CS241
Project 3: SortedList

In this project, you will implement a doubly-linked list that maintains a list of integers in sorted order. In other words, the user has no control over where in the list items are inserted; they will automatically be placed into the correction position to keep the list in sorted order at all times. We will call this data structure a SortedList.

Furthermore, in an attempt to minimize traversal time, when a new integer is added to the list, the data structure will attempt to guess if the newly-inserted integer will end up being placed in the first half or the second half of the list, and will traverse the list either from the front or back to find the correct insertion position. Details of how to do this are provided below. Note that this is only a guess, and sometimes it is inevitable (but determinable) that the wrong direction of traversal will be selected.

SortedLists are based on the following node structure, which follows the standard doubly-linked list paradigm:

```
struct node
{
    int data;
    node *prev;
    node *next;
};
```

The SortedList structure itself maintains two pointers, one to the head of the list and one to the tail. The items within the list (the integers stored) increase in value from head to tail.

SortedLists have a few operations you will need to implement:

- Add an item to a SortedList. Given a new integer called `item`, this operation places `item` in the correct spot within the SortedList. You may assume `item` does not already exist in the list (items will never be duplicated).

  When the correct location for the item is somewhere in the middle of the list (i.e., the correct position is not either before the head or after the tail), traversing the list is obviously required to find the correct position in which to place the item. In an attempt to make this traversal as fast as possible, a SortedList will assume that the items it stores are roughly equally spread out in the range between the smallest integer (stored at the head of the list) and the largest integer (stored at the tail of the list), and therefore an item to be inserted will *probably* be in the first half of the list if it is numerically closer to the smallest integer rather than the largest, and therefore traversing the list to find the correct insertion point should begin from the head of the list and traverse the list forwards (following next pointers). Similarly, if the item to be inserted is numerically closer to the largest item in the list rather than the smallest, then traversal should begin at the tail of the list and operate backwards (following prev pointers).

  For example, if a list (of any length) currently has a smallest integer of 100 and a largest integer of 200, then adding 110 to the list *probably* means 110 will end up towards the first half of the list, and so one should traverse the list starting from the head to find the correct insertion point. To be clear, this is only a guess, not a guarantee. Certainly, if the example list consisted of 100, 101, 102, 103, 104, and 200, then traversing the list forwards to find the correct spot for 110 will take longer than if we traversed it backwards.

If an item to be inserted is equally spaced between the largest and smallest elements, traverse from the head forwards.

- Delete an item from a SortedList. Given an integer called `item`, this operation removes `item` from the SortedList if it is present in the list, and does nothing if `item` is not present.

  If the item to be deleted is not located at the head or the tail of the list, traversing the list will obviously be required to discover where (and if) the item is present in the list. In this situation, the SortedList will use the same estimation procedure as for insertion to make an educated guess as to whether a forwards or backwards traversal will be faster.

- Take a slice of a SortedList. Given two integers called `low` and `high`, this operation creates a new SortedList consisting of all the integers from the original SortedList that are **greater than or equal** to `low`, and **less than** `high`. For instance, in a list containing the integers [2 5 7 10 12 15], a slice given `low=6` and `high=15` should return a new SortedList of [7 10 12].

- Clear the SortedList. This operation removes all the integers from the SortedList.

- Test if the SortedList is empty. This operation returns true if the SortedList is empty, and false otherwise.

What you need to do:

- Download the starter code, consisting of main.cpp, SortedList.h, and SortedList.cpp.

- Implement all the methods that are not already implemented. I suggest going in this order:

  - Write the default constructor and isEmpty() first. These should be trivial.

  - Get add() and the operator<< overload working first. Write some code in main() to make a new SortedList and add some integers to it, and cout the list to make sure everything is going in OK.

    Note: Add returns the number of items traversed in the list to find the correct position to insert at. This should be printed in your output (the file reading loop already does this for you).

    Note: The operator<< overload should print out the list forwards and backwards (and size counts manually determined from the two traversals), so you can double check all your pointers are working. This is a great "sanity check" operation you can use as you're implementing the later functions to check if you missed any pointers somewhere.

    At this point you can start reading from the sample input files.

  - Write clear() next. Remember, you need to traverse and delete all the items. Then write the destructor. Write code in main() to test clear(). You can't really manually test the destructor, since it is called automatically.

- Write the copy constructor, assignment operator, and remove method in whatever order you like. Write code in main() to test the copy constructor and assignment operator.

- Write slice last --- slice requires the copy constructor and/or assignment operator to be implemented correctly. Write code in main() to test.

General hints and suggestions:

- Make sure to test all your operations on empty lists, lists with one element, and lists with more than one element. Also make sure to test your operations where manipulation of the list will involve the head, the tail, or both together.

- I recommend splitting insert into a number of cases:
  - What to do with an empty list (because you will need to change head and tail).
  - What to do with a list with only one element (because you will need to change either head or tail).
  - What to do with a multiple-element list (then break this up by traversal direction; head or tail still may need to change).

- You can split up remove and slice similarly.

- The code for the copy constructor and assignment operator is almost identical (since they are both copying operations).