SYLLABUS

1.Linked Lists:

2.Singly linked lists:

3.representation and operations,

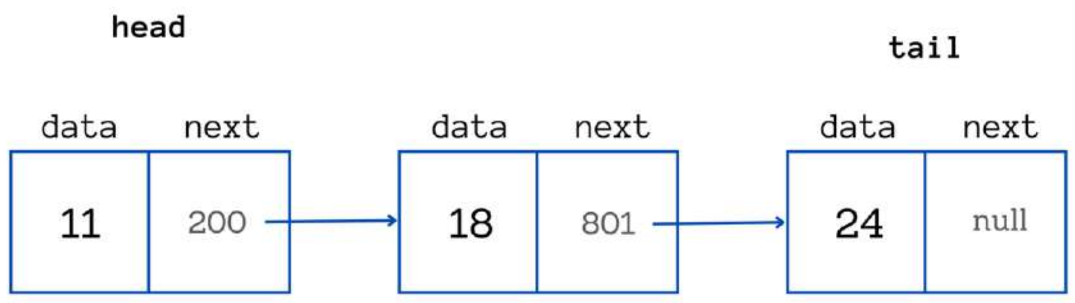
4.doubly linked lists and circular linked lists,

Comparing arrays and linked lists,

5. Applications of linked lists.

1.) LINKED LISTS

Certainly! Let's explore **Linked Lists**, a fundamental data structure in computer science. Linked lists consist of nodes, where each node contains data and a reference (link) to the next node in the sequence. Unlike arrays, linked lists allow for efficient insertion or removal of elements from any position in the list, as the nodes are not stored contiguously in memory.



Here are the key points about linked lists:

What is a Linked List?

- A linked list is a linear data structure that consists of a series of nodes connected by pointers.
- Each node contains data and a reference to the next node in the list.
- Linked lists allow for dynamic memory allocation and efficient insertion and deletion operations compared to arrays.

Types of Linked Lists

1. **Singly Linked List**:
   - Each node has a reference to the next node.
   - Example: A simple singly linked list:

```
struct Node {
    int data;
    struct Node* next;
};
```

2. **Doubly Linked List**:
   - Each node has references to both the next and previous nodes.
   - Allows traversal in both directions.
   - Example: A doubly linked list:

```
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
```

3. **Circular Linked List**:
   - The last node points back to the first node, forming a circular structure.
   - Example: A circular singly linked list or circular doubly linked list.
4. **Header Linked List**:
   - Contains an additional header node that does not store data.
   - Used for easier list management.

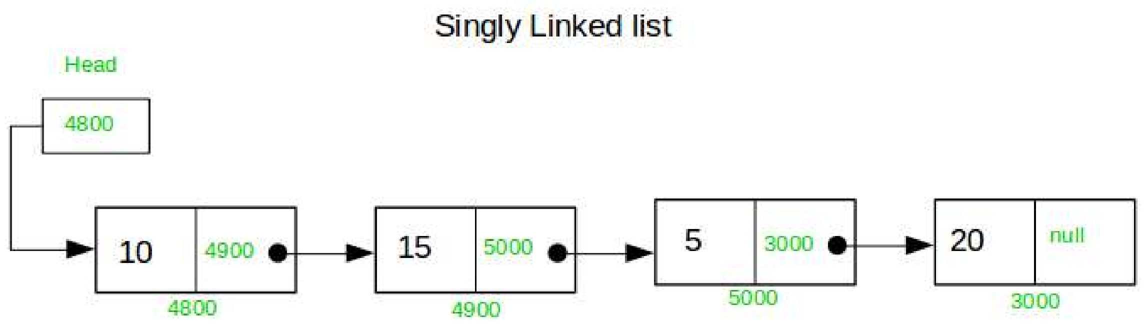
Operations of Linked Lists

- **Insertion**:
  - Adding a new node to the list.
- **Search**:
  - Finding an element in the list (iterative or recursive).
- **Length Calculation**:
  - Finding the number of nodes in the list (iterative or recursive).
- **Deletion**:
  - Removing a node from the list (by key or position).
- **Other Operations**:
  - Getting the Nth node, reversing the list, etc.

