A2: Linked Vectors

1 Overview

In this assignment, we will explore how to load, store, and manipulate ordered data in a custom object and analyze your design. You should work to make your code efficient. Additionally, this assignment will develop your understanding of memory organization by managing a structure made of linked nodes.

2 Objectives

In this assignment you will:

- Write a C++ templated class.
- Write the code for a basic iterator.
- Analyze the runtime of your hybrid container.

3 Useful Resources

Review the lecture notes and provided example code for some insight into how linking nodes works. You will also want to read the C++ references provided below:

- Useful algorithms: `<algorithm>`
- Vector class: `std::vector`

4.1 Programming Problem

Complete the following programming assignment and submit your answers to Autolab via the "A2 Programming" assignment. Submit only your LinkedVector.hpp file.

Expect this section to take 15-20 hours of setting up your environment, reading through documentation and base code, and planning, coding, and testing your solution.

Problem 4. (60 points)

Your task is to implement a new container that is a cross between a linked list and a vector. This new container will be used to store an sequence of unique elements in increasing order. In order to provide access to the sequence of data stored within, you will also create an inner class iterator.

The data organization will be done as follows:

- The values will be stored within a doubly-linked list using the `cse250::vNode` struct.
  - The head of the list should be maintained within the `LinkedVector<T>` class.
- Initially there should be a single node in the list, when the container is empty.
- Upon removal we never call rebalance, so once the list is resized it should remain that size.

4.1.1 Size Parameters

Parameters that we will use when discussing size:

- n – the size of the container (number of values stored in your `LinkedVector<T>`).
- k – the last node in the sequence of nodes in the `cse250::vNode` linked list (starting from 0).
- $k+1$ – the number of `cse250::vNode`s in the linked list (node 0 is the head, node k is the tail).
- n_i – the number of elements stored in vector `data` in the `cse250::vNode` at the i-th position in the list.

4.1.2 Insertion Data Organization

Insertion of elements should maintain storage in increasing order. You may assume that the template type `T` will have an `operator<`, `operator==`, and `operator!=` implemented (for comparisons). These should be used to maintain the ordering among elements.

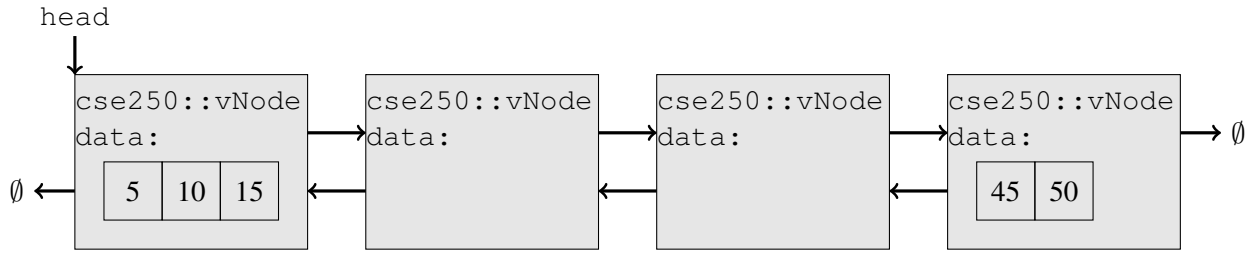
Note: **insert should never create or remove linked list nodes**. Only subsequent calls to `rebalance` should create new nodes and we never decrease the number of nodes within the list.

4.1.3 When and How to Rebalance

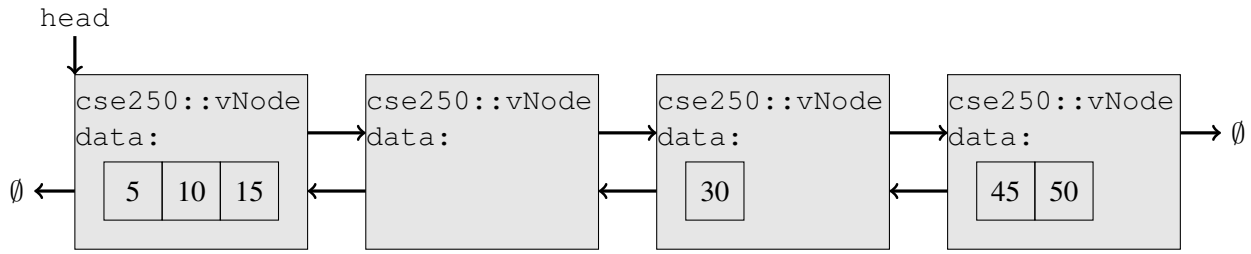
The following conditions must be maintained during insertion:

- Insertion should maintain that for each i, $0 \leq i \leq k$: $n_i \leq 10(k+1)$. If this is violated, a call should be made by `insert` to `rebalance`.
- Insertion of an item that could go at the end of the `data` vector in one node or at the start of the `data` vector of the neighboring node should insert into the node with the smaller size (see section 4.1.4 regarding insertion of 42 for illustration). In the event of a tie, insert into the left node.

– If a value goes between the end of one node and the beginning of another node but there are empty nodes between, insert into the rightmost empty node. E.g., suppose the following is the state of your LinkedVector:



If you were to insert the value 30, it should result in the following:



- No values should be stored in duplicate.

Additionally, the following should be met:

- Initially, when $n = 0$, there should be one `cse250::vNode` in the list (so $k = 0$ initially).

If the insertion into node i caused the size n_i to violate its upper bound (i.e., $n_i > 10(k + 1)$), a rebalance call should perform the following task:

(1) Allocate a new doubly-linked list with $\lfloor \sqrt{n} \rfloor + 1$ nodes.

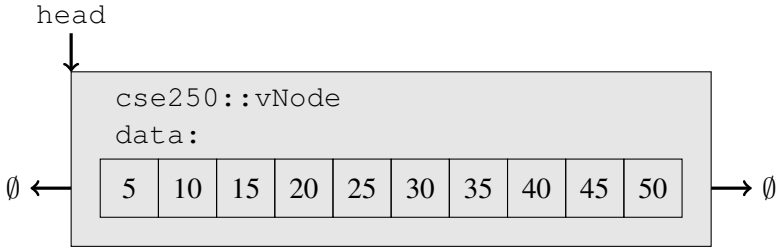
(2) Evenly distribute the values into the newly allocated nodes.

  - If the items don't distribute evenly, distribute the extra items across the list starting from the first node.

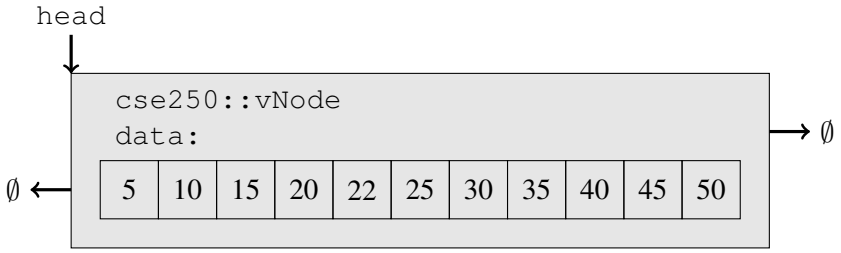
(3) Cleanup the old list and use the new list for storage.

4.1.4 Insertion and Rebalance Example

Suppose we have inserted the values $5, 10, 15, \ldots, 50$. Your internal list would look like this:



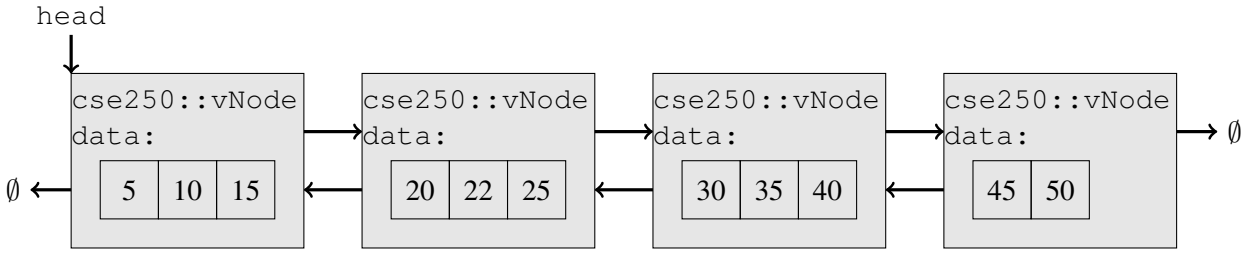
In this case, you would have $n = 10$ values, $k = 0$ (since we have $k + 1 = 1$ nodes), and $n_0 = 10$. Now, suppose that 22 is inserted. This would give us:



In this case, you would have $n = 11$, $k = 0$, and $n_0 = 11$. This violates the property that $n_0 \leq 10(k + 1)$ since $11 \not\leq 10$, thus requiring a rebalance.

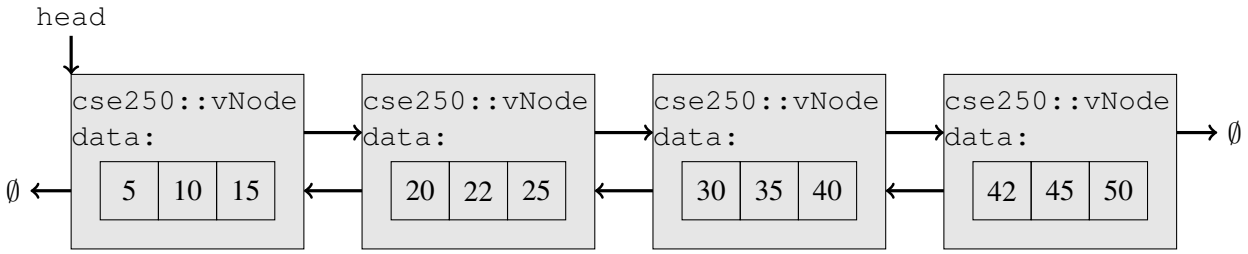
- Start by computing a new value of k: $k = \lfloor \sqrt{n} \rfloor = 3$.
- Create a new list with $k + 1 = 4$ nodes.
- Redistribute the values among these newly created nodes.

The result from calling `rebalance` would be the following:



Note that the last `vNode` only has 2 elements since we could not evenly distribute the 11 values across 4 nodes. In this case, you would have $n = 11$, $k = 3$, and $n_0 = 3$, $n_1 = 3$, $n_2 = 3$, $n_3 = 2$.

Lastly, consider the insertion of 42. This could go in either of the last two nodes since $40 \leq 42 \leq 45$, however the last node is storing fewer elements in `data`. Therefore, we would add 42 to the last node resulting in:



In this last case, you would have $n = 12$, $k = 3$, and $n_0 = 3$, $n_1 = 3$, $n_2 = 3$, $n_3 = 3$.

4.1.5 LinkedVector Methods Overview

- Constructor `LinkedVector<T>::LinkedVector();`
  - perform any default initialization necessary.
- Destructor `LinkedVector<T>::~LinkedVector();`
  - cleanup any allocated memory.
- `bool LinkedVector<T>::insert(const T& value);`
  - insert the value into your storage in increasing order.
  - duplicate values should only be stored once.
  - return `true` if the value is inserted and return `false` if it already exists.
- `void LinkedVector<T>::rebalance();`
  - rebalance the load by creating new linked nodes to meet the required load balancing.
  - this private method should only be invoked by `LinkedVector<T>::insert`.
- `LinkedVector<T>::Iterator LinkedVector<T>::find(const T& value) const;`
  - return an iterator with the location of the value if found.
  - if the value was not found, return an iterator to the end.
- `bool LinkedVector<T>::erase(const T& value);`
  - if the value was present, remove it and return `true`.
  - if the value was not present, return `false`.
- `std::size_t LinkedVector<T>::size() const;`
  - return the count of the number of items currently stored.
- `LinkedVector<T>::Iterator LinkedVector<T>::begin();`
  - return an iterator to the beginning of the sequence stored within the `LinkedVector`.
- `LinkedVector<T>::Iterator LinkedVector<T>::end();`
  - return an iterator to the end of the sequence stored within the `LinkedVector`.

4.1.6 LinkedVector::Iterator Methods Overview

- Constructor `LinkedVector<T>::Iterator::Iterator(LinkedVector<T>* container);`
  - perform any default initialization necessary (for the iterator).
  - the base code already initializes the associated container to the argument value and position to `nullptr` (this position is probably incorrect and should be fixed).
- Destructor `LinkedVector<T>::Iterator::~Iterator();`
  - cleanup any allocated memory (for the iterator).
  - if this method is unneeded, delete it.
- `T LinkedVector<T>::Iterator::operator*();`
  - dereference operator.
  - return the data at the current position in the sequence.
  - invoking operator* at the end of a sequence is undefined behavior.
- `LinkedVector<T>::Iterator& LinkedVector<T>::Iterator::operator++();`
  - prefix ++ operator.
  - increment the iterator to the next value in the sequence.
  - return a reference to `*this`.
  - invoking ++ at the end of a sequence is undefined behavior.
- `LinkedVector<T>::Iterator LinkedVector<T>::Iterator::operator++(int);`
  - postfix ++ operator.
  - increment the iterator to the next value in the sequence.
  - return a copy of the iterator before it was modified.
  - invoking ++ at the end of a sequence is undefined behavior.
- `bool LinkedVector<T>::Iterator::operator==(const LinkedVector<T>::Iterator& rhs) const;`
  - return `true` if both iterators point to the same element within the same sequence.
  - return `false` otherwise.

4.1.7 Assumptions/Notes

- Insertion of up to 1000000 elements may occur.
- After calling `LinkedVector<T>::begin()` or `LinkedVector<T>::end()`, no container modification will occur.
  - If the container is modified, new Iterators will be constructed and old Iterators will be discarded.
  - You don't have to worry about whether or not the Iterator is still valid.
- The number of nodes $(k + 1)$ never decreases throughout the lifetime of your `LinkedVector<T>`.

- We will not perform any copying of your container, but we will create multiple separate containers.
- You should make your operations as efficient as possible.
  - There is no specific runtime required to complete the assignment.
  - There is not much flexibility in the implementation, but you should be mindful of what can be done efficiently.