0 and hence this term need not be included to `r`. The reader is advised to take up some more examples with variable number of terms and validate the working of `PolyAdd()` function.

6.19.2 Adding two long positive integers using a circular list

A long integer is one that contains a very large number of digits. The programming languages like C, C++, Java, etc offer `int` and `long` data types that can be declared. However, these data types have their limitations in terms of the number of digits. This is because, as you have already studied from Chapter1 an *integer* variable occupies 2 bytes and a *long* takes 4 bytes of memory space. Now the fact is that we cannot handle very large numbers that is beyond the range of `int` or `long`.

This leads to the notion in using a linked list that can store a very large number of digits. Generally we do not have any restriction in the number of nodes of a linked list.

Each node can be made to hold few digits of a long number. Essentially, we break the large number into blocks of 4 or 5 digits each and store each block in a node. All of these nodes are linked together to form a chain of digits representing the original number. For example the number,

8 9 9 9 3 1 6 3 2 9 8 9 8 7 0 5 3 0

can be rearranged 5 digits per block and the resultant structure is as follows,

8 9 9	9 3 1 6 3	2 9 8 9 8	7 0 5 3 0
Block - 4	Block - 3	Block – 2	Block - 1

Each block represents a linked list node and we wish to use a circular list with a header node to store a long integer. The objective of this application is:

Given two long positive integers, represented as a circular list, are to be added and the result to be stored is a circular list again.

The Method

To add two long integers, the method is similar to what we do with a pencil and paper. Let us see how this is done,

```
 9 7 8 6 2 3 7 6 8 8 1 0 8 7
         7 6 9 2 4 3 7 8 9 1
 ---------------------------
 9 7 8 7 0 0 6 9 3 1 8 9 7 8
 ---------------------------
```

We start adding the least significant digits of the two numbers. If the sum (partial sum) results in a carry, then it is brought over to the next pair of digits, and so on. Because of an addition operation we get a sum and a carry (a sum need not always generate a carry). We shall adopt the same method, for the numbers stored in a circular list. The only difference is that in the pencil and paper method we add two digits at a time and in the case of a circular list, we add two blocks (two nodes).

Data 1	9786	23768	81087
Data2		76924	37891

But the question is how to separate the sum and carry? For example, consider the least significant block,

$$\begin{array}{r} 81087 \\ 37891 \\ \hline 118978 \\ \hline \end{array} \qquad \begin{array}{l} \text{Sum} = 118978 \ \% \ 100000 = 18978 \\ \text{Carry} = 118978 \ / \ 100000 = 1 \end{array}$$

When we add two 5 digit numbers, it produces a 6 digit number, if there is a carry. To get the sum find the *mod* of the partial sum with 10000 and to get the carry, divide the number by 100000. Even when you do not get a carry, this method works. For instance,

$$\begin{array}{r} 17200 \\ 21321 \\ \hline 38521 \\ \hline \end{array} \qquad \begin{array}{l} \text{Sum} = 38521 \ \% \ 100000 = 38521 \\ \text{Carry} = 38521 \ / \ 100000 = 0 \end{array}$$

Let us go back to our original example which has produced the carry. The next block considered is,

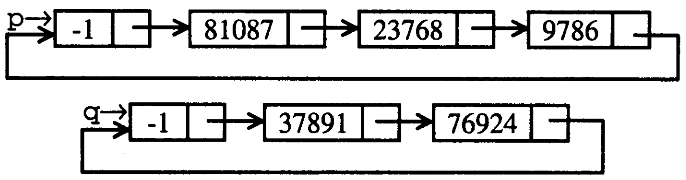
$$\begin{array}{r} 23768 \\ 76924 \\ \hline 100692 \\ \hline \end{array} \qquad \begin{array}{l} \text{Sum} = 00692 + 1 = 00693 \\ \text{Carry} = 1 \end{array}$$

The previous carry is added to the current sum. Considering the 3rd block of number 1 with the current carry, we have

$$\begin{array}{r} 9786 \\ 1 \\ \hline 9787 \\ \hline \end{array}$$

Hence, final sum is $= \boxed{9787} \ \boxed{00693} \ \boxed{18978}$

The least significant block in each data is stored as the starting node and the next block as the second node and so on. Finally the most significant block will be the last node. This way of storing the digits would help us to add the two numbers starting from the first node (see Figure 6.40).



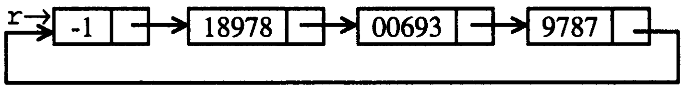


Fig. 6.40 Circular list representation of two data elements and its sum

Even the final list r is stored just similar to p and q and perhaps the resultant number may be used for further calculations. The subsequent sum requires this sum to be in the same format as that of any other number. For example,

$s = d_1 + d_2;$
$t = s + d_3;$

To add s and d_3, both should have their least significant digit blocks to be stored as shown in Figure 6.40.

Implementation

Before we show the program for this application, let us summarize the steps required to accomplish our task. The pseudo code is shown in Figure 6.41.

```
Step 1: p = link(p), q = link(q);   // point to first node
Step 2: while (info(p) != -1 and info(q) != -1) do
        {
              find sum and carry;        // use mod and integer division
              r = Insert(sum) ;          // r is the resultant list
              p = link(p);
              q = link(q);               // advance both the pointer.
        }
Step 3: while (info(p) != -1) // copy remaining nodes from p to r
        {
              find the sum by adding the previous carry (if any).
              Also get the new carry.
              r = Insert(sum);
        }
Step 4: while (info(q) != -1) // copy remaining elements from q to r
        {
              find the new sum and carry.
              r = Insert(sum);
        }
Step 5: if (any final carry)
              r = Insert(carry),
Step 6: return r;
```

Figure 6.41 Pseudo code for Long Add(p, q)

The Program 6.30 shows the `LongAdd(p,q)` function that accepts two circular lists containing the two numbers. Then it returns the sum via a local variable `r` of type `NODE`. Let us assume that each node can have a maximum of 5 digits.

Program 6.30
Adding two long positive integers

```
NODE LongAdd (NODE p, NODE q)
{  /* Add two long integers pointed by p and q */
   /* The result is returned through r        */
    NODE r = NULL;
    long int d = 100000L;
    long int num, total, carry;
    p = p->link;
    q = q->link;
    carry = 0;
    while (p->info != -1 && q->info != -1)
    {
         total = p->info + q->info + carry;
         num = total % d;
         p = p->link;
         q = q->link;
         carry = total / d;
         r = Insert(r, num);
    }
    while (p->info != -1)
    {
         total = p->info + carry;
         num = total % d;
         p = p->link;
         carry = total / d;
         r = Insert(r, num);
    }
    while (q->info != -1)
    {
         total = q->info + carry;
         num = total % d;
         q = q->link;
         carry = total / d;
         r = Insert(r, num);
    }
    if (carry == 1) r = Insert (r, carry);
    return r;
}
```

The first `while` loop scans through both the lists by adding the `info` fields of the corresponding nodes.

If a carry is generated it is brought to the next node and so on. The variables sum and carry are obtained using % and / as explained earlier. Since, the numbers need not be of same length, the remaining nodes from `p` or `q` are copied to `r`. In this process, we must take care of any possible carry generated. Try with some more numbers that could generate a carry through out. The complete program for this problem is given in Chapter 10.

6.20 LINKED LIST WITH ARRAYS (Simulating Pointers)

For most of the applications, we can implement the desired linked lists using dynamic allocation and C pointers. There is another convenient and efficient method that uses an array of nodes and simulating pointers by integers that indexes into this array. Suppose we use an array node, each element of which has two fields called info and link. The `node[0]`, for example, will have info and link in which the link field will have the address of `node[1]`. Similarly, the `node[1].link` will contain the address of `node[2]`, and so on. In general, the `node[i].link` will have the address of `node[i+1]` (see Figure 6.41(b)).

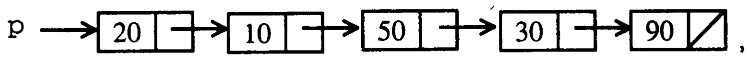


Fig. 6.41(a) Linked List

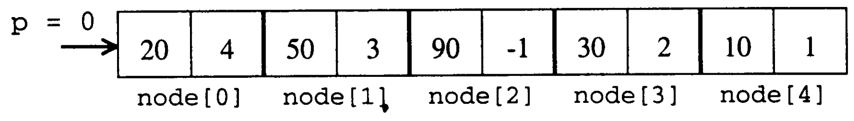


Fig. 6.41(b) Linked List using simulated pointers

For example, the first node of the linked list, 20 is at 0th location and the next element 10 is at location 4 specified in its `link` field. The last node 90 has a –1 in its `link` field which is same as NULL. The structure definition can now be written as,

```
struct SimList
{
      int info;
      int link;
};
struct SimList node[MAX];
```

With this definition we can have a maximum number of nodes fixed by `MAX`.

Initializing the List

The next work is to initialize the array such that it forms a linear linked list. This is done by storing the address of `node[1]` in `node[0].link`, the address of `node[2]` in `node[1].link`, etc. The link field of the last node will contain –1. The C code to do this is shown below:

```
for (i = 0; i < n-1; i++)
     list[i].link = i + 1;
list[n-1].link = -1;
```

Allocating a node

Once the pool of memory is available, we can request for a node to be allocated from this available list and use it for building the linked list. This is what we do in a pointer implemented linked list through *malloc()* function. The difference, however, is that the memory is obtained dynamically from a special memory area called heap. The heap structure is also organized as a linked list and *malloc()* allocates dynamic memory and free() deallocates the memory, this means that the dynamic memory will be put back into the availability list.

The Program 6.31 shows the *getnode()* function. We shall assume that avail is the pointer (integer variable) that points to the availability list and is initialized to 0.

Program 6.31
Getting a node from availability list

```
int getnode()
{
     int y;
     if (avail == -1)
     {
          printf("Overflow\n");
          exit(1);
     }
     y = avail;
     avail = list[avail].link;
     return y;
}
```

For example, a node is allocated only when avail is not –1. Because, when avail is –1, it means that all the nodes have been allocated or the availability list is Full. Whenever, `getnode()` function is invoked, the current value of avail is the location which will be allocated (or returned) and once this node is allocated, avail is advanced to the next node. Therefore, avail always points to the first available node in the availability list.

The various list operations like insertion, deletion, etc., can be performed using a similar kind of concept as it was used in pointer implementation. This is left as an exercise to the reader.

6.21 SUMMARY

- In the linked representation, each data object is represented as a **node**. Each node consists of two components - **value or info field** and a **link** field.
- Since each node has exactly one link, this structure is called as a **singly linked list**. The link field of the last node has a **NULL** or **0**.
- Any list structure should be equipped with the following operations:
  - **Building** a linked list
  - **Inserting** a new element at a specified place. This specified place may be- the index, or info or direct address, etc.
  - **deleting** a specific node. The specific node may be- its info, index or address, etc.
  - **destroying** the entire list.
  - **finding** a specific node based upon the info.
  - **searching** for a given node.
  - **displaying** the list.
  - **counting** the number of nodes.
- The stack and queue (*apq* and *dpq* also) can be implemented using linked lists. The advantage is that these data structures can grow dynamically.
- One of the advantages is that the ***deque*** can very easily be implemented using a linked list rather than using an array.
- There several examples discussed in this chapter that uses linked list. For instance, building an ordered list, merging two ordered lists, reversing a list, etc.
- A **circular list** is one wherein the last node points to the first node.
- A **doubly linked list** is one that has two link fields for each node for easier traversing on either side.
- Without using dynamic allocation and pointers, a list can be realized using an array.

6.22 EXERCISES

6.1 What is a linked list? How is it different from an array?

6.2 Write C functions for the following:
(a) Build a *n* node linked list, where *n* is obtained from the user.
(b) Display the *info* field of alternate nodes only.
(c) Assuming that the *info* fields are integers, add all the elemental values of the nodes and display the same.

6.3 Develop a singly linked linear list to store the following Employee data:
[*empid, name, desig, salary*]
The employee records are to be added in the front only. Write C functions for the following:
(a) To build the Employee list.
(a) To update the salary of all employees by 10%.
(b) To delete an employee bearing *id* number 777.

(c) To print the total salary of all Project Leaders (*PL*) and Software Engineers (*SE*) separately.
(d) To search for an Employee based upon his/her *id* number.

6.4 Write a C function `Search()` to search for a given key in a singly linked list. If the key is **found** then return *true*, else add the key to end of the list and return *false*.

6.5 Consider a singly linked list with a header node. The header node contains the length of the list. This information will be updated whenever an insertion or deletion occurs in the list. Write C functions to build such a list and return the length of the list simply by accessing the header node.

6.6 Give at least four differences between a circular list and an ordinary linked list.

6.7 Develop C functions to do the following (no header node allowed):
(a) To implement a stack and queue.
(b) To find the length of the list.
(c) To reverse the list.
(d) To delete all nodes that matches with the key.
(e) To concatenate two circular lists.
(d) To split a circular list into two lists, given the index value.
(e) To destroy the entire list.

6.8 What do you understand by a Doubly Linked List? Explain with a neat diagram.

6.9 Write C functions to insert before a key node, after a key node, delete a node with its *info* and build an ordered list.

6.10 Assume that we wish to store a set of names in the *info* field of a doubly linked list. Since, each name may be of different lengths, we can use a *char* pointer for the *info* field as shown below:

```
struct Dlist {
        char *p;
        struct Dlist *left;
        struct Dlist *right;
};
```

Allocate only required memory for p using *realloc*(). How will you manage if the name field is to be updated after the list is built?

6.11 Design an algorithm/program to implement a singly linked list using an array. Write functions to insert an integer delete an element from the list and build the linked list.

6.12 Clearly distinguish between static and dynamic implementation techniques of linked lists.

6.13 Write a C program to add two long positive floating point numbers using linked lists.

6.14 Develop an algorithm to add two long signed integers using linked lists.

Chapte 7

Trees

7.1 INTRODUCTION

To represent certain complex hierarchical relationships linked lists or other data structures are not useful. **Trees** are ideal data structures for representing hierarchical structure, fast searching, priority queues, etc., There are wide varieties of trees and all can not be discussed in this book.

In the present chapter we study two basic varieties: **general trees** or (simply trees) and **binary trees**. The discussion starts with the basic definitions, properties of trees and binary trees, tree operations, etc.,

In addition the chapter covers the following topics:

- Linked and implicit representation of binary trees.
- Tree traversal methods- in-order, preorder and post-order.
- Finding the depth, height, number of leaf nodes, number of internal nodes, copying a tree, etc.
- Deleting a node in a binary tree.
- Priority queues with heap structure.
- Threaded binary trees.

7.2 GENERAL TREES

So far we have discussed data structures that were basically called as linear data structures. Trees are **nonlinear data structures** mainly used to represent hierarchical data.

Example 7.1

Take for example a simple family tree arranged in a hierarchical manner. The tree starts with the head of the family Kamath at the top (see Figure 7.1). He has three children Balu, Anusha and Krishna listed in the next level.

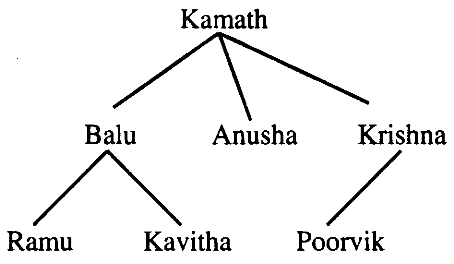


Fig. 7.1 A family tree

Balu in turn has two children Ramu and Kavitha, Krishna has just one child Poorvik and Anusha has no children. The relationships are represented using the edges or realtions among various family members. For example, Kavitha's father is Balu and her grand father is Kamath. Also, from this hierarchical representation, it is easy to identify Kamath's descendants, Ramu's ancestors, and so on.

Example 7.2

Let us consider another example, an organization, where the hierarchical structure is followed. In organizations the highest level is Chief Executive Officer (CEO) and under him/her there may be several Vice Presidents (VP) and then each VP may have a set of Project Manager (PM) and the next level is Group Leaders (GL), and so on (see Figure 7.2).

The third level indicates Project Managers and the fourth level-the Group Leaders. There are three Project Managers PM1, PM2 and PM3 for VP-Production, two PMs for VP - Marketing and only one PM for VP – customer - support. Notice that CEO is in the top most level and hence he would have all privileges.

