DM852
Introduction to Generic Programming

Spring 2022

Assignment 2

This is the second, effectively mandatory, assignment in DM852.

1 Formalities

1.1 Working Together?

The assignment must be completed individually and you are not allowed share code with each other. For higher level questions, and general programming issues, it is ok to discuss with other course participants, but not to an extend where you are discussing details of code for the assignment.

1.2 Submission

The assignment consists only of implementation work, though you must provide some documentation and notes in the header files as described. On the IMADA Git server you have a dedicated exam repository, called "Repo_exam_⟨*username*⟩", which must be used for submitting. You must place code in the exact files specified below, as your submission will be evaluated in part automatically.

The deadline for submitting is

Monday, 16 May, at 13:00 CEST

Whatever is in your exam repository at the deadline is considered your submission.

2 Tasks

The assignment consists of implementing 2 of the basic data structures: a linked list and a binary search tree. The purpose is to train skills and competences in implementation of generic data structures in C++. Your code must be valid C++20 (i.e., compile with `-std=c++20`, or `-std=c++2a` on GCC < 10). You are allowed to use the C++ standard library, as long as underlying intend of the assignment is followed. Please ask if you are in doubt if it is ok to use a particular standard library feature.

You should test your implementation your self, and all your test files should be put into `asg2/test/`. It may be useful to create a Makefile to compile and run all your tests easily. Remember that you can compile with sanitizers or run programs through Valgrind to find many problems.

In addition, every time you push to your exam repository a series of automated tests are run. You can see the results of them at `https://dalila.imada.sdu.dk` after logging in with your SDU credentials.[1] You should aim to make all tests succeed, but note that they do not cover all aspects of the assignment, and their status therefore does not alone determine how your submission is evaluated. The automated tests are of experimental nature, so additional checks may be added. You should not rely on these tests as your primary means of testing, only as a supplement. Please do contact the lecturer if you find errors or misleading results in them, or in general if you have problems regarding the Git server and the testing server.

2.1 A Generic Doubly Linked List

Files: `asg2/src/List.hpp`
Implement a doubly linked list as a class template `List` in the namespace `DM852`. The class template must have a single parameter, which in the following is assumed to be called `T`.
The class template should fulfil the following requirements:

- It must store elements of type `T`.
- It must have a nested type `value_type = T`.
- It must have two nested class types `iterator` and `const_iterator`.
  - Both must be models of the BidirectionalIterator concept (see lecture slides).
  - Both must have a nested type `value_type = T`.
  - `iterator` must have a nested type `reference = T&`, which must be the return type of its `operator*`.
  - `const_iterator` must have a nested type `reference = const T&`, which must be the return type of its `operator*`.
  - Note the requirement of decrementability of the past-the-end iterator, see `end()`.

[1]See also the general note on automated testing in the course, available on itslearning.

  - An **`iterator`** must be convertible to a **`const_iterator`** (but not the other way around).

- It must be a Regular type, i.e., be DefaultConstructible, Copyable, and EqualityComparable.

- It must also implement a move constructor and move assignment operator.

- It must additionally have the following member functions. Some of them have pre-conditions, which you must document as a comment at the function declaration in the header file. Some member functions also requires the element type to model certain concepts, and those requirements must be documented as well. If a member function does not run in constant time (ignoring memory allocation), you must document the computational complexity. If you have considered an alternative implementation to achieve a different computational complexity, write a short note about it as well.

  - `int size() const;`
    Returns the number of elements stored.
    Must return in constant time.

  - `bool empty() const;`
    Returns `true` iff the container is empty. I.e., `empty() == (size() == 0)`.
    Must return in constant time.

  - `void push_back(const value_type &elem);`
    `void push_back(value_type &&elem);`
    Inserts a new element in the end, either by copy or by move.

  - `iterator insert(const_iterator pos, const value_type &elem);`
    `iterator insert(const_iterator pos, value_type &&elem);`
    Inserts a new element before the one stored at `pos`, and returns an iterator to the node for the newly inserted element. May not invalidate any iterators.

  - `void clear();`
    Erase all elements.

  - `void pop_back();`
    Erase the last element.

  - `void erase(const_iterator pos);`
    Erase the element stored at `pos`.

  - `value_type &front();`
    `const value_type &front() const;`
    Two overloads for accessing the first element.
    Must return in constant time.

- `value_type &back();`
  `const value_type &back() const;`
  Two overloads for accessing the last element.
  Must return in constant time.
- `iterator begin();`
  `const_iterator begin() const;`
  Two overloads for accessing the node storing the first element.
  Returns a past-the-end iterator if the container is empty.
  Must return in constant time.
- `iterator end();`
  `const_iterator end() const;`
  Two overloads that returns a past-the-end iterator.
  Must return in constant time.
  Note that if the container is not empty, then it should be possible to decrement this iterator in order to access the last element of the container. That is, `back()` is equivalent to `*--end()`.

2.2 A Generic Binary Search Tree Mapping Keys to Values

Files: `asg2/src/Tree.hpp`
Implement a binary search tree as a class template `Tree` in the namespace `DM852`. The class template must have three parameters:

- `Key`
- `Value`
- `Comp`, which must default to `std::less<Key>`. An object `comp` of type `Comp` must be a binary predicate, that conceptually generalizes the <-relation. Mathematically this means it must induce a so-called strict weak ordering[2]

  Practically, for two `const Key` objects `a` and `b`, the expression `comp(a, b)` must be valid, and return `bool`. The return value must be `true` if `a` is to be considered ordered before `b`. The two keys are thus equal if `!comp(a, b) && !comp(b, a)`.

Thus, the declaration of the class template must look similar to the following

```
template<typename Key, typename Value, typename Comp = std::less<Key>
struct Tree;
```

The class template should fulfil the following requirements:

- The tree must be an associative data structure that maps `Key`s to `Value`s. The relative ordering of keys is determined by the comparator of type `Comp`.

[2] `https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings`

- It must have a nested type `value_type = std::pair<const Key, Value>`.
- It must have two nested class types `iterator` and `const_iterator`.
  - Both must be models of the BidirectionalIterator concept (see lecture slides).
  - Both must have a nested type `value_type = std::pair<const Key, Value>`.
  - `iterator` must have a nested type `reference = std::pair<const Key, Value>&`, which must be the return type of its `operator*`.
  - `const_iterator` must have a nested type `reference = const std::pair<const Key, Value>&`, which must be the return type of its `operator*`.
  - Note the requirement of decrementability of the past-the-end iterator, see `end()`.
  - An `iterator` must be convertible to a `const_iterator` (but not the other way around).
- It must be a Regular type, i.e., be DefaultConstructible, Copyable, and EqualityComparable.
- It must also implement a move constructor and a move assignment operator.
- It must also have a constructor `Tree(Comp comp)`, for constructing the tree with a specific comparator instance (the default constructor must default construct a comparator).
- It must additionally have the following member functions. Some of them have pre-conditions, which you must document as a comment at the function declaration in the header file. Some member functions also requires the element type to model certain concepts, and those requirements must be documented as well. If a member function does not run in constant time (ignoring memory allocation), you must document the computational complexity. If you have considered an alternative implementation to achieve a different computational complexity, write a short note about it as well.
  - `int size() const;`
    Returns the number of elements stored.
    Must return in constant time.
  - `bool empty() const;`
    Returns `true` iff the container is empty. I.e., `empty() == (size() == 0)`.
    Must return in constant time.

- `std::pair<iterator, bool>`
  `insert(const Key &key, const Value &value);`
  `std::pair<iterator, bool>`
  `insert(Key &&key, Value &&value);`
  Inserts a new element in the tree, either by copy or by move, or overwrites the value for `key` if it already exists.
  Returns an iterator pointing to the newly inserted/updated element, and a boolean being `true` if a new element was inserted, and `false` if an existing element was updated.
- `iterator find(const Key &key);`
  `const_iterator find(const Key &key) const;`
  Two overloads that looks up the given key and returns an iterator pointing to the element, or a past-the-end iterator if no such key exists.
- `void clear();`
  Erase all elements.
- `void erase(const Key &key);`
  Erase the key-value pair with the given key, or do nothing if no such key exists.
- `void erase(const_iterator pos);`
  Erase the element stored at `pos`.
  Note: you may need to use `const_cast` internally in your implementation.
- `value_type &front();`
  `const value_type &front() const;`
  Two overloads for accessing the first element.
  Must return in constant time.
- `value_type &back();`
  `const value_type &back() const;`
  Two overloads for accessing the last element.
  Must return in constant time.
- `iterator begin();`
  `const_iterator begin() const;`
  Two overloads for accessing an iterator to the first element.
  Returns a past-the-end iterator if the container is empty.
  Must return in constant time.
- `iterator end();`
  `const_iterator end() const;`
  Two overloads that returns a past-the-end iterator.
  Must return in constant time.
  Note that if the container is not empty, then it should be possible

to decrement this iterator in order to access the last element of the container. That is, `back()` is equivalent to `*--end()`.

You are welcome to make the tree self-balancing, e.g., make it a red-black tree, to make certain member functions run in amortized logarithmic time, but it is not a requirement.

3 Hints

- The specification in the sections above are for the `public` interface of the classes. How you internally implement the data structures is up to you, as long as the spirit of assignment is fulfilled. If you are in doubt, you should ask before proceeding. In particular, both containers are still node-based, so you will probably still need some nested node class. However, such a node class should not be exposed to the user and should be kept as `private`. You can use `friend class` declarations to make `private` member functions available for other classes.

- Try testing with an element type which does not have a default constructor.

- Try testing with an element type which is not copyable, but only moveable, e.g., one you make your self, or a `std::unique_ptr`. For such an element type there will be member functions that can not be used. Remember to document their requirement on the element type.

- The requirement of the iterators being bidirectional implies that the expression `*--end()` is valid when the container is non-empty. Without this requirement one could implement an iterator by holding an pointer to some private node type, which then represents a position in the container. However, a straight-forward implementation would then have `end()` return an iterator that stores a `nullptr`, but how would one then decrement that iterator to find the last element?

  One way to solve this is to have an iterator hold a second pointer, that points to the `List`/`Tree` it self, such that it knows which container it came from. The node pointer can then still be `nullptr` to represent past-the-end.

  Another sort-of-solution is to have a special past-the-end-node. However that comes with an extra memory overhead, and it may be difficult to allow for element types that are not DefaultConstructible. Remember to document the requirements on the element type.