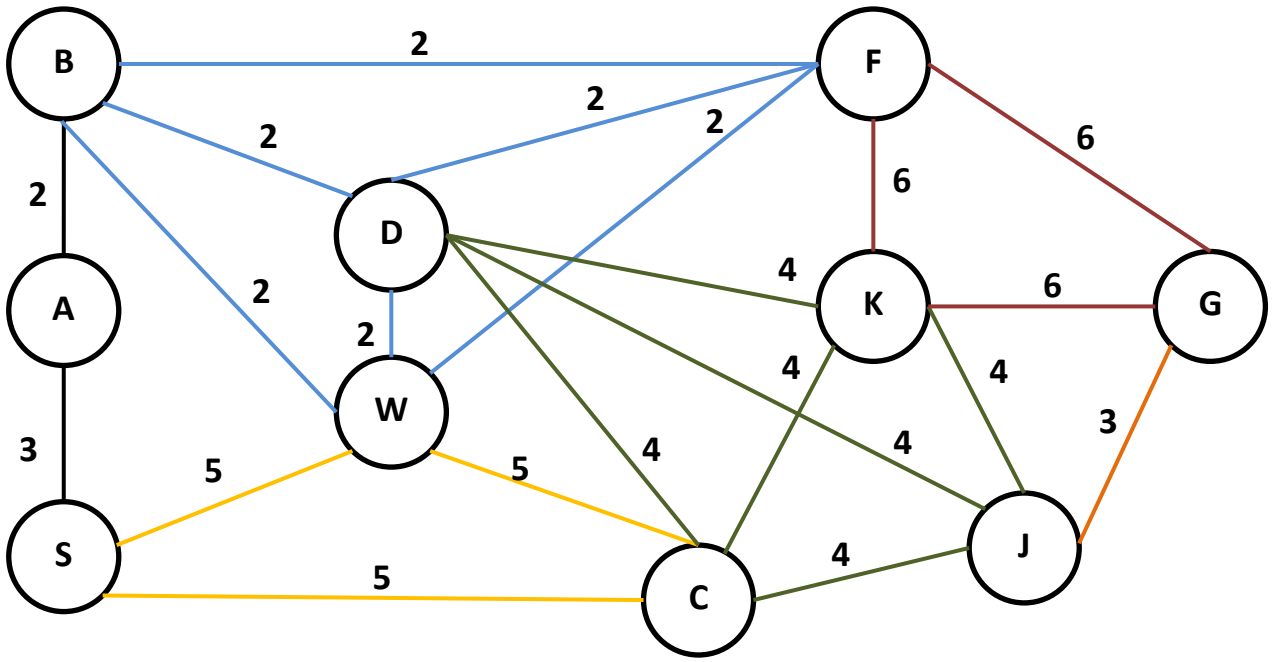


Fig. 2: Structural Map of House

i. Given the map, shown in Fig. 2, Construct a graph to represent the structure of the ground floor.

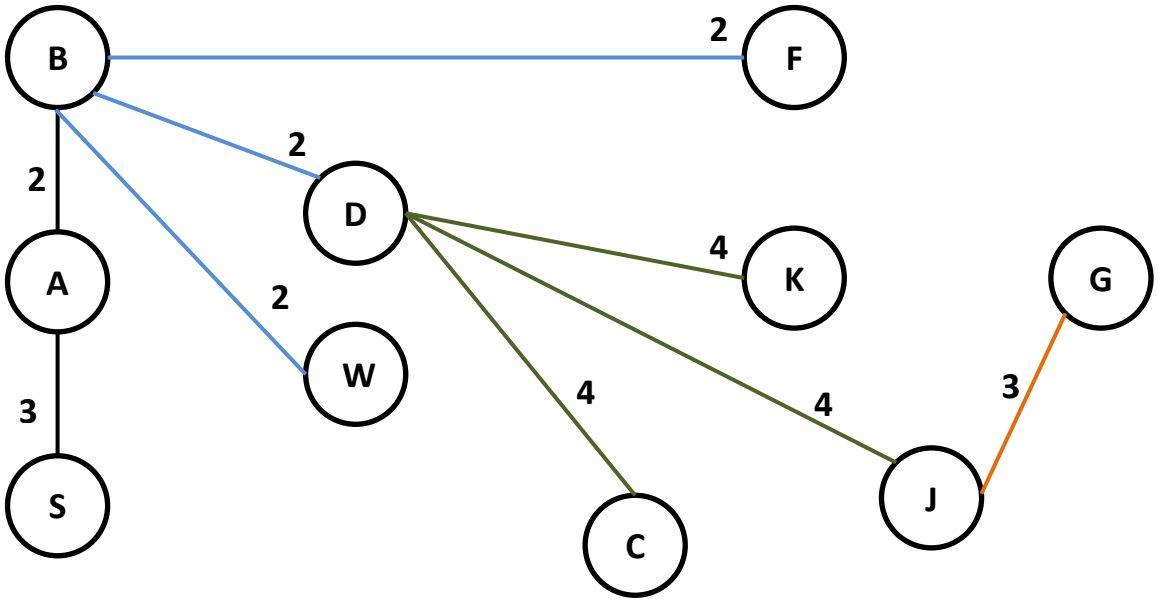
Solution:



ii. Is it possible for them to walk through every doorway exactly ***once*** with the minimum cost tree? Suggest name of one algorithm and then dry run your algorithm to show each transition.

Solution:

Kruskal's Algorithm



Minimum Cost Tree = 2+2+2+2+3+3+4+4+4=26

Q5: **[1+3+1+1 =6]**

There are three tree traversal methods in-order, pre-order and post-order. Given the iterative and recursive code *(Q5-TreeTraversals.cpp)* provided answer the following questions

a. Write down specifications of your system i.e. CPU capacity, RAM, Cache (if applicable) and virtual memory size [The specification will directly impact the values in part b]

Solution: CPU: 2.40GHz, Ram 8GB, Virtual Memory =2816MB

b. With different input sizes mentioned, find the elapse time required for in-order, preorder and post-order traversal using recursive solution and iterative solution, and fill the following table

Solution:

Table 1: Comparative analysis

Input Size	Recursive Solution (Time in nano secs)			Iterative Solution (Time in nano secs)		
	In-order	**Preorder**	**Post-order**	**In-order**	**Preorder**	**Post-order**
10	382	152	166	2804	1203	708
50	599	392	395	4825	1654	1559
500	2855	2271	2273	3788	3563	5673

c. What is the efficiency in terms of Big Oh notation for each algorithm (recursive and iterative)?

Solution: The efficiency of all the algorithms is $O(n)$.

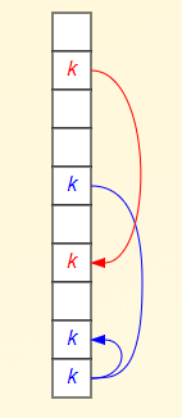
d. What trend do you observe in table 1? Which version (recursive/iterative) in general grows faster? How can you justify difference in computational time? Write crisp and clear statements. Ambiguous statements won't earn you any credit.

Solution: Both versions grow at the same rate because complexity of both of them is $O(n)$ but iterative version is taking more time as it's doing some extra computation.

Q6: **[(2+3+2)+2+1=10]**

Coalesced hashing is a technique for implementing a hash table. It's an open addressing technique which means that all keys are stored in the array itself. However, as opposed to other open addressing techniques, it also uses nodes with next-pointers to form collision chains. Coalesced hashing, also called coalesced chaining, is a strategy of collision resolution in a hash table that forms a hybrid of separate chaining and

open addressing. For example, Coalesced hash table holding five keys in two collision chains is shown in the following figure (Keys of the same color hash to the same bucket.)



