Midterm I Exam

15-122 Principles of Imperative Computation
Frank Pfenning

October 9, 2012

Name: **Sample Solution** Andrew ID: **fp** Section:

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.

Please keep in mind that this is a sample solution*, not a model solution. Problems admit multiple correct answers, and the answer the instructor thought of may not necessarily be the best or most elegant.*

	Bit-level Operations	Linear Search	Binary Search	Hop Lists	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	**20**	**20**	**30**	**30**	**100**
Max	20	20	30	30	100
Grader					

1 Bit-Level Operations (20 pts)

In this problem we explore the representation of sets of letters `'A'` through `'Z'` as 32-bit words. We assign a particular bit position in a 32-bit word to each letter, starting with 0 for `'A'`, 1 for `'B'`, etc. all the way to 24 for `'Y'` and 25 for `'Z'`. A 32-bit word (of type `int` in C0) represents the set of all letters whose corresponding bit is set to `1`. For example, the word

						Z	Y	X	···	D	C	B	A	letter
31	30	29	28	27	26	25	24	23	···	3	2	1	0	bit position
0	**0**	**0**	**0**	**0**	**0**	1	1	0	···	0	1	0	1	bit value

represents the set $\{\texttt{A}, \texttt{C}, \texttt{Y}, \texttt{Z}\}$ if all bits 3 through 23 are 0. Bits 26 through 31 must always be 0 since they do not correspond to any letter in the alphabet.

Task (20 pts). Complete the following table. You may use only numeric constants in **decimal or hexadecimal** notation, and the following bit-level operations on values of type `int`:

`&` `|` `^` `<<` `>>` `~` `<` `<=` `>=` `>` `==` `!=` `pos(`c`)`

where the function `pos` converts a character to its bit position (for example, `pos('D') == 3`).

We assume we have variables x and y of type `int` representing *sets of letters*, and a variable c of type `char` representing a letter. We already filled in two answers for you.

The empty set	`0x00000000`
The set of all letters `A` through `Z`	`0x03FFFFFF`
The set $\{\texttt{C}, \texttt{D}, \texttt{E}, \texttt{G}\}$	`0x0000005C`
Testing if the letter c is in set x (computing a boolean)	`x & (1 << pos(c)) != 0`
Adding a letter c to a set x	`x | (1 << pos(c))`
The intersection of sets x and y	`x & y`
The union of sets x and y	`x | y`
The set of all letters `A` through `Z` not in x	`x ^ 0x03FFFFFF`
Replacing every letter in the set x with its successor in the alphabet, except `Z` which becomes `A`	`((x << 1) & 0x03FFFFFF) | (x >> 25)`

2 Linear Search (20 pts)

In this problem we explore search through arrays, with particular attention to contracts. Rather than requiring arrays to be sorted, we will be working with *peaked arrays*. An array of integers is *peaked* if its values are *strictly increasing* from the start of the array to the peak and *strictly decreasing* from the peak to the end of the array. The strictly increasing segment or the strictly decreasing segment could be empty, which means that the peak might occur at the first element or last element of the array, respectively. It might be helpful to draw a generic diagram of a peaked array for yourself.

We will use the functions `is_peaked` and `gt_seg` in contracts about peaked arrays. No other functions will be allowed in your contracts, but you can use any ordinary arithmetic operations, comparisons, and array accesses.

```
bool is_peaked(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
  ;

bool gt_seg(int x, int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
  ;
```

They have the following specification

`is_peaked(A, lower, upper)` if the segment $A[lower..upper)$ is *peaked*.

`gt_seg(x, A, lower, upper)` if $x > A[lower..upper)$, that is, x is strictly greater than any element in the array segment from $lower$ (inclusively) to $upper$ (exclusively).

Task (20 pts). Complete the function `find_peak_lin` that performs a *linear search* through a non-empty peaked array to return the index of its peak. Your code must satisfy all stated contracts and **you are not allowed to change the given contracts and code**. You are not allowed to call the specification functions `gt_seg` and `is_peaked`, which are reserved to be used in contracts only. The body of your loop should have either one or two statements; if it is just one, leave the other line blank.

```
int find_peak_lin(int[] A, int n)
//@requires 0 < n && n <= \length(A);
//@requires is_peaked(A, 0, n);
//@ensures 0 <= \result && \result < n;
//@ensures gt_seg(A[\result], A, 0, \result);
//@ensures gt_seg(A[\result], A, \result+1, n);
{
  int i = 0;
  while (i < n-1 && A[i] < A[i+1])
    //@loop_invariant 0 <= i && i <= n-1;
    //@loop_invariant gt_seg(A[i], A, 0, i);
    //@loop_invariant is_peaked(A, i, n);
    {
      i = i+1;
    }
  return i;
}
```

3 Binary Search and Contracts (30 pts)

We continue programming with peaked arrays from Problem 2.

Task 1 (20 pts).

Complete the function `find_peak_bin` that performs a *binary search* through a non-empty peaked array to return the index of its peak. The pre- and postconditions are the same as for the linear search function `find_peak_lin`.

This time, we have given you **all** the code and ask you to fill in loop invariants. They should be strong enough to guarantee safety of all array accesses and, together with the negated loop guard, prove the postcondition. **You may not modify the code, the preconditions, and the postconditions**. You do not need to have exactly 4 loop invariants. If you need fewer, just leave some blank; if you need more, write them legibly at the right of the page. We have left space for optional `@assert` annotations which you can use for brief assertions that may help you in reasoning about the code, might help us in assigning partial credit.

```
int find_peak_bin(int[] A, int n)
//@requires 0 < n && n <= \length(A);
//@requires is_peaked(A, 0, n);
//@ensures 0 <= \result && \result < n;
//@ensures gt_seg(A[\result], A, 0, \result);
//@ensures gt_seg(A[\result], A, \result+1, n);
{
  int lower = 0;
  int upper = n-1;
  while (lower < upper)
    //@loop_invariant 0 <= lower && lower <= upper && upper < n;
    //@loop_invariant is_peaked(A, lower, upper+1);
    //@loop_invariant gt_seg(A[lower], A, 0, lower);
    //@loop_invariant gt_seg(A[upper], A, upper+1, n);
    {
      int mid = lower + (upper-lower)/2;
      //@assert mid < upper;
      if (A[mid] < A[mid+1])
        lower = mid+1;
      else //@assert A[mid] > A[mid+1];
        upper = mid;
    }
  //@assert lower == upper;
  return lower;
}
```

Task 2 (5 pts). State which integer quantity strictly decreases on every iteration of the loop and is bounded from below, thereby assuring termination. What bound does it never fall below?

$upper - lower$ strictly decreases each time and is bounded from below by 0.

Task 3 (3 pts). How many iterations will the while loop perform in the worst case for a peaked array of 4 million elements?

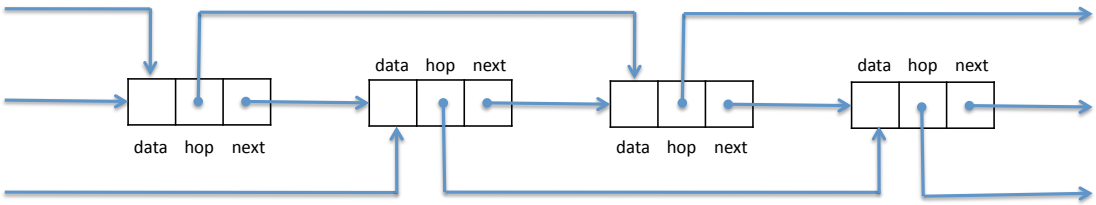
4 million is slightly less than $2^{10} * 2^{10} * 2^2 = 2^{22}$ so it may take up to 22 iterations until the peak is found.

Task 4 (2 pts). What is the asymptotic complexity of `find_peak_bin(A, n)` as a function of n in big-O notation? You do not have to explain your answer.

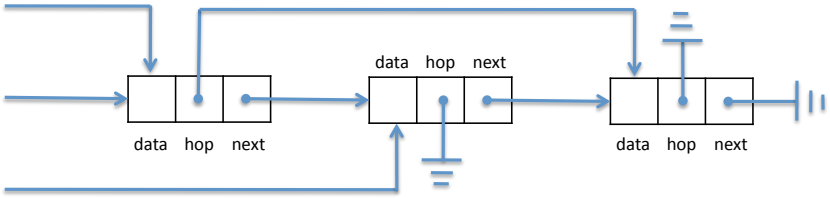
$O(\log(n))$

4 Hoplists (30 pts)

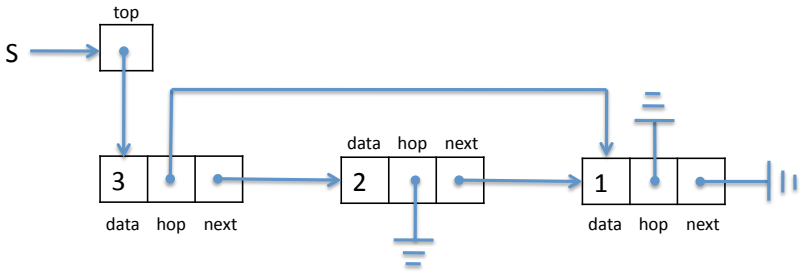
An employee of Reverse Polish Systems (which holds the patents on the Clac programming language) had the somewhat ill-conceived idea to improve the speed of the language interpreter by replacing linked lists in the implementation of stacks using *hop lists*. A *hop list* is a form of linked list where every node has **two** pointers: one to the next node in the list (the `next` pointer), and one *hopping over* the next node to the one that follows (the "`hop` pointer").



The empty hop list is just `NULL`. Non-empty hop lists are always terminated with a `next` pointer that is `NULL`. At the end of a hop list, the second-to-last (if it exists) and the last `hop` pointers should both be `NULL`. This is illustrated in the following diagram:



Finally, here is the data structure for a stack S, implemented using a hop list, after the numbers 1, 2, and 3 have been pushed onto it, in this order (so the top of the stack is 3).



The implementation starts as follows:

```
struct hoplist_node {
  int data;
  struct hoplist_node* hop;
  struct hoplist_node* next;
};
typedef struct hoplist_node hoplist;

struct stack_header {
  hoplist* top;
};
typedef struct stack_header* stack;
```

Task 1 (10 pts).

Having taken 15-122, the employee's first task was to write new `is_stack` and `is_hoplist` functions. Unfortunately, the code below is *unsafe* in that it contains **one or more** unguarded pointer dereferences. Please circle each expression which could raise a null-pointer exception and show on the right-hand side of the page how you would fix the code so that it (a) is now safe, and (b) works correctly as intended to check whether hoplists and stacks are valid.

Do **not** circle or modify any pointer dereferences that can already proved to be safe. For full credit, you should replace each offending line by one or more, but you should not rewrite the whole function.

```
bool is_hoplist(hoplist* start) {
  hoplist* p = start;
  hoplist* q = (p == NULL ? NULL : p->next); /* !! */
  while (p != NULL) {
    if (p->next != q) return false;
    q = p->hop;
    p = p->next;
  }
  return q == NULL;
}

bool is_stack(stack S) {
  return S != NULL && is_hoplist(S->top); /* !! */
}
```

Next, the employee decided to extend the interface to stacks so that some operations could be performed more efficiently. She was particularly proud that the `rot` operation of Clac could be implemented **without any memory allocation**. The new function `has3` checks that the stack has the required three elements, while the function `rot3` performs the rotation. Recall the rotation of a stack with z at the top, y second, and x third down the stack should change the stack so that x is now at the top, z is second, and y is third.

Task 2 (20 pts).

Complete the new functions `has3` and `rot3` explained above. The new functions are on the *inside* of the extended stack library and should work with the internals of the representation. For full credit, your implementation should exploit the existence of the `hop` field. You do not need to use exactly the number of lines indicated; you may leave lines blank or write additional lines legibly below if necessary.

```
bool has3(stack S)
//@requires is_stack(S);
{
  hoplist* p = S->top;
  return p != NULL && p->next != NULL && p->hop != NULL;
}

void rot3(stack S)
//@requires is_stack(S) && has3(S);
//@ensures is_stack(S) && has3(S);
{
  hoplist* p = S->top;
  int z = p->data;
  p->data = p->hop->data;
  p->hop->data = p->next->data;
  p->next->data = z;
  return;
}
```