



SUNWAY
INT'L BUSINESS SCHOOL

Programma Name: BCS HONS

Course Code: CSC 2516

Course Name: Data structure and Algorithm

Internal Exam

Date of Submission: 2021/08/24

Submitted By:

Submitted To:

Student Name: Subash Sunar

Faculty Name: Prakash Chandra

IUKL ID: 041902900078

Department: LMS

Semester: Fourth Semester

Intake: September 2019

QUESTION NO 1

The process of collecting and organizing data in the best way is known as Data Structures. Here the operations on data are performed in an efficient way. Its functionality supports a specific purpose of accessing and performing operations in a given appropriate ways.

Data structures have gained its importance for the following reasons:

- a. In software design, Data structures are known to be major factors for collection, storing and organizing of data rather than algorithms in some programming languages.

- b. In almost every software system and program, Data structures are often included now-a-days.
- c. Often Data structures are used in the combination of algorithms. This allows the management of large amounts of data in an efficient way.
- d. No matter what problem are you solving, in one way or another you have to deal with data — whether it's an employee's salary, stock prices, a grocery list, or even a simple telephone directory.
- e. Based on different scenarios, data needs to be stored in a specific format. We have a handful of data structures that cover our need to store data in different formats.

Commonly used Data Structures

Let's first list the most commonly used data structures, and then we'll cover them one by one:

- 1. Arrays
- 2. Stacks
- 3. Queues
- 4. Linked Lists
- 5. Trees
- 6. Graphs
- 7. Tries (they are effectively trees, but it's still good to call them out separately).
- 8. Hash Tables

QUESTION NO 2

A Linear Search is the most basic type of searching algorithm. A Linear Search sequentially moves through your collection (or data structure) looking for a matching value. In other words, it looks down a list, one item at a time, without jumping.

Think of it as a way of finding your way in a phonebook. A Linear Search is starting at the beginning, reading every name until you find what you're looking for. In complexity terms this is an $O(n)$ search - the time taken to search the list, gets bigger at the same rate as the list does

Linear Search: Steps on how it works:

Here is simple approach is to do Linear Search:

- a. Start from the leftmost element of array and one by one compare the element we are searching for with each element of the array.

- b. if there is a match between the element we are searching for and an element of the array, return the index.
- c. If there is no match between the element we are searching for and an element of the array, return -1.

//linearSearch Function

```
int linearSearch(int data [], int length, int Val) {  
for (int i = 0; i <= length; i++) {if (Val == data[i]) {  
    return i; } //end if  
} //end for  
return -1; //Value was not in the list } //end linearSearch Function
```

Analysis of Linear Search

As we have seen throughout this tutorial that Linear Search is certainly not the absolute best method for searching but do not let this taint your view on the algorithm itself. People are always attempting to better versions of current algorithms in an effort to make existing ones more efficient. Not to mention that Linear Search as shown has its place and at the very least is a great beginner 's introduction into the world of searching algorithms. With this in mind we progress to the asymptotic analysis of the Linear Search:

Worst Case:

The worst case for Linear Search is achieved if the element to be found is not in the list at all. This would entail the algorithm to traverse the entire list and return nothing. Thus, the worst-case running time is:

$O(N)$.

Average Case:

The average case is in short revealed by insinuating that the average element would be somewhere in the middle of the list or $N/2$. This does not change since we are dividing by a constant factor here, so again the average case would be:

$O(N)$.

Best Case:

The best case can be reached if the element to be found is the first one in the list. This would not have to do any traversing spare the first one giving this a constant time complexity or:

$O(1)$.

QUESTION NO 3

//DOUBLY LINKED LIST APPLICATION

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

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

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
    struct Node *prev;
```

```
};
```

```
struct Node *head;
```

```
struct Node *getTail()
```

```
{
```

```
    struct Node *temp = head;
```

```
    //getting to the end node
```

```
    while (temp->next != NULL)
```

```
    {
```

```

        temp = temp->next;
    }
    return temp;
}

void insertAtBegining(int val)
{
    struct Node *ptr;
    ptr = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = head;
    if (ptr == NULL)
        printf("\noverflow\n");

    else
    {
        ptr->data = val;
        ptr->next = head;
        ptr->prev = NULL;
        if (head != NULL)
            temp->prev = ptr;
        head = ptr;
    }
}

void insertAtPos(int val, int pos)
{
    struct Node *ptr;
    ptr = (struct Node *)malloc(sizeof(struct Node));

```

```

if (ptr == NULL)
{
    printf("\noverflow\n");
    return;
}
if (pos == 1)
{
    insertAtBeginning(val);
    return;
}
struct Node *temp = head;

//getting to the position
for (int i = 1; i < pos - 1; i++)
{
    temp = temp->next;
}

//adding the value at the position
ptr->data = val;
ptr->next = temp->next;
ptr->prev = temp;
temp->next = ptr;
}

void insertAtEnd(int val)

```

```

{
    struct Node *ptr;
    ptr = (struct Node *) malloc(sizeof(struct Node));

    if (ptr == NULL)
    {
        printf("\noverflow\n");
        return;
    }
    struct Node *temp = head;
    //getting to the end node
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    ptr->data = val;
    ptr->next = NULL;
    ptr->prev = temp;
    temp->next = ptr;
}

```

```

void deleteAtBeginning()
{
    struct Node *temp = head;
    if (head == NULL)
    {
        printf("\nUNDERFLOW\n");
    }
}

```

```

        return;
    }

    temp = head->next;
    temp->prev = NULL;
    free(head);
    head = temp;
}

void deleteAtPos(int pos)
{
    if (head == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    if (pos == 1)
    {
        deleteAtBegining();
        return;
    }

    struct Node *temp = head;
    struct Node *ptr;

