CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Fourteen
30 July 2012

Review of hash tables

Hash tables

- Hash tables offer $O(1)$ performance for add/find/remove operations in the **average case**.
- These excellent **time** costs come at the expense of additional **space** cost.
  - We use a very large array to store the user's data.

Hash table interfaces

- For hash tables there are two principal interfaces:
  - One in which the key is **inside** the record being stored.
  - One in which the **key** is separate from the **value** of the record being stored.

Hash table interfaces

- For hash tables there are two principal interfaces:
  - One in which the key is **inside** the record being stored.
  - One in which the **key** is separate from the **value** of the record being stored.

More common

Hash table interfaces

- Key inside record:

```
interface HashTable<T> {
  void add (T o);
  T get (T o);
}
```

Hash table interfaces

- Key inside record:

```
interface HashTable<T> {
  void add (T o);
  T get (T o);
}
```

E.g., user might want to store **`Student`** objects in the hash table. Then **`T`** would be **`Student`**.

```
class Student {
  int _studentId:
  String _firstName, _lastName;
  ...
}
```

Hash table interfaces

- Key inside record:

```
interface HashTable<T> {
  void add (T o);
  T get (T o);
}
```

Usage:

```
HashTableImpl<Student> hash = new HashTableImpl<Student>();
hash.add(new Student(123, “Surely”, “Temple”));  // O(1)
...
Student s = hash.get(new Student(123));  // O(1)
```

Hash table interfaces

- Key separate from record:

```
interface HashTable<K,V> {
  void put (K key, V value);
  V get (K key);
}
```

Hash table interfaces

- Key separate from record:

```
interface HashTable<K,V> {
  void put (K key, V value);
  V get (K key);
}
```

Here, the key type **K** could be **Integer** (for student id), and value type **V** would be **Student**:

```
class Student {

  String _firstName, _lastName;
  ...
}
```

Hash table interfaces

- Key separate from record:

```
interface HashTable<K,V> {
  void put (K key, V value);
  V get (K key);
}
```

Usage:

```
HashTableImpl<Integer,Student> hash = new HashTableImpl<Integer,Student>();
hash.add(123, new Student("Surely", "Temple"));  // O(1)
...
Student s = hash.get(123);  // O(1)
```

`hashCode()`

- Fundamental to all hash tables is the ability to convert an arbitrary object `o` into an `int`.
  - E.g., a `Student` object can be represented as an `int` using the student id.
- o's integer representation is used to determine where inside the hash table's intern array o will be stored

```
void add (T o) {
  int idx = hashFunction(o.hashCode());
  ...  // have to handle collisions
  _array[idx] = o;
}
```

`hashCode()`

- The Java **`Object`** class provides a built-in **`hashCode()`** method that converts every **`Object`** into an **`int`**.
  - By default, **`hashCode()`** simply returns the object's address in memory (an **`int`**).
  - Subclasses of **`Object`** can override **`hashCode()`** to do something more meaningful or to enhance performance, e.g.:

```
class Student {
  int _studentId;
  String _firstName, _lastName;
  public int hashCode () {
    return _studentId;
  }
}
```

Override
`Object.hashCode()`

Caches.

Caches

- Having concluded our discussion of hash tables, we can now show a useful example of *combining* two data structures to build a third: in this case, a *cache*.
- Consider a situation in which a program needs to retrieve data from a container that is *slow*.
  - The slow speed might arise due to a long distance over which the data must travel, or to the slow data rate at which a device can deliver information.

Caches

- Examples:
  - A web browser downloads a webpage from an *external server*. Server is far away.
  - A spreadsheet program loads a file from *disk*. Disk is slow.
  - The CPU must read the value of a variable stored in *main memory* (instead of on-chip storage). RAM is slow.
- In each case, the program *fetches* data from *secondary storage* and loads it into *primary storage*.
  - Primary storage is faster and "closer" to the user than secondary storage.
  - What is "slow" in one context may be "fast" in another.

Caches

- Examples:
  - A web browser downloads a webpage from an *external server*.
    - Primary storage: computer memory (RAM) and/or disk.
    - Secondary storage: web server.
  - A spreadsheet program loads a file from *disk*.
    - Primary storage: computer memory (RAM).
    - Secondary storage: disk.
  - The CPU must read the value of a variable stored in *main memory* (instead of on-chip storage).
    - Primary storage: CPU registers.
    - Secondary storage: computer memory (RAM).

Caches

- Now, suppose that the *same* data X tends to be fetched from secondary storage *repeatedly*.
- In this case, we can save time by introducing an *intermediary* data container -- a *cache* -- that "remembers" the data fetched from secondary storage.
- A **cache** is a data structure that offers *high-speed* access to a *small* amount of data that must otherwise be written to/read from a *slower*, secondary storage container.

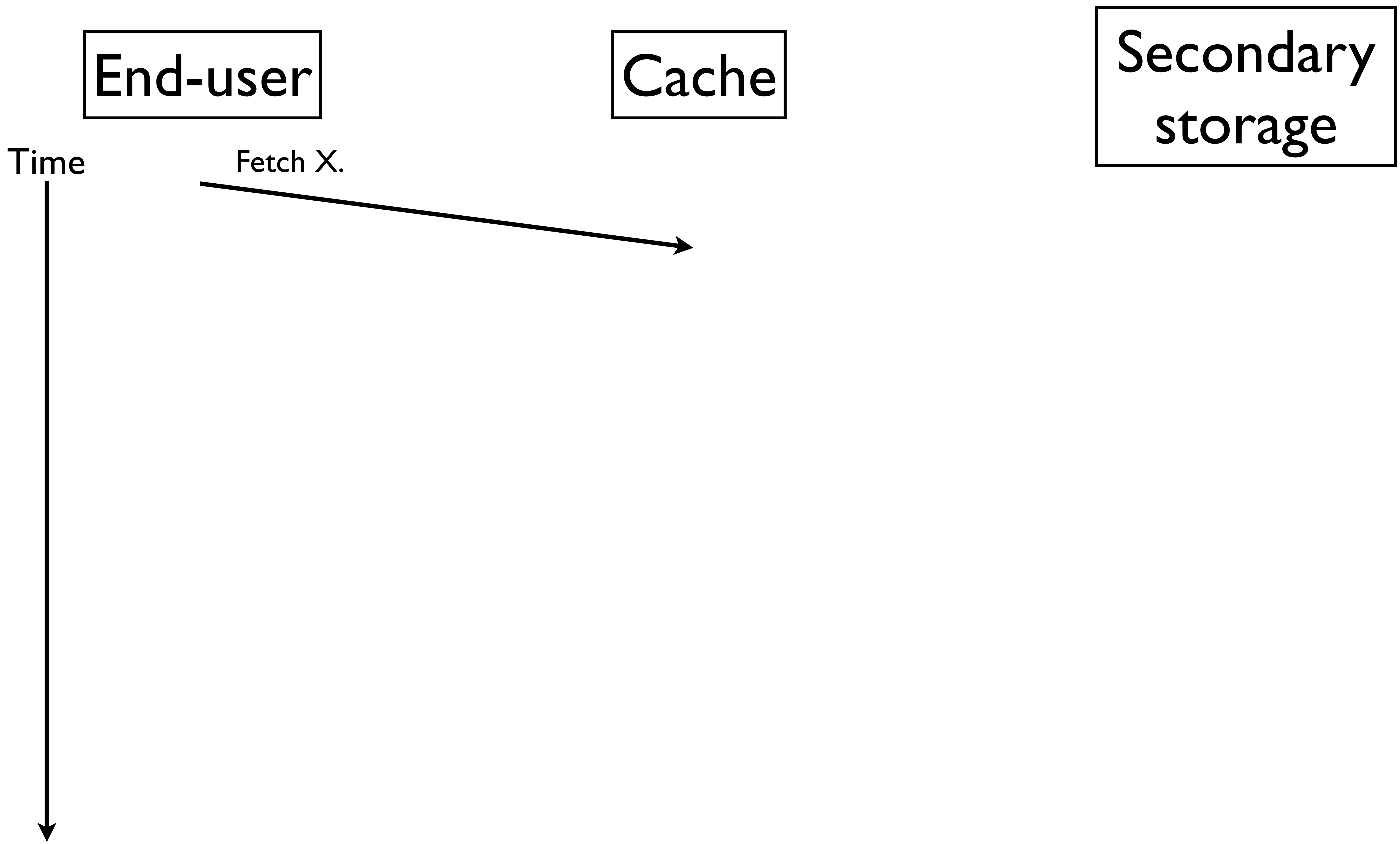
Caches: small and fast

- Caches are inherently *fast* and *small*:
  - *Fast* because they reside in primary storage, not secondary storage.
    - If they were slow, we'd forget the cache and just access secondary storage directly.
  - *Small* because they are typically more expensive than secondary storage.
    - If they were cheap, we'd just store *everything* in the cache and forget secondary storage.

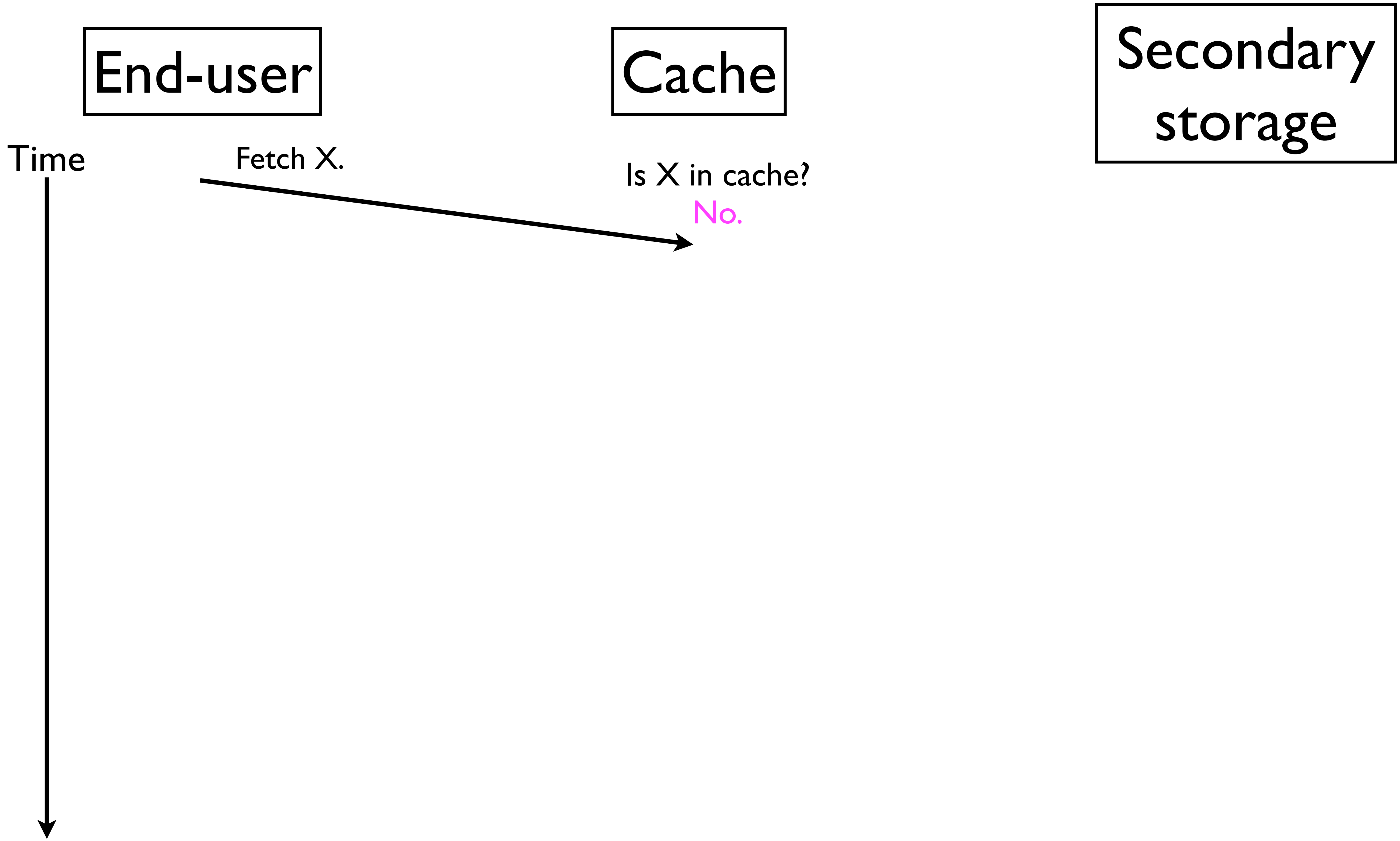
Caches in action

- A user’s request to fetch data X from secondary storage is “intercepted” by the cache:
  - If the cache already contains X, then the cache *returns* X to the user immediately.
    - Fetching X from secondary storage is unnecessary.
  - Otherwise (cache does not contain X), the cache *forwards* the user’s request to secondary storage.
- Both *read* and *write* caches exist; here, we deal only with *read* caches.

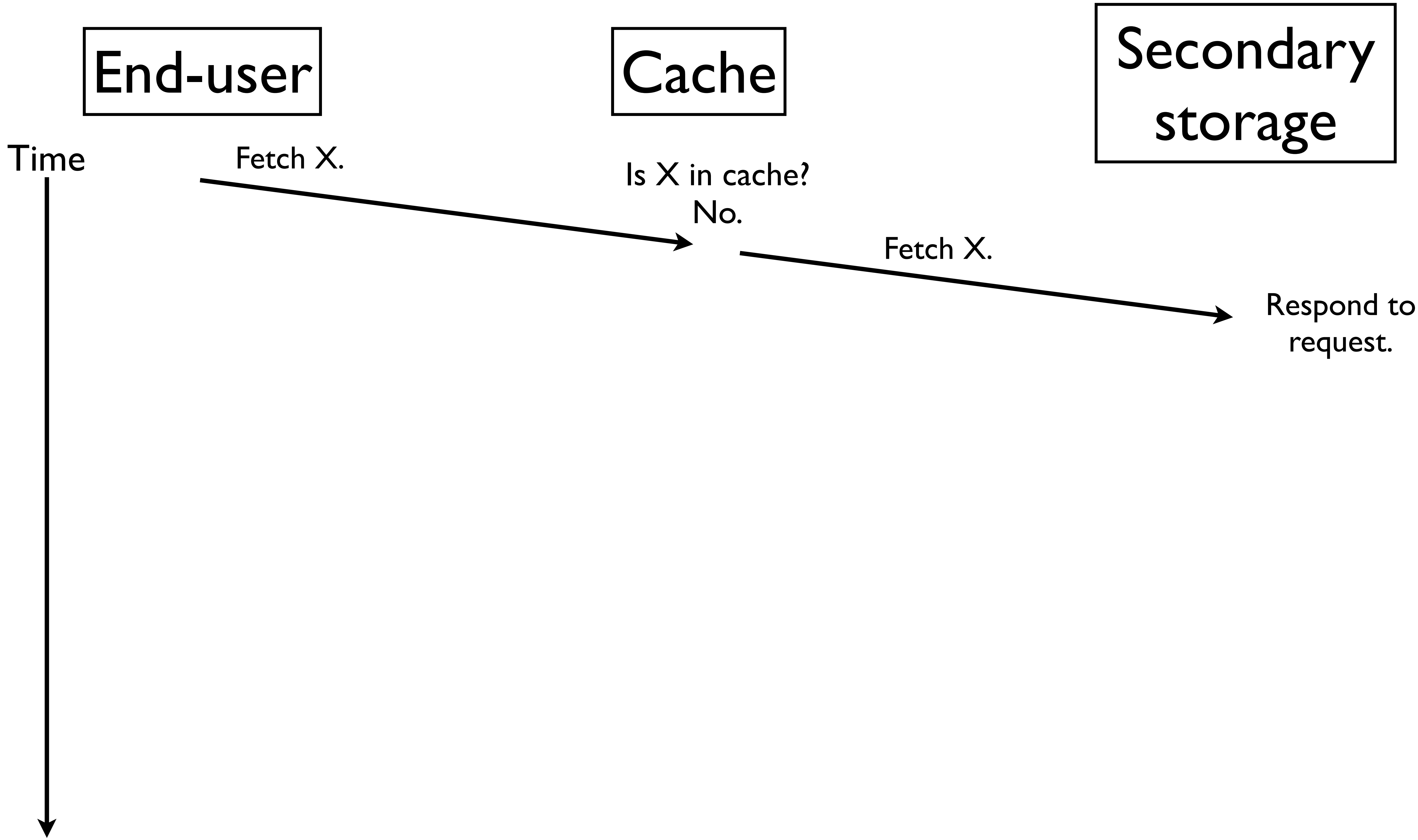
Caches



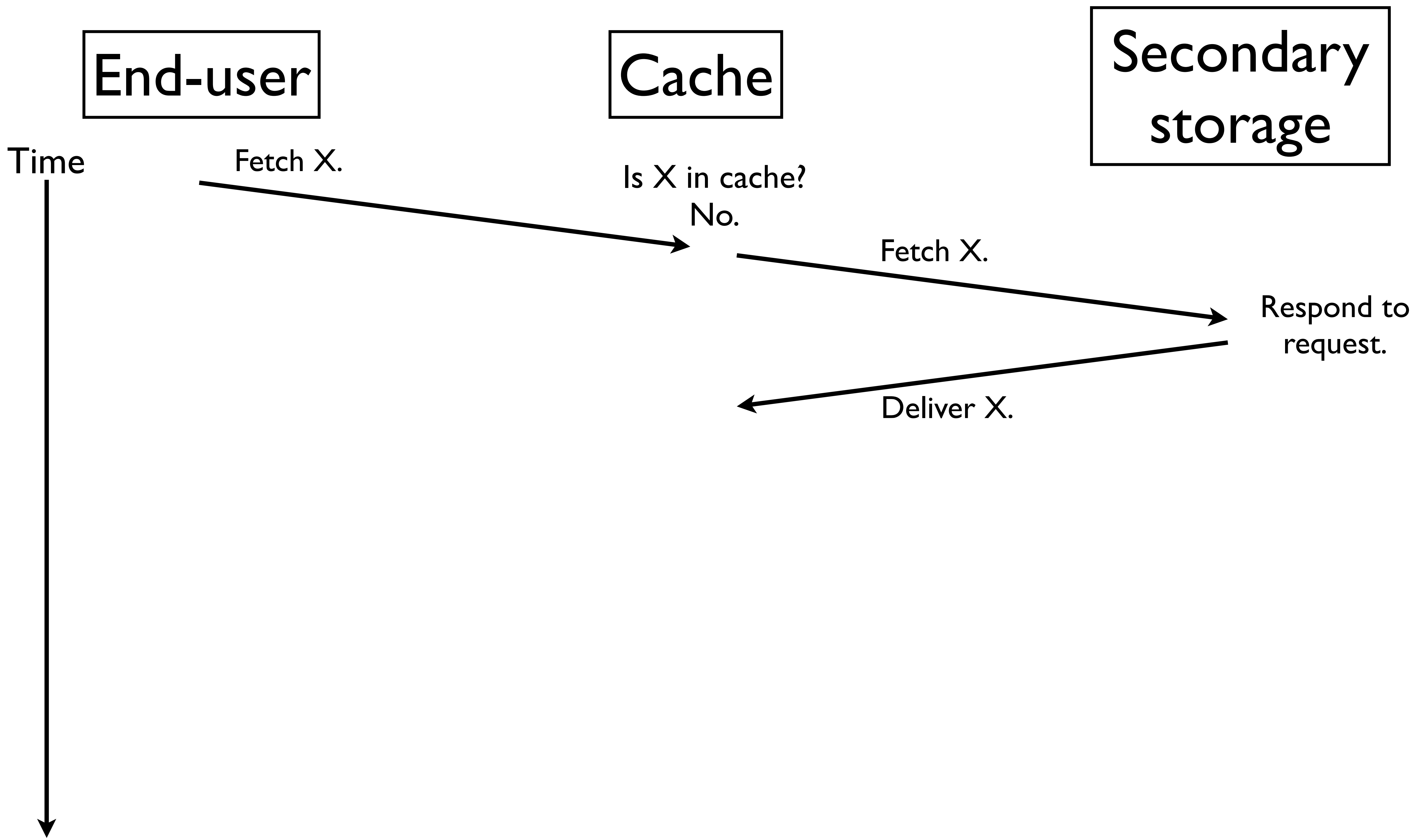
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache?
No.

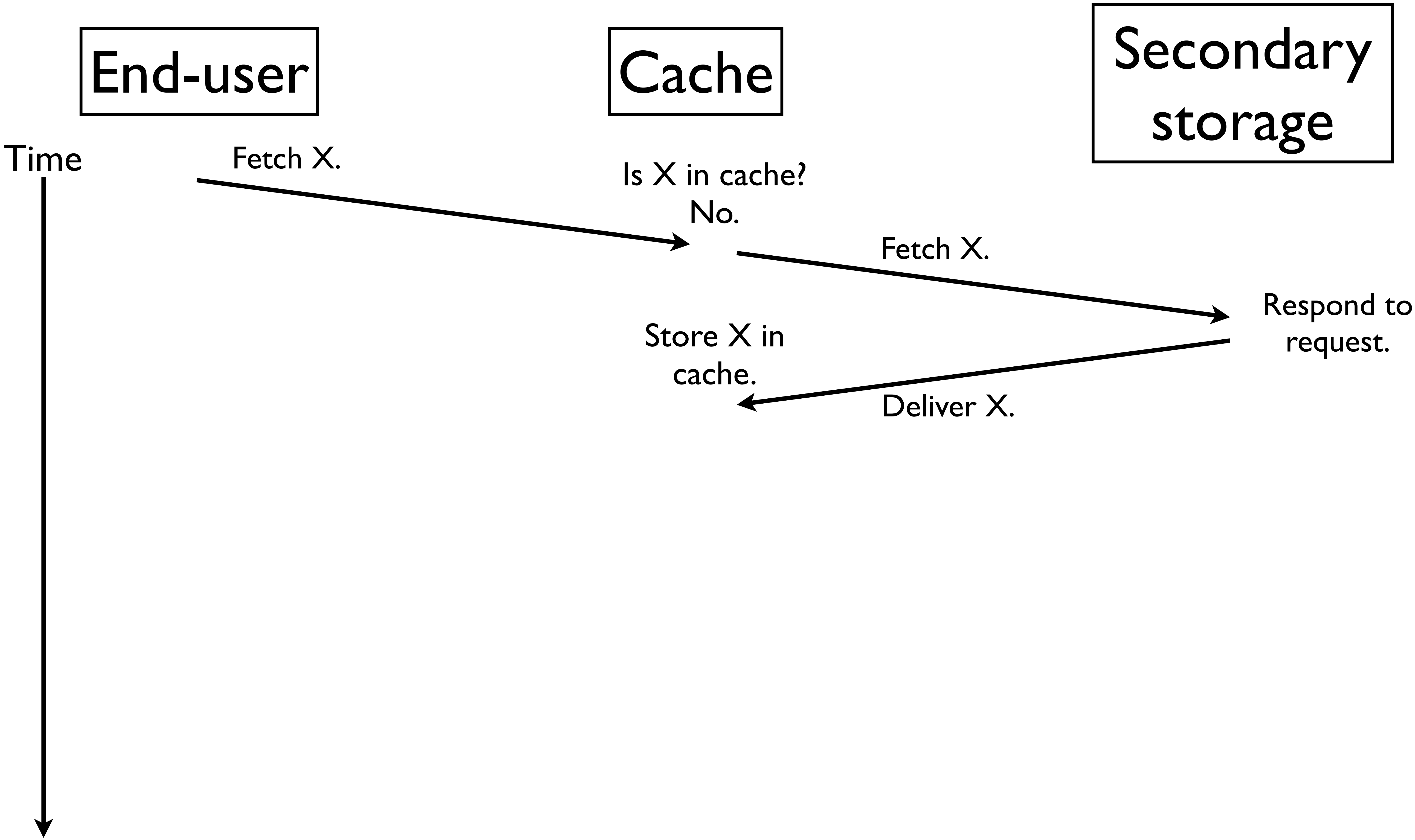
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache?
No.
Fetch X.
Respond to request.

