Laboration #2: Prioritetskö

Deadline: 2015-05-12

I den här laborationen skall ni implementera en abstrakt datatyp som är en variant av kö, vilken beskrivs i kapitel 10 i kursboken. Den abstrakta datatypen kö hanterar elementen i ordningen först in först ut (FIFO). Det första elementet som placeras i kön är alltid det första elementet att plockas bort ur kön, följt av det andra osv.

I en del fall är FIFO-strategin alltför simpel för den aktivitet som modelleras. En akutmottagning på ett sjukhus måste till exempel schemalägga patienter efter prioritet. En patient som anländer till akutmottagningen med ett mer allvarligt problem kommer att gå före andra, lägre prioriterade patienter, även om de andra väntat längre. Den här typen av kö kallas för *prioritetskö*. Elementen läggs till i prioritetskön efter prioritet och då element skall plockas bort ur kön kommer det element med högst prioritet att plockas bort först. En prioritetskö är användbar i många olika situationer. Faktum är att man även kan använda en prioritetskö för att implementera en sorteringsalgoritm – placera samtliga element i prioritetskön och plocka bort dem ur kön ett efter ett för att få ut dem i ordning.

Det huvudsakliga syftet med den här laborationen är att implementera en prioritetskö på ett flertal olika sätt. Ni kommer att få möjlighet att experimentera med fält, länkade listor och en speciell typ av träd som kallas för hög (*heap*). Då ni är färdiga med era implementeringar skall ni köra ett antal tester och reflektera över de olika implementeringarnas styrkor och svagheter. Vi tillhandahåller en klient som testar och mäter prestandan på respektive implementering.

Gränssnittet

Prioritetskön kommer att lagra en mängd heltal där heltalen utgör prioriteten. Större heltal anses ha högre prioritet än mindre och kommer således att plockas bort först. Följande funktioner utgör gränssnittet till prioritetskön:

```
pqueueADT NewPQueue(void);
void FreePQueue(pqueueADT pqueue);
bool IsEmpty(pqueueADT pqueue);
bool IsFull(pqueueADT pqueue);
void Enqueue(pqueueADT pqueue, int newValue);
int DequeueMax(pqueueADT pqueue);
```

`Enqueue` används för att lägga till nya element i prioritetskön. **`DequeueMax`** returnerar värdet av elementet i prioritetskön med högst prioritet vilket sedan tas bort från prioritetskön. För en detaljerad beskrivning av funktionernas beteende och användning se filen **`pqueue.h`**.

Att implementera prioritetskön

Det finns en mängd olika datastrukturer ni skulle kunna använda för att representera en prioritetskö. Vi kommer i den här laborationen att utforska och jämföra ett flertal olika datastrukturer. Den första implementeringen lagrar prioritetköns element i ett osorterat fält. Den andra implementeringen representerar prioritetskön med hjälp av en sorterad

länkad lista. Den tredje implementeringen är en hybridstrategi som kombinerar fält och länkad lista, en så kallad *chunklist*. Den fjärde implementeringen representerar prioritetskön med en hög (eng. *heap*) (denna struktur beskrivs senare i laborationsspecifikationen). De två första implementeringarna tillhandahålls (osorterat fält och länkad lista), de två sista är er uppgift att implementera.

Prioritetskö representerad med osorterat fält

Implementeringen av prioritetskön med hjälp av ett osorterat fält är redan färdigställd och finns bland de filer som delas ut med den här laborationen. Ni bör gå igenom koden och försäkra er om att ni förstår den. Implementeringen använder ett osorterat, statiskt allokerat, fält som begränsar antalet element som kan lagras i prioritetskön. Strategin för insättning är helt enkelt att insättning görs sist i fältet. Då ett element, det med högst prioritet, skall plockas ut ur prioritetskön görs en linjärsökning för att finna det. Ni kommer att genomföra ett antal körningar på den ovan beskrivna implementering för att bedöma dess styrkor och svagheter samt för att bekanta er med de funktioner för testning och tidtagning som tillhandahålls.

Prioritetskö implementerad med länkad lista

Implementeringen av prioritetskön med hjälp av en länkad lista är även den färdigställd och delas ut tillsammans med övrig kod till laborationen. Den länkade listan är en enkellänkad lista i vilken noderna endast består av ett heltalsvärde och en pekare. Värdena lagras sorterade i fallande ordning för att förenkla borttagning av elementet med högst prioritet. Insättning av element i den här implementeringen är dock något svårare då det krävs en sökning för att finna korrekt position i listan för insättning av ett nytt värde. Ovan beskrivna implementering har ingen begränsning på antalet element som kan lagras i prioritetskön. Ni kommer även för denna implementering att genomföra ett antal körningar.

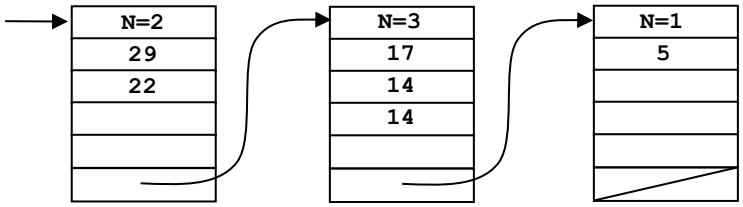
Prioritetskö implementerad som sorterad chunklist

Varken det osorterade fältet eller den länkade listan är särskilt effektiv i avseendena minnesförbrukning och tidskomplexitet. Det är nu er tur att konstruera en implementering som är en kombination av det osorterade fältet och den länkade listan för att se vilka fördelar en hybridstrategi har att erbjuda. En egenskap från den ursprungliga implementeringen av den länkade listan kommer att behållas, nämligen att representationen är sorterad. Vi skall dock försöka reducera den länkade listans väl tilltagna minnesförbrukning och de långsamma traverseringarna genom att låta varje nod i listan utgöras av ett block av element. En chunklist kombinerar alltså fältrepresentationen och den länkade listan så att strukturen utgörs av en länkad lista med block där varje block innehåller ett fält som kan innehålla ett antal element istället för enbart ett.

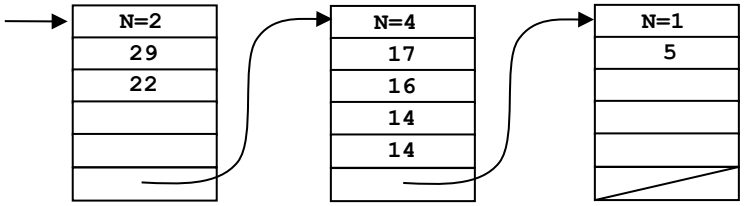
Genom att lagra flera element i varje block blir det möjligt att reducera minnesförbrukningen jämfört med prioritetskö implementerad som länkad lista då pekarna i listan kommer att utgöra en mindre del, totalt sett, av minnesförbrukningen. Då blocken är av konstant storlek kommer insättning av ett element i ett block aldrig kräva

att fler än *k* element byter plats, där *k* är maximalt antal element per block. Tiden det tar att finna korrekt position vid insättning av ett nytt element reduceras även den, då det blir möjligt att "hoppa över" hela block av element, istället för att jämföra elementen ett efter ett. Elementen lagras fortfarande i fallande ordning för att förenkla operationen att plocka bort element ur kön.

