Lab 2 - Exercise 2

Måns Aronsson (manar189)
Nisse Bergman (nisbe033)

Task 1 - **S1 = S2**

```
// Default constructor
Set::Set() : counter{0}

    head = new Node;
    tail = new Node;
    head->next = tail;
    tail->prev = head;
}
```

```
// Copy constructor
Set::Set(const Set& source)
    : Set{}
{
    Node* ptrSource = source.head->next;
    Node* ptrOrg = head;

    while (ptrSource != source.tail)
    {
        Node* newNode = new Node(ptrSource->value, ptrOrg->next, ptrOrg);

        ptrOrg->next = newNode;
        tail->prev = newNode;
        ptrOrg = ptrOrg->next;
        ptrSource = ptrSource->next;

        ++counter;
    }
}
```

```
// Copy-and-swap assignment operator
Set& Set::operator=(Set source) {

    std::swap(head, source.head);
    std::swap(tail, source.tail);
    std::swap(counter, source.counter);

    return *this;
}
```

```
// Make the set empty
void Set::make_empty() {

    Node* ptr = head->next;

    while (ptr != tail)
    {
        Node* temp = ptr;
        ptr = ptr->next;
        --counter;
        delete temp;
    }

    head->next = tail;
    tail->prev = head;
}
```

```
// Destructor
Set::~Set() {

    make_empty();
    delete head;
    delete tail;
}
```

The number of elements in **S1** is n, and the number of elements in **S2** is m.

The overloaded assignment operator uses the Copy and Swap idiom. A copy of **S2** is created by the copy constructor when the operator is called-by-value.

In the copy constructor, the default constructor is called. Here the operations to create the default Set are of constant time-complexity.

In the copy constructor, the while-loop is called n times for **S2**. In the loop, the constructor of Node is called. However this, and all other operations in the loop, is of constant time.

Because we have lists, only the member variables head, tail and counter need to be swapped for **S2** and **S1.** *std::swap* is of constant time-complexity because **S1** and **S2** and lists.

As the swap has been performed, the destructor is called on the original set of **S1** which now resides in **S2.** The set is destroyed using *make_empty()* and then removes the head and tail node. The while loop in *make_empty()* is called m times and performs constant operations.

With linear complexity for copy constructor and destructor we get:
T(n, m) = n + m = O(max(n, m))

The resulting <u>time complexity</u> for the overloaded assignment operator is therefore O(max(n, m)).

Task 2 - **S1 * S2**

```
   /** Overloaded operator*: Set intersection S1*S2
    *
    * S1*S2 is the Set of elements in both Sets S1 and set S2
    * Return a new Set representing the intersection of S1 with S2, S1*S2
    *
    */
    friend Set operator*(Set S1, const Set& S2) {
        return (S1 *= S2);
    }
```

```
// Modify *this such that it becomes the intersection of *this with Set S
Set& Set::operator*=(const Set& S) {
    Node* ptrOrg = head->next;
    Node* ptrS = S.head->next;

    while (ptrOrg != tail && ptrS != S.tail) {

        if (ptrOrg->value < ptrS->value) {
            ptrOrg->prev->next = ptrOrg->next;
            ptrOrg->next->prev = ptrOrg->prev;

            Node* temp = ptrOrg;
            ptrOrg = ptrOrg->next;

            delete temp;
            --counter;
        }
        else if (ptrOrg->value == ptrS->value) {
            ptrOrg = ptrOrg->next;
            ptrS = ptrS->next;

        }
        else if (ptrOrg->value > ptrS->value) {
            ptrS = ptrS->next;
        }
    }
    while (ptrOrg != tail) {
        ptrOrg->prev->next = ptrOrg->next;
        ptrOrg->next->prev = ptrOrg->prev;

        Node* temp = ptrOrg;
        ptrOrg = ptrOrg->next;

        delete temp;
        --counter;
    }

    return *this;
}
```

The number of elements in **S1** is n, and the number of elements in **S2** is m.

The operator* is overloaded and calls operator*=. In operator*, **S1** is copied which is a linear operation.
In the worst case there are no elements that intersect in **S1** and **S2.** In that case, the while loops will take one step for each element in the two lists. All operations inside of the loops are of constant time O(1).
The time complexity depends on how many times the loop of the function is executed, which in the worst case, together with the copying of **S1,** results in T(n, m) = n + n + m = O(n+m)

The resulting time complexity for the overloaded multiplication assignment operator is therefore O(n+m).

Task 3 - **k + S1**

```
// Conversion constructor
Set::Set(int n)
    : Set{}  // create an empty list
{
    // IMPLEMENT before HA session on week 16
    Node* newNode = new Node(n, tail, head);
    head->next = newNode;
    tail->prev = newNode;
    ++counter;
}
```

```
// Modify *this such that it becomes the union of *this with Set S
// Add to *this all elements in Set S (repeated elements are not allowed)
Set& Set::operator+=(const Set& S) {

    Node* ptrOrg = head->next;
    Node* ptrS = S.head->next;

    while (ptrOrg != tail && ptrS != S.tail) {

        if (ptrOrg->value < ptrS->value) {
            ptrOrg = ptrOrg->next;
        }
        else if (ptrOrg->value == ptrS->value) {
            ptrOrg = ptrOrg->next;
            ptrS = ptrS->next;

        }
        else if (ptrOrg->value > ptrS->value) {
            Node* newNode = new Node(ptrS->value, ptrOrg, ptrOrg->prev);

            ptrOrg->prev->next = newNode;
            ptrOrg->prev = newNode;

            ptrS = ptrS->next;
            ++counter;
        }
      }
    while (ptrS != S.tail) {
        Node* newNode = new Node(ptrS->value, ptrOrg, ptrOrg->prev);

        ptrOrg->prev->next = newNode;
        tail->prev = newNode;

        ptrS = ptrS->next;
        ++counter;
    }

    return *this;
}
```

A conversion constructor will be called for the int variable k, which in turn calls the default constructor. Both these operations are done in constant time.

The addition operator copies the left set and then returns the union of the two sets. Because k is on the left side of the addition operator, copying the set is done in constant time.

The addition assignment operator is then called where all elements (not repeated) of S are added to k. The worst case is when all elements of S are added to k. The first while loop is then executed first time and the second while loop is then executed n times. All the operations within the loops are done in constant time O(1). This gives the Time Complexity T(n) = n + 1 = O(n).

The resulting time complexity for the overloaded addition assignment operator is therefore O(n).