

# Programmazione Avanzata e parallela

## Lezione 08

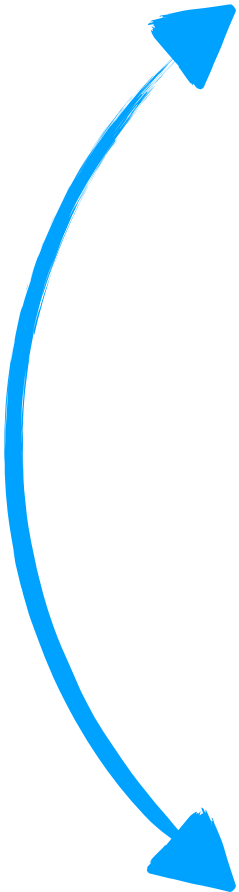
# Gerarchia di memoria

## Cache e memoria principale

- Effetti della latenza per l'accesso alla memoria
- Cache e gerarchia di cache
- Linee di cache
- Associatività delle cache

# Alcuni numeri

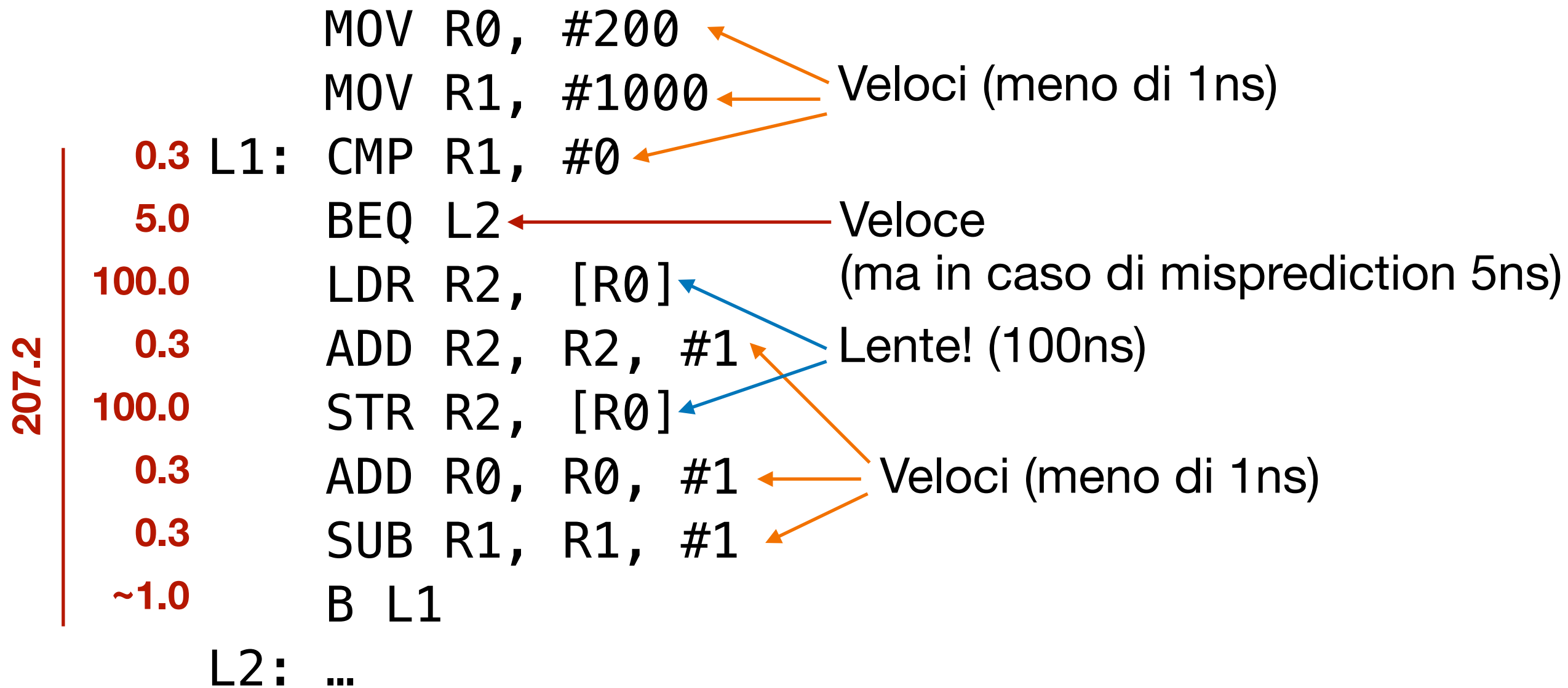
## E perché servono le cache



Tempo	Operazione	Note
0.3ns	ADD, OR, SUB, ...	
0.5ns	Cache L1 dati	
1ns	La luce percorre ~30cm	Ci si aspetta rimanga costante
5ns	Branch misprediction penalty	
5-7ns	Cache L2	~2020
10ns	DIV	
19ns	Cache L3	
100ns	Mutex Lock/Unlock	
100ns	Accesso alla memoria principale	~2020. Sono 200-300 cicli

