```c
// header  files
#include "SimpleBST_Utility.h"

// functions

/*
Name: clearTree
Process: recursively deallocates tree nodes using a post order traversal
Function input/parameters: working pointer for recursion (StudentDataType *)
Function output/parameters: none
Function output/returned: empty tree (NULL)
Device input/file: none
Device output/monitor: none
Dependencies: free, clearTree (recursively)
*/
CityDataType *clearTree( CityDataType *wkgPtr )
 {
   // if the working pointer is not NULL
   if(wkgPtr != NULL)
    {
      // delete the left child
       // function: clearTree
      clearTree(wkgPtr->leftChildPtr);

      // delete the right child
       // function: clearTree
      clearTree(wkgPtr->rightChildPtr);

      // clear the working pointer
        // function: free
      free(wkgPtr);
    }
   // return the working pointer
    return wkgPtr;
 }

/*
Name: compareStrings
Process: compares two strings as follows:
         - if left string is greater than the right string,
         returns value greater than zero
         - if left string is less than the right string,
         returns value less than zero
         - if strings are equal but one is longer, longer one is greater
         - otherwise, returns zero
Function input/parameters: two strings to be compared (const char *)
Function output/parameters: none
Function output/returned: result as specified above (int)
Device input/file: none
Device output/monitor: none
Dependencies: strlen
*/
int compareStrings( const char *leftStr, const char *rightStr )
 {
   // declare variables
   int index = 0;
   int result = 0;

   // while there are still letters and if the results are not the same length
   while(result == 0)
    {
      // check if the lengths of the strings are the same
       // function: strlen
      if(index > strlen(leftStr) || index > strlen(rightStr))
       {
        // return 0
         return 0;
       }
      // count the difference
      result = leftStr[index] - rightStr[index];

      // increment the index
      index++;
    }
   // return the difference
   return result;
 }

/*
Name: copyTree
Process: recursively duplicates the provided tree
         using a pre order traversal strategy
Function input/parameters: working pointer for recursion (CityDataType *)
Function output/parameters: none
Function output/returned: pointer to duplicate tree (CityDataType *)
Device input/---: none
```

```
Device output/---: none
Dependencies: initializeCityNodeFromNode, copyTree (recursively)
*/
CityDataType *copyTree( CityDataType *wkgPtr )
 {
   // declare variables
   CityDataType *newNode;

   // if the working pointer is not NULL
   if(wkgPtr != NULL)
    {
     // create new node
      // function: initializeCityNodeFromNode
     newNode = initializeCityNodeFromNode(*wkgPtr);

     // copy the left child
      // function: copyTree
     newNode->leftChildPtr = copyTree(wkgPtr->leftChildPtr);

     // copy the right child
      // function: copyTree
     newNode->rightChildPtr = copyTree(wkgPtr->rightChildPtr);

     // return the copied tree
     return newNode;
    }
   // return null
   return NULL;
 }

/*
Name: countTreeNodes
Process: finds the number of nodes in a given BST
Function input/parameters: working pointer for recursion (CityDataType *)
Function output/parameters: none
Function output/returned: number of nodes found (int)
Device input/---: none
Device output/---: none
Dependencies: countTreeNodeHelper
*/
int countTreeNodes( CityDataType *rootPtr )
 {
   // declare variables
   int count = 0;

   // count the nodes
    // function: countTreeNodeHelper
   countTreeNodeHelper(rootPtr, &count);

   // return the count
   return count;
 }

/*
Name: countTreeNodeHelper
Process: recursively finds the number of nodes in a given BST
Function input/parameters: working pointer for recursion (CityDataType *)
Function output/parameters: pointer to number of nodes (int *)
Function output/returned: none
Device input/---: none
Device output/---: none
Dependencies: countTreeNodeHelper (recursively)
*/
void countTreeNodeHelper( CityDataType *cdtPtr, int *counter )
 {
   // if the pointer is not null
   if(cdtPtr != NULL)
    {
     // increment count
     *counter = *counter + 1;

     // count left children
      // function: countTreeNodeHelper
     countTreeNodeHelper(cdtPtr->leftChildPtr, counter);

     // count right children
      // function: countTreeNodeHelper
     countTreeNodeHelper(cdtPtr->rightChildPtr, counter);

    }
 }

/*
Name: displayData
Process: displays data sorted by city name
Function input/parameters: root pointer (CityDataType *)
Function output/parameters: none
```

```
Function output/returned: none
Device input/---: none
Device output/---: none
Dependencies: displayInOrder
*/
void displayData( CityDataType *rootPtr )
 {
   // initialize the count
   int *count = 0;

   // call the displayInOrder function
   displayInOrder(rootPtr, count);
 }

/*
Name: displayInOrder
Process: recursively displays tree with numbered values
         using in order traversal,
         dynamically creates and frees string for display
Function input/parameters: working pointer for recursion (CityDataType *),
                           pointer to count (int *)
Function output/parameters: none
Function output/returned: none
Device input/file: none
Device output/monitor: none
Dependencies: cityDataToString, malloc, sizeof, printf, free,
              displayInOrder (recursively)
*/
void displayInOrder( CityDataType *cdtPtr, int *count )
 {
   // declare and initialize variables
    // function: malloc w/ sizeof
   char *destStr = (char*)malloc(sizeof(char)*MAX_STR_LEN);

   // if the root is not null
   if(cdtPtr != NULL)
    {
      // display the left child
       // function: displayInOrder
      displayInOrder(cdtPtr->leftChildPtr, count);

      // store the data in destStr
       // function: cityDataToString
      cityDataToString(destStr, *cdtPtr);

      // display the root
       // function: printf
      printf("%s\n", destStr);

      // display the right child
       // function: displayInOrder
      displayInOrder(cdtPtr->rightChildPtr, count);

    }
 }

/*
Name: findMax
Process: finds the maximum of two integer values
Function input/parameters: two integer values (int)
Function output/parameters: none
Function output/returned: larger of the two numbers (int)
Device input/---: none
Device output/---: none
Dependencies: none
*/
int findMax( int oneVal, int otherVal )
 {
   // declare variables
   int maxVal;

   // if oneVal is bigger
   if(oneVal > otherVal)
    {
      // oneVal is maximum
      maxVal = oneVal;
    }
   // else
   else
    {
       // otherVal is maximum
       maxVal = otherVal;
    }
    // return the max
    return maxVal;
 }
```

```
/*
Name: findOptimalTreeHeight
Process: finds the optimal height of a given BST
Function input/parameters: pointer to root (CityDataType *)
Function output/parameters: none
Function output/returned: optimal tree height (int)
Device input/---: none
Device output/---: none
Dependencies: countTreeNodes
*/
int findOptimalTreeHeight( CityDataType *cdtPtr )
 {
   // declare variables
   int optHeight = 0;
   int numNodes;

   // count the nodes
    // function: countTreeNodes
   numNodes = countTreeNodes(cdtPtr);

   // loop until max nodes are created
   while(numNodes > 0)
    {
      // divide each time by 2
      numNodes /= 2;

      // increment height
      optHeight++;
    }
   // return the optimal height
   return optHeight;
 }

/*
Name: findActualTreeHeight
Process: recursively finds the actual height of a given BST
Function input/parameters: pointer to root (CityDataType *)
Function output/parameters: none
Function output/returned: actual tree height (int)
Device input/---: none
Device output/---: none
Dependencies: findActualTreeHeight (recursively), findMax
*/
int findActualTreeHeight( CityDataType *cdtPtr )
 {
   // declare variables
   int leftSide = 0;
   int rightSide = 0;

   // if the pointer is not null
   if(cdtPtr != NULL)
    {
     // find length of left side
      // function: findActualTreeHeight
     leftSide = findActualTreeHeight(cdtPtr->leftChildPtr);

     // find length of right side
      // function: findActualTreeHeight
     rightSide = findActualTreeHeight(cdtPtr->rightChildPtr);

     // find max height of tree + 1
      // function: findMax
     return findMax(leftSide, rightSide) + 1;
    }
   // return -1
   return -1;
 }

/*
Name: getBstDataFromFile
Process: uploads data from file with unknown number of data sets,
         allows for Boolean flag to show upload
Function input/parameters: file name (char *), verbose flag (bool)
Function output/parameters: none
Function output/returned: pointer to newly created BST
Device input/file: data from HD
Device output/monitor: none
Dependencies: openInputFile, readStringToLineEndFromFile, initializeBST
              checkForEndOfInputFile, readIntegerFromFile,
              readCharacterFromFile, readStringToDelimiterFromFile,
              readStringToLineEndFromFile,  initializeCityNodeFromData,
              insert, printf, closeInputFile
*/
CityDataType *getBstDataFromFile( const char *fileName, bool verbose )
 {
   // declare variables
   CityDataType *dataHolder = NULL;
```

```
  CityDataType *tempHolder = 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);


    // 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();

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

       }
      // initialize the node
       // function: initializeCityNodeFromData
      tempHolder = initializeCityNodeFromData(cityName, population);

      // insert data
       // function: insert
      dataHolder = insert(dataHolder, *tempHolder);

      // clear the temp
       // function: free
      free(tempHolder);

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


     }
    // if verbose is true
      if(verbose)
       {
        // print out end of file message
         // function: printf
        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: initializeBST
Process: sets BST root pointer to NULL, root pointer is returned by address
Function input/parameters: address of working pointer (CityDataType **)
Function output/parameters: address of updated working pointer
                            (CityDataType **)
Function output/returned: none
Device input/file: none
Device output/monitor: none
Dependencies: none
*/
void initializeBST( CityDataType **bstPtr )
 {
   // set bstPtr to null
   *bstPtr = NULL;
 }

/*
Name: initializeCityNodeFromData
Process: captures data from individual data items,
         dynamically creates new node,
         copies data, and returns pointer to new node
Function input/parameters: data to be copied (char *, int, double)
Function output/parameters: none
Function output/returned: pointer to new node as specified (CityDataType *)
Device input/file: none
Device output/monitor: none
Dependencies: malloc, sizeof, strcpy
*/
CityDataType *initializeCityNodeFromData( const char *name, int population )
 {
   // declare variables
    CityDataType *newNode;

   // create space for a new node
    // function: malloc w/ sizeof
    newNode = (CityDataType*)malloc(sizeof(CityDataType));

   // initialize name
    // function: strcpy
    strcpy(newNode->name, name);

   // initialize population
    newNode->population = population;

   // set left child to null
    newNode->leftChildPtr = NULL;

   // set right child to null
    newNode->rightChildPtr = NULL;

   // return the new node
    return newNode;
 }

/*
Name: initializeCityNodeFromNode
Process: captures data from source node pointer, dynamically creates new node,
         copies data, and returns pointer to new node
Function input/parameters: node to be copied (CityDataType)
Function output/parameters: none
Function output/returned: pointer to new node as specified (CityDataType *)
Device input/file: none
Device output/monitor: none
Dependencies: initializeCityNodeFromData
*/
CityDataType *initializeCityNodeFromNode( const CityDataType source )
 {
   // return initializeCityNodeFromData
   return initializeCityNodeFromData(source.name, source.population);
 }

/*
Name: insert
Process: recursively searches for available location in BST,
         creates new node and returns it to the calling function,
         if node is already in tree, data is overwritten
Function input/parameters: working pointer for recursion (CityDataType *),
                           node to be inserted (const CityDataType)
Function output/parameters: none
Function output/returned: pointer to new node and subtrees
                          as specified (CityDataType *)
Device input/file: none
Device output/monitor: none
Dependencies: compareStrings, setCityNodeData, initializeCityNodeFromNode,
              insert (recursively)
```

```
*/
CityDataType *insert( CityDataType *wkgPtr, const CityDataType inData )
 {

   // check if wkg pointer is null
   if(wkgPtr != NULL)
    {
      // check if the name is found
       // function: compareStrings
      if(compareStrings(inData.name, wkgPtr->name) == 0)
       {
        // overwrite the data
         // function: setCityNodeData
         setCityNodeData(wkgPtr, inData);

       }
      // if left < right
       // function: compareStrings
      else if(compareStrings(inData.name, wkgPtr->name) < 0)
       {
         // recurse with the left child
          // function: insert
         wkgPtr->leftChildPtr = insert(wkgPtr->leftChildPtr, inData);

       }
      // if left > right
       // function: compareStrings
      else if(compareStrings(inData.name, wkgPtr->name) > 0)
       {
        // recurse with the right child
         // function: insert
        wkgPtr->rightChildPtr = insert(wkgPtr->rightChildPtr, inData);

       }
    }
    else
     {
       // otherwise create a new node
       return initializeCityNodeFromNode(inData);

     }
   // return the working pointer
   return wkgPtr;
 }

/*
Name: isEmpty
Process: tests root node for NULL, returns result
Function input/parameters: pointer to root node (CityDataType *)
Function output/parameters: none
Function output/returned: result of test as specified (bool)
Device input/file: none
Device output/monitor: none
Dependencies: none
*/
bool isEmpty( CityDataType *sdtPtr)
 {
   // test if root is null
   return sdtPtr == NULL;
 }

/*
Name: removeFromMax
Process: recursively searches for max node,
         when found, node is unlinked from tree and returned
Function input/parameters: pointer to parent and child nodes (CityDataType *)
Function output/parameters: none
Function output/returned: pointer to removed node (CityDataType *)
Device input/file: none
Device output/monitor: none
Dependencies: none
*/
CityDataType *removeFromMax( CityDataType *parentPtr,
                                                CityDataType *childPtr )
 {
   // if the right child isn't null
   if(childPtr->rightChildPtr != NULL)
    {
      // return max
       // function: removeFromMax
      return removeFromMax(childPtr, childPtr->rightChildPtr);
    }
   // parent's right child is now the child's left child
   parentPtr->rightChildPtr = childPtr->leftChildPtr;

   // return the child pointer
   return childPtr;
```

```
 }

/*
Name: removeItem
Process: searches for item, if found, creates new node from search pointer,
         then removes item from tree using helper function,
         otherwise returns NULL
Function input/parameters: address of pointer to root node (CityDataType *),
                           node to be removed with at least city name key
Function output/parameters: address of updated root node pointer
                            (CityDataType **)
Function output/returned: pointer to dynamically created duplicate
                          of removed item (CityDataType *)
Device input/file: none
Device output/monitor: none
Dependencies: search, initializeCityNodeFromNode, removeItemHelper
*/
CityDataType *removeItem( CityDataType **rootPtr, const char *cityName )
 {
   // declare variables
   CityDataType *searching;
   CityDataType *newNode;

   // look for item
    // function: search
   searching = search(*rootPtr, cityName);

   // if the search is null
   if(searching != NULL)
    {
     // if found, create new node
      // function: initializeCityNodeFromNode
     newNode = initializeCityNodeFromNode(*searching);

     // then remove item
      // removeItemHelper
     *rootPtr = removeItemHelper(*rootPtr, cityName);

     // return new node
     return newNode;
    }
   // return null
   return NULL;
 }

/*
Name: removeItemHelper
Process: recursively searches for item, removes node,
         returns dynamic memory of removed node to OS,
         returns updated link to parent (at each recursive level),
         only one return at end of function
Function input/parameters: pointer to working node (CityDataType *),
                           node to be removed with at least city name key
                           (const CityDataType)
Function output/parameters: none
Function output/returned: link to recursive parent
Device input/file: none
Device output/monitor: none
Dependencies: compareStrings, setCityNodeData, removeFromMax, free,
              removeItemHelper (recursively)
*/
CityDataType *removeItemHelper( CityDataType *wkgPtr, const char *cityName )
 {
   // declare variables
    CityDataType *removedMax;

     // check if value is less than current
      // function: compareStrings
     if(compareStrings(cityName, wkgPtr->name) < 0)
      {
        // assign left child to recursion to the left
         // function: removeItemHelper
       wkgPtr->leftChildPtr = removeItemHelper(wkgPtr->leftChildPtr, cityName);

      }

     // check if value is greater than current
       // function: compareStringSegments
     else if(compareStrings(cityName, wkgPtr->name) > 0)
      {
        // assign right child to recursion to the right
         // function: removeItemHelper
        wkgPtr->rightChildPtr =
                          removeItemHelper(wkgPtr->rightChildPtr, cityName);
      }
```

```
// - result: we found it!


   // check for left node NULL/not there
   else if(wkgPtr->leftChildPtr == NULL)
    {
      // assign pointer to right node
      wkgPtr = wkgPtr->rightChildPtr;

    }

   // check for right node NULL/not there
   else if(wkgPtr->rightChildPtr == NULL)
    {
      // assign pointer to left node
      wkgPtr = wkgPtr->leftChildPtr;

    }


// result: handled easy stuff


    // check for left child has right child
    else if(wkgPtr->leftChildPtr->rightChildPtr != NULL)
     {

      // call removeFromMax to that left child
      removedMax = removeFromMax(wkgPtr, wkgPtr->leftChildPtr);

      // replace the data in the removed node with what was in the max
       // function: setContractorNodeData
      setCityNodeData(wkgPtr, *removedMax);

      // then must deallocate node
       // function: free
      free(removedMax);
     }

// - assume left child has no right child
    else
     {
       // - replace the data in the removed node with the left child data
        // function: setContractorNodeData
       setCityNodeData(wkgPtr,  *wkgPtr->leftChildPtr);

       // put left child in temp
       removedMax = wkgPtr->leftChildPtr;

       // - link around the left child
       wkgPtr->leftChildPtr = wkgPtr->leftChildPtr->leftChildPtr;

       // - don't forget to deallocate
         // function: free
       free(removedMax);
     }

   // return the working pointer
   return wkgPtr;

 }

/*
Name: search
Process: recursively searches for item, if found, returns pointer to node,
         otherwise, returns NULL
Function input/parameters: pointer to working node (CityDataType *),
                           node to be found with city name key
                           (const char *)
Function output/parameters: none
Function output/returned: link to found node, or NULL, as specified
Device input/file: none
Device output/monitor: none
Dependencies: compareStrings, search (recursively)
*/
CityDataType *search( CityDataType *wkgPtr, const char *cityName )
 {
  // if the node is not null
    if(wkgPtr != NULL)
     {

       // if item is found
        // function: compareStringSegments
       if(compareStrings(cityName, wkgPtr->name) == 0)
        {
          // return working pointer
          return wkgPtr;
```

```
        }
       // if item is not found
        // function: compareStringSegments
       else if(compareStrings(cityName, wkgPtr->name) > 0)
        {
          // search the left child
           // function: search
         return search(wkgPtr->rightChildPtr, cityName);

        }
        else
         {
           // otherwise search the right child
            // function: search
             search(wkgPtr->leftChildPtr, cityName);
         }
     }
     // return null
     return NULL;
 }

/*
Name: setCityNodeData
Process: copies data from source node to destination pointer
Function input/parameters: node to be copied (CityDataType)
Function output/parameters: destination pointer (CityDataType *)
Function output/returned: none
Device input/file: none
Device output/monitor: none
Dependencies: strcpy
*/
void setCityNodeData( CityDataType *destPtr, const CityDataType source )
 {
   // set the name
    // function: strcpy
    strcpy(destPtr->name, source.name);

   // set the population
    destPtr->population = source.population;
 }

/*
Name: cityDataToString
Process: sets data from node to formatted string
Function input/parameters: node with data to be set (CityDataType)
Function output/parameters: string array with result (char [])
Function output/returned: none
Device input/file: none
Device output/monitor: none
Dependencies: sprintf
*/
void cityDataToString( char str[], CityDataType cityData )
 {
   // print the result
    // function: sprintf
   sprintf(str, "City name: %s, Population: %d\n", cityData.name,
                                                          cityData.population);
 }
```