Technische Universität

BERGAKADEMIE FREIBERG

Fakultät: Mathematik und Informatik

Studiengang: Angewandte Informatik

Double-ended-que

Bearbeiter: Naumann Marco

Martrikelnummer: 64645

.....

Unterschrift

Inhaltsverzeichnis:

Einleitung:

In dieser Seminararbeit wird sich im Rahmen des Seminars „Datenstrukturen in C“ mit der Datenstruktur[f1] „Double-ended-que“ auseinandergesetzt.

Wie ist der Aufbau und die Funktionsweise einer Double Ended Que…

Wie ist meine Arbeit aufgebaut

Was ist eine Double-ended-que? (Kurz, Deque)

Eine Deque ist eine unterform der Klassischen Warteschlange (Engl. Queue). Während eine normale Warteschlange von nur einer Seite befüllt, und von der anderen geleert werden kann, stellt die Deque einen Zugriff auf sowohl den Anfang (Enque) als auch das Ende (Dequeue) zur Verfügung.

Somit folgt die Deque nicht dem klassischen „First In First Out“ (FIFO)[f2] Prinzip, welches man im Allgemeinen von einer Warteschlange erwartet.

Dies schränkt zwar die Applikationen der Deque ein, jedoch eröffnet dies aber auch einzigartige Verwendungsmöglichkeiten und Verringert z.B. Zugriffszeiten und rechen Leistungen benötigt zur Verwaltung der Deque.

Grundlagen: [q4]

Commented [MN1]: NICHT ausführlich genug, jeder begriff muss erklärt werden!

Um zu verstehen wie eine Deque funktioniert, sollte man anfänglich erstmal die Warteschlange selbst unter die Lupe nehmen.

In der Informatik stellt die Klassische Warteschlange nichts anderes als eine Ansammlung von Objekten dar, welche mit gewissen Regeln verwaltet werden.
Eine Warteschlange hat, klassischer weise, 2 enden und beide haben einen eigenen Namen, und Funktion.

Zu Beginn gibt es das Ende an welche Elemente in die Warteschlange übergeben werden können, und somit der Warteschlange hinzugefügt werden. Dieser wird im Allgemeinen als „Back“, „Tail, oder „Rear“ (auf Deutsch, Hinten) bezeichnet. Die andere Seite der Warteschlange wird verwendet, um Elemente aus dieser zu laden, und somit aus der Warteschlange zu entfernen. Im Allgemeinen wird dieser als „Head“, also Kopf, der Warteschlange bezeichnet.

Dieser Simple Aufbau ist im end Effekt der Warteschlange nachempfunden, welche wir Menschen z.B. an einem Kiosk bilden an denen wir in den heißen Sommermonaten Eis kaufen wollen.

Während die vorrausgehenden Regeln dafür genüge sind eine Datenstruktur als Queue zu definieren, werden dieser meist zusätzliche Funktionalitäten mitgegeben. Eine der Solchen zusätzlichen Funktionalitäten wäre z.B. die „Peek“ Funktion, mit welcher das erste Element (also, das am Kopf der Warteschlange befindliche Objekt, das was als nächstes geladen werden würde), zu laden, ohne es der Warteschlange zu entfernen.

Commented [MN2]: Welche regeln!

Die Regeln, welche eine Warteschlange zu Grunde liegen sorgen dafür, dass es sich um eine FIFO[f2] (First in First Out) Daten Struktur handelt.

Strukturell kann eine Queue als Datenstruktur unbegrenzt groß im Speicher wachsen und ist lediglich von der Speichergröße selbst begrenzt. Jedoch, je nach Implementierung kann dies von der Norm abweichen. Somit kann zwar die Datenstruktur Theoretisch unbegrenzt anwachsen, jedoch werden in der Praxis Implementierungen wie der „Ringpuffer“[f3] genutzt, welche eine Fixierte maximal Größe besitzen und mit den Funktionalitäten einer Deque vergleichbar sind.

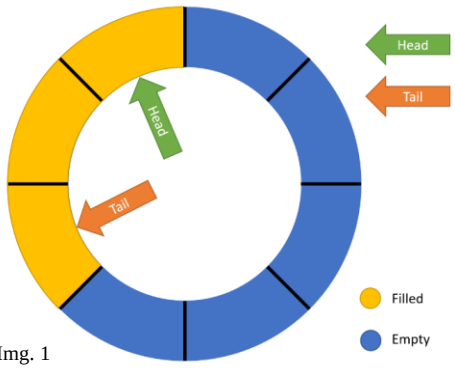
Aufbau der Deque:

Der Wesentliche Unterschied zwischen den Klassischen Warteschlangen und eine Double-Ended-Que ist, dass bei einer Deque die Beschränkungen, dass Queues nur einseitig gefüllt, und einseitig entleert werden können wegfällt.

