Project Documentation

class: Data Structures and Algorithms

Content:

A. ***Container of the abstract data type***

1. *Specification and interface.*

Interface and domain of an Iterator of the container:

domain:

I ={it|it is an iterator over a container with elements of type TElem }

interface:

- init(it, m)
  - **description**: creates a new iterator for a container
  - **pre**: m is a map
  - **post**: it ∈ I and it points to the first element in c if c is not empty or it is not valid
- getCurrent(it, e)
  - **description**: returns the current element from the iterator
  - **pre**: it ∈ I, it is valid
  - **post**: e ∈ TElement, e is the current element from it
- next(it)
  - **description**: moves the current element from the container to the next element or makes the iterator invalid if no elements are left
  - **pre**: it ∈ I, it is valid
  - **post**: the current element from it points to the next element from the container
- valid(it)
  - **description**: verifies if the iterator is valid
  - **pre**: it ∈ I
  - **post**: valid will return true if it points to a valid map of the container and false otherwise.

Interface and domain of the abstract data type map:

domain:

M = {m|m is a map with elements e = (k,v), where k ∈ TKey and v ∈TValue}

interface:

- init(m, relation)
  - **description**: creates a new empty map
  - **pre**: true, relation which is the relation between the elements in our case the elements should be in alphabetical order
  - **post**: m∈M, m is an empty map.
- destroy(m)
  - **description**: destroys a map
  - **preconditions**: m∈M
  - **postconditions**: m was destroyed

Specification of the operations on the abstract data type map:

-add(m, k, v)
- **description**: add a new key-value pair to the map (the operation can be called put as well)
- **precondition**: m ∈ M, k ∈ TKey, v ∈ TValue
- **postcondition**: m′ ∈ M, m′ = m ∪ < k, v >
- **exception**: the function will rise an exception if there is already a pair with k as the key

- remove(m, k)
- **description**: removes a pair with a given key from the map
- **precondition**: m ∈ M, k ∈ TKey
- **postcondition** : v ∈ TValue, where, v = v', if there exist a pair <k,v'> ∈ m or m' ∈ M, m'= m\<k,v'> and the value "0" otherwise

- search(m, k)
- **description**: searches for the value associated with a given key in the map
- **precondition**: m ∈ M, k ∈ TKey
- **postcondition**: v ∈ TValue, where, v = v' if there is <k,v'> ∈ m and the value "0" otherwise

- iterator(m, it)
- **description**: returns an iterator for a map
- **preconditions**: m∈M
- **postconditions**: it ∈ I, it is an iterator over m.

- size(m)
- **description**: returns the number of pairs from the map
- **precondition** :m ∈ M
- **postcondition**: size ← the number of pairs from m

-getRoot(m)
- **description**: The function will return the root of the BST
- **precondition:** m ∈ M
- **postcondition:** getRoot ← the root of the BST

For the interface some of the function I used in the interface are private so I will enumerate then down below:

- relation (a, b)
- **description:** The function is the relation that we use at the container
- **precondition:** a ∈ TElement, b ∈ TElement
- **postcondition:** The function will return true is a>b (alphabetical) or false otherwise

- minimum (m, n)
- **description:** The function will return a not which is minimum after below the node n
- **precondition:** m ∈ M, n ∈ Node
- **postcondition:** minimum ← the minimum value from the tree below n

- sizeRec(m, nr, n)
  description: The function will return the number of elements in the container
  precondition: $m \in M$, $n \in$ Node ,$n \in$ Int
  postcondition: sizeRec $\leftarrow$ the number of pairs from m
- insertRec(m, n, e)
  description: The function will actually add a pair to the map
  precondition: $m \in M$, $n \in$ Node ,$e \in$ TElement
  postcondition: $m' \in M$, $m' = m \cup < k, v >$
- ↑relation(e1,e2)
  description: The function will define a relation between two elements
  precondition: $e1 \in$ TElement, $e2 \in$ TElement
  postcondition: relation $\leftarrow$ true if $e1 > e2$ and false otherwise
- removeRec(m, n, k)
  description: The function will actually remove a pair from the map
  precondition: $m \in M$, $n \in$ Node ,$k \in$ Tkey
  postcondition: $m' \in M$, $m' = m \setminus < k, v >$

2. Representation of the abstract data type

TElement:
  key : string
  value: string

Node:
  info: TElement
  left: ↑ BSTNode
  right: ↑ BSTNode

SortedMap:
  root: ↑ BSTNode
  relation: which in our case is alphabetical relation.

Iterator:
  sm: ↑ SortedMap
  s : stack<Node*>
  currentNode: ↑Node

The representation will be on a binary search tree of elements, where every element has a key and a value, and every element can appear only one time. The iterator will have a current position from the binary search tree.

Pseudocode for every function in the iterator:

```
subalgorithm init (it, sm) is:
        [it].sm ← sm
        @allocate Node nod
        nod ← [it].[sm].root
        while (not nod = NILL) execute
                [it].s.push(nod)
                nod ← [nod].left
        end-while
        if (not [it].s.empty()) then
                [it].currentNode ← [it].s.top()
        else
                [it].currentNode ← NILL
        end-if
end-subalgorithm

function getCurrent(it) is:
        getCurrent ← [it].[currentNode].info;
end - function

subalgorithm next(it) is:
        @allocate Node nod
        nod ← [it].s.top()
        [it].s.pop()


        if (not [nod].right = NILL) then
                nod ← [nod].right
                while (not nod = NIL) execute
                        [it].s.push(nod)
                        nod ← [nod].left
                end-while
        end-if
        if (not [it].s.empty()) then
                [it].currentNode ← [it].s.top()
        else
                [it].currentNode ← NILL
        end-if
end - subalgorithm

function valid(it) is:
        if ([it].currentNode = NILL) then
                valid ← false
        else
                valid ← true
end - function
```

Pseudocode for every function in the container:

```
subalgorithm init(m,relation) is:
        [m].root ← NILL
        [m].relation ← r
end - subalgorithm

subalgorithm add(m, k, v) is:
        @ allocate TElement something
        something.key ← k
        something.value ← v
        @ allocate String exist
        exist ← m.search(k)
        if (not exist = "0") then
                @throw error "The element is already in the container"
        end - if
        m.insertRec(m.root, something)
end - subalgorithm

function insertRec()
        if node = NIL then
                @allocate(node)
                [node].info ← e
                [node].left ← NIL
                [node].right ← NIL
        else if relation([node].info, e) then
                [node].left ← insert
                insertRec([node].left, e)
        else
                [node].right ← insert
                insertRec([node].right, e)
        end-if
        insert rec ← node
end-function

subalgorithm remove (m, k) is:
        @ allocate String exist
        exist ← m.search(k)
        if (exist = "0") then
                @throw error "The element is not in the container"
        m.removeRec(m.root, k)
        end - if
end - subalgorithm
```

```
function removeRec(m, n, k) is:
        @allocate int isRoot
        isRoot ← 0
        if (n == NILL) then
                removeRec ← n
        else if (k < [n].info.key) then
                [n].left ← removeRec([n].left, k)
        else if (k > [n].info.key) then
                [n].right ← removeRec([n].right, k)
        else:
                if ([n].left = NIL and [n].right = NIL) then
                        if ( n = [m].root)
                                [m].root = NIL
                        @delete n
                        n ← NILL
                else if ([n].right = NIL) then
                        if (n = [m].root) then isRoot ← 1
                        @allocate Node aux
                        aux ← n
                        n ← [n].left
                        delete aux
                        if (isRoot = 1) then
                                [m].root ← n
                else if ([n].left = NIL) then
                        if (n = [m].root) then isRoot ← 1
                        @allocate Node aux
                        aux ← n
                        n ← [n].right
                        delete aux
                        if (isRoot = 1) then
                                [m].root ← n
                else
                        if (n = [m].root) then isRoot ← 1
                        @allocate Node aux
                        aux ← [m].minimum([n].right)
                        [n].info = [aux].info
                        [n].right = removeRec([n].right, [aux].info.key)
                        if (isRoot = 1) then
                                [m].root ← n
                end - if
        end - if
        removeRec ← n
  end - function

function search(m, k) is:
        @allocate TElement somehting
        something.key ← k
        something.value ← "0"
        @allocate Node currentNode
```

```
        while (not currentNode = NIL and not [currentNode].info.key = key)

                if (not [m].relation([currentNode].info, something))
                        currentNode = [currentNode].right
                else
                        currentNode = [currentNode].left
        end - while
        if (currentNode = NIL) then
                search ← "0"
        search ← [currentNode].info.value

function iterator(m) is:
        @allocate Iterator{m} to it
        iterator ← it
end - function

function size(m) is:
        @allocate int nr
        nr ← 0
        [m].sizeRec(nr, [m].root)
        size ← nr
end - function

subalgorithm sizeRec(nr, n) is:
        if (n = NILL) then return;
        end - if
        nr ← nr + 1
        sizeRec ← sizeRec(nr, [n].left)
        sizeRec ← sizeRec(nr, [n].right)
end - subalgorithm

function getRoot(m) is:
        getRoot ← [m].root
end - function

function minimum(m, n) is:
        @allocate Node currentNode
        currentNode ← n
        while (not [currentNode].left = NIL)
                currentNode ← [currentNode].left
        end - while
        minimum ← currentNode
end - function

function relation(a, b) is:
        if (a.key > b.key) then
                relation ← true
        relation ← false
end - function
```

3. Complexity of the operations:

a) Container

The function **init** has the general complexity: O(1)
The function **destroy** has the general complexity: O(1)
The function **getRoot** has the general complexity: O(1)
The function **relation** has the general complexity: O(1)
The function **add** has the general complexity: O(1)
The function **insertRec** has the general complexity: O(n)
The function **remove** has the general complexity: O(1)
The function **removeRec** has the general complexity: O(n)
The function **search** has the general complexity: O(n)
The function **iterator** has the general complexity: O(1)
The function **size** has the general complexity: O(1)
The function **sizeRec** has the general complexity: O(n)
The function **minimum** has the general complexity: O(n)

b) Iterator

The function **init** has the general complexity: O(n)
The function **getCurrent** has the general complexity: O(1)
The function **next** has the general complexity: O(n)
The function **valid** has the general complexity: O(1)

<u>Computing a complexity for the search function:</u>

Best Case:

The best case is when the element that we are searching is actually on the first position in our binary search tree, having then the complexity $\Theta(1)$. Just the first number is checked, no matter how large the binary tree is.

Worst Case:

The worst case possible is that the element we are searching is actually on the last position on the binary tree, then the function will have the complexity $\Theta(n)$. We have to check all numbers from the binary tree.

Average Case:

The average case is computed by the formula:

$$\sum_{I \in D} P(I) \cdot E(I)$$

where:

- D is the domain of the problem, the set of every possible input that can be given to the algorithm, in our case {a..z}x{a..z} because k can takes values from a to z and the same v.
- I is the input data
- P(I) is the probability that we will have I as input
- E(I) is the number of operation performed by the algorithm for input I

For our example D would be the set of all possible binary trees with n leafs:
For our example I could be a subset of D in which:

- One I represents all the binary trees where the first element being the one that we are looking for
- One I represents all the binary trees where the second element is the one that we are looking for

…

P(I) is usually considered equal for every I

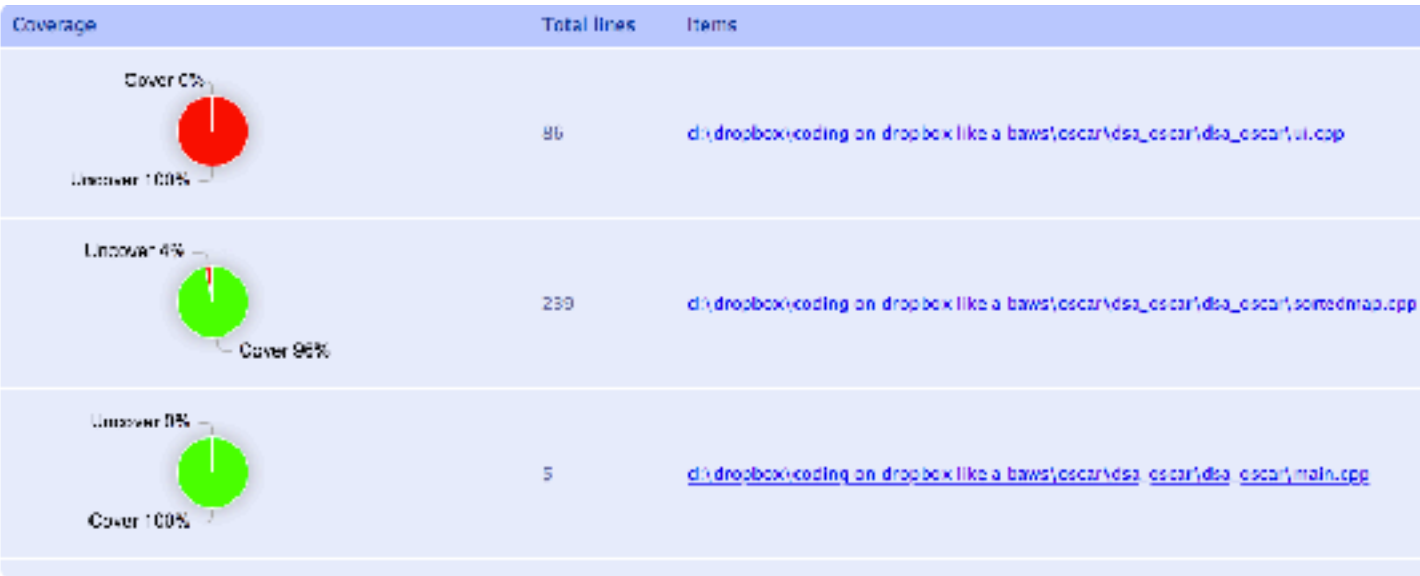
So the complexity would be something like:

$$\sum_{i=1.}^{n} (n+10) = \sum_{i=1}^{n} n + \sum_{i=1}^{n} 10 = n*n + 10*n = n^2 + 10*n \in O(n^2).$$

So the average case is actually $O(n^2)$

4. Test Coverage and Tests

Here is the code coverage (It was the best I could do):



```
}
```

For testing all the function in the container I've made a class Tests in which there will be a function for each function in container. For example the class Tests:

```
void Tests::testAll(){
    testAdd();
    testRemove();
    testSearch();
    testSize();
    testRelation();
    testIterator();

}

void Tests::testAdd(){
    SortedMap sm{&relation};
    sm.add("Ghost","A");
    sm.add("Chair","B");
    assert(sm.size() == 2);
    sm.add("Inspiration","C");
    sm.add("Apple","d");
    sm.add("Word","haha");
    assert(sm.size() == 5);
    try {
        sm.add("Ghost", "bla bla");
            assert(false);
    } catch (std::string &e){
            assert(true);
    }
}

void Tests::testSize(){
    SortedMap sm{&relation};
    assert(sm.size() == 0);
    sm.add("Ghost","A");
    sm.add("Chair","B");
    assert(sm.size() == 2);
    sm.remove("Ghost");
    assert(sm.size() == 1);
}

void Tests::testSearch(){
    SortedMap sm{&relation};
    sm.add("Ghost","A");
    sm.add("Chair","B");
    assert(sm.search("Ghost") == "A");
    assert(sm.search("Something") == "0");
}
```

```
void Tests::testIterator(){
    SortedMap sm{&relation};
   Iterator it2 = sm.iterator();
   assert(it2.valid() == false);
    sm.add("Ghost","A");
    sm.add("Chair","B");
    sm.add("Inspiration","C");
    sm.add("Apple","d");
    sm.add("Word","haha");
   Iterator it = sm.iterator();
    assert(it.valid() == true);
    assert(it.getCurrent().key == "Apple");
    it.next();
    assert(it.getCurrent().key == "Chair");
    it.next();
    it.next();
    it.next();
    it.next();
    assert(it.valid() == false);

}

void Tests::testRelation(){
    TElement a,b;
    a.key = "aa";
    b.key = "bb";
    a.value = "";
    b.value = "";
    assert(relation(a, b) == false);
    assert(relation(b, a) == true);
}

void Tests::testRemove(){
    SortedMap sm{&relation};
    sm.add("Ghost","A");
    sm.add("Chair","B");
    sm.add("Inspiration","C");
    sm.add("Apple","d");
    sm.add("Word","haha");
    sm.remove("Ghost");
    assert(sm.size() == 4);
    sm.remove("Apple");
    assert(sm.size() == 3);
    sm.remove("Chair");
    assert(sm.size() == 2);
    sm.remove("Inspiration");
    assert(sm.size() == 1);
    sm.remove("Word");
```

```
        assert(sm.size() == 0);
        try {
            sm.remove("Ghost");
                assert(false);
        } catch (std::string &e){
                assert(true);
        }
}
```

B. Application

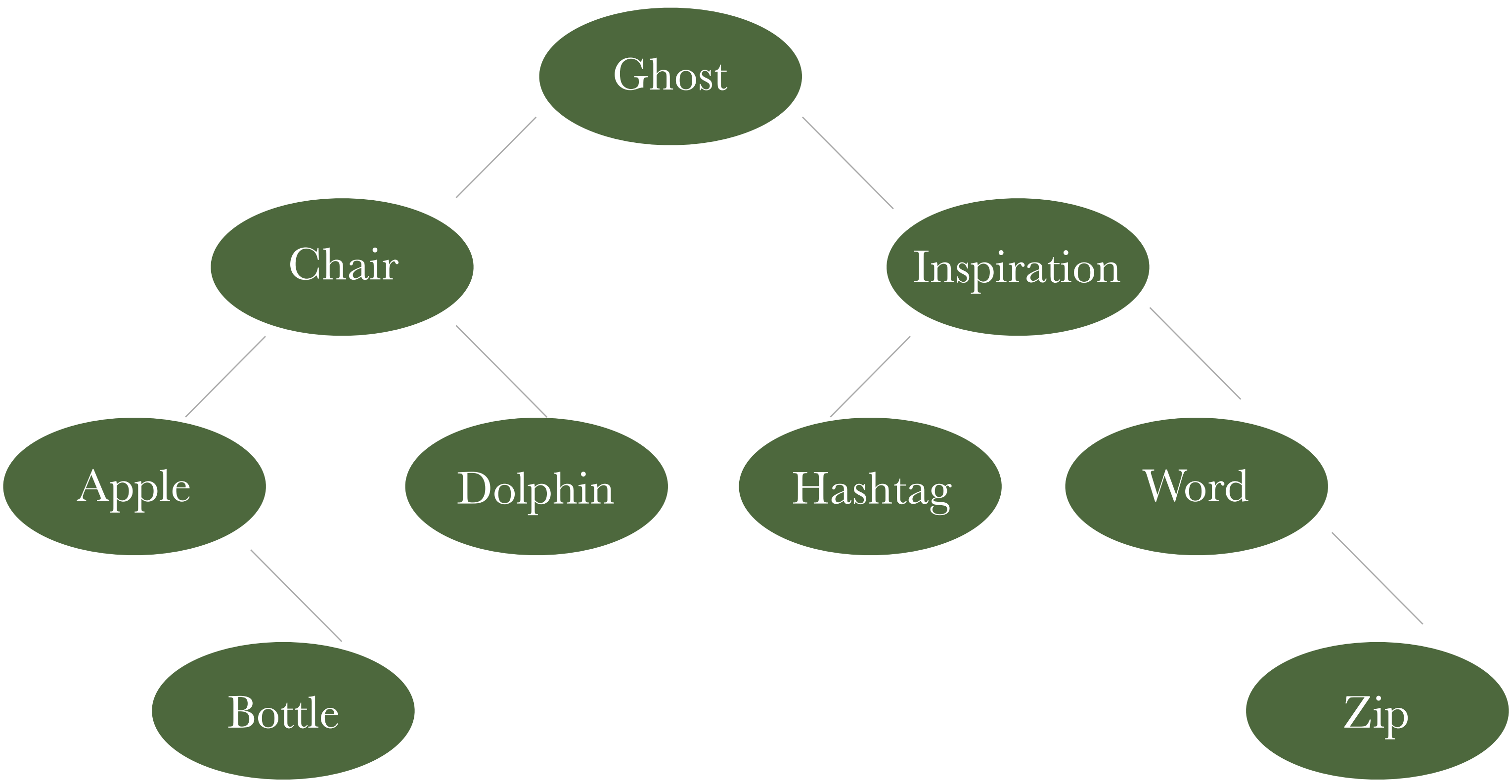
1. Problem statement.

The given problem:

ADT SortedMap – implementation on a binary search tree.

The problem that I thought about:

We would like to create a dictionary (for example the Macmillan Dictionary). In this application you can find a given word and read its definition, add a new word with its definition, and remove a word. The reason why I chose this problem is that if you want to search for a word (key) it would be faster because you don't have to search the whole tree you only go on the branch that suits the word you are searching for. At the same time the problem I've chose suits my container because the key of the map is the word and the value of the map is the definition of that word, and it is better being sorted because the dictionaries are always sorted in an alphabetical order. Down below is a representation of a few words and how the program will store them and will be ready for testing on the application:



2. Solution for the chosen problem:

In this part I will write all the user interface functions:

```
subalgorith readFromFile(m) is:
   fileName ← "/Users/galoscar/Documents/College/Semester 2/Data
                  Structures and Algorithms/DSAProject/DSAProject/Words.txt"
   line ← “”
   @open_for_reading(file)
   if not file.isOpen
      @print the message "Something went wrong and the file wasn't open”
   end-if
   while @getline(file, line) do
      @initialize found1, found2
      key ← “”
      value ← “”
      found1 ← line.find("|")
      found2 ← line.find("\n")
      key ← line.substr(0,found1)
      value ← line.substr(found1+1, found2)
      add(m, key, value);
   end-while
end - subalgorithm

subalgorithm printMenu() is:
   @print the message "1 - Add a new word into the database"
   @print the message "2 - Show a meaning of a word by its key”
   @print the message "3 - Delete a word from the database"
   @print the message"4 - Display the number of elements in the dictionary”
   @print the message”5 - Display all words in alphabetically order”
   @print the message "0 - Exit"
end - subalgorithm

subalgorithm addWord(m) is:
   @print the message "Enter the word (key): "
   key ← “”
   @read key from keyboard
   @print the message "Enter the definition of the word (value)"
   Value ← “”
   @read value from keyboard
   add(m, key, value)
end-subalgorithm

subalgorithm removeWord(m) is:
   @print the message "Enter the word (key): "
   key ← “”
   @read key from keyboard
   remove(key)
end-subalgorithm
```

```
subalgorithm displayWord(m) is:
   @print the message "Enter the word to receive a definition: "
   Key ← “”
   @ read key from keyboard
   value ← “”
   value ←search(m, key);
   if value ← “0" then
      @print the message "There is no such word in our database"
   else
      @print the message "For the given word: “, key, “The definition is”, value
   end-if
end-subalgorithm

subalgorithm displayNoWord(m) is:
   @print the message "The number of words in the database is : “, size(m)
end-subalgorithm

subalgorithm displayFromIt(m) is:
  Iterator it ← iterator(m);
  while it.valid() = true do
          @ print “Word: “, it.getCurrent().key, " -> Meaning" it.getCurrent().value
          it.next()
  end-while
end-subalgorithm

subalgorithm run(m) is:
   readFromFile(m)
   while true do
      printMenu()
      command ← 0
      @print the message "Input the command: "
      @read command from keyboard
      if command = 0 then
         @print the message "Thank you for using the program"
         @break
      try
         if command = 1 then
            addWord(m);
         end-if
         if command = 2
            displayWord(m);
         end-if
         if command = 3 then
            removeWord(m);
         end-if
         if command = 4 then
            displayNoWord(m);
         end-if
```

```
            if command = 5 then
                    displayFromIt(m);
            end-if
        catch (exception) {
           @print exception
        end-try
    end-while
end-subalgorithm
```