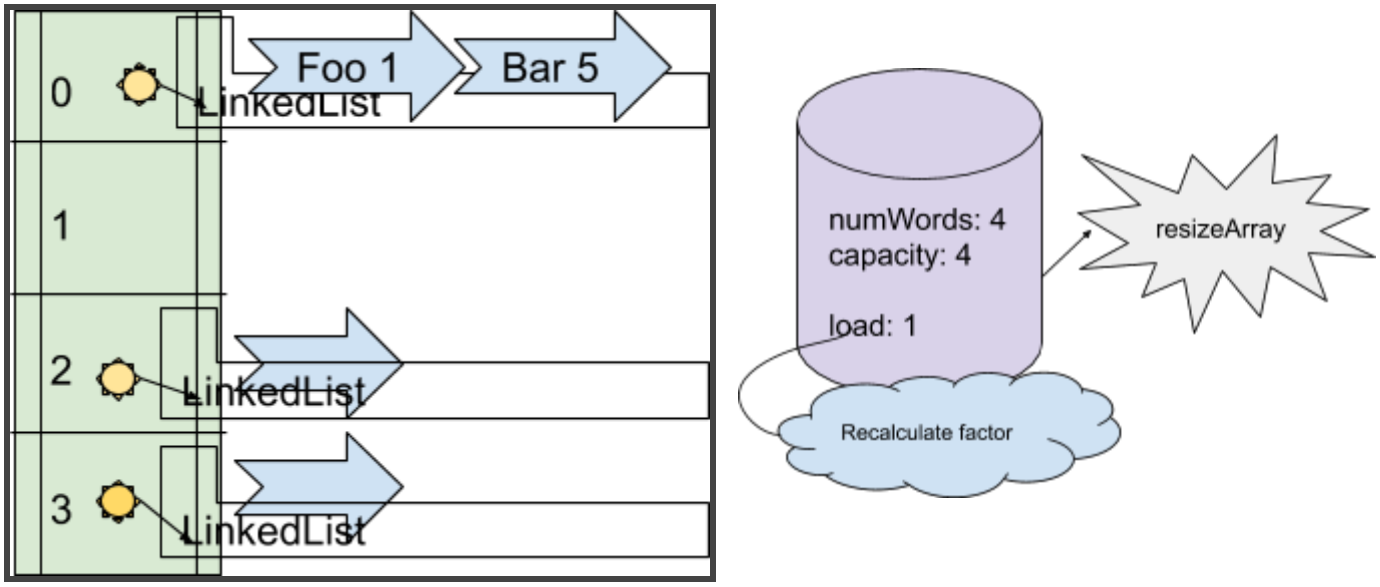
P5

This project revolved around the implementation of a HashTable to store and interact with dictionary words and the frequency of their appearances in various media (see p5 spec). I achieved this via a HashTable consisting of an array of pointers to LinkedLists, each consisting of Nodes. In this diagram, the array is **green**, the LinkedList clear (where initiated), and Nodes are **blue**. The stars are the pointers, once initialized, to LinkedLists in heap.



This diagram depicts the HashTable data structure on the left, and shows what happens just as we are about to add our 5th word; the load has been recalculated and is >=1. `ResizeArray()` will be invoked as a result.

No STL was used in this project. Essentially all components of the HashTable are in heap. See README.md for commentary on memory bottlenecks and preparation.

1. Node

Nodes are part of a LinkedList, and will receive the data input from user. They contain the critical data (word, frequency value read-in), as well as value returned from hashing once assigned, and a pointer to the next node, if any.

`Node()`

Not used. Throws `std::runtime_error("Not Implemented")`

`Node(std::string word, int freq)`

Node initialized to be one containing given word and frequency, with no “next” Node, and a negative (invalid) hash value.

`~Node()`

LinkedLists are responsible for deallocation. Doesn't release other nodes, just its own.

`Node& Node::operator=(const Node& other)`
`Node(const Node& other)`

This Node will be initialized to be one with the same info as provided Node.

`int getFreq() const;`

Returns the frequency value this node contains.

`std::string getWord() const;`

Returns the wordvalue this node contains.

`unsigned int getHashVal() const;`

Returns the hash value this node contains.

`Node* getNext() const;`

Returns the pointer to next node this node contains.

`void setNext(Node* next);`

Sets the pointer to next Node to given Node.

`void setHashVal(unsigned int hashVal);`

Sets the hash value to be the passed-in int. Note: the hash is evaluated in a different class (HashTable).

`std::string toString() const;`

Prints the Node value as simply "word freq", e.g. "foo 99".

2. LinkedList

The LinkedList is singly linked, and responsible for the memory allocation of all Nodes. These are the chains in the HashTable array that allow for chaining during collisions. Tracks own size. The LinkedList contains a head Node pointer that points to other Node pointers.

`LinkedList();`

Initialize to have head pointing to nullptr, and size of 0.

`LinkedList(const LinkedList& other);`
`LinkedList& operator=(const LinkedList& other);`

Not implemented; throws `std::runtime_error("Not Implemented")`.

`~LinkedList();`
Deletes all nodes associated with this list, and then the head (and list) itself.

`Node* at(int ith) const;`
Returns the node pointer at specified location in this list. **`nullptr,`** if not valid.

`void addToFront(Node* node);`
Point head at this Node; it's in front, now. Set next on given node to what head used to point at.

`void removeFront();`
Delete the head node, pointing head at what it used to point at. Does nothing if head already null.

`int getSize() const;`
Return the tracked size of this LinkedList.

`bool isEmpty() const;`
If head is pointed at **`nullptr`**, return `true`. Else, `false`.

`std::string toString() const;`
Useful printing function, for testing. Shows where head, etc. is.

`bool removeWord(std::string word);`
Searches this LinkedList for given word. Deletes node and stitches List back together, and returns **`true`** if successful. If word is not present or otherwise unsuccessful, returns **`false`**.

`Node* findWord(std::string word) const;`
Traverses LinkedList to return node containing search key. Returns **`nullptr`** if not found.

(PRIVATE FUNCTIONS)

`Node* at(Node* current, int tracking, int ith) const;`
This private helper function helps recursively track down the word in question.

3. HashTable

The HashTable consists of an array of pointers to LinkedLists. These LinkedLists contain the (pointers to) nodes. The HashTable contains many useful functions, and tracks its own capacity, wordcount, and load factor itself. It is responsible for deleting all LinkedLists inside its bounds.

`HashTable();`
Initializes to hold LinkedList pointer array of arbitrary size (say, start with 2048). Store that capacity, and intiailize wordcount and load to 0.

`HashTable(const HashTable& other);`

`HashTable& operator=(const HashTable& other);`

Not implemented; throws `std::runtime_error("Not Implemented")`.

`~HashTable();`

Destructor runs through each node in list and deallocates memory of LinkedList, before deleting the array.

`int getCapacity() const;`

Returns the capacity value this HashTable holds.

`int getNumWords() const;`

Returns the number of words this HashTable holds.

`int getLoad() const;`

Returns the current loadfactor this HashTable has stored.

`Node* getData(std::string word);`

Returns the node associated with a given word. First, the HashTable determines which array index/LinkedList *ought* to contain the word; then it goes and gets it. Returns **`nullptr`** if not found.

`std::string toString() const;`

Returns a representation of the current HashTable. Gets unwieldy at large values, but useful for testing.

`Node* convert(std::string word, int freq);`

Given arbitrary word and frequency values, create a new (unlinked) **`Node*`** in memory that contains those values. To be used in conjunction with **`add`** (which isn't always used for user input!).

`void add(Node* node);`

Given a **`Node*`**, add to the HashTable. Will first **`recalculate`**() load factor and call **`resize`**() if necessary, both being private functions. Will determine the appropriate array index based on current size, so that the hash need not be recalculated on every resize. Uses that LinkedList's `addToFront()` to quickly prepend to the list.

`bool removeWord(std::string word);`

Given an arbitrary word, will remove from the HashTable. Will determine which array index to look for based on the hashvalue of the passed-in word (so, it calls the function that generates the hash value) and current size of the array. Then, invokes `LinkedList->(removeWord())` to delete **`Node`*** associated with word. Returns **`true`** if successful; otherwise, **`false`**.

PRIVATE FUNCTIONS

`unsigned int generateHash(const std::string& key);`

~~Borrowed from lab8 implementation. For a given immutable string, return an int that represents its hashvalue. I used the homegrown ASCII-string-to-int approach I used in lab, without references. It is a performance bottleneck.~~

I implemented the DJB2 algorithm at the last minute. Massive gains!!! Please see README for reference.

`void recalculateLoad();`

Quickly resets the current Load to equal the Wordcount/Array size. I used *10 to give an int value.

`void resizeArray();`

This is a doozy.

- We create a holding swapArray LinkedList** and give it the same pointer values as the current Array, preserving the connections.
- We perform a simple delete [] on the HashTable's own array, reinitializing it with twice the capacity. We double the capacity.
- We loop through the old array (with the old capacity serving to bound our loop), and essentially "pop" head from each LinkedList* until the index's LinkedList* is empty. We utilize **`add`** (which re-calculates the appropriate *new* index using the *new* capacity).

This briefly creates a new node before deleting the old one; it is relatively space efficient as a result, as we will only have n+1 node memory reserved at any given time.

4. Main

Make build is meant to produce a `./database` file. `Main.cpp` is the means by which end users will ingest a file (e.g. ./dataset_large.tsv) and interact (via the HW spec's description of the **`:g`** `word`, **`:p`** `word frequency`, **`:r`** `word`, and **`:q`** commands). `main`() begins by initing a new hashtable in memory (not in the stack), and then invokes ingestFile. It then initiates a loop to receive user command input and invokes the helper `cmd` functions you see below.

Here I describe the functions that I used to strip down `main()` a bit.

`bool ingestFile(HashTable* table, std::string filename);`

Takes a hashtable and the filename. Checks to see if filename is legit and to load it. Returns **`true`** if succesful, **`false`** if not.

`void getCmd(HashTable* table, std::string word);`

Takes a hashtable and a word passed from **`main()`**. Executes hashtable's getWord() function, which prints the input if exists, else an error message to **stdout**.

`void putCmd(HashTable* table, std::string newWord, int newFreq);`

Takes a hashtable and a word passed from **`main()`**. Executes hashtable's add and convert functions, which creates a new node and adds it to the hashtable. Checks to see if provided frequency is valid. Prints a success message if success, else an error to **stdout**.

`void removeCmd(HashTable* table, std::string removalWord);`

Takes a hashtable and a word passed from **`main()`**. Executes hashtable's removeWord() function, and prints success message if success, else an error message to **stdout**.