Eine Deque kann sowohl am Kopf als auch am Tail Objekte entgegennehmen, oder auch wieder entfernen lassen.
Somit kann man die Datenstruktur in beide Richtungen konstruieren, und de-konstruieren, was dafür sorgt, dass die Zugriffszeiten auf z.B. das zuletzt angehängten Elements dieselbe ist, wie das aller erste Element, welches der Warteschlange hinzugefügt wurde.
Im Austausch dafür, haben Klassische Deque Implementierungen jedoch keinerlei Zugriffsmöglichkeiten auf Elemente innerhalb der Deque, welche nicht Head, oder Tail sind.

Dafür wichtig, ist, dass die Position des Kopfes, sowie des Tails bekannt ist. Dies kann in C z.B. mit einem Pointer auf eben diese bewerkstelligt werden.
Die einzelnen Positionen der Queue dann wiederrum kennen Ihre Nachbarn, wie man dies schon in der Datenstruktur Double-Linked-List finden kann.

<u>Deque mit Fixierter Länge</u>



Img. 1

Ist die Länge der Deque limitiert wie im Bsp. Links auf 8 Objekte, dann kann die Deque zwar nicht dynamisch anwachsen, jedoch kann man in diesem Beispiel sehr anschaulich das Dynamische verhalten des Head und Tail Pointers betrachten.

Wenn keine Objekte in der Deque gespeichert sind, zeigen Head- und Tail-zeiger auf „NULL“, woraus man lesen kann, dass die Datenstruktur leer sein muss, unabhängig was tatsächlich im Speicher aktuell steht. Mit dem Einfügen des ersten Objekts werden nun die Pointer definiert und in die Struktur gesetzt. Solange nur ein Objekt gespeichert ist, zeigen Head- und Tail-zeiger auf dasselbe Objekt, was die einzige Definierte stelle ist.

Fängt man nun an die Struktur mit weiteren Daten zu füllen, so legt man die Daten in den dafür vorgesehenen Speicher ab und definiert diesen somit, während der Head Zeiger immer auf das zuletzt angehängte Objekt von vorn, und der Tail Zeiger auf das zuletzt angehängte Objekt von hinten Zeigt. Die Zeiger Operationen müssen vor dem Definieren des Speichers erfolgen, damit, wenn man versucht mehr Objekte in die Queue zu legen, als Ihre Kapazität erlauben würde ein Fehler erkennbar ist. Versucht man dies nämlich, dann würde bei dem Versuch den Head bzw. Tail Zeiger zu bewegen

auffallen, dass diese versuchen auf dieselbe Position zu zeigen, was als Indikator genommen werden kann, dass die Queue ihre Maximale Kapazität erreicht hat und zum Abbruch der Funktion führt.

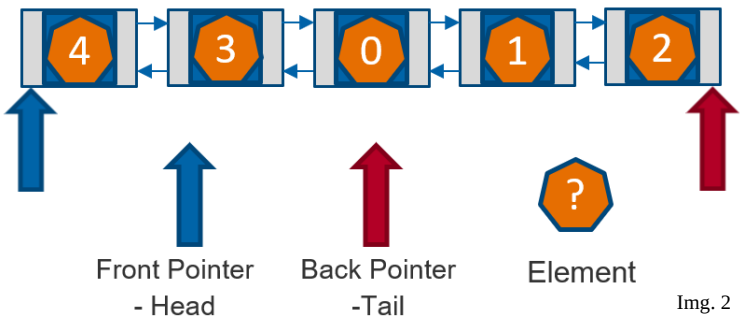
Beim Laden der Objekte nun aus der Queue, werden nacheinander am Head oder am Tail pop Funktionen ausgeführt, welche den Head, bzw. Tail Zeiger nach dem Laden eines Objektes auf eine vorherige Position zurücksetzt und der Speicher einfach in einen Undefinierten zustand verbleibt. Das Interessante daran ist, es ist nicht notwendig die einfüge Reinfolge einzuhalten, und man kann z.B. alles von Hinten auslesen, von Vorn, oder Beliebig.

Somit ist es möglich z.B. permanent ein Objekt im Speicher zu haben und durch das Einfügen eines weiteren von vorn, und das Entfernen eines von hinten, kann man nun die Pointer, und den benutzten Speicher „im Kreis“ durch diesen wandern lassen. Dieses Verhalten wäre bei einer Deque von einer nicht statischen Größe ein verhalten, worauf man achten muss.

