Data Structure Lab Report

Submitted By: Rezwan Ahmed Ratul

Student ID: 06224205101011

Department of Computer Science & Engineering

Khwaja Yunus Ali University

November 22, 2025

Contents

Experiment 1: Introduction to Data Structure

Experiment Name

Random Number Generation and Performance Analysis

Objective

- To understand basic input/output operations in C programming
- To implement random number generation using standard library functions
- To measure and analyze execution time of program operations
- To develop menu-driven programs for user interaction

Theory

Data structures are specialized formats for organizing, processing, and storing data. The foundation of working with data structures begins with understanding how to generate, store, and manipulate data efficiently.

Random Number Generation: The `rand()` function generates pseudo-random numbers. The `srand()` function seeds the random number generator using the current time to ensure different sequences in each execution.

Time Measurement: The `clock()` function from `time.h` allows us to measure program execution time by capturing timestamps before and after operations. The difference, divided by `CLOCKS_PER_SEC`, gives the elapsed time in seconds.

Question (a)

Write a C/C++ program to display a menu with exit function to choose any operation.

Code

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

void main()
{
    int i,n,Data[10000];
    srand(time(NULL));
    printf("Enter the amount of numbers: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        Data[i] = (rand() %100);
    }

    printf("\nThe %d Numbers are as follows:\n",n);

    for(i=0;i<n;i++)
    {
        printf("%5d",Data[i]);
    }
}
```

Question (b)

Write a C/C++ program to read N inputted numbers randomly, where N is inputted from keyboard and measure execution time.

Code

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

void main()
{
    int i,n,Data[10000];
    srand(time(NULL));
    printf("Enter the amount of numbers: ");
    scanf("%d",&n);
    clock_t begin = clock();
    for(i=0;i<n;i++)
    {
        Data[i] = (rand() %100);
    }
    printf("\nThe %d Numbers are as follows:\n",n);
    for(i=0;i<n;i++)
    {
        printf("%5d",Data[i]);
    }
    clock_t end = clock();
    double time_spent = (double)(end - begin) /
   CLOCKS_PER_SEC;
    printf("\nTotal time spend : %f Seconds",time_spent);
}
```

Conclusion

This experiment successfully demonstrated basic data handling operations in C programming. We learned to generate random numbers using `rand()` and `srand()` functions, store them in arrays, and display them. The time measurement functionality revealed that execution time increases with the number of elements processed. This is essential for understanding more complex data structures and their performance characteristics.

Experiment 2: String Processing

Experiment Name

Implementation of String Manipulation Operations

Objective

- To implement fundamental string processing algorithms from scratch
- To understand string representation and manipulation in memory
- To develop custom functions for string operations without using built-in library functions
- To create a comprehensive menu-driven string processing application

Theory

Strings are sequences of characters terminated by a null character (`\0`). String processing is fundamental in computer science and involves various operations:

String Length: Counting characters until the null terminator is reached.

Substring: Extracting a portion of a string from a starting position for a specified length.

Concatenation: Joining two strings end-to-end to form a new string.

Indexing: Finding the position of a pattern within a string using pattern matching algorithms.

Insertion: Adding a substring at a specific position within the original string.

Deletion: Removing occurrences of a pattern from the string.

Replacement: Substituting occurrences of a pattern with a new substring.

Dynamic memory allocation using `malloc()` is crucial for creating strings of variable length and returning modified strings from functions.

