Kevin Jonaitis
36393693

Project 2 Final Document

Introduction

The purpose of this project is to implement a main memory manager for **variable size** partitions using linked lists. This project will also compare different allocations strategies using a simulation.

Data Structures

Main Memory

Main memory array:

| size/allocated | \\\\A \\\\\| size/allocated| size/free| pointer to previous free | pointer to next free| B | size /allocated| size/allocated| \\\\\\\ C \\\\\\\ | size/allocated |

Each size/pointer block is an int.

Pointer: an index to first part of previous/next block
Size: The size of the free space/allocated space. Negative means free, positive means allocated

Driver Release Linked-List

This is a list of allocated blocks, by the driver to randomly release a block

| Index of allocated block 1 | → | Index of allocated block 2 | -> | ... | -> | Index of allocated block k |

System Architecture

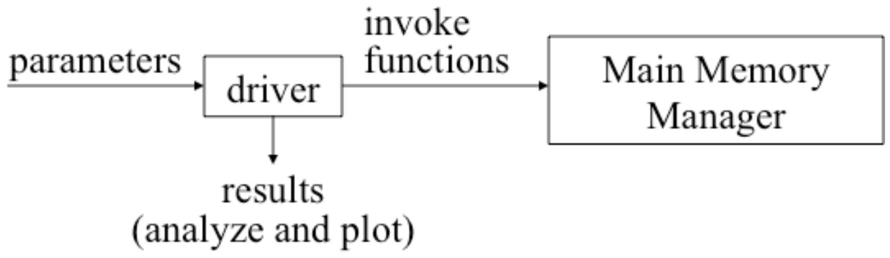
The architecture picture is as below. A main program calls the driver, and sets it up with specific parameters. The driver runs the main memory manager multiple times in "simulations", with each simulation running different parameters. The driver invokes functions on each "simulation" to run it.

Therefore, the driver will run the functions like determineNextBlockToRelease, and send commands such as mm_release and mm_request.

The main memory manager mainly implements mm_request and mm_release. Specifically, when running these request, the call structure is as follows:

mm_request()[called by driver] -> firstFit/bestFit()[called by main memory manager] ->mm_reqest(size,index)[called by main memory manager]

This way, firstFit/bestFit determines the index that will be allocated in a request, and returns a bad value if there is one. mm_request(size,index) handles the internals of the request.



The program will run as follows:

The main memory will be simulated in an array, so all "pointers" will actually be integers that correspond to indexes for different spots in main memory.

Free space will have a tag for size/allocated on each side of the allocated memory, and also pointers to previous/next free space.
Allocated space will simply size/allocated on each size of the allocated memory.

Main memory manager will be the actual "program", and will simulate allocating and releasing memory through the "invoke functions".
This functions are:

Request(numberOfBlocksToRequest)
Release(indexOfBlockToRelease)

For the allocation implementations, we will use First-Fit and Best-fit, as they are the most straightforward to implement.

The parameters specify what will be run in the specific simulation. It will include the following:

mem_size: size of main memory(in words, or “ints”)
a: average memory request size
d: standard distribution
strategy: first fit or best fit
sim_step: the number of steps per a sim

When initializing main memory, the strategy will be passed into the constructor, and determine which method is called for request().

Simulations will be run on main memory numerous times. The data will be output to a file. The steps to the simulation are as follows:

1. choose *a* and *d*
2. run simulator (driver) with one strategy s1 (sim_step times)
3. determine average memory utilization u_{s1} and average search time s_{s1}
4. run simulator with another strategy s2 (sim_step times)
5. determine average memory utilization u_{s2} and average search time s_{s2}
6. choose a different *a*
7. repeat steps 2-6
8. plot values of u_{s1} and u_{s2} against *a* (i.e., 2 curves)
9. plot values of s_{s1} and s_{s2} against *a* (i.e., 2 curves)
10. choose a different *d* and repeat steps 2-9 (with the same set of values of *a*)

With respects to recording data to the file, each iteration of the simulation will record memory usage, and time to find the memory.
At the end of each simulation, the “average” memory usage and average search time will be calculated.
These values, along with the a,d, and the strategy (first fit or best fit) will be printed on a single line of a file.
This process will be repeated for each value of a, and d, so there will be a * d lines in the file.

For actual values, we will sue the following:
sim_step: 10,000
a: 10 different values, being fractions of total mm; 1/10 * mm, 2/10 * m etc.
d: 4 different values, 0.1 * mm, 0.3 * mm, 0.5 * mm, mm
mm: use 10,000 ints for the size of MM

Test Cases

Request test cases:

- -Normal request
- -Request when all the memory is full
- -Request when the memory is empty
- -Specific test cases for each request type:
    - First-fit
        - -Allocate first block, allocate second block, free first block, then make a request(should use first block)
    - -Resume fit:
        - -Allocate first block, allocate second block, allocate third block, free first block, make a request, free second block, make a request, free first block, free third block, make a request (should use third block, not first one)

Release test cases:

- -Normal release
- -Release when there's no more memory available
- -Make sure it's releasing randomly, and not in-order

Combining free space:

- -Test combing two adjacent free memory spaces
- -Test combing THREE adjacent free memory spaces(if left and right are free, and you free the middle)

Pseudo Code

int mm_release(void *p)

```
    leftPointer = getLeftPointer();
    rightPointer = getRightPointer();
    totalFreeSize = p->size;
    if(left is free):
        totalFreeSize += leftPointer + 2; //2 is 1 for pointer, 1 for size/alloc
    if(right is free):
        totalFreeSize += rightPointer + 2; //2 is 1 for pointer, 1 for size/alloc
    if(left is free and right is free):
        setSize(leftPointer);
        setPointers(leftPointer,leftPointer->leftPointer,rightPointer-> rightPointer);
    else if (left)
        setSize(…)
        setPointers(…)
    else if (right)
        setSize(…)
        setPointers(…)
    else //just this one
        setSize(…)
        setPointers(…)
```

int firstFit_request(int p, int size)

```
    counter = 0;
    currentPointer = p;
    currentSize = getSize(p);
    while(currentSize != -1)
        counter++;
        if(currentSize > size)
            request(p,size);
            break
        p = getNextPointer(p)
    recordUtilizationAndSearchTime(counter)
```

int bestFit_request(int p, int size)

```
    counter = 0;
    int bestDifference;
    int bestDifferenceIndex;
    currentPointer = p;
    currentSize = getSize(p);
    while(currentSize != -1)
        counter++;
        if(currentSize > size && currentSize – size < bestDifference)
            bestDifference = currentSize – size;
            bestDifferenceIndex = p;
        p = getNextPointer(p)

    request(bestDifferenceIndex,size)
    recordUtilizationAndSearchTime(counter)
```

mm_request(int indexToRequest,size)

```
    CreateHeaderForExtraFreeSpace(indexToRequest,size)
    FixPointersToPreviousAndNextFreeSpace()
    getNextStartIndex(indexToRequest)
    setFree/AllocatedTag(indexToRequest,Size)
```

int getSizeOfNextRequest()

```
    return d * nextGaussian() + a
```

int recordUtilizationAndSearchTime(count)

```
    percentage = countTotalMemoryUtilization()
    addDataToGlobalData(percentage,count)
```

void selectBlockToRelease()

```
    randomValue = rand()
    counter = 0
    int currentFreeBlock = globalStartingFreeBlock
    while(counter != randomValue)
        counter++;
        getNextFreeBlock(currentFreeBlock)
    mm_release(currentFreeBlock)
```

Driver Code

```
for (i=0; i<sim_step; i++) {
  do {
    **get size n of next request**
    mm_request(n) }
  while (request successful)
  **record memory utilization**
  **select block p to be released**
  mm_release(p) }
```

Experiment Code

```
        For each value of d
                For each value of a
                        data = runSimluation(a,d,firstFit)
                        writeToFile(data)
                        data = runSimulation(a,d,bestFit)
                        writeToFile(data)
                plotValues(file)
```

Differences from Preliminary Document

Pointers

We did not implement the functionality of the “pointers” in the code. Rather, we simulated how the pointers would work. Because we only count a tick in “search time” when we look through a freed block, we can simply iterate over all blocks, and whenever we find a “free” block, we can increment the counter. This decreases the complexity of the implementation, while maintaining true to the simulation. It was also assumed that the program always knows what the "first" free pointer is at all times. This can easily be implemented in the release phase: when you release a block of memory, if the block's index is before your globally stored min pointer, replace this pointer with the new pointer.

Simulation Values

We slightly changed the values of a and d used in the simulation. In particular, here is the relevant code:

```
int mm = 100000;
int steps = 100000;
int[] a = new int[] { 100,200,300,500,(int) (0.1*mm),(int) (0.2*mm),(int) (0.3*mm)};
int[] d = new int[] { (int) (0.01 * mm), (int) (0.1 * mm), (int) (0.3 * mm)};
```

Test Cases

These test cases were tested mainly for correctness. Using the Eclipse debugger, I would verify by hand that the values in the array were correct. The values checked for the first test case are shown in the comments of the test code

```
import static org.junit.Assert.*;

import org.junit.Test;
/***
 * These test cases are manually tested, and values in memory are checked
to make sure they contain the correct values
 *
 * @author Kevin Jonaitis
 *
 */
public class TestCases {


	@Test
	public void testRequest() {
		MemoryManager m = new MemoryManager(14,MemoryManager.FIRSTFIT);
		m.mmRequest(5);
		m.mmRequest(5);
//		index 0: 5
//		index 6: 5
//		index 7: 5
//		index 13: 5
//		index 14: -6
//		index 15: -1
//		index 16: -1
//		index 21:
		}

	public void testNoSpaceRequest() {
		MemoryManager m = new MemoryManager(20,MemoryManager.FIRSTFIT);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRequest(10);
	}
	/**
	 * Is not needed, as the program will never attempt to release memory
it hasn't allocated;
	 * the allocaiton linked-list takes care of that
	 */
	public void testReleaseNonExistantMemory() {
		assertTrue(true);

	}

	public void testJustEnoughSpaceAllocation() {
		MemoryManager m = new MemoryManager(16,MemoryManager.FIRSTFIT);
		m.mmRequest(5);
		assertTrue(m.mmRequest(5));
		}
	public void testBadAllocation() {
		MemoryManager m = new MemoryManager(16,MemoryManager.FIRSTFIT);
		m.mmRequest(5);
		assertFalse(m.mmRequest(5));
		}
	public void testPerfectAllocation() {
		MemoryManager m = new MemoryManager(12,MemoryManager.FIRSTFIT);
		m.mmRequest(5);
		assertTrue(m.mmRequest(5));
		}

	//This test case will attemtp at combing two items together
	public void testRemoveMiddle() {
		MemoryManager m = new MemoryManager(20,MemoryManager.FIRSTFIT);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRelease(0);
		m.mmRelease(10);
		m.mmRelease(5);
	}
```

```java
	public void testRemoveLeft() {
		MemoryManager m = new MemoryManager(20,MemoryManager.FIRSTFIT);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRelease(10);
		m.mmRelease(5);
	}

	public void testRemoveRight() {
		MemoryManager m = new MemoryManager(20,MemoryManager.FIRSTFIT);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRequest(3);
		m.mmRelease(10);
	}
}
```

Code

MemoryManager.java

```java
import java.util.LinkedList;
import java.util.Random;


public class MemoryManager {

	//The allocation strategies
	public static final int FIRSTFIT = 0;
	public static final int NEXTFIT = 1;
	public static final int BESTFIT = 2;


	public static final int TAGSIZE = 2;


	int currentFreeIndex; //This is the index of the current freed block
	int allocationStrategy;
	int[] memory;

	int count; // The amount of holes searched in this iteration

	Random r; //Used to randomly generate a block to release

	public LinkedList<Integer> allocatedSpaces = new
LinkedList<Integer>();


	public MemoryManager(int memorySize, int allocationStrategy) {
		memory = new int[memorySize + TAGSIZE];
		r = new Random();
		memory[0] = memorySize * -1; //First size tag
		memory[1] = -99; //Pointer to previous free block
		memory[2] = -99; //Pointer to next free block
		currentFreeIndex = 0;
		memory[memorySize + TAGSIZE - 1] = memorySize * -1; //Second tag 
size
		this.allocationStrategy = allocationStrategy;
		count = 0;
	}
	
	public void resetCount() {
		count = 0;
	}
	public int getCount() {
		return count;
	}
	
	public void releaseRandomBlock() {
		if(allocatedSpaces.size() == 0)
			return;
		int randomValue = r.nextInt(allocatedSpaces.size());
		int index = allocatedSpaces.remove(randomValue);
		mmRelease(index);
	}
	
	/**
	 * Grabs the next free memory connected to this block's freed memory.
	 * If the currentIndex is a free memory, it will use the pointer to 
get the next free memory.
	 * 
	 * Otherwise, it will seek allocation by allocation until it finds a 
free memory
	 * @param currentIndex
	 * @return
	 */
	public int getNextFreeMemory(int currentIndex, int startIndex) {
		do {
			currentIndex = getRightMemoryIndex(currentIndex);
			if(currentIndex == -1)
				currentIndex = 0;
			if (currentIndex == startIndex)
					return -1;
			if(memory[currentIndex] < 0)
				return currentIndex;
		}while(true); //If we loop through all values and there's 
nothing empty, we'll get this
	}
	/**
	 * Allocate a piece of memory, and return false if it fails, true if 
it works
	 * @param memorySize
	 */
	public boolean mmRequest(int memorySize) {
		if(allocationStrategy == BESTFIT){
			int startIndex = currentFreeIndex;
			int bestFit = Integer.MAX_VALUE;
			int bestFitIndex = -1;
			boolean allocated = false; // Whether a successful 
allocation was made or not

			do{
				if(doesFit(currentFreeIndex, memorySize)) {
					if(Math.abs(memory[currentFreeIndex]) < 
bestFit) {
						bestFit = 
Math.abs(memory[currentFreeIndex]);
						bestFitIndex = currentFreeIndex;
					}
				}
				//Turns -1 when we've looped through the whole index
				currentFreeIndex = 
getNextFreeMemory(currentFreeIndex,startIndex);
				count++;
			} while(currentFreeIndex != -1);

			if(bestFitIndex != -1) {
				alloc(bestFitIndex,memorySize);
				allocated = true;
				allocatedSpaces.add(bestFitIndex);
			}

			 if(currentFreeIndex == -1) { //Reset when we get to the 
end
					currentFreeIndex = 0;
				}
			return allocated;
		} else {
			int startIndex = currentFreeIndex;
			int bestFit = -1;
			boolean allocated = false; // Whether a successful 
allocation was made or not

			do{
				if(doesFit(currentFreeIndex, memorySize)) {
					alloc(currentFreeIndex,memorySize);
					allocated = true;
					allocatedSpaces.add(currentFreeIndex);
					break;

				}
				currentFreeIndex = 
getNextFreeMemory(currentFreeIndex,startIndex);
				count++;
			} while(currentFreeIndex != -1);


			if(allocationStrategy == FIRSTFIT)
				currentFreeIndex = 0;
			else if(currentFreeIndex == -1) { //for best fit, we keep 
the index where it is, but if it's -1 reset to 0
				currentFreeIndex = 0;
			}
			//System.out.println("Current index:" + currentFreeIndex);
			return allocated;
		}

	}

	public void mmRelease(int indexToBeReleased) {
		//System.out.println("Releasing index:" + indexToBeReleased);

		int leftMemoryIndex = getLeftMemoryIndex(indexToBeReleased);
		int rightMemoryIndex = getRightMemoryIndex(indexToBeReleased);
		int size = memory[indexToBeReleased];
		int newSize = -1;

		if(leftMemoryIndex != -1 && memory[leftMemoryIndex] < 0
				&& rightMemoryIndex != -1 && memory[rightMemoryIndex] 
< 0) { //left and right
				newSize = memory[indexToBeReleased] +
memory[leftMemoryIndex] * -1 + memory[rightMemoryIndex] * -1 + 4; //8 is
for 4 sizes, and 4 pointers that are no longer needed


                    //Clear all the current data
                    clearAllDataAllocated(indexToBeReleased);
                    clearAllDataFree(leftMemoryIndex);
                    clearAllDataFree(rightMemoryIndex);

                    //Set new pointers
                    memory[leftMemoryIndex] = newSize * -1;
                    memory[leftMemoryIndex + 1] = 99; //
                    memory[leftMemoryIndex + 2] = 99;
                    memory[leftMemoryIndex + newSize + 1] = newSize * -1;
          } else if(leftMemoryIndex != -1 && memory[leftMemoryIndex] < 0)
{ //Left is free
               newSize = memory[indexToBeReleased] +
memory[leftMemoryIndex] * -1 + 2;


               //Clear all the current data
               clearAllDataAllocated(indexToBeReleased);
               clearAllDataFree(leftMemoryIndex);

               //Set new pointers
               memory[leftMemoryIndex] = newSize * -1;
               memory[leftMemoryIndex + 1] = 99; //
               memory[leftMemoryIndex + 2] = 99;
               memory[leftMemoryIndex + newSize + 1] = newSize * -1;

          } else if (rightMemoryIndex != -1 && memory[rightMemoryIndex] <
0) { //Just the right
               newSize = memory[indexToBeReleased] +
memory[rightMemoryIndex] * -1 + 2;

               //Clear all the current data
               clearAllDataAllocated(indexToBeReleased);
               clearAllDataFree(rightMemoryIndex);

               //Set new pointers
               memory[indexToBeReleased] = newSize * -1;
               memory[indexToBeReleased + 1] = 99; //
               memory[indexToBeReleased + 2] = 99;
               memory[indexToBeReleased + newSize + 1] = newSize * -1;
```

```
		} else { //Just this item
			newSize = memory[indexToBeReleased];


			//Clear all the current data
			clearAllDataAllocated(indexToBeReleased);

			//Set new pointers
			memory[indexToBeReleased] = newSize * -1;
			memory[indexToBeReleased + 1] = 99; //
			memory[indexToBeReleased + 2] = 99;
			memory[indexToBeReleased + newSize + 1] = newSize * -1;
		}

	}

	/**
	 * Clear all the data associated with this index
	 * @param index
	 */
	public void clearAllDataFree(int index) {
		int size = memory[index] * -1;
		memory[index] = 0;
		memory[index + 1] = 0; //left pointer is 0
		memory[index + 2] = 0; //right pointer is 0
		memory[index + size + 1] = 0;
	}

	/**
	 * Clear all the data associated with this index
	 * @param index
	 */
	public void clearAllDataAllocated(int index) {
		int size = memory[index];
		memory[index] = 0;
		memory[index + size + 1] = 0;
	}

	public void alloc(int currentIndex, int size) {

		//System.out.println("Requesting index:" + currentIndex + " with 
size" + size);
		int freeSize = memory[currentIndex] * -1; // Value is negative, 
so multiply by -1
```

```
            memory[currentIndex] = size; // Set left size tag
            memory[currentIndex + 1] = 0; // Set the pointers to 0
            memory[currentIndex + 2] = 0; // Set the pointers to 0
            memory[currentIndex + size + 1] = size; // Set right size tag, 
giving one more than the total space

            int rightIndex = getRightMemoryIndex(currentIndex);
            if(rightIndex == -1) //We fully allocated the whole space, 
return
                  return;

            //MEMORY MUST CLEAR CORRECTLY IN ORDER FOR THIS TO WORK
            if(memory[rightIndex] > 0) //We perfectly allocated a space, so 
no need to add the free space
                  return;
            int sizeLeft = freeSize - size - 2; //2 for the new tags,

            memory[rightIndex] = sizeLeft * -1;
            memory[rightIndex + 1] = 99;
            memory[rightIndex + 2] = 99;
            memory[rightIndex + sizeLeft + 1] = sizeLeft * -1; //Go forward 
free space + the size pointer

      }
      /**
       * The following two methods return the beginning indexes of 
allocated memory blocks to the left and right of the current allocated 
memory block
       * @param currentIndex
       * @return
       */
      public int getLeftMemoryIndex(int currentIndex) {
            if(currentIndex - 1 < 0)
                  return -1;
            else {
                  int size = memory[currentIndex - 1];
                  return currentIndex - 1 - Math.abs(size) - 1; //current 
index - 1 to get to size - size to get to beginning of size - 1 to get to 
index
            }
      }

      public int getRightMemoryIndex(int currentIndex) {
            if(currentIndex + Math.abs(memory[currentIndex]) + 2 >=
memory.length)
				return -1;
			else
				return currentIndex + Math.abs(memory[currentIndex]) + 2;
// Current index + the size of the currentIndex + tag for this + tag for
next
	}

	/**
	 * Checks to see if there's enough memory at this space to create an
insert.
	 * NOTE that when we allocate free memory, we need at least 3 spaces
for the tag + 2 pointers for the rest of the free space
	 * If this doesn't exist, we can't satisfy the request. Also note a
perfectly fitting allocation works as well.
	 */
	public boolean doesFit(int currentIndex, int size) {
		if(memory[currentIndex] >= 0) // This space is occupied
			return false;

		//If the space requested EXACTLY equals the size, or the space
requested has at least 3 extra spaces after it, return true
		if(memory[currentIndex] == (size * -1) || memory[currentIndex]
<= ((size + 4) * -1))
			return true;
		else
			return false;
	}

	public void setFreeTags(int index, int freeSpaceSize) {
		memory[index] = freeSpaceSize * -1; // Set free/allocated tag
along with the amount of space that is free

	}

	public int getMemoryUtilization() {
		int total = 0;
		for(Integer i : allocatedSpaces) {
			total = total + memory[i];
		}
		return total;
	}

	public int getSearchTime() {
		int currentCount = this.count;
		resetCount();
		return currentCount;
	}
}
```

Driver.java

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.Random;

public class Driver {
	static PrintWriter writer;
	MemoryManager m;

	int simSteps;
	int memorySize;
	Random r;
	int a;
	int d;
	int strategy;

	//Values used for calculations
	int[] memoryUtilization;
	int[] searchTime;

	public static void main(String[] args) throws
FileNotFoundException, UnsupportedEncodingException {
		writer = new PrintWriter("results.txt", "UTF-8");
		writer.write("d,a,Average Memory Utilization, Average Search
Time, Strategy\n");
		Driver driver = null;
		Driver driver2 = null;
		int mm = 100000;
		int steps = 100000;
		int[] a = new int[] { 100,200,300,500,(int) (0.1*mm),(int)
(0.2*mm),(int) (0.3*mm)};
		int[] d = new int[] { (int) (0.01 * mm), (int) (0.1 * mm),
(int) (0.3 * mm)};
```

```
		for(int i = 0; i < d.length; i++) {
			for(int j = 0; j < a.length; j++){
				driver = new 
Driver(steps,mm,a[j],d[i],MemoryManager.FIRSTFIT);
				driver.run();
			}
			for(int j = 0; j < a.length; j++){
				driver2 = new 
Driver(steps,mm,a[j],d[i],MemoryManager.BESTFIT);
				driver2.run();

			}
		}


		System.out.println("Done");
		writer.close();

	}

	public Driver(int simSteps, int memorySize, int a, int d,int 
strategy) {
		this.simSteps = simSteps;
		this.memorySize = memorySize;
		this.a = a;
		this.d = d;
		this.strategy = strategy;
		this.memoryUtilization = new int[simSteps];
		this.searchTime = new int[simSteps];
		this.m = new MemoryManager(memorySize,strategy);
		r = new Random();
	}

	public int recordUtilization(int currentStep) {
		memoryUtilization[currentStep] = m.getMemoryUtilization();
		searchTime[currentStep] = m.getSearchTime();
		return -1;
	}

	public int getRequestSize() {
		int requestSize = -1;
		while(requestSize < 2 || requestSize > memorySize) // Minimum 
request size allowed is 2; anything smaller will cause an error when you
```

```
go from allocated to deallocated, because allocated spots take a minimum
of 2 spaces, but free spaces take a minimum of 4
                requestSize = (int) (d * r.nextGaussian() + a);
            return requestSize;
        }

        public void run() {
            boolean success = true;
            for(int i = 0; i <simSteps; i++){
                do {
                    int n = getRequestSize();
                    success = m.mmRequest(n);

                } while(success);
                recordUtilization(i);
                m.releaseRandomBlock();
            }
            compileStatistics();
        }

        private void compileStatistics() {
            double averageMemoryUtilization = 0;
            double averageSearchTime = 0;
            double averageMemoryRequest = 0;
            for(int i = 0; i <simSteps; i++) {
                averageMemoryUtilization = averageMemoryUtilization +
memoryUtilization[i];
                averageSearchTime = averageSearchTime + searchTime[i];
            }
            averageMemoryUtilization = averageMemoryUtilization / simSteps;
            averageMemoryUtilization = (averageMemoryUtilization /
this.memorySize) * 100;
            averageSearchTime = averageSearchTime / simSteps;
            averageMemoryRequest = (((double) a) / this.memorySize) * 100;
            double standardDeviation = (((double) d) / this.memorySize) *
100;

            String line = standardDeviation + "," + averageMemoryRequest +
"," + averageMemoryUtilization + "," + averageSearchTime + "," + strategy
+ "\n";
            System.out.print(line);
            writer.write(line);
        }
}
```