Project 1, Square Lists

**Objectives**

The objective of this programming assignment is to have you practice implementing a new data structure and also to gain some experience using the data structure from the C++ STLs.

**Introduction**

For this project, you will implement a square list data structure. A square list is a linked list of linked lists where each linked list is guaranteed to have a maximum of O(√n) items (O(√n) + 1 to be exact, but…):

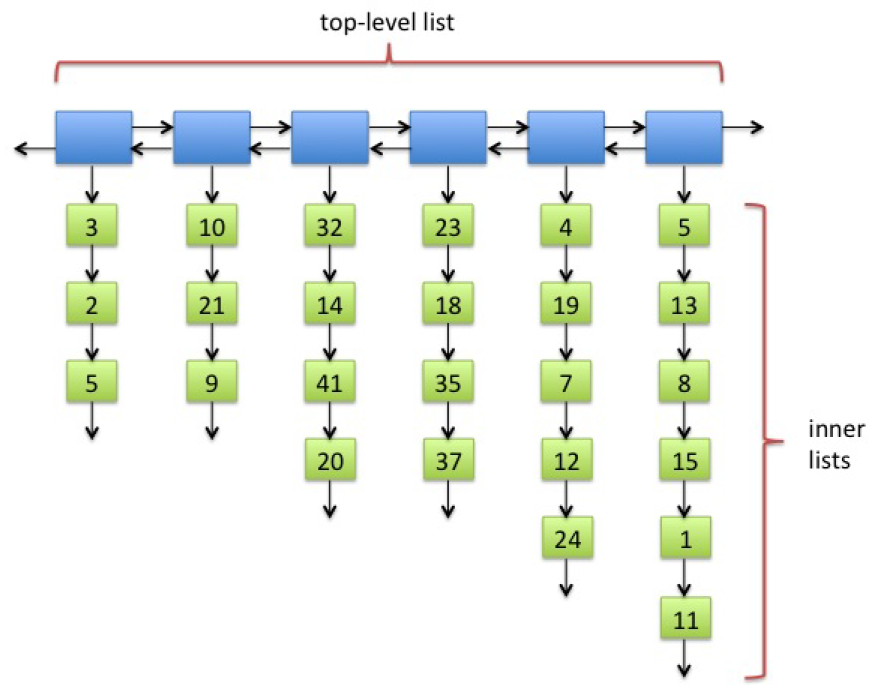


Fig. 1: A typical square list.

Figure 1 shows a typical square list of 25 integer values. The positions of the items are important. The ordering of the items in this example is: 3, 2, 5, 10, 21, 9, 32, 14, 41, 20, 23, ..., 15, 1, 11.

The idea of a square list is that we want to keep the length of the top-level and of all the inner lists bounded by O(√n). That way we can find the ith item of the list in maximum of O(√n) time. For example, if we want to find the 9th item in the list, we can progress down the top-level list and check the length of the inner lists. We know the 9th item cannot be in the first inner list, since it only has 3 items. It also cannot be in the second inner list, since the first two lists combined only has 6 items. Instead, we can find the 9th item of the square list by looking for the 3rd item of the third inner list, which turns out to be 41.

To accomplish this search in O(√n) time, we must be able to determine the length of each inner list in O(1) time. In the worst case, we have to search through O(√n) items of the top-level list before we find the inner list that holds the ith item. An additional O(√n) steps will allow us to find the desired item in that inner list, since the length of each inner list is also O(√n).

The main difficulty in maintaining a square list is that as we add items to and remove items from a square list, the length of the inner lists can change. This can happen in two ways. First, obviously, when we add items to or remove items from an inner list, the length of that inner list changes. Secondly, the length of an inner list relative to O(√n) can also change when we add or remove items elsewhere in the square list because doing so changes the value of n. For example, the 5th inner list in Figure 1 has 5 items. This happens to be exactly √25. If we removed all 10 items from the first 3 inner lists, that would leave us with only 15 items in the entire square list. Now the length 5th inner list is bigger than √15 even though the length of the 5th inner list remained at 5.

