Exam Algorithms and Data Structures with Answers Friday, April 12, 2019, 09:00 – 12:00

Instructions — please read carefully!

- Please write your student number and study programme on this page.
  Do *not* write your name anywhere.
- 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. There is always a unique correct answer.
  Put 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 open problems you can earn maximally 30 points. Your exam grade is $p/10$ where p is the number of points you earned.

Your personal information

Student Number:	
Study Programme:	

Answers to multiple choice questions

Please use only the capital letters A, B, C, D.

1:		2:		3:		4:		5:	
6:		7:		8:		9:		10:	

Points and Grades

Do not write anything here.

	Part I	Part II						
	Multiple-Choice	Question 1	Question 2	$\sum$	$\sum/10$	lab	final
Maximum	40	30	30	100	10	10	10
Reached							

Part I: Multiple Choice Questions

1. On an empty stack, the following functions are performed:

```
push(1);
push(3);
push(2);
int x = pop();
int y = pop() + 2;
while (x < y) {
  push(x);
  x++;
}
```

How many elements are on the stack afterwards?

A. 2

B. 3

C. 4

D. 5

2. Consider the following definition of the type `Queue`:

```
typedef struct Queue {
  int *array;
  int back;
  int front;
  int size;
} Queue;
```

If a `Queue` is not empty, then the `front` is the array position of the oldest item. Consider the following implementation of the operation `dequeue`:

```
1 int dequeue(Queue *qp) {
2   int item;
3   if (isEmptyQueue(*qp)) {
4     queueEmptyError();
5   }
6   qp->front = (qp->front + 1) % qp->size;
7   item = qp->array[qp->front];
8   return item;
9 }
```

Which of the following is true?

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

B. When `qp->front` in line 7 is replaced with `qp->front - 1`, this is a correct implementation of `dequeue`.

C. When lines 6 and 7 are swapped, this is a correct implementation of `dequeue`.

D. When `qp->front` in line 7 is replaced with `qp->front - 1`, and lines 6 and 7 are swapped, this is a correct implementation of `dequeue`.

3. Suppose a linked list has 4 elements. How many non-NULL pointers does the list contain?

    A. 2

    B. 3

    C. 4

    D. 5

4. Consider the pointer and array representations for binary trees.
   Which of the following is correct?

    A. The array representation uses less memory.

    B. The pointer representation provides access to the parent node.

    C. The array representation only works for complete binary trees.

    D. The array representation provides access to the parent node.

5. Let ST be the compressed suffix trie for the string `BANANAS`.
   What is the number of nodes in ST?

    A. 7

    B. 11

    C. 13

    D. 17

6. Consider the following grammar.
   Recall that { … } means '0 or more times'.

   ⟨*formula*⟩ ::= ⟨*literal*⟩ | '(' ⟨*literal*⟩ { '&' ⟨*literal*⟩ } ')' .

   ⟨*literal*⟩ ::= ⟨*atom*⟩ | '!' ⟨*atom*⟩ .

   ⟨*atom*⟩ ::= 'T' | 'F' | ⟨*letter*⟩ .

   ⟨*letter*⟩ ::= 'a' | . . . | 'z' .

   Which of the following can be produced by ⟨*formula*⟩?

    A. `!phi`

    B. `!a & b`

    C. `(a & !T & p)`

    D. `(F & !T) & p`

7. Suppose T is an expression tree for a prefix expression using only positive integers and the binary operators $+$, $-$, $*$ and $/$.

   Which of the following is true?

    A. T has an even number of nodes.

    B. Each node in T has two children.

    C. All branches in T have at least length two.

    D. Each node in T has zero or two children.

8. Consider the following implementation of `upheap`.

```c
void upheap (Heap *hp, int n) {
  if ( XXX ) {
    swap(&(hp->array[n]),&(hp->array[n/2]));
    upheap(hp,n/2);
  }
}
```

Which condition should replace XXX in line ... ?

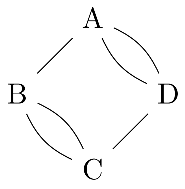
A. `n>1 && hp->array[n] > hp->array[n*2]`

B. `n>1 && hp->array[n] > hp->array[n/2]`

C. `n>0 && hp->array[n] > hp->array[n/2]`

D. `n>0 && hp->array[n] > hp->array[n*2]`

9. Recall that an Euler path is a path in which every edge occurs exactly once. Consider the graph below.



Which of the following is true?

