Binary Search Trees

Outline and Required Reading:

• The Dictionary ADT  (§ 8.1)
• Binary Search Trees  (§ 9.1)
Dictionary ADT

**Dictionary** — searchable collection of \((k, e)\) pairs, also called “items”

- **\(k = \text{key}\)** — used to facilitate search for given item
- **\(e = \text{element}\)** — information of interest
- **examples:**
  a) **student records**
     - key = student’s ID number
     - element = student’s name, address, grades
  b) **phone book**
     - key = person’s name
     - element = person’s phone number

**Operations on Dictionaries** — dictionaries support following operations:

1. **searching**
2. **inserting**
3. **deleting**
Ordered Dictionaries — “order” relations are defined on the keys, so the relative order of any two keys can be determined, e.g. \( \text{key}(p) \leq \text{key}(r) \)

- examples: 
  a) student records
  \( \text{ID\_number\_1} > \text{ID\_number\_2} \)
  b) phone book
  \( \text{name\_1} > \text{name\_2} \)

Unordered Dictionaries — no “order” relation is assumed on the keys, and only equality testing between keys can be done

- example: 
  a) colour-based dictionary
  \( \text{item\_1} = (\text{blue}, \text{see}) \)
  \( \text{item\_2} = (\text{blue}, \text{sky}) \)
  \( \text{item\_3} = (\text{green}, \text{grass}) \)

In general, it is possible to have multiple items with the same key.
Dictionary ADT: Interface

**Interface Methods**

```java
public int size();  /* return the number of items in D */

public boolean isEmpty();  /* return true if D is empty */

public ObjectIterator elements();  /* return the elements stored in D */

public ObjectIterator keys();  /* return the keys stored in D */

public Object findElement(Object k);  /* if D contains item with key equal to k, then return */  /* element of such item, else return NO_SUCH_KEY */

public ObjectIterator findAllElements(Object k);  /* return an iterator of all elements with key equal to k */
```
public void insertItem(Object k, Object e);
/* insert an item with element e and key k into D */

public Object removeElement(Object k);
/* remove from D item with key equal to k, and return its */
/* element; if D has no such item, return NO_SUCH_KEY */

public ObjectIterator removeAllElements(Object k);
/* remove from D items with key equal to k, and return an */
/* iterator of their elements */

Dictionary ADT: Interface (cont.)
Map Interface in java.util.*
[ from: http://java.sun.com/j2se/1.3/docs/api/java/util/Map.html ]

Implementing Classes:  MapTree, Hashtable, WeakHashMap, etc.

- “Map” is an object that maps keys to values.
- A map cannot contain duplicate keys; each key can map to at most one value.
- Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

The Map.Entry Interface

- The Map interface contains a static nested interface called Entry.
- Map.Entry is used to access key-value pairs as individual objects outside of the Map.

Key methods of the Map.Entry interface:
  - Object getKey() - Returns the key portion of this Entry.
  - Object getValue() - Returns the value portion of this Entry.
  - Object setValue(Object value) - Replaces the value portion of this Entry with the parameter. The replaced value is returned.
### Map Interface - Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean containsKey(Object key)</code></td>
<td>Returns <code>true</code> if this map contains a mapping for the specified key.</td>
</tr>
<tr>
<td><code>boolean containsValue(Object value)</code></td>
<td>Returns <code>true</code> if this map maps one or more keys to the specified value.</td>
</tr>
<tr>
<td><code>boolean equals(Object o)</code></td>
<td>Compares the specified object with this map for equality.</td>
</tr>
<tr>
<td><code>Object get(Object key)</code></td>
<td>Returns the value to which this map maps the specified key.</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>Returns <code>true</code> if this map contains no key-value mappings.</td>
</tr>
<tr>
<td><code>Object put(Object key, Object value)</code></td>
<td>Associates the specified value with the specified key in this map (optional operation).</td>
</tr>
<tr>
<td><code>Object remove(Object key)</code></td>
<td>Removes the mapping for this key from this map if present (optional operation).</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>Returns the number of key-value mappings in this map.</td>
</tr>
</tbody>
</table>
**Binary Search Trees**

**Binary Search Tree** — binary tree $T$ such that each internal node $v$ of $T$ stores an item $(k, e)$ of a dictionary $D$, and

- all keys stored at nodes in left subtree of $v$ are less than $k = \text{key}(v)$
- all keys stored at nodes in right subtree of $v$ are greater than or equal to $k = \text{key}(v)$

\[
\text{key}(v.\text{leftChild}) < \text{key}(v) \leq \text{key}(v.\text{rightChild})
\]

This is a BST! This is NOT a BST!
Example 1 [database of student records as a BST]

- key: used to facilitate ordering, searching, inserting in the BST
- element: information of users’ interest – irrelevant from the BST point of view
Properties of Binary Search Trees

- external nodes are Null_Nodes – they do not store any key or element of D !!! (used for coding simplicity)
- the leftmost item has the smallest key
- the rightmost item has the largest key
- an inorder traversal of a binary search tree visits the keys in non-decreasing order !!!

Schedule of Inorder Visits: 5, 17, 32, 44, 48, 51, 56
Other Properties of Binary Search Trees

- All keys greater than a given key \( k \) are in
  1. the right subtree of node \( p \) associated with \( k \), and
  2. the right subtrees of all right-ancestors of \( p \)

- All keys smaller than a given key \( k \) are in
  1. the left subtree of node \( p \) associated with \( k \), and
  2. the left subtrees of all left-ancestors of \( p \)
Quiz Question:

Assume the reference to an arbitrary node (v) in a BST tree is given. Propose an algorithm for finding the difference between the key stored in this node and the smallest key in the tree. What is the worst case complexity of your algorithm?
Example 2  [ finding smaller/greater keys in a BST ]

In the given BST find all keys
(a) greater than $p=20$;
(b) smaller than $q=55$. 
Binary Search Trees (cont.)

Example 3  [ verifying BST ]

Is the following tree a BST?
Example 4  [ storing linked list / array in BST ]

Suppose the numbers in a linked list / array are: 11, 5, 8, 16, 4, 19, 17. Store them in a BST.
Binary Search Trees (cont.)

Is the procedure for storing an unordered set of keys (elements) in a BST unique?!

Suppose the set contains: 11, 5, 8, 16, 4, 19, 17.
These numbers can be stored in a BST in either of the following ways ...

Advantages: ... ???
Disadvantages: ... ???
Example 7  [ Would “insertion in the middle” of a BST work??? ]

Suppose we have to insert key=5 in the following BST.

6 ends up in the left subtree of 5.  
WRONG!

BST property preserved everywhere.  
OK!
**Binary Search Trees (cont.)**

**Motivation for Use of BST**

- BST enable efficient search or counting the number of appearances of a particular key/element
  - search / counting method recursively proceeds in one subtree only
  - in the case of arrays and linked lists:
    1. if we search for an element – all elements might need to be examined
    2. if we count the appearance of an element – all elements need to be examined

**Example 5** [ search for number 27 in a BST ]

```
27 < 44, look left of 44
17 44
   56

27 > 17, look right of 17
17 44
   56

null node reached, 27 not in T!
17 44
   56
```
Binary Search Trees – Implementation

**Item Class** – contains key-element pairs – instances of this class get stored in the positions (nodes) of the underlying BST

```java
public class Item {
    private Object key, elem;
    protected Item (Object k, Object e){
        key = k;
        elem = e; }
    public Object key() { return key; }
    public Object element() { return elem; }
    public void setKey(Object k) { key = k; }
    public void setElement(Object e) { elem = k; }
}
```
Binary Search Trees – Implementation  (cont.)

BinarySearchTree Class

```java
public class BinarySearchTree implements Dictionary {
    Comparator C;    /* contains set of rules for comparing keys */
    BinaryTree T;    /* underlying binary tree */
    ...
    public BinarySearchTree(Comparator c) {
        C = c;
        T = new BinaryTree(); }
    ...
    public Position findPosition(Object key, Position p) { ... }  
    public void insertItem(Object k, Object e) { ... }
    public Object removeElement(Object k) { ... }
}
```
BST – Implementation: Search

**findPosition** Method — belongs to BinarySearchTree class!
- returns a node with a specific key, or
  the last external node found

```java
public Position findPosition(Object key, Position pos) {
    if (T.isExternal(pos)) { return pos; }; /* terminate */
    else {
        Object curKey = key(pos);
        if (C.isLessThan(key, curKey)) {
            return findPosition(key, T.leftChild(pos));
        } else if (C.isGreaterThan(key, curKey)) {
            return findPosition(key, T.rightChild(pos));
        } else return pos;
    }
}
```
Performance of findPosition for Tree of Height h

- method starts at the root and goes down 1 level at the time
- \( O(1) \) time spent per node \( \Rightarrow h+1=O(h) \) is the worst-case RT

General Performance of findPosition

\[
\frac{\log_2(n+1)-1}{n+1} \leq h \leq \frac{n-1}{2}
\]

\[\Rightarrow\quad O(\log(n)) \leq \text{worst case RT} \leq O(n)\]

- full binary tree
- unbalanced tree

when \( n \) is known, but the "shape" of the tree is unknown
Example 6  [ determining complexity of Binary Search using BST ]

BS walk through an array =
= findPosition in corresponding balanced BST

balanced BST  ⇒  h ≈ log₂(n+1)-1

Thus, O(h) = O(log₂(n)) is the worst case RT of the BS algorithm on a sorted array.
BST – Implementation: Insertion

