INDEX

1BM19CS046

NAME: DEVKANSAL STD.: 3 SEC.: D ROLL NO.: 1BM19CS046 SUB.:

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
LAB1	7/12/23	Stack Implementation		
LAB2	28/12/23	1. Infix to postfix 2. Postfix evaluation		
LAB3	10/1/24	1. Circular Queue 2. Linear Queue		
LAB4	18/1/24	1. Singly LL insertion		
LAB5		2. Singly LL deletion 3. Leet code (Min Stack)		
LAB6	25/1/24	1. Operations on LL 2. LL as Stack 3. LL as Queue		
LAB7	1/2/24	1. Doubly LL 2. Leet code		
LAB8	16/2/24	1. Binary Search Tree 2. Leet code		
LAB9	22/2/24	1. BFS 2. DFS 3. Leetcode		
LAB10	29/2/24	1. Hashing		

1BM19CS046

INDEX

NAME: DEVKANSAL STD.: 3 SEC.: D ROLL NO.: 1BM19CS046 SUB.: _____

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
LAB1	7/12/23	Stack Implementation		
LAB2	28/12/23	1. Infix to postfix 2. Postfix evaluation		
LAB3	11/1/24	1. Circular Queue 2. Linear Queue		
LAB4	18/1/24	1. Singly LL insertion		
LAB5		2. Singly LL deletion 3. Leet code (Min stack)		
LAB6	25/1/24	1. Operations on LL 2. LL as stack 3. LL as Queue		
LAB7	1/2/24	1. Doubly LL 2. Leet code		
LAB8	6/2/24	1. Binary Search Tree 2. Leet code		
LAB9	22/2/24	1. BFS 2. DFS 3. Leetcode		
LAB10	29/2/24	1. Hashing		

LAB 1

→ Write a program for stack implementation:

```
#include <stdio.h>
#define MAX 4

int top = -1;
int stack_arr[MAX];

void push(int data)
{
    if (top == MAX-1)
    {
        printf("stack overflow");
    }
else {
    printf("enter element to be added");
    scanf("%d", &data);

        top = top+1;
        stack_arr[ ] = data;
}
}
```

```
void pop()
{
    int value;
    if (top == -1)
    {
        Printf("underflow");
value = stack.arr[top];
    top = top - 1
    return; }}

void display()
{
    if (top == -1) {
        Printf("underflow")
    }
else
Printf("elements in stack")
    for(int i = top; i >= 0; i++)
    {
    printf("%d" stack.arr[i]); }}
int main()
{
    int choice;
    while (1)
    {
```

```
printf("1. Push \n 2. Pop \n 3. Display \n
        4. Exit");
printf("enter your choice");
scanf("%d", &choice);
switch (choice)
{
    case 1 : Push ();
             break;
    case 2 : Pop ();
             break;
    case 3 : display ();
             break;
    case 4 : exit (0);
}}
```

OUTPUT

1. Push the element
2. Pop element
3. Display
4. Exit

Enter your choice : 1

Enter the element to be pushed : 4

1. Push the element
2. Pop element
3. Display
4. Exit

enter choice : 2
enter the element to pop : 4

1. Push element
2. Pop element
3. Display
4. Exit

Enter your choice : 4

LAB 2

→ WAP to count infix to postfix evaluation

```
# include <stdio.h>
# include <stdlib.h>
# define MAX 100

char stack_arr[MAX];
int top = -1;
void push (char x)
{
    if (top == MAX - 1)
    {
        printf ("overflow");
    }
else { top = top + 1;
       stack_arr[top] = x; }}

void pop ()
{
    char c;
    if (top == -1)
    {
        printf ("underflow");
    }
else {
    x = stack_arr[top]
```

```
return ; }}

int Precedence (char symbol)
  {
    if (symbol == '^')
      return 3;
  else if (symbol == '*' || symbol == '/')
      return 2;
  else if (symbol == '+' || symbol == '-')
      return 1;
  else
      return -1;
      }
void infix-to-postfix (char infix[ ],
                       char postfix[ ])
  {
    int i = 0; j = 0;
    char symbol, top;
    push ('#');
  while (symbol = infix[i++] != '\0')
    {
      if (symbol == '(')
        push (symbol);
    else if (isalnum (symbol))
      postfix[j++] = symbol;
```

```
else if (symbol == ')')
        {
        while stack_arr[top] != '('
        {
        postfix[j++] = pop();
        }
        top = pop();
        }
else
    { while (Prudence(stack_arr[top] >=
                    Prudence(symbol))
        {
        postfix[j++] = Pop();
        }
        push(symbol);
    } }
while (stack[top] != '#')
        {
        postfix[j++] = pop();
        }
        postfix[j] = '\0';
    }
    int main()
    {
    char infix[MAX], postfix[MAX];
    Printf("enter infix ");
```

```
else if (symbol == ')')
        {
    while stack_arr[top] != '('
    {
    postfix[j++] = pop();
    }
    top = pop();
    }
else
    { while (Prudence(stack_arr[top] >=
                    Prudence(symbol))
    {
    postfix[j++] = Pop();
    }
    push(symbol);
    } }
while (stack[top] != '#')
    {
    postfix[j++] = pop();
    }
    postfix[j] = '\0';
}
int main()
{
char infix[MAX], postfix[MAX];
Printf("enter infix ");
```

```
scanf("%s", infix);
infix_to_postfix(infix, postfix);
printf("\n postfix is %s", postfix);
return 0;
}
```

