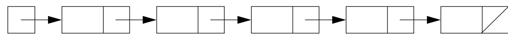
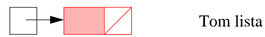
Enkellänkad lista – ”header” och ”sentinel”

Enkellänkad lista

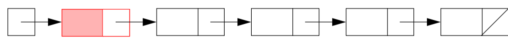
Tom lista



Enkellänkad lista med ”header”



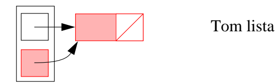
Tom lista



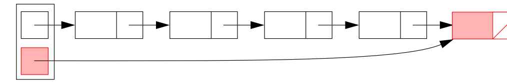
- skapar strukturell enhetlighet för alla datanoder – inget specialfall för pekaren till första datanoden

© Tommy Olsson, IDA/LiTH

Enkellänkad lista – ”header” och ”sentinel”, forts.



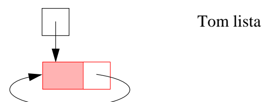
Tom lista



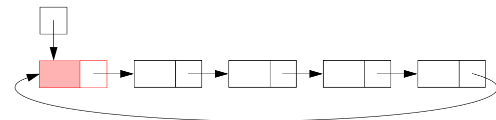
- insättning först och sist kan göras i konstant tid, O(1)
  - borttagning skiljer
- sentinel-noden förenklar sökning
  - sökt värde placeras i ”sentinel”-noden innan sökningen inleds
  - sökt värde hittas alltid, om inte annat i ”sentinel”-noden

© Tommy Olsson, IDA/LiTH

Enkellänkad cirkulär lista med ”header”/”sentinel”



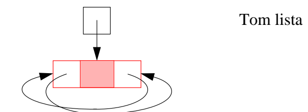
Tom lista



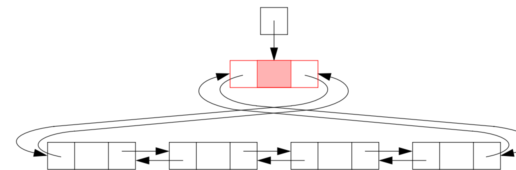
- cirkulariteten bör medföra en fördel jämfört med en rak lista
- insättning först och sist kan göras i konstant tid, O(1)
  - borttagning skiljer
- sökning förenklas – den skuggade noden kan användas som ”sentinel”

© Tommy Olsson, IDA/LiTH

Dubbellänkad cirkulär lista med ”header”/”sentinel”



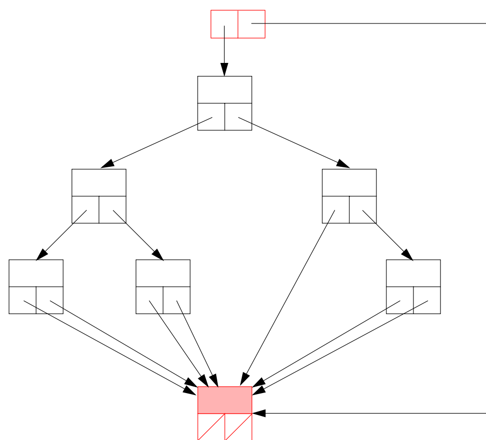
Tom lista



- eliminerar specialfall
- algoritmerna för insättning och borttagning förenklas

© Tommy Olsson, IDA/LiTH

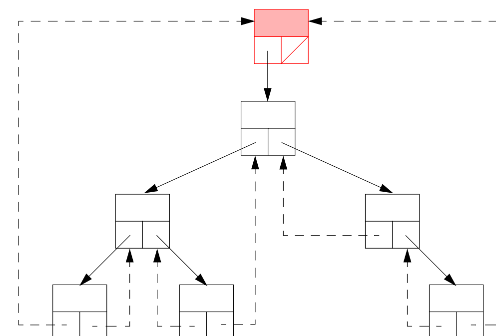
Träd med ”sentinel”



- förenklar algoritmer
- denna trädstruktur används till exempel för implementering av ”top-down”-splayträd

© Tommy Olsson, IDA/LiTH

”Threaded tree”

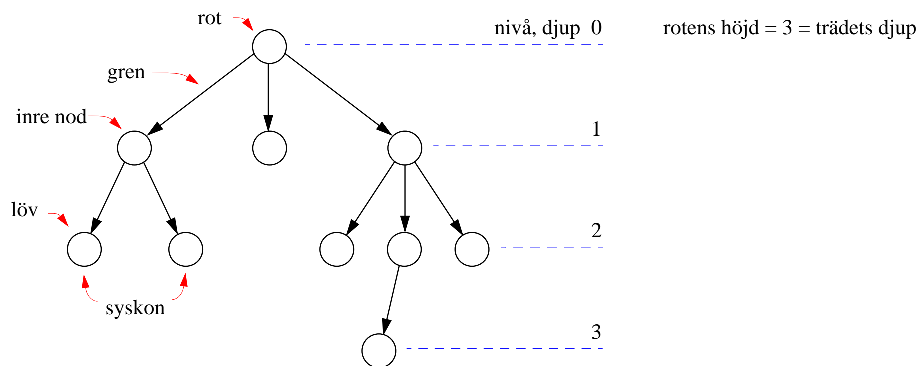


- pekarna i lövnoderna utnyttjas för att peka på noder högre upp i trädet
  - information om pekarnas tolkning måste lagras
- den skuggade noden kan agera både ”header” och ”sentinel”

© Tommy Olsson, IDA/LiTH

Trädbegrepp

Träd är en form av graf.



rotens höjd = 3 = trädets djup

höjd räknas från löv och uppåt – löv har höjden 0

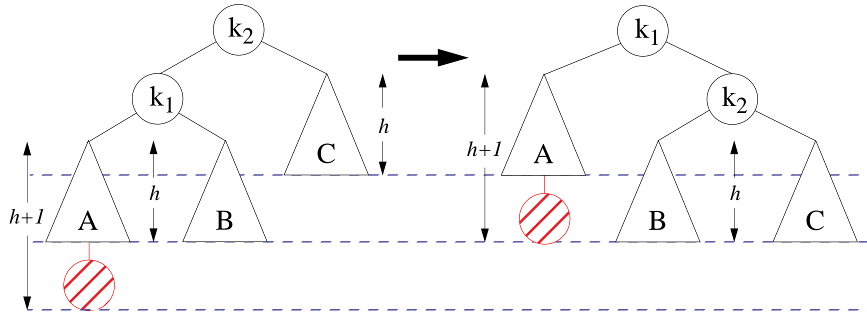
© Tommy Olsson, IDA/LiTH

Trädbegrepp, forts.

- träd – riktad acyklisk graf med extravillkoret att en nod endast får nås via *en* annan nod
- noder – startnod/rotnod – slutnod/lövnod – inre nod – båge/gren – …
- grad – för nod eller för träd = antalet barn/subträd en nod har eller *kan* ha maximalt
  - grad 2 – binärt träd
- storlek = antalet noder i ett träd
- barn/avkomma – förälder/anfader – syskon – …
- väg – väglängd – medelväglängd
- djup = nivå – roten har djupet/nivån 0
- höjd = den längsta vägen från en nod till ett löv – alla löv har höjden 0
- komplett träd
  - helt fyllt till näst sista nivån
  - alla noder (löv) på sista nivån är tätt packade till vänster
  - ett komplett träd kan lagras i ett fält

© Tommy Olsson, IDA/LiTH

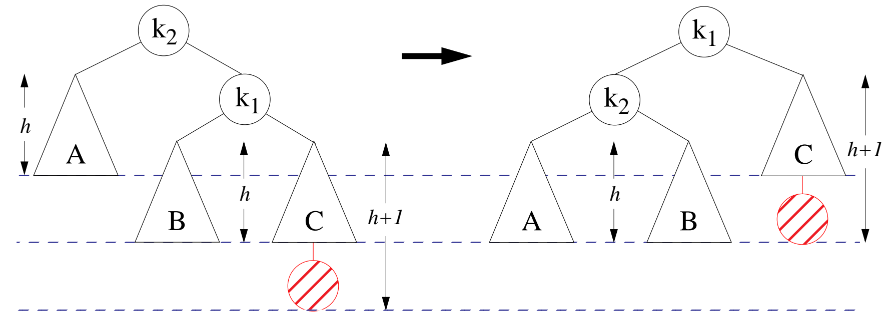
AVL-träd – enkelrotation med vänster barn



Om den streckade noden sätts in under B hjälper ej denna rotation.

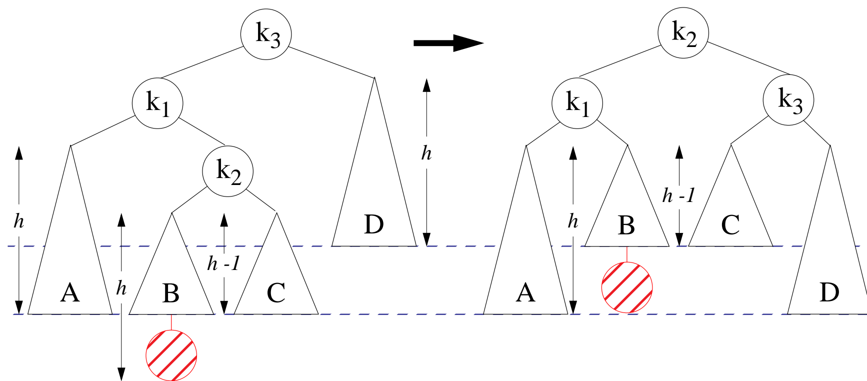
© Tommy Olsson, IDA/LiTH

AVL-träd – enkelrotation med höger barn



© Tommy Olsson, IDA/LiTH

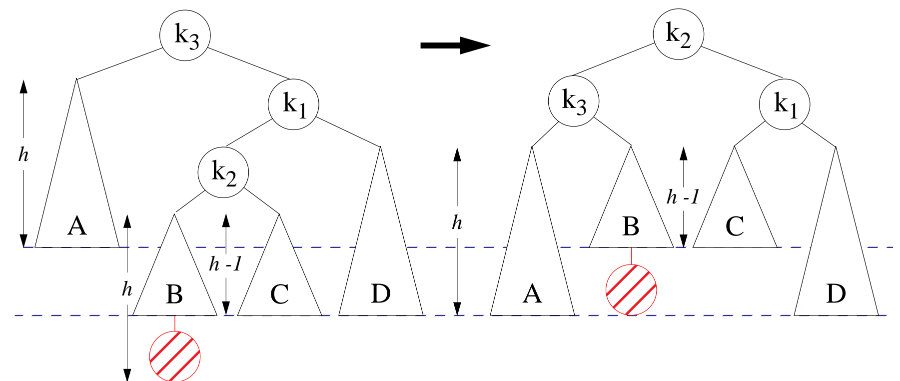
AVL-träd – dubbelrotation med vänster barn



- I princip uppstår obalansen genom insättning i höger subträd till noden k_1
- uppdelningen i två subträd B och C har att göra med hur vi löser obalansen genom att rotera upp moden k_2

© Tommy Olsson, IDA/LiTH

AVL-träd – dubbelrotation med höger barn

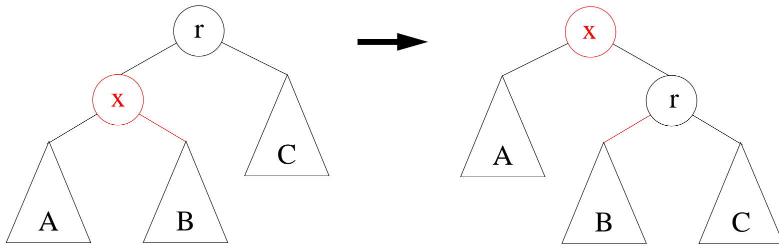


© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – enkelrotation zig-left och zag-right

Enkelrotation görs endast om splaynoden är ett direkt barn till rotnoden, annars någon form av dubbelrotation.

Zig-left utförs då splaynoden **x** är vänsterbarn till rotnoden i trädet (**r**).

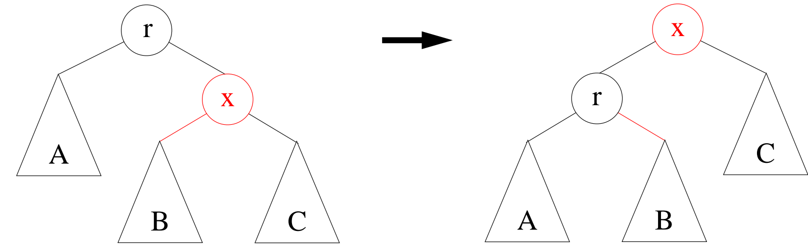


Splaynoden x roteras till roten i trädet – splayningen är klar.

© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – enkelrotation zig-left och zig-right, forts.

Zig-right utförs då splaynoden **x** är högerbarn till rotnoden i trädet (**r**).



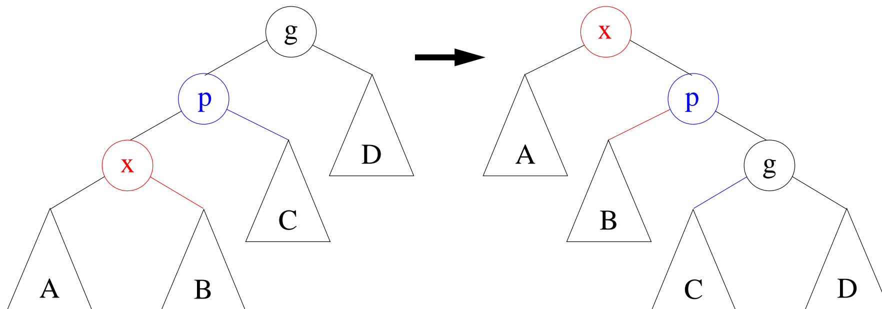
Splaynoden x roteras till roten i trädet – splayningen är klar.

© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – dubbelrotation zig-zig-left och zig-zig-right

Någon slags dubbelrotation utförs alltid om splaynoden har en farförälder.

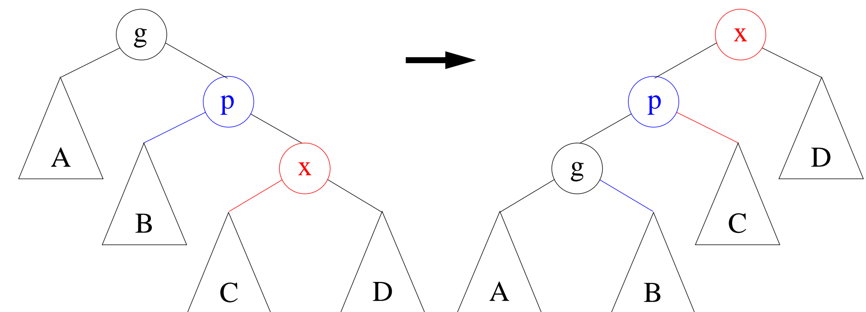
Zig-zig-left utförs då splaynoden **x** är vänsterbarn till föräldern **p** och **p** är vänsterbarn farföräldern **g**.



© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – dubbelrotation zig-zig-left och zig-zig-right, forts.

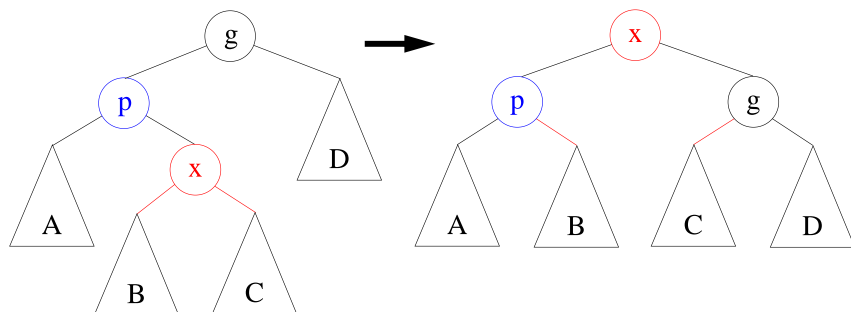
Zig-zig-right utförs då splaynoden **x** är högerbarn till föräldern **p** och **p** är högerbarn till farföräldern **g**.



© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – dubbelrotation zig-left-zag-right och zig-right-zag-left

Zig-left-zag-right utförs när splaynoden **x** är högerbarn till föräldern **p** och **p** är vänsterbarn till farföräldern **g**.

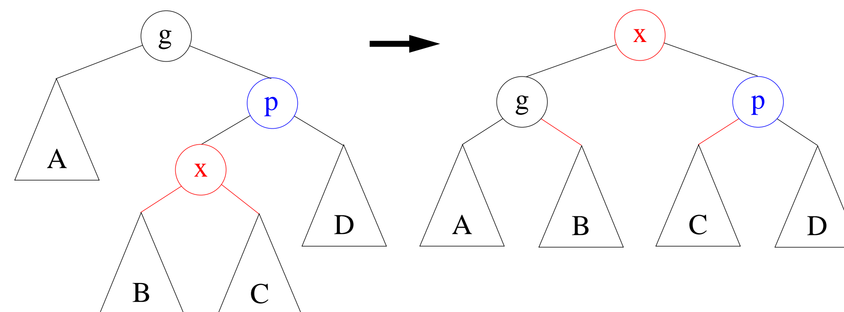


Samma slags rotation som AVL-trädets *dubbelrotation med vänster barn*.

© Tommy Olsson, IDA/LiTH

Bottom-up-splayning – dubbelrotation zig-left-zag-right och zig-right-zag-left, forts.

Zig-right-zag-left utförs när splaynoden **x** är vänsterbarn till föräldern **p** och **p** är högerbarn till farföräldern **g**.



Samma slags rotation som AVL-trädets *dubbelrotation med höger barn*.

© Tommy Olsson, IDA/LiTH

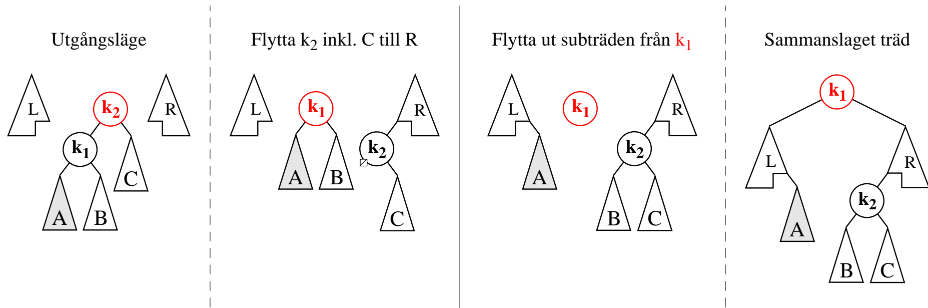
Top-down-splayning – enkelrotation zig-left

Enkelrotation utförs endast då en dubbelrotation inte kan utföras.

Zig-left utförs

- om noden k_1 är vänsterbarn till rotnoden k_2 och k_1 är det sökta värdet, eller
- om det sökta värdet är mindre än k_1 och vänster subträd till k_1 (A) är tomt

Här visas även den avslutande sammanslagningen (efter den heldragna linjen):



- effekten på mitträdet motsvarar en enkelrotation – k_1 hamnar i roten av mitträdet

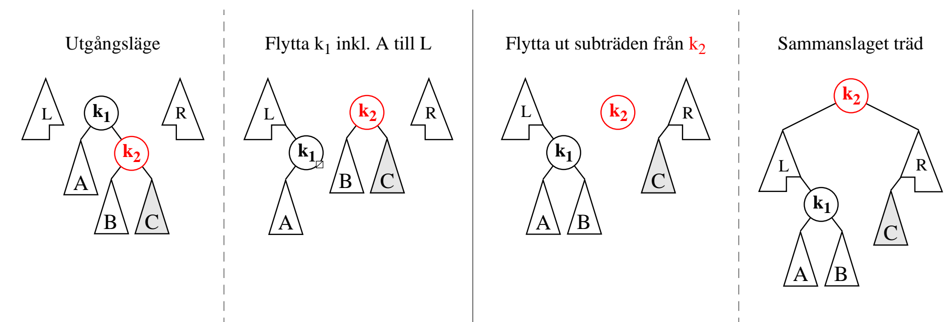
© Tommy Olsson, IDA/LiTH

Top-down-splayning – enkelrotation zig-right

Zig-right utförs

- om noden k_2 är högerbarn till rotnoden k_1 och k_2 är det sökta värdet, eller
- om det sökta värdet är större än k_2 och höger subträd till k_2 (C) är tomt.

Här visas även den avslutande sammanslagningen (efter den heldragna linjen):

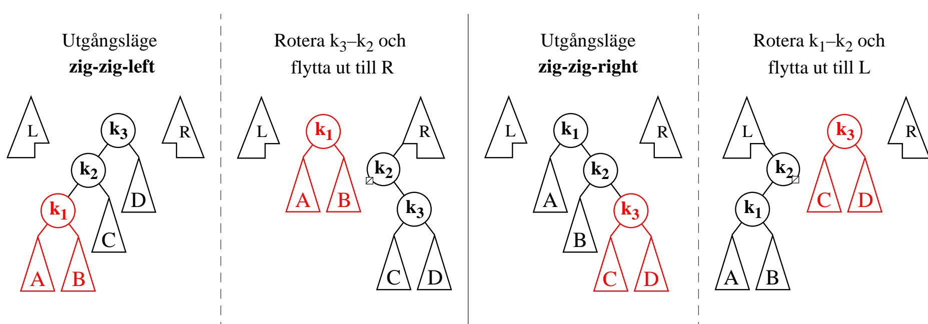


© Tommy Olsson, IDA/LiTH

Top-down-splayning – dubbelrotation zig-zig

Dubbelrotation utförs alltid då det är möjligt.

- *zig-zig-left* görs om det sökta värdet hör hemma i subträdet k_1 och det trädet inte är tomt – annars zig-left
- *zig-zig-right* görs om det sökta värdet hör hemma i subträdet k_3 och det trädet inte är tomt – annars zig-right

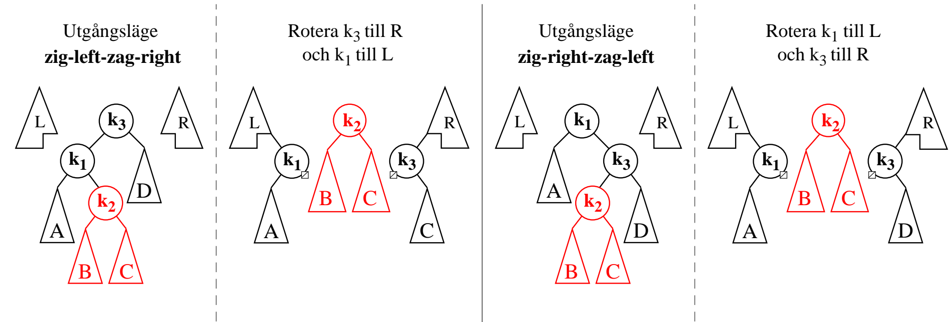


- effekten på mitträdet motsvarar en dubbelrotation – k_1 resp. k_3 hamnar i roten av mitträdet.

© Tommy Olsson, IDA/LiTH

Top-down-splayning – dubbelrotation zig-zag

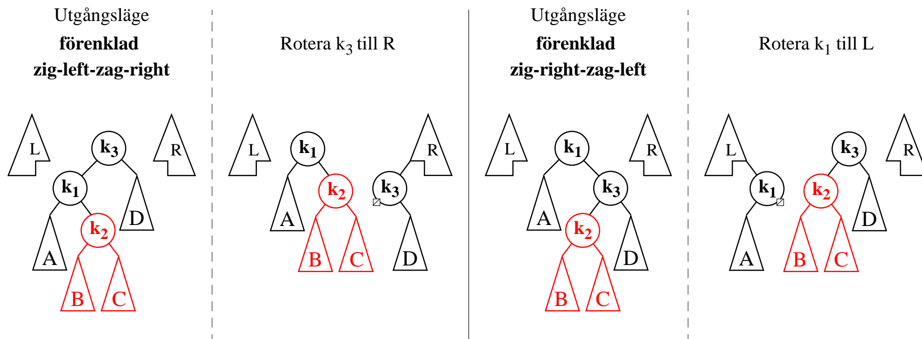
- *ig-left-zag-right* utförs om det sökta värdet hör hemma i vänster innerträd (k_2) och det inte är tomt – annars zig-left
- *zig-right-zag-left* utförs om det sökta värdet hör hemma i höger innerträdet (k_2) och det inte är tomt – annars zig-right



© Tommy Olsson, IDA/LiTH

Förenklad zig-zag – alternativ till zig-zag

Motsvarar en enkelrotation (zig):



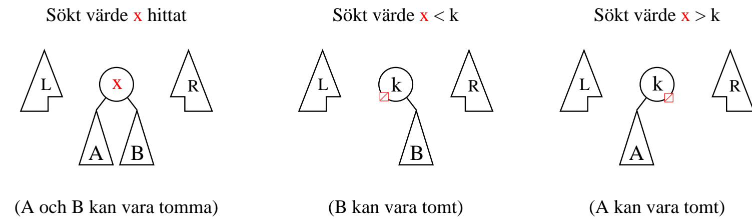
- gör koden påtagligt enklare – algoritmen sammanfaller med den för zig
- kräver två iterationssteg men eliminerar en del kontroller av olika fall
- innebär *inte* två på varandra följande enkelrotationer
  – vad som ska ske i nästa splaysteg beror på trädets nya form och innehåll

© Tommy Olsson, IDA/LiTH

Splayningen upphör

Splayningen upphör när

- det sökta värdet x finns i rotnoden i mittenträdet, eller
- det subträd till rotnoden där det sökta värdet hör hemma är tomt

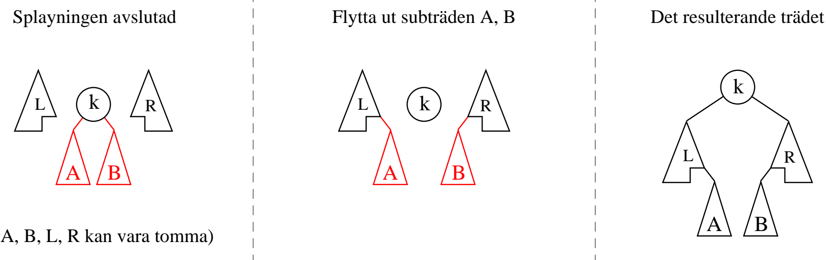


Sista steget i splayningen är att slå ihop de tre träden till ett träd.

© Tommy Olsson, IDA/LiTH

Sammanslagning av träden

- eventuella subträd till mitträdets rot flyttas ut till sidoträden
- den nod som blir kvar i mittenträdet blir rotnod när träden sedan slås ihop



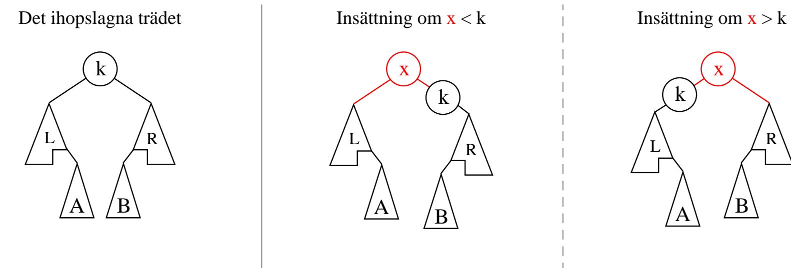
Om inte det sökta värdet x finns i trädet (x = k) gäller följande:

- om x < k så gäller att x > alla värden i L (inklusive A; L < A).
- om x > k, så gäller att x < alla värden i R (inklusive B, B < R).

© Tommy Olsson, IDA/LiTH

Och sedan, efter splayningen?

- enbart sökning – klart – värdet finns i rotnoden om det fanns i trädet
- sätta in ett nytt värde x – ska bli rotnod:



- ta bort ett värde x – finns i rotnoden om värdet finns i trädet:
  – om rotnoden endast har ett subträd kan rotnoden tas bort och ersättas av subträdet, *annars*
  – splaya inorder efterföljade till roten av vänster subträd och sätt in höger subträd som dess högerbarn.

  (visas ej med figurer)

© Tommy Olsson, IDA/LiTH

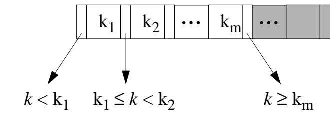
B-träd

- sökträd av vanligtvis hög grad, eller *ordning*, typiskt 100 eller mer
- två slags noder (block)
  - inre noder där utvalda nycklar och pekare/adresser till subträd lagras
  - lövnoderna där dataposterna lagras
- en primärt filorienterad trädstruktur – används till exempel i filsystem och databaser
  - ordningen bestäms typiskt av hur många nyckel-pekar-par som ryms i ett minnesblock
- de B-träd vi ska studera kallas *B^+-träd*

© Tommy Olsson, IDA/LiTH

B^+-träd av ordning *M*

- **M** är B^+-trädets ordning
- **L** är antalet dataposter som ryms i ett löv
- en inre nod kan lagra upp till **M**-1 *nycklar* (upp till **M** *barn*)
  - alla inre noder utom roten ska ha mellan ⌈**M** / 2⌉ och **M** barn
  - en nyckel i en inre nod anger den minsta nyckeln i det högra subträdet
- roten är antingen ett löv eller ett inre nod med minst 2 barn
- alla löv ska finnas på samma djup och ska innehålla ⌈**L** / 2⌉ till **L** dataposter
  - gäller förstås inte om totala antalet poster är mindre än ⌈**L** / 2⌉
- löven kan vara sammanlänkade för att möjliggöra snabb inordertraversering
  - även andra ”genvägar” kan finnas



Relaterade begrepp:

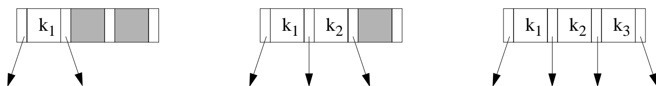
- *noddelning* – görs vid överskott i en nod och man inte vill eller kan adoptera bort
- *nodsammanslagning* – görs vid underskott i en nod om det ej är möjligt att låna/adoptera
- *lån/adoption* – görs vid underskott i en nod och det är möjligt att låna/adoptera från ett närmaste syskon
- *balansering* – görs vid noddelning och adoption och innebär att innehållet fördelas jämnt mellan två noder

© Tommy Olsson, IDA/LiTH

B^+-träd av låg ordning

- M = 3, ⌈**M** / 2⌉ = 2

  - en inre nod ska ha två eller tre subträd
- M = 4, ⌈**M** / 2⌉ = 2



  - en inre nod ska ha två, tre eller fyra subträd

© Tommy Olsson, IDA/LiTH

Insättning

- sök upp lövet där posten (nyckeln) hör hemma
- om det finns plats – sätt in – klart!
- annars (överskott) – dela lövet genom att skapa ett nytt högersyskon
  - fördela posterna jämnt (balansera) – de med större nycklar i det nya lövet
  - nyckel-pekar-par för det nya lövet sätts in till höger om det delade lövets position i föräldranoden
- om föräldranoden (en inre nod) är full – dela
  - fördela nyckel-pekar-paren jämnt (balansera) – de större nycklarna i den nya noden
  - sätt in nyckel-pekar-par för den nya noden i föräldranoden, till höger om den delade nodens position
  - upprepa vid behov
- om proceduren når rotnoden och denna är full – dela rotnoden
  - en ny rotnod skapas
  - pekaren till den gamla rotnoden sätt in längst till vänster
  - nyckel-pekar-par för den gamla rotnodens nya syskon sätts in
  - en ny rotnod har alltid två barn

B-träd växer i höjd vid roten, inte vid löven.

© Tommy Olsson, IDA/LiTH

Insättning, forts.

Om ett nytt B-träd ska skapas från en befintlig mängd av poster kan detta göras effektivare än med vanlig insättning.

- sortera posterna
- skapa en rotnod och ett första löv
  - sätt in pekare till det första lövet längst till vänster i rotnoden
  - fyll på det första lövet till önskad fyllnadsgrad nåtts
- skapa ett nytt löv och fyll till önskad fyllnadsgrad
  - minsta nyckeln och en pekare till lövet sätts in i nästa lediga position i föräldranoden
- då en föräldranod nått önskad fyllnadsgrad
  - dela och balansera föräldranoden
  - sätt in den nya nodens nyckel-pekar-par i föräldranoden
  - upprepa vid behov
  - om rotnoden behöver delas skapas en ny rotnod
- upprepa från tredje punkten ovan till dess alla indata har satts in

© Tommy Olsson, IDA/LiTH

Borttagning

- sök upp lövet där posten ska finnas
- ta bort posten
- om lövet är åtminstone halvfullt – minst $\lceil \mathbf{L} / 2 \rceil$ poster återstår – behöver inte trädstrukturen åtgärdas
  - om posten var den med det minsta nyckelvärdet – justera i föräldranoderna – klart!
- annars – om det finns ett *högersyskon* med fyllnadsgrad > $\lceil \mathbf{L} / 2 \rceil$ – *adoptera*
  - fördela posterna jämnt mellan de två löven (balansera)
  - ny minsta nyckel i högersyskonet – justera i föräldranoderna
- annars – om det finns ett *högersyskon* men dess fyllnadsgrad är $\lceil \mathbf{L} / 2 \rceil$ – *slå ihop*
  - posterna i högersyskonet flyttas över
  - ta bort högersyskonets nyckel-pekar-par i föräldern
  - om underskott uppstår i föräldern åtgärdas det analogt (balansera? slå ihop?)
- om det inte finns ett högersyskon används *vänster syskon*
- om sammanslagningen når rotnoden och då endast ett barn återstår (underskott)
  - ta bort rotnoden
  - det enda återstående barnet blir ny rotnod

B-träd minskar i höjd vid roten, inte vid löven.

© Tommy Olsson, IDA/LiTH

Hashtabeller

Associativ sökstruktur – söknycklar avbildas på tabelladresser

- *hashfunktion* – en nyckeltransformation som ger *hemadressen* för en post/nyckel i hashtabellen
- *kollision* – två poster med olika nycklar erhåller samma hemadress
- *kollisionshanteringsmetoder*
  - linjär sondering
  - kvadratisk sondering
  - dubbelhashning
  - separat länkning
- *klungbildning*
  - *primär klungbildning* – hamnar man i en klunga vid insättning kommer klungan att växa genom insättning sist
  - *sekundär klunga* – består av poster med samma hemadress och samma sonderingsväg
- dimensionering ofta viktig
  - *fyllnadsgraden* får normalt ej bli för hög – typiskt 50–70 %
  - olika kollisionshanteringsmetoder kan ha speciella *storlekskrav* (primtal, multipel av något värde, …)

© Tommy Olsson, IDA/LiTH

Hashfunktioner

Beräknar en nyckels *hemadress* i hashtabellen

- ska vara snabb att beräkna
- bör täcka hela tabellen
- ska ge jämn spridning i tabellen
- vara reproducerbar

Olika metoder som hashfunktioner kan baseras på

- divisionsmetoden – dividera nyckeln med tabellstorleken
- mittkvadratmetoden – kvadrera nyckeln och välj siffrorna i mitten som adress
- sifferurvalsmetoder – plocka ut väl valda siffror ur nyckeln som adress
- ”folding”-metoder – dela upp i nyckeln i delar och addera dessa
- längdberoendemetoder – när nyckelvärden kan variera i längd (t.ex. strängar) låt längden inverka

Många hashfunktioner opererar på heltalsvärden – nycklar måste *förbearbetas* för att erhålla ett heltalsvärde.

En *perfekt hashfunktion innebär att* varje nyckel kan avbildas på en unik hemadress – endast i specialfall.

© Tommy Olsson, IDA/LiTH

Kollisionshantering

Kollision innebär att olika poster/nycklar erhåller samma hemadress – oundvikligt i de allra flesta fall.

Kollisioner kan hanteras på olika sätt:

- flera poster lagras på hemadressen
  - poster tilldelas minne dynamisk och länkas till hemadressen – separat länkning
  - hashtabellen kan vara *hinkorganiserad* – varje adress rymmer ett fixt antal poster
- en del av hashtabellen avsätts som *överskottsarea*
  - hashfunktionen adresserar i detta fall endast den övriga delen – *primärarean*
  - kolliderade poster placeras i överskottsarean och ”länkas”
- en alternativ, ledig plats i hashtabellen söks upp
  - linjär sondering
  - kvadratisk sondering
  - dubbelhashning

© Tommy Olsson, IDA/LiTH

Linjär sondering (med steget 1)

Cirkulär linjärsökning efter ledig plats:

$h_0 = \text{hash}(k)$ *hemadress*

$h_i = (h_0 + i)$ **mod** T $i = 1, 2, \ldots, T$.

Sonderingen avbryts om:

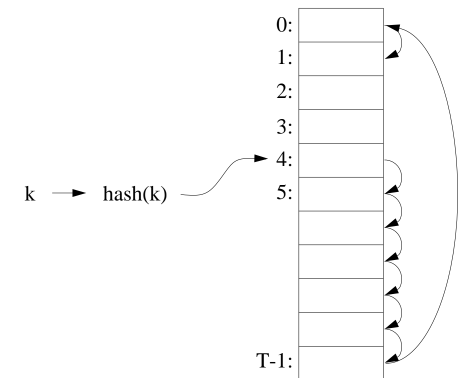
- ledig plats påträffas
- post med samma nyckel påträffas
- åter till hemadressen – tabellen är full

Fördelar:

- enkel
- fullständig tabelltäckning

Nackdelar:

- klungbildning – både *primär* och *sekundär*



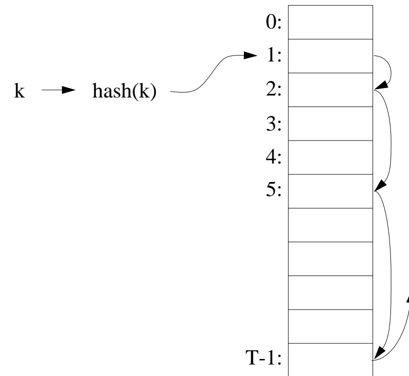
© Tommy Olsson, IDA/LiTH

Kvadratisk sondering

Syftar till att eliminera *primär* klungbildning.

$h_0 = hash(k)$ *hemadress*

$h_i = (h_0 + i^2)$ **mod** T $i = 1, 2, \ldots, T / 2.$



Sonderingen avbryts om:

- ledig plats påträffas
- post med samma nyckel påträffas
- $i > T / 2$ (sedan ej garanterat finna plats och adresser återbesöks)

Fördelar:

- relativt enkel
- eliminerar i princip primär klungbildning

Nackdelar:

- *sekundär* klungbildning
- ej fullständig tabelltäckning

© Tommy Olsson, IDA/LiTH

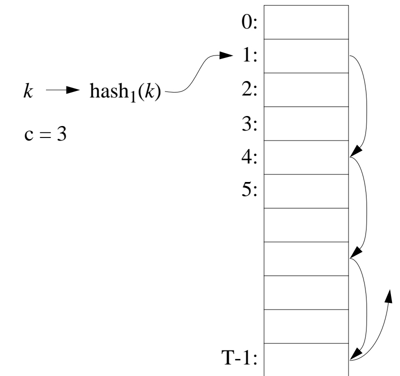
Dubbelhashning

Syftar till att undvika även sekundär klungbildning.

$h_0 = hash1(k)$ *hemadress*

$c = hash2(k) = R - (k$ **mod** $R)$ R primtal < T

$h_i = (h_0 + i * c)$ **mod** T $i = 1, 2, \ldots, T.$



Sonderingen avbryts om:

- ledig plats påträffas
- post med samma nyckel påträffas
- tabellen full

Fördelar:

- eliminerar i princip både primär och sekundär klungbildning

Nackdelar:

- mer komplicerad

© Tommy Olsson, IDA/LiTH

Linjär hashning

Dynamisk tabell.

- *hinkorganiserad* – på varje adress kan rymmas flera poster.
- initialt minst 2 hinkar i tabellen.
- tabellen växer och krymper genom *hinkdelning* resp. *hinksammanslagning.*

En serie av hashfunktioner används:

$h_i(k) = k$ **mod** $(2^i * M)$ $i = 0, 1, 2, \ldots$

- en eller två av dessa används i taget, beroende på om en hink är delad eller ej
- M är det initiala antalet hinkar i tabellen.
- initialt används hashfunktionen $h_0(k)$
- efter första hinkdelningen används både $h_0(k)$ och $h_1(k)$
- då samtliga M hinkar delats har antalet hinkar fördubblats och alla poster är spridda enbart med $h_1(k)$:
  - motsvarar en tabell med initial storlek $2M$
  - enbart $h_1(k)$ används ”initialt”
  - efter första hinkdelningen används även $h_2(k)$

© Tommy Olsson, IDA/LiTH

Linjärhashningsfunktioner

$hash_i(k) = k$ **mod** $(2^i \cdot M)$ $i = 0, 1, 2, \ldots$ M = initial tabellstorlek

$hash_0(k) = k$ **mod** $(2^0 \cdot M) = k$ **mod** $1 \cdot M = [\, M == 2 \,] = k$ **mod** 2 (resten vid division med 2; 0 eller 1)

$hash_1(k) = k$ **mod** $(2^1 \cdot M) = k$ **mod** $2 \cdot M = [\, M == 2 \,] = k$ **mod** 4 (resten vid division med 4; 0, 1, 2 eller 3)

$hash_2(k) = k$ **mod** $(2^2 \cdot M) = k$ **mod** $4 \cdot M = [\, M == 2 \,] = k$ **mod** 8 (resten vid division med 8; 0 – 7)

$hash_3(k) = k$ **mod** $(2^3 \cdot M) = k$ **mod** $8 \cdot M = [\, M == 2 \,] = k$ **mod** 16 (resten vid division med 16; 0 – 15)

…

© Tommy Olsson, IDA/LiTH

Olika sätt att kategorisera sorteringsmetoder

- intern – extern
  - *intern*: alla data ryms i primärminnet – vanligtvis direktadresserbara – ”sortering av fält”
  - *extern*: data på fil – samsortering i samband med sekventiell läsning/skrivning av filer
- jämförande – distributiv
  - *jämförande*: parvis jämförelse av värden (nycklar) – platsbyte eller förflyttning för att ordna
  - *distributiva*: ett värde i sig (sorteringsnyckeln eller viss del av) bestämmer förflyttning
- grundläggande teknik, för jämförande metoder

  urval – utbyte – insättning – samsortering
- enkel – avancerad
  - *enkla*: enkla algoritmer – bygger ofta direkt på en grundläggande teknik – typiskt $O(n^2)$
  - *avancerade*: mer invecklade algoritmer – förfinar tekniken – typiskt $O(n \bullet {}^{m}\log n)$, m vanligtvis 2.
- stabil – ej stabil
  - en stabil metod bibehåller ordningen inbördes mellan lika värden
- naturlig – ej naturlig

  står arbetsinsatsen i proportion till graden av ordning hos indata?

Analys av sorteringsalgoritmer

Framtagning av tidskomplexiteten baseras på analys av

- antalet *jämförelser*
- antalet *förflyttningar* eller *byten* – ett byte = tre förflyttningar

Val av metod kan påverkas av

- *datamängden*
  - en enkel metod kan duga för små datamängder
- *initialordning*
  - *inversioner* – antalet oordnade par i indata; ett mått på (o)ordningen hos indata
  - tre analysfall: *ordnat – omvänt ordnat – slumpmässigt ordnat*
  - ”bästafall” – ”värstafall” – ”normalfall”
- *minnesbehov* – n+1 – 2•n – 3•n
- *stabilitet* – element med likvärdiga nycklar bibehåller inbördes ordning

Interna sorteringsmetoder

- *urval* – Selectionsort, Heapsort
- *utbyte* – Bubblesort, Shakersort, Quicksort
- *insättning* – Insertionsort, Shellsort
- *samsortering* – Mergesort (intern samsorteringsmetod)
- *distributiva* – Radixsort (Bucketsort), Countsort

Anm. Det generella begreppet ”merge sort” (samsortering) används för att generellt beteckna metoder som bygger på samsorteringstekning, dvs även de externa samsorteringsmetoderna.

Externa sorteringsmetoder

Samsorteringsteknik används – två grundläggande tekniker:

- *balanserad* samsortering
  - bestämd, ökande delsekvenslängd i varje pass
- *naturlig* samsortering
  - så långa ordnade delsekvenser som möjligt genereras i varje situation

Metoder

- *balanserad 2-vägs samsortering*
- *naturlig 2-vägs samsortering*
- *2-vägs Polyphase Merge*
  - avancerad form av balanserad tvåvägs samsortering
  - antalet delsekvenser i filerna bygger på Fibonaccis talserie
- *Replacement Selection* – ingen sorteringsmetod
  - teknik för att generera initiala delsekvenser för naturliga samsorteringsmetoder

Bubblesort

```
template<typename Comparable>
void bubblesort(vector<Comparable>& v)
{
   for (int p = v.size(); p > 1; --p)
   {
      for (int j = 0; j < p - 1; ++j)
      {
         if (v[j] > v[j + 1])
         {
            swap(v[j], v[j + 1]);
         }
      }
   }
}
```

(bubblar tunga element mot slutet av fältet)

© Tommy Olsson, IDA/LiTH

Shakersort

```
template<typename Comparable> void shakersort(vector<Comparable>& v)
{
   int left      = 0;
   int right     = v.size() - 1;
   int last_swap = right;           // position för senaste byte

   while (left < right)
   {
      for (int i = right; i > left; --i)
         if (v[i - 1] > v[i]) {
            swap(v[i - 1], v[i]);
            last_swap = i;
         }
      left = last_swap;

      for (int i = left; i <= right; ++i)
         if (v[i - 1] > v[i]) {
            swap(v[i - 1], v[i]);
            last_swap = i;
         }
      right = last_swap - 1;
   }
}
```

© Tommy Olsson, IDA/LiTH

Selectionsort

```
template<typename Comparable>
void selectionsort(vector<Comparable>& v)
{
   for (int p = 0; p < v.size() - 1; ++p)
   {
      int  min = p;

      for (int j = p + 1; j < v.size(); ++j)
      {
         if (v[j] < v[min])
            min = j;
      }

      swap(v[min], v[p]);
   }
}
```

© Tommy Olsson, IDA/LiTH

Insertionsort

```
template<typename Comparable>
void insertionsort(vector<Comparable>& v)
{
   for (int p = 1; p < v.size(); ++p)
   {
      Comparable  tmp = v[p];
      int         j;

      for (j = p; j > 0 && tmp < v[j - 1]; --j)
      {
         v[j] = v[j - 1];
      }

      v[j] = tmp;
   }
}
```

© Tommy Olsson, IDA/LiTH

Shellsort

Den ursprunliga versionen med gap som är multiplar av 2.

```
template<typename Comparable>
void shellsort(vector<Comparable>& v)
{
   for (int gap = v.size() / 2; gap > 0; gap /= 2)
   {
      for (int i = gap; i < v.size(); ++i)
      {
         Comparable  tmp = v[i];
         int         j;

         for (j = i; j >= gap && tmp < v[j - gap]; j -= gap)
         {
            v[j] = v[j - gap];
         }

         v[j] = tmp;
      }
   }
}
```

© Tommy Olsson, IDA/LiTH

Heapsort

(Heapsort 1)

```
template<typename Comparable>
void percolate_down(vector<Comparable<& v, int i, int n)
{
   Comparable  tmp = v[i];
   int         child;

   for (; i * 2 <= n; i = child)
   {
      child = i * 2;                                         // välj vänster barn

      if (child != n && v[child + 1] > v[child])
         child++;                                            // välj höger barn i stället

      if (tmp < v[child])
         v[i] = v[child];                                    // ”hål” vandrar ner
      else
         break;                                              // klart!
   }

   v[i] = tmp;                                               // sätt in tmp i ”hålet”
}
```

© Tommy Olsson, IDA/LiTH

(Heapsort 2)

```
template<typename Comparable>
void Heapsort(vector<Comparable>& v)
{
   for (int i = v.size() / 2; i >= 0; --i)   // generera initial heap
   {
      percolate_down(v, i, v.size());
   }

   for (int j = v.size() - 1; j > 0; --j)
   {
      swap(v[0], v[j]);                      // flytta sista elementet till roten

      percolate_down(v, 0, j);               // återställ heap-ordningen för återstående heap
   }
}
```

© Tommy Olsson, IDA/LiTH

Mergesort

(Mergesort: 1)

```
template<typename Comparable>
void Mergesort(vector<Comparable>& v)
{
   vector<Comparable>  tmp_vector(v.size());

   msort(v, tmp_vector, 0, v.size() - 1);
}


template<typename Comparable>
void
msort(vector<Comparable>& v, vector<Comparable>& tmp, int left, int right)
{
   if (left < right)
   {
      int  center = (left + right) / 2;

      msort(v, tmp, left, center);
      msort(v, tmp, center + 1, right);
      merge(v, tmp, left, center + 1, right);
   }

}
```

© Tommy Olsson, IDA/LiTH

(Mergesort 2)

```
template<typename Comparable>
void merge(vector<Comparable>& v, vector<Comparable>& tmp,
           int left_pos, int right_pos, int right_end)
{
   int  left_end   = right_pos - 1;
   int  tmp_pos    = left_pos;
   int  n_elements = right_end - left_pos + 1;

   while (left_pos <= left_end && right_pos <= right_end)
   {
      if (v[left_pos] <= v[right_pos])
         tmp[tmp_pos++] = v[left_pos++];
      else
         tmp[tmp_pos++] = v[right_pos++];
   }

   while (left_pos <= left_end)                    // element kvar i vänster halva?
      tmp[tmp_pos++] = v[left_pos++];

   while (right_pos <= right_end)                  // element kvar i höger halva?
      tmp[tmp_pos++] = v[right_pos++];

   for (int i = 0; i < n_elements; ++i, --right_end)
      v[right_end] = tmp[right_end];
}
```

© Tommy Olsson, IDA/LiTH

Quicksort

(Quicksort 1)

Sorteringen med den enkla metoden (insertionsort) görs i ett separat, sista steg.

```
const unsigned int CUTOFF = 3;


template<typename Comparable>
void Quicksort(vector<Comparable>& v)
{
   qsort(v, 0, v.size() - 1);                 // qsort sorterar till CUTOFF

   insertionsort(v);
}
```

© Tommy Olsson, IDA/LiTH

(Quicksort 2)

```
template<typename Comparable>
void qsort(vector<Comparable>& v, int left, int right)
{
   if (left + CUTOFF <= right)
   {
      Comparable  pivot = median3(v, left, right);
      i = left;
      j = right - 1;

      for (;;)
      {
         while (v[++i] < pivot);
         while (v[--j] > pivot);

         if (i < j)
            swap(v[i], v[j]);
         else
            break;
      }

      swap(v[i], v[right - 1]);      // flytta pivot
      qsort(v, left, i - 1);
      qsort(v, i + 1, right);
   }
}
```

© Tommy Olsson, IDA/LiTH

(Quicksort 3)

```
template<typename Comparable>
Comparable& median3(vector<Comparable>& v, int left, int right)
{
   int center = (left + right) / 2;

   if (v[left] > v[center])
      swap(v[left], v[center]);

   if (v[left] > v[right])
      swap(v[left], v[right]);

   if (v[center] > v[right])
      swap(v[left], v[center]);

   swap(v[center], v[right - 1]);          // ”göm” pivot

   return v[right - 1];                    // returnera pivot

}
```

© Tommy Olsson, IDA/LiTH