Data Structure

Chapter 5: Linked List

Study Notes for Engineering Students

5.1 Introduction

A linked list is a linear data structure in which elements, called nodes, are not stored in contiguous memory locations. Instead, each node holds its own data together with a reference (or link) to the next node in the sequence. This chain of links, starting from a special node called the head, allows the entire list to be traversed one node at a time.

Arrays store elements in adjacent memory cells and are accessed using an index, which makes array access very fast but makes insertion and deletion expensive, since shifting elements is often required. Linked lists solve this problem by allowing insertion and deletion in constant time once the correct position is located, because only a few pointers need to be updated rather than shifting the whole structure.

Linked lists are dynamic in nature: memory for a new node is allocated at run time, so the size of a linked list can grow or shrink during program execution without wasting memory or requiring the programmer to declare a fixed size in advance. This flexibility makes linked lists the underlying structure for many higher-level abstractions such as stacks, queues, graphs (adjacency lists) and dynamic memory allocators.

Key characteristics of a linked list:

- Dynamic size — grows and shrinks during execution.
- Efficient insertion and deletion, especially at the beginning of the list.
- No memory wastage due to over-allocation, unlike static arrays.
- Sequential access only — unlike arrays, a node cannot be accessed directly by index; the list must be traversed from the head.
- Extra memory is required for each node to store the address/link part, in addition to the data.

5.2 Terminologies

Before studying the operations on linked lists, it is important to understand the basic vocabulary used to describe their structure.

- **Node:** The basic building block of a linked list. Each node is a small record that stores a data item along with one or more links (pointers) to other nodes.
- **Address:** The memory location where a node is stored. In a linked list, nodes are scattered in memory, and the address of a node is what allows another node to “point” to it.
- **Pointer (Link):** A variable inside a node that stores the address of another node. The pointer is what physically connects one node to the next, forming the chain of the list.
- **Information (Data field):** The part of a node that stores the actual value or payload — for example, a number, a name, or a record — that the list is meant to hold.

- **Next:** The name commonly given to the pointer field of a node that stores the address of the following node in the list. Following the Next pointers repeatedly is how the list is traversed.
- **Null pointer:** A special pointer value (commonly written as NULL) that does not point to any node. The Next field of the last node in a singly linked list is set to NULL to indicate that the list ends there.
- **Empty list:** A linked list that contains no nodes at all. It is represented by setting the head pointer itself to NULL, meaning there is nothing to traverse.
- **Head/Start pointer:** A separate pointer, external to the nodes, that stores the address of the first node of the list. The head pointer is the only entry point into the list; if it is lost, the entire list becomes inaccessible.

5.3 Operations on List

A linked list supports three fundamental operations: searching, insertion and deletion. Each of these relies on traversal — moving from node to node using the Next pointer — since random access by index is not possible.

Searching

Searching a linked list means finding whether a given value is present, and if so, obtaining a pointer to the node that contains it.

- Start from the head pointer and set a temporary pointer to it.
- Compare the data of the current node with the target value.
- If they match, the search is successful and the node's address is returned.
- If not, move the temporary pointer to the next node using its Next field.
- If the temporary pointer becomes NULL before a match is found, the value is not present in the list.

Since every node must potentially be visited, searching a singly linked list takes O(n) time in the worst case, where n is the number of nodes.

Insertion

Insertion adds a new node to the list. Depending on where the node is added, insertion is classified into three cases.

1. Insertion at the beginning — the new node's Next pointer is set to the current head, and then the head pointer is updated to point to the new node. This is an O(1) operation.
2. Insertion at the end — the list is traversed until the last node (the one whose Next is NULL) is found, and that node's Next pointer is updated to point to the new node, whose own Next is set to NULL. This takes O(n) time unless a tail pointer is maintained.
3. Insertion at a given position — the list is traversed to the node just before the desired position; the new node's Next is set to that node's current Next, and that node's Next is then updated to point to the new node.

In every case, the order in which the pointers are updated matters: the new node must be linked to its successor before the predecessor is linked to the new node, otherwise the rest of the list is lost.

Deletion

Deletion removes an existing node from the list and, correspondingly, adjusts the pointers so the chain remains intact.

1. Deletion at the beginning — the head pointer is simply moved to point to the second node (head->Next), and the old first node is freed. This is an O(1) operation.
2. Deletion at the end — the list is traversed to the second-last node, whose Next pointer is then set to NULL after the last node is freed. This takes O(n) time.
3. Deletion of a node at a given position (or with a given value) — the node just before the target node is located, and its Next pointer is updated to skip over the target node and point directly to the node after it, before the target node is freed.

A special case is deleting the only node in a one-node list, after which the head pointer must be reset to NULL so the list correctly becomes empty.