output:

enter a infix expression:

a * b + c * d - e

The postfix is: ab * cd + e -

→ WAP to evaluate postfix expressions:

```
#include <stdio.h>
#define max 20

int stack [max];
int top = -1;
void push (int a)
{
  stack [++top] = a;
}
int pop ()
{
  return stack [top--];
}
void main ()
{
  char postfix [MAX];
printf ("enter postfix exp");
 scanf ("%d", postfix);

 int res=0, a, b;
for (int i=0; i < strlen (postfix); i++)
    {
      if (isdigit (postfix [i]))
        push (postfix [i] - '0');
else
      { b = pop ();
        a = pop ();
```

```
switch (postfix[i])
  {
  case '+' : push (a+b);
             break;
  case '-' : push (a-b);
             break;
  case '*' : push (a*b);
             break;
  case '/' : push (a/b);
             break
  case '^' : push (a^b);
             break;
    }}
res = pop();
printf("%s %d", postfix, res);
}
```

output:

Enter postfix exp: 23 * 31 * + 5 -

23 * 31 * + 5 - = 4

LAB 3 (Linear Queue)

```
#include <stdio.h>
#define size 30

int queue [size];
int front = -1;
int rear = -1;
void insert (int a)
{
    if (rear == size -1)
{
    printf ("overflow");
    return;
    }
    else {
      if (front == -1)
          front = 0;
    queue [++rear] = a;
    }
    }
void delete ()
    {
      if (front == -1 || front > rear)
      {
        printf ("Queue empty");
    }
    else
    { front ++; } }
```

```
void display () {
    if (front == -1)
    {
      printf ("Queue empty");
      return;
    }
  printf ("Queue");
  for (int i = front; i <= rear; i++)
    {
      printf ("%d", queue[i]);
    }
}
void main ()
{
  int choice;
  int a;
  while (1)
  {
  printf ("1. Insert\n 2. delete\n
                3. display\n");
  scanf ("%d", &choice);

  switch (choice)
  {
case 1: printf ("enter an element");
        scanf ("%d", &a);
        insert (a)
        display ();
        break;
```

```
case 2 : delete ();
         display();
         break;
case 3 : display ();
         break ();
    }}}
```

output :

1. Insert
2. Delete
3. Display

choice : 1

enter an element : 7

Queue : 7

1. Insert
2. Delete
3. Display

choice : 1

enter an element : 8

Queue : 4 8

1. Insert
2. Delete
3. Display

Enter choice : 2

Queue 8

LAB 4

→ WAP to insert element at various positions in a linked list:

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

Struct node {
        int data;
        Struct node * next;
        };
void push (Struct node ** head_ref, int
                        new_data)
  {
Struct node * new node = (struct node *)
        malloc (size of (struct node));
new node -> data = new data;
  new node -> next = * (head_ref);
    (* head_ref) = new node;
    }
void append (struct node ** head_ref, int
                        new_data)
  {
Struct node * new node = (struct node *) malloc
        (size of (Struct node));
```

```
struct node *last = (*head_ref);
new node -> data = new data;
new node -> next = NULL;
if (*head_ref == NULL)
{ (*head_ref) = new node;
    return;
}
while (last.next != NULL)
{
  last = last -> Next;
  last -> Next = new node;
  return;
}
void specpos (int data, int position)
  {
    Struct Node *new node = (struct node*)
     malloc (size of) (Struct Node))
new node -> data = new data;
    int i,
Struct node * temp = head;
if (position == 1)
  {
    new node -> next = top;
      head = ptr;
      return;
}
```

```
for (i=1; i<position-1; i++)
    {
      top = top -> next;
    }
new node -> next = temp -> next;
    temp -> next = new node;
  }
void display (struct node * node)
    {
    while (node != NULL)
    {
  printf ("%d", node -> data);
   node = node -> next;
    }}
int main()
{
  struct node * head = NULL;
    int choice, value, position;
   while (1)
  {
printf (" 1. Push\n 2. append\n 3. Insert
     4. display\n 5. exit: ");
 printf ("enter your choice");
 scanf ("%d" & choice);
    switch (choice)
    {
```

