Lecture 5: Dictionaries

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Dictionary / Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- *Search(S, k)* – A query that, given a set S and a key k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.
- *Insert(S,x)* – A modifying operation that augments the set S with the element x.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key.

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- Logical *Predecessor(S,x), Successor(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next smallest (largest) element in S, or NIL if x is the maximum (minimum) element.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

Problem of the Day

What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list,
- A doubly-linked unsorted list,
- A singly-linked sorted list, and finally
- A doubly-linked sorted list.

Solution Blank

	singly unsorted	singly sorted	doubly unsorted	doubly sorted
Search(L, k)				
Insert(L, x)				
Delete(L, x)				
Successor(L, x)				
Predecessor(L, x)				
Minimum(L)				
Maximum(L)				

Solution

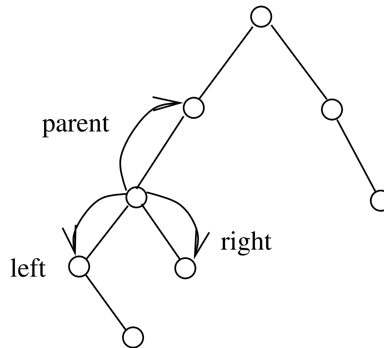
Dictionary operation	singly unsorted	double unsorted	singly sorted	doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

Binary Search Trees

Binary search trees provide a data structure which efficiently supports all six dictionary operations.

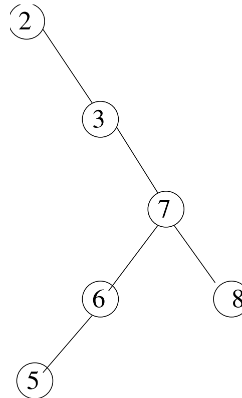
A binary tree is a rooted tree where each node contains at most two children.

Each child can be identified as either a left or right child.



Binary Search Trees

A binary *search* tree labels each node x in a binary tree such that all nodes in the left subtree of x have keys $< x$ and all nodes in the right subtree of x have keys $> x$.



The search tree labeling enables us to find where any key is.

Implementing Binary Search Trees

```
typedef struct tree {
      item_type item;
      struct tree *parent;
      struct tree *left;
      struct tree *right;
} tree;
```

The parent link is optional, since we can usually store the pointer on a stack when we encounter it.

Searching in a Binary Tree: Implementation

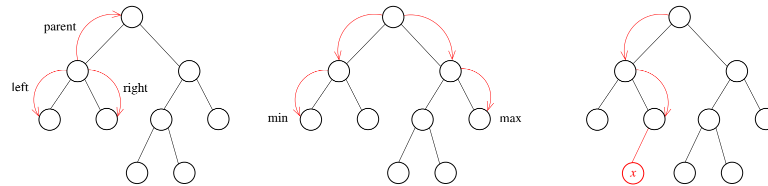
```
tree *search_tree(tree *l, item_type x)
{
      if (l == NULL) return(NULL);

      if (l->item == x) return(l);

      if (x < l->item)
            return( search_tree(l->left, x) );
      else
            return( search_tree(l->right, x) );
}
```

Searching in a Binary Tree: How Much Time?

The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees – recursive structure, recursive algorithm.
This takes time proportional to the height of the tree, $O(h)$.

Maximum and Minimum

Where are the maximum and minimum elements in a binary search tree?

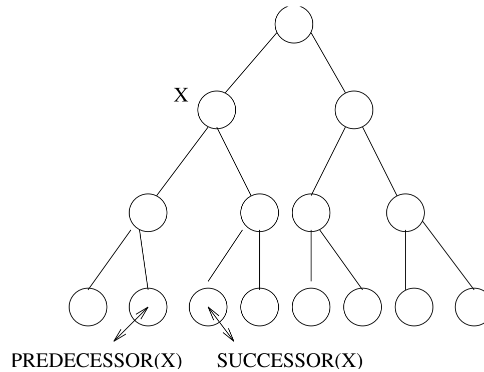


Finding the Minimum

```
tree *find_minimum(tree *t)
{
      tree *min; (* pointer to minimum *)

      if (t == NULL) return(NULL);

      min = t;
      while (min->left != NULL)
            min = min->left;
      return(min);
}
```

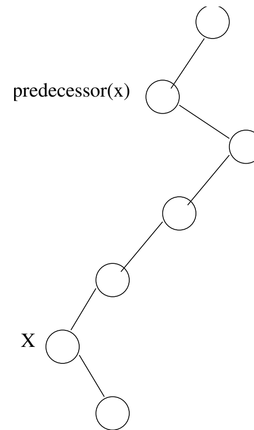
Finding the max or min takes time proportional to the height of the tree, $O(h)$.

Where is the Predecessor?: Internal Node



If X has two children, its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

Where is the Successor?: Leaf Node

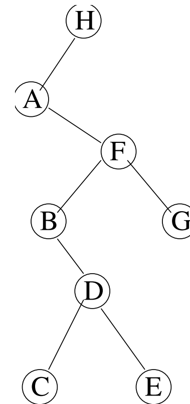


If it does not have a left child, a node's predecessor is its first left ancestor.

The proof of correctness comes from looking at the in-order traversal of the tree.

