MPCS 51040 – C Programming

Lecture 6 – Analysis of Algorithms, Linked List, Stack, Queue, Set

Dries Kimpe

October 31, 2016

Overview

- General Info
- Algorithm Analysis
- Linked Lists
- Stacks
- Heap
- Homework 5

Quiz

Grading not entirely done...

Coding Exercise

Some remarks:

- Don't deviate from the prescribed/requested behaviour: for example, printing extra output or returning different values (or assigning a different meaning to the returned values).
- Control flow: keep things simple, and avoid nesting when possible (see next slide).
- Be careful about making assumptions (consciously or not): for example, assuming input strings are the same length.

Which is easier to read?

```
1  size_t strlen (const char * str)
2  {
3      if (str)
4      {
5          size_t len = 0;
6          while (*str++)
7              ++len;
8          return len;
9      }
10     else
11     {
12         return 0;
13     }
14 }
```

```
1  size_t strlen (const char * str)
2  {
3      if (!str)
4          return 0;
5
6      size_t len = 0;
7
8      while (*str++)
9          ++len;
10
11     return len;
12 }
```

Less indentation is usually better. If you have to nest too deep, maybe you split the function...

Unit Testing

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.
Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing. ***Ideally, each test case is independent from the others****. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.*

Unit Test Frameworks

All unit tests share some common elements:

- Need to report the results
- Need to make it simple to add tests
  - Including setup and teardown of test environments

Because of this, every programming language typically has one or more unit test frameworks/libraries available.

In this course, we will be using `cunit`.

`http://cunit.sourceforge.net/`;
See 'example code' for a quick start.

CUnit

Available as a library (pre-installed on linux.cs.uchicago.edu)

Test Assertions

- CU_ASSERT_FATAL
- CU_ASSERT_TRUE_FATAL
- CU_ASSERT_FALSE_FATAL
- CU_ASSERT_EQUAL_FATAL
- CU_ASSERT_NOT_EQUAL_FATAL
- CU_ASSERT_PTR_EQUAL_FATAL
- CU_ASSERT_PTR_NOT_EQUAL_FATAL
- CU_ASSERT_PTR_NULL_FATAL
- CU_ASSERT_PTR_NOT_NULL_FATAL
- CU_ASSERT_STRING_EQUAL_FATAL
- CU_ASSERT_STRING_NOT_EQUAL_FATAL
- CU_ASSERT_NSTRING_EQUAL_FATAL
- CU_ASSERT_NSTRING_NOT_EQUAL_FATAL
- CU_ASSERT_DOUBLE_EQUAL_FATAL
- CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL

_FATAL versions immediatately end the current test function. Beware of memory leaks, uninitialized variables, etc. that might result from the code skipped due to a _FATAL assert!

See arb_unittest.c in hw4 directory

Analysis of Algorithms

Definition

Study of the resource requirements (memory, CPU) of algorithms when applied to certain problems.

Why?

- Often multiple algorithms exist (sorting for example)
  - How to compare?
- Predict performance on different data
- Determine if there a faster algorithm is possible by comparing algorithm to theory

Worst-Case Analysis

The execution time of most algorithms depend on the exact data they are operating on. Example:

- Search: element might be at the beginning or end
- Sorting: data might be partially sorted

Most analysis of algorithms studies the *worst case* behaviour of the algorithm:

- Many algorithms often exhibit worst case (search)
- Unless you can predict how often the worst case behaviour will occur, known average or best case does not inspire confidence.
- For many algorithms tackling the same problem, best case is often the same.

O-notation

A mathematical description for expressing worst case behaviour; It limits the upper bound of a (mathematical) function within a certain constant factor:

Definition

if $g(n)$ is an upper bound for $f(n)$, this means that $\exists n_0, \exists c : \forall n \geq n_0 : cg(n) > f(n)$

In other words, for a large enough value of n and beyond, $g(n)$ will *always* be an upper bound, even though for smaller values of n, it might not be.

- The analysis focuses on the behaviour for *large* problems

If $g(n)$ is an upper bound for $f(n)$ we can say that $f(n)$ is $O(g(n))$.

O-notation

Some obversations

- Constant running time: $O(k) = O(1)$
  (this follows from the definition as you can pick your c of the definition to be k)
- Only the 'biggest' term matters, since, for large enough values, those will dominate.
  - $O(n^5 + n^4) = O(n^5)$
  - $O(kn^5) = O(n^5)$
  - $O(f1(n)) + O(f2(n)) = max(O(f1(n)), O(f2(n)))$

Examples

- Finding an element in an array is $O(n)$
  (with n the number of elements in the array)
- Sorting a list or array can never be $O(1)$. Why?

O-notation and Algorithms

Space vs Time

O-notation is typically applied for estimating the *time complexity* and *space (memory) complexity* of an algorithm.

How does O-notation relate to analyzing an algorithm (such as search)?

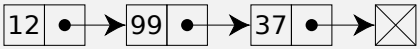
- It is hard to estimate how many 'steps' a statement or expression will take to evalute. Memory usage is often more explicit and easier to analyze.
  - but the exact number doesn't matter! (constants don't matter)
  - If a multiplication takes 100 cycles on one system, and 1000 cycles on another, the analysis is not affected. However, if the algorithm changes *the number* of additions as the size of the input data increases, than that does matter.
- Only loops *depending on the the data* matter; Otherwise the cost of the loop is constant (even though it may be large)!
- O-notation does not really give us information about the actual running time of the algorithm; Only about how the worst case scenario for that running time will change as the input data changes.

See tables 4-1 in the O'Reilly book for some common cases;
Insertion sort;

What are linked lists

Linked List

A linked list is a data structure (i.e. it stores data). There are multiple kinds (single, double, circular, skiplist, ...) but the principle is the same: a list *node* holds a pointer to another node belonging to the same list.



Typical operations:

- Query the size of the list
- Insert an item at a specific position
- Remove an item
- Search for an item
- ...

The type used to represent the list might or might not be a pointer (cfr. hw5). In general: the exact interface does not matter and might differ between implementations; The available operations and their time&space complexity will not.

Why linked lists?

Why not use arrays?

Linked lists and arrays serve different purposes and have different strengths and weaknesses. Some examples:

- Arrays are less flexible
  - In order to add elements, you might have to create a new array and copy all existing elements.
    - This would invalidate any pointers to existing array elements!
  - Adding elements in the middle creates similar issues
- Arrays require contiguous memory blocks
- Linked lists are flexible, but generally are slower to access and have higher overhead.
  - Might need to traverse the list to get to the n^{th} element.
  - We need to store link information

Forward Declarations and Incomplete types

Linked Lists in C

```
1  // Struct without tag
2  typedef struct {
3      int a;
4  } MyStruct;
5
6  MyStruct a;
7
8  // Need tag to refer to self
9  struct Link
10 {
11     int data;
12     struct Link * next;
13 };
```

- The struct on line 2 does not have a tag (which is OK)
- In order to link to other structs of the same type, a tag is needed (line 9)

Generic Data Structures

```
1 // Single-linked list storing
2 // void * pointers
3 struct ListItem
4 {
5     void * data;
6     struct ListItem * next;
7 };
```

- Other than `void *`, there is no good way to provide 'generic' data structures.
- Decide (and document) if you are storing a value type or not. Consequences if not storing value types!
  - Destruction
  - Operations such as testing for equality or partial order.
- By using `void *`, we no longer can rely on the compiler to catch type errors.

Single Linked list implementation demo

Linked Lists

There are many different kinds of linked lists (see Chapter 5 in the O'Reilly book); Considered on-topic:

- Singly-linked list
- Doubly-linked list
- Circular linked list

There are other minor variations; For example:

- Explicitly storing and maintaining the size of the list
- Keeping track of the list tail and/or head
- Availability of non-optimal operations (such as moving backwards in a singly-linked list)

Make sure you understand the complexity of the particular implementation of a data structure before using it!
(You need it anyway to determine the complexity of your own code)

Overview: Stacks and Queues

One-line summary:

stack LIFO structure: Last In First Out

queue FIFO structure: First In First Out

Stack

Operations:

- **push** place something on the stack
- **pop** remove the last item placed on the stack
- **peek** access (but don't remove) the top of the stack
- **size** return the number of elements on the stack

- A stack is easily implemented on top of a linked list, but can also be implemented on top of an array.
- Even if a stack is implemented on top of a linked list, this does not mean we should 'expose' all linked list functionality!
  - This would be breaking the abstraction
  - It would not be possible to swap between different stack implementations

Implement stack using an array

Queue

Operations:

enqueue	Add an element
dequeue	Remove the oldest element
size	Return the number of elements
peek	Access but not remove the oldest element

- As was the case with stacks, queues can be implemented using a linked list.
- Same caveat (about providing a 'pure' interface) applies
- A *deque* supports addition and removal on both end of the queue; Often implemented using a doubly-linked list.

Homework 4 - Discussion

Implementation

- What choice for `arb_int_t`?
- How was operation implemented?

Assignment: Homework 5 & Reading

Homework Assignment

See `https://mit.cs.uchicago.edu/mpcs51040-aut-16/mpcs51040-aut-16//raw/master/homework/hw5/hw5.pdf`

Due next week

- Walk through the steps for reading
- Discuss byte swapping

HW5 info

Reading Assignment

O'Reilly Mastering Algorithms in C:

Required Chapter 8, 9 & 10