Lab Assignment #10
Data Structures 2

Due 5/1/2020 at 11:59pm on bCourses

The assignment will be graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your classes, properties and functions **exactly match** the names specified in the problem statements, input and output variables to each function are in the **correct order** (i.e. use the given function header exactly), and each function performs exactly what it is asked to. Instructions for submitting your assignment are included at the end of this document.

Skeleton codes are provided on bCourses for `wordHashTable` and `minHeap`, along with some additional instructions. Follow the instructions and fill in your implementations. **DO NOT MODIFY** any code that is given in the skeleton code or you may lose points, although you may feel free to add codes. For the rest of the problems, you will write from scratch.

Suppress all non-required outputs.

Useful Resources

- Guide to class constructor methods

1 A Simplified Hash Table - `wordHashTable`

In this problem, you will implement the hash table data structure that you learned in class. To do so, you will also utilize the `myWordNode` and `wordList` classes you built in Lab09 (solution codes for these are provided on bcourses). A hash table has a defined capacity and pre-allocated memory space. An element's placement in the hash table is determined by the hash function specified for the hash table. The use of pre-allocated memory space and a hash function allow insertion, deletion, and searching of elements in a hash table to be as fast as $O(1)$, near constant time. Hash tables are often used in applications which require fast data I/O abilities.

Hash Table Terminology and Details

This section gives a brief review of hash table terminology and introduces some implementation details in this problem.

- An **element** in hash table is just like a node in a linked list. For our purposes, we will have a word, occurrences and pointers.
- To store the element (myWordNode), you will need a hash function to calculate where in the hash table you are going to store the element.
- The underlying data structure of hash table is usually a simple array with a pre-defined size, which is called the **capacity** of a hash table. Memory pre-allocation enables constant time I/O within the array. We call each element of the array a **bucket**.
- To store, remove, or search for an element (myWordNode) within a hash table, you are going to calculate an index using the hash function with the input `word`. The index output of the hash function will represent which bucket of the hash table the element should be located.

- A good **hash function** will evenly distribute its elements over the hash table array while a poor hash function will result in collisions. The hash function returns a **hash value** of a key, which is used as the bucket index in the hash table array. **Collisions** occur when multiple elements get hashed to the same bucket as seen in bucket 152 below.

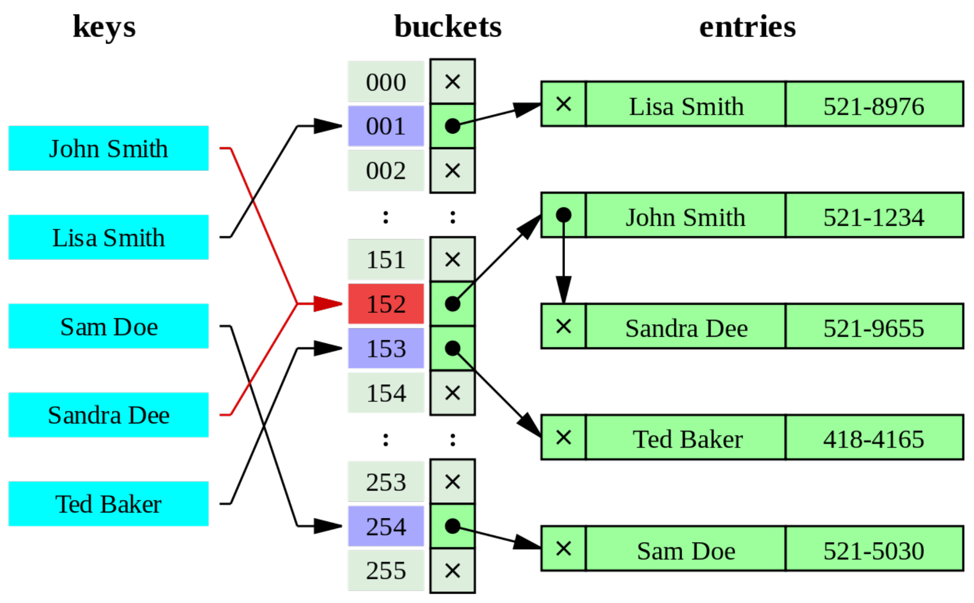

Figure 1: An Illustration of Hash Table

- The **size** of a hash table is the number of elements it currently has. For our hash table, this will be indicated by uniqueWords. **Do not confuse it with capacity - capacity is fixed and size can change.** A new hash table is initialized with 0 size.

- For our hash table, the **fill ratio** is defined using size/capacity. It is used to determine when the hash table needs to expand and **rehash**. For best performance, it is desired to stay below the range of 0.5 and 0.75. We will discuss rehash later.

Now that you know the basic terminology and structure, we are going to implement a hash table capable of storing English words. The hash table will utilize the class `myWordNode` and `wordList` from Lab09. The input to your hash table will be a word, but you are to place a node into the table.

1.1 Download solutions for `myWordNode` and `wordList` from bcourses

These two codes will be utilized as a substructure of the hash table due to an imperfect hash. Each bucket that has had an element hashed to it will contain a `wordList`. Grading will be done using our own copy of these two codes, so do not change them.

1.2 Polynomial hash function for dictionary word

`function index = wordHashFunction(word,cap)`		
Input	Type	Description
`word`	`1xN char`	A word to be hashed.
`cap`	`1x1 double`	Capacity of hash table.
Output	Type	Description
`index`	`1x1 double`	The hash value for the given input word. This value is the range [1, capacity].

The hash function is an important part of the hash table. Ours will be defined outside the hash table class and passed in as a function handle when initializing the hash table. The output index will be used to determine which bucket the input word shall be located.

We will utilize a function based on prime numbers and ASCII values. The polynomial equation to be used is:

$$H = a_1(X_1)^1 + a_2(X_2)^2 + a_3(X_3)^3 ... + a_n(X_n)^n$$

where a_n is the ASCII value of the nth char of the input char array and X_n is the nth prime number in increasing order. The maximum length char array you will need to hash is 25 characters.
Hint: think about how `mod()` can be used to relate H, the capacity, and the index. Remember that index must be in the range [1, `cap`].

For example, the word 'Raja' $\longrightarrow$ a_1, a_2, a_3, a_4 $\longrightarrow$ 82, 97, 106, 97.

$$H = 82(2)^1 + 97(3)^2 + 106(5)^3 + 97(7)^4 = 247,184$$

Given a capacity of 10, the index would be 5. Similarly, 'Xin' would hash to 2 and 'Morgan' would hash to 4.

1.3 `wordHashTable` class - Properties

Edit the skeleton code for wordHashTable to include the following properties:
`buckets`: where elements added to the hash table are stored.
`capacity`: an integer representing the number of buckets within the hash table.
`currentFR`: a ratio representing the current fill ratio of the hash table.
`maxFillRatio`: the maximum allowable fill ratio of the hash table.
`hashFunction`: the hash function being utilized by the hash table.
`totalWords`: the total number of words stored in the hash table. It is initialized to be 0.
`uniqueWords`: the number of unique words stored in the hash table. It is initialized to be 0. This also represents the number of elements that have been added to the hash table and the table size.

`wordHashTable` also contains functions, as described in the following sections.

1.4 `wordHashTable` class function - `createTable`

`wordHashTable` requires a function to initialize the hash table and define the properties. For this, we will create the function `createTable` with the following inputs and no output:

capacity: the desired capacity of our hash table.
maxFillRatio: the threshold at which our hash table rehashes.
hashFunction: the hash function to be utilized by our hash table.

These inputs should be used to initialize the properties of the hash table object.

1.5 `wordHashTable` class function - insertWord

To insert a word into the hash table, we will utilize `insertWord`, which will have an average time complexity of **O(1)** and a worst-case complexity of **O(n)**. This function will accept **word, a 1xM char**, and will create and place a node (`myWordNode`) into the hash table. The node will have the property `word` defined as the input word, and the property `occurrences` set to 1. The bucket that this node gets placed into is determined based on the hash function. If the bucket determined by the hash function is currently empty, initialize a `wordList` and insert the node into the list. Otherwise, simply insert the node into the existing list. List interaction shall be handled using the code produced in Lab09.

Remember to update the hash table properties `totalWords`, `uniqueWords`, and `currentFR`.

As `currentFR` approaches `maxFillRatio`, the efficiency of the hash table decreases. To keep the hash table efficient, after each `insertWord` operation, you will update the `currentFR` ratio of the hash table. If `currentFR` exceeds (>) the `maxFillRatio`, the hash table must be rehashed, a process described in the next section.

1.6 Rehash - This has been done for you - read for understanding

An overly crowded refrigerator does not have the best cooling efficiency. The same efficiency loss can be seen in a hash table with a high fill ratio. In order to maintain efficiency, we need to rehash a hash table when the fill ratio is high.

When rehashing occurs, the capacity doubles. This change in capacity changes the bucket that each word would hash to. That being said, all elements in the table need to be updated (rehashed) based on the new capacity - that is, placed into their new location in the new table. Remember to update `currentFR`.

Hint: The `rehash` function works by creating a variable that contains the existing hash table (cell array), looping through all buckets, and then rehashing each element within the buckets. The function must also rehash a word according to the number of occurrences.

1.7 `wordHashTable` class function - `searchWord`

Databases often use hash tables. Thus, the ability to search for a given piece of data is an important part of the hash table. For our hash table, we will implement `searchWord`, which accepts a **word**, a 1xN char array, and outputs a `myWordNode` `node` with empty pointers. This function will have an average time complexity of $\mathbf{O(1)}$ and a worst-case complexity of $\mathbf{O(n)}$. The search process can be accomplished with the following steps:

1. Obtain the index corresponding to `word` via the hash function.
2. Examine the bucket corresponding to the index obtained in step 1. Two possible conditions exist.
   - The bucket is empty.
   - The bucket contains a list. In this case, search the list for the given word.

   If the word cannot be found, the output, `node`, should be `'The hash table does not ... contain the word WORD'`, where WORD is the input word.

Hint: If the bucket is not empty, you can utilize the `wordList` structure and functions created in Lab09 to retrieve the node using the given input `word`.

1.8 `wordHashTable` class function - `deleteWord`

Deletion of data is also a necessary requirement for hash tables. For this hash table, we will be implementing the function `deleteWord` to accomplish this. The input for `deleteWord` will be a char array, `word`, a 1xM char array, and will output `removedNode`. This function will have an average time complexity of $\mathbf{O(1)}$ and a worst-case complexity of $\mathbf{O(n)}$.

The deletion process can be accomplished with the following steps:

1. Obtain the index corresponding to the `word` via the hash function.
2. Look into that bucket.
3. Examine the bucket corresponding to the index obtained in step 1. Two possible conditions exist.
   - The bucket is empty.
   - The bucket contains a list. In this case, delete the node with the given word from the list.

   If the word cannot be found, the output, `removedNode`, should be `'The hash table ... does not contain the word WORD'`, where WORD is the input word.
4. If a `myWordNode` is found and deleted, remember to adjust the properties: `uniqueWords`, `totalWords` and `currentFR`.

Hint: If the bucket is not empty, you can utilize the wordList structure and functions created in Lab09 to remove the node using the given input word.

2 Compare Hash and Linked List Storage

```
function [listTimes,tableTimes] = compareStorage(txt, cap, fR, hF, add, del)
```

Input	Type	Description
`txt`	`1xN char`	A text to be parsed and placed into memory.
`cap`	`1x1 double`	Initial capacity of hash table.
`fR`	`1x1 double`	Maximum allowable fill ratio for hash table.
`hF`	`function_handle`	Function handle for hash function to be used for hash table.
`add`	`1xM char`	Additional word to store into memory.
`del`	`1xO char`	Additional word to delete from memory.
Output	**Type**	**Description**
`listTimes`	`3x1 double`	The computation time required to parse and store all words in `txt` into `wordList`, add an individual word into `wordList`, and to delete a word from `wordList`.
`tableTimes`	`3x1 double`	The computation time required to parse and store all words in `txt` into `wordHashTable`, add an individual word into `wordHashTable`, and to delete a word from `wordHashTable`.

