Thadomal Shahani Engineering College

Bandra (W.), Mumbai- 400 050.

CERTIFICATE

Certify that Mr./~~Miss~~ Tanmay Ritesh Sarode

of Computer Department, Semester III with

Roll No. 2203145 has completed a course of the necessary

experiments in the subject Data Structures under my

supervision in the **Thadomal Shahani Engineering College**

Laboratory in the year 2023 - 2024

Teacher In-Charge

Head of the Department

Date 23/10/2023

Principal

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

CONTENTS

Aim: Implement Stack ADT using array.

Program:

```
#include <stdio.h>
#include <stdlib.h>

struct stack
{
  int size;
  int * arr;
  int top;
};

int isFull(struct stack * s)
{
  if(s->top == s->size-1)
  {
    return 1;
  }
  return 0;
}

int isEmpty(struct stack * s)
{
```

```
  if(s->top == -1)
  {
    return 1;
  }
  return 0;
}


void push (struct stack * s)
{
  int val =0;
  if(isFull(s))
  {
    printf("Stack Overflow.\n");
  }
  else
  {
    printf("Enter the vaue to be pushed: \n");
    scanf("%d", &val);
    s->top++;
    s->arr[s->top] = val;
  }
```

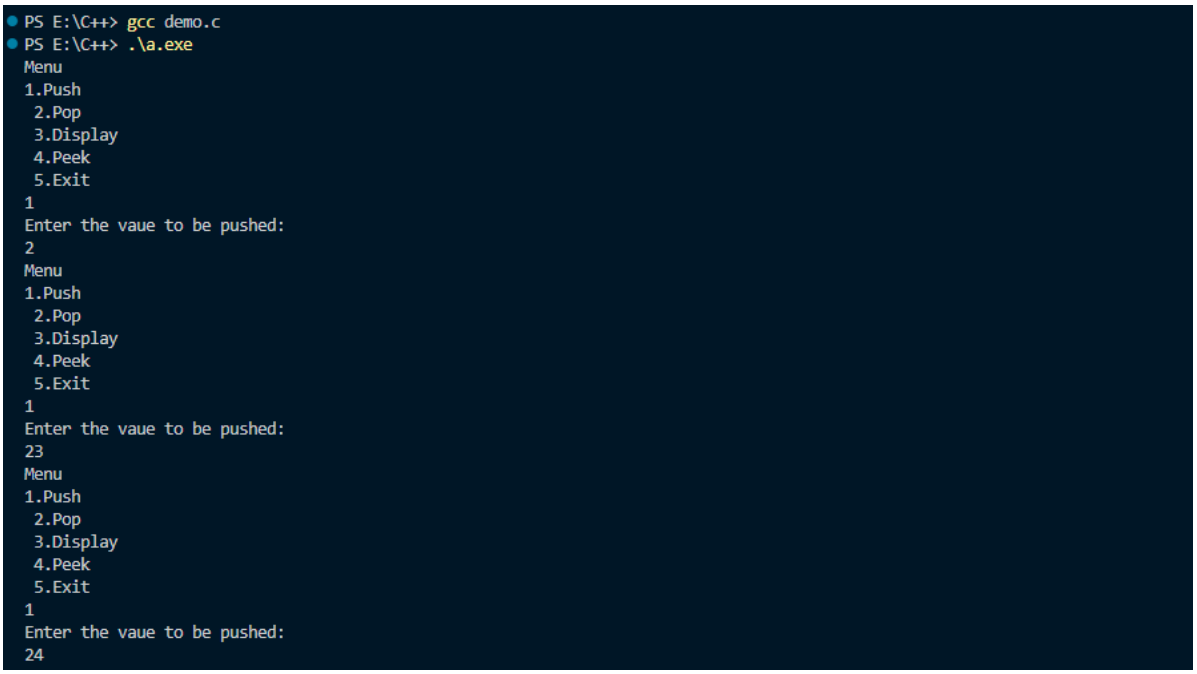
```
}

void pop(struct stack * s)
{
  int val=0;
  if(isEmpty(s))
  {
    printf("Stack Underflow\n");
  }
  else
  {
    val = s->arr[s->top];
    s->top--;
    printf("The popped value is: %d", val);
  }
}

void peek(struct stack * s )
{
  if(isEmpty(s))
  {
    printf("Stack Underflow.\n");
  }
```

```
  else
  {
    printf("Peeked value: %d", s->arr[s->top]);
  }
}

void Display(struct stack * s)
{
  int i = s->top;
  while(i != -1)
  {
    printf("%d\n", s->arr[i]);
    i--;
  }
}


int main()
{
  struct stack * s = (struct stack *) malloc(sizeof(struct stack));
  s->top = -1;
  s->arr = (int *)malloc(s->size*sizeof(int));
  s->size = 3;
```

```
int choice;
while(1)
{

printf("Menu\n");
printf("1.Push\n 2.Pop \n 3.Display \n 4.Peek \n 5.Exit");
scanf("%d", &choice);
switch(choice)
{
  case 1:
  push(s);
  break;
  case 2:
  pop(s);
  break;
  case 3:
  Display(s);
  break;
  case 4:
  peek(s);
  break;
  case 5:
```

```
    exit(0);
    default:
    printf("Invalid Choice.\n");
    break;
  }
  }
  return 0;
}
```

Output:



```
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
3

24
23
2
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
2
The popped value is: 24
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
4
Peeked value: 23
```

```
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
4
Peeked value: 23
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
2
The popped value is: 23
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
2
The popped value is: 2
Menu
1.Push
 2.Pop
 3.Display
 4.Peek
 5.Exit
2
Stack Underflow
```

Roll no: 145 Name: Tanmay-R. Sarode
Class: C3 Batch: C31

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Stack

A stack is an ordered collection of elements like array but it has a special feature that deletion & insertion of elements can be done only from one end called the 'TOP' of the stack.

Due to this property, it is called last in first out.

Operation on stack.

1. Create stack: We can create a stack.

2. Push stack: This is used to insert an element in the stack. However in an array, any element can be added at any place but in stack, we can add it on the top of the stack

3. Pop stack: This is used to remove an element in the stack. As pushing an element works on the top of stack popping also works at the top of stack.

4. Peep stack: Used to peek elements of the stack one by one using top variable which points a particular element in stack.

5. Empty stack: Used to make the stack empty.

6. Full stack: Used to full the stack.

7. Stack count: counts the no of elements in the stack.

8. destroy stack: destroys the count in the stack

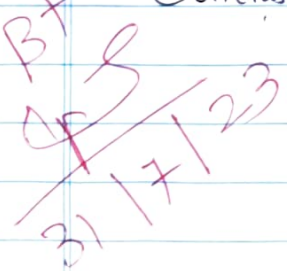
Applications-

1) Converting algebraic expressions from one form to other.
   - Infix to Postfix.
   - Infix to Prefix.
2) Evaluation of postfix expression
3) Paranthesis balancing in compilers
4) Recursive applications.
5) Depth first search traversal of graph

Stack is an abstract data type.

1. > Isfull () : used to check whether stack is full or not
2) Is empty (): used to check whether stack is empty or not
3. push (x): used to push x in the stack
4. pop (): used to delete one element from top of the stack.
5. peek(): used to get the top most element of the stack
6. size(): used to get no of elements present into the stack

Conclusion: Hence stack is implemented successfully.



Aim: Convert an infix expression to Postfix expression using stack

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct stack
{
   int top;
   char* arr;
   int size;
};

char StackTop(struct stack* sp)
{
   return sp->arr[sp->top];
}

int isEmpty(struct stack *sp)
{
  if(sp->top == -1)
  {
     return 1;
```

```
  }
  else
  {
     return 0;
  }
}

int isFull(struct stack *sp)
{
  if(sp->top == sp->size-1)
  {
     return 1;
  }
  else
  {
     return 0;
  }
}

void push(struct stack *sp,char val)
{
   if(isFull(sp))
   {
```

```
        printf("Stack Overflow");
    }
    else
    {
        sp->top++;
        sp->arr[sp->top] = val;
    }
}

void pop(struct stack * sp)
{

  if(isEmpty(sp))
  {
    printf("Stack Underflow");

  }
  else
  {
     sp->top--;

  }
}
```

```
int precedence(char val)
{
   if(val=='^' || val=='$')
   {
      return 3;
   }
   else if(val=='/'|| val=='*')
   {
      return 2;
   }
   else if(val=='+' || val=='-')
   {
      return 1;
   }
   else
   {
      return 0;
   }
}

int isSymbol(char val)
{
```

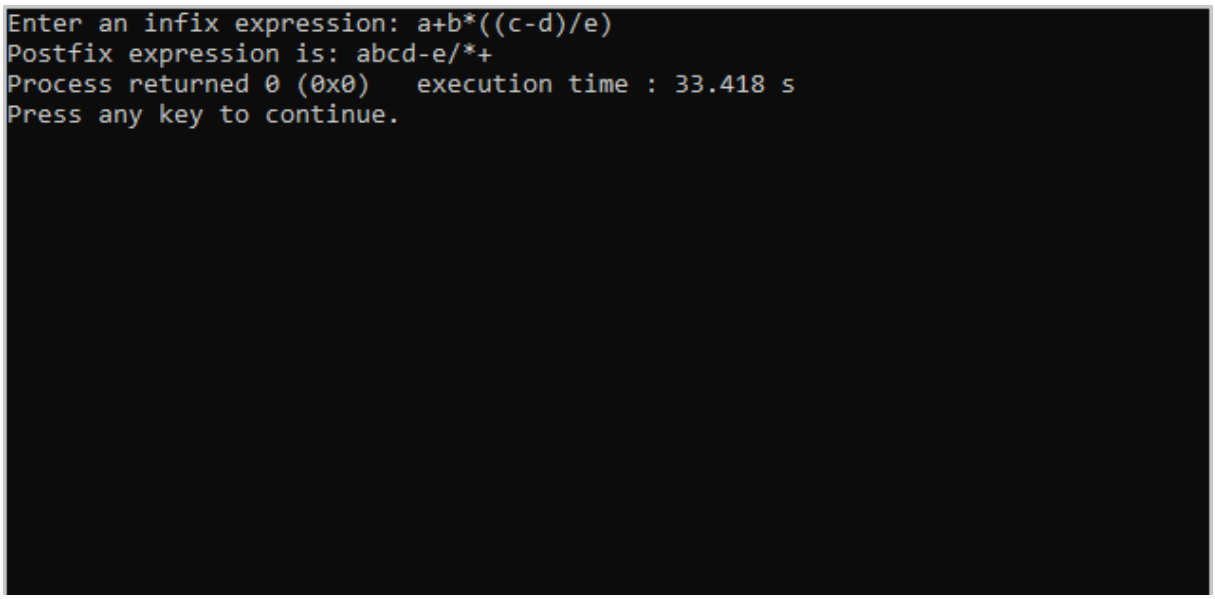
```
    if(val=='+' || val=='-' || val=='/' || val=='*' || val=='^' || val=='$'  )
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

char* infixToPostfix(char infix[50])
{
    struct stack* sp = (struct stack*)malloc(sizeof(struct stack));
    sp->size = 50; // Increase stack size to handle larger expressions
    sp->top = -1;
    sp->arr = (char*)malloc(sp->size * sizeof(char));
    char* postfix = (char*)malloc((strlen(infix) + 1) * sizeof(char));
    int i = 0; // Track infix traversal
    int j = 0; // Track postfix addition
    while (infix[i] != '\0')
    {
        if (!isSymbol(infix[i]) || infix[i] == '(' || infix[i] == ')')
```

```
{
    if (infix[i] == '(')
    {
        push(sp, infix[i]);
    }
    else if (infix[i] == ')')
    {
        while (!isEmpty(sp) && StackTop(sp) != '(')
        {
            postfix[j] = StackTop(sp);
            pop(sp);
            j++;
        }
        if (!isEmpty(sp) && StackTop(sp) == '(')
        {
            pop(sp); // Remove the '(' from the stack
        }
    }
    else
    {
        postfix[j] = infix[i];
        j++;
```

```
        }
        i++;
    }
    else
    {
        // Handle operator precedence here
        while (!isEmpty(sp) && precedence(StackTop(sp)) >=
precedence(infix[i]))
        {
            postfix[j] = StackTop(sp);
            pop(sp);
            j++;
        }
        push(sp, infix[i]);
        i++;
    }
}

// Empty the remaining operators in the stack
while (!isEmpty(sp))
{
    postfix[j] = StackTop(sp);
    pop(sp);
```

```
        j++;
    }

    postfix[j] = '\0';

    free(sp->arr);
    free(sp);

    return postfix;
}


int main()
{
    char infix[50];
    printf("Enter an infix expression: ");
    gets(infix);
    printf("Postfix expression is: %s", infixToPostfix(infix));

    return 0;
}
```

Output:


Enter an infix expression: a+b*((c-d)/e)
Postfix expression is: abcd-e/*+
Process returned 0 (0x0)   execution time : 33.418 s
Press any key to continue.

Name: Tanmay Sarode
C3 145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Infix to postfix conversion

An infix expression is what we write like a+b or a*b but infix expressions are not efficient to evaluate hence postfix expressions are used for faster performance

For converting an infix to postfix expression we need to follow some rules.

- Starting from left to right in an expression, take one character for consideration and if it is not a symbol (+, ^, *, -, /) we directly append it to the postfix string.
- If it is an opening paranthesis ('(') then it is directly pushed onto the stack.
- If it is a symbol we first check the precendence of the symbol at the top of stack and if its precedence is lower than the current symbol we push it onto the stack.
- In case of ~~same~~ greater precedence we pop out the symbol and append it to the postfix string. Also applicable for higher precedence.
- If it is a closing paranthesis we keep on popping out the symbols until we get the opening paranthesis but do not append the opening paranthesis & closing paranthesis to the string
- In case of same precedence we check the associativity if it is left to right then we pop otherwise we push

Precedence list:

^
* /
+ -

↑ Increasing order of precedence

Only '^' has an associativity of right to left the rest of the above symbols have ~~a preceden~~ associativity of left to right.

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

If we take the infix expression

a - b / (c * d ^ e)

character	stack	postfix string
a		a [Character direct append]
-	-	a
b	-	a b
/	- /	a b [precedence / > -]
(	- / (	a b ['(' direct push]
c	- / (	a b c
*	- / (*	a b c [Any symbol can be pushed on parenthesis]
d	- / (*	a b c d
^	- / (^	a b c d * [∵ precedence ^ > *]
e	- / (^	a b c d * e

∴ Postfix expression : a b c d * e ^ / - [Pop out the rest of the elements]

A
S
F
14/8/23

Aim: Evaluate Postfix expression using stack ADT.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct stack
{
   int size;
   int top;
   int *arr;
};

int StackTop(struct stack *sp)
{
   return sp->arr[sp->top];
}

int isEmpty(struct stack *sp)
{
   return (sp->top == -1);
}
```

```
int isFull(struct stack *sp)
{
    return (sp->top == sp->size - 1);
}

void push(struct stack *sp, int val)
{
    if (isFull(sp))
    {
        printf("Stack Overflow\n");
        exit(1);
    }
    sp->top++;
    sp->arr[sp->top] = val;
}

void pop(struct stack *sp)
{
    if (isEmpty(sp))
    {
        printf("Stack Underflow\n");
        exit(1);
    }
```

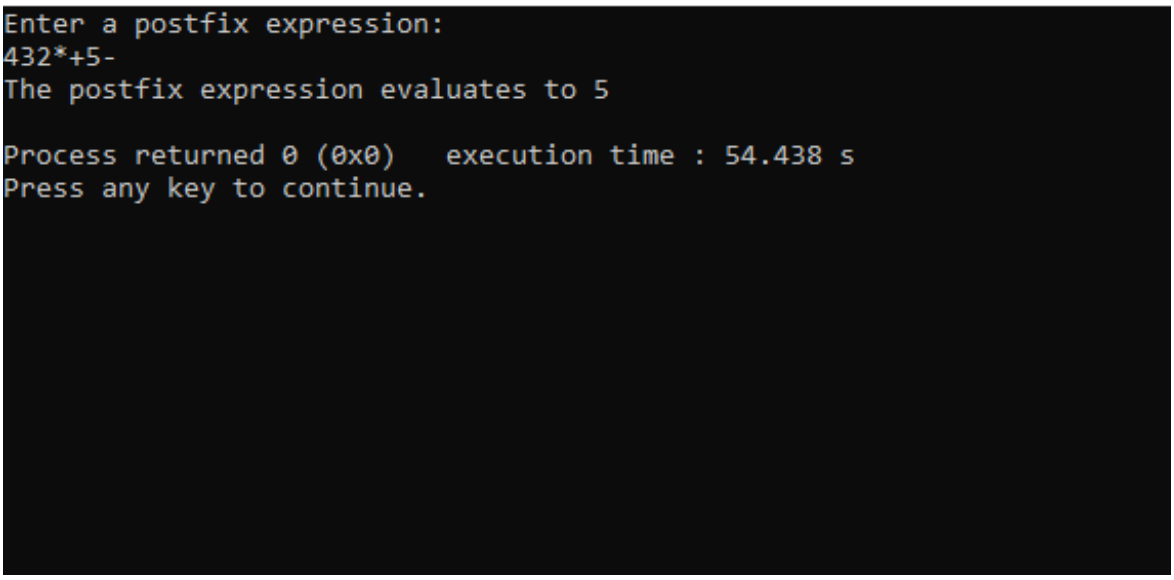
```
    sp->top--;
}

int isSymbol(char val)
{
    return (val == '*' || val == '-' || val == '+' || val == '/' || val == '^');
}

int power(int x, int n)
{
    int pow = 1;
    for (int i = 0; i < n; i++)
    {
        pow = pow * x;
    }
    return pow;
}

int postEval(char postfix[50])
{
    struct stack *sp = (struct stack *)malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
```

```
sp->arr = (int *)malloc((sp->size) * sizeof(int));

int i = 0;

while (postfix[i] != '\0')
{
   if (!isSymbol(postfix[i]))
   {
      int num = postfix[i] - '0';
      push(sp, num);
      i++;
   }
   else
   {
      int op1, op2;

      if (!isEmpty(sp))
      {
         op1 = StackTop(sp);
         pop(sp);
         op2 = StackTop(sp);
         pop(sp);
```

```
switch (postfix[i])
{
case '+':
   push(sp, op2 + op1);
   break;
case '-':
   push(sp, op2 - op1);
   break;
case '*':
   push(sp, op2 * op1);
   break;
case '/':
   push(sp, op2 / op1);
   break;
case '^':
   push(sp, power(op2, op1));
   break;
default:
   printf("Invalid Expression.\n");
   break;
}
i++;
```

```
            }
        }
    }

    int result = StackTop(sp);
    free(sp->arr);
    free(sp);

    return result;
}

int main()
{
    char postfix[50];
    int result;
    printf("Enter a postfix expression: \n");
    scanf("%s", postfix);
    result = postEval(postfix);
    printf("The postfix expression evaluates to %d\n", result);

    return 0;
}
```

Output:


Enter a postfix expression:
432*+5-
The postfix expression evaluates to 5
Process returned 0 (0x0) execution time : 54.438 s
Press any key to continue.

Name: Tanmay Sarode

C3 145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Postfix Evaluation

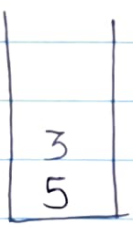
For evaluating a postfix expression some rules need to be followed.

- Starting from the left we keep on traversing the expression until we find a symbol [+, -, /, ^, *]
- ~~Once we find a symbol~~ for values other than the above mentioned symbols we push them into the stack.
- Once we find a symbol we pop out the first two elements of the stack and perform the required operation and push the result back into the stack.
- Once we have traversed through the expression there would only be one value in the stack if the expression is correct and that would be the result required
- Pop out the result out of the stack.

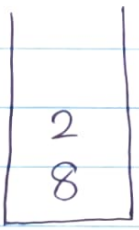
Consider the postfix expression

53+2*697-/-

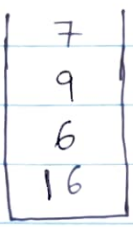
Stack



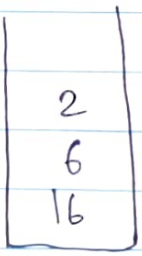
5 + 3 = 8 → push back



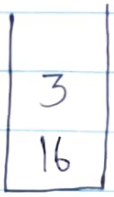
8 * 2 = 16



9 − 7 = 2



6 / 2 = 3



16 − 3 = 13

Hence the value of the postfix expression is 13.

For error handling the postfix evalution code. After all the evaluations are complete we need to check if there are more than 1 element present in the stack, if there are ~~more~~ multiple elements present than the postfix expression is wrong or invalid.
So we would check if stack Top is > 0 if it is then we print Invalid expression otherwise we push the result in the stack and print it out.

One more thing we need to check is that if there is an invalid symbol in the postfix expression. So in the switch case we would consider the cases of invalid symbols such as '$' & '?' and if these symbols are present we would stop the further evaluation and print out "invalid expression".

A
CS
21/8/23

Aim: Implement Linear Queue ADT using array.

Program:

```
#include <stdio.h>
#include <stdlib.h>


struct Queue{

    int front;
    int rear;
    int size;
    int *arr;

};

int isFull(struct Queue* q)
{
    if(q->rear == q->size-1)
    {
        return 1;
    }
    else
    {
```

```
        return 0;
    }
}

int isEmpty(struct Queue* q)
{
    if(q->front == q->rear)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void Create(struct Queue* q)
{
    printf("Enter the size of the queue: \n");
    scanf("%d", &(q->size));
    q->arr = (int*)malloc((q->size)*(sizeof(int)));
    q->front = -1;
    q->rear = -1;
```

```
    printf("Queue created successfully\n");
}


void Enqueue(struct Queue* q)
{
   int n;

   if(isFull(q))
   {
    printf("Queue is Full\n");
   }
   else
   {
       printf("Enter the element you want to enqueue: \n");
       scanf("%d",&n);
       q->rear++;

       q->arr[q->rear] = n;
   }

}
```

```
void Dequeue(struct Queue* q)
{
   int val=-1;
   if(isEmpty(q))
   {
      printf("No element for Dequeue\n");
   }
   else
   {
      q->front++;
      val = q->arr[q->front];
      printf("Dequeued element: %d\n", val);
   }

}
void Display(struct Queue* q)
{

   int i=q->front+1;
   while(i<= q->rear)
   {
      printf("Element %d: %d\n",i+1,q->arr[i]);
      i++;
```

```
    }

}

int main()
{
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
   int choice;
   printf(" 1.Create \n 2.Enqueue\n 3.Dequeue \n 4.Display \n 5.Exit\n");
   while(1)
   {
      printf("Enter your Choice: ");
      scanf("%d",&choice);

      switch(choice)
      {
      case 1:
         Create(q);
         break;
      case 2:
         Enqueue(q);
         break;
```

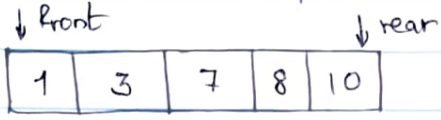
```
        case 3:
            Dequeue(q);
            break;
        case 4:
            Display(q);
            break;
        case 5:
            free(q->arr);
            free(q);
            exit(0);
            break;
        default:
            printf("Invalid choice.\n");
            break;
        }
    }

    return 0;
}
```

Output:

```
 1.Create
 2.Enqueue
 3.Dequeue
 4.Display
 5.Exit
Enter your Choice: 1
Enter the size of the queue:
4
Queue created successfully
Enter your Choice: 2
Enter the element you want to enqueue:
1
Enter your Choice: 2
Enter the element you want to enqueue:
2
Enter your Choice: 2
Enter the element you want to enqueue:
3
Enter your Choice: 2
Enter the element you want to enqueue:
4
Enter your Choice: 2
Queue is Full
Enter your Choice: 3
Dequeued element: 1
Enter your Choice: 4
Element 2: 2
Element 3: 3
Element 4: 4
Enter your Choice: 3
Dequeued element: 2
Enter your Choice: 3
Dequeued element: 3
Enter your Choice: 3
Dequeued element: 4
Enter your Choice: 3
No element for Dequeue
Enter your Choice: 5

Process returned 0 (0x0)   execution time : 37.723 s
Press any key to continue.
```

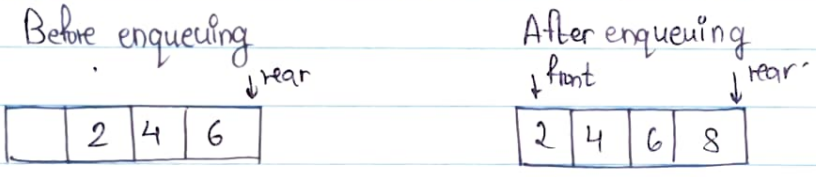
· Linear Queue

A linear queue is a basic data structure that follows the First in First out (FIFO) principle. It's a simple arrangement of elements in a linear order, like a line or queue in the real world. When elements are added, they go to the back, and when removed, they come out from the front. This structure is often used in programming for managing tasks, resources or data that need to be processed in the order they were added.



The above diagram is the basic structure of a queue. It contains two pointers which is front and rear. All the insert operations occur through the rear pointer and deletion occurs through the front.
The operations that can be performed on the queue are.

1. Enqueue: Adds an element to the rear(end) of the queue.



2. Dequeue: Removes and returns the element from the front of the queue.

3. Front: Returns the element at the front of the queue without removing it.

4. isEmpty: Checks if the queue is empty.

5. isFull: Checks if the queue is ~~empty~~ full.

6. Size: Returns the number of elements currently in the queue.

For implementing the is Full function you need to check if front is equal to rear - 1. If it is full return 1 and return 0 otherwise.

For implementing the is Empty function you need to check if the rear variable is equal to the size - 1. If it is empty return 1 otherwise return 0;

7. Display: Used to display the elements present in the queue.

Aim: Implement Circular Queue using array.

Program:

```
#include <stdio.h>
#include <stdlib.h>


struct CirQueue{

    int front;
    int rear;
    int size;
    int *arr;

};

int isFull(struct CirQueue* q)
{
    if((q->rear+1)%q->size == q->front)
    {
        return 1;
    }
    else
    {
```

```
        return 0;
    }
}

int isEmpty(struct CirQueue* q)
{
    if(q->front == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void Create(struct CirQueue* q)
{
    printf("Enter the size of the queue: \n");
    scanf("%d", &(q->size));
    q->arr = (int*)malloc((q->size)*(sizeof(int)));
    q->front = -1;
    q->rear = -1;
```

```
    printf("Queue created successfully\n");
}


void Enqueue(struct CirQueue* q)
{
   int n;

   if(isFull(q))
   {
    printf("Queue is Full\n");
   }
   else
   {
       printf("Enter the element you want to enqueue: \n");
       scanf("%d",&n);
       if(isEmpty(q))
       {
        q->front = 0;
       }

       q->rear = (q->rear+1)%q->size;
       q->arr[q->rear] = n;
```

```
  }


}

void Dequeue(struct CirQueue* q)
{
   int val;
   if(isEmpty(q))
   {
      printf("No element for Dequeue\n");
   }
   else
   {
     val = q->arr[q->front];
     if(q->front == q->rear)
     {
        q->front = q->rear = -1;

     }
     else
     {
        q->front = (q->front+1)%q->size;
```

```
    }

    printf("Dequeued element : %d\n", val );
  }



}
void Peek(struct CirQueue *q)
{
  int val;
  if(isEmpty(q))
  {
    printf("No element to peek.\n");
  }
  else
  {
    val = q->arr[q->front];
    printf("Peeked element: %d\n",val);
  }

}
```

```
void Display(struct CirQueue* q)
{

    int i= q->front;
    if(isEmpty(q))
    {
        printf("No element to display.\n");
    }
    else
    {
        printf("Elements: \n");

         while(i!= q->rear)
        {
        printf("%d ",q->arr[i]);
        i = (i+1)%q->size;
        }
        printf("%d ",  q->arr[i]);

   }

}
```

```
int main()
{
  struct CirQueue* q = (struct CirQueue*)malloc(sizeof(struct CirQueue));
  int choice;
  printf(" 1.Create \n 2.Enqueue\n 3.Dequeue \n 4.Display \n 5.Peek\n 6.Exit\n");
  while(1)
  {
     printf("Enter your Choice: \n");
     scanf("%d",&choice);

     switch(choice)
     {
     case 1:
        Create(q);
        break;
     case 2:
        Enqueue(q);
        break;
     case 3:
        Dequeue(q);
        break;
```

```
        case 4:
            Display(q);
            break;
        case 5:
            Peek(q);
            break;
        case 6:
            free(q->arr);
            free(q);
            exit(0);
            break;
        default:
            printf("Invalid choice.\n");
            break;
        }
    }

    return 0;
}
```

Output:

```
 1.Create
 2.Enqueue
 3.Dequeue
 4.Display
 5.Peek
 6.Exit
Enter your Choice:
1
Enter the size of the queue:
4
Queue created successfully
Enter your Choice:
2
Enter the element you want to enqueue:
1
Enter your Choice:
2
Enter the element you want to enqueue:
2
Enter your Choice:
2
Enter the element you want to enqueue:
3
Enter your Choice:
2
Enter the element you want to enqueue:
4
Enter your Choice:
4
Elements:
1 2 3 4 Enter your Choice:
3
Dequeued element : 1
Enter your Choice:
3
Dequeued element : 2
Enter your Choice:
3
Dequeued element : 3
Enter your Choice:
3
Dequeued element : 4
Enter your Choice:
3
No element for Dequeue
Enter your Choice:
2
Enter the element you want to enqueue:
23
```

```
Enter your Choice:
4
Elements:
23 Enter your Choice:
5
Peeked element: 23
Enter your Choice:
6
```

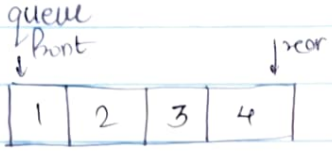
C3-145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

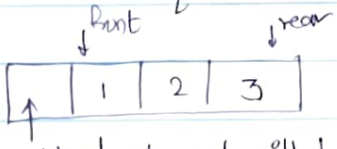
Circular Queue

A circular queue is a data structure that combines the properties of both queues and circular ~~queues~~ buffers. It is used to efficiently manage a fixed size collection of elements, allowing insertion and deletion operations to be performed in a circular manner. The circular nature of the queue means that when the end of the queue is reached, the next element is inserted at the beginning of the queue, effectively forming a circle.

Linear queue



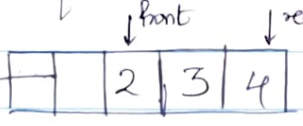
Circular queue



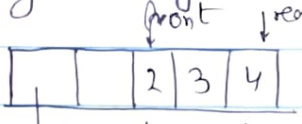
Next element will be inserted here in a circular manner

Why circular queues?

- Circular queues are particulary efficient when you have a fixed-size buffer and want to optimize space usage. In a linear queue, as elements are dequeued the empty spaces accumulate at the front of the queue, leading to inefficient use of memory. Circular queues address this by reusing the space at the beginning of the queue, as elements are dequeued, preventing wastage of memory.



linear queue cannot be used



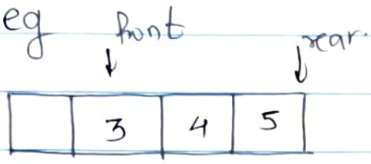
element can be inserted here.

Operations performed on a Circular Queue.

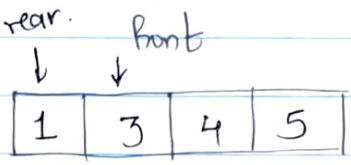
1. Enqueue: To insert an element in a circular queue we cannot normally increment we have to use circular increments to ensure that we use the memory efficiently.

Circular increments → (i+1) % size of queue

For eg

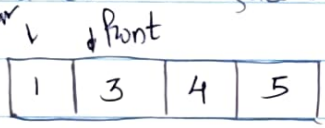


Before enqueuing

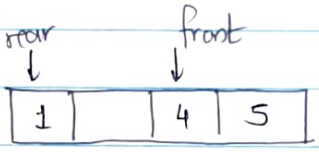


After enqueuing

2. Dequeue: To delete an element we need to circular increment front for a circular Queue.



Before dequeing



3. Display: Used to display the elements of the queue.

4. IsFull: The is full function checks of the circular queue is full or not by checking the following condition that if (rear + 1) == front

5. Is Empty: The is Empty function checks if the circular queue is empty or not by checking if front == rear.

6. Create: The create function takes in the size of queue and allocates the memory.

7. Peek: Prints out the element at the front of the queue

A1
9/4/23

Aim: Implement Singly Linked list ADT.

Program:

```
#include <stdio.h>
#include <stdlib.h>

struct Node
{
   int data;
   struct Node * next;
};

void CreateLinkedList(struct Node * head)
{
   int i =0,n;

   struct Node * current;
   current = head;
  printf("Enter the number of elements in the linked list: ");
   scanf("%d",&n);
   printf("Enter element : ");
   scanf("%d", &head->data);
   head->next = NULL;
   for(i=1;i<n;i++)
```

```
    {
        current->next = (struct Node*)malloc(sizeof(struct Node));
        current = current->next;
        printf("Enter element : ");
        scanf("%d", &(current->data));
        current->next = NULL;
    }

}

void Display(struct Node* head)
{
    int i = 1;
    struct Node *ptr = (struct Node*)malloc(sizeof(struct Node));
    ptr = head;
    while(ptr->next != NULL)
    {
        printf("Element %d = %d \n", i,ptr->data);
        ptr = ptr->next;
        i++;
    }
    printf("Element %d = %d \n" ,i,ptr->data);
}
```

```
struct Node * InsertAtBeginning(struct Node * head)
{
   int data;
   struct Node *ptr = (struct Node*)malloc(sizeof(struct Node));
   printf("Enter the data you want to insert: ");
   scanf("%d", &data);
   ptr->data = data;
   ptr->next = head;
   return ptr;
};

struct Node* InsertAtIndex(struct Node * head)
{
   int ind, data,i=0;
   struct Node *ptr = (struct Node*)malloc(sizeof(struct Node));
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   printf("Enter the data you want to insert and its index position: ");
   scanf("%d%d", &data,&ind);
   p = head;
   ptr->data = data;
   while(i!= ind- 1)
   {
```

```
        p = p->next;
        i++;
    }
    ptr->next = p->next;
    p->next = ptr;
    return head;
};

struct Node * InsertAtEnd(struct Node * head)
{
    int data;
    struct Node *ptr = (struct Node*)malloc(sizeof(struct Node));
    struct Node *p = head;
    printf("Enter the data you want to insert: ");
    scanf("%d", &data);
    ptr ->data = data;
    ptr-> next = NULL;
    while(p->next != NULL)
    {
        p= p->next;
    }
    p->next= ptr;
```

```
    return head;

};

struct Node* DeleteFirst(struct Node* head)
{
    struct Node *ptr =head;
    head = head->next;
    free(ptr);
    return head;

};

struct node * DeleteAtIndex(struct Node * head)
{
    int ind,i=0;
    struct Node *ptr = head;
    struct Node *p;
    printf("Enter the index at which you want to delete: ");
    scanf("%d",&ind);
    while(i!=ind-1)
    {
        ptr = ptr->next;
```

```
        i++;
    }
    p = ptr->next;
    ptr->next= p->next;
    free(p);
    return head;
};

struct Node* DeleteAtEnd(struct Node* head)
{
  struct Node*ptr = head;
  struct Node*p = head->next;

  while(p->next != NULL)
  {
      p = p->next;
      ptr = ptr->next;
  }
  ptr->next = NULL;
  free(p);
  return head;
};
```

```
int main()
{
   int choice;
   struct Node *head = (struct Node*)malloc(sizeof(struct Node));
   printf(" 1. Create List \n 2. Display \n 3.Insert At the Beginning\n 4.Insert at Index \n 5.Insert at End \n 6.Delete the first Node\n 7.Delete node at Index \n 8.Delete at End \n 9.Exit \n");

  while(1)
  {
   printf("Enter your choice: ");
   scanf("%d", &choice);

   switch(choice)
   {
   case 1:
      CreateLinkedList(head);
      break;
   case 2:
      Display(head);
      break;
   case 3:
```

```
        head = InsertAtBeginning(head);
        break;
    case 4:
        head = InsertAtIndex(head);
        break;
    case 5:
        head = InsertAtEnd(head);
        break;
    case 6:
        head = DeleteFirst(head);
        break;
    case 7:
        head = DeleteAtIndex(head);
        break;
    case 8:
        head = DeleteAtEnd(head);
        break;
    case 9:
        exit(0);
        break;
    }
}
```

```
    return 0;
}
```

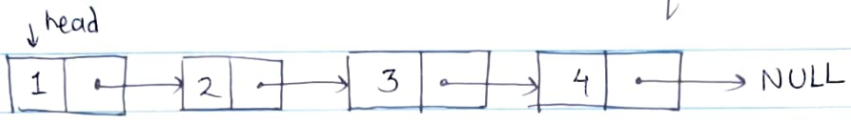
Output:

```
 1. Create List
 2. Display
 3.Insert At the Beginning
 4.Insert at Index
 5.Insert at End
 6.Delete the first Node
 7.Delete node at Index
 8.Delete at End
 9.Exit
Enter your choice: 1
Enter the number of elements in the linked list: 4
Enter element : 1
Enter element : 2
Enter element : 3
Enter element : 4
Enter your choice: 2
Element 1 = 1
Element 2 = 2
Element 3 = 3
Element 4 = 4
Enter your choice: 3
Enter the data you want to insert: 0
Enter your choice: 4
Enter the data you want to insert and its index position: 45 3
Enter your choice: 5
Enter the data you want to insert: 12
Enter your choice: 2
Element 1 = 0
Element 2 = 1
Element 3 = 2
Element 4 = 45
Element 5 = 3
Element 6 = 4
Element 7 = 12
Enter your choice: 8
Enter your choice: 7
Enter the index at which you want to delete: 2
Enter your choice: 2
Element 1 = 0
Element 2 = 1
Element 3 = 45
Element 4 = 3
Element 5 = 4
Enter your choice: 9

Process returned 0 (0x0)   execution time : 177.624 s
Press any key to continue.
```

Linked list

A linked list is a fundamental data structure used in computer science and programming to store and manage collections of data. It is a linear data structure where elements called nodes, are linked together through pointers and references. Each node of a linked list contains

1. Data: This is the actual value or information stored in a node.
2. Pointer: This is a reference to the next node in the sequence



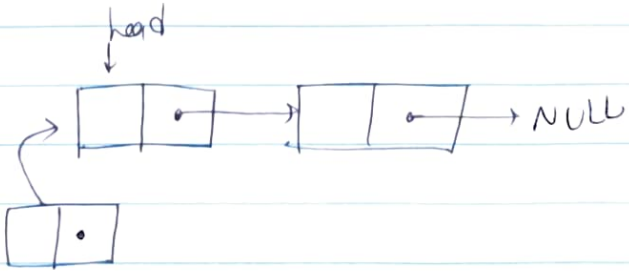
- Advantages of Linked list
  1. Linked lists can easily grow or shrink in size making them suitable for situations where the size of data collection is unknown in advance.
  2. Linked list do not require contiguous memory allocation.

Some basic operations performed on a linked list are.

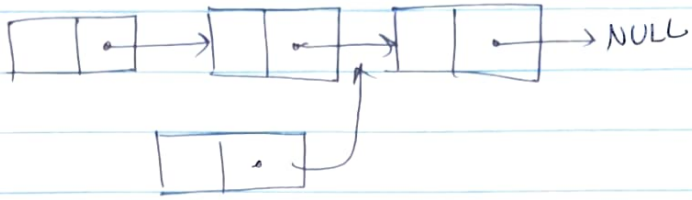
1. Create Linked list: Used to create an empty linked list

2. Display: Used to display the elements present in the linked list

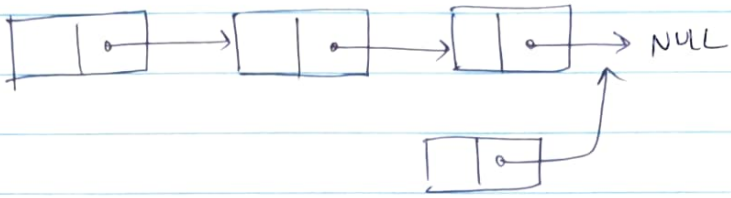
3. Insert: Insersion in linked list can occur in three ways.
   - At the Beginning:



- In the middle :-



- At the end :



4. Delete : Deletion in a linked list can occur in three ways.
   - At the beginning - where the first node is deleted
   - In the middle - where a node at any random index position is deleted.
   - At the end - where the final node is deleted

5. Size : Returns the number of elements in the list

6. IsEmpty : Checks if the linked list is empty.

4/9/23

Aim: Implement Stack / Queue using linked list.

Program:

Stack using Linked

```
#include <stdio.h>
#include <stdlib.h>

struct Node* top = NULL;


struct Node
{
   int data;
   struct Node* next;
};

int isFull(struct Node* n)
{
   if(n==NULL)
      return 1;
   else
      return 0;

}
```

```
int isEmpty()
{
   if(top==NULL)
      return 1;
   else
      return 0;
}

void push()
{
   struct Node* n = (struct Node*)malloc(sizeof(struct Node));
   int val;
   if(isFull(n))
   {
      printf("Stack Overflow.\n");
   }
   else
   {
      printf("Enter the element to be pushed: \n");
      scanf("%d",&val);
      n->data = val;
      if(top == NULL)
```

```
        {
            top = n;
            n->next = NULL;
        }
        else
        {
          n->next = top;
          top = n;
        }
    }

}

void pop()
{
    if(isEmpty())
    {
        printf("Stack Underflow\n");
    }
    else
    {
        struct Node* ptr = top;
        printf("Popped element: %d\n", ptr->data);
```

```
        top = top->next;
        free(ptr);
    }

}

void Peek()
{
    if(isEmpty())
    {
        printf("No element to Peek\n");
    }
    else
    {
        printf("Peeked element: %d\n", top->data);
    }
}

void Display()
{

    struct Node* ptr = top;
    if(isEmpty())
```

```
    {
        printf("No element to display.\n");
    }
    else
    {
        printf("Elements: \n");
    while(ptr->next  != NULL)
    {
        printf("%d\n",ptr->data );

        ptr = ptr->next;
    }
     printf("%d\n", ptr->data);

    }
}

int main()
{
   int choice;
   while(1)
   {
      printf(" 1.Push\n 2.Pop \n 3.Peek\n 4.Display\n 5.Exit\n");
```

```
    printf("Enter your choice: \n");
    scanf("%d", &choice);
    switch(choice)
    {
    case 1:
     push();
     break;
    case 2:
     pop();
     break;
    case 3:
     Peek();
     break;
    case 4:
     Display();
     break;
    case 5:
     exit(1);
     break;

    }
}
```

```
    return 0;
}
```

Queue Using Linked

```
#include <stdio.h>
#include <stdlib.h>

struct Node* f = NULL;
struct Node* r =NULL;

struct Node
{
    int data;
    struct Node* next;
};


int isEmpty(struct Node* f)
{
    if(f == NULL)
    {
        return 1;
```

```
    }
    else
    {
        return 0;
    }

}
int isFull(struct Node * N)
{
    if(N==NULL)
        return 1;
    else
        return 0;
}

void enqueue(){
    struct Node *n = (struct Node *) malloc(sizeof(struct Node));
    int val;
    printf("Enter element to be enqueued: \n");
    scanf("%d", &val);
    if(n==NULL){
        printf("Queue is Full\n");
    }
```

```
    else{
        n->data = val;
        n->next = NULL;
        if(f==NULL){
            f=r=n;
        }
        else{
            r->next = n;
            r=n;
        }
    }
}


void Dequeue()
{
    struct Node* ptr = f;
    if(isEmpty(f))
    {
        printf("No element for dequeue");
    }
    else
    {
```

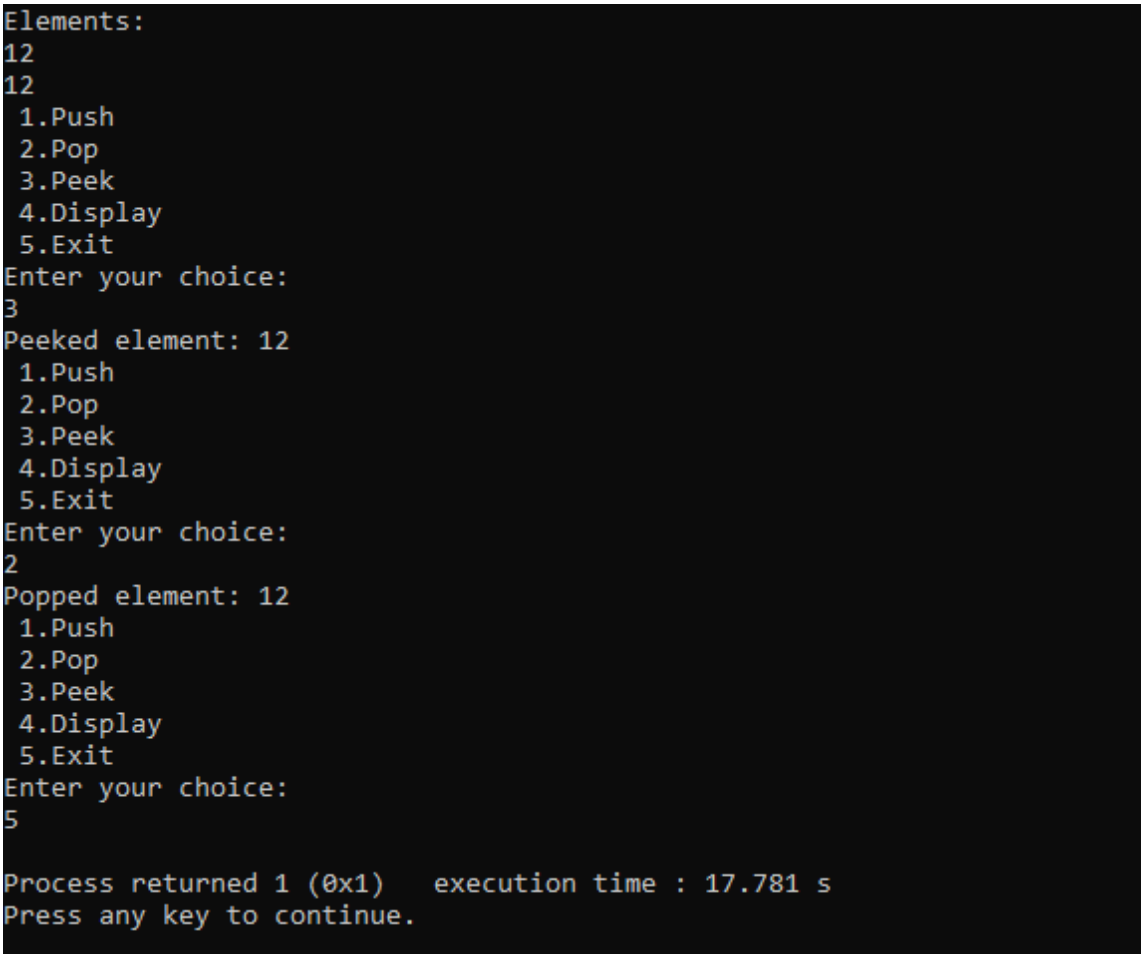
```
      f= f->next;
      printf("Dequeued element: %d\n", ptr->data);
      free(ptr);
    }
}

void Peek()
{
    printf("Peeked element: %d\n", f->data);
}

void Display(struct Node*ptr)
{
    int i =0;
    while(ptr->next != NULL)
    {
        printf("Element %d: %d\n", i+1, ptr->data);
        ptr= ptr->next;
        i++;
    }
    printf("Element %d: %d\n", i+1, ptr->data);
}
```

```
int main()
{

   int choice;
   while(1)
   {
      printf(" 1.Enqueue \n 2.Dequeue \n 3.Peek \n 4.Display \n
5.Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice)
      {
      case 1:
         enqueue();
         break;
      case 2:
         Dequeue();
         break;
      case 3:
         Peek();
         break;
      case 4:
```

```
            Display(f);
            break;
        case 5:
            exit(1);
            break;
        }
    }


    return 0;
}
```

Output:

Stack Using Linked List



Q using linked list

```
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 1
Enter element to be enqueued:
1
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 1
Enter element to be enqueued:
2
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 1
Enter element to be enqueued:
3
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 4
Element 1: 1
Element 2: 2
Element 3: 3
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 3
Peeked element: 1
 1.Enqueue
 2.Dequeue
 3.Peek
 4.Display
 5.Exit
Enter your choice: 2
Dequeued element: 1
```

C3-145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

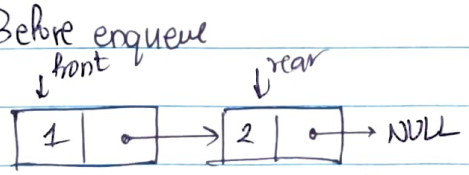
Queue using Linked list.

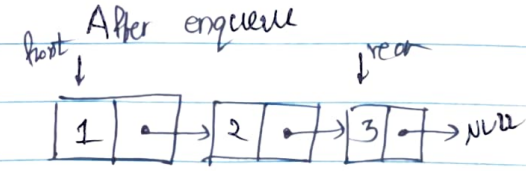
- Node definition.
  - First define a structure for the nodes in the linked list.

```
struct Node{
        int data;
        struct Node *next;
};
```

- Initialization
  - Allocate memory for the linked list.

- Enqueue:
  - To enqueue(add) an element to the queue:
    - Create a new node with the data you want to enqueue.
    - If the linked list is empty, set the new node as the front and rear of the queue.
    - If the linked list is not empty, set the 'next' of the current rear node to point to the new node and update the rear pointer to the new node.

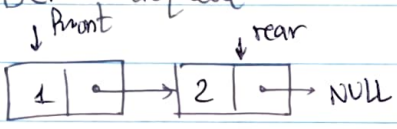




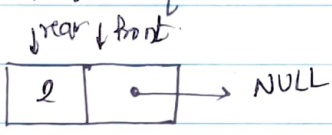
Here front and rear are pointers of type struct Node.

- Dequeue
  - To dequeue (remove) an element from the queue
    - Check if the queue is empty. If it is, return an error or a special value (depending on program.).
    - If the queue is not empty, remove the node where the front pointer is pointer towards.
    - Update the front pointer to point to the next node in the queue.

Before dequeue



After dequeue



- Peek: To print out the element pointed by the front pointer.
- is Empty : If the front pointer is null then the queue is empty.
- is Full : If the newly allocated node still points to NULL then all the memory is full.

A
4/11/23

Stack using Linked list

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. When stack is implemented using a linked list, you create a linked list where each element holds a reference to the next element in the stack

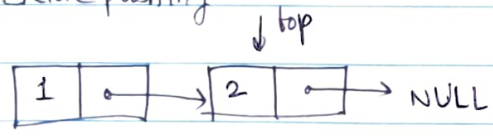
- Node structure : In a linked-list based stack, each element contains two parts data and a pointer to next node

```
struct Node{
    int data;
  struct Node * next;
};
```

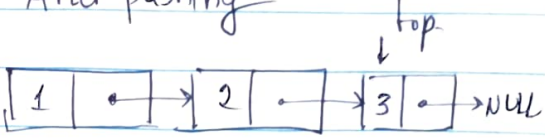
- Unlike a typical linked list where you might only maintain a reference to the head, in a stack, you maintain a reference to the top element.

- Push (insertion): To add an element to the stack, you create a new node and place it at the top of the linked list. You update the top reference to the newly added node

Before pushing

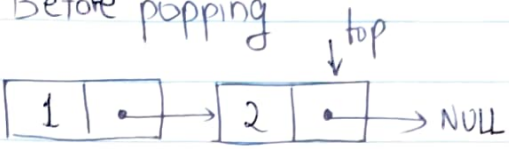


After pushing

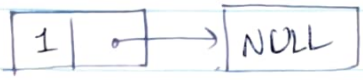


- Pop(deletion) : To remove an element from the stack, you remove the element pointed to by the top reference. After removing the top element, you update the top reference to point to the next element in the stack.

Before popping



After popping



- Peek - Return the item at the top of the stack without removing it
- size (): Return no of elements in the stack.

1

A.3

Q.2

11/9/23

Aim: Implement Binary Search tree ADT using Linked List

Program:

```
#include <stdio.h>
#include <stdlib.h>

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

struct node *Insert(struct node *root)
{
   int data;
   printf("Enter the data to be inserted: ");
   scanf("%d", &data);

   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   if (newNode == NULL)
   {
      printf("Memory allocation failed.\n");
      exit(1);
```

```
    }

    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;

    struct node *current = root;
    struct node *prev = NULL;

    while (current != NULL)
    {
        prev = current;
        if (data == current->data)
        {
            printf("Duplicate values cannot be inserted in a Binary search
tree\n");
            free(newNode);
            return root;
        }
        else if (data > current->data)
        {
            current = current->right;
        }
```

```
        else
        {
            current = current->left;
        }
    }

    if (prev == NULL)
    {
        root = newNode;
    }
    else if (data > prev->data)
    {
        prev->right = newNode;
    }
    else
    {
        prev->left = newNode;
    }
    return root;
}

struct node *Create()
{
```

```
int n, i, nums[50];
struct node *root = NULL;
struct node *current = NULL;
struct node *newNode = NULL;
struct node *prev = NULL;

printf("Enter the no of elements to be inserted: \n");
scanf("%d", &n);
printf("Enter %d elemnts: ", n);

for (i = 0; i < n; i++)
{
   scanf("%d", &nums[i]);
}

for (i = 0; i < n; i++)
{
   newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = nums[i];
   newNode->left = NULL;
   newNode->right = NULL;

   current = root;
```

```
    while (current != NULL)
    {
        prev = current;
        if (nums[i] == current->data)
        {
            printf("Duplicate values cannot be inserted in a Binary search tree\n");
            free(newNode);
            return root;
        }
        else if (nums[i] > current->data)
        {
            current = current->right;
        }
        else
        {
            current = current->left;
        }
    }

    if (prev == NULL)
    {
```

```
            root = newNode;
        }
        else if (nums[i] > prev->data)
        {
            prev->right = newNode;
        }
        else
        {
            prev->left = newNode;
        }
    }

    return root;
}

struct node *Search(struct node *ptr, int key)
{
    if (ptr == NULL)
    {
        printf("No elements found in the tree\n");
        return NULL;
    }
    else if (key == ptr->data)
```

```
    {
        return ptr;
    }
    else if (key > ptr->data)
    {
        return Search(ptr->right, key);
    }
    else
    {
        return Search(ptr->left, key);
    }
}

void isFound(struct node *ptr)
{
    if (ptr != NULL)
    {
        printf("Found %d\n", ptr->data);
    }
    else
    {
        printf("Element not found\n");
```

```
    }
}

void inOrder(struct node *root)
{
    if (root != NULL)
    {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
    printf("\n");
}

struct node *inOrderPredecessor(struct node *root)
{
    root = root->left;
    while (root->right != NULL)
    {
        root = root->right;
    }

    return root;
```

```
}

struct node *Delete(struct node *root, int value)
{

   struct node *iPre;
   if (root == NULL)
   {
      return NULL;
   }
   if (root->left == NULL && root->right == NULL)
   {
      free(root);
      return NULL;
   }

   if (value < root->data)
   {
      root->left = Delete(root->left, value);
   }
   else if (value > root->data)
   {
      root->right = Delete(root->right, value);
```

```
    }
    else
    {
        iPre = inOrderPredecessor(root);
        root->data = iPre->data;
        root->left = Delete(root->left, iPre->data);
    }
    return root;
}

int main()
{
    struct node *root = NULL;
    int choice, key;
    while (1)
    {
        printf(" 1. Create \n 2. Search \n 3. Insert \n 4. Delete \n 5. Inorder
\n 6. Exit\n");
        printf("Enter your choice: \n");
        scanf("%d", &choice);

        switch (choice)
        {
```

```
case 1:
   if (root != NULL)
   {
      free(root);
   }
   root = Create();
   break;
case 2:
   if (root == NULL)
   {
      printf("Tree is empty. Please create a tree first.\n");
      break;
   }
   printf("Enter the element to be searched: \n");
   scanf("%d", &key);
   struct node *n = Search(root, key);
   isFound(n);
   break;
case 3:
   if (root == NULL)
   {
      printf("Tree is empty. Please create a tree first.\n");
```

```
        break;
    }
    root = Insert(root);
    break;
case 4:
    if (root == NULL)
    {
        printf("Tree is empty. Please create a tree first.\n");
        break;
    }
    printf("Enter the element to be deleted: \n");
    scanf("%d", &key);
    root = Delete(root, key);
    break;
case 5:
    inOrder(root);
    break;
case 6:
    free(root);
    exit(0);
default:
    printf("Invalid choice\n");
```

```
            break;
        }
    }

    return 0;
}
```

Output:

```
 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
Enter your choice:
1
Enter the no of elements to be inserted:
5
Enter 5 elemnts: 12 23
123 22 1
 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
Enter your choice:
5

1

12
22

23
123



 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
Enter your choice:
2
Enter the element to be searched:
22
Found 22
 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
```

```
Enter your choice:
4
Enter the element to be deleted:
23
 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
Enter your choice:
5

1

12
22
123



 1. Create
 2. Search
 3. Insert
 4. Delete
 5. Inorder
 6. Exit
Enter your choice:
6

Process returned 0 (0x0)   execution time : 40.413 s
Press any key to continue.
```

C3 - 145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Binary Search tree

. Binary search trees (BSTs) are a fundamental data structure in computer sciena and are a type of binary tree with certain properties that make them useful for efficient searching and sorting operations.

. A binary tree is a hierarchical data structure consisting of nodes, where each node has at most two children, refered to as the left child and the right child.

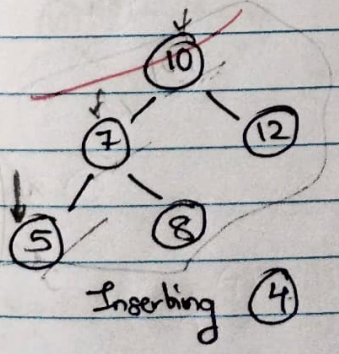
. In a BST, for each node:
  . All nodes in its left subtree have values less than the node's value
  . All nodes in right subtree have values greater than the node's value

This property ensures that the tree is sorted, making searching, insertion, deletion operations efficient with a time complexity of O(log n).

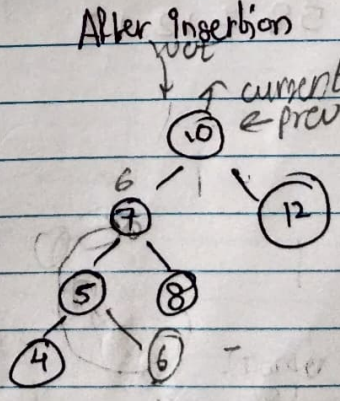
. Trees are used in various applications, including database indexing, hierarchical data representation, parsing

Operations associated with the Binary Tree
. Insertion: The ability to insert a new node into the tree, respecting the rules of a binary tree

Before insertion



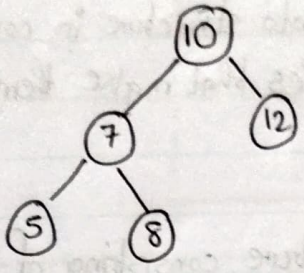
After insertion



Inorder: The order to calculate the inorder of a BST is

left subtree elements → root → right subtree elements

Consider the tree



The inorder is 5 7 8 10 12

Preorder: The order to calculate the preorder of a BST is

root → left subtree elements → right subtree elements

Hence the preorder of the previous tree is

10 7 5 8 12

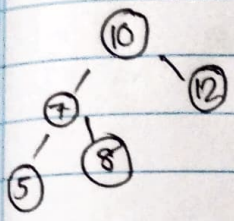
Postorder: The order to calculate the postorder of a BST is

left subtree elements → right subtree elements → root

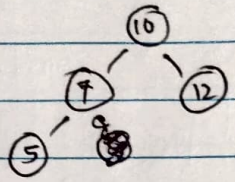
Hence the postorder of the previous tree is

5 8 7 12 10

- Deletion: The ability to remove a specific node from the tree while maintaining the binary tree properties.



Before Deletion



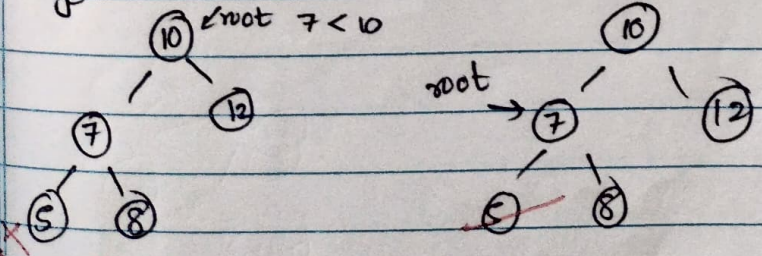
After Deleting (8)

If you want to delete the node (7). The question arises that which node out of (5) & (8) should take (7)'s place. To calculate this we need to calculate the inorder of the tree. And we can replace the (7) node with it's inorder successor or inorder predecessor.
For the above tree its inorder is

5 7 8 10 12

Now from the inorder the inorder successor and predecessor of 7 are 5 and 8 respectively hence we can replace (7) by (5) and (8).

- Searching: The ability to search for a specific value within the tree efficiently, typically using a binary search. For searching you would start from root comparing if the value to be searched is greater or smaller than the root value, if it is greater you set the root to the right subtree and if it is smaller then ur root will be set to left subtree



For searching (7)

Element found

Aim: Implement graph traversal techniques a) BFS b)DFS

Program:

BFS

```
#include <stdio.h>
#include <stdlib.h>


struct Queue{

    int front;
    int rear;
    int size;
    int *arr;

};

int isFull(struct Queue* q)
{
    if(q->rear == q->size-1)
    {
        return 1;
    }
    else
```

```
    {
        return 0;
    }
}

int isEmpty(struct Queue* q)
{
    if(q->front == q->rear)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void Create(struct Queue* q)
{
    printf("Enter the size of the queue: \n");
    scanf("%d", &(q->size));
    q->arr = (int*)malloc((q->size)*(sizeof(int)));
    q->front = -1;
```

```
    q->rear = -1;
    printf("Queue created successfully\n");
}


void Enqueue(struct Queue* q, int data)
{


   if(isFull(q))
   {
    printf("Queue is Full\n");
   }
   else
   {

      q->rear++;

      q->arr[q->rear] = data;
   }

}

int Dequeue(struct Queue* q)
```

```
{
    int val=-1;
    if(isEmpty(q))
    {
        printf("No element for Dequeue\n");
        return -1;

    }
    else
    {
        q->front++;
        val = q->arr[q->front];
        return val;
    }

}
void Display(struct Queue* q)
{

    int i=q->front+1;
    while(i<= q->rear)
    {
        printf("Element %d: %d\n",i+1,q->arr[i]);
```

```
        i++;

    }

}

int main()
{
  struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
  q->size = 10;
  q->front = -1;
  q->rear = -1;
  q->arr = (int*)malloc((q->size)* sizeof(int));
  int visited[7] = {0,0,0,0,0,0,0};
  int a[7][7]={
    {0,1,1,1,0,0,0},
    {1,0,1,1,0,0,0},
    {1,1,0,1,1,0,0},
    {1,0,1,0,1,0,0},
    {0,0,1,1,0,1,1},
    {0,0,0,0,1,0,0},
    {0,0,0,0,1,0,0}
  };
```

```
  int i =0,n,j;
  printf("%d", i);
  visited[i] = 1;
  Enqueue(q, i);

  while(!isEmpty(q))
  {
    n = Dequeue(q);
    for(j = 0 ; j < 7; j++)
    {
      if(a[n][j] == 1 && visited[j] == 0)
      {
        printf("%d", j);
        visited[j] = 1;
        Enqueue(q,j);
      }
    }
  }


}
```

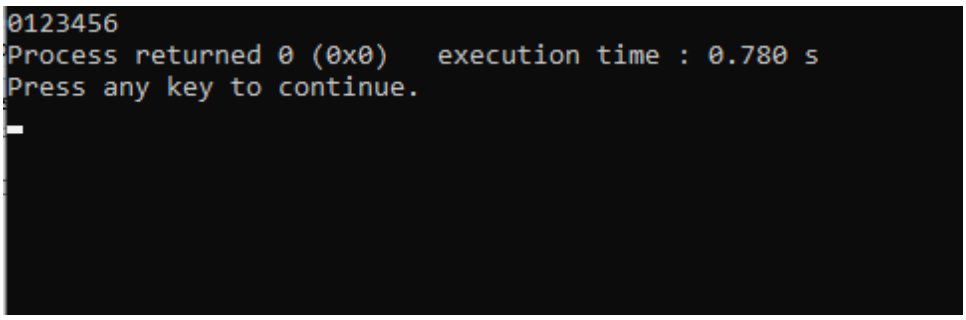
DFS

```
#include<stdio.h>
#include<stdlib.h>

int visited[7]={0,0,0,0,0,0,0};

int A[7][7]={
   {0,1,1,1,0,0,0},
   {0,0,1,0,0,0,0},
   {1,1,0,1,1,0,0},
   {1,0,1,0,1,0,0},
   {0,0,1,1,0,1,1},
   {0,0,0,0,1,0,0},
   {0,0,0,0,1,0,0}
};

void DFS(int i){
   int j;
   printf("%d",i);
   visited[i]=1;
   for(int j=0;j<7;j++){
      if(A[i][j]==1 && !visited[j]){
         DFS(j);
```

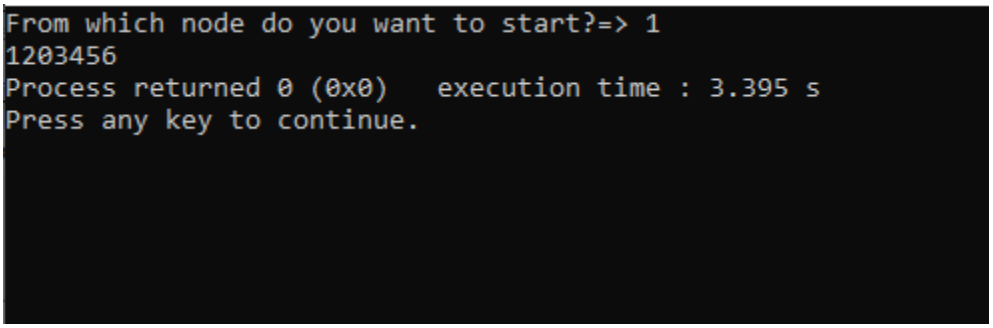
```
        }
    }
}

int main()
{
    int start;
    printf("From which node do you want to start?=> ");
    scanf("%d",&start);
    DFS(start);
    return 0;
}
```

Output:

BFS



DFS


From which node do you want to start?=> 1
1203456
Process returned 0 (0x0)   execution time : 3.395 s
Press any key to continue.

Experiment 9

C3-145

- Breadth - First Search (BFS):
- BFS is a graph traversal algorithm that explores all the vertices at the current depth before moving on to vertices at the next depth.
- It uses a queue to keep track of nodes to visit, ensuring that you explore nodes level by level
- BFS is typically used to find the shortest path in unweighted graphs, as it visits nodes in order of their distance from the source node.

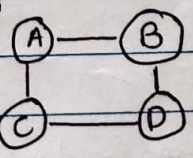
Properties:

- BFS can also be used to check the connectivity of a graph and identify connected components
- It is generally more memory-intensive than DFS due to the queue data structure.

Usage

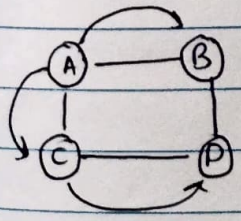
- BFS is suitable for finding the shortest path in an unweighted graph.
- It is used in network routing algorithms, web crawling etc.

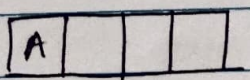
Consider the graph.



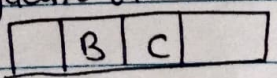
If we start BFS from node 'A', it explores nodes level by level. First it visits 'A', then 'B' and 'C' and then finally 'D'

Here the BFS is A B C D

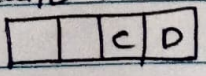


Queue 

Add adjacent of A in Queue & ∵ A is visited pop A



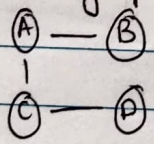
Add adjacent of B in Queue & ∵ B is visited pop B



Pop all elements hence BFS is A B C D.

- Depth First Search (DFS):
  - DFS is a graph traversal algorithm that explores as far as possible along a b before backtracking.
  - It uses a stack (or recursion) to keep track of the nodes to visit, going deep branch before exploring other branches.
  - DFS is useful for tasks like topological sorting, cycle detection, and exploring in a graph.
- Properties:
  - DFS might not find the shortest path in a graph, as it explores one bran deeply before considering other branches.
  - DFS can be more memory-efficient than BFS when implemented with recursion, as it to remember as much state.
- Usage
  - DFS is useful for tasks like topological sorting, cycle detection and exploring a

Consider the same graph :



- If we start DFS from node 'A', it explores as deep as possible before It will visit nodes in the order 'A' → 'B' → 'D' → 'C'.
  Hence the DFS will be A B D C.

16/10/23

Aim: Implement Hash table

Program:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int H[10]={1}, element[10] = {0};
  int i, n, eles, inc;
  printf("Enter the number of elements in the table: \n");
  scanf("%d", &eles);
  printf("Enter the elements that should be added in the table: \n");
  for (i = 0; i < eles; i++)
  {
    scanf("%d", &n);
    inc = n % 10;
    if (element[inc] == 0)
    {
      H[inc] = n;
      element[inc] = 1;
    }
    else
```

```
    {
      while (element[inc] != 0 && inc < 10)
      {
        inc++;
      }
      if (inc < 10) // Check if inc is within array bounds
      {
        H[inc] = n;
        element[inc] = 1;
      }
    }
  }

  for (i = 0; i < 10; i++)
  {
    printf("%d\n", H[i]);
  }
}
```

Output:

```
Enter the number of elements in the table:
4
Enter the elements that should be added in the table:
2
22
23
24
Hash Table:
0
0
2
22
23
24
0
0
0
0

Process returned 0 (0x0)   execution time : 5.030 s
Press any key to continue.
```

Hashing

C3-145

A hash table is a data structure which stores the data in the form of key value pair using hash-function, the position of the data to be calculated. While searching, search element is also hashed using hash function to go to the hash key. The corresponding hash table entry is checked. If the values are present then the search is 'successfull' or its a 'faliure'. A ~~collect~~ collision occurs when two or more keys are hashed to the same hash key. Suppose x_1 and x_2 are the keys, then there is a collision if $h(x_1) = h(x_2)$. A hash table can be implemented by using two dimensional array can be implemented and that as the total number of keys, however. This consumes a lot of memory.

```
for (int i = 0; i<10; i++)
    {
        hk = x[i] %size;
        h[hk][count[hk]] = x[i];
        count[hk]++
    }
```

Also a given key can be first hashed as

"hk = key % size and then searched as.

```
for (i=0; i<n; i++) {
        if ( h[hx][i] == key)
        return 1;
}
return 0;
```

Example

Hash table :-

h(x)

Index	99	33	23	44	56	43	19
0							19*
1							
2							
3		33	33	33	33	33	33
4			23*	23	23	23	23
5				44*	44	44	44
6					56*	56	56
7						43*	43
8							
9	99	99	99	99	99	99	99

23 | 10 | 23

Roll no: 2203145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

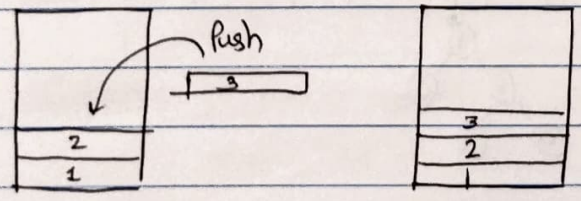
Assignment 1

1. Write a short note on abstraction. Explain stack ADT.

→ 1) A big program is never written as a monolithic piece of program instead broken down in smaller modules.
2) The main program utilizes services of functions appearing at level 1.
3) Similarly a function at level 1 utilizes serves of functions written at level 2.
4) Main program uses the services of next function without knowing their implementation details. A level of abstraction is created.
5) When an abstraction is created at any level our concern is limited to 'what it can do' and 'how it is done'.

Stack ADT.

1) A stack ADT is a fundamental data structure that represents a collection of elements with two primary operations: push and pop. It follows the Last-In-First-Out (LIFO) principle meaning that the last element added to the stack is the first one to be removed.
2) Push: This operation is used to add an element to the top of the stack. It involves placing an element onto the top of the stack. When you push an element onto the stack, it becomes the new top element and the previous elements are below it

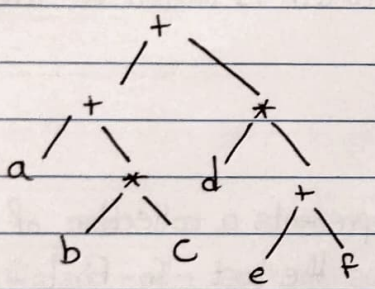


3) Pop: This operation is used to remove and return the top element of the stack. It involves taking the top element off the stack and returning its value. After a pop operation, the element that was below the removed element becomes the new top of the stack.
4) Peek is used to view the topmost element in the stack.
5) Display is used to show all the elements in the stack.

2. Explain expression trees with example.

→ An expression tree, also known as a parse tree, is a binary tree used to represent mathematical expressions in a way that preserves their hierarchical structure. Each node in the tree represents an operator or an operand, and the structure of the tree reflects the order of operations in the expression. Expression trees are a useful data structure for evaluating and manipulating mathematical expressions.

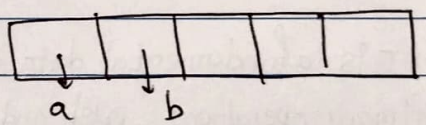
Consider the expression a + (b*c)+d * (e+f)



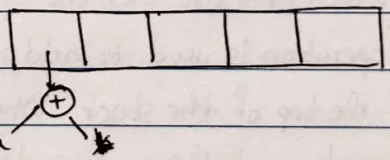
Constructing an expression tree using postfix expression

a b + c +

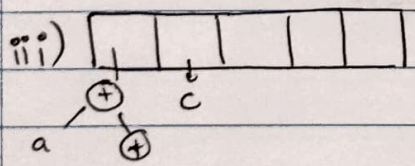
i)



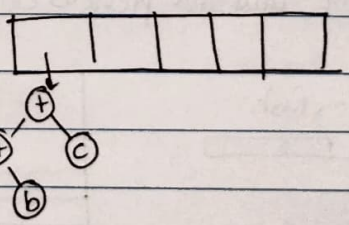
ii)



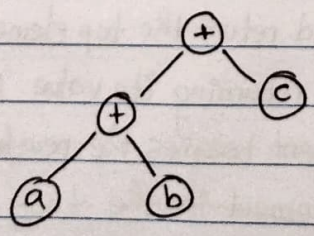
iii)



iv)



Hence the final expression tree



5. WAP to implement priority queue
   1) Insert 2) Delete 3) Search 4) Display 0) Exit

→
```
#include <stdio.h>
#define max 5
int front = -1;
int rear = -1;
int ch, i;
int q[max];
void full()
{
  if (rear == max-1)
    printf("Full");
}
void empty()
{
  if (front == -1 || rear == -1)
    printf("Underflow");
}
void insertion_sort()
{
  if (front == -1)
    if (rear == max-1)
      printf("Full");
    else {
      printf("Enter value to enqueue: ");
      scanf("%d", &q[i]);
      if (front == -1 && rear == -1)
        front = 0;
        rear = 0; }
    else
```

```
{ rear= rear+1; }
   a[rear]= val ;
  }
void normaldequeue ()
   { if (rear ==-1)
     printf("empty");
    else
     { val = a[front];
      if (front ==rear)
       { front =-1;
          rear==1; }
      else { front++ ; }
      printf ("deleted val is %d", val); }
    }
void complexenqueue ()
   { int i;
     printf(" Enter the value: ");
     scanf ("%d", &val);
    if (rear== max-1) "
          printf (" full");
    elseif ( rear == -1 && front == -1)
      { front = 0;
        rear =0;
        a[0] = val }
    else {
      for ( i= front; i<= rear ; i++)
         { if ( val> a[i] )
              break;
           For (j=rear ; j<= i ; j--)
```

```
{ a[j+1] = a[j] }
  a[i] = val ; }
    rear = rear +1 ;
    }
    }
void complex dequeue ()
  { int j;
  if (rear == -1)
    printf ("Empty");
  else if (front == rear)
  { val = a[front]
    front = rear = -1 ;
    printf ( "deleted val %d ", val) ;
  else
    { int pos = 0 ;  rear=a[0];
    for (i=0 ; i<= rear ; i++)
    { if (a[i] >max)
      { pos = i ;
        max = a[i] ; }
      }
    for (j= pos ; j<= rear -1 ; j++)
      { a[j] = a[j+1] ; }
        rear= rear -1 ;
    printf (" deleted val % d", val);
    }
  }
void display ()
  { if (rear = -1)
    printf ("Empty") ;
    else
```

```
for (i= front ; i<=rear ; i++)
   printf ("%d", a[i]);
}
void search()
{ printf("Enter the val to be searched : ");
   scanf("%d", &val);
   for (i=front ; i<=rear ; i++)
   { if (val == a[i])
        printf ("Element found");
      else
        printf("not found");
}     }
void main()
{ int c;
  while ()
{ printf("Enter 1. normal enqueue 2. normal dequeue 3. complex enqueue, 4. complex dequeue 5. display
      6. search 7. exit");
 scanf("%d", &c);
switch (c)
{ case1:
  normalenqueue();
   break;
 case 2:
  normal dequeue();
   break;
 case 3:
  complex enqueue();
   break;
case 6:
  complex dequeue();
   break;
case 5:
 display();
 break;
case 6:
 search();
 break;
case 7;
 exit(0);
default :
  printf("Invalid choice");
}
}
```

4. Write a short note on hashing technique and its significance

→ Hashing is a fundamental technique in computer science used for efficient data storage retrieval and manipulation. It involves the transformation of data into a fixed size value called a hash code or hash value. These hash values are generated by applying a hash function to the original data. Hashing techniques are significant in various fields of computer science, including data structures, databases, cryptography and more.

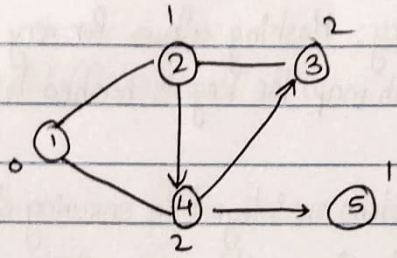
Significance of Hashing Techniques:

1. Data Retrieval Efficiency: Hashing allows for very fast data retrieval. When data is stored in a hash table or hash map, the key is hashed to find the exact location of the value.
2. Data Security: Hash functions are integral in ensuring data security. Cryptographic hash functions are designed to be irreversible and produce unique hash values for different inputs.
3. Load Balancing: Hashing can be used to distribute data across multiple servers or storage devices evenly. This is particularly crucial in large-scale systems.
4. Caching: Hashing is used in caching to quickly check whether a specific item is in the cache or not. Caches like the common used LRU and LFU caches.
5. Blockchain and Cryptocurrency: Blockchain technologies, which underlie cryptocurrencies like Bitcoin, heavily rely on hashing for creating secure and tamper-evident blocks.
6. Efficient searching and Retrieval: Hashing is used in data structures such as hash tables, which offer constant-time average case complexity for inserting, searching, and deleting data. This makes hashing indispensable for creating efficient databases and dictionaries.

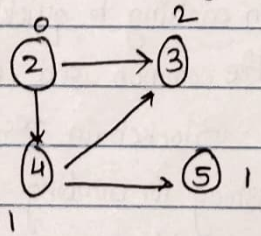
5. Explain on topological sorting with example

→ i) It is a linear ordering of graph vertices such that for every directed edge uv from vertex u to vertex v, comes before v in the order.
2) It is applicable on DAG (Direct cyclic graph).
3) It has linear running time complexity.
4) Topological sort is not unique.

eg:



Step 1 : write indegree of all the vertices and select v with a indegree
step 2: Delete edge of the one vertex having 0 in and update graph and display the deleted edge vertex



Repeat step 1 & 2, until all the elements are displayed

ii) 1, 2
iii) 1, 2, 4
iv) 1, 2, 4, 3
v) 1, 2, 4, 3, 5
or
iv) 1, 2, 4, 5
v) 1, 2, 4, 5, 3

23/10/23

Roll no: 2203145

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Assignment 2

1. Explain AVL tree with example.

→ 1. An AVL tree is a binary search tree, which means that for each node:
- All nodes in the left subtree have value less than the node's value,
- All nodes in the right subtree here have values greater than the node's value.

2. The balance factor of a node is defined as the height of its right subtree minus the height of its left subtree. The balance factor always -1, 0, or 1 for any node in an AVL tree.

3. The height of an AVL tree is guranteed to be logarithmic, resulting in efficient operation.

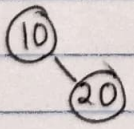
Insertion in an AVL Tree:

1. Perform a standard binary search tree insert as you would in a regular binary search tree.
2. After inserting the new node, work your way up from the newly inserted node to the root, updating the balance factor and checking if the tree is still balanced
3. If any point, you find a node with a balance factor of -2, or 2 it's unbalanced. To restore balance, perform one or more of the following rotations:
- Left - left (LL)
- Left - right (LR)
- Right right (RR)
- Right left (RL)

Lets say we want to insert the values 10, 20, 30, 40 and 50 into an empty AVL tree.

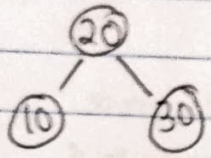
1. Insert 10:

Insert 20:

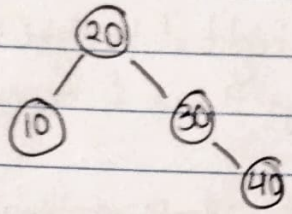


Insert 30:

RR rotation

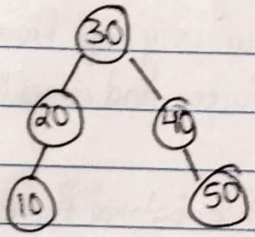


Insert 40:



Insert 50:

RR rotation

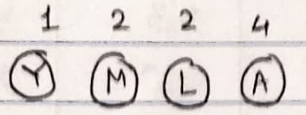


After inserting all the elements, the AVL tree remains balanced with the height difference between the left and right subtrees of any node being at most 1.

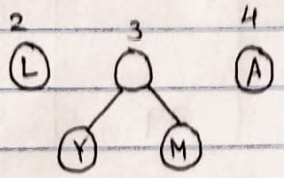
2. Apply Huffman coding for the word 'MALAYALAM'. Give the Huffman code for each symbol

Data	Weight
M	2
A	4
L	2
Y	1

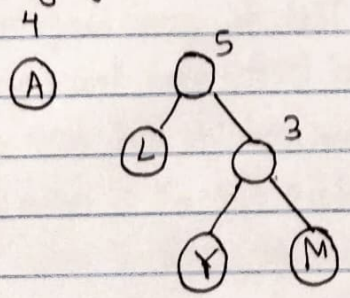
Step 1: Initial forest of trees



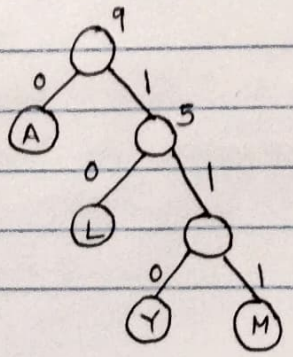
Step 2: Merging Y and M



Step 3: Merging L and (Y, M)



Step 4: Merging A and (L, Y, M)



Step 5:

Data	Huffman code
A	0
L	10
Y	110
M	111

3. Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique: 99, 33, 23, 44, 56, 43, 19.

→

Linear probing:

	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0							19*
1							
2							
3		33	33	33	33	33	33
4			23*	23*	23*	23*	23*
5				44*	44*	44*	44*
6					56	56	56
7						43*	43*
8							
9	99	99	99	99	99	99	99

* = Collision

Number of collision = 4.

PTO

Quadratic probing:

	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0							19*
1							
2							
3		33	33	33	33	33	33
4			23*	23*	23*	23*	23*
5				44*	44*	44*	44*
6					56	56	56
7						43*	43*
8							
9	99	99	99	99	99	99	99

* = Collision

Number of collision = 4

4. Explain operations of BST

→ A Binary search tree (BST) is a binary tree data structure where each node has at most two children, and the nodes are arranged such that for each node:

1. All nodes in the left subtree have values less than the node's value.
2. All nodes in the right subtree have values greater than the node's value.

The different operations performed on a BST are:

1. Search Operation:

To search for a value in a BST, you start at the root node and follow the tree's structure based on comparisons between the target value and the node's value.

- Start at the root node
- If the target value matches the current node's value, you have found the element
- If the target value is less than the current node's value, continue searching in the left subtree
- If the target value is greater than the current node's value, continue searching in the right subtree.
- Repeat these steps until you find the value or reach a leaf node (indicating the value is not in the tree).

The time complexity for searching in a balanced BST is O(log n) on average, where n is the number of nodes.

2. Insertion operation:

To insert a new value into a BST:

- Start at the root node.
- Compare the value to be inserted with the current node's value.
- If the value is less than the current node's value, move to the left subtree.
- If the value is greater, move to the right subtree.
- Repeat this process until you find an empty spot (a leaf node.)
- Insert the new value as a child.

The time complexity for insertion in a balanced BST is O(log n) on average.

3. Deletion Operation

There are 3 cases to consider when deleting a node:

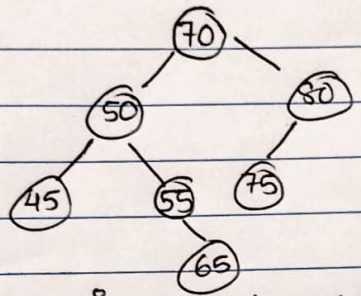
- Node to be deleted is a leaf node (no children): In this case, you can simply remove the node.
- Node to be deleted has one child: Replace the node to be deleted with its child.
- Node to be deleted has two children: This is the most complex case. You can replace the node to be deleted with the smallest node from the right subtree (Inorder successor) or the largest node from the left subtree (Inorder predecessor). This node will maintain the BST properties. Then, recursively delete the node you used as a replacement.

The time complexity for deletion in a balanced BST is O(log n) on average

4. Traversal Operations:

Inorder Traversal : Visits nodes in sorted order, making it useful for printing values in ascending order

Consider the tree



Inorder of the above tree is 45 50 55 65 70 75 80

Preorder Traversal : Visits the current node before its children

Preorder of the above tree is 70 50 45 55 65 80 75

Postorder Traversal : Visits the current node after its children

Postorder of the above tree is 45 65 55 50 75 80 70

23/10/23