Kaarel Kaasla
Tanel Marran
(Lab group 1)

Software Testing Homework 1: Debugging

Issue descriptions

Max binary heap

Name	Line number	Broken line	Fixed line
Issue 1, bug 1	71	return index / 2;	return (index-1) / 2;
Issue 1, bug 2	35-36	if (valueList.size() > (2 * index) - 1) {return (2 * (index) - 1);}	if (valueList.size() > (2 * index) + 1) {return (2 * (index)+ 1);}
Issue 1, bug 3	89	valueList.set(index,rightChild);	valueList.set(index,leftChild);
Issue 1, bug 4	96	A missing method call	BubbleDown(rightChildIndex);
Issue 2, bug 5	121	valueList.set(parent, index);	valueList.set(index, parent);
Issue 3, bug 6	155, 158	BubbleUp(indexOfRemoveElem); and BubbleDown(indexOfRemoveElem);	BubbleDown(indexOfRemoveElem); and BubbleUp(indexOfRemoveElem);

Genetic algorithm

Name	Line number	Broken line	Fixed line
Issue 1, bug 1	18, 19	for (int i = 7; i >0; i--) and for (int j = 7; j >0; j--)	for (int i = 7; i >=0; i--) and for (int j = 7; j >=0; j--)
Issue 1, bug 2	46, 49	if (viewableElem-curren	if (viewableElem-curren

		t==diff) return false; and return true;	t==diff) return true; and return false;
Issue 1, bug 3	46	if (viewableElem-current==diff) return true;	if (Math.abs(viewableElem-current==diff)) return true;
Issue 2, bug 4	74	halfPop.individuals.add(nextGenIv2);	newPop.individuals.add(nextGenIv2);

Max binary heap

Issue 1, bug 1

Method	**Variables**	**Intro and analysing info**	**Deciding point and final conclusion**	**Heuristics used**
TestHeap.main	Heap heap = new Heap(heapList); heapList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]	Since the code before creating the instance creates an arraylist, adds elements to it, and prints it, this is the first line where something could go wrong.	This is a good entry point since it allows us to go into the Heap class and explore its methods and program flow more in-depth.	1, 2, 3, 5
Heap.heapify()	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] lastIndex = 12 parentIndex = 6	It is important to check if the parentIndex variable is calculated correctly.	Since it is known that the parent index of a heap element can be calculated by ⌊(index - 1) / 2⌋, the parent index should be 5, but is 6. Therefore, there is something wrong in the calculation of this	1, 3, 5

			index.	
Heap.elemParentIndex(int index)	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] index = 12	It was evident while heapifying that this method returns an incorrect value. Therefore, it should be inspected further.	As was mentioned, the parent index of a heap element can be calculated by ⌊(index - 1) / 2⌋. However, this method calculates it index / 2. The calculation should be corrected accordingly.	1

Bug found in Heap.java line 71
Wrong line: return index / 2
Correct line: return (index - 1) / 2

Issue 1, bug 2

Method	**Variables**	**Intro and analysing info**	**Deciding point and final conclusion**	**Heuristics used**
Heap.heapify()	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] lastIndex = 12 parentIndex = 5 i = 5	Since we corrected the parentIndex calculation, we don't need to start the debugging process from the beginning of the code but rather the next line after the assignment of the now-correct parentIndex variable (the for-loop that follows)	This is a good breakpoint to go further into the code since we don't skip anything after fixing the last bug and at the same time, it allows us to go into the BubbleDown method.	2, 3
Heap.BubbleDown(int index)	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]	We should check whether the left and right indices of the	It can be seen from the variables that the leftChild has the value 4	1, 3, 5

	index = 5 currentElem = 8 leftChild = 4 rightChild = 0 leftChildIndex = 9 rightChildIndex = 12	children nodes are calculated correctly and this is done using methods to get both indices and values for both left and right children.	while it should have 1 and has the index 9 while should have 11. Therefore there is something wrong with calculating left node and its index and should be investigated further.	
Heap.leftChildElem(int index)	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] Index = 5	This is the method that calculates the left child value and since we saw that it gives an incorrect value, this method should be explored more in-depth.	This method calls the leftChildIndex method which is the one where the index is calculated so that method should also be explored.	1, 3
Heap.leftChildIndex(int index)	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] Index = 5	This is the method which calculates the left child index (and then passes it to leftChildElem which gives the value) so the mistake has to be in this method.	It is known that the left child index of a binary heap node is calculated (2 * index) + 1. However, in the given implementation, the calculation is (2 * index) - 1 which causes the error. Thus the bug is found and correcting it fixes both the incorrect index value and subsequently also the incorrect node value.	1, 2, 3

Bug found in Heap.java lines 35-36
Wrong lines: if (valueList.size() > (2 * index) - 1) {return (2 * (index) - 1);}
Correct lines: if (valueList.size() > (2 * index) + 1) {return (2 * (index) + 1);}