It uses the concept of Open Addressing(linear probing) to find first empty place for colliding element and the concept of Separate Chaining to link the colliding elements to each other through pointers. Inside the hash table, each node has three fields, ***h(key):*** The value of hash function for a key, ***Data:*** The key itself, ***Next:*** The link to the next colliding elements.

Modify the *Q6-hashingLinearProbing.cpp* to perform the following operations for Coalesced hashing and the following functions

a. INSERT(key): The insert Operation inserts the key according to the hash value of that key if that hash value in the table is empty otherwise the key is inserted in first empty place found after applying QUADRATIC PROBING.
b. SEARCH(key): Returns True if key is present, otherwise return False.

```
#include<iostream>
using namespace std;

//template for generic type
template<typename K, typename V>

//Hashnode class
class HashNode
{
public:
      V value;
      K key;
    HashNode *next;

      //Constructor of hashnode
      HashNode(K key, V value)
      {
            this->value = value;
            this->key = key;

      }
};

//template for generic type
```

```
template<typename K, typename V>

//Our own Hashmap class
class HashMap
{
      //hash element array
      HashNode<K, V>** arr;
      int capacity;
      //current size
      int size;
      //dummy node

      HashNode<K, V>* dummy;

     //HashMap *next;
public:
      HashMap()
      {
            //Initial capacity of hash array
            capacity = 10;
            size = 0;

            arr = new HashNode<K, V> * [capacity];

            //Initialise all elements of array as NULL
            for (int i = 0; i < capacity; i++)
                  arr[i] = NULL;

            //dummy node with value and key -1
            dummy = new HashNode<K, V >(-1, -1);

      }
      // This implements hash function to find index
      // for a key
      int hashCode(K key)
      {
            return key % capacity;
      }

      //Function to add key value pair
      void insertNode(K key, V value)
      {
            bool flag = false;
            int probe = 0;
            int oldHash = 0;

            HashNode<K, V>* temp = new HashNode<K, V>(key, value);
            temp->next = NULL;

            // Apply hash function to find index for given key
            int hashIndex = hashCode(key);

            HashNode<K, V>* previous = arr[hashIndex];
```

```
            //find next free space
            while (arr[hashIndex] != NULL && (arr[hashIndex]->key != key ||
arr[hashIndex]->key != -1))
            {
                  flag = true;
                  cout << "\n .........Collision is Detected.......";

                  oldHash = hashIndex;
                  probe++;
                  key = key + (probe * probe);
                  hashIndex = hashCode(key);

                  if (arr[hashIndex] != NULL && previous->key ==
arr[hashIndex]->key)
                        previous = arr[hashIndex];

                  cout << "\nProbe =" << probe << "," << "HashIndex = " <<
hashIndex << endl;
            }

            //if new node to be inserted increase the current size
            if (arr[hashIndex] == NULL || arr[hashIndex]->key == -1)
                  size++;

            arr[hashIndex] = temp;
            if (flag)
                  previous->next = arr[hashIndex];
      }

      //Function to delete a key value pair
      V deleteNode(int key)
      {
            // Apply hash function to find index for given key
            int hashIndex = hashCode(key);

            //finding the node with given key
            while (arr[hashIndex] != NULL)
            {
                  //if node found
                  if (arr[hashIndex]->key == key)
                  {
                        HashNode<K, V>* temp = arr[hashIndex];

                        //Insert dummy node here for further use
                        arr[hashIndex] = dummy;

                        // Reduce size
                        size--;
                        return temp->value;
                  }
                  hashIndex++;
                  hashIndex %= capacity;

            }
```

```
            //If not found return null
            cout << "Key not found";
            return 0;
      }

      //Function to search the value for a given key
      V get(int k, int val)
      {
            // Apply hash function to find index for given key
            int key = k ;
            int hashIndex = hashCode(key);
            int counter = 0;
            int probe = 0;
            //finding the node with given key
            while (arr[hashIndex] != NULL)
            {
                  //int counter = 0;
                  if (counter++ > capacity) //to avoid infinite loop
                        return -20;
                  //if node found return its value
                  if (arr[hashIndex]->key == k && arr[hashIndex]->value ==
val){
                      cout <<"\n\n Value Found:" ;
                  return arr[hashIndex]->value;
                  }

                 probe++;
                  key = key + (probe*probe);
                  cout<< "key="<< key << "Probe="<< probe <<endl;
                  hashIndex = hashCode(key);
            }

            //If not found return -10
            return -10;
      }

     //Search Key
      bool search(int key)
      {
            // Apply hash function to find index for given key
            int hashIndex = hashCode(key);
            int counter = 0;
            int probe = 0;
            //finding the node with given key
            while (arr[hashIndex] != NULL)
            {
                  //int counter = 0;
                  if (counter++ > capacity) //to avoid infinite loop
                        return false;
                  //if node found return its value
                  if (arr[hashIndex]->key == key)
                        return true;
```

```
                probe++;
                key = key + (probe * probe);
                hashIndex = hashCode(key);
           }
           return false;
     }

     //Return current size
     int sizeofMap()
     {
           return size;
     }

     //Return true if size is 0
     bool isEmpty()
     {
           return size == 0;
     }

     //Function to display the stored key value pairs
     void display()
     {
           for (int i = 0; i < capacity; i++)
           {
                if (arr[i] != NULL && arr[i]->key != -1)
                      cout << arr[i] << "=====>>"<< "key = " << arr[i]->key
<< " , value = " << arr[i]->value << ", next = " << arr[i]->next <<",  Index=
" <<i<< endl;
                //cout << i << endl;
           }
     }
};

//Driver method to test map class
int main()
{
     HashMap<int, int>* h = new HashMap<int, int>;

     h->insertNode(1, 1);
     //h->display();
     h->insertNode(2, 26);

     h->insertNode(3, 13);
     h->display();

     h->insertNode(2, 39);
     h->insertNode(2, 93);
     h->insertNode(4, 87);
    h->display();

     /*h->display();

     h->insertNode(3, 33);
     h->insertNode(2, 60);
```

```
	cout << "h->sizeofMap()" << endl;
	cout << h->sizeofMap() << endl;
	cout << "h->display()";
	h->display();

	cout << h->deleteNode(2) << endl;
	cout << "h->sizeofMap()" << endl;
	cout << h->sizeofMap() << endl;
	h->display();
	cout << h->isEmpty() << endl;*/

	cout << h->get(2,39) << endl;

	system("pause");
	return 0;
}
```

