Mustafa Habeeb

CS 251

Program 02 Analysis

file: ServiceQueueSlow.h

vector function runtimes.

source: www.cplusplus.com.

function	Runtime
size()	constant
begin()	constant
end()	constant
find(a,b)	Linear (from first to last)
insert()	Linear
erase()	Linear
push_back()	constant

1)

```
89
90      /**
91       * Function: length()
92       * Description:  returns the current number of
93       *    entries in the queue.
94       *
95       * RUNTIME REQUIREMENT:  O(1)
96       */
97      int  length() {
98
99              return the_queue.size();   // placeholder
100
101         }
```

The length() functions objective is to return the entries in the queue.

The runtime is required to be O(1), constant, in which it is met.

The value for the_queue.size() will return the amount that is in the list. The size() function has a runtime of $\Theta(1)$ therefore, the requirements have been met.

2)

```
/**
 * Function: snapshot()
     * param:  buzzers is an integer vector passed by ref
 * Description:  populates buzzers vector with a "snapshot"
     *              of the queue as a sequence of buzzer IDs
     *
 *
 * RUNTIME REQUIREMENT:  O(N)  (where N is the current queue
 *      length).
 */
void snapshot(std::vector<int> &buzzers) {
        buzzers.clear();    // you don't know the history of the
                            //   buzzers vector, so we had better
                            //   clear it first.

        // Note: the vector class over-rides the assignment operator '='
        //   and does an element-by-element copy from the RHS vector (the_vector)
        //   in this case) to the LHS vector (buzzers) in this case).

        buzzers = the_queue;

    }
```

The snapshot() function is to display whatever is in your vector.

The runtime is required to be O(N), linear, in which the runtime is met.

The 'buzzers' is the vector that the elements need to be in. By saying buzzers = the_queue, the vector is going element-by-element as said in the description, therefore the vector is being copied in $\Theta(N)$ time.

3)

```
 *
 * Return:  If the buzzer isn't actually currently in the
 *      queue, the queue is unchanged and false is returned
 *      (indicating failure).  Otherwise, true is returned.
 *
 * RUNTIME REQUIREMENT:  O(1)
 */
bool take_bribe(int buzzer) {
     std::vector<int>::iterator it;

     it = find(the_queue.begin(), the_queue.end(), buzzer);
      if(it == the_queue.end())
          return false;
      else {
          the_queue.erase(it);
          the_queue.insert(the_queue.begin(), buzzer);
          return true;
      }
    }
```

1) 2) 3) 4)

funtion is suppose to take in an integer and find that value through the list.

if the value is found push to the front of the list.

The runtime requirement is suppose to be O(1), however this function does not meet the requirement.

1) the find() function that takes in 3 parameters; first, last, val.
   → The first 2 parameters use .begin() & .end()
      ↳ the .begin() is used to locate the begining of the vector
      ↳ the .end() is used to locate the end of the vector

begin() & end() have constant runtime, O(1).

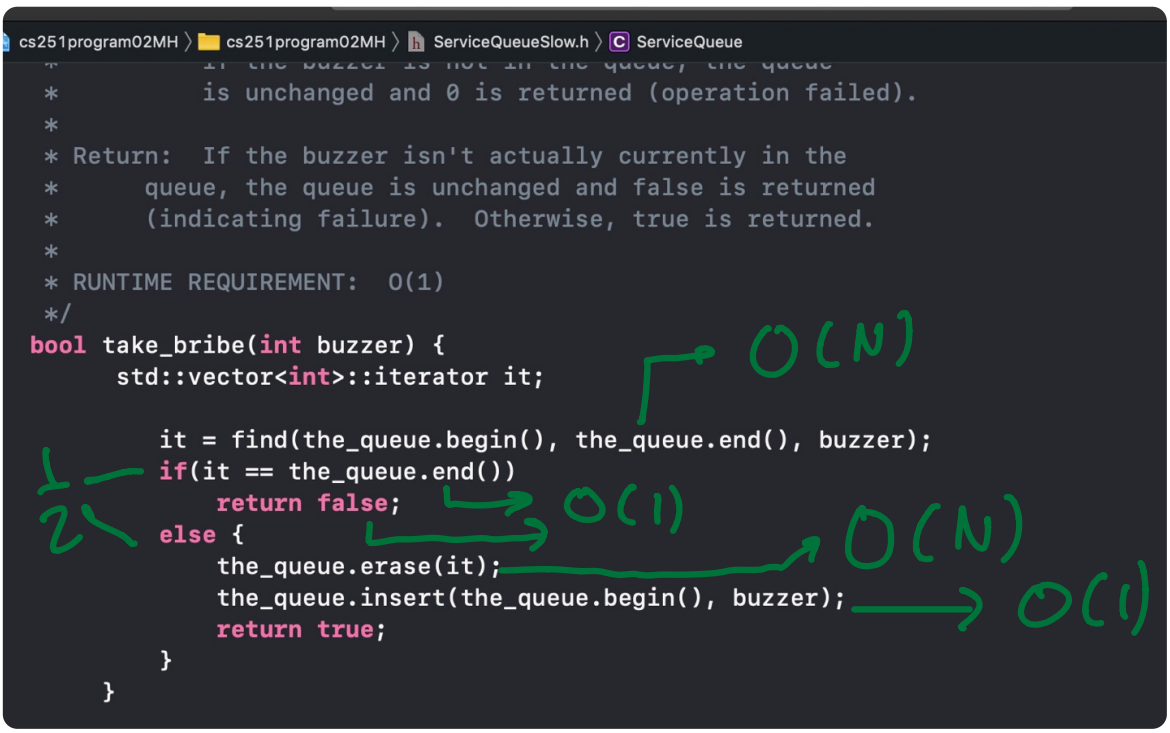
However, the find() has a runtime of O(N) cause it traverses from the begin to the end to find the value.

2) .end() is constant, O(1).

3) .erase() is Linear, O(N) because it has to go through the entire list and delete each value.

4) insert() inserts a value And is considered linear time, O(N) However for this specific instance since the value is being placed in the beginning it is not traversing through the list. Therefore,

for this specific case I am going to say constant time for this statement. O(1)



$O(N * (\frac{1}{2}(1 + N)))$

$O(N (\frac{1}{2} + \frac{N}{2}))$

$O(\frac{N}{2} + \frac{N^2}{2})$

$O(N + N^2) = \Theta(N^2)$

Overall, the take bribe function runs $\Theta(N^2)$.

4)

```
/**
 * function: seat()
 * description:  if the queue is non-empty, it removes the first
 *   entry from (front of queue) and returns the
 *   buzzer ID.
 *   Note that the returned buzzer can now be re-used.
 *
 *   If the queue is empty (nobody to seat), -1 is returned to
 *   indicate this fact.
    *
    * Returns:  buzzer ID of dequeued customer, or -1 if queue is empty.
 *
 * RUNTIME REQUIREMENT:  O(1)
 */
int seat() {
      int buzzer;

      if(the_queue.size()==0)
         return -1;
      else {
         buzzer = the_queue[0];
         the_queue.erase(the_queue.begin(), the_queue.begin()+1);
         buzzer_bucket.push_back(buzzer);
         return buzzer;
      }
   }
```

O(1)

$\frac{1}{2}$

O(1)

O(N)

O(1)

$$O\left(\frac{1}{2}(1+N)\right)$$

$$O\left(\frac{1}{2}+\frac{N}{2}\right) = \boxed{\Theta(N)}$$

runtime for this function is $\Theta(N)$.

6)

```
/**
 * function: kick_out()
 *
 * description:  Some times buzzer holders cause trouble and
 *      a bouncer needs to take back their buzzer and
 *      tell them to get lost.
 *
 *      Specifially:
 *
 *      If the buzzer given by the 2nd parameter is
 *      in the queue, the buzzer is removed (and the
 *      buzzer can now be re-used) and 1 (one) is
 *      returned (indicating success).
 *
 * Return:  If the buzzer isn't actually currently in the
 *      queue, the queue is unchanged and false is returned
 *      (indicating failure).  Otherwise, true is returned.
 *
 * RUNTIME REQUIREMENT:  O(1)
 */
bool kick_out(int buzzer) {
     std::vector<int>::iterator it;

        it = find(the_queue.begin(), the_queue.end(), buzzer);
        if(it == the_queue.end())
            return false;
        else {
            the_queue.erase(it);
            buzzer_bucket.push_back(buzzer);
            return true;
        }
   }
```

O(N)

1/2

O(1)

O(N)

O(1)

$$O\left(N + \frac{1}{2}(1+N)\right)$$

$$O\left(N + \frac{1}{2} + \frac{N}{2}\right) =$$

$$\Theta(N)$$

the runtime is $\Theta(N)$

5)

cs251program02MH 〉 cs251program02MH 〉 ServiceQueueSlow.h 〉 ServiceQueue

```
 *
 * RUNTIME REQUIREMENT:  O(1)  ON AVERAGE or "AMORTIZED"
 *          In other words, if there have been M calls to
 *      give_buzzer, the total time taken for those
 *      M calls is O(M).
 *
 *      An individual call may therefore not be O(1) so long
 *      as when taken as a whole they average constant time.
 *
 */
int  give_buzzer() {
     int b;

         // take top reusable buzzer if possible
         if(buzzer_bucket.size() != 0) {
             b = buzzer_bucket.back();
             buzzer_bucket.pop_back();
         }
         // otherwise, queue must contain exactly buzzers
         //    0..the_queue.size()-1
         // and therefore, the next smallest buzzer must be
         //    the_queue.size()
         else {
             b = the_queue.size();
         }
         the_queue.push_back(b);
         return b;
    }
```

O(1)

O(1)

1/2

1/2

O(1)

O(1)

$$O\left(\frac{1}{2}(1+1)\right) =$$

$$O\left(\frac{1}{2}+\frac{1}{2}\right) =$$

$$O(1) = \boxed{\Theta(1)}$$