Lecture 13.1

Topics
1. Data Type – Introduction
2. Linear List ADT – Introduction
3. Linear List – Linked List

---

1. Data Type – Introduction

Abstract Data Type

In general, Abstract Data Type (ADT) is the type for some "composite/combined values/objects" whose behavior is defined by a set of operations.

For example, a linear list may be specified as an ADT in which one can provide a specification for data as well as operations (to perform on the given data). The ADT specification is independent of any representation and programming language that may be used to implement the specification.

All representations of the indicated ADT must satisfy the structural specification, and this specification provides a mean to validate the representation. Furthermore, all representations can be used interchangeably in applications with a given data type.

The following specification refers to a linear list.

```
ADT LinearList {
  data
    Ordered collections of zero or more elements (finite number)
  operations
    isEmpty()      : return true if empty, false otherwise
    size()          : return the size of the list
    get(index)     : return the index-th element
    index(x)       : return the index of the first occurrence of x
    remove(index)  : remove and return the index-th element,
                   : elements with higher index have their index
                   : reduced by 1
    add(index, x)  : add x to the structure at location index-th
    output()       : output the list elements
}
```

The above interface refers to a linear list where elements will be stored (ordered) in some fashion. However, there is no "concrete" or actual information for the implementation just yet, but one must form or organize data points before using the linear list as data structure (to store these data points).

2. Linear List ADT – Introduction

One of the most basic and simple data structures is the linear list, or just list. Its primary purpose is to represent and to store data on disk or in memory. Element of a list can be of any arbitrary type. It may be integer, floating-point, complex, or user's defined type, but all elements must be of the same type.

In general, the list **a(i)** will have the list elements of $a_0, a_1, a_2, a_3, \ldots, a_{n-1}$.

- The size of the list is **n**, where a null list is a list with `n = 0`.

- For any non-null list, a_{i+1} follows a_i and a_{i-1} precedes a_i. The first element is a_0 and the last element is a_{n-1}. The position of a_i in the list is i.

To complete the definition for a "**list Data Abstract Type**" (list ADT) (besides the data points), there is a set of operations that one can use to perform on this list ADT. The user defines this set of operations and use them to work with data points belonging to the list.

Some of the popular operations are given as follows,

`printList()`	To print all elements of a list.
`element(n)`	To return current element at location **`n`**.
`insert(value, i)`	To insert **`value`** to location **`i`** in the list.
`remove(i)`	To remove the element at location **`i`** in the list.
`current()`	To return the current element.
`next()`	To return the next element.
`previous()`	To return the previous element.

There are many representations of lists that based on the structure of their memory storage such as contiguous and non-contiguous memory storages.

Let's look at them next.

2.1 Contiguous Memory Storage – Array Based Representation

Arrays can be used to represent the list. It can either be static array or dynamic array. From this selection, the operations are thus provided (i.e., either built-in such as array of `int`'s or defined for other derived types) and performed accordingly.

2.2 Non-Contiguous Memory Storage – Linked Representation

Linked base representation provides alternatives for implementing the lists. In particular, linked list ADT will be the implementation for most of the general lists. There are several variations of linked lists such as singly-linear linked list, doubly linked list, circular linked list, etc.

Let's work with (singly-linear) linked list next.

3. Linear List – Linked List

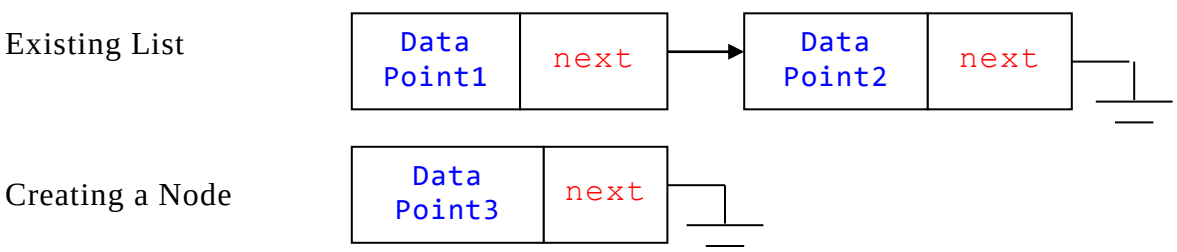
Linked list ADT is one of the choices for implementing the lists with **non-contiguous memory storage**.

There are two basic operations that must be provided to any list – **insertion** and **removal**.

There are definition and implementation variations given to these operations and other utilities to help managing the list.

In this lecture, let us look at two specific insertion and removal operations given below.

```
insert(Node *aNodePtr, LinkedList aList);
removeFirst(LinkedList aList);
```



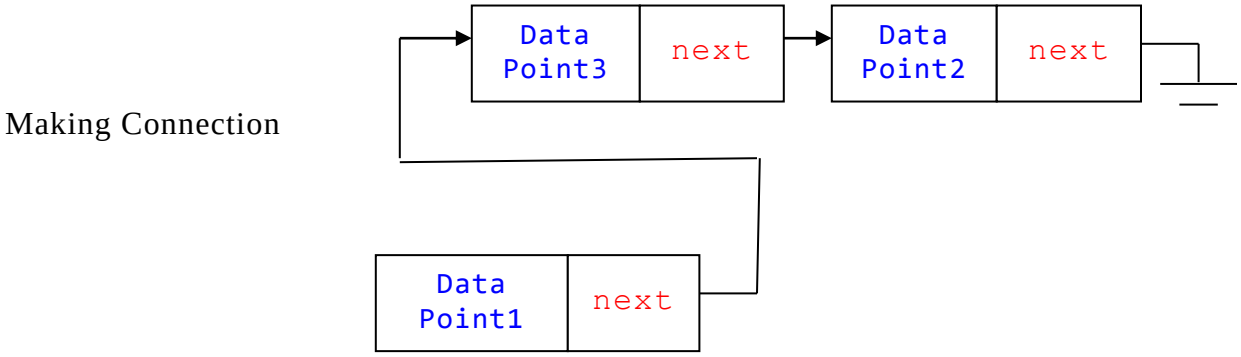


Figure 1 Steps in inserting a node to an existing list

The `removeFirst()` function will remove the first node in a given list.

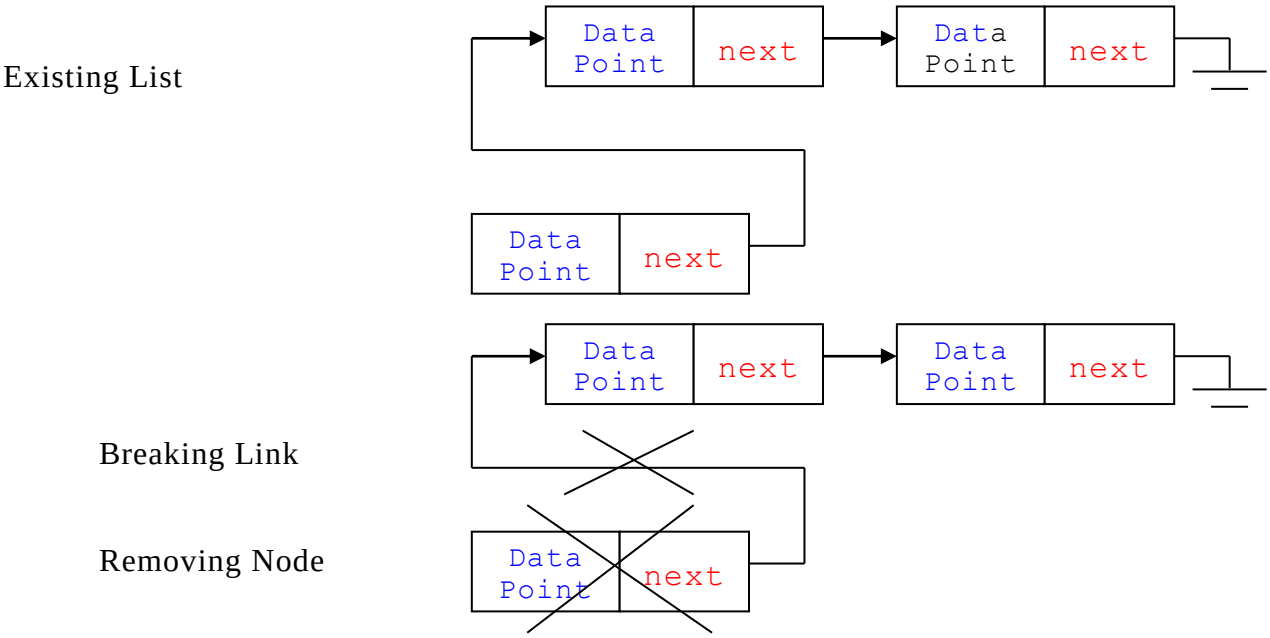


Figure 2 Steps in removing the first node

3.1 Linked List ADT

A linked list consists of a number of structures (objects of the same data types) called **nodes**. Each node has a pointer/reference that points to the next node in the list. This creates a chain where the first node points to the second node, the second node points to the third node, and so on.

The node is mostly dynamic and will hold a single "***data point***". Note that the interpretation of "data point" should not be taken just a one single value of an integer or floating-point; instead, it can be an object of any type which may provide the insight of a data representation.

A single node is depicted in **Figure 3** below.

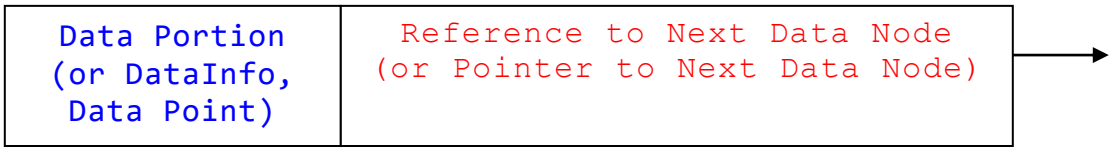


Figure 3 A node in a Linked List

The pointer that points to the next node is called the ***next*** pointer. The last node's `next` pointer points to a **`NULL`** (i.e., `0`). Most of the practical linked list will also have a so-called **head** node (also called list head or dummy node).

The head node may be assigned to a_0. In such cases, the head node has no data members but only holds the address of the next node a_1 (first node) in the list.

3.2 Linked List Declaration

A general linked list is declared as follows,

```
struct ListNode {
  SomeType value;
  struct ListNode* next;
};
```

To use the list, a head will also need to be declared as

```
struct ListNode* head;
```

One can also **`typedef`** the `ListNode*` so that the `ListNode` structure can be declared as follows:

```
typedef struct ListNode* ListNodePtr;

struct ListNode {
  SomeType value;
  ListNodePtr next;
};

ListNodePtr head;
```

Before using the **head** in any linked list operation, one must be aware of the value of **head** (i.e., what is being stored in head).

3.3 Linked List Operations

As with any general list, a linked list will have the following basic operations:

```
append(SomeType);       // Appending a node to a list
append(SomeType obj, ListNodePtr myList);

insert(SomeType);       // Inserting a node into a list
insert(SomeType obj, ListNodePtr myList, ListNode* location);

remove(SomeType);       // Deleting/Removing a node from a list
remove(SomeType obj, ListNodePtr myList);

displayList(void);            // Displaying the list
displayList(ListNodePtr myList);
```

Several utilities may also help to characterize the list as below.

```
int isEmpty(ListNode*);
int isLast(ListNode*);
```

3.4 Notation Issues

One may want to **`typedef`** several types to get the **type names** that have more meaningful interpretation and better presentation of coding. For examples, in the second version of the `insert()` operation, `myList` is the actual list of interest, and this is of type `ListNode`.

In this case, `typedef` can be used as follows,

```
typedef struct ListNode* ListNodePtr;

struct ListNode {
  SomeType value;
  ListNodePtr next;
};

typedef struct ListNodePtr LinkedList;
typedef struct ListNodePtr Location;
```

Using the above `typedef`'s, one can rewrite the above declarations as,

```
append(SomeType);       // Appending a node to a list
append(SomeType obj, LinkedList myList);

insert(SomeType);       // Inserting a node into a list
insert(SomeType obj, LinkedList myList, Location location);

remove(SomeType);       // Deleting/Removing a node from a list
remove(SomeType obj, LinkedList myList);

displayList(void);            // Displaying the list
displayList(LinkedList myList);
```

and,

```
int isEmpty(LinkList);
int isLast(LinkList);
```

Figure 4 depicts a linked list where the last node is grounded and the first node is referred to by the head.

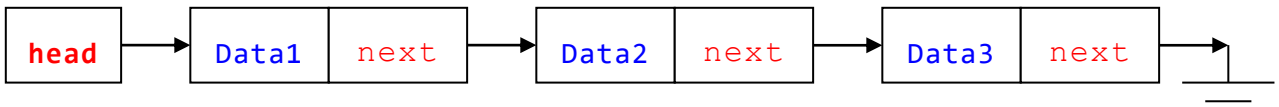


Figure 4 Singly Linked List

3.4 Linked List Basic Operations

There are two basic operations that must be provided to any list – **insertion** and **removal**. There are several variants of these operations.

We will look at two specific versions of these operations indicated below,

```
insertFirst(NodePointer* aNodeAddress, LinkedList aListAddress);
removeFirst(LinkedList aListAddress);
```