```
void Postorder(TNODE p)
{
     if (p)
     {
          Postorder(p->left);
          Postorder(p->right);
          printf("%d ", p->info);
     }
}

/* to allocate dynamic memmory */
TNODE talloc(void)
{
     return(TNODE) malloc(sizeof(struct Tree));
}
```

Sample Run

```
1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 1
Enter the Element: 56

1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 1
Enter the Element: 100

1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 1
Enter the Element: 72

1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 2
56 100 72
1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 3
56 72 100
1 Create  2 Preorder 3 Inorder 4 Postorder
5 Exit
Enter Choice: 4
72 100 56
```

10. 13

Write recursive C programs for

(a) Searching an element in a given list of integers using binary search method.

(b) Solving the Towers of Hanoi problem.

```
        int n = 0;
        int i = 0;
        FILE *fp;
        fp = fopen("str.dat", "r");
        if (!fp)
        {
              printf("Error in opening file\n");
              exit(1);
        }
        while (!feof(fp))
        {
              fscanf(fp, "%s", &t);
              strcpy(a[i],t);
              i++, n++;
        }
        InsertionSort(a, n);
        printf("The sorted Strings:\n");
        for (i = 0; i < n; i++)

              printf("%s\n",a[i]);
        printf("\n");
        fclose(fp);

}

void  InsertionSort(char a[][MAX], int n)
{
        int j,p,x;
        char k[MAX];

        for (j = 1; j < n; j++)
        {
              strcpy(k,a[j]);
              for (p = j-1; (p >= 0  &&  strcmp(k, a[p]) < 0); p--)
                    strcpy(a[p+1], a[p]);
              strcpy(a[p+1], k);
        }
}
```

Sample Run

Contents of "str.dat"

```
zoo
balloon
ant
eminent
```

```
The sorted Strings:
ant
balloon
eminent
zoo
```

Additional Problems

INTRODUCTION

This is a special section for those who are ready to take up the challenging problems and grasp the solution. The author has made an attempt to present few mind breaking exercises and also give its solutions. Readers are advised to read this section after understanding thoroughly all the chapters of this text.

[illegible]

AP 1.1 What is the output of the following program?

```
#include <stdio.h>
int x, r;
void main()
{
     x = 5;
     r = f(&x) * f(&x);
     printf("%d\n", r);
}
```

Show the values of j after each iteration.

Solution: You may notice that the key point is the static declaration for variable `i` in function `inc()`. For each call count retain its old value. Hence, we can show the tracing of the `for` loop as below:

0	0
1	1
2	3
3	6
4	10

So the final value of `j` is 10.

AP 1.6 Assume two arrays p[1:n] and q[1:n] that are uninitialized and a variable count initialized to 0. Consider the following functions set() and iset():

```
int set (int i)
{
     count++;
     q[count] = i;
     p[i] = count;
}
int iset (int i)
{
     if (p[i] <= 0 || p[i] > count)    /* 1 */
          return 0;
     if (q[p[i]] != i)                 /* 2 */
          return 0;
     return 1;
}
```

(i) Assume that we call the function set() in the following sequence:

```
set(7); set(3); set(9);
```

After these calls, what is the value of count, what is contents of q[1], q[2], q[3] and p[7], p[3], p[9]?

Solution: The values of count and `i` on entry to `set()` are 0, 7 respectively. Hence, `q[1]` = 7 and `p[7]` = 1. At the end, we have `p` and `q` holding values as shown below:

```
q[1] = 7, q[2] = 3, q[3] = 9
p[7] = 1, p[3] = 2, p[9] = 3
```

You can notice here that the contents of `q` are the index values of `p` and the contents of `p` are the index values of `q`.

(ii) Show that if `set(i)` has not been called for some `i`, then regardless of what `p[i]` contains, `iset(i)` will return *false*.

Solution: To solve this problem we must prove that either of `if` conditions give always *true*. This means that what ever be the value of `p[i]` (0 or positive or negative) both or at least one condition evaluates to *true*.

