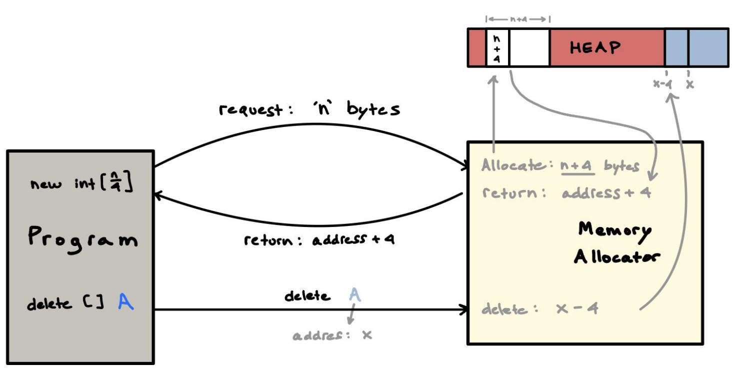
Memory Allocator

- takes care of **heap** memory-management
- Uses a ***data structure*** to remember how memory was allocated.
    - The data structure is itself, in memory

Memory Allocator is <u>given a Heap</u>:

- A program is allocated a block of the RAM memory to use as it’s *heap*
- The heap is part of RAM memory that is assigned to the program
- This block can start and end at any global/RAM memory address. Not up to us.

Mutator-Allocator Relationship:

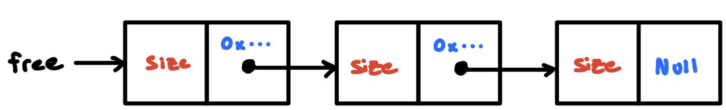


Note: new int[4]

- sizeof(int) = 4 bytes
- 16 bytes total

Free List Algorithm

<u>**Data Structure**</u>



- **linked list**
- Each ***Node*** contains: size of block, address of next free block
- stored in the **heap**
- **free**
    - pointer to the head of the list
    - stored in **global memory**

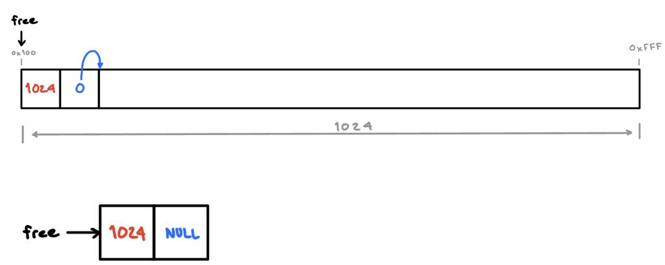
1. **Maintain the free list in increasing address order**
2. **Combine any adjacent free blocks**
    a. 2 free blocks adjacent to it ⇒ 2 merges are required
    b. 1 free block adjacent to it ⇒ 1 merge required
    c. 0 free blocks adjacent to it ⇒ 0 merges are required

<table>
<tr><th colspan="2">Example</th></tr>
<tr><td>

Starting Block:

- 1K (1024 bytes) heap
- Starts at 0x100 (256)
- ends at 0x4FF (1279)
- **free** → first address(0x100)

</td><td>



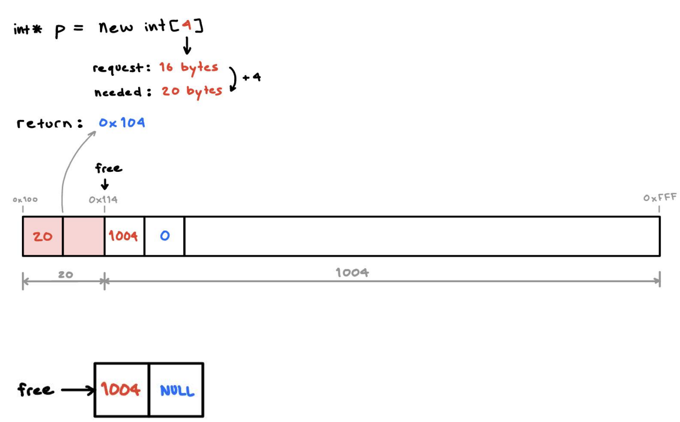
</td></tr>
<tr><td>

new int[4]

- 16 byte <u>request</u>
- 20 bytes <u>needed</u>

1. Look through free list for (20-bytes)
2. Split free-block into 2
    a. block A: size 20
    b. block B: size 1004
3. **Return** the address of **second word** of block A

</td><td>

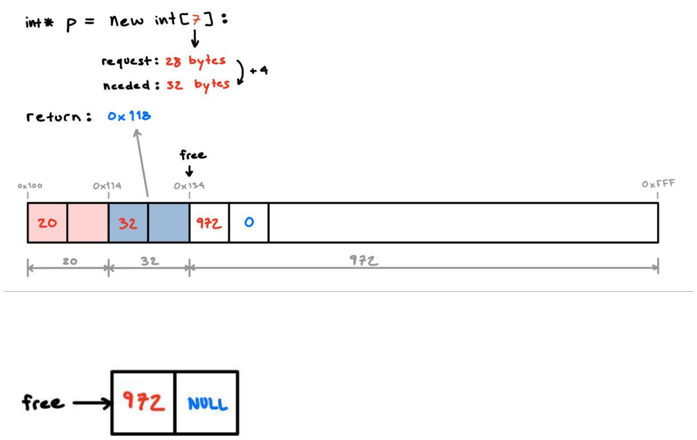


</td></tr>
</table>

new int[7]

- 28 byte <u>request</u>
- 32 bytes <u>needed</u>

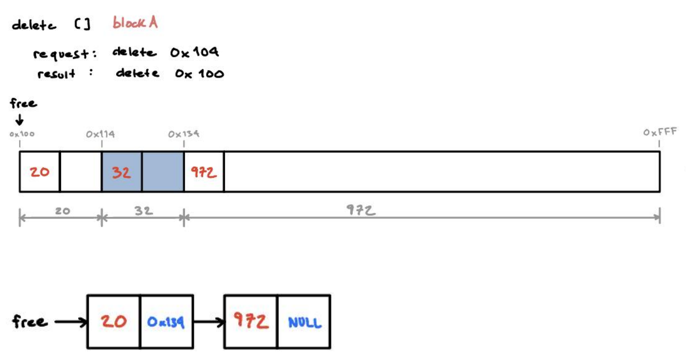
1. Look for (32-bytes)
2. Split free-block:
   a. block A: size 32
   b. block B: size 972
3. **Return** the address of **second word** of block A



delete *blockA*

- **Request:** delete 0x104
- **Delete:** 0x100

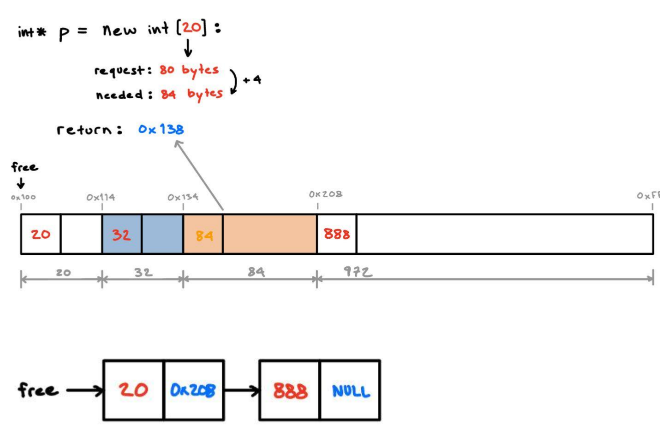
1. Add (20, 0x134) to free List
2. Free → 0x100



new int[20]

- 80 byte <u>request</u>
- 84 bytes <u>needed</u>

1. Look for (84-bytes)
   a. free: no good
   b. 0x134: good
2. Split 972 block:
   a. block A: size 84
   b. block B: size 888
3. **Return** the address of **second word** of block A



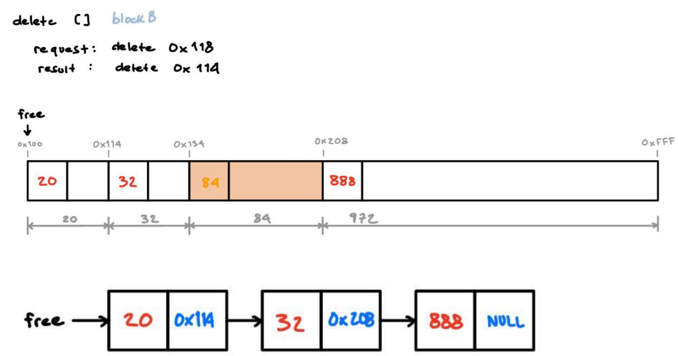
<table>
<tr>
<td>

delete *blockB*

- **Request:** delete 0x118
- **Delete:** 0x114

Add (32, 0x114) to free List **(between 0x100 & 0x208)**

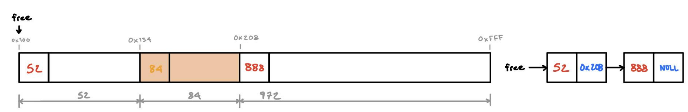
</td>
<td>



</td>
</tr>
<tr>
<td>

Combine adjacent free blocks:

</td>
<td>

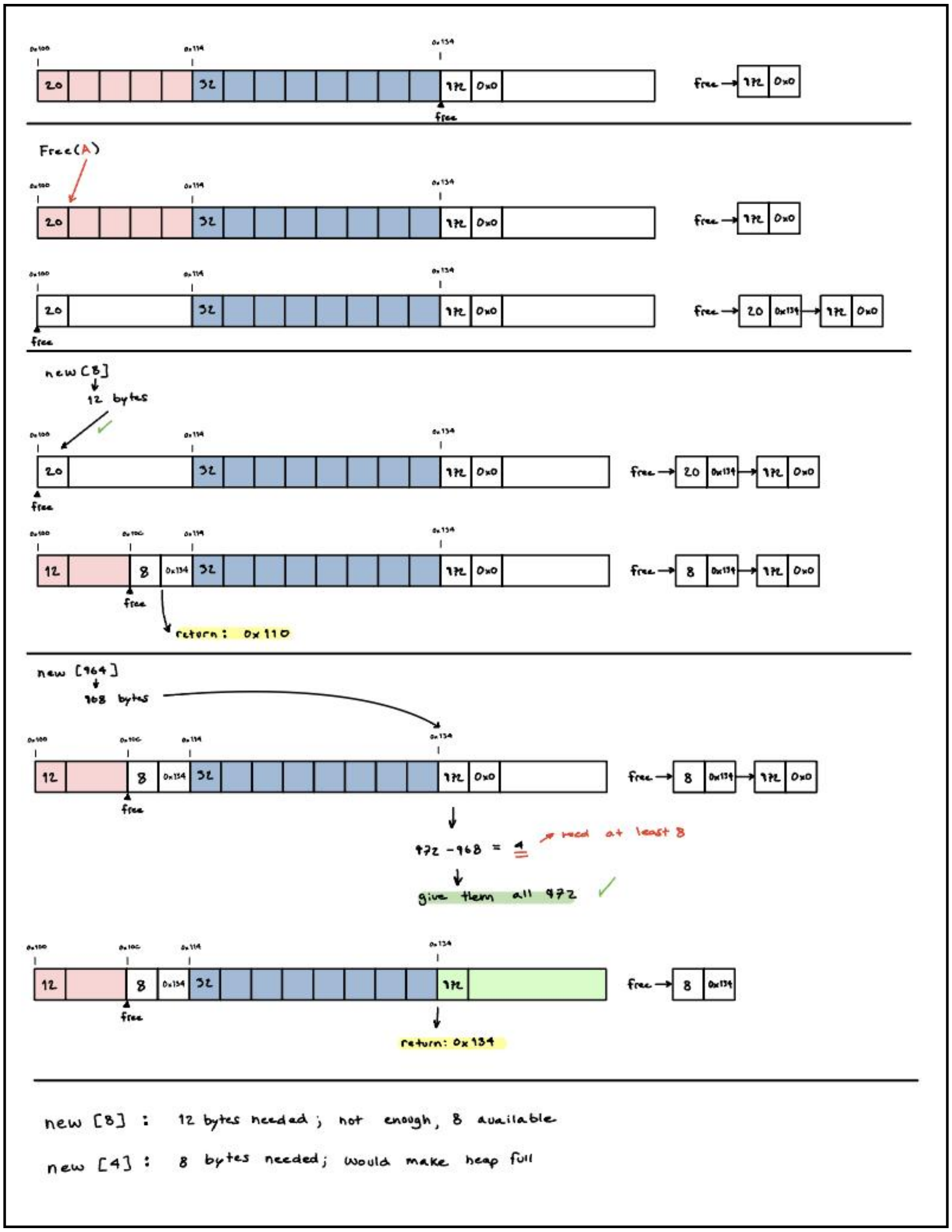


</td>
</tr>
</table>

Edge Cases

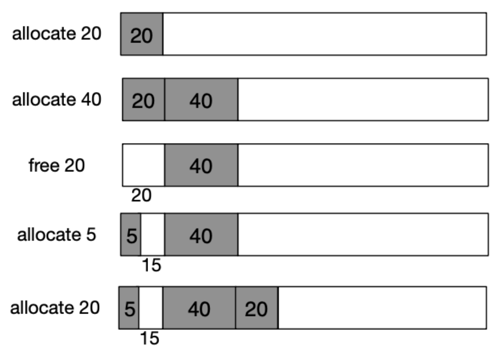
1. **32 bytes free. User requests 24 bytes:**
    - 28 bytes are needed
    - The free block gets split into 2 blocks: 28 bytes & 4 bytes
    - A free block which is 4 bytes is not big enough to store its size as well as the address of the next free block. Need at least 8 bytes.
    - **solution:** allocate the entire block of 32 instead of splitting it
2. **Not enough memory available**
    a. return NULL
3. **Heap becomes full**
    a. What would happen to the ***free*** pointer?

Example 2



Issue - Fragmentation

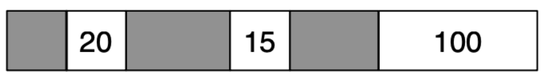
- Caused by repeated allocation/deallocation
- Even though the required bytes of memory are available, they are not available in a contiguous block and therefore cannot be allocated



The last request to allocate 20 bytes of memory. There is a block of 15 bytes available, but it is not enough, so a bigger free block is broken to allocate the 20 bytes.

Solution

Don't always choose the 1st block of RAM big enough to satisfy an allocation request



Allocate 10 bytes

First Fit	Find the **first** block that works	20-byte block ⇒ leaving a hole of 10
Best Fit	Find the block that **best** fits Leaves **smallest**/**no** hole behind • prone to creating the smallest and least reusable holes	15-byte block ⇒ leaving a hole of 5
Worst Fit	Find the **biggest** block	100-byte block ⇒ leaving a 90-byte block

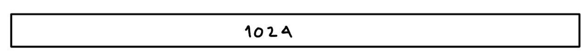
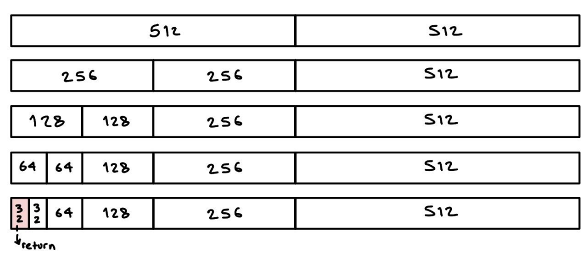
Binary Buddy System

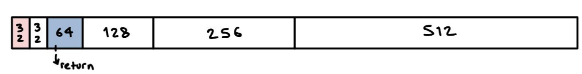
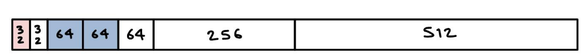
Given: n byte request

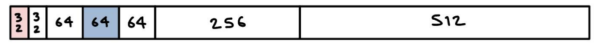
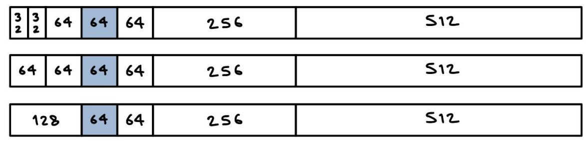
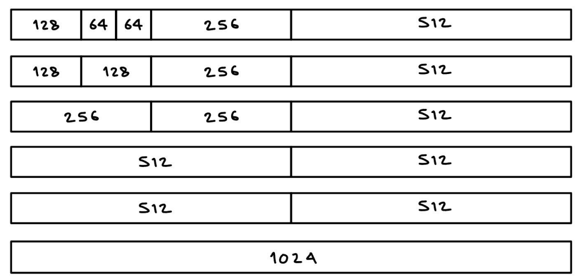
1. Look for smallest block of size **2^k** >= n
2. If not found, split next smallest into “**buddies**” of the same size
    a. 1 buddy: is **allocated**
    b. 1 buddy: **placed in the free list**
3. Repeat until desired size is found

Given: ‘free’

1. add to free list
2. combine buddies until no more are next to each other

<table>
<tr><th colspan="2">Example</th></tr>
<tr><td>heap is 1024 bytes as before.</td><td>
</td></tr>
<tr><td>20 Byte Request
• allocator needs 24 bytes
• Look for: 2^5 = 32 bytes</td><td>
</td></tr>
<tr><td>40 byte request
• allocator needs 44 bytes
• Look forL: 2^6 = 64 bytes

1. Find first available
2. Returns starting address of the 2nd word</td><td>
</td></tr>
<tr><td>50 Byte Request:
• allocator needs: 54 bytes
• look for: 64 bytes</td><td>
</td></tr>
</table>

1. Not found: split into buddies 2. Allocate 1, put the other in free-list	
Free the first 64-byte block: - add to free list	
Free the 32-byte block: 1. Add to the free list 2. Combine free buddies	
Free the 64-byte block: 1. Add to the free list 2. Combine free buddies	

Bookkeeping

Each block is assigned a code (1st word)

- code(largest block) = 1
  - buddy 1: add a 0
  - buddy 2: add a 1

Find the buddy of a block	flip last bit
Size of a block	(heap-size) / 2^{n-1} (where n = # bits in code)

Allocation	Deallocation
Search the free list for an appropriate sized block (based on codes)	Search buddy in the free list (flip the last bit). Merge if found.

ie: 32-bytes • $(1024/2^{6-1} = 32)$ • look for a **6** digit code • Not found: make do with a 5-digit code	• code(newly merged block) = chop off last bit

Downside

internal fragmentation: by insisting to allocate blocks of a certain size, extra memory is allocated/wasted

Implicit Memory Management: Garbage Collection

Garbage Collection

- automatically deallocate memory once that memory is no longer needed

Methods

Each one has a different method of determining: “when is the memory no longer needed?”

Method	How	Limitation
Reference Counting	Keep track of: # pointers that point to each block • **deallocate:** count = 0	**Circular references** • a block of memory that refers to another block of memory which refers back to the first block • Not referenced anywhere else in the program. Since their reference counts are not zero, they would never be deallocated

<table>
<tr><td>Mark & Sweep</td><td>Mark
Mark blocks of the heap reachable from pointers in: stack + global variables
follow blocks that contain pointers to discover new parts of the heap that are also reachable
sweep phase
deallocate all marked blocks</td><td>Have to stop the program:
cannot have the program making any changes to the memory</td></tr>
<tr><td>Copying Collector</td><td>Splits the heap in 2: from & to
allocated from: from part of the heap
When from is full: copies the reachable parts → to
The roles of from and to are reversed</td><td>Pro:
since memory is copied between halves, it can be laid out to avoid fragmentation
Cons:
program needs to halt
Halves the amount of heap memory available</td></tr>
</table>

Generational Garbage Collection

Copying collectors: works well when few objects survive collection

mark-and-sweep: works well when most objects survive collection

Most objects die young:

- objects are split into **generations**:

- new objects are allocated in the **youngest generation** and **collected** through copying
- Objects that survive these collections are moved to an **older generation** which uses **mark-and-sweep**
- **younger generations** collected more frequently than **older generations.**