Midterm 2 Solutions

15-122 Principles of Imperative Computation

Thursday 2nd April, 2020

Name: Harry Bovik

Andrew ID: bovik

Recitation Section: S

Instructions

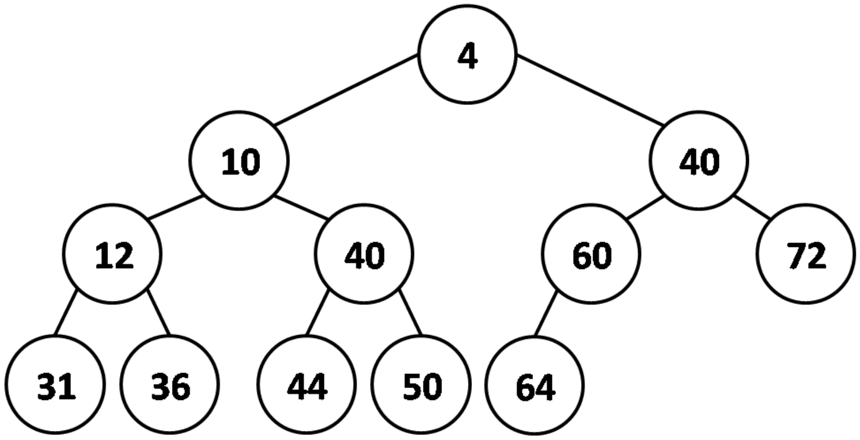
- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems on 21 pages (including 0 blank pages at the end).
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.

	Max	Score
Priority Queues	20	
Heterogeneous Data Structures	31	
Tree Sort	40	
Scanning Hash Tables	45	
Total:	136	

1 Priority Queues (20 points)

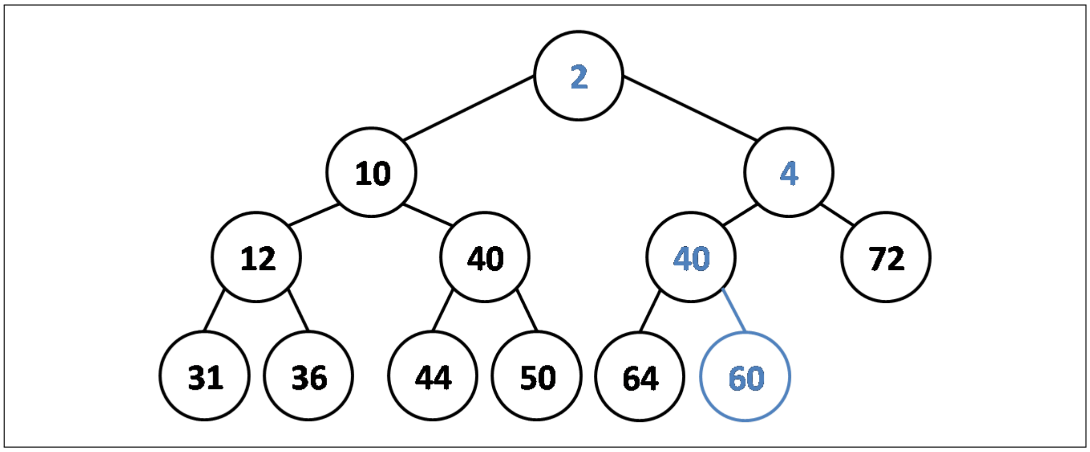
This task is about *priority queues* implemented as *min-heaps*.

Task 1.1 Consider the min-heap shown below. The numbers indicate the priority of a node.



