MIT World Peace University
Advanced Data Structures

Theory Assignment

Naman Soni Roll No. 10

Contents

1 Explain AA Trees with an example.

Ans. AA trees are a self-balancing binary search tree that were invented by Arne Andersson in 1993. They are similar to red-black trees and provide efficient average-case performance for a variety of operations, including search, insertion, and deletion.

AA trees are balanced by enforcing two properties:

- **Level property:** Every leaf node has the same level, which is the distance from the root.
- **Skew property:** Every right child of a node has a level less than or equal to its parent, and every left child of a node has a level strictly less than its parent.

To maintain these properties, AA trees use two operations:

- **Skew:** If a node has a right child whose level is the same as its own, rotate the node to the left so that its right child becomes its parent.
- **Split:** If a node has two left children whose levels are the same, rotate the node to the right so that its left child becomes its parent, and then increment the level of the new right child.

Here's an example of an AA tree:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
```

In this tree, every leaf node has the same level (4), and the skew property is satisfied because every right child has a level less than or equal to its parent, and every left child has a level strictly less than its parent.

To perform an insertion, let's insert the value 15 into the tree. First, we traverse the tree as we would in a normal binary search tree until we find the appropriate location to insert the new node:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25

Insert 15:
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
           \
           15
```

Now, we need to rebalance the tree to maintain the level and skew properties. We start by performing a skew operation on the new node's parent:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
         /   \
        15   19
```

Next, we perform a split operation on the new node's grandparent:

```
      12
    /    \
   7     18
  / \   /  \
 3   9 15   20
/ \        / \
1   5      19  25
```

Now, the tree is balanced according to the level and skew properties.

2 Explain B+ Tree with an example.

Ans. A B+ tree is a type of self-balancing tree that is commonly used for indexing and organizing large amounts of data in databases and file systems. It has a similar structure to a B-tree but with some key differences, such as all data being stored in the leaf nodes and internal nodes only containing keys.

Here's an example of a B+ tree:

```
            +----+
            | 30 |
            +----+
         /          \
     +----+          +----+
     | 50 |          | 70 |
     +----+          +----+
    /      \        /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
```

In this example, each node has a maximum of three keys and four pointers to child nodes. The tree is balanced because all paths from the root to the leaf nodes have the same length.

To perform a search in the B+ tree, we start at the root and compare the search key to the keys in the node. If the search key is less than the first key, we follow the leftmost pointer to the next node. If the search key is greater than or equal to the last key, we follow the rightmost pointer to the next node. Otherwise, we follow the pointer to the node whose key range contains the search key.

For example, if we wanted to search for the value 20 in the above tree, we would start at the root node and find that the key range of the leftmost child node is less than 20, but the key range of the second child node contains 20, so we would follow the pointer to that node:

```
             +----+
             | 30 |
             +----+
            /       \
    +----+            +----+
    | 50 |            | 70 |
    +----+            +----+
   /      \          /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
                    ^
                    |
                search
```

Then, we would search the leaf node for the value 20 and find that it is present in the node:

```
    +----+
    | 20 |
    +----+
     /  \
```

```
  prev  next
```

To perform an insertion in the B+ tree, we start at the root and traverse the tree to find the leaf node where the new key should be inserted. If the node has room for the new key, we insert it and update the keys in the internal nodes on the path from the root to the leaf. If the node is full, we split it into two nodes and promote the median key to the parent node. This may propagate up the tree, causing additional splits and promotions until the root is reached and possibly requiring the tree to grow taller.

For example, if we wanted to insert the value 35 into the above tree, we would start at the root node and find that the key range of the rightmost child node is greater than 35, so we would follow the pointer to that node:

```
            +----+
            | 30 |
            +----+
          /        \
    +----+          +----+
    | 50 |          | 70 |
    +----+          +----+
    /      \        /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
                   ^
                   |
                 insert
```

Since the node is full, we split it into two nodes and promote the median key (40) to the parent node:

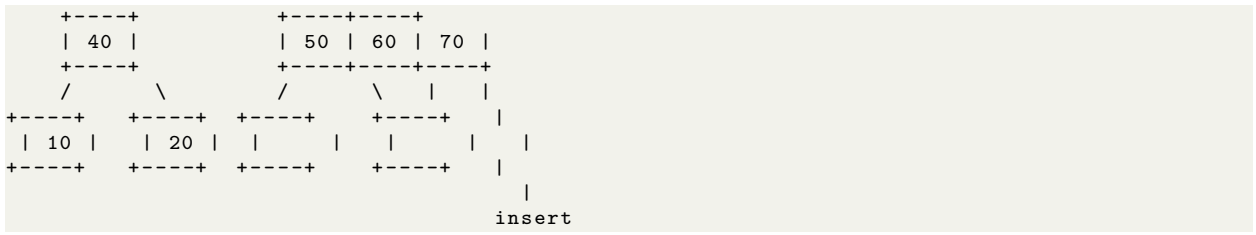
```
        +----+
        | 30 |
        +----+
      /        \
+----+          +----+----+
| 40 |          | 50 | 70 |
+----+          +----+----+
/      \           /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  |    |   | 60 |
+----+   +----+  +----+   +----+
                    ^
                    |
                  insert
```

Now, we want to insert the value 45 into the tree. We start at the root node and find that the key range of the rightmost child node is greater than 45, so we follow the pointer to that node:

```
          +----+
          | 30 |
          +----+
        /        \
   +----+          +----+----+
   | 40 |          | 50 | 70 |
   +----+          +----+----+
  /      \        /      \   |
+----+   +----+  +----+   +----+
| 10 |   | 20 |  |    |   | 60 |
+----+   +----+  +----+   +----+
                   ^
                   |
                 insert
```

Since the node is full, we split it into two nodes and promote the median key (50) to the parent node:

```
            +----+
            | 30 |
            +----+
          /        \
```



Now, the B+ tree has been updated with the new value 45. The tree remains balanced and all paths from the root to the leaf nodes still have the same length.‘’