# Rekursion

| Keywords | Notes |
|---|---|
| Rekursiv algoritme | En algoritme der kalder sig selv igen<br>Divide and conquer<br>Divide on the way down and conquer on the way back up |
| En god rekursiv funktion indeholder: | 1. base case - En base case er en funktionskald der kan løses uden rekursivitet. Som regelt en if statement der tjekker om en variabel er nået 0.<br>2. Arbejd mod base case - En værdi der bliver decrementeret. Denne variabel skal også være input til funktionen<br>3. En *return*<br><br>Eksempler på rekursiv funktion:<br><br>*int f(int x) {*<br>*  if (x == 0)*<br>*  return 0;*<br>*  else*<br>*  return 2*f(x-1)+x*x;*<br>*}*<br>*int sumNaturalNumbers(int n) {*<br>*  if (n<1)*<br>*  return 0;*<br>*  return n + sumNatrualNumbers(n-1);*<br>*}* |

# Tidskompleksitet

| Keywords | Notes |
|---|---|
| N | Antal elementer i funktionskald input |
| Tidskompleksitet | Må ikke indeholde konstanter, kun N og funktioner af N |
| Nestede loops | De forskellige tidskompleksitet bliver ganget sammen |

| Keywords | Notes |
|---|---|
| | **Eksempel:**<br><br>*for(int i = 0; i > N; i++) {*<br>*for(int j = 0; 1 > N; j++) {*<br>*j *= 2*<br>*}*<br>*}*<br><br>Vil have følgende tidskompleksitet for inderste og yderste<br>- Inderste: $O(\log(N))$<br>- Yderste: $O(N)$<br>- Samlet: $O(N*\log(N))$<br><br>Flere loops, så ganges de individuelle loops sammen |
| For loops i sekvens | Man ser på den der har den største tidskompleksitet<br><br>**Eksempel:**<br>*hej = 0*<br>*for(int i = 0; i > N; i++) {*<br>*for(int j = 0; 1 > N; j *= 2) {*<br>*hej += 1*<br>*}*<br>*}*<br>*for(int i = 0; i > N; i++) {*<br>*hej += 1*<br>*}*<br><br>Da det øverste for loop ser man bort fra det andet og den samlede tidskompleksitet bliver:<br>- Samlet: $O(N*\log(N))$ |
| Forskellige tidskompleksiteter | Best case - **Ω**(N)<br>"Average case" - Θ (N) er 'best of all worst cases'<br>Worst case - O(N) |
| T(N) | Udregningstiden for en funktion med N elementer |
| | |

| Hvad der sker | Hvad går i fra og til | Tommelfingerregel | Hvad bliver tidskompleksiteten (O) | Eksempel | Svar |
|---|---|---|---|---|---|
| Dividere med k for hver iteration af i | Fra N til 1 | Halvering af problem | $O(log_k(N))$ | for (int i = N; i <1; i =i/4) | $O(log_4(N))$ eller $O(\log(N))$ |
| Dobler med k for hver iteration af i | Fra 1 til N | Halvering af problem | $O(log_k(N))$ | for (int i = 0; i > N; i =i*4) | $O(log_4(N))$ eller $O(\log(N))$ |
| Simpelt for loop | Fra 1 til N eller omvendt | For loop | $O(N)$ | for (int i = 0; i >= N; i++) | $O(N)$ |

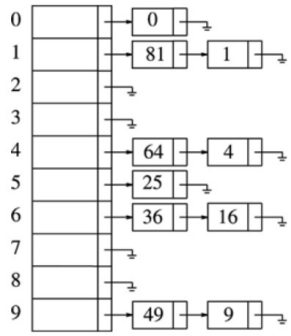
| | | | | | |
|---|---|---|---|---|---|
| To for loops i sekvens | Begge fra 1 til N eller omvendt | For loop | $O(N)$ | for (int i = 0; i >= N; i++) | $O(N)$ |
| To nestede for loops | Begge fra 1 til N | For loops | $O(N^2)$ | | $O(N^2)$ |
| Fibonacci sekvens | N'te fibonacci tal | | $O(2^N)$ | | |
| | | | | | |

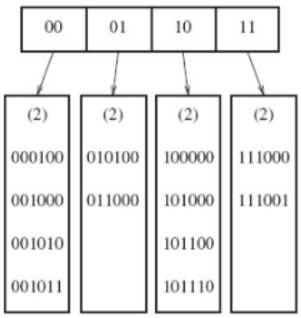
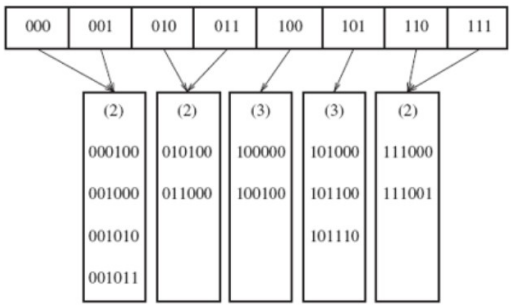
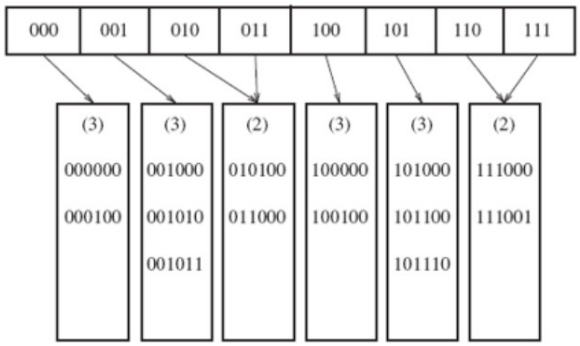
## Simple datastrukturer/datatyper

| **Keywords** | **Notes** |
|---|---|
| Stack | Man kan bruge push og pop på stacken, FILO (First in last out)<br><br>**Push:**<br>- Indsætter element på toppen af stacken<br>**Pop:**<br>- Fjerner øverste element fra stacken |
| Queue | Man kan bruge enqueue og dequeue på en queue, FIFO (First in first out)<br>Kan være cirkulær buffer<br><br>**Enqueue:**<br>- Put element i Queue<br>**Dequeue:**<br>- Fjerne første element i queue |
| List | Linked list: hvert element i listen linker til det næste element |

## Hashing

| **Keywords** | **Notes** |
|---|---|
| Hvad er hashing | Hashing er en abstrakt datatype, hvor objekter kan henetes og gemmes i konstant tid - O(1) - eller næsten.<br><br>Der kræves, at hvert objekt har en entydig nøgle, hvilket kan skabes igennem en hashing funktion.<br><br>Normalt er objekterne lagret i et array hvis størrelse er et primtal |
| Loadfaktor | Betegnes som $\lambda$ og er lig med antallet af elementer i tabellen divideret med tabellens størrelse og er derfor altid mindre end 1.<br><br>Loadfaktoren bør i de fleste løsninger ved mindre end 0.5 |
| Tradeoff | Ved at bruge hashing opnår man en høj hastighed på bekostning af plads, da man skal bruge cirka dobbelt så meget plads som dataen fylder for at opnå konstant tid ved insættelse |
| Hash funktion | En simpel og effektiv hash funktion: (Java)<br><br>`Public static int hash( string key, int tableSize )`<br>`{`<br>`  int hashVal = 0;` |

| Keywords | Notes |
|---|---|
| | ```<br>for( int i = 0; i < key.length( ); i++ )<br>hashVal = 37 * hashVal + key.charAt( i );<br><br>hashVal %= tableSize;<br><br>if ( hashVal < 0 )<br>hashVal += tableSize;<br><br>Return hashVal;<br>}<br>``` |
| Kollision | Uanset hvor god en hashfunktion man vælger, så vil det forekomme, at to eller flere objekter hasher til det samme indeks i tabellen. Dette kaldes en kollision.<br><br>Kollisioner kan løses på forskellige måder:<br>- Separate chaining<br>- Linear probing<br>- Quadratic probing<br>- Double hashing |
| Seperate chaining | Hash funktion: % 10<br>  <br> |
| Linear probing | Når en kollision konstateres, placeres objektet på den næste ledige plads.<br>Problemet med denne løsning er at der vil opstå klyngedannelse (primary clustering)<br>Dette kan medføre uacceptable læse- og skrive-tider afhængig af loadfaktoren $\lambda$ |
| Quadratic probing | Når en kollision konstateres, prøver man i stedet næste kvadrattal.<br>Første kollision: +1^2<br>Anden kollission: +2^2<br>Tredje kollision: +3^2<br>Osv.<br><br>Hvis TableSize er et primtal større end 3 og loadfaktoren er mindre end 0.5, så kan der altid findes en ledig plads. |
| Double hashing | To hash funktioner.<br>Den første beregner indeks<br>Den anden beregner beregner step size ved kollision<br><br>En af de bedste slags probing.<br>Stopper clustering |
| Rehashing | Når $\lambda$ bliver for stor, skal tabellen udvides.<br>Dette kaldes rehashing |

| Keywords | Notes |
| --- | --- |
| | Det kan gøre når:<br>- $\lambda >= 0.5$<br>- Når en indsættelse fejler<br>Tabelstørrelsen fordobles eller<br>Til næste primtal (23 -> 47)<br>En O(n) operation |
| Extendible hashing | Pointers til buckets<br>  <br><br>Addressen udvides ved indsættelse af 100100:<br>  <br><br>000000 indsættes. Pointers 000 og 001 får hver deres bucket<br>  <br> |
| State-of-the-art hashing | Cuckoo-hashing<br>Hopscotch hashing |
| Cuckoo-hashing | The power of two choices<br>To tabeller; To tabeller; To hash-funktioner (måske)<br>Hvis pladsen er optaget, så smid beboeren ud.<br>Egenskaber:<br>- Risiki for cykler; dog meget lille, hvis $\lambda < 0.5$<br>- Det duer ikke hvis $\lambda > 0.5$ |
| Hopscotch-hashing | En forbedring af linear probing<br>I visse moderne arkitekturer er nærhed (proximity) vigtigere end antallet af forsøg<br>Stort set ikke behov for rehashing hvis $\lambda < 0.5$.<br>Er Cuckoo-hashing overlegen på de fleste parametre |

| Keywords | Notes |
|---|---|

| | Item | Hop |
|---|---|---|
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1000 |
| 10 | E | 0000 |
| 11 | G | 1000 |
| 12 | F | 1000 |
| 13 | | 0000 |
| 14 | | 0000 |
| ... | | |

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11

Hopscotch hashing table. Hop kolonnen fortæller hvilke positioner blandt dens eget og de næste tre indekser der er optaget af denne hash værdi.

F.eks. Ved indeks 8 er hop 0010. Det betyder at kun position 10 indeholder en værdi der hasher til indeks 8

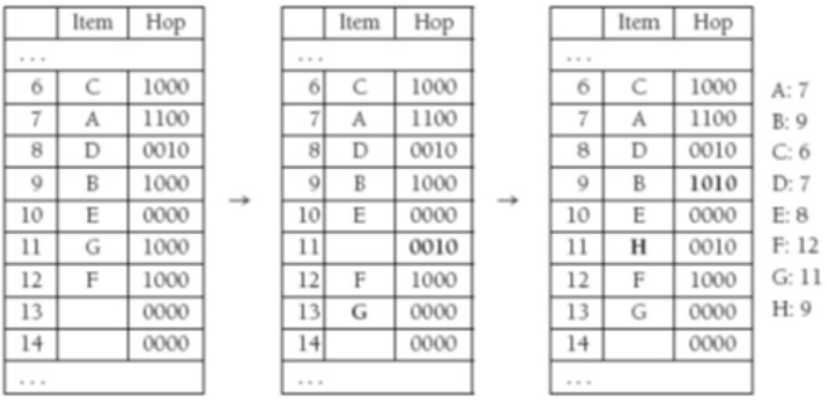
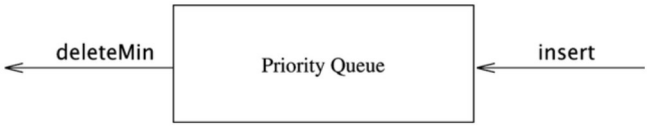
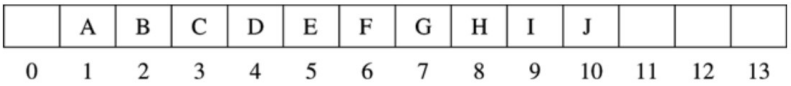
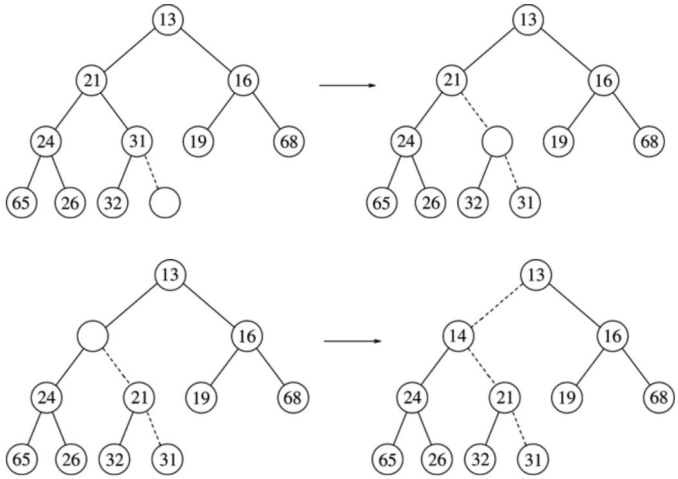
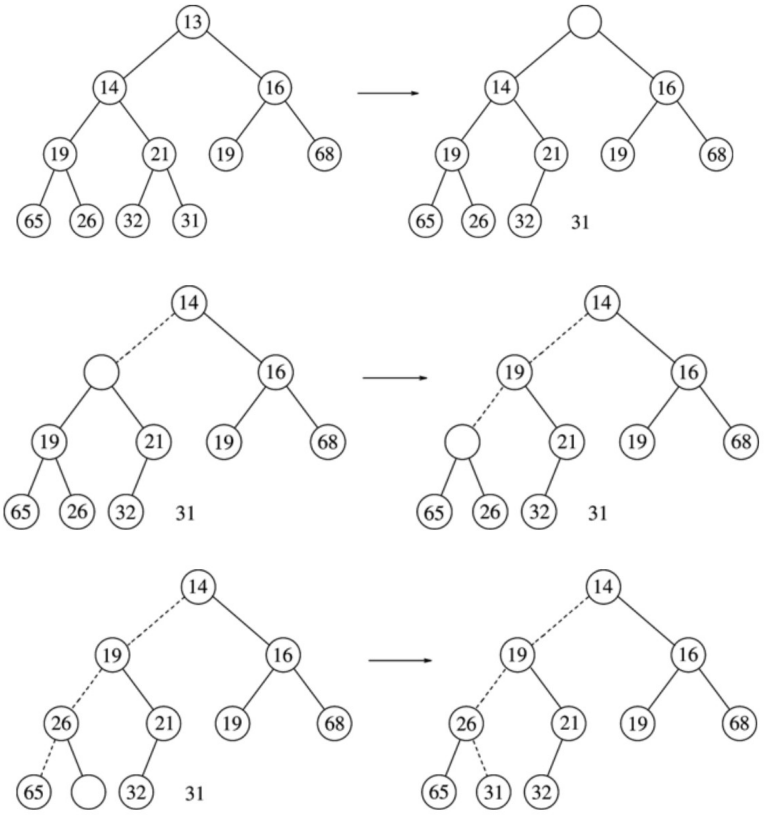



**Figure 5.47** Hopscotch hashing table. Attempting to insert *H*. Linear probing suggests location 13, but that is too far, so we evict *G* from position 11 to find a closer position

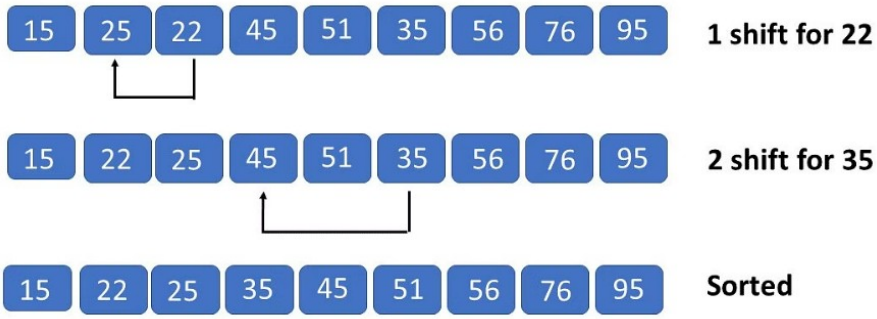
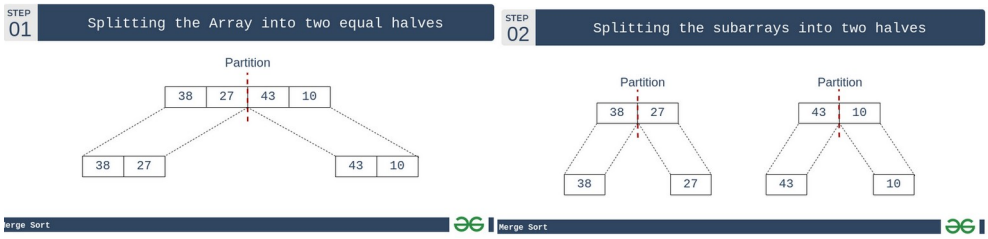
Man bliver nød til at flytte elementer hvis der ikke er plads i de næste tre indekser.

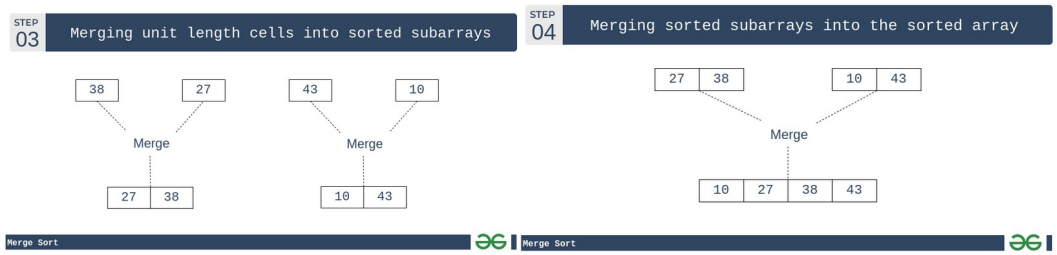
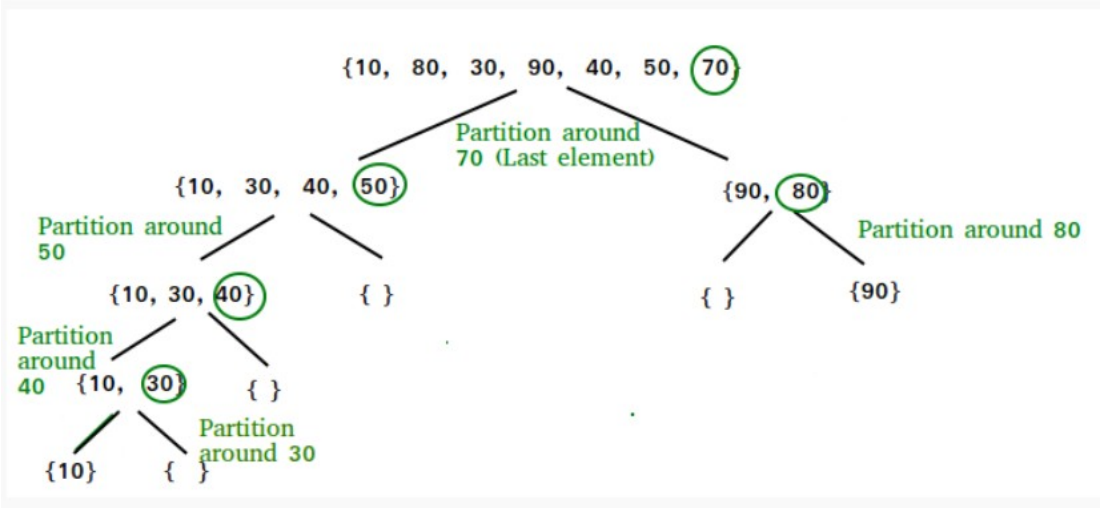
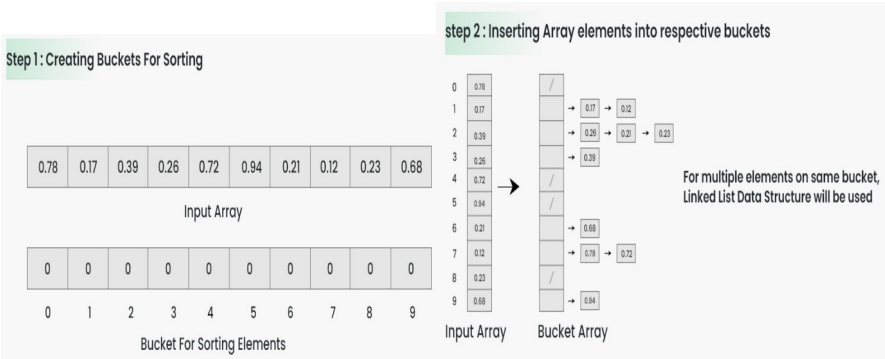
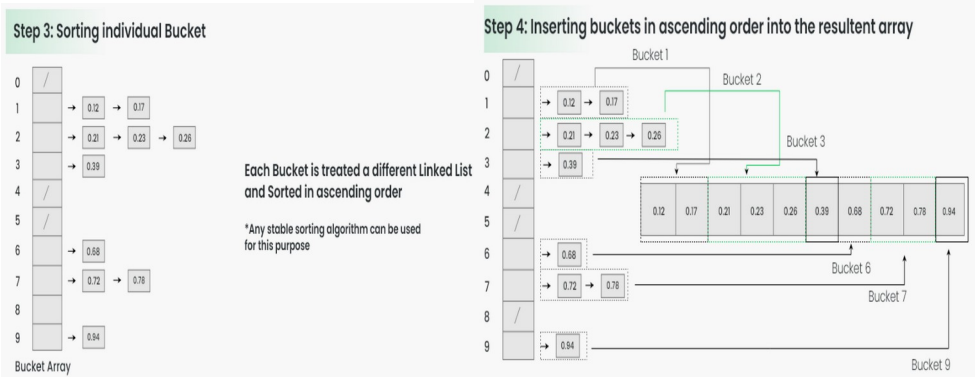
# Priority queues

| Keywords | Notes |
|---|---|
| Prioritetskø | Baggrund: Behovet for at kunne afvikle kører efter et andet princip en FIFO<br>Eksempler:<br>- Printjobs<br>- CPU-schedulering<br>- Hasteorder<br>Nødvendige operationer:<br>- Insert<br>- deleteMin<br>Insert med O(logN) kan tilsikres<br>  <br><br>Følgende skal gælde for alle noder i en prioritetskø: Alle efterkommerer skal være større end noden |

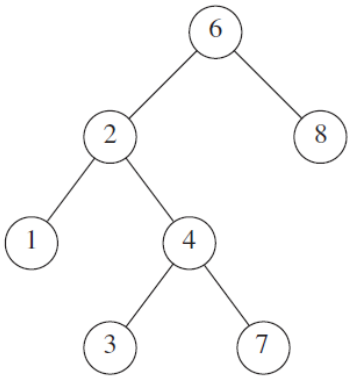
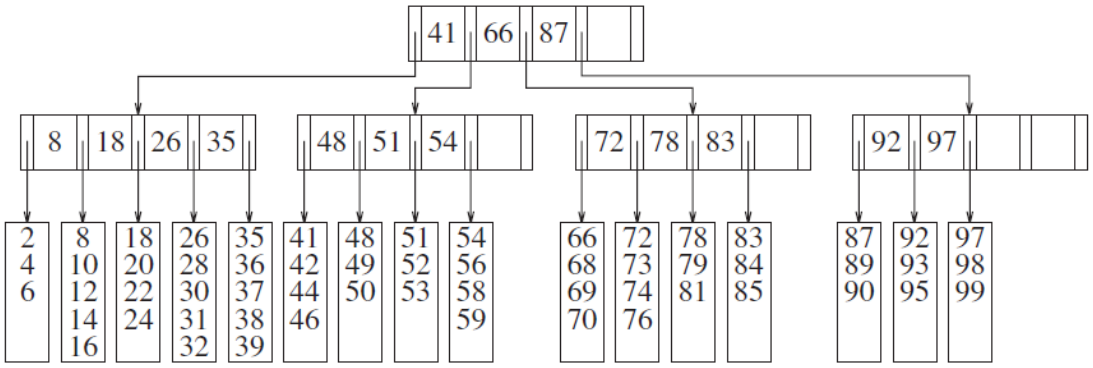
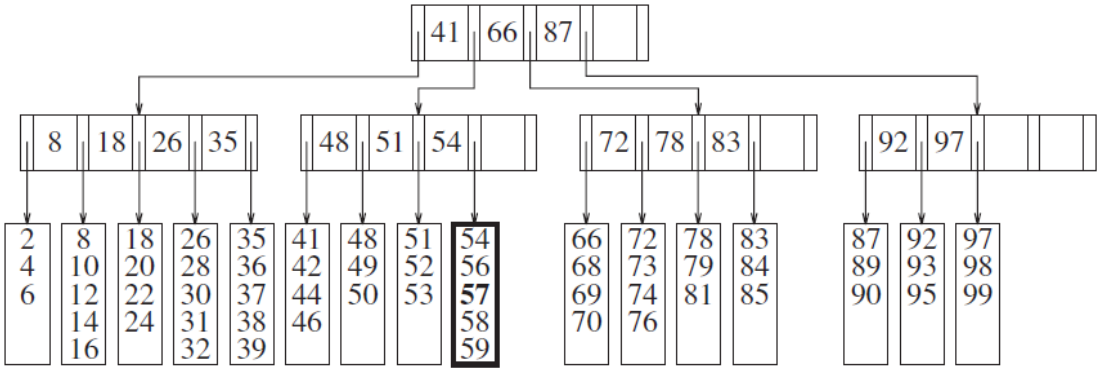
| Keywords | Notes |
|---|---|
| Binære træer | Et priority queue er lavet med et binært træ som datastruktur og implementeres som et array hvor indeks 0 ikke benyttes<br>  |
| Heap order property | Roden indeholder det mindste element<br>Enhver node er mindre end alle dens efterkommere.<br>En prioritetskø kaldes sommetider for en binary heap. |
| Insert | Insert 14<br>  |
| DeleteMin |  |
| Sortering | Prioritetskø kan bruges til sortering fra mindst til størst i $O(N \cdot \log(N))$ tidskompleksitet |

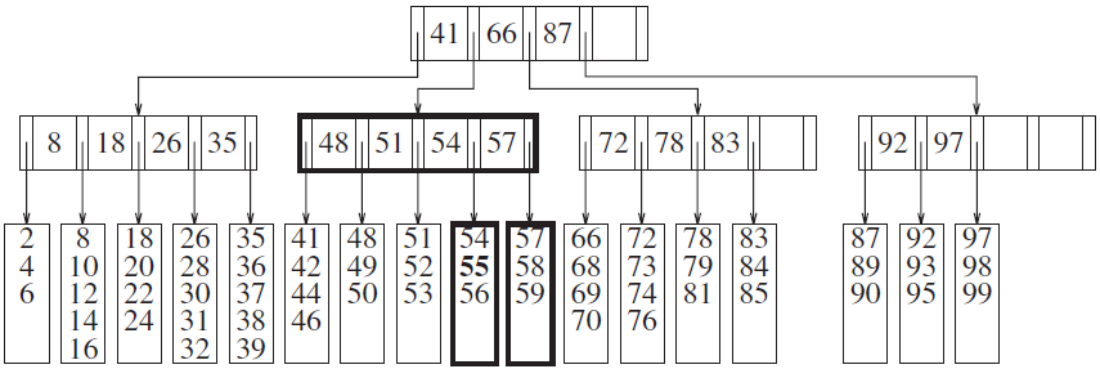
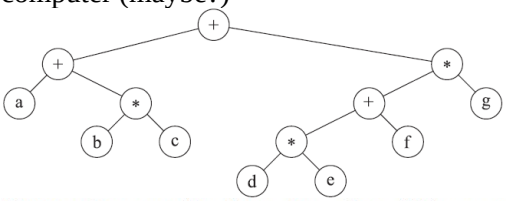
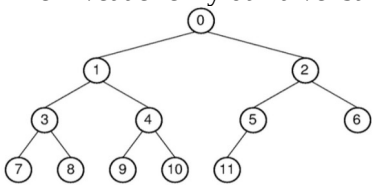
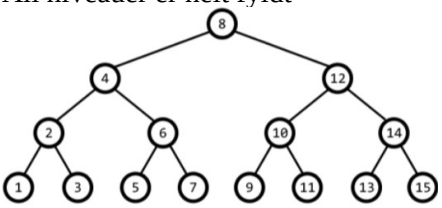
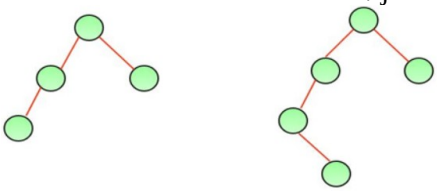
# Sortering

| Keywords | Notes |
|---|---|
| Inversions | Usorterede talpar, hvor det gælder at i<j men a[i]>a[j]. Eksempel:<br>Liste: 34, 8, 64, 51, 32, 21<br>Inversions: (34,8) (34,32) (34,21) (64,51) (64,32) (64,21) (51,32) (51,21) (32,21)<br><br>Det gennemsnitlige antal inversions er N(N-1)/4 |
| Shellsort | Shell sort is a sorting algorithm that is highly efficient and is based on the insertion sort algorithm. This algorithm avoids large shifts, as in insertion sort, where the smaller value is on the far right and must be moved to the far left. Shell Sort reduces its time complexity by utilising the fact that using Insertion Sort on a partially sorted array results in fewer moves.<br>  <br>The method begins by sorting pairs of elements far apart from each other, then gradually narrows the gap between elements to be compared. |
| Telescoping | “Telescoping series is a series where all terms cancel out except for the first and last one. This makes such series easy to analyze.” |
| bubblesort | Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.<br><br>Start med at placer STØRST korrekt<br>Placer 2. STØRSTe korrekt<br>Fortsæt…<br><br>O(N^2) |
| Mergesort | Split the array continuously into halves until a length of 1 is reached, then the lowest pairs of halves are sorted and then the sorted subarrays are connected until at complete sorted array is achieved<br>  |

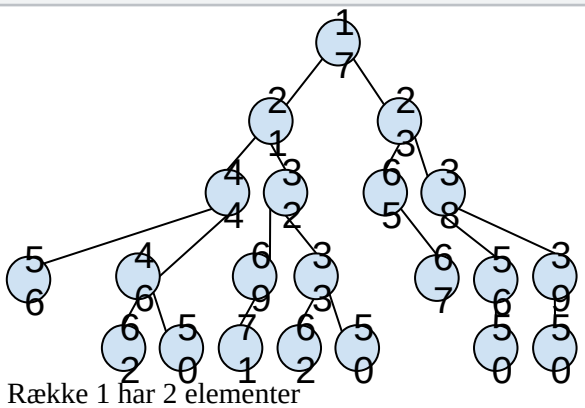
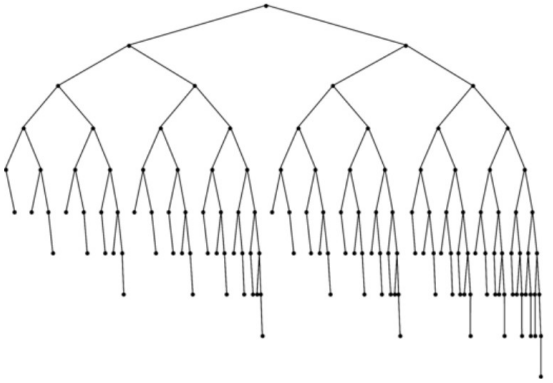
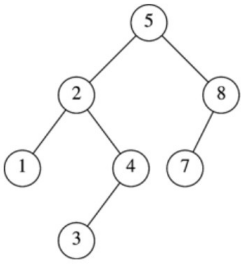
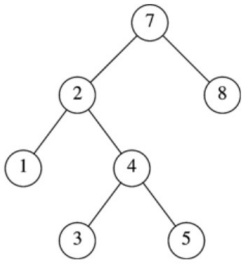
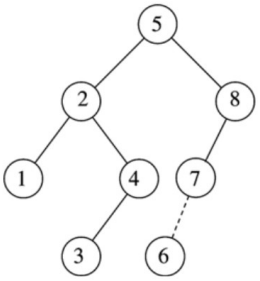
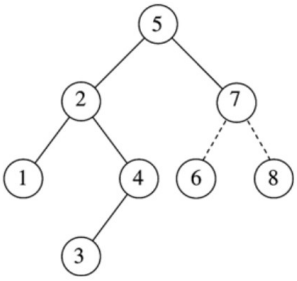
| Keywords | Notes |
|---|---|
| |  <br>O(N log N) |
| Heapsort | Tager elementer i array in i et binary heap. Fra heapet tages det største element fra heap og puttes in i arrayet der skal sorteres.<br><br>O(N log N) |
| Quicksort | Choose an element to partition around. Several policies can be used. Eg. first, last, middle, random.<br><br>After pivot element has been chosen (using some policy), the array is split into two. One with larger and one with smaller elements.<br>  |
| Bucketsort |  <br>  |

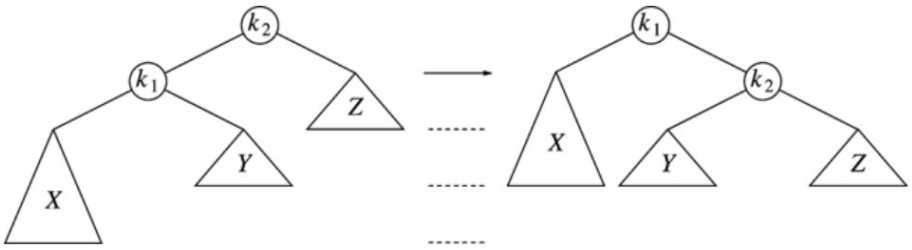
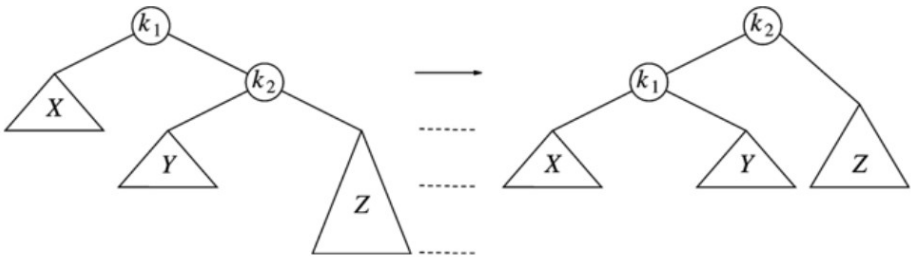
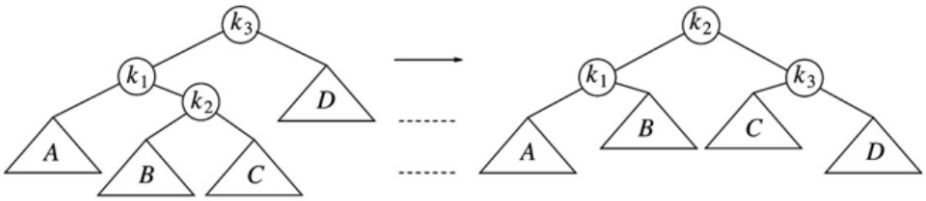
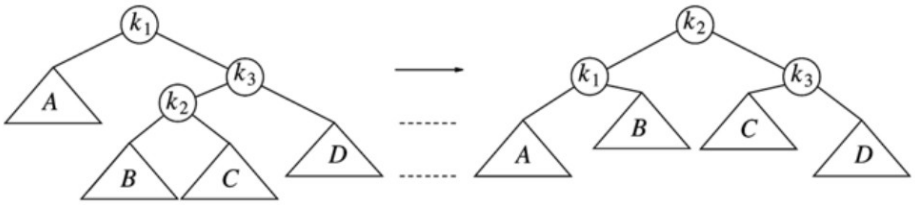
# Træer

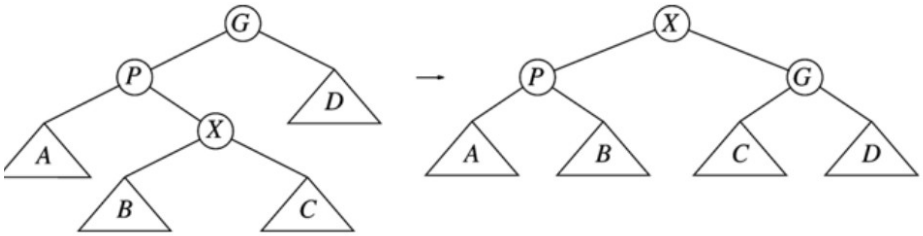
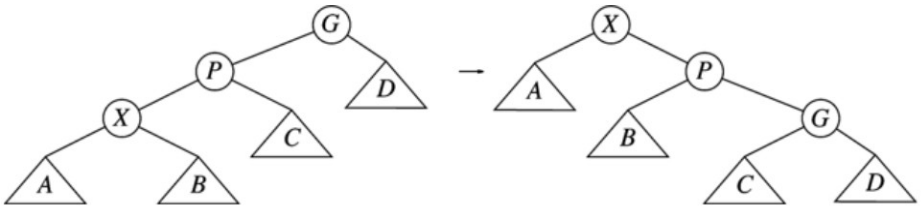
| Keywords | Notes |
|---|---|
| Binært søgetræ | Søgetræ hvor hver node har 0-2 børn, medmindre det er et B-træ af højere orden.<br>  <br><br>Har følgende ting:<br>- Blade (1, 3, 7, 8)<br>- Rod (6)<br><br>Alle noder i det venstre subtræ er mindre end noden, og noder i det højre subtræ er højere, ingen dubletter i B-træet grundet dette |
| 5 ordens B-træ<br><br>Bog side (s. 168) |  <br><br>All noder der ikke er blade har 3-5 børn og 2-4 keys(værdierne)<br>Insertion af 57 vil falde ind mellem 41 og 66 i roden, her er det over 54 og indsættes i denne tabel:<br>  <br><br>Indsættelse af 55 vil splitte bladet i 2, som følgende: |

| Keywords | Notes |
| --- | --- |
| |  |
| Expression Trees | Nodes are operators, except for leafs, this is how mathematical equations are written in a computer (maybe?)<br>  <br>Expression tree for `(a + b * c) + ((d * e + f ) * g)` |
| Dybde/depth | Afstand fra roden til noden man er interesseret i |
| Højde/height | Længste afstand fra noden man er interesseret i til blad |
| Komplet B-træ | Alle niveauer er fyldt fra venstre mod højre, ikke komplet hvis 6 fjernes<br>  |
| Perfekt B-træ | All niveauer er helt fyldt<br>  <br>Har $N = 2^{h+1} - 1$ noder |
| Balanceret B-træ | For at et træ er balanceret må højdeforskellen mellem 2 subtræer højest være 1<br>  <br>Venstre er balanceret, højre er ikke |
| Insert, remove | Inserts an element |

| Keywords | Notes |
|---|---|
| | `private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )`<br>`{`<br>`  if( t == null )`<br>`  return new BinaryNode<>( x, null, null );`<br><br>`  int compareResult = x.compareTo( t.element );`<br><br>`  if( compareResult < 0 )`<br>`  t.left = insert( x, t.left );`<br>`  else if( compareResult > 0 )`<br>`  t.right = insert( x, t.right );`<br>`  else`<br>`  ; // Duplicate; do nothing`<br>`  return t;`<br>`}`<br>Removes an element<br>`private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )`<br>`{`<br>`  if( t == null )`<br>`  return t; // Item not found; do nothing`<br><br>`  int compareResult = x.compareTo( t.element );`<br><br>`  if( compareResult < 0 )`<br>`  t.left = remove( x, t.left );`<br>`  else if( compareResult > 0 )`<br>`  t.right = remove( x, t.right );`<br>`  else if( t.left != null && t.right != null ) // Two children`<br>`  {`<br>`  t.element = findMin( t.right ).element;`<br>`  t.right = remove( t.element, t.right );`<br>`  }`<br>`  else`<br>`  t = ( t.left != null ) ? t.left : t.right;`<br>`  return t;`<br>`}` |
| contains | Returns true if an element is in the B-tree |
| findMin, findMax | **findMin:** leftmost element<br>`private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )`<br>`{`<br>`  if( t == null )`<br>`  return null;`<br>`  else if( t.left == null )`<br>`  return t;`<br>`  return findMin( t.left );`<br>`}`<br><br>**findMax:** right most element<br>`private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )`<br>`{`<br>`  if( t != null )`<br>`  while( t.right != null )`<br>`  t = t.right;`<br><br>`  return t;`<br>`}` |
| isEmpty | Returns true if the tree is empty |
| makeEmpty | Removes all items |
| printTree | Prints tree in sorted order |
| Balance condition | Dybden af træet er $O(\log(N))$<br>Giver en søgetid på $O(\log(N))$<br>Holder kun så længe træet ikke er højre eller venstre tungt |
| Internal path length | $D(N) = D(i) + D(N - i - 1) + N - 1$ |

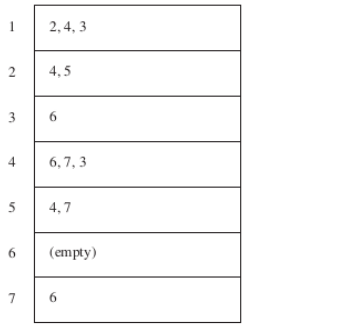
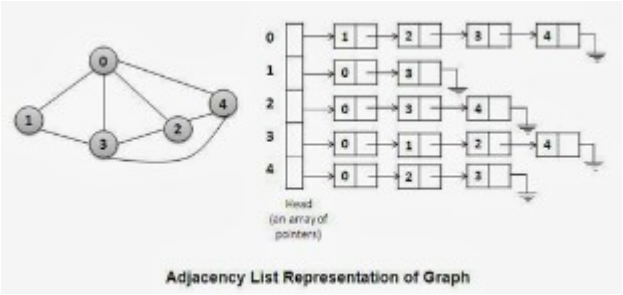
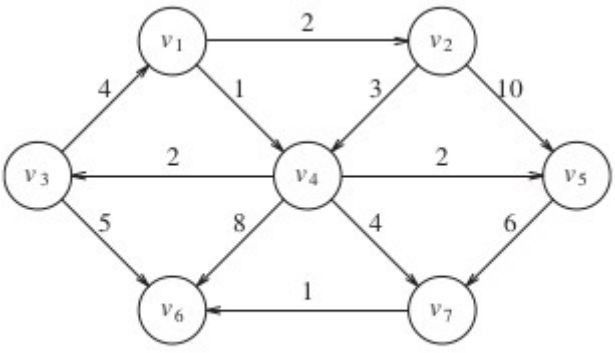
| Keywords | Notes |
|---|---|
| |  <br>Række 1 har 2 elementer<br>Række 2 har 4 elementer<br>Række 3 har 7 elementer<br>Række 4 har 7 elementer<br>$1*2+2*4+3*7+4*7=59$ |
| **AVL-træ** | Et binært søgetræ, der altid er balanceret.<br>Det vil sige at højdeforskellen på en hvilken som helst nodes børn aldrig er større end 1.<br>Dybden er også altid O(logN)<br>  |
| AVL vs Non-AVL |  <br>AVL<br>  <br>Non-AVL |
| Rotationer | Ændringer i træet, som tilsikrer balancen, kaldes rotationer.<br>Dette kan kræves ved indsættelse af en ny node. F.eks. 6 i AVL træet ovenover.<br>   <br>Behovet for rotation kan opstå i fire tilfælde:<br>1. Indsættelse i venstre subtræ af venstre barn af $\alpha$<br>2. Indsættelse i højre subtræ af venstre barn af $\alpha$<br>3. Indsættelse i venstre subtræ af højre barn af $\alpha$<br>4. Indsættelse i højre subtræ af højre barn af $\alpha$ |

| Keywords | Notes |
| --- | --- |
| Single rotation for 1 |  |
| Single rotation for 4 |  |
| Left/Right double rotation for 2 |  |
| Left/Right double rotation for 3 |  |
| balance() | `private AvlNode<AnyType> balance( AvlNode<AnyType> t )`<br>`{`<br>`  if( t == null )`<br>`  return t;`<br><br>`  if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )`<br>`  if( height( t.left.left ) >= height( t.left.right ) )`<br>`  t = rotateWithLeftChild( t );`<br>`  else`<br>`  t = doubleWithLeftChild( t );`<br>`  else`<br>`  if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )`<br>`  if( height( t.right.right ) >= height( t.right.left ) )`<br>`  t = rotateWithRightChild( t );`<br>`  else`<br>`  t = doubleWithRightChild( t );`<br><br>`  t.height = Math.max( height( t.left ), height( t.right ) ) + 1;`<br>`  return t;`<br>`}` |
| Splaying | I nogle applikationer er det meget sandsynligt at en nyligt læst node vil blive efterspurgt meget snart igen.<br>Derfor kunne det måske være smart at forfremme noden i hierarkiet, hvis den ligger dybt.<br>Dette kan gøres med gentagne rotation, og modsat AVL-træet er det ikke nødvendigt at vedligeholde height-attributten. |

| Keywords | Notes |
| --- | --- |
| Splaying zig-zag |  |
| Splaying zig-zig |  |

# Graffer

| Keywords | Notes |
| --- | --- |
| Vertex/node | Noderne i en graf. De noteres som regel med et identificerende træk fx. Et tal eller bogstav |
| edge/arcs | Edges mellem noder to noder. De kan være vægtet og/eller directed |
| directed | En edge går fra $v_i$ node til $v_j$. Noters som ($v_i$, $v_j$) |
| digraphs | Directed graph |
| adjacent | Node w er **adjacent** til v hvis og kun hvis $(v, w) \in E$ |
| cost/weight | En edge kan have en kost tilkoblet |
| path | En **path** i en graf er en sekvens a noder $w_1$, $w_2$, $w_3$,...,$w_N$ der opfylder ($w_i$, $w_{i+1}$) $\in E$<br>En **simple path** er en **path** udelukkende med unike noder. |
| length | Antal edges i en **path** |
| acyclic | En **digraph** uden ringe |
| DAG | Directed Acyclic Graph |
| Strongly connected | Directed graf er **strongly connected** hvis der er en path fra hver node til alle andre. |
| connected | Undirected graf er **connected** hvis der er en path fra hver node til alle andre. |
| Weakly connected | Directed graf er **weakly connected** hvis man ser borte for retninger og der er en path fra hver node til alle andre, |
| complete graph | En **complete** graf har en edge mellem alle vertices |
| adjacency matrix | God til dense (tætte) grafer<br>Pladskompleksitet $\Theta(V^2)$<br>Lagres som A[u][v] hvor (u,v) er edges. Værdierne er true/false hvis der er kant |

| Keywords | Notes |
|---|---|
| dense | $\|E\| = \Theta(\|V\|^2)$<br>Antal kanter er cirka lig antal noder i anden |
| Adjacency list | God til sparse (tynde) grafer<br>Pladskompleksitet $O(\|E\|+\|V\|)$ |
| sparse | En graf hvor antal edges er cirka lineært afhængig af antallet af vertices<br>  <br><br>**Figure 9.2** An adjacency list representation of a graph<br>  <br><br>**Adjacency List Representation of Graph** |
| Topologisk sortering | En ordning af noder i en directed, acyklisk graf, således at hvis der er en sti fra u til v, så skal v komme efter u i sorteringen<br>Cykliske grafer kan ikke sortes topologisk<br>Der kan være mere end en løsning<br>Alle noder skal besøges |
| Korteste sti (Shortest path) | I en O sammenhæng er det at finde den korteste afstand fra en bestemt node **s** til alle andre noder lige så hurtigt som at finde den korteste afstand fra **s** til en anden node<br><br>4 muligheder<br>1. Unweighted shortest path O(\|E\|+\|V\|)<br>2. Weighted shortest path O(\|E\| log\|V\|)<br>3. Weighted shortest path with negative edges O(\|E\|*\|V\|)<br>4. Weighted acyclic graph special case O(N)<br><br>1. USP (unweighted shortest path)<br>Breadth-first search - noder er besøgt i 'niveauer': tættest først, fjernes sidst |
| Dijkstra | Det er en grådig algoritme; løser problemet trinvis ved altid at vælge, hvad der ser ud til at være den bedste mulighed.<br>Vælger altid den nærmeste node<br>Virker ikke med negative omkostninger<br>O(\|E\| log\|V\|)<br>  <br><br>**Figure 9.8** A directed graph $G$ |

| Keywords | Notes |
|---|---|

**Grafen er undirected nedenunder**

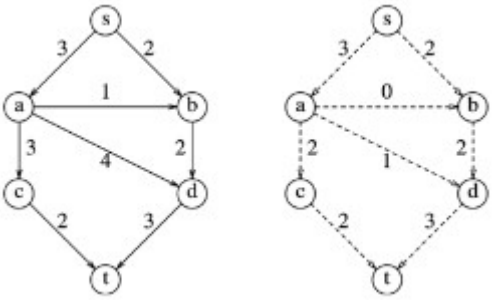
| | Initial State | | | $v_3$ Dequeued | | | $v_1$ Dequeued | | | $v_6$ Dequeued | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ |
| $v_1$ | F | $\infty$ | 0 | F | 1 | $v_3$ | T | 1 | $v_3$ | T | 1 | $v_3$ |
| $v_2$ | F | $\infty$ | 0 | F | $\infty$ | 0 | F | 2 | $v_1$ | F | 2 | $v_1$ |
| $v_3$ | F | 0 | 0 | T | 0 | 0 | T | 0 | 0 | T | 0 | 0 |
| $v_4$ | F | $\infty$ | 0 | F | $\infty$ | 0 | F | 2 | $v_1$ | F | 2 | $v_1$ |
| $v_5$ | F | $\infty$ | 0 | F | $\infty$ | 0 | F | $\infty$ | 0 | F | $\infty$ | 0 |
| $v_6$ | F | $\infty$ | 0 | F | 1 | $v_3$ | F | 1 | $v_3$ | T | 1 | $v_3$ |
| $v_7$ | F | $\infty$ | 0 | F | $\infty$ | 0 | F | $\infty$ | 0 | F | $\infty$ | 0 |
| Q: | $v_3$ | | | $v_1, v_6$ | | | $v_6, v_2, v_4$ | | | $v_2, v_4$ | | |

| | $v_2$ Dequeued | | | $v_4$ Dequeued | | | $v_5$ Dequeued | | | $v_7$ Dequeued | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ | *known* | $d_v$ | $p_v$ |
| $v_1$ | T | 1 | $v_3$ | T | 1 | $v_3$ | T | 1 | $v_3$ | T | 1 | $v_3$ |
| $v_2$ | T | 2 | $v_1$ | T | 2 | $v_1$ | T | 2 | $v_1$ | T | 2 | $v_1$ |
| $v_3$ | T | 0 | 0 | T | 0 | 0 | T | 0 | 0 | T | 0 | 0 |
| $v_4$ | F | 2 | $v_1$ | T | 2 | $v_1$ | T | 2 | $v_1$ | T | 2 | $v_1$ |
| $v_5$ | F | 3 | $v_2$ | F | 3 | $v_2$ | T | 3 | $v_2$ | T | 3 | $v_2$ |
| $v_6$ | T | 1 | $v_3$ | T | 1 | $v_3$ | T | 1 | $v_3$ | T | 1 | $v_3$ |
| $v_7$ | F | $\infty$ | 0 | F | 3 | $v_4$ | F | 3 | $v_4$ | T | 3 | $v_4$ |
| Q: | $v_4, v_5$ | | | $v_5, v_7$ | | | $v_7$ | | | empty | | |

**Figure 9.19** How the data change during the unweighted shortest-path algorithm

Network flow

Hvad er den maksimale flow gennem en graf fra start til slut, når vægten af kanter er flow-begrænsningen

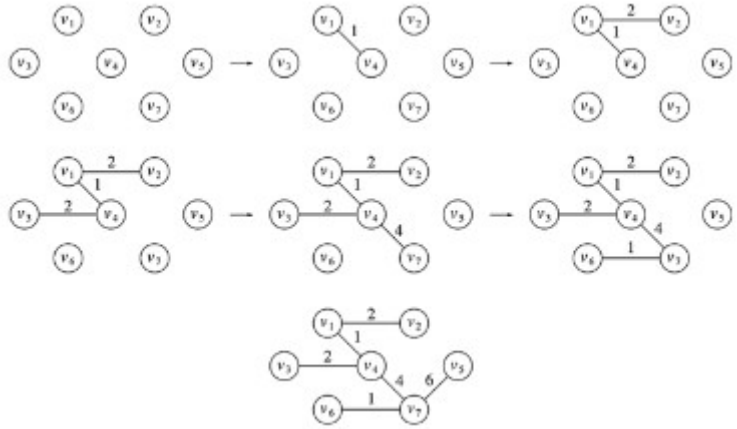
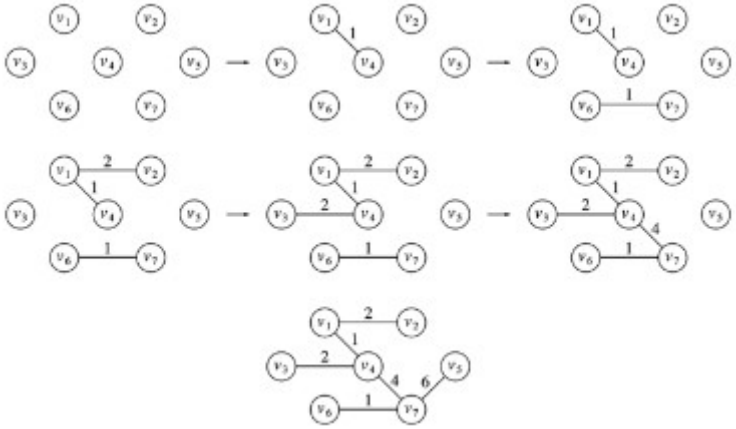



Man skal starter med at kigge på start noden og den edge med mindst vægt.

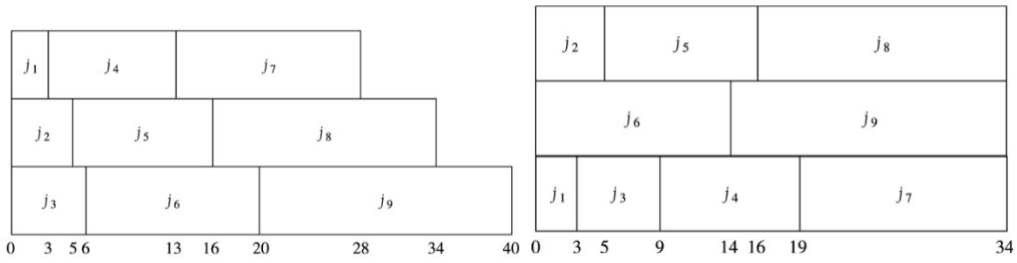
Minimum spanning tree

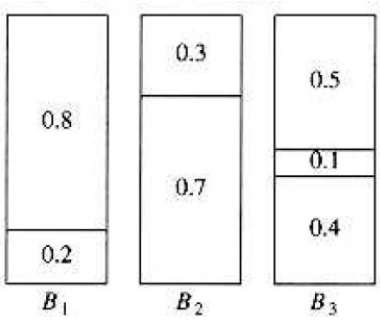
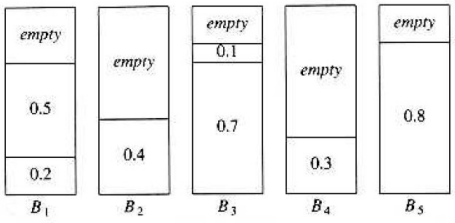
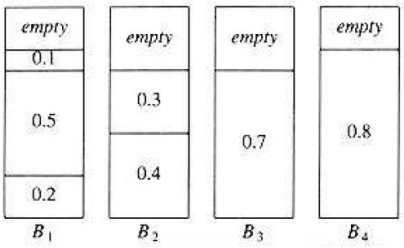
Gælder for en undirected, connected graf
Man skal forbinde alle noder med minimal omkostning
Der er ofte mere end en løsning
Kan udføres på directed graphs

Der findes to metoder: Prim og kruskal.
**PRIM**
I prim skal man start med en edge med lavest kant kost. Derefter tager man den næstlaveste kant kost forbundet til de tidligere forbundet noder og tilføjer dem til minimum spanning tree. Osv.

| Keywords | Notes |
| --- | --- |
| |  <br>**Kruskal**<br>I Kruskal kigger man på disjunkte mængder. Her vil man gerne tage den laveste kant kost der forbinder to disjunkte nodesamlinger. Disjunkte mængder er mængder, hvis fællesmængde er tom.<br>  |

# Algoritmedesign

| Keywords | Notes |
| --- | --- |
| Scheduling | Shortest time first. |
| Multiprocessers |  <br>Reorganiser så kortest mulige totale tid. |
| Huffman | Læs PDF side 351. Den er nem at forstå 🙂 |

| Keywords | Notes |
|---|---|
| Bin packing optimal |  For **0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8** |
| Bin packing next fit |  For **0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8** |
| Bin packing first fit |  **Figure 10.24 First fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8** |

## Gode C++ biblioteker

| Keywords | Notes |
|---|---|
| iostream | #include <iostream> |
| vector | #include <vector> |
| Math | #include <cmath> |
| strings | #inlcude <string> |
| array | #include <array> |
| String manipulation | #include <sstream> |

# Binary heap funktioner

| Keywords | Notes |
|---|---|
| Make BinaryHeap | BinaryHeap<int> heap(100);<br>100 er størrelse, int kan skiftes ud for at sætte andre typer i heapen |
| isEmpty | 1 hvis heap er tom |
| findMin | Finder mindste element, hvis det er strings, så finder man den laveste sum a ascii karakterer |
| deleteMin | Sletter mindste element.<br>Giver man den heap.deleteMin(i), så får i værdien af det mindste element |
| insert | Indsæt element i heap, skal passe til type defineret i BinaryHeap |
| makeEmpty | Sletter alle elementer i heap |
| **Ikke standard alt under her** | |
| printHeap | Printer alle elementer først, i rækkefølge af værdi. |
| size | Giver størrelsen af heapen (længden) |
| findElementIndex(elem) | Returnere index på det element  man giver den, giver -1 hvis det ikke findes |
| decreaseKey(elem,val) | Decreases the value in the element |
| increaseKey(elem,val) | Increases the value in the element |
| remove() | Removes an element and reestablishes the heap |
| contains() | Returnere 1 eller 0 alt efter om element er i heapen |
| perculateUp() | Used as a helper function to reestablish the heap order. |
| | |
| | |
| | |