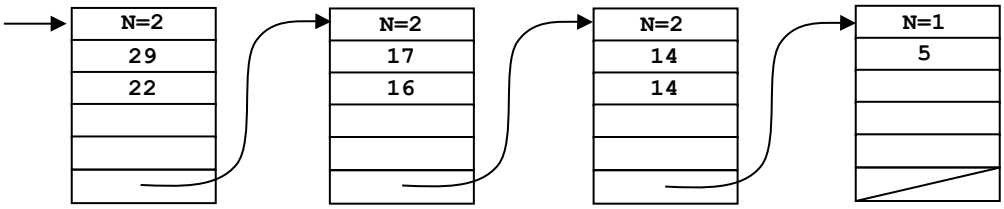
För att få bättre förståelse för hur representationen med hjälp av en chunklist fungerar, betrakta figuren nedan som illustrerar en prioritetskö med en blockstorlek på 4. Då varje block inte behöver vara fullt finns det många möjliga representationer av en prioritetskö innehållande samma element. Prioritetskön i figuren, innehållande elementen 29, 22, 17, 14, 14 och 5 kan delas in i tre block som följer (det första fältet i varje block är antalet element som används i aktuellt block).



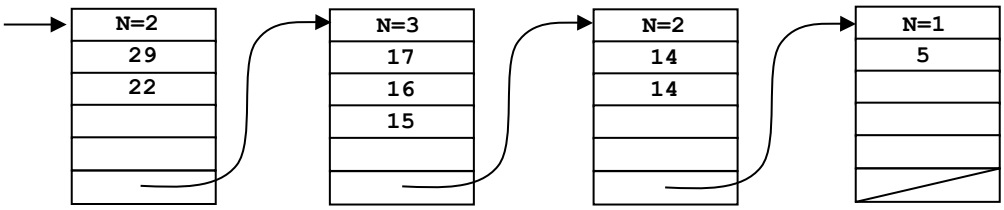
Antag att du vill sätta in värdena **16** och **15**. Insättning av **16** är relativt enkelt då blocket i vilken insättningen skall ske endast innehåller tre element och plats finns för ytterligare ett. Vid insättning av värdet **16** flyttar vi fram värdet **14** mot slutet av blocket och sätter in värdet **16** i den tomma positionen som skapats mellan värdena **17** och **14**. Vi behöver dock inte göra några förändringar på den pekare som länkar samman blocken i listan. Konfigurationen efter insättning av **16** ser därför ut som följer:



Om vi nu försöker sätta in värdet **15** blir det något mer komplicerat. Det block i vilket insättning skall ske är fullt. För att skapa plats för värdet **15** måste det block i vilket värdet skall placeras delas. En enkel strategi består i att dela blocket i två delar och placera hälften av det ursprungliga blockets innehåll i varje block. Efter uppdelningen (men före insättning av **15**) ser prioritetskön ut så här:



Efter uppdelning av blocket återstår endast insättning av värdet `15` i korrekt block.



Er uppgift är att implementera prioritetskön som en länkad lista bestående av block och den strategi som beskrivs ovan för insättning av element.

- Ni måste implementera en sorterad chunklist men besluten rörande datastrukturens exakta detaljer är upp till er (om en dubbellänkad lista används, skall en dummy-nod användas, är listan cirkulär, krävs en markör etc.). Tänk noggrant igenom vad som krävs för en effektiv implementering av de olika operationerna. Undvik redundans och onödigt komplexitet, speciellt då det inte tillför någon fördel. Försäkra er om att ni har en god förståelse för hur er datastruktur fungerar innan ni börjar koda. Rita bilder. Fundera på hur en tom prioritetskö ser ut och hur datastrukturen förändras då ett block delas upp i två delar. Storleken på blocken i listan skall anges med en konstant **`MAX_ELEMS_PER_BLOCK`**, vilken skall gå att förändra.

- Om insättning av ett element sker i ett block som är fullt skall den ovan beskrivna strategin användas där en delning av blocket i två delar sker innan insättning av elementet.

- Att ta bort ett element ur prioritetskön innebär att ta bort det första elementet ur det första blocket. Det är upp till er att avgöra om de resterande elementen i blocket skall byta plats efter det att det första elementet tagits bort. Om ni tar bort det sista elementet ur ett block skall ert program frigöra det minnesutrymme som är associerat med blocket.

