

Memory and Data Locality

(The C Memory Model)

<https://cs.pomona.edu/classes/cs140/>

Outline

Topics and Learning Objectives

- Motivate our discussion on memory
- Discuss memory access timing
- Discuss the C memory model
- Discuss locality

Exercise

- Sets and Lists (tangentially related)

Exercise

Code Demos

Process Memory

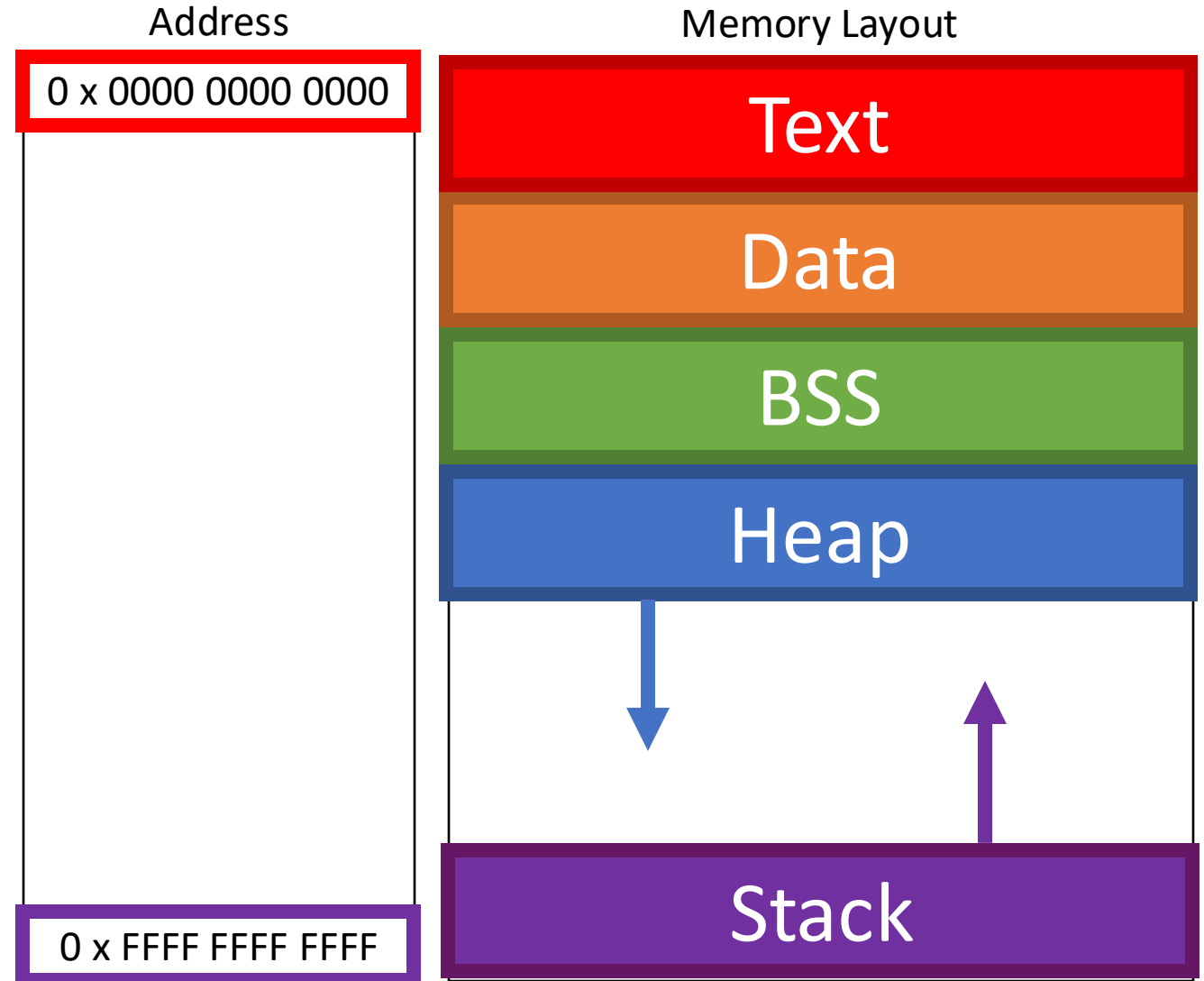
Text: contains your compiled code

Data: contains **initialized** static and global variables

BSS: contains uninitialized static and global variables

Heap: contains dynamically allocated memory

Stack: contains local variables



Process Memory

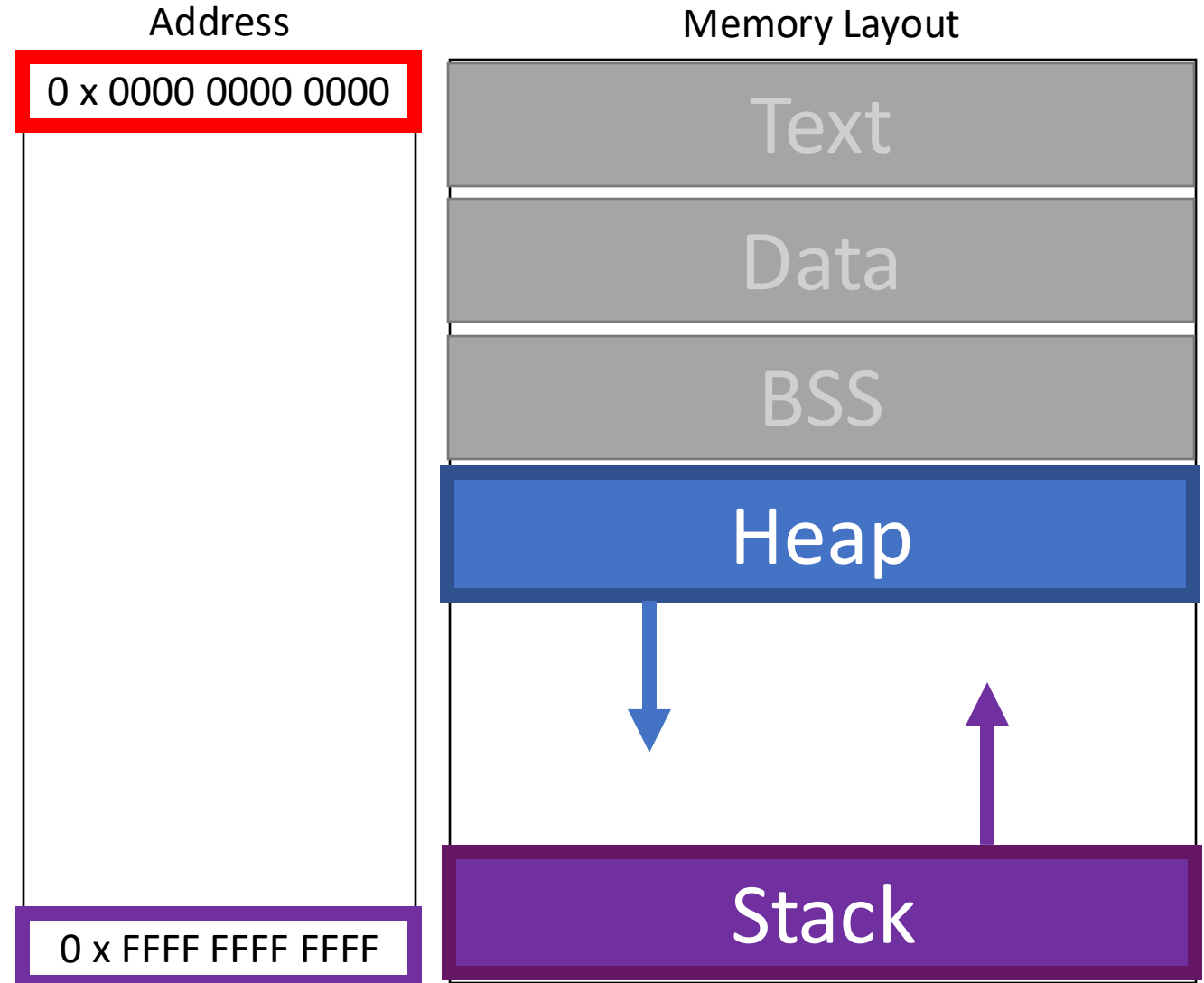
Text: contains your compiled code

Data: contains **initialized** static and global variables

BSS: contains uninitialized static and global variables

Heap: contains dynamically allocated memory

Stack: contains local variables

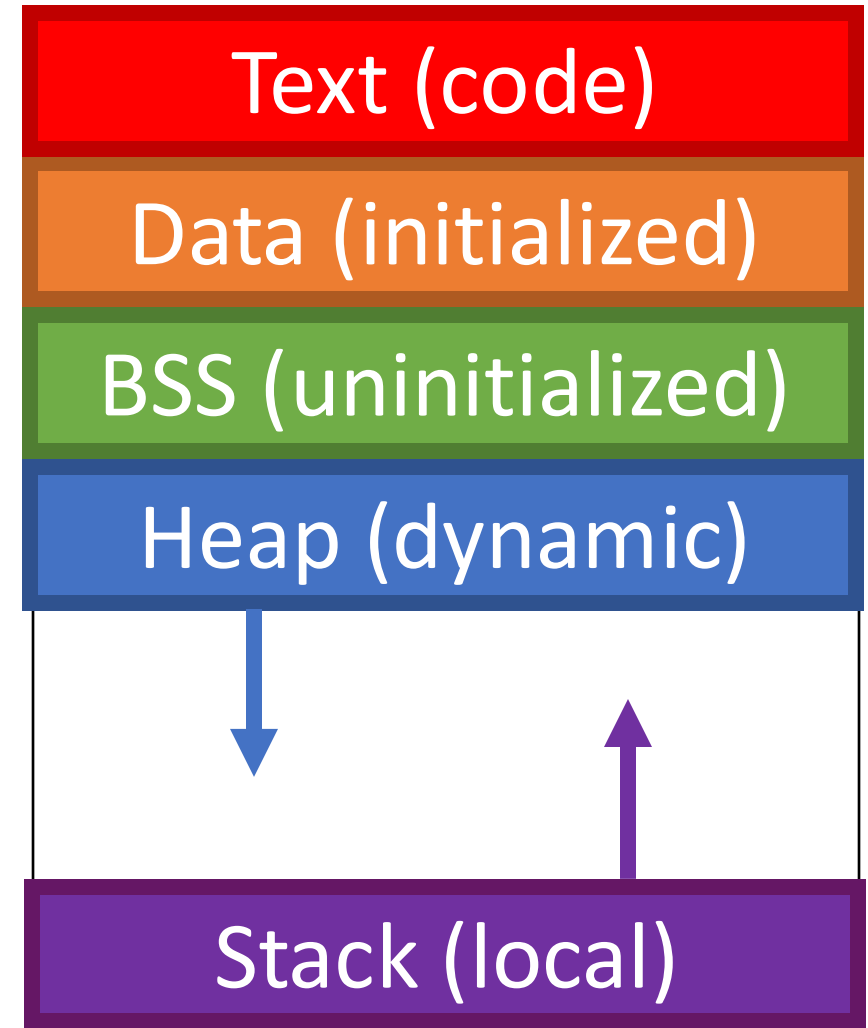


Read through the code for a few moments.

```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```

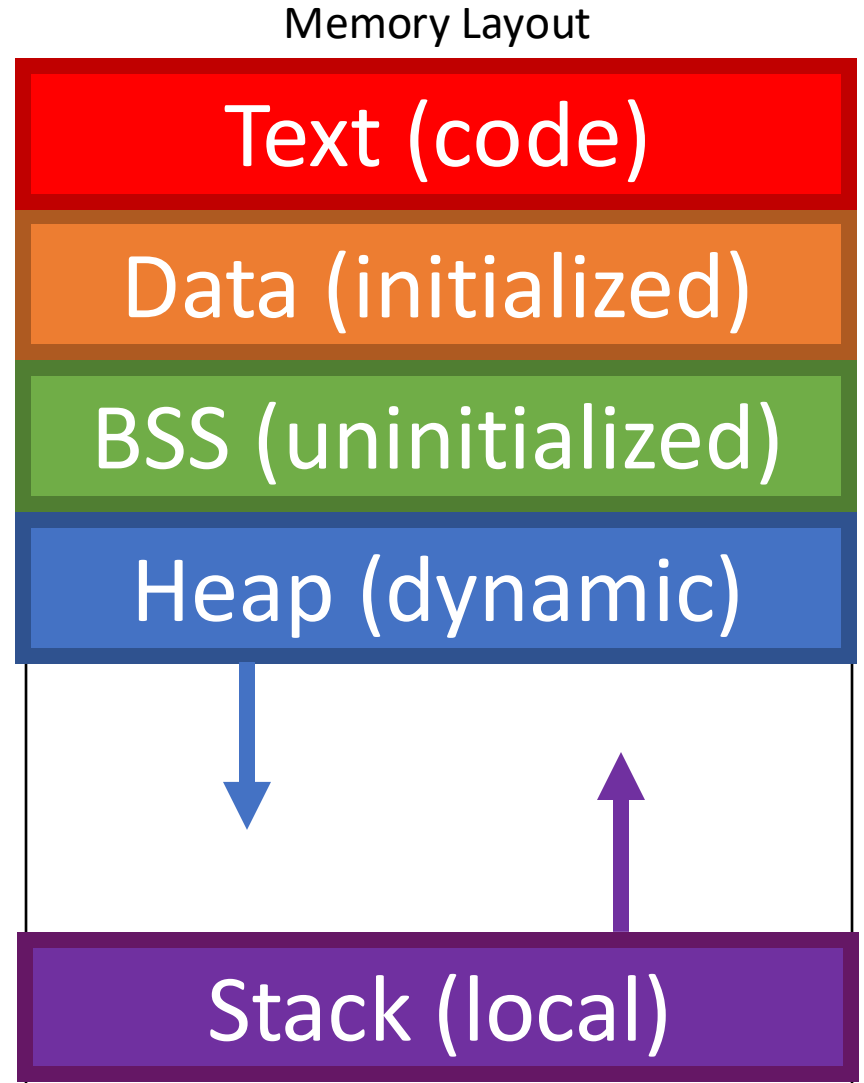
Memory Layout



What goes in **Text**?

```
char* string = "hello";  
int iSize;
```

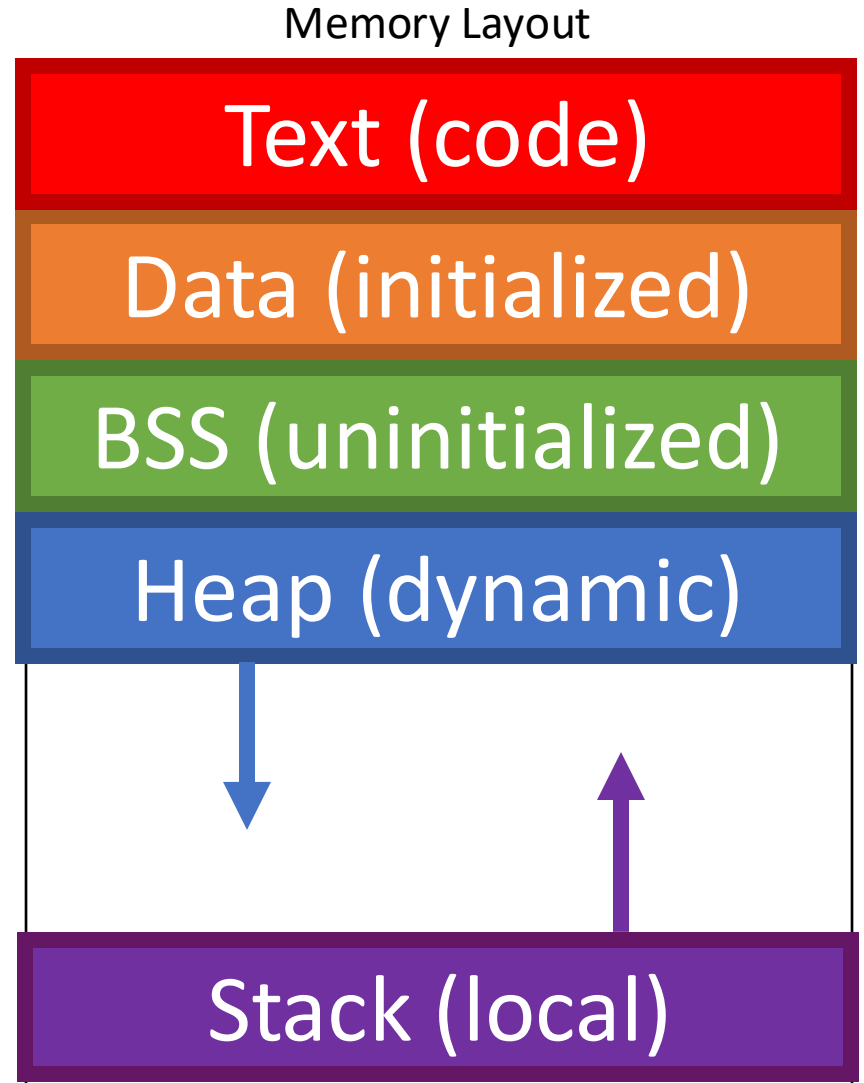
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in **Text**?

```
char* string = "hello";  
int iSize;
```

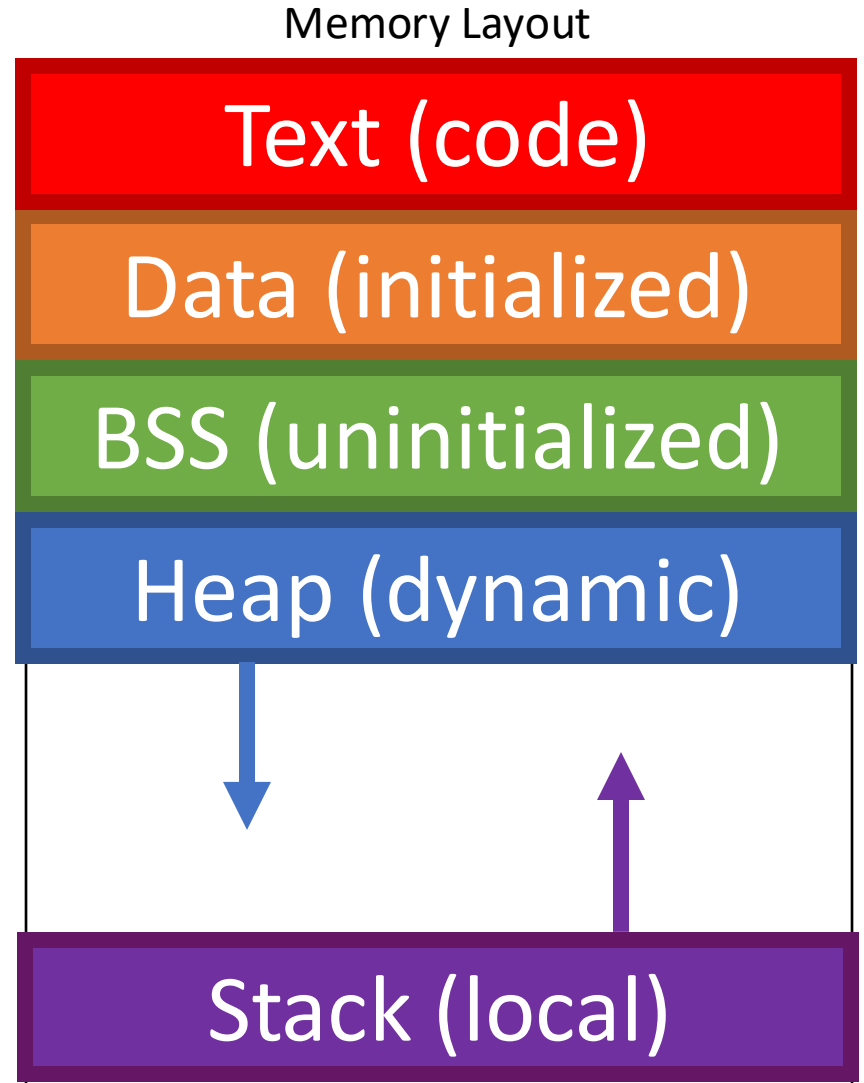
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in Data?

```
char* string = "hello";  
int iSize;
```

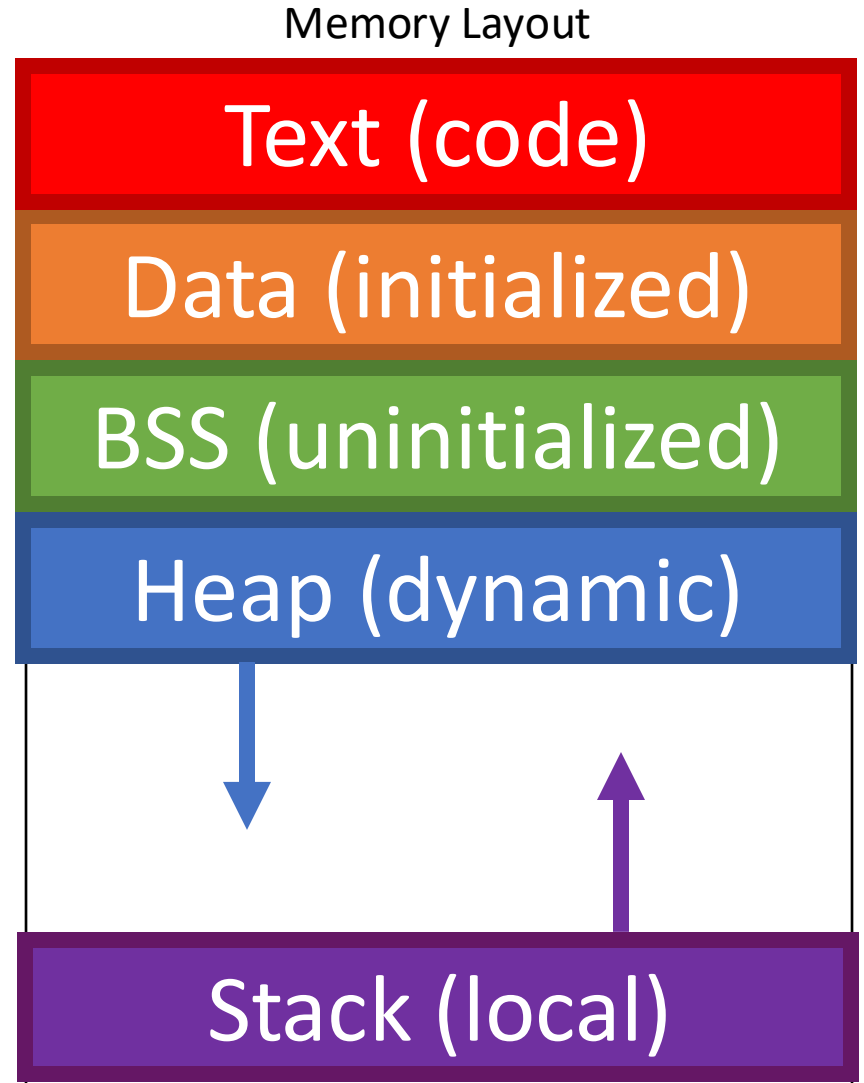
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in Data?

```
char* string = "hello";  
int iSize;
```

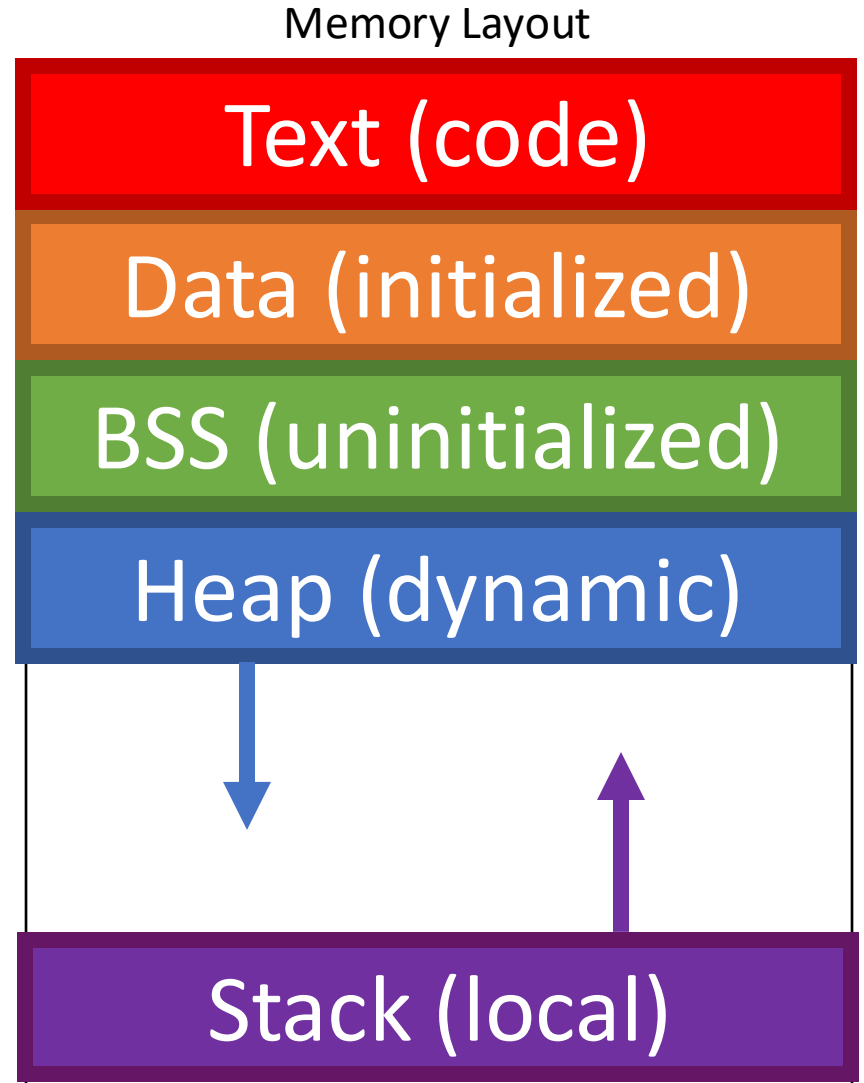
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in BSS?

```
char* string = "hello";  
int iSize;
```

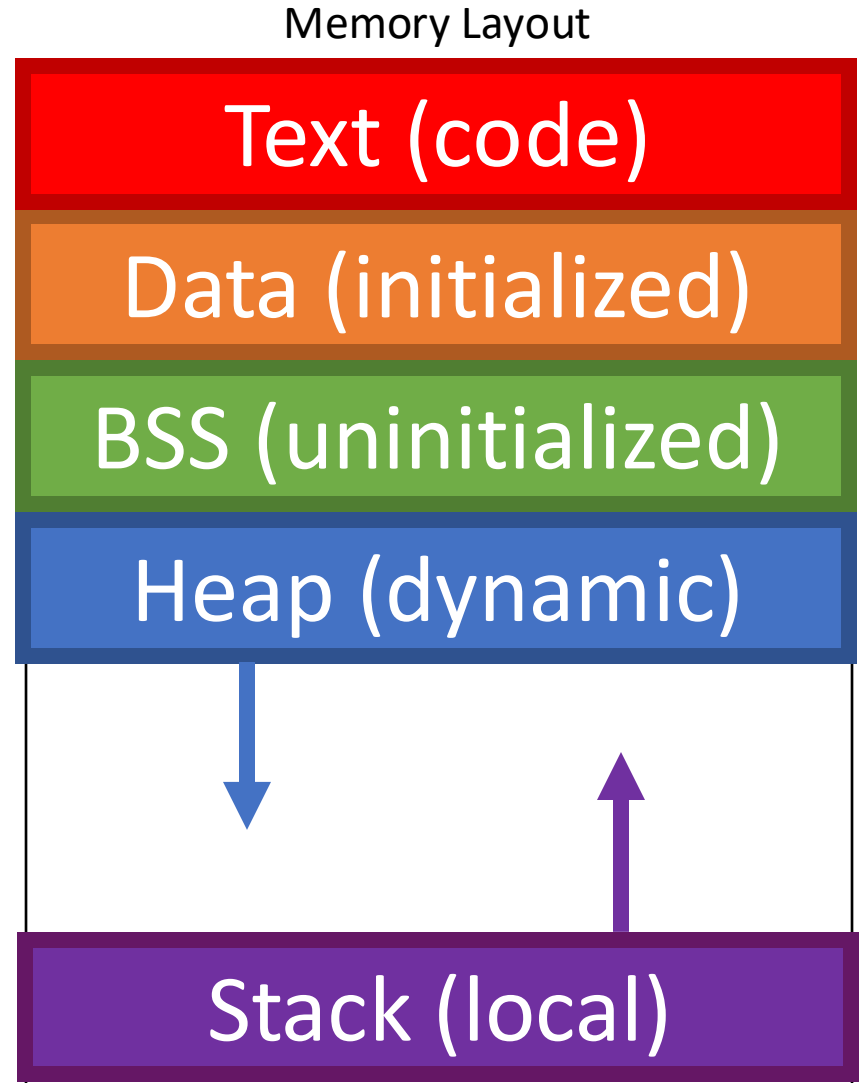
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in BSS?

```
char* string = "hello";  
int iSize;
```

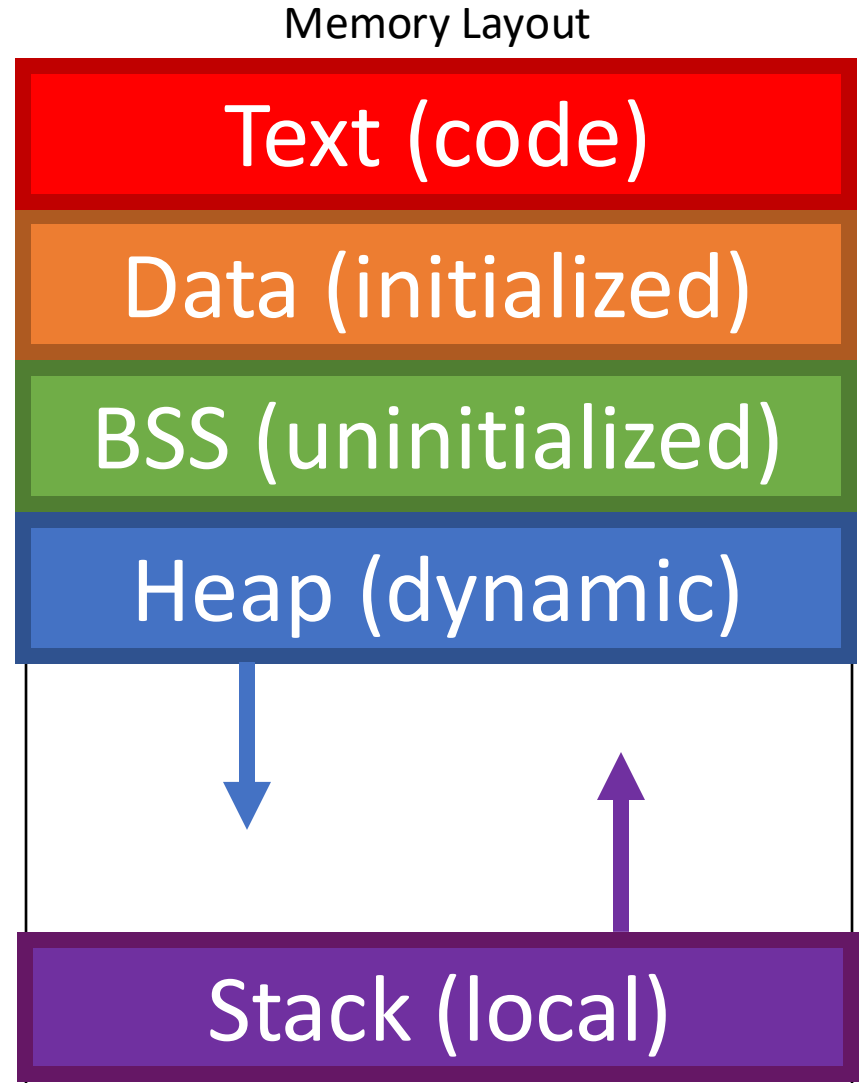
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in **Heap**?

```
char* string = "hello";  
int iSize;
```

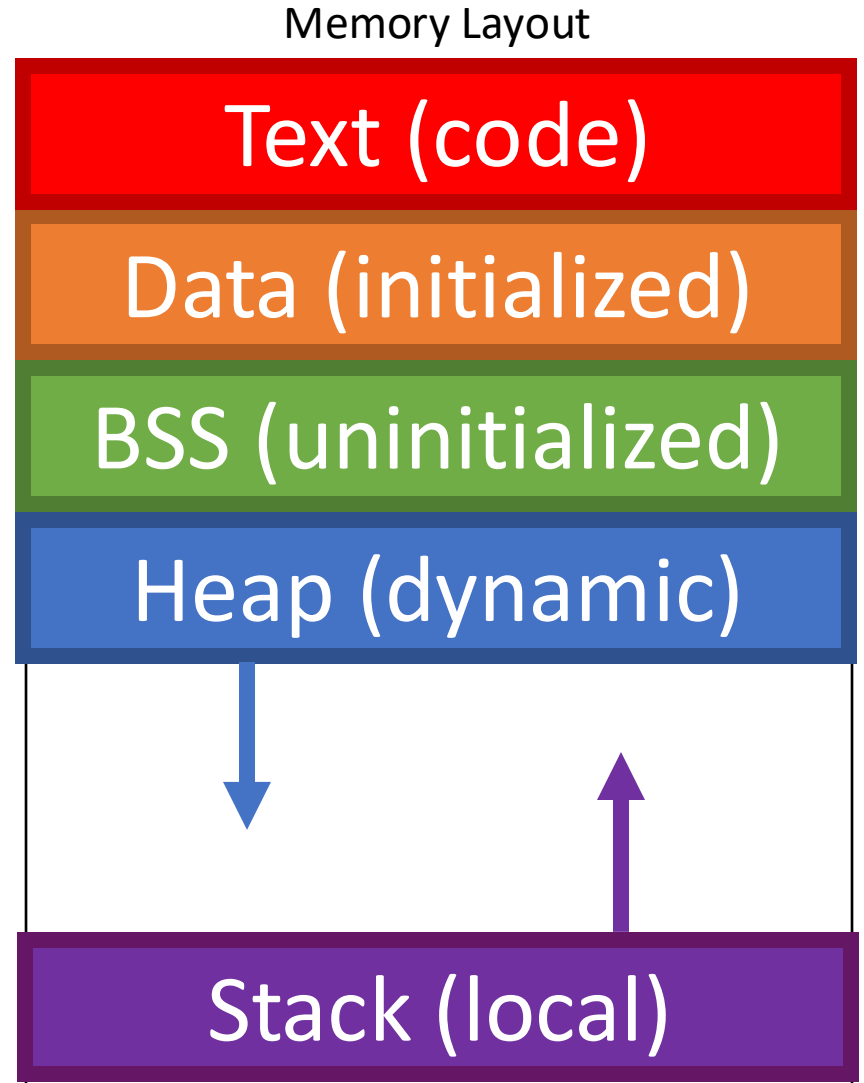
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in **Heap**?

```
char* string = "hello";  
int iSize;
```

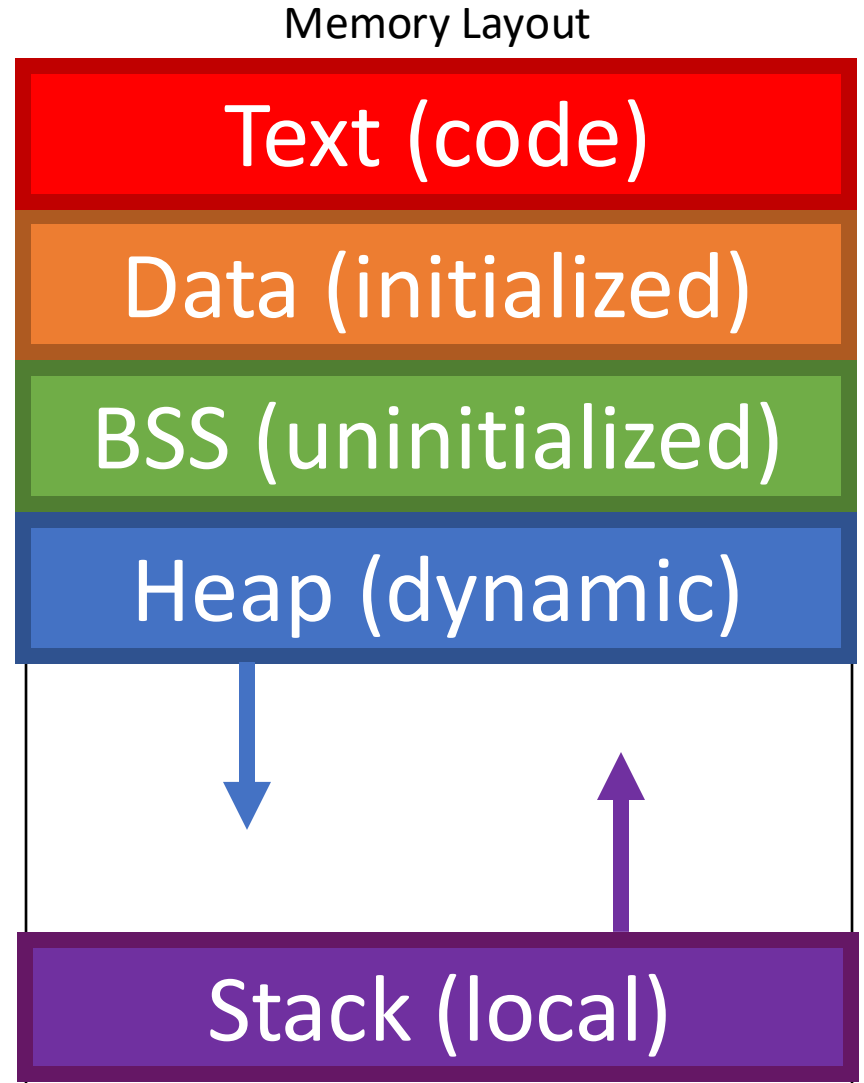
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in **Stack**?

```
char* string = "hello";  
int iSize;
```

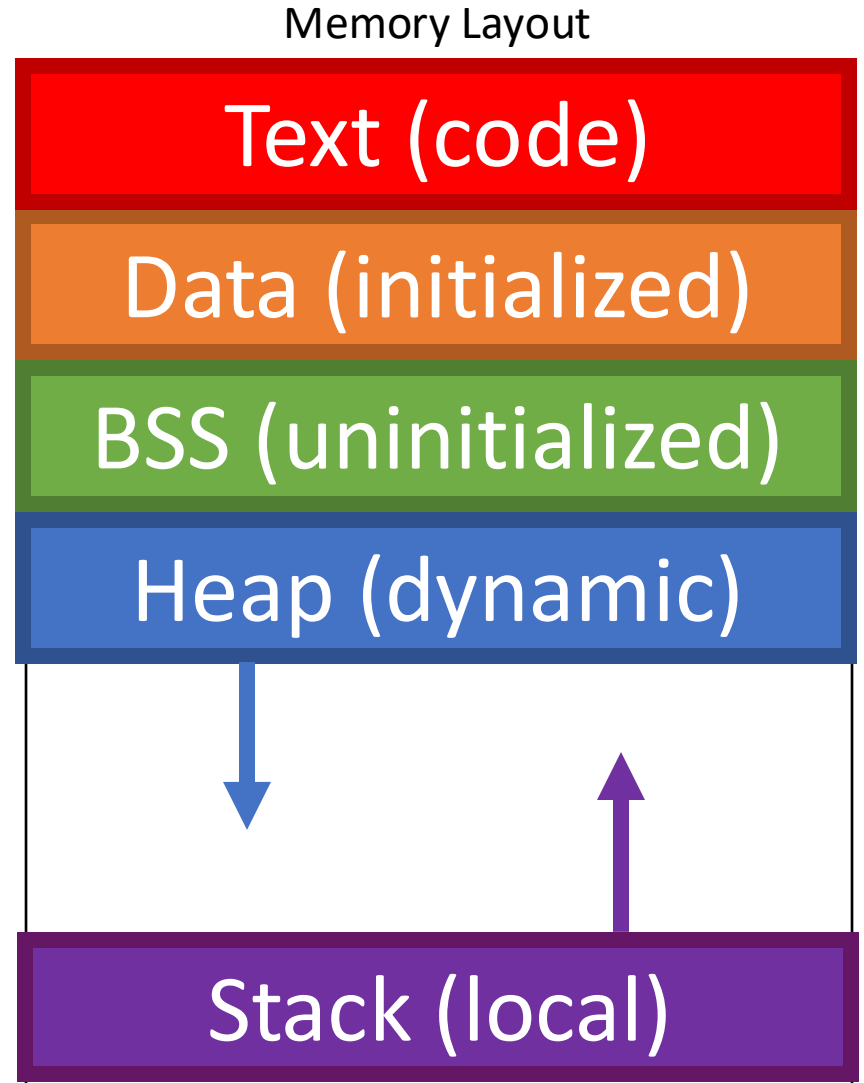
```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



What goes in **Stack**?

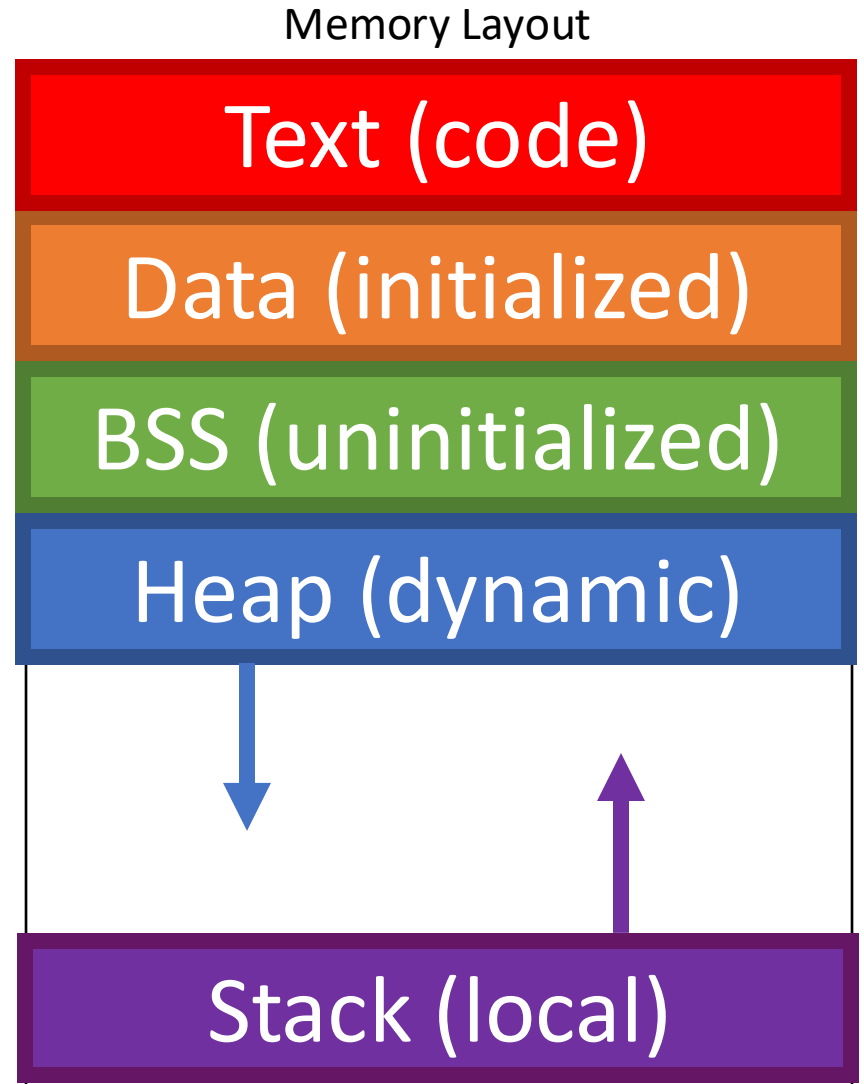
```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



Stack and Heap Resources

- <https://www.cs.princeton.edu/courses/archive/fall07/cos217/lectures/06MemoryAllocation-3x1.pdf>
- <https://doc.rust-lang.org/1.6.0/book/the-stack-and-the-heap.html>
- <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/08/lec.html>
- <https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>
- <https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap>

Principle of Locality

- Locality: programs tend to reuse data and instructions near those they have used recently
- Temporal locality: recently referenced items are likely to be referenced again in the near future
- Spatial locality: items with nearby addresses tend to be referenced close together in time

(YouTube) Efficiency with Algorithms, Performance with Data Structures

Discontiguous data structures are the root of all evil. This is simply a fact. If you don't believe me, I'll try to convince you. Specifically, please say "no" to linked lists. OK. Please. Please say no to linked lists. There is almost nothing more harmful you can do to the performance of an actual modern microprocessor than to use a linked list data structure.

Chandler Carruth (Engineer at Google) @ CppCon 2014

JUST SAY NO TO LINKED LISTS