In-Order Traversal

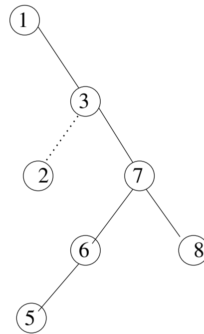
```
void traverse_tree(tree *l)
{
        if (l != NULL) {
                traverse_tree(l->left);
                process_item(l->item);
                traverse_tree(l->right);
        } }
```



This traversal visits the nodes in $ABCDEFGH$ order. Because it spends $O(1)$ time at each of n nodes in the tree, the total time is $O(n)$.

Tree Insertion

Do a binary search to find where it should be, then replace the termination NIL pointer with the new item.



Insertion takes time proportional to tree height, $O(h)$.

Tree Insertion Code

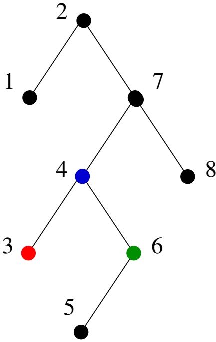
```
insert_tree(tree **l, item_type x, tree *parent)
{
      tree *p; (* temporary pointer *)

      if (*l == NULL) {
            p = malloc(sizeof(tree)); (* allocate new node *)
            p->item = x;
            p->left = p->right = NULL;
            p->parent = parent;
            *l = p; (* link into parent's record *)
            return;
      }

      if (x < (*l)->item)
            insert_tree(&((*l)->left), x, *l);
      else
            insert_tree(&((*l)->right), x, *l);
}
```

Tree Deletion

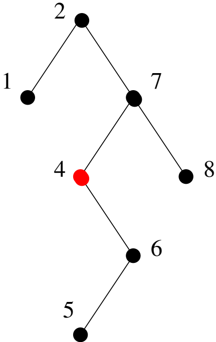
Deletion is trickier than insertion, because the node to die may not be a leaf, and thus effect other nodes.
There are three cases:

- Case (a), where the node is a leaf, is simple - just NIL out the parents child pointer.
- Case (b), where a node has one chld, the doomed node can just be cut out.
- Case (c), relabel the node as its successor (which has at most one child when z has two children!) and delete the successor!

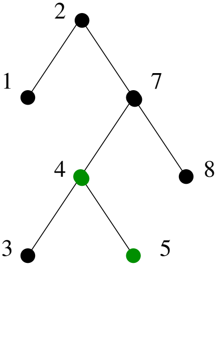
Cases of Deletion



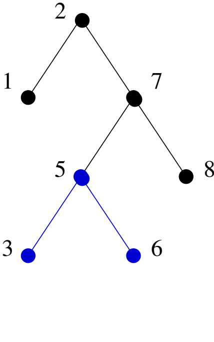
initial tree



delete node with zero children (3)



delete node with 1 child (6)



delete node with 2 children (4)

Binary Search Trees as Dictionaries

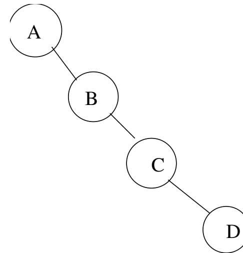
All six of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.
The best height we could hope to get is $\lg n$, if the tree was perfectly balanced, since

$$\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$$

But if we get unlucky with our order of insertion or deletion, we could get linear height!

Tree Insertion: Worst Case Height

insert(a)
insert(b)
insert(c)
insert(d)



If we are unlucky in the order of insertion, and take no steps to rebalance, the tree can have height $\Theta(n)$.

Tree Insertion: Average Case Analysis

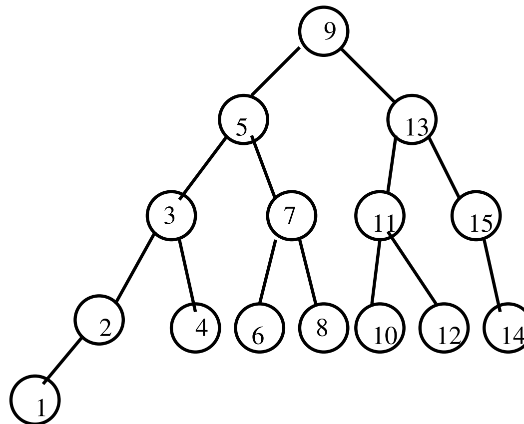
In fact, binary search trees constructed with random insertion orders *on average* have $\Theta(\lg n)$ height.

Why? Because half the time the insertion will be closer to the median key than an end key.

Our future analysis of Quicksort will explain more precisely why the expected height is $\Theta(\lg n)$.

Perfectly Balanced Trees

Perfectly balanced trees require a lot of work to maintain:



If we insert the key 1, we must move every single node in the tree to rebalance it, taking $\Theta(n)$ time.

Balanced Search Trees

Therefore, when we talk about "balanced" trees, we mean trees whose height is $O(\lg n)$, so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take $O(\lg n)$ time.

No data structure can be better than $\Theta(\lg n)$ in the worst case on all these operations.

Extra care must be taken on insertion and deletion to guarantee such performance, by rearranging things when they get too lopsided.

Red-Black trees, *AVL trees*, *2-3 trees*, *splay trees*, and *B-trees* are examples of balanced search trees used in practice and discussed in most data structure texts.

Where Does the Log Come From?

Often the logarithmic term in an algorithm analysis comes from using a balanced search tree as a dictionary, and performing many (say, n) operations on it.

But often it comes from the idea of a balanced binary tree, partitioning the items into smaller and smaller subsets, and doing little work on each of $\log(n)$ levels.

Think binary search.