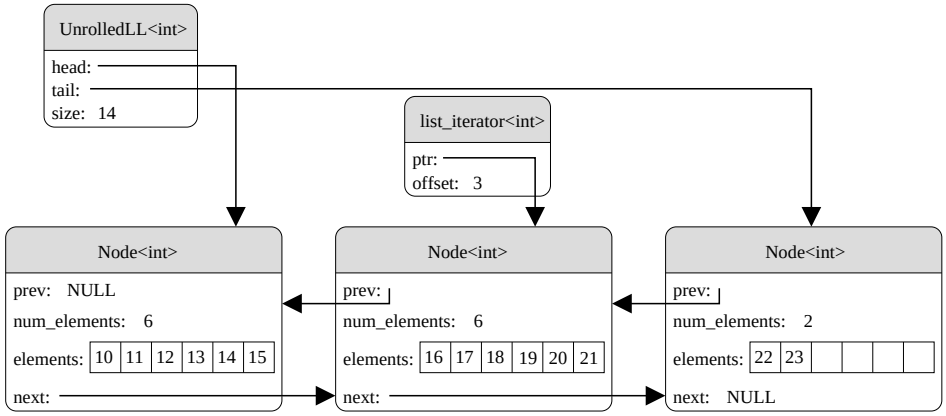
CSCI-1200 Data Structures — Spring 2019
Homework 5 — Unrolled Linked Lists

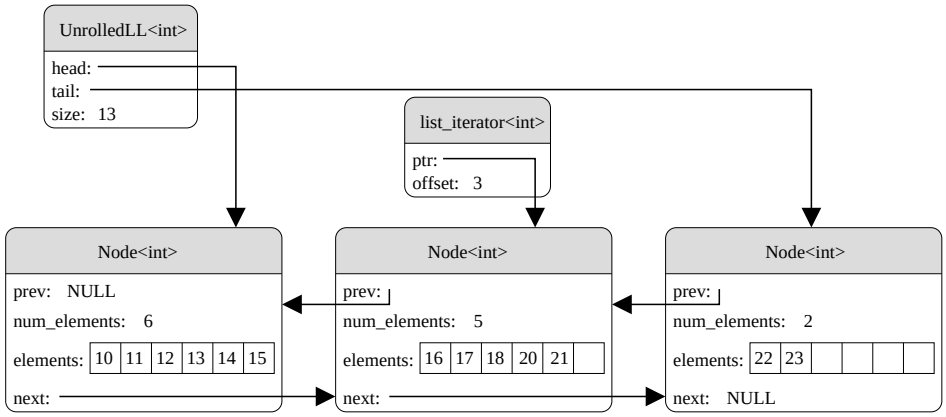
In this assignment you will modify the `dslist` class from Lecture 11 to implement an *unrolled linked list* data structure. This data structure is very similar to a standard doubly linked list, except that *more than one element* may be stored at each node. This data structure can have performance advantages (both in memory and running time) over a standard linked list when storing small items and can be used to better align data in the cache. You're welcome to read more about unrolled linked lists on Wikipedia (but you're not allowed to search for or use any code or pseudocode that might be available online!). **Please read the entire handout before beginning your implementation.**

To complete the assignment you will modify the helper classes (`Node` and `list_iterator`) as well as the main class, `dslist`, which will be renamed "`UnrolledLL`". Below is a picture of the relationships between the classes (compare this to the picture of the `dslist` class from Lecture 11):



Each `Node` object contains a *fixed size* array (size = 6 in the above example) that will store 1 or more elements from the list. The elements are ordered from left to right. From the outside, this unrolled linked list should perform exactly like an STL list containing the numbers 10 through 23 in sorted order. Note that to match the behavior, the `list_iterator` object must also change. The iterator must keep track of not only which `Node` it refers to, but also which element within the `Node` it's on. This can be done with a simple offset index. In the above example, the iterator refers to the element "19".

Just like regular linked lists, the unrolled linked list supports speedy `erase` operations from the middle of the list. For example, if we call erase with the iterator shown above, this is the resulting picture:



Note that the `size` of the list has decreased by one, the element 19 has been removed from the middle node, the elements after 19 have been shifted to the left, and the number of elements in that node has been

decreased. It's perfectly OK for some of the nodes to be only partially filled. (If we tried to enforce that all nodes stay full, this operation would become just as expensive as the STL `vector::erase` operation!) If removing an element causes a node to become empty, the entire node should be removed from the list.

Your Tasks

Your new data structure should be a templated class named `UnrolledLL` and its public interface should mimic the STL `list` container class. It should support the member functions: `size`, `front`, `back`, `push_back`, `pop_back`, `push_front`, `pop_front`, `erase`, and `insert`. Like an STL `list`, the public interface provides bi-directional forward iterators (i.e., `++` and `--` operations), but you don't need to do reverse iterators. Your class constructors, assignment operator, & destructor should properly allocate and deallocate the necessary dynamic memory. Similar to Homework 3, you must provide print functionality for debugging – this time by implementing the stream output operator for your data structure that prints an ASCII version of the internal data structures. We provide examples of the information that should be contained in this output, but your formatting may be different.

The number of elements stored in each `Node` is fixed at compile time, but you should try different values as you work. Please use a global constant defined at the top of your `UnrolledLL` class header file to control this. Your submitted assignment should have 6 elements per node:

```
const int NUM_ELEMENTS_PER_NODE = 6;
```

To get started, we highly recommend that you study and heavily borrow from the `dslist` class that we discussed in Lecture 11. The code is available on the course website. We provide a `main.cpp` file that will exercise the required features of your class, but it *does not include tests for each "corner" case.* You should add your own test cases to the bottom of the `main.cpp` file. Discuss in your `README.txt` file your strategy for thoroughly debugging your assignment.

To earn full credit, your program must be free of memory errors and memory leaks. The homework server is configured to run Dr. Memory on your program to check for memory problems.

Evaluation

What is the order notation of each of the member functions in your class? Discuss any differences between the `UnrolledLL`, `dslist`, and STL `list` classes. Evaluate the memory usage of your initial implementation in the *worst case*. Give a specific example sequence of operations where the data structure is inefficient. What is the average number of elements per node in this case? Discuss in your `README.txt`.

Extra Credit

For extra credit, change your implementation to improve this worst case memory performance. The typical strategy for this data structure is to incrementally merge or split the contents of neighboring nodes such that each node is at least 50% full (except possibly the last node) at the end of each operation. Which operations did you modify? The order notation of these operations should *not change*. Discuss in your `README.txt`.

Additional Information

You may not use STL vectors or lists for the *implementation* of your new data structure. STL lists are only allowed in the `main.cpp` to test the new data structure and make sure it matches the standard interface for lists.

Use good coding style when you design and implement your program. Be sure to make up new test cases to fully test your program and don't forget to comment your code! Use the template `README.txt` to list your collaborators and any notes you want the grader to read. A one day extension will be granted for HW5 if by Wednesday 11:59 PM you have 5 points on Test Case 10.