TND004: Lab 2

Joel Paulsson, *joepa811*
William Uddmyr, *wilud321*

April 20, 2020

Exercise 2

Time complexity

To avoid duplicated code two functions, `insert()` and `deleteNode()` was implemented, see Figure 1.

```
//Insert node
void Set::insert(int n, Set::Node* tailNode ) {          O (1)

    Node* newNode = new Node(n, tailNode, tailNode->prev);

    tailNode->prev->next = newNode; // Point the heads next to newNode
    tailNode->prev = newNode; // Point the tails previous to newNode

    counter++;
}

void Set::deleteNode(Set::Node* node){          O (1)

    node->next->prev = node->prev; // Point tail to head
    node->prev->next = node->next; // Point head to tail

    delete node;

    counter--;
}
```

Figure 1: *Private member functions* `insert()` *&* `deleteNode()`.

Case 1: `S1 = S2`
When executing the statement `S1 = S2` a call to the overloaded operator `=` will be made, see Figure 2. This operator make use of the so called copy-and-swap idiom i.e it make use of the existing copy constructor to create a temporary copy using call by value, see Figure 3. The data is swapped using `std::swap()` and the temporary copy is deleted by the destructor. The `std::swap()` have a worst case time complexity $T(n) = \mathcal{O}(n)$ according to the documentation, but in our case it's $\mathcal{O}(1)$.

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

    // Copy-and-swap idiom: Make use of existing copy constructor to create a temp copy (call by value),
    // swap the data and delete the temp copy.

    std::swap(head, source.head);
    std::swap(tail, source.tail);          O(1)
    std::swap(counter, source.counter);

    return *this;
}
```

Figure 2: *The overloaded operator.* `=`.

Hence, the worst case time complexity when executing the statement results in $T(n) = \mathcal{O}(n)$.

```
// Copy constructor
Set::Set(const Set& source)
    : Set{}  // create an empty list
{
    Node* ptr = source.head->next;

    while(ptr != source.tail){          O(n)

        insert(ptr->value, tail);       O(1)

        ptr = ptr->next;
    }
}
```

Figure 3: *The copy constructor.*

Case 2: `S1 * S2`
When executing the statement `S1 * S2` a call to the overloaded operator `*` is made, Figure 4. This operator takes `S1` by-value and `S2` as a reference. Therefore, a call to the copy constructor will be made for the argument `S1`, similar to Case:1, above. The function makes a call to the overloaded operator `*=`, Figure 5. This operator iterates through the sets ($T(n) = \mathcal{O}(n)$) and make comparisons between the values of the left- and right hand side in the body of the loop. It make use of the previous mentioned `deleteNode()` ($T(n) = \mathcal{O}(1)$) to remove unwanted nodes. The resulting time complexity is therefore: $T(n) = \mathcal{O}(n)$, where n is the number of elements in the largest set.

```
friend Set operator*(Set S1, const Set& S2) {
    return (S1 *= S2);
}
```

Figure 4: *Friend declaration of operator *.*

```
Set& Set::operator*=(const Set& S) {

    Node* A = head->next;
    Node* B = S.head->next;

    while (A != tail && B != S.tail) {          O(n)

        if (A->value < B->value) {
            A = A->next;
            deleteNode(A->prev);
        }
        else if (A->value > B->value) {
            B = B->next;
        }
        else {  //if (A->value == B->value) { // values to keep
            A = A->next;
            B = B->next;
        }
    }

    // Remove remaining nodes in A.
    while(A != tail ){
        A = A->next;
        deleteNode(A->prev);
    }
    return *this;
}
```

Figure 5: *Overload operator* `*=`.

Case 3: `k + S1`

Executing the statement `k + S1` is very similar to Case: 2, above, but starts by converting the integer k into a set by calling the conversion constructor, $T(n) = \mathcal{O}(1)$. Thereafter calls are made to the overloaded operators `+` and `+=`, respectively, with the similar performance as `*` and `*=` but using the `insert()` function instead of the `deleteNode()`. And once again the time complexity is $T(n) = \mathcal{O}(n)$, where n is the number of elements in the largest set..

```
friend Set operator+(Set S1, const Set& S2) {
    return (S1 += S2);
}
```

Figure 6: *Friend declaration of operator +* `=`.

```
Set& Set::operator+=(const Set& S) {

    Node* A = head->next;
    Node* B = S.head->next;

    // Insert union
    while (B != S.tail && A != tail) {        O(n)

        if (A->value < B->value) {
            A = A->next;
        }
        else if (A->value > B->value) {

            insert(B->value, A); // Update A
            B = B->next;
        }
        else { //if (B->value == A->value) {
            B = B->next;
            A = A->next;
        }
    }

    //If there is remaining elements in B:
    while (B != S.tail) {        O(n)
        insert(B->value, A);
        B = B->next;
    }

    return *this;
}
```

Figure 7: *Overload operator +* `=`.

Conclusion

A summary of the time complexity analysis of the statements is presented in Table 1.

Table 1: *Results.*

Operation	**Time complexity** $T(n)$
`S1 = S2`	$T(n) = \mathcal{O}(n)$
`S1 * S2`	$T(n) = \mathcal{O}(n)$
`k + S1`	$T(n) = \mathcal{O}(n)$