# Linked Lists ADT

By Omieno K.Kelvin

Department of Computer Science-MMUST (www.mmust.ac.ke)

## Linked List

Sequentially related information can be stored in non-sequential storage locations

**List Nodes** are the containers that hold the information; **Links** are stored with the information to determine the location of the next item; Links are usually array subscripts or pointers; The position of the first item is stored separately

### Where are the Nodes

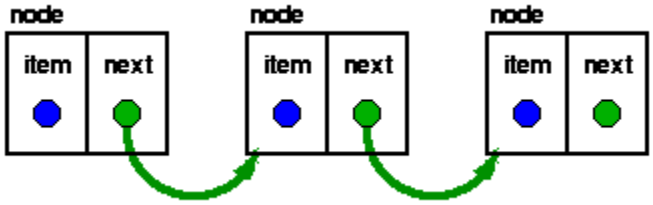
- Stored in an Array
- Array subscripts represent the location of a node
- Links are simply int's (subscript values)
- Dynamically Allocated (from heap)
- Node locations are actual addresses
- Links are pointers to nodes (Node *)

### Linked Lists in an Array

An array of ***Node's*** (Node store [MAX];) provides storage for the linked list. class ***Node*** contains members ***data*** and ***next*** ; In this example, the first node in the list is at 3

The linked list is a very flexible **dynamic data structure**: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.




Each node of the list has two elements

1. the item being stored in the list *and*
2. a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

### ***Handle for the list***

The variable (or handle) which represents the list is simply a pointer to the node at the *head* of the list.

### ***Adding to a list***

The simplest strategy for adding an item to a list is to:

a. allocate space for a new node,
b. copy the item into it,
c. make the new node's next pointer point to the current head of the list *and*
d. make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list.

An alternative is to create a structure for the list which contains both head and tail pointers:

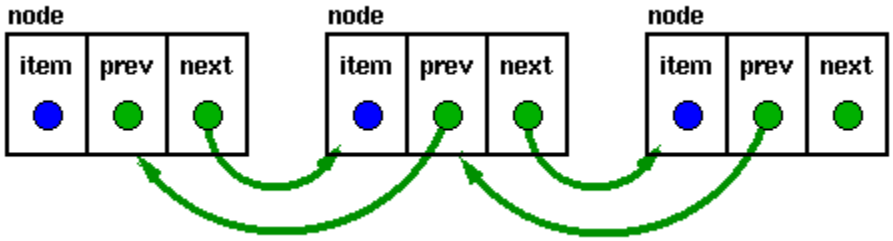
```
struct fifo_list {
        struct node *head;
        struct node *tail;
        };
```

## List variants

### a) Circularly Linked Lists

By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in struct t_node in our implementation), points to the current "tail" of the list, then the "head" is found trivially via tail->next, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.

## b) Doubly Linked Lists




Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
    void *item;
    struct t_node *previous;
    struct t_node *next;
    } node;
```

## Lists in arrays

Although this might seem pointless (Why impose a structure which has the overhead of the "next" pointers on an array?), this is just what memory allocators do to manage available space.

Memory is just an array of words. After a series of memory allocations and de-allocations, there are blocks of free memory scattered throughout the available heap space. In order to be able to re-use this memory, memory allocators will usually link freed blocks together in a *free list* by writing pointers to the next free block in the block itself. An external free list pointer points to the first block in the free list. When a new block of memory is requested, the allocator will generally scan the free list looking for a freed block of suitable size and delete it from the free list (re-linking the free list around the deleted block). Many variations of memory allocators have been proposed: refer to a text on operating systems or implementation of functional languages for more details. The entry in the index under *garbage collection* will probably lead to a discussion of this topic.

## Key terms

**Dynamic data structures**

> **Structures which grow or shrink as the data they hold changes. Lists, stacks and trees are all dynamic structures.**

## More intrusive notes

- The Ordered List ADT
    - An *ordered list* object is an ordered sequence of zero or more elements of some type. If the all the elements are of the same type, it is a *homogeneous* ordered list, otherwise it is a *heterogeneous* ordered list.
    - Operations include
        - Checking if the list is empty
        - Finding the length of the list
        - Storing a new element at one end or a specified position in the list
        - Retrieving an element from one end or a specified position in the list
        - Retrieving elements of the list from one end to the other (iteration)
        - Removing an element at one end or a specified position in the list
        - Comparing elements
        - Output the elements in order
    - The array ADT can be used to implement a homogeneous ordered list ADT.
        - Complications in the implementation due to the static nature of arrays.
        - Useful in some special cases, e.g., polynomials.
- Singly Linked Lists
    - Motivation (vs Arrays)
        - Adding new data
        - Deleting data
        - Storage efficiency
    - Simple representation
        - Private data and link - cannot access data from pointer to node
        - Public data, or public access functions - violates encapsulation principle
            - Any function can access, not only linked list manipulation functions
            - Only linked list manipulation functions must manipulate the data
        - Data class inside linked list class - cannot add more data to linked list class without adding more to every node.
        - Nested classes - does not provide reusability of data class

        - Separate list class and data class, with linked list manipulation functions as friends
    - Reusable representation
- Circular Linked Lists
    - Representation
    - A dummy head node
- Doubly Linked Lists
    - Motivation
        - Can only move one way
        - Hard to delete a known node
        - Hard to insert before a known node
    - Representation
    - Ease of previously difficult operations
    - A dummy head node

## Exam Style Questions

- Define the ordered list ADT.
- Give a C++ header file showing the classes for a decent linked list implementation. You need not give the function declarations, but you must declare the data members.
- Draw a picture of a {linked list,circular linked list,doubly linked list} with nodes containing the integer values 1, 16, 27, 92. Do not use any dummy nodes.
- What is the difference between an internal and an external iterator?
- What is the space complexity of a {linked list,circular linked list, doubly linked list} for storing N fixed size items of data?
- What is the big-O time complexity of {traversing,inserting a node at the front,inserting a node at the end} a {linked list, circular linked list,doubly linked list}? Explain your answer.