1. Linear Data Structures

1. $\boxed{\text{10 points}}$ **TEXT** This exercise is about stacks. Write down nine operations which can be executed on an empty stack in this order, one operation per line. After each operation, but in the same line, write down the contents of the stack after the operation, with the bottom element first.

   In addition, ensure the following:

   - At some point during your sequence of operations the stack should contain at least three items.
   - After the ninth and last operation the stack should contain at most two items.
   - Use random and different numbers from 0 to 100 as items.

   Solution:

1.	push(15)	15
2.	push(17)	15,17
3.	pop()	15
4.	push(23)	15,23
5.	push(42)	15,23,42
6.	pop()	15,23
7.	pop()	15
8.	pop()	
9.	push(11)	11

2. $\boxed{\text{10 points}}$ **TEXT** Consider the following definition of the type Queue:

```c
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`:

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

   What is wrong with this implementation? Explain your answer and make suggestions how the function should be changed.

   Solution:

   Line 6 is problematic: if `(qp->front + 1) % qp->size` in line 5 evaluated to 0, then `qp->front - 1` is -1 and not a valid index for the array. The return value of the function should be `qp->array[qp->front]` with the old value of `qp->front`. One way to repair the function is to replace lines 5 and 6 with these three lines:

```c
  int item = qp->array[qp->front];
  qp->front = (qp->front + 1) % qp->size;
  return item;
```

   The quick fix to replace `[qp->front - 1]` in line 6 by `[(qp->front - 1) % qp->size]` will *not* work. For example, `(-1) % 10` is still `verb-1` in C. What *does* work, is to change line 6 to `return [(qp->front - 1 + qp->size) % qp->size];`.

3. 10 points **TEXT** Consider the following grammar.

```
<id>       ::= <letter> { <posdigit> } [ '*' ]
<label>    ::= <letter> { <num> }
<num>      ::= <posdigit> { <digit> }
<digit>    ::= '0' | <posdigit>
<posdigit> ::= '1' | '2' | '3' | '4'
<letter>   ::= 'a' | 'b' | 'c'
```

(a) Write down two expressions which can be produced by `<id>` but not by `<label>`.
(b) Write down two expressions which can be produced by `<label>` but not by `<id>`.
(c) Write down one expression which can be produced both by `<id>` and by `<label>`.

Solution:

(a) `a*`
`b2*`

(b) `a100`
`c404`

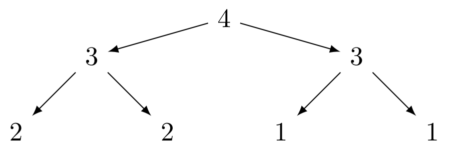
(c) `b` (or also `b2`)

2. Trees

1. 5 points **PDF** Draw a heap which contains all digits of your student number as elements. For example, if your student number would be 3214123 then the heap should contain seven nodes with the elements 3, 2, 1, 4, 1, 2 and 3.
Next to the drawing, also write down the array representation of the heap.

Solution:

For the given example student number s3214123 this is a heap:



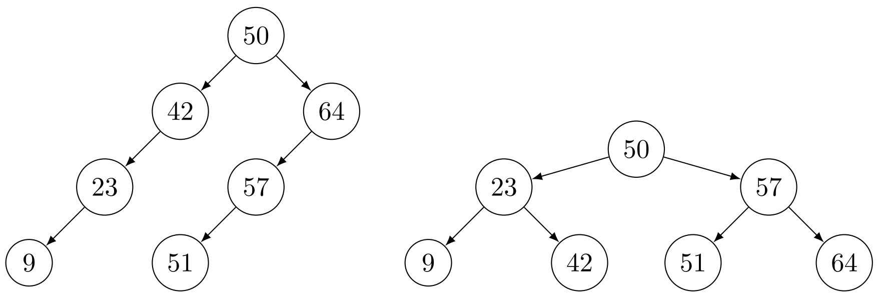
The array representation for this heap is:

index	0	1	2	3	4	5	6	7
item		4	3	3	2	2	1	1

2. 10 points **PDF** Draw two search trees which contain the elements 9, 42, 50, 51, 57, 64 and your age in years. One of the trees should have at most two leaves and the other one should have at least three leaves.

Solution:

Suppose my age is 23. Then I can draw the following two search trees. It is also possible to draw search trees with only one leaf or with three leaves for these elements.



3. 10 points **CODE** This problem is about binary trees. Their definition in C is as follows.

```
typedef struct TreeNode *Tree;

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

Define an efficient C function with prototype `Tree singleBranch(Tree tr)` that at each node with two children removes the right child and its descendants. That is, the resulting tree should consist of a single branch.

Make sure that no memory leaks occur.

You may call `freeTree(Tree t)` from the solution of Exercise 2.3 in the lecture notes.

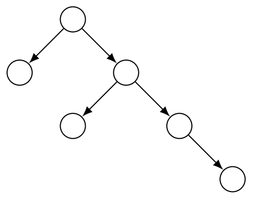
Solution:

```
Tree singleBranch(Tree tr) {
  if (tr != NULL) {
    if (tr->leftChild != NULL) {
      if (tr->rightChild != NULL) {
        /* two children */
        freeTree(tr->rightChild);
        tr->rightChild = NULL;
      }
      /* recursion for left child */
      tr->leftChild = singleBranch(tr->leftChild);
    } else {
      /* only right child or no children */
      tr->rightChild = singleBranch(tr->rightChild);
    }
  }
  return tr;
}
```

4. $\boxed{\text{10 points}}$ **CODE** This question is about binary trees as described in the previous question.

Define an efficient C function with prototype `int childProduct(Tree tr)` which returns the product of the number of left children and the number of right children. You may define additional helper functions to be called from `childProduct`.

For example, given the tree below the function `childProduct` should return 6.



Solution:

```
int lrChildren(Tree tr, int side) {
  if (tr == NULL) {
    return 0;
  }
  int k = 0;
  if (tr->leftChild != NULL) {
    k += lrChildren(tr->leftChild,side);
    k += (side == 0);
  }
  if (tr->rightChild != NULL) {
    k += lrChildren(tr->rightChild,side);
    k += (side == 1);
  }
  return k;
}

int childProduct(Tree tr) {
  return lrChildren(tr,0) * lrChildren(tr,1);
}
```

5. $\boxed{\text{10 points}}$ **PDF** This question is about tries and has three parts; each part relies on its successor.

You are given the following dictionary: `{am, are, be, been, could, had, has, have}`.

First, add your first name and your last name to the dictionary, using lower case letters. (If you have multiple first or last names, choose one of each. Also, leave out any prefixes such as “van” or “de”). Use the resulting dictionary for this whole exercise.

Now, make the following three drawings. For each step, briefly explain any additional assumptions or decisions you made. If you decide to use a stop character to mark the end of words, please use the `$` symbol.

(a) Draw a standard trie T for the dictionary including the words above and your first and last name.

(b) Draw a compressed trie T’ based on the standard trie T, from part a.

(c) Draw a compact trie T” based on the compressed trie T’, from part b.

Please upload your answers for (a), (b) and (c) together as one single PDF file.

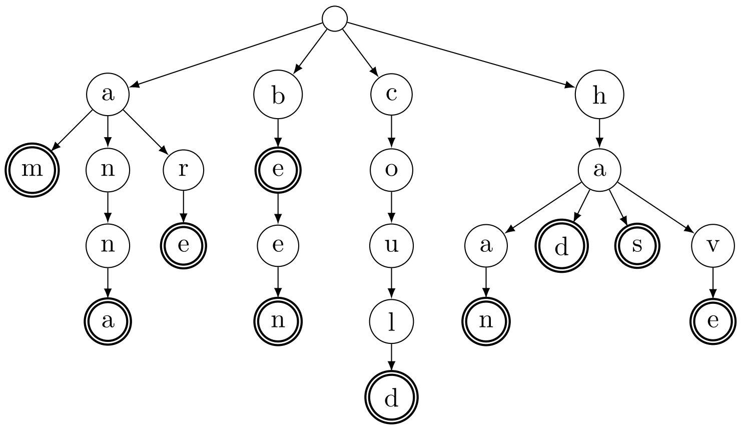
Solution:

Suppose my name is “Anna de Haan”. Then I add `anna` and `haan` to the dictionary:

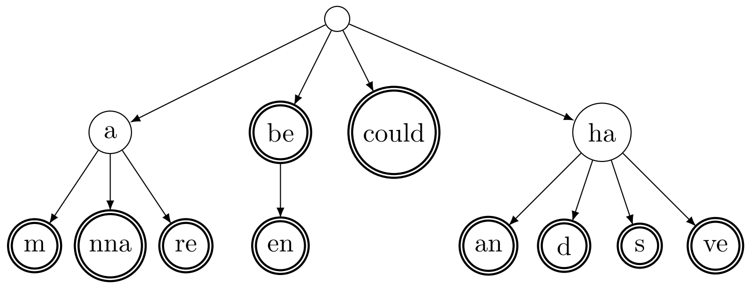
```
am, anna, are, be, been, could, haan, had, has, have
```

There are prefixes of other entries in this dictionary (“be” is a prefix of “been”), so we need to mark the end of words. I decide not to use a stop character, but instead mark each node where a word ends with a thick double border.

(a) The standard trie:



(b) The compressed trie:

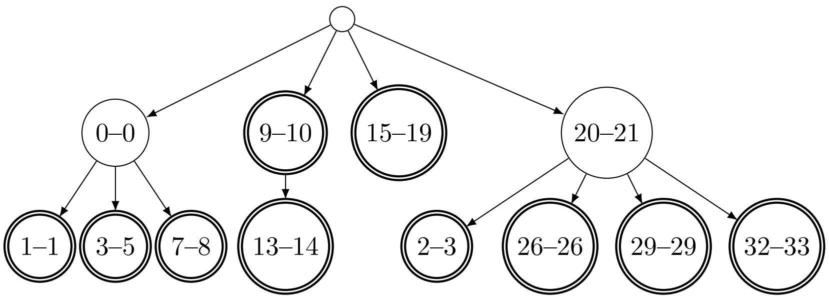


(c) For the compact trie, first we need indices for the containing of all words:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
a m a n n a a r e b  e  b  e  e  n  c  o  u  l  d

20 21 22 23 24 25 26 27 28 29 30 31 32 33
 h  a  a  n  h  a  d  h  a  s  h  a  v  e
```

Then we can draw the compact trie:

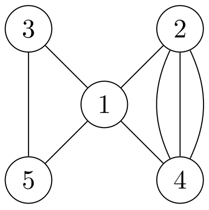


3. Graphs

1. $\boxed{\text{5 points}}$ **PDF** Draw a connected undirected graph with 5 nodes and 8 edges which has an Euler cycle. Number the nodes and write down an Euler cycle of this graph.

Solution:

The graph below has the Euler cycle $(2, 1, 5, 3, 1, 4, 2, 4, 2)$.



2. $\boxed{\text{5 points}}$ **TEXT** Suppose we apply BFS and DFS to the same simple connected graph. What do we know about the height of the two resulting spanning trees? Explain and justify your answer!

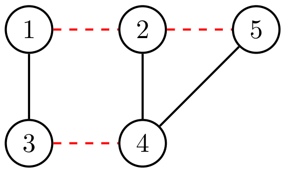
Solution:

In the BFS spanning tree each branch from the root to a leaf corresponds to a shortest path. Hence the height of the BFS spanning tree is lower than or equal to the height of the DFS spanning tree.

3. $\boxed{\text{15 points}}$ **PDF** This exercise is about simple connected graphs (undirected and without weights). Suppose for each graph $G = (V, E)$ we also have a function $f \colon E \to \{\text{red}, \text{black}\}$ which assigns to each edge in G one of two labels.

(a) Define an algorithm $\mathsf{AlternatingDistance}(G, f, v, w)$ in pseudocode that returns the length of a shortest path from v to w in G such that the edges along the path have alternating labels. If there is no path with alternating labels from v to w, the algorithm should return NoPath.

For example, given the graph below and $v = 1$ and $w = 5$ the algorithm should return 3 and not 2 because (1,3,4,5) is a shortest path with alternating labels, whereas (1,2,5) does not have alternating labels.



(b) What is the time complexity of your algorithm in terms of the number of nodes n and the number of edges e? Explain your answer.

Solution:

(a) We use Breadth-First-Search, but we enqueue nodes together with a label to ensure that only edges with alternating color are used. Moreover, we use two distance functions, one for paths starting with red, one for paths starting with black.

```
**algorithm** AlternatingDistance(G,f,v,w)
    **input** graph G = $\langle V, E \rangle$, edge-labelling $f$, nodes v and w
    **return** length of a shortest path with alternating labels from v to w in G
        if it exists, otherwise $\infty$
    dist $\leftarrow$ empty function from $\{\text{red}, \text{black}\} \times V$ to $\mathbb{N} \cup \{\infty\}$
    dist(red,v) $\leftarrow$ 0
    dist(black,v) $\leftarrow$ 0
    Q $\leftarrow$ empty queue of triples (node,label,label)
    enqueue((v,red,red))
    enqueue((v,black,black))
    **while** Q not empty **do**
        (u,startColour,nextColour) $\leftarrow$ dequeue()
        **forall** e incident with u such that f(e)=nextColour **do**
            z $\leftarrow$ the other node incident with e
            **if** z = w **then** /* we found w and can stop */
                return dist(startColour,u) + 1
            /* z $\neq$ w, so w not yet found, and we continue */
            **if** dist(startColour,z) = $\infty$ **then**
                dist(startColour,z) = dist(startColour,u) + 1
                otherColour $\leftarrow$ **if** nextColour = black **then** red **else** black
                enqueue((z,startColour,otherColour))
    **return** NoPath /* we found no path to w */
```

(b) Let n be the number of nodes of the graph and e the number of edges. The given algorithm will enqueue and dequeue each node at most twice, and the **forall** loop will run for each edge at most four times, hence the complexity of the whole algorithm is in $\mathcal{O}(2n + 4e) = \mathcal{O}(n + e)$.