# 15-122 : Principles of Imperative Computation

## Fall 2012

## Assignment 4

(Theory Part)

Due: Thursday, October 18, 2012 at the **beginning** of lecture

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this weeks homework will give you some practice working with amortized analysis, memory management, hashtables, and recursion. You can either type up your solutions or write them *neatly* by hand in the spaces provided. You should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

| Question | Points | Score |
|---|---|---|
| 1 | 5 | |
| 2 | 4 | |
| 3 | 3 | |
| 4 | 8 | |
| Total: | 20 | |

You *must* include this cover sheet.
Either type up the assignment using
15-122-theory4.tex, or print out this PDF.

1. **Amortized Analysis.**

(1) (a) There are $n$ ECE students who want to get into the Gates-Hillman Center after 6pm. Unfortunately 15-122 TAs control the building and charge a toll for entrance. The toll policy is the following: for some $k < n$, the $i$-th student is charged $k^2$ tokens when $i \equiv 0 \bmod k$, or zero otherwise. In other words, if $i$ is a multiple of $k$, then student $i$ is charged $k^2$ tokens. Otherwise, student $i$ enters for free. With this policy, how much do the students pay *altogether*?

**Solution:**

(1) (b) In Soviet Russia, TAs pay you. However, Soviet Russia is also communist, and therefore, the students must split the money evenly between themselves. If the 15-122 TAs in Soviet Russia used the same policy for entering the Gavrilovich-Hashlov Center, how much will each student end up with in the end? (i.e. what is the amortized cost *per student*?)

**Solution:**

Famous Fred Hacker's friend, Ned Stacker, loves stacks. He loves them so much that he implemented a queue using two stacks in the following way:

- A Staqueue has an "in" and an "out" stack.
- To enqueue an element, the element is pushed on the "in" stack.
- To dequeue an element, there are two cases:
  - If the "out" stack is non-empty, then we simply pop from the "out" stack.
  - Otherwise, we reverse the "in" stack onto the "out" stack by sequentially popping elements from the "in" stack and pushing them onto the "out" stack. We then pop from the "out" stack.

(1) (c) Fred Hacker says that Ned's implementation is too slow. What is the *worst case* cost for dequeuing an element in terms of the number of elements in the staqueue, $n$?

**Solution:**

(2) (d) Cheer up Ned by giving the amortized cost for dequeue. You must explain your answer. *Hint: consider n calls to dequeue.*

**Solution:**

2. **Memory Management.**

Consider the following two implementations of a stack. The first implementation uses a linked list to create the stack data structure:

```
struct list_stack {
  struct list_node* top;
  struct list_node* bottom;
};

struct list_node {
  char data;
  struct list_node* next;
};
```

and the second implementation uses an unbounded array (recall that unbounded arrays are never full. They will double in size as needed):

```
struct uba_stack {
    struct ubarray* top;
};

struct ubarray {
  int limit;        /* 0 < limit */
  int size;         /* 0 <= size && size <= limit */
  char[] elems;     /* \length(elems) == limit */
};
```

Recall that in $C_0$,

- a `char` is represented using 8 bits
- an `int` is represented using 32 bits
- a pointer is represented using 64 bits

An array of characters of size $n$ takes exactly $8n + 64$ bits, where $8n$ is for the array of $n$ characters, and $2 \times 32$ bits for two `int`s: one to record the length of the array and one to record the size of its elements. These `int`s are not the same as the `limit` and `size` of the `ubarray`, and will occupy separate memory. Also note that the array is actually a pointer (so a character array in a `struct` occupies 64 bits for this pointer, and then an additional $8n + 64$ bits elsewhere in memory for the elements).

The size of a `struct` is the sum of the sizes of its elements. In C (and $C_0$), this will be rounded up to a multiple of the size of the largest element in order to preserve memory alignment. Here, this means that a `list_node` occupies 128 bits of memory. Do not worry about the alignment rounding for arrays.

Consider a program that uses a stack of characters and can allocate only 1 MB ($2^{20}$ bytes, or $8 \times 2^{20}$ bits) of heap memory for this stack. Local variables, in this case the single pointer to the `list_stack` or `uba_stack` structure in the heap, are stored on the call stack and do not count against this limit.

(2) (a) What is the maximum number of characters that can be stored using a `list_stack`? Show your work.

**Solution:**

(2) (b) What is the maximum number of characters that can be stored using a `uba_stack`? Show your work. Assume that the array in the `uba` is not restricted to have sizes that are powers of two, and will grow to have whatever size will store the most characters in the allowed memory.

**Solution:**

3. **Hash Tables.**

In Java, strings are hashed using the following function:

$$(s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-2] * 31^1 + s[n-1] * 31^0) \ \% \ m$$

where $s[i]$ is the ASCII code for the $i$th character of string $s$, $n$ is the length of the string, and $m$ is the hash table size.

(1) (a) If 15251 strings were stored in a hash table of size 4126, what would the load factor of the table be?

**Solution:**

(2) (b) Using the hash function above with a table size of 4126, give an example of two strings that would "collide" and would be stored in the same chain. Note that strings are case sensitive and string length must be $\geq 3$. Briefly describe your reasoning. (Think of short strings please!)

**Solution:**

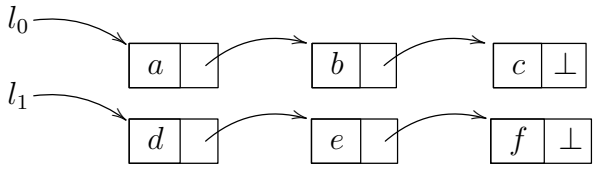
4. **Memory Contracts.** As lists are defined as `structs`, in $C_0$ we can only interact with them by reference—that is, with pointers. As a result, we must be careful that two lists which we would like to think of as separate are actually stored in distinct memory. For value types, like `int`s, this is automatically handled by the type system. For reference types (pointers), we can reason with contracts.

   For the purposes of this question, we will consider operations on non-circular, `NULL`-terminated lists. You may assume that you have a specification function:

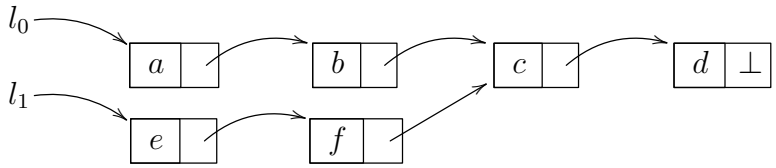
   ```
   bool is_nt_list(list *L)
   ```

   which correctly checks a list for these properties.

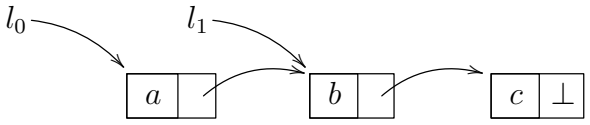
   Two lists can either be distinct,

   Or they can share some tail.

   Note that once the lists join, they cannot diverge. Also, note that one list may be contained within the other.

   We can think of this as a special case of the lists sharing a tail—that tail just happens to be one of the lists in its entirety.

   By convention, we treat `NULL` as the (valid) empty list. This means that `is_nt_list(NULL)` will return true, and a list that is `NULL` is distinct from everything, as it does not occupy memory.

(3) (a) Write the following function to check if two lists occupy different memory. Your function should return true if there is no list node that is common to both lists, false if the lists share some tail, or raise an annotation failure if either list is not a valid `NULL`-terminated list. There are several ways to do this; the fastest is $O(n)$, but an $O(n^2)$ solution is acceptable. Use the given contracts, and don't modify them or add other preconditions or postconditions.

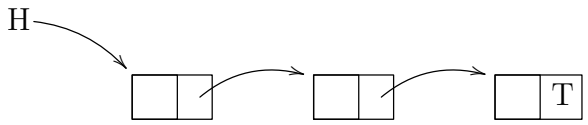
**Solution:**

```
bool lists_pwise_distinct(list *L0, list *L1)
//@requires is_nt_list(L0) && is_nt_list(L1);
{
```

A list segment can be described by two list pointers:

```
bool is_segment(list *H, list *T)
{
  list *p;
  for (p = H; p != NULL && p != T; p = p->next) {}
  return p == T;
}
```

`is_segment(H,T)` is true if and only if `H` is connected to `T` via list nodes.




Notice that `is_segment(H,NULL)` is true if and only if `H` is a `NULL`-terminated list, assuming the list is not circular.

The functions above are useful in a function that copies a list:

```
list *nt_list_clone(list *L)
//@requires is_nt_list(L);
//@ensures is_nt_list(\result);
//@ensures lists_pwise_distinct(\result, L);
{
    list *new_head = NULL;
    list *new_tail = NULL;
    list *p = L;

    while(p != NULL)
    //@loop_invariant is_nt_list(new_head);
    //@loop_invariant lists_pwise_distinct(new_head, L);
    //@loop_invariant is_segment(new_head, new_tail);
    {
        // copy the node
        list *new_node = alloc(list);
        new_node->data = p->data;
        new_node->next = NULL;
        //@assert lists_pwise_distinct(new_node, L);

        // if this is the first node, set up the clone
        if(new_tail == NULL)
        {
          new_tail = new_node;
          new_head = new_node;
        }
        // otherwise add to the clone
        else
        {
          new_tail->next = new_node;
          new_tail = new_node;
        }

        // follow the original list
        p = p->next;
    }

    return new_head;
  }
```

You will now prove the contracts for `nt_list_clone`. Don't worry about equality of the elements—for this problem, we're only concerned with the structure of the lists in memory. You may assume that memory returned by `alloc` is distinct from any memory already in use, per the `@assert` statement in the loop body.

(1) (b) Show that the loop invariants hold upon entering that the loop.

**Solution:**

(3) (c) Show that the loop invariants hold after an iteration of the loop, given that they held at the beginning of that iteration.

**Solution:**

(1) (d) Show that the postconditions are satisfied. *HINT: look at the loop invariants!*

**Solution:**