SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

Systemarchitektur SS 2021

Präsenzblatt 8 (Lösungsvorschläge)

Hinweis: **Dieses Aufgabenblatt wurde von Tutoren erstellt. Die Aufgaben sind für die Klausur weder relevant noch irrelevant, die Lösungsvorschlage weder korrekt noch inkorrekt.**

Aufgabe 8.1: Matrix im Cache

In dieser Aufgabe wollen wir die Auswirkungen von Caches auf die Geschwindigkeit eines Programms in der Realität beobachten.

Betrachten Sie dazu den folgenden Ausschnitt eines C-Programms:

```
...
int A[n][m];
int B[m][p];
int C[n][p];
...
for (int i = 0; i < n; i++)
    for (int j = 0; j < p; j++)
        for (int k = 0; k < m; k++)
            C[i][j] += A[i][k] * B[k][j];
...
```

a) Was berechnet dieses Programm?

b) Analysieren Sie die Komplexität der Operation. Wie entwickelt sich die Ausführungszeit im Bezug auf die Werte `n`, `m` und `p`?

c) An welchen Stellen wird auf den Speicher zugegriffen? Wie viele Speicherzugriffe werden insgesamt ausgeführt?

d) Betrachten Sie diese Speicherzugriffe nun im Hinblick auf Cache-Nutzung. An welchen Stellen wird die Annahme der räumlichen und/oder zeitlichen Lokalität erfüllt bzw. verletzt?

e) Können Sie den Code so abändern, dass er bei gleicher Komplexität schneller ausgeführt werden kann?

Lösungsvorschlag:

a) Es berechnet die Matrixmultiplikation $C_{ij} = (AB)_{ij} = \sum_{k=0}^{m-1} A_{ik} \cdot B_{kj}$.

b) Durch die drei For-Schleifen ist die Ausführungszeit proportional zu $n \cdot p \cdot m$.

c) Auf den Speicher wird nur in der untersten Zeile zugegriffen, die Variablen der For-Schleife können direkt in Registern verwaltet werden. Pro Ausführung dieser Zeile benötigt man höchstens vier Speicherzugriffe (Lesen von A_{ik},B_{kj} und C_{ij}, sowie Überschreiben von C_{ij}). Damit werden insgesamt höchstens $n \cdot p \cdot m \cdot 4$ Speicherzugriffe benötigt.

d) Zeitliche Lokalität bei C_{ij} und räumliche Lokalität bei A_{ik}. Bei B_{kj} wird keine der Annahmen erfüllt, sodass aufeinanderfolgende Zugriffe auf verschiedene Einträge der Spalte j zu Cache-Misses führen.

