Name: PLOSCAR SERGIU RAZVAN

Group: 812

Task: 30

Project Documentation
Data Structures and Algorithms

Contents

1. Task

ADT Sorted Multi Map - implementation on a hash table, collision resolution by separate chaining

2. ADT Specification

- SMM= {smm| smm is a Sorted Multimap with pairs TKey, TValue, where we can define a relation R on the set of all possible keys}

- The general elements of the container are pairs of TKey, TValue
  The interface for TKey contains the following operations:

  - assignment ($k_1 \leftarrow k_2$)
    pre: $k_1, k_2 \in$ TKey
    post: $k_1' = k_2$

  - equality ($k_1 = k_2$)
    pre: $k_1, k_2 \in$ TKey
    post:

$$equal \begin{cases} True, if \ \ k_1 = k_2 \\ False, otherwise \end{cases}$$

  The interface for TValue contains the following operations:

  - assignment ($v_1 \leftarrow v_2$)
    pre: $v_1, v_2 \in$ TValue
    post: $v_1' = v_2$

  - equality ($v_1 = v_2$)
    pre: $v_1, v_2 \in$ TValue
    post:

$$equal \begin{cases} True, if \ v_1 = v_2 \\ False, otherwise \end{cases}$$

- Iterator = { it | it – iterator over Sorted Multimap }

3. ADT Interface

- Sorted MultiMap

  - init (smm, R)
    pre: R – relation on the set of all possible keys
    post: smm ∈ SMM, smm = ø

  - destroy (smm)
    pre: smm ∈ SMM
    post: smm was destroyed (allocated memory was freed)

  - add (smm, k, v)
    pre: smm ∈ SMM, k ∈ TKey, v ∈ TValue
    post: the pair <k,v> was added into smm

  - remove (smm, k, v)
    pre: smm ∈ SMM, k ∈ TKey, v ∈ TValue
    post: the pair <k,v> was deleted from smm (if it was in it)

  - search (smm, k, l)
    pre: smm ∈ SMM, k ∈ TKey, l ∈ $\mathbb{L}$
    post:
    $\begin{cases} true\ and\ l\ is\ the\ list\ of\ values\ associated\ with\ k, if\ k\ is\ in\ smm \\ false\ and\ l = \emptyset\ otherwise \end{cases}$
  - size(smm)
    pre: smm ∈ SMM
    post: returns the number of pairs that are currently in the smm

  - keys(smm,l)
    pre: smm ∈ SMM, l ∈ $\mathbb{L}$
    post: l is the list of unique keys found in the smm

  - values(smm,l)
    pre: smm ∈ SMM, l ∈ $\mathbb{L}$
    post: l is the list of unique values found in the smm

- pairs(smm,l1,l2)
  pre: smm ∈ SMM, l1 ∈ L, l2 ∈ L
  post: On the same position of l1 and l2 we have the key with respective value. Together l1 and l2 contain all the unique pairs found in the smm

- iterator (smm, it)
  pre: smm ∈ SMM
  post: it ∈ Iterator, it is an iterator over smm

- Iterator

- init (it, smm)
  pre: smm ∈ SMM
  post: it ∈ Iterator, it – iterator over smm pointing to first non-empty node

- next (it)
  pre: it ∈ Iterator, it is a valid iterator
  post: it' – pointing to the next element

- valid (it)
  pre: it ∈ Iterator
  post: $valid(it) = \begin{cases} True\ if\ it\ valid \\ False, otherwise \end{cases}$

- getCurrent (it, n)
  pre: it ∈ Iterator
  post: n ∈ Node, n – the current node pointed by it

4. ADT Representation

- Node:
- key: TKey
- value: TValue
- next: ↑Node

- HashTable:
- elems: ↑Node[]
- capacity: Integer
- nrofpairs: Integer
- Relation: TRelation
- hashFunction: TFunction

- Iterator:
- smm : ↑SMM
- pos: Integer
- currentPos: ↑Node

5. Pseudocode implementation

subalgorithm init(smm,R):

    @smm.nrofpairs <- 0

    @smm.capacity <- 53

    @allocate(elems)

    @smm.Relation <- R

end-subalgorithm

subalgorithm destroy(smm)

    @free smm.elems

end-subalgorithm

```
subalgorithm add(smm,k,v)
    pos <- hashFunction(k)
    smm->elems[pos].temp <- smm->elems[pos].head
    wasAdded <- false;
    while ( smm->elems[pos].temp != NIL && was Added == false )
        if( smm->Relation(k,smm->elems[pos].temp->key) == true )
        smm->elems[pos].addNodeBefore(k,v,smm->elems[pos].temp)
        was Added <- true
        end-if
        else
        smm->elems[pos].temp <- smm->elems[pos].temp->next
        end-else
    end-while
    if( smm->elems[pos].head == NIL )
    smm->elems[pos].addNodeFirst(k,v)
    end-if
    else
        if( was Added == false )
        smm->elems[pos].addNodeEnd(k,v)
        end-if
    end-else
    smm->nrofpairs<- smm->nrofpairs+1
end-subalgorithm
```

```
subalgorithm remove(smm,k,v)
    pos <- hashFunction(k)
    smm->elems[pos].delNode(k,v)
    smm->nrofpairs—
end-subalgorithm

subalgorithm search(smm,k,v,l)
    pos <- hashFunction(k)
    smm->elems[pos].temp <- smm->elems[pos].head
    while (smm->elems[pos].temp != NIL )
        if (smm->elems[pos].temp->key == k)
            l.push_back(smm->elems[pos].temp->value)
        end-if
        smm->elems[pos].temp = smm->elems[pos].temp->next
    end-while
    if ( l.size() > 0 )
        search <- true
    else
        search <- false
end-subalgorithm

subalgorithm size(smm)
    size <- smm.nrofpairs
end-subalgorithm
```

```
subalgorithm keys( smm,l )
    for ( i = 0; i < smm.getCapacity() ; 1 )
        smm->elems[i].temp <- smm->elems[i].head
        while ( smm->elems[i].temp != NIL )
            exists <- false
            for ( auto k : l )
                if( k == smm->elems[i].temp->key )
                    exists <- true
                end-if
            end-for
            if( exists == false )
                l.push_back ( smm->elems[i].temp->key )
            end->if
            smm->elems[i].temp <- smm->elems[i].temp->next
        end-while
    end-for
end-subalgorithm

subalgorithm values( smm,l )
    for ( i = 0; i < smm.getCapacity() ; 1)
        smm->elems[i].temp <- smm->elems[i].head
        while ( smm->elems[i].temp != NIL )
            exists <- false
            for ( auto v : l )
```

```
                    if( v == smm->elems[i].temp->value )
                        exists <- true
                    end-if
                end-for
                if( exists == false )
                    l.push_back ( smm->elems[i].temp->value )
                end->if
                smm->elems[i].temp <- smm->elems[i].temp->next
            end-while
        end-for
end-subalgorithm

sublalgorithm pairs(smm,l1,l2)
        for( i=0; i < smm->getCapacity(); 1 )
            smm->elems[i].temp <- smm->elems[i].head
            while (smm->elems[i].temp != NIL )
                exists1 <- false
                exists2 <- false
                for ( auto a: l1 )
                    if( a == smm->elems[i].temp->key )
                        exists1<-true
                    end-if
                end-for
                for ( auto a: l2 )
```

```
                    if( a == smm->elems[i].temp->value )
                        exists2<-true
                    end-if
                end-for
                if( exists1 == false || exists2 == false )
                    l1.push_back(smm->elems[i].temp->key)
                    l2.push_back(smm->elems[i].temp->value)
                end-if
                smm->elems[i].temp <- smm->elems[i].temp->next
            end-while
        end-for
```
end-subalgorithm

subalgorithm hashFunction(smm, k)

```
alphabet <-
"0abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
pos <- alphabet.find(k)
if ( pos != -1 )
    hashFunction<- pos
end-if
else
    hashFunction <- 0
end-else
```
end-subalgorithm

```
subalgorithm Alphabetical(smm,k1,k2)
        if ( k1 < k2 )
                Alphabetical <- true
        end-if
        else
                Alphabetical <-false
        end-else
end->subalgorithm

subalgorithm Iterator(smm, it)
        @it.pos<- 0
        @it.smm <- smm
        ok <- false
        while ( ok == false && it.pos < 53 )
                if( smm.getElems()[it.pos].head == NIL )
                        it.pos<- it.pos + 1
                end-if
                else
                        it.currentPos <- smm.getElems()[ it.pos ].head
                        ok <- true
                end-else
        end-while
        if (  it.pos >= 53 )
                it.currenPos = NIL
```

```
        end-if
end-subalgorithm


subalgorithm next( it )
        if ( it.currentPos -> next != NIL )
                it.currentPos <- it. currentPos -> next
        end-if
        else
                it.pos <- it.pos + 1
                ok <- false
                while ( ok == false && it.pos < 53 )
                        if( it.smm.getElems()[it.pos].head == NIL )
                                it.pos <- it.pos+1
                        end-if
                        else
                                it.currentPos <- it.smm.getElems()[it.pos].head
                                ok <- true
                        end-else
                end-while
                if( it.pos >= 53 )
                        it.currentPos <- NIL
                end-if
        end-else
end-subalgorithm
```

subalgorithm valid(it)

valid <- (it.currentPos != NIL)

end-subalgorithm

subalgorithm getCurrent (it)

if (it->valid())

getCurrent <- it.currentPos

end-if

end-subalgorithm

6. Operation complexities

Add

- Best Case: O(1) - We add it as the first node
- Average Case O(n) – We go through the nodes till we find the correct position to insert
- Worst Case O(n) – The pair needs to be added at the end

Remove

- Best Case: O(1) - We delete the first node
- Average Case O(n) – We go through the nodes till we find the correct position to delete
- Worst Case O(n) – The last node needs to be deleted

Search : O(n) - we go through all nodes at the correct position in the hash table and check every value

Size: O(1) – we just return the size of the container that we've been updating after every add/remove operation

Iterator: O(m* n) where m is the capacity

Iterator

Init: O(m* n) where m is the capacity

Next: O(m* n) where m is the capacity

Valid: O(1) we just verify an expression

getCurrent: O(1) we just return the element

7. Tests

```
#include "SMM.h"
#include <assert.h>
#include "Test.h"

void Test::testAdd_and_Iterator()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	Iterator j{ smm };
	assert(j.valid() == false);
	assert(smm.size() == 0);
	smm.add("ana", "hi ana");
	assert(smm.size() == 1);
	smm.add("ada", "hi ada");
	assert(smm.size() == 2);
	smm.add("axa", "hi axa");
	assert(smm.size() == 3);
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	smm.add("bd", "test");
	smm.add("bda", "test2");
	assert(smm.size() == 7);
	int nr = 0;
	Iterator i{ smm };
	while (i.valid())
	{
	if (nr == 0)
	assert(i.getCurrent()->key == "ada" && i.getCurrent()->value == "hi ada");
	if (nr == 1)
	assert(i.getCurrent()->key == "ana" && i.getCurrent()->value == "hi ana");
	if (nr == 2)
	assert(i.getCurrent()->key == "axa" && i.getCurrent()->value == "hi axa");
	if (nr == 3)
	assert(i.getCurrent()->key == "bailey" && i.getCurrent()->value == "Email");
	if (nr == 4)
	assert(i.getCurrent()->key == "bd" && i.getCurrent()->value == "test");
	if (nr == 5)
	assert(i.getCurrent()->key == "bda" && i.getCurrent()->value == "test2");
	if (nr == 6)
	assert(i.getCurrent()->key == "benny" && i.getCurrent()->value == "cool");
	i.next();
	nr++;
	}
}
```

```
void Test::testRemove()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	smm.add("ana", "hi ana");
	smm.add("ada", "hi ada");
	smm.add("axa", "hi axa");
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	smm.remove("axa", "hi axa");
	assert(smm.size() == 4);
	smm.remove("ada", "hi ada");
	assert(smm.size() == 3);
	smm.remove("bailey", "Email");
	assert(smm.size() == 2);
	int nr = 0;
	Iterator i{ smm };
	while (i.valid())
	{
	if (nr == 0)
	assert(i.getCurrent()->key == "ana" && i.getCurrent()->value == "hi ana");
	if (nr == 1)
	assert(i.getCurrent()->key == "benny" && i.getCurrent()->value == "cool");
	i.next();
	nr++;
	}
}

void Test::testSearch()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	smm.add("ana", "hi ana");
	smm.add("ana", "hello ana");
	smm.add("ana", "hey there ana");
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	std::vector<std::string> l;
	smm.search("ana", l);
	int nr = 0;
	for (auto email : l)
	{
		if (nr == 0)
			assert(email == "hi ana");
		if (nr == 1)
			assert(email == "hello ana");
		if (nr == 2)
			assert(email == "hey there ana");
		nr++;
	}
	std::vector<std::string> k;
	smm.search("10ana", k);
	assert(k.size()==0);
}

void Test::testKeys()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	smm.add("ana", "hi ana");
	smm.add("ana", "hello ana");
	smm.add("ana", "hey there ana");
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	std::vector<std::string> l;
```

```
	smm.keys(l);
	int nr = 0;
	for (auto key : l)
	{
		if (nr == 0)
			assert(key == "ana");
		if (nr == 1)
			assert(key == "bailey");
		if (nr == 2)
			assert(key == "benny");
		nr++;
	}
}

void Test::testValues()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	smm.add("ana", "hi");
	smm.add("ana", "hi");
	smm.add("ana", "hey");
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	std::vector<std::string> l;
	smm.values(l);
	int nr = 0;
	for (auto value : l)
	{
		if (nr == 0)
			assert(value == "hi");
		if (nr == 1)
			assert(value == "hey");
		if (nr == 2)
			assert(value == "Email");
		if (nr == 3)
			assert(value == "cool");
		nr++;
	}
}

void Test::testPairs()
{
	SortedMultiMap smm{ SortedMultiMap::Alphabetical };
	smm.add("ana", "hi");
	smm.add("ana", "hi");
	smm.add("ana", "hey");
	smm.add("benny", "cool");
	smm.add("bailey", "Email");
	std::vector<std::string> k;
	std::vector<std::string> v;
	smm.pairs(k, v);

	for (int i=0;i<k.size();i++)
	{
		if (i == 0)
			assert(k[i]=="ana" && v[i] =="hi");
		if (i == 1)
			assert(k[i] == "ana" && v[i] == "hey");
		if (i == 2)
			assert(k[i] == "bailey" && v[i] == "Email");
		if (i == 3)
			assert(k[i] == "benny" && v[i] == "cool");
```

```
	}
}

void Test::testAll()
{
	testAdd_and_Iterator();
	testRemove();
	testSearch();
	testKeys();
	testValues();
	testPairs();
}
```

8. Problem Statement

You are put in charge of administrating an email database where each user owning an account can receive messages. You must make sure that all incoming messages reach their destination and that for any account all received messages are stored.

9. Problem Explanation

The hash function will associate the emails (pairs of email address and message – a string) to the corresponding position in the database (the hash table). Because an address can obviously receive more than one email, all emails received by an address will be stored in a linked list (separate chaining).

10. Problem Solution

subalgorithm add(smm,k,v)

pos <- hashFunction(k)

smm->elems[pos].temp <- smm->elems[pos].head

wasAdded <- false;

while (smm->elems[pos].temp != NIL && was Added == false)

if(smm->Relation(k,smm->elems[pos].temp->key) == true)

smm->elems[pos].addNodeBefore(k,v,smm->elems[pos].temp)

```
            was Added <- true
            end-if
            else
            smm->elems[pos].temp <- smm->elems[pos].temp->next
            end-else
        end-while
        if( smm->elems[pos].head == NIL )
        smm->elems[pos].addNodeFirst(k,v)
        end-if
        else
            if( was Added == false )
            smm->elems[pos].addNodeEnd(k,v)
            end-if
        end-else
        smm->nrofpairs<- smm->nrofpairs+1
```

end-subalgorithm

11. Solution Complexity

Add

- Best Case: O(1) - We add it as the first node
- Average Case O(n) – We go through the nodes till we find the correct position to insert
- Worst Case O(n) – The pair needs to be added at the end