Der Grundgedanke hinter der Double-Ended-Queue bestehend aus Blöcken ist, dass man sich Allocations-zeit spart. Diese Ersparnis kommt zu Stande, indem man durch einen Allocations-prozess platz für mehrere zu Speichernde Objekte schafft. Diese „Blöcke“ in derer man dann mehrere Objekte Speichen kann müssen jedoch auch individuell gespeichert werden, damit diese kontinuierlich adressierbar bleiben. Auch ist hierbei zu beachten, dass die bisherig einfache Implementierung der Head und Tailpointer an Komplexität zunimmt. Hierbei kann der Fall auftreten, dass innerhalb eines Blocks weder Head-, noch Tail-Pointer zu finden sind. Auch ist es nicht verpflichtend, dass Head- oder Tail-Pointer überhaupt im selben Block sind, da mit ansteigender Länge der Deque diese zunehmend auseinanderrücken. Somit werden hierbei zwei neue Variablen eingeführt, welche immer auf den Block zeigen, in dem sich der entsprechende Head- oder Tail-Pointer befindet. Der Vorteil dieser Implementation ist zwar der Zuwachs an Zugriff und Speicher Geschwindigkeit, jedoch kostet dies Speichereffektivität und kann zu sehr leeren, jedoch reservierten, Bereichen im Speicher führen.

Ein Schematischer Aufbau dieser Implementation ist in Abbildung [einfügen] zu sehen.

Erzeugt man nun eine neue Warteschlange so muss zu beginn erst eine Karte mit einer festgelegten länge definiert werden. Zudem wird der Element-counter hier wieder auf 0 gesetzt, da die Warteschlange nach dem Erzeugen noch keine Elemente hält.

Fügt man nun ein Element in die Warteschlang am Head ein, so wird überprüft, ob diese bereits Objekte beinhaltet. Ist dem nicht der Fall, so muss zu beginn ein Block erzeugt werden. Dieser wird in die (relative) Mitte der Karte gelegt, um genügend Platz in beide Richtungen zu haben die Datenstruktur wachsen zu lassen. Die Pointer auf Head und Tail werden gesetzt, und ebenso müssen die Pointer auf den Block, in derer diese sich befinden, definiert werden. Zuletzt wird das zu speichernde Element in den Block gelegt und somit der Speicher definiert.

Wiederholt man dies nun und fügt weitere Objekte ein werden sich die Pointer innerhalb eines solchen Blocks so verhalten wie man es schon bei der Statischen Deque beobachtet hat. Verlässt jedoch einer der Pointer den Definierten Index Bereich des Blocks so wird nicht wie bei der Statischen Implementierung der Pointer an das andere Ende des definierten Bereiches gelegt, sondern ein Neuer Block erzeugt. Mit erzeugen eines neuen Blocks muss in diesem Bsp. dementsprechend auch der Head Pointer, in diesen gelegt werden, was durch den HeadBlock Pointer signalisiert wird.

Dies kann nun wiederholt werden, bis die Karte an einer Stelle überläuft und der Headblock-, bzw. Tailblock- Pointer den Definierten Speicherbereich der Karte verlässt. Wenn dies der Fall ist, muss überprüft werden, ob die Karte voll ist, oder ob nur ein kleiner Teil des Speichers tatsächlich belegt wurde.

Im Fall, dass die Karte tatsächlich voll ist und einseitig überläuft, wird eine neue Karte erstellt. Die Neue Karte ist in dieser Implementation doppelt so groß wie die Alte Karte. Als nächstes muss berechnet werden, wo die Alte Karte in die neue gelegt werden muss. Hierzu ist zu beachten, an welcher stelle der Überlauf stattgefunden hat. Ist dies am Head passiert, so muss der erstellte freie Speicherbereich vor der alten Karte sein. Betrachtet man nun die Abbildung [einfügen] so wird eindeutig, dass man die Alte Karte um X in die Neue verschoben legen muss. Hierbei berechnet sich X einfach aus der Länge der Neuen Karte (NL) minus die Länge der Alten Karte (L). Der dadurch berechnete Index ist somit der Versatz, um welchen die Alte Karte mit Hilfe von „memcpy“ in die Neue kopiert werden muss.

Falls der Überlauf am Tail stattfindet, muss hierbei die Alte Karte lediglich in die neue gelegt werden, da diese kürzer ist als die Neue und somit der Freie Speicher nach der Alten Karte übrigbleibt.

Hierbei jedoch abzufangen ist ein überlauf der Karte, während die Warteschlange selbst jedoch im Verhältnis kaum Elemente beinhaltet, der Speicher also nahezu leer ist. Dieses Verhalten tritt auf, wenn innerhalb der Deque mindestens ein Element befindlich ist und wiederholt, am z.B. Head Objekte angehängt werden, welche im Anschluss am Tail entfernt werden. In der Statischen Deque würde sich hierbei der Definierte Bereich des Speichers im Kreis bewegen. In dieser Implementation würde jedoch wiederholt in eine Richtung neuer Speicher reserviert werden, während die Warteschlange selbst nur wenige Objekte beinhaltet.

Somit muss beim versuch die Karte zu vergrößern zuerst überprüft werden, wie viele Blöcke tatsächlich Daten beinhalten. Wenn die Anzahl dieser unter einem Schwellenwert liegt, in dieser Implementation 30%, muss ein Teil der Freien Speicherblöcke vor die Belegten geschoben werden.

Dafür muss die Mitte der Karte und damit, die zu bewegende Distanz, des Definierten Speichers berechnet werden. Abhängig davon, an welcher Seite der Überlauf stattgefunden hat, muss der Definierte Speicher so gelegt werden, dass entweder der Head oder der Tail in der Mitte der Karte ist. Im Anschluss davon müssen die HeadBlock- bzw. Tailblock- Pointer wieder angepasst werden.

Nach dem Verschieben des Definierten Speichers, kann die Deque somit weiterverwendet werden. Das wichtige dabei ist, nach Abschluss dieser Funktion ist die Größe der Karte konstant geblieben und die Blöcke selbst, sowie die Head- bzw. Tail-Pointer wurden auch nicht verschoben.

Daher der Grundgedanke der Deque. Im Gegensatz zur Queue kann die Deque jedoch sowohl am Head, als auch am Tail Objekte entgegen nehmen und diese wieder ausgeben, was den Hauptunterschied zur Klassischen Warteschlange darstellt. Somit liefert die Deque in der Standard-Implementierung für sowohl den Head als auch den Tail dieselben Funktionen, welche bei der normalen Warteschlange nur einmalig existieren. So hat der Head eine Push Funktion, welche Daten der Warteschlange anhängt, aber auch der Tail bietet diese Funktionalitäten. Genauso haben beide eine Pop Funktion, welche eben diese der Warteschlange wieder entnimmt. Optional und häufig bei umfangreicheren Implementierungen zu finden ist eine Peek Funktion, welche es ermöglicht die Daten am Head, oder Tail abzurufen ohne diese der Warteschlange zu Entfernen. Durch eine derartige Implementierung ist für den Head und Tail eine Zugriffsgeschwindigkeit von O(1) gewährleistet, wofür die Datenstruktur jedoch an Komplexität zunimmt. Hierbei zu beachten gilt, dass eine Deque nicht fundamental besser als die Klassische Warteschlange ist, jedoch in spezifischen Anwendungen andere Datenstruktur-modelle an Performance übertrifft. Eines der besten Beispiele wofür eine Deque verwendet werden kann ist ein Work-Stealing-Algorithm worauf im späteren verlauf unter [todo] genauer eingegangen wird.

Die Illusion, dass Computer Prozesse gleichzeitig ausführen, ist Ihrer Geschwindigkeit und ihrer Optimierung zu verdanken. Hierbei werden Prozesse in sogenannte „Threads“ aufgeteilt, welche nacheinander oder auch teilweise durcheinander ausgeführt werden können. Dies dient dazu, einen Prozessor Kern so beschäftigt wie möglich zu halten, da ein nicht arbeitender Kern zeit verschwendet in derer er hätte Arbeiten können. Heutzutage sind Multithreading Systemen kombiniert mit Multiprocessing Systeme in nahezu jedem Computer der Welt zu finden. Eine der Fundamentalsten fragen ist es wie man Prozesse effektiver ausführen kann, um noch mehr Geschwindigkeit aus den vorhandenen Systemen herauszuholen. Hierbei können Double-Ended-Queues verwendet werden, um sogenannte Work-Stealing-Algorythems

Die Illusion das Computer Prozesse gleichzeitig ausführen ist ihrer Geschwindigkeit und Optimierung zu verdanken. Zudem werden Prozesse in Threads aufgeteilt, welche nacheinander oder durcheinander ausgeführt werden können. Diese dienen dazu, einen Prozessor Kern nicht warten zu lassen. Zeit, in der ein Prozessor auf etwas wartet, ist verschwendete Rechenkapazität und der Gedanke hierbei ist es, Ihn schon an anderen Threads arbeiten zu lassen während auf z.B. antworten von Webseiten gewartet wird. So werden bei Parallelrechnern verschiedene scheduling Strategien verwendet, wie z.B. das Work-Stealing.

Work-Stealing ist eine Multithreading Strategie wie sie in Parallelrechnern Anwendung findet. Hierbei haben verschiedene Prozessoren oder Kerne jeweils Warteschlangen worin ihre Arbeitsaufträge (Threads) auf eine Bearbeitung warten. Während mit dem Start eines Programms nur eine gewisse Anzahl an Threads generiert werden, und es versucht wird diese so sinnvoll wie möglich auf die Prozessoren aufzuteilen, können einzelne Threads bei Ihrem Sequenziellen ausführen auch neue generieren. Diese hier neu generierten Threads werden dann standardmäßig an das Ende der Eigenen Warteschlange des Prozessors angehängt. Beim Ausführen eines Threads kann es jedoch auch dazu kommen, dass der Aktuelle Thread pausiert werden muss. In diesem Fall würde der Prozessor versuchen einen anderen Thread aus dem Head seiner Warteschlange zu laden. Alternativ kann ein Thread beendet sein, was zu einem gleichen verhalten wie beim Pausieren führt. Kommt es nun zu dem Fall, dass ein Prozessor, Worker, versucht auf seine Warteschlange zuzugreifen, diese jedoch leer ist, kann dieser anstelle seiner eigenen Warteschlange proaktiv (im Vergleich zum Work-Sharing) auf die eines andern Prozessors zugreifen. Somit wird der Prozessor zu einem Stealer. Da es jedoch ineffektiv ist den Prozessor-Cash für einzelne Threads zu verwerfen, und sequenzielle Threads dies meist auch nicht benötigen würde ein eingreifen des Stealers auf die folgenden Aufträge des Workes hierbei eine relativ große Störung erzeugen. Dies kann umgangen werden, indem ein Stealer nicht versucht sich Threads vom Head der Warteschlange eines Workers zu nehmen, sondern stattdessen auf den Tail zugreift. Hierbei kommt das Verhalten einer Double-Ended-Queue zum einsatz. Als letztes kann ein Thread noch einen Thread Aktivieren, welches diesen aus der Warteschlange des Prozessors heraus an den Kopf der Deque setzt, aber den aktuellen Thread erst zu ende bearbeitet.

Stielt somit ein Prozessor einem anderen einen Auftrag, so können diese nun parallel nebeneinander ausgeführt werden, was die Ausführgeschwindigkeit des Prozesses erhöht, und die Prozessoren am Arbeiten hält. Beim Versuch einen Thread von einem anderen Prozessor zu stehlen, wird eine Warteschlange eines anderen Prozessors zufällig gewählt. Falls diese auch leer ist und somit zu einem Stealer geworden ist, wird dies so oft wiederholt, bis eine Warteschlange gefunden wurde, welche Threads beinhaltet.

Alle drei Implementationen, welche in 3. Benannt wurden, sind mit unterschiedlichen Ansprüchen belegt. Konzeptionell funktionieren diese Warteschlangen alle gleich, dennoch kann man Vermutungen zu Ihrer Performance aufstellen. Während die Implementation mit einem Array eine Fixierte Größe besitzt, müsste diese alle anderen Implementationen im Zugriff und Speichergeschwindigkeit übertreffen. Kurz darauf müsste die Dynamische Implementation mit Blöcken einzusortieren sein. Die Implementation mit einer Double-Linked-List müsste hierbei die langsamsten Zugriffe und Speichervorgänge aufweisen.

Im Rahmen dieser Seminararbeit wurden alle drei Implementationen vollständig implementiert und verglichen.Hierbei sind zwei versuche erstellt worden, um die Funktionalität zu beweisen.

In diesem Versuch wurden alle drei Datenstrukturen miteinander verglichen, wie diese sich verhalten, wenn sich bereits Daten in der Warteschlange befinden. Dabei wurden 10 Millionen Einträge jeweils in diese gelegt bevor der Test gestartet wurde. Hierbei wurden die vorherig benannten Erwartungen an diese gestellt. Die Ergebnisse davon sind auf Seite [], [] und [] zu finden und bestätigen somit die Vermutungen.

Betrachtet man nun die Einzelnen Implementationen ist zu sehen, dass diese ein annährend Lineares verhalten beim Einfügen der Daten in die Datenstruktur aufweisen. Dies bedeutet, dass alle einfüge Vorgänge hierbei in etwa die gleiche Zeit in Anspruch nehmen. Dies ist im Zweiten Diagramm auf den jeweiligen Seiten sichtbar. Die dort sichtbaren Abweichungen sind zu erklären, indem nicht gewährleistet ist, dass der Benchmark Prozess während der gesamten Ausführzeit ohne Unterbrechung die CPU zugewiesen bekommen hat.

Zudem kann bei der Implementation mit der Double-Linked-List und derer mit den Blöcken noch in Frage gestellt werden, ob die malloc aufrufe alle eine gleiche Geschwindigkeit aufweisen um Speicher zu Allozieren. Um dem entgegenzuwirken wurde der Test mit dementsprechend vielen Daten vollzogen, um einen sinnvollen durschnitt bilden zu können.

In diesem Benchmark wurde versucht das verhalten ein weiteres mahl zu bestätigen, was auf den Seiten [], [] und [] ersichtlich ist. Zudem wurde dieser Versuch gehalten, um zu sehen, ob es einen merklichen unterschied macht, wenn die Warteschlange zu beginn leer ist. Dies sollte bei der Implementation mit Blöcken zumindest theoretisch einen Unterschied machen, da diese mit ansteigender Länge zunehmend weniger Allocationsprozesse der Karte selbst durchführen muss. Dies ist jedoch in den Benchmarks nicht festzustellen. Es lässt sich somit daraus schließen, dass diese zunehmend langen malloc Prozesse einen insignifikanten unterschied im vergleich zur gewonnenen Performance im vergleich zur Implementation mit einer Double-Linked-List aufweist.

Benchmarks Zugriff

Zuletzt wurde in diesem Versuch das Verhalten der Warteschlangen beobachtet, wenn aus diesen Elementen entfernt werden. Hierbei zu sehen ist, dass wie auch beim Einfügen der Elemente dies einen annährend Lineares verhalten aufweist. Auf dem 1. Graf auf Seite [einfügen] kann gesehen werden, dass die Zugriffszeit zwischen Block Implementierung und Array nahezu identisch ist. Zudem verhalten sich die Zugriffszeit Abweichungen wieder so, dass man diese mit der zugewiesenen Zeit der CPU erklären kann.

Fazit:

Zugriffszeit auf Head und Tail. Der Treibende Faktor des Gedankens hinter der Double-Ended-Queue. Wie in den vorrangehenden Kapitel 4 Benchmak gezeigt, erfüllen die Implementierungen der Deque welche im Kapitel 3 Aufbau eingeleitet wurden die Kriterien hinter diesem Gedanken. Eine Zugriffszeit und Einfüge zeit von O(1) wurde hierbei im Kapitel 4 aufgezeigt. Dazu lässt sich sagen, dass die Implementierung mit der Double-Linked-List sehr viel langsamer ist als die andern beiden. Dies kann jedoch damit erklärt werden, da diese tatsächlich nur den Speicher belegt, der auch gebraucht wird und somit für jedes neue Element Speicher allozieren, bzw. für jedes entfernte Element diesen wieder freigeben muss. Somit ist diese Implementierung die Performanteste, wenn es darum geht wenig Speicher für die Warteschlange zu benötigen. Die Implementation aus einem Array hingegen ist die Speicher aufwendigste. Denn falls ein Nutzer diese verwenden möchte, muss dieser einen entsprechend großen Speicher für die Warteschlange zur Verfügung stellen und läuft Gefahr das dieser dennoch nicht ausreichen kann. Die Vielseitigkeit einer Deque die Dynamisch anwachsen kann wie die aus der Double-Linked-List und einen performanten zugriff wie die des Arrays aufweist ist hierbei bei der Implementation mit Blöcken jeweils gegeben. Zwar ist diese im direkten Vergleich mit den anderen jeweiligen Aspekten was die Implementierungen mit Array und Double-Linked-List besonders macht ein wenig hinterher, dafür verbindet diese beide Aspekte miteinander. Somit ist als Fazit zu sagen, dass die Implementation aus Blöcken wahrscheinlich am meisten Anwendung in der Praxis finden wird.

Zugriff und Zugriffszeit auf Head und Tail. Der Treibende Gedanke hinter der Datenstruktur Double-Ended-Queue. Wie in den Benchmarks dieser Arbeit bereits erläutert, sind die drei Implementationen, welche hier vorbereitet wurden, sehr unterschiedlich und doch gleich. Der Anspruch an eine Double-Ended-Queue ist es eine Zugriffszeit O(1) nicht nur auf den Head einer Warteschlange zu gewährleisten, wie dies schon von normalen Warteschlangen getan wird, sondern auch auf den Tail. Wie im Kapitel der Benchmarks bereits belegt, weisen die Implementationen, welche im Rahmen dieser Arbeit entstanden sind, diesen Aspekt auf. Dennoch sind sie unterschiedlich. Dabei wurden drei verschiedene Möglichkeiten eine Deque aufzubauen vorgestellt. Baut man diese mithilfe eines Arrays, so ist diese im Vergleich mit den anderen Implementationen die schnellste, um auf die Daten zuzugreifen. Dies kommt jedoch zu kosten der Fähigkeit dynamisch anwachsen zu können. Somit hat diese Implementation eine Statische Länge und kann volllaufen. Zudem verbraucht Sie im Vergleich zu den anderen Implementationen viel Speicher welcher evtl. nicht benötigt wird, da dieser mit erstellen der Deque bereits reserviert werden muss. Ist die Warteschlange somit nahezu leer, ist die diese im Speicher genau so groß wie als würde sie fast Ihre maximale Kapazität erreicht haben. Um diesem Verhalten entgegenzukommen und dennoch die Funktionalitäten der Etablierten Funktionen zu wahren, wurde als ein weiterer Ansatz eine Double-Ended-Que aus einer Double-Linked-List implementiert. Diese hat den Vorteil, dass sie zu einem Dynamisch anwachsen kann, und somit nicht volllaufen kann. Zudem wird von dieser Implementation nur der Speicher belegt, welcher tatsächlich von der Warteschlange benötigt wird. Vergleicht man jedoch die Performance dieser Implementation mit der des Arrays, wird sehr schnell klar, dass diese im vergleich Langsam ist. Um die Performanten Zugriffs- und Speicherzeiten jedoch zu wahren ist als letztes eine Implementation entstanden, welche den Mittelweg zwischen Speicheffizienz und Performance bietet. Im direkten vergleich mit den anderen jeweilig ein wenig Schlächter, bietet die Implementation aus Blöcken jedoch eine Zugriffs- und Speicherzeit fast wie die des Arrays, mit der Fähigkeit Dynamisch anwachsen zu können. Hierbei wird beim Einfügen direkt für weitere einfüge Prozesse Speicher reserviert. Weiterer Speicher wird jedoch erst reserviert, wenn der Aktuelle Speicher in jeweils eine Richtung voll ist. Da diese Implementation somit performant, dynamisch und relativ Speicher freundlich ist, ist anzunehmen, dass diese Implementation, die am weitesten verbreitet bzw. verwendete ist.

Daher der Grundgedanke der Deque. Im Gegensatz zur Queue kann die Deque jedoch sowohl am Head, als auch am Tail Objekte entgegen nehmen und diese wieder ausgeben, was den Hauptunterschied zur klassischen Warteschlange darstellt. Somit liefert die Deque in der Standard-Implementierung für sowohl den Head als auch den Tail dieselben Funktionen, welche bei der normalen Warteschlange nur einmalig existieren. So hat der Head eine Push Funktion, welche Daten der Warteschlange anhängt, aber auch der Tail bietet diese Funktionalitäten. Genauso haben beide eine Pop Funktion, welche eben diese der Warteschlange wieder entnimmt. Optional und häufig bei umfangreicheren Implementierungen zu finden ist eine Peek Funktion, welche es ermöglicht die Daten am Head, oder Tail abzurufen ohne diese der Warteschlange zu Entfernen. Durch eine derartige Implementierung ist für den Head und Tail eine Zugriffsgeschwindigkeit von O(1) gewährleistet, wofür die Datenstruktur jedoch an Komplexität zunimmt. Hierbei zu beachten gilt, dass eine Deque nicht fundamental besser als die klassische Warteschlange ist, jedoch in spezifischen Anwendungen andere Datenstrukturmodelle an Performance übertrifft. Eines der besten Beispiele wofür eine Deque verwendet werden kann, ist ein Work-Stealing-Algorithm worauf im späteren verlauf unter \ref{WorkStealing} genauer eingegangen wird.