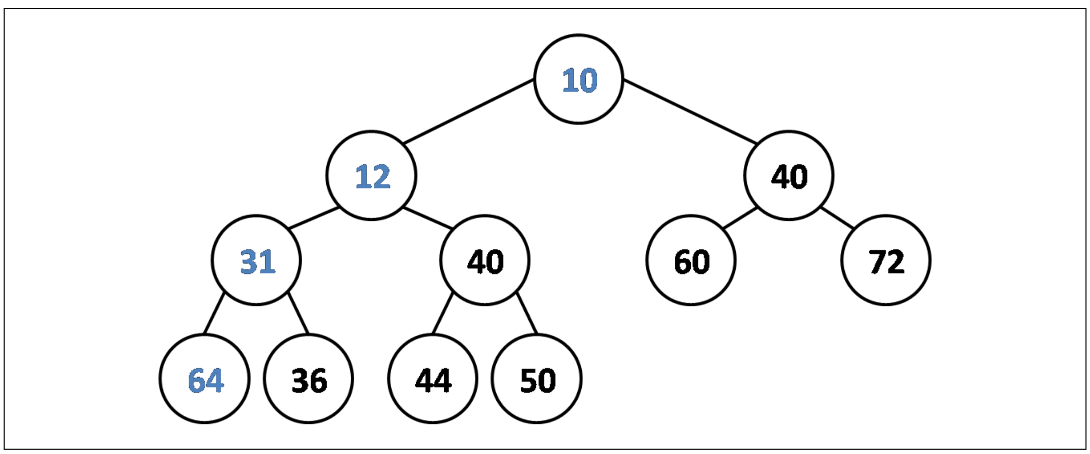
4pts

a. Draw the resulting heap after inserting a new node with priority 2 into the heap above, using the `pq_add` function discussed in class:



4pts

b. Draw the resulting heap after removing the node with the highest priority from the *original* heap above, using the `pq_rem` function discussed in class:



Task 1.2 In the next tasks, you may assume the min-heap implementation of priority queues seen in class.

3pts **a.** What is the asymptotic complexity (tightest and simplest) of using the function `pq_add` to insert n items into an initially empty priority queue? Justify your answer briefly.

$O(\ n \log n\ )$

Because we make n calls, each bounded by $\log n$.

3pts **b.** What is the asymptotic complexity (tightest and simplest) of using the function `pq_rem`, to remove 7 items from an n-element heap (you may assume $n > 7$)? Justify your answer briefly.

$O(\ \log n\ )$

Because we make 7 calls, each bounded by $\log n$.

3pts **c.** What is the asymptotic complexity (tightest and simplest) of calling the function `pq_peek` n times on a non-empty n-element heap? Justify your answer briefly.

$O(\ n\ )$

Because we make n call to an $O(1)$ function.

(Note that this is especially silly since each call returns the same item.)

3pts **d.** When inserting or removing an element, one of the heap invariants is temporarily violated while the other holds throughout. Circle the one that is temporarily violated.

Shape invariant   Ordering invariant

2 Heterogeneous Data Structures (31 points)

In this exercise, we are going to explore *heterogeneous* queues, allowing a client to store elements of *different* types in *one* queue. An immediate thought might be to use `void*` as the type for the queue's elements. However, since `\hastag` can only be used in contracts, but not in code in C1, we would lose the ability to process the elements *depending* on their *type*. To make an element's actual type available to C1 code, we introduce the following struct:

```
struct tagged_elem_header {
  int tag; // 0 = int*, 1 = string*, 2 = bool*
  void* value;
};
typedef struct tagged_elem_header tagged_elem;
```

The field `tag` describes the type of the element and the field `value` its value. We use the integer `0` for type `int*`, the integer `1` for type `string*`, and the integer `2` for type `bool*`.

7pts **Task 2.1** Complete the function `new_tagged_string`, which creates a new tagged element. Make sure that your implementation satisfies the given contract:

```
tagged_elem* new_tagged_string(string s)
//@ensures \result != NULL;
//@ensures \result->tag == 1;
//@ensures string_equal(*(string*)(\result->value), s);

//@ensures \hastag(            string*            , \result->value);
{
  tagged_elem* new =            alloc(tagged_elem)            ;

  string* s_ptr = alloc(string)                               ;

  *s_ptr = s                                                  ;

  new->tag = 1                                                ;

  new->value = (void*)s_ptr                                   ;

  return new;
}
```

In addition to the function `new_tagged_string` that you have just implemented, you can assume the existence of analogous functions `new_tagged_int` and `new_tagged_bool`, with the following signatures and with contracts analogous to `new_tagged_string`'s:

```
tagged_elem* new_tagged_int(int i);
tagged_elem* new_tagged_bool(bool b);
```

Here is some C1 code that uses these functions:

```
1 tagged_elem* elem1 = new_tagged_string("Cogito ergo sum.");
2 //@assert \hastag(string*, elem1->value);
3 tagged_elem* elem2 = new_tagged_int(122);
4 //@assert \hastag(bool*, elem2->value);
5 tagged_elem* elem3 = new_tagged_bool(true);
6 //@assert \hastag(void*, elem3->value);
7 int i = *(int*)(elem2->value);
8 int t3 = elem3->tag;
```

5pts **Task 2.2** Given the above code, fill in the blanks:

The assert statement on line 2 evaluates to ______ true ______

The assert statement on line 4 evaluates to ______ false ______

The assert statement on line 6 evaluates to ______ false ______

The integer `i` on line 7 evaluates to ______ 122 ______

The integer `t3` on line 8 evaluates to ______ 2 ______

2pts **Task 2.3** Given the above code, complete the following statements:

UNUSED

```
int x = ______ *(int*) ______(elem2->value);

bool* y = ______ (bool*) ______(elem3->value);
```

6pts **Task 2.4** Complete the function `print_elem` that prints the `value` field of input `T`. Use the appropriate print function from the `conio` library (see page 20 for a reference) for each possibility for the field `tag`.

```
void print_elem(tagged_elem* T)
//@requires T != NULL;
{
  if (T->tag == ___0___)

    ___printint(*((int*)(T->value)))___;

  else if (T->tag == ___1___) {

    ___print(*((string*)(T->value)))___;

  else if (T->tag == ___2___) {

    ___printbool(*((bool*)(T->value)))___;

  else error("Unknown tag");
}
```

2pts **Task 2.5** The interface of queues is recalled on page 20 of this exam. Complete the below type definition to make the queue store pointers to `tagged_elem` instances:

```
typedef ___tagged_elem*___ elem;
```

1pt **Task 2.6** Here is some C1 code that uses the function `print_elem` that you have just implemented:

UNUSED

```
queue_t Q = queue_new();
enq(Q, new_tagged_int(213));
enq(Q, new_tagged_bool(true));
enq(Q, new_tagged_string("Cogito ergo sum."));
print_elem(deq(Q)); println("");
```

What will this code print? Write your answer in the box below:

213

5pts **Task 2.7** Define the type `print_elem_fn` of functions that print values of type `elem`, and use it to implement the function `print_queue(Q, f)` that prints the contents of the queue `Q` using print function `f`. Calling this function destroys the queue.

```
typedef ___void___ print_elem_fn ___(elem e)___;

void print_queue(queue_t Q, print_elem_fn* f)
```

```
//@requires Q != NULL;
{
  while (!queue_empty(Q))
    (*f)(deq(Q));

}
```

3pts **Task 2.8** Using `print_elem`, complete the function `print_queue` that prints the content of the queue passed as an argument. The queue interface is recalled on page 20.

UNUSED

```
void print_queue(queue_t Q)
//@requires Q != NULL;
{
  while (!queue_empty(Q))
    output_elem(deq(Q));

}
```

3 Tree Sort (40 points)

Rob learned about binary search trees (BST) this week, and that sparked an idea about a new algorithm to sort an array: insert all elements into a BST and read them off from smallest to biggest, something he was told is called in-order traversal. He proudly calls it *tree sort*.

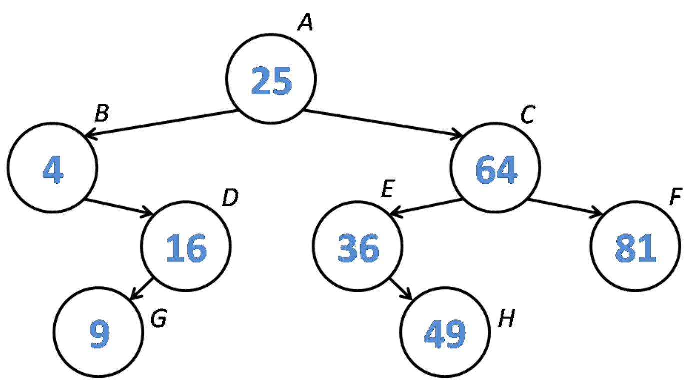
Task 3.1 Before working on the details, he asks for your help getting a good grasp on how BSTs work.

The following list of integer keys is used to build a BST, not necessarily in the order given:

$$49, \quad 16, \quad 36, \quad 81, \quad 25, \quad 4, \quad 64, \quad 9$$

2pts

a. The shape of the resulting tree is shown below. Fill in each node with one key from the list so that the resulting tree is a BST. *(The letters A–H next to the nodes will be needed in a later task.)*



Note that this is the *only* correct answer.

2pts

b. Give a specific insertion order for the keys above that results in the tree you have just filled in.

$25, 4, 64, 16, 36, 81, 9, 49$

(Any sequence so that a parent is inserted before a child is correct.)

2pts

c. Recall that the in-order traversal of a binary tree is the sequence of its entries which places the entries in the left subtree of each node before the entry in the node itself and continues with the entries in its right subtree.

What is the in-order traversal of the tree in task 1a?

$4, 9, 16, 25, 36, 49, 64, 81$

This is the *only* correct answer.

For the next few tasks, we will be extending the code for binary search trees discussed in class. Relevant portions are repeated here for your convenience.

```
// typedef ______* entry;  // Type of data in the tree

typedef struct tree_node tree;
struct tree_node {
  entry data;      // != NULL
  tree* left;
  tree* right;
};

bool is_tree(tree* T); // Representation invariant for generic trees
bool is_bst(tree* T);  // Representation invariant for BST

tree* bst_insert(tree* T, entry e)
/*@requires is_bst(T) && e != NULL; @*/
/*@ensures is_bst(\result); @*/ ;
```

For this exercise, you will not need anything more than what is given above.

2pts **Task 3.2** As a warm-up, help Rob write the function `size(T)` which returns the number of nodes in the tree `T`. *Hint: it's very short when done recursively.*

```
int size(tree* T)
//@requires is_tree(T);
//@ensures \result >= 0;
{

  if (T == NULL) return 0;
  return size(T->left) + 1 + size(t->right);

}
```

9pts **Task 3.3** Emboldened by this achievement, Rob attempts to implement a recursive function `inorder(T, A, lo, n)` that uses in-order traversal to copy the elements of a tree `T` into a segment of an array `A` starting at index `lo`. The array has size `n`, which is large enough for doing this safely. The function returns the number of elements written into `A`. This is as far as he has gone. Please help him complete his task. *Hint: draw pictures!*

```
int inorder(tree* T, entry[] A, int lo, int n)
//@requires n == \length(A);
//@requires 0 <= lo && lo <= n;
//@requires is_tree(T);
//@requires lo + size(T) <= n;
//@ensures \result == ______________size(T)______________;
{
  if (T == NULL) return ______________0______________;

  int s_left = inorder(____T->left____, A, ____lo____, n);

  A[________lo + s_left________] = ____T->data____;

  int s_right = inorder(____T->right____, A, __lo + s_left + 1__, n);

  return ______________s_left + 1 + s_right______________;
}
```

2pts **Task 3.4** What is the complexity of `inorder` as a function of the size t of the input tree `T`?

$O($______t______$)$

3pts **Task 3.5** With `inorder` done, Rob is ready (for you) to implement his new sorting algorithm. Recall that tree sort sorts an array `A` by inserting each of its `n` elements into a BST and then by doing an in-order traversal to read them off.

```
void tree_sort(entry[] A, int n)
//@requires n == \length(A) &&  ____(SEE NEXT TASK)___;
//@ensures is_sorted(A, 0, n);
{
  tree*T = NULL__________________________________________;

  for (int i = 0; i < n; i++) {

    T = bst_insert(T, A[i])______________________________;
  }

  inorder(T, A, 0, n)____________________________________;
}
```

2pts **Task 3.6** Tree sort, as conceived by Rob and implemented above, has a flaw: it will fail its postconditions for some arrays that the sorting algorithms you have studied would happily process. Give a 3-element array (using integers for simplicity) for which tree sort will produce an incorrect result. Then, give a precondition on its input that disallows such arrays (either write it in English or use a function seen in a previous homework).

Example array that tree sort will sort incorrectly:

2	3	2

Additional precondition: `A` contains no duplicates (*or* `no_dupes(A,0,n)`)

4pts **Task 3.7** How good is this fixed-up tree sort? Answer the following questions.

Worst-case complexity: $O(n^2)$

The worst-case can occur when the array is already sorted.

Tree sort is an in-place algorithm? *(circle one)* Yes $\boxed{\text{No}}$

A few days later, Rob learns about AVL trees. Since AVL trees are a special form of binary search trees, tree sort will work also if he were to use an implementation of AVL trees!

6pts **Task 3.8** Again, he first needs to wrap his head around AVL trees. Answer the following questions to help him out. *Refer to the nodes of the tree in task 1 using the letters A–H.*

Is the tree in task 1 an AVL tree? *(circle one)* Yes $\boxed{\text{No}}$

If not, it has height violations at node(s) B

To fix them, we need to do the following rotations: *(you may not need all lines)*

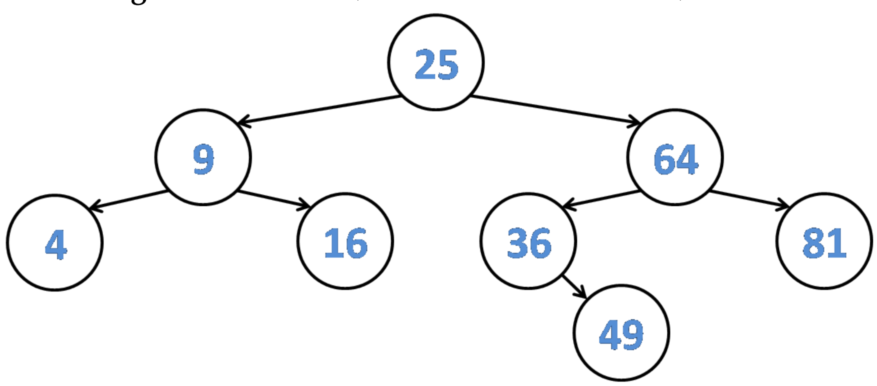
Rotate right at node D

Rotate left at node B

Rotate ______ at node ______

Rotate ______ at node ______

Draw the resulting AVL tree here: *(enter numbers in the nodes)*



2pts **Task 3.9** What is the worst-case complexity of tree sort after updating it to use AVL trees?

$O(n \log n)$

Rob mentions tree sort to Frank. Frank shows him the following non-recursive implementation of in-order traversal, which uses a (generic) stack to remember the parts of the tree that still need to be visited. (The stack interface is recalled on page 20 of this exam.)

```
1  void inorder2(tree* T, entry[] A, int n)
2  //@requires is_tree(T) && n == size(T);
3  //@requires n == \length(A);
4  {
5    stack_t S = stack_new();
6    int i = 0;
7
8    while (T != NULL || !stack_empty(S))
9    //@loop_invariant 0 <= i && i <= n;
10   {
11     if (T != NULL) {
12       push(S, (void*)T);
13       T = T->left;
14     } else { // T == NULL
15       T = (tree*)pop(S);
16       A[i] = T->data;          // THIS LINE
17       i++;
18       T = T->right;
19     }
20   }
21 }
```

Rob is not convinced of the safety and termination of this function.

2pts **Task 3.10** Line 9 does *not* support the safety of the array access `A[i]` on line 16. Why? How could you extend the loop guard on line 8 to ensure this access is safe?

Because the loop invariant allows `A[n]`, which is unsafe since `n == \length(A)`

Change loop guard to `((/*as above */) && i < n)`

2pts **Task 3.11** In English, describe a loop invariant about the stack `S` that ensures that the dereference `T->data` on line 16 is safe.

`S` contains only non-empty trees.

5 (bonus) **Task 3.12** Why does the loop on lines 8–20 terminate? Frank explains that this is because of a variant of the method seen in class. This new method relies on *two* bounded quantities and goes as follows: at each iteration of the loop,

- *either* the first quantity strictly decreases but cannot go below a certain value (and we don't care how the second quantity changes),
- *or* the first quantity stays the same but the second quantity strictly decreases and is bounded by another value.

In the function above, what are these quantities and what are their bounds?

Quantity 1: `n - i`, which is bounded by 0

Quantity 2: `size(T) -- or height(T)`, which is bounded by 0

4 Scanning Hash Tables (45 points)

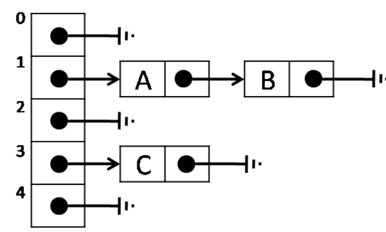
With just creation, lookup and insertion functions, the hash library interface seen in class for hash dictionaries was minimal. It is reproduced on page 21 of this exam. In this exercise, we will equip it with two operations that allow iterating through the entries in a hash dictionary. These operations, together called an *iterator*, are

- `entry hdict_first(hdict_t H) /*@requires H != NULL; @*/;`
  The call `hdict_first(H)` returns the first entry in the hash dictionary `H`, or `NULL` if `H` is empty.
- `entry hdict_next(hdict_t H) /*@requires H != NULL; @*/;`
  Each call to `hdict_next(H)` returns a next entry from `H`, or `NULL` if there are no more entries in `H`.

One can iterate through all the entries in a hash dictionary `H` by first calling `hdict_first(H)` and then repeatedly calling `hdict_next(H)` until `NULL` is returned.

For example, given the operation `print_entry(e)` which prints entry `e` on one line, the following function prints all the entries in hash dictionary `H`.

```
void print_hdict(hdict_t H) {
  for (entry e = hdict_first(H); e != NULL; e = hdict_next(H))
    print_entry(e);
}
```



Applied to the hash table on the right, the initial call to `hdict_first` will return entry A and print it. This will be followed by three calls to `hdict_next`: the first two will returns entries B and C in that order; the last will return `NULL` since the hash table does not contain other entries.

We begin by implementing the functions `hdict_first` and `hdict_next`. To do so, we extend the struct `hdict_header` seen in class with two fields:

- `last_node` points to the node containing the entry that the iterator reported the last time `hdict_first` or `hdict_next` were called. If the hash dictionary is empty or all nodes have been visited, `last_node` is `NULL`.
- `last_idx` is the hash table index of the chain where `last_node` is found. It can be arbitrary when `last_node` is `NULL`.

In the above example, after returning A, `last_node` points to that entry and `last_idx` contains 1; after returning B, `last_idx` still contains 1 but `last_node` points to B; after returning C, `last_idx` is 3. After the final call to `hdict_next`, `last_node` is `NULL`.

The relevant type declarations are as follows:

```
typedef struct chain_node chain;
struct chain_node {
  entry entry;
  chain* next;
};

typedef hdict* hdict_t;
```

```
typedef struct hdict_header hdict;
struct hdict_header {
  int size;
  chain*[] table;
  int capacity;
  int last_idx;        // NEW
  chain* last_node;    // NEW
};
```

7pts **Task 4.1** Implement the helper function `first_from(H, i)` that returns the entry of the first node in the first non-empty chain of `H` starting at table index `i`, and `NULL` if no such node exists. You will need to update the fields `last_node` and `last_idx` appropriately.

