Algorithms and complexity:

2 The dictionary problem: search trees
The dictionary problem

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 keys such that at least the following operations (methods) are supported:
• search (find, access)
• insert
• delete

Intuition: english-german dictionary
The dictionary problem (2)

- Search(id)
- Insert?
- Delete?

class ListNode {
    int id;
    string name;
    ListNode next;
}

string SequentialSearch (int k){
    n = first;
    while (n != null) {
        if (k == n.id) return n.name;
        n = n.next;
    }
    return "not found";
}
The dictionary problem (3)

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

- **the frequency** of the operations:
  - mostly insertion and deletion (dynamic)
  - mostly search (static)
  - approximately the same frequencies

- **other operations** to be implemented:
  - set operations: union, intersection, difference quantity, ...
  - enumerate the set in a certain order (e.g. ascending by key)

- **the complexity** of the solution: average case, worst case, amortized worst case

- **the place where the data is stored**: main memory, hard drive, WORM (write once read multiple)
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 are a generalisation of linked lists (each element can have more than one successor)

```
tree
  id
  name
  next
  ...
  id
  name
  next
  null
```
Trees as graphs (1)

Trees are

- special graphs:
  - in general, a graph \( G = (N, E) \) consists of a set \( N \) of nodes and a set \( E \) of edges
  - the edges are either directed or undirected
  - nodes and edges can be labelled
- a tree is a connected acyclic graph, where:
  \# nodes = \# edges + 1
- a general and central concept for the hierarchical structuring of information:
  - decision trees
  - code trees
  - syntax trees
Trees as graphs (2)

Several kinds of trees can be distinguished:

- undirected tree (with no designated root)

- rooted tree (one node 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 as graphs (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 as graphs (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
Examples

- tree
- not a tree
- not a tree (but two trees!)
Structural properties of trees

- **Depth of a node** $k$: number of 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:
  - $h($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.
  $\Rightarrow$ all leaves have the same depth.
Applications of trees

Use of trees for the dictionary problem:

- **node**: stores one key
- **tree**: stores a set of keys
- enumeration of the complete set of data
**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 $t_i$ 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 binary search trees (2)

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

```java
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 mode)

Search for key $s$ ends in the internal node $k$ with $k.key == s$ or in the leaf whose interval contains $s$
Standard binary search 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.
How is the search for key $s$ implemented in a leaf search tree? (leaf = node with 2 null pointers)

```java
k = root;
if (k == null) return false;
while (k.left != 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 binary search 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 SearchTree {
    SearchNode root;
    SearchTree () { // Constructor for empty tree
        root = null;
    }
    // ...
}
/* 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 binary search trees (7)

Alternative tree structure:

- instead of leaf \( \approx \text{null} \), set leaf \( \approx \) 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 mode)
Insertion of a node with key $s$ in search tree $t$.

Search for $s$:

1. search for $s$ ends in a node with $s$: don‘t insert (otherwise, there would be duplicated keys)

2. search ends in leaf $b$: make $b$ an internal node with $s$ as its key and two new leaves.

$\implies$ tree remains a search tree!
Standard binary search 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 $[\log_2 (n+1)]$.
Standard binary search 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) {
    if (root == null) {
        root = new SearchNode (c);
        return true;
    } else return insert (root, c);
}
Standard binary search trees (10)

```java
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:
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?
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.
**Example**

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

- **a)**
  - $k$ has a right child $s$, but no left child.
  - $v$ is the root.

- **b)**
  - $k$ has a right child $s$ and a left child $t_l$.
  - $v$ is the root.

- **c)**
  - $k$ has a right child $s$, a left child $t_l$, and a right child $t_r$.
  - $v$ is the root.

- **d)**
  - $k$ has a right child $s$, a left child $t_l$, and a right child $t_r$.
  - $v$ is the root.
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;
    }
    // ...
}
// now n.left != null and n.right != null
SearchNode q = pSymSucc(n);
if (n == q) { // right child of q is SymSucc(n)
    n.content = q.right.content;
    q.right = q.right.right;
    return true;
} else { // left child of q is SymSucc(n)
    n.content = q.left.content;
    q.left = q.left.right;
    return true;
}
// boolean delete(SearchNode vn, SearchNode n, int c)

// 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;
}
Standard binary search 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;
}