Block 3

DATA STRUCTURES

UNIT 11
Introduction to Data Structures: Array 5

UNIT 12
Lists 21

UNIT 13
Stacks and Queues 43

UNIT 14
Trees 59

UNIT 15
Files 87
Curriculum Design Committee

Dr. B.D. Acharya  
Dept. of Science & Technology  
New Delhi

Prof. Adimurthi  
School of Mathematics  
TIFR, Bangalore

Prof. Archana Aggarwal  
CESP, School of Social Sciences  
JNU, New Delhi

Prof. R. B. Bapat  
Indian Statistical Institute, New Delhi

Prof. M.C. Bhandari  
Dept. of Mathematics  
IIT, Kanpur

Prof. R. Bhatia  
Indian Statistical Institute, New Delhi

Prof. A. D. Dharmadhikari  
Dept. of Statistics  
University of Pune

Prof. O.P. Gupta  
Dept. of Financial Studies  
University of Delhi

Prof. S.D. Joshi  
Dept. of Electrical Engineering  
IIT, Delhi

Dr. R. K. Khanna  
Scientific Analysis Group  
DRDO, Delhi

Prof. Susheel Kumar  
Dept. of Management Studies  
IIT, Delhi

Prof. Veni Madhavan  
Scientific Analysis Group  
DRDO, Delhi

Prof. J.C. Mishra  
Dept. of Mathematics  
IIT, Kharagpur

Prof. C. Musili  
Dept. of Mathematics and Statistics  
University of Hyderabad

Prof. Sankar Pal  
ISI, Kolkata

Prof. A.P. Singh  
PG Dept. of Mathematics  
University of Jammu

Faculty Members
School of Sciences, IGNOU

Dr. Deepika  
Prof. Poornima Mital  
Dr. Atul Razdan  
Prof. Parvin Sinclair  
Prof. Sujatha Varma  
Dr. S. Venkataraman

Course Design Committee

Prof. C.A. Murthy  
ISI, Kolkata

Prof. S.B. Pal  
IIT, Kharagpur

Dr. B.S. Panda  
IIT, Delhi

Prof. C.E. Veni Madhavan  
IISC, Bangalore

Faculty Members
School of Sciences, IGNOU

Dr. Deepika  
Prof. Poornima Mital  
Dr. Atul Razdan  
Prof. Parvin Sinclair  
Dr. S. Venkataraman

Block Preparation Team

Dr. S. Venkataraman  
School of Sciences  
IGNOU

Units 11, 12 and 13 are modified versions of units 1, 2 and 3 of Block 4 of CS-04 and units 14 and 15 are modified versions of units 1 and 3 of Block 5 of CS-04, respectively.

January 2008
© Indira Gandhi National Open University, 2008
ISBN-978-81-266-3296-1

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means without written permission from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University’s office at Maidan Garhi, New Delhi-110 068.

Printed and Published on behalf of Indira Gandhi National Open University, New Delhi, by Director, School of Sciences.
In this Block, we will discuss data structures. Data structures provides methods for storing and working with complex data. Take for example, a program that plays chess. The program has to analyse the consequences of each move made. There can be several alternative responses to a move and the program has to analyse all these alternatives. The data types that we have already are not useful in this regard. We have to create new data structures to store and work with such data.

In this block, we will introduce you to some other data structures. In the first unit, we will introduce you to arrays. This is the simplest of data structures and it can be implemented using the standard arrays available in C.

In the second unit, we will discuss lists. They can be implemented using both arrays and linked lists. Linked lists are examples of dynamic data structures that can grow or shrink, depending on the amount of information it has to hold.

In the third unit, we will discuss stacks and queues. Stacks are data structures in which data is organised on First In Last Out (FIFO) principle. Queues are data structures that help us mimic the queues that we come across in real life as well as in applications of computers.

In the fourth unit, we will discuss trees. Trees are generally used to store ordered data that are not linearly ordered. For example, in computer games, when we want to analyse all the consequences of a particular move, we store all the possible outcomes in the form of a tree and analyse it. In the last unit, we will discuss files briefly.
## UNIT 11 INTRODUCTION TO DATA STRUCTURES: ARRAY

### Structure

<table>
<thead>
<tr>
<th>Section</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>Objectives</td>
<td></td>
</tr>
<tr>
<td>11.2 Program Analysis</td>
<td>5</td>
</tr>
<tr>
<td>11.3 Arrays as Data Structures</td>
<td>8</td>
</tr>
<tr>
<td>11.4 Creation of Arrays and Elementary Operations</td>
<td>9</td>
</tr>
<tr>
<td>11.5 Storage of Arrays in Main Memory</td>
<td>14</td>
</tr>
<tr>
<td>11.6 Sparse Arrays</td>
<td>17</td>
</tr>
<tr>
<td>11.7 Summary</td>
<td>18</td>
</tr>
<tr>
<td>11.8 Solutions/Answers</td>
<td>19</td>
</tr>
</tbody>
</table>

### 11.1 INTRODUCTION

This Unit is the introductory unit on data structures. In Sec. 11.2, we introduce you to program analysis and the concept of computational complexity. Since these topics are studied in the Design and analysis of algorithms course we will not go into great details. In Sec. 11.3, we will start our study of data structures with the simplest data structure, namely Array. You have already seen arrays in Unit 6 and Unit 7 of Block 2. In this unit, we are going to look at this array as a data structure. In Sec. 11.4 of this Unit, we will see how to create arrays and perform some elementary operations on them. In Sec 11.5, we will discuss different ways of storing data in an array like row major, column major methods. In the last section, Sec 11.6, we will discuss sparse arrays which are large arrays in which most entries are the same, usually 0.

### Objectives

After studying this unit, you should be able to
- explain the benefits of program analysis;
- define a data structure;
- perform basic operations on arrays; and
- explain how data is stored in sparse arrays.

### 11.2 PROGRAM ANALYSIS

In this section, we introduce you to *program analysis*. What do we mean by this? After all, there are many ways of analysing a program. For instance, we can analyse a program from any of the following points of view.

i) Verifying that it satisfies the requirements

ii) Proving that it runs correctly without any logical errors.

iii) Determining if it is readable.

iv) Checking that modifications can be made easily, without introducing new errors.

v) We may also analyse program execution time and the storage complexity associated with it, i.e. how fast does the program run and how much storage it requires.
Another related question can be: How big must its data structure be and how many steps will be required to execute its algorithm?

Since this course concerns data representation and writing programs, we shall analyse programs in terms of storage and time complexity.

Performance Issues

In considering the performance of a program, we are primarily interested in

i) how fast does it run?

ii) how much storage does it use?

Generally we need to analyse efficiencies, when we need to compare alternative algorithms and data representations for the same problem or when we deal with very large programs.

We often find that we can trade time efficiency for space efficiency, or vice-versa. For finding any of these i.e. time or space efficiency, we need to have some estimate of the problem size. Let’s assume that some number \( N \) represents the size of the problem. The size of the problem or \( N \) can reflect one or more features of the problem, for instance \( N \) might be the number of input data values, or it is the number of elements of an array etc. In the case of algorithms that take integers as input like the algorithm to find the hcf, it is the number of digits in the input. For example, \( N \) may be the number of names of persons that we want to sort in alphabetical order.

Let us consider an example here. Suppose that we are given two algorithms for finding the largest value in a list of \( N \) numbers. It is also given that second algorithm executes twice the number of instructions executed by the first algorithm for each \( N \) value.

Let first algorithm execute \( S \) number of instructions. Then second algorithm would execute \( 2S \) instructions. If each instruction takes 1 unit of time, say 1 millisecond then for \( N = 10, 100, 1000 \) and 10000 we shall have the number of operations and estimated execution time as given below:

<table>
<thead>
<tr>
<th>( N )</th>
<th>Algorithm I</th>
<th>Algorithm II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Instructions</td>
<td>Estimated Execution Time</td>
</tr>
<tr>
<td>10</td>
<td>10S</td>
<td>10 msec</td>
</tr>
<tr>
<td>100</td>
<td>100S</td>
<td>100 msec</td>
</tr>
<tr>
<td>1000</td>
<td>1000S</td>
<td>1000 msec</td>
</tr>
<tr>
<td>10000</td>
<td>10000S</td>
<td>10000 msec</td>
</tr>
</tbody>
</table>

1 millisecond = 1 msec = 1/1000 sec

You may notice that for larger values of \( N \), the difference between execution time of the two algorithm is appreciable, and one may clearly say that Algorithm II is slower than Algorithm I. Also, as the problem size becomes larger and larger, Algorithm I performs better and better than Algorithm II. This kind of performance improvement is termed as *order of improvements*. Two algorithm may compare with each other by a constant factor, i.e. improvement from one to another does not change as the problem size gets larger. For example, one of them can be two times faster than the other and it always remain two times better regardless of the problem size.

In both of the above algorithms, the time taken grows linearly with input size. Actually, instead of time, we usually talk in terms of the number of basic operations the algorithm has to perform as a function of the size of the input. For example, to add two \( n \) digit integers, the computer has to perform \( n \) operations. We have to add each digit in the first number to the corresponding digit in the second number. In this case, the number of operations grows linearly with the size of the input. On the other hand, the usual algorithm for multiplying two \( n \) digit numbers performs about \( n^2 \) operations. This is because, under the usual multiplication algorithm, we have to multiply each digit of the first number with the each digit of the second number. See **Fig. 1** for
the number of operations required to add and multiply 3 digit numbers.

\[
\begin{array}{ccc}
5 & 4 & 7 \\
3 & 3 & 7 \\
5 & 4 & 7 \\
3 & 3 & 7
\end{array}
\]

Fig. 1: Number of operations for adding and multiplying two 3 digit numbers.

Therefore, the number of operations required to multiply two n-digit numbers grow as the square of the size of the numbers. Such algorithms are called quadratic algorithms. What we are actually doing is to compare the growth rate of algorithms with known functions; in the above two cases we see that the growth is comparable to a linear polynomial and a quadratic polynomial, respectively.

There is a standard notation for such comparisons called the big-Oh notation. If \( f(n) \) and \( g(n) \) are two functions defined on natural numbers, we say that \( f(n) = O(g(n)) \) if \( |f(n)| \leq K|g(n)| \) where \( K \) is a constant independent of \( n \). If \( f(n) \) is the number of operations an algorithm has to perform on an input of size \( n \), if \( f(n) = O(n) \), we say that the algorithm is linear algorithm; if \( f(n) = O(n^2) \), we say that the algorithm is a quadratic algorithm. More generally, if \( f(n) = O(g(n)) \) for some polynomial \( g(n) \), we say that the algorithm is a polynomial time algorithm. We discuss the analysis (and design) of algorithms in the Design and Analysis of Algorithms course.

**Average Case and Worst Case Analysis**

An algorithm chooses an execution path depending on the set of data values (input). Therefore, an algorithm may perform differently for two different sets of data values. If we take a set of data values for which the algorithm takes the longest possible execution time, it leads us to the worst case execution time. On the other hand, an average case execution time is the execution time takes by algorithm for an expected range of data values. For analysis of an algorithm to predict its average or worst case execution time, we need to make certain assumption such as assuming that all operations take about the same amount of time.

For example, suppose we have \( n \) student cards with the enrolment numbers \( a_1, a_2, \ldots, a_n \) written on them and they are in a box. A number is called out and you have to take out the card with that number. Let us assume that the cards are arranged in ascending order, i.e. that \( a_1 \) is the topmost card, \( a_n \) is the card at the bottom and \( a_i < a_{i+1} \) for \( 1 \leq i \leq n-1 \). If you do not know this and you have to search through the cards and find the \( k \)th card, you have to check through \( k \) cards. So, the number of operations you need depends on the card you want to find. In the worst case, if you want to find the \( n \)th card, you have to search through \( n \) cards.

What is the average time you take to find any card? Suppose that all the cards are equally probable to be called. Then, the probability of any card being called out is \( \frac{1}{n} \) and the average number of cards you have to search through is \( \sum_{i=1}^{n} (i) \cdot \frac{1}{n} \) since the number of cards you have to search through to find the \( i \)th card is \( i \). The sum is

\[
\frac{1}{n} \sum_{i=1}^{n} i + \frac{2}{n} \sum_{i=1}^{n} i + \cdots + \frac{n}{n} \sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{n+1}{2}
\]

One issue we have to deal with when we want to optimise the performance is the representation of data. We have find a representation that optimises the time (or space) involved in working with the data. We use **Data structures** for this. We do not merely want to represent data, but also work with them. So, our data structures should allow us to perform the required operations in an efficient way. So, we can say that

\[
\text{Data Structures} = \text{Organised Data} + \text{Allowed operations}
\]
In the next section, we begin our study of data structures with arrays, the simplest of data structures.

11.3 ARRAYS AS DATA STRUCTURES

In applications where we have a small number of items to handle, we tend to specify separate variables names for each item. When we have to keep track of more pieces of related data, we need to organise data, in such a way that we can use one name to refer to several items. Let us see this through a simple example. Consider the following problem:

Read 25 numbers and print them in reverse order.

The problem requires all the numbers as they are read. Further we cannot print anything until all 25 numbers are read; therefore, we need to store all the twenty five numbers. Reading 25 numbers in 25 different variables will be quite cumbersome and so would be writing these numbers in reverse order. It is much simpler to call the numbers NUM\_1, NUM\_2, NUM\_3, \cdots, NUM\_25.

Each number is a NUM and numbers are distinguished by subscripts. Also, they are read in succession. Thus, we can abbreviate this sequence as NUM\_i for \(i = 1, 2, \cdots, 25\). Such a subscripted variable is called an array. More formally, an array is a finite, ordered set of homogeneous elements which are stored in adjacent cells in memory. Arrays are usually used when a program includes a list of recurring elements.

You are probably wondering ‘What is new about arrays? We have discussed them already in Block 2.’ Here, we are going to study array as an abstract object without reference to any language. The set of integers exists independently of any representation or implementation. In this set you can carry out basic operations like addition, subtraction, multiplication and division. The C data type `int` is a particular implementation of this. It allows you to carry out all the basic operations like addition, multiplication etc. This is not the full set of integers, but only integers between –32767 and 32767 according to C89 specification. Similarly, we can think of an abstract data type called array with a specified set of operations.

One characteristic feature of the array is that it takes the same amount of time to access any element in the array. As we have seen, in C, elements of an array can be accessed using subscripts placed in square brackets[]. Repetition over a sequence of values of \(i\) may also be implemented using a loop construct. For example, the following statement reads all 25 values:

```c
for (i = 0; i < 25; ++i)
    scanf("%d",NUM\[i\]);
```

A similar approach works out for printing the values.

The simplest form of an array is a one-dimensional array or vector. As stated earlier, the various elements of an array are distinguished by giving each piece of data separate index or subscript. The subscript of an element designates its position in array’s ordering. An array named \(A\) which consists of \(N\) elements can be depicted as shown in Fig. 2.

![Fig. 2: One dimensional array.](image)

Arrays can be multi-dimensional. Any array defined to have more than one dimension is considered to be multi-dimensional array. An array can be 2-dimensional, 3-dimensional, 4-dimensional, or N-dimensional although they rarely exceed three dimensions. Two-dimensional arrays, sometimes called matrices, are quite common. The best way to think...
about a two-dimensional array is to visualise a table of columns and rows: the first dimension in
the array refers to the rows, and the second dimension refers to the columns. Let us see an
example of a 2-dimensional array.

A collection of data about the grades of students in a class in the four different exams can be
represented using a 2-dimensional arrays. If we have 10 students and each given grades in 4
exams, we can depict it as in Fig. 3. Each cell in this table contains a grade value for the student

### Fig. 3: An example of a two dimensional array.

<table>
<thead>
<tr>
<th>Student number</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Number (given by the corresponding row number) and exam number (given by the
appropriate column no.). We may map it on to an array $A$ of order $10 \times 4$. $A[I][J]
represents an element of $A$, where $I$ runs from 1 to 10 and $J$ runs from 1 to 4. $A[3][4]$ will
have the grade value of 3rd student in fourth exam, $A[8][1]$ will have the grade value of 8th
student in first exam, and so on.

By convention, the first subscript of a 2-dimensional array refers to a row of the array, while the
second subscript refers to a column of the array.

In general, an array of the order $M \times N$ (read as $M$ by $N$) consists of $M$ rows, $N$ columns and
$MN$ elements. It may be depicted as in Fig. 4. Let us now discuss the syntax and semantics of an

### Fig. 4: A 2 dimensional array.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>......</th>
<th>......</th>
<th>N - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M - 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

array. We can divide our discussion in three parts:

- Array declaration
- Storage of Arrays in Main Memory
- Use of Arrays in Programs

In the next section, we will discuss creation of arrays and elementary operations that can be
performed on arrays.

### 11.4 CREATION OF ARRAYS AND ELEMENTARY OPERATIONS

Three things need to be specified to declare an array in most of the programming languages:
Data Structures

- the array name
- the type of data to be stored in array elements
- the subscript range

In C language the array declaration is as follows:

```c
int A[24];
float B[100][25];
```

In the first declaration A is the array name; the elements of A can hold integer data and the number of elements is 24 i.e. subscripts range from 0 to 23.

