CS250 Winter 2003

Week 9, Lecture 1:
Binary search trees (BST’s)
Binary search trees

Search trees are data structures that support operations that include search, minimum, predecessor, successor, insert, and delete. A search tree can be used to implement a dictionary or a priority queue.

A dictionary is an ADT for storing elements with operations insert, delete, and test membership in the dictionary.

A priority queue is an ADT for storing elements with a key, with operations to insert as well as to have access to the element with the largest key.
Definition of BST

A binary search tree is organized in a binary tree. A binary tree is a tree where each node has at most 2 children. A binary tree is said to be full if every internal node has two children. Note that the text calls this “complete”, but it is not the most commonly accepted name, and will mean something else in Lecture 9-2.

Such a tree can be represented by a linked data structure, where each node contains a key field and pointers to the left and right children and the parent node. If a child or the parent is missing, then the pointer is nil.

The keys in a BST satisfy the BST property:

<table>
<thead>
<tr>
<th>Let ( x ) be a node. If ( y ) is a node in the left subtree of ( x ) then ( key[y] \leq key[x] ). If ( y ) is a node in the right subtree of ( x ) then ( key[x] \leq key[y] ).</th>
</tr>
</thead>
</table>
Application of tree traversal from Lecture 8-1

The BST property allows one to print out all the keys in a BST by a simple recursive algorithm, called *inorder tree traversal*.

```c
inorderTreeTraversal(x) {
    if x != nil {
        inorderTreeTraversal(left[x])
        print key[x]
        inorderTreeTraversal(right[x])
    }
}
```

This algorithm runs in time $\Theta(n)$, where $n$ is the number of nodes in the tree rooted at $x$.  

Example of tree traversal

The inorder tree traversal prints the keys in each of these two BST in the order 2, 3, 5, 5, 7, 8.
Querying a BST

The most common operation performed on a BST is searching for a key stored in the tree. Besides the search query, a BST supports minimum, maximum, predecessor, and successor queries.

All query operations on a BST can be supported in time $O(h)$ for a BST of height $h$.

What is the height $h$, in terms of the number of nodes $n$, for a binary tree?
Querying a BST: Searching

Given a pointer to the root \( x \) of a BST, find a target key \( k \).

\[
\text{BSTsearch}(x, k) \{
    \text{if } x = \text{nil or } k = \text{key}[x]
    \quad \text{return } x
    \text{if } k < \text{key}[x]
    \quad \text{return BSTsearch(left[x], k)
    \quad \text{else}
    \quad \quad \text{return BSTsearch(right[x], k)
    \}
\]

On most computers, the iterative version of this algorithm is more efficient. Why?
Querying a BST: Minimum and maximum

The element with minimum key is found by following the left child pointers from the root, until a nil is found. This is guaranteed by the BST property. Can you give a full proof?

```plaintext
BSTminimum(x) {
    while left[x] != nil
        x = left[x]
    return x
}
```

Symmetrically for the maximum key:

```plaintext
BSTmaximum(x) {
    while right[x] != nil
        x = right[x]
    return x
}
```
Querying a BST: Predecessor and successor

Goal: to find the predecessor or the successor of a node \( x \) in the (sorted) order determined by an inorder tree traversal.

Algorithm idea for successor: the next larger key is the minimum of right[\( x \)] if it is nonempty; otherwise if \( x \) has a successor, it is the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \).

Why? If \( x \) has a successor and right[\( x \)] = nil, then the closest “\( \leq \)” relation in the BST comes from the closest left child relation among the ancestors. E.g. leaf node 5 (p.4, first tree)’s successor is root node 5.

Leftover question: if right[\( x \)] \( \neq \) nil, why can’t the successor of \( x \) be the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \)?
Successor: pseudo-code

```plaintext
BSTsuccessor(x) {
    if right[x] != nil
        return BSTminimum(right[x])

    y = parent[x]
    // found key[y] >= key[x] if x = left[y]
    while y != nil and x = right[y] {
        // lookup the next ancestor
        x = y
        y = parent[x]
    }
    return y
}
```

The algorithm for BSTpredecessor is symmetric.
Insertion and deletion in a BST

BSTinsertion

Goal: insert node $z$, associated with key[$z$], into the tree rooted at $T$.

Algorithm idea: do a BST search of $k = \text{key}[$z$]$ in $T$ to find where it should sit, while keeping track of the parent of the currently explored node $x$.

Pseudo-code on the next slide...

BSTdeletion

Can you figure it out?

BSTinsertion and BSTdeletion can be made to run in time $O(h)$. 
Insertion pseudo-code

```
BSTinsertion(T, z) {
    y = nil  // remembers parent[x]
    x = root[T]

    while x != nil {
        y = x  // go down the BST
        if key[z] <= key[x]
            x = left[x]
        else
            x = right[x]
    }
    p[z] = y  // the leaf that parents z is y

    // insert z as correct child of found leaf y
    if y = nil  // T was empty (T = nil as input)
        root[T] = z
    else if key[z] < key[y]
        left[y] = z
    else
        right[y] = z
}
```
Deletion in a BST

Goal: in a BST $T$, delete a node $z$.

Algorithm idea: consider all cases for $z$. What if $z$ has no children? Easy: delete $z$. What if it has only one child? Not too hard: delete $z$ and link its child with its parent. (In this last case, why it the resulting tree still a BST?)

What if $z$ has two children?

Hint: consider the successor of $z$. Among other things, one can prove that the successor of a node with two children (such as $z$) has no left child. Can you prove that if the key of successor[$z$] replaced the one of $z$, the resulting tree would still be a BST?
Deletion pseudo-code

BSTdeletion(T, z) {

    // have y point to node to be taken out
    if left[z] = nil or right[z] = nil
        y = z
    else
        y = BSTsuccessor(z)

    // set x to non-nil child of y (to nil o.w.)
    // NB: if y = successor(z), y has <= one child
    if left[y] != nil
        x = left[y]
    else
        x = right[y]

    // take out y (may it be z or its successor)
    // if y has a child, make its parent y’s
    if x != nil
        parent[x] = parent [y]
// if y was the root, x becomes the new root
if parent[y] = nil
    root[T] = x
else if y = left[parent[y]]  // y=left child
    left[parent[y]] = x
else  // y=right child
    right[parent[y]] = x

// if y = successor(z), then copy the
// contents of y into z
if y != z
    key[z] = key[y]
}