Details

The main purpose of this problem is to have you interact with linked lists and hash tables and see the benefits of each. Through the use of the hash function and the buckets, a hash table breaks up long lists into separate much shorter lists which leads to a near O(1) for add and remove of an individual element, whereas a double linked list is O(n). In this problem, you will demonstrate these differences by timing the following three operations.
To demonstrate these differences, you are going to:

- Parse a given input text, `txt`, into individual words. For each word, you will store it in **`wordList`** (a linked list) and in the **`wordHashTable`** (hash table). Record the computation time required to parse the `txt` and store all words in the **`wordList`** and **`wordHashTable`** as the first element of `listTimes` and `tableTimes` respectively.

- Record the computation time required to add the input word, `add`, to both the **`wordList`** and to the **`wordHashTable`** and record this time as the second element of the `listTimes` and `tableTimes` respectively.

- Record the computation time required to delete the word `del` from both the **`wordList`** and to the **`wordHashTable`** and record this time as the third element of the `listTimes` and `tableTimes` respectively.

Note: The computation time required to make and update the **`myWordNode`**s should be included in the total computation time when dealing with **`wordList`**.

Tips

Parse: to analyze (a string of characters) in order to associate groups of characters with the syntactic units of the underlying grammar.
In other words, you are to split up the char array `txt` into the individual words. This may be accomplished by your own method, or MATLAB's built in function `strtok` (flashback to Midterm 1 - the "ummmm" program).

3 Binary Heap - minHeap

We will now work with another data structure, called a binary heap. Our implementation of the binary heap will be considered a min heap. A min heap uses the binary tree structure with parents having lower values than their two children.

Jargon

This section is a general introduction to the binary heap and the jargon that we will be using.

There are multiple ways to implement binary trees. Binary trees can be thought of as a 'tree' of nodes. The top of Figure 2 can help you visualize the binary tree structure.

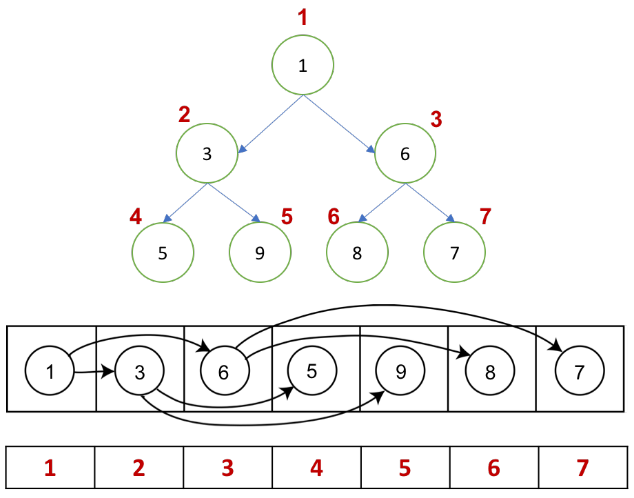

Figure 2: A Binary Heap Tree Example

The bottom half of Figure 2 will is a representation of the array format we will utilize for our min heap implementation. Nodes are represented in red (outside the circle) and values are represented in black (inside the circle). Node 1, the **root** or **parent**, has two **children**: node 2, the left child, and node 3, the right child. Similarly, node 3 has two children: node 6, the left child, and node 7, the right child.

In the binary heap tree,

- The value of a child is always greater than the value of the parent.
- The index of the left child is equal to two times the index of its parent i.e., $i_L = 2 \cdot i_P$
- The index of the right child is equal to two times the index of its parent plus one i.e., $i_R = 2 \cdot i_P + 1$

3.1 Define `minHeap` class

Edit the skeleton code for the `minHeap` class to include the following three properties:
`values`: a public property that contains the values that are currently in the heap. It should be initialized as empty.
`capacity`: a private property that is the maximum number of elements that can be contained within the heap. It should be intialized as 0.
`heapSize`: a private property that represents the number of elements currently contained within the heap. It should be intialized as 0.

3.2 `minHeap` class function - `createHeap`

`createHeap` will be used to initialize the instance of minHeap and update the properties according to the following inputs:
nodes: the maximum number of nodes that will be contained within the heap.
The `createHeap` function should update the `capacity` of the instance. It should also preallocate the `values` array as an array with each value set to inf. You may find the MATLAB built-in function `inf()` useful.

3.3 `minHeap` class function - `checkSize`

The function `checkSize` will have no inputs and should report the number of elements within the heap. **Hint:** Think of what property tracks current heap size.

3.4 `minHeap` class function - `push`

You will now define a function, `push` that allows the user to push an element onto the heap - that is, add a value to the heap. This function will have one input, **`value`, a 1x1 double**, and no outputs. Insertion of the `value` is constant time, $O(1)$, due to the use of indexing; however, the process to maintain the heap structure has a worst case complexity of $O(log(n))$.

The steps to push an element onto the heap are as follows:

1. Update your heap size to reflect the additional value added to the heap. If heap size exceeds capacity, display the message `'The heap capacity has been exceeded. Value cannot ... be added.'` and end the operation.

2. Insert the `value` into the next available index (think how this relates to heap size).
3. Reestablish the min heap structure by swapping parent and children nodes as necessary. Remember, no child node should have a lower value than the parent nodes' value.
   Hint: You can calculate the parent's index by altering the equations provided in the Jargon section.

For example, if you add the values [1, 5, 8, 3, 9, 6, 7] from left to right to an instance of minHeap, you will produce the values array seen in Figure 2.

3.5 `minHeap` class function - `pop`

You will now define a function, `pop` that removes the minimum value from the heap. This function will have no inputs and will output the minimum value that was removed from the heap as **`value`, a 1x1 double**. Removal of the minimum value from the heap is constant time, $O(1)$, due to the minimum value being located in the first element of the array. The process to maintain the heap structure has a worst case complexity of $O(log(n))$.

The steps to pop an element onto the heap are as follows:

1. Assign the first element of the `values` array as the output.
2. Swap the value at index `heapSize` of the `values` array into the first element of the `values` array.
3. Replace the value you moved from the index of `heapSize` with `Inf` so the value that was moved is not duplicated.
4. Update `heapSize` to represent the change in the number of elements in the heap.
5. Reestablish the min heap structure by comparing the parent node with its two children nodes. If the parent value is larger than both children values, swap with the child that has the lower value. If the parent value is only larger than the value of one child, swap it with that child.
   Hint: You can calculate the parent's index using the equations provided in the Jargon section. The number of swaps that likely will occur is in relation to the height of the tree (how many layers the tree has).

Note: If the `heapSize` is 0 when `pop` is called, the output, `value`, should be assigned as `'The ... heap is empty.'`

For example, if you were to call `pop` on the binary heap tree in Figure 2, you would start by replacing the value in position 1 with a value of 7. You would then compare the value of 7 with its 2 children, value 3 and value 6. 7 is larger than both of its children, and so we swap its place with the smaller of the two children - in this case, 3. Thus, 3 becomes the new binary heap tree root. We then compare 7 with its next two new children: 5 and 9. 7 is only larger than 5, so we then swap their positions.

Testers

The tester for this assignment gives a general overview of the things you should be looking for. There is no way to include all test cases so you should look at the tester code to learn how to interact with your hash table and the min heap. Once you have done so, you should do additional test cases on your own. For example, make randomized arrays and insert the values into the minHeap or write your own texts for problem 2.

Each section of the testers have brief headers, but we are unable to provide details descriptions of each test case. If you get an error from a tester, you must locate that particular test case and diagnose the cause. You will not ask on Piazza ”Why the tester gives me an error?”

Submission Guidelines

This homework is a long one and please follow the submission instructions when submitting it!

All classes and functions must have the exact same names as those specified in the respective sections. All functions must have the exact the same names and input parameters as those specified in the respective sections. All files names must also follow either the specifications or MATLAB's naming rules.

The following files are to be submitted for this assignment:

Your final zip file for this assignment should contain the following:

- wordHashFunction.m
- wordHashTable.m
- compareStorage.m
- minHeap.m