LINKED IMPLEMENTATION OF LIST

[1]Sonu Khandwal, [2]Mayank Saini

[1,2]Department of Information Technology,

Dronacharya College of Engineering, Gurgaon (India)

ABSTRACT

This paper address about the link list, its types and operations on link list like creation, traversing, insertion, deletion, searching a node in link list.

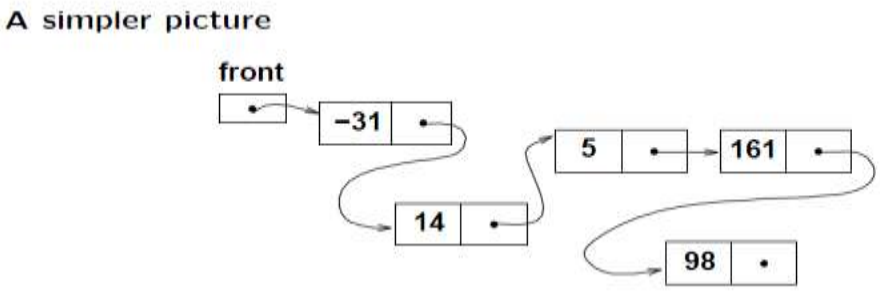
I INTRODUCTION

A link list is an ordered collection of finite homogeneous data elements called nodes where the linear order is maintained by means of links or pointers that is each pointer is divided into two parts: first contain the info of the element and the second part, called the link field contains the address of the next node in the list.The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

We will be discussing about two type of link list:

- Singly link list.
- Doubly link list

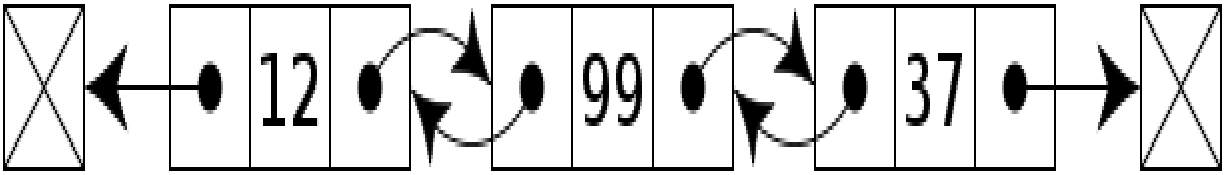
1.1 Singly Link List



1.2 Doubly linked list

Main article: Doubly linked list

In a **doubly linked list**, each node contains, besides the next-node link, a second link field pointing to the *previous* node in the sequence. The two links may be called **forward(s)** and **backwards**, or **next** and **prev** (**previous**).



A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

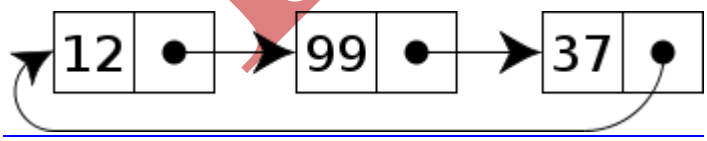
A technique known as XOR-linking allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

1.3 Multiply linked list

In a multiply linked list, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). While doubly linked lists can be seen as special cases of multiply linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.

1.4 Circular list

In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be 'circular' or 'circularly linked'; otherwise it is said to be 'open' or 'linear'.



A circular linked list

In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

1.5 Sentinel Nodes

Main article: Sentinel node

In some implementations, an extra sentinel or dummy node may be added before the first data record and/or after the last one. This convention simplifies and accelerates some list-handling algorithms, by ensuring that all links can be safely dereference and that every list (even one that contains no data elements) always has a "first" and "last" node.

1.6 Empty lists

An empty list is a list that contains no data records. This is usually the same as saying that it has zero nodes. If sentinel nodes are being used, the list is usually said to be empty when it has only sentinel nodes.

1.7 Hash linking

The link fields need not be physically part of the nodes. If the data records are stored in an array and referenced by their indices, the link field may be stored in a separate array with the same indices as the data records

1.8 .Linearly linked lists

Singly linked lists

Our node data structure will have two fields. We also keep a variable first Node which always points to the first node in the list, or is null for an empty list.

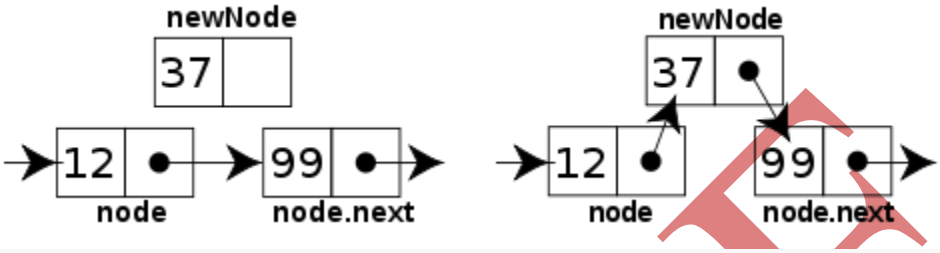
```
{
  data; // The data being stored in the node
  Node next // A reference to the next node, null for last node
}
record List
{
   Node firstNode // points to first node of list; null for empty list
}
```

Traversal of a singly linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```
node := list.firstNode
while node not null
   (do something with node.data)
   node := node.next
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done directly; instead, one must keep track of the previous node and insert a node after it

.



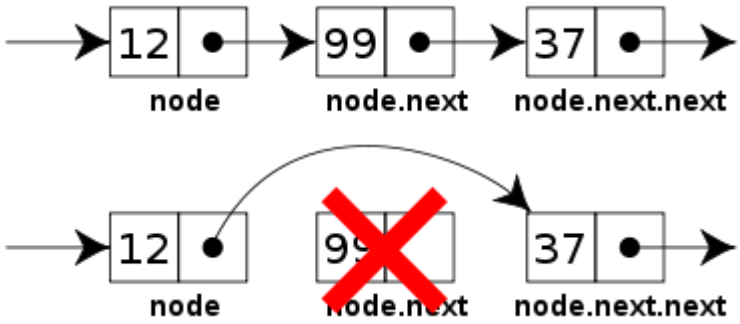
function

```
insertAfter(Node node, Node newNode) // insert newNode after node
    newNode.next := node.next
    node.next    := newNode
```

Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```
function insertBeginning(List list, Node newNode) // insert node before current first node
    newNode.next   := list.firstNode
    list.firstNode := newNode
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.




```
function removeAfter(Node node) // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
function removeBeginning(List list) // remove first node
    obsoleteNode := list.firstNode
```

```
list.firstNode := list.firstNode.next // point past deleted node
destroy obsoleteNode
```

1.9 Circularly linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using *null*. For lists with a front and a back (such as a queue) one stores a reference to the last node in the list. The next node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly linked lists can be either singly or doubly linked.

Both types of circularly linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing first Node and last Node, although if the list may be empty we need a special representation for the empty list, such as a last Node variable which points to some node in the list or is null if it's empty; we use such a last Node here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

II ALGORITHMS

Assuming that some Node is some node in a non-empty circular singly linked list, this code iterates through that list starting with some Node

```
function iterate(someNode)
 if someNode ≠ null
  node := someNode
  do
   do something with node.value
   node := node.next
  while node ≠ someNode
```

Notice that the test "**while** node ≠ someNode" must be at the end of the loop. If the test was moved to the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```
function insertAfter(Node node, Node newNode)
  if node = null
   newNode.next := newNode
```

```
else
  newNode.next := node.next
  node.next := newNode
```

REFERENCE

1) www.google.co.in

2) http://en.wikipedia.org/wiki/Linked_list