Solution 1 :

```
#include <bits/stdc++.h>
using namespace std;

class sparse_matrix {

    public: int MAX = 100;

    static int data[][] = new int[MAX][3];

    static int row, col;

    static int len;

    sparse_matrix(int r, int c)
    {

        row = r;

        col = c;

        len = 0;
    }

    public: void insert(int r, int c, int val)
    {

        if (r > row || c > col) {
            cout << "Invalid entry" << endl;
        }

        else {

            data[len][0] = r;

            data[len][1] = c;

            data[len][2] = val;

            len++;
        }
    }

    void add(sparse_matrix b)
    {

        if (row != b.row || col != b.col) {
            cout << "Matrix dimensions not compatible for Addition" << endl;
        }

        else {

            int apos = 0, bpos = 0;
            sparse_matrix result = new sparse_matrix(row, col);

            while (apos < len && bpos < b.len) {
```

```
            if (data[apos][0] > b.data[bpos][0] ||
              (data[apos][0] == b.data[bpos][0] &&
               data[apos][1] > b.data[bpos][1]))

            {

                result.insert(b.data[bpos][0],
                              b.data[bpos][1],
                              b.data[bpos][2]);

                bpos++;
            }

            else if (data[apos][0] < b.data[bpos][0] ||
            (data[apos][0] == b.data[bpos][0] &&
              data[apos][1] < b.data[bpos][1]))

            {

                result.insert(data[apos][0],
                              data[apos][1],
                              data[apos][2]);

                apos++;
            }

            else {

                int addedval = data[apos][2] + b.data[bpos][2];

                if (addedval != 0)
                    result.insert(data[apos][0],
                                  data[apos][1],
                                  addedval);
                apos++;
                bpos++;
            }
        }

        while (apos < len)
            result.insert(data[apos][0],
                          data[apos][1],
                          data[apos++][2]);

        while (bpos < b.len)
            result.insert(b.data[bpos][0],
                          b.data[bpos][1],
                          b.data[bpos++][2]);

        result.print();
    }
}

sparse_matrix transpose()
{
```

```
    sparse_matrix result = new sparse_matrix(col, row);

    result.len = len;

    int count[] = new int[col + 1];
    for (int i = 1; i <= col; i++)
        count[i] = 0;

    for (int i = 0; i < len; i++)
        count[data[i][1]]++;

    int[] index = new int[col + 1];

    index[1] = 0;

    for (int i = 2; i <= col; i++)

        index[i] = index[i - 1] + count[i - 1];

    for (int i = 0; i < len; i++) {

        int rpos = index[data[i][1]]++;

        result.data[rpos][0] = data[i][1];

        result.data[rpos][1] = data[i][0];

        result.data[rpos][2] = data[i][2];
    }

    return result;
}

void multiply(sparse_matrix b)
{

    if (col != b.row) {

        cout << "Matrix dimensions not compatible for multiplication" << endl;

        return;
    }
    b = b.transpose();
    int apos, bpos;

    sparse_matrix result = new sparse_matrix(row, b.row);

    for (apos = 0; apos < len;) {
        for (bpos = 0; bpos < b.len;) {
            int c = b.data[bpos][0];
            int tempa = apos;
            int tempb = bpos;
            int sum = 0;
            while (tempa < len && data[tempa][0] == r
                   && tempb < b.len && b.data[tempb][0] == c) {

                if (data[tempa][1] < b.data[tempb][1])
```

```
                tempa++;

            else if (data[tempa][1] > b.data[tempb][1])

                tempb++;
            else

                sum += data[tempa++][2] * b.data[tempb++][2];
        }

        if (sum != 0)
            result.insert(r, c, sum);

        while (bpos < b.len && b.data[bpos][0] == c)

            bpos++;
    }

    while (apos < len && data[apos][0] == r)

        apos++;
  }

  result.print();
}

void print()
{
  cout << "Dimension: " + row + "x" + col << endl;
  cout << "Sparse Matrix: \nRow Column Value" << endl;

  for (int i = 0; i < len; i++) {

      cout << data[i][0] + " "
                + data[i][1] + " " + data[i][2] << endl;
  }
}

int main()
{

  sparse_matrix a = new sparse_matrix(4, 4);
  sparse_matrix b = new sparse_matrix(4, 4);

  a.insert(1, 2, 10);
  a.insert(1, 4, 12);
  a.insert(3, 3, 5);
  a.insert(4, 1, 15);
  a.insert(4, 2, 12);
  b.insert(1, 3, 8);
  b.insert(2, 4, 23);
  b.insert(3, 3, 9);
  b.insert(4, 1, 20);
  b.insert(4, 2, 25);

  cout << "Addition: ";
```

```
        a.add(b);
        cout << "\nMultiplication: ";
        a.multiply(b);
        cout << "\nTranspose: ";
        sparse_matrix a_transpose = a.transpose();
        a_transpose.print();
        return 0;
    }
}
```

