Name:- Chad Huntebrinker

Answer the following Questions

**Q1: [3 Points]** discuss the following statement: Open addressing is usually faster than chained hashing when the load factor is low.

|  |
| --- |
| Use this space to answer the question  This statement is true. Open addressing is usually faster than chaining when the load factor is low. The reason why is because if there is collision on a hash table using chaining, then there is now two variables in that spot with a pointer connecting them (meaning more complex which means more time required searching). This causes complexity and makes open addressing a better choice. However, as the load factor increases, the effectiveness of open addressing goes down and chaining becomes the better option for a hash table. In conclusion, the statement “Open addressing is usually faster than chained hashing when the load factor is low” is true. |

**Q2: [5 points]** Use the numbers listed below to create an AVL tree

15, 31, 7, 1, 13, 21, 19, 23, 5, 36, 87, 4, 73

Show every rotation or update applied to the tree throw the process of building the tree.

|  |
| --- |
| Use this space to answer the question                We have a case of left-left, so we must adjust 19, 21, and 31. So, we must balance the tree:            We have a case of right-right. So, we must balance the whole tree with the number by replacing 21 with 31. Also, the right child for 21 will be 23 now:      We now have a cases of right-left with 4, 5, 1. First, we must get a right-right:    Then, we balance it:      We have a case of right-left with 36, 87 73. First, we must get a right-right:    Then, we balance it:    So, the final tree is: |

**Q3: [4 points]** Compare the storage requirement for a hash table with open addressing, a table size of 750, and a load factor of 0.75 with a hash table that uses chaining and gives the same performance.

|  |
| --- |
| Use this space to answer the question  I’m assuming that table size means the number of elements that will be put into the hash tables. If that is the case, then 750 will be the number of filled cells for open addressing (because 1 cell = 1 spot in open addressing hash table).  Open addressing Load Factor: 0.75 = 750 / x ---> x = 1000 (this is the storage size of the hash table)  Chaining:  (Note left side of the equation is open addressing performance, right side is chaining performance, and L is the Load Factor for the chained hash table).  ½ (1 + (1/(1 – 0.75))) = 1 + L / 2  5/2 = 1 + L / 2  3 = L  3 = 750 / x ---> x = 250  So, open addressing requires a hash table with size of 1,000 while chaining requires a hash table size of 250 in order that both have the same performance. |

**Q4: [8 points]** Use chaining method CPP files provided in Module 11 to complete the following:

1- write a function to add a pair element to the hash table.

2- write a function to remove an element to the hash table.

3- write a method to traverse the hash table.

4-write a method to rehash the hash table using a size passed as argument, the size should be grater than the original hash table size. **NOTE: you will need to apply some modification to the changing cpp file.**

|  |
| --- |
| Use this space to answer the question  //Chad Huntebrinker  //CS 303  #include<list>  #include<iostream>  #include<iterator>  #include<utility>  #include<string>  #include<cmath>  using namespace std;  void addPair(list<pair<string, char>>\*& hashTable, int hashSize, string key, char userChar);  void deleteItem(list<pair<string, char>>\*& hashTable, int hashSize, string key, char userChar);  void goThruHashTable(list<pair<string, char>>\* hashTable, int hashSize);  void reHash(list<pair<string, char>>\* &hashTable, int &hashSize);  void main() {  //First, we are going to populate the hash table with some values.  //Put those values in an array.  pair<string, char> orig[] = { pair<string,char>("r98",'A'),  pair<string,char>("11y",'L'),  pair<string,char>("q54",'G'),  pair<string,char>("p88",'O'),  pair<string,char>("bb1",'R'),  pair<string,char>("www",'I'),  pair<string,char>("nbn",'T'),  pair<string,char>("pop",'H'),  pair<string,char>("87u",'M'),  pair<string,char>("ede",'S'),  pair<string,char>("kk8",'Z'),  pair<string,char>("711",'L'),  pair<string,char>("mom",'G'),  pair<string,char>("28m",'O')  };  int dataSize = \*(&orig + 1) - orig;  int hashSize = 20;  char userInput, userValue;  list<pair<string, char>>\* hashTab = new list<pair<string, char>>[hashSize];  int index;  string userKey;  //Create the original hash table  for (int i = 0; i < dataSize; i++) {  index = 0;  for (int x = 2; x >= 0; x--) {  index += ((int)((orig[i].first)[x]) \* (int)pow(31, x)); //((int)orig[i]\* 31) % 11;  }  index %= hashSize;  hashTab[index].push\_back(orig[i]);  }  //Make a loop for the user input.  do {  cout << "Enter 1 for Add" << endl;  cout << "Enter 2 for Delete" << endl;  cout << "Enter 3 for Traverse" << endl;  cout << "Enter 4 for Re-Hash" << endl;  cout << "Enter 5 for Quit" << endl << endl;  cin >> userInput;  if (userInput == '1') {  cout << "Please enter the key of the value you want to add (must be 3 characters): ";  cin >> userKey;  cout << "Please enter the character you want to add: ";  cin >> userValue;  addPair(hashTab, hashSize, userKey, userValue);  }  else if (userInput == '2') {  cout << "Please enter the key of the value you want to delete (must be 3 characters): ";  cin >> userKey;  cout << "Please enter the character you want to delete: ";  cin >> userValue;    deleteItem(hashTab, hashSize, userKey, userValue);  }  else if (userInput == '3') {  goThruHashTable(hashTab, hashSize);  }  else if (userInput == '4') {  reHash(hashTab, hashSize);  }  } while (userInput != '5');  delete[] hashTab;  system("pause");  }  //This function adds a new value to the hash table from the user's inputted key and value.  void addPair(list<pair<string, char>>\* &hashTable, int hashSize, string key, char userChar) {  pair<string, char> temp(key, userChar);    int index = 0;  for (int x = 2; x >= 0; x--) {  index += ((int)((temp.first)[x]) \* (int)pow(31, x));  }  index %= hashSize;  hashTable[index].push\_back(temp);  }  //This function deletes a value from the hash table if both the user's key and value match up  //to a key and value in the hash table. Otherwise, an error message is outputted.  void deleteItem(list<pair<string, char>>\* &hashTable, int hashSize, string key, char userChar) {  pair<string, char> temp(key, userChar);  int index = 0;    //Find the index of the value.  for (int x = 2; x >= 0; x--) {  index += ((int)((temp.first)[x]) \* (int)pow(31, x));  }  index %= hashSize;  list<pair<string, char>>::iterator b = hashTable[index].begin();  list<pair<string, char>>::iterator e = hashTable[index].end();  for (; b != e; b++) {    //If the keys and values match up, delete them.  if (b->second == userChar && b->first == key) {  hashTable[index].remove(\*b);  break;  }    //Otherwise, output an error message.  else {  cout << "This item does not exist with that key." << endl;  }  }  }  //This function iterates and outputs the hash table.  void goThruHashTable(list<pair<string, char>>\* hashTable, int hashSize) {  for (int i = 0; i < hashSize; i++) {  cout << i << ": ";  list<pair<string, char>>::iterator b = hashTable[i].begin();  list<pair<string, char>>::iterator e = hashTable[i].end();  for (; b != e; b++) {  cout << b->second << "->";  }  cout << "NULL" << endl;  }  }  //This function rehashes the hash table only if the user's inputted size is larger than the size  //of the current hash table.  void reHash(list<pair<string, char>>\* &hashTable, int &hashSize) {  int newSize;  int index;  //Make sure the user's inputted size is larger than the original size.  do {  cout << "Please enter the new size you want the hashTable to be: ";  cin >> newSize;  if (newSize <= hashSize) {  cout << "Incorrect value. Must be greater than the size of the current Hashtable." << endl << endl;  }  } while (newSize <= hashSize);  list<pair<string, char>>\* newHashTable = new list<pair<string, char>>[newSize];  //Rehash the old hash table's values into the new hash table  for (int i = 0; i < hashSize; i++) {  list<pair<string, char>>::iterator b = hashTable[i].begin();  list<pair<string, char>>::iterator e = hashTable[i].end();  index = 0;  while (b != e) {  for (int x = 2; x >= 0; x--) {  index += ((int)((b->first)[x]) \* (int)pow(31, x)); //((int)orig[i]\* 31) % 11;  }  index %= newSize;  newHashTable[index].push\_back(\*b);  ++b;  }  }  //Delete the old hash table's values and put in the new one.  delete[] hashTable;  hashTable = newHashTable;  hashSize = newSize;  } |

**Q5: [5 points]** For the items in the 8-element table below, compute hash\_fcn(key) % table.size() for table sizes of 8 and 13. What would be the position of each item in tables of these sizes using open addressing and chaining? (use tables when answering the question and show the details of how you hashed the values )

**Name hash\_fun()**

"Kim" 274

"Adam" 9869

"John" 96448

"Steve" 2879

"Ali" 38

"Sarah" 100

"Tom" 2227

"Messi" 8191

|  |
| --- |
| Use this space to answer the question  Table Size: 8  Open Addressing    Chaining    Table size: 13  Open Addressing    Chaining |