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