CC Lecture 13

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Heap Storage Allocation

- This strategy involves the reserving of a large contiguous block of memory commonly called the **heap**.
- **Heap** is used for allocating space for objects created at run time.

  e.g. nodes of dynamic data structures like linked lists & trees.
- Dynamic memory allocation and deallocation are based on the requirements of the program
  - C : manual: using malloc and free
  - C++: manual: using new and delete
  - Java: semi-automatic: using new and garbage collection
  - Lisp: automatically by runtime system

Memory Manager

- **Heap Memory Manager** manages heap memory by implementing the mechanisms for allocation and deallocation.
- **Goals**
  - space efficiency to minimize fragmentation
  - program efficiency by taking advantage of locality of objects in memory and make the program run faster
  - low overhead by efficient allocation and deallocation
- Heap is maintained either as a **doubly linked list** or as **bins** of free memory chunks.

Allocation and Deallocation

- Initially, the heap is **one large and contiguous** block of memory.
- As **allocation** requests are satisfied, chunks are cut off from this block and given to the program.
- As **deallocations** are made, chunks are returned to the heap and are free to be allocated again (holes).
- After a number of allocations and deallocations, memory becomes **fragmented** and is not contiguous.

Allocation and Deallocation

- Allocation from a fragmented heap may be made either in a **first-fit** or **best-fit** manner.

- After a deallocation, we try to **coalesce** (join together) contiguous holes and make a bigger hole (free chunk)

First-Fit and Best-Fit Allocation Strategies

- The **first-fit** strategy picks the **first** available chunk that satisfies the allocation request.
- The **best-fit** strategy searches and picks the **smallest** (**best**) possible chunk that satisfies the allocation request.

- Both strategies chop off a block of the required size from the chosen chunk, and return it to the program.
- And the rest ***remains*** in the heap.

- **Best-fit** strategy has been shown to reduce fragmentation in practice, better than first-fit strategy.

Next-fit strategy

- **Next-fit strategy** tries to allocate the object in the chunk that has been ***split recently***
  - Tends to improve speed of allocation
  - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times

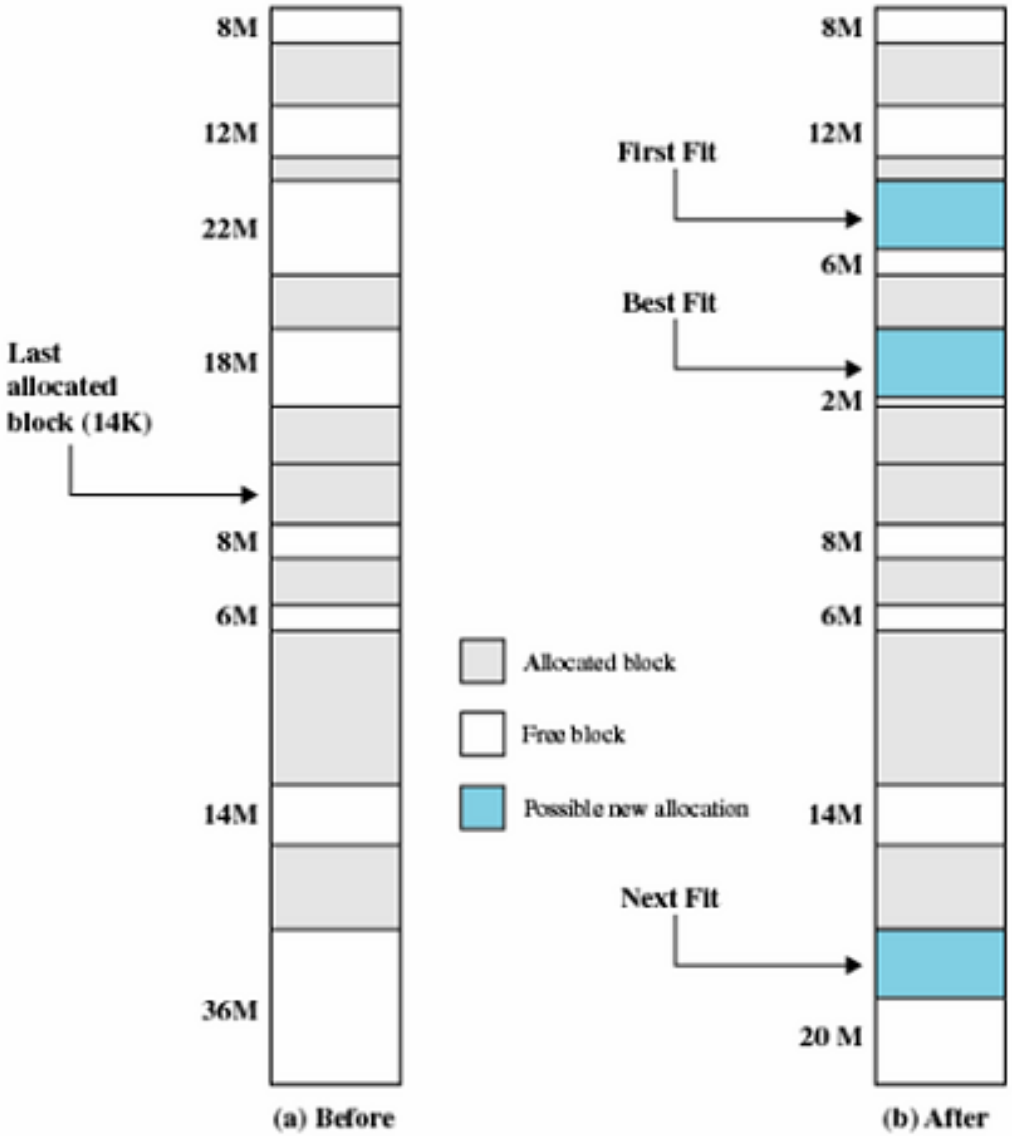
    (cache behaviour may be better)

- The overall the speed of the programming increases; when **doubly linked list** approach for storing/managing heaps is used.

Summary

- **Best-Fit**:
  - Closest in size to the request .
- **First-Fit**:
  - Scans the main memory from the beginning and first available block that is large enough .
- **Next-Fit**:
  - Scans the memory from the location of last placement and chooses next available block that is large enough.

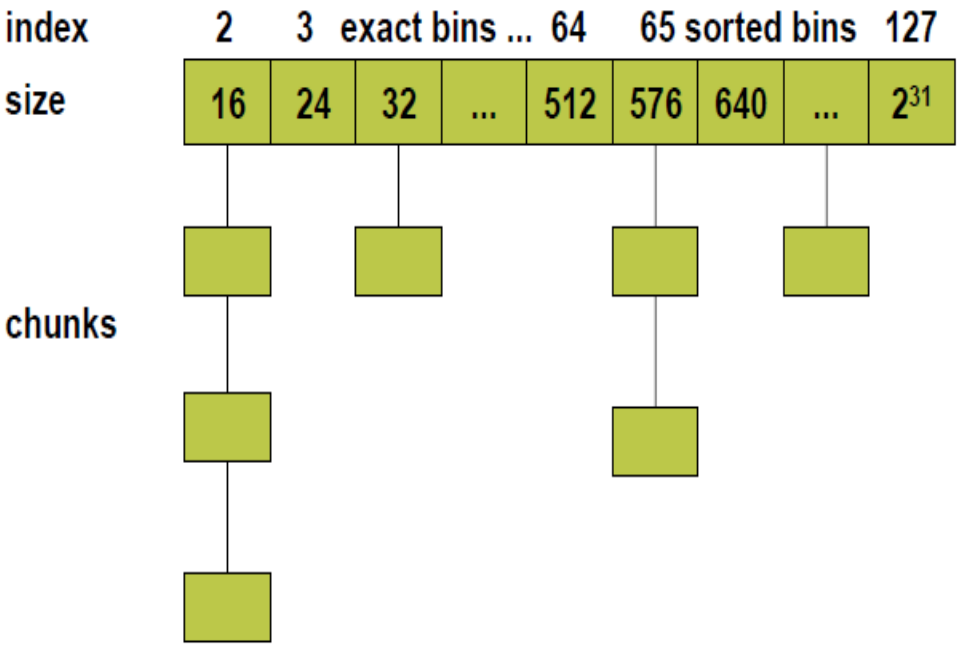
Example: Allocation of 16 MB block using three placement algorithms



Bin-based Heap

- Free space is organized into **bins** according to their sizes
  - Lea Memory Manager in GCC
- More bins for smaller sizes, as there are more small objects
- A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
  - Chunks are all of the **same** size
- Then there are bins of approximately logarithmically (double previous) size
  - Chunks are **ordered by size**
- The **last** chunk in the **last** bin is the **wilderness chunk**, which gets us a chunk by going to the *operating system*

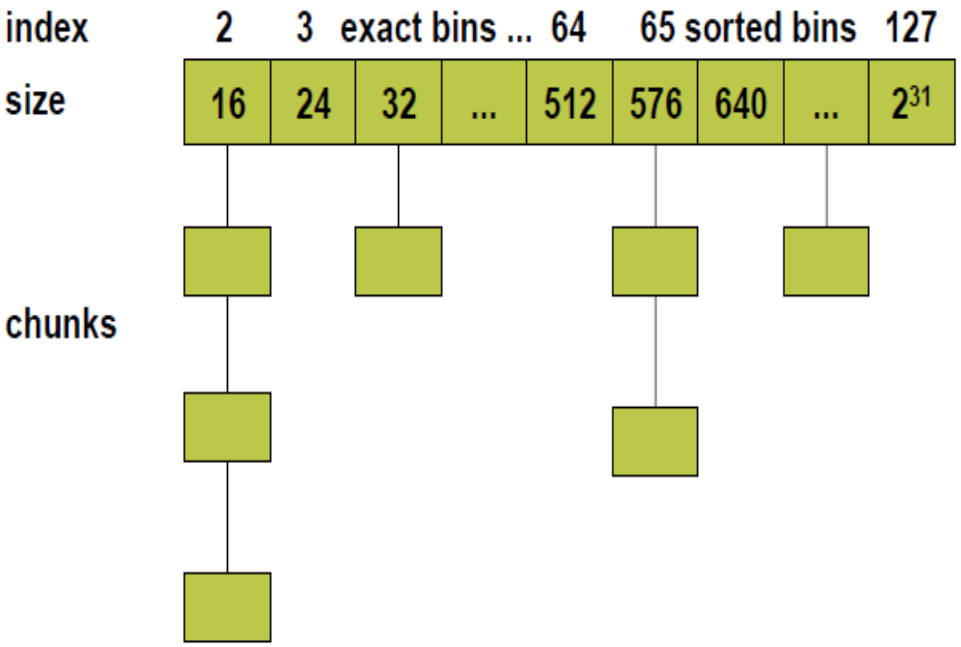
Example (Ref: From Lea's article on memory manager in GCC)



- Available chunks are maintained in bins, grouped by size.
- There are 128 fixed-width bins, approx. double the size of previous.
- Bins for sizes less than 512 bytes each hold only exactly one size (spaced 8 bytes apart).
- Searches for available chunks are processed in smallest-first, ***best-fit*** order.

http://gee.cs.oswego.edu/dl/html/malloc.html

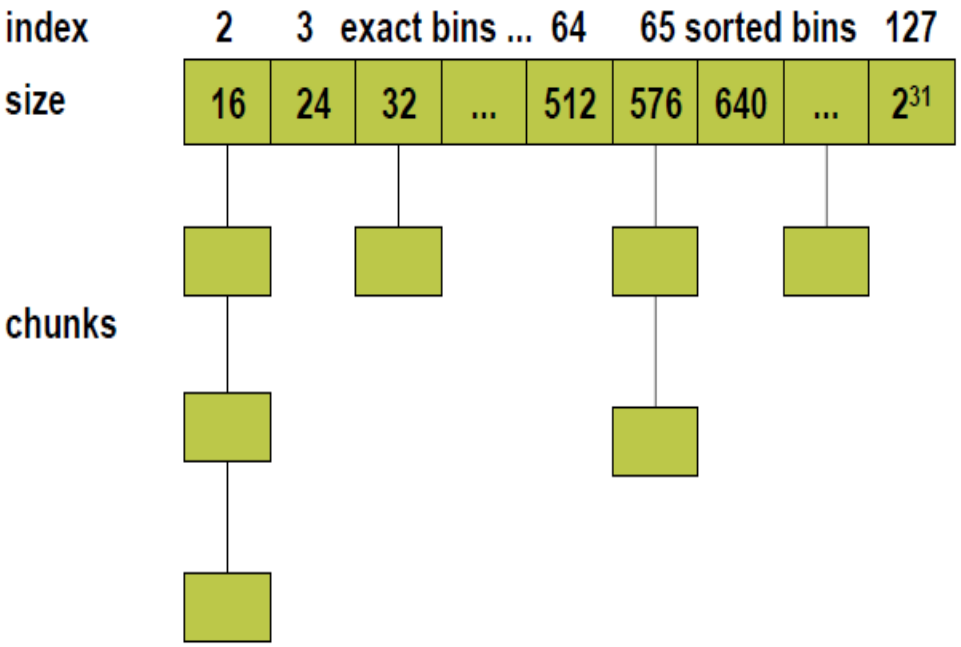
Example (Ref: From Lea's article on memory manager in GCC)