Deque mit Dynamischer Länge

Da das Verhalten der Pointer nun bekannt ist, kann man sich an eine Dynamische Implementierung machen. Jedoch, dynamisch ist nicht gleich dynamisch und im Rahmen dieser Seminararbeit sind zwei Implementierungen vorbereitet.

1. Deque – mit Linked-list
   Die erste Implementierung ist die, objektiv, einfacher zu verstehende.
   Das Prinzip der Linked-list bekannt, speichert man den Kopf und den Tail der Liste in einer Control Funktion ab und hat nun somit eine Double-Ended-Que, ohne viel programmieren zu müssen.



Img. 2

   Jedes Element kennt die Adressen der Nachbar Elemente und der Head und Tail sind bekannt. Erzeugt man nun ein Neues Element, am Kopf z.B. dann alloziert man neuen Speicher für das Element, weist dem Neuen Kopf den Alten Kopf in seinem Back Pointer zu, und gibt dem alten Kopf den neuen Kopf als Front Pointer.
   Der Front Pointer des Neuen Kopfes zeigt auf NULL, was das vordere ende der Queue signalisiert und man überschreibt den Kopf in der Control Funktion mit dem Neuen Kopf und hat somit ein neues Element der Liste hinzugefügt.

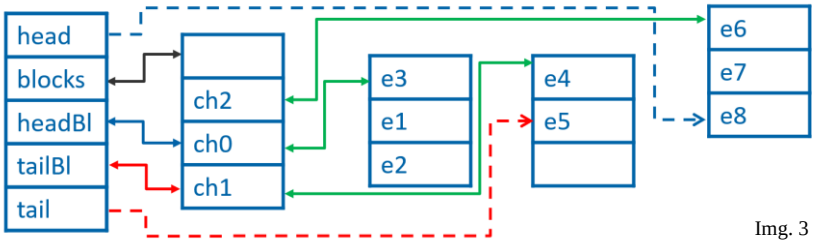
Analog würde dies dann auch mit dem Tail funktionieren und man erhält somit eine Datenstruktur, welche Potenziell auf eine unbegrenzte Größe anwachsen kann, oder kleiner wird, wenn ein Element aus ihr entfernt wird.
Die Limitierenden Faktoren hierbei sind lediglich der verfügbare Speicher, sowie die Zeit die es benötigt ein neues Speicherelement zu allozieren.

Um nicht für jedes neue Element neuen Speicher jedes Mal aufwändig neu zu allozieren, kann man mit einem Mal direkt Speicher für Mehrere Speichervorgänge allozieren.

2. Deque – mit Blocks

Der Grundgedanke hierbei ist, dass man sich Allocations-zeit spart, indem man platz für mehrere Elemente mit einem Allocations-prozess zur Verfügung stellt.
Hierzu benötigt man ein Control-Element, eine „Karte“ in welchem die Allozierten Speicherbereiche für die Elemente abgelegt werden und die Speicherbereiche Selbst.

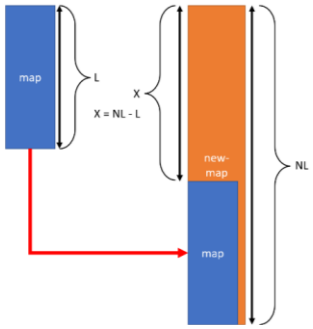
Das Control-Element kann man recht kurzhalten und beinhaltet lediglich Zeiger.
Der Zeiger auf die „Karte“ wird benötigt, um im Speicher die Element-Blöcke wieder zu finden und ist nichts anderes als ein Pointer auf ein Array voller Pointer. Innerhalb dieser Karte legt man nun zwei weitere Zeiger, den „HeadBlock-/TailBlock-Pointer“. Diese werden benötigt zur Orientierung innerhalb der Karte und zeigen immer auf das Karten-Element wo Ihr jeweiliger Head-/Tail-Pointer zu finden ist.
Die Head- und Tail-Pointer selbst müssen natürlich auch gespeichert werden.



Img. 3

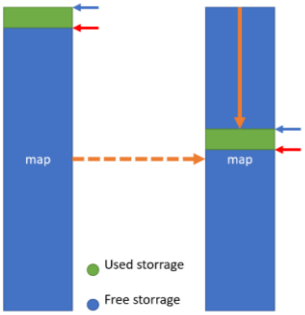
Mit diesem Groben Aufbau muss man nur noch Pointer bewegen und evtl. Speicher Allozieren, um das Verhalten einer Deque darzustellen.
Erzeugt man also ein solches Objekt, wird das Control-Element zur Verfügung gestellt, welches auch eine Karte mit einer anfänglichen Größe mitbringt.
Fügt man nun Objekte in die Deque hinzu, wird je nachdem wie viele Elemente sich schon in dieser befinden entweder ein neuer Block erzeugt, oder ein bestehender verwendet, in welchem dann das neu hinzugefügte Objekt gespeichert wird.
Fügt man z.B. ein Element am Kopf hinzu, so wird, solang der Block, in dem sich der Head Pointer befindet, nicht schon nach oben hin gefüllt ist einfach der Head-Pointer um eine Position nach oben bewegt, und das Element dort im schon vorbereiteten Speicher abgelegt.

Ist der Block jedoch voll, muss erst der HeadBlock-Pointer um eine Position verschoben werden, sowie ein neuer Block Alloziert werden, und der Headpointer auf die, in dem Fall unterste Position gelegt werden, bevor das neue Element dann abgelegt werden kann.
Durch das Anpassen der Block-größen an das zu erwartende Problem, kann man somit wieder dynamisch die Deque anwachsen lassen.
Sollte es jedoch zu dem Fall kommen, dass die Karte selbst voll ist, also das der HeadBlock- oder TailBlock Pointer die Erste oder Letzte Position erreicht hat, muss man hier nur die Karte vergrößern ohne, dass die Daten, welche in den Blöcken selbst gespeichert sind, bewegt werden müssen.



Img. 4

Ein weiteres Problem, welches jedoch abgefangen werden muss, ist, wenn man wiederholt Objekte von Vorn einer vorhandene Queue anhängt, und mit gleicher Geschwindigkeit diese von hinten wieder weck nimmt. Ohne das man sich mit diesem fall auseinander setzt, würden konstant nach oben hin neue Blöcke erzeugt, der Speicher selbst jedoch leer bleiben und das Obere ende der Karte wiederholt erreicht werden, was ein re-sizen dieser zur folge hat nur um noch mehr Leeren Speicher zu erzeugen.



Img. 5

Eine simple Lösung für dieses Problem ist es, die vorhandene Karte einfach neu auszurichten, wenn der tatsächliche füllstand der Karte unter einem gewissen wert liegt.

Während es dafür verschiedene Ansatz Möglichkeiten gibt, um dies zu bewerkstelligen, muss man die Performance der Deque jedoch im Blick behalten und vergleichen.

Fachwortverzeichnis:

f1: [q1] *Datenstruktur* „In der Informatik und Softwaretechnik ist eine Datenstruktur ein Objekt, welches zur Speicherung und Organisation von Daten dient. Es handelt sich um eine Struktur, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen."

f2: [q2] *FIFO* „First In – First Out, ist gleichbedeutend mit „First come, first served" und bezeichnet jegliche das Verfahren der Speicherung, bei denen diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden."

f3: [q3] *Ringpuffer* „Warteschlangen sind häufig als Ringpuffer mit je einem Zeiger auf Anfang (In-Pointer) und Ende (Out-Pointer) implementiert. Die Besonderheit des Ringpuffers ist, dass er eine feste Größe besitzt. Dabei zeigt der In-Pointer auf das erste freie Element im Array, das den Ringpuffer repräsentiert, und der Out-Pointer auf das erste belegte Element in dem Array. Im Unterschied zum Array werden die ältesten Inhalte überschrieben wenn der Puffer voll ist, um weitere Elemente in den Ringpuffer ablegen zu können. Eine Implementierung des Ringpuffers sollte für den Fall, dass der Ringpuffer voll ist, entweder einen Pufferüberlauf signalisieren oder zusätzlichen Speicherplatz bereitstellen. In anderen Fällen kann das Überschreiben alter Elemente der Warteschlange und damit der Datenverlust gewollt sein."

F4: [q5] *Double-Linked-List* Eine verkettete Liste, in welchem jedes Element einen Zeiger auf das Vorrangehende und das Nachfolgende Element hat.

Quellenverzeichnis:

q1: 20. September 2021 - https://de.wikipedia.org/wiki/Datenstruktur

q2: 20. September 2021 - https://de.wikipedia.org/wiki/First_In_%E2%80%93_First_Out

q3: 21. September 2021 - https://de.wikipedia.org/wiki/Warteschlange_(Datenstruktur)
Abs.: Implementierung als Ringpuffer

q4: 21. September 2021 - https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

q5: 21. September 2021 - https://de.wikipedia.org/wiki/Liste_(Datenstruktur)

Abs.: Doppelt verkettete Liste

Selbständigkeitserklärung:

Hiermit versichere ich, dass die vorliegende Seminararbeit selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Seminararbeit angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus anderen Schriften entnommen wurden sind, habe ich als solche kenntlich gemacht.

Ort, Datum: Unterschrift: