Aaron Pham - 400232791 - phama10

Problem 1

P1.1

Consider an initially-empty stack S and the following sequence of operations:

```
PUSH(S, 3), POP(S), PUSH(S, 17), PUSH(S, 5),
PUSH(S, 15), POP(S), POP(S), POP(S)
```

Illustrate the result of each operation (clearly indicate the content of the stack after the operation and, in case of a `POP`, the returned value by the operation

Solution

Stack S is initially empty, or $S = \emptyset$

The sequence of operations is as follows:

1. `PUSH(S, 3)`: $S = \{3\}$
2. `POP(S)`: $S = \emptyset$, returned value is 3
3. `PUSH(S, 17)`: $S = \{17\}$
4. `PUSH(S, 5)`: $S = \{17, 5\}$
5. `PUSH(S, 15)`: $S = \{17, 5, 15\}$
6. `POP(S)`: $S = \{17, 5\}$, returned value is 15
7. `POP(S)`: $S = \{17\}$, returned value is 5

8. `POP(S)` : $S = \emptyset$, returned value is 17

P1.2

Consider an initially-empty queue Q and the following sequence of operations:

```
ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 17),
ENQUEUE(Q, 5), ENQUEUE(Q, 15), DEQUEUE(Q),
DEQUEUE(Q), DEQUEUE(Q)
```

Illustrate the result of each operation (clearly indicate the content of the queue after the operation and, in case of a `DEQUEUE`, the returned value by the operation

Solution

Queue Q is initially empty, or $Q = \emptyset$

The sequence of operations is as follows:

1. `ENQUEUE(Q, 3)` : $Q = \{3\}$
2. `DEQUEUE(Q)` : $Q = \emptyset$, returned value is 3
3. `ENQUEUE(Q, 17)` : $Q = \{17\}$
4. `ENQUEUE(Q, 5)` : $Q = \{17, 5\}$
5. `ENQUEUE(Q, 15)` : $Q = \{17, 5, 15\}$
6. `DEQUEUE(Q)` : $Q = \{5, 15\}$, returned value is 17
7. `DEQUEUE(Q)` : $Q = \{15\}$, returned value is 5
8. `DEQUEUE(Q)` : $Q = \emptyset$, returned value is 15

P1.3

Assume we have a stack implementation `MyDynArrayStack` using dynamic arrays: the implementation supports `N` push operations with an amortized runtime complexity of $\Theta(1)$ per operation, and the implementation supports `POP`, `EMPTY`, and `SIZE` operations with runtime time complexity of $\Theta(1)$

Provide a *queue* implementation that uses `MyDynArrayStack` and supports any valid sequence of `N` `ENQUEUE` and `DEQUEUE` operations with an amortized runtime complexity of $\Theta(1)$ per operation. Explain why your implementation has the stated amortized runtime complexity for `ENQUEUE` and `DEQUEUE` operations.

Solution

Queue implementation `MyDynArrayQueue` using two `MyDynArrayStack`, `sIn` and `sOut`:

```
Algorithm Queue
    $sIn$ := MyDynArrayStack()
    $sOut$ := MyDynArrayStack()
```

```
Algorithm ENQUEUE(x)
    $sIn$.PUSH($x$)
```

```
Algorithm DEQUEUE()
    if $sOut$.EMPTY() then
        while $\neg sIn$.EMPTY() do
            $sOut$.PUSH($sIn$.POP())
        end while
    end if
    return $sOut$.POP()
```

where `sIn` is used to store elements, and `sOut` is used to store elements in reverse order.

Amortized runtime complexity explanation:

1. `ENQUEUE`

    > - `PUSH` to `sIn` has an amortized runtime complexity of $\Theta(1)$ per operation, as stated in the problem.

2. `DEQUEUE`

    > - When `sOut` is empty, the transfer of element from `sIn` to `sOut` has a runtime complexity of $\Theta(N)$ with worst case.
    > - However, the item was moved once per enqueue-dequeue cycle, so the total time spent is proportional to N, thus making amortized runtime complexity of $\Theta(1)$ per operation.

Problem 2

Consider the relations courses(`prog` , `code` , `name`) (that models courses named name and identified by the program `prog` the course is part of, e.g., SFWRENG, and the course code code, e.g., `2C03`) and `enrolled(prog, code, sid)` (that models students with identifier `sid` enrolling for a course identified by program `prog` and course code `code`).

We want to compute the list of all pairs (`sid`, `name`) in which `sid` is a student identifier s and name is the name of a course the student with identifier s is enrolled in. To compute this list, we developed the following two nested-loop algorithms:

```
**Algorithm** CEJOIN(courses, enrolled)
    $output := \emptyset$.
    **for** $(p_c, c_c, n_c) \in$ courses **do**
        **for** $(p_e, c_e, s_e) \in$ enrolled **do**
            **if** $p_c = p_e$ **and** $c_c = c_e$ **then**
                add $(s_e, n_c)$ to $output$.
            **end if**
        **end for**
    **end for**
    **return** $output$
```

```
**Algorithm** ECJOIN(courses, enrolled)
    $output := \emptyset$.
    **for** $(p_e, c_e, s_e) \in$ enrolled **do**
        **for** $(p_c, c_c, n_c) \in$ courses **do**
            **if** $p_c = p_e$ **and** $c_c = c_e$ **then**
                add $(s_e, n_c)$ to $output$.
            **end if**
        **end for**
    **end for**
    **return** $output$
```

Assume we have significantly more students enrolled for courses than courses (`|enrolled| > |courses|`).

P2.1

Assume we are running algorithm `CEJOIN` and `ECJOIN` on a computer $\mathbb{C}$ where *every* instruction takes *exactly the same* amount of time to execute. Argue why `CEJOIN` must be faster than `ECJOIN` when running on computer $\mathbb{C}$?

Solution

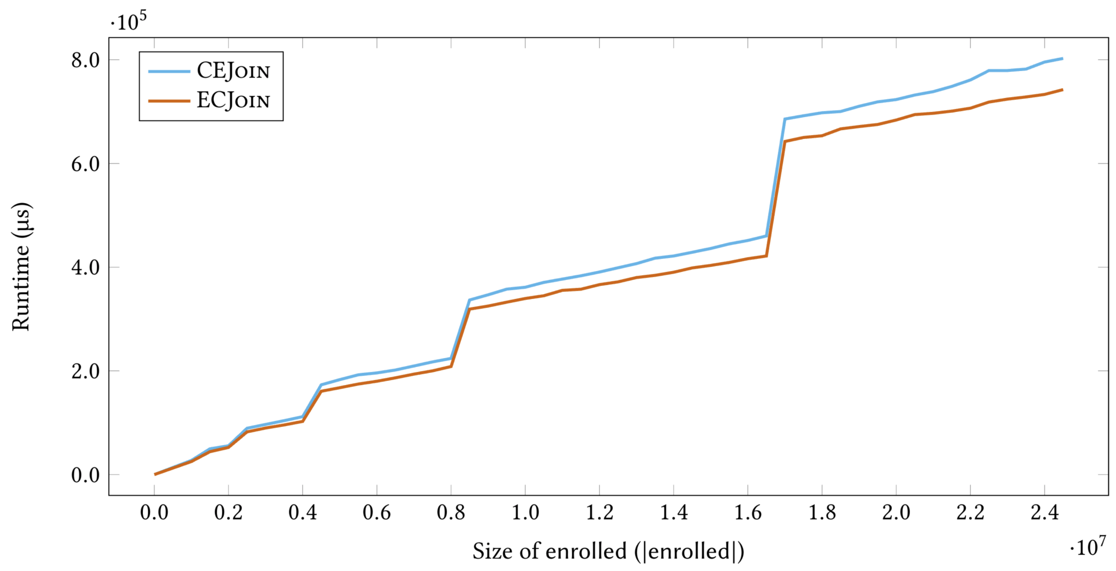
Given $|enrolled| > |courses|$, the difference between the two lies in the number of iterations of the inner loop.

The outer loop table will only be scanned once, and the inner loop will be scanned for each iteration of the outer loop in the nested-loop algorithm:

- `CEJOIN` will iterate over `enrolled` once, and `courses` over each iteration. Since `|enrolled| > |courses|`, `CEJOIN`'s inner loop will result in fewer iteration.
- `ECJOIN` will iterate over `courses` once, and `enrolled` over each iteration. Since `|enrolled| > |courses|`, `ECJOIN`'s inner loop will means more iterations, comparing to `CEJOIN`.

Thus, we can conclude that `CEJOIN` must be faster than `ECJOIN` when running on computer \mathbb{C

P2.2



Implementation of `CEJOIN` and `ECJOIN` in impl_22.cpp shows that `ECJOIN` is actually faster than `CEJOIN`. Explain why this is the case in a real-world system.

Solution

Given the processor is using fast *caches*, the inner loop of `ECJOIN` will be faster than `CEJOIN` due to *cache locality*.

Since `|enrolled| > |courses|`, the inner loop of `ECJOIN` will have better cache locality, as it will be accessing the same memory locations more frequently. Additionally, `ECJOIN`'s outer loop will only run once, therefore we can expect `ECJOIN` to be faster comparing to `CEJOIN`, since `CEJOIN` will have to access `enrolled` each iteration, which would be slower comparing to accessing `courses` (since every item in `enrolled` might not be cached).

P2.3

The measurements in the above figure have a few sudden jumps, e.g., at 1 500 000, 2 500 000, 4 500 000, 8 500 000, and 17 000 000. Explain what causes these jumps.

Solution

These jumps are probably caused by the *cache size* and *cache associativity*.

As the data size increases, the number of elements might not fit into processor's L1, L2, L3 cache, thus, causing more frequent cache misses. At these points, the dataset is so large such that the processor might have to access on slower main memory, which induces these bump we observed from the graph.

We can also expect context-switching overhead given the code might be running in multiple processes or threads. However, the implementation implies that this code is running in a single thread, so we can rule out context-switching overhead.

P2.4

Write an algorithm that efficiently computes the same result as `CEJOIN` and `ECJOIN` in all-case $\Theta(|enrolled| \log_2(|courses|))$. In the design of your algorithm, you may require that either enrolled or courses is ordered. Argue why your algorithm is correct and why it has the runtime complexity we specified.

Solution

Assuming `courses` is sorted and ordered by `(prog, code)` pair, the following algorithm can be used:

```
**Algorithm** EFFJOIN(courses, enrolled)
    $output := \emptyset.$
    $course :=$ sort$(course,$ by $= (p_c, c_c))$
    **for** $(p_e, c_e, n_e) \in$ enrolled **do**
        $i :=$ binary-search$(courses, (p_e, c_e))$
        **if** $i \neq$ null **then**
            add $(s_e, n_c)$ to $output.$
        **end if**
    **end for**
    **return** $output$
```

With the following binary search algorithm:

```
**Algorithm** binary-search(course, (p, c))
    $l := 0$
    $r :=$ len$(course) - 1$
    **while** $l \leq r$ **do**
        $m := l + (r - l)/2$
        **if** $course[m].p_c = p$ **and** $course[m].c_c = c$ **then**
            **return** $course[m]$
        **else if** $course[m].p_c < p$ **or** $(course[m].p_c =$
        $p$ **and** $course[m].c_c < c)$ **then**
            $l := m + 1$
        **else**
            $r := m - 1$
        **end if**
    **end while**
    **return** null
```

In all cases this would yield a runtime complexity of
$\Theta(|enrolled| \log_2{(|courses|)})$

Correctness

1. Sorting `course`, has time complexity of $\Theta(|courses| \log_2{|courses|})$, which will be executed once.
2. The binary search has time complexity of $\Theta(\log_2{|courses|})$, which will be executed for each iteration of `enrolled`.
3. The for loop iterate each element in `enrolled` once, which has time complexity of $\Theta(|enrolled|)$.

Thus, the total time complexity is $\Theta(|enrolled| \log_2{(|courses|)})$.