Heap-Based Allocation

A **heap** is a region of storage in which sub-blocks can be **allocated** and **deallocated** at arbitrary times.

Heaps are required for the **dynamically allocated** pieces of **linked data structures**, and for all the **objects** such as fully general character strings, **lists**, and **sets**, whose **size** may change as a result of an assignment statement or other update operation.

The principal concerns for heaps are **speed** and **space**

Space concerns can be further subdivided into issues of **internal fragmentation** and **external fragmentation**.

Internal fragmentation occurs when a **storage-management** algorithm allocates a block that is larger than required to hold a given object; the extra space is unused. Internal fragmentation occurs when a storage-management algorithm allocates blocks that is larger than required to hold a given object; the extra space is then unused.

External fragmentation occurs when the blocks that have been assigned to active objects are scattered through the **heap** in such a way that the remaining, unsued space is composed of multiple blocks: there may be a lot of free space but no one piece of space may be large enough to hold future data.

Many storage-management algorithms maintain a single linked list— the **free list**—of heap blocks not currently in use. Initially the list consists of a single block comprising the entire heap. At each allocation request the algorithm searches the list for a block of appropriate size. With a **first fit**algorithm we select the first block on the list that is large enough to satisfy the request. With a **best fit** algorithm we search the entire list to find the smallest block that is large enough to satisfy the request. In either case, if the chosen block is significantly larger than required, then we divide it into two and return the unneded portion to the free list as a smaller block. When a block is deallocated and returned to the free list, we check to see whether either or both of the physically adjacent blocks are free; if so, we coalesce them.

One would expect a *best fit* algorithm to do a better job of reserving large blocks for large requests. At the same time, it has higher allocation cost than a *first fit* algorithm, because it must always search the entire list, and it tends to result in a larger number of very small "left-over" blocks. Which approach—results in lower external fragmentation depends on the distribution of size requests.

In any algorithm that maintains a single free list, the cost of allocation is linear in the number of free blocks. To reduce this cost to a constant, some storage management algorithms maintain separate free lists for blocks of different sizes. Each request is rounded up to the next standard size (at the cost of internal fragmentation) and allocated from the appropriate list. In effect the heap is divided into "pools", one for each standard size. The division may be static or dynamic. Two common mechanisms for dynamic pool adjustment are known as the **buddy system** and the **Fibonacci heap**. In the buddy system, the standard block sizes are powers of two. If a block of size 2^k is needed, but none is available, a block of size 2^{k+1} is split in two. One of the halves is used to satisfy the request; the other is placed on the kth free list. When a block is deallocated, it is coalesced with its "buddy" —the other half of the split that created it—if that buddy is free. Fibonacci heaps are similar, but use Fibonacci numbers for the standard sizes, instead of powers of two. The algorithm is slightly more complex, but leads to slightly lower internal fragmentation, because the Fibonacci sequence grows more slowly than 2^k.

The problem with external fragmentation is that the ability of the heap to satisfy requests may degrade over time. Multiple free lists may help, but clustering small blocks in relatively close physical proximity, but they do not eliminate the problem. It is always possible to devise a sequence of request that cannot be satisfied, even though the total space required is less than the size of the heap. If memory is partitioned among size pools statically, one need only exceed the maximum number of request of a given size.s If pools are dynamically re-adjusted, one can "checkerboard" the heap by allocating a large number of small blocks and then deallocating every other one, in order of physical address, leaving an alternating pattern of small free and allocated blocks. To eliminate external fragmentation, we must be prepared to **compact** the heap, by moving already-allocated blocks. This task is complicated by the need to find and update all outstanding references to a block that is being moved.