**DSA LAB RECORD**

**B.HARITHA**

**AP19110010287**

**CSE-G**

**PROGRAM 22**

**Title:**

Sum of the natural numbers

**Objective:**

Write a C Program to find the sum of natural numbers using function.

**Explanation:**

Given n where n>=0, we have sum = 1+2+....+n

**Pseudo Code:**

>Sum N natural numbers

1.SET = n

2.sum =0  
3.for i=1 till n DO

{

4. Sum+ i

}

5. Return sum

6.END

**CODE:**

#include <stdio.h>

//function to add from 1 to n

int sum\_natural(int n){

int i,sum=0;

//loops runs from 1 to n

for(i=1;i<=n;i++)

sum += i;

return sum;

}

int main(int argc, char const \*argv[]){

int n;

//inputs n

printf("Enter some n\n");

scanf("%d",&n);

//prints sum

printf("Sum till of %d natural numbers is %d",n,sum\_natural(n));

return 0;

}

**OUTPUT:**

Enter some n

5

Sum till of 5 natural numbers is 15

**CONCLUSION:**

By using this code we can get the exact information that we expect . and the time complexity is O(n). And we can also it by using math formulas like sum\_n = n\*(n+1)/2.

**PROGRAM 23:**

**Title:**

Factorial using recursion

**Objective:**

Write a C Program to find factorial of number using recursion.

**Explanation:**

Given a number n, where n>=0, factorial of number n! = 1x2x3….(n-1)x(n)

**Pseudo Code:**

**>Factorial**

START

1. recursion(n)
2. {
3. If n== 0or n==1
4. Return 1
5. Return n\*recursion(n-1)
6. }
7. end

**code:**

#include <stdio.h>

int recurFact(int n){

//stop condition for recursive stack

if(n == 0|| n==1)

return 1;

return n\*recurFact(n-1);

}

int main(int argc, char const \*argv[])

{

int n;

printf("Enter n\n");

scanf("%d",&n);

printf("Factorial of %d is %d\n",n,recurFact(n));

return 0;

}

**OUTPUT:**

Enter n

5

Factorial of 5 is 120

**CONCLUSION:**

This code helps us to find the actual result of the given function, and its time complexity is O(n\*).

**PROGRAM 24**

**Title:**

Fibonacci series

**Objective:**

Write a C Program to generate the Fibonacci series.

**Explanation:**

This program is helpful For finding the nth fibonacci number, Fn = Fn-1 + Fn-2.

**Pseudo Code:**

>Fibonacci

1. a=0,b=1,c=0

2. For i between 0 to n:

3.{

4.c=a+b,a=b,b=c.

5. }

6.END

**CODE:**

#include <stdio.h>

//prints the fibonacci series till n numbers

void fibonnaciSeries(int n){

int i, a = 0, b=1,c;

for(i=0;i<n;i++){

c = a+b;

a = b;

b = c;

//prints nth fibonacci number

printf("%d\n",c);

}

}

int main(int argc, char const \*argv[])

{

int n;

//inputting n

printf("Enter a n\n");

scanf("%d",&n);

//calling function to print the first n fibonacci numbers.

fibonnaciSeries(n);

return 0;

}

OUTPUT:

**Enter a n**

**6**

**1**

**2**

**3**

**5**

**8**

**13**

**Conclusion:**

This code is very helpful in finding expected results. The time complexity is O(n).

PROGRAM 25

**Title:**

This question is to Implementation of structure

**Objective:**

Write a C Program using structure for entering details of the five students as

name, admission number, Date of birth, department and display all the details.

**Explanation:**

Inputting and outputs of structure having some objects. In c we access the structure objects by ‘ .’ for example if we have a structure student, which contains objects name, rollno, let stu be the variable initializing the structure then we can access the objects by, stu.name, stu.roll no. We do the same with any number of objects.

**Pseudo Code:**

>Entering details

1.set student

S[5]

2.for i=1 till 5 DO

3.{

4.input s

5.objects data

6.}

7. END

**Code:**

#include <stdio.h>

//structure for date

typedef struct Date{

int month;

int day;

int year;

}date;

//structure for student

typedef struct Student{

char name[30];

int adminNo;

date DOB;

char dept[30];

}student;

int main()

{

student s[5];

int i;

//inputtting the details of 5 students

printf("Enter details: \n");

for(i=0;i<5;i++){

printf("Enter\n");

printf("Name");

scanf("%s",s[i].name);

printf("Admin No: ");

scanf("%d",&s[i].adminNo);

printf("Date of Birth:\n");

printf("Date: ");

scanf("%d",&s[i].DOB.day);

printf("Month: ");

scanf("%d",&s[i].DOB.month);

printf("year: ");

scanf("%d",&s[i].DOB.year);

printf("Department: ");

scanf("%s",s[i].dept);

}

//printing details of five students

printf("\n Details: \n");

for(i=0;i<5;i++){

printf("Name: %s\n",s[i].name );

printf("Admin No: %d\n",s[i].adminNo );

printf("Date of Birth: %d/%d/%d\n",s[i].DOB.day,s[i].DOB.month,s[i].DOB.year);

printf("Department: %s\n\n",s[i].dept);

}

return 0;

}

**Outputs:**

sample input

Enter details:

Enter

Namehari

Admin No: 765

Date of Birth:

Date: 12

Month: 5

year: 2002

Department: interpreter

sample input

details:

Name : hari

Admin No: 765

Date of Birth: 12/5/2002

Department: interpreter

**Conclusion:**

This code is giving expected results. The time complexity of entering and displaying is constant. If instead of five students we need to enter and display n students, the time complexity would be O(n).

PROGRAM 26

**Title:**

This is helpful to find String length

**Objective:**

Write a C program to find length of string using pointers.

**Explanation:**

In C programs, strings end with a null character, represented by ‘\0’. THus to find the string length we can simply count the characters till ‘\0’ from 0

**Pseudo Code:**

>Sum N natural numbers

1.Read str[]

2. Set \*ptr = str,

Count =0

3. While \*ptr != ‘\0’

{

4.count ++ , ptr++

}

5.return count  
6.end

**Code**

#include <stdio.h>

//calculates the length of the string and return the length

int string\_ln(char\* p){

int count = 0;

//runs the loop till the end of the list

while(\*p != '\0'){

count++;

p++;

}

return count;

}

int main(int argc, char const \*argv[])

{

char str[40];

//inputting string

printf("Enter a string:\n");

gets(str);

//printing the length

printf("The length of the entered string is %d\n",string\_ln(str));

return 0;

}

OUTPUT

Enter a string:

stay home stay safe

The length of the entered string is 19

**Conclusion:**

There are 19 characters in “C programming”, therefore it gives expected results. The time complexity is O(n). WHere n is the length of the string.

PROGRAM 27

**Title:** String copy

**Objective:**

Write a C program to copy one string to another using pointers.

**Explanation:**

To copy a string to another we can just initialize every character of the string to the other, till the end of the string which is represented by ‘\0’.

**Pseudo Code:**

>Sum N natural numbers

1.set \*dest,

\*src

2.While \*src != ‘\0’

3. \*dest++ = &src++

4.\*dest = ‘\0’

5.END

**CODE:**

#include <stdio.h>

void copystr(char \*dest,char \*src)

{

while(\*src!='\0')

\*dest++=\*src++;

\*dest='\0';

}

int main(int argc, char const \*argv[])

{

char str1[30],str2[30];

//inputting string

printf("Enter a string:\n");

gets(str1);

copystr(str2,str1);

printf("String 1:\n");

printf("%s\n",str1 );

printf("string 2:\n");

printf("%s\n",str2 );

return 0;

}

OUTPUT:

Enter a string:

stay home stay safe

String 1:

stay home stay safe

string 2:

stay home stay safe

**Conclusion:**

The function copies the string to another string, and the only exception is that of space, if desitated string is smaller than the source string then the whole string does not get copied.The time complexity is O(n).

**PROGRAM 28**

**Title:** String compare

**Objective:**

Write a C program to compare two strings using pointers.

**Explanation:**

For comparing strings we traverse through the string simultaneously if we both string reach ‘\0’ at same time it would mean they are of equal length. If one reached earlier than the other it would mean they are of unequal lengths.

**Pseudo Code:**

>Sum N natural numbers

1.Read str1, str2

2.while ( \*str1 == \*str)

3.{

4. If str1 or str2 reached end of string the break

5.}

6. If str 1 ==str 2 == ‘/0’ return 1

7. Else return 0

7. end.

**Code:**

#include <stdio.h>

int strcomp(char \*str1,char \*str2)

{

while(\*str1 == \*str2){

if ( \*str1 == '\0' || \*str2 == '\0' )

break;

str1++;

str2++;

}

if( \*str1 == '\0' && \*str2 == '\0' )

return 1;

return 0;

}

int main(int argc, char const \*argv[])

{

char str1[30],str2[30];

//inputting string

printf("Enter a string:\n");

gets(str1);

printf("Enter a string:\n");

gets(str2);

if(strcomp(str1,str2) == 1) printf("Stringers are equal\n");

else printf("strings are not equal\n");

return 0;

}

OUTPUT:

Enter a string:

stay home

Enter a string:

stay safe

strings are not equal

**Conclusion:**

Since the stay safe is of 8 characters and the stay home is of 8 characters, it is not equal. The time complexity is O(n), where n is the length of the shorter string.

**PROGRAM 29**

**Title:**

Reversing the string

**Objective:**

Write a C program to find the reverse of a string recursively and non-recursively.

**Explanation:**

We can reverse string by swapping the elements of the first half with the last half.

**Pseudo Code:**

>Recursively

/\* assuming that n is the length of the string\*/

1.set beg=0 , end = n-1

2.recrev (str,beg,end):

{

3.if beg >=eng return

4. swap(str+beg, str, end)

5. Recrev ( str, beg ++, end --)

6.}

7.end

>Non-Recursively

/\*assuming that n is the length of the string and str is the string to be reversed\*/

1.set beg=0 ,end = n-1

2.for i form 0 to n/2 {

3.swap (str+beg , str +end)

4.beg ++ ,end--

5.}

6.END.

**CODE**

#include <stdio.h>

void recRev(char \*x, int beg, int end)

{

char temp;

if (beg >= end)

return;

temp = \*(x+beg);

\*(x+beg) = \*(x+end);

\*(x+end) = temp;

recRev(x, ++beg, --end);

}

int string\_ln(char\* p){

int count = 0;

//runs the loop till the end of the list

while(\*p != '\0'){

count++;

p++;

}

return count;

}

void rev(char \*str)

{

int len, i;

char \*beg, \*end, temp;

len = string\_ln(str);

beg = str;

end = str;

for (i = 0;i < (len - 1 ) ; i++ )

end++;

for ( i = 0 ; i < len/2 ; i++ )

{

temp = \*end;

\*end = \*beg;

\*beg = temp;

beg++;

end--;

}

}

int main(int argc, char const \*argv[])

{

char str1[30];

//inputting string

printf("Enter a string:\n");

gets(str1);

printf("reversing string using recursion\n");

recRev(str1,0,string\_ln(str1)-1);

printf("%s\n",str1);

printf("Reversing the reversed string using iteration\n");

rev(str1);

printf("%s\n",str1);

return 0;

}

Enter a string:

haritha

reversing string using recursion

ahtirah

Reversing the reversed string using iteration

haritha

**Conclusion:**

Since the inputted string is “haritha” the reversed string found is “hraitha”, and the reverse of the is again “haritha”. Thus both functions are reversing the string and giving expected results. The time complexity for reversing string is O(n). BUt since we don't know the length of the string we need to find the length of the string explicitly which adds another O(n) to complexity.

**PROBLEM 30**

**Title:**

Binary search traversal.

**Objective:**

Create a binary tree and output the data with 3 tree traversals

**Explanation:**

A binary tree is a tree in which no node can have more than two children.

Pre-order: The traversal goes in format root, left,right.

In-order: The traversal goes in format left, root, right.

Post-order: The traversal goes in format left, right, root.

**Pseudo Code:**

>PreOrder

1.preorder(root):

2.if root == NULL return

3. Print root -> data

4. preorder(root -> left)

5. Preorder (root ->right)

6. end

>InOrder

1.Inorder(root):

2. If root == NULL return

3. inorder(root ->left)

4.print root ->data

5.inorder (root ->right)

6. End

>PostOrder

1.postorder(root):

2.if root == NULL return

3. postorder(root->left)

4.postorder(root ->right)

5.print root->data

6.end

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define initmemory() (struct node\*)malloc(sizeof(struct node))

struct node {

int data;

struct node \*left;

struct node \*right;

};

struct node\* insert(){

struct node \*newnode;

int x;

printf("Enter data:");

scanf("%d",&x);

if(x==-1)

return NULL;

newnode = initmemory();

newnode->data = x;

printf("left child of %d:\n",x);

newnode->left = insert();

printf("right child of %d:\n",x);

newnode->right = insert();

return newnode;

}

void postOrder(struct node \*root) {

if (root == NULL){

return;

}

postOrder(root->left);

postOrder(root->right);

printf("%d ",root->data);

}

void inOrder(struct node \*root) {

if(root == NULL) return;

inOrder(root->left);

printf("%d ",root->data);

inOrder(root->right);

}

void preOrder( struct node \*root) {

if(root == NULL)return;

printf("%d ",root->data);

preOrder(root->left);

preOrder(root->right);

}

int main() {

struct node\* root = insert();

int num,i;

int data;

printf("\nPost Order:\n");

postOrder(root);

printf("\nPre Order\n");

preOrder(root);

printf("\nIn Order\n");

inOrder(root);

return 0;

}

OUT PUT

Enter data:31

left child of 31:

Enter data:33

Left child of 33

Enter data:-3

Right child of 33

Enter data:34

left child of 34

Enter data:-3

Right child of 34

Enter data:-3

Right child 31

Enter data:35

Post order :34 33 -4 33 35 31

Preorder :31 33 34 35 -33 -4

Inorde :33 34 31 -4 33 35

**Conclusion:**

This program helps us to learn how to write code for preorder, postorder,inorder . All the traversals are O(n) worst, best, average cases.

**PROGRAM 31**

**Title:**

Binary Search Tree

**Objective:**

Create a Binary Search Tree(BST) and search for a given value in BST.

**Explanation:**

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).

**Pseudo Code:**

>Sum N natural numbers

1.search(N,T):

2.false if T is empty

3.true if T=N

4.search(N,T.left) if N<T

5.search(N,T.right) if N>T

6.END

**Code:**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#define initmemory() (struct node\*)malloc(sizeof(struct node))

typedef struct node {

int data;

struct node \*left;

struct node \*right;

}node;

node\* insert(node\* root, int data) {

if(root == NULL) {

node\* node = initmemory();

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

} else {

if(data <= root->data) {

root->left = insert(root->left, data);

}

else {

root->right = insert(root->right, data);;

}

return root;

}

}

int bstSearch(node\* root, int search)

{

if (root == NULL)

return 0;

if(root->data == search)

return 1;

if (root->data < search)

return 2\*(bstSearch(root->right, search));

return 2\*(bstSearch(root->left, search));

}

int main(int argc, char const \*argv[])

{

node\* root = NULL;

int num,i,search,data,pos;

printf("enter initial tree size\n");

scanf("%d", &num);

printf("Enter the elements in tree\n");

for(i=0;i<num;i++){

scanf("%d", &data);

root = insert(root, data);

}

printf("\nEnter search element\n");

scanf("%d",&search);

pos = bstSearch(root,search);

printf("found at depth %f\n",log2(pos));

return 0;

}

OUTPUT:

**enter initial tree size**

**6**

**Enter the elements in tree**

**8**

**5**

**4**

**3**

**2**

**1**

**Enter search element**

**5**

**found at depth 1.000000**

**CONCLUSION:**

This code helps us to know how to write binary search tree.Time complexity for searching is O(log n). This search algorithm won't be efficient to find the exact location in the tree.

**PROGRAM 33**

**Title:**

Shortest paths in graphs

**Objective:**

Write a program to find All-to-all Shortest paths in a Graph.

**Explanation:**

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

**Pseudo Code:**

>BFS

1.BFS (G, s) //Where G is the graph and s is the source node

2. let Q be queued.

3.Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

5. mark s as visited.

6.while ( Q is not empty)

7.//Removing that vertex from queue,whose neighbour will be visited now

8.v = Q.dequeue( )

10.//processing all the neighbours of v

11.for all neighbours w of v in Graph G

12.if w is not visited

13. Q.enqueue( w ) //Stores w in Q to further visit its neighbour

14. Mark was visited.15.

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

// Creating a node

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;

}

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

Queue contains

0 Resetting queue Visited 0

Queue contains

2 1 Visited 2

Queue contains

1 4 Visited 1

Queue contains

4 3 Visited 4

Queue contains

3 Resetting queue Visited 3

**Conclusion:**

This code helps to find All-to-all Shortest paths in a Graph.

Time complexity for BFS is O(n+e).

**PROGRAM 34**

**Title**:

Implementing Stacks using Arrays

**Objective:**

Write a C program to implement the STACK operation using array as a data structure. Users must be given the following choices to perform relevant tasks.

1. Push an element on to the STACK.

2. Pop and element from the STACK.

3. Peek the STACK.

4. Display the STACK.

5. Exit the program.

**Explanation:**

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek: Returns top element of stack

**Pseudo Code:**

**>Push**

/\*Assuming that array-stack can hold maximum N elements\*/

/\*initial stack index; top = -1\*/

1.Read item

2.if(top == N-1)

3.print(“overflow”)

4.else

5.top++

6.stack[top] = item

7.END

**>Pop**

/\*Assuming that array-stack can hold maximum N elements\*/

1.if(top == -1)

2.print(“underflow”)

3.else{

4.top--

5.return stack[top+1]

}

6.END

>Display

1.while(top != -1){

2.print stack[top]

3.top--

}

4.END

**Code**:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define max 1000//max elements in stack  //defining stack  typedef struct STACK{  int ar[max];  int top;  }stack;  //function to push elements into stack  void push(stack \*s, int data){  //overflow condition  if(s->top >= max-1){  printf("overflow\n");  return;  }  s->top++;  s->ar[s->top] = data;  }  //function to pop elements  int pop(stack \*s){  //underflow condition  if(s->top < 0) {  printf("Underflow\n");  return INT\_MIN;  //raise: if element stored == INT\_MIN , the function fails  }  int temp = s->ar[s->top];  s->top--;  return temp;  }  //return the top element without disturbing the top  int peek(stack s){  return s.ar[s.top];  }  //displays elements from top  void display(stack s){  int i;  if(s.top == -1){  printf("Empty\n");  }  for(i =s.top;i>-1;i--){  printf("%d\n",s.ar[i]);  }  printf("\n");  }  //main  int main(int argc, char const \*argv[])  { //initialized variables needed  stack s;  s.top = -1;  int choice,data;  //runs loop till user chooses exit --> 5  while(1){  //menu  printf("\n1. Push an element on to the STACK.\n"  "2. Pop and element from the STACK.\n"  "3. Peek the STACK.\n"  "4. Display the STACK.\n"  "5. Exit the program.\n");  scanf("%d",&choice);  //performs action according the choice  switch(choice){  case 1:{  printf("\nEnter an element to add\n");  scanf("%d",&data);  push(&s,data);  break;  }case 2:{  data =pop(&s);  if(data != INT\_MIN)  printf("%d is removed\n",data);  break;  }case 3:{  printf("%d is the top\n",peek(s) );  break;  }case 4:{  display(s);  break;  }case 5:{  exit(0);  break;  }default: printf("no such option choose again\n");  }  }  return 0;  } |

**OUTPUT**

**Push 2 elements**

1. Push an element on to the STACK.

2. Pop and element from the STACK.

3. Peek the STACK.

4. Display the STACK.

5. Exit the program.

1

Enter an element to add

44

1. Push an element on to the STACK.

2. Pop and element from the STACK.

3. Peek the STACK.

4. Display the STACK.

5. Exit the program.

1

Enter an element to add

56

Peak element

1. Push an element on to the STACK.

2. Pop and element from the STACK.

3. Peek the STACK.

4. Display the STACK.

5. Exit the program.

3

56 is the top

display

1. Push an element on to the STACK.

2. Pop and element from the STACK.

3. Peek the STACK.

4. Display the STACK.

5. Exit the program.

4

56

44

|  |  |
| --- | --- |
| **Pop and Display:**  1. Push an element on to the STACK.  2. Pop and element from the STACK.  3. Peek the STACK.  4. Display the STACK.  5. Exit the program.  2    56 is removed  1. Push an element on to the STACK.  2. Pop and element from the STACK.  3. Peek the STACK.  4. Display the STACK.  5. Exit the program.  4  44 | **Exiting program**  1. Push an element on to the STACK.  2. Pop and element from the STACK.  3. Peek the STACK.  4. Display the STACK.  5. Exit the program.  5 |

**Conclusion:**

The output of the code depends upon the instruction we gives

The time complexity for push, pop , peek is O(1) for best, worst and average cases.

And for Display it is O(n).

**PROGRAM 35:**

**Title:**

Reversing a string using Stacks

**Objective:**

**Write a C program to reverse a string using STACK.**

**Explanation:**

**When give with a string like “C Programming” it needs to be converted into “gninmmargorp C”. We can achieve this by using STACK data structure since it follows FILO(First In Last Out).**

**Pseudo Code:**

**>Reversing**

**/\*Assuming that string is less than equal to the max size of the stack\*/**

**1.READ string, INITIALIZE stack, i = 0**

**2 .FOR i < string.size()**

**push(string[i])/\*pushes into stack\*/**

**3.WHILE (a = pop() != ‘0’)**

**string[i] = a**

**4.PRINT string**

**5.END**

**Note:**

pushing and popping pseudo codes are already explained above.

**Code**:

#include <stdio.h>

#define max 1000

//globally initialized stack

char stack[max]; int top = -1;

//function which return the top element of the global stack

char pop(){

//underflow condition

if(top == -1)

return '0';

char res = stack[top];

top--;

return res;

}

//function pushes <char> data into the global stack

void push(char data){

//overflow condition

if(top == max-1)

printf("overflow\n");

top++;

stack[top] = data;

}

//main

int main(){

char string[max];

//inputting the string

scanf("%[^\n]%\*c",string);

int i ;

//pushes all the elements in the string into stack

for(i = 0;string[i] !='\0';i++){

push(string[i]);

}

//pops all elements from stack a d initializes to string

for(i=0;i<max;i++){

int c = pop();

if(c == '0')

break;

string[i] = c;

}

//prints the string after reversing

printf("%s",string);

return 0;

}

OUTPUT:

**HELLO WORLD**

**DLROW OLLEH**

CONCLUSION:

It prints the output as expected. Exception when the string contains ‘0’ since it was given as the “underflow” condition in pop() function. This can be overcomed by using a linked list.

It takes O(n) for pushing all elements into the stack and O(n) to pop all the elements. Therefore time complexity is O(2n), where n is the number of characters in the string.

**PROGRAM 36:**

**Title:**

Conversion of In-Fix to Postfix

**Objective:**

Write a C program to convert the given infix expression to postfix expression using STACK.

**Explanation:**

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

**Pseudo Code:**

>Postfix conversion

//postfix conversion

1.Scan x from left to right and REPEAT steps 3to 0 for each element of X 2.UNTIL the STACK is empty:

3.IF an operand is encountered ADD it to EXP

4.IF ‘(‘ is encountered, PUSH in to STACK

5.IF an operator is encountered, THEN{

6.repeatedly POP form STACK and add it to EXP,which has the same precedence or higher precedence than operator

7.PUSH operator to STACK

8.}

9.IF a ‘)’ is encountered, THEN

repeatedly POP from STACK and add it to EXP,till ‘(‘

/\*Don't add the ‘)’ into the expression\*/

10.}

11.END

**Code:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define max 1000

//initializing stack

typedef struct Stack{

int top;

int arr[max];

}stack;

//pops a element

char pop(stack \*s){

if (s->top != -1)

return s->arr[s->top--] ;

return '#';

}

//push a element

void push(stack \*s, char op){

s->arr[++s->top] = op;

}

//preccedence

int Prec(char ch){

switch (ch) {

case '+': return 1;

case '-': return 1;

case '\*': return 2;

case '/': return 2;

case '^': return 3;

}

return -1;

}

//convert infix two postfix

void in2post(char\* exp) {

int i, k;

stack s; s.top = -1;

for (i = 0, k = -1; exp[i]; ++i) {

//if is a variable

if ((exp[i] >= 'a' && exp[i] <= 'z') || (exp[i] >= 'A' && exp[i] <= 'Z'))

exp[++k] = exp[i];

//if a opening bracket

else if (exp[i] == '(')

push(&s, exp[i]);

//if a closing bracket

else if (exp[i] == ')'){

//pops all elements form the stack till '('

while (s.top != -1 && s.arr[s.top] != '(')

exp[++k] = pop(&s);

//if there is no '('

if (s.top == -1 && s.arr[s.top] != '(')

printf("Invalid expression\n");

//pops '('

else

pop(&s);

}

//if is a operation

else {

//pops till the precendence of stack is greater than current element

while (s.top != -1 && Prec(exp[i]) <= Prec(s.arr[s.top]))

exp[++k] = pop(&s);

//pushes the current operation into the stack

push(&s, exp[i]);

}

}

exp[++k] = pop(&s);

exp[++k] = '\0';

printf( "%s", exp );

}

//main

int main() {

//driver code

char exp[] = "a+b\*(c^d)+(e-f/g)\*c+d";

printf( "Infix- expression: %s\n", exp );

printf("%s","postfix expression:" );

in2post(exp);

return 0;

}

**OUTPUT**

Infix- expression: a+b\*(c^d)+(e-f/g)\*c+d

postfix expression:abcd^\*+efg/-c\*+d+

**Conclusion:**

This code helps to get the output that we excpected. The time complexity of the program is O(n^2) in best, average, worst cases. Not very reliable for large expressions. Space complexity is O(n).

**PROGRAM 37:**

**Title:**  Conversion of In-Fix to Pre-Fix

**Objective:**

Write a C program to convert the given infix expression to pre-fix expression using STACK.

**Explanation:**

When the operators are before operands then it is a prefix expression. We can convert infix to prefix by reversing the prefix expression and running through the postfix function.

**Pseudo Code:**

>Pre fix conversion

1.READ string

2.REVERSE string

3./\*NOTE while reversing ‘(‘ becomes ‘)’ and ‘)‘ becomes ‘(‘\*/

4.expression = REVERSE(POSTFIX(string))

5.END

**Code**:

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define max 1000

//intializing stack

typedef struct Stack{

int top;

char arr[max];

}stack;

//pops a element

char pop(stack \*s){

if (s->top != -1)

return s->arr[s->top--] ;

return '#';

}

//push a element

void push(stack \*s, char op){

s->arr[++s->top] = op;

}

//precedence

int Prec(char ch){

switch (ch) {

case '+': return 1;

case '-': return 1;

case '\*': return 2;

case '/': return 2;

case '^': return 3;

}

return -1;

}

//displays elements from top

void display(stack s){

int i;

if(s.top == -1){

printf("Empty\n");

}

for(i =s.top;i>-1;i--){

printf("%c",s.arr[i]);

}

printf("\n");

}

//convert infix two prefix

void in2pre(char\* exp) {

int i, n;

for(n = 0;exp[n];n++);

stack s; s.top = -1;

stack pre;pre.top = -1;

for (i = n-1; i>=0; i--) {

//if is a variable

if ((exp[i] >= 'a' && exp[i] <= 'z') || (exp[i] >= 'A' && exp[i] <= 'Z'))

push(&pre,exp[i]);

//if a opening bracket(reverse in prefix) therefore '('' = ')'

else if (exp[i] == ')')

push(&s, exp[i]);

//if a closing bracket

else if (exp[i] == '('){

//pops all elements form the stack till '('

while (s.top != -1 && s.arr[s.top] != ')')

push(&pre,pop(&s));

//if there is no '('

if (s.top == -1 && s.arr[s.top] != ')')

printf("Invalid expression\n");

//pops '('

else

pop(&s);

}

//if is a operation

else {

//pops till the precendence of stack is greater than current element

while (s.top != -1 && Prec(exp[i]) <= Prec(s.arr[s.top]))

push(&pre,pop(&s));

//pushes the current operation into the stack

push(&s, exp[i]);

}

}

push(&pre,pop(&s));

display(pre);

}

//main

int main() {

//driver code

char exp[] = "(a-b/c)\*(a/k-l)";

printf( "Infix- expression: %s\n", exp );

printf("%s","prefix expression:" );

in2pre(exp);

return 0;

}

**Output:**

**Infix- expression: (a-b/c)\*(a/k-l)**

**prefix expression:\*-a/bc-/akl**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |

**Conclusion:**

This code helps us to get the desired output. The time complexity of the program is O(n^2) in best, average, worst cases. Not very reliable for large expressions. Space complexity is O(n).

PROGRAM 38:

**Title: Evaluation of Post-fix and Pre-fix expressions.**

**Objective:**

Write a C program to evaluate the given pre-fix expression, post-fix expression.

**Explanation:**

Post-fix evaluation: While reading the expression from left to right, push the element in the stack if it is an operand.Pop the two operands from the stack, if the element is an operator and then evaluate it.Push back the result of the evaluation. Repeat it till the end of the expression.

Pre-fix evaluation: We do the same thing as post-fix evalution but we read the expression from right to left.

**Pseudo Code:**

>Post Fix evaluation

/\*Reading of expression takes place from left to right\*/

1. Repeat steps 2 to 8 till the end of expression:

{

2. READ the next element

3. IF element is operand THEN

4. PUSH the element in the STACK

5. IF element id operator THEN

{

6. POP the element into the STACK

7. Evaluate the expression formed by the two operands and the operator

8. PUSH the result into STACK

}

}

9. END

**Node:** we do the same thing prefix but read expression from right to left

**Code:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define max 1000

//intializing stack

typedef struct Stack{

int top;

int arr[max];

}stack;

//pops a element

int pop(stack \*s){

if (s->top > -1)

return s->arr[s->top--] ;

return -1;

}

//push a element

void push(stack \*s, int op){

s->arr[++s->top] = op;

}

//displays elements from top

void display(stack s){

int i;

if(s.top == -1){

printf("Empty\n");

}

for(i =s.top;i>-1;i--){

printf("%d\n",s.arr[i]);

}

printf("\n");

