Midterm 2 Exam

15-122 Principles of Imperative Computation

Thursday 2nd April, 2020

Name: _____

Andrew ID: _____

Recitation Section: _____

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 5 problems on 15 pages (including 2 blank pages at the end).
- 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.

	Max	Score
Amortized Cost	10	
Linked Lists	30	
Hash Tables	25	
Trees	30	
Priority Queues	30	
Total:	125	

1 Amortized Cost (10 points)

Consider a data structure where we perform n operations. The cost of the i^{th} operation is i if i is a power of 2; otherwise, the cost of operation i is 1.

Task 1.1 If we were to use standard worst case analysis, we would say that the longest single operation has a cost in

$O($_____$)$

Since there are n operations, this would lead us to say that the worst case runtime complexity for the entire series of n operations is in

$O($_____$)$

But this doesn't seem like a tight bound on the n operations: the worst case scenario does not occur for each of the n operations. We can use amortized analysis using $tokens$ to determine the overall cost of n operations in this scenario.

Suppose we pay 3 tokens for each operation. Tokens that are not used to pay for the operation are banked. Complete the following table that shows the number of tokens banked after each operation, for n = 16.

Task 1.2

Operation	Operation Cost	Tokens Banked After Operation
1	1	2
2	2	3
3	1	5
4	4	4
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		

Task 1.3 Based on our analysis, it seems that the number of banked tokens never falls below

so the amount that we prepay for each operation is enough to pay for future operations.

Task 1.4 Therefore, we can say that the total amortized cost for n operations, as a function of n, is

tokens, which is in

$O($_____$)$

Task 1.5 The amortized cost per operation is in

$O($_____$)$

2 Linked Lists (30 points)

This question deals with linked lists, given by the following definitions:

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

typedef struct linkedlist_header linkedlist;
struct linkedlist_header {
    list* start;
    list* end;
};
```

Any valid linked list structure has a list segment from the `start` to the `end`, as described by the `is_segment` data structure invariant. An empty linked list structure consists of one struct `linkedlist_header` where the `start` and `end` point to the same dummy node.

```
1 bool is_segment(list* start, list* end) {
2     if (start == NULL || end == NULL) return false;      // NONE
3     if (start == end) return true;                       // NONE
4     if (start->next == end) return true;                 // 2
5     return is_segment(start->next, end);                 // 2
6 }
```

The above code has annotations to the right which explain which lines are needed to justify safety for the given line.

5pts **Task 2.1** Finish the `is_sorted` function that returns true if the linked list contains a non-decreasing series of integers. Respect the loop invariants. You don't have to use every line.

```
bool is_sorted(linkedlist* T)
//@requires T != NULL;
//@requires is_segment(T->start,T->end);
{
    _____

    _____

    for (list* p = T->start; p->next != T->end; p = p->next)
    //@loop_invariant p != T->end && is_segment(p, T->end);
    {
        _____

        _____

    }

    _____
}
```

10pts **Task 2.2** As in the written homework on pointer safety and the example on the previous page, indicate to the right of each line below which line number(s) need to be referenced to show that the line of code to the left is *safe*.

Your analysis must be precise and minimal: only list the line(s) upon which the safety of a pointer dereference depends. Don't use the lack of a memory error on one line to prove safety on a later line.

