Linköping University
Department of Science and Technology/ITN
Aida Nordman

TND004: Data Structures
Lab 2

Goals

- To implement a dynamic data structure, doubly-linked list.
- To use pointers.
- To use big-Oh notation to analyse algorithms in terms of how their running time or space requirements grow as the input size grows.
- To use C++ classes and overloaded operators.

Preparation

You must perform the tasks listed below before the start of the lab session *Lab2 HA*.

- Download the files for this exercise from the course website. Similar to 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-Oh notation was introduced in lecture 2 and lecture 3, while doubly linked lists where discussed in lecture 4.
- Read the section "*A class to represent sets using doubly-linked lists*" of this lab description.
- Study the class `Set` interface given in the file `set.h`.
- Some member functions of class `Set` are explicitly marked as "`// IMPLEMENT before Lab1 HA`", see files `set.h` and `set.cpp`. At least, these functions must be implemented before the **HA** lab session. Note that all `Set` functions must have linear time complexity with respect to the size of the input lists. You can test your code with the program given in `lab2.cpp`.
- Review exercise 3 of set 1 of exercises, in the TNG033 course[1]. This exercise of TNG033 course 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.
- Read the section "Presenting lab and deadlines".

Other member functions can be added to class `Set`, besides the given ones. In this case, the extra added functions should not belong to the public interface of the class (i.e. they should be private member functions).

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: ...`".

[1] Login is **`TNG033`** and password is **`TNG033ht13`**.

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` functions have **linear time complexity**, in the worst-case.

Section 3.5 of the course book presents a possible implementation for doubly-linked lists. Although in this lab you also need to implement doubly-linked lists, there are some important differences between the implementation required in this exercise 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`).

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 through the use of assertions.

Similar to lab 1, assertions are used to help testing 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*").

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;` 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 similar to the problem of merging two sorted sequences. An algorithm to merge sorted sequences efficiently was discussed in the TNG033 course[2] (see solution for set 1 of exercises, exercise 3).

- 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 $R - S = \{3\}$ [3].

- Overloaded `operator<=` such that `R <= S` returns `true` if R is a *subset* of S. Otherwise, `false` is returned. Set R is a subset of S if and only if every member of R is a member of S. For instance, if $R = \{1, 8\}$ and $S = \{1, 2, 8, 10\}$ then R is a subset of S (i.e. `R <= S` is true), while S is not a subset of R, (i.e. `R <= S` is false).

- Overloaded `operator<` such that `R < S` returns `true`, if R is a *proper subset* of S. A set R is a proper subset of a set S if R is strictly contained in S and so necessarily excludes at least one member of S. For instance, `S < S` always evaluates to `false`, for any set S. **Hint:** use `operator<=` to implement this function.

- Overloaded `operator==` such that `R == S` returns `true`, if R is a subset of S and S is a subset of R **(**i.e. both sets have the same elements). Otherwise, `false` is returned. **Hint**: use `operator<=` to implement this function.

- Overloaded `operator!=` such that `R != S` returns `true`, if R (S) has an element that does not belong to set S (R), i.e. R and S have different elements.

Note that all `Set` functions must have a linear time complexity.

---

[2] Login is `TNG033` and password is `TNG033ht13`.

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

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.

Other member functions can be added to class `Set`, besides the ones given in `set.h`. In this case, the extra added functions should not belong to the public interface of the class.

Code requirements

Do not spread the use of `new` and `delete` all over the code. Instead, define two private member functions.

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

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

Thus, operations `new` and `delete` must only appear in the functions above. Class `Set` member functions should call the private member functions above, whenever it's needed to allocate memory for a new node or deallocate a node's memory.

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 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 functions used to express the 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.

- Use of global variables is not allowed, but global constants are accepted.
- 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. Perform code refactoring whenever possible.
- 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.
- All `Set` functions must be executed in linear time, in the worst-case.
- STL containers cannot be used in this exercise[4].
- Bring your written answer for exercise 2 and be prepared to discuss it with the lab assistant.

If your solution 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.

[4] Exception applies to the class `Set` constructor that builds a set with elements given in a `std::vector`.

Appendix

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 bugs 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 is a number of 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.

- DrMemory is available for Windows, Linux, and Mac. DrMemory should be used with programs compiled with clang, as briefly described here. Note that DrMemory does not produce reliable memory diagnostics for programs compiled in Visual Studio (MVSC compiler).
- AddressSanitizer (ASan) is another tool to detect memory related problems in the code, though it cannot detect memory leaks on Windows (nor in Xcode). You can read here how to use Asan, when compiling and linking with clang, from the command line on Windows.
- Valgrind available for Linux.
- Visual leak detector for Visual C++ is a library, named `vld.h`, that can be included in C++ programs. It's easy to use and install. The downside is that it only detects memory leaks and it can only be used with programs built in Visual Studio.