Question

Write a complete C/C++ menu-based program to perform the following string processing operations:

1. Input a string
2. Find the length of the string
3. Substring a string
4. Compare two strings
5. Indexing
6. Concatenation
7. Insert a string
8. Delete a/all string
9. Replace a/all string

Code

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

int strlength(char str[]) {
    int l;
    for (l = 0; str[l] != '\0'; l++);
    return l;
}

char* SubString(char str[], int st, int nofc) {
    int l = strlength(str), i;
    if (st < 0 || st >= l || nofc <= 0 || st + nofc > l) {
        char *empty = (char *)malloc(1);
        if (empty) empty[0] = '\0';
        return empty;
    }

    char *sub = (char *)malloc(nofc + 1);
    for (i = 0; i < nofc; i++) {
        sub[i] = str[st + i];
    }
    sub[nofc] = '\0';
    return sub;
}

char* Concatenation(char str1[], char str2[]) {
    int l1 = strlength(str1), l2 = strlength(str2), i;
    char *res = (char *)malloc(l1 + l2 + 1);
    for (i = 0; i < l1; i++) res[i] = str1[i];
    for (i = 0; i < l2; i++) res[l1 + i] = str2[i];
    res[l1 + l2] = '\0';
    return res;
}

int Indexing(char str[], char pattern[]) {
    int len = strlength(str), pat = strlength(pattern), i, j;
    for (i = 0; i <= len - pat; i++) {
        int found = 1;
        for (j = 0; j < pat; j++) {
            if (str[i + j] != pattern[j]) {
                found = 0;
                break;
            }
        }
        if (found) return i;
    }
    return -1;
}

char* Insertion(char str[], int pos, char insert[]) {
    int l1 = strlength(str), l2 = strlength(insert), i = 0, j
  ;
    if (pos < 0 || pos > l1) return NULL;

    char *res = (char *)malloc(l1 + l2 + 1);
    for (; i < pos; i++) res[i] = str[i];
    for (j = 0; j < l2; j++) res[i++] = insert[j];
    for (j = pos; j < l1; j++) res[i++] = str[j];
    res[i] = '\0';
    return res;
}

char* Delete(char str[], char pattern[]) {
    int pos = Indexing(str, pattern);
    if (pos == -1) return str;

    int len = strlength(str), pat = strlength(pattern);
    char *res = (char *)malloc(len - pat + 1);
    int i = 0, j = 0;
    while (i < len) {
        if (i == pos) {
            i += pat;
        } else {
            res[j++] = str[i++];
        }
    }
    res[j] = '\0';
    return res;
}

char* Replace(char str[], char old[], char new[]) {
    int pos = Indexing(str, old);
    if (pos == -1) return str;

    char *left = SubString(str, 0, pos);
    char *right = SubString(str, pos + strlength(old),
                            strlength(str) - pos - strlength(
   old));
    char *temp = Concatenation(left, new);
    char *final = Concatenation(temp, right);

    free(left);
    free(right);
    free(temp);

    return final;
}

```

```c
char* DeleteAll(char str[], char pattern[]) {
    char *temp = str, *res;
    while (Indexing(temp, pattern) != -1) {
        res = Delete(temp, pattern);
        if (temp != str) free(temp);
        temp = res;
    }
    return temp;
}

char* ReplaceAll(char str[], char old[], char new[]) {
    char *temp = str, *res;
    while (Indexing(temp, old) != -1) {
        res = Replace(temp, old, new);
        if (temp != str) free(temp);
        temp = res;
    }
    return temp;
}

int main() {
    int option, st, nofc, Loc;
    char str1[100], str2[100], str3[100];
    char *result;

    while (1) {
        printf("\n1. Input a string\n");
        printf("2. String length\n");
        printf("3. Display string\n");
        printf("4. Substring of the string\n");
        printf("5. Concatenate two strings\n");
        printf("6. Indexing of a Sub String\n");
        printf("7. Insert a string at a position\n");
        printf("8. Delete a substring\n");
        printf("9. Replace a substring\n");
        printf("10. Delete All Pattern\n");
        printf("11. Replace All Pattern\n");
        printf("0. Exit\nEnter your option: ");
        scanf("%d", &option);
        getchar();

        switch (option) {
            case 1:
                printf("Enter a string: ");
                gets(str1);
                break;

            case 2:
                printf("Length of the string: %d\n", strlength(str1));
                break;

