CSC190: Computer Algorithms and Data Structures
Lab 4
Assigned: Feb 9, 2015; Due: Feb 23, 2015 @ 10:00 a.m.

1 Objectives

In the first part of this lab, you will implement general *operations* for a specific type of linked list called the *ordered* list. In the second part of the lab, you will use the general operations you had implemented in the first part of the lab to create a simplified hash table. We have provided files that you must use as a starting point to complete this lab. You will download the contents in the folder `Lab4` which contains two folders (`code` and `expOutput`) into your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `lab4.c` and include necessary libraries in `lab4.h` as required for implementation. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. Folder `expOutput` contains outputs expected for the supplied `main.c` file. Note that we will **NOT** use the same `main.c` file for grading your lab. Do **NOT** change the names of these files or functions.

2 Grading

It is **IMPORTANT** that you follow all instructions provided in this lab very closely. Otherwise, you will lose a *significant* amount of marks as this lab is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this lab (total of 20 points):

- Successful compilation of all program files with no memory leaks i.e. the following command results in no errors (2 points):
  ```
  gcc main.c lab4.c -o run
  valgrind --quiet --leak-check=full --track-origins=yes ./run 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 2
  ```
- Output from Part 1 exactly matches expected output (4 points)
- Output from Part 2 exactly matches expected output (6 points)
- Code content (8 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. Late submissions will **NOT** be accepted.

Part 1: Implement Interfaces for an Ordered Linked List

In this part, you will implement functions that will allow for the construction of an ordered linked list. Every node in the linked list is a structure of type `Node` that has members `data` and `next`. The structure `Node` has been defined for you in the `lab4.h` file. In the lectures, it has been mentioned that nodes can be inserted into any location in the linked list. In this lab, a constraint is introduced for node insertion. This constraint requires that the nodes inserted into the linked list must preserve an order. For instance, suppose that you are inserting a node into a linked list that already contains two nodes with `data` members having values 1 and 3. If the new node to be inserted has the value 2 in its `data` member, then this node should be inserted in between the two nodes originally present in the linked list. Hence, the insertion operation preserves the order of values in the member `data` of all nodes in the linked list. You will implement the following functions:

- `void insertNode(int d, struct Node ** lPtr);`
- `void deleteNode(int d, struct Node ** lPtr);`

- `void printList(struct Node * L);`

Into the `insertNode` function, the value of the `data` member of the new node and a double pointer `lPtr` which stores the memory address of the pointer to the linked list are passed as arguments. This function will dynamically allocate space for the new node and store the passed value `d` into the `data` member of the newly allocated structure space. This function then inserts the new node into the linked list so that the order of the `data` members of all nodes in the linked list is preserved (ascending order). The second function to be implemented is `deleteNode`. This function will delete the node first encountered in the linked list whose member `data` has the value `d`. Into this function, the integer `d` and a double pointer `lPtr` are passed as arguments. The last function to be implemented in this part is `printList` which will print the value of the `data` member in every node of the linked list represented by `L`. Test your implementation via:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

Ensure that your output matches the sample output file `P1.txt` located in the `expOutput` folder.

Part 2: Implement Interfaces for a Simplified Hash Table

In this part, you will make use of the functions you created in Part 1 to implement a simple hash table. In a hash table, elements (also known as keys) are stored in a manner that enables more efficient retrieval (when compared to storing all keys into a single linked list). A key that is to be stored into a hash table can be directly mapped into an index with a hash function. The key is then stored in a location associated with that index in the hash table. If all keys map to a unique index then these keys can be accessed in O(1) as the only computation necessary is the mapping of the key to the index. If several keys map to the same index, then a *collision* will occur. When this happens, it is not possible to store two conflicting keys at the same location. There are many methods to resolve such conflicts. In this lab, you will explore the *separate chaining method* which utilizes ordered linked lists.

Suppose that there are n keys and that these keys are integers. Each key k will be stored as a `Node` structure in which the `data` member will hold the value of the key. Suppose that the hash table has r rows. Each row represents a bucket. Each bucket will store all the keys that map to the same index (i.e. all colliding keys) as an ordered linked list. A key is mapped to an index via the hash function. The hash function used in this lab is $k\%r$ where k is the key and r is the number of buckets in the hash table. If a key has remainder 0 when divided by r, then it will be associated with bucket 0. If the remainder is 1, then it will be associated with bucket 1 and so on. The single pointer pointing to the ordered list containing all colliding keys belonging to the i^{th} bucket is stored as the i^{th} element of the hash table. The entire hash table is essentially an array of single pointers and hence a double pointer can represent the hash table.

