Introduction to Search Trees

Want to maintain an *ordered* collection (dictionary) $D$ of $n$ items $(k, e)$ to support

- **findElement(k)**: If $D$ contains an item with key equal to $k$, then return the element. Otherwise, return a *sentinel* `NO_SUCH_KEY`
- **insertItem(k,e)**: Duplicates may or may not be allowed.
- **removeElement(k)**
- **closestElement(k)**: Find the item whose key value is closest to $k$.

Binary search trees (BST) is a good choice (expected time $O(\log n)$ if there are no `removeElement()` calls)
Balanced binary search trees (e.g. AVL trees) is the method of choice if we want to guarantee worst case time $O(\log n)$
Straightforward Implementations are $O(n)$

- An unordered array is such that inserting takes $O(1)$ time, but searching or removing takes $O(n)$
- Can be used to maintain log files (frequent insertions, rare searches or removals)
- An ordered array is such that inserting and removing takes $O(n)$ time, but searching takes only $O(\log n)$ time.
- Can be used for lookup tables (frequent searches, with rare insertions or removals)
- A binary search tree attempts to do better
Definition

A binary search tree is a binary tree $T$ such that at each internal node $v$

- An item $(k, e)$ is stored
- Keys stored at nodes in the left subtree of $v$ are less than or equal to $k$
- Keys stored at nodes in the right subtree of $v$ are greater than $k$

External nodes do not hold items but serve as place holders
Search: Most Frequent Usage

A successful search traverses a path starting at the root and ends at an internal node

Node find(Key k, Node p) {
    if (isExternal(p)) return p;
    if (k == p.key) return p;
    if (k < p.key) return find(k, p.left);
    return find(k, p.right);
}
Insertion Follows Find

• Let \( w \) be an external node returned by \( \text{find}(k, \text{root}) \). If we had access to a parent pointer, we can accomplish \( \text{insert}(k, \text{root}) \) by hanging the item to the parent.

• This is particularly easy if we had the (placeholder) external nodes found in extended binary trees.

• If \( w \) is an internal node, and if duplicates are permitted, we call the algorithm recursively on the left child of \( w \).
**Insertion takes** $O(h)$ **time**

If parent pointers are not available, we can use recursion to keep track of the parent

```java
Node insert(Key k, Node p) {
    if (isExternal(p)) return new Node(k, null, null);
    if (k <= p.key) p.left = insert(k, p.left);
    else if (k > p.key) p.right = insert(k, p.right);
    return p;
}
```
Three Cases for Removal

- Removal also follows a find()
- If the node $w$ to be removed is such that
  1. Both children of $w$ are external, then we set the parent of $w$ to be external
  2. Only one of $w$’s children $v$ contains a valid item, we set the child of the parent of $w$ to be $v$.

- All cases are easily implemented if we have parent pointers
Three Cases for Removal

If the node $w$ to be removed is such that both children of $w$ contain valid items, we find a replacement for the item at $w$ and then remove the replacement.

The replacement will satisfy one of the two conditions mentioned.
Removal Code

Recursion keeps track of parent pointers

Node remove(Key k, Node p) {
    if (isExternal(p)) return p;
    if (k < p.key) p.left = remove(k, p.left);
    else if (k > p.key) p.right = remove(k, p.left);
    else if (!isExternal(p.left) && !isExternal(p.right) {
        rep = findMin(p.right);
        p.item = rep.item;
        p.right = remove(rep.key, p.right);
    }
    else if (isExternal(p.left)) p = p.right;
    else p = p.left;
    return p;
}
Good and Bad Binary Search Trees

• Given (offline) a set of keys, there exist many binary search trees that can be built based on these keys

• Questions
  – Assume: All keys are equally likely candidates for searches. What is the best BST? How to construct one?
  – Assume: All keys are not equally likely candidates, but the probabilities are known. What is the best BST? How to construct one?

• Given two trees, which one is better for successful (unsuccessful) search? (Assume equally likely situation)
Good and Bad Binary Search Trees

• Metrics for $n$ node BST

  – The *internal path length* $I$ is the sum of the lengths of the paths from the root to each internal node (Example: 12)

  – The *external path length* $E$ is the sum of the lengths of the paths from the root to each external (placeholder) node (Example: 25)

  \[ E = I + 2n \]

• Which tree is better for successful search?
Good and Bad Binary Search Trees

Assuming that all searches are equally likely in a tree with $n$ nodes

- The tree with maximum path length is the one with the degenerate linear structure: $I = 0.5(n^2 - n)$

- The tree with minimum path length is a complete binary tree. Its length is (approximately) $O(n \log n)$

- The “average” path length over all binary trees is interesting . . .
  
  - We could look at all possible binary search trees with $n$ nodes
  
  - If all of them are equally likely, we could use the heights of each tree to compute the expected height (If all of them are not equally likely, then we would need some sort of probability distribution.)

- . . . But we are actually interested in finding the cost of a sequence of $n$ insert(), find(), and remove() operations
Expected Height of Binary Search Tree

- We circumvent this problem by the following reasonable model
  - No find() or remove()
  - $n$ insert() calls are modeled by an array of $n$ distinct number
  - Any of the $n!$ permutations of the input is equally likely
  - The height of the leftmost branch of the resulting binary tree is the expected height.

- Example

```
Insertion order: 9 3 1 0 6 1 8 5

Left chain has 3 nodes
```

```
Insertion order: 8 9 5 1 0 3 6 1

Left chain has 4 nodes
```
Expected Height of Leftmost Branch

- Crucial observation: The number of nodes on the leftmost branch is equal to the number of minimum changes in the input array.
- Expected number of minimum changes in a randomly permuted array:
  - Let $Y$ be the random variable that represents the number of minimum changes.
  - Let $x_i$ be the indicator random variable that is true if the $i$th number is the minimum. Thus $Y = \sum x_i$.
  - The probability that the $i$th number is the minimum is $\frac{1}{i}$.
  - $E[Y] = \sum E[x_i] = \sum \frac{1}{i} = \ln n + O(1)$
- Expected height is $\Theta(\log n)$