            case 3:
                printf("Inputted string: ");
                puts(str1);
                break;

            case 4:
                printf("Enter the starting position: ");
                scanf("%d", &st);
                printf("Enter number of characters: ");
                scanf("%d", &nofc);
                getchar();
                result = SubString(str1, st, nofc);
                printf("Substring is: ");
                puts(result);
                free(result);
                break;

            case 5:
                printf("Enter second string: ");
                gets(str2);
                result = Concatenation(str1, str2);
                printf("Concatenation is: ");
                puts(result);
                free(result);
                break;

            case 6:
                printf("Enter the pattern to find: ");
                gets(str2);
                Loc = Indexing(str1, str2);
                if (Loc == -1)
                    printf("Pattern not found.\n");
                else
                    printf("Pattern found at position: %d\n",
    Loc);
                break;

            case 7:
                printf("Enter the string to insert: ");
                gets(str2);
                printf("Enter the position to insert at: ");
                scanf("%d", &Loc);
                getchar();
                result = Insertion(str1, Loc, str2);
                if (result != NULL) {
                    printf("After insertion: ");
                    puts(result);
                    free(result);
                } else {
                    printf("Invalid position.\n");
                }
                break;

            case 8:
                printf("Enter the pattern to delete: ");
                gets(str2);
                result = Delete(str1, str2);
                printf("After deletion: ");
                puts(result);
                break;

            case 9:
                printf("Enter the old substring: ");
                gets(str2);
                printf("Enter the new substring: ");
                gets(str3);
                result = Replace(str1, str2, str3);
                printf("After replacement: ");
```

```
                    puts(result);
                    break;

                case 10:
                    printf("Enter the pattern to delete all: ");
                    gets(str2);
                    result = DeleteAll(str1, str2);
                    printf("After deleting all: ");
                    puts(result);
                    break;

                case 11:
                    printf("Enter the old substring: ");
                    gets(str2);
                    printf("Enter the new substring: ");
                    gets(str3);
                    result = ReplaceAll(str1, str2, str3);
                    printf("After replacing all: ");
                    puts(result);
                    break;

                case 0:
                    exit(0);

