AA Trees, Treaps
Observations

There are many variations of balanced binary trees. The prominent among them are Red Black trees, B-trees (again of many kinds), weak AVL trees etc.; they all share the same property that (1) height of the tree is $O(\log_2 n)$, where $n$ is the number of nodes and (2) the worst-case time for each of the operations of search, insert, delete is $O(\log_2 n)$.

Limitations of Balanced Search Trees: Balanced search trees require storing an extra piece of information per node. Their worst-case, average-case, and best-case performance are essentially identical. We do not win when easy inputs occur – would be nice if the second access to the same piece of data was cheaper than the first. The 90-10 rule – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items; it would be nice to get easy wins for the 90% case.

An AA tree is another kind of self-balancing binary search tree, a variation of Red Black trees, where there are fewer rotation cases so easier to code, especially deletions (eliminates about half of the rotation cases). [The implementation and number of rotation cases in Red-Black Trees is relatively complex]. AA-trees still have $O(\log n)$ searches in the worst-case, although they are slightly less efficient empirically.
AA-Tree Ordering Properties

An AA-Tree is a binary search tree with all the ordering properties of a red-black tree:

1. Every node is colored either red or black!
2. The root is black
3. External nodes are black
4. If a node is red, its children must be black
5. All paths from any node to a descendent leaf must contain the same number of black nodes (black-height, not including the node itself)
   PLUS!
6. Left children may not be red.
An AA-Tree Example

No left red children!! Half of red-black tree rotation cases eliminated!!
Representation of Balancing Info!

- The level of a node (instead of color) is used as balancing info!
- “red” nodes are simply nodes that located at the same level as their parents!
- For the tree on the previous slide:
Redefinition of “Leaf”

- Both the terms leaf and level are redefined:
- A leaf in an AA-tree is a node with no black internal-node as children!
**Redefinition of “Level”**

The level of a node in an AA-tree is:
- leaf nodes are at level 1
- red nodes are at the level of their parent
- black nodes are at one less than the level of their parent as in red-black trees, a black node corresponds to a level change in the corresponding 2-3 tree!
Implications of Ordering Properties

1. Horizontal links are right links! [because only right children may be red!]
2. There may not be double horizontal links! [because there cannot be double red nodes.]
Implications of Ordering Properties

3. Nodes at level 2 or higher must have two children
4. If a node does not have a right horizontal link, its two children are at the same level
5. Any simple path from a black node to a leaf contains one black node on each level.
Example: Insert 45

First, insert as for simple binary search tree
Newly inserted node is red
**Example: Insert 45**

After insert to right of 40:

Problem: **double right horizontal links** starting at 35, need to **split**
Split: Removing Double Reds

Problem: With G inserted, there are two reds in a row

Split is a simple left rotation between X and P

P's level increases in the AA-tree
Example: Insert 45

After split at 35:

Problem: left horizontal link at 50 is introduced, need to skew
Skew: Removing Left Horizontal Link

**Problem:** left horizontal link in AA-tree

Skew is a simple right rotation between X and P

P remains at the same level as X
Example: Insert 45

After skew at 50:
Problem: double right horizontal links starting at 40, need to split
Example: Insert 45

After split at 40:
Problem: left horizontal link at 70 introduced (50 is now on same level as 70), need to skew
Example: Insert 45

After skew at 70:
Problem: double right horizontal links starting at 30, need to split
Example: Insert 45

After split at 30:
Insertion is complete (finally!)
void AATree::Insert( Link &root, Node &add) {
    if (root == NULL)  // have found where to insert y
        root = add;
    else if (add->key < root->key)  // <= if duplicate ok
        Insert(root->left, add);
    else if (add->key > root->key)
        Insert(root->right, add);
    else handle duplicate if not ok

    skew(root);  // do skew and split at each level
    split(root);
}
Skew: Remove Left Horizontal Link

void AATree::skew(Link &root) {
    // root = X
    if (root->left->level == root->level)
        rotate_right(root);
}

AATree skews a node X by removing its left horizontal link.
Split: Remove Double Reds!

```cpp
void AATree::split(Link &root) {  // root = X
    if (root->right->right->right->level == root->level)
        rotate_left(root);
}
```
Skew may cause double reds

First, we apply skew, then we do split if necessary.

After a split, the middle node increases a level, which may create a problem for the original parent

- parent may need to skew and split.
**AA-Tree Removal**

**Rules:**
1. If node to be deleted is a red leaf, e.g., 10, remove leaf, done.
2. If it is parent to a single internal node, e.g., 5, it must be black; replace with its child (must be red) and recolor child black.
3. If it has two internal-node children, swap node to be deleted with its in-order successor.
   - If in-order successor is red (must be a leaf), remove leaf, done.
   - If in-order successor is a single child parent, apply second rule.

In both cases the resulting tree is a legit AA-tree (we haven’t changed the number of black nodes in paths).

3. If in-order successor is a black leaf, or if the node to be deleted itself is a black leaf, things get complicated…
Black Leaf Removal

Follow the path from the removed node to the root
At each node \( p \) with 2 internal-node children do:
• if either of \( p \)'s children is two levels below \( p \)
  • decrease the level of \( p \) by one
• if \( p \)'s right child was a red node, decrease its level also
• \texttt{skew}(p); \texttt{skew}(p\rightarrow\text{right}); \texttt{skew}(p\rightarrow\text{right}\rightarrow\text{right});
• \texttt{split}(p); \texttt{split}(p\rightarrow\text{right});

In the worst case, deleting one leaf node, e.g., 15, could cause six nodes to all be at one level, connected by horizontal right links
• but the worst case can be resolved by 3 calls to \texttt{skew}(), followed by 2 calls to \texttt{split}()!
Black Leaf Removal

- Level 3
  - 30
  - 70
- Level 2
  - 15
  - 50
  - 60
  - 85
  - 20
  - 35
  - 55
  - 65
  - 80
  - 90
- Level 1
  - 5
  - 20
  - 35
  - 55
  - 65
  - 80
  - 90

Remove 5: decrease 15's level
Black Leaf Removal

Level 3
30

Level 2
50 60 85

Level 1
15 20 35 55 65 80 90

Level 2
50 60

Level 1
15 20 35 55 65 80 90

p
decrease level

p
skew(p)

p
skew(p→right)
Black Leaf Removal
Black Leaf Removal
procedure Skew (var t: Tree);
var temp: Tree;
begin
  if t.right.right .level = t.level then
    begin { rotate right }
      temp := t;
      t := t.right;
      temp.right := t.right;
      t.right := temp;
    end;
  end;
end;

procedure Split (var t: Tree);
var temp: Tree;
begin
  if t.right .level = t.level then
    begin { rotate left }
      temp := t;
      t := t.right;
      temp.right := t.left;
      t.left := temp;
      t.level := t.level + 1;
    end;
end;

procedure Insert (var x: data;
var t: Tree; var ok: boolean);
begin
  if t = bottom then begin
    new (t);
    t.key := x;
    t.left := bottom;
    t.right := bottom;
    t.level := 1;
    ok := true;
  end else begin
    if x < t.key then
      Insert (x, t.left, ok)
    else if x > t.key then
      Insert (x, t.right, ok)
    else ok := false;
    Skew (t);
    Split (t);
  end;
end;

procedure Delete (var x: data;
var t: Tree; var ok: boolean);
begin
  ok := false;
  if t <> bottom then begin
    { 1: Search down the tree and }
    { set pointers last and deleted. }
    last := t;
    if x < t.key then
      Delete (x, t.left, ok)
    else begin
      deleted := t;
      Delete (x, t.right, ok);
    end;
    { 2: At the bottom of the tree we }
    { remove the element (if it is present). }
    if (t = last) and (deleted <> bottom)
    and (x = deleted.key) then
      begin
        deleted.key := t.key;
        deleted := bottom;
        t := t.right;
        dispose (last);
        ok := true;
      end;
    { 3: On the way back, we rebalance. }
    else if (t.left .level < t .level - 1)
    or (t.right .level < t .level - 1) then
      begin
        t.level := t.level - 1;
        if t.right .level > t .level then
          t.right .level := t .level;
        Skew (t);
        Skew (t.right);
        Skew (t.right.right);
        Split (t);
        Split (t.right);
      end;
  end;
end;
Balanced BST Summary

AVL Trees: maintain balance factor by rotations

2-3 Trees: maintain perfect trees with variable node sizes using rotations

2-3-4 Trees: simpler operations than 2-3 trees due to pre-splitting and pre-merging nodes, wasteful in memory usage

Red-black Trees: binary representation of 2-3-4 trees, no wasted node space but complicated rules and lots of cases

AA-Trees: simpler operations than red-black trees, binary representation of 2-3 trees
Randomized Binary Search Trees – Treaps
**What is a Treap?**

We consider randomized alternative(s) to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

A **Treap** is a binary tree in which every node has both a **search key and a priority**, where the in-order sequence of search keys is sorted, and each node’s priority is smaller than the priorities of its children. In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities. **Note:** A treap is a BST with heap-ordered priorities (but it is not a heap as it is not required to be a complete binary tree).
We assume from now on that all the keys and priorities are distinct.

Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it’s a heap, the node \( v \) with highest priority must be the root. Since it’s also a binary search tree, any node \( u \) with \( \text{key}(u) < \text{key}(v) \) must be in the left subtree, and any node \( w \) with \( \text{key}(w) > \text{key}(v) \) must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

Another way to describe the structure is that a treap is exactly the binary search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.

A third description interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a \( T \) whose joint lies on the topmost point. The \( T \) splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively [See the picture in the next slide]. This interpretation has some interesting applications in computational geometry [we will skip details of this].
Geometric Representation of a Treap

A treap. Letters are search keys; numbers are priorities. A geometric interpretation of the same treap.
Treap Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

To insert a new node z, we start by using the standard binary search tree insertion algorithm. Priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a rotation at z, a local operation that decreases the depth of z by one and increases its parent’s depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.
Treap Operations

The overall time to insert $z$ is proportional to the depth of $z$ before the rotations—we must walk down the treap to insert $z$, and then walk back up the treap doing rotations. Another way to say this is that the time to insert $z$ is roughly twice the time to perform an unsuccessful search for $\text{key}(z)$.

To delete a node, we just run the insertion algorithm backward in time. Suppose we want to delete node $z$. As long as $z$ is not a leaf, perform a rotation at the child of $z$ with smaller priority. This moves $z$ down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at $z$. When $z$ becomes a leaf, chop it off.
Splitting and Joining a Node

We sometimes want to split a treap $T$ into two treaps $T_<$ and $T_>$ along some pivot key $\pi$, so that all the nodes in $T_<$ have keys less than $\pi$ and all the nodes in $T_>$ have keys bigger than $\pi$. A simple way to do this is to insert a new node $z$ with $\text{key}(z) = \pi$ and $\text{priority}(z) = -\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at $\pi$ is roughly twice the time to (unsuccessfully) search for $\pi$.

Similarly, we may want to join two treaps $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse — create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.
Cost of Operations

**Search:** A successful search for key $k$ takes $O(\text{depth}(v))$ time, where $v$ is the node with $\text{key}(v) = k$. For an unsuccessful search, let $v^-\text{ be the inorder predecessor of } k$ (the node whose key is just barely smaller than $k$), and let $v^+$ be the inorder successor of $k$ (the node whose key is just barely larger than $k$). Since the last node examined by the binary search is either $v^-$ or $v^+$, the time for an unsuccessful search is either $O(\text{depth}(v^+))$ or $O(\text{depth}(v^-))$.

**Insert/Delete:** Inserting a new node with key $k$ takes either $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ time, where $v^+$ and $v^-$ are the predecessor and successor of the new node. Deletion is just insertion in reverse.

**Split/Join:** Splitting a treap at pivot value $k$ takes either $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ time, since it costs the same as inserting a new dummy root with search key $k$ and priority $-\infty$. Merging is just splitting in reverse.

**Note:**
- In the worst case, the depth of an $n$-node treap is $\Theta(n)$, so each of these operations has a worst-case running time of $\Theta(n)$.
- There are other variations like Random Priorities and Skip Lists [we omit]
References

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html
https://www.geeksforgeeks.org/2-3-trees-search-and-insert/

Write Something; Pradip Srimani; Monai, Rono