15-122 : Principles of Imperative Computation, Spring 2013

Homework 4 Theory

Due: Thursday, March 7, 2013, at the **beginning** of lecture

Name: ______________________

Andrew ID: ______________________

Recitation: ______________________

The written portion of this week's 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	6	
3	3	
4	6	
Total:	20	

You *must* use this printout, include this cover sheet, and staple the whole thing together before turning it in. Either type up the assignment using 15122-theory4.tex, or print this PDF and write your answers *neatly* by hand.

1. **Amortized Analysis.**

(1) (a) There are n students $\{s_0, s_1, ..., s_{n-1}\}$ 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 $k < n$, student s_i is charged k^2 tokens when $i \equiv 0 \bmod k$, or zero otherwise. If i is a multiple of k, then student s_i is charged k^2 tokens. Otherwise, the student enters for free. With this policy, how much do the students pay *altogether*?

Solution:

(1) (b) In Soviet Russia, TAs pay you (the TAs really want students at their office hours). However, Soviet Russia is also communist, so 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 after 6pm – student s_i is paid k^2 tokens when $i \equiv 0 \bmod k$ – 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.

A token can pay for the $O(1)$ cost of a stack push or pop; your answers should be in terms of tokens (not in terms of Big-O notation).

(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?

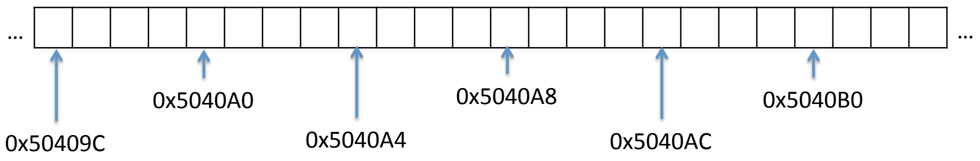
Solution:

(2) (d) Cheer up Ned by giving and justifying the amortized cost for the queue operations (enqueue and dequeue). You'll need to show that it is possible to account for the eventual cost of a dequeue by paying some of that cost each time you enqueue. (It may help to consider n calls to dequeue in a staqueue with n elements.)

Solution:

2. **Memory Management.**

Recall that pointers in C0 are just addresses in the *heap*, and the heap is just one portion of the memory, which we can think of as an enormous array of *bytes*.



For the purposes of this class, we will always think of a *byte* as containing 8 *bits*. In the reference C0 implementation:

- a `char` is represented using 8 bits (1 byte)
- an `int` is represented using 32 bits (4 bytes)
- a pointer is represented using 64 bits (8 bytes)

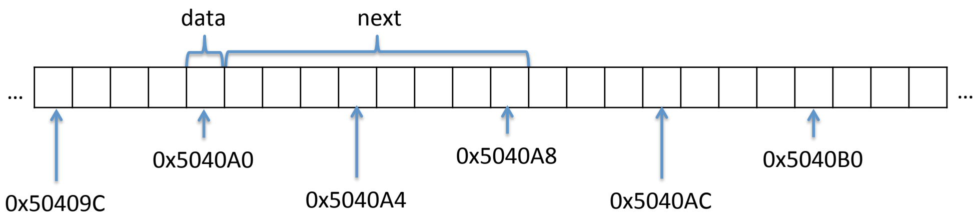
Every time `alloc(ty)` is called, there is code behind the scenes that reserves a portion of the heap large enough to store the desired data, initializes that portion of the heap to default values, and returns the address of the beginning of that portion of the heap.

Consider a stack of characters implemented using a linked list:

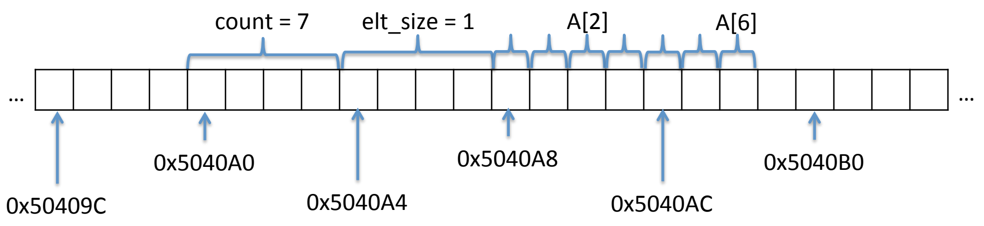
```
struct list_stack {
  struct list_node* top;
  struct list_node* bottom;
};

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

This is the most compact way that C0 could choose to store a `struct list_node` whose address – the thing returned from `alloc(struct list_node)` – was `0x5040A0`:



An array in C0 includes some extra information: every time `alloc_array(ty,n)` is called, the default implementation of C0 works behind the scenes to reserve not just enough space for the `n` elements of type `ty`, but *also* 8 extra bytes: 4 bytes to store an integer representing how many bytes one array element takes up, and another 4 bytes to store an integer representing the number of elements in the array (this is how we implement `\length()` and check for array bounds errors). If `alloc_array(char, 7)` returned `0x5040A0`, then this would be the most compact way C0 could choose to store this array of seven `char`s:



The actual *value* of an array is an address (same as for a pointer), so it takes 64 bits (8 bytes) to store the address where an array lives.

Remember that we can also implement a stack of characters with an array:

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

(1) (a) In this problem, we want to fit our *entire* heap inside of 1 kB (1024 bytes, or 8192 bits) of continuous memory addresses.

Suppose the lowest address that is part of the heap is `0x504000`. What is the highest address that `alloc(char)` could conceivably return? What is the highest address that `alloc(int*)` could conceivably return? (Give answers in hex.)

Solution:

`alloc(char):` `alloc(int*):`

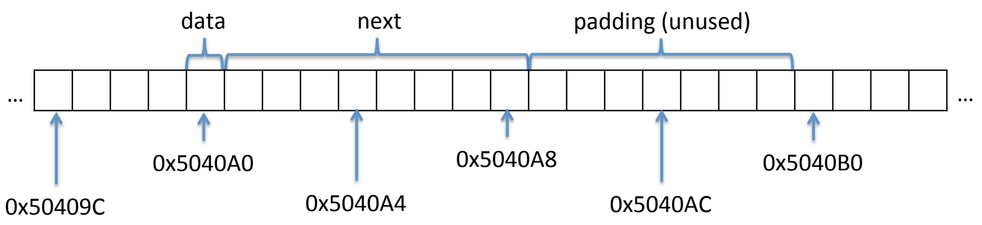
(2) (b) What is the maximum number of characters that can be stored using a `list_stack` if we pack everything as tightly as possible into a 1 kB heap? Assume you have a dummy node. Show your work.

Solution:

(2) (c) What is the maximum number of characters that can be stored using an `array_stack` if we pack everything as tightly as possible into a 1 kB heap? Assume you use every position in the array (don't ignore the 0th array element as we did in lecture 9). Show your work.

Solution:

(1) (d) Because addresses are given in terms of bytes, the C0 implementation obviously cannot reserve *less* than a byte at a time. Because of a property called *alignment*, C0 will actually reserve *more* than a byte at a time. In particular, the `alloc()` and `alloc_array()` implementations always reserve multiples of 16 bytes (128 bits), so that even if we just need 1, 4, or 9 bytes, the allocation will reserve all 16:



In light of this, how many characters can be stored with a `list_stack`?

Solution:

How many characters can be stored with an `array_stack`?

Solution:

(While it doesn't matter for this particular question, alignment is a bit more complicated than we have described it here, and influences the positioning of fields like `data` and `next` within a struct. You'll learn more about alignment in upper-level courses.)

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 ith character of string s, n is the length of the string, and m is the hash table size.

(1) (a) If 15122 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 ≥ 3. Briefly describe your reasoning. (Think of short strings please!)

Solution:

4. **Linked Lists.**

   We want to find the maximum element in a linked list. The recursive data structure of a linked list, as seen in lecture, is defined as:

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;
```

The client provides a helper function `elem_compare`. Its signature is:

```
int elem_compare(elem a, elem b)
//@ensures -1 <= \result && \result <= 1;
  ;
```

and it returns -1 if `a` is less than `b`, 0 if `a` is equal to `b`, and 1 if `a` is greater than `b`.

We've also written a function `leq`. Its signature is:

```
bool leq(list* start, list* end, elem e)
//@requires is_segment(start, end);
  ;
```

and it returns true if every node in the list from `start` (inclusive) to `end` (exclusive) is less than or equal to `e`. We treat the `end` list node as a dummy node, as we do in lecture, and this dummy node may not even contain valid data.

You can also use the `is_segment` function described in lecture. Its signature is:

```
bool is_segment(list* start, list* end);
```

and it returns true if `start` and `end` delineate a valid list segment. Unlike the `is_segment` from lecture, this `is_segment` function has no precondition: it will just return `false` if `start` or `end` are `NULL`.

(2) (a) Finish the `find_max` function. It takes in two pointers, which should delineate a valid list segment, and returns the maximum element in the list segment. You will need to provide one more precondition in order for line `7` to make sense, and the loop invariant you give on line 11 should be true initially, should be preserved by every iteration of the loop, and should be strong enough to prove the postcondition.

Solution:

```
/*  1 */  elem find_max(list* start, list* end)

/*  2 */  //@requires is_segment(start, end);

/*  3 */  //@requires ________________________________________;

/*  4 */  //@ensures leq(start, end, \result);

/*  5 */  {

/*  6 */    list* curr = start;

/*  7 */    elem max = ________________________________________;

/*  8 */    while (___________________________________________)

/*  9 */      //@loop_invariant is_segment(start, curr);

/* 10 */      //@loop_invariant is_segment(curr, end);

/* 11 */      //@loop_invariant ______________________________;

/* 12 */      {

/* 13 */        curr = ________________________________________;

/* 14 */        if (________________________________________) {

/* 15 */          _____________________________________________;

/* 16 */        }

/* 17 */      }

/* 18 */    return max;

/* 19 */  }
```

(2) (b) Prove that the first two loop invariants, on lines `9` and `10`, are valid loop invariants. (They are true initially and are preserved by every iteration of the loop.)

Remember that, when discussing preservation of the loop invariant, you want to account for *curr* (the value in `curr` before the loop guard is checked and returns `true`) and *curr′* (the value in `curr` just before the loop guard is next checked).

Solution:

(2) (c) Prove that every pointer dereference you make is safe.

Solution: