DM852
Introduction to Generic Programming

Spring 2022

Assignment 1

This is the first in a series of effectively mandatory assignments in DM852. Note that the next assignments may build on this one.

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_$\langle username \rangle$", 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

Friday, 8 April, 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 basic implementation skills in C++, while including a few of the foundational aspects of generic programming. 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 `asg1/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 Doubly Linked List For Strings

Files: `asg1/src/List.{hpp,cpp}`
Implement a doubly linked list as a type **`List`** in the namespace **`DM852`**. It should fulfil the following requirements:

- It must store elements of type **`std::string`**.
- It must have a nested class **`Node`**, which must have the following public member variables:
  - **`std::string data;`** Is the element stored in this node.
  - **`Node *next;`**
    Must point to the **`Node`** storing the next element, or be **`nullptr`** if it is the last.
  - **`Node *prev;`**
    Must point to the **`Node`** storing the previous element, or be **`nullptr`** if it is the first.

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

- It must be a Regular type, i.e., be DefaultConstructible, Copyable, and EqualityComparable.
- 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. Similarly, 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.
  - `bool empty() const;`
    Returns `true` iff the container is empty. I.e., `empty() == (size() == 0)`.
  - `void push_back(const std::string &elem);`
    Inserts a new element in the end.
  - `Node *insert(Node *node, const std::string &elem);`
    Inserts a new element before the one stored in **node**, and returns a pointer to the node for the newly inserted element.
  - `void clear();`
    Erase all elements.
  - `void pop_back();`
    Erase the last element.
  - `void erase(Node *node);`
    Erase the element stored in **node**.
  - `std::string &front();`
    `const std::string &front() const;`
    Two overloads for accessing the first element.
  - `std::string &back();`
    `const std::string &back() const;`
    Two overloads for accessing the last element.
  - `Node *begin();`
    `const Node *begin() const;`
    Two overloads for accessing the node storing the first element. Returns `nullptr` if `empty()`.
  - `Node *end();`
    `const Node *end() const;`
    Two overloads that returns `nullptr`.

2.2 A Binary Search Tree Mapping Integers to Strings

Files: `asg1/src/Tree.{hpp,cpp}`
Implement a binary search tree as a type `Tree` in the namespace `DM852`. It should fulfil the following requirements:

- The tree must be an associative data structure that maps `int`s to `std::string`s.
- It must have a nested class `Node`, which must have the following public member variables:
  - `const int key;` Is the immutable key stored in this node.
  - `std::string value;` Is the value for the key stored in this node.
  - `Node *parent;`
    Must point to the parent `Node` of this node in the tree, or be `nullptr` if it is the root.
  - `Node *left;`
    `Node *right;`
    Must point to the left/right child `Node` of this node in the tree, or be `nullptr` if no such child exists.
- The nested class `Node` must additionally have the following member functions for in-order iteration through the tree:
  - `Node *next();`
    `const Node *next() const;`
    Two overloads that returns the node with the smallest key larger than the key in this node. Returns `nullptr` if no such node exists.
  - `Node *prev();`
    `const Node *prev() const;`
    Two overloads that returns the node with the largest key smaller than the key in this node. Returns `nullptr` if no such node exists.
- It must be a Regular type, i.e., be DefaultConstructible, Copyable, and EqualityComparable.
- 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. Similarly, 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.

- `bool empty() const;`
  Returns `true` iff the container is empty. I.e., `empty() == (size() == 0)`.
- `std::pair<Node*, bool>`
  `insert(int key, const std::string &value);`
  Inserts a new element in the tree, or overwrites the value for `key` if it already exists. Returns a pointer to the newly inserted/updated node, and a boolean being `true` if a new node was inserted, and `false` if an existing node was updated.
- `Node *find(int key);`
  `const Node *find(int key) const;`
  Two overloads that looks up the given key and returns a pointer to the node containing it, or `nullptr` if no such node exists.
- `void clear();`
  Erase all elements.
- `void erase(int key);`
  Erase the key-value pair with the given key, or do nothing if no such key exists.
- `void erase(const Node *node);`
  Erase the element stored in `node`.
  Note: you may need to use `const_cast` on the argument.
- `std::string &front();`
  `const std::string &front() const;`
  Two overloads for accessing the first element.
- `std::string &back();`
  `const std::string &back() const;`
  Two overloads for accessing the last element.
- `Node *begin();`
  `const Node *begin() const;`
  Two overloads for accessing the node storing the first element. Returns `nullptr` if `empty()`.
- `Node *end();`
  `const Node *end() const;`
  Two overloads that always returns `nullptr`.

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.