

Lab 1 – FIFO med cirkulär implementation

Objektorienterad programmering i C++

Dynamisk minneshantering och smarta pekare
'pre/postconditions'.

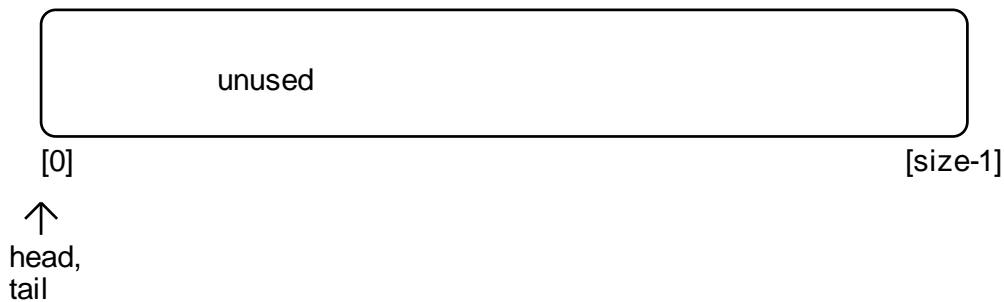
Lab 1 – FIFO med cirkulär implementation

En av de klassiska Abstrakta Datatyperna är Kö. En kö är en FIFO-struktur, ”*First In First Out*” (precis som i en civiliserad MacDonalds-kö). Nya element stoppas in längst bak i kön (tail) och det äldsta elementet plockas ut längst fram (head). I datasammanhang används köer som buffertar mellan en ‘dataproducent’ och en ‘datakonsument’. Om dessa producerar respektive konsumerar data i samma takt behövs i princip ingen buffert men i verkligheten är det vanligt att olika enheter och processer har skiftande snabbhet. En buffert där data kan lagras tillfälligt kan då utjämna tillfälliga skillnader i processhastighet. T.ex. används buffertar i samband med tangentbordsinmatning, nätverksinterface, seriekommunikation, utskriftshantering och filhantering. För en ADT Kö kan lämpliga operationer vara

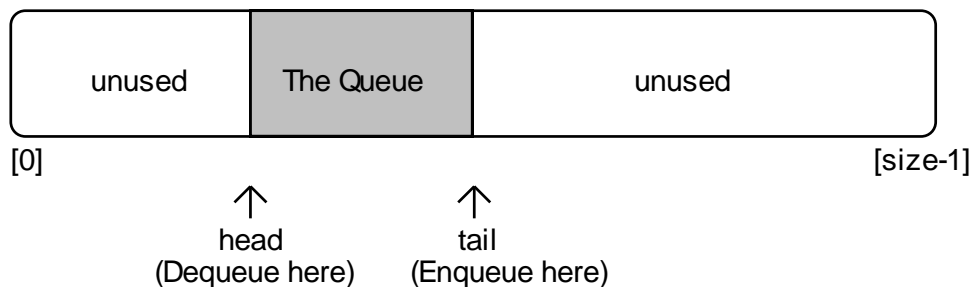
- enqueue - stoppa in nytt element längst bak i kön
- dequeue - ta ut elementet längst fram i kön
- length - aktuellt antal element i kön
- full - kön full ?
- empty - kön tom ?
- capacity - kön maximala storlek?

En kö kan implementeras på många sätt. I många varianter använder man en array för att lagra elementen i kön. Av effektivitetsskäl ska man aldrig flytta element i arrayen när element läggs till eller tas bort. I stället använder man sig av två ‘heltals-pekare’, head och tail (typen int) som innehåller index i arrayen för köns första respektive sista element. Dessa index rör sig framåt i arrayen allt eftersom kön växer respektive krymper. Följande figurer beskriver implementationen av en kö i en *cirkulär* array.

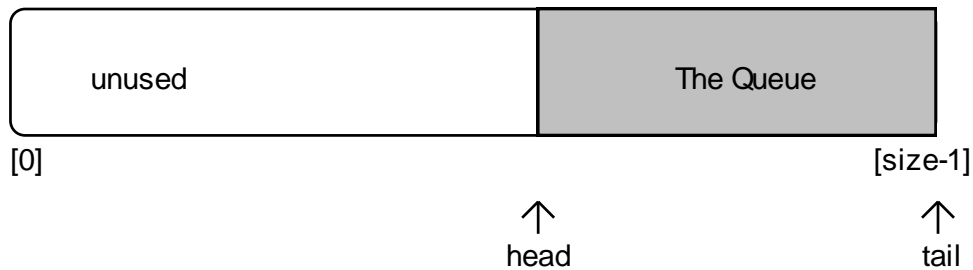
Tom kö:



Kön efter en viss tid:

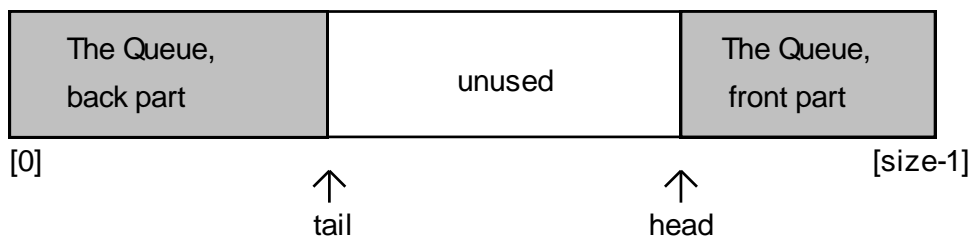


Kön när arrayens övre gräns är nådd är:



I detta läge låter man kön *internt* bli cirkulär så att nästa enqueue sker i början av den lediga delen på arrayen. För att få head och tail att 'hoppa' från size-1 till 0 vid uppräknings kan en if-sats eller modulus-operatorn % användas (vilken tror du går snabbast?).

