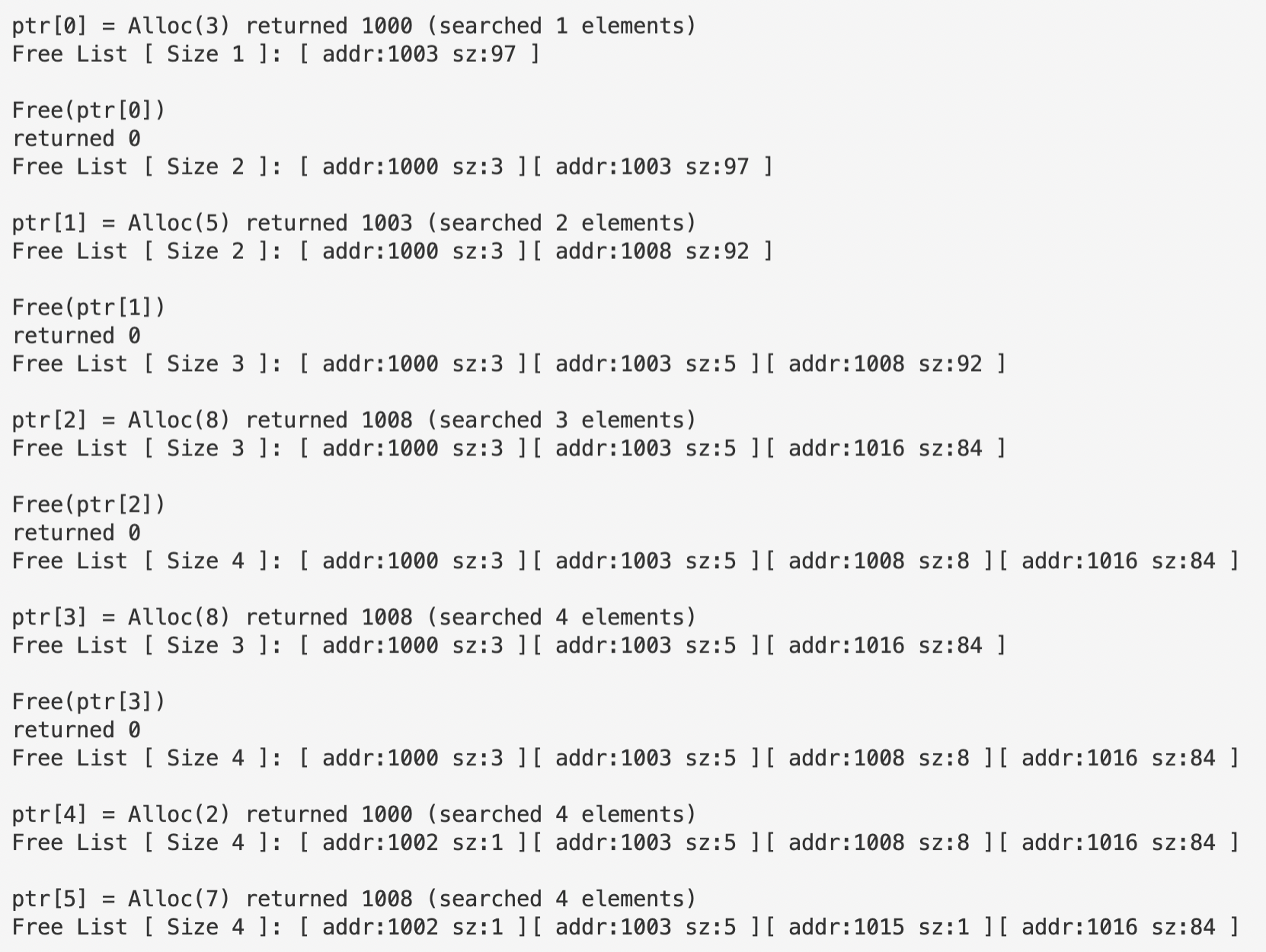
**Cpt 17**

**Homework (Simulation)**

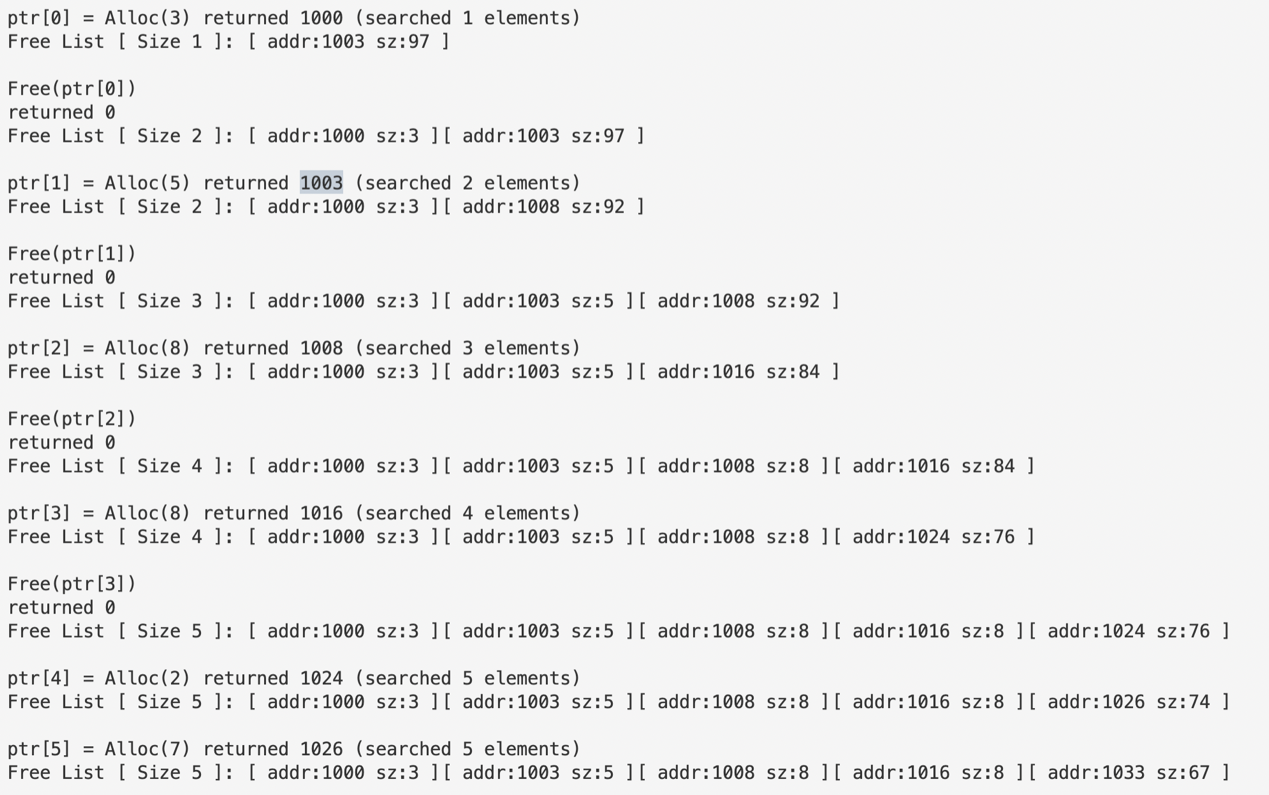
1. **First run with the flags -n 10 -H 0 -p BEST -s 0 to generate a few random allocations and frees. Can you predict what alloc() / free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?**

Run command python3 malloc.py -n 10 -H 0 -p BEST -s 0 -c



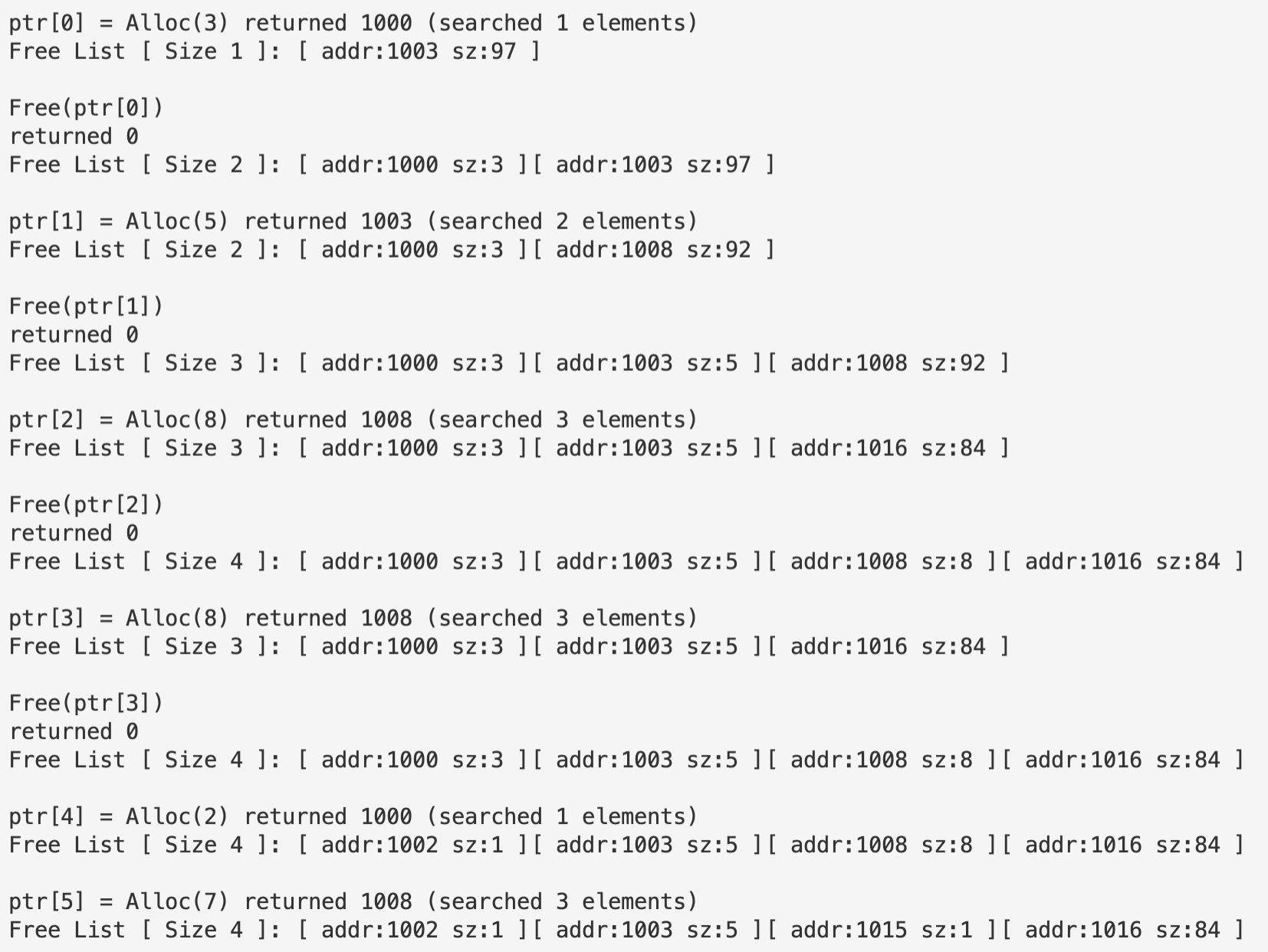
Since the policy used here is BEST, the system will always search for the fittest free space chunk for each call of allocation. Note that adjacent free chunks will not be merged in this case. Alloc() returns the initial address of the address space that it has just assigned, and free() returns 0 if the freeing action is successful.

1. **How are the results different when using a WORST fit policy to search the free list (-p WORST)? What changes?**



Policy WORST automatically search for the largest chunk in the free list. In the above case, since the space allocated altogether hasn’t reached half of the size of free space, every time we call alloc() the system will always allocate our desired space in the last chunk of the free list.

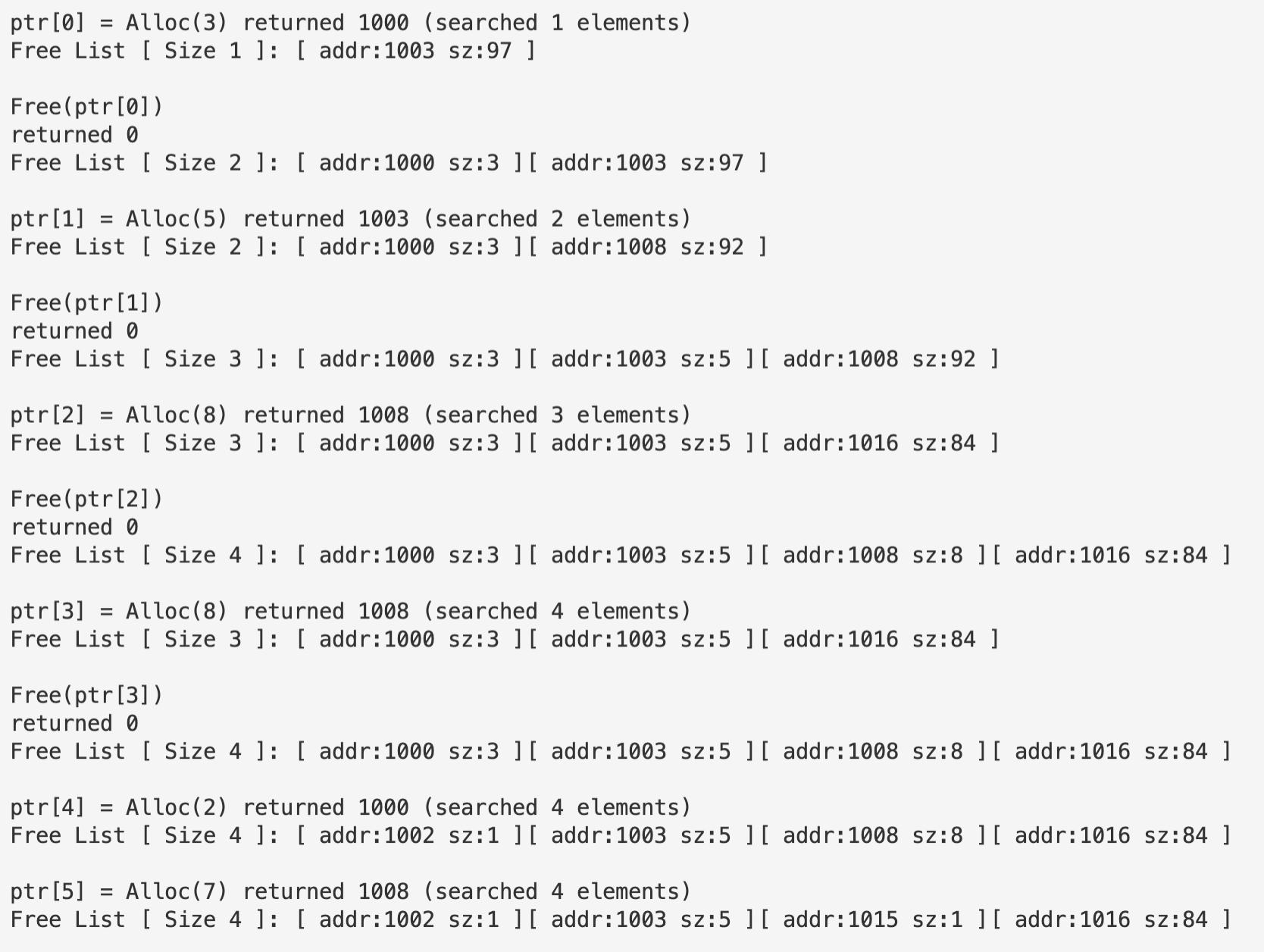
1. **What about when using FIRST fit (-p FIRST)? What speeds up when you use first fit?**



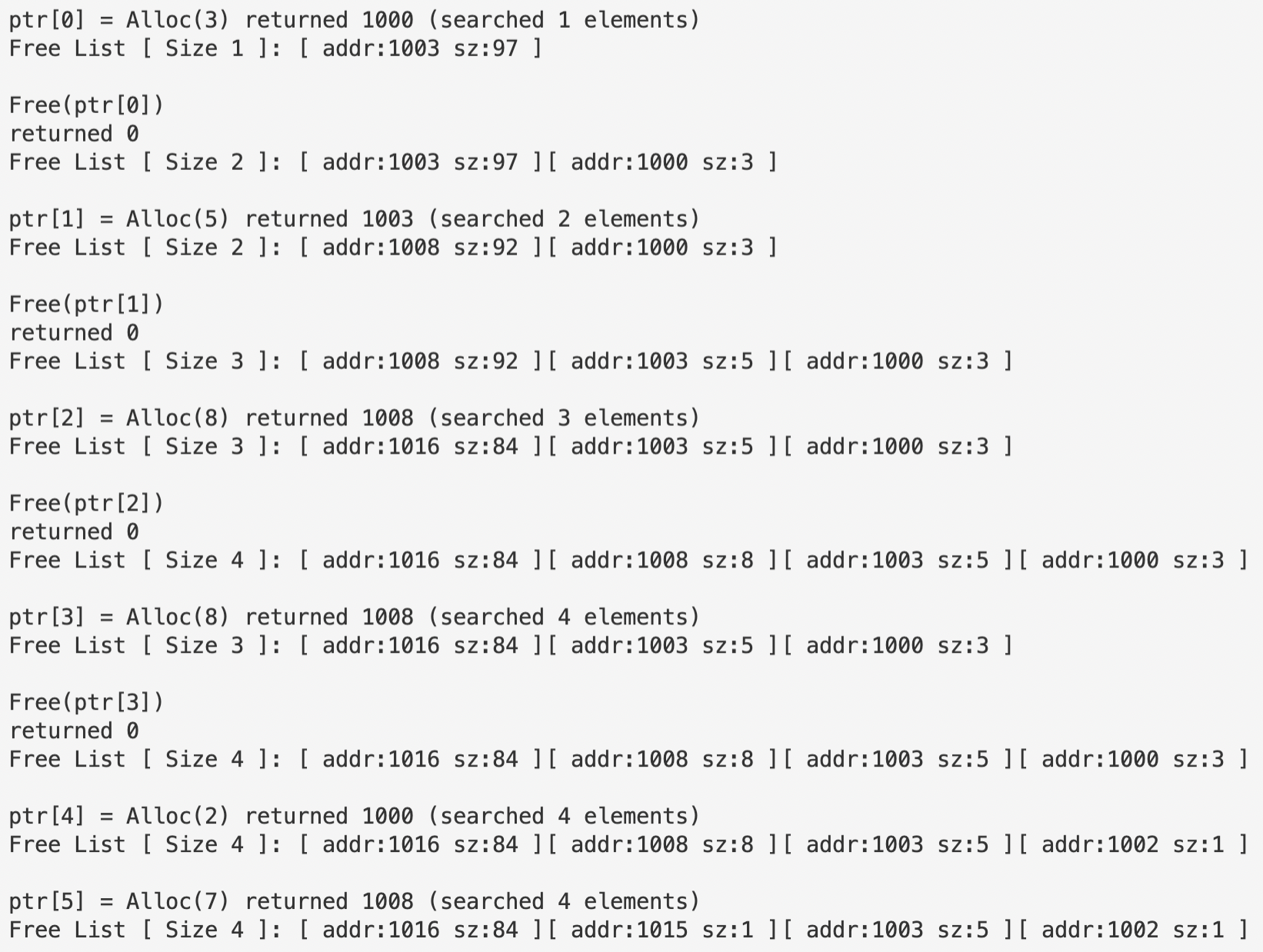
FIRST, aka first fit, will allocate the space user demands in the first chunk that fits. In the case above, the free list is arranged in the order of the initial address, so the space would be allocated in the chunk with the smallest address and also adequate capacity to hold the allocated space.

1. **For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (-l ADDRSORT, -l SIZESORT+, -l SIZESORT-) to see how the policies and the list orderings interact.**

python3 malloc.py -n 10 -H 0 -p BEST -s 0 -l SIZESORT+ -c

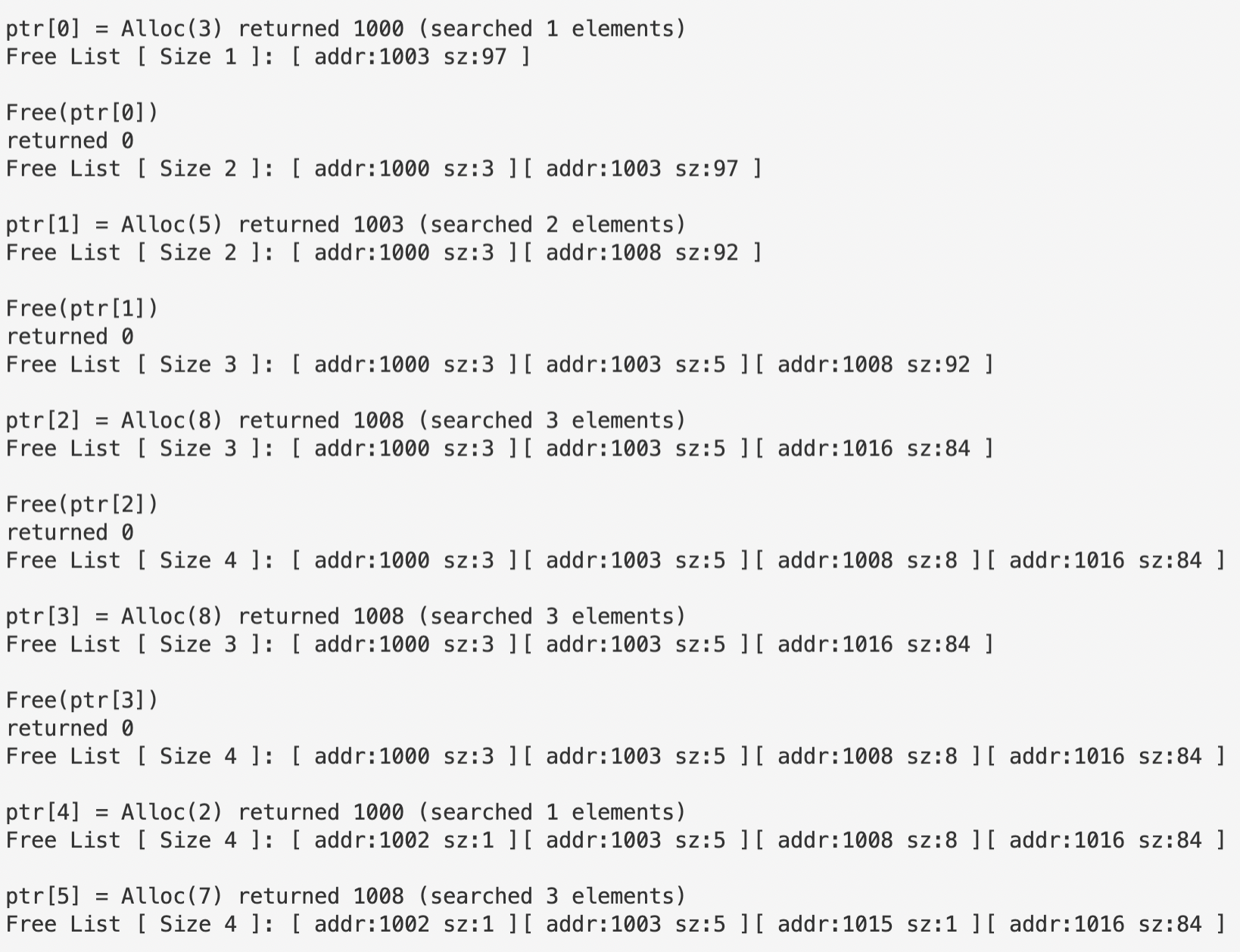


python3 malloc.py -n 10 -H 0 -p BEST -s 0 -l SIZESORT- -c

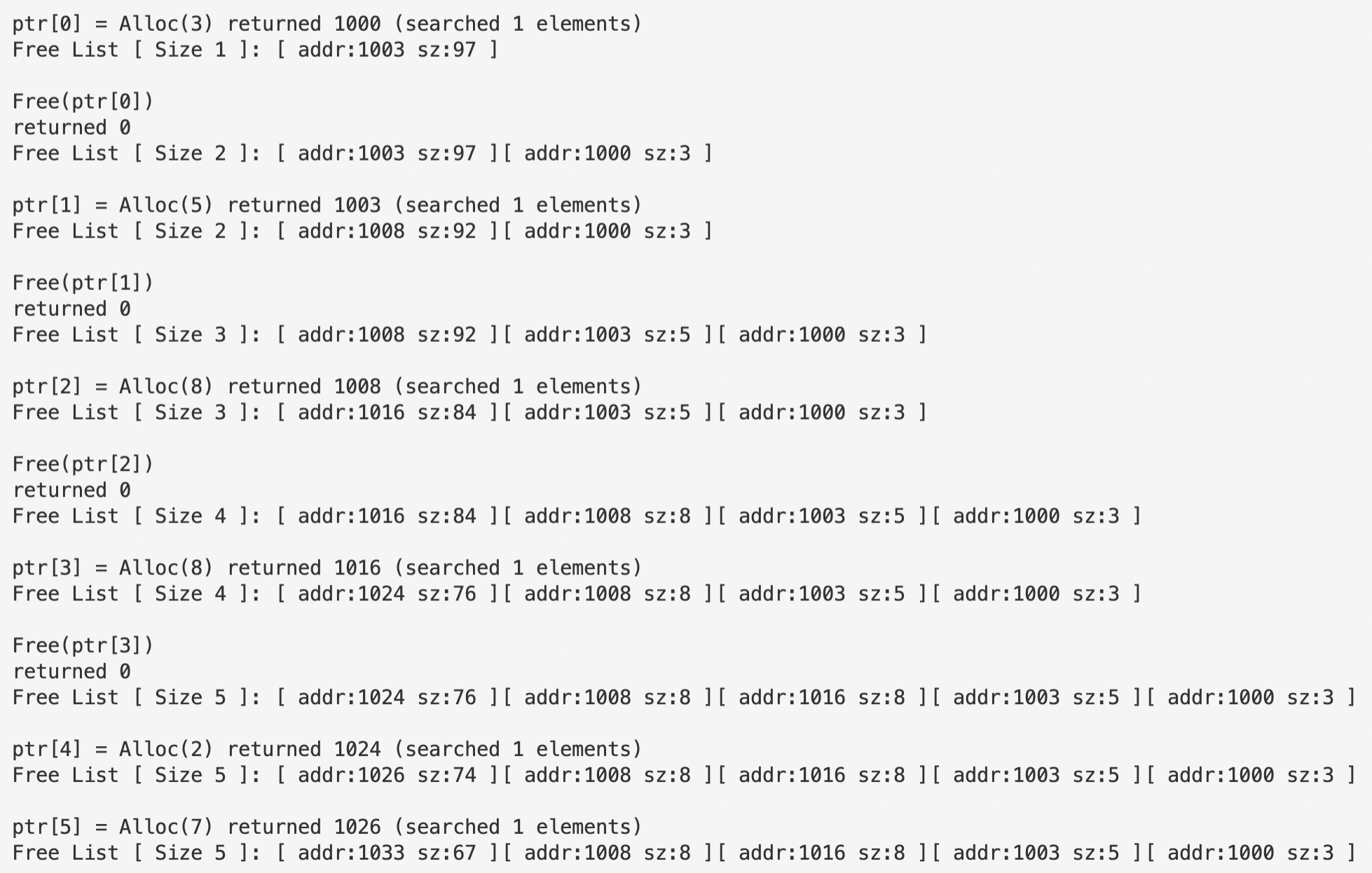


For BEST and WORST, take BEST for example, as is shown in the screenshot above, the time spent searching is completely identical regardless of the orderings of the free list, which is because those two policies will always traverse through the whole list to find either the fittest or the largest chunk.

python3 malloc.py -n 10 -H 0 -p FIRST -s 0 -l SIZESORT+ -c



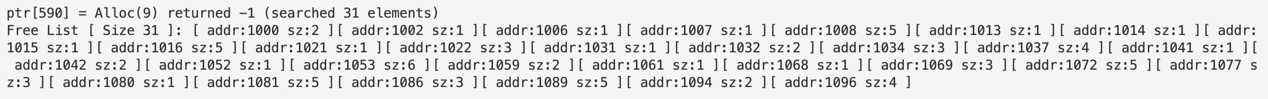
python3 malloc.py -n 10 -H 0 -p FIRST -s 0 -l SIZESORT- -c



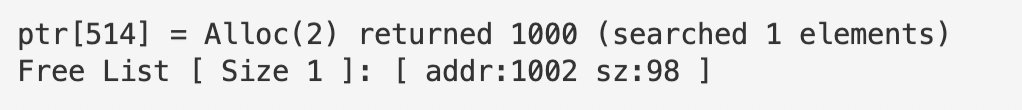
However, if we use policy FIRST, the ordering of free lists does make a difference. The total searching times for ADDRSORT, SIZESORT+ and SIZESORT- are respectively 13, 13 and 5. It’s evident that SIZESORT- cuts the searching time dramatically. This should be attributed to the fact that SIZESORT- will always place the largest chunk first, thus with policy FIRST, alloc() command either hits the first target it tries in the free list, or it’s doomed to fail even if it traverses throughout the whole free list. So, though it seems that in this case the policy combination works perfectly fine, there may well be cases where user tries to allocate multiple huge address space chunks, which could cause a huge time waste.

1. **Coalescing of a free list can be quite important. Increase the number of random allocations (say to -n 1000). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the -C flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?**

python3 malloc.py -s 0 -n 1000 -c



python3 malloc.py -s 0 -n 1000 -C -c



python3 malloc.py -s 0 -n 1000 -C -l SIZESORT+ -c



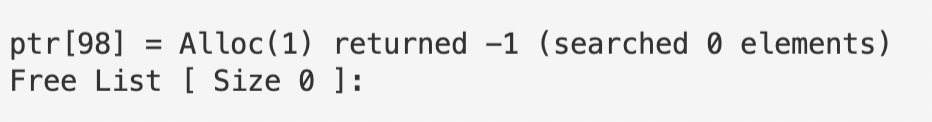
The above three screenshots shows the outcome of the last alloc() without coalescing, with coalescing and with both coalescing and SIZESORT+ ordering for policy BEST. It is clear that coalescing guarantees the system a better performance both regarding the size of the free list and searching time, which is strongly connected with the size of the free list. However, if we also demand the system to organize the free list by the size of each chunk, the performance will again be poor. This is because SIZESORT disrupts the adjacency regarding the address of neighbor elements in the free list, which in turn hinders the effectiveness of coalescing.

1. **What happens when you change the percent allocated fraction -P to higher than 50? What happens to allocations as it nears 100? What about as the percent nears 0?**

python3 malloc.py -s 0 -P 60 -n 100 -c



python3 malloc.py -s 0 -P 99 -n 100 -c



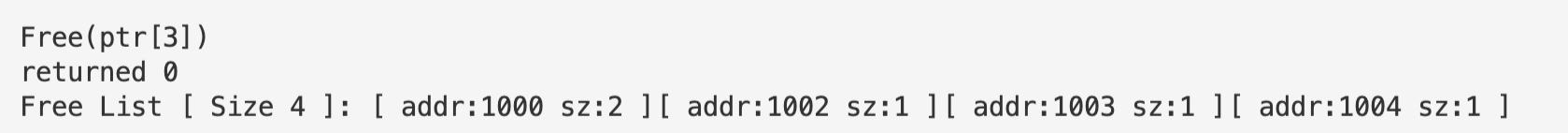
python3 malloc.py -s 0 -P 1 -n 100 -c



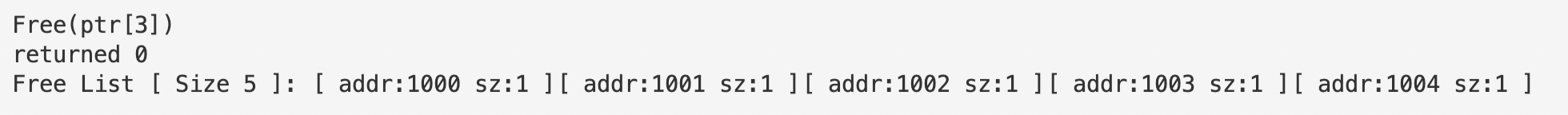
When the -P flag has a value of 60, the length of the final free list is 9. As it approaches 100, in which case there would only be allocating but no freeing, the space demanded gradually crams up and in the end takes up the whole address space. However, when the value approaches 0, the actual ratio of alloc() is 50%, which is because free() needs to have an authentic object to act upon, meaning that number of free() should always be equal or less than that of alloc(). That said, when the ratio is adjusted below 50%, every call of alloc() is followed by free(), which makes the sum of the size of the elements in the free list just the size of the whole address space.

1. **What kind of specific requests can you make to generate a highly fragmented free space? Use the -A flag to create fragmented free lists, and see how different policies and options change the organization of the free list.**

python3 malloc.py -s 0 -S 5 -A +4,-0,+3,-1,+2,-2,+1,-3 -c



python3 malloc.py -s 0 -S 5 -A +4,-0,+3,-1,+2,-2,+1,-3 -p WORST -c



python3 malloc.py -s 0 -S 5 -A +4,-0,+3,-1,+2,-2,+1,-3 -p FIRST -c