```
21 void mystery(linkedlist* T)
22 //@requires T != NULL;                              // NONE
23 //@requires T->start != T->end;                     // _____
24 //@requires is_segment(T->start,T->end);            // _____
25 //@ensures is_segment(T->start,T->end);             // _____
26 {
27   list* A = T->start->next;                         // _____
28   T->start->next = T->end;                          // _____
29   while(A != T->end)                                // _____
30   //@loop_invariant is_segment(T->start, T->end);   // _____
31   //@loop_invariant is_segment(A, T->end);          // _____
32   {
33     list* B = A->next;                              // _____
34     A->next = T->start;                             // _____
35     T->start = A;                                   // _____
36     A = B;                                          // _____
37   }
38 }
```

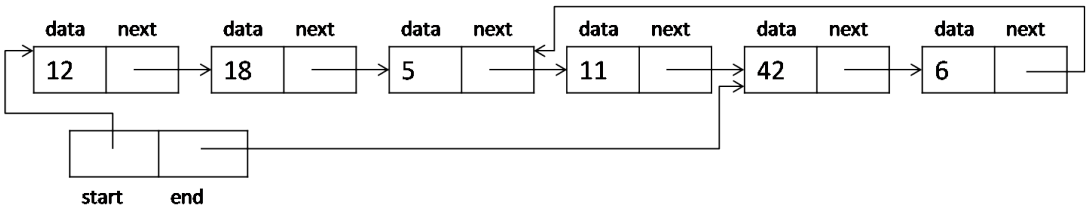
5pts **Task 2.3** Explain why the loop invariants on line 30 and line 31 of the mystery function are true *initially*. Cite line numbers and also give brief explanations.

5pts **Task 2.4** Prove that the loop invariant on line 31, `is_segment(A, T->end)`, is preserved by an arbitrary iteration of the loop. Cite line numbers and also give brief explanations.

You do *not* have to prove the loop invariant on line 30 is preserved.

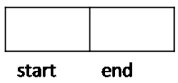
To show: `is_segment(A', T->end)`

5pts **Task 2.5** Illustrate the result of running the mystery function on this pointer structure:



Either write "non-termination," "assertion failure," or "memory error" in the box below, or else draw the pointer structure after the mystery function runs.

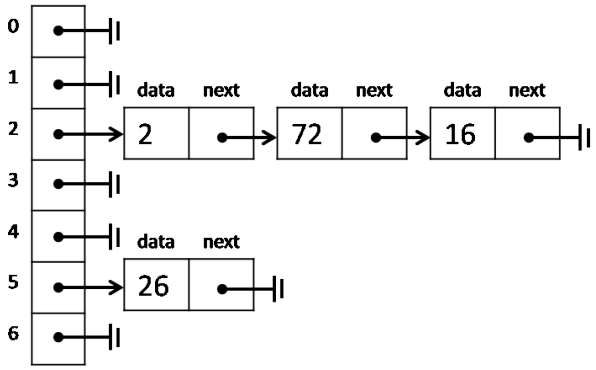
You do not have to label the "data" and "next" fields in your illustration.



3 Hash Tables (25 points)

In the questions on this page, we will consider adding integers to a **separate-chaining hash table with a capacity of 7 that does *not* resize**. Our hash table will use the rather unfortunate hash function $hash(x) = x$.

Here's one such hashtable after 16, 72, 26, and 2 have been added:



3pts **Task 3.1** What's the exact (not big-O) load factor of the hash table above?

3pts **Task 3.2** Give 4 distinct nonnegative integers that, if added to an initially-empty hash table in any order, will ensure that attempting to look up 3 in the hash table will take as long as possible.

3pts **Task 3.3** The integer 3 should *not* be part of your answer in the above task. Why not?

3pts **Task 3.4** Give 8 distinct nonnegative integers that, if added to an initially-empty hash table in any order, will minimize the worst-case time of attempting to look up *any* integer in the hash table.

3pts **Task 3.5** For the 8 integers you added in part Task 4, give an integer that takes the worst-case time to look up in the hash table. (Your answer should be the same regardless of the order those 8 elements are added, and your answer needn't be in the table.)

For the three tasks on this page, we will imagine we are using a hash set to implement the kind of analysis we did in the DosLingos homework assignment. Our hash set elements are structs containing a word and a frequency count; we want to read in a series of words (like the Scrabble dictionary) and then update the frequency counts for those words as we see them in our corpus (the complete works of Shakespeare). Therefore, two structs will be treated as equivalent if they contain the same word.

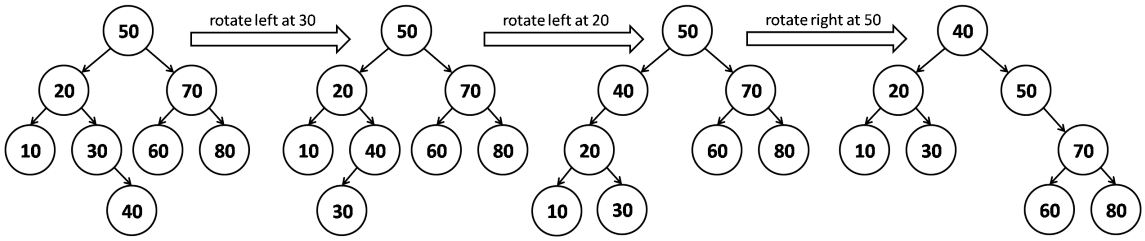
4pts **Task 3.6** Why would it be a bad idea to have the hash function *always return zero*?

4pts **Task 3.7** Why would it be a bad idea to have the hash function depend on the *frequency counts* for those words?

2pts **Task 3.8** Which of these two bad ideas is worse, and why?

4 Trees (30 points)

14pts **Task 4.1** In *splay trees*, when a new element is inserted into a tree using the usual BST insertion algorithm, rotations are then performed to make the newly-added element the *root* of the post-insertion tree. This is the series of rotations that might be performed after the insertion of 40 into the tree, for example:



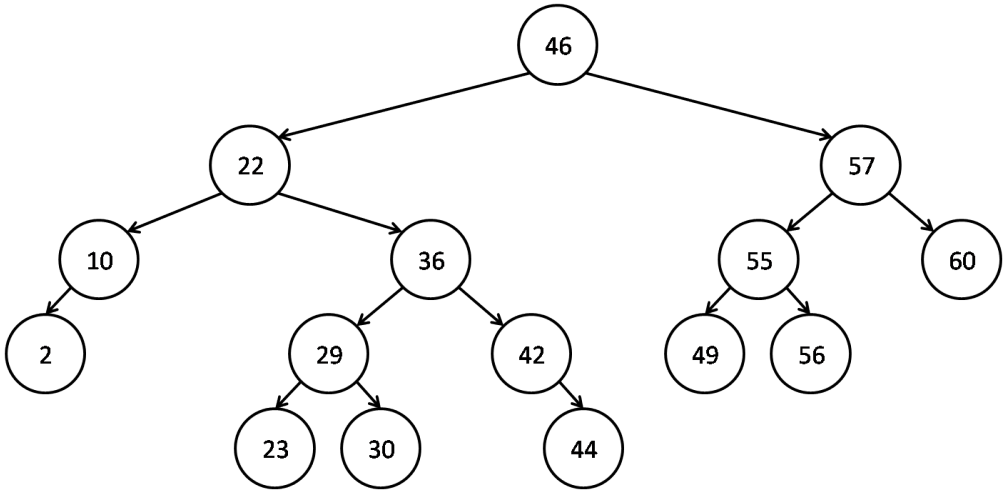
Finish the code below for splay tree insertion, using the rotation functions discussed in class:

```
tree* rotate_left(tree* T)  /*@requires T != NULL && T->right != NULL @*/ ;
tree* rotate_right(tree* T) /*@requires T != NULL && T->left != NULL @*/ ;
```

Make sure that, for all the code you write, you can use contracts to prove the absence of `NULL` pointer dereferences or precondition violations! The function `leaf(x)` allocates a new leaf node with data `x`.

```
tree* tree_insert(tree* T, elem x, elem_compare_fn* comp)
//@requires is_tree(T);
//@ensures is_tree(\result);
//@ensures \result != NULL;
{
  if (T == NULL) return leaf(x);
  int r = (*comp)(x, T->data);
  if (r == 0) {
    T->data = x;                /* modify in place */
  } else if (r < 0) {           /* x < T->data     */

    _____________________________________________________________;

    _____________________________________________________________;
  } else {                      /* x > T->data     */

    _____________________________________________________________;

    _____________________________________________________________;
  }
  return T;
}
```

16pts **Task 4.2** Consider the following AVL tree where elements are integers.



In this question, we consider the insertion of several elements into this tree. ***Each insertion is into the tree above, the insertions are not cumulative.*** Record ***all*** the nodes where a regular BST insertion causes AVL balance invariant to be violated (there may be none, or more than one!). If violations occur, list the sequence of rotations the AVL insertion algorithm will perform tree to restore the invariant.

We've given you the first two examples.

Inserting 1 causes a violation at *10*.
This is fixed (if necessary) by the following rotation(s): *rotate right at 10*.

Inserting 17 causes a violation *nowhere*.
This is fixed (if necessary) by the following rotation(s): *N/A*.

Inserting 31 causes a violation _____
This is fixed (if necessary) by the following rotation(s):

_____

Inserting 45 causes a violation _____
This is fixed (if necessary) by the following rotation(s):

_____

Inserting 47 causes a violation _____
This is fixed (if necessary) by the following rotation(s):

_____

Inserting 62 causes a violation _____
This is fixed (if necessary) by the following rotation(s):

_____

5 Priority Queues (30 points)

This question deals with unbounded, generic priority queues implemented as heaps. These priority queues avoid getting full by using the unbounded array trick of doubling the size of their internal array when the heap gets full; the client doesn't need to know how this works.

