Tidskomplexitet och elementära datastrukturer

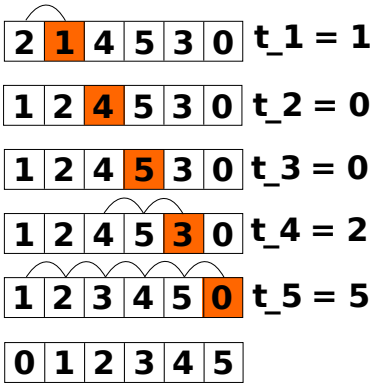
Denna lektion kommer vi prata lite kort om *tidskomplexitet* - hur man mäter tidsåtgången för en algoritm och hur det hänger ihop med tävlingsprogrammering. Sedan kommer vi studera ett antal grundläggande datastrukturer lite närmre.

Tidskomplexitet

Vi kommer börja vår studie av tidskomplexitet med att illustrera en algoritm som används för att sortera tal, *insertion sort*.

Antag att vi har en lista av tal $a_0, a_1, ..., a_{n-1}$ och vill ha denna lista i sorterad ordning. Ett sätt att göra detta vore att först sortera a_0 (som ju redan är sorterad), sedan sortera a_1 genom att placera in den antingen till vänster eller höger om a_0, sedan sortera a_2 genom att placera in den på rätt plats med a_0 och a_1, och så vidare.

Vi går alltså igenom varje tal i listan, ett efter ett, och placerar in det på rätt ställe i den sorterade listan.



En intressant fråga är hur lång tid det faktiskt tar att köra insertion sort. Inom datavetenskapen är man inte så ofta intresserad av den faktiskt klocktiden som algoritmen tar, utan vi försöker istället uppskatta hur körtiden växer som en funktion av indatans storlek. För en sorteringsalgoritm är indatans storlek antalet element vi sorterar, n, så vi är intresserade av hur tiden, $T(n)$ växer i förhållande till n.

För att analysera det måste vi först specifiera algoritmen i pseudokod, så att vi vet exakt vilka steg vi utför.

Algoritm 1 Insertion sort

```
1: procedure InsertionSort(A)                ▷ Sorterar listan A som har N element
2:     for i ← 1 till N − 1 do
3:         j ← i
4:         så länge j > 0 och A[j] < A[j − 1]
5:             Byt plats på A[j] och A[j − 1]
6:             j ← j − 1
```

För att analysera hur lång tid algoritmen tar börjar vi med att anta att alla småoperationer tar lika lång tid, och att den tiden är exakt 1 (av någon obestämd enhet). Vi måste vara noga med att göra rimliga antaganden av vad en liten operation är, men de flesta har en hyfsad intuition för detta.

I vårt program utför varje rad en liten operation, men de två looparna kan göra att vissa rader exekveras fler än en gång. Den yttre for-loopen kommer köras $N - 1$ gånger. Antalet gånger den

inre loopen kör varierar dock beroende på hur listan vi sorterar ser ut. Vi antar därför att den för ett visst i kör t_i gånger.

Nu kan vi skriva ut hur lång tid varje rad tar, samt hur många gånger de körs:

Algoritm 2 Insertion sort

1:	**procedure** InsertionSort(A)	▷ Sorterar listan A som har N element
2:	**for** $i \leftarrow 1$ till $N-1$ **do**	▷ Körs $N-1$ gånger, kostnad 1
3:	$j \leftarrow i$	▷ Körs $N-1$ gånger, kostnad 1
4:	**så länge** $j > 0$ och $A[j] < A[j-1]$	▷ Körs $\sum_{i=1}^{N-1} t_j$ gånger, kostnad 1
5:	Byt plats på $A[j]$ och $A[j-1]$	▷ Körs $\sum_{i=1}^{N-1} t_j$ gånger, kostnad 1
6:	$j \leftarrow j-1$	▷ Körs $\sum_{i=1}^{N-1} t_j$ gånger, kostnad 1

Vi kan nu summera kostnaden för varje rad: $T(n) = (N-1)+(N-1)+\left(\sum_{i=1}^{N-1} t_j\right)+\left(\sum_{i=1}^{N-1} t_j\right)+\left(\sum_{i=1}^{N-1} t_j\right) = 3\left(\sum_{i=1}^{N-1} t_j\right)+2N-2$.

Oftast är det vi är intresserade av hur algoritmen beter sig i det *värsta fallet*. Det värsta fallet för insertion sort är när listan är i omvänd ordning. Då kommer while-loopen i algoritmen köras i gånger, dvs $t_i = i$.

Övning 1. Argumentera för varje det i värsta fallet gäller att $t_i = i$.

Sätter vi in detta får vi $T(n) = 3\left(\sum_{i=1}^{N-1} i\right)+2N-2 = 3\frac{(N-1)(N-2)}{2}+2N-2 = \frac{3}{2}(N^2-3N+2)+2N-2 = \frac{3}{2}N^2-\frac{5}{2}N+1$.

$T(N)$ växer alltså kvadratiskt mot antalet element N. Vi skriver detta på följande sätt: $T(N) = O(N^2)$. Detta kallas för *asymptotisk notation*, och fångar beteendet hos $T(N)$ för "stora" problem.

På samma vis, om $T(N) = 2n+15$ växer $T(N)$ ungefär linjärt mot N, så vi säger att $T(N) = O(N)$.

Formellt defineras notationen på följande vis:

Definition 1. Vi säger att en funktion $f(n) = O(g(n))$ om $\frac{f(n)}{g(n)} \leq c$ för alla $n \geq n_0$ där c och n_0 är positiva konstanter som vi väljer.

Rent intuitivt betyder detta att $f(n)$ växer långsammare eller lika snabbt som $g(n)$. Till exempel är $an^2+bn+c = O(n^2)$, men även $an+b = O(n^2)$. Med hjälp av denna stora-O-notation har vi alltså ett enkelt sätt att jämföra hur snabba olika algoritmer är.

Vi bevisar nu att med denna definition så är tiden som insertion sort tar i värsta fallet $O(N^2)$.

Exempel 1. Visa att $\frac{3}{2}N^2-\frac{5}{2}N+1 = O(N^2)$.

Bevis. När $N \geq 1$ gäller $\frac{3}{2}N^2-\frac{5}{2}N+1 \leq 2N^2+N^2+N^2 \leq 4N^2 \Rightarrow \frac{\frac{3}{2}N^2-\frac{5}{2}N+1}{N^2} \leq 4$ för $N \geq 2$. Med konstanterna $c = 4$ och $n_0 = 1$ uppfylls alltså villkoret i definitionen. □

Vi kallar tiden som en algoritm tar med avseende på problemstorleken för algoritmens *tidskomplexitet*. Motsvarande kan vi mäta hur mycket minne en algoritm tar med hjälp av asymptotisk notation, något vi kallar algoritmens *minneskomplexitet*. Insertion sort sparar alltid ungefär N heltal i minnet, så den har minneskomplexiteten $O(N)$.

För konstanter k säger vi att $k = O(1)$. För att avgöra om den algoritm man har kommit på är snabb nog kan man använda följande tabell. Den uppskattar hur stort problem en algoritm med en viss komplexitet kan lösa på någon sekund.

Komplexitet	n
$O(\log n)$	$2^{(10^7)}$
$O(\sqrt{n})$	10^{14}
$O(n)$	10^7
$O(n \log n)$	10^6
$O(n\sqrt{n})$	10^5
$O(n^2)$	$5 \cdot 10^3$
$O(n^2 \log n)$	$2 \cdot 10^3$
$O(n^3)$	300
$O(2^n)$	24
$O(n2^n)$	20
$O(n!)$	11

När man löser ett problem ska dock fokus alltid vara på att ha en korrekt algoritm. Ett program som går långsamt men ger rätt svar kan ge delpoäng, medan ett snabbt program som svarar att $1 + 1 = 3$ inte är mycket att ha.

Komplexitetsanalys kan också användas för att bestämma *undre gränser* till ett problem. För att resonera kring undre gränser inför vi Omega-notationen, som är motsvarande stora-O-notationen, fast åt andra hållet. Eftersom $n = O(n^2)$ säger Omega-notationen att $n^2 = \Omega(n)$. Dvs att n^2 växer alltså snabbare än n.

Notationen är vettig, för vi vill kunna konstatera att en viss algoritm måste ha en undre gräns på sin körtid. Påståendet "Algoritm X kräver minst $O(n^2)$ tid i värsta fallet" är helt värdelöst, eftersom $O(n^2)$ beskriver en *övre* gräns, inte en undre gräns. Istället vill vi säga att "Algoritm X kräver minst $\Omega(n^2)$ tid i värsta fallet".

Övning 2. Vad har insertion sort för undre körtid? Hitta en så bra gräns som möjligt, och bevisa att det är den bästa möjliga undre gränsen.

Övning 3. Ge en $O(n)$-algoritm och en $O(1)$-algoritm för att summera de n första heltalen.

Övning 4. Visa med definitionen för asymptotisk notation att $10n^2 + 7n - 5 + \log^2 n = O(n^2)$. Ge ett exempel på konstanter c, n_0 som uppfyller villkoret.

Övning 5. Visa att $f(n) + g(n) = O(max\{f(n), g(n)\})$.

Övning 6. Är $2^{n+1} = O(2^n)$? Är $2^{2n} = O(2^n)$?

Övning 7. Visa att $(n + a)^b = O(n^b)$ för positiva konstanter $a, b > 0$.

Datastrukturer

När vi programmerar använder vi olika *datastrukturer* för att enklare representera vår data, och för att förhoppningsvis snabbare kunna utföra de operationer vi vill göra. En datastruktur stödjer ett antal *operationer*, och varje operation har en viss tidskomplexitet.

Vektor

Först, lite repetition från lektionen om C++.

Den mest grundläggande datastrukturen är en *vector*. En vector lagrar ett visst antal element av en specifik datatyp. De två grundläggande operationerna som en vector stödjer är att läsa och skriva till en viss plats i vectorn.

För att använda en vector i C++ måste man inkludera `#include<vector>`.

Man deklarerar en vector såhär:

```
1 vector<datatyp> v(n);
```

där *datatyp* är den typ du vill lagra i vectorn, och n är antalet element.

För att läsa och skriva till en variabel använder du hakparantesoperatorn:

```
vector<int> v(10);

v[0] = 4;
v[1] = 5;

cout << (v[0] + v[1]) << endl; // Skriver ut 9
```

Notera här att en vector är *nollindexerad*. Det innebär att det första värdet i vectorn har index 0, och det sista värdet har index $n - 1$.

Du kan lägga till ett värde i slutet av en vector med funktionen *push_ back()*.

```
vector<int> v; // skapa en tom vector

v.push_back(1);
v.push_back(2);

cout << v[0] + v[1] << endl; // Skriver ut 3
```

Man kan också ta bort och lägga in element i en vector med *erase()* och *insert()*.

Då använder man:

```
vector<int> v(3);
v[0] = 10; //[10, 0, 0]
v[1] = 1337; //[10, 1337, 0]
v.erase(v.begin() + 1); // ta bort element med index 1
// [10, 0]
v.insert(v.begin() + 0, 42); // sätt in element med index 0
// [42, 10, 0]
```

Storleken på en vector kan således ändras ganska ofta. För att få ut den nuvarande storleken på en vector använder man funktionen *size*:

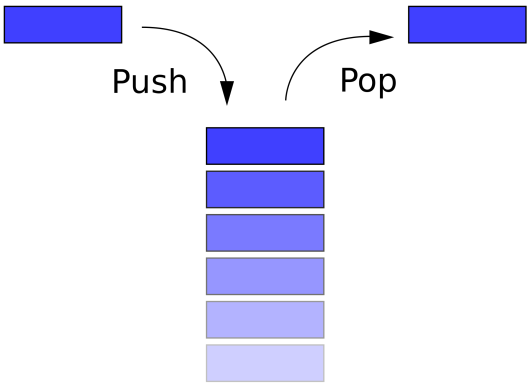
```
vector<int> v(3);
cout << v.size() << endl; // 3
v.push_back(1);
v.push_back(2);
cout << v.size() << endl; // 5
v.erase(v.begin() + 1);
cout << v.size() << endl; // 4
```

Komplexiteten för operationerna är:

- `v[i]` - $O(1)$
- `v.insert()` - $O(k)$, om det finns k element efter det du sätter in
- `v.erase()` - $O(k)$, om det finns k element efter det du tar bort
- `v.push_back()` - $O(1)$
- `v.size()` - $O(1)$

Stack

En *stack* är en datastruktur som beter sig ungefär som en stor hög med pannkakor som du håller på att laga. När du lagat en ny pannkaka lägger du den längst upp i högen, och när du tar en pannkaka tar du också den som ligger längest upp i högen.



Figur 1: En stack

.

Man skapar en stack i C++ genom att inkludera `#include<stack>` och sedan:

```
stack<datatyp> s;
```

Stacken stödjer tre operationer. Att lägga till ett element högst upp i stacken kallas för *push()*:

```
stack<int> s; //[]

s.push(1); //[1]
s.push(2); //[2, 1]
s.push(3); //[3, 2, 1]
```

Att ta bort elementet högst upp i stacken kallas för *pop()*:

```
stack<int> s; //[]

s.push(1); //[1]
s.push(2); //[2, 1]
s.pop(); //[1]
s.pop(); //[]
```

Slutligen kan man självfallet också ta reda på vilket element som är i toppen av stacken med hjälp av *top()*. Observera att du inte får göra top() på en tom stack.

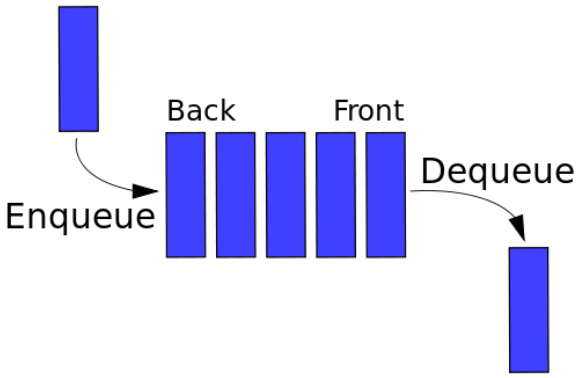
```
stack<int> s; //[]

s.push(1); //[1]
cout << s.top() << endl; // skriver ut 1
s.push(2); //[2, 1]
cout << s.top() << endl; // skriver ut 2
s.pop(); //[1]
s.pop(); //[]
cout << s.top() << endl; // fel!!!
```

En stack har också funktionerna *size()* och *empty()*, som returnerar storleken resp. huruvida stacken är tom eller inte.

Kattisövning (ID: dagy:reverse)**.** Lös problemet *Omvändning* genom att använda en stack istället för en vector.

Stackens operationer har alla komplexiteten $O(1)$.



Figur 2: Kö. Av Vegpuff/Wikipedia

Kö

Datastrukturen *queue* är motsvarigheten till en helt vanlig kö. Dess operationer är nästan precis som stackens, förutom att man använder *front()* istället för *top()* för att plocka ut elementet som är längst fram.

Operationen *push()* lägger alltså till ett element längst bak i kön, och *pop()* tar bort det element som är längst fram i kön.

För att använda en kö i C++ måste du inkludera `#include<queue>`. Den används på följande vis:

```
queue<int> q;

q.push(1); //[1]
q.push(2); //[1, 2]
cout << q.front() << endl; // skriver ut 1
q.pop(); // [2]
q.push(3); //[2, 3]
q.push(4); //[2, 3, 4]
q.pop(); //[3, 4]
cout << q.front() << endl; // skriver ut 3
cout << q.size() << endl; // skriver ut 2
```

Precis som en stack har kön komplexitet $O(1)$ för alla sina operationer.

Övning 8. Implementera en kö med två stackar, och analysera körtiden för alla operationer.

Övning 9. Implementera en stack med två köer, och analysera körtiden för alla operationer.

