Computer Science 331

Binary Search Trees: Definition and Searching

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #12

Outline

The Dictionary ADT

A *dictionary* is a finite set (no duplicates) of elements.

Each element is assumed to include

- A **key**, used for searches.
  - Keys are required to belong to some ordered set.
  - The keys of the elements of a dictionary are required to be distinct.
- Additional **data**, used for other processing.

Examples:

- a dictionary (word is the key, definition is the data)
- telephone book (name is the key, phone number is the data)

Similar to Java's `Map` (unordered) and `SortedMap` (ordered) interfaces.

Operations Supported

A dictionary must support the following operations:

- `search(k)`: Search for and return a reference to an element with key `k` if one exists. Throw a `KeyNotFoundException` if there is no such element.
- `insert(k,x)`: Add the element `x` with key `k` if no element with the same key is included already. Throw a `KeyFoundException` if an element with the same key already belongs to the set.
- `delete(k)`: Remove the element with key `k`. Throw a `KeyNotFoundException` if no such element is in the dictionary

May also include `isEmpty()`, `size()`

Linked lists and arrays are two data structures that can be used to implement dictionaries. *Binary Search Trees* are another (subject of today's lecture).

Binary Tree

A **binary tree** T is a hierarchical, recursively defined data structure, consisting of a set of **vertices** or **nodes**.

A binary tree T is **either**

- an "empty tree,"

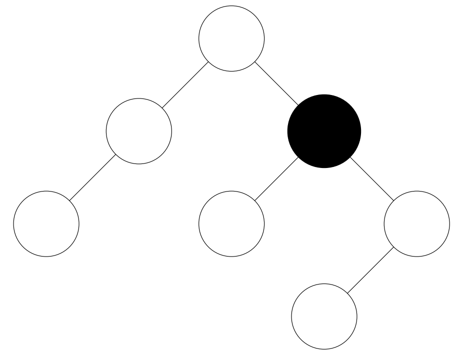
or

- a structure that includes
  - the **root** of T (the node at the top)
  - the **left subtree** T_L of T ...
  - the **right subtree** T_R of T ...

... where both T_L and T_R are also binary trees.

Example and Implementation Details

Example:



Each node has a:

- parent:
- left child:
- right child:

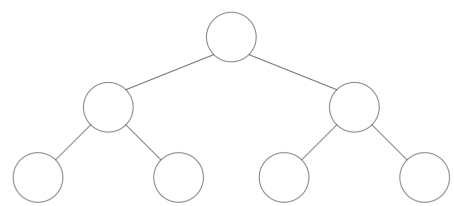
Additional Terminology

Additional terms related to binary trees:

- **siblings**:
- **descendant (of N)**:
- **ancestor (of N)**:
- **leaf**:
- **size**:
- **depth (of N)**:
- **height**:

Note: depth and height are sometimes (as in the text) defined in terms of number of nodes as opposed to number of edges.

Size vs. Depth: One Extreme

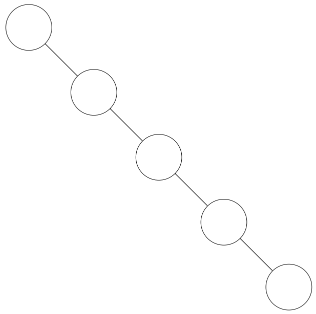


- Size:
- Height:
- Relationship:

This binary tree is said to be *full*:

- all leaves have the same depth
- all non-leaf nodes have exactly two children

Size vs. Depth: Another Extreme



- Size:
- Height:
- Relationship:

This binary tree is essentially a linked list.

Binary Search Tree

A **binary search tree** T is a data structure that can be used to implement a dictionary.

- T is a binary tree
- Each element of the dictionary is stored at a node of T, so

$$\text{dictionary size} = \text{size of } T$$

- In order to support efficient searching, elements are arranged to satisfy the **Binary Search Tree Property** . . .

Binary Search Tree Property

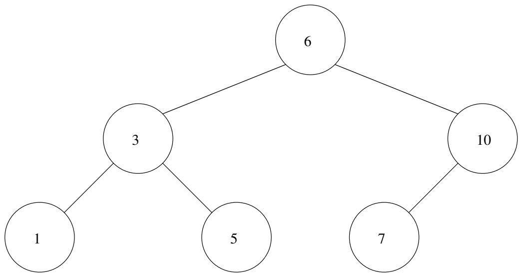
Binary Search Tree Property: If T is nonempty, then

- The left subtree T_L is a binary search tree including all dictionary elements whose keys are *less than* the key of the element at the root
- The right subtree T_R is a binary search tree including all dictionary elements whose keys are *greater than* the key of the element at the root

Example

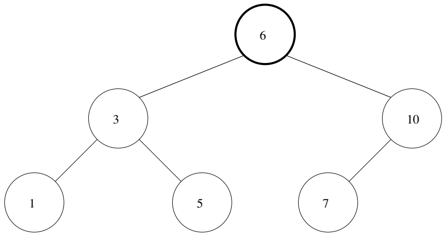
One binary search tree for a dictionary including elements with keys

$$\{1, 3, 5, 6, 7, 10\}$$



Searching: An Example

Searching for 5:



Nodes Visited:

- Start at 6 :
- Next node
- Next node

Binary Search Tree Data Structure

```
public class BST<E extends Comparable<E>,V> {
  protected bstNode<E,V> root;
  ...

  protected class bstNode<E,V> {
    E key;
    V value;
    bstNode<E,V> left;
    bstNode<E,V> right;
    ...
  }
}
```

`bstNode` can also include a reference to its parent

A Recursive Search Algorithm

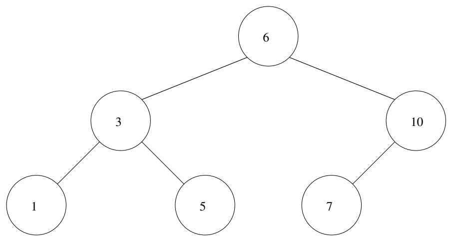
```
public V search(bstNode<E,V> T, E key)
    throws KeyNotFoundException {
  if (T == null)

  else if (key.compareTo(T.key) == 0)

  else if (key.compareTo(T.key) < 0)

  else

}
```


Analysis: Correctness and Running Time

Partial Correctness (tree of height h):

Termination and Bound on Running Time (tree of height h):

Minimum Finding: The Idea



Idea:

-
-

Example:

A Recursive Minimum-Finding Algorithm

```
// Precondition:  T is non-null
// Postcondition:  returns node with minimal key,
//   null if T is empty

public bstNode<E,V> findMin(bstNode<E,V> T) {
  if (T == null)

  else if (T.left == null)

  else

}
```


Analysis: Correctness and Running Time

Partial Correctness (tree of height h):

- Exercise (similar to proof for `Search`)

Termination and Bound on Running Time (tree of height h):

- worst case running time is $\Theta(h)$ (and hence $\Theta(n)$)
- Proof: exercise

Next lecture...

Think about how to do

- insertion (hint: modify `search`)
- deletion (four separate cases need to be handled)

Key: inserting/deleting in such a way that the resulting tree still satisfies the BST property.