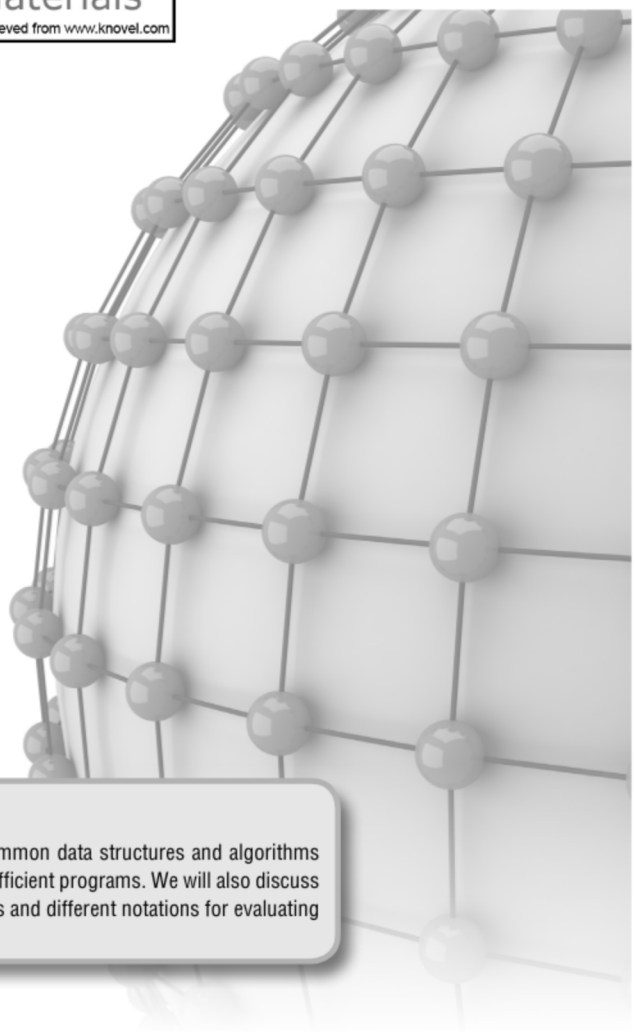
CHAPTER 2

Introduction to Data Structures and Algorithms



LEARNING OBJECTIVE

In this chapter, we are going to discuss common data structures and algorithms which serve as building blocks for creating efficient programs. We will also discuss different approaches to designing algorithms and different notations for evaluating the performance of algorithms.

2.1 BASIC TERMINOLOGY

We have already learnt the basics of programming in C in the previous chapter and know how to write, debug, and run simple programs in C language. Our aim has been to design good programs, where a good program is defined as a program that

- runs correctly
- is easy to read and understand
- is easy to debug *and*
- is easy to modify.

A program should undoubtedly give correct results, but along with that it should also run efficiently. A program is said to be efficient when it executes in minimum time and with minimum memory space. In order to write efficient programs we need to apply certain data management concepts.

The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Data structure is a crucial part of data management and in this book it will be our prime concern. A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

- Compiler design
- Statistical analysis package
- Numerical analysis
- Artificial intelligence
- Operating system
- DBMS
- Simulation
- Graphics

When you will study DBMS as a subject, you will realize that the major data structures used in the Network data model is graphs, Hierarchical data model is trees, and RDBMS is arrays.

Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently. Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design. This is because representing information is fundamental to computer science. The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

Be it any problem at hand, the application of an appropriate data structure provides the most efficient solution. A solution is said to be efficient if it solves the problem within the required resource constraints like the total space available to store the data and the time allowed to perform each subtask. And the best solution is the one that requires fewer resources than known alternatives. Moreover, the cost of a solution is the amount of resources it consumes. The cost of a solution is basically measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

Today computer programmers do not write programs just to solve a problem but to write an efficient program. For this, they first analyse the problem to determine the performance goals that must be achieved and then think of the most appropriate data structure for that job. However, program designers with a poor understanding of data structure concepts ignore this analysis step and apply a data structure with which they can work comfortably. The applied data structure may not be appropriate for the problem at hand and therefore may result in poor performance (like slow speed of operations).

Conversely, if a program meets its performance goals with a data structure that is simple to use, then it makes no sense to apply another complex data structure just to exhibit the programmer's skill. When selecting a data structure to solve a problem, the following steps must be performed.

1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centred view of the design process. In the approach, the first concern is the data and the operations that are to be performed on them. The second concern is the representation of the data, and the final concern is the implementation of that representation.

There are different types of data structures that the C language supports. While one type of data structure may permit adding of new data items only at the beginning, the other may allow it to be added at any position. While one data structure may allow accessing data items sequentially, the other may allow random access of data. So, selection of an appropriate data structure for the problem is a crucial decision and may have a major impact on the performance of the program.

2.1.1 Elementary Data Structure Organization

Data structures are building blocks of a program. A program built using improper data structures may not work as expected. So as a programmer it is mandatory to choose most appropriate data structures for a program.

The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.).

While a data item that does not have subordinate data items is categorized as an elementary item, the one that is composed of one or more subordinate data items is called a group item. For example, a student's name may be divided into three sub-items—first name, middle name, and last name—but his roll number would normally be treated as a single item.

A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth.

Moreover, each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a *primary key*, and the values K_1, K_2 ... in such field are called keys or key values. For example, in a student's record that contains roll number, name, address, course, and marks obtained, the field roll number is a primary key. Rest of the fields (name, address, course, and marks) cannot serve as primary keys, since two or more students may have the same name, or may have the same address (as they might be staying at the same place), or may be enrolled in the same course, or have obtained same marks.

This organization and hierarchy of data is taken further to form more complex types of data structures, which is discussed in Section 2.2.

2.2 CLASSIFICATION OF DATA STRUCTURES

Data structures are generally categorized into two classes: *primitive* and *non-primitive* data structures.

Primitive and Non-primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.
Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

Linear and Non-linear Structures

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

C supports a variety of data structures. We will now introduce all these data structures and they would be discussed in detail in subsequent chapters.

Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).

In C, arrays are declared using the following syntax:

```
type name[size];
```

For example,

```
int marks[10];
```

The above statement declares an array `marks` that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in `marks[0]`, second element in `marks[1]`, so on and so forth. Therefore, the last element, that is the 10th element, will be stored in `marks[9]`. In the memory, the array will be stored as shown in Fig. 2.1.

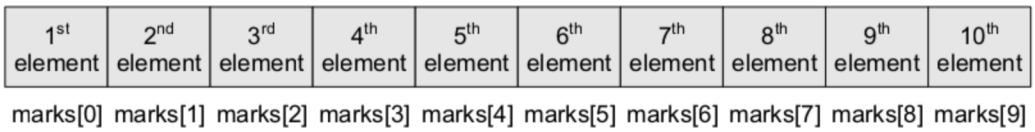


Figure 2.1 Memory representation of an array of 10 elements

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

However, these limitations can be solved by using linked lists. We will discuss more about arrays in Chapter 3.

Linked Lists

A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list. In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance.

In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node 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. Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available. Figure 2.2 shows a linked list of seven nodes.

Figure 2.2 Simple linked list

Note *Advantage*: Easier to insert or delete data elements
Disadvantage: Slow search operation and requires more memory space

Stacks

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the `top` of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

In the computer's memory, stacks can be implemented using arrays or linked lists. Figure 2.3 shows the array implementation of a stack. Every stack has a variable `top` associated with it. `top` is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted. There is another variable `MAX`, which is used to store the maximum number of elements that the stack can store.

If `top = NULL`, then it indicates that the stack is empty and if `top = MAX-1`, then the stack is full.

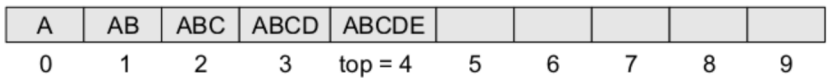


Figure 2.3 Array representation of a stack

In Fig. 2.3, `top = 4`, so insertions and deletions will be done at this position. Here, the stack can store a maximum of 10 elements where the indices range from 0–9. In the above stack, five more elements can still be stored.

A stack supports three basic operations: `push`, `pop`, and `peep`. The `push` operation adds an element to the top of the stack. The `pop` operation removes the element from the top of the stack. And the `peep` operation returns the value of the topmost element of the stack (without deleting it).

However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full.

Similarly, before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the `rear` and removed from the other end called the `front`. Like stacks, queues can be implemented by using either arrays or linked lists.

Every queue has `front` and `rear` variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in Fig. 2.4.

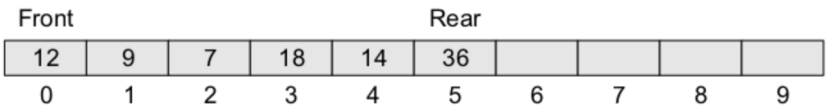


Figure 2.4 Array representation of a queue

Here, `front = 0` and `rear = 5`. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the `rear` would be incremented by 1 and the value would be stored at the position pointed by the `rear`. The queue, after the addition, would be as shown in Fig. 2.5.

Here, `front = 0` and `rear = 6`. Every time a new element is to be added, we will repeat the same procedure.

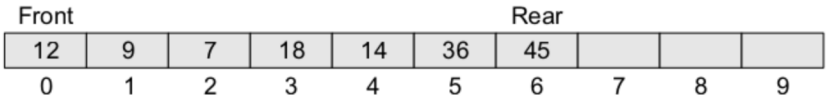


Figure 2.5 Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of `front` will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 2.6.

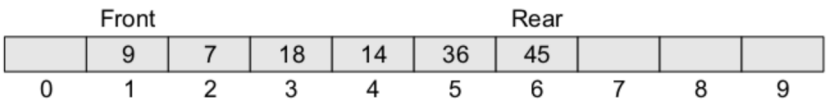


Figure 2.6 Queue after deletion of an element

However, before inserting an element in the queue, we must check for overflow conditions. An overflow occurs when we try to insert an element into a queue that is already full. A queue is full when `rear = MAX-1`, where `MAX` is the size of the queue, that is `MAX` specifies the maximum number of elements in the queue. Note that we have written `MAX-1` because the index starts from 0.

Similarly, before deleting an element from the queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If `front = NULL` and `rear = NULL`, then there is no element in the queue.

Trees

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree. The root element is the topmost node which is pointed by a 'root' pointer. If `root = NULL` then the tree is empty.

Figure 2.7 shows a binary tree, where `R` is the root node and T_1 and T_2 are the left and right sub-trees of `R`. If T_1 is non-empty, then T_1 is said to be the left successor of `R`. Likewise, if T_2 is non-empty, then it is called the right successor of `R`.

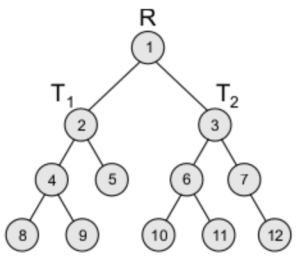


Figure 2.7 Binary tree

In Fig. 2.7, node 2 is the left child and node 3 is the right child of the root node 1. Note that the left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

Note *Advantage*: Provides quick search, insert, and delete operations
Disadvantage: Complicated deletion algorithm

Graphs

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure 2.8 shows a graph with five nodes.

A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

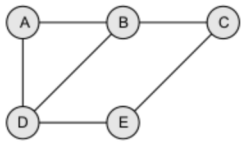


Figure 2.8 Graph

Note that unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as *neighbours*. For example, in Fig. 2.8, node A has two neighbours: B and D.

Note *Advantage*: Best models real-world situations
Disadvantage: Some algorithms are slow and very complex

2.3 OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

Many a time, two or more operations are applied simultaneously in a given situation. For example, if we want to delete the details of a student whose name is X, then we first have to search the list of students to find whether the record of X exists or not and if it exists then at which location, so that the details can be deleted from that particular location.

2.4 ABSTRACT DATA TYPE

An *abstract data type* (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.

To further understand the meaning of an abstract data type, we will break the term into 'data type' and 'abstract', and then discuss their meanings.

Data type Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include `int`, `char`, `float`, and `double`.

When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data. For example, an `int` variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators `+`, `-`, `*`, and `/`. In other words, the operations that can be performed on a data type are an inseparable part of its identity. Therefore, when we declare a variable of an abstract data type (e.g., stack or a queue), we also need to specify the operations that can be performed on it.

Abstract The word 'abstract' in the context of data structures means *considered apart from the detailed specifications or implementation*.

In C, an abstract data type can be a structure considered without regard to its implementation. It can be thought of as a 'description' of the data in the structure with a list of operations that can be performed on the data within that structure.

The end-user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

For example, when we use a stack or a queue, the user is concerned only with the type of data and the operations that can be performed on it. Therefore, the fundamentals of how the data is stored should be invisible to the user. They should not be concerned with how the methods work or what structures are being used to store the data. They should just know that to work with stacks, they have `push()` and `pop()` functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

Advantage of using ADTs

In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

2.5 ALGORITHMS

The typical definition of algorithm is 'a formally defined procedure for performing some calculation'. If a procedure is formally defined, then it can be implemented using a formal language,

and such a language is known as a *programming language*. In general terms, an algorithm provides a blueprint to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Algorithms are mainly used to achieve *software reuse*. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java.

An algorithm is basically a set of instructions that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

2.6 DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.

A complex algorithm is often divided into smaller units called modules. This process of dividing an algorithm into modules is called modularization. The key advantages of modularization are as follows:

- It makes the complex algorithm simpler to design and implement.
- Each module can be designed independently. While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

There are two main approaches to design an algorithm—top-down approach and bottom-up approach, as shown in Fig. 2.9.

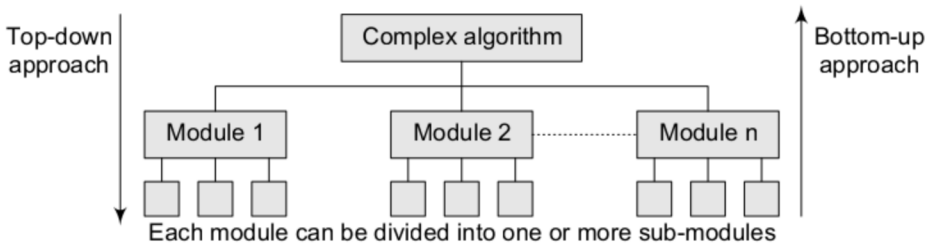


Figure 2.9 Different approaches of designing an algorithm

Top-down approach A top-down design approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved. Top-down design method is a form of stepwise refinement where we begin with the topmost module and incrementally add modules that it calls.

Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Bottom-up approach A bottom-up approach is just the reverse of top-down approach. In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules. The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.

Top-down vs bottom-up approach Whether the top-down strategy should be followed or a bottom-up is a question that can be answered depending on the application at hand.

While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.

Top-down approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.

Although the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module's boundaries as seen from the clients. But all this is difficult to be done in a strict bottom-up strategy. Some top-down activities need to be performed for this.

All in all, design of complex algorithms must not be constrained to proceed according to a fixed pattern but should be a blend of top-down and bottom-up approaches.

2.7 CONTROL STRUCTURES USED IN ALGORITHMS

An algorithm has a finite number of steps. Some steps may involve decision-making and repetition. Broadly speaking, an algorithm may employ one of the following control structures: (a) sequence, (b) decision, and (c) repetition.

Sequence

By sequence, we mean that each step of an algorithm is executed in a specified order. Let us write an algorithm to add two numbers. This algorithm performs the steps in a purely sequential order, as shown in Fig. 2.10.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Figure 2.10 Algorithm to add two numbers

Decision

Decision statements are used when the execution of a process depends on the outcome of some condition. For example, if `x = y`, then print `EQUAL`. So the general form of `IF` construct can be given as:

```
IF condition Then process
```

A condition in this context is any statement that may evaluate to either a true value or a false value. In the above example, a variable `x` can be either equal to `y` or not equal to `y`. However, it cannot be both true and false. If the condition is true, then the process is executed.

A decision statement can also be stated in the following manner:

```
IF condition
      Then process1
ELSE  process2
```

This form is popularly known as the `IF-ELSE` construct. Here, if the condition is true, then `process1` is executed, else `process2` is executed. Figure 2.11 shows an algorithm to check if two numbers are equal.

Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as `while`, `do-while`, and `for` loops. These loops execute one or more steps until some condition is true. Figure 2.12 shows an algorithm that prints the first 10 natural numbers.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
             PRINT "EQUAL"
        ELSE
             PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

Figure 2.11 Algorithm to test for equality of two numbers

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

Figure 2.12 Algorithm to print the first 10 natural of

Programming Examples

1. Write an algorithm for swapping two values.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET TEMP = A
Step 4: SET A = B
Step 5: SET B = TEMP
Step 6: PRINT A, B
Step 7: END
```

2. Write an algorithm to find the larger of two numbers.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A>B
           PRINT A
        ELSE
           IF A<B
              PRINT B
        ELSE
              PRINT "The numbers are equal"
              [END OF IF]
        [END OF IF]
Step 4: END
```

3. Write an algorithm to find whether a number is even or odd.

```
Step 1: Input number as A
Step 2: IF A%2 =0
              PRINT "EVEN"
        ELSE
              PRINT "ODD"
        [END OF IF]
Step 3: END
```

4. Write an algorithm to print the grade obtained by a student using the following rules.

```
Step 1: Enter the Marks obtained as M
Step 2: IF M>75
              PRINT O
Step 3: IF M>=60 AND M<75
              PRINT A
Step 4: IF M>=50 AND M<60
              PRINT B
Step 5: IF M>=40 AND M<50
              PRINT C
        ELSE
              PRINT D
```

Marks	Grade
Above 75	O
60–75	A
50–59	B
40–49	C
Less then 40	D

```
            [END OF IF]
        Step 6: END
```

5. Write an algorithm to find the sum of first N natural numbers.

```
Step 1: Input N
Step 2: SET I = 1, SUM = 0
Step 3: Repeat Step 4 while I <= N
Step 4:         SET SUM = SUM + I
                SET I = I + 1
        [END OF LOOP]
Step 5: PRINT SUM
Step 6: END
```

2.8 TIME AND SPACE COMPLEXITY

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Generally, the space needed by a program depends on the following two parts:

- *Fixed part*: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- *Variable part*: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

However, running time requirements are more critical than memory requirements. Therefore, in this section, we will concentrate on the running time efficiency of algorithms.

2.8.1 Worst-case, Average-case, Best-case, and Amortized Time Complexity

Worst-case running time This denotes the behaviour of an algorithm with respect to the worst-possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Amortized running time Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

2.8.2 Time–Space Trade-off

The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution. But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a time–space trade-off among algorithms.

So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time. On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

2.8.3 Expressing Time and Space Complexity

The time and space complexity can be expressed using a function `f(n)` where `n` is the input size for a given instance of the problem being solved. Expressing the complexity is required when

- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function `f(n)` is the Big O notation. It provides the upper bound for the complexity.

2.8.4 Algorithm Efficiency

If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Let us consider different cases in which loops determine the efficiency of an algorithm.

Linear Loops

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
      statement block;
```

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

```
f(n) = n
```

However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

```
for(i=0;i<100;i+=2)
       statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

```
f(n) = n/2
```

Logarithmic Loops

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)
      statement block;
```

```
for(i=1000;i>=1;i/=2)
      statement block;
```

Consider the first `for` loop in which the loop-controlling variable `i` is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of `i` doubles. Now, consider the second loop in which the loop-controlling variable `i` is divided by 2. In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when `n = 1000`, the number of iterations can be given by `log 1000` which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

```
f(n) = log n
```

Nested Loops

Loops that contain loops are known as *nested loops*. In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

Linear logarithmic loop Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is `log 10`. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as `10 log 10`.

```
for(i=0;i<10;i++)
      for(j=1; j<10;j*=2)
            statement block;
```

In more general terms, the efficiency of such loops can be given as `f(n) = n log n`.

Quadratic loop In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
      for(j=0; j<10;j++)
            statement block;
```

The generalized formula for quadratic loop can be given as $f(n) = n^2$.

Dependent quadratic loop In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
      for(j=0; j<=i;j++)
            statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

$$1 + 2 + 3 + \cdots + 9 + 10 = 55$$

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates $(n + 1)/2$ times. Therefore, the efficiency of such a code can be given as

$$f(n) = n\ (n + 1)/2$$

2.9 BIG O NOTATION

In today's era of massive advancement in computer technology, we are hardly concerned about the efficiency of algorithms. Rather, we are more interested in knowing the generic order of the magnitude of the algorithm. If we have two different algorithms to solve the same problem where one algorithm executes in 10 iterations and the other in 20 iterations, the difference between the two algorithms is not much. However, if the first algorithm executes in 10 iterations and the other in 1000 iterations, then it is a matter of concern.

We have seen that the number of statements executed in the program for n elements of the data is a function of the number of elements, expressed as $f(n)$. Even if the expression derived for a function is complex, a dominant factor in the expression is sufficient to determine the order of the magnitude of the result and, hence, the efficiency of the algorithm. This factor is the Big O, and is expressed as $O(n)$.

The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of n. For example, if a sorting algorithm performs n^2 operations to sort just n elements, then that algorithm would be described as an $O(n^2)$ algorithm.

When expressing complexity using the Big O notation, constant multipliers are ignored. So, an $O(4n)$ algorithm is equivalent to $O(n)$, which is how it should be written.

If $f(n)$ and $g(n)$ are the functions defined on a positive integer number n, then

$$f(n) = O(g(n))$$

That is, f of n is Big-O of g of n if and only if positive constants c and n exist, such that $f(n) \leq cg(n)$. It means that for large amounts of data, $f(n)$ will grow no more than a constant factor than $g(n)$. Hence, g provides an upper bound. Note that here c is a constant which depends on the following factors:

- the programming language used,
- the quality of the compiler or interpreter,
- the CPU speed,
- the size of the main memory and the access time to it,
- the knowledge of the programmer, and
- the algorithm itself, which may require simple but also time-consuming machine instructions.

We have seen that the Big O notation provides a strict upper bound for $f(n)$. This means that the function $f(n)$ can do better but not worse than the specified value. Big O notation is simply written as $f(n) \in O(g(n))$ or as $f(n) = O(g(n))$.

Here, n is the problem size and $O(g(n)) = \{h(n): \exists$ positive constants c, n_0 such that $0 \leq h(n) \leq cg(n), \forall\ n \geq n_0\}$. Hence, we can say that $O(g(n))$ comprises a set of all the functions $h(n)$ that are less than or equal to $cg(n)$ for all values of $n \geq n_0$.

> If $f(n) \leq cg(n)$, $c > 0$, $\forall\ n \geq n_0$, then $f(n) = O(g(n))$ and $g(n)$ is an asymptotically tight upper bound for $f(n)$.

Examples of functions in $O(n^3)$ include: $n^{2.9}$, n^3, $n^3 + n$, $540n^3 + 10$.

Examples of functions not in $O(n^3)$ include: $n^{3.2}$, n^2, $n^2 + n$, $540n + 10$, $2n$

To summarize,

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.
- If we simply write O, it means same as worst case O.

Now let us look at some examples of `g(n)` and `f(n)`. Table 2.1 shows the relationship between `g(n)` and `f(n)`. Note that the constant values will be ignored because the main purpose of the Big O notation is to analyse the algorithm in a general fashion, so the anomalies that appear for small input sizes are simply ignored.

Table 2.1 Examples of `f(n)` and `g(n)`

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as $O(1)$
- Linear time algorithm: running time complexity given as $O(n)$
- Logarithmic time algorithm: running time complexity given as $O(\log n)$
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Table 2.2 shows the number of operations that would be performd for various values of `n`.

Table 2.2 Number of operations for different functions of `n`

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

Example 2.1 Show that $4n^2 = O(n^3)$.

Solution By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting $4n^2$ as $h(n)$ and n^3 as $g(n)$, we get

$$0 \le 4n^2 \le cn^3$$

Dividing by n^3

$$0/n^3 \le 4n^2/n^3 \le cn^3/n^3$$
$$0 \le 4/n \le c$$

Now to determine the value of `c`, we see that $4/n$ is maximum when $n=1$. Therefore, $c=4$.

To determine the value of n_0,

$$0 \le 4/n_0 \le 4$$
$$0 \le 4/4 \le n_0$$

$0 \leq 1 \leq n_0$

This means $n_0=1$. Therefore, $0 \leq 4n^2 \leq 4n^3 \; \forall \; n \geq n_0=1$.

Example 2.2 Show that $400n^3 + 20n^2 = O(n^3)$.

Solution By definition, we have

$0 \leq h(n) \leq cg(n)$

Substituting $400n^3 + 20n^2$ as $h(n)$ and n^3 as $g(n)$, we get

$0 \leq 400n^3 + 20n^2 \leq cn^3$

Dividing by n^3

$0/n^3 \leq 400n^3/n^3 + 20n^2/n^3 \leq cn^3/n^3$

$0 \leq 400 + 20/n \leq c$

Note that $20/n \longrightarrow 0$ as $n \longrightarrow \infty$, and $20/n$ is maximum when $n = 1$. Therefore,

$0 \leq 400 + 20/1 \leq c$

This means, $c = 420$

To determine the value of n_0,

$0 \leq 400 + 20/n_0 \leq 420$

$-400 \leq 400 + 20/n_0 - 400 \leq 420 - 400$

$-400 \leq 20/n_0 \leq 20$

$-20 \leq 1/n_0 \leq 1$

$-20 \; n_0 \leq 1 \leq n_0$. This implies $n_0 = 1$.

Hence, $0 \leq 400n^3 + 20n^2 \leq 420n^3 \; \forall \; n \geq n_0=1$.

Example 2.3 Show that $n = O(n\log n)$.

Solution By definition, we have

$0 \leq h(n) \leq cg(n)$

Substituting n as $h(n)$ and $n\log n$ as $g(n)$, we get

$0 \leq n \leq c \; n \log n$

Dividing by $n\log n$, we get

$0/n \log n \leq n/n \log n \leq c \; n \log n/ \; n \log n$

$0 \leq 1/\log n \leq c$

We know that $1/\log n \longrightarrow 0$ as $n \longrightarrow \infty$

To determine the value of c, it is clearly evident that $1/\log n$ is greatest when $n=2$. Therefore,

$0 \leq 1/\log 2 \leq c = 1$. Hence $c = 1$.

To determine the value of n_0, we can write

$0 \leq 1/\log n_0 \leq 1$

$0 \leq 1 \leq \log n_0$

Now, $\log n_0 = 1$, when $n_0 = 2$.

Hence, $0 \leq n \leq cn \log n$ when $c= 1$ and $\forall \; n \geq n_0=2$.

Example 2.4 Show that $10n^3 + 20n \neq O(n^2)$.

Solution By definition, we have

$0 \leq h(n) \leq cg(n)$

Substituting $10n^3 + 20n$ as $h(n)$ and n^2 as $g(n)$, we get

$0 \leq 10n^3 + 20n \leq cn^2$

Dividing by n^2

$0/n^2 \leq 10n^3/n^2 + 20n/n^2 \leq cn^2/n^2$

$0 \leq 10n + 20/n \leq c$

$0 \leq (10n^2 + 20)/n \leq c$

Hence, $10n^3 + 20n \neq O^2(n^2)$

Limitations of Big O Notation

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

2.10 OMEGA NOTATION (Ω)

The Omega notation provides a tight lower bound for $f(n)$. This means that the function can never do better than the specified value but it may do worse.

Ω notation is simply written as, $f(n) \in \Omega(g(n))$, where n is the problem size and $\Omega(g(n)) = \{h(n): \exists$ positive constants $c > 0$, n_0 such that $0 \leq cg(n) \leq h(n)$, $\forall\, n \geq n_0\}$.

Hence, we can say that $\Omega(g(n))$ comprises a set of all the functions $h(n)$ that are greater than or equal to $cg(n)$ for all values of $n \geq n_0$.

If $cg(n) \leq f(n)$, $c > 0$, $\forall\, n \geq n_0$, then $f(n) \in \Omega(g(n))$ and $g(n)$ is an asymptotically tight lower bound for $f(n)$.

Examples of functions in $\Omega(n^2)$ include: n^2, $n^{2.9}$, $n^3 + n^2$, n^3

Examples of functions not in $\Omega(n^3)$ include: n, $n^{2.9}$, n^2

To summarize,

- Best case Ω describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.
- If we simply write Ω, it means same as best case Ω.

Example 2.5 Show that $5n^2 + 10n = \Omega(n^2)$.

Solution By the definition, we can write

$$0 \leq cg(n) \leq h(n)$$
$$0 \leq cn^2 \leq 5n^2 + 10n$$

Dividing by n^2

$$0/n^2 \leq cn^2/n^2 \leq 5n^2/n^2 + 10n/n^2$$
$$0 \leq c \leq 5 + 10/n$$

Now, $\lim_{n \to \infty} 5 + 10/n = 5$.

Therefore, $0 \leq c \leq 5$.

Hence, $c = 5$

Now to determine the value of n_0

$$0 \leq 5 \leq 5 + 10/n_0$$
$$-5 \leq 5 - 5 \leq 5 + 10/n_0 - 5$$

$-5 \leq 0 \leq 10/n_0$

So $n_0 = 1$ as $\lim_{n \to \infty} 1/n = 0$

Hence, $5n^2 + 10n = \Omega(n^2)$ for $c=5$ and $\forall\, n \geq n_0=1$.

Example 2.6 Show that $7n \neq \Omega(n^2)$.

Solution By the definition, we can write

$0 \leq cg(n) \leq h(n)$
$0 \leq cn^2 \leq 7n$

Dividing by n^2, we get

$0/n^2 \leq cn^2/n^2 \leq 7n/n^2$
$0 \leq c \leq 7/n$

Thus, from the above statement, we see that the value of c depends on the value of n. There does not exist a value of n_0 that satisfies the condition as n increases. This could fairly be possible if $c = 0$ but it is not allowed as the definition by itself says that $\lim_{n \to \infty} 1/n = 0$.

