Chapter 2

Advanced Data Structures and Recurrence Relations

Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.

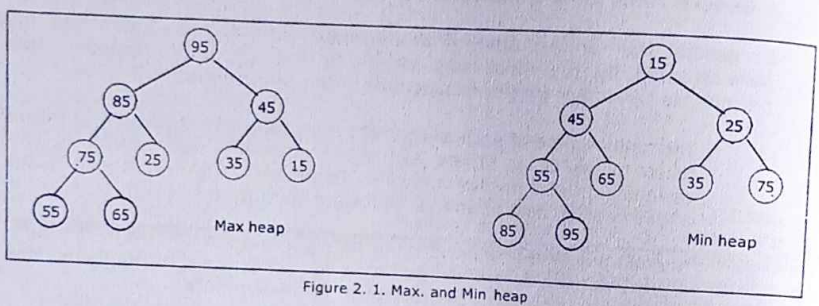


Figure 2. 1. Max. and Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children. Figure 2.1 shows the maximum and minimum heap tree.

Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2*i.
- The right child of an element stored at location *i* can be found in location 2*i + 1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

For example let us consider the following elements arranged in the form of array as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

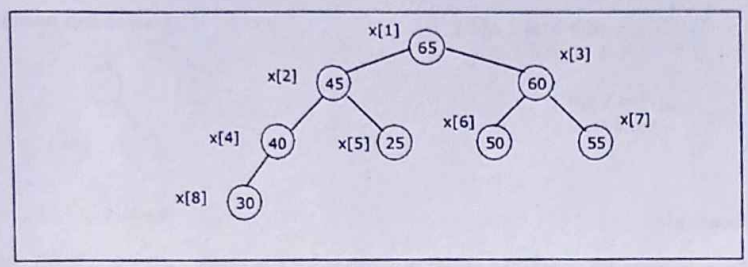


Figure 2. 2. Heap Tree

Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it

requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

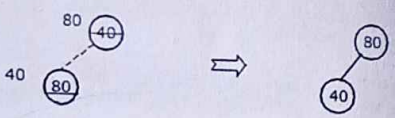
```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        integer i, n;
        i = n;
        item = a[n] ;
        while ( (i > 1) and (a[⌊i/2⌋] < item ) do
        {
                a[i] = a[⌊ i/2 ⌋ ] ;                    // move the parent down
                i =⌊i/2⌋ ;
        }
        a[i] = item ;
        return true ;
}
```

Example:

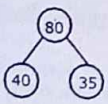
Form a heap by using the above algorithm for the given data 40, 80, 35, 90, 45, 50, 70.

1. Insert 40:

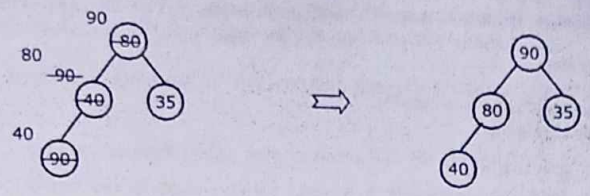
2. Insert 80:



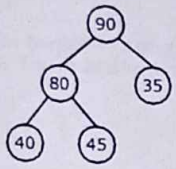
3. Insert 35:



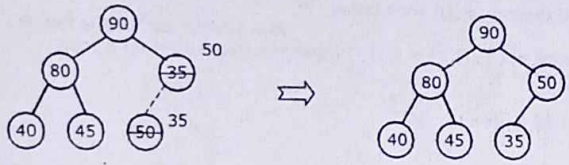
4. Insert 90:



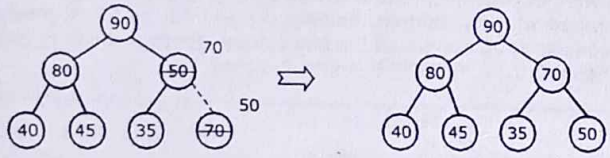
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

```
delmax (a, n, x)
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

```
adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a
single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                        // a position for item is found
                else a[⌊j / 2⌋] = a[j] // move the larger child up a level
                j = 2 * j;
        }
        a [⌊j / 2⌋] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appear as the leave node, hence re-heap is completed. This is shown in figure 2.3.

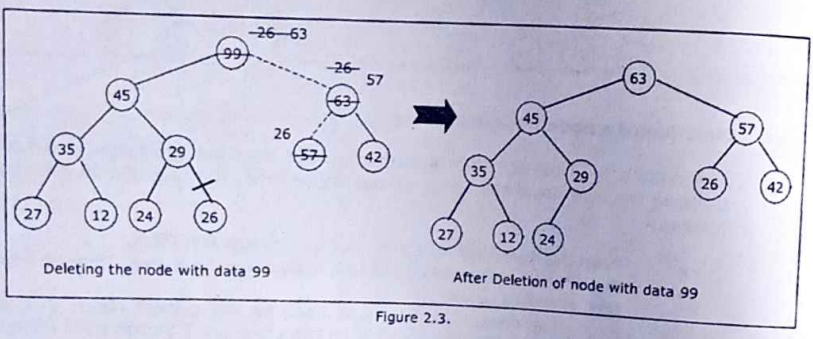


Figure 2.3.

Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap.

Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.

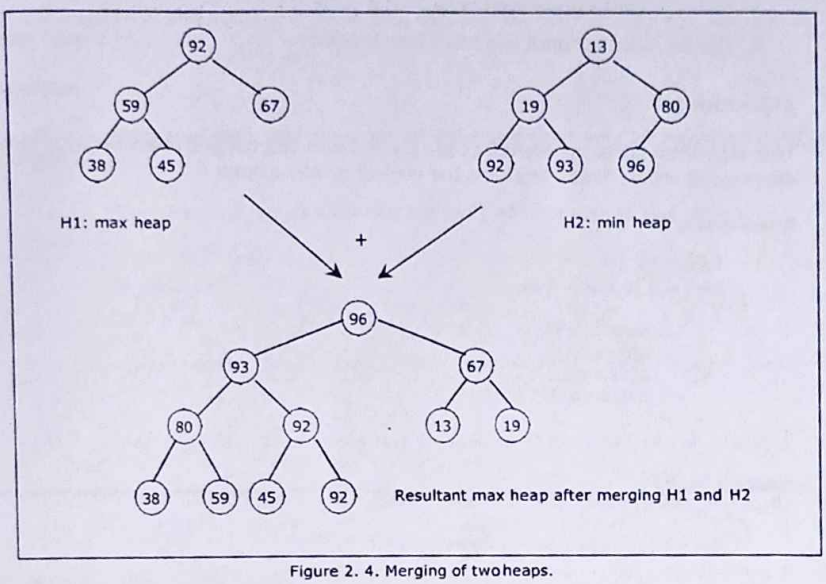


Figure 2. 4. Merging of two heaps.

Applications of heap tree:

They are two main applications of heap trees known:

1. Sorting (Heap sort) and
2. Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

```
heapsort(a, n)
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[I];
                a[i] = a[1];
                a[1] = t;
                adjust (a, 1, i – 1);
        }
}
```

```
heapify (a, n)
//Readjust the elements in a[n] to form a heap.
{
        for i ←⌊ n/2 ⌋to 1 by – 1 do adjust (a, i, n);
}
```

```
adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i+1) are combined
// with a(i) to form a single heap, 1≤i≤n. No node has an address greater
// than n or less than 1.
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[⌊ j / 2 ⌋] = a[j] // move the larger child up a level
                j = 2 * j;
        }
        a [⌊ j / 2 ⌋] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes O(log k), where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time O(log k), where 'k' is the number of elements in the heap at the time. Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

Thus, for *n* elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

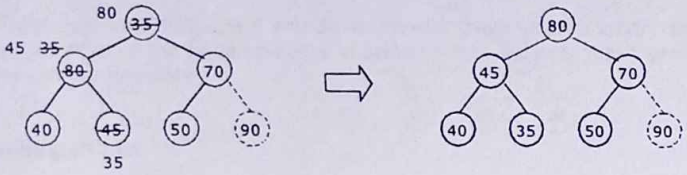
Example 1:

Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

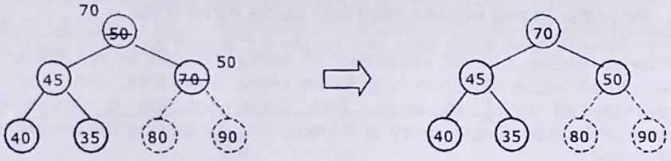
Solution:

First form a heap tree from the given set of data and then sort by repeated deletion operation:

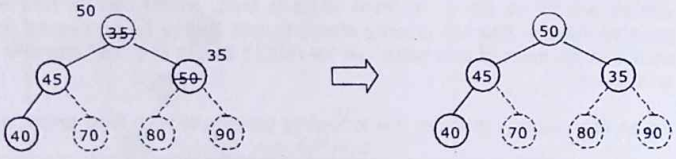
1. Exchange root 90 with the last element 35 of the array and re-heapify



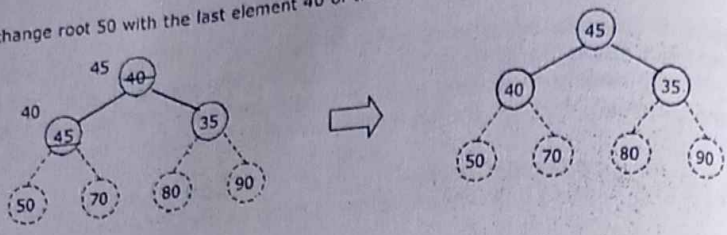
2. Exchange root 80 with the last element 50 of the array and re-heapify



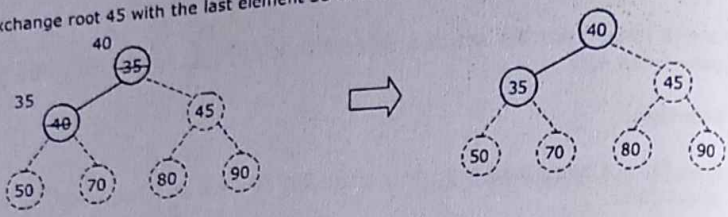
3. Exchange root 70 with the last element 35 of the array and re-heapify



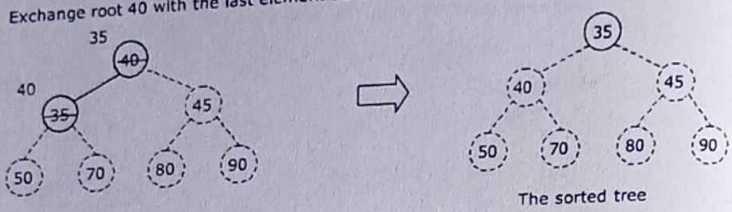
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on.

As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Binary Search Trees:

A binary search tree has binary nodes and the following additional property. Given a node t, each node to the left is "smaller" than t, and each node to the right is "larger". This definition applies recursively down the left and right sub-trees. Figure shows a binary search tree where characters are stored in the nodes.

