

# Mock Exam 2

ETH Zurich

December 3, 2014

Name: \_\_\_\_\_

Group: \_\_\_\_\_

Question 1	/ 13
Question 2	/ 10
Question 3	/ 13
Total	/ 36

## 1 Contracts (13 points)

We are interested in an adventure game in which knights wander through unknown lands. A knight owns coins that can be used for recruiting villagers as companions and for healing wounds. He also has a reputation ranging from -5 to 5, which is subtracted from the cost of recruiting and healing (see examples below). Even though the cost is reduced for the knights with good reputation, it can never go below zero. All the actions listed below allow the knight to gain experience points.

Here are some actions a knight can do:

- *recruit* a villager. This is possible if:
  - The knight does not have a companion already
  - The villager does not have a knight as a leader already
  - The knight can afford the recruiting cost for the villager. For example, if the villager cost is 3 and the knight has a bad reputation of -2, he would need 5 coins to recruit this villager. However, if the knight has a positive reputation of 5, he can recruit the villager for free.

The knight gains as many experience points as coins spent.

- *dismiss* a villager as a companion for free (gaining 5 experience points). This is only possible if the knight has a companion already.
- *heal* one or more wounds. The cost of healing is the same as the number of wounds to be healed; the knight must be able to afford the cost. For example, if the knight has 3 coins and a good reputation of 1, he can afford to heal up to 4 wounds. The knight gains as many experience points as wounds healed.

Your task is to add contracts to the deferred classes *KNIGHT* and *VILLAGER*, so that the informal specification above (together with the feature comments) is reflected in each class interface.

Please note:

- Assume that the void safety option of the compiler is turned off. This means that, when appropriate, you have to explicitly check whether the objects are void or not.
- The number of dotted lines is not indicative of the number of missing contract clauses.
- You need to write **True** at places where you think no explicit contract is necessary: leaving a postcondition empty gives you 0 point for that section.
- The following features from class *INTEGER* may be useful:

```
class INTEGER
feature
  max (other: INTEGER): INTEGER
    -- The greater of current object and 'other'.

    -- Other features omitted.
end
```

**deferred class**

*KNIGHT*

**feature** -- Access

*wounds*: *INTEGER*

-- Number of wounds the knight currently has.

*coins*: *INTEGER*

-- Number of coins owned. They are used to pay for recruiting and healing.

*reputation*: *INTEGER*

-- Affects positively or negatively the cost of recruiting and healing.

*experience*: *INTEGER*

-- Experience points gained by performing actions.

*companion*: *VILLAGER*

-- Companion of current knight, possibly Void.

**feature** -- Basic operations

*recruit* (*a\_villager*: *VILLAGER*)

-- Recruit a villager.

**require**

.....  
.....  
.....  
.....  
.....

**deferred**

**ensure**

.....  
.....  
.....  
.....

**end**

*dismiss*

-- Dismiss the current companion.

**require**

.....  
.....  
.....  
.....

**deferred**  
**ensure**

.....  
.....  
.....  
.....

**end**

*heal* (*w*: *INTEGER*)  
-- Heal w wounds.

**require**

.....  
.....  
.....  
.....  
.....

**deferred**  
**ensure**

.....  
.....  
.....  
.....  
.....

**end**

**invariant**

.....  
.....

```
.....  
.....  
.....  
  
end  
  
deferred class  
  VILLAGER  
  
feature -- Access  
  
  recruiting_cost : INTEGER  
    -- Positive cost of recruiting the current villager .  
  
feature -- Status report  
  
  has_leader: BOOLEAN  
    -- Does the current villager have a leader?  
  
feature -- Status setting  
  
  set_has_leader (hl: BOOLEAN)  
    -- Set the "has leader" status for the current villager .  
  require  
  
    .....  
  
    .....  
  
  deferred  
  ensure  
  
    .....  
  
    .....  
  
  end  
  
invariant  
  
  .....  
  
  .....
```

## 2 Data Structures (10 points)

A *bag* (also called *multiset*) is a generalization of a set, where elements are allowed to appear more than once. For example, the bag  $\{a, a, b\}$  consists of two copies of  $a$  and one copy of  $b$ . However, a bag is still unordered, so the bags  $\{a, b, a\}$  and  $\{a, a, b\}$  are equivalent.

Below you will find source code of a linked representation of the bag data structure; this representation is very similar to a regular singly-linked list, except for the following:

- In addition to the value and the reference to the next cell, each bag cell stores the number of copies of its value (see Figure 1).
- For a given value, at most one cell storing that value should appear in the data structure.

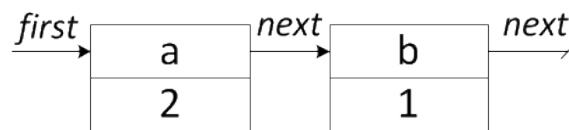


Figure 1: A possible linked representation of the bag  $\{a, a, b\}$ .

In the class `LINKED_BAG` below fill in the implementations of the following two features:

1. `remove (v: G; n: INTEGER)`, which removes as many copies of  $v$  as possible, up to  $n$ . For example, removing one copy of  $a$  from the bag  $\{a, a, b\}$  will result in a bag  $\{a, b\}$ , while removing two copies of  $c$  from the same bag will not change it.
2. `subtract (other: LINKED_BAG [G])`, which removes all elements of *other* from the current bag. For example, taking the bag  $\{a, a, b\}$  and subtracting  $\{a, b, c\}$  from it will yield the bag  $\{a\}$ .

Your implementation should satisfy the provided contracts.

```

class
    LINKED_BAG [G]

feature -- Access

    occurrences (v: G): INTEGER
        -- Number of occurrences of 'v'.
    do
        ...
    end

feature -- Element change

    add (v: G; n: INTEGER)
        -- Add 'n' copies of 'v'.
    require
        n_positive: n > 0
    do
        ...
    ensure
        n_more: occurrences (v) = old occurrences (v) + n
    end
    
```

```
remove (v: G; n: INTEGER)  
  -- Remove as many copies of 'v' as possible, up to 'n'.
```

```
  require
```

```
    n_positive: n > 0
```

```
  local
```

```
    .....
```

```
    .....
```

```
    .....
```

```
  do
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

```
    .....
```

.....  
.....  
.....  
.....  
.....

```
ensure  
  at_most_n_less: occurrences (v) = (old occurrences (v) - n).max (0)  
end
```

```
subtract (other: LINKED_BAG [G])  
  -- Remove all elements of 'other'.
```

```
require  
  other_exists: other /= Void
```

```
local
```

.....  
.....  
.....

```
do
```

.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

```
end
```



```
feature {LINKED_BAG} -- Implementation
```

```
  first: BAG_CELL [G]  
    -- First cell.
```

```
end
```

```
class
```

```
  BAG_CELL [G]
```

```
create
```

```
  make
```

```
feature -- Initialization
```

```
  make (v: G)  
    -- Create a cell with a single copy of 'v'.  
  do  
    value := v  
    count := 1  
  end
```

```
feature -- Access
```

```
  value: G  
    -- Value.  
  
  count: INTEGER  
    -- Number of copies.  
  
  next: BAG_CELL [G]  
    -- Next cell.
```

```
feature -- Setting
```

```
  set_count (new_count: INTEGER)  
    -- Set 'count' to 'new_count'.  
  do  
    count := new_count  
  end  
  
  set_next (c: BAG_CELL [G])  
    -- Set 'next' to 'c'.  
  do  
    next := c  
  end
```

```
invariant
```

```
  positive_count: count > 0
```

```
end
```

### 3 Recursion (13 points)

#### Task 1

The function *n\_th\_element* (see below) should implement a recursive algorithm that, given a list *a*, computes the *n*-th element of a sorted list (in ascending order) that contains the same elements as the list *a*. Note that list *a* does not need to be sorted. See the example in task 2 to get an idea of what the correct output of function *n\_th\_element* should look like. *Complete the implementation by filling in the missing expressions. Note that the expected implementation uses recursion.*

```

n_th_element (a: ARRAYED_LIST [INTEGER]; n: INTEGER): INTEGER
  require
    a /= Void and 1 <= n and n <= a.count
  local
    i, element, pivot: INTEGER
    not_greater, greater: ARRAYED_LIST [INTEGER]
  do
    print ("n = " + n.out + "%N")
    pivot := a [n]

    -- Split up the elements of list 'a', such that 'not_greater' contains all elements
    -- that are less than or equal to 'pivot' (except for the pivot itself),
    -- and 'greater' contains all elements that are greater than 'pivot'.
    create not_greater.make (0)
    create greater.make (0)
    from
      i := 1
    until
      a.count < i
    loop
      element := a [i]
      if element <= pivot and i /= n then
        not_greater.extend (element)
      elseif pivot < element then
        greater.extend (element)
      end
      i := i + 1
    end

    if (.....) then

      Result := (.....)

    elseif (.....) then

      Result := (.....)
      .....

    elseif (.....) then

      Result := (.....)
  
```

```
..... )  
end  
end
```

### Task 2

In the following code snippets, function *n\_th\_element* is called with different inputs. Write down the output that is printed to the console for each snippet once function *n\_th\_element* has been properly implemented. Note that the function *n\_th\_element* prints out the argument *n* in each call.

Assume that variable *a* was declared as follows:

```
local  
a: ARRAYED_LIST [INTEGER]
```

### Example

```
create a.make (0) -- Create an empty list.  
a.extend (1)  
a.extend (2)  
a.extend (-2)  
print ("result = " + n_th_element (a, 1).out)
```

Output:

```
n = 1  
n = 1  
result = -2
```

### Snippet 1

```
create a.make (0)  
a.extend (0)  
print ("result = " + n_th_element (a, 1).out)
```

Output:

.....

.....

.....

.....

.....

.....

### Snippet 2

```
create a.make (0)
a.extend (5)
a.extend (1)
a.extend (3)
a.extend (2)
a.extend (1)
print ("result = " + n_th_element (a, 5).out)
```

Output:

.....  
.....  
.....  
.....  
.....  
.....