Theory 1

Introduction

SS 09

Prerequisite of Theory I

- Basic knowledge on data structures and algorithms, mathematics
- Programming language, such as C++
- Sufficient knowledge on English for reading research papers

Course Goal

- Get an in-depth knowledge on the data structures and graph algorithms, as well as database systems
- Improve the problem solving ability by doing the exercises
- Practicing skills in conducting research work:
  - Reading papers efficiently
  - Writing reviews and surveys

Course load

- 7-8 exercises
  - exercises posted one week ahead
  - exercise classes take place on each 2nd Tuesday (two hours)
  - hand in your solution before the exercise class starts!
- 5-6 Reading assignments (get bonus, by submitting high quality reviews on the assigned papers)
  - reviews sent per email
- Final exam

Topics

- Tree, balanced tree
- Hashing, Dynamic tables, Randomization
- Text search
- Relational algebra, relational calculus
- Relational DB design theory
- Transaction theory
- ...

- http://dbis.informatik.uni-freiburg.de/index.php?course=SS09/Kursvorlesung/TheoryI/index.html

Algorithm

- An algorithm is a sequence of computational steps that transform the input into the output, e.g. sorting
- the behavior an algorithm for a given application depends on
  - the number of items
  - the structure of the items,
  - possible restrictions on the item values, and the kind of storage device to be used: main memory, disks, or tapes.
- we assume the algorithms are correct.

Applications of algorithms

- The Human Genome Project has the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis.
- The Internet enables people all around the world to quickly access and retrieve large amounts of information. In order to do so, clever algorithms are employed to manage and manipulate this large volume of data.
- Electronic commerce enables goods and services to be negotiated and exchanged electronically. Public-key cryptography and digital signatures core technologies used and are based on numerical algorithms and number theory.

Hard problems

- There are some problems, for which no efficient solution is known.
- Why are NP-complete problems interesting?
  - although no efficient algorithm for an NP-complete problem has ever been found, it is unknown whether or not efficient algorithms exist for NP-complete problems.
  - the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.
  - if a problem is proven NP-complete, one can instead spend your time developing an efficient algorithm that gives a good, but not the best possible, solution.

Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

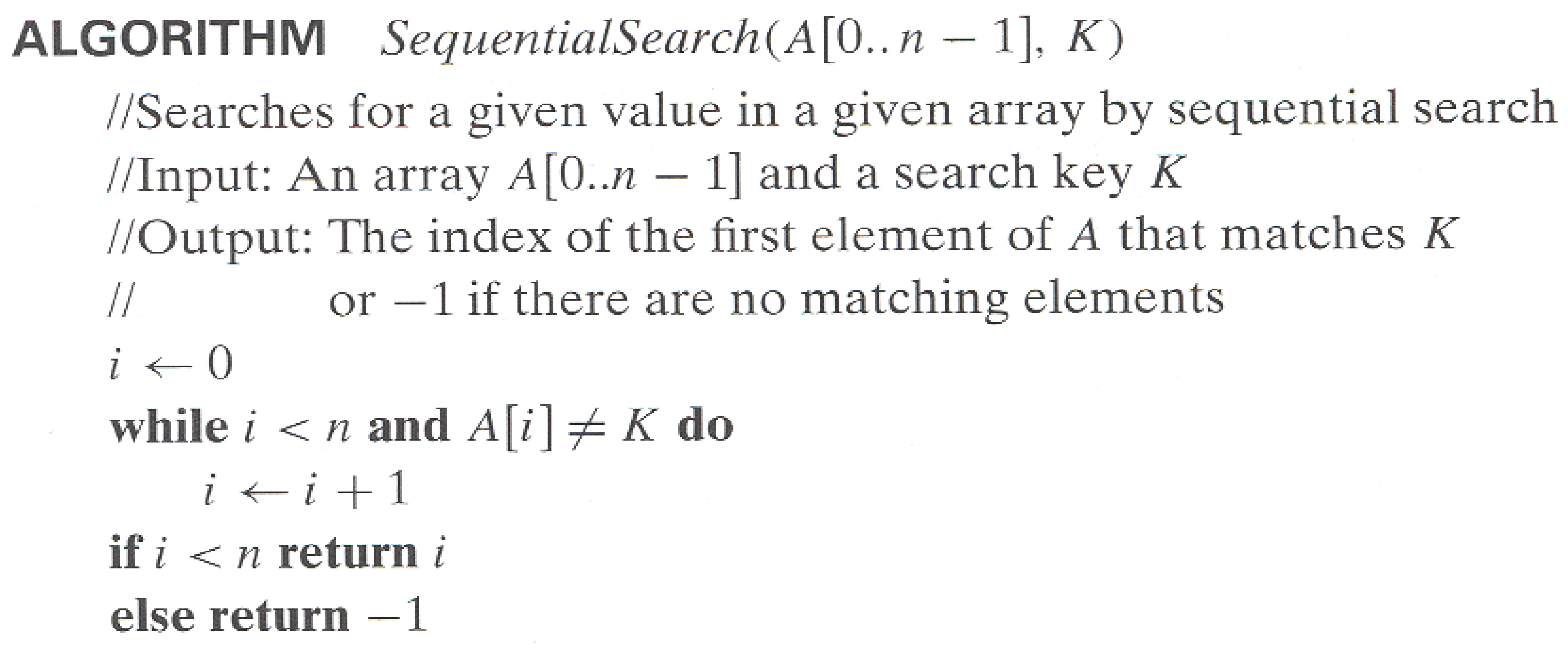
- Approaches:
  - theoretical analysis
  - empirical analysis

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs. So, avg = expected under uniform distribution.

Example: Sequential search



```
ALGORITHM  $SequentialSearch(A[0..n-1], K)$
    //Searches for a given value in a given array by sequential search
    //Input: An array $A[0..n-1]$ and a search key $K$
    //Output: The index of the first element of $A$ that matches $K$
    //        or $-1$ if there are no matching elements
    $i \leftarrow 0$
    while $i < n$ and $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

- Worst case
- Best case
- Average case

Types of formulas for basic operation's count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with unknown multiplicative constant

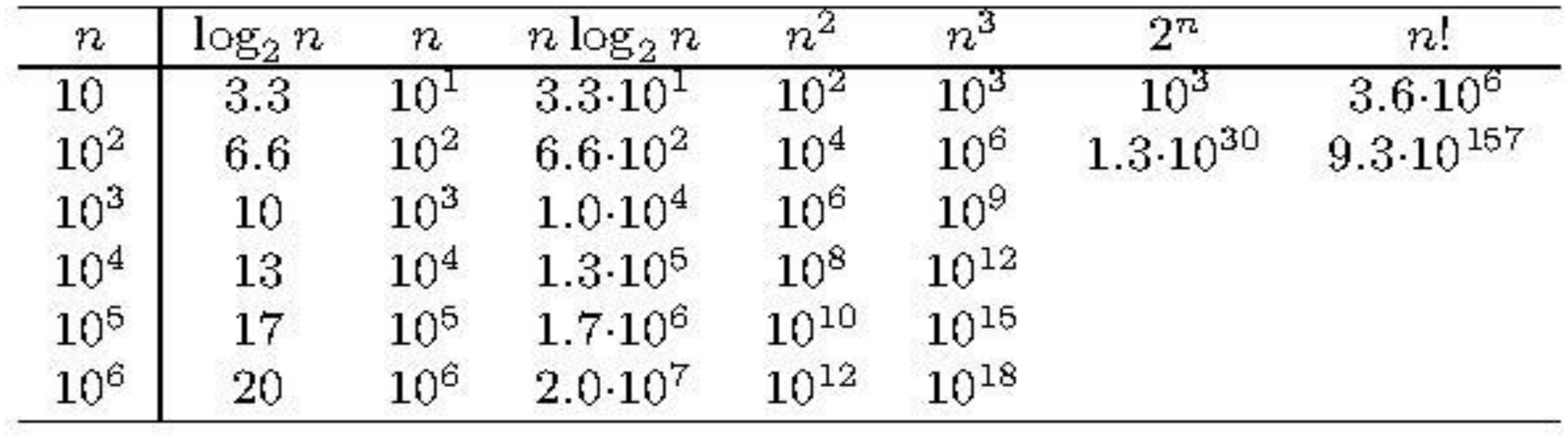
  e.g., $C(n) \approx cn^2$

Order of growth

- Most important: Order of growth within a constant multiple as $n \to \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Values of some important functions as *n*



n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow <u>no faster</u> than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

Useful summation formulas and rules

$\Sigma_{l\leq i\leq n} 1 = 1+1+\ldots+1 = n - l + 1$

In particular, $\Sigma_{l\leq i\leq n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1\leq i\leq n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1\leq i\leq n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0\leq i\leq n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0\leq i\leq n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c\Sigma a_i \qquad \Sigma_{l\leq i\leq u} a_i = \Sigma_{l\leq i\leq m} a_i + \Sigma_{m+1\leq i\leq u} a_i$

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Theory I
Algorithm Design and Analysis

(1 The Dictionary Problem: Search Trees)

Prof. Th. Ottmann

The Dictionary Problem

The dictionary problem can be described as follows:

Given: a set of objects (data) where each element can be identified by a unique ***key*** (integer, string, ...).

Goal: a structure for storing the set of objects such that at least the following operations (methods) are supported:

- search (find, access)
- insert
- delete

The Dictionary Problem (2)

The following conditions can influence the choice of a solution to the dictionary problem:

- The place where the data are stored: main memory, hard drive, tape, WORM (write once read multiple)
- The frequency of the operations:
  – mostly insertion and deletion (dynamic)
  – mostly search (static)
  – approximately the same frequencies
  – not known
- Other operations to be implemented:
  – Enumerate the set in a certain order (e.g. ascending by key)