Let us check on these three values of p[i] (for some value of i) and also assume that `count` = 0.

1. When `p[i]` = 0:	`p[i]` <= 0 is *true*	returned value is false.
2. When `p[i]` = negative:	`p[i]` <= 0 is *true*	returned value is false.
3. When `p[i]` = positive:	`p[i]` > count is *true*	returned value is false.

When `p[i]` is positive (but not 0) but less than count, then statement 1 evaluate to false and so statement 2 is executed. But this situation will not occur at all, because with out calling `set(i)` for some `i`, the value of `count` will not change. Its value will remain as 0. Therefore, if we prove that the statement 1 is *true* always, then the returned value must be *false*.

AP 1.7 What is the output of the following C code?

```
void main()
{
     int x, y;
     x = 10; y = 3;
     f(&y, &x, &x);
     printf("%d %d\n", x, y);
}
void f(int *x, int *y, int *z) /* x = 3, y = 10, z = 10 */
{
     *y += 4;
     *z = *x + *y + *z;
}
```

Solution: The function `f()` is invoked by sending the address of `y` and `x`. During the execution of `f()`, we have values of `x` = 3, `y` = 10, and `z` = 10.

```
y = y + 4 = 10 + 4 = 14;
```

This value is written at the memory location pointed by `y` (because it is a pointer variable) and also pointed by `z`. The reason why variable `z` is also afftected is because `y` and `z` point to the address of `x` (see `main()` program where address of `x` is passed to both `y` and `z` in `f()`). Next statement is executed as,

AP 4.1 Consider the following recursive function:

```
int f(int x)
{
     if (x > 100) return x - 10;
     else return f(f(x + 11));
}
```

Assume that the function is invoked as f(95), what is the returned value?

Solution: The steps that are executed by the function `f()` are shown below:

```
(1) f(95)
(2) f(f(106)) // returned value is 96
(3) f(f(107)) // returned value is 97
(4) f(f(108)) // returned value is 98
(5) f(f(109)) // returned value is 99
(6) f(f(110)) // returned value is 100
(7) f(f(111)) // returned value is 101
```

Since 101 is greater than 100, no more recursive calls will be initiated. Hence x – 100 101 – 10 = 91 will be returned to the previous calling program. Since no other work is to be done in this environment this value is carried until the *main* program is reached. Therefore, the final value returned by this function is 91.

AP 4.2 Consider the following C code:

```
int Trial(int a, int b, int c)
{
     if ((a >= b) && (c < b))
          return b;
     else if (a >= b)
               return Trial(a,c,b);
          else return Trial(b,a,c);
}
```

What does this function do?

Solution: Let us take up an example to find out what happens when we execute this function.

```
Trial(5,10,7) -> Trial(10,5,7) -> Trial(10,7,5)
```

The final value returned is 7. This means that the function `Trial()` finds the middle element out of three elements.

AP 4.3 Suppose you are given an array `a[1:n]` *and a function* `Reverse(a,i,j)` *which reverses the order of elements in* `a` *between positions* `i` *and* `j` *(both inclusive). What does the following sequence do:*

```
Reverse(a, 1, k);
Reverse(a, k+1, n);
Reverse(a, 1, n);
```

where, $1 \leq k < n$.

Solution: Let us take an example string and execute the three calling sequences and assume k = 2:

```
Reverse("abcd",1,2) ▶▶ bacd
Reverse("bacd",3,4) ▶▶ badc
Reverse("badc",1,4) ▶▶ cdab
```

Comparing the final value with the original string, we can easily note that the three sequence of operations rotates a left by k positions.

To check our conclusion, let us consider another example with k = 3 and a = "abcd".

```
Reverse("abcd",1,3) ▶▶ cbad
Reverse("cbad",4,4) ▶▶ cbad
Reverse("cbad",1,4) ▶▶ dabc
```

You can see that the string is reversed left by k (=3) positions.

