Midterm 2 Exam

15-122 Principles of Imperative Computation

Tuesday 15th November, 2022

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 16 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	
Universal Search	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 }
```

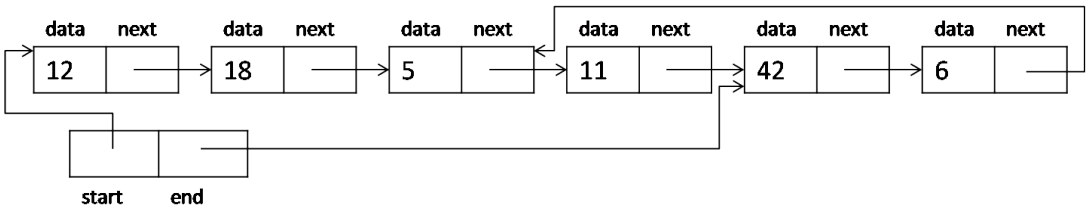
5pts **Task 2.3** Explain why the loop invariants on line 30 and line 31 of the mystery function are true *initially*. Cite line numbers and also give brief explanations.

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.

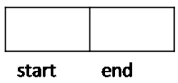
To show: `is_segment(A', T->end)`

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.

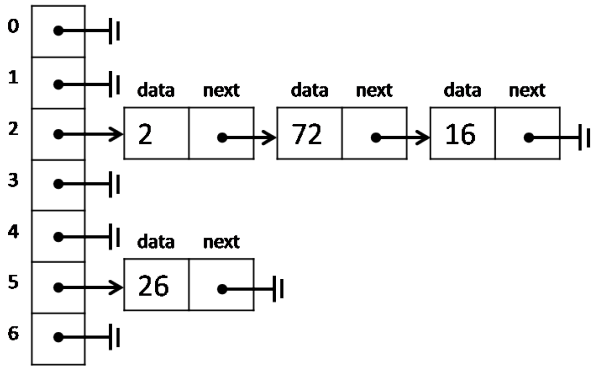
You do not have to label the "data" and "next" fields in your illustration.



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 dictionary to implement the kind of analysis we did in the DosLingos homework assignment. Our hash dictionary entries 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.

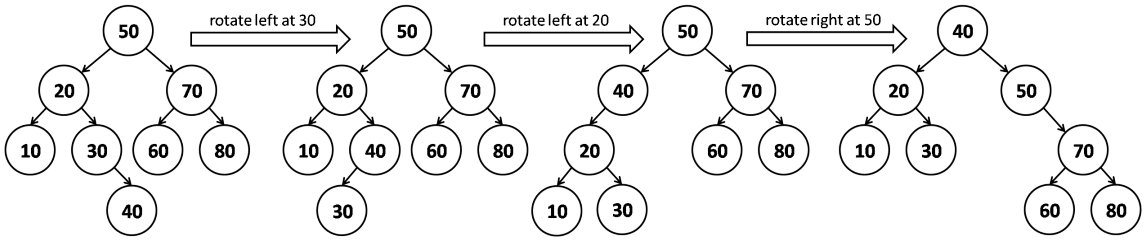
4pts **Task 3.6** Why would it be a bad idea to have the hash function *always return zero*?

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:



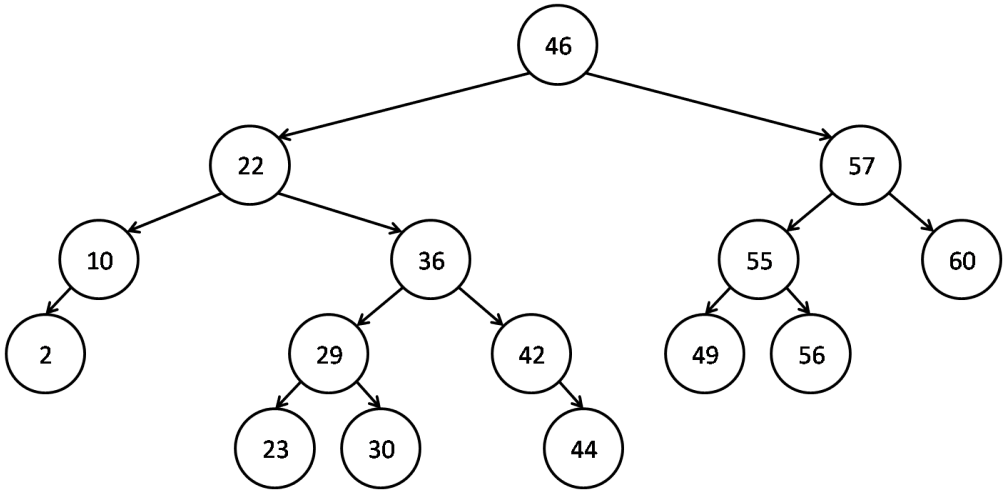
Finish the code below for splay tree insertion, using the rotation functions discussed in class:

```
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 Universal Search (30 points)

In this question, we will extend the **hash dictionary library** with a new operation, `hdict_iterate`, which will allow us to find an entry in a dictionary on the basis of any criteria we want. The interface for generic hash dictionaries seen in class is reproduced on page **??** of this exam.