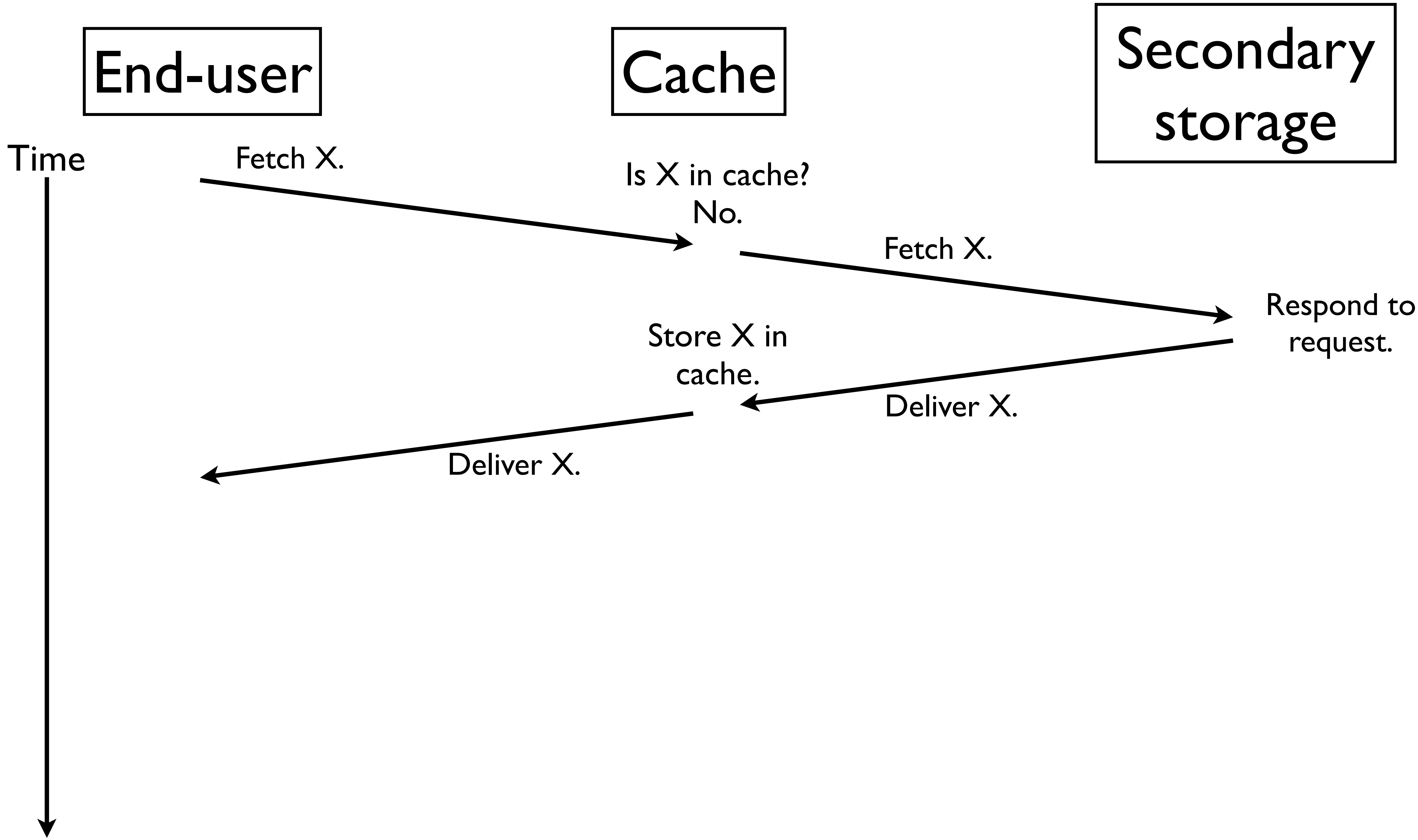
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Deliver X.

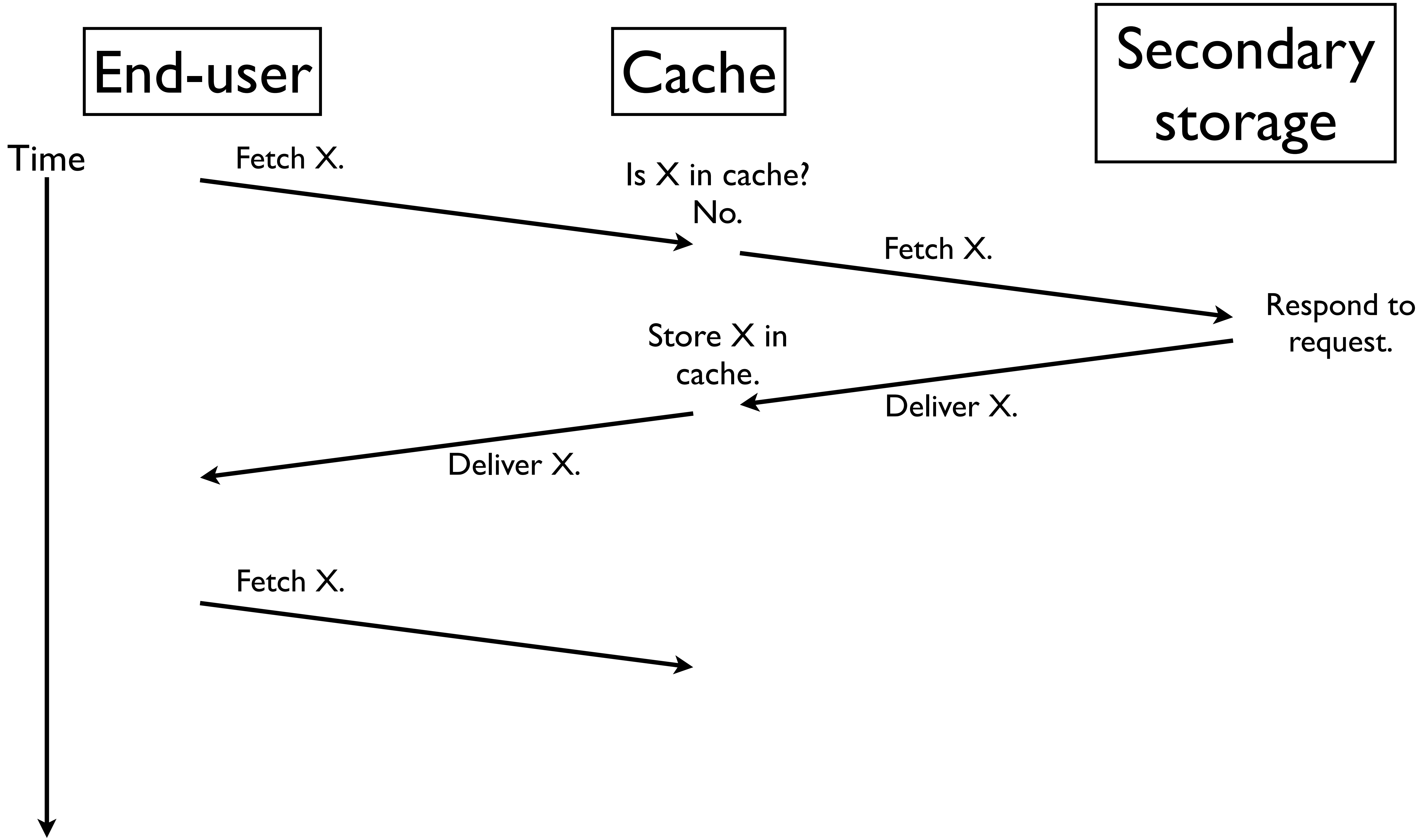
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Store X in cache.
Deliver X.

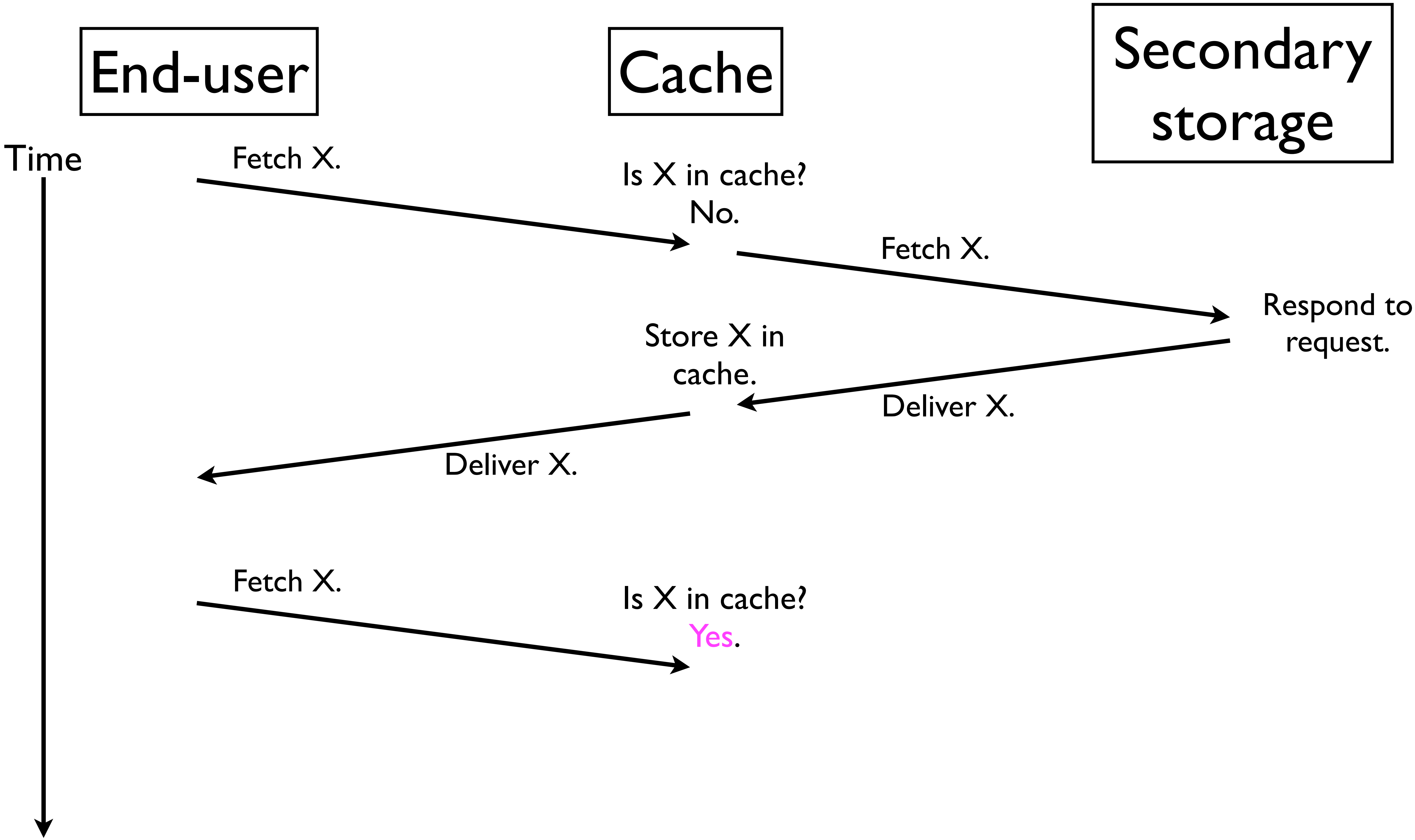
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Store X in cache.
Deliver X.
Deliver X.

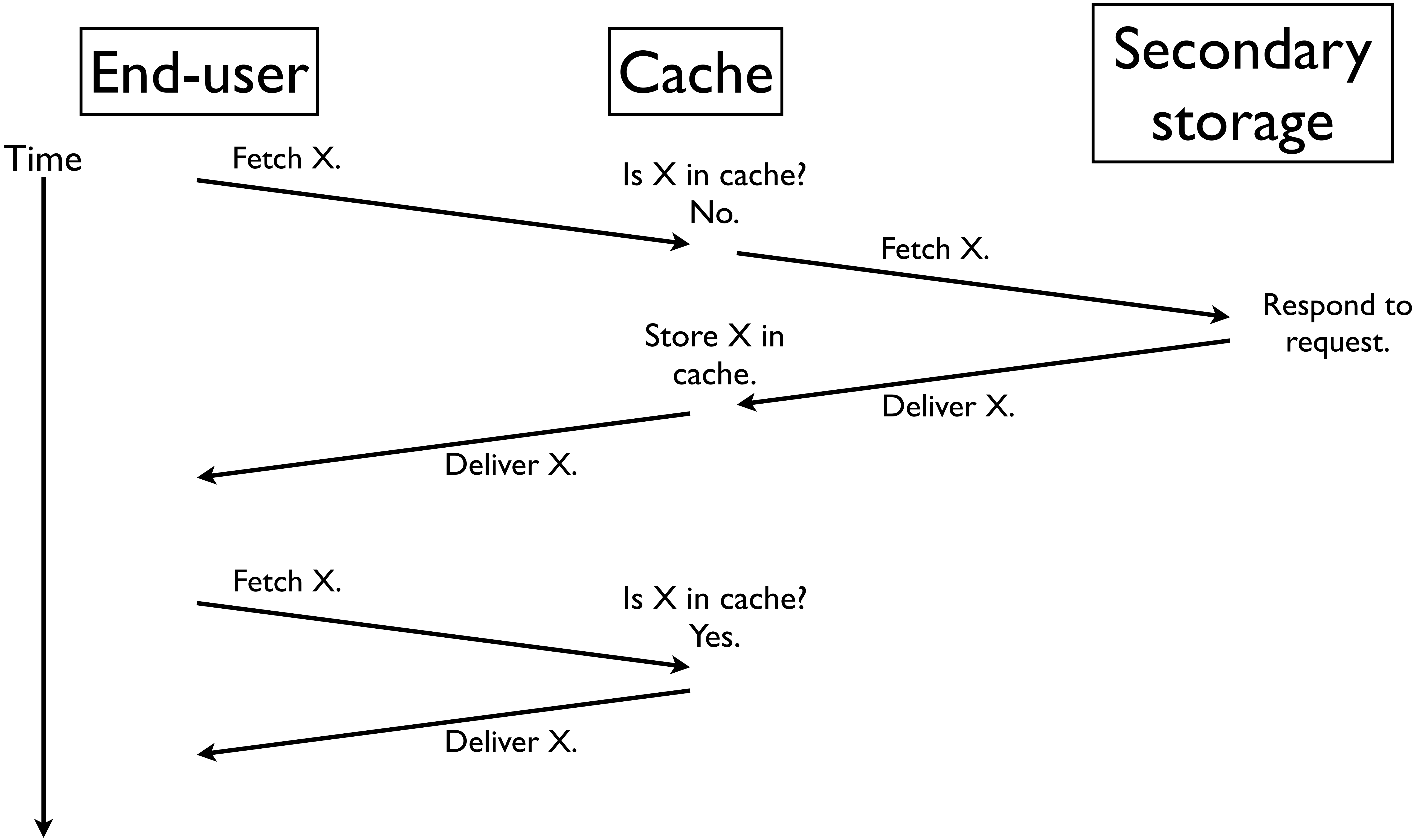
Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Store X in cache.
Deliver X.
Deliver X.
Fetch X.

Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Store X in cache.
Deliver X.
Deliver X.
Fetch X.
Is X in cache? Yes.

Caches


End-user
Cache
Secondary storage
Time
Fetch X.
Is X in cache? No.
Fetch X.
Respond to request.
Store X in cache.
Deliver X.
Deliver X.
Fetch X.
Is X in cache? Yes.
Deliver X.

Caches: definitions

- If the user requests item X from the cache, and X is contained in the cache, then we have a **cache hit**.
- Otherwise, if X is *not* in the cache, then we have a **cache miss**.
    - X must then be fetched from secondary storage.
- The size of the cache is always *finite*.
- For every cache miss: if the cache is *full*, the cache must decide which element to "forget", i.e., **evict**.
- The choice of which data to evict can affect the cache **miss rate** (fraction of cache accesses that miss) and thereby the performance of the computer system.

Eviction policies

- The algorithm that decides which object to evict is called an **eviction policy**.
- The choice of eviction policy can make a large impact on system performance.
- An *optimal* eviction policy determines which element o in the cache will not be used again for the longest period of time, and then evicts o.
  - This minimizes the expected cache miss rate.
- Unfortunately, this optimal policy is rarely achievable because it's difficult to predict which items will be needed in the future.

Least-recently-used caches

- One of the most commonly implemented eviction policies is *least-recently-used* (LRU).
- Whenever we must evict an element from the cache, we pick the least-recently-used element.
    - *Justification*: It seems reasonable that an item that has not been used in a long time will continue not to be requested for a while longer.
- Empirically, LRU has shown to perform "similarly" to the *optimal* eviction policy in many practical applications.

LRU in action

Time

Cache contents

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

LRU in action

Time

Cache contents

Cache miss

A

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

LRU in action

Time

Cache contents

A

Cache miss — A B

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

Time	Cache contents	
	A	
	A	**B**
	A	**B**

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

	Time	Cache contents		
		A		
		A	B	
		A	B	
Cache miss		A	C	B was LRU.

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

Time

Cache contents	
A	
A	B
A	B
A	C
A	C

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

Time	Cache contents		
	A		
	A	B	
	A	B	
	A	C	
	A	C	
Cache miss	A	B	C was LRU.

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

Time

Cache contents	
A	
A	B
A	B
A	C
A	C
A	B
A	B

LRU in action

- How would an LRU cache (with 2 slots) handle the following sequence of requests?
  - A B A C A B B C

Time	Cache contents		
	A		
	A	B	
	A	B	
	A	C	
	A	C	
	A	B	
	A	B	
Cache miss	C	B	A was LRU.

There were 5 cache misses out of 8 accesses; hence, cache miss rate is 0.625.

LRU Cache

- We wish to construct a Cache ADT that uses the LRU eviction policy.
- The cache will mediate access to some other, arbitrary secondary storage container.
- The user will request data by calling `Cache.get(key)` and expect the associated *value* to be returned.
- If `key` is not stored in the cache, then the cache should forward the request to the secondary storage.

LRU Cache interface

- Before designing a Java interface for the LRU cache, let's first conceptualize how the user might access the secondary storage *without* the cache.
- Suppose the secondary storage has the following interface:

```
interface Storage<K,V> {
  // Fetches and returns the data specified by key
  V get (K key);
}
```

- Here, the *key* might be the URL of a web page we're fetching, and the *value* might be the web page itself, e.g.:

```
WebServer<String,Webpage> server = new WebServer<String,Webpage>();
Webpage page = server.get("http://my.website.com");
```

LRU Cache interface

- Now, let's define a Java interface for an LRU cache:

```
// Least-recently-used (LRU) cache.
// The get(key) method should take O(1) time
// for an n-element cache.
//
// Implementing classes should offer a
// constructor with one parameter of type
// Storage that specifies the cache's
// secondary storage.
interface LRUCache<K,V> {
  V get (K key);
}
```

LRU Cache usage

- Instead of writing:

```
WebServer<String,Webpage> server = new WebServer<String,Webpage>();
Webpage page = server.get("http://my.website.com");
...
page = server.get("http://my.website.com");
...
page = server.get("http://my.website.com");
```

- ...we write instead:

```
WebServer<String,Webpage> server = new WebServer<String,Webpage>();
LRUCache<String,Webpage> cache =
  new LRUCacheImpl<String,Webpage>(server);
Webpage page = cache.get("http://my.website.com");
...
page = cache.get("http://my.website.com");
...
page = cache.get("http://my.website.com");
```

LRU Cache usage

- Instead of writing:

```
WebServer<String,Webpage> server = new WebServer<String,Webpage>();
Webpage page = server.get(“http://my.website.com”);
...
page = server.get(“http://my.website.com”);
...
page = server.get(“http://my.website.com”);
```

- ...we write instead:

```
WebServer<String,Webpage> server = new WebServer<String,Webpage>();
LRUCache<String,Webpage> cache =
  new LRUCacheImpl<String,Webpage>(server);
Webpage page = cache.get(“http://my.website.com");      Cache miss: call server.get(...)
...
page = cache.get(“http://my.website.com”);              Cache hit
...
page = cache.get(“http://my.website.com”);              Cache hit
```

LRU Cache implementation

- The LRUCache interface imposes the constraint that **`get(key)`** must operate in $O(1)$ time for an n-element cache.
- Each call to **`get(key)`** must potentially:

1. Determine whether the desired object (specified by **`key`**) is stored in the cache in $O(1)$ time.
2. If **`key`** *is* in cache, then:

   (a) Make **`key`** the MRU item in $O(1)$ time.

   (b) Return the **`key`**'s associated *value* in $O(1)$ time.

LRU Cache implementation

3. Else (`key` is *not* in cache):

    (a) Call `value = _secondaryStorage.get(key)`.

    (b) Find the *least*-recently-used (LRU) item in $O(1)$ time.

    (c) Replace the LRU item with (`key,value`), which is now the *most*-recently-used (MRU) item in the cache, in $O(1)$ time.

LRU Cache implementation

- To associate each key with its value, we need a **`Node`** (inner-)class:

```
static class Node {
  K _key;
  V _value;
}
```

But what will be the "underlying storage" for the cache entries themselves?

LRU Cache implementation

- Implementation sketch of **`LRUCache`**:

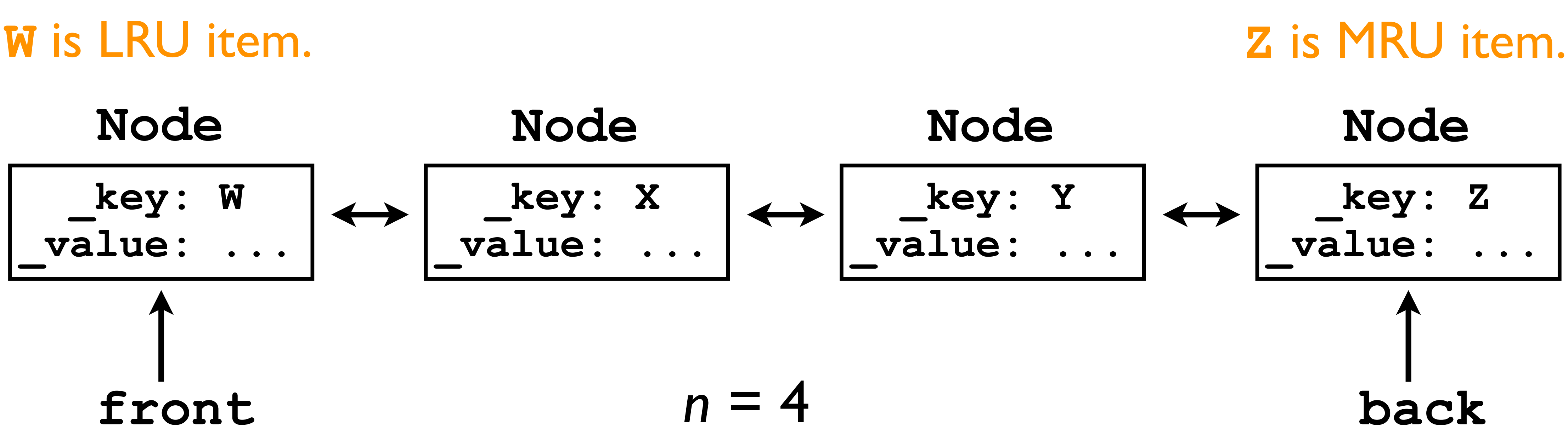
```
class LRUCacheImpl<K,V> implements LRUCache<K,V>{
  static class Node {
    K _key;
    V _value;
  }
  Storage<K,V> _secondaryStorage;
  ...

  LRUCacheImpl (Storage<K,V> secondaryStorage) {
    _secondaryStorage = secondaryStorage;
  }

  V get (K key) {
    // If key in cache
    //    Fetch value from cache
    // Else
    //    value = _secondaryStorage.get(key);
    //    Store value in cache (evict LRU if necessary)
    // Make key the MRU item
    // Return value;
  }
}
```

But what will be the "underlying storage" for the cache entries themselves?

LRU Cache implementation

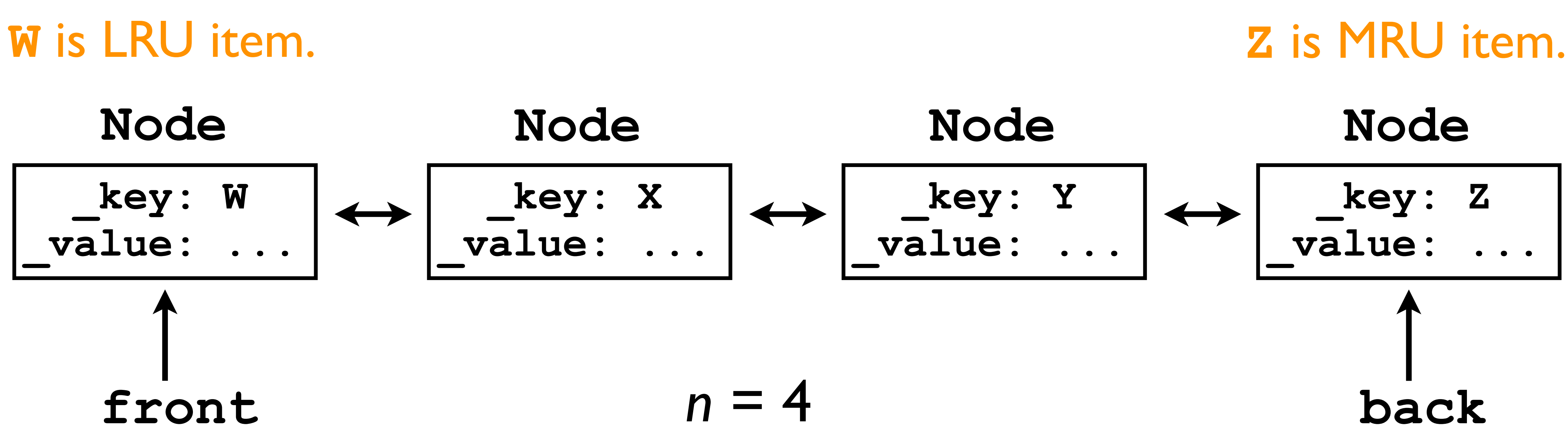
- Our "underlying storage" will consist of 2 components:

1. A *queue* of **`Nodes`** to hold the *relative order* in which data are accessed.
    - For *n*-element cache, max length of queue is *n*.
    - LRU at the *front*, MRU at the *back* of the queue.
    - Each **`Node`** will contain both a *key* (e.g., URL) and corresponding *value* (e.g., webpage).



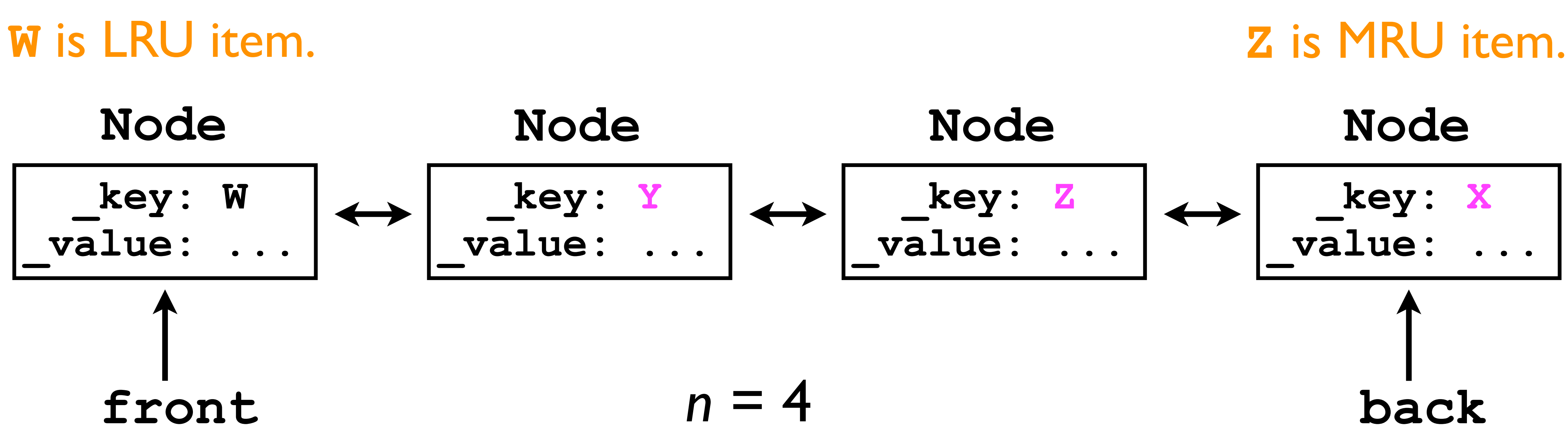
LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its **`Node`** to the *back* of the queue because it's now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
  - We must evict the LRU item to make room.



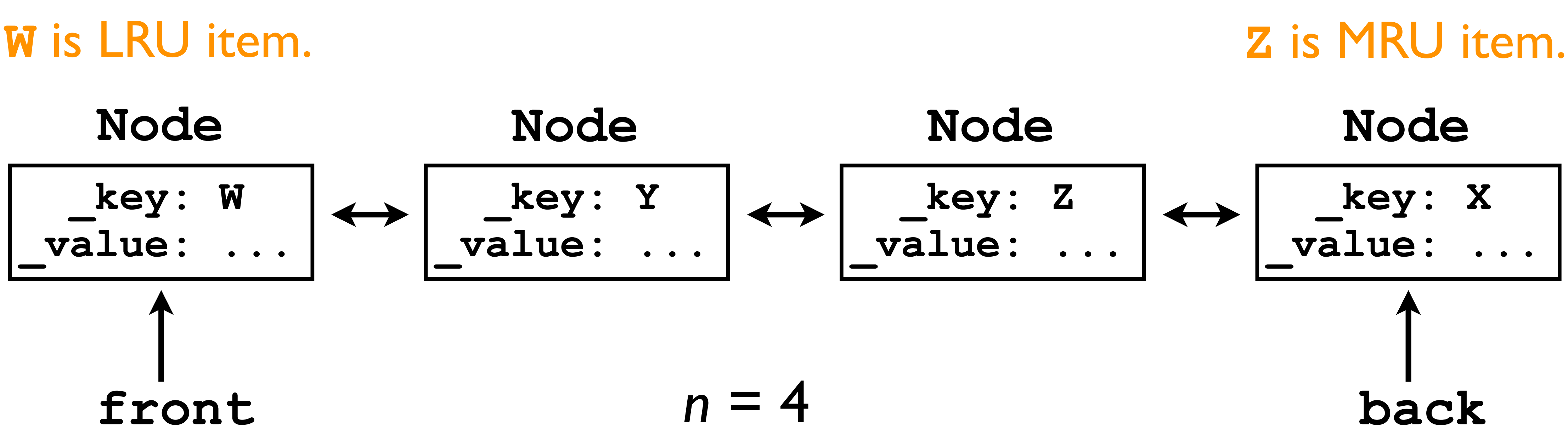
LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its `Node` to the *back* of the queue because it's now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
  - We must evict the LRU item to make room.



LRU Cache implementation

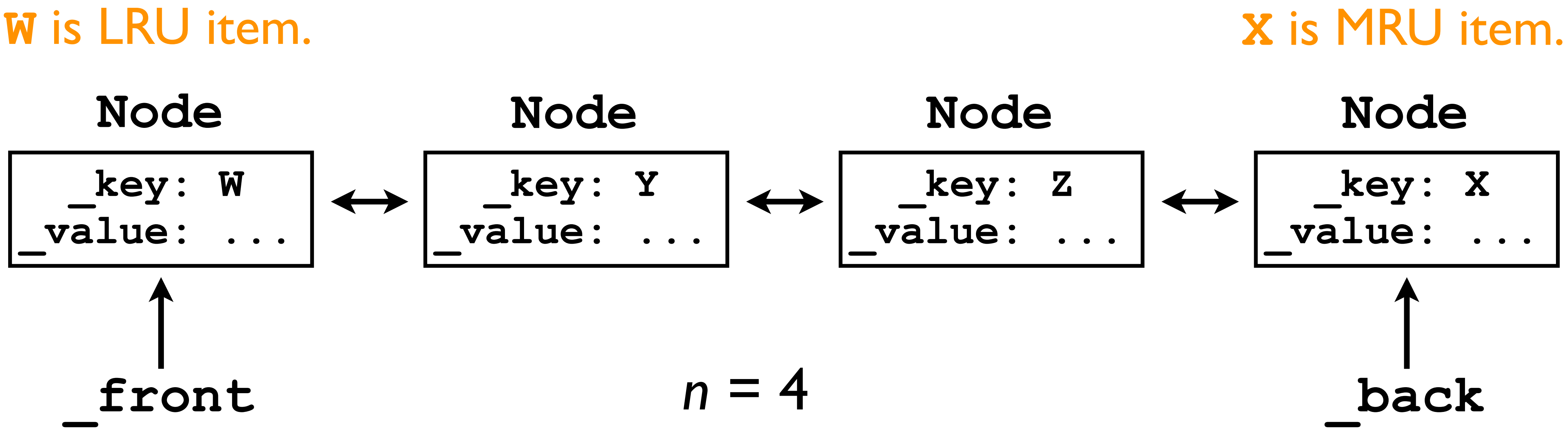
- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its **`Node`** to the *back* of the queue because it's now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
  - We must evict the LRU item to make room.



LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its **`Node`** to the *back* of the queue because it's now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.

  - We must evict the LRU item to make room.

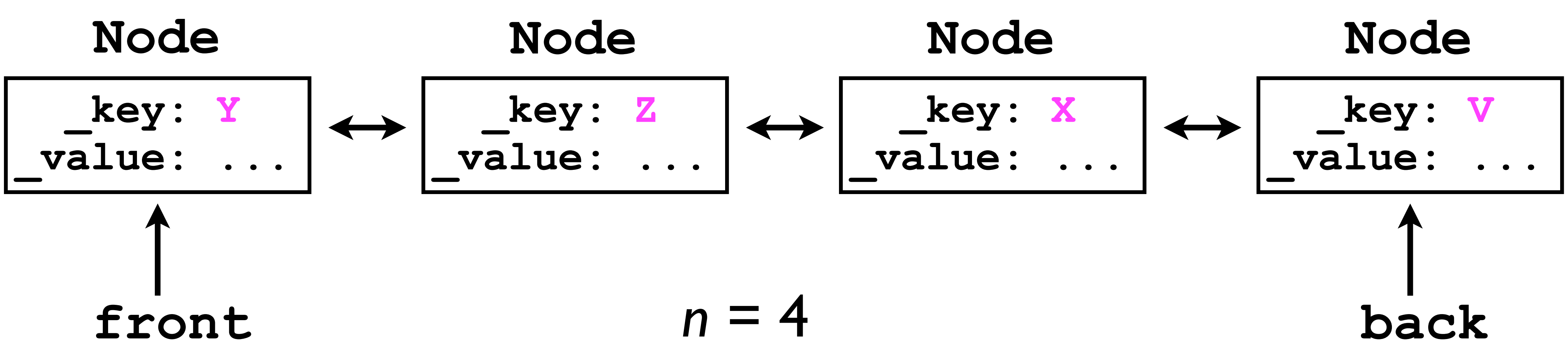


LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its **`Node`** to the *back* of the queue because it's now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
  - We must evict the LRU item to make room.

