

What is a Data structure?

Data structure is one of the most **fundamentals** subject in Computer Science & in-depth understanding of this topic is very important especially when you are into **development/programming** domain where you **build efficient software systems & applications.**

Definition-

In computer science, a data structure is a **data organization, management and storage** format that enables **efficient access and modification.**

In Simple words-

Data Structure is a **way** in which data is stored on a computer.

Why do we need Data structures?

- Each Data Structure allows data to be stored in specific manner.
- Data Structure allow efficient data search and retrieval.
- Specific data structures are decided to work for specific problems.
- It allows to manage large amount of data such as large database and indexing services such as hash table.



ID	FirstName	Surname	Age
1	John	Jones	35
2	Tracey	Smith	25
3	Anno	McNeill	30
4	Andrew	Francis	37
5	Gillian	Carpenter	32
6	Karen	Rogers	22
7	Amy	Sanders	42
8	Kevin	White	18
9	Charlie	Anderson	40
10	Mary	Brown	26
11	Andrew	Smith	32
12	James	Francis	28
13	Karen	Jones	30
14	Edward	Kent	32
15	Jenny	Smith	26
16	Angela	Jones	41

Algorithms

What is an Algorithm ?

Dictionary Definition : A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Formal Definition : An algorithm is a finite set of instructions that are carried in a specific order to perform specific task.

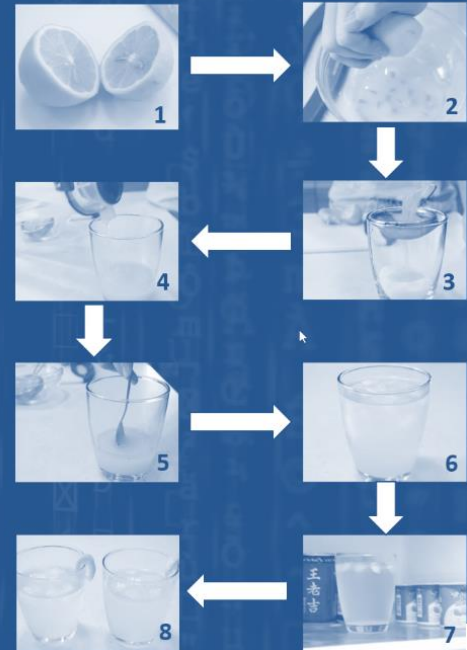
Algorithms typically have the following characteristics –

- **Inputs :** 0 or more input values.
- **Outputs :** 1 or more than 1 output.
- **Unambiguity :** clear and simple instructions.
- **Finiteness :** Limited number of instructions.
- **Effectiveness :** Each instruction has an impact on the overall process.

Real World example of an Algorithm –

Algorithm(aka process) to make a lemonade –

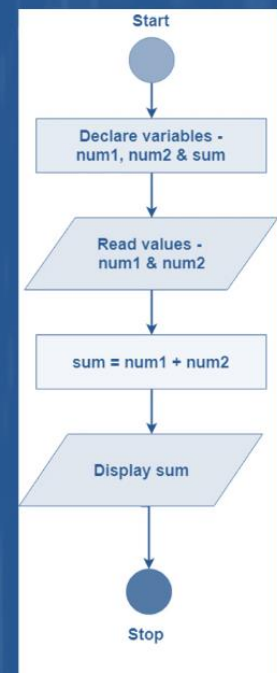
1. Cut your lemon in half.
2. Squeeze all the juice out of it that you can.
3. Pour your juice into a container with 1/4 cup (2 oz) sugar.
4. Add a very small amount of water to your container.
5. Stir your solution until sugar dissolves.
6. Fill up container with water and add ice.
7. Put your lemonade in the fridge for five minutes.
8. Serve and enjoy!



Example of an Algorithm in Programming –

Write an algorithm to add two numbers entered by user. –

1. Step 1: Start
2. Step 2: Declare variables num1, num2 and sum.
3. Step 3: Read values num1 and num2.
4. Step 4: Add num1 and num2 and assign the result to sum.($sum \leftarrow num1 + num2$)
5. Step 5: Display sum
6. Step 6: Stop



program = Algorithm + Data Structure


What is Abstract Data Type?

Definition: ADTs are entities that are definitions of data and operations but do not have implementation details.

Two ways of looking at Data Structures-

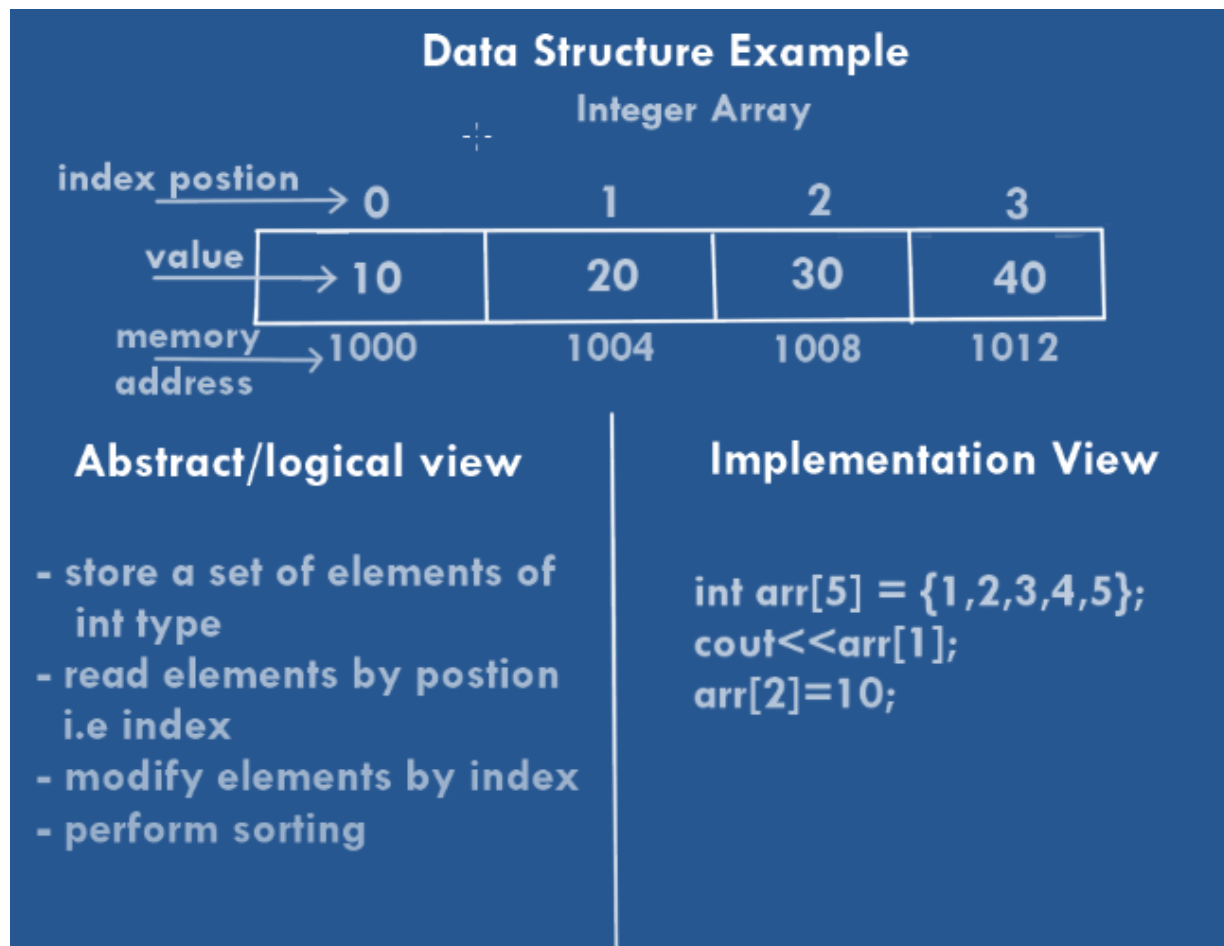
- Mathematical/Logical/Abstract Models/Views
- Implementation