```
case 1:
    printf("enter element to push");
    scanf("%d", &value);
    push(&head, value);
    break;
case 2:
printf("enter the element to append");
scanf("%d", &value);
    append(&head, value);
        break;
case 3:
    printf("enter element to be inserted at
                specpos");
scanf("%d", &value);
printf("enter pos");
    scanf("%d", &position");
    specpos(&head, value);
    break;
case 4: display(head);
        break;
case 5: exit(0);
}
    printf("the created list is");
        display(head);
    }
```

Output:

1. Push
2. Append
3. Specpos
4. Display
5. Exit

Enter your choice : 1
Enter element to push : 4
Created list is : 4

1. Push
2. append
3. Specpos
4. Display
5. exit

Enter your choice : 2
Enter element to append : 8
Created list is : 4 8

LAB 5

→ WAP for deletion of elements:

```
#include <stdio.h>
struct Node {
        int data;
        struct node * next;
    };
    struct Node * head;
void pop()
  {
   struct node * ptr;
   if (head == NULL)
    {
        if ("empty");
    }
    else {
          head = ptr;
          head = ptr.next;
         free (ptr);
          return; } }
void append()
    {
      struct node * ptr * ptr1;
        if (head == NULL)
          {
Printf("list empty"); }
else if (head.next == NULL)
```

```
} free(head);
  head=NULL;
  if("Node is deleted");
  }
  else {
     ptr=head;
  while(ptr->next!=NULL)
  {
  ptr1=ptr;
   ptr=ptr.next;}
ptr1.next=NULL;
     free(ptr);
Printf("Last node deleted"); }}

void deepee()
  {
  struct node *ptr *ptr1;
      int loc,i;
scanf("%d",&loc);
    ptr=head;
for(i=0;i<loc;i++)
  {
    ptr1=ptr;
    ptr=ptr1.next;
  if (ptr==NULL)
    pf("%d",loc);
      return;
    }}
```

```
ptr1.next = ptr.next;
free(ptr);
printf("deleted node at %d", loc); }
void push()

void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
    printf("Nothing to print");
    }
    else
    {
    printf("\n");
    while(ptr != NULL)
        {
        Printf("%d", ptr.data);
        ptr = ptr.next;
        }
    }
}
```

```
int main()
{
int choice, value;
while(1)
{
printf("1. Push 2.Pop 3. append
        4. display 5. Speel 6. exit");
scanf("%d", &choice);
    switch(choice)
    {
    case 1:
    printf("enter");
    scanf("%d", &value);
    push(&head, value);
    printf("created list is");
        display();
        break;
case 2:
        pop();
        break();
Case 3:
        append();
        break();
Case 4:
        display();
Case 5: Speelpop(); break;

Case 6: exit(0); }}
```

OUTPUT:

1. Push
2. Pop
3. Append
4. Display
5. Specpos
6. Exit

Enter your choice : 1
the created list is : 1

1. Push
2. Pop
3. Append
4. Display
5. Specpos
6. Exit

Enter your choice : 2
List is : 1 2

1. Push
2. Pop
3. Append
4. Display
5. Specpos
6. exit

enter your choice : 4
List is : 1

Leet code (minstack)

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

typedef struct
{
    int value;
    int main; }
    structnode;
typedef struct
{
    Struct node * array;
    int capacity;
    int top;
}
    min stack;
minstack * minstackcreate()
{
minstack * stack = (minstack*)
    malloc(sizeof(minstack));
Stack -> capacity = 10;
Stack -> capacity = 20;
stack -> array = (StackNode*)
    malloc(stack -> capacity *
    sizeof(struct node));
Stack -> top = -1;
    return stack; }
```

```
void minStackPush(minStack *obj, int val)
{
    if(obj->top == obj->capacity-1)
    {
        obj->capacity *= 2;
        obj->array = (StackNode *)realloc
        (obj->array, obj->capacity *
        sizeof(StackNode));
    }

StackNode Newnode;
Newnode.value = val;
Newnode.min = (obj->top == -1)?
val: (val < obj->array[obj->top].min);
obj->array[++(obj->top)] = NewNode;
}

void minStackPop(minStack *obj)
{
    if(obj->top != -1)
    {
        obj->top--; } }

int minStackTop(minStack *obj)
{
    if(obj->top != -1)
    {
        return obj->array[obj->top].
        value; }
```

```
return -1;
}
int minStack GetMin(minStack * obj)
{ if (obj -> top != -1)
{
return obj -> array[obj -> top]
min; }
return -1; }

void minStackFree(minStack * obj)
{
free(obj -> array);
free(obj); }
```

LAB 6 [operations on LL]

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node Node;