- here is a list of x bytes chunks.
- Head/index of the linked list is maintained.
- Suppose we need 24 bytes, it is empty then we take from the next i.e.32 bytes.
- In the list at index 576, the chunks can be of different sizes but they will be less than the sizes in the next list.
- These are ordered by size in increasing order.

http://gee.cs.oswego.edu/dl/html/malloc.html

Example (Ref: From Lea's article on memory manager in GCC)



- If programmer has used up the entire heap, i.e. there are no chunks available then automatic call to the operating system is made to release more memory to the heap of this particular program.

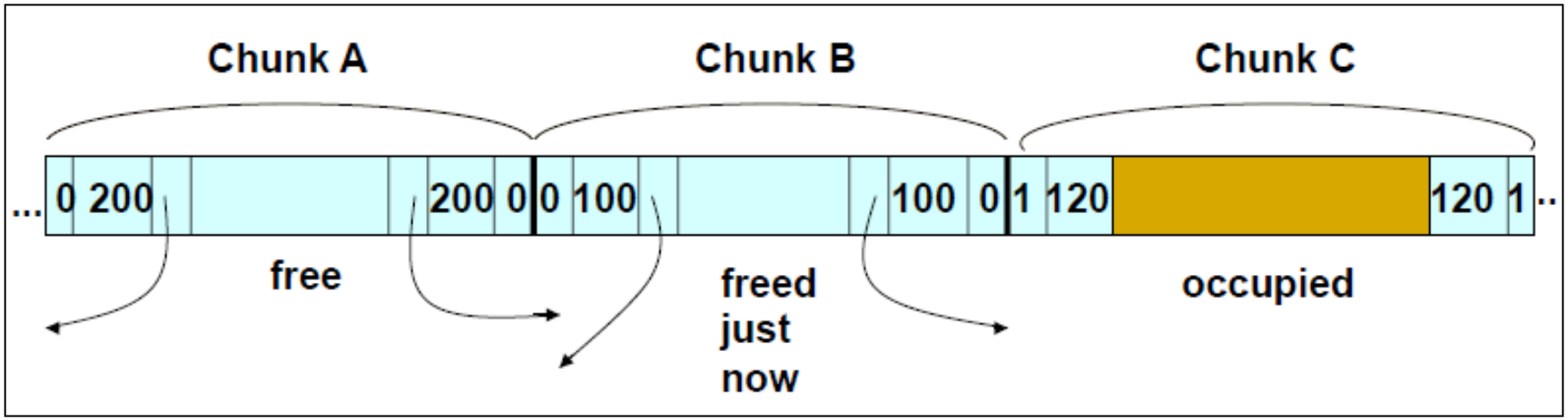
http://gee.cs.oswego.edu/dl/html/malloc.html

Managing and Coalescing Free Space

- Fragmentation should be **reduce** by coalescing adjacent chunks
- Many small chunks together cannot hold one large object
- In the Lea memory manager, there is no coalescing in the exact size bins, only in the sorted bins.

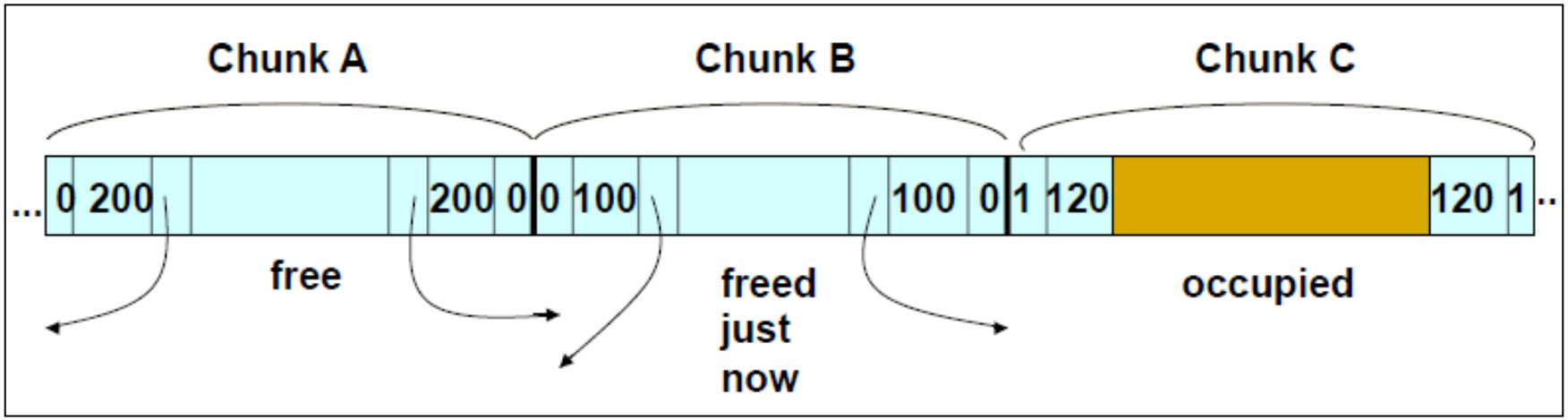
- **Boundary tags** (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
- A **doubly linked list** of free chunks is maintained

Boundary Tags



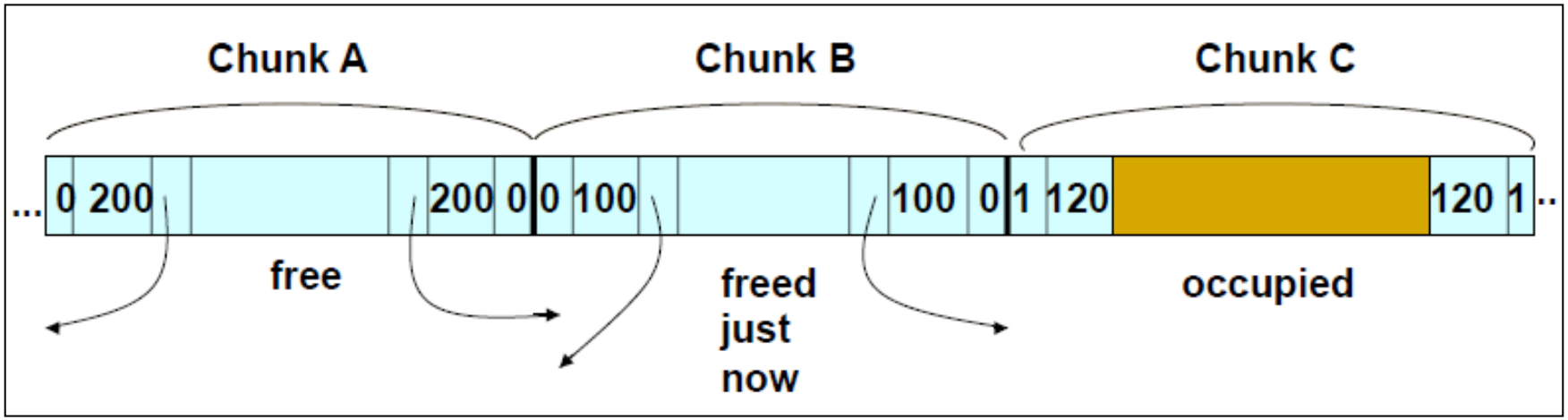
- Chunk A is free chunk B has been freed just now
- The address of chunk B starts just after the address of chunk A.
- So, these 2 are actually contiguous since B has been released just now; it is possible to combine these 2 chunks and make a bigger chunk.

Boundary Tags



- On both ends of A, 0 indicated memory is free, 200 indicates its size
- On both ends of B, 0 indicates memory is free, 100 indicates its size
- On both ends of C, 1 indicates memory is occupied, 120 indicates its size

Boundary Tags

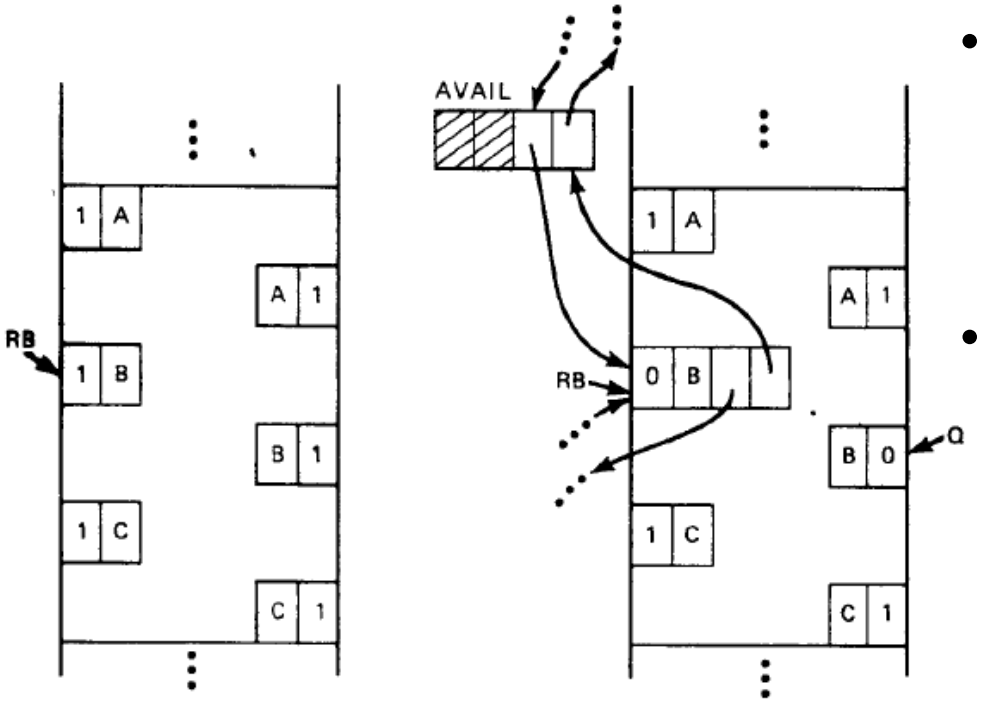


- As A and B are adjacent, they can be combined into one chunk
- The new size will be slightly more than 300
- Merged chunk AB might have to be placed in different bin (if Lea's memory manager is used)

Four different cases to place the freed block

1. Place the freed block (no free neighbor to collapse with) in the front of the availability list
2. Collapse with a free neighbor(previous) and place in front of availability list
3. Collapse with a free neighbor(next) and place in front of availability list
4. Collapse with two free neighbors (previous and next) and place in front of availability list

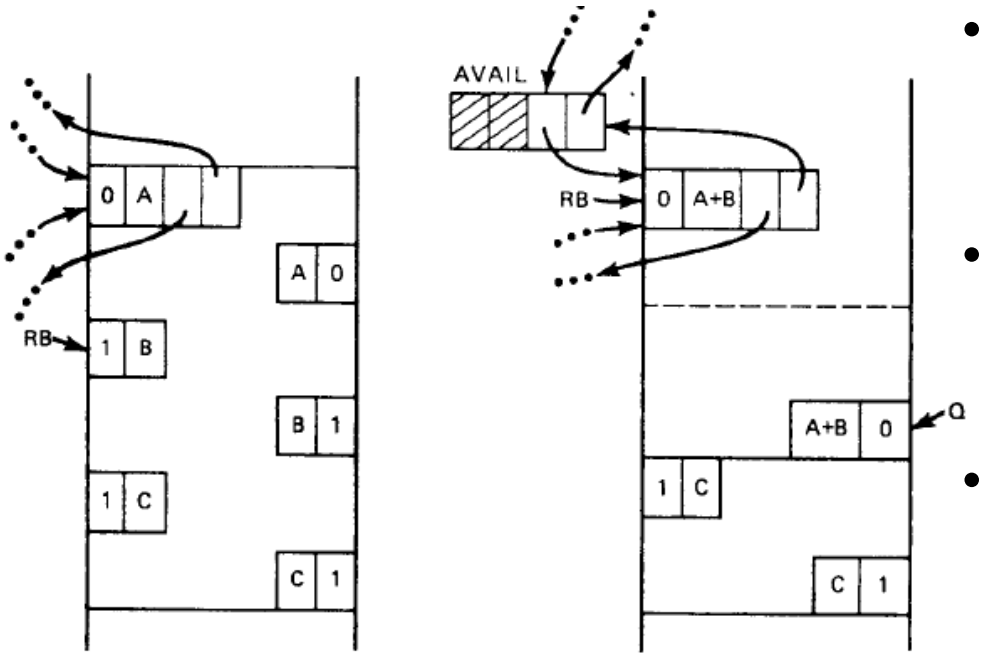
Case 1



RB = address of block that is free
AVAIL = head of availability list
Q = pointer that denotes the predecessor of the node being freed

- The freed block pointed to by RB is of size B and has no immediate neighbours that are already marked as free.
- That is, both the block with a size field of A and the block with the size field of C are already allocated and hence their flag fields are marked with a 1.
- In this case, the free block is simply placed at the front of the availability list

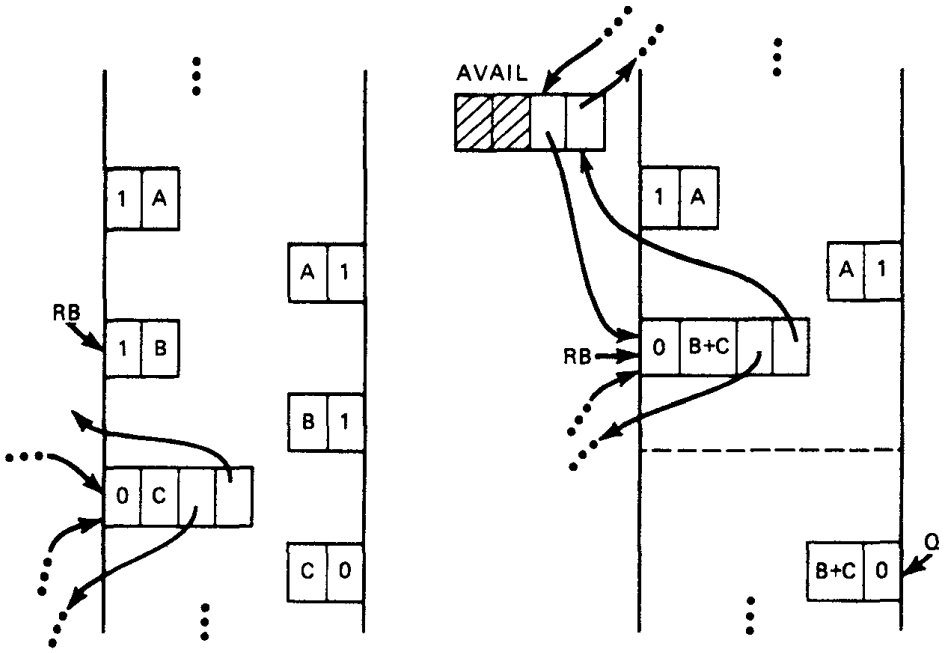
Case 2



RB = address of block that is free
AVAIL = head of availability list
Q = pointer that denotes the predecessor of the node being freed

- The neighbour block of size A is shown to be on the availability list.
- Collapse the block pointed to by RB and its neighbour of size A to a new block of size A + B.
- In collapsing the blocks, the block of size A is removed from the availability list.
- Insert the new collapsed block at the front of the availability list.

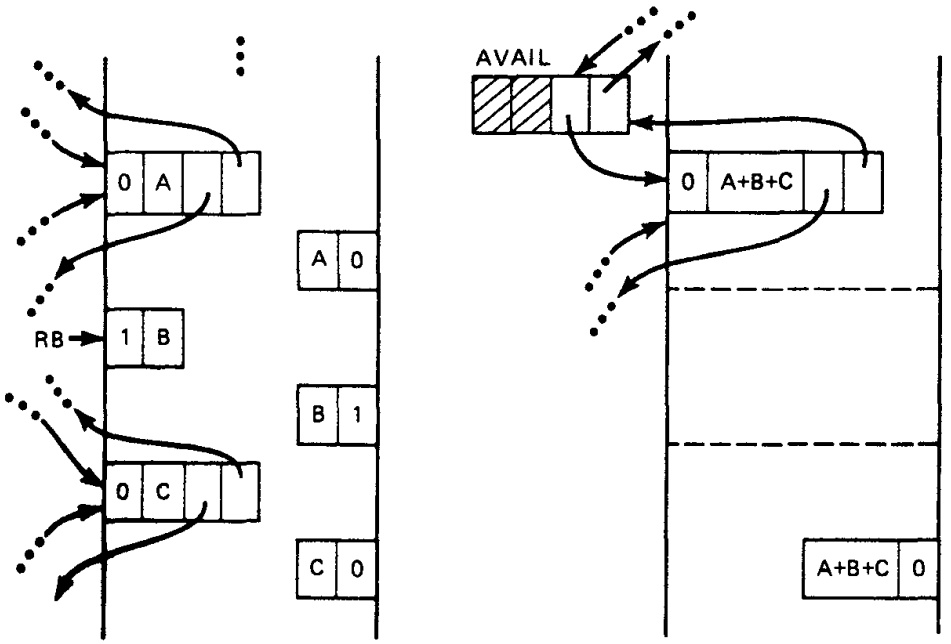
Case 3



RB = address of block that is free
AVAIL = head of availability list
Q = pointer that denotes the predecessor of the node being freed

- Here, the freed block is collapsed with a neighbour that succeeds it in heap storage.
- The freed block is collapsed with the block of length C, and the block of length C is removed from the free list.
- The final step places the collapsed block at the head of the availability list.

Case 4



- The fourth case combines the effects of cases 2 and 3 discussed previously.
- Both neighbour blocks are collapsed, and the large collapsed block is again inserted as the first block in the availability list.

RB = address of block that is free
AVAIL = head of availability list

Issues in manual deallocation

- Memory leaks
- Dangling pointer dereferencing

Issue : memory leaks
(losing the pointer to the allocated memory)

```
#include <stdlib.h>
void function_which_allocates(void) {
    /* allocate an array of 45 floats */
    float *a = malloc(sizeof(float) * 45);
    /* additional code making use of 'a' */
    /*return to main, having forgotten to free the memory we malloc'd */
}
int main(void) {
    function_which_allocates();
    /* the pointer 'a' no longer exists, and therefore cannot be freed, but
    the memory is still allocated. a leak has occurred. */
}
```

Dangling pointer

```
{
    char *dp = NULL;
    /* ... */
    {
            char c;
            dp = &c;
    }
     /* c falls out of scope */
    /* dp is now a dangling pointer
     */
}
```

- Deleting an object from memory explicitly or by destroying the stack frame on return does not alter associated pointers.
- The pointer still points to the same location in memory (even though it may now be used for other purposes).

Dangling pointer

```
#include <stdlib.h>
void func() {
    char *dp = malloc(…);
    /* … */
    free(dp);
    /* dp now becomes a dangling
    pointer */
    dp = NULL;
    /* dp is no longer dangling */
    /* … */
}
```

- A pointer becomes dangling when the block of memory it points to is freed.
- One way to avoid this is to make sure to reset the pointer to null after freeing its reference.

Issues in manual deallocation

- **Memory leaks**
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- **Dangling pointer dereferencing**
  - Referencing deleted data
- Both are serious and hard to debug
- Solution???

Stack vs. Heap memory allocation

PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible
Data type structure	Linear	Hierarchical

https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation

Example

(ex. From Aho-Ullman book)

- Suppose the heap consists of **seven chunks**, starting at address 0. The sizes of the chunks, in order, are **80, 30, 60, 50, 70, 20, 40 bytes**.
- When we place an object in a chunk, we put it at the high end if there is enough space remaining to form a smaller chunk (so that the smaller chunk can easily remain on the linked list of free space).
- However, we cannot tolerate chunks of fewer that 8 bytes, so if an object is almost as large as the selected chunk, we give it the entire chunk and place the object at the low end of the chunk.
- If we request space for objects of the following sizes: **32, 64, 48, 16,** in that order, what does the free space list look like after satisfying the requests, if the method of selecting chunks is

  (a) First Fit. (b) Best Fit.