# Un semplice codice

## E perché servono le cache



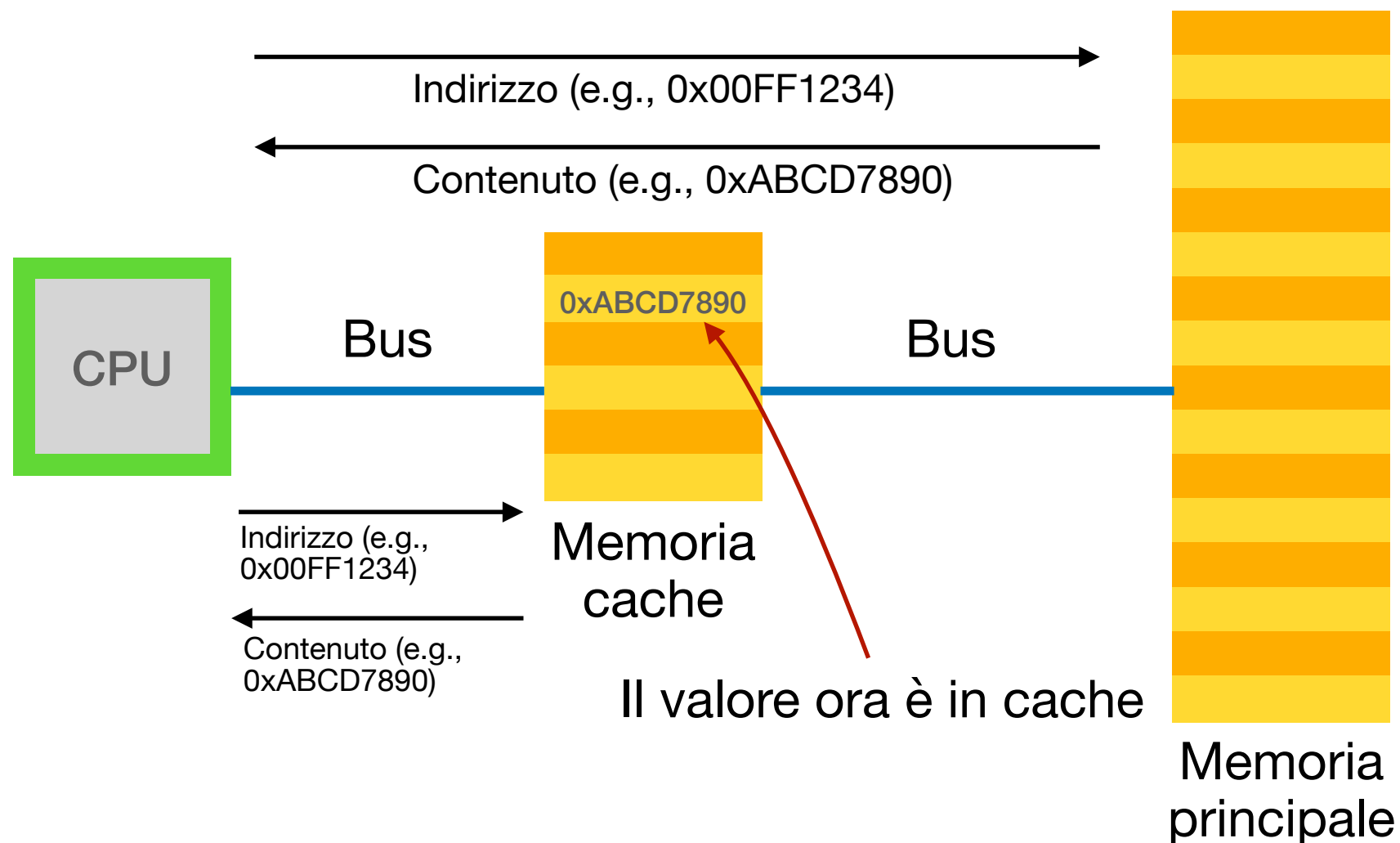
# Effetto degli accessi lenti

## Cache e memoria principale

- Se gli accessi fossero solo a registri lo stesso codice sarebbe oltre 20 volte più veloce
- Possiamo tenere i valori che dobbiamo usare spesso nei registri
- Purtroppo i registri sono pochi
- Rendere la memoria più veloce non è fattibile (sia tecnicamente che in termini di costo)
- Possiamo mettere della memoria più rapida (ma più piccola) per mantenere i dati più recenti a cui abbiamo acceduto

# Aggiunta di una cache

## E gli effetti



## Effetti

Se la cache è più rapida della memoria principale  
ogni accesso ad un indirizzo appena visitato sarà più rapido

# Principi della cache

## E perché funzionano

- Ci aspettiamo che le cache funzionino se sono soddisfatti due principi
- Località temporale. Un indirizzo a cui abbiamo avuto accesso di recente verrà acceduto nuovamente a breve.
- Località spaziale. Se abbiamo acceduto a un indirizzo  $x$  di recente ci aspettiamo di accedere anche a  $x + 1$ ,  $x - 1$ , etc. (i.e., gli indirizzi vicini)
- Se questi due aspetti sono rispettati le cache sono efficaci nel rendere l'esecuzione più veloce

# Di cosa fare cache

## Dati e istruzioni

- Aggiungere una cache ci permette di rendere più rapidi gli accessi successivi (e.g., quelli provati da LOAD e STORE)
- Abbiamo ignorato che leggere l'istruzione successiva richiede anche quello un accesso alla memoria (all'indirizzo indicato dal program counter)
- Ha senso avere anche una cache per le istruzioni, non solo per i dati!
- Nel caso di un loop potenzialmente una volta eseguito una volta tutte le istruzioni saranno già nella cache



# Dimensioni di una cache

## Tradeoffs

- Solitamente più è grande la cache e più è lenta
- Per questo solitamente vi sono più livelli di cache con diverse tradeoff tra dimensioni e prestazioni:
- **L1dati e L1istruzioni**  
Due cache separate (una per i dati e una per le istruzioni)  
dimensioni in decine di KB. e.g., 32KB ciascuna in Zen 4
- **L2**  
Non distingue tra dati e istruzioni, specifica per il singolo core  
Dimensioni fino a MB. e.g., 1MB per zen4
- **L3**  
Più grande e più lenta di L2, spesso condivisa tra più core.  
Dimensioni di decine di MB. e.g., 32MB per zen4

# Come la cache accede ai dati

**Hint: non un byte alla volta**

- La cache non carica mai un solo byte alla volta
- Un cache carica una intera **cache line**
- In molti casi questa cache line ha dimensione di 64bytes (qui quindi contenere, per esempio, 16 interi di 32 bit)
- Una volta fatto un accesso, l'accesso agli altri indirizzi nella stessa cache line sarà “gratis”, sono stati già stati caricati in cache
- Questo si basa sul principio di località spaziale

# Associatività

## Dove la cache mette i dati

- Una cache deve poter associare un indirizzo a una specifica linea di cache
- Nel caso un indirizzo possa essere in un solo “slot” la cache è detta “**direct-mapped**” o “**1-way associative**”
- Se si leggono due indirizzi che condividono lo stesso “slot” la vecchia linea di cache viene rimossa e inserita quella nuova
- Rapida da implementare ma poco efficiente in termini di “hit rate” (quale percentuale degli indirizzi richiesti sono in cache)

# Associatività

## Dove la cache mette i dati

- Opposto al caso “direct mapped” vi è quello “**fully associative**”
- In questo caso ogni linea di cache può essere inserita in ogni “slot”
- Questo aumenta l’“hit rate” ma rende la cache più difficile da costruire e rendere veloce
- Serve selezionare una politica di rimpiazzo (replacement policy): se tutti le posizioni sono occupate quale viene rimossa?
- Una politica comune è LRU (least recently used): la linea di cache che è stata acceduta più indietro nel tempo

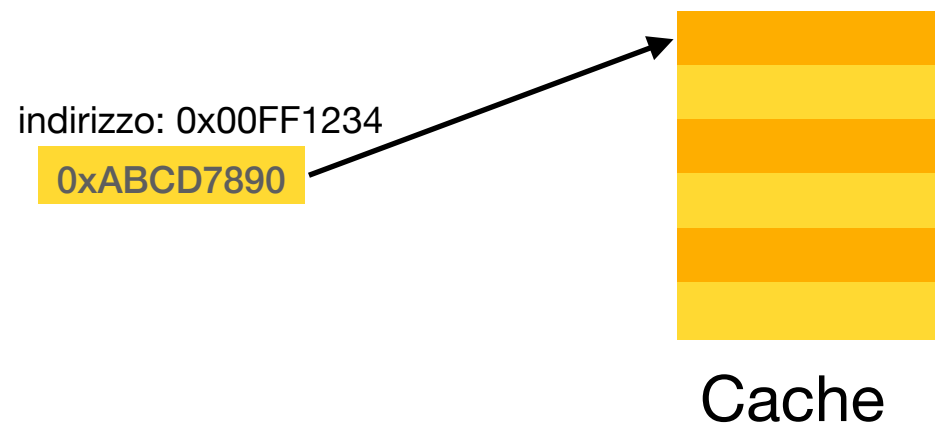
# Associatività

## Dove la cache mette i dati

- Un compromesso è una cache  $k$ -way associative, in cui ogni indirizzo può essere associato a  $k$  slot
- Rende più facile l'implementazione mantenendo alcuni vantaggi delle cache fully associative
- Vi è un tradeoff tra velocità e prestazioni. Alcuni esempi, in Zen 4 L1 e L2 sono 8-way associative e L3 è 16-way associative

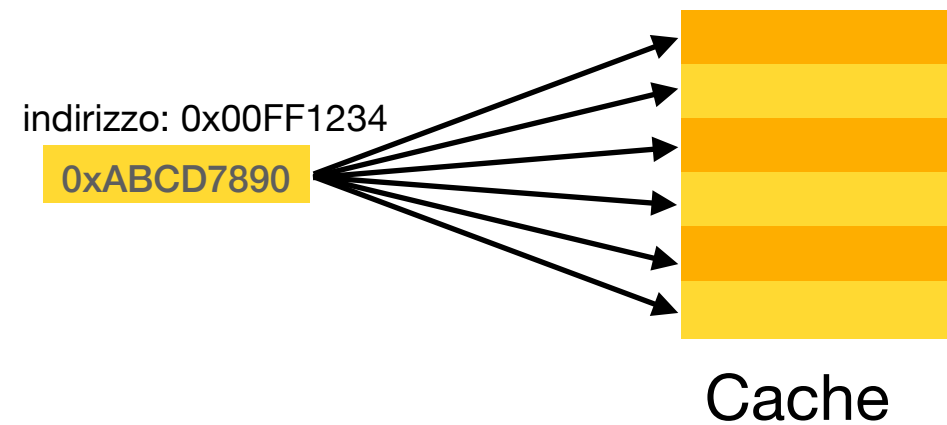
# Associatività

## E gli effetti



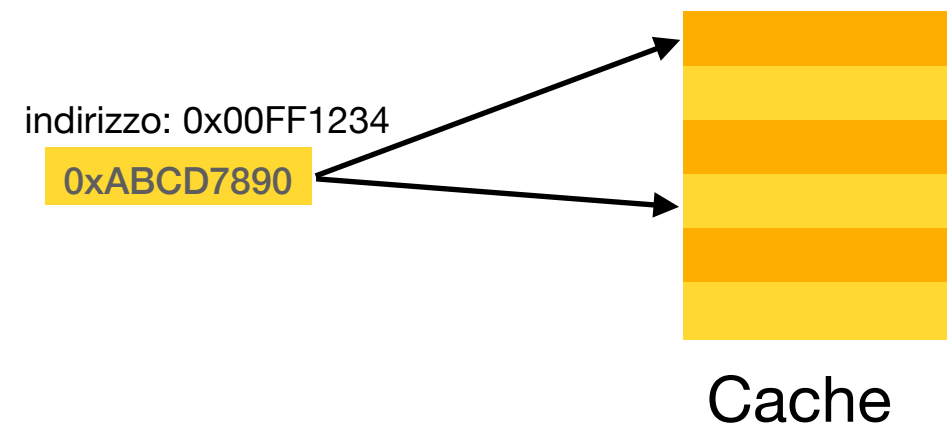
### Direct mapped

L'indirizzo determina in modo univoco dove può essere inserita la linea di cache



### Fully associative

La linea di cache può essere inserita in ogni posizione



### 2-way associative

La linea di cache può essere inserita in 2 posizioni (dipendono dall'indirizzo)

# Disposizione in memoria

# Disposizione in memoria

## Come usare meglio le cache

- Disposizione in memoria e utilizzo delle cache
- L'effetto dell'allineamento
- Array-of-structures (AoS) vs structure-of-arrays (SoA)
- Cache e ricerca binaria (parte 1)



# Liste concatenate

## Effetto sulla cache

```
struct node {  
    int key;  
    struct node * next;  
}
```



Padding per far iniziare  
l'indirizzo a un multiplo di 8 bytes

Rappresentazione in memoria  
(interi di 32 bit, puntatori di 64 bit)

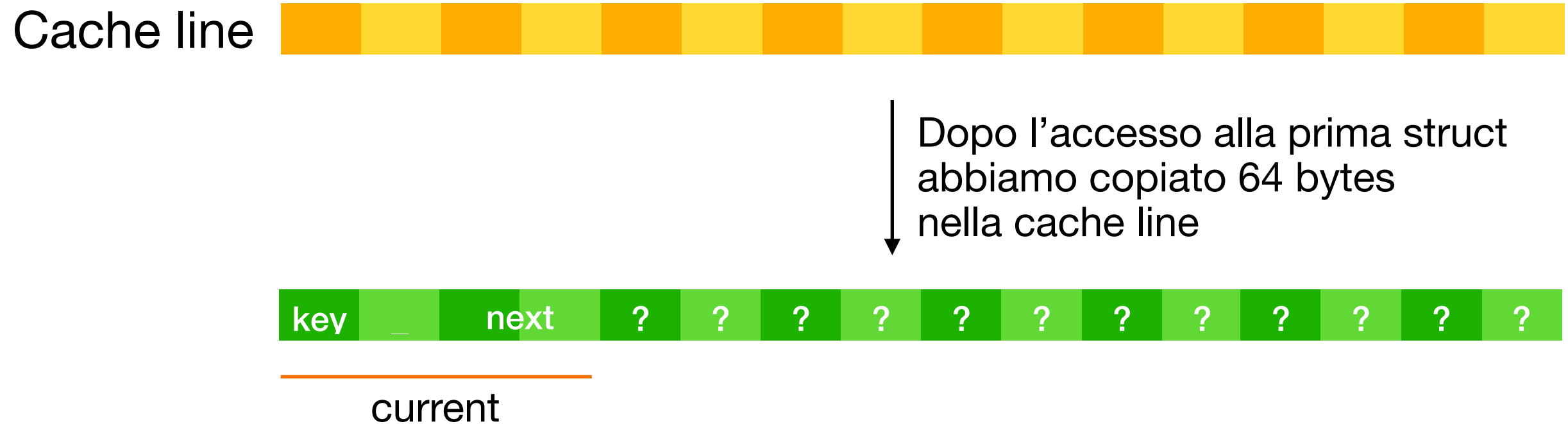
```
int sum = 0;  
struct node * current = head;  
while (current != NULL)  
{  
    sum += current->key;  
    current = current->next;  
}
```

Sommiamo solo il campo “key”  
di una lista di strutture “foo”

**Quale è l'effetto sulla cache?**

# Liste concatenate

## Effetto sulla cache



- Al primo accesso copiamo il valore nella cache line
- Il puntatore è già nella cache quando ci accediamo
- Però il prossimo nodo potrebbe non essere nella stessa cache line

# Liste concatenate srotolate

## Effetto sulla cache

```
struct node {  
    int key[N];  
    bool valid[N];  
    struct node * next;  
}
```



Rappresentazione in memoria  
(interi di 32 bit, puntatori di 64 bit)

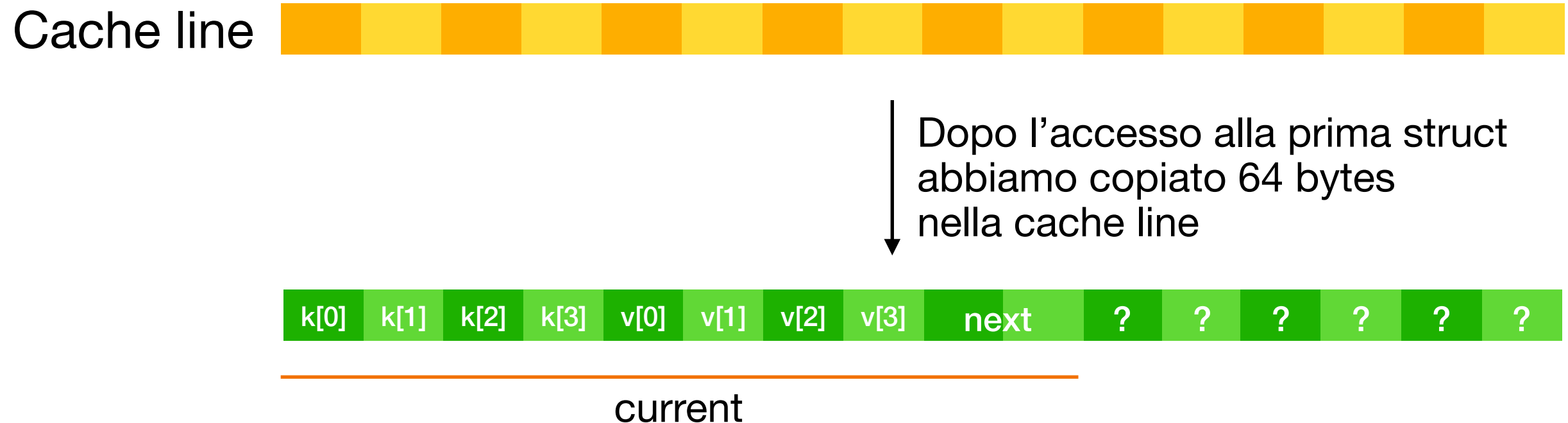
```
int sum = 0;  
struct node * current = head;  
while (current != NULL)  
{  
    for (int i = 0; i < N; I++) {  
        if (current.valid[i])  
            sum += current.key[i];  
    }  
    current = current.next;  
}
```

Questa volta per ogni nodo  
sommiamo un vettore di valori

**Quale è l'effetto sulla cache?**

# Liste concatenate

## Effetto sulla cache



- Al primo accesso copiamo il valore nella cache line
- Il puntatore è già nella cache quando ci accediamo
- Ma anche le chiavi: paghiamo la penalità solo una volta per nodo (assumendo che l'intero nodo stia nella cache line)

# Requisiti di allineamento

# Allineamento

## E memoria “sprecata”

- In molte architetture una struttura può richiedere più bytes della somma dei byte occupati dai suoi membri
- Molte architetture hanno requisiti di allineamento per l'accesso alla memoria:
  - A volte “LOAD” (o istruzioni equivalenti) posso accedere solo a indirizzi multipli di 4 o 8 (bytes)
  - E se anche l'accesso non allineato è consentito solitamente quello allineato è più efficiente
  - Rimane comunque possibile fare accessi non allineati leggendo (in modo allineato) più del necessario

# Allineamento

## E memoria “sprecata”

- Il compilatore allineerà i membri delle strutture in modo che sia possibile accederci in modo allineato
  - e.g., facendo in modo che per leggere un valore da 64bit sia necessario un solo LOAD
- Per standard il compilatore deve rispettare l'ordine dei membri delle strutture
- Queste due strutture avranno una diversa rappresentazione in memoria:
  - `struct foo {int a; int * b; int c};`
  - `struct bar {int * b; int c; int a};`

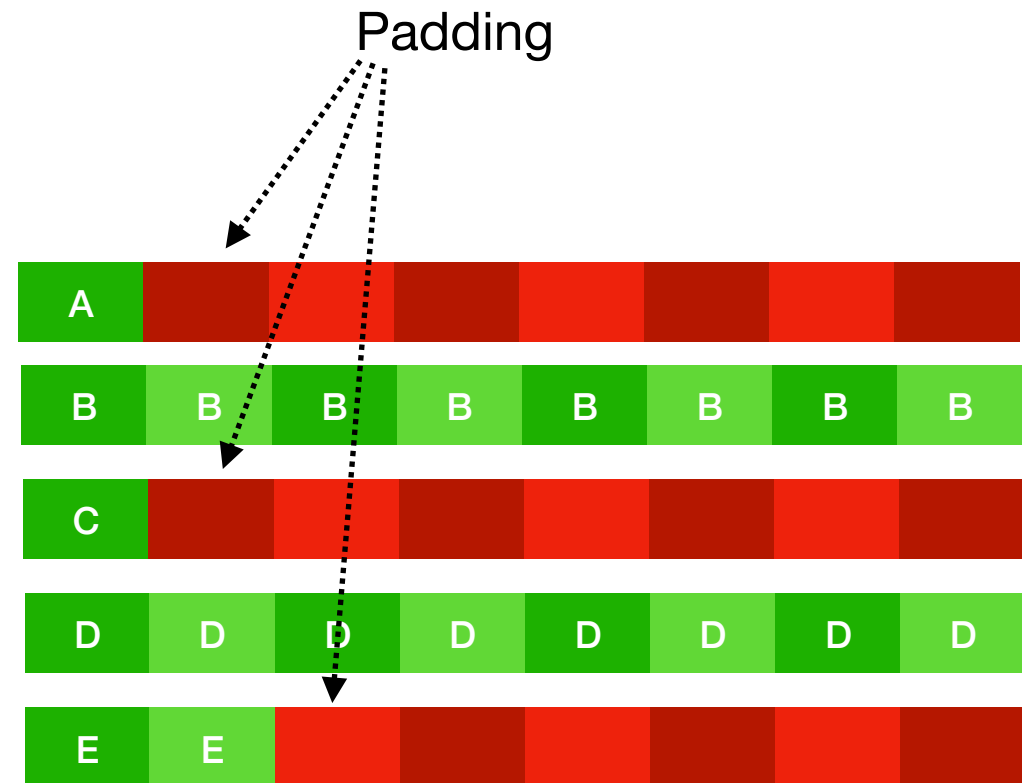
# Allineamento

## Presenza del padding

```
struct foo {  
    int8_t a;  
    int64_t b;  
    int8_t c;  
    int64_t d;  
    int16_t e;  
};
```

Spazio “utile” (somma delle dimensioni dei membri):

$$1 + 8 + 1 + 8 + 2 = 20 \text{ byte}$$



Rappresentazione in memoria  
(ogni blocco è un byte)

Spazio utilizzato: 40 byte



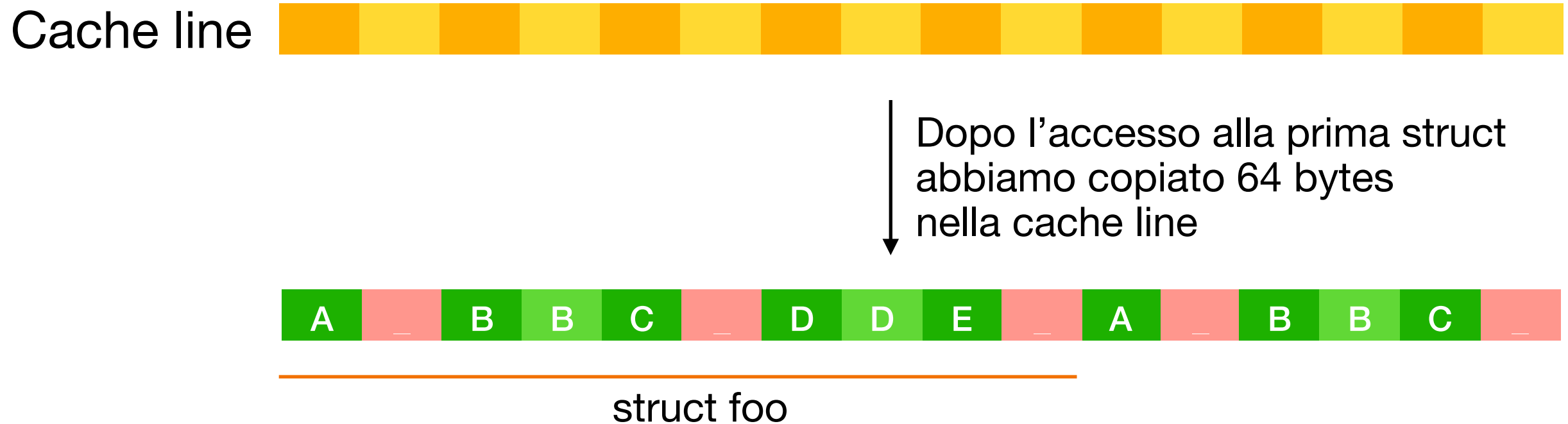
# Allineamento

## Array di struct

- Perché c'è padding anche dopo l'ultimo membro?
- Quando allochiamo un vettore di struct foo facciamo:
  - `malloc(N * sizeof(struct foo))`
- Quindi un array non può “aggiungere padding” tra una struttura e la successiva
- Quindi il compilatore deve aggiungere padding a fine struttura affinché in un array ogni struttura inizi a un indirizzo allineato

# Allineamento

## Effetto sulla cache

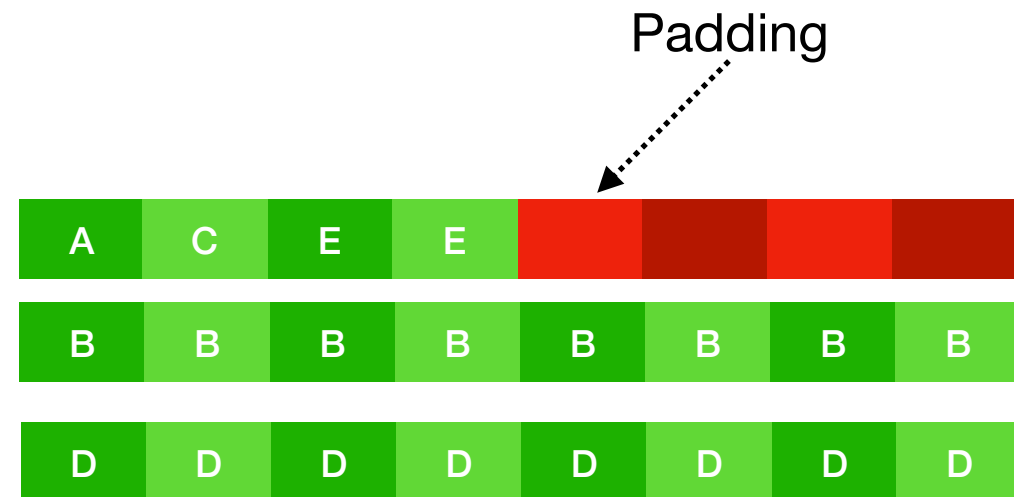


- In una linea di cache riusciamo a tenere meno di due strutture
- Metà dello spazio di una struttura è sprecato da padding che quindi è sicuramente inutilizzato
- Possiamo migliorare l'occupazione della memoria?

# Allineamento

## Riordinare i membri di una struttura

```
struct bar {  
    int8_t a;  
    int8_t c;  
    int16_t e;  
    int64_t b;  
    int64_t d;  
};
```



Rappresentazione in memoria  
(ogni blocco è un byte)

Spazio utilizzato: 24 byte

Spazio “utile” (somma delle  
dimensioni dei membri):

$$1 + 8 + 1 + 8 + 2 = 20 \text{ byte}$$

# Allineamento

## Effetto sulla cache

Cache line



Dopo l'accesso alla prima struct  
abbiamo copiato 64 bytes  
nella cache line



struct bar

- In una linea di cache riusciamo a tenere oltre due strutture
- Solo il 12 bytes su 64 sono sprecati da padding (~18%)

# Allineamento

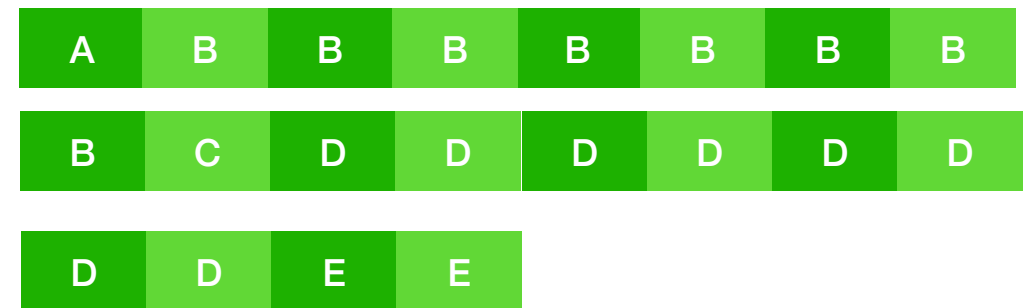
## Forzare l'assenza di padding

- E se volessimo minimizzare l'occupazione di memoria anche a scapito delle prestazioni?
  - Potremmo avere vincoli di posizionamento in memoria dei diversi membri
  - O verificare che il tempo aggiuntivo speso per accedere ai membri è bilanciato dalla minore occupazione di memoria
- Possiamo forzare l'assenza di padding tramite metodi non standard (supportati da gcc e clang)
- Da **non** fare nella maggior parte dei casi

# Allineamento

## Eliminare il padding

```
struct __attribute__((packed)) baz {  
    int8_t a;  
    int64_t b;  
    int8_t c;  
    int64_t d;  
    int16_t e;  
};
```



Rappresentazione in memoria  
(ogni blocco è un byte)

Spazio utilizzato: 20 byte

Spazio “utile” (somma delle  
dimensioni dei membri):

$$1 + 8 + 1 + 8 + 2 = 20 \text{ byte}$$

**Attenzione:** l'accesso a diversi dei membri sarà meno efficiente!

# Array di strutture e strutture di array

# Struct e cache lines

## Un array di struct

```
struct foo {  
    int a;  
    int b;  
    int c;  
    int d;  
}
```



Rappresentazione in memoria  
(interi di 32 bit)

```
int sum = 0;  
for (int i = 0; i < N; i++)  
{  
    sum += v[i].a;  
}
```

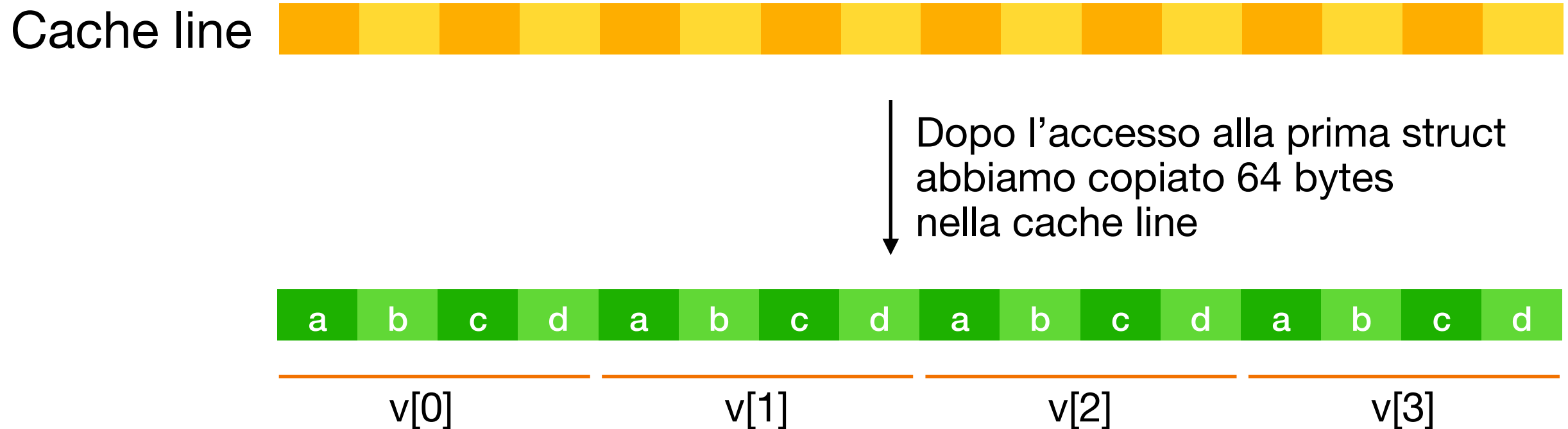
Sommiamo solo il campo “a”  
di un vettore di strutture “foo”

**Quale è l'effetto sulla cache?**



# Struct e cache lines

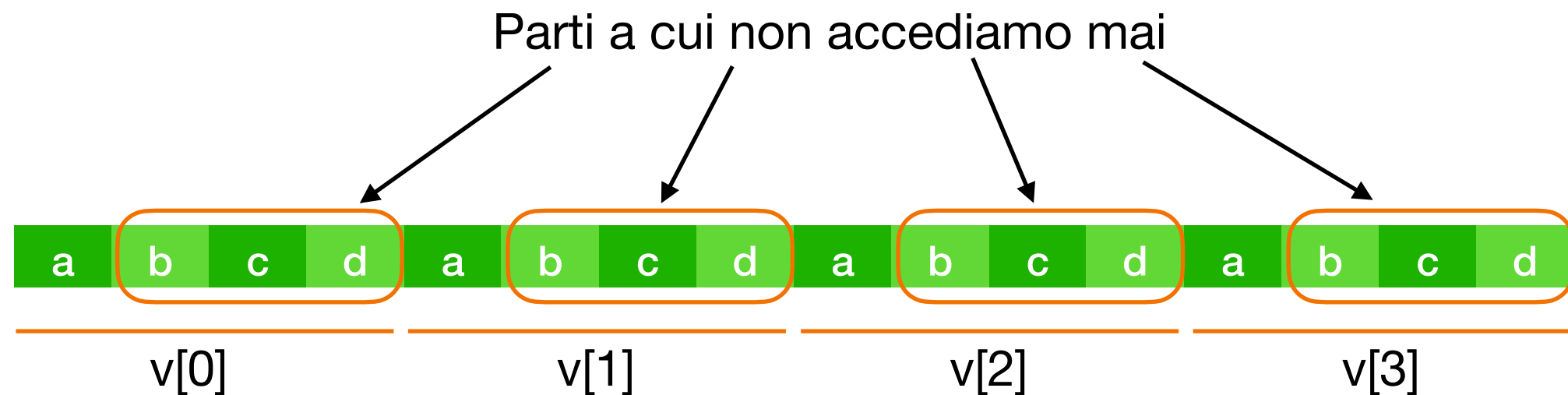
## Un array di struct



- Al primo accesso copiamo il valore nella cache line
- Per  $v[1]$ ,  $v[2]$  e  $v[3]$  il valore è già in cache e non dobbiamo accedere di nuovo alla memoria
- Stiamo usando in modo efficiente la cache?

# Struct e cache lines

## Un array di struct



- Su 64 bytes di cache line noi ne usiamo 16 (4 bytes per intero)
- Questo implica che “sprechiamo” tre quarti di una linea di cache
- Possiamo fare di meglio?

# Array di strutture

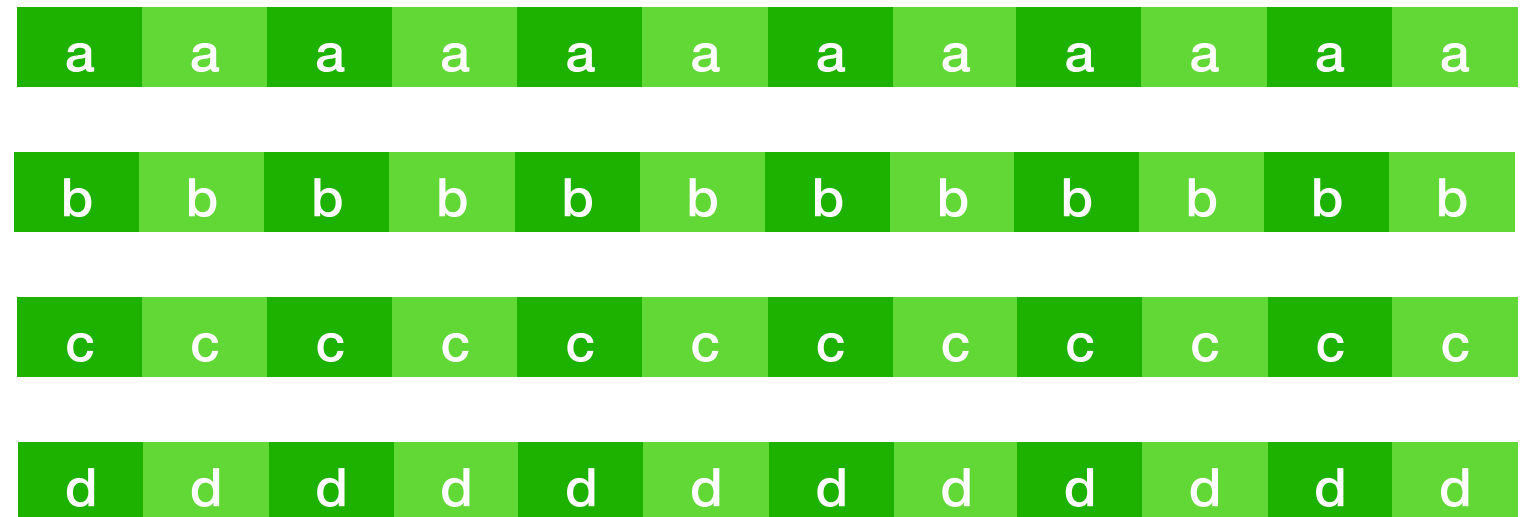
## Ed effetto sulla memoria

- Un pattern comune è avere un insieme di strutture (solitamente in un array) e dover fare una operazione comune a uno dei membri
- In questo caso può essere conveniente sostituire un array di struct (*array of structures* - AoS) a una struttura di array (*structure of arrays* - SoA)
- Invece di avere  $N$  strutture con  $k$  membri ciascuna...
- ...abbiamo una struttura con  $k$  array di lunghezza  $N$ , uno per membro della struttura originaria

# Struct e cache lines

## Una struct di array

```
struct foo_arrays {  
    int * a;  
    int * b;  
    int * c;  
    int * d;  
}
```



Rappresentazione in memoria

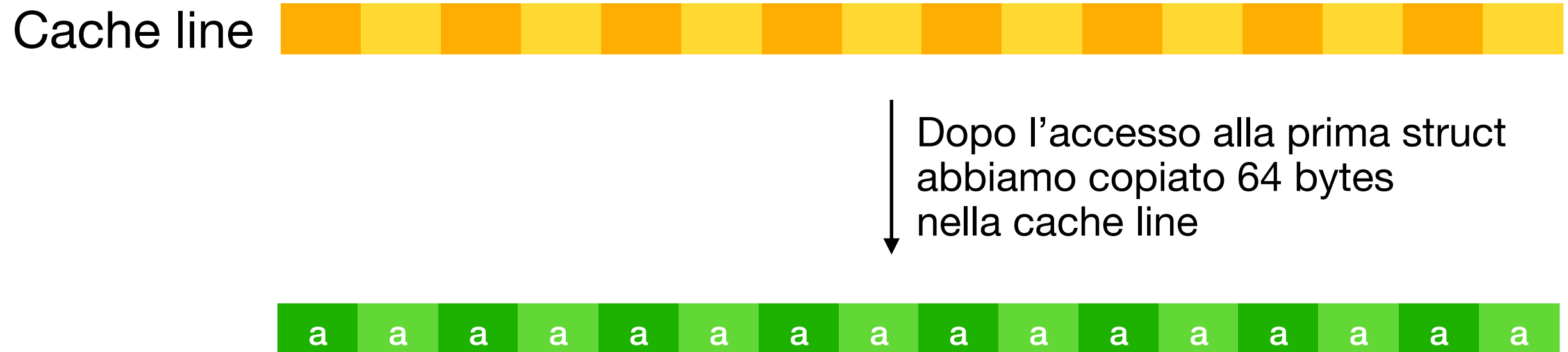
```
int sum = 0;  
for (int i = 0; i < N; i++)  
{  
    sum += x.a[i];  
}
```

Sommiamo solo gli elementi di uno degli array

**Quale è l'effetto sulla cache?**

# Struct e cache lines

## Una struct di array



- Al primo accesso copiamo il valore nella cache line
- I successivi 15 valori saranno già in cache
- Stiamo usando la cache in modo più efficiente (ogni byte è usato)

# AoS vs SoA

## Quando sono convenienti

- Se iteriamo spesso su tutta la collezione accedendo a un singolo membro della struttura allora SoA è più adatto
- Un altro vantaggio di SoA è che iterando su un vettore di int/float (e non strutture) è più facile generare istruzioni vettoriali che operano su più elementi alla volta
- Se accediamo a più membri di una sola struttura alla volta allora AoS è meglio: tutti i membri saranno spazialmente vicini in memoria
- In generale SoA genera codice meno leggibile...
- ...quindi come ogni ottimizzazione usatela quando serve (e.g., non su un vettore di 3 strutture)

# Rivisitare la ricerca binaria

# Ricerca binaria

## Disposizione in memoria

- Rivisitiamo la ricerca binaria vedendo l'effetto sulla cache
- Caso interessante perché ha due pattern differenti:
  - Indirizzi di memoria con buona località spaziale
  - Indirizzi di memoria con buona località temporale
- Possiamo trovare una disposizione in memoria più efficiente?



# Ricerca binaria

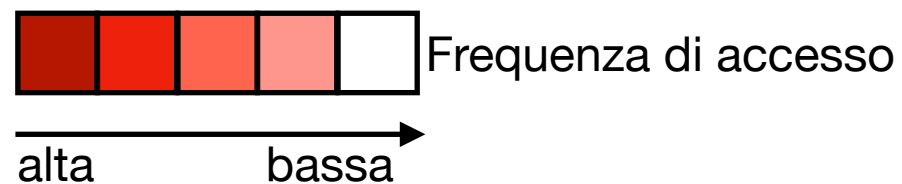
## Pattern di accesso

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

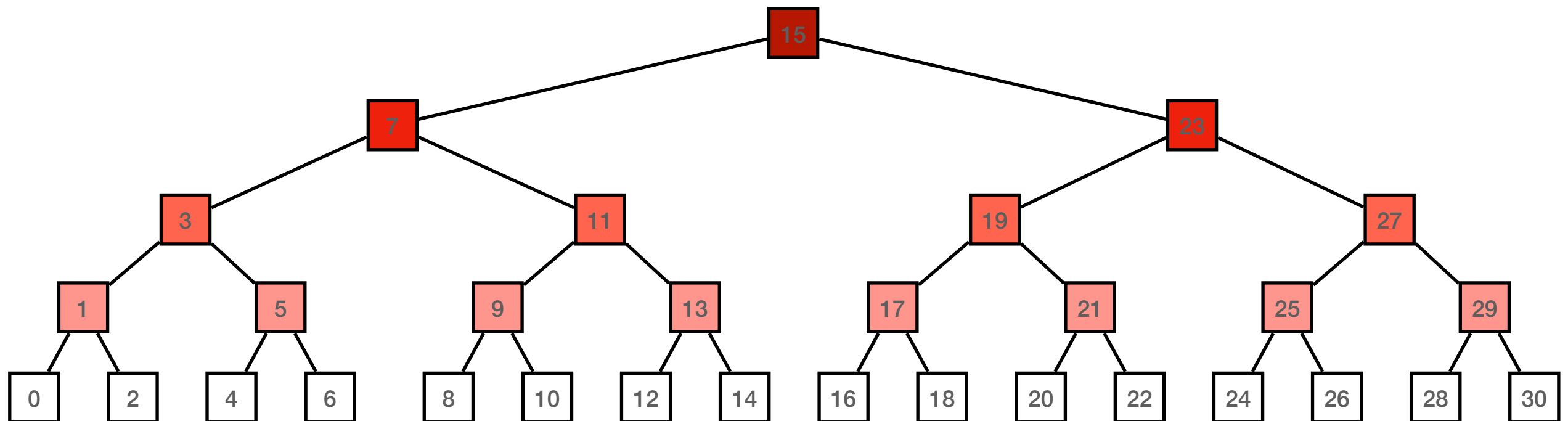
- Consideriamo due fattori che influenzano la presenza di valori nella cache quando effettuiamo una ricerca binaria su un array
- **Località spaziale:** quando accediamo a valori vicini in memoria?
- **Località temporale:** ci sono valori a cui accediamo spesso?

# Ricerca binaria

## Pattern di accesso: località temporale

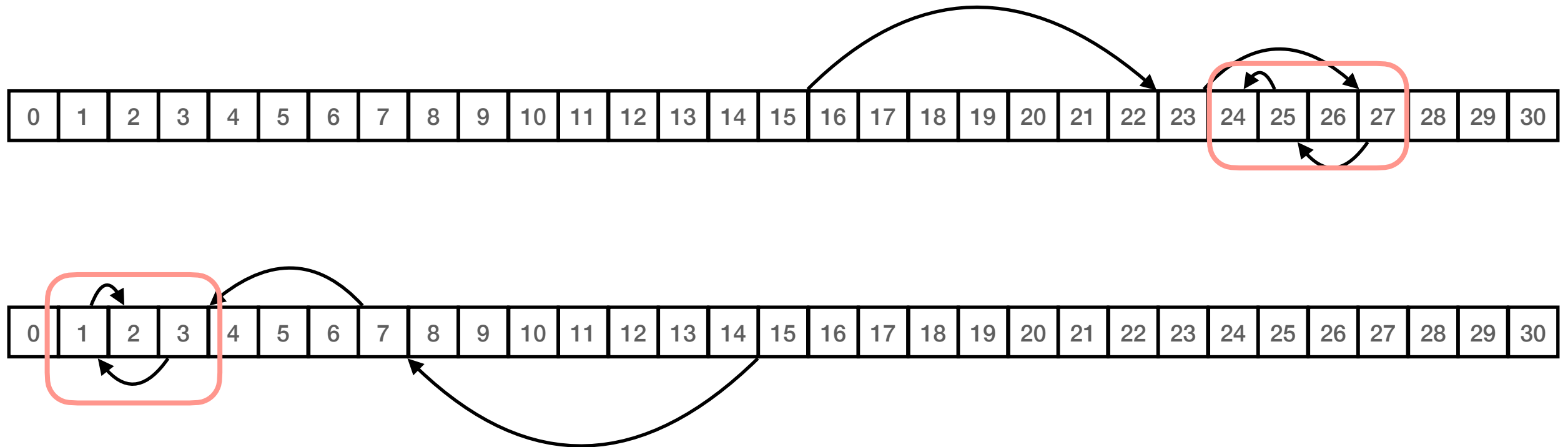


I valori “al centro” sono visti più di frequente perché “ci passiamo” per ogni ricerca  
Questo è più evidente se rappresentiamo la ricerca come un albero:



# Ricerca binaria

## Pattern di accesso: località spaziale



- Notiamo che verso la fine della ricerca gli intervalli da cercare sono piccoli
- Questo garantisce una buona località spaziale, dato che fanno valori caricati nella stessa cache line
- All'inizio della ricerca la località spaziale è invece molto bassa: “saltiamo” a valori molto distanti in memoria che non saranno già caricati in cache

# Ricerca binaria

## Come migliorare

- La località spaziale non è buona all'inizio della ricerca...
- ...ma quella temporale è buona
- Viceversa per la fine della ricerca
- Possiamo trovare una disposizione in memoria che sia più efficiente?
- *Osservazione*: non serve che l'array sia ordinato in ordine crescente, ci va bene anche ogni altra disposizione in cui possiamo sfruttare un ordine tra gli elementi!