

```

1 // package MMS;
2
3 /**
4  * Represents a list of Nodes. Each node holds a reference to a memory block.
5  * <br> (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
6  */
7 public class List {
8
9     private Node first = null; // The first (dummy) node of this list
10    private Node last = null; // The last node of this list
11    private int size = 0; // Number of elements (nodes) in this list
12
13    /**
14     * Constructs a new list of Node objects, each holding a memory block (MemBlock
15     * object)
16     */
17    public List() {
18        // Creates a dummy node and makes first and last point to it.
19        first = new Node(null);
20        last = first;
21    }
22
23    /**
24     * Adds the given memory block to the end of this list.
25     * Executes efficiently, in O(1).
26     * @param block The memory block that is added at the list's end
27     */
28    public void addLast(MemBlock block) {
29        Node newNode = new Node(block);
30        size++;
31        last.next = newNode;
32        last = newNode;
33    }
34
35    /**
36     * Adds the given memory block at the beginning of this list.
37     * Executes efficiently, in O(1).
38     * @param block The memory block that is added at the list's beginning
39     */
40    public void addFirst(MemBlock block) {
41        Node newNode = new Node(block);
42        newNode.next = first.next;
43        first.next = newNode;
44        if (size == 0) {
45            last = newNode;
46        }
47        size++;
48    }
49
50    /**
51     * Gets the node located at the given index in this list.
52     * @param index The index of the node to get, between 0 and size - 1
53     * @return The node at the given index
54     * @throws IllegalArgumentException If index is negative or greater than size -
55     */
56    public Node getNode(int index) {
57        if (index < 0 || ((size > 0) && index > (size - 1)) || ((size == 0) &&
58        index > size)) {
59            throw new IllegalArgumentException("index must be between 0 and (size -
60        1)");
61        }
62    }
63
64    /**
65     * Returns the size of the list.
66     */
67    public int size() {
68        return size;
69    }
70
71    /**
72     * Returns the first node of the list.
73     */
74    public Node firstNode() {
75        return first;
76    }
77
78    /**
79     * Returns the last node of the list.
80     */
81    public Node lastNode() {
82        return last;
83    }
84
85    /**
86     * Returns the next node of the given node.
87     */
88    public Node nextNode(Node node) {
89        return node.next;
90    }
91
92    /**
93     * Returns the previous node of the given node.
94     */
95    public Node previousNode(Node node) {
96        return null;
97    }
98
99    /**
100    * Returns the node at the given index.
101    */
102    public Node nodeAt(int index) {
103        return getNode(index);
104    }
105
106    /**
107    * Returns the node at the given index.
108    */
109    public Node nodeAt(int index) {
110        return getNode(index);
111    }
112
113    /**
114    * Returns the node at the given index.
115    */
116    public Node nodeAt(int index) {
117        return getNode(index);
118    }
119
120    /**
121    * Returns the node at the given index.
122    */
123    public Node nodeAt(int index) {
124        return getNode(index);
125    }
126
127    /**
128    * Returns the node at the given index.
129    */
130    public Node nodeAt(int index) {
131        return getNode(index);
132    }
133
134    /**
135    * Returns the node at the given index.
136    */
137    public Node nodeAt(int index) {
138        return getNode(index);
139    }
140
141    /**
142    * Returns the node at the given index.
143    */
144    public Node nodeAt(int index) {
145        return getNode(index);
146    }
147
148    /**
149    * Returns the node at the given index.
150    */
151    public Node nodeAt(int index) {
152        return getNode(index);
153    }
154
155    /**
156    * Returns the node at the given index.
157    */
158    public Node nodeAt(int index) {
159        return getNode(index);
160    }
161
162    /**
163    * Returns the node at the given index.
164    */
165    public Node nodeAt(int index) {
166        return getNode(index);
167    }
168
169    /**
170    * Returns the node at the given index.
171    */
172    public Node nodeAt(int index) {
173        return getNode(index);
174    }
175
176    /**
177    * Returns the node at the given index.
178    */
179    public Node nodeAt(int index) {
180        return getNode(index);
181    }
182
183    /**
184    * Returns the node at the given index.
