Chapter 10  
Efficient Binary Search Trees

- Optimal Binary Search Trees
- AVL Trees
- Red-Black Trees
- Splay Trees

A binary search tree (BST) is a binary tree, that may be empty or satisfies the following properties:
- Every element has a unique key.
- The keys in a nonempty left(right) sub-tree must be smaller(larger) than the key in the root of the sub-tree.
- The left and right sub-trees are also binary search trees.

Example: Two binary search trees

Internal node

External node
Operations in Binary Search Trees

- **Binary Search Trees (BST)**
  - Searching $\Rightarrow O(h)$, $h$ is the height of BST
  - Insertion $\Rightarrow O(h)$
  - Deletion $\Rightarrow O(h)$
  - Can be done quickly by both key value and rank

Two Algorithms of Searching a BST

```c
element * Recursive_search (tree_pointer root, int key)
{ /* return a pointer to the node that contains key. If there is no such node, return NULL */
  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data) return Recursive_search(root->leftChild,key);
  return Recursive_search(root->rightChild,key);
}
```

```c
element * Iterative_search (tree_pointer tree, int key)
{    while (tree)
{ if (key == tree->data) return tree;
  if (key < tree->data) tree = tree->leftChild;
  else tree = tree->rightChild;
}
return NULL;
}
```

$O(h)$, $h$ is the height of BST.
Inserting into a BST

- Step 1: Check if the inserting key is different from those of existing elements
- Step 2: Run \texttt{insert\_node} function

Algorithm of Inserting into a BST

```c
void insert_node (treePointer *node, int num) {
    treePointer ptr, temp = modifiedSearch(*node, num);
    if (temp || !(*node)) {
        MALLOC(ptr,sizeof(*ptr));
        ptr->data = num;
        ptr->leftChild = ptr->rightChild = NULL;
        if (*node) {
            if (num<temp->data) temp->left_child=ptr;
            else temp->right_child = ptr;
        }
        else *node = ptr;
    }
}
```
### Deletion from a BST

- Delete a non-leaf node with two children
  - Replace the largest element in its left sub-tree
  - Or Replace the smallest element in its right sub-tree
  - Recursively to the leaf \( O(h) \)

**Delete 60**

![BST Diagram](image)

- **Delete 60**

  - (a) tree before deletion of 60
  - (b) tree after deletion of 60

### Height of a BST

- The Height of the binary search tree is \( O(\log n) \), on the average. If you insert in a sorted order using the insertion algorithm, you'll obtain a degenerate BST.
  - **Best case** \( O(h) = O(\log n) \)
  - **Worst case** (skewed) \( O(h) = O(n) \)
Balanced Binary Search Trees

- There are binary search trees that guarantees balance, such as **AVL Trees**, **2-3 Trees**, and **Red-Black Trees**.
  - A Balanced Search Tree has a worst case height of $O(\log_2 n)$
- Balance factor of a node:
  \[
  \text{(height of left subtree)} - \text{(height of right subtree)} = -1, 0, \text{ or } 1.
  \]

Balanced BST

- Example 1: Rotate 20

- Example 2:
  - Nodes v and w decrease in height
  - Nodes y and z increase in height
  - Node x remains at same height
AVL Trees

- AVL (Adel’son-Vel’skii and Landis) trees are balanced.
- 
  **Height-Balanced Property:** An AVL Tree is a *binary search tree* such that for every internal node v of T, the *heights of the children of v can differ by at most 1*. That is, each node has a balance factor of -1, 0, or 1.

```
  44
 /   \
 2    78
 \
 17   88
   /   /
  32   50
    /   /
   148  62
```

Height of an AVL Tree

- The **height** of an AVL tree storing n keys is $O(\log n)$.
- Balance is maintained by the **insertion** and **deletion** algorithms. Both take $O(\log n)$ time. For example, if an insertion causes un-balance, then some rotation is performed.
Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example: `insertItem(54)`

Example:

Before insertion:

```
17
32
44
50
62
78
88
```

After insertion:

```
17
32
44
50
62
78
88
```

Insertion Example (cont’d)

Unbalanced, 4-2=2 > 1

Unbalanced

Balanced
Trinode Restructuring

- let \((a, b, c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

Case 1: single rotation
- (a left rotation about \(a\))

Case 2: double rotation
- (a right rotation about \(c\), then a left rotation about \(a\))

Restructuring of Single Rotations

- single rotation
- single rotation
Restructuring of Double Rotations

Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example: removal of 32

Before deletion of 32

After deletion
Rebalancing after a Removal

- Let $z$ be the first unbalanced node encountered while traveling up the tree from $w$. Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.
- We perform $\text{restructure}(x)$ to restore balance at $z$.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.

Comparing Some Structures

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential list</th>
<th>Linked list</th>
<th>AVL tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for $x$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Search for $k$th item</td>
<td>$O(1)$</td>
<td>$O(k)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delete $x$</td>
<td>$O(n)$</td>
<td>$O(1)^1$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delete $k$th item</td>
<td>$O(n - k)$</td>
<td>$O(k)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Insert $x$</td>
<td>$O(n)$</td>
<td>$O(1)^2$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Output in order</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

1. Doubly linked list and position of $x$ known.
2. If position for insertion is known.
Red-Black Trees

A red-black tree can also be defined as a **binary search tree** that satisfies the following properties:

- **Root Property**: the root is black
- **External Property**: every leaf is black
- **Internal Property**: the children of a red node are black
- **Depth Property**: all the leaves have the same black depth

Height of a Red-Black Tree

- **Theorem**: A red-black tree storing \( n \) items has **height** \( O(\log n) \)
  - **Proof**: The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is \( O(\log n) \)
  - The search algorithm for a binary search tree is the same as that for a binary search tree
  - By the above theorem, **searching** in a red-black tree takes \( O(\log n) \) time
Insertion in Red-Black Tree

To perform operation of inserting an item, we execute the insertion algorithm for binary search trees and color red the newly inserted node $z$ unless it is the root.

- We preserve the root, external, and depth properties.
- If the parent $v$ of $z$ is black, we also preserve the internal property and we are done.
- Else ($v$ is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree.

Example: insert(4) causes a double red:

```
6    3    8
3
```

Remedying a Double Red

- Consider a double red with child $z$ and parent $v$, and let $w$ be the sibling of $v$.

Case 1: $w$ is black

- The double red is an incorrect replacement of a 4-node.
- Restructuring: we change the 4-node replacement.

Case 2: $w$ is red

- The double red corresponds to an overflow.
- Recoloring: we perform the equivalent of a split.
Restructuring

A restructuring remedies a child-parent double red when the parent red node has a black sibling

- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved

Restructuring (cont.)

- There are four restructuring configurations depending on whether the double red nodes are left or right children
Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent $v$ and its sibling $w$ become black and the grandparent $u$ becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent $u$

![Recoloring Diagram](image)

Analysis of Insertion

Algorithm `insertItem(k, o)`

1. We search for key $k$ to locate the insertion node $z$
2. We add the new item $(k, o)$ at node $z$ and color $z$ red
3. while $doubleRed(z)$
   - if isBlack(sibling(parent(z)))
     
     ```
     z ← recolor(z)
     ```
   - else { sibling(parent(z)) is red }
     
     ```
     z ← recolor(z)
     ```

- Recall that a red-black tree has $O(\log n)$ height
  - Step 1: takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2: takes $O(1)$ time

- Step 3: takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time

- Thus, an insertion in a red-black tree takes $O(\log n)$ time
Insertion 35 into a RB Tree

- Recolor case (parent has sibling)

- If parent has no sibling: swap parent-grandparent colors, and then rotate right around grandparent

Insertion 35 into a RB Tree

- Rotation doesn't work in right-left case
Deletion in Red-Black Tree

- To perform operation remove($k$), we first execute the deletion algorithm for binary search trees.
- Let $v$ be the internal node removed, $w$ the external node removed, and $r$ the sibling of $w$.
  - If either $v$ of $r$ was red, we color $r$ black and we are done.
  - Else ($v$ and $r$ were both black) we color $r$ double black, which is a violation of the internal property requiring a reorganization of the tree.
- Example: **remove(8) causes a double black:**
Remedying a Double Black

The algorithm for remedying a double black node \( w \) with sibling \( y \) considers three cases

Case 1: \( y \) is black and has a red child
- We perform a restructuring, equivalent to a transfer, and we are done

```
         y
        / \
       o  w
```

\[ \rightarrow \]

```
         y
        / \
       o  w
```

Remedying a Double Black (cond’t)

Case 2: \( y \) is black and its children are both black
- We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation

```
         y
        / \
       o  w
```

\[ \rightarrow \]

```
         y
        / \
       o  w
```
### Remedying a Double Black (cond’t)

**Case 3:** $y$ is red

- We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies.

```
  y   o
3   3
```

```
  y   o
3   4
```

- Deletion in a red-black tree takes $O(\log n)$ time.