A. This graph has an Euler path.

B. If we add a loop at A, then the graph has an Euler path.

C. If we delete any edge, then the graph has an Euler path.

D. If we delete any edge, then the graph has an Euler cycle.

10. Which of the following search algorithms does *not* find a shortest path?

A. Depth-First Search.

B. Breadth-First Search.

C. Dijkstra's algorithm.

D. the A* algorithm.

Part II: Open Questions

1. 30 points

a. (20 points) The type `List` is defined by

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

Define a C function with prototype `List removeAll(List li, int n)` that, given a list `li` and an integer `n`, returns `li` with all occurrences of `n` removed.
For example, if list `li` represents the sequence [7,3,5,7,7,1,3,5,7], then `removeAll(li,7)` should yield the sequence [3,5,1,3,5]. Make sure that no memory leaks occur.

Solution: Solution with recursion:

```
List removeAll(List l, int n) {
  if ( l == NULL ) {
    return l;
  }
  if ( l->item == n ) {
    List rl = removeAll(l->next,n);
    free(l);
    return rl;
  } else {
    l->next = removeAll(l->next,n);
    return l;
  }
}
```

Solution with iteration:

```
List removeAll(List l, int n) {
  List lf;
  while ( l != NULL && l->item == n ) {
    lf = l;
    l = l->next;
    free(lf);
  }
  if ( l == NULL ) {
    return l;
  }
  /* now l != NULL and l->item != n  */
  List l2 = l;
  while ( l2->next != NULL ) {
    if ( l2->next->item == n ) {
      lf = l2->next;
      l2->next = l2->next->next;
      free(lf);
    } else {
      l2 = l2->next;
    }
  }
  return l;
}
```

b. (10 points) The type `Tree` is defined by

```
typedef struct TreeNode *Tree;

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

Define a C function with prototype `void printInOrderAndFree(Tree t)` that, given a tree `t`, prints the items in the nodes while performing an inorder traversal of the tree, and moreover frees all memory used by `t`.

Make sure that no memory leaks occur.

Solution:

```
void printInOrderAndFree(Tree t) {
  if (t != NULL) {
    printInOrderAndFree(t->leftChild);
    printf("%d ",t->item);
    printInOrderAndFree(t->rightChild);
    free(t);
  }
  return;
}
```

2. $\boxed{\text{30 points}}$
(NOTE: We also announced that G is *not a weighted graph* in this question.)

In this problem, we consider simple graphs $G = \langle V, E\rangle$, so there are no loops and no parallel edges. It is *not* given that G is connected.
a. (20 points) Define an algorithm FindShortestPath(G,v,w) in pseudocode that returns a shortest path from v to w in G, or returns NoPath when there is no path between v and w. A path is represented by a sequence of edges.

Solution:

```
algorithm FindShortestPath(G,v,w)
    input graph G = $\langle V, E\rangle$ with nodes v and w
    return a shortest path from v to w in G if it exists, otherwise NoPath
    Q ← empty queue of nodes
    f ← empty function from $V$ to $E$
    p ← empty sequence of edges
    if v = w then /* trivial case */
        return p
    give v the label VISITED
    enqueue(v)
    while Q not empty do
        u ← dequeue()
        forall unlabeled e incident with u do
            z ← the other node incident with e
            if z = w then /* we found w */
                add e to p
                while u ≠ v do /* we walk back using f */
                    put f(u) on head of p
                    u ← the other node incident with f(u)
                return p
            /* z ≠ w, so w not yet found, and we continue */
            give e the label VISITED
            if z has no label then
                f(z) ← e
                give z the label VISITED
                enqueue(z)
    /* we found no path to w */
    return NoPath
```

b. (10 points) Use FindShortestPath to define an algorithm Has2Paths(G,v,w) in pseudocode that determines whether graph G contains (at least) two fully different paths from v to w (where v $\neq$ w). Here *fully different* means: having no edge in common.

Solution:

```
algorithm Has2Paths(G,v,w)
    input graph G with nodes v and w
    return Yes if G contains two fully different paths from v to w, otherwise No
    p ← FindShortestPath(G,v,w)
    if p = NoPath then /∗ there is not even one path ∗/
        return No
    /∗ now remove the edges in p from G ∗/
    forall edges e in p do
        remove e from G
    p ← FindShortestPath(G,v,w)
    /∗ is there a second path? ∗/
    if p = NoPath then
        return No
    else
        return Yes
```