**THE INSTITUTE OF FINANCE MANAGEMENT**



**FACULTY OF COMPUTING MATHEMATICS - FCM**

**DEPARTMENT OF COMPUTER SCIENCE**

**BACHELOR DEGREE IN COMPUTER SCIENCE**

**CSU 07317:DATA STRUCTURE AND ALGORITHMS - ASSIGNMENT**

|  |  |  |
| --- | --- | --- |
| **S/N** | **NAME** | **REGISTRATION NUMBER** |
| **1.** | **CHRISTINA S KISAMBU** | **IMC/BCS/2216198** |
| **2.** | **RODGERS J SWAI** | **IMC/BCS/2226482** |
| **3.** | **HABIBA A DAUDI** | **IMC/BCS/2211330** |
| **4.** | **MWESIGA G MTEMBEI** | **IMC/BCS/2227037** |
| **5.** | **SARAH J MKISI** | **IMC/BCS/2214387** |
| **6.** | **SAUDA A KASSIM** | **IMC/BCS/2210043** |
| **7.** | **SHAMSA A BELEKO** | **IMC/BCS/2216303** |
| **8.** | **KELVIN S MGUHI** | **IMC/BCS/2222452** |
| **9.** | **ASHIRI J MWANGIA** | **IMC/BCS/2226943** |
| **10.** | **NELSON L HENDRY** | **IMC/BCS/2222246** |

**YEAR TWO 2023/2024**

Questions

LINKED LIST DATA STRUCTURE:

1. Explain what about linked list
2. Implementation of linked list
3. With an example explain how to search and sort data in linked list

TREE DATA STRUCTURE:

1. Explain what about Tree data structure
2. Types of Tree Data Structure
3. How to implement binary tree

Qn1) Explain what about linked list

**Linked list:** is the linear data structure in which data element are stored dynamically. Data elements (nodes) are connected in sequence to form linear list. Each data element contains a link to another element along with the data present it. Node has two fields; first field contain data and another field is pointer which has address that keeps a reference to the next node.

The following are some of aspects of linked data representation;

* **Nodes.** The basic building blocks of linked data structure where by each node contains data and a reference to the next node in the sequence.
* **Pointers or References.** The connections between the nodes are established using pointers. The pointers indicate the memory location of the next node in the sequence.
* **Dynamic memory Allocation** Each element can be removed or added without the need to reallocate a fixed amount of memory.
* **Memory Requirement** Linked list do not require elements to be stored in contiguous memory locations, nodes can be scattered throughout the memory.
* **Ease of Insertion and deletion** Insertion or deletion in linked list is more straightforward than in contiguous structures. To insert or delete an element, you only need to adjust the references in the neighboring nodes.

The following are some examples of linked data representation.

* **Singly Linked list:** In a singly linked list, each node contains data and a reference (link) to the next node in the sequence. The last node typically points to null.

Node 1 -> Node 2 -> Node 3 -> ... -> Node N -> null

* **Doubly Linked list:** In a doubly linked list, each node has references to both the next and the previous nodes, allowing for traversal in both directions.

null <- Node 1 <-> Node 2 <-> Node 3 <-> ... <-> Node N -> null

* **Circular Linked List:** IN circular linked list the last node is connected to the first, forming a loop. Node 1 -> Node 2 -> Node 3 -> ... -> Node N -> Node 1
* **Linked List as a Stack:** A linked list can be used to implement a stack, where elements are added or removed from one end (the top).

Qn2) Example of linked list (C programming language)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

int main() {

struct Node\* head = NULL;

struct Node\* second = NULL;

struct Node\* third = NULL;

struct Node\* fourth = NULL;

head = (struct Node\*)malloc(sizeof(struct Node));

second = (struct Node\*)malloc(sizeof(struct Node));

third = (struct Node\*)malloc(sizeof(struct Node));

fourth = (struct Node\*)malloc(sizeof(struct Node));

fourth->data = 7;

fourth->next =NULL ;

head->data = 3;

head->next = second;

second->data = 5;

second->next = third;

third->data = 1;

third->next = fourth;

struct Node\* temp = head;

printf("Linked list elements: ");

while (temp != NULL) {

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

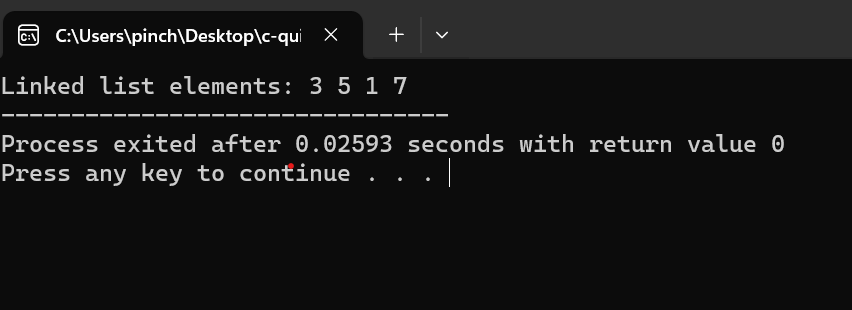
temp = temp->next;

}

return 0;

}

Output:



Qn3) With an example explain how to search and sort data in linked list

search and sort data in a linked list, consider the following steps:

**Search**:

* Traverse the linked list by starting at the head node and moving through each node until the target element is found.
* If the element is found, you can stop the traversal and return the node containing the element.
* If the entire list is traversed and the element is not found, it means the element does not exist in the linked list.

**Sort**:

* To sort a linked list, you can employ various sorting techniques such as bubble sort, insertion sort, merge sort, or quick sort.
* For instance, to sort a singly linked list, you can use algorithms like bubble sort or merge sort to rearrange the elements in ascending or descending order 1.
* Another approach involves iteratively comparing and rearranging nodes to achieve the desired ascending or descending order.

**Example of c program to implement searching and sorting in linked list**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to append a new node to the linked list

void append(struct Node\*\* head, int data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = data;

new\_node->next = NULL;

if (\*head == NULL) {

\*head = new\_node;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = new\_node;

}

}

// Function to display the linked list

void display(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

printf("%d -> ", current->data);

current = current->next;

}

printf("NULL\n");

}

// Function to search for an element in the linked list

int search(struct Node\* head, int target) {

struct Node\* current = head;

while (current != NULL) {

if (current->data == target) {

return 1; // Element found

}

current = current->next;

}

return 0; // Element not found

}

// Function to perform bubble sort on the linked list

void bubble\_sort(struct Node\* head) {

int swapped;

struct Node \*ptr1, \*lptr = NULL;

// Check for empty list

if (head == NULL)

return;

do {

swapped = 0;

ptr1 = head;

while (ptr1->next != lptr) {

if (ptr1->data > ptr1->next->data) {

// Swap data

int temp = ptr1->data;

ptr1->data = ptr1->next->data;

ptr1->next->data = temp;

swapped = 1;

}

ptr1 = ptr1->next;

}

lptr = ptr1;

} while (swapped);

}

// Function to free the memory allocated for the linked list

void free\_list(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

struct Node\* next = current->next;

free(current);

current = next;

}

}

int main() {

struct Node\* myLinkedList = NULL;

// Append elements to the linked list

append(&myLinkedList, 3);

append(&myLinkedList, 1);

append(&myLinkedList, 4);

append(&myLinkedList, 1);

append(&myLinkedList, 5);

// Display the linked list

printf("Linked List: ");

display(myLinkedList);

// Search for an element

int targetElement = 4;

if (search(myLinkedList, targetElement)) {

printf("%d found in the linked list.\n", targetElement);

} else {

printf("%d not found in the linked list.\n", targetElement);

}

// Sort the linked list

bubble\_sort(myLinkedList);

// Display the sorted linked list

printf("Sorted Linked List: ");

display(myLinkedList);

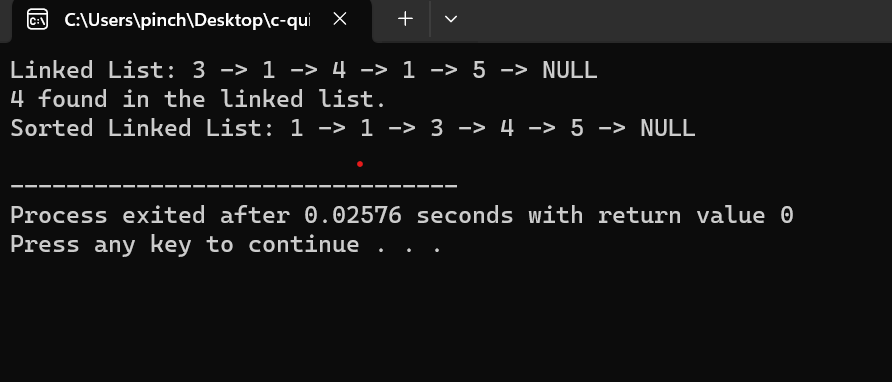
// Free the memory allocated for the linked list

free\_list(myLinkedList);

return 0;

}

Output:



**TREE DATA STRUCTURE:**

1.Explain what about tree data structure.

Tree data structure is a hierarchical arrangement of nodes where each node has a parent child relationship it starts with a root node at the top, which can have child nodes branching out.it has root node, which is the topmost node and each node can have zero or more child nodes. Nodes with no children are called leaves so most are used for organize data they play a crucial role in algorithms and data storage, providing efficient ways to manage and retrieve information

2.Type of tree data structure

There are different type of tree and each has its advantage and tailored to specific requirement in term of search speed, insertion, deletion and memory usage

* **Binary Trees** each node has at most two children, usually referred to as the left child and the right child or a collection of elements called a node each node has 0 and 1 use binary tree it ensure efficient searching, insertion and deletion operation
* **Binary search tree** also knowns as an ordered binary tree is a variant of binary in which the node is arranged in an order all the nodes in the left sub tree have a value less than that of the root node. The same rule is applicable to every sub tree in the tree
* **AVL Trees** a type of self-balancing binary search tree where the balance factor of each node is maintained to ensure logarithmic height providing efficient search, insertion and deletion operation.
* **Red Black Trees** another self-balancing binary search tree where each node has an extra attribute representing the color (red or black) to ensuring balance height and efficient operation.
* **Trie (prefix tree)** A tree structure used for storing a dynamic set or associative array where the keys are usually sequences like string trie structure are particularly efficient for search operation on string data
* **B trees** this is another type of tree where balanced tree structure designed for efficient storage and retrieval in databases and file systems. They have a variable number of child nodes per node, allowing them to hander large dataset with good performance

3. How to implement binary tree

Binary tree implementation in C programming language.

**Node structure.**

The basic structure for a node is defined where by each node contains an integer(data), a pointer to the left child (left) and a pointer to the right child(right)

**CreateNode Function**

The createNode function dynamically allocates memory for a new node, initializes its data with a given value, and sets its left and right pointers to NULL, it then returns a pointer to newly created node.

**Main function-Creating Binary Tree.**

In main function we have created a simple binary tree

The root node has data ‘1’.

The left child of the root has data ‘2’.

The right child of the root has data ‘3’.

The left child of the root’s left child has data ‘4’.

The right child of the root’s left child has data ‘5’.

Accessing and printing Data

The part of the ‘main’ function prints some basic information about the nodes

The data of the root node, the data of the left child of the root, and the data of the right child of the root.

#include <stdio.h>

#include <stdlib.h>

// Define a basic structure for a node in the binary tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node with the given data

struct Node\* createNode (int data) {

struct Node\* newNode = (struct Node\*) malloc (sizeof (struct Node));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

int main () {

// Creating a simple binary tree

struct Node\* root = createNode (1);

root->left = createNode (2);

root->right = createNode (3);

root->left->left = createNode (4);

root->left->right = createNode (5);

// Accessing and printing some data

printf ("Root data: %d\n", root->data);

printf ("Left child of root: %d\n", root->left->data);

printf ("Right child of root: %d\n", root->right->data);

return 0;

}

Output:

