Week 7: Asymptotic Analysis, Circularly Linked Lists, and Implementing a Queue with a Circularly Linked List

CSCI 2100 Data Structures

Fall 2017

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

What is the goal of analysis of algorithms?
- To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)

What do we mean by running time analysis?
- Determine how running time increases as the size of the problem increases.
Input Size

- Input size (number of elements in the input)
  - size of an array
  - polynomial degree
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph
Types of Analysis

- **Worst case**
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are

- **Best case**
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

- **Average case**
  - Provides a prediction about the running time
  - Assumes that the input is random

Lower Bound $\leq$ Running Time $\leq$ Upper Bound
How do we compare algorithms?

- We need to define a number of **objective measures**.

(1) Compare execution times?

   **Not good**: times are specific to a particular computer!!

(2) Count the number of statements executed?

   **Not good**: number of statements vary with the programming language as well as the style of the individual programmer.
Ideal Solution

- Express running time as a function of the input size $n$ (i.e., $f(n)$).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

CSCI 2100
Example

- Associate a "cost" with each statement.
- Find the "total cost" by finding the total number of times each statement is executed.

**Algorithm 1**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[0] = 0;</td>
<td>(c_1)</td>
</tr>
<tr>
<td>arr[1] = 0;</td>
<td>(c_1)</td>
</tr>
<tr>
<td>arr[2] = 0;</td>
<td>(c_1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>arr[N-1] = 0;</td>
<td>(c_1)</td>
</tr>
</tbody>
</table>

\[c_1 + c_1 + \ldots + c_1 = c_1 \times N\]

**Algorithm 2**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0; i&lt;N; i++)</td>
<td>(c_2)</td>
</tr>
<tr>
<td>arr[i] = 0;</td>
<td>(c_1)</td>
</tr>
</tbody>
</table>

\[(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2\]
Another Example

**Algorithm 3**

```
sum = 0;
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        sum += arr[i][j];
```

**Cost**

\[
c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2
\]
To compare two algorithms with running times $f(n)$ and $g(n)$, we need a rough measure that characterizes how fast each function grows.

*Hint:* use rate of growth

Compare functions in the limit, that is, asymptotically! (i.e., for large values of $n$)
Rate of Growth

- Consider the example of buying *elephants* and *goldfish*:
  
  \[ \text{Cost: } \text{cost}_\text{of}_\text{elephants} + \text{cost}_\text{of}_\text{goldfish} \]
  
  \[ \text{Cost} \sim \text{cost}_\text{of}_\text{elephants} \text{ (approximation)} \]

- The low order terms in a function are relatively insignificant for large \( n \)

\[ n^4 + 100n^2 + 10n + 50 \sim n^4 \]

\[ i.e., \ we \ say \ that \ n^4 + 100n^2 + 10n + 50 \text{ and } n^4 \text{ have the same rate of growth} \]
Asymptotic Notation

- **O notation**: asymptotic “less than”:
  - $f(n) = O(g(n))$ implies: $f(n) \leq g(n)$

- **Ω notation**: asymptotic “greater than”:
  - $f(n) = \Omega(g(n))$ implies: $f(n) \geq g(n)$

- **Θ notation**: asymptotic “equality”:
  - $f(n) = \Theta(g(n))$ implies: $f(n) = g(n)$
We say \( f_A(n) = 30n + 8 \) is order \( n \), or \( O(n) \). It is, at most, roughly proportional to \( n \).

\( f_B(n) = n^2 + 1 \) is order \( n^2 \), or \( O(n^2) \). It is, at most, roughly proportional to \( n^2 \).

