Exam Algorithms and Data Structures in C Friday, April 8, 2016, 9 - 12 h.

Your personal information

Name:	
Student Number:	
Study Program:	

Exam instructions
– please read carefully –

- Please write your name, student number, and study program on this page.
- The first part of the exam consists of 10 multiple choice questions. Read carefully each question and the four options, and select the option that answers the question correctly. When more than one option answers the question correctly, select the most informative option.
  Provide your answers for the multiple choice questions **in the table below**.
- The second part of the exam consists of 2 open problems. Write your answers in the boxes below the problems. If you need additional space, use the last page of the exam, and provide a reference to the problem.
  Please work out your answer on scratch paper, and only write the final version on the exam.
- Your exam grade is computed as follows. For every correct answer to the multiple choice questions, you will earn 4 points. For each of the 2 open problems you can earn maximally 30 points. Your exam grade is $p/10$ where p is the number of points you earned.

This is a version of the exam with answers. The correct answers to the multiple choice questions are printed in boldface.

Part I: Multiple Choice Questions

1. On an empty queue, the following actions are performed:

   `enqueue(8); enqueue(3); dequeue(); enqueue(2); enqueue(dequeue());`
   `dequeue(); enqueue(5); dequeue(); dequeue();`

   What is the result of the last `dequeue()`?

   A. 2.

   B. 3.

   C. 5.

   D. 8.

2. The function `insertInOrder()` inserts a number in an increasingly ordered list. When the number already occurs in the list, the function does nothing.
   Consider the following code for `insertInOrder`:

```
 1 list insertInOrder(list li, int n) {
 2   List newList = NULL;
 3   if ( ??? ) {
 4     List newList = malloc(sizeof(struct ListNode));
 5     assert(newList!=NULL);
 6     newList->item = n;
 7     newList->next = li;
 8     return newList;
 9   }
10   if ( li->item < n ) {
11     li->next = insertInOrder(li->next,n);
12   }
13   return li;
14 }
```

   What should replace the condition `???` in line 3 in order to obtain a correct definition of `insertInOrder`?

   A. `li!=NULL && n <= li->item`

   B. `li!=NULL && n < li->item`

   C. `li==NULL || n <= li->item`

   D. `li==NULL || n < li->item`

3. Consider the following grammar:

   $\langle code \rangle$ ::= $\langle number \rangle$ $\langle word \rangle$.

   $\langle number \rangle$ ::= { $\langle digit \rangle$ } .

   $\langle word \rangle$ ::= $\langle capital \rangle$ { $\langle lowercase \rangle$ } .

   $\langle capital \rangle$ ::= 'A' | 'B | 'C' .

   $\langle lowercase \rangle$ ::= 'a' | 'b' | 'c' .

   $\langle digit \rangle$ ::= '0' | '1' | '2' | '3' .

   Which string is **not** a production of $\langle code \rangle$?

   A. 223Ccca

   B. 3212b

   C. 1B

   D. Abc

4. Which of the following is a **correct** statement about binary trees T?

   A. The number of edges in T is equal to the number of nodes plus one.

   B. The number of leaves is even.

   C. In postorder traversal, the root is visited last.

   D. Every node except the root has a sibling.

5. When implementing a search tree in C, the pointer representation is generally preferred over the array representation. Which of the following statements is a **correct** argument for this preference?

   A. The array representation may lead to inefficient memory usage.

   B. The pointer representation admits recursive functions.

   C. The array representation leads to programs with higher time complexity.

   D. All of the above.

6. Let T be a text with length n, and let ST be the suffix trie for text T. Moreover, let P be a pattern with length m. Which of the following statements is **correct**?

   A. If P occurs in T, there is a node in ST containing P.

   B. P occurs in T if and only if there is a path in ST from the root to a leaf that corresponds with P.

   C. ST has n leaves and at most $2n$ nodes.

   D. ST enables checking whether P occurs in T in $\mathcal{O}(1)$ time.

7. This question is about the algorithm Downheap for heaps where the largest value is in the root. Some occurrences of the node variables v, lc, rc are replaced by X, Y, Z.

```
algorithm Downheap(v)
    input : node v in a heap, with possibly a conflict
        with the heap order between v and its children
    result : heap order is restored
    if v has at least one child then
        lc ← the left child of v
        rc ← the right child of v (or lc, when v has no right child)
        if (value of lc) > (value of v) and (value of lc) > (value of rc) then
            swap the values of X and Y
            Downheap(X)
        else if (value of rc) > (value of v) then
            swap the values of Y and Z
            Downheap(Z)
```

How to replace X, Y, Z in order to obtain a correct algorithm?

A. X by rc, Y by lc, Z by v.

B. X by rc, Y by v, Z by lc.

C. X by lc, Y by rc, Z by v.

D. X by lc, Y by v, Z by rc.

8. Let ST be the standard trie for the collection $W = \{\text{her, his, there, this}\}$, and CT the compressed trie for W. Which of the following statements about the number of nodes in ST and CT is **correct**?

A. ST contains 12 nodes, CT contains 7 nodes.

B. ST contains 13 nodes, CT contains 7 nodes.

C. ST contains 12 nodes, CT contains 8 nodes.

D. ST contains 13 nodes, CT contains 8 nodes.

9. Which of the following statements about graphs is **wrong**?

A. If a graph contains less edges than nodes, it is not connected.

B. Every cycle in a graph is a path.

C. If a graph is connected and has no cycles, it is a tree.

D. If all nodes in a graph have an odd degree, then the number of nodes is even.

10. This question is about the application DFS(G,v) of the algorithm Depth-First Search to graph G and node v in G. Which of the following statements is **wrong**?

A. If G is connected, then DFS visits all edges.

B. If G is connected, then DFS visits all nodes.

C. If DFS visits all nodes, then G is connected.

D. If DFS visits all edges, then G is connected.

Part II: Open Questions

1. 30 points The type `Tree` is defined by

```
typedef struct TreeNode *Tree;

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

a. Define a C function with prototype `Tree buildPerfectTree(int h)` that, given a non-negative integer h, returns a perfect binary tree with height h in which every node contains an integer that indicates the depth of that node.
So e.g. `buildPerfectTree(2)` yields a binary tree of height 2 with seven nodes: the root contains 0, its children contain 1 and the other four nodes contain 2.
Hint: use an auxiliary function.

b. Define a C function with prototype `Tree onlyLeft(Tree tr)` that, given a binary tree tr, returns the subtree of tr that corresponds with the leftmost branch in tr. All nodes in tr that are not in the result are to be freed.
So e.g. `onlyLeft(buildPerfectTree(2))` yields a tree with 3 nodes: the root contains 0 and only has a left child, which contains 1 and only has a left child that is a leaf and contains 2.

Solution: a.

```
Tree buildPerfectTreeAux( int h, int n ) {
  Tree new = NULL;
   if ( h >= n ) {
    new = malloc(sizeof(struct TreeNode));
    assert(new != NULL);
    new -> item = n;
    new->leftChild = buildPerfectTreeAux(h,n+1);
    new->rightChild = buildPerfectTreeAux(h,n+1);
  }
  return new;
}

Tree buildPerfectTree( int h ) {
  return buildPerfectTreeAux(h,0);
}
```

b. There are two readings possible of the term *leftmost branch*, leading to diferent solutions.

- The branch containing the root and only left children. Here the answer is

```
Tree onlyLeft( Tree tr) {
  if ( tr != NULL ) {
    freeTree(tr->rightChild);
    tr->rightChild = NULL;
    tr->leftChild = onlyLeft(tr->leftChild);
  }
  return tr;
}
```

- The branch containing the root, left children when present, and right children only in case the parent in the branch has no left child. Here the answer is

```
Tree onlyLeft( Tree tr) {
  if ( tr == NULL ) {
    return NULL:
  }
  if ( tr->leftChild != NULL ) {
    freeTree(tr->rightChild);
    tr->rightChild = NULL;
    tr->leftChild = onlyLeft(tr->leftChild);
  } else if ( tr->rightChild != NULL ) {
    tr->rightChild = onlyLeft(tr->rightChild);
  }
  return tr;
}
```

Both solutions use

```
void freeTree(Tree t) {
  if ( t != NULL ) {
    freeTree(t->leftChild);
    freeTree(t->rightChild);
    free(t);
  }
  return;
}
```

2. $\boxed{\text{30 points}}$ a. Define in pseudocode an algorithm ShortestPathLength that determines the length of a shortest path between two (possibly equal) nodes in a connected simple graph G. The length of a path is defined as the number of edges in it.
Hint: consider a variant of Breadth-First Search. Do not forget to specify the input and the output of the algorithm.

b. Indicate the time complexity of the algorithm, and motivate your answer.

Solution: a.

```
algorithm ShortestPathLength(G,v,w)
    input connected simple graph G with nodes v and w
    output length of a shortest path in G between v and w
    if v = w then
        return 0
    give v the label VISITED
    create an empty queue Q
    enqueue(v,0)
    while Q not empty do
        (u,n) ← dequeue()
        forall nodes x adjacent to u do
            if x = w then
                return n+1
            if x is unlabeled then
                give x the label VISITED
                enqueue (x,n+1)
```

b. The algorithm enqueues and dequeues every node once: these actions each take $\mathcal{O}(1)$ time for each node. Moreover, the algorithm processes each edge twice: once from each incident node. Both times, the algorithm performs some actions that are in $\mathcal{O}(1)$. So the time complexity is $\mathcal{O}(n+m)$ where n is the number of nodes and m is the number of edges. (Since G is connected, we know that $n+1 \leq m$, so n is in $\mathcal{O}(m)$ and the time complexity can be written as $\mathcal{O}(m)$.)