Exam Algorithms and Data Structures in C Friday, April 13, 2018, 9 - 12 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. Consider the following definition of the type `Stack`:

```
typedef struct Stack {
  int *array;
  int top;
  int size;
} Stack;
```

The `top` of a `Stack` is the index of the first free position in `array`, so in an empty stack we have `top==0`. Consider the following implementation of the operation `push`:

```
1 void push (int value, Stack *stp) {
2   stp->top++;
3   stp->array[stp->top] = value;
4   if (stp->top == stp->size) {
5     doubleStackSize(stp);
6   }
7 }
```

Which of the following is correct?

A. This is a correct implementation of `push`.

B. When lines 2 and 3 are interchanged, this is a correct implementation of `push`.

C. When lines 4-6 are moved to before line 2, this is a correct implementation of `push`.

D. When lines 2 and 3 are interchanged, and then lines 4-6 are moved to before line 2, this is a correct implementation of `push`.

2. On an empty max-priority queue, the following actions are performed:

```
enqueue(2); enqueue(6); enqueue(4); removeMax();
enqueue(5); enqueue(removeMax() - 2); removeMax();
enqueue(removeMax() - 2); removeMax(); removeMax();
```

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

A. 1.

B. 2.

C. 3.

D. 4.

3. Which of the following statements about binary search trees containing integer values is **correct**?

A. The root has 2 children.

B. All values in the search tree are different.

C. If the tree has height h and contains n nodes, then h is in $\mathcal{O}(\log(n))$.

D. Preorder traversal visits the nodes in increasing order wrt. the values in the nodes

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

   ⟨*code*⟩ ::= ⟨*word*⟩ `[` ⟨*num*⟩ | ⟨*word*⟩ `]` ⟨*word*⟩ .

   ⟨*word*⟩ ::= ⟨*letter*⟩ ⟨*letter*⟩ `[` ⟨*letter*⟩ `]` .

   ⟨*num*⟩ ::= ⟨*digit*⟩ `{` ⟨*digit*⟩ `}` .

   ⟨*letter*⟩ ::= 'A' | 'B' | 'C' .

   ⟨*digit*⟩ ::= '1' | '2' | '3' .

   Which string can **not** be produced by ⟨*code*⟩ ?

   A. ABBA

   B. BC1322CA

   C. CB31AACA

   D. ABCBACCA

5. This question is about converting a binary tree in array representation into a binary tree in pointer representation. In the array representation, non-negative integers are elements in the tree, while -1 represents an absent node. Moreover, index 0 is not used.
   Consider the following implementation of the operation `arrayToTree`. When `array` is an array of length `n` representing a binary tree, then `arrayToTree(array,n,1)` yields the binary tree in pointer representation.

```
 1 Tree arrayToTree(int *array, int n, int p) {
 2   Tree t;
 3   if ( XXX ) {
 4     return NULL;
 5   }
 6   t = malloc(sizeof(struct TreeNode));
 7   assert(t!=NULL);
 8   t->item = array[p];
 9   t->leftChild = arrayToTree(array, n, 2*p);
10   t->rightChild = arrayToTree(array, n, 2*p+1);
11   return t;
12 }
```

   Which condition should replace `XXX` in line 3?

   A. `n < p || array[p] == -1`

   B. `n <= p || array[p] == -1`

   C. `array[p] == -1 || n < p`

   D. `array[p] == -1 || n <= p`

6. The following statements are about heaps H that implement a max-priority queue for integers. Which of the following statements is **not** correct?

   A. H is a complete binary tree.

   B. If node w is a child of node v, then (value in w) $\leq$ (value in v).

   C. H contains at most one node with 1 child.

   D. RemoveMax is implemented by: remove and return the integer in the root, and restore the heap with Upheap.

7. We consider the compact suffix trie ST for the string `ACAATG.....TTCAGX`. The string consists of 1000 characters `A`, `C`, `G`, `T`, `X`. The character `X` only occurs at the end of the string. Which of the following statement about ST is **wrong**?

   A. ST has 1000 leaves.

   B. ST has at most 2000 nodes.

   C. ST contains a node that contains the one-character string `X` .

   D. Every node in ST has at most 4 children.

8. Consider the following prefix expression: `- * 8 + 5 7 + - 4 / 7 8 * 7 4` . What can we say about the expression tree T for this expression?

   A. T has height 4 and contains 7 leaves.

   B. T has height 4 and contains 8 leaves.

   C. T has height 5 and contains 7 leaves.

   D. T has height 5 and contains 8 leaves.

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

   A. It is possible that the number of nodes with odd degree is odd.

   B. It is possible that the number of nodes with odd degree is even.

   C. It is possible that the number of nodes with even degree is odd.

   D. It is possible that the number of nodes with even degree is even.

10. $G = \langle V, E \rangle$ is an undirected graph with $\#E = 1000$, i.e. G contains 1000 edges. Which of the following statements is true?

    A. $\#V \leq 2000$, i.e. G contains at most 2000 nodes.

    B. If all nodes have degree 4, then $\#V = 500$.

    C. If $\#V \leq 1000$ then G is connected.

    D. If G is a tree then $\#V = 1000$.

Part II: Open Questions

1. [30 points] This problem is about binary trees. Their C type reads

```
typedef struct TreeNode *Tree;

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

a. (10 points) Define an efficient C function with prototype `int count(Tree tr)` that counts and returns the number of nodes in `tr` with 0 or 2 children.

Solution:

```
int count ( Tree tr ) {
  if ( tr == NULL ) {
    return 0;
  }
  if ( tr->rightChild == NULL ) {
    if ( tr->leftChild == NULL ) { /* no children */
      return 1;
    } else {                       /* left child, no right child */
      return count(tr->leftChild);
    }
  } else {
    if ( tr->leftChild == NULL ) { /* right child, no left child */
      return count(tr->rightChild);
    } else {                       /* two children */
      return count(tr->leftChild) + count(tr->rightChild) + 1;
    }
  }
}
```

A shorter definition:

```
int count ( Tree tr ) {
  if ( tr == NULL ) {
    return 0;
  }
  return count(tr->leftChild) +
         count(tr->rightChild) +
         ( (tr->leftChild==NULL) == (tr->rightChild==NULL) ) ;
         /* the last term adds 1 in case of 0 or 2 children */
}
```

b. (20 points) Define an efficient C function with prototype `Tree reduce(Tree tr)` that reduces a binary tree by removing all its nodes with exactly one child. So the resulting tree consists of all nodes of the input tree with 0 or 2 children. See to it that no memory leaks occur.

Solution:

```
Tree reduce ( Tree tr ) {
  Tree returnTree = NULL;
  if ( tr == NULL ) {
    return tr;
  }
  if ( tr->rightChild == NULL ) {
    if ( tr->leftChild == NULL ) { /* no children */
      return tr;
    } else {                       /* left child, no right child */
      returnTree = reduce(tr->leftChild);
      free(tr);
      return returnTree;
    }
  } else {
    if ( tr->leftChild == NULL ) { /* right child, no left child */
      returnTree = reduce(tr->rightChild);
      free(tr);
      return returnTree;
    } else {                       /* two children */
      tr->leftChild = reduce(tr->leftChild);
      tr->rightChild = reduce(tr->rightChild);
      return tr;
    }
  }
}
```

2. 30 points This problem is about algorithms for simple weighted graphs and simple cycles. A simple graph is a graph without loops and without parallel edges. A weighted graph is a graph where every edge has a positive integer called its *weight*. A simple cycle is a cycle with at least one edge, in which all edges are different. The length of a (simple) cycle is the sum of the lengths of the edges in it.

a. (20 points) Define in pseudocode an efficient algorithm that determines, given a simple weighted graph G and an edge e in G, the minimal length of a simple cycle in G containing e. When there is no such cycle, the output is ∞.
Do not forget to specify the input and the output of the algorithm. When you use an auxiliary algorithm, you must define it, too.

Solution: An adaptation of Dijkstra's algorithm. We look for the shortest path between the nodes v and w that are incident with edge e. For this shortest path, edge e will not be used. When we have found such a shortest path from v to w that avoids edge e, we know that its extension with e forms a shortest cycle containing e.

```
algorithm MCLedge (G, e)
    input simple weighted graph G with edge e
    output minimal length of a simple cycle in G containing e if it exists,
        otherwise ∞
    v, w ← the nodes incident with e
    s ← nodes(G)
    forall u in S do
        d(u) ← if u=v then 0 else ∞
    while S is not empty do
        u ← node in S with minimal value of d
        if u=w then
            return d(u) + weight(e)
        remove u from S
        forall z with (u,z) ∈ edges(E) - {e} do
            d(z) ← min(d(z), d(u) + weight(u,z))
    return ∞
```

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 initialization of d in the first **forall** statement takes $\mathcal{O}(n)$ computation steps. The body of the **while** loop is executed at most n times. The selection of the node with minimal d-value can be done efficiently with RemoveMin when the nodes are in a priority queue implemented with a heap. Every RemoveMin takes $\mathcal{O}(\log(n))$ computation steps. The relaxation in the last **forall** statement is performed $\mathcal{O}(m)$ times. Every relaxation takes $\mathcal{O}(\log(n))$ times, to adjust the position of the node in question in the priority queue. All in all, we have a time complexity of $\mathcal{O}((n+m)\log(n))$.