Computer Science 331
Binary Search Trees

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #14–15

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.

Permits the following operations:

- search by key
- insert (key/data pair)
- delete an element with specified key

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

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,"

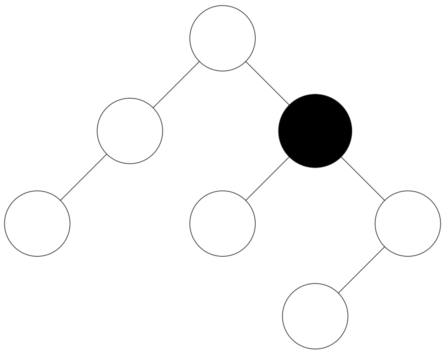
or

- 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: unique node above a given node
- left child: node in left subtree directly below a given node (root of left subtree)
- right child: node in right subtree directly below a given node (root of right subtree)

Each of these may be `null`

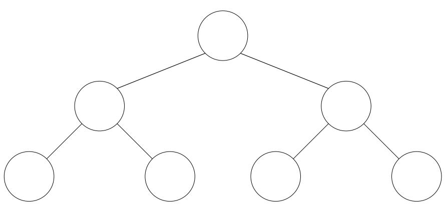
Additional Terminology

Additional terms related to binary trees:

- **siblings**: two nodes with the same parent
- **descendant (of N)**: any node occurring in the tree with root `N`
- **ancestor (of N)**: root of any tree containing node `N`
- **leaf**: node with no children
- **size**: number of nodes in the tree
- **depth (of N)**: length (# of edges) of path from the root to `N`
- **height**: length of longest path from root to a leaf ($height(emptytree) = -1$)

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. Height: One Extreme



- Size: 7
- Height: 2
- Relationship:

$$n = 1 + 2 + 4 = \sum_{i=0}^{h} 2^i$$

$$= 2^{h+1} - 1,$$

and

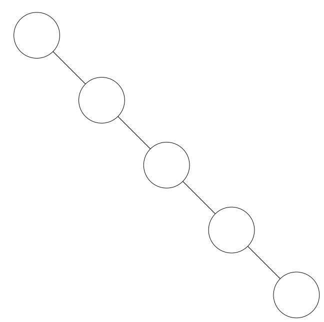
$$h = \log_2(n+1) - 1$$

This binary tree is said to be *full*:

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

Upper bound: a binary tree of height h has size *at most* $2^{h+1} - 1$.

Size vs. Height: Another Extreme



- Size: 5
- Height: 4
- Relationship: $n = h + 1$

Essentially a linked list!

Lower bound: a binary tree with height h has size *at least* $h + 1$.

Binary Search Tree

A **binary search tree** T is a data structure that can be used to store and manipulate a finite ordered set or mapping.

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

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

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

Binary Search Tree Property

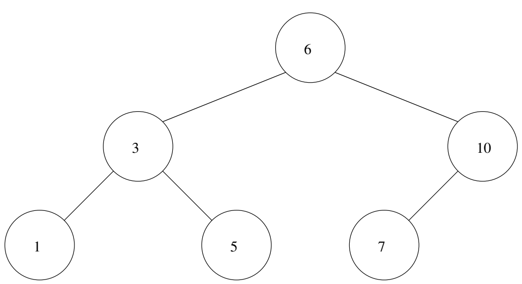
Binary Search Tree Property: If T is nonempty, then

- 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

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



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

Specification of "Search" Problem:

Precondition 1:

a) T is a BST storing values of some type V along with keys of type E

b) key is an element of type E stored with a value of type V in T

Postcondition 1:

a) Value returned is (a reference to) the value in T with key key

b) T and key are not changed

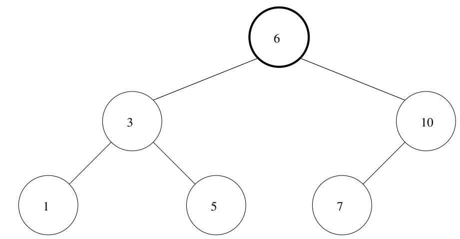
Precondition 2: same, but key is not in T
Postcondition 2:

a) A notFoundException is thrown

b) T and key are not changed

Searching: An Example

Searching for 5:



Nodes Visited:

- Start at 6 : since $5 < 6$, search in left subtree
- Next node 3 : since $5 > 3$, search in right subtree
- Next node 5 : equal to key, so we're finished

A Recursive Search Algorithm

```
public V search(bstNode<E,V> T, E key)
    throws notFoundException {
  if (T == null)
    throw new notFoundException();
  else if (key.compareTo(T.key) == 0)
    return T.value;
  else if (key.compareTo(T.key) < 0)
    return search(T.left, key);
  else
    return search(T.right, key);
}
```


Partial Correctness

Proved by induction on the height of T:

1. Base cases are correct (easy: height -1 or 0)
2. Assume that the algorithm is partially correct for all trees of height $\leq h-1$. By the BST property:
   - if key == root.key, correctness of output is clear by inspection of the code
   - otherwise, by the BST property:
     - if key < root.key, it is in the left subtree (or not in the tree)
     - otherwise key > key.root and it must be in the right subtree (or not in the tree)

   In either case, algorithm is called recursively on a subtree of height at most $h-1$ and outputs correct result by assumption

Termination and Running Time

Let Steps(T) be the number of steps used to search in a BST T in the worst case. Then there are positive constants c_1, c_2 and c_3 such that

$$\mathtt{Steps(T)} \leq \begin{cases} c_1 & \text{if height}(\mathtt{T}) = -1, \\ c_2 & \text{if height}(\mathtt{T}) = 0, \\ c_3 + \max(\text{Steps}(\mathtt{T.left}), \text{Steps}(\mathtt{T.right})) & \\ & \text{if height}(\mathtt{T}) > 0. \end{cases}$$

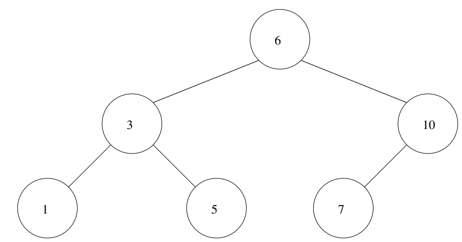
Exercise: Use this to prove that

$$\text{Steps}(\mathtt{T}) \leq c_3 \times \text{height}(\mathtt{T}) + \max(c_1, c_2)$$

Exercise: Prove that $\text{Steps}(\mathtt{T}) \geq \text{height}(\mathtt{T})$ as well.

$\Longrightarrow$ The worst-case cost to search in T is in $\Theta(\text{height}(\mathtt{T}))$.

Minimum Finding: The Idea



Idea: value in a node is the minimum if the node has no left child

- recursively (or iteratively) visit left children
- first node with no left child encountered contains the minimum key

Example: minimum is 1

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)
    return null;
  else if (T.left == null)
    return T;
  else
    return findMin(T.left);
}
```


Analysis: Correctness and Running Time

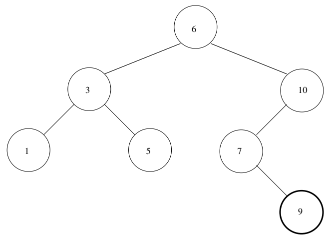
Partial Correctness (tree of height h):

- Exercise (similar to proof for `Search`)

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

- after each recursive call, the height is reduced by at least 1
- worst case running time is $\Theta(h)$ (and hence $\Theta(n)$)

Insertion: An Example



Idea: use `search` to find empty subtree where node should be

Nodes Visited (inserting 9):

- Start at 6 : since $9 > 6$, new node belongs in right subtree
- Next node 10 : since $9 < 10$, new node belongs in left subtree
- Next node 7 : since $9 > 7$, new node belongs in right subtree
- Next node `null`: insert new node at this point

A Recursive Insertion Algorithm

```
// Non-recursive public function calls recursive worker function
public void insert(E key, V value)
  { root = insert(root, key, Value); }

protected
bstNode<E,V> insert(bstNode<E,V> T, E newKey, V newValue) {
  if (T == null)
    T = new bstNode<E,V>(newKey,newValue,null,null);
  else if (newKey.compareTo(T.key) < 0)
    T.left = insert(T.left, newKey, newValue);
  else if (newKey.compareTo(T.key) > 0)
    T.right = insert(T.right, newKey, newValue);
  else
    throw new FoundException();
  return T;
}
```


Analysis: Correctness and Running Time

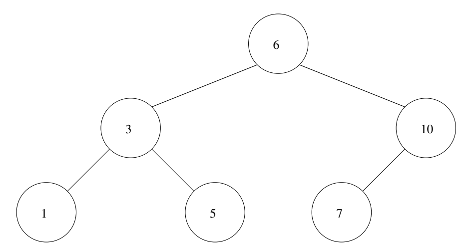
Partial Correctness (tree of height h):

- 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

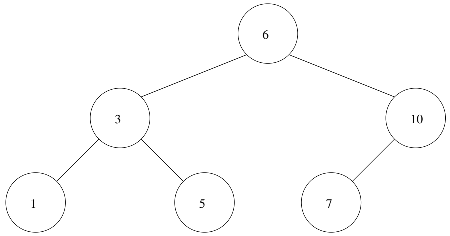
Deletion: Four Important Cases



Key is/has ...

1. Not Found (Eg: Delete 8)
2. At a Leaf (Eg: Delete 7)
3. One Child (Eg: Delete 10)
4. Two Children (Eg: Delete 6)

First Case: Key Not Found



Idea: search for key 8, throw `notFoundException` when not found

Nodes Visited (delete 8):

- Start at 6 : since $8 > 6$, delete 8 from right subtree
- Next node 10 : since $8 < 10$, delete 8 from left subtree
- Next node 7 : since $8 > 7$, delete 8 from right subtree
- Next node `null`: conclude that 8 is not in the tree

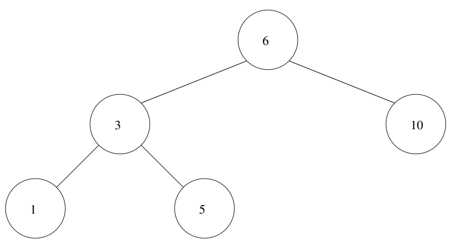
Algorithm and Analysis

```
protected bstNode<E,V> delete(bstNode<E,V> T, E key) {
  if (T != null) {
    if (key.compareTo(T.key) < 0)
      T.left =  delete(T.left, key);
    else if (key.compareTo(T..key) > 0)
      T.right = delete(T.right,key);
    else if ...
      // found node with given key
  }
  else
    throw new notFoundException();
  return T;
}
```

Correctness and Efficiency For This Case:

- tree is not modified if key is not found (base case will be reached)
- worst-case cost $\Theta(h)$ (same as `search`)

Second Case: Key is at a Leaf



Idea: set appropriate pointer in parent to `null`

Nodes Visited (delete 7):

- Start at 6 : since $7 > 6$, delete 7 from right subtree
- Next node 10 : since $7 < 10$, delete 7 from left subtree
- Next node 7 : set pointer to left child of parent to `null`

Algorithm and Analysis

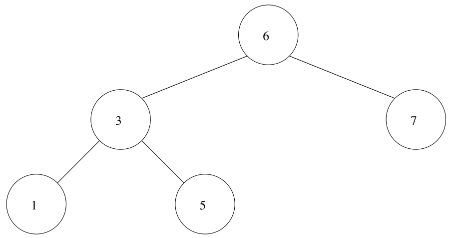
Extension of Algorithm:

```
else if (T.left == null && T.right == null)
  T = null;
```

Correctness and Efficiency For This Case:

- test detects whether the node is a leaf
- replacing `T` with `null` deletes the leaf at `T`
- removing a leaf does not affect BST property
- worst-case cost is $\Theta(h)$ for this case ($\Theta(h)$ to locate leaf, $\Theta(1)$ to remove it)

Third Case: Key is at a Node with One Child



Idea: remove node, put the one subtree in its place

Nodes Visited (delete 10):

- Start at 6 : since 10 > 6, delete 10 from right subtree
- Next node 10 : set pointer to right child of parent to child of 10

Algorithm and Analysis

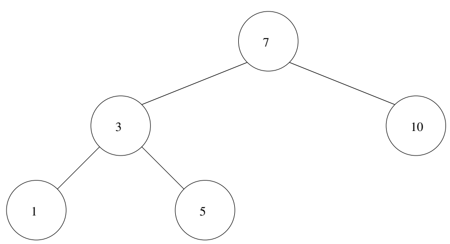
Extension of Algorithm:

```
else if (T.left == null)
  T = T.right;
else if (T.right == null)
  T = T.left;
```

Correctness and Efficiency For This Case:

- `T` is replaced with its one non-empty subtree
  - node originally at `T` is deleted
  - BST property still holds (new subtree at `T` still contains keys that were in the old subtree)
- worst case cost is $\Theta(h)$ ($\Theta(h)$ to locate node, $\Theta(1)$ to remove it)

Fourth Case: Key is at a Node with Two Children



Idea: replace node with its successor (minimum in the right subtree)

Nodes Visited (delete 6):

- Start at 6 : found node to delete
- replace data at node with data from the node of minimum key in the right subtree
- delete node with minimal key from the right subtree

Algorithm and Analysis

Extension of Algorithm:

```
else {
  bstNode<E,V> min = findMin(T.right);
  T.key = min.key; T.value = min.value;
  T.right = delete(T.right, T.key);
}
```

Correctness and Efficiency For This Case:

- BST property holds: all entries in the new right subtree have keys > the smallest key from the original right subtree
- worst case cost is $\Theta(h)$:
  - `findMin` costs $\Theta(h)$ (from last lecture)
  - recursive call deletes a node with at most one child from a tree of height $< h$ (cost is $\Theta(h)$)

More on Worst Case

All primitive operations (`search`, `insert`, `delete`) have worst-case complexity $\Theta(n)$

- all nodes have exactly one child (i.e., tree only has one leaf)
- Eg. will occur if elements are inserted into the tree in ascending (or descending) order

On average, the complexity is $\Theta(\log n)$

- Eg. if the tree is full, the height of the tree is $h = \log_2(n+1) - 1$
- the height of a randomly constructed tree (inserting n elements uniformly randomly) is $3 \log_2 n$ for sufficiently large n (see lecture supplement)

Need techniques to ensure that all trees are close to full

- want $h \in \Theta(\log n)$ in the worst case
- one possibility: red-black trees (next three lectures)

References

Data Structures & Algorithms in Java (Lafore)

- Chapter 8 Discusses in more detail, including algorithms for tree traversals