  – Set operations: union, intersection, difference quantity, ...
  – Split
  – construct
- Measure for estimating the solution: average case, worst case, amortized worst case
- Order of executing the operations:
  – sequential
  – concurrent

The Dictionary Problem (3)

Different approaches to the dictionary problem:

- Structuring the complete universe of all possible keys: hashing
- Structuring the set of the actually occurring keys: lists, trees, graphs, ...

Trees (1)

Trees are

- generalized lists
  (each list element can have more than one successor)

- special graphs:
  – in general, a graph $G = (V,E)$ consists of a set V of vertices and
  a set $E \subseteq V \times V$ of edges.
  – the edges are either directed or undirected.
  – vertices and edges can be labelled (they contain further information).

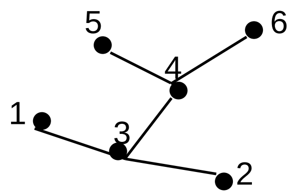
  A tree is a connected acyclic graph, where:
  # vertices = # edges + 1

- A general and central concept for the hierarchical structuring of information:
  – decision trees
  – code trees
  – syntax trees

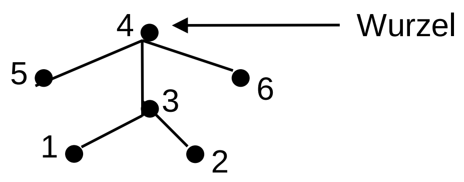
Trees (2)

Several kinds of trees can be distinguished:

- Undirected tree (with no designated root)



- Rooted tree (one node [= vertex] is designated as the root)

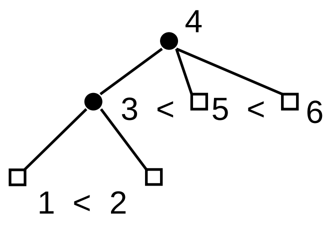


  - From each node *k* there is exactly one path (a sequence of pairwise neighbouring edges) to the root
  - the parent (or: direct predecessor) of a node *k* is the first neighbour on the path from *k* to the root
  - the children (or: direct successors) are the other neighbours of *k*
  - the rank (or: outdegree) of a node *k* is the number of children of *k*

Trees (3)

- Rooted tree:
  – root: the only node that has no parent
  – leaf nodes (leaves): nodes that have no children
  – internal nodes: all nodes that are not leaves
  – order of a tree *T*: maximum rank of a node in *T*
  – The notion *tree* is often used as a synonym for *rooted tree*.

- Ordered (rooted) tree: among the children of each node there is an order, e.g. the < relation among the keys of the nodes



- Binary tree: ordered tree of order 2; the children of a node are referred to as left child and right child.

- Multiway tree: ordered tree of order > 2

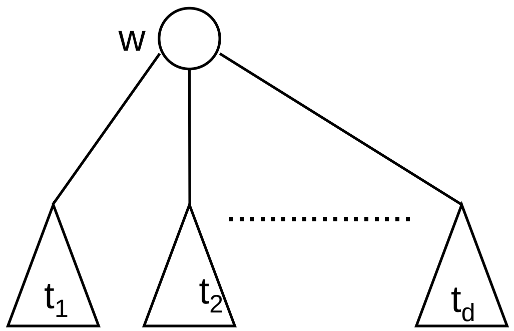
Trees (4)

A more precise definition of the set M_d of the ordered rooted trees of order d ($d \geq 1$):

- A single node is in M_d
- Let $t_1, \ldots, t_d \in M_d$ and w a node. Then w with the roots of $t_1, \ldots, t_d$ as its children (from left to right) is a tree $t \in M_d$. The t_i are subtrees of t.
  - According to this definition each node has rank d (or rank 0).
  - In general, the rank can be $\leq d$.
  - Nodes of binary trees either have 0 or 2 children.
  - Nodes with exactly 1 child could also be permitted by allowing empty subtrees in the above definition.

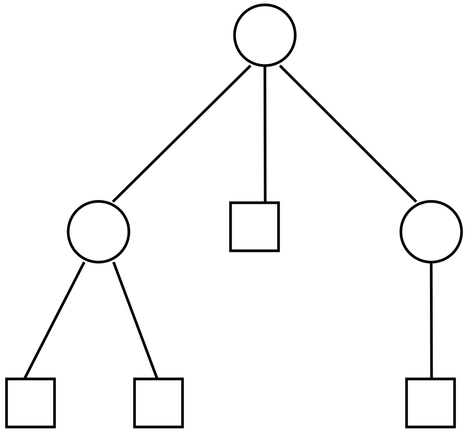
Recursive Definition

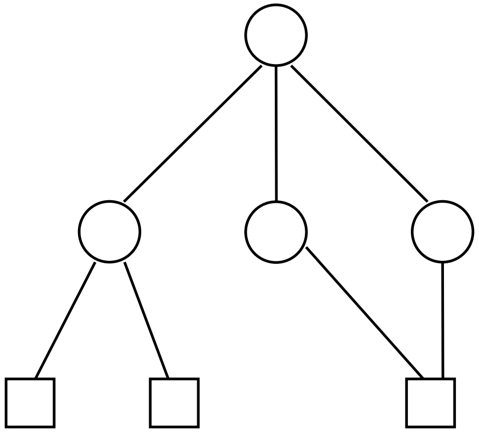
- □ is a tree of order *d*, with height 0.
- Let $t_1,\ldots,t_d$ be disjoint trees of order d. Then another tree of order *d* can be created by making the roots of $t_1,\ldots,t_d$ the successors of a newly created root *w*. The height *h* of the new tree is max $\{h(t_1),\ldots,h(t_d)\}+1$.

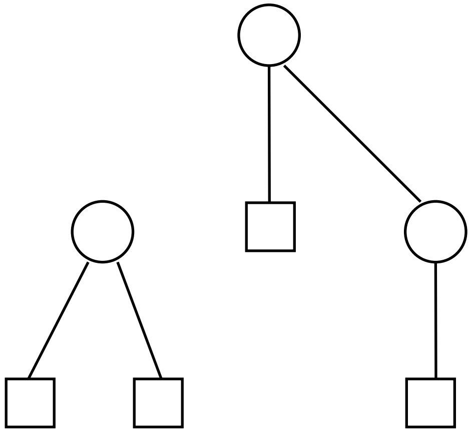


Convention: *d* = 2 binary trees, *d* > 2 multiway trees.

Examples


tree


not a tree


not a tree

(but two trees!)

Structural Properties of Trees

- Depth of a node *k*: # edges from the tree root until *k* (distance of *k* to the root)
- Height *h*(*t*) of a tree *t*: maximum depth of a leaf in *t*.
  Alternative (recursive) definition:
  – $h(\text{leaf}) = 0$
  – $h(t) = 1 + \max\{t_i \mid \text{root of } t_i \text{ is a child of the root of } t\}$ (t_i is a subtree of *t*)
- Level *i*: all nodes of depth *i*
- Complete tree: tree where each non-empty level has the maximum number of nodes.
  → all leaves have the same depth.

Applications of trees

Use of trees for the dictionary problem:

- Node: stores one data object
- Tree: stores a set of data
- Advantage (compared to hash tables): enumeration of the complete set of data (e.g. in ascending order) can be accomplished easily.

Standard binary search trees (1)

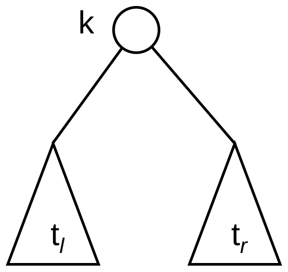
Goal: Storage, retrieval of data (more general: dictionary problem)

Two alternative ways of storage:

- Search trees: keys are stored in internal nodes
  leaf nodes are empty (usually = *null*), they represent intervals between the keys
- Leaf search trees: keys are stored in the leaves
  internal nodes contain information in order to direct the search for a key

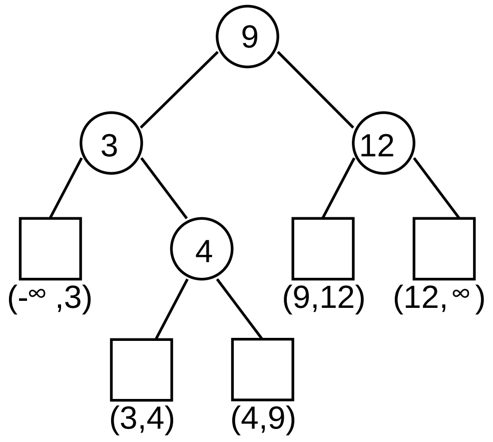
Search tree condition:

For each internal node k: all keys in the left subtree t_l of k are less (<) than the key in k and all keys in the right subtree t_r of k are greater (>) than the key in k



Standard trees (2)

Leaves in the search tree represent intervals between keys of the internal nodes

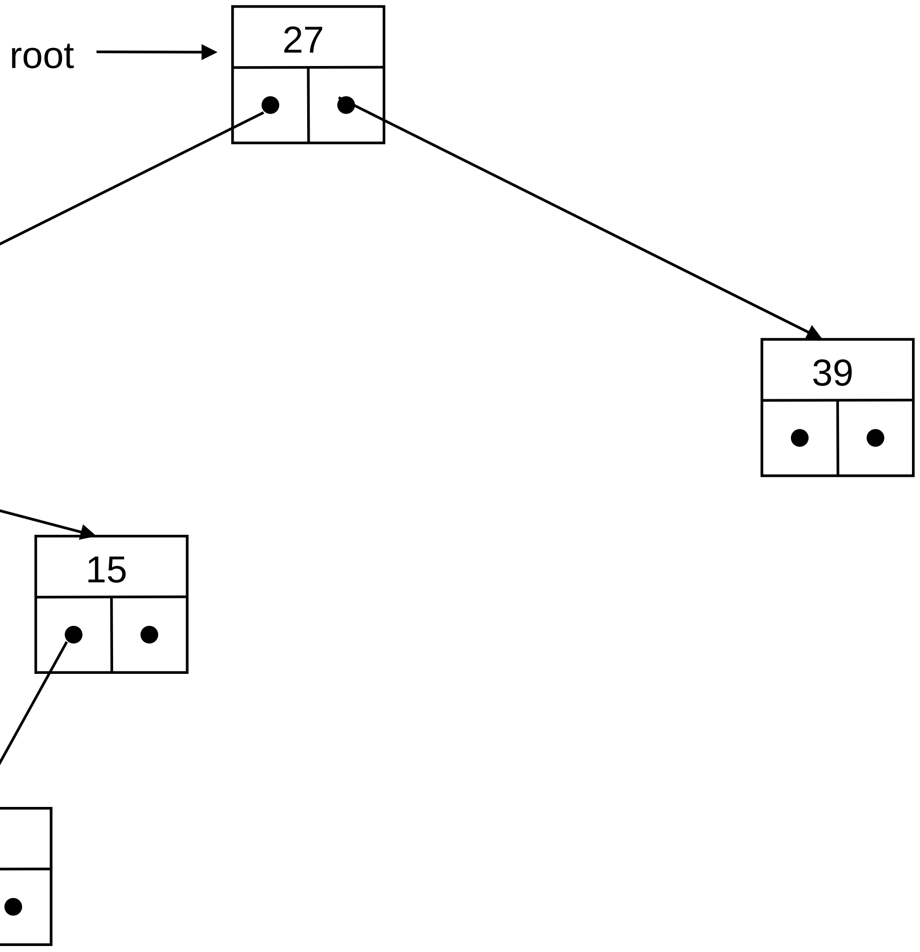


How can the search for key *s* be implemented? (leaf null)

```
k = root;
while (k != null) {
    if (s == k.key) return true;
    if (s < k.key)  k = k.left;
    else            k = k.right
}
return false;
```

Example (without stop node)

Search for key *s* ends in the internal node *k* with *k*.*key* == *s*
or in the leaf whose interval contains *s*



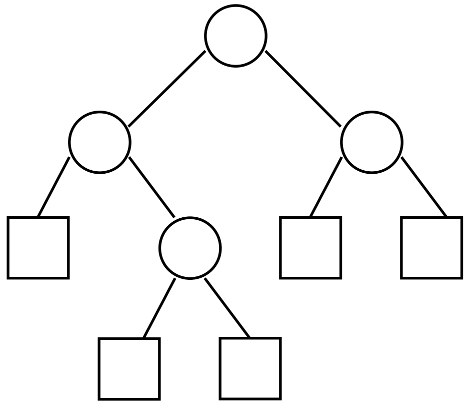
Standard trees (3)

Leaf search tree:

- Keys are stored in leaf nodes
- Clues (routers) are stored in internal nodes, such that $s_l \leq s_k \leq s_r$
  (s_l : key in left subtree, s_k : router in k, s_r : key in right subtree)
  "=" should not occur twice in the above inequality
- Choice of s: either maximum key in t_l (usual) or minimum key in t_r.

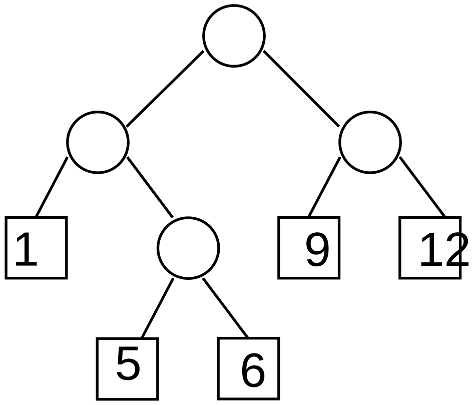
Example: leaf search tree

Leaf nodes store keys, internal nodes contain routers.



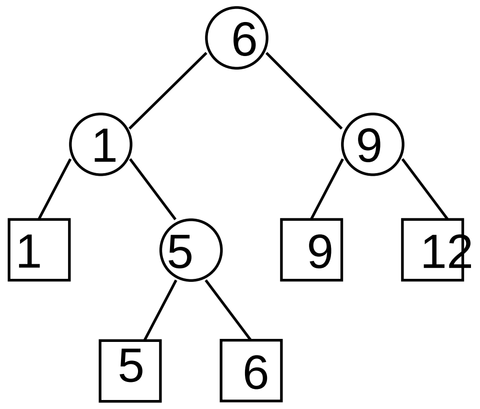
Example: leaf search tree

Leaf nodes store keys, internal nodes contain routers.



Example: leaf search tree

Leaf nodes store keys, internal nodes contain routers.

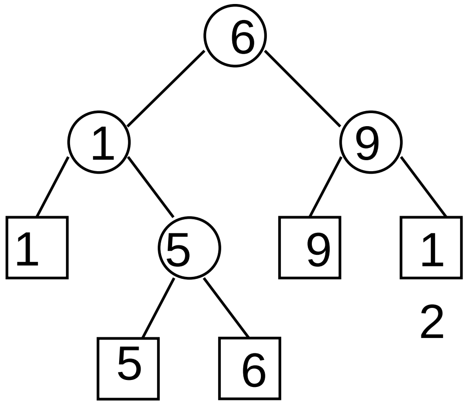


Standard trees (4)

How is the search for key *s* implemented in a leaf search tree?
(leaf   node with 2 *null* pointers)

```
k = root;
if (k == null) return false;
while (k.left != null) {          // thus also k.right != null
    if (s <= k.key) k = k.left;
    else k = k.right;
}                                 // now in the leaf
return s==k.key;
```

In the following we always talk about search trees (not leaf search trees).



Standard trees (5)

```
class SearchNode {
    int content;
    SearchNode left;
    SearchNode right;
    SearchNode (int c){     // Constructor for a node
        content = c;        // without successor
        left = right = null;
    }
} //class SearchNode

class SearchTree {
    SearchNode root;
    SearchTree () {         // Constructor for empty tree
        root = null;
    }
    // ...

}
```

Standard trees (6)

```
/* Search for c in the tree */
boolean search (int c) {
    return search (root, c);
}
boolean search (SearchNode n, int c){
    while (n != null) {
        if (c == n.content) return true;
        if (c < n.content) n = n.left;
        else n = n.right;
    }
    return false;
}
```

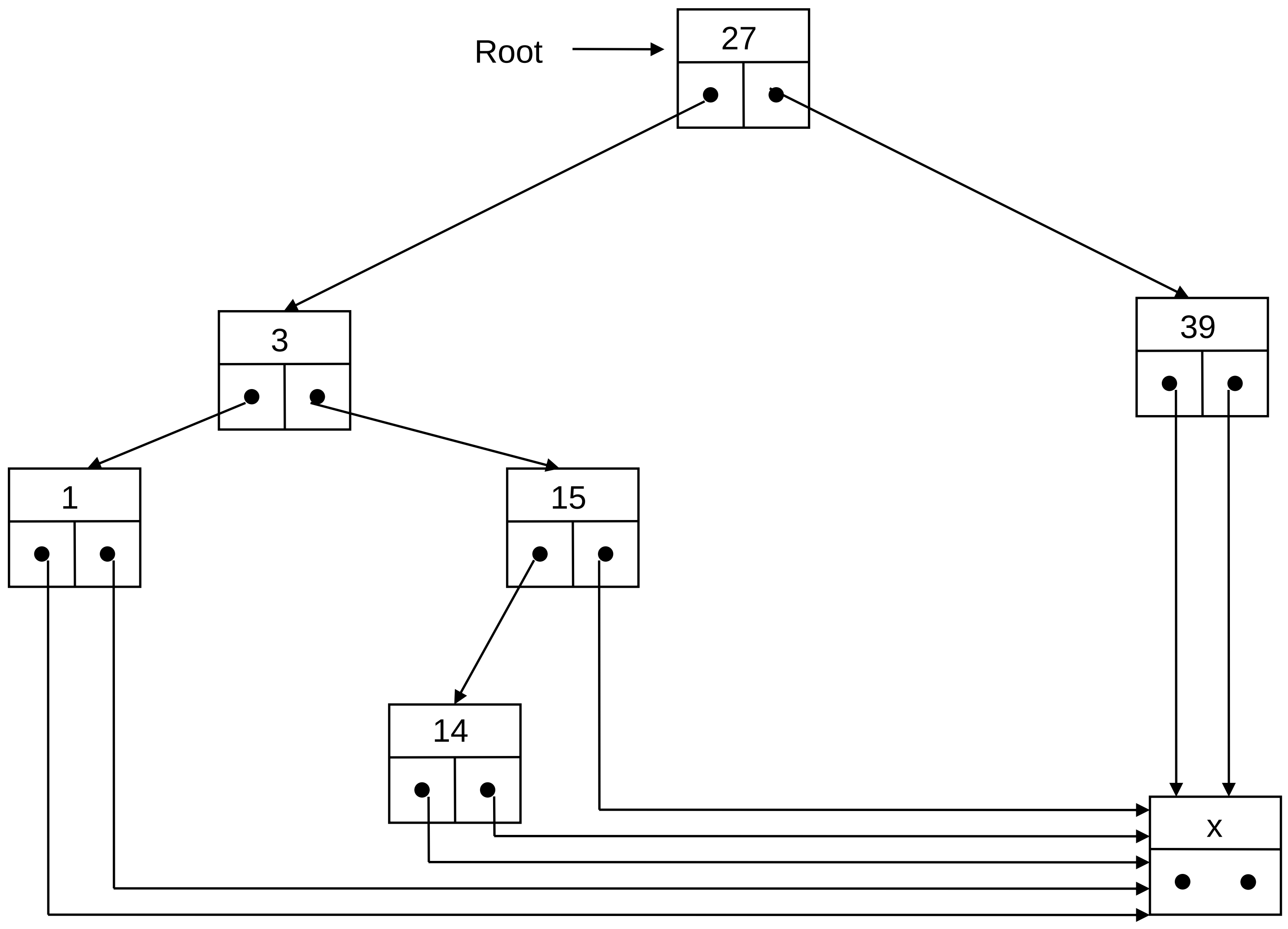
Standard trees (7)

Alternative tree structure:

- Instead of leaf ≅ *null*, set leaf ≅ pointer to a special “stop node” *b*
- For searching, store the search key *s* in *b* to save comparisons in internal nodes.

Use of a stop node for searching!

Example (with stop node)



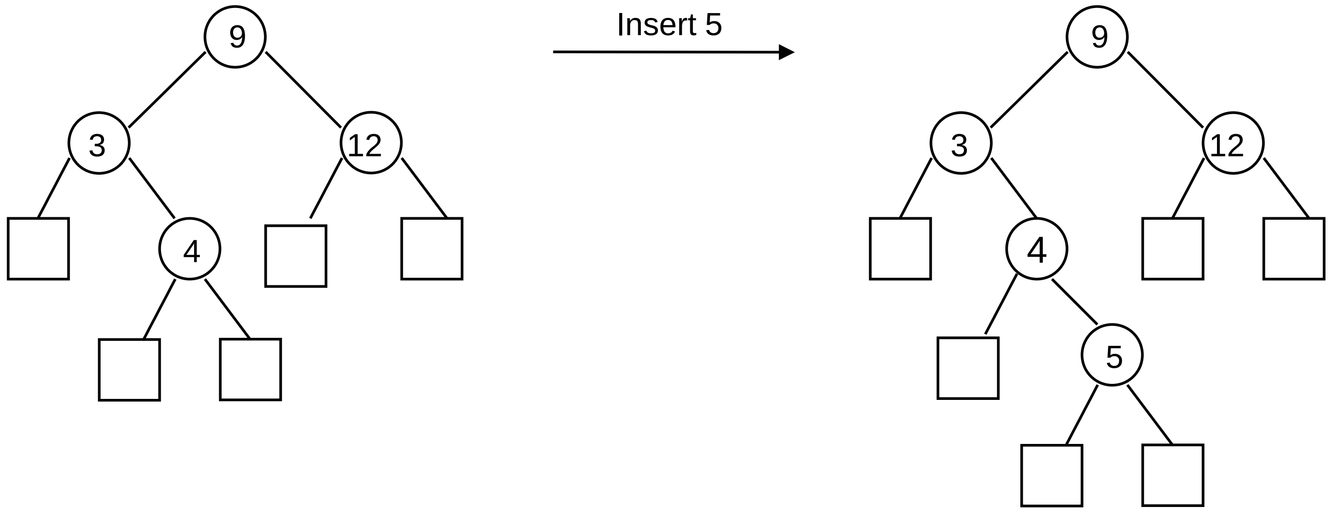
Standard trees (7)

Insertion of a node with key *s* in search tree *t*:

- Search for *s* ends in a node with *s*: don‘t insert (otherwise, there would be duplicated keys)
- Search ends in leaf *b*: make *b* an internal node with *s* as its key and two new leaves.

$\rightarrow$ tree remains a search tree!

Standard trees (8)



- Tree structure depends on the order of insertions into the initially empty tree
- Height can increase linearly, but it can also be in O(log *n*), more precisely $\lceil \log_2 (n+1) \rceil$.

Standard trees (9)

```
int height() {
    return height(root);
}
int height(SearchNode n){
    if (n == null) return 0;
    else return 1 + Math.max(height(n.left), height(n.right));
}
/* Insert c into tree; return true if successful
   and false if c was in tree already */
boolean insert (int c) {    // insert c
    if (root == null){
        root = new SearchNode (c);
        return true;
    } else return insert (root, c);
}
```

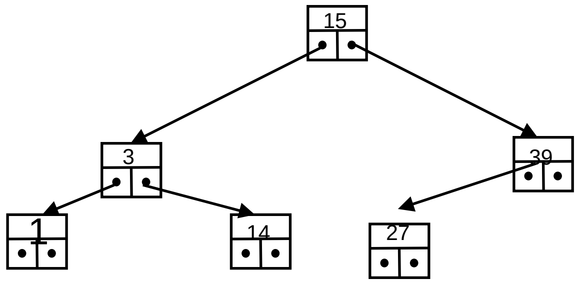
Standard trees (10)

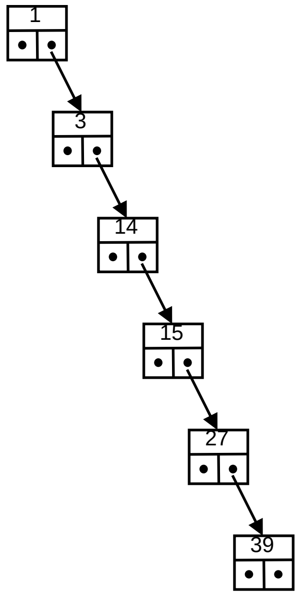
```
boolean insert (SearchNode n, int c){
    while (true){
        if (c == n.content) return false;
        if (c < n.content){
            if (n.left == null) {
                n.left = new SearchNode (c);
                return true;
            } else n = n.left;
        } else {    // c > n.content
            if (n.right == null) {
                n.right = new SearchNode (c);
                return true;
            } else n = n.right;
        }
    }
}
```

Special cases

The structure of the resulting tree depends on the order, in which the keys are inserted. The minimal height is $\lceil \log_2 (n + 1) \rceil$ and the maximal height is n.

Resulting search trees for the sequences 15, 39, 3, 27, 1, 14 and
1, 3, 14, 15, 27, 39:





Standard trees (11)

A standard tree is created by iterative insertions in an initially empty tree.

- Which trees are more frequent/typical: the balanced or the degenerate ones?
- How costly is an insertion?

Standard trees (11)

Deletion of a node with key *s* from a tree (while retaining the search tree property)

- Search for *s*.
  If search fails: done.
  Otherwise search ends in node *k* with *k*.*key* == *s* and

- *k* has no child, one child or two children:
  (a) no child: done (set the parent's pointer to *null* instead of *k*)
  (b) only one child: let *k*'s parent *v* point to *k*'s child instead of *k*
  (c) two children: search for the smallest key in *k*'s right subtree, i.e. go right and then
  to the left as far as possible until you reach *p* (the symmetrical successor of *k*);
  copy *p*.*key* to *k*, delete *p* (which has at most one child, so follow step (a) or (b))

Symmetrical successor

Definition: A node *q* is called the symmetrical successor of a node *p* if *q* contains the smallest key greater than or equal to the key of *p*.

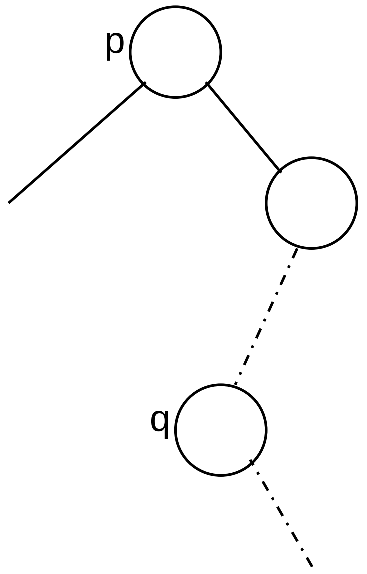
Observations:

- The symmetrical successor *q* of *p* is leftmost node in the right subtree of *p*.
- The symmetrical successor has at most one child, which is the right child.

Finding the symmetrical successor

Observation: If *p* has a right child, the symmetrical successor always exists.

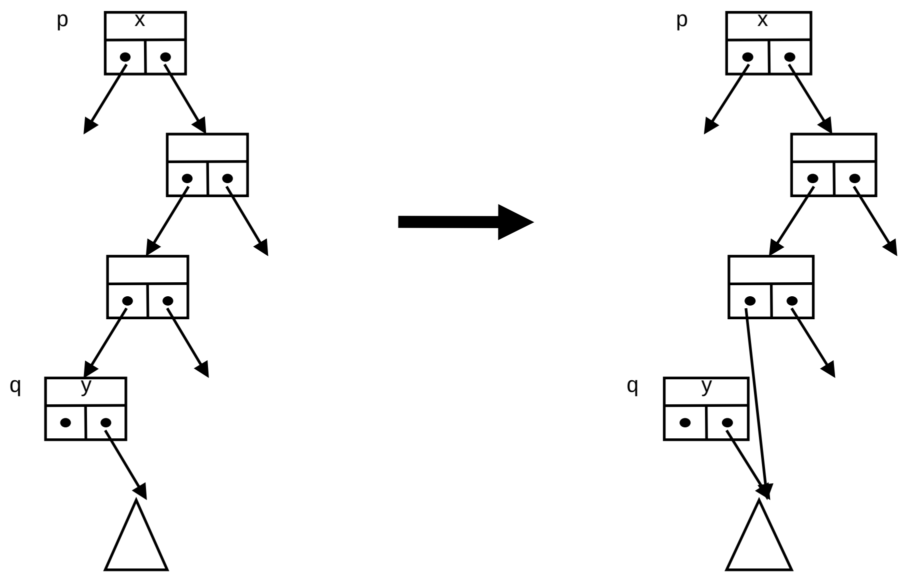
- First go to the right child of *p*.
- From there, always proceed to the left child until you find a node without a left child.



Idea of the *delete* operation

Delete *p* by replacing its content with the content of its symmetrical successor *q*. Then delete *q*.

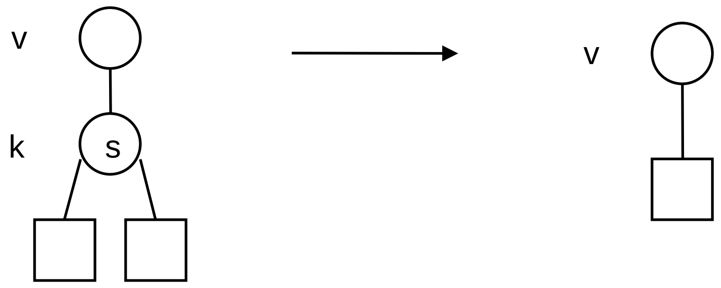
Deletion of *q* is easy because *q* has at most one child.



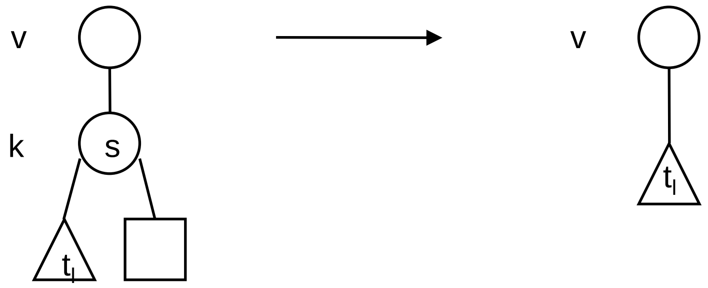
Examples

k has no internal child, one internal child or two internal children:

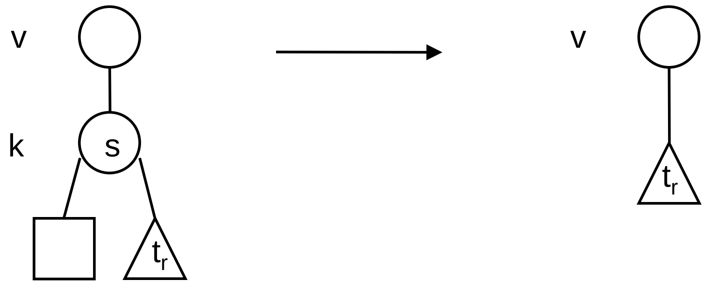
a)



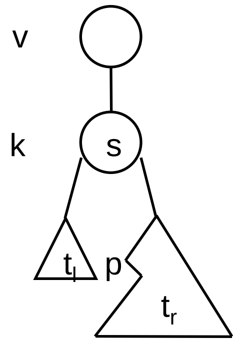
b)



c)



d)



Standard trees (12)

```
boolean delete(int c) {
    return delete(null, root, c);
}
// delete c from the tree rooted in n, whose parent is vn
boolean delete(SearchNode vn, SearchNode n, int c) {
    if (n == null) return false;
    if (c < n.content) return delete(n, n.left, c);
    if (c > n.content) return delete(n, n.right, c);
    // now we have: c == n.content
    if (n.left == null) {
        point (vn, n, n.right);
        return true;
    }
    if (n.right == null) {
        point (vn, n, n.left);
        return true;
    }
    // ...
```

Standard trees (13)

```
    // now n.left != null and n.right != null
    SearchNode q = pSymSucc(n);
    if (n == q) {   // right child of q is pSymSucc(n)
        n.content = q.right.content;
        q.right = q.right.right;
        return true;
    } else {        // left child of q is pSymSucc(n)
        n.content = q.left.content;
        q.left = q.left.right;
        return true;
    }
} // boolean delete(SearchNode vn, SearchNode n, int c)
```

Standard trees (14)

```
// let vn point to m instead of n;
// if vn == null, set root pointer to m
void point(SearchNode vn, SearchNode n, SearchNode m) {
    if (vn == null) root = m;
    else if (vn.left == n) vn.left = m;
    else vn.right = m;
}
// returns the parent of the symmetrical successor
SearchNode pSymSucc(SearchNode n) {
    if (n.right.left != null) {
        n = n.right;
        while (n.left.left != null) n = n.left;
    }
    return n;
}
```