Data Structures

Unit 3 : Linked Lists

1. Introduction to Linked Lists

i) Definition :
- 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.
- Each node consists of :
  - i) Data → stores the actual value
  - ii) Pointer (Next) - stores the address of the next node
- Unlike arrays, LL do not require contiguous memory allocation, making them more memory efficient in dynamic scenarios

ii) Linked list as an abstract Data type :
- A linked list ADT supports operations like insertion, deletion, traversal, searching, reversing and concatenation.
- It follows the principle of dynamic memory allocation, which allows it to grow and shrink as needed.
- Basic operations on a LL.
  - i) Insertion : Adding an element at the beginning, end, or at a specific position.
  - ii) Deletion : Removing an element from the LL
  - iii) Traversal : Accessing all elements in sequence
  - iv) Reversing : Changing the order of elements
  - v) Copying : Creating a duplicate LL
  - vi) Concatenation : Merging two LL into one
  - vii) Deleting the entire list : Freeing all allocated memory
  - viii) Searching : Finding an element in LL

iii) Comparision of sequential (Array) and Linked List Organization

Array	Linked List
• Contiguous (fixed size) memory allocation	• Dynamic (grows and shrinks as needed) memory allocation
• Insertion / deletion is costly requires shifting element	• Insertion/deletion is effecient (only pointer updates)
• Access time - O[1] - direct indexing	• O(n) access time - sequential traversal.
• Fixed size - can cause wastage	• Uses only required memory.
• No extra memory needed.	• Requires additional memory for pointers.
• Simple to implement	• Complex due to pointers

iv) Advantages of Linked List :

- Dynamic Memory : Efficient use of memory as nodes are allocated dynamically.
- Efficient insertions/deletions : No need to shift elements like arrays
- Flexible Size : Can grow or shrink as needed.
- Ease of Modification : Supports efficient modifications compared to arrays.

v) Disadvantages of Linked List :

- Extra memory usage : Requires additional memory for pointers.
- Sequential Access : Cannot access elements directly like arrays.
- More Complex Implementation : Requires careful handling of pointers
- Cache inefficiency : Not contiguous in memory, leading to slower access times.

vi) Realization of Linked list

i) Using Arrays

- LL can be implemented using arrays, where each element contains the data & reference to the next index in the array.
- How it works :
  i) Each index in the array acts as a node
  ii) A separate array maintains the reference to the next node.
- Example :-

```
struct Node {
    int data;
    int next;
};
const int SIZE = 100;
Node list[SIZE];
int head = -1, freeIndex = 0;
void insert (int data) {
    if (freeIndex >= SIZE){
        cout << "List full";
        return;
    }
    list[freeIndex].data = data;
    list[freeIndex].next = head;
    head = freeIndex;
    freeIndex ++;
}
```

ii) Using Dynamic Memory Management

- Memory for nodes is allocated dynamically at runtime using pointers.
- How it works :
  i) Each node is dynamically created using 'new'.
  ii) 'next' pointer connects node
  iii) Memory is freed using 'delete'.
- Example :-

```
class Node {
  public :
    int data;
    Node* next;
    Node (int val) {
      { data = val; next = NULL; }
};
class linkedList {
  Node * head;
public :
  linkedList () { head = NULL; }
  void insert (int val) {
    Node * newNode = new Node (val);
    newNode -> next = head;
    head = newNode;
  }
```

```
void display()
{
    int temp = head;
    while (temp != -1) {
        cout<< list[temp].data <<"->";
        temp = list[temp].next;
    }
    cout<<"NULL\n";
}
int main() {
    insert(10);
    insert(20);
    insert(30);
    display();
    return 0;
}
```

```
void display() {
    Node* temp = head;
    while (temp) {
        cout<<temp->data<<"->";
        temp = temp->next;
    }
    cout << "NULL\n";
}
};
int main() {
    linkedList l1;
    l1.insert(10);
    l1.insert(20);
    l1.insert(30);
    l1.display();
    return 0;
}
```

vii)

