Department of Informatics

Prof. Dr. Michael Böhlen
Binzmühlestrasse 14
8050 Zurich
Phone: +41 44 635 4333
Email: boehlen@ifi.uzh.ch

University of Zurich^UZH

Informatics II — Final Exam

Spring 2018 — 28.05.2018

Name: ____________________ Matriculation number: ____________

Advice

You have 90 minutes to complete the exam of Informatik II. The following rules apply:

- Answer the questions on the exam sheets or the backside. Mark clearly which answer belongs to which question. Additional sheets are provided upon request. If you use additional sheets, put your name and matriculation number on each of them.
- Check the completeness of your exam (19 numbered pages).
- Use a pen in blue or black colour for your solutions. Pencils and pens in other colours are not allowed. Solutions written in pencil will not be corrected.
- Stick to the terminology and notations used in the lectures.
- Only the following materials are allowed for the exam:
  - One A4 sheet (2-sided) with your personal handwritten notes, written by yourself. Sheets that do not conform to this specification will be collected.
  - A foreign language dictionary is allowed. The dictionary will be checked by a supervisor.
  - No additional items are allowed except calculators. Computers, pdas, smart-phones, audio-devices or similar devices may not be used. Any cheating attempt will result in a failed test (meaning 0 points).
- Put your student legitimation card (“Legi”) on the desk.

Signature:

Correction slot — Please do not fill out the part below

Exercise	1	2	3	4	Total
Points Achieved					
Maximum Points	22	15	18	15	70

Exercise 1 — 22 Points

1.1 [2 points] Let T be a binary search tree without duplicates. The lowest common ancestor of two nodes $n_1 \in T$ and $n_2 \in T$ is the node in T with the largest depth that has both n_1 and n_2 as descendants. Each node is a descendant of itself.

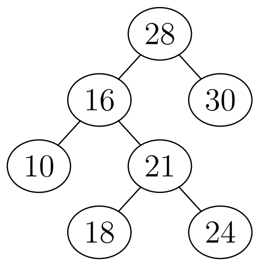


Figure 1: Binary search tree

Given the binary search tree in Figure 1, determine the lowest common ancestor of the following pairs of nodes:

i) 18 and 10: 16

ii) 24 and 16: 16

1.2 [8 points] Assume a binary search tree T with integers as keys. Each node has pointers to its children but no pointer to its parent. Use C to define T. Use C or pseudocode to describe an algorithm that takes two nodes of the tree as arguments and returns the pointer to the lowest common ancestor.

Node definition:

```c
struct node {
  int val;
  struct node* left;
  struct node* right;
};
```

code/bst.c

Recursive solution:

```c
struct node* lca_rec(struct node* root, struct node* n1, struct node* n2) {
  if (root==NULL) return NULL;
  if (root->val > n1->val && root->val > n2->val) {
    return lca_rec(root->left, n1, n2);
  }
  if (root->val < n1->val && root->val < n2->val) {
    return lca_rec(root->right, n1, n2);
  }
  return root;
}
```

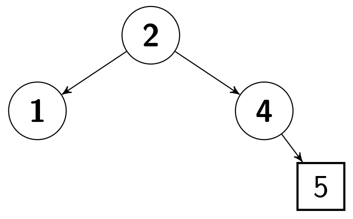
code/bst.c

Iterative solution:

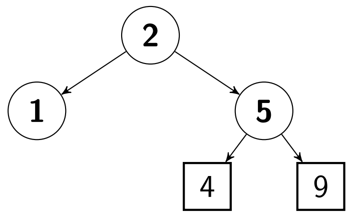
```c
struct node* lca(struct node* root, struct node* n1, struct node* n2) {
  while(root!=NULL) {
    if (root->val > n1->val && root->val > n2->val) root=root->left;
    if (root->val < n1->val && root->val < n2->val) root=root->right;
  }
  return root;
}
```

code/bst.c

1.3 [12 points] Consider the red-black tree in Figure 2a where black nodes are denoted with a circle and red nodes are denoted with a square. Table 1 illustrates the cases and actions if value **9** is inserted in the red-black tree of Figure 2a. Figure 2b shows the resulting tree.



(a) Before insertion of 9



(b) After insertion of **9**

Figure 2: Inserting 9 into a red-black tree

Operation	Case	Action	Arguments
Inserting 9	Case 3 mirrored	assign color	4, red
		assign color	5, black
		left rotate	4
Inserting 13	Case 3 mirrored	assign color	10, black
		assign color	9, red
		left rotate	9
Inserting 12	Case 1 mirrored	assign color	10, red
		assign color	9, black
		assign color	13, black
	Case 3 mirrored	assign color	2, red
		assign color	5, black
		left rotate	2

Table 1: Insert cases and actions

Perform the following sequence of insertion operations on the red-black tree in Figure 3: insert `13`, insert `12`. Each operation is applied to the result of the previous operation. For each operation fill in the missing parts in Table 1 and draw the resulting red-black tree.

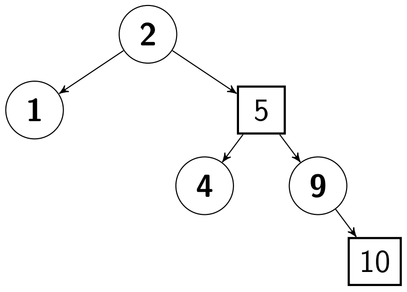
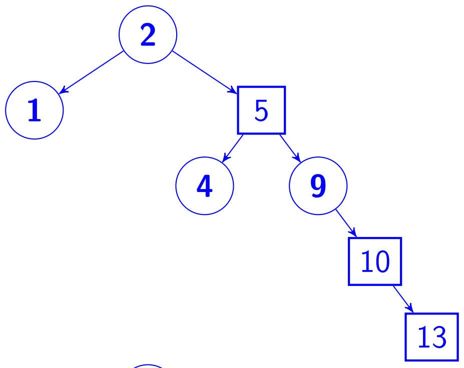
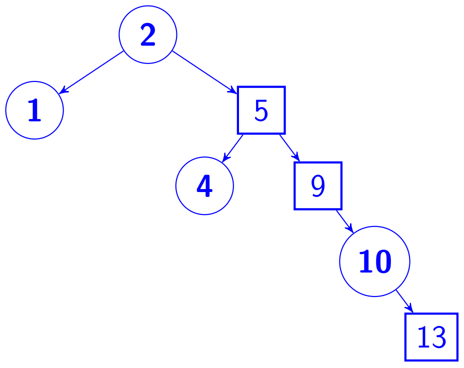
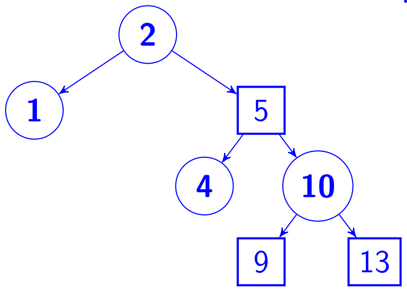


Figure 3: Red-black tree

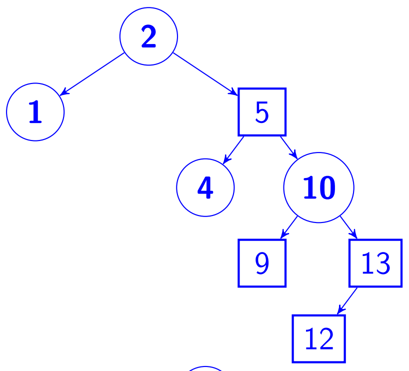
Inserting 13:

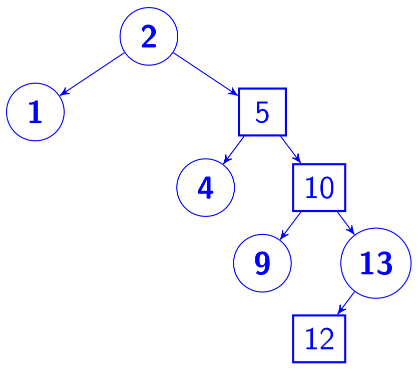


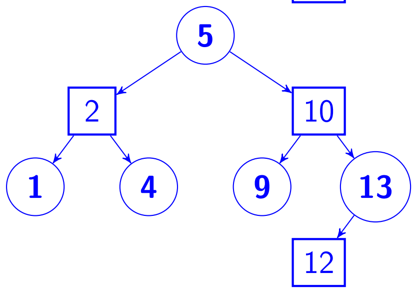




Inserting 12:



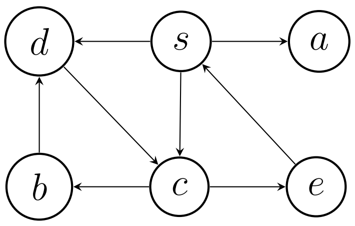




Name: Matriculation number:

Exercise 2 15 Points

2.1 [3 points] Consider the graph in Figure 4 together with its adjacency list representation.



adj(s) = [a,c,d]
adj(a) = [ ]
adj(c) = [e,b]
adj(b) = [d]
adj(d) = [c]
adj(e) = [s]

Figure 4: Graph and its adjacency list

State the order in which the nodes are visited during, respectively, a Breadth First Search (BFS) and Depth First Search (DFS). The search starts at node `s`.

i) **Breadth First Search** (BFS):

Solution: s a c d e b

ii) **Depth First Search** (DFS)

Solution: s a c e b d

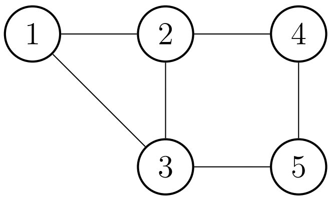
2.2 [3 points] List two data structures that can be used to represent graphs. For each data structure determine the worst time complexity (big O notation) for deciding if node x and node y are connected.

Adjacency List: O(V)
Adjacency Matrix: O(1)

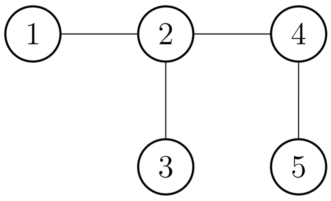
2.3 [9 points] A graph is said to be `Biconnected` if:

(a) It is connected, i.e., it is possible to reach every vertex from every other vertex, by a simple path.

(b) Even after removing any vertex the graph remains connected.



(a) Biconnected graph



(b) Not a biconnected graph

Given an undirected `connected graph G` and a start vertex `s`, write a pseudocode algorithm that determines if the connected graph G is biconnected? In an undirected graph two connected vertices, v_1 and v_2, are represented by two edges as $e_1(v_1, v_2)$ and $e_2(v_2, v_1)$ from v_1 to v_2 and v_2 to v_1, respectively.

```
Algorithm: Init(G)
foreach $v \in G.V$ do
    v.color = W; v.deleted = false;
return isBiConnected(G);
```

```
Algorithm: isBiConnected(G)
foreach $v \in G.V$ do
    v.deleted = true;
    if isConnected(G) == false then
        return false;
return true;
```

```
Algorithm: isConnected(G)
InitQueue(Q);
s = randVertex(G);
Enqueue(Q,s) ;
while $Q \neq \phi$ do
    v = Dequeue(Q) ;
    v.color = G;
    foreach $u \in v.adj$ do
        if $u.color == W$ and $u.deleted == false$ then
            Enqueue(Q,u)
foreach $v \in G.V$ do
    if $v.color == W$ and $v.deleted == false$ then
        return false;
foreach $v \in G.V$ do
    v.color = W;
    v.deleted =false;
return true;
```

```
Algorithm: randVertex(G)
foreach $v \in G.V$ do
    if $v.deleted == false$ then
        return v;
```

Name: Matriculation number:

Exercise 3 18 Points

3.1 [6 points] Consider a function $roll(S, n, k)$ that can be applied to a stack S. The roll function rotates the top $n \geq 0$ elements of stack S by $k \geq 0$ positions in a circular fashion. Figure 6 illustrates three examples of the *roll* function.

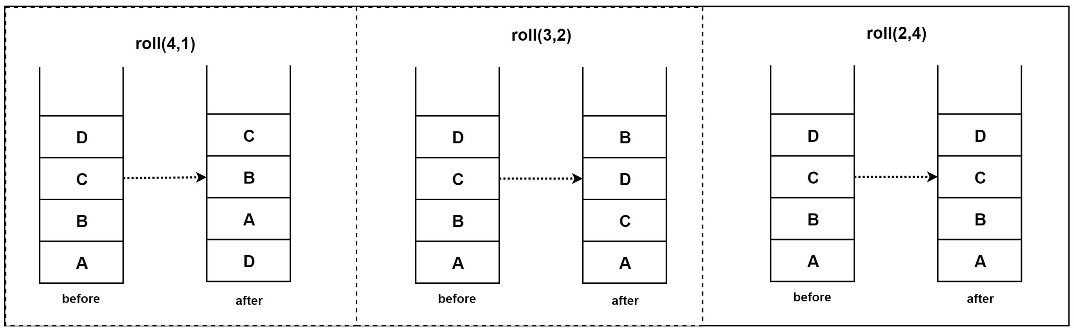


Figure 6: Examples of Roll operator

Design and sketch an algorithm that implements $roll(S, n, k)$. For non-valid inputs the stack must be left unchanged. Note that k can be larger than m, in which case the *roll* operation does more than one complete rotation. The performance of your algorithm must be $O(n)$.

```
Algorithm: roll(Stack s,int n,int k)
if $n < 0$ or $k < 0$ or $n > s.size()$ then return;
Initialize array $arr$ of length $n$;
for $i=0$ to $n-1$ do arr[i] = s.pop(); ;
for $i=n-1$ to $0$ do s.push(arr[(i+k) mod n]); ;
```

3.2 [4.5 points] Consider hash table `HT` with 7 slots and hash function $h(k) = k \mod 7$.

Draw the hash table after inserting, in the given order, values $19, 26, 13, 48, 17$.

Assume collisions are handled by

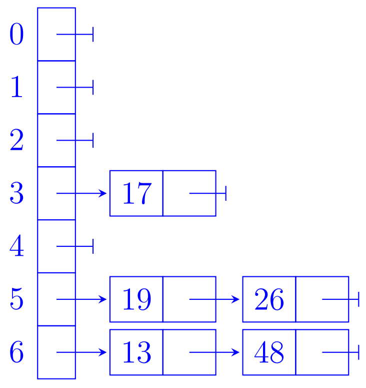
(a) chaining

(b) linear probing

(c) double hashing with secondary hash function $h2(k) = 5 - (k \mod 5)$ for the step size

For each collision resolution scheme show the hash tables after all insertions have been performed.

chaining



linear probing

0	13
1	48
2	
3	17
4	
5	19
6	26

double hashing

0	
1	48
2	26
3	17
4	
5	19
6	13

3.3 [7.5 points] Answer the following:

(a) [1.5 points] Consider an initially empty hash table of size M and hash function `h(x) = x mod M`. In the worst case what is the time-complexity to insert `n` keys into the table if chaining is used to resolve collisions. Assume that overflow chains are implemented as unordered linked lists. Give a brief justification for your answer.

O(n)

(b) [1.5 points] What is the answer for question (a) if the overflow lists are ordered? Give a brief justification for your answer.

$\mathbf{O}(n^2)$

(c) [1.5 points] Consider the same hash table and function as in task (a), but assume that collisions are resolved using linear probing, and $n \leq \frac{M}{2}$. In the worst case what is the time complexity (in big O notation) to insert n keys into the hash table? Give a brief justification for your answer.

$\mathbf{O}(n^2)$

(d) [1.5 points] How big must the hash table be if we have 60000 items in a hash table that uses open addressing (linear probing) and we want a load factor of 0.75?

$$TableSize = \frac{n}{\alpha} = \frac{6000}{0.75} = 80000$$

(e) [1.5 points] What is the expected number of comparisons to search for a key if we must store 60000 items in a hash table that uses open addressing (linear probing) and we have a load factor of 0.75.

$$\frac{(1+\frac{1}{(1-\alpha)})}{2} = \frac{(1+\frac{1}{1-0.75})}{2} = 2.5$$

Exercise 4 — 15 Points

We are given a rod of metal of length n. We are also given a pricing table of m different cut lengths $l_1, ..., l_m$, and their corresponding prices, $p_1, ..., p_m$. We assume the table is sorted by cut length so that l_1 is the smallest cut length we can sell. We want to cut the rod into different segments to maximize the sum of all segment prices hence maximizing the profit. A piece of length i is worth p_i CHF as listed in Figure 7.

length l_i	3	5	7
price p_i	6	7	10

Figure 7: Prices for rods of different lengths

Thus, if we have a rod of length 10 the most beneficial strategy is to cut the rod into three pieces of length 3, which gives you a benefit of 18 CHF.

4.1 [2 points] Assume $r(n)$ denotes the maximum profit you can get for a rod of length n and price table p. Formulate a recursive definition of $r(n)$.

$$r(n) = \begin{cases} 0 & \text{if } n < l_1 \\ \max_{i=1..m}(p_i + r(n - l_i)) & \text{otherwise} \\ \forall k \text{ such that } l_k \leq n & \end{cases}$$

4.2 [4 points] Assume an array $l[1...m]$ of different cut lengths and an array $p[1...m]$ of corresponding prices. Write a recursive algorithm that computes the maximal profit for cutting a rod of length n into pieces. Formulate a recurrence for the runtime complexity of your algorithm. Determine the asymptotic runtime complexity by solving the recurrence.

```
Algorithm: CutRod(p,n)
if $n == 0$ then
    return 0;
r = - $\infty$ ;
for $i=1$ to $n$ do
    r = max(r,p[i] + CutRod(p,n-i));
return r;
```

4.3 [3 points] In order to efficiently compute the maximal profit by cutting a rod of length n into pieces, a dynamic programming solution with arrays $r[0...n]$ and $c[0...n]$ can be used. Element $r[i]$ in array r contains the maximum profit earned for cutting the rod of length i. Element $c[i]$ in array c contains a cut-length that is part of a solution that yields the maximal profit for a rod of length i. Given the prices for different cut lengths in Table 2, complete Table 3 with the maximal profits and optimal cut lengths.

length l_i	3	5	7
price p_i	6	7	10

Table 2: Prices for rods of different lengths

Fill the table for rod of length 10

length l_i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
c[i]	0	1	5	8	10	13	17	18	22	25	30

Table 3: Maximal profits and optimal cut lengths

4.4 [6 points] Assume an array $l[1...m]$ of different cut lengths and an array $p[1...m]$ of corresponding prices. Describe an algorithm that uses the memoization technique to compute the maximum profit for cutting a rod of length n into different pieces. Also compute an array $c[0...\text{n}]$ with the values of the cut lengths that give the optimal value. You can use either C or pseudocode for your solution.

One way we can do this is by writing the recursion as normal, but store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in r[n].

```
Algorithm: InitArray(p,n)
let r[0..n] be a new array
for i=0 to n do
    r[i] = - ∞;
return CutRodDP(p,n,r);
```

```
Algorithm: CutRodDP(p,n)
if r[n] ≥ 0 then
    return r[n];
if n == 0 then
    q = 0;
else
    q = - ∞;
    for i=1 to n do
        q = max(q,p[i] + CutRodDP(p,n-i,r));
    r[n] = q;
return q;
```