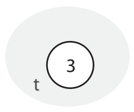
1 Abstract Data Types (11 Points)

In this task you will write an abstract data type for a simple tree structure that stores integers in its nodes and whose nodes always have either no children (leaves) or two children (inner nodes). The ADT for TREE should contain the following six functions:

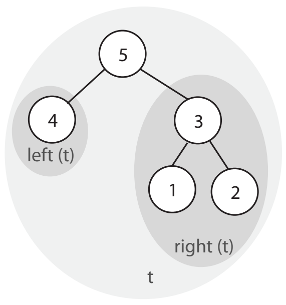
- *make*: Creation function that given an INTEGER argument *i* returns a TREE with *i* stored in the root node.
- *merge*: Given two arguments *t1* and *t2* of type TREE and a third argument *i* of type INTEGER, this function connects the two trees by adding a new root node containing *i* and storing *t1* as left subtree and *t2* as right subtree.
- *root*: Returns the INTEGER stored in the root node of a TREE.
- *left*: Returns the left subtree of a TREE.
- *right*: Returns the right subtree of a TREE.
- *has_children*: Returns True if the TREE has left and right subtrees, False otherwise.

Example 1



$t = make\ (3)$
$root\ (t) = 3$
$has\_children\ (t) = False$
$left\ (t)$ −−> not allowed
$right\ (t)$ −−> not allowed

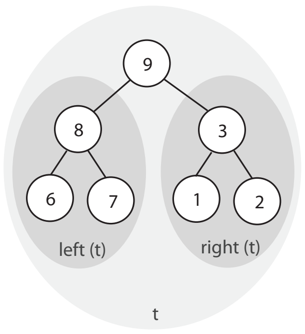
Example 2



$t = merge\ (make\ (4),\ merge\ (make\ (1),\ make\ (2),\ 3),\ 5)$
$root\ (t) = 5$
$has\_children\ (t) = True$
$left\ (t)$ −−> see Figure
$right\ (t)$ −−> see Figure
$root(\ left\ (t)) = 4$

Example 3

$t = merge\ (merge\ (make\ (6),\ make\ (7),\ 8),\ \ right\ \ (merge\ (make\ (4),\ merge\ (make\ (1),\ make\ (2),\ 3),\ 5)),\ \ 9)$



$root\ (t) = 9$
$has\_children\ (t) = True$
$left\ (t)$ −−> see Figure
$right\ (t)$ −−> see Figure
$root\ (\ left\ \ (right\ \ (t))) = 1$

TYPES

TREE

FUNCTIONS

- make: INTEGER $\rightarrow$ TREE
- merge: TREE $\times$ TREE $\times$ INTEGER $\rightarrow$ TREE
- root: TREE $\rightarrow$ INTEGER
- left: TREE $\not\rightarrow$ TREE
- right: TREE $\not\rightarrow$ TREE
- has_children: TREE $\rightarrow$ BOOLEAN

PRECONDITIONS

- P1: left (t) require has_children (t)
- P2: right (t) require has_children (t)

AXIOMS

- A1: root (make (i)) = i
- A2: root (merge (t1, t2, i)) = i
- A3: left (merge (t1, t2, i)) = t1
- A4: right (merge (t1, t2, i)) = t2
- A5: has_children (make (i)) = False
- A6: has_children (merge (t1, t2, i)) = True

2 System Architecture (20 Points)

For the following two problems, describe the system architecture in the following form:

- Name one *architectural pattern* that you will use (not design pattern).
- Draw a diagram that describes your system architecture.
- Quickly explain in words how the system works.
- State the three most important advantages of using this architecture.
- State the two most important disadvantages of using this architecture.

2.1 E-mail Filter

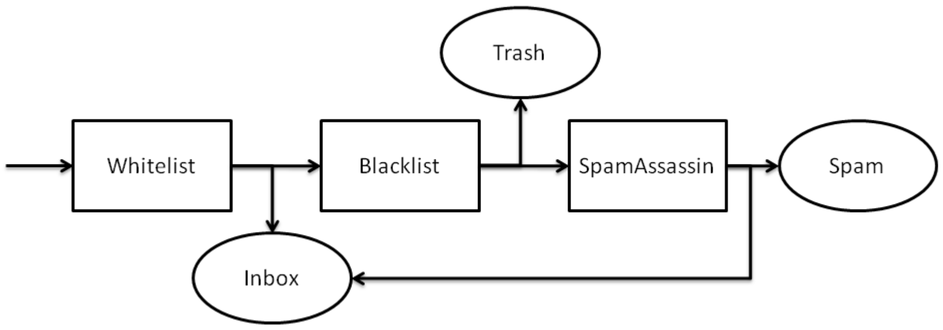
An e-mail system filters incoming e-mails with a whitelist (e-mails from senders on the whitelist are accepted), a blacklist (e-mails from senders on the blacklist are deleted), and the Spamassassin tool (e-mails that do not pass this check are marked as spam). The system will run on a single-core server machine, but may be moved to a multi-core server if the load gets too high.

Solution

Architectural Pattern Name: (1 point)

Pipe-and-Filter.

Diagram: (2 points)



Description: (2 points)

The system consists of multiple filters which are linked in sequence. An incoming email is given to the first filter. Then, each filter either moves the Email to a destination (inbox, trash, spam), or hands it off to the next filter. The emails that are left at the end will go to the inbox.

Three Most Important Advantages: (3 points)

- Scalability: Individual filters can be run in parallel when the system moves to a multi-core machine.
- Extendibility: Filters can be added or removed without affecting the overall system.

Two Most Important Disadvantages: (2 points)

- Integrity: Order of filters affects result.

2.2 Airplane Monitoring

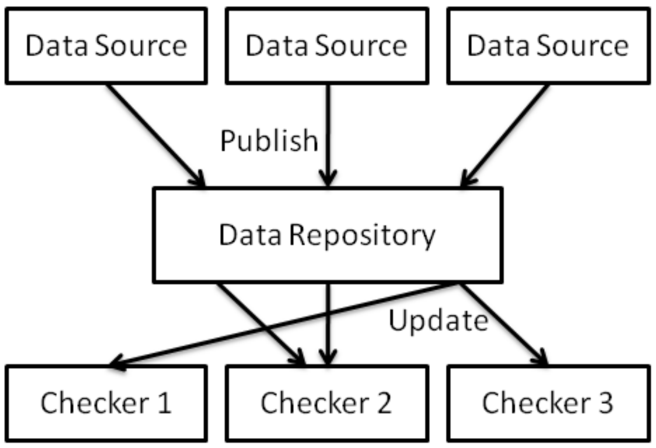
In an airplane, there are many sensors: speed, altitude, cabin pressure, fuel level, etc. The monitoring system performs different checks on the sensor data. If a problem is noticed, the system either shows a warning to the pilot (e.g. low on fuel), or in a dangerous situation may react automatically (e.g. by dropping oxygen masks). The system will run on a multi-core machine and should do the checks in near real-time when new sensor data comes in.

Solution

Architectural Pattern Name: (1 point)

Event-based.

Diagram: (2 points)



Description: (2 points)

The system consists of a data repository, which stores the current values of all sensor data. Different checkers can subscribe for changes in different data types. When the data sources generate new data, the data repository informs the subscribed checkers that new data is available. The data checkers are individual modules and thus can perform their checks in parallel as soon as new data is available.

Three Most Important Advantages: (3 points)

- Response time: Checks are performed as soon as new data is available.
- Performance: Checks can be performed in parallel.
- Extendibility: New checks can be added without changes in the system.

Two Most Important Disadvantages: (2 points)

- Integrity of checks: The checks are done independent of each other, so they could possibly launch contradicting actions as a result of the check.
- Data repository is single point of failure.

3 Testing (17 Points)

The feature *extend* of class *LINKED_LIST* is shown in the following listing, along with some of the features used in *extend*.

```
class LINKED_LIST [G]
create make

feature

  make
      -- Create an empty list.
    ensure
      count = 0
      index = 1

  first_element: like new_cell
      -- Head of list

  last_element: like first_element
      -- Tail of list

  new_cell (v: like item): LINKABLE [like item]
      -- A newly created instance of the same type as 'first_element'.

  active: like first_element
      -- Element at cursor position

  count: INTEGER
      -- Number of items in the list

  index: INTEGER
      -- Index of current cursor position (is between 1...(count+1))

  after: BOOLEAN
      -- Is there no valid cursor position to the right of cursor?
    ensure
      Result = (index = count + 1)

  is_empty: BOOLEAN
      -- Is structure empty?
    ensure
      is_empty = (count = 0)

  start
      -- Move cursor to first position.
    ensure
      index = 1

  forth
      -- Move cursor to next position.
    require
      not after
    ensure
      index = old index + 1

  put_right (v: like item)
      -- Add 'v' to the right of cursor position.
      -- Do not move cursor.
    ensure
      count = old count + 1
      index = old index

[1] extend (v: like item)
      -- Add 'v' to end.
      -- Do not move cursor.
    local
      p: like first_element
      l: like last_element
```

```
       do
[2]       p := new_cell (v)
[3]       if is_empty then
[4]          first_element := p
[5]          active := p
          else
[6]          l := last_element
[7]          if l /= Void then
[8]             l.put_right (p)
[9]             if after then
[10]               active := p
                end
             end
          end
[11]      count := count + 1
       ensure
          count = old count + 1
          index = old index
       end
```

extend adds an element to the end of a *LINKED_LIST*. *first_element* and *last_element* point to the first and the last element in the list, respectively. If the list is empty, *first_element* and *last_element* are Void. *active* points to the element at the current cursor position. If the cursor is off the list, *active* is Void.

In program analysis:

- A *definition* of a variable x (a local variable, argument or class attribute) consists of statements performing creation, initialization, assignment of a value to x or actual argument substitution if x is an argument of a feature.
- A *use* of variable x consists of statements using x without changing its value. There are two kinds of uses:
  - *P-use*: use in the predicate (decision) of an if- or loop-statement
  - *C-use*: all other uses

In the above listing, v is a passed-in argument, so line [1] is a definition of v, denoted by $v[1]$, that is, the variable name followed by line number of the definition.

In the statement $p := new\_cell\ (v)$, v is C-used, so line [2] is a C-use of v, whose value is defined in line [1]. In other words, line [1] and [2] form a def-use pair for variable v. This def-use pair is denoted by $v[1{-}2]C$, that is, the variable name, followed by two dash-separated numbers representing the definition and use location of that variable, followed by the type of use, either C or P.

Questions

(1) Please find all definitions of variables in the above listing. (1 point each)
p[2]
first_element[4]
active[5]
l[6]
active[10]
count[11]

(2) Please find all def-use pairs for the definitions listed in question (1). For each def-use pair, use the described notation io indicate it is a P-use or a C-use. (1 point each)
p[2-4]C
p[2-5]C
l[6-7]P
l[6-8]C
p[2-8]C
p[2-10]C
count[11-11]C

(3) In software testing, the All def-use criterion is a data-flow coverage criterion, it is satisfied if all def-use pairs are examined by at least one test case. Please construct a test suite which satisfies all def-use criterion for local variable *p*.

Test case No.1 (2 points)

```
create l.make
l.extend (Void)
```

Test case No.2 (2 points)

```
create l.make
l.extend (Void)
l.start
l.forth
l.extend (Void)
```