AP 4.4 The following piece of C code computes Fibonacci numbers recursively. Assume that you are given an array f[0:M] with all elements initialized to zero.

```
int Fib(int n)
{
    if (n > M) return -1;              /* Error */
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (_______)                       /* 1 */

        return _______________;        /* 2 */

    t = Fib(n-1) + Fib(n-2);
    ______________________             /* 3 */

    return t;
}
```

Now fill in the blanks with expressions/statements to make `Fib()` *store and reuse computed Fibonacci values.*

Solution: The notion in this problem is to store the computed values of `Fib(n)` for a particular value of n. This stored value in the array `f` will be used by later calls. For

example, if we make a call with n = 3, i.e. `Fib(3)`, by storing the value of `Fib(2)` already in array `f`, it is easier to reduce the number of recursive calls. Hence, we can write

for statement 1: `f[n] != 0`
for statement 2: `f[n]`
for statement 3: `f[n] = t;`

AP 4.5 Write the statements 1, 2, and 3 in the following C function so that it computes the depth of a binary tree.

```
int Depth (Tree t)
{
    int x, y;
    if (!t) return 0;
    x = Depth(t->left);
(1) _____________________

(2) if (x > y) return _________

(3) else return _____________
```

Solution: The solution to this problem is straight forward and it has already been discussed in Chapter 7 (Page 243). Therefore, the statements can be written as,

```
(1) y = Depth(t->right);
(2) ++x;
(3) ++y;
```

AP 5.1 Suppose a stack implementation supports, in addition to PUSH and POP, an additional operation REVERSE, which reverses the order of the elements on the stack.

To implement a queue using the above stack implementation, show how to implement (i) QInsert (inserts an element in the queue) using a single operation and (ii) DQueue (deletes an element from the queue) using a sequence of three operations.

Solution: (i) Since the QInsert operation is to be done using a single operation, we simply call PUSH so that the elements will be inserted with the stack pointer pointing to the recently inserted element.

(ii) As we need to delete the oldest element inserted with just a POP. Because, the stack pointer will be pointing to the most recently inserted element. Hence, we use the following three operations:

(1) REVERSE
(2) POP
(3) REVERSE

The operation (1) brings the oldest element to the top of the stack. Later we extract the element using POP – operation (2). Finally using operation (3), we arrange the elements in the original fashion.

Solved Problems

AP 7.1 Prove that a rooted labeled binary tree can't be uniquely constructed given its postorder and preorder traversals.

Solution: Referring to Section 7.6.2 (page 236), it is evident that a labeled binary tree can be uniquely constructed given its preorder and inorder traversals. With out the inorder traversal, it is not possible to construct the tree uniquely.

AP 7.2 A complete n-ary tree is one in which every node has 0 or n sons. If x is the number of internal nodes of a complete n-ary tree, what is the number of leaves in it?

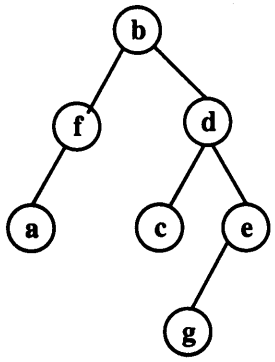
Solution: We know that, a complete binary tree with x internal nodes has $x + 1$ leaves (the proof is based upon the height of the binary tree). The problem now is to be solved for n-ary tree. Hence, we have

No. of leaves = $x * (n - 1) + 1$

AP 7.3 Draw the binary tree with nodes a, b, c, d, e, f and g for which the inorder and postorder traversals are as follows:

Inorder:	***a f b c d g e***
Postorder:	***a f c g e d b***

Solution: From Section 7.6.2, the answer can readily be written as,



AP 7.4 Draw the min-heap that results from insertion of the following elements in order into as initially empty min-heap: 7, 6, 5, 4, 2, 3, 1. Show the result after the deletion of the root of the heap.