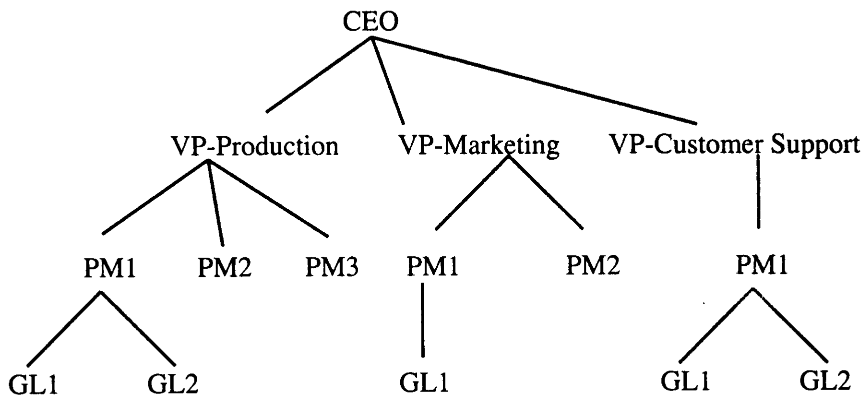


Fig. 7.2 A Corporate Hierarchical Structure

Example 7.3

A much simpler example is a text editor like MS-Word, Notepad, DOS Editor, Word Star, etc. Every text editor comes with a set of menus. These menus are arranged in a hierarchical fashion as shown in Figure 7.3 for MS-Word type of editors. The main menu consists of File, Edit, View, Insert, etc. File menu contains seven submenus New, Open, Save, etc. When you click on save you may not get any more submenus, this means that no further edges a lines in the next level exists.

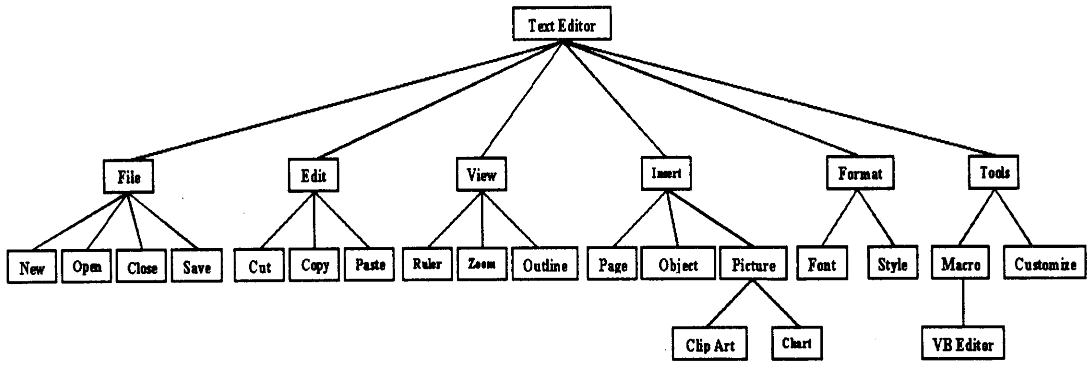


Fig. 7.3 MS-Word Text Editor Menus

However, Insert menu has submenu called Picture and under this we have the types of pictures that could be inserted to your document - Clip Art or Chart.

The real advantage of this kind of hierarchical model is to enable the designers to organize and design software into a modular fashion. In fact, you can see here that a complex problem of designing a text editor is divided into simpler and smaller modules that can be solved with much less effort.

Definition

A **tree T** is a finite, non-zero-empty set of nodes,

$$T = \{r\} \cup T_1, \cup T_2 \cup \ldots \cup T_n;$$

with the following properties:

(1) A designated node of the set, *r*, is called the root of a tree.

(2) The remaining nodes are partitioned into $n \geq 0$ subsets, $T_1, T_2, \ldots T_n$ each of which is a tree.

For convenience, we shall use

$$T = \{ r, T_1, T_2, \ldots T_n\}$$

The definition of tree is recursive; however it is finite. In short, *a tree is a finite non-empty set of elements in which one of the elements is a root and the others are subtrees.* Figure 7.4 shows few example trees.

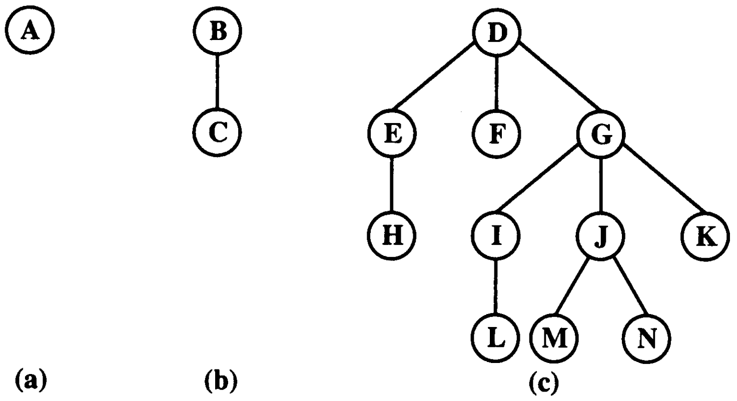


Fig. 7.4 Trees

Compare these trees with that of Figure 7.1, 7.2 and 7.3, which are nothing but the same types of trees. In Figure 7.1 the root node is Kamath, CEO is the root in Figure 7.2 and Text Editor in the Figure 7.3.

Terminology

When drawing a tree, each element is represented as a **node**. The nodes are connected using **edges**. Each subtree is drawn in a similar way as its root at top and subtrees below the root. We will use terms like *children nodes*, *parent nodes*, *grand child*, *grand parent*, *ancestor*, *descendent*, *leaves*, *levels* and *degree* while discussing various topics in trees. These terms will be taken in *binary trees* section elaborately.

7.3 BINARY TREES

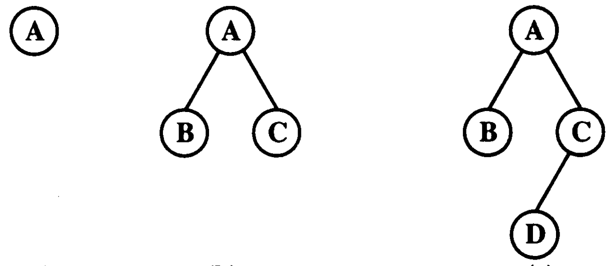
A **binary tree T** is a finite set of nodes one of with the following properties:

(1) Either the set is empty, T = 0, or

(2) The set consists of a root *r*, and exactly two distinct binary trees called ***left subtree*** (T_L) and a **right subtree** (T_R).

That is, $T = \{r, T_L, T_R\}$

The trees shown in Figure 7.5 are all binary trees and the definition of binary tree is also recursive.



(a) (b) (c)

Fig. 7.5 Binary trees

The essential difference between a binary tree and a tree are:

- A binary tree can be empty, whereas a tree cannot.
- Each element in a binary tree has exactly two subtrees (one of these two may be empty). An element in a tree can have any number of subtrees.
- The subtrees of each element in a binary tree are ordered. In other words, we can easily distinguish the left and right subtrees. The subtrees in a tree are unordered.

Terminology - revisited

Children and parent

The edges in a tree connect an element node and its **children** nodes. In Figure 7.4(c), for example, E, F and G are the children of D and D is their **parent** node.

Sublings

The children of the same parent are called as **sublings**. In Figure 7.4(c), E, F and G are sublings, because their parent is D. Similarly, I, J and K are sublings of G. Notice that H and I are not sublings, as their parents are different.

Left and Right descendent

Every node that is reachable from a node, say V, is called a **descendent** of V. For example, in Figure 7.4(c), L is a descendent of G where as H is not. If a descendent node appears on the left subtree, then it is called a **left-descendent**, otherwise it is called as **right-descendent**.

Leaves

In a binary tree or a tree, a node with no children is called as a **leaf node**. In Figure 7.4(b) C is the only leaf node and in Figure 7.4(c) H, F, L, M, N and K are all leaves. A root generally does not have a parent node.

Level

By definition, the root is at the highest level and its children nodes are at one level more than the root. In general, the **level** of any node is one more than its father. In Figure 7.4(c), the root D is at level 0, its children E, F and G are at level 1, H, I, J, K are at level 2, and so on.

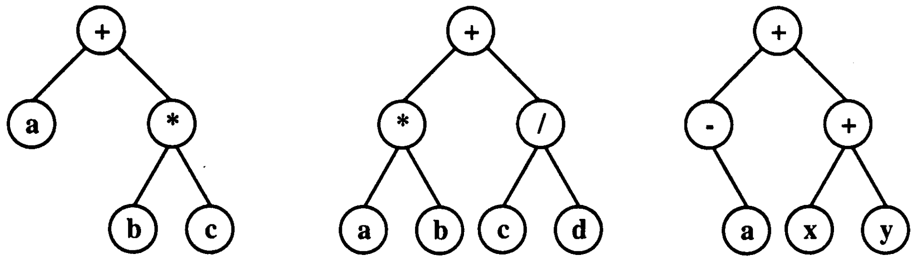
Depth (or Height)

The **depth** of a binary tree is the longest path from the root to any leaf node. Hence, the depth of Figure 7.4(a) is 0, Figure 7.4(b) is 1 and Figure 7.4(c) is 3.

Degree

The **degree** of a node is the number of children it has. The degree of a leaf node is zero and the degree of node G in Figure 7.4(c) is 3.

Arithmetic expressions can be represented using binary trees. Figure 7.6 shows binary trees that represent arithmetic expressions. Each operator +, -, *, / must have one or two operands. The precedence of the operators is implicit in the way the tree is built.



(1) a + b * c **(2) (a * b) + (c / d)** **(3) (-a) + (x + y)**

Fig. 7.6 Expression trees

For instance, in Figure 7.6(1) the sub expression b * c is evaluated first which becomes the right operand (right subtree) for the operator +. Chapter 10 gives the C code for building expression trees and evaluating them.

7.4 PROPERTIES OF BINARY TREES

Property 1

A binary tree with *n* nodes, $n > 0$, has exactly $n - 1$ edges.

Property 2

A binary tree of height h, $h \geq 0$, has at least h and at most $2^{h+1} - 1$ nodes. Recall that the root node is at level 0 and children of root are at level 1 and so on.

Property 3

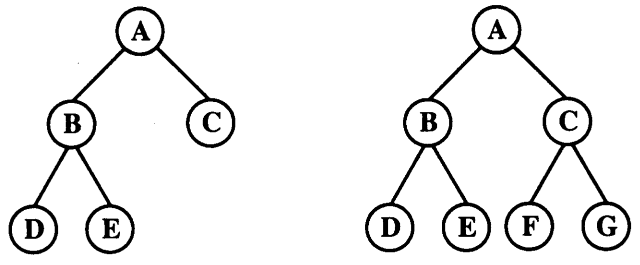
The height of a binary tree with n nodes, $n \geq 0$, has at most n and at least $[log_2(n + 1)]$
That is, $h \geq [log_2(n + 1)]$

Property 4

The maximum number of leaves in a binary tree with n nodes, $n \geq 0$, is $= l = 2^h$.
Or, $h = log_2 l$

Property 5 [Strict Binary Tree]

If every non-zero leaf node (or **internal node**) has no empty left and right subtree, then such a binary tree is called as a **strict binary tree** (see Figure 7.7).



(a) Strict binary tree. **(b) Complete binary tree.**

Fig. 7.7 Special binary trees

Property 6 [Complete binary tree]

A binary tree of height h that contains exactly $2^{h+1} - 1$ elements is called a **complete binary tree.** The binary tree shown in Figure 7.7(b) is a complete binary tree and the tree shown in Figure 7.7(a) is not.

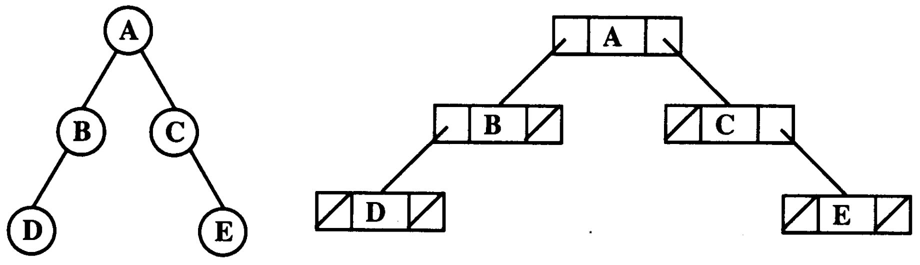
At each level you must lave 2^l nodes to make the tree as complete binary tree. In Figure 7.7(b) at level 0, you have node A and at level 1there are 2 nodes B and C (2^1) and at level 2 there are four nodes, D, E, F, and G. (2^2) and so on. Total number of nodes is $2^{2+1} - 1 = 7$.

7.5 REPRESENTATION OF BINARY TREES

There are two types of representing binary trees, namely

- Implicit representation (using arrays).
- Linked representation (using pointers).

The most popular way to represent a binary tree is using the second method. The first method will be discussed in Section 7.9. Each element or node is represented by exactly two link fields to have a left subtree link and a right subtree link. Similar to a linked list, every node will have an `info` field. This node structure appears in Figure 7.8. The figure has only two leaf nodes.