```
/*** Client interface for priority queues ***/
typedef void* elem; // Library treats this as an unknown pointer type

// f(x,y) returns true if x is STRICTLY higher priority than y
typedef bool higher_priority_fn(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

/*** Library interface for priority queues ***/
// typedef ______* heap_t;

bool heap_empty(heap_t H)
  /*@requires H != NULL; @*/ ;

heap_t heap_new(higher_priority_fn* prior)
  /*@requires prior != NULL; @*/
  /*@ensures \result != NULL && heap_empty(\result); @*/ ;

void heap_add(heap_t H, elem x)
  /*@requires H != NULL && x != NULL; @*/ ;

elem heap_rem(heap_t H)
  /*@requires H != NULL && !heap_empty(H); @*/
  /*@ensures \result != NULL; @*/ ;
```

We'll also use a simple interface for reading words from a file:

```
/*** Interface for files ***/
// typedef ______* file_t;
bool has_more_words(file_t F)
  /*@requires F != NULL; @*/ ;
string get_next_word(file_t F)
  /*@requires F != NULL && has_more_words(F); @*/ ;
```

20pts **Task 5.1** Unbounded arrays of strings are useful when we have a file and we need to read in all the words from a file in order without knowing how many words are in the file:

```
arr_t read_in_inputs(file_t F)
//@ensures \result != NULL;
{
    arr_t A = arr_new(0);
    while (has_more_words(F)) {
        string s = get_next_word(F);
        arr_add(A, s);
    }
    return A;
}
```

Implement the `read_in_inputs` function using ***only*** the `heap` interface and the `file` interface. ***Your function, like the UBA-using function above, should return an array where the strings are in the same order as they were in the file.***

You'll need to define some helper structs and functions first. Make sure that all your operations are ***provably safe***. If you can't prove that an operation is safe from the loop invariants and preconditions, use an `//@assert` statement to ensure safety.

```
// You can use the blank pages in the back if you need more space for
// helper structs and functions, but clearly indicate this!









string[] read_in_inputs(file_t F)
//@requires F != NULL;
{
    int n = 0;
    heap_t H = ____________________________________________________;

    while (has_more_words(F)) {









    }
    //@assert n >= 0;
    string[] A = alloc_array(string, n);
    for (int j = 0; j < n; j++)
    //@loop_invariant 0 <= j;
    {









    }
    return A;
}
```

10pts **Task 5.2** Heaps are implemented as they were implemented in class:

```
typedef struct heap_header heap;
struct heap_header {
  int limit;                  /* limit > 1 */
  int next;                   /* 1 <= next && next <= limit */
  elem[] data;                /* \length(data) == limit */
  higher_priority_fn* prior; /* != NULL */
};
```

Write a simple *recursive* function (***no loops!***) that searches for an element in an heap; if multiple equivalent elements exist, any one of them can be returned.

Assume that two equivalent elements will always have the same priority. Use this assumption and the heap ordering invariant to make your search as efficient as possible (though it will still have worst case performance in $O(n)$). Add preconditions to the helper function to ensure safety, if necessary.

```
typedef bool elem_equiv_fn(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

elem heap_search_helper(heap_t H, elem x, elem_equiv_fn* equiv, int i)
//@requires is_heap(H) && x != NULL && equiv != NULL;

// Returns NULL if no elements equivalent to x are in the heap
elem heap_search(heap_t H, elem x, elem_equiv_fn* equiv)
//@requires is_heap(H) && x != NULL && equiv != NULL;
{
  return heap_search_helper(H, x, equiv, 1);
}
```

(This page intentionally left blank.)

(This page intentionally left blank.)