TDDD86 – Laboration #6

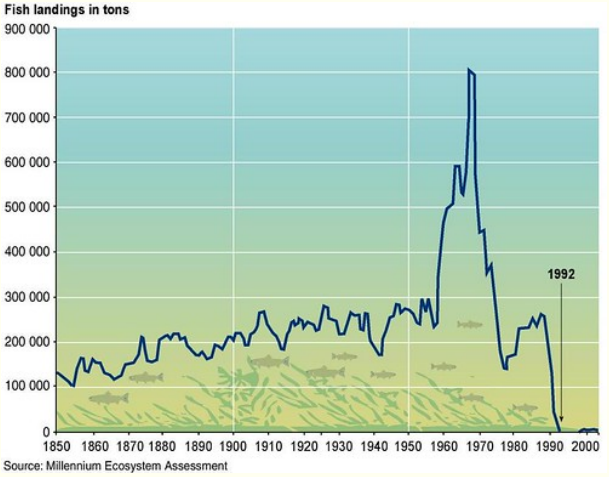
24 augusti 2022

I den här uppgiften ska du implementera två datastrukturer: en vektor och en prioritetskö. Du ska använda dem i en diskret händelse simulering (discret event simulation) för att bättre analysera och förstå ett problem kopplade till hållbar utveckling. Bäge 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 enkel och vi behöver alla verktyg som kan hjälpa oss med att bättre förstå konsekvenserna av fiskskörden. Filerna du behöver för att komma igång med denna lab 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 filarna **answers.txt**, **MyVector.h** och **MyPriorityQueue.h** är med. Informera sedan din assistent genom att maila honom/henne.

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 förstå 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.

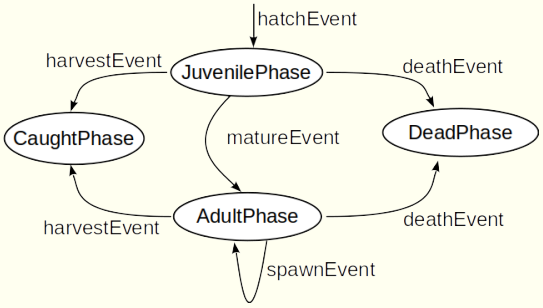


Figur 1: Kollapsen av torskbeståndet utanför den Kanadensiska östkusten 1992 [1]

Metoden `run()` i klassen `Simulation` i den tilldelade koden ligger i hjärtat av denna diskret händelse simulering. Det finns olika sorts simuleringar. Visa kan modelera kontinuerlig tid, ibland med hjälp av partiella differentialekvationer. Den här simulering är istället styrd av "händelser" eller "events". Dessa händelser läggs till, eller pushas in i, en prioritetskö där tiden tills att de ska "processas" definierar deras relativa prioriteter. När det är tid för en händelse att processas, plockas den från prioritetskön 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 en specifik säsong (se Figur 2). Händelserna läggs i simuleringens prioritetskö 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 ändra 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 prioritetskön.
- *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.



Figur 2: Möjliga fase för ett fisk

- *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 prioritetskön. En förväntad tid till nästa gång ägg ska läggas beräknas. En deathEvent eller en spawningEvent läggs till prioritetskön 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 periodisk. 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 större än (`MIN_KEPT_FISH`) kan bidra till produktionen.

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

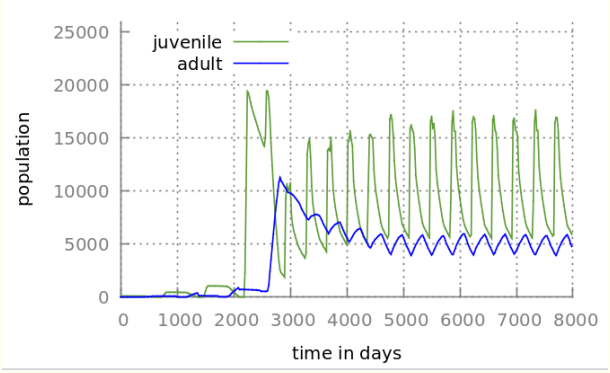
`MAX_AGE` `STARVE_THRESHOLD`	En fisks livslängd beror på tillgänglihet av mat. Värdet α defineras som kvoten `(juvenilePopulation+adultPopulation)/STARVE_THRESHOLD` och representerar proportionen av maten som redan används av nuvarande fisk. Det förväntade genomsnittet för livslängden är då $(1-\alpha) \times$ `MAX_AGE`.
`AVG_HATCH_TM`	Genomsnittlig antal dagar det tar ett ägg att kläckas.
`AVG_MATURATIOM_TM`	Genomsnittlig antal dagar det tar en juvenil fisk att bli vuxen.

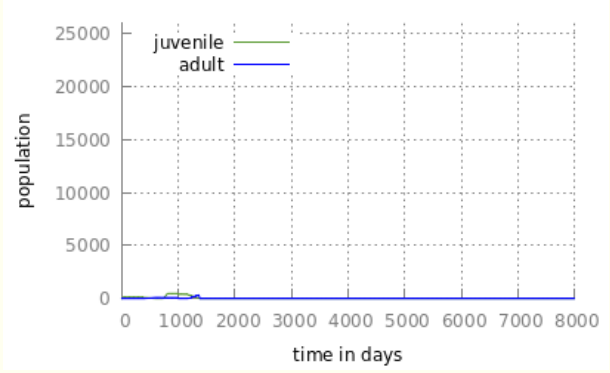
`MAX_AVG_EGGS_NUM`
`MIN_SPAWNING_CONC`
`SPAWN_THRESHOLD`
	Antalet ägg som produceras av varje fisk beror på om det finns tillräkligt med vuxna fisk. Värdet β representerar proportionen av antalet vuxna individer i jämförelse med en tröskel `SPAWN_THRESHOLD`. Den defineras som kvoten `(adultPopulation/SPAWN_THRESHOLD)`. Det genomsnittliga antalet ägg är 0 om β är mindre än `MIN_SPAWNING_CONC`, och $(min(1,\beta) \times$ `MAX_AVG_EGGS_NUM`$)$ annars.

`HARVEST_START`
`HARVEST_PERIOD`
`MIN_KEPT_AGE`
`MIN_CATCHABLE_AGE`
`LANDING`
	Fiskskörd börjar `HARVEST_START` dagar efter simulerings start. Den upprepas varje `HARVEST_PERIOD` dagar. Fisken väljes slumpmässigt. En vald fisk med mer än `MIN_CATCHABLE_AGE` kan fångas, annars är den för liten för att fångas av fisknätet. Fångad fisk som är äldre än `MIN_KEPT_AGE` bidrar till produktionen (här `LANDING`). Produktionen (dvs. `LANDING`) är uttryckt som summan av antalet dagar av alla valda fiskar med minst `MIN_CATCHABLE_AGE` och `MIN_KEPT_AGE` dagar.

`SIMULATION_HORIZON`	Inga händelser exekveras efter `SIMULATION_HORIZON` dagar.
`PRINT_PERIOD`	statistik skrivs ut varje `PRINT_PERIOD` dagar.

Tabell 1: Simulations parametrar.





```
MAX_AGE= 2000;                    HARVEST_START= 1000;
STARVE_THRESHOLD= 20000;          HARVEST_PERIOD= 366;
HATCH_TM= 40;                     MIN_CATCHABLE_AGE= 550;
AVG_MATURATION_TM= 500;           MIN_KEPT_AGE= 600;
                                  LANDING= 70000;
MAX_AVG_EGGS_NUM= 100;
MIN_SPAWN_CONC= 0.05;             SIMULATION_HORIZON= 8000;
SPAWN_THRESHOLD= 1000             PRINT_PERIOD= 20;
```

Tabell 2: Standard parametrar

Del A: Simulering och skörd

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

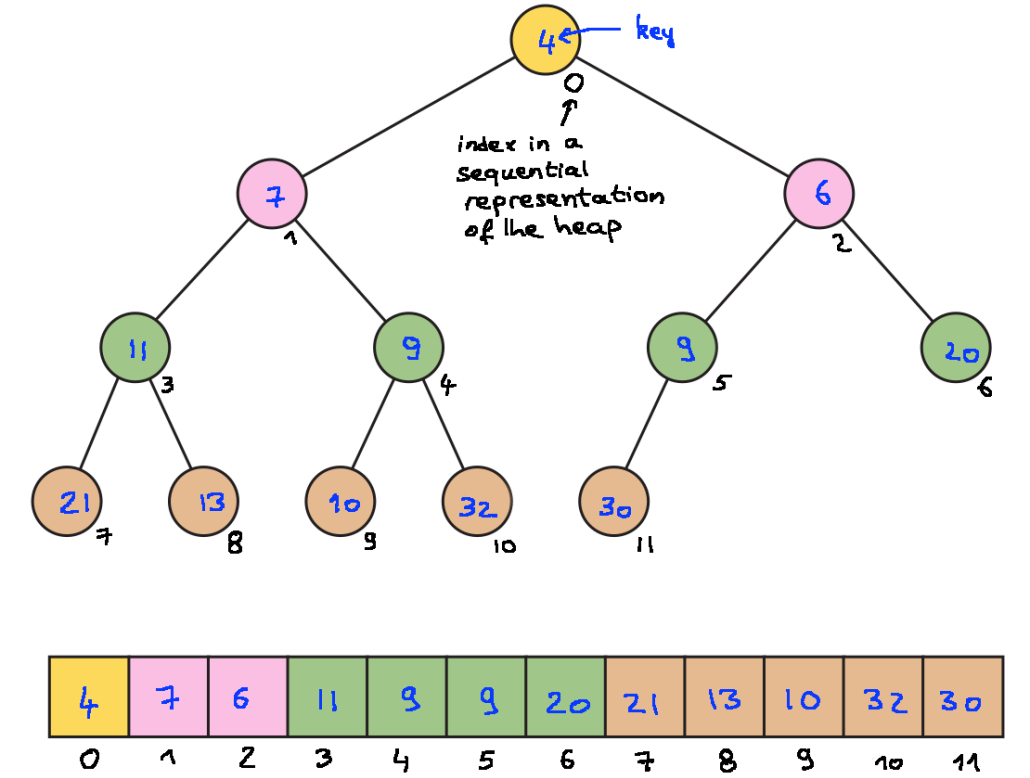
- Identifiera de metoder där variablarna `eventQueue` och `allTheFish` används. Vilken funktion har de?
- Experimentera med olika värden för parametrarna `LANDING` och `MIN_CATCHABLE_AGE`. Fiskbeståndet kollapsar när antalet fisk blir 0. Identifiera **tre** olika par `(LANDING, MIN_CATCHABLE_AGE)` där fiskbeståndet inte kollapsar, där `LANDING` är maximal (dvs., fiskbeståndet kollapsar om man ökar `LANDING` med 10.000) och 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) för varje par. Påverkas hållbarheten av fiskbståndet av `LANDING` eller av `MIN_CATCHABLE_AGE`? Ge en hypotes som håller med dina experiment.
- 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 och kollar om det finns minnesläckor. Reflektera och rapportera eventuella minnesläckor. Sätt tillbacka standard parametrar från Figur 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` $\leq$ `capacity`. Skriv de följande publika medlemmar (du kan behöva ytterligare privata medlemmar).

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

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:



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

- **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 `T`s, använd inte någon annan datastruktur.
- **Implementera kropparna** för alla medlemsfunktioner och konstruktorer i `MyVector.h`. Flera operationer är likartade. Undvik redundans i din kod genom att skapa ytterligare hjäpfunktioner i din `MyVector.h`. (Deklarera dem `private` så att inte utomstående kod kan anropa dem.)
- **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 tillbacka standard parameterer från Figur 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 prioritetskö 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 prioritetskö 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 påbörjat version av headerfilen. Lägg märke till att denna fil implementerar en template-klass och att definitionerna måste inkluderas i `.h` filen. Internt ska du använder 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.

`MyPriorityQueue()`	Använd standard konstruktor.
`~MyPriorityQueue()`	Använd standard destruktor.
`push(element)`	I denna metod ska du lägga till det givna elementet till prioritetskön. $\mathcal{O}(log(n))$.
`pop()`	I denna metod ska du ta bort det elementet med högsta prioritet. $\mathcal{O}(log(n))$.
`top()`	I denna metod ska du returnerar elementet med högsta prioritet. $\mathcal{O}(1)$.
`empty()`	I denna metod ska du returnera om prioritetskön 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:

- **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 `MyPriorityQeueue.h`.
- **Jämför effektivitet** Använd de standard parametrar från Figur 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` (under `Build` i Qt Creator) 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].