Dynamic Sets

CLRS Reading: Chapters 10, 12, pages 228 -- 252, 286 -- 307

Dynamic Sets*

Search(set, key)
Returns a pointer to object in set with specified key.

Insert(set, obj)
Update set to add the object obj.

Delete(set, obj)
Update set to remove the object obj.

Minimum(set)
Returns object in set with smallest key.

Maximum(set)
Returns object in set with largest key.

Successor(set, obj)
Returns object in set whose key follows that of obj.

Predecessor(set, obj)
Returns object in set whose key precedes that of obj.

*Versus dynamic bags.
### Unsorted Arrays

**Unsorted Array**

<table>
<thead>
<tr>
<th>N</th>
<th>512 170 275 897 061 912 087 503</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Sets, not bags**

* Passing a pointer (i.e., an index) to Delete improves the situation considerably.

### Sorted Arrays

**Sorted Arrays**

**Arguments to *s are pointers**

<table>
<thead>
<tr>
<th>N</th>
<th>061 087 170 275 503 512 897 912</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Passing a pointer (i.e., an index) to Delete improves the situation considerably.
Sorted Linked Lists

Search Insert* Delete* Successor* Predecessor* Max Min

Arguments to *s are pointers

Sorted Circular Doubly-Linked Lists

Search Insert* Delete* Successor* Predecessor* Max Min

Arguments to *s are pointers
The Best of Both Worlds

Binary Tree

Binary Search Trees
### Sorting Nodes

**Inorder-Tree-Walk**($x$)

1. If $x \neq \text{NIL}$
   1. Inorder-Tree-Walk($x$.left)
   2. Print $x$.key
   3. Inorder-Tree-Walk($x$.right)

### Searching Trees

**Tree-Search**($x$, $k$)

1. If $x = \text{NIL}$ or $k = x$.key
   - Return $x$
2. If $k < x$.key
   - Return Tree-Search($x$.left, $k$)
3. Else return Tree-Search($x$.right, $k$)
Largest and Smallest

Tree-Minimum(x)
   while x.left ≠ NIL
      x = x.left
   return x

Before and After

Tree-Predecessor(x)
   if x.left ≠ NIL
      return Tree-Maximum(x.left)
   y = x.p
   while y ≠ NIL and x = y.left
      x = y
      y = y.p
   return y
**Tree-Insert**($T$, 733)

Walk down tree to find a leaf position for $z$. Let $y$ be $z$’s prospective parent node in $T$. Attach $z$ as left or right child of $y$.

**Insertion Code**

```
Tree-Insert($T$, $z$)
    $y = \text{NIL}$
    $x = T.\text{root}$
    while $x \neq \text{NIL}$
        $y = x$
        if $z.\text{key} < x.\text{key}$
            $x = x.\text{left}$
        else $x = x.\text{right}$
    $z.p = y$
    if $y = \text{NIL}$
        $T.\text{root} = z$
    elseif $z.\text{key} < y.\text{key}$
        $y.\text{left} = z$
    else $y.\text{right} = z$
```
Deleting a Node Is A Pain*

If \( z \) Has Two Children

\[ y = \text{successor}(z) \]
Copy \( y \)'s key and info into node \( z \)
delete node \( y \)

*Although some cases are easy.
<table>
<thead>
<tr>
<th>Case</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $z$ has no children or only a right child, replace $z$ with its</td>
<td>$\text{if } z.left = \text{NIL}$</td>
</tr>
<tr>
<td>right child</td>
<td>$\text{Transplant}(T, z, z.right)$</td>
</tr>
<tr>
<td>If $z$ has only a left child, replace $z$ with its left child</td>
<td>$\text{elseif } z.right = \text{NIL}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Transplant}(T, z, z.left)$</td>
</tr>
<tr>
<td>Otherwise, prepare to replace $z$ with its successor</td>
<td>$\text{else } y = \text{Tree-Successor}(z)$</td>
</tr>
<tr>
<td>If the successor is not $z$’s child, replace the successor’s node</td>
<td>$\text{if } y.p \neq z$</td>
</tr>
<tr>
<td>with its right child. (It has no left child.)</td>
<td>$\text{Transplant}(T, y, y.right)$</td>
</tr>
<tr>
<td></td>
<td>$y.right = z.right$</td>
</tr>
<tr>
<td></td>
<td>$y.right.p = y$</td>
</tr>
<tr>
<td>Replace $z$ with its successor</td>
<td>$\text{Transplant}(T, z, y)$</td>
</tr>
<tr>
<td></td>
<td>$y.Left = z.left$</td>
</tr>
<tr>
<td></td>
<td>$y.left.p = y$</td>
</tr>
</tbody>
</table>