.



**Vidyavardhini’s**

**College of Engineering & Technology**

Vasai Road (W)

**Department of Computer Engineering**

**Laboratory Reference Manual**

|  |  |  |  |
| --- | --- | --- | --- |
| Semester | III | Class | SE |
| Course No. | CSL303 | | |
| Course Name | Data Structure Lab | | |



**Vidyavardhini’s College of Engineering & Technology**

**Vision**

To be a premier institution of technical education; aiming at becoming a valuable resource for industry and society.

**Mission**

We at VCET aim

* To provide technologically inspiring environment for learning.
* To promote creativity, innovation and professional activities.
* To inculcate ethical and moral values.
* To cater personal, professional and societal needs through quality education.

**Department Vision:**

To evolve as a centre of excellence in the field of Computer Engineering to cater the industrial & societal needs.

**Department Mission:**

* To provide quality technical education with the aid of modern resources.
* To inculcate creative thinking through innovative ideas and project development.
* To encourage life-long learning, leadership skills, entrepreneur skills with ethical & moral values.

**Program Education Objectives (PEOs):**

The Program Educational Objectives (PEOs) of Bachelors Degree program in Computer Engineering are:

**PEO1**: To facilitate learners with a sound foundation in the mathematical, scientific and

engineering fundamentals to accomplish professional excellence and succeed in

higher studies in computer engineering domain.

**PEO2**: To enable learners to use modern tools effectively to solve real life problems in the

field of computer engineering.

**PEO3**: To equip learners with extensive education necessary to understand the impact of

computer technology in a global and social context.

**PEO4**: To inculcate professional and ethical attitude, leadership qualities, commitment to

societal responsibilities and prepare the learners for life-long learning to built up a

successful career in Computer Engineering.

**Program Specific Outcomes (PSOs):**

The Program Specific Outcomes (PEOs) of Bachelors Degree program in Computer Engineering are:

**PSO1:** Analyze problems and design applications of database, networking, security, web

technology, cloud computing, machine learning using mathematical skills and

computational tools

**PSO2:** Develop computer based systems to provide solutions for organizational, societal

problems by working in multidisciplinary teams and pursue a career in IT industry

**Program Outcomes (POs):**

Engineering Graduates will be able to:

* **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
* **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
* **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
* **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
* **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
* **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
* **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
* **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
* **PO9. Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
* **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
* **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
* **PO12. Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Course Objectives**

|  |  |
| --- | --- |
| 1 | To implement basic data structures such as arrays, linked lists, stacks and queues. |
| 2 | Solve problem involving graphs, and trees |
| 3 | To develop application using data structure algorithms |
| 4 | Compute the complexity of various algorithms. |

**Course Outcomes**

|  |  |  |  |
| --- | --- | --- | --- |
| At the end of the course student will be able to: | | PO/PSO | Bloom Level |
| CSL301.1 | Implement Linear Data Structure and handle insertion, deletion, traversal operations using array. | Implement | Apply(level 3) |
| CSL301.2 | Apply stack operations to convert and evaluate expression | Apply | Apply(level 3) |
| CSL301.3 | Implement linear, circular or priority queues using arrays | Implement | Apply(level 3) |
| CSL301.4 | Implement Singly, Circular or Doubly Linked list | Implement | Apply(level 3) |
| CSL301.5 | Implement ADT using insertion, deletion and searching operations on Binary tree. | Implement | Apply(level 3) |
| CSL301.6 | Implement Graph Traversal Techniques: BFS and DFS. | Implement | Apply(level 3) |

**Mapping of Experiments with Course Outcomes**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Experiment | Course Outcomes | | | | | |
| CSL  301.1 | CSL  301.2 | CSL  301.3 | CSL.  301.4 | CSL  301.5 | CSL  301.6 |
| Implement Stack ADT using array. | 3 | - | - | - | - | - |
| Convert an Infix expression to Postfix expression using stack ADT. | - | 3 | - | - | - | - |
| Evaluate Postfix Expression using Stack ADT. | - | 3 | - | - | - | - |
| Implement Linear Queue ADT using array. | - | - | 3 | - | - | - |
| Implement Priority Queue ADT using array. | - | - | 3 | - | - | - |
| Implement Singly Linked List ADT. | - | - | - | 3 | - | - |
| Implement Circular Linked List ADT. | - | - | - | 3 | - | - |
| Implement Binary Search Tree ADT using Linked List. | - | - | - | - | 3 | - |
| Implement Graph Traversal techniques (any1) – a) Depth First Search b) Breadth First Search | - | - | - | - | - | 3 |
| Implement Binary Search method. | - | - | - | - | - | 3 |
| Course Project | 3 | 3 | 3 | 3 | 3 | 3 |

Enter correlation level 1, 2 or 3 as defined below

1: Slight (Low) 2: Moderate (Medium) 3: Substatial (High)

If there is no correlation put “—“.

**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr. No.** | **Name of Experiment** | **D.O.P.** | **D.O.C.** | **Page No.** | **Remark** |
| 1 | Implement Stack ADT using array. |  |  |  |  |
| 2 | Convert an Infix expression to Postfix expression using stack ADT. |  |  |  |  |
| 3 | Evaluate Postfix Expression using Stack ADT. |  |  |  |  |
| 4 | Implement Linear Queue ADT using array. |  |  |  |  |
| 5 | Implement Priority Queue ADT using array. |  |  |  |  |
| 6 | Implement Singly Linked List ADT. |  |  |  |  |
| 7 | Implement Circular Linked List ADT. |  |  |  |  |
| 8 | Implement Binary Search Tree ADT using Linked List. |  |  |  |  |
| 9 | Implement Graph Traversal techniques (any1) – a) Depth First Search b) Breadth First Search |  |  |  |  |
| 10 | Implement Binary Search method. |  |  |  |  |
| 11 | Course Project |  |  |  |  |

D.O.P: Date of performance

D.O.C : Date of correction

|  |
| --- |
| Experiment No.1 |
| Implement Stack ADT using array. |
| Date of Performance:25/07/2023 |
| Date of Submission:13/08/2023 |

**Experiment No. 1:** **To implement stack ADT using arrays**

**Aim: To implement stack ADT using arrays.**

**Objective:**

1) Understand the Stack Data Structure and its basic operators.

2) Understand the method of defining stack ADT and implement the basic operators.

3) Learn how to create objects from an ADT and invoke member functions.

**Theory:**

A stack is a list in which all insertions and deletions are made at one end, called the top. It is a collection of contiguous cells, stacked on top of each other. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as Last In First Out (LIFO) lists.



The operations that can be performed on a stack are push, pop which are main operations while auxiliary operations are peek, isEmpty and isFull. Push is to insert an element at the top of the stack. Pop is deleting an element that is at the top most position in the stack. Peek simply examines and returns the top most value in the stack without deleting it.

Push on an already filled stack and pop on an empty stack results in serious errors so isEmpty and isFull function checks for stack empty and stack full respectively. Before any insertion, the value of the variable top is initialized to -1.

**Push Operation**



**Pop Operation**



**Peek Operation**



**Algorithm:**

PUSH(item)

1. If (stack is full)

Print “overflow”

2. top = top + 1

3. stack[top] = item

Return

POP()

1. If (stack is empty)

Print “underflow”

2. Item = stack[top]

3. top = top – 1

4. Return item

PEEK()

1. If (stack is empty)

Print “underflow”

2. Item = stack[top]

3. Return item

ISEMPTY()

1. If(top = -1)then

return 1

2. return 0

ISFULL()

1. If(top = max)then

return 1

2. return 0

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define max 15

int top = -1, a[max];

void push()

{

int x;

if (top == max - 1)

{ printf("stack overflow condition");

} else

{ printf("enter element to the stack\n"); scanf("%d", &x); top++; a[top] = x;

}

}

void display()

{ int i;

for (i = top; i >= 0; i--)

{ printf("%d\n",a[i]);

}

}

void pop()

{ if (top == -1)

{ printf("stack underflow condition\n");

}

else

{ printf("pop element:%d\n", a[top]);

top--;

}

}

int main()

{

int ch;

while (1)

{

printf("enter your choice\n1.Push\n2.Pop\n3.Display\n4.Exit\n");

scanf("%d", &ch);

switch (ch)

{ case 1: push(); break;

case 2: pop(); break;

case 3:

display(); break;

case 4:

exit(0); break;

default:

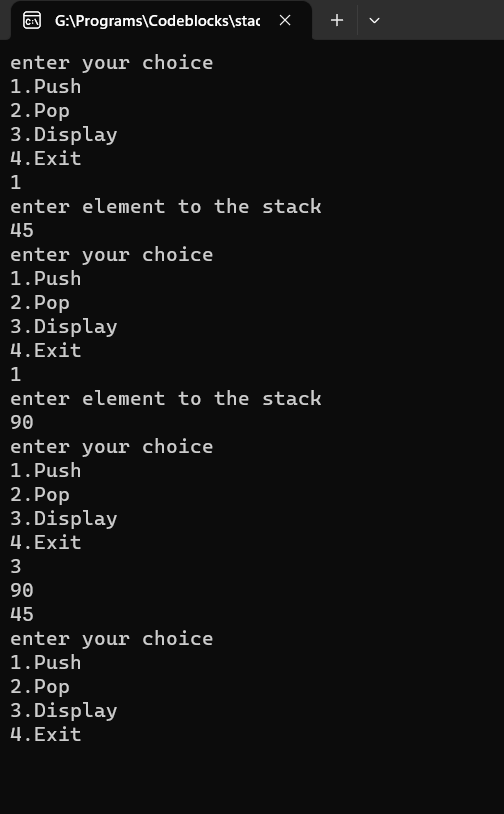
printf("invlaid choice\n"); break;

}

} return 0;

}

**Output:**



**Conclusion:**

Stack is one of the simplest data structures to implement and execute. It used LIFO operation i.e (last in first out) method to carry out push and pop operations. Time complexity of the program was O(1).Here Arrays are used to stored data instead of linked list so it was easier to display. Also we can use stack as function ton carry out another tasks.

Applications:

1. Well formed-ness of Parenthesis
2. Infix to postfix conversation
3. Postfix evaluation
4. Prefix evaluation

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.2 |
| Convert an Infix expression to Postfix expression using stack ADT. |
| Date of Performance: |
| Date of Submission: |

.

**Experiment No. 2: Conversion of Infix to postfix expression using stack ADT**

**Aim: To convert infix expression to postfix expression using stack ADT.**

**Objective:**

1) Understand the use of Stack.

2) Understand how to import an ADT in an application program.

3) Understand the instantiation of Stack ADT in an application program.

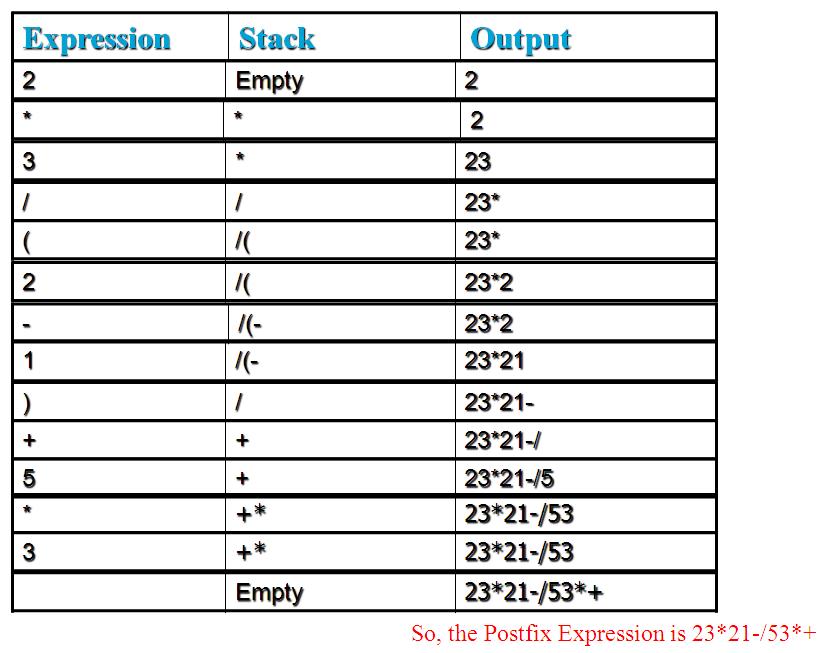
4) Understand how the member functions of an ADT are accessed in an application program.

**Theory:**

Postfix notation is a way of writing algebraic expressions without the use of parentheses or rules of operator precedence. The expression (A+B)/(C–D) would be written as AB+CD-/ in postfix notation. An expression is scanned from user in infix form; it is converted into postfix form and then evaluated without considering the parenthesis and priority of the operators.

An arithmetic expression consists of operands and operators. For a given expression in an postfix form, stack can be used to evaluate the expression. The rule is whenever an operands comes into the string push it on to the stack and when an operator is found then last two elements from the stack are poped and computed and the result is pushed back on to the stack. One by one whole string of postfix expression is parsed and final result is obtained at an end of computation that remains in the stack.

Conversion of infix to postfix expression



**Algorithm :**

**Conversion**

1. Read the symbol one at a time from the input expression.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then check its incoming priority

If stack is empty, push operator on stack.

If the top of stack is opening parenthesis, push operator on stack

If it has higher priority operator than the top of stack, push operator on stack.

Else pop the operator from the stack and output it, repeat step 4

1. If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
2. If there is more input go to step 1
3. If there is no more input, pop the remaining operators to output.

**Code:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

#define max 20

char stack[max],infix[max],postfix[max];

int top=-1;

void push(char);

char pop();

int is\_empty();

void convert();

int priority(char);

int main()

{

top=-1;

gets(infix);

convert();

printf("%s",postfix);

return 0;

}

void push(char c)

{

top++;

stack[top]=c;

}

char pop()

{

char c;

if(is\_empty()==1)

{

printf("underflow");

}

else {

c=stack[top];

top--;

return c;

}

}

int is\_empty()

{

if(top==-1)

return 1;

else return 0;

}

void convert(){

char symbol,next;

int i=0,j=0;

symbol=infix[i];

for(i=0;i<strlen(infix);i++)

{

switch(symbol)

{

case '(': push(symbol);

break;

case ')':

while((next=pop())!='(')

postfix[j++]=next;

break;

case '+':

case '-':

case '\*':

case '/':

case '^':

while(!is\_empty()&& priority(stack[top])>=priority(symbol))

postfix[j++]=pop();

push(symbol);

break;

default :

postfix[j++]=symbol;

}

}

while(!is\_empty())

{

postfix[j++]=pop();

}

postfix[j]='\0';

}

int priority(char symbol)

{

switch (symbol)

{

case '+':

case '-':

return 1;

break;

case '\*':

case '/':

return 2;

break;

case '^':

return 3;

break;

default:

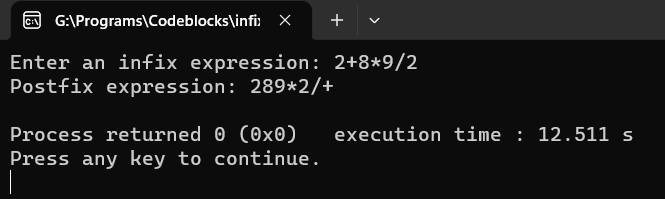
return 0;

break;

}

}

**Output:**

****

**Conclusion:**

Using stack implementation and converting infix expressions to postfix expressions is a fundamental operation in computer science and programming. It's often used in compilers, calculators, and various applications that involve evaluating mathematical expressions. The conversion process involves changing the order of operators and operands in a way that makes the expression easier to evaluate.We here manipulate the operators rather than the operands and carry out the function.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.3 |
| Evaluate Postfix Expression using Stack ADT. |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 3: Evaluation of Postfix Expression using stack ADT**

**Aim :** **Implementation of Evaluation of Postfix Expression using stack ADT**

**Objective:**

1) Understand the use of Stack.