[\(YouTube\) Efficiency with Algorithms, Performance with Data Structures](#)

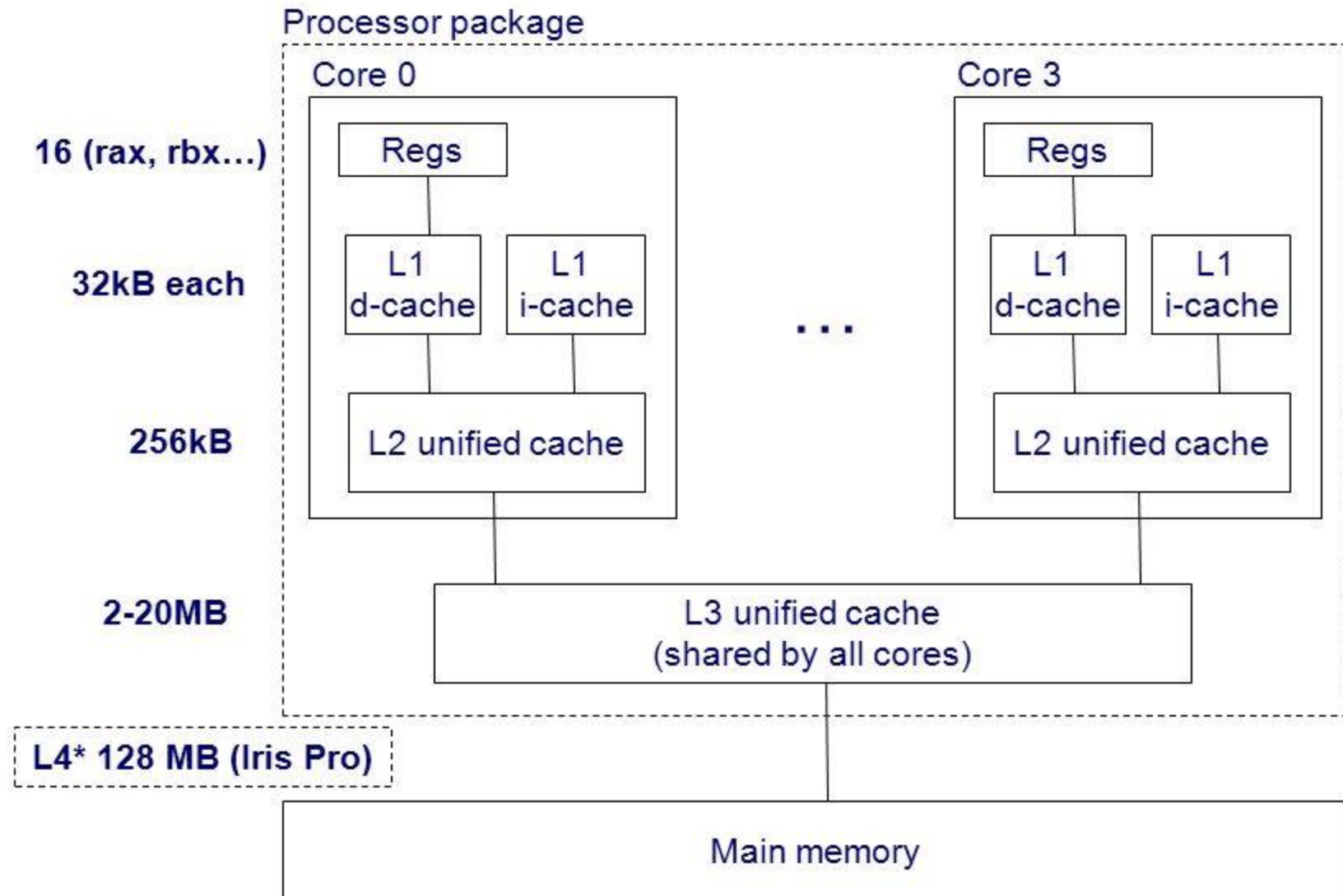
Discontiguous data structures are the root of all evil. This is simply a fact. If you don't believe me, I'll try to convince you. Specifically, please say "no" to linked lists. OK. Please. Please say no to linked lists. There is almost nothing more harmful you can do to the performance of an actual modern microprocessor than to use a linked list data structure.

Chandler Carruth (Engineer at Google) @ CppCon 2014

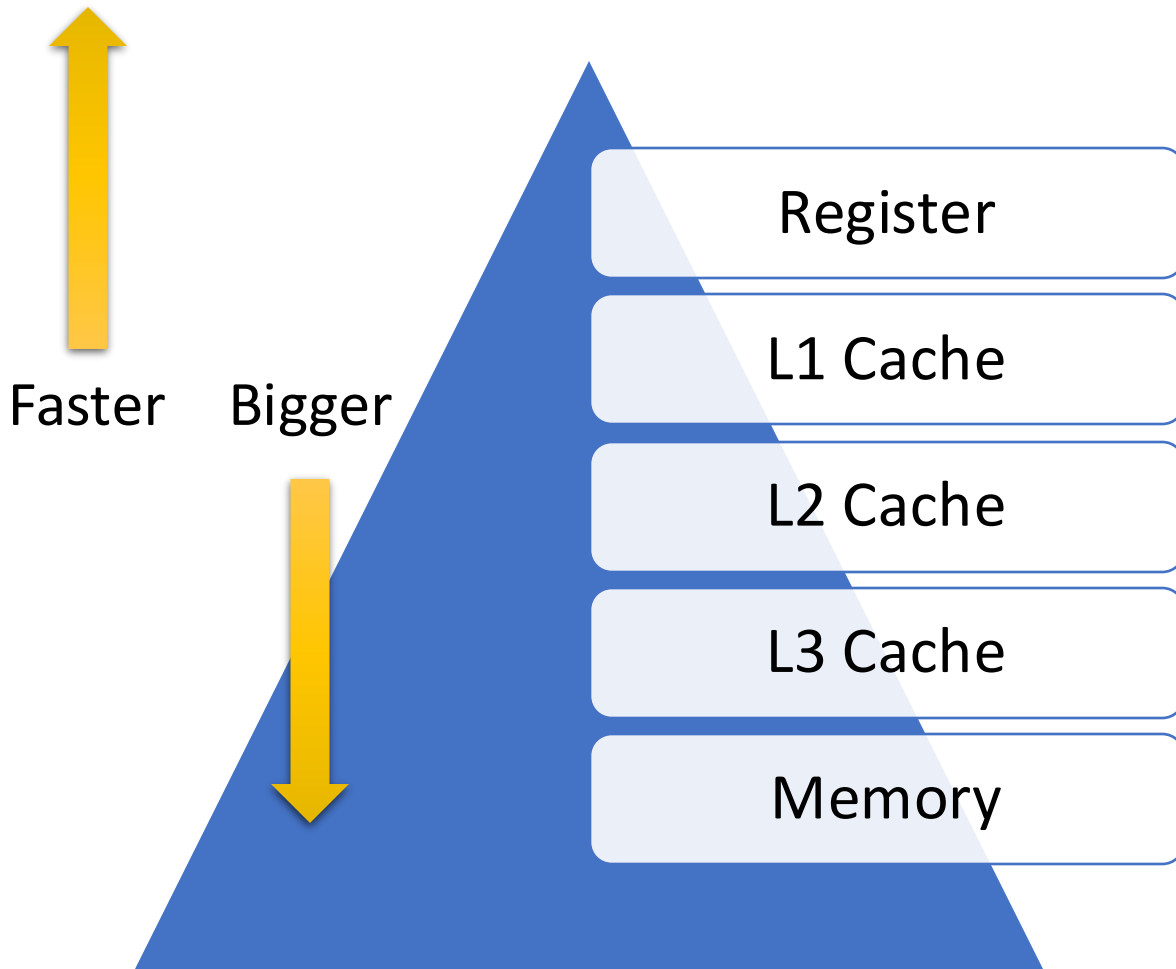
DISCONTIGUOUS DATA STRUCTURES ARE
THE ROOT OF ALL (PERFORMANCE) EVIL

Intel Core i7 cache hierarchy (2014)

Notice that
the amount
of space gets
bigger as you
go down the
hierarchy



(Simplified) Cache View

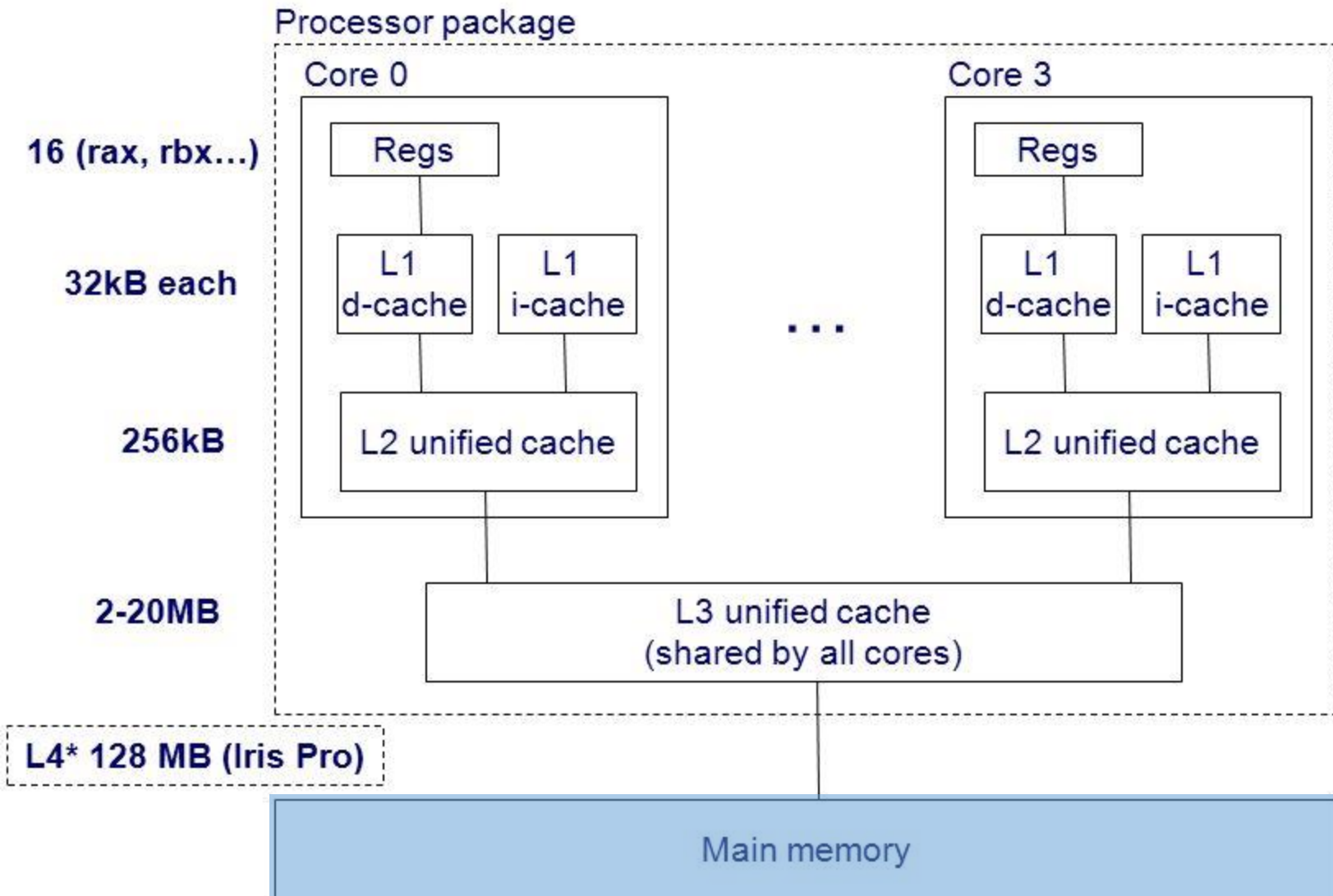
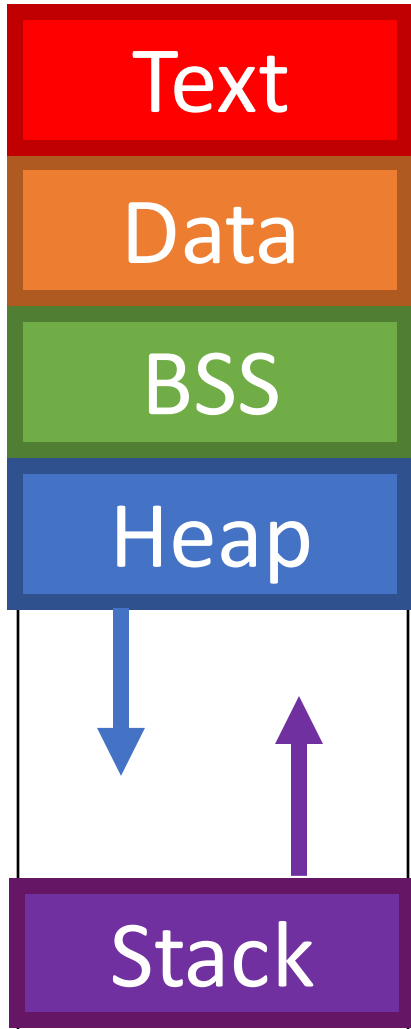


1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1 Cache
4. On **miss**, looks in L2 Cache
5. On **miss**, looks in L3 Cache
6. On **miss**, **finds** it in memory
(in “virtual” memory)

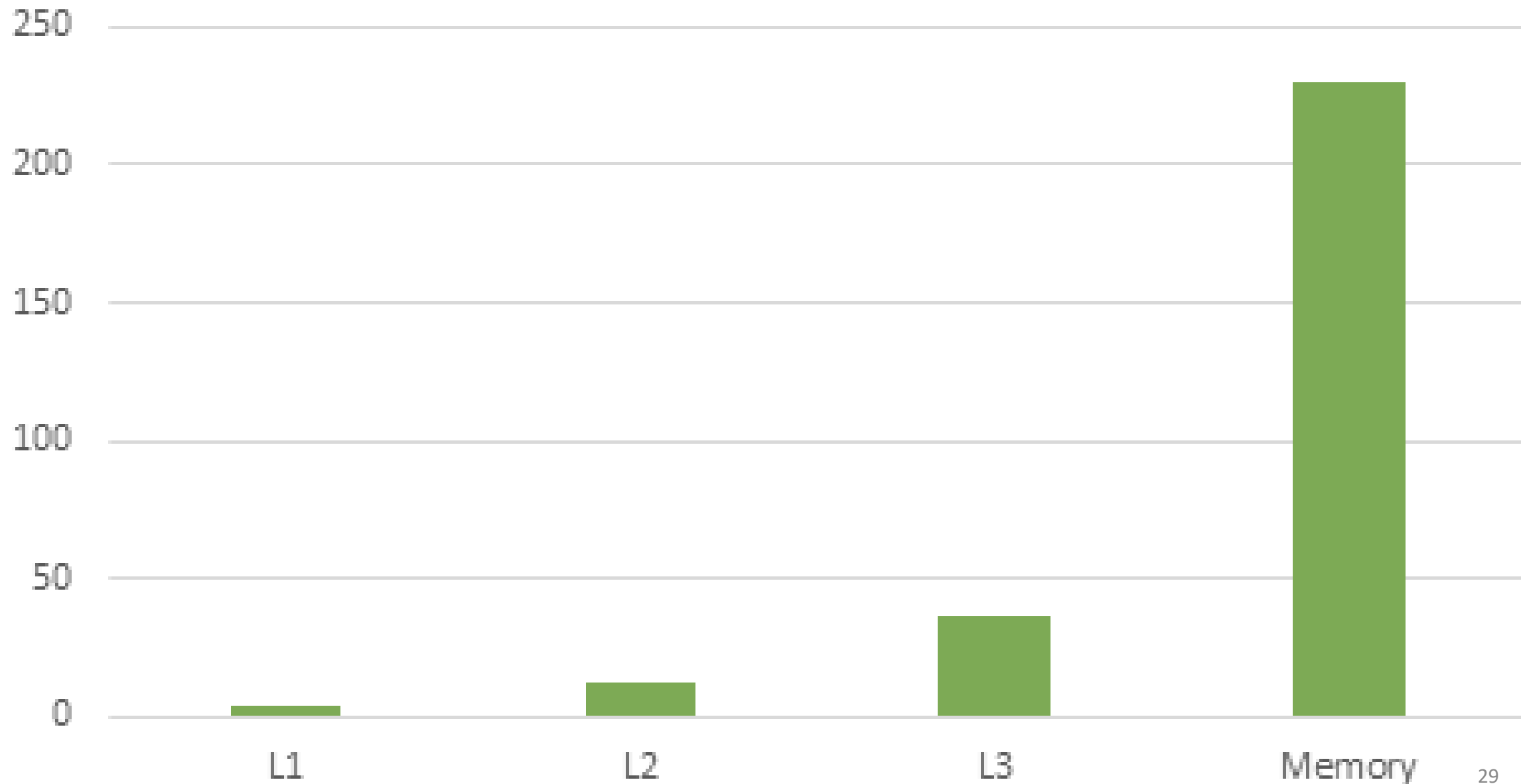
Entity	Time (nanoseconds)	Note
One cycle on a 3 GHz processor	1	
L1 cache reference	0.5	
Branch mis-predict	5	
L2 cache reference	7	14x L1 cache
Mutex lock/unlock	25	
Main memory reference	100	20x L2; 200x L1
Compress 1k bytes with Snappy	3,000	
Send 1K bytes over 1 Gbps network	10,000	
Read 4K randomly from SSD	150,000	
Read 1 MB sequentially from main memory	250,000	
Round trip with the same datacenter	500,000	
Read 1 MB sequentially from SSD	1,000,000	4x main memory
Disk seek	10,000,000	20x datacenter round trip
Read 1 MB sequentially from Disk	20,000,000	80x main memory; 20x SSD
Send package CA -> Netherlands -> CA	150,000,000	

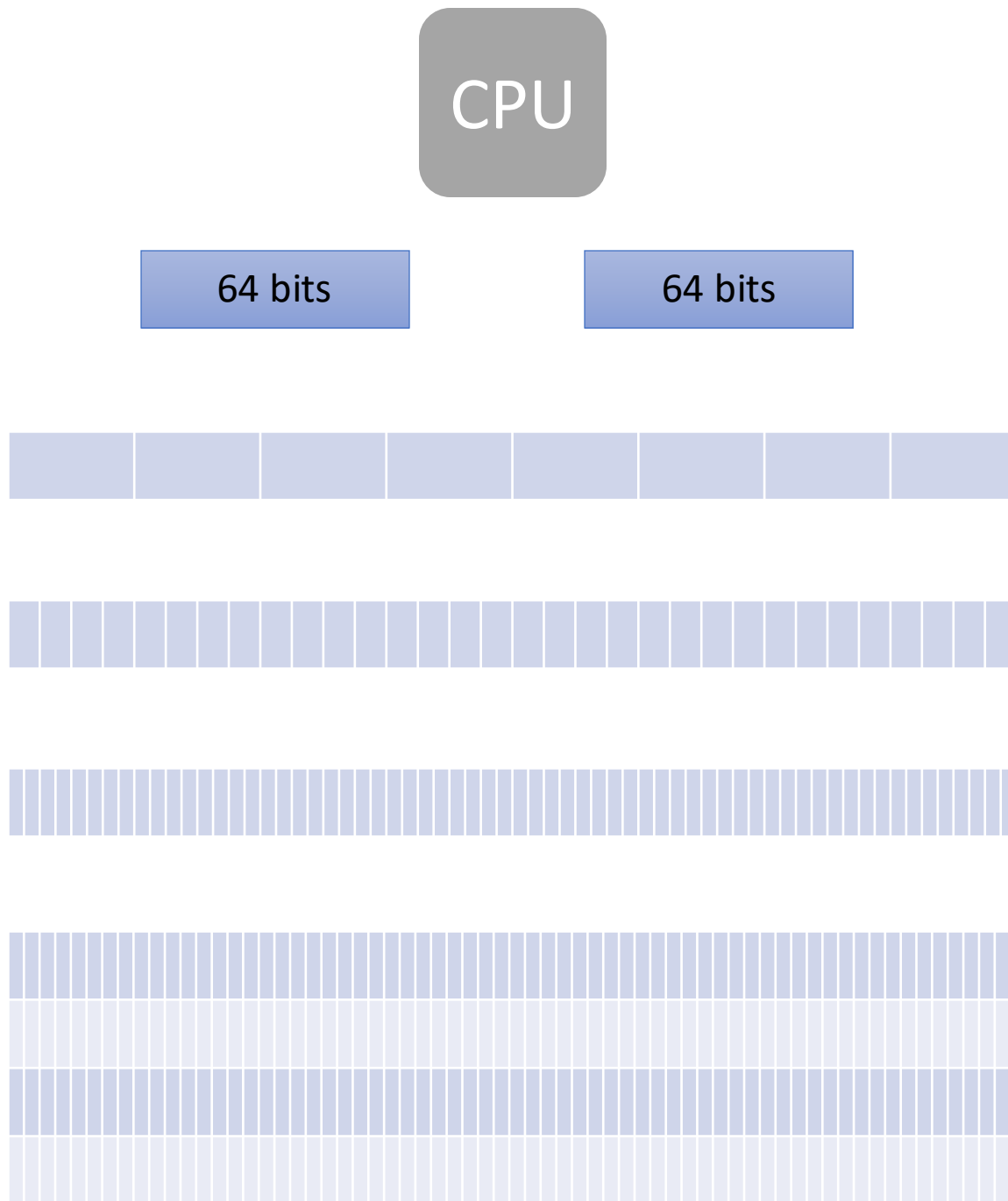
Intel Core i7 cache hierarchy (2014)

Memory Layout

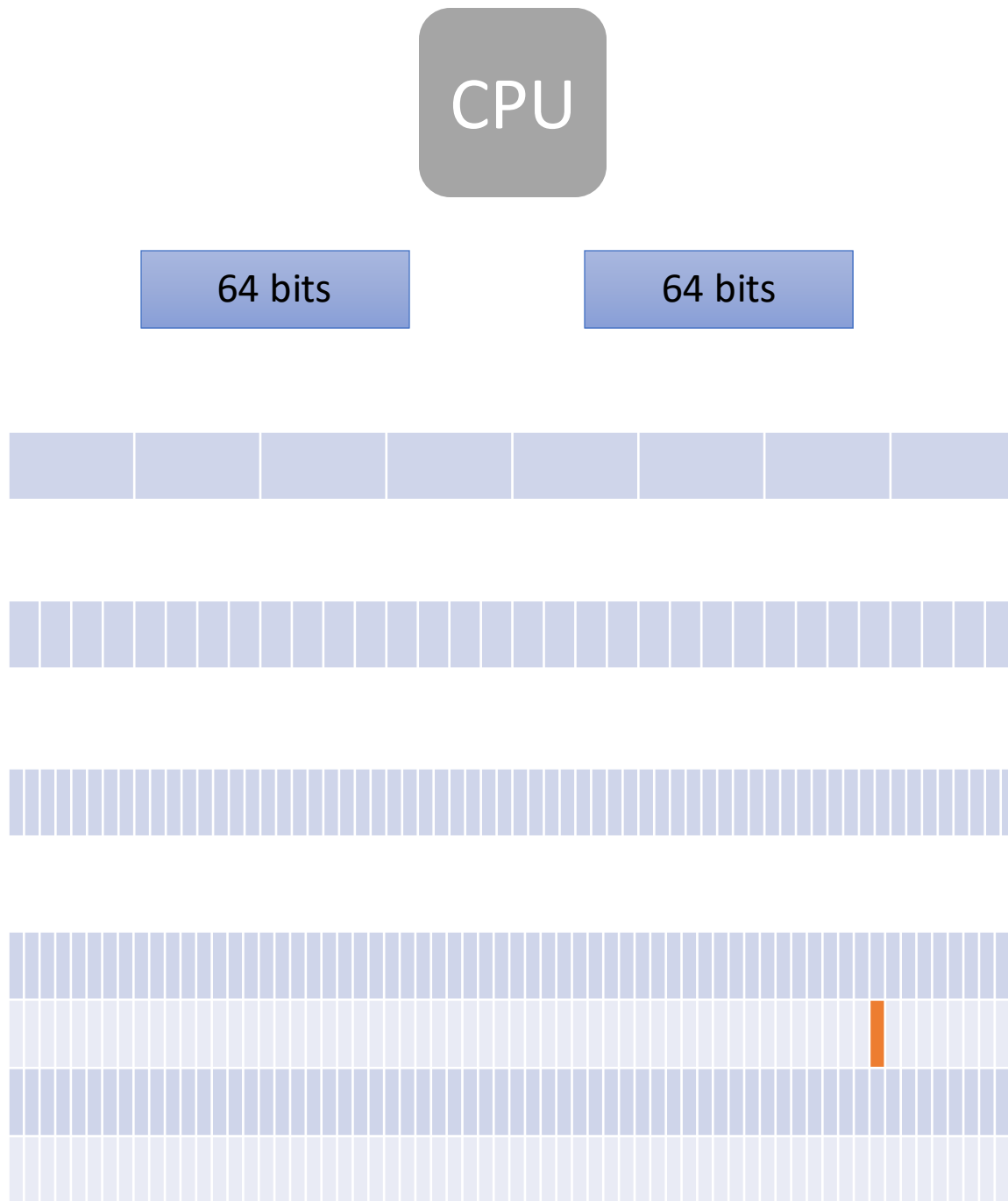


Intel Haswell Latency (cycles)

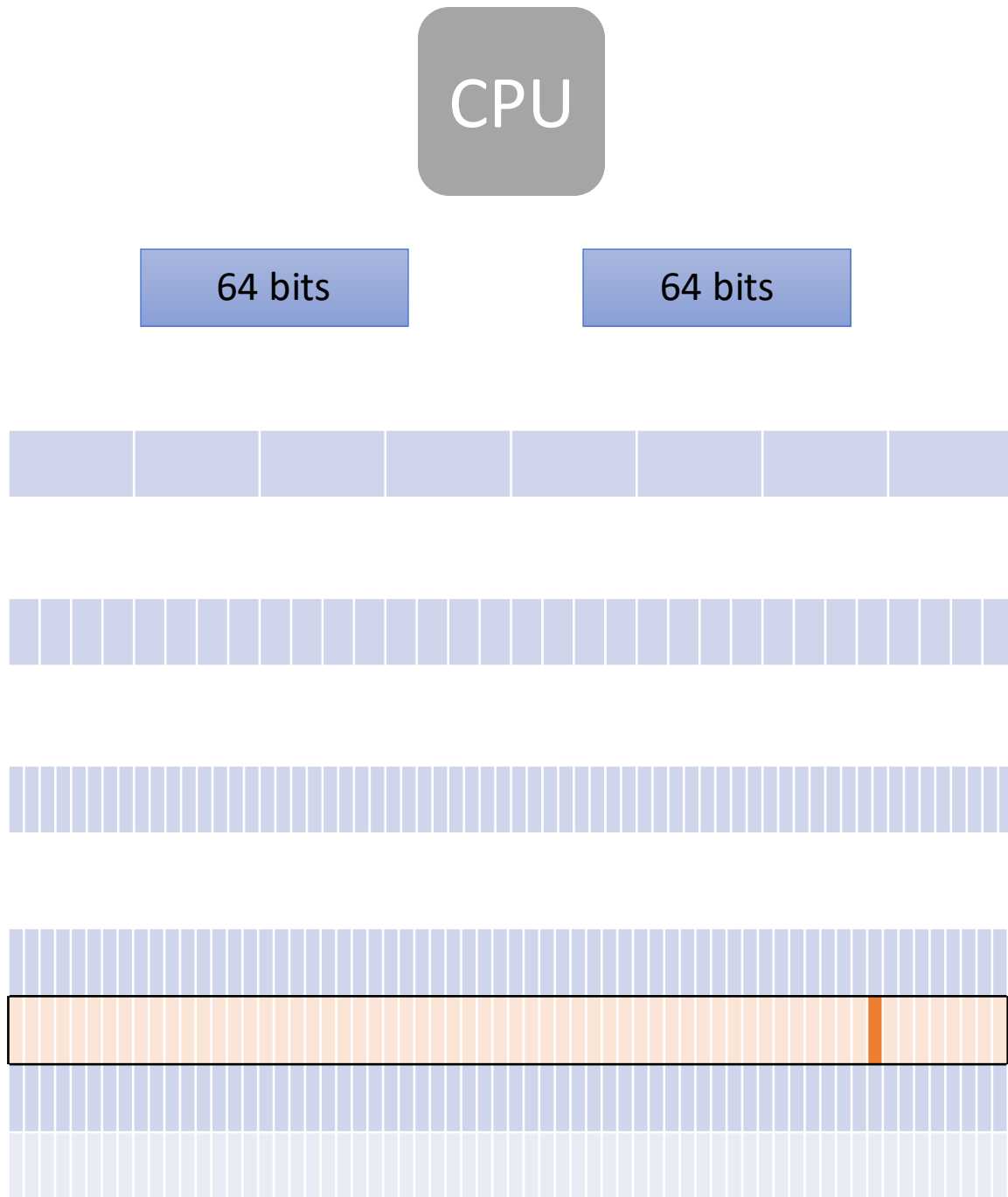




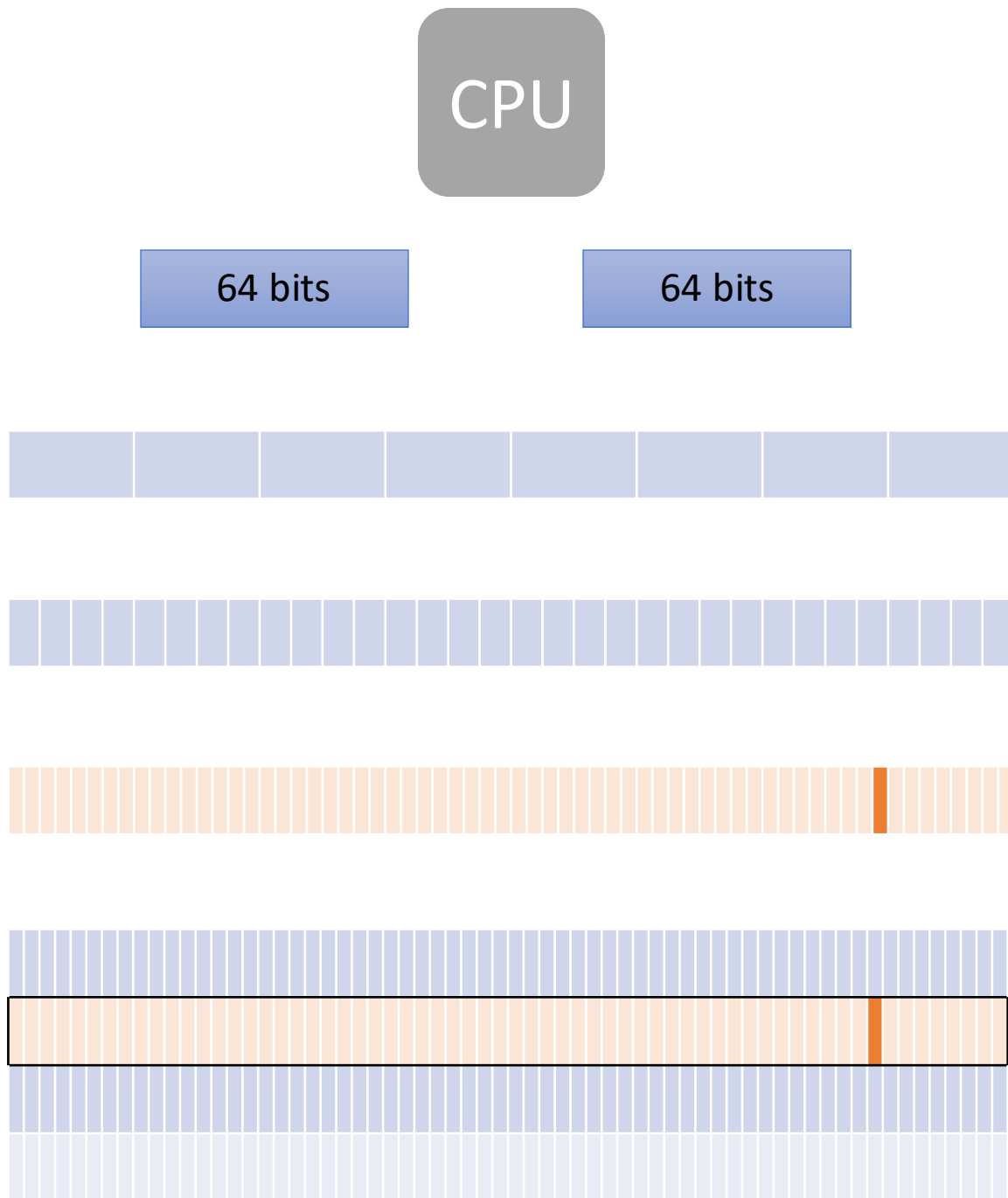
1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)



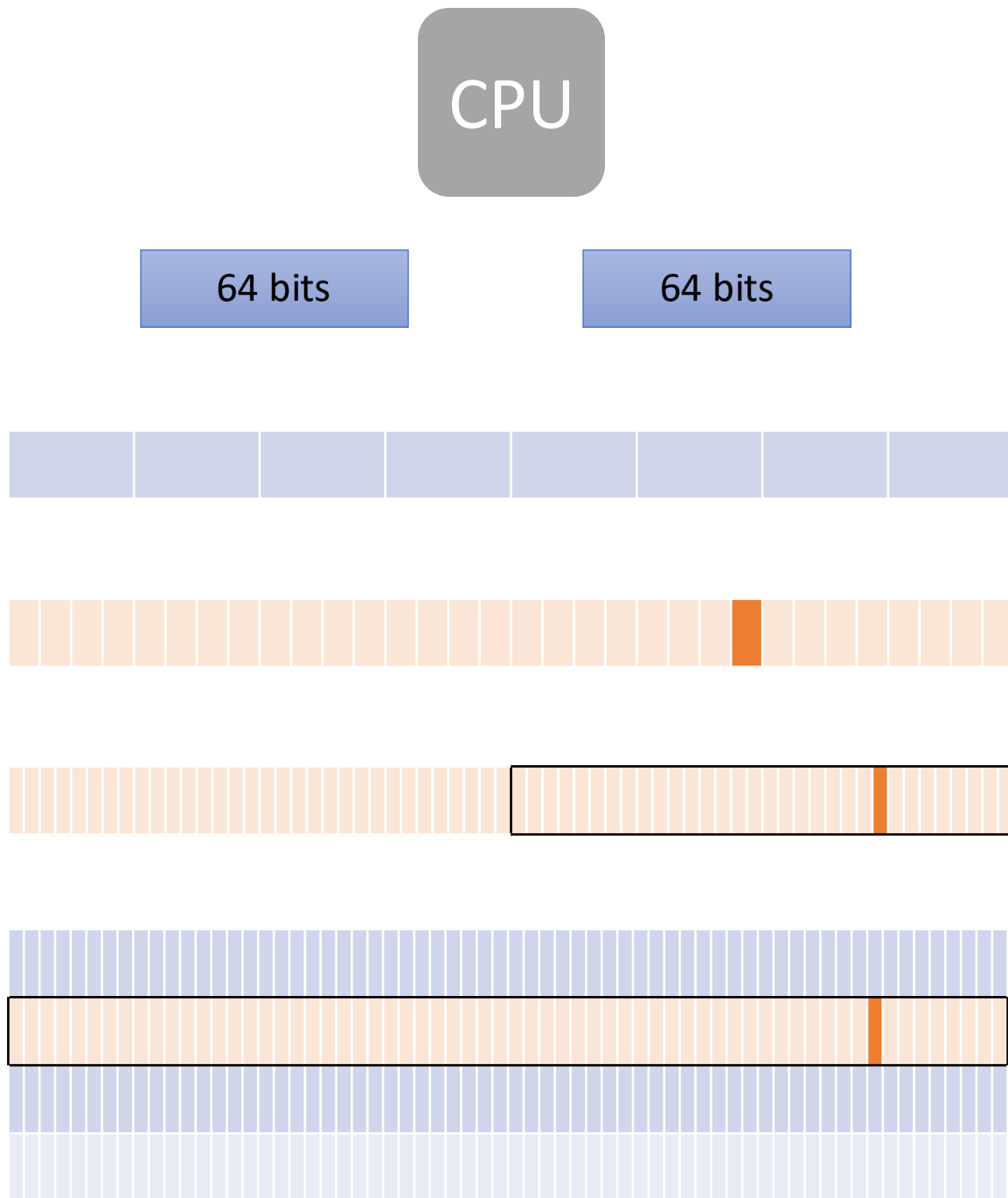
1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)



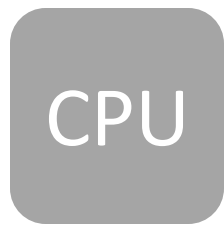
1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)



1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)

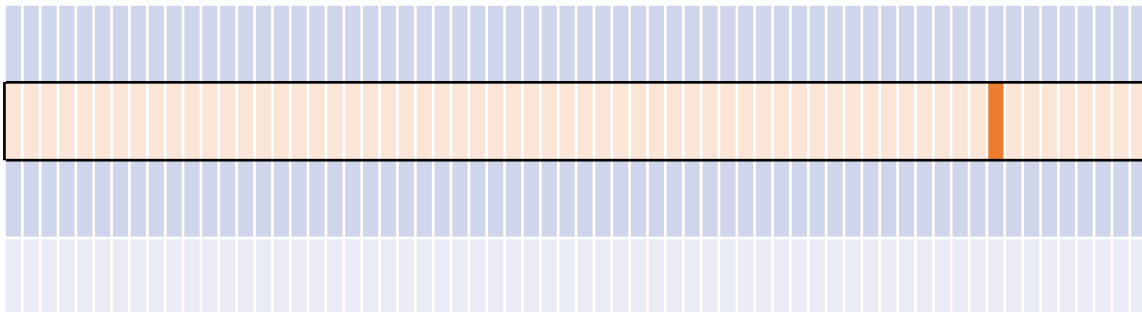
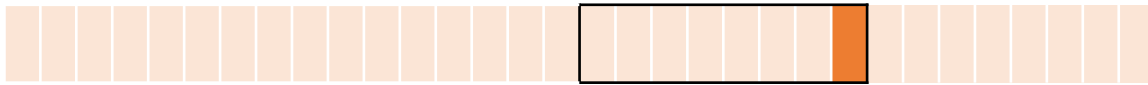
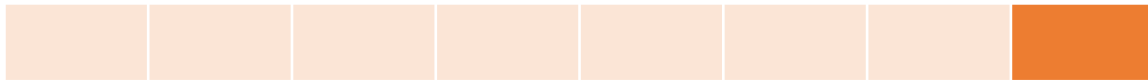


1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)

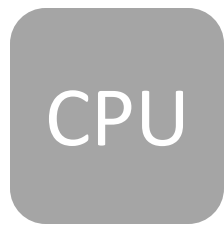


64 bits

64 bits

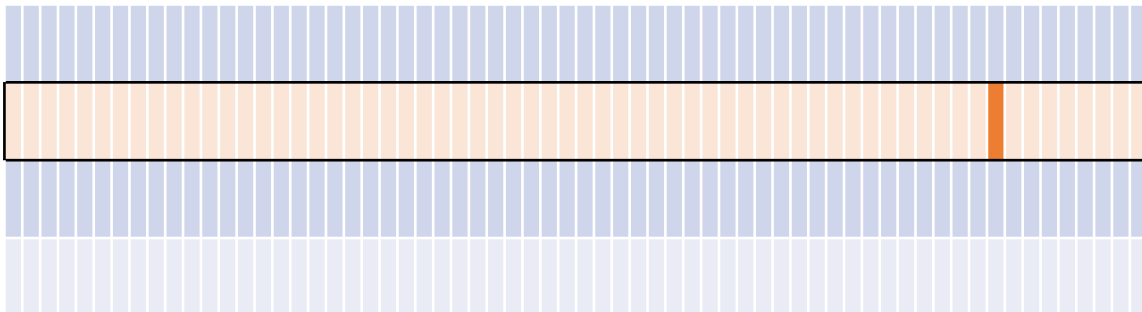
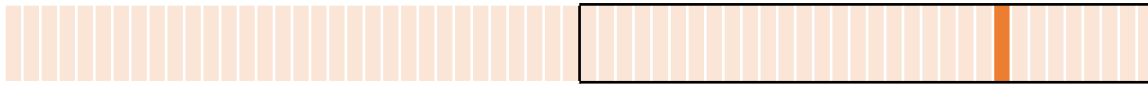
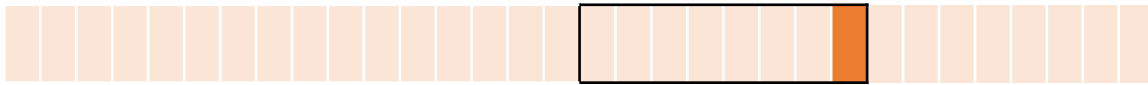
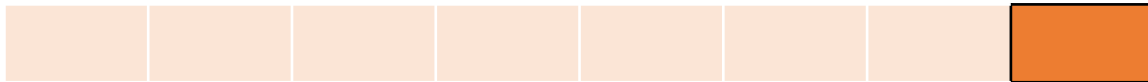


1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)



64 bits

64 bits

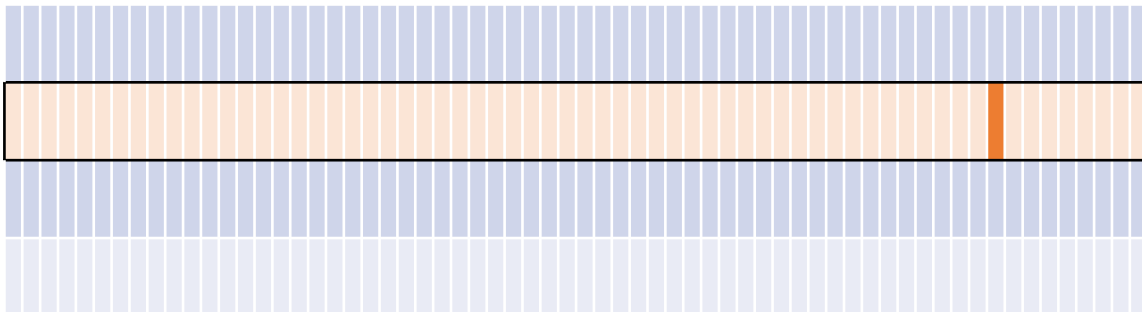
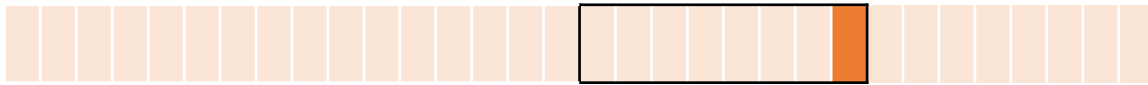
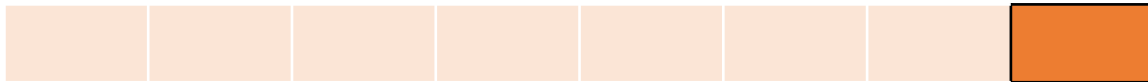


1. CPU needs a piece of memory
 2. Needs to load it into a register
 3. Looks in L1
 4. On **miss**, looks in L2
 5. On **miss**, looks in L3
 6. On **miss**, **finds** it in memory
- (in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

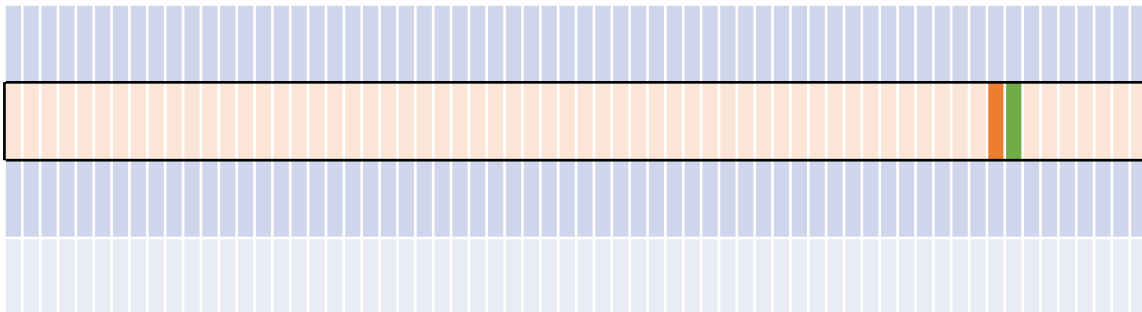
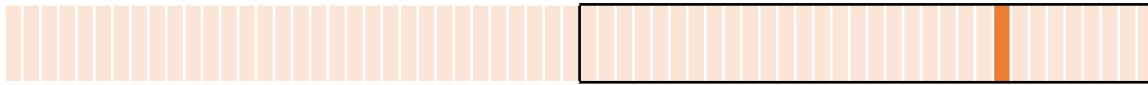
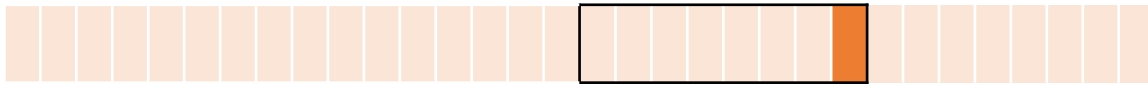
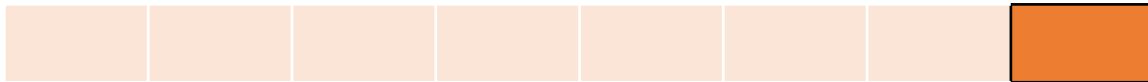
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

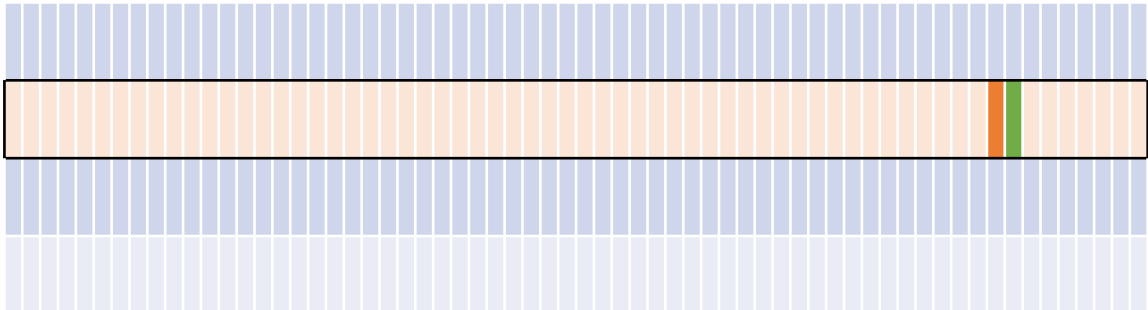
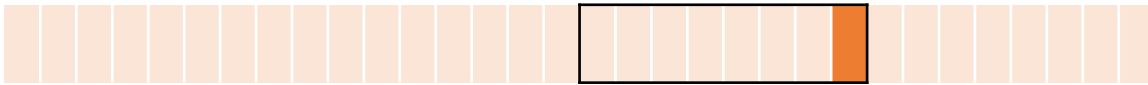
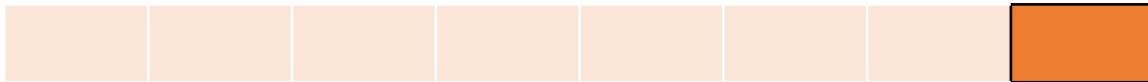
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

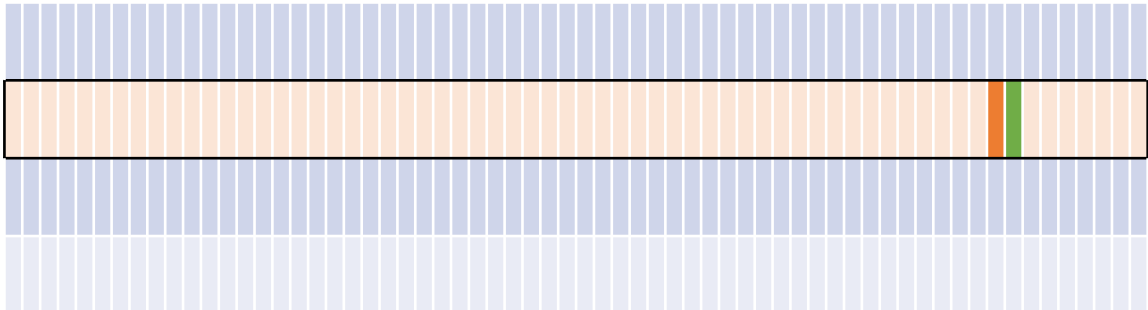
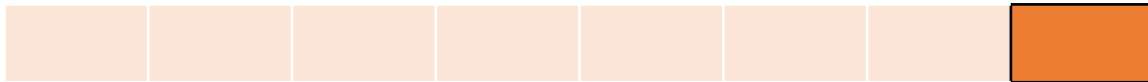
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

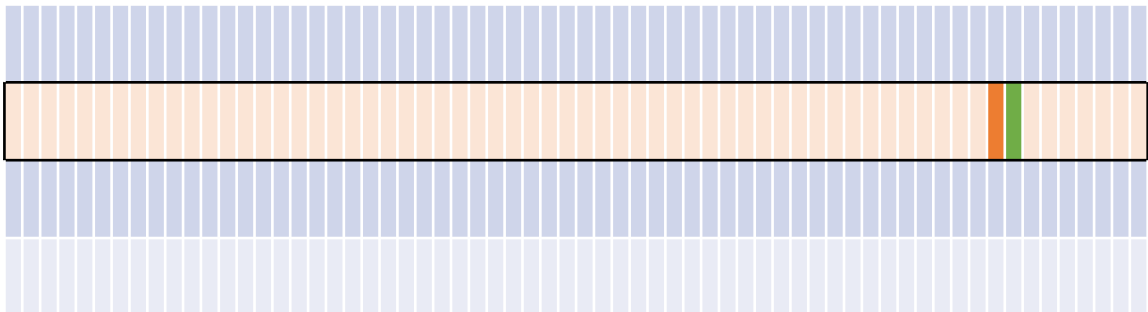
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

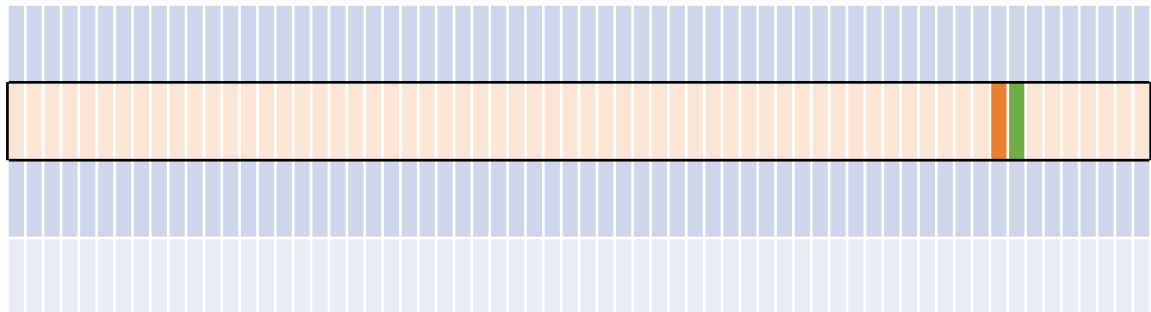
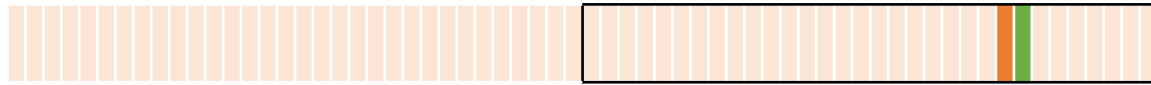
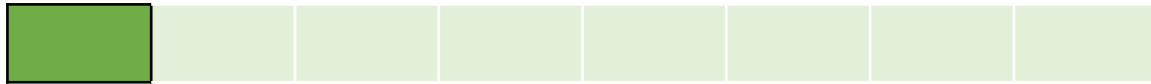
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

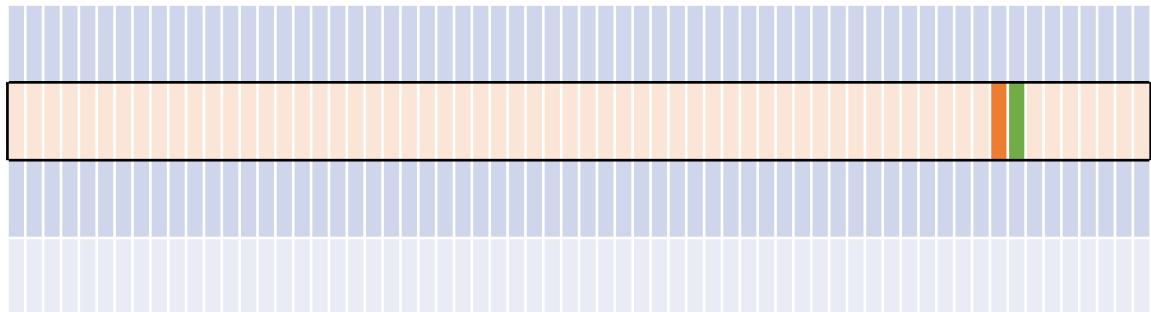
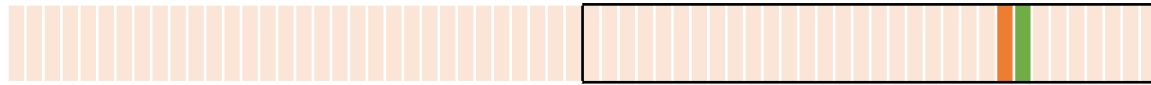
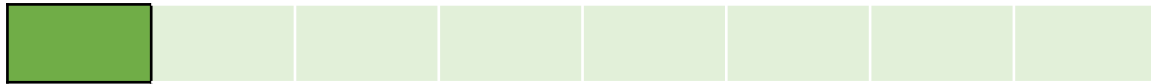
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item was right next to the first item?

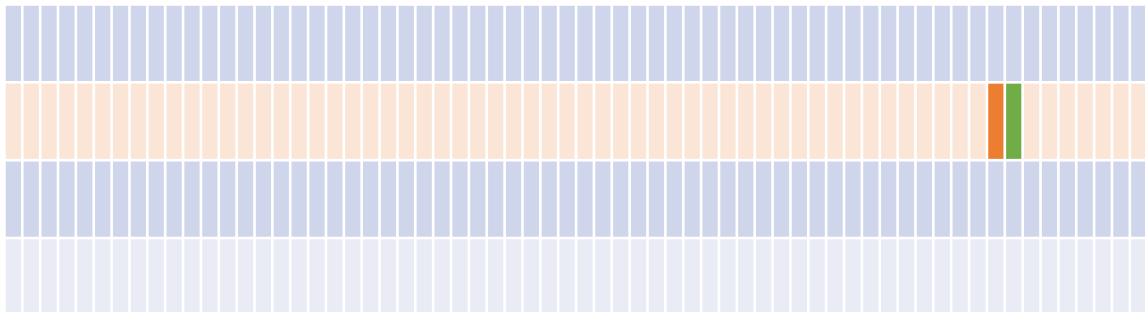
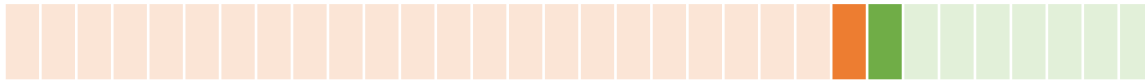
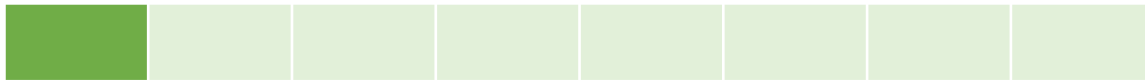
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits



What if the next item is far away from these two?

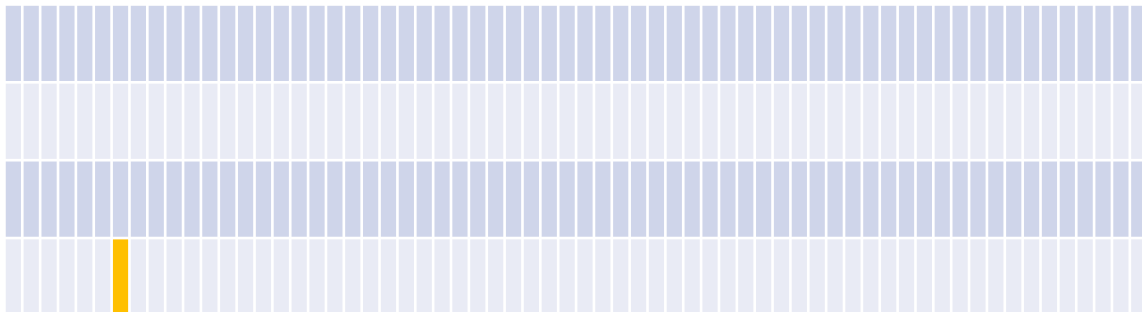
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

CPU

64 bits

64 bits

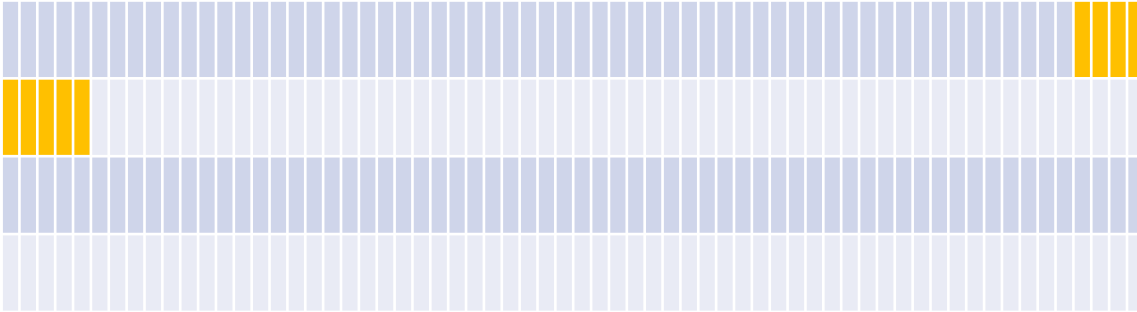


What if the next item is far away from these two?

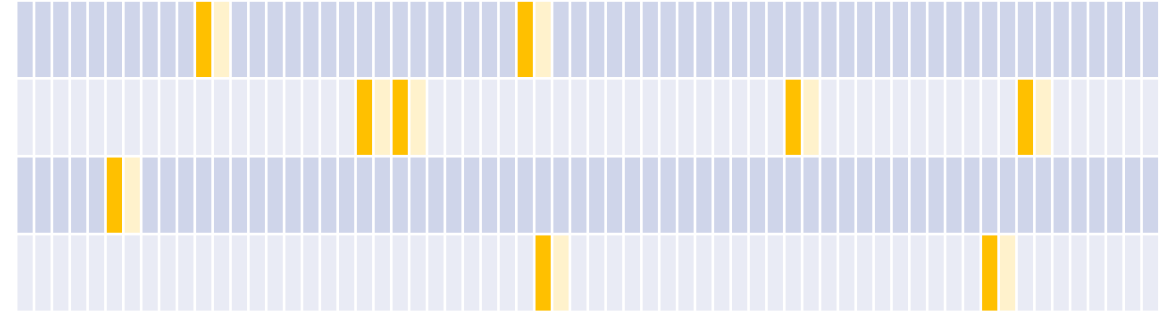
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, **finds** it in memory

(in “virtual” memory)

std::vector

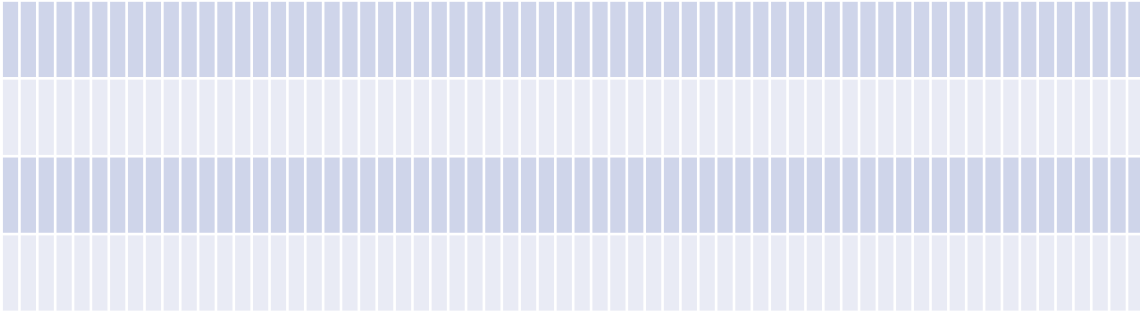


vs std::list

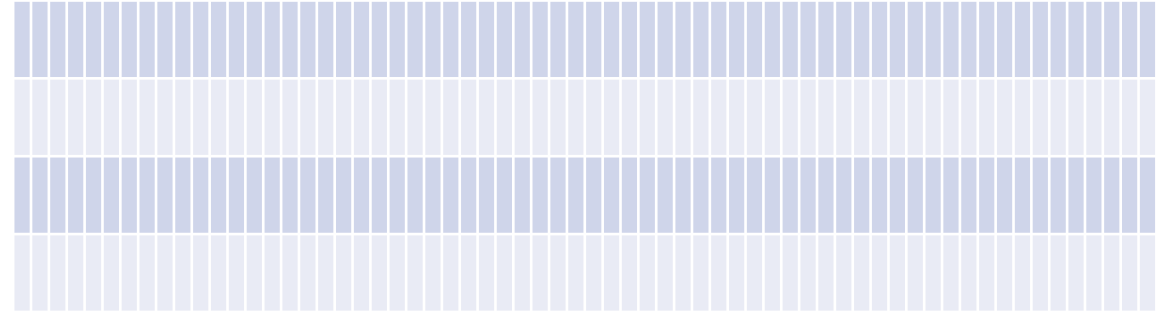


Which has better locality?
Meaning, which will work better with cache?

Adjacency Matrix



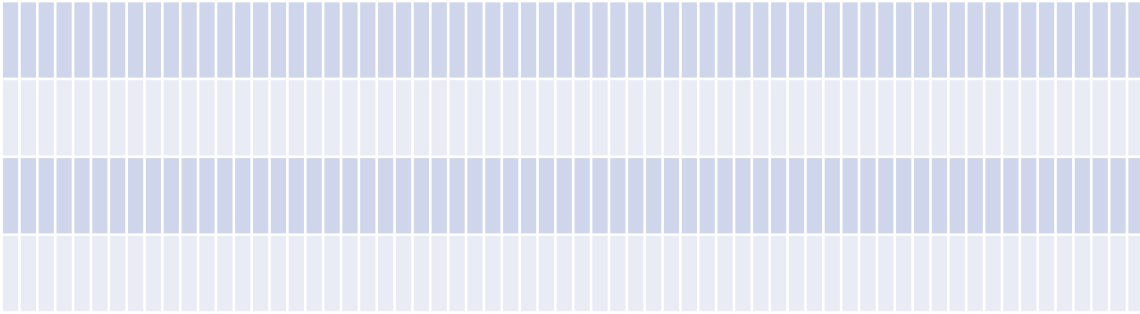
Adjacency List



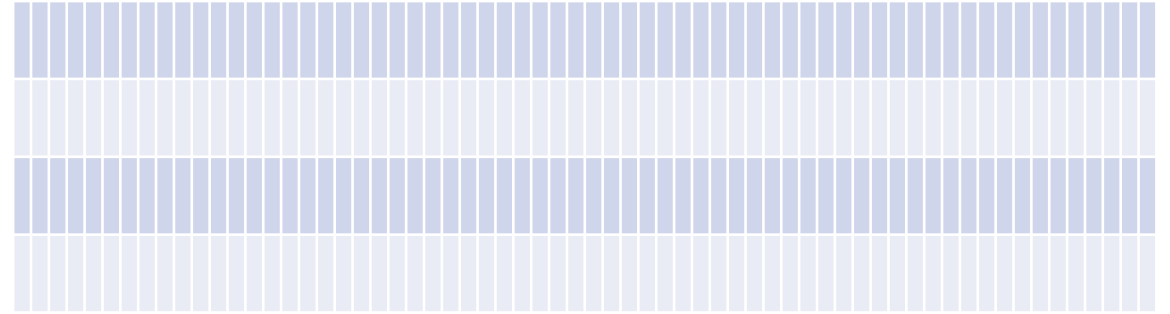
What does this picture look like for:

1. An Adjacency Matrix
2. An Adjacency List

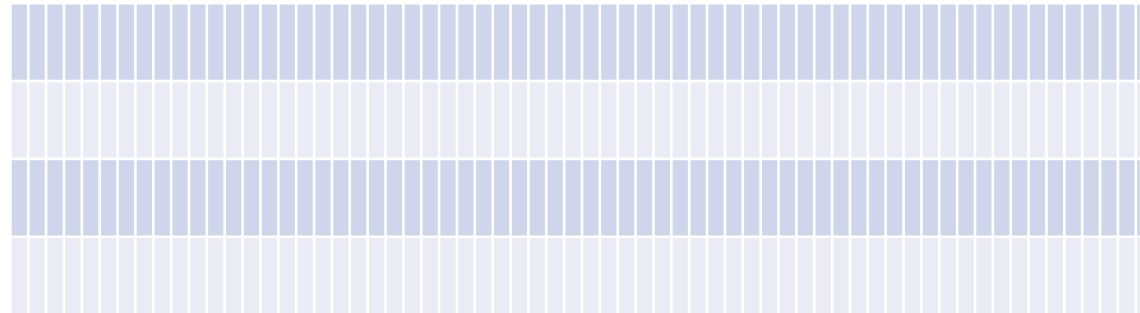
Sorted Array



Search Tree



Heap



What does
extract min
look like?

What does this picture look like for:

1. A Sorted Array
2. A Binary Search Tree
3. A Heap