Netaji Subhas University of Technology
Department of Computer Science Engineering

Lab Practical File

Data Structures

Professor
Prof. R.S. Rao

Student
Kushagra Lakhwani - 2021UCI8036

Academic Session 2021-2022

Contents

Requirements

The following system configuration was used for the implementation of the practicals.

1. Operating System: GNU/Linux Arch
2. Compiler: GCC 12.1.0
3. Editor: VS Code with C/C++ extension
4. Formatter: ClangFormat

1 Mean and Median

1.1 Objective

Write a program to find the mean and the median of the numbers stored in an array.

1.2 Algorithm

```
Start
Step 1 -> declare function to calculate mean
    double mean(int arr[], int size)
    declare int sum = 0
    Loop For int i = 0 and i < size and i++
    Set sum += arr[i]
End
    return (double)sum/(double)size

Step 2  -> declare function to calculate median
    double median(int arr[], int size)
    call sort(arr, arr+size)
    IF (size % 2 != 0)
    return (double)arr[size/2]
End
    return (double)(arr[(size-1)/2] + arr[size/2])/2.0

Step 3  -> In main()
    Declare int arr[] = {3,5,2,1,7,8}
    Declare int size = sizeof(arr)/sizeof(arr[0])
    Call mean(arr, size)
    Call median(arr, size)
Stop
```

1.3 Code

```c
#include <stdio.h>
int main(int argc, char const *argv[])
{
    int arr[50], n;
    printf("Enter the size of the arr:\n");
    scanf("%d", &n);
    printf("enter the elements: \n");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Your array is:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    printf("your sorted array here: \n");
    int i, j, a;
    for (i = 0; i < n; ++i)
        for (j = i + 1; j < n; ++j)
            if (arr[i] > arr[j])
            {
                a = arr[i];
                arr[i] = arr[j];
                arr[j] = a;
            }
    printf("Your array is:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        int a = arr[i];
        sum += a;
    }
    int mean = sum / n;
    int median = arr[n / 2];

    printf("\n");
    printf("your mean is given as: %d\n", mean);
    printf("your median is given as: %d\n", median);
}
```

1.4 Output

```
Enter the size of the arr:
5

enter the elements:
21
23
44
55
11

Your array is:
21 23 44 55 11
your sorted array here:
Your array is:
11 21 23 44 55

your mean is given as: 30
your median is given as: 23
```

2 Array Insertion/Deletion

2.1 Objective

Write a Program to insert and delete an element from an array.

2.2 Algorithm

2.2.1 Deletion

```
Start
Set J = K
Repeat steps 4 and 5 while J < N
    Set LA[J] = LA[J + 1]
    Set J = J+1
Set N = N-1
Stop
```

2.2.2 Insertion

```
Begin
IF N = MAX, return
ELSE
N = N + 1
SEEK Location index
For All Elements from A[index] to A[N]
    Move to next adjacent location
    A[index] = New_Element
End
```

2.3 Code

```c
#include <stdio.h>
int main(int argc, char const *argv[])
{
    int arr[100];
    int i, item, pos, size = 7;
    printf("Enter 7 elements: ");
    // reading array
    for (i = 0; i < size; i++)
        scanf("%d", &arr[i]);
    // print the original array
    printf("Array before insertion: ");
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
```

```c
    // read element to be inserted
    printf("Enter the element to be inserted: ");
    scanf("%d", &item);
    // read position at which element is to be inserted
    printf("Enter the position at which the element is to be inserted: ");
    scanf("%d", &pos);
    // increase the size
    size++;
    // shift elements forward
    for (i = size - 1; i >= pos; i--)
        arr[i] = arr[i - 1];
    // insert item at position
    arr[pos - 1] = item;
    // print the updated array
    printf("Array after insertion: ");
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

2.4 Output

```
Enter 7 elements: 2 4 6 5 1 5 7
Array before insertion: 2 4 6 5 1 5 7
Enter the element to be inserted: 3
Enter the position at which the element is to be inserted: 2
Array after insertion: 2 3 4 6 5 1 5 7
```

3 Search inside Array

3.1 Objective

Write a program to search for a number in array.

3.2 Algorithm

```
Start
    [Initialize counter variable. ] Set i = 0
    Repeat Step 04 and 05 for i = 0 to i < n
    if a[i] = x, then jump to step 07
    [Increase counter. ] Set i = i + 1
    [End of step 03 loop. ]
    Print x found at i + 1 position and go to step 09
    Print x not found (if a[i] != x, after all the iteration of the
    above for loop. )
Stop
```

3.3 Code

```c
#include <stdio.h>
int main(int argc, char const *argv[])
{

    int arr[10], Size, i, Search, Flag;
    printf("\n Please Enter the size of an array : ");
    scanf("%d", &Size);
    printf("\n Please Enter %d elements of an array: \n", Size);
    for (i = 0; i < Size; i++)
        scanf("%d", &arr[i]);
    printf("\n Please Enter the Search Element : ");
    scanf("%d", &Search);
    Flag = 0;
    for (i = 0; i < Size; i++)
        if (arr[i] == Search)
        {
            Flag = 1;
            break;
        }
    if (Flag == 1)
        printf("\n We found the Search Element %d at Position %d ", Search, i + 1);
    else
        printf("\n Sorry!! We haven't found the the Search Element %d ", Search);
```

```
    return 0;
}
```

3.4 Output

```
Please Enter the size of an array : 5

Please Enter 5 elements of an array:
1 2 3 4 5

Please Enter the Search Element : 2

We found the Search Element 2 at Position 2
```

4 Sort Array

4.1 Objective

Write a program to sort an array

4.2 Algorithm

```
1  START
2      INITIALIZE arr[] ={5, 2, 8, 7, 1 }..
3      SET temp =0
4      length= sizeof(arr)/sizeof(arr[0])
5      PRINT "Elements of Original Array"
6      SET i=0 REPEAT STEP 7 and STEP 8 UNTIL i<length
7      PRINT arr[i]
8      i=i+1
9      SET i=0 REPEAT STEP 10 to STEP UNTIL i<n
10     SET j=i+1 REPEAT STEP 11 UNTIL j<length
11     if(arr[i]>arr[j]) then
12         temp = arr[i]
13         arr[i]=arr[j]
14         arr[j]=temp
15         j=j+1
16         i=i+1
17     PRINT new line
18     PRINT "Elements of array sorted in ascending order"
19     SET i=0 REPEAT STEP 17 and STEP 18 UNTIL i<length
20     PRINT arr[i]
21     i=i+1
22     RETURN 0
23 END.
```

4.3 Code

```c
#include <stdio.h>
int main()
{
    // Initialize array
    int arr[] = {5, 2, 8, 7, 1};
    int temp = 0;
    // Calculate length of array arr
    int length = sizeof(arr) / sizeof(arr[0]);
    // Displaying elements of original array
    printf("Elements of original array: \n");
    for (int i = 0; i < length; i++)
        printf("%d ", arr[i]);
    // Sort the array in ascending order
```

```
    for (int i = 0; i < length; i++)
        for (int j = i + 1; j < length; j++)
            if (arr[i] > arr[j])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
    printf("\n");
    // Displaying elements of array after sorting
    printf("Elements of array sorted in ascending order: \n");
    for (int i = 0; i < length; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

4.4 Output

```
Elements of original array:
5 2 8 7 1
Elements of array sorted in ascending order:
1 2 5 7 8
```

5 Linked List

5.1 Objective

Write a program to implement a linked list.

5.2 Algorithm

```
Create a class Node which has two attributes: data and next. Next is a
   pointerto the next node.
Create another class which has two attributes: head and tail.

addNode() will add a new node to the list:
    a. Create a new node.
    b. It first checks, whether the head is equal to null which means the
   list is
    empty.
    c. If the list is empty, both head and tail will point to the newly
   added node.
    d. If the list is not empty, the new node will be added to end of the
   list such that tail's next will point to the newly added node. This new
    node will become the new tail of the list.

    display() will display the nodes present in the list:

    Define a node current which initially points to the head of the list.
    a. Traverse through the list till current points to null.
    b. Display each node by making current to point to node next to it in
   each iteration.
```

