SEARCH TREES

Binary Search Trees

BINARY SEARCH TREES
Keys are assumed to come from a total order.

Items are stored in order by their keys.

This allows us to support nearest neighbor queries:

- Item with largest key less than or equal to k
- Item with smallest key greater than or equal to k
Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
+ similar to the high-low children’s game
+ at each step, the number of candidate items is halved
+ terminates after \(O(\log n)\) steps

Example: find(7)
A search table is an ordered map implemented by means of a sorted sequence
+ We store the items in an array-based sequence, sorted by key
+ We use an external comparator for the keys

Performance:
+ Searches take $O(\log n)$ time, using binary search
+ Inserting a new item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to make room for the new item
+ Removing an item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to compact the items after the removal

The lookup table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
We define **binary search tree** as a **proper binary tree** storing keys (or key-value entries) at its **internal nodes** and satisfying the following property:

Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $key(u) \leq key(v) \leq key(w)$

**External nodes** do not store items

- We use the leaves as “placeholders” (sentinels)
- Represented as **null** references in practice,

**An inorder traversal** of a binary search tree visits the keys in increasing order
To search for a key \( k \), we trace a downward path starting at the root.

The next node visited depends on the comparison of \( k \) with the key of the current node.

If we reach a leaf, the key is not found.

Example: get(4):

Call TreeSearch(4, root)

The algorithms for nearest neighbor queries are similar.
A successful search for key 65 in a binary search tree;

An unsuccessful search for key 68 that terminates at the leaf to the left of the key 76.
Algorithm TreeSearch is recursive and executes a constant number of primitive operations for each recursive call.

We'll talk about various strategies to maintain an upper bound of $O(\log n)$ on the height soon.
To perform operation `put(k, o)`, we search for key k (using TreeSearch) insertions, which always occur at a leaf).

Assume a proper binary tree supports the following update operation
+ `expandExternal(p, e)`: Stores entry e at the external position p, and expands p to be internal, having two new leaves as children.

**Algorithm** `TreeInsert(k, v)`: 

**Input**: A search key k to be associated with value v

```
p = TreeSearch(root(), k)
if k == key(p) then
    Change p’s value to (v)
else
    expandExternal(p, (k, v))
```

executes in time $O(h)$
EXAMPLE OF INSERT

Insertion of an entry with key 68 into the search tree

Finding the position to insert

the resulting tree
Deleting an entry from a binary search tree might happen anywhere in the tree.

To perform operation remove(k), we search for key k by calling TreeSearch(root( ), k) to find the position p storing an entry with key equal to k (if any).

- If search returns an external node, then there is no entry to remove.
- Otherwise,
  - at most one of the children of position p is internal,
  - Or position p has two internal children.
Deletion when at most one of the children of position $p$ is internal.

- Let position $r$ be a child of $p$ that is internal (or an arbitrary child, if both are leaves).
- Remove $p$ and the leaf that is $r$’s sibling, while promoting $r$ upward to take the place of $p$.

executes in time $O(h)$
Deletion position $p$ has two internal children

- Locate position $r$ containing the entry having the greatest key that is strictly less than that of position $p$ (the rightmost internal position of the left subtree of position $p$)
- Use $r$'s entry as a replacement for the one being deleted at position $p$.
- Delete the node at position $r$ from the tree.

executes in time $O(h)$
/** An implementation of a sorted map using a binary search tree. */

```
public class TreeMap<K,V> extends AbstractSortedMap<K,V> {
    // To represent the underlying tree structure, we use a specialized subclass of the
    // LinkedBinaryTree class that we name BalanceableBinaryTree (see Section 11.2).
    protected BalanceableBinaryTree<K,V> tree = new BalanceableBinaryTree<>();

    /** Constructs an empty map using the natural ordering of keys. */
    public TreeMap() {
        super();            // the AbstractSortedMap constructor
        tree.addRoot(null);  // create a sentinel leaf as root
    }

    /** Constructs an empty map using the given comparator to order keys. */
    public TreeMap(Comparator<K> comp) {
        super(comp);        // the AbstractSortedMap constructor
        tree.addRoot(null);  // create a sentinel leaf as root
    }

    /** Returns the number of entries in the map. */
    public int size() {
        return (tree.size() - 1) / 2;  // only internal nodes have entries
    }

    /** Utility used when inserting a new entry at a leaf of the tree */
    private void expandExternal(Position<Entry<K,V>> p, Entry<K,V> entry) {
        tree.set(p, entry);          // store new entry at p
        tree.addLeft(p, null);       // add new sentinel leaves as children
        tree.addRight(p, null);
    }
```

16

© 2014 Goodrich, Tamassia, Goldwasser
Omitted from this code fragment, but included in the online version of the code, are a series of protected methods that provide notational shorthands to wrap operations on the underlying linked binary tree. For example, we support the protected syntax `root()` as shorthand for `tree.root()` with the following utility:

```java
protected Position<Entry<K,V>> root() { return tree.root(); }
```

`/** Returns the position in p's subtree having given key (or else the terminal leaf). */`

```java
private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
    if (isExternal(p))
        return p; // key not found; return the final leaf
    int comp = compare(key, p.getElement());
    if (comp == 0)
        return p; // key found; return its position
    else if (comp < 0)
        return treeSearch(left(p), key); // search left subtree
    else
        return treeSearch(right(p), key); // search right subtree
}
```
```java
/** Returns the value associated with the specified key (or else null). */
public V get(K key) throws IllegalArgumentException {
    checkKey(key); // may throw IllegalArgumentException
    Position<Entry<K,V>> p = treeSearch(root(), key);
    rebalanceAccess(p); // hook for balanced tree subclasses
    if (isExternal(p)) return null; // unsuccessful search
    return p.getElement().getValue(); // match found
}

/** Associates the given value with the given key, returning any overridden value. */
public V put(K key, V value) throws IllegalArgumentException {
    checkKey(key); // may throw IllegalArgumentException
    Entry<K,V> newEntry = new MapEntry<>(key, value);
    Position<Entry<K,V>> p = treeSearch(root(), key);
    if (isExternal(p)) { // key is new
        expandExternal(p, newEntry);
        rebalanceInsert(p); // hook for balanced tree subclasses
        return null;
    } else { // replacing existing key
        V old = p getElement().getValue();
        set(p, newEntry);
        rebalanceAccess(p); // hook for balanced tree subclasses
        return old;
    }
}
```
** Removes the entry having key k (if any) and returns its associated value. */

```java
public V remove(K key) throws IllegalArgumentException {
    checkKey(key); // may throw IllegalArgumentException
    Position<Entry<K,V>> p = treeSearch(root(), key);
    if (isExternal(p)) { // key not found
        rebalanceAccess(p); // hook for balanced tree subclasses
        return null;
    } else {
        V old = p.getElement().getValue();
        if (isInternal(left(p)) && isInternal(right(p))) { // both children are internal
            Position<Entry<K,V>> replacement = treeMax(left(p));
            set(p, replacement.getElement());
            p = replacement;
        } // now p has at most one child that is an internal node
        Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
        Position<Entry<K,V>> sib = sibling(leaf);
        remove(leaf);
        remove(p); // sib is promoted in p’s place
        rebalanceDelete(sib); // hook for balanced tree subclasses
        return old;
    }
}
```
JAVA IMPLEMENTATION 5

```java
92     /** Returns the position with the maximum key in subtree rooted at Position p. */
93   protected Position<Entry<Key,Value>> treeMax(Position<Entry<Key,Value>> p) {
94       Position<Entry<Key,Value>> walk = p;
95       while (isInternal(walk))
96           walk = right(walk);
97       return parent(walk); // we want the parent of the leaf
98   }
99     /** Returns the entry having the greatest key (or null if map is empty). */
100    public Entry<Key,Value> lastEntry() {
101        if (isEmpty()) return null;
102        return treeMax(root().getElement());
103    }
104    /** Returns the entry with greatest key less than or equal to given key (if any). */
105    public Entry<Key,Value> floorEntry(Key key) throws IllegalArgumentException {
106        checkKey(key); // may throw IllegalArgumentException
107        Position<Entry<Key,Value>> p = treeSearch(root(), key);
108        if (isInternal(p)) return p.getElement(); // exact match
109        while (!isRoot(p)) {
110            if (p == right(parent(p)))
111                return parent(p).getElement(); // parent has next lesser key
112            else
113                p = parent(p);
114        }
115        return null; // no such floor exists
116    }
```
```java
/** Returns the entry with greatest key strictly less than given key (if any). */
public Entry<K,V> lowerEntry(K key) throws IllegalArgumentException {
    checkKey(key); // may throw IllegalArgumentException
    Position<Entry<K,V>> p = treeSearch(root(), key);
    if (isInternal(p) && isInternal(left(p)))
        return treeMax(left(p)).getElement(); // this is the predecessor to p
    // otherwise, we had failed search, or match with no left child
    while (!isRoot(p)) {
        if (p == right(parent(p)))
            return parent(p).getElement(); // parent has next lesser key
        else
            p = parent(p);
    }
    return null; // no such lesser key exists
```
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
    for (Position<Entry<K,V>> p : tree.inorder())
        if (isInternal(p)) buffer.add(p getElement());
    return buffer;
}

/** Returns an iterable of entries with keys in range [fromKey, toKey]. */
public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
    if (compare(fromKey, toKey) < 0)                 // ensure that fromKey < toKey
        subMapRecursively(fromKey, toKey, root(), buffer);
    return buffer;
}

private void subMapRecursively(K fromKey, K toKey, Position<Entry<K,V>> p, 
                                ArrayList<Entry<K,V>> buffer) {
    if (isInternal(p))
        if (compare(p getElement(), fromKey) < 0)
            // p's key is less than fromKey, so any relevant entries are to the right
            subMapRecursively(fromKey, toKey, right(p), buffer);
        else {
            subMapRecursively(fromKey, toKey, left(p), buffer);    // first consider left subtree
            if (compare(p getElement(), toKey) < 0) {
                buffer.add(p getElement());                     // so add it to buffer, and consider
                subMapRecursively(fromKey, toKey, right(p), buffer); // right subtree as well
            }
        }
}
### Performance of a Binary Search Tree

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get, put, remove</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>firstEntry, lastEntry</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>ceilingEntry, floorEntry, lowerEntry, higherEntry</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>subMap</td>
<td>$O(s + h)$</td>
</tr>
<tr>
<td>entrySet, keySet, values</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* subMap implementation can be shown to run in $O(s + h)$ worst-case bound for a call that reports $s$ results.
Augmenting a standard binary search tree with occasional operations to reshape the tree and reduce its height

- Examples: AVL trees, splay trees, and red-black trees

The primary operation to rebalance a binary search tree is known as a **rotation**

- Allows the shape of a tree to be modified while maintaining the search-tree property.

"rotate" a child to be above its parent

$O(1)$ time with a linked binary tree representation
ALGORITHM FOR ROTATION

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
ALGORITHM FOR ROTATION (CONT.)

1. Remember value of root.left (temp = root.left)
1. Remember value of root.left (temp = root.left)

2. Set root.left to value of temp.right
1. Remember value of root.left (temp = root.left)
2. Set root.left to value of temp.right
3. Set temp.right to root
1. Remember value of root.left (temp = root.left)
2. Set root.left to value of temp.right
3. Set temp.right to root
4. Set root to temp
ALGORITHM FOR ROTATION (CONT.)

root

Node
left = 
right = 
data = 10

Node
left = 
right = 
data = 5

Node
left = 
right = null

data = 7

Node
left = 
right = null

data = 15

Node
left = 
right = null

data = 20

Node
left = 
right = null

data = 40

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
**Trinode Restructuring** is a compound rotation operations with the goal to restructure the subtree rooted at the *grandparent* \( z \) in order to reduce the overall path length to current node \( x \) and its subtrees.

**Algorithm** `restructure(x)`:

**Input:** A position \( x \) of a binary search tree \( T \) that has both a parent \( y \) and a grandparent \( z \)

**Output:** Tree \( T \) after a trinode restructuring (which corresponds to a single or double rotation) involving positions \( x, y, \) and \( z \)

1. Let \((a, b, c)\) be a left-to-right (inorder) listing of the positions \( x, y, \) and \( z \), and let \((T_1, T_2, T_3, T_4)\) be a left-to-right (inorder) listing of the four subtrees of \( x, y, \) and \( z \) not rooted at \( x, y, \) or \( z \).
2. Replace the subtree rooted at \( z \) with a new subtree rooted at \( b \).
3. Let \( a \) be the left child of \( b \) and let \( T_1 \) and \( T_2 \) be the left and right subtrees of \( a \), respectively.
4. Let \( c \) be the right child of \( b \) and let \( T_3 \) and \( T_4 \) be the left and right subtrees of \( c \), respectively.
FOUR KINDS OF CRITICALLY UNBALANCED TREES

- **Left-Left** (parent balance is -2, left child balance is -1)
  - Rotate right around parent

- **Left-Right** (parent balance -2, left child balance +1)
  - Rotate left around child
  - Rotate right around parent

- **Right-Right** (parent balance +2, right child balance +1)
  - Rotate left around parent

- **Right-Left** (parent balance +2, right child balance -1)
  - Rotate right around child
  - Rotate left around parent
EXAMPLE OF A TRINODE RESTRUCTURING OPERATION 1

(a) Single rotation

(b) Single rotation
Each light purple triangle represents a tree of height $k$. Each leaf has a value of $2^j$ for some $j$. Balancing a left-left tree requires rotating the right child of the left subtree to the left.
The dark purple trapezoid represents an insertion into this tree, making its height $k + 1$. 

The formula $h_R - h_L$ is used to calculate the balance of each node.

The heights of the left and right subtrees are unimportant; only the relative difference matters when balancing.
When the root and left subtree are both left-heavy, the tree is called a Left-Left tree.
A Left-Left tree can be balanced by a rotation right.
BALANCING A LEFT-LEFT TREE (CONT.)

![Diagram of a balanced left-left tree]

- Root node with value 25
- Left child node with value a
- Right child node with value 50
- Right child's right child node with value c

The tree is balanced with a rotation at the root node.
EXAMPLE OF TRINODE RESTRUCTURING OPERATION 2

(c)

(double rotation)

(d)

(double rotation)
Balancing a Left-Right Tree

\[ k - (k + 2) \]

\[ (k + 1) - k \]

\[ +1 \]

\[ -2 \]
A Left-Right tree cannot be balanced by a simple rotation right.
Subtree \( b \) needs to be expanded into its subtrees \( b_L \) and \( b_R \).
40 is left-heavy. The left subtree can now be rotated left.
The overall tree is now Left-Left and a rotation right will balance it.
BALANCING A LEFT-RIGHT TREE (CONT.)

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
In the previous example, an item was inserted in \( b_L \). We now show the steps if an item was inserted into \( b_R \) instead.
Rotate the left subtree left

Balancing a Left-Right Tree (Cont.)

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
Rotate the tree right
BALANCING A LEFT-RIGHT TREE (CONT.)

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
/** A specialized version of LinkedBinaryTree with support for balancing. */
protected static class BalanceableBinaryTree<K,V> extends LinkedBinaryTree<Entry<K,V>> {
  //---------- nested BSTNode class ----------
  // this extends the inherited LinkedBinaryTree.Node class
  protected static class BSTNode<E> extends Node<E> {
    int aux=0;
    BSTNode(E e, Node<E> parent, Node<E> leftChild, Node<E> rightChild) {
      super(e, parent, leftChild, rightChild);
    }
    public int getAux() { return aux; }
    public void setAux(int value) { aux = value; }
  } //----- end of nested BSTNode class -----

  // positional-based methods related to aux field
  public int getAux(Position<Entry<K,V>> p) {
    return ((BSTNode<Entry<K,V>>) p).getAux();
  }
  public void setAux(Position<Entry<K,V>> p, int value) {
    ((BSTNode<Entry<K,V>>) p).setAux(value);
  }

  // Override node factory function to produce a BSTNode (rather than a Node)
  protected
  Node<Entry<K,V>> createNode(Entry<K,V> e, Node<Entry<K,V>> parent,
                             Node<Entry<K,V>> left, Node<Entry<K,V>> right) {
    return new BSTNode<>(e, parent, left, right);
  }
}
/** Relinks a parent node with its oriented child node. */
private void relink(Node<Entry<K,V>> parent, Node<Entry<K,V>> child,
boolean makeLeftChild) {
    child.setParent(parent);
    if (makeLeftChild)
        parent.setLeft(child);
    else
        parent.setRight(child);
}
/** Rotates Position p above its parent. */
public void rotate(Position<Entry<K, V>> p) {
    Node<Entry<K, V>> x = validate(p);
    Node<Entry<K, V>> y = x.getParent(); // we assume this exists
    Node<Entry<K, V>> z = y.getParent();  // grandparent (possibly null)
    if (z == null) {
        root = x; // x becomes root of the tree
        x.setParent(null);
    } else
        relink(z, x, y == z.getLeft()); // x becomes direct child of z
    // now rotate x and y, including transfer of middle subtree
    if (x == y.getLeft()) {
        relink(y, x.getRight(), true); // x's right child becomes y's left
        relink(x, y, false);          // y becomes x's right child
    } else {
        relink(y, x.getLeft(), false); // x's left child becomes y's right
        relink(x, y, true);           // y becomes left child of x
    }
}
```java
/** Performs a trinode restructuring of Position x with its parent/grandparent. */
public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
    Position<Entry<K,V>> y = parent(x);
    Position<Entry<K,V>> z = parent(y);
    if ((x == right(y)) == (y == right(z))) { // matching alignments
        rotate(y);
        return y;
    } else { // opposite alignments
        rotate(x);
        rotate(x);
        return x; // x is new subtree root
    }
}
```