МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Технология разработки программного обеспечения

Лабораторная работа №1

**Разработка класса структуры данных**

Факультет: АВТ

Группа: АММ - 17

Студенты: Кузьмин Д.С.

Сизов Л.С.

Проверил: Романов Е.Л.

Новосибирск, 2017

# 1 Цель работы

Реализовать на ЯП Scala структуру данных.

# 2 Исходные данные

Структура данных по варианту 7: Циклический список

# 3 Описание работы

Методы и классы реализованной структуры

by.xotonic.collections.CyclicList – класс циклического списка

by.xotonic.collections.CyclicList#head – поле-указатель на первый элемент в коллекции

by.xotonic.collections.CyclicList#sort – сортировка списка

by.xotonic.collections.CyclicList#toString – преобразовать список в строку

by.xotonic.collections.CyclicList#prepend – добавить элемент в начало списка

by.xotonic.collections.CyclicList#add – добавить элемент в конец списка

by.xotonic.collections.CyclicList#apply – получить значение элемента по порядковому номеру в списке

by.xotonic.collections.CyclicList#nonEmpty – факт непустоты листа

by.xotonic.collections.CyclicList#isEmpty – факт пустоты листа

by.xotonic.collections.CyclicList#remove – удалить элемент по порядковому номеру

by.xotonic.collections.CyclicList#foreach – проитерировать по элементам списка

by.xotonic.collections.CyclicList#foreachValue – проитерировать по значениям списка

by.xotonic.collections.CyclicList#CyclicList – конструктор по умолчанию

# 4 Заключение

Была реализована коллекция на ЯП Scala. Некоторый функционал выполнен в функциональном стиле (сортировка, обход элементов). Код реализации проходит приложенные в листинге модульные тесты.

# 5 Листинг программы

package by.xotonic.collections

import com.typesafe.scalalogging.StrictLogging

import org.scalactic.Snapshots

import scala.annotation.tailrec

/\*\*

\* Cyclic list

\*/

class CyclicList[T]() extends StrictLogging with Snapshots {

private var head: Option[Node[T]] = Option.empty

def sort(cmp : (T,T) => Int) : CyclicList[T] = {

logger.debug("Sorting")

var c = new CyclicList[T]

if (isEmpty) {

logger.debug("Collection is empty")

return c

}

if (head.get.prev == head.get) {

logger.debug(s"Collection has only head: ${head.get}")

c.add(head.get.data)

return c

}

val x = apply(0).get

var c2 = new CyclicList[T]

var c3 = new CyclicList[T]

foreachValue(i => {

val y = cmp(i, x)

if (y < 0) c.add(i)

else if (y == 0) c2.add(i)

else c3.add(i)

})

c = c.sort(cmp)

c2 = c2.sort(cmp)

c3 = c3.sort(cmp)

c2.foreachValue(i => c.add(i))

c3.foreachValue(i => c.add(i))

logger.debug(s"Sort step: ${snap(c, c2, c3, x, head)}")

c

}

override def toString: String = {

val sb = new StringBuilder

sb.append("[")

foreachValue(v => sb.append(v).append(","))

sb.append("]")

sb.toString

}

def prepend(x: T, pos: Int = 0) = head match {

case None =>

logger.debug(s"Adding $x to head")

val reallyFirst = new Node[T](x)

head = Some(reallyFirst)

logger.debug(head.toString)

case Some(theHead) =>

walkToPosition((prev, node) => {

logger.debug(s"Adding $x after $node")

val newNode = new Node[T](x)

newNode.next = node.next

node.next = newNode

logger.debug(s" new node: $newNode, changed node: $node, head: $head")

}, pos, theHead)

}

def add(x: T) = head match {

case None => logger.debug(s"Set $x as head"); head = Some(new Node[T](x))

case Some(theHead) =>

val newNode = new Node[T](x)

newNode.prev = theHead.prev

theHead.prev.next = newNode

newNode.next = theHead

theHead.prev = newNode

if (theHead.next == theHead) {

theHead.next = newNode

}

logger.debug(s"new=$newNode head=$theHead")

}

/\*\*

\* Get element by position relative the head

\* @param pos position

\* @return

\*/

def apply(pos: Int): Option[T] = head flatMap {

firstNode => {

Some(walkToPosition((\_, node) => node.data, pos, firstNode))

