CSE 326: Data Structures
Lecture #7
Binary Search Trees

Henry Kautz
Winter Quarter 2002

**Binary Trees**

- **Properties**
  - Notation:
    - \( \text{depth(tree)} = \text{MAX} \{ \text{depth(leaf)} \} = \text{height(root)} \)
    - max \# of leaves = \( 2^{\text{height(root)}} \)
    - max \# of nodes = \( 2^{\text{height(root)}} + 1 \)
    - max depth = \( n - 1 \)
    - average depth for \( n \) nodes = \( \sqrt{n} \)
    (over all possible binary trees)

- **Representation:**

**Dictionary & Search ADTs**

- Operations
  - create
  - destroy
  - insert
  - find
  - delete

- **Notation:**
  - `insert`
  - `find(key/leaf)`
  - `find(key/leaf)`

- **Dictionary:** Stores values associated with user-specified keys
  - keys may be any (homogenous) comparable type
  - values may be any (homogenous) type
  - implementation: data field is a struct with two parts

- **Search ADT:** keys = values

**Naïve Implementations**

<table>
<thead>
<tr>
<th></th>
<th>unsorted array</th>
<th>sorted array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>( \Theta(n) ) (if no shrink)</td>
<td>( O(n) )</td>
<td>( \Theta(n) ) (if no shrink)</td>
</tr>
<tr>
<td>find</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>delete</td>
<td>( \Theta(n) ) (if no shrink)</td>
<td>( O(n) )</td>
<td>( \Theta(n) ) (if no shrink)</td>
</tr>
</tbody>
</table>

**Goal:** fast find like sorted array, dynamic inserts/deletes like linked list

**Binary Search Tree Dictionary Data Structure**

- **Search tree property**
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
  - result:
    - easy to find any given key
    - inserts/deletes by changing links
In Order Listing

- visit left subtree
- visit node
- visit right subtree

In order listing:
2→5→7→9→10→15→17→20→30

Finding a Node

Node *k find(Comparable x, Node * root) {
  if (root == NULL)
    return root;
  else if (x < root->key)
    return find(x, root->left);
  else if (x > root->key)
    return find(x, root->right);
  else
    return root;
}

runtime:

Insert

Concept: proceed down tree as in Find; if new key not found, then insert a new node at last spot traversed

void insert(Comparable x, Node * root) {
  assert (root != NULL);
  if (x < root->key)
    if (root->left == NULL)
      root->left = new Node(x);
    else insert(x, root->left);
  else if (x > root->key)
    if (root->right == NULL)
      root->right = new Node(x);
    else insert(x, root->right);
  else return root;
}

Tricky Insert

C++ trick: use reference parameters

void insert(Comparable x, Node * & root) {
  if (root == NULL)
    root = new Node(x);
  else if (x < root->key)
    insert( x, root->left );
  else insert( x, root->right );
}

Works even when called with empty tree –
node * myTree = NULL;
insert( something, myTree );
sets the variable myTree to point to the newly created node

Digression: Value vs. Reference Parameters

- Value parameters (Object foo)
  - copies parameter
  - no side effects
- Reference parameters (Object & foo)
  - shares parameter
  - can affect actual value
  - use when the value needs to be changed
- Const reference parameters (const Object & foo)
  - shares parameter
  - cannot affect actual value
  - use when the value is too intricate for pass-by-value
Really Tricky Insert

```c
void insert(Comparable x, Node * & root) {
    Node * & target = find(x, root);
    if (target == NULL) {
        target = new Node(x);
    }
}
```

BuildTree for BSTs

Suppose $a_1, a_2, \ldots, a_n$ are inserted into an initially empty BST:
1. $a_1, a_2, \ldots, a_n$ are in increasing order
2. $a_1, a_2, \ldots, a_n$ are in decreasing order
3. $a_i$ is the median of all, $a_i$ is the median of elements less than $a_i$, $a_i$ is the median of elements greater than $a_i$, etc.
4. data is randomly ordered

Analysis of BuildTree

- Worst case is $O(n^2)$
  
  $1 + 2 + 3 + \ldots + n = O(n^2)$

- Average case assuming all input sequences are equally likely is $O(n \log n)$
  - equivalently: average depth of a node is $\log n$
  - proof: see Introduction to Algorithms, Cormen, Leiserson, & Rivest

Proof that Average Depth of a Node in a BST constructed from random data is $O(\log n)$

- Calculate sum of all depths, divide by number of nodes
- $D(n) = \text{sum of depths of all nodes in a random BST containing } n \text{ nodes}$
- $D(n) = D(\text{left subtree}) + D(\text{right subtree}) + 1^*(\text{number of nodes in left and right subtrees})$
- $D(n) = D(L) + D(n-L-1) + (n-1)$
- For random data, all subtree sizes equally likely
  
  \[
  D(n) = \left[ \frac{1}{n} \sum_{L=0}^{n-1} (D(L) + D(n-L-1)) \right] + (n-1)
  \]
- $D(n) = D(a \log n)$

Deletion

![Deletion Tree]

Why might deletion be harder than insertion?

FindMin/FindMax

```c
Node * min(Node * root) {
    if (root->left == NULL) {
        return root;
    } else {
        return min(root->left);
    }
}
```

How many children can the min of a node have?
**Successor**

Find the next larger node in this node’s subtree.
- not next larger in entire tree

```c
Node * succ(Node * root) {
    if (root->right == NULL)
        return NULL;
    else
        return min(root->right);
}
```

How many children can the successor of a node have?

---

**Predecessor**

Find the next smaller node in this node’s subtree.

```c
Node * pred(Node * root) {
    if (root->left == NULL)
        return NULL;
    else
        return max(root->left);
}
```

---

**Deletion - Leaf Case**

Delete(17)

---

**Deletion - One Child Case**

Delete(15)

---

**Deletion - Two Child Case**

Delete(5)

replace node with value guaranteed to be between the left and right subtrees: the successor

Could we have used the predecessor instead?

---

**Deletion - Two Child Case**

Delete(5)

always easy to delete the successor – always has either 0 or 1 children!
**Deletion - Two Child Case**

Delete(5) 5

3

2

1

Finally copy data value from deleted successor into original node

**Lazy Deletion**

- Instead of physically deleting nodes, just mark them as deleted
  + simpler
  + physical deletions done in batches
  + some adds just flip deleted flag
  - extra memory for deleted flag
  - many lazy deletions slow finds
  - some operations may have to be modified (e.g., min and max)

**Dictionary Implementations**

<table>
<thead>
<tr>
<th></th>
<th>Unsorted Array</th>
<th>Sorted Array</th>
<th>Linked List</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>find + O(1)</td>
<td>O(n)</td>
<td>find + O(1)</td>
<td>O(Depth)</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>find + O(1)</td>
<td>O(n)</td>
<td>find + O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

BST’s looking good for shallow trees, *i.e.* the depth D is small (log n), otherwise as bad as a linked list!