The prototype of `hdict_iterate` and relevant types are as follows:

```
typedef void* entry;

typedef entry pick_fn(entry x, entry y);

entry hdict_iterate(hdict_t H, entry base, pick_fn* f)
/*@requires H != NULL && f != NULL; @*/ ;
```

Here, `pick_fn` is the type of any function that takes two entries as input and returns an entry. If the hash dictionary `H` contains entries `e1`, `e2`, ..., `en`, then the call `hdict_iterate(H, base, f)` returns the value computed by

```
f(... f(f(base, e1), e2) ..., en)
```

that is, it computes the result of `f(base, e1)`, then calls `f` on that and `e2`, and so on all the way to `en`. If `H` is empty, `hdict_iterate(H, base, f)` simply returns `base`.

We will now **implement** `hdict_iterate` as part of the hash dictionary library. Recall the type declarations for hash dictionaries seen in class:

```
typedef struct chain_node chain;
struct chain_node {
  entry data;  // != NULL
  chain* next;
};
```

```
typedef struct hdict_header hdict;
struct hdict_header {
  chain*[] table; // \length(table) == capacity
  int capacity;   // 0 < capacity
  // OMITTING FIELDS IRRELEVANT TO THIS QUESTION
};
typedef hdict* hdict_t;
```

You may also assume that the specification function `is_hdict` for hash dictionaries has been implemented for you.

11pts **Task 5.1** Complete the following code for `hdict_iterate`.

```
entry hdict_iterate(hdict* H, entry base, pick_fn* f)
//@requires is_hdict(H) && f != NULL;
{
  // Initialize result

  entry result = ____________________________________________________;

  // Go over every table index

  for (int i = _____________; ____________________; _____________)

  //@loop_invariant ____________________ && ____________________;
  {
    // Go over every entry in this bucket

    for (chain* p = _____________; _____________; _____________)

      result = ____________________________________________________;
  }
  return result;
}
```

2pts **Task 5.2** Assuming that `f` has constant cost and that `H` has capacity m and contains n entries, what is the complexity of `hdict_iterate(H, base, f)`?

$O($_____$)$

We want to use our extended hash dictionary library in a fancy implementation of a new, super-fast web browser. To speed things up, it stores recently-visited web pages in a hash dictionary: when a user asks for a page it first tries to retrieve it from the hash dictionary instead of downloading it from the Internet (which takes much longer).

The entries stored in the hash table are as follows:

```
struct webcache_entry {      // entry
  string  URL;               // key
  string* page;
  int     last_visited;
};
typedef struct webcache_entry cache_entry;
```

An entry contains the URL of the stored page, the page itself, and the time the page was last visited (an **int** corresponding to the number of seconds since January 1st, 2000 — you do not need to worry about overflow).

10pts **Task 5.3** Implement the function `least_recently_visited(e1,e2)`, of type `pick_fn`, which returns the entry among `e1` and `e2` which has been visited the longest time ago as per their `last_visited` fields. We allow `e1` to be `NULL` to represent a time in the distant future.

The major challenge in this task is correctly switching between the genericity of the type `entry` *and the specificity of the type* `cache_entry`. *Your code should be provably safe.*

```
entry least_recently_visited(entry e1, entry e2)

//@requires _____________________________________________________;

//@requires _____________________________________________________;

//@ensures ______________________________________________________;
{
  if (e1 == NULL) return e2;

  int t1 = _____________________________________________->last_visited;

  int t2 = _____________________________________________->last_visited;

  if (_____________________________________________________) return e1;
  return e2;
}
```

4pts **Task 5.4** Using `hdict_iterate` and `least_recently_visited`, complete the following statement so that the variable `lrv` is set to the entry in hash dictionary `H` that has been visited the least recently, or to `NULL` if `H` does not contain any entry.

```
entry lrv = ___________________________________________________;
```

The function `hdict_iterate` goes over all entries in a hash dictionary. We can use it for a lot more than finding an entry we are interested in. What about printing the URL of every page stored in our dictionary?

3pts **Task 5.5** Complete the following code which will print every entry in the dictionary. Note that `e1` *is* allowed to be `NULL`.

```
entry print_URL(entry e1, entry e2)
//@requires e2 != NULL;
/*@ OMITTING CONTRACTS SIMILAR TO PREVIOUS TASKS @*/
{
  println(_____________________________________________________________);
  return NULL;
}

void print_all_stored_URLs(hdict_t H)
//@requires H != NULL;
{
  ____________________________________________________________________;
}
```

(This page intentionally left blank.)

(This page intentionally left blank.)