                default:
                    printf("Invalid Input. Try again\n");
        }
    }
    return 0;
}
```

Conclusion

This experiment successfully demonstrated the implementation of comprehensive string manipulation operations. We developed custom functions for all major string operations without relying on built-in library functions, which deepened our understanding of string representation in memory and pointer manipulation. The menu-driven interface provides an interactive way to test each operation. Key learning outcomes include proper memory management using dynamic allocation, pointer arithmetic for string traversal, and algorithm design for pattern matching and string modification operations.

Experiment 3: Searching and Sorting

Experiment Name

Searching and Sorting Algorithms in C++

Objective

The objective of this experiment is to:

- Implement and understand the Bubble Sort algorithm to sort an array of integers.
- Implement and analyze Linear Search for finding elements in an unsorted array.
- Implement Iterative and Recursive Binary Search algorithms to efficiently search elements in a sorted array.
- Compare different searching techniques in terms of their implementation and efficiency.

Theory

Searching and sorting are fundamental operations in computer science used to organize and retrieve data efficiently.

3.3.1 Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The algorithm continues until no swaps are required, indicating that the list is sorted. **Time Complexity:** Worst-case: $O(n^2)$, Best-case: $O(n)$ (optimized version).

3.3.2 Linear Search

Linear Search is a straightforward method for finding a target element in a list. It checks each element sequentially until the target is found or the end of the list is reached. **Time Complexity:** $O(n)$.

3.3.3 Binary Search

Binary Search is an efficient search algorithm that works on sorted arrays. It divides the search interval in half repeatedly until the target is found or the interval is empty. It can be implemented iteratively or recursively. **Time Complexity:** $O(\log n)$.

Question

Write a C++ program to perform sorting and searching operations on an array of integers. The program should implement the following:

1. **Bubble Sort:** Sort an unsorted array in ascending order using the Bubble Sort algorithm.
2. **Linear Search:** Search for a target element using linear search.
3. **Iterative Binary Search:** Search for a target element in the sorted array using iterative binary search.
4. **Recursive Binary Search:** Search for a target element in the sorted array using recursive binary search.

Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Bubble Sort function
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

// Iterative binary search
int bs(vector<int> &a, int tar) {
    int st = 0, end = a.size() - 1;
    while (st <= end) {
        int mid = st + (end - st) / 2;
        if (a[mid] < tar) st = mid + 1;
        else if (a[mid] > tar) end = mid - 1;
        else return mid;
    }
    return -1;
}

// Recursive binary search
int rbs(vector<int> &a, int st, int end, int tar) {
    if (st > end) return -1;
    int mid = st + (end - st) / 2;
    if (a[mid] == tar) return mid;
    else if (a[mid] > tar) return rbs(a, st, mid - 1, tar);
    else return rbs(a, mid + 1, end, tar);
}

// Linear search
int linearSearch(const vector<int>& a, int tar) {
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == tar) return i;
    }
    return -1;
}

// Function to print the vector
void printVector(const vector<int>& arr) {
    for (int val : arr) cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> a = {64, 34, 25, 12, 22, 11, 90};
    cout << "Original array: ";
    printVector(a);

    int tar;
    cout << "Enter target to search: ";
    cin >> tar;

    cout << "Linear Search Index: " << linearSearch(a, tar)
   << endl;

    bubbleSort(a); // Sorting before binary search
    cout << "Sorted array: ";
    printVector(a);

    cout << "Iterative Binary Search Index: " << bs(a, tar)
   << endl;
    cout << "Recursive Binary Search Index: " << rbs(a, 0, a.
   size() - 1, tar) << endl;

    return 0;
}
```

Conclusion

In this experiment, we successfully implemented Bubble Sort, Linear Search, and both iterative and recursive versions of Binary Search.

- Bubble Sort efficiently sorted the array, allowing Binary Search to work correctly.
- Linear Search works on unsorted arrays but is less efficient than Binary Search on sorted arrays.
- Binary Search significantly reduced the number of comparisons, demonstrating its efficiency on sorted data.

The experiment helped understand the practical implementation and differences between searching and sorting algorithms in C++.

Experiment 4: Array Operations

Experiment Name

Implementation of Array-Based Data Structure Operations

Objective

- To implement fundamental array operations including insertion, deletion, and traversal
- To understand and implement Linear Search and Binary Search algorithms
- To implement the Bubble Sort algorithm for array sorting
- To analyze the time complexity and efficiency of different search algorithms
- To develop a comprehensive menu-driven array manipulation program

Theory

Arrays are the simplest and most fundamental data structure, consisting of a collection of elements stored in contiguous memory locations. Understanding array operations is crucial for working with more complex data structures.

Linear Search: A sequential search algorithm that checks each element until the target is found or the end is reached. Time complexity: O(n).

Binary Search: An efficient search algorithm for sorted arrays that repeatedly divides the search interval in half. Time complexity: O(log n). The array must be sorted before applying binary search.

Bubble Sort: A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Time complexity: O(n^2).

Insertion: Adding a new element to the array, typically at the end.

Deletion: Removing an element from the array and shifting subsequent elements to fill the gap.

