L1

EECS-3311 – Lab – Sorted Trees

12/28/18 12:33:20 PM

1 Goals

```
require
     Lab0 done
     read accompanying document: Eiffel101
ensure
     submitted on time
     no submission errors
rescue
     ask for help during scheduled labs
     attend office hours for TA William
```

In the first month of the course, you are required to submit a sequence of three labs:

- Lab1: *Sorted Trees*. Based on material on recursion and tree data structures from EECS2030, and EECS2011. Also introduces you to SEQ [G] of Mathmodels where G is a generic parameter and SEQ is a mathematical model of sequences.
- Lab2: *Sorted Maps*. Use of FUN [K, V] in Mathmodels
- Lab3: *Sorted* Variants. Combines material from Lab1 and Lab2

This Lab and others are accompanied by the accompanying document *Eiffel01* which you must read and understand in the first 3 weeks of the term.

Goals of Lab1:

- Develop familiarity and competence with Eiffel: the language, method and tools
- Review and code with basic OO concepts such as polymorphism, dynamic binding, static typing, generic parameters and recursion
- Use *Design by Contract* (preconditions, postconditions and class invariants) for specifying API's and Unit Testing (test driven development and regression testing) for developing reliable code.
- Constructing implementations that satisfy *specifications*
- Tools Use: Use the IDE to browse code, edit, compile, unit test with ESpec, document the design, and develop competence in the use of the debugger.
- Understand BON class diagrams for describing design decisions
- Use genericity and void safe programming constructs (**attached** and **detachable**) for abstraction and reliability
- Introduce a few Design pattern examples (iterator and template)

2 Getting started

These instructions are for when you work on one of the EECS Linux Workstations or Servers (e.g *red*). You should not compile on *red* as it is a shared server; compile on your workstation. Invoke the Eiffel IDE from the command line as `estudio18.11` (aliased to `estudio`) and the command line compiler is `ec18.11` (aliased to `ec`).

2.1 Retrieve and compile Lab1

```
> ~sel/retrieve/3311/lab1
```

This will provide you with a starter directory `sorted-tree`. The directory has the following structure:

Table 1 Lab1 Directory Structure

```
sorted-tree/
├─── model
|    ├─── node
|    |    ├─── basic_node.e
|    |    ├─── element.e
|    |    ├─── node.e
|    |    └─ tree_path.e
|    ├─── sorted_bst.e
|    ├─── sorted_rbt.e
|    └─ sorted_tree_adt.e
├─── root
|    └─ root.e
├─── sorted-tree.ecf
└─ tests
     ├─── instructor
     |    ├─── bst_extend_test.e
     |    ├─── bst_tests.e
     |    ├─── rbt_tests.e
     |    ├─── rbt_tests2.e
     |    ├─── sorted_tree_adt_tests.e
     |    ├─── student_bst_tests.e
     |    └─ student_test_node.e
     └─ student
          └─ student_tests.e
```

The three classes in **red** are **incomplete**, and you are required to complete all the *features* (queries and commands) in these classes, given the *specifications* (preconditions, postconditions and class invariants).

You can now compile the Lab.

> `estudio sorted-tree/sorted-tree.ecf &`

where `sorted-tree.ecf` is the Eiffel configuration file for this Lab. The root class (file `root.e`) looks like this:

```
class
    ROOT
inherit
    ARGUMENTS
    ES_SUITE
create
    make
feature {NONE} -- Initialization

    make
            -- Run application.
        do
            add_test (create {STUDENT_TESTS}.make)
            add_test (create {STUDENT_TEST_NODE}.make)
            add_test (create {BST_EXTEND_TEST}.make)

            -- get the above tests working,
            -- then uncomment tests below

            -- add_test (create {STUDENT_BST_TESTS}.make)
            -- add_test (create {BST_TESTS}.make)
            -- add_test (create {RBT_TESTS}.make)
            -- add_test (create {RBT_TESTS2}.make)
            show_browser
            run_espec
        end
end
```

Some of the tests are commented out. The project will compile and when you execute the Lab in workbench mode (Control-Alt-F5)[1], you will see the following *ESpec* Unit Testing report:

