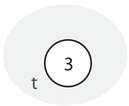
1 Abstract Data Types (11 Points)

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:

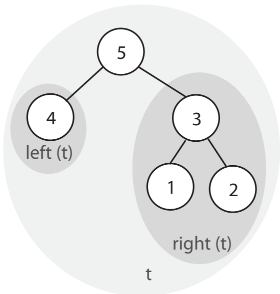
- *make*: Creation function that given an INTEGER argument *i* returns a TREE with *i* stored in the root node.
- *merge*: Given two arguments *t1* and *t2* of type TREE and a third argument *i* of type INTEGER, this function connects the two trees by adding a new root node containing *i* and storing *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

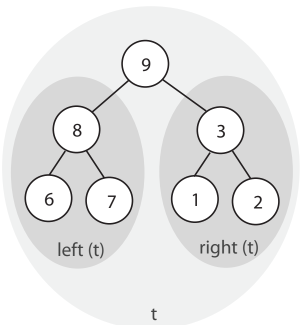
Example 2



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

Complete the ADT description below by filling in the missing parts in the FUNCTIONS, PRECONDITIONS, and AXIOMS sections. In the FUNCTIONS part of the ADT, you should add the appropriate function symbol in the dotted space. The axioms you propose should be sufficiently complete (but you do not need to prove sufficient completeness). The number of lines for preconditions and axioms may not correspond to the number of actual preconditions and axioms you have to provide.

TYPES TREE

FUNCTIONS

- make: INTEGER ...... TREE
- merge: TREE $\times$ TREE $\times$ INTEGER ...... TREE
- root: TREE ...... INTEGER
- left: TREE ...... TREE
- right: TREE ...... TREE
- has_children: TREE ...... BOOLEAN

PRECONDITIONS

- P1: ....
- P2: ....
- P3: ....
- P4: ....

AXIOMS

- A1: ....
- A2: ....
- A3: ....
- A4: ....
- A5: ....
- A6: ....
- A7: ....
- A8: ....

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.

Architectural Pattern Name:

. . . .

Diagram:

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Description:

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Three Most Important Advantages:

. . . .

. . . .

. . . .

. . . .

. . . .

Two Most Important Disadvantages:

. . . .

. . . .

. . . .

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.

Architectural Pattern Name:

. . . .

Diagram:

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Description:

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Three Most Important Advantages:

. . . .

. . . .

. . . .

. . . .

. . . .

Two Most Important Disadvantages:

. . . .

. . . .

. . . .

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.

. . . .

. . . .

. . . .

. . . .

Please find all def-use pairs, if any, for the definitions listed in question (1). For each def-use pair, use the described notation to indicate if it is a P-use or a C-use.

. . . .

. . . .

. . . .

. . . .

. . . .

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 the *all def-use criterion* for local variable `p`.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .