asgn6: towers AKA mm..OOMs

by Peter Alvaro
edits by Kerry Veenstra, Riya Aggarwal
CSE 13S, Spring 2024
Document version 1 (changes in Section 9)

Due Wednesday May 22nd, 2024, at 11:59 pm
Draft Due Monday, May 20th, 2024, at 11:59 pm

In this assignment, we will take our first foray into abstract datatypes (ADTs), the building blocks of large systems. We start you off with a linked list ADT that can be compiled to support lists containing different kinds of items. Modifying the list ADT to contain *key/value pairs* instead of integers provides the basic functionality of an *associative array*, in which we can look up integer values by a string key. Using this linked list of key/value pairs as a building block, we implement a hash table ADT that allows us to *get* (or look up) an integer value given its string key, or to *put* (or insert / update) the integer value associated with a key. We ultimately use this hash table to solve a common practical problem: processing a large text file an assigning unique integer identifiers to words that reappear.

Along the way, you will need to implement two performance optimizations, each of which will improve the performance of your ADTs by over a factor of 100. These performance improvements make software that would otherwise be unusable practical, and require careful reasoning about how we traverse our data structures.

1 Introduction

In this assignment, we are going to de-mystify what is probably the most "magical" data structure in all of computer science: the *hash table*. The hash table goes by a variety of other names that you may have enountered in other languages or in other classes, including *maps*, *hashmaps*, *dictionaries* (or `dicts` in Python), and *associative arrays*. This last name is probably the most descriptive of how hash tables work in practice: they allow us to look up data not by its *position* in some ordered array, but by providing some meaningful "key," such as a human-readable string. For example, in Python we might record the favorite songs of a few users in this way:

Example Python dict

```
favs = {}
favs['Alice'] = 'Rhymes like dimes'
favs['Bob'] = 'Flava in ya ear'
favs['Carol'] = 'Scarlet Begonias'
print("Bob's favoriate song is %s" % favs['Bob'])
```

Here, `favs` was a python `dict` (or hash table) that allowed us to look up songs by peoples' names. Hash tables are special for two main reasons:

- They are *convenient*, allowing us fetch data based on keys.
- They are *fast*. With a good hash table implementation, looking up a key can be almost as efficient as looking up an item in an array using its index. Performance matters—if it didn't, we probably would be programming in an easier language than C!

But how do hash tables achieve both of these miraculous properties? Recall that looking up items in an array via their index is fast because 1) it is fast to read memory at a particular location (this is why it is called "random-access memory" or RAM), *and* it is fast to do math. If given an index into an array, all we need to do is some math and then read some memory, so we can do it very quickly. If, on the other hand, we want to find items based not on *where* they are in memory but on *what* they are (e.g., whether a string has a particular value), our best strategy if the data is stored in an array is to scan the whole array. The bigger the array, the longer we can expect a lookup to take—not a great property when we are dealing with big data. On the surface, if our data is stored in an array, it seems like we are faced with a tradeoff: convenience or performance.
Hash tables address this problem by using a hash function to map keys to positions in an array. This allows for quick lookups, insertions, and deletions. However, hash tables can also encounter issues like collisions, where different keys map to the same position. Various strategies, such as chaining and open addressing, are used to handle these collisions efficiently.
Recall the linked list data structure we studied in lecture. Linked lists make a number of different tradeoffs than arrays. They are (arguably) a little better suited to implementing big associative arrays, because they grow dynamically. But looking for a particular item still requires traversing the list from one end to the other. In Part II of this assignment, you will implement a simple associative array as a linked list of (key, value) pairs. You will see that while tuning the data structure can give huge performance improvements ($> 100\times$!), traversing a linked list remains an unacceptably slow way to perform lookups.

In the following sections, we will explore:

- The fundamentals of hash functions and their importance.
- How to handle collisions effectively.
- The implementation of a simple associative array using linked lists.
- Performance considerations and optimizations.

By the end of this assignment, you will have a deeper understanding of hash tables and their powerful capabilities in computer science.

1.1 Hashes and Cubbies

Wouldn't it be great if we didn't need to trade off between convenience and performance? What if there was a data structure that (like array indexing) allowed us to *use math* to figure out how to get *most* of the way to where a particular piece of data is or belongs, while still supporting lookups based on key? It turns out that such a hybrid data structure *does* exist: the hash table.

It is useful to think of a hash table as a collection of lockers or cubbies. Within each cubbie is a bunch of items; books, pens and pencils, sweaters and so on. Looking inside a cubbie requires traversing (i.e., picking up and looking at) all of the items until you find what you are looking for. If the cubbie is cluttered this could take awhile but if there are just a few items it should be quick! Looking in a cubbie is a lot like traversing a list. But what about finding your cubbie? This is typically very fast, since the cubbies are numbered. Perhaps on the first day of school you need to hunt around a bit, but in general you can 1) go straight to a cubbie using math, and then 2) begin a traversal of a small list. This is *way* better than removing the cubbies and placing everyone's belongings in one giant closet—then, to find your sweater, you would need to look at all the items in the entire school.

When you think about it, libraries work in a similar way. First, you take the book that you are looking for (typically a string) and convert it into some weird number in the Dewey decimal system. Using that number, I can *very quickly* find the shelf where the book belongs, if the library has a copy. But once I am there, I still need to scan the shelf. The shelf is like a linked list that I traverse; the whole library was like an array that I indexed into to find my shelf.

The Dewey decimal numbering system made it easy to find the shelf where my book belongs—even though those numbers are completely meaningless to me! It did what is called *hashing*: converting from a

space of key (typically strings, but keys could be anything) to a (much smaller) space of numbers. A good hash function should minimize *collisions*, where different items map to the same number. After all, if the Dewey decimal system said all books belonged on the same shelf, or just three shelves, those shelves would be cluttered and hard to search. Good hash functions are outside the scope of this class. We will provide you with a poor hash function, which should illustrate that these techniques are very powerful no matter what.

The basic recipe[1] for hashing (called "chained hash tables") is:

1. Create an array of lists (e.g., linked lists).
2. For each item to be inserted or looked up, take the key (e.g., a string such as "Alice") and *hash it* into a number (e.g., 12353453).
3. Take the modulus of this (typically large) number and the number of elements in your list. This (e.g., 15) is called the "hash bucket."
4. Insert (or search) the list at that bucket.

That's it! When the number of buckets is 1, this devolves into list search; when the number of buckets is large enough there are no collisions (one book per shelf).

2 Part I: Linked list

In the starter code, we share a basic linked list implementation much like the one we went over in class. This linked list is intended to be polymorphic in the sense that the same code (which describes how to create a list, add elements, lookup elements, etc) should compile and run no matter how the *item* type is defined. That is to say, a list is a list, whether it is a list of ints, or char arrays, or pointers, or structs, or whatever.

Making the linked list polymorphic was mostly pretty easy, since the implementation passes around `items` without caring what is inside them. The one exception is the `list_find` function, which is supposed to take an item and, if the list contains a matching item, to return a pointer to that item. But what does it mean to "match" an item? This will depend on the type of `item`. Two integers match if they are identical under equality (==), whereas two strings match if `strcmp()` returns 0 when called with both. As we will see, two key/value pairs should match if their keys are the same.

To handle this special behavior of matching while remaining type-agnostic, `list_find` requires a *pointer to a function* as one of its inputs. The job of this function, `cmp`, is to report whether two items match. Hence, for each way of interpreting an item we need a different implementation of `cmp`.

The `toy` target in the Makefile will create (once you have implemented a few missing functions) a simple program that you can use to test the starter code implementation of the linked list. Read what `toy.c` is doing.

2.1 Step 1: Complete the linked list implementation

Did you read what `toy.c` was doing? It tests a linked list by inserting a bunch of data into it, removing an element from the middle, and making sure that traversals of the list still work. To do this, it calls a function (`list_remove`) that we have not provided. `list_remove`, like `list_find`, traverses a list until it finds a node matching its `item` argument. Then, instead of returning that node, it removes it from the list—after `list_remove`, the list should be unchanged except that it has one element less. Implement `list_remove`.

`toy` also allocates a bunch of heap memory and, at the end, attempts to give it up. You will also need to implement `list_destroy`. Implement `list_destroy`, prototyped in ll.h. Note that list destroy takes a `LL **`—that is, a *pointer to a pointer to* a linked list. This is because it is `list_destroy`'s job not only to free the memory pointed to by the linked list pointer, but to set that pointer to NULL.

When `list_destroy` is called, all of the heap memory allocated by the list should be given back to the OS. You can check this yourself:

[1] [For more details, refer to the example 1 at the end of the document]

Checking heap allocation of the toy program

```
$ valgrind ./toy
```

To test your implementation, ensure that `toy.c` exercises the following: some definitions are written, and a few others need to be added.

- **Insertion:** Insert multiple elements into the list.
- **Traversal:** Traverse the list to ensure all elements are correctly inserted.
- **Removal:** Remove an element from the list and verify the list's integrity.
- **Destruction:** Destroy the list and ensure all allocated memory is freed.

Test your code. Make sure that you are cleaning up all the heap memory.

Tip: To compile your code, make sure you edit[2] the Makefile to remove the other dependencies from the `all` target:

Makefile

```
#all: toy bench1 bench2
all: toy
```

3 Part II

3.1 Some simple benchmarking

We provide a testing function called `getwords` that "kicks the tires" on your implementation by reading through a list of dictionary words (about 104K words) and inserting each word (along with an ID number) into your linked list. Build the Makefile target `bench1`. Run it.

Running the bench1 benchmark

```
$ time ./bench1
```

Good heavens, that is slow. What is going on? Just inserting every word in the dictionary takes forever!!

On my laptop, this runs for almost a minute. Fix this [3]. A minor but important change to the linked list implementation can improve the performance by several orders of magnitude!! After this fix, the function runs on my laptop in about 40 milliseconds.

Tip: To ensure that the bench1 file can successfully read the dictionary file located at /usr/share/dict/words, you need to have the appropriate package installed. Run the following command to install the wamerican package, which provides the dictionary file:

```
$ sudo apt install wamerican
```

3.2 Creating a Hashtable API

Using a linked list to store key/value pairs is *a way* to implement a hash table, but it's unwieldy. I just want something that I can give a string to and get back an int!

[2] bench1 and bench2 will be used in later half of the assignment
[3] Editing definitions of already written functions in ll.c is allowed

Implement the Hashtable API by "wrapping"[4] the linked list of key/value pairs.

The Hashtable API

```
typedef struct Hashtable Hashtable;

struct Hashtable {
    // your data structure here
};

Hashtable * hash_create(void);
bool hash_put(Hashtable *, char *key, int val);
int * hash_get(Hashtable *, char *key);
void hash_destroy(Hashtable *);
```

3.2.1 Task Description:

It is your job to provide an implementation of the Hashtable API, shown above.

- `hash_create` allocates memory for an initializes a hash table. Given a key (in this case a string that is 255 characters or less),
- `hash_get` returns a pointer to the `int` that is the value for that key. If the key is not found in the hash table, it returns NULL.
- Given a key and an `int` value, `hash_put` stores the key and value; if there is already a value associated with that key, it should be replaced with the new one.
- `hash_destroy` frees all the memory associated with a hash table.

3.2.2 Example Usage:

```
Hashtable *table = hash_create();
hash_put(table, "Alice", 1);
hash_put(table, "Bob", 2);
int *value = hash_get(table, "Alice");
if (value) {
    printf("Alice: %d", *value);
}
hash_destroy(table);
```

Tips for Implementation:

- Start by defining the Hashtable structure and implementing the hash_create function.
- Implement the hash_put function to insert key-value pairs into the correct bucket. It should return true when a new key-value pair is successfully inserted or when an existing key's value is successfully updated. It should return False in case of failure.
- Implement the hash_get function to retrieve values based on keys.

[4] "Wrapping" the linked list of key/value pairs means creating a higher-level interface (the Hashtable API) that uses the linked list internally to manage key/value pairs. This abstraction hides the details of the linked list from the user, providing a simpler and more intuitive way to interact with the data structure. The user of the Hashtable API can store and retrieve values using keys without needing to know how the linked list operates internally. Example - hash_get: Retrieves the value associated with a given key. It uses the hash function to determine the appropriate bucket and list_find to locate the key in the linked list. If found, it returns a pointer to the value. If not, it returns NULL.

- Implement the hash_destroy function to free all allocated memory.

Note that we are building a tower. Just as the linked list uses the `item` datatype and its helper function, your hash table can (and should) use the linked list. It is up to you to figure out how—most of the work has already been done!

4 Part III: More performance tuning

Now, let's use this thing for a real task: assigning unique integer IDs to a HUGE list of strings containing duplicates. Build the Makefile target `bench2`. Run it.

Did you read it first? `bench2` is simulating a real-world workload. It is scanning a file full of strings again and again. For each string, it checks the hash table to see if it has already assigned this string an integer id. If it has not, it picks a *new* id and assigns it to the string in the hash table. Note that unlike the workload in `bench1`, which stressed how quickly we could add items to the tail of a linked list, this workload is much harder, stressing probes of the items.

I bet the performance—even after your optimization to the linked list—is poor, huh? I bet it scales poorly, too. You can experiment with its scaling behavior by providing arguments. For example:

Running the bench2 benchmark

```
$ ./bench2 5
```

Will scan the input file 5 times instead of just once. Try running it for a few small numbers. How long do you think it would take to scan the input file 100 times? What about 1000?

5 Part V: A hash table

The time has come to actually implement a hash table. You can do this!

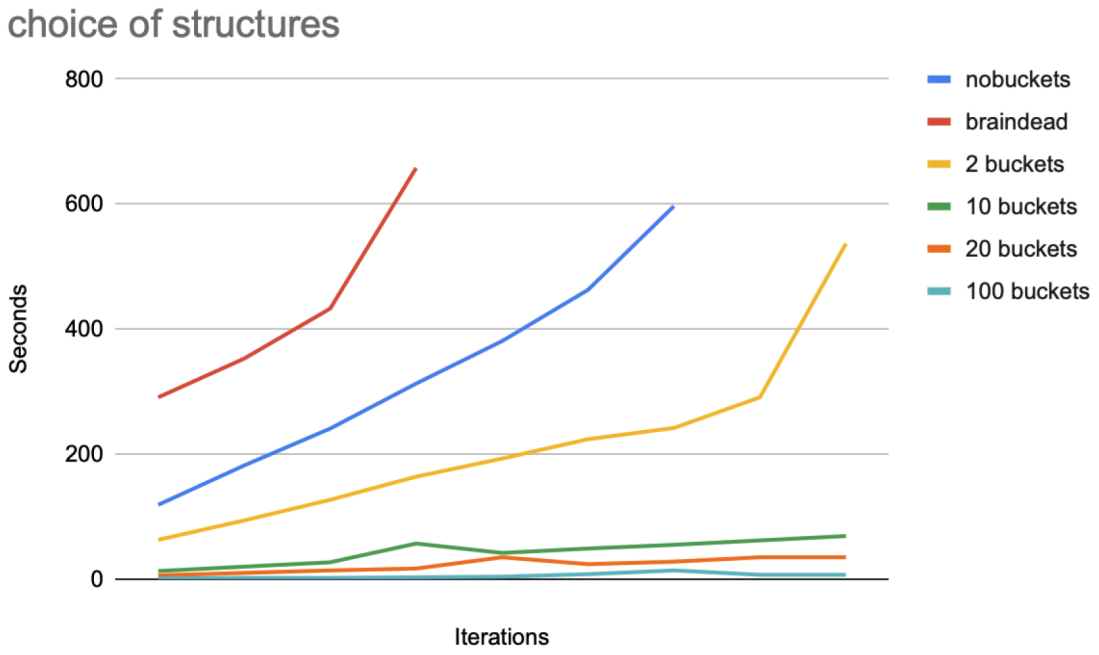
You do not need to worry about the hash function. We have provided you with a really crummy one, called `hash`. The basic idea is that you can turn a string into a number by looking at the first few bytes of that string and interpreting them as numbers. A little later in the course when we discuss bitwise operations we can talk about how you could write a crummy hash function yourselves. In practice, we typically borrow one rather than reinventing the wheel. Please feel free to swap this out with something better.

Reimplement the functions in `hash.h` to use the magic of hashing.

5.1 Does it matter?

It does. The plot below shows my reference implementation on `bench2`:

1. *In red*, before the first optimization to the linked list,
2. |
   textitIn blue, after that optimization but before implementing hashing,
3. |
   textitIn yellow, hashing, with just 2 buckets,
4. |
   textitIn green, 10 buckets, etc.



With 1000 buckets, my laptop can run 100 iterations in just over 10 seconds. Back of the envelope, since the file contains roughly 100K words, that is... $(100000 \times 100)/10 \approx 1$M records per second, I think, which is respectable.

6 The coup de grace

Congratulations: you are now not just a simple programmer, but a *library writer*. You have written a piece of code that, instead of being run once, can be used over and over again to support a myriad of applications. Of course, what is the fun of creating a library without using it to do something real?

For the final phase of tower building, you must build a simple application that uses your hash table implementation. The job of your program, called *uniqq*, is to read a file from `stdin` and output only the *number of unique lines* in that file (followed by a newline).

Create a file called `uniqq.c`[5] and a Makefile target for `uniqq`. You may feel free to reuse code from *xd*for reading inputs from `stdin`.

Make sure to test your code! We recommend that you use the UNIX utilities `uniq` and `wc` for testing[6].

7 Taking stock AKA tl;dr

Building towers means managing a big bunch of files, and this can get confusing. Below is a key to help you get organized:

7.1 Deliverables

For full credit on this assignment, you must:

1. Implement `list_remove` and `list_ destroy`. When you have done this, `toy` should build and run without error, and `valgrind` should report that there are no memory leaks.

[5](prompts for input, spacing, etc) is same as uniq
[6]Refer Example 2 in end of the document

File name	Purpose	You create	You Edit	You run
item.h	Item type		X	
item.c	Item implementation		X	
ll.h	Linked List type		X	
ll.c	Linked List implementation		X	
hash.h	Hash type		X	
hash.c	Hash implementation	X	X	
Makefile			X	
toy.c	Test code for LL			X
bench1.c	First benchmark			X
bench2.c	Second benchmark			X
uniqq.c	Unique word-counting application	X		X

2. Adapt the linked list to operate over key/value pairs by changing the definition of item and replacing the implementation of `cmp`. When you have done this, `bench1` should build and run without error, but will run very slowly!

3. Fix this performance issue by changing the implementation of the linked list. You should be able to achieve this in *negative lines of code* (i.e., by simplifying the code). After your fix, `bench1` should run (without error) in **one second or less.** After making the changes, ensure to run Valgrind to check for memory leaks and address any issues that arise.

4. Implement the Hashtable API by providing implementations of `hash_create`, `hash_put`, `hash_destroy`, and `hash_get`. When you have done this, `bench2` should build and run without error, but will run very slowly!

5. Fix this performance issue by implementing the magic that is hashing. You may be surprised to improve the performance of your system by a few more orders of magnitude. Your program should go from running about one minute per scan of the file to fractions of a second. Confirm that `bench2` runs *fast as hell*, free from errors, and without memory leaks.

6. Finally, use your library to implement a unique word counting application, `uniqq`. Test your implementation.

NOTE: In last your Makefile should only have executable for uniqq. Make sure you change name uniq-counter to uniqq. Also, don't include main.o since we dont have a main file.

7.2 In your design doc...

Make sure to discuss:

7.2.1 Draft report

- For Part I, describe your approach to implementing `list_remove` and `list_destroy`. Also, mention how you plan to ensure that all dynamically-allocated memory is cleaned up.
- For Part II, explain any initial ideas you have for optimizing the linked list to improve performance.
- For Part III, outline your plan for implementing the hash table, including the choice of hash function and collision handling method.
- Describe how you plan to use the hash table to implement the `uniqq` application. Include any initial thoughts on how you will create test inputs and check the output of `uniqq`.

7.2.2 Final report

- In Part I, you implemented *garbage collection*—routines that clean up dynamically-allocated memory. How did you make sure the memory was all cleaned up? How did you check?
- In Part II, you made a major optimization to the linked list optimization. What was it, and why do you think it changed the performance of `bench1` so much?
- In Part III, you implemented hash tables. What happens to the performance of `bench2` as you vary the number of buckets in your hash table? How many buckets did you ultimately choose to use?
- How did you make sure that your code was free from bugs? What did you test, and how did you test it? In particular, how did you create inputs and check the output of `uniqq`?

8 In Conclusion

By building a tower—an application, atop a hash table, atop a linked list, atop a key/value pair datatype—you were able to efficiently implement a complex task via a sophisticated data structure and algorithm. You may find yourself wondering (as many have before you why a powerful language like C does not have hash tables built-in, as (for example) arrays are. We hope this assignment has shown that, if you ever need a hash table, it should be quite straightforward to whip one up, in just a few dozen lines of code. A few dozen lines of code for a few orders of magnitude speedup is a great trade!

9 Revisions

Version 1: Original Draft.

Version 2: Made changes in the following sections:

1. Introduction - Changes were made in the following sections:
2. Added section 3.2.1 and changed the format.
3. New section 3.2.2.
4. Example 1 and Example 2 were included.

Example 1

Basic recipe for hashing in chained hash tables:
[Let's consider a smaller hash table with only 3 buckets (0-2)]

- Create an array of lists (linked lists):

```
[0] -> NULL
[1] -> NULL
[2] -> NULL
```

- Insert item with key "Alice":
  1. Hash the key "Alice" to a number:
     - Assume the hash function hashes "Alice" to 12353453.
  2. Calculate the hash bucket:
     - Take the modulus of 12353453 % 3 = 2
     - The hash bucket is 2.
  3. Insert "Alice" into the list at bucket 2:

```
[0] -> NULL
[1] -> NULL
[2] -> [Alice] -> NULL
```

- Insert item with key "Bob":
  1. Hash the key "Bob" to a number:
     - Assume the hash function hashes "Bob" to 9876543.
  2. Calculate the hash bucket:
     - Take the modulus of 9876543 % 3 = 0
     - The hash bucket is 0.
  3. Insert "Bob" into the list at bucket 0:

```
[0] -> [Bob] -> NULL
[1] -> NULL
[2] -> [Alice] -> NULL
```

- Insert item with key "Dave":
  1. Hash the key "Dave" to a number:
     - Assume the hash function hashes "Dave" to 11111111.
  2. Calculate the hash bucket:
     - Take the modulus of 11111111 % 3 = 2
     - The hash bucket is 2.
  3. Insert "Dave" into the list at bucket 2:

```
[0] -> [Bob] -> NULL
[1] -> NULL
[2] -> [Dave]->[Alice] -> NULL
```

- Search for item with key "Bob":
  1. Hash the key "Bob" to a number:
     - The hash function hashes "Bob" to 9876543.
  2. Calculate the hash bucket:
     - Take the modulus of 9876543 % 3 = 0
     - The hash bucket is 0.
  3. Search the list at bucket 0:
     Traverse the list at bucket 0 and find "Bob".

```
[0] -> [Bob] -> NULL
[1] -> NULL
[2] -> [Dave]->[Alice] -> NULL
```

Example 2

To test your code, you can use the UNIX utilities `uniq` and `wc`. For example:

- Create a test file `test.txt` with some lines, some of which are duplicates:

```
apple
banana
apple
orange
banana
grape
```

- Run your `uniqq` program with the test file:

```
$ ./uniqq < test.txt
4
```

  This should output 4 since there are 4 unique lines: `apple`, `banana`, `orange`, and `grape`.

- Verify the result using `uniq` and `wc`:

```
$ sort test.txt | uniq | wc -l
4
```

  This should also output 4, confirming that your program works correctly.