`http://www.cs.unh.edu/~ruml/cs758`

Structure	Find	Insert	Delete
List (unsorted)			
List (sorted)			
Array (unsorted)			
Array (sorted)			
Heap			
Hash table			
Binary tree (unbalanced)			
Binary tree (balanced)			

set operations: $\cup, \cap, -$

Binary Search Trees

node: data, left, right, parent

What's the invariant?

Next

if no right child, want lowest ancester 'on the right'

succ(x)
1. if right child exists
2.   return min under right child
3. else
4.   return up(x)

up(x)
5. $p \leftarrow x$'s parent
4. if p doesn't exist or x is p's left child
5.   return p
6. else
7.   return up(p)

Insert

insert (n)
1. n's parent $\leftarrow$ find-parent(n, root, nil)
2. if parent is nil
3.   root $\leftarrow n$
4. else
5.   if n should be before parent
6.   parent's left child $\leftarrow n$
7.   else
8.   parent's right child $\leftarrow n$

find-parent(n, curr, parent)
9. if curr doesn't exist
10.   return parent
11. if n should be before curr
12.   return find-parent(n, curr's left child, curr)
13. else
14.   return find-parent(n, curr's right child, curr)

Break

- asst 2
- asst 3
- Steve's office hours survey

Deletion Outline

3 cases of delete(n):

1. no kids: pointer from parent $\leftarrow$ nil
2. 1 kid: substitute child for n at parent
3. 2 kids: let successor be s.
   note s is in n's right subtree and has no left child.
   (a) s takes n's place at parent
   (b) n's left subtree becomes s's
   (c) somehow, rest of n's right subtree becomes s's...

will split 3(c) into 2 cases...

Deletion Outline, Again

4 cases of delete(n):

1. no kids or no left child: substitute right subtree at parent
2. no right child: substitute left subtree at parent

now we have the hard 2-kids cases:

3. successor s is n's right child:
   (a) substitute s for n
   (b) add n's left subtree as s's left subtree
4. successor s is deeper:
   (a) substitute s's right subtree for s
   (b) add n's right subtree as s's right subtree
   (c) as above, substitute s for n
   (d) as above, add n's left subtree as s's left subtree

Moving Subtrees

put new where old was:

```
substitute(old, new)
if old's parent is nil
    root ← new
else
    if old is parent's left child
        parent's left child ← new
    else, parent's right child ← new
if new ≠ nil
    new's parent ← old's parent
```

Deletion

delete(n)

if n has no left child
    substitute(n, n's right subtree) *case 1*
else if n has no right child
    substitute(n, n's left subtree) *case 2*
else
    $s \leftarrow$ min in n's right subtree
    if n is not s's parent *case 4*
        substitute(s,s's right subtree)
        s's right subtree $\leftarrow$ n's right subtree
        s's right child's parent $\leftarrow s$
    substitute(n,s) *cases 3 and 4*
    s's left subtree $\leftarrow$ n's left subtree
    s's left child's parent $\leftarrow s$

Random Deletion/Insertion Behavior

Jeff Eppinger: don't try this at home!

Random Deletion/Insertion Behavior

Jeff Eppinger: don't try this at home! Delete should alternate between successor and predecessor.

ACM's 1983 George E. Forsythe Award for best undergraduate student paper

Real solution: balanced trees!

EOLQs

- What's still confusing?
- What question didn't you get to ask today?
- What would you like to hear more about?

Please write down your most pressing question about algorithms and put it in the box on your way out.
Thanks!