# 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.
- 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 twice. The last submitted version counts. Always submit for the first check. If you submit only for the last check, you are taking unnecessary risk.
- The starter code contains a **main.cpp** file containing rudimentary tests. These tests are not sufficient. You have to create additional tests that include at least all functions that are included in the assignment. In the end, you have to submit a **main.cpp** file with the tests that you created. It is possible that your tests will be inspected.
- If you need help, you can either **(1)** come to a lab session, or **(2)** post your question on Piazza in a private question (with visibility set to 'all instructors'.) Do not mail directly to an instructor.
- 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 constants by their ASCII codes. (Never write 97 instead of 'a'.)
- Do not implement anything in header files. In real life, short methods should be implemented in header files, while longer methods should be implemented in **.cpp** files. Because of the way that we test, this is not possible in assignments. 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 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. In the ideal situation, the key/value pair is always stored at the position that is dictated by its hash value. Unfortunately, it can happen that different keys have the same index. There are different ways to deal with this problem: With *closed addressing*, the table consists of lists (or vectors) of key/value pairs, so that more than one key/value pair can be stored at the same position. With *open addressing* a key/value pair can be stored at a different index, when its original index is occupied.

This has consequences for lookup: If one is looking for a key/value pair, and it is not present at its index, this does not imply that the key/value pair is not there. It could be the case that the original index was occupied, and that the wanted key/value pair was put at another position, so we have to look at different positions. In general, it is hard to know when to stop searching. If one never deletes anything, one can stop looking as soon as an empty position is encountered, because insertion would have put the key/value pair there. If one allows deletion, the situation gets a bit ugly: An empty position may have been occupied at the moment that the key/value pair was inserted, and because of that the key/value pair was not inserted at this position. So there is no guarantee that we can stop looking when we encounter an empty position. This can in principle be solved by distinguishing two types of empty positions. Positions that have always been empty, and positions that contained a key/value pair that was later deleted.

This is ugly, and we will not ask you to implement this. Instead, we will ask you to implement a variant of open addressing, where due to a clever trick, it never happens that there is a key/value pair that is not at its correct position, with an empty place betweent the correct position and their current position. This form of hashing is called *Robin Hood* hashing. I will later explain why it has this name.

# 2 Robin Hood Hashing

Robin Hood hashing uses a form of *linear probing*. If a key/value pair must be put at index $i$, but this index is already occupied, we try index $(i + 1)$. If that position is also taken, we try $(i + 2)$, etc.
We use the following terminology:

- Let $k$ be a key. We write $h = \mathbf{hash}(k)$ for its hash value.
- We assume a hash **table** of size $S$. Valid indices $i$ have $0 \leq i < S$. We always assume that the **table** has at least one empty place. This implies

that, if one starts inserting, there must be at least two empty places. If not, one must rehash.

- For a key/value pair $(k, v)$ we define its *primary position* $\mathbf{prim}(k)$ as $\|\mathbf{hash}(k) \bmod S\|$. This is the place where we prefer to insert $(k, v)$ if no other key is in the way.
- We call the position at which $(k, v)$ is eventually placed in the **table**, the *effective position* of $(k, v)$.

Robin Hood hashing uses the following two invariants:

1. For a given key/pair value, if one walks from its primary position to its effective position, one will not encounter an empty place.
2. For a given key/pair value, if one walks from its primary position to its effective position, there is no other key, for which one encounters both its primary and effective position.

Part 1 is easy to understand. We want the effective position as close as possible to the primary position, so if we enounter an empty place, the key/value pair should have been put there.

Part 2 is a bit harder to grasp. One also needs to consider that fact that **table** should be treated as circular. After index $S - 1$ comes index 0. Suppose there are two keys $k_1, k_2$ in the hash table: Let $\mathbf{prim}(k_1)$ and $\mathbf{eff}(k_1)$ be the primary and effective positions of $k_1$. Similarly, let $\mathbf{prim}(k_2)$ and $\mathbf{eff}(k_2)$ be the primary and effective position of $k_2$.

If one starts walking to the right at $\mathbf{prim}(k_1)$, the following sequences are allowed by **(2)**:

$$\mathbf{prim}(k_1)\ \mathbf{eff}(k_1)\ \mathbf{prim}(k_2)\ \mathbf{eff}(k_2)$$
$$\mathbf{prim}(k_1)\ \mathbf{prim}(k_2)\ \mathbf{eff}(k_1)\ \mathbf{eff}(k_2)$$
$$\mathbf{prim}(k_1)\ \mathbf{eff}(k_2)\ \mathbf{eff}(k_1)\ \mathbf{prim}(k_2)$$

The following sequences are not allowed by **(2)**:

$$\mathbf{prim}(k_1)\ \mathbf{eff}(k_1)\ \mathbf{eff}(k_2)\ \mathbf{prim}(k_2)$$
$$\mathbf{prim}(k_1)\ \mathbf{prim}(k_2)\ \mathbf{eff}(k_2)\ \mathbf{eff}(k_1)$$
$$\mathbf{prim}(k_1)\ \mathbf{eff}(k_2)\ \mathbf{prim}(k_2)\ \mathbf{eff}(k_1)$$

This covers all cases. If one fixes $\mathbf{prim}(k_1)$ at the first position, there remain 6 permutations. In each of the allowed sequences, if one rotates $\mathbf{prim}(k_2)$ to the first place, and exchanges $k_1$ with $k_2$, the result is again an allowed sequence. The same applies to the forbidden sequences.

The combination of Part 1 and Part 2 makes lookup easy. Suppose we want to lookup $k_1$. First compute $p = \mathbf{prim}(k_1)$. Start walking to the right, and compare the keys to $k_1$. If we encounter an empty place, then Part 1 guarantees that we can stop looking. (This solves the deletion problem mentioned in the introduction). If we encounter a key $k_2$ this means that we have arrived at its effective position $\mathbf{eff}(k_2)$. We can compute its primary position $\mathbf{prim}(k_2)$. If this position also comes after $\mathbf{prim}(k_1)$, we can stop looking because of Part 2. If $\mathbf{eff}(k_1)$ would occur later, this would result in a forbidden sequence.

### 2.1 Mathematical Description

We give a description in mathematical terms. The assignment can be made without it. For positions $p_1, p_2, p_3$, define **between**$(p_1, p_2, p_3)$ as follows: If one starts walking to the right from position $p_1$, the position $p_2$ is encountered before $p_3$. This is the case if at least one of the following three inequalities holds:

$$p_1 \leq p_2 < p_3$$
$$p_3 < p_1 \leq p_2$$
$$p_2 < p_3 \leq p_1$$

We now define our delicate invariants: Let $(k, v)$ be key/value pair that occurs in **table**. Let $p_1$ be its primary position, let $p_2$ be its effective position.

1. There is no $q$, with **between**$(p_1, q, p_2)$, s.t. **table**$[q]$ is empty.
2. There does not exist a key/value pair $(k', v')$ that occurs in **table**, which has primary position $q_1$ and effective position $q_2$, s.t. both **between**$(p_1, q_1, p_2)$ and **between**$(p_1, q_2, p_2)$.

### 2.2 Distance Fields

When searching for a key $k_1$, we can stop when we encounter a key $k_2$ whose **prim**$(k_2)$ comes after **prim**$(k_1)$ because of Part 2 of the invariant. We do not want to recompute **prim**$(k_2)$ for every key $k_2$ that we encounter, and check if it is before **prim**$(k_1)$. Instead, we will just store the distance to **prim**$(k_1)$ together with $k_1$ in the table. When looking for $k_1$, we start at **prim**$(k_1)$ with `dist = 0`. Whenever we move a position to the right, we increase `dist`. If we encounter a cell that has its distance less than our `dist`, we can stop looking, because then we know that **eff**$(k_2)$ and **prim**$(k_2)$ are after **prim**$(k_1)$.

## 3 Implementation

The definition of map is as follows:

```
struct map
{
   using keytype = int;
   using keyhash = standard_hash< keytype > ;
   using keycmp = standard_cmp< keytype > ;
   // using keyhash = case_insensitive_hash;
   // using keycmp = case_insensitive_cmp;

   using valtype = std::string;

public:
   double max_load_factor;
```

```
      // Must lie between 0 and 1.

   size_t truesize;
      // The number of cells that are in use.

   struct cell
   {
      keytype key;
      valtype val;
      size_t dist;
         // We use std::numeric_limits< size_t > :: max( ) to
         // indicate that the cell is not in use.

      cell( )
         : dist( std::numeric_limits< size_t > :: max( ))
      { }

      cell( const keytype& key, const valtype& val, size_t dist );
      cell( keytype&& key, valtype&& val, size_t dist );

      bool isempty( ) const
         { return dist == std::numeric_limits< size_t > :: max( ); }

      void clear( )
      {
         key = keytype( );  // Not really needed.
         val = valtype( );
         dist = std::numeric_limits< size_t > :: max( );
      }

      void print( std::ostream& out ) const;
   };

   std::vector< cell > table;
};
```

As usual, you should test with different keytypes and valtypes. During implementation, use

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

because that's the easiest combination. Later you must also try `std::string` for `keytype` and `valtype`, together with a case-insensitive comparator.

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

file **map.cpp**. There are some methods already present in **map.cpp** that you should not delete. In addition to implementing the methods of `map`, you must create your own tests in **main.cpp**. You have to submit **map.cpp** and **main.cpp**.

1. Implement method

   ```
   size_t exactposition( const keytype& key ) const;
   ```

   This method must return the exact position of `key` in `table`. It is equal to the hash value modulo the current table size.

2. Implement method

   ```
   size_t wrap( size_t pos ) const;
   ```

   This method returns a valid index, obtained by substracting `table. size( )` if `pos >= table. size( )`. It should be sufficient to do this at most once.

