Final Exam

15-122 Principles of Imperative Computation
Frank Pfenning
December 14, 2012

Name:  Sample Solution  Andrew ID:  fp  Section:

Instructions

• This exam is closed-book with one sheet of notes permitted.
• You have 180 minutes to complete the exam.
• There are 6 problems on 17 pages.
• 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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Spiral Sort</th>
<th>Amortized Analysis</th>
<th>Binary Search Trees</th>
<th>Safety in C</th>
<th>Binary Trees</th>
<th>Union Find</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prob 1</td>
<td>Prob 2</td>
<td>Prob 3</td>
<td>Prob 4</td>
<td>Prob 5</td>
<td>Prob 6</td>
</tr>
<tr>
<td>Score</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Max</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Grader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total 250
1 Spiralsort (45 pts) [C0]

In this problem, we discuss spiral sort, a variant of insertion sort. Its chief (and perhaps its only) virtue is that its code is exceedingly short. Here is the code in C0.

```c
void spiralsort(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures is_sorted(A, 0, n);
{
    for (int i = 0; i < n; i++)
        for (int k = 0; k < i; k++)
            if (A[i] < A[k])
                swap(A, i, k);
    return;
}
```

**Task 1** (10 pts). Suppose we have reached the state shown in the first line below just before testing the loop guard \( k < i \) in the inner loop. Show the values of \( i, k, \) and \( A \) every time we reach this point in the program again with a different state of the array \( A \). We have filled in the second line for you to get you started.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( k )</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>2 5 7 4 1 9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2 4 7 5 1 9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2 4 5 7 1 9</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1 4 5 7 2 9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1 2 5 7 4 9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1 2 4 7 5 9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1 2 4 5 7 9</td>
</tr>
</tbody>
</table>
**Task 2** (20 pts). Fill in the missing loop invariants, using the usual arithmetic operators and the following contract functions as needed:

- `is_sorted(A, lower, upper)` if the segment `A[lower..upper)` is sorted in ascending order.
- `le_segment(x, A, lower, upper)` if `x ≤ A[lower..upper)`, which means that `x ≤ A[i]` for all `i` such that `lower ≤ i < upper`.
- `ge_segment(x, A, lower, upper)` if `x ≥ A[lower..upper)`, which means that `x ≥ A[i]` for all `i` such that `lower ≤ i < upper`.

For full credit, your loop invariants should actually be loop invariants, which means they should hold initially each time a loop is entered, and you should be able to prove that they are preserved on each iteration. Moreover, your loop invariants should be strong enough to prove the postcondition. But we are not asking you to actually carry out or write down these proofs.

```plaintext
void sort(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures is_sorted(A, 0, n);
{
    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i && i <= n;
        //@loop_invariant is_sorted(A, 0, i);
        for (int k = 0; k < i; k++)
            //@loop_invariant 0 <= k && k <= i;
            /* next two are equivalent under the last one */
            //@loop_invariant ge_seg(A[i], A, 0, k);
            //@loop_invariant k == 0 || A[k-1] <= A[i];

            //@loop_invariant is_sorted(A, 0, i);
            if (A[i] < A[k])
                swap(A, i, k);
        return;
    }
```
Task 3 (5 pts). What is the asymptotic time complexity of this function in big-O notation?

\[ C(n) \in O(n^2) \]

Task 4 (10 pts). We experimentally determined that sorting a random array with 4000 elements on a particular machine takes 0.4 seconds. What are the approximate expected running times for arrays of size:

- 4000: 0.4 seconds
- 8000: 1.6 seconds
- 12000: 3.6 seconds
2  Amortized Analysis (40 pts) [C0]

When implementing unbounded arrays on an embedded device, a programmer is concerned that doubling the size of the array when we reach its limit may use precious memory resources too aggressively. So she decides to see if she can increase it by a factor of $\frac{3}{2} = 1.5$ instead, rounding up if the result is not an integral number.

We consider here mainly the `uba_add` operation that adds a new element to the end of the array. For convenience, we count the exact number of write operations to arrays. We ignore reads and the time needed to allocate new memory.

**Task 1** (15 pts). We now carry out the the amortized analysis for this version of unbounded arrays using the accounting method. Below we show the situation just after we have increased the size of the array from `size` (which was the old limit) to a new `limit`.

Next we walk through the proof that insertion still has constant amortized cost. Express your answers below in terms of `size`, `limit` and `k` in general terms, rather than the the specific numbers in the diagram above. For the purpose of this proof, you should assume `size` is evenly divisible by 4 (so you don’t have to worry about rounding).

Assume at this point we have $k \geq 0$ tokens. In the worst case, each operation is an addition to the end (increasing `size`). Each add takes 1 write operation, and in addition we need to set aside three tokens.

After `size`/2 insertions we reach `limit` and will have to increase the array size again, allocating a new array with $(3/2) \times limit$ elements and copying `limit = (3/2) \times size` elements from the old array to the new array. At this point we have a total of $k + 3 \times (size/2) - (3/2) \times size = k$ tokens left. This is not negative, so the number of available tokens must always be non-negative. Since we have treated the worst case of operations, each insertion has constant amortized time.
Task 2 (5 pts).

Give a simple bound on the total number of write operations for \( n \) consecutive insert operations starting with an empty unbounded array of size 1 with the new strategy. Hint: consider the intuitive interpretation of tokens and the amortized analysis.

\[
4 \times n
\]

Contrast this with a bound on the total number of write operations for \( n \) consecutive insert operations starting with an empty unbounded array of size 1, when we double the size each time we run out of space in the array.

This would only be \( 3 \times n \), so the \( 3/2 \) version may be slower.

The programmer is somewhat unsure of her proof. She decides to augment her code in order to check at runtime that she never runs out of tokens and that therefore each operation on unbounded array has constant amortized time. For that purpose, she adds a new field to the data structure for unbounded arrays called `tokens`.

```c
struct uba_header {
    int limit; /* 0 < limit */
    int size; /* 0 <= size && size <= limit */
    int tokens; /* tokens >= 0 */ /* NEW! */
    elem[] A; /* \length(A) == limit */
};
```

Task 3 (10 pts). Add lines of code and assertions in the `uba_resize` function below to properly maintain the `tokens` field and verify that it does not become negative. Write the new line on the right and indicate clearly where in the function they belong.

```c
void uba_resize(uba L, int new_limit)
... {
    elem[] B = alloc_array(elem, new_limit);
    for (int i = 0; i < L->size; i++)
        //@loop_invariant 0 <= i && i <= L->size;
        { B[i] = L->A[i];
            //@assert(L->tokens > 0); /* check if positive */
            L->tokens--; /* account for write to B[i] */
        }
    L->limit = new_limit;
    /* L->size remains unchanged */
    L->A = B;
    return;
}
```
**Task 4** (10 pts). Complete the function `uba_add` below to implement the 3/2 resizing strategy. You may use the C0 library functions `max_int()` and `min_int()` as needed.

```c
void uba_add(uba L, elem e)
//@requires is_uba(L);
//@ensures is_uba(L);
{
    if (L->size == L->limit) {
        assert(L->size <= 2*(max_int()/3)-1); /* check for int overflow */
        uba_resize(L, 3*((L->size+1)/2)); /* resize, rounding up */
    }
    L->tokens += 3; /* maintain tokens field */

    //assert L->size < L->limit;
    L->A[L->size] = e;
    L->size++;
    return;
}
```
3 Binary Search Trees (35 pts)

Task 1 (5 pts). State the ordering invariant for binary search trees.

For every node \( u \) in the tree, the keys of all nodes in the left subtree are strictly smaller than \( k \), while the keys of all nodes in the right subtree are strictly larger than \( k \), where \( k \) is the key of \( u \).

Task 2 (10 pts). Find an order of insertions of the elements 10, 20, 30, 40, 50 into an empty binary search tree without rebalancing operations such that the resulting tree has the following shape.

Insertion sequence: 10, 50, 20, 40, 30

Label each of the nodes of the tree with the element stored there.
Task 3 (5 pts). State the balance invariant for AVL trees.

The heights of the left and right subtrees of all nodes in the tree differ by at most one.

Task 4 (15 pts). Using the standard rebalancing operations on AVL trees, show the state of the tree after each insertion in the sequence from Task 2. You should be showing a sequence of 5 trees; if you use any auxiliary trees please clearly mark them as such.
4 Safety in C (40 pts)

In C, we consider a statement safe if it has defined behavior. A precondition for safety is an assertion which, if true, guarantees that the following statement will execute safely. As an example, assume we have a variable \( x \) declared and initialized with \( \text{int } x = \ldots \) for some unknown value. The assertion \( x < 0 \) is a safety precondition for an increment of \( x \).

\[
\text{ASSERT}(x < 0); \\
x = x+1;
\]

The precondition \( x < 0 \), while guaranteeing safety, poses some unnecessary restrictions, because the meaning of \( x + 1 \) is defined for many other values of \( x \). The weakest precondition for safety imposes the fewest restrictions under which the following statement is safe. In the above example, the weakest precondition would be \( x < \text{INT\_MAX} \).

\[
\text{ASSERT}(x < \text{INT\_MAX}); \\
x = x+1;
\]

If a statement is always safe, the weakest precondition is just “true”; if a statement is never safe, the weakest precondition is just “false”.

**Task 1** (5 pts). Fill in the weakest preconditions for safety for each of the following statements. You may use macros in stdlib.h, bool.h, and limits.h. You should assume that all initialization code returns without aborting and that there are no intervening statements.

```c
int x = \ldots; int y = \ldots;
int A[13];

\text{ASSERT}(0 <= x && x < 13 && y <= \text{INT\_MAX}-x) ;

A[x] = x*y;
```

**Task 2** (5 pts).

```c
int x = \ldots;
int *B = malloc(17*sizeof(int));

\text{ASSERT}(B != \text{NULL} && x > \text{INT\_MIN}) ;

*B = x-1;
```
Task 3 (5 pts).

```c
int x = ...; int y = ...;
int *C = calloc(42, sizeof(int));

ASSERT(C != NULL && 0 <= x && x < 42 && x != 1 && y < INT_MAX);
C[x] = (y+1)/(1-x);
```

Task 4 (5 pts). In this task, the compound statement (for loop) is considered as a single statement which requires a precondition for safety.

```c
int x = ...;
int A[13];
A[0] = 59;

ASSERT(x <= 13);
for (int i = 1; i < x; i++)
    A[i] = A[i-1];
```

Task 5 (10 pts). In this task the weakest precondition for safety might be more complicated, so we leave room for multiple lines. Again, the compound statement (here a while loop) is considered a single statement requiring a safety precondition.

```c
unsigned int u = ...; unsigned int w = ...;
int *B = malloc(42*sizeof(int));
int *C = calloc(42, sizeof(int));

ASSERT(u >= 42 || (B != NULL && C != NULL && 41-u <= w && w < 42));

while (u < 42) {
    B[u] = C[w];
    u++;
    w--;
}
```
Task 6 (5 pts). In this task, do not assume that unsigned ints are 32 bits wide. You may use functions and macros in `<stdlib.h>` and `<limits.h>`.

```c
unsigned int k = ...;
ASSERT(k < 8*sizeof(unsigned int));
k = 1 << k;
```

Task 7 (5 pts).

```c
unsigned int n = ...;
unsigned int sum = 0;
ASSERT(true);
for (unsigned int u = 0; u < n; u++)
  sum = sum + u;
```
5 Binary Tries (45 points) [C]

A binary trie is a trie for storing bit sequences where each node has exactly two successors: a lo successor indicating a bit 0 and a hi successor indicating a bit 1.

We make a further simplifying assumption, namely that any bit sequence stored in a binary trie has the same length \( k \). Together with the fact that there are only two possible successors at each node in the trie, we do not need to store any information in the nodes at all except for the successors!

A bit sequence is in the trie if we reach a non-NULL node after traversing the trie, following the lo link for a 0 bit, and the hi link for a 1 bit. For example, the following trie contains the bit sequences 00 and 10 (so \( k = 2 \)).

Here is our implementation of tries, together with a function that checks the trie invariants. We do not use a header node.

```c
struct bintrie {
    struct bintrie* lo;
    struct bintrie* hi;
};

typedef struct bintrie bt;

bool is_bt(bt *B, int i, int k) {
    if (!(0 <= k)) return false;
    if (B == NULL) return i == k+1;
    if (!(0 <= i && i <= k)) return false;
    return is_bt(B->lo, i+1, k) && is_bt(B->hi, i+1, k);
}
```
We want to use our tries generically and not force a particular representation of bit sequences. So when we test membership we pass it a generic argument \( w \) intended to be interpreted as a bit sequence, and a pointer to a function \( \text{bit} \) such that \( \text{bit}(w, i) \) is false if the \( i \)th bit of \( w \) is 0 and true if the \( i \)th bit of \( w \) is 1. (So if \( w \) is the sequence 0100000, then \( \text{bit}(w, 1) \) should return true and \( \text{bit}(w, 0), \text{bit}(w, 2), \text{bit}(w, 3), \ldots, \text{bit}(w, 6) \) should return false.) Here, \( i \) must be in the range from \( 0 \leq i < k \).

**Task 1** (10 pts). Complete the following declaration for the membership test in a binary trie. You may silently assume `<stdbool.h>` and any other standard library you need has been included. Your types should be compatible with the ones we used in the `is_bt` function above.

```c
bool bt_member(bt *B, void* w, bool (*bit)(void* w, int i), int k);
```

**Task 2** (15 pts). Complete the function `bt_member`. Use *iteration*, not recursion.

```c
bool bt_member(bt *B, void* w, bool (*bit)(void* w, int i), int k) {
    REQUIRE(is_bt(B, 0, k) && w != NULL && bit != NULL);
    for (int i = 0; i < k; i++) {
        if (B == NULL) return false;
        if (!(*bit)(w, i))
            B = B->lo;
        else
            B = B->hi;
    }
    return B != NULL;
}
```

**Task 3** (5 pts). What is the asymptotic complexity of determining if a bit string of length \( k \) is in a trie with \( n \) bit strings stored in it? Express your answer in the big-O notation.

\[ O(\_\_\_\_\_\_ k \_\_\_\_\_\_\_\_\_) \]
Task 4 (10 pts). Write a function `bit_uint` that can be used to extract the $i$th bit from an unsigned integer $x$. For it to be used as an argument to `bt_member` we pass a pointer $w$ to the unsigned integer and not the integer itself. You should not assume any particular size for unsigned ints, but you may assume that `<stdbool.h>`, `<stdlib.h>`, and `<limits.h>` have been included. Make sure all your operations are safe, assuming the preconditions are satisfied.

```c
bool bit_uint(void* w, int i) {
    REQUIRES(w != NULL);
    REQUIRES(0 <= i && i < 8*(int)sizeof(unsigned int));
    return (*(unsigned int*)w & (1 << i)) != 0;
}
```

Task 5 (5 pts). Assume we have properly initialized a binary trie $B$ and inserted some elements into $B$. Show how you would call it to verify that the unsigned int 57 is in the trie. Feel free to add some declarations or statements before the `assert`. For full credit, you should not allocate anything on the heap.

```c
int main() {
    bt *B = ...;
    ...

    unsigned int u = 57;
    assert(bt_member(B, &u, &bit_uint, 8*sizeof(unsigned int)));
}
```
6 Union-Find (45 points) [C]

When the number of nodes is fixed, it is convenient and efficient to implement a union-find data structure as an array. However, in many situations nodes are added dynamically during the computation. In such situations we can represent the union-find data structure using pointers.

We have the following invariants:

1. The canonical representative of an equivalence class points to itself. We say this is a node at depth 0.

2. A member $u$ of an equivalence class at depth $n > 0$ points to another node $w$ at depth $n - 1$. We refer to $w$ as the parent of $u$ since the nodes in the equivalence class form a tree with the canonical representative at the root.

```c
struct eq_node {
    void *data;
    struct eq_node *parent;
};

typedef struct eq_node eq;
```

**Task 1** (15 pts). In the pointer-based representation, we may not have a handle on every element in an equivalence class or its depth, so we cannot easily check for global consistency of our data structure. However, we can check if a given node is a valid member of an equivalence class. It is acceptable if your implementation does not terminate on some nodes that are invalid, analogous to our potentially nonterminating `is_segment` function for linked lists.

```c
bool is_eq(eq *u) {
    while (u != NULL && u != u->parent)
        u = u->parent;
    return u != NULL;
}
```
Task 2 (5 pts). When we create a new element, it will always be in its own equivalence class.

```c
eq *eq_new(void *e) {
    eq *u = xmalloc(sizeof(struct eq_node));
    u->data = e;
    u->parent = u;
    return u;
}
```

Task 3 (15 pts).
Write `eq_find` to find the canonical representative of an equivalence class. Your function should be recursive. For full credit, it should perform path compression so that every node on the path subsequently points to its canonical representative.

```c
eq *eq_find(eq *u) {
    REQUIRES(is_eq(u));
    eq *r = u;

    if (r != r->parent) {
        r = eq_find(u->parent);
        u->parent = r;
    }
    ENSURES(is_eq(r));
    return r;
}
```

Task 4 (10 pts).
Write `eq_union` to join two equivalence classes. Since this simple version does not track depth information, you can break the tie in an arbitrary way.

```c
void eq_union(eq *u, eq *w) {
    REQUIRES(is_eq(u) && is_eq(w));
    u = eq_find(u);
    w = eq_find(w);

    u->parent = w;
    return;
}
```