In general, any \( O(n^2) \) function is faster-growing than any \( O(n) \) function.
On a graph, as you go to the right, a faster growing function eventually becomes larger...

\[ f_A(n) = 30n + 8 \]

\[ f_B(n) = n^2 + 1 \]
More Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$
## Back to Our Example

### Algorithm 1

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[0] = 0;</td>
</tr>
<tr>
<td>arr[1] = 0;</td>
</tr>
<tr>
<td>arr[2] = 0;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>arr[N-1] = 0;</td>
</tr>
</tbody>
</table>

\[
c_1 + c_1 + ... + c_1 = c_1 \times N
\]

### Algorithm 2

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0; i&lt;N; i++)</td>
</tr>
<tr>
<td>arr[i] = 0;</td>
</tr>
</tbody>
</table>

\[
(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2
\]

- Both algorithms are of the same order: $O(N)$
Example (cont’d)

Algorithm 3

```
sum = 0;
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        sum += arr[i][j];
```

Cost

\[ c_1 + c_2 \cdot (N+1) + c_2 \cdot N \cdot (N+1) + c_3 \cdot N^2 = O(N^2) \]
Asymptotic notations

- **$O$-notation**

\[
O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}.
\]

$g(n)$ is an \textit{asymptotic upper bound} for $f(n)$.
Big-O Visualization

\( O(g(n)) \) is the set of functions with smaller or same order of growth as \( g(n) \)
Examples

- $2n^2 = O(n^3)$:
  \[ 2n^2 \leq cn^3 \implies 2 \leq cn \implies c = 1 \text{ and } n_0 = 2 \]

- $2n^2 = O(n^2)$:
  \[ 2n^2 \leq cn^2 \implies c \geq 2 \implies c = 2 \text{ and } n_0 = 1 \]

- Then, $2n^2 = O(n^3)$ or $O(n^2)$?
More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0: 30n+8 \leq cn$, $\forall n>n_0$.
    - Let $c=31$, $n_0=8$. Assume $n>n_0=8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$. 
Big-O example, graphically

- Note $30n+8$ isn’t less than $n$ anywhere ($n>0$).
- It isn’t even less than $31n$ everywhere.
- But it is less than $31n$ everywhere to the right of $n=8$.  

\[ cn = 31n \]
\[ n > n_0 = 8 \]
\[ 30n+8 \in O(n) \]
No Uniqueness

- There is no unique set of values for $n_0$ and $c$ in proving the asymptotic bounds.

- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2$
    - for all $n \geq 5$
      - $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$
    - for all $n \geq 1$
      - $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants $c$ and $n_0$ that satisfy the asymptotic notation relation.
Asymptotic notations (cont.)

- **Ω - notation**

  \[ \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \]

  \[ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} . \]

  \[ \Omega(g(n)) \text{ is the set of functions with larger or same order of growth as } g(n) \]

  \[ g(n) \text{ is an asymptotic lower bound for } f(n). \]
Examples

- $5n^2 = \Omega(n)$

  $\exists c, n_0$ such that: $0 \leq cn \leq 5n^2 \implies cn \leq 5n^2 \implies c = 1$ and $n_0 = 1$

- $100n + 5 \neq \Omega(n^2)$

  $\exists c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$

  $100n + 5 \leq 100n + 5n (\forall n \geq 1) = 105n$

  $cn^2 \leq 105n \implies n(cn - 105) \leq 0$

  Since $n$ is positive $\implies cn - 105 \leq 0 \implies n \leq 105/c$

  $\implies$ contradiction: $n$ cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$
Asymptotic notations (cont.)

- **Θ-notation**

\[ \Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} . \]

\( \Theta(g(n)) \) is the set of functions with the same order of growth as \( g(n) \).

\( g(n) \) is an **asymptotically tight bound** for \( f(n) \).
Examples

- \( n^2/2 - n/2 = \Theta(n^2) \)
  
  - \( \frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \ \forall n \geq 0 \quad \Rightarrow \quad c_2 = \frac{1}{2} \)
  
  - \( \frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n \cdot \frac{1}{2} n \ (\forall n \geq 2) = \frac{1}{4} n^2 \quad \Rightarrow \quad c_1 = \frac{1}{4} \)

- \( n \neq \Theta(n^2) \): \( c_1 n^2 \leq n \leq c_2 n^2 \)

\( \Rightarrow \) only holds for: \( n \leq 1/c_1 \)
Examples

- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$
  
  $\Rightarrow$ only holds for: $n \leq c_2 /6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

  $\Rightarrow c_2 \geq n / \log n, \forall n \geq n_0$ - impossible
Subset relations between order-of-growth sets.
Why is Big O taught instead of Big Theta?

- A tight bound is sometimes harder to compute", and often useless too.
- Normally, even when people talk about $O(g(n))$ they actually mean $\Theta(g(n))$
- Then, $2n^2 = O(n^3)$ or $O(n^2)$?
- Think of it this way.

Suppose you have 10 dollars in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars." Your statement is absolutely true, though not terribly precise. So, we usually use a precise term in the Big O notation.
Big-Oh Rules

● Use the smallest possible class of functions
  ▪ Say “$2n$ is $O(n)$” instead of “$2n$ is $O(n^2)$”

● Use the simplest expression of the class
  ▪ Say “$3n + 5$ is $O(n)$” instead of “$3n + 5$ is $O(3n)$”
Common orders of magnitude
## Table 1.4 Execution times for algorithms with the given time complexities

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n) = \log n$</th>
<th>$f(n) = n$</th>
<th>$f(n) = n \log n$</th>
<th>$f(n) = n^2$</th>
<th>$f(n) = n^3$</th>
<th>$f(n) = 2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003 $\mu$s*</td>
<td>0.01 $\mu$s</td>
<td>0.033 $\mu$s</td>
<td>0.1 $\mu$s</td>
<td>1 $\mu$s</td>
<td>1 $\mu$s</td>
</tr>
<tr>
<td>20</td>
<td>0.004 $\mu$s</td>
<td>0.02 $\mu$s</td>
<td>0.086 $\mu$s</td>
<td>0.4 $\mu$s</td>
<td>8 $\mu$s</td>
<td>1 ms$^\dagger$</td>
</tr>
<tr>
<td>30</td>
<td>0.005 $\mu$s</td>
<td>0.03 $\mu$s</td>
<td>0.147 $\mu$s</td>
<td>0.9 $\mu$s</td>
<td>27 $\mu$s</td>
<td>1 s</td>
</tr>
<tr>
<td>40</td>
<td>0.005 $\mu$s</td>
<td>0.04 $\mu$s</td>
<td>0.213 $\mu$s</td>
<td>1.6 $\mu$s</td>
<td>64 $\mu$s</td>
<td>18.3 min</td>
</tr>
<tr>
<td>50</td>
<td>0.005 $\mu$s</td>
<td>0.05 $\mu$s</td>
<td>0.282 $\mu$s</td>
<td>2.5 $\mu$s</td>
<td>64 $\mu$s</td>
<td>13 days</td>
</tr>
<tr>
<td>$10^2$</td>
<td>0.007 $\mu$s</td>
<td>0.10 $\mu$s</td>
<td>0.664 $\mu$s</td>
<td>10 $\mu$s</td>
<td>1 ms</td>
<td>4 $\times 10^{15}$ years</td>
</tr>
<tr>
<td>$10^3$</td>
<td>0.010 $\mu$s</td>
<td>1.00 $\mu$s</td>
<td>9.966 $\mu$s</td>
<td>1 ms</td>
<td>1 s</td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.013 $\mu$s</td>
<td>0 $\mu$s</td>
<td>130 $\mu$s</td>
<td>100 ms</td>
<td>16.7 min</td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td>0.017 $\mu$s</td>
<td>0.10 ms</td>
<td>1.67 ms</td>
<td>10 s</td>
<td>11.6 days</td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.020 $\mu$s</td>
<td>1 ms</td>
<td>19.93 ms</td>
<td>16.7 min</td>
<td>31.7 years</td>
<td></td>
</tr>
<tr>
<td>$10^7$</td>
<td>0.023 $\mu$s</td>
<td>0.01 s</td>
<td>0.23 s</td>
<td>1.16 days</td>
<td>31709 years</td>
<td></td>
</tr>
<tr>
<td>$10^8$</td>
<td>0.027 $\mu$s</td>
<td>0.10 s</td>
<td>2.66 s</td>
<td>115.7 days</td>
<td>3.17 $\times 10^7$ years</td>
<td></td>
</tr>
<tr>
<td>$10^9$</td>
<td>0.030 $\mu$s</td>
<td>1 s</td>
<td>29.90 s</td>
<td>31.7 years</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1 $\mu$s = $10^{-6}$ second.

$^\dagger$1 ms = $10^{-3}$ second.
Circularly Linked Lists

- A **circularly linked list** has the same kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle.

- If we traverse the nodes of a circularly linked list from any node by following **next** pointers, we eventually visit all the nodes and cycle back to the node from which we started.

- Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the **cursor**. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.
Circularly Linked Lists

- There are two positions of particular interest in a circular list.
- The first is the element that is referenced by the cursor, which is called the **back**, and the element immediately following this in the circular order, which is called the **front**.
- Although it may seem odd to think of a circular list as having a front and a back, observe that, if we were to cut the link between the node referenced by the cursor and this node’s immediate successor, the result would be a singly linked list from the front node to the back node.
Circularly Linked Lists

- We define the following functions for a circularly linked list:
  - `front()`: Return the element referenced by the cursor; an error results if the list is empty.
  - `back()`: Return the element immediately after the cursor; an error results if the list is empty.
  - `advance()`: Advance the cursor to the next node in the list.
  - `add(e)`: Insert a new node with element e immediately after the cursor; if the list is empty, then this node becomes the cursor and its next pointer points to itself.
  - `remove()`: Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to null.
CircleList.h

```cpp
#ifndef CIRCLELIST_H
#define CIRCLELIST_H

template<typename Object>
class CircleList {

public:
    CircleList();                   // constructor
    ~CircleList();                  // destructor
    bool empty() const;             // is list empty?
    const Object& front() const;    // return front element
    const Object& back() const;     // return back element
    void advance();                 // advance cursor
    void add(const Object& e);      // add after cursor
    void remove();                  // remove after cursor

private:
    struct CNode {                  // node struct
        Object elem;
        CNode* next;
    };
    CNode* cursor;                 // head of list

#include "CircleList.cpp"

#endif
```
Code Fragment 3.30 presents the class’s constructor and destructor. The constructor generates an empty list by setting the cursor to NULL. The destructor iteratively removes nodes until the list is empty. We exploit the fact that the member function remove (given below) deletes the node that it removes.

```
CircleList::CircleList() : cursor(NULL) { } // constructor

CircleList::~CircleList() { while (!empty()) remove(); } // destructor
```

**Code Fragment 3.30**: The constructor and destructor.
We present a number of simple member functions in Code Fragment 3.31. To determine whether the list is empty, we test whether the cursor is NULL. The advance function advances the cursor to the next element.

```cpp
bool CircleList::empty() const
{ return cursor == NULL; } // is list empty?

const Elem& CircleList::back() const
{ return cursor->elem; } // element at cursor

const Elem& CircleList::front() const
{ return cursor->next->elem; } // element following cursor

void CircleList::advance()
{ cursor = cursor->next; } // advance cursor
```

**Code Fragment 3.31**: Simple member functions.
Next, let us consider insertion. Recall that insertions to the circularly linked list occur after the cursor. We begin by creating a new node and initializing its data member. If the list is empty, we create a new node that points to itself. We then direct the cursor to point to this element. Otherwise, we link the new node just after the cursor. The code is presented in Code Fragment 3.32.

```cpp
void CircleList::add(const Elem& e) {
    CNode* v = new CNode;
    v->elem = e;
    if (cursor == NULL) {
        v->next = v;
        cursor = v;
    } else {
        v->next = cursor->next;
        cursor->next = v;
    }
    // add after cursor
    // create a new node
    // list is empty?
    // v points to itself
    // cursor points to v
    // list is nonempty?
    // link in v after cursor
}
```

Code Fragment 3.32: Inserting a node just after the cursor of a circularly linked list.
Finally, we consider removal. We assume that the user has checked that the list is nonempty before invoking this function. (A more careful implementation would throw an exception if the list is empty.) There are two cases. If this is the last node of the list (which can be tested by checking that the node to be removed points to itself) we set the cursor to NULL. Otherwise, we link the cursor’s next pointer to skip over the removed node. We then delete the node. The code is presented in Code Fragment 3.33.

```cpp
void CircleList::remove() {
    CNode* old = cursor->next;
    if (old == cursor) // the node being removed
        cursor = NULL; // removing the only node?
    else
        cursor->next = old->next; // link out the old node
    delete old; // delete the old node
}
```

*Code Fragment 3.33: Removing the node following the cursor.*
```cpp
#include "CircleList.h"
#include <iostream>
using namespace std;

int main()
{
    CircleList<string> playList;

    if (playList.empty())
        cout << "You successfully made an empty list!" << endl;

    playList.add("Stayin Alive"); // [Stayin Alive*]
    cout << playList.front() << endl; // print Stayin Alive
    cout << playList.back() << endl; // print Stayin Alive

    playList.add("Le Freak"); // [Le Freak, Stayin Alive*]
    cout << playList.front() << endl; // print Le Freak
    cout << playList.back() << endl; // print Stayin Alive

    playList.add("Jive Talkin"); // [Jive Talkin, Le Freak, Stayin Alive*]
    cout << playList.front() << endl; // print Jive Talkin
    cout << playList.back() << endl; // print Stayin Alive

    playList.advance(); // [Le Freak, Stayin Alive, Jive Talkin*]
    cout << playList.front() << endl; // print Le Freak
    cout << playList.back() << endl; // print Jive Talkin

    playList.advance(); // [Stayin Alive, Jive Talkin, Le Freak*]
    cout << playList.front() << endl; // print Stayin Alive
    cout << playList.back() << endl; // print Le Freak

    playList.remove(); // [Jive Talkin, Le Freak*]
    cout << playList.front() << endl; // print Jive Talkin
    cout << playList.back() << endl; // print Le Freak

    playList.add("Disco Inferno"); // [Disco Inferno, Jive Talkin, Le Freak*]
    cout << playList.front() << endl; // print Disco Inferno
    cout << playList.back() << endl; // print Le Freak

    return 0;
}
```
In order to implement the queue operation **enqueue**, we first invoke the function **add**, which inserts a new element just after the cursor, that is, just after the rear (or back) of the queue. We then invoke **advance**, which advances the cursor to this new element, thus making the new node the rear of the queue. The process is illustrated in Figure 5.5.
In order to implement the queue operation \texttt{dequeue}, we invoke the function \texttt{remove}, thus removing the node just after the cursor, that is, the front of the queue. The process is illustrated in Figure 5.6.

\textbf{Figure 5.6}: Dequeueing an element (in this case “LAX”) from the front queue represented as a circularly linked list: (a) before the operation; (b) after removing the node immediately following the cursor.
```cpp
#ifndef LINKED_QUEUE_H
#define LINKED_QUEUE_H

#include "CircleList.h"

template<typename Object>
class LinkedQueue {

public:
    LinkedQueue(); // constructor
    int size() const; // number of items in stack
    bool empty() const; // is the queue empty?
    const Object& front() const; // get the front element
    void enqueue(const Object& e); // push element onto stack
    void dequeue(); // pop the stack
    void printQ(); // print elements of queue

private:
    CircleList<Object> C; // linked list of elements
    int n; // number of elements

};

#include "LinkedQueue.cpp"
#endif
```
LinkedQueue class

LinkedQueue::LinkedQueue()
 : C(), n(0) {} // constructor

int LinkedQueue::size() const
 { return n; } // number of items in the queue

bool LinkedQueue::empty() const
 { return n == 0; } // is the queue empty?

const Elem& LinkedQueue::front() const throw(QueueEmpty) {
 if (empty())
   throw QueueEmpty("front of empty queue");
 return C.front(); // list front is queue front
}

Code Fragment 5.19: Constructor and accessor functions for the LinkedQueue class.
The definition of the queue operations, enqueue and dequeue are presented in Code Fragment 5.20. Recall that enqueuing involves invoking the add function to insert the new item immediately following the cursor and then advancing the cursor.

Before dequeuing, we check whether the queue is empty, and, if so, we throw an exception. Otherwise, dequeuing involves removing the element that immediately follows the cursor. In either case, we update the number of elements in the queue.

```cpp
void LinkedQueue::enqueue(const Elem& e) {
    C.add(e);
    C.advance();
    n++;
}

void LinkedQueue::dequeue() throw(QueueEmpty) {
    if (empty())
        throw QueueEmpty("dequeue of empty queue");
    C.remove();
    n--;}
```

**Code Fragment 5.20:** The enqueue and dequeue functions for LinkedQueue.