CONTENTS

Chapters and Topics **page**

CHAPTER – 27

TREE SEARCHING

SEQUENTIAL SEARCH

The simplest form of a search is the **sequential search**, which is easily applicable to a table organized either as an array or as a linked list. Storing a table as a lined list has the advantage that the size of the table can be increased dynamically as needed.

Algorithm for sequential insertion search for a linked list:

```
q = null;
for (p = table; p != null && k(p) != key; p = next(p) )
    q = p;
if (p != null)      /* found key  */
    return (p);
/* insert a new node   */
s = getnode();
k(s) = key;
r(s) = rec;
next(s) = null;
if ( q == null)
    table = s;
else
    next(q) = s;
return (s);
```

efficiency of sequential searching:

Assume that there are no insertions or deletions, so that only a search takes place through a table of constant size n. A successful search will take on the average $(n + 1)/2$ comparisons, and an unsuccessful search will take n comparisons; the search would be of the order of n or $O(n)$.

BINARY SEARCH TREES

Binary Search Tree can be used to store a file in order to make sorting a file more efficient. In which all the left descendants of a node with key ***key*** have keys that are less than ***key***, and all the right descendants have keys that are greater than or equal to ***key***. The inorder traversal of such a binary tree yields the file sorted in ascending key order.

Using **binary tree** notation, we assume that each node contains four fields: ***k***, which holds the record's key value, ***r***, which holds the record itself, and ***left*** and ***right***, which are pointers to the left and right sub trees.

Algorithm searching for the key (key):

```
p = tree;
while ( p != null && key != k(p) )
        p = (key < k(p) ? left (p) : right (p) );
return (p);
```

A **binary search** actually uses a sorted array as an implicit **binary search tree**. The middle element of the array can be thought of as the root of the tree, the lower half of the array can be considered the left sub tree, and the upper half can be considered the right sub tree.

The advantage of using a **binary search tree** over an array is that a tree enables search, insertion, and deletion operations to be performed efficiently. If an array is used, an insertion or deletion requires that approximately half of the elements of the array be moved. On the other hand, the **search tree** requires that only a few pointers be adjusted.

Algorithm to insert into a Binary Search Tree:

Does a **binary search** insert a new record in to the tree if the search is unsuccessful? The function ***maketree*** may differ from algorithm to algorithm based on the structure being used. In this algorithm it is assumed that ***maketree*** constructs a **binary tree** consisting of single node whose information field is passed as an argument and returns a pointer to the tree.

```
q = null;
p = tree;
while (p != null)
{
    if (key == k(p) )
        return (p);
    q = p;
    if (key < k(p) )
        p = left (p);
    else
        p = right (p);
}
v = maketree (rec, key);
if ( q == null)
    tree = v;
else
    if (key < k(q) )
        left (q) = v;
    else
        right (q) = v;
return (v);
```

After inserting a new record, the tree retains the property of being sorted in an inorder traversal.