Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

SINGLY LINKED LIST

Abstract

Singly linked list overview with its advantages and disadvantages. Types of singly linked list and pseudo code for various operations

Vandana M Ladwani
vandanamd@pes.edu

PES UNIVERSITY

Contents

Singly Linked list

Overview

Linked lists and arrays are similar in the sense that they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy access to elements through index. Once the array is set up, access to any element is convenient and fast.

Disadvantages of Array

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible, deletion just overwrites the content in the memory location that too at the expense of shifting elements

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Array allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked list allocates memory for each element separately and only when necessary. A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the data item/items to which it is linked in the list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the linked data item.

Advantages of linked lists

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (released) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deleting a data item from the given position.
4. Linked list is used as an important data structure for implementing many complex applications

Disadvantages of linked lists

1. It consumes more space because every node requires an additional pointer to store address of the link node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists

Basically we can put linked lists into the following four items:

1. Singly Linked List.
2. Doubly Linked List.
3. Circular Singly Linked List.
4. Circular Doubly Linked List.

A singly linked list is the one in which all nodes are linked together in sequential manner. Each node holds a single pointer which points to the next item. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by two links which helps in accessing both the successor node (link node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (link). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular doubly linked list is one, which has both the successor pointer and predecessor pointer and last node and first node are connected

Comparison between array and linked list:

ArrayList	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient	Insertions and Deletions are efficient
Elements are in contiguous memory locations	Elements are not in contiguous memory locations
May result in memory wastage if all the allocated space is not used	Since memory is allocated dynamically (as per requirement) there is no wastage of memory.
Sequential and random access is faster	Sequential and random access is slow

Singly Linked List

A linked list allocates space for each element separately in a block of memory called “node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "link" field which is a pointer used to link to the link node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). Head/start is a pointer to the first node in the linked list.

The basic operations in a singly linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing
- Reversing a list
- Concatenating two lists

Creating a node

- Get a node using malloc
- If memory is allocated successfully
    1. Set data part
    2. Set link part

Insertion of a Node

Memory is to be allocated for the new node. The new node will contain empty data field and empty link field. The data field of the new node is set to the value given by the user. The link field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.
- Inserting a node at the beginning:

Inserting a node at front

- Get the new node using createnode().
- new_node = createnode();
- If the list is empty then head = new_node.
- If the list is not empty, follow the steps given below:
  - new_node -> link = head;
  - head = new_node;

Inserting a node at the end

- Get the new node using createnode()
  new_node = createnode();
- If the list is empty then head = new_node.
- If the list is not empty follow the steps given below:
  - temp = head;
  - while(temp -> link != NULL)
    - temp = temp -> link;
    - temp -> link = new_node;

Inserting a node at intermediate position

- Get the new node using createnode().
  new_node = createnode();
- If the position is greater than length of linked list+1, specified position is invalid.
- Store the heading address (which is in head pointer) in temp and prev pointers. Then traverse the temp pointer unto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:
  If position is 1

Insert at front

If position=length of linked list+1

Insert at end

Else

if intermediate position

- prev -> link = new_node;
- new_node -> link = temp;

Deletion of a node

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.
- Deleting a node at the beginning:

Deleting a node from front of linked list

If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = head;
head = head -> link;
free(temp);
```

Deleting a node at the end

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = prev = head;
while(temp -> link != NULL)
{
 prev = temp;
 temp = temp -> link;
}
prev -> link = NULL;
free(temp);
```

Deleting a node at Intermediate position

- If list is empty then display 'Empty List' message
- If the list is not empty (has atleast two nodes), follow the steps given below.

if(pos > 1 && pos < nodectr)

- temp = prev = head;
- ctr = 1;
- while(ctr < pos)
    - prev = temp;
    - temp = temp -> link;
    - ctr++;
- prev -> link = temp -> link;
- free(temp);

Traversal and displaying a list (Left to Right)

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached.

Traversing a list involves repeating following steps till head becomes NULL

- Assign the address of head pointer to a temp pointer.
- Display the information from the data field of each node
- Advance head pointer