Question

Write a complete C/C++ menu-based program to perform the following operations on an array of integers:

1. Create an array with random elements
2. Display all elements of the array
3. Search for an item using Linear Search
4. Search for an item using Binary Search (after sorting)
5. Insert a new item into the array
6. Delete an existing item from the array
7. Sort the array using Bubble Sort
8. Exit the program

Code

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

void traverse(int *a, int l);
int LinearSearch(int *a,int n,int item);
void BubbleSort(int *a,int l);
int BinarySearch(int *a,int st,int end,int item);
void insert(int *a, int *l, int item);
void delete(int *a, int *l, int item);

void main() {
    int i, n = 0, loc, item, a[10000];
    srand(time(NULL));
    char op;

    while(1) {
        printf("\n1. Create an Array\n");
        printf("2. Display the Array\n");
        printf("3. Search an item (Linear Search)\n");
        printf("4. Search an item (Binary Search)\n");
        printf("5. Insert an item\n");
        printf("6. Delete an item\n");
        printf("7. Sort using Bubble Sort\n");
        printf("0. Exit\nEnter your option: ");
        op = getche();
        printf("\n");

        switch(op) {
            case '1':
                printf("Enter the number of elements: ");
                scanf("%d", &n);
                for(i = 0; i < n; i++) {
                    a[i] = (rand() % 1000) + 1;
                }
                break;

            case '2':
                printf("The %d numbers are:\n", n);
                traverse(a, n);
                break;

            case '3':
                printf("Enter the item to search (Linear): ")
;
                scanf("%d", &item);
                loc = LinearSearch(a, n, item);
                if(loc == -1)
                    printf("Item not found!\n");
                else
                    printf("Item found at location %d\n", loc
);
                break;

            case '4':
                printf("Enter the item to search (Binary): ")
;
                    scanf("%d", &item);
                    loc = BinarySearch(a, 0, n-1, item);
                    if(loc == -1)
                        printf("Item not found!\n");
                    else
                        printf("Item found at location %d\n", loc
);
                    break;

                case '5':
                    printf("Enter the number to insert: ");
                    scanf("%d", &item);
                    insert(a, &n, item);
                    printf("Array after inserting %d:\n", item);
                    traverse(a, n);
                    break;

                case '6':
                    printf("Enter the number to delete: ");
                    scanf("%d", &item);
                    delete(a, &n, item);
                    printf("Array after deleting %d:\n", item);
                    traverse(a, n);
                    break;

                case '7':
                    BubbleSort(a, n);
                    printf("Sorted array:\n");
                    traverse(a, n);
                    break;

                case '0':
                    exit(0);
        }
    }
}

void traverse(int *a, int l) {
    for(int i = 0; i < l; i++) {
```

```c
        printf("%5d", a[i]);
    }
    printf("\n");
}

int LinearSearch(int *a,int n,int item) {
    for(int i = 0; i < n; i++) {
        if(a[i] == item)
            return i;
    }
    return -1;
}

void BubbleSort(int *a,int l) {
    for(int i = 0; i < l-1; i++) {
        for(int j = 0; j < l-i-1; j++) {
            if(a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int BinarySearch(int *a,int st,int end,int item) {
    while(st <= end) {
        int mid = st + (end - st)/2;
        if(item < a[mid]) end = mid - 1;
        else if(item > a[mid]) st = mid + 1;
        else return mid;
    }
    return -1;
}