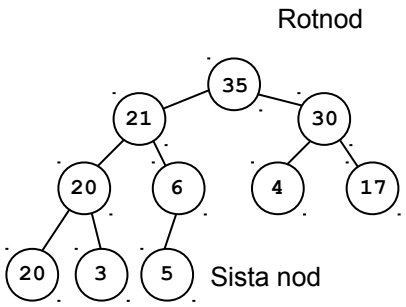
Prioritetskö implementerad som hög

Även om binära sökträd, som kommer att beskrivas på föreläsning och beskrivs i kapitel 13 i kursboken, skulle kunna utgöra en god representation av en prioritetskö så finns det en annan typ av binärt träd som utgör ett bättre alternativ i det här fallet, nämligen en *hög* (eng. *heap*). En hög är ett binärt träd som har följande två egenskaper:

- Det är ett *komplett* binärt träd dvs. ett träd där samtliga nivåer är fulla (alla noder har två barn), förutom möjligen den lägsta nivån vilken fylls från vänster till höger.

- Värdet i varje nod är större än eller lika med värdet av dess barn.

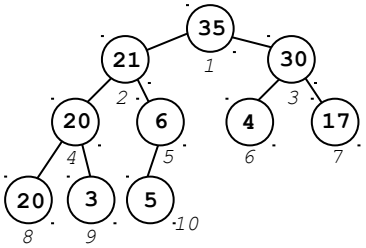
Här är en konceptuell bild av en liten hög:



Notera att en hög skiljer sig från ett binärt sökträd i två viktiga avseenden. För det första, i ett binärt sökträd är noderna sorterade, medan noderna i en hög är ordnade i en mycket svagare bemärkelse. Det sätt på vilket en hög är ordnad är dock tillräckligt för att implementera operationerna på en prioritetskö effektivt. För det andra kan ett binärt sökträd ha många olika former. En hög måste vara ett komplett binärt träd vilket innebär att varje hög som innehåller 10 noder har exakt samma form som alla andra högar med 10 noder.

Representation av hög med ett fält

Ett sätt att hantera en hög är att använda en definition av noder precis som den som används i kursboken för binära sökträd. Det är dock möjligt att utnyttja det faktum att en hög måste vara ett komplett binärt träd och skapa en enkel fältrepresentation för att på så sätt slippa arbeta med pekare. Tänk er att noderna i en hög är numrerade nivå för nivå så här:



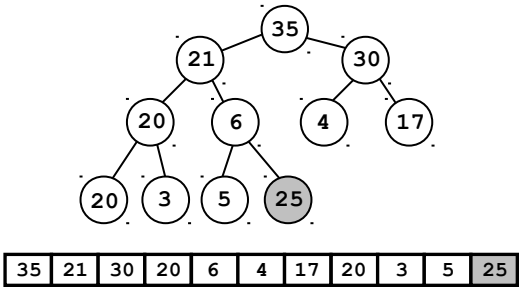
se nu på motsvarande fältrepresentationen av samma hög

35	21	30	20	6	4	17	20	3	5
1	2	3	4	5	6	7	8	9	10

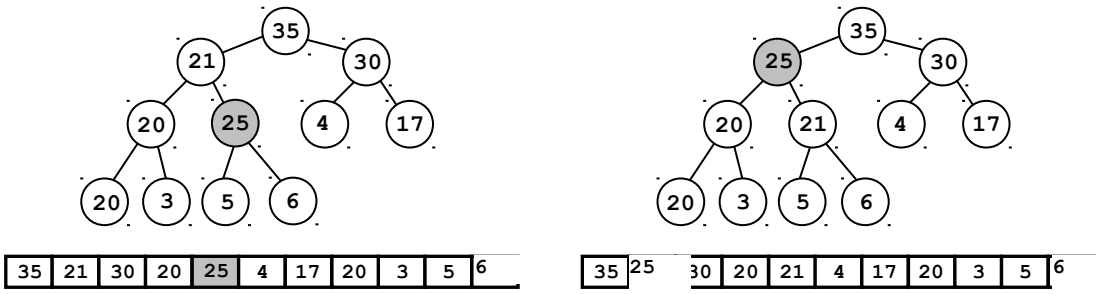
ni kan dividera godtyckligt nodnummer med 2 för att få nodnumret på dess förälder. Nod nummer 11 är t.ex. förälder till nod nummer 5. De två barnnoderna till nod *i* är 2*i* och 2*i* + 1 barnnoderna till nod nummer 3 är alltså noderna 6 och 7. Kom ihåg att även om många figurer i den här laborationsspecifikationen illustrerar en hög med hjälp av ett träd så kommer ni att representera er hög med ett fält. Notera också att indexen börjar på 0 i C.

Insättning

En insättning i en hög utförs annorlunda än dess funktionella motsvarighet i ett binärt sökträd. Den nya noden placeras längst ner i trädet så att trädet bevaras komplett (det vill säga första vakanta plats från vänster räknat). Noden skiftas sedan uppåt i trädet genom att byta plats med sin förälder ända till dess att den är efter sin förälder i sorteringsordningen. Antag t.ex. att vi vill sätta in värdet `25` i vår hög. Först lägger vi till en ny nod längst ner i högen:

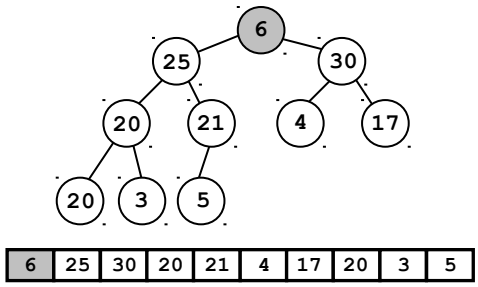


Vi jämför värdet av den insatta noden med värdet av dess förälder och om nödvändigt byter vi plats på dem. Då vår hög är representerad med ett fält är det möjligt att byta plats på noderna enbart genom att skifta plats på elementen i fältet. Då en nod bytt plats i högen jämförs nodens värde med dess nya förälders värde och noden fortsätter att förflyttas uppåt till dess att den befinner sig i korrekt position. Den här operationen kallas ibland för *bubble-up*.

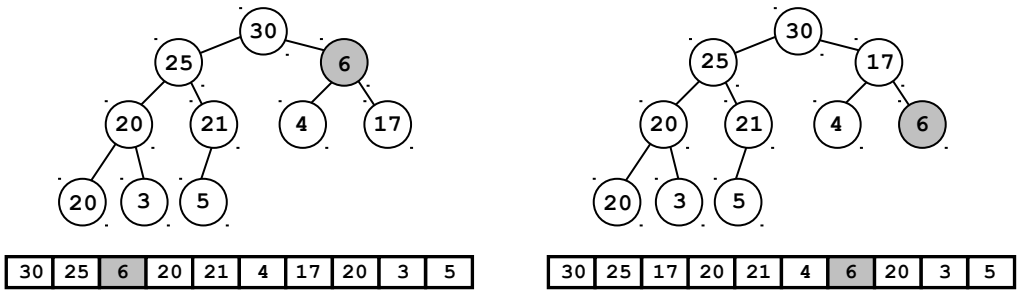


Borttagning

Givet ordningen på noderna i en hög så återfinns det största värdet alltid i roten där det är lätt att komma åt. Efter det att rotnoden tagits bort måste de resterande noderna ordnas om så att högen består. Istället för att ta bort rotnoden och ordna om resterande noder kan vi lämna rotnoden där den är och kopiera värdet från den sista noden till rotnoden och istället ta bort den sista noden.



Vi har nu åter ett komplett binärt träd vars vänstra och högra delträd är högar. Det enda problemet är att elementet i roten kan vara (och vanligtvis är) i fel position. För att återställa sorteringsordningen i högen kan vi skifta rotnoden nedåt i trädet genom att byta plats med det största (i sorteringsordningen) av sina barn tills dess att den är före sina barn i sorteringsordningen eller botten är nådd.



Ovan beskrivna operation för att flytta en felaktigt placerad rotnod genom att skifta den nedåt i högen kallas ibland för *heapify*-ing.

Er andra uppgift i den här laborationen är att implementera prioritetskön som en hög med hjälp av ett fält. Ni skall inte använda ett statiskt allokerat fält för att representera högen utan fältet skall inledningsvis vara relativt litet för att sedan växa vid behov.

Tillvägagångssätt

Vid genomförandet av laborationen föreslås följande tillvägagångssätt:

Uppgift 1 – Osorterat fält och sorterad länkad lista

Läs igenom implementeringarna av prioritetskön som delats ut till den här laborationen (fält och länkad lista). Kör tester och tidtagningar på de båda implementeringarna och skriv ner resultaten från körningarna i bifogat blad (se sidan 10). Kom ihåg att köra tidsmätningarna utan Visual Studios debugger, dvs starta med ctrl+F5 eller från kommandoraden (cmd.exe eller Windows PowerShell).

Uppgift 2 – Sorterad chunklist

Det är nu er tur att skriva er egen implementering av prioritetskön. Det kan vara lämpligt, men inte nödvändigt, att utgå från implementeringen av den enkellänkade listan då dess struktur liknar den lista ni skall implementera. Då er implementering är färdig skall ni köra samma tester och tidtagningar på den som i uppgift 1. Även här skall ni skriva ner resultaten i bifogat blad på sidan 10.

Uppgift 3 – Heap

Då ni är färdiga med implementeringen av er chunklist är det dags att implementera prioritetskön med hjälp av en hög. Som alltid, tänk innan ni kodar. Försäkra er om att ni förstår högens struktur och operationerna för insättning och borttagning av element. Det får inte finnas någon övre gräns för antalet element er hög kan lagra. I stället för att använda ett fält med konstant storlek skall ni inledningsvis allokera plats för rimligt antal element. Då det ursprungliga fältet är fullt skall ni allokera ett större fält och kopiera innehållet till det nya fältet och sedan frigöra det minne som är associerat med det ursprungliga fältet. Då implementeringen är klar, kör tester och tidtagningar och skriv ner resultaten på bifogat blad.

Uppgift 4 – Besvara frågor

Precis som laborationen illustrerar så finns det många olika datastrukturer vi kan använda för att representera en prioritetskö. Varje datastruktur vi använder för att representera prioritetskön utgör en kompromiss mellan mängden utnyttjat minne, operationernas effektivitet, komplexitet vid implementering och underhåll av kod osv.

För varje implementering (de två utdelade och de två du implementerat), använd modulen `performance.c/h` för att testa respektive implementerings prestanda. Baserat på resultaten vid de olika körningarna skriv ner resultaten för de implementeringarna i bifogat blad på sidan 10. Besvara frågorna på sidan 11.

Tips

Frigör minne. Ni är ansvariga för att frigöra det minne som allokerats dynamiskt. Er implementering får inte läcka något minne och er implementering av funktionen FreePQueue ansvarar för att allt minne frigörs. Vi rekommenderar att ni skriver hela programmet utan att frigöra något minne. Då ert program fungerar, gå igenom koden och lägg noggrant till kod för att frigöra allokerat minne.

Tänk innan ni kodar. Mängden kod som krävs för att färdigställa de två implementeringarna är inte stor men ni kommer att märka att det krävs en del tankearbete för att få den att fungera. Använd papper och penna för att rita bilder över hur er implementering beter sig.

Testa noggrant. Ni vill säkert inte överraskas av en retur på laborationen på grund av att vi funnit en mängd buggar som ni inte upptäckt på grund av otillräcklig testning. Försäkra er om att ni noggrant testat samtliga operationer!

Utdelad kod

På kursens hemsida finns den kod som delas ut till den här laborationen. Följande kod delas ut:

`main.c`	Huvudprogram
`performance.h/c`	Modul med tidtagningsfunktioner för att testa effektivitet
`pqueuetest.h/.c`	Modul med funktioner som utför enkla funktionstester på pqueue
`pqueue.h`	Gränssnitt till prioritetskön
`pqarray.c`	Implementering av **`pqueue.h`** med hjälp av ett osorterat fält
`pqlist.c`	Implementering av **`pqueue.h`** med hjälp av en enkellänkad lista

För att komma igång; skapa ett projekt och lägg till klientfilerna och önskad implementering av gränssnittet **`pqueue.h`**.

Att lämna in

Som vanligt skall ni lämna in er kod i PingPong. Ni behöver enbart lämna in de två implementeringarna ni skrivit tillsammans det ifyllda bladet från nästa sida samt svaren på frågorna på sidan 11.

Lycka till!

Summera resultaten för implementeringarna i bladet nedan. Det är möjligt att en del operationer utförs så snabbt att det inte är möjligt att mäta tidsåtgången. De långsammare operationerna skall dock gå att mäta då storleken på prioritetskön ökar. Genomför provkörningarna på tre olika storlekar: N = 10000, 20000 och 40000. Det är möjligt att ni behöver använda ännu större värde på N om ni har en snabb dator. Skriv ner de tider samt minnesförbrukning som programmet skriver ut vid körning av de olika implementeringarna av prioritetskön. Genomför även en teoretisk komplexitetsanalys av de olika operationerna som en funktion av N (antalet element i prioritetskön).

	Fält	**Länkad lista**	**Chunklist**	**Heap**
Minnesförbrukning N_1				
Minnesförbrukning $2N_1$				
Minnesförbrukning $4N_1$				
Enqueue N_1				
Enqueue $2N_1$				
Enqueue $4N_1$				
Enqueue O (Ordo)				
DequeueMax N_1				
DequeueMax $2N_1$				
DequeueMax $4N_1$				
DequeueMax O (Ordo)				
Heapsort N_1 (slumptal)				
Heapsort $2N_1$				
Heapsort $4N_1$				
Heapsort O (Ordo)				
Heapsort N_1 (sorterad)				
Heapsort $2N_1$				
Heapsort $4N_1$				
Heapsort O (Ordo)				
Heapsort N_1 (bakvänd)				
Heapsort $2N_1$				
Heapsort $4N_1$				
Heapsort O (Ordo)				

Frågor (skall besvaras och lämnas in med laborationen)

Ta er tid att besvara nedanstående frågor avseende körningarna ovan. Vi förväntar oss inte någon uppsats utan bara att ni visar oss att ni har funderat över problematiken.

1) Stämmer de vid körningarna observerade tiderna överens med din komplexitetsanalys? Tar de funktioner som har samma komplexitet lika lång tid dvs. om operation A och B båda är O(n), tar de lika lång tid? Skall de ta lika lång tid?

2) Upprepa körningarna med din implementering av prioritetskö med en chunklist. Justera konstanten **`MAX_ELEMS_PER_BLOCK`** till mindre och större värden. Vad kan ni säga om relationen mellan storleken på blocken, minnesförbrukningen och de olika operationernas effektivitet? Vad framstår som en optimal storlek på blocken om prioritetskön innehåller 2000, 10000 eller 20000 element?

3) För att effektivisera prestanda för en operation som används ofta måste man ibland offra prestanda för någon annan, förhoppningsvis mindre använd operation. Är det bättre att optimera **`Enqueue`** på bekostnad av **`DequeueMax`** eller vice versa i fallet prioritetskö?

4) Vilka är de primära styrkorna och svagheterna för respektive implementering i den här laborationen? Vilken implementering verkar mest tilltalande om en så snabb implementering som möjligt eftersöks? Om ni önskar använda så lite minnesutrymme som möjligt? Om ni behöver få koden skriven och avlusad på kortast möjliga tid?