CSCE 221-201, Fall 2019

Applying Lists to Memory Management

Sources: T. A. Standish, *Data Structures in Java*, Addison-Wesley, 1998; R. Decker, *Data Structures*, Prentice-Hall, 1989.

The Heap

When you use `new` or `malloc` to dynamically allocate some space, the run-time system handles the mechanics of actually finding the required free space of the necessary size.

When you make an object inaccessible (in Java) or use `free` (in C) or `delete` (in C++), again the run-time system handles the mechanics of reclaiming the space.

We are now going to look at HOW one could implement dynamic allocation of objects from the heap. The reasons are:

- Basic understanding.
- Techniques are useful in other applications.
- Not all languages provide dynamic allocation, including assembler. You can use these ideas to "simulate" it.

What is the Heap?

The **heap** is an area of memory used to store objects that will by dynamically allocated and deallocated.

Memory can be viewed as one long array of memory locations, where the address of a memory location is the index of the location in the array.

Thus we can view the heap as a long array of bytes.

Contiguous locations in the heap (array) are grouped together into blocks. Blocks can be *different sizes*.

When a request arrives to allocate n bytes, the system

- finds an available block of size at least n,
- allocates the n bytes requested from that block, and
- returns the address of the starting byte allocated.

Blocks are classified as either **free** or **allocated**.

Initially, the heap consists of a single, free, block containing the entire array.

Heap Data Structures

Once blocks are allocated, the heap might get chopped up into alternating allocated and free blocks of varying sizes.

We need a way to locate all the free blocks.

This will be done by keeping the free blocks in a *linked list*, called the **free list**.

The linked list is implemented using *explicit array indices* as the "pointers".

Each block has some **header** information, which includes the size of the block and any required "pointers". (For simplicity, we will ignore the space required for the header.)

Allocation

When a request arrives to allocate n bytes, scan the free list looking for a block that is big enough.

There are two strategies for choosing the block to use:

- **first fit**: stop searching as soon as you find a block that is big enough, OR
- **best fit**: find the smallest block that is big enough. If you find a block that is exactly the required size, you can stop then. If no block is exactly the required size, then you have to search the whole free list to find the smallest one that is big enough.

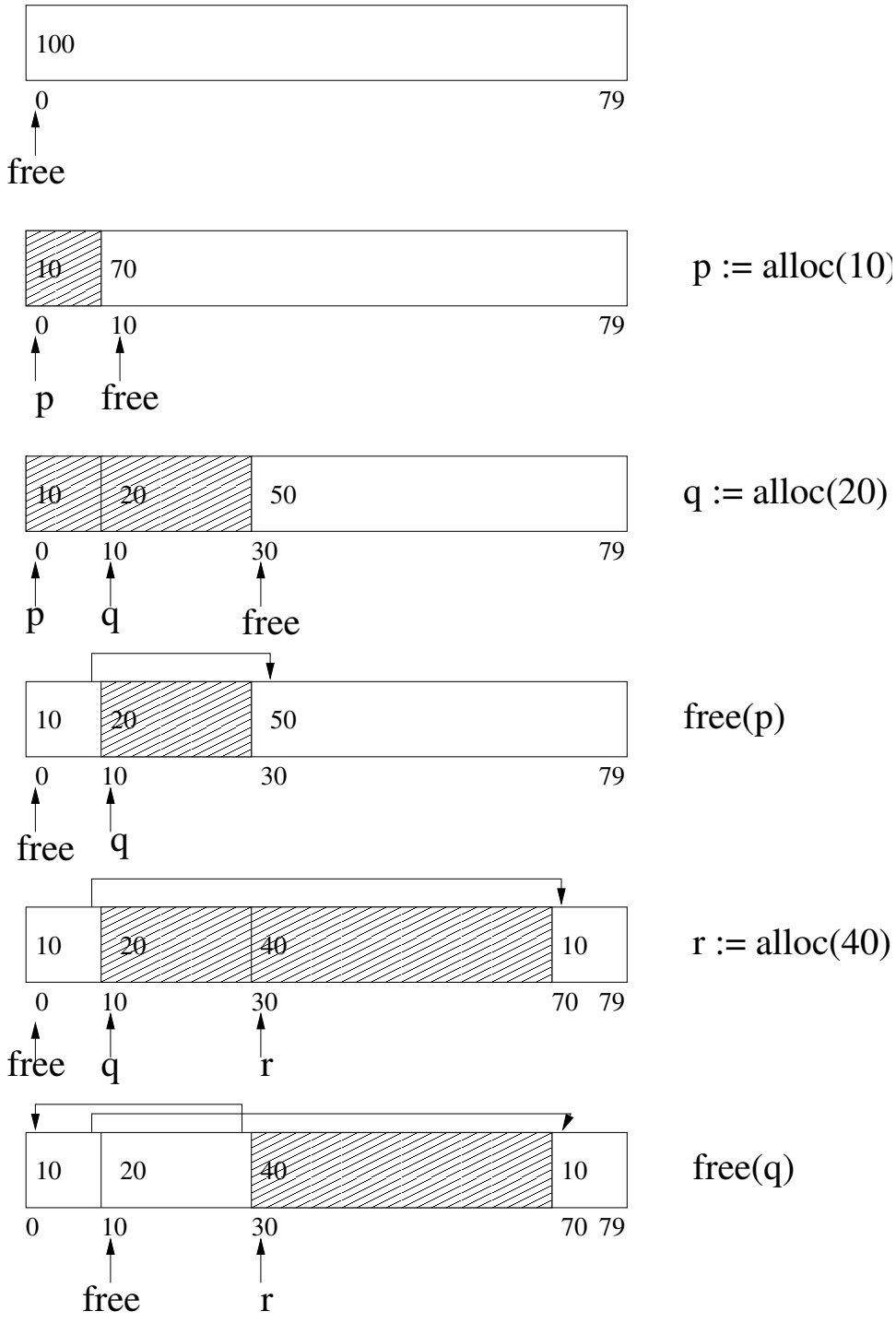
If the block found is bigger than n, then break it up into two blocks, one of size n, which will be allocated, and a new, smaller, free block. The new, smaller, free block will replace the original block in the free list.

If the block found is exactly of size n, then remove it from the free list.

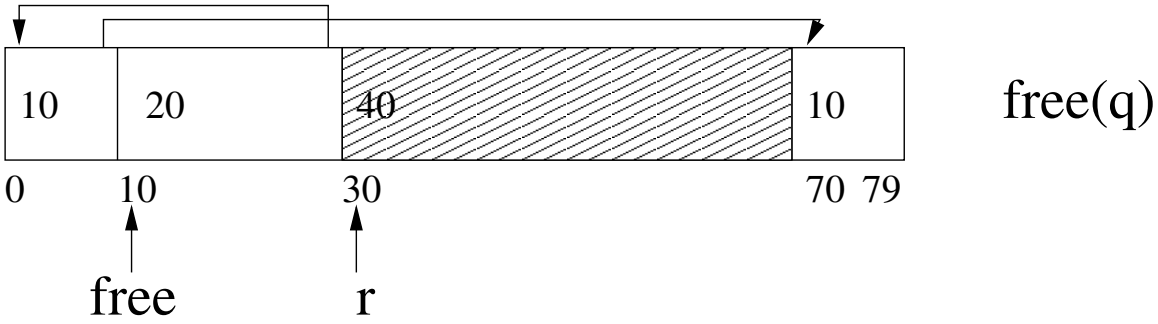
If no block large enough is found, then can't allocate.

Deallocation

When a block is deallocated, as a first cut, simply insert the block at the front of the free list.



Fragmentation



Problem with previous example: If a request comes in for 30 bytes, the system will check the free list, and find a block of size 20, then a block of size 10, and finally a block of size 10. None of the blocks is big enough and the allocation will fail.

But this is silly! Clearly there is enough free space, in fact there are 30 contiguous free bytes! The problem is that the space has been artificially divided into separate blocks due to the past history of how it was allocated.

This phenomenon is called **fragmentation**.

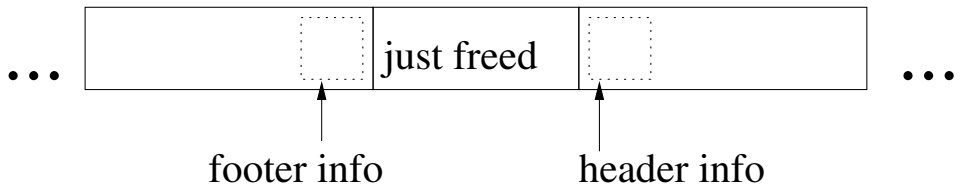
Coalescing

A solution to fragmentation is to **coalesce** deallocated blocks with free (physical) neighbors. Be careful about the use of the word neighbor:

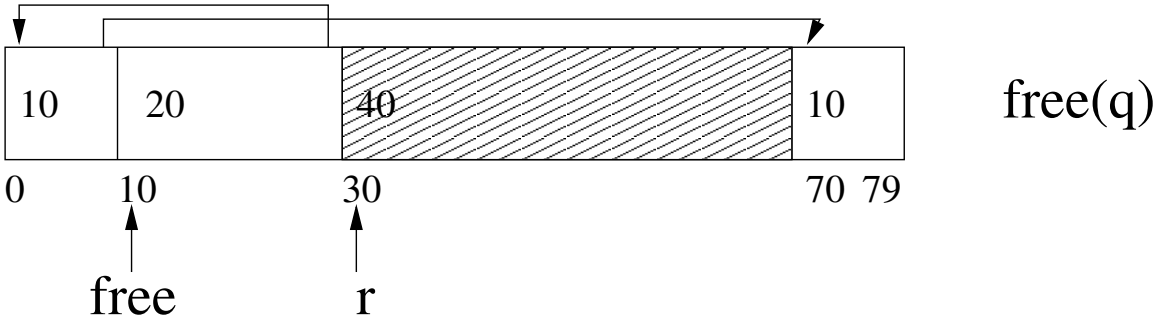
- **physical neighbor:** actual physical space is adjacent
- **virtual neighbor:** blocks are adjacent in the free list, but not necessarily in memory.

To facilitate this operation, we will need additional space overhead in the header, and it will also help to keep "footer" information at the end of each block to:

- make the free list doubly linked, instead of singly linked
- indicate whether the block is free or not
- replicate some info in the footer so that the status of the *physical* neighbors of a newly deallocated node can be efficiently determined



More Insidious Fragmentation



However, coalescing will not accommodate a request for 40 bytes. There are 40 free bytes, but they are not physically contiguous.

The problem is that two of the free blocks are interrupted by the allocated block r.

This is a serious problem with allocation schemes, when the sizes requested can be arbitrary.

Large free blocks keep getting chopped up into smaller and smaller blocks, so it gets harder to satisfy large requests, even if there is enough total space available.

Compaction

The solution to this problem is called **compaction**. The concept is simple: move all the allocated blocks together at the beginning of the heap, and compact all the unallocated blocks together into a single large free block.

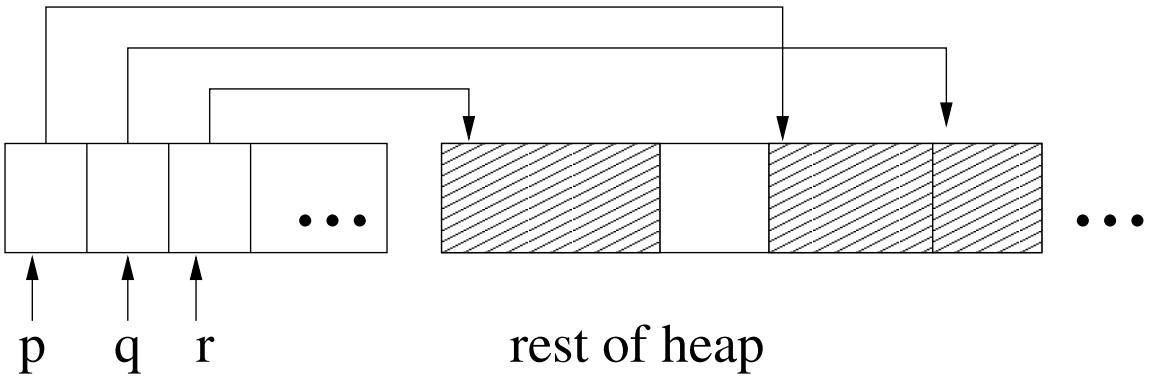
The difficulty though is that if you "move" a block, i.e., copy the information in a block to another location in the heap, you change its address. And you already gave out the original address to the user when the block was allocated!

Master Pointers

A solution is to use **double indirection**, with **master pointers**.

- A special area of the heap contains "master" pointers, which point to (hold the address of) allocated blocks.
- The addresses of the master pointers never change — they are in a fixed part of the heap.
- The address returned by the allocate procedure is the address of the master pointer.
- The *contents* of a master pointer can change, so that when the block being pointed to by a master pointer is moved as part of a compaction, the address is updated in the master pointer.
- But the user, who received the master pointer address, is unaffected.

Master Pointers (cont'd)



Costs:

- Additional space for the master pointers
- Additional time: have to do two pointer dereferences, instead of just one
- Unpredictable "freezing" of execution for a significant period of time, when the compaction occurs. It's hard to predict when compaction will be needed; while it is going on, the application has to pause; and it can take quite a while if the memory is large.

But there really isn't any feasible alternative, if you want to do compaction.

1 Garbage Collection

The above discussion of deallocation assumes the memory allocation algorithm is somehow informed about which blocks are no longer in use:

- In C and C++, this is done by the programmer, using `free` and `delete`.
- In Java, the run-time system does this automatically.

This process is part of **garbage collection:**

- identifying inaccessible memory
- management of the free list to reduce the effects of fragmentation

One of the challenging aspects of garbage collection is how to correctly identify inaccessible space, especially how to do it incrementally, so the application does not suddenly pause while it's happening.

There are many interesting algorithms for doing garbage collection with different performance tradeoffs.