**Square List Maintenance**

Our goal is to make sure that the top-level list and all the inner lists have lengths bounded by O(√n). It is too expensive to require that our square list *always* has √n inner lists, each with √n items. Instead, we maintain the following two conditions:

**Condition 1:**

Every inner list has ≤ 2 √n items.

**Condition 2:**

There are no adjacent short inner lists, where short is defined as having ≤ √n/2 items.

Notice that neither condition says anything about the length of the top-level list. Instead, we claim that if Condition 2 holds, then the top-level list cannot have more than 4 √n items. To see this, suppose the contrary. That is, suppose that the top-level list has more than 4 √n items. (Yes, this is the beginning of a proof by contradiction.) Then, there must be more than 2 √n inner lists that are not short (otherwise, two of the short inner lists would be adjacent). Thus, the total number of items in these non- short lists must exceed 2 √n × √n/2 = n. This is a contradiction because n is the number of items (by definition) and cannot exceed itself. Therefore, the number of inner lists (and thus the length of the top-level list) must be bounded by 4 √n.

These observations allow us to maintain the O(√n) bounds on the lengths of the top-level list and the inner lists using the following procedure:

|  |
| --- |
| Consolidate:   1. Traverse the top-level list. 2. Whenever an empty inner list is encountered, remove that inner list. 3. Whenever two adjacent short inner lists are encountered, merge them into a single inner list. (See Figures 2 and 3.) 4. Whenever an inner list is found to have more than 2 √n items, split them into two lists of equal length. (See Figure 4.) |

Some notes on the Consolidate procedure:

* Our strategy for this project is to run Consolidate after every operation that adds an item to or removes an item from the square list. (This is a simplification. See Addendum for a longer explanation.)
* When two short lists are merged into one, the order of the items in the square list must not change. In Figure 2, the original order of the items in the short lists where 10, 21, 32, 14. In Figure 3, the order of the items in the merged list is the same.
* We need the data structure for the inner list to support a merge operation in constant time. A singly linked list with a tail pointer would work.
* In Figure 4, the inner list that is too long has an even number of items. If a long list has an odd number of items, then after the split, one list will have one more item than the other. This does not affect the asymptotic running time.
* When a long list is split, the order of the items must be preserved. (See Figure 4.)
* Without any splits, the total running time for the Consolidate procedure is O(√n), because we can merge short lists in constant time.
* The split step can be costly because it takes O(t) time to split an inner list in half, where t is the length of the inner list. We can show using amortized analysis that splits do not happen very often. The proof is not hard but is beyond the scope of this course. The amortized analysis gives an amortized running time of O(√n) for most of the list operations (except searching). The amortized analysis shows that any mix of m list operations (not including searching) will take a total running time of O(m√n). Thus, the amortized time for each of the m square list operations is O(√n). Although, it is tempting to think of the amortized running time as an "average" running time, this is not accurate because the amortized analysis does not depend on the sequence of operations being "nice" or "average" in any way. Even when an adversary chooses the nastiest sequence of operations which results in the maximum number of splits, the total running time for that sequence of m operations will still be bounded by O(m√n).

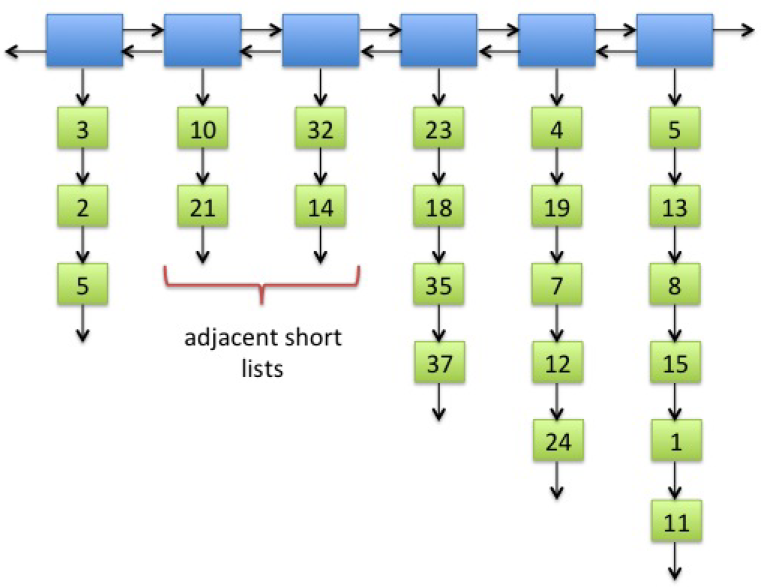


Fig. 2: A square list with adjacent short inner lists. Note that 2 < √22/2 ≈ 2.345.

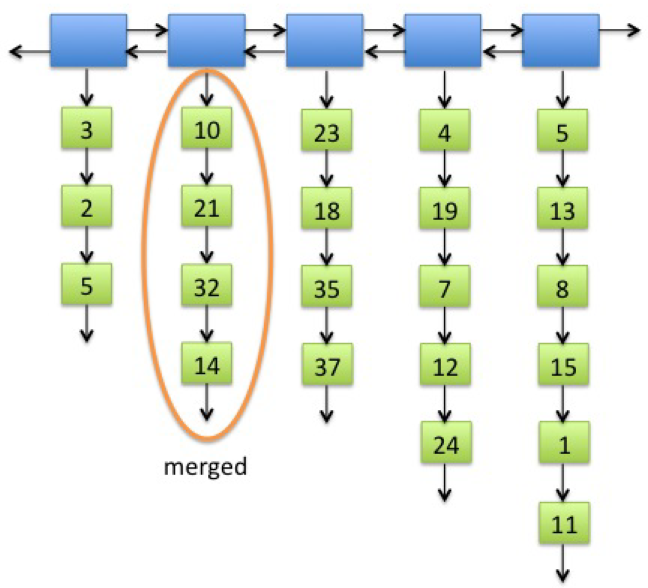


Fig. 3: Adjacent short lists merged.

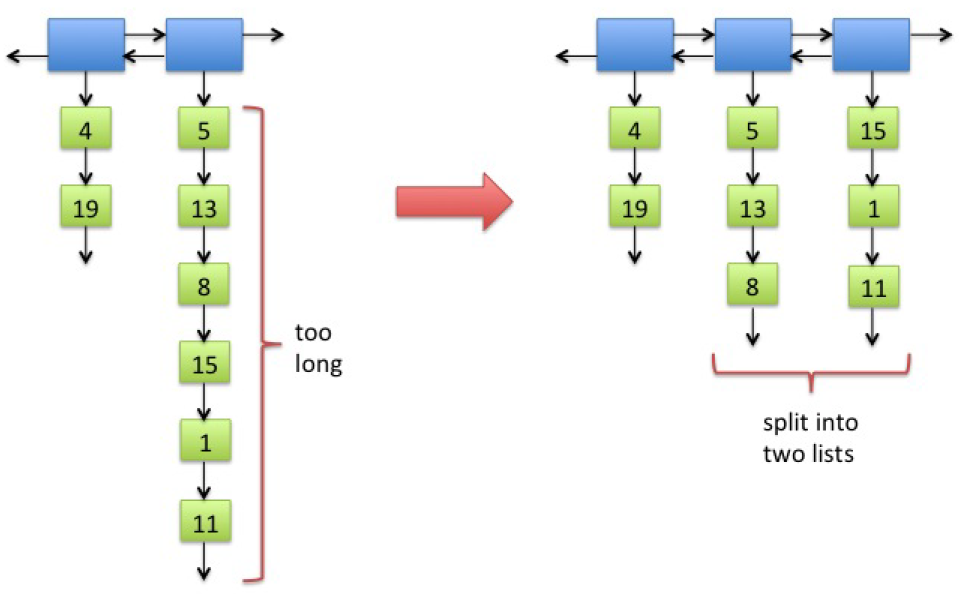


Fig. 4: A long inner list split into two lists. Note that 6 > 2 √8 ≈ 5.657.

**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 a square list data structure that hold C++ integer values. The inner lists must be implemented as a custom singly-linked lists with (at minimum) a front and tail pointer. This implementation ***should be your own work***. In particular, you must not use any data structures from C++ STLs for the ***inner lists***. For the top-level list, you must use the generic LinkedList (called list) class from the C++ STLs.

This assignment specifies the interface between the main program and your square list implementation, but you are free to design the data structure for the inner list as you please (subject to performance considerations listed below). Note that design is part of the grading criteria, so you do need to apply good design principles to your inner list data structure.

The header file for your inner list class called InnerSquareList (which is custom) is given. Read through the functions and their specified run times. You are:

1. NOT allowed to change the function names or parameters
2. Should not need to add any other functions
3. Should not need to make any changes to the header file at all

The header file for your SquareList class is also given. Much like the previous class:

1. NOT allowed to change the function names or parameters
2. Should not need to add any other functions
3. Should not need to make any changes to the header file at all

Finally, SquareList is accessible from a main program called Proj1Test.cpp. The output of Proj1Test.cpp should look something like the file Proj1Test.txt. The output may differ somewhat depending, for example, on how you split long lists with odd length. Your code must compile with Proj1Test.cpp ***without alteration***.

**Implementation Notes**

Here we list some recommendations and point out some traps and pitfalls.

* Start this project early. You will most likely need all the time you get. (Those that start 24 hours before… not going to get it done!)
* Apply the incremental programming methodology. Implement one or two methods and fully debug these methods before writing more code. Implement and fully debug your inner list data structure before you do anything for the SquareList class.
* Your data structure for the inner lists ***cannot*** be a doubly linked list.
* A front and tail pointer is not a node (which would not be very useful in a singly-linked list). A tail pointer is simply a reference to the last node in the linked list. Tail pointers make the merge operation fast, since we don't have to search the linked list to find the last node.
* Common errors while implementing singly linked lists:
  + Off-by-one errors: inserting or removing an item in the position one spot before or after
  + Forgetting to update the size of the list
  + Forgetting to update the tail pointer when the last item is inserted or removed
* You do not have to implement the full suite of list operations for the inner list data structure — just whatever is needed to support the SquareList operations.
* Be paranoid! Check that you are not merging a list with itself.
* The logic for the consolidate() method can be tricky. Think through this before coding.
* Do not use two iterators simultaneously for the top-level list. You will need to use the remove() method when you delete empty inner lists and when you merge short lists. When you use one iterator to invoke the remove() method, it will invalidate the other iterator.
* Do write your own main program for testing. Proj1Test.cpp is a basic test that checks a few features and that your program will compile correctly. It is not a comprehensive test. Just because your program runs correctly with Proj1Test.cpp, it does not mean you will receive 100. It does not mean your program does not have bugs. Your program will be tested and graded against other main programs with bigger, meaner and nastier test cases. Just as an example, Proj1Test.cpp never checks if removeFirst() works correctly when the square list is empty, but you should.
* Document your code!!

**What to Submit**

Follow the course project submission procedures. You should copy over all of your C++ source code under the src directory. You must also supply a make file. In particular, the following Unix commands should work. (Check this.)

cd ~/cscs221proj/proj1

make

make run

make clean

Finally, all of this should be ***zipped*** together and named Project2.zip.

**Discussion Topics**

Here are some topics to think about to help you understand square lists. You can discuss these topics with other students without contradicting the course Academic Conduct Policy.

1. Suppose you start with an empty square list and keep inserting items in the front of the list. When does the first merge occur? ***(Draw it out!!)***

2. What is the smallest number of items you can have in a square list that has 11 inner lists?

3. Do we ever encounter long inner lists that have to be split (other than the first inner list) if we only allowed insertion and removal at the beginning of the list?

4. After you split an inner list, is it possible that the same inner list has to be split again after the very next square list operation? after two operations? when could the next split occur?

5. Can you ever encounter 3 short lists in a row during the Consolidate procedure? Does it matter? and should you write code whose correctness depends on the answer to these questions?

(Fall 2013)