C++ project report February 2022

Luca Poli [852027]

l.poli6@campus.unimib.it

Introduction

A set is a dynamic data structure where duplicates are not allowed, which is read-only accessible and editable through appropriate operations.

To achieve dynamism, I chose to use a doubly concatenated list, reached with a pointer at the beginning and one at the end; this structure is based on the node, that is, the internal element that makes it up.

Why specifically a list and not a dynamic array?

The main operations on a set will be the total read, insertion or deletion of an element; a list allows these operations in similar time frames as an array, but does not present the problem of reaching the maximum size. In that case, it will have to be expanded by copying it into a larger array, but losing efficiency.

The only disadvantage of using a list is in random access to a node (done with the [] operator).

Internal data structure: The node

We can say that a list is a series of nodes concatenated together, where each node stores a single value. A node contains three pieces of data: the value, a pointer to the next node, and a pointer to the previous node. I chose a double concatenation to simplify queuing and to allow the use of a bidirectional iterator.

The node has a basic implementation, characterized by the base constructor, the copy constructor, the assignment operator, and a number of secondary constructors for each eventuality.

Data structure: The set

The structure itself

So the set is represented by a start and end pointer to the queue of nodes, the a variable that tracks its size, and a functor used to determine when two elements are equal

The fundamental methods of each class

Like any class we need some basic methods such as constructors, destructors, operators and other methods to read internal parameters.

Of builders we have:

- the default constructor that initializes the empty structure,
- the copy-constructor that initializes the set by copying each value from the other set entirely (it does not just "link" it),
- two secondary constructors that insert into the set of elements taken from a static array or a pair of iterators, respectively, discarding duplicate elements.

The destructor simply invokes a dedicated method that walks the list backward de allocating each node.

Regarding operators we have:

- the assignment that creates a copy (by value as the copy-constructor) of the passed set and assigns the copy to itself
- the comparison (==) that scrolls through the list of the set and for each element looks for whether it repeats in the other set
- The random access operator [] that scrolls through the set list until it reaches the correct index
- the print operator, which scrolls through the list and prints each

item Finally, we have other methods:

- Of search, including a private one that scrolls the list until finding the element (using the functor) and returning its pointer to the node; and a public one that simply calls the previous one and returns a boolean
- Others to get the size of the set

The set-specific methods

They are add (to add an item) and remove (to remove it), respectively:

- Add: first checks for the presence of the value and if so, throws the appropriate exception (explained in more detail later) and terminates; if not, it proceeds with queuing (position is indifferent, however, queuing is preferred to give a greater sense of chronological order)
  - Verification is done using the private search method
  - Queuing is accomplished as in any double list with final pointer
- Remove: first it searches for the value and if it is absent it throws the appropriate exception (explained better later) and terminates; otherwise it performs its deletion
  - Verification is done using the private search method
  - The deletion proceeds as a normal node deletion in a double list

The generic methods

They are:

- The filter_out: which uses a pair of iterators to locate elements that satisfy the templated property in an additional functor
    - Of the functor it is necessary to implement the operator () that specifies the property
- Concatenation (+): copies the first set to the output set; and uses a pair of iterators on the second set to insert each element into the copied set
- Intersection (-): uses a pair of iterators on the first set to insert each element, which is present in the second, into the output set

The three generic methods, by choice, return a pointer to a new set; ceding the responsibility of de allocation to the caller, this is because returning with a copy would have been too onerous in terms of efficiency.

Iterator

To ensure a greater degree of freedom of use, I chose to implement a bidirectional iterator; it consists of two pointers to nodes (pnode, backnode), the first pointing to the current node, the second pointing to the node read at the previous iteration.

These two variables denote 3 logical states of the iterator:

- Normal state: it can be moved both forward and backward; thus, increment and decrement operations are accomplished by moving the backnode to pnode, and moving pnode to the next (or previous in the case of decrement)
- End state: the iterator is in an end state, when pnode has passed either end, it is thus out of the list; however, it is still possible to re-enter via the reverse operation (increment if in head-1, decrement if in tail+1), through the help of backnode.
- Dead state: the iterator is in an error state; this situation is reached when using the iterator in End state with the incorrect operation (decrement if in head-1, increment if in tail+1).
    - Note: I chose at the design level to allow the incorrect operation, so the responsibility remains with the user.
        - (just as it is possible to increment a unidirectional end iterator it must be possible to do so with the bidirectional one)

Two iterators are defined as equal when they point to the same node (the two pnodes match)

The pair of iterators (begin, end) corresponds to an iterator placed at the head of the list, and an iterator placed in the end state

Exception handling

As far as exceptions are concerned, I used two types; the first does parate from the standard library is the out_of_range exception and is thrown when an incorrect index is used in the [] operator.

Instead, the second type was defined by me, is called set_element_error and is derived from the standard domain_error exception.

It has two constructors, the first allowing its general use as if it were a standard exception; the second allows its use specific to the set case, through the involvement of a boolean. It allows the distinction between a presence error (used in add) and an absence error (used in remove) and returns the corresponding standard error message.

Main_test

In this file, the main features of the set are tested on 3 types of data: integers, strings, and entries (from an address book); in fact, in the first part of the file the functors and some default data that will be used in the tests are defined.

In the second part, 3 methods are defined to separate the three categories of tests:

- Fundamental tests: in which we test the copy constructor, add, remove, find operations, operators, and prints by method and iterator
- Generic tests: in which we test the 3 generic methods (filter_out, operator+, operator-)
- Iterator testing: in which we test the specific behavior of iterators in end-state
    - Note: Part of the code is commented out because it generates the expected and wanted crashes with such instructions

Finally in the third part, that is, in the main itself, you create the sets and call the three methods defined earlier.