Documentation

Student: Paşcu Maria Florina Aura

Assigned project

27. ADT SparseMatrix – representation using triples (value ≠ 0) with implementation on a binary search tree.

Problem statement:

A manager of a deposit of car tires on the verge of bankruptcy needs an application to manage it. The deposit is a rectangular space and every square meter contains a shelf with a number of tires, but the deposit is mostly empty. A shelf is found by the position in the deposit, that is the line and the column where it is based on the upper left corner of the deposit. The application needs to manage every delivery and sale of the goods.

A sparse matrix is a reasonable ADT to use for this problem, because the data is structured on lines and columns, and also there are a lot of values that are 0 and don't need to be stored.

ADT SparseMatrix specification and interface:

A Sparse Matrix is a container that represents a two-dimensional array that contains many values of 0. In this case we save only the elements that are different from 0. An element can be accessed by its position in the matrix, that is its line and column, which are unique for every element.

Domain: SM = {sm | sm is a sparse matrix with elements e = (line, column, value) where line, column - integers and value ∈ TElem}

- init(matrix, nrL, nrC):
  - Description: creates a new matrix with nrL lines and nrC columns
  - Pre: nrL, nrC - integers
  - Post: matrix ∈ SM, sm is a sparse matrix with nrL lines and nrC columns with all its elements 0
- nrLines(matrix, nrL):
  - Description: returns the number of lines of the matrix
  - Pre: matrix ∈ SM
  - Post: nrL - integer, nrL is the number of lines of the matrix
- nrColumns(matrix, nrC):
  - Description: returns the number of columns of the matrix

Pre: matrix ∈ SM
Post: nrC - integer, nrC is the number of columns of the matrix

- element(matrix, i, j, e):
  Description: returns the element from the line i and column j
  Pre: matrix ∈ SM, i, j - integers, valid positions in the matrix
  Post: e ∈ TElem, e is the element from the line i and column j of the matrix
  @throw exception if position is invalid
- modify(matrix, i, j, e):
  Description: changes the value of the element from line i and column j with e
  Pre: matrix ∈ SM, i, j - integers, valid positions in the matrix, e ∈ TElem
  Post: m' ∈ SM, where m' is the same as m but with the element from line i and column j updated to e
  @throw exception if position is invalid
- iterator(matrix, it):
  Description: returns an iterator for the matrix
  Pre: matrix ∈ SM
  Post: it ∈ It, it is an iterator for the matrix

Interface for iterator:

- init(it, matrix):
  Description: creates an iterator for a sparse matrix
  Pre: matrix ∈ SM
  Post: it ∈ It, it is an iterator
- valid(it):
  Description: returns true if the current element is valid, returns false otherwise
  Pre: it ∈ It, it is an iterator for a sparse matrix
  Post: true - if the current element is valid, false otherwise
- getCurrent(it, c):
  Description: returns the current element
  Pre: it ∈ It, it is an iterator for a sparse matrix
  Post: c - Cell, c.line and c.column represent the current position of c.e
- next(it):
  Description: goes to the next element in the iterator
  Pre: it ∈ It, valid(it), it is an iterator for a sparse matrix
  Post: it' ∈ It, the current element from it' is the next element from it

Interface for BinarySearchTree:

- elementBST(root, i, j, e)
  Description: returns the node that has i, j as position in the matrix
  Pre: root - ↑BSTNode, line, column - integers, valid positions

Post: e - ↑BSTNode, e is the node which has the line i and column j

- addBST(root, i, j, e)
  Description: adds a new node that has as info line i, column j and value e
  Pre: root - ↑BSTNode, i, j - integers, valid positions, e ∈ TElem
  Post: root' - ↑BSTNode, root' is the same tree, but with the new element added
- deleteBST(root, i, j)
  Description: deletes the node that has as info line i and column j
  Pre: root - ↑BSTNode, i, j - integers, valid positions
  Post: root' - ↑BSTNode, root' is the same tree, but without the deleted node
- modifyBST(root, i, j, e)
  Description: updates the node that has as info line i, column j with the value e
  Pre: root - ↑BSTNode, i, j - integers, valid positions, e ∈ TElem
  Post: root' - ↑BSTNode, root' is the same tree, but with the updated node
- findMin(root, e)
  Description: finds the node with the minimum position from the subtree starting from root
  Pre: root - ↑BSTNode
  Post: e - ↑BSTNode, e has the minimum position from the subtree root

ADT SparseMatrix representation:

Representation on Binary Search Tree

- Cell:
  line: integer
  column: integer
  Info: TElem
- BSTNode**:**
  line: integer
  column: integer
  info: TElem
  left: ↑BSTNode
  right: ↑BSTNode
- BinarySearchTree:
  Root: ↑BSTNode
- SparseMatrix:
  bst: BinarySearchTree
  nrL: integer
  nrC: integer
- IteratorSM:

sm: matrix
current: ↑BTSNode
s: stack

Implementation of ADT:

- subalgorithm init(matrix, nrL, nrC):

  @init matrix
  matrix.nrL <- nrL
  matrix.nrC <- nrC
  matrix.bst.root <- NIL

  Complexity: θ(1)
- subalgorithm nrLines(matrix, nrL)

  nrL <- matrix.nrL

  Complexity: θ(1)
- subalgorithm nrColumns(matrix, nrC)

  nrC <- matrix.nrC

  Complexity: θ(1)
- subalgorithm element(matrix, i, j, e)

  @throw exception if i,j invalid
  elementBTS(matrix.bts.root, i, j, e)

  Complexity: as elementBST
- subalgorithm elementBST(root, i, j, e)

```
if root = NIL then
    e <- 0
else if i = [root].line and j = [root].column then
    e <- [root].info
else if i < [root].line or (i = [root].line and j < [root].column) then
    elementBST([root].left, i, j, e)
else
    elementBST([root].right, i, j, e)
```

  Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h) where n is the number of elements in the tree and h is the height of the tree
- subalgorithm modify(matrix, i, j, e)

```
@throw exception if i,j invalid
element(matrix, i, j, elem)
if e = 0 and elem = 0 then
    @do nothing
else if elem = 0 then
    addBST(root, i, j, e)
```

```
        else if e = 0 then
                deleteBST(root, i, j)
        else
                modifyBST(root, i, j, e)
```
Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

- subalgorithm addBST(root, i, j, e)
```
        if root = NIL then
                allocate(root)
                [root].line <- i
                [root].column <- j
                [root].info <- e
                [root].left <- NIL
                [root].right <- NIL
        else if i < [root].line or (i = [root].line and j < [root].column) then
                addBST([root].left, i, j , e)
        else
                addBST([root].right, i, j , e)
```
Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

- subalgorithm deleteBST(root, i, j)
```
        if i < [root].line or (i = [root].line and j < [root].column) then
                delete([root].left, i, j)
        else if i > [root].line or (i = [root].line and j > [root].column) then
                delete([root.right, i, j)
        else
                if [root].left = NIL and [root].right = NIL then
                        root<-NIL
                else if [root].left = NIL then
                        root<-[root].right
                else if [root].right = NIL then
                        root<-[root].left
                else
                        findMin([root].right, temp)
                        [root].line <- [temp].line
                        [root].column <- [temp].column
                        [root].info <- [temp].info
                        deleteBST([root].right, [temp].line, [temp].column)
```
Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

- subalgorithm findMin(root, e)
```
        if [root].left = NIL then
                e <- root
        findMin([root].left, e)
```
Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

- subalgorithm modifyBST(root, i, j, e)

  if i = [root].line and j = [root].column then

  [root].ifo <- e

  else if i < [root].line or (i = [root].line and j < [root].column) then

  modify([root].left, i, j, e)

  else

  modify([root].right, i, j, e)

  Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)
- subalgorithm iterator(matrix, it):

  init(it, matrix)

Implementation for Iterator

- subalgorithm init(it, matrix)

  @init it

  it.sm <- matrix

  @init it.s

  node <- it.sm.bst.root

  while node != NIL execute

  push(it.s, node)

  node <- [node].left

  if !empty(s) then

  it.current <- top(it.s)

  else

  it.current <- NIL

  Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

  Computation : best case occurs when we have only one node, which is the root, or the root does not have a left child; worst case happens when the tree is degenerate and every node has only a left child - in this case every node is covered; average case happens when the root has a left child and there also exists right children

- subalgorithm getCurrent(it, c)

  c.line <- [it.current].line

  c.column <- [it.current].column

  c.info <- [it.current].info

  Complexity: θ(1)
- subalgorithm valid(it)

  if it.current = NIL then

  valid <- true

  else

  valid <- false

  Complexity: θ(1)

- subalgorithm next(it)

```
node <- pop(it.s)
if [node].right != NIL then
    node <- [node].right
    while node != NIL execute
        push(it.s, node)
        node <- [node].left
if !empty(it.s) then
    it.current <- top(it.s)
else
    it.current <- NIL
```

Complexity: Best case: θ(1); Worst case: O(n); Average case: O(h)

Tests

```
SparseMatrix s{ 10, 10 };
assert(s.nrLines() == 10);
assert(s.nrColumns() == 10);
assert(s.getBTS().root == NULL);
s.modify(5, 3, 10);
s.modify(2, 7, 6);
s.modify(7, 5, 9);
s.modify(7, 1, 2);
s.modify(4, 8, 5);
s.modify(10, 3, 1);
s.modify(3, 7, 9);
s.modify(2, 4, 1);
s.modify(2, 5, 2);
assert(s.element(5, 3) == 10);
s.modify(5, 3, 0);
assert(s.element(5, 3) == zero);
s.modify(2, 7, 3);
assert(s.element(2, 7) == 3);
s.modify(10, 3, 0);
assert(s.element(10, 3) == zero);
s.modify(7, 1, 0);
assert(s.element(7, 1) == zero);
s.modify(4, 8, 0);
assert(s.element(4, 8) == zero);

IteratorSM it = s.iteratorSM();
assert(it.getCurrent().line == 2);
assert(it.getCurrent().column == 4);
```

```
assert(it.getCurrent().info == 1);
assert(it.valid() == true);
it.next();
assert(it.getCurrent().line == 2);
assert(it.getCurrent().column == 5);


SparseMatrix s1{ 5, 5 };
IteratorSM it1 = s1.iteratorSM();
assert(it1.valid() == false);

SparseMatrix s2{ 3, 3 };
s2.modify(1, 1, 3);
s2.modify(1, 3, 2);
s2.modify(1, 1, 0);
assert(s2.getBTS().root->column == 3);
```

Problem solution

```
init(matrix, 10, 10)
initialize(matrix)
while true execute
    @print menu
    @read command
    if command = 1 then
        nrLines(matrix, nrL)
        @print nrL
        //Complexity: θ(1)
    else if command = 2 then
        nrColumns(matrix, nrC)
        @print nrC
        //Complexity: θ(1)
    else if command = 3 then
        @read line, column
        element(matrix, line, column, e)
        @print e
        //Complexity: Best case: θ(1); Worst case: O(n); Average case:
        //O(h)
    else if command = 4 then
        @read line, column, e
        modify(matrix, line, column, e)
```

```
            //Complexity: Best case: θ(1); Worst case: O(n); Average case:
            //O(h)
        else if command  = 5 then
            iterator(matrix, it)
            while it.valid execute
                getCurrent(it, c)
                @print c.line, c.column, c.info
                next(it)
                //Complexity: O(n * h)
        else if command = 0 then
            break
```

- initialize(matrix)

```
        modify(matrix, 5, 3, 10);
        modify(matrix, 2, 7, 6);
        modify(matrix, 7, 5, 9);
        modify(matrix, 7, 1, 2);
        modify(matrix, 4, 8, 5);
        modify(matrix, 10, 3, 1);
        modify(matrix, 3, 7, 9);
        modify(matrix, 2, 4, 1);
        modify(matrix, 2, 5, 2);
```

Menu:

1. Print the number of lines.
2. Print the number of columns.
3. Print the value from a given position.
4. Modify the value from a given position.
5. Print the deposit.