void insert(int *a, int *l, int item) {
    a[*l] = item;
    (*l)++;
}

```

```c
void delete(int *a, int *l, int item) {
    int i, found = 0;
    for(i = 0; i < *l; i++) {
        if(a[i] == item) {
            found = 1;
            break;
        }
    }
    if(found) {
        for(int j = i; j < *l-1; j++) {
            a[j] = a[j+1];
        }
        (*l)--;
    } else {
        printf("Item not found!\n");
    }
}
```

Conclusion

This experiment provided hands-on experience with fundamental array operations and search algorithms. We successfully implemented both Linear and Binary Search, observing that Binary Search is significantly faster for sorted arrays but requires the preprocessing step of sorting. The Bubble Sort implementation, while not the most efficient sorting algorithm, clearly demonstrates the sorting principle. We learned that search algorithm selection depends on whether the data is sorted and the frequency of search operations. The practical implementation reinforced the importance of understanding time complexity when choosing appropriate algorithms for different scenarios.

Experiment 5: Linked List

Experiment Name

Implementation and Manipulation of Singly Linked List

Objective

- To understand the concept of dynamic memory allocation and linked structures
- To implement a singly linked list with various operations
- To compare linked list performance with array-based structures
- To implement search algorithms for both sorted and unsorted linked lists
- To maintain sorted order during insertion operations

Theory

A linked list is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation.

Node Structure: Each node contains two parts: data (info) and a pointer to the next node (next).

Advantages over Arrays:

- Dynamic size - can grow or shrink during runtime
- Efficient insertion and deletion - no need to shift elements
- No memory wastage from unused allocated space

Disadvantages:

- No random access - must traverse from the beginning
- Extra memory for storing pointers
- Not cache-friendly due to non-contiguous memory

Operations:

- **Traversal:** Visiting each node sequentially
- **Search:** Finding a node with specific data
- **Insertion:** Adding a new node while maintaining order
- **Deletion:** Removing a node and adjusting pointers
- **Sorting:** Rearranging nodes in ascending/descending order

Question

Write a complete C/C++ menu-based program to perform the following operations on a singly linked list of integers:

1. Create a linked list using random numbers
2. Create a linked list by inputting numbers from the user
3. Display all elements of the linked list
4. Sort the linked list in ascending order
5. Search for an item in an unsorted linked list
6. Search for an item in a sorted linked list
7. Insert a new item into the linked list while maintaining sorted order
8. Delete an existing item from the linked list
9. Exit the program

Code

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

struct LinkList {
    int info;
    struct LinkList *next;
};

typedef struct LinkList LKL;

LKL *CreateList(LKL *, int);
void Traverse(LKL *);
void ListSort(LKL *);
LKL *SearchUSL(LKL *, int);
LKL *SearchSL(LKL *, int);
LKL *InputList(LKL *, int);
LKL *Insert(LKL *, int);
LKL *Delete(LKL *, int);
void FreeList(LKL *st);

int main() {
    int n;
    LKL *St = NULL, *Loc;
    srand(time(NULL));
    char option;

    while (1) {
        printf("\n1. Create a Link List Using Random Numbers\n");
        printf("2. Create a Link List Using Inputted Numbers\n");
        printf("3. Display the Linked List\n");
        printf("4. Sort the Linked List\n");
        printf("5. Search an Item from Linked List (Unsorted)\n");
        printf("6. Search an Item from Linked List (Sorted)\n
");
        printf("7. Insert an Item into the Sorted Linked List
\n");
        printf("8. Delete an Item from the Linked List\n");
        printf("0. Exit\nEnter your option : ");
        option = getche();

        switch (option) {
        case '1':
            St = NULL;
            printf("\nEnter amount of Node : ");
            scanf("%d", &n);
            St = CreateList(St, n);
            break;

        case '2':
            St = NULL;
            printf("\nEnter amount of Node : ");
            scanf("%d", &n);
            St = InputList(St, n);
            break;

        case '3':
            printf("\nThe Linked List is:\n");
            Traverse(St);
            break;

        case '4':
            printf("\nThe Sorted Linked List is:\n");
            ListSort(St);
            Traverse(St);
            break;

        case '5':
            printf("\nEnter the Item for search : ");
            scanf("%d", &n);
            Loc = SearchUSL(St, n);
            if (Loc == NULL)
                printf("Item is not in the list.\n");
            else
```

```c
                printf("Item found at location : %p.  Item is %d\n",
                       (void *)Loc, Loc->info);
            break;

        case '6':
            printf("\nEnter the Item for search : ");
            scanf("%d", &n);
            Loc = SearchSL(St, n);
            if (Loc == NULL)
                printf("Item is not in the list.\n");
            else
                printf("Item found at location : %p.  Item is %d\n",
                       (void *)Loc, Loc->info);
            break;

        case '7':
            printf("\nEnter the value you want to insert : ");
            scanf("%d", &n);
            St = Insert(St, n);
            printf("\n%d has been inserted into the list\n", n);
            Traverse(St);
            break;

        case '8':
            printf("\nEnter the item you want to delete : ");
            scanf("%d", &n);
            St = Delete(St, n);
            Traverse(St);
            break;

        case '0':
            FreeList(St);
            exit(0);
            break;
        }
    }
    return 0;
}

LKL *CreateList(LKL *st, int n) {
    LKL *i = NULL, *tmp;
    while (n--) {
        tmp = (LKL *)malloc(sizeof(LKL));
        tmp->info = (rand() % 1000) + 1;
        tmp->next = NULL;

        if (st == NULL) {
            st = tmp;
            i = st;
        } else {
            i->next = tmp;
            i = tmp;
        }
    }
    return st;
}

LKL *InputList(LKL *st, int n) {
    LKL *i = NULL, *tmp;
    printf("Enter %d numbers separated by space:\n", n);
    while (n--) {
        tmp = (LKL *)malloc(sizeof(LKL));
        scanf("%d", &tmp->info);
        tmp->next = NULL;

        if (st == NULL) {
            st = tmp;
            i = st;
        } else {
            i->next = tmp;
            i = tmp;
        }
    }
    return st;
}

```

```c
void Traverse(LKL *st) {
    LKL *i;
    for (i = st; i != NULL; i = i->next)
        printf("%5d", i->info);
    printf("\n");
}

void ListSort(LKL *st) {
    LKL *i, *j;
    for (i = st; i != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->info > j->info) {
                i->info = i->info ^ j->info;
                j->info = i->info ^ j->info;
                i->info = i->info ^ j->info;
            }
        }
    }
}

LKL *SearchUSL(LKL *St, int n) {
    LKL *i;
    for (i = St; i != NULL; i = i->next) {
        if (i->info == n)
            return i;
    }
    return NULL;
}

LKL *SearchSL(LKL *St, int n) {
    LKL *i;
    for (i = St; i != NULL && i->info <= n; i = i->next) {
        if (i->info == n)
            return i;
    }
    return NULL;
}

LKL *Insert(LKL *st, int n) {
    LKL *tmp = (LKL *)malloc(sizeof(LKL));
    tmp->info = n;
    tmp->next = NULL;

    if (st == NULL || n < st->info) {
        tmp->next = st;
        return tmp;
    }

    LKL *i = st;
    while (i->next != NULL && i->next->info < n) {
        i = i->next;
    }

    tmp->next = i->next;
    i->next = tmp;

    return st;
}

LKL *Delete(LKL *st, int n) {
    if (st == NULL) {
        printf("List is empty.\n");
        return st;
    }

    LKL *tmp, *prev;

    if (st->info == n) {
        tmp = st;
        st = st->next;
        free(tmp);
        return st;
    }

    prev = st;
    tmp = st->next;
    while (tmp != NULL && tmp->info != n) {
        prev = tmp;
        tmp = tmp->next;
    }
```

```c

    if (tmp == NULL) {
        printf("Item not found in list.\n");
        return st;
    }

    prev->next = tmp->next;
    free(tmp);

    return st;
}

void FreeList(LKL *st) {
    LKL *tmp;
    while (st != NULL) {
        tmp = st;
        st = st->next;
        free(tmp);
    }
}
```

Conclusion

This experiment successfully demonstrated the implementation and manipulation of singly linked lists. We observed that linked lists provide flexibility in dynamic memory allocation and efficient insertion/deletion operations compared to arrays. The sorted insertion function showcased how to maintain order during insertion, which is more efficient than sorting after each insertion. Search operations in sorted linked lists demonstrated early termination optimization. Key insights include understanding pointer manipulation, proper memory management to avoid leaks, and the trade-offs between linked lists and arrays. The experience with dynamic data structures is fundamental for implementing more complex structures like trees and graphs.

Experiment 6: Stack

Experiment Name

Implementation of Stack Data Structure Using Linked List

Objective

- To understand the Last-In-First-Out (LIFO) principle of stack data structure
- To implement stack operations using a linked list (dynamic implementation)
- To perform Push, Pop, and Display operations on a stack
- To understand the advantages of linked list implementation over array implementation
- To apply stack concepts for solving real-world problems

Theory

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added to the stack is the first one to be removed.

Key Operations:

- **Push:** Insert an element at the top of the stack
- **Pop:** Remove and return the top element from the stack
- **Display/Traverse:** View all elements in the stack
- **Peek/Top:** View the top element without removing it

Stack Implementation Methods:

- **Array-based:** Fixed size, simple implementation, but limited capacity

- **Linked List-based:** Dynamic size, no overflow (until memory is full), uses dynamic memory allocation

Applications of Stack:

- Function call management (recursion)
- Expression evaluation and conversion (infix to postfix)
- Undo mechanism in text editors
- Backtracking algorithms
- Browser history (back button)
- Syntax parsing in compilers

Time Complexity:

- Push operation: O(1)
- Pop operation: O(1)
- Display operation: O(n)

Question

Write a complete C/C++ menu-based program to implement a stack using a singly linked list. The program should provide the following operations:

1. **Push:** Insert an item onto the top of the stack
2. **Pop:** Remove and return the top item from the stack
3. **Display:** Show all elements currently in the stack
4. **Exit:** Terminate the program

Code

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

struct S {
    int info;
    struct S *next;
};

typedef struct S STACK;

STACK *Push(STACK *, int);
int Pop(STACK **);
void Display(STACK *);

void main()
{
    int x;
    STACK *A = NULL;
    srand(time(NULL));
    char op;

    while(1)
    {
        printf("\n\n1. Push Operation using Linked List Stack
  ");
        printf("\n2. Pop Operation using Linked List Stack");
        printf("\n3. Display Stack");
        printf("\n0. Exit\nEnter your option : ");

        op = getche();
        switch(op)
        {
        case '1':
            printf("\nEnter the item for Push : ");
            scanf("%d", &x);
            A = Push(A, x);
```

```
                break;
            case '2':
                x = Pop(&A);
                if(x == -1)
                    printf("\nStack is Empty\n");
                else
                    printf("\nPopped item is %d\n", x);
                break;
            case '3':
                Display(A);
                break;
            case '0':
                exit(0);
                break;
            default:
                printf("\nInvalid Option\n");
            }
    }
}

STACK *Push(STACK *Top, int item)
{
    STACK *New = (STACK *)malloc(sizeof(STACK));
    New->info = item;
    New->next = Top;
    Top = New;
    return Top;
}

int Pop(STACK **Top)
{
    int item;
    STACK *temp;
    if((*Top) != NULL)
    {
        temp = *Top;
        item = temp->info;
        *Top = (*Top)->next;
        free(temp);
    }
    else
        item = -1;
    return item;
}

void Display(STACK *Top)
{
    if(Top == NULL)
    {
        printf("\nStack is Empty\n");
        return;
    }
    printf("\nStack elements (Top to Bottom): ");
    while(Top != NULL)
    {
        printf("%d ", Top->info);
        Top = Top->next;
    }
    printf("\n");
}
```

Conclusion

This experiment successfully demonstrated the implementation of a stack using a linked list structure. The linked list-based approach provides dynamic sizing capability, eliminating the overflow condition that exists in array-based implementations. We implemented the fundamental stack operations (Push and Pop) with O(1) time complexity, making them highly efficient. The LIFO principle was clearly observed during the operations. Understanding stack implementation is crucial as it forms the foundation for more complex data structures and algorithms. The experience gained from this experiment is applicable to solving various computational problems including recursion, backtracking, and expression evaluation. The linked list implementation also reinforced our understanding of dynamic memory management and pointer manipulation in C programming.