Binary Search Trees

Search

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst case O(N) for either searching or inserting

*Need data structure with optimal efficiency for searching and inserting*

Symbol Table

Defn: abstract data structure of items with keys that supports two basic operations: *insert* a new item, and *return* an item with a given key
## Symbol Table: ADT

- **Insert** a new item
- **Search** for an item (items) having a given key
- **Remove** a specified item
- **Sort** the symbol table
- **Select** the kth largest item in a symbol table
- **Join** two symbol tables

Also may want construct, test if empty, destroy, copy...

---

## Binary Search Tree

**Def'n:** a binary tree that has a key associated with each of its internal nodes, with the additional property that the key in any node is $\geq$ keys in all nodes of left subtree and $\leq$ keys in all nodes in right subtree

Essential property of BST is that **insert** is as easy to implement as **search**

---

## A Word on Binary Search Trees

- A binary tree node is proper iff it has
  - exactly zero children
  - OR exactly two children
- What happens with tree of two items?
- **Answer:** we count NULLs as external nodes
  - each “bottom node” actually points to two NULLS
  - each “half node” actually points to one node and one NULL
- This means all elements are “internal nodes”
**Node: Concrete Implementation**

Node in Binary Tree

```c
struct node
    {Item item; node *left, *right};
typedef node *link;
```

- A node contains some information, and points to its left child node and right child node
- Efficient for moving down a tree from parent to child

---

**Search (§9.1.1)**

- To search for a key $k$, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of $k$ with the key of the current node
- If we reach a non-matching leaf, the key is not found and we return a null position
- Example: search(4, root)

```c
Algorithm search(k, v)
if T.isExternal(v) // not found
    return Position(null)
if k < key(v)
    return search(k, T.leftChild(v))
else if k = key(v)
    return Position(v)
else // k > key(v)
    return search(k, T.rightChild(v))
```

---

**Insertion (§9.1.2)**

- To perform operation insert($k$, $e$), we search for key $k$
- Assume $k$ is not already in the tree, and let let $w$ be the leaf reached by the search
- We insert $k$ at node $w$ and expand $w$ into an internal node
- Example: insert(5, e)
Sort: Binary Search Tree

Inorder Traversal

Algorithm inorder(T,v)
   for left child w of v do
      inorder(T,w)
   visit node v
   for right child x of v do
      inorder(T,x)

simply stated, sorting a binary search tree is inorder traversal of the tree

Properties of BSTs

- Best case (balanced): about lg N nodes between root and each external node/leaf
- Worst case (unbalanced): about N nodes between root and each external node/leaf
- With random data:
  - trees are likely to be well-balanced on average
  - same reason that quicksort likely to partition adequately on average

Modifications to a Theme

- What if we want to remove an internal node from the tree?
- What if we want to make a particular node the root?
- What if we want to insert at the root, rather than a leaf?
- What if we want to combine two BSTs?
- What if pigs could fly?

Common feature: working with internal nodes (and flying pigs)
Deletion: Binary Search Tree

- First, search for the node to remove
  - if you don’t find it, do nothing
- If you find it, examine its children
  - if no children, trivial to remove
  - if one child, remove node and replace with child
  - if two, replace with a “combined” tree of both
- Key observation
  - all in LHS subtree <= all in RHS subtree
  - partition RHS so that its smallest node is its root
    - must be some such node, since RHS is not empty
    - new root has a right child, but no left child
  - make new root’s left child the LHS subtree

Joining Two Children, Illustrated

Rotations

- Right Rotation: RR(P)
- Left Rotation: LR(P)
Common Technique: Rotations

- **Rotation:**
  - interchange the role of a parent and one of its children in a tree...
  - while still preserving the BST ordering among the keys in the nodes
- **The second part is tricky**
  - right rotation: copy the right link of the left child to be the left link of the old parent
  - left rotation: copy the left link of the right child to be the right link of the old parent

*Rotation is a local change involving only three links and two nodes*

---

Summary: Binary Search Trees

- **Linear structures (arrays & linked lists)**
  - either insertion, search, or both are $O(N)$
- **Tree structures**
  - each node points to two others (left, right)
  - all nodes are ordered: left <= root <= right
  - modification of nodes
    - external is easy
    - internal require rotations
  - in general, operations on BSTs are:
    - $O(\log N)$ average
    - $O(N)$ worst case
    - worst case is easy to generate