**Problem Set 5: Queue, Sequential Search, Binary Search**

1. You are given the following Queue class:
3. public class Queue<T> {
4. public Queue() { ... }
5. public void enqueue(T item) { ... }
6. public T dequeue() throws NoSuchElementException { ... }
7. public boolean isEmpty() { ... }
8. public int size() { ... }
9. }

Complete the following *client* method (*not* a Queue class method) to implement the peek feature, using only the methods defined in the Queue class:

// returns the item at the front of the given queue, without

// removing it from the queue

public static <T> T peek(Queue<T> q)

throws NoSuchElementException {

/\*\* COMPLETE THIS METHOD \*\*/

}

Derive the worst case big *O* running time of your implementation. You may assume that the constructors and methods of the Queue class all have a worst case *O(1)* running time.

1. \* Suppose there is a long line of people at a check-out counter in a store. A new counter is opened, and people in the even positions (second, fourth, sixth, etc.) in the original line are directed to the new line. If a check-out counter line is modeled using a Queue class, we can implement this "even split" operation in this class.

Assume that a Queue class is implemented using a CLL, with a rear field that refers to the last node in the queue CLL, and that the Queue class already contains the following constructors and methods:

public class Queue<T> {

public Queue() { rear = null; }

public void enqueue(T obj) { ... }

public T dequeue() throws NoSuchElementException { ... }

public boolean isEmpty() { ... }

public int size() { ... }

}

Implement an additional method in this class that would perform the even split:

// extract the even position items from this queue into

// the result queue, and delete them from this queue

public Queue<T> evenSplit() {

/\*\* COMPLETE THIS METHOD \*\*/

}

Derive the worst case big *O* running time of your implementationYou may assume that the constructors and existing methods of the Queue class all have a worst case *O(1)* running time.

1. Given the following sequence of integers:
2. 3, 9, 2, 15, -5, 18, 7, 5, 8
   1. What is the average number of comparisons for a successful search assuming all entries are searched with equal probability? Show your work.
   2. Suppose the search probabilities for the elements of this list are, respectively:
   3. 0.1, 0.3, 0.05, 0.2, 0.05, 0.1, 0.05, 0.1, 0.05

What is the average number of comparisons for successful search with these search probabilities? Show your work.

* 1. Rearrange the list entries in a way that would result in the lowest number of comparisons on the average for successful search, given the above probabilities of search. What is this lowest number of comparisons? Show your work.

1. An adaptive algorithm to lower average match time for sequential search is to move an item by one spot toward the front every time it is matched. (Unless it is already at the front, in which case nothing is done on a match.) Complete the following modified sequential search on a linked list with this move-toward-front adaption. Assume a generic Node class with data and next fields.
2. public class LinkedList<T> {
3. private Node<T> front;
4. int size;
5. ...
6. // moves the target one place toward the front
7. // doesn't do anything if target is in the first node
8. // returns true if target found, false otherwise
9. public boolean moveTowardFront(T target) {
10. // COMPLETE THIS METHOD
11. }
12. }
13. \* (Done in lecture)

An alternative algorithm for searching on a sorted array of size *n* works as follows. It divides the array into *m* contiguous blocks each of size *s*. (Assume that *s* divides into *n* without remainder).

Here is the algorithm to search for a key *k* in sorted array *A*.

Compare *k* with the last entry in the first block, i.e. *A[s-1]*

If there is match, then stop with success

Otherwise, check if *k* < *A[s-1]*

If so, perform a sequential search on the block of entries from

*A[0]* to *A[s-2]*. If there is a match, stop with success,

otherwise stop with failure.

If *k* is not < *A[s-1]* then continue the process by

repeating the above on the second block, and so on.

* 1. a) What is the **worst** case number of searches for success?
  2. b) What is the **average** case number of searches for success?

1. **WORK OUT THE SOLUTION TO THIS PROBLEM, AND TURN IT IN AT RECITATION**

A variant of binary search, called *lazy* binary search, works as described in the following algorithm, where t is the target to search, and n is the size of the array:

left <-- 0

right <-- n-1

while (left < right) do

mid <-- (left + right)/2

if (t > A[mid]) then

left <-- mid + 1

else

right <-- mid

endif

endwhile

if (t == A[left]) then

display "found at position", left

else

display "not found"

endif

* 1. Trace this algorithm on the following array, with 46 as the search target:

10 15 25 30 45 46 48 72 76 80 93

* 1. How many comparisons are made by the time a match is found? How does your answer compare with that for regular binary search?
  2. Repeat with 40 as the target. How many comparisons are made until failure is detected? How does your answer compare with the corresponding answer in Problem 1?

1. Implement a recursive solution for the lazy binary search algorithm described in the previous problem. Your solution should have a static public method that accepts an array of integers, and an integer target to search for, and return a boolean for found/not found. The public method should call your static recursive method, which should be declared private.