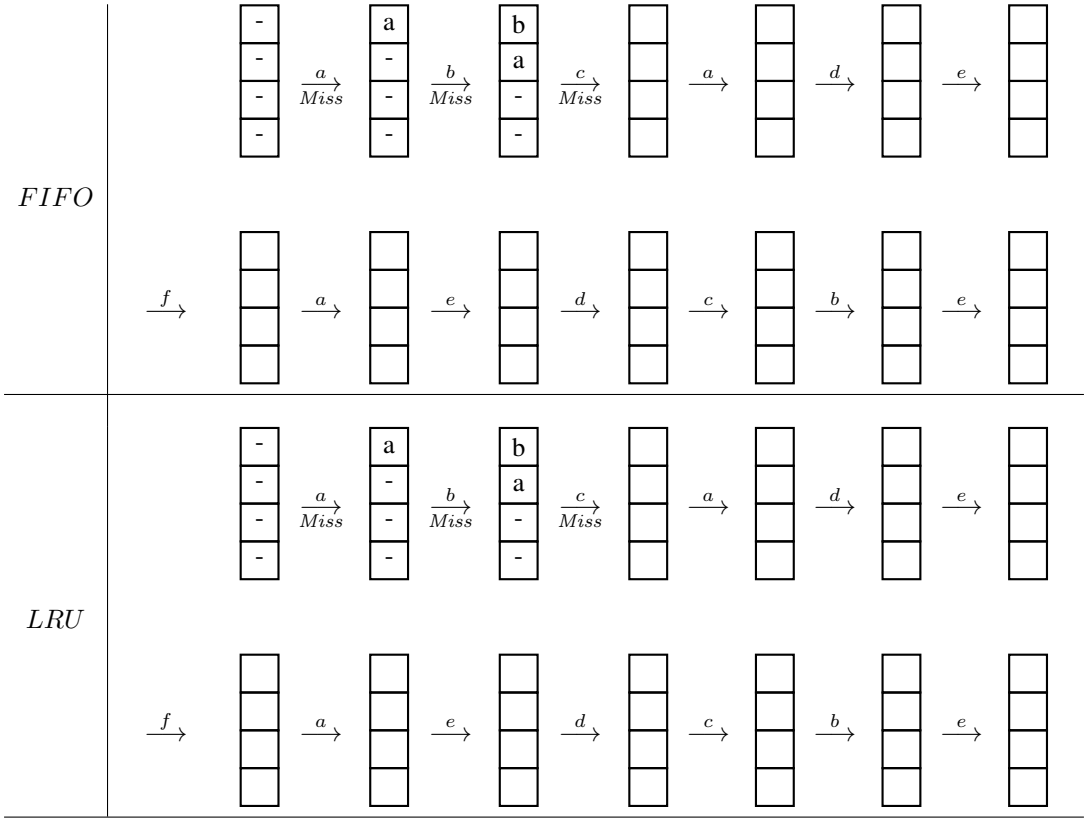
e) Wir vertauschen die unteren zwei For-Schleifen. Dadurch wird nun auch bei B_{kj} räumliche Lokalität ausgenutzt. Wir gehen hier davon aus, dass `matrixB[k]` ein langes Array im Hauptspeicher ist. Im Gegensatz zu vorher halten wir nun `k` fest und zählen `j` hoch. Da wir nicht nur einzelne Bytes sondern ganze Speicherblöcke in den Cache laden, haben wir eine also eine gute Chance, dass nach dem Laden von B_{kj} auch $B_{k(j+1)}$, $B_{k(j+2)}$, usw. bereits im Cache vorhanden sind. Damit wird die Anzahl der Cache-Misses reduziert und das Programm läuft schneller.