185    */
186    public Node nodeAt(int index) {
187        return getNode(index);
188    }
189
190    /**
191    * Returns the node at the given index.
192    */
193    public Node nodeAt(int index) {
194        return getNode(index);
195    }
196
197    /**
198    * Returns the node at the given index.
199    */
200    public Node nodeAt(int index) {
201        return getNode(index);
202    }
203
204    /**
205    * Returns the node at the given index.
206    */
207    public Node nodeAt(int index) {
208        return getNode(index);
209    }
210
211    /**
212    * Returns the node at the given index.
213    */
214    public Node nodeAt(int index) {
215        return getNode(index);
216    }
217
218    /**
219    * Returns the node at the given index.
220    */
221    public Node nodeAt(int index) {
222        return getNode(index);
223    }
224
225    /**
226    * Returns the node at the given index.
227    */
228    public Node nodeAt(int index) {
229        return getNode(index);
230    }
231
232    /**
233    * Returns the node at the given index.
234    */
235    public Node nodeAt(int index) {
236        return getNode(index);
237    }
238
239    /**
240    * Returns the node at the given index.
241    */
242    public Node nodeAt(int index) {
243        return getNode(index);
244    }
245
246    /**
247    * Returns the node at the given index.
248    */
249    public Node nodeAt(int index) {
250        return getNode(index);
251    }
252
253    /**
254    * Returns the node at the given index.
255    */
256    public Node nodeAt(int index) {
257        return getNode(index);
258    }
259
260    /**
261    * Returns the node at the given index.
262    */
263    public Node nodeAt(int index) {
264        return getNode(index);
265    }
266
267    /**
268    * Returns the node at the given index.
269    */
270    public Node nodeAt(int index) {
271        return getNode(index);
272    }
273
274    /**
275    * Returns the node at the given index.
276    */
277    public Node nodeAt(int index) {
278        return getNode(index);
279    }
280
281    /**
282    * Returns the node at the given index.
283    */
284    public Node nodeAt(int index) {
285        return getNode(index);
286    }
287
288    /**
289    * Returns the node at the given index.
290    */
291    public Node nodeAt(int index) {
292        return getNode(index);
293    }
294
295    /**
296    * Returns the node at the given index.
297    */
298    public Node nodeAt(int index) {
299        return getNode(index);
300    }
301
302    /**
303    * Returns the node at the given index.
304    */
305    public Node nodeAt(int index) {
306        return getNode(index);
307    }
308
309    /**
310    * Returns the node at the given index.
311    */
312    public Node nodeAt(int index) {
313        return getNode(index);
314    }
315
316    /**
317    * Returns the node at the given index.
318    */
319    public Node nodeAt(int index) {
320        return getNode(index);
321    }
322
323    /**
324    * Returns the node at the given index.
325    */
326    public Node nodeAt(int index) {
327        return getNode(index);
328    }
329
330    /**
331    * Returns the node at the given index.
332    */
333    public Node nodeAt(int index) {
334        return getNode(index);
335    }
336
337    /**
338    * Returns the node at the given index.
339    */
340    public Node nodeAt(int index) {
341        return getNode(index);
342    }
343
344    /**
345    * Returns the node at the given index.
346    */
347    public Node nodeAt(int index) {
348        return getNode(index);
349    }
350
351    /**
352    * Returns the node at the given index.
353    */
354    public Node nodeAt(int index) {
355        return getNode(index);
356    }
357
358    /**
359    * Returns the node at the given index.
360    */
361    public Node nodeAt(int index) {
362        return getNode(index);
363    }
364
365    /**
366    * Returns the node at the given index.
367    */
368    public Node nodeAt(int index) {
369        return getNode(index);
370    }
371
372    /**
373    * Returns the node at the given index.
374    */
375    public Node nodeAt(int index) {
376        return getNode(index);
377    }
378
379    /**
380    * Returns the node at the given index.
381    */
382    public Node nodeAt(int index) {
383        return getNode(index);
384    }
385
386    /**
387    * Returns the node at the given index.
388    */
389    public Node nodeAt(int index) {
390        return getNode(index);
391    }
392
393    /**
394    * Returns the node at the given index.
395    */
396    public Node nodeAt(int index) {
397        return getNode(index);
398    }
399
400    /**
401    * Returns the node at the given index.
402    */
403    public Node nodeAt(int index) {
404        return getNode(index);
405    }
406
407    /**
408    * Returns the node at the given index.
409    */
410    public Node nodeAt(int index) {
411        return getNode(index);
412    }
413
414    /**
415    * Returns the node at the given index.
416    */
417    public Node nodeAt(int index) {
418        return getNode(index);
419    }
420
421    /**
422    * Returns the node at the given index.
423    */
424    public Node nodeAt(int index) {
425        return getNode(index);
426    }
427
428    /**
429    * Returns the node at the given index.
430    */
431    public Node nodeAt(int index) {
432        return getNode(index);
433    }
434
435    /**
436    * Returns the node at the given index.
437    */
438    public Node nodeAt(int index) {
439        return getNode(index);
440    }
441
442    /**
443    * Returns the node at the given index.
444    */
445    public Node nodeAt(int index) {
446        return getNode(index);
447    }
448
449    /**
450    * Returns the node at the given index.
451    */
452    public Node nodeAt(int index) {
453        return getNode(index);
454    }
455
456    /**
457    * Returns the node at the given index.
458    */
459    public Node nodeAt(int index) {
460        return getNode(index);
461    }
462
463    /**
464    * Returns the node at the given index.
465    */
466    public Node nodeAt(int index) {
467        return getNode(index);
468    }
469
470    /**
471    * Returns the node at the given index.
472    */
473    public Node nodeAt(int index) {
474        return getNode(index);
475    }
476
477    /**
478    * Returns the node at the given index.
479    */
480    public Node nodeAt(int index) {
481        return getNode(index);
482    }
483
484    /**
485    * Returns the node at the given index.
486    */
487    public Node nodeAt(int index) {
488        return getNode(index);
489    }
490
491    /**
492    * Returns the node at the given index.
493    */
494    public Node nodeAt(int index) {
495        return getNode(index);
496    }
497
498    /**
499    * Returns the node at the given index.
500    */
501    public Node nodeAt(int index) {
502        return getNode(index);
503    }
504
505    /**
506    * Returns the node at the given index.
507    */
508    public Node nodeAt(int index) {
509        return getNode(index);
510    }
511
512    /**
513    * Returns the node at the given index.
514    */
515    public Node nodeAt(int index) {
516        return getNode(index);
517    }
518
519    /**
520    * Returns the node at the given index.
521    */
522    public Node nodeAt(int index) {
523        return getNode(index);
524    }
525
526    /**
527    * Returns the node at the given index.
528    */
529    public Node nodeAt(int index) {
530        return getNode(index);
531    }
532
533    /**
534    * Returns the node at the given index.
535    */
536    public Node nodeAt(int index) {
537        return getNode(index);
538    }
539
540    /**
541    * Returns the node at the given index.
542    */
543    public Node nodeAt(int index) {
544        return getNode(index);
545    }
546
547    /**
548    * Returns the node at the given index.
549    */
550    public Node nodeAt(int index) {
551        return getNode(index);
552    }
553
554    /**
555    * Returns the node at the given index.
556    */
557    public Node nodeAt(int index) {
558        return getNode(index);
559    }
560
561    /**
562    * Returns the node at the given index.
563    */
564    public Node nodeAt(int index) {
565        return getNode(index);
566    }
567
568    /**
569    * Returns the node at the given index.
570    */
571    public Node nodeAt(int index) {
572        return getNode(index);
573    }
574
575    /**
576    * Returns the node at the given index.
577    */
578    public Node nodeAt(int index) {
579        return getNode(index);
580    }
581
582    /**
583    * Returns the node at the given index.
584    */
585    public Node nodeAt(int index) {
586        return getNode(index);
587    }
588
589    /**
590    * Returns the node at the given index.
591    */
592    public Node nodeAt(int index) {
593        return getNode(index);
594    }
595
596    /**
597    * Returns the node at the given index.
598    */
599    public Node nodeAt(int index) {
600        return getNode(index);
601    }
602
603    /**
604    * Returns the node at the given index.
605    */
606    public Node nodeAt(int index) {
607        return getNode(index);
608    }
609
610    /**
611    * Returns the node at the given index.
612    */
613    public Node nodeAt(int index) {
614        return getNode(index);
615    }
616
617    /**
618    * Returns the node at the given index.
619    */
620    public Node nodeAt(int index) {
621        return getNode(index);
622    }
623
624    /**
625    * Returns the node at the given index.
626    */
627    public Node nodeAt(int index) {
628        return getNode(index);
629    }
630
631    /**
632    * Returns the node at the given index.
633    */
634    public Node nodeAt(int index) {
635        return getNode(index);
636    }
637
638    /**
639    * Returns the node at the given index.
640    */
641    public Node nodeAt(int index) {
642        return getNode(index);
643    }
644
645    /**
646    * Returns the node at the given index.
647    */
648    public Node nodeAt(int index) {
649        return getNode(index);
650    }
651
652    /**
653    * Returns the node at the given index.
654    */
655    public Node nodeAt(int index) {
656        return getNode(index);
657    }
658
659    /**
660    * Returns the node at the given index.
661    */
662    public Node nodeAt(int index) {
663        return getNode(index);
664    }
665
666    /**
667    * Returns the node at the given index.
668    */
669    public Node nodeAt(int index) {
670        return getNode(index);
671    }
672
673    /**
674    * Returns the node at the given index.
675    */
676    public Node nodeAt(int index) {
677        return getNode(index);
678    }
679
680    /**
681    * Returns the node at the given index.
682    */
683    public Node nodeAt(int index) {
684        return getNode(index);
685    }
686
687    /**
688    * Returns the node at the given index.
689    */
690    public Node nodeAt(int index) {
691        return getNode(index);
692    }
693
694    /**
695    * Returns the node at the given index.
696    */
697    public Node nodeAt(int index) {
698        return getNode(index);
699    }
700
701    /**
702    * Returns the node at the given index.
703    */
704    public Node nodeAt(int index) {
705        return getNode(index);
706    }
707
708    /**
709    * Returns the node at the given index.
710    */
711    public Node nodeAt(int index) {
712        return getNode(index);
713    }
714
715    /**
716    * Returns the node at the given index.
717    */
718    public Node nodeAt(int index) {
719        return getNode(index);
720    }
721
722    /**
723    * Returns the node at the given index.
724    */
725    public Node nodeAt(int index) {
726        return getNode(index);
727    }
728
729    /**
730    * Returns the node at the given index.
731    */
732    public Node nodeAt(int index) {
733        return getNode(index);
734    }
735
736    /**
737    * Returns the node at the given index.
738    */
739    public Node nodeAt(int index) {
740        return getNode(index);
741    }
742
743    /**
744    * Returns the node at the given index.
745    */
746    public Node nodeAt(int index) {
747        return getNode(index);
748    }
749
750    /**
751    * Returns the node at the given index.
752    */
753    public Node nodeAt(int index) {
754        return getNode(index);
755    }
756
757    /**
758    * Returns the node at the given index.
759    */
760    public Node nodeAt(int index) {
761        return getNode(index);
762    }
763
764    /**
765    * Returns the node at the given index.
766    */
767    public Node nodeAt(int index) {
768        return getNode(index);
769    }
770
771    /**
772    * Returns the node at the given index.
773    */
774    public Node nodeAt(int index) {
775        return getNode(index);
776    }
777
778    /**
779    * Returns the node at the given index.
780    */
781    public Node nodeAt(int index) {
782        return getNode(index);
783    }
784
785    /**
786    * Returns the node at the given index.
787    */
788    public Node nodeAt(int index) {
789        return getNode(index);
790    }
791
792    /**
793    * Returns the node at the given index.
794    */
795    public Node nodeAt(int index) {
796        return getNode(index);
797    }
798
799    /**
800    * Returns the node at the given index.
801    */
802    public Node nodeAt(int index) {
803        return getNode(index);
804    }
805
806    /**
807    * Returns the node at the given index.
