Exam Algorithms and Data Structures in C Monday, May 14, 2018, 18:30 - 21:30 h.

Your personal information

Name:	
Student Number:	
Study Programme:	

Exam instructions
– please read carefully –

- Please write your name, student number, and study programme 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. Read carefully each problem and write your answers in the boxes below the problems.
  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 stack (with operations `push` and `pop`) and an empty queue (with operations `enqueue` and `dequeue`), the following actions are performed:

   ```
   push(1); enqueue(2); push(3); enqueue(4);
   push(dequeue()); enqueue(pop()); push(pop());
   enqueue(dequeue()); dequeue()
   ```

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

   A. 1.

   B. 2.

   C. 3.

   D. 4.

2. The function `findInList()` is defined by

   ```
   int findInListIt(List li, int n) {
     while ( ??? ) {
       li = li->next;
     }
     if ( li == NULL ) {
       return 0;
     } else {
       return 1;
     }
   }
   ```

   Which condition should replace the `???` part in line 2 in order to obtain a correct function?

   A. `li != NULL || li->item != n`

   B. `li->item != n || li != NULL`

   C. `li != NULL && li->item != n`

   D. `li->item != n && li != NULL`

3. Suppose an empty search tree is given. We add (in the order indicated, from left to right) the following integers:

   7 9 8 2 6 4 10 1 5 3

   In which order do we obtain these integers when we traverse the search tree in postorder traversal?

   A. 1 3 5 4 6 2 8 10 9 7

   B. 1 2 3 4 5 6 7 8 9 10

   C. 7 2 1 6 4 3 5 9 8 10

   D. 7 9 8 2 6 4 10 1 5 3

4. Which of the following statements about search trees T with height h containing integers is **wrong**?

A. Inorder traversal of T yields the integers in ascending order.

B. T contains more that $2h$ nodes.

C. Deciding whether integer x occurs in T can be done in $\mathcal{O}(h)$ time.

D. Let node l be a descendant of the right child of node k. Moreover, k contains integer x and l contains integer y. Then $x < y$.

5. Consider the following grammar. (Recall: `[ ]` are used for '0 or 1 time', `{ }` are used for '0 or more times'.)

$\langle code \rangle$ ::= $\langle word \rangle$ [$\langle num \rangle$ | $\langle word \rangle$] $\langle num \rangle$.

$\langle word \rangle$ ::= { $\langle letter \rangle$ } .

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

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

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

Which string can **not** be produced by $\langle code \rangle$?

A. ACB21

B. C2

C. 23AC

D. 132

6. We have a heap which contains the integers 0, 1, 2, ..., 98, 99. We consider the pointer representation PR and the array representation AR of this heap. Which of the following statements is **true**?

A. The height of PR is 6 and the size of AR is 100.

B. The height of PR is 6 and the size of AR is 101.

C. The height of PR is 7 and the size of AR is 100.

D. The height of PR is 7 and the size of AR is 101.

7. Which statement about the algorithm Heapsort is **true**?
(Recall: when more than one statement is true, select the most informative option.)

A. Its time complexity is $\mathcal{O}(n^2)$.

B. Its time complexity is $\mathcal{O}(n\sqrt{n})$.

C. Its time complexity is $\mathcal{O}(n\log(n))$.

D. Its time complexity is $\mathcal{O}(n)$.

8. Let a collection W of $m > 1$ words be given, with n the sum of the lengths of the words. Let ST be a standard trie for W, and CT a compressed trie. So in ST every node (except the root) contains a letter, and in CT every node (except the root) contains a nonempty string.
Which of the following statements is **wrong**?

A. CT contains no nodes with branching degree 1, while ST may contain such nodes.

B. The number of nodes in ST is in $\mathcal{O}(n)$, while CT has at most $2m$ nodes.

C. The memory required for ST is in $\mathcal{O}(n)$, while for CT it is in $\mathcal{O}(m)$.

D. ST and CT have the same number of leaves.

9. We consider simple undirected graphs G that contain 10 nodes. Recall that a graph is simple when it has no loops and no parallel edges. Which of the following statements is **true** for at least one such graph?

   A. G contains a cycle with length 3 in which all nodes have degree 3.

   B. G is connected and contains 8 edges.

   C. G contains 90 edges.

   D. For every n with $1 \leq n \leq 10$ there is a node in G with degree n.

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

   A. In a connected graph, there is a path between every two nodes.

   B. If all nodes in a graph have an even degree, there is an Euler cycle.

   C. The number of nodes with odd degree is even.

   D. A cycle is a path with identical first and last node.

Part II: Open Questions

1. 30 points This problem is about linked lists. Their C type reads

```
typedef struct ListNode* List;

struct ListNode {
  int item;
  List next;
};
```

a. (15 points) Define an efficient C function with prototype `List addAfterEven(List li)`. It adds after every node in `li` that contains an even number `n`, a new node with the number `n+1`. Do not make a copy of the input list, but modify and return the input list. When you use an auxiliary function, define it, too.

Solution:

```
List addAfterEven ( List li ) {
  if ( li == NULL ) {
    return NULL;
  }
  li->next = addAfterEven(li->next);
  if ( li->item % 2 == 0 ) {
    List newList = malloc(sizeof(struct ListNode));
    assert( newList != NULL );
    newList->item = li->item + 1;
    newList->next = li->next;
    li->next = newList;
  }
  return li;
}
```

b. (15 points) This problem is about binary trees. Their C type reads

```
typedef struct TreeNode *Tree;

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

We call a node `nod` in a binary tree *special* when the following holds:

> `nod` is the right child of its parent node;
> `nod` has a sibling (the left child of the parent node) `sib` with `nod.item == sib.item`.

Define an efficient C function with prototype `Tree removeNodes(Tree tr)` that removes all special nodes in `tr`. When a special node is removed, all nodes below it are removed, too. Do not make a copy of the input tree, but modify and return the input tree. When you use an auxiliary function, define it, too. See to it that no memory leaks occur.

Solution:

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

Tree removeNodes ( Tree tr ) {
  if ( tr == NULL ) {
    return NULL;
  }
  if ( tr->leftChild == NULL ) {
    tr->rightChild = removeNodes(tr-> rightChild);
    return tr;
  }
  if ( tr->rightChild == NULL ) {
    tr->leftChild = removeNodes(tr->leftChild);
    return tr;
  }
  if ( (tr->leftChild)-> item == (tr->rightChild)-> item ) {
    freeTree(tr->rightChild);
    tr->rightChild = NULL;
  } else {
    tr->rightChild = removeNodes(tr-> rightChild);
  }
  tr->leftChild = removeNodes(tr->leftChild);
  return tr;
}
```

A shorter definition:

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

2. $\boxed{\text{30 points}}$ This problem is about paths in simple graphs. A simple graph is a graph without loops and without parallel edges. The length of a path is the number of the edges in it.

a. (20 points) Define in pseudocode an efficient algorithm that determines, given a simple graph G with nodes v, w and edge e, the minimal length of a path in G from v to w that passes *exactly once* through edge e. If no such path exists, the algorithm returns ∞.
Do not forget to specify the input and the output of the algorithm. When you use an auxiliary algorithm, you must define it, too.
Hint: define and use an auxiliary algorithm to find the lenght of the shortest path (i.e. a path with a minimal number of edges) between two nodes.

Solution: First the auxiliary algorithm, a variant of breadth-first search:

```
algorithm ShortestPath(G,v,w)
    input simple graph G with nodes v and w
    output length (number of edges) of a shortest path from v to w
    if v = w then
        return 0
    create an empty queue Q
    give v the label 0
    enqueue(v)
    while Q not empty do
        u ← dequeue()
        n ← label of u
        forall edges e incident with u do
            x ← the other node incident with e
            if x = w then
                return n+1;
            if x has no label then
                give x the label n+1
                enqueue(x)
    return ∞
```

Now the required algorithm. The idea is as follows. Let a and b the nodes of edge e. Any shortest path from v to w that passes e exactly once, consists of three parts. Either first a shortest path from v to a that does not use e, followed by e in the a-b direction, followed by a shortest path from b to w that does not use e. Or first a shortest path from v to b that does not use e, followed by e in the b-a direction, followed by a shortest path from a to w that does not use e.

```
algorithm ShortestPathWithEdge(G,v,e,w)
    input simple graph G with nodes v and w and edge e
    output length (number of edges) of a shortest path from v to w, passing exactly once through e
    H ← G minus e
    a,b ← nodes incident with e
    return min(ShortestPath(H,v,a) + 1 + ShortestPath(H,b,w),
               ShortestPath(H,v,b) + 1 + ShortestPath(H,a,w) )
```

b. (10 points) Analyze the time complexity of the algorithm you gave in part (a), in terms of the number m of edges and the number n of nodes of the graph.

Solution: The algorithm ShortestPath enqueues and dequeues every node once, and it inspects every edge at most twice. For every node and every edge, the operations performed can be done in $\mathcal{O}(1)$ time. So the time complexity of ShortestPath is in $\mathcal{O}(n+m)$.

The algorithm ShortestPathWithEdge makes a copy of G, which can be done in $\mathcal{O}(n+m)$ time. Finding the nodes incident with e is done in $\mathcal{O}(1)$ time. Finally the algorithm ShortestPath is performed 4 times, which is done in $\mathcal{O}(n+m)$ time.

Conclusion: the time complexity of the algorithm is in $\mathcal{O}(n+m)$.