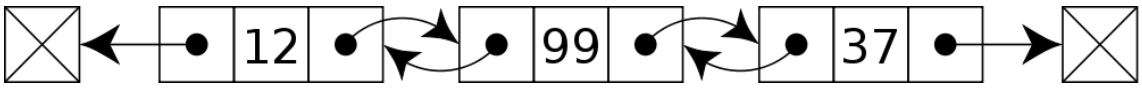
Länkade listor

En länkad lista är ungefär som en vector, i att den innehåller ett visst antal element i ordning. Skillnaden är dock att ordningen avgörs med hjälp av att varje element har två *pekare* till det föregående elementet och nästa element.

Det finns alltså ett första element som har en pekare till nästa element, som har en pekare till nästa element och så vidare.

För att använda en länkad lista i C++ använder du `#include<list>`, och skapar den med:

```
list<int> l;
```



Figur 3: Länkad lista

Du kan lägga till saker längst fram och längst bak i listan med funktionerna *push_front()* och *push_back()*.

Du kan också få ut värdena längst fram och längst bak med *front()* och *back()*:

```
list<int> l;

l.push_front(2); //[2]
cout << l.back() << endl; // skriver ut 2
l.push_front(3); //[3, 2]
cout << l.front() << endl; // skriver ut 3
l.push_back(4); //[3, 2, 4]
cout << l.back() << endl; // skriver ut 4
l.push_front(5); //[5, 3, 2, 4]
cout << l.back() << endl; // skriver ut 4
```

För att gå igenom listan använder du iteratorer. En iterator är ett objekt som pekar på ett visst värde i listan. Iteratorn som pekar på det första elementet heter *begin()*, och elementet som pekar på elementet efter det sista heter *end()*. Iteratorn har datatypen *list<datatyp>::iterator*.

Du flyttar fram en iterator i listan genom att använda *++*-operatorn på den, och flyttar tillbaka den genom att använda *–*operatorn på den.

Slutligen kan du få ut värdet som en iterator *it* pekar på med hjälp av **it*.

```
list<int> l;

l.push_back(1); // [1]
l.push_back(2); // [1,2]
l.push_back(3); // [1,2,3]
l.push_back(4); // [1,2,3,4]
l.push_back(5); // [1,2,3,4,5]

list<int>::iterator it = l.begin(); //it = l[0]

cout << *it << endl; // skriver ut 1

it++; // it = l[1]
it++; // it = l[2]
cout << *it << endl; // skriver ut 3
it--; //it = l[1]
cout << *it << endl; // skriver ut 2


// Loopar igenom alla värden i listan i ordning och skriver ut dem.
for(it = l.begin(); it != l.end(); it++){
    cout << *it << endl;
}
```

Loopen i slutet visar hur man kan använda iteratorer för att gå igenom alla värden i en lista. Man initialiserar *it* till att peka på det första värdet. Eftersom end() pekar på ett element *efter* det sista i listan vill vi bara loopa så länge *it* inte är lika med end(). Slutligen, efter varje iteration av listan

vill vi flytta fram iteratorn ett steg.

Mängder och uppslagningar

Under introduktionshelgen gick vi även igenom datastrukturen `set` (för mängder) och `map` (för uppslagningar). Hur dessa två strukturer fungerar internt kommer vi inte ta upp nu ty de är väldigt komplicerade, men det passar på att repetera strukturerna samt deras komplexiteter. De två strukturerna är principiellt ekvivalenta. Skillnaden ligger endast i att alla element i en `map` är associerade med något visst värde, medan vi i ett `set` endast sparar själva nycklarna.

Insättningar, borttagningar och uppslagningar går i samtliga strukturer i *logaritmisk* tid - $O(\log n)$, där n är antalet element i strukturen. Det kan tyckas vara sämre än en `vector`, men man får inte glömma att man kan lagra *vad som helst* i dessa strukturer, och göra uppslagningar i $O(\log n)$ istället för $O(n)$!

En annan nyttig egenskap hos dessa strukturer är att de lagrar alla element i stiganded ordning. Dvs:

```
1  set<int> s;
2  s.insert(7);
3  s.insert(3);
4  s.insert(3);
5  s.insert(1);
6
7  for(set<int>::iterator it = s.begin(); it != s.end(); ++it){
8    cout << *it << endl;
9  }
```

skriver ut `1 3 7`.

Vi kan med andra ord hämta ut det lägsta värdet i ett sätt med `*s.begin()`, samt ta bort det lägsta värdet med `s.erase(s.begin());`. Denna variant gör att ett `set` även kan användas som en så kallad *prioritetskö* - en kö där alla element är sorterade, och elementet man plockar ut hela tiden är det lägsta. I slutet av lektionen kommer vi titta på en implementation av just en prioritetskö, en *binär heap*, som man dock inte kan göra uppslagningar eller borttagningar från (annat än av det lägsta elementet).

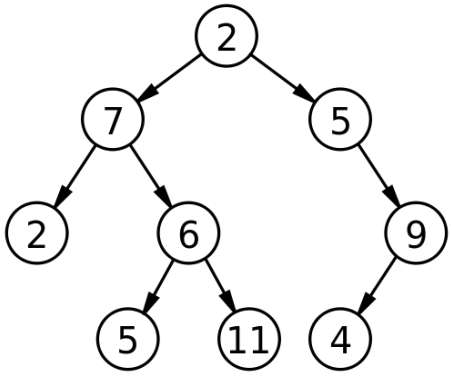
Värt att känna till är även multi-motsvarigheterna till mängden och uppslagningen, nämligen `multimap` och `multiset`. Dessa fungerar som vanligt, undantaget att man kan lagra *flera* kopior av varje element. Jämför:

```
1  multiset<int> s;
2  s.insert(7);
3  s.insert(3);
4  s.insert(3);
5  s.insert(1);
6
7  for(set<int>::iterator it = s.begin(); it != s.end(); ++it){
8    cout << *it << endl;
9  }
```

som istället skriver ut `1 3 3 7`.

Binära träd

Ett *binärt träd* är ett träd där vi låtit en av noderna i trädet vara *roten* i trädet. Varje nod har sedan 0, 1 eller 2 *barn* i trädet.



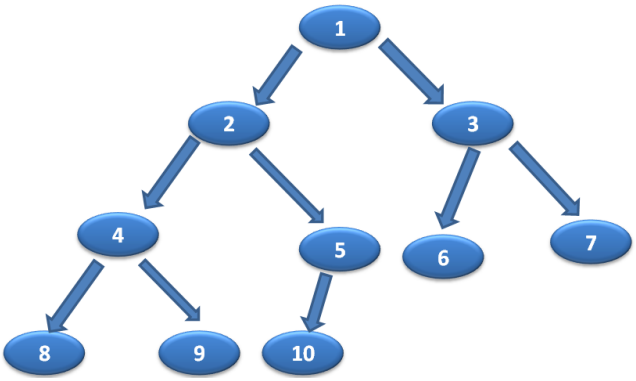
Figur 4: Ett binärt träd

I figuren är noden med värdet 2 trädets rot. Dess barn är noderna med värdena 7 och 5. Eftersom 7 är barn till 2 kallar vi 7 för *föräldern* till 2.

Vi kallar en nod som inte har något barn för *löv*. I figuren är alltså noderna med värdena 2, 5, 11 och 4 trädets löv.

En nods *nivå* är dess avstånd från roten. Roten har alltså nivån 0, rotens barn har nivån 1, rotens barnbarn har nivån 2 och så vidare.

Vi kallar ett binär träd *komplett* om alla nivåer är fulla, förutom möjligtvis den sista. Om den sista nivån inte är full måste alla noder vara så långt till vänster som möjligt.



Figur 5: Ett komplett binärt träd

I ett komplett träd kan vi numrera alla noder som i figur 5. Vi numrerar dem uppifrån och ned, vänster till höger.

Denna representation gör att vi enkelt kan lagra ett binärt träd i en vector. En viktig egenskap med denna numrering är att det är väldigt enkelt att beräkna vilket nummer en nods förälder, vänstra barn och högra barn har. Om en viss nod har nummer i har föräldern helt enkelt nummer $\frac{i}{2}$, det vänstra barnet har nummer $2i$, och det högra barnet har nummer $2i + 1$.

Vi implementerar dessa som funktioner:

Obs: om du använder en vector för att representera ett binärt träd på detta sätt måste den ha storleken $n + 1$ där n är antalet element i trädet, eftersom trädet är ett-indexerat och vectorn är noll-indexerad!

```
1: Funktion PARENT(i)
2:     returnera i/2
3: Funktion LEFT(i)
4:     returnera 2i
5: Funktion RIGHT(i)
6:     returnera 2i + 1
```

Heapen

En *heap* är en speciell variant av ett komplett binärt träd. I heapen gäller nämligen alltid följande villkor: *en nod ar alltid högre värde än sina barn*. Detta villkor är transitivt, vilket innebär att en nod alltid har högre värde än alla sina barn, barnbarn, osv. Framförallt kommer roten i trädet alltid vara det största värdet i hela heapen, och det är denna egenskap vi vill åt. Heapen är alltså en datastruktur som vi kommer kunna använda för att alltid få ut det högsta (eller lägsta, om vi vänder på villkoret) värdet av en samling objekt.

```
1: Funktion GET-MAX(tree)
2:     returnera tree[1]
```

De ytterligare operationerna vi vill att heapen ska stödja är att lägga till element, ta bort det största värdet, samt öka värdet av en nod.

När vi lägger till ett element i trädet lägger vi först till det i slutet av den vector som vi sparar vårt träd i. På så viss kommer trädet fortsätta vara komplett. Om värdet vi lade till är större än sin förälder tillkommer dock problemet att vi kanske bryter mot heap-villkoret. Vi kan behöva byta plats på noden vi lade till samt dess förälder. Vi kan till och med behöva göra det upprepade gånger om värdet vi lade till är väldigt stort. Vi implementerar det med följande procedur:

```
1: Funktion ADD-ELEMENT(x, tree)
2:     tree.push_back(x)
3:     Bubble − Up(tree.size() − 1, tree)
```

```
1: Funktion BUBBLE-UP(idx, tree)
2:     så länge idx > 1
3:         om tree[idx] > tree[Parent(idx)]
4:             Byt plats på tree[idx] och tree[Parent(idx)]
5:             idx ← Parent(idx)
6:         annars
7:             Avbryt loopen
```

[1]

När vi tar bort det största elementet i heapen kommer trädet inte längre vara komplett. För att åtgärda det ersätter vi elementet med det sista elementet i trädet. Nu kan vi ha problemet att detta element inte uppfyller heap-villkoret. För att fixa det gör vi som när vi lade till ett element, fast tvärtom. Vi tar istället elementet och byter plats på det med det största av sina två barn, ända tills elementet antingen är ett löv eller faktiskt är större än båda sina barn.

```
1: Funktion REMOVE-MAX(x, tree)
2:     tree[1] ← tree[tree.size() − 1]
3:     tree.pop_back()
4:     Bubble − Down(1, tree)
```

```
1:  Funktion BUBBLE-DOWN(idx, tree)
2:      så länge true
3:          largest ← idx
4:          om Left(idx) < tree.size() och tree[Left(idx)] > tree[largest]
5:              largest ← Left(idx)
6:          om Right(idx) < tree.size() och tree[Right(idx)] > tree[largest]
7:              largest ← Right(idx)
8:          om largest == idx
9:              Avbryt loopen
10:         annars
11:             Byt plats på tree[idx] och tree[largest]
12:             idx ← largest
```

[1]

Hemtal

Kattisinlämning 1 (id: dagy:heap). Lätt

Kattisinlämning 2 (id: dagy:tagvaxling). Lätt

Kattisinlämning 3 (id: guessthedatastructure). Medel

Kattisinlämning 4 (id: dagy:rostkophard). Medel

Kattisinlämning 5 (id: juryjeopardy). Svår

Kattisinlämning 6 (id: dagy:talfamiljer). Svår