5.3 Code

```c
#include <stdio.h>
#include <stdlib.h>
// Represent a node of singly linked list
struct node
{
    int data;
    struct node *next;
};
// Represent the head and tail of the singly linked list
struct node *head, *tail = NULL;
// addNode() will add a new node to the list
void addNode(int data)
{
    // Create a new node
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
```

```
    newNode->data = data;
    newNode->next = NULL;
    // Checks if the list is empty
    if (head == NULL)
    {
        // If list is empty, both head and tail will point to new node
        head = newNode;
        tail = newNode;
    }
    else
    {
        // newNode will be added after tail such that tail's next will point to newNod
        tail->next = newNode;
        // newNode will become new tail of the list
        tail = newNode;
    }
}
// display() will display all the nodes present in the list
void display()
{
    // Node current will point to head
    struct node *current = head;
    if (head == NULL)
    {
        printf("List is empty\n");
        return;
    }
    printf("Nodes of singly linked list: \n");
    while (current != NULL)
    {
        // Prints each node by incrementing pointer
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
int main()
{
    // Add nodes to the list
    addNode(1);
    addNode(2);
    addNode(3);
    addNode(4);
    // Displays the nodes present in the list
```

```
    display();
    return 0;
}
```

5.4 Output

```
Nodes of singly linked list:
1 2 3 4
```

6 Linked List Deletion

6.1 Objective

Write a program to insert and delete a node from linked list.

6.2 Algorithm

```
Start
Step 1 -> Implement a function to insert at the beginning of linked list
    void insertAtBeginning(struct Node** head, int newdata)
    Allocate memory for new node
    Set data of new node = newdata
    Set next pointer of new node = head
    Set head pointer = new node
End
Step 2 -> Implement a function to insert at the end of the linked list
    void insertAtEnd(struct Node** head, int newdata)
    Allocate memory for newNode
    Set data of newNode = newdata
    Set next pointer of newNode = NULL
    Let lastNode = Head
    If(lastNode = NULL)
        Set head = newNode
        return
    Else
        Traverse till lastNode -> next!=NULL
        Set lastNode -> next=newNode
return
End
Step 3 -> Implement a function to insert after a given node
void insertAfter(struct Node* prev_Node, int newdata)
    if(prev_Node = NULL)
        print("previous node cannot be NULL")
        return
    Else
        Allocate memory for newNode
        Set data of newNode = newdata
        Set next pointer of newNode = pre_Node -> next
        Set next pointer of prev_Node = newNode
End
Step 4 -> Implement a function to delete a given node
void deleteNode(struct Node** head, int key)
    Let Node*temp = head
    Traverse while(temp!=NULL)
    If(temp -> data=key)
        Free(temp)
        Return
    Else
        temp=temp -> next
    Stop
```

6.3 Code

```c
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
void InsertAtBeginning(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    *head_ref = new_node;
}
void insertAfter(struct Node *prev_node, int new_data)
{
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }
    struct Node *new_node =
        (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
void InsertAtEnd(struct Node **head_ref, int new_data)
{
    struct Node *new_node =
        (struct Node *)malloc(sizeof(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 deleteNode(struct Node **head_ref, int key)
{
    struct Node *temp = *head_ref, *prev;
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL)
        return;
    prev->next = temp->next;
    free(temp);
}
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
int main()
{
    struct Node *head = NULL;
    append(&head, 6);
    push(&head, 7);
    push(&head, 1);
    append(&head, 4);
    insertAfter(head->next, 8);
    printf("Created Linked list is: ");
    printList(head);
    return 0;
}
```

6.4 Output

```
Created Linked list is: 1 7 8 6 4
Linked list after deletion: 1 7 6 4
```

7 Stack

7.1 Objective

Write a program to implement a stack using an array.

7.2 Algorithm

```
Start
Step 1 ->  Implement a function to push elements into stack
    void push()
    if(top > size of array -1)
        print("stack overflow")
    else
        input the value to be pushed
        top=top+1
        stack[top]=new_element
End
Step 2 -> Implement a function to pop from stack
    void pop()
    if(top < -1)
        print("stack underflow")
    else
        delete stack[top]
        top=top-1
    End
Stop
```

7.3 Code

```c
#include <stdio.h>
int stack[100], choice, n, top, x, i;
void push(void);
void pop(void);
void display(void);
int main(int argc, char const *argv[])
{
    top = -1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d", &n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d", &choice);
```

```
        switch (choice)
        {
        case 1:
            push();
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("\n\t EXIT POINT ");
            break;
        default:
            printf("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }
    } while (choice != 4);
    return 0;
}
void push()
{
    if (top >= n - 1)
        printf("\n\tSTACK is over flow");
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d", &x);
        top++;
        stack[top] = x;
    }
}
void pop()
{
    if (top <= -1)
        printf("\n\t Stack is under flow");
    else
    {
        printf("\n\t The popped elements is %d", stack[top]);
        top--;
    }
}
void display()
{
```

```c
    if (top >= 0)
    {
        printf("\n The elements in STACK \n");
        for (i = top; i >= 0; i--)
            printf("\n%d", stack[i]);
        printf("\n Press Next Choice");
    }
    else
        printf("\n The STACK is empty");
}
```

7.4 Output

```
  Enter the size of STACK[MAX=100]: 4

        STACK OPERATIONS USING ARRAY
    --------------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:2

 Enter the Choice:2

         The popped elements is 2
 Enter the Choice:3

 The STACK is empty
 Enter the Choice:4

         EXIT POINT
```

8 Queue

8.1 Objective

Write a program to implement a queue using an array

8.2 Algorithm

```
Queue operations work as follows:

    two pointers FRONT and REAR
    FRONT track the first element of the queue
    REAR track the last element of the queue
    initially, set value of FRONT and REAR to -1

Enqueue Operation

    check if the queue is full
    for the first element, set the value of FRONT to 0
    increase the REAR index by 1
    add the new element in the position pointed to by REAR

Dequeue Operation

    check if the queue is empty
    return the value pointed by FRONT
    increase the FRONT index by 1
    for the last element, reset the values of FRONT and REAR to -1
```

8.3 Code

```c
// Queue implementation in C

#include <stdio.h>
#define SIZE 5

void enQueue(int);
void deQueue();
void display();

int items[SIZE], front = -1, rear = -1;

int main()
{
    // deQueue is not possible on empty queue
    deQueue();
```

```
    // enQueue 5 elements
    enQueue(1);
    enQueue(2);
    enQueue(3);
    enQueue(4);
    enQueue(5);

    // 6th element can't be added to because the queue is full
    enQueue(6);

    display();

    // deQueue removes element entered first i.e. 1
    deQueue();

    // Now we have just 4 elements
    display();

    return 0;
}

void enQueue(int value)
{
    if (rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (front == -1)
            front = 0;
        rear++;
        items[rear] = value;
        printf("\nInserted -> %d", value);
    }
}

void deQueue()
{
    if (front == -1)
        printf("\nQueue is Empty!!");
    else
    {
        printf("\nDeleted : %d", items[front]);
        front++;
        if (front > rear)
```

```c
            front = rear = -1;
        }
}

// Function to print the queue
void display()
{
    if (rear == -1)
        printf("\nQueue is Empty!!!");
    else
    {
        int i;
        printf("\nQueue elements are:\n");
        for (i = front; i <= rear; i++)
            printf("%d  ", items[i]);
    }
    printf("\n");
}
```

8.4 Output

```
Queue is Empty!!
Inserted -> 1
Inserted -> 2
Inserted -> 3
Inserted -> 4
Inserted -> 5
Queue is Full!!
Queue elements are:
1  2  3  4  5

Deleted : 1
Queue elements are:
2  3  4  5
```

9 Factorial

9.1 Objective

Write a program for Factorial

9.2 Algorithm

```
factorial(n)
    If n == 1 then return 1
    Else
        f = n * factorial(n - 1)
    Return f
```

9.3 Code

```c
#include <stdio.h>
typedef unsigned long long ull;

ull factorial(ull i)
{
    if (i == 0)
        return 1;
    return factorial(i - 1) * i;
}
int main(int argc, char const *argv[])
{

    int n;
    ull fact = 1;
    printf("Enter an integer: ");
    scanf("%d", &n);

    // shows error if the user enters a negative integer
    if (n < 0)
        printf("Error! Factorial of a negative number doesn't exist.");
    else
        fact = factorial(n);
    printf("Factorial of %d = %llu", n, fact);

    return 0;
}
```

9.4 Output

```
Enter an integer: 11
Factorial of 11 = 39916800
```

10 Linked List Operations

10.1 Objective

Write a program for insertion, deletion and traversal in linked list.

10.2 Algorithm

10.2.1 Traversal

```
Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply process to PTR -> DATA
Step 4: SET PTR = PTR->NEXT
[END OF LOOP]
Step 5: EXIT
```

10.2.2 Insertion

```
(AT BEGINNING)
    Step 1: IF AVAIL = NULL
    Write OVERFLOW
    Go to Step 7
    [END OF IF]
    Step 2: SET NEW_NODE = AVAIL
    Step 3: SET AVAIL = AVAIL -> NEXT
    Step 4: SET NEW_NODE -> DATA = VAL
    Step 5: SET NEW_NODE -> NEXT = HEAD
    Step 6: SET HEAD = NEW_NODE
    Step 7: EXIT

(AT END)
    Step 1: IF AVAIL = NULL
    Write OVERFLOW
    Go to Step 10
    [END OF IF]
    Step 2: SET NEW_NODE = AVAIL
    Step 3: SET AVAIL = AVAIL -> NEXT
    Step 4: SET NEW_NODE -> DATA = VAL
    Step 5: SET NEW_NODE -> NEXT = NULL
    Step 6: SET PTR = HEAD
    Step 7: Repeat Step 8 while PTR -> NEXT != NULL
    Step 8: SET PTR = PTR -> NEXT
    [END OF LOOP]
    Step 9: SET PTR -> NEXT = NEW_NODE
    Step 10: EXIT

(AFTER A GIVEN NODE)
    Step 1: IF AVAIL = NULL
    Write OVERFLOW
```

```
    Go to Step 12
    [END OF IF]
    Step 2: SET NEW_NODE = AVAIL
    Step 3: SET AVAIL = AVAIL -> NEXT
    Step 4: SET NEW_NODE -> DATA = VAL
    Step 5: SET PTR = HEAD
    Step 6: SET PREPTR = PTR
    Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
    Step 8: SET PREPTR = PTR
    Step 9: SET PTR = PTR -> NEXT
    [END OF LOOP]
    Step 1 : PREPTR -> NEXT = NEW_NODE
    Step 11: SET NEW_NODE -> NEXT = PTR
    Step 12: EXIT
```

10.2.3 Deletion

```
(AT BEGINNING)
    Step 1: IF HEAD = NULL
    Write UNDERFLOW
    Go to Step 5
    [END OF IF]
    Step 2: SET PTR = HEAD
    Step 3: SET HEAD = HEAD -> NEXT
    Step 4: FREE PTR
    Step 5: EXIT
    (AT END)
    Step 1: IF HEAD = NULL
    Write UNDERFLOW
    Go to Step 8
    [END OF IF]
    Step 2: SET PTR = HEAD
    Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
    Step 4: SET PREPTR = PTR
    Step 5: SET PTR = PTR -> NEXT
    [END OF LOOP]
    Step 6: SET PREPTR -> NEXT = NULL
    Step 7: FREE PTR
    Step 8: EXIT

(AFTER A NODE)
    Step 1: IF HEAD = NULL
    Write UNDERFLOW
    Go to Step 10
    [END OF IF]
    Step 2: SET PTR = HEAD
    Step 3: SET PREPTR = PTR
    Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
    Step 5: SET PREPTR = PTR
    Step 6: SET PTR = PTR -> NEXT
    [END OF LOOP]
    Step 7: SET TEMP = PTR
```

```
    Step 8: SET PREPTR -> NEXT = PTR -> NEXT
    Step 9: FREE TEMP
    Step 10 : EXIT
```

10.3 Code

```c
#include <stdio.h>
#include <stdlib.h>
// Create a node
struct Node
{
    int data;
    struct Node *next;
};
// Insert at the beginning
void insertAtBeginning(struct Node **head_ref, int new_data)
{
    // Allocate memory to a node
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    // insert the data
    new_node->data = new_data;
    new_node->next = (*head_ref);
    // Move head to new node
    (*head_ref) = new_node;
}
// Insert a node after a node
void insertAfter(struct Node *prev_node, int new_data)
{
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
// Insert the the end
void insertAtEnd(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    struct Node *last = *head_ref; /* used in step 5*/
    new_node->data = new_data;
```

```c
    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;
}
// Delete a node
void deleteNode(struct Node **head_ref, int key)
{
    struct Node *temp = *head_ref, *prev;
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;
        free(temp);
        return;
    }
    // Find the key to be deleted
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }
    // If the key is not present
    if (temp == NULL)
        return;
    // Remove the node
    prev->next = temp->next;
    free(temp);
}
// Search a node
int searchNode(struct Node **head_ref, int key)
{

    struct Node *current = *head_ref;
    while (current != NULL)
    {
        if (current->data == key)
            return 1;
        current = current->next;
```

```c
    }
    return 0;
}
// Sort the linked list
void sortLinkedList(struct Node **head_ref)
{
    struct Node *current = *head_ref, *index = NULL;
    int temp;
    if (head_ref == NULL)
        return;
    else
        while (current != NULL)
        {
            // index points to the node next to current
            index = current->next;
            while (index != NULL)
                if (current->data > index->data)
                {
                    temp = current->data;
                    current->data = index->data;
                    index->data = temp;
                }
            index = index->next;
            current = current->next;
        }
}
// Print the linked list
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

int main(int argc, char const *argv[])
{
    struct Node linkedList;
    insertAtBeginning(&linkedList, 30);
    insertAtEnd(&linkedList, 20);
    insertAtEnd(&linkedList, 10);
    printList(&linkedList);
```

```
    deleteNode(&linkedList, 20);
    printList(&linkedList);

    sortLinkedList(&linkedList);
    return 0;
}
```

10.4 Output

11 Height of a Binary Tree

11.1 Objective

Write a program to find the height of a Binary tree

11.2 Algorithm

```
maxDepth()
1. If tree is empty then return -1
2. Else
     (a) Get the max depth of left subtree recursively  i.e., call maxDepth( tree->left-subtree)
     (a) Get the max depth of right subtree recursively  i.e., call maxDepth( tree->right-subtree)
     (c) Get the max of max depths of left and right subtrees and add 1 to it for the current node.
         max_depth = max(max dept of left subtree, max depth of right subtree) + 1
     (d) Return max_depth
```

11.3 Code

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

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int maxDepth(struct node *node)
{
    if (node == NULL)
        return -1;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
```

```
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node *newNode(int data)
{
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Height of tree is %d", maxDepth(root));

    getchar();
    return 0;
}
```

11.4 Output

```
Height of tree is 2
```

12 Inorder Traversal

12.1 Objective

Write a program for Inorder Traversal of Binary Search Tree.

12.2 Algorithm

```
Traverse the left sub-tree in in-order
Visit the root
Traverse the right sub-tree in in-order
Repeat Steps 2 to 4 while TREE != NULL
INORDER(TREE -> LEFT)
Write TREE -> DATA
INORDER(TREE->RIGHT)
[END OF LOOP]
END
```

12.3 Code

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

typedef struct BST
{
    int data;
    struct BST *leftChild, *rightChild;
} node;

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

node *newNode(int data)
{
    node *temp;
    temp = (node *)malloc(sizeof(node));
    if (temp == NULL)
    {
```

```
        fprintf(stderr, "Memory failure \n ");
        exit(1);
    }
    temp->data = data;
    temp->leftChild = NULL;
    temp->rightChild = NULL;
    return temp;
}

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

    if (root == NULL)
        root = newNode(data);
    else
    {
        if (data < root->data)
            root->leftChild = insert(root->leftChild, data);
        else
            root->rightChild = insert(root->rightChild, data);
    }

    return root;
}
int main()
{

    struct node *bst = NULL;
    bst = insert(bst, 70);
    insert(bst, 10);
    insert(bst, 90);
    insert(bst, 40);
    insert(bst, 30);
    insert(bst, 60);
    inorder(bst);

    return 0;
}
```

12.4 Output

```
 10 30 40 60 70 90
```

13 Quick Sort

13.1 Objective

Write a program to Quick Sort an array

13.2 Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
if (leftmostIndex < rightmostIndex)
  pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
  quickSort(array, leftmostIndex, pivotIndex - 1)
  quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
set rightmostIndex as pivotIndex
storeIndex <- leftmostIndex - 1
for i <- leftmostIndex + 1 to rightmostIndex
if element[i] < pivotElement
  swap element[i] and element[storeIndex]
  storeIndex++
swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

13.3 Code

```c
// Quick sort in C

#include <stdio.h>

// function to swap elements
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high)
{

    // select the rightmost element as pivot
    int pivot = array[high];

    // pointer for greater element
```

```
    int i = (low - 1);

    // traverse each element of the array
    // compare them with the pivot
    for (int j = low; j < high; j++)
    {
        if (array[j] <= pivot)
        {

            // if element smaller than pivot is found
            // swap it with the greater element pointed by i
            i++;

            // swap element at i with element at j
            swap(&array[i], &array[j]);
        }
    }

    // swap the pivot element with the greater element at i
    swap(&array[i + 1], &array[high]);

    // return the partition point
    return (i + 1);
}

void quickSort(int array[], int low, int high)
{
    if (low < high)
    {

        // find the pivot element such that
        // elements smaller than pivot are on left of pivot
        // elements greater than pivot are on right of pivot
        int pi = partition(array, low, high);

        // recursive call on the left of pivot
        quickSort(array, low, pi - 1);

        // recursive call on the right of pivot
        quickSort(array, pi + 1, high);
    }
}

// function to print array elements
```

```c
void printArray(int array[], int size)
{
    for (int i = 0; i < size; ++i)
        printf("%d  ", array[i]);
    printf("\n");
}

// main function
int main()
{
    int data[] = {8, 7, 2, 1, 0, 9, 6};

    int n = sizeof(data) / sizeof(data[0]);

    printf("Unsorted Array\n");
    printArray(data, n);

    // perform quicksort on data
    quickSort(data, 0, n - 1);

    printf("Sorted array in ascending order: \n");
    printArray(data, n);
}
```

13.4 Output

```
Unsorted Array
8  7  2  1  0  9  6
Sorted array in ascending order:
0  1  2  6  7  8  9
```

14 Depth First Search

14.1 Objective

Write a program for Depth first search.

14.2 Algorithm

```
For the DFS implementation we'll put each vertex of the graph into one of two categories:
    1. Not Visited
    2. Visited
Now we'll mark each vertex as visited while avoiding cycles.
The algorithm will go as follows:
    Start by putting any one of the graph's vertices on top of a stack.
    Take the top item of the stack and add it to the visited list.
    Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
    Keep repeating steps 2 and 3 until the stack is empty.
```

14.3 Code

```c
// DFS algorithm in C
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int vertex;
    struct node *next;
};
struct node *createNode(int v);
struct Graph
{
    int numVertices;
    int *visited;
    // We need int** to store a two dimensional array.
    // Similarly, we need struct node** to store an array of Linked lists
    struct node **adjLists;
};
// DFS algo
void DFS(struct Graph *graph, int vertex)
{
    struct node *adjList = graph->adjLists[vertex];
    struct node *temp = adjList;
    graph->visited[vertex] = 1;
```

```
    printf("Visited %d \n", vertex);
    while (temp != NULL)
    {
        int connectedVertex = temp->vertex;
        if (graph->visited[connectedVertex] == 0)
            DFS(graph, connectedVertex);
        temp = temp->next;
    }
}
// Create a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
// Create graph
struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node *));
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}
// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```c
// Print the graph
void printGraph(struct Graph *graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node *temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main()
{
    struct Graph *graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    printGraph(graph);
    DFS(graph, 2);
    return 0;
}
```

14.4 Output

```
Adjacency list of vertex 0
2 -> 1 ->

Adjacency list of vertex 1
2 -> 0 ->

Adjacency list of vertex 2
3 -> 1 -> 0 ->

Adjacency list of vertex 3
2 ->

Visited 2
```

```
Visited 3
Visited 1
Visited 0
```

15 Breadth First Search

15.1 Objective

Write a program for Breadth first search.

15.2 Algorithm

```
Create a queue Q
Mark v as visited and put v into Q
While Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
```

15.3 Code

```c
// BFS algorithm in C

#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue
{
    int items[SIZE];
    int front;
    int rear;
};

struct queue *createQueue();
void enqueue(struct queue *q, int);
int dequeue(struct queue *q);
void display(struct queue *q);
int isEmpty(struct queue *q);
void printQueue(struct queue *q);

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int);
```

```c
struct Graph
{
    int numVertices;
    struct node **adjLists;
    int *visited;
};

// BFS algorithm
void bfs(struct Graph *graph, int startVertex)
{
    struct queue *q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q))
    {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

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

        while (temp)
        {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0)
            {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

// Creating a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
```

```c
}

// Creating a graph
struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

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

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue *createQueue()
{
    struct queue *q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}
```

```c
// Check if the queue is empty
int isEmpty(struct queue *q)
{
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(struct queue *q, int value)
{
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Removing elements from queue
int dequeue(struct queue *q)
{
    int item;
    if (isEmpty(q))
    {
        printf("Queue is empty");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
        {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}
```

```c
// Print the queue
void printQueue(struct queue *q)
{
    int i = q->front;

    if (isEmpty(q))
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++)
        {
            printf("%d ", q->items[i]);
        }
    }
}

int main()
{
    struct Graph *graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}
```

15.4 Output

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2
```

```
Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

List of Figures