Name: SATYAJIT GHANA **Roll Number**: 17ETCS002159

Laboratory 1

Title of the Laboratory Exercise: Data types, local variables and Random number generation

1. Introduction and Purpose of Experiment

   Students get familiar with the data types and local variables and random number generation. Basic concepts such as data types and local variables are part and parcel of almost all the c programs. Hence sound knowledge is most essential in this regard. Also, the random number generation essential for many applications, for ex. rolling a dice for many in gaming applications such as backgammon which requires a random number generation from 1 to 6.

2. Aim and Objectives

   Aim

   To design and develop a C programs using Data types, local variables and Random number generation to demonstrate the use and significate of the same in programming.

   Objectives

   At the end of this lab, the student will be able to

   - Use variables of the basic data types with proper declarations
   - Read and validate the input data
   - Generate random numbers for any application

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Questions

Name: SATYAJIT GHANA **Roll Number**: 17ETCS002159

Demonstrate the use of data types, local variables and Random numbers by designing appropriate algorithms for the below problems. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of the algorithm. Describe your learning along with the limitations of overall approach if any. Suggest how these can be overcome.

i. Write a C program to illustrate random number generation. Modify the program to generate a random number between 75 to 85

ii. Write a C program to find sum of n elements, allocate memory dynamically using malloc() and calloc() function. Modify the program include both the allocation strategies in a single program.

iii. Combine both random number generation and memory allocation in a single program to demonstrate the allocation of random number of memory blocks.

1. Calculations/Computations/Algorithms

```
Step 1: Start

Step 2: Declare variables min, max, n

Step 3: seed random with current system time

Step 4: Read min, max and n from user

Step 5: Declare i <- 0

Step 6: Repeat steps until i = n

   6.1 print rand()%(max – min +1 ) + min
   6.2 i <- i + 1
Step 7: Stop
```

2. Presentation of Results

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "timeutil.h"

/*
 * Author   : Satyajit Ghana
 * Reg.No   : 17ETCS002159
 * */

void help(void);

char *name;

int main(int argc, char **argv) {
    int n = 1, min = 0, max = RAND_MAX;
    char c;
    name = *argv;
    /* Seed rand with current system time */
    srand(time(0));

    while (--argc > 0 && (++argv)) {
        switch((c = **argv)) {
            case 'l':
                min = atoi(*++argv);
                max = atoi(*++argv);
                argc -= 2;
                break;
            case 'n':
                n = atoi(*++argv);
                argc--;
                break;
            case 'h':
                help();
                return 0;
            default:
                printf("%s: invalid character : %c\n", name, c);
                break;
        }
    }

    if (argc > 0) {
        help();
    }

    clock_t begin = clock();
    while(n--) {
        printf("%d\n", (rand()%(max - min + 1)) + min);
    }
    clock_t end = clock();
    printf("\nExecution Time : %.6fs\n", time_exec(begin, end));

    return 0;
}

void help() {
        printf("\nUSAGE: %s n <number> l <lower> <upper>\n"
        "\t n <numer>: quantity of pseudo-numbers to generate\n"
        "\t l <lower_limit> <upper_limit>\n"
        "\t example: %s 10 l 2 10\n"
        "\t\tgenerates 10 pseudo-random numbers from [2, 10]\n", name, name);
}
```

Figure 1.1: Source Code

Figure 1.2 : Build Successful

Figure 1.3 : Run Successful

3. Analysis and Discussions

`rand()` is an library function that is defined in `rand.c` as

```
/* Return a random integer between 0 and RAND_MAX.  */
int
rand (void)
{
  return (int) __random ();
}
```

If we take a look at how `__random` works

```
/* If we are using the trivial TYPE_0 R.N.G., just do the old linear
congruential bit.  Otherwise, we do our fancy trinomial stuff, which
is the same in all the other cases due to all the global variables
that have been set up.  The basic operation is to add the number at
the rear pointer into the one at the front pointer.  Then both
pointers are advanced to the next location cyclically in the table.
The value returned is the sum generated, reduced to 31 bits by
throwing away the "least random" low bit. Note: The code takes
advantage of the fact that both the front and rear pointers can't
```