Output:

```
Addition:
Dimension: 4x4
Sparse Matrix:
Row   Column    Value
1       2       10
1       3        8
1       4       12
2       4       23
3       3       14
4       1       35
4       2       37

Multiplication:
Dimension: 4x4
Sparse Matrix:
Row   Column    Value
1       1        240
1       2        300
1       4        230
3       3         45
4       3        120
4       4        276

Transpose :
Dimension : 4x4
Row   Column    Value
1       4       15
2       1       10
2       4       12
3       3        5
4       1       12
```

Solution 2:

```
#include <bits/stdc++.h>
using namespace std;

class DualStack
{
    int *arr;
    int size;
    int top1, top2;
        public:
        DualStack(int MaxSize)
        {
            size = MaxSize;
            arr = new int[MaxSize];
            top1 = -1;
            top2 = size;
        }

        bool IsFull()
        {
            if (top1 >= top2 - 1)
            {
```

```
		cout << "Stack Overflow\n";
		return 1;
	}
	return 0;
}

void push1(int x)
{
	if (!IsFull())
	{
		top1++;
		arr[top1] = x;
	}
	else
	{
		exit(1);
	}
}

void push2(int x)
{
	if (!IsFull())
	{
		top2--;
		arr[top2] = x;
	}
	else
	{
		exit(1);
	}
}

int pop1()
{
  if (top1 >= 0 )
  {
    int x = arr[top1];
    top1--;
    return x;
  }
  else
  {
     cout << "Stack UnderFlow";
     exit(1);
  }
}

int pop2()
{
  if (top2 < size)
  {
    int x = arr[top2];
    top2++;
    return x;
  }
  else
  {
```

```
                cout << "Stack UnderFlow";
                exit(1);
            }
        }
};

int main(){
        int MaxSize;
        cout << "Enter the size of array" << endl;
        cin >> MaxSize;
        DualStack ds(MaxSize);
        int choice, x;
        while(true)
        {
                cout << "Press 1 to push in stack 1 \nPress 2 to push in stack 2 \nPress 3 to pop in
stack 1 \nPress 4 to pop in stack 2 \nPress 5 to exit \n";
                cin >> choice;
                if (choice == 5)
                        break;
                switch(choice)
                {
                        case 1: cout << "Enter the element" << endl;
                        cin >> x;
                        ds.push1(x);
                        break;
                        case 2: cout << "Enter the element" << endl;
                        cin >> x;
                        ds.push2(x);
                        break;
                        case 3: ds.pop1();
                        break;
                        case 4: ds.pop2();
                        break;
                        default : cout << "Invalid input" << endl;
                }
        }
        return 0;
}
```

