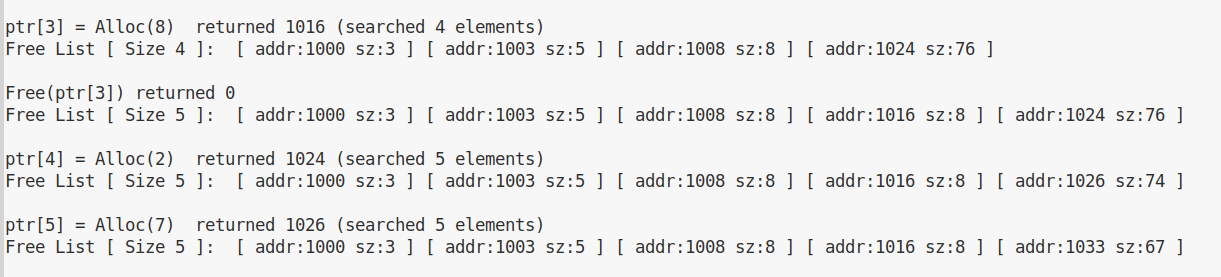
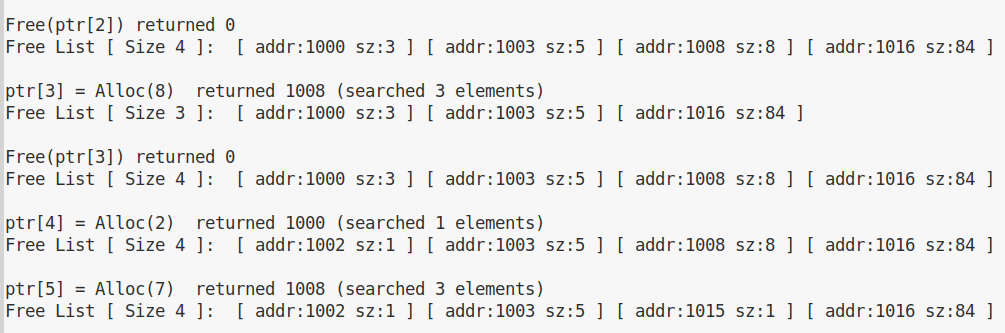
**Gruppe: 15 Luis Nothvogel, Tobias Schoch**

**17.2**

Bei Verwendung der WORST fit Policy fällt auf, dass es ein Fragment mehr in der „free list“ gibt (Blau umrandet). Dies passiert durch die Verwendung des größten freien Fragmentes in der „free list“. In diesem Fall wird nicht ein anderes passendes Fragment ausgewählt, sondern immer das größte und das war bei diesem Beispiel die verbleibenden 76 bzw. 74 Byte. Dadurch entsteht ein weiteres Element in der „free list“.



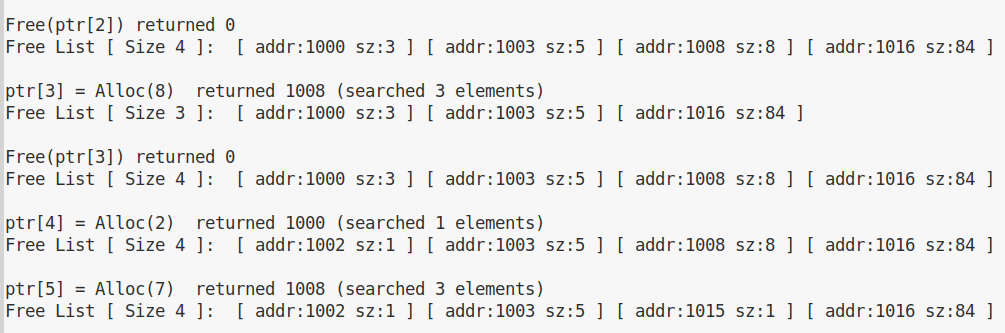
Worst fit



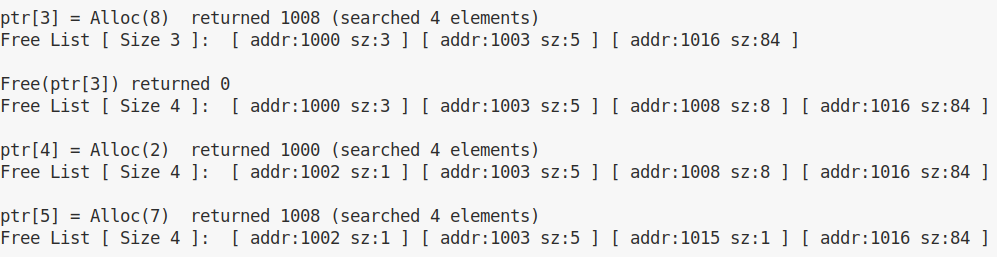
Best fit

**17.3**

Gemäß der „FIRST fit Policy“ wird in der „free list“ immer das erste passende Fragment gewählt. Das kann man auch in der Ausgabe beobachten. Die „FIRST fit Policy“ ist wesentlicher schneller, was sich in der Anzahl der durchsuchten Elemente wiederspiegelt. Bei der WORST Policy und der BEST Policy wurde durch jedes Element gegangen, während bei der FIRST Fit Policy lediglich das erste passende verwendet wird. Deshalb haben wir auch bei WORST zum Schluss jeweils 4 durchsuchte Elemente und bei FIRST Fit weniger.



First fit



Best fit

**17.4**

Bei der Listensortierung SIZESORT+ sowie ADDRSORT ist die Free Liste der Größe nach aufsteigend sortiert. Also das größte Fragment ist am Ende der Liste.

Bei der Listensortierung SIZESORT- ist die Free Liste der Größe nach absteigend sortiert. Also das kleinste Fragment ist am Ende der Liste.

**Einfluss von SIZESORT+ und ADDRSORT:**

* FIRST Fit: Keinerlei Einfluss

*FIRST FIT*

****

**Einfluss von SIZESORT-:**

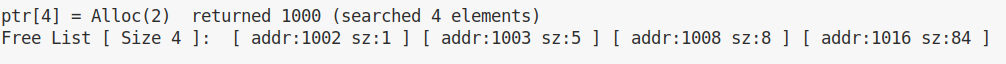
* FIRST Fit: Die Blöcke sind nun in der perfekten Reihenfolge, also mit dem Größten Element zuerst. Dadurch kann FIRST Fit logischerweise enorme Zeit einsparen.

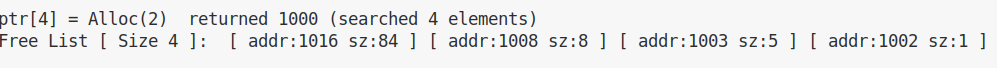
*FIRST Fit*



Die Listensortierungen haben keinerlei Einfluss auf BEST und WORST Fit, da diese selbst noch alle Blöcke durchgehen und diese nochmal für sich selbst sortieren. Daher ist die Geschwindigkeit und die Größe der Free Liste gleichgeblieben.

*Best fit – keine Änderung bei der Größe der Free Liste oder der Geschwindigkeit*





*Worst fit – keine Änderung bei der Größe der Free Liste oder der Geschwindigkeit*

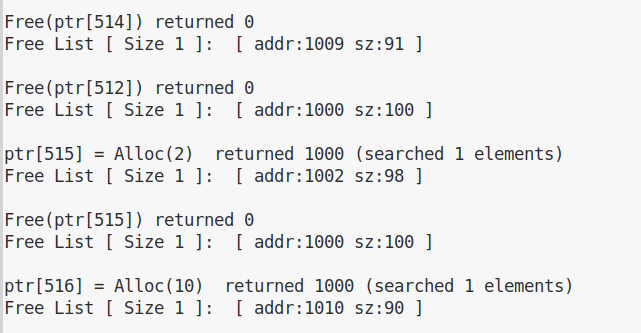




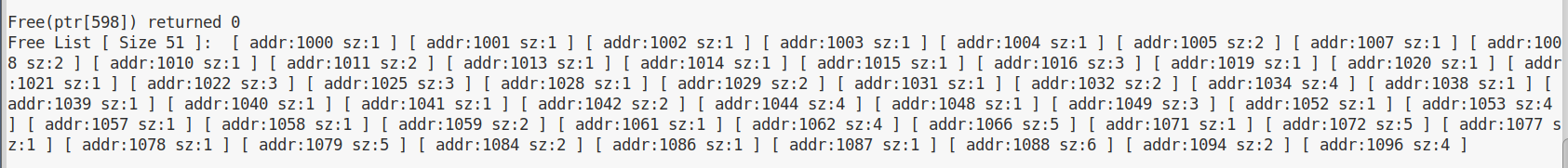
**17.5**

Nach 1000 Ausführungen kann es möglich sein, dass man sehr viele kleine Blöcke hat. Da diese stets aufgesplittet werden, aber nicht mehr zusammengefügt werden. Mit Coalescing ist dies jedoch möglich, da sich Nachbarblöcke verknüpfen können um die external Fragmentation zu vermindern. Durch die Verknüpfung ist es nun auch einfacher für größere Reservierungen einen freien Platz zu bekommen. Auch für die Schnelligkeit ist es von großem Vorteil, da die „Free List“ weniger Elemente hat, die nach passender Größe durchsucht werden müssen.

Unten in den Schaubildern kann man schön sehen, was für einen Unterschied in der „Free List“ diese Technik hat.



*Mit Coalescing*



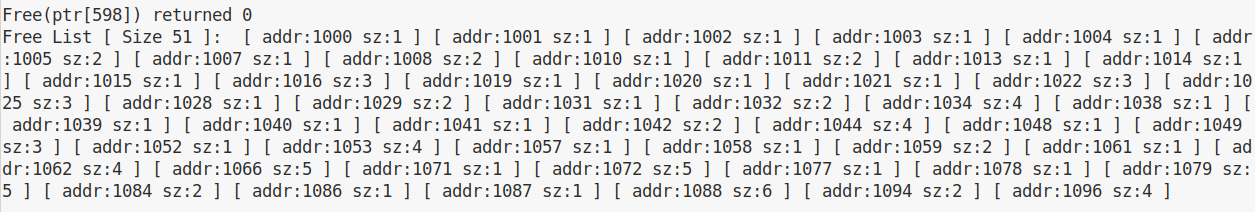
*Ohne Coalescing*

Wenn man die entsprechenden verschiedenen Policies ausführt, dann sieht man direkt was für eine große Effizienz Coalescing hat.

**Ohne Coalescing**

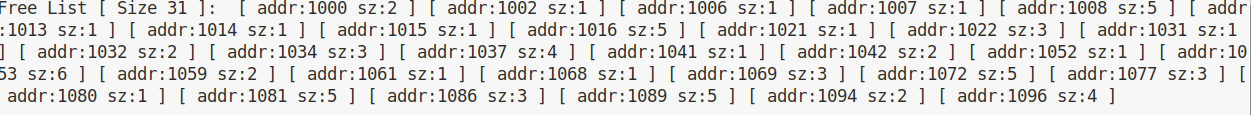
FIRST List Size: 51





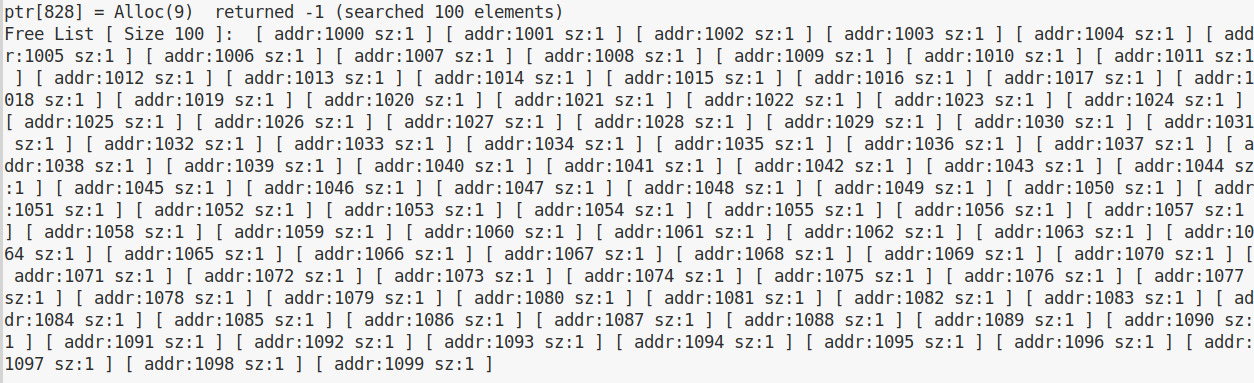
BEST List Size: 31





WORST List Size: 100





**Mit Coalescing**

FIRST List Size: 1





BEST List Size: 1





WORST List Size: 1



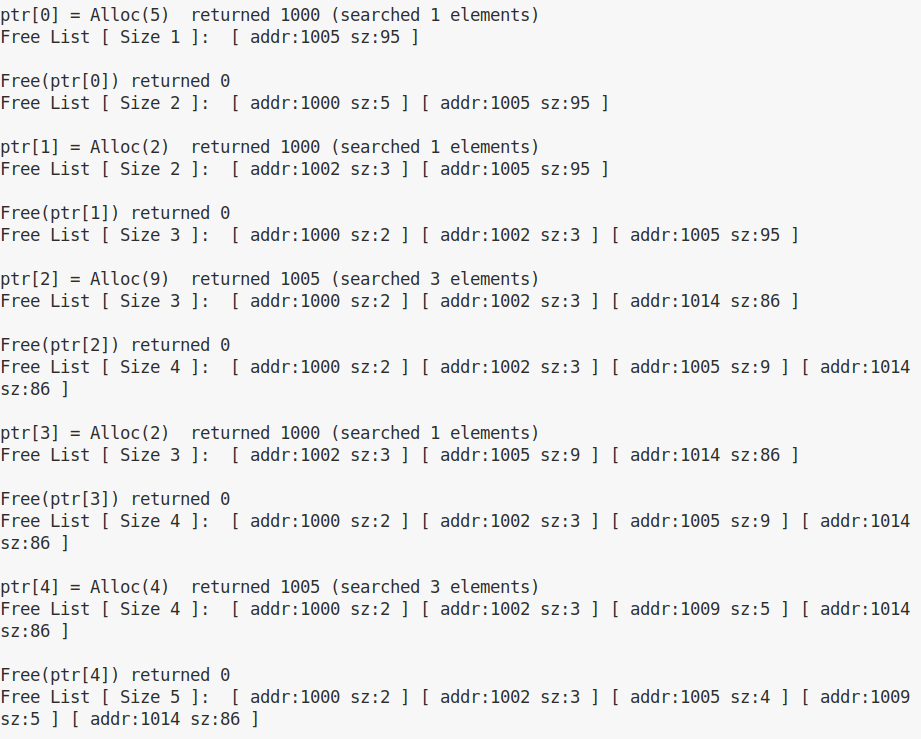


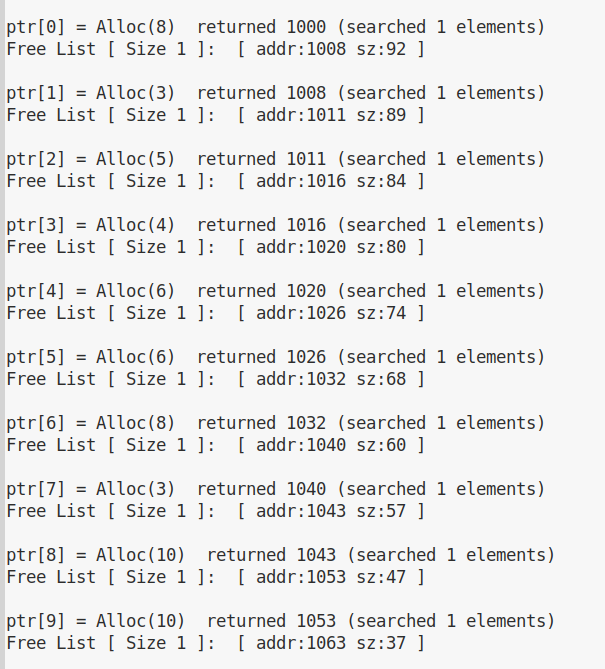
Die Sortierung der Liste ist nur für die FIRST fit Policy relevant, sowie die Next Fit Policy, da diese lediglich das erste bzw. das nächste freie passende Element verwenden, während die Anderen zuerst jedes Mal für sich selber „sortieren“ was der größte bzw. kleinste Block ist.

**17.6**

-P als Flag beschreibt die Wahrscheinlichkeit, dass eine zufällige Aktion eine Speicherreservierung ist. Desto höher die Wahrscheinlichkeit, desto niedriger ist die Wahrscheinlichkeit, dass es kein free wird. Das heißt mit einem Wert von 75, besteht eine Wahrscheinlichkeit von 75%, dass eine Aktion eine Speicherreservierung ist.

Wenn die Wahrscheinlichkeit auf 1 gesetzt wird, dann führt das Programm zuerst eine Speicherreservierung aus, da es noch kein Free ausführen kann. Danach wird mit einer Wahrscheinlichkeit von 99% free ausgeführt. Danach findet eine Speicherreservierung statt, da wieder kein free ausgeführt werden kann usw. Man kann ein schönes Pattern erkennen.



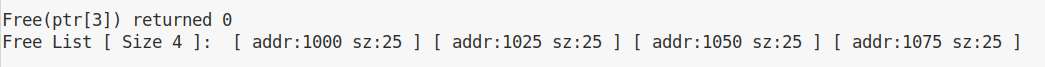
Wenn die Wahrscheinlichkeit auf 99 gesetzt wird, dann führt das Programm bei einer Aktion zu 99% eine Speicherreservierung aus und reserviert Speicher. Hier kann man schön sehen, wie 10-mal Speicher reserviert wird, aber kein einziges Mal Speicher befreit wird.

**17.7**

*./malloc.py +25,+25,+25,+25,-0,-1,-2,-3 -c*

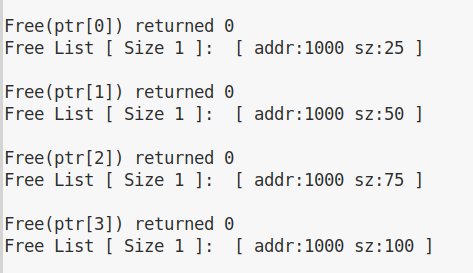
Mit dieser Anfrage für die Speicherreservierung haben wir nun die Größe der „Free List“ vervierfacht.

In der Liste sind nun 4 Elemente mit jeweils einer Größe von 25, die im Anschluss direkt befreit werden mit free (-0,-1,-2,-3).

**

Ich interpretiere die Aufgabe so, dass wir die abzuarbeitende Liste von dem obigen Beispiel verwenden sollen:

*./malloc.py +25,+25,+25,+25,-0,-1,-2, -3 -c -C*



Wenn wir nun mit Coalescing arbeiten (-C), dann haben wir stets nur ein Element in der „Free List“. Dank Coalescing verbinden sich stets die Nachbarelemente, sofern diese Frei sind.

*./malloc.py +25,+25,+25,+25,-0,-1,-2, -3 -c -C -l SIZESORT+*

*./malloc.py +25,+25,+25,+25,-0,-1,-2, -3 -c -C -l SIZESORT-*

Mit “SIZESORT+/-“ oder „ADDRSORT“ zu arbeiten erübrigt sich hier, da alle Elemente gleich groß sind.

Daher erledigt sich auch BEST, WORST und FIRST, da alle stets das erste Element bekommen und dieses dann auch frei ist.