```
wrap on the same call by not testing the rear pointer if the front
one has wrapped.  Returns a 31-bit random number. */
```

As the source-code of `__random` from glibc states, it uses Linear Congruential Generator to generate the pseudo-random numbers, to generate a different set of numbers at every run event, the algorithm is seeded with a seeding value, and here we take the system time to do so.

The generator is defined by:

$$X_{n+1} = (aX_n + c)\ mod\ m$$

Where

$m, 0 < m - modulus$
$a, 0 < a < m - multiplier$
$c, 0 \leq c < m - increment$
$X_0, 0 \leq X_0 < m - \boldsymbol{the\ seed}$

4. Conclusions

A benefit of LCGs is that with appropriate choice of parameters, the period is known and long. Although not the only criterion, too short a period is a fatal flaw in a pseudorandom number generator.

While LCGs are capable of producing pseudorandom numbers which can pass formal tests for randomness, this is extremely sensitive to the choice of the parameters m and a. For example, `a = 1` and `c = 1` produces a simple modulo-m counter, which has a long period, but is obviously non–random.

C uses LCGs internally inside rand for generating these pseudorandom numbers, this can be verified by looking at the definition written in `glibc`. This cannot be used for cryptographic purposes at all, due to the fact that two systems producing generating random numbers using this algorithm may generate the same numbers arising conflicts and encryption will fail.

Nevertheless this algorithm still holds good for small applications such as in an embedded systems where memory is severely limited.

5. Comments

1. Limitations of Experiments

`rand` that used LCGs cannot be used for random number simulations in large scale operations such as a large scale Monte-Carlo Simulation. This has very high memory usage and a large amount of passes needs to be done to create close to random generator.

As the sequence of pseudo random numbers is fixed, there is a correlation among sequential random numbers. This effect can prove to be a major problem if one uses random numbers to create points in a k-dimensional space (as with Monte Carlo methods). The points will not fill up the space but will line up on n-dimensional hyperplanes. The number of hyperplanes is roughly the kth root of the constant c.

2. Limitations of Results

As the definition of `rand` states, it is a pseudo random generator and the randomness of the generated number depends on the seed value.

The random numbers generated are however limited from `0` to `RAND_MAX` and this generates only Integer values.

3. Learning happened

We learnt that `rand()` uses LCG to generate pseudo random numbers. And the generator needs to be seeded with some value to generate a new set of numbers at two different runtimes. `rand()` can only output integers and does not support generation of decimal values.

4. Recommendations

The code can be further generalized to generate random numbers, such as Floats and Doubles, and an implementation should be made to generate random characters and/or strings.

Name: SATYAJIT GHANA CSE SEC-C Reg No: 17ETCS002159

Laboratory 2

Title of the Laboratory Exercise: Stacks

Introduction and Purpose of Experiment

Stacks and queues are very important data structures used in many real time applications. This experiment introduces the development of Stack ADT.

1. Aim and Objectives

   Aim

   - To develop stack ADT and to use them for string applications

   Objectives

   At the end of this lab, the student will be able to

   - Design and develop and use stack and demonstrate its operations

2. Pseudo Code

```
1. Stack – Push
   If top > MAX
           Print Stack Overflow
           End
   Else
           Set top = top + 1
           Set stack[top] = value
   End

2. Stack – Pop
   If top  < 0
           Print Stack Underflow
           End
   Else
           Set top = top - 1
           Return stack[top + 1]
   End

3. Stack – Display
   Set i = 0
   While i < top
           Print stack[i]
           Set i = i +1;

   End
```

3. Implementation in C

```c
void push(Stack* mystack, void* data) {

    if (mystack -> top >= mystack -> MAX) {
        printf("\n*Stack Overflow Detected !*\n");
        return;
    }
    mystack -> data = realloc(mystack -> data, (mystack -> top + 2) * sizeof *(mystack -> data));
    if (mystack -> data != NULL) (mystack -> data)[(++mystack -> top)] = data;
    else printf("\n*cannot allocate memory !*\n");
}

void pop(Stack* mystack) {
    if (mystack -> top -1 < 0) {
        printf("\n*Stack Underflow detected !*\n");
        return;
    }
    mystack -> top--;
    /* Resize the data array as the elements are removed */
    mystack -> data = realloc(mystack -> data, (mystack -> top + 1) * sizeof *(mystack-> data));
    /* Print the removed data */
    printf("removed");
    ds((char*)(mystack->data)[mystack -> top+1]);
}

void display(Stack* mystack) {
    for (int i = mystack -> top  ; i >= 0 ; i--) ds(*(char**)(mystack->data + i))
}
```

Figure 1 Stack

Figure 2 Queue

4. Presentation of Results

```
--- STACKS USING DYNAMIC ALLOCATIONS ---
1.Push 2.Pop 3.Display 4.Exit
Enter your choice : 1
Enter your data : Satyajit Ghana

--- STACKS USING DYNAMIC ALLOCATIONS ---
1.Push 2.Pop 3.Display 4.Exit
Enter your choice : 3
DEBUG--**(char**)(mystack->data + i) : Satyajit Ghana*

--- STACKS USING DYNAMIC ALLOCATIONS ---
1.Push 2.Pop 3.Display 4.Exit
Enter your choice : 2
removed
DEBUG--*(char*)(mystack->data)[(mystack -> top)+1] : Satyajit Ghana*
```

Explanation:

The list of choices are presented to the user, which are push, pop, display and exist, the first three are the basic operations that can be performed on a stack. 1.Push is selected and then the data to be pushed is taken as terminal input, which is treated as a string, since the aim is to make a Stack ADT that can work with strings, then the push function is called, which checked if the stack is full or not, if not then the data is added to the stack, and the value of the top is incremented. Then the next option 3. Display is selected to display all the data stored in the stack, this calls the display function, which prints the data stored from 0 to top, if the value of top is less than 0 then the stack is empty and nothing is printed. Then the option 2. Pop is selected, this removes the data from the top of the stack, hence the value of top is decremented and the data that was deleted is printed on the screen. The whole logic to Pop being that the last element added is removed first, or Last-In-First-Out.

5. Conclusions

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal. In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

Name: SATYAJIT GHANA CSE SEC-C Reg No: 17ETCS002159

Laboratory 3

Title of the Laboratory Exercise: Queues

Introduction and Purpose of Experiment

Queues are very important data structures used in many real time applications. This experiment introduces the development Queue ADT.

1. Aim and Objectives

   Aim

   - To develop Queue ADT and to use them for string applications

   Objectives

   At the end of this lab, the student will be able to

   - Design and develop and use queue and demonstrate its operations

2. Pseudo Codes

```
1. Queue – Enqueue
   If tail > MAX
         Print Queue Overflow
         End
   Else
         Set tail = tail + 1
         Set queue[tail] = value
   End

2. Queue – Dequeue
   If head  > tail or tail == -1
         Print Queue Underflow
         End
   Else
         Set head = head + 1
         Return queue[head - 1]
   End

3. Queue – Display
   Set i = head
   While i < tail
         Print queue[i]
         Set i = i +1;
   End
```

3. Implementation in C

```c
void enqueue(Queue* myqueue, void* data) {

    if (myqueue -> tail >= myqueue -> MAX) {
        printf("\n*Queue Overflow Detected !*\n");
        return;
    }
    myqueue -> data = realloc(myqueue -> data, (myqueue -> tail + 2) * sizeof *(myqueue -> data));
    if (myqueue -> data != NULL) {
        if (myqueue -> head == -1) {
            myqueue -> head = 0;
        }
        (myqueue -> data)[++myqueue -> tail] = data;
    } else {
        printf("\n*cannot allocate memory !*\n");
        return;
    }
}

void dequeue(Queue* myqueue) {
    if (myqueue -> head > myqueue -> tail) {
        printf("\n*Queue Underflow detected !*\n");
        return;
    }
    myqueue -> head ++;
    printf("removed"); ds((char*)(myqueue->data)[myqueue -> head - 1]); /* Remove the data */
}

void display(Queue* myqueue) {
    if (myqueue -> head > myqueue -> tail || myqueue -> head == -1) {
        printf("\nQueue is Empty\n");
        return;
    }
    for (int i = myqueue -> head  ; i <= myqueue -> tail ; i++) {
        ds(*(char**)(myqueue->data + i)) /* Display the data */
    }
}
```

Figure 1 Queue

4. Presentation of Results

```
--- QUEUES USING DYNAMIC ALLOCATIONS ---
1.Enqueue 2.Dequeue 3.Display 4.Exit
Enter your choice : 1
Enter your data : Satyajit Ghana

--- QUEUES USING DYNAMIC ALLOCATIONS ---
1.Enqueue 2.Dequeue 3.Display 4.Exit
Enter your choice : 3
DEBUG--**(char**)(myqueue->data + i) : Satyajit Ghana*

--- QUEUES USING DYNAMIC ALLOCATIONS ---
1.Enqueue 2.Dequeue 3.Display 4.Exit
Enter your choice : 2
removed
DEBUG--*(char*)(myqueue->data)[myqueue -> head - 1] : Satyajit Ghana*
```

Explanation:

The list of choices are presented to the user, which are enqueuer, dequeue, display and exit. The first option 1. Enqueue is selected, the user is then asked for the data to input, the input is taken from the terminal as a String, since we are making a program for String Queue, the data taken is sent to enqueuer method, which adds it to the tail of the queue, the basic principle being that the first element is the head and the last element is the tail. Then the next option 3. Display is selected to view the current queue, which calls the display function, the elements from head to tail are displayed, i.e. the elements that are added first are displayed first and the elements added in the end are displayed at the end. Then the option 2. Dequeue is selected which removed the element from the head, hence incrementing the head pointer or the head data. The removed element is then displayed on the screen.

5. Conclusions

The simplest two search techniques are known as Depth-First Search (DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.

Breadth-First Search with a Queue, in breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

Name: SATYAJIT GHANA CSE SEC-C Reg No: 17ETCS002159

Laboratory 4

Title of the Laboratory Exercise: Infix to Postfix

Introduction and Purpose of Experiment

Infix is a more human readable form of expression, but the operator priority needs to be checked properly for evaluating it, hence postfix and prefix notations are used that make it easier to evaluate an expression without worrying about the precedence of the operator, since the expression is now already organizes in the required form.

1. Aim and Objectives
   - To create a method to convert an infix expression to a postfix expression.

2. Pseudo Codes

```
1. Infix2Postfix
   1. Begin
   2. for (each token in the expression)
      a. if (token is a operand) output.add(token)
      b. else if (token == '(') push(token)
      c. else if (token == ')')
         i. while (!stack.isEmpty() && stack.peek != '(') output.add(stack.pop())
        ii. stack.pop()
      d. else
         i. while (!stack.isEmpty() && precedence(token) <= precedence(stack.peek())
            output.add(stack.pop())
        ii. stack.push(token)
   3. while (!stack.isEmpty()) output.add(stack.pop())
   4. print output
   5. End
```

3. Implementation in C

```
int precedence(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return -1;
    }
}

int infix2postfix(char *expr) {
    const char delim[] = " \n";

    Vector* output = newMinimalVector(string_compare, print_one_string);
    Stack* stack = newStack(strlen(expr));

    char* token = strtok(expr, delim);

    while (token != NULL) {
        if (isdigit(token[0]) || strlen(token) > 1) { /* operand was encountered */
            output -> add(output, token);
        } else if (strlen(token) == 1
            && token[0] == '(') {
            stack -> push(stack, token);
        } else if (strlen(token) == 1
            && token[0] == ')') {
            while (!stack -> isEmpty(stack) && *(char*)(stack -> peek(stack)) != '(' ) {
                output -> add(output, stack -> pop(stack));
```

```
            }
            stack -> pop(stack);
        } else { /* operator was encountered */
            while (!stack -> isEmpty(stack) && precedence(token[0]) <=
precedence(*(char*)(stack -> peek(stack)))) {
                output -> add(output, stack -> pop(stack));
            }
            stack -> push(stack, new_string(token));
        }
        token = strtok(NULL, delim);
    }

    while (!stack -> isEmpty(stack)) {
        output -> add(output, stack -> pop(stack));
    }

    output -> print(output);

    return 1;
}
```

4. Presentation of Results

```
Enter an Infix Expression : 3 + 2 * 1 - 5
Given Infix Expression : 3 + 2 * 1 - 5
Postfix Expression : 3 2 1 * + 5 -
```

Explanation:

The Infix Expression is input as a string, which is a line, and read until a new line is encountered, this string is the infix expression that is to be converted into a postfix expression. The string expression is used as a formal argument for infix2postfix function.

The expression string is then tokenized using ‘ ’ as the delimiter, since the operator and the operands are separated using spaces. The tokenized input is then parsed using the infix-to-postfix pseudocode/algorithm. A Stack is created to store the operators, if the token is a operand it is directly added to the output, if there is an opening parenthesis then it is pushed to the stack, if a closing parenthesis is found then elements from the stack are popped until an opening parenthesis is found. For operators being encountered, there are added to the stack and any operator with greater than or equal to the precedence of the current token operator is popped and added to the output. The output is stored in a vector, which is then dispayed.

Laboratory 5

Title of the Laboratory Exercise: Linked Lists

Introduction and Purpose of Experiment

A Linked List is a linear form of Data Structure, which is dynamically allocated, and size of can be increased to any length, provided there's memory available to allocate it.

1. Aim and Objectives
   - To develop a Linked List for Strings and provide basic linked list operations
2. Pseudo Codes

```
1. newLink
       1. Begin
       2. newList = malloc(sizeof *newList)
       3. newList -> next = NULL
       4. newList -> data = data
       5. return newList
       6. End
2. add_to_end_of_list
       1. Begin
       2. if (head == NULL) head = toad; return;
       3. toadd -> next = head
       4. head = toadd
       5. End
3. add_to_beg_of_list
       1. Begin
       2. if (head == NULL) head = toadd; return;
       3. add_to_beg_list(head -> next, toadd)
       4. End
4. remove_from_end_of_list
       1. Begin
       2. if (head == NULL) print "List is Empty"
       3. temp = head
       4. head = head -> next
       5. return temp
       6. End
5. print_list
       1. Begin
       2. if (head == NULL) return;
       3. print_list(head -> next)
       4. print head -> data
       5. End
```

3. Implementation in C

linked_list.h

```
struct LinkedList {
    void* data;
    struct LinkedList* next;
};
typedef struct LinkedList LinkedList;
```

linked_list.c

```
LinkedList* newLink(void * data) {
    LinkedList* newList = malloc(sizeof *newList);
    newList -> next = NULL; newList -> data = data; return newList;
}
void add_at_end_of_list(LinkedList** head, LinkedList* toadd) {
    if (*head == NULL) {*head = toadd; return;} toadd -> next = (*head); *head = toadd;
}
void add_at_beg_of_list(LinkedList** head, LinkedList* toadd) {
    if (*head == NULL) {*head = toadd;return;} add_at_beg_of_list(&((*head) -> next), toadd);
}
LinkedList* remove_from_end_of_list(LinkedList** head) {
    if (*head == NULL) printf("\nList is Empty !\n");
    LinkedList* temp = *head; *head = (*head) -> next; return temp;
}
LinkedList* remove_from_beg_of_list(LinkedList** head) {
```

```
    if (*head == NULL) printf("\nList is Empty !\n");
    LinkedList* temp = *head; LinkedList* prev = NULL;
    while (temp -> next != NULL) { prev = temp; temp = temp -> next;}
    if (prev == NULL) return NULL; prev -> next = NULL; return temp;
}
void print_list(LinkedList* head) {
    if (head == NULL) {return;} print_list(head -> next); ds((char*)(head -> data));
}
```

main.c

```
void menu();
int main() {
    LinkedList* head = NULL;
    char* input;
    while(1) {menu();int choice = next_int();
        switch (choice) {
            case 1:printf("\nEnter your Data : ");get_line(&input);
                add_at_end_of_list(&head, newLink(input));break;
            case 2:printf("\nEnter your Data : ");get_line(&input);
                add_at_beg_of_list(&head, newLink(input));break;
            case 3: {LinkedList *deleted = remove_from_end_of_list(&head);
                if (deleted != NULL) {printf("\nDeleted Link : ");print_link(deleted);}}
                break;
            case 4: {LinkedList *deleted = remove_from_beg_of_list(&head);
                if (deleted != NULL) {printf("\nDeleted Link : ");print_link(deleted);}}
                break;
            case 5: print_list(head);break;
            case 6:return 0;
            default:printf("\nInvalid Choice !\n");
          }
    }
}
void menu() {
    printf("\n---------- Linked Lists ----------------------\n"
     "1.\tAdd at the End of List\n" "2.\tAdd at the Beginning of List\n"
     "3.\tRemove from the End of the List\n" "4.\tRemove from the Beginning of the List\n"
     "5.\tDisplay the List\n" "6.\tExit\n" "\nYour Choice : "); }
```

4. Explanation

This is a menu driven program where Data can be added to the List at the beginning and at the end, the data/links can be removed from both beginning and end. The head pointer variable points to the beginning of the List. The Linked list implemented here is a singly linked list, hence the head pointer is very important. The user input is taken as a string and added to the list, although the list implemented here can take in any kind of data since it's a void*. The data is sent as a formal parameter to newLink which creates a new Link using malloc and returns it, this Link is then passes as a formal parameter to add to the list, either at the end or at the beginning. In add_at_end_of_list, the toAdd is made to point to what head was pointing to and the head pointer now points to the toAdd. In add_at_beg_of_list, a recursive function is used where the head -> next is passed as a formal parameter, when the value is found to be NULL the toAdd element is added there. Deletion works similarly, the element to be deleted from the linked list is found and the link before it is made to point to NULL and the deleted link is returned. When a link from the middle is to be removed then the link for the link to be deleted is made to point to the link that the toBeDeleted link was pointing to, pretty straight forward and simple. To print the linked list another recursive algorithm is used, since the structure itself is recursive in nature the algorithms too become recursive, to hence print, the head pointer is passes as the formal parameter, and head -> next is used a formal parameter inside the function, if it is NULL then nothing is printed, else the data stored in that link is printed.

Laboratory 6

Title of the Laboratory Exercise: Stacks and queues

Introduction and Purpose of Experiment

Stacks and queues are very important data structures used in many real time applications. This experiment introduces the development of stack and queue ADT and applying them together.

1. Aim and Objectives

   Aim

   - To develop stack and queue ADT and to use them for string applications

   Objectives

   At the end of this lab, the student will be able to

   - Design and develop and use stack and demonstrate its operations
   - Design and develop and use queue and demonstrate its operations

2. Experimental Procedure
   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

3. Calculations/Computations/Algorithms

Algorithms:

<u>Push:</u>

Step 1 : Start

Step 2 : Create a new node with the value to be inserted.

Step 3 : If the stack is empty, set the next of the new node to null.

Step 4 : If the stack is not empty, set the next of the new node to top.

Step 5 : Finally, increment the top to point to the new node.

Step 6 : Stop

Pop:

Step 1 : Start

Step 2 : If the stack is empty, terminate the method as it is stack underflow.

Step 3 : If the stack is not empty, increment the top to point to the next node.

Step 4 : Hence the element pointed to by the top earlier is now removed.

Step 5 : Stop

Enqueue:

Step 1 : Start

Step 2 : Create a new node with the value to be inserted.

Step 3 : If the queue is empty, then set both front and rear to point to newNode.

Step 4 : If the queue is not empty, then set next to the rear of the new node and the rear to point to the new node.

Step 5 : Stop

Dequeue:

Step 1 : Start

Step 2 : If the queue is empty, terminate the method.

Step 3 : If the queue is not empty, increment the front to point to the next node.

Step 4 : Finally, check if the front is null, then set rear to null also. This signifies an empty queue.

Step 5 : Stop

Implementation in C:

```c
#include <stdio.h>
#include "vector.h"
#include "input_helper.h"
#include "linked_list.h"

enum {
    STACK=1,
    QUEUE=2
```

```
};

void stack_menu();
void main_menu();
void queue_menu();

int main() {
    LinkedList* head = NULL;
    int choice;
    char* input;
    main_menu();
    choice = next_int();
    switch (choice) {
        case STACK: {
            while (1) {
                stack_menu();
                choice = next_int();
                switch (choice) {
                    case 1: {
                        printf("\nEnter your Data : ");
                        get_line(&input);
                        add_at_end_of_list(&head, newLink(input));
                    }
                        break;
                    case 2: {
                        LinkedList* deleted = remove_from_end_of_list(&head);
                        if (deleted != NULL) {
                            printf("\nDeleted : ");
                            print_link(deleted);
                        }
                    }
                        break;
                    case 3: {
                        print_list(head);
                    }
                        break;
                    case 4:
                        return 0;
                    default:
                        printf("\nInvalid Choice !\n");
                }
            }
        }
        case QUEUE: {
            while (1) {
                queue_menu();
                choice = next_int();
                switch (choice) {
                    case 1: {
                        printf("\nEnter your Data : ");
                        get_line(&input);
                        add_at_end_of_list(&head, newLink(input));
                    }
                        break;
                    case 2: {
                        LinkedList* deleted = remove_from_beg_of_list(&head);
                        if (deleted != NULL) {
                            printf("\nDeleted : ");
                            print_link(deleted);
```

```
                        }
                    }
                        break;
                    case 3: {
                        print_list(head);
                    }
                        break;
                    case 4:
                        return 0;
                    default:
                        printf("\nInvalid Choice !\n");
                }
            }
        }
        default:
            printf("\nWrong Choice !\n");
            break;
    }
}

void stack_menu() {
    printf("\n---------- Stacks using Linked Lists ----------------------\n"
           "1.\tPush\n"
           "2.\tPop\n"
           "3.\tDisplay\n"
           "4.\tExit\n"
           "\nYour Choice : ");
}

void queue_menu() {
    printf("\n---------- Queues using Linked Lists ----------------------\n"
           "1.\tEn-queue\n"
           "2.\tDe-queue\n"
           "3.\tDisplay\n"
           "4.\tExit\n"
           "\nYour Choice : ");
}

void main_menu() {
    printf("\n-----------STACK AND QUEUES USING LINKED LISTS------\n"
           "1.\tStacks\n"
           "2.\tQueues\n"
           "\nYour Choice : ");
}
```

4. Presentation of Results

```
-----------STACK AND QUEUES USING LINKED LISTS------
1.    Stacks
2.    Queues

Your Choice : 1
1
```

```
---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice : 1
10
1
10

Enter your Data :
---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice : 1
1

Enter your Data : 123
123

---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice : 3
3

DEBUG--*(char*)(head -> data) : 10*

DEBUG--*(char*)(head -> data) : 123*

---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice : 2
2

Deleted :
DEBUG--*(char*)(link -> data) : 123*

---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice : 3
```

```
3

DEBUG--*(char*)(head -> data) : 10*

---------- Stacks using Linked Lists ----------------------
1.    Push
2.    Pop
3.    Display
4.    Exit