c. What is the worst case complexity analysis for insertion and searching in hash table using coalesced hashing?

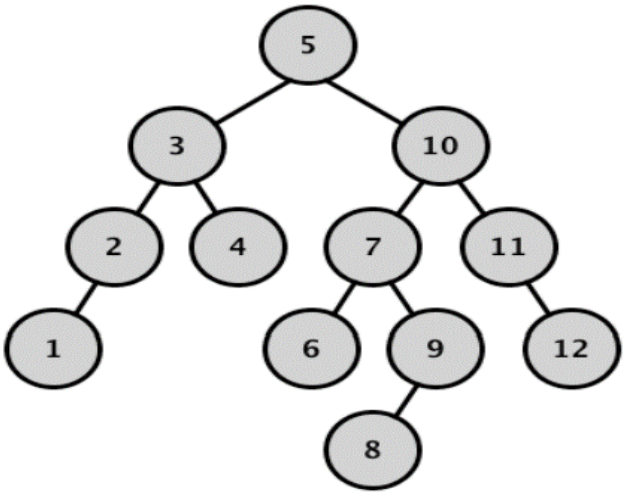
Solution: $O(N)$.

The hash function used is **h=(key)% (total number of keys)**
For the given input sequence,
Input: {20, 35, 16, 40, 45, 25, 32, 37, 22, 55},
the output of the Coalesced hash table is shown for first 4 inputs

After Inserting 20			After Inserting 35 and 16			After Inserting 40 (Collision is detection and resolved using quadratic probing)		
Hash Value	Data	Next	Hash Value	Data	Next	Hash Value	Data	Next
0	20	Null	0	20	Null	0	20	1 (Indicating Chaining)
1		Null	1		Null	1	40	Null
2		Null	2		Null	2		Null
3		Null	3		Null	3		Null
4		Null	4		Null	4		Null
5		Null	5	35	Null	5	35	Null
6		Null	6	16	Null	6	16	Null
7		Null	7		Null	7		Null
8		Null	8		Null	8		Null
9		Null	9		Null	9		Null

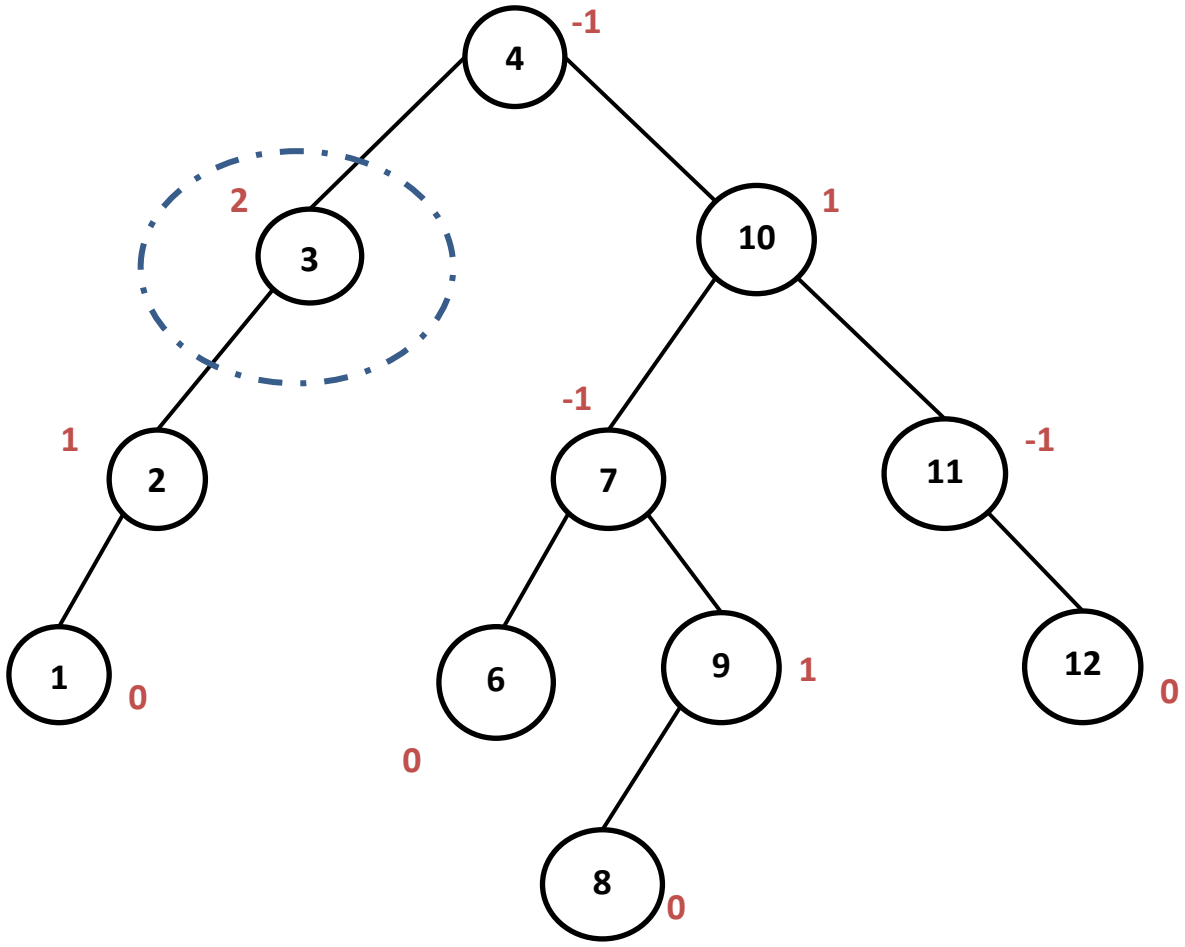
Q7: **[3+4=7]**

Given the following AVL Tree:



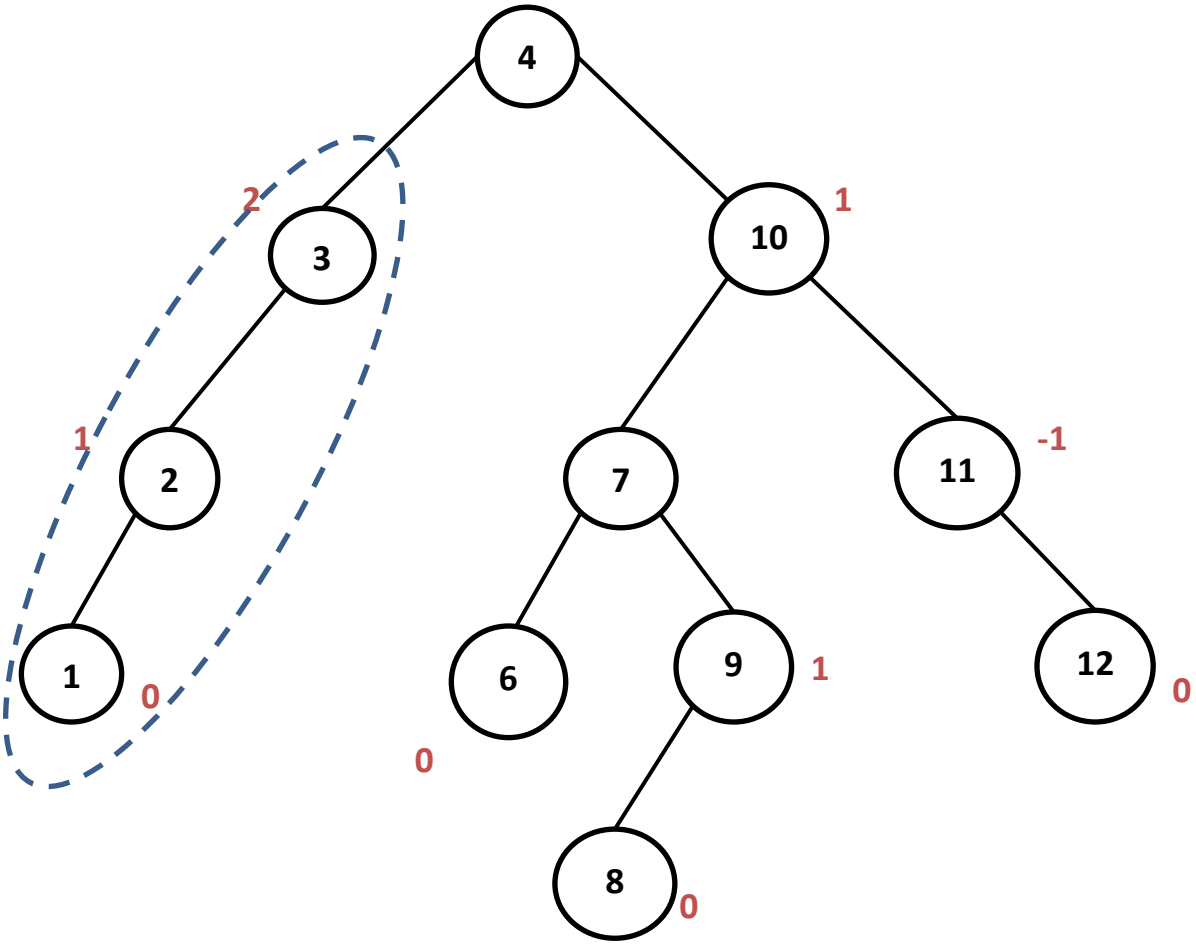
a. Draw the resulting BST after **5** is removed, but before any rebalancing takes place. Label each node in the resulting tree with its balance factor. Replace a node with both children using an appropriate value from the node's left child.

Solution:

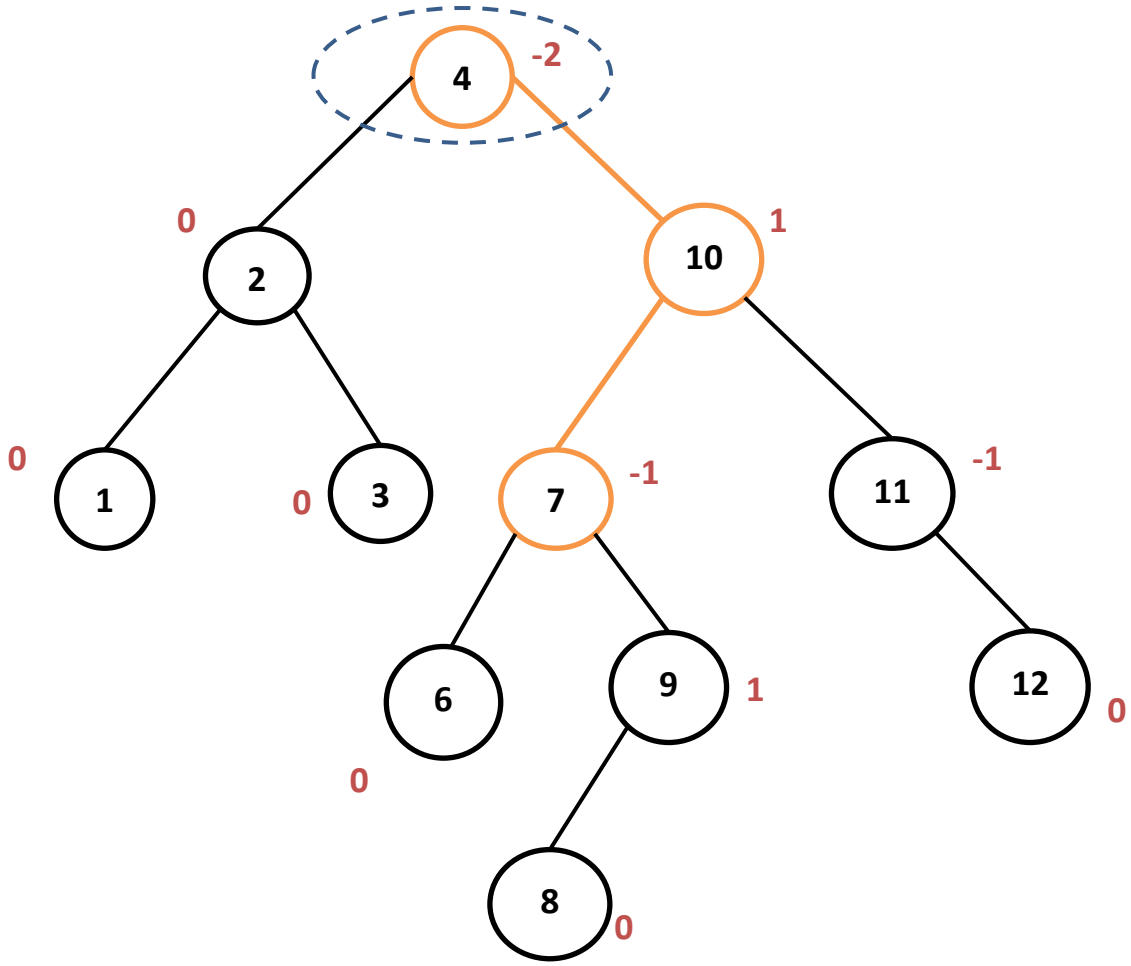


b. Now **rebalance the tree** that results from (a). Draw a new tree for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from an RL or LR rotation). You do not need to label these trees with balance factors.

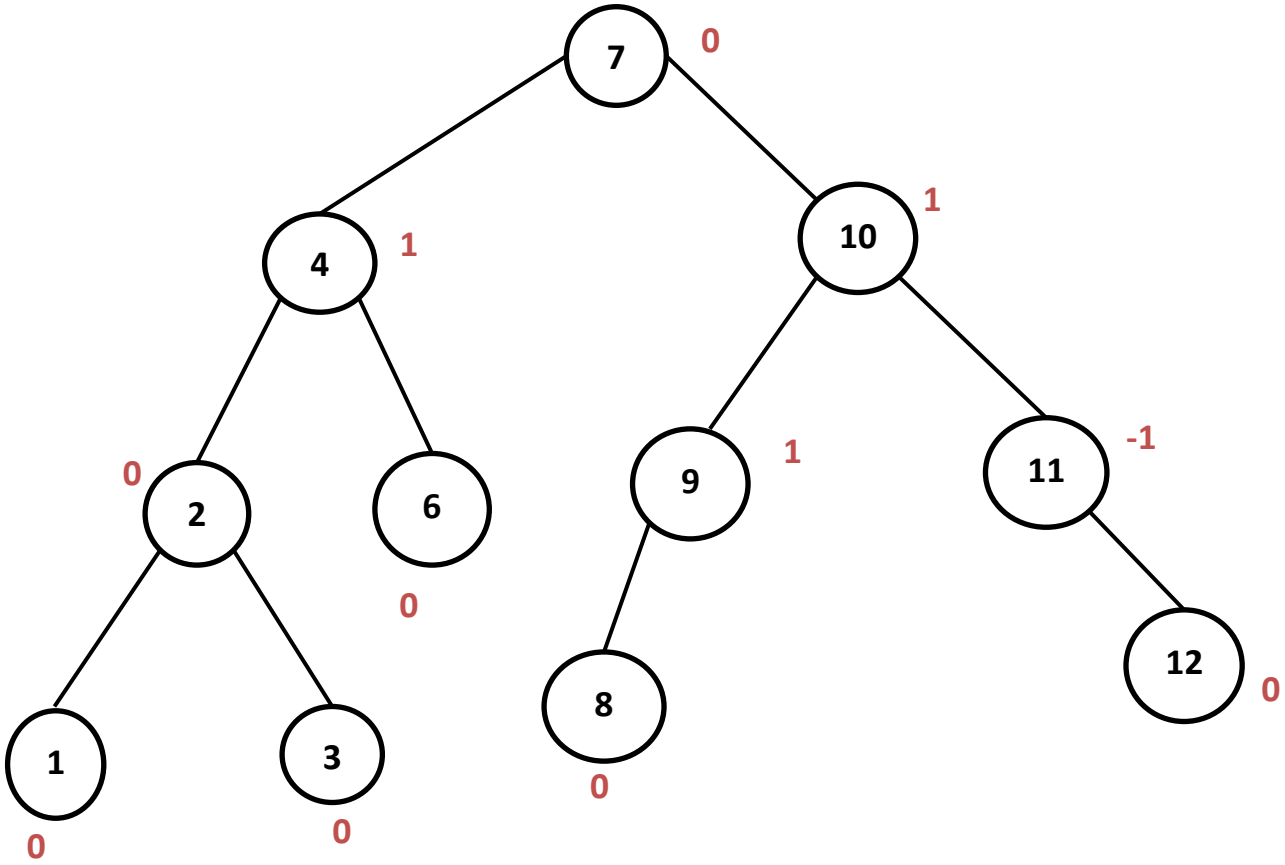
Solution:



An LL Rotation will be performed to generate the following AVL tree



Here we will look for the tallest grandchild of the root node. RL rotation will be performed resulting in the following balanced AVL Tree.


7
0
4
1
10
1
0
2
6
0
9
1
11
-1
1
0
3
0
8
0
12
0