Node * creatNode(int value)
{
Node *newnode=(Node*)malloc
                (sizeof(Node));
newnode->data=value;
newnode->next=NULL;
    return newNode;
}
void displayList(Node *head)
{
while(head!=NULL)
{ printf("%d->" head->data);
head = head->next; }
printf("NULL");
}
```

LAB 6 [Operations on LL]

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node Node;
Node * creatNode(int value)
{
Node *newnode=(Node*)malloc
                (sizeof(Node));
newnode->data=value;
newnode->next=NULL;
    return newNode;
}
void displayList(Node *head)
{
while(head!=NULL)
{ printf("%d->",head->data);
head=head->next;}
printf("NULL");
}
```

```
Node * sortList(Node * head)
{
if(head==NULL || head->next==NULL)
{
    return head;
}
int swapped;
Node *temp;
Node *end = NULL;
do {
    swapped=0;
    temp=head;
while(temp->next != end)
{ if(temp->data > temp->next->data
  { int tempData=temp->data;
temp->data=temp->next->data;
temp->next->data=tempData;
    swapped=1; }
temp=temp->next; }
    end=temp; }
while(swapped)
    return head; }

Node * reverseList(Node * head)
  { Node * prev=NULL;
    Node * current=head;
    Node * nextnode=NULL;
```

```
while(current != NULL)
  {
    Next Node = current->next;
    current -> next = prev;
      prev = current;
       current = nxtnode;
      }
return prev; }
Node * concatenation (Node *L1, Node *L2)
{ if (L1 = NULL)
       return L2;
Node * temp = L1;
while (temp->next != NULL)
   {
     temp = temp -> next;
     }
     temp -> next = L2;
       return L1; }
int main ()
  { Node * L1 = Create Node (3)
L1 -> next = Create Node (1);
L1 -> next = Create Node (1);
L1 -> next -> next = Create Node (4);
 Node * L2 = Create Node (2)
   L2 -> next = Create Node (5);

Print ("original list");
  display List (L1);
```

```
while(current != NULL)
  {
    Next Node = current->next;
    current -> next = prev;
      prev = current;
       current = nxtnode;
    }
return prev; }
Node * concatenation (Node *L1, Node *L2)
{ if (L1 = NULL)
      return L2;
Node * temp = L1;
while (temp->next != NULL)
  {
    temp = temp -> next;
  }
  temp -> next = L2;
    return L1; }
int main ()
  { Node * L1 = Creat Node (3)
L1 -> next = Creat Node (1);
L1 -> next = Creat Node (1);
L1 -> next -> next = Creat Node (4);
 Node * L2 = Creat Node (2)
   L2 -> next = Creat Node (5);

Print ("original list");
  display List (L1);
```

```
Printf("og List 2");
   display list(L2);
L1 = sort list(L1);
Printf("sorted List 1: ");
   display List(L1);
L1 = sort List(L1);
Printf("sorted List");
   display list(L1);
   L1 = reverse List(L1);
   Printf("reversed List");
   display list(L1);
Node * concatenated = concatenated list
                      (L1,L2);
Printf("concat list: ")
   displayList(concat list);
   return 0; }
```

output

```
1
enter Data : 30
1
enter data : 10
1
enter data : 40
2
4
10 30 40
3
4
40 30 10
```

Stack Implementation using LL

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
        int data;
        struct Node *next; };
struct Node *head; *new node, *temp;
        *temp1;
void push()
{ new node=(struct Node*)malloc(sizeof
                (struct Node));
printf("enter data to push");
scanf("%d", &new node->data);
new node->next=head;
    head=new node; }
void display()
{ printf("The created stack is");
    temp=head;
    while (temp!=NULL)
    { printf("%d", temp->data);
      temp=temp->next; }}

void pop()
{
    if (head==NULL)
printf("stock empty");
    else
    { head=head->next; }}
```

```
int main()
{ int ch;
  while(1)
  { printf("\n 1. Push \n 2. Pop \n
          3. Display \n 4. Exit");
scanf("%d", &ch);
switch(ch)
  { case 1: push();
            Display();
            break;
  case 2: pop()
          Display();
          break;
  Case 3: Display();
          break;
  Case 4: exit(0); }}}
```

Output:

1. Push
2. Pop
3. Display
4. Exit

choice: 1

enter data: 20

The stack is 20.

classmate
Date
Page

Queue Implementation of LL

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
        int data;
        struct Node *next; };
struct Node *head, *NewNode,
        *temp, *temp1;
void PushQueue ()
    { NewNode = (struct Node*) malloc
            (sizeof (struct Node));
printf ("enter data to push ");
scanf ("%d", &newnode -> data);
  newnode -> next = head;
    head = newNode; }
void Display()
  { printf ("\n Queue is ");
    temp = head;
while (temp1 != NULL) {
  printf ("%d", temp -> data);
    temp = temp -> Next; } }

void popQueue ()
  {
    if (head == NULL)
      printf ("Queue empty");
use {
```

```
temp = temp1 = head;
while (temp -> next != NULL)
    {
    temp1 = temp;
    temp = temp -> next; }
temp1 -> next = NULL;
    free(temp); }}

int main()
  {
    int ch;
    while (1)
    {
    Printf ("1. Push \n 2. Pop \n 3- Displa

    scanf ("%d", &ch);
    switch (ch)
      {
    case 1: Push queue ();
            Display ();
            break;
    Case 2: Pop queue ();
            Display ();
            break;
```

```
case 3 : Display();
         break;
    }}}
```

OUTPUT

```
1. Push
2. Pop
3. Display
   choice: 1
   enter element : 50
1. Push
2. Pop
3. Display
   choice: 1
   enter element : 60

The created list is 50 60.
```

Lab 7 Doubly LL

```
#include <stdio.h>
#include <stdlib.h>
  struct Node { int data;
      struct Node *prev;
      struct Node * next;
       };
struct Node * create Node(int data)
   {
   struct Node* newNode=(struct node*)
      malloc(size of (struct Node));
   if (New Node == NULL)
       { Printf("Memory allocation fail")
      exit(1); }
New Node -> data = data;
newNode -> Prev = NULL;
New Node -> next = NULL;
return New Node; }
void insert left (struct Node *head,
   struct Node * target, int data)
   {
struct Node * New Node = create Node(Data);
   if (target -> prev -> Next = Newnode;
    newNode -> Prev = target -> prev;
       }
```

else { head = newnode; }
newnode → next = target;
target → prev = newnode; }
void DeleteNode (struct Node * head, int value)
{
struct Node * current = head;
while current != NULL
{ if (current → data == value)
{ if (current → prev != NULL)
{ current → prev → next = current → next; }
else
{ head = current → next;
if (current → next != NULL)
{
current → next → prev = current → prev;
} free(current); return; }
current = current → next; }
printf ("Node %d found, value); }

void display (struct Node * head)
{
if ("Empty LL");
while (head != NULL)
{
Printf ("%d" head → data)
head = head → next;
} Printf ("NULL"); }

```
int main()
{ struct Node *head = NULL;
  head = createNode(1);
  head->next = createNode(2);
  head->next->prev = head
  head->next->next = createNode(3);
head->next->next->prev = head->
                          next;

display(head);
insertLeft(head, head->next, 10);
  printf("After insertion");
      display(head);

  DeleteNode(head, 2);
  printf("After deletion");
    display(head);
      return 0;
    }
```

Output

Doubly Linked List : 1 ⟷ 2 ⟷ 3 ⟷ NULL

After insertion :

Doubly LL : 1 ⟷ 10 ⟷ 2 ⟷ 3 ⟷ NULL

After Deletion :

DoublyLL : 1 ⟷ 10 ⟷ 3 ⟷ NULL

Left Code

```
#Include <stdlib.h>
Struct List Node * Split List Topart( Struct list
  Node * head, int *, int * returnsize )
  {
    Struct List Node * current = head;
      int length = 0;
      while (current) {
          length ++;
    current = current -> next; }
  int Part size = length / k;
   int extra nodes = length % k;
  Struct List Node ** result = (Struct List Node)
      malloc (k * (size of (Struct Node)));
    current = head;
      for (i = 0; i < k; i++)
  { Struct list Node * part head = current
    int part length = part size +
     (i < extra Nodes ? 1 : 0);
  for (int j = 0; j < part length - 1 && extra
    nodes ; i++)
    {
      current = current. next;
    }
```

Date
Page

Leet Code

```
#include <stdlib.h>
struct ListNode * SplitListTopart(struct List
    ListNode * head, int *, int * returnSize)
    {
      Struct ListNode * current = head;
        int length = 0;
        while (current) {
            length ++;
      current = current -> next; }
    int part size = length / k;
     int extra nodes = length % k;
    Struct List Node ** result = (struct ListNode)
        malloc (k * (size of (Struct Node)));
      current = head;
        for (i = 0; i < k; i++)
    { struct list Node * part head = current
      int part length = part size +
      (i < extra Nodes ? 1 : 0);
    for (int j = 0; j < part length - 1 && extra
      nodes ; i++)
      {
        current = current . next;
      }
```

```
if (current)
{
    struct ListNode * partHead = current;
int partLength = partSize + (i < extra nodes);
for(int j = 0; j < partLength - 1; current = current->next;
                                   i++)
else
{ result[i] = NULL; }
    *returnSize = k;
    return result;
}
```

Output:

[[1], [2], [3], [ ], [ ]]

BINARY SEARCH TREE

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

typedef struct * bst
{ int data;
  struct bst * left;
  struct bst * right;
} node;
node * create()
{ node *temp;
printf("enter data");
temp=(node *)malloc(sizeof(node));
scanf("%d", temp->data);
return temp;
}
void insert(node * root, node * temp)
{
  if (temp-> data < root-> data)
  {
    if(root -> left != NULL)
    {
      insert(root -> left, temp);
    }else
    { root -> lft = temp; }}
```

```
if (temp->data > root->data)
{
    if (root->right != NULL)
    {
        insert(root->right, temp)
    } else
    { root->right = temp; }}}

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

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

void postorder(node *root)
{
  if (root != NULL)
  {
```

```
postorder(root->left);
Postorder(root->right);
Printf("%d", root.data);
}}

int main()
{ int ch;
node *root = New *temp;
do
{ Printf("enter your choice");
printf("1. Insert 2. Preorder 3 Inorder
        4. Postorder");
switch(ch)
{ case 1: insert(root, temp);
        break;
case 2: preorder(root);
        break;
case 3: inorder(root);
        break;

case 4: postorder(root);
        break;
case 5: exit(0);
        break;
}} while(ch!=5)
        return 0;
    }
```

Output:

Enter your choice 2

1. Insert
2. Preorder
3 Inorder
4 Postorder

Enter value to be Inserted: 2

Enter Choice : 3

1. Insert
2. Preorder
3. Inorder
4. Postorder

Enter choice: 4

Enter value to be inserted: 3

Leet Code

```
int Get length (struct List Node * head)
{ struct List Node * temp = head;
  int count = 0;
  while (temp != NULL)
    {
      count ++;
      temp = temp -> next ;
    } return count ; }
Struct List Node * rotate right [struct
        Listnode * head, int k].
  {
    int length = Get length (head);
    Struct list Node * temp = head;
    Struct list Node * temp1;
  if (head == NULL) return head;
  else if (head -> next == NULL)
          return head;
  else
    { int i = length % k;
      while (i > 0)
    { temp = head;
      while (temp -> next = NULL)
        { temp1 = temp;
```

