# Lecture 18.1

Topics
1. Linear List – Linked List

---

## 1. Linear List – Linked List

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);
```

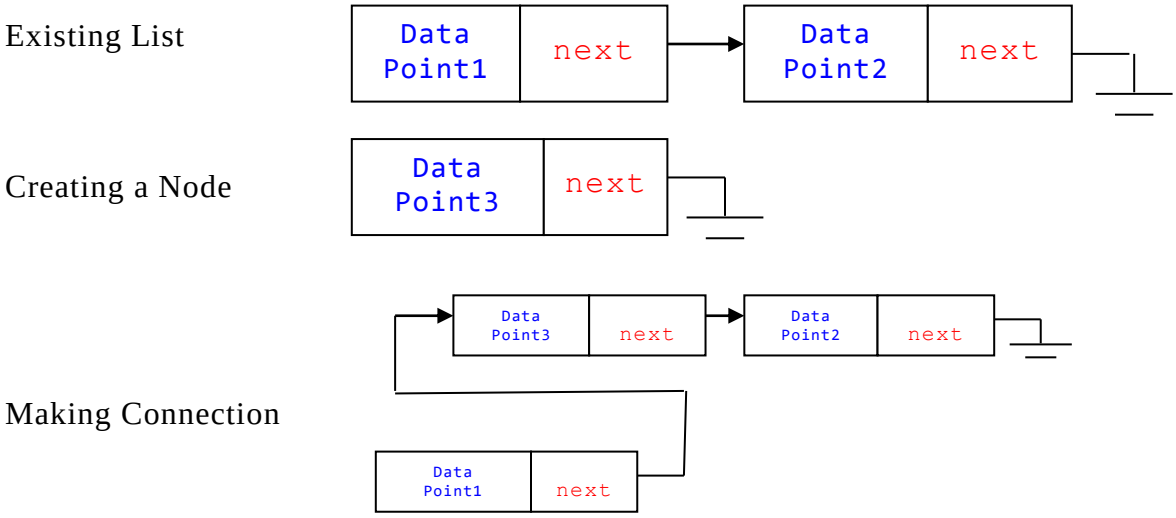


**Figure 1** Steps in inserting a node to an existing list

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

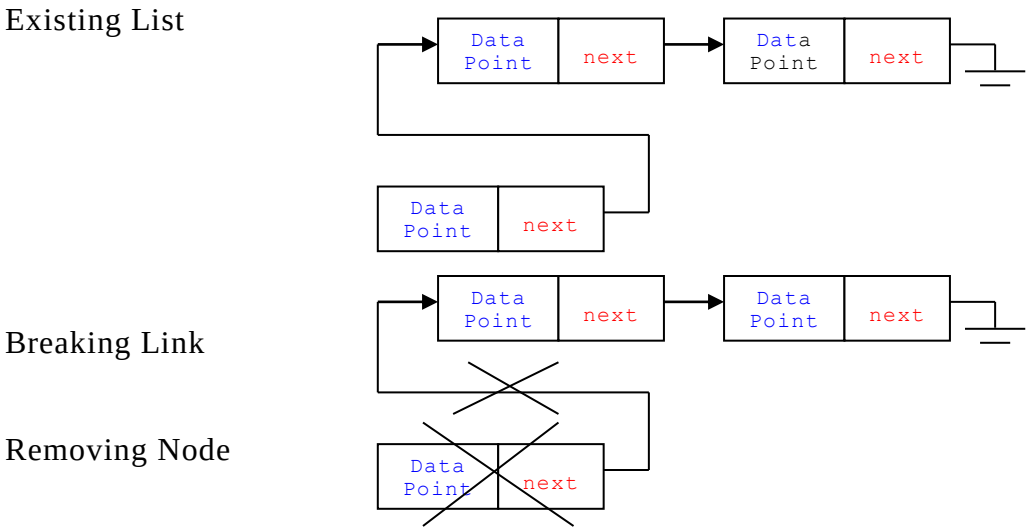


**Figure 2** Steps in removing the first node

Discussion will be given in class.

### 1.1 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 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.

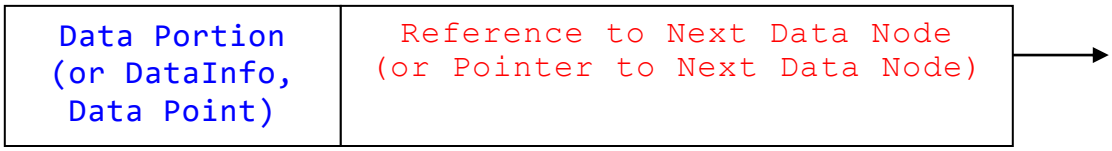


**Figure 3** A node in a Linked List

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) or just a pointer.

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.

### 1.2 Linked List Declaration

A general linked list is declared as follows,

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

Any node of this `ListNode` structure contains a `next` pointer to an object of the same type (which is another node) as the declaring structure. This is known as a **self-referential** data structure.

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

```
struct ListNode* head;

struct ListNode* myList;
```

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;

ListNodePtr myList;
```

Before using the **head/myList** in any linked list operation, one must be aware of the value of **head/myList** (i.e., what is being stored in head), which is the address of the first node. By checking the address first, this will avoid any problem when applying operations to an empty list.

### 1.3 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*);
```

### 1.4 Notation Issues

One may want to **`typedef`** to get the **type names** that have more meaningful interpretation and better presentation of coding.

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 ListAddr;
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.

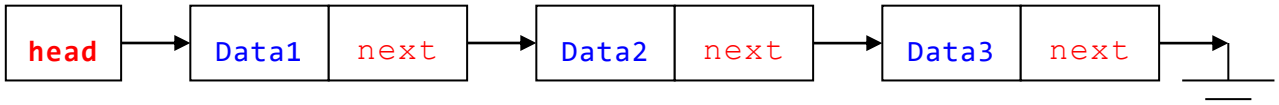


**Figure 4** Singly Linked List

### 1.4 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,

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