Output:

```
Enter the size of array
3
Press 1 to push in stack 1
Press 2 to push in stack 2
Press 3 to pop in stack 1
Press 4 to pop in stack 2
Press 5 to exit
1
Enter the element
4
Press 1 to push in stack 1
Press 2 to push in stack 2
Press 3 to pop in stack 1
Press 4 to pop in stack 2
Press 5 to exit
1
Enter the element
3
Press 1 to push in stack 1
Press 2 to push in stack 2
Press 3 to pop in stack 1
Press 4 to pop in stack 2
Press 5 to exit
2
Enter the element
2
Press 1 to push in stack 1
Press 2 to push in stack 2
Press 3 to pop in stack 1
Press 4 to pop in stack 2
Press 5 to exit
2
Enter the element
1
Stack Overflow
```

Solution 3:

```
#include <bits/stdc++.h>
using namespace std;

class Queue
{
   stack <int> s1, s2;

   public:
   void enQueue(int x)
   {
      s1.push(x);
   }

   int deQueue()
   {
      if (s1.empty() && s2.empty())
      {
         cout << "Queue is empty";
         exit(0);
      }
      if (s2.empty())
      {
         while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
         }
      }
      int x = s2.top();
      s2.pop();
      return x;
   }
};

int main()
{
   Queue q;
   while(true)
         {
                  cout << "Press 1 to enQueue \nPress 2 to deQueue 2 \nPress 3 to exit \n";
                  int choice, x;
                  cin >> choice;
                  if (choice == 3)
                           break;
                  switch(choice)
                  {
                           case 1: cout << "Enter the element" << endl;
                                          cin >> x;
                                          q.enQueue(x);
                                          break;
                           case 2: cout << q.deQueue() << endl;
                                          break;
                           default : cout << "Invalid input" << endl;
                  }
         }
   return 0;
```

}

Output:

```
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
1
Enter the element
5
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
1
Enter the element
6
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
1
Enter the element
7
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
2
5
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
2
6
Press 1 to enQueue
Press 2 to deQueue 2
Press 3 to exit
3
```

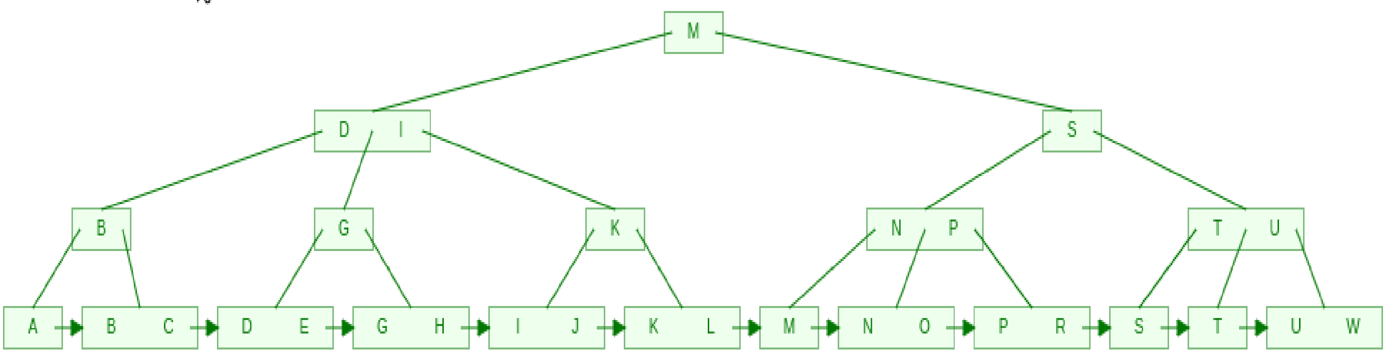
Solution 4:

We must also keep track of the number of items in the collection as if it exceeds the size, we'll not be able to store them.

