

Data Structures and Algorithms – Week 1 Code Samples

Lecturer: Arran Stewart

This zip file contains code samples we will refer to during week 1.

To understand the code samples well, you should make use of the textbook, which is *Data Structures and Problem Solving Using Java* (4th edition) by Mark Allen Weiss.

The outline for the Data Structures and Algorithms course contains suggested readings from the textbook that will help you understand the code here.

If there are sections of the textbook you particularly should read for some code listings, I will mention them here.

Stack: array implementation

To understand the `Stack.java` code, make sure you are familiar with the basics of the Java language, which are discussed in Chapter 1, “Primitive Java”, of the Weiss textbook.

You should also be familiar with *arrays* (discussed in section 2.4 of the textbook).

Weiss discusses stack and queue abstract data types in section 6.6 of the textbook. An *implementation* of stacks and queues, using arrays, is given in chapter 16 (“stacks and queues”) of the Weiss textbook. Refer to section 16.1, “Dynamic array implementations”, if you don’t understand the implementation given here.

```
1  import CITS2200.Overflow; //exception classes for error cases
2  import CITS2200.Underflow;
3
4  public class Stack {
5
6      public static int CAPACITY = 4; // default size for demo
7
8      int[] stack;           // stores objects in a stack
9      int topOfStack;        // index of the most recently added element
10
11     /**
12      * create an empty stack of default CAPACITY
13      */
14     public Stack() {
15         stack = new int[CAPACITY];
16         topOfStack = -1;
17     }
18
19     /**
20      * @return true if the stack is empty or false otherwise
21      */
22     public boolean isEmpty() {
23         return topOfStack == -1;
24     }
25
26     /**
27      * @return true if the stack is full or false otherwise
28      */
29     public boolean isFull() {
30         return topOfStack + 1 == CAPACITY;
31     }
32
33
34     /**
35      * Get the most recently inserted item in the stack.
36      * Does not alter the stack.
37      * @return the most recently inserted item in the stack.
38      * @throws Underflow exception if the stack is empty.
39      */
40     public int top() {
```

```

41     if (isEmpty()) {
42         throw new Underflow( "Stack empty so can not top" );
43     }
44     return stack[topOfStack];
45 }
46
47 /**
48  * Remove the most recently inserted item from the stack
49  * @throws Underflow exception if the stack is empty
50  */
51 public void pop() {
52     if (isEmpty()) {
53         throw new Underflow( "Stack empty so can not pop" );
54     }
55     topOfStack = topOfStack - 1;
56 }
57
58 /**
59  * Insert a new item into the stack
60  * @param x the item to insert
61  */
62 public void push( int x )
63 {
64     if( isFull() ) {
65         throw new Overflow( "Stack full so can not push " );
66     }
67     topOfStack = topOfStack + 1;
68     stack[ topOfStack ] = x;
69 }
70 }

```

Queue: array implementation

To understand the code listing here, you should be familiar with the readings mentioned for the *Stack* listing given previously.

```
1  import CITS2200.Overflow; //exception classes for error cases
2  import CITS2200.Underflow;
3
4  public class Queue {
5
6      public static int CAPACITY = 4;    // default queue capacity for demo
7
8      int[] queue;    // stores objects in a queue
9      int head;    // index of the head element
10     int length;    // number of objects in the queue
11
12     /**
13      * create an empty queue of default CAPACITY
14      */
15     public Queue() {
16         queue = new int[CAPACITY];
17         head = 0;
18         length = 0;
19     }
20
21     /**
22      * @return true if the queue is empty, false otherwise
23      */
24     public boolean isEmpty() {
25         return length == 0;
26     }
27
28     /**
29      * @return true if the queue is full, false otherwise
30      */
31     public boolean isFull() {
32         return length == CAPACITY;
33     }
34
35     /**
36      * add a new item to the queue if there is space
37      * @param item the item to add
38      * @throws Overflow exception if the stack is full
39      */
40     public void enqueue(int item) {
41         if (isFull()) {
42             throw new Overflow("Queue full so can not enqueue");
43         }
44         queue[(head + length) % CAPACITY] = item;
45         length = length + 1;
46     }
```

```

47
48  /**
49   * remove the head item in the queue
50   * @return the head item value if available
51   * @throws Underflow exception if the stack is empty
52   */
53 public int dequeue() {
54     if (isEmpty()) {
55         throw new Underflow("Queue empty so can not dequeue");
56     }
57     int item = queue[head];
58     head = (head + 1) % CAPACITY;
59     length = length - 1;
60     return (item);
61 }
62 }

```

Linked list node

This linked list node uses a Java feature called *generics*.

You will see it is defined using the class declaration `public class ListNode<T>`. The “T” acts a little like a blank space in a template: we can create different sorts of linked list by filling in the “T” with different types.

For instance, the following code creates a linked list node holding a `boolean`:

```
ListNode<int> myNode = new ListNode<int>(true);
```

whereas this code creates a linked list node holding a `char`:

```
ListNode<char> myNode = new ListNode<char>('a');
```

Java generics are discussed in the textbook in section 4.7, “Implementing generic components using Java 5 Generics”.

```
1  /**
2   * Node for a linked list of generic type (T)
3   * @author rachelcardell-oliver
4   * @version nov 2015
5   */
6  public class ListNode<T> {
7
8      public T value;
9      public ListNode<T> next;
10
11     /**
12      * Create a link node for a linked list
13      * @param val the value to be placed in this node
14      */
15     public ListNode(T value) {
16         this.value = value;
17         next = null;
18     }
19
20 }
```

Queue: linked list implementation

The following code implements the Queue abstract data type, using a linked list.

The Weiss textbook discusses how to implement stacks and queues in section 15.2, “Linked list implementations”.

```
1  import CITS2200.Underflow;
2
3  public class LinkedListStringQueue {
4
5      private ListNode<String> head;    // reference to the head of the queue
6      private ListNode<String> tail;    // reference to the tail of the queue
7      private int size;                // number of objects in the queue
8
9      /**
10       * create an empty queue
11       */
12     public LinkedListStringQueue() {
13         head = null;
14         tail = null;
15         size = 0;
16     }
17
18     /**
19      * add a new item to the queue
20      * @param item the item to add
21      */
22     public void enqueue(String a) {
23         ListNode<String> newnode = new ListNode<String>(a);
24         if (isEmpty()) {
25             head = newnode;
26         } else {
27             tail.next = newnode;
28         }
29         tail = newnode;
30         size = size + 1;
31     }
32
33     /**
34      * remove the head of the queue and return its value
35      * @return the head item value if available
36      * @throws Underflow exception if the queue is empty
37      */
38     public String dequeue() {
39         if (isEmpty()) {
40             throw new Underflow("Queue empty so can not dequeue");
41         }
42         String item = head.value;
43         head = head.next;
44         size = size - 1;
45         return (item);
46     }
47 }
```

```

46     }
47
48     /**
49     * return the value of the head of the queue
50     * @throws Underflow exception if the queue is empty
51     */
52     public String examine() {
53         if (isEmpty()) {
54             throw new Underflow("Queue empty so can not dequeue");
55         }
56         return(head.value);
57     }
58
59     /**
60     * @return true if the queue is empty, false otherwise
61     */
62     public boolean isEmpty() {
63         return size == 0;
64     }
65
66     /**
67     * @return size of the if the queue is empty
68     */
69     public int getSize() {
70         return size;
71     }
72
73     /**
74     * Print contents of the queue to System.out
75     */
76     public void printQueue() {
77         if (size > 0) {
78             System.out.println("Queue is empty");
79         } else {
80             System.out.println("Queue has "+ size +"elements:");
81             ListNode<String> q = head;
82             while (q != null) {
83                 System.out.println(q.value);
84                 q = q.next;
85             }
86         }
87     }
88 }
89
90

