C++ Lab 4: Sparse Vectors (Part A)

C++ Lab 4 explores a number of very important approaches and patterns in programming. It is also rather complicated, so a lot of the details will be laid out for you. The main focus of Lab 4 is the concept of a *sparse representation* of data. In Labs 2 and 3, you implemented a 2D matrix. The internal representation of the matrix included every single cell of the matrix; this is called a *dense representation* of the matrix.

However, in many cases a dense representation is not the most optimal approach, and sometimes it is not even possible! For example, in situations where vectors or matrices are very large, but are filled with mostly zero values, a *sparse representation* can be used. In this approach, only nonzero values are stored, and the missing values are assumed to be zero. In cases where there are very few nonzero values, the sparse representation uses memory more efficiently, and can even yield much faster computations.

This lab assignment is focused on building a *sparse vector* that implements this pattern. However, it is quite an undertaking, so it has been split into two parts, each doable in a week. If you follow the approach outlined below, you shouldn't have too much difficulty with this assignment.

The `SparseVector` Interface Specification

The sparse vector class will represent a mathematical vector of some fixed size. Because the internal representation is sparse, the vector will be able to span as many as 2,147,483,647 elements! (That's `INT_MAX` for those of you keeping score at home.)

The name of your class will be **`SparseVector`**, and should be declared in `SparseVector.hh` and defined in `SparseVector.cc`. When all is said and done, it will provide these member functions:

- `SparseVector(int size)`: Single-argument constructor that takes the number of elements in the vector. Note that this is a *fixed-size* vector; its size cannot be increased or decreased.
- `SparseVector(const SparseVector &sv)`: A copy-constructor that should duplicate the passed-in `SparseVector` instance.
- `~SparseVector()`: Destructor to release dynamic resources.
- `int getSize() const`: Total-size (or vector dimension) accessor.
- `int getElem(int col) const`: Accessor for individual elements.
- `void setElem(int col, int value)`: Mutator for individual elements.
- The following handy operator overloads:
    - `operator=(const SparseVector &)`
    - `operator+=(const SparseVector &)`
    - `operator-=(const SparseVector &)`
    - `operator+(const SparseVector &) const`
    - `operator-(const SparseVector &) const`
    - `operator==(const SparseVector &) const`
    - `operator!=(const SparseVector &) const`

  You will have to figure out the correct return types for yourself. (Refer to the C++ operator-overloading guidelines to make sure you follow the best practices!)

  It is desired that your implementation of addition and subtraction be efficient; specifically, with a time-complexity proportional to the number of non-zero values in the vectors being added or subtracted.

This functionality will be built out in two phases, to keep this from being overwhelming.

Internal Representation

A **linked list** is a data structure that contains a sequence of nodes in specific order: each node points to the next one in the list. There are several different kinds of linked lists:

- In a *singly linked list*, each node only points to the next node. Going forward through the list is easy, but you can't go backwards.
- In a *doubly linked list*, each node points to the previous node and the next node. This makes it possible to go forward and backward in the list.

The `SparseVector` will use a **singly linked list** to represent its internal data. All nonzero values will appear as nodes in this list. In addition, no node should have a value of zero.

You can use a `struct` like this to represent the non-zero values in the sparse vector:

```
// A linked-list node for our sparse vector elements.
struct node
{
  int   index;   // Element number, in the range [0, size)
```

```
    int   value;   // The value of this element.
    node *next;    // A pointer to the next node in the linked-list.

    // Node constructor - simply initializes the data-members.
    node(int index, int value, node *next = 0) : index(index), value(value), next(next) { }
  };
```

It's easy to make nodes with code like this:

```
node *n1 = new node(someColumn, someValue);
node *n2 = new node(13, -3423);

n1->next = n2;
```

Through the magic of default arguments, we can also specify a next-node pointer to the `node()` constructor, like this:

```
node *n1 = new node(17, -4);
node *n2 = new node(13, 6, n1);
```

Use a linked list to represent the non-zero elements of your sparse vectors. The most important constraint is to keep the nodes ordered by `index`. We will talk more about using this linked list strategy in class.

Part A - Due in Week 1

This week's tasks focus on building out the basic functionality of the `SparseVector` class. All of the arithmetic operations will be left to next week; they will be much easier once you have worked through these more basic tasks. The tasks for this week are:

- single-argument constructor and copy constructor
- destructor
- assignment operator (`operator=`)
- `getElem` and `setElem` functions

As mentioned before, your class declaration should go into `SparseVector.hh`, and the definitions into `SparseVector.cc`. You don't need to declare any of the arithmetic operations in `SparseVector.hh` yet, since you won't implement them this week. Also, put this week's files into a **`~/cs11/cpp/lab4a`** directory.

It should go without saying by now, but make sure that you use `const` everywhere that is appropriate, and catch invalid arguments with `assert` statements. Be sure to document what are valid inputs, in comments on each function.

Here is a suggested approach to how to implement the `SparseVector` functionality. (If you have a different way that you would like to do it, feel free to do so. Just make sure that your code conforms to the specified interface and functionality, and that it passes the test code included with this assignment.)

1. **Types, data-members, and simple constructor**

   Write up the `SparseVector` class-declaration in `SparseVector.hh`, leaving out all the member functions for the time being.

   Add the `node` type-declaration to the `SparseVector` class declaration. Make sure that it has `private` access, so that it is not visible to clients of the `SparseVector` class.

   Now that your major types are declared, add the `SparseVector` data-members. This includes the size, and a `node`-pointer to the first nonzero element in the list.

   Write the `SparseVector(int size)` constructor, and the `getSize()` member function. These should be very simple to write!

2. **Destructor and helper function**

   Remember that the destructor and assignment operator both have to clean up the current contents of your class. Since they both share that common operation, the simplest approach is to factor that operation out into a helper function that both functions can use.

   Add a `void clear()` helper function to your `SparseVector` declaration and definition. It is only for internal use in your class, so make sure it is `private`.

   The `clear()` helper should go through the `SparseVector`'s internal linked list, and free every single element in the list. You can do this with a simple `while`-loop, calling `delete` on each node in sequence. (Make sure to get the pointer to the next node in the list, before deleting the current node!) Once all nodes are deleted, make sure you set the first-node pointer to 0, to indicate that the list is empty.

   Once you have completed your `clear()` helper function, implement the `SparseVector` destructor. Your destructor should be *really simple* now - it just calls the `clear()` helper.

3. **Copy-constructor and helper function**

   Just like before, the copy-constructor and the assignment operator both have to make a copy of another `SparseVector`'s contents. So, again, make the common functionality into a helper function, so that you only have to write that code once.

   Add a `void copyList(const SparseVector &sv)` helper function to your `SparseVector` declaration and definition. Again, this helper is

for internal use only, so make sure it is `private`.

The `copyList()` helper should exactly duplicate the passed-in `SparseVector` instance. So, `size` should be set to `sv.getSize()`, and a **copy** needs to be made of the linked-list inside `sv`. Since the `copyList()` function is within the `SparseVector` class, it can go ahead and directly access the private data-members of `sv`. That should make it much easier to copy `sv`'s list.

The `copyList()` helper performs its task by iterating over all linked list nodes in `sv`'s internal list. As it iterates, it will allocate new nodes and link them together, producing an exact copy of the list in `sv`.

Once you have completed the `copyList()` helper, you can now easily implement the `SparseVector` copy-constructor. The copy-constructor should just call the `copyList()` helper function, passing the `SparseVector` instance being copied.

4. **Assignment operator (`operator=` member-function)**

   The astute reader will notice that we now have everything necessary to implement our assignment operator. All of the hard work has already been done. So, implement the `operator=` for `SparseVector`, using the `clear()` and `copyList()` helper functions you have already written.

   Don't forget to check for self-assignment! If you don't, your assignment operator will be completely broken. So, check for self-assignment first, then use `clear()` and `copyList()` to perform the assignment operation.

5. **Get-element member function**

   The `int getElem(int idx)` function should be straightforward to implement next. All it does is to iterate through the list of nodes, comparing each node's index to the requested index-value. If the specified index is found, return that node's value. However, if the requested index is not found, return the default value of zero:

   - If you hit the end of the linked list, you know the requested index isn't present.
   - Since the linked-list nodes are in increasing order of index, you also know that the requested index isn't present when the current node's index is greater than the requested index.

   So, if either of those conditions occurs, your function can return zero.

6. **Set-element member function**

   This is the last major function to implement for this week. It is also the hardest one to write, or it *would* be if you tried to knock it all out in one shot. However, the name of the game is to decompose the problem into smaller ones that can be more easily solved.

   The `setElem()` function is challenging because of the different cases that it needs to handle:

   - If the specified value is *nonzero*, `setElem` must find the node for the specified index and store the value. If no node exists, a new one must be created, *in the proper position* in the list.
   - If the specified value is *zero*, `setElem` must find the node for the specified index, and *remove it* from the list. If no node exists, nothing needs to be done!

   This clearly can be handled by two helper-functions, each of which handles one of the above special cases. So, create one helper function:

   ```
   void setNonzeroElem(int index, int value)
   ```

   This helper handles the first case, of either updating an existing node's value, or adding a new node in the right place if it doesn't exist. Make sure to assert that the value argument is not zero! Also, this helper is *not* for users of your class to call, so make it private!

   Then, create a second helper function:

   ```
   void removeElem(int index)
   ```

   This helper handles the second case, of finding and removing a node from the list. If there isn't any node with the specified index, this function can simply return without doing anything.

   Finally, your `setElem` function can simply call the two helper functions, when the appropriate situations occur.

   Both of these helper functions may need to modify the linked list, so this means you will need to have a pointer for the current node in the list, and a pointer to the previous node in the list. This will allow you to add a node, or remove a node, and update the pointers accordingly.

7. **Debug helper-function**

   With a complex piece of programming like this, it's highly likely that there are bugs lurking somewhere in your code. The best way to find and eliminate bugs is to detect them close to where they actually occurred. Half the battle of debugging is knowing where the problem is actually occurring.

   You should add another private helper function to "monitor the health" of your linked list nodes. Specifically, the function should verify that the nodes in your list are in the proper order, based on `index` values. (You could call it `checkListOrder()`, for example.) If any nodes are out of order, print some debug info such as what nodes are out of order, and what their values are. Then, trip an assertion.

Once you have this function implemented, you can call it as a sanity-check in other parts of your code. For example:

- Call it at the end of `removeElem()` and `setNonzeroElem()`, to make sure you didn't mangle your list.
- Call it on your list after making a copy of another list.
- Call it in `getElem()`, before trying to traverse the list.

You may think of other places to call this function too.

That's It?? Too Easy!

Once you have completed this week's functionality, you can compile and test your class with the supplied test routine checksva.cc.

Make sure that the test completes 100% successfully, with no segmentation faults or memory errors of any kind. The test program does nothing invalid, so you should see a 100% pass rate. If you can't figure out why a particular test is failing, feel free to ask for help.
