Searching

Dan Barrish-Flood
Dynamic Sets

• Manage a changing set $S$ of elements. Every element $x$ has a key, key[$x$].

• Operations:
  – SEARCH($k$): return $x$, s.t. key[$x$] = $k$ (or NIL)
  – INSERT($x$): Add $x$ to the set $S$
  – DELETE($x$): Remove $x$ from the set $S$ (this uses a pointer to $X$, not a key value).
  – MINIMUM: Return the element w/the min key.
  – MAXIMUM: Return the element w/the max key.
  – SUCCESSOR($x$): Return the next largest element.
  – PREDECESSOR($x$): Return the next smallest element.
Implementing Dynamic Sets with unordered array

- SEARCH takes \( \Theta(n) \)
- INSERT takes \( \Theta(1) \) if we don’t care about duplicates.
- DELETE takes \( \Theta(n) \), so do MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.
- How to deal with fixed length limitation? Grow the array as needed...
Growable arrays

\textbf{INSERT}(x)

1. if next\_free[A] > length[A] then
2. \quad \text{GROW}(A)
3. \quad A[\text{next\_free}[A]] \leftarrow x
4. \quad \text{next\_free}[A] \leftarrow \text{next\_free}[A] + 1

\textbf{GROW}(A)

1. allocate B to be $2 \cdot \text{length}[A]$
2. for $i \leftarrow$ to next\_free[A] – 1 do
3. \quad B[i] \leftarrow A[i]
4. \quad \text{next\_free}[B] \leftarrow \text{next\_free}[A]
5. free A
6. A \leftarrow B
Running time of INSERT

Observation: a GROW only happens when the size is a power of 2.

Total time for n insertions:

\[
n + \sum_{k=0}^{\lfloor \log n \rfloor} 2^k
\]

\[
= n + 2^{\lfloor \log n \rfloor + 1} - 1
\]

\[
< n + 2^{\log n + 2}
\]

\[
= n + 4n
\]

\[
= \Theta(n)
\]

Dividing by n, the time for INSERT = \Theta(1)
Sorted Arrays

• INSERT takes $\Theta(n)$ time, so does DELETE
• MIN, MAX, SUCC, PRED take $\Theta(1)$ time.
• What about SEARCH? Use *binary search*:
  – Guess the middle
  – If it’s correct, we’re done.
  – If it’s too high, recurse on the left half
  – If it’s too low, recurse on the right half
Running Time for Binary Search

T(1) = 1
T(n) = T(n/2) + 1, for n \geq 2

Do the domain transformation:
n = 2^k and k = \lg n, etc.
telescope, back-substitute, you know the routine: you end with:
T(n) = \Theta(\lg n), worst-case
Linked Lists for Dynamic Sets

• **Unordered** (pretty much same as unordered array, running-time-wise)
  – **INSERT** in $\Theta(1)$ (stick it at the head)
  – **SEARCH** in $\Theta(n)$ (linear search)
  – **DELETE** in $\Theta(n)$
    • $\Theta(1)$ on doubly-linked list (sort of...)
  – **MIN, MAX** in $\Theta(n)$
  – **SUCC, PRED** in $\Theta(n)$
Linked Lists (Continued)

• Sorted
  – **INSERT** in $\Theta(n)$ ...need to find the right place
  – **SEARCH** in $\Theta(n)$ ...no binary search on linked lists!
  – **DELETE** in $\Theta(n)$ for singly linked list
    • $\Theta(1)$ for doubly-linked list, sort of...
  – **MIN** in $\Theta(1)$
  – **MAX** in $\Theta(1)$ with end pointer
  – **SUCC, PRED** can take $\Theta(1)$
    • **PRED** iff doubly-linked
Can we get all operations to run in time $O(\lg n)$?

• Yes, pretty much, using trees.
• A **binary search tree** is a binary tree with a key at each node $x$, where:
  – All the keys in the left subtree of $x \leq \text{key}[x]$
  – All the keys in the right subtree of $x \geq \text{key}[x]$
Two binary search trees

How do we search? Find min, max? insert, delete?
Here’s a nice BST

we’ll refer back to this slide for SEARCH, MIN, MAX, etc.
Node representation for BST

Node Representation

\[ \begin{align*}
X & \quad P(x) \\
21 & \quad \text{KEY}(x) \\
& \quad \text{LEFT}(x) \\
& \quad \text{RIGHT}(x)
\end{align*} \]

\[ \begin{align*}
\text{NOTE: } \square &= \text{NIL}
\end{align*} \]
Code for BST SEARCH

```plaintext
TREE-SEARCH(x, k)
1 if x = NIL or k = key[x]
2 then return x
3 if k < key[x]
4 then return TREE-SEARCH(left[x], k)
5 else return TREE-SEARCH(right[x], k)

ITERATIVE-TREE-SEARCH(x, k)
1 while x ≠ NIL and k ≠ key[x]
2 do if k < key[x]
3 then x ← left[x]
4 else x ← right[x]
5 return x
```

These run in time O(h), where h is the height of the tree.
Code for BST MIN, MAX (both run in $O(h)$)

For MIN, follow *left* pointers from the root.

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2    do x ← left[x]
3  return x
```

For MAX, follow *right* pointers from the root.

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2    do x ← right[x]
3  return x
```
BST SUCCESSOR, PREDECESSOR

SUCCESSOR is broken into two cases. SUCCESSOR runs in $O(h)$, where $h$ is the height of the tree. The case for PREDECESSOR is symmetric.

```
TREE-SUCCESSOR(x)
1 if right[x] ≠ NIL
2    then return TREE-MINIMUM(right[x])
3 y ← p[x]
4 while y ≠ NIL and x = right[y]
5    do x ← y
6    y ← p[y]
7 return y
```
Turning a BST into a sorted list (or printing it)

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      then INORDER-TREE-WALK(left[x])
3          print key[x]
4      INORDER-TREE-WALK(right[x])
```

Since the procedure is called (recursively) exactly twice for each node (once for its left child and once for its right child), and does a constant amount of work, the tree walk takes $\Theta(n)$ time for a tree of $n$ nodes. (Later, we will see about pre- and post-order tree walks.)
BST INSERT runs in $O(h)$

TREE-INSERT($T$, $z$)
1   $y \leftarrow$ NIL
2   $x \leftarrow$ root[$T$]
3   while $x \neq$ NIL do $y \leftarrow x$
4       if key[$z$] < key[$x$]
5           then $x \leftarrow$ left[$x$]
6       else $x \leftarrow$ right[$x$]
7   $p[z] \leftarrow y$
8   if $y =$ NIL then root[$T$] $\leftarrow z$
9       else if key[$z$] < key[$y$] then left[$y$] $\leftarrow z$
10      else right[$y$] $\leftarrow z$

$\triangleright$ Tree $T$ was empty

4-Searching
BST DELETE runs in O(h)

```
TREE-DELETE(T, z)
1   if left[z] = NIL or right[z] = NIL
2     then y ← z
3   else y ← TREE-SUCCESSOR(z)
4   if left[y] ≠ NIL
5     then x ← left[y]
6   else x ← right[y]
7   if x ≠ NIL
8     then p[x] ← p[y]
9   if p[y] = NIL
10    then root[T] ← x
11   else if y = left[p[y]]
12     then left[p[y]] ← x
13     else right[p[y]] ← x
14   if y ≠ z
15     then key[z] ← key[y]
16     copy y’s satellite data into z
17 return y
```
TREE_DELETE case one
TREE_DELETE case two
TREE_DELETE case three