Linked lists are used for implementing stacks, queues, and other abstract data types. They are also essential for dynamic memory allocation and handling various data structures.
□□

2.) Singly Linked Lists: Representation and Operations

A singly linked list is a linear data structure where elements (nodes) are arranged in a sequential order.



nodes are not stored in contiguous memory locations. Instead, each node contains:

1. **Data:** This field stores the actual information or value held by the node.
2. **Next Pointer:** This pointer variable holds the memory address of the next node in the sequence. The last node's `next` pointer typically points to `NULL`, indicating the end of the list.

Representation in C:

Here's a basic C structure to represent a node in a singly linked list:

C

```c
struct Node {
  int data; // Data field to store information
  struct Node* next; // Pointer to the next node
};
```

Head Pointer:

A singly linked list typically has a head pointer, which points to the first node in the list. If the list is empty, the head pointer will be `NULL`.

C

```c
struct Node* head = NULL; // Initially an empty list
```

Operations on Singly Linked Lists:

1. Traversal:

To visit each node in the list, you start from the head pointer and follow the `next` pointers until you reach a node with a `NULL` pointer (the last node). Here's an example:

C

```c
void printList(struct Node* head) {
  while (head != NULL) {
    printf("%d ", head->data);
    head = head->next;
  }
  printf("\n");
}
```

2. Search:

To search for a node with a specific data value, you traverse the list starting from the head and compare the data of each node with the target value.

C

```c
struct Node* search(struct Node* head, int target) {
  while (head != NULL) {
    if (head->data == target) {
      return head; // Node found
    }
    head = head->next;
  }
  return NULL; // Node not found
}
```

3. Insertion:

There are different scenarios for insertion in a singly linked list:

- **Insertion at the beginning:** Create a new node, update its `data` field and make its `next` pointer point to the current head. Then, update the head pointer to point to the new node.
- **Insertion at the end:** Traverse the list to find the last node (the one with `next` as `NULL`). Create a new node, set its `data` and `next` (to `NULL`). Then, update the `next` pointer of the last node to point to the newly created node.
- **Insertion after a specific node:** Traverse the list to find the node before the desired insertion position. Create a new node, set its `data` and `next` pointer to point to the node currently after the position. Then, update the `next` pointer of the previous node to point to the new node.

4. Deletion:

Deletion also involves different scenarios:

- **Deletion from the beginning:** If the list is not empty, update the head pointer to point to the second node (the current head's `next`). Free the memory of the deleted node.
- **Deletion from the end:** Traverse the list to find the second last node (the one whose `next` points to the last node). Set its `next` pointer to `NULL` (effectively removing the last node). Free the memory of the deleted node.

- **Deletion of a specific node:** Traverse the list to find the previous node of the node to be deleted. Update the `next` pointer of the previous node to point to the node after the one being deleted. Free the memory of the deleted node.

  Remember: When deleting nodes, it's crucial to properly handle memory deallocation to prevent memory leaks.

These are the fundamental operations for singly linked lists. With practice, you can implement these operations and create more complex functionalities using singly linked lists in your C programs.

Here are the C code examples for Singly Linked List and Doubly Linked List:

1. Singly Linked List:

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

struct Node {
  int data;
  struct Node* next;
};

// Function prototypes
void insertAtBeginning(struct Node** head_ref, int new_data);
void insertAtEnd(struct Node** head_ref, int new_data);
void printList(struct Node* node);

int main() {
  // Empty list
  struct Node* head = NULL;

  // Inserting nodes at the beginning
  insertAtBeginning(&head, 10);
  insertAtBeginning(&head, 20);
  insertAtBeginning(&head, 30);

  // Print the linked list
  printf("Linked list: ");
  printList(head);

  // Inserting a node at the end
  insertAtEnd(&head, 40);

  // Print the linked list again
  printf("\nLinked list after insertion at end: ");
  printList(head);

  return 0;
}

// Inserts a new node at the beginning of the list
void insertAtBeginning(struct Node** head_ref, int new_data) {
  // Allocate memory for the new node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

  // Put in the data
  new_node->data = new_data;

  // Make next of new node as head
  new_node->next = (*head_ref);

  // Move the head to point to the new node
  (*head_ref) = new_node;
}

// Inserts a new node at the end of the list
void insertAtEnd(struct Node** head_ref, int new_data) {
  // Allocate memory for the new node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

  // Put in the data
  new_node->data = new_data;
  new_node->next = NULL;

  // If the list is empty, make the new node as head
  if (*head_ref == NULL) {
    (*head_ref) = new_node;
    return;
  }
  // Traverse the list to find the last node
  struct Node* last = *head_ref;
  while (last->next != NULL) {
    last = last->next;
  }

  // Change the next of last node
  last->next = new_node;
  return;
}

// Prints the contents of the linked list
void printList(struct Node* node) {
  while (node != NULL) {
    printf("%d ", node->data);
    node = node->next;
  }
  printf("\n");
}
```

4.) Doubly Linked Lists

- A doubly linked list (DLL) is a type of linked list where each node contains two pointers: one pointing to the previous node and another pointing to the next node.
- It allows traversal in both forward and backward directions.
- Here's the structure of a DLL node in C:

```
struct Node {
    int data;
    struct Node* prev; // Pointer to the previous node
    struct Node* next; // Pointer to the next node
};
```

Operations on Doubly Linked Lists

1. **Insertion**:
   - Inserting a new node at the beginning, end, or a specific position.
   - Example: Insert a node with a given value at the end of the list.
2. **Deletion**:
   - Removing a node from the beginning, end, or a specific position.
   - Example: Delete the first occurrence of a specific value from the list.
3. **Traversal**:
   - Traversing the DLL in both forward and backward directions.
   - Example: Print the elements of the list.
4. **Reverse a Doubly Linked List**:
   - Reverse the order of elements in the list.

DOUBLE LINKED LIST

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

  int data;

  struct Node* prev;

  struct Node* next;

};

// Function prototypes

void push(struct Node** head_ref, int new_data);
void insertAfter(struct Node* prev_node, int new_data);

void deleteNode(struct Node** head_ref, int key);

void printList(struct Node* node);

int main() {

  // Empty list

  struct Node* head = NULL;

  // Inserting nodes in the beginning

  push(&head, 10);

  push(&head, 20);

  push(&head, 30);

  // Print the doubly linked list

  printf("Doubly Linked List: ");

  printList(head);

  // Insert a node after the second node

  insertAfter(head->next, 40);

  // Print the doubly linked list again
  printf("\nDoubly Linked List after insertion: ");

  printList(head);

  // Delete the node with data 20

  deleteNode(&head, 20);

  // Print the doubly linked list after deletion

  printf("\nDoubly Linked List after deletion: ");

  printList(head);

  return 0;

}

// Function to insert a node at the beginning of the list

void push(struct Node** head_ref, int new_data) {

  // Allocate memory for the new node

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

  // Put in the data

  new_node->data = new_data;

  // Make next of new node as head and previous as NULL
  new_node->next = (*head_ref);

  new_node->prev = NULL;

  // If the linked list is empty, make the new node as head

  if (*head_ref != NULL) {

    (*head_ref)->prev = new_node;

  }

  // Move the head to point to the new node

  (*head_ref) = new_node;

}

// Function to insert a node after a given node

void insertAfter(struct Node* prev_node, int new_data) {

  // If the previous node is NULL, then there is no list

  if (prev_node == NULL) {

    printf("The given previous node cannot be NULL");

    return;

  }
```

```
 // Allocate memory for the new node

 struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

 // Put in the data

 new_node->data = new_data;

 // Make next of new node as the next of prev_node

 new_node->next = prev_node->next;

 // Make next of prev_node as new node

 prev_node->next = new_node;

 // Make prev of new node as prev_node

 new_node->prev = prev_node;

 // If the new node is not the last node, make the prev of next of new
node as new node

 if (new_node->next != NULL) {

   new_node->next->prev = new_node;

 }

}
```

```
// Function to delete a node in the linked list

void deleteNode(struct Node** head_ref, int key) {

  // Store head node

  struct Node* temp = *head_ref;

  // If list is empty or head is the key

  if (*head_ref == NULL || (*head_ref)->data == key) {

    *head_ref = (*head_ref)->next; // Change head

    if (*head_ref != NULL) {

      (*head_ref)->prev = NULL; // If head is not NULL, then make prev of
head as NULL

    }

    free(temp);

    return;

  }

  // Search for the key to be deleted

  while (temp != NULL && temp->data != key) {

    temp = temp->next;
  }

  // If key is not found

  if (temp == NULL) return;

  // If the node to be deleted is not a last node

  if (temp->next != NULL) {

    temp->next->prev = temp->prev;

  }

  // Change prev of next node
```