808    */
809    public Node nodeAt(int index) {
810        return getNode(index);
811    }
812
813    /**
814    * Returns the node at the given index.
815    */
816    public Node nodeAt(int index) {
817        return getNode(index);
818    }
819
820    /**
821    * Returns the node at the given index.
822    */
823    public Node nodeAt(int index) {
824        return getNode(index);
825    }
826
827    /**
828    * Returns the node at the given index.
829    */
830    public Node nodeAt(int index) {
831        return getNode(index);
832    }
833
834    /**
835    * Returns the node at the given index.
836    */
837    public Node nodeAt(int index) {
838        return getNode(index);
839    }
840
841    /**
842    * Returns the node at the given index.
843    */
844    public Node nodeAt(int index) {
845        return getNode(index);
846    }
847
848    /**
849    * Returns the node at the given index.
850    */
851    public Node nodeAt(int index) {
852        return getNode(index);
853    }
854
855    /**
856    * Returns the node at the given index.
857    */
858    public Node nodeAt(int index) {
859        return getNode(index);
860    }
861
862    /**
863    * Returns the node at the given index.
864    */
865    public Node nodeAt(int index) {
866        return getNode(index);
867    }
868
869    /**
870    * Returns the node at the given index.
871    */
872    public Node nodeAt(int index) {
873        return getNode(index);
874    }
875
876    /**
877    * Returns the node at the given index.
878    */
879    public Node nodeAt(int index) {
880        return getNode(index);
881    }
882
883    /**
884    * Returns the node at the given index.
885    */
886    public Node nodeAt(int index) {
887        return getNode(index);
888    }
889
890    /**
891    * Returns the node at the given index.
892    */
893    public Node nodeAt(int index) {
894        return getNode(index);
895    }
896
897    /**
898    * Returns the node at the given index.
899    */
900    public Node nodeAt(int index) {
901        return getNode(index);
902    }
903
904    /**
905    * Returns the node at the given index.
906    */
907    public Node nodeAt(int index) {
908        return getNode(index);
909    }
910
911    /**
912    * Returns the node at the given index.
913    */
914    public Node nodeAt(int index) {
915        return getNode(index);
916    }
917
918    /**
919    * Returns the node at the given index.
920    */
921    public Node nodeAt(int index) {
922        return getNode(index);
923    }
924
925    /**
926    * Returns the node at the given index.
927    */
928    public Node nodeAt(int index) {
929        return getNode(index);
930    }
931
932    /**
933    * Returns the node at the given index.
934    */
935    public Node nodeAt(int index) {
936        return getNode(index);
937    }
938
939    /**
940    * Returns the node at the given index.
941    */
942    public Node nodeAt(int index) {
943        return getNode(index);
944    }
945
946    /**
947    * Returns the node at the given index.
948    */
949    public Node nodeAt(int index) {
950        return getNode(index);
951    }
952

```

```

58     } else {
59         ListIterator current = new ListIterator(first.next);
60         for (int i = 0; i < index; i++) {
61             current.next();
62         }
63         return current.current;
64     }
65 }
66
67 /**
68  * Gets the memory block located at the given index in this list.
69  * @param index The index of the memory block to get, between 0 and size - 1
70  * @return The memory block at the given index
71  * @throws IllegalArgumentException
72  *         If index is negative or greater than size - 1
73  */
74 public MemBlock getBlock(int index) {
75     return getNode(index).block;
76 }
77
78 /**
79  * Gets given memory block.
80  * @param block The given memory block
81  * @return The index of the memory block, or -1 if the memory block is not in
this list
82  */
83 public int indexOf(MemBlock block) {
84     int index = 0;
85     ListIterator iterator = iterator();
86     while (iterator.hasNext()) {
87         MemBlock currentBlock = iterator.next();
88         // System.out.println("" + currentBlock);
89         if (currentBlock == block) {
90             return index;
91         }
92         index++;
93     }
94     return -1;
95 }
96
97 /**
98  * Adds a new node to this list, as follows:
99  * Creates a new node containing the given memory block,
100  * and inserts the node at the given index in this list.
101  * For example, if this list is (m7, m3, m1, m6), then
102  * add(2,m5) will make this list (m7, m3, m5, m1, m6).
103  * If the given index is 0, the new node becomes the first node in this list.
104  * If the given index equals the list's size - 1, the new node becomes the last
node in this list.
105  * If the new element is added at the beginning or at the end of this list,
106  * the addition's runtime is O(1). Otherwise is it O(size).
107  * @param block The memory block to add
108  * @param index Where to insert the memory block
109  * @throws IllegalArgumentException
110  *         If index is negative or greater than the list's size - 1
111  */
112 public void add(int index, MemBlock block) {
113     if (index < 0 || ((size > 0) && index > (size - 1))) {
114         throw new IllegalArgumentException("index must be between 0 and (size -
1)");
115     }
116     Node newNode = new Node(block);

```

```

117     Node theNodeBefore = getNode(index - 1);
118     theNodeBefore.next = newNode;
119     newNode.next = theNodeBefore.next;
120 }
121
122 /**
123  * Removes the first memory block from this list.
124  * Executes efficiently, in O(1).
125  * @throws IllegalArgumentException
126  *     If trying to remove from an empty list
127  */
128 public void removeFirst() {
129     if (size == 0) {
130         throw new IllegalArgumentException("Trying to delete value from an
empty list");
131     }
132     first = first.next;
133 }
134
135 /** Removes the given memory block from this list.
136  * @param block The memory block to remove
137  */
138 public void remove(MemBlock block) {
139     int blockIndex = indexOf(block);
140     if (blockIndex != -1) {
141         if (blockIndex == 0) {
142             removeFirst();
143             return;
144         }
145         Node theNodeBefore = getNode(blockIndex - 1);
146         theNodeBefore.next = getNode(blockIndex).next;
147     } else {
148         System.out.println("'" + block);
149     }
150 }
151
152 /**
153  * Returns an iterator over this list, starting with the first element.
154  * @return A ListIterator object
155  */
156 public ListIterator iterator() {
157     return new ListIterator(first.next);
158 }
159
160 /**
161  * A textual representation of this list.
162  * @return A string representing this list
163  */
164 public String toString() {
165     StringBuilder str = new StringBuilder("[ ");
166     // Creates a pointer to the first element
167     Node current = first.next; // skips the dummy
168     while (current != null) {
169         str.append(current.block + " ");
170         current = current.next;
171     }
172     str.deleteCharAt(str.length()-1);
173     str.append(" ]");
174     return str.toString();
175 }
176 }

```

```

1 // import java.util.ListIterator;
2
3 // package MMS;
4 /**
5  * Represents a managed memory space (also called "heap"). The memory space is
  managed by three
6  * methods: <br> <b> malloc </b> allocates memory blocks, <br> <b> free </b>
  recycles memory blocks,
7  * <br> <b> defrag </b> reorganizes the memory space, for better allocation and
  rescheduling.
8  * <br> (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
9  */
10 public class MemorySpace {
11
12     // A list that keeps track of the memory blocks that are presently allocated
13     private List allocatedList;
14
15     // A list that keeps track of the memory blocks that are presently free
16     private List freeList;
17
18     private int oldLength = 0;
19
20     /**
21      * Constructs a managed memory space ("heap") of a given maximal size.
22      *
23      * @param maxSize The size of the memory space to be managed
24      */
25     public MemorySpace(int maxSize) {
26         // Constructs and intilaizes an empty list of allocated memory blocks, and
  a free list containing
27         // a single memory block which represents the entire memory space. The base
  address of this single
28         // memory block is zero, and its length is the given memory size (maxSize).
29         allocatedList = new List();
30         freeList = new List();
31         freeList.addLast(new MemBlock(0, maxSize));
32     }
33
34     /**
35      * Allocates a memory block.
36      *
37      * @param length The length (in words) of the memory block that has to be
  allocated
38      * @return the base address of the allocated block, or -1 if unable to allocate
39      */
40     public int malloc(int length) {
41         // Scans the freeList, looking for the first free memory block whose length
  equals at least
42         // the given length. If such a block is found, the method performs the
  following operations:
43         // ListIterator iterator = freeList.iterator();
44         ListIterator iterator = new ListIterator(freeList.getNode(0));
45         MemBlock freeMemBlock = null;
46         while (iterator.hasNext() && freeMemBlock == null) {
47             MemBlock nextMeM = iterator.current.block;
48             if (nextMeM.length >= length) {
49                 freeMemBlock = nextMeM;
50             }
51             iterator.next();
52         }
53
54         if (freeMemBlock != null) {

```

```

55 // (1) A new memory block is constructed. The base address of the new
    block is set to
56 // the base address of the found free block. The length of the new
    block is set to the value
57 // of the method's length parameter.
58 MemBlock newMemBlock = new MemBlock(freeMemBlock.baseAddress, length);
59
60 // (2) The new memory block is appended to the end of the
    allocatedList.
61 allocatedList.addLast(newMemBlock);
62
63 // (3) The base address and the length of the found free block are
    updated, to reflect the allocation.
64 // For example, suppose that the requested block length is 17, and
    suppose that the base
65 // address and length of the the found free block are 250 and 20,
    respectively.
66 // In such a case, the base address and length of of the allocated
    block are set to 250 and 17,
67 // respectively, and the base address and length of the found free
    block are updated to 267 and 3, respectively.
68 if (length == freeMemBlock.length) {
69     freeList.remove(freeMemBlock);
70 } else {
71     freeMemBlock.baseAddress = freeMemBlock.baseAddress + length;
72     freeMemBlock.length = freeMemBlock.length - length;
73 }
74
75 // (4) The base address of the new memory block is returned.
76 return newMemBlock.baseAddress;
77 }
78
79 if (length != oldLength) {
80     oldLength = length;
81     defrag();
82     return malloc(length);
83 }
84 return -1;
85 }
86
87 /**
88  * Frees the memory block whose base address equals the given address
89  *
90  * @param address The base address of the memory block to free
91  */
92 public void free(int address) {
93     // Adds the memory block to the free list, and removes it from the
    allocated list.
94     // allocatedList.
95     ListIterator iterator = new ListIterator(allocatedList.getNode(0));
96     while (iterator.hasNext()) {
97         MemBlock nextMeM = iterator.current.block;
98         if (nextMeM.baseAddress == address) {
99             freeList.addLast(nextMeM);
100             allocatedList.remove(nextMeM);
101             break;
102         }
103
104         iterator.next();
105     }
106 }
107
108 /**

```

```

109     * A textual representation of this memory space
110     * @return a string representation of this memory space.
111     */
112     public String toString() {
113         // Returns the textual representation of the free list, a new line, and
then
114         // the textual representation of the allocated list, as one string
115         StringBuilder str = new StringBuilder(freeList.toString());
116         str.append("\n");
117         str.append(allocatedList.toString());
118         return str.toString();
119     }
120
121     /**
122     * Performs a defragmentation of the memory space.
123     * Can be called periodically, or by malloc, when it fails to find a memory
block of the requested size.
124     */
125     public void defrag() {
126         List newFreeList = freeList;
127         ListIterator iterator1 = freeList.iterator();
128         while (iterator1.hasNext()) {
129             MemBlock firstListMeM = iterator1.next();
130             MemBlock secondListMeM = null;
131             ListIterator iterator2 = newFreeList.iterator();
132             while (iterator2.hasNext()) {
133                 MemBlock tempMeM = iterator2.next();
134                 if (firstListMeM.baseAddress + firstListMeM.length ==
tempMeM.baseAddress) {
135                     secondListMeM = tempMeM;
136                 }
137             }
138
139             if (secondListMeM != null) {
140                 MemBlock combinedMeM = new MemBlock(firstListMeM.baseAddress,
firstListMeM.length + secondListMeM.length);
141                 newFreeList.remove(firstListMeM);
142                 newFreeList.remove(secondListMeM);
143                 newFreeList.addLast(combinedMeM);
144             }
145         }
146
147         freeList = newFreeList;
148     }
149 }

```