Your Choice :
```

5. Analysis and Discussions

A stack is a linear data structure which allows the adding and removing of elements in a particular order. New elements are added at the top of the stack. If we want to remove an element from the stack, we can only remove the top element from the stack. Since it allows insertion and deletion from only one end and the element to be inserted last will be the element to be deleted first, it is called the 'Last in First Out' data structure (LIFO). Stack can be implemented using both arrays and linked lists. The limitation, in the case of an array, is that we need to define the size at the beginning of the implementation. This makes our stack static. It can also result in “stack overflow” if we try to add elements after the array is full. So, to alleviate this problem, we use a linked list to implement the stack so that it can grow in real time. The time complexity of Push and Pop are O(1).

A queue is also a linear data structure where insertions and deletions are performed from two different ends. A new element is added from the rear of the queue and the deletion of existing elements occurs from the front. Since we can access elements from both ends and the element inserted first will be the one to be deleted first, the queue is called the 'First in First Out' data structure (FIFO). Similar to stack, the queue can also be implemented using both arrays and linked lists. But it also has the same drawback of limited size. Hence, we will be using a linked list to implement the queue.

6. Conclusions

Stack can be used to implement back/forward button in the browser. Undo feature in the text editors are also implemented using Stack. It is also used to implement recursion. Call and return mechanism for a method uses Stack. It is also used to implement backtracking.

CPU scheduling in Operating system uses Queue. The processes ready to execute and the requests of CPU resources wait in a queue and the request is served on first come first serve basis.

Data buffer - a physical memory storage which is used to temporarily store data while it is being moved from one place to another is also implemented using Queue.

Laboratory 7

Title of the Laboratory Exercise: Binary trees

1. Introduction and Purpose of Experiment

   Linear organization used on arrays, vectors, stacks and queues become inefficient in some applications. Then we choose the structures which provide non-linear organization. Binary tree is a non-linear data structure used in many applications. This experiment introduces binary search trees and its applications.

2. Aim and Objectives

   Aim

   - To develop Binary search tree ADT

   Objectives

   At the end of this lab, the student will be able to

   - Design binary tree ADT
   - Use binary tree ADT and illustrate binary tree traversals

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Implementation:

```
#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;
};

/* 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);
```

```
}

/* Given a binary tree, print its nodes according to the
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
     if (node == NULL)
        return;

     // first recur on left subtree
     printPostorder(node->left);

     // then recur on right subtree
     printPostorder(node->right);

     // now deal with the node
     printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first recur on left child */
     printInorder(node->left);

     /* then print the data of node */
     printf("%d ", node->data);

     /* now recur on right child */
     printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first print data of node */
     printf("%d ", node->data);

     /* then recur on left sutree */
     printPreorder(node->left);

     /* now recur on right subtree */
     printPreorder(node->right);
}

/* Driver program to test above functions*/
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("\nPreorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\nInorder traversal of binary tree is \n");
    printInorder(root);

    printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}
```

5. Presentation of Results

```
Preorder traversal of binary tree is

1 2 4 5 3

Inorder traversal of binary tree is

4 2 5 1 3

Postorder traversal of binary tree is

4 5 2 3 1
```

6. Analysis and Discussions

Time Complexity: O(n)

Let us see different corner cases.

Complexity function T(n) — for all problem where tree traversal is involved — can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Let's do an analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

……

……..

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of $T(0)$ will be some constant say d. (traversing a empty tree will take some constants time)

$$T(n) = n(c+d)$$

$$T(n) = \Theta(n)\ (Theta\ of\ n)$$

Case 2: Both left and right subtrees have equal number of nodes.

$$T(n) = 2T(|\_n/2\_|) + c$$

7. Conclusions

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Laboratory 8

Title of the Laboratory Exercise: Binary trees

1. Introduction and Purpose of Experiment

   Linear organization used on arrays, vectors, stacks and queues become inefficient in some applications. Then we choose the structures which provide non-linear organization. Binary tree is a non-linear data structure used in many applications. This experiment introduces binary search trees and its applications.

2. Aim and Objectives

   Aim

   - To develop Binary search tree ADT

   Objectives

   At the end of this lab, the student will be able to

   - Design binary tree ADT
   - Use binary tree ADT to illustrate the binary tree operations
   - Use binary tree ADT and illustrate binary tree traversals

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

1. Insertion Algorithm

Step 1 : Start

Step 2 : check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

Step 3 : if a new value is less, than the node's value:

Step 3.1 : if a current node has no left child, place for insertion has been found;

Step 3.2 : otherwise, handle the left child with the same algorithm.

Step 4 : if a new value is greater, than the node's value:

Step 4.1 : if a current node has no right child, place for insertion has been found;

Step 4.2 : otherwise, handle the right child with the same algorithm.

Step 5: Stop

2. Deletion Algorithm

Step 1 : Start

Step 2 : Node to be deleted is leaf: Simply remove from the tree.

Step 3 : Node to be deleted has only one child: Copy the child to the node and delete the child.

Step 4 : Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used. The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Step 5: Stop

Implementation:

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

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
```

```
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
```

```
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
```

```
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

5. Presentation of Results

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

6. Analysis and Discussions

Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

7. Conclusions

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is

less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node.

Laboratory 9

Title of the Laboratory Exercise: Graph algorithms

1. Introduction and Purpose of Experiment

   Graph algorithms such Depth First Search (DFS) and Breadth First Search (BFS) are very important in many graph applications which are used to traverse the graphs, check for graph connectivity and to find spanning trees etc. This experiment introduces the applications DFS and BFS.

2. Aim and Objectives

   Aim

   - To apply DFS and BFS algorithm for graph applications

   Objectives

   At the end of this lab, the student will be able to

   - Apply DFS and BFS for graph applications such as graph connectivity check
   - Compare the efficiency of both

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C/java language
   iv. Compile the C/java program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Questions

   Design and develop algorithms to check whether given graph is connected or not using DFS and BFS. Tabulate the output for various inputs and verify against expected values. Analyse the

efficiency of both the algorithms. Describe your learning along with the limitations of both, if any. Suggest how these can be overcome.

5. Calculations/Computations/Algorithms

1. BFS Algorithm

Step 1 : Start

Step 2 : Start by putting any one of the graph's vertices at the back of a queue.

Step 3 : Take the front item of the queue and add it to the visited list.

Step 4 : Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Step 5 : Keep repeating steps 2 and 3 until the queue is empty.

Step 6 : Stop

2. DFS Algorithm

Step 1 : Start

Step 2 : Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Step 3 : If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Step 4 : Repeat Rule 1 and Rule 2 until the stack is empty.

Step 5 : Stop

Implementation :

```c
#include <stdio.h>

#define MAX 10

int G[MAX][MAX]={}, visited[MAX]={}, n, queue[MAX]={}, front = 0, rear = -1;

void menu();
void reset_visited();
void read_adj_matrix();

// BFS and DFS
void DFS(int);
void BFS(int);

int main() {
    menu();

    return 0;
}
```

```c
void menu() {
    while (1) {
        int choice;
        printf("\nBFS and DFS on Graph"
               "\n1.\tEnter Adjacency Matrix"
               "\n2.\tDFS of the Graph"
               "\n3.\tBFS of the Graph"
               "\n4.\tExit"
               "\nYour Choice : ");
        scanf("%d", &choice);
        switch (choice) {
            case 1 :
                read_adj_matrix();
                break;
            case 2 :
                reset_visited();
                printf("\n");
                DFS(0);
                break;
            case 3 :
                reset_visited();
                BFS(0);
                printf("\nThe nodes which are reachable are : \n");
                for (int i = 0 ; i < n ; i++) {
                    if (visited[i]) {
                        printf(" %d", i);
                    } else {
                        printf("\n Not all nodes are reachable, BFS not possible.\n");
                        break;
                    }
                }
                break;
            case 4 :
                return;
            default :
                printf("\nWrong Choice !");
        }
    }
}

void reset_visited() {
    front = 0;
    rear = -1;
    for (int i = 0 ; i < MAX ; i++)
        visited[i] = 0;
}

void read_adj_matrix() {
    printf("\nEnter the number of vertices : ");
    scanf("%d", &n);

    printf("\nEnter the adjacency matrix of the graph : ");

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

```
}

void DFS(int i) {
    printf(" %d", i);
    visited[i] = 1;

    for (int j = 0 ; j < n ; j++) {
        if (!visited[j] && G[i][j] == 1) {
            DFS(j);
        }
    }
}

void BFS(int v) {
    for (int i = 0 ; i < n ; i++) {
        if (!visited[i] && G[v][i]) {
            queue[++rear] = i;
        }
    }

    if (front <= rear) {
        visited[queue[front]] = 1;
        BFS(queue[front++]);
    }
}
```

6. Presentation of Results

```
BFS and DFS on Graph

1.    Enter Adjacency Matrix

2.    DFS of the Graph

3.    BFS of the Graph

4.    Exit

Your Choice : 1

Enter the number of vertices : 8

Enter the adjacency matrix of the graph :


0 1 1 1 1 0 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 0 1 0

1 0 0 0 0 0 1 0

0 1 1 0 0 0 0 1

0 0 0 1 1 0 0 1
```

```
0 0 0 0 0 1 1 0

BFS and DFS on Graph
1.    Enter Adjacency Matrix
2.    DFS of the Graph
3.    BFS of the Graph
4.    Exit
Your Choice : 2
 0 1 5 2 7 6 3 4

BFS and DFS on Graph
1.    Enter Adjacency Matrix
2.    DFS of the Graph
3.    BFS of the Graph
4.    Exit
Your Choice : 3
The nodes which are reachable are :
 0 1 2 3 4 5

BFS and DFS on Graph
1.    Enter Adjacency Matrix
2.    DFS of the Graph
3.    BFS of the Graph
4.    Exit
Your Choice : 4

Process finished with exit code 0
```

7. Analysis and Discussions

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

8. Conclusions

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Laboratory 10

Title of the Laboratory Exercise: Sorting and Searching Algorithms

1. Introduction and Purpose of Experiment

   Sorting provide us with means of organising information to facilitate the retrieval of specific data. Searching methods are designed to take advantage of the organisation of information. By solving these problems students will be able to use sorting algorithms to sort a randomly ordered set of numbers, and search for key element.

2. Aim and Objectives

   Aim

   - To develop programs for sorting algorithms

   Objectives

   At the end of this lab, the student will be able to

   - Create C programs using sorting algorithms such as **insertion sort**
   - Create C programs using sorting algorithms such as **merge sort**
   - Create C program for **linear search**
   - Create C program for **binary search**

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

1. Linear Search

```
Step 1 : Start
Step 2 : Start from the leftmost element of arr[] and one by one compare x
with each element of arr[]
Step 2.1 : If x matches with an element, return the index.
Step 2.2 : If x doesn't match with any of elements, return -1.
Step 3: Stop
```

2. Binary Search

```
Step 1 : Start
Step 2 : Compare x with the middle element.
Step 2.1 : If x matches with middle element, we return the mid index.
Step 2.2 : Else If x is greater than the mid element, then x can only lie in
right half subarray after the mid element. So we recur for right half.
Step 2.3 : Else (x is smaller) recur for the left half.
Step 3: Stop
```