Kön efter ytterligare en tid, "wrapped state":



Uppgift

Implementera klassen Queue. RAII-konceptet med `unique_ptr` ska användas för den interna arrayen. Ett interaktivt menystyrt testprogram ska skrivas i form av en klass där de olika operationerna på en Queue motsvaras av medlemsfunktioner i klassen. *Förslag* till testklass:

```
class TestApp {
private:
    smart pointer object to hold the dynamically Queue object
    bool done;

    // Main menu
    void showMenu();
    char getMenuOption();
    void doMenuOption(char option);

    // Menu-alternativ
    void createQueue();           // Skapar en kö av önskad längd
    void enqueue();               // Frågar efter tal
    void dequeue();              // deleting integer, printing out
    void isEmpty() const;
    void isFull() const;
    void nrElements() const;

public:
    TestApp();
    void run();
};
```

Klassdefinition för klassen Queue:

```
typedef int Type;           // Make Type an alias for int
const int QSIZE = 10;      // Default Queue size

class Queue {
private:
    smart pointer object to hold the dynamically allocated array

    int head;    // index in array for head
    int tail;    // index in array for tail
    int nElem;   // nr of items in queue
    int maxElem; // max capacity of queue
public:
    Queue(int n = QSIZE); // n = max capacity of queue
    /* Pre:    -
    Post: A Queue object with capacity for n data-
    items is created. */

    void enqueue(Type elem);
    /* Pre:    The Queue has not reached its maximum capacity.
    Post: elem is inserted as the newest element in
    Queue. */

    void dequeue(Type &elem);
    /* Pre:    The Queue is not empty.
    Post: elem contains the oldest element in Queue-
    pre and elem is not contained in Queue. */

    int length() const;
    /* Pre:    -
    Post: Number of elements in Queue is returned. */

    bool full() const;
    /* Pre:    -
    Post: If Queue has reached its maximum capacity
    true is returned, else false is returned. */

    bool empty() const;
    /* Pre:    -
    Post: If Queue is true is returned,
    else false is returned. */

    int capacity() const;
    /* Pre:    -
    Post: Maximal number of elements that the Queue can
    hold Queue is returned. */
};
```

`Pre` är en förkortning av Pre-condition och `Post` är en förkortning av Post-condition. Post condition beskriver det tillstånd som ska gälla omedelbart efter det att operationen ifråga är exekverad, under förutsättning att Pre-condition var uppfyllt vid början av exekveringen. Man kan se dessa villkor som ett kontrakt: *om Pre är uppfyllt före exekveringen av en operation så ska Post garanteras efter exekveringen*. Detta innebär att operationer som har Pre-conditions måste föregås av en kontroll. Speciella operationer för sådana kontroller ska alltid finnas (t.ex. full och empty). Om en operation exekveras utan att Pre-condition är uppfyllt så behöver inte Post gälla, resultatet är då odefinierat. Observera att Pre- och Post-conditions inte beskriver hur implementationen ska göras i detalj utan endast vad som ska vara uppfyllt före exekveringen och vad resultatet ska vara efter exekveringen. Väl skrivna Pre- och Post-conditions är därför tillräckliga för att utföra en implementation. Beteckningen Queue-pre i ett Post-condition betyder tillståndet i Queueobjektet som det var *före* exekveringen.

Krav på lösningen

- Queue ska använda en smart pekare till den dynamiskt skapade interna arrayen
- Körning av testprogrammet görs genom att ett TestApp-objekt skapas och dess run-funktion anropas.
- TestApp ska använda en smart pekare till ett dynamiskt skapat Queue-objekt
- `TestApp::createQueue` ska fråga efter önskad kölängd och skapa motsvarande Queue-objekt.
- Lösningen ska vara fri från minnesläckor.
- Innan `TestApp::createQueue` har anropats finns inget Queue-objekt och då ska inte de övriga menyalternativen i TestApp inte vara tillgängliga.
- `TestApp::run()` ska köra meny-loopen tills man väljer att avsluta.
- Användare av testprogrammet ska kunna
 - Skapa en kö med önskad storlek (upprepade gånger).
 - Lägga till ett angivet element (enqueue).
 - Ta bort element (dequeue). Värdet ska fångas upp av testprogrammet och skrivas ut.
 - Testa om kön är tom respektive full.
 - Visa antalet element i kön.
 - Visa köns maximala storlek.

Kom ihåg att alltid kolla pre-conditions för Queue i TestApps funktioner!

- Dina källkodsfiler ska vara kommenterade. Alla filer i lösningen ska ha en inledande kommentardel där filnamn, uppgift och ditt egen namn står.