Implementation documentation

Queue

Implemented methods: enqueue, dequeue, peek, clear, isEmpty, size, toLinkedList

All methods for this data structure only perform simple value assignments, some of them with if-checks. That way all Queue's actions are done in constant time and with constant memory. enqueue() creates a new object, which still requires constant memory. toLinkedList() method is an exception, as its time and memory requirement is O(n), where n is amount of elements in the Queue. Besides the elements, queue only needs to store a constant amount of variables. So, overall, memory requirement for the Queue is O(n).

LinkedList

Implemented methods: add, clear, isEmpty, reset, hasNext, getNext, size

All methods for this data structure only perform simple value assignments, some of them with if-checks. That way all LinkedList's actions are done in constant time and with constant memory. add() creates a new object, which still requires constant memory. Besides the elements, LinkedList only needs to store a constant amount of variables. So, overall, memory requirement for the LinkedList is O(n) where n is amount of elements in the LinkedList.

Tree

Implemented methods: add, remove, getMin, clear, isEmpty, contains, size, toLinkedList

Implemented tree is self-balancing using the red-black principle. Implementation mimics the guidelines presented in the Wikipedia article: `http://en.wikipedia.org/wiki/Red-black_tree`

add: First, tree finds a suitable placement for the new element. In worst case scenario, the placement in question may be found at the bottom of the tree. Because tree's height is below 2 * log(n) (n = amount of elements), finding the placement requires O(log(n)) time. After that several tree balancing techniques are used depending on a specific scenario. Most techniques require constant time, however one scenario requires to iterate the balancing techniques for added node's parent. Balancing on addition requires O(log(n)). Overall, time requirement is O(log(n)). Memory requirement is constant.

remove: Deleted element first must be found, which requires O(log(n)) time. Some balancing techniques are performed in constant time, others iterate upwards, requiring O(log(n)) time. Overall, time requirement is O(log(n)). Memory requirement is constant.

contains: Required element may be a leaf of the tree, in which case finding it requires O(log(n)) time. Same time is required, if tree doesn't contain given element. Overall, time requirement is O(log(n)). Memory requirement is constant.

toLinkedList: All elements are traversed breadth first using the queue. Each element is processed once, so required time is O(n). Elements need to be stored in queue and the LinkedList, which the method will return. Overall memory requirement is O(n)

getMin, clear, isEmpty and size perform at constant time and memory. Besides the elements, LinkedList only needs to store a constant amount of variables, making overall memory requirement O(n).

Tree can be extended to act as a map that holds key-value pairs. Overall memory requirement is not changed in O-notation.

get: finds the key element and returns its value. Time requirement O(log(n)), memory requirement constant.

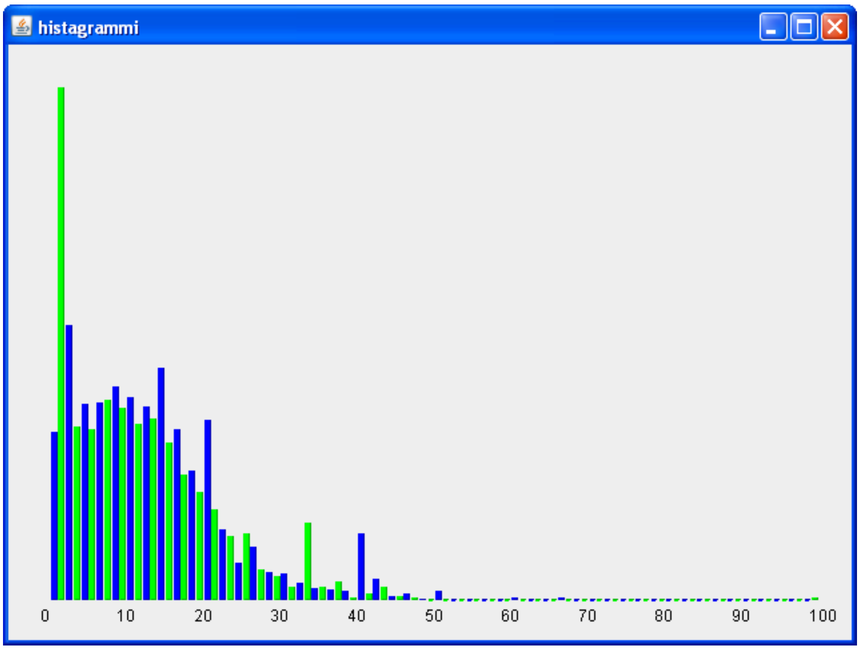
put: Calls contains method. If element is contained, elements value is changed. If not, add is called to add a new element to the tree. Time requirement O(log(n)), memory requirement constant.

Graph building with tracing

Tracing algorithm was largely underestimated in the requirements documentation. Algorithm's rough progression:

1. Choose a vertex v from which point of view other vertices are traced.
2. Choose another point p. Store the direction and distance from v to p. Also store direction and distance from v to p.right, assuming that p is a part of a polygon and thus has a connection with a left and right neighbour, all of which are a part of the same polygon. This data can be represented as a sector of two directions.
3. Repeat 2 for every point in the field. Sectors are stored in a heap, where a sector with the smallest left direction value is placed on top of the heap. Time requirement: O(p*log(p)) Memory requirement: O(p), where p is amount of geometry points.
4. Go through the sectors and remove redundant and overlapping sectors. Each stored sector is pulled from the heap (Time: O(p*log(p))) and stored in a tree (Time: O(p*log(p))), where sectors closest to the v are stored as a left child. Using this technique, amount of stored sectors is reduced significantly, where sector amount is s and s <= p. Memory requirement: O(p).
5. All vertices are checked whether they are obstructed by any sector. Time requirement: O(r*s), where r is amount of points with reflex angles (which is less than or equals p) and s is amount of usable sectors (also less than p). Memory requirement: O(q), where q is amount of points that are unobstructed to the v. That way q <= p.
6. Repeat this for every v in the field.

Overall time requirement: O(v * ((p*log(p)) + (r*s))), where v is amount of all vertices in the field, regardless if they are part of some polygon. That way p <= v, also we have established that r <= p and s <= p. So, in worst case scenario, we could argue that the time to rebuild the graph is: O(v * (p*log(p) + p^2)) = O(v^2 * log(v) + v^3). In current implementation, it is fair to assume that v = p, because with currently implemented tools user can only place two vertices, which are not a part of any polygon. To assume that r = p is a bit of a stretch, however that will happen if user decides to have all of the polygons to be reflex. The interesting bit is studying the relation between s and p.
Because s <= p, we can represent the relation as: s = c * p, where c ≥ 0 and c ≤ 1. This histagram shows the value of c retrieved by empirical means.



This data was collected from 55672 occurences. X-axis represents the value of c (in percents), and height of a column shows how many occurences fit the particular value interval. We can see that c hardly reaches above 0.5 and has a distinct peak at [0.1, 0.2]. This suggests that equating s with p isn't entirely fair. Now time requirement looks like: O(p^2 * (log(p) + c))
Memory requirement: O(p). At any stage of the algorithm no more than p amount of memory allocation is required.