Issue 1, bug 3

Method	Variables	Intro and analysing info	Deciding point and final conclusion	Heuristics used
Heap.BubbleDown(int index)	valueList = [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] index = 5 currentElem = 8 leftChild = 1 rightChild = 0 leftChildIndex = 11 rightChildIndex = 12	Similar to earlier, it's not necessary to start debugging from the start, but rather from a place where a bug was earlier. For this reason, we start from the beginning of the BubbleDown method.	The method starts with assigning values and indices and as can be confirmed, these are fixed now. After that, the method should swap currentElem and leftChild, but instead changes currentElem to rightChild and leftChild to currentElem. For this reason, currentElem should be changed to leftChild, not rightChild in order to get rid of this bug.	3, 5, 8

Bug found in Heap.java line 89
Wrong line: valueList.set(index,rightChild);
Correct line: valueList.set(index,leftChild);

Issue 1, bug 4

Method	Variables	Intro and analysing info	Deciding point and final conclusion	Heuristics used
Heap.BubbleDown(int index)	valueList = [1, 2, 5, 10, 9, 8, 11, 7, 3, 4, 6, 1, 0] index = 2 currentElem = 5 leftChild = 8 rightChild = 11	After fixing the last bug and using the heuristic #8, a reasonable assumption can be made that the BubbleDown	As assumed, the assignment of values and indices at the beginning of the BubbleDown method is now done correctly, but since the	5, 8

	leftChildIndex = 5 rightChildIndex = 6	method works as intended until i = 2. For that reason, we should set the entry point to this method with i = 2 and explore further from there as it's the first iteration where we aren't of what's going to happen.	rightChild is bigger than leftChild and both are bigger than currentElem, the method should swap rightChild with currentElem and recursively call BubbleDown on the rightChildIndex. However, the method doesn't get called which is a bug and should be fixed. Thus, we should add BubbleDown(right ChildIndex) method call.	

Bug found in Heap.java line 96
Wrong line: a missing method call
Correct line: BubbleDown(rightChildIndex);

Issue 2, bug 5

Method	**Variables**	**Intro and analysing info**	**Deciding point and final conclusion**	**Heuristics used**
TestHeap.main	heap.addElem(13);	Before addElem are print statements which just output information and don't include any logic. Therefore it makes sense to examine the addElem method call where the contents of the method aren't explicitly clear.	This is a good entry point into the addElem method since we try to add a specific number into the heap and can use heuristics to compare whether the method adds the element correctly or not.	1, 2, 5, 8
Heap.addElem(int elem)	elem = 13	This is the method which	This method works correctly since it	1, 2

	valueList = [11, 10, 8, 7, 9, 1, 5, 2, 3, 4, 6, 1, 0, 13]	adds the elem variable at the end of the valueList and calls BubbleUp method on it.	adds the element at the end and then calls BubbleUp on the last index of the arraylist which corresponds to the added element	
Heap.BubbleUp(int index){	index = 13 currentElem = 13 parent = 5 parentIndex = 6 valueList = [11, 10, 8, 7, 9, 13, 13, 2, 3, 4, 6, 1, 0, 13]	This method must be explored so it could be concluded whether “bubbling up” of an element is calculated and switched correctly.	Looking at the variables, it can be noticed that currentElem, parent, parentIndex are calculated correctly (can also be assumed based on previous debugging), but the resulting valueList is incorrect as it should be [11, 10, 8, 7, 9, 1, 13, 2, 3, 4, 6, 1, 0, 5]. Looking at the the valueList elements are switched, it becomes apparent that valueList.set(parent, index) is incorrect since the .set method should take arguments index and value in that order, but presently the order is switched and should be changed to valueList.set(index, parent)	1, 2, 3, 8

Bug found in Heap.java line 121
Wrong line: valueList.set(parent, index);

Correct line: valueList.set(index, parent);

Issue 3, bug 6

Method	**Variables**	**Intro and analysing info**	**Deciding point and final conclusion**	**Heuristics used**
TestHeap.main	heap.removeElem(10)	After fixing the last bug, the addition of elements now works as expected so we can examine the removeElem method which removes a given element from the heap.	This is a good entry point to test the removeElem method since it allows us to go further into the method and examine its internal logic.	1, 2, 3
Heap.removeElem(int elem)	valueList = [13, 4, 11, 7, 9, 1, 8, 2, 3, 4, 6, 1, 0, 5] elem = 10 lastElem = 4 indexOfRemoveElem = 1 parentOfRemoved = 13	This method is worth looking into since it removes an element from the heap and if there are any mistakes in it, we can find these by going through the method.	The variable values are correct and the element with value 10 is removed correctly from the list. Since the lastElem variable is smaller than parentOfRemoved, we call BubbleUp(indexOfRemoveElem).	1, 2, 3, 5
Heap.BubbleUp(int index)	valueList = [13, 4, 11, 7, 9, 1, 8, 2, 3, 4, 6, 1, 0, 5] index = 1	Like we concluded earlier, the BubbleUp method works as intended. However, it's still worth debugging through to see if the result of the method call is expected.	We see that the method itself works correctly. However, it becomes evident that we shouldn't have called the BubbleUp method in this case. The BubbleUp method brings an element recursively towards the top of	1, 2, 3, 5, 8

			the heap, but since the element we are trying to bring upwards is smaller than its parent it makes no sense to try to bring it further up, instead it should be brought down instead. Therefore, we should re-examine the removeElem method.	
Heap.removeElem(int elem)	valueList = [13, 4, 11, 7, 9, 1, 8, 2, 3, 4, 6, 1, 0, 5] index = 1	Since we should've not called the BubbleUp method, we should examine the method to see why exactly it was called.	It can be seen that if the lastElem variable is smaller than its parent, BubbleUp is called and additionally, if its larger, BubbleDown is called on it. For the restructuring after removal to work correctly, the logic should be implemented in reverse; calling BubbleDown when the element is smaller since it might also be smaller than its children and calling BubbleUp when it's larger.	3, 8

Bug found in Heap.java lines 155, 158
Wrong lines: BubbleUp(indexOfRemoveElem); [line 155] BubbleDown(indexOfRemoveElem); [line 158]
Correct lines: BubbleDown(indexOfRemoveElem); [line 155] BubbleUp(indexOfRemoveElem); [line 158]

Genetic algorithm

Issue 1, bug 1

Method	Variables	Intro and Analysing info	Deciding point and Final conclusion	Heuristics used
TestGA, main()	Nothing being monitored	This is the first method that uses the Algorithm class, where the bug was reported. The assignment stated that there was no bug in the Population class.	We step into the method to see how the algorithm works.	1, 2
Algorithm, generation()	counter = 0	This is the main recursive method for the algorithm. Since this is the first generation (counter = 0), then we'll start by checking the getFitness method	We step into the getFitness() method for one of the individuals in the populations, to check if everything is in order.	1, 2
Algorithm, getFitness()	iv.list = [7, 7, 0, 2, 5, 1, 6, 4]	Checking the fitness calculation for a individual that we know has a clash	It can be noticed purely from habit that both of the for loops are not reaching the final 0 iteration, which isn't correct.	3

Bug found in Algorithm.java line 18, 19
Wrong line: for (int i = 7; i >0; i--) and for (int j = 7; j >0; j--)
Correct line: for (int i = 7; i >=0; i--) and for (int j = 7; j >=0; j--)

Issue 1, bug 2

Method	Variables	Intro and Analysing info	Deciding point and Final conclusion	Heuristics used
Algorithm, getFitness()	iv.list = [7, 7, 0, 2, 5, 1, 6, 4]	Continuing from where we left off with the last bug, we check the rest of the getFitness() method.	We step into the checkDiagonals() method, to check if everything works there.	1, 2
Algorithm, checkDiagonals()	iv.list = [7, 7, 0, 2, 5, 1, 6, 4]	It was important to understand what this method did before looking for bugs, so we stopped the debug flow and read the code through carefully.	After understanding that it checks that the row difference and column differences are equal, we noticed that the method returns false when it actually found a clash on one of its diagonals.	3, 8

Bug found in Algorithm.java line 46, 49
Wrong line: if (viewableElem-current==diff) return false; and return true;
Correct line: if (viewableElem-current==diff) return true; and return false;

Issue 1, bug 3

Method	Variables	Intro and Analysing info	Deciding point and Final conclusion	Heuristics used
Algorithm, checkDiagonals()	iv.list = [7, 7, 0, 2, 5, 1, 6, 4]	Continuing from where previously left off with the last bug, we check the rest of the checkDiagonals() method.	Notice that we're looking for the difference in distance for both the rows and the columns. This means that when we compare the differences, both values need to be positive, but on line	

			46, the Math.abs() method call is missing.	

Bug found in Algorithm.java line 46
Wrong line: if (viewableElem-current==diff) return true;
Correct line: if (Math.abs(viewableElem-current)==diff) return true;

Issue 2, bug 4

Method	**Variables**	**Intro and Analysing info**	**Deciding point and Final conclusion**	**Heuristics used**
Algorithm, checkDiagonals()	Nothing being monitored.	Continuing from the last bug, we look through the rest of the getFitness and checkDiagonal methods.	Since both of the aforementioned methods have been looked through, we return to the generation() method and continue from there.	
Algorithm, generation()	counter = 0	After checking the fitness calculation and fixing any issues there, we check the nextGeneration() method to see how generations are created.	From the return generation(nextGeneration(population)); line, we step into the nextGeneration() method.	1, 2
Algorithm, nextGeneration()	newPop.individuals.size() = 1	One at a time, we go through each line and step into any new methods that we haven't seen before.	We make sure that both the MateIv() and MutateIv() work as intended by stepping into them once.	1, 2
Algorithm, nextGeneration(	newPop.individuals.size() = 1	After reaching the second MateIv() call	We noticed that the second new,	3

)		of the entire for loop, we notice that the newPop individual list size did not change. There is a mistake here.	mutated individual is not actually added to newPop, but instead halfPop, which was the population used earlier in the method to store the top 50 individuals.	

Bug found in Algorithm.java line 73
Wrong line: halfPop.individuals.add(nextGenIv2);
Correct line: newPop.individuals.add(nextGenIv2);