Algoritmer for søgning og sortering

Dan Witzner Hansen

Søgning og sortering

- **Søgning:** Givet et array af heltal `int[]` *`arr`*, og et tal `x`, afgør om der findes et `i` således at `arr[i] = x`, og returner et sådant `i` hvis det findes

- **Sortering:** Givet et array af heltal `int[] arr` omarranger elementerne, så de står sorteret efter rækkefølge

Anvendelser

- Søgning i telefonbog
- Søgning i databaser
- Sortering af søgeresultater
- Sortering af inbox
- Både søgning og sortering foregår bag kuliserne i mange applikationer

Algoritmer

- En algoritme er en måde at løse et problem på
- En implementation af en algoritme er et program, der løser problemet ved at følge algoritmen
- En algoritme er en abstrakt ide
- En algoritme er sproguafhængig
- Man kan også tale om algoritmer i andre kontekster (ikke computere)

Dagens mål

- betragte forskellige algoritmer til at søge og sortere
- introducere køretider for forskellige algoritmer
- argumentere for at algoritmerne (faktisk implementationerne) er korrekte
- sammenligne fordele og ulemper ved de forskellige algoritmer

Mål med undervisning

- Algoritmer for søgning og sortering er almen dannelse
- I vil nok aldrig i praksis få brug for at implementere disse algoritmer
- Men I kan få brug for at overveje hvilken af disse der er mest hensigstmæssig i en given situation

lineær søgning

- **Søgning:** Givet et array af heltal `int[] arr`, og et tal `x`, afgør om der findes et `i` således at `arr[i] = x`, eller returner et sådant `i` hvis det findes

- **Lineær søgning:** Kig arrayet igennem fra ende til anden

Implementation af lineær søgning

```
int i;
boolean found;

public void linsearch(int x, int[] arr, int n) {
  i = 0; found = false;                           /* pp1 */
  while (!found && i < n)
    {                                             /* pp2 */
      if (arr[i] != x) i++;
      else             found = true;              /* pp3 */
    }                                             /* pp4 */
}
```

Binær søgning

- Antag at arrayet er sorteret

- Slå op midt i arrayet. Hvis `x` er mindre end det tal vi finder her, så kig i nederste halvdel af arrayet, ellers kig i øverste halvdel

- Gentag dette: Slå op midt i den tilbageværende halvdel og afgør hvilken fjerdedel af arrayet, vi skal kigge i

- Gentag indtil man ikke kan halvere mere

Sammenligning af algoritmer

- Lineær søgning svarer til hvad man ville gøre hvis man havde telefonnummer og ville finde navn

- Binær søgning svarer til opslag i telefonbog

Implementation af binær søgning

```
public void binsearch(int x, int[] arr, int n) {
  int a = 0, b = n-1;
  found = false;                              /* pp1 */
  while (!found && a <= b)
    {                                         /* pp2 */
      i = (a+b) / 2;
      if (x < arr[i]) b = i-1;
      else if (arr[i] < x) a = i+1;
      else found = true;                      /* pp3 */
    }                                         /* pp4 */
}
```

Rekursiv binær-søgning

```
private int binarySearch(int[] a, int x, int low, int high){
        if (low > high) return -1;
        int mid = (low + high)/2;
        if (a[mid] == x) return mid;
    else if (a[mid] < x)
            return binarySearch(a, x, mid+1, high);
        else // last possibility: a[mid] > x
            return binarySearch(a, x, low, mid-1); }
```

Hvorfor virker algoritmerne?

Køretidsanalyse

- Lad os tælle hvor mange sammenligninger af heltal man skal lave i hver algoritme, som funktion af længden af arrayet
- Hvor mange skal man højst lave? (værste tilfælde)
- Hvor mange skal man lave i gennemsnit?

Analyse lineær søgning

Køretid for lineær søgning vokser som $f(n) = n$ i længden af arrayet

Binær søgning Analyse

For et array med N elementer kan ½ "glemmes" indtil der kun er 1 element tilbage.

N, N/2, N/4, N/8, ..., 4, 2, 1

– Hvor mange gange kan det gøres?

Tænk på det modsat

– Hvor mange gange skal vi gange med 2 for at nå N?

1, 2, 4, 8, ..., N/4, N/2, N

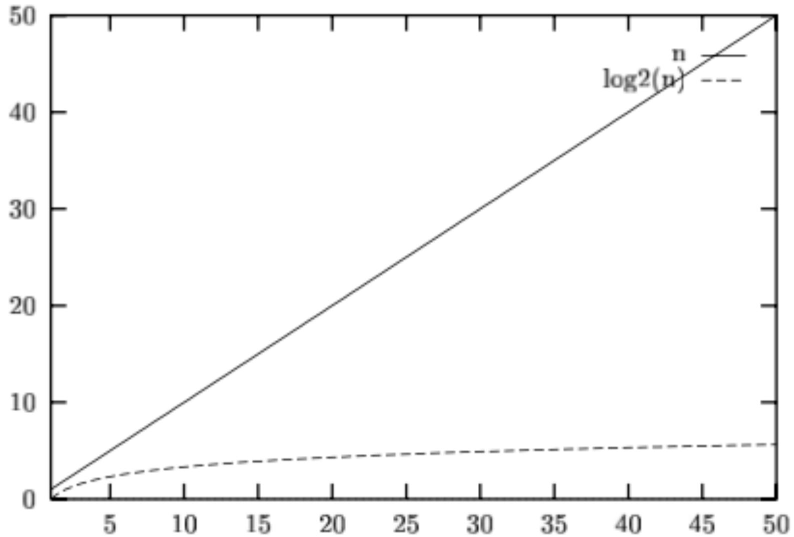
– Lad x være antallet af multiplikationer.

$2^x = N$

$\mathbf{x = \log_2 N}$

Sammenligning

- Køretid for lineær søgning vokser som f(n) = n i længden af arrayet
- Køretid for binær søgning vokser som log2(n) i længden af arrayet



Køretidsanalyser

- Bemærk: Køretidsanalyse var uafhængig af implementation
- Bemærk: Vi interesserede os kun for hvorledes køretiden afhang af størrelsen af input
- Husk at køretidsanalysen er baseret på teoretiske modeller af beregning
- Praktisk køretid kan afhænge af andet end antal sammenligninger
- F.eks. spiller lagerstruktur i computeren ind
- Teorien bør bekræftes af eksperimenter

Sammenligning af søgningsalgoritmer

- Binær søgning er klart hurtigere end lineær søgning
- Binær søgning virker kun korrekt, hvis arrayet er sorteret
- Hvis man vil lave mange søgninger kan det nogle gange betale sig at sortere først

UDVALGSSORTERING OG QUICKSORT

Udvalgssortering

- Engelsk: Selection sort
- Simpel sorteringsalgoritme
- Kigger listen igennem efter mindste element og sætter det først
- Kigger derefter listen igennem efter det næstmindste og sætter det ind som nummer 2
- Fortsætter således indtil listen er sorteret

Implementation

```
private static void swap(int[] arr, int s, int t) {
  int tmp = arr[s];  arr[s] = arr[t];  arr[t] = tmp;
}

// Selection sort

public void selsort(int[] arr, int n)
     // sort arr[0..n-1]
{                                              /* pp1 */
  for (int i = 0; i < n; i++)
    {                                          /* pp2 */
      int least = i;
      for (int j = i+1; j < n; j++)
        {
          if (arr[j] < arr[least])
            least = j;
        }
      swap(arr, i, least);                     /* pp3 */
    }                                          /* pp4 */
}
```

Køretid

- Hvor mange sammenligninger skal man bruge for at finde det mindste element i en liste af n elementer?

