Lecture 7 – Review of Complexity Analysis using the Big O notation
+ Comparing List ADT class implementations
Last Lecture

- We saw how to ...
  - Create linked list
  - Perform operations on a linked list
  - Create a Node class
  - Implement a List ADT class:
    - Using an array
    - Using a linked-list
At the end of this lecture, a student will be able to:

- Complexity Analysis:
  - Analyze the best, worst, and average case running time of various ADT operations implemented by various data structures

- Abstract Data Types (ADTs):
  - Given a set of requirements, select the most appropriate data collection ADT class, taking into account its time and space efficiency
Implementation - Node Class

(/^
 * Node.h
 *
 * Class Definition: Node of a singly linked list
 * in which the data is of "int" data type.
 *
 * Created on:
 * Author:
 */

#pragma once // is shorthand for the 2 #include guards we've seen before:
// ifndef _NODE and define _NODE
#pragma once

class Node
{
public:
    // Public attributes - Why are the attributes public?
    int data;    // The data in the node
    Node* next;  // Pointer to next node

    // Constructors and destructor
    Node();
    Node(int theData);
    Node(int theData, Node* theNextNode);
}; // end Node
/*
 * Node.cpp
 *
 * Class Definition: Node of a singly linked list
 * in which the data is of "int" data type.
 *
 * Created on:
 * Author:
 */

Node::Node()
{
    data = 0;
    next = NULL;
}

Node::Node(int theData)
{
    data = theData;
    next = NULL;
}

Node::Node(int theData, Node* theNextNode)
{
    data = theData;
    next = theNextNode;
}

// end Node.cpp
List ADT class

/*
 * List.h
 *
 * Class Description: A list ADT - inspired from List from our textbook.
 * Class Invariant: List position of elements start at 1 (not 0).
 *
 * Note 1: This is a link-based implementation of a data collection List ADT class
 * as its underlying data structure is a singly-headed singly-linked list (SHSL).
 *
 * Note 2: The public methods have the same declaration as the methods of the array-
 * based implementation of List class.
 *
 * Inspired on: January 2019
 * Author: AL
 */

#pragma once
#include "Node.h"

using namespace std;
class List {

private:
    int elementCount; // Number of elements currently stored in the list.
    Node *head; // Pointer to the first node in the list
Construct a List object (containing a linked list)

CODE:
```
const int elementCount = 0;
const ListNode* head = NULL;
```
Today’s menu

- Review of the Big-O notation
- Compare the various implementations of our List ADT class:
  - Position-oriented versus value-oriented
  - Array-based implementation versus link-based implementation
Satisfying Client’s Requirements

- When developing a solution, we must, as part of our design (Step 2)
  - Choose the data collection ADT (List, Stack, Queue, etc…) that would best solve the problem along with its most efficient implementation

- In order to make these choices …
  - We compare **time efficiency** and/or **space efficiency** of various implementations (i.e., data structure + algorithms) of the chosen ADT
Time/Space Efficiency

- Time efficiency of an algorithm is a measure of "time" the algorithm requires to execute.

- Space efficiency of an algorithm is a measure of "space" (e.g. memory, secondary storage) the algorithm requires to execute.

- Expressed using Big O notation.
Big O Notation

- Introduced by P. Bachmann & Edmund Landau
- Upper bound approximation
- \( n \rightarrow \text{size of data} \)
- This approximation is good enough, since we are interested in the time and space efficiency of algorithms as 
  \[ n \rightarrow \infty \]
  (i.e. as the size of data collection manipulated by algorithms increases to infinity)
- When the efficiency of an algorithm is found to be \( O(g(n)) \), we say that its efficiency is of order \( g(n) \)
- \( g(n) \rightarrow \text{complexity categories} \)
Commonly used Complexity Categories

- \( g(n) \): 1, \( \log n \), \( n \), \( n \log n \), \( n^2 \), \( n^k \), \( k^n \)
- When an algorithm has a time/space efficiency of
  - \( O(1) \) -> we say that its time/space efficiency is constant
  - \( O(\log n) \) -> ... logarithmic
  - \( O(n) \) -> ... linear
  - \( O(n^2) \) -> ... quadratic
  - \( O(n^k) \) -> ... polynomial
  - \( O(k^n) \) -> ... exponential
Comparing $O(g(n))$ as size of data collection $n \rightarrow \infty$

Source: http://jsdo.it/Boony/big-o-notation
How to find the efficiency of an algorithm and express it using the Big O notation?

Consider an algorithm ...

is its execution time (or its space) a function of $n$?

- If YES, then select this function of $n$, i.e., $g(n)$ that expresses its efficiency:
  - $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^k)$ or $O(k^n)$

- If NO, i.e., its execution time (or its space) is not a function of $n$, hence it is independent of $n$, then as $n \to \infty$, it is constant, so its efficiency is $O(1)$
Big O Notation - Arithmetic

- **Sequential statements**
  - \(\text{O}(s_1 + s_2) = \max[\text{O}(s_1), \text{O}(s_2)]\)

- **Repeated statements**
  - \(\text{O}(k \times n) = \text{O}(n) \text{ if } k \text{ is constant}\)

- **Nested statements**
  - \(\text{O}(s_1 \times s_2) = \text{O}(s_1) \times \text{O}(s_2)\)
Algorithm 1

if size of A != size of B
    report error and terminate
if size of A == 0  // size of B also 0
    report B is a scrambled version of A and terminate

for each element of array A (at index i)
    letterMatched = false
for each element of array B (at index j)
    if A[i] == B[j]
        remember match at index j, i.e., B[j] = -1
        letterMatched = true
        stop parsing array B, i.e., go to next letter in A

if !letterMatched, i.e., that letter in A was not found in array B
    report B not a scrambled version of A and terminate

report B is a scrambled version of A
Activity - Time Efficiency Analysis

Algorithm 2

```plaintext
if size of A != size of B
    report error and terminate
if size of A == 0  // size of B also 0
    report B is a scrambled version of A and terminate

get an array isInA[100]
initialize each element of array isInA to 0

for each element of array A (at index i)
    isInA[a[i]] += 1

for each element of array B (at index i)
    if element in array B found in array A, i.e., isInA[b[i]] > 0
        Register the match, i.e., isInA[b[i]] -= 1
    else
        report B not a scrambled version of A and terminate

for each element of array isInA (at index i)
    if isInA[i] != 0
        report B not a scrambled version of A and terminate

report B is a scrambled version of A and terminate
```
Activity – Best/Average/Worst case scenarios
Array-based implementation List ADT

Public interface

FriendsBook (client code)

Insert
Remove
Search

List

Class attributes (private):

Data structure array of Profile objects

Array-based implementation List ADT
Link-based implementation List ADT

FriendsBook (client code)

Insert

Remove

Search

List

Class attributes (private):

Data structure

linked list of Profile objects
Comparing both implementations of the position-oriented List ADT using Big O notation

- Time efficiency of their operations (worst case scenario):

<table>
<thead>
<tr>
<th>Operations</th>
<th>array-based</th>
<th>link-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>getElementCount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeAll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retrieve</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing both implementations of the value-oriented List ADT using Big O notation

- Time efficiency of their operations (worst case scenario):

<table>
<thead>
<tr>
<th>Operations</th>
<th>array-based</th>
<th>link-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>getElementCount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeAll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retrieve (search)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Learning Check

- We can now ...
  - Determine the time/space efficiency of an algorithm using the Big O notation
  - Compare the various implementations of our List ADT class:
    - Position-oriented versus value-oriented
    - Array-based implementation versus link-based implementation
Next Lecture

- Another linear data collection -> Stack