---

### Splay Trees

- Binary search trees.
- **Search, insert, delete, and split** have amortized complexity $O(\log n)$ & actual complexity $O(n)$.
- Actual and amortized complexity of **join** is $O(1)$.
- Priority queue and double-ended priority queue versions outperform heaps, etc. over a sequence of operations.
- Two varieties.
  - Bottom up.
  - Top down.
Bottom-Up Splay Trees

- Search, insert, delete, and join are done as in an unbalanced binary search tree.
- Search, insert, and delete are followed by a splay operation that begins at a splay node. When the splay operation completes, the splay node has become the tree root.
- Join requires no splay (or, a null splay is done).
- For the split operation, the splay is done in the middle (rather than end) of the operation.

Splay Node – search(k)

- If there is a pair whose key is k, the node containing this pair is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.
Splay Node – insert(newPair)

- If there is already a pair whose key is `newPair.key`, the node containing this pair is the splay node.
- Otherwise, the newly inserted node is the splay node.

```
20
  10
  6
  2

28
  15
  8

30
  25
```

Splay Node – delete(k)

- If there is a pair whose key is `k`, the parent of the node that is physically deleted from the tree is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.

```
20
  10
  6
  2

28
  15
  8

30
  25
```
**Splay Node – split(k)**

- Use the unbalanced binary search tree insert algorithm to insert a new pair whose key is k.
- The splay node is as for the splay tree insert algorithm.
- Following the splay, the left subtree of the root is S, and the right subtree is B.
- m is set to **null** if it is the newly inserted pair.

```
   m
  /   \
 S     B
```

**Splay**

- Let q be the splay node and q is moved up the tree using a series of splay steps.
- In a splay step, the node q moves up the tree by 0, 1, or 2 levels. Every splay step, except possibly the last one, moves q two levels up.

**Splay step:**

- **If** q = null or q is the root, do nothing (splay is over).
- **If** q is at level 2, do a one-level move and terminate the splay operation. q right child of p is symmetric.

```
      q
   /     \                 RR rotation
 p      c
/     \                      \
 a      b
```

```
      q
   /     \                  \
 a     b
```

```
      q
   /     \                    \
 b     c
```

```
      q
   /     \                  \
 a     b
```

```
      q
   /     \                    \
 b     c
```
Splay Step

- If \( q \) is at a level > 2, do a two-level move and continue the splay operation.

![Diagram showing splay step with RR rotation]

\( q \) right child of right child of \( gp \) is symmetric.

2-Level Move (case 2)

- \( q \) left child of right child of \( gp \) is symmetric.

![Diagram showing 2-level move with RR rotation]
Per Operation Actual Complexity

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, ..., in this order.

  ![Diagram 1](image1)

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, ..., in this order.

  ![Diagram 2](image2)

- Worst-case height = n.
- Actual complexity of search, insert, delete, and split is $O(n)$.

---

Top-Down Splay Trees

- On the way down the tree, split the tree into the binary search trees $S$ (small elements) and $B$ (big elements).
  
  - Similar to split operation in an unbalanced binary search tree.
  
  - However, a rotation is done whenever an LL or RR move is made.
  
  - Move down 2 levels at a time, except (possibly) in the end when a one level move is made.
  
  - When the splay node is reached, $S$, $B$, and the subtree rooted at the splay node are combined into a single binary search tree.
Split A Binary Search Tree

m is the splay node

...
Split A Binary Search Tree

C-C Tsai P.51

Split A Binary Search Tree

C-C Tsai P.52
Split A Binary Search Tree

A
  /  
 a   C
   / 
  c   D
    / 
   d   E

F
  /  
 m   e
   / 
 f   g

Split A Binary Search Tree

A
  /  
 a   C
   / 
  c   D
    / 
   d   E

F
  /  
 m   e
   / 
 f   g
Let $m$ be the splay node.
- RL move from A to C.
- RR move from C to E.
- L move from E to m.
RR Move

Rotation performed.
Outcome is different from split.

L Move
Bottom Up vs Top Down

- Top down splay trees are faster than bottom up splay trees.