UNIT 10 SORTING

Structure **Page Nos.**

10.0 INTRODUCTION

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods. Sorting algorithms can be characterised in the following two ways:

1. Simple algorithms which require the order of n^2 (written as $O(n^2)$)comparisons to sort n items.
2. Sophisticated algorithms that require the $O(n\log_2 n)$ comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order already present in the data.

There are two basic categories of sorting methods: **Internal Sorting and External Sorting.** Internal sorting is applied when the entire collection of data to be sorted is small enough so that the sorting can take place within the main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices. Read and write access times are a major concern in determining sorting performances of such methods.

In this unit, we will study some methods of internal sorting. The next unit will discuss methods of external sorting.

10.1 OBJECTIVES

After going through this unit, you should be able to:

- list the names of some sorting methods;
- discuss the performance of several sorting methods, and
- describe sorting methods on several keys.

10.2 INTERNAL SORTING

In internal sorting, all the data to be sorted is available in the high speed main memory of the computer. We will study the following methods of internal sorting:

1. Insertion sort
2. Bubble sort
3. Quick sort
4. Two-way Merge sort
5. Heap sort

10.2.1 Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example (refer to *Figure 10.1*) before presenting the formal algorithm.

Example : Sort the following list using the insertion sort method:

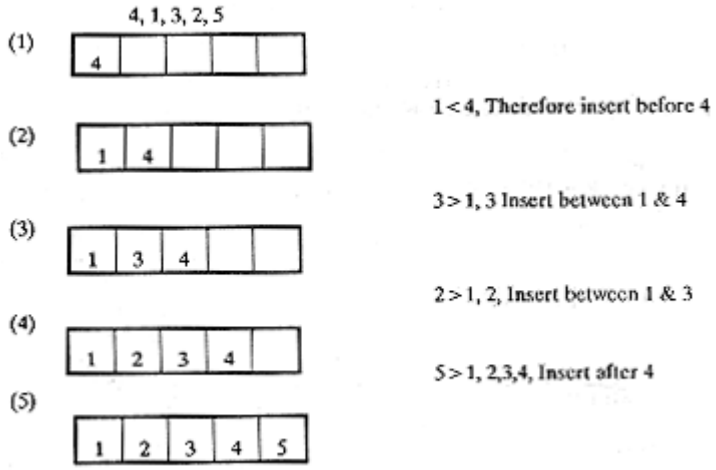


Figure 10.1: Insertion sort

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

10.2.2 Bubble Sort

In this sorting algorithm, multiple swappings take place in one pass. Smaller elements move or 'bubble' up to the top of the list, hence the name given to the algorithm.

In this method, adjacent members of the list to be sorted are compared.If the item on top is greater than the item immediately below it, then they are swapped. This process is carried on till the list is sorted.

The detailed algorithm follows:

Algorithm: BUBBLE SORT

1. Begin

```
2.      Read the n elements
3.      for i=1 to n
        for j=n downto i+1
            if a[j] <=  a[j-1]
                    swap(a[j],a[j-1])
4.      End // of Bubble Sort
```

Total number of comparisons in Bubble sort :

$= (N-1) +(N-2) \ldots + 2 + 1$

$= (N-1)*N / 2 = O(N^2)$

This inefficiency is due to the fact that an item moves only to the next position in each pass.

10.2.3 Quick Sort

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases. The basis of quick sort is the *divide and conquer* strategy i.e. Divide the problem [list to be sorted] into sub-problems [sub-lists], until solved sub problems [sorted sub-lists] are found. This is implemented as follows:

Choose one item A[I] from the list A[ ].

Rearrange the list so that this item is in the proper position, i.e., all preceding items have a lesser value and all succeeding items have a greater value than this item.

1. Place A[0], A[1] .. A[I-1] in sublist 1

2. A[I]

3. Place A[I + 1], A[I + 2] ... A[N] in sublist 2

Repeat steps 1 & 2 for sublist1 & sublist2 till A[ ] is a sorted list.

As can be seen, this algorithm has a recursive structure.

The *divide'* procedure is of utmost importance in this algorithm. This is usually implemented as follows:

1. Choose A[I] as the dividing element.

2. From the left end of the list (A[O] onwards) scan till an item A[R] is found whose value is greater than A[I].

3. From the right end of list [A[N] backwards] scan till an item A[L] is found whose value is less than A[1].

4. Swap A[R] & A[L].

5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.

6. At this point, sublist1 & sublist are ready.

7. Now do the same for each of sublist1 & sublist2.

Program 10.1 gives the program segment for Quick sort. It uses recursion.

```
Quicksort(A,m,n)
int A[ ],m,n
{
        int i, j, k;
        if m<n
        {
                i=m;
                j=n+1;
                k=A[m];
                do
                  do
                     ++i;
                          while (A[i] < k);
                  do
                     --j;
                          while (A[j] > k);
                  if (i < j)
                  {
                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                  }
                     while (i<j);
 temp = A[m];
 A[m] = A[j];
 A[j] = temp;
 Quicksort(A,m,j-1);
 Quicksort(A,j+1,n);
}
```

Program 10.1: Quick Sort

The Quick sort algorithm uses the $O(N \log_2 N)$ comparisons on average. The performance can be improved by keeping in mind the following points.

1. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.

2. Use a better dividing element in the implementations.

It is also possible to write the non-recursive Quick sort algorithm.

10.2.4 2-Way Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea in this is to divide the list into a number of sublists, sort each of these sublists and merge them to get a single sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get n/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called *Concatenate sort*. *Figure 10.2* depicts 2-way merge sort.

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the $0(n \log_2 n)$.

The disadvantage of using mergesort is that it requires two arrays of the same size and space for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

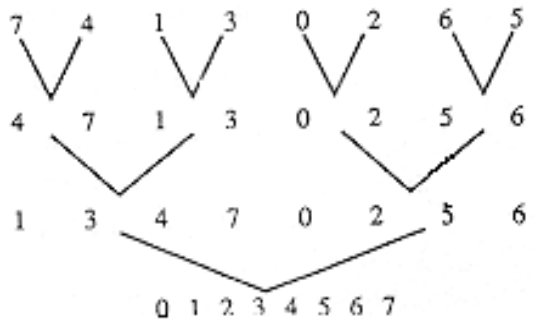


Figure 10.2: 2-way merge sort

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the 0(n $\log_2$n).

The disadvantage of using mergesort is that it requires two arrays of the same size and space for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

10.2.5 Heap Sort

We will begin by defining a new structure called *Heap. Figure 10.3* illustrates a Binary tree.

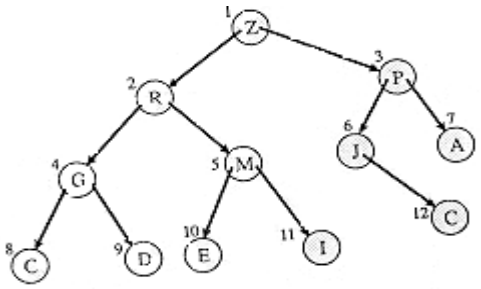


Figure 10.3: A Binary Tree

A complete binary tree is said to satisfy the 'heap condition' if the key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value.

Trees can be represented as arrays, by first numbering the nodes (starting from the root) from left to right. The key values of the nodes are then assigned to array positions whose index is given by the number of the node. For the example tree, the corresponding array is depicted in *Figure 10.4*.

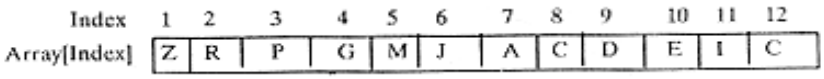


Figure 10.4: Array for the binary tree of figure 10.3

The relationships of a node can also be determined from this array representation. If a node is at position j, its children will be at positions 2j and 2j + 1. Its parent will be at position ⌊J/2⌋.

Consider the node M. It is at position 5. Its parent node is, therefore, at position

$\lfloor 5/2 \rfloor = 2$ i.e. the parent is R. Its children are at positions 2×5 & $(2 \times 5) + 1$, i.e. 10 + 11 respectively i.e. E & I are its children.

A *Heap* is a complete binary tree, in which each node satisfies the heap condition, represented as an array.

We will now study the operations possible on a heap and see how these can be combined to generate a sorting algorithm.

The operations on a heap work in 2 steps.

1. The required node is inserted/deleted/or replaced.
2. The above operation may cause violation of the heap condition so the heap is traversed and modified to rectify any such violations.

Examples: Consider the insertion of a node R in the heap 1.

1. Initially R is added as the right child of J and given the number 13.
2. But, R > J. So, the heap condition is violated.
3. Move R upto position 6 and move J down to position 13.
4. R > P. Therefore, the heap condition is still violated.
5. Swap R and P.
4. The heap condition is now satisfied by all nodes to get the heap of *Figure 10.5*.

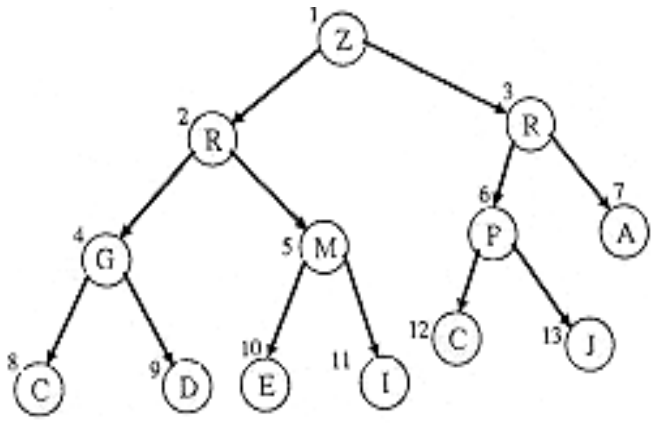


Figure 10.5: A Heap

This algorithm is guaranteed to sort n elements in $(n \log_2 n)$ time.

We will first see two methods of heap construction and then removal in order from the heap to sort the list.

1. Top down heap construction

   - Insert items into an initially empty heap, satisfying the heap condition at all steps.

2. Bottom up heap construction

   - Build a heap with the items in the order presented.

   - From the right most node modify to satisfy the heap condition.

We will exemplify this with an example.

Example: Build a heap of the following using top down approach for heap construction.

PROFESSIONAL

Figure 10.6 shows different steps of the top down construction of the heap.

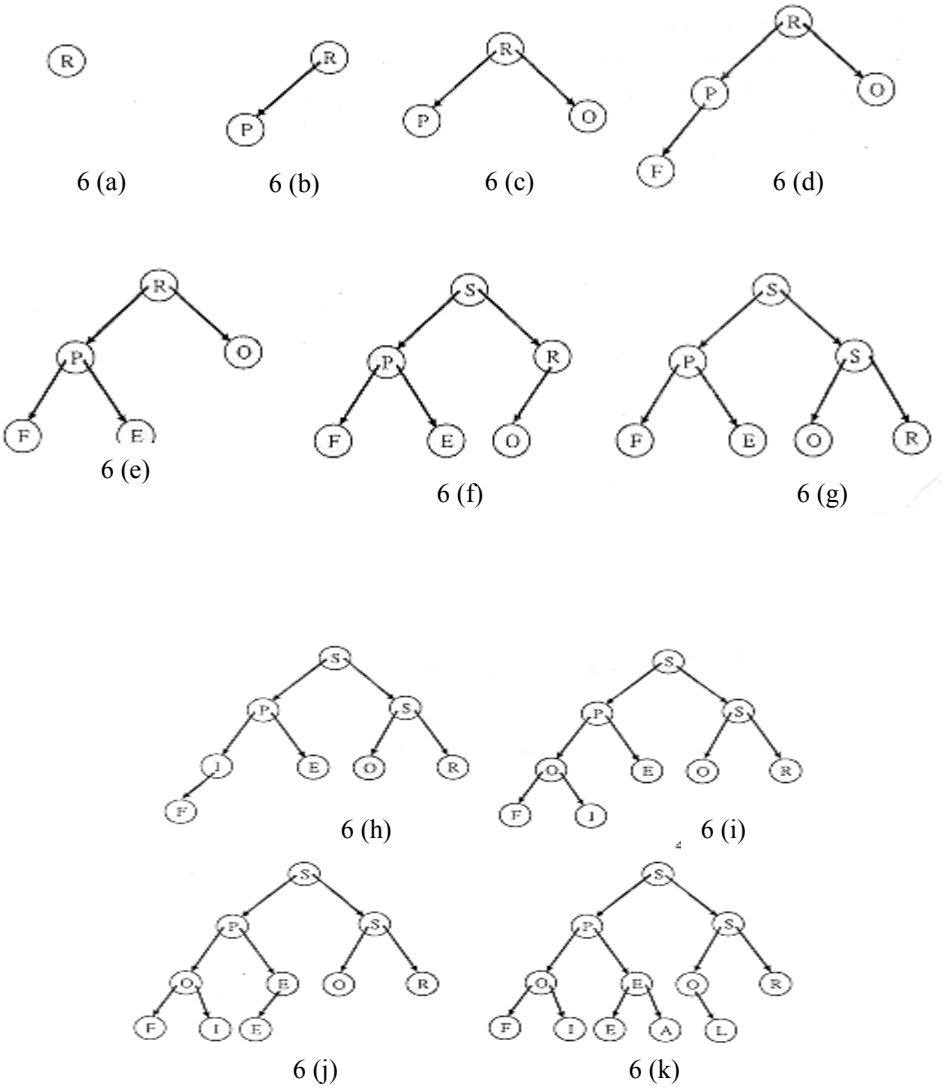


Figure 10.6: Heap Sort (Top down Construction)

Example: The input file is (2,3,81,64,4,25,36,16,9, 49). When the file is interpreted as a binary tree, it results in *Figure 10.7*. *Figure 10.8* depicts the heap.

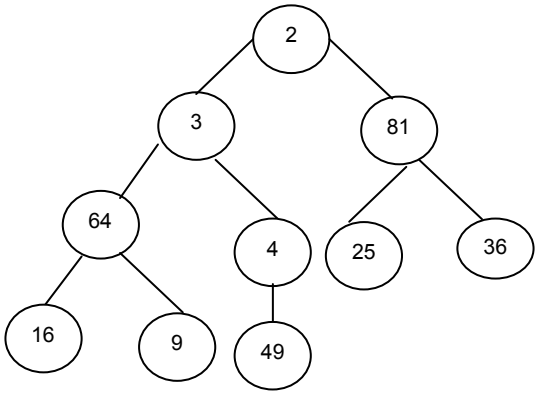


Figure 10.7: A Binary tree

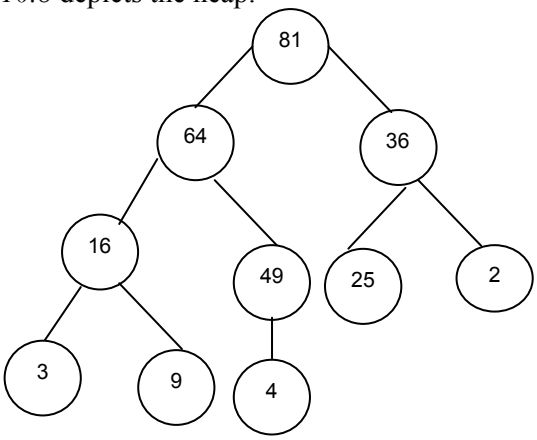
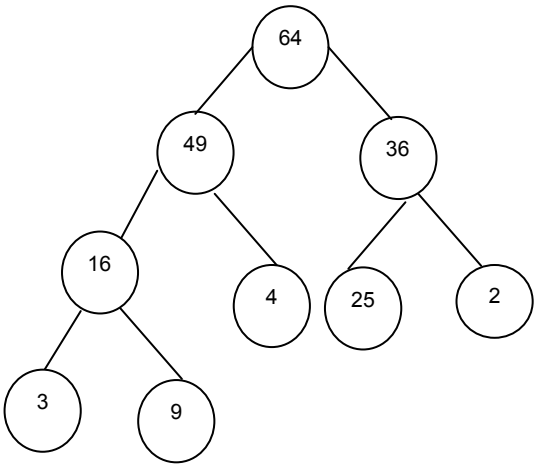
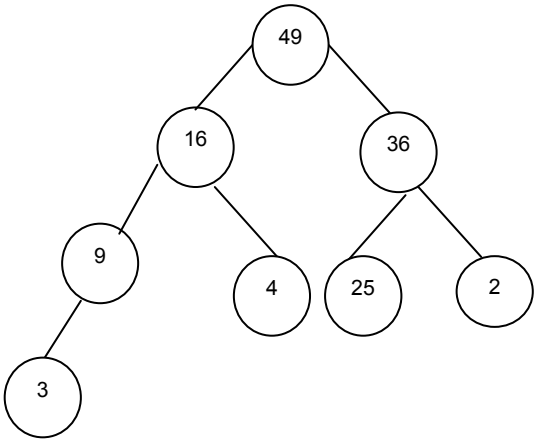


Figure 10.8: Heap of figure 10.7

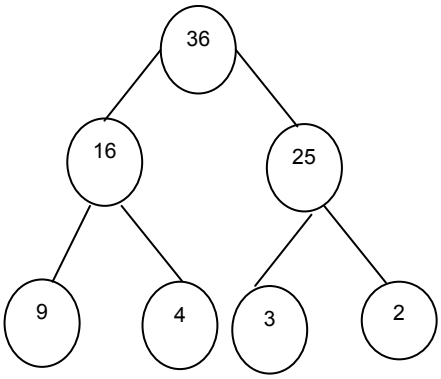
Figure 10.9 illustrates various steps of the heap of *Figure 10.8* as the sorting takes place.



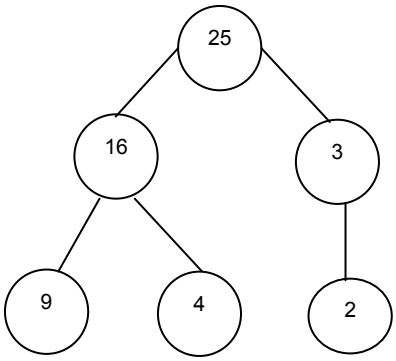
Sorted: 81
Heap size: 9



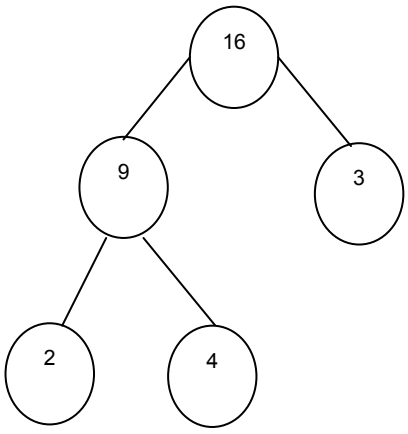
Sorted: 81,64
Heap size: 8



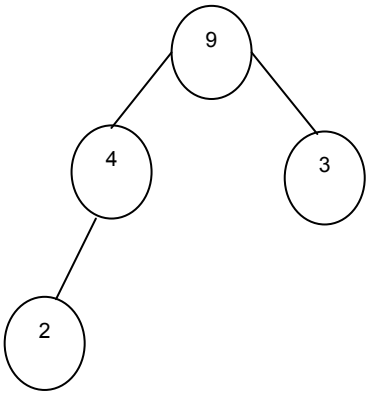
Sorted: 81,64,49
Heap size:7



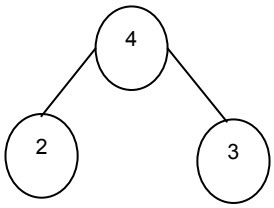
Sorted:81,64,49,36
Heap size:6



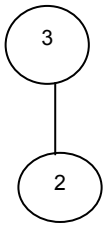
Sorted: 81, 64, 49, 36, 25
Size: 5



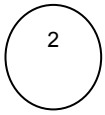
Sorted:81, 64, 49, 36, 25, 16
Size:4



Sorted: 81, 64, 49, 36, 25, 16, 9
Size: 3



Sorted:81, 64, 49, 36, 25, 16, 9, 4
Size: 2



Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3
Size : 1

Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3, 2
Result

Figure 10.9 : Various steps of figure 10.8 for a sorted file

10.3 SORTING ON SEVERAL KEYS

So far, we have been considering sorting based on single keys. But, in real life applications, we may want to sort the data on several keys. The simplest example is that of sorting a deck of cards. The first key for sorting is the suit-clubs, spades, diamonds and hearts. Then, within each suit, sorting the cards in ascending order from Ace, twos to king. This is thus a case of sorting on 2 keys.

Now, this can be done in 2 ways.

1
- Sort the 52 cards into 4 piles according to the suit.
- Sort each of the 4 piles according to face value of the cards.

2
- Sort the 52 cards into 13 piles according to face value.
- Stack these piles in order and then sort into 4 piles based on suit.

The first method is called the MSD (Most Significant Digit) sort and the second method is called the LSD (Least Significant Digit) sort. Digit stands for a key. Though they are called sorting methods, MSD and LSD sorts only decide the *order* of sorting. The actual sorting could be done by any of the sorting methods discussed in this unit.

☞ Check Your Progress 1

1) The complexity of Bubble sort is _____
2) Quick sort algorithm uses the programming technique of _____
3) Write a program in 'C' language for 2-way merge sort.
4) The complexity of Heap sort is _____

10.4 SUMMARY

Sorting is an important application activity. Many sorting algorithms are available. But, each is efficient for a particular situation or a particular kind of data. The choice of a sorting algorithm is crucial to the performance of the application.

In this unit we have studied many sorting algorithms used in internal sorting. This is not a conclusive list and the student is advised to read the suggested books for

exposure to additional sorting methods and for detailed discussions of the methods introduced here.

The following are the *three* most important efficiency criteria:

- use of storage space
- use of computer time
- programming effort.

10.5 SOLUTIONS/ANSWERS

1) $O(N^2)$ where N is the number of elements in the list to be sorted.
2) Divide and Conquer.
3) O(NlogN) where N is the number of elements to be sorted.

10.6 FURTHER READINGS

Reference Books

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein, PHI publications.

2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications.

Reference Websites

http://www.it.jcu.edu.au/Subjects/ cp2001/1998/LectureNotes/Sorting/

http://oopweb.com/Algorithms/Files/Algorithms.html

UNIT 11 ADVANCED DATA STRUCTURES

Structure **Page Nos.**

11.0 INTRODUCTION

In this unit, the following four advanced data structures have been practically emphasized. These may be considered as alternative to a height balanced tree, i.e., AVL tree.

- Splay tree
- Red_Black tree
- AA_tree
- Treap

The key factors which have been discussed in this unit about the above mentioned data structures involve complexity of code in terms of Big oh notation, cost involved in searching a node, the process of deletion of a node and the cost involved in inserting a node.

11.1 OBJECTIVES

After going through this unit, you should be able to:

- know about Splay trees;
- know about Red-Black tree;
- know about AA-trees, and
- know about Treaps.

