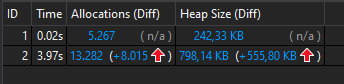
**ALGOS Plan**

Vektoren wurden für die Einträge der historischen Daten verwendet, da das Freien des Speichers dann übernommen wird.

Die Größe der Hashtabelle wurde so gewählt, dass die Tabelle unter 50% Auslastung bleibt, da sonst die quadratische Sondierung eventuell keinen freien Platz mehr finden kann. Falls die Größe keine Primzahl ist passiert kann dies sogar schon unter 50% passieren.

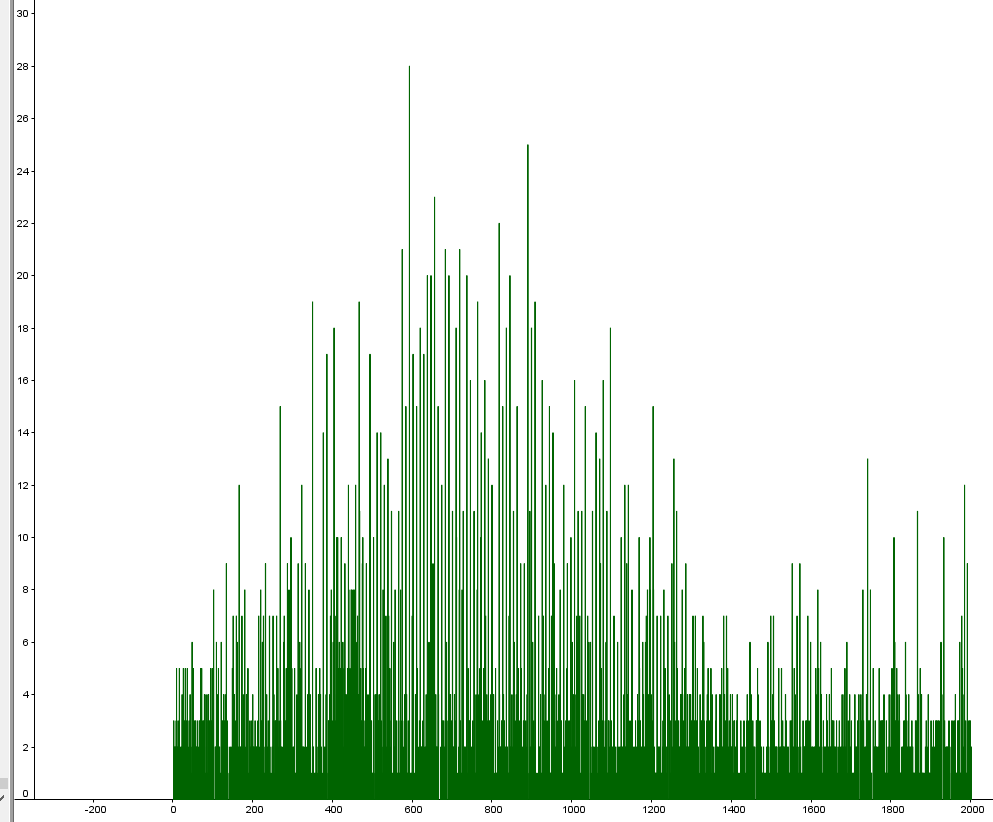
Die Hashtabelle wird nicht mit new allokiert, da sie sonst auf dem Heap gespeichert wird. Der Stack ist schneller. EDIT: die Hashtabelle wird mit new allkoiert, da sich sonst VS aufregt, dass zu viel am Stack gespeichert wird (~500 KB).

Bezüglich der Allokierung der einzelnen stocks: Man könnte sie auch erst allokieren, wenn man die einzelnen index benötigt, jedoch würde bei der löschung die node bestehen bleiben müssen, da sonst ihr deleted attribute nicht gesetzt werden kann, was im endeffekt einfach nur zu random allokierten nodes führen würde. Deswegen allokieren wir sie quasi on startup. Ist zwar eine größere speicherauslastung, aber vereinfacht die handhabung

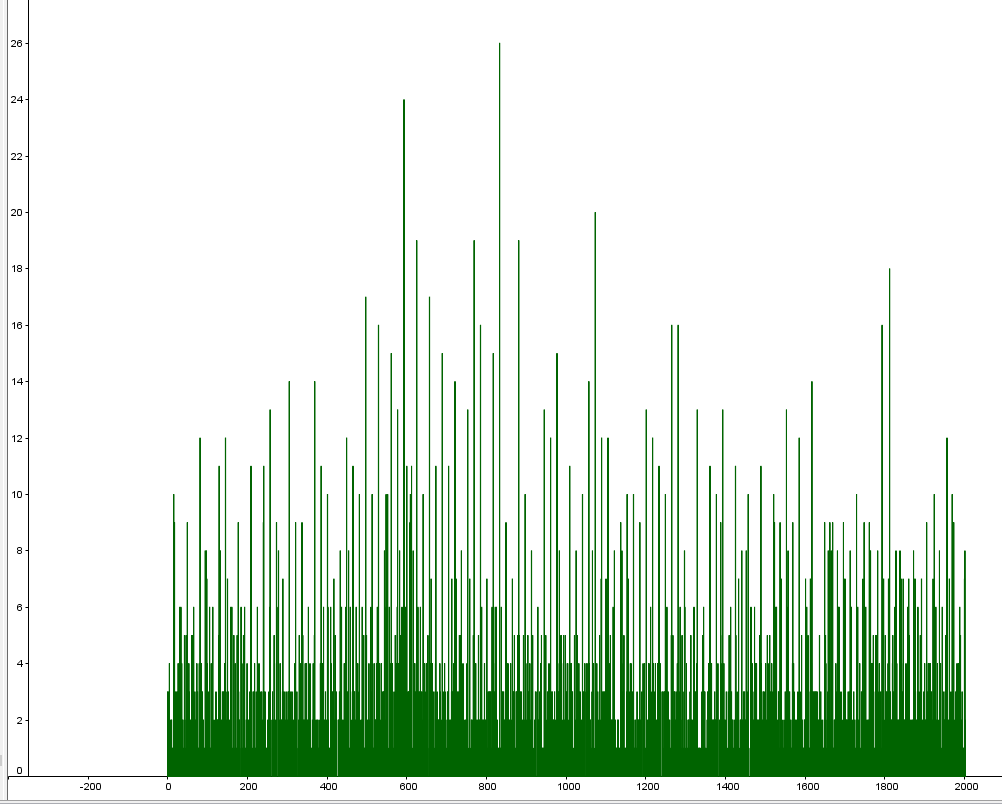


Evaluierung der Hashfunktion:

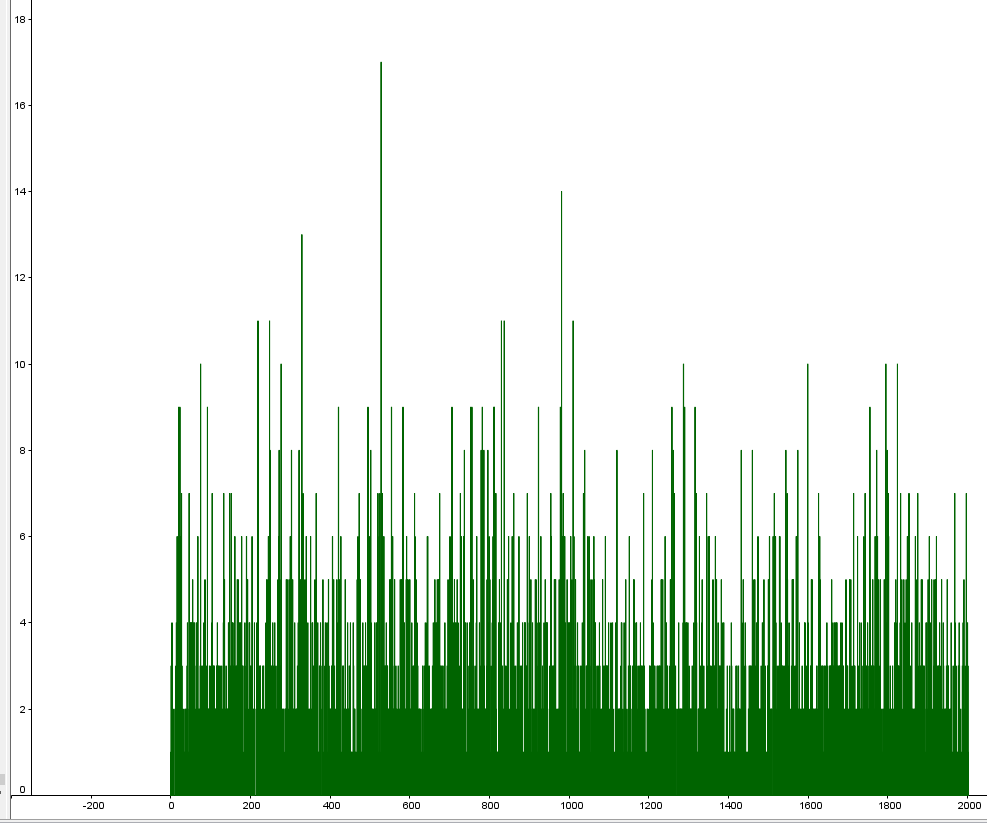
Mit 3er Potenz:



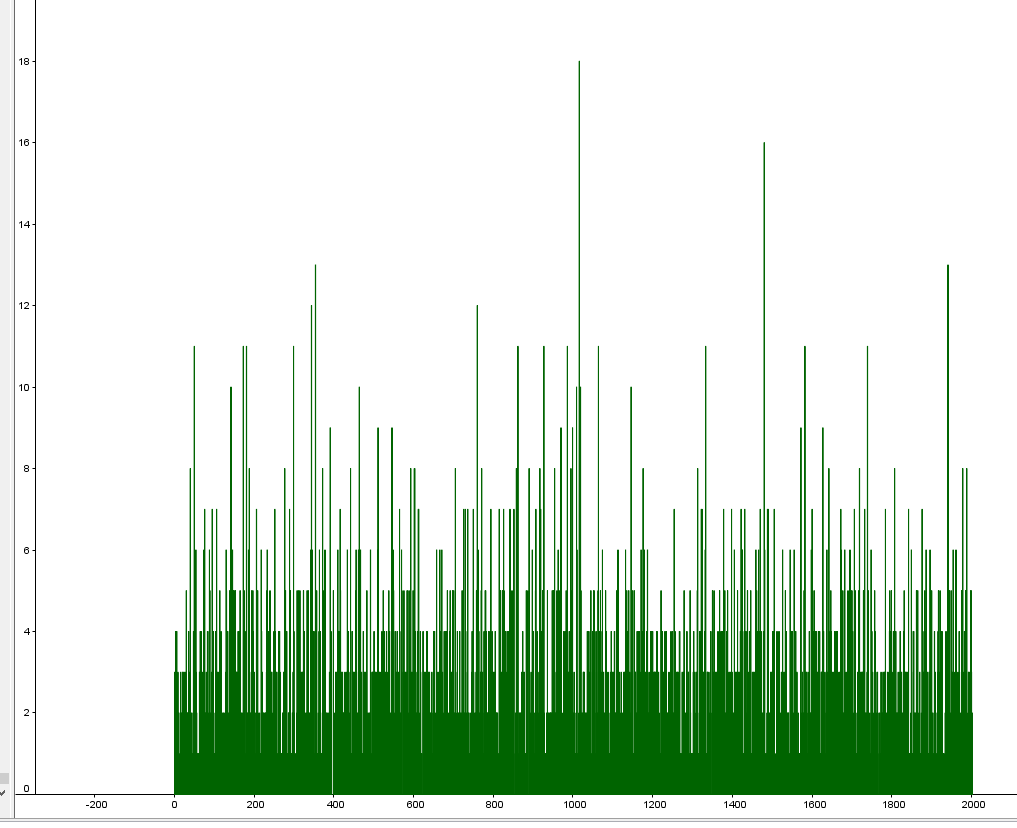
Mit 4er Potenz:



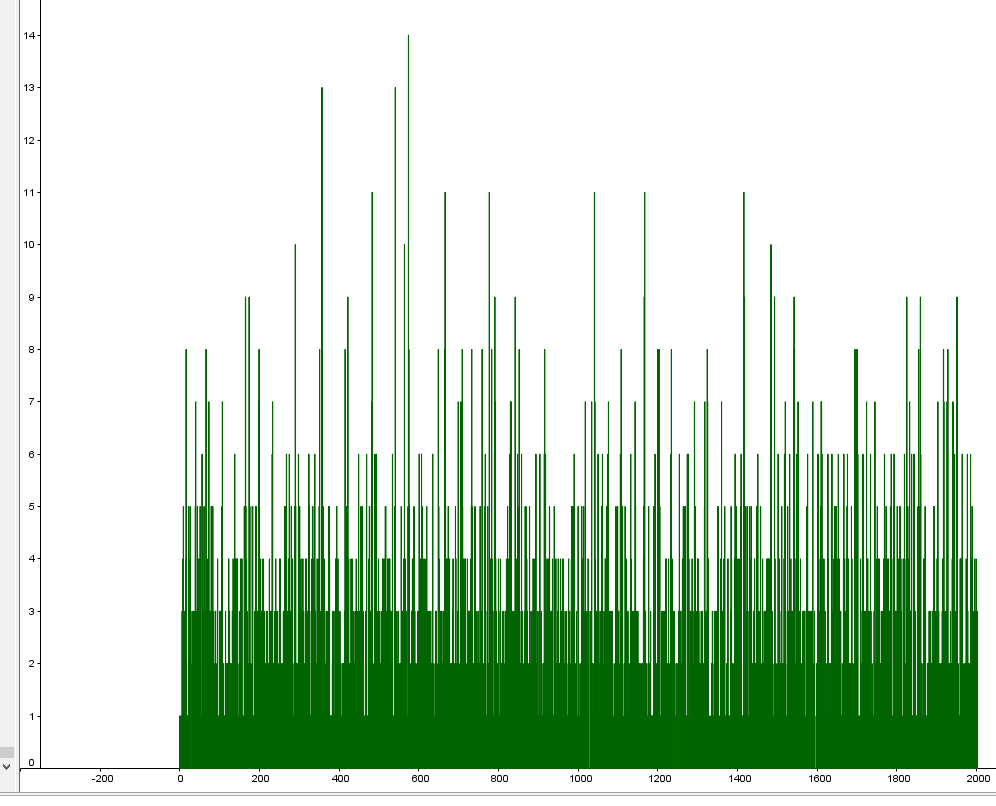
Mit 31er Potenz:



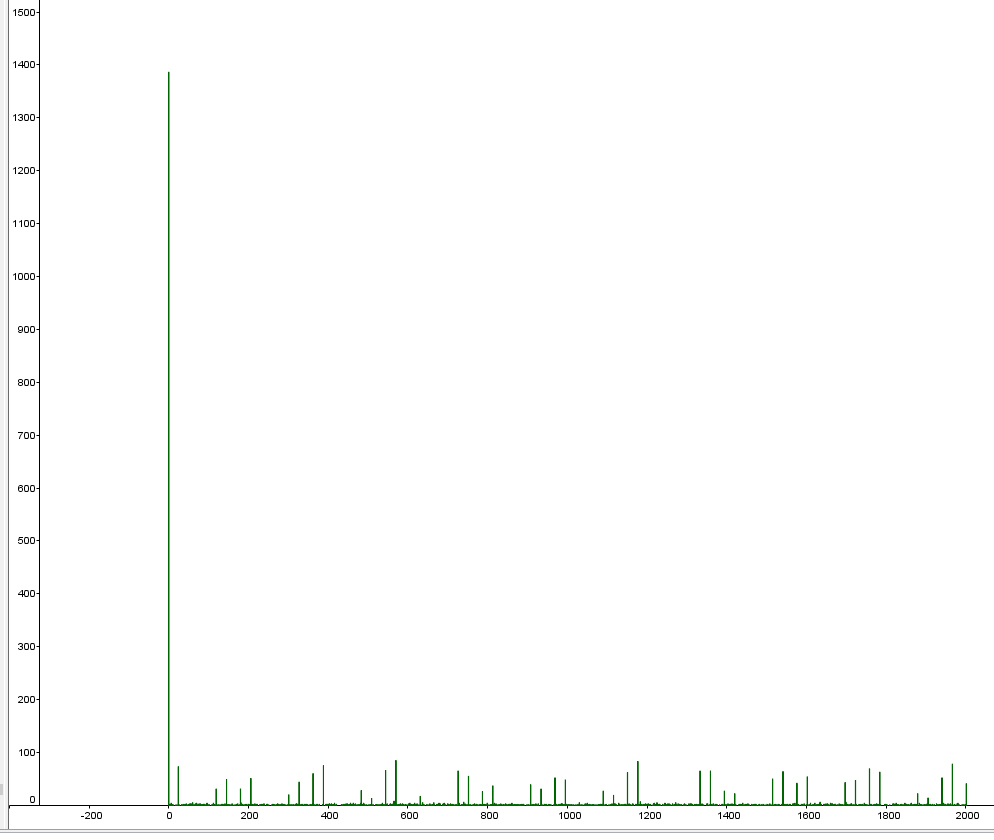
Mit 32er Potenz:



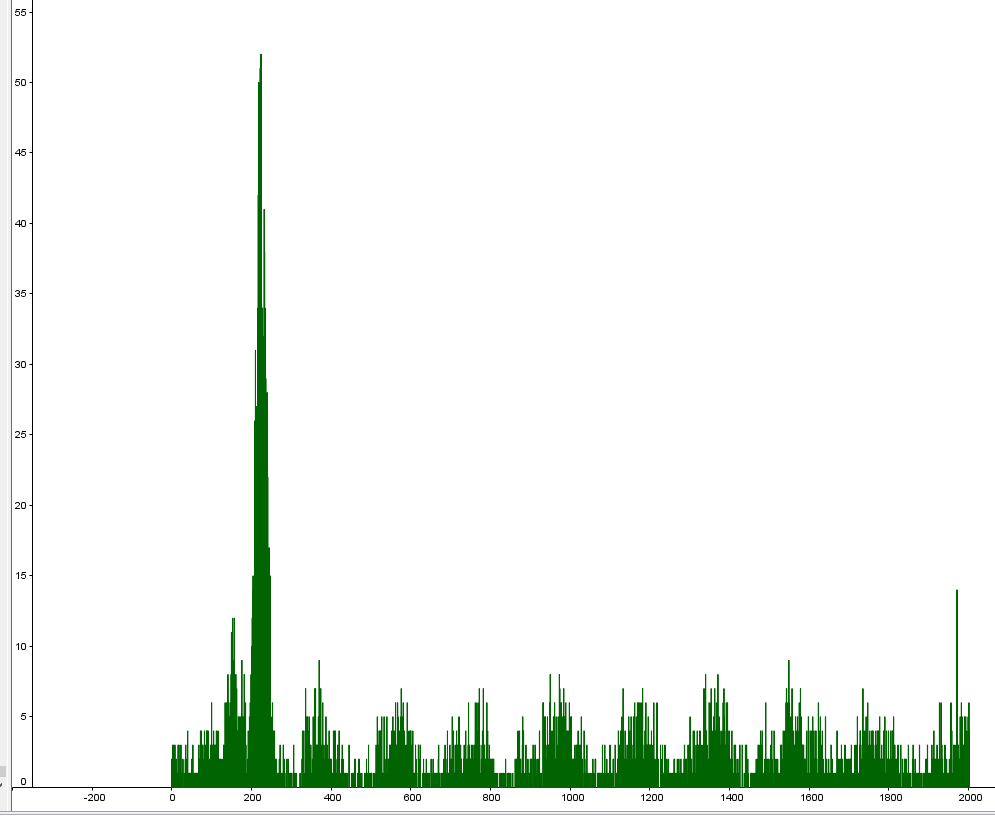
Mit 71er Potenz:



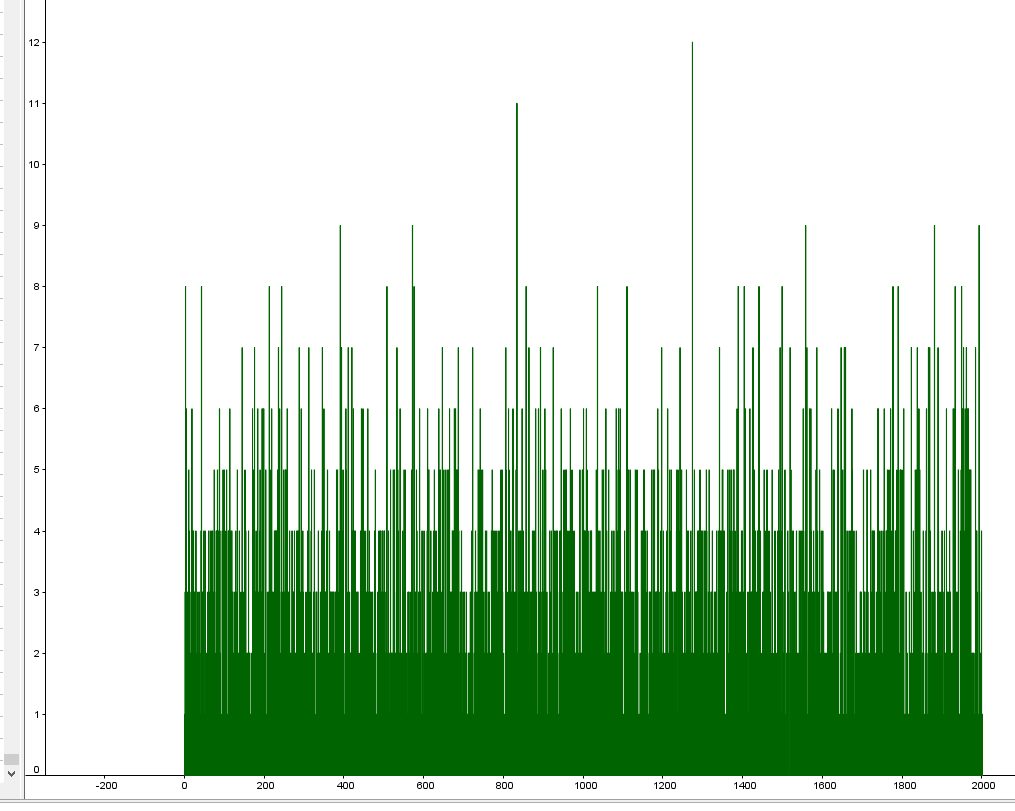
Mit 2003er Potenz:



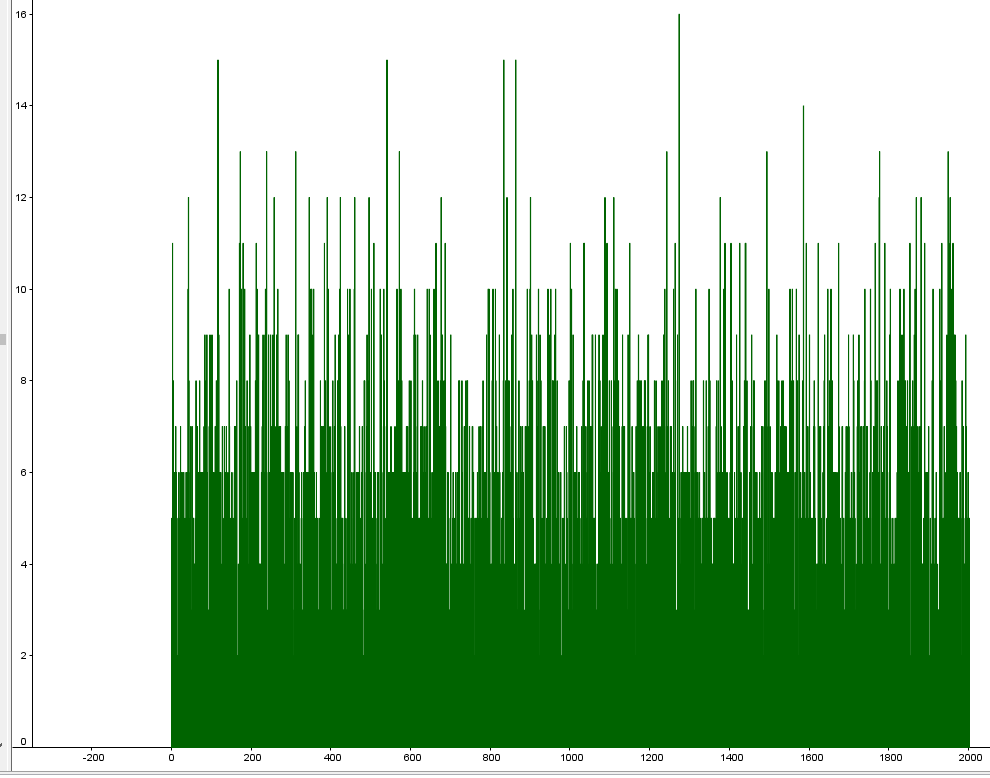
Mit 2004er Potenz:

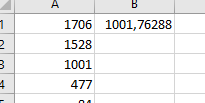


Mit mutiplizieren von jedem char:



Mit 9.999 Werten



Durchschnitt über 100.000: 

Probung:

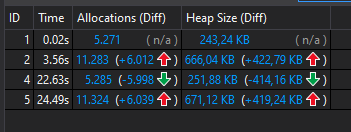
Sollte rekursiv funktionieren, variable dafür ist auch mit uint32\_t deklariert, sodass überall gleich überläuft, sofern so lange nach einem freien platz gesucht wird.

Import:

Da es uns nicht möglich war die Zeilenanzahl vorm Einlesen herauszufinden (da es teilweise sehr neue aktien gibt wo die zeilenanzahl geringer ist) ohne die datei zu durchlaufen, was sowieso beim einlesen gemacht wird, haben wir uns dazu entschlossen jeden eintrag in einen doppelvektor zu speichern und die zeilen mitzuzählen. Danach werden die letzten 30 Vektoren zu Data structs gemacht und in den history vektor gespeichert.

Serialisierung:

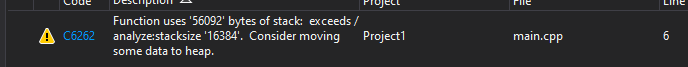
Wir haben uns die library boost angeschaut und ausprobiert, jedoch müsste man ein archive template pro klasse oder struct hinzufügen, was alles sehr unübersichtlich und kompliziert gestaltet. Dabei wird auch noch in textform gespeichert, was zudem unnötig speicher verbraucht. Obwohl es nicht die effizienteste methode ist haben wir uns darauf geeinigt manuell zu serialisieren und deserialisieren. Dabei wird die Hashtabelle durchlaufen und alle Stocks die entweder auf deleted oder filled sind werden in eine CSV datei geschrieben. Zudem werden auch die History Vektoren gespeichert. Beim Laden der Datei wird zunächst geschaut ob die Datei geöffnet werden kann, danach die gesamte Hashtabelle deallokiert und wieder allokiert, um sie quasi zu clearen, und die Einträge, ähnlich wie bei der import Funktion, eingelesen.



Größe mit mit new allokiertem Dictionary



Auf dem heap würde die volle tabelle bissl zu viel platz verbrauchen



Deswegen allokieren wir sie mit new, nachteil daran ist, dass wir nicht nur ein string array machen können, sondern noch das deleted attribut dazuspeichern müssen.

Hashing fürs dictionary, mit gleicher funktion für 10.000 Namen: