Balanced Trees

- BST algorithms can degenerate to worst case performance, which is bad because the worst case is likely to occur in practice, with ordered files, for example

- We will like to keep our trees perfectly balanced (ideally speaking)
  - Corresponds to binary search
  - Insertion and deletion of records is expensive

- In a non-ideal situation, we can allow the binary tree to grow to twice the height of the perfect tree \((2 \lg n)\) and periodically balance it
  - Provides protection against bad worst case performance
  - Improves performance for random keys but does not provide guarantees against quadratic performance in dynamic symbol table
  - Partition to put the median node at the root and recursively do the same for subtrees
  - Algorithm to balance a BST in linear time

```cpp
int tmp = left_child ? left_child->count() : 0;
if (tmp > k)
{
    left_child->partition_rotate ( k );
    *this = rotate_right();
}
if (tmp < k)
{
    right_child->partition_rotate ( k - tmp - 1 );
    *this = rotate_left();
}
return ( *this ); // Return if tmp == k (kth smallest key)
```

```cpp
tree tree::rotate_left()
{
    tree * tmp = right_child;
    right_child = tmp->left_child;
    tmp->left_child = this;
    this = tmp;
    return ( *this );
}
```

```cpp
tree tree::rotate_right()
{
    tree * tmp = left_child;
    left_child = tmp->right_child;
    tmp->right_child = this;
    this = tmp;
    return ( *this );
}
```

```cpp
int temp = left_child ? left_child->count() : 0;
if ( temp > k )
{
    left_child->partition_rotate ( k );
    *this = rotate_right();
}
if ( temp < k )
{
    right_child->partition_rotate ( k - temp - 1 );
    *this = rotate_left();
}
return ( *this ); // Return if temp == k (kth smallest key)
```
Rebalancing improves performance for random keys but does not provide guarantees against quadratic worst-case performance, for *dynamic* symbol tables

* Preferable to have algorithms that do incremental balancing rather than stop the insertion to do complete rebalancing

### Randomized algorithm

- Introduce random decision making into the algorithm itself, such as median of three partitioning in quicksort
- Reduces the chance of worst case scenario, no matter what the input
- Equivalent in the search is *skip list*

### Amortized algorithm

- Do extra work at some point to save time later

### Optimized algorithm

- Provides performance guarantee for every operation
- Require to maintain some structural information in the trees

#### Randomized BSTs

- Items inserted randomly into the *BST*
  - Each item is equally likely to be in the root node of the tree
  - Possible to introduce randomness into the algorithm so that the above property holds without any assumption about the order of items
- Insert a new random node into the tree at the root
  - The probability of this node being at the root is $\frac{1}{1+N}$ when the tree has $N$ nodes
  - Perform root insertion with this probability

```cpp
tree tree::insert_random ( item& i )
{
    if ( rand() < ( 1 / ( 1+count() ) ) )
        insert_at_root ( i );
    else
        if ( i.key() < info.key() )
            left_child->insert_random ( i );
        else
            right_child->insert_random ( i );
}
```
**Property 1** Building a randomized BST is equivalent to building a standard BST from a random initial permutation of the keys. We use about $2N \ln N$ comparisons to construct a randomized BST with $N$ items (no matter in what order the items are presented for insertion), and about $2 \ln N$ comparisons for searches in such a tree.

- Each element is equally likely at the root of the tree
- The property holds for both subtrees as well

- Average case for insertion into randomized and standard BST is the same (except for random number computation)
  
  - The assumption of items arriving at random in standard BST is not required any more

**Property 2** The probability that the construction cost of a randomized BST is more than a factor of $\alpha$ times the average is less than $e^{-\alpha}$.

**Property 3** Making a tree with an arbitrary sequence of randomized insert, remove, and join operations is equivalent to building a standard BST from a random permutation of the keys in the tree.

**Top-down 2-3-4 trees**

- Allow 3-nodes and 4-nodes that can hold 2 or 3 keys, respectively, in addition to the regular binary nodes that hold only one key

**Definition 1** A **2-3-4 search tree** is a tree that either is empty or comprises three types of nodes:

- **2-nodes**, with one key, a left link to a tree with smaller keys, and a right link to a tree with larger keys;
- **3-nodes**, with two keys, a left link to a tree with smaller keys, a middle link to a tree with key values between the node’s keys, and a right link to a tree with larger keys;
- **4-nodes**, with three keys and four links to trees with key values defined by the ranges subtended by the node’s keys.

**Definition 2** A **balanced 2-3-4 search tree** is a 2-3-4 search tree with all links to empty trees at the same distance from the root.

**Red-Black Trees**

Properties of red-black tree

- Binary search tree with one extra bit of storage per node – its color
- No path is more than twice as long as any other
- Tree is approximately balanced
- Fields in a node – color, key, left, right, and parent
- A binary search tree is a red-black tree if the following properties are satisfied
  - Every node is either red or black
  - Every leaf (nil) is black
  - If a node is red then both its children are black
  - Every simple path from a node to a descendant leaf contains the same number of black nodes
• black-height of a node – bh(x) – Number of black nodes on any path from, but not including, a node x to a leaf

Lemma. A red-black tree with n internal nodes has height at most 2 \lg(n + 1)

Rotations
• Insert and delete may result in violation of the red-black properties
• Change the color and pointer structure to restore the properties
• Change pointer structure through rotation
• Left rotation possible only if the right child of the node is non-nil

```c
left_rotate (T, x)
    y ← right[x]
    right[x] ← left[y]
    if left[y] ≠ nil then
        parent[left[y]] ← x
    parent[y] ← parent[x]
    if parent[x] = nil then
        root[T] ← y
    else
        if x = left[parent[x]] then
            left[parent[x]] ← y
        else
            right[parent[x]] ← y
    left[y] ← x
    parent[x] ← y
```

Insertion
• Accomplished in O(lg n) time
• Insert x into tree T as if it were ordinary binary search tree
• Recolor nodes and perform rotations to preserve the red-black property

```c
red_black_insert (T, x)
    tree_insert (T, x)
    color[x] ← red
    while x ≠ root[T] and color[parent[x]] = red do
        if parent[x] = left[parent[parent[x]]] then
            y ← right[parent[parent[x]]]
        if color[y] = red then
            color[parent[x]] ← black
            color[y] ← black
            color[parent[parent[x]]] ← red
            x ← parent[parent[x]]
        else
            if x = right[parent[x]] then
                x ← parent[x]
```

---

**Note:** The code snippets represent operations in a red-black tree, but the specific variables and structure may vary depending on the implementation details not provided here. The focus is on the operations and their descriptions rather than their implementation.
left_rotate (T,x)
color[p[x]] ← black
color[parent[parent[x]]] ← red
right_rotate (T,parent[parent[x]])
else
  y ← left[parent[parent[x]]]
  if color[y] = red then
    color[parent[x]] ← black
    color[y] ← black
    color[parent[parent[x]]] ← red
    x ← parent[parent[x]]
  else
    if x = left[parent[x]] then
      x ← parent[x]
      right_rotate (T,x)
      color[p[x]] ← black
      color[parent[parent[x]]] ← red
      left_rotate (T,parent[parent[x]])
  color[root[T]] ← black