In the next declaration B is the array name; the data type of its elements is real and it is a 2-dimensional array with subscripts ranging from 0 to 99 and 0 to 24. In other languages, the lower limit on an array does not have to be 1. It makes more sense to start the array at a value that corresponds to the context of your data. Also, subscript need not always be positive in some languages. It can be negative or zero. However, not all programming languages allow zero or negative subscripts.

Be careful when using arrays with indexes beginning with 0. Failing to remember that the zero elements is the first item in the array — and therefore, the element at index 5 is the sixth, not the fifth — is a frequent cause of programming bugs.

You can begin your indexes at 0 or at 1 or at any other value, if programming language in use allows it. There is no technical reason to use one method over the other. However most programmers prefer to start arrays at 0 — even though it is easier to begin at 1. The reason for this is that some languages — C and C++ for example — require arrays to begin with zero indexes. If you define most of your arrays the same way, your programs will be easier to convert to these languages.

An array declaration tells the computer two major pieces of information about an array. First, the range of subscripts allow the computer to determine how many memory locations must be allocated. Second the array type tells the computer how much space is required to hold each value. Let us consider the following declarations:

```c
int A[10];
float B[10];
```

The first declaration tells the computer to allocate enough space for the variable A to store 10 integers. The second declaration tells the computer to allocate enough space for the variable B to store 10 reals. Since a real number takes more space than an integer the storage allocated would not be same. We have already discussed declaration of arrays in C in Units 12 and 13 of Block 2.

Operations on Arrays

The array is a **homogeneous structure**, i.e. the elements of an array are of the same type. It is **finite**; it has a specified number of elements. An array is **ordered**; there is an ordering of elements in it as zeroth, first, second etc. Following set of operations are defined for this structure.

i) Creating an array
ii) Initialising an array
iii) Storing an element
iv) Retrieving an element
v) Inserting an element
vi) Deleting an element
Let us now write a function that deletes an element from an array. The function in [11.1] takes the name of the array (which is actually a pointer to an `int`), the index of the element to be removed and the index of the last element as arguments. It removes the element and returns the index of the last element.

```c
int delete_element(int *list, int last_index, int index)
{
    int i;
    for (i = index; i < last_index; i++)
        list[i] = list[i+1];
    return (last_index-1);
}
```

Listing 11.1: A function to delete an element from an array.

Let us now see how to insert an element in an array. The function in [11.2] takes the name of the array, the element to be inserted, the index of the last element and the position where the element has to be inserted.

```c
int insert_element(int *list, int num, int last_index, int index)
{
    int i;
    if (last_index == max_array_size)
        Error("Array is full. Cannot insert element.");
    else {
        for (i = last_index-1; i > index; i--)
            list[i+1] = list[i];
        list[index] = num;
    }
    return (last_index+1);
}
```

Listing 11.2: A function to insert an element in an array.

Here is an exercise to check you understanding of our discussion so far.

**E1)** Write a program that
1) creates an array consisting of elements 3, 4, 5, 6 and 7;
2) removes 5 from the array;
3) inserts 10 after 7 in the array.

We will see how to sort an array of integers using **insertion sort**.

**Example 1:** Suppose we want to sort the list 21, 20, 19, 23, 16, 25. We do this in 5 passes. In the first pass, we look at 20 which is less than 21, yet it appears before 20. So, we exchange 20 and 21. In the second pass, we look at the third element in the list, which is 19. We see that the two elements before it, 21 and 20, are both bigger. So, we move 21 and 20 right by one position and insert in the position occupied by 21 previously. Now, the first three elements are in the correct order. We proceed like this to sort all the elements of the list. In general, in the i^{th} pass, insertion sort ensures that all the i + 1 elements from positions 0 through i are in sorted order. In the i^{th} pass, we move the (i + 1)^{th} element till its correct place. The function [11.3] on the next page finds the correct position of the (i + 1)^{th} element in the i^{th} pass:
Table 1: Interchanges in pass 2.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>19</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>23</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 2: Insertion sort.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>Pass 1</td>
<td>20</td>
<td>21</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>Pass 2</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>Pass 3</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>Pass 4</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>Pass 5</td>
<td>16</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

```c
int find_position(int *list, int pos)
{
    int j;
    for (j = pos; j > 0 && list[j - 1] > list[pos]; j--);
    return (j);
}
```

Listing 11.3: A function that finds the correct position of an element.

After that, we have to insert the \((i + 1)\)th element in the correct position. We can now insert the element in the correct position using the function we wrote for inserting an element in an array.

***

Here is an exercise for you to check your understanding of the previous example.

E2) Write a program that scans an array of integers, sorts them by insertion sort and prints the sorted array.

E3) We have used two different functions to find the correct position of an element and for inserting the element. Both can be performed in one go. Write an insertion sort function that does this.

These array operations apply to arrays of any dimension. We have already used these operations in our programme that lists primes. Here is a program that gives examples some of these operations.

```c
/*Program 11.3: 2-dimensional Array operations example. File name: unit11-matrix-ex.c*/
#include <stdio.h>
#define num_rows 4
#define num_columns 4
void printarray(int mymat[num_rows][num_columns]);
void findelement(int mymat[num_rows][num_columns], int x);
int main(void)
{
    /* Create and initialise an array of ints */
    int i, j, mat[num_rows][num_columns] = { {1, 0, 0, 0} };
    /*print the array */
    printf("First call to printarray ...\n");
    printarray(mat);
    /* Insert elements; Make it 4x4 diagonal matrix */
    for (i = 0; i < num_rows; i++)
        for (j = 0; j < num_columns; j++)
            if (i == j)
```
```c
mat[i][j] = 3;
else
    mat[i][j] = 0;
}
printf("\n\n"itoarray(mat);
for (i = 0; i < num_rows; i++)
    for (j = 0; j < num_columns; j++)
        mat[i][j] = i + j;
printf("\n\n"itoarray(mat);
printf("Third call to printarray...\n");
printarray(mat);
findelement(mat, 4);
return (0);
void printarray(int mat[num_rows][num_columns])
{
    int a, b;
    for (a = 0; a < num_rows; a++)
        for (b = 0; b < num_columns; b++)
            printf("mat[%d][%d]=%d, ", a, b, mat[a][b]);
}
void findelement(int mat[num_rows][num_columns], int x)
{
    int i, j, found;
    found = 0;
    printf("Searching for %d ...\n", x);
    for (i = 0; i < num_rows; j = 0, i++)
    {
        for (j = 0; j < num_rows; j++)
            if (mat[i][j] == x)
            {
                found = 1;
                break;
            }
    }
    if (found == 0)
        printf("Could not find %d", x);
    else
        printf("Found ! mat[%d][%d]=%d\n", i, j, x);
};
```

Here is the output from the program:

```
/*Output from Prog 1.1*/ First call to printarray ...
mat[0][0]=1, mat[0][1]=0, mat[0][2]=0, mat[0][3]=0, mat[1][0]=0, mat[1][1]=0,
mat[1][2]=0, mat[1][3]=0, mat[2][0]=0, mat[2][1]=0, mat[2][2]=0, mat[2][3]=0,
mat[3][0]=0, mat[3][1]=0, mat[3][2]=0, mat[3][3]=0,
Second call to printarray... mat[0][0]=3, mat[0][1]=0, mat[0][2]=0,
mat[0][3]=0, mat[1][0]=0, mat[1][1]=3, mat[1][2]=0, mat[1][3]=0,
mat[2][0]=0, mat[2][1]=0, mat[2][2]=3, mat[2][3]=0,
mat[3][0]=0, mat[3][1]=0,
```
mat[3][2]=0, mat[3][3]=3,
Third call to printarray... mat[0][0]=0, mat[0][1]=1,
mat[0][2]=2,
mat[0][3]=3, mat[1][0]=1, mat[1][1]=2, mat[1][2]=3,
mat[1][3]=4,
mat[3][0]=3, mat[3][1]=4,
mat[3][2]=5, mat[3][3]=6,
Searching for 4 ... Found ! mat[1][3]=4

In this program, line 10 declares and initialises an array of size num_rows × num_columns. Note that although the matrix has four rows, we have given only one row in the declaration. You may recall that in this case, all the remaining elements are set to 0. We can confirm this by a call to the printarray function in line 13 to print the values of the array.

Lines 15 to 22 convert the matrix into a 4 × 4 “diagonal matrix” with 3 along the diagonal. Again, the function printarray prints the array.

Lines 26 to 28 set the value of mat[i][j] to i + j. Again, we call printarray function to print the values of the array.

Here are some exercises for you to test your understanding of array operations.

---

**E4)** Write a program in C that declares a 4 × 4 array and reads the entries of the array from the terminal
  a) Row by row.
  b) Column by column.

**E5)** Write a function in C that takes an array of integers of size 4 × 4 as input and changes the all the elements above the diagonal to zero.

---

We close the section here. In the next section, we will discuss how the entries of an array are stored in the main memory.

### 11.5 STORAGE OF ARRAYS IN MAIN MEMORY

Let us now see how the data represented in an array is actually stored in the memory cells of the machine. Because computer memory is linear, a one-dimensional array can be mapped on to the memory cells in a rather straightforward manner. Storage for element A[I+1] will be adjacent to storage for element A[I] for I = 1, 2, …, N. We assume that the size of each element stored is one unit. To find the actual address of an element one merely needs to subtract one from the position of the desired entry and then add the result to the address of the first cell in the sequence.

Let us look at an example. Consider an array A of 25 elements. We require to find the address of A[4]. If the first cell in the sequence A[0], A[1], A[2], …, A[25] was at address 16, then A[4] would be located at 16 + (5 − 1) = 20, as shown in Fig. 5. Therefore, it is necessary to know the

<table>
<thead>
<tr>
<th>Memory</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>…</th>
<th>…</th>
<th>…</th>
</tr>
</thead>
</table>

*Fig. 5: Storage of Arrays.*
starting address of the space allocated to the array and the size of the each element which is same for all the element of an array. We may call the starting address as a base address and denote it by B. Then the location of Ith element would be

$$B + (I - 1) \times S$$  \hspace{1cm} (1)

where S is the size of each element of array. We refer you to the fourth example program in Unit 12 of Block 2 which illustrates this.

Let us now consider storage mappings for multi-dimensional arrays. As we had seen in previous section that in a 2-dimensional array we think of data being arranged in rows and columns. However Machine’s memory is arranged as a row of memory cells. Thus the rectangular structure of a 2-dimensional array must be simulated. We first calculate the amount of storage area needed and allocate a block of contiguous memory cells of that size. One way to store the data in the cells is row by row. That is, we store first the first row of the array, then the second row of the array and then the next and so on. For example, the array defined by A which logically appears as given in Fig. 5 on the preceding page appears physically as given in Fig. 7. Such a storage scheme is called Row Major Order.

The other alternative is to store the array column by column. It is called Column Major Order. The array of Fig. 8 shows the physical arrangement in Column Major order.

---

E6) Create a two-dimensional array whose number of rows are 10 and columns are 26 and the component type is character.

E7) Show how the array

\[
\begin{bmatrix}
1 & 3 & 7 \\
5 & 2 & 8 \\
9 & 7 & 1
\end{bmatrix}
\]

would appear in the memory when stored in

i) Row major order

ii) Column major order

---

In all implementations of C the storage allocation scheme used is the Row Major Order.
Let us now see how do we calculate the address of an element of a 2-dimensional array, which is mapped in Row Major Order. Consider a $4 \times 6$ array $A[4][6]$. Take $B$ as the array’s base address and $S$ as the size of element of the array.

Remember that in C, array indices start with 0. So, to locate element $A[I][J]$ we must skip $I$ rows (0, 1, 2, ..., $I-1$); each having 6 elements, each element of length $S$ and ($J$) elements of $I$th row, each of length $S$. Therefore, the address of element $A[I][J]$ would be

$$B + I \cdot 6 \cdot S + J \cdot S$$

We may now generalise this expression for a 2-dimensional array

$$A[U_0, U_1]$$

where $U_0 - 1$ and $U_1 - 1$ are the upper bounds of the two subscript ranges.

The location of an element $A[I][J]$ for such an array would be

$$B + I \cdot U_1 \cdot S + J \cdot S$$

The row major order varies the subscripts in right to left order. For example, the elements of a 2-dimensional array $A[U_1, U_2]$ would be stored in following order:

$$A[0, 0]$$
$$A[0, 1]$$
$$\vdots$$
$$A[0, U_2 - 1]$$
$$A[1, 0]$$
$$A[1, 1]$$
$$A[1, 2]$$
$$\vdots$$
$$A[1, U_2 - 1]$$
$$\vdots$$
$$A[U_1 - 1, U_2 - 1]$$

We may generalise it for an N-dimensional array $A[U_0][U_1] \ldots A[U_n-1]$. The elements would be stored in following order:

$$A[0][0] \ldots [0]$$
$$A[0][0] \ldots [1]$$
$$\vdots$$
$$A[0][0] \ldots A[0][A[U_n - 1]]$$
$$A[0][0] \ldots A[1][A[0]]$$
$$\vdots$$
$$A[0][0] \ldots [1][1]$$
$$\vdots$$
$$A[U_0 - 1][A[U_1 - 1] \ldots A[U_n - 1]]$$

Let us see how the above expressions work out for a column major order.

We once again consider a $4 \times 6$ array $A[4][6]$. Also take $B$ as base address and $S$ as size of each element. Then the address of $A[I][J]$ would be

$$B + J \cdot 4 \cdot S + I \cdot S$$

To reach $A[I][J]$ we shall skip $J-1$ columns, each of length $4 \cdot S$ and $I-1$ elements each of length $S$.

Let us now generalise it for an array $A[U_1, U_2]$. 


Following the same logic, the address of $A[i][j]$ would be given as

$$B + j \cdot U_1 \cdot S + i \cdot S$$

The column major order varies the subscripts in left to right order. For example the elements of a 2-dimensional array $A[4][3]$ would be stored in the sequence as given below:

$A[0][0]$
$A[1][0]$
$A[2][0]$
$A[3][0]$
$A[0][1]$
$A[1][1]$
$A[2][1]$
$A[3][1]$
$A[0][2]$
$A[1][2]$
$A[2][2]$
$A[3][2]$

E8) How would a $4 \times 3$ array $A[4][3]$ stored in Row Major Order?
E9) How would a $m \times n$ array $A[m][n]$ stored in Column Major Order?

As we had done for Row Major Order, we may generate the sequence of $N$-dimensional array

$A[U_1][U_2][U_3] \ldots [U_n]$ as stored in Column Major Order. It would be as given below:

$A[0][0] \ldots [0]$
$A[1][0] \ldots [0]$
$\vdots$
$A[U_1-1][0] \ldots [0]$
$A[0][1] \ldots [0]$
$\vdots$
$A[0][2] \ldots [0]$
$\vdots$
$A[0][U_2-1] \ldots [0]$
$\vdots$
$A[U_1-1][U_2-1] \ldots [U_n-1]$

In the numerical computations involving matrices, we often come across matrices where we need to deal with matrices that have lots of zeros. We will be wasting computing space as well as time if we use the usual methods for working with these arrays. We will look at some special methods for representing and working with these matrices.

### 11.6 SPARSE ARRAYS

Often, in numerical computing, we come across matrices with lots of zeros. Such matrices are called **sparse matrices**. Sparse arrays are special arrays which arise commonly in applications. It is difficult to draw the dividing line between the sparse and non-sparse array. Loosely an array is called sparse if an entry (commonly 0) occurs relatively large number of times. For example,
Fig. 9: A sparse array.

in Fig. 9 out of 49 elements, only 6 are non-zero. This is a sparse array. If we store those array through the techniques presented in previous section, there would be much wasted space.

Let us consider 2 alternative representations that will store explicitly one the non-zero elements.

1) Vector representation
2) Linked List representation

We shall discuss only the first representation here in this Unit. The Linked List representation shall be discussed in a later Unit.

Each element of a 2-dimensional array is uniquely characterised by its row and column position. We may, therefore, store a sparse array in another array of the form $A[n+1][3]$ where $n$ is number of non-zero elements.

The sparse array given in Fig. 9 may be stored in the array $A[7][3]$ as shown in Fig. 10. The

$$A = \begin{array}{ccc}
0 & 1 & 2 \\
7 & 7 & 6 \\
1 & 1 & 6 \\
2 & 2 & 5 \\
3 & 4 & 5 \\
4 & 5 & 2 \\
5 & 6 & 5 \\
6 & 7 & 5 \\
\end{array}$$

Fig. 10: Sparse array representation using vector representation.

Elements $A[0][0]$ and $A[0][1]$ contain the number of rows and columns of the sparse array. $A[0][2]$ contains the number of non-zero elements of sparse array. The first and second element of each of the rows store the number of row and column of the non-zero term and the third element stores the value of non-zero term. In other words, each non-zero element in a 2-dimensional sparse array is represented as a triplet with the format (row subscript, column subscript, value).

If the sparse array was one-dimensional, each non-zero element would be represented by a pair.

In general for an N-dimensional sparse array, non-zero elements are represented by an entry with N+1 values.