(a) **Binary tree.** (b) **Linked representation of tree shown in (a).**

Fig. 7.8 Binary tree and its linked representation

Nodes D and E with the left and right links marked as NULLs. The node B does not have any right subtree and so it is marked with NULL and so also node C, except that the left subtree missing.

The structure definition for a binary tree in C language is as follows:

```
struct Tree
{
        int info;
        struct Tree *left;
        struct Tree *right;
};
typedef struct Tree * TNODE;
```

This definition may look similar to a doubly linked list, but remember that a list is a linear data structure where as a tree is a non-linear data structure.

7.6 TREE TRAVERSAL METHODS

Traversing a binary tree helps in displaying the elemental values, to search for a given key, inserting an element in the tree, deleting an element, copying a tree, etc. Visiting each node in the tree in a systematic way is known as **traversing**.

In a singly linked list this task becomes so simple as the links are linearly arranged. Visiting the next node i.e., (i + 1)th node from ith node is by advancing the link. However, in a binary there are two links - the left subtree and the right subtree. Hence, there could be three ways by which traversing can be done as,

1) Preorder
2) Inorder
3) Postorder

We shall discuss the algorithms pertaining to these three traversal methods in the following section and present the C code in Section 7.8.3.

7.6.1 Algorithm and Examples

We shall assume that in all the three traversal methods, the left subtree of a node is traversed before the right subtree. Perhaps the definition of the binary tree helps in designing the traversal algorithms. This means that the recursive definition of tree is used for the algorithms as shown in Figure 7.9, assuming a non-zero empty binary tree pointed by `t`.

```
Algorithm Preorder (t)
{
    if (t) do thru Step 4                              // when tree is not empty
        Step 1: Visit the Root node.
        Step 2: Traverse the left subtree recursively.     // Preorder(left(t))
        Step 3: Traverse the Right subtree recursively.    // Preorder (right(t))
        Step 4: Return.
}
Algorithm Inorder (t)
{
    if (t) do thru Step 4
        Step 1: Traverse the Left subtree recursively.     // Inorder(left(t))
        Step 2: Visit the root node.
        Step 3: Traverse the Right subtree recursively.    // (Inorder(right(t))
        Step 4: Return.
}
Algorithm Postorder (t)
{
    if (t) do thru Step4
        Step 1: Traverse the left subtree recursively.     // Postorder(left(t))
        Step 2: Traverse the Right subtree recursively.    // Postorder(right(t))
        Step 3: Visit the root node.
        Step 4: Return.
}
```

Fig. 7.9 Algorithms for Pre, In and Postorder

The difference in the three traversals is the order by which the root node is visited. For example, in the case of a preorder traversal, each node is visited before its left and right subtrees are traversed. We shall illustrate these traversals with two examples.

Example 1

The example tree that is used for all three traversals is shown in Figure 7.10. Note that the dotted lines indicate empty left subtrees for nodes 2 and 4.

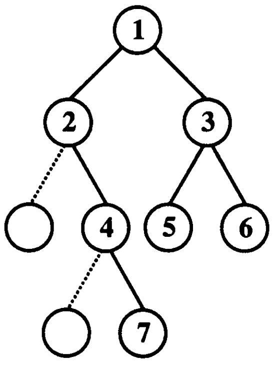


Fig. 7.10 Sample binary tree

(1) Preorder

$\rightarrow 1\ \boxed{\text{LST}}\ \boxed{\text{RST}}$

$\rightarrow 1\ \boxed{\text{2 LST RST}}\ \boxed{\text{3 LST RST}}$

$\rightarrow 1\ 2\ \boxed{\text{4 LST RST}}\ 3\ 5\ 6$

$\rightarrow 1\ 2\ 4\ 7\ 3\ 5\ 6$

(2) Inorder

$\rightarrow \boxed{\text{LST}}\ 1\ \boxed{\text{RST}}$

$\rightarrow \boxed{\text{LST 2 RST}}\ 1\ \boxed{\text{LST 3 RST}}$

$\rightarrow 2\ \boxed{\text{LST 4 RST}}\ 1\ 5\ 3\ 6$

$\rightarrow 2\ 4\ 7\ 1\ 5\ 3\ 6$

(3) Postorder

$\rightarrow \boxed{\text{LST}}\ \boxed{\text{RST}}\ 1$

$\rightarrow \boxed{\text{LST RST 2}}\ \boxed{\text{LST RST 3}}\ 1$