5.4 Types of Lists

Linked List (Singly Linked List)

A singly linked list is the simplest form of linked list, in which each node has exactly one pointer field, Next, that points to the following node. Traversal is possible only in the forward direction, from the head towards the last node, whose Next field is NULL. Because there is only one link per node, singly linked lists use the least memory overhead among the linked structures, but operations that need the previous node — such as deleting a given node when only its own address is known — are less convenient, since the list would otherwise need to be traversed again from the head.

Circular Linked List

A circular linked list is a variation in which the Next pointer of the last node, instead of being NULL, points back to the first node (the head), forming a closed loop. There is no true “last” node in a circular list, since traversal can continue indefinitely around the circle.

Circular lists are especially useful when a list needs to be processed repeatedly in a round-robin fashion — for example, in CPU scheduling among multiple processes, or in multiplayer turn-based applications — because after the last element the traversal naturally continues from the first element again without needing to detect the end and restart manually.

- A circular list can be singly circular (only a forward link that wraps around) or doubly circular (both forward and backward links wrap around).
- An empty circular list is still represented by a NULL head pointer, just like a linear list.
- Care must be taken while traversing a circular list to stop after one full round (for example, by comparing the current pointer to the head) rather than looping forever, since there is no NULL terminator to signal the end.

5.5 Reverse and Merging Linked List

Reversing a Linked List

Reversing a singly linked list means changing the direction of every Next pointer so that the last node becomes the new head and the original head becomes the new last node. This is typically done iteratively using three pointers that track the previous node, the current node, and the next node, so that the link is not lost while it is being redirected.

1. Initialize a pointer prev to NULL and a pointer curr to the head of the list.
2. While curr is not NULL, store curr->Next in a temporary pointer next before it is overwritten.
3. Redirect curr->Next to point to prev instead of the original next node.
4. Move prev forward to curr, and move curr forward to the stored next pointer.
5. When curr becomes NULL, prev is pointing to the new first node; update the head pointer to prev.

This iterative approach visits each node exactly once, giving a time complexity of O(n) and requiring only a constant, O(1), amount of extra memory. Reversal can also be done recursively, but the recursive approach uses O(n) stack space due to the function call stack.

Merging Two Linked Lists

Merging combines two separate linked lists into a single list. The simplest form of merging is concatenation: the Next pointer of the last node of the first list is made to point to the head of the second list, so the two lists are joined end to end.

A more common and useful variant in practice is merging two already-sorted linked lists into one sorted linked list, which is also the key step in the merge sort algorithm for linked lists.

1. Compare the data in the current nodes of both lists.
2. Attach the node with the smaller value to the result list, and advance the pointer of the list from which it was taken.
3. Repeat the comparison and attachment step until one of the two lists is fully consumed.
4. Attach the entire remaining portion of the other list (which is already sorted) to the end of the result list.

Merging two sorted lists of sizes m and n in this manner takes O(m + n) time, since each node from both lists is visited and attached exactly once, and no extra array or additional list needs to be allocated beyond re-linking existing nodes.

5.6 Array, Stacks, Queues — Implementation Using List

Arrays and linked lists are the two basic ways of organising a linear collection of elements, and each can be used as the underlying storage for abstract data types such as stacks and queues.

Array-based vs Linked-list-based Implementation

- An array-based implementation has a fixed capacity decided at creation time; if the structure grows beyond this capacity it must be resized, usually by allocating a larger array and copying all elements over, which is an expensive O(n) operation.

- A linked-list-based implementation grows and shrinks node by node, so there is no fixed upper limit and no costly resizing step, at the cost of extra memory for the pointer field in every node and slightly slower access due to pointer chasing.
- Array-based structures allow O(1) random access to any element by index, which a linked list can never provide.

Stack Implementation Using a Linked List

A stack is a Last-In-First-Out (LIFO) structure, in which insertion (push) and deletion (pop) both happen at the same end, called the top. A linked list implements a stack very naturally by treating the head of the list as the top of the stack.

- Push — a new node is inserted at the head of the list, exactly as in beginning-insertion for a linked list; this is an O(1) operation and never requires resizing.
- Pop — the node at the head is removed and the head pointer is advanced to the next node, exactly as in beginning-deletion; this is also O(1).
- Peek/Top — the data of the head node is read without removing it.

Because both push and pop occur at the head, a linked-list stack needs no tail pointer and no shifting of elements, unlike an array-based stack, which is fine as long as it does not need to grow beyond its declared size.

Queue Implementation Using a Linked List

A queue is a First-In-First-Out (FIFO) structure, in which insertion (enqueue) happens at one end, called the rear, and deletion (dequeue) happens at the other end, called the front. A linked-list implementation of a queue keeps two external pointers, front and rear, in addition to the usual head-based traversal.

- Enqueue — a new node is created and linked after the current rear node, and the rear pointer is then updated to this new node; if the queue was empty, both front and rear are set to the new node.
- Dequeue — the node at the front is removed and the front pointer is advanced to the next node; if the queue becomes empty as a result, the rear pointer is also reset to NULL.

Maintaining an explicit rear pointer is essential here — without it, enqueue would require traversing the entire list every time to find the last node, turning an operation that should be O(1) into an O(n) one. With a rear pointer maintained, both enqueue and dequeue in a linked-list-based queue run in O(1) time, which is why linked lists are commonly preferred over plain arrays for implementing queues that need to grow arbitrarily large.

Multiple Choice Questions (MCQs)

Answer key is provided at the end of this section.

Q1. What does the Next field of a node in a singly linked list store?

(a) The data value of the node
(b) The address of the previous node
(c) The address of the following node
(d) The total number of nodes in the list

Q2. An empty linked list is represented by:

(a) A node whose data field is zero
(b) A head pointer set to NULL
(c) A node whose Next field points to itself
(d) A list with exactly one node

Q3. What is the time complexity of inserting a new node at the beginning of a singly linked list?

(a) O(1)
(b) O(n)
(c) O(log n)
(d) O(n^2)

Q4. In a circular linked list, the Next pointer of the last node points to:

(a) NULL
(b) The previous node
(c) The head (first) node
(d) Itself

Q5. Which of the following is NOT an advantage of a linked list over an array?

(a) Dynamic size
(b) Efficient insertion and deletion
(c) Constant-time random access by index
(d) No need to declare size in advance

Q6. Deleting the last node of a singly linked list, in the worst case, requires traversal that takes:

(a) O(1) time
(b) O(n) time
(c) O(log n) time
(d) O(1) space only, no time cost

Q7. Which extra pointer(s) are conventionally maintained to make enqueue operations O(1) in a linked-list-based queue?

(a) Only a head pointer
(b) Only a tail/rear pointer
(c) Both front and rear pointers
(d) No extra pointer is needed

Q8. While reversing a singly linked list iteratively, which pointer is used to temporarily save the next node before the link is redirected?

(a) prev
(b) curr

(c) next (a temporary pointer)
(d) head

Q9. A stack implemented using a linked list treats which end of the list as the 'top'?

(a) The last node
(b) The head (first) node
(c) The middle node
(d) Any node chosen at random

Q10. Merging two already-sorted linked lists of sizes m and n into one sorted list takes:

(a) O(m + n) time
(b) O(m * n) time
(c) O(log(m + n)) time
(d) O(1) time

Q11. The field of a node that holds the actual data value (not the link) is called the:

(a) Pointer field
(b) Information/Data field
(c) Head field
(d) Null field

Q12. Which of the following requires the LEAST pointer updates in a singly linked list?

(a) Insertion at the beginning
(b) Insertion at the end without a tail pointer
(c) Deletion at the end without a tail pointer
(d) Insertion at an arbitrary middle position

Q13. In a linked-list-based stack, the Pop operation is analogous to which linked list operation?

(a) Insertion at the beginning
(b) Deletion at the beginning
(c) Deletion at the end
(d) Searching

Q14. What happens if the head pointer of a linked list is accidentally lost (overwritten) without saving it elsewhere?

(a) The list automatically reverses
(b) The list becomes circular
(c) All the nodes of the list become inaccessible (memory leak)
(d) Nothing changes; the list still works normally

Q15. A doubly circular linked list differs from a singly circular linked list in that it:

(a) Has no head pointer
(b) Has both forward and backward links that wrap around
(c) Cannot represent an empty list
(d) Cannot be traversed in a loop

Answer Key

1. (c) 2. (b) 3. (a) 4. (c) 5. (c) 6. (b) 7. (c) 8. (c) 9. (b) 10. (a) 11. (b) 12. (a) 13. (b) 14. (c) 15. (b)

Broad / Descriptive Questions

1. Define a linked list. Explain how it differs from an array as a linear data structure, giving at least three points of comparison.

2. Explain the following terms with reference to a linked list, using a neat diagram: Node, Pointer, Null pointer, Head pointer.

3. Write an algorithm to search for a given value in a singly linked list. What is its time complexity, and why?

4. Describe, with algorithms, the three cases of inserting a node into a singly linked list: at the beginning, at the end, and at a given position.

5. Describe, with algorithms, the three cases of deleting a node from a singly linked list: at the beginning, at the end, and at a given position. What special case must be handled when the list has only one node?

6. What is a circular linked list? Explain how it differs from a singly linked list and describe at least one real-world application where a circular linked list is preferred.

7. Write an iterative algorithm to reverse a singly linked list, explaining the role of the prev, curr and next pointers. Derive its time and space complexity.

8. Explain the algorithm for merging two sorted singly linked lists into a single sorted linked list. Illustrate with a suitable example.

9. Explain how a stack can be implemented using a singly linked list. Write algorithms for the Push and Pop operations and state their time complexity.

10. Explain how a queue can be implemented using a singly linked list. Why is it necessary to maintain a separate rear pointer, and what would go wrong if it were not maintained?