2.11 THETA NOTATION (Θ)

Theta notation provides an asymptotically tight bound for $f(n)$. Θ notation is simply written as, $f(n) \in \Theta(g(n))$, where n is the problem size and
$\Theta(g(n)) = \{h(n)$: ∃ positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq h(n) \leq c_2 g(n)$, $\forall\, n \geq n_0\}$.

Hence, we can say that $\Theta(g(n))$ comprises a set of all the functions $h(n)$ that are between $c_1 g(n)$ and $c_2 g(n)$ for all values of $n \geq n_0$.

If $f(n)$ is between $c_1 g(n)$ and $c_2 g(n)$, $\forall\, n \geq n_0$, then $f(n) \in \Theta(g(n))$ and $g(n)$ is an asymptotically tight bound for $f(n)$ and $f(n)$ is amongst $h(n)$ in the set.

To summarize,

- The best case in Θ notation is not used.
- Worst case Θ describes asymptotic bounds for worst case combination of input values.
- If we simply write Θ, it means same as worst case Θ.

Example 2.7 Show that $n^2/2 - 2n = \Theta(n^2)$.

Solution By the definition, we can write

$c_1 g(n) \leq h(n) \leq c_2 g(n)$
$c_1 n^2 \leq n^2/2 - 2n \leq c_2 n^2$

Dividing by n^2, we get

$c_1 n^2/n^2 \leq n^2/2n^2 - 2n/n^2 \leq c_2 n^2/n^2$
$c_1 \leq 1/2 - 2/n \leq c_2$

This means $c_2 = 1/2$ because $\lim_{n \to \infty} 1/2 - 2/n = 1/2$ (Big O notation)

To determine c_1 using Ω notation, we can write

$0 < c_1 \leq 1/2 - 2/n$

We see that $0 < c_1$ is minimum when $n = 5$. Therefore,

$0 < c_1 \leq 1/2 - 2/5$

Hence, $c_1 = 1/10$

Now let us determine the value of n_0

$1/10 \leq 1/2 - 2/n_0 \leq 1/2$
$2/n_0 \leq 1/2 - 1/10 \leq 1/2$
$2/n_0 \leq 2/5 \leq 1/2$

$n_0 \geq 5$

You may verify this by substituting the values as shown below.

$c_1n^2 \leq n^2/2 - 2n \leq c_2n^2$

$c_1 = 1/10$, $c_2 = 1/2$ and $n_0 = 5$

$1/10(25) \leq 25/2 - 20/2 \leq 25/2$

$5/2 \leq 5/2 \leq 25/2$

Thus, in general, we can write, $1/10n^2 \leq n^2/2 - 2n \leq 1/2n^2$ for $n \geq 5$.

2.12 OTHER USEFUL NOTATIONS

There are other notations like little o notation and little ω notation which have been discussed below.

Little o Notation

This notation provides a non-asymptotically tight upper bound for $f(n)$. To express a function using this notation, we write

$f(n) \in o(g(n))$ where

$o(g(n)) = \{h(n) : \exists$ positive constants c, n_0 such that for any $c > 0$, $n_0 > 0$, and $0 \leq h(n) \leq cg(n)$, $\forall n \geq n_0\}$.

This is unlike the Big O notation where we say for some $c > 0$ (not any). For example, $5n^3 = o(n^3)$ is asymptotically tight upper bound but $5n^2 = o(n^3)$ is non-asymptotically tight bound for $f(n)$.

Examples of functions in $o(n^3)$ include: $n^{2.9}$, $n^3 / \log n$, $2n^2$

Examples of functions not in $o(n^3)$ include: $3n^3$, n^3, $n^3 / 1000$

Example 2.8 Show that $n^3 / 1000 \neq o(n^3)$.

Solution By definition, we have

$0 \leq h(n) < cg(n)$, for any constant $c > 0$

$0 \leq n^3 / 1000 \leq cn^3$

This is in contradiction with selecting any $c < 1/1000$.

An imprecise analogy between the asymptotic comparison of functions $f(n)$ and $g(n)$ and the relation between their values can be given as:

$f(n) = O(g(n)) \approx f(n) \leq g(n)$ $\quad$ $f(n) = o(g(n)) \approx f(n) < g(n)$ $\quad$ $f(n) = \Theta(g(n)) \approx f(n) = g(n)$

Little Omega Notation (ω)

This notation provides a non-asymptotically tight lower bound for $f(n)$. It can be simply written as, $f(n) \in \omega(g(n))$, where

$\omega(g(n)) = \{h(n) : \exists$ positive constants c, n_0 such that for any $c > 0$, $n_0 > 0$, and $0 \leq cg(n) < h(n), \forall n \geq n_0\}$.

This is unlike the Ω notation where we say for some $c > 0$ (not any). For example, $5n^3 = \Omega(n^3)$ is asymptotically tight upper bound but $5n^2 = \omega(n^3)$ is non-asymptotically tight bound for $f(n)$.

Example of functions in $\omega(g(n))$ include: $n^3 = \omega(n^2)$, $n^{3.001} = \omega(n^3)$, $n^2 \log n = \omega(n^2)$

Example of a function not in $\omega(g(n))$ is $5n^2 \neq \omega(n^2)$ (just as $5 \neq 5$)

Example 2.9 Show that $50n^3/100 \neq \omega(n^3)$.

Solution By definition, we have

$0 \leq cg(n) < h(n)$, for any constant $c > 0$

$0 \leq cn^3 < 50n^3/100$

Dividing by n^3, we get

$0 \leq c < 50/100$

This is a contradictory value as for any value of c as it cannot be assured to be less than 50/100 or 1/2.

An imprecise analogy between the asymptotic comparison of functions $f(n)$ and $g(n)$ and the relation between their values can be given as:

$f(n) = \Omega(g(n)) \approx f(n) \geq g(n)$ $\quad\quad$ $f(n) = \omega(g(n)) \approx f(n) > g(n)$

Points to Remember

- A data structure is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.
- There are two types of data structures: primitive and non-primitive data structures. Primitive data structures are the fundamental data types which are supported by a programming language. Non-primitive data structures are those data structures which are created using primitive data structures.
- Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.
- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. However, if the elements of a data structure are not stored in sequential order, then it is a non-linear data structure.
- An array is a collection of similar data elements which are stored in consecutive memory locations.
- A linked list is a linear data structure consisting of a group of elements (called nodes) which together represent a sequence.
- A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.
- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical tree structure.
- The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.
- A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships can exist between the nodes.
- An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- An algorithm is basically a set of instructions that solve a problem.
- The time complexity of an algorithm is basically the running time of the program as a function of the input size.
- The space complexity of an algorithm is the amount of computer memory required during the program execution as a function of the input size.
- The worst-case running time of an algorithm is an upper bound on the running time for any input.
- The average-case running time specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
- Amortized analysis guarantees the average performance of each operation in the worst case.
- The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed and the type of the loop that is being used.

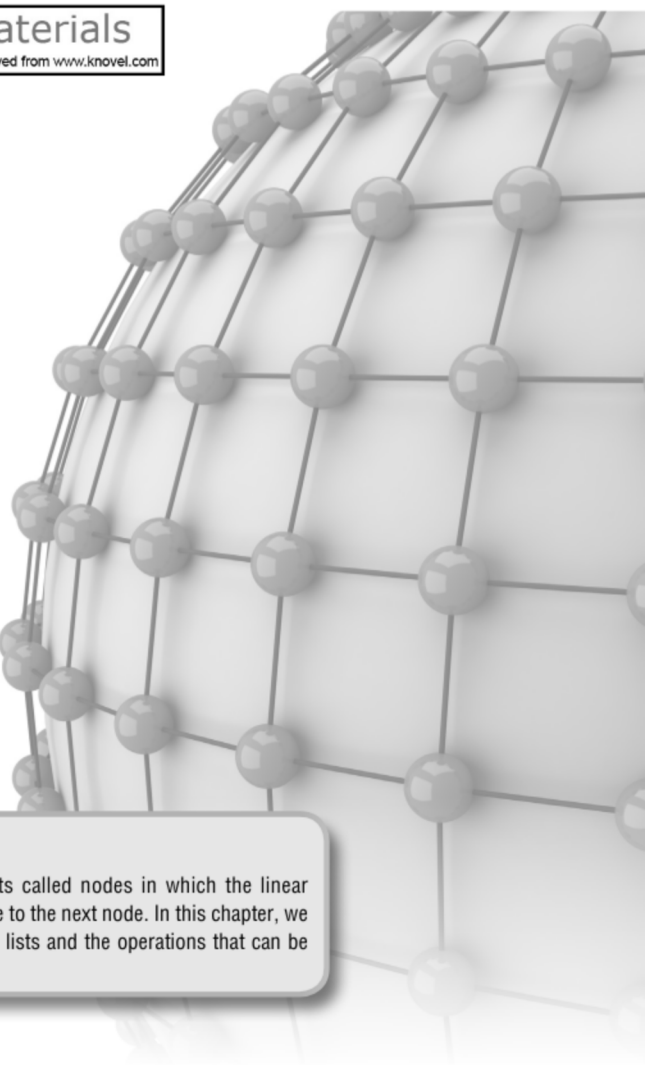
Exercises

Review Questions

1. Explain the features of a good program.
2. Define the terms: data, file, record, and primary key.

CHAPTER 6

Linked Lists



LEARNING OBJECTIVE

A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node. In this chapter, we are going to discuss different types of linked lists and the operations that can be performed on these lists.

6.1 INTRODUCTION

We have studied that an array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as `int marks[10]`, then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

6.1.1 Basic Terminologies

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*. Linked list is a data structure which in turn can be used to implement other data

structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

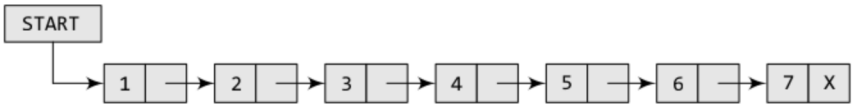


Figure 6.1 Simple linked list

In Fig. 6.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The left part of the node which contains data may include a simple data type, an array, or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL. In Fig. 6.1, the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code:

```
struct node
{
      int data;
      struct node *next;
};
```

Note Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.

START
1

	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	–1

Figure 6.2 START pointing to the first element of the linked list in the memory

Let us see how a linked list is maintained in the memory. In order to form a linked list, we need a structure called *node* which has two fields, DATA and NEXT. DATA will store the information part and NEXT will store the address of the next node in sequence. Consider Fig. 6.2.

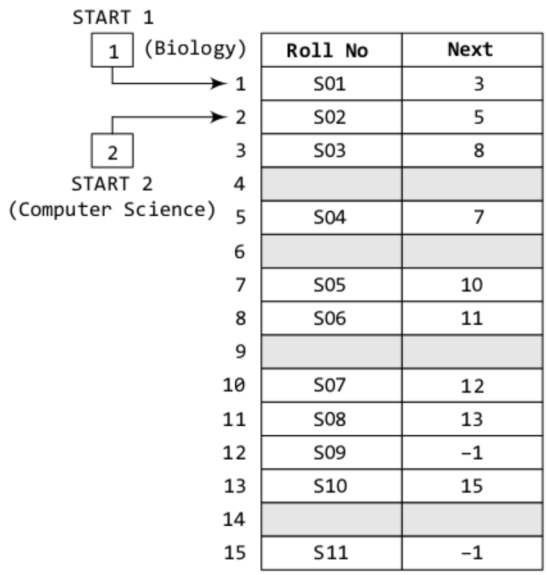
In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START = 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item. The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this

would denote the end of the linked list. When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Note that Fig. 6.2 shows a chunk of memory locations which range from 1 to 10. The shaded portion contains data for other applications. Remember that the nodes of a linked list need not be in consecutive memory locations. In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.

Let us take another example to see how two linked lists are maintained together in the computer's memory. For example, the students of Class XI of Science group are asked to choose between Biology and Computer Science. Now, we will maintain two linked lists, one for each subject. That is, the first linked list will contain the roll numbers of all the students who have opted for Biology and the second list will contain the roll numbers of students who have chosen Computer Science.

Now, look at Fig. 6.3, two different linked lists are simultaneously maintained in the memory. There is no ambiguity in traversing through the list because each list maintains a separate START pointer, which gives the address of the first node of their respective linked lists. The rest of the nodes are reached by looking at the value stored in the NEXT.



	Roll No	Next
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	-1
13	S10	15
14		
15	S11	-1

Figure 6.3 Two linked lists which are simultaneously maintained in the memory

By looking at the figure, we can conclude that roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11. Similarly, roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

We have already said that the DATA part of a node may contain just a single data item, an array, or a structure. Let us take an example to see how a structure is maintained in a linked list that is stored in the memory.

Consider a scenario in which the roll number, name, aggregate, and grade of students are stored using linked lists. Now, we will see how the NEXT pointer is used to store the data alphabetically. This is shown in Fig. 6.4.

6.1.2 Linked Lists versus Arrays

Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations. Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array. For example, if we declare an array as `int marks[20]`, then the array can store a maximum of 20 data elements only. There is no such restriction in case of a linked list.

START
18

	Roll No	Name	Aggregate	Grade	Next
1	S01	Ram	78	Distinction	6
2	S02	Shyam	64	First division	14
3					
4	S03	Mohit	89	Outstanding	17
5					
6	S04	Rohit	77	Distinction	2
7	S05	Varun	86	Outstanding	10
8	S06	Karan	65	First division	12
9					
10	S07	Veena	54	Second division	-1
11	S08	Meera	67	First division	4
12	S09	Krish	45	Third division	13
13	S10	Kusum	91	Outstanding	11
14	S11	Silky	72	First division	7
15					
16					
17	S12	Monica	75	Distinction	1
18	S13	Ashish	63	First division	19
19	S14	Gaurav	61	First division	8

Figure 6.4 Students' linked list

Thus, linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes.

6.1.3 Memory Allocation and De-allocation for a Linked List

We have seen how a linked list is represented in the memory. If we want to add a node to an already existing linked list in the memory, we first find free space in the memory and then use it to store the information. For example, consider the linked list shown in Fig. 6.5. The linked list contains the roll number of students, marks obtained by them in Biology, and finally a NEXT field which stores the address of the next node in sequence. Now, if a new student joins the class and is asked to appear for the same test that the other students had taken, then the new student's marks should also be recorded in the linked list. For this purpose, we find a free space and store the information there. In Fig. 6.5 the grey shaded portion shows free space, and thus we have 4 memory locations available. We can use any one of them to store our data. This is illustrated in Figs 6.5(a) and (b).

Now, the question is which part of the memory is available and which part is occupied? When we delete a node from a linked list, then who changes the status of the memory occupied by it from occupied to available? The answer is the operating system. Discussing the mechanism of how the operating system does all this is out of the scope of this book. So, in simple language, we can say that the computer does it on its own without any intervention from the user or the programmer. As a programmer, you just have to take care of the code to perform insertions and deletions in the list.

However, let us briefly discuss the basic concept behind it. The computer maintains a list of all free memory cells. This list of available space is called the *free pool*.

START
1
(Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	−1

(a)

START
1
(Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4	S12	75	−1
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	4

(b)

Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

We have seen that every linked list has a pointer variable START which stores the address of the first node of the list. Likewise, for the free pool (which is a linked list of all free memory cells), we have a pointer variable AVAIL which stores the address of the first free space. Let us revisit the memory representation of the linked list storing all the students' marks in Biology.

Now, when a new student's record has to be added, the memory address pointed by AVAIL will be taken and used to store the desired information. After the insertion, the next available free space's address will be stored in AVAIL. For example, in Fig. 6.6, when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.

START
1
(Biology)

4
AVAIL

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			−1
15	S11	56	−1

Figure 6.6 Linked list with AVAIL and START pointers

This was all about inserting a new node in an already existing linked list. Now, we will discuss deleting a node or the entire linked list. When we delete a particular node from an existing linked list or delete the entire linked list, the space occupied by it must be given back to the free pool so that the memory can be reused by some other program that needs memory space.

The operating system does this task of adding the freed memory to the free pool. The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space. The operating system scans through all the memory cells and marks those cells that are being used by some program. Then it collects all the cells which are not being used and adds

their address to the free pool, so that these cells can be reused by other programs. This process is called *garbage collection*.

There are different types of linked lists which we will discuss in the next section.

6.2 SINGLY LINKED LISTS

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way. Figure 6.7 shows a singly linked list.

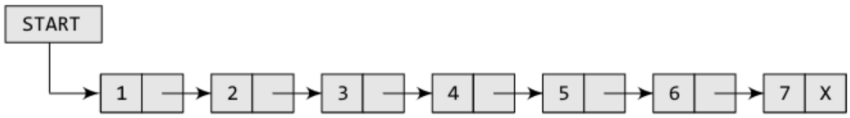


Figure 6.7 Singly linked list

6.2.1 Traversing a Linked List

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. The algorithm to traverse a linked list is shown in Fig. 6.8.

In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list. Then in Step 2, a `while` loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL. In Step 3, we apply the process (e.g., `print`) to the current node, that is, the node pointed by PTR. In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: EXIT
```

Figure 6.8 Algorithm for traversing a linked list

Let us now write an algorithm to count the number of nodes in a linked list. To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Figure 6.9 shows the algorithm to print the number of nodes in a linked list.

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:            SET COUNT = COUNT + 1
Step 5:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Figure 6.9 Algorithm to print the number of nodes in a linked list

6.2.2 Searching for a Value in a Linked List

Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                   SET POS = PTR
                   Go To Step 5
             ELSE
                   SET PTR = PTR -> NEXT
             [END OF IF]
       [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

Figure 6.10 shows the algorithm to search a linked list.

In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made. If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Consider the linked list shown in Fig. 6.11. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

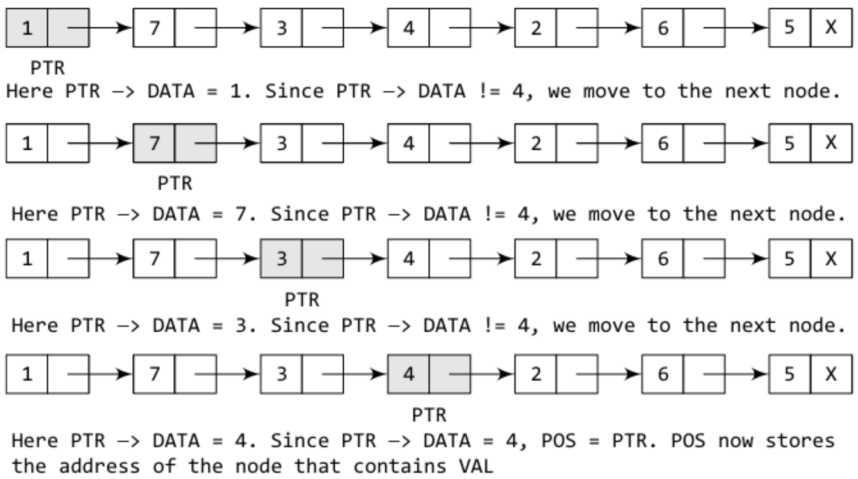


Figure 6.11 Searching a linked list

6.2.3 Inserting a New Node in a Linked List

In this section, we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Inserting a Node at the Beginning of a Linked List

Consider the linked list shown in Fig. 6.12. Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.

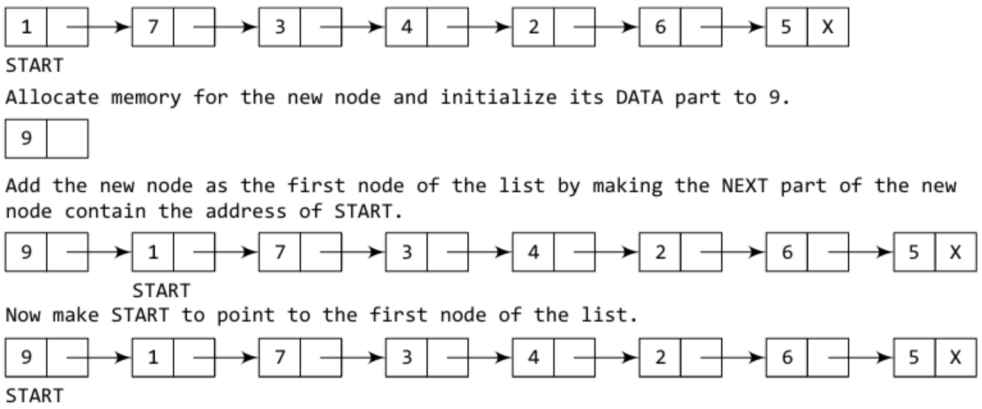


Figure 6.12 Inserting an element at the beginning of a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 6.13 Algorithm to insert a new node at the beginning

Figure 6.13 shows the algorithm to insert a new node at the beginning of a linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE. Note the following two steps:

```
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
```

These steps allocate memory for the new node. In C, there are functions like `malloc()`, `alloc`, and `calloc()` which automatically do the memory allocation on behalf of the user.

Inserting a Node at the End of a Linked List

Consider the linked list shown in Fig. 6.14. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

Figure 6.15 shows the algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the `while` loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

Inserting a Node After a Given Node in a Linked List

Consider the linked list shown in Fig. 6.17. Suppose we want to add a new node with value 9 after the node containing data 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.16.

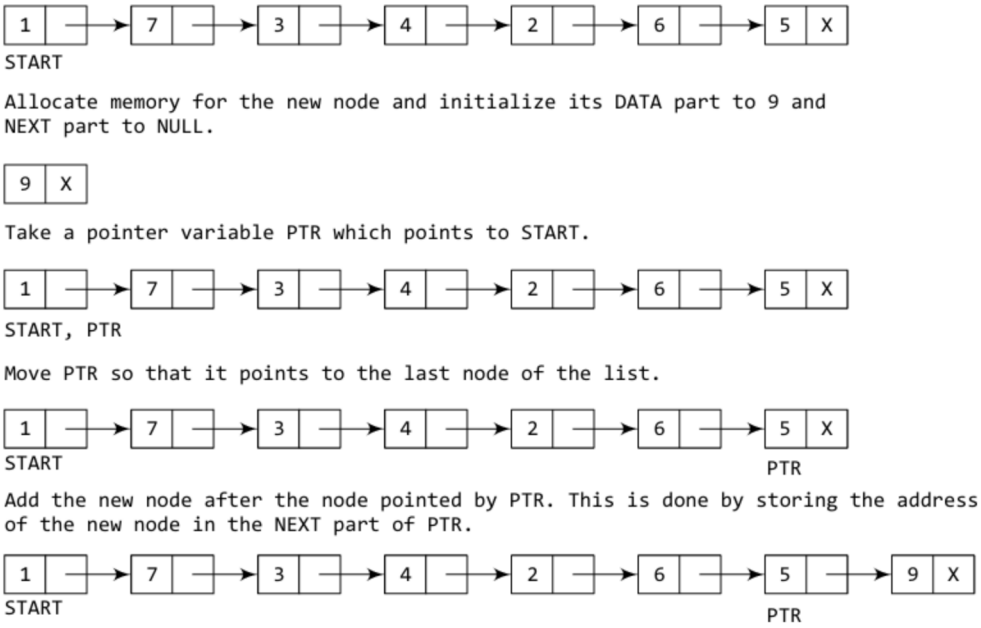


Figure 6.14 Inserting an element at the end of a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.15 Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
          != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 6.16 Algorithm to insert a new node after a node that has value NUM

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

In the `while` loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

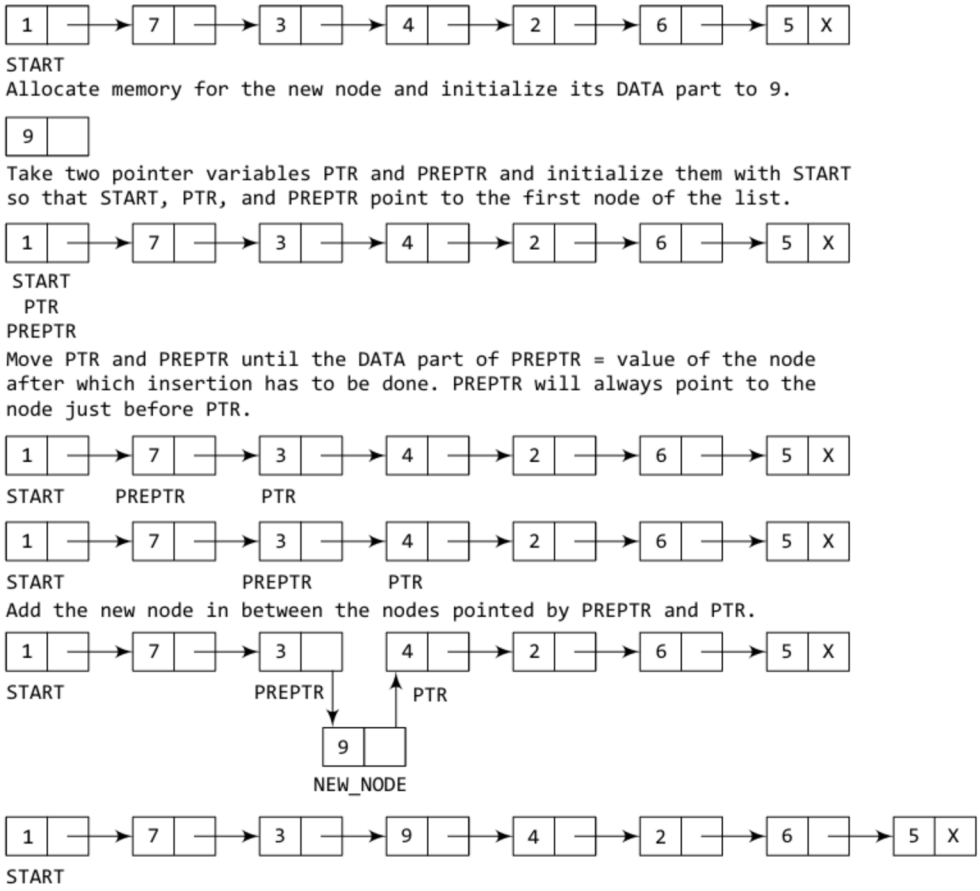


Figure 6.17 Inserting an element after a given node in a linked list

Inserting a Node Before a Given Node in a Linked List

Consider the linked list shown in Fig. 6.19. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.18.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 6.18 Algorithm to insert a new node before a node that has value NUM

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

In the `while` loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach

this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

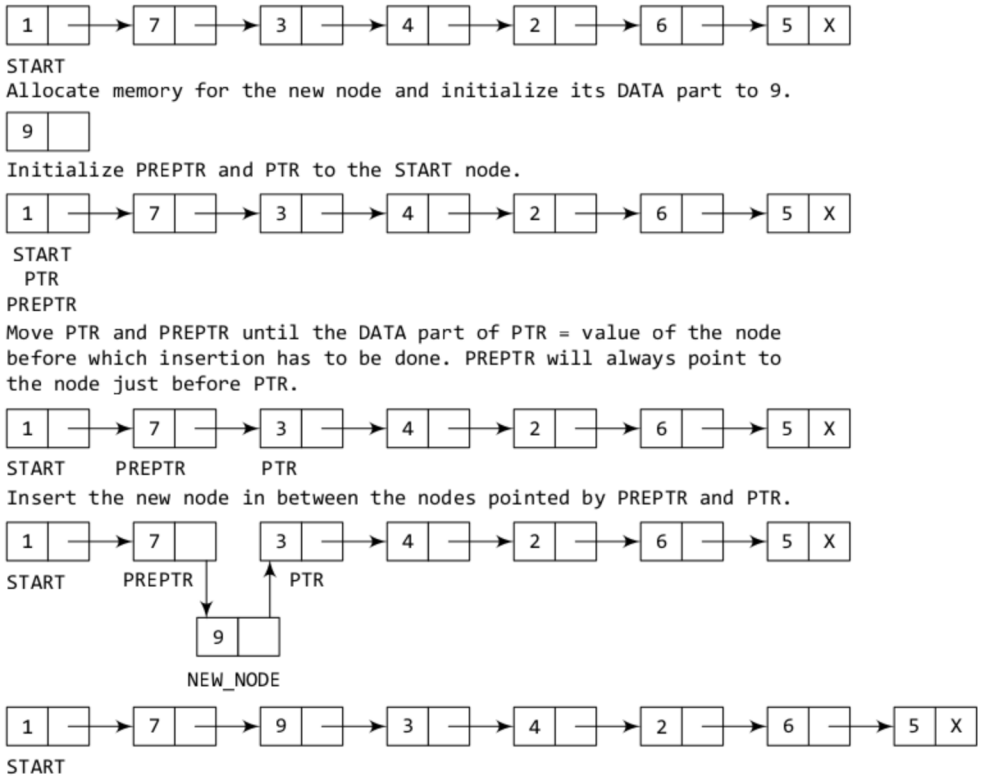


Figure 6.19 Inserting an element before a given node in a linked list

6.2.4 Deleting a Node from a Linked List

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.

Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Deleting the First Node from a Linked List

Consider the linked list in Fig. 6.20. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

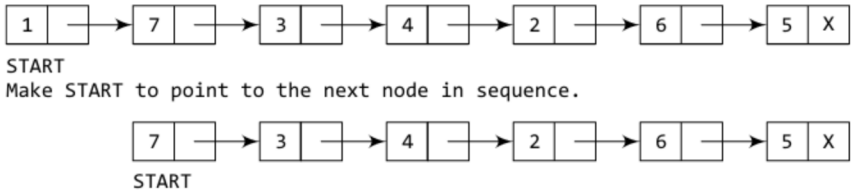


Figure 6.20 Deleting the first node of a linked list

Figure 6.21 shows the algorithm to delete the first node from a linked list. In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Deleting the Last Node from a Linked List

Consider the linked list shown in Fig. 6.22. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

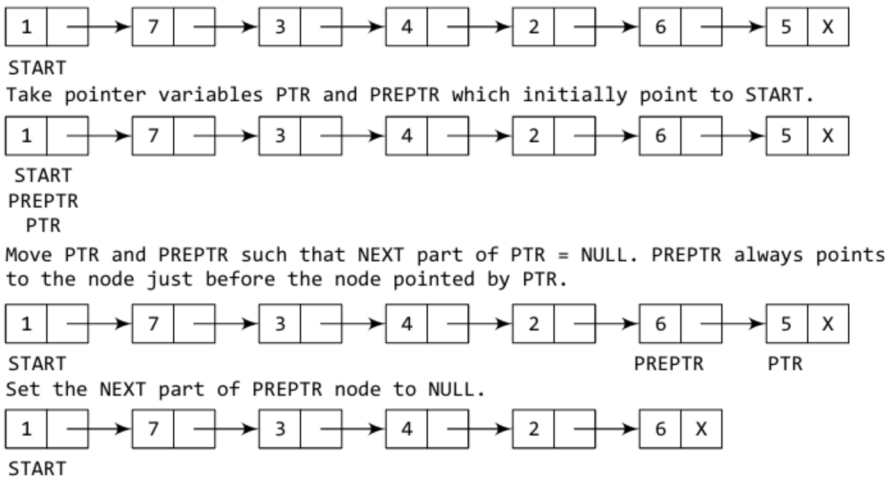


Figure 6.22 Deleting the last node of a linked list

Figure 6.23 shows the algorithm to delete the last node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned back to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.23 Algorithm to delete the last node

Deleting the Node After a Given Node in a Linked List

Consider the linked list shown in Fig. 6.24. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

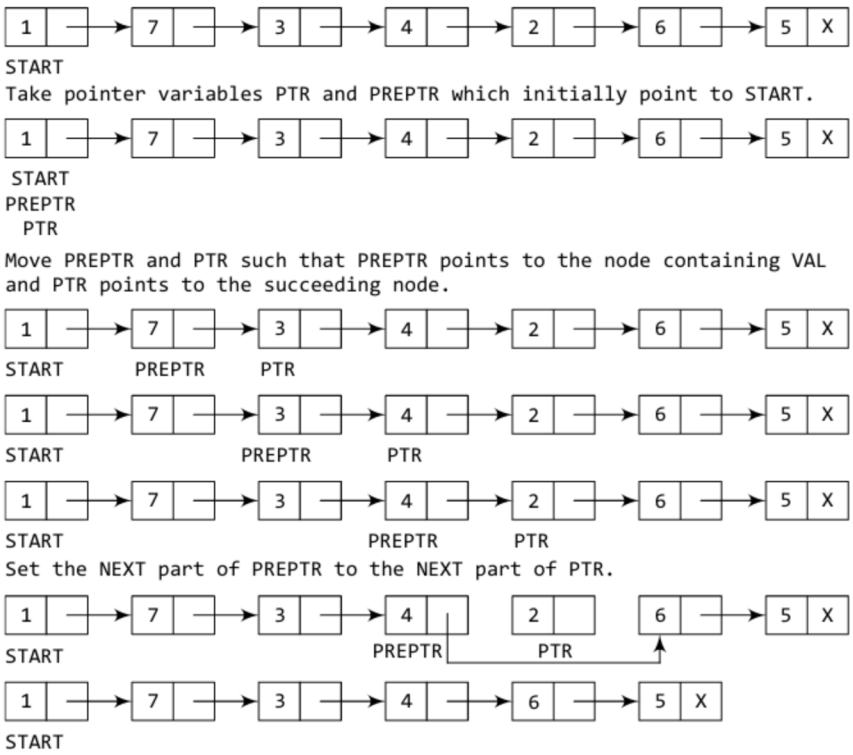


Figure 6.24 Deleting the node after a given node in a linked list

Figure 6.25 shows the algorithm to delete the node after a given node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Figure 6.25 Algorithm to delete the node after a given node

Programming Example

1. Write a program to create a linked list and perform insertions and deletions of all cases. Write functions to sort and finally delete the entire list at once.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *next;
};
struct node *start = NULL;
struct node *create_ll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *insert_before(struct node *);
struct node *insert_after(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_node(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
struct node *sort_list(struct node *);


int main(int argc, char *argv[]) {
        int option;
        do
        {
                printf("\n\n *****MAIN MENU *****");
                printf("\n 1:  Create a list");
                printf("\n 2:  Display the list");
                printf("\n 3:  Add a node at the beginning");
                printf("\n 4:  Add a node at the end");
                printf("\n 5:  Add a node before a given node");
                printf("\n 6:  Add a node after a given node");
                printf("\n 7:  Delete a node from the beginning");
```

```
                printf("\n 8:  Delete a node from the end");
                printf("\n 9:  Delete a given node");
                printf("\n 10: Delete a node after a given node");
                printf("\n 11: Delete the entire list");
                printf("\n 12: Sort the list");
                printf("\n 13: EXIT");
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                case 1: start = create_ll(start);
                        printf("\n LINKED LIST CREATED");
                        break;
                case 2: start = display(start);
                        break;
                case 3: start = insert_beg(start);
                        break;
                case 4: start = insert_end(start);
                        break;
                case 5: start = insert_before(start);
                        break;
                case 6: start = insert_after(start);
                        break;
                case 7: start = delete_beg(start);
                        break;
                case 8: start = delete_end(start);
                        break;
                case 9: start = delete_node(start);
                        break;
                case 10: start = delete_after(start);
                        break;
                case 11: start = delete_list(start);
                        printf("\n LINKED LIST DELETED");
                        break;
                case 12: start = sort_list(start);
                        break;
                }
        }while(option !=13);
        getch();
        return 0;
}
struct node *create_ll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num!=-1)
        {
                new_node = (struct node*)malloc(sizeof(struct node));
                new_node -> data=num;
                if(start==NULL)
                {
                        new_node -> next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr=start;
```

```
                        while(ptr->next!=NULL)
                        ptr=ptr->next;
                        ptr->next = new_node;
                        new_node->next=NULL;
                }
                printf("\n Enter the data : ");
                scanf("%d", &num);
        }
        return start;
}
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        return start;
}
struct node *insert_beg(struct node *start)
{
        struct node *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        new_node -> next = start;
        start = new_node;
        return start;
}
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        new_node -> next = NULL;
        ptr = start;
        while(ptr -> next != NULL)
        ptr = ptr -> next;
        ptr -> next = new_node;
        return start;
}
struct node *insert_before(struct node *start)
{
        struct node *new_node, *ptr, *preptr;
        int num, val;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        printf("\n Enter the value before which the data has to be inserted : ");
        scanf("%d", &val);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        while(ptr -> data != val)
        {
```

```
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next = new_node;
        new_node -> next = ptr;
        return start;
}
struct node *insert_after(struct node *start)
{
        struct node *new_node, *ptr, *preptr;
        int num, val;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        printf("\n Enter the value after which the data has to be inserted : ");
        scanf("%d", &val);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        preptr = ptr;
        while(preptr -> data != val)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next=new_node;
        new_node -> next = ptr;
        return start;
}
struct node *delete_beg(struct node *start)
{
        struct node *ptr;
        ptr = start;
        start = start -> next;
        free(ptr);
        return start;
}
struct node *delete_end(struct node *start)
{
        struct node *ptr, *preptr;
        ptr = start;
        while(ptr -> next != NULL)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next = NULL;
        free(ptr);
        return start;
}
struct node *delete_node(struct node *start)
{
        struct node *ptr, *preptr;
        int val;
        printf("\n Enter the value of the node which has to be deleted : ");
        scanf("%d", &val);
        ptr = start;
        if(ptr -> data == val)
        {
                start = delete_beg(start);
                return start;
        }
        else
        {
```

```
                while(ptr -> data != val)
                {
                        preptr = ptr;
                        ptr = ptr -> next;
                }
                preptr -> next = ptr -> next;
                free(ptr);
                return start;
        }
}
struct node *delete_after(struct node *start)
{
        struct node *ptr, *preptr;
        int val;
        printf("\n Enter the value after which the node has to deleted : ");
        scanf("%d", &val);
        ptr = start;
        preptr = ptr;
        while(preptr -> data != val)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next=ptr -> next;
        free(ptr);
        return start;
}
struct node *delete_list(struct node *start)
{
        struct node *ptr; // Lines 252-254 were modified from original code to fix
unresposiveness in output window
        if(start!=NULL){
                ptr=start;
                while(ptr != NULL)
                {
                        printf("\n %d is to be deleted next", ptr -> data);
                        start = delete_beg(ptr);
                        ptr = start;
                }
        }

        return start;
}
struct node *sort_list(struct node *start)
{
        struct node *ptr1, *ptr2;
        int temp;
        ptr1 = start;
        while(ptr1 -> next != NULL)
        {
                ptr2 = ptr1 -> next;
                while(ptr2 != NULL)
                {
                        if(ptr1 -> data > ptr2 -> data)
                        {
                                temp = ptr1 -> data;
                                ptr1 -> data = ptr2 -> data;
                                ptr2 -> data = temp;
                        }
                        ptr2 = ptr2 -> next;
                }
                ptr1 = ptr1 -> next;
```

```
                    }
            return start;  // Had to be added
    }
```

Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
3: Add a node at the beginning
4: Add the node at the end
5: Add the node before a given node
6: Add the node after a given node
7: Delete a node from the beginning
8: Delete a node from the end
9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: Exit
Enter your option : 3
Enter your option : 73
```

6.3 CIRCULAR LINKED LISTS

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending. Figure 6.26 shows a circular linked list.

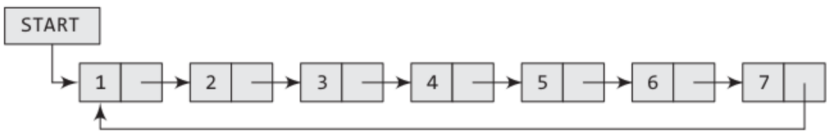


Figure 6.26 Circular linked list

The only downside of a circular linked list is the complexity of iteration. Note that there are no NULL values in the NEXT part of any of the nodes of list.

START → 1

	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

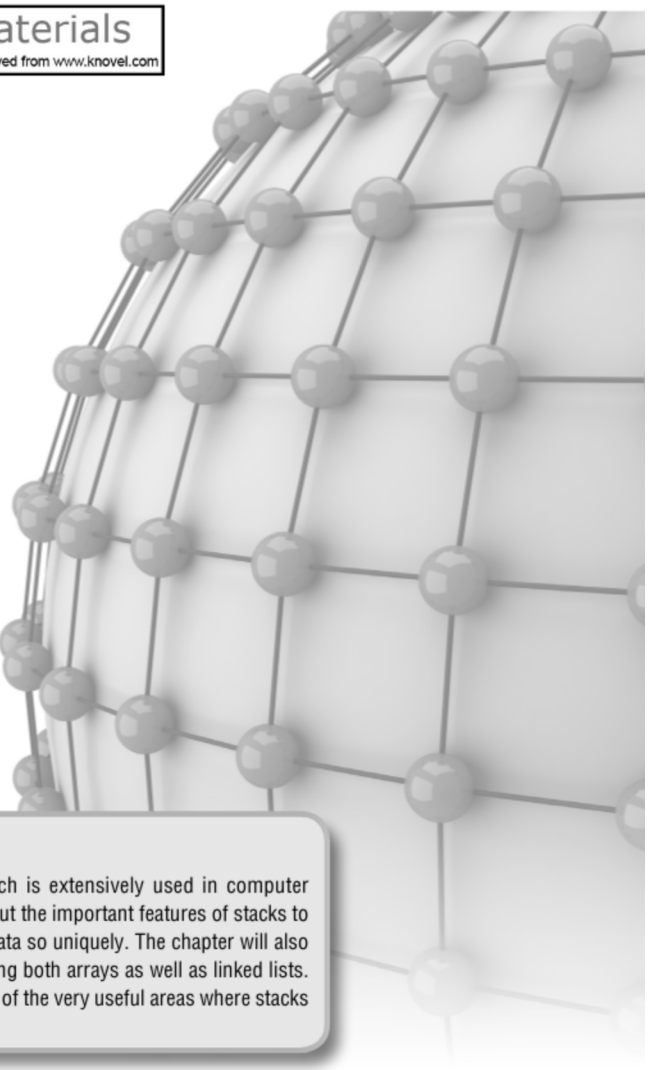
Figure 6.27 Memory representation of a circular linked list

Circular linked lists are widely used in operating systems for task maintenance. We will now discuss an example where a circular linked list is used. When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited. How is this done? The answer is simple. A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually, this is done using either the circular stack or the circular queue. We will read about circular queues in Chapter 8. Consider Fig. 6.27.

We can traverse the list until we find the NEXT entry that contains the address of the first node of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually

CHAPTER 7

Stacks



LEARNING OBJECTIVE

A stack is an important data structure which is extensively used in computer applications. In this chapter we will study about the important features of stacks to understand how and why they organize the data so uniquely. The chapter will also illustrate the implementation of stacks by using both arrays as well as linked lists. Finally, the chapter will discuss in detail some of the very useful areas where stacks are primarily used.

7.1 INTRODUCTION

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 7.1. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

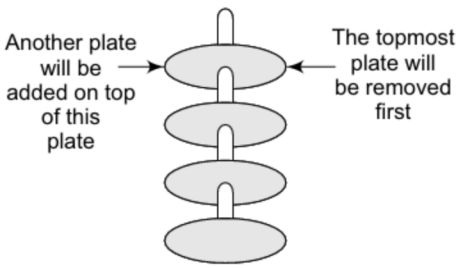


Figure 7.1 Stack of plates

Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

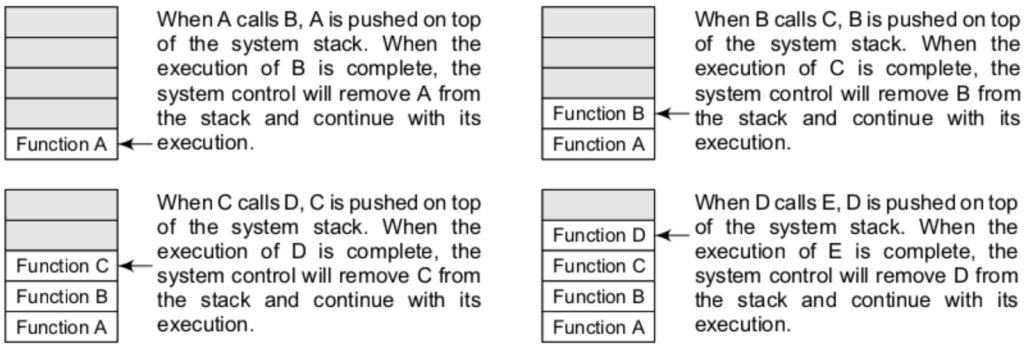


Figure 7.2 System stack in the case of function calls

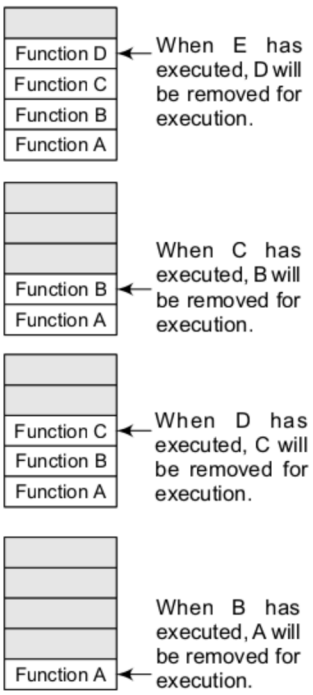


Figure 7.3 System stack when a called function returns to the calling function

In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function. Look at Fig. 7.2 which shows this concept.

Now when function `E` is executed, function `D` will be removed from the top of the stack and executed. Once function `D` gets completely executed, function `C` will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed. This is shown in Fig. 7.3.

The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks.

7.2 ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called `TOP` associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called `MAX`, which is used to store the maximum number of elements that the stack can hold.

If `TOP = NULL`, then it indicates that the stack is empty and if `TOP = MAX-1`, then the stack is full. (You must be wondering why we have written `MAX-1`. It is because array indices start from 0.) Look at Fig. 7.4.

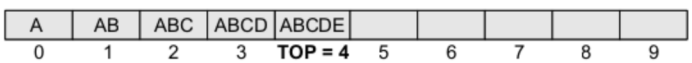


Figure 7.4 Stack

The stack in Fig. 7.4 shows that `TOP = 4`, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

7.3 OPERATIONS ON A STACK

A stack supports three basic operations: `push`, `pop`, and `peek`. The `push` operation adds an element to the top of the stack and the `pop` operation removes the element from the top of the stack. The `peek` operation returns the value of the topmost element of the stack.

7.3.1 Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if `TOP=MAX-1`, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an `OVERFLOW` message is printed. Consider the stack given in Fig. 7.5.

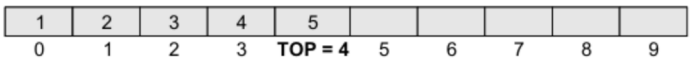


Figure 7.5 Stack

To insert an element with value 6, we first check if `TOP=MAX-1`. If the condition is false, then we increment the value of `TOP` and store the new element at the position given by `stack[TOP]`. Thus, the updated stack becomes as shown in Fig. 7.6.

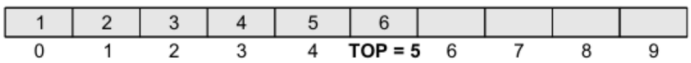


Figure 7.6 Stack after insertion

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

Figure 7.7 shows the algorithm to insert an element in a stack. In Step 1, we first check for the `OVERFLOW` condition. In Step 2, `TOP` is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by `TOP`.

7.3.2 Pop Operation

The `pop` operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if `TOP=NULL` because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an `UNDERFLOW` message is printed. Consider the stack given in Fig. 7.8.

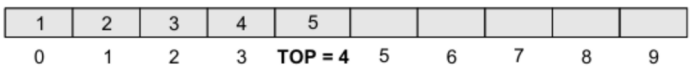


Figure 7.8 Stack

To delete the topmost element, we first check if `TOP=NULL`. If the condition is false, then we decrement the value pointed by `TOP`. Thus, the updated stack becomes as shown in Fig. 7.9.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

Figure 7.9 Stack after deletion

Figure 7.10 shows the algorithm to delete an element from a stack. In Step 1, we first check for the `UNDERFLOW` condition. In Step 2, the value of the location in the stack pointed by `TOP` is stored in `VAL`. In Step 3, `TOP` is decremented.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for Peek operation

7.3.3 Peek Operation

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack. The algorithm for Peek operation is given in Fig. 7.11.

However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned. Consider the stack given in Fig. 7.12.

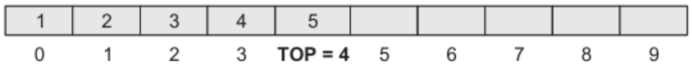


Figure 7.12 Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

PROGRAMMING EXAMPLE

1. Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);

int main(int argc, char *argv[]) {
        int val, option;
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        push(st, val);
                        break;
                case 2:
                        val = pop(st);
                        if(val != -1)
                        printf("\n The value deleted from stack is: %d", val);
                        break;
                case 3:
                        val = peek(st);
                        if(val != -1)
```

```
                        printf("\n The value stored at top of stack is: %d", val);
                        break;
                case 4:
                        display(st);
                        break;
                }
        }while(option != 5);
        return 0;
}
void push(int st[], int val)
{
        if(top == MAX-1)
        {
                printf("\n STACK OVERFLOW");
        }
        else
        {
                top++;
                st[top] = val;
        }
}
int pop(int st[])
{
        int val;
        if(top == -1)
        {
                printf("\n STACK UNDERFLOW");
                return -1;
        }
        else
        {
                val = st[top];
                top--;
                return val;
        }
}
void display(int st[])
{
        int i;
        if(top == -1)
        printf("\n STACK IS EMPTY");
        else
        {
                for(i=top;i>=0;i--)
                printf("\n %d",st[i]);
                printf("\n"); // Added for formatting purposes
        }
}
int peek(int st[])
{
        if(top == -1)
        {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
        return (st[top]);
}
```

Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 500
```

7.4 LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with `n` elements is `O(n)`, and the typical time requirement for the operations is `O(1)`.

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The `START` pointer of the linked list is used as `TOP`. All insertions and deletions are done at the node pointed by `TOP`. If `TOP = NULL`, then it indicates that the stack is empty.

The linked representation of a stack is shown in Fig. 7.13.

Figure 7.13 Linked stack

7.5 OPERATIONS ON A LINKED STACK

A linked stack supports all the three stack operations, that is, `push`, `pop`, and `peek`.

7.5.1 Push Operation

The `push` operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

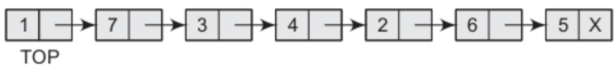


Figure 7.14 Linked stack

To insert an element with value 9, we first check if `TOP=NULL`. If this is the case, then we allocate memory for a new node, store the value in its `DATA` part and `NULL` in its `NEXT` part. The new node will then be called `TOP`. However, if `TOP!=NULL`, then we insert the new node at the beginning of the linked stack and name this new node as `TOP`. Thus, the updated stack becomes as shown in Fig. 7.15.

Figure 7.15 Linked stack after inserting a new node

Figure 7.16 shows the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the `DATA` part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

7.5.2 Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.

Figure 7.17 Linked stack

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

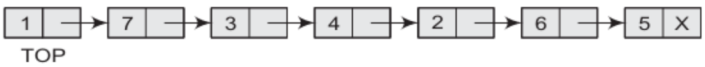


Figure 7.18 Linked stack after deletion of the topmost element

Figure 7.19 shows the algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

PROGRAMMING EXAMPLE

2. Write a program to implement a linked stack.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct stack
{
        int data;
        struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);

int main(int argc, char *argv[]) {
        int val, option;
```

```
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        top = push(top, val);
                        break;
                case 2:
                        top = pop(top);
                        break;
                case 3:
                        val = peek(top);
                        if (val != -1)
                        printf("\n The value at the top of stack is: %d", val);
                        else
                        printf("\n STACK IS EMPTY");
                        break;
                case 4:
                        top = display(top);
                        break;
                }
        }while(option != 5);
        return 0;
}
struct stack *push(struct stack *top, int val)
{
        struct stack *ptr;
        ptr = (struct stack*)malloc(sizeof(struct stack));
        ptr -> data = val;
        if(top == NULL)
        {
                ptr -> next = NULL;
                top = ptr;
        }
        else
        {
                ptr -> next = top;
                top = ptr;
        }
        return top;
}
struct stack *display(struct stack *top)
{
        struct stack *ptr;
        ptr = top;
        if(top == NULL)
        printf("\n STACK IS EMPTY");
        else
        {
```

```
                    while(ptr != NULL)
                    {
                            printf("\n %d", ptr -> data);
                            ptr = ptr -> next;
                    }
            }
            return top;
    }
    struct stack *pop(struct stack *top)
    {
            struct stack *ptr;
            ptr = top;
            if(top == NULL)
            printf("\n STACK UNDERFLOW");
            else
            {
                    top = top -> next;
                    printf("\n The value being deleted is: %d", ptr -> data);
                    free(ptr);
            }
            return top;
    }
    int peek(struct stack *top)
    {
            if(top==NULL)
            return -1;
            else
            return top ->data;
    }
```

Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. Peek
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 100
```

7.6 MULTIPLE STACKS

While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent `OVERFLOW` conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated.

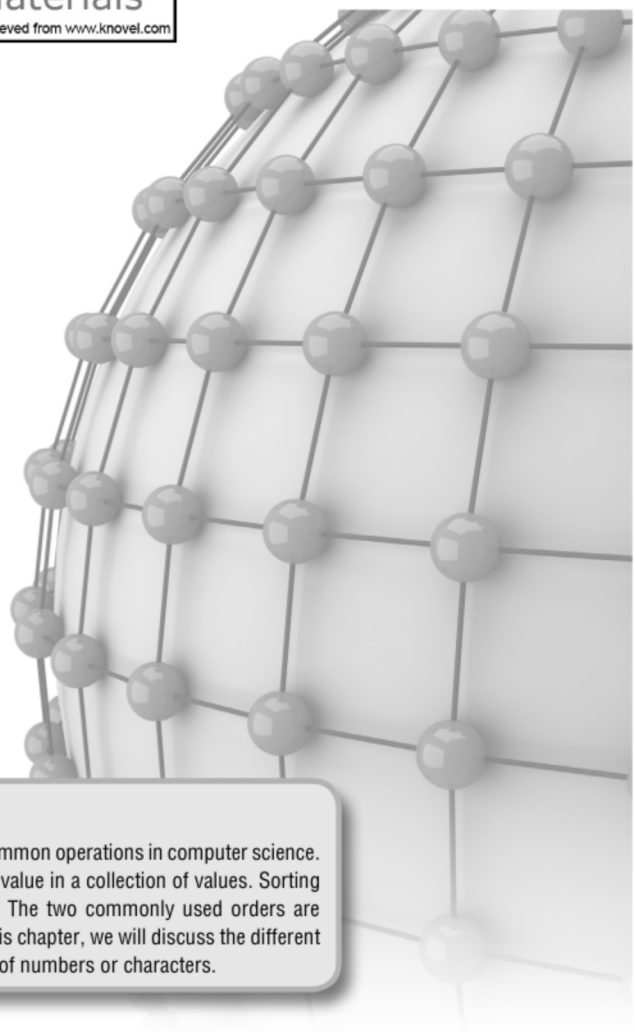
So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size. Figure 7.20 illustrates this concept.

Figure 7.20 Multiple stacks

In Fig. 7.20, an array `STACK[n]` is used to represent two stacks, `Stack A` and `Stack B`. The value of `n` is such that the combined size of both the stacks will never exceed `n`. While operating on

CHAPTER 14

Searching and Sorting



LEARNING OBJECTIVE

Searching and sorting are two of the most common operations in computer science. Searching refers to finding the position of a value in a collection of values. Sorting refers to arranging data in a certain order. The two commonly used orders are numerical order and alphabetical order. In this chapter, we will discuss the different techniques of searching and sorting arrays of numbers or characters.

14.1 INTRODUCTION TO SEARCHING

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
There are two popular methods for searching the array elements: *linear search* and *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity. We will discuss all these methods in detail in this section.

14.2 LINEAR SEARCH

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array `A[]` is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Figure 14.1 shows the algorithm for linear search.

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

Programming Example

1. Write a program to search an element in an array using the linear search technique.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, found = 0, pos = -1;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number that has to be searched : ");
        scanf("%d", &num);
        for(i=0;i<n;i++)
        {
                if(arr[i] == num)
                {
                        found =1;
                        pos=i;
                        printf("\n %d is found in the array at position= %d", num,i+1);
                        /* +1 added in line 23 so that it would display the number in
              the first place in the array as in position 1 instead of 0 */
                        break;
                }
        }
        if (found == 0)
```

```
            printf("\n %d does not exist in the array", num);
            return 0;
    }
```

14.3 BINARY SEARCH

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array. Consider an array `A[]` that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is `VAL = 9`. The algorithm will proceed in the following manner.

```
BEG = 0, END = 10, MID = (0 + 10)/2 = 5
```

Now, `VAL = 9` and `A[MID] = A[5] = 5`

`A[5]` is less than `VAL`, therefore, we now search for the value in the second half of the array. So, we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8`

```
VAL = 9 and A[MID] = A[8] = 8
```

`A[8]` is less than `VAL`, therefore, we now search for the value in the second half of the segment. So, again we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9`

Now, `VAL = 9` and `A[MID] = 9`.

In this algorithm, we see that `BEG` and `END` are the beginning and ending positions of the segment that we are looking to search for the element. `MID` is calculated as `(BEG + END)/2`. Initially, `BEG = lower_bound` and `END = upper_bound`. The algorithm will terminate when `A[MID] = VAL`. When the algorithm ends, we will set `POS = MID`. `POS` is the position at which the value is present in the array.

However, if `VAL` is not equal to `A[MID]`, then the values of `BEG`, `END`, and `MID` will be changed depending on whether `VAL` is smaller or greater than `A[MID]`.

(a) If `VAL < A[MID]`, then `VAL` will be present in the left segment of the array. So, the value of `END` will be changed as `END = MID - 1`.

(b) If `VAL > A[MID]`, then `VAL` will be present in the right segment of the array. So, the value of `BEG` will be changed as `BEG = MID + 1`.

Finally, if `VAL` is not present in the array, then eventually, `END` will be less than `BEG`. When this happens, the algorithm will terminate and the search will be unsuccessful.

Figure 14.2 shows the algorithm for binary search.

In Step 1, we initialize the value of variables, `BEG`, `END`, and `POS`. In Step 2, a `while` loop is executed until `BEG` is less than or equal to `END`. In Step 3, the value of `MID` is calculated. In Step 4, we check if the array value at `MID` is equal to `VAL` (item to be searched in the array). If a match is

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.2 Algorithm for binary search

found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered. In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Complexity of Binary Search Algorithm

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$2^{f(n)} > n$ or $f(n) = \log_2 n$

Programming Example

2. Write a program to search an element in an array using binary search.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10 // Added to make changing size of array easier
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, beg, end, mid, found=0;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);  // Added to sort the array
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        printf("\n\n Enter the number that has to be searched: ");
        scanf("%d", &num);
        beg = 0, end = n-1;
        while(beg<=end)
        {
                mid = (beg + end)/2;
                if (arr[mid] == num)
                {
                        printf("\n %d is present in the array at position %d", num, mid+1);
                        found =1;
                        break;
                }
```

```
                    else if (arr[mid]>num)
                    end = mid-1;
                    else
                    beg = mid+1;
            }
            if (beg > end && found == 0)
            printf("\n %d does not exist in the array", num);
            return 0;
    }

    int smallest(int arr[], int k, int n)
    {
            int pos = k, small=arr[k], i;
            for(i=k+1;i<n;i++)
            {
                    if(arr[i]< small)
                    {
                            small = arr[i];
                            pos = i;
                    }
            }
            return pos;
    }
    void selection_sort(int arr[],int n)
    {
            int k, pos, temp;
            for(k=0;k<n;k++)
            {
                    pos = smallest(arr, k, n);
                    temp = arr[k];
                    arr[k] = arr[pos];
                    arr[pos] = temp;
            }
    }
```

14.4 INTERPOLATION SEARCH

Interpolation search, also known as extrapolation search, is a searching technique that finds a specified value in a sorted array. The concept of interpolation search is similar to how we search for names in a telephone book or for keys by which a book's entries are ordered. For example, when looking for a name "Bharat" in a telephone directory, we know that it will be near the extreme left, so applying a binary search technique by dividing the list in two halves each time is not a good idea. We must start scanning the extreme left in the first pass itself.

In each step of interpolation search, the remaining search space for the value to be found is calculated. The calculation is done based on the values at the bounds of the search space and the value to be searched. The value found at this estimated position is then compared with the value being searched for. If the two values are equal, then the search is complete.

```
INTERPOLATION_SEARCH (A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET LOW = lower_bound,
        HIGH = upper_bound, POS = -1
Step 2:    Repeat Steps 3 to 4 while LOW <= HIGH
Step 3:         SET MID = LOW + (HIGH - LOW) ×
                ((VAL - A[LOW]) / (A[HIGH] - A[LOW]))
Step 4:         IF VAL = A[MID]
                  POS = MID
                  PRINT POS
                  Go to Step 6
                 ELSE IF VAL < A[MID]
                   SET HIGH = MID - 1
                  ELSE
                   SET LOW = MID + 1
                [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.3 Algorithm for interpolation search

However, in case the values are not equal then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position. Thus, we see that interpolation search is similar to the binary search technique. However, the important difference between the two techniques is that binary search always selects the middle value of the remaining search space. It discards half of the values based on the comparison between the value found at the estimated position and the value to be searched. But in interpolation search, interpolation is used to find an item near the one being searched for, and then linear search is used to find the exact item.

The algorithm for interpolation search is given in Fig. 14.3.

Figure 14.4 helps us to visualize how the search space is divided in case of binary search and interpolation search.

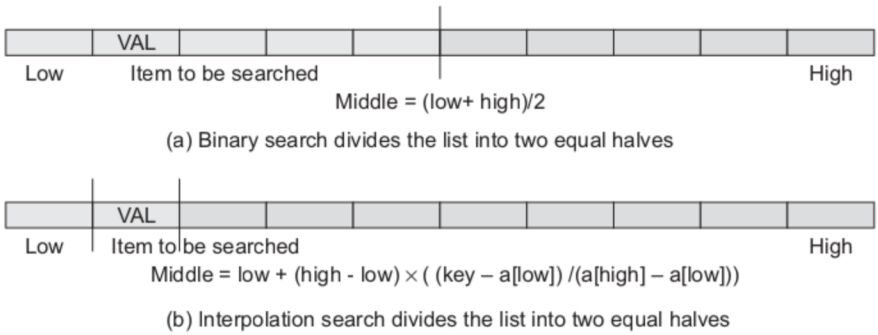


Figure 14.4 Difference between binary search and interpolation search

Complexity of Interpolation Search Algorithm

When `n` elements of a list to be sorted are uniformly distributed (average case), interpolation search makes about `log(log n)` comparisons. However, in the worst case, that is when the elements increase exponentially, the algorithm can make up to `O(n)` comparisons.

Example 14.1 Given a list of numbers `a[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}`. Search for value 19 using interpolation search technique.

Solution

```
Low = 0, High = 10, VAL = 19, a[Low] = 1, a[High] = 21
Middle = Low + (High – Low)×((VAL – a[Low]) /(a[High] – a[Low] ))
       = 0 +(10 – 0) × ((19 – 1) / (21 – 1) )
       = 0 + 10 × 0.9 = 9
a[middle] = a[9] = 19 which is equal to value to be searched.
```

PROGRAMMING EXAMPLE

3. Write a program to search an element in an array using interpolation search.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int interpolation_search(int a[], int low, int high, int val)
{
        int mid;
        while(low <= high)
        {
                mid = low + (high – low)*((val – a[low]) / (a[high] – a[low]));
                if(val == a[mid])
                     return mid;
                if(val < a[mid])
```

```
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return -1;
}
int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = interpolation_search(arr, 0, n-1, val);
        if(pos == -1)
                printf("\n %d is not found in the array", val);
        else
                printf("\n %d is found at position %d", val, pos);
        getche();
        return 0;
}
```

14.5 JUMP SEARCH

When we have an already sorted list, then the other efficient algorithm to search for a value is jump search or block search. In jump search, it is not necessary to scan all the elements in the list to find the desired value. We just check an element and if it is less than the desired value, then some of the elements following it are skipped by jumping ahead. After moving a little forward again, the element is checked. If the checked element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element. However, if the checked element is less than the value being searched for, then we again make a small jump and repeat the process.

Once the boundary of the value is determined, a linear search is done to find the value and its position in the array. For example, consider an array `a[] = {1,2,3,4,5,6,7,8,9}`. The length of the array is 9. If we have to find value 8 then following steps are performed using the jump search technique.

Step 1: First three elements are checked. Since 3 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 2: Next three elements are checked. Since 6 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 3: Next three elements are checked. Since 9 is greater than 8, the desired value lies within the current boundary

1	2	3	4	5	6	7	8	9

Step 4: A linear search is now done to find the value in the array.

The algorithm for jump search is given in Fig. 14.5.

```
JUMP_SEARCH (A, lower_bound, upper_bound, VAL, N)

Step 1: [INITIALIZE] SET STEP = sqrt(N), I = 0, LOW = lower_bound, HIGH = upper_bound, POS = -1
Step 2: Repeat Step 3 while I < STEP
Step 3:         IF VAL < A[STEP]
                  SET HIGH = STEP - 1
                ELSE
                  SET LOW = STEP + 1
                [END OF IF]
                SET I = I + 1
          [END OF LOOP]
Step 4: SET I = LOW
Step 5: Repeat Step 6 while I <= HIGH
Step 6:          IF A[I] = Val
                        POS =  I
                        PRINT POS
                        Go to Step 8
                 [END OF IF]
                 SET I = I + 1
          [END OF LOOP]
Step 7:   IF POS = -1
                PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
          [END OF IF]
Step 8: EXIT
```

Figure 14.5 Algorithm for jump search

Advantage of Jump Search over Linear Search

Suppose we have a sorted list of 1000 elements where the elements have values `0, 1, 2, 3, 4, …, 999`, then sequential search will find the value 674 in exactly 674 iterations. But with jump search, the same value can be found in 44 iterations. Hence, jump search performs far better than a linear search on a sorted list of elements.

Advantage of Jump Search over Binary Search

No doubt, binary search is very easy to implement and has a complexity of `O(log n)`, but in case of a list having very large number of elements, jumping to the middle of the list to make comparisons is not a good idea because if the value being searched is at the beginning of the list then one (or even more) large step(s) in the backward direction would have to be taken. In such cases, jump search performs better as we have to move little backward that too only once. Hence, when jumping back is slower than jumping forward, the jump search algorithm always performs better.

How to Choose the Step Length?

For the jump search algorithm to work efficiently, we must define a fixed size for the step. If the step size is 1, then algorithm is same as linear search. Now, in order to find an appropriate step size, we must first try to figure out the relation between the size of the list (`n`) and the size of the step (`k`). Usually, `k` is calculated as $\sqrt{n}$.

Further Optimization of Jump Search

Till now, we were dealing with lists having small number of elements. But in real-world applications, lists can be very large. In such large lists searching the value from the beginning of the list may not be a good idea. A better option is to start the search from the `k`-th element as shown in the figure below.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ....

Searching can start from somewhere middle in the list rather than from the beginning to optimize performance.

We can also improve the performance of jump search algorithm by repeatedly applying jump search. For example, if the size of the list is 10,00,000 (n). The jump interval would then be $\sqrt{n} = \sqrt{1000000} = 1000$. Now, even the identified interval has 1000 elements and is again a large list. So, jump search can be applied again with a new step size of $\sqrt{1000} \approx 31$. Thus, every time we have a desired interval with a large number of values, the jump search algorithm can be applied again but with a smaller step. However, in this case, the complexity of the algorithm will no longer be $O(\sqrt{n})$ but will approach a logarithmic value.

Complexity of Jump Search Algorithm

Jump search works by jumping through the array with a step size (optimally chosen to be $\sqrt{n}$) to find the interval of the value. Once this interval is identified, the value is searched using the linear search technique. Therefore, the complexity of the jump search algorithm can be given as $O(\sqrt{n})$.

Programming Example

4. Write a program to search an element in an array using jump search.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#define MAX 20
int jump_search(int a[], int low, int high, int val, int n)
{
        int step, i;
        step = sqrt(n);
        for(i=0;i<step;i++)
        {
                if(val < a[step])
                        high = step - 1;
                else
                        low = step + 1;
        }
        for(i=low;i<=high;i++)
        {
             if(a[i]==val)
                  return i;
        }
        return -1;
}

int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = jump_search(arr, 0, n-1, val, n);
```

```
        if(pos == -1)
                printf("\n %d is not found in the array", val);
        else
                printf("\n %d is found at position %d", val, pos);
        getche();
        return 0;
}
```

Fibonacci Search

We are all well aware of the Fibonacci series in which the first two terms are 0 and 1 and then each successive term is the sum of previous two terms. In the Fibonacci series given below, each number is called a Fibonacci number.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The same series and concept can be used to search for a given value in a list of numbers. Such a search algorithm which is based on Fibonacci numbers is called Fibonacci search and was developed by Kiefer in 1953. The search follows a divide-and-conquer technique and narrows down possible locations with the help of Fibonacci numbers.

Fibonacci search is similar to binary search. It also works on a sorted list and has a run time complexity of `O(log n)`. However, unlike the binary search algorithm, Fibonacci search does not divide the list into two equal halves rather it subtracts a Fibonacci number from the index to reduce the size of the list to be searched. So, the key advantage of Fibonacci search over binary search is that comparison dispersion is low.

14.6 INTRODUCTION TO SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if `A` is an array, then the elements of `A` are arranged in a sorted order (ascending order) in such a way that `A[0] < A[1] < A[2] < ..... < A[N]`.

For example, if we have an array that is declared and initialized as

```
int A[] = {21, 34, 11, 9, 1, 0, 22};
```

Then the sorted array (ascending order) can be given as:

```
A[] = {0, 1, 9, 11, 21, 22, 34;
```

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types of sorting:

- ***Internal sorting*** which deals with sorting the data stored in the computer's memory
- ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

14.6.1 Sorting on Multiple Keys

Many a times, when performing real-world applications, it is desired to sort arrays of records using multiple keys. This situation usually occurs when a single key is not sufficient to uniquely identify a record. For example, in a big organization we may want to sort a list of employees on the basis of their departments first and then according to their names in alphabetical order.

Other examples of sorting on multiple keys can be

- Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order.

- In a library, the information about books can be sorted alphabetically based on titles and then by authors' names.
- Customers' addresses can be sorted based on the name of the city and then the street.

Note Data records can be sorted based on a property. Such a component or property is called a **sort key**. A sort key can be defined using two or more sort keys. In such a case, the first key is called the **primary sort key**, the second is known as the **secondary sort key**, etc.

Consider the data records given below:

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

Now if we take department as the primary key and name as the secondary key, then the sorted order of records can be given as:

Name	Department	Salary	Phone Number
Divya	Computer Science	750000	9350123455
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Janak	Telecommunications	1000000	9812345678

Observe that the records are sorted based on department. However, within each department the records are sorted alphabetically based on the names of the employees.

14.6.2 Practical Considerations for Internal Sorting

As mentioned above, records can be sorted either in ascending or descending order based on a field often called as the sort key. The list of records can be either stored in a contiguous and randomly accessible data structure (array) or may be stored in a dispersed and only sequentially accessible data structure like a linked list. But irrespective of the underlying data structure used to store the records, the logic to sort the records will be same and only the implementation details will differ.

When analysing the performance of different sorting algorithms, the practical considerations would be the following:

- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst case performance
- Average case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

14.7 BUBBLE SORT

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements

in ascending order). In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Technique

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-2]` is compared with `A[N-1]`. Pass 1 involves `n-1` comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-3]` is compared with `A[N-2]`. Pass 2 involves `n-2` comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-4]` is compared with `A[N-3]`. Pass 3 involves `n-3` comparisons and places the third biggest element at the third highest index of the array.

(d) In Pass `n-1`, `A[0]` and `A[1]` are compared so that `A[0]<A[1]`. After this step, all the elements of the array are arranged in ascending order.

Example 14.2 To discuss bubble sort in detail, let us consider an array `A[]` that has the following elements:

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
 (a) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (b) Compare 52 and 29. Since 52 > 29, swapping is done.
     30, 29, 52, 87, 63, 27, 19, 54
 (c) Compare 52 and 87. Since 52 < 87, no swapping is done.
 (d) Compare 87 and 63. Since 87 > 63, swapping is done.
     30, 29, 52, 63, 87, 27, 19, 54
 (e) Compare 87 and 27. Since 87 > 27, swapping is done.
     30, 29, 52, 63, 27, 87, 19, 54
 (f) Compare 87 and 19. Since 87 > 19, swapping is done.
     30, 29, 52, 63, 27, 19, 87, 54
 (g) Compare 87 and 54. Since 87 > 54, swapping is done.
     30, 29, 52, 63, 27, 19, 54, 87
```

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

```
Pass 2:
 (a) Compare 30 and 29. Since 30 > 29, swapping is done.
     29, 30, 52, 63, 27, 19, 54, 87
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 63. Since 52 < 63, no swapping is done.
 (d) Compare 63 and 27. Since 63 > 27, swapping is done.
     29, 30, 52, 27, 63, 19, 54, 87
 (e) Compare 63 and 19. Since 63 > 19, swapping is done.
```

```
    29, 30, 52, 27, 19, 63, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
    29, 30, 52, 27, 19, 54, 63, 87
```

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

```
Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
    29, 30, 27, 52, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
    29, 30, 27, 19, 52, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.
```

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

```
Pass 4:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
    29, 27, 30, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
    29, 27, 19, 30, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.
```

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

```
Pass 5:
(a) Compare 29 and 27. Since 29 > 27, swapping is done.
    27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
    27, 19, 29, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.
```

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

```
Pass 6:
(a) Compare 27 and 19. Since 27 > 19, swapping is done.
    19, 27, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.
```

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

```
Pass 7:
(a) Compare 19 and 27. Since 19 < 27, no swapping is done.
```

Observe that the entire list is sorted now.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Figure 14.6 Algorithm for bubble sort

Figure 14.6 shows the algorithm for bubble sort. In this algorithm, the outer loop is for the total number of passes which is `N-1`. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed `N-I` times, where `N` is the number of elements in the array and `I` is the count of the pass.

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N-1 passes in total. In the first pass, N-1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N-2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1$
$f(n) = n\ (n - 1)/2$
$f(n) = n^2/2 + O(n) = O(n^2)$

Therefore, the complexity of bubble sort algorithm is $O(n^2)$. It means the time required to execute bubble sort is proportional to n^2, where n is the total number of elements in the array.

PROGRAMMING EXAMPLE

5. Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, temp, j, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr [i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                        }
                }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
                printf("%d\t", arr[i]);
        getch();
        return 0;
}
Output
Enter the number of elements in the array : 10
Enter the elements : 8    9    6    7    5    4    2    3    1    10
The array sorted in ascending order is :
1    2    3    4    5    6    7    8    9    10
```

Bubble Sort Optimization

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n-1 passes. We may even have an array that will be sorted in 2 or 3

passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to TRUE before each pass and is made FALSE when a swapping is performed. The code for the optimized bubble sort can be given as:

```
void bubble_sort(int *arr, int n)
{
     int i, j, temp, flag = 0;
     for(i=0; i<n; i++)
    {
       for(j=0; j<n-i-1; j++)
       {
          if(arr[j]>arr[j+1])
          {
             flag = 1;
             temp = arr[j+1];
             arr[j+1] = arr[j];
             arr[j] = temp;
          }
       }
      if(flag == 0)   // array is sorted
        return;
   }
}
```

Complexity of Optimized Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take $O(n)$ time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes $O(n^2)$ in all the cases.

14.8 INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

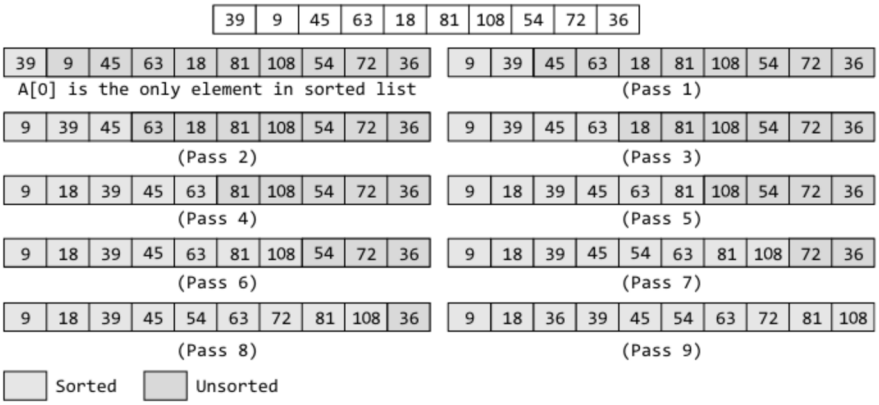
Technique

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming $LB = 0$) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if $LB = 0$).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example 14.3 Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution



Initially, A[0] is the only element in the sorted set. In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted. In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1]. In Pass 3, A[3] will be placed in its proper place. In Pass N-1, A[N-1] will be placed in its proper place to keep the array sorted.

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K-1], we need to compare A[K] with A[K-1], then with A[K-2], A[K-3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K-1], A[K-2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location. The algorithm for insertion sort is given in Fig. 14.7.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:     SET TEMP = ARR[K]
Step 3:     SET J = K - 1
Step 4:     Repeat while TEMP <= ARR[J]
                SET ARR[J + 1] = ARR[J]
                SET J = J - 1
            [END OF INNER LOOP]
Step 5:     SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Figure 14.7 Algorithm for insertion sort

In the algorithm, Step 1 executes a for loop which will be repeated for each element in the array. In Step 2, we store the value of the Kth element in TEMP. In Step 3, we set the Jth index in the array. In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the (J+1)th location.

Complexity of Insertion Sort

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n^2)).

Even in the average case, the insertion sort algorithm will have to make at least (K-1)/2 comparisons. Thus, the average case also has a quadratic running time.

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only $O(1)$ of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

PROGRAMMING EXAMPLE

6. Write a program to sort an array using insertion sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        insertion_sort(arr, n);
        printf("\n The sorted array is:  \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void insertion_sort(int arr[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
                temp = arr[i];
                j = i-1;
                while((temp < arr[j]) && (j>=0))
                {
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = temp;
        }
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 500 1 50 23 76
The sorted array is :
1    23    20    76    500    6    7    8    9    10
```

14.9 SELECTION SORT

Selection sort is a sorting algorithm that has a quadratic running time complexity of $O(n^2)$, thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over

more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique

Consider an array ARR with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1]. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], . . . , ARR[N-1] is sorted.

Example 14.4 Sort the array given below using selection sort.

	39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

The algorithm for selection sort is shown in Fig. 14.8. In the algorithm, during the K^{th} pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], . . . , ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

```
SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Figure 14.8 Algorithm for selection sort

Complexity of Selection Sort

Selection sort is a sorting algorithm that is independent of the original order of elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements;

thus, n-1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n - 1 elements and so on. Therefore,

```
(n - 1) + (n - 2) + ... + 2 + 1
= n(n - 1) / 2 = O(n²) comparisons
```

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

PROGRAMMING EXAMPLE

7. Write a program to sort an array using selection sort algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int smallest(int arr[], int k, int n);
void selection_sort(int arr[], int n);
void main(int argc, char *argv[]) {
        int arr[10], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
}
int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}
void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];
```

```
                    arr[k] = arr[pos];
                    arr[pos] = temp;
                }
        }
```

14.10 MERGE SORT

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the `n`-element array to be sorted into two sub-arrays of `n/2` elements. If `A` is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide `A` into two sub-arrays, A_1 and A_2, each containing about half of the elements of `A`.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size `n/2` to produce the sorted array of `n` elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

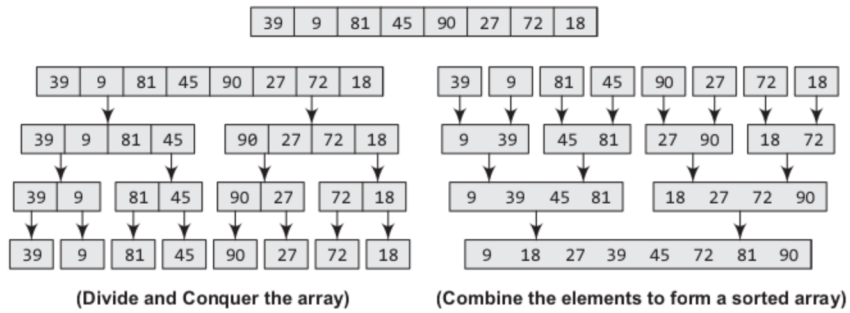
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

Example 14.5 Sort the array given below using merge sort.

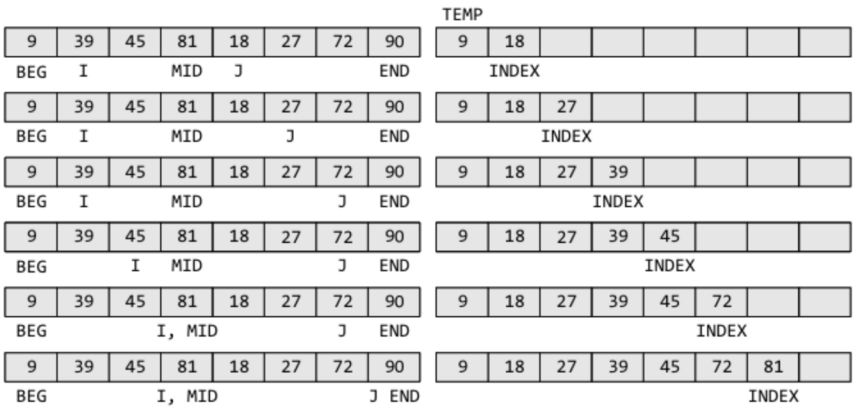
Solution



The merge sort algorithm (Fig. 14.9) uses a function `merge` which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.

Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.



When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.


```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = INDEX + 1, SET I = I + 1
               [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

Figure 14.9 Algorithm for merge sort

The running time of merge sort in the average case and the worst case can be given as O(n log n). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Programming Example

8. Write a program to implement merge sort.

```
#include <stdio.h>
#include <conio.h>
#define size 100

void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        merge_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void merge(int arr[], int beg, int mid, int end)
{
        int i=beg, j=mid+1, index=beg, temp[size], k;
        while((i<=mid) && (j<=end))
        {
                if(arr[i] < arr[j])
                {
                        temp[index] = arr[i];
                        i++;
                }
                else
                {
                        temp[index] = arr[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=end)
                {
```

```
                        temp[index] = arr[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = arr[i];
                        i++;
                        index++;
                }
        }
        for(k=beg;k<index;k++)
        arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
                mid = (beg+end)/2;
                merge_sort(arr, beg, mid);
                merge_sort(arr, mid+1, end);
                merge(arr, beg, mid, end);
        }
}
```

14.11 QUICK SORT

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes $O(n \log n)$ comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as $O(n^2)$. Basically, the quick sort algorithm is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

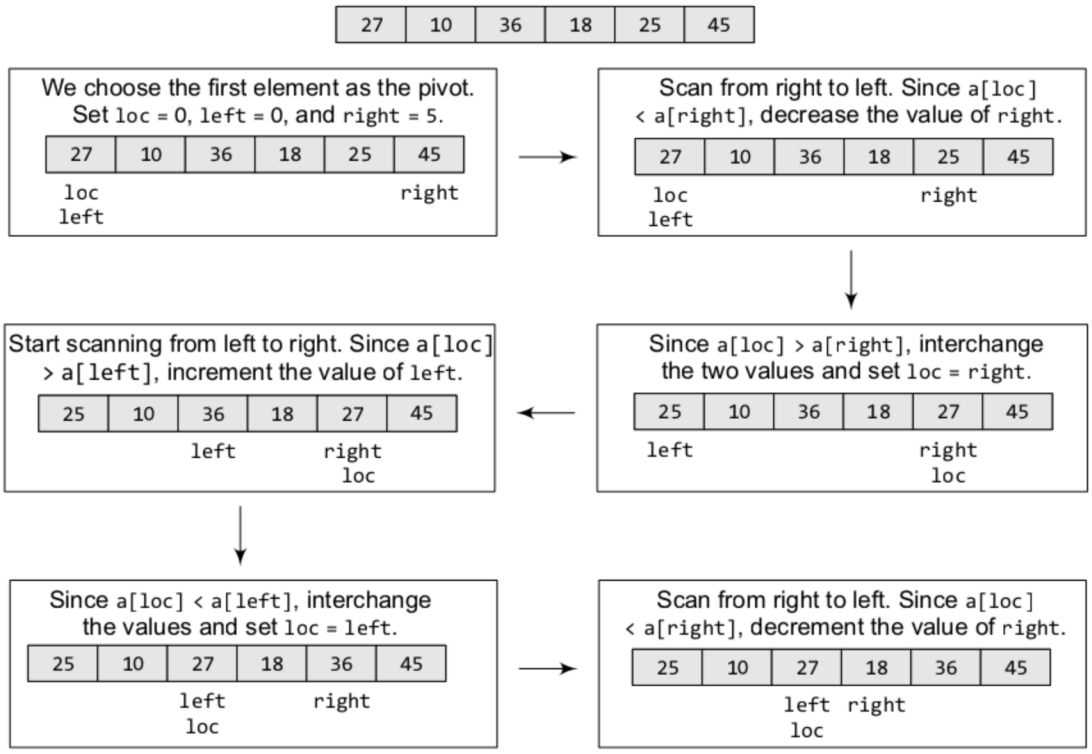
Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

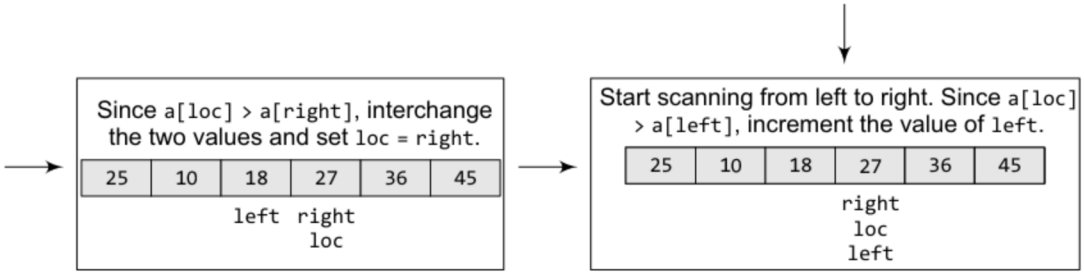
Technique

Quick sort works as follows:

1. Set the index of the first element in the array to `loc` and `left` variables. Also, set the index of the last element of the array to the `right` variable.
   That is, `loc = 0`, `left = 0`, and `right = n-1` (where `n` in the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable `loc`.
   That is, `a[loc]` should be less than `a[right]`.
   (a) If that is the case, then simply continue comparing until `right` becomes equal to `loc`. Once `right = loc`, it means the pivot has been placed in its correct position.
   (b) However, if at any point, we have `a[loc] > a[right]`, then interchange the two values and jump to Step 3.
   (c) Set `loc = right`
3. Start from the element pointed by `left` and scan the array from left to right, comparing each element on the way with the element pointed by `loc`.
   That is, `a[loc]` should be greater than `a[left]`.
   (a) If that is the case, then simply continue comparing until `left` becomes equal to `loc`. Once `left = loc`, it means the pivot has been placed in its correct position.
   (b) However, if at any point, we have `a[loc] < a[left]`, then interchange the two values and jump to Step 2.
   (c) Set `loc = left`.

Example 14.6 Sort the elements given in the following array using quick sort algorithm





Now `left = loc`, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

The quick sort algorithm (Fig. 14.10) makes use of a function `Partition` to divide the array into two sub-arrays.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Figure 14.10 Algorithm for quick sort

Complexity of Quick Sort

In the average case, the running time of quick sort can be given as $O(n \log n)$. The partitioning of the array which simply loops over the elements of the array once uses $O(n)$ time.

In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only $\log n$ nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is $O(\log n)$. And because at each level, there can only be $O(n)$, the resultant time is given as $O(n \log n)$ time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.

However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of $O(n \log n)$.

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

PROGRAMMING EXAMPLE

9. Write a program to implement quick sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                right--;
```

```
                    if(loc==right)
                    flag =1;
                    else if(a[loc]>a[right])
                    {
                            temp = a[loc];
                            a[loc] = a[right];
                            a[right] = temp;
                            loc = right;
                    }
                    if(flag!=1)
                    {
                            while((a[loc] >= a[left]) && (loc!=left))
                            left++;
                            if(loc==left)
                            flag =1;
                            else if(a[loc] <a[left])
                            {
                                    temp = a[loc];
                                    a[loc] = a[left];
                                    a[left] = temp;
                                    loc = left;
                            }
                    }
            }
            return loc;
    }
    void quick_sort(int a[], int beg, int end)
    {
            int loc;
            if(beg<end)
            {
                    loc = partition(a, beg, end);
                    quick_sort(a, beg, loc-1);
                    quick_sort(a, loc+1, end);
            }
    }
```

14.12 RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.