CS200: Balanced Search Trees

Walls & Mirrors Chapters 12, 13

- Homework 4 extension
- Next week: Programming quiz during recit
- midterm 2 April 8th (in class)
- New partners and programming assignment

---

Value Oriented Data Structures

- Value-oriented operations are very common:
  - Find the phone number of John Smith
  - Add a user to a database (e.g., Netflix database).
- To support such uses: Arrange the data to facilitate search/insertion/deletion of an item given its search key

---

Table ADT

- Manage data accessible by search key

Operations

- Create empty
- Is empty?
- Size
- Insert new item
- Delete item with search key
- Retrieve item by search key
- Traverse items (in sorted order)
(Why is traversal useful?)

Pseudocode for Table ADT

- `createTable()`: creates an empty table
- `tableIsEmpty()`: boolean
  - Determines whether a table is empty
- `tableLength()`: integer
  - Determines the number of items in a table
- `tableTraverse()`: TableItemType
  - Traverses a table (in sorted search key order).

---

Pseudocode for Table ADT

- `tableInsert()`: throws TableException
  - Inserts `newItem` into a table whose items have distinct search keys that differ from `newItem`'s search key.
  - Throws exception if unsuccessful.
- `tableDelete(searchKey: KeyType)`: boolean
  - Deletes from a table the item whose search key equals `searchKey`.
  - Returns true if successful, false if no item found.
- `tableRetrieve(searchKey: KeyType)`: TableItemType
  - Returns item whose search key equals `searchKey`. Returns null if not there.

---

Table Items

```java
public abstract class KeyedItem<KT extends Comparable<? super KT>> {
    private KT searchKey;

    public KeyedItem(KT key) {
        searchKey = key;
    }

    public KT getKey() {
        return searchKey;
    }
}
```
Table Items Example

```java
Public class User extends KeyedItem<String> {
    private String id;  // search key
    private String firstName;
    private String lastName; ...
    public User(String userID, String _firstName, ...) {
        super(userID);  firstName = _firstName; ...
    }
}
```

Table Interface

```java
public interface TableInterface<T extends KeyedItem<KT>, KT extends Comparable<? super KT>> {
    // Precondition for all operations:
    // No two items of the table have the same search key.
    // The table’s items are sorted by search key (actually not required)
    public boolean tableIsEmpty();  // Determines whether a table is empty.
    // Postcondition: Returns true if the table is empty; false otherwise
    public int tableLength();  // Determines the length of a table.
    // Postcondition: Returns the number of items in the table.

    Note:  This is the book’s version.  The table’s items need not be sorted
```

Table Interface (cont.)

```java
public void tableInsert(T newItem) throws TableException;  // Inserts an item into a table in its proper sorted order according
                                      // to the item’s search key.
    // Precondition: The item is newItem, whose search key would be
    // unique in the table.
    // Postcondition: If successful, newItem is in its proper order in
    // table.  Otherwise, table is unchanged; throw TableException.
    public KeyedItem tableRetrieve(KT searchKey);  // Retrieves an item with a search key KT from table.
    // Precondition: searchKey is the search key of item to be retrieved.
    // Postcondition: If the retrieval was successful,
    // table item with search key matching KT is returned.
    // If no such item exists, return null.
```

Possible Implementations

- ArrayList sorted by search key
  - Pros: fast access via binary search
  - Cons: may waste space, add and delete are expensive
- Array sorted by search key
  - Similar to ArrayList
- Linked List sorted by search key
  - Cons: expensive retrieval, add and delete
- Binary search tree
  - Pros: fast average access, efficient use of space
  - Cons: poor worst case performance

Performance of Table Implementations

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array-based</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted array-based</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>BST</td>
<td>O(log n) (O(n))</td>
<td>O(log n) (O(n))</td>
<td>O(log n) (O(n))</td>
</tr>
</tbody>
</table>

* Worst case behavior in parentheses
Balanced Search Trees

- The efficiency of binary search trees is related to the tree’s height
  - Height of a binary search tree of n items
    - Maximum: n
    - Minimum: \(\lceil \log_2(n+1) \rceil\)
- Height of a binary search tree is sensitive to the order of insertions and deletions
- Balanced search trees: height management

2-3 Trees

- A 2-3 tree has 2-nodes and 3-nodes
  - 2-nodes: one data item and two children
  - 3-nodes: two data items and three children

2-3 Trees

- 2-node: a single item whose search key is
  - Greater than the left child’s search key(s)
  - Less than the right child’s key(s)

- 3-node: two items whose search keys S and L satisfy:
  - S > left child’s search key(s), less than the middle child’s search key(s)
  - L > middle child’s search key(s), less than the right child’s search key(s)
Properties of 2-3 Trees

- A leaf may contain either one or two data items
- All leaf nodes are at the same level
- Never taller than a minimum-height binary tree
  - A 2-3 tree with \( n \) nodes has height less than \( \lceil \log_2(n + 1) \rceil \)

```
public class TreeNode<T> {
    private T smallItem;
    private T largeItem;
    private TreeNode<T> leftChild;
    private TreeNode<T> midChild;
    private TreeNode<T> rightChild;
    ...
}
```

Traversing 2-3 Trees

```java
inorder(in TreeNode<TwoThreeTree>)
if (ttTree's root node is a leaf) {
    visit the data item(s)
} else if (root has two data items) {
    inorder(left subtree)
    visit first data item
    inorder(middle subtree)
    visit second data item
    inorder(right subtree)
} else {  // root has one data item
    inorder(left subtree)
    visit data item
    inorder(right subtree)
}
```

Inserting 39: space available in leaf

```
inserting 38: no space in leaf, but space in parent
```

A 2-3 node

```
public class TreeNode<T> {
    private T smallItem;
    private T largeItem;
    private TreeNode<T> leftChild;
    private TreeNode<T> midChild;
    private TreeNode<T> rightChild;
    ...
}
```

Searching 2-3 Trees

- Analogous to binary search trees
- As efficient as searching the shortest binary search tree
  - Number of comparisons required to search a 2-3 tree:
    - Approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible
  - Searching a 2-3 tree is \( O(\log_2 n) \)
**Insert**

After inserting 37

- To insert an item:
  - Locate the leaf at which the search would terminate
  - Insert the new item into the leaf
  - If the leaf now contains only two items, you are done
  - If the leaf now contains three items, split the leaf into two nodes, \( n_1 \) and \( n_2 \)
  - If the parent contains two items you are done.
  - Otherwise - it contains three items and has 4 children.

**Inserting 36**

No space in leaf or immediate parent...

**Insert (cont)**

- When an internal contains 3 items:
  - Split the node into two nodes
  - Accommodate the node’s children

**Insert (cont)**

- When the root contains three items
  - Split the root into two nodes
  - Create a new root node

**Delete**

Deleting 70

- Swap with inorder successor

- Root contains three items
- Split the root into two nodes
- Create a new root node
Delete

Deleting 70

(a) 80 90
(b) 60
Delete value from leaf

(c) 90
(d) 80 20
Merge nodes by deleting empty leaf and moving 80 down

Delete

Deleting 100

(a) 90
(b) 10 20
(c) 40
(d) 60 80
Delete 100

Next step:
Delete 100

Delete

Deleting 80

(a) 90
(b) 10 20
(c) 40
(d) 60 80
Delete 80

Swap with inorder successor

Delete

Deleting 80

(a) 90
(b) 10 20
(c) 40
(d) 60 80
Delete 80

Node becomes empty

Swap with inorder successor

Delete

Deleting 80

(a) 90
(b) 10 20
(c) 40
(d) 60 80
Delete 80

Node becomes empty

Remove empty root
Delete

- Locate the node that needs to be deleted
- If it's not a leaf: find its inorder successor and swap
- If the leaf contains two items you can just delete one.
- Otherwise deletion would leave an empty node: need to do some more work

The same operations are performed in interior nodes

Complexity of 2-3 Trees

- Insert?
- Delete?

Comparison with Binary Search Trees

- A 2-3 tree is guaranteed to be balanced.
- A 2-3 will usually have reduced height in comparison to a balanced binary search tree, but requires more time to go through each node.
B-Trees

2-3 tree are a special case of B-trees:

![Diagram showing 2-3 tree structure]

- The B-tree's creators, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the B stands for.

2-3-4 Trees

- Another special case of B-trees
- Algorithms are similar to 2-3 trees
- Some improvements over 2-3 trees:
  - Insert/delete in a single pass from the root
  - The basis for red-black trees which implement 2-3-4 trees using binary trees

2-3-4 Trees

- A 4-node in a 2-3-4 tree:

![Diagram showing 2-3-4 tree example]

- Advantages
  - balanced
  - insertion and deletion operations use only one pass from root to leaf
- Disadvantage
  - more storage than a binary search tree
- Red-Black trees
  - A representation of 2-3-4 trees using binary trees
  - The advantages of a 2-3-4 tree, without the storage cost

Red-Black Trees

- Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree
Red-Black Trees

- Red-Black tree is a binary search tree.
- Search and traversal are the same as a BST.
- Insert and delete – need to translate the 2-3-4 operations

AVL Trees

- Balanced binary search trees
- Named after Adelson-Velsky and Landis
- Can be searched almost as efficiently as a minimum-height binary search tree (same big-O bounds)
- Basic strategy:
  - After each insertion or deletion
    - Check if the tree is still balanced
    - Restore balance if necessary

Rotations restore balance:

- Single rotation
- Double rotation

AVL Trees

- Advantage
  - Efficient operations
- Disadvantage
  - AVL tree implementation more difficult than other search trees. Fine for coding libraries

Search Trees in Java

```java
public class TreeMap<K,V> extends AbstractMap<K,V>
    implements SortedMap<K,V>

Implementation based on Red-Black trees
```

One of these methods is not very efficient:

- boolean containsKey(Object key)
- boolean containsValue(Object value)