Circular Linked Lists

- A circular linked list is a linked list where all nodes are connected to form a circle.
- In a circular linked list, the first node and the last node are connected to each other, forming a circle.
- There is no NULL at the end.
- Circular linked lists are used in various applications, including:
  - Implementation of circular data structures like circular queues and circular buffers.
  - Undo-redo operations in text editors and software programs.
  - Music player playlists.
  - Cache memory management.

Arrays vs. Linked Lists

1. **Arrays**:
   - **Data Storage**:
     - Elements in arrays are stored in **contiguous memory locations**.
     - Allows faster access to an element at a specific index.
   - **Flexibility**:
     - Fixed size (determined during declaration).
     - Not suitable for dynamic memory allocation.
   - **Insertion/Deletion**:
     - Costly for insertion or deletion (requires shifting elements).
   - **Advantages**:
     - Efficient random access.
     - Simple and easy to use.
     - Cache-friendly due to contiguous memory.
   - **Disadvantages**:
     - Fixed size.
     - Inefficient for insertions/deletions.
2. **Linked Lists**:
   - **Data Storage**:
     - Elements in linked lists are **not stored in contiguous locations**.
     - Each element has a reference to the next element.
   - **Flexibility**:
     - Dynamic size (can grow or shrink).
     - Suitable for dynamic memory allocation.
   - **Insertion/Deletion**:
     - Efficient for insertion or deletion (no shifting required).
   - **Advantages**:
     - Dynamic size.

        - Efficient insertions/deletions.
        - Suitable for implementing other data structures.
    - **Disadvantages**:
        - Slower random access.
        - Extra memory overhead for pointers.

5.) Applications of Linked Lists

1. **Implementation of Stacks and Queues**:
    - Linked lists are used to implement both stacks and queues.
2. **Graph Representation**:
    - Adjacency list representation of graphs uses linked lists to store adjacent vertices.
3. **Dynamic Memory Allocation**:
    - Linked lists of free blocks are used for dynamic memory allocation.
4. **Sparse Matrices**:
    - Linked lists can represent sparse matrices efficiently.
5. **Music Players and Image Viewers**:
    - Linked lists allow navigation between previous and next elements (songs, images, etc.).
6. **GPS Navigation Systems**:
    - Linked lists store and manage locations and routes.
7. **Undo/Redo Functionality**:
    - Many software applications use linked lists to implement undo/redo functionality.
8. **Symbol Tables in Compilers**:
    - Linked lists build symbol tables for identifiers in programs.

Remember, linked lists are versatile and find applications in both computer science and real-world scenarios! □□