Week 6: Analysis of Algorithms, Asymptotics, and Singly-Linked Lists

CSCI 2100 Data Structures

Fall 2017

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.
Order Arithmetic

- Big-Oh notation provides a way to compare two functions
- “f(n) is $O(g(n))$” means:

  $f(n)$ is less than or equal to $g(n)$ up to a constant factor for large values of $n$
Categorizing functions

- Big-Oh can be used for categorizing or characterizing functions
- For example, the statements:
  
  \[ 2n + 3 \text{ is } O(n) \text{ and } 5n \text{ is } O(n) \]

  place \( 2n + 3 \) and \( 5n \) in the same category
  - Both functions are less than or equal to \( g(n) = n \), up to a constant factor, for large values of \( n \)
  - If the functions are running times of two algorithms, the algorithms are thus comparable
Definition of Big-Oh

f(n) is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$f(n) \leq c \cdot g(n)$, for $n \geq n_0$

less than or equal up to a constant factor

for large values of $n$
Example

- f(n) = 2n + 5
g(n) = n

- Consider the condition
  \[2n + 5 \leq n\]
  will this condition ever hold? No!

- How about if we tack a constant to n?
  \[2n + 5 \leq 3n\]
  the condition holds for values of n greater than or equal to 5

- This means we can select c = 3 and n₀ = 5
Example

2n+5 is $O(n)$
Use of Big-Oh notation

- Big-Oh allows us to ignore constant factors and lower order (or less dominant) terms

\[ 2n^2 + 5n - 4 \text{ is } O(n^2) \]
Algorithm arrayMax(A,n):

Input: An array A storing n integers.

Output: The maximum element in A.

currentMax ← A[0]

for i ← 1 to n - 1 do
    if currentMax < A[i] then
        currentMax ← A[i]

return currentMax
arrayMax example

- Depending on what operations we decide to count, running time function $f(n) = an + b$
- Regardless, $f(n)$ is $O(n)$
- Or, equivalently, the running time of algorithm arrayMax is $O(n)$
Function categories revisited

- The constant function:  \( f(n) = 1 \)
- The linear function:  \( f(n) = n \)
- The quadratic function:  \( f(n) = n^2 \)
- The cubic function:  \( f(n) = n^3 \)
- The exponential function:  \( f(n) = 2^n \)
- The logarithm function:  \( f(n) = \log n \)
- The \( n \log n \) function:  \( f(n) = n \log n \)
Comparing function categories

- Linear \((n)\) is better than quadratic \((n^2)\) which is better than exponential \((2^n)\)

- Are there any function categories better than linear? Yes!
  - Constant \((1)\)
  - Logarithmic \((\log n)\)

- “Better” means resulting values are smaller (slower growth rates)
Functions by increasing growth rate

- The constant function: $f(n) = 1$
- The logarithm function: $f(n) = \log n$
- The linear function: $f(n) = n$
- The $n \log n$ function: $f(n) = n \log n$
- The quadratic function: $f(n) = n^2$
- The cubic function: $f(n) = n^3$
- The exponential function: $f(n) = 2^n$
Big-Oh in this course

- For this course, you will be expected to assess the running time of an algorithm and classify it under one of the categories, using Big-Oh notation.

- You should be able to recognize, for instance, that, most of the time (not always):
  - Algorithms with single loops are $O(n)$
  - Algorithms with double-nested loops are $O(n^2)$
Big-Oh as an upper bound

- The statement \( f(n) \) is \( O(g(n)) \) indicates that \( g(n) \) is an upper bound for \( f(n) \).

- Which means it is also correct to make statements like:
  - \( 3n+5 \) is \( O(n^2) \)
  - \( 3n+5 \) is \( O(2^n) \)
  - \( 3n+5 \) is \( O(5n + \log n - 2) \)
  - But the statement \( 3n+5 \) is \( O(n) \) is the “tightest” statement one can make.
Relatives of Big-Oh

- **Big Omega \( \Omega \):** lower bound
- **Big Theta \( \Theta \):** the function is both a lower bound and an upper bound
- For this course, only Big-Oh notation will be used for algorithm analysis
Summary

- Big-Oh notation compares functions
- It provides for a mechanism to precisely characterize functions according to typical function categories
- Running time functions of algorithms are assessed more precisely using Big-Oh
  - The notation allows us to ignore constants and lower order terms
Singly linked lists are linear collections of data very much like an array, but instead of the data being arranged in one continuous block, each piece of data is linked to the next one in the chain using pointers coupled with the data.
Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node
Singly Linked List

- A node reference another node, the next reference inside a node is a *link* or *pointer* to another node.
- Moving from one node to another by following a next reference is known as *link hopping* or *pointer hopping*.
- The first and last node of a linked list usually are called the *head* and *tail* of the list, respectively.
- Thus, we can link hop through the list starting at the head and ending at the tail.
- We can identify the tail as the node having a null next reference, which indicates the end of the list.
- A linked list defined in this way is known as a *singly linked list*. 
Singly Linked List

- Like an array,
  - a singly linked list keeps its elements in a certain order.
  - This order is determined by the chain of next links going from each node to its successor in the list.

- Unlike an array,
  - a singly linked list does not have a predetermined fixed size,
  - and uses space proportional to the number of its elements.
  - Likewise, we do not keep track of any index numbers for the nodes in a linked list.
  - So we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list.
### Possible List API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean empty()</td>
<td>// is list empty?</td>
</tr>
<tr>
<td>Elem front()</td>
<td>// return front element</td>
</tr>
<tr>
<td>addFront (Elem)</td>
<td>// add to front of list</td>
</tr>
<tr>
<td>removeFront()</td>
<td>// remove front item list</td>
</tr>
<tr>
<td>addBack (Elem)</td>
<td>// add to back of list</td>
</tr>
<tr>
<td>removeBack()</td>
<td>// remove back item list</td>
</tr>
<tr>
<td>And more...</td>
<td></td>
</tr>
</tbody>
</table>
Times for Some Operations

addFront $O(1)$

**Figure 3.10:** Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) creation of a new node; (c) after the insertion.
Times for Some Operations

removeFront  $O(1)$

**Figure 3.11:** Removal of an element at the head of a singly linked list: (a) before the removal; (b) “linking out” the old new node; (c) after the removal.
Implementing a Generic Singly Linked List

- Textbook Chap. 3.2.4 (SLinkedList.h)

```cpp
#ifndef SLINKEDLIST_H
#define SLINKEDLIST_H

#include <iostream>
#include <stdexcept>

using namespace std;

template <typename Object>
class SLinkedList {
public:
    SLinkedList(); // empty list constructor
    ~SLinkedList(); // destructor
    bool empty() const; // is list empty?
    const Object& front() const; // return front element
    void addFront(const Object& e); // add to front of list
    void removeFront(); // remove front item list

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

#include "SLinkedList.cpp"
#endif
```
C style **structure** is useful for storing an aggregation of elements.

Unlike an array, the elements (known as members) of a structure may be of different types.

```c
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string name;                   // passenger name
    MealType mealPref;            // meal preference
    bool isFreqFlyer;             // in the frequent flyer program?
    string freqFlyerNo;           // the passenger’s freq. flyer number
};
```

This defines a new type called Passenger. Let us declare and initialize a variable named “pass” of this type.

```c
Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };```

The individual members of the structure are accessed using the **member selection operator**, which has the form `struct.name.member`. For example, we could change some of the above fields as follows.

```c
pass.name = "Pocahontas";       // change name
pass.mealPref = REGULAR;        // change meal preference
```
Implementing a Generic Singly Linked List

- Textbook Chap. 3.2.4 (SLinkedList.cpp)

```cpp
template <typename E>
SLinkedList<E>::SLinkedList()
    : head(NULL) { } // constructor

template <typename E>
bool SLinkedList<E>::empty() const
    { return head == NULL; } // is list empty?

template <typename E>
const E& SLinkedList<E>::front() const
    { return head->elem; } // return front element

template <typename E>
SLinkedList<E>::~SLinkedList()
    { while (!empty()) removeFront(); } // destructor

template <typename E>
void SLinkedList<E>::addFront(const E& e) { // add to front of list
    SNode<E>* v = new SNode<E>;
    v->elem = e;
    v->next = head;
    head = v;
} // create new node

// store data

// head now follows v
// v is now the head

template <typename E>
void SLinkedList<E>::removeFront() { // remove front item
    SNode<E>* old = head;
    head = old->next;
    delete old; // save current head
    // skip over old head
    // delete the old head
}
```

Code Fragment 3.20: Other member functions for a generic singly linked list.
Implementing a Generic Singly Linked List

- Textbook Chap. 3.2.4 (testSLL.cpp)

```cpp
#include "SLinkedList.h"
#include <iostream>
using namespace std;

int main() {
    SLinkedList<int> numlist;
    if (numlist.empty())
        cout << "You successfully made an empty list!" << endl;
    numlist.addFront(2);
    cout << numlist.front() << endl; // print: 2
    numlist.addFront(7);
    cout << numlist.front() << endl; // print 7
    numlist.removeFront();
    cout << numlist.front() << endl; // print 2
    SLinkedList<char> charlist;
    charlist.addFront('b');
    charlist.addFront('d');
    cout << charlist.front() << endl; // print 'd'
    charlist.removeFront();
    cout << charlist.front() << endl; // print 'b'
    charlist.removeFront();
    cout << charlist.front() << endl; // print ??
    return 0;
}
```
5.1.5 Implementing a Stack with a Generic Linked List

In this section, we show how to implement the stack ADT using a singly linked list. Our approach is to use the generic singly linked list, called SLinkedList, which was presented earlier in Section 3.2.4. The definition of our stack, called LinkedStack, is presented in Code Fragment 5.7.

To avoid the syntactic messiness inherent in C++ templated classes, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the stack’s elements, called Elem. In this example, we define Elem to be of type string. We leave the task of producing a truly generic implementation as an exercise. (See Exercise R-5.7.)

```cpp
typedef string Elem; // stack element type
class LinkedStack { // stack as a linked list
public:
  LinkedStack(); // constructor
  int size() const; // number of items in the stack
  bool empty() const; // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e); // push element onto stack
  void pop() throw(StackEmpty); // pop the stack
private:
  SLinkedList<Elem> S; // linked list of elements
  int n; // number of elements
};
```

**Code Fragment 5.7:** The class LinkedStack, a linked list implementation of a stack.
LinkedStack.h

$ git fetch origin
$ git checkout –b week6 origin/week6

```cpp
#ifndef LINKED_STACK_H
#define LINKED_STACK_H

#include "SLinkedList.h"

template <typename Object>
class LinkedStack {

public:
    LinkedStack(); // empty list constructor
    int size() const; // number of items in stack
    bool empty() const; // is the stack empty?
    const Object& top() const; // get the top element
    void push(const Object& e); // push element onto stack
    void pop(); // pop the stack

private:
    SLinkedList<Object> S; // linked list of elements
    int n; // number of elements

};

#include "LinkedStack.cpp"
#endif
```