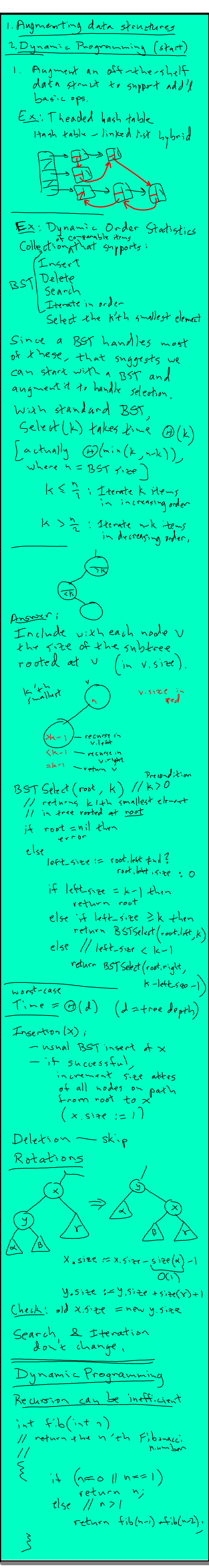
1. Augmenting data structures
2. Dynamic Programming (start)
1. Augment an off-the-shelf data struct to support add'l basic ops.
Ex: Theaded hash table
Hash table - linked list hybrid
Ex: Dynamic Order Statistics
Collection of comparable items that supports:
BST
Insert
Delete
Search
Iterate in order
Select the k'th smallest element
Since a BST handles most of these, that suggests we can start with a BST and augment it to handle selection.
With standard BST, Select(k) takes time Θ(k)
[actually Θ(min(k, n-k)), where n = BST size]
k ≤ n/2 : Iterate k items in increasing order
k > n/2 : Iterate n-k items in decreasing order.
>k
<k
Answer:
Include with each node v the size of the subtree rooted at v (in v.size).
k'th smallest
v
n
v.size in red
>k-1 — recurse in v.left
<k-1 — recurse in v.right
=k-1 — return v
BST Select(root, k) // k > 0 Precondition
// returns k'th smallest element
// in tree rooted at root
if root = nil then
error
else
left_size := root.left ≠ nil ? root.left.size : 0
if left_size = k-1 then
return root
else if left_size ≥ k then
return BSTSelect(root.left, k)
else // left_size < k-1
return BSTSelect(root.right, k - left_size - 1)
worst-case
Time = Θ(d) (d = tree depth)
Insertion(x):
- usual BST insert of x
- if successful, increment size attrs of all nodes on path from root to x
(x.size := 1)
Deletion — skip
Rotations
x
y
α
β
γ
⇒
y
α
x
β
γ
x.size := x.size - size(α) - 1
O(1)
y.size := y.size + size(γ) + 1
Check: old x.size = new y.size
Search & Iteration don't change.
Dynamic Programming
Recursion can be inefficient
int fib(int n)
// return the n'th Fibonacci number
//
{
if (n==0 || n==1)
return n;
else // n > 1
return fib(n-1) + fib(n-2);
}

fib(5)
fib(4) fib(3) — fib(1)
fib(3) fib(2) fib(2) — fib(1) fib(0)
fib(2) fib(1) fib(1) fib(0)
fib(1) fib(0)

Takes time exponential in n.

One fix: memoization

- Maintain a dictionary of (input, output) pairs of recursive calls.
- When making a recursive call, first check the dict to see if it is there. If so return output (no rec. call). otherwise
  - make the recursive call
  - when value is found add that to the dictionary.

Other method:
bottom-up table

useful when space of all subcalls can be neatly indexed in an array. Fill the array from the bottom up, computing each array entry based on previously fill entries.

Ex: Fib sequence

```
fib(n)
 allocate array F[0..n]
 F[0] := 0
 F[1] := 1
 for j := 2 to n
     F[j] := F[j-1] + F[j-2]
 return F[n]
```

Problem: Given a list A[1..n] of positive numbers
Want to find the largest sum of nonadjacent entries from A.

BigSum(A, k) finds largest sum of nonadj numbers in A[1..k] ($k \le n$)

```
 if k = 0 then return 0
 if k = 1 then return A[1]
 // k ≥ 2
 sum1 := BigSum(A, k-1)
 sum2 := BigSum(A, k-2)
 return max(sum1, sum2 + A[k])
```

Initial call: BigSum(A, n).

Dynamic Programming:

```
 allocate table S[0..n]
 // S[j] == BigSum(A,k)
 S[0] := 0
 S[1] := 1

 for j := 2 to n do
     S[j] := max(S[j-1], S[j-2] + A[j])
 return S[n].
```