```
temp = temp->next;
}
temp->next = temp->next;
temp->next = head;
head = temp;
}
```

Output

Test case 1: [1, 2, 3, 4, 5]

k = 6

↳ [5, 1, 2, 3, 4]

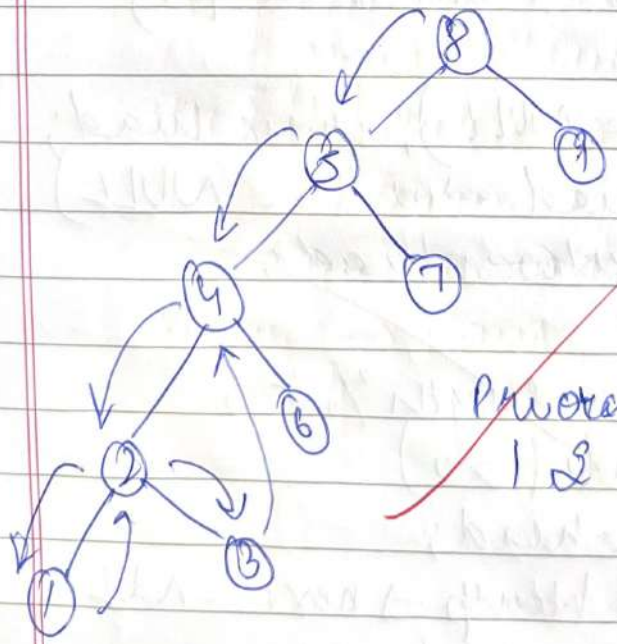
Test Case 2:

head: [0, 1, 2]

k = 4

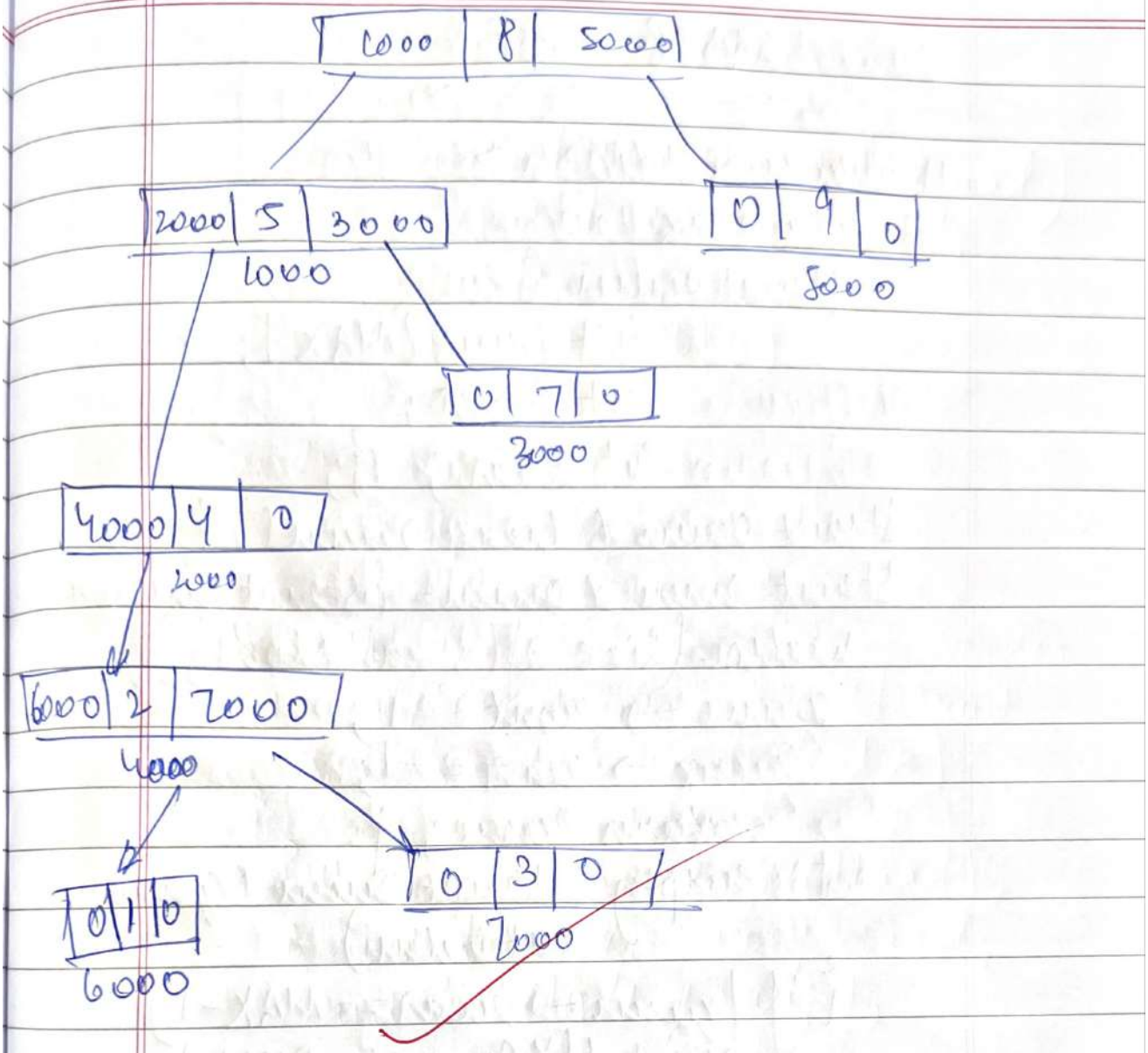
Test case 3: [2, 0, 1]

Tracing



Preorder:

1 2 3 4 6 7 8 9



LAB 8 BFS

```
#include <stdio.h>
#define max 100
    struct Queue {
            int items[MAX];
            int front;
            int rear; };
struct Queue * createQueue() {
struct Queue *Queue = (struct Queue*)
  malloc(sizeof(struct Node));
   queue -> front = -1;
    queue -> rear = -1;
      return Queue; }
int isempty (struct Queue *queue,
            int value)
 { if (queue -> rear == MAX-1)
    printf("Queue is FULL");
else if (queue -> front == -1)
        Queue -> front = 0;
      queue -> rear ++;
queue -> items[queue->rear] = value;
   } }
```

```
int dequeue(struct queue *queue)
{ int item;
  if (isempty(queue))
printf("Queue empty");
    item = -1; } else {
item = queue->front++; }
  return item; }
int objmatrix[MAX][MAX];
int visited[MAX];
void BFS(int start, int nodes){
struct Queue *queue = createQueue();
  visited[start] = 1;
enqueue(queue, start);
while(!isempty(queue)) { int currentNode
  = dequeue(queue); };
printf("current Node");
for(int i=0; i< nodes; i++)
  { if(objmatrix[currentNode][i]
    && notvisited[i])
  { visited[i] = 1;
    enqueue(queue, i); }
}

int main()
{ int nodes, startNode
printf("enter no. of nodes");
scanf("%d", &nodes);
```

```
printf("Enter matrix");
for (i=0 ; i<nodes ; i++)
    for (j=0 ; j<nodes ; j++)
        scanf("%d", &a[i][j]);
BFS(StartNode, Nodes);
}
```

Output:

Enter no. of vertices: 7

Enter matrix: 0 1 0 1 0 0 0

1 0 1 1 0 0 1

1 0 1 1 0 1 0

0 0 0 1 1 0 0

1 1 0 0 1 0 0

The graph is connected.

LAB 9 DFS