11.2 SPLAY TREES

Addition of new records in a Binary tree structure always occurs as leaf nodes, which are further away from the root node making their access slower. If this new record is to be accessed very frequently, then we cannot afford to spend much time in reaching it but would require it to be positioned close to the root node. This would call for readjustment or rebuilding of the tree to attain the desired shape. But, this process of rebuilding the tree every time as the preferences for the records change is tedious and time consuming. There must be some measure so that the tree adjusts itself automatically as the frequency of accessing the records changes. Such a self-adjusting tree is the Splay tree.

Splay trees are self-adjusting binary search trees in which every access for insertion or retrieval of a node, lifts that node all the way up to become the root, pushing the other nodes out of the way to make room for this new root of the modified tree. Hence, the frequently accessed nodes will frequently be lifted up and remain around the root position; while the most infrequently accessed nodes would move farther and farther away from the root.

This process of readjusting may at times create a highly imbalanced splay tree, wherein a single access may be extremely expensive. But over a long sequence of accesses, these expensive cases may be averaged out by the less expensive ones to produce excellent results over a long sequence of operations. The analytical tool used for this purpose is the Amortized algorithm analysis. This will be discussed in detail in the following sections.

11.2.1 Splaying Steps

Readjusting for tree modification calls for rotations in the binary search tree. Single rotations are possible in the left or right direction for moving a node to the root position. The task would be achieved this way, but the performance of the tree amortized over many accesses may not be good.

Instead, the key idea of splaying is to move the accessed node two levels up the tree at each step. Basic terminologies in this context are:

Zig: Movement of one step down the path to the left to fetch a node up.
Zag: Movement of one step down the path to the right to fetch a node up.

With these two basic steps, the possible splay rotations are:

Zig-Zig: Movement of two steps down to the left.
Zag-Zag: Movement of two steps down to the right.
Zig-Zag: Movement of one step left and then right.
Zag-Zig: Movement of one step right and then left.

Figure 11.1 depicts the splay rotations.

Zig:

Zig-Zig:

Figure 11.1: Splay rotations

Splaying may be top-down or bottom-up. In bottom-up splaying, splaying begins at the accessed node, moving up the chain to the root. While in top-down splaying, splaying begins from the top while searching for the node to access. In the next section, we would be discussing the top-down splaying procedure:

As top-down splaying proceeds, the tree is split into three parts:

a) **Central SubTree**: This is initially the complete tree and may contain the target node. Search proceeds by comparison of the target value with the root and ends with the root of the central tree being the node containing the target if present or null node if the target is not present.

b) **Left SubTree**: This is initially empty and is created as the central subtree is splayed. It consists of nodes with values less than the target being searched.

c) **Right SubTree**: This is also initially empty and is created similar to left subtree. It consists of nodes with values more than the target node.

Figure 11.2 depicts the splaying procedure with an example, attempting to splay at 20.

Initially,

The first step is Zig-Zag:

The next step is Zig-Zig:

The next step is the terminal zig:

Finally, reassembling the three trees, we get:

Figure 11.2: Splaying procedure

11.2.2 Splaying Algorithm

Insertion and deletion of a target key requires splaying of the tree. In case of insertion, the tree is splayed to find the target. If, target key is found, then we have a duplicate and the original value is maintained. But, if it is not found, then the target is inserted as the root.

In case of deletion, the target is searched by splaying the tree. Then, it is deleted from the root position and the remaining trees reassembled, if found.

Hence, splaying is used both for insertion and deletion. In the former case, to find the proper position for the target element and avoiding duplicity and in the latter case to bring the desired node to root position.

Splaying procedure

For splaying, three trees are maintained, the central, left and right subtrees. Initially, the central subtree is the complete tree and left and right subtrees are empty. The target key is compared to the root of the central subtree where the following two conditions are possible:

a) Target > Root: If target is greater than the root, then the search will be more to the right and in the process, the root and its left subtree are shifted to the left tree.

b) Target < Root: If the target is less than the root, then the search is shifted to the left, moving the root and its right subtree to right tree.

We repeat the comparison process till either of the following conditions are satisfied:

a) Target is found: In this, insertion would create a duplicate node. Hence, original node is maintained. Deletion would lead to removing the root node.

b) Target is not found and we reach a null node: In this case, target is inserted in the null node position.

Now, the tree is reassembled. For the target node, which is the new root of our tree, the largest node is the left subtree and is connected to its left child and the smallest node in the right subtree is connected as its right child.

Amortized Algorithm Analysis

In the amortized analysis, the time required to perform a set of operations is the average of all operations performed. Amortized analysis considers a long sequence of operations instead of just one and then gives a worst-case estimate. There are three different methods by which the amortized cost can be calculated and can be differentiated from the actual cost. The three methods, namely, are:

- Aggregate analysis: It finds the average cost of each operation. That is, T(n)/n. The amortized cost is same for all operations.
- Accounting method: The amortized cost is different for all operations and charges a credit as prepaid credit on some operations.
- Potential method: It also has different amortized cost for each operation and charges a credit as the potential energy to other operations.

There are different operations such as stack operations (push, pop, multipop) and an increment which can be considered as examples to examine the above three methods.

Every operation on a splay tree and all splay tree operations take O(log n) amortized time.

☞ Check Your Progress 1

1) Consider the following tree. Splay at node 2.

11.3 RED-BLACK TREES

A Red-Black Tree (RBT) is a type of Binary Search tree with one extra bit of storage per node, i.e. its color which can either be red or black. Now the nodes can have any of the color (red, black) from root to a leaf node. These trees are such that they guarantee O(log n) time in the worst case for searching.

Each node of a red black tree contains the field color, key, left, right and p (parent). If a child or a parent node does not exist, then the pointer field of that node contains NULL value.

11.3.1 Properties of a Red-Black Tree

Any binary search tree should contain following properties to be called as a red-black tree.

1. Each node of a tree should be either red or black.

2. The root node is always black.

3. If a node is red, then its children should be black.

4. For every node, all the paths from a node to its leaves contain the same number of black nodes.

We define the number of black nodes on any path from but not including a node x down to a leaf, the black height of the node is denoted by bh (x).

Figure 11.3 depicts a Red-Black Tree.

Figure 11.3: A Red-Black tree

Red-black trees contain two main operations, namely INSERT and DELETE. When the tree is modified, the result may violate red-black properties. To restore the tree properties, we must change the color of the nodes as well as the pointer structure. We can change the pointer structure by using a technique called rotation which preserves inorder key ordering. There are two kinds of rotations: left rotation and right rotation (refer to *Figures 11.4 and 11.5*).

Figure 11.4: Left rotation **Figure 11.5: Right rotation**

When we do a left rotation on a node y, we assume that its right child x is non null. The left rotation makes x as the new root of the subtree with y as x's left child and x's left child as y's right child.

The same procedure is repeated vice versa for the right rotation.

11.3.2 Insertion into a Red-Black Tree

The insertion procedure in a red-black tree is similar to a binary search tree i.e., the insertion proceeds in a similar manner but after insertion of nodes x into the tree T, we color it red. In order to guarantee that the red-black properties are preserved, we then fix up the updated tree by changing the color of the nodes and performing rotations. Let us write the pseudo code for insertion.

The following are the two procedures followed for insertion into a Red-Black Tree:

Procedure 1: This is used to insert an element in a given Red-Black Tree. It involves the method of insertion used in binary search tree.

Procedure 2: Whenever the node is inserted in a tree, it is made red, and after insertion, there may be chances of loosing Red-Black Properties in a tree, and, so, some cases are to be considered inorder to retain those properties.

During the insertion procedure, the inserted node is always red. After inserting a node, it is necessary to notify that which of the red-black properties are violated. Let us now look at the execution of fix up. Let Z be the node which is to be inserted and is colored red. At the start of each iteration of the loop,

1. Node Z is red
2. If P(Z) is the root, then P(Z) is black
3. Now if any of the properties i.e. property 2 is violated if Z is the root and is red OR when property 4 is violated if both Z and P(Z) are red, then we consider 3 cases in the fix up algorithm. Let us now discuss those cases.

Case 1(Z's uncle y is red): This is executed when both parent of Z (P(Z)) and uncle of Z, i.e. y are red in color. So, we can maintain one of the property of Red-Black tree by making both P(Z) and y black and making point of P(Z) to be red, thereby maintaining one more property. Now, this while loop is repeated again until color of y is black.

Case 2 (Z's uncle is black and Z is the right child): So, make parent of Z to be Z itself and apply left rotation to newly obtained Z.

Case 3 (Z's uncle is black and Z is the left child): This case executes by making parent of Z as black and P(P(Z)) as red and then performing right rotation to it i.e., to (P(Z)).

The above 3 cases are also considered conversely when the parent of Z is to the right of its own parent. All the different cases can be seen through an example.Consider a red-black tree drawn below with a node z (17 inserted in it) (refer to *Figure 11.6*).

Figure 11.6: A Red-Black Tree after insertion of node 17

Before the execution of any case, we should first check the position of P(Z) i.e. if it is towards left of its parent, then the above cases will be executed but, if it is towards the right of its parent, then the above 3 cases are considered conversely.

Now, it is seen that Z is towards the left of its parent (refer to *Figure 11.7*). So, the above cases will be executed and another node called y is assigned which is the uncle of Z and now cases to be executed are as follows:

Figure 11.7: Z is to the left of it's parent

Case 1: Property 4 is violated as both z and parent(z) are red (refer to *Figure 11.8*).

Figure 11.8: Both Z and P(Z) are red

Now, let us check to see which case is executed.

Case 2: The application of this case results in *Figure 11.9*.

Figure 11.9: Result of application of case-2

Case 3: The application of this case results in *Figure 11.10*.

Figure 11.10: Result of application of case-3

Finally, it resulted in a perfect Red-Black Tree (*Figure 11.10*).

11.3.3 Deletion from a Red-Black Tree

Deletion in a RBT uses two main procedures, namely,

Procedure 1: This is used to delete an element in a given Red-Black Tree. It involves the method of deletion used in binary search tree.

Procedure 2: Whenever the node is deleted from a tree, and after deletion, there may be chances of losing Red-Black Properties in a tree and so, some cases are to be considered in order to retain those properties.

This procedure is called only when the successor of the node to be deleted is Black, but if y is red, the red- black properties still hold and for the following reasons:

- No red nodes have been made adjacent
- No black heights in the tree have changed
- y could not have been the root

Now, the node (say x) which takes the position of the deleted node (say z) will be called in procedure 2. Now, this procedure starts with a loop to make the extra black up to the tree until

- X points to a black node
- Rotations to be performed and recoloring to be done
- X is a pointer to the root in which the extra black can be easily removed

This while loop will be executed until x becomes root and its color is red. Here, a new node (say w) is taken which is the sibling of x.

There are four cases which we will be considering separately as follows:

Case 1: If color of w's sibling of x is red
Since W must have black children, we can change the colors of w and p (x) and then left rotate p (x) and the new value of w to be the right node of parent of x. Now, the conditions are satisfied and we switch over to case 2, 3 and 4.

Case 2: If the color of w is black and both its children are also black.

Since w is black, we make w to be red leaving x with only one black and assign parent (x) to be the new value of x. Now, the condition will be again checked, i.e. x = left (p(x)).

Figure 11.11: Application of case-1

Figure 11.12: Application of case-2

Figure 11.13: Application of case-3

Figure 11.14: Application of case-4

Case 3: If the color of w is black, but its left child is red and w's right child is black.

After entering case-3, we change the color of left child of w to black and that of w to be red and then perform right rotation on w without violating any of the black properties. The new sibling w of x is now a black node with a red right child and thus case 4 is obtained.

Case 4: When w is black and w's right child is red.

This can be done by making some color changes and performing a left rotation on p(x). We can remove the extra black on x, making it single black. Setting x to be the root causes the while loop to terminate.

Note: In the above *Figures 11.11, 11.12, 11.13* and *11.14*, α, α', β, β', γ, ε are assumed to be either red or black depending upon the situation.

11.4 AA-Trees

Red-Black trees have introduced a new property in the binary search tree, i.e., an extra property of color (red, black). But, as these trees grow, in their operations like insertion, deletion, it becomes difficult to retain all the properties, especially in case of deletion. Thus, a new type of binary search tree can be described which has no property of having a color, but has a new property introduced based on the color which is the information for the new. This information of the level of a node is stored in a small integer (may be 8 bits). Now, AA-trees are defined in terms of level of each node instead of storing a color bit with each node. A red-black tree used to have various conditions to be satisfied regarding its color and AA-trees have also been designed in such a way that it should satisfy certain conditions regarding its new property, i.e., level.

The level of a node will be as follows:

1. Same of its parent, if the node is red.
2. One if the node is a leaf.
3. Level will be one less than the level of its parent, if the node is black.

Any red-black tree can be converted into an AA-tree by translating its color structure to levels such that left child is always one level lower than its parent and right child is always same or at one level lower than its parent. When the right child is at same level to its parent, then a horizontal link is established between them. Thus, we conclude that it is necessary that horizontal links are always at the right side and that there may not be two consecutive links. Taking into consideration of all the above properties, we show a AA-tree as follows (refer to *Figure 11.15*).

Figure 11.15: AA-tree

After having a look at the AA-tree above, we now look at different operations that can be performed at such trees.

The following are various operations on a AA-tree:

1. Searching: Searching is done by using an algorithm that is similar to the search algorithm of a binary search tree.
2. Insertion: The insertion procedure always start from the bottom level. But, while performing this function, either of the two problems can occur:

   (a) Two consecutive horizontal links (right side)
   (b) Left horizontal link.

While studying the properties of AA-tree, we said that conditions (a) and (b) should not be satisfied. Thus, in order to remove conditions (a) and (b), we use two new functions namely skew( ) and split( ) based on the rotations of the node, so that all the properties of AA-trees are retained.

The condition that (a) two consecutive horizontal links in an AA-tree can be removed by a left rotation by split( ) whereas the condition (b) can be removed by right rotations through function show( ). Either of these functions can remove these condition, but can also arise the other condition. Let us demonstrate it with an example. Suppose, in the AA-tree of *Figure 11.15*, we have to insert node 50.

According to the condition, the node 50 will be inserted at the bottom level in such a way that it satisfies Binary Search tree property also (refer to *Figure 11.16*).

Figure 11.16: After inserting node 50

Figure 11.17: Split at node 39(left rotation)

Now, we should be aware as to how this left rotation is performed. Remember, that rotation is introduced in Red-black tree and these rotations (left and right) are the same as we performed in a Red-Black tree. Now, again split ( ) has removed its condition but has created skew conditions (refer to *Figure 11.17*). So, skew ( ) function will now be called again and again until a complete AA-tree with a no false condition is obtained.

Figure 11.18: Skew at 55 (right rotation)

Figure 11.19: Split at 45

A skew problem arises because node 90 is two-level lower than its parent 75 and so in order to avoid this, we call skew / split function again.

Figure 11.20: The Final AA-tree

Thus, introducing horizontal left links, in order to avoid left horizontal links and making them right horizontal links, we make 3 calls to skew and then 2 calls to split to remove consecutive horizontal links (refer to *Figures 11.18, 11.19* and *11.20*).

A Treap is another type of Binary Search tree and has one property different from other types of trees. Each node in the tree stores an item, a left and right pointer and a priority that is randomly assigned when the node is created. While assigning the priority, it is necessary that the heap order priority should be maintained: node's priority should be at least as large as its parent's. A treap is both binary search tree with respect to node elements and a heap with respect to node priorities.

☞ Check Your Progress 2

1) Explain the properties of red-black trees along with an example.

……

……

11.5 SUMMARY

This is a unit of which focused on the emerging data structures. Splay trees, Red-Black trees, AA-trees and Treaps are introduced. The learner should explore the possibilities of applying these concepts in real life.

Splay trees are binary search trees which are self adjusting. *Self adjusting* basically means that whenever a splay tree is accessed for insertion or deletion of a node, then that node pushes all the remaining nodes to become root. So, we can conclude that any node which is accessed frequently will be at the top levels of the Splay tree.

A Red-Black tree is a type of binary search tree in which each node is either red or black. Apart from that, the root is always black. If a node is red, then its children should be black. For every node, all the paths from a node to its leaves contain the same number of black nodes.

AA-trees are defined in terms of level of each node instead of storing a color bit with each node. AA-trees have also been designed in such a way that it should satisfy certain conditions regarding its new property i.e. level.

The priorities of nodes of a Treap should satisfy the heap order. Hence, the priority of any node must be as large as it's parent's. Treap is the simplest of all the trees.

11.6 SOLUTIONS/ANSWERS

Check Your Progress 1

Ans. 1

Check Your Progress 2

1) Any Binary search tree should contain following properties to be called as a red-black tree.

1. Each node of a tree should be either red or black.
2. The root node is always black.
3. If a node is red then its children should be black.
4. For every node, all the paths from a node to its leaves contain the same number of black nodes.

Example of a red-black tree:

11.7 FURTHER READINGS

Reference Books

1. *Data Structures and Algorithm Analysis in C by Mark Allen Weiss, Pearson Education*

Reference Websites

http://www.link.cs.cmu.edu/splay/

http://www.cs.buap.mx/~titab/files/AATrees.pdf

UNIT 12 FILE STRUCTURES

Structure **Page Nos.**

12.0 INTRODUCTION

The structures of files change from operating system to operating system. In this unit, we shall discuss the fundamentals of file structures along with the generic file organisations.

