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

// local constant definitions
const char SEMI_COLON = ';';
void addElementWithData( StudentArrayType *adtArray, const char *name,
                                    char gender, int studentId, double gpa )
    {
      // check for resized
       // function: checkForResize
        checkForResize(adtArray);
      // add the name
        // function: strcpy
        strcpy(adtArray->array[adtArray->size].name, name);
      // add the gender
        adtArray->array[adtArray->size].gender = gender;
      // add the student id
        adtArray->array[adtArray->size].studentId = studentId;
      // add the gpa
        adtArray->array[adtArray->size].gpa = gpa;
      // increment the size
        adtArray->size = adtArray->size + 1;
    }

void addElementFromElement( StudentArrayType *adtArray,
                                                const StudentDataType student )
    {
      // call the addElementWithData function
        // function: addElementWithData
        addElementWithData(adtArray, student.name, student.gender, \
                                            student.studentId, student.gpa );
    }

void checkForResize( StudentArrayType *adtArray )
    {
       // initialize variables
       int  index;
       // create a new array
       StudentDataType *newArray;
       // test if size is at capacity
       if(adtArray->size == adtArray->capacity)
        {
          // double the capacity
           adtArray->capacity = (adtArray->capacity *PRECISION);
          // dynamically allocate memory for the new array
           // function: malloc
          newArray = (StudentDataType*)malloc(sizeof(StudentDataType) \
                                                  *adtArray->capacity);
          // loop through the array
          for(index = 0; index < adtArray->size; index++)
           {
             // copy new size to array
              // function: deepCopy
             deepCopy(&newArray[index], adtArray->array[index]);
           }
          // free up space for array
           // function: free
          free(adtArray->array);
          // set the new to the old array
          adtArray->array = newArray;
      }
    }

StudentArrayType *clearStudentArrayType( StudentArrayType *arrayPtr )
    {
       // clear the array
        // function: free
       free(arrayPtr->array);
       // free the pointer
        // function: free
       free(arrayPtr);
       // return the pointer
       return arrayPtr;
    }

void copyArrayType( StudentArrayType *dest, const StudentArrayType src )
    {
      // initialize variables
      int element;
      // test if the capacity of dest is less than src
      for(element = 0; element < src.size; element++)
       {
         // check for resizing
         checkForResize(dest);
         // update the size
         dest->array[element] = src.array[element];
```

```
        // increment size
        dest->size = dest->size + 1;
      }
    }

void deepCopy( StudentDataType *dest, const StudentDataType src )
    {
      // assign src elements to dest elements
        // copy name from src to dest
          // function: strcpy
          strcpy(dest->name, src.name);
       // copy gender from src to dest
          dest->gender = src.gender;
       // copy student id from src to dest
          dest->studentId = src.studentId;
       // copy gpa from src to dest
          dest->gpa = src.gpa;
    }

void displayData( const StudentArrayType adtArray, const char *subTitle )
    {
      // initialize variables
      int newIndex;
      StudentDataType data;
      // check if the array is empty
      if(adtArray.size != NULL_CHAR)
       {
         // print the title
          // function: printf
         printf("\nDisplay Title - %s\n", subTitle);

         // loop through the array
         for(newIndex = 0; newIndex < adtArray.size; newIndex++)
          {
            // shorthand for array[newIndex]
            data = adtArray.array[newIndex];
            // print the array
              // function: printf
            printf("[ %d ]: %s, %c, %d, %0.5f\n", newIndex, data.name, \
                    data.gender, data.studentId, data.gpa);
          }
       }
       else
       {
         printf("Data not found - Data Aborted\n");
       }

    }

bool findElement( StudentDataType *foundElement,
              const StudentArrayType adtArray, const StudentDataType searchValue )
    {
      // initialize variables
      int size;
      // loop through the array
      for(size = 0; size < adtArray.size; size++)
        {
          // test if element and searchVal are the same
           // function: strcmp
          if(strcmp(adtArray.array[size].name, searchValue.name) == 0)
            {
              // copy to array
                // function: deepCopy
              deepCopy(foundElement, adtArray.array[size]);
              // return true
              return true;
            }
        }
        // set the element to zero
         // function: setEmptyElementData
        setEmptyElementData(foundElement);
        // otherwise return false
        return false;
    }

void getFileName( char *fileName )
    {
      // prompt user for file
        // function: printf
        printf("Enter file name: ");

      // store user input in variable
        // function: scanf
        scanf("%s", fileName);
    }

StudentArrayType *initializeStudentArrayType( int initialCapacity )
```

```
    {
      // create the array pointer
      StudentArrayType *newArray;
       // dynamically allocate space for new pointer
        // function: malloc
      newArray = (StudentArrayType*)malloc(sizeof(StudentDataType));
      // assign the array capacity to the initialCapacity
      newArray->capacity = initialCapacity;
      // initialize array size to zero
      newArray->size = 0;
      // allocate the array itself
        // function: malloc
      newArray->array = (StudentDataType*)malloc(sizeof(StudentDataType) \
                                            * newArray->capacity);
      // return array pointer
      return newArray;

    }

bool removeElement( StudentDataType *removedElement,
                    StudentArrayType *adtArray, const StudentDataType searchVal )
    {
      // initialize variables
      int index, updatedIndex;
      // loop through the array
      for(index = 0; index < adtArray->size; index++)
       {
         // tests if element is the search value
           // function: strcmp
         if(strcmp(adtArray->array[index].name, searchVal.name) == 0)
           {
             // if found, store the element to removeElement
               // function: deepCopy
             deepCopy(removedElement, adtArray->array[index]);
             // shift all data down by decrementing the size
             adtArray->size = adtArray->size - 1;
             // loop through the updated array
             for(updatedIndex = index; updatedIndex < adtArray->size; \
                                                          updatedIndex++)
               {
                 // copy updated array to the dest array
                 // function: deepCopy
                 deepCopy(&adtArray->array[updatedIndex], \
                                      adtArray->array[updatedIndex + 1]);
               }
             // return true
             return true;
           }
        // end loop
       }
       // set the removed element to zero
         // function: setEmptyElementData
       setEmptyElementData(removedElement);
       // return false
       return false;
     }


void runMerge( StudentArrayType *adtArray,
                                  int lowIndex, int middleIndex, int highIndex )
    {
      // initialize variables
      int leftSide, leftSet, rightSide, rightSet;
      int size = highIndex - lowIndex + 1;
      int indexSrc, indexMerging;
      // initialize pointer
      StudentDataType *tempArray;
      // allocate space for a temp arrays
        // function: malloc
      tempArray = (StudentDataType*)malloc(sizeof(StudentDataType)*size);
      // load the data into the array
      indexSrc = lowIndex;
      // loop through array
      for(indexMerging = 0; indexMerging < size; indexMerging++)
        {
          // copy into temp from original array
           // function: deepCopy
          deepCopy(&tempArray[indexMerging], adtArray->array[indexSrc]);
          // increment the original array index
          indexSrc++;
        }
      // define the indices
        // left index starts at zero
      leftSide = 0;
       // the left side limit starts at the middle of the left side
      leftSet = middleIndex - lowIndex;
       // right index starts at left side of the right section
```

```
    rightSide = leftSet + 1;
     // right side limit start in the middle of the right side
    rightSet = highIndex - lowIndex;
     // first subset starts at the first index
    indexSrc = lowIndex;
    // loop until left side and right side run out of elements
    while(leftSide <= leftSet && rightSide <= rightSet)
     {
       // if the left element is bigger than the right element
         // function: strcmp
       if(strcmp(tempArray[leftSide].name, tempArray[rightSide].name) < 0)
        {
          // copy elements into the original array
           // function: deepCopy
          deepCopy(&adtArray->array[indexSrc], tempArray[leftSide]);
          // increment the left index
          leftSide++;
        }
       else
        {
          // otherwise copy temp into the original array
            // function: deepCopy
          deepCopy(&adtArray->array[indexSrc], tempArray[rightSide]);
            // increment the right index
          rightSide++;

        }
        // increment the original array's index
       indexSrc++;
     }
     // loop through the left side of array
     while(leftSide <= leftSet)
       {
         // copy left side into new array from original
         adtArray->array[indexSrc] = tempArray[leftSide];
         // increment the left side index
         leftSide++;
         // increment the original array index
         indexSrc++;
       }
      // loop through the right side of array
     while(rightSide<= rightSet)
       {
         // copy right side into the temp array from original
         adtArray->array[indexSrc] = tempArray[rightSide];
         // increment the right side index
         rightSide++;
         // increment the original array index
         indexSrc++;
       }
       // free up the temp array memory
        // function: free
    free(tempArray);
  }

void runMergeSort( StudentArrayType *adtArray )
  {
    // initialize variables
     int smallIndex, bigIndex;
    // set the bounds for the merging
     smallIndex = 0;
     bigIndex = adtArray->size - 1;
    // merge sort the array
     // function: runMergeSortHelper
     runMergeSortHelper(adtArray, smallIndex, bigIndex);
  }

void runMergeSortHelper( StudentArrayType *adtArray,
                                                  int lowIndex, int highIndex )
  {
     // initialize variables
     int middleIndex;

    // merge until there's no more elements
    if(lowIndex < highIndex)
      {
        // calculate the middle element
        middleIndex = (highIndex + lowIndex) / PRECISION;
        // merge the left side of the array
         // function: runMergeSortHelper
        runMergeSortHelper(adtArray, lowIndex, middleIndex);
       // merge the right side of the array
        // function: runMergeSortHelper
        runMergeSortHelper(adtArray, middleIndex + 1, highIndex);
       // merge the two subsets together
        // function: runMerge
        runMerge(adtArray, lowIndex, middleIndex, highIndex);
```

```
        }
    }

int runPartition( StudentArrayType *adtArray, int lowIndex, int highIndex )
    {
      // initialize variables
      int swapping, moving;
      int swapped;
      // point to first value
      swapping = lowIndex;
      // set the moving value
      swapped = lowIndex;
      // loop through the array
      for(moving = lowIndex; moving <= highIndex; moving++)
       {
         // check if the moving index is less than the present value
           // function: strcmp
         if(strcmp(adtArray->array[moving].name, \
                                  adtArray->array[swapped].name ) < 0)
           {
             // incrment the array index
             swapping++;
             // swap the present value with the moving value if condition true
               // function: swapElements
             swapElements(adtArray, swapping, moving);

           }
       }
       // otherwise swap array value with the pointing value
        // function: swapElements
       swapElements(adtArray, swapped, swapping);
       // return the pointer index
       return swapping;
    }

void runQuickSort( StudentArrayType *adtArray )
    {
      // initialize variables
      int smallIndex, bigIndex;
      // create the bounds of the array
      smallIndex = 0;
      bigIndex = adtArray->size - 1;
      // quick sort the array
       // function: runQuickSortHelper
      runQuickSortHelper(adtArray, smallIndex, bigIndex);
    }

void runQuickSortHelper( StudentArrayType *adtArray,
                                                    int lowIndex, int highIndex )
    {
      // initialize variables
      int partition;
      // test to see if lowIndex is less than highIndex
      if(lowIndex < highIndex)
       {
         // partition the array
          // function: runPartition
         partition = runPartition(adtArray, lowIndex, highIndex);
         // sort one subset
          // function: runQuickSortHelper
         runQuickSortHelper(adtArray, lowIndex, partition - 1);
         // sort the second subset
          // function: runQuickSortHelper
         runQuickSortHelper(adtArray, partition + 1, highIndex);
       }
    }

void setEmptyElementData( StudentDataType *element )
    {
       // set the name to null
       *element->name = NULL_CHAR;
       // set gender to X
       element->gender = 'X';
       // set studentId to 0
       element->studentId = 0;
       // set gpa to 0
       element->gpa = 0;
    }

void showTitle()
    {
       // print title
        // function: printf
        printf("Data Management with Log2N Sorting\n");
        printf("==================================\n");
    }
```

```
void swapElements( StudentArrayType *adtArray, int leftIndex, int rightIndex )
    {
      // initialize variables
       StudentDataType temp;
       // left element is stored in a temp
        // function: deepCopy
       deepCopy(&temp, adtArray->array[leftIndex]);
       // left element is swapped with right element
         // function: deepCopy
       deepCopy(&adtArray->array[leftIndex], adtArray->array[rightIndex]);
       // right element is stored in temp
        // function: deepCopy
        deepCopy(&adtArray->array[rightIndex], temp);
    }

bool uploadStudentData( StudentArrayType *adtArray, const char *fileName )
    {
      // initialize variables
      char inputName[ STD_STR_LEN ];
      int inputStudentId;
      char inputGender;
      double inputGpa;
     // open the input file
       // function: openInputFile
      if( openInputFile( fileName ) )
        {
          // read prime
           // function: readStringToDelimiterFromFile
          readStringToDelimiterFromFile( SEMI_COLON, inputName );
          // read the whole file
           // function: checkForEndOfInputFile
          while( !checkForEndOfInputFile())
           {
             // read in the studentId
              // function: readIntegerFromFile
             inputStudentId = readIntegerFromFile();
             // read in comma
               // function: readCharacterFromFile
             readCharacterFromFile();
             // read in the gender
              // function: readCharacterFromFile
             inputGender = readCharacterFromFile();
             // read in comma
               // function: readCharacterFromFile
             readCharacterFromFile();
             // read in the gpa
               // function: readDoubleFromFile
             inputGpa = readDoubleFromFile();
             // add all elements to the variables in the stuct
              // function: addElementWithData
             addElementWithData(adtArray, inputName, inputGender, \
                                                 inputStudentId, inputGpa);
              // read re-prime
               // function: readStringToDelimiterFromFile
             readStringToDelimiterFromFile( SEMI_COLON, inputName );
           }
          // close the file
           // function: closeInputFile
        closeInputFile();
        // return true
        return true;
       }
       // return false
       return false;
    }
```