CS271: Data Structures

Instructor: Dr. Stacey Truex

Project #0

To loosen up those rusty programming fingers and proof logic minds, you will design and implement a `Set` template class in C++ and write some formal set proofs. A set is an *unordered* collection of *unique* elements. For example,

$\{1, 8, 3\}$ is a proper set, but

$\{1, 8, 3, 1\}$ is not.

Submission

This project is meant to be completed in groups. You should work in your Unit 0 groups. Implementation solutions should be written in `C++` and proof solutions should be written in LaTeX. Only one submission (the last submission uploaded to canvas) will be graded per group. Submissions should be a compressed file following the naming convention: `NAMES_cs271_project0.zip` where `NAMES` is replaced by the first initial and last name of each group member. For example, if Dr. Truex and Dr. Lall were in a group they would submit one file titled `STruexALall_cs271_project0.zip`. **You will lose points if you do not follow the course naming convention**. Your `.zip` file should contain a *minimum* of 5 files:

1. `makefile`
2. `set.cpp`
3. `test_set.cpp`
4. `set_proofs.pdf`
5. `commits.pdf`: a commit history for your GitHub project

Additional files such as a `set.h` header file are welcome. The above merely represent the minimum files required for project completion. Details for each are as follows.

Implementation Specifications

Set Class

Implement your `Set` template class with a singly linked list. Elements should be inserted at the head of the list. Your set class should support the following operations:

- insert(x): `s.insert(x)` should insert an element `x` into the set `s`. For example:

```
Set<string> s;
s.insert("hello");
s.insert("world");
```

  should result in the set {`"world"`, `"hello"`}.

- remove(x): `s.remove(x)` should remove an element `x` from the set `s`. For example, using the set above:

```
s.remove("hello");
```

should result in the set {`"world"`}.

- cardinality(): `s.cardinality()` should return the cardinality of (i.e., # of elements in) the set `s`. For example, using the set above:

  ```
  s.cardinality();
  ```

  should return `1`.

- empty(): `s.empty()` should indicate whether `s` is the empty set ($s = \emptyset$). For example, using `s` from above:

  ```
  s.empty();
  ```

  should return `false`.

- contains(`x`): `s.contains(x)` should indicate whether an element `x` is contained in the set `s` ($x \in s$). For example, using the set above:

  ```
  s.contains("world");
  ```

  should return `true`.

- `==` operator: `s == t` should indicate whether `s` contains the same elements as `t` ($s = t$). For example:

  ```
  Set<int> s;
  s.insert(1);
  s.insert(2);

  Set<int> t;
  t.insert(2);
  t.insert(1);

  if(s == t){
      cout << "the sets are equivalent" << endl;
  }
  ```

  should result in the printing of the statement `"the sets are equivalent"`.

- `<=` operator: `s <= t` should indicate whether the set `s` is a subset of the set `t` ($s \subseteq t$). For example, using the set `s` and `t` above:

  ```
  s.insert(3);
  if(s <= t){
      cout << "s is a subset of t" << endl;
  }
  if(t <= s){
      cout << "t is a subset of s" << endl;
  }
  ```

should result in the printing of the statement `"t is a subset of s"`.

- `+` operator: `s + t` should return the union of the set `s` and the set `t` ($s \cup t$). Ordering of elements in the union set should be consistent with elements of the set on the right hand side of the operator (in this case `t`) being inserted into the empty set ($\emptyset$) followed by the elements the set on the left hand side of the operator (in this case `s`). For example, using the sets `s` (`{3, 2, 1}`) and `t` (`{5, 1, 2}`) from above:

```
t.insert(5);
Set<int> u = s + t;
```

  should result in the set `u` = {`3, 2, 1, 5`}.

- `&` operator: `s & t` should return the intersection of the set `s` and the set `t` ($s \cap t$). Ordering of elements in the intersection set should be consistent with inserting elements of the set on the left hand side of the operator into the empty set ($\emptyset$). In this case ∀ `x` ∈ `s`, `x` should be inserted whenever `x` ∈ `t`. For example, using the sets `s` and `t` from above:

```
Set<int> v = s & t;
```

  should result in the set `v` = {`1, 2`}.

- `-` operator: `s - t` should return the difference of the set `s` and the set `t` ($s \setminus t$). Ordering of elements in the difference set should be consistent with inserting elements of the set on the left hand side of the operator into the empty set ($\emptyset$). In this case ∀ `x` ∈ `s`, `x` should be inserted whenever `x` ∉ `t`. For example, using the sets `s` and `t` from above:

```
Set<int> d = s - t;
```

  should result in the set `d` = {`3`}.

Unit Testing

In addition to the functionality above, you are expected to implement a `to_string()` method which returns a string with the elements in your set *separated by a single space* and starting at the head. For example, using the union set `u` generated in the above specifications:

```
cout << u.to_string() << endl;
```

should result in the printing of the string `"3 2 1 5"`. Your `to_string()` method will be required for your class to pass testing.

For each `Set` method included in your `Set` class, write a unit test method in a separate unit test file that *thoroughly* tests that method. Think, in addition to common cases: what are my boundary cases? edge cases? disallowed input? Each method should have *its own* test method. Note that your test file will be run against other class submissions.

An example test file `test_set_example.cpp` has been provided and demonstrates (1) a general outline of what is expected in a test file for this course and (2) a guide on how your projects will be tested after submission. The tests included in `test_set_example.cpp` are not exhaustive. The unit testing in your `test_set.cpp` file should be much more complete. Additionally, for grading purposes, your code will be put through significantly more thorough testing than what is represented by `test_set_example.cpp`. Passing the tests in this example file should be viewed as a lower bound. **Do not change the includes of the `test_set_example.cpp` file**. These match the file that will be used to test your project.

Documentation

The expectation of all coding assignments in this course is that they are well-documented. This means that logic is documented with line comments and method pre- and post- conditions are properly documented immediately after the method's parameter list.

Pre-conditions and post-conditions are used to specify precisely what a method does. However, a pre-condition/post-condition specification does not indicate how that method accomplishes its task (if such commenting is necessary it should be done through line level comments). Instead, pre-conditions indicate what must be true before the method is called while the post-condition indicates what will be true when the method is finished.

Makefile

With each project you should be submitting a corresponding makefile. Once unpacking your `.zip` file, the single command `make` should create a `test` executable. The command `./test` should then run all the unit tests in your `test_set.cpp` file evaluating your `Set` class.

Efficiency

Each project in this course will additionally be evaluated for efficiency. Each method detailed in the `Set` class specifications of this document will be called 1 time using a very large example. The total time to execute all methods will be clocked.

Proofs

For each of the following complete a formal proof.

- Prove inductively that a set S with cardinality $n \geq 1$ has exactly 2^n unique subsets.
- Prove inductively that a set S with cardinality $n \geq 2$ has exactly $\frac{n \cdot (n-1)}{2}$ unique subsets of cardinality 2.
- Prove inductively that the complement of the union of any n sets $S_1, S_2, ..., S_n$ is equivalent to the intersection of each of their individual complements (i.e., that $\overline{S_1 \cup S_2 \cup \cdots \cup S_n} = \overline{S_1} \cap \overline{S_2} \cap \cdots \cap \overline{S_n}$) for all $n \geq 1$. Hint: it may be helpful to remember De Morgan's Law:

$$\overline{S \cup T} = \overline{S} \cap \overline{T}$$

- Prove by contradiction that the intersection of any set S_1 with the difference of any set S_2 and S_1 is the empty set (i.e., $S_1 \cap (S_2 \setminus S_1) = \emptyset$).

Rubric

Note that any coding projects that do not compile with the provided `test_set_example.cpp` file will be given a 0. All projects that are able to be successfully compiled will be graded using the following rubric.

<table>
<tr><td rowspan="20">C++ Implementation</td><td colspan="3">does not compile: 0/40</td></tr>
<tr><td colspan="3">40 Total Points</td></tr>
<tr><td rowspan="3">Code</td><td>Completeness
met submission requirements</td><td>10 pts</td></tr>
<tr><td>Correctness
passes unit testing</td><td>22 pts</td></tr>
<tr><td>Validation
implementation deductions
ex: set not implemented as a singly linked list</td><td>——</td></tr>
<tr><td rowspan="5">Efficiency</td><td>Time Test</td><td>2 pts</td></tr>
<tr><td>encountered error - could not complete time test</td><td>0/2</td></tr>
<tr><td>takes over 2x fastest submission</td><td>1/2</td></tr>
<tr><td>within 2x fastest submission</td><td>2/2</td></tr>
<tr><td>fastest submission</td><td>3/2</td></tr>
<tr><td rowspan="5">Documentation</td><td>Documentation</td><td>3 pts</td></tr>
<tr><td>extremely sparse documentation</td><td>0/3</td></tr>
<tr><td>missing comments or pre- and post-conditions</td><td>1/3</td></tr>
<tr><td>documentation lacks detail in areas</td><td>2/3</td></tr>
<tr><td>detailed comments & pre- and post-conditions</td><td>3/3</td></tr>
<tr><td rowspan="5">Testing</td><td>Unit tests</td><td>3 pts</td></tr>
<tr><td>does not expand on example test file</td><td>0/3</td></tr>
<tr><td>not all functions tested or
testing not implemented as unit testing or
no variation in templates</td><td>1/3</td></tr>
<tr><td>caught some of the bugs in classmates' code</td><td>2/3</td></tr>
<tr><td>caught most bugs in classmates' code</td><td>3/3</td></tr>
</table>

<table>
<tr><td rowspan="9">LaTeX Proof PDF</td><td colspan="3">6 Total Proof Points</td></tr>
<tr><td rowspan="5">1 randomly selected proof</td><td>Correctness</td><td>3 pts</td></tr>
<tr><td>incomplete, barebones, or missing</td><td>0/3</td></tr>
<tr><td>demonstrates significant error in understanding
either proof logic or underlying concept</td><td>1/3</td></tr>
<tr><td>errors in writing or small error in logic</td><td>2/3</td></tr>
<tr><td>well-written, complete proof</td><td>3/3</td></tr>
<tr><td rowspan="3">other 3 proofs</td><td>Completeness</td><td>1pt each</td></tr>
<tr><td>incomplete, barebones, or missing</td><td>0/1</td></tr>
<tr><td>valid attempt to complete proof</td><td>1/1</td></tr>
</table>