

TDDD86 – Laboration #6

7 november 2024

I den här uppgiften ska du implementera två datastrukturer: en vektor och en prioritetsskö. Du ska använda dem i en diskret händelse simulering (discret event simulation) för att bättre analysera och förstå ett problem kopplade till hållbar utveckling. Bägge datastrukturerna är template-baserade. Simuleringen kommer att representera utvecklingen av fiskbeståndet i ett område med begränsade resurser. Modellen är enkel men illustrerar vikten av att utveckla och adoptera effektiva datastrukturer för att kunna analysera komplexa system. Flera faktorer påverkar fiskbeståndet, från vattentemperaturen till fiskskörden. Att reglera fiskskörden för att uppnå ett hållbart fiskbestånd är inte enkelt och vi behöver alla verktyg som kan hjälpa oss för att bättre förstå konsekvenserna av fiskskörden. Filerna du behöver för att komma igång med denna labb kommer att finnas som `labb6.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 Lab 6 redovisning``` och en `git push`. Se till att filerna [answers.txt](#), [MyVector.h](#) och [MyPriorityQueue.h](#) är med. Informera sedan din assistent genom att maila honom/henne.



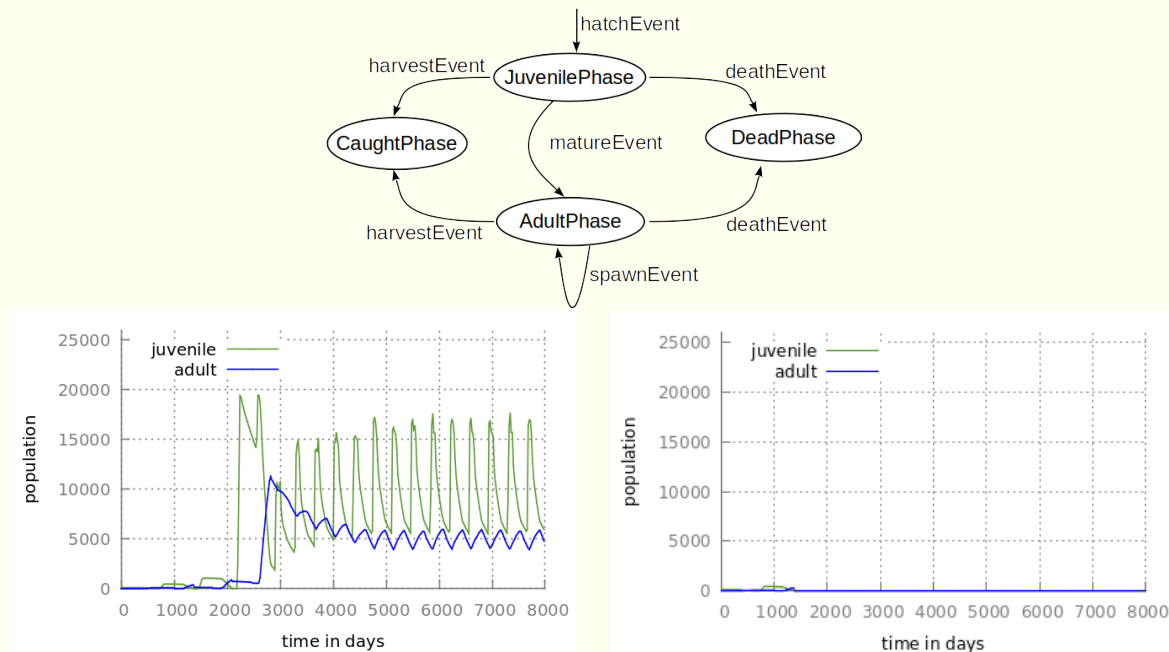
Figur 1: Kollapsen av torskbeståndet utanför den Kanadensiska östkusten 1992 [1]

Komplexa system och hållbarhet

En hållbar fiskskörd kräver en avancerad förståelse av komplexa system. Ett känt exempel där icke-anpassade fiskskörd policyer ledde till negativa och hårda konsekvenser är överfiskandet av torskfisk utanför den kanadensiska östkusten. Där har överfiskandet av torsk mellan 1970 och 1992 resulterat i att torskbeståndet kollapsade med påtagliga ekologiska, sociala och ekonomiska konsekvenser. Simulering är ett verktyg som kan vara till stor hjälp för att kunna resonera kring, och analysera, sådana komplexa system. I den här labben ska du jobba med en enkel diskret händelse simulering av hur fiskskörd (hur mycket fiskas, hur ofta, hur stora fisk) kan påverka hållbarheten av fiskbeståndet.

Metoden `run()` i klassen `Simulation` i den tilldelade koden ligger i hjärtat av denna diskret händelse simulering. Händelser läggs pushas in i en prioritetsskö. När det är tid för en händelse att processas, plockas den från prioritetsskön och exekveras. Detta kan leda till att nya händelser pushas in i kön. Det här fortsätter så länge kön inte är tom. Vi kan också avbryta simuleringen som vi gör i labben med en `SIMULATION_HORIZON` (se metoden `run` i klassen `Simulation`).

I denna förenklade modellen börjar simuleringen med ett antal ägg (se `main` metoden). Ett ägg kläcks (hatches) och ger en juvenil fisk som kan bli vuxen. En fisk kan dö eller bli fångad. En vuxen fisk kan ge ägg varje år vid



Figur 2: Möjliga fisk faser och simulerings resultat som visar att fiskneståndet kan kollapsa på grund av överfiskning.

en specifik säsong (se Figur 2). Händelserna läggs i simuleringens prioritetsskö och ordnas baserat på den tid de ska processas eller exekveras. När en händelse exekveras kan den resultera, beroende på händelsen, i att nya ägg ska kläckas, att en fisk ändrar fas, att fiskskörd börjar, etc. Vi listar några händelser här:

- *hatchEvent*: En ägg kläcks och resulterar i en juvenil fisk. Fiskens beräknat livslängd beror på antalet fisk individer som konkurrerar om mat. En juvenil fisk förväntas bli vuxen efter en viss tid. Beroende på den beräknade livslängd och den beräknade individuell mognadstid (maturation time) läggs en *matureEvent* eller en *deathEvent* till prioritetsskön.
- *matureEvent*: En juvenil fisk blir vuxen. Ett förväntad datum där fisken lägger ägg beräknas. All vuxen fisk lägger ägg under två specifika månader varje år. Beroende på förväntad livslängd och datum till nästa gång fisken ska lägga ägg läggs en *deathEvent* eller en *spawningEvent* för den specifika fisken.
- *spawningEvent*: Ett antal ägg läggs. Antalet ägg beror på antalet vuxna fiskar som finns. Om det finns för få fiskar läggs det inga ägg. För varje ägg, en förväntad tid till att ägget kläcks beräknas och en *hatchEvent* läggs till prioritetsskön. En förväntad tid till nästa gång ägg ska läggas beräknas. En *deathEvent* eller en *spawningEvent* läggs till prioritetsskön beroende på vilken händelse behöver exekveras först för den specifika fisken.
- *deathEvent*: En juvenil eller vuxen fisk dör efter en individuell beräknat livslängd.
- *harvestEvent*: Fisken skördas periodiskt. Vid varje skörd, en viss mängd fisk ska produceras (LANDING). Fiskar väljs slumpmässigt. Fisken kan vara för liten för att fångas (MIN_CATCHABLE_FISH). Bara fisk som är större än (MIN_KEPT_FISH) kan bidra till produktionen.

Filen `config.h` har flera parametrar som styr simuleringen. Dessa är listade i Tabell 1.

Del A: Simulering och skörd

Gå genom koden för simuleringen. Utgår från de standard parametrar från Tabell 2. Svaren ska dokumenteras i filen `answers.txt`.

- Identifiera de metoder där variablerna `eventQueue` och `allTheFish` används. Vilken funktion har de?
- Experimentera med olika värden för parametrarna `LANDING` och `MIN_CATCHABLE_FISH`. Fiskbeståndet kollapsar när antalet fisk blir 0. Identifiera ett par (`LANDING`, `MIN_CATCHABLE_FISH`) där fiskbeståndet inte kollapsar, där `LANDING` är maximal (dvs., fiskbeståndet kollapsar om man ökar `LANDING` med

MAX_AGE STARVE_THRESHOLD	En fisks livslängd beror på matens tillgänglighet I <code>Event.cpp</code> används kvoten $\alpha = (\text{juvenilePopulation} + \text{adultPopulation}) / \text{STARVE_THRESHOLD}$ och representerar proportionen av maten som redan används av nuvarande fisk. Det förväntade genomsnittet för livslängden är då $(1 - \alpha) \times \text{MAX_AGE}$.
AVG_HATCH_TM	Genomsnittlig antal dagar det tar ett ägg att kläckas.
AVG_MATURATION_TM	Genomsnittlig antal dagar det tar en juvenil fisk att bli vuxen.
MAX_AVG_EGGS_NUM MIN_SPAWNING_CONC SPAWN_THRESHOLD	Antalet ägg som produceras av varje fisk beror på om det finns tillräckligt med vuxna fisk. Värdet β representerar proportionen av antalet vuxna individer i jämförelse med en tröskel <code>SPAWN_THRESHOLD</code> . Den definieras som kvoten $(\text{adultPopulation} / \text{SPAWN_THRESHOLD})$. Det genomsnittliga antalet ägg är 0 om β är mindre än <code>MIN_SPAWNING_CONC</code> , och $(\min(1, \beta) \times \text{MAX_AVG_EGGS_NUM})$ annars.
HARVEST_START HARVEST_PERIOD MIN_KEPT_AGE MIN_CATCHABLE_AGE LANDING	Fiskskörd börjar <code>HARVEST_START</code> dagar efter simuleringens start. Den upprepas varje <code>HARVEST_PERIOD</code> dagar. Fisken väljes slumpmässigt. En vald fisk med mer än <code>MIN_CATCHABLE_AGE</code> kan fångas, annars är den för liten för att fångas av fisknätet. Fångad fisk som är äldre än <code>MIN_KEPT_AGE</code> bidrar till produktionen (här <code>LANDING</code>). Produktionen (dvs. <code>LANDING</code>) är uttryckt som summan av antalet dagar av alla valda fiskar med minst <code>MIN_CATCHABLE_AGE</code> och <code>MIN_KEPT_AGE</code> dagar.
SIMULATION_HORIZON	Inga händelser exekveras efter <code>SIMULATION_HORIZON</code> dagar.
PRINT_PERIOD	statistik skrivs ut varje <code>PRINT_PERIOD</code> dagar.

Tabell 1: Simulerings parametrar.

```

MAX_AGE= 2000;           MAX_AVG_EGGS_NUM= 100;           MIN_KEPT_AGE= 600;
STARVE_THRESHOLD= 20000; MIN_SPAWN_CONC= 0.05;           LANDING= 70000;
HATCH_TM= 40;            HARVEST_START= 1000;           SIMULATION_HORIZON= 8000;
AVG_MATURATION_TM= 500; MIN_CATCHABLE_AGE= 550;        PRINT_PERIOD= 20;

```

Tabell 2: Standard parametrar

10.000) eller där `MIN_CATCHABLE_AGE` är minimal (dvs., fiskbeståndet kollapsar om man sänker den med 50 dagar). På grund av randomisering kan du behöva köra simuleringen ett par gånger (2-3 gånger). Påverkas hållbarheten av fiskbeståndet av `LANDING` eller av `MIN_CATCHABLE_AGE`? Ge en hypotes som är konsistent med dina experiment som förklarar relationen mellan produktionen (`LANDING`) och storleken på näten (`MIN_CATCHABLE_AGE`).

- Identifiera var `Fish` och `Event` objekten allokeras dynamiskt (med `new`) och när motsvarande minnet befrias. Minska `SIMULATION_HORIZON` till 1200 dagar, `LANDING` till 12000 och `STARVE_THRESHOLD` till 2000. Använd Valgrind för att kolla upp om det finns minnesläckor. Upptäcker Valgrind några minnesläckor? Kan du skapa minnesläckor igenom att kommentera en rad i koden och få Valgrind att upptäcka dem? I så fall vilken rad i vilken fil? Argumentera annars. Sätt tillbaka standard parametrar från Tabell 2.

Del B: Array-listen MyVector

Vi börjar med att implementera array-listen `MyVector`. Du hittar deklarationerna i kodskelettet `MyVector.h`. Notera att det handlar om en klass-template och att definitionerna (dvs., metodernas kroppar) ska inkluderas i samma fil som deklarationerna. Internt använder datastrukturen en array `T* storage` av `T element`, där `T` är template parametern. Arrayens längd kan sparas i instansvariabeln `unsigned capacity`. Datastrukturen använder en instans variabel till, `unsigned numberOfElements`, för att representera antalet element den innehåller. Observera att `numberOfElements ≤ capacity`. Skriv de följande publika medlemmar (du kan behöva ytterligare privata medlemmar).

I filen `MyVector.h` finns en påbörjad version av headerfilen som deklarerar ovanstående medlemmar. Du kommer att behöva modifiera denna fil för att slutföra arbetet. I synnerhet behöver du göra följande:

- **Lägg till kommentarer** i `MyVector.h`. Definitionerna måste inkluderas i `.h`-filen eftersom det handlar om en template-klass.
- **Deklarera nödvändiga privata medlemmar** i `MyVector.h`, så som privata instansvariabler. Din inre datastruktur *måste* vara en array/ett fält av `Ts`, använd inte någon annan datastruktur.

<code>MyVector()</code>	Konstruktorn initialiserar en ny tom lista med kapacitet 1.
<code>~MyVector()</code>	Destruktorn frigör allt dynamiskt allokerat minne som tillhör din lista.
<code>MyVector(other)</code>	Konstruktorn resulterar i en kopia av <code>other</code> . Kopian ska ha sin egen array med samma innehåll som <code>others</code> array. $\mathcal{O}(N)$.
<code>operator=(other)</code>	Tilldelar en kopia av <code>other</code> . Glöm inte att: hantera fallet där en <code>MyVector</code> tilldelas sigsjälv, frigöra gamla arrayen, skapa en egen array som är en kopia av den andra. $\mathcal{O}(N)$.
<code>push_back(element)</code>	Lägga till elementet till slutet av listan. $\mathcal{O}(1)$ (amorterade komplexitet).
<code>pop_back()</code>	Ta bort det sista elementet från listan. $\mathcal{O}(1)$.
<code>operator[](i)</code>	Returnera en (const) referens till elementet som ligger i det 0-baserade indexet i listan. Anta att <code>i</code> är giltig $\mathcal{O}(1)$.
<code>empty()</code>	Returnera om listan är tom. $\mathcal{O}(1)$.
<code>begin()</code>	Returnera en pekare till början av listan. $\mathcal{O}(1)$.
<code>end()</code>	Returnera en pekare till ett element som ligger "efter sista elementet i listan". Detta involverar bara pekare aritmetik $\mathcal{O}(1)$.
<code>clear()</code>	Frigör allt dynamiskt allokerat minne som används av din lista och skapa en ny tom lista med kapacitet 1.
<code>size()</code>	I denna metod ska du returnera antalet element som finns i listan. $\mathcal{O}(1)$.

Tabell 3: Publika medlemmar för `MyVector`.

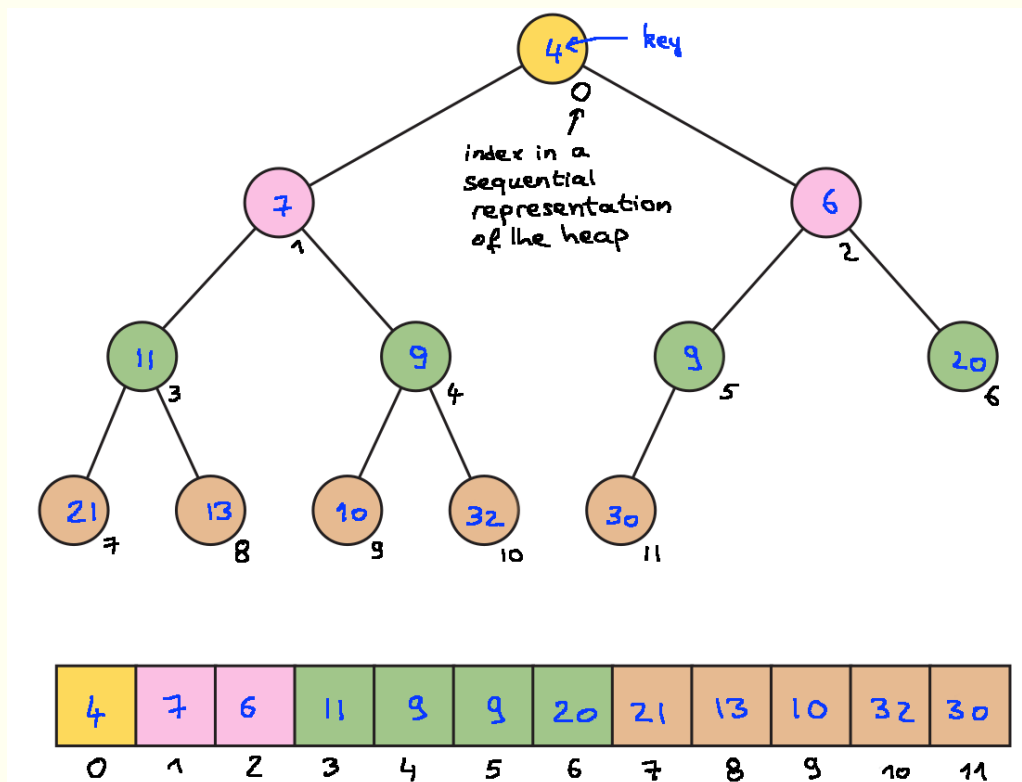
- **Implementera kropparna** för alla medlemsfunktioner och konstruktorer i `MyVector.h`. Flera operationer är likartade. Undvik redundans i din kod genom att skapa ytterligare hjälpfunktioner i din `MyVector.h`. (Deklarera dem `private` så att inte utomstående kod kan anropa dem.)
- **Testa `MyVector`:** Ersätt filen `main.c` med `test-harness-myvector.cpp` i konfigurationsfilen `fisher.pro`. Se till att göra en `clean all` och en `rebuild all` (under `Build` i Qt Creator) för att säkerställa att ändringar i konfigurationsfilen tas hänsyn till. Din implementation av `MyVector` ska klara alla tester (se till att ersätta 0 med 1 i början av `test-harness-myvector.cpp` för att kunna köra testerna). När du har klarat testerna, använd valgrind för att säkerställa att din kod resultera inte i memory-leakage.
- **Använd `MyVector` istället för `vector`** för variabeln `allTheFish` i `Simulation.h`. Minska värdet på `SIMULATION_HORIZON` till 1200 dagar, `LANDING` till 12000 och `STARVE_THRESHOLD` till 2000. Använd valgrind och se till att det inte finns några minnesläckor. Sätt tillbaka standard parametrarna från Tabell 2.

Del C: `MyPriorityQueue`

I del C i den här labben kommer du att implementera datastrukturen `MyPriorityQueue`. Den ska användas istället för `std::priority_queue` i simuleringen. En prioritetsskö kan effektivt representeras i sekvensiellt minne med hjälp av ett komplett träd där varje nod har högre prioritet än dess barn. Du ska implementera en template-klass `MyPriorityQueue<T, C>` som motsvarar en prioritetsskö där elementen är av typ `T` och jämförelsen mellan elementens prioritet görs med ett objekt av typ `C`. Konkret, för vår simulation, `T` kommer att instansieras till `Event*` medan `C` kommer att instansieras till `EventComparator`. Ett `EventComparator` ger högsta prioritet till det `Event` med lägst `eventTime` eftersom simuleringen går genom dem i den ordningen. I filen `MyPriorityQueue.h` finns det en påbörjad version av headerfilen. Lagg märke till att denna fil implementerar en template-klass och att definitionerna måste inkluderas i `.h` filen. Internt ska du använda dig av en `vector` av typ `MyVector<T>` för att sekvensiellt lagra din heap, samt en komparator av typ `C` för att jämföra elementen av typ `T` som lagras i din `vector`. Skriv de följande publika medlemmar.

<code>MyPriorityQueue()</code>	Använd standard konstruktör.
<code>~MyPriorityQueue()</code>	Använd standard destruktör.
<code>push(element)</code>	I denna metod ska du lägga till det givna elementet till prioritetsskön. $\mathcal{O}(\log(n))$.
<code>pop()</code>	I denna metod ska du ta bort det elementet med högsta prioritet. $\mathcal{O}(\log(n))$.
<code>top()</code>	I denna metod ska du returnerar elementet med högsta prioritet. $\mathcal{O}(1)$.
<code>empty()</code>	I denna metod ska du returnera om prioritetsskön är tom. $\mathcal{O}(1)$.

I filen `MyPriorityQueue.h` finns en påbörjad version av headerfilen som deklarerar ovanstående medlemmar. Du kommer att behöva modifiera denna fil för att slutföra arbetet. I synnerhet behöver du göra följande:



Figur 3: En Min-Heap: ett sekvensiellt representation av ett komplett träd.

- **Lägg till kommentarer** i `MyPriorityQueue.h`. Definitionerna måste inkluderas i `.h`-filen eftersom detta är en template-klass.
- **Deklarera nödvändiga privata medlemmar** i `MyPriorityQueue.h`, så som privata instansvariabler eller privata medlemsfunktioner vilka behövs för att implementera beteendet. Din inre datastruktur *måste* vara en `MyVector<T>`, använd inte någon annan datastruktur.
- **Implementera kropparna** för alla medlemsfunktioner och konstruktorer i `MyPriorityQueue.h`.
- **Testa `MyPriorityQueue`**: Ersätt `main.cöch test-harness-myvector.cppmed test-harness-myprio.cpp` i konfigurationsfilen `fisher.pro`. Se till att göra en `clean all` och en `rebuild all` för att säkerställa att ändringar i konfigurationsfilen tas hänsyn till. Din implementation av `MyPriorityQueue` ska klara alla tester (se till att ersätta 0 med 1 i början av `test-harness-myprio.cpp` för att kunna köra testerna). Du behöver inte använda valgrind eftersom `MyPriorityQueue` allokerar inte explicit i `heapen` och bygger istället på `MyVector`.
- **Jämför effektivitet** Använd de standard parametrar från Tabell 2. Jämför skillnad i hur långt tid det tar att köra simuleringen med en `MyVector` där `push_back` har $\mathcal{O}(1)$ amorterade tids komplexitet, och en där `push_back` har $\mathcal{O}(N)$ amorterade tids komplexitet. Se till att använda `MyVector` både i `MyPriorityQueue.h` och i `Simulation::harvestTonnageNow`. Se också till att göra en `clean all` och en `rebuild all` för att säkerställa att ändringar i header filer tas hänsyn till. Detta är till exempel relevant när du byter mellan en $\mathcal{O}(1)$ och $\mathcal{O}(N)$ amorterade komplexitet för `push_back` i `MyVector`. Rapportera om resultaten i `answers.txt`.

Referenser

- [1] P. Rekacewicz, E. Bournay, and UNEP/GRID-Arendal. Collapse of atlantic cod stocks off the east coast of newfoundland in 1992. Millenium Ecosystem Assessment. 2007. <https://www.grida.no/resources/6067>. [Online; accessed 03-Novemeber-2021].