3. Merge Sort

```
Step 1 : Start
Step 2 : If r > l
Step 2.1 : Find the middle point to divide the array into two halves:
            middle m = (l+r)/2
Step 2.2 : Call mergeSort for first half:
            Call mergeSort(arr, l, m)
Step 2.3 : Call mergeSort for second half:
            Call mergeSort(arr, m+1, r)
Step 2.4 : Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
Step 3: Stop
```

4. Insertion Sort

```
Step 1 : Start
// Sort an arr[] of size n
insertionSort(arr, n)
Step 2 : Loop from i = 1 to n-1.
```

Step 3: Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Step 4: Stop

```c
#include <stdio.h>

#define MAX 100

int list[MAX] = {1, 2, 3, 4, 5}, n = 5;

void menu();
void load_data();
void linear_search();
void binary_search();
int bsearch(int [], int, int, int);
void merge_sort(int [], int, int);
void merge(int [], int, int, int);
void insertion_sort(int [], int);

void print_array(int[], int);

int main() {
    menu();
    return 0;
}

void menu() {
    while (1) {
        int choice;
        printf("\nSearching and Sorting"
               "\n1.\tEnter List Data"
               "\n2.\tLinear Search"
               "\n3.\tBinary Search"
               "\n4.\tMerge Sort"
               "\n5.\tInsertion Sort"
               "\n6.\tExit"
               "\nEnter your choice : ");
        scanf("%d", &choice);

        switch (choice) {
            case 1 :
                load_data();
                break;
            case 2 :
                linear_search();
                break;
            case 3 :
                binary_search();
                break;
            case 4 :
                printf("\nBefore Sorting : \n");
                print_array(list, n);
                merge_sort(list, 0, n-1);
                printf("\nAfter Sorting : \n");
                print_array(list, n);
                break;
            case 5 :
                printf("\nBefore Sorting : \n");
```

```
                print_array(list, n);
                insertion_sort(list, n);
                printf("\nAfter Sorting : \n");
                print_array(list, n);
                break;
            case 6 :
                return;
            default :
                printf("Wrong Choice !\n");
        }
    }
}

void load_data() {
    printf("\nEnter the number of elements : ");
    scanf("%d", &n);

    printf("\nEnter %d Elements : ", n);
    for (int i = 0 ; i < n ; i++) {
        scanf("%d", &list[i]);
    }
}

void linear_search() {
    int to_search;
    printf("\nEnter the element to search for : ");
    scanf("%d", &to_search);

    int found = 0;
    for (int i = 0 ; i < n ; i++) {
        if (list[i] == to_search) {
            printf("\nFound %d at %d\n", to_search, i);
            found = 1;
            break;
        }
    }

    if (found == 0) {
        printf("\nElement not found ! \n");
    }
}

void binary_search() {
    int to_search;
    printf("\nEnter the element to search for : ");
    scanf("%d", &to_search);

    int found;
    if ((found = bsearch(list, to_search, 0, n-1)) > -1) {
        printf("\nFound %d at %d\n", to_search, found);
    } else {
        printf("\nElement not found !\n");
    }
}

int bsearch(int arr[], int ele, int start, int end) {
    if (end >= start) {
        int mid = (start + end) / 2;
```

```
        if (arr[mid] == ele)
            return mid;

        if (arr[mid] > ele)
            return bsearch(arr, ele, start, mid - 1);

        return bsearch(arr, ele, mid + 1, end);
    }

    return -1;
}

void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;

        merge_sort(arr, l, m);
        merge_sort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0 ; i < n1 ; i++)
        L[i] = arr[l + i];
    for (int j = 0 ; j < n2 ; j++)
        R[j] = arr[m + 1 + j];

    int i = 0;
    int j = 0;
    int k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
```

```
    }

}

void insertion_sort(int arr[], int n) {
    int key, j;
    for (int i = 0 ; i < n ; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j = j - 1;
        }

        arr[j+1] = key;
    }
}

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

5. Presentation of Results

```
Searching and Sorting

1.     Enter List Data

2.     Linear Search

3.     Binary Search

4.     Merge Sort

5.     Insertion Sort

6.     Exit

Enter your choice : 1

Enter the number of elements : 10

Enter 10 Elements : 10 9 8 7 6 5 4 3 2 1

10 9 8 7 6 5 4 3 2 1


Searching and Sorting

1.     Enter List Data

2.     Linear Search

3.     Binary Search

4.     Merge Sort
```

```
5.     Insertion Sort
6.     Exit
Enter your choice : 4
Before Sorting :
 10 9 8 7 6 5 4 3 2 1

After Sorting :
 1 2 3 4 5 6 7 8 9 10

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 2
Enter the element to search for : 5
Found 5 at 4

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 3
Enter the element to search for : 5
Found 5 at 4

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
```

```
5.     Insertion Sort
6.     Exit
Enter your choice : 5
Before Sorting :
 1 2 3 4 5 6 7 8 9 10

After Sorting :
 1 2 3 4 5 6 7 8 9 10

Searching and Sorting
1.     Enter List Data
2.     Linear Search
3.     Binary Search
4.     Merge Sort
5.     Insertion Sort
6.     Exit
Enter your choice : 6
Process finished with exit code 0
```

6. Analysis and Discussions

Linear Search:

Time Complexity : O(n)

Binary Search:

Time Complexity:

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is Theta(Logn).

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Merge Sort:

Time Complexity: O(n logn)

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

Insertion Sort:

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

7. Conclusions

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.