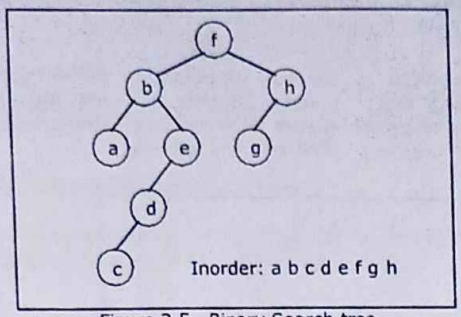


Figure 2.5. Binary Search tree

Figure 2.5 also shows what happens if you do an inorder traversal of a binary search tree: you will get a list of the node contents in sorted order. In fact, that's probably how the name inorder originated.

Binary Tree Searching:

The search operation starts from root node R, if item is less than the value in the root node R, we proceed to the left child; if item is greater than the value in the node R, we proceed to its right child. The process will be continued till the item is found or we reach to a dead end. The Figure 2.6 shows the path taken when searching for a node "c".

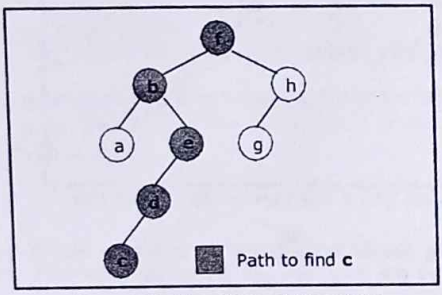


Figure 2.6. Searching a binary tree

Why use binary search trees?

Binary search trees provide an efficient way to search through an ordered collection of items. Consider the alternative of searching an ordered list. The search must proceed sequentially from one end of the list to the other. On average, n/2 nodes must be compared for an ordered list that contains n nodes. In the worst case, all n

nodes might need to be compared. For a large collection of items, this can get very expensive.

The inefficiency is due to the one-dimensionality of a linked list. We would like to have a way to jump into the middle of the list, in order to speed up the search process. In essence, that's what a binary search tree does. The longest path we will ever have to search is equal to the height of the tree. The efficiency of a binary search tree thus depends on the height of the tree. For a tree holding n nodes, the smallest possible height is log(n).

To obtain the smallest height, a tree must be balanced, where both the left and right sub trees have approximately the same number of nodes. Also, each node should have as many children as possible, with all levels being full except possibly the last. Figure 2.7 shows an example of a well-constructed tree.

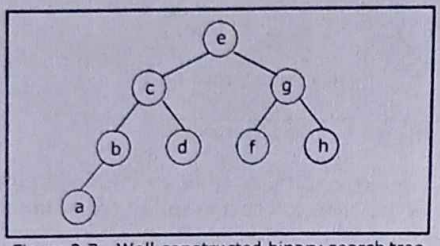


Figure 2.7. Well constructed binary search tree.

Unfortunately, trees can become so unbalanced that they're no better for searching than linked lists. Such trees are called *degenerate trees*. Figure 2.8 shows an example. For a degenerate tree, an average of n/2 comparisons are needed, with a worst case of n comparisons – the same as for a linked list.

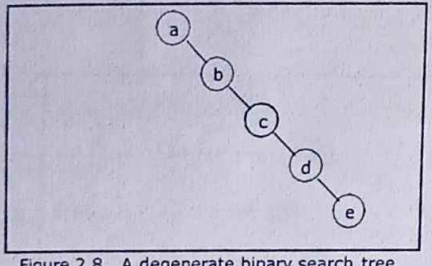


Figure 2.8. A degenerate binary search tree.

When nodes are being added and deleted in a binary search tree, it's difficult to maintain the balance of the tree. We will investigate methods of balancing trees in the next section.

Inserting Nodes into a Binary Search Tree:

When adding nodes to a binary search tree, we must be careful to maintain the binary search tree property. This can be done by first searching the tree to see whether the key we are about to add is already in the tree. If the key cannot be found, a new node is allocated and added at the same location where it would go if

the search had been successful. Figure 2.9 shows what happens when we add some nodes to a tree.

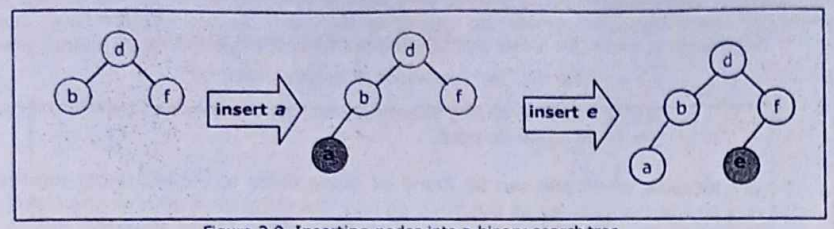


Figure 2.9. Inserting nodes into a binary search tree.

Deleting nodes from a Binary search tree:

Deletions from a binary search tree are more involved than insertions. Given a node to delete, we need to consider these tree cases:

1. The node is a leaf
2. The node has only one child.
3. The node has two children.

Case 1: It is easy to handle because the node can simply be deleted and the corresponding child pointer of its parent set to null. Figure 2.10 shows an example.

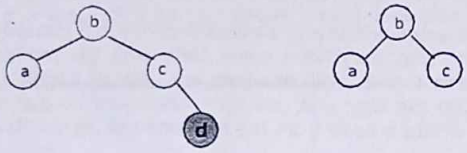


Figure 2. 10. Deleting a leaf node.

Case 2: It is almost as easy to manage. Here, the single child can be promoted up the tree to take the place of the deleted node, as shown in figure 2.11.

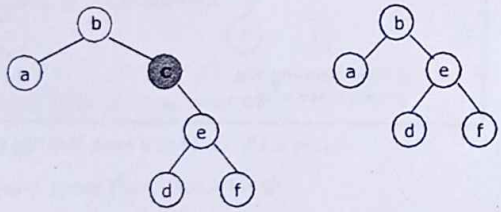


Figure 2. 11. Deleting a node that has one child.

Case 3: The node to be deleted has two children, is more difficult. We must find some node to take the place of the one deleted and still maintain the binary search tree property. There are two obvious cases:

- The inorder predecessor of the deleted node.
- The inorder successor of the deleted node.

We can detach one of these nodes from the tree and insert it where the node to be deleted.

The predecessor of a node can be found by going down to the left once, and then all the way to the right as far as possible. To find the successor, an opposite traversal is used: first to the right, and then down to the left as far as possible. Figure 2.12 shows the path taken to find both the predecessor and successor of a node.

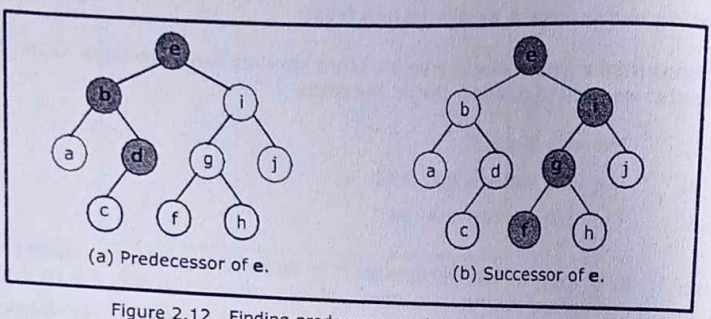


Figure 2.12. Finding predecessor and successor of nodes.

Both the predecessor and successor nodes are guaranteed to have no more than one child, and they may have none. Detaching the predecessor or successor reduces to either case 1 or 2, both of which are easy to handle. In figure 2.13 (a), we delete a node from the tree and use its predecessor as the replacement and in figure 2.13 (b), we delete a node from the tree and use its successor as the replacement.

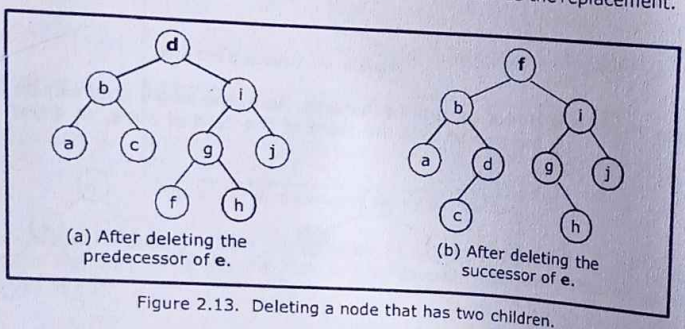


Figure 2.13. Deleting a node that has two children.

Balanced Trees:

For maximum efficiency, a binary search tree should be balanced. Every un-balanced trees are referred to as degenerate trees, so called because their searching performance degenerates to that of linked lists. Figure 2.14 shows an example.

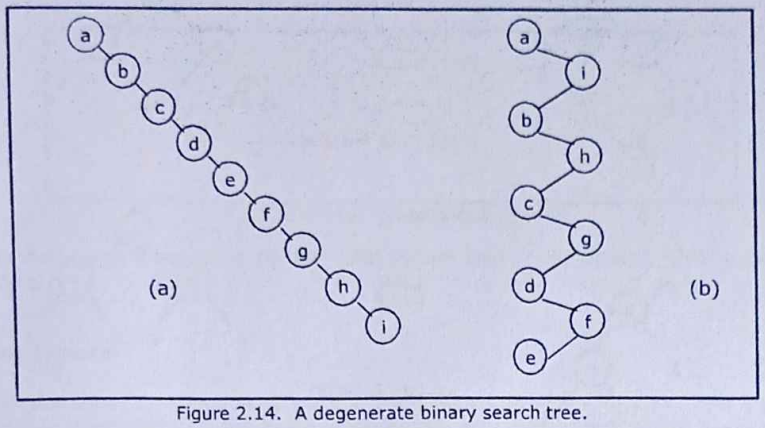


Figure 2.14. A degenerate binary search tree.

The first tree was built by inserting the keys 'a' through 'i' in sorted order into binary search tree. The second tree was built using the insertion sequence. a-g-b-f-c-e-d. This pathological sequence is often used to test the balancing capacity of a tree. Figure 2.15 shows what the tree in 2.14-(b) above would look like if it were *balanced*.

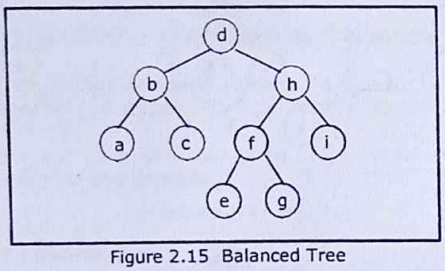


Figure 2.15 Balanced Tree

Balancing Acts:

There are two basic ways used to keep trees balanced.

1) Use tree rotations.
2) Allow nodes to have more than two children.

Tree Rotations:

Certain types of tree-restructurings, known as rotations, can aid in balancing trees. Figure 2.16 shows two types of single rotations.

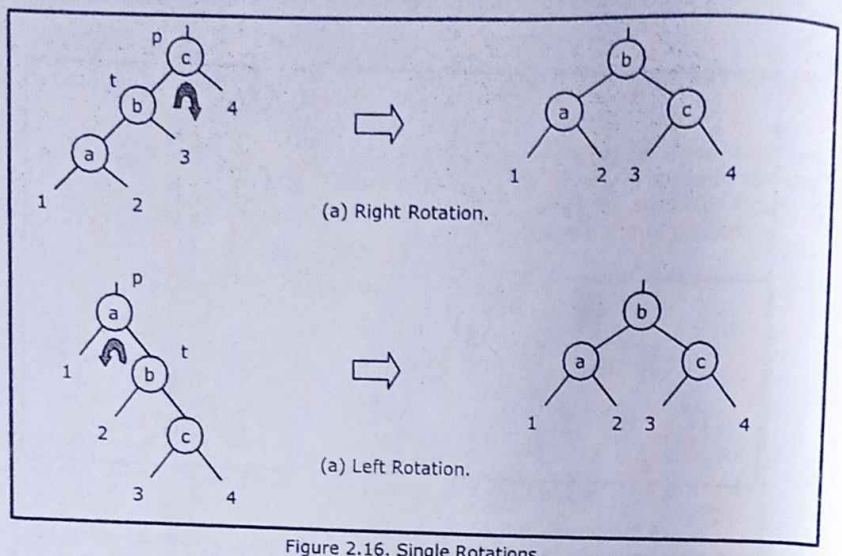


Figure 2.16. Single Rotations

Another type of rotation is known as a double rotation. Figure 2.17 shows the two symmetrical cases.

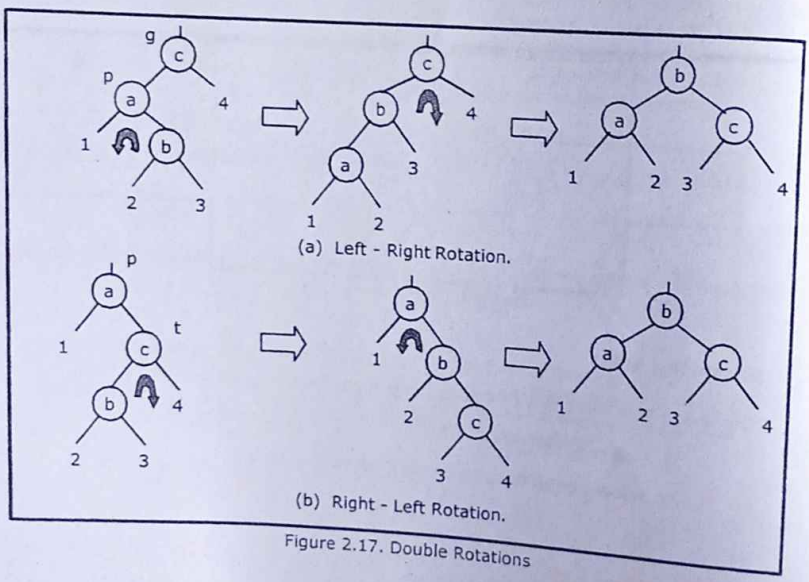


Figure 2.17. Double Rotations

Both single and double rotations have one important feature: in order traversals of the trees before and after the rotations are preserved. Thus, rotations help balance trees, but still maintain the binary search tree property.

Rotations don't guarantee a balanced tree, however for example, figure 2.18 shows a right rotations that makes a tree more un-balanced. The trick lies in determining when to do rotation and what kind of notations to do.

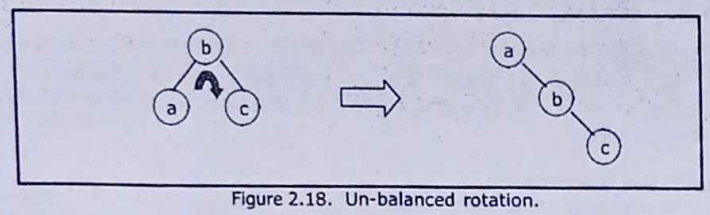


Figure 2.18. Un-balanced rotation.

The Red-black trees have certain rules to be used to determine when a how to rotate.

Dictionary:

A Dictionary is a collection of pairs of form (*k*, *e*), where *k* is a key and *e* is the element associated with the key *k* (equivalently, *e* is the element whose key is *k*). No two pairs in a dictionary should have the same key.

Examples:

1. A word dictionary is a collection of elements; each element comprises a word, which is the key and the meaning of the word, pronunciation and etymologies etc. are associated with the word.

2. A telephone directory with a collection of telephone numbers and names.

3. The list of students enrolled for the data structures course, compiler and Operating System course. The list contains (CourseName, RollID) pairs.

The above last two examples are dictionaries, which permit two or more (key, element) pairs having the same key.

Operations on Dictionaries:

- Get the element associated with a specified key from the dictionary.
  For example, **get(*k*)** returns the element with the key *k*.
- Insert or put an element with a specified key into the dictionary.
  For example, **put(*k*, *e*)** puts the element *e* whose key is *k* into the dictionary.

- Delete or remove an element with a specified key.
  For example, **remove(*k*)** removes the element with key *k*.