---

[1] Read *Eiffel101* for details

FAILED (11 failed & 2 passed out of 13)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	13
All Cases	2	13
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TESTS	
FAILED	NONE	t1: describe test t1 here
FAILED	NONE	t2: describe test t2 here
FAILED	NONE	t3: describe test t1 here
FAILED	NONE	t4: describe test t4 here
Test2	STUDENT_TEST_NODE	
PASSED	NONE	t0: create and check root node
FAILED	Check assertion violated.	t1: check that is_equal work
PASSED	NONE	t2: add Left and Right nodes to root but not in sorted order
FAILED	Check assertion violated.	t3: test inorder traversal command but it's not in sorted key K order (LL,1)(Bob,2)(LR,3)(Zak,4)(RL,5)(Alexa,6)(RR,7)
FAILED	Check assertion violated.	t4: test inorder traversal query more tests needed
FAILED	Check assertion violated.	t5: check sibling query
FAILED	Check assertion violated.	t6: check inner_child query RL is inner child of Alexa
FAILED	Check assertion violated.	t7: check outer_child root query
Test3	BST_EXTEND_TEST	
FAILED	Postcondition violated.	t1: Get extend working for value and reference keys If you get this working, ESpec will properly display remaining tests

2.2 Get the initial Unit Tests working

The Red Bar means that some of the tests fail. You must get all the tests to work and obtain a Green Bar.

- STUDENT_TESTS: you must write your own tests and we will check your tests. You may insert as many tests as you wish in this class.
- STUDENT_TEST_NODE: We provide you with some basic tests to get all the features in class BASIC_NODE working correctly. See the `To Do` hints in this class.
- BST_EXTEND_TEST: This class has a single test to help you get feature {`SORTED_BST`} `extend_node` working (see `TO DO` below).

```
class
    SORTED_BST[K -> COMPARABLE, V -> ANY]

inherit
    SORTED_TREE_ADT[K,V]

create
    make_empty

feature{NONE} -- private commands

    extend_node(a_item: TUPLE[key:K; val:V]; a_node: NODE[K,V])
            -- helper method to extend `node' with `a_item'
        do
            -- TO DO --
            check False end
        end

    remove_node(a_node: NODE[K,V])
            -- helper method to remove `node'
        local

        do
            -- TO DO --
            check False end

        end

end
```

Once you have command {`SORTED_BST`} `extend_node` and all the incomplete features in class `BASIC_NODE` working, you should now obtain a Green Bar:

PASSED (13 out of 13)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	13	13
All Cases	13	13
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TESTS	
PASSED	NONE	t1: describe test t1 here
PASSED	NONE	t2: describe test t2 here
PASSED	NONE	t3: describe test t1 here
PASSED	NONE	t4: describe test t4 here
Test2	STUDENT_TEST_NODE	
PASSED	NONE	t0: create and check root node
PASSED	NONE	t1: check that is_equal work
PASSED	NONE	t2: add Left and Right nodes to root but not in sorted order
PASSED	NONE	t3: test inorder traversal command but it's not in sorted key K order (LL,1)(Bob,2)(LR,3)(Zak,4)(RL,5)(Alexa,6)(RR,7)
PASSED	NONE	t4: test inorder traversal query more tests needed
PASSED	NONE	t5: check sibling query
PASSED	NONE	t6: check inner_child query RL is inner child of Alexa
PASSED	NONE	t7: check outer_child root query
Test3	BST_EXTEND_TEST	
PASSED	NONE	t1: Get extend working for value and reference keys If you get this working, ESpec will properly display remaining tests

2.3 Get all the remaining Unit Tests working

You may uncomment and compile the rest of the tests in class ROOT. You then are required to get all these remaining tests working.

Important Note: *To check syntax, a compile (shortcut F7) is sufficient. But it is best to freeze (Control-F7) before running unit tests. Run the unit tests often! (even after very small changes to your code). When you compile, ensure that the compilation succeeded (reported at the bottom of the IDE). When you run unit tests, ensure that all your routines terminate (can also be checked in the IDE). If you keep running the tests without halting the current non-terminating run, you will keep adding new non-terminating processes to the workstation, and the workstation will choke on all the concurrently executing processes. Study how to use your tools effectively and efficiently.*

PASSED (63 out of 63)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	63	63
All Cases	63	63
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TESTS	
PASSED	NONE	t1: describe test t1 here
PASSED	NONE	t2: describe test t2 here
PASSED	NONE	t3: describe test t1 here
PASSED	NONE	t4: describe test t4 here
Test2	STUDENT_TEST_NODE	
PASSED	NONE	t0: create and check root node
PASSED	NONE	t1: check that is_equal work
PASSED	NONE	t2: add Left and Right nodes to root but not in sorted order
PASSED	NONE	t3: test inorder traversal command but it's not in sorted key K order (LL,1)(Bob,2)(LR,3)(Zak,4)(RL,5)(Alexa,6)(RR,7)
PASSED	NONE	t4: test inorder traversal query more tests needed
PASSED	NONE	t5: check sibling query
PASSED	NONE	t6: check inner_child query RL is inner child of Alexa
PASSED	NONE	t7: check outer_child root query
Test3	BST_EXTEND_TEST	
PASSED	NONE	t1: Get extend working for value and reference keys If you get this working, ESpec will properly display remaining tests
Test4	STUDENT_BST_TESTS	
PASSED	NONE	t1: check basic tree operations for values and references
PASSED	NONE	jt0: basic checks and path and path_of queries of a tree
PASSED	NONE	jt1: remove node with two children extend is sensitive to order, out is inorder
PASSED	NONE	jt2: remove root node that has children
PASSED	NONE	jt3: test path_of
PASSED	NONE	jt4: test path
PASSED	NONE	jt5: test find_largest
PASSED	NONE	jt6: basic check for reference keys
PASSED	NONE	jt6: test tree iterator
PASSED	NONE	jt8: test tree equality, same sequence and paths
Test5	BST_TESTS	
PASSED	NONE	t0: add to tree
PASSED	NONE	t1: remove nodes without children
PASSED	NONE	t2: remove nodes with left child
PASSED	NONE	t3: remove nodes with right child
PASSED	NONE	t4: remove nodes with two children
PASSED	NONE	t5: remove nodes with two children, string version
PASSED	NONE	t6: many insertions/removals(invariant testing)
PASSED	NONE	t7: find min
PASSED	NONE	t8: find max

There are more … BST_TESTS than shown in this image …
Note: We provide you with a complete Red-Black tree (class SORTED_RBT), and the tests for this class will automatically succeed provided you get all the prior tests to pass.

2.4 List of features to implement to Specifications

Complete the following functions in BASIC_NODE (see STUDENT_TEST_NODE):

- `is_equal`
- `traverse_inorder`
- `inorder`
- `sibling`
- `inner_child`
- `outer_child`

Complete the following functions of SORTED_TREE_ADT:

- `find_largest`
- `find_smallest`
- `is_equal`

Complete SORTED_BST [K, V]

Notes:

- NODE (with features for red black trees) inherits from BASIC_NODE
- Get STUDENT_TEST_NODE working, then STUDENT_BST_TESTS. That will get the vast majority of tests to succeed.
- SORTED_RBT will work automatically

2.5 Documenting Architecture: BON/UML Class Diagrams

A critical way to document a design (and the design decisions) is via a BON class diagram. Use the EiffelStudio IDE to generate BON (or UML) class diagrams. For our Lab, the IDE generates the following:

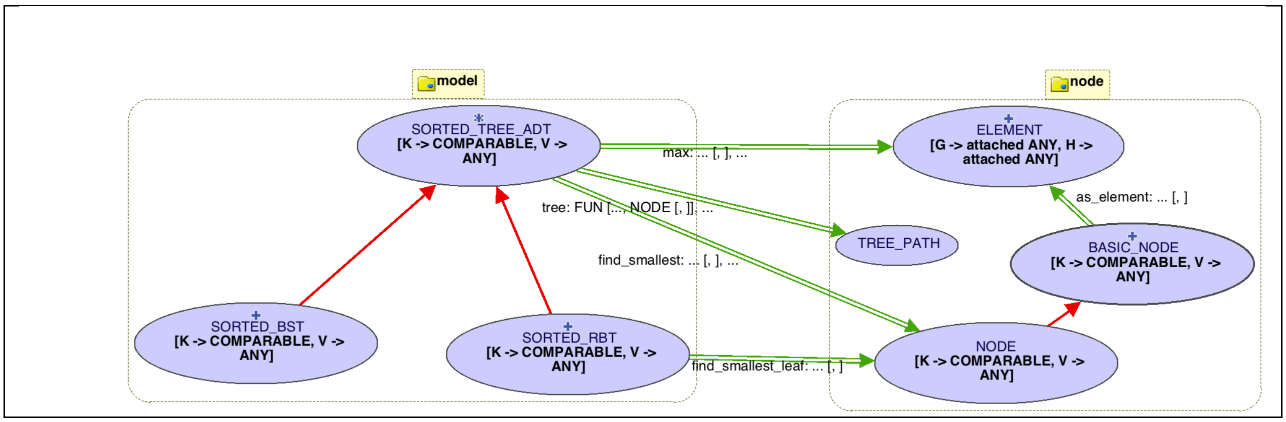


Figure 1 BON class diagram (IDE generated)

This diagram shows some important characteristics of the design:

- The diagram shows two clusters: *model* and *node*. Each cluster contains classes (shown as ellipses). The green double arrows denote *client-supplier* relationships and the red single arrow denotes an *inheritance* relationship between classes.
- The "*" decorator denotes *deferred* classes and the "+" decorator denotes *effective* classes. A deferred class has at least one *routine* (either a query or a command) that is deferred, i.e. has no implementation. Such a class cannot be instantiated at runtime, and thus does not have explicit constructors.
- Classes are always written using UPPER_CASE. *Features* (queries and commands) are written using lower case.
- Deferred class SORTED_TREE_ADT [K, V] has two *generic* parameters, K for keys and V for values. Generic parameter K is *constrained* to be COMPARABLE, needed for a sorted order.

- Most of the features of deferred class SORTED_TREE_ADT [K, V] are effected (implemented) using class NODE. Some examples are shown below:

```
deferred class
	SORTED_TREE_ADT [K -> COMPARABLE, V -> ANY]
...
feature {NONE} -- root

	root: detachable NODE [K, V]
			-- root of the tree

	as_node_array: ARRAY [NODE [K, V]]
			-- array representation of tree with nodes
		do
			create Result.make_empty
			to_array (root, Result)
			Result.compare_objects
		end

feature -- model

	model: SEQ [NODE [K, V]]
			-- tree as a linear sequence of nodes in-order
		do
			create Result.make_from_array (as_node_array)
		end

	model_path: FUN [K, SEQ [STRING_8]]
			-- path from root to key K
		do
			create Result.make_empty
			if attached root as l_root then
				across Current as a_key loop
					Result.extend ([a_key.item, path (l_root, a_key.item)])
				end
			end
		end
```

As shown above, class SORTED_TREE_ADT has a *root* node attribute, and a *model_path* query that maps each key K in the tree to a path from the root node to that key. The sorted tree also has a *model* query which is a sorted sequence of nodes in the tree. Queries *model* and *model_path* are implemented using features of class NODE [K, V]. The *model* and *model_path* queries are used to specify all the other feature of this class.

Two private features (*exported* to NONE) of SORTED_TREE_ADT are deferred and must be implemented in effective descendants such as SORTED_BST [K, V]:

```
feature{NONE} -- deferred commands

	remove_node(a_node: NODE[K,V])
			-- remove `a_node` from the tree
		require
			tree_has_node: model.has (a_node)
		deferred
		end

	extend_node(a_item: TUPLE[K,V]; a_node: NODE[K,V])
			-- extend tree rooted at `a_node`
			-- with a new node with key and value of `a_item'
		require
			tree_has_node: model.has (a_node)
		deferred
		end
```

You will effect *extend_node* and *remove_node* in descendant class SORTED_BST [K,V] (sorted binary search tree). Public features such as *extend* and *remove* use the above private features. The implementation of *extend_node* must satisfy the contracts of *extend* (shown below).

```
deferred class
	SORTED_TREE_ADT [K -> COMPARABLE, V -> ANY]
...
feature


	extend (a_item: TUPLE [key: K; val: V])
			-- extend tree to include tree with a node
			-- containing the key and value of a_item
		require
			item_unique: not has (a_item.key)
		do
			if attached root as r then
				extend_node (a_item, r)
			else
				root := create {attached NODE [K, V]}.make (a_item)
			end
		ensure
			increment_count:
				count = old count + 1
				model_path.count ~ (old model_path).count + 1
			extended: Current [a_item.key] ~ a_item.val
			inserted: model ~ old_model.inserted (a_item, index_of (a_item.key))
		end

	index_of (a_key: K): INTEGER_32
			-- returns index of a_key if it exists, returns -1 otherwise
		require
			key_exists: has (a_key)
		local
			i: INTEGER_32
		do
			i := as_array.lower
			across
				as_array as l_c
			loop
				if l_c.item.key ~ a_key then
					Result := i
				end
				i := i + 1
			end
		ensure
			index_matches_element: model [Result] ~ node_of (a_key)
		end
```

Figure 2 Contracts of {SORTED_TREE}extend

Postconditions with tags *increment_count* and *extended* are classical contracts. With Mathmodels, we can provide complete contracts. In fact, we may omit the classical contracts altogether.

2.6 Design by Contract (DbC), Class invariants and Iterator Design Pattern

Figure 2 provides contracts for command *extend* of SORTED_TREE_ADT. This class also has *invariants* that must be satisfied by all routines (whether function routines or command routines). For example, below a class invariant is:

$$\forall n \in \texttt{as\_node\_array}: \texttt{n.left.parent=n} \wedge \texttt{n.right.parent=n}$$

The invariant asserts that each left and right child of node *n* in the tree must point to its parent. In Eiffel, the invariant is written as shown below in Figure 3.

```
invariant

    child_has_correct_parents:
            -- child of every node points to its parent
        across as_node_array as n all
            (attached n.item.left as left implies left.parent = n.item)
            and then
            (attached n.item.right as right implies right.parent = n.item)
        end
        -- ∀n ∈ as_node_array:  n.left.parent=n ∧ n.right.parent=n
```

Figure 3 Class invariant for SORTED_TREE_ADT, described mathematically and in Eiffel across notation

The *left* and *right* child nodes of a parent might be *Void*. With Eiffel Void Safe settings in the ECF file, the keyword **detachable** is used to indicate a detachable type.

```
left: detachable NODE[K,V] assign set_left
        -- reference to left child
```

It is thus possible that *n.left* is *Void*. In the invariant, we must first check that *n.left* is **attached** before using it, otherwise the code will fail with a null pointer exception.

The universal quantifier `∀n` is written using the **across** construct. This construct is based on the *iterator design pattern* (to be discussed in further detail in class).

2.7 Template Design Pattern

The public command {SORTED_TREE_ADT} *extend* uses private deferred routine *extend_node* (which is to be effected in descendant classes). This is an example of the *template design pattern*: all trees use the public command *extend* to add a node to the tree, whereas *extend* itself describes the skeleton of an algorithm for the operation, deferring some steps to client subclasses (e.g. red-black trees may implement *extend_node* differently from binary search trees).

Command *extend* has meaningful preconditions and postconditions using the *model* and classical queries as shown in Figure 3.

2.8 Design architecture: BON diagram using the draw.io Template

The IDE Drawing tool is a good starting point for the BON class diagram. But we obtain a better view of the design using the draw.io tool.[2]

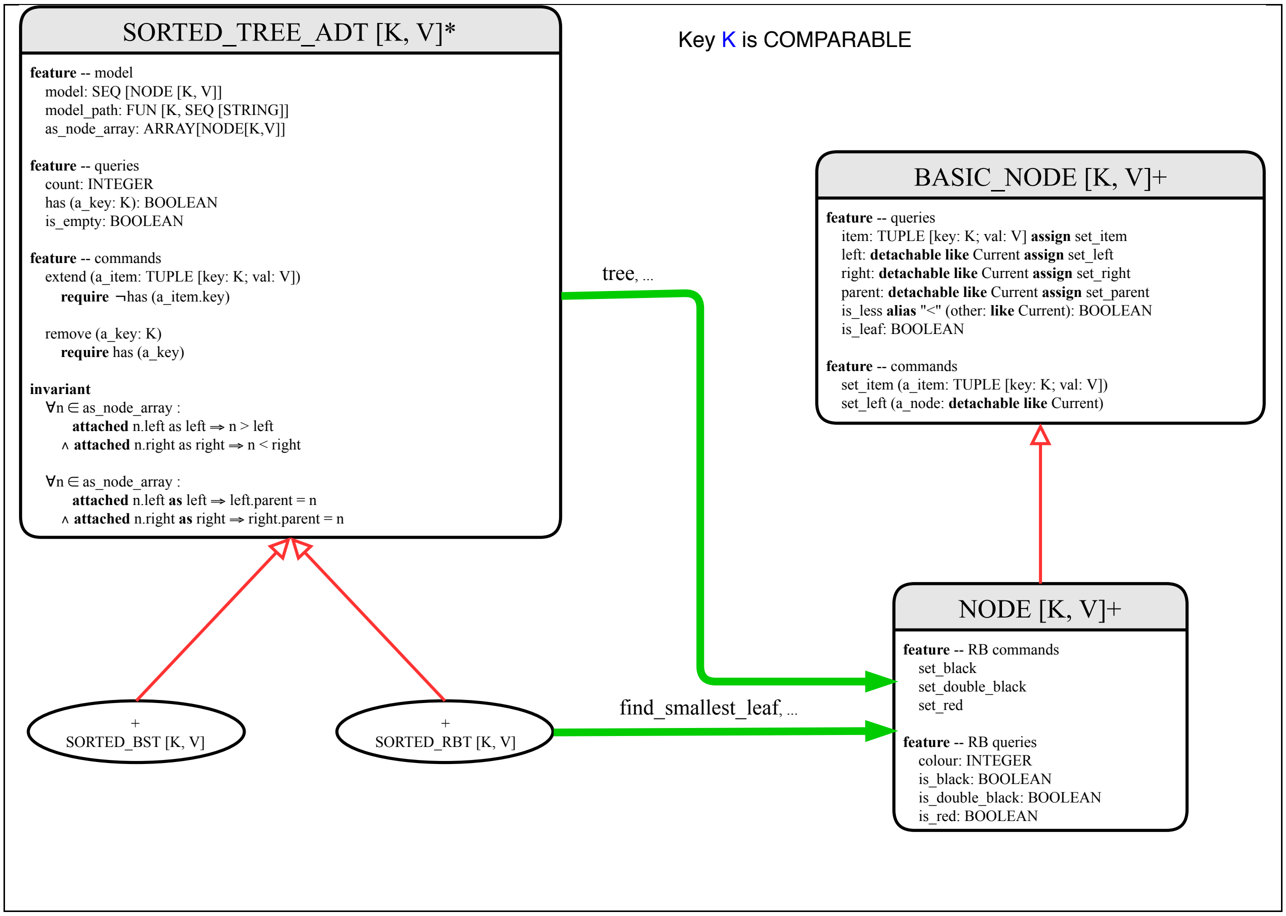


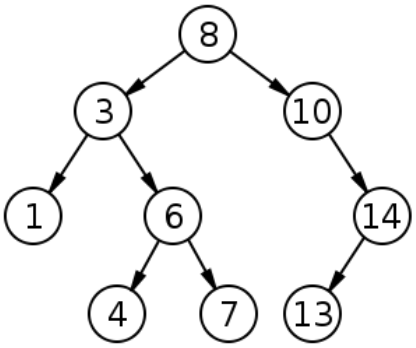
Figure 4 BON class diagram (draw.io template)

The draw.io diagram is constructed manually, which allows for the selective provision of classes, their relationships, their features, their signatures, their contracts and class invariants. The design architecture is thereby better described.

3 Sorted Trees Data Structures

You may recall tree structures from your Data Structure and Algorithms courses. Binary search trees (BSTs), also called ordered or sorted binary trees, are structures that store items (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items. We will use BSTs to lookup an item by its key K, e.g., finding the phone number (the value V) of a person by their name (the key K).

[2] http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon



From Wikipedia

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees.

A BST is not necessarily balanced, although a red back tree (RBT) is balanced with rotations. For balanced trees, on average, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables. A test to check the correctness of a binary search tree may look as follows:

```
tree: SORTED_BST [INTEGER, STRING]


t1: BOOLEAN
        do
                comment ("t1: check basic tree operations")
                create tree.make_empty
                tree.extend ([20, "twenty"])
                tree.extend ([10, "ten"])
                Result := tree [10] ~ "ten" and tree [20] ~ "twenty"
                check Result end
                Result := tree.min ~ [10, "ten"] and tree.max ~ [20, "twenty"]
                check Result end
        end
```

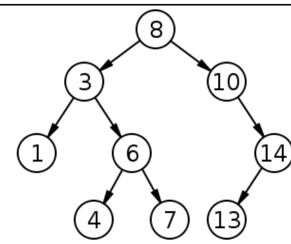
We may use the array indexing notation *tree*[10] to efficiently access the corresponding value “ten” in the tree, because query *item* is aliased to “[]”:

```
item alias "[]" (a_key: K): V
                -- returns the value in the tree associated with a_key
        do
                check
                        attached node_of (a_key) as l_node
                then
                        Result := l_node.value
                end
        ensure
                result_correct: Result ~ model [index_of (a_key)].value
                model_unchanged: model ~ old model.deep_twin
        end
```

Thus, *tree*[10] is actually a call to *tree.item*(10).

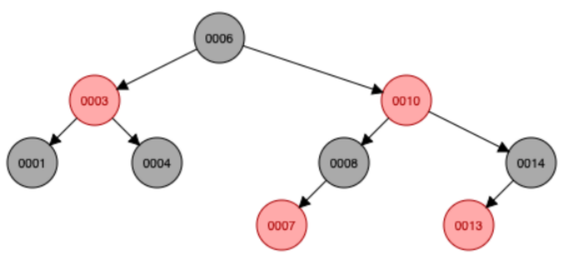
In our ESpec test classes, we often write a *setup* command that is executed before each tests. Then individual tests can check individual operations on that tree. An example of command routine setup is shown below:

```
setup
			-- is executed at the beginning of each test
		do
			create tree.make_empty
			tree.extend (8, "eight")
			tree.extend (3, "three")
			tree.extend (1, "one")
			tree.extend (10, "ten")
			tree.extend (14, "fourteen")
			tree.extend (13, "thirteen")
			tree.extend (6, "six")
			tree.extend (4, "four")
			tree.extend (7, "seven")
		end
```



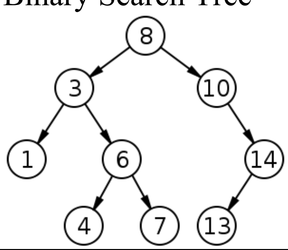
A red-black tree may have a different path from a BST (thus different models) as shown below:

Red Black Tree




```
tree.extend (8, "eight")
tree.extend (3, "three")
tree.extend (1, "one")
tree.extend (10, "ten")
tree.extend (14, "fourteen")
tree.extend (13, "thirteen")
tree.extend (6, "six")
tree.extend (4, "four")
tree.extend (7, "seven")
Path-to-7:  right, left, left
```

Binary Search Tree




```
Path-to-7:  left, right, right
```

Figure 5 Path of Red-Black Tree from root to a node vs. Binary Search Tree

3.1 Testing class BASIC_NODE

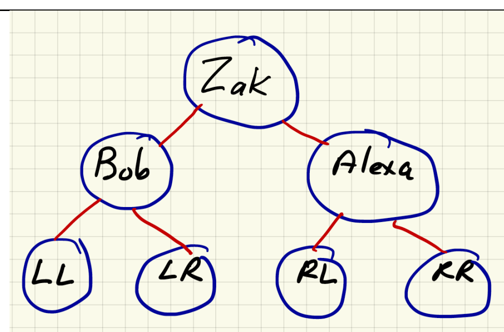
Class BASIC_NODE is used to construct tree structures, but these tree structures are not necessarily in sorted order. The IDE may be used to generate a Chart View[3] as follows:

```
class
        BASIC_NODE [K -> COMPARABLE, V -> ANY]
General
        cluster: node
        description:
                "BASIC_NODE is is used by a tree
                to store a key K and a value V.
                It also has references to its parent node,
                and left and right children nodes.
                Settings may be void-safe (where types by default
                are are attached).

                Key K and value V are both attached.
                For V to be detachable, we would have to write
                'V -> detachable ANY', and make other adjustments.

                A node does not have any secrets, thus all
                features are public. There are no invariants."
        create: make
Ancestors
        COMPARABLE*
Queries
        key: K
        value: V
        item: TUPLE [K, V]
        left: detachable BASIC_NODE [K, V]
        right: detachable BASIC_NODE [K, V]
        parent: detachable BASIC_NODE [K, V]
        inner_child: detachable BASIC_NODE [K, V]
        outer_child: detachable BASIC_NODE [K, V]
        sibling: detachable BASIC_NODE [K, V]
        is_equal (other: BASIC_NODE [K, V]): BOOLEAN
        is_greater alias ">" (other: BASIC_NODE [K, V]): BOOLEAN -- (from COMPARABLE)
        is_leaf: BOOLEAN
        is_less alias "<" (other: BASIC_NODE [K, V]): BOOLEAN
        ...
Commands
        replace_node (a_node: BASIC_NODE [K, V])
        set_item (a_item: TUPLE [K, V])
        set_left (a_node: detachable BASIC_NODE [K, V])
        set_parent (a_node: detachable BASIC_NODE [K, V])
        set_right (a_node: detachable BASIC_NODE [K, V])
        traverse_inorder
        ...
```

Figure 6 Chart View of Basic Node

Consider the following unsorted tree structure built using BASIC_NODE [K, V]:

```
feature -- setup

    LL: STRING = "LL" -- grand child
    LR: STRING = "LR"
    RL: STRING = "RL"
    RR: STRING = "RR" -- left child
    root, left, right: BASIC_NODE [STRING, INTEGER]
    nLL, nLR, nRL, nRR: BASIC_NODE [STRING, INTEGER]

    setup
            -- called before each test
        local
            l_result: BOOLEAN
        do
            create root.make ([zak, 4])
            create left.make ([bob, 2])
            create right.make ([alexa, 6])
            root.set_left (left)
            root.right := right
```



[3] See EiffelStudio101.

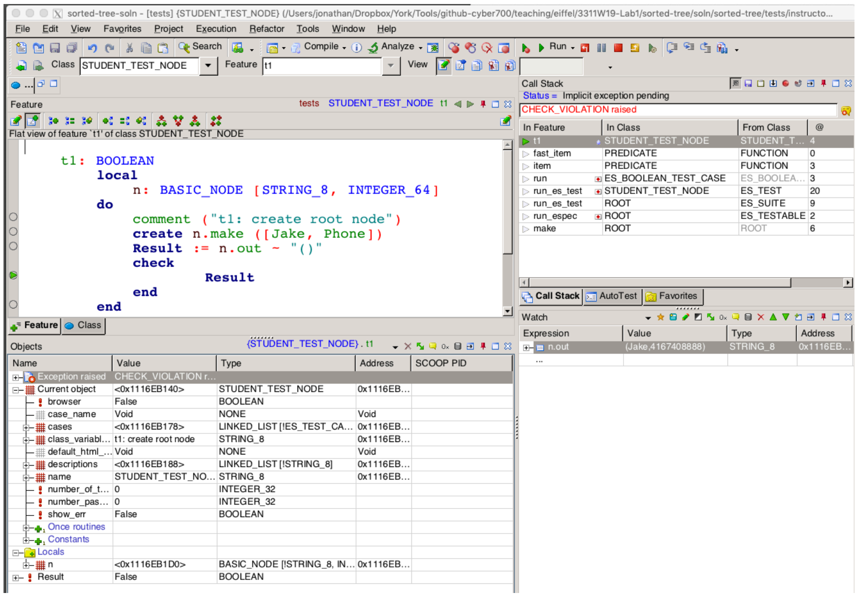
The setters for *left* and *right* node use the *assign* construct (shown below) while still conforming to Design by Contract.

```
class
	BASIC_NODE [K -> COMPARABLE, V -> ANY]
	...
feature -- queries
	item: TUPLE [key: K; val: V] assign set_item
			-- returns the current item
			-- NOTE: by default item is immutable outside of Current
			-- assign set_item allows users to call item := 4
			-- this is interpereted as item.set_item(4)

	left: detachable like Current assign set_left
			-- pointer to left child


feature -- commands

	set_item (a_item: TUPLE [key: K; val: V])
			-- sets Current item to a_item
		do
			item := a_item
		end

	set_left (a_node: detachable like Current)
			-- updates left child to a_node
			-- updates left child's parent to Current if attached
		do
			left := a_node
			if attached left as l then
				l.parent := Current
			end
		ensure
			node_set: left = a_node
			child_has_correct_parent:
				attached left as l implies l.parent = Current
		end
```

Thus, we may write `root.left := a_left` which is an alias for `root.set_left(a_left)`. The assignment must satisfy the contract of command `set_left`, otherwise an exception is generated if the contract is violated.

3.2 Initially Tests Fail

If you execute the tests (F5) then the debugger will halt at a failing test as shown below. You must learn how to use the debugger (see Eiffel101).



3.3 Void safety

You are required to read the section on Void Safety in Eiffel101 to understand the keywords **attached** and **detachable**.

4 Abstraction, Design by Contract and Design Correctness

The use of generic parameters in SORTED_TREE [K,V] is a form of *abstraction by parameterization*. We seek generality by allowing the same mechanism (or algorithm) to be adapted to many different contexts by providing it with information in that context.

Read the Section in Eiffel101 on Abstraction, DbC and Information Hiding.

5 To Submit

1. Add correct implementations as specified.
2. Work incrementally one feature at a time. Run all regression tests before moving to the next feature. This will help to ensure that you have not added new bugs, and that the prior code you developed still executes correctly.
3. Add at least 4 tests of your own to STUDENT_TESTS, i.e. don't just rely on our tests.
4. Don't make any changes to classes other than the ones specified.
5. Ensure that you get a green bar for all tests. Before running the tests, always freeze first.

You must make an electronic submission as described below.

1. On Prism (Linux), *eclean* your system, freeze it, and re-run all the tests to ensure that you get the green bar.
2. *eclean* your directory *sorted-tree* again to remove all EIFGENs.

Submit your Lab from the command line as follows:

```
submit 3311 Lab1 sorted-tree
```

You will be provided with some feedback. Examine your feedback carefully. Submit often and as many times as you like.

Remember

- Your code must compile and execute on the departmental Linux system (Prism) under CentOS7. That is where it must work and that is where it will be compiled and tested for correctness.
- Equip each test t with a *comment* ("t: …") clause to ensure that the ESpec testing framework and grading scripts process your tests properly. (Note that the colon ":" in test comments is mandatory.). An improper submission will not be given a passing grade.
- The directory structure of your folder *sorted-tree* **must** be a superset of Table 1.