# AD - noter

Usama Bakhsh

April 7, 2024

## Contents

# 1 Intro, beviser, løkkeinvarianter

Figure 1: Insertion sort psedocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

For loop invariants:

- Initialization: It is true prior to the ûrst iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: The loop terminates, and when it terminates, the invariant4usually along with the reason that the loop terminated4gives us a useful property that helps show that the algorithm is correct.

Bevis typer:

- Direkte bevis: Man beviser en implikation $(A \Rightarrow B)$ ved at antage at hypotesen A er sand og derefter vise at konklusionen B er sand.
- Kontrapositivt bevis: I stedet for et direkte bevist med $(A \Rightarrow B)$ så siger vi $\neg B \Rightarrow \neg A$
- Modstridsbevis: Vis en sætning $S$ ved at se om $\neg S$ er umulig
- Induktionsbevis: Bruges til formler hvor sætning $S$ indekseres for ikke-negativ heltal det vil sige $n \in N$. Gøres i trin
  - Basis: $S(0)$

  - Induktion: Vis at $S(n) \Rightarrow S(n+1)$, for ethvert $n \in N$

- Bevis via invariant:
  - Invariant: En påstand der altid gælder (på specifikke steder) i en algoritme
  - Termineringsbetingelse: En påstand der gælder når en algoritme terminerer

# 2 Tidskompleksitet og asymptotisk notation

Whatever du ved det godt. Repetition wallah

# 3 Del-og-hersk, rekursionsligninger

Recall that for divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

**Divide** the problem into one or more subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

**Combine** the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion ***bottoms out*** when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

Figure 2: Del og hersk handler i bund grund om rekusion. Husk bottum ups

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. ■

Figure 3: For at løse rekusionsligning bruger vi master theorom. Der er tre cases for master theorom.
Hvis den fundet køretid er større end $f(n)$, så er det case 1. Hvis de er lige med hinanden, så er det case 2. $f(n) >$ den fundet køretid, så case 3.

Der kan bruge substitionsmetoden i 4.3 i CLRS. Jeg forstår den ikke, men du kan altid lade algoritmen gå fra k til n og se hvilken køretid du får.

# 4 Nedre grænser; Amortiseret analyse

Amortiseret analyse kan deles op i to metoder og en analyse.

## Aggregate analyse

Dette handler om at analysere værste omkostninger ved en eller flere operationer. Tænk på MULTIPOP, POP og PUSH i stack, eller den binær tæller. Kig på slides med binær tæller eller i 16.1 i CLRS. Vi finder the upper bound af hver n operationer og giver den det samme kostværdi. Selv forskelig operationer har den samme gennemsnits værdi.

## Regnskabsmetoden/The accounting method

Denne metode handler om at finde worst-case omkostninger ved at holde regnsskaber. Vi lægger "penge" til side og knytter vi dem til elementer i datastrukturen. I regnskab In the accounting method we charge early operations more than their cost. That way we charge less in later opeartions. The amount we charge each opeartions, is its amortized cost, and we save the extra charge as "prepaid credit". If we undercharge early on, the credit becomes negative, and the total amortized cost incurred at that time will not be an upper bound.We must therefore not undercharge in the beginning.

## Potentialmetoden/The potential method

# 5 LSM træer, Fibonaccihobe

Fibonaccihobe er en mergeable-heap. Det vil sige at den kan gøre disse fem operationer. Fordel ved Fib-hobe er at deres amortiseret værdi er meget lav, så den er god at bruge, når man skal lave mange instruktioner.

- MAKE-HEAP() creates and returns a new heap containing no elements
- INSERT(H,x) inserts element x, whose key has already been filled in, into heap H
- MINIMUM(H) returns a pointer to the element in heap H whose key is minimum
- EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element
- UNION($H_1, H_2$) creates and returns a new heap that contains all the elements of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation

Udover dette har Fib-hobe også disse to operationer.

- DECREASE-KEY(H,x,K) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value
- DELETE(H,x) deletes element x from heap H

## 5.1 struktur af fib hobe

Fib hobe er et binært søge træer med ekstra propeties. Den følger min-heap-propety hvilket betyder at "the key of a node is greater than or equal to the key of its parent" - CLRS. Efter Root-listen stiger noderne kun i værdi jo længere ned du går i træet.
Hver node x har pointer til dens forældre x.p, en pointer til barnet og til sin egen række eller sig selv hvis den er alene. Dette kaldes Circular, doubly linked lists.
Sidste to struktur er en numerisk værdi x.degree der fortæller hvor mange børn en knude har. Det er kan være mange børn. "The boolean-valued attribute x:mark indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE-KEY operation in Section 19.3, we will just set all mark attributes to FALSE." - CLRS.

| Procedure | Binary heap (worst-case) | Fibonacci heap (amortized) |
|---|---|---|
| MAKE-HEAP | $\Theta(1)$ | $\Theta(1)$ |
| INSERT | $\Theta(\lg n)$ | $\Theta(1)$ |
| MINIMUM | $\Theta(1)$ | $\Theta(1)$ |
| EXTRACT-MIN | $\Theta(\lg n)$ | $O(\lg n)$ |
| UNION | $\Theta(n)$ | $\Theta(1)$ |
| DECREASE-KEY | $\Theta(\lg n)$ | $\Theta(1)$ |
| DELETE | $\Theta(\lg n)$ | $O(\lg n)$ |

**Figure 19.1** Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by $n$.

Figure 4: Figure over amortiseret værdi af Fib-hobe, sammenlignet med Binærhobes tidskompleksitet

## 5.2 Fib hobe operation

**MAKE-FIB-HEAP()** laver en tom hob uden nogle træer.
**FIB-HEAP-INSERT(H,x)** Linje 1-4 initialiser nogle strukturalle attributes

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

Figure 5: Pseudo kode til FIB-HEAP-INSERT(H,x)

til knude x. If statement på linje 5 tjekker om hobbene er tom. Hvis den er så er linje 6-7 til får at til at lave en root liste kun med x. Ellers vil linje 8 insætte x ind i root liste. Linje 9 checker om x er den mindste værdi, hvis den er så skal skal hobbens minimum værdi sættes til at være lig med x. H.n er antallet

af knuder som opdateres med +1.
**FIB-HEAP-UNION(H1,H2)** Sammensætter to hobber til en ny hob. De gamle hobber bliver destrueret. Linje 1-3 sammensææte root listen til en ny hob. Linje 2,4 og 5 finder den mindste værdi. Linje 6 ændre antallet af knuder +1

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5       H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Figure 6: Pseudo kode til FIB-HEAP-UNION(H1,H2)

**FIB-EXTRACT-MIN(H)**:

1. **Find Minimum**: Første trin er at finde noden, der indeholder det mindste nøgleværdi i heapen. Denne operation tager $O(1)$ tid i en Fibonacci-heap, fordi det mindste element altid er gemt på rodniveauet af en af træerne i skovstrukturen, som udgør heapen.

2. **Fjern Minimumnode**: Når minimumnoden er fundet, fjernes den fra heapen. Dette indebærer at skære minimumnoden fra dens forælder og fusionere dens børn med rodlisten. Børnene til minimumnoden bliver rødder til nye træer i skoven. Dette trin kan tage $O(\log n)$ tid, hvor $n$ er antallet af noder i heapen.

3. **Konsolider Træer**: Efter fjernelse af minimumnoden kan heapen muligvis overtræde egenskaberne for en Fibonacci-heap. Specifikt kan der være træer med samme grad (antal børn). For at opretholde heapens egenskaber skal disse træer fusioneres. Dette gøres ved at linke træer med samme grad, indtil der ikke er to træer i heapen med samme grad. Dette trin tager $O(\log n)$ tid.

4. **Opdater Minimumpegeren**: Efter konsolidering skal minimumpegeren i heapen opdateres til at pege på det nye mindste element. Denne operation tager $O(1)$ tid, fordi det mindste element er blandt rødderne af træerne i heapen.

For CONSOLIDATE(H) skal vi lidt i dybten.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

Figure 7: Pseudo kode til FIB-EXTRACT-MIN(H)

1. **Allokering og Initialisering**: Proceduren begynder med at oprette og initialisere et array, hvor hver indgang bliver nulstillet.

2. **For-løkke for Rodliste**: Hver rod i rodlisten bliver behandlet. Hvis to rødder har samme grad, bliver de linket sammen, og graden af den ene øges. Dette gentages, indtil ingen andre rødder har samme grad som den aktuelle rod.

3. **While-løkke for Linkning af Rødder**: Denne løkke linker gentagne gange roden af det træ, der indeholder den behandlede node, med et andet træ, hvis rod har samme grad. Dette fortsætter, indtil ingen andre rødder har samme grad som den aktuelle rod.

4. **Afslutning og Genopbygning af Rodlisten**: Når alle rødder er behandlet, tømmes rodlisten, og arrayet bruges til at genopbygge den. Resultatet er en Fibonacci-heap med den ønskede struktur.

5. **Opsummering af Resultat**: Heaps bliver opdateret og manipuleret, indtil den ønskede struktur er opnået, og den minimale node er udtrukket.

Denne proces med konsolidering sikrer, at Fibonacci-heapen opretholder sine invarianter og struktur, og at operationen for udtrækning af minimum kan udføres effektivt.

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]          // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]

FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

Figure 8: Pseudo kode til CONSOLIDATE(H)+FIB-HEAP-LINK(H,x,y)

### 5.3 Decreasing a key and deleting a node

1. **FIB-HEAP-DECREASE-KEY Proceduren**:

   (a) Linjer 1-3 sikrer, at den nye nøgle ikke er større end den nuværende nøgle for noden x, og tildeler derefter den nye nøgle til x. Hvis x er en rod, eller hvis x:nøgle er mindre end y:nøgle, hvor y er x's forælder, behøves ingen strukturelle ændringer, da min-heap orden

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3      if y.mark == FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```

Figure 9: Pseudo kode til DECREASE-KEY

ikke er blevet overtrådt. Linjer 4-5 tester for denne betingelse.

(b) Hvis min-heap orden er blevet overtrådt, kan mange ændringer forekomme. Vi starter med at skære x fra sin forælder i linje 6. Skæreproceduren "skærer" forbindelsen mellem x og dens forælder y, hvilket gør x til en rod. Vi bruger markattributterne til at opnå de ønskede tidsgrænser. De registrerer en lille del af historien for hver node.

(c) Vi er ikke færdige endnu, fordi x måske er det andet barn, der er blevet skåret fra sin forælder y, siden y blev linket til en anden node. Derfor forsøger linje 7 af FIB-HEAP-DECREASE-KEY at udføre en kaskadeskæringsoperation på y.

(d) Når alle kaskadeskæringer er foretaget, opdaterer linjer 8-9 af FIB-HEAP-DECREASE-KEY afslutningsvis H:min om nødvendigt. Den

eneste node, hvis nøgle ændrede sig, var noden x, hvis nøgle blev mindre. Således er den nye minimale node enten den oprindelige minimale node eller node x.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, −∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

Figure 10: Pseudo kode til FIB-HEAP-DELETE

# 6 Dynamisk programmering

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

## 6.1 Rod-cutting

### 6.1.1 Den simple rekursive

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max{q, p[i] + CUT-ROD(p, n - i)}
6  return q
```

Figure 11: Pseudo kode til den simple rekursive Cut-Rod naive approuch. p er et array over priser og n er længden af staven

Den naive approuch har en ekspotentiel køretid. For hver n+1 fordobles køretiden. Vi regner med at rekusionstræet har $2^{n-1}$ blade.

**Problemformulering:**

Givet en stav af længde $n$ tommer og en tabel med priser $P_i$ for $i$ tommer (hvor $i = 1, 2, \ldots, n$), skal du finde den maksimale indtægt $R_n$, der kan opnås ved at skære staven i stykker og sælge dem.

**Eksempel:**

Lad os antage, at du har en stav af længde 4 tommer, og priserne for hver tomme af staven er som følger:

$$P_1 = \$2$$
$$P_2 = \$5$$
$$P_3 = \$7$$
$$P_4 = \$8$$

For at maksimere indtægten skal du finde ud af, hvordan du skal skære staven. For eksempel vil det at skære den i to stykker af længde 1 og et stykke af længde 2 give en samlet indtægt på $\$2 + \$2 + \$5 = \$9$.

**Løsningsmetode:**

Vi kan løse dette problem ved hjælp af dynamisk programmering. Følgende trin kan anvendes:

1. Identificer delproblemerne.
2. Formuler rekurrensrelationen.
3. Anvend memoization eller tabulering (bottom-up approuch) til at beregne løsningerne til delproblemerne.
4. Returnér løsningen.

Rekurrensrelationen for rodskæringsproblemet er:

$$R_i = \max_{1 \leq j \leq i}(P_j + R_{i-j})$$

hvor $R_i$ er den maksimale indtægt for en stav af længde $i$ tommer.

## 6.2 Memoized eller bottom-up

Begge tilgange har køretiden $\Theta(n^2)$.
Der er normalt to ækvivalente måder at implementere en dynamisk programmeringsmetode på. Løsninger til rodskæringsproblemet illustrerer begge metoder. Den første tilgang er top-down med memoization. I denne tilgang skriver du proceduren rekursivt på en naturlig måde, men ændret til at gemme resultatet af hvert delproblem (normalt i en array eller en hash tabel). Proceduren kontrollerer nu først, om den tidligere har løst dette delproblem. Hvis det er tilfældet, returnerer den den gemte værdi og sparer dermed yderligere beregninger på dette niveau. Hvis ikke, beregner proceduren værdien på sædvanlig vis, men gemmer den også. Vi siger, at den rekursive procedure er blevet memoized: den husker, hvilke resultater den tidligere har beregnet.
Den anden tilgang er metoden nedefra og op. Denne tilgang afhænger typisk af en naturlig opfattelse af størrelsen på et delproblem, således at løsningen af et bestemt delproblem kun afhænger af løsningerne af mindre delproblemer. Løs delproblemerne i størrelsesorden, med de mindste først, og gem løsningen på hvert delproblem, når det først er løst. På denne måde, når du løser et bestemt delproblem, er der allerede gemte løsninger for alle de mindre delproblemer, som dets løsning afhænger af. Du behøver kun at løse hvert delproblem én gang, og når du først ser det, har du allerede løst alle dets forudsætningsmæssige delproblemer.
Disse to tilgange giver algoritmer med samme asymptotiske køretid, bortset fra under usædvanlige omstændigheder, hvor top-down tilgangen faktisk ikke rekursivt undersøger alle mulige delproblemer. Nedefra-og-op-tilgangen har ofte meget bedre konstantfaktorer, da den har lavere overhead for procedurekald.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0                        // already have a solution for length n?
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n                 // i is the position of the first cut
7          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
8  r[n] = q                           // remember the solution value for length n
9  return q

BOTTOM-UP-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  r[0] = 0
3  for j = 1 to n                     // for increasing rod length j
4      q = -∞
5      for i = 1 to j                 // i is the position of the first cut
6          q = max{q, p[i] + r[j - i]}
7      r[j] = q                       // remember the solution value for length j
8  return r[n]
```

Figure 12: Pseudo kode til to tilgange til DP

### 6.3 LCS

## 7 Grådige algoritmer

Held og lykke I guess

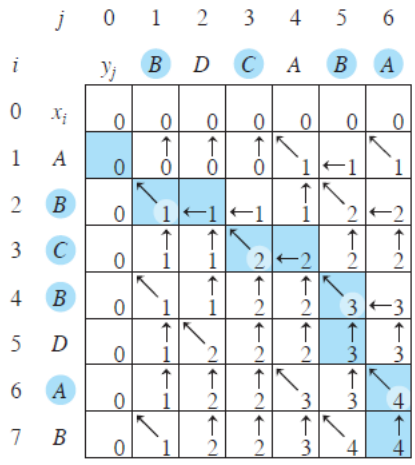

**Figure 14.8** The $c$ and $b$ tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row $i$ and column $j$ contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of $X$ and $Y$. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each "↖" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Figure 13: Billede til hvordan LCS ser ud.

### 7.1 An activity-selection problem

### 7.2 Knapsack problem

### 7.3 Huffman code

## 8 Binære søgetræer

### 8.1 Binærsøge træer

Binærsøge træer er sorteret således at alt til venstre for roden er mindre end roden.
Alt til højre for roden er større end roden.
Det kun roden, hvis forældre er NIL.
$\Theta(n)$ er køretiden for at komme igennem træet.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Figure 14: **INORDER-TREE-WALK(x)** printer alle værdierne i et træ i rækkefølge fra mindst til størst.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

Figure 15: TREE-SEARCH & ITERATIVE-TREE-SEARCH. x er pointer til et subtree og k er nøglen. For at søge gennem hele træet kan man sige x=T.root

### 8.2 Querying a binary search tree

### 8.3 Insertion and deletion

## 9 Disjunkte mængder

## 10 Mindste udspændende træ

## 11 Korteste veje

## 12 Beregningsgeometri

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x

TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Figure 16: Min-Max. Rimelig intuitiv i forhold til binærsøge træers propeties

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)   // leftmost node in right subtree
3  else // find the lowest ancestor of x whose left child is an ancestor of x
4      y = x.p
5      while y ≠ NIL and x == y.right
6          x = y
7          y = y.p
8      return y
```

Figure 17: Tree-succesor definers som det næste knude som findes ved inorder-walk

```
TREE-INSERT(T, z)
 1   x = T.root                  // node being compared with z
 2   y = NIL                     // y will be parent of z
 3   while x ≠ NIL               // descend until reaching a leaf
 4       y = x
 5       if z.key < x.key
 6           x = x.left
 7       else x = x.right
 8   z.p = y                     // found the location—insert z with parent y
 9   if y == NIL
10       T.root = z              // tree T was empty
11   elseif z.key < y.key
12       y.left = z
13   else y.right = z
```

Figure 18: Foregår som TREE-SEARCH. y er et trailing point og z's forældre. z er knuden som skal indsættes i træet. Som mange andre binæsøgetræers algoritmer er køretiden O(h), hvor h er højden af træet. z bliver indsat i bunden af træet / leaf