MCS-375: Algorithms: Analysis and Design Handout #DS3
San Skulrattanakulchai
Gustavus Adolphus College Sep 19, 2014

Binary Search Trees

CLRS: Ch 12.1–12.3

- A *dynamic set* is a collection of *items* that can change over time as new items are inserted and old ones are deleted. An item has a *key* field and other fields called *satellite data*.
- A *dictionary* ADT is a dynamic set that supports the following three operations: searching for an item, inserting a new item, and deleting an existing item. A classic application requiring a dictionary is maintaining the symbol table in a compiler.
- A *total order* is a set S with a binary relation $\prec$ on S that is areflexive and transitive, and that any two elements in the set are comparable, i.e., given $s, s' \in S$, exactly one of the folloiwng three relations holds: $s = s'$, $s \prec s'$, or $s' \prec s$.
- These are some operations on totally-ordered dynamic set `S` (also called sorted set):
  - SEARCH(`S`, `k`): return the node `x` in `S` with `x.key`=`k`, or `NIL` if no such node
  - INSERT(`S`, `x`): add node `x` to set `S`
  - DELETE(`S`, `x`): delete node `x` from set `S`
  - MINIMUM(`S`): returns the node in `S` containing the minimum key
  - MAXIMUM(`S`): returns the node in `S` containing the maximum key
  - SUCCESSOR(`S`, `x`): returns the node in `S` that immediately follows node `x` in the inorder traversal of `S`, or `NIL` if no such node
  - PREDECESSOR(`S`, `x`) returns the node in `S` that immediately precedes node `x` in the inorder traversal of `S`, or `NIL` if no such node
- A *binary search tree* (bst) `T` is a binary tree used to represent dynamic set of totally-ordered items. Each node represents an item; the node's *key* represents the value of that item.

  Each node `x` of `T` has 4 fields:

  - `x.key` contains the key,
  - `x.p` references the parent node if any,
  - `x.left` references the left child if any, and
  - `x.right` references the right child if any.

  If `x` does not have a parent, or does not have a left child, or does not have a right child, then `x.p`, or `x.left`, or `x.right` contains the special value `NIL`, respectively. A node may have field(s) for satellite data as well.

  The keys of `T` are in *symmetric order* or *inorder*, i.e., `x.key` $\leq$ `y.key` $\leq$ `z.key` for any nodes `x, y, z` such that `x` is in the tree rooted at `y.left` and `z` is in the tree rooted at `y.right`.

- Binary search tree algorithms work by traversing some root-to-leave path in the tree. Therefore, their running time is proportional to the height of the tree. We will study schemes to keep this height close to $\log n$, and thus make the BST algorithms efficient. (See the Binary Tree Height Lemma.)

  Searching for a key: To find a given key `k` in the binary search tree `T`, start at the root, traverse the path down to a node containing `k`, or to a leaf if `k` is not in `T`

```
procedure SEARCH(x, k) {
    if x = NIL then
        return "k is not in T"
    else if k < x.key then
        return SEARCH(x.left, k)
    else if k > x.key then
        return SEARCH(x.right, k)
    else
        return x
}
```

  The simple path from the root of `T` to a node `x` is called `x`'s *access path.*

  The minimum key is in the leftmost node of `T`; the maximum key is in the rightmost node of `T`. So a BST can also be used as a priority queue.

Let `x` be a node in `T` that is not leftmost. The *predecessor* of `x` is the rightmost descendant of `x.left` if `x.left` $\neq$ `NIL`; it is the nearest ancestor of `x` having `x` in its right subtree if `x.left`=`NIL`.

The leftmost node of `T` has no predecessor.

The notion of *successor* of a node `x` that is not rightmost is similar.

Inserting a key: To insert a new key `k` in binary search tree `T`, create a new node `x` having key `k`, find the empty leaf position for `x`, and make `x` a child of that leaf's parent.

Deleting a node: To delete node `z` from the binary search tree `T`,

(i) if `z` has no children, replace `z` in `T` by `NIL`;

(ii) if `z` has exactly one child `x`, replace `z` in `T` by `x`;

(iii) if `z` has two children, letting `y` be the successor of `z` in `T`, letting α be the left subtree of `z`, and letting β be the right subtree of `z` but having the subtree rooted at `y` transplanted by the right subtree of `y`, transplant the tree rooted at `z` by the tree (`y`, α, β).