Solution:

Fig. 1 Insert 7

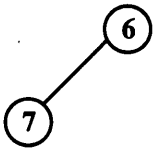


Fig. 2 Insert 6

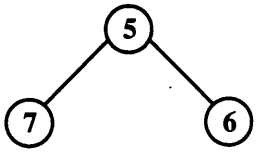


Fig. 3 Insert 5

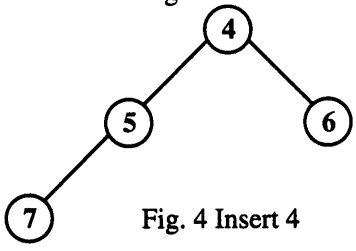


Fig. 4 Insert 4

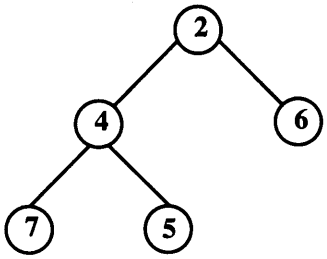


Fig. 5 Insert 2

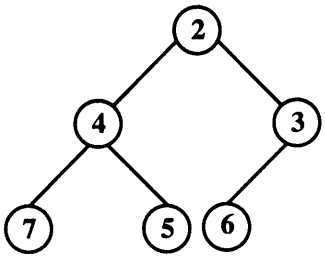


Fig. 6 Insert 3

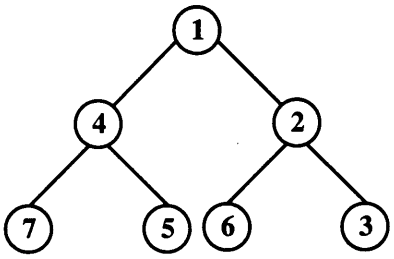


Fig. 7 Insert 1

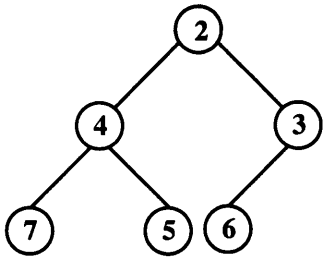


Fig. 8 After deleting root (element 1)

AP 7.5 Consider the following nested representation of binary trees: (X Y Z) indicates Y and Z are the left and right subtrees, respectively of node X. Note that Y and Z may NULL, or further nested. Which of the following represents a valid binary tree?

```
(1) (1 2 (4 5 6 7))
(2) (1(2 3 4) (5 6 7))
(3) (1 ((2 3 4) 5 6) 7)
(4) (1 (2 3 NULL) (4 5))
```

Solution:

(1) In the first case, you find that there are four elements 4, 5,.6 7 that can not make valid binary tree.

(2) In this case, the left and right subtrees for node 1 are (2 3 4) and (5 6 7). Similarly, for each these subtrees, you have left and right subtrees. Hence, the second choice represents a valid binary tree.

(3) & (4) These sequences do not have the required binary structure.

AP 7.6 Let LASTPOST, LASTIN and LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal, respectively, of a complete binary tree. Which of the following is alwcys true?

(1) LASTIN = LASTPOST
(2) LASTIN = LASTPRE
(3) LASTPRE = LASTPOST

Solution: The second combination is always true.

AP 7.7 Insert the following elements into a binary search tree in the order specified below:

15, 32, 20, 9, 3, 25, 12, 1

Solution:

(1) (15 NULL NULL)
(2) (15 NULL 32)
(3) (15 NULL (32 20 NULL))
(4) (15 9 (32 20 NULL))
(5) (15 (9 3 NULL) (32 20 NULL))
(6) (15 (9 3 NULL) (32 (20 NULL 25) NULL))
(7) (15 (9 3 12) (32 (20 NULL 25) NULL))
(8) (15 (9 (3 1 NULL) NULL) (32 (20 NULL 25) NULL))