```
#include <stdio.h>
#define MAX 10
void dfs(int graph[MAX vertices][MAX];
void num vertices (Root visited[MAX];
int vertex){
    visited[(vertex)] = true;
    int i;
for(int i=0; i<numvertices; i++)
  {
    if [graph(vertex[i]) = kk | v[i])
dfs(graph, num, vertices, visited[i]);
  }}
    bool is connected (int graph(MAX)
  { bool visited[MAX] = false;
dfs( graph, num-vertices, visited, a)
    int i;
for [int i=0; i< num-vertices; i++)
  { if (!visited[i]) {
      out false;
    } return true;
    };
```

```
int main()
{
int num_vertices;
printf("enter vertices");
scanf("%d", &num_vertices);
int i, j;
int graph[MAX][MAX Vertices]
printf("Enter Matrix");
for(i=0; i<mod_vertices; i++)
{ for(j=0; j<mod_vertices; j++)
{ scanf("%d", graph[i][j]);
} }
if(isconnected(graph, num_vertices)
{
printf("The graph is connected");
else printf("Not connected");
} return 0; }
```

Output

Enter No. of vertices: 7

Enter matrix: 0 1 0 0 0 0 0

1 1 0 0 1 0 1

0 0 1 0 1 0 1

0 0 1 1 1 1 0

0 0 1 0 0 0 1

0 0 0 0 1 0 1

1 1 0 0 0 1 0

Enter starting vertex: 2

2 1 3 4 5 0 6

CLASSMATE
Date
Page

LAB 10 Hashing Linear Probing

```
# define Table_size 100
# define key_length 5
# define Max_Name_length 50
struct employee {
    char key [Max_Name_length];
    char designation [Max_designation_length];
    float salary ; };
struct hash table {
    struct employee * table [Table_size];
};
int hash_function (const char *key, n)
{
    int sum=0 ;
    for (i=0; key[i]; i++)
    { sum= key[i] ; }
    return sum ;
} void insert (struct hashtable, struct
    employee * emp ) {
int index=hash_function (emp->key,
    Table_size);
while (ht -> table [index] != NULL)
index = (index+1) % Table_size;
ht -> table [index] != NULL
ht -> table [index] = emp;
}
```

```
struct employee * search ( struct hashtable
    * ht , const char * key )
{ int index = hash ( key , Table_size )

while ( ht -> table[index] -> key , key ) != 0
  { return ht -> table [index]
  } index = ( index + 1 ) % Table_size ;
  } return NULL ;
  } int main ()
    { struct hashtable ht ;
      struct employee * emp ;
char key [key_length] ;
char filename [100] ; char line [100] ;
for ( i = 0 ; i < Table_size ; i ++ )
      { ht table [i] = NULL ; }
printf ( " enter file name " ) ;
  scanf ( " %s " , filname ) ;
    if ( fil == NULL ) {
  printf ( " error " ) ; return 1 ; }
while ( line , size of ( line ) , fil ) }
emp = ( struct employee * ) malloc ( size of
          ( struct employee ) ) ;
  insert ( & ht , emp ) ; }
int choice
```

```
do { printf ("1 search 2 exit");
printf ("\n choice"); scanf (choice) {
case 1: search ();
        break;
case 2: printf ("exit");
        break;
} while (choice != 2);
} return 0; }
```

Output

Item found at index 6.

Hacker Rank
Swapping Subtrees function

```
void swapAtLevel(node *root, int k, level)
{ if (root == NULL) return;
  if (level % k == 0) {
  node *temp = root -> left;
    root -> left = temp; }
swapAtLevel(root -> left, k, level+1);
swapAtLevel(root -> right, k, level+1);
int **swapNodes(int indexRows, int
    cols, int **indexes, queries)

Node **Node = (Node **)malloc(indexRows
                size of (Node));
for (i=0; i<indexRows; i++) {
  int leftIndex = index[i][0];
  int rightIndex = index[i][1];
if (leftIndex != -1) nodes(i+1) -> left =
    nodes[leftIndex]

if (rightIndex != -1) nodes(i+1) ->
    right = nodes(right -> root);
```

```
for (P=1 ; i<=n ; i++)
    {if (s[i]) count++ ;
    if (count==n)
    Printf("connected");
else Printf("not connected");
    return 0;
}
```

output: enter vertices : 6

Enter Matrix :

0	1	1	0	0	1	1 → 2
1	1	0	0	0	1	2 → 4
1	1	1	0	0	0	4 → 3
0	1	1	0	1	0	3 → 5
0	1	0	0	1	0	3 → 6
1	1	0	0	1	0	

Connected !

Smoke