}

//evaluation of post fix expression where number are having spaces after them

void evaluatePost(char \*exp){

int i, num = 0;

stack operand; operand.top = -1;

for(i= 0;exp[i];i ++){

//if is a number

if(exp[i]-'0' >= 0 && exp[i]-'0' <=9){

//updates the number

if(num == 0) num = exp[i]-'0';

else num = num\*10 + exp[i]-'0';

}

//if is a space

else if(exp[i] == ' '){

//pushes the element

push(&operand, num);

num = 0;

}

//if is a operand

else{

//pops two number from stack and performs operations

int op2 = pop(&operand);

int op1 = pop(&operand);

if(op2 == -1 || op1 == -1){

printf("invalid\n");

return;

}

//pushes the result into the stack

switch(exp[i]){

case '\*':{

push(&operand, (op1\*op2));

break;

}

case '/':{

push(&operand,(op1/op2));

break;

}

case '+':{

push(&operand,(op1+op2));

break;

}

case '-':{

push(&operand,(op1-op2));

break;

}

}

}

}

//if the stack is not empty then the ex[ression is invalid

if(operand.top != 0){ printf("invalid\n");

//return;

}

display(operand);

}

//evaluation of pre fix expression where number are having spaces before them

//is same as post fix expect that it evaluate the expression backward

void evaluatePre(char \*exp){

int n;

for(n=0;exp[n];n++);

int i, num = 0;

stack operand; operand.top = -1;

for(i = n-1;i>-1;i--){

if(exp[i]-'0' >= 0 && exp[i]-'0' <=9){

if(num == 0) num = exp[i]-'0';

else num = num\*10 + exp[i]-'0';

}

else if(exp[i] == ' '){

push(&operand, num);

num = 0;

}

else{

int op2 = pop(&operand);

int op1 = pop(&operand);

if(op2 == -1 || op1 == -1){

printf("invalid\n");

return;

}

switch(exp[i]){

case '\*':{

push(&operand, (op1\*op2));

break;

}

case '/':{

push(&operand,(op1/op2));

break;

}

case '+':{

push(&operand,(op1+op2));

break;

}

case '-':{

push(&operand,(op1-op2));

break;

}

}

}

}

if(operand.top != 0){ printf("invalid\n");

return;

}

display(operand);

}

int main(){

//expression for post should have space after every number and no space anywhere else

//example "34 45 \*23 +/" is valid

//"3 4 5 \* 5 / -" is invalid

char exp[] = "8 2 +3 5 +/";

evaluatePost(exp);

//expression for post should have space before every number and no space anywhere else

//example " 34 45\* 23+/" is valid

//" 3 4 5 \* 5 / - " is invalid

char exp2[] = "+ 8\* 2 4";

evaluatePre(exp2);

return 0;

}

OUTPUT

Expression = 8 2 + 3 5 +

1

Expression = +8 \* 2 48+ 8\* 2 4 2 4 8\* 2 4

16

**Conclusion:**

By using this code we can get the expected results. But the input is restricted and complex in this algorithmic design. Which can be improved. The time complexity is O(N) for both prefix and postfix evaluation.

**PROGRAM 39:**

**Title:**

Implementation of Linear Queue

**Objective:**

Write a C program to implement a Linear-Queue, user must choose the following options:

1. Add an element to the Queue – EnQueue.
2. Remove an element from the Queue – DeQueue.
3. Display the elements of the Queue.
4. Terminate the program.

**Explanation:**

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

**Pseudo Code:**

>EnQueue in Array Queue

1. IF rear == NULL THEN

{

2. rear = front = 0

3. QUEUE[0] = ITEM

}

4. ELSE IF rear = N - 1 THEN

5. PRINT “OverFLow”

6. ELSE

{

7. QUEUE[rear + 1] = ITEM

8. rear = rear + 1

}

9. END

**>DeQueue in Array Queue**

1. IF front == NULL THEN PRINT “Queue Empty”

2. ELSE

{

ITEM = QUEUE[front]

3. IF front == rear THEN front = rear = NULL

4. ELSE front = front+1

}

5. END

**>Display in Queue**

5. WHILE (front != rear){

6. PRINT QUEUE[front]

7. top++

}

8. PRINT QUEUE[rear]

9. END

**Code:**

#include <stdio.h>

#include <stdlib.h>

# define MAX\_SIZE 100

//adding a eleemt into the queue

void enqueue(int \*queue, int \*rear,int \*front){

//overflow condition

if(\*rear == MAX\_SIZE-1){

printf("Overflow\n Aborting...");

return;

}

//inserts a element in rear

int ele;

printf("Enter the element to insert");

scanf("%d",&ele);

if (\*rear == -1)

{

\*front = 0;

}

\*rear += 1;

\*(queue+\*rear) = ele;

}

//displays elements from front to back

void display(int \*queue, int rear, int front){

int i;

printf("now queue (front.....to .....back:)\n");

for(i=front;i<rear;i++){

printf("%d ",\*(queue+i));

}

printf("%d ",\*(queue+rear));

}

//deletes a element form front

void dequeue(int \*queue, int \*front,int \*rear){

//underflow condition

if((\*front)==-1){

printf("\nunderflow.. aborting");

return;

}

int temp = \*(queue+(\*front));

if(\*front== \*rear){\*front= -1; \*rear= -1;}

else{\*front = \*front + 1;}

printf("\n%d is deleted",temp);

}

int main(){

int ans, queue[MAX\_SIZE], front = -1, rear = -1;

//menu

while(1){

printf("\nMENU\n"

"\n1.Insert an element "

"\n2. delete an element "

"\n3.Display queue"

"\n4. Exit");

scanf("%d",&ans);

//does according to option choosed

switch(ans){

case 1: enqueue(queue, &rear, &front);break;

case 2: dequeue(queue,&front,&rear);break;

case 3: display(queue, rear, front);break;

case 4: exit(0);break;

}

}

return 0;

}

**OUTPUT:**

**MENU**

**1.Insert an element**

**2. delete an element**

**3.Display queue**

**4. Exit1**

**Enter the element to insert46**

**MENU**

**1.Insert an element**

**2. delete an element**

**3.Display queue**

**4. Exit1**

**Enter the element to insert48**

**dequeue and Display:**

**MENU**

**1.Insert an element**

**2. delete an element**

**3.Display queue**

**4. Exit3**

**now queue (front.....to .....back:)**

**46 48**

**Exiting program:**

**MENU**

**1.Insert an element**

**2. delete an element**

**3.Display queue**

**4. Exit4**

**...Program finished with exit code 0**

**Press ENTER to exit console.**

**Conclusion:**

The code runs as expected. The time complexity of deleting and insertion is constant and for displaying it is O(n).

PROGRAM 40:

**Title: Implementation of Circular Queue**

**Objective:**

Write a C program to implement a Circular-Queue, user must choose the following options:

1. Add an element to the Queue – EnQueue.
2. Remove an element from the Queue – DeQueue.
3. Display the elements of the Queue.
4. Terminate the program.

**Explanation:**

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

**Pseudo Code:**

>EnQueue in Circular Queue

/\*Assumption that the circular queue is stored in QU with size n\*/

1. IF (front == 0 AND rear == n-1) or (front == rear +1)THEN

PRINT “OverFLow”

ELSE

{

IF front == NULL THEN

SET front = 0, rear = 0

ELSE IF rear = n-1 THEN

SET rear = 0

ELSE

SET rear = (rear + 1)%n

}

SET QUEUE[rear] = ITEM

END

>DeQueue in Circular Queue

1. IF front == NULL THEN

PRINT “OverFLow”, RETURN.

ELSE

{

SET DITEM = QUEUE[front]

IF front == rear THEN front = rear = NULL

ELSE IF front = n-1 THE front = 0

ELSE front = (front + 1)%n

}

END.

**Code:**

Insert the elements:

**MENU:circular queue**

**1.Insert an element**

**2.delete an element**

**3.Display queue**

**4.Exit1**

**Enter the element to insert0 is rear**

**44**

**MENU:circular queue**

**1.Insert an element**

**2.delete an element**

**3.Display queue**

**4.Exit1**

**Enter the element to insert1 is rear**

**45**

**Deleting the element:**

**MENU:circular queue**

**1.Insert an element**

**2.delete an element**

**3.Display queue**

**4.Exit2**

**44 is deleted**

**Display**

**MENU:circular queue**

**1.Insert an element**

**2.delete an element**

**3.Display queue**

**4.Exit3**

**now queue (front.....to .....back:)**

**44 45**

**Exiting code:**

**MENU:circular queue**

**1.Insert an element**

**2.delete an element**

**3.Display queue**

**4.Exit4**

**Conclusion:**

The time complexity is the same as with a linear queue, which is constant for enqueue and dequeue and o(n) for displaying. But the queue space is utilized.

**PROGRAM 41**

**Title:**

Initializing and displaying a node of five elements

**Objective:**

Write a C program to create a single linked list with 5 nodes. (5 integers are taken from user input) and display the linked-list elements.

**Explanation:**

We need to create a linked list using a dynamic initialization of memory.

**Pseudo Code:**

>Create node function.

/\*Assume that it is a function where data is passed\*/

1.SET data

2.newnode = NEW node //allocates memory to node

3.node->data = data

4.node->next = NULL

5.RETURN node

6.END

**>Creating linked-list of 5 nodes**

1. SET ptr=head=NULL

2. FOR i from 0 to 5

/\*for creating n nodes: i from 0 to n\*/

{

3. READ data

4. IF head == NULL: head = ptr = CREATENODE(data)

5. ELSE

{

6. ptr->next = CREATENODE(data)

7. ptr = ptr->next

}

}

8. RETURN head

9. END

**>Displaying linked list**

/\*head is a instance of the actual head here\*/

1. WHILE head != NULL

2. {

3. PRINT head->data

4. head = head->next

5. }

6. END

**Code:**

#include <stdio.h>

#include <stdlib.h>

//declaration if a node

typedef struct Node{

int data;

struct Node \*next;

}node;

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

//function which creates a list of 5 nodes

node\* createList(){

int n=5,data;

node \*p, \*head = NULL;

//runs loop 5 times

while(n--){

printf("Enter a number\n");

scanf("%d",&data);

if(head == NULL){

//intializing newnode as head

head = createNode(data);

p = head;

}

else{

p->next = createNode(data);

p = p->next;

}

}

//return the list of 5nodes

return head;

}

//displays elemets in linked list till null

void display(node \*head){

while(head!=NULL){

printf("%d->",head->data);

head = head->next;

}

printf("NULL\n");

}

//main

int main()

{

//driver code

node \*head=createList();

display(head);

return 0;

}

OUTPUT:

**Enter a number**

**21**

**Enter a number**

**35**

**Enter a number**

**47**

**Enter a number**

**55**

**Enter a number**

**62**

**21 -> 35 -> 47 -> 55 ->62 -> NULL**

**Conclusion:**

This code helps us to get the expected outputs. The time complexity for this code is constant. But if instead of 5 nodes we are to create n nodes then the time complexity for both inputting and displaying would be O(n).

**PROGRAM 42**

**Title:**

Searching a element in linked list

**Objective:**

Write a C program to search an element in a singly-linked list.

**Explanation:**

When given with an integer we search for the element in the linked list and print the position (index+1) of the element. If not found should print -1 Here we will use linear search because list might not be sorted.

**Pseudo Code:**

**>Search**

/\*Search is the element to be searched\*/

1. SET pos = 1

2. WHILE head != NULL

{

3. IF head->data == search //if found

4. RETURN pos

5. head = head->next

6. pos++

}

7. RETURN -1 //element not in list

8. END

**Code**:

#include <stdio.h>

#include <stdlib.h>

//declaration if a node

typedef struct Node{

int data;

struct Node \*next;

}node;

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

//function which creates a list of n nodes

node\* createList(){

int n,data;

node \*p, \*head = NULL;

printf("\n How many elements to enter?");

scanf("%d", &n);

//runs loop 5 times

while(n--){

printf("Enter a number\n");

scanf("%d",&data);

if(head == NULL){

//intializing newnode as head

head = createNode(data);

p = head;

}

else{

p->next = createNode(data);

p = p->next;

}

}

//return the list of 5nodes

return head;

}

//return position of the search element

int search(node \*head,int search){

int pos=1;

while(head != NULL){

if(head->data == search) return pos;

head = head->next;

pos++;

}

//not found condition

return -1;

}

//main

int main(int argc, char const \*argv[])

{

//driver code

node \*head = createList();

int s;

printf("element to be searched\n");

scanf("%d",&s);

printf("found at %d\n",search(head,s));

return 0;

}

OUTPUT:

**How many elements to be entered:4**

**Enter a number**

**12**

**Enter a number**

**21**

**Enter a number**

**35**

**Enter a number**

**47**

**Element to be searched**

**35**

**Found at 3**

**Conclusion:**

This code will be applicable for all cases, and gives expected output. The time complexity of search is the same as that of a linear search, i.e O(n) for worst, best, average cases. The space complexity is constant.

**PROGRAM 43:**

**Title:**

Implementation of Linked Lists

**Objective:**

Write a C program to perform the following tasks:

1. Insert a node at the beginning of a singly-linked list.
2. Insert a node at end of a singly-linked list.
3. Insert a node at the middle of a singly-linked list.
4. Delete a node from the beginning of the singly-linked list.
5. Delete a node from the end of a singly-linked list.

**Explanation:**

**Insertion in beginning of a list:** When we add a new node the new node becomes the head of the list thus this function needs to return the head of the newnode

**Insertion at the end of a list:** This function needs to add a new node at a=end of node and if the list is empty return a new node.

**Insertion in middle of a list:** This function needs to add a node in n/2th position where n is the total number of nodes in the linked-list.

**Deletion in beginning of a list:** This function needs to delete the free memory of the first node and return the next node of the head. If empty should return empty.

**Deletion at the end of a list:** This function needs to delete a node at end if only one node is present and return NULL.

**Pseudo Code:**

>Insertion in the beginning of a list

/\*This algorithm deals with insertion in the beginning of the linked list\*/

/\*Initialize the pointer\*/

1. newptr = CREATENODE(data)//\*refer to 41st algo for CREATENODE()

/\*error handling

IF newptr == NULL

PRINT “no space”

RETURN\*/

2. newnode->next = head

3. RETURN newnode

4. END

>Insertion in the middle of a list

1. ptr = head

2. IF ptr == NULL RETURN

3. WHILE ptr->next != NULL

{

4. IF head->next->next != NULL

{

5. head = head->next->next//this pointer moves faster

6. ptr = head->next//this pointer moves slower

}

7. ELSE BREAK

}

8. tmp = ptr->next

9. ptr->next = CREATENODE(data)

10. ptr->next->next = tmp

11. END

>Insertion at end of list - recursive method

/\*Initialize the pointer\*/

1. INSERT-END(head,data):

2. IF head == NULL

3. RETURN createNode(data)

4. head->next = INSERT-END(head->next,data)

5. END

>Deletion at beginning of list

1. IF head == NULL

2. RETURN head

3. /\*free head memory\*/

4. RETURN head->next

5. END

Deletion at end of list - recursive method

1. DELETE-END(head):

2. IF head == NULL

3. RETURN NULL

4. if(head->next == NULL)

5. RETURN NULL /\*free head memory\*/

6. head->next = DELETE-END(head->next)

7. END

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define init() ((struct node\*)malloc(sizeof(struct node)))

typedef struct node{

int data;

struct node \*next;

struct node \*prev;

}node;

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

//inserts a node in the begging

node\* insertBeg(node \*head,int data){

node \*newNode = createNode(data);

newNode->next = head;

return newNode;

}

//inserts at middle

//if nodes = even it adds at nodes/2

//if nodes = odd adds at nodes/2 + 1

void insertMiddle(node \*head,int data){

node \*ptr = head;

if(ptr == NULL){

//if list empty doesnot add eleemet

printf("empty\n");

return;

}

//runns till the end

while(head->next != NULL){

if(head->next->next != NULL){

head = head->next->next;//runs fast

ptr = ptr->next;//runs half the iterations

}

else break;

node \*temp = ptr->next;

ptr->next = createNode(data);

ptr->next->next = temp;

}

//recusive function which insert the node at the end

node\* insertEnd(node \*head, int data){

//if empty return a newlist with one element

if(head == NULL){

return createNode(data);

}

//calls insertend function

head->next = insertEnd(head->next, data);

return head;

}

//deletes node in the begging

node\* deleteBeg(node \*head){

node \*temp = head;

//node empty thus returns null

if(head == NULL){

printf("Empty\n");

return NULL;

}

//ideltes the node

printf("%d deleted\n",temp->data);

free(temp);

//returns the head's next elements

return head->next;

}

//deletes the last last node(not recursive)

node\* deleteEnd(node \*head){

node \*ptr = head;

//if empty list return NULL

if(head == NULL ){

printf("empty\n");

return NULL;

}

//if a single element returns NULL

if(head->next == NULL){

printf("%d deleted\n",head->data);;

free(temp);

return NULL;

}

//last node deleted

head->next = deleteEnd(head->next);

return head;

}

//displays the list

void display(node \*head){

while(head!=NULL){

printf("%d->",head->data);

head = head->next;

}

printf("NULL\n");

}

//main

int main () {

int choice,data;

node \*head ;

while(1){

//menu

printf("\n\*\*\*Main Menu\*\n");

printf("\nChoose one option from the following list ...\n"); printf("\n===============================================\n");

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert middle.\n4.Delete num at the begining \n5.Delete num at the end\n6.Display\n7.Exit\n");

printf("\nEnter your choice?\n");

scanf("\n%d",&choice);

//performs operation according to the choice

switch(choice){

case 1:{

printf("Enter the data to be inserted\n");

scanf("%d",&data);

head = insertBeg(head, data);

break;

}case 2:{

printf("Enter the data to be inserted\n");

scanf("%d",&data);

head = insertEnd(head, data);

break;

}case 3:{

printf("Enter the data to be inserted\n");

scanf("%d",&data);

insertMiddle(head, data);

break;

}case 4:{

head = deleteBeg(head);

break;

}case 5:{

head = deleteEnd(head);

break;

} case 6:{

printf("The list:\n");

display(head);

break;

}

case 7: {exit(0);break;}

}

}

return 0;

}

**Output:**

|  |  |
| --- | --- |
| Add begin, display  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **1**  **Enter the element to be inserted**  **12**  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **6**  **The list**  **12 -> NULL** | Add End, display  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **2**  **Enter the data to be inserted**  **14**  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **6**  **The list:**  **12 -> 14 -> NULL** |
| Delete end Display  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **5**  **14 is deleted**  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **6**  **The list:**  **13 -> NULL** | Terminate  **\*\*\*meanu\***  **Choose one option from the following list…..**  **===============================**    **1.Insert in beginning**  **2.insert at last**  **3.insert middle**  **4.Delete num at beginning**  **5.delete num at end**  **6. display**  **7.Exit**  **Enter your choice?**  **7** |

**Conclusion:**

The complexity of insertion in beginning and deletion in the beginning are constant. While the complexities of deletion at end , insertion at end and insertion in middle are of O(n).

**PROGRAM 44:**

**Title:**

Creation of Doubly Linked list

**Objective:**

Write a C program to create a doubly linked list with 5 nodes.

**Explanation:**

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.

**Pseudo Code:**

>Create node function

/\*Assume that it is a function where data is passed\*/

1. SET data

2. newnode = NEW node //allocates memory to node

3. node->data = data

4. node->next = NULL

5. node ->prev = NULL

6. RETURN node

7. END

>Creating linked-list of 5 nodes

1. SET ptr=head=NULL

2. FOR i from 0 to 5

/\*for creating n nodes: i from 0 to n\*/

{

3. READ data

4. IF head == NULL: head = CREATENODE(data); ptr = head

5. ELSE

{

6. temp = CREATENODE(data)

7. ptr->next = temp

8. temp->prev = ptr

9. ptr = ptr->next

}

}

10. RETURN head

11. END

>Displaying linked list

/\*head is a instance of the actual head here\*/

1. WHILE head != NULL

2. {

3. PRINT head->data

4. head = head->next

5. }

6. END

**Code:**

#include <stdio.h>

#include <stdlib.h>

//double linked list node

typedef struct node

{

int data;

struct node \*next;

struct node \*prev;

}node;

node\* createNode(int data){

node \*newNode = ((node\*)malloc(sizeof(node)));

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

//function which creates a double list of 5 nodes

node\* createList(){

int n=5,data;

node \*p, \*head = NULL,\*temp;

//runs loop 5 times

while(n--){

printf("Enter a number\n");

scanf("%d",&data);

//for the first node

if(head == NULL){

//initializing newnode as head

head = createNode(data);

p = head;

}

//fpr other nodes

else{

temp = createNode(data);

p->next = temp;

temp->prev = p;

p = p->next;

}

}

//return the list of 5nodes

return head;

}

//displays nodes from head to the end

void display(node \*head){

while(head != NULL){

printf("%d ",head->data);

head = head->next;

}

printf("\n");

}//main

int main(){

node \*head = createList();

display(head);

return 0;

}

**output:**

Enter a number

2

Enter a number

5

Enter a number

6

Enter a number

77

Enter a number

98

2 5 6 77 98

CONCLUSION:This code works as expected and the time complexity is similar to that of linked-list. But it occupies more space since it stores both the address of previous and the next node.

**PROGRAM 45:**

**Title:**

Creation of Circular Linked list

**Objective:**

Write a C program to create a circular linked list with 5 nodes.

**Explanation:**

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. So the only change in creation function from linked list, is that the nth node instead of pointing to NULL. It points to the head.

**Pseudo Code:**

\*create node function is same as 41st problem

>Creating Circular linked-list of 5 nodes

1. SET ptr=head=NULL

2. FOR i from 0 to 5

/\*for creating n nodes: i from 0 to n\*/

{

3. READ data

4. IF head == NULL: head = ptr = CREATENODE(data)

5. ELSE

{

6. ptr->next = CREATENODE(data)

7. ptr = ptr->next

}

}

8. ptr->next = head//last created node points to head

9. RETURN head

10. END

>Displaying Circular linked list

/\*head is a instance of the actual head here\*/

1. ptr = head

//runs till it again reaches the address of where it started

2. WHILE head != ptr

3. {

4. PRINT head->data

5. head = head->next

6. }

7. END

**Code :**

#include <stdio.h>

#include <stdlib.h>

//declaration of a node

typedef struct Node{

int data;

struct Node \*next;

}node;

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

//function which creates a circualar list of 5 nodes

node\* createList(){

int n=5,data;

node \*p, \*head = NULL;

//runs loop 5 times

while(n--){

printf("Enter a number\n");

scanf("%d",&data);

if(head == NULL){

//intializing newnode as head

head = createNode(data);

p = head;

}

else{

p->next = createNode(data);

p = p->next;

}

}

//intializes the last eleemt next to head

p->next = head;

//return the list of 5nodes

return head;

}

//displays elemets in linked list till reaches the head

void display(node \*head){

node \*ptr = head;

//if list is not empty

if(head != NULL){

while(head->next != ptr){

printf("%d->",head->data);

head = head->next;

}

printf("%d connected to %d",head->data,ptr->data);

}

//if list is empty

else

printf("NULL\n");

}

//main

int main()

{

//driver code

node \*head=createList();

display(head);

return 0;

}

Enter a number

3

Enter a number

5

Enter a number

7

Enter a number

22

Enter a number

9

3->5->7->22->9 connected to 3

**Conclusion:**

This code is also a simple variation with a linked list. Thus its time complexities are also similar with linked lists for insertion and deletion.

**PROGRAM 46:**

**Title:**

Implementation Stack Using Linked Lists

**Objective:**

Write a C program to implement the stack using linked lists

**Explanation:**

The push operator is similar to the insertion in the beginning in the linked list. And dThe pop operator similar to the selection from end of the linked list(refer problem 43)

**Pseudo Code:**

>Pushing in Linked-Stack

/\*Get new node for the ITEM to be pusshed\*/

newptr = NEW Node

newptr->data = ITEM

newptr->next = NULL

/\*Add new node at the top\*/

IF top == NULL THEN

top = newptr

ELSE

{

newptr->next = top

top = newptr

}

END

>Popping from a Linked-Stack

1. IF top == NULL THEN

2. PRINT “Stack Empty, Underflow”

3. ELSE

{

4. PRINT top->data

5. top = top->link

}

6. END

**Code**

#include <stdio.h>

#include <stdlib.h>

//declaration of a node

typedef struct Node{

int data;

struct Node \*next;

}node;

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

//pushes a node into the stack

node\* push(node \*head,int data){

node \*newNode = createNode(data);

newNode->next = head;

return newNode;

}

//pops an element from the stack

node\* pop(node \*head){

node \*temp = head;

//node empty thus returns null

if(head == NULL){

printf("Empty\n");

return NULL;

}

//deletes the node

printf("%d deleted\n",temp->data);

free(temp);

//returns the stack after popping

return head->next;

}

//prints the top element

void peek(node \*head){

if(head == NULL) {

printf("empty\n");

return;

}

printf("%d\n",head->data);

}

//displays the stack from top to the end

void display(node \*head){

while(head!=NULL){

printf("%d\n",head->data);

head = head->next;

}

}

int main(){

//intialized variables needed

node \*top = NULL;

int choice,data;

//runs loop till user chooses exit --> 5

while(1){

//menu

printf("\n1. Push an element on to the STACK.\n"

"2. Pop and element from the STACK.\n"

"3. Peek the STACK.\n"

"4. Display the STACK.\n"

"5. Exit the program.\n");

scanf("%d",&choice);

//performs action according the choice

switch(choice){

case 1:{

printf("\nEnter an element to add\n");

scanf("%d",&data);

top = push(top,data);

break;

}

case 2:{

top = pop(top);

break;

}

case 3:{

peek(top);

break;

}

case 4:{

display(top);

break;

}

case 5:{

exit(0);

break;

}

default: printf("no such option choose again\n");

}

}

return 0;

}

**Output:**

|  |  |  |
| --- | --- | --- |
| **push2 elements**  **1. Push an element on to the STACK.**  **2. Pop and element from the STACK.**  **3. Peek the STACK.**  **4. Display the STACK.**  **5. Exit the program.**  **1**    **Enter an element to add**  **55**    **1. Push an element on to the STACK.**  **2. Pop and element from the STACK.**  **3. Peek the STACK.**  **4. Display the STACK.**  **5. Exit the program.**  **1**    **Enter an element to add**  **22** | **Pop andDisplay**    **1. Push an element on to the STACK.**  **2. Pop and element from the STACK.**  **3. Peek the STACK.**  **4. Display the STACK.**  **5. Exit the program.**  **2**  **22 deleted**  **1. Push an element on to the STACK.**  **2. Pop and element from the STACK.**  **3. Peek the STACK.**  **4. Display the STACK.**  **5. Exit the program.**  **4**  **55** |  |

**Display**

**1. Push an element on to the STACK.**

**2. Pop and element from the STACK.**

**3. Peek the STACK.**

**4. Display the STACK.**

**5. Exit the program.**

**3**

**22**

**1. Push an element on to the STACK.**

**2. Pop and element from the STACK.**

**3. Peek the STACK.**

**4. Display the STACK.**

**5. Exit the program.**

**4**

**22**

**55**

Exiting program

**1. Push an element on to the STACK.**

**2. Pop and element from the STACK.**

**3. Peek the STACK.**

**4. Display the STACK.**

**5. Exit the program.**

**5**

**Conclusion:**

This code is helpful to display the expected result. Time complexity Pushing is O(1) and popping is O(n), that is the popping operation is slower compared to the implementation using arrays.

**PROGRAM 47:**

**Title:**

Implementing queues using LInked lists

**Objective:**

Write a C program to implement the queue using a linked list.

**Explanation:**

The enqueue is similar to insertion in beginning and dequeue is similar to deletion in beginning in the linked list. SInce we have a reference point of where to delete in both cases.

**Pseudo Code:**

>EnQueue in Queue

/\*Allocate space for ITEM to be inserted\*/

1. newptr = NEW node//allocating memory

2. newptr->info = item; newptr->link = NULL

/\*insert in queue\*/

3. IF rear = NULL

{

4. front = newptr

5. rear = newptr

}

6. ELSE

{

7. rear->link = newptr

8. rear = newptr

}

9. END

>DeQueue in Queue

1. IF front = NULL THEN

2. PRINT “queue empty”

3. ELSE

{

4. item = front->info

5. IF front = rear THEN

6. front = rear = NULL

7. ELSE front = ront->link

}

8. End

**Code**

#include<stdio.h>

#include <stdlib.h>

//declaration of a node

typedef struct Node{

int data;

struct Node \*next;

}node;

struct Queue {

node \*front, \*rear;

};

//function which creates nodes

node\* createNode(int data){

node \*n = ((node\*)malloc(sizeof(node)));

n->data = data;

n->next = NULL;

return n;

}

void enQueue(struct Queue\* q, int data)

{

// Create a new LL node

node\* temp = createNode(data);

if (q->rear == NULL) {

q->front = q->rear = temp;

return;

}

q->rear->next = temp;

q->rear = temp;

}

void deQueue(struct Queue\* q)

{

if (q->front == NULL)

return;

node\* temp = q->front;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

free(temp);

}

void display(struct Queue \*q){

node \*save = q->front;

while(q->front != q->rear){

printf("%d-> ",q->front->data);

q->front = q->front->next;

}

printf("%d\n",q->rear->data );

q->front = save;

}

int main(int argc, char const \*argv[])

{

//initialized variables needed

struct Queue \*q = ((struct Queue\*)malloc(sizeof(struct Queue)));

int choice,data;

//runs loop till user chooses exit --> 5

while(1){

//menu

printf("\n1. EnQueue an element on to the STACK.\n"

"2. Dequeue and element from the STACK.\n"

"3. Display the STACK.\n"

"4. Exit the program.\n");

scanf("%d",&choice);

//performs action according the choice

switch(choice){

case 1:{

printf("\nEnter an element to add\n");

scanf("%d",&data);

enQueue(q, data);

break;

}

case 2:{

deQueue(q);

break;

}

case 3:{

display(q);

break;

}

case 4:{

exit(0);

break;

}

default: printf("no such option choose again\n");

}

}

return 0;

}

OUTPUT:

Enqueue- 2 elements

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**1**

**Enter an element to add**

**24**

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**1**

**Enter an element to add**

**25**

**Display:**

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**3**

**24 -> 25**

**Dequeue and display**

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**2**

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**3**

**24**

Exiting program

**1. EnQueue an element on to the STACK.**

**2. Dequeue and element from the STACK.**

**3. Display the STACK.**

**4. Exit the program.**

**4**

**Conclusion:**

The time complexity of both insertion and selection is constant since we have reference address where to delete. To display the time complexity is O(n).