Föreläsning 9

Sortering

Föreläsning 9

- Sortering
- Sortering och Java API
- Urvalssortering
- Insticksortering
- Söndra och härska
- Shellsort
- Mergesort
- Heapsort
- Quicksort
- Bucketsort
- Radixsort

Sortering

Med sortering ska vi här mena att ordna element efter elementens nycklar i stigande (eller fallande) ordning

Vi ska lära oss använda sorteringsmetoderna i Java's API men också lära oss implementera egna. Det gör vi av flera skäl:

- För att kunna implementera dem vid behov
- För att förstå när vi ska använda vilken metod
- För att algoritmerna är bra exempel på problemlösning och idéerna är tillämpbara även på andra problem
- För att träna oss på att analysera algoritmer

Egenskaper hos sorteringsalgoritmer

- Antal jämförelser: värst, medel, bäst
- Antal platsbyten: värst, medel, bäst
- Minnesbehov
- Stabilitet: Behåller den ordningen på element som är lika stora?

Sortering i Java API

- java.util.Arrays.sort kan anropas med en array av godtycklig primitiv typ eller av object. Består arrayen av Object måste dessa implementera Comparable (och därmed ha compareTo-metoden) eller så måste en Comparator skickas med.
- Primitiva datatyper sorteras med quicksort och objekt med mergesort. Båda är O(*n* log *n*)
- java.util.Collections.sort sorterar collections som implementerar List (arrayList och LinkedList). Också här måste man implementera Comparable eller skicka med en Comparator. Denna kopierar objekten till en array, sorterar arrayen och kopierar tillbaka.

Method sort in Class Arrays	Behavior
`public static void sort(int[] items)`	Sorts the array `items` in ascending order.
`public static void sort(int[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items[toIndex]` in ascending order.
`public static void sort(Object[] items)`	Sorts the objects in array `items` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `items` must implement the `Comparable` interface and must be mutually comparable.
`public static void sort(Object[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items[toIndex]` in ascending order using their natural ordering (defined by method `compareTo`). All objects must implement the `Comparable` interface and must be mutually comparable.
`public static <T> void sort(T[] items, Comparator<? super T> comp)`	Sorts the objects in `items` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`.
`public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp)`	Sorts the objects in `items[fromIndex]` to `items[toIndex]` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`.
Method sort in Class Collections	**Behavior**
`public static <T extends Comparable<T>> void sort(List<T> list)`	Sorts the objects in `list` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `list` must implement the `Comparable` interface and must be mutually comparable.
`public static <T> void sort (List<T> list, Comparator<? super T> comp)`	Sorts the objects in `list` in ascending order as defined by method `comp.compare`. All objects must be mutually comparable.

Läs mer i KW 8.1

Algoritmer

Vi kommer att fokusera denna föreläsning på att undersöka olika sorteringsalgoritmer. Andra algoritmer och varianter av dessa kan du hitta massor av på nätet.

I diskussionen tänker vi oss för enkelhetens skull alltid att vi sorterar en array a[] med *n* element av typen int och att index går från 0 till *n*-1.

Urvalssortering (Selection Sort)

Ide:

Hitta det minsta elementet och sätt det först. Hitta det näst minsta elementet och sätt det på nästa plats osv.

Algoritm:

För index = 0 till n-2

    minIndex = index

    För i = index+1 till n-1

        Om a[i]<a[minIndex]

            minIndex = i

    byt plats på a[index] och a[minIndex]

Implementera en static selectionSort för int[]

Analys urvalssortering

```
För plats = 0 till n-2
  minIndex = plats
    För i = plats+1 till n-1
     Om a[i]<a[minIndex]
       minIndex = i
  byt plats på a[plats] och a[minIndex]
```

Antalet jämförelser

$(n-1)+(n-2)+(n-3)+\ldots+1=n(n-1)/2=O(n^2)$

oberoend array så att Bästa, medel, värsta alla är $O(n^2)$

Antal platsbyten: $n-1=O(n)$

Minneskrav: O(1) (utöver arrayen)

Stabil: Ej med normal implementering

Användning

Den är enkel och intuitiv att implementera. Dessutom om man har en array av struktar i ett språk som C där nyckeln är liten och går snabbt att jämföra medans datat i structen är enormt och därmed tar tid att flytta kan denna metod vara optimal eftersom den behöver mycket få platsbyten. I Java flyttar vi normalt endast referenser så då spelar detta mindre roll.

Insticksortering (Insertion sort)

Ide:

Om vi har ett sorterat material kan vi stoppa in ett tal till på rätt ställe och då har vi fortfarande ett sorterat material. Vi börjar med första talet som sorterat material och stoppar in nästa tal. Nu har vi en sorterad lista med två tal. Nu stoppar vi in tredje talet på rätt ställe.

Algoritm:

För index = 1 till n-1

    Sätt in a[index] på rätt plats i arrayen: a[0],…,a[index-1]

Detaljerad:Kolla!

```
För index = 1 till n-1
        data = a[index]
        dataIndex=index
        Medans dataIndex>0 och data<a[dataIndex-1]
                a[dataIndex] = a[dataIndex-1]
                dataIndex--
        a[dataIndex] = data
```

Implementera en static insertionSort för int[]

Analys insticksortering

```
För index = 1 till n-1
  data = a[index]
  dataIndex=index
  Medans dataIndex>0 och data<a[dataIndex-1]
    a[index] = a[dataIndex]
    dataIndex--
  a[dataIndex] = data
```

Antal jämförelser

Värst: $1+2+3+\ldots+(n-1)=O(n^2)$

Bäst: $n-1=O(n)$ (sorterad)

Tilldelningar

Följer jämförelserna: Värst $O(n^2)$, Bäst $O(n)$

Observera att vi här inte gör ett byte (swap med tmp-lagring) utan bara en tilldelning

Minneskrav: O(1) (utöver arrayen)

Stabil: Ja

Användning

Även om den gör färre jämförelser än urvalssortering (ca hälften) är den fortfarande $O(n^2)$ i snitt och olämplig för stora arrayer. Däremot är den en av de snabbaste algoritmerna för små arrayer. Vad är då nyttan med att sortera en liten array snabbt? Jo en del andra algoritmer sorterar genom att sortera små delarrayer. Dessa algoritmer kan då optimeras genom att små arrayer sorteras med insticksortering. Dessutom är den snabb O(n) på sorterade arrayer och då också ganska snabb på ganska sorterade arrayer. Något vi ska använda i nästa algoritm.

Jämförelse av kvadratiska sorteringsalgoritmer

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

Söndra och härska (Divide and conquer)

Innan vi ger oss på några lite mer avancerade sorteringsalgoritmer ska vi säga några ord om det design paradigm shellsort, mergesort och quicksort hör till.

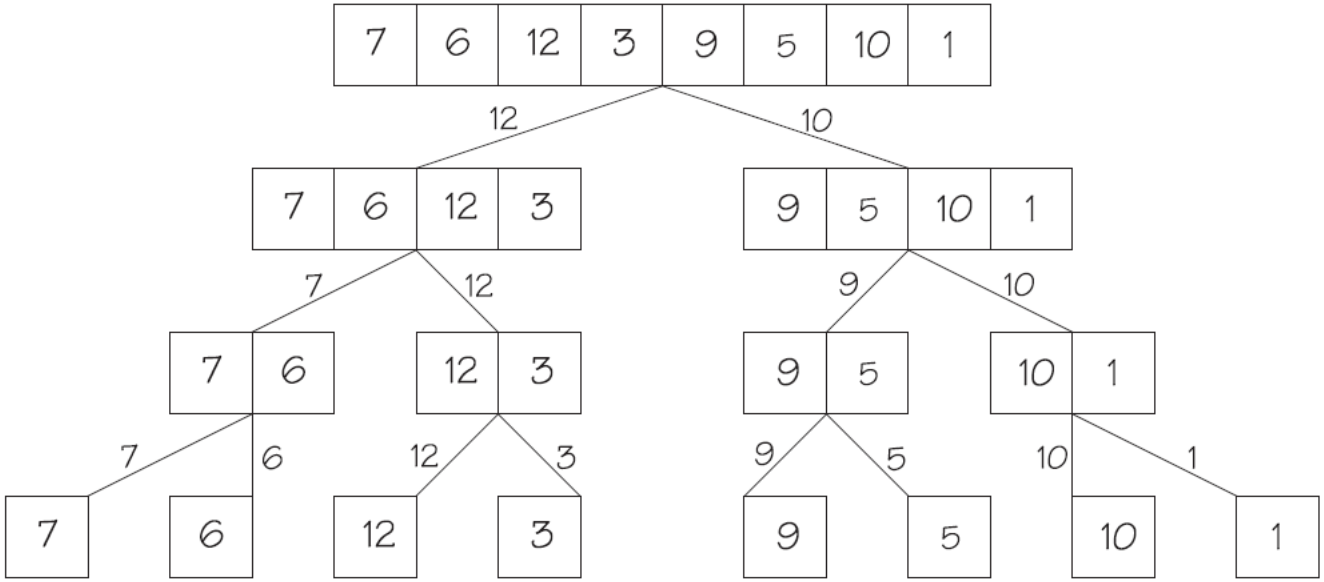
En söndra och härska algoritm bryter ner ett problem i två eller fler subproblem av samma (eller närliggande) typ. Dessa i sin tur brytes också ner i två eller fler subproblem. Algoritmen fortsätter så till dess att subproblemen blivit så små att de är lätta att lösa och kombinerar då dellösningarna till en lösning på hela problemet oftast i samma steg som problemet bröts ner.

Precis som det låter gör man detta oftast och enklast med rekursion.

Exempel på söndra och härska

Största värdet i en array:

```
int max(int a[],int low,int high){
  int m1,m2;
  if(low==high)
    return a[low];
  else{
    m1=max(a,low,(low+high)/2);
    m2=max(a,(low+high)/2+1,high);
    if (m1>m2)
      return m1;
    else
      return m2;
  }
}
```


7 6 12 3 9 5 10 1
12
10
7 6 12 3
9 5 10 1
7
12
9
10
7 6
12 3
9 5
10 1
7
6
12
3
9
5
10
1
7 6 12 3 9 5 10 1

Exponentiering

Hur många multiplikationer behöver man utföra för att beräkna 7^4?

```
exp(a,b)
  Om b==1 return a
  p=exp(a,b/2)
  Om b%2==0 return p*p
  return p*p*a
```

Precis som binär sökning i sorterad array är detta inte riktigt en söndra och härska algoritm eftersom vi delar upp problemet i ett subproblem. Namnet decrease and conquer har föreslagits för dessa algoritmer.

Shellsort (Donald L. Shell)

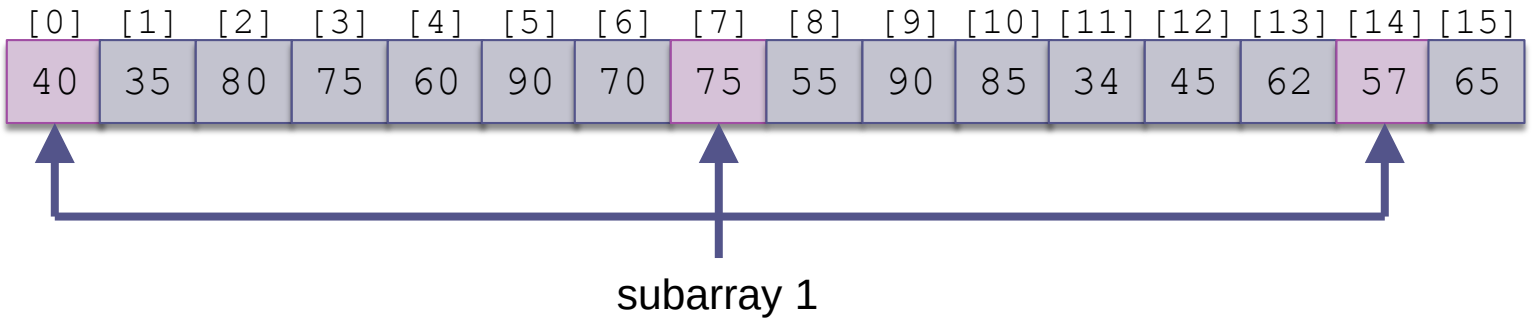
Kanske inte det tydligaste exemplet på söndra och härska men kan ses som en sådan. Grundiden är att dela upp vår array i små subarrayer (utspridda över arrayen) och sortera dessa med insticksortering. Vi har nu en något mer sorterad array. Denna delar vi upp i lite färre subarrayer som vi sorterar med insticksortering. Vi har nu en ännu mer sorterad array som vi delar upp i ännu färre subarrayer som vi sorterar med insticksortering. Till slut sorterar vi hela den nu nästan sorterade arrayen med insticksortering.

