CSCI 152, Performance and Data Structures, Assignment 4

Rules for Assignments

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' is never a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, correct memory management, readability, style, and layout. We use **valgrind** for memory checks. We apply high quality standards during grading.
- Assignments are graded with the help of autograders. Make sure that your submitted code can be compiled. If you don't know how to complete a task, insert an empty function, so that your code compiles in all cases.
- In order to make it possible to adopt your code to our automatic checkers, and to give more feedback, we check your code at least twice. The last submitted version counts. Use the earlier checks. If you submit only for the last check, you are taking unnecessary risk.
- If you need help, first your question on Piazza. Questions that do not contain code can be asked publicly. Questions that contain code must be posted as private question. Don't mail questions directly to an instructor, because individual instructors are not always available. Piazza is monitored almost full time, so you will get a quicker answer.
- Don't wait until the last moment with starting to work on the assignment.
- You must write all submitted code by yourself! Even allowing others to see your code, or looking at others' code is already academic misconduct.

C^{++} Coding Rules

- Avoid unitialized variables. If you don't have a value to initialize a variable, you are almost certainly declaring it too early. C^{++} allows declarations of variables almost everywhere. Declare variables where you need them for the first time.
- Make sure to use `size_t` for all indexing. (Not `int`, nor `unsigned int`.) Don't write character contants by their ASCII codes. (Write 'a' instead of 97.)
- Do not implement anything in header files. In real life, short methods can be implemented in header files, while longer methods should be implemented in **.cpp** files. We test with our own header files, so everything that you write in a header file, will be ignored.
- You are not allowed to use any STL containers or library functions, unless explicitly mentioned in the assignment. If in doubt, ask in Piazza.
- Don't use `printf`, `malloc`, `free`, or `memcpy`. Don't use `#define`, except in include guards.
- Don't use `0` or `NULL` to represent the null-pointer. The only correct representation of the null-pointer is `nullptr`. It is OK to write `if(p)` if you want to test that `p` is not the null-pointer.
- Avoid `break` and `continue`. They are just `goto` in disguise. In nearly all cases, they can be avoided by changing the condition of the **while** loop, which always results in more readable code.
- Don't `throw` and `catch` an exception inside the same function. That is not what exceptions are intended for.
- Avoid assignments in constructor bodies. Use member initializers wherever possible.
- Don't write: `if(b) return true; else false;` Just write `return b;`

1 Introduction

Goal of this assignment is to make you familiar with the use of hashing for implementation of the map ADT, and also with the map ADT itself. Hashing uses a hash function to obtain an index into a vector `std::vector` or array, in which the key/value pair for the key will be stored, if it is there. In this way, searching the complete vector can be avoided. Since multiple keys can have the same index, key/value will be stored in buckets. For this purpose, we use `std::list`. If the average bucket size is kept constant, hashing will insert and find key/value pairs in constant time.

Similar to the previous assignment, the map must be implemented in such a way that it is easy to change the types. This applies to the key type, the value type, the hash function, and also to the comparison function. There are different test cases for the different types, and you must try them all:

1. Key type is `std::string`, and value type is `int`.
2. Key type is `int`, and value type is `std::string`.
3. Key type is `std::string`, value type is `int`, and the map must be case insensitive. As in the previous assignment, this means that 'AsTaNa' and 'aStAnA' will be considered the same string. In order to implement this, you can use `case_insensitive_cmp` from the previous assignment, but you have to write an additional `case_inensitive_hash`.

In order to represent key/value pairs, we use `std::pair< keytype, valtype >`. Using `std::pair` is easy. Here are some example of its use:

```
std::pair< double, std::string > p1 = { 400, "four hundred" };
std::cout << p1. first << "\n";  // Prints 400.
std::cout << p1. second << "\n"; // Prints "four hundred"
p1. first = 100000;
p1. second = "one million";
```

In order to implement the buckets, we use

```
using buckettype = std::list< std::pair< keytype, valtype >> ;
```

Here `std::list` is the implementation of doubly linked list in the STL. In order to use it, you need to understand **iterator**s. Iterators occur very frequently in the STL. An iterator is like a pointer, but it is associated with a container type, and it hides the details of its implementation. In order to go through linked list in *C*, one needs to write something like this:

```
for( listnode* p = lst. first; p != NULL; p = p -> next )
   greet( *p );
```

We are doing C^{++} here, and we want to hide from the user how linked lists are implemented. In particular, we want to hide the name of the pointer to the first node, the fact that the list ends with a null-pointer, and how the next

element is obtained. This can be obtained by introducing a new type with name `list<T> :: iterator`, and redefining `*, ++, --` for this type. Using `list<T> :: iterator`, we can write

```
for( std::list<T> :: iterator p = lst. begin( );
     p != lst. end( ); ++ p )
{
   greet( *p );
}
```

Since `std::list<T> :: iterator` is a bit long, one normally just writes `auto p = lst. begin( )`. In addition to `std::list<T> :: iterator`, list has also `std::list<T> :: const_iterator`, which is like an iterator, but cannot change the value that it points to. A **const** list has only `const_iterator`. These are the main methods of `std::list`:

```
std::list<T> :: iterator       // The iterator type of list<T>
std::list<T> :: const_iterator // The const_iterator type.

lst. begin( );       // Iterator to the first element.
lst. end( );         // Iterator, one-beyond-the-last element.

lst. push_back(t);   // Adds t to the end.
lst. push_front(t);  // Adds t to the front.
lst. pop_back( );    // Removes last element.
lst. pop_front( );   // Removes first element.

lst. erase( iter );  // Erase the element that iter points to.
lst. size( );        // Size of the list.
```

As with assignment 3, we will also measure the performance. You must use the same classes `timer` and `timetable` that were provided in exercise 7. Explanations of their use were given in exercise 7.

Download the starter code, which contains the files **map.h**, **map.cpp**, **main.cpp** and the **Makefile**.

As usual, all methods of map must be implemented in the file **map.cpp**. In addition, you must create additional tests in the **main.cpp**. You have to submit **map.cpp** and **main.cpp**. Make sure that the `#if`s in **main.cpp** are set to use the measurements at the end. It must compile with key type `std::string` and value type `int`.

2 Implementation of Hashmap with (string, int)

Make sure that

```
using keytype = std::string;
using keyhash = standard_hash< keytype > ;
using keycmp = standard_cmp< keytype > ;
using valtype = int;
```

1. Implement the two methods

   ```
   buckettype& getbucket( const keytype& key );
   const buckettype& getbucket( const keytype& key ) const;
   ```

   Get the hash value modulo the current number of buckets, and use it to return the proper bucket.

2. Implements the two methods

   ```
   static buckettype::iterator
   find( const keytype& key, buckettype& bk );

   static buckettype::const_iterator
   find( const keytype& key, const buckettype& bk );
   ```

   These methods return either an iterator to a pair, whose `first` equals `key`, or `bk. end( )` if `key` is not present. (Remember this pattern, all methods that use an iterator for finding something use `end( )` for representing failure.)

   Make sure to use `cmp`, and not built-in `==`. Remember that `cmp` returns 0 when the arguments are considered equal.

   In order to access the first element of an iterator of a list of pairs, write `it -> first`.

3. Write the method `size_t nrbuckets_needed( size_t sz ) const` that computes the number of buckets needed when the hashmap has size `sz`. It should be the smallest power of three, which multiplied by the maximum load factor yields a number greater or equal to `sz`. In addition, it must never be less than three.

4. Write method `void rehash( size_t nrbuckets )`. Iterate through every list in `oldbuckets`, and reinsert the key/value pairs into the current hashmap. Use `getbucket()` to find the bucket where key/value must be reinserted.

5. Write method `bool insert( const keytype& key, const valtype& val )`. First use `getbucket( )` to find the bucket in which `key/val` must be situated. Make sure to keep the bucket as a reference, otherwise you will be inserting into a copy, which is useless. After that, use `find( )` to check if `key` is already present. If `find( )` does not return the `end( )` of the bucket, just return `false` and that's all. Otherwise, check if a rehash is needed (by calling `rehash_needed( )`.)

   If yes, call `rehash( nrbuckets_needed( current_size + 1 ))`, call `getbucket( )` again to get the new bucket of `key` and add `{ key, val }` to it.

   If not, simply add `{ key, val }` to the bucket that you have already.

6. Write method `bool contains( const keytype& key ) const`, that decides if the map heared about `key`. First use `getbucket( )`, then use `find( )` to check if the bucket contains the key.

7. Write the two `at` methods

   ```
   const valtype& at( const keytype& key ) const;
   valtype& at( const keytype& k );
   ```

   Both methods must throw `std::out_of_range` if `key` is not present.

8. Write `valtype& operator[] ( const keytype& key )`. First use `getbucket( )` and `find( )` to check if `key` is present. If `find( )` does not return `end( )` of the bucket, return the second field of the value that the returned iterator points to.

   Otherwise, use `push_back( { key, valtype( ) } )` to add a default value to `key` and return the second element of the added pair.

   As with `insert( )`, you first have to check if a rehash is needed before creating the new pair.

9. Write `bool erase( const keytype& key )`. Use `getbucket( )` and `find( )` to get an iterator for `key`. It `key` is present, use the `erase` method of list to delete `key`. You don't need to rehash if the load factor gets low.

10. Write the methods

    ```
    size_t size( ) const;
    bool empty( ) const;
    void clear( );
    ```

11. Check that your code also compiles with

    ```
    using keytype = double;
    using keyhash = standard_hash< keytype > ;
    using keycmp = standard_cmp< keytype > ;
    using valtype = std::string;
    ```

    (Use the Kazakh counting example).

3 Case Insensitive Map

In order to obtain a map that uses case insensitive strings as keys, use

```
using keytype = std::string;
using keyhash = case_insensitive_hash;
using keycmp = case_insensitive_cmp;
using valtype = double;
```

You can reuse `case_insensitive_cmp` from the previous assignment, but `case_insensitive_hash` has to be written:

12. Implement

```
size_t case_insensitive_hash::operator( )
                        ( const std::string& s ) const
```

    This hash function must be case independent! This means that if one replaces a few uppercase letters by their corresponding lowercase letters (or reverse), then the hash value must not change. You can use function `tolower( char )`. Don't make a lowercase copy of the complete string.

4 Performance Measurements

You can now do the performance measurements.

```
using keytype = std::string;
using keyhash = standard_hash< keytype > ;
using keycmp = standard_cmp< keytype > ;
using valtype = int;
```

The code for the measurements is already present in **main.cpp**. It is similar to assignment 3. Use `g++ -O3 -flto`, and no valgrind. (You should use it during testing, however.) The performance should be $O(n)$. If you are not getting similar performance, you are doing something wrong, and you will lose points. Collect the measurements and include them in a comment in **main.cpp**.

5 Submission

Submit your files **map.cpp** and **main.cpp**. Make sure that your submitted main compiles with

```
using keytype = std::string;
using keyhash = standard_hash< keytype > ;
using keycmp = standard_cmp< keytype > ;
using valtype = int;
```