If you are interested in alternate representations of sparse matrices and methods for working with them, you may refer to Sec. 2.7 in the book ‘Numerical recipes in C’ and the references given there.

11.7 SUMMARY

In this Unit, we saw

i) the benefits of program analysis.
Introduction to Data Structures: Array

11.8 SOLUTIONS/ANSWERS

E1) /*Program 11.1. A simple example of Array operations
File name:unit11-insert-element.c*/
#include<stdio.h>
define max_array_size 20
void Error(char *message);
void Message(char *message);
int printarray(int *list,int limit);
int delete_element(int *list, int last_index, int index);
int insert_element(int *list, int num, int last_index, int index);
int main()
{
  int a[max_array_size]={3,4,5,6,7}, last = 5;
  printarray(a,last);
  last = delete_element(a,last,2);
  Message("After deleting 5 the array is:");
  printarray(a,last);
  last = insert_element(a,10,last,4);
  Message("After inserting 10 in the fifth position the array is:");
  printarray(a,last);
  return (0);
}
void Error(char *message)
{
  printf("n");
  fprintf(stderr,"Error! %s\n",message);
}
void Message(char * message)
{
  fprintf(stdout,"nMessage: %s\n",message);
}
int printarray(int *list,int last)
{
  int i;
  for(i = 0; i < last; i++)
    printf("%d\n", list[i]);
  return (0);
}
int delete_element(int *list, int last_index, int index)
{
  int i;
  for(i = index; i < last_index; i++)
    list[i]=list[i+1];
  return (last_index-1);
}
int insert_element(int *list, int num, int last_index, int index)
{
  int i;
  if (last_index == max_array_size) {
    Error("Array is full. Cannot insert element.");
    return (-1);
  }
else {
    for (i = last_index-1; i > index; i--)
        list[i+1]=list[i];
    list[index]=num;
}
return(last_index+1);
}

E2) /*Program 11.2. A example to show sorting of arrays.
Insertion sort. File name:unit11-insortn.c*/
#include<stdio.h>
#define max_array_size 20
int find_position(int *a, int num);
int insert_number(int *a, int cpos, int npos, int num);
int main()
{
    int i, k = 0, array[max_array_size] = { 0 }, list_size;
    printf("Enter the size of the list:\n");
    scanf("%d", &list_size);
    printf("Enter the elements of the array:\n");
    for (i = 0; i < list_size; i++)
        scanf("%d", &array[i]);
    for (i = 1; i < list_size; i++) {
        k = find_position(array, i);
        if (k != i)
            insert_number(array, i, k, array[i]);
    }
    printf("The sorted array is:\n");
    for (i = 0; i < list_size; i++)
        printf("%d\n", array[i]);
    return (0);
}
int find_position(int *list, int pos)
{
    int j;
    for (j = pos; j > 0 && list[j - 1] > list[pos]; j--);
    return (j);
}
int insert_number(int *list, int cpos, int npos, int num) /* cpos is the current position of num;
npos is the new position of num*/
{
    int i;
    /*Shift elements to the right by one place*/
    for (i = cpos; i > npos; i--)
        list[i] = list[i - 1];
    /*Insert the number at the new position.*/
    list[npos] = num;
    return 0;
}

E3) int insertion_sort(int *list, int LL)
{
    int cp, j, temp;
    for (cp = 1; cp < LL; cp++)
    {
        temp = list[cp];
        for (j = cp; j > 0 && list[j - 1] > temp; j--)
            list[j] = list[j - 1];
        list[j] = temp;
    }
    return (0);
}
UNIT 12  LISTS

12.1 INTRODUCTION

In Unit 11 of this Block we discussed a basic data structure, arrays. Arrays, although available in almost all the programming languages, have certain limitations on structuring and accessing data. In this Unit we turn our attention to another data structure called the List.

Lists, like arrays are used to store ordered data. A List is a linear sequence of data objects of the same type. Real-life events such as people waiting to be served at a bank counter or at a railway reservation counter, may be implemented using List structures. In computer science, Lists are extensively used in data base management systems, process management, operating systems, editors etc.

In Sec. 12.2, we introduce basic terminology related to Lists. In Sec. 12.3, we discuss static implementation of Lists using arrays. In Sec. 12.4, we discuss dynamic implementation of lists using pointers. We also discuss various operations that can be performed on a List like insertion of an element, deletion of an element etc. We had already seen how to store sparse matrices using arrays in Unit 11. Here, we will discuss storage of sparse arrays using linked Lists. In Sec. 12.5, and Sec. 12.6, we will discuss some variants of linked Lists called doubly linked Lists and circular linked Lists, respectively. In the last two sections, we will discuss some applications of lists to garbage collection and storage allocation.

Objectives

After studying this unit, you should be able to

• store data structures in computer memory in two different ways viz. sequential allocation and linked allocation;
• differentiate between a linear list and an array;
• implement linear lists in terms of built-in data types in C;
• code following algorithms using a linked list represented as an array of records;
  • creating a linked list
  • inserting an element at a specified location in a linked list
  • deleting an element from a linked list.
12.2 BASIC TERMINOLOGY

In this section we will introduce you to basic terminology. We begin by defining a List.

Definition 1: A linear List is an ordered set consisting of a variable number of elements to which addition and deletions can be made. A linear list displays the relationship of physical adjacency.

The first element of a List is called head of List and the last element is called the tail of List.

Every element of List, unless it is the head has a predecessor and every element of the List, unless it is the tail of the List, has a successor.

The elements in a List are tied together by their successor-predecessor relationship.

Following are some of the basic operations that may be performed on a List:

- Create a List
- Check for an empty List
- Search for an element in a List
- Search for a predecessor or a successor of an element of a List
- Delete an element from a List
- Add an element at a specified location of a List
- Retrieve an element from a List
- Update an element of a List
- Sort a List
- Print a List
- Determine the size or number of elements in a List
- Delete a List

More complex operations may be performed on a List. However a complex operation would generally turn out to be a combination of two or more of the above basic operations.

A List can be implemented statically or dynamically using an array index or pointers respectively. We will discuss static implementation of Lists in the next section.

12.3 STATIC IMPLEMENTATION OF LISTS

Static implementation is the simplest implementation. Its size is fixed and allocated at compilation time. A List can be implemented as an array as follows:

Let our List elements be names of all the colours, say BLUE, RED, YELLOW, GREEN and ORANGE.

We may have an array List declared as List[LIST_SIZE], and fix its size as 8. Therefore, we have something like in Fig. 1 on the next page. The elements are sequentially stored in LIST[0], LIST[1], . . . , LIST[4]. LIST[5] through LIST[7] are allocated but not used.
The predecessor of LIST[0] (or head) is NIL; LIST[4] is the tail of the List and has no successor. We may also tabulate the predecessors and successors of the other elements of the List as in Table 1.

Fig. 1: A list declared as an array.

Table 1: Predecessors and successors.

<table>
<thead>
<tr>
<th>Data</th>
<th>Array Index</th>
<th>Predecessor Index</th>
<th>Successor Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLUE</td>
<td>0</td>
<td>NIL</td>
<td>1</td>
</tr>
<tr>
<td>RED</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>YELLOW</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>GREEN</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>ORANGE</td>
<td>4</td>
<td>3</td>
<td>NIL</td>
</tr>
</tbody>
</table>

Since the elements are sequentially stored, we don’t need to store the predecessor and successor indices. Any element in the List can be accessed through its index.

Now let us see an Insert operation. If we want to insert an element at Kth position i.e. after LIST[K-1].

To do this we must shift elements LIST[K] through LIST[Last] to respectively LIST[K+1] through LIST[Last+1]. At the same time we must also check that Last + 1 does not exceed the value of SIZE.

LIST[Last] is nothing but the tail of the List.

Let us see the operations to be performed in an algorithmic form.

```c
/*Check if Last+1 is less than or equal to Size*/
if (Last + 1 > List_size - 1)
    error; /*overflow*/
else{
    /*Shifting elements from K + 1 through Last + 1*/
    for(i = Last; i > K; i--)
        List[i] = List[i-1];
    /*Insert element in the Kth position*/
    List[K]=element
    Last=Last+1;
}
```

In the delete operation we need to shift element in upwards direction and also decrement the value of tail by 1.

**Example 2:** Let us now write a C program that creates a list that can hold 10 strings. We will set the sets the first five elements to BLUE, RED, YELLOW, GREEN and ORANGE. Then we will insert CYAN in the third position.

```c
/*Program-12.1. Example of List implementation using array.
 File name:unit12-listarray.c*/
#include <stdio.h>
```
#include <string.h>
#define MAX_L 10
#define MAX_WD 20
int last=0;
int ins_element(char array[MAX_L][MAX_WD], int pos, char *text);
int del_element(char array[MAX_L][MAX_WD], int pos);
int print_array(char array[MAX_L][MAX_WD]);
void Error(char *message);
int main()
{
    char colours[MAX_L][MAX_WD];
    int i;
    for (i = 0; i < MAX_L; i++)
        strcpy(colours[i], "-" AUTHOR: ");
    ins_element(colours, 0, "BLUE" AUTHOR: ");
    ins_element(colours, 1, "RED" AUTHOR: ");
    ins_element(colours, 2, "YELLOW" AUTHOR: ");
    ins_element(colours, 3, "GREEN" AUTHOR: ");
    ins_element(colours, 4, "ORANGE" AUTHOR: ");
    print_array(colours);
    ins_element(colours, 2, "CYAN" AUTHOR: ");
    print_array(colours);
    del_element(colours, 3);
    print_array(colours);
    return 0;
}

Here is the function that inserts an element in a list.

int ins_element(char array[MAX_L][MAX_WD], int pos, char *text)
{
    int j = MAX_L - 1;
    if(last >= j){
        Error("Error! Overflow!!")
    }
    else{
        for (; j > pos; j--)
            strcpy(array[j], array[j - 1]);
        strcpy(array[pos], text);
        last=last+1;
    }
    return 0;
}

We use the Error() function that we used in the answer to exercise 1 of Unit 11 to print an error message if the list is already full. Here is the function that deletes a particular element from the list.

int del_element(char array[MAX_L][MAX_WD], int pos)
{
    int i;
    for (i = pos; i < MAX_L - 1; i++)
        strcpy(array[i], array[i - 1]);
    strcpy(array[MAX_L - 1], "-" AUTHOR: ");
    last = last-1;
    return 0;
}

You may recall that list[i] is a pointer to the i th row of the array. Also, note the use of strcpy() here. This is necessary because C language does not allow assigning one array to another; we have to copy one array to another element by element.

***
E1) Write C functions that:
   i) Print the array in our example.
   ii) Return the i\textsuperscript{th} element of the array.
   iii) Return the number of elements in the array.

You must have noticed that array implementations of a List has certain drawbacks. These are:

1) Memory storage space is wasted; very often the List is much shorter than the array size declared.

2) List cannot grow in its size beyond the size of the declared array if required during program execution.

3) Operations like insertion and deletion at a specified location in a List required a lot of movement of data, therefore, leading to inefficient and time consuming algorithms.

Some of these drawbacks can be avoided if we implement lists using pointers. We will do so in the next section.

### 12.4 POINTER IMPLEMENTATION OF LISTS

In this section, we will discuss lists implemented through pointers. We will also compare this with the static implementation of lists through arrays. Let us now go back to the example we used in the earlier section. Before we actually implement the list, let us discuss the concept through diagrams.

Each element of a linked list is called a **node**. For the List elements BLUE, RED, YELLOW, GREEN and ORANGE, we can form a linked List using pointers. The last node in the List points to NULL. The structure of such a List may be schematically shown in [Fig. 2](#). We have indicated that the pointer in the last node points to the NULL pointer by a \. This is a singly linked List structure i.e. each of its elements have

- data and
- a pointer pointing to next element of the List

For an empty List the Head points to NULL.

The primary advantage of Linked Lists over arrays is that Linked Lists can grow and shrink in size during their lifetime. In particular, their maximum size need not be known in advance. In practical applications, this often makes it possible to have several data structures share the same space, without paying particular attention to their relative size at any time.

A second advantage of Linked Lists is that they provide flexibility in allowing the items to be rearranged efficiently. This flexibility is gained at the expense of quick access to any arbitrary item in the List. This will become more apparent below, after we have examined some of the basic properties of Linked Lists and some of the fundamental operations we perform on them.

Now, this explicit representation of the ordering allows certain operations to be performed much more efficiently than would be possible for arrays. For example, suppose that we want to move
the GREEN to the beginning of the List. In an array, we would have to move every item to
make room for the new item at the beginning; in a linked List, we just change three links.

Let us now to write a C program that create a linked list with the colours BLUE, RED,
YELLOW and GREEN. We will do this in stages.

Example 3: We start by defining a self referential structure called node. This is called a self
referential structure because the second component of the structure is a pointer to another
structure of the same type. Instead of referring to this again and again as struct node, we use
the typedef statement to create a new type called Node. Here is a small program that creates a
linked list containing one node. Let us examine this program.

```c
/*Program 12.2. Example node creation in
Linked lists. File name:unit12-myfirstll.c*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_WD 20
struct node{
    char colour[MAX_WD];
    struct node *next;};
typedef struct node Node;
Node *CreateNode(char *colour);
void Error(char *message);
int main()
{
    Node *head = NULL;
    head=CreateNode("RED");
    printf("%s", head->colour);
    return 0;
}
Node *CreateNode(char *Colour)
{
    Node *ptr;
    if((ptr = malloc(sizeof(Node)))"){
        ptr->next= NULL;
        strcpy(ptr->colour,Colour);
        return (ptr);
    }
    else{
        Error("Unable to create node!");
        return (ptr);
    }
}
void Error(char *message)
{
    fprintf(stderr,"Error! %s\n",message);
}
```

In lines 7 to 9 of this program we define a self referential structure called node. The first
component of this structure is a character array of size MAX_WD which we have #defined to be
20 earlier in line 6.

Line 20 calls the function CreateNode() and assigns the value returned by it to the pointer
head. As we will see, CreateNode() returns a pointer to a newly created node and head
will also point to this node. The function CreateNode() creates a node with the colour
passed to it as argument. The function calls malloc() to allocate memory and assigns the
pointer returned by malloc() to the local variable ptr which is a pointer to Node. If the
malloc() is unable to allocate memory it will return NULL pointer. In this case, the value of
the expression ptr=malloc(sizeof(Node)) will be 0 and so the else part part of the if
statement will print an error message. Otherwise, the line ptr->next=NULL initialises the
node to NULL and copies the name of the colour, which is a string, on to colour part of the
Node. Line 26 returns the pointer to the main program. The \texttt{printf()} statement in line 17 checks if the node has been created successfully. The situation after creation of the node is given in Fig. 3. Throughout this example, we will say ‘the node RED’, ‘the node BLUE’ etc. instead of saying ‘the node containing RED’, ‘the node containing BLUE’ etc.

We will now see how to insert a node at the beginning of the list. We will now insert a new node containing the colour BLUE at the beginning of the list. Here are the steps involved (We have shown the steps in Fig. 4):

a) Create a new node using \texttt{CreateNode()} function and make \texttt{new} point to the same location as the pointer returned by \texttt{CreateNode()}.

\begin{verbatim}
new = CreateNode("BLUE");
\end{verbatim}

b) Make the pointer in \texttt{new} point to where \texttt{head} points, i.e. the node RED.

\begin{verbatim}
new->next = head;
\end{verbatim}

c) Make the pointer \texttt{head} point at the \texttt{new} node, making it the first node.

\begin{verbatim}
head = new;
\end{verbatim}

Let us now add the colour GREEN at the end of the list.

a) We need a pointer to the last node. We make pointer \texttt{current} point to \texttt{Node} and initially make it point to the same node as \texttt{head}. See Fig. 5a on the following page.

\begin{verbatim}
current = head;
\end{verbatim}

b) Then, we advance \texttt{current} till it points to the last node in the list through a while loop:

\begin{verbatim}
while (current->next != NULL)
current = current->next;
\end{verbatim}

Initially, \texttt{current} points to the node containing BLUE. So, \texttt{current->next} is a pointer to where the pointer in the node containing BLUE points; this is the node containing RED. Since this isn’t NULL, the statement in the \texttt{while} loop is executed. The effect of the statement ‘\texttt{current=\texttt{current}\rightarrow\texttt{next};}’ is to make \texttt{current} point to the where the pointer in the node corresponding to BLUE points; this is the node containing RED. But, the pointer in the node corresponding to RED points to the NULL pointer and so the condition in the \texttt{while} loop is not satisfied. Now, we have the pointer \texttt{current} pointing to the last node, namely the one corresponding to RED. 

---

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{A linked list with one node.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{Inserting a new node at the beginning of a list.}
\end{figure}

---

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{Inserting a new node at the end of a list.}
\end{figure}
Data Structures

(a) Create a new pointer to the first node.

(b) Advance the pointer current to point to RED

(c) Create a new node and set the value of name to GREEN and make the pointer in RED point to GREEN

Fig. 5: Adding a new node at the end of a list.

c) As before, we create a new node, set the first component of the node as GREEN.

    new = CreateNode("GREEN");
    current->next = new;
    new->next = NULL;

Then, we use the statement current->next=new to make the pointer in the node corresponding RED point to the newly created node. Finally, we make the pointer in the newly added node point to NULL, since this is the last node.

Let us see how we can add a node in the middle of a linked list. Let us add a node corresponding to YELLOW after RED. Here are the steps involved. See Fig. 6.

1) Make current point to the first node.

2) Advance current to point to the node containing RED. Use a new pointer called prev. Make prev and current point to the same node.

3) Advance current to point to the next node containing GREEN.

4) Create a new node. We have to make this pointer in this new node point to the node containing RED. The pointer current points to GREEN. So, the statement new->next=current; makes new->next also point at GREEN. Then, we have to make the pointer in the node containing RED to point to the new node. The pointer prev points to RED; so we can achieve this using the statement prev->next=new;

Here is the complete listing of the C program.

```c
/*Program to demonstrate the creation and insertion of nodes in a linked list. File name:uni12-myfirstll-2.c*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_WD 20

struct node{
    char colour[MAX_WD];
    struct node *next;};
typedef struct node Node;
Node *CreateNode(char *colour);
void Error(char *message);
void printlist(Node *);
int main()
```
Fig. 6: Adding a node in the middle.

```c
15 {  
17   head = NULL;
18   head = CreateNode("RED");
19   new = CreateNode("BLUE");
20   new->next = head;
21   head = new;
22   /*Insert node GREEN at the end. */
23   current = head;
24   while (current->next != NULL)
25     current = current->next;
26   new = CreateNode("GREEN");
27   current->next = new;
28   new->next = NULL;
29   printlist(head);
30   /*Insert YELLOW after RED */
31   current = head;
32   /*current points to BLUE */
33   current = current->next;
34   /*Current points to RED now */
35   new = CreateNode("YELLOW");
36   prev = current;
37   /*save the value of current in prev.
38    prev points to RED now. */
39   current = current->next;
40   /*current points to GREEN now. */
```
Here is an exercise to test your understanding of the earlier material.

E2) Write a program that creates a linked List with entries 1, 2, 3, 4 and 5.

The aim of the example above was to help you understand the process of creating nodes. Obviously, the procedure will be tedious if we want to create a list with 100 nodes. In practice, we will need a function that creates and insert a new node. We will see how to do this in the next example.

Example 4: In this example, let us write functions for inserting new nodes in a list. First let us write a function that adds a node at the beginning of an existing linked list.

```c
void add_node(Node **headref, char cname[MAX_WD])
{
    Node *new = CreateNode(cname);
    new->next = *headref;
    *headref = new;
}
```

You may have noticed **headref, a pointer to pointer! Why do we need to do this? Remember that, in C, if we want a function to change an object, we have to pass a pointer to the object that we want to change. Here, when we add a new node, head will point to this new node instead of wherever it was pointing before. Since, we want to change the contents of the pointer variable head we have to pass a pointer to head, not just head. So, we will pass the
value of head by the statement add_node(&head). So, the function must accept a pointer to a pointer variable.

Let us now write a function that inserts a new node at the n^{th} position, regardless of where it is, in the beginning, in the end or in the middle. The function will take the pointer to the head pointer, the colour of the new node and the position where we want to insert the new node as the arguments. The function should do the following:

If \textit{position} = 0 call add_node() to insert a node at the beginning;
else
Create a new pointer called current and advance it point to \textit{position};
If current does not point to the last element of the list, pass the pointer current to add_node_mid();
elseif
Pass the pointer current to add_node_end().

Here is the function that inserts a node in the middle:

\begin{verbatim}
void add_node_mid(Node **headref, char name[MAX_WD])
{
    Node *current, *new, *prev;
    current = *headref;
    prev = current;
    current = current->next;
    new = CreateNode(name);
    prev->next = new;
    new->next = current;
}
\end{verbatim}

Here is a function that inserts a node in the end.

\begin{verbatim}
void add_node_end(Node **headref, char name[MAX_WD])
{    Node *current, *new;
    current = *headref;
    new = CreateNode(name);
    new->next = NULL;
    current->next = new;
}
\end{verbatim}

Here is a function that checks the position where we want to insert the node and call appropriate functions to insert them in the correct position.

\begin{verbatim}
void ins_node(Node **headref, char cname[MAX_WD], int num)
/*Function that inserts the colour cname after num nodes */
{
    Node *current;
    int count = 1;
    current = *headref;
    if (num == 0)
        add_node(headref, cname);
    else
        {           //End of if */
            while (count < num)
            {
                if (current->next == NULL)
                {
                    printf("\nThere are only %d nodes.\n", count);
                    printf("Cannot insert node after position %d.\n", num);
                    exit(1);
                }
            }
        }
}
\end{verbatim}
current = current->next;
count += 1;
} /*End of while */
/*check if num is the last node. */
if (current->next == NULL)
    add_node_end(&current, cname);
else
    add_node_mid(&current, cname);
}

***
Here are some exercises to test your understanding of the previous example.

E3) Write a function that returns the number of nodes in the list we created in the previous example.

E4) Write a function that will delete the node after the n\textsuperscript{th} node.

Let us now discuss an application of linked list, namely addition of two polynomials. Linked lists are convenient when we have to add two polynomials of high degree with many coefficients zero. For example, consider the problem of adding two polynomials.

$$x^5 - 12x^3 + 5x^7 - 8x^3 + x + 1$$ and $$x^7 - 12x^{14} + 7x^{10} + 7x^3 + 2x^3 + x + 9$$.

If we use arrays, we will need three arrays, one of size 27, another of size 25 and a third one of size 27 to hold the answer. Further, there are many terms which are 0 in both the polynomials, yet we have to take them into account. We will see how to use linked lists to add these polynomials.

Example 5: Let us first define the structures that will hold the terms. Note the use of \texttt{typedef} in this.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct poly {
    int coeff;
    int deg;
    struct poly *next;
} Poly;
```

Let us now write a function in C that adds a term of a given degree and given coefficient to an existing polynomial. The function assumes that polynomial is constructed in such a way, starting from the head node, the terms are arranged in descending order with the head pointing to the highest degree term. Given a term, it checks if the polynomial has any terms. If it doesn’t have any terms, it adds a new term. Otherwise, it inserts the term at an appropriate place. If there is already a term with given degree, it adds the coefficient of the new term to the existing term. Here is the full listing of the function with copious comments. Please go through it carefully.

```c
void add_term(Poly ** poly1, int coeff, int degree) {
    Poly *new, *current, *prev;
    current = *poly1;
    new = malloc(sizeof(Poly));
    new->coeff = coeff;
    new->deg = degree;
    new->next = NULL;
    /*The polynomial has no terms */
    if (*poly1 == NULL)
        *poly1 = new;
    else
        /* If the term being added has higher degree than the highest degree term in poly1, add the term in the beginning */
        if (current->deg < degree) {
            new->next = current;
            current = new;
        } else
            /* Insert the term in the middle */
            prev = current;
            while (current->deg < degree) {
                prev = current;
                current = current->next;
            }
            prev->next = new;
            new->next = current;
        }
    return;
}
```
*poly1 = new;
} else {
/* Advance current till it points to the last term(node) or
to a term of degree not greater than the degree of the terms
we want to insert. */
    while (current->deg > degree && current->next != NULL)
    current = current->next;
/*If we have reached the last term of poly and the degree
of the last term is greater than that of the term we want
to insert, add the term at the end. */
    if (current->next == NULL && current->deg > degree)
        current->next = new;
    else {
/*If we have found a term with the same degree as the term we
want to insert, merely add the coefficient of the term we want
insert to coefficient of the existing term of the same degree; */
    if (current->deg == degree) {
        current->coeff = current->coeff + coeff;
    } else {
/*otherwise add a new term. */
    prev = current;
    prev->next = new;
    new->next = current;
    };
}

Here are some exercises for you to check your understanding of the representing polynomials
using linked lists.

---

E5) Using the function in the example above, write a program that represents the polynomial
\[ x^22 - 28x^{17} + 12x^{13} - 34x^{11} + x^9 + x^2 + x + 1 \]
as linked list.

E6) Using the function in the example above, write functions for adding, subtracting and
multiplying two polynomials.

---

12.4.1 Storage of Sparse Arrays using Linked List

It is often necessary to deal with large arrays in which many of the element has a zero value.
Such arrays are called sparse arrays (see Unit 11 of this Block). We have already discussed one

```
0 0 3.5 0 0 0 0 0
0 1.2 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 5.5
0 0 0 0 2.5 0 0 0
0 0 0 0 0 6.7 0
```

---

Fig. 7: A sparse array.

---

of the methods of storing these sparse arrays, i.e. by using a 3-tuple for each element. Often non
zero elements need to be added or deleted. That requires a lot of data movement in a static
storage system. An improvement over this would be to store these non zero element as a linked
List of 3 tuples instead of using an array. **Fig. 8 on the next page** illustrates the linked List for
the sparse array given in **Fig. 7**. As you can see, there is node for each non-zero element. In
each node, the first element is the row number, the second element is the column number, the
third element is the non-zero element and the last component is a pointer to the next node. For example, the first row has a non-zero element 3.5 in the third column. The first node represents this element. We close the section here. In the next section, we discuss doubly linked lists.

```
1 3 3.5 2 2 1.2 4 8 5.5 5 5 2.5 6 5 6.7
```

**Fig. 8: Linked list for sparse array.**

### 12.5 DOUBLY LINKED LISTS

In the Linked Lists discussed in previous section, the traverse the List in one direction. In many applications it is required to traverse a List in both directions. This 2-way traversal can be realised by maintaining two link fields in each node instead of one. We call such a structure a Doubly Linked List. Each element of a doubly linked List structure has three fields

- data value
- a link to its successor
- a link to its predecessor

The predecessor link is called the left link and the successor link is known as the right link.

Here is the definition of the node of a doubly linked list that holds integer data.

```c
typedef struct dlnode {
    int data;
    struct dlnode *left;
    struct dlnode *right;
} Dlnode;
```

Since there are nodes in both the directions, the traversal of the List can be in any direction. We have a structure as given in **Fig. 9**. Note that the left link of lefmost node and right link of rightmost node are NULL.

**Fig. 9: A doubly linked list.**

#### Insertion of a Node

To insert a node into a doubly linked List to the right of a specified node, we have to consider several cases. These are as follows:

1. If the List is empty, i.e. the left and right link of specified node pointed to by say variable HEAD are NULL. An insertion in this node is simply making left and right pointers point to the new node and left and right pointers of new node to be set to NULL.
2. If there is a predecessor and a successor to the given node. In such a case we need to readjust pointers of the specified node and its successor node. The procedure is shown in **Fig. 10 on the facing page**.
3. The insertion is to be done after the right most node in the List. In such a case only the right link of the specified node i.e. the rightmost node is to be changed.
Here is a function that creates a new node and returns a pointer to it.

```c
Dlnode *getnode()
{
    Dlnode *p;
    p = malloc(sizeof(Dlnode));
    return (p);
}
```

Here is a function that inserts a node before the node at pos.

```c
void ins_dlnode_lt(Dlnode ** dlptr, int x, int pos)
{
    Dlnode *current, *new, *prev;
    int count = 1;
    if (*dlptr == NULL) {
        printf("Unable to insert to the left.");
        exit(1);
    }
    if (pos == 1) {
        /*When the node has to be inserted at the start of list.*/
        new = getnode();
        current = *dlptr;
        *dlptr = new;
        new->data = x;
        new->right = current;
        new->left = NULL;
    } else {
        /*Advance the pointer.*/
        for (; count < pos; count++)
            current = current->right;
        /*Create a new node.*/
        new = getnode();
        new->data = x;
        /*prev stores the node to the left of the pos*/
        prev = current->left;
        current->left = new;
        new->left = prev;
        prev->right = new;
        new->right = current;
    }
}
```

Here is a function that inserts a node after the node at pos position.

```c
void ins_dlnode_rt(Dlnode ** dlptr, int x, int pos)
{
    Dlnode *prev, *current, *new;
    int count = 1;
    current = *dlptr;
```
if (*dlptr == NULL) {
    if (pos > 1) {
        printf("Unable to insert new node.");
        exit(1);
    }
    if (pos == 1) {
        new = getnode();
        new->right = NULL;
        new->left = NULL;
        new->data = x;
        *dlptr = new;
    }
} else {
    new = getnode();
    new->data = x;
    for (; count < pos; count++)
        current = current->right;
    prev = current;
    current = current->right;
    new->left = prev;
    prev->right = new;
    new->right = current;
}

Here is a small program that illustrates the use of these functions.

#include <stdio.h>
#include <stdlib.h>

typedef struct dlnode {
    int data;
    struct dlnode *left;
    struct dlnode *right;
} Dlnode;

Dlnode *getnode();
void ins_dlnode_rt(Dlnode ** dlptr, int x, int pos);
void ins_dlnode_lt(Dlnode ** dlptr, int x, int pos);
int print_dlist(Dlnode * dlptr);

int main()
{
    Dlnode *head = NULL;
    ins_dlnode_rt(&head, 1, 1);
    print_dlist(head);
    ins_dlnode_lt(&head, 2, 1);
    print_dlist(head);
    ins_dlnode_rt(&head, 3, 2);
    print_dlist(head);
    ins_dlnode_lt(&head, -1, 1);
    print_dlist(head);
    return 0;
}

You may find it instructive to make diagrams like the ones we did in example 4 for these operations.

You may have noticed that the program uses a function to print the list. You may like to write one on your own. We have left it as an exercise to you. Try the following exercises.

E7) Can you guess what will be the contents of the list at the end of the program above?

E8) Write a function that prints the contents of the doubly linked list.
Another way of overcoming the disadvantages of singly linked list is a circular list. We discuss circular lists in the next section.

### 12.6 CIRCULAR LINKED LIST

Another alternative to overcome the drawback of singly linked list structure is to make the last element point to the first element of the list. The resulting structure is called a **circularly linked list** (Fig. 11). A circularly linked list is a list in which the link field of the last element contains a pointer to the first element of the list. The last element of the list no longer points to a NULL value.

![Circular List Diagram](Fig. 11: A Circular list.)

The definition of circular list is similar to that of circular list. Insertion of nodes in the middle is similar. However, insertion at the end or in the beginning is different. We leave it to you to modify the functions that we wrote for singly linked list for a circular list.

E9) Modify the functions that we wrote for inserting nodes in a linked list for a circular list.

We conclude this section here. In the next section, we discuss storage allocation.

### 12.7 STORAGE ALLOCATION

Initially, the operating system determines the areas available for allocation to users. We will use the term ‘node’ to designate a unit of the storage space.

Nodes are allocated to users in response to their requests. The number of nodes and the size of each node are decided keeping the following factors in mind.

1) Contiguity of space improves performance, especially for sequential access.
2) Having a large number of nodes leads to greater storage management effort.
3) Having fixed size nodes can lead to wastage of space.

The tradeoff can be summarised as:

1) Large nodes provide contiguous space and hence improve performance. They should be variable in size to prevent excessive wastage of space.
2) Small nodes improve flexibility. Much space is not wasted but their management is complex.

Having once decided the number and size of the nodes we now turn our attention to the actual allocation of these nodes. We make the assumption that we have multiple nodes of varying sizes and we want a storage space of size M. This can be done in one of the following ways:

- **1. Best Fit Method**

All available nodes are checked. Assuming the size of a node is represented by N.

Let \( D = N - M \)
The node whose value of N gives the least value for D is chosen and allocated.

The advantages of this method are obviously the minimal wastage of space.

The disadvantages are

- it involves searching all available nodes.
- it tends to increase the number of very small free blocks, when D not equal to 0.2.

2. First Fit Method

The available nodes are checked till we find one whose size N is greater than or equal to M, the requested size.

The advantage of this is obviously a smaller search among the available List of nodes. The disadvantage, like in the Best fit method could be that very small free blocks could be created. The way out of this is to fix a reasonable size C such that on getting the node of size NM,

\[ N(N - M) \leq C \]

allocate the entire node of size N

Else

Allocate space of size M

reserve node of size (n-M) for further use.

12.8 STORAGE POOLS

The collection of all nodes available is the Storage Pool. We will now deal with the management of this pool. This can be done in one of the following ways:

1) Bit Tables
   This method uses an array containing one bit per node. It is generally used when all nodes are of the same size, usually 1 block. A bit value of ‘0’ indicates that the corresponding node is free, and a value of ‘1’ indicates that it has been allocated. A separate file mechanism is needed to indicated the nodes allocated to a specific file. The advantages of this system are that the table can be kept in core memory so that allocation and deallocation (i.e. setting the bits to ‘1’ or ‘0’) costs are minimal.

2) Table of Contents
   This uses a file per unit (device, file system etc.) to describe the space allocation for the unit. This file will have (typically) the following data for each node - its size, whether allocated or not, if allocated - the name of the file, owner’s identification, date of creation etc.
   Like the bit table, this table of contents has to be searched to find free space for allocation. This problem may be overcome by keeping the records of free nodes in the ‘Free Space Table of Contents’. Allocation then means getting a suitable node from the ‘Free Space Table of Contents’ and moving it to the ‘Table of Contents’ after suitable updating. Freeing a node implies the reverse process.

3) Linked allocation
   Nodes can be linked together to overcome the limitations of the above two methods. In this method, each node will have a link to the next node in the List. Initially all nodes will be part of a free nodes List. On allocation to a file, a node will be detached from the free space List and added to the allocated List for that file. When a node is deallocated, it is detached from the allocated List of the file and attached to the free nodes List.
Linked Lists have been discussed in an earlier section. Thus we know Linked Lists can be singly or doubly linked depending upon the needs. This method provides an implicit gain in storage - in cases where Tables or files overlap, sharing common parts. The set of common nodes can be part of the allocation Lists of all the sharing tables.

The advantages of linked allocation are directly related to the case of operations on Linked Lists. Simple insertion and deletion from a linked List implies simplicity in inserting/deleting from a file. Ease of combination of Lists implies ease of joining files.

### 12.9 GARBAGE COLLECTION

Deallocation of nodes can take place in two levels:

1) The application which claimed the node releases it back to the operating system.

2) The operating system calls storage management routines to return free nodes to the free space.

For example Deallocation as in (1) occurs in a C program with the statement `free(x)` where `x` is space earlier allocated by a `malloc` call. (2) is usually implemented by the method of Garbage Collection. This requires the presence of a ‘Marking’ bit on each node. It runs in two phases. In the first phase, all non-garbage nodes are marked. In the second phase all non-marked nodes are collected and returned to the free space. Where variable size nodes are used it is desirable to keep the free space as one contiguous block. In this case, the second phase is called Memory compaction.

Garbage Collection is usually called when some program runs out of space. It is a slow process and its use should be obviated by efficient programming methods.

### 12.10 FRAGMENTATION, RELOCATION AND COMPACTION

Fragmentation literally means splitting. We have seen from our discussions in Sec. 12.7 that the ‘Best fit’ and ‘First fit’ algorithms result in creation of small blocks of space. These constitute wastage of space as they can be used to satisfy only requests for small blocks. This can be illustrated by the following example:

Consider the following space divided into three nodes of size 100 each.

```
A (Size: 100)
B (Size: 100)
C (Size: 100)
```

The following have to be allocated using first fit:

1) Size 50. This can be allocated from A. So, we get

```
A ← Occupied (Size: 50)
← Free (Size: 50)
B ← Free (Size: 100)
C ← Free (Size: 100)
```

2) Size 80. This can be allocated from B.

```
A ← Occupied (Size: 50)
← Free (Size: 50)
B ← Free (Size: 80)
← Free (Size: 20)
C ← Free (Size: 100)
```
3) Size 20. This can be allocated from A.

```
A ← Occupied (Size: 50)
      ← Occupied (Size: 20)
      ← Free (Size: 30)
B ← Free (Size: 80)
      ← Free (Size: 20)
C ← Free (Size: 100)
```

4) Size 50. This can be allocated from C.

```
A ← Occupied (Size: 50)
      ← Occupied (Size: 20)
      ← Free (Size: 30)
B ← Free (Size: 80)
      ← Free (Size: 20)
C ← Occupied (Size: 50)
      ← Free (Size: 50)
```

5) Size 75

Now there is no contiguous block of size 75 though the actual space available is 30 + 20 + 50 = 100

In this state of fragmentation, only requests of size \( \leq 50 \) can be satisfied.

This calls for ‘Relocation’ of the allocated blocks A[0-70], B[0-80], and C[0-50], so that the free blocks A[70-100], B[80-100] and C[50-100] can be ‘compacted’ to form one free block of size 100. The relocation of B and C will result in a change in their address. This change must be reflected wherever B and C are used.

![Fig. 12](image)

Relocation and Compaction usually form the second phase of Garbage collection. A[0-70], B[0-80], and C[0-50] constitute the ‘non-garbage’ nodes or ‘marked nodes’ which are relocated and A[70-100], B[80-100], and C[50-100] are the garbage nodes which are ‘compacted’ to release space.

### 12.11 SUMMARY

In this Unit we dealt with List and Linked List structures storage management. List structures allow us to store individual data elements. In Linked Lists these elements are interconnected by pointers. Beyond singly linked structure, we find several variations of List structures, e.g. doubly Linked Lists and circular Linked Lists. The Linked Lists allow us great flexibility in organising our information. We also discussed a related concept, i.e. of storage management in this Unit.

Storage is available for allocation on peripheral devices and in the main memory. It is managed either by means of a bit-table, table of contents file or by Linked Lists. Space is allocated either
using the best fit or first fit algorithms. Free space management is done by garbage collection which relocates fragmented free space and compacts it to get a continuous chunk of free space.

### 12.12 SOLUTIONS/ANSWERS

E1) Solution to i) is given in the listing below:

```c
int print_array(char array[MAX_L][MAX_WD])
{
    int i;
    printf("n");
    for (i = 0; i < last; i++)
        printf("%s\n", array[i]);
    return 0;
}
```

E3) /*Function that counts the number of nodes.
File name: unit12-ans-ex3.c.*/

```c
int cnt_node(Node *head)
{
    int count=0;
    for(count=0;head != NULL;count++)
        head=head->next;
    return(count);
}
```

E4) /*Function that deletes node after the.
th node File name: unit12-ans-ex-4.c.*/

```c
void del_node(Node **headref, int num)
{
    int count;
    Node *current,*temp;
    current=*headref;
    /*current now points to the head of the list.*/
    if(num==0){
        /*We want to delete the first node.*/
        *headref=current->next;
        /*Make head point at the second node.*/
        free(current);
        /*Free the memory used by the first node.*/
    }
    else{
        for(count = 0;count < num-1 ;count++)
            current=current->next;
        temp=current->next;
        current->next=current->next->next;
        free(temp);
    }
}
```
UNIT 13 STACKS AND QUEUES

13.1 INTRODUCTION

In the previous unit we discussed linear lists and their implementations. Lists may be modelled in many different types of data structures. We have been concentrating on structuring data in order to insert, delete, or access items arbitrarily. Actually, it turns out that for many applications, it suffices to consider various (rather stringent) restrictions on how the data structure is accessed. Such restrictions are beneficial in two ways; first, they can alleviate the need for the program using the data structure to be concerned with its details (for example, keeping track of links to or indices of items); second, they allow simpler and more flexible implementations, since fewer operations need to be supported. Two of such data structures are the focus of this Unit. These are Stacks and Queues. These are two special cases of linear lists.

Stacks and Queues are very useful in computer science. Stacks are used in compilers in parsing an expression by recursion, in memory management in operating systems etc. Queues find their use in CPU scheduling, printer spooling, message queueing in computer networks etc. The list of applications of stacks and queues in real life is enormous. In Sec. 13.2 of this Unit, we first define both the structures. Afterwards, in Sec. 13.3, we shall discuss their operations and implementations. In Sec. 13.4, Sec. 13.5 and Sec. 13.6 of this Unit, we shall take up some of the simple example applications of stacks and queues. In Sec. 13.7, we shall discuss priority queues.

Objectives

After studying this unit, you should be able to

• define stack and queue data structures;

• explain the operations that can be performed on them; and

• explain some applications of the stack and queue structures.

13.2 DEFINITION OF STACKS AND QUEUES

A stack is a linear data structure in which data is inserted and deleted at one end (same end), i.e. data is stored and retrieved in Last In, First Out (LIFO) order. The most recently arrived data object is the first one to depart from a stack. A stack operates somewhat like a busy executive’s ‘in’ box; work piles up on a stack and whenever the executive is ready to do some work, he takes it off the top. This might mean that something gets stuck in the bottom of stack for some
Data Structures

time, but a good executive would presumably manage to get the stack emptied periodically. It
turns out that sometimes a computer program is naturally organised in this way, postponing
some tasks and doing others. Thus, pushdown stacks appear as the fundamental data structure
for many algorithms. We may draw a stack in any one of the forms as in Fig. 1. Each one of the
above have an open and one closed end. The data movement (i.e. storage and retrieval) takes
place only at the open end, i.e. data is stored and retrieved in last in first out (LIFO) order. We
generally use the form given in [1a] having the open end in up direction. We will see a great
many applications of stack in the unit.

The open end of the stack is called the top of the stack. The store and retrieval operations for a
stack are called **PUSH** and **POP**, respectively. Fig. 2 shows how a sample stack evolves through
the series of PUSH and POP represented by the sequence:

\[
A*SAM*P*L*ES*T***A*CK**
\]

Each letter in this list means “PUSH” (the letter); each asterisk means “POP”. A “PUSH”
operation on an object places it on the top of the stack while a “POP” operation removes the top
most object on the stack for carrying out some operation or other.

In Fig. 2, there are 16 columns, including the first column as the first column, which contains
the labels of the 4 rows. In first row we have a symbol, a character or an asterisk. In the next 2
rows, we have the action initiated by the symbol, either a PUSH or a POP operation. In the
fourth row, we have the status of the stack after the operation. For example, in the first row the

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>*</th>
<th>S</th>
<th>A</th>
<th>M</th>
<th>P</th>
<th>L</th>
<th>E</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>A</td>
<td></td>
<td>S</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>A</td>
<td>S</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>P</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>A</td>
<td>M</td>
<td>L</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>S</td>
<td></td>
<td>S</td>
<td>A</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>S</td>
<td>M</td>
<td>A</td>
<td>S</td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2: An illustration of stack operations.

second column, we have ‘A’. Since this pushed A into the stack, we have the character ‘A’ in the
second row. Nothing has been popped, so the third row is empty. The last row shows the stack
with the character ‘A’. In the third column, we have the asterisk in the first row, which indicates
a POP operation. The row corresponding to POP operation has ‘A’, since ‘A’ has been popped.
In this last row we have an empty stack since we have popped the only character in the stack.
Now you can study the remaining columns to understand the PUSH and POP operations.

Another fundamental restricted-access data structure is called the Queue. Again, only two basic
operations are involved; one can insert an item into the Queue at the beginning and remove an

![Fig. 1: Depicting stacks.](image-url)
item from the end. Perhaps our busy executive’s ‘in’ box should operate like a Queue, since then work that arrives first would get done first. In a stack, something can get buried at the bottom, but in a Queue everything is processed in the order received. Queues obey a “First In First Out (FIFO)” discipline. We may draw Queue in any one of the forms given in Fig. 3. Queue is marked with two open ends called front and rear. In the next section, we will discuss implementation of stacks.

13.3 STACK OPERATIONS AND IMPLEMENTATIONS

Basic operations on stack are as follows:
1) Create a stack.
2) Check whether a stack is empty.
3) Check whether a stack is full.
4) Initialise a stack.
5) Push an element onto a stack (if stack is not full).
6) Pop an element from a stack (if stack is not empty).
7) Read a stack top.
8) Print the entire stack.

Stack (a special case of list) can be implemented as one of the following data structures:

- Array
- Linked list

13.3.1 Array Implementation

The simplest way to represent a stack is by using a one-dimensional array, say stack[N] with room for N elements. The first element will be stack[1], and so on. An associated variable top points to the top element of the stack. Type definition for a sequentially allocated stack is

```c
#define STACK_SIZE_MAX 100 /* Maximum stack size.*/
typedef struct {int key;
}element; element
stack[STACK_SIZE_MAX];
int top = -1; /* Denotes an empty stack. */
```
To check whether the stack is empty, we just need to check the value of top. If it is empty, the value of top is -1 and the function returns 0; otherwise it returns 1.

```c
int stackempty()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
```

Given a sequentially allocated stack, and a value to be pushed, this procedure makes the new top of the stack to be that value.

```c
void add (int *top, element item)
{
    if (*top >= STACK_SIZE_MAX - 1) {
        Error("Stack overflow!");
        return;
    }
    stack[++*top]=item;
}
```

13.3.2 Pointer Implementation

Although this method of allocating storage is adequate for many applications, there are many other applications where the sequential allocation method is inefficient and therefore not acceptable.

For such applications, we store a stack element in a structure with a pointer to the next lower element on the stack.

```c
typedef struct node {
    int data;
    struct node *next;
} Item;
```

Suppose S is a pointer to topmost node in the stack. Here is a function to check if the stack is empty.

```c
int IsEmpty(Item * S)
{
    if (S != NULL)
        return 0;
    else
        return 1;
}
```

PUSH, POP and TOP operations involve inserting, deleting and reading item at the top of this list structure.

Here is how we carry out the Pop operation.

```c
void Pop(Item ** S)
{
    Item *current, *FirstCell;
    current = *S;
```
if (IsEmpty(*S)) {
    printf("Empty Stack");
    return;
} else {
    FirstCell = current;
    *S = current->next;
    free(FirstCell);
}

We leave the rest of the operations as exercises to you.

E2) Write C functions for carrying Push and Top operations. Also, write a function that prints the contents of the stack. Write a small C function that
   a) Pushes 4, 5 and 7 into the stack.
   b) Prints the contents of the stack.
   c) Pops the stack.
   d) Prints the contents of the stack again.

In the next section, we will see some applications of stacks.

### 13.4 STACK APPLICATIONS

Stacks are simple structures that figure prominently in many algorithms.

Many algorithms implement basic stack operations in hardware because they naturally implement function call mechanisms: Save the current environment on entry to a procedure by pushing information onto a stack, restore the environment on exit by using information popped from the stack. Some calculators and some computing languages base their method of calculations on stack operations explicitly: Every operation pops its arguments from the stack and returns its results to the stack. In this section, we shall consider two of the many applications of stacks.

#### 13.4.1 Infix to Postfix Conversion

You are already familiar with infix notation where the operator is placed between the operands as in the expression $2 + 3$. Here the operator $+$ is placed between the operands 2 and 3. When we have an expression like $2 + 3 \cdot 5$, we first evaluate $3 \times 5 = 15$ and add it to 2, because multiplication and division have higher priority than addition and subtraction. If we want to change priorities we use brackets; in $(2 + 3) \cdot 5$, we first add 2 and 3 and add multiply the result by 5.

Polish logician Łukasiewicz invented a postfix notation for writing expressions without brackets in 1920s. In a postfix notation, we write the operator after the operands as in $2, 3, +$. (Usually the operators are separated by spaces in postfix notation. Here, we are using comma to increase clarity.) The Reverse Polish Notation was invented by Charles Hamblin in mid 1950s. We can convert any notation in infix notation to this notation in an unambiguous way. Converting an expression in infix notation to RPN is an interesting application of stacks.

Before we proceed further, let us see how we can evaluate an expression in RPN. Consider the following expression in RPN:

$$3, 5, -2, 3, +, /, 5, +$$ (1)

The rule is
1. We read the expression from left to write till we reach an operator.

2. Apply the operator to the two operands immediately preceding the operator.

3. Replace the operator and the two operands by the answer and continue reading right.

When we apply this rule, this is how we will evaluate the expression in eqn. (1) on the preceding page. The details are in Table 1.

Table 1: Evaluation of an expression in RPN notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,5−2,3,+/5,+</td>
<td>The first operator we encounter is −. Apply it to the two operands 3 and 5. Replace 3, 5 and − by −2 in the expression.</td>
</tr>
<tr>
<td>−2,2,3,+/5,+</td>
<td>The next operator is +. Apply it to 2 and 3 and replace 2, 3 and + by 5 in the expression.</td>
</tr>
<tr>
<td>−2,5,+/5,+</td>
<td>The next operator is /. Replace −2, 5 and / by ( -\frac{2}{5} ) in the expression.</td>
</tr>
<tr>
<td>( \frac{2}{5},5,+ )</td>
<td>The next operator is +. Apply it.</td>
</tr>
<tr>
<td>5−( \frac{2}{5} )=( \frac{23}{5} )</td>
<td>There are no operators left. Answer is ( \frac{23}{5} ).</td>
</tr>
</tbody>
</table>

How do we convert an expression in infix notation to RPN? Consider the expression \( 5 + (3 - 5)/(2 + 3) \). We first convert the expressions in brackets to reverse polish notation. The expression becomes \( 5 + [3, 5, -]/[2, 3, +] \). We treat the converted forms in square brackets to be operands.

There are no more expressions in round brackets to be converted. Now, we apply the priority rules. / has higher priority than +, so the expression becomes \( 5 + [3, 5, -]/[2, 3, +] \). In the next step, we convert this to \( [5, [3, 5, -], [2, 3, +]/], +] \). Now, we write the expression without square brackets.

\[ 5, 3, 5, -, 2, 3, +, /, + \]

Here is an exercise for you.

**E3)** Evaluate the expression \( 5, 3, 5, -, 2, 3, +, /, + \) and check that its value is the same as \( 5 + (3 - 5)/(2 + 3) \).

We may write a general algorithm as follows:

1) Initialise the stack to be empty.

2) For each character in the input string, if it is an operand, append it to the output. If it is an operator that has higher precedence than the operator on the top of the stack or if the stack is empty, push it onto the top of the stack. If the incoming operator has the same or lower precedence than the operator on the stack, pop the stack and append it to the output. Repeat this process till the operator on the top of the stack has lower precedence than the incoming operator or the stack is empty. After this, add the incoming operand to the top of the stack.

3) If the input end is encountered, pop the elements in the stack one by one and append them to the output.

Let us write a C programme that converts an infix expression to RPN. We will put some restrictions on the infix expression to keep our program simple. We will assume...
1. The numbers are single digit numbers and negative integers are not allowed.

2. The expression does not contain round brackets; we will do the conversion purely according to priority. Also, we will assume that the expression does not contain the division operator /.

Here is the program; this uses the stack functions we defined earlier.

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAX_STRING 20
typedef struct node {
    int data;
    struct node *next;
} Item;
int IsEmpty(Item * S);
void EmptyStack(Item ** S);
void Push(int x, Item ** S);
int Top(Item * S);
void Pop(Item ** S);
void print_stack(Item * S);
void Error(char *message);
int ishigher(char op1, char op2);
int main()
{
    Item *S=NULL;
    char c,op, out[MAX_STRING];
    int i=0,j=0;
    while((c=getchar()) != '\n'){\n        if (c == ' ')
            continue;
        if(isdigit(c))
            out[i++]=c;
        else if(IsEmpty(S))
            Push(c,&S);
        else{
            while(!IsEmpty(S) && ishigher(Top(S),c)){
                out[i++]=Top(S);
                Pop(&S);
            }
            Push(c,&S);
        }
    }
    while(!IsEmpty(S)){
        out[i++]=Top(S);
        Pop(&S);
    }
    out[i]= '\n';
    for(j = 0; j <= i-1; j++)
        printf("%c",out[j]);
    return(0);
}
```

Listing 13.1: A program to convert an infix expression to RPN.

We have used the function isdigit() from the standard C library which is defined in <ctype.h> to checks whether the character input is a digit or a non-digit. The only new thing in the program is the function ishigher() which checks the priority of the operators and
returns 1 or 0 depending on whether the first operator has higher priority than the second operator or not. We ask you to write such a function in the next exercise.

E4) Write a function `ishigher()` that checks the priority of operators as described above.

We can also use stacks to evaluate expressions in postfix notation. To do this, when an operand is encountered, it is pushed onto the stack. When an operator is encountered, it is applied to the first two operands that are obtained by popping the stack and the result is pushed onto the stack. For example, the postfix expression

\[ 853 + 9 \ast + 4 + \]

is evaluated as follows: On reading 8, 5 and 3 the stack contents are as follows:

```
<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>On reading</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>and added</td>
<td>8</td>
</tr>
<tr>
<td>The result</td>
<td>8</td>
</tr>
<tr>
<td>8 = 5 + 3</td>
<td>is</td>
</tr>
<tr>
<td>pushed onto</td>
<td>8</td>
</tr>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>Next, 9</td>
<td>8</td>
</tr>
<tr>
<td>is pushed</td>
<td>8</td>
</tr>
<tr>
<td>onto the</td>
<td>8</td>
</tr>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>On reading</td>
<td>8</td>
</tr>
<tr>
<td>∗</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>9 ∗ 8 = 72</td>
<td>is</td>
</tr>
<tr>
<td>pushed onto</td>
<td>8</td>
</tr>
<tr>
<td>the stack</td>
<td></td>
</tr>
<tr>
<td>On finding</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>80</td>
</tr>
<tr>
<td>72 and 8</td>
<td>are</td>
</tr>
<tr>
<td>popped out</td>
<td>80</td>
</tr>
<tr>
<td>and 72 + 8 = 80 is pushed onto the stack</td>
<td></td>
</tr>
<tr>
<td>Now 4 is pushed onto the stack</td>
<td>80</td>
</tr>
<tr>
<td>Finally, a + is read and 4 and 80 are popped, the result 4 + 80 is pushed onto the stack</td>
<td>84</td>
</tr>
</tbody>
</table>
```

End of the string is encountered. Therefore, stack is popped and 84 is the result.

E5) Write a function that evaluates an expression in RPN that is given as string. You can make all the assumptions that we made in the program we wrote for converting an infix expression to an expression in RPN.
In the next section, we will discuss Queues.

13.5 QUEUES: OPERATIONS AND IMPLEMENTATION

In a multiuser system, there will be requests from different users for CPU time. The operating system puts them in queue and they are disposed on FIFO (First In, First Out) basic. We will discuss the queue data structure and the operations that it allows in this section. Similar to stack operations, operations that can be carried out on a queue are:

1) Create a queue.
2) Check whether a queue is empty.
3) Check whether a queue is full.
4) Add item at the rear of the queue (enqueue).
5) Remove item from front of queue (dequeue).
6) Read the front of the queue.
7) Print the entire queue.

As we did in the case of stacks, we can give an array representation of a queue. We define a queue as a structure containing the array and two variables, front and rear to denote the present position of its front and rear elements.

We may define a queue as follows:

```c
const max = 100;
typedef struct q_type{
    elementtype queue[max];
    int front, rear;
}Qtype;
```

As an example of this representation of a queue, consider a queue of size 6. Assume that queue is initially empty. We want to insert elements RED, BLACK and BLUE, delete RED and BLACK and insert GREEN, WHITE and YELLOW.

Following figure gives a trace of the queue contents for this sequence of operations:

```
RED
RED
RED
RED
RED
BLACK
BLACK
BLACK
BLACK
BLUE
BLUE
BLUE
BLUE
GREEN
GREEN
GREEN
GREEN
WHITE
WHITE
WHITE
YELLOW
```

Now, if we try to insert ORANGE, an overflow occurs even though the first two cells are free. To avoid this drawback, we can arrange these elements in a circular fashion with QUEUE[0] following QUEUE[N-1]. It is then called a circular array representation. We may depict a circular queue as shown in Fig. 4 on the next page. We also initialise the values of q.front
and \( q->\text{rear} \) to \( \text{max}-1 \). So, initially, when the queue is empty, \( q->\text{front} \) and \( q->\text{rear} \) have the same value.

The procedure for checking whether a circular queue is empty and for inserting and elements in a circular queue are given below:

```c
int empty(Qtype *q)
{
    return((q->front == q->rear)?1:0);
}
void qinsert(Qtype *q, int x)
{
    int newrear;
    if (q->rear == max-1)
        newrear = 0;
    else
        newrear = q->rear + 1;
    if (newrear == q->front);
        Error("QUEUE OVERFLOW");
    else{
        q->rear=newrear;
        q->queue[q->rear]=x;
    }
}
int qdelete(Qtype *q)
{
    if (empty(q)){
        Error("Underflow!");
        exit(1);
    }
    if (q->front == max-1)
        q->front = 0;
    else
        (q->front)++;
    return(q->queue[q->front]);
}
```

Queues are important in simulation models. They serve several purposes like repositories for scheduled events, holding areas for entities moving through the system etc.

Similarly, we may write procedures for other operations on queues.

The second approach for implementing queues is by using the dynamic storage allocation through the use of pointers. We can define a queue consisting of records where each record contains a pointer to the record that comes after it. Therefore, we may declare a queue as

```c
struct qrec{
    elementtype data;
    struct qrec *next;
};
struct qrec *qptr
```

We must also specify the items at the front and rear of the queue. This may be done by the following declaration.

```c
struct qtype{
    struct qrec *front;
}```
This kind of an implementation is a singly linked list implementation of the queue. Recall the Queue implementation using circular arrays. Similarly, we can implement queues using circular lists. In this list, each node points to the next and the chain of pointers eventually form a loop back to the first one.

In such a case the declaration becomes simplified as follows:

```c
struct qrec {
    elementtype data;
    struct qtype next;
};
struct qrec *qtype;
struct qrec *q;
```

You may notice that circular list implementation requires special attention for insertion and deletion with no elements or with just one element.

### 13.6 PRIORITY QUEUES

Many applications involving queues require priority queues rather than simple FIFO strategy. Each queue element has a associated priority value and the elements are served on a priority basis instead of using the order of arrival. For elements of same priority, the FIFO order is used. For example, in a multi-user system, there will be several programmes competing for use of the central processor at the same time. The programs have a priority value associated to them and are held in a priority queue. The program with the highest priority is given the first use of the central processor.

Scheduling of jobs within a time-sharing system is another application of queues. In such a system many users may request processing at a time and computer time is divided among these requests. The simplest approach sets up one queue that store all requests for processing. Computer processes the request at the front of the queue and finishes it before starting on the next. Same approach is also used when several users want to use the same output device, say a printer.

In a time sharing system, another common approach used is to process a job only for a specified maximum length of time. If the program is fully processed within that time, then the computer goes on to the next process. If the program is not completely processed within the specified time, the intermediate values are stored and remaining part of the program is put back on the queue. This approach is useful in handling a mix of long and short jobs.

### 13.7 SUMMARY

A stack is a list in which retrievals, insertion and deletions take place at the same position. It follows the last in first out (LIFO) mechanism. Compiler implements recursion by generating code for creating and maintaining an activation stack, i.e. a run time stack that holds the state of each active subprogram. A queue is a list in which retrievals and deletions can take place at one end and insertions occur at another end. In follows first in, first out (FIFO) order.

Queues are employed in many situations. The items on queues may be vehicles waiting at a crossing, cars waiting at the service station, customers in a cinema ticket counter etc.

### 13.8 SOLUTIONS/ANSWERS
E1) `void POP(int *top)`
```c
{  
    if (stackempty())  
        Error("No element to POP.");  
    else  
        --*top;  
}
```

E2) Function for Push operation
```c
`void Push(int x, Item **S)`
```
```c
{  
    Item *tmp;  
    tmp = malloc(sizeof(Item));  
    if (tmp == NULL) {  
        Error("Out of space!");  
        exit(1);  
    } else {  
        tmp->data = x;  
        tmp->next = *S;  
        *S = tmp;  
    }
}
```

Function for Top operation.
```c
`int Top(Item *S)`
```
```c
{  
    if (!IsEmpty(S))  
        return S->data;  
    printf("Empty Stack!");  
    return 0;  
}
```

Function for printing the stack.
```c
`void print_stack(Item *S)`
```
```c
{  
    if (IsEmpty(S))  
        printf("The stack is empty.");  
    else {  
        printf("Printing elements in the stack...
");  
        while (S->next != NULL) {  
            printf("%d
", S->data);  
            S = S->next;  
        }  
        printf("%d
", S->data);  
    }
}
```

A program that creates the stack and carries out stack operations.
```c
`void Error(char *message)`
`int main()`
```c
{  
    Item *myStack = NULL;  
    Push(4, &myStack);  
    Push(5, &myStack);  
    Push(7, &myStack);  
    print_stack(myStack);  
    Pop(&myStack);  
    print_stack(myStack);  
    printf("Element on top is \n%d", Top(myStack));  
    return 0;  
}
E3) `int isigher(char op1, char op2)
   {
      if ((op1 == '+' || op1 == '-') && (op2 == '*'))
         return(0);
      else
         return(1);
   }`

E4) /*Program-13.3. A program to evaluate an
expression in RPN. File name: unit13-evaluate-RPN.c*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
define MAX_STRING 20
typedef struct node {
   int data;
   struct node *next;
} Item;
int IsEmpty(Item * S);
void EmptyStack(Item ** S);
void Push(int x, Item ** S);
int Top(Item * S);
void Pop(Item ** S);
void print_stack(Item * S);
void Error(char *message);
int isigher(char op1, char op2);
char *infix2rpn(char *input);
ingeval(char op, int a, int b);
int main()
{
   int i,j,op1,op2;
   char out[MAX_STRING];
   Item *myStack = NULL;
infix2rpn(out);
   printf("RPN form is\n");
   for(i=0;out[i] != '\n';i++)
      printf("%c",out[i]);
   printf("\n");
   for(i = 0; out[i] != '\n';i++)
      if(isdigit(out[i]))
         Push(out[i]'-0',&myStack);
      else
      {
         op1= Top(myStack);
         Pop(&myStack);
         op2= Top(myStack);
         Pop(&myStack);
         j=eval(out[i],op1,op2);
         Push(j,&myStack);
      }
   printf("Value is %d",Top(myStack));
   return 0;
}
int IsEmpty(Item * S)
{
   if (S != NULL)
      return 0;
   else
      return 1;
}
void EmptyStack(Item ** S)
{
   Item *current = *S;
if (S == NULL)
    printf("Error! No stack to empty!\n");
else
    while (!IsEmpty(current))
        Pop(S);
)
void Push(int x, Item ** S)
{
    Item *tmp;
    tmp = malloc(sizeof(Item));
    if (tmp == NULL) {
        Error("Out of space!\n");
        exit(1);
    } else {
        tmp->data = x;
        tmp->next = *S;
        *S = tmp;
    }
}
int Top(Item * S)
{
    if (!IsEmpty(S))
        return S->data;
    printf("Empty Stack!\n");
    return 0;
}
void Pop(Item ** S)
{
    Item *current, *FirstCell;
    current = *S;
    if (IsEmpty(*S)) {
        printf("Empty Stack\n");
        return;
    } else {
        FirstCell = current;
        *S = current->next;
        free(FirstCell);
    }
}
void print_stack(Item * S)
{
    if (IsEmpty(S))
        printf("The stack is empty.\n");
    else {
        printf("Printing elements in the stack..\n\n");
        while (S->next != NULL) {
            printf("%d\n", S->data);
            S = S->next;
        }
        printf("%d\n", S->data);
    }
}
void Error(char *message)
{
    fprintf(stderr,"Error! %s\n",message);
}
int ishigher(char op1, char op2)
{
    if( (op1 == '+') || op1 == '-' ) && (op2 == '*')
        return(0);
    else
        return(1);
}
char *infix2rpn(char *out)
{
    Item *S=NULL;
    char c;
    int i=0;
    while((c=getchar()) != '\n'){
        if (c == ' ')
            continue;
        if(isdigit(c))
            out[i++]=c;
        else if(IsEmpty(S))
            Push(c,&S);
        else{
            while(!IsEmpty(S) && ishigher(Top(S),c)){
                out[i++]=Top(S);
                Pop(&S);
            }
            Push(c,&S);
        }
    }
    while(!IsEmpty(S)){
        out[i++]=Top(S);
        Pop(&S);
    }
    out[i]=\n';
    return(0);
}
int eval(char op, int a, int b)
{
    switch (op){
        case('+'):
            return(a + b);
        case('-'):
            return(b - a);
        case('*'):
            return(a*b);
    }
}
UNIT 14  TREES

Structure

14.1 Introduction  
   Objectives 59
14.2 Basic Terminology 59
14.3 Binary Trees 64
   Inorder Traversal
   Post order Traversal
   Preorder Traversal
   Level by Level Traversal
14.4 Binary Search Trees 66
   Operations on a BST
   Insertion in Binary Search Tree
   Deletion of a node in BST
   Search for a key in BST
14.5 Summary 73
14.6 Solutions/Answers 73

14.1 INTRODUCTION

In the previous block we discussed Arrays, Lists, Stacks and Queues. In this we will discuss trees. The concept of trees is one of the most fundamental and useful concepts in computer science. Trees have many variations, implementations and applications. Trees find their use in applications such as compiler construction, database design, windows, operating system programs, etc. What is a tree? A tree structure is one in which items of data are related by edges. More formally, a Tree is a particular kind of graph, an acyclic, connected graph. A Tree contains no loops or cycles. If all these are Greek and Latin to you, do not worry. To make the discussion and definition of trees understandable, we will discuss graphs briefly in this Unit. You will study graphs in greater detail in MMTE-001, Graph theory course in the 3rd semester. In this Unit our attention will be restricted to rooted trees. In Sec. 14.2, we will introduce you to the basic terminology related to trees. In Sec. 14.3 we will discuss binary trees, a special type of trees. In Sec. 14.4, we will discuss how to traverse a binary tree. In Sec. 14.5, we will see how to search a binary tree.

Objectives

After studying this unit, you should be able to

• define a tree, a rooted tree, a binary tree, and a binary search tree
• differentiate between a general tree and a binary tree
• describe the properties of a binary search tree
• write programs for insertion, deletion and searching of an element in a binary search tree
• show how an arithmetic expression may be stored in a binary tree
• build and evaluate an expression tree
• write programs for preorder, in order, and post order traversal of a tree

14.2 BASIC TERMINOLOGY

Before we formally discuss trees formally, we will discuss some common examples of trees in an informal way. Trees are encountered frequently in everyday life. An example is found in the
organisational chart of a large corporation. Computer Science in particular makes extensive use of trees. For example, in databases it is useful in organising and relating data. It is also used for scanning, parsing, generation of code and evaluation of arithmetic expressions in compiler design.

A very common example is the ancestor tree as given in Fig. 1. This tree shows the ancestors of LAKSHMI. Her parents are VIDYA and RAMKRISHNA; RAMKRISHNA’S PARENTS are SUMATHI and VIJAYANANDAN who are also the grand parents of LAKSHMI (on father’s side); VIDYA’S parents are JAYASHRI and RAMAN and so on.

![Family Tree I](image1)

We can also have another form of ancestor tree as given in Fig. 2.

![Family Tree II](image2)

We could have also generated the image of tree in Fig. 1 as in Fig. 3.

![Family Tree III](image3)

All the above structures are called rooted trees. A tree is said to be rooted if it has one node, called the root that is distinguished from the other nodes. In Fig. 1, the root is LAKSHMI, in Fig. 2 the root is KALYANI and in Fig. 3 the root is LAKSHMI. We usually draw trees with the root at the top. Each node (except the root) has exactly one node above it, which is called its parent; the nodes directly below a node are called its children. We sometimes carry the analogy to family trees further and refer to the grandparent or the sibling of a node.

Let us now discuss some basic concepts in graph theory to prepare the ground for the study of trees. We start with a formal definition of a graph.

**Definition 2**: A (simple) graph $G$ consists of a set $V$ of vertices (or nodes) and a set $E$ of edges (or arcs). We write $G = (V, E)$ where $V$ is a finite non-empty set of vertices. $E$ is a subset of $V \times V$, the set of (unordered) pairs of elements in $V$.

Therefore, $V(G)$, read as ‘$V$ of $G$’ is the set of vertices and $E(G)$, read as ‘$E$ of $G$’ is the set of edges. An edge $e = (v, w)$ is a pair of vertices $v$ and $w$, and is said to be incident with $v$ and $w$. A graph may be pictorially represented as in Fig. 4.
We have numbered the nodes as 1, 2, 3, 4 and 5. So,

\[ V(G) = \{1, 2, 3, 4, 5\} \]

and

\[ E(G) = \{(1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (1, 3), (3, 5)\} \]

You may notice that we wrote the edge incident with node 1 and node 5 as \((1, 5)\); we could have also written \((5, 1)\) instead. The same applies to all edges. Here, we do not attach ordering of the vertices. This is an unordered graph or a simple graph.

**Definition 3:** By a subgraph of a graph \((V(G), E(G))\), we mean a graph \((V(H), E(H))\), where \(V(H) \subset V(G)\) and \(E(H) \subset E(G)\).

For example, the graph in Fig. 5 is a subgraph of the graph in Fig. 4 with \(V(H) = \{3, 4, 5\}\) and \(E(H) = \{(3, 4), (3, 5), (4, 5)\}\).

We can also attach importance to the order of the vertices. We then get an ordered graph. In this, each vertex is represented by an ordered pair. So, we consider \((1, 5)\) and \((5, 1)\) as different edges. We can represent a directed graph pictorially as in Fig. 6.

We indicate the direction by an arrow. The set of vertices for this graph remains the same as that of the earlier example, i.e.

\[ V(G) = \{1, 2, 3, 4, 5\} \]

However, the set of edges would be

\[ E(G) = \{(1, 2), (2, 3), (3, 4), (5, 4), (5, 1), (1, 3), (5, 3)\} \]

Did you notice the difference? Also, note that arrow is always from tail vertex to head vertex. In our further discussion on graphs, we will refer to directed graphs as digraphs and undirected graphs as simply graphs.

**Definition 4:** Two vertices, \(v\) and \(w\), in a graph are adjacent if \((v, w)\) is in \(E(G)\). In the case of digraphs, \(v\) and \(w\) are adjacent if either \((v, w)\) or \((w, v)\) is in \(E(G)\).
Data Structures

In Fig. 7, vertices 1 and 2 are adjacent, but 1 and 4 are not adjacent.

**Definition 5:** A path from a vertex v to a vertex w is a sequence of vertices starting with v and ending in w with each vertex adjacent to the next. v is called the starting vertex and w is called the end vertex. We say that vertices in the sequences lie on the path or simply on the path joining the vertices v and w. A path in which the starting vertex and the end vertex are the same is called a cycle.

In Fig. 6, 1, 3, 4 is a path joining 1 and 4. In Fig. 4 on the preceding page, 5, 4, 3, 2 is a path joining 5 and 2. In Fig. 4 on the facing page, 1, 2, 3, 1 is a cycle.

Notice that, in Fig. 4, we can always find a path joining any two vertices. Such graphs are called connected graphs. If there are two vertices in a graph which are not connected by path, we say that the graph is disconnected. For example the graph in Fig. 7 is disconnected.

Notice that, although the graph in Fig. 7 looks like two different graphs, it is a single graph with \( V(G) = \{1, 2, 3, 4\} \) and \( E(G) = \{(1, 2), (3, 4)\} \). In this graph there is no path connecting 1 and 4. Also, you can see that the graph has two ‘pieces’. They are called connected components of the graph. A connected component of a graph is a maximal connected subgraph of a graph. In other words, a connected component must be a connected graph and it should not be a proper subgraph of any other connected subgraph of G. For example, \( V(G_1) = \{1, 2\} \), \( E(G_1) = \{(1, 2)\} \) is a connected component of G.

**Definition 6:** The degree of a vertex of a graph is the number of edges incident on the vertex v. In the case of a digraph, the in degree of a vertex v is the number of edges that end in v and the out degree is the number of vertices that start in v.

For example, in Fig. 4, the degree of 1 is three and the degree of 3 is four.

**Definition 7:** A graph is called a tree if it is connected and does contain any cycle.

![Fig. 8](image)

Fig. 8 is an example of a tree.

In this Unit, we will discuss only a special class of trees called rooted trees. In what follows, we will also use the term node for the vertex of a tree and branch for the edge of a tree.

In Fig. 2, root is KALYANI. The three sub-trees are rooted at BABU, RAJAN and JAYASHRI. Sub-trees with JAYASHRI as root has two sub-trees rooted at SUKANYA and VIDYA and so on. The nodes of a tree have a parent-child relationship.

The root does not have a parent; but each one of the other nodes has a parent node associated to it. A node may or may not have children, i.e. it may be of degree 1. A node that has no children is called a leaf node.
If a tree has \( n \) nodes, one of which is the root then there would be \( n - 1 \) branches. It follows from the fact that each branch connects some node to its parent, and every node except the root has one parent. Nodes with the same parent are called siblings. Consider the tree given in Fig. 9.

K, L, and M are all siblings. B, C, D are also siblings.

There is exactly one path between any two nodes and between the root and each of the other nodes in the tree in particular. If there is more than one path between 2 nodes in a graph, the graph will contain a cycle; the graph will not be acyclic. If there is no path between two nodes, the graph will not be connected. In either case the graph will not be a tree. Nodes with no children are called leaves, or terminal nodes. Nodes with at least one child are sometimes called non-terminal nodes. We sometime refer to non-terminal nodes as internal nodes and terminal nodes as external nodes.

The length of a path is the number of branches (edges) on the path. Further if \( n \) lies on the unique path from the root to \( i \), the \( n \) is an ancestor of \( i \) and \( i \) is a descendant of \( n \). Also there is a path of length zero from every node to itself, and there is exactly one path from the root to each node.

Let us now see how these terms apply to the tree given in Fig. 9. A path from A to K is A-D-G-J-K and the length of this path is 4.

A is ancestor of K and K is descendant of A. All the other nodes on the path are also descendants of A and ancestors of K.

The depth of any node \( n_i \) is the length of the path from the root to \( n_i \). Thus, the root is at depth 0(zero). The height of a node \( n_i \) is the longest path from \( n_i \) to a leaf. Thus all leaves are at height zero. Further, the height of a tree is same as the height of the root. For the tree in Fig. 9, F is at height 1 and depth 2. D is at height 3 and depth 1. The height of the tree is 4. Depth of a node is sometimes also referred to as level of a node.

An acyclic graph which is not connected is called a forest. Each component of such a graph will be a tree; for example, if we remove the root and the edges connecting it from the tree in Fig. 9, we are left with a forest consisting of three trees rooted at A, D and G, as shown in Fig. 10. Let us now list some of the properties of a tree:
Data Structures

Properties of a tree

1. Any node can be root of the tree each node in a tree has the property that there is exactly one path connecting that node with every other node in the tree.

2. Each node, except the root, has a unique parent and every edge connects a node to its parent. Therefore, a tree with N nodes has N-1 edges.

We close the discussion of general trees here. In the next section, we will discuss binary trees.

14.3 BINARY TREES

By definition, a Binary tree is a tree which is either empty or consists of a root node and two disjoint binary trees called the left subtree and right subtree. In Fig. 11 a binary tree T is depicted with a left subtree, L(T) and a right subtree R(T).

A binary tree, no node can have more than two children. So, every node in a binary tree has 0, 1 or no children. Binary trees are special cases of general trees. The terminology we have discussed in the previous section applies to binary trees also.

Let us list the properties of binary trees:

1. Recall from the previous section the definition of internal and external nodes. A binary tree with N internal nodes has maximum of \((N + 1)\) external nodes: Root is considered as an internal node.

2. The external path length of any binary tree with N internal nodes is 2N greater than the internal path length.

3. The height of a full binary tree with N internal nodes is about \(\log_2 N\).

As we shall see, binary trees appear extensively in computer applications, and performance is best when the binary trees are full (or nearly full). You should note carefully that, while every binary tree is a tree, not every tree is a binary tree.

A full binary tree or a complete binary tree is a binary tree in which all internal nodes have degree and all leaves are at the same level. The Fig. 11 illustrates a full binary tree.
The degree of a node is the number of non-empty sub trees it has. A leaf node has a degree zero.

Implementation
A binary tree can be implemented as an array of nodes or a linked list. The most common and easiest way to implement a tree is to represent a node as a struct consisting of the data and pointer to each child of the node. Because a binary tree has at most two children, we can keep direct pointers to them. A binary tree node declaration in C may look like:

```c
struct tnode {
    struct tnode *left;
    int data;
    struct tnode *right;
};
typedef struct tnode Tnode;
ttypedef TNodePtr;
```

Let us now consider a special case of binary tree. It is called a 2-tree or a strictly binary tree. It is a non-empty binary tree in which either both sub trees are empty or both sub trees are 2-trees. For example, the binary trees in Fig. 13a and Fig. 13b are 2-trees, but the trees in Fig. 13c and 13d are not 2-trees.

![Binary Trees](image)

**Fig. 13**

Binary trees are most commonly represented by linked lists. Each node can be considered as having 3 elementary fields: a data field, left pointer, pointing to left sub tree and right pointer pointing to the right sub tree.

![Linked List Representation of a Binary Tree](image)

**Fig. 14:** Linked list representation of a Binary Tree

Fig. 14 contains the linked storage representation of a binary tree Fig. 11. A binary tree is said to be complete (See Fig. 11) if it contains the maximum number of nodes possible for its height. In a complete binary tree:

1. The number of nodes at level 0 is 1.
2. The number of nodes at level 1 is 2.
3. The number of nodes at level 2 is 4, and so on.
4. The number of nodes at level i is \(2^i\). Therefore for a complete binary tree with \(k\) levels contains \(\sum_{i=0}^{k} 2^i\) nodes.

E6) How many different trees are there with three nodes? Draw each.

E7) Give level, degree and height of each node of the tree in Fig. 15.
We conclude this section here. In the next section, we will take up tree traversal.

### 14.4 TRAVERSALS OF A BINARY TREE

By a **traversal** of a graph is to visit each node exactly once. In this section we shall discuss traversal of a binary tree. It is useful in many applications. For example, in searching for particular nodes. Compilers commonly build binary trees in the process of scanning, parsing, generating code and evaluation of arithmetic expression. Let T be a binary tree. There are a number of different ways to proceed. The methods differ primarily in the order in which they visit the nodes. The four different traversals of T are In order, Post order, Preorder and Level-by-level traversal.

#### 14.4.1 In order Traversal

It follows the general strategy of Left-Root-Right. In this traversal, if T is not empty, we first traverse

1. the left sub tree and recursively traverse in order this tree;
2. then visit the root node of T; and
3. then traverse (in order) the right sub tree.

Consider the binary tree given in Fig. 11. Let us see how this is traversed in in order traversal.

1. The root node is A. In in order traversal, we first go to node B, the node to the left of A.
2. Recursively, B is now the root node, so we go to D. The node traversed so far are B, D.
3. D is a leaf node. So we have to go to the right node of B. B does not have a right node.
   We have exhausted all the descendents of B and so we go to the root of B which is A. The nodes traversed are D, B, A.
4. Now, we start traversing from C. C is the root node which is a left node. We go to its left node E. The nodes traversed are D, B, A, E.
5. E is a leaf node. So, we go to root node which is C. The nodes traversed so far are D, B, A, E, C.
6. Next, we go G. G is now the root node with a left descendent H. We go there.D, B, A, E, C, H.
7. G does not have a descendent on the right. So, we go to G. The nodes traversed are D, B, A, E, C, H, G.
8. From G, we move up to its root node C. We have exhausted all descendents of C. We go to the root of C, which is A. So, the final list of nodes in the order in which they are traversed is D, B, A, E, C, H, G.

Here is an exercise for you.

E8) Give the order of vertices in which the vertices are traversed in a inorder traversal of the graph in Fig. 13a.
A binary tree can be used to represent arithmetic expressions if the node value can be either operators or operand values and are such that:

1. each operator node has exactly two branches
2. each operand node has no branches, such trees are called expression trees.

Let us traverse this tree in in order traversal.

1. Tree, T, at the start is rooted at ‘−’;
2. Since left(T) is not empty; current T becomes rooted at ‘+’;
3. Since left(T) is not empty; current T becomes rooted at ‘A’.
4. Since left(T) is empty; we visit root i.e. A.
5. We access T’ root i.e. ‘+’.
6. We now perform in order traversal of right(T).
7. Current T becomes rooted at ‘*’.
8. Since left(T) is not empty; Current T becomes rooted at ‘B’ since left(T) is empty; we visit its root i.e. B; check for right(T) which is empty, therefore, we move back to parent tree. We visit its root i.e. ‘*’.
9. Now in order traversal of right(T) is performed; which would give us ‘C’. We visit T’s root i.e. ‘D’ and perform in order traversal of right(T); which would give us ‘*’ and ‘E’.

Therefore, the complete listing is

A + B * C - D * E

You may note that expression is in infix notation. The in order traversal produces a (parenthesized) left expression, then prints out the operator at root and then a (parenthesized) right expression. This method of traversal is probably the most widely used. The following is a C function for in order traversal of a binary tree

```c
void inorder (TNorePtr tptr)
{
    if (tptr != NULL){
        inorder (tptr->left);
        printf(‘\%d’, tptr->data);
        inorder (tptr->right);
    }
}
```

Please notice that this procedure, like the definition for traversal is recursive.

### 14.4.2 Post order Traversal

In this traversal we first traverse left(T) (in post order); then traverse Right(T) (in post order); and finally visit root. It is a Left-Right-Root strategy, i.e.

**Traverse the left sub tree In Post order.**
Traverse the right sub tree in Post order.

Visit the root.

For example, a post order traversal of the tree given in Fig. 16 would be

\[ ABC + \times \neg \]

You may notice that it is the postfix notation of the expression

\[(A + (B \times C)) - (D \times E)\]

We leave the details of the post order traversal method as an exercise. Here is an exercise for you.

E9) Write a C function for post order traversal.

14.4.3 Preorder Traversal

In this traversal, we visit root first; then recursively perform preorder traversal of Left(T); followed by pre order traversal of Right(T) i.e. a Root-Left-Right traversal, i.e.

Visit the root

Traverse the left sub tree preorder.

Traverse the right sub tree preorder.

A preorder traversal of the tree given in Fig. 16 would yield

\[ - + A \times BC \times DE \]

It is the prefix notation of the expression

\[(A + (B \times C)) - (D \times E)\]

Here is an exercise for you.

E10) Write a C function for preorder traversal.

14.4.4 Level by Level traversal

In this method we traverse level-wise i.e. we first visit node root at level ’0’ i.e. root. There would be just one. Then we visit nodes at level one from left to right. There would be at most two. Then we visit all the nodes at level ’2’ from left to right and so on. For example the level by level traversal of the binary tree given in Fig. 11 will yield

\[ A B C D E F G H I J K \]

This traversal is different from other three traversals in the sense that it need not be recursive, therefore, we may use queue kind of a data structure to implement it, while we need stack kind of data structure for the earlier three traversals.

E11) Traverse the tree given in Fig. 17 in preorder, in order, post order and level by level giving a list of nodes visited.
We close this section here. In the next section, we discuss binary search trees.

### 14.5 BINARY SEARCH TREES (BST)

A **Binary Search Tree**, BST, is an ordered binary tree T such that either it is an empty tree or

1. each data value in its left sub tree is less than the root value,
2. each data value in its right sub tree is greater than the root value, and
3. left and right sub trees are again binary search trees.

Fig. 18 depicts a binary search tree, while the one in Fig. 18b is not a binary search tree. (Why?) Clearly, duplicate items are not allowed in a binary search tree. You may also notice that an in order traversal of a BST yields a sorted list in ascending order.

#### 14.5.1 Operations on a BST

We now give a list of the operations that are usually performed on a BST.

1. **Initialization of a BST**: This operation makes an empty tree.
2. **Check whether BST is Empty**: This operation checks whether the tree is empty.
3. **Create a node for the BST**: This operation allocates memory space for the new node; returns with error if no space is available.
4. **Retrieve a node’s data**.
5. **Update a node’s data**.
6. **Insert a node in BST**.
7. **Delete a node (or sub tree) of a BST**.
8. Search for a node in BST.

9. Traverse (in inorder, preorder, or post order) a BST.

We shall describe some of the operations in detail.

### 14.5.2 Insertion in a BST

**Inserting a node to the tree:** To insert a node in a BST, we must check whether the tree already contains any nodes. If tree is empty, the node is placed in the root node. If the tree is not empty, then the proper location is found and the added node becomes either a left or a right child of an existing node. The logic works this way:

```plaintext
add-node(node, value)
{
    if (two values are same)
        duplicate;
        return (FAILURE)
    }
else if (value < value in current node)
    if (left child exists)
        add-node (left child, value);
    else{
        allocate new node and make left child point to it;
        return (SUCCESS);
    }
else if (value > value in current node)
    if (right child exists)
        add-node (right child, value);
    else{
        allocate new node and make right child point to it;
        return (SUCCESS);
    }
}
```

The function continues recursively until either it finds a duplicate (no duplicate strings are allowed) or it hits a dead end. If it determines that the value to be added belongs to the left-child sub tree and there is no left-child node, it creates one. If a left-child node exists, then it begins its search with the sub tree beginning at this node. If the function determines that the value to be added belongs to the right of the current node, a similar process occurs.

Let us consider the BST given in Fig. 19a.
If we want to insert 5 in the BST in Fig. 19a, we first search the tree. If the key to be inserted is found in tree, we do nothing (since duplicates are not allowed), otherwise a nil is returned. In case a nil is returned, we insert the data at the last point traversed. In the example above a search operation will return nil on not finding a right, sub tree of tree rooted at 4. Therefore, 5 must be inserted as a right child of 4.

### 14.4.3 Deletion of a node

Once again the node to be deleted is searched in BST. If found, we need to consider the following possibilities:

(i) If node is a leaf, it can be deleted by making its parent pointing to nil. The deleted node is now unreferenced and may be disposed off. For example, we if we delete the node 4 in Fig. 19a the resulting BST will be the one in Fig. 20

![Fig. 20: Deletion of a Terminal Node](image)

(ii) If the node has one child, its parent’s pointer needs to be adjusted. For example for node 1 to be deleted from BST given in Fig. 19a the left pointer of node 6 is made to point to child of node 1 i.e. node 4 and the new structure would be as in Fig. 21

![Fig. 21: Deletion of a Node with one child](image)

(iii) If the node to be deleted has two children; then the value is replaced by the smallest value in the right sub tree or the largest key value in the left sub tree; subsequently the empty node is recursively deleted. Consider the BST in Fig. 22a on the following page. If the node 6 is to be deleted then first its value is replaced by smallest value in its right subtree i.e. by 7. After we do this, the tree will be as in Fig. 22b on the next page. Now we need to delete this empty node as explained in (iii). Therefore, the final structure would be as in Fig. 22c.

![Fig. 22a on the following page](image)

![Fig. 22b on the next page](image)

![Fig. 22c](image)

### 14.5.3 Search for a key in a BST

To search the binary tree for a particular node, we use procedures similar to those we used when adding to it. Beginning at the root node, the current node and the entered key are compared. If the values are equal success is output. If the entered value is less than the value in the node, then it must be in the left-child sub tree. If there is no left-child sub tree, the value is not in the tree i.e. a failure is reported. If there is a left-child subtree, then it is examined the same way. Similarly, if the entered value is greater than the value in the current node, the right child is searched. Fig. 23 on the following page shows the path through the tree followed in the search for the key H.

![Fig. 23 on the following page](image)
Data Structures

Fig. 22: Removing a key with two children in a BST.

(a)

Fig. 23: Searching for a key in a BST.

(b)

(c)

find-key (key value, node){
    if (two values are same){
        print value stored in node;
        return (SUCCESS);
    }
    else if (key value < value stored in current node){
        if (left child exists)
            {
                find-key (key-value, left child);
            }
        else
            {
                there is no left subtree.,
                return (string not found)
            }
    }
    else if (key-value > value stored in current node){
        if (right child exists)
            {
                find-key (key-value, right child);
            }
        else{
        else{
14.6 SUMMARY

This unit introduced the tree data structure which is an **acyclic, connected, simple** graph. Terminology pertaining to trees was introduced. A special case of general case of general tree, a binary tree was focussed on. In a binary tree, each node has a maximum of two subtrees, left and right subtree. Sometimes it is necessary to traverse a tree, that is, to visit all the tree’s nodes. Four methods of tree traversals were presented in order, post order, preorder and level by level traversal. These methods differ in the order in which the root, left subtree and right subtree are traversed. Each ordering is appropriate for a different type of applications.

An important class of binary trees is a complete or full binary tree. A full binary tree is one in which internal nodes completely fill every level, except possibly the last. A complete binary tree where the internal nodes on the bottom level all appear to the left of the external nodes on that level. Fig. 6a shows an example of a complete binary tree. We conclude this section here. In the next section, we will summarise this unit.

14.7 SOLUTIONS/ANSWERS

E1) The Total number of different trees with 3 nodes are 5. See Fig. 24.

Fig. 24

E2)

<table>
<thead>
<tr>
<th>Node</th>
<th>Level</th>
<th>Degree</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

E3) B, A, F, D, G, C, E

E4) Postorder(TnodePtr tptr)
{
    If (tptr ! = NULL) {
        Postorder (tptr-> left);
        Postorder (tptr-> right);
        printf ('\%d', tptr-> data);
    }
}
E5) Preorder (TnodePtr tptr)
{
    if (tptr ! = NULL) {
        printf ('\%d', data);
        Preorder (tptr->left);
        Preorder (tptr->right);
    }
}

E6) Preorder: *+-EAB+CD
    Inorder: E+A-B*C+D
    Postorder: EA+B-CD + *-
    Level by level: *- + +BCDEA
UNIT 15  FILES

Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1 Introduction</td>
<td>87</td>
</tr>
<tr>
<td>Objectives</td>
<td></td>
</tr>
<tr>
<td>15.2 Terminology</td>
<td>87</td>
</tr>
<tr>
<td>15.3 File Organisation</td>
<td>88</td>
</tr>
<tr>
<td>15.4 Sequential Files</td>
<td>89</td>
</tr>
<tr>
<td>Structure</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>Disadvantages</td>
<td></td>
</tr>
<tr>
<td>Areas of Use</td>
<td></td>
</tr>
<tr>
<td>15.5 Direct File Organisation</td>
<td>90</td>
</tr>
<tr>
<td>15.6 Indexed Sequential File Organisation</td>
<td>91</td>
</tr>
<tr>
<td>15.7 Summary</td>
<td>91</td>
</tr>
<tr>
<td>15.8 Solutions/Answers</td>
<td>92</td>
</tr>
</tbody>
</table>

15.1 INTRODUCTION

In this Unit, we will discuss storage data in the computers. The data is stored in computers in the form of files. We introduce you the basic terminology related to file organisation in Sec. 15.2. In Sec. 15.3, we discuss various ways in which data is organised in files. In Sec. 15.4 to Sec 15.6, we will discuss various kinds of file organisations and their advantages and disadvantages. It will be useful if you recapitulate the units of Block 2 to refresh your knowledge of syntax related to file handling in C and also Unit 14 of this block on Tree structures.

Objectives

After studying this unit, you should be able to

• define the various terms related to files;

• describe the different ways in which data is organised in files; and

• discuss the advantages and disadvantages types of files.

15.2 TERMINOLOGY

We will now define the terms of the hierarchical structure of data collection stored in computers.

1) Field: It is an elementary data item characterised by its size, length and type.
   For example:
   Name : a character type of size 10
   Age: a numeric type

2) Record: It is a collection of related fields that can be treated as a unit from an applications point of view.
   For example:
   A university could use a student record with the fields, university enrolment no., Name
   Major subjects

3) File: Data is organised for storage in files. A file is a collection of similar, related records.
   It has an identifying name.
   For example: “STUDENT” could be a file consisting of student records for all the pupils in a university.
4) **Index:** An index file corresponds to a data file. It’s records contain a key field and a Pointer to that record of the data file which has the same value of the key field.

Indexing will be discussed in detail later in the unit.

The data stored in files is accessed by software which can be divided into the following two categories:

1) **User Programs:** These are usually written by a Programmer to manipulate retrieved data in the manner required by the application.

2) **File Operations:** These deal with the physical movement of data in and out of files. User programs effectively use file operations through appropriate programming language syntax. The File Management system manages the independent files and acts as the software interface between the user programs and the file operations.

File operations can be categorised as-

i) **CREATION** of the file

ii) **INSERTION** of records in the file

iii) **UPDATION** of previously inserted records

iv) **RETRIEVAL** of previously inserted records

v) **DELETION** of records

vi) **DELETION** of the file.

---

### 15.3 FILE ORGANISATION

File organisation can most simply be defined as the method of storing Data record in a file and the subsequent implications on the way these records can be accessed. The factors involved in selecting a particular file organisation for use are:

- Ease of retrieval
- Convenience of updates
- Economy of storage
- Reliability
- Security
- Integrity

Different file organisations accord the above factors differing weightages. The choice must be made depending upon the individual needs of the particular application in question.

We now introduce in brief the various commonly encountered file organisations.

1) **Sequential Files**

   Data records are stored in some specific sequence e.g. order of arrival value of key field etc. Records of a sequential file cannot be accessed at random i.e. to access the nth record, one must traverse the preceding \((n - 1)\) records. Sequential files will be dealt with at length in the next section.

2) **Relative Files**

   Each data record has a fixed place in a relative file. Each record must have associated with it in integer key value that will help identify this slot. This key, therefore, will be used for insertion and retrieval of the record. Random as well as sequential access is possible.

Relative files can exist only on random access devices like disks.
3) **Direct Files**
These are similar to relative files, except that the key value need not be an integer. The user can specify keys which make sense to her application.

4) **Indexed Sequential Files**
An index is added to the sequential file to provide random access. An overflow area needs to be maintained to permit insertion in sequence.

5) **Indexed Files**
In this file organisation, no sequence is imposed on the storage of records in the data file, therefore, no overflow area is needed. The index however, is maintained in strict sequence. Multiple indexes are allowed on a file to improve access.

## 15.4 SEQUENTIAL FILES

We will now discuss in detail the sequential file organisation as defined in Sec. 15.2. Sequential files have data records stored in a specific sequence.

A sequentially organised file may be stored on either a serial-access or a direct-access storage medium.

### 15.4.1 Structure

To provide the “sequence” required a “key” must be defined for the data records. Usually a field whose values can uniquely identify data records is selected as the key. If a single field cannot fulfil this criterion, then a combination of fields can serve as the key. For example in a file, which keeps student records, a key could be student no.

### 15.4.2 Operations

1) **Insertion**: Records must be inserted at the place dictated by the sequence of the keys. As is obvious, direct insertions into the main data file would lead to frequent rebuilding of the file. This problem could be mitigated by reserving overflow areas in the file for insertions. But this leads to wastage of space and also the overflow areas may also be filled. The common method is to use transaction logging. This works as follows:
   i) collect records for insertion in a transaction file in their order of arrival.
   ii) when population of the transactions file has ceased, sort the transaction file in the order of the key of the primary data file.
   iii) merge the two files on the basis of the key to get a new copy of the primary sequential file.

Such insertions are usually done in a batch mode when the activity/program, which populates the transaction file, have ceased. The structure of the transaction files records will be identical to that of the primary file.

2) **Deletion**: Deletion is the reverse process of insertion. The space occupied by the record should be freed for use. Usually deletion (like-insertion) is not done immediately. The concerned record is written to a transaction file. At the time of merging the corresponding data record will be dropped from the primary data file.

3) **Update**: Update is a combination of insertion and deletions. The record with the new value is inserted and the earlier version deleted. This is also done using transaction files.

4) **Retrieval**: User programs will often retrieve data for viewing prior to making decisions, therefore, it is vital that this data reflects the latest state of the data if the merging activity has not yet taken place.

Retrieval is usually done for a particular value of the key field. Before return in to the user, the data record should be merged with the transaction record (if any) for that key value.
The other two operations “creation” and “deletion” of files are achieved by simple programming language statements.

### 15.4.3 Disadvantages

Following are some of the disadvantages of sequential file organisation:

- Updates are not easily accommodated
- By definition, random access is not possible
- All records must be structurally identical. If a new field has to be added, then every record must be rewritten to provide space for the new field.
- Continuous areas may not be possible because both the primary data file and the transaction file must be looked during merging.

### 15.4.4 Areas of Use

Sequential files are most frequently used in commercial batch oriented data processing where there is the concept of a master file to which details are added periodically. Examples of this are payroll applications.

---

**E12) Describe the record structure to be used for the lending section of a library.**

**E13) Write a program in ‘C’ language to insert the following records into a file ‘PERSONAL’**

- Adam Bede, 47, Engineer
- Silas Marner, 50, Doctor

Use a name field of size 20, age field of size 2 and profession field of size 20.

**E14) Merge the following, sequenced on NO:**

<table>
<thead>
<tr>
<th>Transactions No.</th>
<th>Name</th>
<th>Master No.</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Beta</td>
<td>1</td>
<td>Delta</td>
</tr>
<tr>
<td>4</td>
<td>Alpha</td>
<td>2</td>
<td>Lambda</td>
</tr>
<tr>
<td>7</td>
<td>Gamma</td>
<td>8</td>
<td>Phi</td>
</tr>
</tbody>
</table>

---

### 15.5 DIRECT FILE ORGANISATION

It offers an effective way to organise data when there is a need to access individual records directly.

To access a record directly (or random access) a relationship is used to translate the key value into a physical address. This is called the mapping function $R_R(key\ value)\rightarrow Address$.

Direct files are stored on DASD (Direct Access Storage Device).

A calculation if performed on the key value to get an address. This address calculation technique is often termed as hashing. The calculation applied is called a hash function.

Here we discuss a very commonly used hash function called Division - Remainder

**Division-Remainder Hashing**

According to this method, key value is divided by an appropriate number, generally a prime number, and the division of remainder is used as the address for the record.
The choice of appropriate divisor may not be so simple. If it is known that the file is to contain \( n \) records, then we must, assuming that only one record can be stored at a given address, have divisor \( n \).

Also we may have a very large key space as compared to the address space. Key space refers to all the possible key values. The address space possibly may not match the actual number of key values in the file, therefore a one to one mapping may not be there. That is calculated address may not be unique. It is called Collision, i.e.

\[ R(K1) = R(K2) \text{ but } K1 \neq K2 \]

Two unequal keys have been calculated to have the same address. The keys are called synonyms.

There are various approaches to handle the problem of collisions. One of these is to hash to buckets. A bucket is a space that can accommodate multiple records. A discussion on buckets and other such methods to handle collisions is out of the scope of this Unit.

### 15.6 INDEXED SEQUENTIAL FILE ORGANISATION

When there is need to access records sequentially by some key value and also to access records directly by the same key value, the collection of records may be organised in an effective manner called Indexes Sequential Organisation.

You must be familiar with search process for a word in a language dictionary. The data in the dictionary is stored in sequential manner. However an index is provided in terms of thumb tabs. To search for a word we do not search sequentially. We access the index that is the appropriate thumb tab, locate an approximate location for the word and then proceed to find the word sequentially.

To implement the concept of indexed sequential file organisations, we consider an approach in which the index part and data part reside on a separate file. The index file has a tree structure and data file has a sequential structure. Since the data file is sequenced, it is not necessary for the index to have an entry for each record following figure shows a sequential file with a two-level index.

Level 1 of the index holds an entry for each three-record section of the main file. The level 2 indexes level 1 in the same way.

When the new records are inserted in the data file, the sequence of records need to be preserved and also the index is accordingly updated.

Two approaches are used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the static index contents may change but the structure does not change. In case of dynamic indexing approach, insertions and deletions in the main data file may lead to changes in the index structure.

Both dynamic and static indexing techniques are useful depending on the type of application.

### 15.7 SUMMARY

This Unit dealt with the methods of physically storing data in the files. The terms - fields, records and files were defined. The organisation types were introduced.

The various file organisation were discussed. Sequential File Organisation finds in use in application areas where batch processing is more common. Sequential Files are simple to use
and can be stored on inexpensive media. They are suitable for applications that require direct access to only particular records of the collection. They do not provide adequate support for interactive applications.

In Direct file organisation there exists a predictable relationship between the key used and by program to identify a particular record and or programmer that record’s location on secondary storage. A direct file must be stored on a direct access device. Direct files are used extensively in application areas where interactive processing is used.

An Indexed Sequential file supports both sequential access by key value and direct access to a particular record given its key value. It is implemented by building an index on top of a sequential data file that resides on a direct access storage device.

15.8 SOLUTIONS/ANSWERS

1) The following record structure could take care of the general requirements of a lending library. Member No., Member Name, Book Classification, i.e. Book Name, Author, Issue Date, Due Date.

2) No model answer is given.

3) (1) Sort Transaction file
   No. | Name
   4   | Alpha
   6   | Beta
   7   | Gamma

(2) Merge to get
   No. | Name
   1   | Delta
   2   | Lambda
   4   | Alpha
   6   | Beta
   7   | Gamma
   8   | Phi

If a sequential file on a disc is to occupy the least possible space its records must be stored continuously i.e. with no unused space between them.

In case of addition or deletion of a record, the file must be rewritten to maintain its sequential order without spaces.