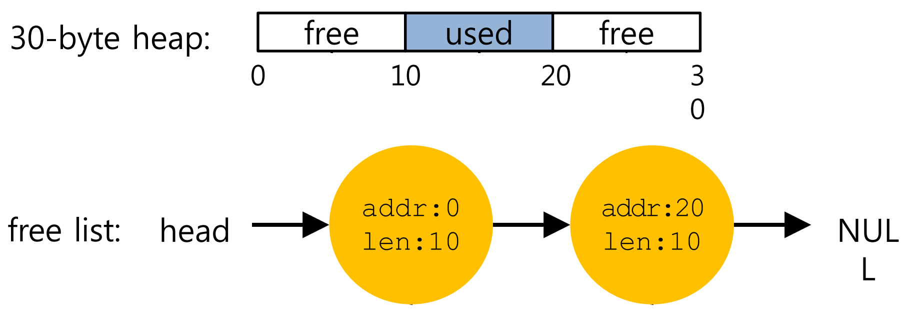
OSTEP

Memory Virtualization
Free Space Management

Splitting

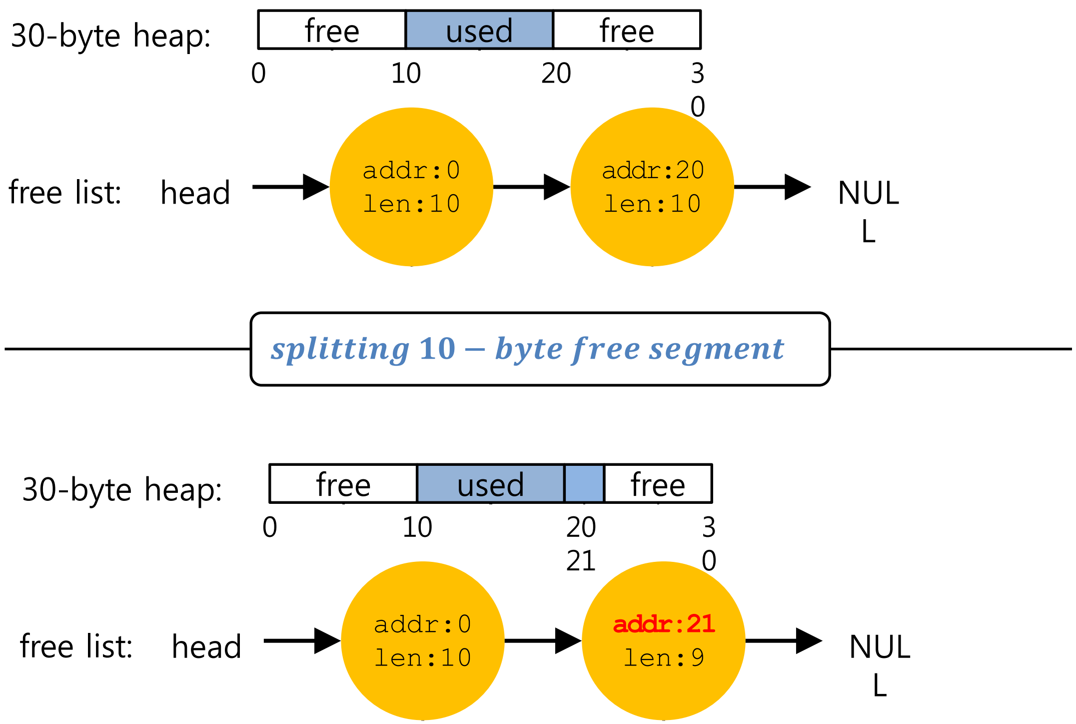
Finding a free chunk of memory that can satisfy the request and splitting it into two.

- When request for memory allocation is **smaller** than the size of free chunks.



Splitting(Cont.)

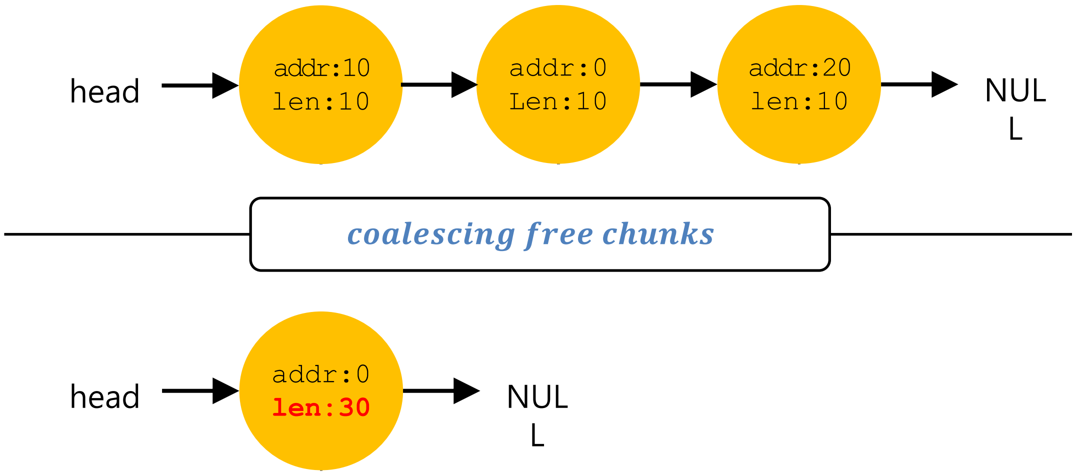
Two 10-bytes free segment with **1-byte request**



Coalescing

If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.

Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



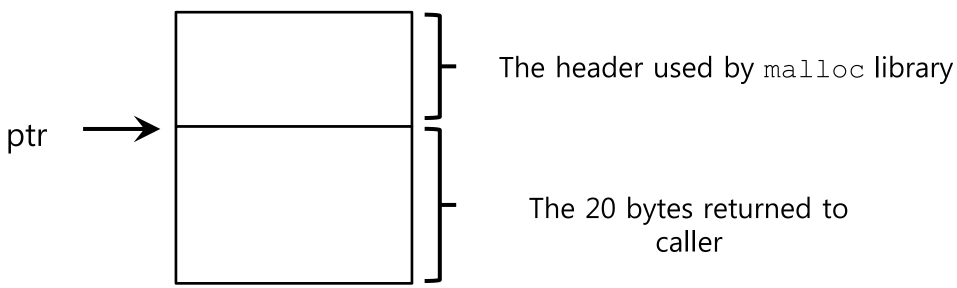
Tracking The Size of Allocated Regions

The interface to `free(void *ptr)` does **not take** a **size parameter**.

- How does the library **know the size** of memory region that will be back **into free list**?

Most allocators store **extra information** in a **header** block.

```
ptr = malloc(20);
```



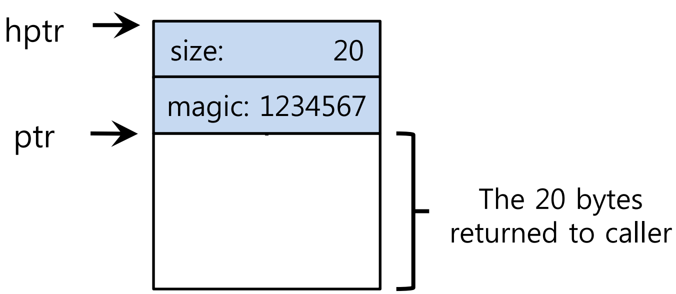
An Allocated Region Plus Header

The Header of Allocated Memory Chunk

The header minimally **contains the size** of the allocated memory region.

The header may also contain

- Additional pointers to speed up deallocation
- A magic number for integrity checking



Specific Contents Of The Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

The **size** for free region is the **size of the header plus the size of the space** allocated to the user.

- If a user **request N bytes**, the library searches for a free chunk of **size N plus the size of the header**

Simple pointer arithmetic to find the header pointer.

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

Embedding A Free List

The memory-allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**.

- The library **can't use** `malloc()` to build a list **within itself**.

Embedding A Free List(Cont.)

Description of a node of the list

```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

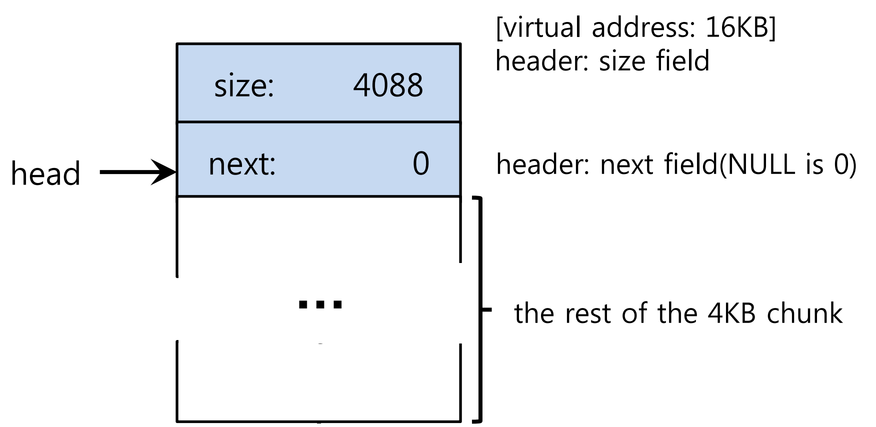
Building heap and putting a free list

- Assume that the heap is built vi `mmap()` system call.

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

A Heap With One Free Chunk

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



Embedding A Free List: Allocation

If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.

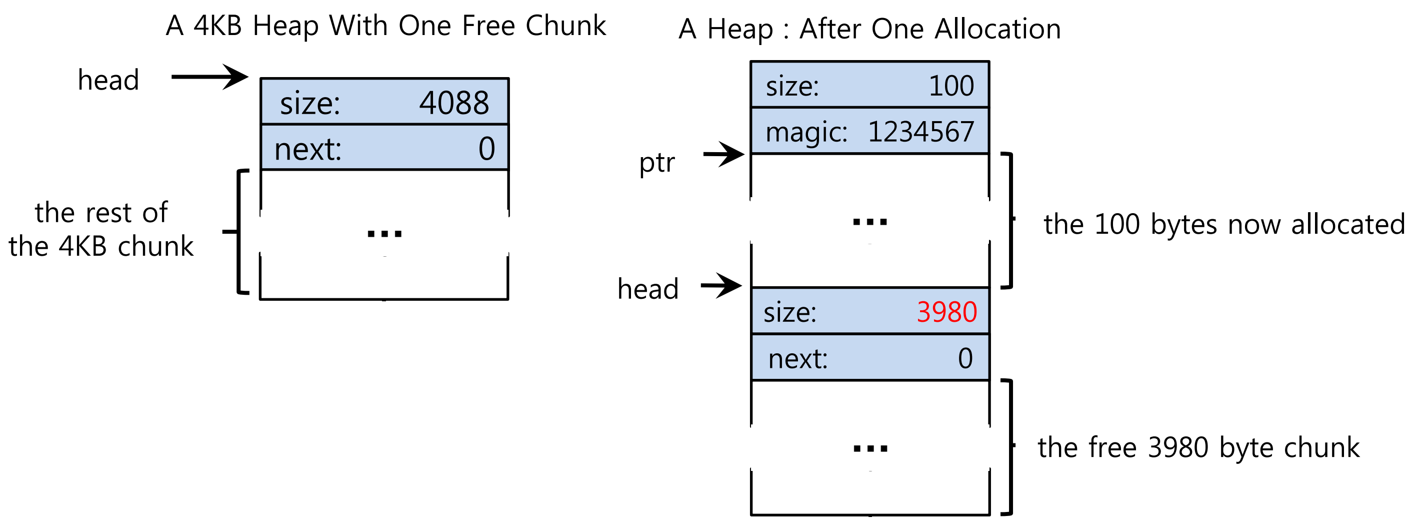
The library will

- **Split** the large free chunk into two.
  - **One** for the **request** and the **remaining** free chunk
- **Shrink** the size of free chunk in the list.

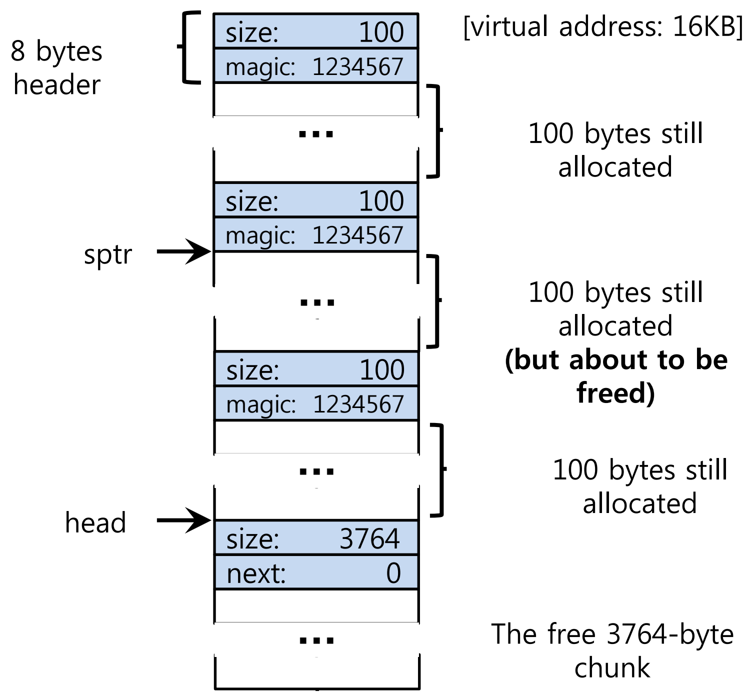
Embedding A Free List: Allocation(Cont.)

Example: a request for 100 bytes by `ptr = malloc(100)`

- Allocating 108 bytes out of the existing one free chunk.
- shrinking the one free chunk to 3980(4088 minus 108).



Free Space With Chunks Allocated

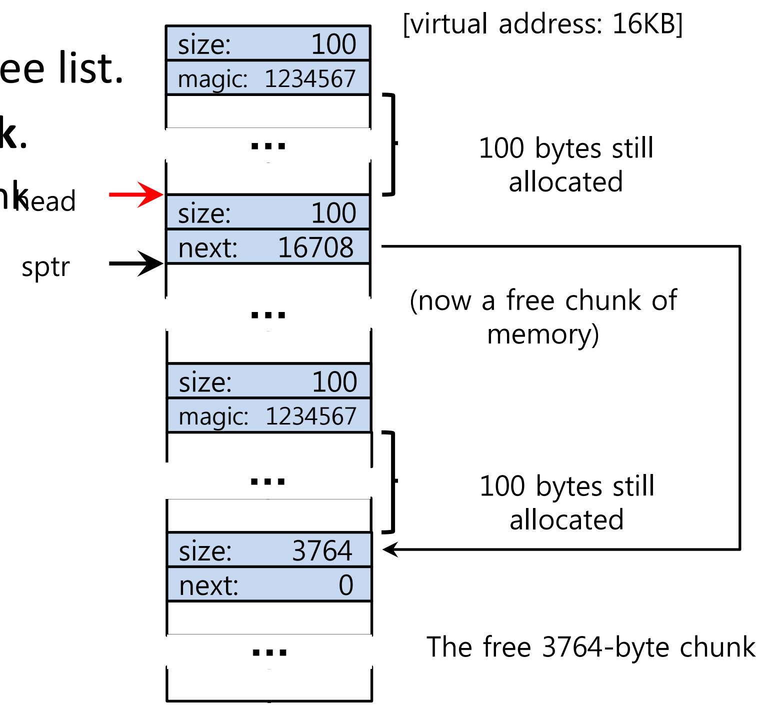


Free Space With Three Chunks Allocated

Free Space With `free()`

Example: `free(sptr)`

- The 100 bytes chunks is **back into** the free list.
- The free list will **start** with **a small chunk**.
  - The list header will point the small chunk

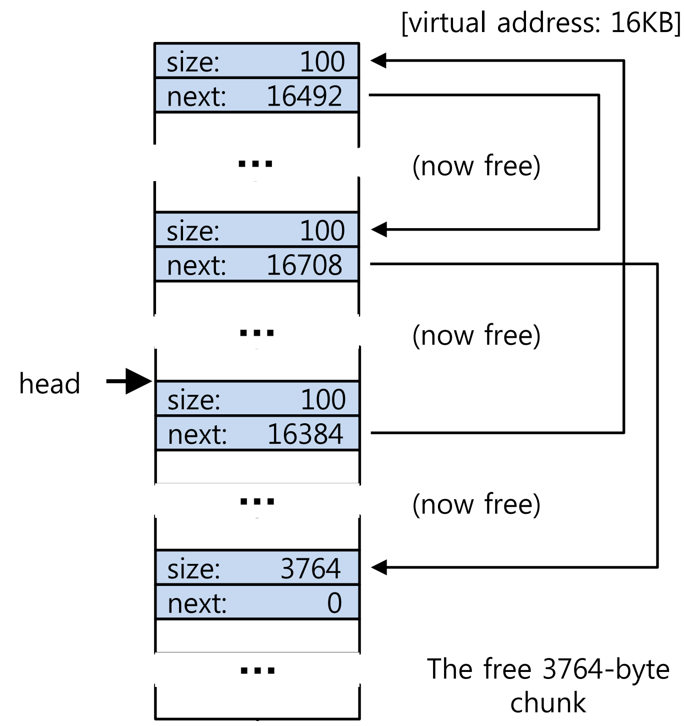


Free Space With Freed Chunks

Let's assume that the last two in-use chunks are freed.

External Fragmentation occurs.

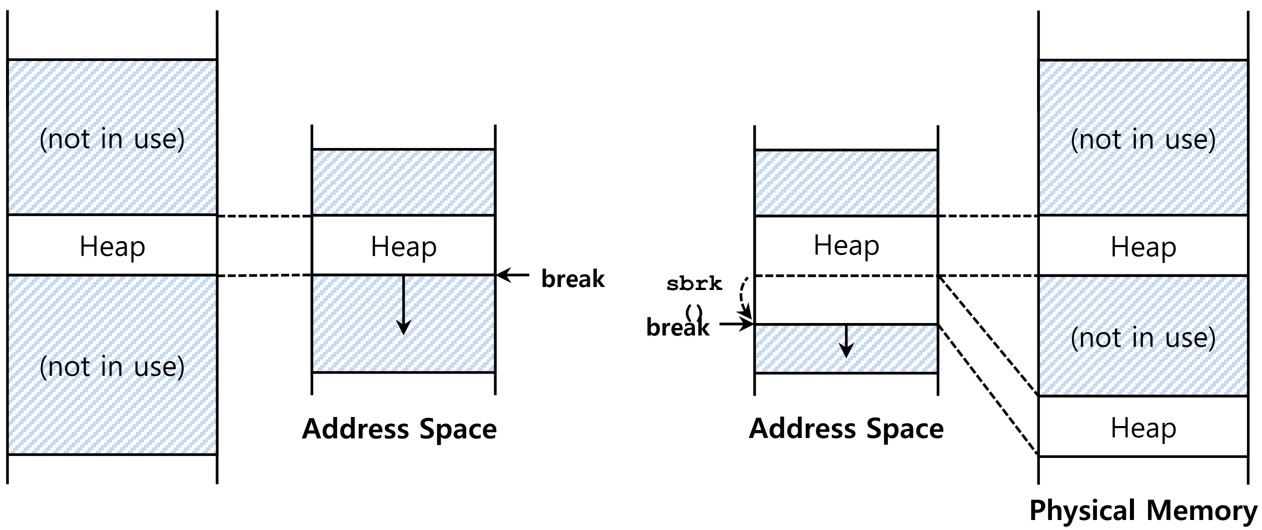
- **Coalescing** is needed in the list.



Growing The Heap

Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out.

- e.g., `sbrk()`, `brk()` in most UNIX systems.



Managing Free Space: Basic Strategies

Best Fit:

- Finding free chunks that are **big or bigger than the request**
- Returning the **one of smallest** in the chunks **in the group** of candidates

Worst Fit:

- Finding the **largest free chunks** and allocation the amount of the request
- **Keeping the remaining chunk** on the free list.

Managing Free Space: Basic Strategies(Cont.)

First Fit:

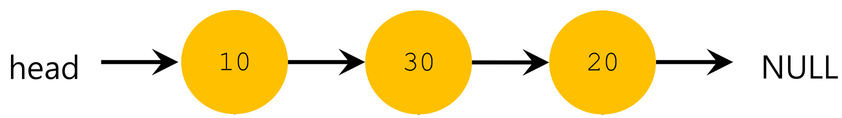
- Finding the **first chunk** that is **big enough** for the request
- Returning the requested amount and remaining the rest of the chunk.

Next Fit:

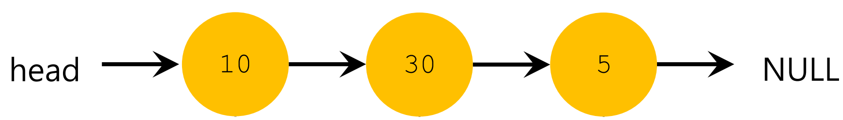
- Finding the first chunk that is big enough for the request.
- Searching at **where one was looking** at instead of the begging of the list.

Examples of Basic Strategies

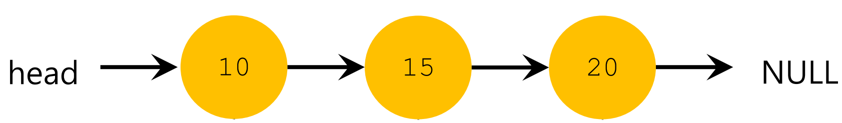
Allocation Request Size 15



Result of Best-fit



Result of Worst-fit



Other Approaches: Segregated List

Segregated List:

- Keeping free chunks in different size in a separate list for the size of popular request.
- New Complication:
  - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
- **Slab allocator** handles this issue.

Other Approaches: Segregated List(Cont.)

Slab Allocator

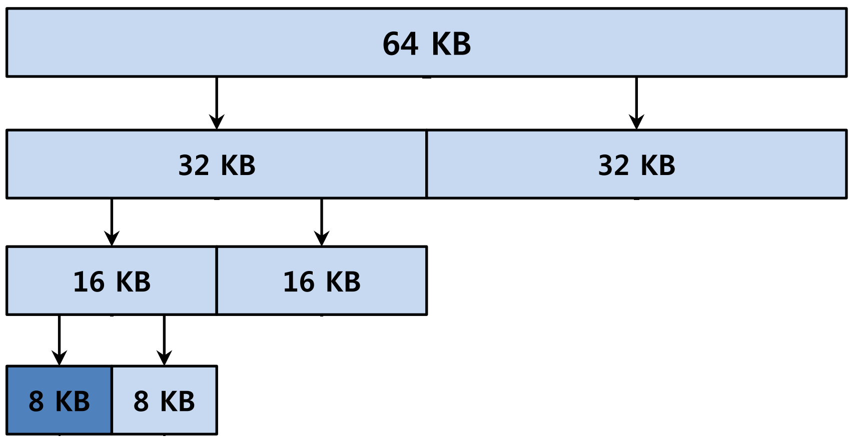
- Allocate a number of object caches.
  - The objects are likely to e requested frequently.
  - e.g., locks, file-system inodes, etc.

- **Request some memory** from a more general memory allocator when **a given cache is running low** on free space.

Other Approaches: Buddy Allocation

Binary Buddy Allocation

- The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.



64KB free space for 7KB request

Other Approaches: Buddy Allocation(Cont.)

Buddy allocation can suffer from **internal fragmentation**.

Buddy system makes **coalescing** simple.

- **Coalescing** two blocks in to the next level of block.