Dictionary with Unique Keys – no two elements can have the same key

Dictionary with Duplicate Keys – two elements can have the same key

Node Insertion in BST with Unique Keys

• procedure for inserting item \((k,e)\):

  1. call method \(\text{treeSearch}(k, T.\text{root}())\) (i.e. \(\text{findPosition}(k, T.\text{root}())\)) and let \(w\) be the position returned by this method
  2. \(\text{expandExternal}(w)\), such that the internal node now contains \((k,e)\)
procedure for inserting item \((k,e)\):

1. call method treeSearch\((k,T.root())\)
   and let \(w\) be an internal node
   returned

2. recursively apply treeSearch\((k,w)\)
   on right child of \(w\) until an external
   node is reached

3. expandExternal on the last
   external node as in the case of
   insertion with unique keys

Performance of BST Insertion

(2) assumes \(\text{key}(v_{\text{leftChild}}) < \text{key}(v) \leq \text{key}(v_{\text{rightChild}})\), hence:

\[
\text{worst case performance} \in O(h)
\]
Otherwise, if we allow \( \text{key}(v.\text{leftChild}) \leq \text{key}(v) \leq \text{key}(v.\text{rightChild}) \), then in the following special case:

```
key=2

                     2
                    / \
                   2   2
                  /       \
                 2   2   2
```

worst case RT of insert = \( O[ ] \) ??
public void insertItem(Object k, Object e) throws InvalidKeyException {
    checkKey(k);  /* check whether the given key is valid */
    Position insPos = T.root();
    do {
        insPos = findPosition(k, insPos);
        if (T.isExternal(insPos)) break;         /* terminate do-while loop */
        else insPos = T.rightChild(insPos);
    } while (true)
    T.expandExternal(insPos);
    Item newItem = new Item(k, e);  /* (k,e) pair - object*/
    T.replaceElement(insPos, newItem);
}
**BST – Implementation: Deletion**

Node Deletion in BST – more complex operation than insertion, since we must adequately “fill” any “holes” that may appear – the BST property must be preserved! Two special cases should be considered when inserting item \( w = (k, e) \):

- **Case A** one child of \( w \) (\( z \)) is an external node
- **Case B** both children of \( w \) are internal nodes

Node Deletion: Case A – simply remove \( w \) and \( z \) from \( T \) by means of operation removeAboveExternal(\( z \)); replace \( w \) with the sibling of \( z \)
BST – Implementation: Deletion (cont.)

Node Deletion: Case B

(1) find node \( y \) with smallest greater key than \( w \)‘s key; \( y \) is, in fact, the left-most internal node in the right subtree of \( w \)

(2) copy the content of \( y \) into \( w \)

(3) remove \( y \) and its leaf-node \( x \) by means of \( \text{removeAboveExternal}(x) \)

NOTE: \( x \) (i.e. \( y \)) immediately follows \( w \) in the inorder traversal of \( T \) – the BST property preserved!

Performance of BST Deletion

– in both cases, \( \text{worst case RT} \in O(h) \)
public Object removeElement(Object k) throws InvalidKeyException {
    Object toReturn;
    checkKey(k); /* check whether the given key is valid */
    Position remPos = findPosition(k, T.root());
    if(T.isExternal(remPos)) { return NO_SUCH_KEY; } /* unsucc. search */
    else {
        toReturn = remPos.element();
        if (T.isExternal(T.leftChild(remPos)))
            remPos=T.leftChild(remPos);
        else if (T.isExternal(T.rightChild(remPos)))
            remPos=T.rightChild(remPos);
        /* final else - boh children of remPos internal */
else {
    Position swapPos = remPos;
    remPos = T.rightChild(swapPos);
    do {
        remPos = T.leftChild(remPos);
    } while (T.isInternal(remPos));
    swap(swapPos, T.parent(remPos));
    t.removeAboveExternal(remPos)
    return toReturn;
} } }
### BST: Performance

#### Linked Structure Implementation

- space complexity: $O(n)$
- running times:

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>findElement, insertElement, removeElement</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>findAllElements, removeAllElements</td>
<td>$O(h+s)$</td>
</tr>
</tbody>
</table>

S – size of iterators returned by these methods

In general, BST is expected to provide better performance than any linear (sequential) dictionary.

However, in the worst case (e.g. highly unbalanced tree) BST performance is still linear, since $h=(n-1)/2$ !!!
Binary Search Trees: Questions

Q.1 In Case 2 of BST deletion, could node w be simply replaced with one of its children? Why are we looking for the left-most element in the right sub-tree? Is there any other candidate element to replace w? Explain!

Q.2 Write a utility function for finding the maximum element in a (sub)tree?

Q.3 The following items are to be placed in a binary search tree:
   27, 8, 35, 42, 5, 19, 40, 9, 12, 16, 27, 2.
   a) Construct a binary search tree (in diagram form) for the above items in the order given.
   b) How many comparisons would you require to locate the number 40 in your search tree?