Real world Example



←
smartphone

<h4>Abstract/logical view</h4> <ul style="list-style-type: none"> - 4 GB RAM - Snapdragon 2.2GHz processor - 5.5 inch LCD screen - Dual Camera - Android 8.0 - call() - text() - photo() - video() 	<h4>Implementation view</h4> <pre>class Smartphone{ private: int ramSize; string processorName; float screenSize; int cameraCount; string androidVersion; public: void call(); void text(); void photo(); void video(); };</pre>
---	--



Representation of algorithm can written By:-

In natural language (English) / pseudo-code / diagrams (Flow chart) / etc.

Pseudo- code:-

A mixture of natural language and high – level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm. **Pseudo- code** is more structured than usual language but less formal than a programming language.

Ex.:- Algorithm to find the maximum number in array

input: An array with n integers

output: The Maximum element in A

```
currentMax ← A[0]
for i ← 1 to n-1 do
if currentMax < A[i] then currentMax ← A[i]
return currentMax
```

Ex: An algorithm to find sum n numbers for N range

- 1- Start
- 2- Read N
- 3- Sum =0
- 4- For I= 1 to N
 - 4.1 sum = sum + I
 - 4.1 next I
- 5- print Sum
- 6- End

What Makes a Good Algorithm?

Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?

- Faster
- Less space
- Easier to code
- Easier to maintain

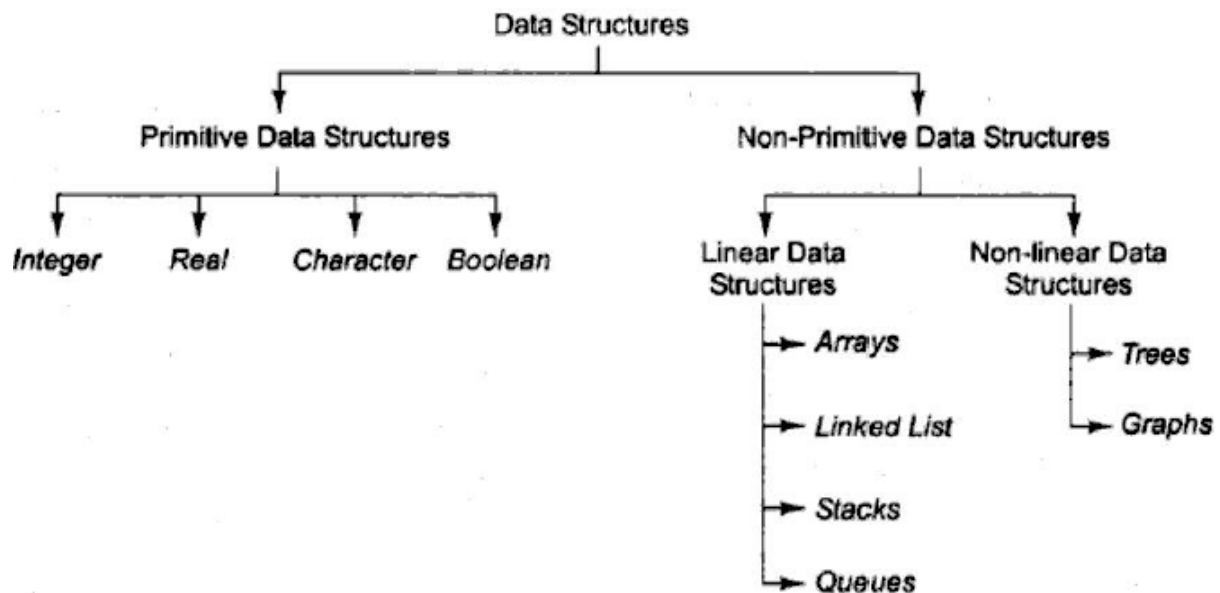


Fig.1 Classifications of data structures

Classification of data structure

Data structures are broadly divided into two:

1. Primitive data structures: These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc. These primitive data structures are the basis for the discussion of more sophisticated (non-primitive) data structures.
2. Non-primitive data structures: It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.

How to choose the suitable data structure:-

For each set of data, there are different methods to organize these data in a particular data structure.

To choose the suitable data structure, we must use the following criteria:-

- 1- Data size and the required memory.
- 2- The dynamic nature of the data.
- 3- The required time to obtain any data element from the data structure.
- 4- The programming approach and the algorithm that will be used to manipulate these data.

Assignment -1-

- Write an algorithm for the following
 - a- count even & odd numbers in given range

LINKED LIST DATA STRUCTURE

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field. Following Fig 5:1 shows a typical example of node.

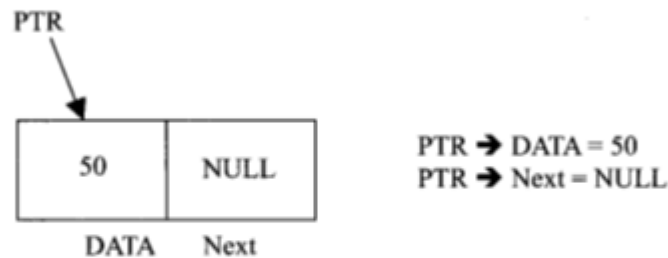


Fig. 5.1. Nodes.



Fig. 5.2. Linked List.

Fig.5.2.shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START = NULL if there is no list (*i.e.*; NULL list or empty list).



Fig. Linked List representation in memory.

Explanation:

Because each node of a linked list has two components, we need to declare each node as a class or struct. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodename
{
    int info;
    nodename *link;
};
```

The variable declaration is as follows:

```
nodename *head;
```

Linked List: Some Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 5-4.

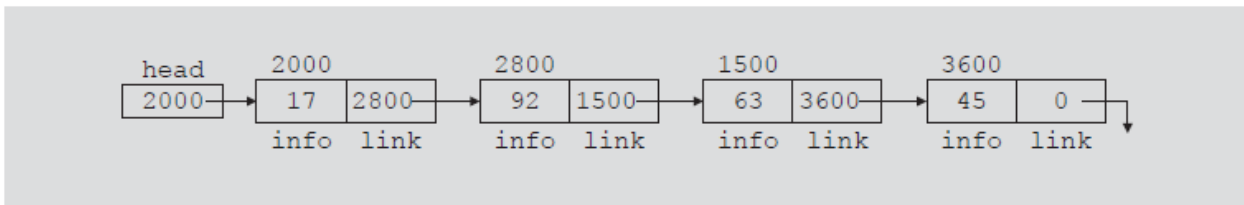


FIGURE 5-4 Linked list with four nodes

This linked list has four nodes. The address of the first node is stored in the pointer head.

Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int.

Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table Table 5-1 shows the values of head and some other nodes in the list shown in Figure 5-4.

TABLE 5-1 Values of head and some of the nodes of the linked list in Figure 5-4

	Value	Explanation
head	2000	
head->info	17	Because head is 2000 and the info of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because head->link is 2800 and the info of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

```
current = head;
```

copies the value of head into current. Now consider the following statement:

```
current = current->link;
```

This statement copies the value of current->link, which is 2800, into current.

Therefore, after this statement executes, current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 5-5.

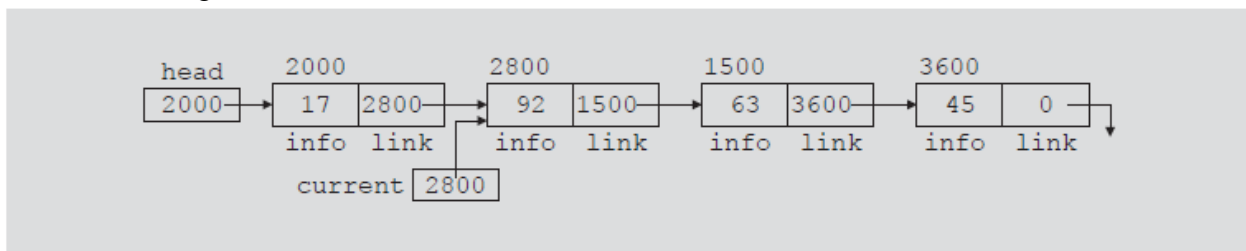
FIGURE 5-5 List after the statement `current = current->link;` executes

Table 5-2 shows the values of current, head, and some other nodes in Figure 5-5

TABLE 5-2 Values of current, head, and some of the nodes of the linked list in Figure 5-5

	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

TRAVERSING A LINKED LIST

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, display the elements of the list, and delete an item from the list.

These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer **head** to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to maintain the list). Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

```
current = head;
while (current != NULL)
{
    //Process current
    current = current->link;
}
```

For example, suppose that **head** points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;
while (current != NULL)
{
    cout << current->info << " ";
    current = current->link;
}
```

LINKED LIST ALGORITHMS

This section discusses the algorithms of linked list data structures. Consider the following definition of a node. (For simplicity, we assume that the info type is **int**.)

```
struct nodename
{
    int info;
    nodename *link;
};
```

We will use the following variable declaration:

```
nodename *head, *p, *q, *newNode;
```

ALGORITHM FOR INSERTING A NODE

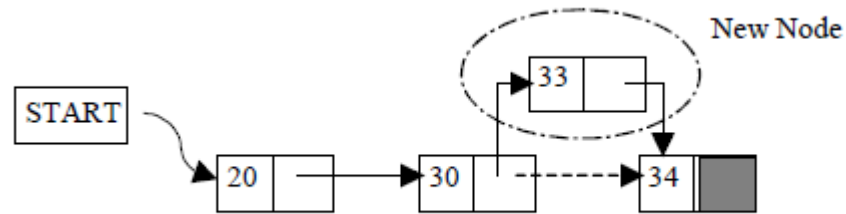


Fig. 5.14. Insertion of New Node

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

Insert a Node at the beginning

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. If (START equal to NULL)
 - (a) NewNode -> Link = NULL
5. Else
 - (a) NewNode -> Link = START
6. START = NewNode
7. Exit

Insert a Node at the end

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. NewNode -> Next = NULL
5. If (START equal to NULL)
 - (a) START = NewNode
6. Else
 - (a) TEMP = START
 - (b) While (TEMP -> Next not equal to NULL)
 - (i) TEMP = TEMP -> Next
7. TEMP -> Next = NewNode
8. Exit

Insert a Node at any specified position

1. Input DATA and POS to be inserted
2. initialize TEMP = START; and k = 0
3. Repeat the step 3 while(k is less than POS)

- (a) $TEMP = TEMP \rightarrow Next$
 - (b) If (TEMP is equal to NULL)
 - (i) Display “Node in the list less than the position”
 - (ii) Exit
 - (c) $k = k + 1$
4. Create a New Node
 5. $NewNode \rightarrow DATA = DATA$
 6. $NewNode \rightarrow Next = TEMP \rightarrow Next$
 7. $TEMP \rightarrow Next = NewNode$
 8. Exit

Consider the linked list shown in Figure 5-6.

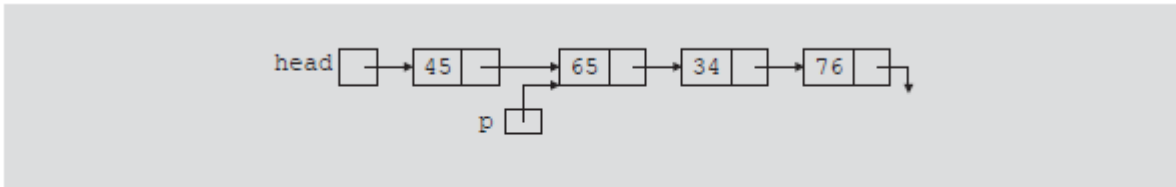


FIGURE 5-6 Linked list before item insertion

Suppose that **p** points to the node with **info 65**, and a new node with **info 50** is to be created and inserted after **p**. Consider the following statements:

```

newNode = new nodename;           //create newNode
newNode->info = 50;                //store 50 in the new node
newNode->link = p->link;
p->link = newNode;

```

Table 5-3 shows the effect of these statements.

TABLE 5-3 Inserting a node in a linked list

Statement	Effect
<code>newNode = new nodeType;</code>	
<code>newNode->info = 50;</code>	
<code>newNode->link = p->link;</code>	
<code>p->link = newNode;</code>	

Note that the sequence of statements to insert the node, that is,

```

newNode->link = p->link;
p->link = newNode;

```

is very important because to insert **newNode** in the list we use only one pointer, **p**, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link = newNode;
newNode->link = p->link;
```

Figure 5-7 shows the resulting list after these statements execute.

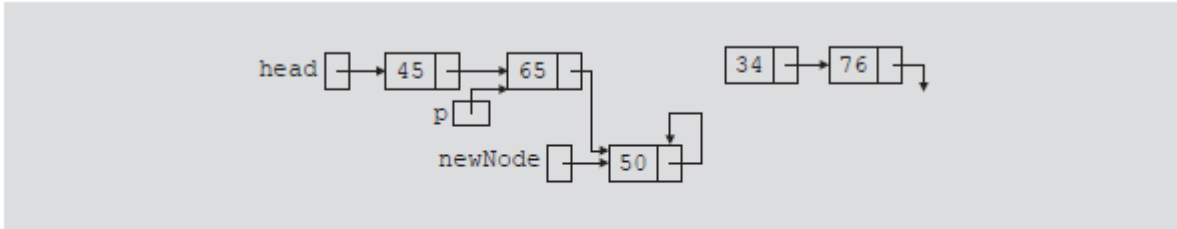


FIGURE 5-7 List after the execution of the statement `p->link = newNode;` followed by the execution of the statement `newNode->link = p->link;`

From Figure 5-7, it is clear that **newNode** points back to itself and the remainder of the list is lost. Using two pointers, we can simplify the insertion code somewhat. Suppose **q** points to the node with **info 34**. (See Figure 5-8.)

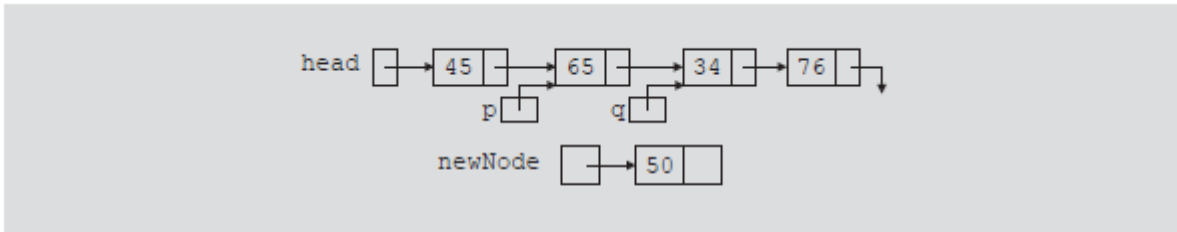


FIGURE 5-8 List with pointers **p** and **q**

The following statements insert **newNode** between **p** and **q**:

```
newNode->link = q;
p->link = newNode;
```

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

```
p->link = newNode;
newNode->link = q;
```

Table 5-4 shows the effect of these statements.

TABLE 5-4 Inserting a node in a linked list using two pointers

Statement	Effect
<code>p->link = newNode;</code>	
<code>newNode->link = q;</code>	

ALGORITHM FOR DELETING A NODE

Consider the linked list shown in Figure 5-9.

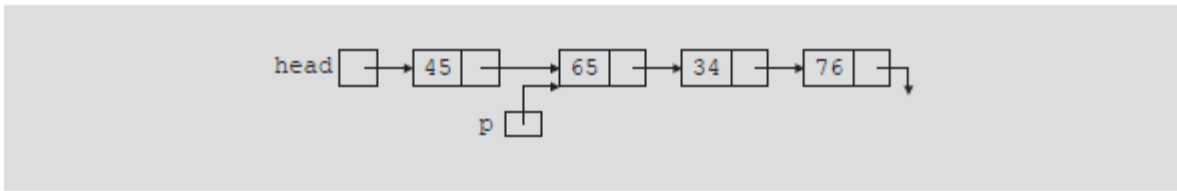


FIGURE 5-9 Node to be deleted is with info 34

Suppose that the node with **info 34** is to be deleted from the list. The following statement removes the node from the list:

```
p->link = p->link->link;
```

Figure 5-10 shows the resulting list after the preceding statement executes.

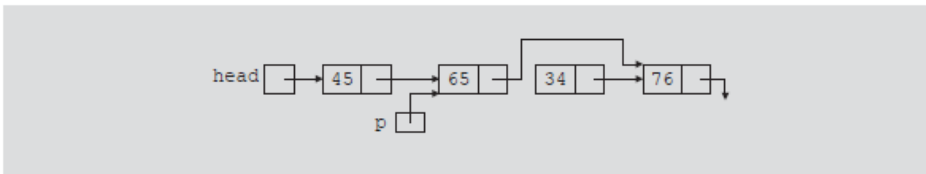


FIGURE 5-10 List after the statement `p->link = p->link->link;` executes

From Figure 5-10, it is clear that the node with **info 34** is removed from the list.

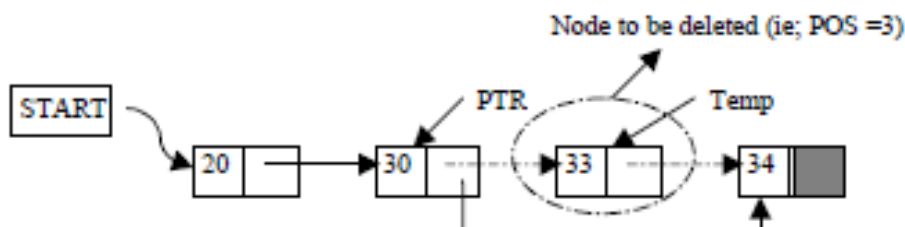
However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link;
p->link = q->link;
delete q;
```

Table 5-5 shows the effect of these statements.

TABLE 5-5 Deleting a node from a linked list

Statement	Effect
<code>q = p->link;</code>	
<code>p->link = q->link;</code>	
<code>delete q;</code>	



Deletion of a Node

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

1. Input the DATA to be deleted
2. if ((START -> DATA) is equal to DATA)
 - (a) TEMP = START
 - (b) START = START -> Next
 - (c) Set free the node TEMP, which is deleted
 - (d) Exit
3. HOLD = START
4. while ((HOLD -> Next -> Next) not equal to NULL)
 - (a) if ((HOLD -> NEXT -> DATA) equal to DATA)
 - (i) TEMP = HOLD -> Next
 - (ii) HOLD -> Next = TEMP -> Next
 - (iii) Set free the node TEMP, which is deleted
 - (iv) Exit
 - (b) HOLD = HOLD -> Next
5. if ((HOLD -> next -> DATA) == DATA)
 - (a) TEMP = HOLD -> Next
 - (b) Set free the node TEMP, which is deleted
 - (c) HOLD -> Next = NULL
 - (d) Exit
6. Disply "DATA not found"
7. Exit

ALGORITHM FOR SEARCHING A NODE

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
 - (a) Display "The data is found at POS"
 - (b) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
 - (a) Display "The data is not found in the list"
8. Exit

ALGORITHM FOR DISPLAY ALL NODES

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

1. If (START is equal to NULL)
 - (a) Display “The list is Empty”
 - (b) Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP == NULL)
4. Display “TEMP → DATA”
5. TEMP = TEMP → Next
6. Exit