COMP20010: Data Structures and Algorithms I

Assignment 2

Q1.A)

```
function clearRecursively(stackToClear)

        Input: a stack to clear
        Output: the amount of elements removed

        if (stackToClear.empty())
                return 0
        stackToClear.pop()
        return 1 + clearRecursively(stackToClear)

end function
```

Q1.B)

```
function reverseStack(reverse, a, b)

        Input: reverse, the stack you want to reverse
                a, b two empty stacks of the same type of reverse
        Output: the reversed stack

        a := reverse
        while (a.size() != 0)
                b.push(a.pop)
        return b

end function
```

Q1.C)

```
function reverseQueue(q)

        Input: the queue you want to reverse
        Output: the reversed queue

        if (q.empty())
                return q
        data := q.dequeue()
        q := reverseQueue(q)
        q.enqueue(data)
        return q

end function
```

Q1.D)

Firstly,
One stack represents the front of the deque and the other represents the back of the deque.
Basically at all times one stack is just the reverse of the other.

insertFront(e)
    Push element e onto the front stack
    Then set the back stack equal to the reverse of the front stack
    running time: O(n)

insertBack(e)
    Push el onto the back stack
    Then set the front stack equal to the reverse of the back stack
    running time: O(n)

eraseFront()
    pop from the front stack
    Then set the back stack equal to the reverse of the front stack
    running time: O(n)

eraseBack()
    pop from the back stack
    Then set the front stack equal to the reverse of the back stack
    running time: O(n)

front()
    return top() of front stack
    running time: O(1)

back()
    return top() of back stack
    running time: O(1)

size()
    return the size of the front stack
    running time: O(1)

empty()
    return size() == 0
    running time: O(1)

Q1.E)
We will assume that SA is the ‘front’ stack and that SB is the ‘back’ stack.

```
function insertFront(e)

        Input: the element to put at the front

        SA.push(e)
        SB := SA.clone.reverse()

end function


function insertBack(e)

        Input: the element to put at the back

        SB.push(e)
        SA := SB.clone.reverse()

end function


function eraseFront()

        SA.pop()
        SB := SA.clone.reverse()

end function


function eraseBack()

        SB.pop()
        SA := SB.clone.reverse()

end function


function front()

        output: the element at the front of the deque

        return SA.top()

end function


function back()

        output: the element at the back of the deque

        return SB.top()

end function
```

```
function size()
        output: the number of elements in the deque
        return SA.size()
end function

function empty()
        output: a boolean value representing if the deque is empty
        return size() == 0
end function
```

Q2)

Parts A and B have been provided in a folder labeled Q2
Part A is the Student.java File
Part B is the PriorityStudents.java File

The Output of the code:

```
name: Aimee Quinn       age: 21     GPA: 2.7
name: Emilie Gibbs      age: 20     GPA: 3.2
name: Damion Sanders    age: 25     GPA: 3.2
name: Mira Weiss        age: 19     GPA: 3.5
name: Walker Holloway   age: 22     GPA: 3.8
name: Arianna Reeves    age: 20     GPA: 3.9
name: Nataly Ware       age: 21     GPA: 4.0
name: Aleah Gaines      age: 19     GPA: 4.1
name: Jeremy Schwartz   age: 18     GPA: 4.6
name: Lisa Boone        age: 22     GPA: 4.7
name: Karsyn Terry      age: 20     GPA: 4.8
name: Adelyn Walter     age: 24     GPA: 4.95
```

Q3.A)
which of the schemes use the array supporting the hash table exclusively and which of the schemes use additional storage external to the hash table.

-Open addressing can not tolerate a load factor above 1
-separate chaining can tolerate a load factor above 1

Q3.B)
The currently original String hash function implemented performs very poorly on certain classes of strings, including URLs.
(The poor performance is because of how the function samples characters in strings over 15 characters in length)

So basically in really long strings FilePaths and URLS are an example, There are many recurring elements and since the amount to skip was so small and not Radom there was a chance to hit the same element the same time in lots of different strings and they'd be hashed to the same number.

Especially Urls from the same website where they always have the same base!

the version mentioned in the lectures might be a better option because
Hashing large strings will be somewhat more expensive, as the new hash function examines every character, but Hashtables performance will, on balance, improve, as hash collisions will be vastly reduced.

Q3.C)

h(i) = (3i + 5)mod11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

h(12) = 8
h(44) = 5
h(13) = 0
h(88) = 5
h(23) = 8
h(94) = 1
h(11) = 5
h(39) = 1
h(20) = 10
h(16) = 9
h(5) = 9

Hash Table:

0	1	2	3	4	5	6	7	8	9	10
13	94				44			12	16	20
	39				88			23	5	
					11					

Parts D and E files have been provided in a folder labeled Q3
QD is in CountWords.java
QE is in CountWordsDictionary.java

Q3.D)
10 Most frequent words (Output of my program):

to	appeared	49 times (the most)
and	appeared	42 times
of	appeared	42 times
the	appeared	39 times
pleasure	appeared	27 times
pain	appeared	24 times
who	appeared	21 times
is	appeared	18 times
a	appeared	18 times
that	appeared	18 times (the 10th most)

For Question E I looped through every word, ran that word through a hash function then stored that unique hash in a hash table along with how many times that unique hash appeared (1 if it was its first time); I tried 2 different ways (The second way is the way in my code)

The second way I tried I just used the output of the hash function as the key for the for the hash table.

Q3.E.i)
Collisions using polynomial evaluation with 41: 43

Q3.E.ii)
Collisions using polynomial evaluation with 17: 1077

Q3.E.iii)
Collisions using a cyclic shift with 7: 587

Q3.E.iv)
Collisions using the old java hashCode: 298

The first way I used % on the output to map it to a range between 0 and 1000000 and I got the following collisions
Q3.E.i)
Collisions using polynomial evaluation with 41: 26024

Q3.E.ii)
Collisions using polynomial evaluation with 17: 26908

Q3.E.iii)
Collisions using a cyclic shift with 7: 28969

Q3.E.iv)
Collisions using the old java hashCode: 26332