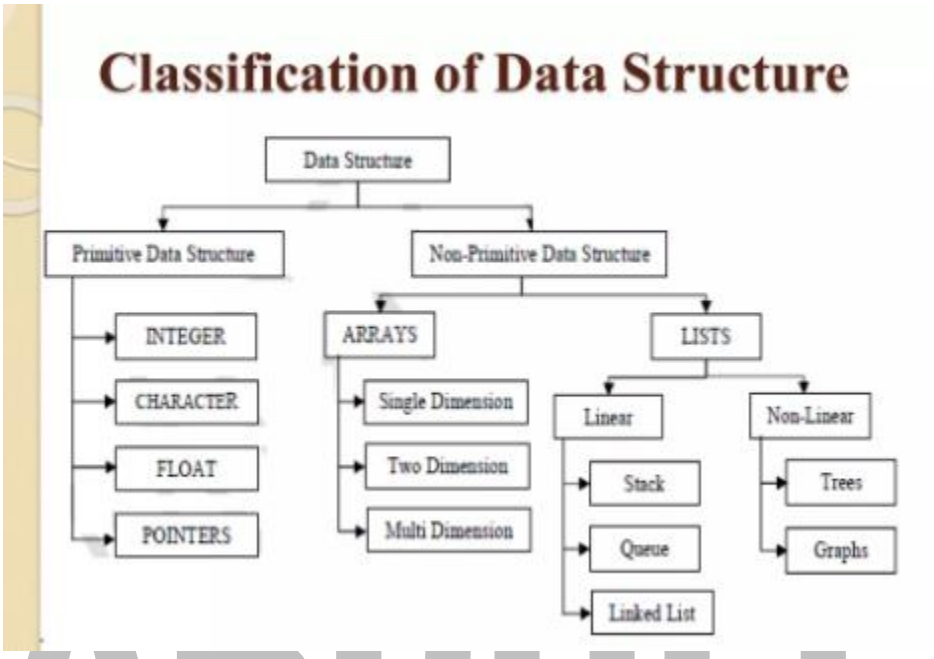
When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as **"Node"**.



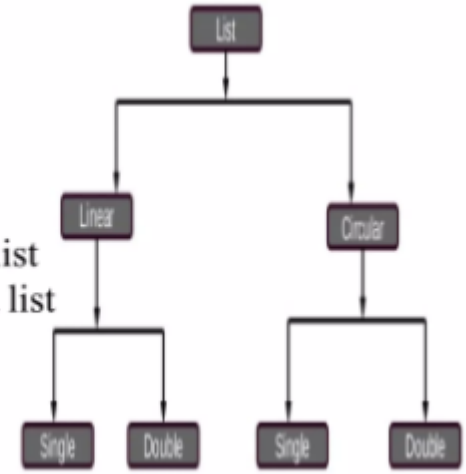
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items called ***nodes***.
- Lists are the most commonly used non-primitive data structures.
- Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - o The second part contains the memory address of the next node in the list. Also called Link part.

Lists

Types of linked lists:

- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list



A ***linked list***, in its simplest form, is a collection of ***nodes*** that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called *next*, to the next node of the list. In addition, each node stores its associated element. (See Figure 3.9.)

Figure 3.9: Example of a singly linked list of airport codes. The *next* pointers are shown as arrows. The null pointer is denoted by ∅.

The *next* pointer inside a node is a ***link*** or ***pointer*** to the next node of the list. Moving from one node to another by following a *next* reference is known as ***link hopping*** or ***pointer hopping***. The first and last nodes of a linked list are called the ***head*** and ***tail*** of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the tail. We can identify the tail as the node having a null *next* reference. The structure is called a ***singly linked list*** because each node stores a single link.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of *next* links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

Linked List

"Single linked list is a sequence of elements in which every element has link to its next element in the sequence."

In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

Single linked list

A singly linked list contains two fields in each node - an information field and the linked field.

- The ***information*** field contains the data of that node.
- The ***link*** field contains the memory address of the next node.

There is only one link field in each node, the linked list is called singly linked list.

It is noteworthy that we cannot as easily delete the last node of a singly linked list, even if we had a pointer to it. In order to delete a node, we need to update the *next* link of the node immediately ***preceding*** the deleted node. Locating this node involves traversing the entire list and could take a long time.

Doubly Linked Lists

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the ***doubly linked*** list. In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list, respectively. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

In computer science, a **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

Header and Trailer Sentinels

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a ***header*** node just before the head of the list, and a ***trailer*** node just after the tail of the list. These "dummy" or ***sentinel*** nodes do not store any elements. They provide quick access to the first and last nodes of the list. In particular, the header's *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.

Advantages of DLL over the singly linked list:

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

Disadvantages of DLL over the singly linked list:

- All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

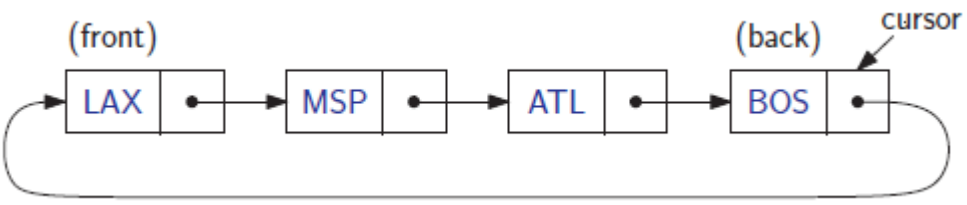
Figure 3.12: A doubly linked list with sentinels, *header* and *trailer*, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null *prev* pointer for the *header* nor do we show the null *next* pointer for the *trailer*.

3.4.1 Circularly Linked Lists

A ***circularly linked list*** has the same kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle. If we traverse the nodes of a circularly linked list from any node by following **next** pointers, we eventually visit all the nodes and cycle back to the node from which we started.

Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the ***cursor***. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.

There are two positions of particular interest in a circular list. The first is the element that is referenced by the cursor, which is called the ***back***, and the element immediately following this in the circular order, which is called the ***front***.



A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

We define the following functions for a circularly linked list:

front(): Return the element referenced by the cursor; an error results if the list is empty.

back(): Return the element immediately after the cursor; an error results if the list is empty.

advance(): Advance the cursor to the next node in the list.

add(e): Insert a new node with element e immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.

remove(): Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.