Aufgabe 8.2: Ersetzungsstrategien

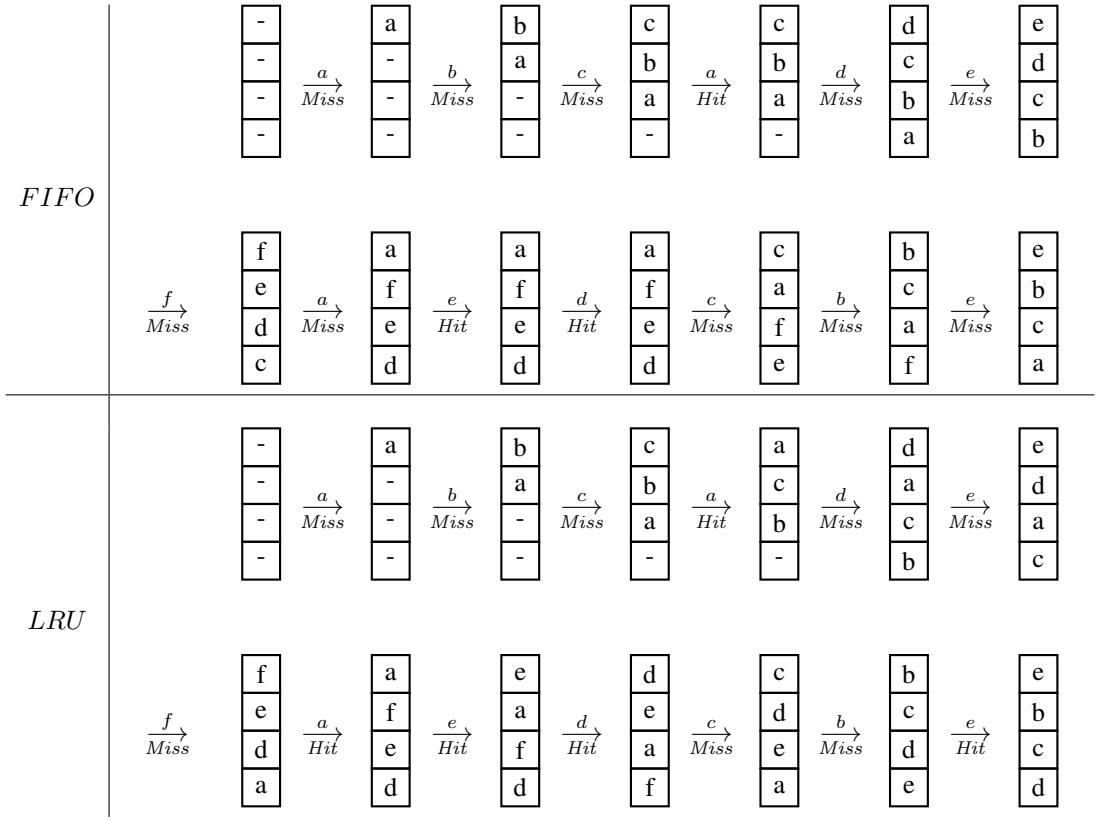
Betrachten Sie folgende Zugriffssequenz von Speicherblöcken:

a b c a d e f a e d c b e

Berechnen Sie die Cache-Inhalte nach Anwendung der Cache-Ersetzungsstrategien FIFO und LRU ausgehend von einem leeren, vollassoziativen Cache der Größe 4, indem Sie die folgende Grafik vervollständigen. Geben Sie jeweils an, ob der Zugriff ein Hit oder Miss war.



Lösungsvorschlag:



Aufgabe 8.3: Pseudo-LRU

Zur Realisierung der LRU-Ersetzungsstrategie für einen k-fach assoziativen Cache sind mindestens $\log(k!)$ Bits nötig um die Reihenfolge der Cacheelemente bezüglich ihres letzten Zugriffs zu speichern. In der Praxis wird daher häufig auf Pseudo-LRU zurückgegriffen, das sich ähnlich zu LRU verhält, aber lediglich $k-1$ Bits benötigt.

Zur Veranschaulichung der Funktionsweise der Pseudo-LRU-Strategie nehmen wir an, dass diese Bits, wie in Abbildung 1 dargestellt, baumartig angeordnet sind. Hierdurch wird ein Pfad impliziert, der zum nächsten zu ersetzenden Cacheelement führt. Eine 1 (0) bedeutet hierbei, dass der rechte (linke) Zweig genommen wird.

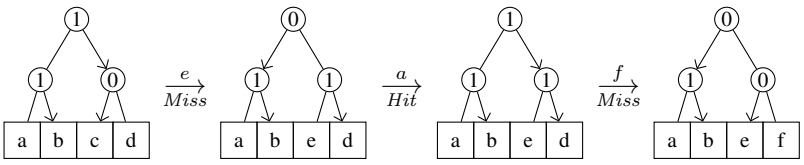


Abbildung 1: Verhalten eines 4-fach assoziativen PLRU Caches für die Zugriffsequenz e, a, f.

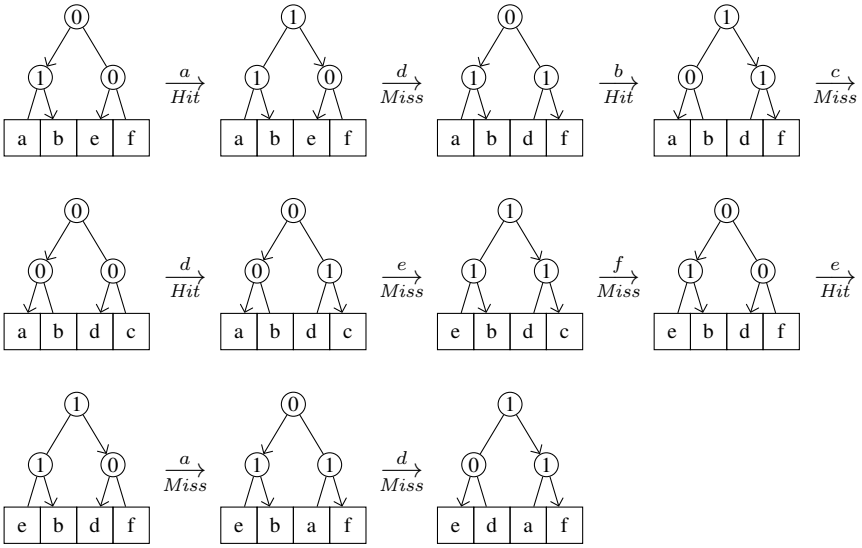
Nach einem Zugriff (sowohl Hit als auch Miss) werden die Bits auf dem Zugriffspfad so angepasst, dass sie jeweils vom zugegriffenen Element wegzeigen. In Abbildung 1 sind insgesamt drei solcher Zugriffe dargestellt.

Ausgehend von dem rechten PLRU-Cachezustand in Abbildung 2 soll die Zugriffsequenz

adbcdefead

abgearbeitet werden. Geben Sie die jeweiligen Zwischenschritte (Cachezustände) explizit an.

Lösungsvorschlag:



Aufgabe 8.4: Bonus: Least Frequently Used

LFU (*least frequently used*) ist ein Algorithmus, bei dem zu jedem Element in einem Cache gespeichert wird, wie oft dieses gelesen wurde. Wenn der Cache voll ist, wird das Element ersetzt, auf das am wenigsten zugegriffen wurde.

1. Worin liegen Vor- und Nachteile von LFU?
2. Überlegen Sie sich, wie LFU implementiert werden könnte. Sehen Sie hier weitere Vor- bzw. Nachteile von LFU?

Lösungsvorschlag:

1. Ein Vorteil von LFU ist, dass, wenn ein Element oft gebraucht wird, es nicht aus dem Cache gelöscht wird, weil es in letzter Zeit nicht gebraucht wurde.
   Ein Nachteil von LFU ist, dass wenn ein Element anfangs oft gebraucht wurde und später nicht mehr, dass es dann eher nicht mehr aus dem Cache verschwindet, obwohl es nicht mehr gebraucht wird.
2. Eine naive Implementierung:
   Man speichert für jedes Element die Anzahl der Zugriffe als Integer. Wenn auf das Element zugegriffen wurde, wird diese Zahl um 1 erhöht. Um das Element zu finden, welches ersetzt wird, muss jedes Element betrachtet werden und der Integerwert verglichen werden, was lineare Laufzeit im Vergleich zur Cachegröße hat und nicht besonderes schnell ist. Ein weiterer Nachteil ist, dass ein neues Element schlechte Überlebenschancen hat, wenn der Cache voll ist, da es anfangs nur einmal verwendet wurde und wenn kurz darauf ein neues Element in den Cache will, dieses als erstes ersetzt wird.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

System Architecture SS 2021

Tutorial Sheet 8 (Suggested Solutions)

Note: **This task sheet was created by tutors. The tasks are neither relevant nor irrelevant for the exam, the suggested solutions are neither correct nor incorrect.**

Problem 8.1: Matrix in cache

In this exercise we want to judge the effects of caches on the speed of a program in reality.
Consider the following part of a C-program.

```
...
int A[n][m];
int B[m][p];
int C[n][p];
...
for (int i = 0; i < n; i++)
    for (int j = 0; j < p; j++)
        for (int k = 0; k < m; k++)
            C[i][j] += A[i][k] * B[k][j];
...
```

a) What does this program calculate?

b) Analyse its complexity. What is the relation between the execution time change and the values `n`, `m` and `p`?

c) Where in the program is the memory accessed? How many memory accesses are executed in total?

d) Now consider these memory accesses with respect to cache usage. Where is the assumption of spatial and/or temporal locality fulfilled/violated?

e) Can you modify the code so that it can be executed faster while keeping the same complexity?

Suggested solution:

a) It calculates the matrix multiplication$C_{ij} = (AB)_{ij} = \sum_{k=0}^{m-1} A_{ik} \cdot B_{kj}$.

b) Because of the three for-loops, the execution time is proportional to $n \cdot p \cdot m$.

c) Only the bottom most line accesses the memory, the variables of the for-loops can be stored in registers. For each execution of this line we need at most four memory accesses (Reading of A_{ik}, B_{kj} and C_{ij}, and rewriting of C_{ij}). Thus, in total we need at most $n \cdot p \cdot m \cdot 4$ memory accesses.

d) Temporal locality in C_{ij} and spatial locality in A_{ik}. In B_{kj} none of the assumptions are fulfilled, so that serial accesses to different entries in the row j lead to cache misses.

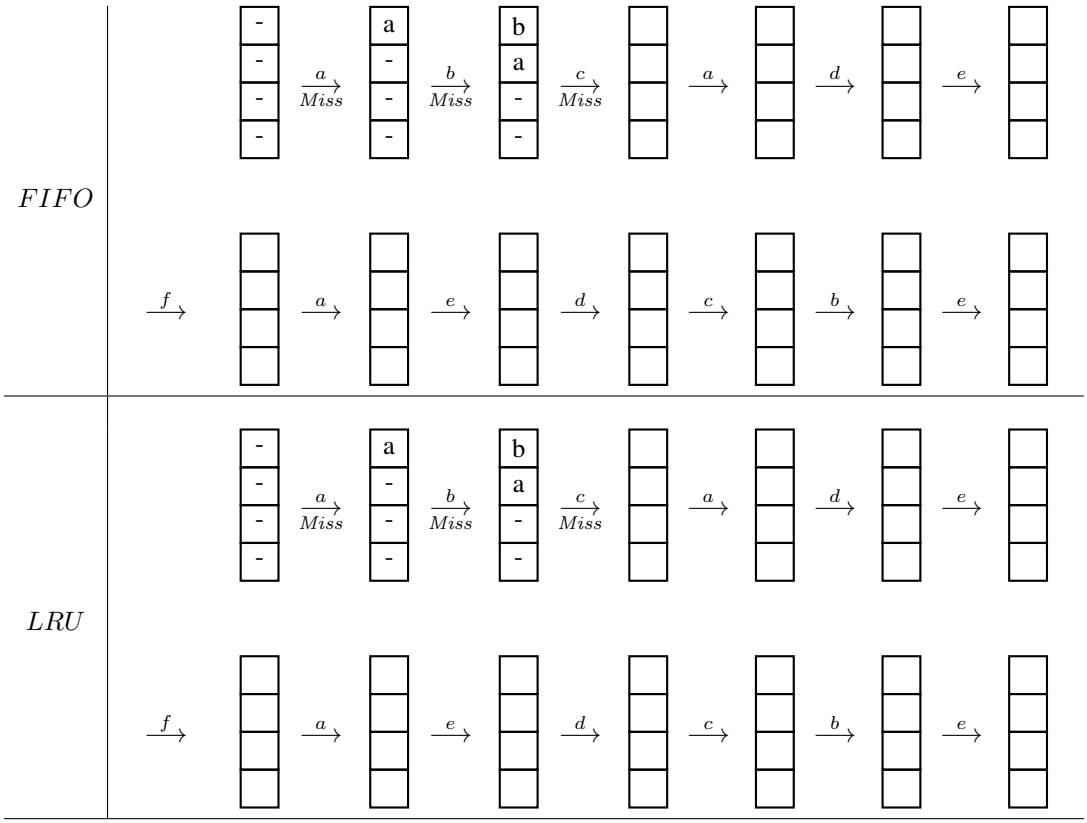
e) We exchange the two for-loops at the bottom. That way we also utilize spatial locality in B_{kj}. We assume that `matrixB[k]` is a long array in the main memory. Opposed to before we now fix `k` and count `j`. Since we don not just load single bytes, but entire memory blocks into the cache, we now have a decent chance that after loading B_{kj}, $B_{k(j+1)}$, $B_{k(j+2)}$ are also already in the cache. Thus, we reduce the number of cache misses and the program executes faster.

Problem 8.2: Replacement strategies

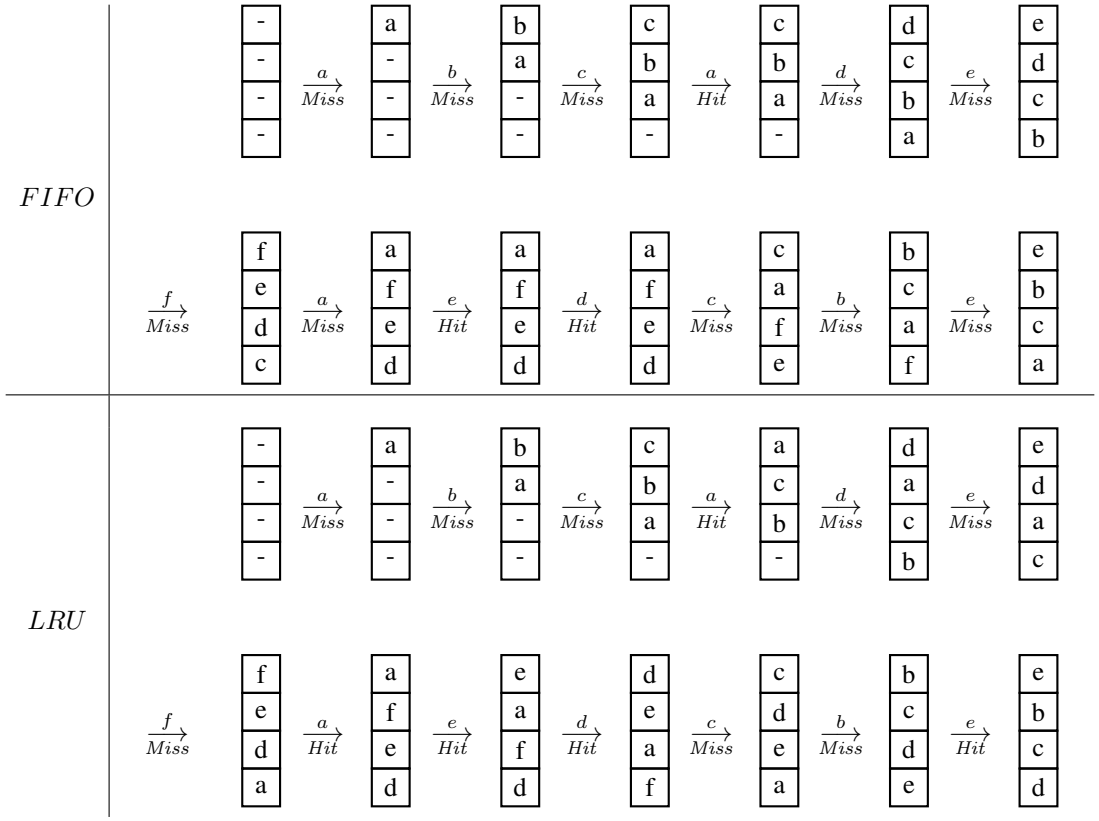
Consider the following access sequence for memory blocks:

a b c a d e f a e d c b e

Calculate the cache entries when utilizing the replacement strategies FIFO and LRU (respectively), starting with an empty, fully-associative cache of size 4. Determine for every access whether it is a hit or a miss.



Suggested solution:



Problem 8.3: Pseudo-LRU

In order to realise the LRU-replacement policy in a k-times associative cache at least $\log(k!)$ bits are required (for storing the order of the cache elements with respect to to their last access). Because of that, pseudo-LRU is often used in praxis, which behaves similarly to LRU but only needs $k-1$ bits.

To visualize the way pseudo-LRU operates, we assume that these bits (as shown in picture 1) are stored in a tree-like manner. This implies a path, which leads to the cached element that should be replaced. A 1 (0) indicates that the right (left) branch is taken.

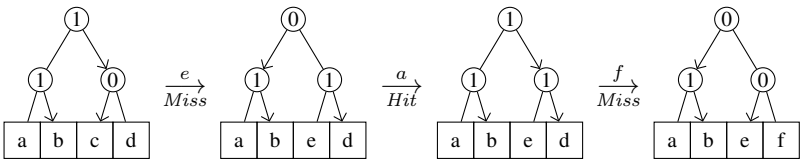


Abbildung 2: Behaviour of a 4-times associative PLRU cache for the access sequence e, a, f.

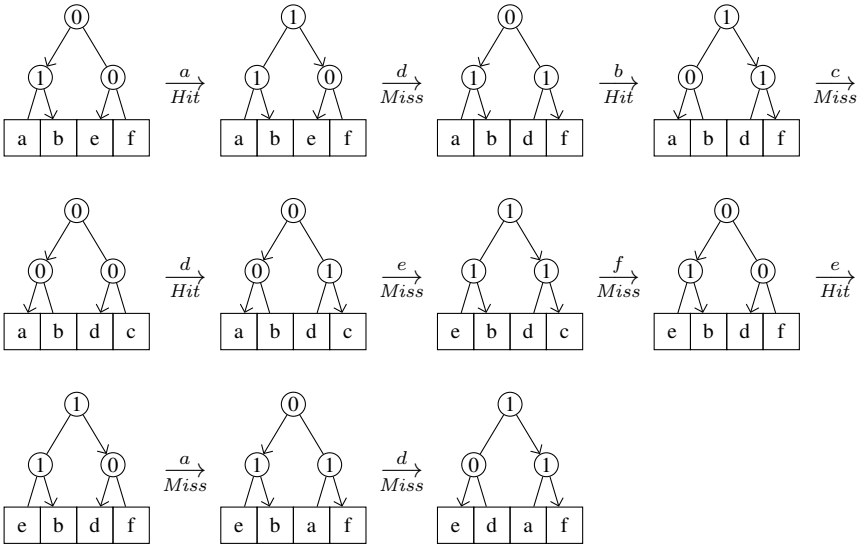
After an access (either hit or miss) the bits on the access path are changed such that they lead away from the accessed element. In picture 1 there are three such accesses.

Starting with the right PLRU-cache state in the picture 2 execute the following access sequence

adbcdefead

Write down each partial result explicitly.

Suggested solution:



Problem 8.4: Bonus: Least Frequently Used

LFU (*least frequently used*) is an algorithm that stores for each element in a cache how often it was accessed. When the cache is full, the element which was accessed the least is replaced.

1. What are advantages and disadvantages of LFU?
2. How could LFU be implemented? Can you find additional advantages or disadvantages of LFU?

Suggested solution:

1. An advantage is that an element that is used often will not be replaced because it was not used recently. A disadvantage is that, if an element is first a lot at first and not at all later, it will not disappear from the cache, although it is no longer used.
2. A naive implementation:
   For each element we store the number of accesses as an integer. When an element is accessed, this number will be increased by one. To find the element to be replaced, we have to check each element and compare their integer values. This has linear runtime with respect to cache size and is not very fast. Another disadvantage is that a new element has bad survival chances when the cache is already full: Since it has only been accessed once, it is replaced when another element enters the cache.