Question bank
End-Sem Examination- Winter 2023

Academic Year:2023-2024 **Semester: I**

Name of Programme:Pattern: 2022

Name of Course:Fundamentals of Data Structure **CourseCode:COMP222001**

==========

UNIT 3

a) Design a program to implement a doubly linked list for inserting an element at the given position, and calculate its time complexity.

Ans.

Now, let us discuss inserting a node in DLL. To insert a node, say `Current`, we have to modify four links as each node points to its predecessor as well as successor. Let us assume that the node `Current` is to be inserted in between the two nodes say `node1` and `node2`. We have to modify the following links:

`node1->Next, node2->Prev, Current->Prev,` and `Current->Next`

Program for insertion of a node at the given position :

PROGRAM CODE 6.12

```
void DList :: Insert_Pos(DLL_Node* NewNode, int pos)
{
   DLL_Node *temp = Head;
   int count = 1;
   if(Head == Null)
      Head = Tail = NewNode;
   else if(pos == 1)      // insert before head
   {
      NewNode->Next = Head;
      Head->Prev = NewNode;
      Head = NewNode;
   }
   else
   {
      while(count != pos)
      {
         temp = temp->Next;
         if(temp != Null)
            count++;
         else
            break;
      }
      if(count == pos)
      {
         (temp->Prev)->Next = NewNode;
         NewNode->Prev = temp->Prev;
         temp->Prev = NewNode;
      }
      else
       cout << "The node position is not found" << endl;
   }
}
```

Time Complexity :
Insertion at beginning : O(1)
Insertion at any other position : Time complexity: O(n)

b) Develop a pseudocode for deleting an element from singly linked list at the given position and explain the same using pictorial representation.

Ans.

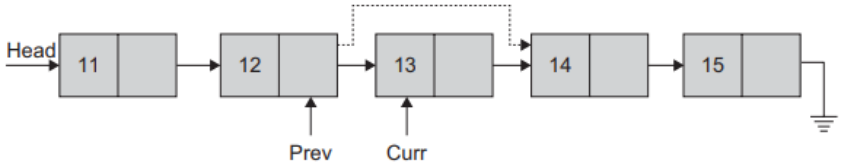


Fig. 6.21 Link manipulations for deletion of a node

```
void Llist :: DeleteNode(int pos)
{
   int count = 1, flag = 1;
   Node *curr, *temp;
```

```
   temp = Head;
   if(pos == 1)
   {
      Head = Head->link;
      delete temp;
   }
   else
   {
      while(count != pos - 1)
      {
         temp = temp->link;
         if(temp == Null)
         {
            flag = 0; break;
         }
         count++;
      }
      if(flag == 1)
      {
         curr = temp->link;
         temp->link = curr->link;
         delete curr;
      }
      else
         cout << "Position not found" << endl;
   }
}
```

c) List various operations performed on the singly linked list and explain search operation in detail.

Ans. operations performed on the singly linked list

1. **Traversal**: Visiting each node in the list and accessing its data. This can be done iteratively or recursively.
2. **Insertion**: Adding a new node at the beginning, end, or after a specific node.
3. **Deletion**: Removing a node from the beginning, end, or after a specific node.
4. **Searching**: Finding a specific node based on its data value.
5. **Length**: Determining the total number of nodes in the list.
6. **Print**: Displaying the data of each node in the list.

Searching in Singly Linked Lists:

Searching for a specific element in a singly linked list is an important operation. Here's how it works:

1. Input:
- The head pointer of the linked list.
- The element to search for.

2. Algorithm:
- Initialize a temporary pointer (temp) to the head of the list.
- Traverse the list by repeatedly moving the temp pointer to the next node.
- At each node, compare the data in the node with the search element.
    - If the data matches, return the node containing the element.
    - If the data doesn't match, keep traversing until the temp pointer reaches null (end of list).
- If the entire list is traversed without finding a match, return null to indicate the element is not present.

3. Time Complexity:
- Searching in a singly linked list is linear time complexity (O(n)).
- This is because, in the worst-case scenario, the search needs to traverse the entire list to reach the end.

d)Show how to reverse a singly linked list

Ans.

PROGRAM CODE 6.28

```
void Llist :: Reverse()
{
    Node *curr, *prev, *next;
    prev = Head;
    curr = Head->link;
```

JCTURES USING C++

```
    prev->link = Null
    while(temp != Null)
    {
       next = temp->link;
       temp->link = prev;
       prev = temp;
       temp = next;
    }
    head = prev;
}
```

e) Develop and explain a pseudo-code to merge two singly linked lists.
Ans .

PROGRAM CODE 6.29

```
void Llist :: concatanate(Llist A)
{
   Node *X, *Y;
   X = Head;
   Y = A.Head;
   while(X->link != Null)
   {
      X = X->link;
   }
   X->link = Y;
   Head = X;
}

//A call to concatenate:
{
   Llist L1, L2;
   L1.Create(); L2.Create();
   L1.Concatanate(L2);
}
```

f) What are the fields of a node of singly and doubly link list and explain how to point the first node in the link list.

Ans. Fields of a Node

Singly Linked List:

A node in a singly linked list has two fields:

1. **Data**: This field stores the actual data element, like an integer, character, or a complex object.
2. **Next**: This field stores the memory address of the next node in the list. It acts as a pointer that links the nodes together.

Doubly Linked List:

A node in a doubly linked list has three fields:

1. **Data**: Similar to singly linked lists, this field stores the actual data element.
2. **Next**: This field stores the memory address of the next node in the list.
3. **Prev**: This field stores the memory address of the previous node in the list. It allows for traversal in both directions.

Pointing the First Node

In both singly and doubly linked lists:

- The first node is typically referred to as the head.
- To point the first node, you need to create a pointer variable (e.g., head) and assign the memory address of the first node to it**.**

h) Show how to detect a cycle in a Singly linked list and also find the starting node of the cycle.

Ans.

There are two common algorithms to achieve this:

1. Two pointer approach (Floyd's cycle-finding algorithm):

- This approach uses two pointers, slow and fast.
- The slow pointer moves by one node at a time, while the fast pointer moves by two nodes at a time.
- If there is a cycle, the fast pointer will eventually catch up to the slow pointer inside the cycle.
- Once they meet, the cycle is confirmed.
- To find the starting node, reinitialize the slow pointer to the head and keep the fast pointer at the meeting point.
- Now, move both pointers one step at a time until they meet again. This point will be the starting node of the cycle.

2. **Hashing**:

This approach uses a hash table to store visited nodes.
Traverse the list and for each node, check if it exists in the hash table.
If it does, a cycle is detected

Q. Write ADT for Circular Linked List.
Ans.
Abstract Data Type (ADT) for Circular Linked List
Data:

- A collection of nodes, each containing:
    - Data element of any type
    - Pointer to the next node in the list

Operations:

- **IsEmpty**(): Checks if the list is empty.
- **GetHead**(): Returns the head node of the list.
- **Insert**(**data**): Inserts a new node with the given data after the last node.
- **InsertAt**(**data**, **index**): Inserts a new node with the given data at the specified index (0-based).
- **DeleteHead**(): Deletes the head node and updates the head pointer.
- **DeleteAt**(**index**): Deletes the node at the specified index (0-based) and updates the list.
- **Search**(**data**): Searches for a node with the given data and returns its index, or -1 if not found.
- **Traverse**(): Visits each node in the list and performs an action on its data.
- **Length**(): Returns the number of nodes in the list.

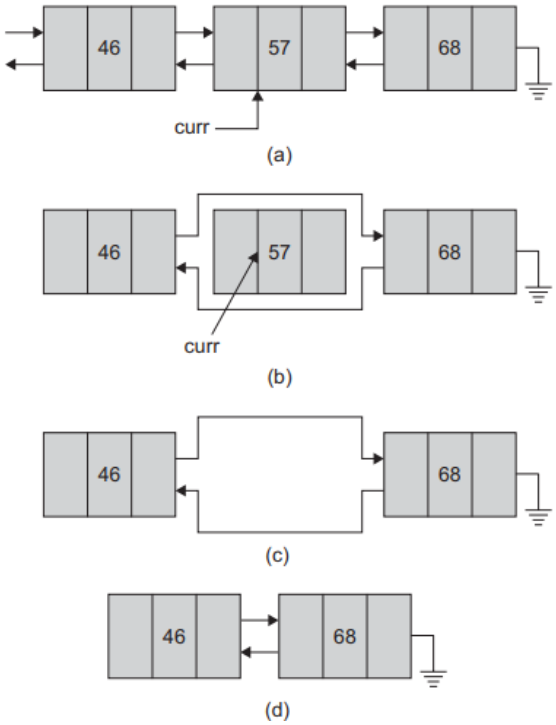
Properties:

- The last node points to the head node, forming a circular loop.
- Insertions and deletions can be performed at any point in the list.
- Searching requires traversing the entire list.
- Accessing a specific node requires starting from the head and traversing the list until the desired node is found.

j) What is a doubly linked list? Explain the process of deletion of an element from a doubly linked list with an example.
Ans. A doubly linked list is a linear data structure similar to a singly linked list, but with an additional pointer in each node. This extra pointer points to the previous node in the list, allowing traversal in both forward and backward directions.

Process of deletion a node from doubly linked list :

Deleting from a DLL needs the deleted node's predecessor, if any, to be pointed to the deleted node's successor. In addition, the successor, if any, should be set to point to the predecessor node as shown in Fig. 6.30.



Write theory of the above diagram by your own, that how we deleted that node

The core steps involved in this process are the following:

```
(curr->Prev)->Next = curr->Next;
(curr->Next)->Prev = curr->Prev;
delete curr;
```

k) Use Generalized Linked List to represent a polynomial of the following example.
Ans. a)9x^5 + 7xy^4 + 10x
Do on your own

l) List various operations performed on the doubly linked list and explain search operation in detail.

Ans. Doubly linked lists offer a more versatile and efficient data structure compared to singly linked lists due to their bi-directional traversal capability. Here are some common operations performed on doubly linked lists:

1. **Traversal**:
   - Forward traversal: Visiting each node in the list from head to tail, accessing the data and performing desired actions.
   - Backward traversal: Visiting each node in the list from tail to head, accessing the data and performing desired actions.
2. **Insertion**:
   - Insertion at the beginning: Adding a new node as the head of the list.
   - Insertion at the end: Adding a new node as the last node of the list.
   - Insertion at any position: Adding a new node after a specific node based on its index or data value.
3. **Deletion**:
   - Deletion at the beginning: Removing the head node of the list.
   - Deletion at the end: Removing the last node of the list.
   - Deletion at any position: Removing a specific node based on its index or data value.
4. **Searching**:
   - Searching by data value: Finding the node containing a specific data element.
   - Searching by index: Finding the node at a specific position in the list.
5. **Length**:
   - Determining the total number of nodes in the list.
6. **Reverse**:
   - Reversing the order of nodes in the list.
7. **Concatenation**:
   - Combining two doubly linked lists into a single list.
8. **Split**:
   - Dividing a doubly linked list into two separate lists.

Explanation of Search Operation:

1. **Input**:
   - The head pointer of the doubly linked list.
   - The data value to search for.
2. **Algorithm**:

There are two main approaches to search for a node in a doubly linked list:

a) Iterative approach:

1. Initialize two pointers: current and previous, both pointing to the head node initially.
2. Loop through the list while current is not null:
   - Compare the data of the current node with the search value.
   - If a match is found, return the current node.
   - Move both current and previous pointers to the next and previous nodes, respectively.
3. If the entire list is traversed without finding a match, return null.

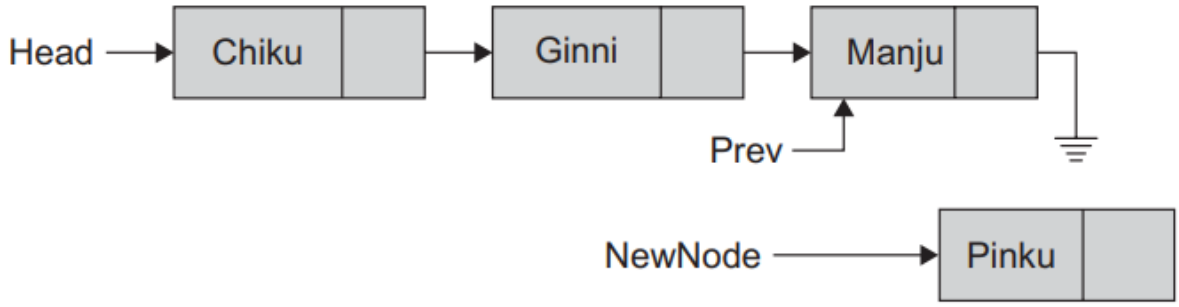
b) Recursive approach:

1. Define a recursive function that takes the head node and the search value as arguments.
2. If the head node is null, return null.

3. If the data of the head node matches the search value, return the head node.
4. Otherwise, call the function recursively with the next pointer and the search value.
5. Return the result of the recursive call.

3. Time Complexity:
Searching a doubly linked list is typically an O(n) operation in the worst case

Q. Write a CPP function to insert a new node at the end of a singly linked list.
Ans.




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

struct Node {
  int data;
  Node* next;
};
void addNode(Node* head, int data) {

  Node* newNode = new Node;
  newNode->data = data;
  newNode->next = NULL;

  if (head == NULL) {
    head = newNode;
    return;
  }

  Node* temp = head;
  while (temp->next != NULL) {
    temp = temp->next;
  }

  temp->next = newNode;
}
Int main()
```

```
{
  Node* head = NULL;


  addNode(head, 10);
  addNode(head, 20);
  addNode(head, 30);

  return 0;
}
```

Q. Write different types of linked list and compare them.
Ans.

S. No.	Singly linked list	Doubly linked list
1	In case of singly linked lists, the complexity of insertion and deletion is O(n)	In case of doubly linked lists, the complexity of insertion and deletion is O(1)
2	The Singly linked list has two segments: data and link.	The doubly linked list has three segments. First is data and second, third are the pointers.
3	It permits traversal components only in one way.	It permits two way traversal.
4	We mostly prefer a singly linked list for the execution of stacks.	We can use a doubly linked list to execute binary trees, heaps and stacks.
5	When we want to save memory and do not need to perform searching, we prefer a singly linked list.	In case of better implementation, while searching, we prefer a doubly linked list.
6	A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.

Q. Develop a pseudocode for representation of a polynomial using singly linked list.
Ans.

- Create Polynomial:
    1. Initialize an empty linked list.
    2. Set the head pointer to NULL.
- Add Term:
    1. Create a new node with the given coefficient and exponent.
    2. If the list is empty, set the new node as the head of the list.
    3. Otherwise, traverse the list to find the appropriate insertion position:

        - If the current term's exponent is greater than the new term's exponent, insert the new node after the current node.
        - Otherwise, move to the next node and repeat the comparison.
    4. Update the next pointer of the new node and the previous node (if not head) to maintain the linked structure.

Q. Develop a psudocode for inserting a new header element at the beginning of the circular linked list.
Ans.

1. Check if the list is empty:
    - If head is NULL, then the list is empty.
        - Set head to **new_node**.
        - Set **new_node->next** to itself (to form the circular structure).
    - Otherwise, the list is not empty.
2. Allocate memory for a temporary pointer:
    - Create a pointer variable called **temp**.
3. Set temp to point to the current head node:
    - **temp = head.**
4. Traverse the list to find the last node:
    - **While temp->next != head:**
        - **temp = temp->next.**
5. Insert the new header node:
    - Set **new_node->next** to the current head node: **new_node->next = head.**
    - Set the next pointer of the last node to the new head node**: temp->next = new_node.**
    - Update the head pointer to point to the new head node: **head = new_node**

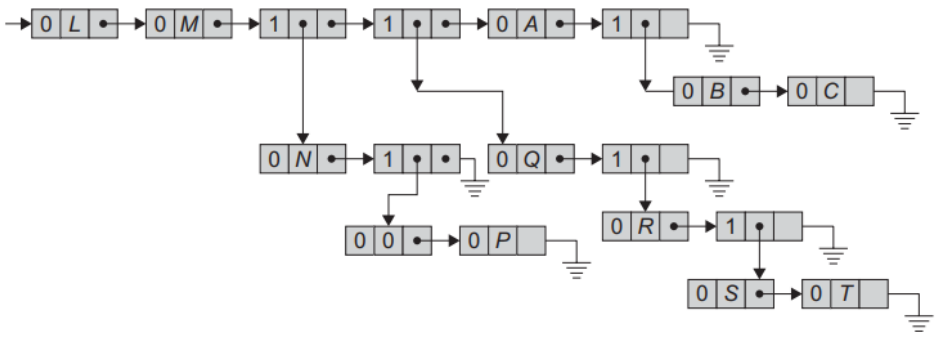
Q. Write a CPP program to copy one linked list into another linked list.
Ans.

PROGRAM CODE 6.25

```
Node *Llist :: CopyList()
{
   Node *current = Head;
   Node *newList = Null;
   Node *Tail = Null;
   while(current != Null)
   {
      if(newList == Null)
      {
         newList = new Node;
         newList->Data = current->Data;
         newList->link = Null;
         Tail = newList;
      }
      else
      {
         Tail->link = new Node;
         Tail = Tail->link;
         Tail->Data = current->Data;
         Tail->link = Null;
      }
      current = current->link;
   }
   return(newList);
}
```

Q. Use a Generalized Linked List to represent the following example.
(p,q(r,s(t,u,v),w)x,y)=L
Ans. Try It on your own, refer the below figure

The set $X = \{L, M, \{N, \{O, P\}\}, \{Q, \{R, \{S, T\}\}, A, \{B, C\}\}$ is pictorially represented using a generalized linked list in Fig. 6.76.



Q. Design a program that merges two sorted linked lists into a single sorted linked list.
Ans. Similar to Concatenate problem already done.

t)Develop a function to detect a loop in a linked list and break the loop if one exists.
Ans.

```
bool detectAndBreakLoop(Node* head) {
  if (!head || !head->next) return false; // List is empty or has only one node

  Node* slow = head;
  Node* fast = head->next;

  while (slow != fast) {
    if (!fast || !fast->next) return false; // Reached end of list, no loop

    slow = slow->next;
    fast = fast->next->next;
  }

  // Loop exists

  Node* prev = slow;
  slow = slow->next;
  while (slow != head) {
    prev = slow;
    slow = slow->next;
  }

  prev->next = nullptr; // Break the loop

  return true; // Loop found and broken
}
```

Q. Evaluate the trade-offs between a singly linked list and a doubly linked list in terms of memory usage and performance.
Ans. Memory Usage:

- **Singly Linked List:**
  - Each node requires only two memory locations: one for data and one for the next pointer.
  - More memory-efficient than a doubly linked list.
- **Doubly Linked List:**
  - Each node requires three memory locations: one for data, one for the next pointer, and one for the previous pointer.
  - Requires slightly more memory per node than a singly linked list.

Performance:

- **Insertion/Deletion**:
  - **Singly Linked List:**
    - Insertion at the beginning is efficient (only need to modify the head pointer).
    - Insertion/deletion in the middle requires iterating through the list until the target node is found.
    - Deletion requires updating the previous node's next pointer.
  - **Doubly Linked List:**

        - Insertion/Deletion at the beginning or end is efficient (can use the previous pointer).
        - Insertion/deletion in the middle is easier (modify both next and previous pointers).
        - Deletion does not require finding the previous node.
- **Traversal**:
    - Singly Linked List:
        - Only allows forward traversal (from head to tail).
    - Doubly Linked List:
        - Allows traversal in both directions (forward and backward).
- **Search**:
    - Singly Linked List:
        - Requires starting from the head and iterating until the target element is found.
    - Doubly Linked List:
        - Can start from either head or tail and iterate until the target element is found.
        - Potentially faster for searching near the end of the list.

Q. Assess the advantages and disadvantages of using a linked list compared to an array for dynamic data structures.

Ans. Advantages of Linked Lists over Arrays for Dynamic Data Structures

- **Dynamic size**: Linked lists can grow or shrink dynamically during runtime, unlike arrays whose size is fixed once declared. This makes them ideal for situations where the data size is unknown beforehand or changes frequently.
- **Efficient insertion/deletion**: Inserting or deleting elements at any position in a linked list is relatively efficient, requiring only updating the pointer of the previous and next nodes. In contrast, arrays require shifting all elements after the insertion/deletion point, which can be time-consuming for large arrays.
- **No memory waste**: Arrays allocate contiguous memory blocks regardless of the actual data size, potentially leading to wasted space. Linked lists allocate memory only for the data itself, making them more memory-efficient.
- **Better for non-sequential access**: Linked lists are better suited for situations where non-sequential access to data is frequent. Accessing a specific element in a linked list requires iterating through the list until the element is found, which is still faster than shifting elements in an array.

Disadvantages of Linked Lists compared to Arrays

- **Memory overhead**: Each node in a linked list requires additional space for the pointer, making them slightly less memory-efficient than arrays for storing the same amount of data.
- **Slower random access**: Random access (accessing an element by its index) is much slower in linked lists than in arrays. Arrays allow direct access to any element by calculating its memory address based on its index. Linked lists, however, require iterating through the list until the desired index is reached.

- **Complexity**: Implementing and manipulating linked lists can be slightly more complex than working with arrays due to the need to manage pointers and handle edge cases.
- **Cache efficiency**: Linked lists tend to be less cache-efficient than arrays due to their non-contiguous memory allocation. This can lead to performance penalties on some architectures

Q. Compare the time complexity of inserting an element at the beginning of a linked list with inserting at the end.

Ans.

Inserting at the Beginning:

- Singly Linked List: O(1)

Inserting at the End:

- Singly Linked List: O(n)

Explanation:

Singly Linked Lists:

- **Inserting at the Beginning:**
  - Only requires modifying the head pointer to point to the new node.
  - No need to iterate through the list.
  - Time complexity is constant, O(1).
- **Inserting at the End:**
  - Requires traversing the entire list until reaching the last node.
  - Need to update the last node's next pointer to point to the new node.
  - Time complexity depends on the list length, O(n).

Q. How does a doubly linked list differ from a singly linked list in terms of structure and advantages?

Ans.

Structurally

Both singly and doubly linked lists are data structures built on nodes containing data and a pointer to the next node. However, they differ in the number of pointers per node:

- Singly linked list: Each node has one pointer (next) that points to the next node in the list.
- Doubly linked list: Each node has two pointers (next and previous) that point to the next and previous nodes in the list.

This additional pointer in the doubly linked list creates a bidirectional chain, allowing for traversal in both forward and backward directions.

Advantages

Here's how these structural differences translate to advantages:

Double Endedness:

- Doubly linked lists:
  - Can be traversed in both directions, making them efficient for operations requiring bidirectional access, like implementing a dictionary or performing undo/redo functionalities.
- Singly linked lists:

    - Can only be traversed in one direction, making them less flexible for such operations.

Insertion/Deletion:

- Doubly linked lists:
    - Easier and faster insertion/deletion in the middle of the list, as both next and previous pointers need to be updated.
    - Faster deletion, as finding the previous node is not required.
- Singly linked lists:
    - Insertion/deletion in the middle requires traversing the list to find the target node, making it slower.
    - Deletion requires finding the previous node to update its next pointer, potentially slower.

Searching:

- Doubly linked lists:
    - Can search in both directions, potentially faster if the target element is closer to the tail.
- Singly linked lists:
    - Can only search in the forward direction, potentially slower if the target element is closer to the tail.

Complexity:

- Insertion/Deletion:
    - Singly Linked List:
        - Beginning: O(1)
        - End: O(n)
        - Middle: O(n)
    - Doubly Linked List:
        - Beginning: O(1)
        - End: O(1)
        - Middle: O(1)
- Traversal:
    - Singly Linked List:
        - Forward: O(n)
    - Doubly Linked List:
        - Forward: O(n)
        - Backward: O(n)
- Search:
    - Singly Linked List:
        - O(n)
    - Doubly Linked List:
        - O(n)

Memory Usage:

- Singly Linked List:
    - More memory-efficient, as each node has one pointer.
- Doubly Linked List:
    - Requires more memory, as each node has two pointers.