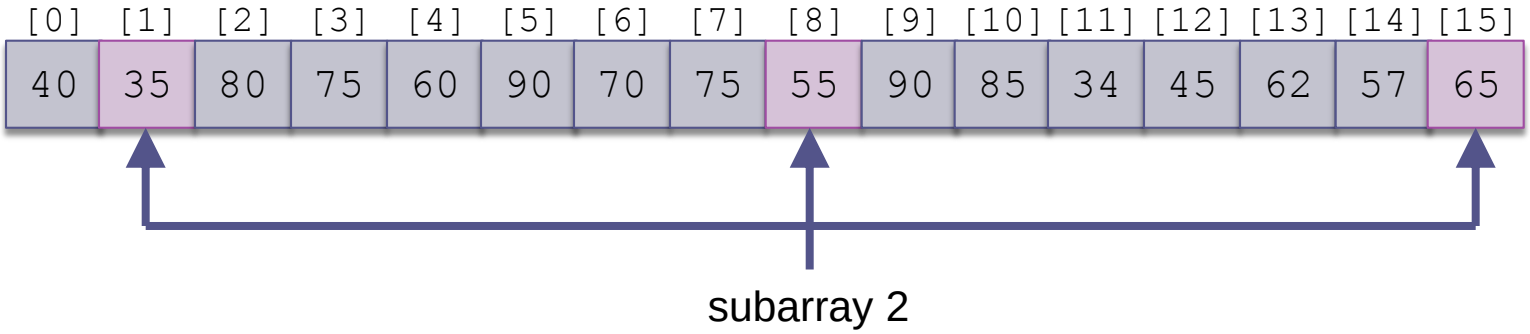
Anledningen till att detta blir effektivt är:

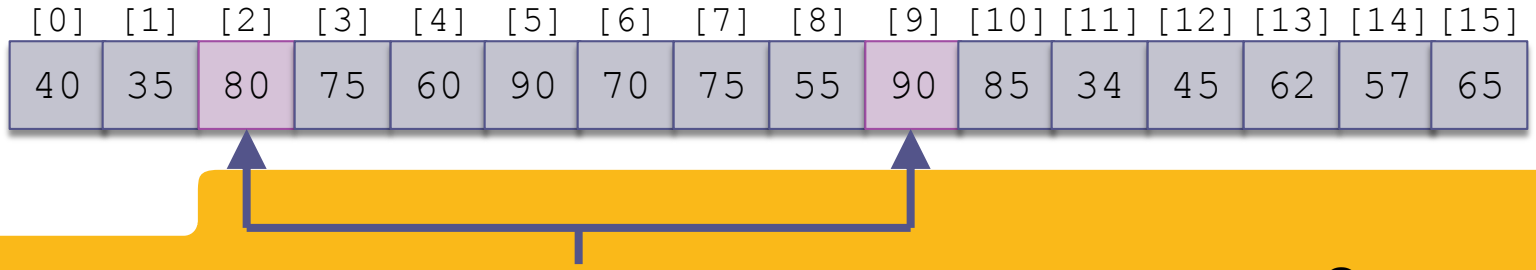
1. Insticksortering är snabb på små arrayer
2. Instickningssortering drar nytta av ordning som redan finns i arrayen (O(n) för en sorterad array)
3. Element som befinner sig på helt fel plats flyttas direkt långt istället för att sakta via jämförelser med grannar flyttas till rätt plats

Exempel på en sortering

För att förstå principen ska vi titta på en sortering av en konkret array med 15 platser. Först väljer vi ett gap som ger oss våra subarrayer. Vi börjar här med gap = 7. Vilket ger totalt 7 subarrayer

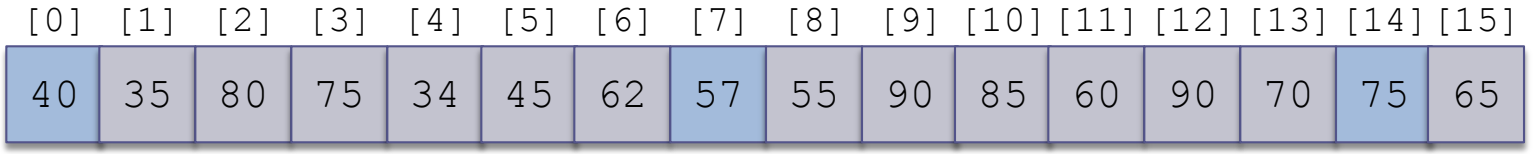




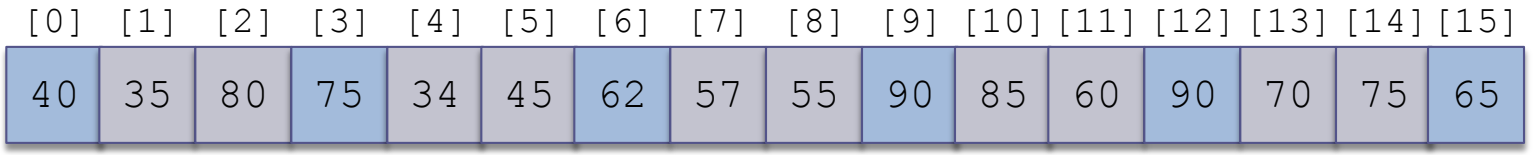


Osv.

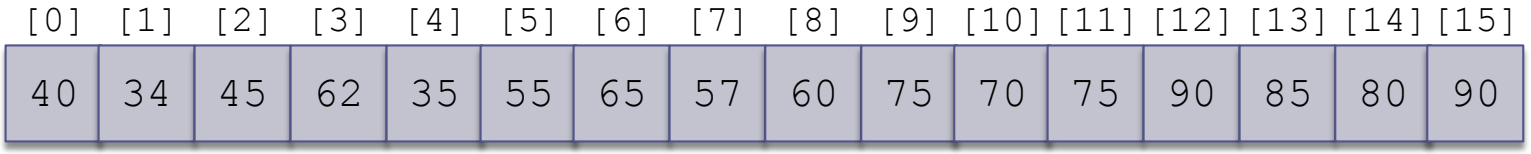
Dessa sju subarrayer sorteras nu var och en (dock inte en klart och sedan nästa) med instickssortering och vi får:



Vi väljer nu gap=3 och får då tre subarrayer



Dessa tre subarrayer sorteras nu var och en med instickssortering och vi får:



Vi väljer nu gap=1 och får då en array och sorterar denna med instickssortering. Detta går då fort då den redan är nästan sorterad och vi för varje instick inte behöver leta så långt innan vi hittar rätt plats.

Shell sort algoritm

```
gap = n/2
medans gap>0
        För index = gap till n-1
                Instickssortera elementet i sin subarray
        Om gap==2 gap=1
        annars gap=gap/2.2
Detaljerat:
gap = n/2
medans gap>0
        För index = gap till n-1
                data = a[index]
                dataIndex=index
                Medans dataIndex>gap-1 och data<a[dataIndex-gap]
                        a[dataindex] = a[dataIndex-gap]
                        dataIndex-=gap
                a[dataIndex] = data
        Om gap==2 gap=1
        annars gap=gap/2.2
```

Analys Shell sort

Hur vi väljer våra gap är avgörande för prestandan. Den generella algoritmen är ännu inte analyserad och ingen teori ger oss de bäst gapen. Väljer man 2^k (dvs 16, 8, 4, 2, 1) får man $O(n^2)$ men väljer man 2^k-1(dvs 15, 7, 3, 1) får man $O(n^{3/2})$. Andra val ger ännu bättre resultat. Empiriskt har man visat att om man börjar med n/2 och sedan delar gapet på 2,2 får man $O(n^{5/4})$ eller kanske $O(n^{7/6})$.

Minneskrav: O(1) (utöver arrayen)

Stabil: Nej

Användning:

Är långsammare än de vi ska titta på senare men har den fördelen att den inte använder något extra utrymme (in place) och används därför ibland i inbyggda system (finns i linuxkärnan). Den kan också användas för mindre subarrayer i andra algoritmer som riskerar att bli ineffektiva på mindre arrayer.

Mergesort

Ett mycket typiskt exempel på söndra och härska.

Algoritm:

Dela upp arrayen i två lika stora arrayer

Sortera vänster array

Sortera höger array

Sammanfoga de två arrayerna till en sorterad array (merge)

Hur sorterar vi då vänster array? Jo vi delar upp den i två delar….

Stoppvillkoret är att en array med ett element är sorterad

Algoritm merge

villkor: a[0],…,a[n-1] och b[0],…,b[m-1] sorterade

merge(a,b,c)

```
indexa=0,indexb=0, indexc=0
Medans indexa<n och indexb<m
        Om a[indexa]<=b[indexb]                 //(= - ger stabil mergesort)
                c[indexc++]=a[indexa]
                indexa++
        annars
                c[indexc++]=b[indexb]
                indexb++
Medans indexa<n
        c[indexc++]=a[indexa]
        indexa++
Medans indexb<m
        c[indexc++]=b[indexb]
        indexb++
```

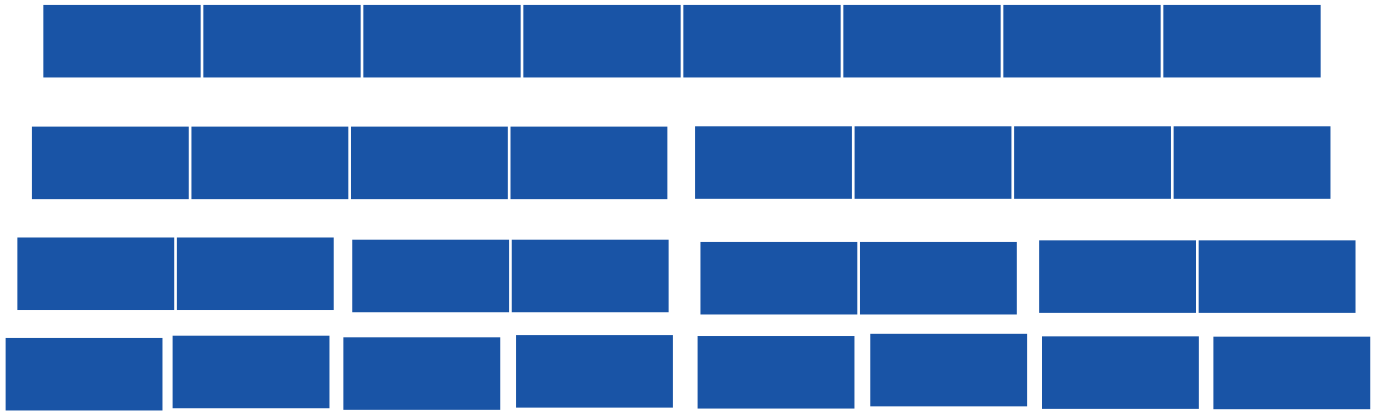
Tidsåtgången är O(n+m) och extra minne krävs O(n+m) med denna algoritm

Algoritm för mergesort

mergesort(a)

Om a.length == 1 return
b[0,…]=a[0,…,a.length/2-1]
c[0,…]=a[a.length/2,….,a.length-1]
mergesort(b)
mergesort(c)
merge(b,c,a)
return

Analys mergesort



I varje steg halveras längden av arrayen vilket ger att antalet steg blir $\log_2(n)$. I varje steg kopieras alla element två gånger vilket ger 2n. Tilldelningar:O(nlogn)
Jämförelser: Värst:O(nlogn), Bäst:O(n)
Minnesbehovet när det är som störst är förutom arrayen 2n = O(n). Det finns dock versioner som sorterar på plats (dessa får dock $O(n(\log n)^2)$) och versioner som endast behöver n/2 extra utrymme.
Stabil: Ja
Variant: Det förekommer att man slutar använda mergesort vid en viss storlek S och då istället gör instickssortering. S är då typiskt valt så att hela arrayen får plats i cashminnet.

Användning och jämförelse

- Heapsort är ofta snabbare och kräver mindre minne men är inte stabil.
- Quicksort är normalt snabbare för arrayer som får plats i ram-minnet men har ett värsta fall som är $O(n^2)$.
- Mergesort används bla då man sorterar länkade listor där O(n) för att nå ett element saktar ner andra algoritmer.
- Den fungerar också bra att parallellisera och har den fördelen att den kan sortera en del i taget. Merge kan dessutom användas utan att man behöver tillgång till hela materialet på en gång om stora material måste ligga på tex fil.
- Mycket vacker enkel algoritm som belysande exempel på rekursion och mera specifikt söndra och härska

Heapsort

Heapsort använder en heap för att sortera:

Algoritm

Sätt in elementen ett efter ett i en heap

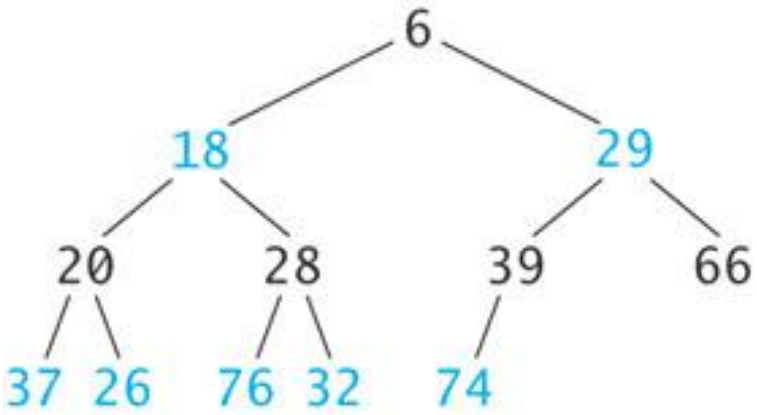
Ta ut elementen ett efter ett och sätt tillbaka dem i arrayen

Men vad är en heap. En heap är ett speciellt sorts binärt träd. Det används för att implementera en prioritetskö och för att utföra heapsort. Låt oss börja med att studera detta träd.

Heap

En heap är ett komplett binärt träd där:

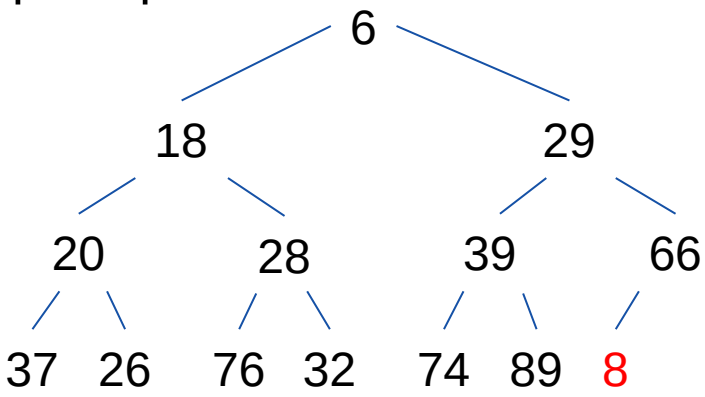
- rotens nyckel är en av de minsta nycklarna i trädet
- varje icke tomt subträd är en heap



Insättning

Algoritm:

1. Sätt in ny nod på första lediga plats
2. Medans noden inte är rot och nodens nyckel är mindre än förälderns nyckel:
   byt plats på förälder och nod



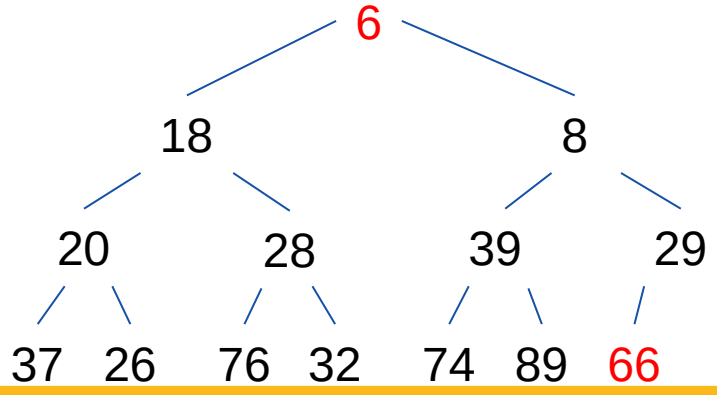
Borttagning

Vi tar endast bort rot-noden

Algoritm

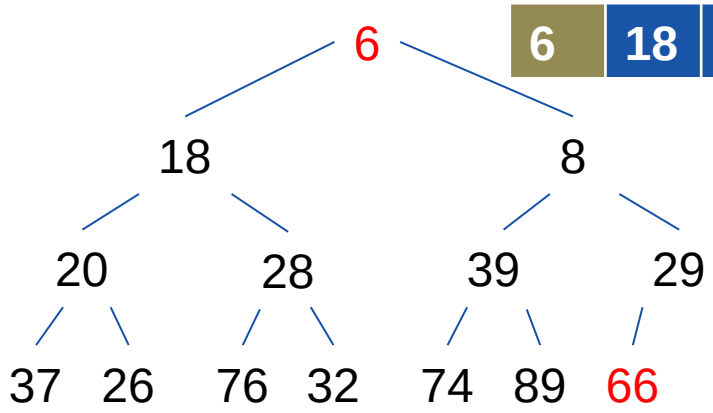
1. Byt ut rotnoden med sista noden i trädet S
2. Så länge S har barn och S har större nyckel än något av sina barn:

   swappa S med minsta barnet



Implementering av en heap

Eftersom en heap är ett komplett binärt träd är det onödigt att implementera det som en länkad datastruktur. En array fungerar perfekt. Vi lägger helt enkelt nivåerna med elementen i ordning från vänster till höger. En nod med index p har då vänster barn med index 2p+1, höger barn med index 2p+2 och sin förälder med index (p-1)/2.



6	18	8	20	28	39	29	37	26	76	32	74	89	66

Heapsort

Tillbaka till heapsort. Algoritmen:

Sätt in elementen ett efter ett i en heap

Ta ut elementen ett efter ett och sätt tillbaka dem i arrayen

Fungerar bra men har ett problem. Den kräver n extra minne. Detta visar sig onödigt vi kan implementera heapsort utan att använda extra minne. För att kunna använda arrayen som minne ska vi göra en ändring. Vi ska använda en omvänd heap där rotens nyckel är största nyckeln i trädet.

Heapsort inplace

heapstorlek = 1
Medans heapstorlek<n
    e = a[heapstorlek]
    sätt in e i heapen
    heapstorlek++ (kan anses ingå i att sätta in)
Medans heapstorlek>0
    ta bort ett element från heapen och sätt e= detta
    heapstorlek-- (kan anses ingå i att ta bort)
    a[heapstorlek]=e

Läs mer i boken!

Quicksort

Får ni läsa in själva från boken. Läs gärna från fler källor. Är den t.ex stabil? Vad är starka och svaga sidor? När används den?

Bucketsort

Är ej en jämförelsealgoritm och kan därför vara bättre än O(nlogn).

Ide: Säg att vi ska sortera heltal med möjliga värden 0-10000. Vi skapar då en array res med 10001 platser som är nollställda. Sedan kan vi från fil eller array helt enkelt gå igenom materialet och göra res[inläst]++.

Den här varianten blir oslagbart effektiv O(n) med den komplikationen att vi måste skapa en för många fall för stor array för att täcka in alla fall (dock kan allt tolkas som heltal).

Normalt brukar man med bucketsort använda lite större hinkar (säg 0-9, 10-19, 20-29, osv). Man måste då faktiskt flytta tal till de olika hinkarna och dessutom sedan sortera dessa (med samma metod eller någon annan metod) och sedan flytta tillbaka dem.

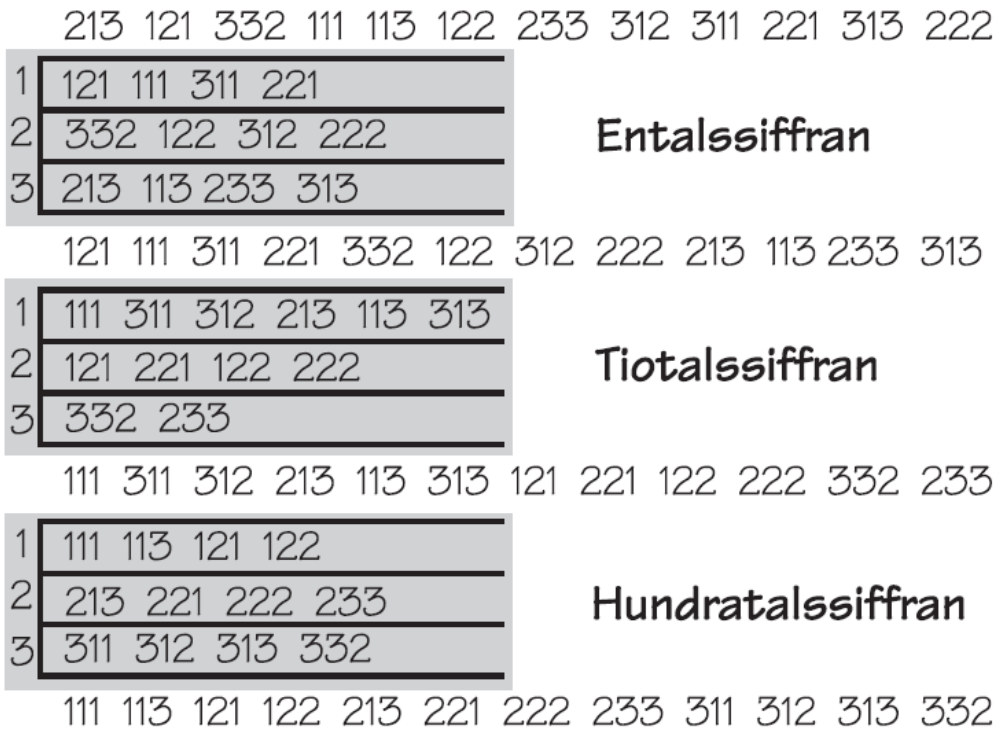
Titta gärna på pigeonhole sort också

Radixsort

LeastSignificantDigit Radix sort:

- Gruppera talen baserat på den minst signifikanta siffran.
- Upprepa grupperingen för varje mer signifikant siffra.

Grupperingen görs effektivt med bucketsort så att inga jämförelser behövs utan vi behöver endast gå igenom datat en gång per sortering.



Läs gärna mer även om denna