3. The size of `table. size( )` will always be a power of three. Write function `static size_t minpow3( size_t sz );` that returns the smallest power of 3 that is `>= sz`, but never less than 3. This means `minpow3(15)` returns 27, and `minpow(2)` returns 3.

4. Implement

   ```
   size_t find( const keytype& key, size_t pos ) const;
   ```

   The specification of this function is subtle: `pos` must always be the exact position of `key`. If `key` is present, then `find` returns a number `d`, s.t. `table[ wrap( pos + d ) ]` contains `key`.

   If `key` is not present, then it returns the smallest number `d`, s.t. `table[ wrap( pos + d ) ]` is either empty, or its distance is less than `d`. Because of the invariants, we know that this is the moment we can stop looking for `key`.

   The typical way in which `find` is used, is as follows:

   ```
   size_t pos = exactposition( key );
   size_t dist = find( key, pos );
   pos = wrap( pos + dist );
   if( !table[ pos ]. isempty( ) && table[ pos ]. dist == dist )
      // Found!.
   ```

   Further action depends on which function we are in. If we only want to lookup `key`, we are done at this point.

   If we are trying to insert `key`, then `pos` is the point where `key` should be inserted, using `dist`. The key that is currently at position `pos` must be moved upwards.

5. Implement method

   ```
   void insert_shift_up( size_t pos, cell& c );
   ```

   This method must swap `table[ pos ]` with `c`. After that, as long as `c` is not empty, it must be reinserted. Move upward, increasing `c.dist`, until either an empty position is encountered, or a position whose distance is less than `c.dist`. Swap, and continue with the new `c` until `c` is empty.

   Robin Hood hashing is called 'Robin Hood hashing' because a cell with a greater distance is poorer than a cell with a small distance. The poor steal the place from the rich.

6. Write method `bool contains( const keytype& key )`. Its form is explained under Task 4.

7. Write method

   ```
   bool insert( const keytype& key, const valtype& val )
   ```

   First try to find `key`, as described under Task 4. If `key` is present, nothing further needs to be done.

   Otherwise, check if a rehash is needed. Call `rehash_needed( truesize + 2 )`. If it is needed, call `rehash` with `truesize + 2`. (At this moment, this function does nothing, but you will fill it later.) After that, recompute `pos` and `dist`.

   Create a cell, be sure to fill in the right distance. After that, call `insert_shift_up` with `pos`, increase `truesize` and return `true`.

8. We now write `rehash( size_t sz )`. Calculate the required capacity, and create a new `map` by calling a constructor. Go through `table`, and insert the non-empty cells into the copy. After that, swap the copy with `*this`, or move the copy into `*this`. Don't use copying assignment!

9. 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.

10. Write `valtype& operator[] ( const keytype& key )`. First lookup `key` in the usual way. If `key` is present, we return return its `val`. Otherwise, proceed as with `insert`, using the default `valtype( )` as value.

11. Write `bool erase( const keytype& key )`. First check if `key` is present, using the method described under Task 4. If `key` is not present, then there is nothing left to be done.

Otherwise, let `pos` be the position of `key`. We need to consider the possibility that there are cells at positions after `pos` that can be moved to the left. These cells can be recognized by the fact that they are non-empty and have distance different from zero. This is not difficult in principle but unfortunately one cannot write a simple **for** loop, due to the possibility of wrap around. Create a variable `pos2 = wrap( pos + 1 )`, and copy `table[pos2]` to `table[pos]` as long as it is non-empty, and its `dist` is nonzero. Don't forget to decrease the `dist` fields.

Use `pos = pos2; pos2 = wrap( pos2 + 1 )` to proceed to the next cell.

At the end, decrease `truesize` and clear `table[ pos ]`. You don't need to rehash when the load factor gets low.

12. Write the `clear( )` method.

During testing, you can use the print method that is given, and you can use `checkinvariants( )`.

# 4 Test with Int/Int

If you use

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

you can run the tests marked with `int/int` in the file **main.cpp**. We advice that you do that. The tests are quite extensive.

# 5 String with Case-Insensitive Map

In order to make sure that your implementation does not depend on the concrete key and value types used, we also make some tests with

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

The resulting map must be case insensitive. As in Assignment 3, this means that 'AsTaNa' and 'aStAnA' will be considered the same string. In order to implement this, you can reuse `case_insensitive_cmp` from the previous assignment, but you have to write an additional case insensitive hash function:

13. Implement

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

This hash function must be really 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 )`, but don't make a lowercase copy of the complete string!

# 6 Submission

Submit your files **map.cpp** and **main.cpp** as separate files. Don't use an archiver. Make sure that your submitted **main.cpp** can be compiled with

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

You can use other combinations during testing, but make sure that they are turned off with `#if 0`.

Don't delete or change the methods already present in **map.cpp**!