    //getting to the position
    for (int i = 1; i < pos - 1; i++)

```



```

{
    temp = temp->next;
}

if (temp == NULL)
{
    printf("\nthere are no values to delete\n");
    return;
}

ptr = temp->next;
temp->next = ptr->next;
ptr->next->prev = temp;
free(ptr);
}

void deleteFromEnd()
{
    struct Node *temp = getTail()->prev;
    struct Node *ptr;

    ptr = temp->next;
    temp->next = ptr->next;
    free(ptr);
}

```

```

void printFromEnd()
{

    struct Node *tail = getTail();
    printf("\n List: [");
    struct Node *temp = tail;
    while (1)
    {

        printf(" %d ,", temp->data);
        temp = temp->prev;

        if (temp->prev == NULL)
        {
            printf(" %d ", temp->data);
            break;
        }
    }
    printf("\n");
}

void printList()
{
    printf("\n List: [");
    struct Node *temp = head;
    while (1)
    {

```

```

    printf(" %d ,", temp->data);
    temp = temp->next;

    if (temp->next == NULL)
    {
        printf(" %d ", temp->data);
        break;
    }
}
printf("\n");
}

void search(int key)
{
    struct Node *temp = head;

    int i = 1;

    int found = 1;

    if (temp == NULL)
    {
        printf("\nlist is empty\n");
        return;
    }

    while (temp->data != key)
    {
        temp = temp->next;

        i++;

        if (temp == NULL)
        {

```

```
        printf("\n%d not found\n", key);
        found = 0;
        break;
    }
}

if (found)
    printf("\nThe %d is found at %d\n", key, i);
}

int main()
{

    head = NULL;

    insertAtBeginning(3);
    insertAtBeginning(2);
    insertAtBeginning(1);

    insertAtPos(4, 4);
    insertAtEnd(5);
    printList();
    printFromEnd();

    printf("\nDeleting\n");
    deleteAtBeginning();
    deleteFromEnd();
}
```

```

deleteAtPos(2);

printList();

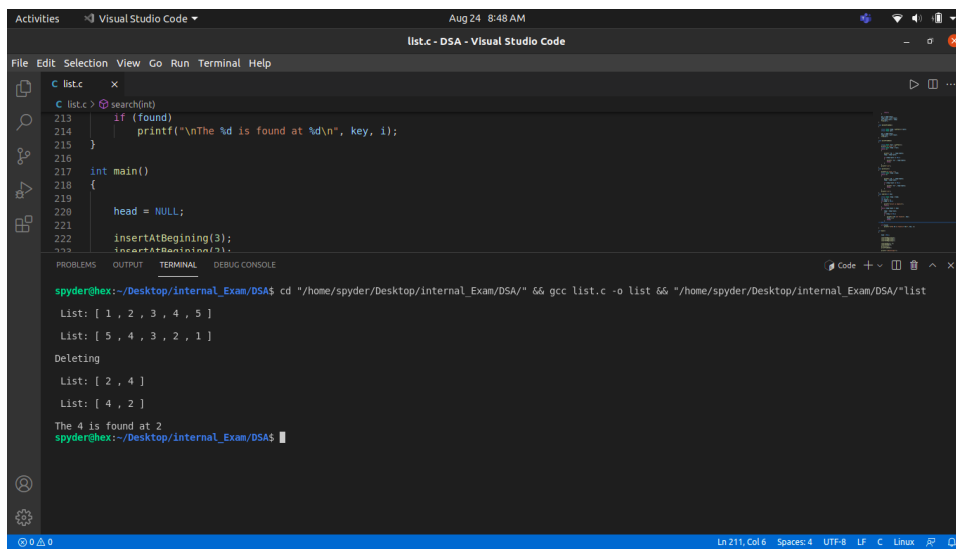
printFromEnd();

search(4);

return 0;

}

```



The screenshot shows the Visual Studio Code interface with a C file named `list.c` open. The code defines a linked list structure and functions for inserting, deleting, and printing the list. The terminal output shows the program's execution, including the initial list state, insertion of a new node, deletion of a node, and the final list state after searching for the value 4.

```

C list.c
213     if (found)
214     {
215         printf("\nThe %d is found at %d\n", key, i);
216     }
217 int main()
218 {
219     head = NULL;
220     insertAtBeginning(3);
221     insertAtBeginning(5);
222     // ...
223 }

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
spyder@hex:~/Desktop/internal_Exam/DSA$ cd ~/home/spyder/Desktop/internal_Exam/DSA/ && gcc list.c -o list && ./list
List: [ 1 , 2 , 3 , 4 , 5 ]
List: [ 5 , 4 , 3 , 2 , 1 ]
Deleting
List: [ 2 , 4 ]
List: [ 4 , 2 ]
The 4 is found at 2
spyder@hex:~/Desktop/internal_Exam/DSA$

```

QUESTION NO 4

a)

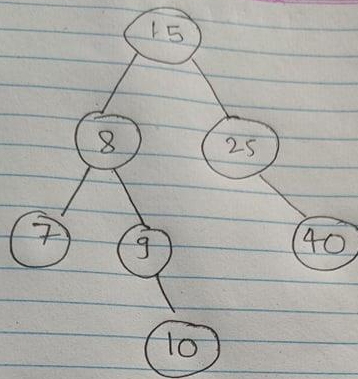


Fig: Binary search tree

b)

Inorder traversal

[7, 8, 9, 10, 15, 25, 40]

Pre order traversal

[15, 8, 7, 9, 10, 25, 40]

Post order traversal

[7, 10, 9, 8, 40, 25, 15]

.....THANKYOU.....