```
// header file
#include "HashUtility.h"

// constants
const int MIN_HASH_CHARACTERS = 10;

// functions
/*
Name: addHashItem
Process: adds new item to hash table
Function input/parameters: pointer to hash table (HashTableType *),
                           city name (const char *), city population (int)
Function output/parameters: updated pointer to hash table (HashTableType *)
Function output/returned: none
Device input/---: none
Device output/---: none
Dependencies: generateHashIndex, initializeCityNodeFromData, insert
*/
void addHashItem( HashTableType *hashData, char *cityName, int cityPop )
 {
    // declare variables
    CityDataType *newNode;
    int element;

    // create a new node
     // function: initializeCityNodeFromData
    newNode = initializeCityNodeFromData(cityName, cityPop);

    // create a new index
     // function: generateHashIndex
    element = generateHashIndex(*hashData, cityName);

    // insert the new node at the hash index
     // function: insert
    hashData->table[element] = insert(hashData->table[element], *newNode);
 }

/*
Name: clearHashTable
Process: clears contents of hash table, and then hash table itself
Function input/parameters: pointer to hash table (HashTableType *)
Function output/parameters: none
Function output/returned: NULL
Device input/---: none
Device output/---: none
Dependencies: clearTree, free
*/
HashTableType *clearHashTable( HashTableType *hashData )
 {
  // declare variables
  int item;

  // loop through the table
  for(item = 0; item < hashData->capacity; item++)
   {
      // clear each element
        // function: clearTree
      clearTree(hashData->table[item]);

   }

  // clear the data
   // function: free
   free(hashData->table);

  // clear the table
   // function: free
   free(hashData);

  // return null
   return NULL;
 }

/*
Name: copyHashTable
Process: creates new hash table and makes duplicate
Function input/parameters: pointer to source hash table (HashTableType *)
Function output/parameters: none
Function output/returned: pointer to new hash table (HashTableType *)
Device input/---: none
Device output/---: none
Dependencies: initializeHashTable, copyTree
*/
HashTableType *copyHashTable( HashTableType *source )
 {
   // declare variables
```

```
   HashTableType *tableCopy;
   int item;

   // initialize new table
    // function: initializeHashTable
   tableCopy = initializeHashTable(source->capacity);

   // loop through the table
   for(item = 0; item < source->capacity; item++)
    {
      // copy data from old table to new one
       // function: copyTree
      tableCopy->table[item] = copyTree(source->table[item]);
    }
   // copy over the capacity
   tableCopy->capacity = source->capacity;

   // return the new table
   return tableCopy;
 }

/*
Name: findMean
Process: finds the mean of a set of integers
Function input/parameters: integer array (int *), size (int)
Function output/parameters: none
Function output/returned: mean of values (double)
Device input/---: none
Device output/---: none
Dependencies: none
*/
double findMean( int *array, int size )
 {
    // declare variables
    double mean = 0;
    int item;

    // loop through the array
    for(item = 0; item < size; item++)
     {
       // add up all the elements
       mean += array[item];
     }

    // divide the sum by the size
    mean = mean / size;

    // return the mean
    return mean;
 }

/*
Name: findMedian
Process: finds the median of a set of integers,
         assumes all input arrays will have an odd number of values
Function input/parameters: integer array (int *), size (int)
Function output/parameters: none
Function output/returned: median of values (int)
Device input/---: none
Device output/---: none
Dependencies: none
*/
int findMedian( int *array, int size )
 {
    // declare variables
    int outer, inner;
    int minValue, newMedian;
    int middleVal = 0;

    // loop through the array
    for(outer = 0; outer < size - 1; outer++)
     {
       // set min
       minValue = outer;

       // loop through sub array
       for(inner = outer + 1; inner < size; inner++)
        {
         // check if current < min
         if(array[inner] < array[minValue])
          {
            // set the current as min
            minValue = inner;
          }
        }
        // if current > min
        if(array[outer] > array[minValue])
```

```
        {
          // swap elements
          newMedian = array[outer];
          array[outer] = array[minValue];
          array[minValue] = newMedian;
        }
     }

    // check if there's one median
    if(size % 2 != 0)
     {
       // find the median
       middleVal = array[(size - 1) /2];
     }
    // otherwise assume there's two
    else
     {
       // find median of the two
       middleVal = (array[size / 2 - 1] + array[size / 2]) / 2;
     }

    // return the median
    return middleVal;
}


/*
Name: generateHashIndex
Process: finds hashed index for given data item,
         sums integer values of city name characters,
         if city name length is less than MINIMUM_HASH_CHARACTERS,
         repeats going over the city letters as needed to meet this minimum
Function input/parameters: hash table (const HashTableType),
                           city name (const char *)
Function output/parameters: none
Function output/returned: generated hash index (int)
Device input/---: none
Device output/---: none
Dependencies: strlen
*/
int generateHashIndex( const HashTableType hashData, const char *cityName )
 {
   // declare variables
   int index, newHash = 0;
   int name = strlen(cityName);
   int items = strlen(cityName);

   // check if the length of cityName is under the min
   if(items < MIN_HASH_CHARACTERS)
    {
      // set the length to the min
      name = MIN_HASH_CHARACTERS;
    }
   // loop through the string
   for(index = 0; index < name; index++)
    {
      // add up the hash
      newHash += cityName[index % items];
    }
   // return the hash
   return newHash % hashData.capacity;

 }

/*
Name: getHashDataFromFile
Process: uploads data from city file with unknown number of data sets,
         provides Boolean parameter to display data input success
Function input/parameters: file name (char *), capacity (int),
                           verbose flag (bool)
Function output/parameters: none
Function output/returned: pointer to newly created hash table
                            (HashTableType *)
Device input/file: data from HD
Device output/---: none
Dependencies: openInputFile, initializeHashTable, readStringToDelimiterFromFile,
              readStringToLineEndFromFile, checkForEndOfInputFile,
              readCharacterFromFile, readIntegerFromFile,
              addHashItem, printf, closeInputFile
*/
HashTableType *getHashDataFromFile( const char *fileName,
                                                int capacity, bool verbose )
 {
  // declare variables
   HashTableType *dataHolder = NULL;
   char header[STD_STR_LEN];
   int nameRank;
```

```
   char cityName[STD_STR_LEN];
   int population;

   // open provided file
    // function: openInputFile
   if(openInputFile(fileName))
    {
     // if the display flag is true
         if(verbose)
          {
            // print the loading statement
             // function: printf
             printf("\n Begin Loading Data From File . . . \n");
          }
    // prime the loop by reading the header
     // function: readStringToLineEndFromFile
    readStringToLineEndFromFile(header);

      // initialize the table
       // function: initializeHashTable
       dataHolder = initializeHashTable(capacity);

    // read the rank
     // function: readIntegerFromFile
    nameRank = readIntegerFromFile();

    // loop through the file
      // function: checkForEndOfInputFile
     while(!checkForEndOfInputFile())
      {

       // read the comma
        // function: readCharacterFromFile
       readCharacterFromFile();

       // read the city
        // function: readStringToDelimiterFromFile
       readStringToDelimiterFromFile(COMMA, cityName);

       // read the population
       // function: readIntegerFromFile
       population = readIntegerFromFile();


       // add item
       // function: insert
       addHashItem(dataHolder, cityName, population);

       // if verbose is true
       if(verbose)
        {
         // print out data
          // function: printf
         printf("%d) City Name: %s, Population: %d\n", nameRank, cityName,
                                                                  population);

        }



       // print out the next number
        // function: readIntegerFromFile
       nameRank = readIntegerFromFile();

     }
     // if verbose is true
       if(verbose)
        {
         // print out end of file message
         printf("\t\t\t\t\t\t    . . . End Loading Data From File\n\n");
        }
    }
    // close the file
      // function: closeInputFile
    closeInputFile();

    // return pointer holding the data
    return dataHolder;
 }

/*
Name: initializeHashTable
Process: dynamically creates new hash table with internal components
Function input/parameters: capacity (int)
Function output/parameters: none
Function output/returned: pointer to newly created hash table
                          (HashTableType *)
```

```
Device input/file: data from HD
Device output/---: none
Dependencies: malloc w/sizeof, initializeBST
*/
HashTableType *initializeHashTable( int capacity )
 {
   // declare variables
   HashTableType *newHash;
   int index;

   // make space for hash table
   newHash = (HashTableType*)malloc(sizeof(HashTableType));

   // make space for the table
   newHash->table = (CityDataType**)malloc(sizeof(CityDataType*)*capacity);

   // if the hash pointer is not null
   if(newHash != NULL)
    {
      // set up the capacity
      newHash->capacity = capacity;

      // loop through the hash table
      for(index = 0; index < newHash->capacity; index++)
       {
         // assign each item to null
         initializeBST(&newHash->table[index]);
       }
      // return new hash table
      return newHash;
    }
    // return null
    return NULL;
 }

/*
Name: removeHashItem
Process: acquires hashed item, returns
Function input/parameters: pointer to hash table (HashTableType *),
                           city name (const char *)
Function output/parameters: none
Function output/returned: pointer to removed item (CityDataType *), or NULL
Device input/---: none
Device output/---: none
Dependencies: generateHashIndex, removeItem
*/
CityDataType *removeHashItem( HashTableType *hashData, const char *cityName )
 {
   // declare variables
   int items;

   // generate the new index
    // function: generateHashIndex
   items = generateHashIndex(*hashData, cityName);

   // return the data without the removed item
    // function: removeItem
   return removeItem(&hashData->table[items], cityName);
 }

/*
Name: searchHashTable
Process: searches for value in table, returns pointer if found
Function input/parameters: hash table (const HashTableType),
                           city name (const char *)
Function output/parameters: none
Function output/returned: pointer to found item (CityDataType *), or NULL
Device input/---: none
Device output/---: none
Dependencies: generateHashIndex, search
*/
CityDataType *searchHashTable( const HashTableType hashData,
                                                    const char *cityName )
 {
    // declare variables
    int items;
    CityDataType *searchCity;

    // generate a new index
     // function: generateHashIndex
    items = generateHashIndex(hashData, cityName);

    // search for the value in table
     // function: search
    searchCity = search(hashData.table[items], cityName);

    // return the search
```

```
    return searchCity;
 }

/*
Name: showHashTableStatus
Process: displays item counts from each BST element in the hash table,
         displays highest and lowest number of items in an element,
         displays range between highest and lowest,
         displays the mean and median,
         and displays the total number of nodes found,
         all in a formatted structure
Function input/parameters: hashTable (const HashTableType)
Function output/parameters: none
Function output/returned: none
Device input/---: none
Device output/monitor: hash data status displayed as specified
Dependencies: malloc w/sizeof, countTreeNodes, printf,
              findMean, findMedian, free
*/
void showHashTableStatus( const HashTableType hashData )
 {
   // declare and initialize variables
   int index, totalNums = 0;
   double mean;
   int median, range;
   int *numArray, minCount, maxCount;


   // create space for int array
     // function: malloc w/ sizeof
   numArray = (int*)malloc(sizeof(int)*hashData.capacity);

   // loop through the array
   for(index = 0; index < hashData.capacity; index++)
      {
       // update the count
        // function: countTreeNodes
       numArray[index] = countTreeNodes(hashData.table[index]);
      }

    // initialize the counts after filling up the array
    minCount = numArray[0];
    maxCount = numArray[0];

   // print the BST items title
    // function: printf
    printf("\nBST Items:     ");

   // loop up to the capacity
   for(index = 0; index < hashData.capacity; index++)
    {
     // count the total nodes
     totalNums += numArray[index];

     // if the num of nodes are less than the min
     if(numArray[index] < minCount)
      {
        // reassign the min
        minCount = numArray[index];
      }
     // else if the num of nodes are greater than the max
     else if(numArray[index] > maxCount)
      {
        // reassign the max
        maxCount = numArray[index];
      }
     // display the array
      // function: printf
     printf("%3d ", numArray[index]);
    }

   // calculate the mean
    // function: findMean
   mean = findMean(numArray, hashData.capacity);

   // calculate the median
    // function: findMedian
   median = findMedian(numArray, hashData.capacity);

   // calculate the range
   range = maxCount - minCount;

   // reset the index
   index = 0;

   // print the spacing
    // function: printf
```

```
  printf("\n              ");

  // while at a specific index
  while(index < hashData.capacity)
   {
     // print out certain amount of dashes
      // function: printf
     printf( " ---");

     // increment the index
     index++;
   }

   // print the hash index title
    // function: printf
   printf("\nHash Index:    ");

   // iterate through the array
   for(index = 0; index < hashData.capacity; index++)
    {
      // display the index
       // function: printf
      printf("%3d ", index);
    }

  // display the data

   // display max nodes
    // function: printf
   printf("\nMax nodes in one element    :    %d\n", maxCount);

   // display min nodes
    // function: printf
   printf("Min num nodes in one element:    %d\n", minCount);

   // display range
    // function: printf
   printf("Range (min to max)          :    %d\n", range);

   // display mean
    // function: printf
   printf("Mean num nodes              : %0.2f\n", mean);

   // display median
    // function: printf
   printf("Median node num             :    %d\n", median);

   // display total amount of nodes
    // function: printf
   printf("Total nodes processed       :   %d\n", totalNums);


   // reset the index
   index = 0;

   // print on the next line
    // function: printf
   printf("\n");

   // while at a specific index
   while(index < hashData.capacity)
    {
      // print out tailored ending line
       // function: printf
      printf("====");

      // increment index
      index++;
    }
   // clear the array
   free(numArray);
}
```