Solution 5:

1. CREATE TWO STACKS : ASCENDING, DESCENDING
2. PUSH -INFINITY INTO ASCENDING
3. PUSH +INFINITY INTO DESCENDNG
4. IF THE CURRENT ELEMENT IS > THE ASCENDING TOP
5. CHECK IT WITH THE DESCENDING TOP
6. OTHERWISE, MOVE ELEMENTS FROM THE ASCENDING TO THE DESCENDING UNTIL THE CURRENT ELEMENT BECOMES > THE ASCENDING TOP
7. IF THE CURRENT ELEMENT IS < THE DESCENDING TOP
8. THEN THIS IS THE INSERTION POINT
9. OTHERWISE, MOVE ELEMENTS FROM THE DESCENDING TO THE ASCENDING UNTIL THE CURRENT ELEMENT BECOMES < THE DESCENDING TOP
10. PUSH THE CURRENT ELEMENT INTO THE ASCENDING STACK
11. PROCESS NEXT ELEMENT TO THE DESCENDING IF ANY
12. NOW MOVE ALL ELEMENTS OF ASCENDING TO DESCENDING STACK
13. HENCE WE HAVE ALL ELEMENTS IN ASCENDING ORDER

Solution 6:



Solution 7:

(a) Doubly Ended Queue, because it allows insertions and deletions at both ends and nowhere in between.

(b) Hash Map, because it offers constant access time for key value pairs and allows both insertions and deletions in costant time as well

(c) Stack, because if we find a dead end we can pop that node out and push a new node to go on a different path.

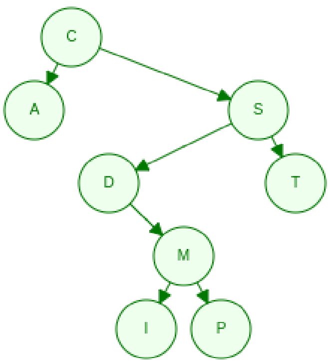
(d) Linked list, because merging is efficient in linked lists.

(e) Queue, because we are managing sort of a telephone waiting line with first come first serve basis.

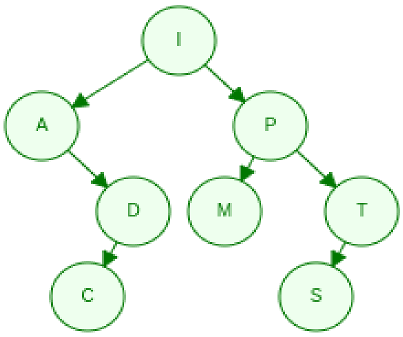
Solution 8:

In a binary search tree, order of insertion does matter as we can see from this example:

C, S, D, T, A, M, P, I



I, P, M, A, T, D, S, C



The input elements were the same but different trees were created. Hence Proved!

Solution 9:

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

class Hash
{
    int BUCKET;

    list <int> *table;
    public:
    Hash(int V);

    void insertItem(int x);

    int hashFunction(int i)
    {
        return ((2*i+5) % BUCKET);
    }

    void displayHash();
};

Hash::Hash(int b)
{
    this->BUCKET = b;
    table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
{
    int index = hashFunction(key);
    table[index].push_back(key);
}

void Hash::displayHash()
{
    for (int i = 0; i < BUCKET; i++)
    {
        cout << i;
        for (auto x : table[i])
            cout << " --> " << x;
        cout << endl;
    }
}

int main()
{
    int a[] = {12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5};
    int n = sizeof(a)/sizeof(a[0]);
    Hash h(11);
    for (int i = 0; i < n; i++)
    {
        h.insertItem(a[i]);
    }
    h.displayHash();
```

```
    return 0;
}
```

Output:

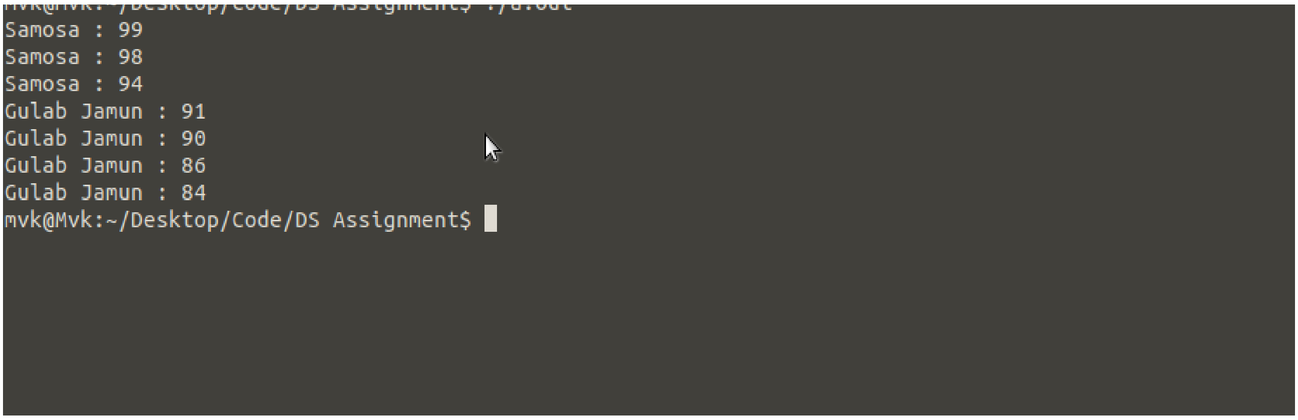
```
0
1 --> 20
2
3
4 --> 16 --> 5
5 --> 44 --> 88 --> 11
6 --> 94 --> 39
7 --> 12 --> 23
8
9 --> 13
10
mvk@Mvk:~/Desktop/Code/DS Assignment$
```

Solution 10:

```
#include<bits/stdc++.h>
using namespace std;

struct node
{
    int key;
    struct node *left, *right;
};

struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp -> key = item;
    temp -> left = temp -> right = NULL;
    return temp;
}

static int c = 0;

void inorder(struct node *root, int a, int b)
{
    if (root != NULL)
    {
        inorder(root -> right, a, b);
        if (c <= a)
            cout << "Samosa : ";
        else if (c > a && c <= a + b)
            cout << "Gulab Jamun : ";
        else
            return;
        cout << root -> key << endl;
        c++;
        inorder(root -> left, a, b);
```

```
    }
}

struct node* insert(struct node* node, int key)
{
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node -> left  = insert(node -> left, key);

    else if (key > node->key)
        node -> right = insert(node -> right, key);

    return node;
}

int main()
{
    int arr[] = {98, 26, 84, 72, 83, 94, 90, 78, 91, 99, 66, 82, 86, 55, 43, 60, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    struct node *root = NULL;
    root = insert(root, arr[0]);
    for(int i = 1; i < n; i++)
    {
        insert(root, arr[i]);
    }
    inorder(root, log(n), sqrt(n));

    return 0;
}
```

Output:



Solution 11:

```
#include <bits/stdc++.h>
using namespace std;
```

```
class ClearableStack
{
	stack <int> s;

public:
	void push(int x)
	{
		s.push(x);
	}
	void pop()
	{
		s.pop();
	}
	void clearstack()
	{
		while(!s.empty())
			s.pop();
	}
};

int main()
{
	ClearableStack cs;
	cs.push(1);
	cs.push(2);
	cs.push(3);
	cs.clearstack();
	return 0;
}
```

Solution 12:

```
Insert(Item, head, A, B)
	Node *ptr = head
	while (ptr -> next != A)				// Find location of A
		ptr = ptr -> next
	while (ptr -> next != B)
		if(Item <= ptr -> next -> info)			// Find the right position to insert Item
								// between  A and B
			ptr -> next -> prev = &Item
			Item -> next = ptr -> next -> next
			Item -> prev = ptr
			ptr -> next = &Item
			break
		ptr = ptr -> next
	ptr -> next -> prev = &Item				// if Item > B insert Item just after B
	Item -> next = ptr -> next -> next
	Item -> prev = ptr
	ptr -> next = &Item
```

Solution 13:

1) Declare a character stack S.
2) Now traverse the expression string.
3) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.
4) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Solution 14:

```
1.  declare static variable int_part = 0
2.  void Separate(float n)
3.  if n less than 1                              // Base case
4.          print "decimal part = " n
5.          print "integer part = " int_part
6.  else                                          // Recursive case
7.          int_part++
8.  call separate(n--)
```