Ugeaflevering 8g

Niels Abildskov (vzn250), Niklas Marcussen (gwv160)

2. december 2023

1 Introduktion

Denne rapport omhandler implementering og test af *cycliske køer*. En vigtig egenskab ved cycliske køer er, at når beholderen[1] er fyldt, vil det næste element, der tilføjes, overskrive det ældste element i køen[2], hvilket gør dem velegnede til cirkulær buffering af data. Dette gør det muligt at opretholde en konstant mængde hukommelsesforbrug, hvilket kan være afgørende i indlejrede systemer eller applikationer med begrænset hukommelse. Til sidst i rapporten vil vi diskutere hvorledes vores implimentering dækker hele specifikationen og om implimentationen vil returnere en ubehandlet *exception*. Derudover vil vi diskutere fordele og ulemper af vores imperative tilgang i modsætning til en funktionel tilgang.

2 Programbeskrivelse

Vores modul, *CyclicQueue*, implementerer en cyklisk kø som en singleton-instants og definerer et sæt operationer som den skal understøtte. Disse operationer er skrevet i det imperative programmeringsparadigme. Selve køen er defineret ved to mutable option<int>'s, first og last, samt et array der holder option<Value>'s, hvor type Value = int. Følgende funktioner er deklareret som vores interface (de indrykkede lister beskriver hvordan vi forventer funktionen skal opføre sig givet en bestemt input-type):

- val create : n : int -> unit: Lav en ny cyclisk kø med længde n.
  - $n \leq 0$: Kast en exception da man ikke kan allokere et array med ikke-positiv størrelse.
  - $n > 0$: Allokér q, og sæt både first og last til None.
- val enqueue : e : Value -> bool: Indsæt nyt element bagerst i køen, og returner true hvis der var plads.
  - length () < q.Length: Indsæt elementet og returnér true.
  - q.Length <= 0 or length () >= q.Length: Returnér false.
- val dequeue: unit -> Value option: Træk det forreste element ud af køen. Hvis denne er tom returneres false.
  - not isEmpty (): Returnér første element
  - q.Length <= 0 or isEmpty (): Returnér None da køen er tom.
- val isEmpty: unit -> bool: Tjek om køen er tom eller ej.
  - q.Length > 0: Returner true hvis $first.Value = last.Value$
  - q.Length <= 0: (create ikke kaldt endnu) returner 0
- val length: unit -> int: Returnerer længden af køen.
  - q.Length <= 0: Returner 0 da create ikke er kaldt endnu.
  - q.Length > 0: Returner længden ved Math.Abs (last − first) + 1
- val toString : unit -> string: Returnerer køens elementer, repræsenteret som en (komma-separaret) string.

[1] Kunne eksempelvis være et array der bruges til at repræsentere køen.
[2] Med "det ældste element"menes der at så snart et element er blevet dequeue'et, kan det overskrives

- q.Length <= 0: Returnér en tom string.
- q.Length > 0: Returnér listen af elementer fra first til last, separaret af ", ".

Til disse definerer vi tre private hjælpe-funktioner, navnlig: val nextIndex : x : int -> int, val isValidState : unit -> bool og val isFull : unit -> bool der henholdsvis

1. Beregner det næste indeks i array'et, da first og last begge skal køre cirkulært over array'et. Denne kunne implementeres som $(x + 1) \mod q.length$.
2. Tjekker om create er kaldt[3] ved $q.length > 0$.
3. Tjekker om køen er fuld.

3 Afprøvning

Til afprøvningen af vores implimentation har vi lavet en *blackbox test suite*, der tester alle funktionerne beskrevet i den udleverede signaturfil. Dette gør vi igennem specifikke test funktioner der bruger assert funktionen til at returnere true hvis den testede funktion fungerer korrekt. Vi bruger en funktion runTests () der igennem en hjælpefunktion testAndReport som tager et testnavn og en testfunktion som parameter. Denne hjælpefunktion bliver kaldt en gang for hver test funktion, hvor den, hvis testen returnerer true printer den "*<pågældende test>: Passed*". Hvis alle vores testcases består, vil den til sidst printe *All tests passed*. Eftersom vi tester vores funktioner på; ikke initialized køer, tomme køer, ikke tomme køer og næsten fyldte køer sikre vi at dække alle input-partitions og alle specifikationerne. Desuden sikrer vi også at der ikke sker nogle *unhandled exceptions* igennem vores funktion testExceptionHandling og en failwith hvis man prøver at lave en kø med en størrelse der er ≤ 0.

4 Diskussion

Nogle grunde til, at man kunne vælge at implementere en cyclisk kø i modsætning til en generél kø indebærer

- Kun én allokering: Når man først har oprettet sin kø-struktur, har man hvad man skal bruge. Når der indsættes nye elementer vil man eksempelvis ikke skulle oprette en ny knude/nyt element til en linked-list, hvilket en generel kø ville være betinget af. Dette er godt for bl.a. high-performance kode eftersom (de-)allokeringer ofte tager lang tid.
- Hurtig enqueue/dequeue: I cycliske køer skal man blot holde styr på 2 pointers/indexes ind til ens data, hvilket er hurtigere end hvis man ville implementere en generel kø vha. arrays; i dette tilfælde ville man skulle implementere enqueue ligesom en push-operation på en stack. Dequeue ville så skulle finde det "sidste"element ($\neq NIL$) i array'et ($O(n)$), og returnere det. Hvis dequeue derimod skulle køre i $O(1)$ ville enqueue køre i $O(n)$ da elementer så skulle indsættes bagfra. Med cycliske køer får vi både enqueue og dequeue til at køre i $O(1)$.

Med hensyn til den egentlige implementering, ville man også kunne implementere cykliske køer funktionelt. En funktionel implementering ville givet vis bytte det lave abstraktionsniveau for mere kompakt kode. Det kunne bl.a. indebære, at vores toString funktion ville være rekursiv frem for iterativt. Dertil ville vi skulle repræsentere vores cycliske kø som f.eks. (Value option[] * int * int) og returnere et sådan "objekt"til vores caller, da vi ikke ville kunne benytte mutable variable, som vi gør det i vores imperative løsning. Generelt er imperative løsninger ofte mere intuitive og nemmere at implementere for små opgaver, men mister overskueligheden som projektets størrelse stiger. Derudover har funktionelle sprog en stor fordel pga. de ikke har et koncept af *state*. Hvis et stort program har en global state, vil alle linjer i programmet potentielt kunne ændre denne, hvilket risikerer at gøre programmet uforudsigeligt og tilbøjeligt til at indeholde fejl.

[3] I nogle sammenhænge kunne dette kaldes *undefined behaviour*, og derfor ikke være noget vi burde tjekke for.

5 Konklusion

Vi har i denne rapport har behandlet design og test af cycliske køer. Vi implementerede en cyklisk kø med fokus på grundlæggende operationer som enqueue og dequeue, samtidig med at vi sikrede et konstant hukommelsesforbrug. Vores tilgang var imperativ. I vores undersøgelse anvendte vi metoder til testning, der sikrede, at vores cykliske kø opfyldte alle de nødvendige krav og håndterede alle potentielle fejltilstande effektivt. Den anvendte teststrategi fokuserede på at vurdere køens funktionalitet under forskellige forhold, herunder dens evne til at håndtere tomme, ikke tomme og næsten fulde tilstande, samt dens respons på uventede inputs. Denne tilgang bidrog til at bekræfte kodens korrekthed.
Samlet konkluderer vi cycliske køers effektivitet og nytte i situationer, hvor hurtige operationer og hukommelseseffektivitet er afgørende. Selvom den imperative tilgang viste sig effektiv, kunne en funktionel tilgang overvejes for yderligere forbedringer i større systemer.