The Dictionary Problem: Search Trees

Summer Term 2011

Jan-Georg Smaus

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 \( E \subset V \times V \) 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:
  - all vertices \( k \) have 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 (2)

Several kinds of trees can be distinguished:
- Undirected tree (with no designated root)
- Rooted tree (one node [= vertex] is designated as the root)
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:
  - the children of each node are somehow ordered, i.e., there is the "leftmost child", the "second child from the left", ..., the "rightmost child".
  - In the graphical representation of a tree, this is a-priori not the case. We have to explicitly state it.

- Binary tree: ordered tree of order 2; the children of a node (if there are 2) 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 \( (d \geq 1) \):

Consider a set of nodes \( V \).
- Each node in \( V \) is in \( M_d \)
- \( L_1, \ldots, L_d \in M_d \) and \( w \) a node in \( V \). Then \( w \) with the roots of \( L_1, \ldots, L_d \) as children (from left to right) is a tree \( t \in M_d \). The \( L_i \) are subtrees of \( t \).
- According to this definition each node has rank \( d \) (or rank 0).

Nodes of binary trees either have 0 or 2 children.
- Variation of the definition: allowing for rank \( \leq d \).
  - For binary trees, nodes with exactly 1 child could then also be permitted.

Illustration of the Definition

Examples

- tree
- not a tree (but two trees!)

Note that we chose to depict inner nodes as circles and leaves as boxes.

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:
  - \( h(\text{leaf}) = 0 \)
  - \( h(t) = 1 + \max\{h(t_i) \mid \text{root of } t_i \text{ is a child of the root of } t \} \)
  - \( t_i \) 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.

Labelled Vertices

- We mentioned that vertices (and edges) in a graph may be labelled, but in the above definitions on trees, we did not talk about labels yet – the set of nodes \( V \) was a "black box".
- We now consider trees where either the inner nodes or the leaves or both are labelled.
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 Binary Search 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)

```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

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 \( s_l \leq s_k < s_r \) (\( s_l \): key in left subtree, \( s_k \): router in \( k \), \( s_r \): key in right subtree)
- Choice of \( s \): maximum key in \( t_l \).
- Alternative convention (less common): require \( s_l < s_k \leq s_r \) and choose \( s \) as minimum key in \( t_r \).
Example: leaf search tree

Leaf nodes store keys, internal nodes contain routers.

```
1
/ \
5 6
```

23.05.2011 Theory 1 - Search Trees

Standard Binary Search Trees (4)

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

From now on ...

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

Standard Binary Search Trees (5)

```java
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 Binary Search Trees (6)

```java
/* 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)

**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.

Tree remains a search tree!

Standard Binary Search Trees (8)

`
// Insert c into tree; return true if successful and false if c was in tree already` `*`
```java
boolean insert (int c) { // insert c
    if (root == null){
        root = new SearchNode (c);
        return true;
    } else return insert (root, c);
}
```

Standard Binary Search Trees (9)

```java
int height(){
    return height(root);
}
```
```java
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 Binary Search Trees (10)

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

We will address these questions in the next chapter.
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 the 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.

Illustration
- $k$ has no internal child or one internal child:
  - a) 
  - b) (left) 
  - b) (right) 
- $k$ has two internal children:
  - c)
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 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)

// 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;
}