Køretid

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$$

- Så køretiden vokser kvadratisk i længden af listen

- I praksis vil det sige, at de andre algoritmer vi kommer til at se er langt hurtigere for større lister

Opgave

- Noterne opg 4.10.1.
- Dvs. Kør `selsort` manuelt på en liste med 6 elementer
- Noter hvordan listen ser ud efter hvert gennemløb af den ydre løkke
- Noter værdien af variablene `i`, `least` efter hvert gennemløb af den ydre løkke

Torsdag

- På torsdag vil jeg fortsættem med sortering

I dag

- Hobsortering
  - Køretider
  - Implementation
  - Korrekthed
- Sammenligning af sorteringsalgoritmer
- Opsummering

Læringsmål

- Til eksamen forventer vi at I kan
  - Forklare hver algoritme
  - Huske køretidresultater
  - I store træk udlede køretidsresultater
  - Kunne skitsere hvorfor hver algoritme virker
  - Kunne redegøre for fordele og ulemper ved hver algoritme
- Vi forventer ikke at I kan gennemgå korrekthedsbeviser i detaljer

Quicksort

- Del-og-hersk algoritme
- Vælg et element x (kaldet pivotelementet) fra listen
- Flyt alle elementer mindre end x hen før det, og alle elementer større end x hen efter det
- Sorter rekursivt listen af elementer mindre end x
- Sorter rekursivt listen af elementer større end x

Illustration

alle elementer, usorterede

opdel:

$\leq$ x, usorterede	x	$\geq$ x, usorterede

sorter:

$\leq$ x, sorterede	x	$\geq$ x, sorterede

færdig:

alle elementer, sorterede

Implementation

```
private void qsort(int[] arr, int a, int b) {
  if (a < b)
    {
      int i = a, j = b;
      int x = arr[(i+j) / 2];                        /* pp1 */
      do {                                           /* pp2 */
       while (arr[i] < x) i++;                       /* pp3 */
       while (arr[j] > x) j--;                       /* pp4 */
       if (i <= j)
         {
           swap(arr, i, j);
           i++; j--;
         }                                           /* pp5 */
      } while (i <= j);                              /* pp6 */
      qsort(arr, a, j);                              /* pp7 */
      qsort(arr, i, b);                              /* pp8 */
    }                                                /* pp9 */
}

public void quicksort(int[] arr, int n) {
  qsort(arr, 0, n-1);
}
```

Køretider

- Opdelingsskridtet kræver ca `n` sammenligninger
- I værste fald bliver man ved med at vælge det mindste element som pivot element. Da skal man i det rekursive kald sortere en liste på `n`-1 elementer
- Værste falds køretid

$$n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Gennemsnit

- Et andet ekstrem er at man vælger det midterste element
- Da får køretiden T(n) bestemt af ligningen

$$T(n) = n + 2 \times T(\frac{n}{2})$$

$$\begin{aligned}
T_{best}(n) &= n + 2 \cdot T_{best}(\tfrac{n}{2}) \\
&= n + 2 \cdot (\tfrac{n}{2} + 2 \cdot T_{best}(\tfrac{n}{4})) \\
&= n + n + 4 \cdot T_{best}(\tfrac{n}{4}) \\
&= n + n + 4 \cdot (\tfrac{n}{4} + 2 \cdot T_{best}(\tfrac{n}{8})) \\
&= n + n + n + 8 \cdot T_{best}(\tfrac{n}{8}) \\
&= \underbrace{n + n + \cdots + n}_{\log_2(n)\ \text{terms}} \\
&= n \log_2(n)
\end{aligned}$$

$$T(n) = n \times \log(n)$$

Opgave

- Kør quicksort manuelt på arrayet

$$\{35, 62, 28, 50, 11, 45\}$$

- Hold styr på arrayet, kaldestakken og variablene `a`, `b`, `i`, `j`

Konklusion for quicksort

- Quicksort har en gennemsnitskøretid assymptotisk med $n \times \log(n)$
- Quicksort har en køretid i værste tilfælde på n^2

HOBSORTERING

Køretider for quicksort

- Quicksort har en gennemsnitskøretid assymptotisk med $n \times \log(n)$
- Quicksort har en køretid i værste tilfælde på n^2

Hobsortering

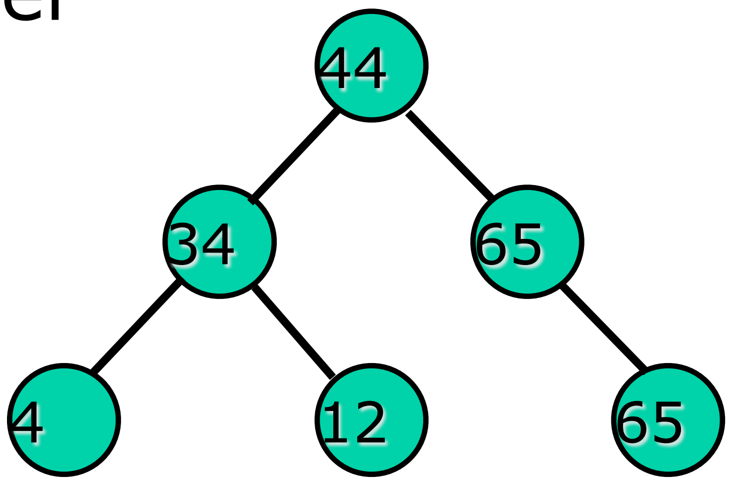
- Engelsk: Heapsort
- Har bedre teoretiske egenskaber end quicksort: $n \times \log(n)$
- Køretid er garanteret
- Kører i praksis almindeligvis 2-3 gange langsommere end quicksort

Hobsortering, algoritmen

- Minder om udvalgssortering
- Find største element og sæt det sidst, find derefter næststørste ...
- Afgørende forskel: Vi finder største element ved at holde elementerne i en særlig datastruktur (en hob)
- Hobsortering er mere kompleks end de foregående, men den er hurtig

Binære træer

- Et (binært) træ er enten et blad med et tal eller en knude med et tal og et eller to undertræer



- Man taler om børn, forældre, søskende etc
- På tegningen er knuden med 44 træets rod

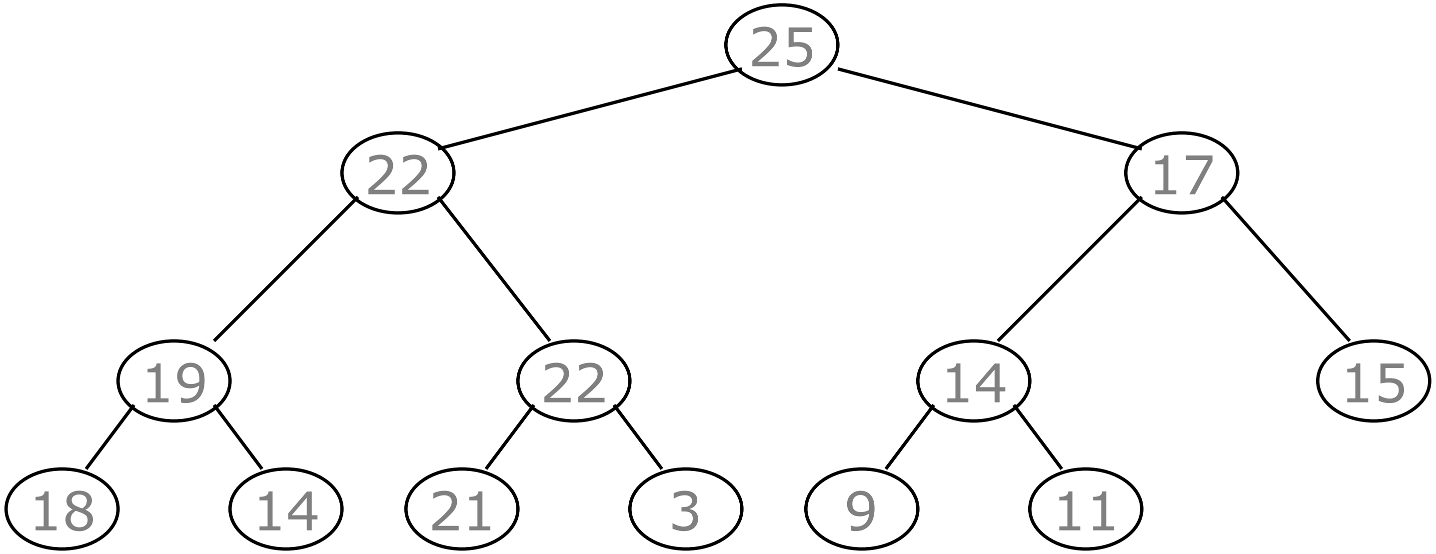
Træer i Java

```java
public class BinaryTree
{
    private BinaryTree leftChild;
    private BinaryTree rightChild;
    private int n;

    public BinaryTree(BinaryTree leftChild, BinaryTree rightChild, int n)
    {
        ...
    }

    public boolean isLeaf()
    {
        return (null == leftChild) && (null == rightChild);
    }

    public BinaryTree leftChild()
    {
        return leftChild;
    }
    ...
}
```

Lister som træer

- Vi vælger en anden løsning:
- Lister repræsenterer træer:
  - 0. element er rod
  - i. element har 2i+1 og 2i+2 som børn
- På den måde kan vi arbejde med træer i Java uden at introducere nye klasser
- Advarsel: Ikke alle træer svarer til lister!

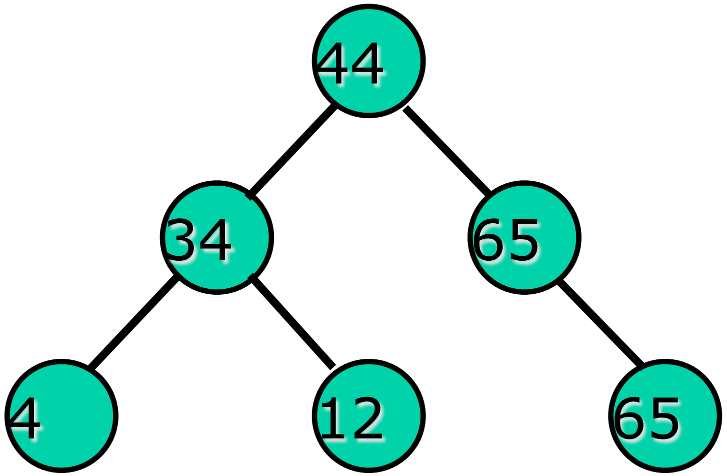
Eksempel



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

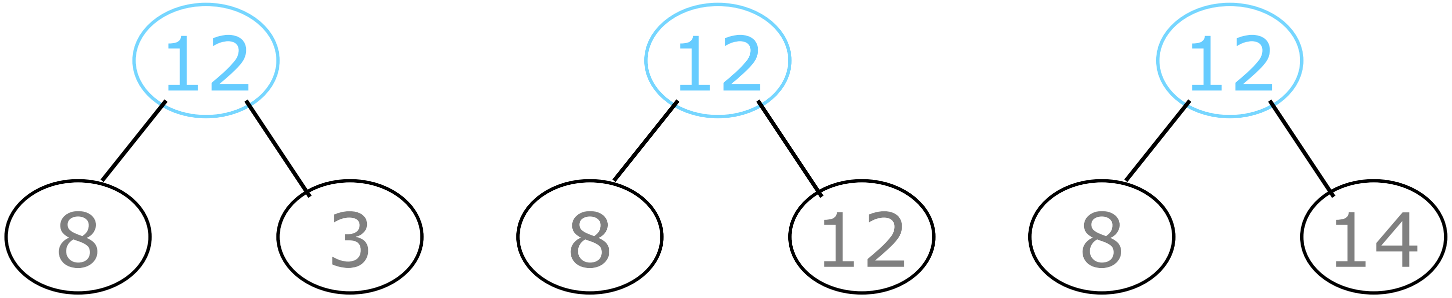
Hobe

- En knude tilfredsstiller *hobbetingelsen*, hvis dens børn har mindre (lig) værdi end den selv



- En *hob* er et træ, hvor alle knuder tilfredsstiller hobbetingelsen

Eksempel



Blå knude opfylder hobbetingelsen

Blå knude opfylder hobbetingelsen

Blå knude opfylder IKKE hobbetingelsen

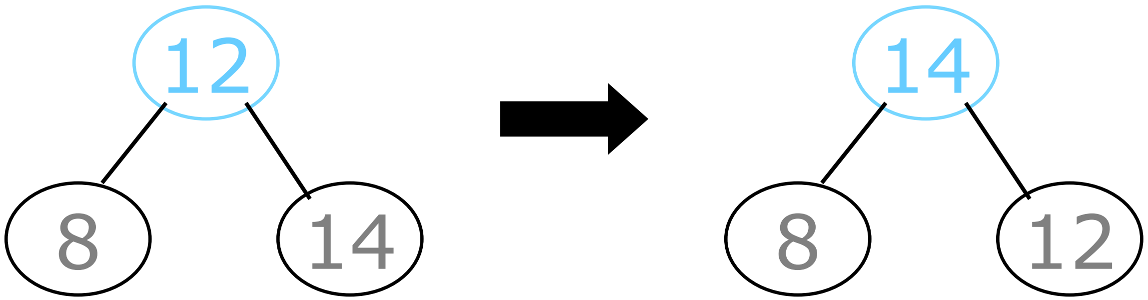
Hob egenskab

- I en hob er knuden det største element
- Næststørste element er et af knudens børn
- Mindste element er et blad

Hobsortering ide

- Arranger elementer i en hob
- Udtag rod (største element)
- Arranger resterende elementer i hob og gentag
- Afgørende: Vi skal kunne lave et træ om til en hob på en effektiv måde

Hobifikation basis

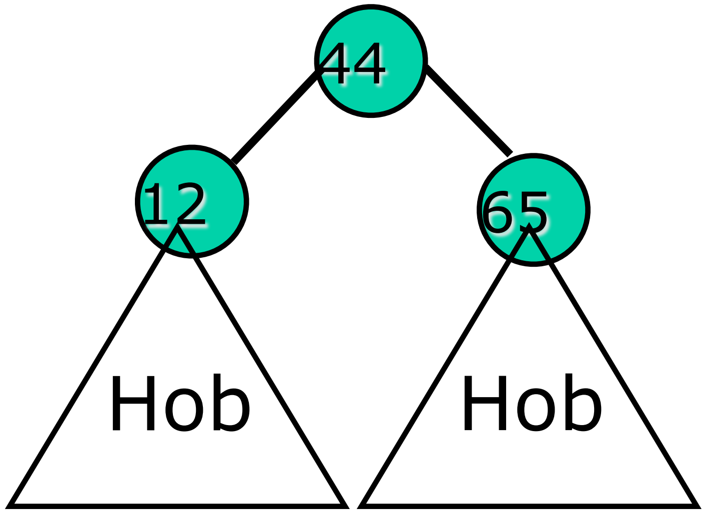


Blue knude har **ikke** hob-egenskab

Nu har blå knude hob-egenskab

Hobifikation af en knude

Antag først vi er givet et træ, hvor undertræer er hobe

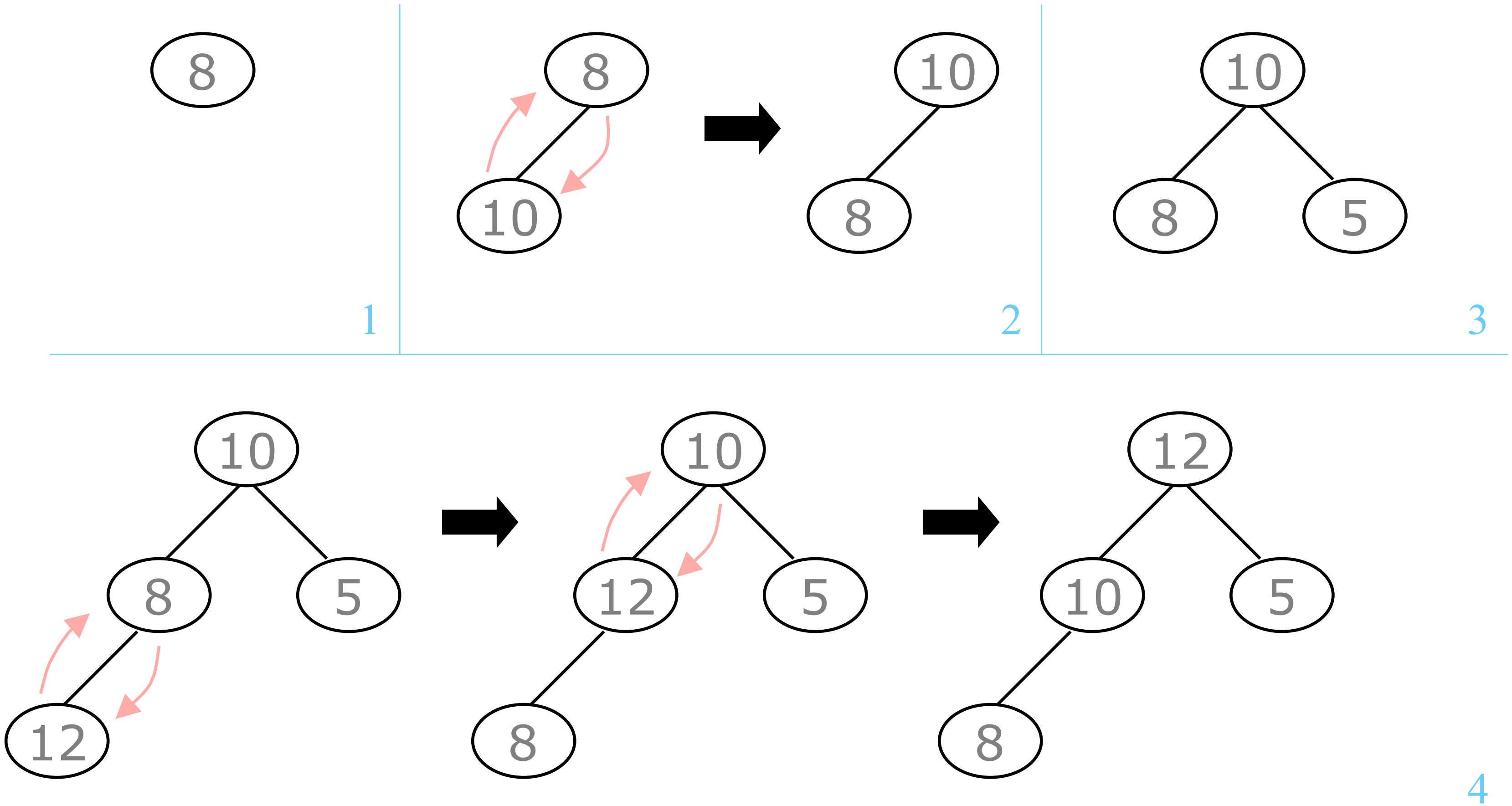


- Hvis vi bytter om på 44 og 65 risikerer vi at højre deltræ ikke længere er en hob
- Vi bliver da nødt til at kalde algoritmen rekursivt på højre deltræ

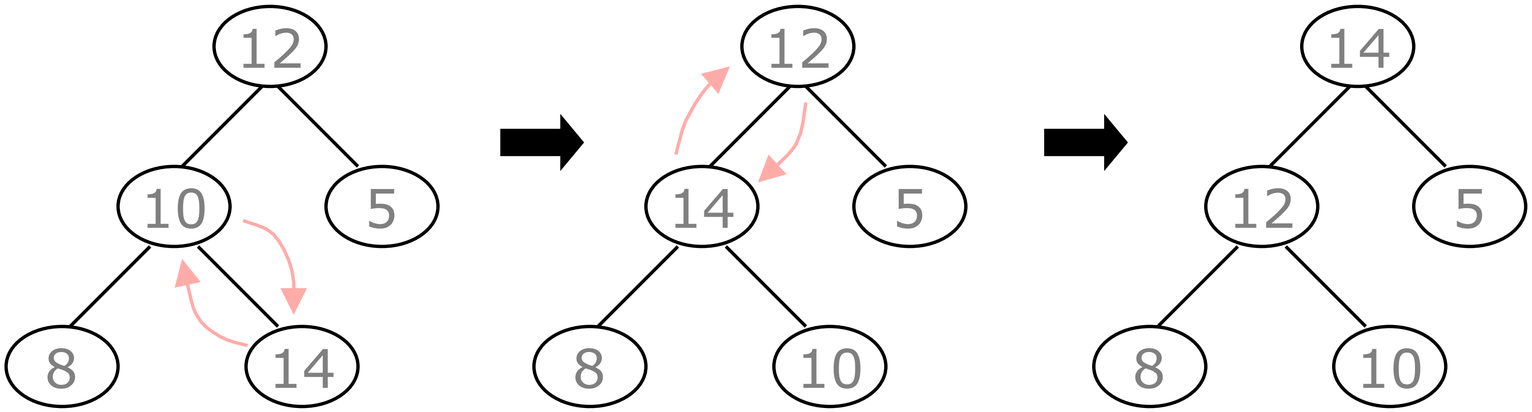
Hobifikation af træ

- Start nedefra: Hvert blad er allerede en hob
- Hobificer derefter alle næst-nederste knuder
- Fortsæt op, et niveau ad gangen
- Slut med at hobificere roden

Konstruktion af hob



Bemærk andre “børn” bliver ikke påvirket

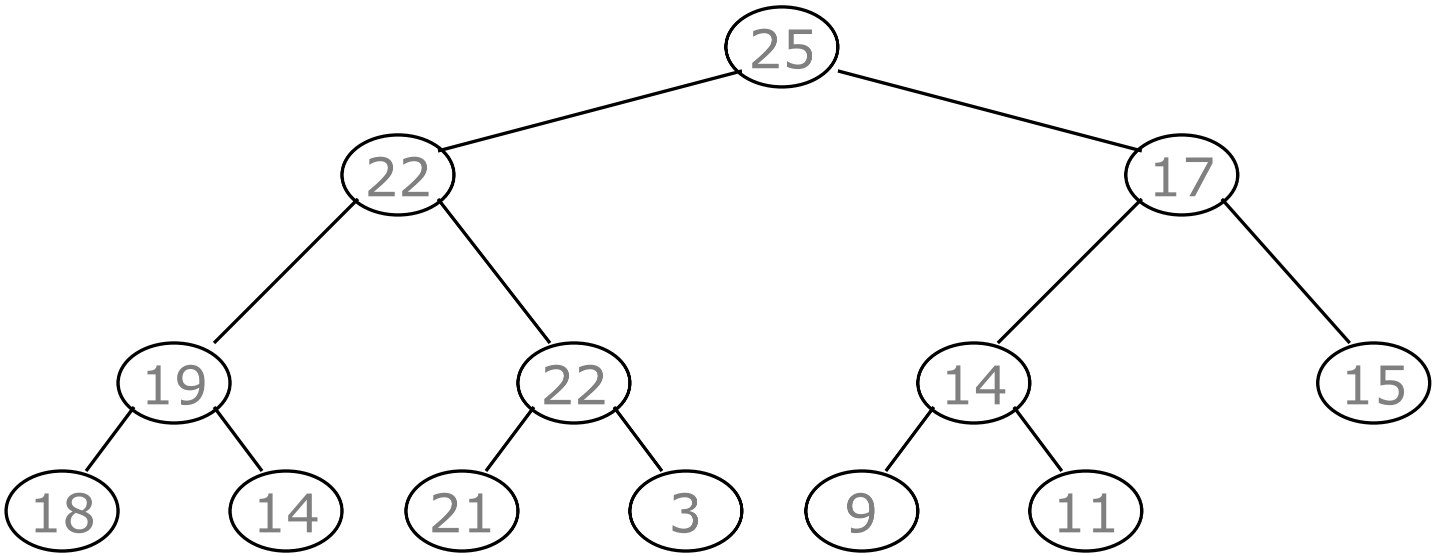


Opgave

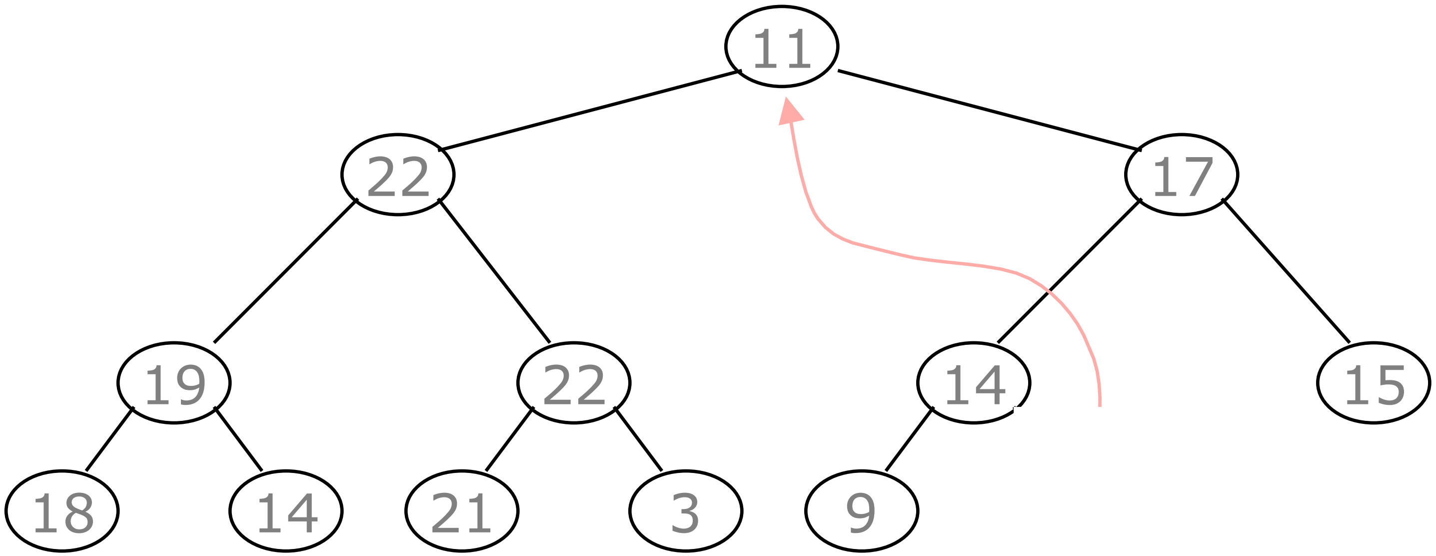
- Hobificer træet svarende til listen

44	99	42	71	2	64

Hobificeret betyder ikke sorteret



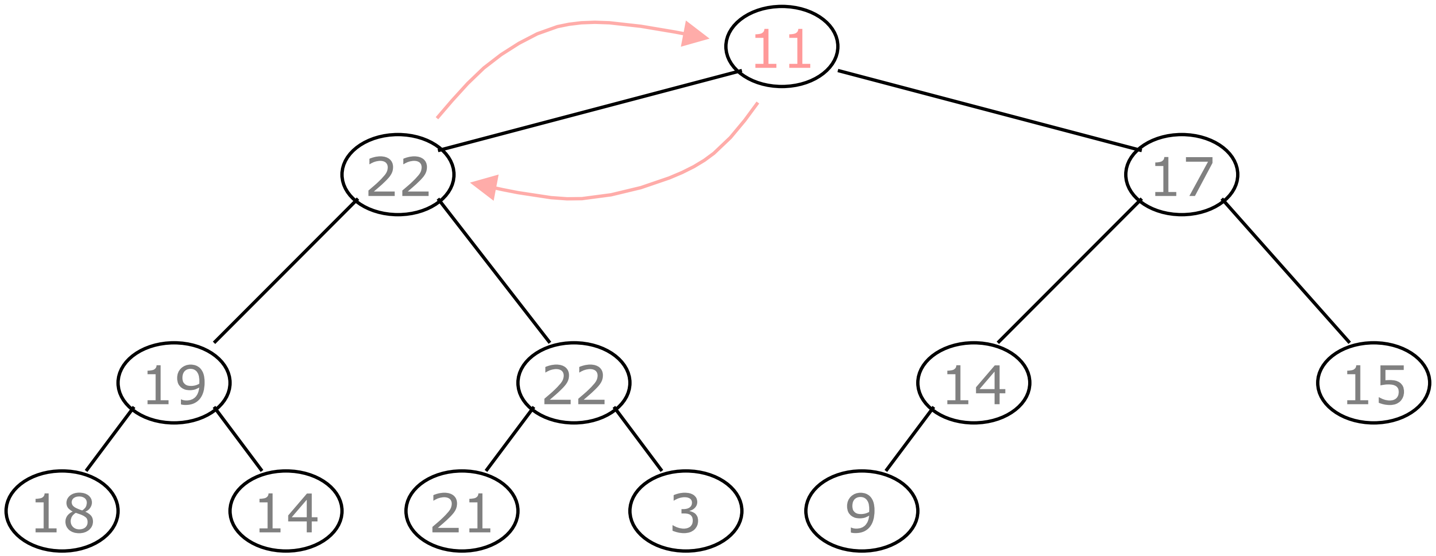
Største tal i roden

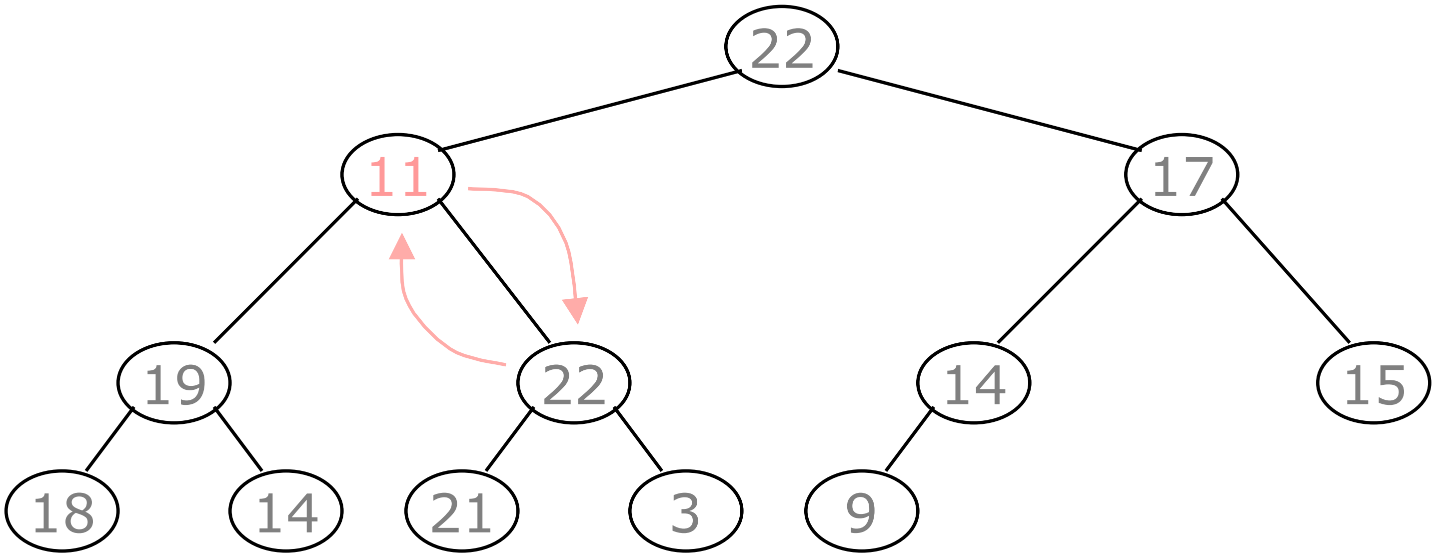


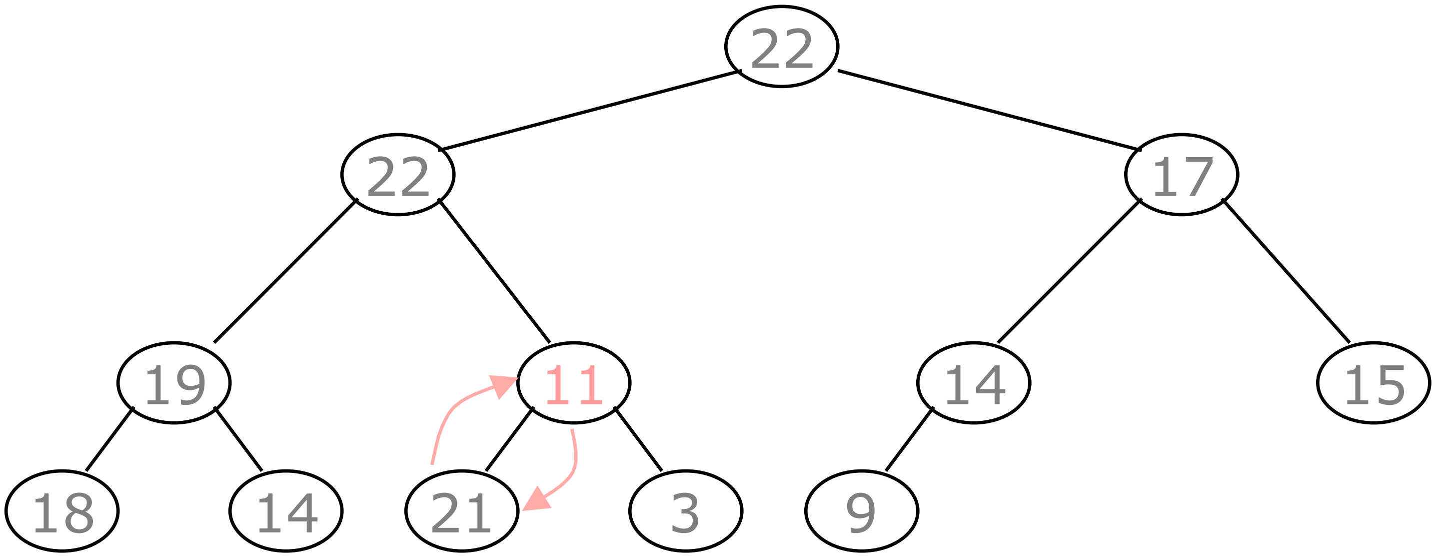
- Fjern det element mest til højre og dybest (højste index) og brug det som rod

Heapificering

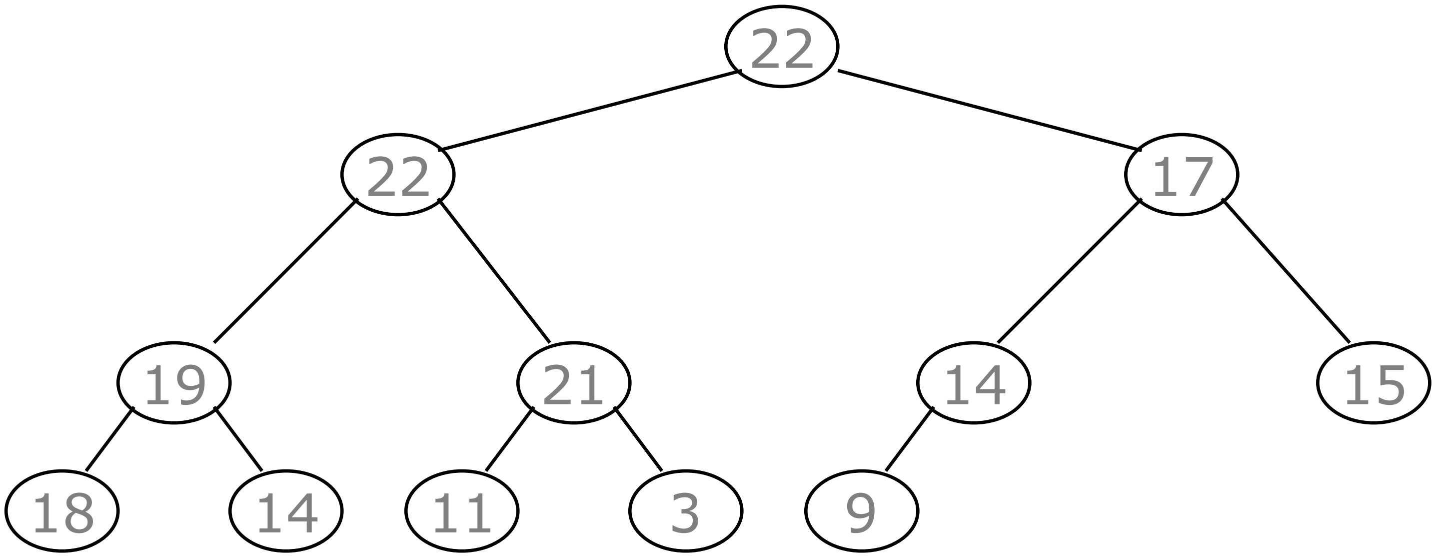
- Roden opfylder ikke længere hob egenskab




22
11
17
19
22
14
15
18
14
21
3
9


22
22
17
19
11
14
15
18
14
21
3
9

Fortsæt med at fjerne og hobificer indtil listen er tom



Hobsortering

- Skridt 1:
  - Hobificer arrayet fra 0 til n-1
- Skridt 2:
  - Udtag roden af hoben og erstat den med `arr[n-1]`
  - Dvs ombyt `arr[n-1] og arr[0]`
  - Hobificer arrayet fra 0 til n-2
  - Ombyt `arr[n-2] og arr[0]`
  - osv. indtil slut

Hvad sker der I liste repræsentationen

- Byt roden (første element) og det element der er mest til højre og dybest (sidste)

0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

- … sidste element ”eksisterer ikke mere”

Gentag hobificering

- Hobificering af roden (index 0, værdi11)...

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
22	22	17	19	21	14	15	18	14	11	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
9	22	17	19	22	14	15	18	14	21	3	22	25

- ...og igen.. Fjern og erstat roden or erstat med “sidste” element

Implementation

```
public void heapsort(int[] arr, int n) {
  for (int m=n/2; m >= 0; m--)
    heapify(arr, m, n-1);
                                                  /* pp1 */
  for (int m=n-1; m >= 1; m--)
    {                                             /* pp2 */
      swap(arr, 0, m);                            /* pp3 */
      heapify(arr, 0, m-1);                       /* pp4 */
    }                                             /* pp5 */
}
```

Implementation

```
private void heapify(int[] arr, int i, int k)
     // heapify node arr[i] in the tree arr[0..k]
{
  int j = 2 * i + 1;                                /* pp1 */
  if (j <= k)
    {
      if (j+1 <= k && arr[j] < arr[j+1])
        j++;                                        /* pp2 */
      if (arr[i] < arr[j])
        {
          swap(arr, i, j);                          /* pp3 */
          heapify(arr, j, k);                       /* pp4 */
        }
    }                                               /* pp5 */
}
```

Køretider

- Lad n være antallet af knuder i træet
- Hvis træet svarer til liste er det højst log(n) dybt
- Det kræver højst log(n) rekursive kald, at hobificere en knude (dybden af træet)
- Det kræver n*log(n) sammenligninger at hobificere et træ

Køretider for hobsortering

- Skridt 1 er en hobificering af et træ
- Skridt 1 kræver n*log(n) sammenligninger
- Skridt 2 indeholder en hobificering af en knude (roden)
- Denne kræver log(n) sammenligninger
- Skridt 2 gentages n gange
- Ialt er køretiden assymptotisk med n*log(n)

Opgave

- Kør hobsortering på listen

44	99	42	71	2	64

Sammenligning af sorteringsalgoritmer

- Køretider

	Værste fald	Gennemsnit
Udvalgssortering	$O(N^2)$	$O(N^2)$
Quicksort	$O(N^2)$	$O(n\log_2(N))$
Hobsortering	$O(n\log_2(N))$	$O(n\log_2(N))$

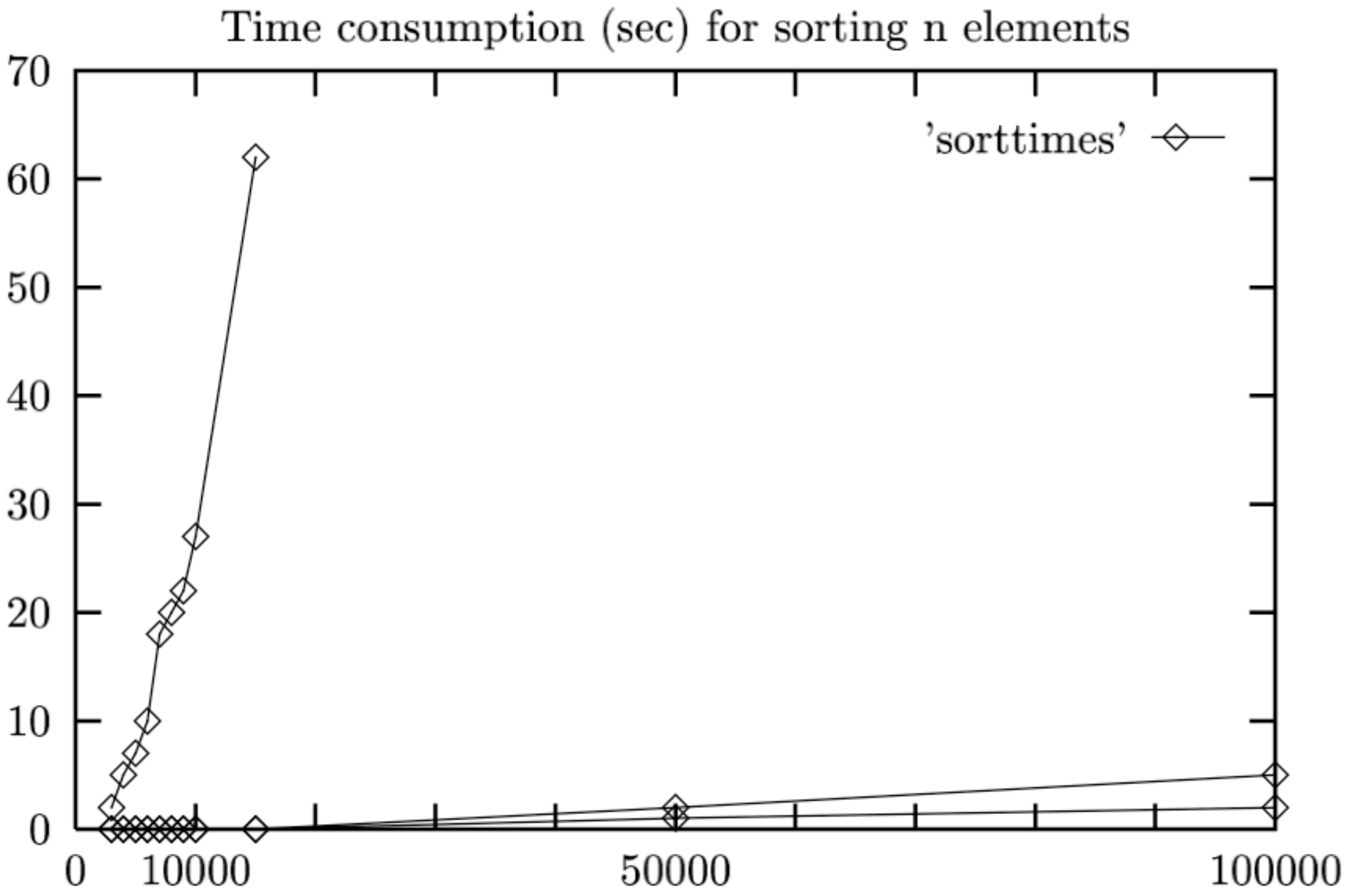
- I praksis er quicksort ofte hurtigere end hobsortering

- Hvad er vigtigst: Gennemsnitstiden eller værste falds tiden?

Eksperimentelle data

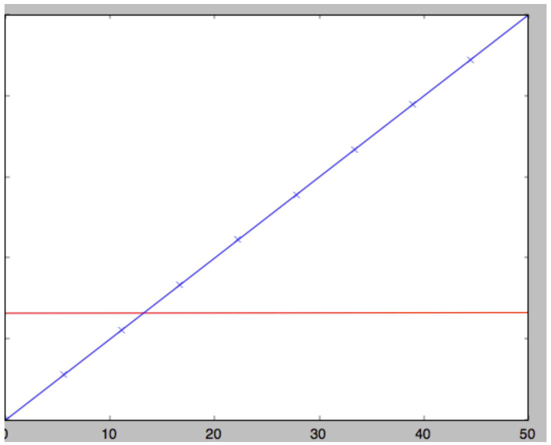
n	Selection sort	Quicksort	Heap Sort
3,000	2	0	0
4,000	5	0	0
5,000	7	0	0
6,000	10	0	0
7,000	18	0	0
8,000	20	0	0
9,000	22	0	0
10,000	27	0	0
15,000	62	0	0
50,000	766	1	2
100,000	3 995	2	5
500,000	(21 hours)	11	30
1,000,000	(111 hours)	27	66

Eksperimentelle data



Søgning og sortering

- Kan det betale sig at sortere før man søger?
- Hvor mange gange skal man søge i en liste før det kan betale sig at sortere først?



Opsummering

- Betragtede to almindelige og klassiske problemer i datalogi: Søgning og sortering
- Betragtede algoritmer (abstrakte løsninger) og deres implementation
- Algoritmerne kan analyseres uafhængigt af implementationerne
- Vi så at forskellige løsninger havde forskellige egenskaber mht. køretid
- Teoretisk analyse bør testes eksperimentelt

Næste forelæsning