CMSC 341 Data Structures — All Sections — Spring 2018

Project 1: Sorted Square List with Circular Buffers

Due: Friday, February 23, 8:59:59pm

Links: [Project Submission] [Late Submissions] [Project Grading] [Grading Guidelines] [Academic Conduct]

Change Log

Modified items are in highlighted in blue and orange colors.

[Saturday Feb 24, 07:30pm] The submission instructions have been fixed to reflect the complete set of test programs released, and include modified instructions for compiling them.

[Monday Feb 19, 02:40pm] The descriptions of the SortedSquareList functions `find()` and `findAt()` incorrectly declared them as returning void; they have been corrected to "int". (The declarations in SortedSquareList.h were correct.)

[Sunday Feb 18, 02:50pm] The first set of test programs have been released–see the section "Test Programs".

[Friday Feb 16, 05:35pm] A small modification to the last part of the "removing an entry" algorithm: we added an IF-condition to the final ELSE (the "shift in from the right side by default" action). Also, those instructions were reformatted to look more like pseudocode to help you follow it, and a typo was fixed in the next paragraph. (Changes are blue-highlighted; NOTE: your browser probably also displays all links as blue; none of those were changed.)

[Wednesday Feb 14, 05:12pm] Several small hints and clarifications added. (These changes are orange-highlighted.)

Objectives

The objective of this programming assignment is to have you review C++ programming using following the features: object-oriented design, dynamic memory allocation (of arrays) and pointer manipulation.

Introduction

Note: Circular Buffers are described in Section 5.2.4 of our textbook (p. 211). Please read that section before proceeding.

An interesting, relatively straightforward data structure is the *square list*: it is meant to impose the semantics of a linear list of values with ordering, but is internally implemented as a "square" structure, i.e. a conceptual two-dimensional array, except more dynamic. The classic version does not use arrays at all, instead building a linked list of linked lists. It starts with a top-level linked list, each node of which points to a secondary linked list that represents an ordered sequence of values, one stored per node. The idea is that there is enough information in the nodes of the top-level linked list to allow us to bypass entire sublists while searching for the value that we want, finally only searching the values of one sublist instead of the entire list. In order to guarantee efficiency, we must keep the lengths of all of the sublists relatively equal. The definition of "relatively equal" varies among algorithms, but in all cases is designed to put constraints on the variance so that we compute firm bounds on the runtime for all of the operations on the list.

A good description of one implementation of a square list using linked lists is given in the Fall 2013 CMSC 341 Project 1 description here. You should review that project description. That version demonstrates how the square list allows someone to fetch, modify, remove or add a value at a specific position in the list, specified as a numerical index, all in $O(\sqrt{n})$ time. An array would take a longer O(n) time for the add and remove operations, and a simple linear linked list would take O(n) all of these four.

The linked list-based square list has certain desirable performance characteristics which derive from fundamental properties of linked lists in general: very fast insertion and deletion, and small, independent units of allocation/deallocation. Linked lists also support a concept of ordering: you can deterministically reference the n-th object in the list. The one flaw of a linked list is that it does not efficiently support direct indexing like an array does: you must walk the list link-by-link to get to that n-th object, which is expensive. The linked list square list is just using a trick to decrease the lengths of the lists it must walk.

Our version of the square list for this project, while having similar functional goals, will be a completely different implementation. Specifically, we want to support additional functionality, as well as better performance goals for some of the operations.

Your goal for this project will be to implement a square list with the following requirements:l

- retrieve entries by value, as well as by sorted index, in *O(log n)* time
- Add or remove items in $O(\sqrt{n})$ time.
- Grow dynamically as the number of entries increases, with the growth operation taking O(1) amortized time. (your professor will explain this briefly).

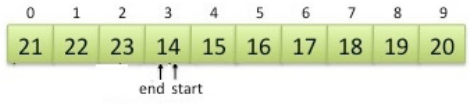
  In order to be able to retrieve any item–by value *or* by sorted order index– in O(log n) time, we need two things. First, being able to retrieve by sorted order index (i.e.,"find the n-th smallest value") requires the list to be kept in sorted order. Even then, both value- and index-based retrieval still take $O(\sqrt{n})$ time if we use a linked list-based square list, but our goal to achieve O(log n) time. To achieve that, We want to be able to use binary search, but that doesn't work on linked lists. It *does* on an array. So we will implement our square list as an array-of-arrays.

  Now, if an array works so well, why do most people use the "linked list of linked lists" approach? The array implementation comes with its own issues: First, growing or shrinking an array is expensive. Second, insertion and deletion require shifting lots of elements around. Insertion or deletion at the head of a list is the most expensive of all. As we shall see later, our implementation of a square list will need to do exactly this operation (insertion/deletion at the ends) frequently. How can we address this?

  This is a good time to introduce a third primitive data structure: the circular buffer. A circular buffer, typically used to implement a queue (review *queues* if you don't remember), is a tweak on arrays that solves the problem of efficient insertion and deletion for a specific common case: where you are constantly adding to or removing from the ends of a list. The insight is that you don't actually have to shift the other elements when you remove one from the head of the list: you can instead adjust the sense of where the list begins and ends, by just incrementing a *head* index which indicates the position of the current list head. Likewise, you maintain a *tail* index: to add an element to the tail, you just insert where *tail* indicates, and then increment tail (NB: the tail index doesn't point to the last stored item, but rather the index right after). This gives us a "sliding buffer"; to make it circular, we just "wrap the pointers around". When either the head or tail index reaches the end of the array, we reset it back to 0. So, at some periods in the lifetime of a circular buffer, the first part of the queue entries might be stored at the very end of the array, with the rest of the entries continuing at the beginning of the array.

  In the example below, 14 is at the start of the queue and 21 is at the end. If we added another item to the FIFO queue, it would go in index #1. If we remove an item from the FIFO queue, 14 would be removed. [diagrams Courtesy Prof. Chang]

  If the tail catches up to the head (with wraparound), that means your list is full, as in the following diagram:



If, on the other hand, the head index catches up to the tail, your list is empty. Unfortunately, a completely full circular buffer will appear just like a completely empty one based on just examining the head and tail indices: in both cases, `head == tail`. That is why most implementations (including ours) also use a somewhat redundant `size` field.

When circular buffers are used to implement queues, the adding/deleting always proceeds from left to right: adding increments the tail index (with wraparound), and deleting increments the head index (again, with wraparound). However, there is nothing that says you cannot add to the "left" end of the buffer ("left" meaning the elements at the head of the queue, not the left end of the array). Adding/deleting from either end of a circular buffer is symmetrial, and either can be done, even intermixed, in constant time: O(1). That is why we are bothering with all this circular buffer mess: to get this very fast add-or-delete-at-the-ends behavior.

So this is our plan: to implement our square list as an array-of-arrays, as planned. However, instead of using simple arrays, we will construct it as a circular buffer of circular buffers: that way, we get the best of both worlds: with a little extra bit of work, we can still do direct indexing and use binary search on both the top-level list and the column sublists just as with simple arrays, but the circular buffer extension will give us very fast shifting performance.

What about the other performance goals? That carries over from the properties of a good square list: as long as we can keep it relatively square, it only takes $O(\sqrt{n})$ time for many operations. So, we have to be clever in how we restructure our square list throughout the various operations in order to maintain certain structural properties (mainly, its "squareness") that give us provably good performance measures. Most square list implementations use an occasional rebalancing step to take a badly-shaped square (one dimension being very different from the other, or columns varying too much in lengths), and rebalance it. We, however, will use a different approach: our list operations will by their design *keep* the square balanced. The strategy for that is described in the next section.

How to Implement

You will implement a class called `SortedSquareList` that supports adding new values, retrieving data by value or index, and removing values, as per the interface are provided later. It will do this by allocating an array to hold a list of references to sub-arrays, the "columns" of the square list. This top-level--or backbone--array must be dynamically allocated, to allow us to expand it as needed. This array must be a true array, to allow us to do efficient binary search. It must be implemented as a circular buffer, to make insertion and deletion of columns efficient. This array must be declared to hold pointers to SSLColumn objects, describe next.

The SortedSquareList class will also have other data and function members as necessary to achieve the functionality and performance specified in the rest of this document. Note that you must support doing binary searches for both lookup by value *and* by index in O(log n) time. It might be useful to add other data members to facilitate efficient operation, and you are free to do so.

The default constructor should create an initial `2` x `2` square list. Another constructor should allow the initial square dimension to be specified. (This dimension length should be stored in `m_capacity`.)

Next, each column must itself be an object of class `SSLColumn` . The SSLColumn class has as one of its members a pointer to a dynamically allocated array of the actual values the user is storing. The SSLColumn class must contain, or provide a means for computing, enough information to allow you to determine whether that column does, or should, contain the value you are looking for.

The SSLColumn class must implement binary search in looking for values. When retrieving by index, though, at this level, you can just simply index directly, since you *are* using a real array.

The default constructor should create an initial `2`-element column, to match the default capacity of the SortedSquareList class. Another constructor should allow the initial square dimension to be specified. (This dimension length should be stored in `m_capacity`.)

All the dynamically allocated arrays in the program must be just that: real arrays. You cannot use the *vector* class, nor any other class from the STL, nor any outside source (found online, borrowed from a friend, etc.), on this project. You are allowed to use the variou stream classes and objects (e.g., cin and cout), math library, string library, and possibly other such low-level packages. If in doubt, please ask first.

Keeping the Square List Shapely

The following is a detailed description of the strategy for keeping the square list square. It is important to note again that the "square list" we are holding in our data structure is probably of a smaller dimension that the actual collection of arrays we are storing it in. Our actual arrays might be of dimension 8: in other words, our top-level SortedSquareList object would have dynamically allocated an array with room for 8 columns, i.e., be sized to hold up to 8 pointers to SSLColumn objects, and each SSLColum object would then have dynamically allocated an array of 8 ints for holding the actual values. So, our total capacity would be 64 values. However, if at some particular time this particular square list were holding 24 values, it would be using a 5 x 5 subset of the 64 cells, and the balancing algorithm would (a) ensure that the data would be stored in 5 contiguous columns (the meaning of "contiguous" here taking wraparound into account), with each of these 5 columns storing 4 or 5 values in contiguous elements (again, modulo wraparound for it being a circular buffer), likely located at a different offset in each column.

The square list is always "square", in the sense that it uses the minimum number of columns, as well as the minimum number of entries per column, additionally keeping these two dimensions as close as possible. So, for holding 17-25 entries, it should store the data in 5 columns of either 4 or 5 values. No column would contain more than 5 entries, and none, with the possible exception of the last (i.e., rightmost) column, would be shorter than 4 entries. Once again, it bears mentioning that the actual size of the underlying backbone array and column arrays is likely much larger than this (see "Allocation Strategy" below), but we will only be using a fraction of each array. The algorithm described here ensures you will always be using a square subset of the actual space.

The algorithm for maintaining the properties of the square list depend on a number we will call its "square dimension" (hereafter referred to by the acronym SD): this is the square root of the current number of entries, rounded up to the next larger integer. We will always enforce the property that both the number of columns and column length will be equal to or at most 1 less than, the square dimension. For example, if we were holding between 17 and 25 values, SD == 5, and we would have 4 or 5 columns of 4 or 5 values. If we subsequently grew into the 26-36 range, SD == 6; 37-49 range: SD == 7; and so on.

When adding an entry into a column, we would first insert the new value in the appropriate column and position. If the value being added is between columns, it should be appended to the end of the earlier column, instead of being inserted at the head of the following column.) We must then check to see if the resulting length of the column (number of active entries, not the actual array capacity) exceeds the SD of the new total number of values. This is the upper limit we are enforcing. If our column exceeds this limit, we resolve the violation by shifting one of the entries to an adjacent column, using the following strategy:

1. If the insertion column is the left- or rightmost column, shift towards the other end;
   ELSE: compute the mean length of the set of columns to either side, and shift towards the side with the lower mean.

2. Having chosen the direction, "shifting" consists of taking the entry at the appropriate end of the current column, and moving it to the appropriate position in the adjacent column. So, if we are shifting to the right, we would take the *last* entry in the column, and insert it as the first entry in the next column (the one to the right). Alternatively, shifting to the left would mean removing the *first* item from the current column and adding it as the last entry in the previous column (the one to its left).

3. This shift might have caused the receiving column to in turn exceed *its* SD limit, in which case we would execute a shift again, possibly multiple times, in the same direction as the first shift.

4. If we need to shift beyond the leftmost or rightmost active column (the active columns being defined as the entries that are part of the active range of the top-level circular buffer), we just extend the active range by decrementing m_start or incrementing m_end, of course watching out for wraparound. Just as when we extend a circular buffer of ints, the space for the int was actually there all along in the underlying array, when we extend the set of active columns, the SSLColumn that we add has actually been there all along; it was created when the square list was first created, but we were just not actively using it. Removing a column as described in the next section is an analagous

delete-but-not-really-delete operation.

Warning: you should check that the column isn't already at capacity before inserting the new value, and handle it appropriately if it is. This also applies as you shift values: shifting a value into a column might exceed its capacity. Note that the condition is temporary: any column where adding a new value would exceed its capacity would also be exceeding the SD, and you would therefore be immediately shifting a value out the other end; however, the column cannot even temporarily hold extra data. You must anticipate this potential overflow and do the shift out before doing the shift in.

When removing an entry, we do roughly the opposite of the operations for adding a value. After finding and removing the entry from the column, we check to see if the resulting length is less than the lower limit: SD - 1. If it is, we resolve this violation by shifting in one of the entries *from* an adjacent column, using the following strategy:

1. For the first shift:
   - IF (either side has a column that is at its upper limit, i.e., == SD)
     shift in from that side (choosing the right if both sides qualify);
   - ELSE IF (either side has a column below the minimum (SD - 1))
     shift in from that side;
   - ELSE IF (not an end column)
     shift in from the right side by default
   - (implied final ELSE: do nothing; the column just became (or already was) the one short end column)
2. The shift might have caused the donating~~receiving~~ column to in turn drop below the SD - 1 lower limit, in which case we would execute a shift again, possibly multiple times, from the same direction as the first shift.
3. The last column might end up being short: this is okay.
4. If the first or last column drops to length 0, we must remove the column from the outer list. This does not mean the SSLColumn object is actually deleted: it fact, it must be left as-is, because we might need it again shortly. We instead "remove" the column from the top-level circular buffer by updating the `m_last` (or `m_first`, as the case may be) and `m_size` data members to indicate it is no longer in the set of active columns.

   Adding and removing are the only operations that alter the shape of the square list, and the above instructions will restore the desired shape.

Allocation Strategy

Since all our storage will be in the form of arrays (an array of SSLColumn pointers in the SortedSquareList object, and an array of ints in each SSLColumn object), we have to worry about how large to allocate each array. We will start off dimensioning each at 2, allowing us to store up to 4 values. If the user tries to add a 5th value, we will grow it to a 4x4 array-of-arrays, then 8x8, 16x16, 32x32, and so on. The pattern is to double the dimensions each time, which will quadruple the capacity with each expansion. (This is similar to the growth strategy of most implementations of the *vector* class, except being 1-dimensional, they only double the capacity each round.) This exponential strategy gives us what is known as amortized constant time cost for copying the elements from the old to the new.

So when do you trigger the resizing? When adding a new value to the square list, you would first compare the old size to the current capacity, and if it's at the limit, you must grow before adding the new value. Growing consists of allocating a new top-level buffer, allocating all new columns of the newly increased dimension, then copying all the elements over column-by-column. This will result in half of the new columns being half-full, and the other half not being used at all yet. Since everything is a circular buffer, there's no need to try to match the positions: just copy your N old columns into the lower N columns of the new list, and for each column, copy the values into the first N elements of the larger array. You must be careful about making sure you delete all of the old data structures: you are sure to have a memory leak or two at first.

NOTE: we never shrink the square lists, even if enough data is removed to allow us to do so. This makes life much easier for you.

Assignment

Note: Running time is one of the most important considerations in the implementation of a data structure. Programs that produce the desired output but exceed the required running times are considered wrong implementations and will receive substantial deductions during grading.

Your assignment is to implement the variant of the square list data structure described above: a version that is structured as a circular buffer of circular buffers. To repeat a prohibition from above: you must build it using only primitive data structures, plus classes of your own design. You many not use *vectors* or any other classes from the C++ STL collection, nor any classes or data structures acquired from the web or from associates. ***Violating this will likely cause you to receive a failing grade on the project***.

For this project, we are detailing much of the requirements for the overall structure and behavior of your data structure, so you will have limited high-level design choices. There will be plenty of design work to do on the low-level implementation details, though.

You are required to use the class definitions given in `SSLColumn.h` and `SortedSquareList.h`. (Specifications are given below.) The `SSLColumn` class is an implementation of a simple circular buffer of `int` values. The `SortedSquareList` class is the backbone circular buffer that manages the list of SSLColumn objects representing the columns of data.

```
// file: SSLColumn.h
//
// UMBC CMSC 341 Spring 2018 Project 1
//
// Header file for SSLColumn ("sorted square list column")
// This is mostly an implementation of a circular buffer
// See project description for details.
//
// Unless otherwise noted, any position argument is relative to
// the stored data, and not an absolute index into the underlying array.
//

#ifndef _SSLCOLUMN_H_
#define _SSLCOLUMN_H_

class SSLColumn {

public:
   // Constructor, default size is 2
   SSLColumn(int n=2);

   // Copy constructor
   // Makes exact copy, including relative positions in circular buffers
   SSLColumn(const SSLColumn& other);

   // Destructor
   ~SSLColumn();

   // overloaded assignment operator
   // We can copy into differently-sized array.  Copying done in-place if
   // target has sufficient capacity; else, target's internal array
   // will be reallocated to a larger size.
   const SSLColumn& operator=(const SSLColumn& rhs) ;

   // Adds item to correct sorted order position in circular buffer.
   // Throws exception if out of room.
```

```
   // REQUIREMENT: must execute in O(n) time, where n is size of column
   void add(int data);

   // Adds item to top/left of circular buffer.
   // Throws exception if out of room.
   // REQUIREMENT: must execute in O(1) time
   void addFirst(int data) ;

   // Adds item to bottom/right of circular buffer.
   // Throws exception if out of room.
   // REQUIREMENT: must execute in O(1) time
   void addLast(int data) ;

   // returns position of item in column, -1 if not found.
   // REQUIREMENT: must execute in O(log(n)) time, where n is size of column
   int find(int data);

   // Retrieves value at index pos in circular buffer, based on sort order.
   // If pos == -1, retrieves last value in list, a la Python
   // Throws exception if pos >= size.
   // REQUIREMENT: must execute in O(1) time.
   int findAt(int pos);

   // Removes item from column, returning original position of item,
   // -1 if not found.
   // REQUIREMENT: must execute in O(n) time, where n is size of column
   int remove(int data);

   // Removes top/left item from circular buffer and returns it.
   // Throws exception if empty.
   // REQUIREMENT: must execute in O(1) time
   int removeFirst(void) ;

   // Removes bottom/right item from circular buffer and returns it.
   // Throws exception if empty.
   // REQUIREMENT: must execute in O(1) time
   int removeLast(void) ;

   // return maximum number of items this buffer can hold
   int capacity() ;

   // return number of items currently held in the buffer
   int size() ;

   // debugging function. Prints out contents.
   void dump();

   // grading function used to examine private data members.
   // Do not implement!
   bool inspect (int* &buf, int &cap, int &size, int &start, int &end);

private :
   int *m_data;    // pointer to dynamically allocated array for buffer
   int m_capacity;  // length of the allocated space pointed by m_data
   int m_size;      // number of active items in buffer
```

```
   // index of the first active item in the buffer (virtual index 0)
   int m_start;
   int m_end;       // index AFTER last active item in buffer

  // STUDENT-ADDED MEMBERS HERE
  ...
  ...
  // END STUDENT-ADDED MEMBERS

};

#endif
```

```
// file: SortedSquareList.h
//
// UMBC CMSC 341 Spring 2018 Project 1
//
// Header file for Sorted Square List
// See project description for details.
//
// Unless otherwise noted, any position argument is relative to
// the stored data, and not an absolute index into the underlying array.
//

#ifndef _SORTEDSQUARELIST_H_
#define _SORTEDSQUARELIST_H_

#include "SSLColumn.h"

class SortedSquareList {

public:
   // default constructor
   SortedSquareList() ;

   // copy constructor
   SortedSquareList(const SortedSquareList& other) ;

   // destructor
   ~SortedSquareList() ;

   // overloaded assignment operator
   // We can copy into differently-sized square list, as long as it is large
   // enough to hold all of the actual stored data.
   const SortedSquareList& operator=(const SortedSquareList& rhs) ;

   // Adds item to correct position in square list, based on sort order.
   // Must grow dynamically allocated structures if we run out of room;
   // REQUIREMENT: must execute in O(sqrt(n)) time, where n is
   // number of items currently stored.
```

```
    void add(int data) ;

    // returns position of item in list, -1 if not found.
    // REQUIREMENT: must execute in O(log(n)) time
    int find(int data);

    // Retrieves value at index pos in square list, based on total sort order.
    // Throws exception if pos beyond end
    // REQUIREMENT: must execute in O(log(n)) time
    int findAt(int pos);

    // Finds and removes item from square list.
    // Returns true if data found, false otherwise.
    // REQUIREMENT: must execute in O(sqrt(n)) time
    bool remove(int data);

    // return maximum number of active items this square list can hold with
    // current allocation
    int capacity();

    // Total number of items in the square list as a whole, across all columns.
    int size();

    // debugging function, prints out contents of data structure
    void dump();

    // grading function used to examine private data members.
    bool inspect (SSLColumn** &buf, int &cap, int &size, int &start, int &end);

private :
    // Top-level array of pointers to SSLColumns.
    // Each entry of the array is a pointer.
    SSLColumn **m_cols;
    int *m_colInfo;  // Useful for caching some info about each column
    int m_capacity;  // length of the allocated space pointed by m_cols
    int m_size;      // number of active columns in top-level buffer
    // index of first active column in the top-level buffer(virtual index 0);
    int m_start;
    int m_end;      // index AFTER last active column in top-level buffer

  // STUDENT-ADDED MEMBERS HERE
  ...
  ...
  // END STUDENT-ADDED MEMBERS

};

#endif
```

Specifications

Here are the specifics of the assignment, including a description for what each member function must accomplish.

Requirement: Place the implementation of the `SSLColumn` member functions in a file called `SSLColumn.cpp` and the

implementation of the `SortedSquareList` member functions in `SortedSquareList.cpp`.

Requirement: All member functions must be implemented *from scratch*. In particular, you are not allowed to use any classes from the Standard Template Library (STL), not even `vector`.

Requirement: ***You are not allowed to modify any part of the provided header files*** SSLColumn.h and SortedSquareList.h except to add additional helper function and data members in the section between the labels:

```
// STUDENT-ADDED MEMBERS HERE
...
...
// END STUDENT-ADDED MEMBERS
```

We will be doing a strict check, and if there were any additions, deletions, or modifications to any other part of either of these files, you will receive a substantial deduction!

Requirement: Do not use any global variables!

Requirement: Your code must compile on GL with the **exact** Unix commands given below. (See "How to Submit".)

Requirement: Your code must not have any memory leaks. When you run your code under `valgrind` on GL, it must report:

```
All heap blocks were freed -- no leaks are possible
```

This means you must be thorough with your destructor implementations.

Requirement: Your implementation must be *efficient*. We will provide tests later that will validate the performance of your code meets the requirements set out earlier in this specification.

Requirement: Many of the required functions described below throw exceptions. All of the exceptions we require are defined in `stdexcept` so all you need to do is

```
#include <stdexcept>
```

at the top of your source.
You should not throw any other exception types outside your code; if you define your own exception classes, you must catch them and handle them completely inside your own functions.

These are the member functions of the `SSLColumn` class you ***must*** provide, with the exact signatures:

- ```
  SSLColumn(int n=2) ;
  ```

  This is the constructor for the `SSLColumn` class. It should initialize the data members and allocate memory to hold at least `n` `int` values. The member `m_data` should point to this memory. The default value of `n` is `2`.

- ```
  SSLColumn(const SSLColumn& other) ;
  ```

  This is the copy constructor for the `SSLColumn` class. The copy constructor must create a complete, exact copy of the `SSLColumn` object `other`. The new copy must have separately allocated memory for the circular buffer of `int`. By "exact copy", we mean that the circular buffer properties must be preserved: each and every circular buffer must duplicate data arrays element-for-element, and the start and end indices must be preserved. Do not use the assignment operator to implement the copy constructor. Just don't. No really, it's a bad idea.

- ```
  ~SSLColumn() ;
  ```

  This is the destructor for the `SSLColumn` class. You must deallocate memory. You should (almost) never call a destructor explicitly.

- ```
  const SSLColumn& operator=(const SSLColumn& rhs) ;
  ```

  This is the overloaded assignment operator. If you forgot what that means, crack open your C++ textbook again. Remember to check for self-assignment. The `SSLColumn` objects on the left hand side (LHS) and right hand side of the assignment (RHS) are not required to have the same capacity. Typically, you would gut the LHS object (deallocate and reallocate dynamic objects, etc.) and recreate it to make it an exact duplicate of the RHS. We will not do that here; ours is spiritually more like assigning across mixed numerical types. If your LHS SSHColumn has capacity equal to or greater than the RHS's active size, you should just transfer the active data. (This will allow us to resize our structures by creating a new empty larger one, then just doing an assignment to copy the data.) You do not have to preserve the positioning within the circular buffer, as you did with the copy constructor: you can just copy it starting at index `0` in the target array.

  If the LHS does not have to capacity to hold the active data of the RHS, you should then reallocate the dynamic data in the LHS to match the capacity of the RHS. Remember to deallocate space of the LHS object first.

  You can't use the copy constructor to implement the assignment operator. That would create a third object, which isn't what you want. Really, that doesn't work. If you don't "remember" why you have to check for self-assignment, then go read up on assignment operators. It will actually save you time.

- ```
  void add(int data);
  ```

  This function should add the value in the `data` parameter to the column's circular buffer in the correct sorted position. Duplicates are allowed (meaning you don't have to check for them). This will require shifting items after the insertion point down one position. Remember to be aware of the wrap around. If the buffer is full, then throw an `overflow_error` exception. This exception is defined in `stdexcept`.
  This function must execute in O(n) time, where n is the number of items in the column.

- ```
  void addFirst(int data);
  ```

  This function should add the value in the `data` parameter as the new first item in the column. The caller is responsible for ensuring the sort order is not violated. Remember to be aware of the wrap around. If the buffer is full, then throw an `overflow_error` exception. This exception is defined in `stdexcept`.
  This function must execute in O(1) time.

- ```
  void addLast(int data);
  ```

  This function should add the value in the `data` parameter as the new last item in the column. The caller is responsible for ensuring the sort order is not violated. Remember to be aware of the wrap around. If the buffer is full, then throw an `overflow_error` exception. This exception is defined in `stdexcept`.
  ```

This function must execute in O(1) time.

- ```
  int find(int data);
  ```

This function retrieves the index in the column of the value in the `data` parameter. The index is relative to the active part of the circular buffer, and is ***not*** the index in the underlying array. If the value was not found, return -1.
This function must execute in O(log(n)) time, where n is the number of items in the column.

- ```
  int findAt(int pos);
  ```

This function retrieves the value at index `pos` in the column, retrieving the last item if pos == -1. The index is relative to the active part of the circular buffer, and is ***not*** the index in the underlying array. Other than the special "-1" case, if `pos` is outside the range of legal data positions, throw a `range_error` exception. This exception is defined in `stdexcept`.
This function must execute in O(1) time.

- ```
  int remove(int data);
  ```

This function should remove the item in the column with the value specified by the `data` parameter. It should return the original position in the column where the item was found and deleted, or -1 if not found.
This function must execute in O(n) time, where n is the number of items in the column.

- ```
  int removeFirst(void);
  ```

This function should remove first item in the column and return it. Remember to be aware of the wrap around. If the buffer is empty, then throw an `underflow_error` exception. This exception is defined in `stdexcept`.
This function must execute in O(1) time.

- ```
  int removeLast(void);
  ```

This function should remove last item in the column and return it. Remember to be aware of the wrap around. If the buffer is empty, then throw an `underflow_error` exception. This exception is defined in `stdexcept`.
This function must execute in O(1) time.

- ```
  int capacity();
  ```

This function should return the number of `int` values that can be stored in the amount of space allocated in the array of `int` that `m_data` points to. I.e., it's the length of the array of the circular buffer.

- ```
  int size();
  ```

  Returns the number of items stored in the circular buffer.

- ```
  void dump();
  ```

  This is a debugging function that prints out the contents of the `SSLColumn` object. See sample outputs below for suggested format. (You should follow the formatting as closely as possible, but don't worry about the exact spacing.)

- ```
  bool inspect (int* &buf, int &cap, int &size, int &start, int &end);
  ```

  This function is used for grading. You do not need to implement anything for this function. In fact, it is important that you don't provide any code for this function, since the grading programs have their own implementations and your code won't compile if the compiler sees two implementations.

These are the member functions of the `SortedSquareList` class:

- ```
  SortedSquareList();
  ```

  This is the default constructor for the `SortedSquareList` class. It should initialize all of the data members and allocate space for a 2x2 (2 columns, each with room for 2 values) circular-buffer-of-circular-buffers. Note that `m_cols` is an array of pointers to `SSLColumn` instances. You need to initialize this array at construction time to already point to a full complement of constructed SSLColumn objects, each of the same size.

- ```
  SortedSquareList(const SortedSquareList& other);
  ```

  This is the copy constructor for the `SortedSquareList` class. You have to make a complete, exact copy of the `other` object, down to the level of having the circular buffers identical in array content, start, and end points. Note that when we say "identical in array content", we are still requiring a deep copy--you should not just do a shallow copy of the m_cols array. Make good use of the `SSLColumn` copy constructor. The admonition to not use the `SortedSquareList` assignment operator to implement this copy constructor also holds here.

- ```
  ~SortedSquareList();
  ```

  This is the destructor. You must deallocate space used by the `SSLColumn` objects that the array of pointers in `m_buffers` points to. Remember that you don't do that by calling the `SSLColumn` destructor explicitly, because you never call destructors explicitly.

- ```
  const SortedSquareList& operator=(const SortedSquareList& rhs);
  ```
  ```

This is the overloaded assignment operator for the `SortedSquareList` class. Read the reminders about assignment operators written in the specifications of the assignment operator for the `SSLColumn` class. They also apply here. So do the comments about what to do with size mismatches: in a nutshell, if the LHS is larger than the RHS, just transfer the data, but if the LHS is smaller (specifically, LHS's capacity < RHS's size), you must reallocate everything in the LHS to match the size of the RHS, and then copy.

- ```
  void add(int data);
  ```

This member function adds the value in the `data` parameter to the square list, in sorted position. The code for this function (and for remove()) are responsible for keeping the square list "square": a detailed description of the exact process is given above. Unless you are at the full capacity of this data structure, this function should not require allocating any actual SSLColumn objects: all of those should have been constructed when this SortedSquareList object itself was created. However, the rebalancing process might cause this function to shift data into an existing-but-unused SSLColumn by extending the head or tail of the top-level circular queue of SSLColumn objects managed by this class. In other words, you do not allocate the SSLColumn, you just start using one that was dormant (i.e., not previously in the active part of the circular buffer).

Before actually inserting the new value, this function must first test to see if the existing data structures are at capacity, and if so, it must grow the square list, as described earlier. The rebalancing process is designed to never try to grow beyond the actually allocated number of columns until every column is completely full, and conversely, it will never ask to grow a column beyond its maximum size until every column has been filled, so a simple comparison of size to capacity for the square list is all that is required.

Resizing will require allocating a new larger top-level circular buffer, creating all-new SSLColumn objects (more, and larger) to fill it, then copying the old SSLColumns' data into some of the new SSLColumns (the rest of the new columns are again for future expansion), then deleting all of the old SSLColumns, and finally deleting the old top-level circular buffer from this square list.

- ```
  int find(int data);
  ```

This function retrieves the index in the list of the value in the `data` parameter. The index is relative to the sorted order of all the data stored in the entire square list, and does not have any relationship to to where it is actually stored. If the value was not found, return -1.
This function must execute in O(log(n)) time.

- ```
  int findAt(int pos);
  ```

This function retrieves the value at index `pos` in the list, in sorted order, retrieving the last item if pos == -1. Other than the special "-1" case, if `pos` is outside the range of legal data positions, throw a `range_error` exception. This exception is defined in `stdexcept`.
This function must execute in O(log(n)) time.
  ```

- ```
  bool remove(int data);
  ```

  This function should remove the item in the list with the value specified by the `data` parameter. It should return true if the item was successfully found and deleted, or false if not found.
  This function must execute in O(sqrt(n)) time.

- ```
  int size();
  ```

  Returns the number of values currently being stored in the data structure.

- ```
  void dump();
  ```

  As before, this is a debugging function that prints out the contents of the entire data structure. Make good use of `SSLColumn::dump()`. See sample output below for suggested format.

- ```
  bool inspect (SSLColumn** &buf, int &cap, int &size, int &start, int &end);
  ```

  As with `SSLColumn::inspect()`, this function is used for grading. Just don't do anything with this function, you will be fine.

Test Programs

The following test programs are provided to jump-start your imagination on how you should test your programs. They are illustrative, but in no way exhaustive. In the future, we will provide less hand-holding for the testing phase of your projects; you should be able to imagine where the potentially buggy edge cases are, and write your own test programs to exercise that code. This is a big part of what you will be doing in the real world.

Three very important things about the set of programs we are providing:

A. These are ***not*** the programs we will be using to grade your projects;

B. The test programs do not test all possible scenarios; it will not be in any way an acceptable excuse for broken functionality in grading, to say: "...but it worked for all the test programs you gave us!"

C. Already for this first project, and even more so for all following projects, you might not get the exact same results as what we show in our sample output. If you do get the same results, that's good. However, if you get slightly different results, it is your responsibility to determine whether your program really needs fixing, or it is just a reasonable variation in interpretation.

Note that if your program does not successfully compile without errors or warnings, or if it dies with a segfault or other such unexpected error, it is highly likely to not pass our grading tests, either. To put it in math terms, passing our tests is a necessary, but not sufficient, condition.

- `p1test01.cpp` and `p1test01.txt` (sample output).
- `p1test02.cpp` and `p1test02.txt` (sample output).
- `p1test03.cpp` and `p1test03.txt` (sample output).
- `p1test06.cpp` and `p1test06.txt` (sample output).

- p1test07.cpp and p1test07.txt (sample output).
- p1test08.cpp and p1test08.txt (sample output).
- p1testXX.cpp and p1testXX.txt (sample output).

These files are also available on GL in the directory:

```
/afs/umbc.edu/users/p/a/park/pub/cs341/00Proj1/
```

Implementation Notes

- Be careful about how you implement find-by-index (as opposed to by-value). Even if you can determine in O(log(n)) time the position within the particular column the value is stored in, if it takes you O(sqrt(n)) to determine where that column starts in the overall ordering of all data (for example, by adding up the sizes of the columns before it), the runtime will be O(sqrt(n)), exceeding the required O(log(n)) time.
- Remember to wrap around when indices go past the end of the array.
- Don't confuse `size` and `capacity`. The `size` is the number of items stored in the circular buffer. The `capacity` is how much space was allocated.

How to Submit

You must submit the following files to the `proj1` directory.

- `SSLColumn.h`
- `SSLColumn.cpp`
- `SortedSquareList.h`
- `SortedSquareList.cpp`

Even if you ended up not adding any new members to`SSLColumn.h` or `SortedSquareList.h` you should still submit them. You must have still edited them to remove the placeholder lines with "...".

For this project (and definitely for future projects), it is probably best for you to do the primary development in some directory other than the submit directory. That way, you can create all manner of other files (other test programs, reminders to yourself, etc.) that you obviously do not want to submit. Let us say you initially developed your files in the directory "~/work/cs341/p1" (this is completely made up; you can organize your work directories any way you want). If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using this Unix command from the directory you have been working in:

```
linux1% cp SSLColumn.h SSLColumn.cpp SortedSquareList.h SortedSquareList.cpp ~/cs341proj/proj1/
```

Use the Unix script command to show that your code compiles:

```
linux1% cd ~/cs341proj/proj1/
linux1% cp ../../00Proj1/p1test*.cpp .
linux1% script
Script started, file is typescript
linux1% g++ p1test01.cpp SSLColumn.cpp -o t01.out
linux1% g++ p1test02.cpp SSLColumn.cpp -o t02.out
linux1% g++ p1test03.cpp SSLColumn.cpp -o t03.out
linux1% g++ p1test06.cpp SSLColumn.cpp SortedSquareList.cpp -o t06.out
linux1% g++ p1test07.cpp SSLColumn.cpp SortedSquareList.cpp -o t07.out
linux1% g++ p1test08.cpp SSLColumn.cpp SortedSquareList.cpp -o t08.out
```

```
linux1% g++ p1testXX.cpp SSLColumn.cpp SortedSquareList.cpp -o tXX.out
[COMPILATION OF ALL TESTS]

linux1% exit
exit
Script done, file is typescript
```

Do remember to exit from the script command. This creates a file called `typescript` that will record any compilation errors. Yes, we know you can edit this file, but the compilation errors will just show up when we compile the programs again and you will still get lots of points deducted. This step is to compel you to fix any changes needed to get your program to compile on GL without any errors.

Note: `cd` to the appropriate directory if you are submitting late.

Run `t01.out` thru `t10.out` under `valgrind`:

```
linux1% valgrind ./t01.out
...
linux1% valgrind ./t02.out
...
linux1% valgrind ./t10.out
```

If you do not see

```
All heap blocks were freed -- no leaks are possible
```

after each run, then you have a memory leak. You are not ready to submit. Go back and fix your program. If `valgrind` reports other memory errors (e.g., reading from invalid memory or writing to invalid memory) then you don't have a memory leak, but you have some other sort of bug. You should also go back and fix your program.

Now you can delete the executable files with

```
rm t??.out
```

Then you should just have 5 files in your submission directory. Check using the `ls` command. You can also double check that you are in the correct directory using the `pwd` command. (You should see your username instead of `xxxxx`.)

```
linux1% ls
SortedSquareList.cpp  SortedSquareList.h  SSLColumn.cpp SSLColumn.h  typescript
linux1% pwd
/afs/umbc.edu/users/c/h/chang/pub/cs341/xxxxx/proj1
linux1%
```
