Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe and Sarah Winkler

6 May 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 7

Exercise 1 (Binary Search)

a) Write a function in C which performs binary search in a given array. Try to implement the algorithm with a loop, as opposed to the recursive approach shown in the lecture. Use the type `short int` for the array as well as for the variables `left`, `right` and `middle` and compute the midpoint with the following commands:

```
middle = left + right;
middle = middle / 2;
```

b) Create a dynamic array to test your implementation: let the user enter the array size n, allocate memory respectively and fill the array with values $0, 1, 2, \ldots, n-1$.

c) Now test your program by creating a large array, e.g. with size $n = \texttt{SHRT\_MAX} - 1$ (the latter is defined in `limits.h`), and search for the last value in the array. What happens?

Solution idea: To avoid an overflow, compute the midpoint as $\text{left} + \lfloor(\text{right} - \text{left})/2\rfloor$ instead of $\lfloor(\text{right} + \text{left})/2\rfloor$.

Exercise 2 (Cyclically Sorted Sequences)

A sequence $x_1, \ldots, x_n$ is called *cyclically sorted* if there exists some index i such that the list $x_i, x_{i+1}, \ldots, x_n, x_1, \ldots, x_{i-1}$ is weakly increasing, i.e. it holds that

$$x_i \leq x_{i+1} \leq \ldots \leq x_n \leq x_1 \leq \ldots \leq x_{i-1}$$

Provide a pseudo code function which, given a cyclically sorted integer array `clist` of length `n`, computes the index `i` of the minimal element. The algorithm should have time complexity $O(log(n))$.

Solution: We assume that all values occur at most once (otherwise doubles have to be eliminated first).

Using a variant of binary search to solve this exercise, we can maintain two variables `left=0` and `right=n-1` which specify the search interval. As usual, one computes `mid = left + (right - left) / 2`. If `clist[mid] <= clist[right]`, the minimal element must be between `left` and `mid`. Thus we set `right=mid`, otherwise `left=mid+1`. With this approach the search interval is decreased until a single element is left. Since the search space is halved in every iteration, the time complexity is in $O(log(n))$.

Exercise 3 (Deletion in Binary Search Trees)

Consider binary trees as described by the following record:

Listing 1 Record describing a binary tree.

```
record btree =
begin
   key : integer;
   data : ...;
   left, right : ^btree;
end
```

Use pseudo code to describe an algorithm that deletes the element associated with a certain key from a tree. You may assume that there are no duplicate keys and exclude the case where the element that is to be removed occurs at the root.

Solution: See Listing 2.

Exercise 4 (Binary Search Trees)

In this exercise you have to implement the data structure and basic operations for binary search trees in C.

a) Define a `struct` specifying a binary search tree as described in Listing 1. The type of a node's data may be chosen freely.

b) Provide a function `getData` which checks whether a given key occurs in the tree and returns the respective data.

c) Write a function `insert` to add a new element if the respective key does not yet occur.

d) Implement a function `delete` to remove an element from the tree. Try to consider the case where the root gets deleted as well.

e) What is the time complexity of these operations if i) the inserted elements are distributed randomly, ii) the elements are inserted in increasing order?

Solution: See the program `btree.c`.

Listing 2 Deletion in binary search trees.

Input: pointer to the root of a binary search tree T, key x

Output: *false* if x does not occur, *true* otherwise. The tree is altered such that the element with key x is deleted if it exists.

```
begin
  N := T;                                          /* search for x in tree */
  while N ≠ nil and N^.key ≠ x do
    P := N;
    if x < N^.key then
      N := N^.left;
    else
      N := N^.right;
  if N = nil then
    return false;                                  /* fail if x was not found */
  if N ≠ T then                                    /* case 1: N has no left child */
    if N^.left = nil then
      if x ≤ P^.key then
        P^.left := N^.right;
      else
        P^.right := N^.right;
    else if N^.right = nil then                    /* case 2: N has no right child */
      if x ≤ P^.key then
        P^.left := N^.left;
      else
        P^.right := N^.left;
    else                                           /* case 3: N has two children */
      N1 := N^.left;                               /* search tree for predecessor N1 of N */
      P1 := N;
      while N1^.right ≠ nil do
        P1 := N1;
        N1 := N1^.right;                           /* N1 is predecessor of N */
      P1^.right := N1^.left;                       /* delete N1 */
      N^.key = N1^.key;                            /* copy content of N1 to N */
      N^.data = N1^.data;
  return true
end
```