

## Plan for dagen

- Opsamling på opgave fra sidst.
- ◦ Søgning og problem løsning
- ◦ Sortering
  - selection sortering
  - heap sortering
- Opgave til næste uge ]

## Top down programming

Eksempel fra sidst: "Læs et labyrint på den viiste form ind i et tegn array" original problem  
- niveau 0

### Overordnet algoritme

1. læs størrelse på labyrint
2. opret labyrint array
3. læs fra fil og put i array

#### niveau 1

Vi er stadig langt fra et program

ad 3.)

læs resterende linjer enkeltvis  
for hver linje  
→ flyt alle tegn fra den  
laeste linje over i  
den tilsvarende linje i  
array

#### niveau 2

Vi begynder at se  
tegn på kontrolstruktur  
(en løkke)

ad 2)

læs linjen  $N \times M$  fra filen  
split linjen i  $N$  og  $M$   
konverter  $N$  og  $M$  fra string  
til heltal

Man tager ikke altid  
1 for 2.

Jeg vælger oftest  

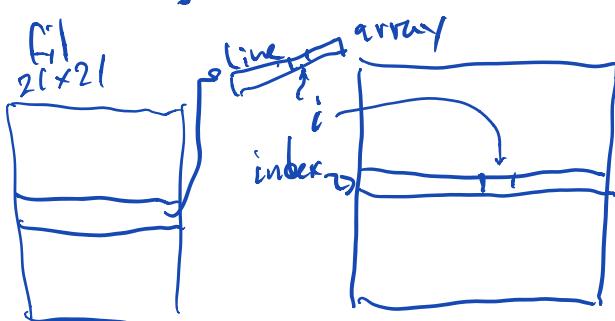
- den største klump
- eller den jeg er  
mest usikker på

ad 3)

```
while (fil not empty)
    Line = read line;
    for hvert tegn c i line {
        LabyrintL[i - index] = c;
    }
}
```

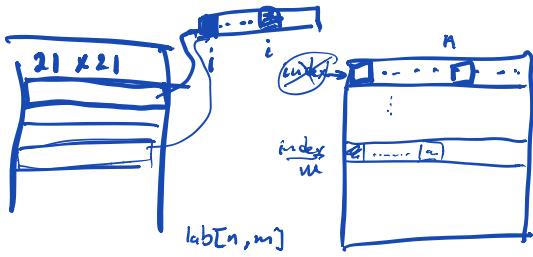
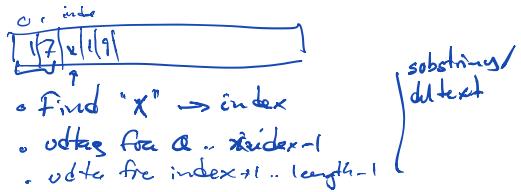
på et tidspunkt  
begynder der at blive  
bekov for variable  
Den indtører jeg  
sammen med tegninger

→ Generelt: Det er  
uforsvarligt at  
programme algoritmer  
uden kladde papir  
til tegninger

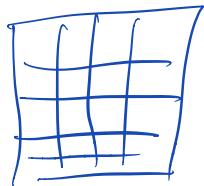


ad3)  
på dette tidspunkt tror jeg  
nok på ideen til at jeg  
gør over til at prøve den  
i visual studio.

Nogle af de overstigende  
noter henvær oppe som  
kommentarer



[ . 2 . 1 . . . . . . . . ]



## Om at lave løkker

Alle løkker har på overfladen denne struktur  
while ( betingelse ) {  
    "gør noget"  
}

Der er dog også nogle andre ting der  
altid er i en løkke:

nedtælling: efter hver gennemløb af løkken  
vil vi gerne være "tættere på"  
at være færdig"

opstart: vi skal have startet løkken  
sådan at alt arbejdet bliver  
gjort ( så vi ikke glemmer det  
første )

princippet/relationen: I en løkke er der ofte en  
celle variable der arbejdes  
sammen. Det kan f.eks være  
løkke index, array, sum-so-far,  
element vi leder efter osv.

I klassisk datalogi kaldes det  
"invarianten"  $\Rightarrow$  det der "ikke  
ændrer sig"

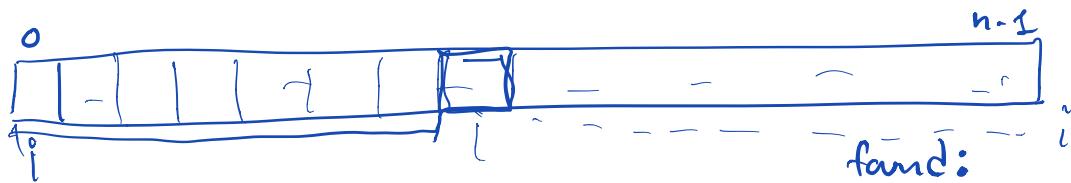
```

static 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 */
}

```

principle

- if `found = true`, then  $0 \leq i \leq n - 1$  and  $arr[i] = x$ ;
- if `found = false`, then  $x$  is not in  $arr[0..(n-1)]$ .

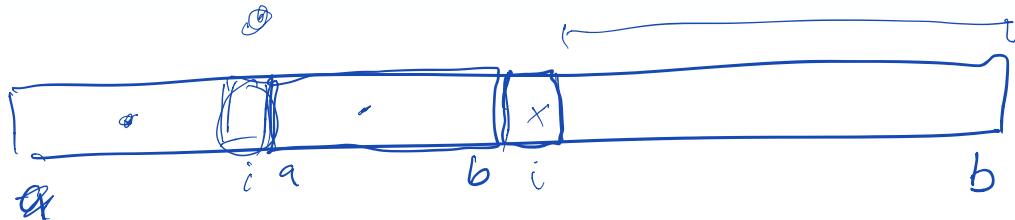


$i = \text{forst} \text{ om vi ikke har undersøgt}$

```

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

```



Hvis  $x$  findes i arra

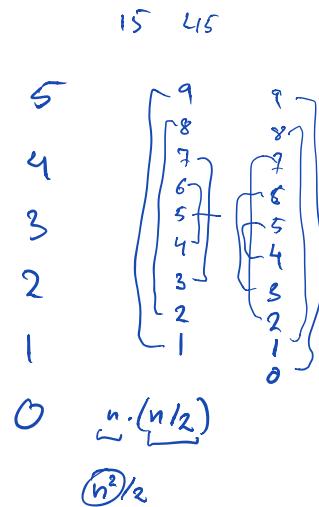
- sa er det mellem  $a$  og  $b$
- ellers så er det ikke mellem  $a$  og  $b$

1	2	3	4	5	6
2	1	3	2	4	7
4	2	5	3	6	8
8	3	7	4	8	9
16	4	8	5	9	10
32	5	10	6	11	20
		1000.000	700	1000.000.000	80

~

# Selection softening

0	1	2	3	4	5
35	62	28	50	11	45
11	62	28	50	35	45
11	28	62	50	35	45
11	28	35	62	50	45
11	28	35	45	62	50
11	28	35	45	50	62
11	28	35	45	50	62



Hvad er princippet?

Lin n frods

bin  $\log(n)$  got

sel  $n^2$  read, fuld

10	$\sim$	50
20	$\sim$	200
40	$\sim$	800
100	$\sim$	5000
1000	$\sim$	500,000
1000.000	$\sim$	5000 000 000 000

## Heap sort

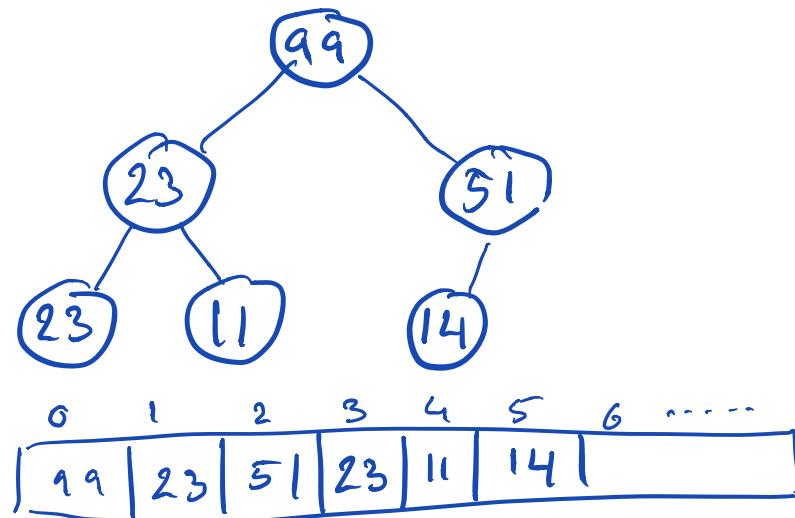
En bunke (heap) virker sådan at man kan

- sætte elementer ind
- tage største element ud (fjerne det)

Kan bruges til sortering ved at:

1. sætte alle elementer ind
2. tage elementerne ud

Fidusen er: Det kan gøres smart!



⑩

0	1	2	3	4	5	6	
10	27	15	9	12	57	19	.....

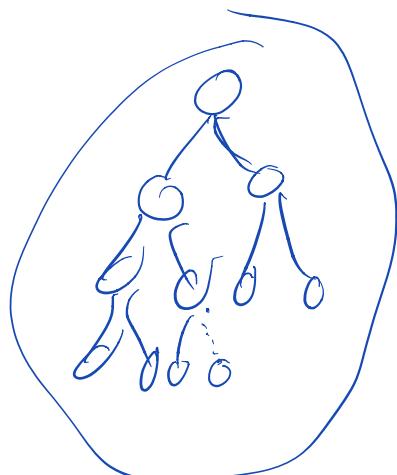
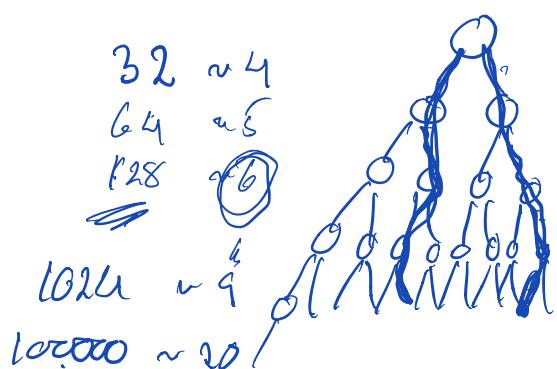
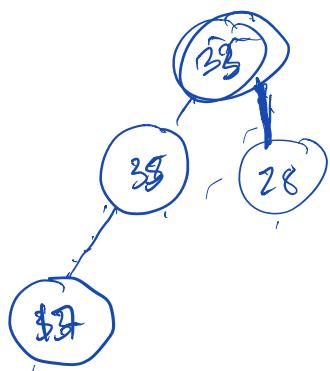
h

princip: elemente  $[0..h]$  or an heap

80  
58  
40  
35

— —

$$n \sim n + \log(n)$$



```

private static void heapify(int[] arr, int i) int k)
{
    int j = 2 * i + 1; venstre gren /* pp1 */
    if (j <= k) er i en indre knude
    {
        if (j+1 <= k && arr[j] < arr[j+1]) og den mindste /* pp2 */
        end venstre
        j++;
        if (arr[i] < arr[j]) er roden fjernet placeret /* pp3 */
        {
            swap(arr, i, j); /* pp4 */
            heapify(arr, j, k); /* pp5 */
        }
    } orden undertræ
}

```