Header Node

- It is an extra node present at the beginning of the LL, which contains metadata or act as a starting reference.
- How it works : It does not store useful data but helps in list traversal and management
- Example :

```
class Node {
  public:
    int data;
    Node* next;
    Node(int val) {
        data = val;
        next = NULL;
    }
};
```

```
class linkedlist {
    Node* head;
public:
    linkedlist() { head = NULL; }
    void insert(int val) {
        Node* newNode = new Node(val);
        newNode->next = head->next;
        head->next = newNode;
    }
    void display() {
        Node* temp = header->next;
        while (temp) {
            cout<<temp->data<<"->";
            temp = temp->next;
        }
        cout << NULL\n";
    }
};
```

2. Linked List Operations :

i) Insert Node at Beginning :

```
void insertAtBeginning(int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    head = newNode;
}
```

ii) Delete a node

```
void deleteFirstNode() {
    if (head == nullptr)
        return;
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

iii) Traverse Linked List

```
void traverseList() {
    Node* temp = head;
    while (temp) {
        cout << temp->data << "->";
        temp = temp->next;
```

iv) Copy into another LL

```
Node* copyList(Node* head) {
    if (!head)
        return nullptr;
    Node* newhead = new Node
                    (head->data);
    Node* current = head->next;
    Node* newCurrent = newHead;
    while (current) {
        newCurrent->next = new Node
                           (current->data);
        newCurrent = newCurrent->next;
        current = current->next;
    }
    return newHead;
}
```

v) Reverse a linked list

```
void reverseList() {
    Node* prev = nullptr;
    Node* current = head;
    Node* next;
    while (current) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}
```

vi) Delete a linked list

```
void deleteList(Node* &head) {
    Node* temp;
    while (temp) {
        temp = head;
        head = head->next;
        delete temp;
    }
}
```

vii) Concatenate 2 singly LL

```
void concatenatelist( Node* head1,
        Node* head2) {
    if (!head1)
        return head2;
    Node* temp = head1;
    while (temp->next) {
        temp = temp->next;
    }
    temp->next = head2;
}
```

viii) Searching an element

```
bool search(int key) {
    Node* temp = head;
    while (temp) {
        if (temp->data == key)
            return true;
        temp = temp->next;
    }
    return false;
}
```

ix) Insert a node at the end

```
void insertAtEnd(int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

3. Types of Linked List

M T W T F S
Page No.:
Date:

1) Linear Linked List :

- It is a sequence of nodes where each node points to the next node and the last node points to NULL.

```
void display (Node* head) {
  while (head) {
    cout << head -> data << " -> ";
    head = head -> next;
  }
  cout << "NULL" << endl;
}
```

2) Doubly Linked List

- It contains pointers to both the previous and next nodes.
- Each node has both next & prev pointers.
- Allows bidirectional traversal
- Operations : Insert, Delete, traverse /~~and~~ search.

- insert after an element

```
void insertAfter (Node* prevNode,
                  int val) {
  if (! prevNode)
    return;
  Node* newNode = new Node
                  (val);
  newNode -> next = prevNode
                  -> next;
  prevNode -> next = newNode;
  newNode -> prev = prevNode;
  if (newNode -> next)
    NewNode -> next -> prev = newNode;
}
```

- Delete an element

```
void deleteNode (Node* & head,
                 Node* del) {
  if (!head || ! del)
    return;
  if (head == del)
    head = del -> next;
  if (del -> next)
    del -> next -> prev = del -> prev;
  if (del -> prev)
    del -> prev -> next = del -> next;
  delete del;
}
```

- Read & display

```
void display (Node* head) {
  while (head) {
    cout << head -> data << " -> ";
    head = head -> next;
  }
  cout << "NULL" << endl;
}
```

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

3) Generalized Linked List.

- Stores heterogenous data.
- Used in expression evaluation, file systems, and AI.
- Unlike standard linked list, a node may contain a pointer to another list.