Edith Cowan University
CSP2348
Data Structures
Assignment 2

Martin Ponce
Student 10371381

Tutor: Jitian Xiao

May 2, 2015

Contents

1 Introduction

This report examines array, singly-linked-list and binary tree data structures and algorithms to interact with them. Implementation of these data structures and algorithms in Java will be outlined in the report, along with analysis of algorithms used.

Throughout this report, it is assumed that the selection of algorithms is based on time efficiency rather than space efficiency. In other words, an algorithm with higher time efficiency but lower space efficiency will be selected over an algorithm with a lower time efficiency and higher space efficiency.

The array data structure will be demonstrated through the implementation of a simple lotto game. The lotto game allows up to 1000 players, with each player picking six unique integers, between 1 and 45. Each player and their respective lotto tickets are represented inside a two-dimensional array, while the winning numbers are represented inside a one-dimensional array. Merge sort is used to sort player tickets and winning numbers arrays, while binary search and an adaptation of an array merge algorithm is used to determine if a player's ticket contains the winning numbers.

2 Arrays

The array data structure is demonstrated through the implementation of a simple lotto game. The lotto game allows up to 1000 players, with each player picking six unique integers, between 1 and 45, which makes up their lotto ticket. Each player and their respective lotto tickets are represented inside a two-dimensional array, while the winning numbers are represented inside a one-dimensional array.

The following classes have been created to represent the lotto game:

- `Main`: The executable class, contains `main()` method
- `PlayerTickets`: Generates a two-dimensional array which represents each player, and their picks for the lotto ticket
- `WinningNumbers`: Generates a one-dimensional array which represents the winning numbers for the lotto game
- `WinningPlayers`: Contains logic to determine who the winning players are, which of their numbers match the winning numbers, and their winner class category
- `Sorter`: Helper class to sort `PlayerTickets` and `WinningNumbers` arrays
- `Randomizer`: Helper class to generate random numbers for each player pick and winning number

2.1 Sorting

In order to use more efficient search algorithms such as binary search, the arrays must be sorted first. The merge sort algorithm has been selected due to its time efficiency of $O(n\ log\ n)$. This algorithm is implemented as a static method of the `Sorter` class, as shown in Java code 2.1 and 2.2.

2.1.1 Merge sort algorithm

To sort a[left...right] into ascending order:

1. If left $\leq$ right:
   - 1.1. Let m be an integer about midway between left and right
   - 1.2. Sort a[left...m] into ascending order
   - 1.3. Sort $a[m+1$...right] into ascending order
   - 1.4. Merge a[left...m] and $a[m+1$...right] into auxiliary array b
   - 1.5. Copy all components of b into a[left...right]
2. Terminate

(Watt & Brown, 2001, p. 54)

2.1.2 Merge sort Java method

```java
1  private static void mergeSort(int low, int high) {
2
3      // 1.0 If left (low) < right (high)
4      if(low < high) {
5
6          // 1.1 Let m (mid) be an integer about midway between left and right
7          int mid = low + (high - low) / 2;
8
9          // 1.2 Sort a[left...m] into ascending order
10         mergeSort(low, mid);
11
12         // 1.3 Sort a[m+1...right] into ascending order
13         mergeSort(mid + 1, high);
14
15         // 1.4 Merge a[left...m] and a[m+1...right] into auxiliary array b
16         // call merge() which is O(n)
17         merge(low, mid, high);
18     }
19 }
```

Java code 2.1: Merge sort method

At line 17, the `mergeSort()` method calls supporting method `merge()`, as shown in Java code 2.2, in order to perform step 1.4 of the merge sort algorithm.

```java
private static void merge(int low, int mid, int high) {

    // iterate from low through to high
    for(int i = low; i <= high; i++) {

        // copy each element from the array to sort, to each element into temp array
        mergeTempArray[i] = mergeArrayToSort[i];
    }

    // 1.0 Set i = low, set j = mid + 1, set k = low
    int i = low;
    int j = mid + 1;
    int k = low;

    // 2.0 While i <= mid AND j <= high, repeat:
    while(i <= mid && j <= high) {

        // 2.1 If mergeTempArray[i] <= mergeTempArray[j],
        if(mergeTempArray[i] <= mergeTempArray[j]) {

            // 2.1.1 Copy mergeTempArray[i] into mergeArrayToSort[k], then increment i and k
            mergeArrayToSort[k] = mergeTempArray[i];
            i++;

        // 2.2 If mergeTempArray[i] > mergeTempArray[j],
        } else {

            // 2.2.1 Copy mergeTempArray[j] into mergeArrayToSort[k], then increment j and k
            mergeArrayToSort[k] = mergeTempArray[j];
            j++;
        }
        k++;
    }

    // 3.0 While i <= mid,
    while(i <= mid) {

        // 3.1 Copy mergeTempArray[i] into mergeArrayToSort[k], then increment i and k
        mergeArrayToSort[k] = mergeTempArray[i];
        k++;
        i++;
    }
}
```

Java code 2.2: Merge method

2.1.3 Merge sort analysis

As Watt and Brown (2001, p. 54 - 55) explain, analysis of the merge sort algorithm's time complexity involves counting the number of comparisons made during the operation. Let $n = \text{right} - \text{left} + 1$ be the length of the array, and let $C(n)$ be the total number of comparisons required to sort n values.

Step 1.1 involves dividing the array into two subarrays, $n/2$. The left subarray takes around $C(n/2)$ comparisons to sort, and similarly, the right subarray takes around $C(n/2)$ comparisons to sort.

Step 1.4 involves the merging of each subarray into a sorted array and takes about $n - 1$ comparisons to complete. Therefore:

$$C(n) \approx \begin{cases} 2C(n/2) + n - 1 & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases} \tag{2.1}$$

Simplifying equation 2.1:

$$C(n) \approx n \times \log_2 n \tag{2.2}$$

Therefore the time complexity is $O(n \log n)$.

Space complexity is $O(n)$, since step 1.4 requires an auxiliary array of length n to temporarily store the sorted array.

2.1.4 Merge sort console output

The following console outputs demonstrate that the merge sort algorithm is functioning correctly, and sorts player lotto picks and winning numbers in ascending order as desired. Note that the examples shown truncate actual results down to ten results from a thousand.

```
***********************
*** UNSORTED ARRAYS ***
***********************

Player 0001 picks: [16][14][37][31][07][42]
Player 0002 picks: [20][12][26][23][44][16]
Player 0003 picks: [10][32][20][35][02][24]
Player 0004 picks: [06][23][19][35][42][25]
Player 0005 picks: [07][17][28][41][29][38]
Player 0006 picks: [16][05][36][31][04][23]
Player 0007 picks: [20][01][34][37][07][18]
Player 0008 picks: [30][29][10][27][34][05]
Player 0009 picks: [03][27][13][38][28][32]
Player 0010 picks: [08][24][45][14][02][07]

Winning Numbers:   [21][22][10][03][04][13]
```

```
***********************
**** SORTED ARRAYS ****
***********************

Player 0001 picks: [07][14][16][31][37][42]
Player 0002 picks: [12][16][20][23][26][44]
Player 0003 picks: [02][10][20][24][32][35]
Player 0004 picks: [06][19][23][25][35][42]
Player 0005 picks: [07][17][28][29][38][41]
Player 0006 picks: [04][05][16][23][31][36]
Player 0007 picks: [01][07][18][20][34][37]
Player 0008 picks: [05][10][27][29][30][34]
Player 0009 picks: [03][13][27][28][32][38]
Player 0010 picks: [02][07][08][14][24][45]

Winning Numbers:   [03][04][10][13][21][22]
```

2.2 Searching

In order to determine the winners of the lotto game, the program must be able to search each number from the winning numbers array within each array of player tickets. Total number of winners within each winner class category must be calculated and displayed, as well as the result for an individual player, which simulates a player requesting their ticket to be checked for winning numbers.

For both functions to be implemented, two search algorithms have been selected, binary search and an adaptation of the merge array algorithm to sequentially compare components of two sorted arrays. Both of these algorithms may be implemented since the arrays have been sorted in ascending order. These algorithms are implemented as instance methods within the `WinningPlayers` class, as shown in Java code 2.3 and 2.4.

2.2.1 Binary search algorithm

To find which (if any) component of the sorted (sub)array a[left...right] equals target:

1. Set $l =$ left, $r =$ right
2. While $l \leq r$, repeat:
   - 2.1. Let m be an integer about halfway between l and r
   - 2.2. If target equals $a[m]$, terminate with answer m
   - 2.3. If target is less than $a[m]$, set $r = m - 1$
   - 2.4. If target is greater than $a[m]$, set $l = m + 1$
3. Terminate with answer none

(Watt & Brown, 2001, p. 43)

2.2.2 Binary search Java method

```java
private int binarySearch(int[] array, int target) {

    // 1.0 Set l = left, and r = right (substituted with low and high respectively)
    int low = 0;
    int high = array.length - 1;

    // 2.0 While l <= r, repeat:
    while(low <= high) {

        // 2.1 Let m (mid) be an integer about midway between l and r
        int mid = low + (high - low) / 2;

        // 2.2 If target equals a[m], terminate with answer m
        if(target == array[mid]) {
            return mid;

        // 2.3 If target is less than a[m], set r = m - 1
        } else if(target < array[mid]) {
            high = mid - 1;

        // 2.4 If target is greater than a[m], set l = m + 1
        } else {
            low = mid + 1;
        }
    }

    // 3.0 Terminate with answer none
    return -1;
}
```

Java code 2.3: Binary search method

2.2.3 Binary search analysis

Analysis of the binary search algorithm's time complexity involves counting the number of comparisons made during the operation. Let $n = \text{right} - \text{left} + 1$ be the length of the array, and assume that steps 2.2 to 2.4 are implemented as a single comparison. At most, these steps are iterated as many times as n can be halved until it reaches 0. Therefore the number of comparisons is $\text{floor}(\log_2 n) + 1$ (Watt & Brown, 2001). The time complexity for binary search is $O(\log n)$.

2.2.4 Binary search console output

The console output below displays the total number of winners in each winner class. To be considered a winner in this lotto game, a player must match at least 3 picks in their ticket to the winning numbers.

Each winner class is categorised by the number of matches a player may have between the picks in their ticket, and the winning numbers:

- 1st class: 6 picks match winning numbers
- 2nd class: 5 picks match winning numbers
- 3rd class: 4 picks match winning numbers
- 4th class: 3 picks match winning numbers

```
***********************
**** BINARY METHOD ****
***********************

1st class winners: 0
2nd class winners: 0
3rd class winners: 4
4th class winners: 26
```

The console output below displays results for individual players.

```
** BINARY TICKET CHECKING **

Player 0005 did not win. Thanks for playing lotto.
Better luck next time!

Player 0500 is a 4th class winner!
Your winning numbers are: [11][33][45]

Player 0564 did not win. Thanks for playing lotto.
Better luck next time!

Player 0897 did not win. Thanks for playing lotto.
Better luck next time!
```

The console output below displays the result after a user inputs their player number.

```
************************
****** USER INPUT ******
************************

ENTER YOUR PLAYER NUMBER TO CHECK IF YOU HAVE A WINNING TICKET:
500
** BINARY METHOD TICKET CHECK **
Player 0500 is a 4th class winner!
Your winning numbers are: [11][33][45]
```

2.2.5 "Merge search" algorithm

The following algorithm is an adaptation of the arrays merging algorithm. Rather than merge two sorted arrays into one sorted array, the algorithm is used to sequentially compare components from two sorted arrays to find a match between player lotto tickets and winning numbers.

To find which (if any) component from $a1[l1...r1]$ equals any component from $a2[l2...r2]$:

1. Set $i = l1$, set $j = l2$, let matchTally be the number of matches found, let playerMatchString record the matching components
2. While $i \leq r1$ and $j \leq r2$, repeat:
    - 2.1. If $a1[i] < a2[j]$:
        - 2.1.1. Increment i
    - 2.2. If $a1[i] > a2[j]$:
        - 2.2.1. Increment j
    - 2.3. If $a1[i] == a2[j]$:
        - 2.3.1. Increment matchTally
        - 2.3.2. Add matching component to playerMatchString
3. Terminate with answer matchTally

Adaptation of merging algorithm by Watt and Brown (2001, p. 46).

2.2.6 "Merge search" Java method

```java
private int mergeSearch(int[] playerTicket, int[] winningNumbers) {

    // 1.0 Set i = l1, set j = l2

    // i tracks playerTicket left
    int i = 0;
    // j tracks winningNumbers left
    int j = 0;
    // tracks how many matches found in loop
    int matchTally = 0;

    //  2.0 While i <= r1 AND j <= r2, repeat:
    while(i < playerTicket.length && j < winningNumbers.length) {

        // 2.1 If a1[i] < a2[j]:
        if(playerTicket[i] < winningNumbers[j]) {

            // 2.1.1 Increment i
            i++;

            // 2.2 If a1[i] > a2[j]:
        } else if(playerTicket[i] > winningNumbers[j]) {

            // 2.2.1 Increment j
            j++;

            // 2.3 If a1[i] == a2[j]
        } else {

            // 2.3.1 Increment matchTally
            matchTally++;

            // playerMatchString for checking individual ticket

            // update playerMatchString with matching array value
            playerMatchString += "[";

            // formatting: if value is less than 10, pad with leading zero
            if(winningNumbers[i] < 10) {
                playerMatchString += "0";
            }

            // complete the rest of the string
            playerMatchString += playerTicket[i] + "]";

            // increment i
            i++;
        }
    }

    // 3.0 Terminate
    return matchTally;
}
```

Java code 2.4: "Merge search" method

2.2.7 "Merge search" analysis

Analysis of the "merge search" algorithm's time complexity involves counting the number of comparisons made during the operation. Let $n_1 = \text{right1} - \text{left1} + 1$ be the length of $a1$[left1...right1], and let $n_2 = \text{right2} - \text{left2} + 1$ be the length of $a2$[left2...right2]. Let $n = n_1 + n_2$ be the total number of compared components (Watt & Brown, 2001, p. 48).

Assuming that step 2 is implemented as a single comparison, the loop is repeated at most, $n-1$ times (Watt & Brown, 2001, p. 48 - 49). Therefore the time complexity is $O(n)$. Space complexity is of $O(1)$ since no copies are made during the operation.

2.2.8 "Merge search" console output

The console output below displays the total number of winners in each winner class.

```
***********************
**** MERGE METHOD *****
***********************

1st class winners: 0
2nd class winners: 0
3rd class winners: 4
4th class winners: 26
```

The console output below displays results for individual players.

```
** MERGE TICKET CHECKING **

Player 0005 did not win. Thanks for playing lotto.
Better luck next time!

Player 0500 is a 4th class winner!
Your winning numbers are: [11][33][45]

Player 0564 did not win. Thanks for playing lotto.
Better luck next time!

Player 0897 did not win. Thanks for playing lotto.
Better luck next time!
```

The console output below displays the result after a user inputs their player number.

```
************************
****** USER INPUT ******
************************

ENTER YOUR PLAYER NUMBER TO CHECK IF YOU HAVE A WINNING TICKET:
500
** MERGE METHOD TICKET CHECK **
Player 0500 is a 4th class winner!
Your winning numbers are: [11][33][45]
```

2.2.9 Binary search vs. "merge search" comparison

- Binary search
  - Time complexity: $O(\log n)$
  - Space complexity: $O(1)$
- "Merge search"
  - Time complexity: $O(n)$
  - Space complexity: $O(1)$

3 Linked lists

The linked list data structure (singly linked list) is demonstrated through a list of students which stores their marks for a particular unit. Each node in the singly linked list represents a student. A student's ID number is used as the identifier, while storing results for Assignment 1, Assignment 2 and Exam Result. The list includes functions to insert new students into the list (while maintaining an ascending order based on a student's id number) and can determine which student has the highest overall mark within the list.

Additional functions have been added to the list, allowing a specific student to be removed from the list, and print the list in reverse, descending order based on student ID numbers. These additional methods have been implemented twice, in two separate packages.

`com.martinponce.csp2348.a2.linkedlistprogramming` maintains the original self-contained and executable class with additional methods `delete_unit_result()` and `reverse_print_unit_result()` appended.

`com.martinponce.csp2348.a2.linkedlistprogramming.alternative` contains a rewrite of the original `UnitList` class attempting to resolve issues with deleting a first node in the original `UnitList` class. Methods shown in this report from the alternative package will be labelled appropriately.

The following classes are created within `com.martinponce.csp2348.a2.linkedlistprogramming.alternative`:

- `Main`: The executable class, contains `main()` method
- `UnitList`: Defines the singly linked list header
- `UnitListNode`: Defines the singly linked list node

3.1 Deleting

In order to delete a specific node from the linked list, the program must be able to search for a key, which in this case is a student ID number. If the key is found within the list, the program will delete that specific node from the list.

For the function to be implemented, a singly linked list deletion algorithm has been created as shown below.

Implementation of the Java method requires a `void` method and therefore cannot return any data. A first node should not be physically deleted, even if it is logically deleted, otherwise the list will lose reference to all other nodes in the list. This will cause all other nodes to be deleted instead.

The exclusion of a variable which tracks the head of the linked list in the original `UnitList` class, this has caused issues where the deletion of the first node is not permanent. Therefore, the development of an alternative `UnitList` class is included in this report and source files, which tracks the head, and enables permanent deletion of the first node.

3.1.1 Delete target node algorithm

To delete *del* in SLL headed by *head*:

1. If list is empty:
   - 1.1. Terminate
2. Else let *current* be node *head*, and *previous* be null
3. While *current* does not match *del*, repeat:
   - 3.1. If end of list is reached:
     - 3.1.1. Terminate
   - 3.2. Else set node *previous* to node *current*
   - 3.3. Set node *current* to *current*'s next node
4. If node to be deleted is first node:
   - 4.1. Set *head* to *current*'s next node
5. Else set *previous*'s next node to *current*'s next node
6. Set *current*'s next node to null
7. Terminate

3.1.2 Delete target node Java method

```java
private static void delete_unit_result(UnitList u_list, int ID) {

    if(u_list == null) {
        System.out.println("\nError: List is empty!");
        return;
    } else if(ID < 999 || ID > 9999) {
        System.out.println("\nError: Student number "
                + ID + " is outside valid range!");
        return;
    }

    // cursors to traverse list
    UnitList current = u_list;
    UnitList previous = null;

    // traverse list
    while(current.student_ID != ID) {

        // if cursor traversed to end of list and target not found,
        if(current.next == null) {

            // print error message
            System.out.println("\nError: Student "
                    + ID + " not deleted. Student does not exist!");
            return;

        // else continue traversing
        } else {
            previous = current;
            current = current.next;
        }
    }

    // if current is at first node, and target matched
    // implied after exiting while loop and previous being null
    if(previous == null) {

        // TODO: fix issue where "deleted" first node not permanent

        // print action performed
        System.out.println("\nDeleted first student: " + current.student_ID);

        u_list = current.next;

    // else current is somewhere else down the list, and target matched
    } else {

        // print action performed
        System.out.println("\nDeleted student: " + current.student_ID);
        // set previous's next node to current's next node
        previous.next = current.next;
        // set current's next to null
        current.next = null;
    }

    print_unit_result(u_list);
}
```

Java code 3.1: Delete target node method

```java
1  public void deleteUnitResult(int studentID) {
2
3      // if list is empty,
4      if(isEmpty()) {
5          // print error message and terminate
6          System.out.println("\nError: List is empty!");
7          return;
8
9      // else if studentID outside rage,
10     } else if(studentID < 999 || studentID > 9999) {
11         // print error message and terminate
12         System.out.println("\nError: Student " + studentID + " is outside valid range!");
13         return;
14     }
15
16     // set cursors to start of list
17     cursorCurrent = head;
18     cursorPrevious = null;
19
20     // while current studentID does not match target studentID
21     while(cursorCurrent.getStudentID() != studentID) {
22
23         // if reached end of list,
24         if(cursorCurrent.getNext() == null) {
25             // print error message and terminate
26             System.out.println("\nError: Student " + studentID
27                     + " not deleted. Student does not exist!");
28             return;
29
30         // else continue traversing
31         } else {
32             cursorPrevious = cursorCurrent;
33             cursorCurrent = cursorCurrent.getNext();
34         }
35     }
36
37     // if targetID found in very first node,
38     if(cursorPrevious == null) {
39
40         // set head to current next
41         head = cursorCurrent.getNext();
42         // print success message
43         System.out.println("\nDeleted Student " + studentID + " (first node).");
44
45     // else targetID found somewhere else down the list,
46     } else {
47
48         // print success message
49         System.out.println("\nDeleted Student " + studentID + ".");
50         // set previous next to current next
51         cursorPrevious.setNext(cursorCurrent.getNext());
52         // set current next to null
53         cursorCurrent.setNext(null);
54     }
55     length--;
56 }
```

Java code 3.2: Alternative delete target node method

Line 41 in Java code 3.2 shows that an instance variable `head` is set to `cursorCurrent`'s next node, allowing persistent deletion of a first node.

3.1.3 Delete analysis

Analysis of this singly linked list delete algorithm involves counting traversals. Assuming *current* and *previous* traverse through each node in a single operation, they traverse between 0 to $n - 1$ nodes before finding or not finding the target key. Or

$(n-1)/2$ times on average. Therefore, the time complexity of this delete algorithm is $O(n)$ (Watt & Brown, 2001, p. 83). Space complexity is $O(1)$ since no copies are made.

3.1.4 Delete console output

The series of console outputs below demonstrates the execution of Java code 3.1, the delete implementation within the original `UnitList` class. It successfully "deletes" the first student 1111, but after subsequent calls to the method, student 1111 reappears in the list. However, deletions elsewhere in the list are persistent, as seen with student 1114.

```
Deleted first student: 1111

Student_No.: 1112
A1_mark: 10
A2_mark: 6
Exam_mark: 50

Student_No.: 1114
A1_mark: 14
A2_mark: 21
Exam_mark: 30

...
```

```
Deleted student: 1114

Student_No.: 1111
A1_mark: 17
A2_mark: 22
Exam_mark: 30

Student_No.: 1112
A1_mark: 10
A2_mark: 6
Exam_mark: 50

Student_No.: 1116
A1_mark: 8
A2_mark: 16
Exam_mark: 35

...
```

```
Deleted student: 1116

Student_No.: 1111
A1_mark: 17
A2_mark: 22
Exam_mark: 30

Student_No.: 1112
A1_mark: 10
A2_mark: 6
Exam_mark: 50

Student_No.: 1122
A1_mark: 11
A2_mark: 19
Exam_mark: 40

...
```

The series of console outputs below demonstrate the execution of Java code 3.2, the deletion implementation within the alternative `UnitList` class. This exhibits a persistent deletion of the first node, or in this case, student 1111.

```
Deleted Student 1111 (first node).

Student: 1112
A1 Mark: 10
A2 Mark: 6
Exam Mark: 50
Next Node: 1114

Student: 1114
A1 Mark: 14
A2 Mark: 21
Exam Mark: 30
Next Node: 1116

...
```

```
Deleted Student 1116.

Deleted Student 1145.

Student: 1112
A1 Mark: 10
A2 Mark: 6
Exam Mark: 50
Next Node: 1114

Student: 1114
A1 Mark: 14
A2 Mark: 21
Exam Mark: 30
Next Node: 1122

Student: 1122
A1 Mark: 11
A2 Mark: 19
Exam Mark: 40
Next Node: 1189

...
```

References

Watt, D. A., & Brown, D. F. (2001). *Java Collections - An Introduction to Abstract Data Types, Data Structures and Algorithms* (1st ed.). New York: John Wiley & Sons.