1 Buffer Replacement Strategies (7 Points)

A database buffer is a component of a DBMS that explicitly manages the buffering of disk pages in main memory and it implements the common buffer interface of `get(pageID)`, `load(pageID)`, and `evict()`. In this exercise you are given a sequence of page accesses caused by some DBMS workload and a DBMS buffer of three disk pages.

1. Your task is to write down the actions performed by the buffer as well as its intermediate states. As buffer replacement strategies you will use

   (a) First-In, First-Out (FIFO): the data item that was *loaded the longest time ago* will be evicted (3 Points)

   (b) Least Frequently Used (LFU): the data item that was *used the least* will be evicted (3 Points)

   This means you have to perform the task once per buffer replacement strategy, so two times in total. Below is the list of page accesses. Note that our buffer makes no distinction between read and write accesses.

```
1 read  p2
2 write p3
3 write p2
4 read  p1
5 read  p0
6 write p4
7 write p1
8 write p3
9 read  p0
10 read  p1
```

   Use the following notation to write down the actions and the state of the buffer. As an example we will use the FIFO replacement strategy as in (a).

Page Access	Actions	Buffer
-	-	$\top$, $\top$, $\top$
`read p2`	`load(p2)`	p_2, $\top$, $\top$
	`get(p2)`	p_2, $\top$, $\top$
`write p3`	`load(p3)`	p_2, p_3, $\top$
	`get(p3)`	p_2, p_3, $\top$
`write p2`	`get(p2)`	p_2, p_3, $\top$
`read p1`	`load(p1)`	p_2, p_3, p_1
	`get(p1)`	p_2, p_3, p_1
`read p0`	`evict()`	p_3, p_1, $\top$
	`load(p0)`	p_3, p_1, p_0
	`get(p0)`	p_3, p_1, p_0
...		

   When writing down your solution using FIFO, it is a good idea to keep the oldest page at the front of the buffer. For LFU you should write down some additional information that helps to identify which page was least frequently used and will be evicted next. If two pages have been accessed equally often, the older page in the buffer is evicted first.

2. The quality of a buffer replacement strategy can be measured by the *miss ratio*, that is the number of page accesses that triggered a `load` (called *misses*) divided by the number of total page accesses.

Smaller miss ratios mean less disk accesses and result in improved performance for the DBMS. Calculate the miss ratio for each replacement strategy and the given workload in Exercise 1.1.

(1 Point)

2 Compression (5 Points)

In the lecture, we looked at data transfer with and without compression and learned about the potential benefits of using compression.

1. In this context, we learned that compressing data to save bandwidth only makes sense if $T_{trc}(n) < T_{truc}(n)$. Compute the minimal compression speed $S_{compress}$, i.e. the speed with which the original data can be compressed, such that compressing the data is worthwhile. The time required to compress data of size n is given by $T_{compress}(n) = \frac{n}{S_{compress}}$. Assume the following: (2.5 Points)

   - Original data size $n = 5$ GByte
   - Data compression ratio $\frac{7}{2}$
   - Bandwidth $BW = 570$ MByte/s
   - Decompression speed $S_{decompress} = 2800$ MByte/s

2. In our formula to compute the time for compressed transfer T_{trc}, we assumed that compression and decompression must not overlap with the transfer. Most compression algorithms used in practice[1], however, support streaming and hence enable overlapping compression and decompression with the transfer.

   The formula to compute the transfer time with streamable compression is

   $$T_{trs}(n) = \max\left(T_{compress}(n), T_{tr}(n_c), T_{decompress}(n_c)\right)$$

   Calculate for the following scenario the transfer time with and without compression and clearly state which option yields the lowest total transfer time (including compression and decompression). (2.5 Points)

   - Original data size $n = 3$ GByte
   - Data compression ratio $\frac{10}{3}$
   - Bandwidth $BW = 400$ MByte/s
   - Compression speed $S_{compression} = 450$ MByte/s
   - Decompression speed $S_{decompression} = 1370$ MByte/s

   Note: For simplicity we assume 1 GByte = 1000 MByte.

3 Jupyter Notebook: Buffer Implementation (3 Points)

In the lecture, your learned about database buffers and different buffer replacement strategies. In addition, you were shown a Jupyter Notebook illustrating the functionality of a buffer using Least Recently Used (LRU) as eviction strategy.

There are serveral other buffer replacement strategies in addition to the ones discussed in the lecture. For example, Most Recently Used (MRU) is a recency-based policy that evicts the data item that was *used most recently*.

In this exercise, we ask you to implement a buffer with **MRU** as the replacement strategy. For this, implement the corresponding `load(pageID)`, `get(pos)`, and `evict()` methods of the `MRUBuffer` class in `assignment02.ipynb`.

[1] `zlib`, `bzip2`, `lz4`, `zstd`, `brotli`, `lzma2`, etc.

4 Horizontal Partitioning & Indexing (5 Points)

Table 1 shows a table of movies taken from the IMDb website. Each movie has an associated `movie_id` that uniquely identifies the tuple. In addition, the table contains information about the `title`, `director`, `year`, and `rating` of the movies.

movie_id	title	director	year	rating
1	The Shawshank Redemption	Frank Darabont	1994	9.2
2	Top Gun: Maverick	Joseph Kosinski	2022	8.3
3	Inception	Christopher Nolan	2010	8.8
4	A Beatiful Mind	Ron Howard	2001	8.2
5	Ratatouille	Brad Bird	2007	8.1
6	Django Unchained	Quentin Tarantino	2012	8.4
7	Harakiri	Masaki Kobayashi	1962	8.6
8	Die Hard	John McTiernan	1988	8.2
9	Coco	Lee Unkrich	2017	8.4
10	Kill Bill: Vol. 1	Quentin Tarantino	2003	8.2

Table 1: Movies table.

1. In this exercise, you will apply partitioning functions to the table to compute horizontal partitions. To save space and work, you can abbreviate a tuple with its `movie_id`. (2 Points)
   (a) $p(t) := t.\text{year} < 2000$
   (b) $p(t) := \text{round}(t.\text{rating})$ (The nearest integer.)
   (c) $p(t) := t.\text{title}[0]$ (The first character of the title.)
   (d) $p(t) := t.\text{year} \mod 3$
2. Assume we have the two indexes in Figure 1 and Figure 2. Note that the tuples in both indexes are abbreviated using the unique `movie_id`. Evaluate the performance of both indexes by executing the following point queries. Assume the performance of an index is based on the number of indirections, i.e. the number of arrows on the way to the element(s) we search for. (2 Points)
   (a) $\sigma_{\texttt{year}=2012}(T)$
   (b) $\sigma_{\texttt{year}=2017}(T)$
   (c) $\sigma_{\texttt{year}=1999}(T)$
   (d) $\sigma_{\texttt{year}=2001}(T)$
3. Based on your findings in 2, argue which of the indexes should be preferred for point queries. (1 Point)

Submission Details:

Upload your submission to our CMS until *November 30, 2022 at 23:59*. Late submissions will not be graded! Submit your solution to each exercise separately on the CMS: submit **a single *pdf* file** to each of exercises 1, 2, 4, and a Jupyter Notebook (**just the *.ipynb* file**) containing your solution to exercise 3. Please put the *names and matriculation numbers* of your team members on each submited pdf.

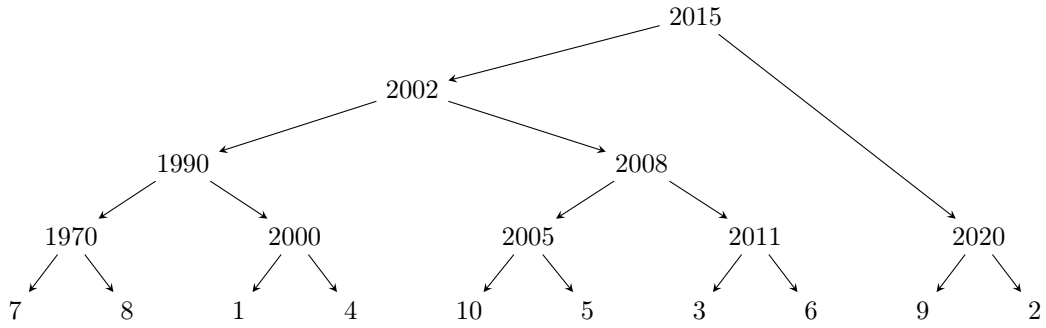

Figure 1: A tree index on `year`.

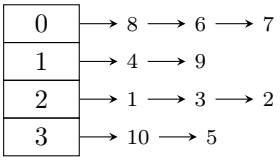

Figure 2: A hash index on `year` created using the partitioning function $p(t) := t.\text{year} \mod 4$.