university of groningen

Algorithms and Data Structures in C

answers exam 11 April 2014

Gerard R. Renardel de Lavalette

problem 1

This problem is about binary trees defined by the following type definition:

```
typedef struct TreeNode *Tree;

struct TreeNode {
  int item;
  Tree leftChild, rightChild;
};
```

a. When is a binary tree a search tree?

b. Define the C function with prototype

```
Tree addInSearchTree(Tree t, int n);
```

that adds `n` to search tree `t` (provided `n` does not occur in `t`) while preserving the search tree property. When `n` occurs in `t`, the returned tree is equal to the input tree.

c. Define the C function with prototype

```
Tree removeFromSearchTree(Tree t, int n);
```

that removes `n` from `t` (provided `n` occurs in `t`) while preserving the search tree property. When `n` does not occur in `t`, the returned tree is equal to the input tree. You may *use* the function with prototype

```
int successor(Tree t);
```

(you do not have to *define* this function). Precondition for the function `successor` is that `t` has a right child. The function `successor` returns the smallest integer `m` in the subtree that has the right child of `t` as root, and it removes the node containing `m`.

problem 1a

a. When is a binary tree a search tree?

A binary tree (containing integers in its nodes) is a search tree when it satisfies the search tree property:

> All nodes k with a value x satisfy: all values in the left subtree of k are smaller than x, and all values in the right subtree of k are greater than x.

At the exam, about 80 % gave the **wrong** answer:

> All nodes k with a value x satisfy: if k has a left child, its value is smaller than x, and if k has a right child, its value is greater than x.

problem 1b

```
Tree addInSearchTree(Tree t, int n) {
  if (t == NULL) {
    t = malloc(sizeof(struct TreeNode));
    assert(t != NULL);
    t->item = n;
    t->leftChild = NULL;
    t->rightChild = NULL;
    return t;
  }
  if (n < t->item) {
    t->leftChild = addInSearchTree(t->leftChild,n);
  } else if (t->item < n) {
    t->rightChild = addInSearchTree(t->rightChild,n);
  }
  return t;
}
```

problem 1c

```
Tree removeFromSearchTree(Tree t, int n) {
  Tree t1;
  if ( t == NULL ) return NULL;
  if ( t->item < n ) {
    t->rightChild = removeFromSearchTree(t->rightChild,n);
    return t;
  }
  if ( n < t->item ) {
    t->leftChild = removeFromSearchTree(t->leftChild,n);
    return t;
  }
  if ( t->rightChild == NULL ) {
    t1 = t->leftChild;
    free(t);
    return t1;
  }
  t->item = successor(t);
  return t;
}
```

problem 2

The C code below defines types and functions for the implementation of lists of integers. However, there are 4 errors in the code so that functions do not work properly and/or memory leaks may occur. Find these errors, indicate what is wrong and repair them.

```c
typedef struct ListNode *List;

struct ListNode {
  int item;
  List next;
};

List addItem(int n, List li) {
  List newList = malloc(sizeof(struct ListNode));
  assert(newList!=NULL);
  newList->item = n;
  newList->next = li;
  return newList;
}
```

problem 2

```
List removeFirstNode(List li) {
  List returnList;
  if ( li == NULL ) {
    printf("list␣empty\n");
    abort();
  }
  returnList = li->next;
  free(li);
  return returnList;
}
```

problem 2

```
27 List insertInOrder(List li, int n) {
28 /* li is sorted in ascending order */
29   List li1;
30   if ( li->item > n || li == NULL  ) { /* ERROR 1: wrong order */
31     return addItem(n,li);
32   }
33   li1 = li;
34   while ( li1->next != NULL && (li1->next)->item < n ) {
35     li1 = li1->next;
36   }                                    /* ERROR 2: see below */
37   return li;
38 }
```

add between 36 and 37: `li1->next = addItem(n,li1->next);`

alternative: replace 33 to 36 by `li1->next = insertInOrder(li1->next,n);`

problem 2

```
39 int removeLastOcc(List *lp, int n) {
40 /* NB: lp is a reference pointer!
41  * the function removes the last occurrence of n from *lp
42  * it returns 1 when an occurrence of n has been removed,
43  * otherwise 0
44  */
45   if ( *lp == NULL ) {
46     return 0;
47   }
48   if ( (*lp)->item == n ) {  /* ERROR 3: swap 48-51 with 52-54 */
49     *lp = removeFirstNode(*lp);
50     return 1;
51   }
52   if ( removeLastOcc(&((*lp)->next),n) ) {
53     return 1;
54   }
55   return 0;
56 }
```

problem 2

```
58 List removeAllOcc(List li, int n) {
59 /* remove all occurrences of n and return the resulting list */
60   if ( li == NULL ) {
61     return NULL;
62   }
63   if ( li->item == n ) {
64     return removeAllOcc(li->next,n);  /* ERROR 4: see below */
65   } else {
66     li->next = removeAllOcc(li->next,n);
67     return li;
68   }
69 }
```

Memory leak! replace 64 by `return removeAllOcc(removeFirstNode(li),n);`

problem 3

This problem is about tries.

a. Let W be a collection of words. Define: T is a standard trie for W.

b. Describe in pseudocode an algorithm to search for a word in a trie.

c. Explain what a suffix trie is, and how it can be used to search for a pattern in a text.

problem 3a

Let W be a collection of words. Define: T is a standard trie for W.

- The root is empty, and every other node contains a letter;
- the children of a node contain different letters and are in alphabetical order;
- the branches from the root correspond exactly with the words in W.

problem 3b

Describe in pseudocode an algorithm to search for a word in a trie.

```
algorithm Search(T,w)
    input standard trie T, word w
    output Yes if w occurs in T, otherwise No
    k ← root of T
    while w not empty do
        x ← first letter of w
        w ← w minus x
        if k has no child containing x then
            return No
        k ← child of k that contains x
    if k is a leaf then
        return Yes
    else
        return No
```

Explain what a suffix trie is, and how it can be used to search for a pattern in a text.

A suffix trie for a text T is a trie for the collection S of suffixes (end segments) of T. Searching for a pattern in T can be done by applying a modification of the algorithm of 3b to the suffix trie. The modification consists in replacing the last 4 lines by

return Yes

As a consequence, the algorithm searches whether w is the prefix of a word in the trie. We use the following fact:

every pattern (substring) of a string is the prefix of a suffix.

Consider the following algorithm:

algorithm BreadthFirstSearch(G,v)
  input connected graph G with node v;
    all nodes and edges are unlabeled
  result labeling of the edges of G with NEW and OLD;
    the edges with label NEW form a spanning tree of G,
    and all nodes are visited (and labeled VISITED)
  . . .

```
. . .
give v the label VISITED
create an empty queue Q
enqueue(v)
while Q not empty do
    u ← dequeue()
    forall e incident with u do
        if e has no label then
            w ← the other node incident with e
            if w has no label then
                give e the label NEW
                give w the label VISITED
            else
                give e the label OLD
```

problem 4

a. What is a *spanning tree* of a connected graph?

b. The algorithm contains one error. Indicate what the error is and repair it.

c. Modify the corrected algorithm into an algorithm FindPath(G,v,w) that finds a path from v to w in graph G.

d. Argue that the path found by FindPath has minimal length. Here the length of a path is the number of edges in it.

problem 4a

What is a *spanning tree* of a connected graph?

A spanning tree of a connected graph is

- a subgraph of that graph,
- that contains all nodes of the graph, and
- is a tree (i.e. connected and without cycles).

problem 4b

The algorithm contains one error. Indicate what the error is and repair it.

"enqueue(w)" is missing:

```
...
    w ← the other node incident with e
    if w has no label then
        give e the label NEW
        give w the label VISITED
        enqueue(w)
    else
        give e the label OLD
```

algorithm FindPath(G,v,w)
    input connected graph G with nodes v and w
    output a stack containing the nodes on a path from v to w
    if v=w **then**
        return stack containing only v
    . . .
                x ← the other node incident with e
                if x = w **then**
                    S ← empty stack
                    push w on S
                    while x ≠ v **do**
                        x ← parent(x)
                        push x on S
                    return S
                if w has no label **then**
                    parent(x) ← u
                    . . .

problem 4d

Argue that the path found by FindPath has minimal length. Here the length of a path is the number of edges in it.

The algorithm FindPath first finds all nodes x one step from v. For these nodes (v,x) is a shortest path, and they are all shortest paths from v with length 1.

Then it finds all the 'new' nodes y that are one step from the nodes at distance 1. These nodes obtain a path (v,x,y). There is no path (v,y) for y is a 'new' node. So (v,x,y) is a shortest path. The paths thus obtained are all shortest paths from v with length 2.

And so on.

Conclusion: every path found by FindPath is a shortest path from v.