```


Four Sorting Algorithms

```
1
2
3  /**
4   * Utility class offering several sorting algorithms
5   */
6
7  public class SortingAlgorithms {
8      /**
9       * Execute the insertion sort algorithm sorting the argument
10      * array. There is no return since the parameter is mutated.
11      *
12      * @param arr the array of short integers to be sorted
13      */
14     public static void insertionSort(short[] arr) {
15         for (int j = 1; j < arr.length; j++) {
16             short key = arr[j];
17             int i = j - 1;
18             while (i >= 0 && arr[i] > key) {
19                 arr[(i + 1)] = arr[i];
20                 i = i - 1;
21             }
22             arr[i + 1] = key;
23         }
24     }
25
26     /**
27      * Execute the selection sort algorithm on an argument array.
28      * There is no return as the parameter is mutated.
29      * @param arr the array of short (numbers) to be sorted
30      */
31     public static void selectionSort(short[] arr) {
32         //build the sorted portion from 0 upwards
33         for (int i = 0; i < arr.length - 1; i++) {
34             //initialise the minimum value and its position
35             short min = arr[i];
36             int minpos = i;
37             //search the unsorted part for its smallest element
38             for (int j = i + 1; j < arr.length; j++) {
39                 if (arr[j] < min) {
40                     min = arr[j];
41                     minpos = j;
42                 }
43             }
44             if (i != minpos) { // swap min into place if necessary
45                 short temp = arr[i]; //copy
46                 arr[i] = arr[minpos]; //transfer
47                 arr[minpos] = temp; //replace
48             }
49         }
50     }
51 }
```

```

50     }
51
52
53
54
55
56
57     /**
58      * Execute the merge sort algorithm sorting the argument array.
59      * There is no return as the parameter is to be mutated.
60      * @param arr the array of short integers to be sorted
61      */
62     public static void mergeSort(short[] arr) {
63         mergeSort(arr, 0, arr.length-1);
64     }
65
66     private static void mergeSort(short[] arr, int l, int r) {
67         if (l < r) {
68             int mid = (l + r) / 2;
69             mergeSort(arr, l, mid);
70             mergeSort(arr, mid + 1, r);
71             merge(arr, l, mid, r);
72         }
73     }
74
75     /**
76      * Merge two parts of array arr from l to mid and mid+1 to r inclusive
77      * @param arr the array containing the portions for merging
78      * @param l left index, used for portion 1 from l to mid
79      * @param mid mid index, user for portion 2 from mid+1 to r
80      * @param r right index
81      */
82     private static void merge(short[] arr, int l, int mid, int r) {
83         int lsize = mid - l + 1;
84         int rsize = r - mid;
85         short[] left = new short[lsize];
86         short[] right = new short[rsize];
87
88         for (int i = 0; i < lsize; i++) {
89             left[i] = arr[l + i];
90         }
91         for (int j = 0; j < rsize; j++) {
92             right[j] = arr[mid + 1 + j];
93         }
94         int i = 0;
95         int j = 0;
96         int k = l;
97         while (i < lsize && j < rsize) {
98             if (left[i] < right[j]) {
99                 arr[k++] = left[i++];
100             } else {

```

```

101         arr[k++] = right[j++];
102     }
103 }
104 while ( i < lsize ) { // Copy rest of first half
105     arr[k++] = left[i++];
106 }
107 while( j < rsize ) { // Copy rest of second half
108     arr[k++] = right[j++];
109 }
110 }
111
112 /**
113  * Execute the quicksort algorithm sorting the argument array.
114  * There is no return as the parameter is to be mutated.
115  * @param arr the array of short integers to be sorted
116  */
117 public static void quickSort(short[] arr) {
118     quickSort(arr, 0, arr.length - 1);
119 }
120
121 /**
122  * Overloads the quickSort method with parameters to set the range to be sorted.
123  */
124 private static void quickSort(short[] arr, int p, int r) {
125     if (p < r) {
126         int q = partition(arr, p, r);
127         quickSort(arr, p, q - 1);
128         quickSort(arr, q + 1, r);
129     }
130 }
131
132 /**
133  * A private method to partition the array arr,
134  * between the indices start and finish inclusive
135  * inclusive. The method selects the element arr[r] as the pivot.
136  *
137  * @param arr the array to be sorted, which is mutated by the method
138  * @param p the lower index of the range to be partitioned
139  * @param r the upper index of the range to be partitioned
140  * @return the index of the point of partition
141  */
142 public static int partition(short[] arr, int start, int finish) {
143     short fence = arr[start]; //get the pivot value
144     int left = start+1;
145     int right = finish;
146     while (right >= left) {
147         while (left <= right && arr[left] <= fence)
148             left++;
149         while (right >= left && arr[right] >= fence)
150             right--;
151         if (right > left) {

```

```

152         short swap = arr[left];
153         arr[left] = arr[right];
154         arr[right] = swap;
155     }
156 }
157 arr[start] = arr[right];
158 arr[right] = fence;
159
160 return right; //return position of the fence
161 }
162
163
164
165
166 }

```

Three Search Algorithms

```

1  /**
2   * Three search algorithms
3   * Sequential and Step with
4   * Binary search from Weiss Fig 5.11
5   */
6  public class SearchAlgorithms {
7
8      public static int NOT_FOUND = -1;
9      public static boolean DEMO = true; //true to print debug statements
10
11     /**
12      * search for an item in an array this is the slowest search,
13      * O(N) but also the simplest
14      *
15      * @param a array to be searched, assumed to be sorted
16      * @param key item being searched for
17      * @return index of key in a if key is found, and NOT_FOUND otherwise
18      */
19     public static int SequentialSearch(int[] a, int key) {
20         for (int i = 0; i < a.length; i++) {
21             if (DEMO) { System.out.println("Testing position "+i); }
22             if (a[i] == key) { // found it!
23                 return i;
24             }
25         }
26         return NOT_FOUND; // if we get here, the item was not found
27     }
28
29     /**
30      * step search for an item in an array
31      * this is slightly faster than sequential search but still O(N)
32      *

```

```

33  * @param a array to be searched, assumed to be sorted
34  * @param key item being searched for
35  * @param step step size for moving through array
36  * @return index of key in a if key is found, and NOT_FOUND otherwise
37  */
38  public static int StepSearch(int[] a, int key, int step) {
39      int i = 0;
40      while (i < a.length) {
41          if (DEMO) { System.out.println("Testing position "+i); }
42          if (a[i] == key) { // found it
43              return i;
44          }
45          if (a[i] < key) { // flip forwards
46              i = Math.min(i + step, a.length - 1);
47          } else { // search back through the block
48              for (int j = i - 1; j > i - step; j--) {
49                  if (DEMO) { System.out.println("Testing position "+j); }
50                  if (a[j] == key) {
51                      return j;
52                  }
53              }
54              return NOT_FOUND;
55          }
56      }
57      return NOT_FOUND; // if we get here, the item was not found
58  }
59
60  /**
61   * Binary search to find key in array a is  $O(\log N)$ 
62   *
63   * @param a array to be searched, assumed to be sorted
64   * @param key item being searched for
65   * @return index of key in a if key is found, and NOT_FOUND otherwise
66   */
67  public static int BinarySearch(int[] a, int key) {
68      int low = 0;
69      int high = a.length - 1;
70      int mid;
71
72      while (low <= high) {
73          mid = (low + high) / 2;
74          // more generally ( a[ mid ].compareTo( x ) < 0 )
75          if (DEMO) { System.out.println("Testing position "+mid); }
76          if (a[mid] < key) { // continue to search lower part
77              low = mid + 1;
78          } else if (a[mid] > key) { // continue to search upper part
79              high = mid - 1;
80          } else { // we've found it
81              return mid;
82          }
83      }

```

```

84     // if we get here, the item was not found
85     return NOT_FOUND; // NOT_FOUND = -1
86 }
87
88 /**
89  * test code
90  */
91 public static void main(String[] args) {
92     //some test cases to see binary search at work
93     int[] a = new int[] { 11, 102, 223, 254, 265, 306, 367, 388, 399, 1000 };
94     int[] testkeys = new int[] { 265, 388, 1000, 200 };
95     for (int key : testkeys) {
96         if (DEMO) { System.out.println("Sequential Search for "+key); }
97         int res1 = SequentialSearch(a, key);
98         if (DEMO) { System.out.println("Step Search for "+key); }
99         int res2 = StepSearch(a, key, 3);
100        if (DEMO) { System.out.println("Binary Search for "+key); }
101        int res3 = BinarySearch(a, key);
102        System.out.println("Found key = " + key + " at position " +
103            res1 + "," + res2 + "," + res3);
104    }
105 }
106
107 }

```