Lab 2

Course goals

- To implement and use data structures and algorithms in application programs.
    - To implement a dynamic data structure, doubly linked list, using concepts such as C++ classes with overloaded operators and dynamic memory allocation
- To analyze and evaluate various data structures in solving computational problems with respect to efficiency and appropriateness

Preparation

Please ensure you read the entire lab description thoroughly, before starting any exercises, including the preparation steps given below. Pay special attention to the code requirements section.

Completing the following tasks before the Lab2 HA session will help you make the most of your lab time

- Download the files for this exercise from the course website. Like lab 1, you can then use CMake to create a project for this lab.
- Compile, link, and execute the program. The first assertion should fail.
- Review lectures 2 to 4. Big-O notation was introduced in lecture 2 and lecture 3, while doubly linked lists where discussed in lecture 4.
- Study the class `Set` interface given in the file `set.h`.
- Some member functions of class `Set` are explicitly marked as "`// IMPLEMENT before Lab2 HA`", see files `set.h` and `set.cpp`. At least, these functions should be implemented before the HA lab session.

  You can test your code with the program given in `lab2.cpp`. Your code should pass the tests in phase 0 to phase 6, before HA lab session. The expected output is shown here.
- Review exercise 3 of set of exercises 1, in the TNG033 course[1]. The exercise introduces an algorithm that can also be used to implement union of two sets (i.e. member function `Set::operator+=`) in linear time. Later in the course, we will use this algorithm again.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "`TND004: ...`".

A class to represent sets using doubly linked lists

In the TNG033 course, there was a lab about implementing a singly linked list to represent sets of integers. In this lab exercise, you go further and implement a class `Set` using instead **sorted doubly linked lists**. Additionally, it is required that all `Set`

[1] Use login is **`TNG033`** and password is **`TNG033ht13`**.

functions have **linear time complexity, in the worst-case**. Therefore, you should be careful when re-using code from your lab solution in TNG033 for this exercise, since TNG033 did not require each `Set` operation to execute in linear time[2].

Course book

Section 3.5 of the course book presents a possible implementation for doubly linked lists. Though it is also required to implement doubly linked lists in this lab, there are some important differences between the implementation required here and the book's implementation, as summarized below. Most of the differences are motivated by the fact that we are going to use doubly linked lists to implement the concept of (mathematical) set. Nevertheless, studying the implementation presented in section 3.5 of the course book is recommended.

- The book presents an implementation of doubly linked lists, while in this lab you are requested to implement the concept of set by using a doubly linked list (though, sets can be implemented in other ways).
- In this lab exercise, the list's nodes are placed in sorted order and there are no repeated values stored in the list. On the contrary, the course book's implementation allows repeated values in lists and lists do not need to be sorted.
- The book uses a template class to implement a generic class `List` of objects. Template classes are not used in this lab.
- Iterator classes are not considered in this exercise, though they are implemented in the course book.
- Move constructor and move assignment operator are part of the book's implementation (lines 26 to 41 of figure 3.16 of course's book). These are not part of the lab.
- The implementation of class `Node` provided with this lab maintains a counter of the total number of existing nodes (a static data member called `count_nodes`).
- This lab follows C++20 approach with respect to overloading comparison operators.

Counting nodes and assertions

A (static) member function named `get_count_nodes` of class `Set` returns the total number of existing nodes. This function is used in the test code to help detecting possible memory leaks by using assertions.

Similar to lab 1, assertions are used to help test your code. A test program is given in the `main` function (see `lab2.cpp`). For instance, the assertion

```
assert(Set::get_count_nodes() == 2);
```

tests whether the total number of existing nodes is equal to 2. If not then the program stops running and a message is displayed with information about the failed assertion (see lab 1, section "*Testing the code: assertions*").

[2] In other words, your code for the `Set` class in TNG033 course may not satisfy the linear time requirement of this lab and, consequently, cannot be simply re-used.

Neither the data member `count_nodes` nor function `get_count_nodes` should be used in the member functions implementations unless you want to add some extra tests.

Exercise 1: class `Set`

In this exercise, you need to implement the class `Set` that represents sets of integers (`int`). Sorted doubly linked list is the data structure used to implement sets in this lab. Notice that sets do not have repeated elements.

Every node of the list stores an integer value. To make it easier to remove and insert an element from the list, the list's implementation uses "dummy" nodes at the head and tail of the list, as discussed in lecture 4 and in the course book. Thus, an empty doubly linked list consists of two nodes pointing at each other (see also figure 3.10 of course book).

The class `Set` provides the usual set operations like set union, difference, subset test, and so on. The class definition is given in the file `set.h`. Extra information about the meaning of the overloaded operators is given below.

- Overloaded `operator==` such that `R == S` returns `true`, if R and S represent the same set (i.e. both sets have the same elements). Otherwise, `false` is returned.
- Overloaded tree-way comparison `operator<=>`. To this end, read the introduction to C++20 comparison operators given in the appendix.
- Overloaded `operator+=` such that `R+= S;` should be equivalent to $R = R \cup S$, i.e. the *union* of R and S is assigned to set R. Recall that the union $R \cup S$ is the set of elements in set R or in set S (without repeated elements). For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R \cup S = \{1, 2, 3, 4\}$.

  Union of sets is conceptually like the problem of merging two sorted sequences. An algorithm to merge sorted sequences efficiently was discussed in the TNG033 course[3]: see solution for set of exercises 1, exercise 3. Though vectors are used in exercise 3 of set of exercises 1, TNG033, **your solution must not copy the lists to vectors**. Instead, your solution must implement the same algorithm directly with the lists (instead of vectors).
- Overloaded `operator*=` such that `R*= S;` should be equivalent to $R = R \cap S$, i.e. the *intersection* of R and S is assigned to set R. Recall that the intersection $R \cap S$ is the set of elements in both R and S. For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R \cap S = \{1, 4\}$.
- Overloaded `operator-=` such that `R-= S;` should be equivalent to $R = R - S$, i.e. the *set difference* of R and S is assigned to set R. Recall that the set difference $R - S$ is the set of elements that belong to R but do not belong to S. For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R - S = \{3\}$ [4].

The file `lab2.cpp` contains a test program for class `Set`. Feel free to add other tests to this file, though the original file must be used when presenting the lab.

[3] Use login is `TNG033` and password is `TNG033ht13`.

[4] The set difference $R - S$ is also known in the literature as the "*relative complement of S in R*".

Code requirements

- All `Set` functions must have a linear time complexity, in the worst-case. You should also strive to implement set member functions by just traversing each list at most once.
- The comparison operators should iterate through each set no more than once.
- Do not spread the use of `new` and `delete` all over the code. Instead, define two private member functions which call `new` and `delete`.

```
// Insert a new Node storing v after the Node pointed by p
void insert_node(Node* p, int v);

// Remove the Node pointed by p
void remove_node(Node* p);
```

  Class `Set` member functions should call the private member functions above, whenever possible, to add or remove a node[5].
- The public class Set interface (given in `set.h`) cannot be modified. Other member functions can be added to class `Set` as private member functions, though.
- Finally, STL containers cannot be used to implement the member functions[6].

Exercise 2: time complexity analysis

This exercise consists in analysing the time complexity for each of the following statements. Use Big-Oh notation and write a **clear motivation** for your analysis. Assume that `S1` and `S2` are two `Set` variables and that `k` is an `int` variable. Assume also that `S1` has $n_1 > 0$ elements and that `S2` has $n_2 > 0$ elements.

- `S1 = S2;`
- `S1 * S2`
- `k + S1`

Make sure that the mathematical functions used to express time complexity clearly relate to variables n_1 and n_2.

Presenting lab and deadlines

The exercises in this lab are compulsory and you should demonstrate your solutions during the lab session *Lab2 RE*. Read the instructions given in the labs webpage and consult the course schedule.

Necessary requirements for approving your lab are given below.

- The code must be readable, well-indented, and use good programming practices. Note that complicated functions and over-repeated code make code quite unreadable and prone to bugs.

[5] Possible exception can be when creating or destroying a list's dummy nodes in the constructors and destructor, respectively.
[6] An exception applies to the `Set` class constructor, which creates a set using elements from an input `std::vector`.

- Compiler warnings that may affect badly the program execution are not accepted, though the code may pass the given tests.
- There are no memory leaks neither other memory related bugs. On the labs webpage you can find a list of tools that can help to check for memory related problems in the code.
- Your code must satisfy the given code requirements.
- Hand in your written answer for exercise 2 to your lab assistant during *Lab 2 RE* and be prepared to discuss it with the lab assistant. Do not forget to indicate the name plus LiU-id of each group member. Unreadable answers will be rejected.

If your code for lab 2 has not been approved in the scheduled lab session *Lab2 RE* then it is considered a late lab. Late labs can be presented provided there is time in a another RE lab session. All groups have the possibility to present one late lab on the extra RE lab session scheduled in the end of the course.

Expected output

```
TEST PHASE 0: default and conversion constructor

TEST PHASE 1: constructor from a vector

TEST PHASE 2: copy constructor

TEST PHASE 3: operator=

TEST PHASE 4: is_member

TEST PHASE 5: cardinality and is_empty

TEST PHASE 6: equality and <=>

TEST PHASE 7: operator+=, operator*=, operator-=

TEST PHASE 8: union, intersection, and difference

TEST PHASE 9: mixed-mode arithmetic
Success!!
```

Appendix

Comparison operators in C++20

C++20 has introduced new rules concerning overloading comparison operators such as `operator==`, `operator!=`, `operator<`, `operator<=`, etc. With these new rules, for most cases, it's only required to overload two comparison operators, `operator==` and `operator<=>`, so that a class supports all six comparison operators. In addition, extra overloads are not required to support mixed-mode comparison, either. Thus, a lot of the boilerplate code that comparison operators introduced until C++20 has now disappeared.

In C++20, if an `operator==` is available for a class then `a != b` is automatically evaluated as `!a.operator==(b)`. In addition, comparison operator calls can be reversed, automatically. For example, `9 == a` evaluates to `a.operator==(9)` (thus, taking advantage of possible automatic conversion from `int` to the type of object `a`). Thus, C++20 understands that equality is symmetric.

A new operator, `operator<=>` (named three-way comparison operator or operator spaceship) has been added in C++20. This operator should return instances of one of the three comparison category classes defined in the library `compare`. These categories are: `std::strong_ordering` (used when all pairs of instances are comparable), `std::weak_ordering` (used when equality only defines an equivalence class), and `std::partial_ordering` (used when a pair of instances `a` and `b` can be incomparable). For instance, `operator<=>` should return instances of `std::strong_ordering` for `int`s, while `operator<=>` should return instances of `std::partial_ordering` for the class `Set` in this exercise. Consider a set `S1`$= \{1, 2, 8\}$ and a set `S2`$= \{3, 9\}$. Then, neither `S1` $<$ `S2` (`S1` is not contained in `S2`) nor `S2` $<$ `S1` (`S2` is not contained in `S1`), i.e. `S1` and `S2` are simply not comparable.

More concretely, for class `Set`, the tree-way comparison `S1<=>S2` should return:

- `std::partial_ordering::equivalent`, if `S1` and `S2` represent the same set;
- `std::partial_ordering::less`, if `S1` $<$ `S2`, i.e. all elements of set `S1` also belong to set `S2` but `S1` and `S2` are not the same set;
- `std::partial_ordering::greater`, if `S2` $<$ `S1`;
- `std::partial_ordering::unordered`, otherwise (i.e. sets `S1` and `S2` are incomparable sets).

If the three-way comparison operator is defined for a class, then it automatically gives support for all four comparison operators, `<`, `<=`, `>`, `>=` (i.e. no need to write code for these operators). For instance, `20 < S1` also compiles (assuming there is a conversion from `int` to `Set`) meaning that 20 is an element of set `S1`, i.e. `S1` $> \{20\}$.

This is just a brief introduction to comparison operators in C++20. You can find more details on many websites and C++20 books.

Memory monitoring tools

C++ programs can be affected by bugs that corrupt memory such as buffer overflows, accesses to a dangling pointer (use-after-free), or memory leaks. It is important to detect this type of bug and eliminate them, though it may not be a trivial task.

Specific memory monitoring software tools can help the programmers to find out if their programs suffer from memory corruption bugs. There are several tools which you can install and try for free. Note that no tool will detect all memory bugs in the code (i.e. all tools have limitations). Thus, using several tools and inspecting carefully the code is the best strategy. Some of these tools are listed below. Take the opportunity and try some of them.

- AddressSanitizer (ASan) is a tool to detect memory related problems in the code, though it cannot detect memory leaks on Windows. In this document, you can find information on how to use Asan, when compiling and linking with clang, from the command line on Windows.
- ASan is also supported by Microsoft C/C++ compiler (MSVC). If you have created the Visual Studio project/solution using the CMake file provided with this lab then the address sanitizer is already enabled. Otherwise, you need to enable it manually. Note that Asan on Windows does not detect memory leaks.
- Valgrind available for Linux.