`W` was LRU item and was evicted.

`V` is now MRU item.



Reality check

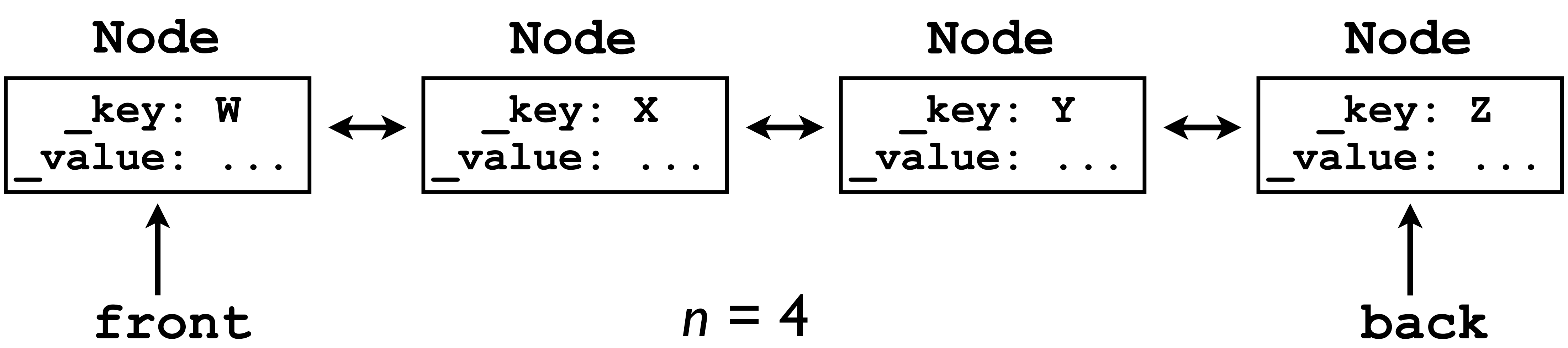
- Suppose the cache stores $n = 3$ elements, and suppose the user requests the following webpages in the following order:

```
cnn.com
google.com
gmail.com
yahoo.com
npr.org
gmail.com
wikipedia.org
gmail.com
npr.org
cnn.com
imdb.com
```

- Show the queue at each step.

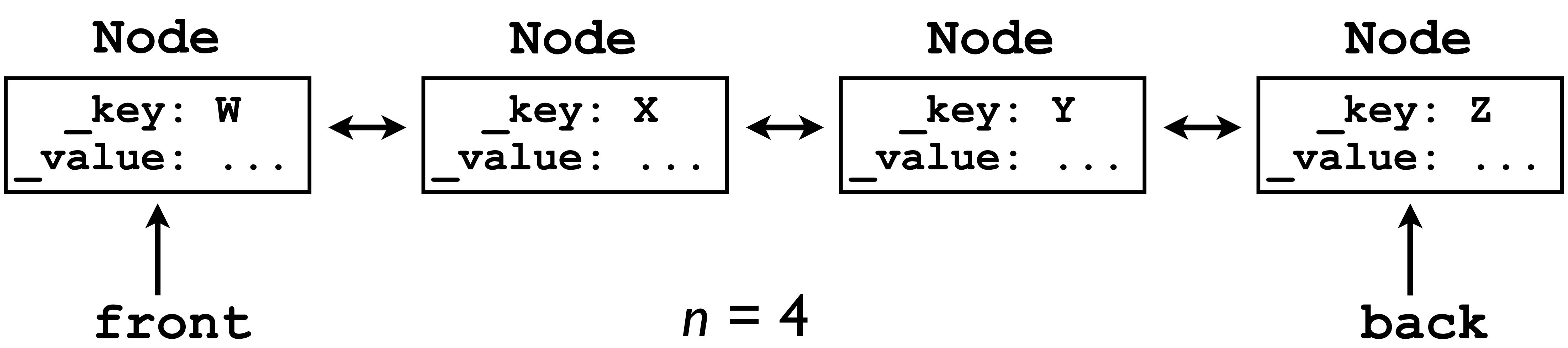
LRU Cache implementation

- Unfortunately, a queue by itself will not suffice to implement the **`LRUCache`** interface.
  - When we want to update a **`Node`**'s position in the queue to MRU, we have to *find* the node.
    - If we just search linearly through the queue, this takes time $O(n)$ (slow).



LRU Cache implementation

- Instead, we can use an additional **HashTable<K,Node>** to “jump” to the desired **Node**.
    - This only takes $O(1)$ time.



LRU Cache implementation

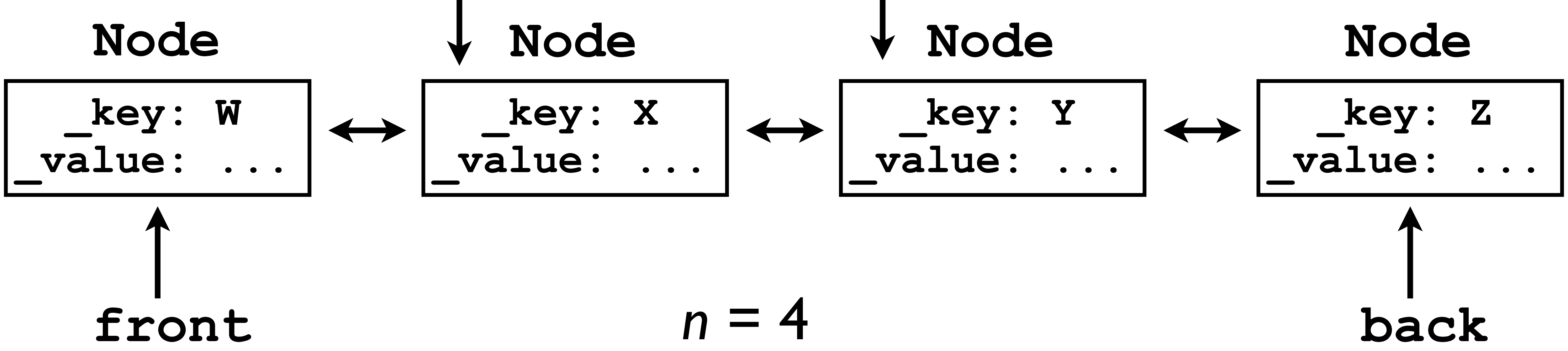
- Every **`key`** stored in the *queue* will also have an entry in a *hash table*.

`_keysToNodesTable`

Key	**Node**
X	
Y	
...	...

The *hash table* affords $O(1)$ access to any cache item, given its key.

The *queue* affords $O(1)$ access to the LRU item (**`_front`**) in the cache.



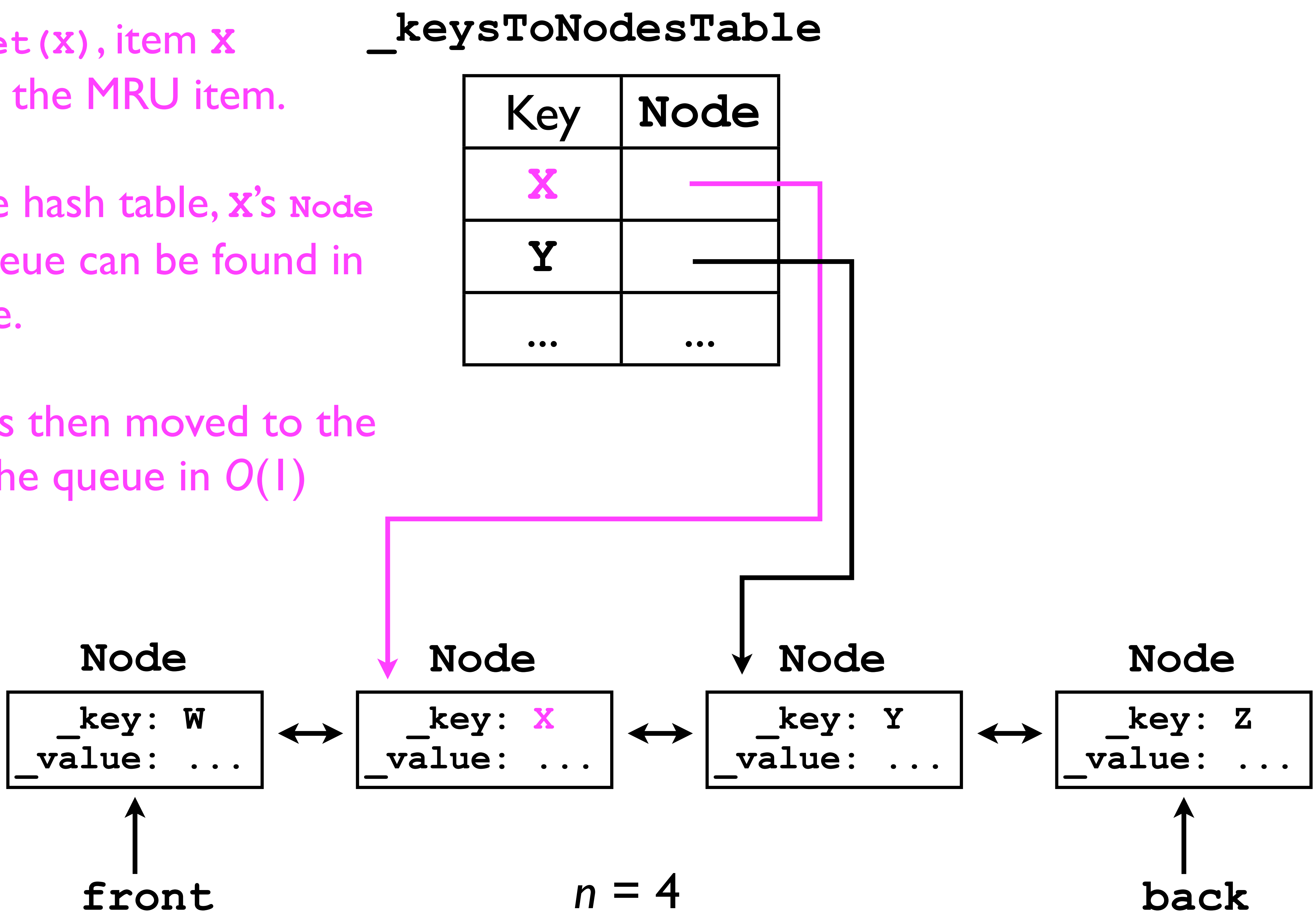
$n = 4$

LRU Cache implementation

Whenever the user calls `cache.get(X)`, item `X` becomes the MRU item.

Using the hash table, `X`'s `Node` in the queue can be found in $O(1)$ time.

Its `Node` is then moved to the *back* of the queue in $O(1)$ time.

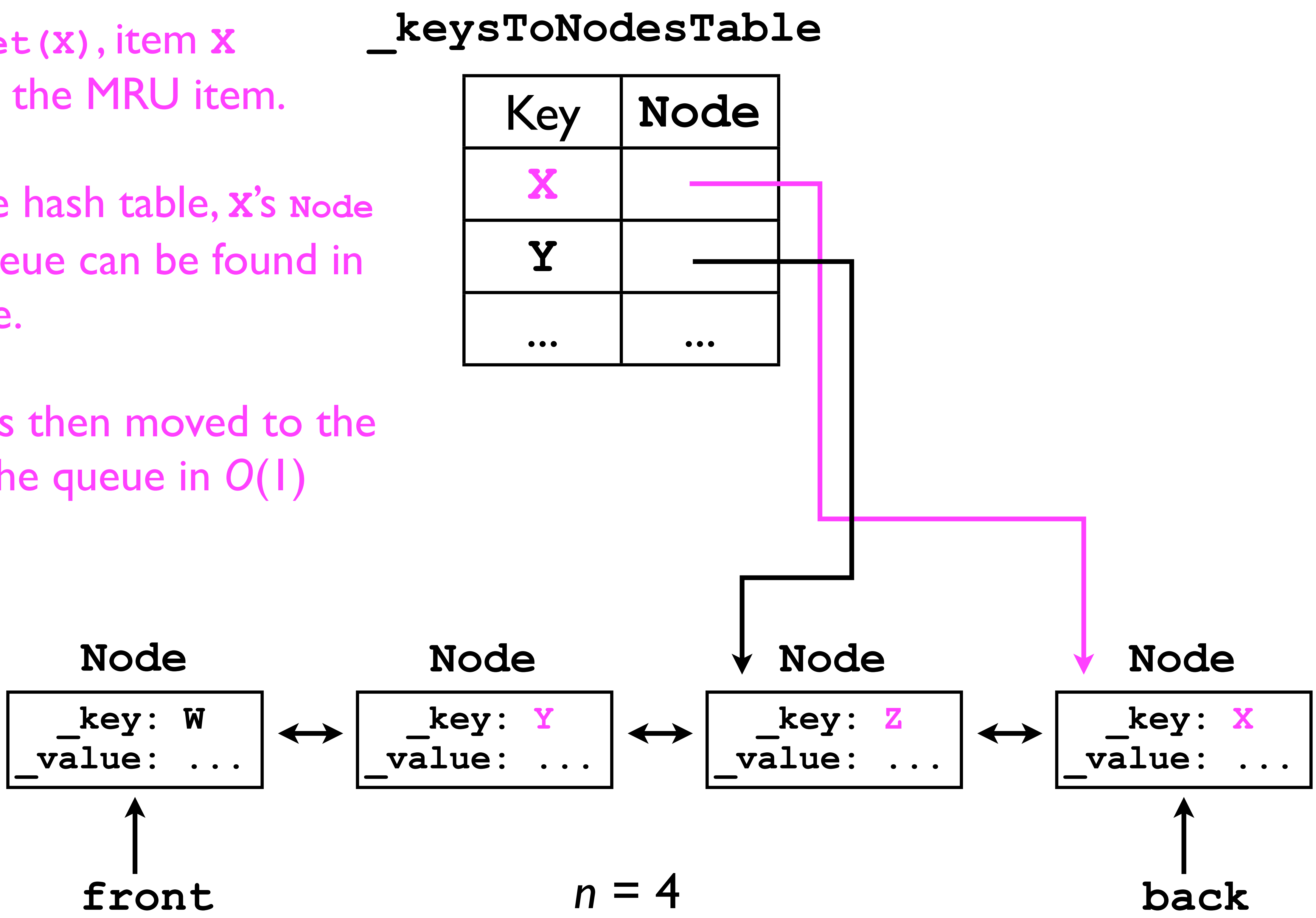


LRU Cache implementation

Whenever the user calls `cache.get(X)`, item `X` becomes the MRU item.

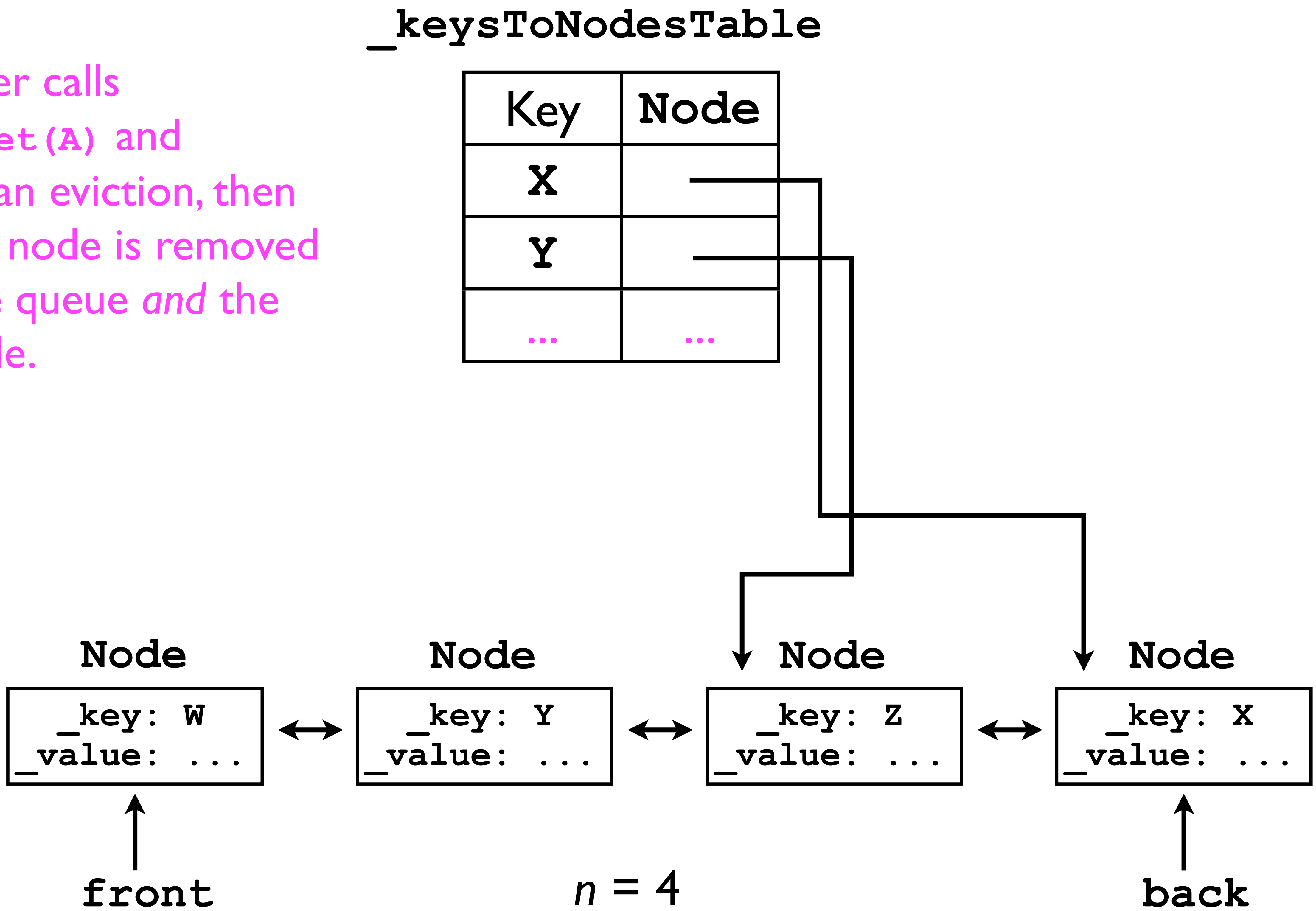
Using the hash table, `X`'s `Node` in the queue can be found in $O(1)$ time.

Its `Node` is then moved to the *back* of the queue in $O(1)$ time.



LRU Cache implementation

If the user calls `cache.get(A)` and triggers an eviction, then the LRU node is removed from the queue *and* the hash table.



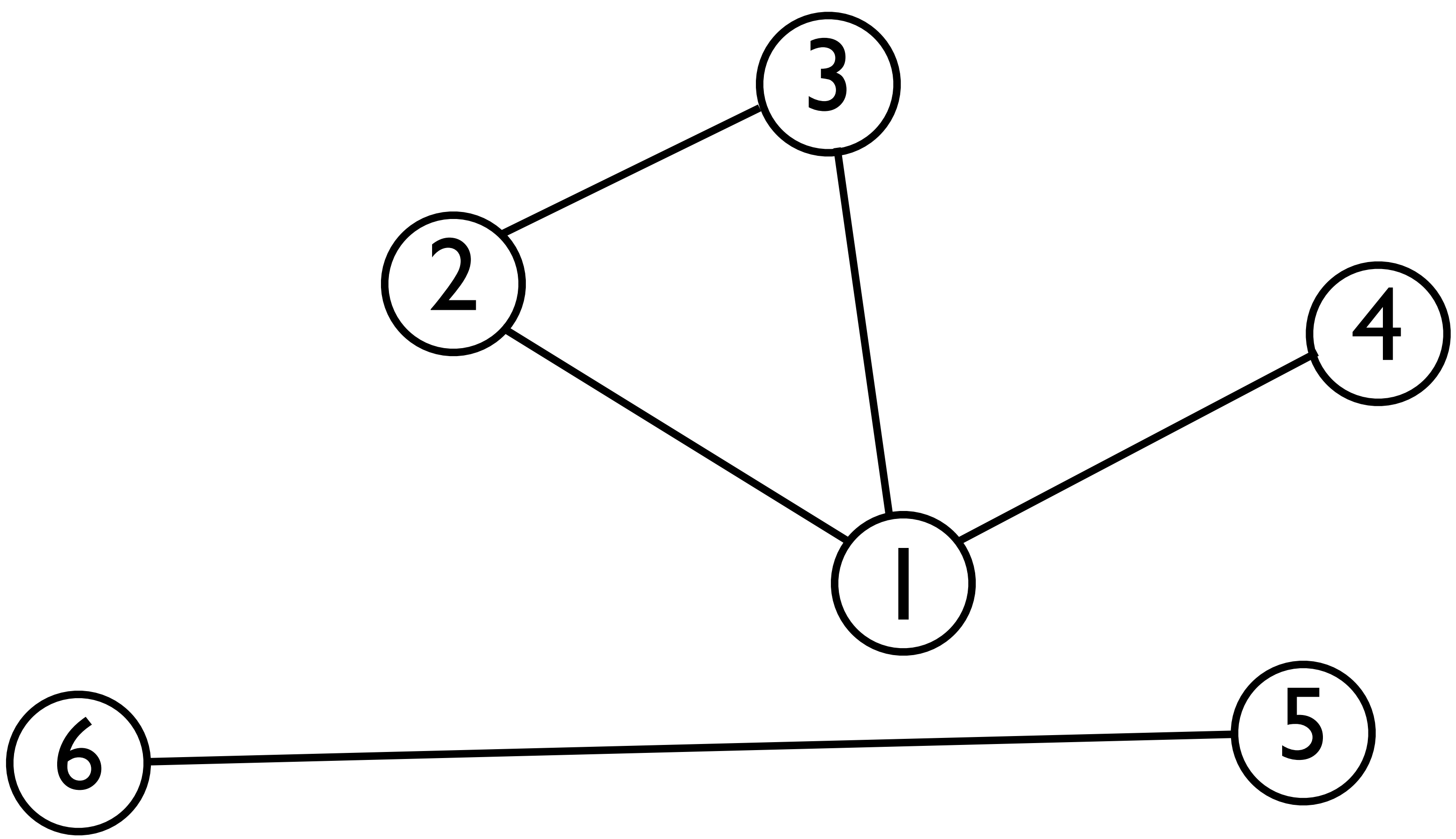
LRU Cache implementation

- In summary:
  - An LRU cache is an example of combining data structures to harness their individual strengths.
  - To implement an LRU cache with *O*(1) time for **`V get (K key)`**, we need fast access both to the LRU item, *and* to an *arbitrary* item specified by **`key`**.
  - A *queue* gives us *O*(1) access to the LRU item (front of queue).
  - A *hash table* gives us *O*(1) access to an arbitrary **`Node`** in the queue.

Graphs.

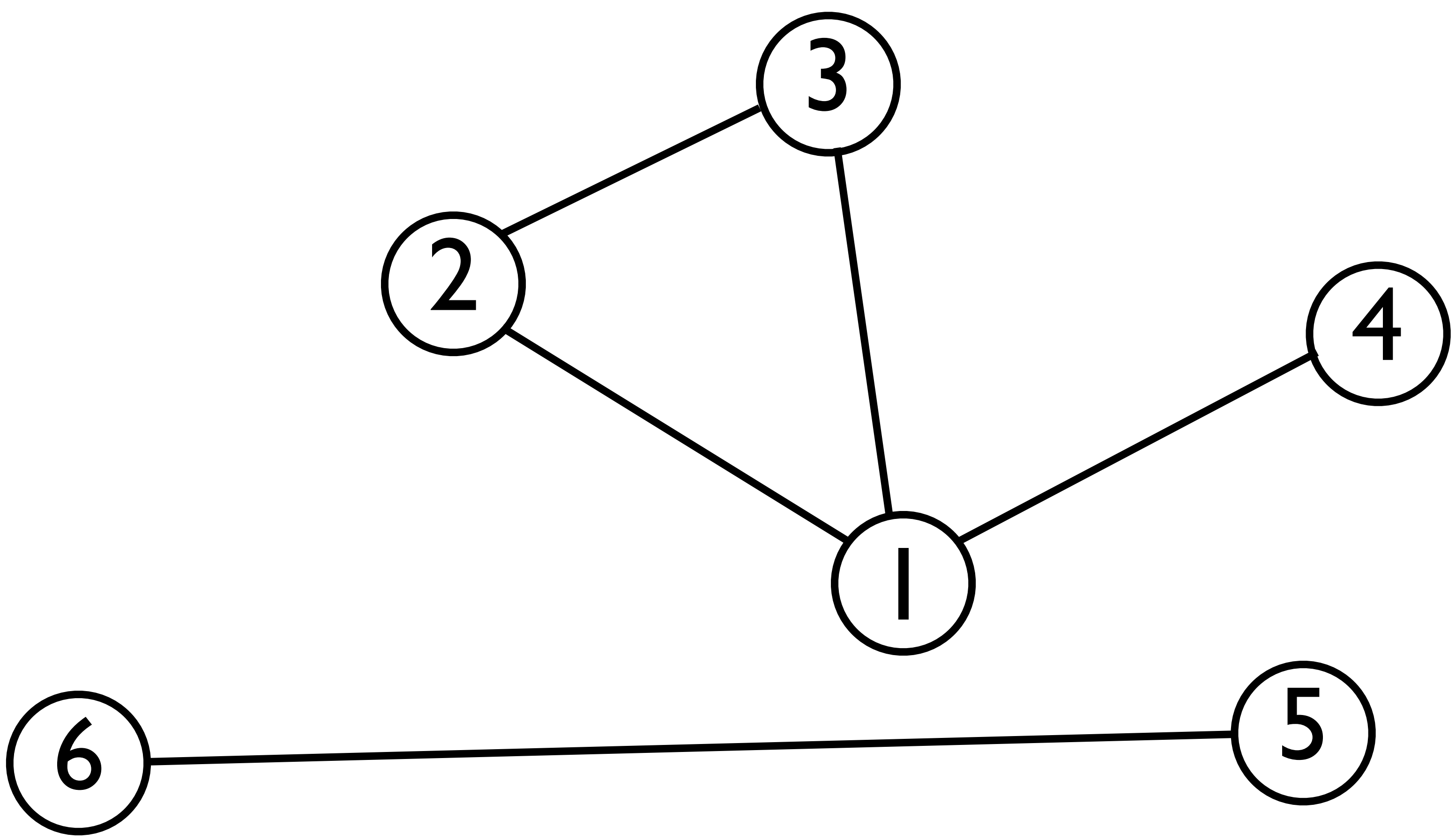
Graphs

- The last fundamental data structure we will cover in this course is a *graph*.
- Mathematically, a **graph** consists of a set N of **nodes** (aka **vertices**) connected by a set E of **edges**.



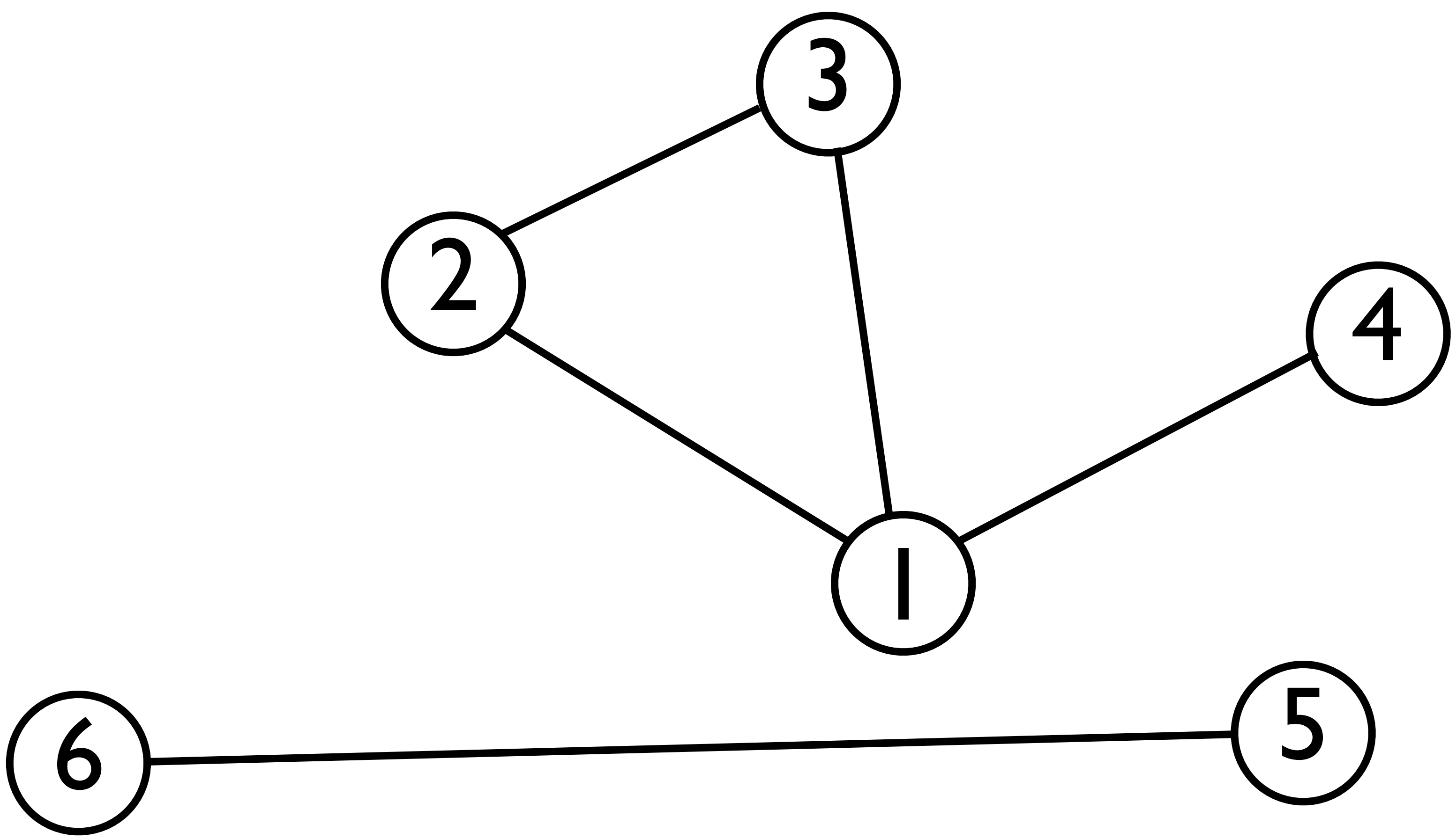
Graphs

- In computer science, graphs are useful for describing *relationships* (edges) among *things* (nodes).
  - E.g., each node might represent a *Facebook user*, and each edge might represent whether two Facebook users are *friends*.



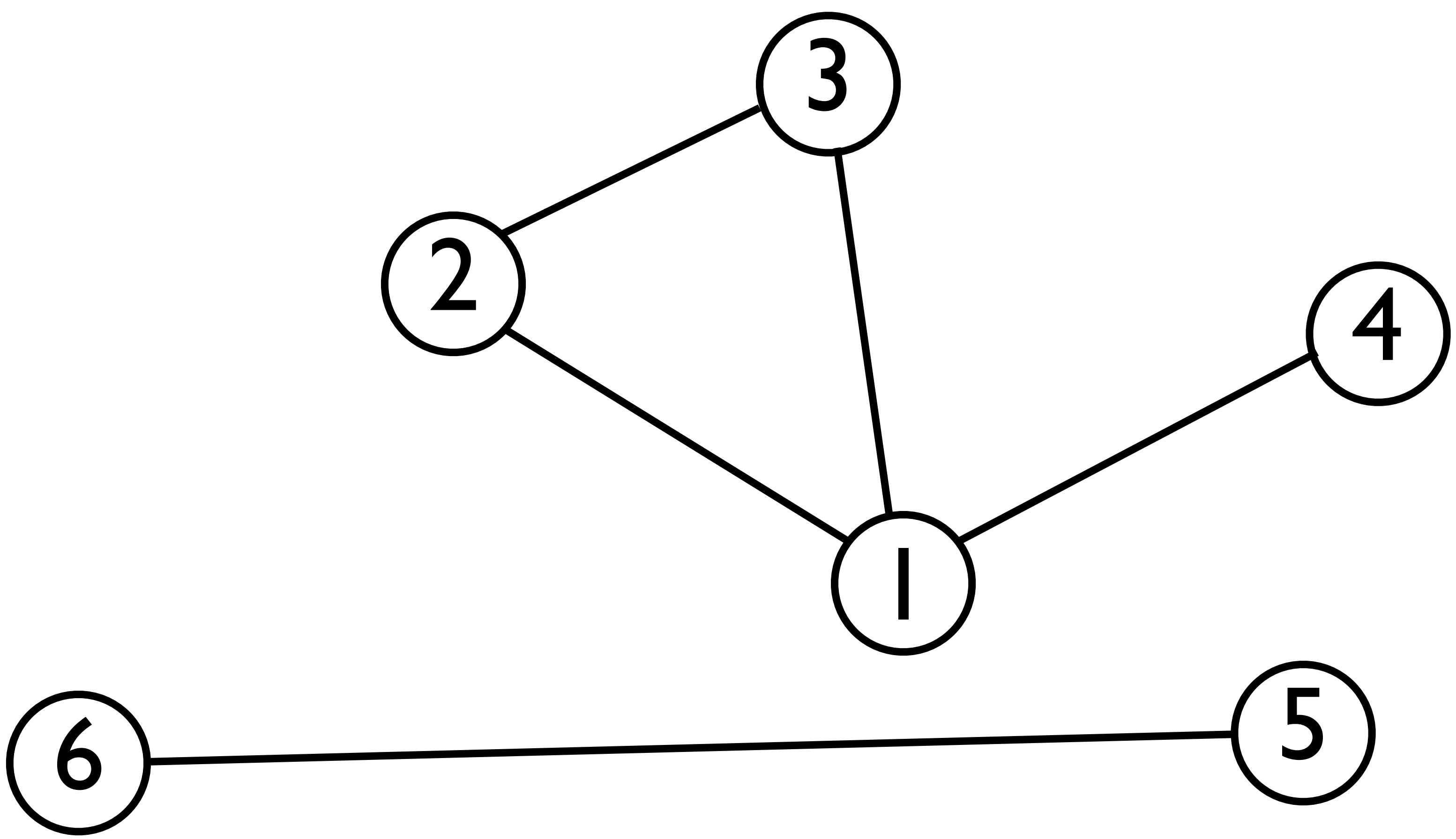
Graphs

- E.g., each node might represent a *computer server*, and each edge represents whether two nodes are *linked* by Ethernet.



Graphs

- Like *trees*, graphs consist of *nodes* and *edges*.
- Unlike trees, graph can contain *cycles*.
- Graphs can be either **undirected** (as below)...



Graphs

- ...or **directed** (as below).
- *Directed graphs* are useful for describing *asymmetric* relationships, e.g., "I know who Rick Santorum is, but he doesn't know who I am."