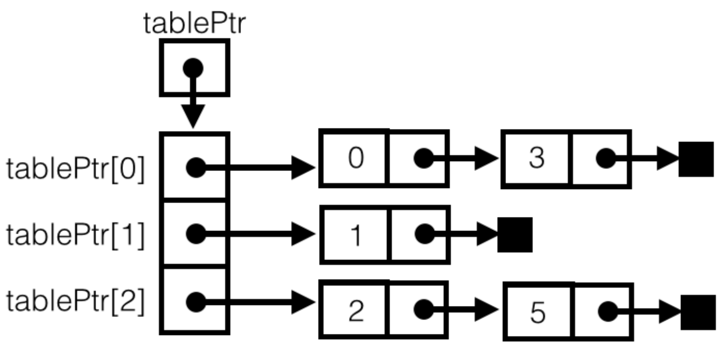

Figure 1: Example

For example, consider Figure 1. This is a pictorial representation of an example of a hash table that you will implement. `tablePtr` is a double pointer of type `struct Node` that represents the hash table. There are three buckets in this example. `tablePtr[0]` is a single pointer of type `struct Node` that points to the first node in the ordered linked list containing all nodes with keys that have a remainder of 0 when divided by 3 (i.e. 0%3 = 0, 3%3 = 0). You may notice that the nodes in the linked list associated with `tablePtr[0]` are arranged in ascending order (with respect to the values of the `data` members of these nodes). Similar, `tablePtr[1]` represents bucket 1 (i.e. all nodes with keys that have a remainder of 1 when divided by 3) and `tablePtr[2]` represents bucket 2 (all nodes with keys that have a remainder of 2 when divided by 3). Suppose that a key with value 4 is to be inserted into the hash table. The hash function will map this key to an index/bucket by computing 4%3. This results in the value 1. Key 4 is then mapped to bucket 1. The key is stored into a dynamically allocated structure of type `Node` and it is inserted into the ordered list represented by `tablePtr[1]`. This will result in key 4 being placed after the node containing the key 1. In order to create/perform these operations on the hash table, you will implement the following functions:

- `struct Node ** createTable(int buckets);`
- `int findHash(int key, int buckets);`
- `void insertTable(int key, int buckets, struct Node ** tablePtr);`
- `void freeTable(struct Node ** tablePtr, int buckets);`

The `createTable` function will dynamically allocate an array with space to store `buckets` number of single pointers of type `struct Node`. The double pointer representing the dynamically allocated space is returned by the function. `findHash` function computes the mapping of the key to the `bucket`/index by applying the `mod` operator to `key` (i.e. `key % buckets`). The resulting value represents the bucket that corresponds to the key. This is returned by the function. `insertTable` inserts a new key into the hash table `tablePtr`. This function will call on the `insertNode` function implemented in the previous part and passes to this function the `key` to be inserted and the single pointer representing the ordered list in the hash table into which the new key is to be inserted. The single pointer passed as an argument to the `insertNode` function is selected based on the results of the `findHash` function. Finally, the `freeTable` function will free all dynamically allocated elements in the hash table represented by the double pointer `tablePtr`. The number of `buckets` in the table is also passed as an argument to this function. Test your implementation via:

- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

Ensure that your output matches the sample output file `P2.txt` located in the `expOutput` folder.

3 Code Submission

For this lab, if your submit via `git`, you will receive **bonus** points. Otherwise you can submit through the `submitcsc190s` command on your ECF machine (no bonus points). Ensure that you submit through only one venue.

3.1 Submission through Git

Once you have completed this lab, you will submit your work by:

- Log onto your ECF account
- Browse into the directory you had cloned in Lab 0 (i.e. `cd ~/UTORID/`)
- Create a folder named `Lab4` (i.e. `mkdir Lab4`) in that cloned directory
- Ensure that your code compiles in the ECF environment
- Copy all your completed code (`lab4.h` and `lab4.c`) into the Lab4 folder
- Browse into the `~/UTORID/` directory
- Add all files in the Lab4 folder (i.e. `git add *`)

- Commit all files that have modified in the Lab 4 folder (i.e. `git commit -m "adding lab files"`)
- Push all changes committed to the `git` server (i.e. `git push origin master`)

3.2 Submission through `submitcsc190s`

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory containing your completed code (`lab4.h` and `lab4.c`)
- Submit by issuing the command:
  `submitcsc190s 5 lab4.h lab4.c`

3.3 Checklist

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error in the ECF environment (if it does not compile then your maximum grade will be 3/20)
- Do not resubmit any files in Lab 4 after the deadline (otherwise we will consider your work to be a late submission)