CS 171: Introduction to Computer Science II

Binary Search Trees
Binary Search Trees

- Symbol table applications
- BST definitions and terminologies
- Search and insert
- Traversal
- Ordered operations
- Delete
Symbol tables and search

- A symbol table is an abstract data type that associates a value with a key

- Primary operations:
  - Insert (put)
  - Search (get)

- An ordered symbol table is a symbol table in which the keys are Comparable objects (keys can be sorted)
Symbol table applications

<table>
<thead>
<tr>
<th>application</th>
<th>purpose of search</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary</td>
<td>find definition</td>
<td>word</td>
<td>definition</td>
</tr>
<tr>
<td>book index</td>
<td>find relevant pages</td>
<td>term</td>
<td>list of page numbers</td>
</tr>
<tr>
<td>file share</td>
<td>find song to download</td>
<td>name of song</td>
<td>computer ID</td>
</tr>
<tr>
<td>account management</td>
<td>process transactions</td>
<td>account number</td>
<td>transaction details</td>
</tr>
<tr>
<td>web search</td>
<td>find relevant web pages</td>
<td>keyword</td>
<td>list of page names</td>
</tr>
<tr>
<td>compiler</td>
<td>find type and value</td>
<td>variable name</td>
<td>type and value</td>
</tr>
</tbody>
</table>

Typical symbol-table applications
Elementary symbol tables

• Using unordered linked list
  – Sequential search: $O(N)$
  – Insert (replace old value if key exists): $O(N)$

• Using ordered array
  – Binary search: $O(\log N)$
  – Insert (need array resizing): $O(N)$
Binary search trees

- Generalized from linked list
  - Using two links per node

- Advantage
  - Fast to search and insert
  - Dynamic data structure

- Combines the flexibility of insertion in linked lists with the efficiency of search in an ordered array
Trees

• What is a tree?
  – **Nodes**: store data and links
  – **Edges**: links, typically directional

• The tree has a top node: **root node**

• The structure looks like reversed from real trees.
Terminology

- **D** is the left child of **B**.
- **B** is the parent of **D** and **E**.
- **E** is the right child of **B**.
- **A** is the root.
- The dashed line is a path.
- **F** is a subtree with **F** as its root.
- **C** is at level 1.
- **G** is at level 2.
- **H** is at level 3.
Terminology

• **Root**
  – The node at the top of the tree.
  – There is only one root.

• **Path**
  – The sequence of nodes traversed by traveling from the root to a particular node.
  – Each path is **unique**
Terminology

• Parent
  – The node that points to the current node.
  – Any node, except the root, has 1 and only 1 parent.

• Child
  – Nodes that are pointed to by the current node.

• Leaf
  – A node that has no children is called a leaf.
  – There can be many leaves in a tree.
Terminology

• **Interior node**
  – An interior node has at least one child.

• **Subtree**
  – Any node can be considered the root of a subtree.
  – It consists of all descendants of the current node.

• **Visit**
  – Checking the node value, display node value etc.

• **Traverse**
  – Visit all nodes in some specific order.
  – For example: visit all nodes in ascending key value.
Terminology

• **Levels**
  – The path length from root to the current node.
  – Recall that each path is unique.
  – Root is at level 0.

• **Height**
  – The maximum level in a tree.
  – $O(\log N)$ for a reasonably balanced tree.

• **Keys**
  – Each node stores a key value and associated data.
Binary search trees

**Definition.** A BST is a binary tree in symmetric order.

A binary tree is either:
- Empty.
- Two disjoint binary trees (left and right).

**Symmetric order.** Each node has a key, and every node’s key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.
BST representation in Java

Java definition. A BST is a reference to a root Node.

A Node is comprised of four fields:
• A Key and a Value.
• A reference to the left and right subtree.

```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node
    {
        /* see previous slide */
    }

    public void put(Key key, Value val)
    {
        /* see next slides */
    }

    public Value get(Key key)
    {
        /* see next slides */
    }

    public void delete(Key key)
    {
        /* see next slides */
    }

    public Iterable<Key> iterator()
    {
        /* see next slides */
    }
}

root of BST
Binary Search Trees

- Symbol table applications
- BST definitions and terminologies
- Search and insert
- Traversal
- Ordered operations
- Delete
BST Demo

• http://algs4.cs.princeton.edu/lectures/32Demo
  BinarySearchTree.mov
Get. Return value corresponding to given key, or null if no such key.
BST Search - Iterative

**Get.** Return value corresponding to given key, or `null` if no such key.

```java
public Value get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.
BST Search - Recursive

BST insert

Put. Associate value with key.

Search for key, then two cases:
- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.
BST Insert - Iterative

- [http://www.mathcs.emory.edu/~cs171000/share/code/BST_Iterative/BST.java](http://www.mathcs.emory.edu/~cs171000/share/code/BST_Iterative/BST.java)
BST Insert - Recursive

