Computer Science

Professor Tom Ellman
Lecture 16

Telephone Book Problem

• Operations:
  – Insert a name and number into the book.
  – Delete a (name,number) entry from the book.
  – Look up the number associated with a given name.

• Issues:
  – What type of data structure(s) should be used to keep track of the names and numbers?
  – How can we make the operations listed above as fast as possible?

Set of Numbers Problem

• Operations:
  – Insert a number into the set.
  – Delete a number from the set.
  – Check whether a number is a member of the set.

• Issues:
  – What type of data structure(s) should be used to keep track of the numbers in the set?
  – How can we make the operations listed above as fast as possible?
A Straw Man

- Represent the set as a linked list.
- Insert new objects into position on the list to maintain an ordering relation.
- Avoid inserting duplicate items.
- Delete an object by searching the list, locating the first node holding the object and removing the node from the list.
- Look up an object by searching the list until finding a node holding it or reaching the end of the list.

```java
public interface SortedSet<T extends Comparable> {
    public boolean isEmpty();
    public T insert(T item);
    public T delete(T item);
    public T member(T item);
    public Queue<T> elements();
}
```

Our Sorted Set implementation will support this interface.

```java
public class SortedListSL<T extends Comparable<T>>
    implements SortedSet<T> {

    private ListElementSL<T> head;

    public SortedListSL() { this.head = null; }

    public boolean isEmpty() { return head==null; }

    // ... Omitted ... 
}
```

Our SortedList implementation is similar to the ListSL implementation.
public T insert(T item) {
    head = insertHelper(item, head);
    return item;
}

private ListElementSL<T> insertHelper(T item, ListElementSL<T> head) {
    if (head==null) return new ListElementSL<T>(item, null);
    else {
        int cmp = item.compareTo(head.data());
        if (cmp<0) return new ListElementSL<T>(item, head);
        else if (cmp==0) return new ListElementSL<T>(item, head.next());
        else head.setNext(insertHelper(item, head.next()));
        return head;
    }
}

Our (recursive) **insert** method keeps the list in order according to the **compareTo** method. It also avoids storing duplicate elements.

public T delete(T item) {
    head = deleteHelper(item, head);
    return item;
}

private ListElementSL<T> deleteHelper(T item, ListElementSL<T> head) {
    if (head==null) return null;
    else {
        int cmp = item.compareTo(head.data());
        if (cmp<0) return head;
        else if (cmp==0) return head.next();
        else {
            head.setNext(deleteHelper(item, head.next()));
            return head;
        }
    }
}

Our (recursive) **delete** method takes advantage of the fact that the list is in order.

public T member(T item) {
    return memberHelper(item, head);
}

private T memberHelper(T item, ListElementSL<T> head) {
    if (head==null) return null;
    else {
        int cmp = item.compareTo(head.data());
        if (cmp<0) return null;
        else if (cmp==0) return head.data();
        else return memberHelper(item, head.next());
    }
}

Our (recursive) **member** method also takes advantage of the fact that the list is in order.
The elements method returns a queue implemented as a doubly linked list.

Evaluation of this Approach

- It’s easy to implement.

- The procedures run slowly if the list is long:
  - Suppose the list has length $N$.
  - Each operation takes about $N/2$ steps on average.
  - We say each operation takes “O(N)” time.

Binary Search Tree

- Store the objects in a tree structure.
- The root of the tree holds a data object.
- The left subtree holds numbers “less” than data.
- The right subtree holds numbers “greater” than data.
- Each subtree stores objects in the same way as the whole tree.
Structure of a Binary Search Tree

Example: \{0, 2, 3, 5, 7, 8, 9\}

Our Binary Search Tree implementation will also support this interface.
public interface PriorityQueue<D extends Comparable<D>> {
    public boolean isEmpty();
    public D insert(D item);
    public D delete();
    public D min();
}

Our Binary Search Tree implementation will also support this interface.

Implementation of Binary Search Trees

- Class `BSTree<T>`:
  - Represents an entire binary search tree.
  - Holds a reference (root) to a `BSTreeNode<T>` object.
  - Most of its methods call corresponding methods of `BSTreeNode` class.
- Class `BSTreeNode<T>`:
  - Represents a single node of a binary search tree.
  - Also represents a subtree of a binary search tree.
  - Holds references to `left (BSTreeNode<T>)`, `right (BSTreeNode<T>)` and `data (T)`.
package bstreetest;

public class BSTree<T extends Comparable<T>> implements PriorityQueue<T>, SortedSet<T> {

    private BSTreeNode<T> root;  // A private instance variable
    // root holds a reference to the
    // root of the tree.

    public BSTree() {
        root = null;  // An empty BSTree has its
        // root equal to null.
    }

    public boolean isEmpty() {
        return root == null;
    }
    // ... Omitted ...

    public T insert(T d) {
        if (isEmpty()) {
            root = new BSTreeNode<T>(d);
        } else {
            root.insert(d);
        }
        return d;
    }

    The BSTree insert method creates a new root node
    if the tree is empty. Otherwise, it passes the insert call
to the root node. In either case it returns its argument.

    public T delete(T d) {
        if (!isEmpty()) {
            root = root.delete(d);
        }
        return d;
    }

    The BSTree delete method does nothing if the tree
    is empty. Otherwise, it passes the delete call to the
    root node. In either case it returns its argument.
public T member(T d) {
    if (isEmpty()) {
        return null;
    }
    return root.member(d);
}

The BSTree member method returns null if the tree is empty. Otherwise, it passes the member call to the root node. This will return the object found in the tree, or else null if no object was found.

public T min() {
    return root.min();
}

class BSTree {
    private T min() {
        return root.min();
    }
}

The BSTree min method just passes the call to the root node. This method may not be invoked on an empty BSTree.

public T deleteMin() {
    T temp = root.min();
    root = root.deleteMin();
    return temp;
}

The BSTree deleteMin method first gets the min value from the root node. It then passes the deleteMin call to the root node. Finally it returns the min value acquired from the root node. This method may not be invoked on an empty BSTree.

public T max() {
    return root.max();
}

class BSTree {
    private T max() {
        return root.max();
    }
}

The BSTree max method just passes the call to the root node. This method may not be invoked on an empty BSTree.

public T deleteMax() {
    T temp = root.max();
    root = root.deleteMax();
    return temp;
}

The BSTree deleteMax method first gets the max value from the root node. It then passes the deleteMax call to the root node. Finally it returns the max value acquired from the root node. This method may not be invoked on an empty BSTree.
public T delete() {
    return deleteMin();
}

To implement the PriorityQueue interface, we need a delete method. It just calls deleteMin.

public Queue<T> elements() {
    Queue<T> elementQueue = new QueueListSL<T>();
    if (root!=null) {
        root.enQueueElements(elementQueue);
    }
    return elementQueue;
}

To implement the SortedList interface, we need an elements method that returns a queue. If the root is null, we just return an empty queue. Otherwise, call a the enQueueElements method of the BSTreeNode class.

class BSTreeNode<T extends Comparable<T>> {
    private T data;
    private BSTreeNode<T> left;
    private BSTreeNode<T> right;

    public BSTreeNode(T d) {
        data = d;
        left = null;
        right = null;
    }

    // . . . Omitted . . .

    A BSTreeNode has private instance variables to hold the stored data and the left and right subtrees.

    The BSTreeNode constructor takes a Comparable object d and stores it in the data variable. The left and right subtrees are initially null.
Finding the Smallest Object in BST

• If the **left** subtree of BST is empty, then return the root **data** of BST.

• Otherwise, return the smallest number in the **left** subtree of BST.

Finding the Largest Object in BST

• If the **right** subtree of BST is empty, then return the root **data** of BST.

• Otherwise, return the smallest number in the **right** subtree of BST.

```java
public T min() {
    if (left == null) {
        return data;
    } else {
        return left.min();
    }
}

public T max() {
    if (right == null) {
        return data;
    } else {
        return right.max();
    }
}
Is an object \( d \) a member of BST?

- If the root data of BST equals \( d \), then return \texttt{true}.
- If \( d < \text{data} \) then if left subtree is empty return \texttt{false}, otherwise look for \( d \) in the left subtree.
- If \( d > \text{data} \) then if right subtree is empty return \texttt{false}, otherwise look for \( d \) in the right subtree.

```java
public T member(T d) {
    int cmp = d.compareTo(data);
    if (cmp == 0) {
        return data;
    }
    if (cmp < 0) {
        if (left == null) {
            return null;
        } else {
            return left.member(d);
        }
    } else {
        if (right == null) {
            return null;
        } else {
            return right.member(d);
        }
    }
}
```

Why return \texttt{data} or \texttt{null}, rather than \texttt{true} or \texttt{false} boolean values?

Hint: If \( d.compareTo(data) \) is zero, \( d.equals(data) \) may be \texttt{true} or \texttt{false}.

Inserting the object \( d \) into BST

- If the root data of BST is equivalent to \( d \), then replace \( d \) with data.
- If \( d < \text{data} \) then insert \( d \) into the left subtree.
- If \( d > \text{data} \) then insert \( d \) into the right subtree.
public void insert(T d) {
    int cmp = d.compareTo(data);
    if (cmp==0) {
        data = d;
        return;
    }
    if (cmp < 0)
        if (left!=null) left.insert(d);
        else left = new BSTreeNode<T>(d);
    else
        if (right!=null) right.insert(d);
        else right = new BSTreeNode<T>(d);
}

How could we modify this definition to allow duplicates, i.e., two more values \(d_1\) and \(d_2\) where \(d_1\).compareTo\(d_2\) is zero?

---

**Deleting the object** \(d\) **from BST**

- If the root \(data\) of BST is equivalent to \(d\), then call a special procedure to delete the root of BST.
- If \(d < data\) then delete \(d\) from the left subtree:
- If \(d > data\) then delete \(d\) from the right subtree.

---

```java
public BSTreeNode<T> delete(T d) {
    int cmp = d.compareTo(data);
    if (cmp == 0) {
        return deleteRoot();
    } else {
        if (cmp < 0) {
            if (left != null)
                left = left.delete(d);
        } else {
            if (right != null)
                right = right.delete(d);
        } return this;
    }
}
```
Deleting the root of BST

• Version 1:
  • If the left subtree of BST is not empty, then delete the largest object \( e \) in the left subtree and store \( e \) in the root of BST.
  • Otherwise, return the right subtree of BST.

• Version 2:
  • If the right subtree of BST is not empty, then delete the smallest object \( e \) in the right subtree and store \( e \) in the root of BST.
  • Otherwise, return the left subtree of BST.

Replacing Root with Maximum of Left Subtree

```
private BSTreeNode<T> deleteRoot() {
    if (left != null) {
        data = left.max();
        left = left.deleteMax();
        return this;
    } else {
        return right;
    }
}
```
Deleting the Smallest Object in BST

- If the left subtree of BST is empty, then the root data is smallest, so just return the right subtree of BST.

- Otherwise, delete the minimum from the left subtree, and return this subtree.
Deleting the Largest Object in BST

- If the right subtree of BST is empty, then the root data is largest, so just return the left subtree of BST.

- Otherwise, delete the maximum from the right subtree, and return this subtree.

```java
public BSTreeNode<T> deleteMin() {
    if (left == null) {
        return right;
    } else {
        left = left.deleteMin();
        return this;
    }
}
```

```java
public BSTreeNode<T> deleteMax() {
    if (right == null) {
        return left;
    } else {
        right = right.deleteMax();
        return this;
    }
}
```
public T value()
{
    return data;
}

public BSTreeNode<T> left()
{
    return left;
}

public BSTreeNode<T> right()
{
    return right;
}

public void enQueueElements(Queue<T> queue) {
    if (left != null) {
        left.enQueueElements(queue);
    }
    queue.enQueue(data);
    if (right != null) {
        right.enQueueElements(queue);
    }
}

The enQueueElements puts all the data in this subtree into the given queue, in order, modifying the queue as a side effect and returning nothing. This method simply traverses the tree following the inorder ordering.

What have we gained?

• The code is more complicated.

• Each operation takes a number of steps that is roughly proportional to the depth of the tree.
Balanced Tree

Unbalanced Tree

Depth of a Balanced Binary Search Tree Containing N Numbers

\[ N = 1 + 2 + 4 + \ldots + 2^d \]
\[ N = 2^0 + 2^1 + 2^2 + \ldots + 2^d \]
\[ N = (2^{d+1}) - 1 \]
\[ N = 2^{d+1} \]
\[ d = \log_2 N - 1 \]
Depth of a Unbalanced Binary Search Tree Containing N Numbers

\[ d = N \]

Moral of the Story

- If the tree is balanced, each operation will take about \( \log_2 N \) steps.
  - Each operation takes \( O(\log N) \) time.
  - Compared to \( O(N) \) time for the straw man.
- If the tree is not balanced, each operation will take about \( N \) steps.
  - Each operation takes \( O(N) \) time.
  - Compared to \( O(N) \) time for the straw man.
- The BST implementation is as fast as or faster than the straw man implementation.
Balancing a Binary Search Tree

• Let’s keep the tree balanced.
• Store size of subtree in each node.
• Rebalance tree periodically after a number of insert or delete operations.
• How often to rebalance the tree?
• How to rebalance the tree?
• Is there a way to keep it balanced at all times?