Exercise 1 (70 points)

In this exercise you will implement a chaining hash table. The table maps unique keys of type `int64_t` to values of type `GenericValue`. The implementation of the value type is provided in the subdirectory for this exercise.

A chaining hash table consists of an array of l *buckets*. The mapping for a given key k is stored in precisely one bucket of the hash table. The index of this bucket is determined by first computing the hash value of the key $h = f_h(k)$, which is usually a 64-bit integer. Since the number of buckets is usually much smaller than the range of a 64-bit integer, the index of the bucket is then calculated by $i = (h \bmod l)$.

For the same reason, the mappings for multiple keys may have to be stored within the same bucket, which is known as a *hash collision*. In a chaining hash table each bucket thus points to the start of a linked list, or chain, in which the actual mappings are stored. Each entry of this linked list stores the key and the associated value of a specific mapping. If a large number of hash collisions occur, the performance of a chaining hash table will degrade as these chains become longer. To avoid this, it is important that the hash function f_h distributes the hash values uniformly, and that the number of buckets l is large enough.

In order to look up the mapping for a key k, we first compute the index i of the corresponding bucket as outlined above. Then, we iterate over the entries in the linked list stored in that bucket, until we find the entry containing the key k. If we reach the end of the chain, no such entry exists and the lookup is unsuccessful.

Insertion in the hash table works very similar to searching: When the chain is traversed and the key of the entry that is to be inserted is already found, the existing entry is overwritten. Otherwise, the new entry is added to the chain.

When an entry should be removed, again the lookup algorithm is used, first. When no entry with the given key is found, nothing needs to be done. Otherwise, the entry is removed from the chain.

Figure 1 shows an example of a hash table with $l = 8$ and the identity hash function $f_h(k) = k$. The entry with the key 25 and the value D, for example, lies in the chain from the bucket at

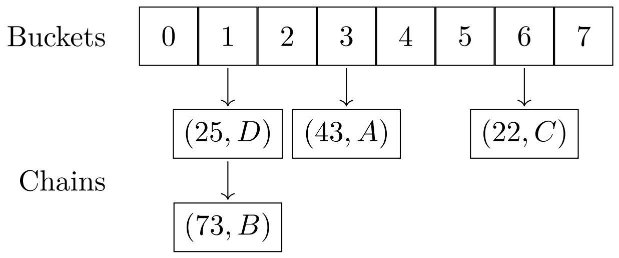

Figure 1: Example of a chaining hash table with $l = 8$ and $f_h(k) = k$.

index 1 since $(f_h(25) \bmod 8) = 1$. Since this table has $n = 4$ entries and a table size of $l = 8$ it has a *load factor* of $\frac{n}{l} = 0.5$.

To ensure that lookups can always be executed in constant time on average, the chains should not become very large. For this, the hash table must keep track of the load factor and resize the table of buckets if necessary. When the load factor grows beyond a given threshold and the table of buckets must be resized, this is called *rehashing*. In that case *all* entries of the hash table must be reinserted into the new, larger table as the index of their bucket can change when l changes. As rehashing is very expensive, it also should not happen very often. This can be guaranteed by growing the size of the table of buckets exponentially.

Your implementation should have a class named `ChainingHashTable`. It should have a member variable of type `std::vector` that stores the array of buckets. Initially, this array should be empty. When an element is inserted into a hash table with an empty array of buckets, its size should be set to 16. For all other insertions, the table should grow to exactly double its previous size when the load factor is larger than 0.5 after the insertion. Shrinking the table after deletions is not necessary. All key-value-pairs should be stored in a public nested class named `Entry` that has the two member variables `key` (should be `const`) and `value`. For simplicity you should use the the identity function as a hash function. You will have to define an additional class that wraps an `Entry` and a next pointer to implement the entry chains. You are not allowed to use a container from the standard library to implement the chains.

`ChainingHashTable` should have a default constructor. It should also be movable, i.e., it should define a move constructor and a move-assignment operator. In addition, implement the following member functions. They should all have amortized constant runtime. During rehashing, entries should be added to the front of the chain instead of the end to guarantee that insertions into a chain have constant complexity.

`size_t size() const`
: Returns the number of entries in the hash table.

`bool contains(int64_t key) const`
: Returns true if the hash table contains an entry with the given key.

`GenericValue& operator[](int64_t key)`
: Searches for an entry with the given key. If it does not exist, creates a new entry. Returns a reference to the value of the entry. Should potentially rehash the table as described above.

`GenericValue& insert(int64_t key, GenericValue&& value)`
: Inserts a new entry with the given key and value. If an entry with the same key already exists, overwrites its value. Returns a reference to the value of the entry. Should potentially rehash the table as described above.

`GenericValue& insert(Entry entry)`
: Another overload for `insert` with the same semantics that takes an `Entry`.

`void erase(int64_t key)`
: Removes the entry with the given key if it exists, does nothing otherwise.

The hash table should also support iteration. For this, implement a public nested class named `iterator` with the following (public) type aliases:

```
value_type = Entry
difference_type = std::ptrdiff_t
reference = Entry&
pointer = Entry*
iterator_category = std::forward_iterator_tag
```

Objects of type `iterator` should be copyable and have a default constructor which initializes the object to some unspecified state. It should have the following overloaded operators:

Dereference: Returns a reference to the entry this iterator points to, should be const-qualified.
Member of pointer: Returns a pointer to the entry this iterator points to, should be const-qualified.
Pre-increment: Increments the iterator so that it points to the next element in the hash table. Returns a reference to itself. The order in which the increment operator traverses the hash table can be chosen arbitrarily as long as it sees each element exactly once. Incrementing the iterator that points to the last element results in an *end iterator*.
Post-increment: Same as pre-increment but returns a copy of the iterator before it was incremented.
Equality: Returns true when both iterators point to the same element or when both are end iterators.
Inequality: Returns the opposite of the equality operator.

The class `ChainingHashTable` should have the following member functions that return iterators and should also have amortized constant runtime:

`iterator begin()`
: Returns an iterator to the first entry in the hash table. Which entry is considered the first can be chosen arbitrarily. If the hash table is empty, returns the end iterator.

`iterator end()`
: Returns the end iterator. The only requirement for end iterators is that they all are equal to each other. They do not have to be able to be incremented or dereferenced.

`iterator find(int64_t key)`
: Returns an iterator that points to the entry with the given key. If no such entry exists, returns the end iterator.

Finally, the destructor of `ChainingHashTable` should make sure that no recursion is used when removing all elements from the chains to avoid stack overflows with very large chains.

Write your implementation into the files `ChainingHashTable.hpp` and `.cpp` in the `lib` directory. It should *not* use `new` and `delete` for dynamic memory management. Use other, better alternatives that were shown in the lecture.

Exercise 2 (30 points)

The Resource Acquisition is Initialization (RAII) idiom was introduced in the lecture as a convenient way to manage the lifetime of resources. In this exercise, you are going to implement simple RAII wrappers for temporary files and directories through API calls.

On UNIX derivatives like Linux, *file descriptors* are used as handles to access files and directories. A file descriptor is a non-negative integer which is usually represented by the `int` type in C++. System calls that interact with files may return file descriptors or take file descriptors as arguments as a way to identify specific files. For example, the `open` POSIX API call opens or creates a file and returns a file descriptor which can be used for further interactions with this file (such as reading from, writing to, or deleting the file).

In C++ terminology, such a file descriptor is a *resource* with a defined lifetime – usually a file descriptor stays valid until it is passed to the `close` API call. It is easy to forget to invoke `close` on a file descriptor when it is no longer needed, which could result in resource leaks. Therefore, file descriptors should be wrapped in suitable RAII types. Going one step further, temporary files and directories themselves can be regarded as resources. When a temporary file or directory is no longer needed, its lifetime ends and it should be deleted. This can also be achieved through suitable RAII wrappers.

In this exercise you should implement two RAII classes, `TempDirectory` and `TempFile`, with the following semantics. `TempDirectory` represents a temporary directory which can either be created with an absolute path or as the child of another `TempDirectory` additionally to the directories name. On construction, `TempDirectory` creates the respective directory through suitable API calls (see below). On destruction, the directory is deleted again through API calls.

`TempFile` represents a temporary file which can only be created within a `TempDirectory`. On construction, `TempFile` creates an empty file with a given name in the correct directory through suitable API calls. On destruction, the file is removed again through API calls.

A `TempDirectory` should only be deleted once it is empty which can be implemented in a straightforward way using shared ownership semantics. All children of a directory assume shared ownership of the directory, so that the `TempDirectory` instance is only destructed once the last child has released ownership (e.g. by being destructed itself). You can assume that no other program can create files or directories within the directories managed by the `TempDirectory` class.

Finally, the `TempDirectory` class should be neither copyable nor movable, whereas the `TempFile` class should be movable but not copyable. Ensure that nobody else (except for your classes themselves of course) can access the resources that are managed by your classes.

For your implementation you should use the API calls provided by the functions that are declared in the header file `lib/OSApi.hpp`. As described above, they use file descriptors of type `int`. Negative values for the file descriptors indicate an invalid descriptor or an error. If your implementation is correct, the API functions should never generate errors. You can find more documentation in the header file. The program scaffold for this exercise also includes the other required header files. You should always use the functions using parent directory file descriptors instead of full paths when possible.

You will need to work with strings in your implementation in order to represent path- and filenames. While it is possible to do this with C-style strings (i.e. `const char*`), this is cumbersome and error-prone. You should instead use the C++ type `std::string` defined in the `<string>` header. Among other operations `std::string` supports assignment from C-style strings, copy- and move-semantics and concatenation through the binary `+` operator. The `c_str()` member function can be used to access the underlying C-style string.

(a) Implement the `TempDirectory` and `TempFile` classes as outlined above. Note that there are no tests that directly test your implementation of `TempDirectory` and `TempFile`, but there are tests that verify the semantics of the command line interface you will implement in the next subtask. It might be helpful to read the full task description before you start.

(b) Implement the `run` function of the `CommandLine` class. The run function takes a single `std::string` argument (`basedir`) which specifies the top-level directory. First of all, the function should create a temporary directory at the location specified by `basedir` using your `TempDirectory` class.

The `CommandLine` should maintain information about the *current working directory* and its parent directories. Initially, the current working directory is the top-level directory specified by `basedir`. Furthermore, it should maintain a list of temporary files (see below). The run function should repeatedly print a prompt (e.g. `>`) and read commands from `std::cin`. The following commands should be supported.

`enter`

Create a new temporary directory in the current working directory and update the current working directory to the newly created directory. Directories should be named `dirN`, where `N` is an integer that is increased for each directory created, starting at `0`.

Your program should print some brief information about the created directory (e.g. "`entering directory /x/y/z`")

`current`

Print the current working directory.

`leave`

Leave the current directory and set the current working directory to the parent of the current directory.

Your program should print some brief information about the directory that was left (e.g. "`leaving directory /x/y/z`")

If the directory that was left is empty it should be deleted (this should not require any additional code if you implemented the `TempDirectory` and `TempFile` classes correctly). If the user leaves the top-level directory, the program should exit.

`create`

Create a new temporary file in the current working directory. Files should be named `fileN`, where `N` is an integer that is increased for each file created, starting at `0`.

Your program should print some brief information about the created file (e.g. "`created file /x/y/z`")

`list`

Print a list of the currently open temporary files. Files should be numbered from `0` to the number of open files.

`remove N`

Remove the N-th currently open temporary file.

Your program should print some brief information about the removed file (e.g. "`removed file /x/y/z`")

If the directory that contained the removed file is empty afterwards, it should also be deleted (this should not require any additional code if you implemented the `TempDirectory` and `TempFile` classes correctly).

`quit`

Quit the program. All remaining open files and directories should be deleted (this should not require any additional code if you implemented the `TempDirectory` and `TempFile` classes correctly).

The test cases contained in the program scaffold test the command line interface and verify that files and directories are actually created/deleted correctly. A sample interaction with the command line interface could look as follows (if `run` is invoked with `"/tmp/raii"` for `basedir`).

```
> current
current directory: /tmp/raii
> enter
entering directory /tmp/raii/dir0
> create
created file /tmp/raii/dir0/file0
> create
created file /tmp/raii/dir0/file1
> leave
leaving directory /tmp/raii/dir0
> create
created file /tmp/raii/file2
> list
[0] /tmp/raii/dir0/file0
[1] /tmp/raii/dir0/file1
[2] /tmp/raii/file2
> remove 1
removed file /tmp/raii/dir0/file1
> list
[0] /tmp/raii/dir0/file0
[1] /tmp/raii/file2
> remove 0
removed file /tmp/raii/dir0/file0
```

At this point, the last file within the directory `/tmp/raii/dir0` has been deleted, so the directory should be deleted as well.

```
> list
[0] /tmp/raii/file2
> leave
leaving directory /tmp/raii
quitting
```

At this point, all remaining files should be deleted.