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

Last week you implemented the simpler features of a `SparseVector` class, which internally represents a vector of integers as a singly linked list of nonzero element values. This week you will complete the functionality of this class, and work out any remaining bugs.

Make a copy of last week's work, into a directory `~/cs11/cpp/lab4b`, and implement this week's functionality on the copy.

Part B - Due in Week 2

This week you will focus on implementing the remaining `SparseVector` operations, including equality operators and simple arithmetic operators. Just like last week, we will build out these features step by step, decomposing them to make them easier to complete.

As before, if you have another way of implementing these features then feel free to do these things in your own way. This is simply a reasonably straightforward way of approaching this assignment, but it certainly isn't the only way...

1. **Equality and inequality operators**

   First, implement the equality operator (`==`) to compare one `SparseVector` instance to another. This should be a straightforward function to write:

   - If the two sparse-vectors are different sizes then they are not equal!
   - Write a single loop that iterates down both sparse-vectors' internal linked lists. The lists are not equal if the two nodes' `index` or `value` fields don't match, or if one list ends before the other.

   Once the equality operator is completed, the inequality operator (`!=`) can be implemented in terms of the equality operator.

2. **Create stubs of helper functions**

   Again, we will use a couple of helper functions to dramatically simplify the implementation of our addition and subtraction operators. However, these helper functions are pretty complicated, so for now you can just add them to your class, but leave the implementation empty. This is called "creating a stub of the function," or simply "stubbing out the function." It is a very helpful approach when you want to hook together more complicated features.

   First, you will want to add a helper function with the signature:

   ```
   void addSubVector(const SparseVector &sv, bool add);
   ```

   Add this to your `SparseVector` class-declaration, and then also add it to the definition, but *don't implement it yet.* This function will implement the logic of adding or subtracting another vector. Obviously, there will be a significant chunk of code to implement this operation, and we can control whether the other vector is added or subtracted with the `bool` argument.

   You will also want to add a second helper function with the signature:

   ```
   void removeZeros();
   ```

   Just like before, add this to your `SparseVector` class-declaration and definition, but don't implement it yet. When two vectors are added or subtracted, it is possible that some elements will end up with zero values. However, it is pretty insane to try to take care of those during the add/subtract operation itself, so after the arithmetic is done, we can call `removeZeros()` to clean out any elements whose value became zero from the math.

   Remember to make both of these helper functions `private`, since they are only for internal use within the `SparseVector`.

3. **Arithmetic operators, compound assignment operators**

   Remember that with arithmetic operators, you generally want to implement the compound assignment operators first, and then implement the simple arithmetic operators in terms of the corresponding compound assignment operators. So, the first operators to implement are the `+=` and `-=` operators. Go ahead and implement each of these operators, using the `addSubVector()` helper function in the implementation. Because of this helper function, each of these operators should be very simple to implement.

   Next, implement the `+` and `-` operators in terms of `+=` and `-=`. Remember that these operators should return a `const SparseVector` containing the result of the addition or subtraction.

4. **Implement `removeZeros()` helper function**

   As mentioned previously, vector addition or vector subtraction will sometimes result in nodes with a value of zero. However, you don't want to deal with removing those nodes during the add/subtract operation. Rather, after the add/subtract is completed, a single pass will be made over the linked list, and any nodes with a value of zero will be removed.

   This helper function should traverse the entire linked list, removing any nodes whose values are zero. Just like with the `setElem()` function, this loop should use two node-pointers instead of just one:

   - `currNode` should be the current node in the loop

- `prevNode` should be the node *immediately before* the current node in the loop

This way it will be easier to remove the zero-value nodes. You can initially set `currNode` to be the first node, and `prevNode` to be 0. Then, at the end of each loop iteration, set `prevNode = currNode`, and then set `currNode = currNode->next`.

Don't use `getElem` or `setElem` inside this function; it will cause your implementation to be *much* slower than it should be. Since `getElem` and `setElem` will be O(N) performance, calling them from `removeZeros` can make it O(N²).

5. **Implement a `checkZeros()` debug helper.**

   This debug helper function serves the same kind of purpose as the "check list order" helper from last week. You can use it as a sanity-check in any member function that changes your SparseVector's internal linked list. (The `setElem()`, `operator+=()`, and `operator-=()` functions, in particular.)

   This function should be private since it is for internal use only. Also, use the function in assertions, so that this test can be compiled out of your class if desired. Finally, if you actually find nodes with zero values, you probably will want to print out the details of the offending nodes.

6. **Implement `addSubVector()` helper function**

   This is probably the most complicated helper function in the whole `SparseVector` implementation. Like the equality operator, it must traverse two linked lists at the same time, updating the local list with the sum or difference of the two vectors.

   The lecture notes contain information about how to add or subtract two sparse-vector data structures. This is a tricky operation, so make sure you do things correctly.

   Again, don't use `getElem` or `setElem` inside this function; it will cause your implementation to be much slower than it should be. Calling `getElem` and `setElem` from this function will make it O(N²).

   One suggestion for this function is, don't use the `add` flag to directly switch between addition and subtraction. Instead, compute a `sign` value from it, and use that in your addition/subtraction operation. In other words, instead of doing this:

```
for (i = 0; i < size; i++)
{
  if (add)
    a[i] = a[i] + b[i];
  else
    a[i] = a[i] - b[i];
}
```

   Do this:

```
sign = (add ? 1 : -1);

for (i = 0; i < size; i++)
a[i] = a[i] + sign * b[i];
```

   This can make your code a bit simpler.

Testing and Debugging

This week you can use another test program, `checkspvect.cc`, to exercise your class' functionality. This program includes the tests from last week, but also has some tests for the sparse-vector equality and arithmetic operations.

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 run into problems, you can use `gdb` to debug your program. See the lecture notes for information on how to use `gdb` for debugging.