2) Understand importing an ADT in an application program.

3) Understand the instantiation of Stack ADT in an application program.

4) Understand how the member functions of an ADT are accessed in an application program

**Theory:**

An arithmetic expression consists of operands and operators. For a given expression in an postfix form, stack can be used to evaluate the expression. The rule is when ever an operands comes into the string push it on to the stack and when an operator is found then last two elements from the stack are poped and computed and the result is pushed back on to the stack. One by one whole string of postfix expression is parsed and final result is obtained at an end of computation that remains in the stack.

**Algorithm**

Algorithm : EVAL\_POSTFIX

Input : E is an expression in an postfix form.

Output : Result after computing the expression.

Data Structure : An array representation of stack is used with top as a pointer to the top most element.

1. Append # as an delimiter at an end of expression.

2. item = READ\_SYMBOL()

3. while item != '#' do

if item =operand then

PUSH(item)

else

op=item

y=POP()

x=POP()

t=x op y

PUSH(t)

end if

item = READ\_SYMBOL()

end while

4. value = POP()

5. stop

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 10

int stack[MAX];

char s[20];

int top, m;

void push(char);

char pop();

void evaluate();

int main()

{

top = -1;

printf("\n Enter the expression:");

gets(s);

evaluate();

return 0; // Return from main

}

char pop()

{

if (top == -1)

{

printf("Stack is empty\n");

return -1;

}

else

{

char ch = stack[top];

top--;

return ch;

}

}

void push(char ch)

{

if (top == MAX - 1)

{

printf("Stack is overflow\n");

}

else

{

top++;

stack[top] = ch;

}

}

void evaluate()

{

int n1, n2, n3;

int i = 0;

n3 = 0; // Initialize n3

while (s[i])

{

if (isdigit(s[i]))

{

m = s[i] - '0';

push(m);

}

else

{

n1 = pop();

n2 = pop();

switch (s[i])

{

case '+':

n3 = n2 + n1;

break;

case '-':

n3 = n2 - n1;

break;

case '/':

n3 = n2 / n1;

break;

case '\*':

n3 = n2 \* n1;

break;

case '%':

n3 = n2 % n1;

break;

default:

printf("Unknown operator\n");

return; // Exit the function instead of exiting the program

}

push(n3);

}

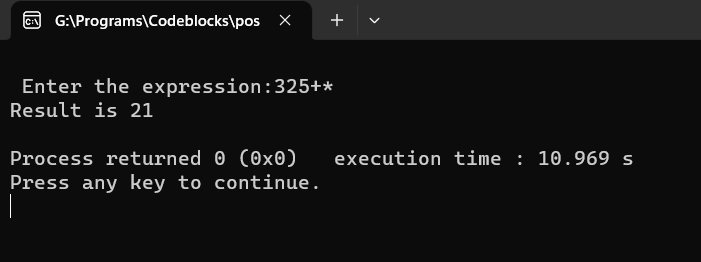
i++;

}

printf("Result is %d\n", n3);

}

**Output:**



**Conclusion:**

Postfix evaluation in C utilizes a stack to efficiently process mathematical expressions without the need for explicit parentheses. Understanding the postfix evaluation process and the underlying stack mechanism is valuable for implementing calculators, expression parsers, and other applications involving mathematical computations.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.4 |
| Implement Linear Queue ADT using Array |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 4: Impalement of Linear Queue ADT using Array**

**Aim: To implement a Queue using arrays.**

**Objective:**

1 Understand the Queue data structure and its basis operations.

2. Understand the method of defining Queue ADT and its basic operations.

3. Learn how to create objects from an ADT and member function are invoked.

**Theory:**

A Queue is an ordered collection of items from which items may be deleted at one end

(called the *front* of the queue) and into which items may be inserted at the other end (the *rear* of the queue). Queues remember things in first-in-first-out (FIFO) order. The basic operations in a queue are: Enqueue - Adds an item to the end of queue. Dequeue - Removes an item from the front



A queue is implemented using a one dimensional array. FRONT is an integer value, which contains the array index of the front element of the array. REAR is an integer value, which contains the array index of the rear element of the array. When an element is deleted from the queue, the value of front is increased by one. When an element is inserted into the queue, the value of rear is increased by one.

**Algorithm:**

ENQUEUE(item)

1. If (queue is full)

Print “overflow”

2. if (First node insertion)

Front++

3. rear++

Queue[rear]=value

DEQUEUE()

1. If (queue is empty)

Print “underflow”

2. if(front=rear)

Front=-1 and rear=-1

3. t = queue[front]

4. front++

5. Return t

ISEMPTY()

1. If(front = -1)then

return 1

2. return 0

ISFULL()

1. If(rear = max)then

return 1

2. return 0

**Code :**

#include <stdio.h>

#include <string.h>

#define max 10

int queue[max];

int rear = -1, front = -1;

void enqueue(int);

void dequeue();

void display();

void peek();

void enqueue(int m) {

if (rear >= max - 1) {

printf("overflow");

} else if (rear == -1 && front == -1) {

front = rear = 0;

queue[rear] = m;

} else {

rear++;

queue[rear] = m;

}

}

void dequeue() {

if (front == -1 && rear == -1) {

printf("underflow");

} else if (front == rear) {

printf("element popped is %d", queue[front]);

front = rear = -1;

} else {

printf("element popped is %d", queue[front]);

front++;

}

}

void display() {

int i = 0;

if (front == -1 && rear == -1) {

printf("Queue is empty\n");

} else {

printf("Elements are:\n");

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

printf("%d\n", queue[i]);

}

}

}

void peek() {

if (front == -1 && rear == -1) {

printf("Queue is empty\n");

} else {

printf("The top element is %d\n", queue[front]);

}

}

int main() {

int x, e;

do {

printf("Enter the choice:\n");

printf("1)Insertion:\n");

printf("2)Deletion:\n");

printf("3)Display:\n");

printf("4)peek:\n");

printf("5)exit:\n");

scanf("%d", &x);

switch (x) {

case 1:

printf("enter the element you want to add:\n");

scanf("%d", &e);

enqueue(e);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

peek();

break;

case 5:

printf("you have exited the program\n");

break;

default:

printf("wrong\n");

break;

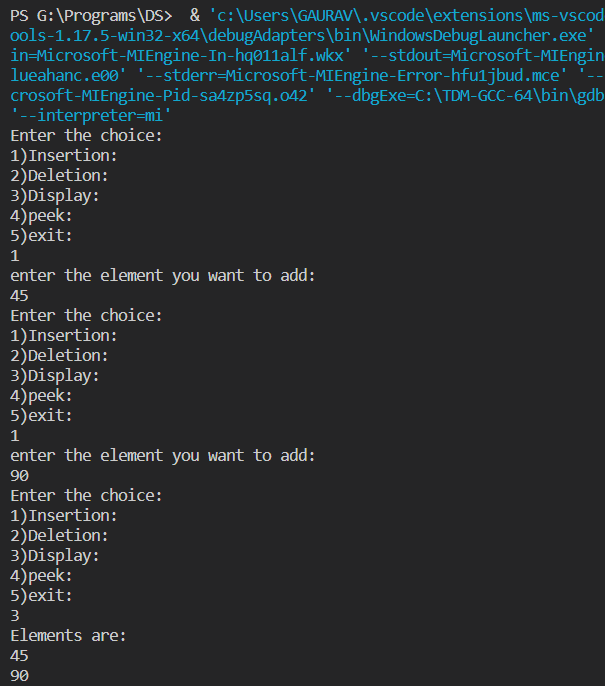
}

} while (x != 5);

return 0;

}

**Output**



**Conclusion:**

A queue in C is a versatile and data structure that overcomes the problems of insertion and deletion of elements whether from the front end or rear end. It follows the order of First In First Out (FIFO) and has the capability of determining the operations in a way that they can be enqueued and dequeued in any way. Queue data structure has a wide range of applications in computer science. Some of the common applications of Queue data structure are:

Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.5 |
| Implement Priority Queue ADT using array |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 5:** **Priority Queue**

**Aim**: ToImplement Priority Queue ADT using array

**Objective:**

Circular Queues offer a quick and clean way to store FIFO data with a maximum size

**Theory:**

A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back..

