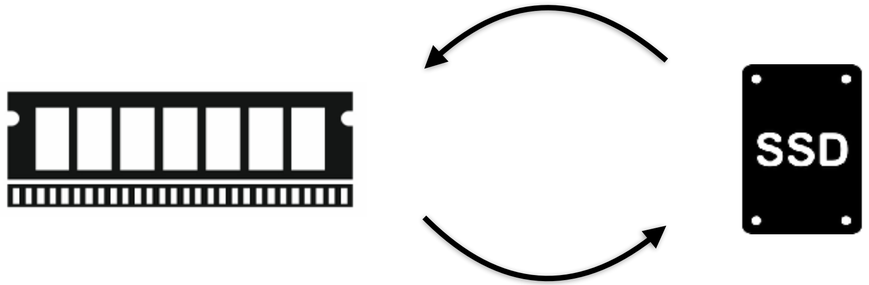
Buffer Management



Database System Technology - Lecture 3, Chapter 9

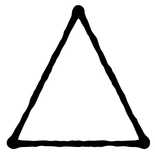
Niv Dayan

Storage

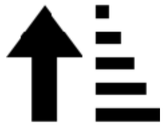
Tables

Buffering

Indexes



Sorting



Operators

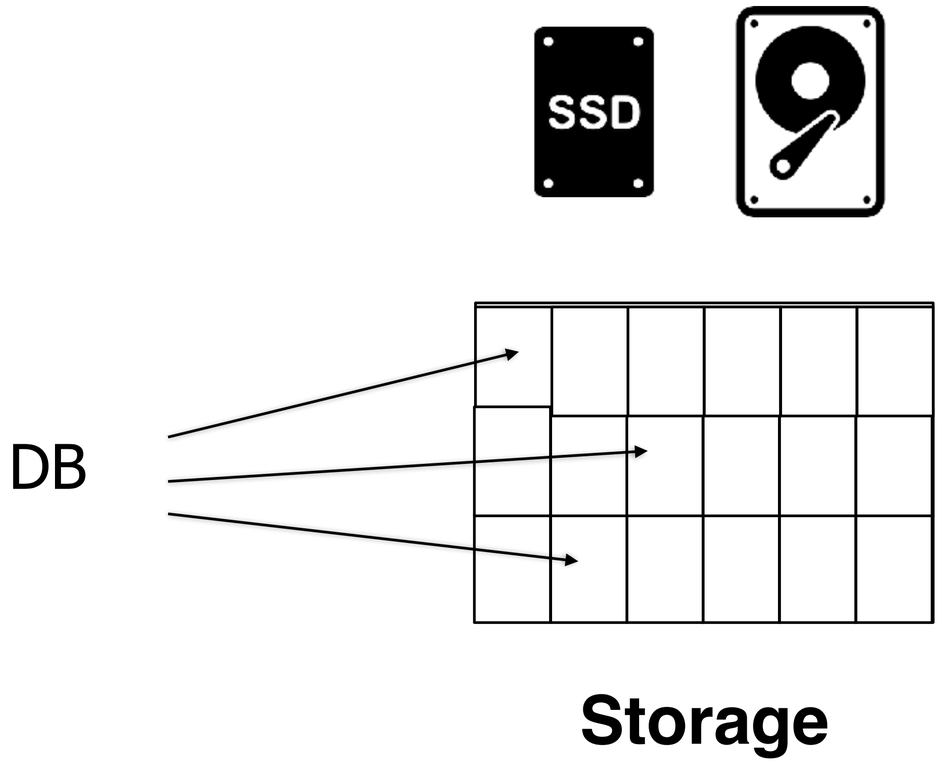
Query Optimization

Transactions

Recovery

Context

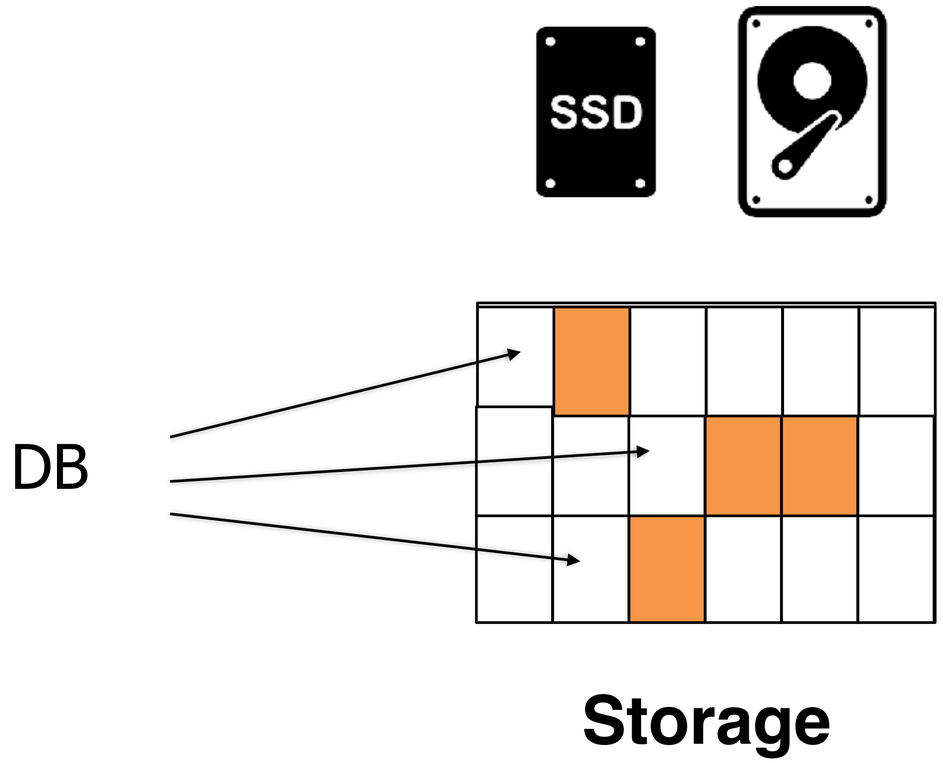
A DB is reading and writing aligned 4KB storage pages



Context

A DB is reading and writing aligned 4KB storage pages

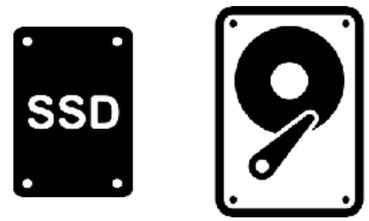
Suppose orange pages are frequently accessed (“hot”)

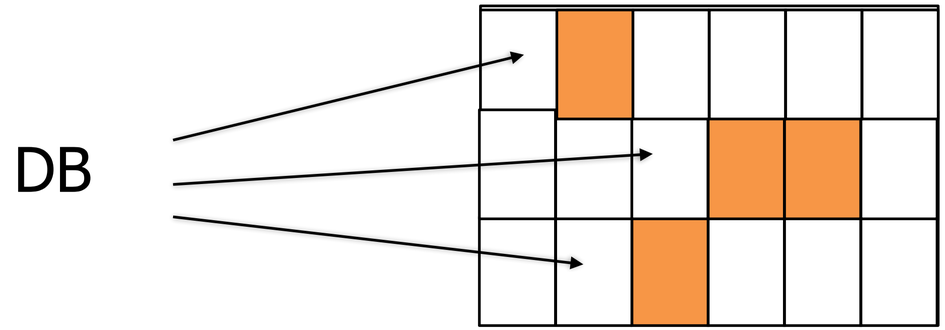


Context

A DB is reading and writing aligned 4KB storage pages

Suppose orange pages are frequently accessed (“hot”)



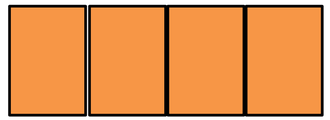


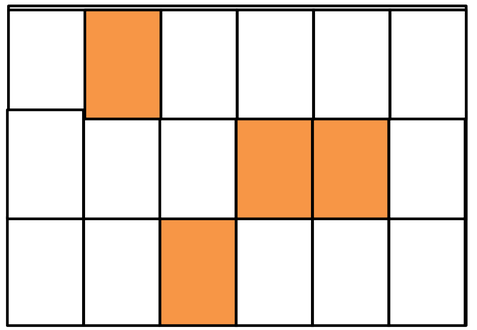
Retrieving these pages over and over is expensive!

Buffer Pool

Keep copies of hot pages in memory

DB 





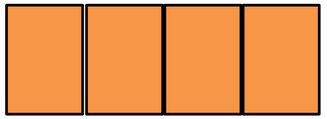
memory

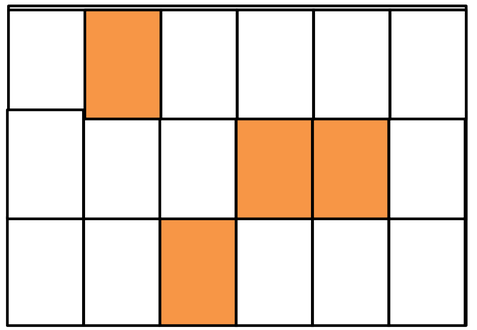
Storage

Buffer Pool

Question 1:

How to structure this buffer pool?





memory

Storage

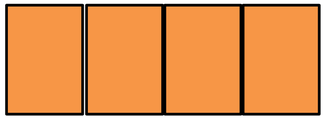
Buffer Pool

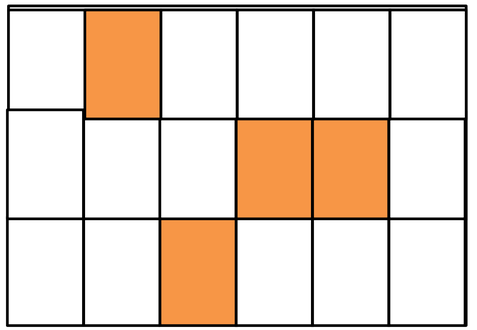
Question 1:

How to structure this buffer pool?

hash table of your choice

e.g., chaining, linear probing, etc.



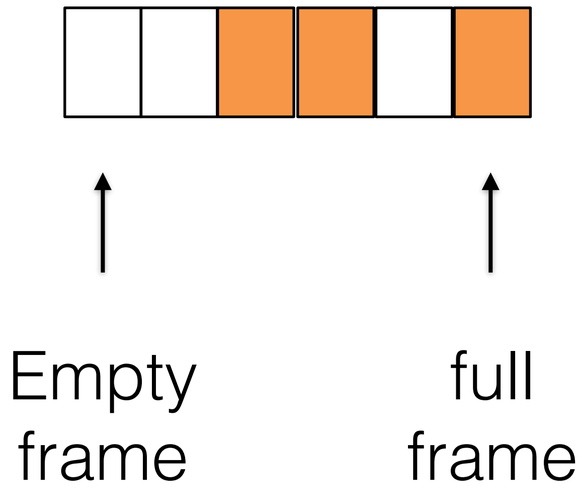


memory

Storage

Buffer Pool

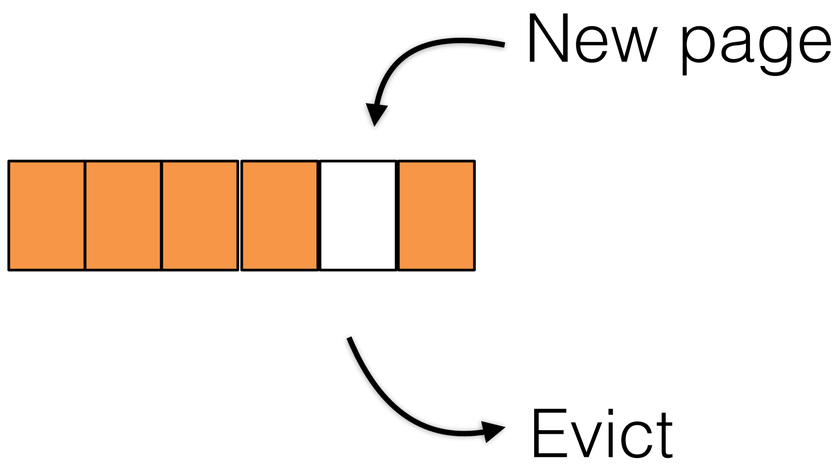
Consist of frames, each containing one page of data (e.g., 4 KB)



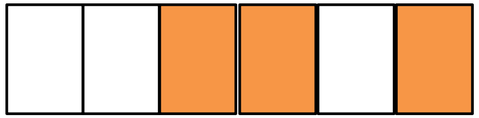
Buffer Pool

Consist of frames, each containing one page of data (e.g., 4 KB)

Eventually it fills up. Must evict pages to clear space.



Buffer Pool

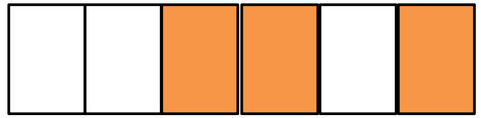


Each frame must keep some metadata

(1) ?

(2) ?

Buffer Pool



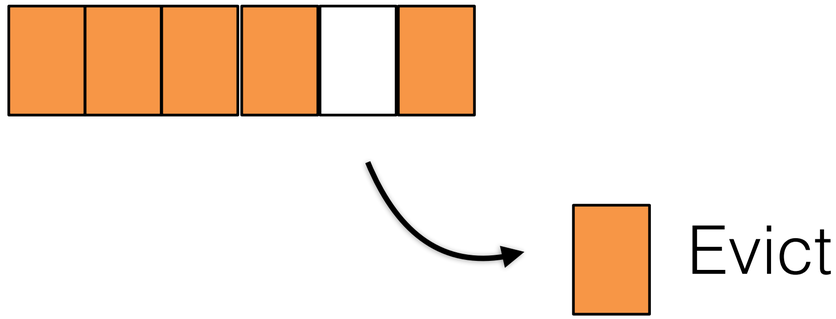
Each frame must keep some metadata

(1) **Pin count** - How many users are currently using this page

(2) **Dirty flag** - indicates whether the page has been updated

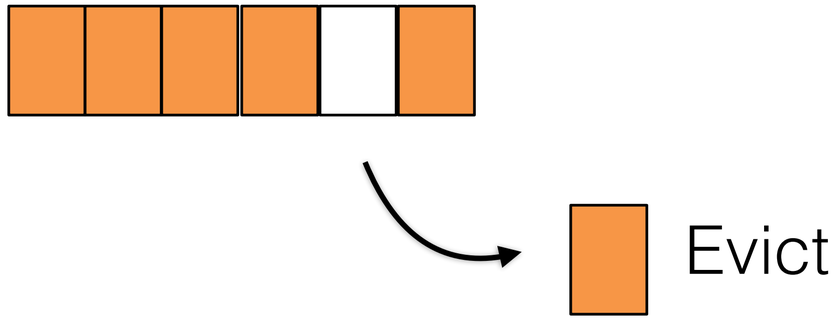
Eviction Policy

Which page to evict when we run out of space?



Eviction Policy

Which page to evict when we run out of space?



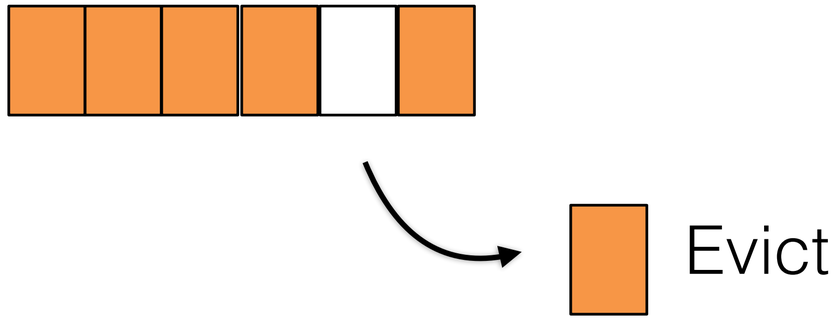
Considerations:

(1) Avoid evicting a page that is likely to be used again

(2) Avoid excessive metadata or CPU overheads to make decision

Eviction Policy

Which page to evict when we run out of space?



Big impact on number of I/Os and CPU efficiency

Depends on the access pattern

We’ll cover 5 eviction policies

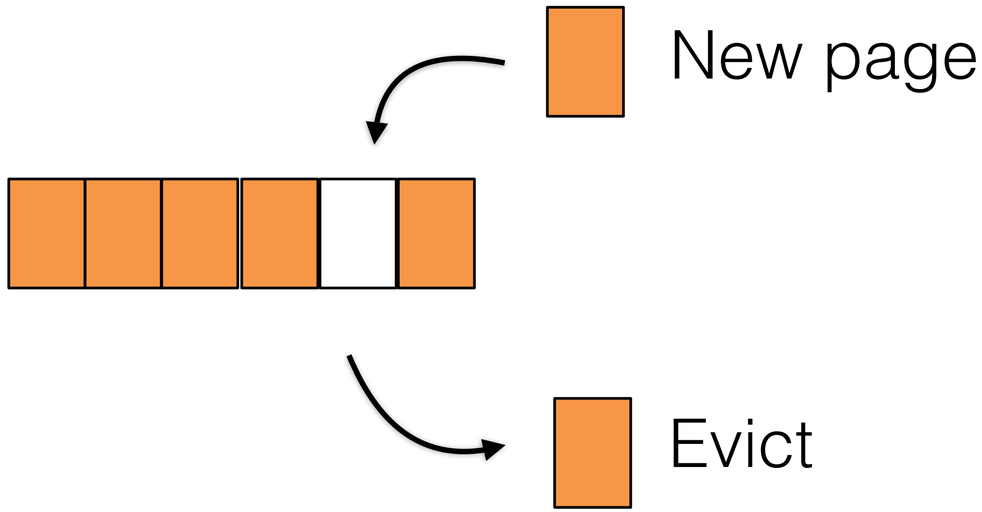
Random **FIFO** **LRU** **Clock** **MRU**

Random Eviction

Evict whichever page collides in the hash table with a new page

Pro: ?

Con: ?

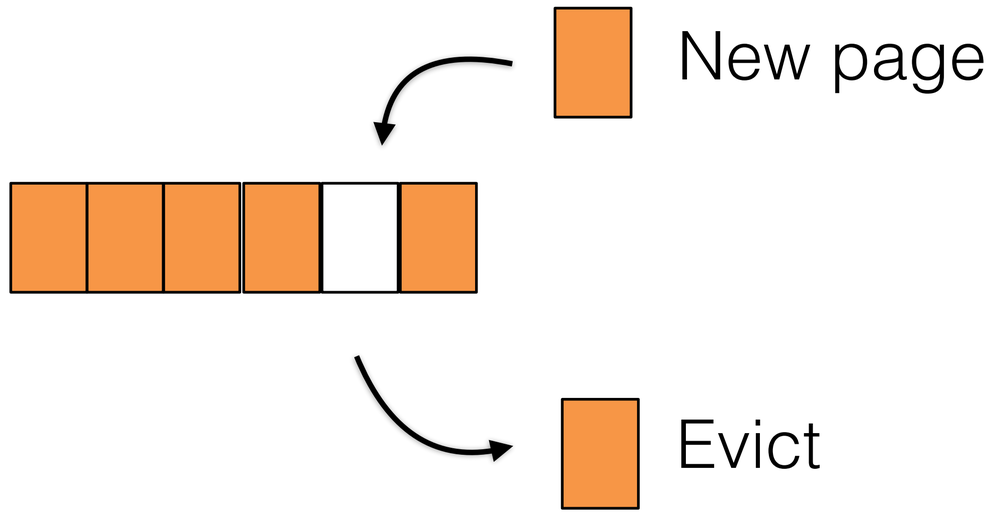


Random Eviction

Evict whichever page collides in the hash table with a new page

Pro: No additional metadata needed

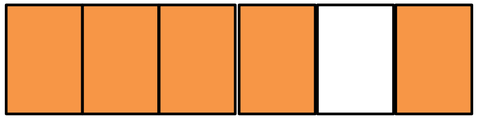
Con: May evict a frequently used page



First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Rationale?

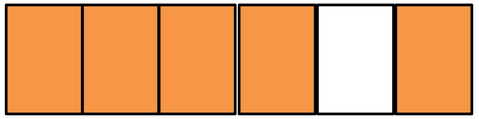


First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Rationale? Less likely to be used again

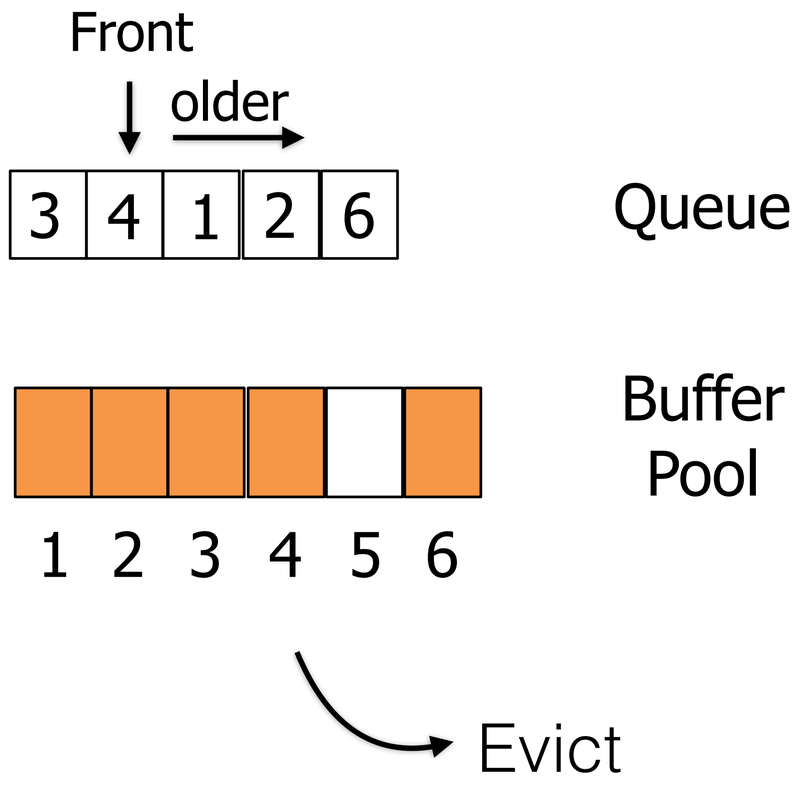
Implementation?



First in First Out (FIFO)

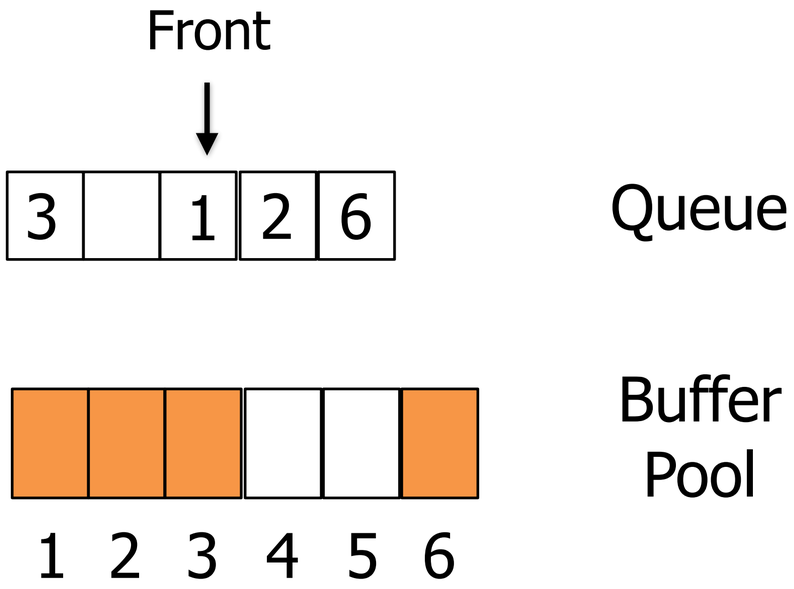
Evict Page that was inserted the longest time ago

Implementation? Using a queue



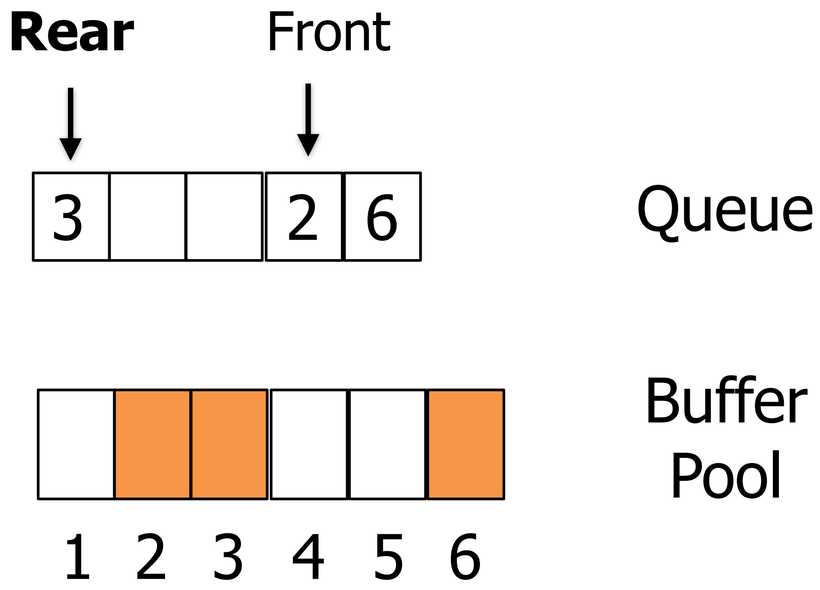
First in First Out (FIFO)

Evict Page that was inserted the longest time ago



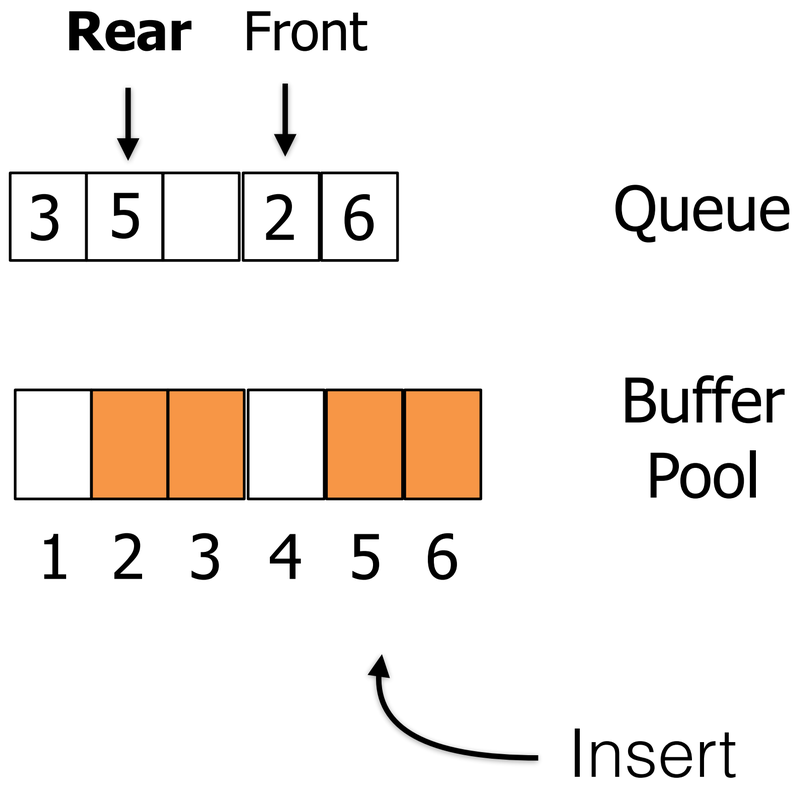
First in First Out (FIFO)

Evict Page that was inserted the longest time ago



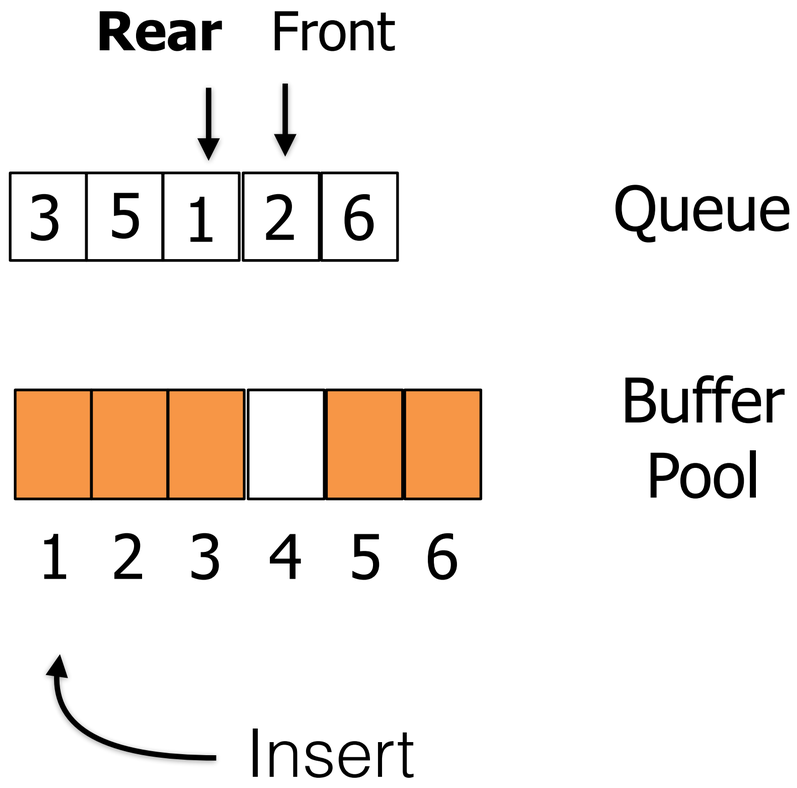
First in First Out (FIFO)

Evict Page that was inserted the longest time ago



First in First Out (FIFO)

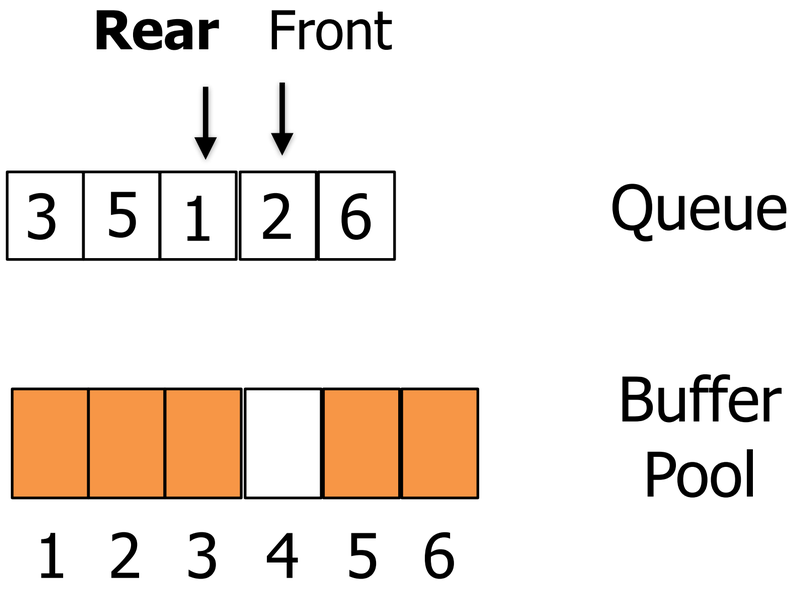
Evict Page that was inserted the longest time ago



First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Problems?



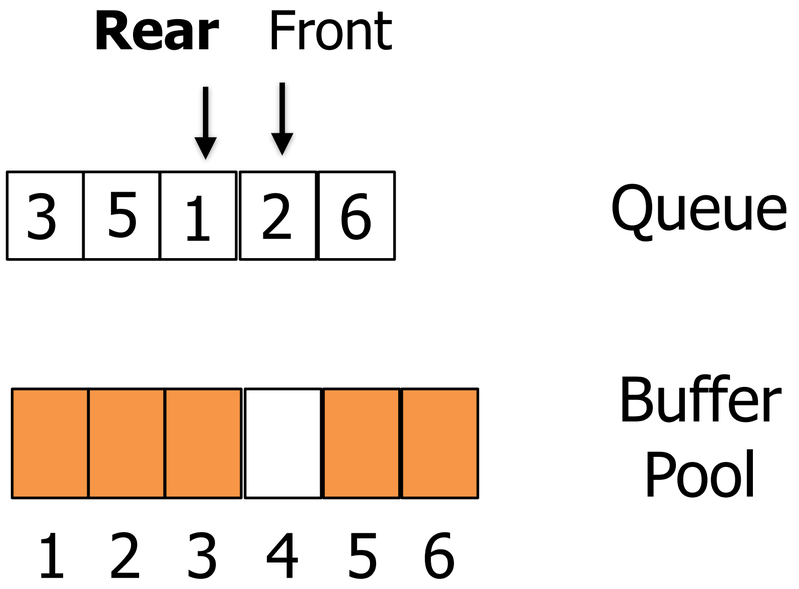
First in First Out (FIFO)

Evict Page that was inserted the longest time ago

Problems?

1. Pages we evict have different frames than pages we insert. Need to have more spare capacity to curb hash collisions. We'll accept this.

2. Oldest page may still be frequently used. We'll try to do better.



Least Recently Used (LRU)

Evict page that was used last the longest time ago

Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation?

Least Recently Used (LRU)

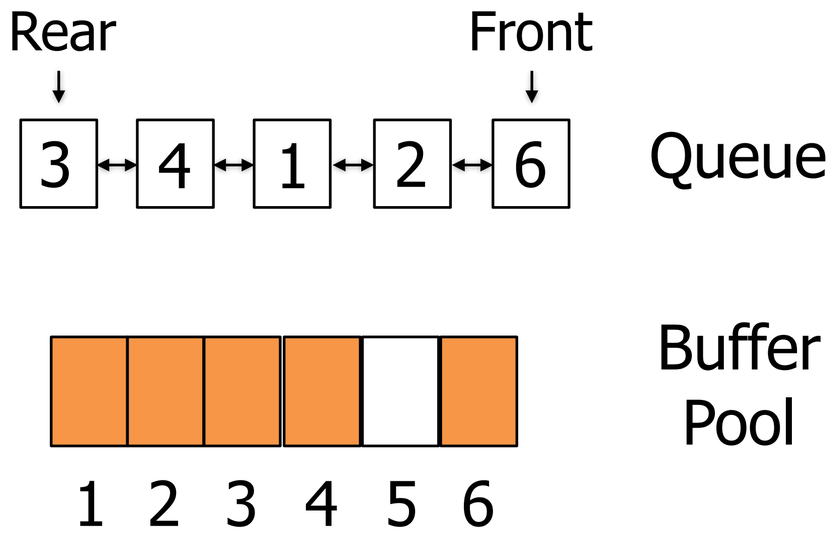
Evict page that was used last the longest time ago

Implementation? Doubly-linked list

Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation? Doubly-linked list

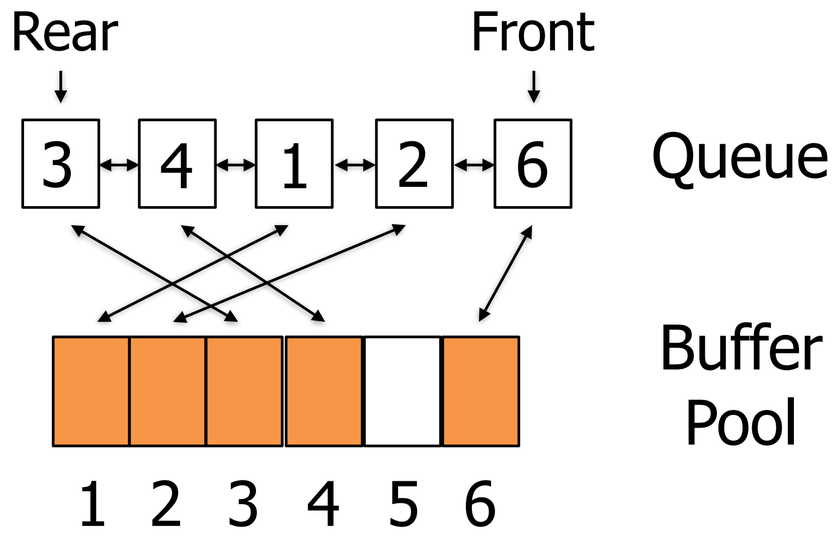


Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation? Doubly-linked list

With pointers linking nodes and buckets



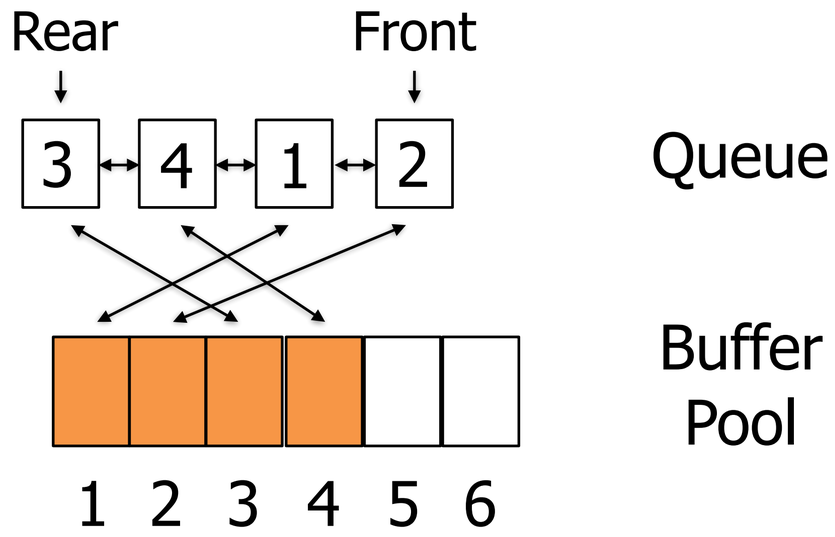
Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation? Doubly-linked list

With pointers linking nodes and buckets

Load to rear and evict front (as before)



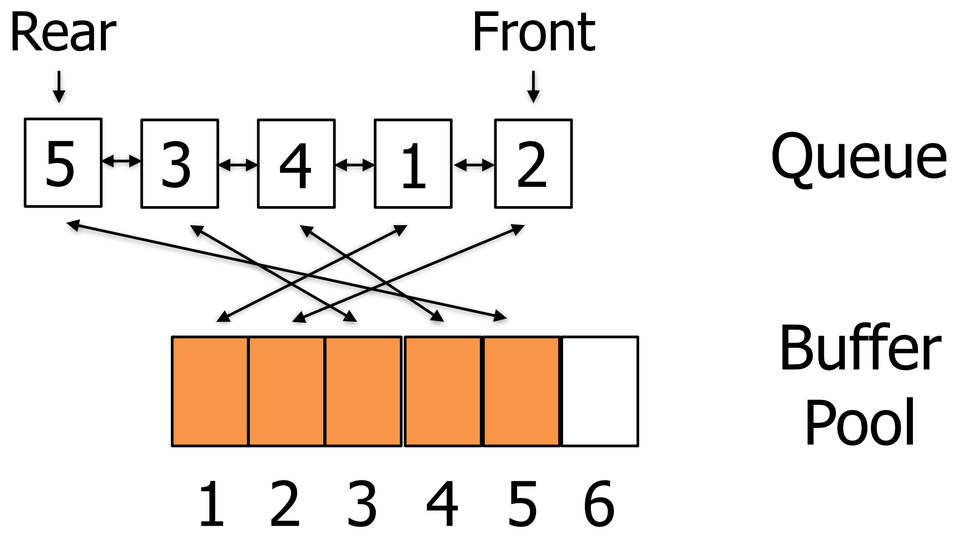
Least Recently Used (LRU)

Evict page that was used last the longest time ago

Implementation? Doubly-linked list

With pointers linking nodes and buckets

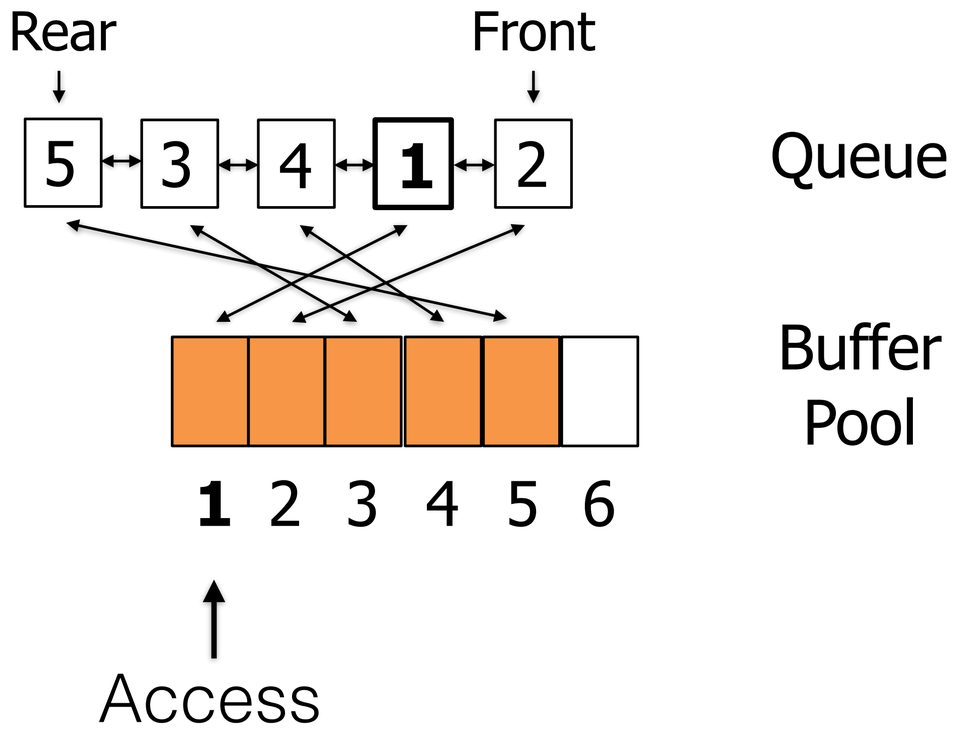
Load to rear and evict front (as before)



Least Recently Used (LRU)

Evict page that was used last the longest time ago

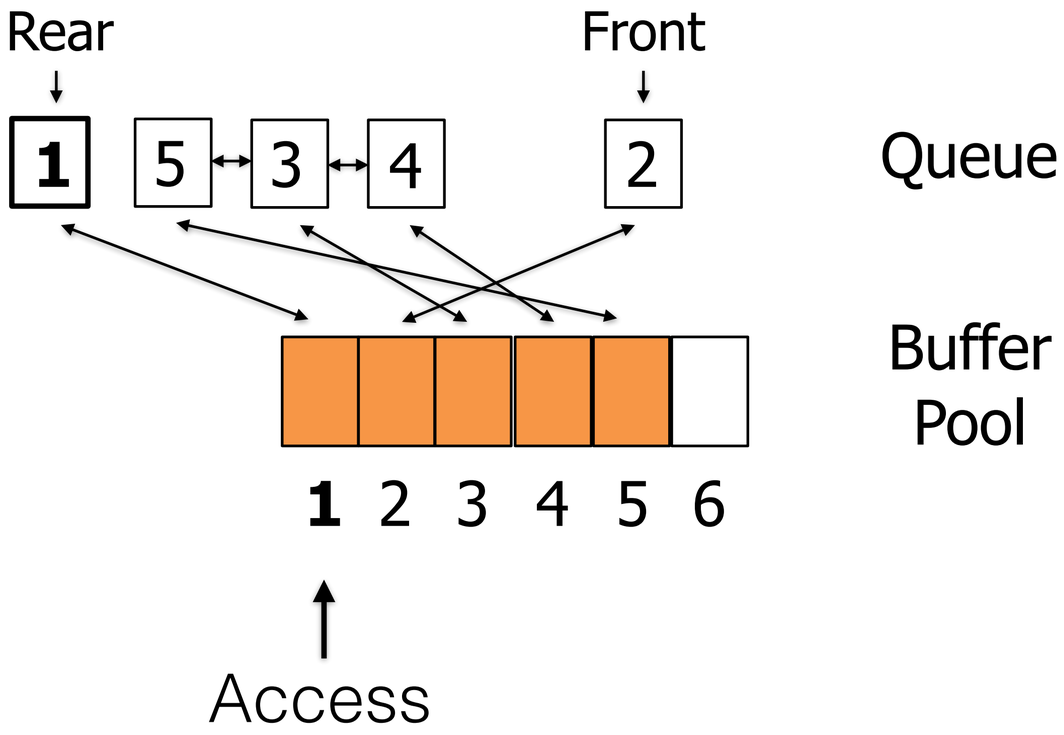
During access, return entry to rear



Least Recently Used (LRU)

Evict page that was used last the longest time ago

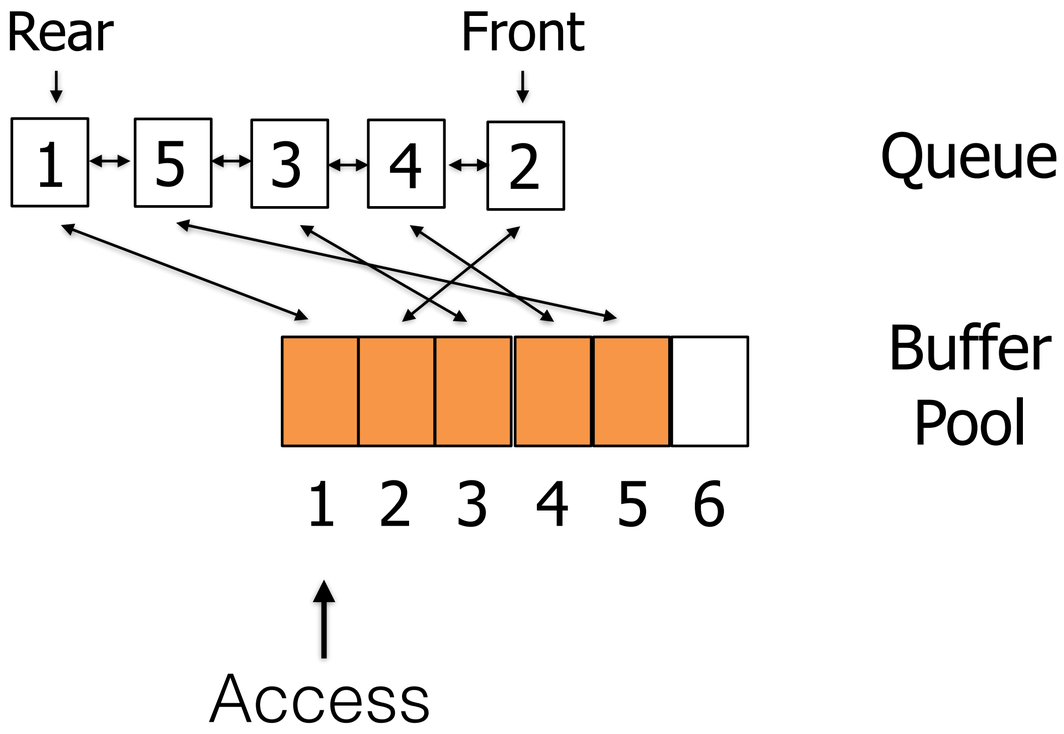
Implementation? Doubly-linked list



Least Recently Used (LRU)

Evict page that was used last the longest time ago

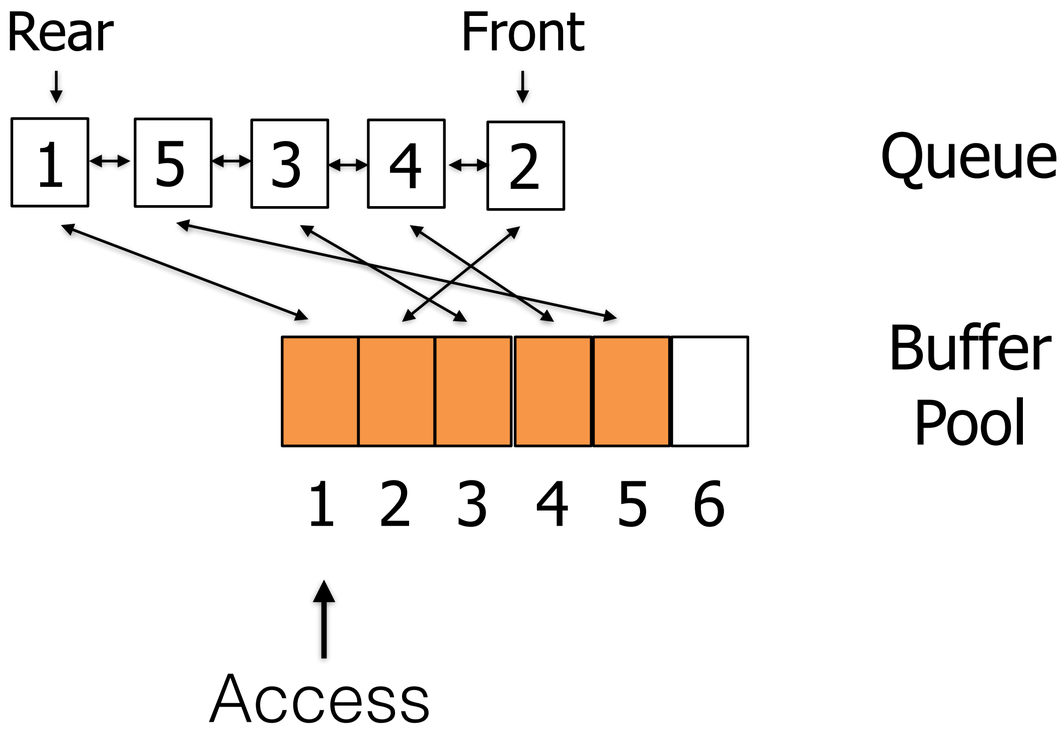
Implementation? Doubly-linked list



Least Recently Used (LRU)

Evict page that was used last the longest time ago

Problems?



Least Recently Used (LRU)

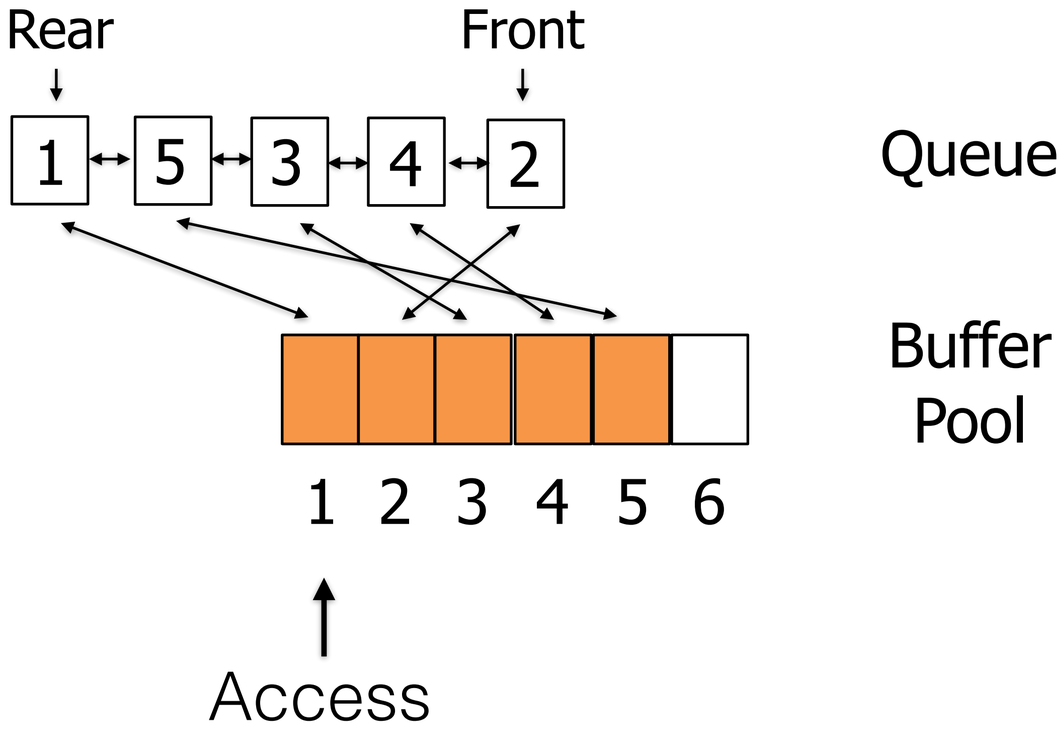
Evict page that was used last the longest time ago

Problems?

(1) CPU overhead to update queue for each access

(2) Metadata overhead for pointers

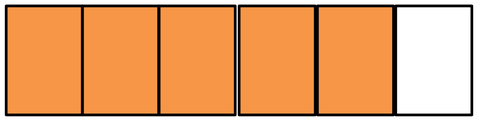
(3) Linked lists are less efficient than arrays due to pointer chasing



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

Implementation?

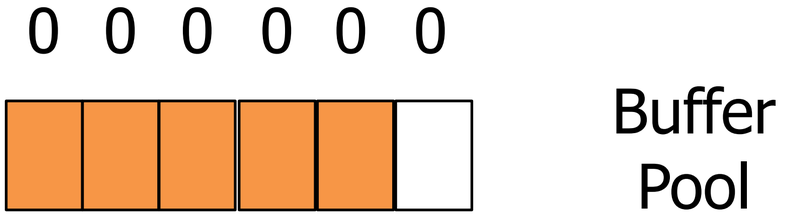


Buffer Pool

Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

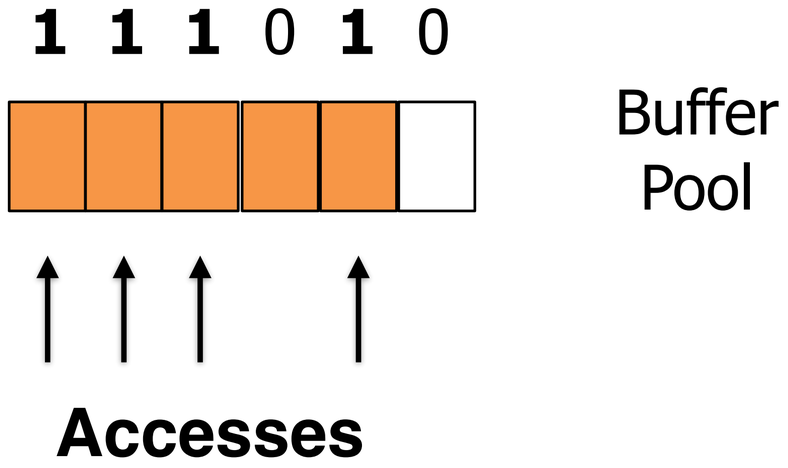
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

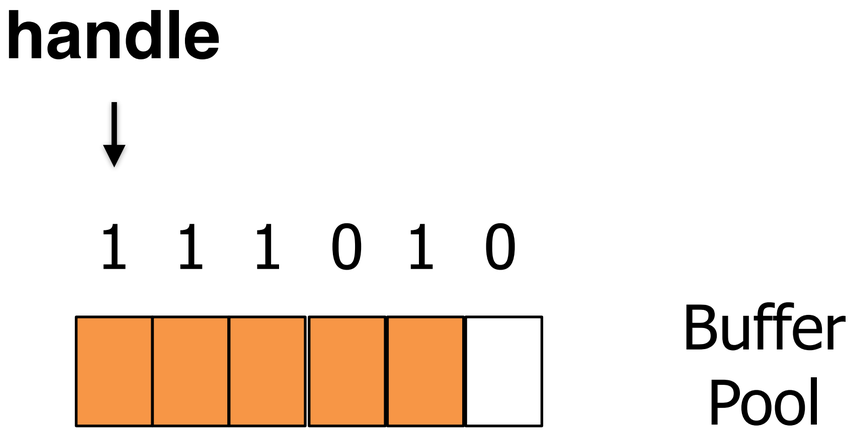
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

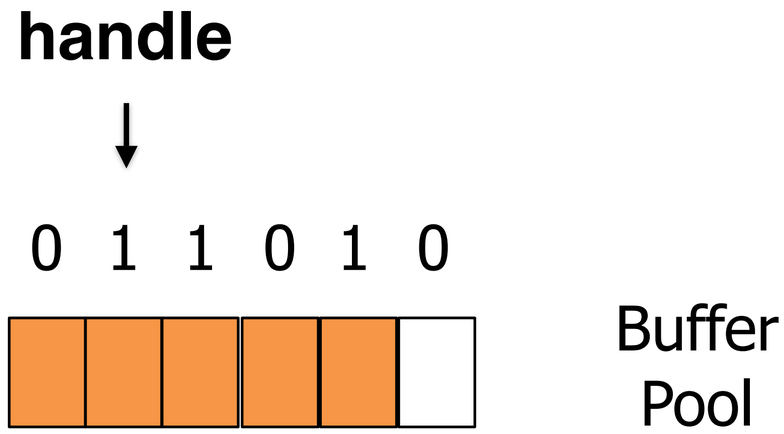
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

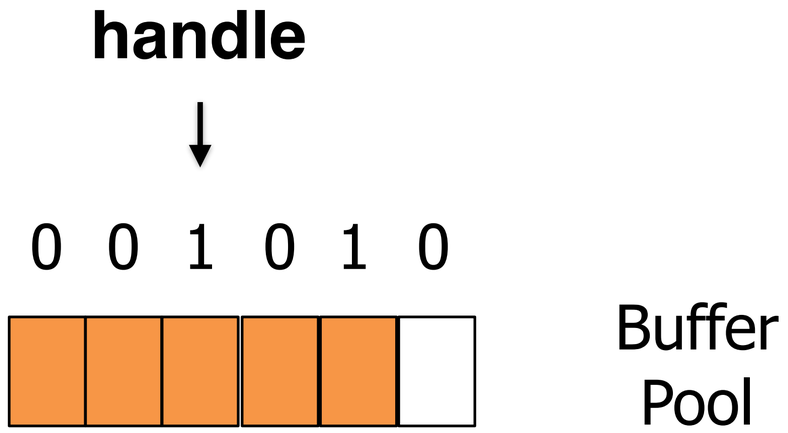
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

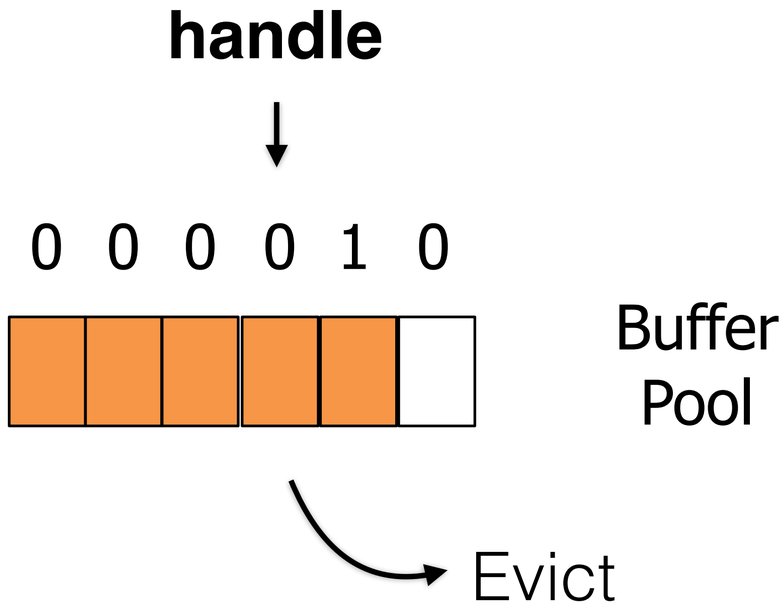
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

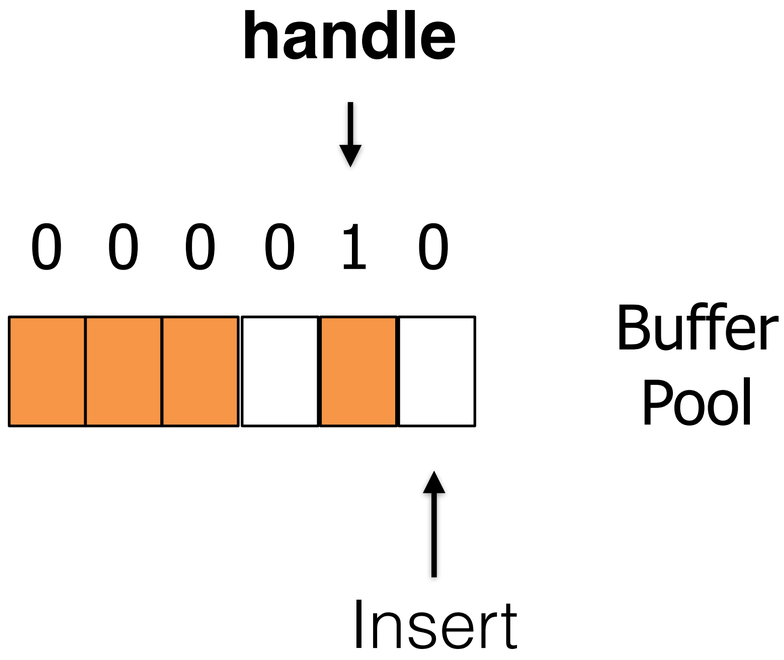
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

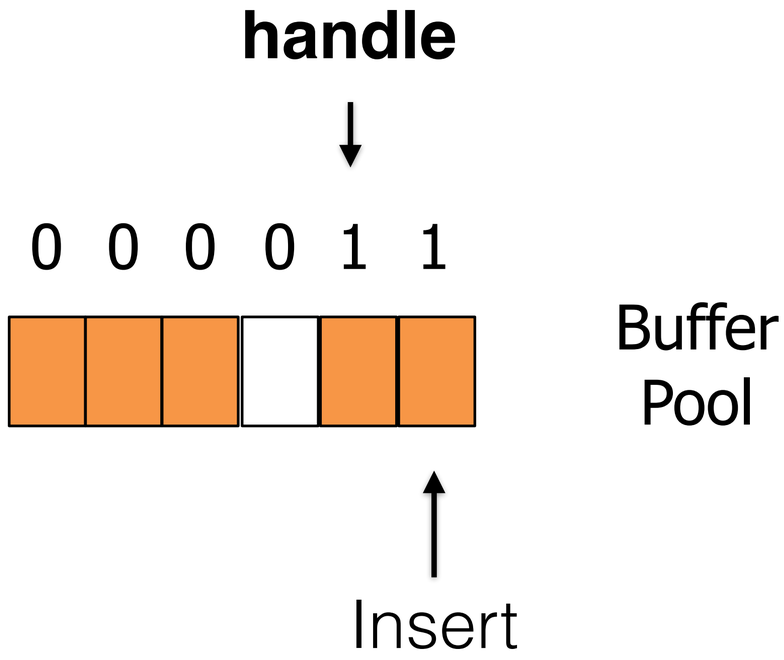
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

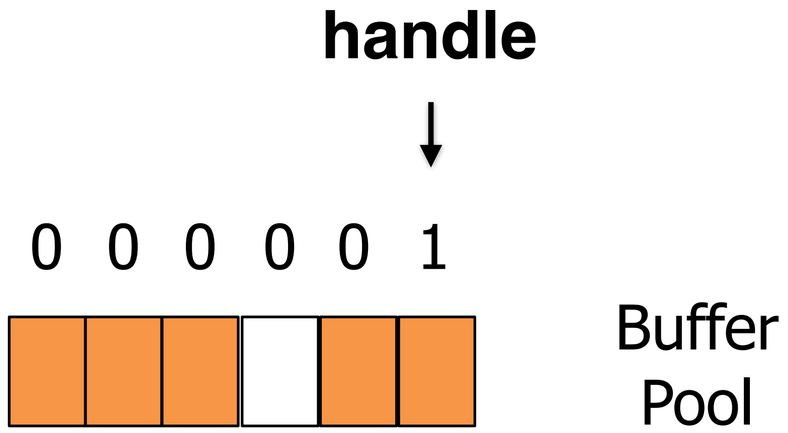
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

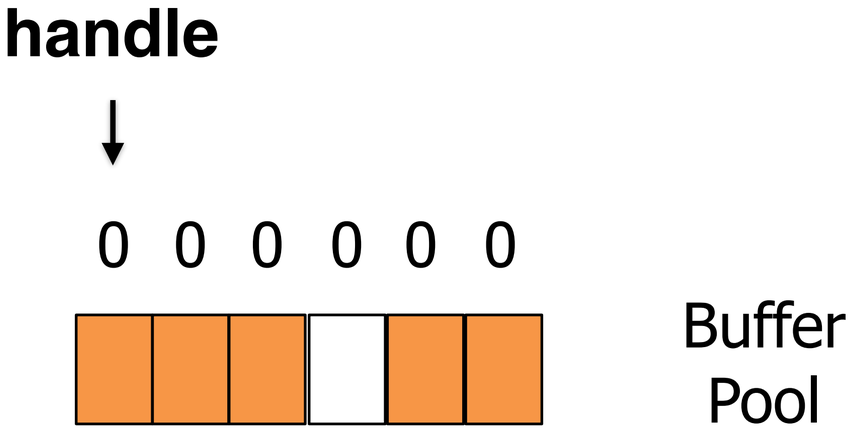
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

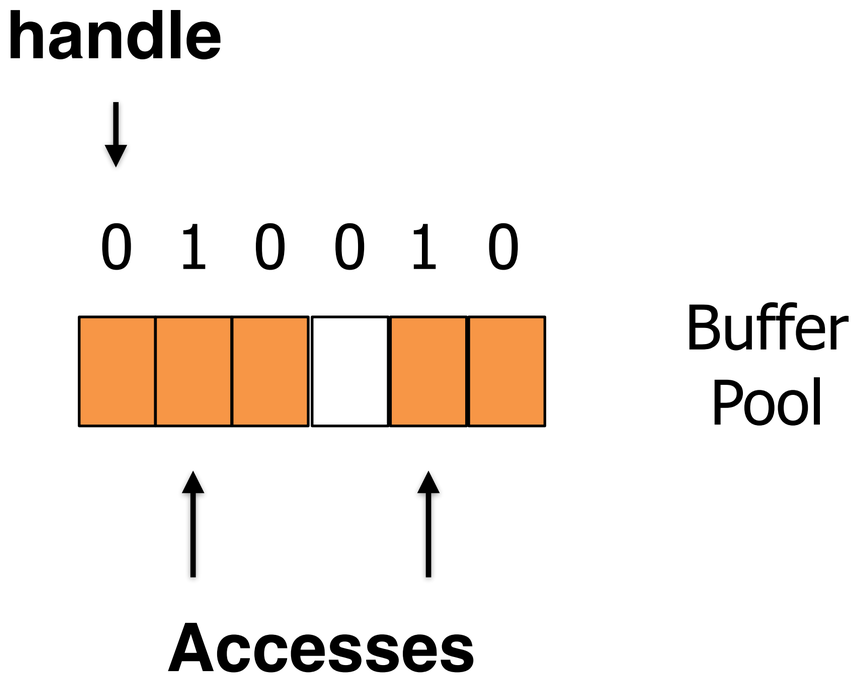
Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

Implementation? Bitmap



Clock

Traverse hash table circularly as a clock.
Evict any entry not used since last traversal.

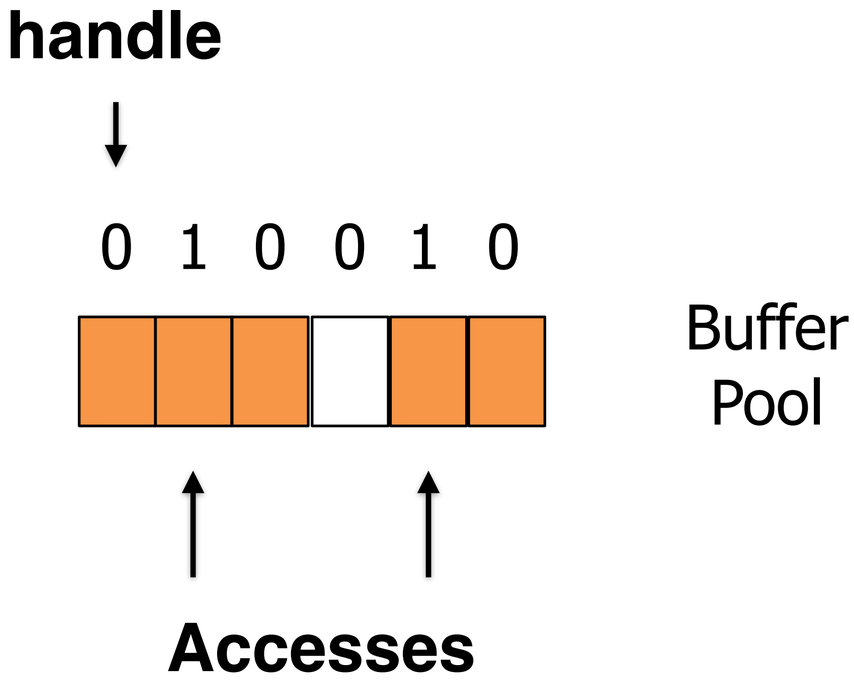
Advantages

(1) lower overheads as there is no queue

(2) bitmap takes little extra space

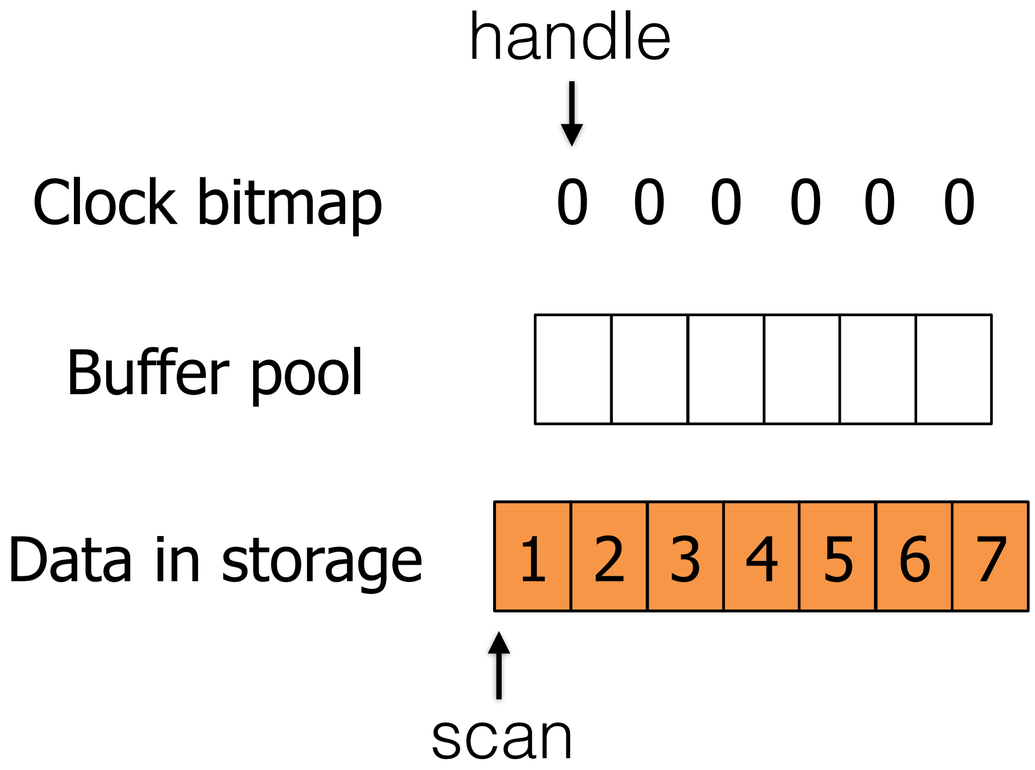
Disadvantages

(1) can evict “hotter” pages than LRU,
But still better than FIFO



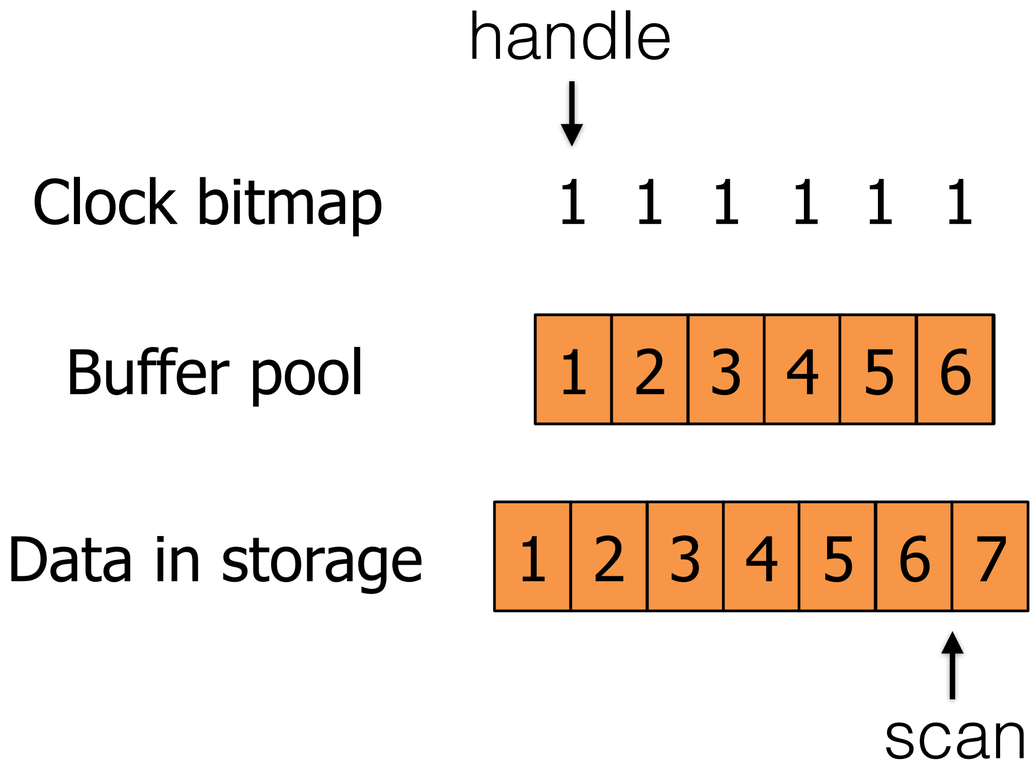
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



Sequential Flooding of Clock

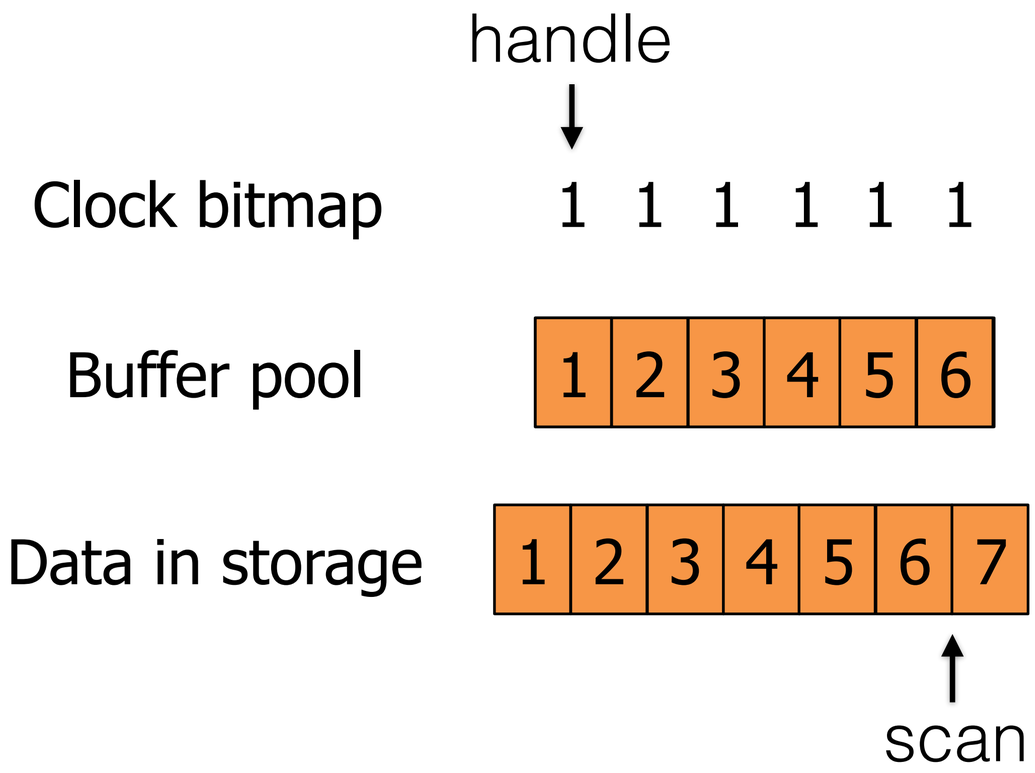
Suppose the DB is repeatedly scanning data larger than the buffer pool



Sequential Flooding of Clock

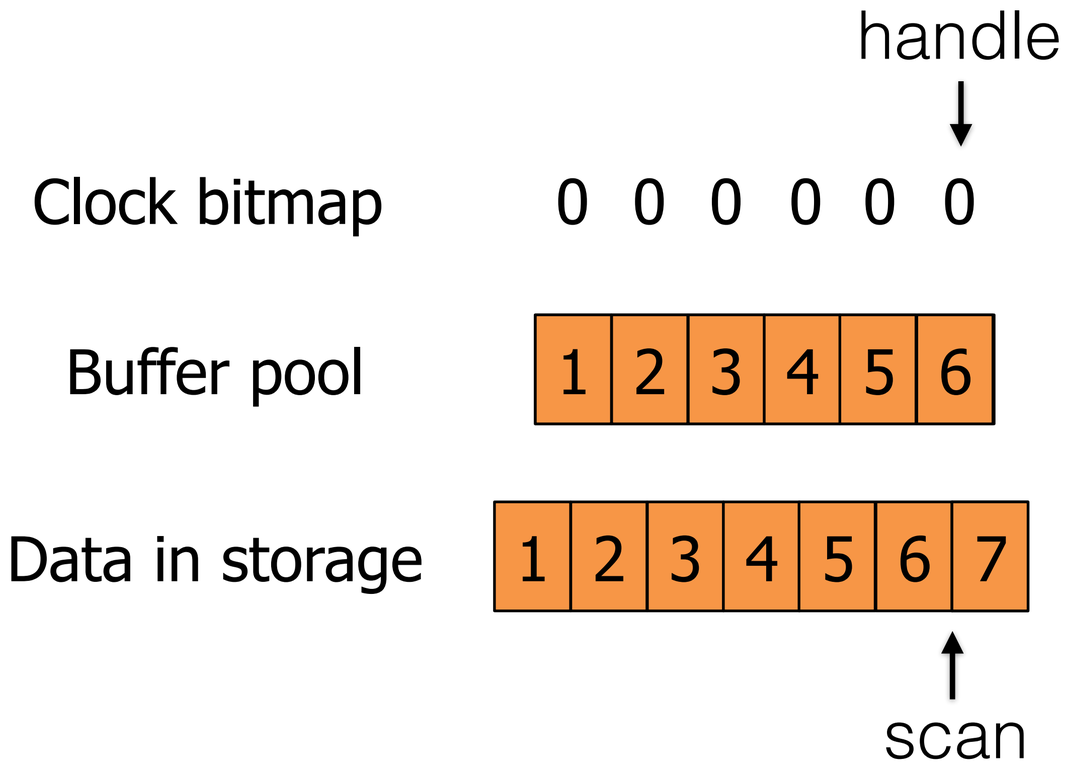
Suppose the DB is repeatedly scanning data larger than the buffer pool

Note the Simplification. Elements would really be randomly mapped in the buffer pool due to hashing.



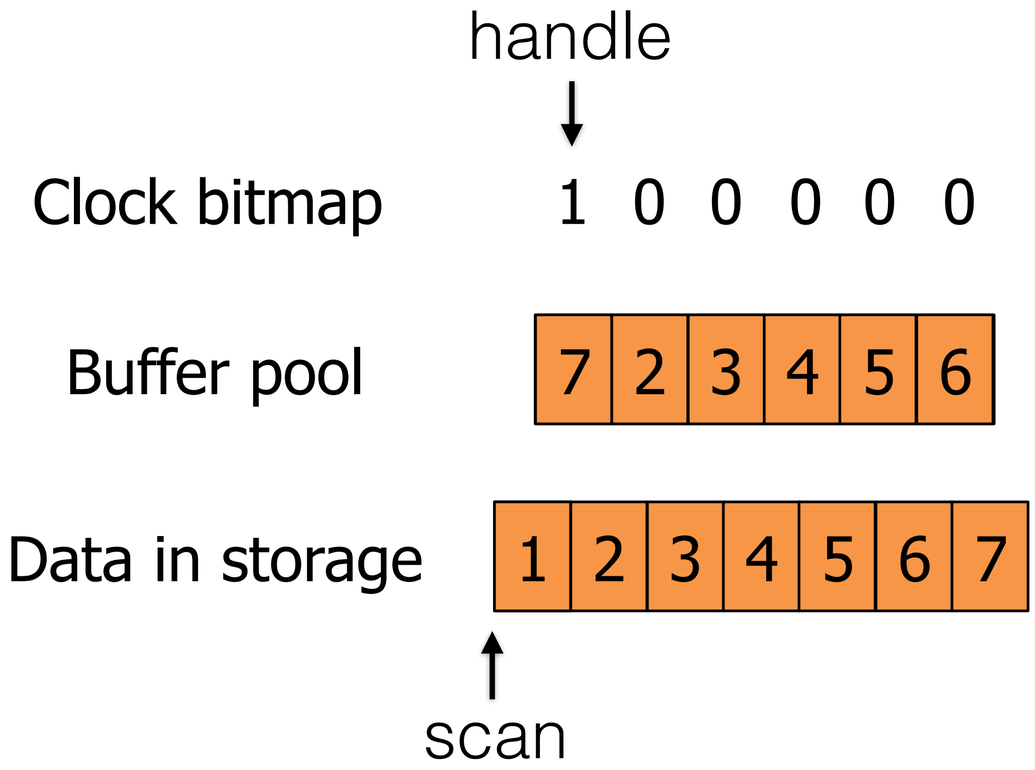
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



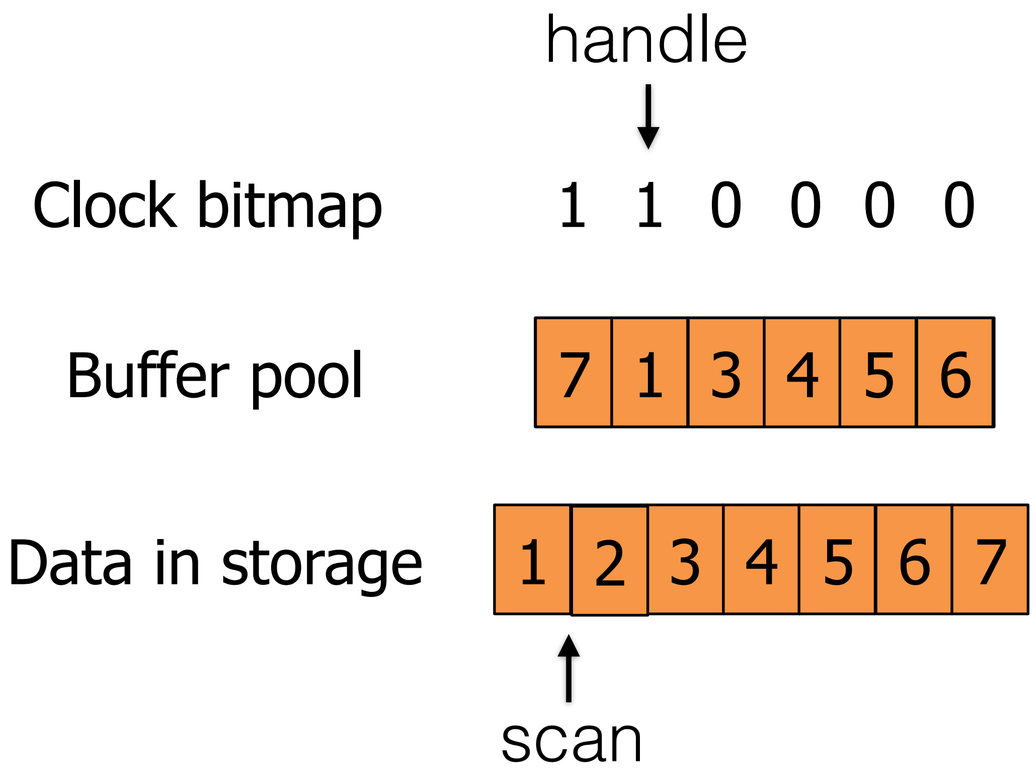
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



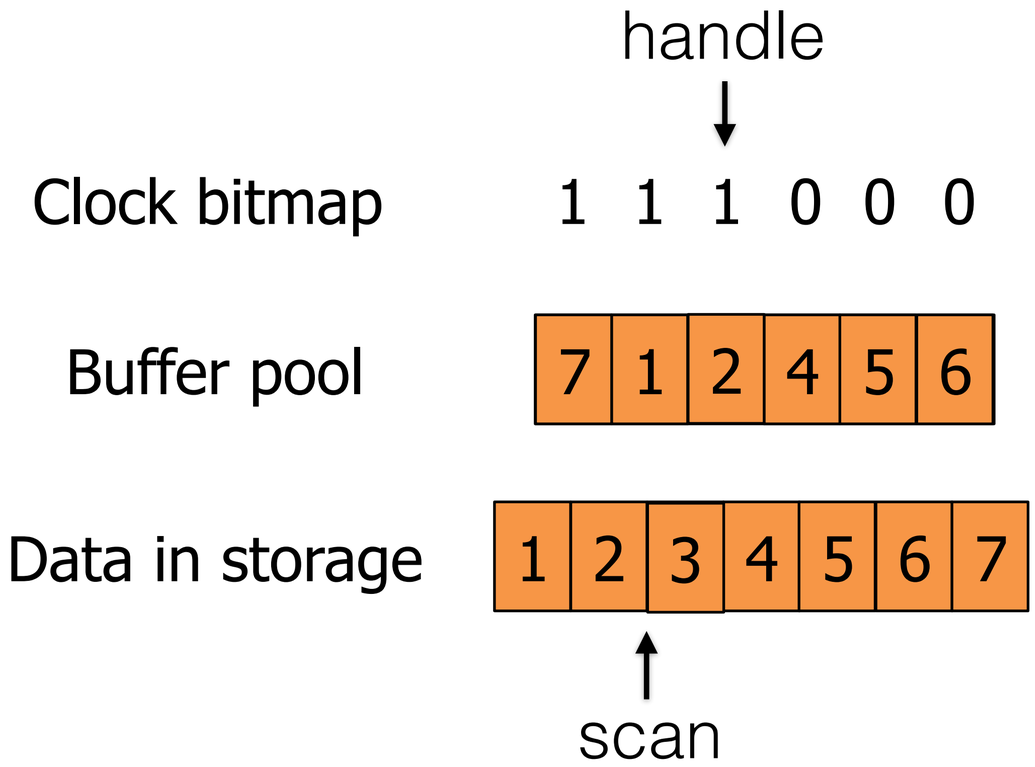
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



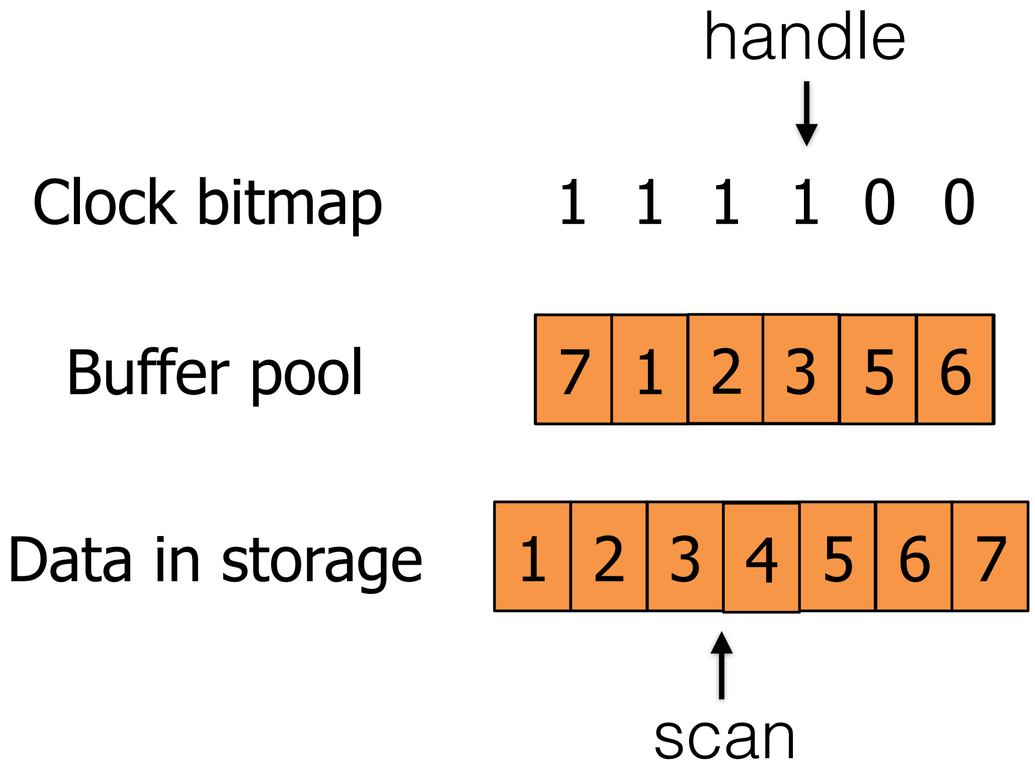
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



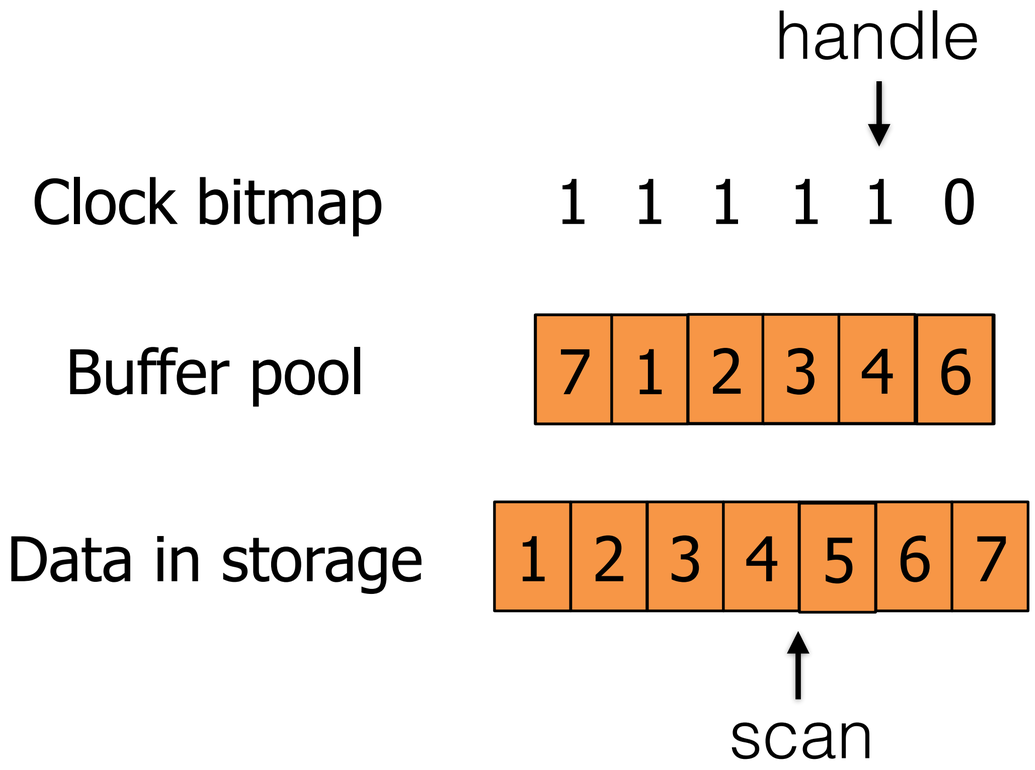
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool



Sequential Flooding of Clock

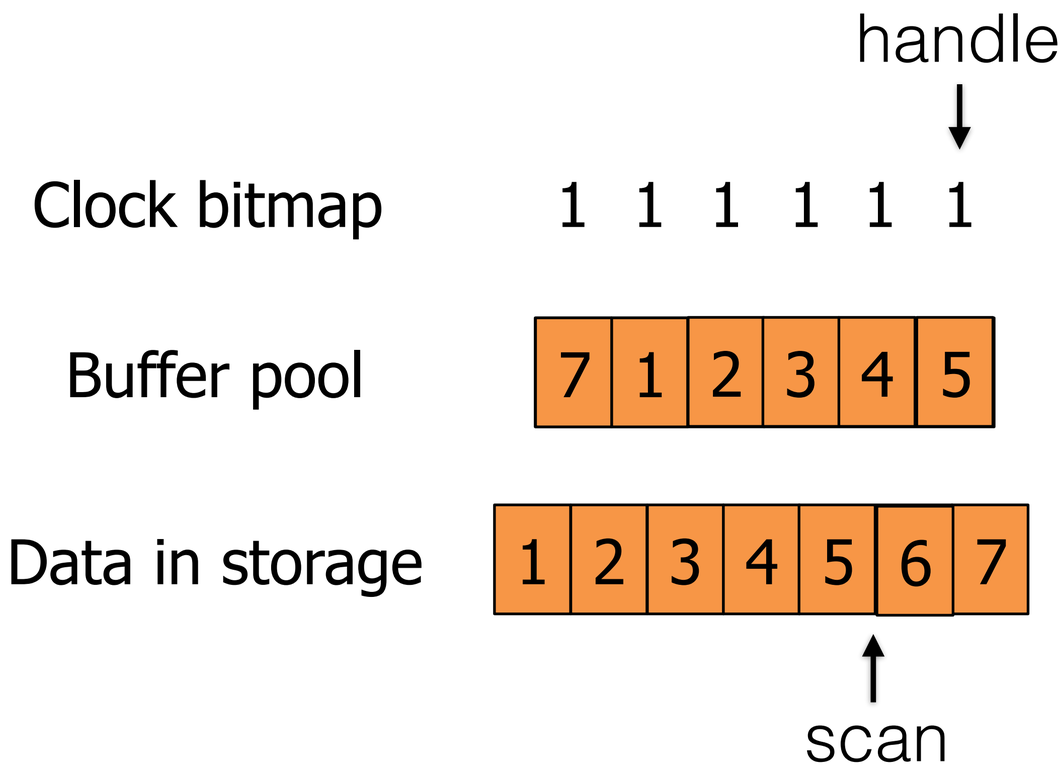
Suppose the DB is repeatedly scanning data larger than the buffer pool



Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool

Clock constantly evicts the page we need to read next!



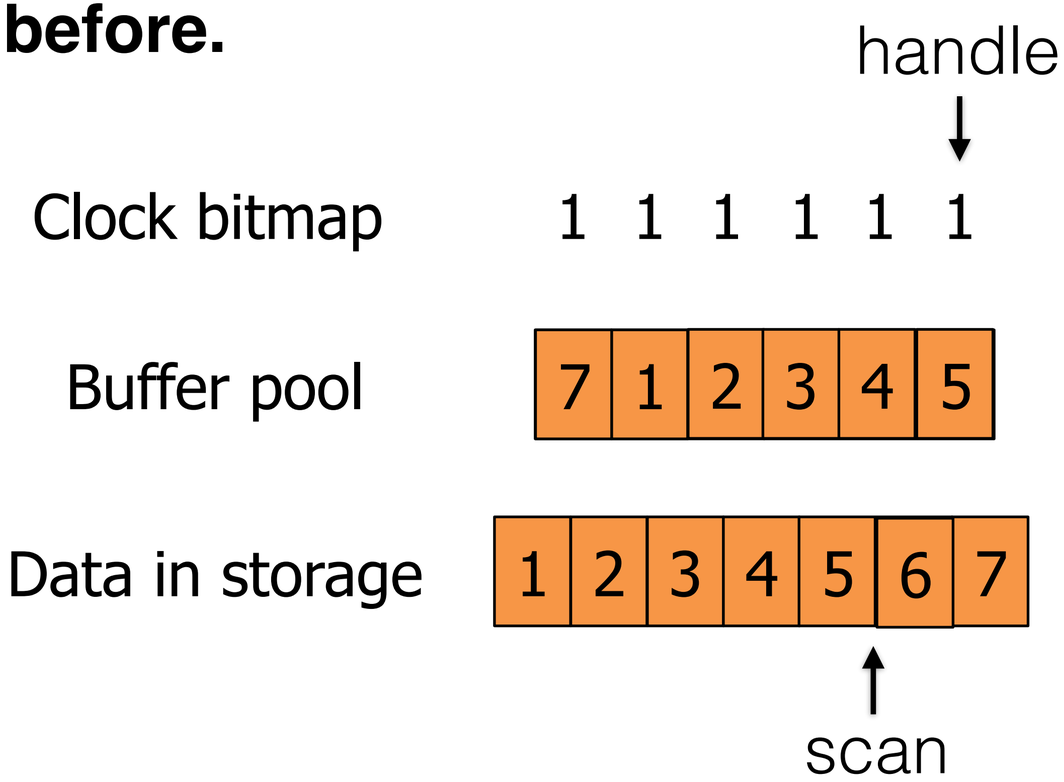
Sequential Flooding of Clock

Suppose the DB is repeatedly scanning data larger than the buffer pool

Clock constantly evicts the page we need to read next!

Also a problem for LRU from before.

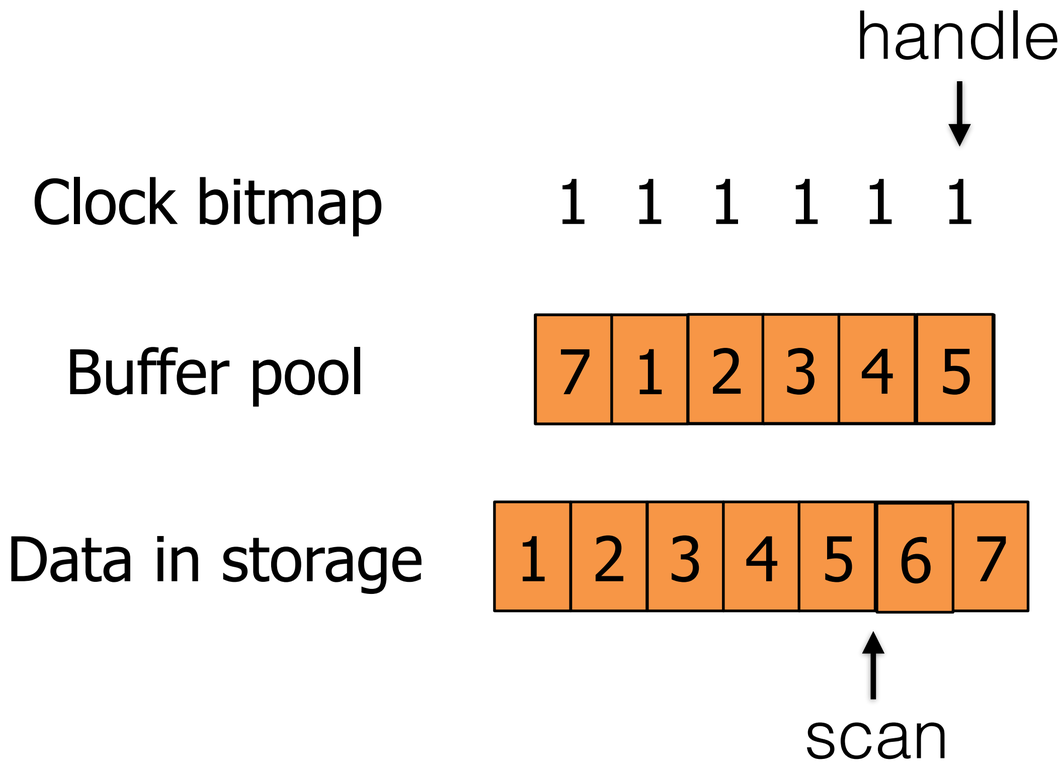
Solution?



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

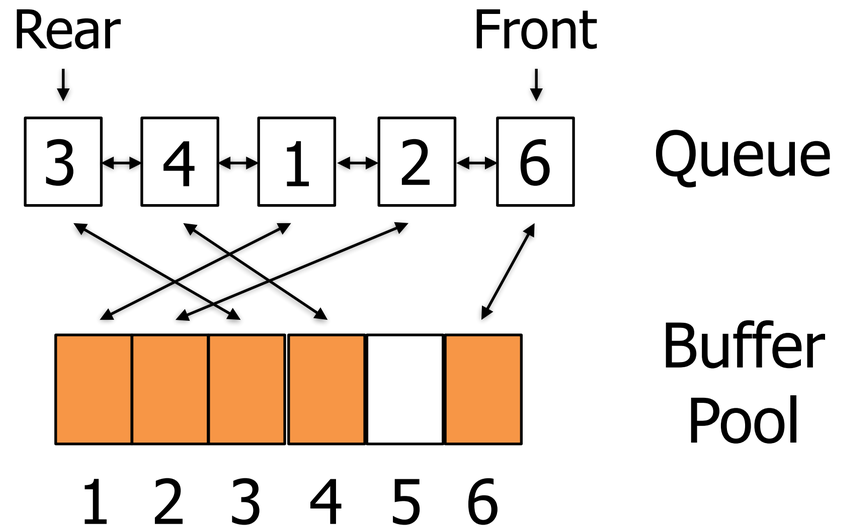
How?



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

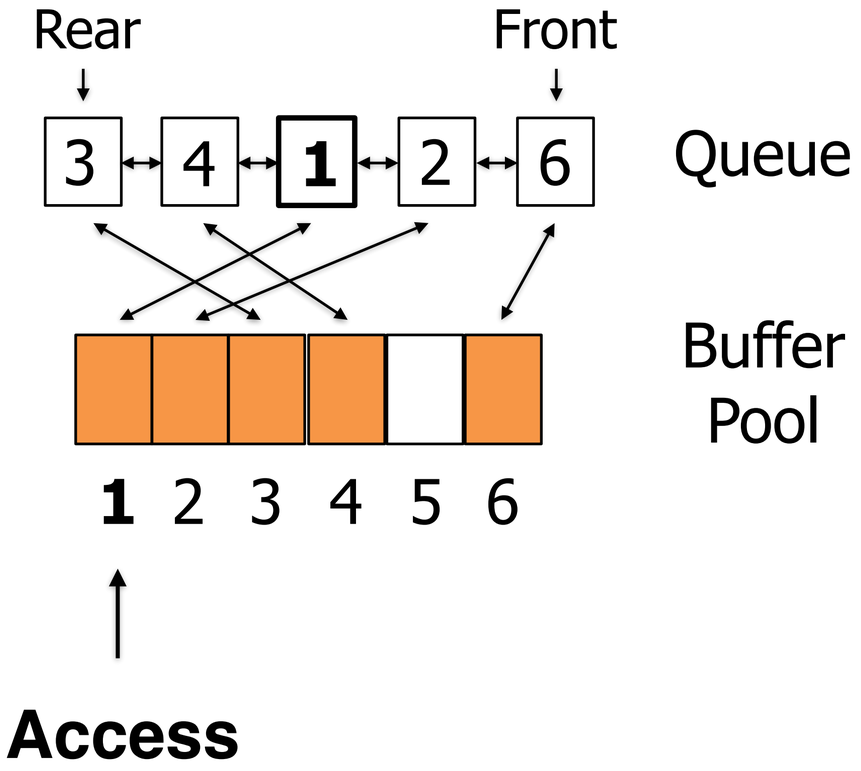
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

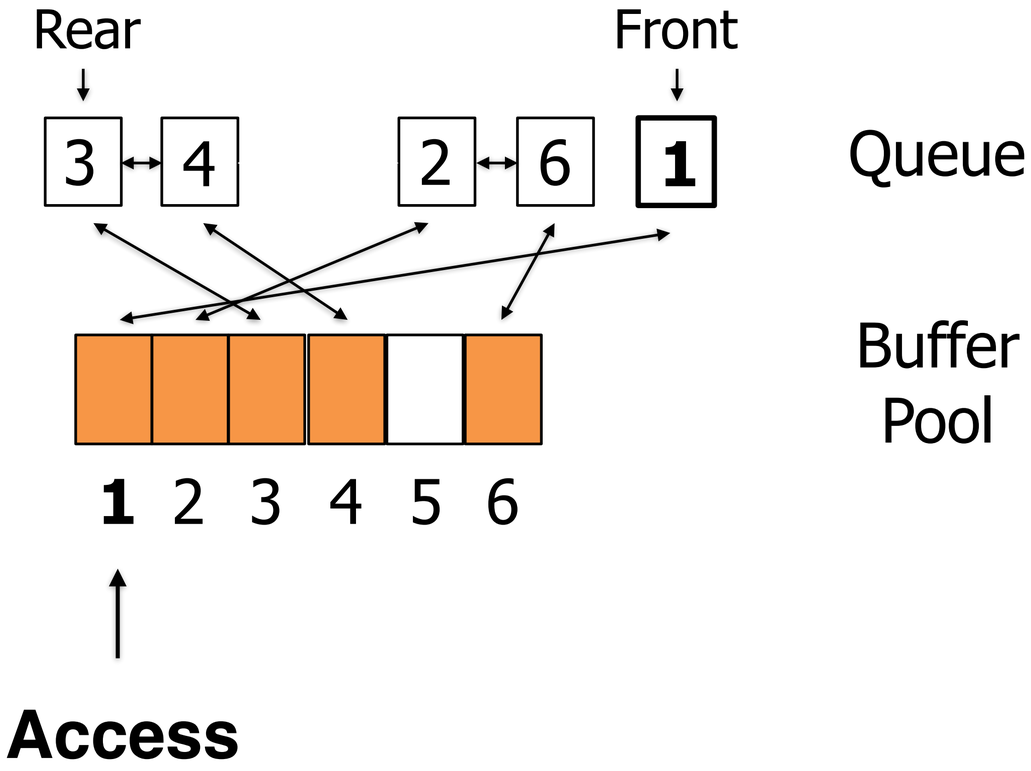
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

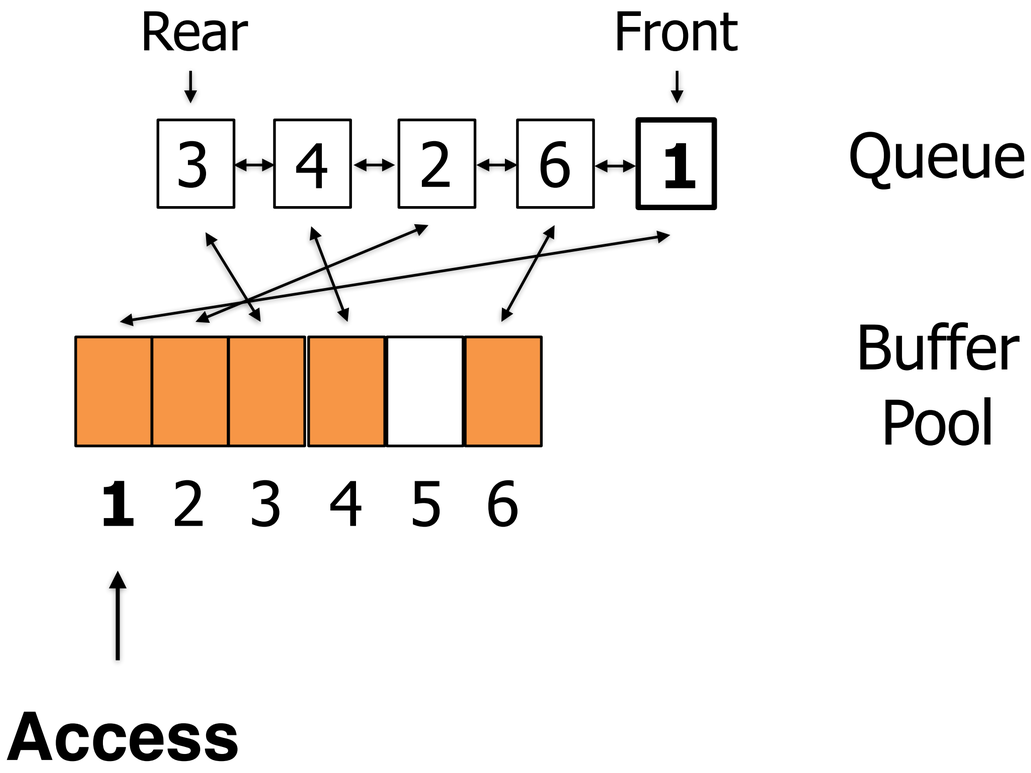
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

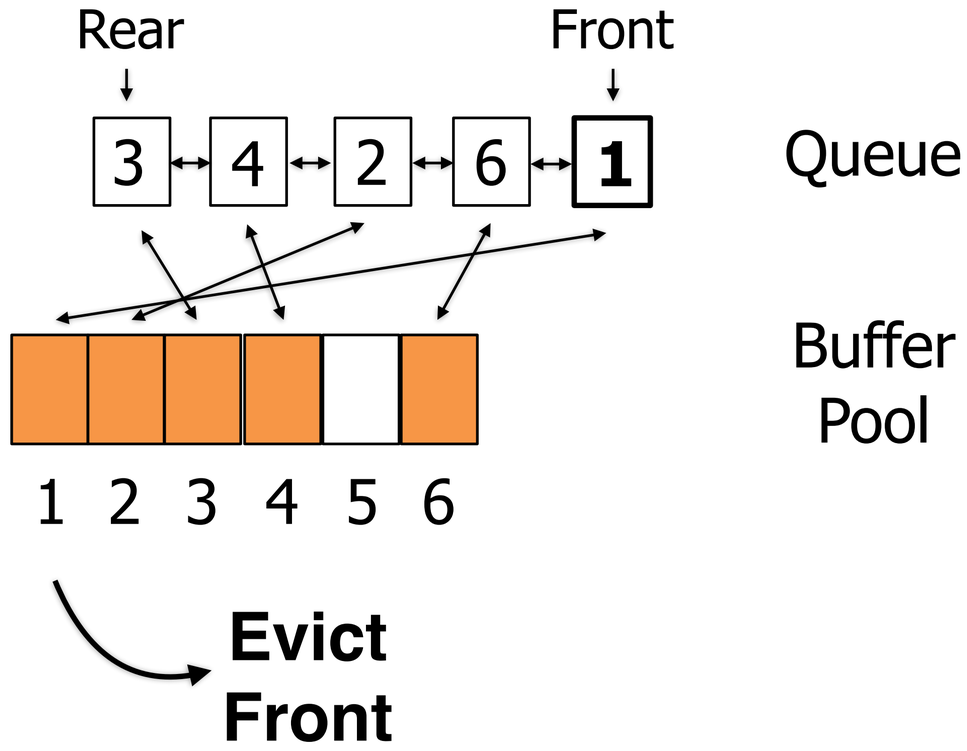
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

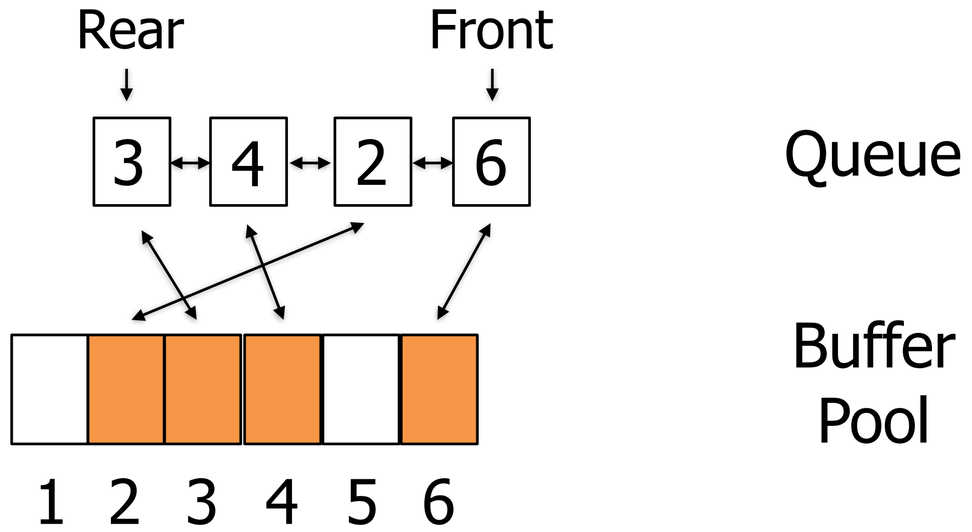
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

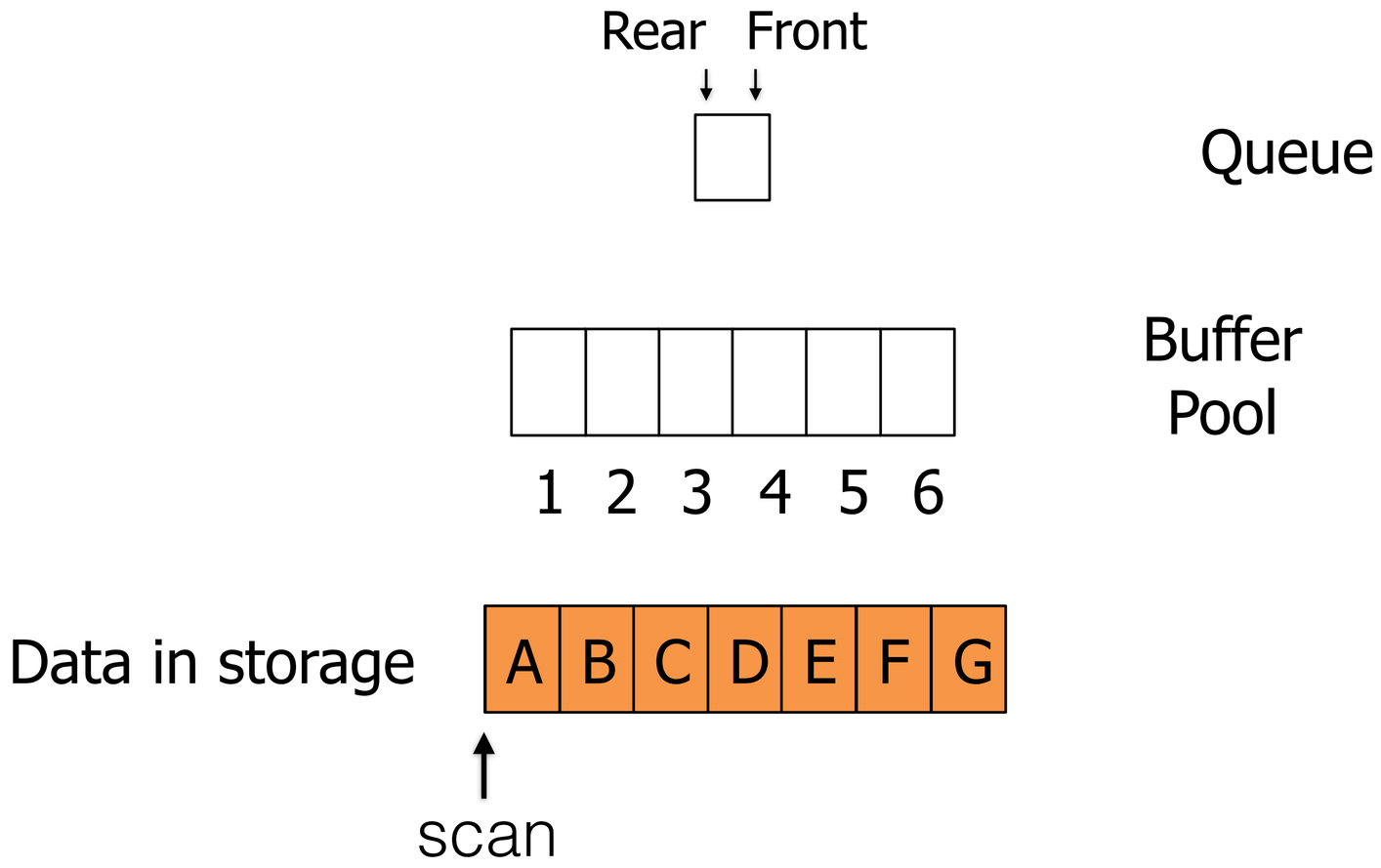
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

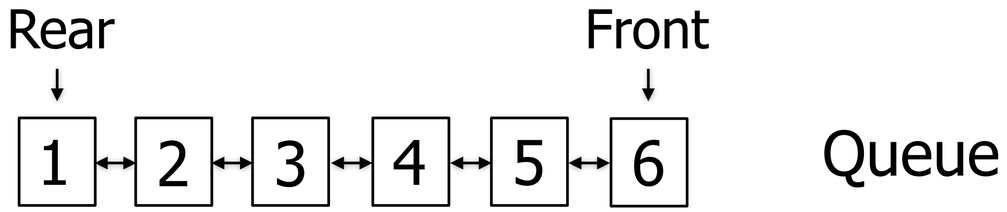
How? Opposite of LRU.



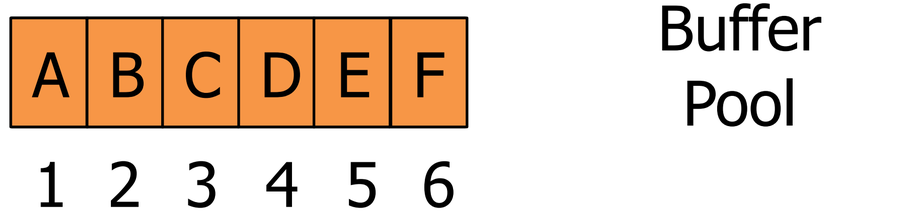
Sequential Flooding of Clock

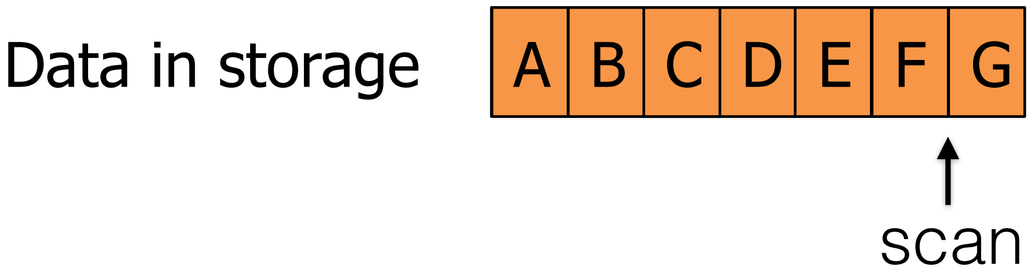
Let's instead evict "most recently used" (MRU) entries?

How? Opposite of LRU.



Note again that Elements would really be randomly mapped in the buffer pool due to hashing.

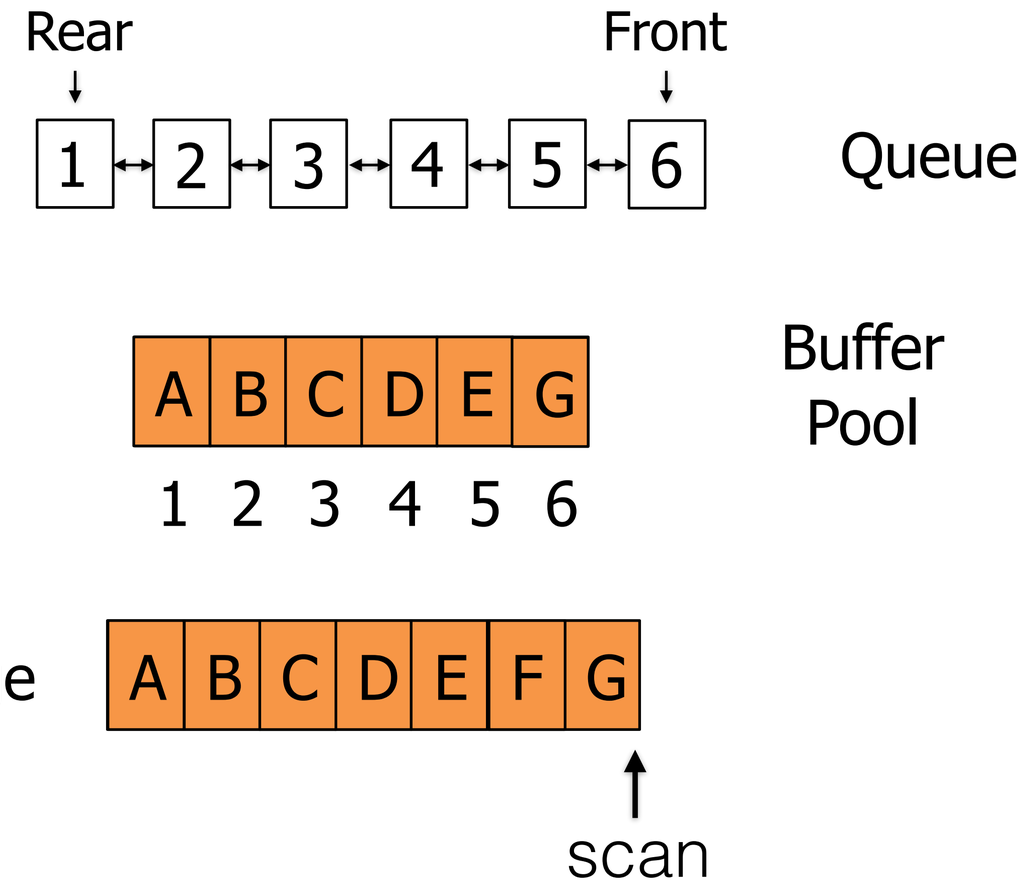




Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

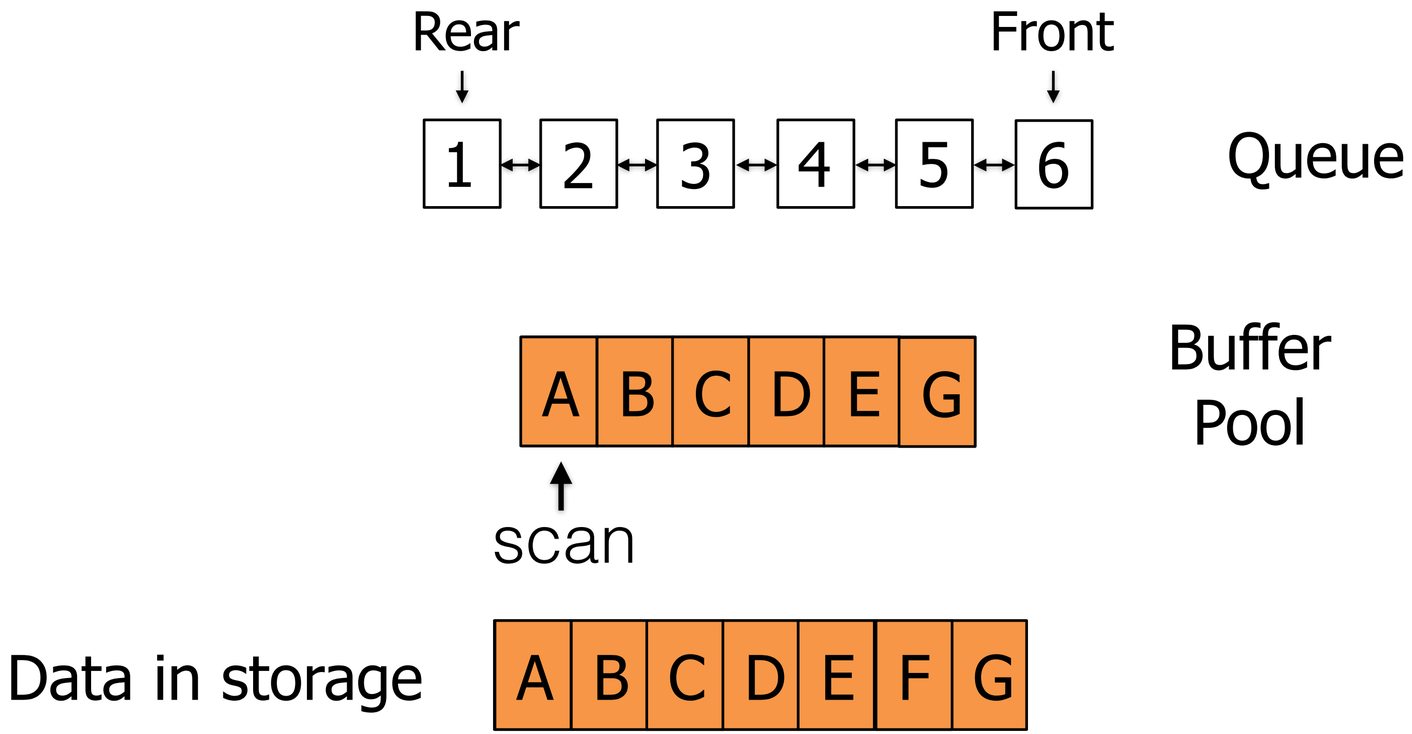
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

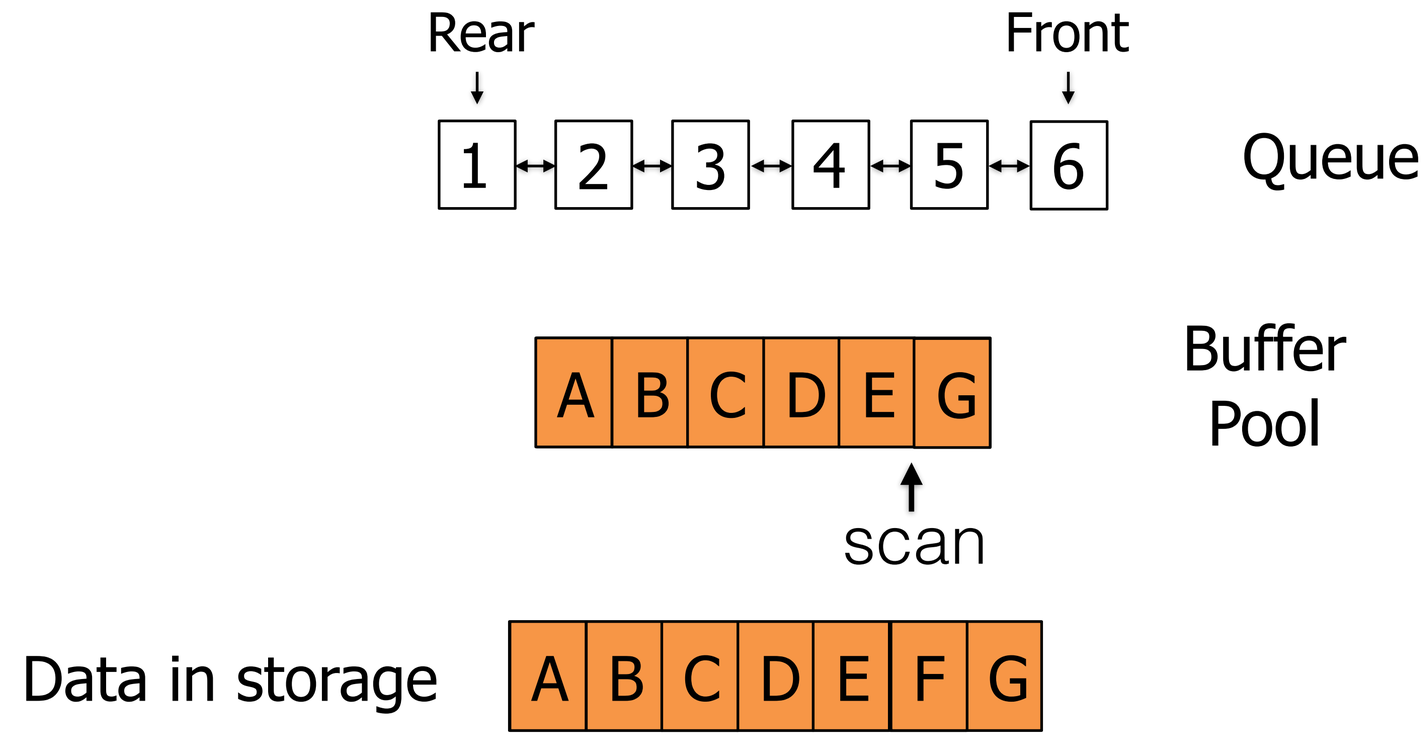
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

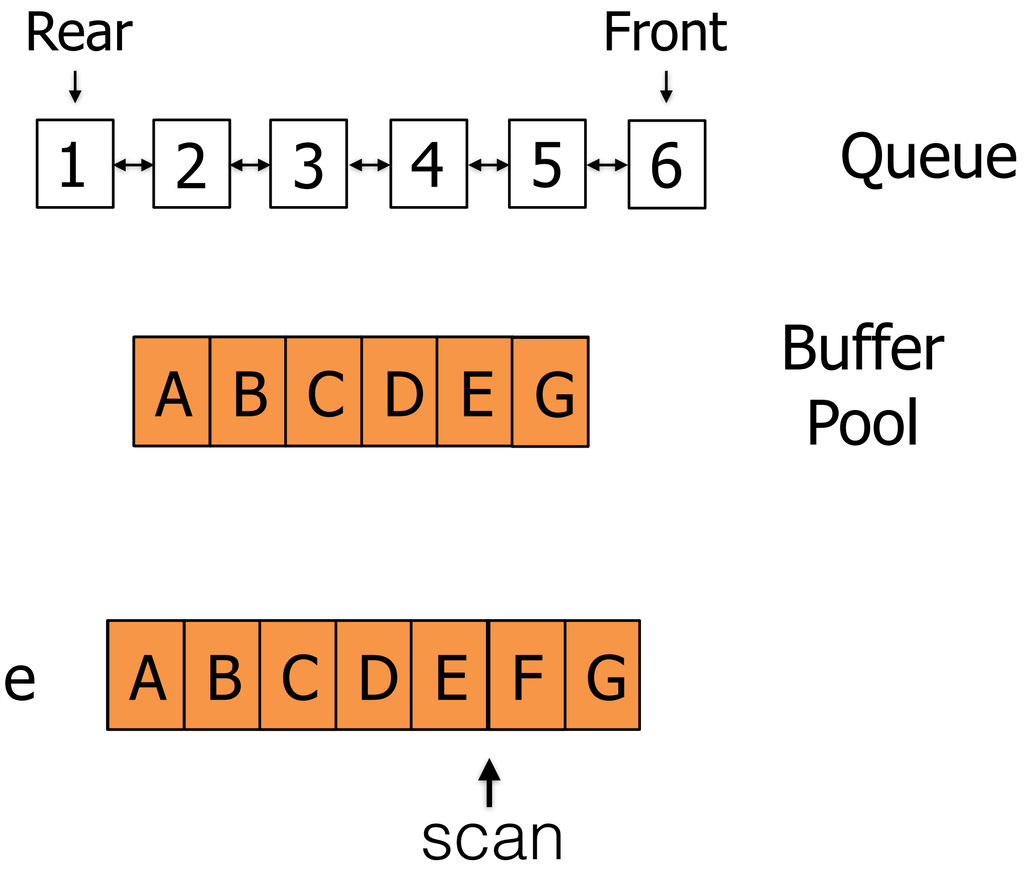
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

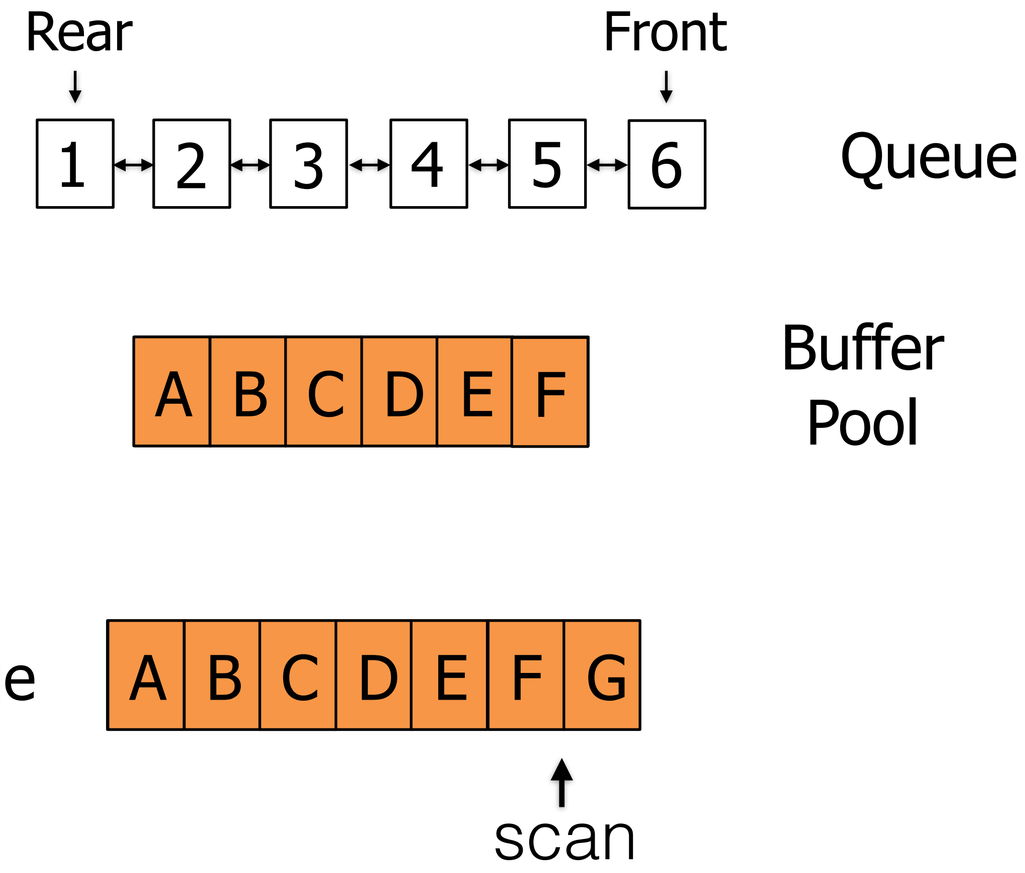
How? Opposite of LRU.



Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

How? Opposite of LRU.

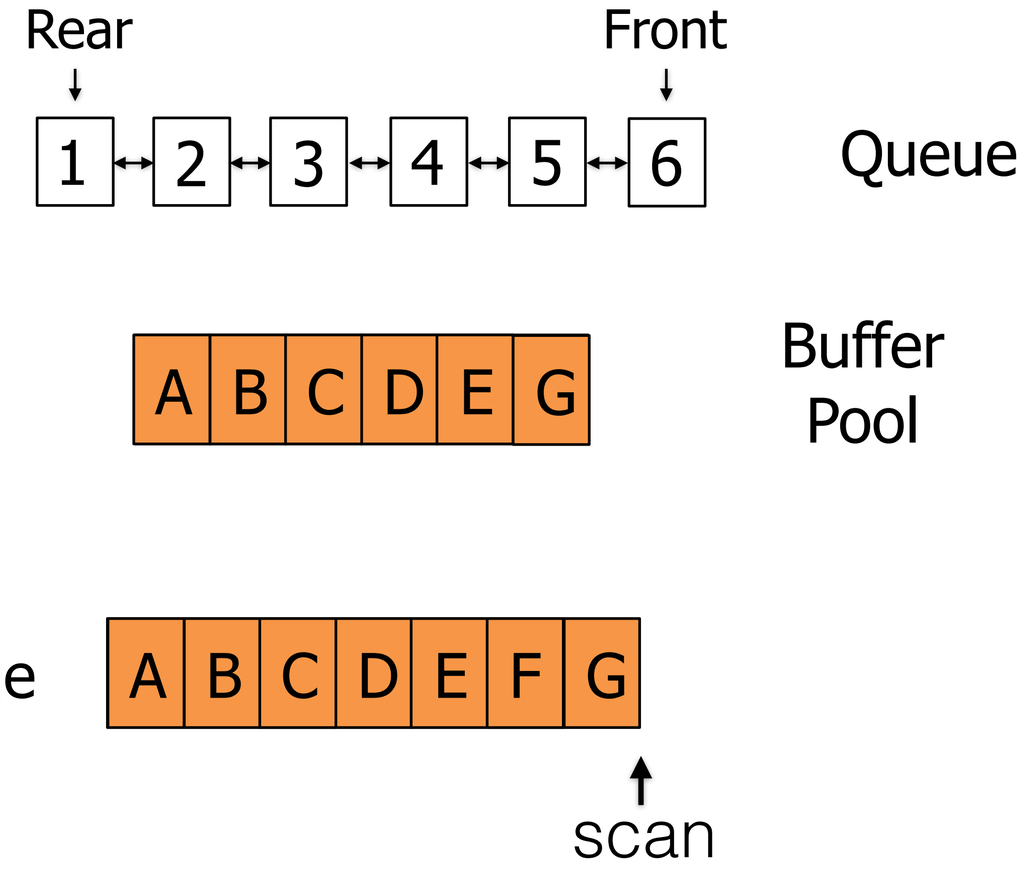


Sequential Flooding of Clock

Let's instead evict "most recently used" (MRU) entries?

How? Opposite of LRU.

Most data stays in the buffer pool.



Sequential Flooding of Clock

In reality, DBs typically use LRU or clock

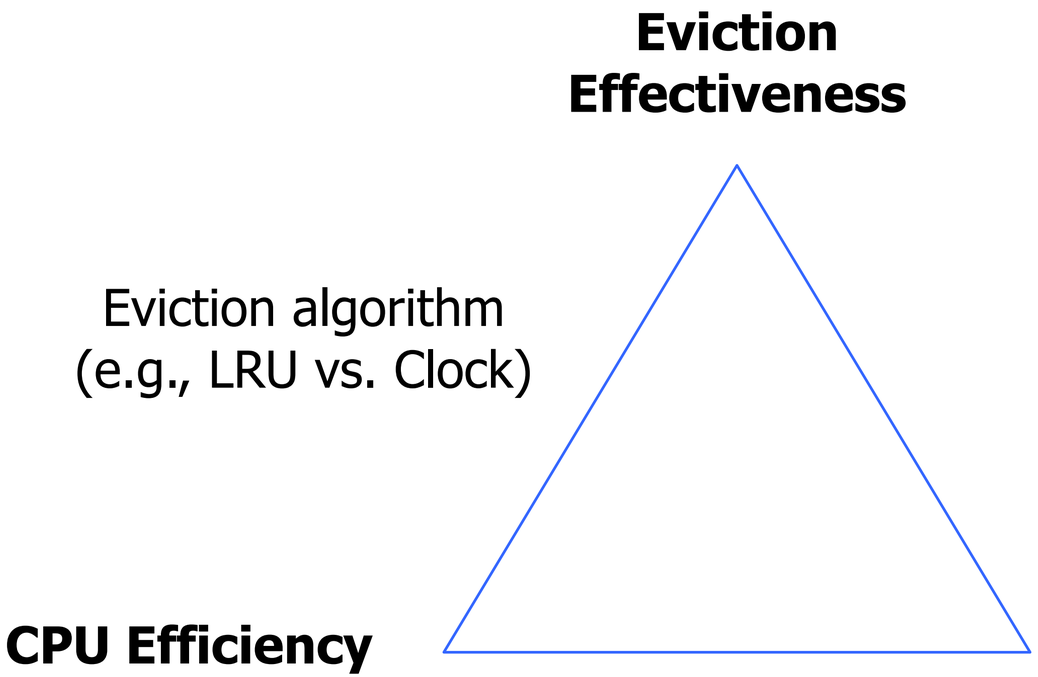
To prevent sequential flooding, they avoid putting scanned data in the buffer pool

Core message: no eviction policy is perfect.

Summary

	Eviction Effectiveness	CPU
Random	Worst	Best
FIFO	Moderate	Moderate
LRU	Best	Worst
Clock	Good	Good

Two trade-offs



Two trade-offs