Southern New Hampshire University
Joshua Wozny
joshua.wozny@snhu.edu
January 13, 2023

CS-300 Week 4: HashTable

REFLECTION

This week we implemented a hash table data structure in C++. This is the second of several data structures that we will implement in C++ to better understand the advantages and disadvantages of each. Each data structure we will be studying can be used to store data for retrieval later using different techniques.

The hash table stores data into buckets for faster storage and retrieval using a hash to divide a dataset into a relatively equal distribution of buckets with a predefined number of buckets. I use a vector with each element of the vector acting as a “bucket”. Within each element of the vector I am storing data as a LinkedList to manage each node of data which store a point of data as well as a pointer the next record in the list - hence the name linked list (see CS-300 Week 3 Reflection for more details).

The biggest challenge was making sure I was initializing each element within the vector correctly, and accessing each associated LinkedList correctly. After getting my pointers correct alot of the rest fell into place. I was able to reuse most of the code from previous modules, particularly LinkedList, allowing me to put most of my energy into thinking about how I wanted it to work in a modular and reusable fashion.

PSEUDOCODE

<u>HashTable</u>

- Methods needed:
    - HashTable();
    - HashTable(int size);
    - virtual ~HashTable();
    - void Insert(Bid bid);
    - void PrintAll();
    - void Remove(string bidId);
    - Bid Search(string bidId);
- HashTable()
    - Set tableSize = DEFAULT value:179;
    - Initialize vector(as table) which will store hashed Bids, vector to be intialized to default number of elements (as defined by tableSize)

- HashTable(int size)
  - Set tableSize to size
  - Initialize vector(as table) which will store hashed Bids, vector to be intialized to default number of elements (as defined by tableSize)
- ~HashTable()
  - vector handles its own memory management, good practice to set table to an empty vector to free up memory at the earliest possible time.
- hash(int bidId) = bidId % tableSize
- Insert(Bid bid)
  - Create a key = hash(bid.BidId (converted from a string to an integer))
  - Create new node
  - If table.at(key) is empty, intialize a LinkedList with the bid
  - Otherwise LinkedList.Append(bid)
- PrintAll()
  - start at element 0 of table
  - if element is empty, continue to next element
  - otherwise LinkedList.PrintList()
    - output current key, bidID, title, amount and fund of each bid in LinkedList
  - continue for each element within table
- Remove(string bidId)
  - Linked List table.at(hash(bidId))
    - if element is empty, do nothing
    - otherwise, LinkedList.Remove(bidId)
- Search(string bidId)
  - Linked List table.at(hash(bidId))
    - if element is empty, return empty bid;
    - otherwise, LinkedList.Search(bidId)