**package** uk.ac.glasgow.etparser.handlers;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.Set;

**import** org.jfree.chart.ChartFactory;

**import** org.jfree.chart.ChartPanel;

**import** org.jfree.chart.JFreeChart;

**import** org.jfree.chart.plot.PlotOrientation;

**import** org.jfree.data.xy.XYSeries;

**import** org.jfree.data.xy.XYSeriesCollection;

**import** org.jfree.ui.ApplicationFrame;

**import** uk.ac.glasgow.etparser.ObjectClass;

**import** uk.ac.glasgow.etparser.ObjectLiveTime;

**import** uk.ac.glasgow.etparser.CommandParser.WayToDealWithErrors;

**import** uk.ac.glasgow.etparser.events.CreationEvent;

**import** uk.ac.glasgow.etparser.events.Event;

**import** uk.ac.glasgow.etparser.events.Event.Check;

**import** uk.ac.glasgow.etparser.handlers.Heap.LiveSizeChart;

**public** **class** ETHeap **extends** Heap {

/\*\*

\* This is analogous to the real heap memory.

\*/

**protected** **static** Heap *theHeap* = **null**;

/\*\*

\* Measures time sequentially (1, 2, 3...).

\*/

**protected** **int** timeSequence;

/\*\*

\* Measures time as function of the allocated objects' sizes.

\*/

**protected** **int** timeSize;

/\*\*

\* Measures time in terms of method entry and method exit.

\*/

**protected** **int** timeMethod;

/\*\*

\* A hash map that keeps the object id and the last event that happened to

\* that object.

\*/

**protected** HashMap<String, ObjectClass> objectStates;

**protected** HashMap<String, ObjectLiveTime> everSeen;

/\*\*

\* A set of all the objects tried to be accessed ever including the once

\* that were not born.

\*/

**protected** Set<String> processedObjects;

**protected** **int** livesize, allocatedMemSize;

**protected** LiveSizeChart chart;

**protected** WayToDealWithErrors dealWithPreaccess, dealWithPostAccess;

/\*\*

\* Initializes the class variables. Private because of Singleton design

\* pattern.

\*/

**protected** Heap() {

chart=**new** LiveSizeChart();

livesize = 0;

allocatedMemSize = 0;

timeSequence = 0;

timeSize = 0;

timeMethod = 0;

objectStates = **new** HashMap<String, ObjectClass>();

processedObjects = **new** HashSet<String>();

everSeen = **new** HashMap<String, ObjectLiveTime>();

}

**public** **void** setDealWithPreaccess(WayToDealWithErrors wayToDeal) {

dealWithPreaccess = wayToDeal;

}

**public** **void** setDealWithPostaccess(WayToDealWithErrors wayToDeal) {

dealWithPostAccess = wayToDeal;

}

**public** **void** createChart(){

chart.setVisible(**true**);

}

/\*\*

\*

\* **@return** an instance of the heap. It is global.

\*/

**public** **static** Heap getTheHeap() {

**if** (*theHeap* == **null**) {

*theHeap* = **new** Heap();

}

**return** *theHeap*;

}

/\*\*

\* Checks whether the event is legal and if it is, it updates the event

\* record in the heap and updates the time.

\*

\* **@param** e

\* instance of the Event super class

\*/

@Override

**public** **void** handle(Event e) {

timeSequence++;

String currentObjectID = e.getObjectID();

String currentEventType = e.getTypeOfEvent();

// if never seen before

**if** (!existsInEverSeen(currentObjectID)) {

// create a new livetime for this object

ObjectLiveTime livetime = **new** ObjectLiveTime(currentObjectID);

// if the event is allocation- great

**if** (currentEventType.equalsIgnoreCase("a")) {

livetime.giveBirth();

allocateObject((CreationEvent) e);

chart.updateChart(timeSequence,livesize);

System.*out*.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation

// report for notborn error

**else** {

System.*out*.println("not born because not allocated "

+ currentObjectID);

e.setCheck(Check.*NOTBORN*);

}

everSeen.put(currentObjectID, livetime);

}

// the object has been seen before

**else** {

// if it wasn't allocated (check whether it has been deallocated

**if** (!existsInHeap(currentObjectID)

&& !everSeen.get(currentObjectID).isDead()) {

// if the event is allocation- perfect

**if** (currentEventType.equalsIgnoreCase("a")) {

everSeen.get(currentObjectID).giveBirth();

allocateObject((CreationEvent) e);

chart.updateChart(timeSequence,livesize);

System.*out*

.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation report not born error

**else** {

System.*out*.println("not born because not allocated2 "

+ currentObjectID);

e.setCheck(Check.*NOTBORN*);

}

}

// it has been allocated

**else** {

// take the object and check it's livetime

ObjectLiveTime currentObjectLivetime = everSeen

.get(currentObjectID);

// if it was never born, probably preaccess before and now again

// or probably dead

**if** (!currentObjectLivetime.isBorn()) {

System.*out*

.println("not live but exists- either preaccess or it was dead "

+ currentObjectID);

e.setCheck(Check.*NOTBORN*);

}

// the object died before this access

**else** **if** (currentObjectLivetime.isDead()) {

e.setCheck(Check.*DEAD*);

System.*out*.println("dead " + currentObjectID);

}

// it's legal to update this object

**else** {

// if the event isn't dead just make the update

e.setCheck(Check.*LEGAL*);

// if it was a death event kill the object in everLived,

// deallocate it from the heap

**if** (currentEventType.equalsIgnoreCase("d")) {

killObject(currentObjectID);

deallocate(currentObjectID);

chart.updateChart(timeSequence,livesize);

System.*out*.println("kill " + currentObjectID);

} **else** {

updateObject(e);

System.*out*.println("legal access " + currentObjectID);

}

}

}

}

}

**protected** **void** deallocate(String objectID) {

ObjectClass remove = objectStates.remove(objectID);

allocatedMemSize -= remove.getSize();

}

**protected** **void** killObject(String objectID) {

everSeen.get(objectID).kill();

livesize -= objectStates.get(objectID).getSize();

}

**protected** **boolean** existsInHeap(String objectID) {

**return** objectStates.get(objectID) != **null**;

}

**protected** **boolean** existsInEverSeen(String objectID) {

**return** everSeen.get(objectID) != **null**;

}

**protected** **void** allocateObject(CreationEvent e) {

String currentObjectID = e.getObjectID();

**int** size = e.getSize();

livesize += size;

allocatedMemSize += size;

ObjectClass object = **new** ObjectClass(size, timeSequence,

currentObjectID);

// ObjectEventRecord record = new ObjectEventRecord(e);

objectStates.put(currentObjectID, object);

e.setCheck(Check.*CREATION*);

CreationEvent ce = (CreationEvent) e;

timeSize += ce.getSize();

}

**protected** **void** allocateObjectCheater(Event e) {

String currentObjectID = e.getObjectID();

ObjectClass object = **new** ObjectClass(0, timeSequence, currentObjectID);

// ObjectEventRecord record = new ObjectEventRecord(e);

objectStates.put(currentObjectID, object);

e.setCheck(Check.*CREATION*);

CreationEvent ce = **new** CreationEvent(e);

timeSize += ce.getSize();

}

**protected** **void** updateObject(Event e) {

String currentObjectID = e.getObjectID();

ObjectClass accessedObject = objectStates.get(currentObjectID);

**if** (accessedObject != **null**) {

String currentEventStatus = e.getTypeOfEvent();

accessedObject.updateEvent(timeSequence, e.getTypeOfEvent());

objectStates.put(currentObjectID, accessedObject);

e.setCheck(Check.*LEGAL*);

**if** (currentEventStatus.equalsIgnoreCase("M")

|| currentEventStatus.equalsIgnoreCase("E")) {

timeMethod++;

}

}

}

**protected** **void** decisionMakerPreaccess(Event e) {

**switch** (dealWithPreaccess) {

**case** *MOVE*:

handleAllocateAtFirstAccess(e);

**break**;

**default**:

handleIgnorePreaccess();

}

}

**protected** **void** decisionMakerPostaccess() {

**switch** (dealWithPostAccess) {

**case** *MOVE*:

handleMoveDeath();

**break**;

**default**:

handleIgnorePostaccess();

}

}

/\*\*

\*

\* **@return** the number of objects tried to be accessed ever.

\*/

**public** **int** getNumObjects() {

System.*out*.println("number in heap " + everSeen.size());

**return** everSeen.size();

}

/\*\*

\*

\* **@param** oid

\* the id of the object we want to access in the heap.

\* **@return** the last event record for the given object.

\*/

**public** ObjectClass getRecord(String oid) {

**return** objectStates.get(oid);

}

/\*\*

\*

\* **@return** the current time expressed as sequence.

\*/

**public** **int** getTimeSequence() {

**return** timeSequence;

}

/\*\*

\*

\* **@return** the current time expressed as allocated object size.

\*/

**public** **int** getTimeSize() {

**return** timeSize;

}

/\*\*

\*

\* **@return** the current time expressed in term of method entry and exit.

\*/

**public** **int** getTimeMethod() {

**return** timeMethod;

}

**protected** **void** handleIgnorePreaccess() {

System.*out*.println("ignore preaccess");

// as this method just ignores preaccess it doesn't do anything

}

**protected** **void** handleAllocateAtFirstAccess(Event e) {

System.*out*.println("allocate at first access");

allocateObjectCheater(e);

updateObject(e);

}

**protected** **void** handleIgnorePostaccess() {

System.*out*.println("ignore postaccess");

// as this method just ignores postaccess it doesn't do anything

}

// no need for this in my opinion. we don't need an object to be dead to

// collect it

**protected** **void** handleMoveDeath() {

System.*out*.println("move death");

}

**public** **void** removeRecord(String objectID) {

objectStates.remove(objectID);

}

**public** HashMap<String, ObjectClass> getObjectStates() {

**return** objectStates;

}

**public** **class** LiveSizeChart **extends** ApplicationFrame {

/\*\*

\*

\*/

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** XYSeries data;

**public** LiveSizeChart() {

**super**("Live size chart");

data = **new** XYSeries("Data");

data.add(0, 0);

XYSeriesCollection dataset = **new** XYSeriesCollection(data);

// based on the dataset we create the chart

JFreeChart chart = ChartFactory.*createXYLineChart*(

"Livesize change over time", "Time", "Livesize", dataset,

PlotOrientation.*VERTICAL*, **false**, **false**, **false**);

// we put the chart into a panel

ChartPanel chartPanel = **new** ChartPanel(chart);

// default size

chartPanel.setPreferredSize(**new** java.awt.Dimension(800, 600));

// add it to our application

setContentPane(chartPanel);

pack();

}

**public** **void** updateChart(**double** x, **double** y) {

data.add(x, y);

}

}

}

package uk.ac.glasgow.etparser.handlers;

import java.util.HashMap;

import java.util.HashSet;

import java.util.Set;

import org.jfree.chart.ChartFactory;

import org.jfree.chart.ChartPanel;

import org.jfree.chart.JFreeChart;

import org.jfree.chart.plot.PlotOrientation;

import org.jfree.data.xy.XYSeries;

import org.jfree.data.xy.XYSeriesCollection;

import org.jfree.ui.ApplicationFrame;

import uk.ac.glasgow.etparser.CommandParser.Heuristic;

import uk.ac.glasgow.etparser.ObjectClass;

import uk.ac.glasgow.etparser.CommandParser.WayToDealWithErrors;

import uk.ac.glasgow.etparser.ObjectClass.EventType;

import uk.ac.glasgow.etparser.ObjectLiveTime;

import uk.ac.glasgow.etparser.events.CreationEvent;

import uk.ac.glasgow.etparser.events.Event;

import uk.ac.glasgow.etparser.events.Event.Check;

public class Heap implements EventHandler {

/\*\*

\* This is analogous to the real heap memory.

\*/

protected static Heap theHeap = null;

/\*\*

\* Measures time sequentially (1, 2, 3...).

\*/

protected int timeSequence;

/\*\*

\* Measures time as function of the allocated objects' sizes.

\*/

protected int timeSize;

/\*\*

\* Measures time in terms of method entry and method exit.

\*/

protected int timeMethod;

/\*\*

\* A hash map that keeps the object id and the last event that happened to

\* that object.

\*/

protected HashMap<String, ObjectClass> objectStates;

protected HashMap<String, ObjectLiveTime> everSeen;

/\*\*

\* A set of all the objects tried to be accessed ever including the once

\* that were not born.

\*/

protected Set<String> processedObjects;

protected int livesize, allocatedMemSize;

protected LiveSizeChart chart;

protected Heuristic heuristic;

protected WayToDealWithErrors dealWithPreaccess, dealWithPostAccess;

/\*\*

\* Initializes the class variables. Private because of Singleton design

\* pattern.

\*/

protected Heap() {

chart=new LiveSizeChart();

livesize = 0;

allocatedMemSize = 0;

timeSequence = 0;

timeSize = 0;

timeMethod = 0;

objectStates = new HashMap<String, ObjectClass>();

processedObjects = new HashSet<String>();

everSeen = new HashMap<String, ObjectLiveTime>();

}

public void setDealWithPreaccess(WayToDealWithErrors wayToDeal) {

dealWithPreaccess = wayToDeal;

}

public void setDealWithPostaccess(WayToDealWithErrors wayToDeal) {

dealWithPostAccess = wayToDeal;

}

public void createChart(){

chart.setVisible(true);

}

/\*\*

\*

\* @return an instance of the heap. It is global.

\*/

public static Heap getTheHeap() {

if (theHeap == null) {

theHeap = new Heap();

}

return theHeap;

}

/\*\*

\* Checks whether the event is legal and if it is, it updates the event

\* record in the heap and updates the time.

\*

\* @param e

\* instance of the Event super class

\*/

@Override

public void handle(Event e) {

timeSequence++;

String currentObjectID = e.getObjectID();

String currentEventType = e.getTypeOfEvent();

// if never seen before

if (!existsInEverSeen(currentObjectID)) {

// create a new livetime for this object

ObjectLiveTime livetime = new ObjectLiveTime(currentObjectID);

// if the event is allocation- great

if (currentEventType.equalsIgnoreCase("a")) {

livetime.giveBirth();

allocateObject((CreationEvent) e);

chart.updateChart(timeSequence,livesize);

System.out.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation

// report for notborn error

else {

System.out.println("not born because not allocated "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

everSeen.put(currentObjectID, livetime);

}

// the object has been seen before

else {

// if it wasn't allocated (check whether it has been deallocated

if (!existsInHeap(currentObjectID)

&& !everSeen.get(currentObjectID).isDead()) {

// if the event is allocation- perfect

if (currentEventType.equalsIgnoreCase("a")) {

everSeen.get(currentObjectID).giveBirth();

allocateObject((CreationEvent) e);

chart.updateChart(timeSequence,livesize);

System.out

.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation report not born error

else {

System.out.println("not born because not allocated2 "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

}

// it has been allocated

else {

// take the object and check it's livetime

ObjectLiveTime currentObjectLivetime = everSeen

.get(currentObjectID);

// if it was never born, probably preaccess before and now again

// or probably dead

if (!currentObjectLivetime.isBorn()) {

System.out

.println("not live but exists- either preaccess or it was dead "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

// the object died before this access

else if (currentObjectLivetime.isDead()) {

e.setCheck(Check.DEAD);

System.out.println("dead " + currentObjectID);

}

// it's legal to update this object

else {

// if the event isn't dead just make the update

e.setCheck(Check.LEGAL);

// if it was a death event kill the object in everLived,

// deallocate it from the heap

if (currentEventType.equalsIgnoreCase("d")) {

killObject(currentObjectID);

deallocate(currentObjectID);

chart.updateChart(timeSequence,livesize);

System.out.println("kill " + currentObjectID);

} else {

updateObject(e);

System.out.println("legal access " + currentObjectID);

}

}

}

}

}

protected void deallocate(String objectID) {

ObjectClass remove = objectStates.remove(objectID);

allocatedMemSize -= remove.getSize();

}

protected void killObject(String objectID) {

everSeen.get(objectID).kill();

livesize -= objectStates.get(objectID).getSize();

}

protected boolean existsInHeap(String objectID) {

return objectStates.get(objectID) != null;

}

protected boolean existsInEverSeen(String objectID) {

return everSeen.get(objectID) != null;

}

protected void allocateObject(CreationEvent e) {

String currentObjectID = e.getObjectID();

int size = e.getSize();

livesize += size;

allocatedMemSize += size;

ObjectClass object = new ObjectClass(size, timeSequence,

currentObjectID);

// ObjectEventRecord record = new ObjectEventRecord(e);

objectStates.put(currentObjectID, object);

e.setCheck(Check.CREATION);

CreationEvent ce = (CreationEvent) e;

timeSize += ce.getSize();

}

protected void allocateObjectCheater(Event e) {

String currentObjectID = e.getObjectID();

ObjectClass object = new ObjectClass(0, timeSequence, currentObjectID);

// ObjectEventRecord record = new ObjectEventRecord(e);

objectStates.put(currentObjectID, object);

e.setCheck(Check.CREATION);

CreationEvent ce = new CreationEvent(e);

timeSize += ce.getSize();

}

protected void updateObject(Event e) {

String currentObjectID = e.getObjectID();

ObjectClass accessedObject = objectStates.get(currentObjectID);

if (accessedObject != null) {

String currentEventStatus = e.getTypeOfEvent();

accessedObject.updateEvent(timeSequence, e.getTypeOfEvent());

objectStates.put(currentObjectID, accessedObject);

e.setCheck(Check.LEGAL);

if (currentEventStatus.equalsIgnoreCase("M")

|| currentEventStatus.equalsIgnoreCase("E")) {

timeMethod++;

}

}

}

protected void decisionMakerPreaccess(Event e) {

switch (dealWithPreaccess) {

case MOVE:

handleAllocateAtFirstAccess(e);

break;

default:

handleIgnorePreaccess();

}

}

protected void decisionMakerPostaccess() {

switch (dealWithPostAccess) {

case MOVE:

handleMoveDeath();

break;

default:

handleIgnorePostaccess();

}

}

/\*\*

\*

\* @return the number of objects tried to be accessed ever.

\*/

public int getNumObjects() {

System.out.println("number in heap " + everSeen.size());

return everSeen.size();

}

/\*\*

\*

\* @param oid

\* the id of the object we want to access in the heap.

\* @return the last event record for the given object.

\*/

public ObjectClass getRecord(String oid) {

return objectStates.get(oid);

}

/\*\*

\*

\* @return the current time expressed as sequence.

\*/

public int getTimeSequence() {

return timeSequence;

}

/\*\*

\*

\* @return the current time expressed as allocated object size.

\*/

public int getTimeSize() {

return timeSize;

}

/\*\*

\*

\* @return the current time expressed in term of method entry and exit.

\*/

public int getTimeMethod() {

return timeMethod;

}

protected void handleIgnorePreaccess() {

System.out.println("ignore preaccess");

// as this method just ignores preaccess it doesn't do anything

}

protected void handleAllocateAtFirstAccess(Event e) {

System.out.println("allocate at first access");

allocateObjectCheater(e);

updateObject(e);

}

protected void handleIgnorePostaccess() {

System.out.println("ignore postaccess");

// as this method just ignores postaccess it doesn't do anything

}

// no need for this in my opinion. we don't need an object to be dead to

// collect it

protected void handleMoveDeath() {

System.out.println("move death");

}

public void removeRecord(String objectID) {

objectStates.remove(objectID);

}

public HashMap<String, ObjectClass> getObjectStates() {

return objectStates;

}

public class LiveSizeChart extends ApplicationFrame {

/\*\*

\*

\*/

private static final long serialVersionUID = 1L;

private XYSeries data;

public LiveSizeChart() {

super("Live size chart");

data = new XYSeries("Data");

data.add(0, 0);

XYSeriesCollection dataset = new XYSeriesCollection(data);

// based on the dataset we create the chart

JFreeChart chart = ChartFactory.createXYLineChart(

"Livesize change over time", "Time", "Livesize", dataset,

PlotOrientation.VERTICAL, false, false, false);

// we put the chart into a panel

ChartPanel chartPanel = new ChartPanel(chart);

// default size

chartPanel.setPreferredSize(new java.awt.Dimension(800, 600));

// add it to our application

setContentPane(chartPanel);

pack();

}

public void updateChart(double x, double y) {

data.add(x, y);

}

}

public void setHeuristic(Heuristic h) {

this.heuristic=h;

}

}

package uk.ac.glasgow.etparser.handlers;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

import java.util.Queue;

import uk.ac.glasgow.etparser.LiveSizeChart;

import uk.ac.glasgow.etparser.ObjectClass;

import uk.ac.glasgow.etparser.ObjectLiveTime;

import uk.ac.glasgow.etparser.CommandParser.Heuristic;

import uk.ac.glasgow.etparser.events.CreationEvent;

import uk.ac.glasgow.etparser.events.Event;

import uk.ac.glasgow.etparser.events.Event.Check;

public class SmartHeap extends Heap {

private Queue<String> allocatedObjects;

private Heuristic heuristic;

private static SmartHeap smartHeap;

private SmartHeap(){

super();

allocatedObjects=new LinkedList<String>() ;

}

// public SmartHeap getTheHeap(){

// if (smartHeap==null){

// smartHeap=new SmartHeap ();

// }

// return smartHeap;

// }

//the same handle method as in Heap, no deallocations this time

// I think there should be deallocations + the new heuristic

@Override

public void handle(Event e) {

String currentObjectID = e.getObjectID();

String currentEventType = e.getTypeOfEvent();

// if never seen before

if (!existsInEverSeen(currentObjectID)) {

// create a new livetime for this object

ObjectLiveTime livetime = new ObjectLiveTime(currentObjectID);

// if the event is allocation- great

if (currentEventType.equalsIgnoreCase("a")) {

livetime.giveBirth();

allocateObject((CreationEvent) e);

allocatedObjects.add(currentObjectID);

System.out.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation

// report for notborn error

else {

System.out.println("not born because not allocated "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

everSeen.put(currentObjectID, livetime);

}

// the object has been seen before

else {

// if it wasn't allocated (check whether it has been deallocated

if (!existsInHeap(currentObjectID)&&!everSeen.get(currentObjectID).isDead()) {

// if the event is allocation- perfect

if (currentEventType.equalsIgnoreCase("a")) {

everSeen.get(currentObjectID).giveBirth();

allocateObject((CreationEvent) e);

allocatedObjects.add(currentObjectID);

System.out

.println("first allocation of " + currentObjectID);

}

// if the event isn't allocation report not born error

else {

System.out.println("not born because not allocated2 "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

}

// it has been allocated

else {

// take the object and check it's livetime

ObjectLiveTime currentObjectLivetime = everSeen

.get(currentObjectID);

// if it was never born, probably preaccess before and now again

// or probably dead

if (!currentObjectLivetime.isBorn()) {

System.out

.println("not live but exists- either preaccess or it was dead "

+ currentObjectID);

e.setCheck(Check.NOTBORN);

}

// the object died before this access

else if (currentObjectLivetime.isDead()) {

e.setCheck(Check.DEAD);

System.out.println("dead " + currentObjectID);

}

// it's legal to update this object

else {

// if the event isn't dead just make the update

e.setCheck(Check.LEGAL);

updateObject(e);

System.out.println("legal access " + currentObjectID);

}

}

}

}

public void setHeuristic(Heuristic h){

heuristic=h;

}

private boolean checkSizeLimitExcess() {

return allocatedMemSize >= 10;

}

private boolean sizeNormal() {

return allocatedMemSize < 8;

}

private void deallocateFirstObjects() {

while (checkSizeLimitExcess() && (!sizeNormal())) {

//get the id ot the first allocated object

//and remove it from the list of allocated objects

String currentObjectID = allocatedObjects.remove();

//get the size or the first allocated object

//and remove it from the heap

int sizeOfObject = objectStates.remove(currentObjectID).getSize();

//kill that object in the ever seen so

//it would be treated as a dead object from now on

everSeen.get(currentObjectID).kill();

//decrease the allocated memory size

allocatedMemSize -= sizeOfObject;

//decrease the livesize

livesize -= sizeOfObject;

}

System.out.println("first");

}

public void deallocateLeastRecentlyUsedObjects() {

//create a list of objects ordered by the time of last access

List<ObjectClass> timeOrderedObjects = getListOfObjectClassTimeSorted();

while (checkSizeLimitExcess() && (!sizeNormal())) {

//take the least recently used object and remove it from the list

String currentObjectID = timeOrderedObjects.remove(0).getID();

//get the size of least recently used object and

//remove the object from the heap

int sizeOfObject = objectStates.remove(currentObjectID).getSize();

//decrease the allocated memory size

allocatedMemSize -= sizeOfObject;

}

System.out.println("least");

}

// getters

public long getLiveSize() {

return livesize;

}

public long getAllocatedMemSize() {

return allocatedMemSize;

}

public void deallocate() {

switch (heuristic) {

case LEASTRECENTLYUSED:

deallocateLeastRecentlyUsedObjects();

break;

case FIRST:

deallocateFirstObjects();

break;

default: deallocateFirstObjects();

}

}

/\*\*

\* Method the takes all objects from the heap and orders them into a list of

\* objects sorted by their time of last access. Good if we want to dispose

\* of least recently used objects.

\*

\* @return list of all objects sorted according to the time of last access

\*/

public List<ObjectClass> getListOfObjectClassTimeSorted() {

Heap theHeap = Heap.getTheHeap();

HashMap<String, ObjectClass> objects = theHeap.getObjectStates();

ArrayList<ObjectClass> listOfObjects = new ArrayList<ObjectClass>();

for (ObjectClass obj : objects.values()) {

listOfObjects.add(obj);

}

Collections.sort(listOfObjects, new Comparator<ObjectClass>() {

public int compare(ObjectClass o1, ObjectClass o2) {

return Integer.compare(o1.getTimeOfLastEvent(),

o2.getTimeOfLastEvent());

}

});

return listOfObjects;

}

}