**Put.** Associate value with key.

```java
public void put(Key key, Value val) {
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else if (cmp == 0)
        x.val = val;
    return x;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.
BST trace: standard indexing client

key value
S 0
E 1
A 2
R 3
C 4
H 5
E 6
X 7

key value
A 8
M 9
P 10
L 11
E 12

black nodes are accessed in search
red nodes are new
grey nodes are untouched
changed value
Bonus Question

• Insert the following keys (in the order) into an empty BST tree
• Case 1
• Case 2
  – S, E, A, C, X, R, H
• Case 3
BST Analysis

- Search cost
- Insertion cost
Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.

**Remark.** Tree shape depends on order of insertion.
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.
**BSTs: mathematical analysis**

**Proposition.** If keys are inserted in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

**Pf.** 1-1 correspondence with quicksort partitioning.

**Proposition.** [Reed, 2003] If keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

**But...** Worst-case height is $N$.

(exponentially small chance when keys are inserted in random order)
Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1-1 if array has no duplicate keys.
### ST Implementations: Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
</tbody>
</table>
Binary Search Trees

- Symbol tables
- Definitions and terminologies
- Search and insert
- Traversal
- Ordered operations
- Delete
Traversals

- Traversal: visit all nodes in certain order

- In-order
  - Left subtree, current node, right subtree

- Pre-order
  - Current node, left subtree, right subtree

- Post-order
  - Left subtree, right subtree, current node
Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```java
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```

**Property.** Inorder traversal of a BST yields keys in ascending order.
Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

inorder(S)
inorder(E)
inorder(A)
enqueue A
inorder(C)
enqueue C
enqueue E
inorder(R)
inorder(H)
enqueue H
inorder(M)
enqueue M
enqueue R
enqueue S
inorder(X)
enqueue X

queue

function call stack
In-order
A C E H M R S X

Pre-order?

Post-order?
Traversal

• In-order
  A C E H M R S X

• Pre-order
  S E A C R H M X

• Post-order
  C A M H R E X S

• How to visit the nodes in descending order?

• What’s the use of pre-order and post-order traversal?
Expression Tree

- In-order traversal results in infix notation
- Post-order traversal results in postfix notation
- Pre-order traversal results in prefix notation

A tree that represents the expression

\[ 3 \times \left( \frac{7+1}{4} \right) + (17 - 5) \]

The upward pointing arrows show how the value of the expression can be computed.
Binary Search Trees

• Symbol table application
• Definitions and terminologies
• Search and insert
• Traversal

• Ordered operations
  – Minimum and maximum
  – Rank: how many keys < a given key?
  – Select: key of given rank

• Delete
Ordered operations

- Minimum and maximum
- Rank: how many keys < a given key?
- Select: key of a given rank k?
**Minimum and maximum**

**Minimum.** Smallest key in table.

**Maximum.** Largest key in table.

---

**Q.** How to find the min / max?
Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node. To implement `size()`, return the count at the root.

Remark. This facilitates efficient implementation of `rank()` and `select()`. 
private class Node
{
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int N;
}

public int size()
{
    return size(root);
}

private int size(Node x)
{
    if (x == null) return 0;
    return x.N;
}

private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val = val;
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
Rank

- Rank(Key key, Node x): how many keys < given key?
- Recursive algorithm
  - (Base) Case 1: tree is empty
  - (Base) Case 2: key == node key
  - Case 3: key < node key
  - Case 4: key > node key
public int rank(Key key) {
    return rank(key, root);
}

private int rank(Key key, Node x) {
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
Selection

Select. Key of given rank.

- Recursive algorithm
  - (Base) Case 1: tree is empty
  - (Base) Case 2: key in current node
  - Case 3: key in left tree
  - Case 4: key in right tree
**Selection**

**Select. Key of given rank.**

```java
public Key select(int k) {
    if (k < 0) return null;
    if (k >= size()) return null;
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k) {
    if (x == null) return null;
    int t = size(x.left);
    if (t > k)
        return select(x.left, k);
    else if (t < k)
        return select(x.right, k-t-1);
    else if (t == k)
        return x;
}
```
Binary Search Trees

• Definitions and terminologies
• Search and insert
• Traversal
• Ordered operations

• Delete
  – Delete minimum and maximum
  – Delete a given key
Delete minimum
Deleting the minimum

To delete the minimum key:
• Go left until finding a node with a null left link.
• Replace that node by its right link.
• Update subtree counts.

```java
public void deleteMin()
{
    root = deleteMin(root);
}

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```
Binary Search Trees

• Definitions and terminologies
• Search and insert
• Traversal
• Ordered operations
• Delete
  – Delete minimum and maximum
  – Delete a given key
Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 0. [0 children]
Hibbard deletion

To delete a node with key \( k \): search for node \( t \) containing key \( k \).

Case 0. [0 children] Delete \( t \) by setting parent link to null.
Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 1. [1 child]
Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

**Case 1.** [1 child] Delete $t$ by replacing parent link.
Hibbard deletion

To delete a node with key \( k \): search for node \( t \) containing key \( k \).

**Case 2.** [2 children]
Hibbard deletion

To delete a node with key \( k \): search for node \( t \) containing key \( k \).

Case 2. [2 children]
- Find successor \( x \) of \( t \).
- Delete the minimum in \( t \)'s right subtree.
- Put \( x \) in \( t \)'s spot.

\[ \xrightarrow{x \text{ has no left child}} \text{but don't garbage collect } x \]
\[ \xrightarrow{\text{still a BST}} \]
public void delete(Key key) {
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;

        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

Surprising consequence. Trees not random (!) $\Rightarrow$ $\sqrt{N}$ per op.
Longstanding open problem. Simple and efficient delete for BSTs.
### ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
</tbody>
</table>

*other operations also become √N if deletions allowed*
Hw5

- BST tree class (to be implemented)
  - BST tree node class
    - key (String type for short name)
    - data (MovieInfo type)
    - left and right children
  - root to the tree
  - methods
    - Insert()
    - findExact()
    - findPrefix()

- IndexTester (provided)
  - Creates an empty BST tree
  - Reads the input movies or actors files
  - Builds a MovieInfo object for each row, insert it into the BST tree
  - Asks for user search string and search for the MovieInfo object