## **Operations of a Priority Queue:**

A typical priority queue supports the following operations:

### ****1) Insertion in a Priority Queue****

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

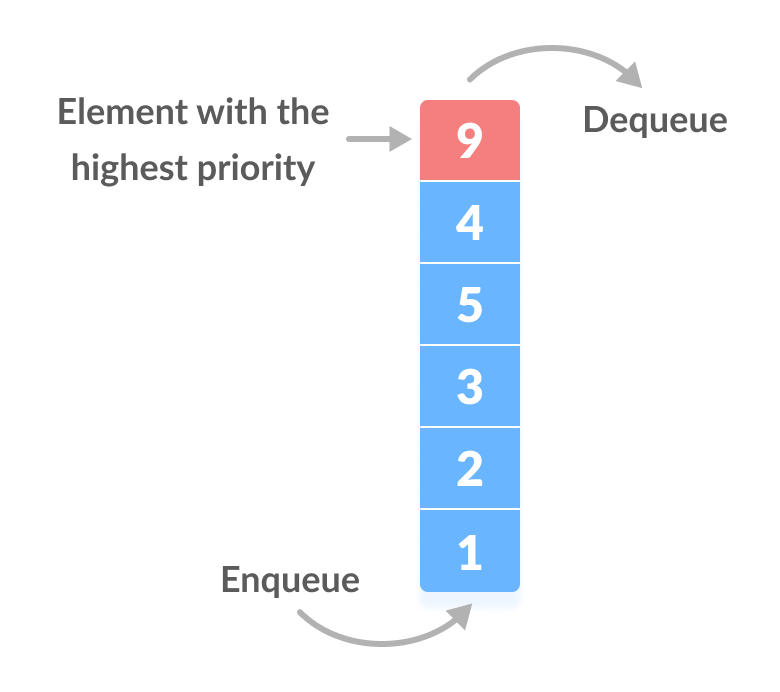
### ****2) Deletion in a Priority Queue****

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

### ****3) Peek in a Priority Queue****

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

**Priority Queue**



**Algorithm**

Algorithm : PUSH(HEAD, DATA, PRIORITY):

Step 1: Create new node with DATA and PRIORITY

Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.

Step 3: NEW -> NEXT = HEAD

Step 4: HEAD = NEW

Step 5: Set TEMP to head of the list

Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY

Step 7: TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: NEW -> NEXT = TEMP -> NEXT

Step 9: TEMP -> NEXT = NEW

Step 10: End

POP(HEAD):

Step 1: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.

Step 2: Free the node at the head of the list

Step 3: End

PEEK(HEAD):

Step 1: Return HEAD -> DATA

Step 2: End

**Code:**

#include<stdio.h>

#include<conio.h>

#define MAX 5

int array[MAX];

int front,rear;

void initialize()

{

front=rear=-1;

}

void enqueue(int x)

{

int i,j;

if(rear==-1)

{

front=rear=0;

array[front]=x;

}

else if(rear+1==MAX)

{

printf("Queue is full....cannot add\n");

}

else

{

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

{

if(x<array[i]) break;

}

rear++;

for(j=rear;j>i;j--)

{

array[j]=array[j-1];

}

array[i]=x;

}

}

int dequeue()

{

int x=-1;

if(front==-1)

printf("Queue is empty\n");

else

{

x=array[front];

front++;

if(front>rear) front=rear=-1;

}

return x;

}

int empty()

{

if(rear==-1) return 1;

else return 0;

}

int full()

{

if(rear==MAX-1) return 1;

return 0;

}

void display()

{

int i=front;

if(front==-1) printf("Queue is empty\n");

else

{

printf("Queue contains:\n");

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

{

printf("%d\n",array[i]);

}

}

}

void main()

{

int choice,x;

clrscr();

initialize();

do

{

printf("1.Enqueue an element\n2.Dequeue an element\n3.Display\n4.Check if queue is empty\n5.Check if queue is full\n6.Exit\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("Enter Your Choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:printf("Enter element to enqueue:");

scanf("%d",&x);

enqueue(x);

break;

case 2:x=dequeue();

if(x!=-1) printf("Element deleted from queue is%d\n",x);

break;

case 3:display();

break;

case 4:if(empty()) printf("Queue is empty\n");

else printf("Queue is not empty\n");

break;

case 5:if(full()) printf("Queue is full\n");

else printf("Queue is not full\n");

break;

case 6:break;

default:printf("Invalid choice\n");

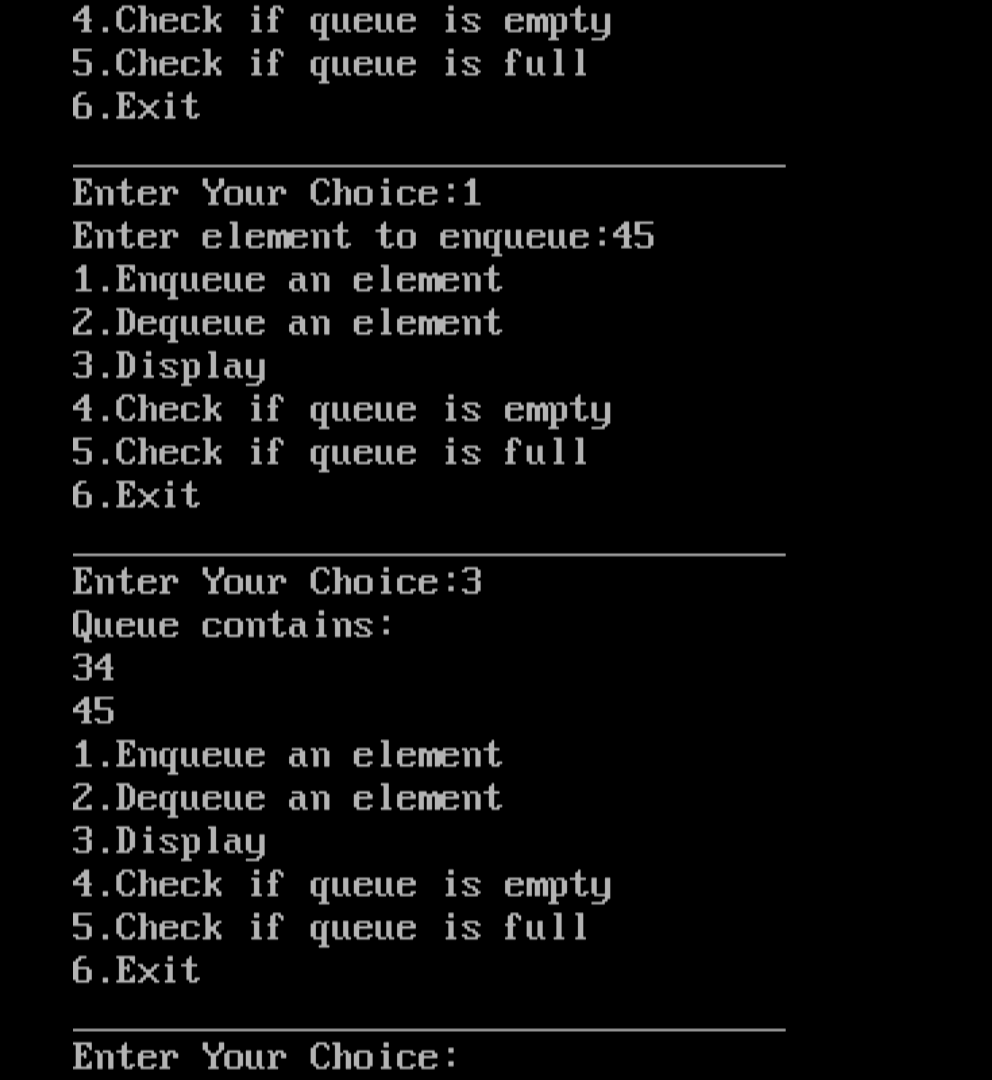
}

}while(choice!=6);

getch();

}

**Output:**

****

**Conclusion:**

A priority queue is a special type of queue in which each element is associated with a priority value. Elements are served based on their priority, with higher priority elements being served first. If elements with the same priority occur, they are served according to their order in the queue.

The time complexity of the program is O(n) but the same program can be redo with the help of heap concept to minimize the time complexity and maximize the efficiency.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.6 |
| Implement Singly Linked List ADT |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 6: Singly Linked List Operations**

**Aim: Implementation of Singly Linked List**

**Objective:**

It is used to implement stacks and queues which are like fundamental needs throughout computer science. To prevent the collision between the data in the hash map, we use a singly linked list.

**Theory :**

A linked list is a ordered collection of finite, homogenous elements referred as a node. Each node consist of two fields: one field for data which is referred as information and other field is an address field to store the address of next element in the list.

The address field of last node contains null value to indicate the end of list. The elements of linked list are not stored in continuous memory location but they are scattered, and still bounded to each other by an explicit link

The structure of linked list is as shown below

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

|  |  |
| --- | --- |
| 2 |  |

|  |  |
| --- | --- |
| 5 |  |

|  |  |
| --- | --- |
| 1 |  |

|  |  |
| --- | --- |
| 12 | null |

Header is a node containing null in its information field and an next address field contains the address of first data node in the list. Various operations can be performed on singly linked list like insertion at front, end and at specified position , deletion at front, end and at specified position, traversal, copying and merging.

**Algorithm**

Algorithm : INSERT\_SPECIFIED(Header, X, Key)

Input : Header is a pointer to header node. X is a data of node to be inserted and Key is data of node after which insertion is to be done.

Output :A singly linked list enriched with newly inserted node.

Data Structure : A singly linked list whose address of starting node is in Header. And two fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. new = GETNODE()
2. if new = NULL then

print “Insufficient Memory”

Exit

3. else

ptr = HEADER

while info(ptr)!= Key AND next(ptr)!= NULL do

ptr = next(ptr)

end while

1. if next(ptr) = NULL then

print “Key not found”

exit

1. else

next(new) = next(ptr)

info(new) = x

next(ptr) = new

end if

end if

1. stop

Algorithm : DELETE\_SPECIFIED(Header, Key)

Input : Header is a pointer to header node. Key is data of node after which is to be deleted.

Output :A singly linked list with removed node.

Data Structure : A singly linked list whose address of starting node is in Header. And two fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. ptr1 = HEADER

ptr = next(ptr1)

1. while ptr!= NULL do

if info(ptr) != Key

ptr1 = ptr

ptr = next(ptr)

else

next(ptr1) = next(ptr)

print(info(ptr))

FREENODE(ptr)

End if

End while

1. if ptr = NULL then

print “ key not found”

end if

1. stop

Algorithm : TRAVERSAL(Header)

Input : Header is a pointer to header node.

Output :A singly linked list is traversed and its data value is printed.

Data Structure : A singly linked list whose address of starting node is in Header. And two fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. ptr = next(HEADER)
2. while ptr!= NULL do

print (Info(ptr))

End while

1. stop

**Code:**

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

void display();

void insert\_at\_start(int n);

void insert\_at\_end(int n);

void insert\_in\_middle(int n);

void delete\_at\_start();

void delete\_at\_end();

void deleteFromMid();

void main() {

int choice, ele, position;

do {

printf("Menu Details\n");

printf("1) Insert at Start\n");

printf("2) Insert at End\n");

printf("3) Insert in Middle\n");

printf("4) Delete at Start\n");

printf("5) Delete at End\n");

printf("6) Delete at Middle\n");

printf("7) Display\n");

printf("8) Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_at\_start(ele);

break;

case 2:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_at\_end(ele);

break;

case 3:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_in\_middle(ele);

break;

case 4:

delete\_at\_start();

break;

case 5:

delete\_at\_end();

break;

case 6:

deleteFromMid();

break;

case 7:

display();

break;

case 8:printf("Program Closed\n");

break;

default:

printf("Invalid Choice\n");

break;

}

} while (choice != 7);

}

void display() {

struct node \*temp = head;

if (temp == NULL) {

printf("Linked List is empty\n");

} else {

printf("The Elements are:\n");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

}

void insert\_at\_start(int n) {

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

newnode->data = n;

newnode->next = head;

head = newnode;

}

void insert\_at\_end(int n) {

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

newnode->data = n;

newnode->next = NULL;

if (head == NULL) {

head = newnode;

return;

}

struct node \*temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newnode;

}

void deleteFromMid() {

int size;

struct node \*temp, \*current,\*ct;

if(head == NULL) {

printf("List is empty \n");

return;

}

else {

ct=head;

while(ct!=NULL){

ct=ct->next;

size++;

}

int count = (size % 2 == 0) ? (size/2) : ((size+1)/2);

if( head != NULL ) {

temp = head;

current = NULL;

for(int i = 0; i < count-1; i++){

current = temp;

temp = temp->next;

}

if(current != NULL) {

current->next = temp->next;

temp = NULL;

}

else {

head = temp->next;

temp = NULL;

}

}

else {

head = NULL;

}

}

size--;

}

void delete\_at\_start() {

if (head == NULL) {

printf("Underflow\n");

} else {

struct node \*temp = head;

head = head->next;

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

free(temp);

}

}

void delete\_at\_end() {

if (head == NULL) {

printf("Underflow\n");

} else if (head->next == NULL) {

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

free(head);

head = NULL;

} else {

struct node \*temp = head;

while (temp->next->next != NULL) {

temp = temp->next;

}

printf("The element deleted: %d\n", temp->next->data);

free(temp->next);

temp->next = NULL;

}

}

void insert\_in\_middle(int n)

{

struct node \*ct;

int size;

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

newNode->data = n;

newNode->next = NULL;

if(head == NULL) {

head = newNode;

}

else {

ct=head;

while(ct!=NULL){

ct=ct->next;

size++;

}

struct node \*temp, \*current;

//Store the mid position of the list

int count = (size % 2 == 0) ? (size/2) : ((size+1)/2);

temp = head;

current = NULL;

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

current = temp;

temp = temp->next;

}

current->next = newNode;

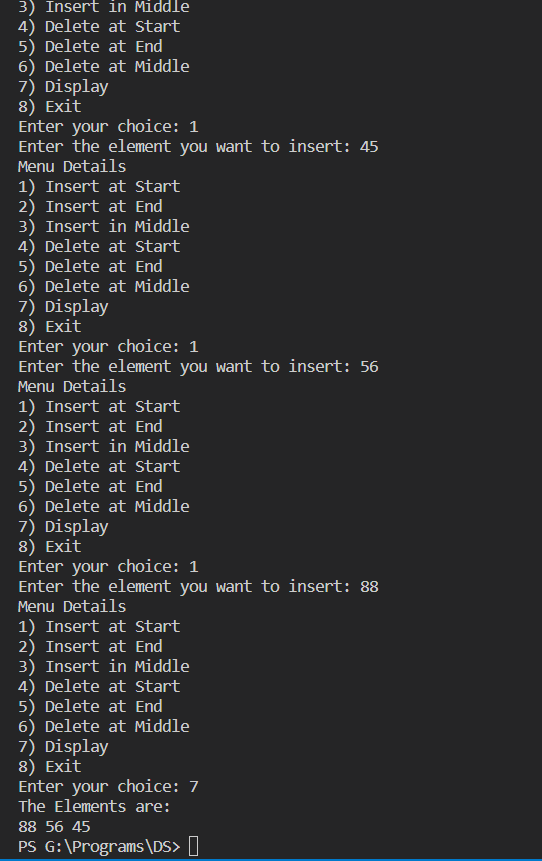
newNode->next = temp;

}

size++;

}

**Output:**

****

**Conclusion:**

A singly linked list in C is a fundamental data structure where each element, known as a node, holds a value and a pointer to the next node in the list. The start of the list is referred to as the head, and the end of the list (which points to NULL) is known as the tail. Singly linked lists are dynamic and allow for efficient insertion and deletion of nodes at any position in the list. However, they do not support direct access to individual elements; one must traverse the list from the head to reach a specific node. Despite this limitation, singly linked lists are widely used due to their simplicity and flexibility in handling data in scenarios where direct access is not a priority

Time Complexity :0(n(square))

Application:

* [Singly linked lists are used to implement other data structures like stack and queue](https://www.prepbytes.com/blog/c-programming/singly-linked-list-program-in-c/).
* [They are used in many real-world applications such as representing polynomials with one or two variables](https://www.prepbytes.com/blog/c-programming/singly-linked-list-program-in-c/)

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.7 |
| Implement Circular Linked List ADT |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 7: Circular Linked List Operations**

**Aim: Implementation of Circular Linked List ADT**

**Objective:** Circular Linked Lists can be used to manage the computing resources of the computer. Data structures such as stacks and queues are implemented with the help of the circular linked lists

**Theory :**

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



**Algorithm**

Algorithm :

* **Step 1:** IF PTR = NULL

  Write OVERFLOW  
 Go to Step 11  
 [END OF IF]

* **Step 2:** SET NEW\_NODE = PTR
* **Step 3:** SET PTR = PTR -> NEXT
* **Step 4:** SET NEW\_NODE -> DATA = VAL
* **Step 5:** SET TEMP = HEAD
* **Step 6:** Repeat Step 8 while TEMP -> NEXT != HEAD
* **Step 7:** SET TEMP = TEMP -> NEXT

[END OF LOOP]

* **Step 8:** SET NEW\_NODE -> NEXT = HEAD
* **Step 9:** SET TEMP → NEXT = NEW\_NODE
* **Step 10:** SET HEAD = NEW\_NODE
* **Step 11:** EXIT

**Code:**

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

void display();

void insert\_at\_start(int n);

void insert\_at\_end(int n);

void insert\_in\_middle(int n);

void delete\_at\_start();

void delete\_at\_end();

void deleteFromMid();

void main() {

int choice, ele, position;

do {

printf("Circular Linked List \n");

printf("1) Insert at Start\n");

printf("2) Insert at End\n");

printf("3) Insert in Middle\n");

printf("4) Delete at Start\n");

printf("5) Delete at End\n");

printf("6) Delete at Middle\n");

printf("7) Display\n");

printf("8) Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_at\_start(ele);

break;

case 2:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_at\_end(ele);

break;

case 3:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

insert\_in\_middle(ele);

break;

case 4:

delete\_at\_start();

break;

case 5:

delete\_at\_end();

break;

case 6:

deleteFromMid();

break;

case 7:

display();

break;

case 8:printf("Program Closed\n");

break;

default:

printf("Invalid Choice\n");

break;

}

} while (choice != 7);

}

void display() {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct node\* temp = head;

do {

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

temp = temp->next;

} while(temp != head);

printf("\n");

}

void insert\_at\_start(int n) {

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

struct node \*temp = head;

newnode->data = n;

newnode->next = head;

if (head != NULL) {

while (temp->next != head) {

temp = temp->next;

}

temp->next = newnode;

} else {

newnode->next = newnode; // For the first node

}

head = newnode;

}

void insert\_at\_end(int n) {

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

struct node \*temp = head;

newnode->data = n;

newnode->next = head;

if (head != NULL) {

while (temp->next != head) {

temp = temp->next;

}

temp->next = newnode;

} else {

head = newnode;

newnode->next = newnode; // For the first node

}

}

void deleteFromMid() {

int size;

struct node \*temp, \*current,\*ct;

if(head == NULL) {

printf("List is empty \n");

return;

}

else {

ct=head;

while(ct!=NULL){

ct=ct->next;

size++;

}

int count = (size % 2 == 0) ? (size/2) : ((size+1)/2);

if( head != NULL ) {

temp = head;

current = NULL;

for(int i = 0; i < count-1; i++){

current = temp;

temp = temp->next;

}

if(current != NULL) {

current->next = temp->next;

temp = NULL;

}

else {

head = temp->next;

temp = NULL;

}

}

else {

head = NULL;

}

}

size--;

}

void delete\_at\_start() {

if (head == NULL) {

printf("Underflow\n");

} else {

struct node \*temp = head;

while (temp->next != head) {

temp = temp->next;

}

struct node \*nextNode = head->next;

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

free(head);

if (nextNode == head) {

head = NULL;

} else {

temp->next = nextNode;

head = nextNode;

}

}

}

void delete\_at\_end() {

if (head == NULL) {

printf("Underflow\n");

} else if (head->next == head) {

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

free(head);

head = NULL;

} else {

struct node \*temp = head;

while (temp->next->next != head) {

temp = temp->next;

}

struct node \*nextNode = temp->next;

printf("The element deleted: %d\n", nextNode->data);

free(nextNode);

temp->next = head;

}

}

void insert\_in\_middle(int n)

{

struct node \*ct;

int size;

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

newNode->data = n;

newNode->next = NULL;

if(head == NULL) {

head = newNode;

}

else {

ct=head;

while(ct!=NULL){

ct=ct->next;

size++;

}

struct node \*temp, \*current;

int count = (size % 2 == 0) ? (size/2) : ((size+1)/2);

temp = head;

current = NULL;

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

current = temp;

temp = temp->next;

}

current->next = newNode;

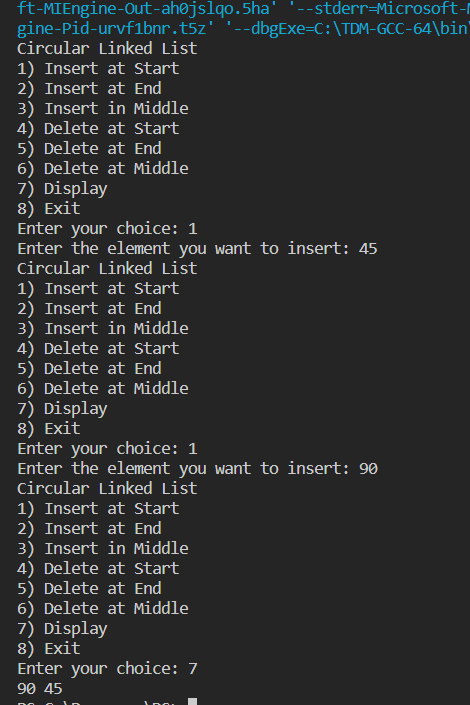
newNode->next = temp;

}

size++;

}

**Output:**



**Conclusion:**

A circular linked list in C is a dynamic data structure that has a cyclic structure, meaning the last node in the list points back to the first node, forming a circle. This unique structure allows for efficient operations at both ends of the list, making it particularly useful in scenarios where data needs to be continuously cycled or rotated. However, it requires careful handling of pointers during insertion and deletion operations to maintain the circular structure. Despite this complexity, circular linked lists are widely used due to their flexibility and efficiency in handling certain types of problems.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.8 |
| Implement Binary Search Tree ADT using Linked List. |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 8: Binary Search Tree Operations**

**Aim :** **Implementation of Binary Search Tree ADT using Linked List.**

**Objective:**

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

**Theory:**

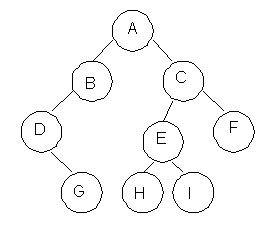
A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words node in a binary tree has at most two children and each child node is referred as left or right child.

Traversals in tree can be in one of the three ways : preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

1. Visit the root node R
2. Traverse the left subtree of R
3. Traverse the right sub tree of R



|  |  |
| --- | --- |
| **Description** | **Output** |
| Visit Root | A |
| Traverse left sub tree – step to B then D | ABD |
| Traverse right sub tree – step to G | ABDG |
| As left subtree is over. Visit root , which is already visited so go for right subtree | ABDGC |
| Traverse the left subtree | ABDGCEH |
| Traverse the right sub tree | ABDGCEHIF |

Inorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Visit the root node R
3. Traverse the right sub tree of R

|  |  |
| --- | --- |
| **Description** | **Output** |
| Start with root and traverse left sub tree from A-B-D | D |
| As D doesn’t have left child visit D and go for right subtree of D which is G so visit this. | DG |
| Backtrack to D and then to B and visit it. | DGB |
| Backtract to A and visit it | DGBA |
| Start with right sub tree from C-E-H and visit H | DGBAH |
| Now traverse through parent of H which is E and then I | DGBAHEI |
| Backtrack to C and visit it and then right subtree of E which is F | DGBAHEICF |

Postorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Traverse the right sub tree of R
3. Visit the root node R

|  |  |
| --- | --- |
| **Description** | **Output** |
| Start with left sub tree from A-B-D and then traverse right sub tree to get G | G |
| Now Backtrack to D and visit it then to B and visit it. | GD |
| Now as the left sub tree is over go for right sub tree | GDB |
| In right sub tree start with leftmost child to visit H followed by I | GDBHI |
| Visit its root as E and then go for right sibling of C as F | GDBHIEF |
| Traverse its root as C | GDBHIEFC |
| Finally a root of tree as A | GDBHIEFCA |

**Algorithm**

Algorithm: PREORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in preorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

visit(ptr)

PREORDER(LSON(ptr))\

PREORDER(RSON(ptr))

End if

1. Stop

Algorithm: INORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in inorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

INORDER (LSON(ptr))

visit(ptr)

INORDER (RSON(ptr))

End if

1. Stop

Algorithm: POSTORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in postorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

PREORDER(LSON(ptr))

PREORDER(RSON(ptr))

visit(ptr)

End if

1. Stop

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int key;

struct node \*right;

struct node \*left;

};

struct node \*newnode(int x) {

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

temp->key = x;

temp->left = NULL;

temp->right = NULL;

return temp;

}

struct node \*insert(struct node \*root, int ele) {

if (root == NULL) {

return newnode(ele);

} else if (ele > root->key) {

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

} else {

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

}

return root;

}

struct node \*getsucc(struct node \*root) {

struct node \*curr = root->right;

while (curr != NULL && curr->left != NULL) {

curr = curr->left;

}

return curr;

}

struct node \*delete(struct node \*root, int ele) {

if (root == NULL) {

return root;

}

if (ele > root->key) {

root->right = delete(root->right, ele);

} else if (ele < root->key) {

root->left = delete(root->left, ele);

} else {

// Case: Node with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

// Case: Node with two children, get the inorder successor

struct node \*temp = getsucc(root);

root->key = temp->key;

root->right = delete(root->right, temp->key);

}

return root;

}

void preorder(struct node \*root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

void inorder(struct node \*root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

void postorder(struct node \*root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

int search(struct node \*root, int x) {

if (root == NULL) {

return 0;

} else if (root->key == x) {

return 1;

} else if (x > root->key) {

return search(root->right, x);

} else {

return search(root->left, x);

}

}

int main() {

int choice, ele, choice2, f;

struct node \*root=NULL;

do {

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

printf("1) Insert\n");

printf("2) Delete\n");

printf("3) Display\n");

printf("4) Search\n");

printf("5) Exit\n");

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

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element you want to insert: ");

scanf("%d", &ele);

root = insert(root, ele);

break;

case 2:

printf("Enter the element you want to delete: ");

scanf("%d", &ele);

root = delete(root, ele);

break;

case 3:

printf("1) Preorder\n2) InOrder\n3) PostOrder\n");

printf("Enter your choice: ");

scanf("%d", &choice2);

switch (choice2) {

case 1:

preorder(root);

break;

case 2:

inorder(root);

break;

case 3:

postorder(root);

break;

default:

break;

}

printf("\n");

break;

case 4:

printf("Element you want to search: ");

scanf("%d", &ele);

f = search(root, ele);

if (f == 1) {

printf("Element found\n");

} else if (f == 0) {

printf("Element not found\n");

}

break;

case 5:

printf("Program Closed\n");

break;

default:

printf("Invalid Choice\n");

break;

}

} while (choice != 5);

return 0;

}

Output:

****

**Conclusion:**

Binary Search Trees (BST) in C language are a fundamental data structure that allow for efficient operations such as lookup, addition, and removal of items. They maintain elements in a sorted order and can dynamically adjust their size. However, to ensure optimal performance, it’s crucial to keep the BST balanced. Unbalanced BSTs can lead to slower operations, which is why self-balancing BSTs like AVL trees or Red-Black trees are often used in practice.

Time Complexity: O(H)

Space Complexity: O(H)

Here H represents the height of the node

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.9 |
| Implementation of Graph traversal techniques - Depth First Search, Breadth First Search |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 9: Depth First Search and Breath First Search**

**Aim :** **Implementation of DFS and BFS traversal of graph**.

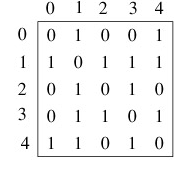
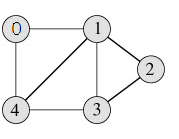
**Objective:**

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

**Theory:**

A graph is a collection of nodes or vertex, connected in pairs by lines referred as edges. A graph can be directed or undirected graph.

One method of traversing through nodes is depth first search. Here we traverse from starting node and proceeds from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



**DFS Traversal –0 1 2 3 4**

**Algorithm**

Algorithm: DFS\_LL(V)

Input: V is a starting vertex

Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

1. if gptr = NULL then

print “Graph is empty” exit

1. u=v
2. OPEN.PUSH(u)
3. while OPEN.TOP !=NULL do

u=OPEN.POP()

if search(VISIT,u) = FALSE then

INSERT\_END(VISIT,u)

Ptr = gptr(u)

While ptr.LINK != NULL do

Vptr = ptr.LINK

OPEN.PUSH(vptr.LABEL)

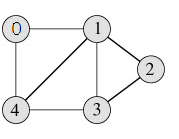
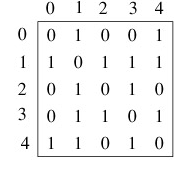
End while

End if

End while

1. Return VISIT
2. Stop

**BFS Traversal**

**BFS Traversal – 0 1 4 2 3**

**Algorithm**

Algorithm: DFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

push(j)

}

i=pop()

print("Visited vertex i")

visited[i]=1

count++

Algorithm: BFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

enqueue(j)

}

i=dequeue()

print("Visited vertex i")

visited[i]=1

count++

**Code:**

#include <stdio.h>

void DFS(int v);

typedef enum boolean { false, true } bool;

int n, arr[10][10];

bool visited[10];

void main() {

int i, j, v;

printf("Enter the Number of nodes:\n");

scanf("%d", &n);

printf("Enter the nodes:\n");

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

for (j = 1; j <= n; j++) {

scanf("%d", &arr[i][j]);

}

}

printf("Enter the starting node of DFS:\n");

scanf("%d", &v);

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

visited[i] = false;

}

DFS(v);

}

void DFS(int v) {

int i, stack[10], top = -1, pop;

top++;

stack[top] = v;

while (top >= 0) {

pop = stack[top];

top--;

if (visited[pop] == false) {

printf("%d ", pop);

visited[pop] = true;

} else

continue;

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

if (arr[pop][i] == 1 && visited[i] == false) {

top++;

stack[top] = i;

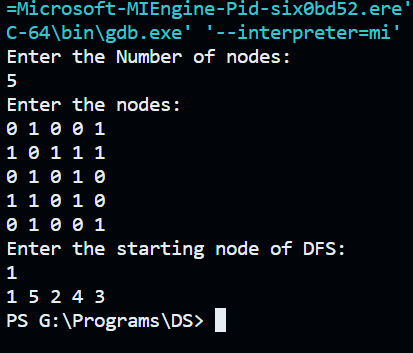
}

}

}

}

Output:



**Conclusion:**

Time Complexity about the above DFS Program is O(n2)

Depth-First Search (DFS) and Breadth-First Search (BFS) are fundamental graph traversal algorithms implemented in the C programming language, serving as versatile tools for exploring the structure of graphs, both directed and undirected. DFS delves deep into a graph, visiting nodes recursively along a branch before backtracking, making it suitable for tasks such as cycle detection and topological sorting. On the other hand, BFS systematically explores the graph level by level, facilitating the shortest path discovery and connected component identification. These algorithms are crucial in various applications, such as pathfinding in maps, network analysis, and puzzle-solving, offering efficient means to uncover relationships and uncover hidden patterns within graph data structures. Their flexibility and adaptability make them essential techniques for solving a wide range of real-world problems in C programming.

**NAME: GAURAV KISHOR PATIL**

**DIV: 2 ROLL NO:54**

**BATCH: C**

|  |
| --- |
| Experiment No.10 |
| Implementation of Binary Search Method |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 10: Binary Search Method**

**Aim :** **Implementation of Binary Search Method**

**Objective:** 1) Understand how to implement Binary Search algorithm.

**Theory:**

The improvement to searching method to reduce the amount of work can be done using binary searching. Binary searching is more efficient than linear searching if an array to be searched is in sorted manner.

Here an key item to be searched is compared with the item at middle of array. If they are equal search is completed. If the middle element is greater than key item searching proceeds with left sub array. Similarly, if middle element is less than key item than searching proceeds with right sub array and so on till the element is found.

For large arrays, this method is superior to sequential searching.

**Algorithm**

Algorithm : FIND(arr, x, first, last)

if (first > last)then

return -1

End if

mid = (first + last) / 2

if (arr[mid] = x)

return mid

End if

if (arr[mid] < x)

return find(arr, x, mid+1, last)

End if

return find(arr, x, first, mid-1)

**Code:**

#include<stdio.h>

#include<conio.h>

int binary\_search(int a[],int n,int limit){

int mid,low,high;

low=0;

high=limit-1;

while(low<=high){

mid=(low+high)/2;

    if(a[mid]==n){

return mid;

    }

    else if(n<a[mid]){

       high=mid-1;

    }

    else if(n>a[mid]){

       low=mid+1;

    }

}

 return 0;

}

void main(){

    int arr[20];

    int n,ele,f,i,j,temp;

    clrscr();

    printf("Enter the number of element you want ot insert:\n ");

    scanf("%d",&n);

    printf("Enter the number of element:\n ");

    for(i=0;i<n;i++){

scanf("%d",&arr[i]);

    }

      for(i=0;i<=n-2;i++){

     for(j=0;j<=n-2;j++){

      if(arr[j]>arr[j+1]){

      temp=arr[j];

     arr[j]=arr[j+1];

arr[j+1]=temp;

      }

     }

    }

   printf("Enter the  element you want ot search:\n ");

scanf("%d",&ele);

f=binary\_search(arr,ele,n);

if(f==0){

}

else{

printf("Element is found at the index %d\n",f);

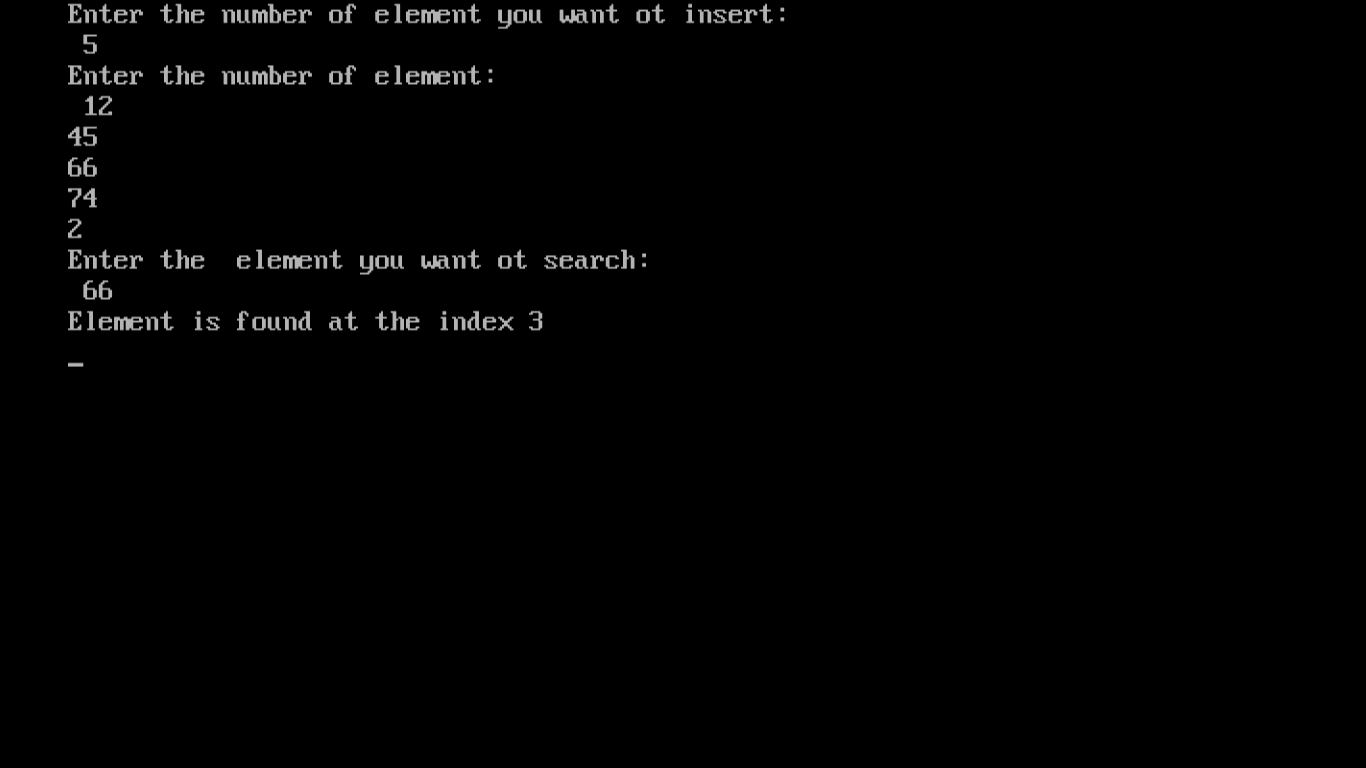
;

}

getch();

}

Output:



**Conclusion:**

Time Complexity: O(log n) – Average/worst case time complexity would be O(1) for the best case of time complexity.

In conclusion, the implementation of the Binary Search algorithm in the C programming language provides an efficient and widely used method for searching elements within a sorted array. By repeatedly dividing the search space in half, Binary Search quickly locates the target element with a time complexity of O(log n), making it suitable for large datasets. Its simplicity and effectiveness make it a fundamental tool in computer science and a go-to choice for searching and retrieval tasks, offering significant performance advantages over linear search methods.

.

|  |
| --- |
| Experiment No.11 |
| Course Project |
| Date of Performance: |
| Date of Submission: |