A file may be defined as a collection of records. Though, a text file doesn't conform to this definition at the first glance, it is also a collection of records and records are words in the file.

Consider a file consisting of information about students. We may name such a file as *Student* file. The typical records of such file are shown in *Figure 12.1*.

Enum	Name	Address	State	Country	Programme
012786345	John	D-51, Nebsarai, Maidan Garhi	Delhi	India	BCA
98387123	Suresh	E-345, Banjara Hills	Hyderabad	India	MCA

Figure 12.1: Typical records of a *Student* file

A file should always be stored in such a way that the basic operations on it can be performed easily. In other words, queries should be able to be executed with out much hassle. We focus, in this unit, on the ways of storing the files on external storage devices. Selection of a particular way of storing the file on the device depends on factors such as retrieval records, the way the queries can be put on the file, the total number of keys in each record etc.

12.1 OBJECTIVES

After going through this unit, you should be able to

- learn the terminology of file structures;
- learn the underlying concepts of Sequential files, and
- know the Indexed sequential file organisation.

12.2 TERMINOLOGY

The following are the definitions of some important terms:

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 application point of view.

   For example:

A university could use a student record with the fields, namely, University enrolment no. and names of courses.

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 students 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.

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

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

ii) **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

   - CREATION of the file
   - INSERTION of records into the file
   - UPDATION of previously inserted records
   - RETRIEVAL of previously inserted records
   - DELETION of records
   - DELETION of the file.

12.3 FILE ORGANISATION

File organisation may be defined as a method of storing records in file. Also, the subsequent implications on the way these records can be accessed. The following are the factors involved in selecting a particular file organisation:

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

Different file organisations accord different weightages to the above factors. 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.

- **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 n^{th} record, one must traverse the preceding (n–1) records. Sequential files will be dealt with at length in the next section.

- **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.

- **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 his application.

- **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.

- **Indexed Flies**

  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.

12.4 SEQUENTIAL FILES

In this section, we shall discuss about Sequential file organisation. 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.

12.4.1 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.

12.4.2 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.

The common method is to use transaction logging. This works as follows:

- It collects records for insertion in a transaction file in their order of their arrival.
- When population of the transactions file has ceased, sort the transaction file in the order of the key of the primary data file
- 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. **Updation:** Updation is a combination of insertion and deletions. The record with the new values 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.

12.4.3 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.

12.4.4 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. Example is payroll applications.

Check Your Progress

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

……

……

……

.....

12.5 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) = Address

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

A calculation is 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 discus 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 actually for the key values in the file. Hence, calculated address may not be unique. It is called Collision, i.e.,

R(K1) = R(K2), where K1 = 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. The student is advised to read some text on bucket Addressing and related topics.

12.6 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 Indexed 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, 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. Consider the 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 used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the contents of the static index 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. Recall the change in height of B-Tree as records are inserted and deleted.

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

A directory is a component of file. Consider a file which doesn't have any keys for its records. When a query is executed on such a file, the time consumed to execute the query is more when compared to another file which is having keys, because, there may be arising necessity where in the file has to be sorted on the field(s) on which the query is based. So, for each query, the file has to be sorted on the field on which the query is based which is cumbersome. In the case of files which have keys, different versions of the files which result due to sorting on the keys are stored in the directory of that file. Such files are called index files and the number of index files vary from file to file. For example, consider the file of *Figure 12.1*. If we designate Enrolment Number (Enum) and Name as keys, then we may have two index files based on the each key. Of course, we can have more than two index files, to deal with queries which use both the keys. Different software store index files in a different manner so that the operations on the records can be performed as soon as possible after the query is submitted.

One of the prominent indexing techniques is Cylinder-Surface indexing.

Since, there exists a primary key for each of the files, there will be an index file based on the primary key. Cylinder-Surface Indexing is useful for such index file. In this type of indexing, the records of the file are stored one after another in such a way that the primary keys of the records are in increasing order. The index file will have two fields. They are cylinder index and corresponding surface indexes. There will be multiple cylinders and there are multiple surfaces corresponding to each cylinder. Suppose that the file needs m cylinders, then cylinder index will have m entries. Each cylinder will be having one entry which corresponds to the largest key value in that cylinder. Assume that the disk has n surfaces which can be used. Then, each surface index has n entries. The k-th entry in surface index for cylinder l^{th} cylinder if the value of the largest key on the l^{th} track of the k^{th} surface. Hence, m.n indicates the total number of surface index entries.

Suppose that the need arises to search for a record whose key value is B. Then, the first step is to load the cylinder index of the file into memory. Usually, each cylinder index occupies only one track as the number of cylinders are only few. The cylinder which holds the desired record is found by searching the cylinder index. Usually, the search takes O(log m) time. After the search of cylinder index, the corresponding cylinder is determined. Based on the cylinder, the corresponding surface index is retrieved to look the record for which the search has started. Whenever a search is initiated for the surface index, usually sequential search is used. Of course, it depends on the number of surfaces. But, usually, the number of surfaces are less. After finding the cylinder to be accessed and after finding the surface to be accessed, the corresponding track is loaded into memory and that track is searched for the needed record.

12.7 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 techniques were discussed. Sequential File Organisation finds 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 to identify a particular record 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.

12.8 SOLUTIONS/ANSWERS

Check Your Progress

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.

12.9 FURTHER READINGS

Reference Books

1. *Fundamentals of Data Structures in C++* by E. Horowitz, Sahani and D. Mehta, Galgotia Publications.

2. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J. Augenstein and Aaron M. Tanenbaum, PHI Publications.

3. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

Reference Websites

http:// www.cs.umbc.edu

http://www.fredosaurus.com