Data Structures and Algorithms Project

26.**ADT SortedList** – implementation on a hash table, collision resolution by separate chaining.

**1.ADT SortedList**

**Domain**

L = {l|l is a list with elements of type TElem, each having a unique position in l of type Integer}

**Interface**

**init(l, r)**

* descr: creates a new, empty list
* pre: r ∈ R
* post: l ∈ L, l is an empty list

**first(l)**

* descr: returns the Integer of the first element
* pre: l ∈ L
* post: first ← p ∈ Integer p = ( the position of the first element from l if l 6= ∅, ⊥ otherwise)

**last(l)**

* descr: returns the Integer of the last element
* pre: l ∈ L post: last ← p ∈ Integer p = ( the position of the last element from l if l 6= ∅, ⊥ otherwise)

**valid(l, p)**

* descr: checks whether a Integer is valid in a list
* pre: l ∈ L, p ∈ Integer
* post: valid ← ( true if p is a valid position in l false otherwise)

**getElement(l, p, e)**

* descr: returns the element from a given Integer
* pre: l ∈ L, p ∈ Integer, valid(p)
* post: e ∈ TElem, e = the element from position p from l throws: exception if p is not valid

**position(l, e)**

* descr: returns the Integer of an element
* pre: l ∈ L, e ∈ TElem
* post: position ← p ∈ Integer p = ( the first position of element e from l if e ∈ l ⊥ otherwise

**insert(l, e)**

* descr: inserts a new element
* pre: l ∈ L, e ∈ TElem
* post: l0 ∈ L, l0 the new list after the element e was added to the l

**remove(l, p, e)**

* descr: removes an element from a given position from a list
* pre: l ∈ L, p ∈ Integer, valid(p)
* post: e ∈ TElem, e is the element from position p from l, l 0 ∈ L, l’ = l - e. throws: exception if p is not valid

**search(l, e)**

* descr: searches for an element in the list
* pre: l ∈ L, e ∈ TElem
* post: search ← ( true if e ∈ l false otherwise)

**isEmpty(l)**

* descr: checks if a list is empty
* pre: l ∈ L
* post: isEmpty ← ( true if l = ∅ false otherwise)

**size(l)**

* descr: returns the number of elements from a list
* pre: l ∈ L
* post: size ← the number of elements from l

**SortedList Iterator**

**init(l, it)**

* descr: returns an iterator for a list
* pre: l ∈ L
* post:it ∈ I, it is an iterator over l

**getCurrent(it, e)**

* descr: returns the current element from the iterator
* pre: it ∈ I, it is valid
* post: e ∈ TElem, e is the current element from it

**next(it)**

* descr: moves the current element from the list 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 list

**valid(it)**

* descr: verifies if the iterator is valid
* pre: it ∈ I
* post: valid ← True, if it points to a valid element from the list False, otherwise

**2.Representation**

Collision resolution by separate chaining

**SortedList:**

e: ↑Node[]

m: Integer

h: TFunction

r: Relation

**Node:**

k: TElem

next: ↑Node

**Iterator:**

l: SortedList

current: Integer

next: ↑Node

**3. Problem statement**

The Lego Toys company decided to be more organized and it decided to create reports each year based on the quantity of Lego Toys they sell every day.

This year the computer broke down and giving only a list of quantities, you will have to find out the quantity of toys sold every day sorted ascending, the best and worst days of productivity, and the number of overall sold toys.

I chose this problem because it can be easily solved using ADT sortedList. This container can easily manage the data because we need the report for the best and worst days of this year and also the report in increasing order.

**4. Representation according to the problem**

**SortedList:**

e: HashTable

r: Relation

**Node:**

k: TElem

next: ↑Node

**Iterator:**

l: SortedList

current: Integer

next: ↑Node

**HashTable:**

**e:** LinkedList[]

h: TFunction

m: Integer

**LinkedList:**

e: ↑Node

h: ↑head

l: size

**5.Implementation of the operations of the ADT**

**Subalgorithm init(sl,r) is:**

R <- relation

**-The complexity is Ɵ(1)**

**Subalgorithm add(sl,w) is:**

Ht.add(w)

**-The complexity is Ɵ(1)**

**Subalgorithm remove(sl,p) is:**

It: IteratorSL

While it.valid execute

P = p -1

If p = 0

Ht.remove([it].info)

It.next

**BC: Ɵ(1)**

**AC: Ɵ(m)**

**WC: Ɵ(m)**

**Function search(sl,w) is:**

search <- ht.search(w)

**BC: Ɵ(1)**

**AC: Ɵ(m)**

**WC: Ɵ(m)**

**Function first(sl) is:**

It: iteratorSL

First <- it.getInfo

**-The complexity is Ɵ(1)**

**Function last(sl) is:**

It: iteratorSL

Info: integer

While it.valid execute

Info <- it.getinfo

It.next

last <- it.getInfo

-**The complexity is Ɵ(m)**

**Function valid(sl, p) is:**

It: iteratorSL

Info: integer

While it.valid execute

P = p – 1

If p == 0

valid <- TRUE

It.next

valid <- FALSE

-**The complexity is Ɵ(p)**

**Function isEmpty(sl) is:**

It: iteratorSL

If it.valid execute

isEmpty <- FALSE

isEmpty <- TRUE

-**The complexity is Ɵ(1)**

**Function getElement(sl, p) is:**

It: iteratorSL

While it.valid execute

P = p – 1

If p == 0

getElement <- It.getInfo

It.next

GetElement <- -1

-**The complexity is Ɵ(p)**

**Function position(sl, x) is:**

It: iteratorSL

While it.valid execute

P = p + 1

If x == it.getInfo

position <- p

It.next

position <- -1

**BC: Ɵ(1)**

**AC: Ɵ(m)**

**WC: Ɵ(m)**

**Function size(sl) is:**

It: iteratorSL

P: Integer

While it.valid execute

P = p + 1

It.next

size <- p

-**The complexity is Ɵ(m)**

**6.Implementation of the operations of the Iterator**

**Subalgoritm next(it) is:**

Current <- current.nextNode

-The complexity is Ɵ(1)

**Function getInfo(it) is:**

GetInfo <- current.info

**-The complexity is Ɵ(1)**

**Subalgoritm Init(it, sl):**

Arr[]: Integer[]

Pos: integer

It: iterator

While it.valid execute

Itsll: iteratorSLL

While itsll.valid execute

Pos = pos + 1

Arr[pos] = itsll.getInfo

Itsll.next

It.next

Sorted(arr)

For I <- 1, pos

If head == NIL execute

NewNode <- Node

NewNode.info <- arr[i]

NewNode->next <- NIL

Head <- newNode

Else

Current = head

While current.nextNode != NIL execute

Current = current.nextNode

Nw <- Node

Nw.info <- arr[i]

Nw.nextNode <- current.nextNode

Current.nextNode <- nw

Current <- head

**-The complexity is Ɵ(m^2)**

**Function valid(it) is:**

If current != NIL execute

Valid <- True

Else

Valid <- False

-**The complexity is Ɵ(1)**

**7.Tests for the ADT implementation**

void \_testSL()

{

cout << "TEST SL\n";

SortedList sl;

assert(sl.isEmpty());

for (int i = 10; i <= 50; i++)

sl.add(i);

assert(!sl.isEmpty());

assert(sl.first() == 10);

assert(sl.last() == 50);

assert(sl.size() == 41);

assert(!sl.search(51));

sl.remove(4);

assert(!sl.search(4));

assert(sl.getElement(3) == 12);

assert(sl.getElement(90) == -1);

assert(sl.position(88) == -1);

assert(sl.position(10) == 1);

assert(sl.valid(10));

assert(!sl.valid(80));

}

void \_testSLL()

{

cout << "SLL TESTS \n";

SinglyLinkedList ls{};

IteratorSLL it{ ls };

int a1 = 0;

int a2 = 2;

int a3 = 4;

it.getFirst();

assert(ls.getSize() == 0);

ls.add(a1);

assert(ls.getSize() == 1);

ls.add(a2);

assert(ls.getSize() == 2);

ls.add(a3);

assert(ls.getSize() == 3);

assert(ls.search(a1));

int a5 = 7;

assert(!(ls.search(a5)));

ls.remove(a1);

assert(ls.getSize() == 2);

ls.remove(a5);

assert(ls.getSize() == 2);

ls.remove(a3);

assert(ls.getSize() == 1);

ls.printAll();

ls.remove(a2);

assert(ls.getSize() == 0);

}

**7.Tests for the ADT implementation**

void \_testHT()

{

cout << "HASH TABLE TESTS \n";

bool seen[115]{ false };

HashTable ht{ 10 };

for (int i = 1; i <= 10; i++)

{

ht.add(i\*i + i);

seen[i\*i + i] = true;

}

int iterations = 0;

IteratorHT it{ ht };

while (it.valid())

{

IteratorSLL itSll{ \*it.getCurrent() };

while(itSll.valid())

{

seen[itSll.getInfo()] = false;

itSll.next();

iterations++;

}

it.next();

}

for (int i = 1; i <= 110; i++)

assert(!seen[i]);

assert(iterations == 10);

}

**8.Solution of the problem**

Subalgorithm Solve() is:

@Print “Problem solving”

Sl: SortedList

For I<-1, 30 execute

Sl.add(rand(1,100))

@Print “Raport/Day increasingly”

It:iteratorSL

Sum: integer

While it.valid execute

Print it.getInfo

Sum = sum + it.getInfo

It.next

@Print “The total:” + sum

@Print “Best and worst days:”

@Print sl.last + “/” + sl.first

Complexities:

Add to ADT:**Ɵ(1)**

Raport/Day increasingly: **Ɵ(m^2) (where m is the length of the list)**

The total sum: **Ɵ(m) (where m is the length of the list)**

Best day: **Ɵ(1)**

Worst day: **Ɵ(1)**

*Solve function complexity*:**Ɵ(m^2) (where m is the length of the list)**