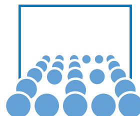
Fundamental Algorithms

Chapter 5: Searching

Michael Bader

Winter 2013/14



Searching

Definition (Search Problem)

Input: a sequence or set A of n elements (objects) $\in \mathcal{A}$, and an element $x \in \mathcal{A}$.
Output: The (smallest) index $i \in \{1, \dots, n\}$ with $x = A[i]$, or NIL, if $x \notin A$.

```
SeqSearch (A:Array[1..n], x:Element) : Integer {
    for i from 1 to n do {
        if x = A[i] then return i;
    }
    return NIL;
}
```

Time Complexity of SeqSearch

$\rightarrow$ count number of comparisons

Worst Case:

- we have to compare every A[i] with $x \Rightarrow n$ comparisons
- occurs if A[n]=x or if $x \notin A$

Time Complexity of SeqSearch (2)

Average Case:

- simplifying assumption: no duplicate elements
- $p :=$ probability that $x = \mathrm{A[i]}$
(assumption: p independent of i)
- expected number of comparisons:

$$\bar{C}(n) = \sum_{i=1}^{n} pi + (1 - np)n = \frac{pn(n+1)}{2} + (1 - np)n$$

- assume that x occurs in A, thus $p = \frac{1}{n}$, then:

$$\bar{C}(n) = \frac{n(n+1)}{2n} + 0n = \frac{n+1}{2}$$

(on average, we have to search through half of the array)

Searching – Divide and Conquer?

Will a divide-and-conquer approach work?

```
DQSearch(A:Array[p..r], x:Integer) : Integer {
  if p=r
  then {
     if x=A[p] then return p
     else return NIL;
  }
  else {
     m := floor((p+r)/2);
     q := DQSearch(A[p,m], x);
     if q = NIL
     then return DQSearch(A[m+1,r], x)
     else return q;
  }
}
```

Binary Search on Sorted Lists

Divide-and-conquer approach only works, if the array is sorted:

```
BinarySearch (A:Array[p..r], x:Integer) : Integer {
    if p=r
    then {
        if x=A[p] then return p
        else return NIL;
    }
    else {
        m := floor((p+r)/2);
        if x <= A[m]
        then return BinarySearch(A[p..m], x)
        else return BinarySearch(A[m+1..r], x)
        end if;
    }
}
```

Time Complexity of BinarySearch

Number of comparisons on an array with n elements:

- similar to divide-and-conquer: $\log n$ subsequent recursive calls
- one comparison per call plus comparison with final element
  $\rightsquigarrow 1 + \log n$
- homework: formulate as recurrence

Discussion:

- What happens if we have to insert/delete elements in our sequence?
  $\Rightarrow$ re-sorting of the sequence required
  $\Rightarrow$ $O(n \log n)$ effort
- therefore: Searching strongly dependent on choice of appropriate data structures for inserting and deleting elements!

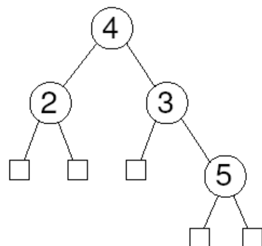
Binary Trees

Definition

A **binary tree** is either an empty tree, or an object (record, struct, . . .) consisting of a key, a reference (pointer, . . .) to a left child, and a reference (pointer, . . .) to a right child. The left and right children are, again, binary trees.

```
BinTree := emptyTree
         | (key : integer;
            leftChild : BinTree;
            rightChild : BinTree;
           );
```

Binary Tree – Example



Notation:

- generate a binary tree:

```
x = (4,(2,emptyTree,emptyTree),
      (3,emptyTree,(5,emptyTree,emptyTree)))
```

- set fields:

```
x.key = 4
x.leftChild = (2, emptyTree, emptyTree)
x.rightChild.key = 3
```

Binary Search Trees

Definition

A binary tree x is called a **binary search tree**, if it satisfies the following properties:

- for **all** keys l that are stored in x.leftChild: $l \leq$ x.key
- for **all** keys r that are stored in x.rightChild: $r \geq$ x.key
- x.leftChild and x.rightChild are binary search trees

Searching in Binary Search Trees

```
TreeSearch( x: BinTree, k:Integer) : BinTree {

  if x = emptyTree then return emptyTree;
  if x.key = k then return x;

  if x.key > k
  then return TreeSearch(x.leftChild, k)
  else return TreeSearch(x.rightChild, k);
}
```

- TreeSearch returns the subtree of x that contains the value k in its top node.
- if k does not occur as a key value in x, then TreeSearch returns an empty tree.

Complexity of TreeSearch

Number of Comparisons:

- 2 comparisons (3 if we count "x=emptyTree" for the leaf case)
- plus the number of comparisons induced by the recursive calls
- each recursive call descends the search tree by one level

Therefore:

- 2l comparisons, if k is found on the l-th level
- worst case: 2h comparisons (h is the height of the tree)

Remarks:

- for a fully balanced tree with n nodes: $O(\log n)$ comparisons
- main problem will be to build (and maintain) a balanced search tree

Inserting into a Binary Tree

```
TreeInsert( x: BinTree, k:Integer) {
   // x is a "call-by-reference"-parameter

   if x = emptyTree
   then
      x := (k, emptyTree, emptyTree);
   else
      if k < x.key
      then TreeInsert(x.leftChild, k)
      else TreeInsert(x.rightChild, k);
}
```

Complexity: again depends on the depth of the tree

Deleting from a Binary Search Tree

Problem: we must not leave a node without key

Options:

- a node with no subtrees can be deleted
- a node with one subtree can be deleted
  (eliminates one level of this partial tree)

Deleting a Node with Two Subtrees:

- replace by leftmost (i.e., smallest) node of right subtree, or
- replace by rightmost (i.e., largest) node of left subtree

Deleting the Left-Most Node

```
DeleteLeftmost (x:BinTree) : Integer {
   // return key value of leftmost node of x,
   // and delete the leftmost node

   if x.leftChild = emptyTree
   then {
      // we've found the leftmost node:
      k := x.key;
      x := x.rightChild;
      return k;
   }
   else
      // descend into left subtree
      return DeleteLeftmost(x.leftChild);
}
```

Deleting the Top Node

```
DeleteTopnode (x:BinTree) {
    // assume that x is non-empty
    if x.rightChild = emptyTree
    then
        x = x.leftChild
    else {
        // delete (and memorize) leftmost node of rightChi
        k = DeleteLeftmost(x.rightChild);
        // make the memorized node the new top node
        x.key = k;
    }
}
```

Deleting in a Binary Tree (Final Implementation)

```
TreeDelete (x:BinTree, k:Integer) {

    if x = emptyTree then return;

    if x.key = k
    then DeleteTopnode(x);
    else
        if k<x.key
        then TreeDelete(x.leftChild, k)
        else TreeDelete(x.rightChild, k);
}
```

Complexity: $O(h)$, where h is the height of the tree