In the previous example, `first_from(H, 1)` returns A's node, `first_from(H, 2)` returns C's node, and `first_from(H, 4)` returns `NULL`.

```
entry first_from(hdict* H, int i)
//@requires is_hdict(H) && 0 <= i && i <= H->capacity;
{
  for (H->last_idx = _i_; H->last_idx < H->capacity; H->last_idx++) {

    chain* bucket = ______H->table[H->last_idx]______;

    if (______bucket != NULL______) { // Found!

      H->last_node = ______bucket______;

      return ______H->last_node->entry______;
    }
  }
  return ______NULL______; // Not found
}
```

2pts **Task 4.2** Implement `hdict_first` so that it returns the entry of the first node in the first non-empty chain of `H`, and `NULL` if no such node exists. In the previous example, that's A's node.

```
entry hdict_first(hdict* H)  //@requires is_hdict(H);
{
  return ______first_from(H, 0)______;
}
```

7pts **Task 4.3** Implement `hdict_next` so that it returns the entry of the next node in the current chain or the first node in the first non-empty chain thereafter. It returns `NULL` if no such entry exists. In our example, successive calls return B's node, then C's node, and finally `NULL`.

```
entry hdict_next(hdict* H)  //@requires is_hdict(H);
{
  if (H->last_node == NULL) return ______NULL______;

  if (__H->last_node->next != NULL__) { // Next entry in current chain

    H->last_node = ______H->last_node->next______;

    return ______H->last_node->entry______;
  }
  // Look for next entry in later chains

  return ______first_from(H, H->last_idx + 1)______;
}
```

Task 4.4 We now consider the cost of iterating through a hash dictionary `H` with n entries and whose table has capacity m. Our **measure of cost** will consist of the number of accesses to the underlying table (e.g., as `H->table[i]`) and to an entry in a chain node (e.g., as `p->entry`).

2pts

a. Consider the example function `print_hdict` on page 15. To print all n entries in the dictionary, how many times are the functions `hdict_first` and `hdict_next` called?

`hdict_first` is called ____1____ time(s)

`hdict_next` is called ____n____ time(s).

2pts

b. What is the worst-case cost of each call *separately*?

`hdict_first` has worst-case cost $O($____m____$)$

`hdict_next` has worst-case cost $O($____m____$)$

1pt

c. Assume that printing a single entry has constant cost. What is the worst-case complexity of `print_hdict` based *only* on these figures?

$O($____nm____$)$

2pts

d. But is this the real cost of `print_hdict`? Overall, how many accesses (*see above definition*) are effectively carried out when calling this function to print all entries in the dictionary? Give the exact value, not a complexity bound.

Total number of accesses: ____$m + n$____

6pts

e. Chances are that your answers to the last two questions are very different. We can use the techniques of amortized analysis to charge a cost (in terms of tokens) to use `hdict_first` and `hdict_next` so that the number of tokens collected during a call to `print_hdict` is at most 1 more than the number of accesses made by this function. Recall that we always need to have enough saved tokens to pay for the true cost of an operation in full.

Cost of `hdict_first`: ____m+1____ token(s), to be used as follows:

- ____m____ token(s), used to ____pay for all future table accesses____
- ____1____ token(s), used to ____pay for the first entry access (if any)____

Cost of `hdict_next`: ____1____ token(s), to be used as follows

- ____1____ token(s), used to ____pay for the next entry access (if any)____
- ______ token(s), used to ____________

Iterators make it easy to implement operations that require scanning all the elements in one or more hash dictionaries. We will examine a couple.

5pts **Task 4.5** Complete the implementation of the function `hdict_merge`. The call `hdict_merge(H1, H2)` returns a new dictionary containing all the elements of `H1` and `H2`. If an entry with the same key appears in both, the new dictionary will contain the one in `H2`. The initial capacity of the new dictionary should be big enough to hold the contents of both `H1` and `H2` without collisions, if we are lucky.

UNUSED

```
hdict* hdict_merge(hdict* H1, hdict* H2)
//@requires is_hdict(H1) && is_hdict(H2);
//@ensures is_hdict(\result);
{
  hdict* H = hdict_new(            H1->size + H2->size            );

  for (   entry e = hdict_first(H1); e != NULL; e = hdict_next(H1)    )

                               hdict_insert(H, e)                        ;

  for (   entry e = hdict_first(H2); e != NULL; e = hdict_next(H2))    )

                               hdict_insert(H, e)                        ;

  return H;
}
```

5pts **Task 4.6** Complete the implementation of the function `hdict_inboth`. The call `hdict_inboth(H1, H2)` returns a new dictionary containing the entries of `H1` whose key are also present in `H2`. The initial capacity of the new dictionary should be big enough to hold the contents of the smallest among `H1` and `H2` without collisions, if we are lucky.

```
hdict* hdict_inboth(hdict* H1, hdict* H2)
//@requires is_hdict(H1) && is_hdict(H2);
//@ensures is_hdict(\result);
{
  hdict* H = hdict_new(           min(H1->size, H2->size)           );

  for (entry e = hdict_first(H1); e != NULL; e = hdict_next(H1))

      if (hdict_lookup(H2, entry_key(e)) != NULL)

          hdict_insert(H, e);

  return H;
}
```

6pts **Task 4.7** Iterators even make it easy to resize a hash dictionary H once its load factor becomes too big: create a temporary hash dictionary with the new capacity, insert all entries from H into it, and finally update the header of H to the values of the header of the temporary dictionary — you do not need to concern yourself with the new iterator fields. Complete the implementation of resize to realize this idea.

```
void resize(hdict* H, int new_capacity)
/* H may not be a valid hash table since H->size == H->capacity */
//@requires H != NULL;
//@requires 0 <= H->size && H->size < new_capacity;
//@requires \length(H->table) == H->capacity;
//@ensures is_hdict(H);
{
  hdict* tmp = hdict_new(new_capacity);

  // Copy contents of H into tmp

  for (entry e = hdict_first(H); e != NULL; e = hdict_next(H))

      hdict_insert(tmp, e);

  // Copy header values of tmp into header of H

  H->capacity = new_capacity;

  H->size = tmp->size;   // not needed

  H->table = tmp->table;
}
```

The *Queue* Interface *(semi-generic)*

```
/************************/
/*** Client interface ***/
/************************/

// typedef ______* elem;

/*************************/
/*** Library interface ***/
/*************************/


// typedef ______* queue_t;

bool queue_empty(queue_t Q)
/*@requires Q != NULL; @*/ ;

queue_t queue_new()
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/ ;

void enq(queue_t Q, elem e)
/*@requires Q != NULL; @*/ ;

elem deq(queue_t Q)
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

The *stack* Interface *(generic)*

```
/*************************/
/*** Library interface ***/
/*************************/

typedef void* elem;
// typedef ______* stack_t;

bool stack_empty(stack_t S)
/*@requires S != NULL; @*/ ;

stack_t stack_new()
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, elem x)
/*@requires S != NULL; @*/ ;

elem pop(stack_t S)
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

Basic Printing Functions

```
void print(string s);      // print string  s to standard output
void printint(int i);      // print integer i to standard output
void printbool(bool b);    // print boolean b to standard output
```

The *Hash Dictionary* Interface *(semi-generic)*

```
/************************/
/*** Client interface ***/
/************************/

// typedef ______* entry;              // Supplied by client
// typedef ______  key;                // Supplied by client

key  entry_key(entry x)                // Supplied by client
  /*@requires x != NULL; @*/ ;
int  key_hash(key k);                  // Supplied by client
bool key_equiv(key k1, key k2);        // Supplied by client


/*************************/
/*** Library interface ***/
/*************************/

// typedef ______* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/
/*@ensures \result == NULL || key_equiv(entry_key(\result), k); @*/ ;

void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL && x != NULL; @*/
/*@ensures hdict_lookup(H, entry_key(x)) == x; @*/ ;
```