EECS3311

ASSIGNMENT #1

MICHAEL WILLIAMS

211087798

JUNE 13 2016

TABLE OF CONTENTS

DOCUMENT INTRODUCTION

The purpose of this assignment was to introduce us to the new programming language, Eiffel. It was designed to test our abilities as beginners to both a new style of programming and a new environment. I was required to implement a generic map ADT and provide fully functional features to it. These features allow both basic client use and manipulation of the map as well as the ability to better secure one's program as the supplier.

By learning how to properly use these features in the assignment, I was able to assist the client in such a way that they would still be able to properly use all the features provided in the way they were intended, while at the same time securing my program from the supplier end, as to not allow the client access to any information they didn't need. I also learned to properly implement both pre and post conditions to restrict and validate the input or output of a function as to ensure both client compliance and supplier correctness.

ABSTRACT

The program created represents a generic map ADT that allows clients to create and manage same type collections of items in an organized and differentiable manner. The ADT is designed so that all items stored inside of the map are unique to one another, using client generated keys. A few basic operations are made available for client use, including the ability to insert and remove generic key-value items into the map as well as to retrieve the stored information associated with a provided key. The client can also check for the presence of specific items provided they know the key, or return all key or value data in the form of a set. All other functions of the class are made unavailable for client use and remain strictly for the supplier's end.

DATA TYPE OBJECTS

Item – a generic object consisting of a key-value pair. Both the key and value can be assigned to any type. They do not need to be the same type.

Imported- none

Exported- ITEM[K,V]

Hidden- implementation

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Set – A container used to store items. Items of the set are all unique. If duplicates of the same item are inserted, it will have the same effect as adding it once.

Imported- none

Exported- SET[I]

Hidden- implementation

Example:

$B = \{2, 4, 6, 8, 10\}$

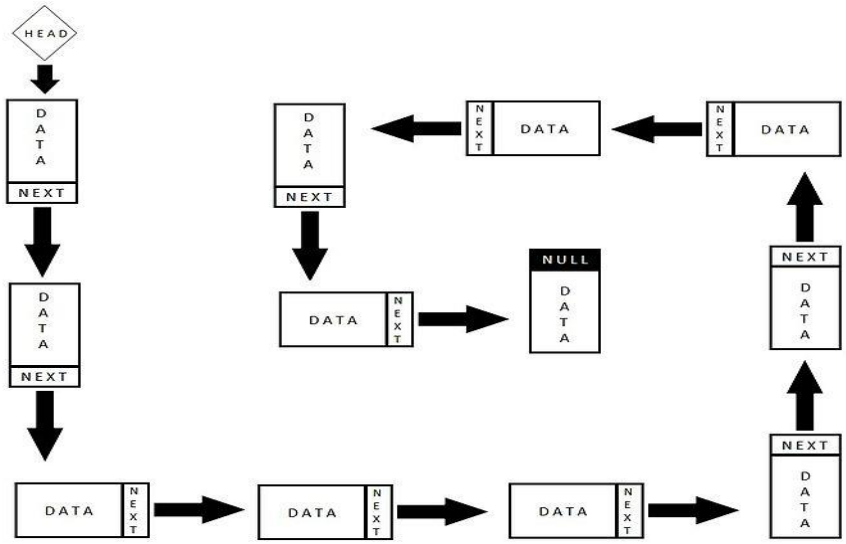
$X = \{a, b, c, d, e, f, g\}$

Linked List – collections of items stored as a list. These items are linked to one another through a reference to the next concurrent item in the list.

Imported- none

Exported- none

Hidden- hidden

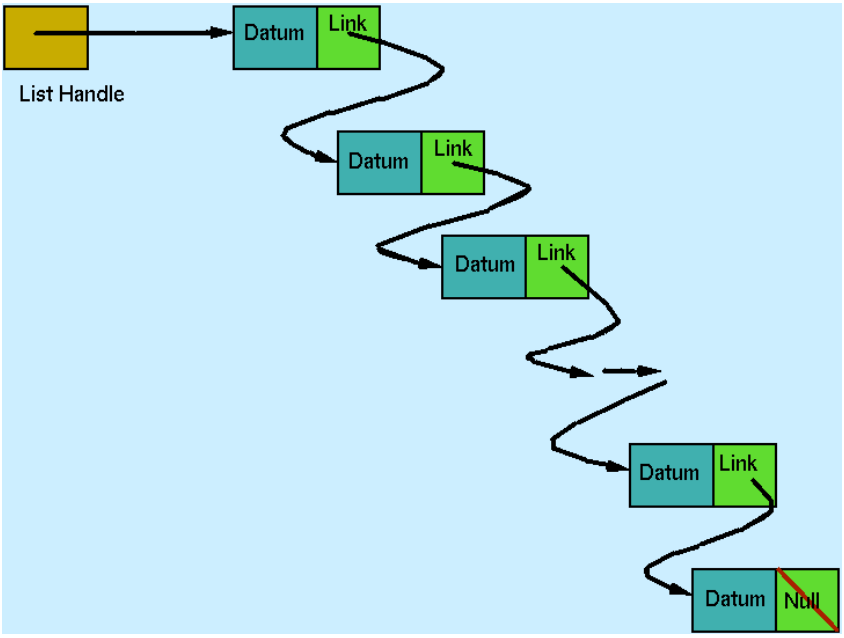


Linked set – This object has all the functionality of a linked list, while maintaining the constraints of a set. This object stores items in the same way as a linked list, but maintains the uniqueness of a set, ie if duplicates of an item are inserted, it will have the same effect as inserting the item once.

Imported- none

Exported- none

Hidden- implementation



OPERATIONS

Make

Sets up the class's initial state when an instance is created

Require: true

Ensure: **count_is_zero:** count = 0

list_is_empty: list.is_empty = true

put(key: K; value: V)

Creates a key-value pair and inserts it into the list. If key already exists, its value is overwritten with the one passed in the function

Require: **key_not_void:** key/=void

value_not_void: value/=void

list_not_full: list.full = false

Ensure: **exists_in_list:** exist(key, value) = true

entry_is_unique: exist1(key, value) = true
-only 1 copy of the key-value pair should exist

unique_key: has1(key) = true
-only 1 copy of the key-value pair should exist

list_size_dosent_increase: count = old count implies
list.count = old list.count

list_size_does_increase: count /= old count implies
list.count = old list.count + 1

count_equals_list_count: count = list.count

get(key:K):V

returns the passed key's associated value, V.

require: **key_not_void:** key /= void

ensure: **count_unchanged:** (count = old count) and (list.count = old list.count)

count_equals_list_count: count = list.count

list_unchanged: list.is_equal (old list) = true

void_result: (Result = void) implies (has1(key) = false)
-if the result is void, has1 should return false

remove(key:K)

removes the item associated with the given key.

Require: **key_not_void:**key /= void

ensure: **key_does_not_exist:** has1(key) = false

count_equals_list_size: count = list.count

list_size_dosent_decrease: count = old count implies
list.count = old list.count

list_size_does_decrease: count /= old count implies
list.count = old list.count – 1

has(key:K; val:V):BOOLEAN

does the specific item exists in the map?

Require: **key_not_void:**key /= void

check_count: count = old count and list.count = old list.count

count_equals_list_count: count = list.count

Result_not_void: Result/=void

list_unchanged: old list.is_equal (list) = true

Result_true: (Result = true) implies ((exist(key, val) = true
and exist1(key, val) = true and has1(key) = true))
-if result returns true, so should the exist, exist1 and has1 functions

Result_false: (Result = false) implies
((exist(key, val) = false and exist1(key, val) = false))

key_set:SET[K]

returns a set of key values from the map

Require: **true**

Ensure: **check_count:** count = list.count and count = old count

count_equals_list_count: count = list.count

list_unchanged: list.is_equal (old list) = true

linked_set_proper_size: Result.count = list.count

linked_set_contains_all_keys: across list as c all
result.has (c.item.getkey) end

value_set:SET[V]

Returns a set of values from the map

Require: true

Ensure: **check_count:** count = list.count and count = old count

count_equals_list_count: count = list.count

list_unchanged: list.is_equal (old list) = true

linked_set_contains_all_keys: across list as c all
result.has (c.item.getvalue) end

exist(key:K; val:V):BOOLEAN

does the specific item exists in the map?

Require: **key_not_void:** key/=void

Ensure: **check_count:** count = old count and list.count = old list.count

count_equals_list_count: count = list.count

Result_not_void: Result/=void

list_unchanged: old list.is_equal (list) = true

Result_true: (Result = true) implies
exist1(key, val) = true and has1(key) = true

Result_false: (Result = false) implies exist1(key, val) = false

exist1(key:K; val:V):BOOLEAN

is there only one pair of the given key/value combination?

Require: **key_not_void:** key/=void

Ensure: **check_count:** count = old count and list.count = old list.count

count_equals_list_count: count = list.count

Result_not_void: Result/=void

list_unchanged: old list.is_equal (list) = true

Result_true: (Result = true) implies
exist(key, val) = true and has1(key) = true

Result_false: (Result = false) implies exist(key, val) = false

has1(key:K):BOOLEAN

is there only one key with given value in the entire map?

Require: **key_not_void:**key/=void

Ensure: **check_count:** count = old count and list.count = old list.count

count_equals_list_count: count = list.count

Result_not_void: Result/=void

list_unchanged: old list.is_equal (list) = true

AXIOMS

note that any code followed by axioms are pseudo code for a better understanding

Axiom1: A new map has a count value of 0 and the linked list is empty

Create map.make results in

Map.count = 0

map.is_empty = true

Axiom2: cannot insert keys or values into the map that have void keys or values

Put(void, anything) = error

Put(anything, void) = error

Put(void, void) = error

Axiom3: removing a key that dosen't exist does not change the state of the map

Remove(key_not_in_map) results in

map = old map

Axiom4: putting an item into the map that contains another item with the same key overwrites the value

Put(key_in_map, value) results in

Map.has(key_in_map, new value) = true

Map.has(key_in_map, old value) = false

Axiom5: Key_set always returns a set of the same size as the map

Key_set.count = map.count

Axiom6: Value_set always returns a set of at most the same size as map

Value_set.count <= map.count

Class Invariants

Axiom7: count value will always be at least 0

Map.count >= 0

Axiom8: all keys in the map are unique to one another

across map all has1(key) = true

Axiom9: all items in the map are unique to one another

Across map all exist1(key, value) = true

HOW TO USE ADT

The ADT provided contains many functions available to the client, as described above. Before the client can use these functions however, they must first create and instantiate the map itself. This can be done in the code using the following:

```
create map.make
```

Once written, the client can then use the functions of the map. Below are the functions available for use. The examples also contain code for testing. For signatures, look at the operations segment of the report.

Using the put function, the user can insert an item into the map. Using the get function, one can attempt to retrieve the value associated with the sent key. The example shows this below.

```
        key := "k1"
        val:= "1"
        map.put(key, val)
        v:= map.get(key)
        comment("t1: get")
        if v=void then
            Result := false
        else
            Result := val.is_equal (v)
        end
    end
```

Using the remove function, the user can remove the given key from the map, provided it actually exists. The has function determines if the map contains the passed key-value item.

```
    key := "k1"
    val:= 1
    map.put(key, val)
    Result:= map.has(key,val)
    map.remove(key);
    comment("t2: remove record")
    Result:= Result AND NOT map.has(key,val)
end
```

The key_set function returns a set containing the keys.

```
        key := 5
        val:= 1
        map.put(key, val)

        key := 2
        val:= 3
        map.put(key, val)

        j := map.key_set

        comment("t30: key_Set record working with value=integer")
        Result := j.has(5) and j.has (2)
    end
```

The value_set function returns a set containing the values.

```
        key := 5
        val:= 1
        map.put(key, val)

        key := 2
        val:= 3
        map.put(key, val)

        j := map.value_set

        comment("t34: value_Set record working with value=integer")
        Result := j.has(1) and j.has (3)
    end
```

Last, the has1 function checks to see if the passed key is unique in the map

```
key := 2
val:= 1
map.put(key, val)

key := 5
val:= 4
map.put(key, val)



comment("t50: has1 record working with key=integer")
Result := map.has1(2) and map.has1 (5)
```

UNDESIRED EVENT DICTIONARY

The only types of errors that can occur in this program are errors originating from incorrect input. By this, I am referring to the precondition violations. These will occur if the client attempts to use a function and chooses to use input information that was specifically restricted from use by the preconditions. For example, when using the put function, the precondition states that the key cannot be void. If a client tries to use this function with key = void, a precondition violation error will occur. Aside from this type of error, the only other types of that errors can occur are syntax errors on the client's end.

DESIGN ISSUES

There are two design choices worth mentioning. The first involves the implementation of the set_key and set_value function. In these functions, I decided to create and instantiate a linked set within the functions and then, after storing the keys or values inside, return it as a set. The reason for this is because a regular set is an abstract class. Thus, it can't be used in the implementation and so requires the use of the linked set.

The other design choice worth mentioning involves an issue with the get function. According to the specifications, if a value is not found, void is to be returned. However, this can only be achieved when using a value of type String. All other types of values cannot return void. Because of this, I decided to leave no restrictions for value types that were not string. Therefore, when a value is not found, its default is simply returned.

I also created a new precondition that value could not equal void, as per the professor's recommendations. Unfortunately this now creates the new issue of not being able to distinguish whether a value does not exist or if its value is being returned from the map. For example, when using the get function, if the value associated with the key is 0, the result returned will be 0 regardless of whether the key actually exists because the default value returned by integers is 0. However it is worth noting that these changes were made specifically according to the professor's directions.

LIST OF ASSUMPTIONS

Assumptions made are associated with the get function. For this function to operate efficiently and effectively all the time, it is assumed that of all items inserted into the map, the corresponding values are never default values. By this, I mean 0 for integers, spaces for characters, etc. When these values are present in the list, and the get function is used with the associated key, problematic results are generated as described above.