STACK IMPLEMENTATION USING ARRAY

```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;
void push(int val) {
   if(top>=n-1)
   cout<<"Stack Overflow"<<endl;
   else {
      top++;
      stack[top]=val;
   }
}
void pop() {
   if(top<=-1)
   cout<<"Stack Underflow"<<endl;
   else {
      cout<<"The popped element is "<< stack[top] <<endl;
      top--;
   }
}
void display() {
   if(top>=0) {
      cout<<"Stack elements are:";
      for(int i=top; i>=0; i--)
      cout<<stack[i]<<" ";
      cout<<endl;
   } else
   cout<<"Stack is empty";
}
int main() {
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
```

```
  cout<<"4) Exit"<<endl;
  do {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
    switch(ch) {
      case 1: {
        cout<<"Enter value to be pushed:"<<endl;
        cin>>val;
        push(val);
        break;
      }
      case 2: {
        pop();
        break;
      }
      case 3: {
        display();
        break;
      }
      case 4: {
        cout<<"Exit"<<endl;
        break;
      }
      default: {
        cout<<"Invalid Choice"<<endl;
      }
    }
  }while(ch!=4);
  return 0;
}
```

QUEUE IMPLEMENTATION USING ARRAY :

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

```
int queue[100], n = 100, front = - 1, rear = - 1;
void Insert() {
   int val;
   if (rear == n - 1)
   cout<<"Queue Overflow"<<endl;
   else {
      if (front == - 1)
      front = 0;
      cout<<"Insert the element in queue : "<<endl;
      cin>>val;
      rear++;
      queue[rear] = val;
   }
}
void Delete() {
   if (front == - 1 || front > rear) {
      cout<<"Queue Underflow ";
      return ;
   } else {
      cout<<"Element deleted from queue is : "<< queue[front] <<endl;
      front++;;
   }
}
void Display() {
   if (front == - 1)
   cout<<"Queue is empty"<<endl;
   else {
      cout<<"Queue elements are : ";
      for (int i = front; i <= rear; i++)
      cout<<queue[i]<<" ";
         cout<<endl;
   }
}
int main() {
   int ch;
```

```
  cout<<"1) Insert element to queue"<<endl;
  cout<<"2) Delete element from queue"<<endl;
  cout<<"3) Display all the elements of queue"<<endl;
  cout<<"4) Exit"<<endl;
  do {
    cout<<"Enter your choice : "<<endl;
    cin>>ch;
    switch (ch) {
      case 1: Insert();
      break;
      case 2: Delete();
      break;
      case 3: Display();
      break;
      case 4: cout<<"Exit"<<endl;
      break;
      default: cout<<"Invalid choice"<<endl;
    }
  } while(ch!=4);
  return 0;
}
```

STACK USING LL

```
#include <iostream>
using namespace std;
struct Node {
  int data;
  struct Node *next;
};
struct Node* top = NULL;
void push(int val) {
  struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
```

```
    newnode->data = val;
    newnode->next = top;
    top = newnode;
}
void pop() {
    if(top==NULL)
    cout<<"Stack Underflow"<<endl;
    else {
        cout<<"The popped element is "<< top->data <<endl;
        top = top->next;
    }
}
void display() {
    struct Node* ptr;
    if(top==NULL)
    cout<<"stack is empty";
    else {
        ptr = top;
        cout<<"Stack elements are: ";
        while (ptr != NULL) {
            cout<< ptr->data <<" ";
            ptr = ptr->next;
        }
    }
    cout<<endl;
}
int main() {
```

```
int ch, val;
cout<<"1) Push in stack"<<endl;
cout<<"2) Pop from stack"<<endl;
cout<<"3) Display stack"<<endl;
cout<<"4) Exit"<<endl;
do {
  cout<<"Enter choice: "<<endl;
  cin>>ch;
  switch(ch) {
    case 1: {
      cout<<"Enter value to be pushed:"<<endl;
      cin>>val;
      push(val);
      break;
    }
    case 2: {
      pop();
      break;
    }
    case 3: {
      display();
      break;
    }
    case 4: {
      cout<<"Exit"<<endl;
      break;
    }
```

```
        default: {
            cout<<"Invalid Choice"<<endl;
        }
      }
  }while(ch!=4);
  return 0;
}
```

QUEUE USING LL :

```
#include <iostream>
using namespace std;
struct node {
  int data;
  struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert() {
  int val;
  cout<<"Insert the element in queue : "<<endl;
  cin>>val;
  if (rear == NULL) {
    rear = (struct node *)malloc(sizeof(struct node));
    rear->next = NULL;
    rear->data = val;
    front = rear;
  } else {
```

```
      temp=(struct node *)malloc(sizeof(struct node));
      rear->next = temp;
      temp->data = val;
      temp->next = NULL;
      rear = temp;
    }
}
void Delete() {
    temp = front;
    if (front == NULL) {
      cout<<"Underflow"<<endl;
      return;
    }
    else
    if (temp->next != NULL) {
      temp = temp->next;
      cout<<"Element deleted from queue is : "<<front->data<<endl;
      free(front);
      front = temp;
    } else {
      cout<<"Element deleted from queue is : "<<front->data<<endl;
      free(front);
      front = NULL;
      rear = NULL;
    }
}
void Display() {
```

```
    temp = front;
    if ((front == NULL) && (rear == NULL)) {
        cout<<"Queue is empty"<<endl;
        return;
    }
    cout<<"Queue elements are: ";
    while (temp != NULL) {
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
int main() {
    int ch;
    cout<<"1) Insert element to queue"<<endl;
    cout<<"2) Delete element from queue"<<endl;
    cout<<"3) Display all the elements of queue"<<endl;
    cout<<"4) Exit"<<endl;
    do {
        cout<<"Enter your choice : "<<endl;
        cin>>ch;
        switch (ch) {
            case 1: Insert();
            break;
            case 2: Delete();
            break;
            case 3: Display();
```

```
        break;
        case 4: cout<<"Exit"<<endl;
        break;
        default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
   return 0;
}
```

program to count all duplicates from string using hashing

```
# include <stdio.h>
# include <stdlib.h>
# define NO_OF_CHARS 256
void fillCharCounts(char *str, int *count)
{
int i;
for (i = 0; *(str+i); i++)
        count[*(str+i)]++;
}
void printDups(char *str)
{
int *count = (int *)calloc(NO_OF_CHARS, sizeof(int));
fillCharCounts(str, count);
int i;
for (i = 0; i < NO_OF_CHARS; i++)
        if(count[i] > 1)
                printf("%c, count = %d \n", i, count[i]);
```

```
free(count);
}
int main()
{
        char str[] = "test string";
        printDups(str);
        getchar();
        return 0;
}
```

Program to find words with ending letter “s”

```
#include
#include
using namespace std;
char str[100];
int main() {
int i, t, j, len;
scanf("%[^ \n]s", str);
len = strlen(str);
str[len] = ' ';
for (t = 0, i = 0; i < strlen(str); i++) {
if ((str[i] == ' ') && (str[i - 1] == 's')) {
for (j = t; j < i; j++) {
printf(“%c”, str[j]);
}
t = i + 1;
printf(" \n");
}
```

```
else
{
if (str[i] == ' ')
{
t = i + 1;
} } } }
```

Print all subsequences of string :

```
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;

// Find all subsequences
void printSubsequence(string input, string output)
{
        // Base Case
        // if the input is empty print the output string
        if (input.empty()) {
                cout << output << endl;
                return;
        }

        // output is passed with including
        // the Ist character of
        // Input string
        printSubsequence(input.substr(1), output + input[0]);

        // output is passed without
        // including the Ist character
        // of Input string
        printSubsequence(input.substr(1), output);
}

// Driver code
```

```
int main()
{
	// output is set to null before passing in as a
	// parameter
	string output = "";
	string input = "abcd";

	printSubsequence(input, output);

	return 0;
}
```

Balanced Parenthesis Program try taking input from user :

```
#include <stdio.h>
#include <stdlib.h>
#define bool int

// structure of a stack node
struct sNode {
	char data;
	struct sNode* next;
};

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data);

// Function to pop an item from stack
int pop(struct sNode** top_ref);

// Returns 1 if character1 and character2 are matching left
// and right Brackets
bool isMatchingPair(char character1, char character2)
{
	if (character1 == '(' && character2 == ')')
		return 1;
```

```
        else if (character1 == '{' && character2 == '}')
                return 1;
        else if (character1 == '[' && character2 == ']')
                return 1;
        else
                return 0;
}

// Return 1 if expression has balanced Brackets
bool areBracketsBalanced(char exp[])
{
        int i = 0;

        // Declare an empty character stack
        struct sNode* stack = NULL;

        // Traverse the given expression to check matching
        // brackets
        while (exp[i])
        {
                // If the exp[i] is a starting bracket then push
                // it
                if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
                        push(&stack, exp[i]);

                // If exp[i] is an ending bracket then pop from
                // stack and check if the popped bracket is a
                // matching pair*/
                if (exp[i] == '}' || exp[i] == ')'
                        || exp[i] == ']') {

                        // If we see an ending bracket without a pair
                        // then return false
                        if (stack == NULL)
                                return 0;
```

```
                        // Pop the top element from stack, if it is not
                        // a pair bracket of character then there is a
                        // mismatch.
                        // his happens for expressions like {(})
                        else if (!isMatchingPair(pop(&stack), exp[i]))
                                return 0;
                }
                i++;
        }

        // If there is something left in expression then there
        // is a starting bracket without a closing
        // bracket
        if (stack == NULL)
                return 1; // balanced
        else
                return 0; // not balanced
}

// Driver code
int main()
{
        char exp[100] = "{()}[]";

        // Function call
        if (areBracketsBalanced(exp))
                printf("Balanced \n");
        else
                printf("Not Balanced \n");
        return 0;
}

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data)
```

```
{
        // allocate node
        struct sNode* new_node
                = (struct sNode*)malloc(sizeof(struct sNode));

        if (new_node == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }

        // put in the data
        new_node->data = new_data;

        // link the old list off the new node
        new_node->next = (*top_ref);

        // move the head to point to the new node
        (*top_ref) = new_node;
}

// Function to pop an item from stack
int pop(struct sNode** top_ref)
{
        char res;
        struct sNode* top;

        // If stack is empty then error
        if (*top_ref == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }
        else {
                top = *top_ref;
```

```
            res = top->data;
            *top_ref = top->next;
            free(top);
            return res;
        }
}
```

Inserting elements in Tree :

```
// C++ program to insert element in Binary Tree
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */

struct Node {
        int data;
        Node* left;
        Node* right;
};

/* Function to create a new node */

Node* CreateNode(int data)
{
        Node* newNode = new Node();
        if (!newNode) {
                cout << "Memory error\n";
                return NULL;
```

```
        }
        newNode->data = data;
        newNode->left = newNode->right = NULL;
        return newNode;
}

/* Function to insert element in binary tree */

Node* InsertNode(Node* root, int data)
{
        // If the tree is empty, assign new node address to root
        if (root == NULL) {
                root = CreateNode(data);
                return root;
        }

        // Else, do level order traversal until we find an empty
        // place, i.e. either left child or right child of some
        // node is pointing to NULL.
        queue<Node*> q;
        q.push(root);

        while (!q.empty()) {
                Node* temp = q.front();
                q.pop();

                if (temp->left != NULL)
                        q.push(temp->left);
                else {
```

```
                temp->left = CreateNode(data);
                return root;
            }

            if (temp->right != NULL)
                q.push(temp->right);
            else {
                temp->right = CreateNode(data);
                return root;
            }
        }
}

/* Inorder traversal of a binary tree */

void inorder(Node* temp)
{
        if (temp == NULL)
            return;

        inorder(temp->left);
        cout << temp->data << ' ';
        inorder(temp->right);
}

// Driver code
int main()
{
        Node* root = CreateNode(10);
```

```
    root->left = CreateNode(11);
    root->left->left = CreateNode(7);
    root->right = CreateNode(9);
    root->right->left = CreateNode(15);
    root->right->right = CreateNode(8);

    cout << "Inorder traversal before insertion: ";
    inorder(root);
    cout << endl;

    int key = 12;
    root = InsertNode(root, key);

    cout << "Inorder traversal after insertion: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Delete an element from tree:

```
// C++ program to delete element in binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
child and a pointer to right child */
struct Node {
    int key;
    struct Node *left, *right;
```

```
};

/* function to create a new node of tree and
return pointer */
struct Node* newNode(int key)
{
        struct Node* temp = new Node;
        temp->key = key;
        temp->left = temp->right = NULL;
        return temp;
};

/* Inorder traversal of a binary tree*/
void inorder(struct Node* temp)
{
        if (!temp)
                return;
        inorder(temp->left);
        cout << temp->key << " ";
        inorder(temp->right);
}

/* function to delete the given deepest node
(d_node) in binary tree */
void deletDeepest(struct Node* root,
                                struct Node* d_node)
{
        queue<struct Node*> q;
        q.push(root);
```

```
// Do level order traversal until last node
struct Node* temp;
while (!q.empty()) {
        temp = q.front();
        q.pop();
        if (temp == d_node) {
                temp = NULL;
                delete (d_node);
                return;
        }
        if (temp->right) {
                if (temp->right == d_node) {
                        temp->right = NULL;
                        delete (d_node);
                        return;
                }
                else
                        q.push(temp->right);
        }

        if (temp->left) {
                if (temp->left == d_node) {
                        temp->left = NULL;
                        delete (d_node);
                        return;
                }
                else
                        q.push(temp->left);
```

```
            }
        }
}

/* function to delete element in binary tree */
Node* deletion(struct Node* root, int key)
{
        if (root == NULL)
                return NULL;

        if (root->left == NULL && root->right == NULL) {
                if (root->key == key)
                        return NULL;
                else
                        return root;
        }

        queue<struct Node*> q;
        q.push(root);

        struct Node* temp;
        struct Node* key_node = NULL;

        // Do level order traversal to find deepest
        // node(temp) and node to be deleted (key_node)
        while (!q.empty()) {
                temp = q.front();
                q.pop();
```

```
            if (temp->key == key)
                key_node = temp;

            if (temp->left)
                q.push(temp->left);

            if (temp->right)
                q.push(temp->right);
        }

        if (key_node != NULL) {
            int x = temp->key;
            deletDeepest(root, temp);
            key_node->key = x;
        }
        return root;
}

// Driver code
int main()
{
        struct Node* root = newNode(10);
        root->left = newNode(11);
        root->left->left = newNode(7);
        root->left->right = newNode(12);
        root->right = newNode(9);
        root->right->left = newNode(15);
        root->right->right = newNode(8);
```

```
    cout << "Inorder traversal before deletion : ";
    inorder(root);

    int key = 11;
    root = deletion(root, key);

    cout << endl;
    cout << "Inorder traversal after deletion : ";
    inorder(root);

    return 0;
}
```