Exercise 1 15P

submission date: 07 Nov (23:59)

The submission works by pushing into your own private repository. Please invite us to your repository until 24. Oct. Create your private repository, by forking from this repository.

Note: Throughout this lab course we will upload frame works and interfaces, that have to be implemented/changed. Try not to change the visible interface of each file/class/function. You can add specifiers (like `const`) wherever appropriate or you can add new functions, but please keep in mind that we have our own tests that depend on the overall interface. You can easily try different templates, or different implementations with other names, but please make sure that one default is defined i.e.:

```
template <class something = defaults>
class TemplatedInterface {...}

using Interface = TemplatedInterface<>
```

The interface does include file names, if you want to create additional files, for example to diversify the solutions for your benchmarks/evaluation, then also provide the appropriate variants in the files provided in our file structure, this could also look like this.

if we provide a file `.../hash_b.h`), then your repository should contain a file with the same name. However it could look something like this:

```
.../hash_b.h:
#include "test_42_magic_precognition.h"
```

Without anything else, however, the file `test_42_magic_precognition.h` should then contain the appropriate class/code (using the predefined interface formerly found in `hash_b.h`)

1.1 Optimizing a Hash Table (common performance bugs) (6)

In the folder `hash` you will find a project that should already compile (see README). It contains a test for a simple hash table with linear probing. This hash table has not been optimized properly. Your task is to find the existing weaknesses in the code and fix them. The code of our hash table is replicated 3 times (`hash_original`, `hash_a`, `hash_b` in the folder implementation). In sub-task a and b you are supposed to change the code in `hash_a` and `hash_b` respectively. Do not change the hash table in the file `hash_original`; it is necessary as a base line for the experiments.

a) Performance Bugs (2)

Improve the performance of the hash table as far as possible, by only changing the `insert` and `findPos` functions. Do not change any other functions or members of the table (you may add new members).

b) Structural Improvement (2)

Improve the table similarly to question a), but for this exercise changing the hash table structure and other members is fair game. You may therefore also change the size of the table (never use more than twice the size of the original solution).

Do not waste all your time trying different options in this subtask. Maybe come back to it, once you have solved the rest of the exercise sheet. Explicitly, you do not have to split the table into separate key and value arrays. This does not help because outputting an element would cause at least two cache misses instead of one (and probing tends to be short). Similarly, you do not have to change the iterator used in our example.

c) Presentation of the Results (2)

Create plots that show your improvements towards an efficient hash table (show insert, find, and unsuccessful find performance). These plots have to be generated using a script which reads the files created by a measurement. The plots should all contain a legend, axis labels, and reasonable scales that show your improvements.

An example script can be found in the evaluation folder. You can adapt this to better show your improvements, or you can write a new script (using the scripting language of your choice, although it has to be human readable, executable, and concise).

Upload the measurements of one of your test runs together with the resulting pdf into your git repository.

If you want to experiment with some additional hash table variants (hash_{a0}, hash_{a1}, ...) you have to create the corresponding file, include it in test.cpp (similar to the others), and add it to the cmake compile targets.

1.2 *d*-ary Heaps and Compile Time Optimization (9)

In this exercise, we want you to create multiple different variants of a d-ary heap datastructure (we use max-heaps, because it's the STL standard). Binary heaps (2-ary heaps) should be known from Algorithms I as a priority queue implementation that uses a full tree, which can be stored implicitly in an array (without pointers).

To compute the indices of child or the parent nodes of any given heap element (with index i) you can use the following formulas:

$$\mathit{child}_j(i) = i \cdot d + j + 1$$
$$\mathit{parent}(i) = \left\lfloor \frac{i-1}{d} \right\rfloor$$

The functionality of the heap is based on the heap invariant $\mathit{pri}(i) < \mathit{pri}(\mathit{parent}(i))$ (for $i > 0$). There should be 4 supported functions:

1. `constructor(c)` creates a table with maximum capacity `c`.
2. `push(x)` inserts `x` into the priority queue. This is done by inserting the element at the back of the array and swapping it with its current parent until the parent is smaller than the element (then the heap invariant is restored).
3. `top()` returns the element with the highest priority (in position 0). Reading the element does not violate the heap invariant, therefore, nothing has to be done.
4. `pop()` removes the element with the highest priority. To restore the heap invariant we replace the minimum with the last element stored in the array. This element can then be moved downwards by repeatedly swapping it with its largest child.
5. `emplace(...)` similar to push, but instead of receiving a full fledged element, emplace receives the inputs for an element construction. This interface can be used to reduce the number of copy constructors that are called (if you want to learn more about this, either read up on move constructors, or un-comment the appropriate test in the correctness test.

Note: there is a template Parameter called `Comp`, this is the comparison function it defaults to `std::less<.>` it should be used instead of simple < or > comparisons (this allows using different types like `std::pair<.,.>` that do not have a comparison operator).

a) Runtime Parameter (2)

Create a d-ary variant, where d can be set in the constructor. It should be possible to set any arbitrary $d \in \mathbb{N}$ (see `pq_a`).

b) Compile Time Parameter (1)

Create a d-ary variant, where d is set as a non-type template parameter (compile time constant). This ensures that the compiler can use the explicit value for any optimizations (here the compiler improves integer division) (see `pq_b`).

c) Power of Two Runtime Parameter (1)

Create a d-ary variant, where d is a runtime parameter, but d is limited to powers of two (constructor parameter $k = \log(d)$). In this case you can optimize the division yourself using bit shifts instead of integer division (see `pq_c`).

d) Addressable Heap (2)

Choose one of the created heap implementations and make it addressable. An interface for the addressable priority queue can be found in `pq_d`. The basic idea is to have a handle for each inserted element. Through this handle, elements can be accessed, and the key can be changed. The handle has to be updated with the position of its element whenever said element is moved.

Note: Handles have to remain valid (they cannot move in memory), even if the element they represent is moved. Thus they cannot be stored in the implicit tree representation. They have to be allocated individually, or be part of some larger handle data-structure (be sure to avoid memory leaks).

Each element in the implicit tree representation has to be linked to its handle and vice versa (through pointers or offsets).

e) Presentation of the Results (3)

Create a plotting script similar to the one in 1.c) that shows a comparison between the heap variants created in a)-d). Also upload a set of measurements and the resulting plots. Your plots should display all necessary performance metrics. Also try different outgoing degrees (try also non-powers of 2 for a and b).