Fundamental Algorithms

Chapter 3: Searching

Harald Räcke

Winter 2015/16

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 $i \in \{1, \ldots, n\}$ 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

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

$\rightarrow$ 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 $x =$ 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:

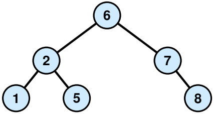
- What happens if we have to insert/delete elements in our sequence?
  $\Rightarrow$ re-sorting of the sequence required
  $\Rightarrow O(n \log n)$ effort
- therefore: Searching strongly dependent on choice of appropriate data structures for inserting and deleting elements!

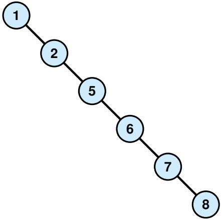
Binary Search Trees

An (**internal**) **binary search tree** stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node v have a smaller key-value than key[v] and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(**External** Search Trees store objects only at leaf-vertices)

Examples:





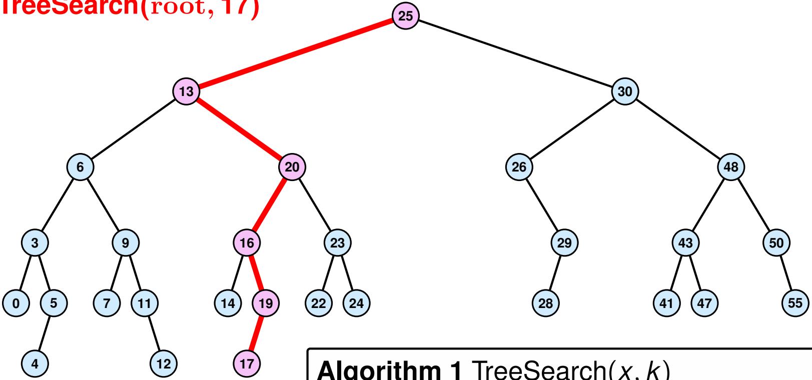
Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- $T.\text{insert}(x)$
- $T.\text{delete}(x)$
- $T.\text{search}(k)$
- $T.\text{successor}(x)$
- $T.\text{predecessor}(x)$
- $T.\text{minimum}()$
- $T.\text{maximum}()$

Binary Search Trees: Searching

TreeSearch(root, 17)

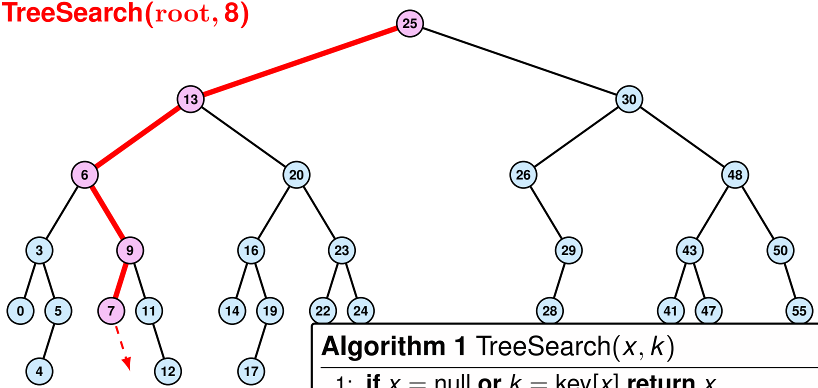


Algorithm 1 TreeSearch(x, k)

1: **if** $x =$ null **or** $k =$ key[x] **return** x
2: **if** $k <$ key[x] **return** TreeSearch(left[x], k)
3: **else return** TreeSearch(right[x], k)

Binary Search Trees: Searching

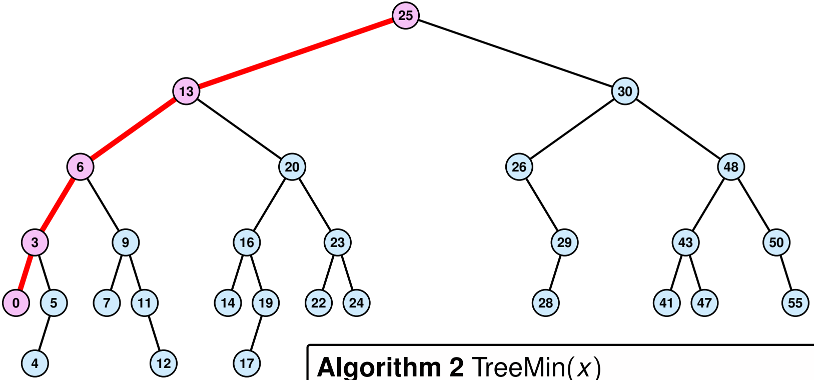
TreeSearch(root, 8)



Algorithm 1 TreeSearch(x, k)

if $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
if $k < \text{key}[x]$ **return** TreeSearch(left[x], k)
else return TreeSearch(right[x], k)

Binary Search Trees: Minimum

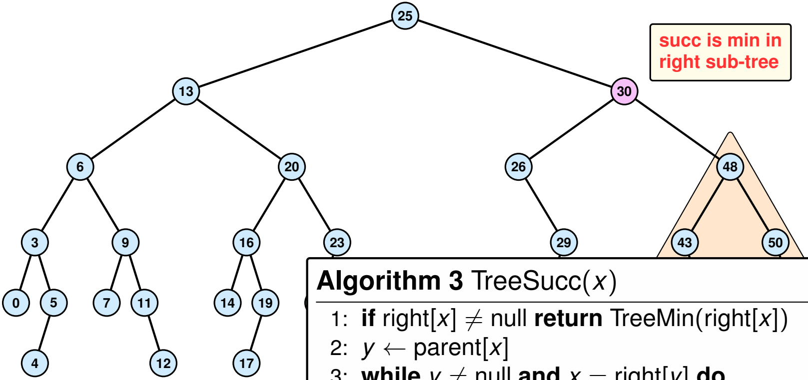


Algorithm 2 TreeMin(x)

1: **if** $x =$ null **or** left[x] $=$ null **return** x

2: **return** TreeMin(left[x])

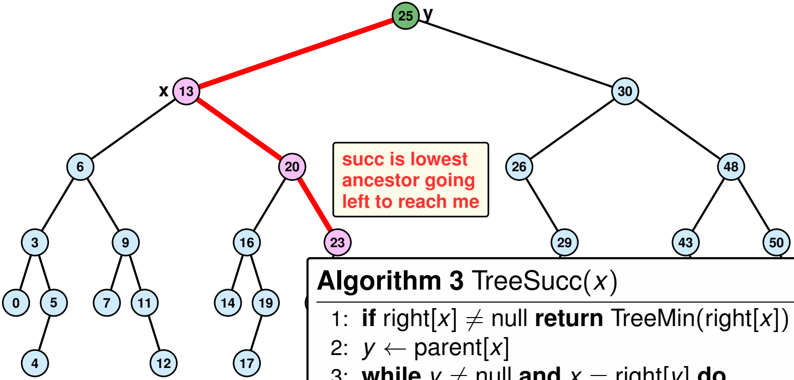
Binary Search Trees: Successor



Algorithm 3 TreeSucc(x)

if right[x] $\neq$ null **return** TreeMin(right[x])
$y \leftarrow$ parent[x]
while $y \neq$ null **and** $x =$ right[y] **do**
    $x \leftarrow y$; $y \leftarrow$ parent[x]
return y;

Binary Search Trees: Successor



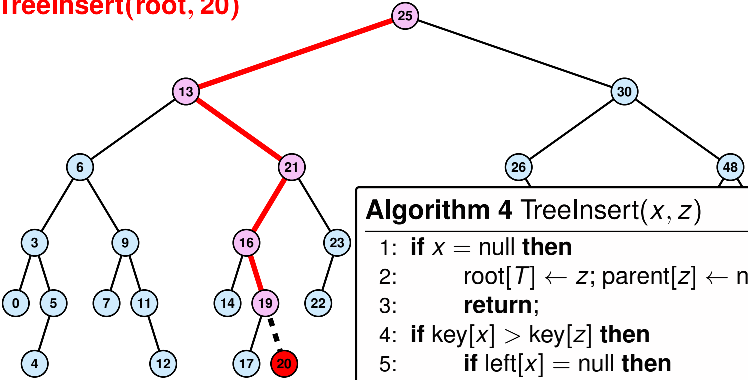
Algorithm 3 TreeSucc(x)

```
if right[x] ≠ null return TreeMin(right[x])
y ← parent[x]
while y ≠ null and x = right[y] do
    x ← y; y ← parent[x]
return y;
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

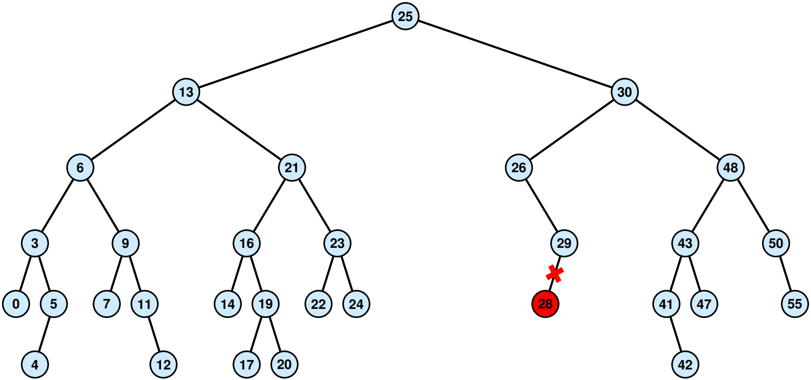


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 4 TreeInsert(x, z)

if $x = \text{null}$ **then**
  root[T] $\leftarrow z$; parent[z] $\leftarrow$ null;
  return;
if key[x] $>$ key[z] **then**
  if left[x] $=$ null **then**
    left[x] $\leftarrow z$; parent[z] $\leftarrow x$;
  else TreeInsert(left[x], z);
else
  if right[x] $=$ null **then**
    right[x] $\leftarrow z$; parent[z] $\leftarrow x$;
  else TreeInsert(right[x], z);

Binary Search Trees: Delete

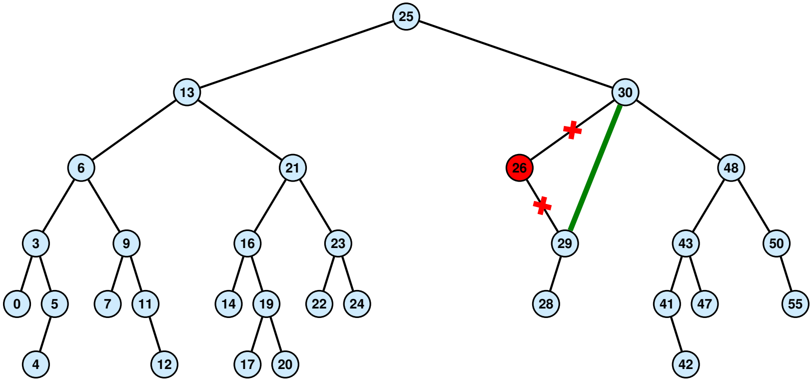


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

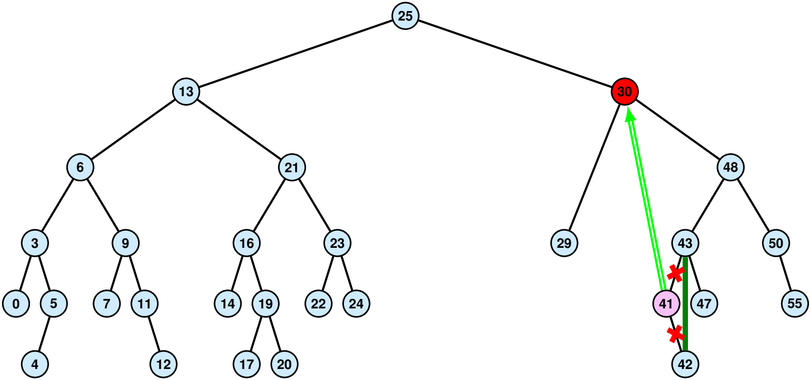


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete



Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

Algorithm 5 TreeDelete(z)

```
if left[z] = null or right[z] = null
      then y ← z else y ← TreeSucc(z);
if left[y] ≠ null
      then x ← left[y] else x ← right[y];
if x ≠ null then parent[x] ← parent[y];
if parent[y] = null then
      root[T] ← x
else
      if y = left[parent[y]] then
            left[parent[y]] ← x
      else
            right[parent[y]] ← x
if y ≠ z then copy y-data to z
```

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.