

Self-Assessment Questions (Computer Science)

1) Write down the disjunctive normal form (DNF) for the Boolean XOR-function. Transform this formula into a conjunctive normal form (KNF) using the distributive laws of a Boolean algebra!

$$(\neg a \wedge b) \vee (a \wedge \neg b) = (\neg a \vee a) \wedge (\neg a \vee \neg b) \wedge (a \vee b) \wedge (b \vee \neg b) = (\neg a \vee \neg b) \wedge (a \vee b)$$

2) Complete the following recursive function definition (in Java). The function *toString* should return a string with the decimal representation of *x*. (e.g., *toString*(-163) = "-0163").

```
String[] digits = {"0","1","2","3","4","5","6","7","8","9"};
static public String toString(int x) {
    if (x == 0) { return "0"; }
    else { if (x < 0) { return "-" + toString(<SOLUTION1>); }
          else {
              String digit = <SOLUTION2>;
              return toString(<SOLUTION3>) + digit;
          }
    }
}
```

<SOLUTION1>: -x

<SOLUTION2>: digits[x%10]

<SOLUTION3>: x/10

3) Modify the definition of the function *toString* such that it uses a loop instead of recursion. The result of the function should remain unchanged.

```
String[] digits = {"0","1","2","3","4","5","6","7","8","9"};
public static String toString(int x){
    String result = < SOLUTION1>;
    String sign = "0";
    < SOLUTION2>
    while (<SOLUTION3>){
        < SOLUTION4>
    }
    return sign + result;
}
```

< SOLUTION1>: ""

< SOLUTION2>: if (x<0) { sign = "-0"; x = -x; }

< SOLUTION3>: x != 0

< SOLUTION4>: result = digits[x%10] + result;
x = x / 10;

4) What is the running time in big O notation for the following methods depending on the length of the list *l*? Derive your answer step-by-step:

(10 min)

```
class IntList {
    public int head;
    public IntList tail;
    IntList(int head, IntList tail) { this.head = head; this.tail = tail; }
}
public static IntList append(IntList l, int x) {
    return (l == null) ? new IntList(x, l) : new IntList(l.head, append(l.tail, x));
}
```

```

public static IntList reverse(IntList l) {
    return (l == null) ? l : append(reverse(l.tail), l.head);
}

```

The method *append* steps through the list once, testing whether the end is reached. If it is, it appends the value *x*. Testing and appending takes $O(1)$ time. The over-all running time is thus $O(n)$ where *n* is the length of the list.

The method *reverse* steps through the list, testing whether the end is reached. Unless it is, it carries out the *append*-method for each list element. Since *append* takes $O(n)$ time and is carried out *n* times, the method *reverse* takes $O(n^2)$ time.

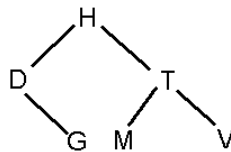
5) Let *n* be the number of leaves in a binary tree *T*. What is the minimum number of inner nodes in *T*?

A binary tree with *n* leaves has at least *n*-1 inner nodes.

6) How many KB memory does it take to hold a bit vector representation of the set of prime numbers less than 1.000.000?

The bit vector representation takes 1.000.000 bits stating for each number between 1 and 1.000.000 whether this number is a prime number or not. This conforms to $1.000.000 / 8 * 1024 \approx 122$ KB of memory.

7) Construct a binary search tree by inserting the sequence of keys H, T, M, D, V, G. The nodes of the search tree are ordered based on alphabetical order.

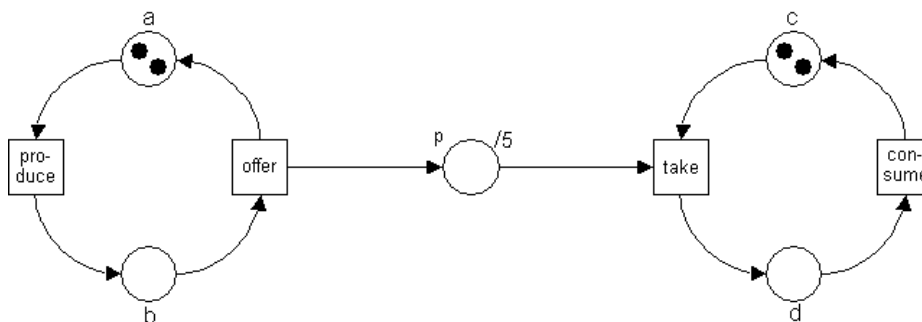


8) Why does one use queues and not stacks as data structures to collect pending service requests for process scheduling?

Using stacks involves the danger that some requests will never be served. The reason is that new requests are stored at the top of the stack and thus served before the existing request which reside further down in the stack.

In queues, however, requests are served in the same order in which they have entered the system. Thus all requests only spend a finite waiting time within the system.

9) Draw a Petri-Net with two producers and two consumers which communicate using a common buffer with capacity 5!



10) Name one example for a simple data type and one example for a structured data type.

Integer, long, short, cardinal, float, double, boolean, character, ...

Record, tuple, array, struct, union, vector