}

}

/\*\*

\* Checks if cyclic list is not empty

\*

\* @return non-emptiness flag

\*/

def nonEmpty = !isEmpty

/\*\*

\* Checks if cyclic list is empty

\*

\* @return emptiness flag

\*/

def isEmpty = head.isEmpty

/\*\*

\* Removes element with index x from cyclic list

\* If x is bigger than list size, then the index of item to remove is calcul ated as

\* <code>length % x</code> - like the list is infinite

\*

\* @param x index of element to remove

\*/

def remove(x: Int) = head.foreach(theHead => {

if (x == 0) {

theHead.prev.next = theHead.next

head = Some(theHead.next)

}

else

walkToPosition((prev, node) => {

logger.debug(s"Removing ${node.data} : ${prev.data}.next = ${node.next.data}.next")

prev.next = node.next

node.prev = prev.prev

}, x, theHead)

})

@tailrec final def walkToPosition[R](fun: (Node[T], Node[T]) => R,

pos: Int,

node: Node[T] = head.get,

prev: Node[T] = head.get,

cur: Int = 0): R = {

// logger.debug(s"Walk step: ${snap(pos, node, prev, cur)}")

if (cur == pos)

fun(prev, node)

else

walkToPosition(fun, pos, node.next, node, cur + 1)

}

final def foreach(fun: Node[T] => Unit) = {

@tailrec def it(node: Node[T], stopNode: Node[T], start: Boolean): Unit = {

// logger.debug(s"Step: ${snap(node, stopNode, start)}")

if (node != stopNode || start) {

fun(node)

it(node.next, stopNode, start = false)

}

}

head.foreach(h => it(h, h, start = true))

}

final def foreachValue(fun: T => Unit) = {

@tailrec def it(node: Node[T], stopNode: Node[T], start: Boolean): Unit = {

// logger.debug(s"Step: ${snap(node, stopNode, start)}")

if (node != stopNode || start) {

fun(node.data)

it(node.next, stopNode, start = false)

}

}

head.foreach(h => it(h, h, start = true))

}

}

package by.xotonic.collections

/\*\*

\* Created by xoton on 07.10.2017.

\*/

class Node[T](initData: T)

{

var next : Node[T] = this

var prev : Node[T] = this

val data = initData

override def toString = s"[$$${prev.data} => $data => $$${next.data}]"

}

package by.xotonic

import by.xotonic.collections.{CyclicList, Node}

import com.typesafe.scalalogging.StrictLogging

import org.scalatest.{FlatSpec, Matchers}

import scala.collection.mutable.ListBuffer

class CyclicListTest extends FlatSpec with Matchers with StrictLogging {

def print[T](c : CyclicList[T]) =

{

val list = new ListBuffer[Node[T]]

c.foreach(data => {

list += data

})

logger.debug(list.toString)

}

"A CyclicList" should "be empty when called with default constructor" in {

val list: CyclicList[Int] = new CyclicList[Int]()

list.isEmpty should be(true)

}

"Add and Get " should "work" in {

val c = new CyclicList[String]

c.add("1")

c.add("2")

c.add("3")

c.add("4")

c.add("5")

c.add("6")

c.add("7")

c.add("8")

logger.debug("List fulfilled")

print(c)

c(0).get should be("1")

c(1).get should be("2")

c(2).get should be("3")

c(3).get should be("4")

c(4).get should be("5")

c(5).get should be("6")

c(6).get should be("7")

c(7).get should be("8")

}

"The Get " should " return repeating nodes if index > size" in {

val c = new CyclicList[String]

c.add("first")

c.add("second")

c(2).get should be("first")

c(3).get should be("second")

}

"Remove " should "work" in {

val c = new CyclicList[String]

c.add("0")

c.add("1")

c.add("2")

c.add("3")

c.remove(2)

c(0).get should be("0")

c(1).get should be("1")

// c(2).get should be("2")

c(2).get should be("3")

}

"You " can "remove element in 0 position" in {

val c = new CyclicList[String]

c.add("0")

c.add("1")

c.add("2")

c.add("3")

c.remove(0)

c(0).get should be("1")

c(1).get should be("2")

c(2).get should be("3")

}

"Sort " should "work" in

{

logger.debug("Start sort test")

val c = new CyclicList[Int]

c.add(1)

c.add(4)

c.add(3)

c.add(5)

c.add(2)

c.add(6)

val r = c.sort((x,y) => { x - y})

logger.debug(r.toString)

r(0).get should be (1)

r(1).get should be (2)

r(2).get should be (3)

r(3).get should be (4)

r(4).get should be (5)

r(5).get should be (6)

}

}