15–122: Principles of Imperative Computation

Midterm Examination 2

November 3, 2011

- There are 19 pages in this examination, comprising 7 questions worth a total of 109 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- DO NOT discuss the test content with anyone outside of this room until tests are graded. Sharing of any information regarding this test, will be considered a violation of the course policy regarding assignments.
- You may refer to one double-sided 8.5" x 11" page of notes, but to no other person or source, during the examination.

Full Name: _____

Andrew ID: _____

Recitation: _____

Question:	1	2	3	4	5	6	7	Total
Points:	8	13	13	15	10	25	25	109
Score:								

1. Memory and Pointers

(2) (a) What does the following function return?

```
int fun2() {
    int *p = alloc(int);
    int *q = alloc(int);
    *p = 3;
     q = p;
    *p = 2;
    return *q;
}
```

Solution:

(2) (b) Identify the error caused by the following code segment by circling the line that will give either a compiler or runtime error. Write, in the space provided, what kind of error you would get, and what you would need to add or change to fix this error.

```
struct node {
     int[] A;
     int size;
};
struct node* ptr = alloc(struct node);
ptr->size = 10;
for (int i=0; i<ptr->size; i++)
   ptr->A[i] = 0;
```

Solution:

(2) (c) In C0, a string is stored internally as `char[]` with a special terminating character `'\0'` as the last element in the array.

Recall that a `char` is 1 byte in memory, an `int` is 4 bytes in memory and an array takes ($n\times$ size of each element) memory, plus two `int`s (one for size, and one for length).
How many **bytes** does the string "15122" require?

Solution:

(2) (d) How many **bytes** would an array of n strings of m-letter words require? You don't have to simplify.

Solution:

2. **Heapify Forever**

Suppose you would like to use a priority queue to keep track of pending jobs *over the course of the lifetime of your machine.* There are several design goals for this implementation of priority queues:

- Different users will insert/delete/deletemin from the priority queue
- Priorities must be unique within the priority queue
- When a user inserts a job, they do not specify a priority. Instead, your algorithm must select for the job the highest unused priority not currently in the PQ.
- Since this data structure will exist for many years, reassignment of deleted priorities will be necessary.

(4) (a) Choose an *auxiliary* data structure to keep track of all of the priorities currently used in the PQ. Please find a structure which will support (a) insertion of keys used, and (b) deletion of keys which are no longer part of the PQ. Both insertion and deletion of keys should be most $O(1)$ amortized time. *Hint: What data structure has these runtime complexities?*

Solution:

(4) (b) You may have noticed something funny about the data structure you've chosen above. What we actually wanted was to keep track of those keys **not used in** the PQ, and we wanted to have fast search for the *smallest priority not currently used in the PQ*. Find a data structure which keeps track of the set of **unused priorities** (assuming we're using $[0, MAX_{INT}]$ as possible priorities), supports $O(\log n)$ insertion and deletion of keys, and can return the smallest unused priority in time $O(\log n)$.

Solution:

(5) (c) Given your data structure from part (b), describe *in words, not code* how to implement a long-living PQ with unique priorities. Please explain the updates necessary to the PQ and your data structure to implement `insert`, `delete` and `deletemin`.

Solution:

3. **Perfect Binary Trees**

Recall the definition of perfect (unordered) binary trees, which contain the maximum possible number of nodes for a given height.

Definition. *Perfect binary tree t with height h.*

1. *An empty tree cannot be perfect.*
2. *If $h = 0$, then the left and right subtrees must be empty.*
3. *Otherwise, $h > 0$, and both subtrees are perfect binary trees of height $h - 1$.*

(7) (a) Write the **specification** function `bool is_perfect_tree(tree T, int height)`, which is total (that is, it returns a value on all inputs) and returns `true` if and only if the tree T, of height h, is a perfect tree, according to the above definition. You may assume that h is the correct height of the tree T.

Solution:

```
bool is_perfect_tree(tree T, height h)
{









}
```

(3) (b) Write the **specification** function `bool is_perfect(bst B)`, which is total (that is, it returns a value on all inputs), and returns `true` if and only if the binary search tree `B` is perfect, according to the above definition. You may use the function

```
int height(tree T)
```

that returns h_T, the height of tree T.

Solution:

```
bool is_perfect(bst B)
{









}
```

(3) (c) How does having a perfect binary tree allow us to reason about the complexity of tree operations? **Hint:** What can we assume if a tree is perfect?

Solution:

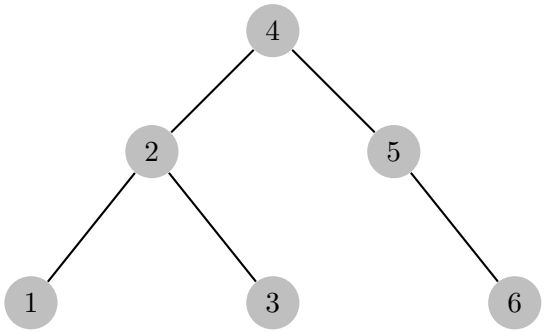
4. **Reversing Trees**

In this question, you will write a function that takes a binary search tree and produces a mirror image of it. Recall that we implement BSTs with the following data types:

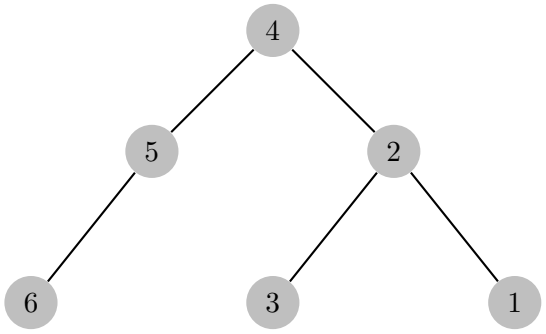
```
typedef struct tree_node *tree;
struct tree_node {
    elem data;
    tree left;
    tree right;
};

typedef struct tree_root *bst;
struct tree_root {
    tree root;
};
```

Given a `bst`, your function should *recursively* reverse the order of the children of its root node. This should be done in place. For example, the mirror image of the tree:



Would be



Note that an in-order traversal of the new, reversed tree should be exactly the reverse of an in-order traversal of the original tree.

(10) (a) Implement the **recursive**, **in-place** helper function `tree_mirror`. You don't need to worry about contracts.

```
void bst_mirror(bst B)
//@requires is_bst(B);  //note: we assume these functions do NOT
//@ensures is_bst(B);   //check for least-to-greatest order
{
    tree_mirror(B->root);
}
```

Solution:

```
void tree_mirror(tree T)
{









}
```

(5) (b) How would you change `tree_mirror` (and `bst_mirror`), such that the functions no longer reverse in place, and instead **return** the reversed tree without modifying the input? *N*o code is required.

Solution:

5. **Trashtables**

Recall our implementation of hash tables, in which we resolve hash collisions using linked lists as chains. Imagine, instead, a hash table, in which chains are implemented using *binary search trees*. This is called a `trashtable`. We define a `trashtable` as follows:

```
struct trashtable {
    int size;
    bst[] A;
};
```

using the implementation of `bst` covered in lecture.

(5) (a) Given the definition of `trashtable`s above, and your knowledge of hash tables, how might a `trashtable` **improve** lookup and insertion times?

Solution:

(5) (b) For hash tables, as implemented in class, we require a function that takes two keys and reports if they are equal or not (for `hmap`s, this is `hmap_ktype_equal`, and for `hset`s, this is `hset_elem_equal`). Why isn't this sufficient for `trashtable`s? What else do we need?

Solution:

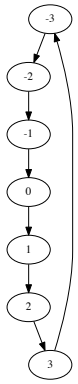
6. Groops

Recall the definition of linked lists and the `is_segment` function that checks if a linked list beginning at `start` eventually arrives at `end`.

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

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

A *groop* is a linked list which is terminated by a pointer back to the `start`. A `groop` with items $-3, -2, -1, 0, 1, 2, 3$ (in this order, -3 at the front of the groop and 3 at the back) would be represented as the following groop (where the arrows represent the `next` pointers):



We define

```
struct elem_t {
  int data;
  struct elem_t *inverse;
  struct elem_t *next;
}
typedef struct elem_t* elem;
struct groop_t {
  elem start;
  elem identity;
};
typedef struct groop_t* groop;
```

As an example, here is a function that checks if the groop has exactly one element (a groop is defined only when it has at least one element).

```
bool groop_has_one_elem(groop G)
//@requires is_groop(G);
{
  return G->start == G->start->next;
}
```

(10) (a) Write a function that checks if a given groop has exactly one cycle: that is, the last element points to the first, and no other `e-> next` pointers reach a node between `start` and `e`. [**Hint:** Keep track of what node you've seen using a hash table for the elements' addresses. The only two functions you have available are `void insert(elem value)`, which will take care of making and hashing keys from values, and `bool lookup(elem value)`, which will return true iff a value has been inserted into the hash table.]

