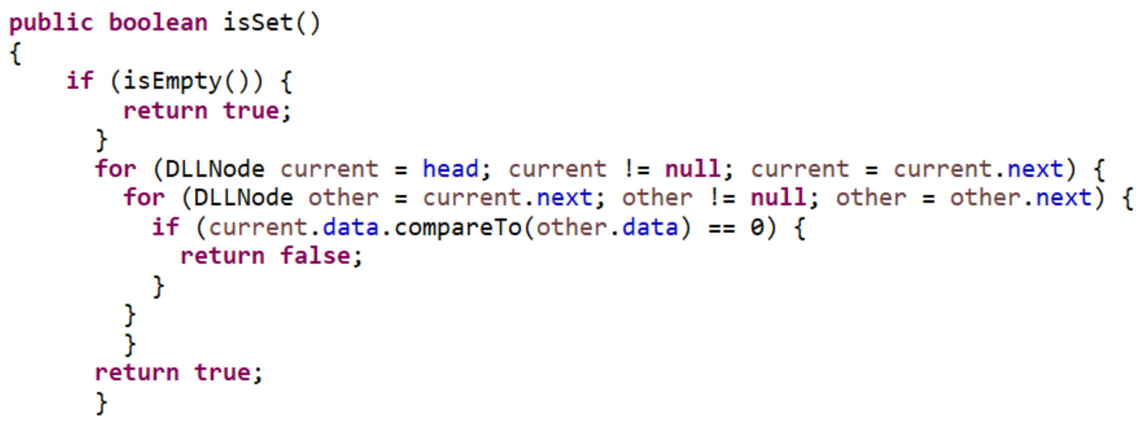
## Part 2: Worst-Case Asymptotic Analysis



```java
public boolean isSet()
{
    if (isEmpty()) {
        return true;
      }
      for (DLLNode current = head; current != null; current = current.next) {
        for (DLLNode other = current.next; other != null; other = other.next) {
          if (current.data.compareTo(other.data) == 0) {
            return false;
          }
        }
        }
      return true;
      }
```

### Asymptotic Analysis

We will perform a worst-case analysis of this algorithm using asymptotic notation. The worst case occurs when all elements in the list are unique (no duplicates), requiring the algorithm to check all pairs of elements before returning true.

Let N be the number of elements in the doubly linked list.

This algorithm consists of the following parts:

Part A: Line 3, with runtime cost T_A(N) = $\Theta(1)$
Part B: Lines 4, with runtime cost T_B(N) outer loop = $\Theta(N)$
Part C: Lines 6-9, with runtime cost T_C(N) inner loop = $N*(N-1)/2 = \Theta(N^2)$
Part D: Line 12, with runtime cost T_D(N) = $\Theta(1)$.

Since these parts are sequentially composed, the overall runtime cost of the algorithm is:

T(N) = T_A(N) + T_B(N) + T_C(N) + T_D(N)

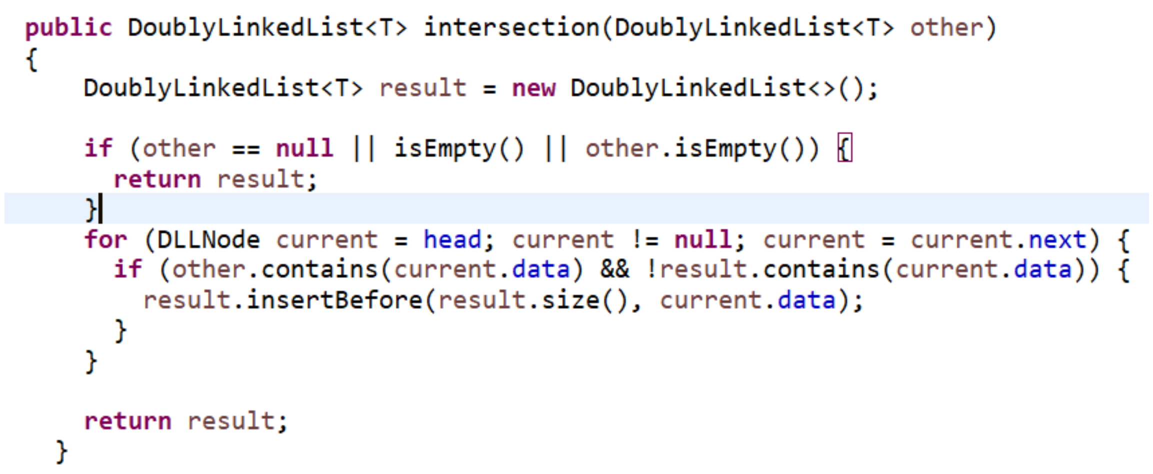
$= \Theta(1) + \Theta(N) + \Theta(N^2) + \Theta(1)$

$= \Theta(N^2)$

Worst-case running time: $\Theta(N^2)$

Memory Cost: $\Theta(1)$

The method (current, other) only uses a constant amount.



```java
public DoublyLinkedList<T> intersection(DoublyLinkedList<T> other)
{
    DoublyLinkedList<T> result = new DoublyLinkedList<>();

    if (other == null || isEmpty() || other.isEmpty()) {
      return result;
    }
    for (DLLNode current = head; current != null; current = current.next) {
      if (other.contains(current.data) && !result.contains(current.data)) {
        result.insertBefore(result.size(), current.data);
      }
    }

    return result;
  }
```

Asymptotic Analysis

We will perform a worst-case analysis of this algorithm using asymptotic notation. The worst case occurs when all elements from the current list are also present in the other list, and all are unique, requiring maximum operations.

Let N be the number of elements in the current list (this list).
Let M be the number of elements in the other list.
Let K be the number of elements in the result list, where $K \le \min(N, M)$.

This algorithm consists of the following parts:

Part A: Line 3, with runtime cost T_A(N, M) = Θ(1).
Part B: Lines 5-7, with runtime cost T_B(N, M) = Θ(1).
Part C: Lines 8-12, with runtime cost T_C(N, M)

Outer loop: Θ(N)
Per iteration i:
- Other: checks all M elements: Θ(M)
- Result: checks i elements: Θ(i)
- size() traverses i elements: Θ(i)
- insertBefore(...) traverses i elements + insert: Θ(i)

Total: Θ(M + i + i + i) = Θ(M + 3i) = Θ(M + i)
Sum over all N iterations (worst case: all N added to result):

$$\sum_{i=0}^{N-1}(M+i) = N \cdot M + \frac{N(N-1)}{2} = \Theta(N \cdot M + N^2)$$

If M = Θ(N), simplifies to $\Theta(N^2)$.

Part D: Line 14, with runtime cost T_D(N, M) = Θ(1).

Overall Time Complexity

$$\begin{aligned} T(N, M) &= T\_A(N, M) + T\_B(N, M) + T\_C(N, M) + T\_D(N, M) \\ &= \Theta(1) + \Theta(1) + \Theta(N \times M + N^2) + \Theta(1) \\ &= \Theta(N \times M + N^2) \end{aligned}$$

When M and N are of the same order ($M = \Theta(N)$): $T(N) = \Theta(N^2)$

Worst-case running time: $\Theta(N \times M + N^2)$, or $\Theta(N^2)$ when $M = \Theta(N)$

Memory Cost

The method creates a new list and up to N nodes in the worst case, each using constant space. Local pointers and method variables also use constant space, so the total memory cost is $\Theta(N)$, which occurs in the worst case when all N elements from the current list are present in the other list ($K = N$).