Solution:

```
bool has_right_cycle(groop G) {









}
```

(5) (b) Write a function which checks the other two important properties of a groop:(1), that for every elem `e` in the groop `G`,

`e->data + G->identity->data == e->data`

and (2), that for every `e` in `G`

`e->data + e->inverse->data == G -> identity`

Solution:

```
bool has_identity_and_inverses(groop G)
//@requires has_right_cycle(G);
//@ensures has_right_cycle(G);
{









}
```

(5) (c) It is also the case that the any groop has `e->inverse->inverse == e`. Write a function to check this.

Solution:

```
bool inverses_idempotent(groop G);
//@requires has_right_cycle(G);
//@requires has_identity_and_inverses(G);
//@ensures  has_right_cycle(G);
//@ensures  has_identity_and_inverses(G);
{



}
```

(5) (d) The final (and tricky) part of a groop is that for every element `e` in the groop, `e->inverse` is part of the groop. Please describe an algorithm *in words; no code is required* to check this property. [**Hint:** Keep track of what nodes you need to see using a list, which you will add to when you have a new inverse to find, and which you will remove from when you find the inverse. Also note that the previous problem means that you don't need to add the inverse of an inverse to the list to find.]

Solution:

7. **Reverse Polish Notation**

Ordinary arithmetic is typically represented in "infix" notation (e.g. `(2 + 2) * 4`). The pattern expressed by infix is, in general `EXP OP EXP` where `EXP` is an expression that evaluates to a single number and OP is a binary operator. However, infix notation is somewhat difficult for computers to parse, because it introduces parentheses to control the order of operations. For example, the expressions `2 + 2 * 4` and `(2 + 2) * 4` evaluate to 10 and 16 (and $10 \neq 16$) respectively.

There is another notation that is significantly easier for computers to parse known as "reverse-polish notation". In reverse-polish notation, the pattern is EXP EXP OP. Note that the order of operands is left to right. So the expression `x y -` would be `x - y` in infix. As an example, the infix expression `(2+2)*4` would be represented as `2 2 + 4 *` in reverse-polish notation. For the purposes of this question, we will restrict the set of operations to `*, +, /, -`, and the set of numerical literals to the integers.

We represent the input expression in reverse polish notation as an array of `string`s. We also provide a library function `string_to_int` which converts a `string` to an `int`. The return value of this function is undefined (could be anything) if the `string` cannot be converted. This function has the following signature:

```
int string_to_int(string s);
```

Our implementation will also require a stack, which has the following interface:

```
typedef struct stack * stack;
stack stack_new();
int pop(stack S);
void push(int e, stack S);
```

(10) (a) Complete the following implementation of a reverse-polish notation evaluator.

```
int evaluate(string[] expr, int n)
//@requires \length(expr) == n;
{
  struct stack* S = stack_new();
  int result;

  for (int i = 0 ; i < n; i++)
  {
    if (string_equal(expr[i], ____))
    {
      int a = __________;
      int b = __________;
      __________________;
    }
    else if (string_equal(expr[i], ____))
    {
      int a = __________;
      int b = __________;
      __________________;
    }
    else if (string_equal(expr[i], ____))
    {
      int a = __________;
      int b = __________;
      __________________;
    }
    else if (string_equal(expr[i], ____))
    {
      int a = __________;
      int b = __________;
      __________________;
    }
    else
    {
      int a = __________________;
      __________________;
    }
  }
  result = _________________;
  return result;
}
```

(5) (b) Consider the reverse-polish notation expression `7 2 * 3 -`. Write out the state of the stack after processing each token. Your final state should be a stack with only the final result of the expression on it. Use the physical top of the stack to mean the actual top of the stack.

Index of token processed	0	1	2	3	4
Stack State					

(5) (c) Given a valid reverse-polish notation expression with k binary operators(i.e., tokens equal to "+", "-", "/" or "*"), how many stack operations (only counting `push` and `pop`) does your evaluator perform as a function of k? Explain briefly.

Solution:

(5) (d) Give an example of an invalid expression in reverse-polish notation, and describe how your evaluator would fail to evaluate it (**Hint:** try evaluating an infix expression).

Solution: