Datastrukturer och algoritmer

Kompendium/komplettering

Innehåll

Inledning

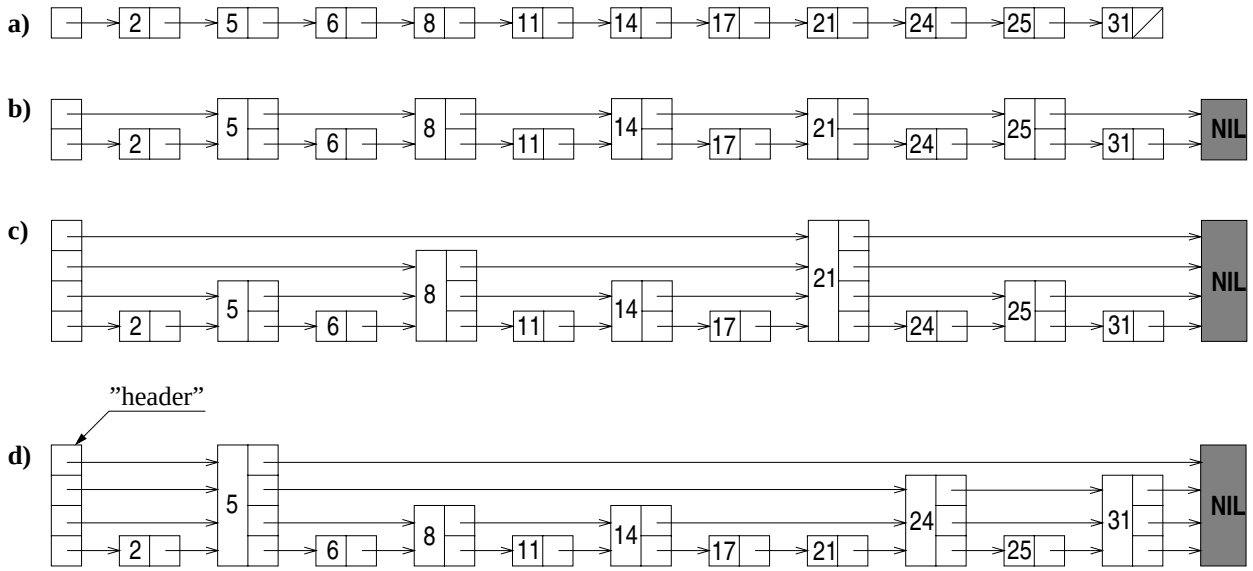
Det här kompendiet ger en översikt över vanligt förekommande datastrukturer och algoritmer, i praktisk användning, teoretiskt eller historiskt.

Underrubriken *Komplettering* avspeglar att kompendiet har en del luckor då det gäller innehåll. Till exempel behandlas inte enkla listor, stack eller kö, utan endast den avancerade liststrukturen skiplista tas upp. Inte heller ingår mängder, grafer och annat som kan förväntas i en mer heltäckande översikt. Analys av datastrukturer och algoritmer har antingen utelämnats eller genomförs mer resonemangsmässigt än teoretiskt.

1 Skiplista

Skiplistan bygger på en sannolikhetsbaserad balansering, i stället för de striktare former av balansering som används i vissa träd. Det gör att algoritmerna för insättning och borttagning i en skiplista blir både enklare och snabbare än motsvarande operationer i balanserade träd.

En skiplista är i grunden en ordnad, enkellänkad lista men vissa listnoder har flera framåtpekare, vilket möjliggör snabbare sökning än i en vanlig enkellänkad lista. Skiplistan är, trots de extra pekarna, minneseffektiv och kan konstrueras med en marginell ökning av antalet pekare per listelement. Ingen annan information behöver heller lagras i noderna.



Figur 1. Exempel på enkellänkad lista och skiplistor.

1.1 Skiplistans uppbyggnad

I en enkellänkad lista med *n* element (figur 1a) kan man behöva söka igenom hela listan för att hitta ett visst värde. Om listan innehåller *n* noder krävs i värsta fall *n* söksteg och förväntad medelsökväg är *n*/2 söksteg. Om varannan nod förses med ytterligare en pekare som pekar två noder framåt (figur 1b) behöver bara *n*/2+1 noder besökas i värsta fall. Om man i var fjärde nod har ytterligare en pekare som pekar fyra steg framåt (figur 1c), får man i värsta fall undersöka *n*/4+2 noder, osv. En lista uppbyggd på detta systematiska sätt kallas för en *perfekt balanserad* skiplista.

En nod med *k* framåtpekare kallas en nivå-*k*-nod. I en perfekt balanserade skiplista gäller att 50% av noderna är nivå-1-noder, 25% är nivå-2-noder, 12.5% är nivå-3-noder, osv. (figur 1c). Om man på detta vis, i var 2^i:te nod har en pekare som pekar *i* noder framåt, fördubblas antalet pekare i listan men det reducerar antalet noder som kan behöva att undersökas till i värsta fall $^2\log n$. NIL-noden som finns sist i skiplistan är en speciell nod som avslutar alla nivåer i listan, en ”sentinel-nod”

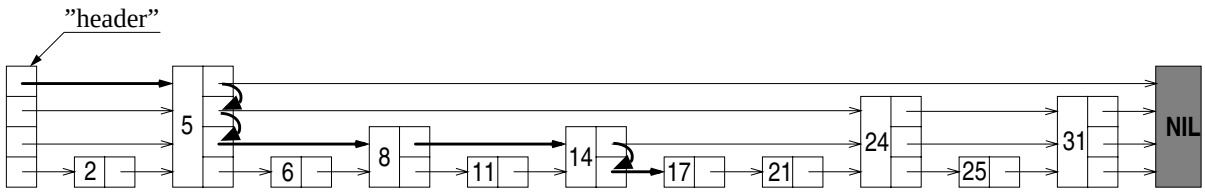
En perfekt balanserad skiplista är effektiv vid sökning men ohanterlig vid insättning och borttagning. Man kan dock använda en mindre strikt, sannolikhetsbaserad bestämning av antalet framåtpekare. Då en ny nod ska sättas in slumpas nivån för den nya noden. Man kan konstruera en slumpgenerator som statistiskt bibehåller fördelningen med 50% nivå-1-noder, 25% nivå-2-noder, osv, men även andra fördelningar kan användas.

I en sannolikhetsbaserad skiplista kommer en nods *i*:te framåtpekare att ange nästa nod i listan som är en nivå-*i* nod eller högre (figur 1d), i stället för att, som i den perfekt balanserade listan, ange noden som är 2^{i-1} noder framåt. Insättning och borttagning av en nod kräver endast en begränsad förändring av listans struktur och framför allt berörs inte andra noders *nivå*.

Operationer på skiplistor inleds vanligtvis med en sökning som startar i *header-noden*. Normalt startar sökningen på den aktuella nivån för listan, dvs den i figurerna översta nivån av pekare. Sökningen leder framåt/nedåt i strukturen, exakt hur beror på vilket värde som söks och listans aktuella innehåll:

- om sökt värde är *större än* det i den aktuella noden stegar man *framåt* på samma nivå
- om sökt värde är *mindre* det i den aktuella noden stegar man *nedåt* i den aktuella noden
- om sökt värde är *lika* med det i den aktuella noden avslutas sökningen

I figur 2 visas sökvägen i det fall värdet 17 söks. Sökningen börjar i header-noden på nivå 4. Den pekaren anger noden 5 (<17) och man stegar framåt på nivå 4 till nod 5 och undersöker vad dess pekare på nivå 4 anger. Det visar sig vara NIL (innehåller ett värde > 17), så man stegar ned till nivå 3 i nod 5 och undersöker vad den pekaren anger. Det är nod 24 (>17), så man stegar ned till nivå 2, osv till dess (i detta fall) sökt värde hittas.



Figur 2. Sökväg i en skiplista vid sökning efter värdet 17.

1.2 Skapa skiplista

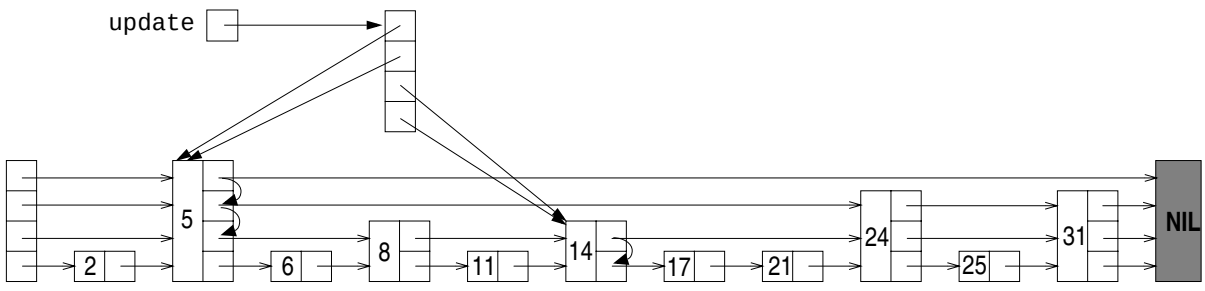
En skiplista ska initialt bestå av en *header-nod* och en NIL-*nod*. Listans nivå sätts till 1 och samtliga pekare i *header-noden* (det kan vara ett fixt antal) ska peka på NIL-*noden*.

1.3 Söka i skiplista

Vid sökningar som kan innebära att listan kan ska ändras, dvs vid insättning och borttagning, sparas pekarna till de noder som innehåller pekare som kan behöva ändras i ett fält, `update`.

- sökt värde placeras i NIL-*noden*, som därmed fungerar som en vaktpost i slutet av listan
- sökningen inleds i *header-noden* på den för tillfället högsta nivån i listan
- så länge pekaren på aktuell nivå anger en nod som innehåller ett värde som är mindre än sökt värde stegar man framåt till nästa nod på den aktuella nivån.
- då ett värde som är mindre än eller lika med det sökta värdet påträffas sparas pekaren till aktuell nod i `update`.
- sökningen fortsätter sedan på närmast lägre nivå

Figur 3 visar innehållet i `uppdate` efter att sökning genomförts efter antingen 15, 16 eller 17.



Figur 3. Innehållet i `uppdate` *efter sökning efter 15, 16 eller 17.*

Fältet `update` i figur 3 ger åtkomst till de noder där framåtpekarna kan behöva ändras, om man vill sätta in eller ta bort det värde som har sökts.

I sökalgoritmen ovan startar man alltid på listans högsta nivå och arbetar framåt/nedåt i list-strukturen. Det finns möjlighet att effektivisera sökningen ytterligare, vilket beskrivs i [1].

1.4 Insättning i skiplista

Insättning utförs i två moment. Först görs en sökning efter det värde som ska sättas in, enligt vad som beskrivits tidigare. Då sökningen är klar har man hamnat i en nod, som antingen innehåller värdet ifråga eller framför vilken en ny nod med sökvärdet ska länkas in. I fältet `update` finns pekare till de noder där det finns pekare som kan behöva ändras då den nya noden sätts in.

Efter den inledande sökningen kontrolleras om sökt nyckel finns i listan eller ej. Om en ny nod ska sättas in beräknas den nya nodens nivå. Om nivån blir större än listans aktuella nivå kompletteras `update` med de ytterligare pekare som detta kräver. De nya pekarna ska samtliga peka på *header-noden*, vilken kommer att vara den direkta föregångaren till den nya noden på de nytillkomna nivåerna. Man bör endast öka listans nivå med en nivå i taget för att undvika att enstaka noder erhåller en onödigt hög nivå jämfört med övriga noder.

Den nya noden länkas in i listan. Upp till den nivå som den nya noden har erhållit ska pekarna i de noder som anges av pekarna i `update` kopieras till den nya noden. Pekarna i de noder som anges av pekarna i `update` ska ändras till att peka på den nya noden.

Om man vill sätta in 15 och noden erhåller nivån 3 ska båda pekarna i nod 14 ändras till att peka på den nya noden, liksom pekaren på nivå 3 i nod 5. Pekaren på nivå 4 i nod 5 påverkas ej.

1.5 Borttagning ur skiplista

Borttagning inleds med en sökning i listan. Om sökt nyckel finns ändras alla pekare som pekar på den nod som ska tas bort till motsvarande pekare i den nod som ska tas bort. Därmed är den sökta noden urlänkad. Slutligen bestäms listans nya aktuella nivå, den borttagna noden kunde ha varit den för tillfället enda noden med den högsta nivån i listan.

Borttagning av nod 17 skulle endast påverka pekaren på den lägsta nivån i nod 14. Däremot skulle borttagning av nod 24 påverka pekarna i noderna 5, 14 och 21.

1.6 Analys

Någon analys av skiplistor görs ej. Vi bara konstaterar att sökning normalt är en arbetsinsats av storleksordningen O(log*n*). Skiplistor kan genom sin relativa enkelhet vara intressanta att använda i stället för balanserade träd. Skiplistan kombinerar det balanserade trädets sökeffektivitet med den raka länkade listans sekventiella egenskaper. För mer detaljer se [1].

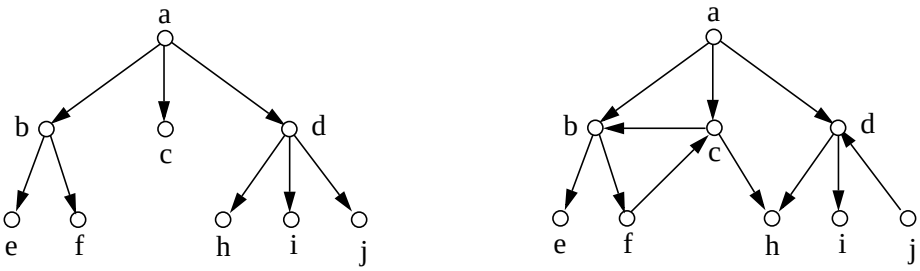
1.7 Litteratur

William Pugh, *Skip List: A Probabilistic Alternative to Balanced Trees*, Communications of the ACM, Juni 1990, Vol. 33, Nr 6, sid. 668-676.

2 Träd

Trädstrukturer gör det möjligt att lagra information på sätt som kan ge effektiva sökstrukturer. Man ser inte så ofta träd förekomma som bibliotekskomponenter, som exempelvis lista, stack, kö och hashtabell, däremot som implementeringsstruktur för andra datastrukturer.

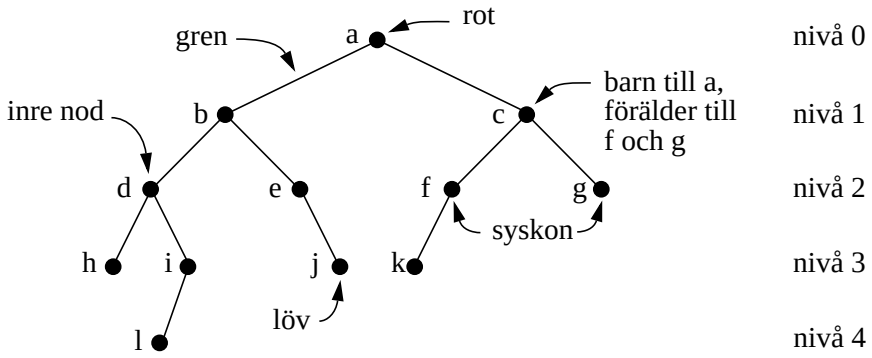
Formellt är träd en begränsad form av *riktad acyklisk graf*. *Riktad* innebär varje båge i grafen har en riktningoch att den kan följas från en nod till en annan men inte tillbaka (vilket rekursion kan lösa). *Acyklisk* innebär att det ej får finns några vägar som leder runt i grafen, så att man kan komma tillbaka till en nod som redan har besökts annat än att gå baklänges. Den begränsning som finns i jämförelse med generella riktade acykliska grafer är att varje nod bara kan ha en båge, eller *gren*, som leder till noden. Grafen till vänster i figur 4 visar ett träd, medan grafen till höger ej är ett träd, dels därför att det finns en cykel b-f-c och dels att noden h kan nås via två olika vägar.



Figur 4. Träd är en form av riktad acyklisk graf (t.v.)

2.1 Terminologi

Den första nod man kommer till i ett träd kallas trädets *rot*. En nods *barn* är de noder som direkt kan nås från noden, även kallade *direkt efterföljare*. En nods *förälder* är den unika nod som är nodens *direkta föregångare*. Noder som har samma förälder kallas *syskon*. En nod utan barn kallas *löv*, *slutnod*, eller *terminalnod*. En nod som inte är ett löv är en *inre nod*.



Figur 5. Terminologi.

En nods *anfäder* är samtliga noder som finns på vägen från roten till noden. Roten är den gemensamma anfadern för alla noder i ett träd. En nods *avkommor* är samtliga noder som har noden som anfader. Observera att en nod i grafteoretisk mening är både sin egen anfader och egen avkomma. En nod är en *äkta anfader* (*äkta avkomma*) till en annan nod om noden ifråga är en anfader (avkomma) till den andra noden, men det omvända inte gäller.

En *väg* i ett träd är den unika följden av bågar mellan en nod och en anfader. *Väglängden* till en nod avser vanligtvis antalet bågar från noden till roten. Väglängden till noden d i trädet i figur 5 är 2. I sökträd är *medelväglängen* av speciellt intresse och den beräknas som summan av alla noders väglängder dividerad med antalet noder. Medelväglängden ger ett mått på förväntat sökarbete.

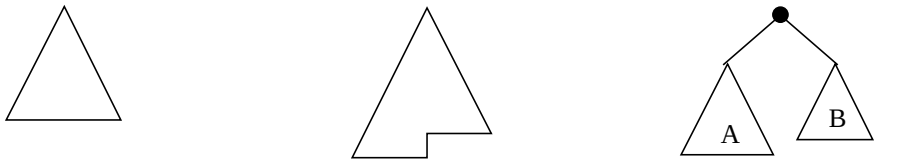
Roten i ett träd sägs befinna sig på nivå 0, rotens barn på nivå 1, dess barnbarn på nivå 2, osv. En nods *djup*, eller *nivå*, är väglängden från roten till noden. En *nods höjd* är längden av den längsta vägen från noden till ett löv (nod b i trädet i figur 5 har höjden 3). Ett *träds höjd* är rotens höjd, dvs den längsta vägen till något löv (trädet i figur 5 har höjd 4). Löv har höjden 0 och ett tomt träd har definitionsmässigt höjden -1.

En nods *grad* är dess antal barn (i figur 5 har nod b grad 2, nod f grad 1 och nod h grad 0). Ett träds *grad* är den högsta graden hos någon noder (trädet i figur 2 har följaktligen grad 2 eftersom ingen nod har fler än två barn). Ett *binärt träd* är ett träd med grad mindre än eller lika med 2 (trädet i figur 5 är ett binärt träd). Ofta avser man dock snarare det maximala antalet barn som trädnoderna *kan* ha, den fysiska begränsningen, när man talar om ett träds grad Ett träd med grad 1 kallas *degenererat*.

Ett träds *storlek* är det totala antalet noder i trädet. Ett subträds storlek är antalet noder i subträdet inklusive subträdets rot.

I ett *orienterat binärt träd* kallas det ena barnet *vänster barn* och det andra *höger barn* (i figur 5 är nod d vänster barn till nod b och nod e är höger barn till nod b).

Ibland är det enbart av intressant att visa ett träds övergripande struktur, inte samtliga noder i trädet. I figur 6 visas några sådana exempel på *symboliserade träd*.



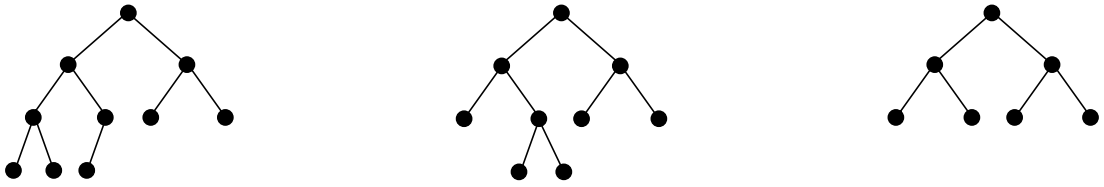
a) godtyckligt träd b) komplett träd c) nod med två subträd

Figur 6. Symboliserade träd.

Ett *komplett binärt träd* med höjd *h* är ett binärt träd som är fullt ner till nivån *h*-1 och med löven på nivå *h* tätt packade åt vänster (se figur 7a).

En nod i ett binärt träd är *full* om den har grad 2 (i figur 5 är noderna a, b, c och d fulla). I ett *fullt träd* har alla noder utom lövnoderna två barn (se exempel 7b).

Ett *perfekt binärt träd* med höjd *h* har enbart löv på nivå *h* och alla dess inre noder är fulla (se figur 7c). Lägger man till en nod i ett fullt binärt träd kommer dess höjd att öka.



a) komplett binärt träd med djup 3 b) fullt men ej komplett c) perfekt binärt träd med djup 2

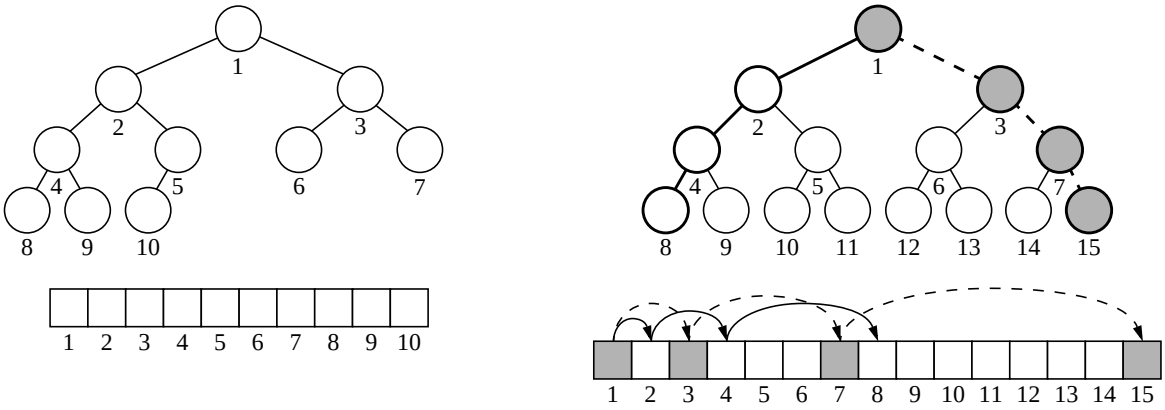
Figur 7. Perfekt, komplett och fullt träd.

Perfekta träd är uppenbarligen speciella men de är till exempel användbara för att analysera träds egenskaper. För ett perfekt träd gäller följande:

- ett perfekt binärt träd med höjd h 0 har $2^{h+1}-1$ noder.
- ett perfekt binärt träd med höjd h har 2^h löv (om antalet noder är n, är $\lfloor n/2 \rfloor + 1$ av dessa löv, dvs finns på djupet h).

Antalet fulla noder i ett binärt träd är antalet löv minus 1.

Kompletta träd är speciella men har praktiska tillämpningar. Genom att numrera noderna nivåvis kan de avbildas på ett fält, vilket exempelvis kan utnyttjas för att implementera en *binär heap*. I figur 8 visas denna avbildning från ett binärt komplett träd till ett fält.



Figur 8. Kompletta träd och deras fältimplementering.

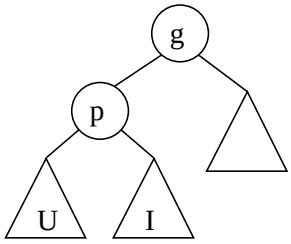
Nedtill i det högra exemplet i figur 8 visas två vägar i det kompletta trädet, en med heldragna pilar och en med streckade pilar, och de element i fältet som motsvarar de noder som ingår i de två vägarna. Att lagra ett träd i ett fält kallas *implicit representation*.

Följande samband mellan trädnoderna i ett komplett binärt träd med *n* noder och fältindex gäller (om fältindex ska starta på 0 får formlerna justeras).

- rotnoden har index 1 och sista lövet har index *n*.
- barnen till den *i*:te noden har index 2*i* respektive 2*i*+1.
- föräldern till den *i*:te noden har index *i*/2 (heltalsdivision).
- sista icke-lövet har index *n*/2.

Det är lätt att med enkel aritmetik på index förflytta sig i ett fältlagrat komplett träd, både nivåledes och i djupled, uppåt eller nedåt, och det kan generaliseras till träd av godtycklig grad.

I vissa binära träd, exempelvis sökträd, används begreppen *utsidesnod* respektive *insidesnod*. Detta avser var en viss nod finns relativt en annan nod (anfader). I trädet i figur 9 är alla noder i subträdet U *utsidesnoder relativt g*, medan alla noder i subträdet I är *insidesnoder relativt g*.

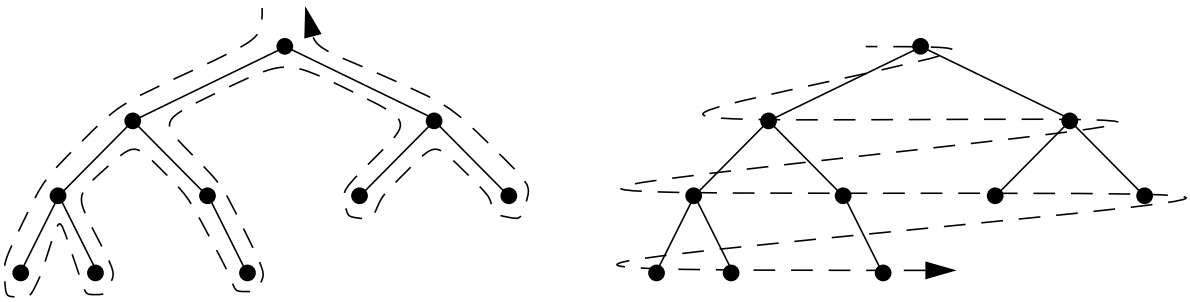


Figur 9. Insides- respektive utsidesnoder.

Den form ett visst träd har beror av vilken typ av träd det är och ibland av hur det har byggts upp. För ett uttryckskträd bestäms formen av det uttryck som trädet representerar, för ett sökträd påverkas formen dels av vilken typ av sökträd det är (till exempel AVL-träd eller splayträd) dels av den ordning som elementen sätts in i trädet.

2.2 Trädtraversering

Att traversera ett träd innebär att besöka alla noder i trädet på ett systematiskt sätt. Man brukar skilja mellan två grundläggande sätt att traversera ett träd, *djupet först* respektive *bredden först (nivåvis)*. I båda fallen kan man dessutom tänka sig att traversera från vänster till höger eller från höger till vänster.



a) djupet först, vänster till höger b) bredden först, vänster till höger

Figur 10. Trädtraversering.

I fallet djupet först kommer varje nod att besökas tre gånger och det kan anses gälla även för löv om man ser det som om deras tomma subträd besöks. En djupet-först-traversering från vänster till höger besöks en nod första gången då traverseringen kommer från dess förälder. Sedan besöks nodens vänstra subträd och därefter noden igen. Efter det besöks nodens högra subträd och återigen noden. Sedan åter till föräldern. En operation som ska utföras på noden kan alltså utföras när noden besöks första gången, eller efter att vänster subträd har besökts, eller efter att även höger subträd har besökts. Dessa tre varianter betecknas *preorder*, *inorder*- respektive *postordertraversering*, eller prefix-, infix- och postfixtraversering. Inordertraversering kan implementeras rekursivt enligt följande:

```
Om trädet inte är tomt:
    besök vänster subträd (rekursionen tar oss sedan tillbaka till föräldern)
    operera på den aktuella noden
    besök höger subträd (rekursionen tar oss sedan tillbaka till föräldern)
```

Om man av någon anledning inte kan använda rekursion kan djupet-först-traversering med viss möda implementeras iterativt med hjälp av en stack. I vissa träd har man en ”extra” båge som leder till föräldranoden och sådana träd kan traverseras iterativt med hjälp av denna extra båge.

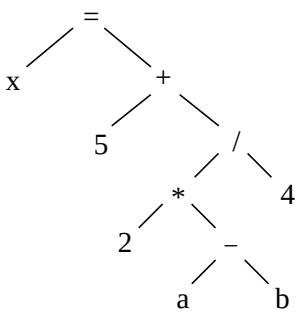
Bredden-först-traversering kan implementeras iterativt med hjälp av en kö, enligt följande:

```
placera (en pekaren till) rotnoden i kön
så länge inte kön är tom:
    hämta nästa nod (pekare) från kön
    placera (pekare till) den aktuella nodens vänsterbarn i kön
    placera (pekare till) den aktuella nodens högerbarn i kön
    operera på den aktuella noden
```

2.3 Uttryckträd

I till exempel kompilatorer används uttryckträd som intern representation av uttryck. Uttrycket x = 5 + 2 * (a - b) / 4 motsvaras av uttryckträdet i figur 11.

Ett uttryck har en hierarkiskt struktur som bestäms av operatorernas prioritet och associativitet samt parenteser. Strukturen framhävs om man parentetriserar ett uttryck fullt ut, vilket för vårt exempel innebär (x = (5 + ((2 * (a – b)) / 4))). Man ser nu tydligt att deluttrycket a–b ska beräknas innan dess värde multipliceras med 2, varefter det resultatet ska divideras med 4, osv. Ett uttryckträds struktur avspeglar entydigt beräkningsordningen för uttrycket, utan att parenteser behöver finnas i uttryckträd. Alla operander är löv och alla operatorer är inre noder.



*Figur 11. Uttryckträd för uttrycket x = 5 + 2 * (a - b) / 4.*

Om vi traverserar uttryckträdet i figur 11 erhåller vi följande resultat.

preorder: `= x + 5 / * 2 - a b 4`
inorder: `x = 5 + 2 * a - b / 4`
postorder: `x 5 2 a b - * 4 / + =`

Resultatet av inordertraverseringen känner vi igen som det givna infixuttrycket, utan parenteserna kring a-b. Ett beräkningsmässigt korrekt infixuttryck går att återskapa genom att införa parenteser kring varje deluttryck (delträd).

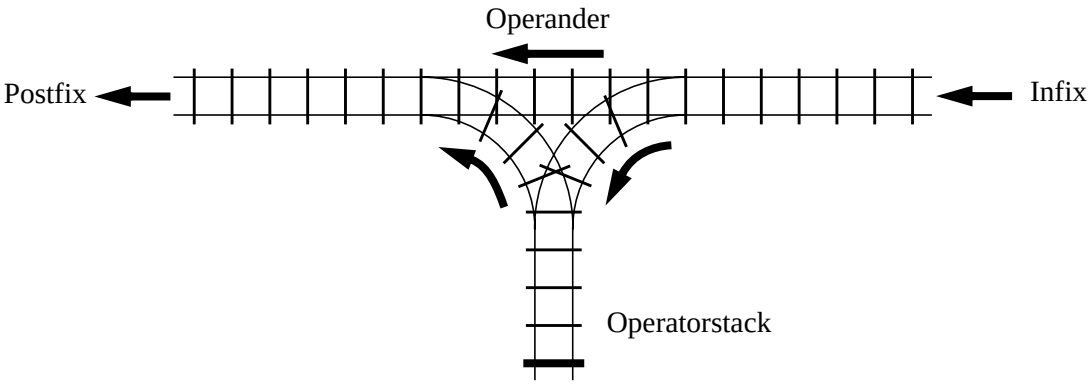
Preorder- och postorderresultaten kallas *polsk notation* respektive *omvänd polsk notation*. I polsk notation kommer operatorerna före sina operander, till exempel kommer - före a och b, * före 2 och deluttrycket a–b, osv. I omvänd polsk notation är det tvärt om, a och b kommer före –, 2 och a–b kommer före *, osv. Dessa notationer kallas även parentesfria eftersom inga parenteser behövs, beräkningsordningen är ändå entydig.

Omvänd polsk notation är mycket användbar. Till exempel kan den ha använts som en mellanrepresentation för att generera uttryckträdet ur det givna infixuttrycket. Omvänd polsk notation gör det också enkelt för ett program att beräkna ett uttrycks värde. Genom att operanderna alltid kommer alltid före den motsvarande operatorn kan man kan läsa ett uttryck från vänster till höger och successivt beräkna deluttryck utan att känna till vad som kommer längre fram.

Generella uttryckträd är naturligtvis betydligt mer komplicerade än det exempel som visas här.

2.3.1 Generering av uttryckträd

Ett sätt att generera det uttryckträd som motsvarar ett visst infixuttryck är att först omvandla till motsvarande postfixnotation. Omvandlingen från infix- till postfixnotation kan göras med hjälp av *järnvägsalgoritmen*, som använder en stack för att mellanlagra operatorer.



Figur 12. Principen för järnvägsalgoritmen.

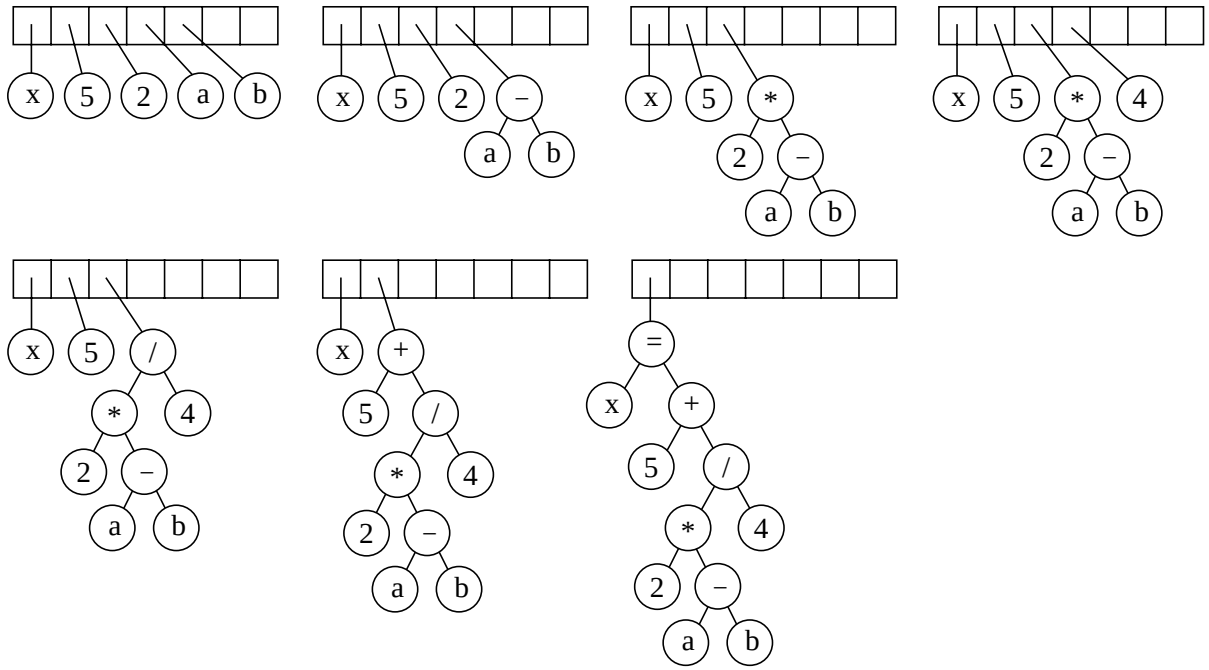
Namnet järnvägsalgoritmen kommer av man kan se förloppet som om infixuttryckets operander och operatorer körs genom en enkel rangerbangård. Bangården har ett inkommande spår och ett utgående spår. Stacken fungerar som ett stickspår, på vilket operatorer körs in från inkommande spår för att senare köras ut på utgående spår.

En operand körs alltid direkt till utspåret och backas aldrig tillbaka, vilket innebär att operandernas inbördes ordning är densamma i utdata som i indata. En operator körs alltid in på stickspåret men operatorer som redan finns på stickspåret och som har högre prioritet ska först köras till utspåret. Om en operator på stickspåret har samma prioritet som inkommande operator avgör associativiteten för den aktuella prioritetsnivån vad som ska ske. Om vänsterassociativitet gäller ska operatorn på stickspåret köras till utspåret, annars inte. Operatorer med lägre prioritet ska alltid stå kvar på stickspåret. Infixuttrycket x = 5 + 2 * (a – b) / 4 kommer att behandlas enligt figur 13, där alltså en stack utgör stickspåret.

Inkommande	**Stack**	**Utgående**
x = 5 + 2 * (a – b) / 4		
= 5 + 2 * (a – b) / 4		x
5 + 2 * (a – b) / 4	=	x
+ 2 * (a – b) / 4	=	x 5
2 * (a – b) / 4	= +	x 5
* (a – b) / 4	= +	x 5 2
(a – b) / 4	= + *	x 5 2
a – b) / 4	= + * (	x 5 2
– b) / 4	= + * (	x 5 2 a
b) / 4	= + * (–	x 5 2 a
) / 4	= + * (–	x 5 2 a b
/ 4	= + *	x 5 2 a b –
4	= + /	x 5 2 a b – *
	= + /	x 5 2 a b – * 4
		x 5 2 a b – * 4 / + =

*Figur 13. Järnvägsalgoritmen applicerad på infixuttrycket x = 5 + 2 * (a - b) / 4.*

Med hjälp av en stack kan ett uttryckträd genereras ur omvänd polska notation. Algoritmen är enkel. För en operand skapas en ny nod och en pekare till noden placeras på stacken. För en binär operator skapas först en ny nod, den översta noden på stacken görs till dess högerbarn, nästa nod från stacken görs till dess vänsterbarn och pekaren till den nya operatornoden placeras på stacken. Uttrycket x 5 2 a b - * 4 / + = kommer att behandlas enligt figur 14.

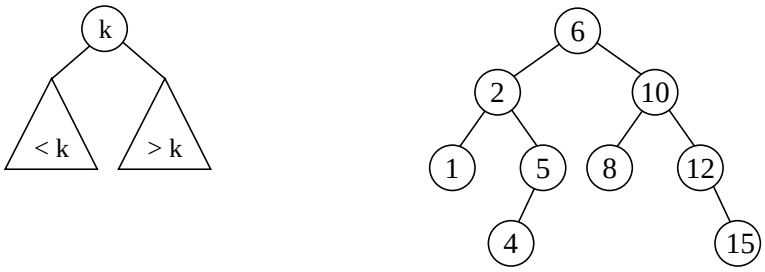


Figur 14. Generering av uttryckträd ur postfixuttrycket x 5 2 a b - 4 / + =.

2.4 Sökträd

Genom att systematiskt ordna data i en trädstruktur kan effektiva sökstrukturer erhållas. Det finns två huvudkategorier av sökträd, binära sökträd respektive mångförgrenade sökträd.

Binära sökträd är binära träd där elementen är ordnade, se figur 15. Ett enkelt binärt sökträdet har inga andra krav än att vara binärt och ordnat. Avancerade sökträd har ofta krav på trädstrukturen, till exempel har AVL-träd och röd-svarta träd ett höjdbalanskrav som medför att de ej kan bli allt för osymmetriska. Binära träd används ofta för intern datalagring, dvs för lagring i datorns arbetsminne under programkörning.



Figur 15. Binärt sökträd.

Mångförgrenade sökträd är träd med hög grad. Sådana används vanligtvis för extern datalagring, dvs lagring på fil på sekundärminne. B-träd är ett exempel på denna typ av sökträd och graden är typiskt hög, kanske 100-400. Högt gradtal i kombination med balanskrav ger litet djup, vilket innebär att antalet läsningar/skrivningar av sekundärminne när man behöver hämta data från sekundärminne eller skriva data till sekundärminne kan hållas litet.

Data i ett binärt sökträd kan antingen lagras i trädnoderna eller i en separat datastruktur som då refereras från trädnoderna. I ett B-träd skiljer sig de inre noderna helt från lövnoderna. I lövnoderna i ett B-träd lagras den egentliga informationen, medan de inre noderna enbart innehåller utvalda söknycklar och referenser till subträd. Den del av ett B-träd som utgörs av de inre noderna kan ses som en indexstruktur för löven, där alla data lagras.

2.4.1 Binära sökträd

Ett binärt sökträd är ett binärt träd där för söknyckeln (k) i en nod är större än nycklarna i nodens vänstra subträdet och mindre än nycklarna i det högra subträdet, se det schematiska trädet till vänster i figur 15. I vissa sökträd tillåts lika nycklar för olika noder. För ett *enkelt binärt sökträd* finns inga andra krav än denna *sökträdsordning*. Några observationer:

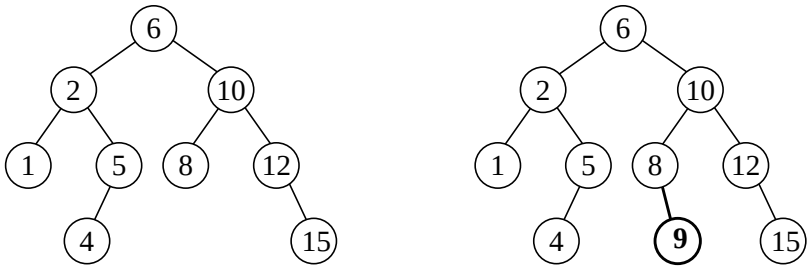
- Den minsta söknyckeln finner vi längst ner till vänster i trädet och den hittas genom att i varje nod följa den vänstra grenen, till dess vi når en nod som saknar vänsterbarn (1).
- Den största söknyckeln finner vi längst ner till höger i trädet och det hittas genom att i varje nod följa den högra grenen, till dess vi når en nod som saknar högerbarn (15).
- Om vi gör en *inordertraversering* (från vänster till höger) erhåller vi elementen i stigande nyckelordning.
- Den nod som besöks direkt före en viss annan nod vid inordertraversering kallas *inorder föregångare*. Inorder föregångaren till en inre nod hittar vi som den högraste noden i den aktuella nodens vänstra subträd. Till exempel är 5 inorder föregångare till 6.
- Den nod som besöks direkt efter en viss annan nod vid inordertraversering kallas *inorder efterföljare*. Inorder efterföljare till en inre nod hittar vi som den vänstraste noden i den aktuella nodens högra subträd. Till exempel är 8 inorder efterföljare till 6.
- Inorder föregångare och inorder efterföljare till en inre nod är noder som antingen är ett löv eller en inre nod med endast ett barn. Detta tillsammans med att det är lätt att hitta dessa noder gör det enkelt att ta bort element ur ett binärt sökträd.

Sökning är en operation som används i sig, för att återfinna ett element med en viss söknyckel, men också som ett första steg i samband med insättning och borttagning. Sökning i ett binärt sökträd kan beskrivas med följande rekursiva algoritm:

```
om trädet är tomt
        sökt värde finns ej i trädet
annars, om sökt värde < elementet i aktuell nod
        sök elementet i vänster subträd
annars, om sökt värde > elementet i aktuell nod
        sök elementet i höger subträd
annars
        elementet finns i den aktuella noden
```

Sökningen i ett någorlunda symmetriskt binärt sökträd är effektiv. För varje nod som undersöks kan det ena av dess två subträd elimineras, Det innebär att sökmängden i varje steg i princip halveras, om trädet är välbalanserat, och sökningen blir effektivitetsmässigt jämförbar med binärsökning.

Vid insättning utförs först en trädsökning enligt ovan. Om den resulterar i att det sökta värde redan finns i trädet är det ett fel om endast unika nycklar tillåts. I annat fall ska en nod med det nya elementet sättas in på den plats dit sökningen ledde. Vid insättning av 9 i det vänstra trädet i figur 16 leder sökningen till det tomma högra subträdet till nod 8 och där ska då sätts en ny lövnod med 9 sätts in. Ett nytt värde sätts alltid in som ett löv och om det inte finns något balanskrav är insättningen klar.



Figur 16. Insättning i binärt sökträd.

Borttagning i ett binärt sökträd är mer komplicerat än insättning. Första steget är att göra en trädsökning. Om sökningen leder till ett tomt (sub)träd finns inte elementet som ska tas bort i trädet. I annat fall stannar sökningen i någon nod och tre fal kan särskiljas:

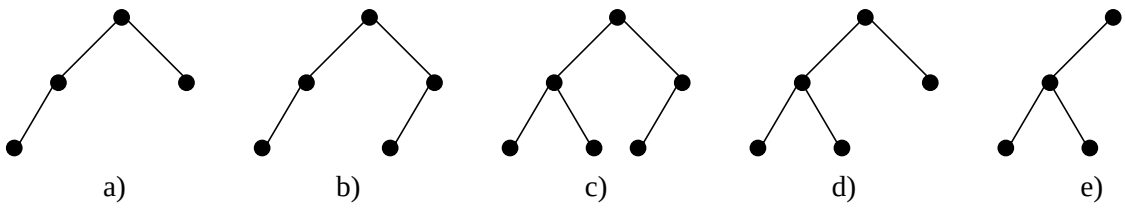
1. Elementet som ska tas bort finns i ett löv. Lövet kan tas bort utan några andra åtgärder. Borttagning av 4 i något av träden i figur 16 föranleder ingen mer åtgärd än att noden tas bort.
2. Elementet som ska tas bort finns i en inre nod med bara ett barn. Detta är också ett enkelt fall, eftersom noden som ska tas bort kan ersättas av sitt enda barn. Borttagning av 5 i något av träden i figur 16 innebär att noden med 5 tas bort och ersätts med dess vänstra subträd, dvs noden med 4.
3. Elementet finns i en inre nod med två barn. Vi kan inte utan vidare ta bort noden, eftersom den har två subträd som måste tas om hand. Lösningen är att söka upp inorder efterföljare och låta den ersätta elementet som ska tas bort. Detta bibehåller ordningen i trädet och noden i fråga är alltid ett löv eller en inre nod med endast ett barn, vilket underlättar uppflyttningen. Trädstrukturen påverkas minimalt. Alternativt kan inorder föregångare väljas som ersättare. Borttagning av 6 i trädet till vänster i figur 16, kan göras genom att inorder efterföljare, 8, söks upp och får ersätta 6, noden där 8 fanns kan enkelt tas bort och högerbarnet 9 placeras på dess plats. Alternativt kan inorder föregångare, 5, sökas upp och ersätta 6.

Om de element man sätter in i ett binärt sökträd är slumpmässigt ordande kommer trädet vanligtvis att få en tämligen symmetrisk form och därmed få goda sökegenskaper. I annat fall, eller om man vill garantera bra sökegenskaper, använder man ett balanserat sökträd.

2.4.2 Balanserade binära sökträd

För att ett binärt sökträd ska vara effektivt att söka i krävs att trädet är *balanserat*, dvs någorlunda symmetriskt. Ett välbalanserat träd har ett djup som är proportionell mot logaritmen av antalet element.

En strikt form av balans för binära träd är så kallad *nodbalans*: För *varje* nod ska gälla att *antalet noder* i dess vänstra subträd och antalet noder i dess högra subträd får skilja med högst en nod. I figur 17 är träden a, b och c exempel på nodbalanserade träd med djup 2. Träden d och e är ej nodbalanserade, eftersom rotnodens vänstra subträd innehåller tre noder, medan de högra endast en respektive ingen nod och skillnaden mellan antalet noder i rotnodens subträd alltså är större än en nod.

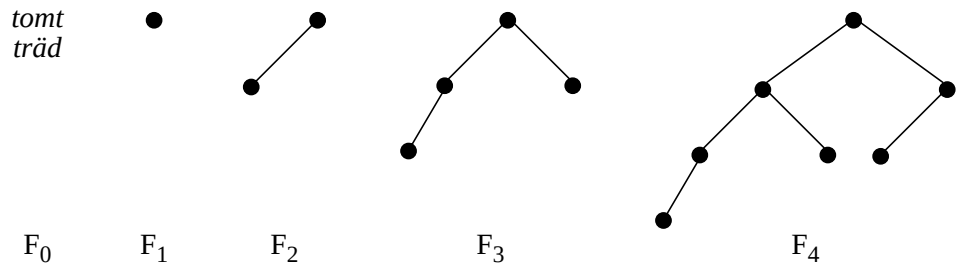


Figur 17. Olika balanserade och obalanserade träd med höjd 2.

Nodbalans är ej praktiskt användbart eftersom arbetet att hålla ett träd balanserat blir större än de vinster man kan erhålla i samband med exempelvis sökning i trädet.

Ett mindre strikt balansvillkor, dessutom är praktiskt användbart, är så kallad *höjdbalans*: För *varje* nod ska gälla att höjden hos dess vänstra subträd och *höjden* hos dess högra subträd får skilja med högst ett. Träden a-d i figur 17 är höjdbalanserade. Träd e är ej höjdbalanserat därför att rotnodens vänstra subträd har höjd 1, det högra har höjd -1, vilket ger höjdskillnaden 2. AVL-trädet är ett höjdbalanserat binärt sökträd.

Fibonacciträd kallas en form av konstruerade binära träd som representerar träd som uppfyller definitionen på höjbalans med minimalt antal noder för ett visst djup. I figur 18 visas de fem första Fibonacciträden.



Figur 18. Fibonacciträd.

Fibonacciträden F_0 och F_1 är givna, F0 är det tomma trädet, F1 är trädet med en enda nod. Fibonacciträden definieras enlig följande:

F_0 är det tomma trädet

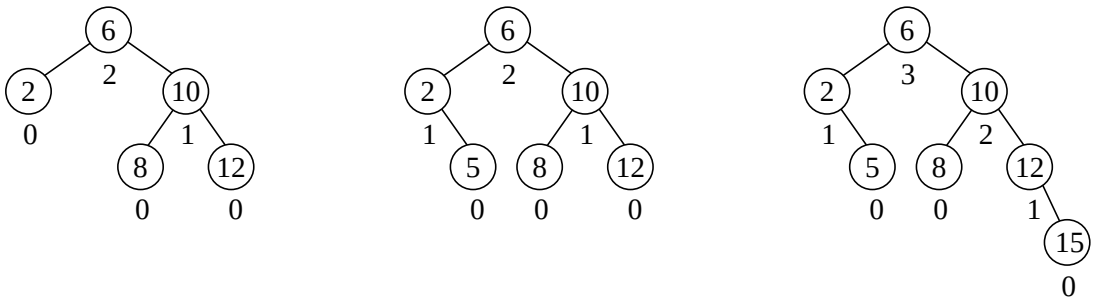
F_1 är trädet med en nod

$F_n = F_{n-1} + \bullet + F_{n-2}$ (låt F_{n-1} vara vänster subträd och F_{n-2} höger subträd under en ny rotnod)

Med hjälp av Fibonacciträd kan man visa att ett höjdbalanserat träd maximalt är 44% djupare än ett perfekt balanserade träd med samma antal noder och garanterat har logaritmisk tidskomplexitet.

2.4.3 AVL-träd

Ett AVL-träd är ett *höjdbalanserat binär sökträd*. Det tillhör också en kategori av träd som kallas *självbalanserande*, eftersom det ingår i insättnings- och borttagningsoperationerna att obalanser som uppkommer justeras. För att göra balanskontrollen enkel lagras i varje nod dess höjd, se trädet till vänster i figur 19.



Figur 19. AVL-träd.

Insättning och borttagning i ett AVL-träd görs inledningsvis som i ett enkelt binärt sökträdet. Vid insättning görs först en trädsökning och sedan sätts ett löv med det nya elementet in i uppsökt

position. I AVL-trädet vandrar man sedan upp till roten baklänges utmed sökvägen och eventuell obalans justeras. Vid borttagning söks noden som ska tas bort upp och sedan utförs borttagning enligt något av de tre fall som beskrivits för borttagning i ett enkelt binärt sökträd. Då ett värde tas bort kommer en nod att tas bort någonstans. Från och med dess förälder och på vägen upp till roten kontrolleras balansen för varje nod och eventuell obalans justeras.

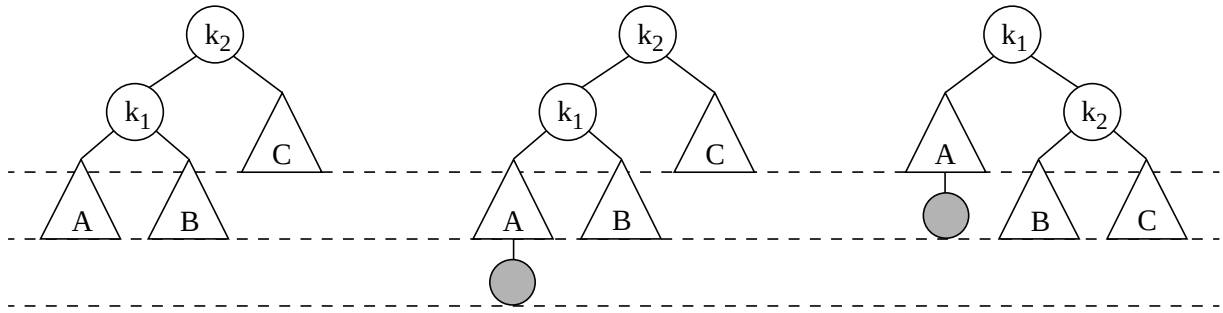
Balanskontroll i ett AVL-träd innebär följande, ör varje nod på vägen upp till rotnoden:

- Om höjdskillnaden mellan nodens vänstra och högra subträd är mindre än eller lika med ett är noden i balans. Höjden som lagras i noden justeras till maxvärdet av subträdens höjder plus 1.
- Om höjdskillnaden är lika med 2 är inte noden i balans. Det åtgärdas genom att en så kallad nodrotation utförs som återställer balansen. Höjderna i de inblandade noderna justeras.

Vid insättning av ett värde i ett AVL-träd kommer högst en obalans att uppstå, vid borttagning kan flera obalanser uppstå.

2.4.3.1 AVL-trädsrotationer

Det finns fyra AVL-trädsrotationer, enkelrotation med vänster barn (EV), enkelrotation med höger barn (EH), dubbelrotation med vänster barn (DV) och dubbelrotation med höger barn (DH). I figur 20 visas EV, allt relaterat till den nod där man upptäckt obalans.

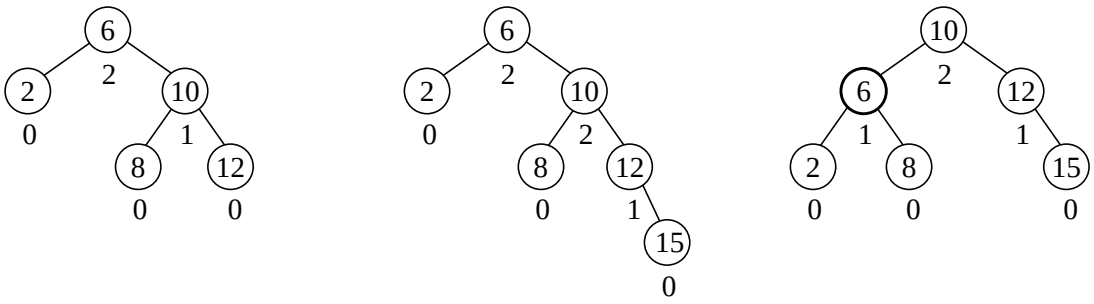


Figur 20. Enkelrotation med vänster barn (EV) i ett AVL-träd.

Trädet till vänster i figur 20 visar ett delträd i ett AVL-träd i balans. Noden med nyckel k_2 är en godtycklig inre nod. Nyckeln k_1 är mindre än k_2. De symboliserade träden A, B och C är subträd med lika höjd. De är godtyckligt stora och kan vara tomma. Nycklarna i träd A är mindre än k_1, nycklarna i träd B är större än k_1 och mindre än k_2 och nycklarna i träd C är större än k_2.

I mitträdet i figur 20 har en nod med ett element med nyckel mindre än k_1 satts in, vilket medfört att trädet A har ökat i djup med 1. Obalans uppstår då i nod k_2, eftersom dess vänstra subträd nu är två nivåer högre är dess högra subträd C. Balansen återställs med EV: vänster barn till obalansnoden (k_1) flyttas upp, obalansnoden (k_2) blir dess högerbarn, och subträdet B blir vänsterbarn till den nedflyttade obalansnoden. Det högra trädet i figur 20 visar resultatet efter rotation. Av det framgår nycklarna att den ursprungliga höjden i trädet återställs och därmed kan ingen mer obalans uppstå högre upp i trädet. EH är en spegelbild av EV.

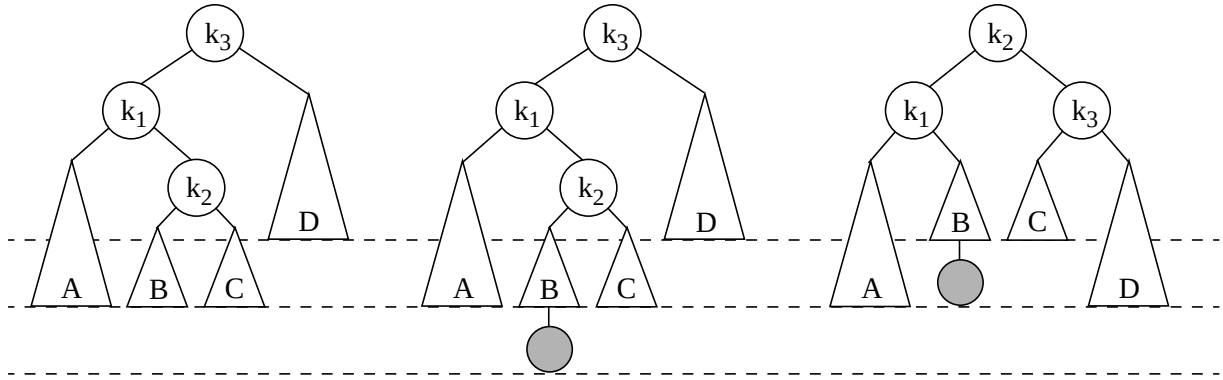
Efter att själva rotationen utförts justeras höjderna i de noder som behöver justeras. Det enda av subträden A, B och C vars höjd påverkats är det subträd där den nya noden satts in och dess höjd är redan justerad då obalansen upptäcks (det var den justeringen som medförde obalansen). Återstår de två noder som roterats, k_1 och k_2. Först justeras höjden i den nedroterade noden k_2 till maxvärdet av dess subträds höjder plus 1. Därefter justeras den upproterade noden k_1 på samma sätt. Det senare behöver egentligen inte göras, såvida inte implementeringen av enkelrotation används för att implementera dubbelrotation. Ett exempel på enkelrotation med höger barn (EH) visas i figur 21.



Figur 21. Insättning av 15 ger obalans i 6 som justeras med EH.

I det vänstra trädet i figur 21 sätts 15 in i nytt löv till höger om 12. Därefter ska höjderna i noderna på vägen upptill roten justeras och eventuell obalans justeras. I noderna 12 och 10 uppstår ingen obalans och endast höjderna justeras. I nod 6 har vänster subträd höjd 0 och höger subträd har fått höjd 2, en skillnad på 2 och alltså obalans. Den första åtgärden är att fastställa vilken typ av rotation som ska göras. Insättning har skett i höger subträd och där till höger om högerbarnet (10), vilket innebär att EH ska utföras. Efter att själva rotationen utförts justeras först höjden i den nedroterade nod 6 (till 1) och sedan höjden i den upproterade nod 10 (till 2).

Om värdet som sattes in i trädet i figur 20 i stället hade hamnat i subträdet B och det ökat i höjd ser vi att EV inte fungerar. Den skuggade noden skulle sitta under B och efter rotation skulle höjdskillnaden mellan A och B vara 2 och noden k_1 skulle vara i obalans. För att hantera en obalans av detta slag krävs så kallad dubbelrotation, se figur 22.

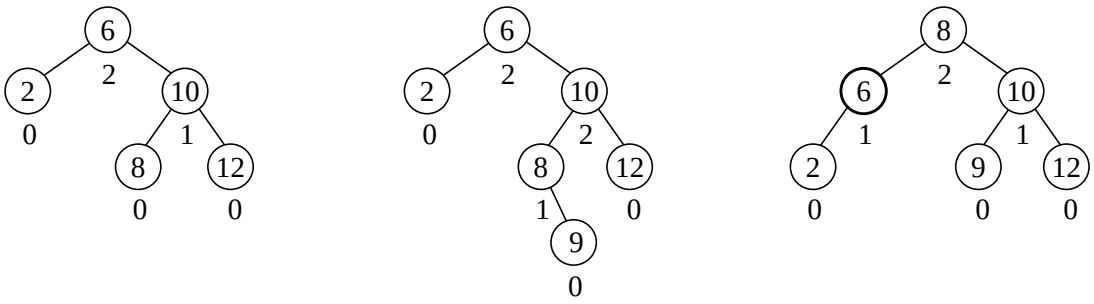


Figur 22. Dubbelrotation med vänster barn (DV) i ett AVL-träd.

Trädet till vänster i figur 22 är ett delträd i ett AVL-träd i balans. Noden k_3 är en godtycklig inre nod, k_1 är dess vänsterbarn och k_2 är högerbarn till k_1; $k_1<k_2<k_3$. Subträden B och C är lika höga, subträden A och D är lika höga och en nivå högre än B och C. Ett specialfall uppstår när A och D är tomma träd – k_2 är då den nod som sätts in (den skuggade i mitträdet).

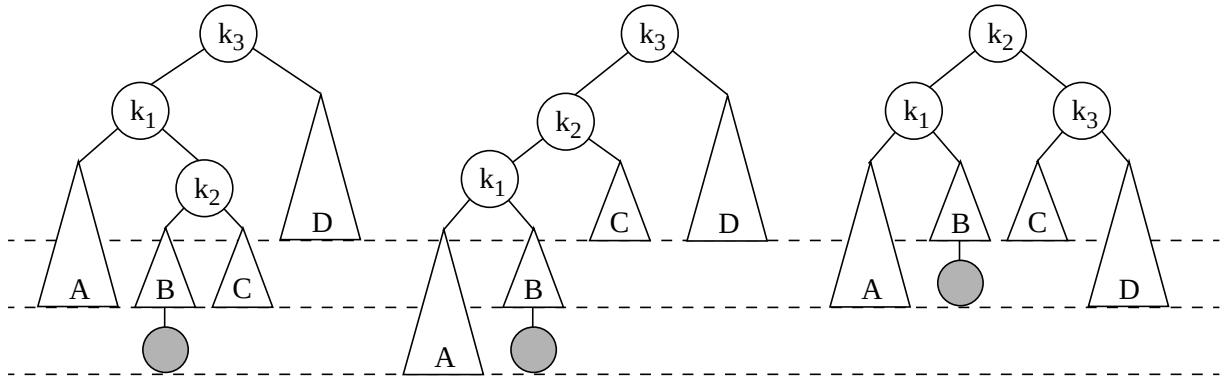
I mitträdet i figur 22 har ett värde satts in som medför att B ökat i djup med 1. Vi får då obalans i noden k_3, eftersom höjdskillnaden mellan subträdet med k_1 som rot och D blivit 2. För att åtgärda denna obalans flyttas noden k_2 upp och görs till ny rot i trädet, noden k_1 blir dess vänsterbarn och noden k_3 dess högerbarn, B blir högerbarn till k_1 och C blir vänsterbarn till k_3. Trädet till höger i figur 22 visar att detta återställer den ursprungliga höjden och därmed balansen i trädet som helhet. Vi kan också se att det inte har någon betydelse om den nyinsatta noden hamnat i C.

Efter att själva dubbelrotationen utförts justeras först höjderna i de noder som roterats ner och sedan justeras höjden i den upproterade noden. I figur 23 visas ett konkret exempel på insättning där en dubbelrotation DH utförs.



Figur 23. Insättning av 9 ger obalans i 6 som justeras med dubbelrotation med höger barn (DH).

En dubbelrotation kan implementeras med två enkelrotationer. I så fall utförs först en enkelrotation på barnet och barnbarnet till obalansnoden. Därefter görs en enkelrotation med det nu upproterade barnbarnet och obalansnoden. Detta visas i figur 24.



Figur 24. Dubbelrotation med vänster barn (DV) utförd med två enkelrotationer (EH+EV).

2.4.4 Röd-svarta träd

Röd-svarta träd (red-black trees) är ett populärt alternativ till AVL-träd. Liksom AVL-träd har röd-svarta träd en logaritmiskt tidskomplexitet och är också ett form av självjusterande träd. Ett röd-svart träd är ett binärt sökträd med följande egenskaper:

1. Varje nod är färgad antingen röd eller svart.
2. Roten är alltid svart.
3. Om en nod är röd måste dess barn vara svarta – flera röda noder i följd får ej förekomma.
4. Varje väg från en nod till ett löv, inklusive lövet, måste innehålla lika många svarta noder.

Man har dessutom konventionen att ett tomt subträd är svart.

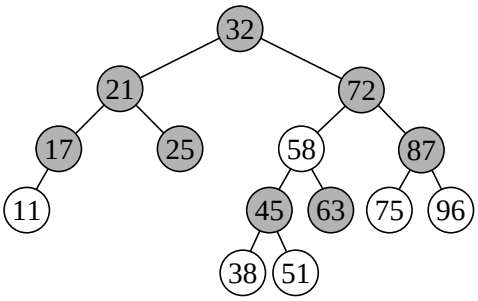
En fördel med röd-svarta träd är att de vanligtvis är snabbare än AVL-träd eftersom balansering kan göras redan vid sökningen ner genom trädet. Genom att noderna i röd-svarta träd har en referens (pekare) till föräldern är det möjligt att använda effektiva iterativa algoritmer.

Det finns två varianter av röd-svarta träd, *bottom-up* och *top-down*. Bottom-up avser att ett pass ner i trädet, för att exempelvis söka upp den lövposition där ett nytt värde ska sättas in, först utförs och sedan ett pass upp i trädet för att kontrollera och justera eventuella obalanser som uppstått. Top-down avser att justeringen av trädet gör redan på vägen ner i trädet och att inget pass uppåt därför behöver genomföras.

Genom att analysera röd-svarta träd kan man visa att, om varje väg från roten till ett tomt träd innehåller *s* svarta noder, måste det finnas minst 2^s-1 svarta noder i trädet. Vidare, eftersom roten alltid är svart och det inte kan finnas två röda noder i följd utmed en väg, är höjden i ett röd-svart träd

som mest $^2\log(n+1)$. Sökning är därmed en operation med garanterat logaritmisk tidskomplexitet. Höjden är för övrigt typiskt densamma som höjden i ett AVL-träd av motsvarande storlek.

I figur 25 visas ett röd-svart träd, som har tillkommit genom insättning av elementen 17, 87, 21, 72, 25, 58, 32, 45, 63, 75, 96, 38, 11 och 51. Svarta noder är skuggade.



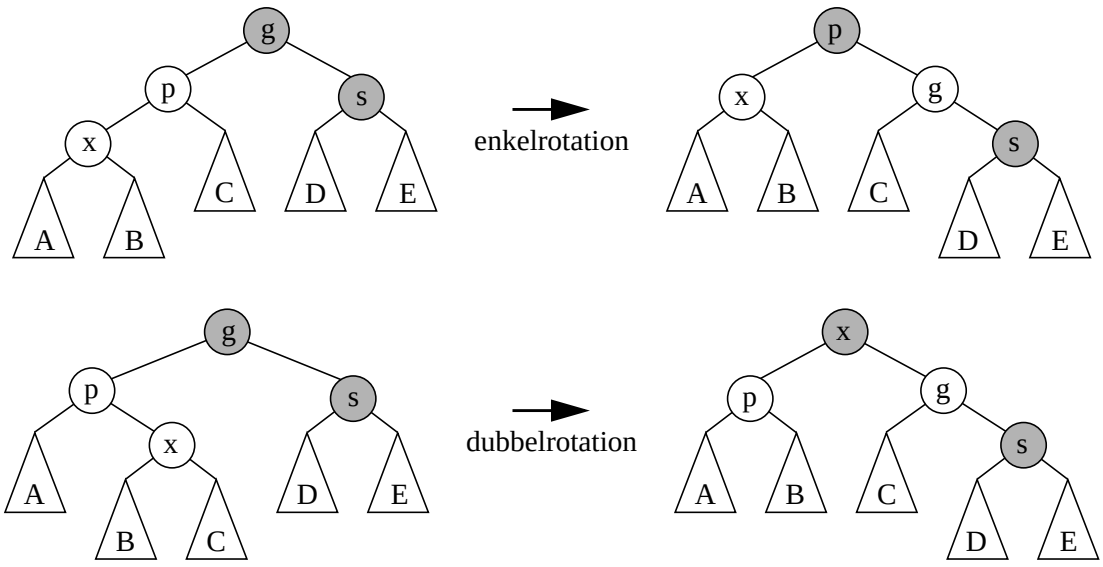
Figur 25. Ett röd-svart träd.

AA-träd är en förenklad form av röd-svart träd, vilka är enklare att implementera och kan vara ett bra val i vissa situationer där man vill ha ett balanserat träd där borttagning ska göras.

2.4.4.1 Insättning i bottom-up röd-svarta träd

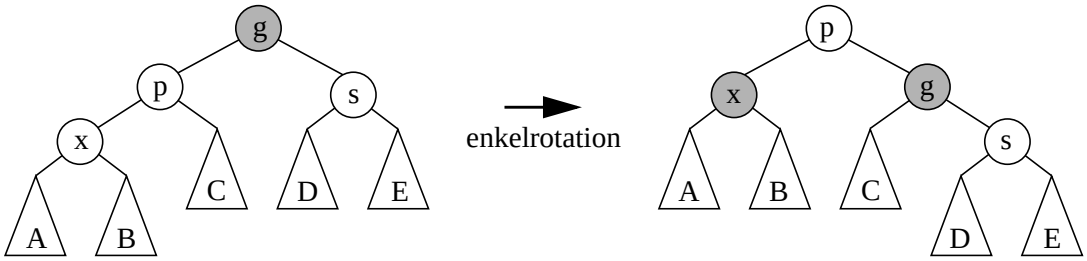
En ny nod sätts in som ett löv, alltid rött. Om den skulle färgas svart skulle det bryta mot egenskap 4 (se ovan). Om föräldern då är svart är insättningen klar. Om föräldern däremot är röd bryter detta mot egenskap 3 och då måste noder färgas om och/eller roteras.

Då föräldern till en röd nod också är röd finns det flera fall att ta hänsyn till, vart och ett med ett spegelvänt symmetrisk fall. Farföräldern är i detta fall alltid svart, annars finns det redan före insättningen två röda noder i följd. Om förälderns syskon är svart och den nya noden är ett utsides barnbarn återställs egenskap 3 med en enkelrotation och lämplig omfärgning. Om den nya noden är ett insides barnbarn återställer en dubbelrotation och lämplig omfärgning egenskap 3. Dessa två fall har generaliserats i figur 26.



Figur 26. Bottom-up-splayning då splaynodens förälders syskon är svart.

Om förälderns syskon (s) i stället är rött fungerar varken enkel- eller dubbelrotationen ovan, eftersom båda resulterar i flera röda noder i följd. I detta fall måste förälderns syskon och den nya rotnoden i subträdet samtliga färgas röda, vilket visas för enkelrotationsfallet i figur 27.



Figur 27. Bottom-up-splayning då splaynodens förälders syskon är rött.

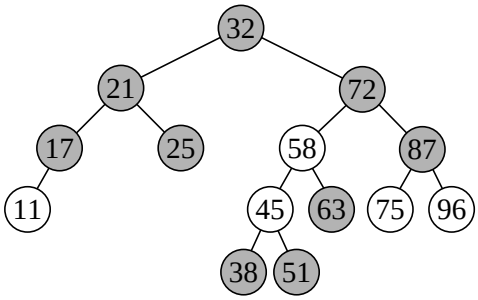
Även i dubbelrotationsfallet kommer roten i det roterade trädet att vara röd, vilket innebär problem om föräldranoden till subträdet också är röd. Man skulle kunna fortsätta att rotera uppåt till dess två röda noder i följd inte längre förekommer eller till dess roten nås och då återställs till svart. Detta innebär dock ett pass tillbaka upp i trädet, precis som i AVL-trädet, vilket man vill undvika.

2.4.4.2 Insättning i top-down röd-svarta träd

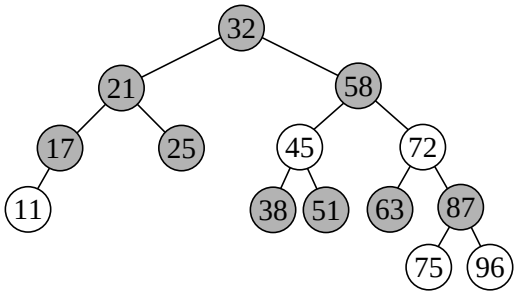
I ett top-down-splayträd utförs färgbyten och rotationer redan på vägen ner genom trädet på ett sätt som garanterar att föräldranodens syskon är svart. Detta innebär att ett rött löv kan sättas in utan ytterligare åtgärd eller med endast en enkel- eller dubbelrotation för att slutjustera trädet.

Figur 28. Omkastning av färger.

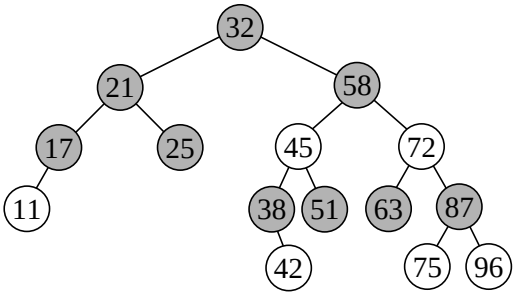
Om man på vägen ner har en nod x med två röda barn, se figur 28, färgas noden röd och barnen svarta, dvs färgerna kastas om (*color flip*). Antalet svarta noder på vägen mellan *x* och löven ändras inte men om *x*:s förälder är röd finns nu två röda noder i följd. Nu visar det sig att syskonet till *x*:s förälder inte kan vara rött och därmed kan någon av rotationerna i figur 26 användas för att justera trädet och återställa egenskap 3. Antag att 42 ska sättas in i trädet i figur 29 nedan.



a. efter att färgen på 45, 38, 51 kastats om.



b. efter enkelrotation 72-58 och färgbyte (58 och 72).



c. efter insättning av 42, som en röd nod.

Figur 29. Insättning av 42 i trädet i figur 25.

När sökningen når noden 45 är detta en nod med två röda barn. Omkastning av färgen medför att 45 blir röd, 38 och 51 svarta, se figur 29 a. Detta ger två röda noder i följd, 58 och 45. Notera att syskonet till 58 är svart. Noden 45 är en utsidesnod i förhållande till farföräldern 72, så en enkelrotation utförs mellan föräldern 58 och farföräldern 72, med tillhörande omfärgning av noder, se figur 29 b. En röd nod med 42 sätts sedan in och eftersom dess förälder är svart är insättningen klar. Om föräldern varit röd hade ytterligare en rotation behövts.

2.4.5 Splayträd

Splayträdet är ett självjusterande binärt sökträd med amorterad logaritmisk kostnad per operation. Detta innebär att en enskild operation kan vara kostsam men att en sekvens av operationer garanteras uppför sig som om varje enskild operation i sekvensen hade uppfört sig logaritmisk.

Idén bakom splayträd är att ofta efterfrågade element ska befinna sig nära roten och därmed ha en kortare sökväg än mindre efterfrågade element. I vissa söksituationer är vissa element mer efterfrågade än andra. Man brukar tala om *90/10-principen*, vilket avser att i 90 procent av sökningarna är det endast 10 procent av elementen som efterfrågas. I sådana fall kan splayträdet vara väl lämpat.

För att efterfrågade element ska befinna sig nära roten utförs *splayning* i samband med varje sökning, vilket innebär att sökt elementet flyttas (roteras) till roten. Om inte det sökta elementet finns splayas i stället något närliggande element.

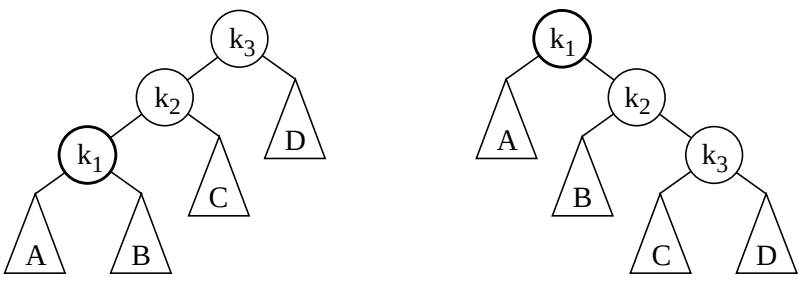
Det finns flera varianter av splayträd, bl.a. *bottom-up-splayträd* och *top-down-splayträd*. I bottom-up-fallet söks först efterfrågat element med en vanlig trädsökning och splayas sedan till roten. I top-down-fallet görs splayning redan på vägen ner och endast ett pass genom trädet behövs.

2.4.5.1 Bottom-up-splayträd

En sökning i ett bottom-up-splayträd inleds med en vanlig trädsökning. Då det sökta elementet hittats roteras noden i fråga, *splaynoden*, till roten genom en följd av dubbelrotationer, utom då splaynoden hamnat direkt under rotnoden, i vilket fall en avslutande enkelrotation utförs.

Tre slags rotationer kan förekomma i ett bottom-up-splayträd, *zig*, *zig-zag* respektive *zig-zig*, var och en i en symmetrisk vänster- och högervariant.

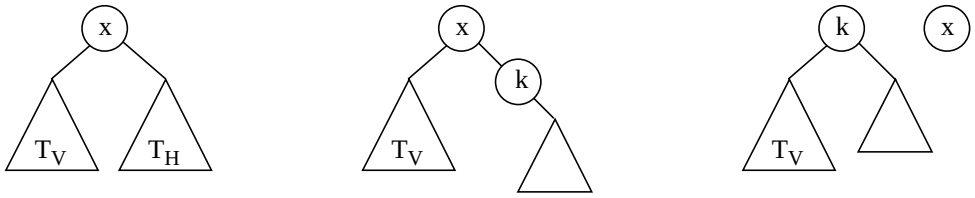
- *zig-left* och *zig-right* är enkelrotationer som endast utförs då splaynoden är vänster- respektive högerbarn till rotnoden och motsvarar enkelrotation med vänster barn respektive med höger barn i ett AVL-träd, se figur 20.
- *zig-left-zag-right* och *zig-right-zag-left* är dubbelrotationer som motsvarar dubbelrotation med vänster barn respektive dubbelrotation med höger barn i ett AVL-träd, se figur 22.
- *zig-zig-left* är en dubbelrotation som utförs då splaynoden är vänsterbarn till sin förälder och föräldern är vänsterbarn till sin förälder, se *figur 30*. *Zig-zig-right* utförs då splaynoden är högerbarn till sin förälder och föräldern är högerbarn till sin förälder. *Zig-zig*-rotationerna saknar motsvarighet i AVL-trädet.



Figur 30. Zig-zig-left i ett bottom-up-splayträd.

Insättning i ett bottom-up-splayträd görs på samma sätt som i ett enkelt binärt sökträd varefter den nya noden med det insatta elementet, en lövnod, splayas till roten.

Vid borttagning söks först det värde (x) som ska tas bort upp, vilket medför splayning till roten. I det allmänna fallet har den uppsplayade noden två subträd, T_V till vänster och T_H till höger, se det vänstra trädet i figur 31.

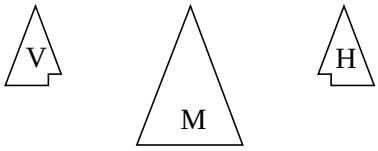


Figur 31. Borttagning i bottom-up-splayträd, efter att värdet som ska tas bort, x, sökts upp.

Efter att noden med värdet som ska tas bort splayats till roten söks antingen inorder föregångare eller inorder efterföljare upp. Det senare innebär att det minsta elementet (k) i höger subträd söks upp och splayas till roten av höger subträd, se mitträdet i figur 31. Den uppsplayade noden k saknar vänsterbarn och där kan T_V placeras. Noden med x kan nu länkas ur och noden k göras till ny rotnod, se det högra trädet i figur 31. Om värdet som ska tas bort är det största värdet kommer T_H att vara tomt och roten T_V kan direkt göras till ny rot i splayträdet då x tas bort.

2.4.5.2 Top-down-splayträd

Top-down-splayning utförs redan i samband med sökningen ner genom trädet. Trädet delas i samband med detta upp i tre delträd, ett vänsterträd, ett mitträd och ett högerträd, se figur 32.



Figur 32. Principen för uppdelning av ett top-down-splayträd vid splayning.

Från början är vänster- och högerträden tomma och mitträdet utgör splayträdet. Vid sökningen flyttas delar av mitträdet successivt till sidoträden. Vänsterträdet får ta emot delträd med element som är mindre än det sökta, högerträdet element som är större.

Rotnoden i de delträd som flyttas till vänster kommer inte att ha något höger subträd och rotnoden i de delträd som flyttas till höger kommer inte att ha något vänster subträd. Denna lediga position hålls reda på och nästa delträd som flyttas ut till den sidan kommer att placeras där. Djupet där insättning av nästa delträd görs ökar därför med enbart en nivå per utflyttning. "Hacket" i de symboliserade höger- och vänsterträden i figur 32 avser den egenskapen. Detta framgår lite mer i detalj när de olika rotationerna beskrivs nedan.

Då ett sökt elementet finns i trädet hamnar noden med detta till slut i mitträdets rot. Eventuella kvarvarande subträd flyttas därefter ut till motsvarande sidoträd och sedan flyttas vänsterträdet till rotnodens vänstra sida och högerträdet till dess högra sida. Slutresultatet är alltså ett träd där sökt värde sitter i roten.

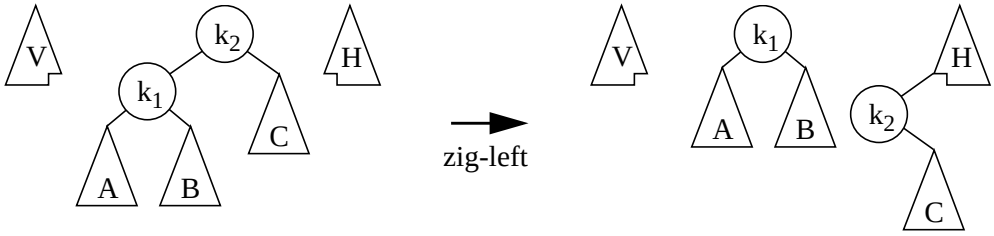
Om det sökta värdet inte finns i trädet kommer sökningen att upphöra då antingen inorder föregångare eller inorder efterföljare hamnat i mitträdets rot. Endast ett subträd till rotnoden kan finnas i detta fall (där det sökta värdet inte kan finnas). Detta flyttas till motsvarande sidoträd och sedan hängs sidoträden in under rotnoden i mitträdet.

Efter splayningen som beskrivs ovan slutförs den operation som föranledde splayningen. Om det enbart var för att ta reda på om ett värde finns i träde är det bara att kontrollera om det är detta som

nu finns i rotnoden. Om det gällde insättning kan det vara fel om värdet redan finns i trädet, i annat fall sätts en ny nod med det värdet in som ny rotnod. Vid borttagning bör värdet finnas i roten, annars kan det vara fel, och det ska då tas bort på något lämpligt sätt (se figur 38). Insättning och borttagning beskrivs lite mer ingående nedan men först top-down-splayträdets rotationer.

De splayrotationer som finns för top-down-splayträd har motsvarigheter i bottom-up-splayträd och benämns lika, dvs *zig*, *zig-zag* och *zig-zig*. Utförandet är annorlunda men effekten är i princip densamma. Så länge en dubbelrotation kan utföras så väljs det, enkelrotation i princip endast som en sista rotation i en sekvens av splayrotationer. Det finns dock *förenklad zig-zag* som innebär att endast en *zig* utförs när situationen egentligen motsvarar en *zig-zag*. Anledningen är att detta ger påtagligt enklare implementering.

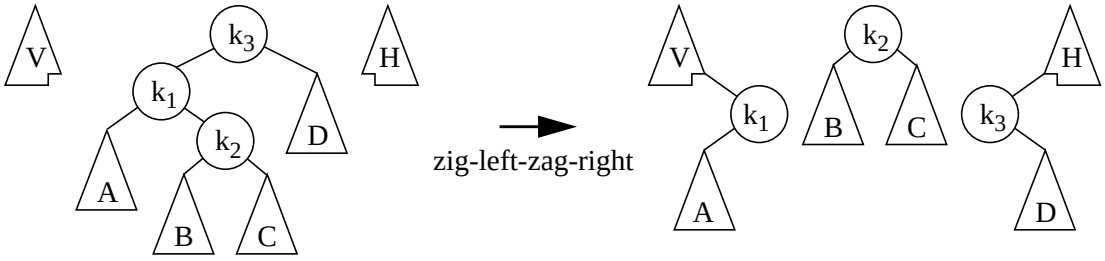
Enkelrotation *zig-left* utförs enligt figur 33. Förutsättningen för att denna ska utföras är att k_1 är det sökta elementet eller att det subträd till k_1 där det sökta elementet hör hemma är tomt, A om sökt elementet är mindre än k_1, B om större än k_1.



Figur 33. Enkelrotation zig-left.

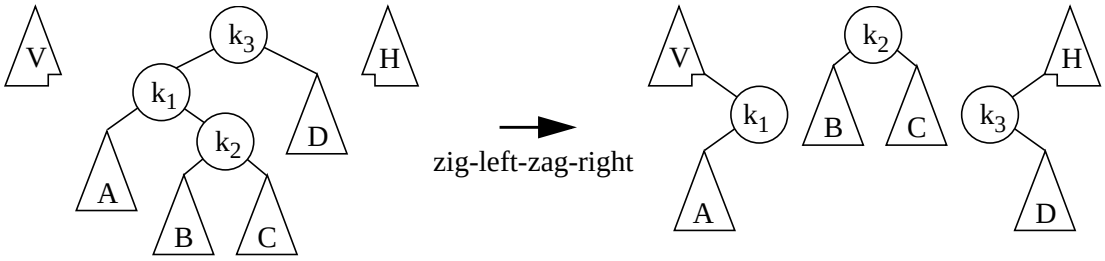
Enkelrotation *zig-left* innebär att rotnoden i mitträdet (k_2) och dess högerbarn (C) flyttas till höger sidoträd (H) och k_1 blir ny rot i mitträdet. Om k_1 var det sökta elementet kan subträden A och B båda innehålla noder eller vara tomma. Om k_1 inte var det sökta elementet måste åtminstone något av subträden A eller B vara tomt, det träd där sökt element skulle ha funnits.

Dubbelrotation *zig-zag* utförs om sökt värde hör hemma i vänster eller höger *innerträd* till mitträdets rot. I figur 34 antas sökt element finnas i subträdet med k_2 som rot. En dubbelrotation *zig-left-zag-right* utförs genom att roten i mitträdet (k_3) och dess högra barn D flyttas till höger sidoträd, vänsterbarnet (k_1) och dess vänstra subträd A flyttas till vänster sidoträd. Subträdet med k_2 som rot blir nytt mitträd. Noderna k_1 och k_3 kommer alltid att sakna höger respektive vänster subträd och det är till dessa positioner som nästa utflyttning görs.



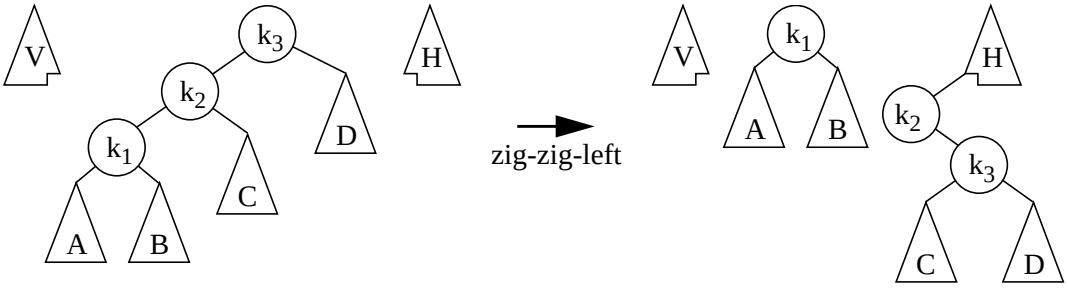
Figur 34. Dubbelrotation zig-left-zag-right.

Förenklad zig-zag-rotation innebär att man i en sådan situation endast utför en enkelrotation, i en *zig-left-zag-right*-situation endast *zig-left*. Om situationen är den som visas figur 34 utförs alltså en *zig-left* enligt figur 33. I figur 34 flyttas k_3 och D till H och k_1 blir ny rotnod i mitträdet med k_2 som högerbarn. Det extra varv som detta medför i huvudslingan för splayalgoritmen uppvägs av den förenkling av koden som erhålls genom att specifik kod för zig-zag-rotationerna elimineras. Observera att detta inte påverkar vad som ska göras i nästa varv – det beror helt på mittträdets (nya) struktur vilken slags rotation som ska väljas, om ytterligare splayning behövs.



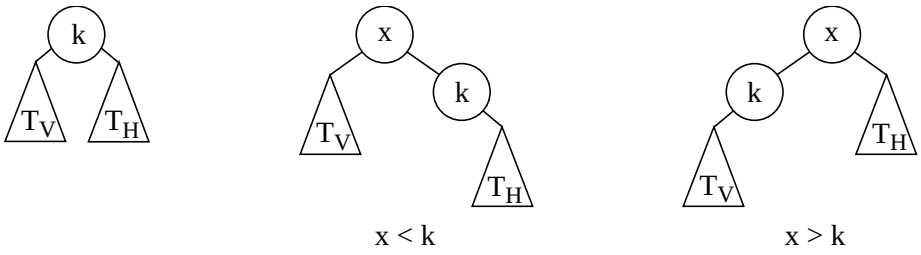
Figur 35. Förenklad zig-left-zag-right.

Dubbelrotation *zig-zig* utförs om sökt värde hör hemma i vänster eller höger *ytterträd* till mitträdets rot. I figur 36 antas sökt element finnas i subträdet med k_1 som rot. Dubbelrotation *zig-zig-left* utförs genom att enkelrotation utförs med rotnoden (k_3)och dess vänstra barn (k_2) och att detta roterade delträd flyttas till höger sidoträd. Subträdet med k_1 som rot blir nytt mitträd. Noden k_2 saknar alltid vänster subträd och nästa subträd som flyttas ut till höger sidoträd placeras där.



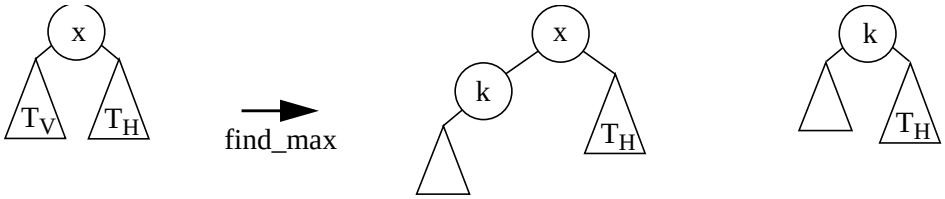
Figur 36. Dubbelrotation zig-zig-left i top-down-splayträd.

Om insättning ska göras ska en ny nod sättas in som rot i trädet noderna i det splayade trädet ska länkas på lämplig plats, se figur 37. Om det nya elementet x är mindre än k ska noden med k vara högerbarn till den nya rotnoden, se mitträdet. Om det nya elementet x är större än k ska noden med k vara vänsterbarn till den nya rotnoden, se det högra trädet.



Figur 37. Insättning i top-down-splayträd, efter inledande sökning.

Om borttagning ska göras har elementet som ska tas bort splayats till rotnoden. Sedan söks det största elementet i vänster subträd (T_V), vilket innebär att noden med det elementet (k) splayas till roten av T_v. Den noden kommer inte att ha något högerbarn och dit flyttas höger subträd (T_H). I figur 38 visas hur borttagning av ett element x görs, efter att x har splayats till roten av trädet. Alternativt kan det minsta elementet i T_h väljas för att ersätta x.



Figur 38. Borttagning ur top-down-splayträd, efter inledande sökning.

Innan man söker största värdet i vänster subträd kontrolleras att subträdet inte är tomt, vilket är fallet om det element som ska tas bort är det minsta värdet i trädet. Om vänster subträd är tomt blir höger subträd (T_H) det nya splayträdet. Om värdet som ska tas bort är det största i trädet blir T_v det nya splayträdet.

2.4.6 B-träd

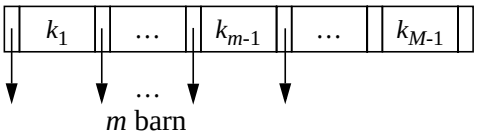
B-träd är en vanlig sökstruktur för sekundärminneslagrade data. Det finns flera varianter men gemensamt är att de är sökträd av vanligtvis hög grad, eller *ordning*, typiskt flera hundra. I kombination med att B-träden är balanserade ger detta sökträd med mycket litet djup.

Ett B-träd av ordning *M* (grad *M*) är ett träd med följande egenskaper:

- Dataposter lagras i lövnoderna.
- Lövnoderna ligger samtliga på samma djup och ska innehålla mellan $\lceil L/2 \rceil$ (gäller ej rotnod, se nedan) och *L* dataposter.
- Inre noder kan lagra upp till *M*-1 nycklar och kan ha upp till *M* motsvarande subträd. Den *i*:te nyckeln är den minsta nyckeln i subträd *i*+1 (det högra subträdet)
- Alla inre noder, utom rotnoden, ska ha mellan $\lceil M/2 \rceil$ och *M* barn.
- Rotnoden är antingen ett löv med 1-*L* dataposter eller en inre nod med mellan 2 och *M* barn.

Ett B-träd med just dessa egenskaper brukar betecknas B^+-träd. En annan variant är B^*-träd, där inre noder ska innehålla minst 2M/3 nycklar och löven minst 2L/3 element. Det som sägs nedan avser B^+-träd då det gäller sådana aspekter.

De krav som ställs på antalet barn till inre noder och antalet dataposter i löven innebär att noderna måste vara minst fyllda till hälften och därmed att trädet inte kan degenerera till motsvarande ett binärt träd. Strukturen hos en inre nod i ett B-träd av ordning *M* visas i figur 39. Det aktuella antalet barn, *m*, ska vara $\lceil M/2 \rceil \le m \le M$ och antalet nyckelvärden *m*-1. I subträdet till vänster om nyckel k_i finns poster med nycklar $k < k_i$ och i subträdet till höger finns poster med nycklar $k \ge k_i$.



Figur 39. Inre nod i ett B-träd av ordning M.

Det finns ett par speciella former av B-träd som kallas 2-3-träd och 2-3-4-träd, vilket är B-träd med ordning 3 respektive 4. Beteckningen 2-3 kan man associera med att en inre nod i ett 2-3-träd kan ha antingen 2 eller 3 barn, och beteckningen 2-3-4 att en inre nod i ett 2-3-4-träd kan ha 2, 3 eller 4 barn.

Värdena på *M* och *L* väljs så stora som möjligt med tanke på att en nod ska rymmas i ett sekundärminnesblock. En inre nod ska ha plats för *M*-1 nycklar och *M* adresser till andra block Om de aktuella nycklarna upptar 16 byte och adresserna 4 byte, går det åt 16*M*-16 byte för nycklarna och 4*M* byte för adresserna, totalt 20*M*-16 byte. Om ett sekundärminnesblock rymmer 8192 byte kan ordningen *M* maximalt vara 410. Om de aktuella dataposterna upptar 256 byte, ryms det maximalt 32 poster i ett löv.

2.4.6.1 Insättning i B-träd

Insättning i ett B-träd inleds med en trädsökning, vilken leder till den lövnod där den datapost som ska sättas in hör hemma. Om det finns plats i noden kan dataposten sättas in utan ytterligare åtgärder, i annat fall har ett *överskott* uppstått. Överskott kan hanteras på två sätt, antingen genom noddelning eller genom att försöka hitta plats i en syskonnod.

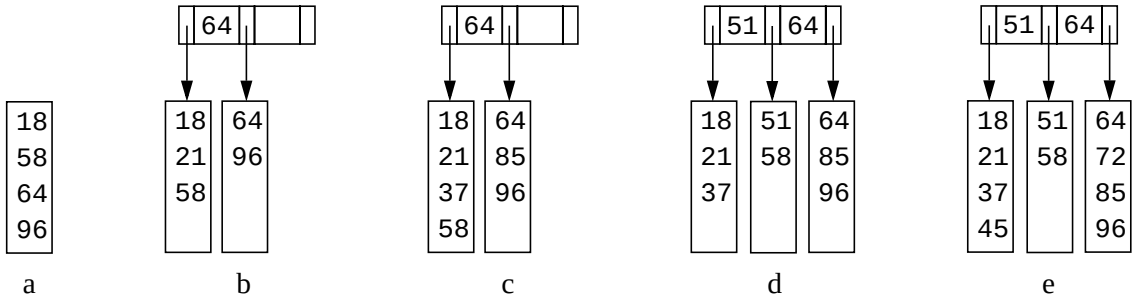
Noddelning innebär att en ny höger syskonnod skapas. Posterna, inklusive den nya posten, fördelas jämnt mellan de två noderna. Detta kallas att *balansera*, vilket måste göras för att uppfylla fyllnadsgradskravet på minst $\lceil L/2 \rceil$ poster i en lövnod. Den nya noden ska nu sättas in i föräldranoden, direkt till höger om den delade noden. Det innebär att alla nycklar och adresser i föräldranoden till höger om den delade nodens position måste flyttas ett steg åt höger för att lämna plats åt den nya nodens nyckel och adress. Om föräldranoden är full går inte detta utan antingen delas den inre noden, ett nytt högersyskon skapas och hälften av nycklarna och adresserna flyttas över dit, eller så kontrolleras om det finns plats i ett närmaste syskon ett antal nyckel-adresspar flyttas över genom balansering. Om samtliga noder i sökvägen inklusive rotnoden är fulla och noddelning alltid görs, kommer delningsförfarandet att till slut medföra att rotnoden delas. Det leder till att en ny rotnod skapas och på detta sätt ökar ett B-träd i djup. Detta förfarande gör också att alla löv kommer att finnas på samma djup, utan att det behövs några speciella åtgärder för att uppfylla det kravet.

Vid försök att hitta plats i en syskonnod inskränks detta till att endast undersöka *ett* närmaste syskon, till exempel närmast högersyskon. Om överskottsnoden råkar vara det högraste syskonet i syskonskaran, undersöks i stället närmaste vänstersyskon. Skulle även syskonnoden var full görs noddelning enligt ovan. Finns det ledigt utrymme i syskonnoden fördelas elementen i de två noderna jämnt, dvs noderna balanseras. Det skulle räcka att endast flytta över *en* post till syskonnoden, men genom att balansera undviks nytt överskott i den annars fortsatt fulla överskottsnoden redan vid nästa insättning.

Som exempel används ett B^+-träd av ordning M=3, dvs ett 2-3-träd, och vi antar att L=4. För enkelhets skull lagras heltal, vilka alltså får representera både söknycklar och elementen för övrigt. Normalt är söknyckeln bara ett bland övriga värden som ett element består av men speciellt i och med att det identifierar posten. Följande element ska sättas in:

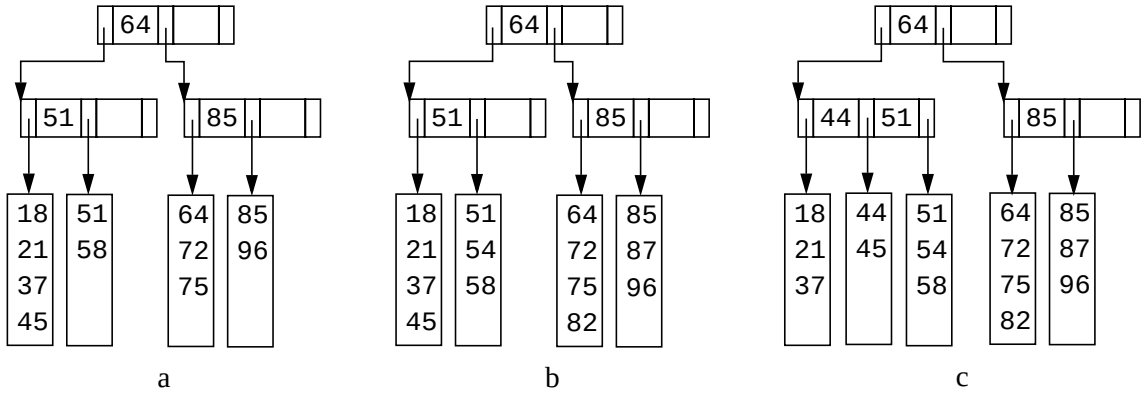
```
64 18 96 58 21 85 37 51 72 45 75 82 54 87 44
```

I figur 40 a visas trädet efter att de fyra första elementen, 64, 18, 96 och 58 satts in. Då 21 ska sättas in uppstår överskott i lövnoden och en noddelning utförs. Noddelning innebär att en ny lövnod skapas och att de fem elementen fördelas jämnt på de två noderna, i detta fall tre i den vänstra och två i den högra. Detta medför i sin tur att en inre nod måste skapas, med de två lövnoderna som barn och med den minsta nyckeln i den högre noden som utvald söknyckel. Resultatet visas i figur 40 b.



Figur 40. Insättning i B-träd av ordning 3 av: 64, 18, 96, 58, 21, 85, 37, 51, 72, 45.

I figur 40 c visas läget efter att ytterligare två element, 85 och 37, satts in. Nästa värde som ska sättas in är 51 och det ger överskott i det vänstra lövet. I detta fall skulle man kunna utnyttja att det finns plats i det högra syskonet och balansera men noddelning väljs och det resulterar i det träd som visas i figur 40 d. Elementen 72 och 45 kan sedan sättas in utan att överskott uppstår, se figur 40 e. Nästa värde, 75, ger överskott i det högraste lövet. Noddelning ger ett fjärde löv och därmed överskott i föräldranoden, dvs rotnoden. Delning av denna ger två inre noder med vardera två löv och dessutom en ny rotnod, se figur 41 a.



Figur 41. Insättning i B-träd av ordning 3, forts: 75, 82, 54, 87, 44.

I figur 41 b visas läget efter insättning av 82, 54 och 87. När till slut 44 sätts in uppstår överskott i det vänstraste lövet, vilket efter noddelning, balansering och insättning i föräldern ger trädet i figur 41 c.

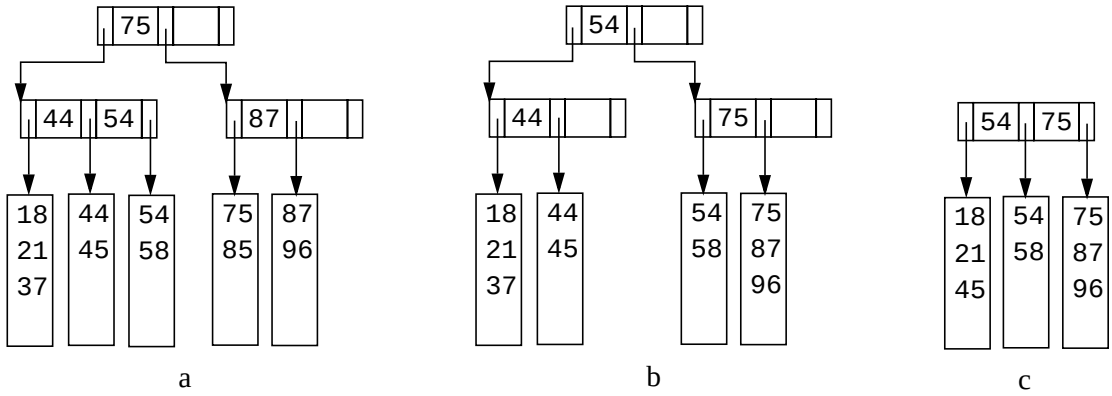
2.4.6.2 Borttagning ur B-träd

Vi borttagning ur ett B-träd söks först det löv upp där posten med den aktuella nyckeln ska finnas. Efter att posten tagits bort kontrolleras fyllnadsgraden. Om antalet återstående poster är större än eller lika med $\lceil L/2 \rceil$ är borttagningen klar, i annat fall har ett *underskott* uppstått. Vid underskott hämtas ett närmaste syskon, till exempel alltid närmaste högersyskon om sådant finns. Om underskottsnoden är den högraste noden i en syskonskara hämtas i stället närmaste vänstersyskon. Om syskonet har fler än $\lceil L/2 \rceil$ poster, *lånas* poster därifrån. I annat fall slås de två noderna ihop genom att alla poster i den högra noden flyttas till den vänstra. Den högra noden, dess motsvarande nyckelvärde och adress i föräldranoden tas bort och de nycklar och adresser som eventuellt finns till höger om denna position i föräldranoden flyttas ett steg åt vänster. Detta kan i sin tur leda till underskott i föräldranoden och då upprepas proceduren med att i första hand försöka låna från ett närmaste syskon och i andra hand sammanslagning med detta syskon. Om samtliga noder i sökvägen råkar ha precis $\lceil M/2 \rceil$ barn, om inga lån är kan göras och rotnoden endast har 2 barn, kommer sammanslagning att ske på varje nivå ända upp till rotnoden. När de två barnen till rotnoden slås ihop bildar den sammanslagna noden ny rotnod och den gamla tas bort. B-trädets djup minskar.

För att visa borttagning ur ett B-träd utgår vi från det träd som blev slutresultatet i insättningsexemplet ovan, se figur 41 c. Följande element ska i tur och ordning tas bort:

```
72 51 64 82 85 37 44
```

Borttagning av 72 medför inget annat än att elementet tas bort ur lövet. Borttagning av 51 ger inte heller något underskott men innebär att det minsta elementet i noden tas bort. Dess söknyckel finns då i en inre nod uppe i trädet, i detta fall i föräldranoden. Den ersätts med 54, den nu minsta nyckeln i lövet. Borttagning av 64 innebär också att det minsta elementet i ett löv tas bort. I detta fall hittar vi söknyckel i rotnoden, där den ersätts med 75. Borttagning av 82 leder till underskott. I detta fall innehåller syskonnoden fler än $\lceil L/2 \rceil$ element och lån kan göras. Elementen ska fördelas jämt mellan noderna, balanseras, men i detta fall flyttas endast ett värde, 85, över till underskottsnoden (upp till $L/4$ poster kan komma att flyttas, om syskonnoden är full). Vid lån kommer de minsta elementen att flyttas över, så söknyckeln i föräldranoden ändras alltid. I detta fall ersätts 85 med 87. Efter dessa fyra borttagningar ser B-trädet ut enligt figur 42 a.



Figur 42. Borttagning ur B-träd: 72, 51, 64, 82, 85, 37, 44.

Borttagning av 85 leder till underskott. Inget lån kan göras, eftersom det enda syskonet har exakt $\lceil L/2 \rceil$ element. Noderna slås ihop genom att 87 och 96 flyttas över till den vänstra noden. När sedan nyckeln 87 och den tillhörande adressen ska tas bort ur föräldranoden uppstår underskott även där. I detta fall går det att låna från syskonet, eftersom den noden har fler än $\lceil M/2 \rceil$ barn. Balanseringen innebär att det högraste lövet, det med 54 och 58, flyttas över till den högra sidan. Efter detta ser trädet ut som i figur 42 b.

Borttagning av 37 kan sedan göras utan underskott och inget annat påverkas heller. Efterföljande borttagning av 44 ger däremot underskott. Inget lån kan genomföras och löven i det vänstra subträdet slås ihop. Detta ger underskott i föräldranoden och i detta fall går det inte att låna, eftersom det högra subträdet har exakt $\lceil M/2 \rceil$ barn. De inre noderna under rotnoden slås ihop och blir ny rotnod. Trädet ser efter detta ut som i figur 42 c.

3 Hashtabeller

En hashtabell är en sökstruktur där varje nyckel avbildas på en adress i tabellen. Sökning beror då inte på hur många poster som finns i tabellen och kan därför i princip utföras i konstant tid.

I realiteten är inte situationen vanligtvis inte så idealisk. *Hashfunktionen* som avbildar nyckelvärden på adresser, så kallade *hemadresser*, kan vanligtvis inte avbilda varje nyckelvärde på en unik adress. Två eller flera nycklar kan få samma hemadress, vilket kallas *kollision*. Då kollision inträffar måste en alternativ plats hittas för den post som inte kan lagras på sin hemadress och för det används en *kollisionshanteringsmetod*.

Dimensionering av hashtabeller är i allmänhet viktigt. Hashfunktioner kan kräva specifika tabellstorlekar för att fungera bra, till exempel att storleken ska vara ett primtal. För flera av de kollisionshanteringsmetoder som behandlas nedan får inte heller tabellen bli för full. Vid en fyllnadsgrad på 70-80% börjar prestanda för vissa metoder att sjunka påtagligt. För att så kallad kvadratisk sondering ska fungera bra går gränsen vid 50%. Detta betyder att man dels måste välja lämplig tabellstorlek, dels kan behöva överdimensionera 20-30%, eller mer.

3.1 Hashfunktioner

En hashfunktion ska vara reproducerbar, snabb och ger en slumpmässig spridning av nyckelvärdena över adressrymden.

Det är inte enkelt att finna bra hashfunktioner och speciellt inga universalfunktioner. Ett problem är att nyckelvärdesmängden ofta är mycket större än adressrymden. Personnummer ger en nyckelvärdesmängd som i princip omfattar 10^{10} möjliga nycklar. Om personnummer användes som nyckel men inte fler än 10.000 personer behöver hanteras är det önskvärt att dimensionera hashtabellen därefter.

Olika slags nyckelvärden kräver olika former av transformationer för att ge en god spridning av adresserna. Ett personnummer kan lagras som en teckensträng men det är också tänkbart att lagra det som ett heltal. Vilken av dessa representationer som används innebär att helt olika transformationer krävs för att ge ett bra resultat.

Ofta är det inte i förväg känt exakt vilka nycklar bland de tänkbara som kommer att uppträda och ofta inte heller exakt hur många poster som kommer att behöva lagras.

Då det gäller att sprida nyckelvärdena jämnt över adressrymden, vore det idealiskt om varje nyckel kunde transformeras till en unik adress. En funktion med den egenskapen kallas för en *perfekt hashfunktion*. Om dessutom hashfunktionen kan generera samtliga adresser, så att hashtabellen kan fyllas till 100%, kallas hashfunktionen för en *minimal perfekt hashfunktion*. Det är endast under mycket speciella förutsättningar som sådana ideala funktioner kan konstrueras. Då det gäller hashfunktioner bygger mycket på empiriska studier och på förmågan att göra bra uppskattningar.

3.1.1 Förbearbetning av nycklar

Innan en nyckel transformeras kan *förbearbetning* (preconditioning) av nyckelvärdet behövas. Nyckelns storlek kan annars göra det svårt eller omöjligt att bearbeta den direkt med aritmetiska eller logiska operationer, vilket många transformationer utgörs av. Till exempel alfanumeriska nycklar, strängar, behöver normalt förbearbetas innan själva hashfunktionen kan appliceras.

Förbearbetning av alfanumeriska nycklar kan göras genom att införa en heltalskod för varje tecken. Siffertecknen 0-9 kan exempelvis kodas med heltalen 1-10, bokstäverna A-Ö med heltalen 11-38, specialtecken som +, -, %, ? med heltal 39, 40, osv. De heltal som erhålls för de tecken som ingår i en nyckel kombineras på något lämpligt sätt. Nyckeln AB3 kan transformeras till heltalet 11124

genom att helt enkelt byta ut tecknen i teckensträngen till motsvarande sifferkoder. Även andra manipulationer av heltalskoderna tänkas för att erhålla ett lämpligt heltal.

Ofta önskas ett resultat som kan lagras i ett datorord (till exempel 32 bitar) så att datorinstruktioner kan appliceras direkt. Själva hashfunktionen appliceras sedan på det erhållna talet.

En annan möjlighet att förbearbeta alfanumeriska nycklar är att utnyttja numeriska koder som används i datorer för att representera tecken, till exempel ASCII-kod eller Unicode. Nyckeln AB3 kan kodas som heltalet 656651 genom att sätta samman de teckenkoderna, 65, 66 och 51.

Ibland är det att föredra att sätta ihop de binära teckenkoderna. Om 8-bitars ISO-kod används och nyckeln A1 transformeras, erhålls 1100000111110001, eftersom tecknet A har koden 11000001 och tecknet 1 har koden 11110001.

3.1.2 Val av hashfunktion

Val av hashfunktion beror av många saker:

- *Kan hashfunktionen välja så att den passar de nycklar som uppträder?* I en kompilator kan detta inte göras, eftersom det är okänt vilka identifierare som program kommer att innehålla. I ett databassystem, speciellt om det används under lång tid, kan däremot så vara fallet.
- *Är beräkningstiden för hashfunktionen kritisk?* Om hashtabellen lagras på sekundärminne är det möjligt att åtkomsttiderna för läsning och skrivning är så långa att en i sig kostsam metod som minimerar antalet sekundärminnesoperationer ändå kan löna sig. I en kompilator däremot, som lagrar symboltabeller i primärminnet, önskas en så snabb metod som möjligt.
- *Hur långa är nycklarna?* Om beräkningstiden är kritisk kanske inte divisionsmetoden kan användas om nycklarna är längre än ett datorord. Om ändå division önskas användas, kan denna kombineras med *folding* som förbearbetningsmetod.
- *Är adresserna binära eller decimala?* En del metoder bygger på specifika maskininstruktioner, som förutsätter antingen binär eller decimal form.
- *Kan index i hashtabellen väljas?* Lagras data på sekundärminne kan detta inte göras, eftersom adresserna redan är ”valda” i filsystemet. En del metoder kräver att tabellstorleken exempelvis är en potens av 2 eller av 10, vilket ställer krav på att vi måste kunna välja index.
- *Hur ser nycklarna ut, finns det något mönster?* Vissa hashfunktioner kan ge hög kollisionsfrekvens om det finns någon form av likformighet hos nycklarna. Å andra sidan kan nyckelvärdena vara så väl spridda och ha en sådan form att det inte behövs någon hashfunktion alls, något som kan vara fallet om nycklar kan väljas.

Det finns ett antal beprövade nyckeltransformationer beskrivna i litteraturen och det är bland dessa som man normalt försöker hitta en lämplig hashfunktion. Närdetta är gjort, gäller det att bestämma vissa parametrar hos den valda transformationen för att anpassa den till de specifika nycklarna. Sist i denna översikt tas ett antal vanliga transformationer upp.

3.1.2.1 Divisionsmetoden

En metod som ofta används i kurslitteratur är *divisionsmetoden*. Den bygger på att nycklarna är heltal eller kan förbearbetas så att ett motsvarande heltal för varje nyckel erhålls. Nycklarna divideras med tabellstorleken *T* och den rest (mellan 0 och *T*-1) som uppstår används som adress i hashtabellen. Om inte första positionen i hastabellen indexeras 0 får man addera det offset som gäller till första positionen, till exempel 1.

3.2 Kollisionshantering

När en ny post som ska sättas in hamnar på samma hemadress som en redan befintlig post har en *kollision* (collision) inträffat och en alternativ plats måste hittas för den nya posten. Det finns i princip två alternativ, antingen söks en ledig plats inom hashtabellen upp eller så används ett separat minnesutrymme. Inom dessa två huvudalternativ finns olika varianter, varav några vanliga tas upp nedan.

Att kollisioner inträffar innebär vanligtvis också att *klungbildning* (cluster) uppstår. I grunden innebär klungbildning att poster klumpar ihop sig på vissa ställen i tabellen men en klunga kan också bestå poster som ligger utspridda men ingår i en gemensam sökväg.

När klungor har uppstått har de ofta en tendens att växa till ytterligare. Klungor innebär ett ökat sökarbete, så det är önskvärt att försöka eliminera eller minska risken för klungbildning. Man brukar i detta sammanhang skilja mellan primär klungbildning och sekundär klungbildning.

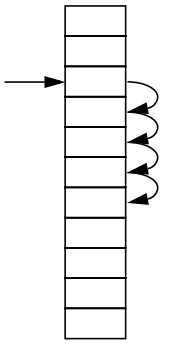
- *Primär klungbildning* – om en kollision inträffar vid insättning kommer den klunga som man hamnat i att utökas genom den nya posten sätts in sist i klungan. Detta är en direkt konsekvens av hur kollisionshanteringsmetoden linjär sondering fungerar, se nedan.
- *Sekundär klungbildning* avser klungor som består av poster med samma hemadress och där kollisionshanteringsmetoden besöker samma alternativadresser. Kollisionshanteringsmetoderna linjär sondering och kvadratisk sondering lider båda av sekundär klungbildning.

För att eliminera eller reducera klungbildning kan man försöka angripa orsakerna till klungbildningen. För primär klungbildning är strategin att under sonderingen hoppa ur den klunga som man hamnat i, vilket är idéen bakom kollisionshanteringsmetoden *kvadratisk sondering*. För att komma åt sekundär klungbildning måste kollisionshanteringsmetod, beroende på nycklarna, kunna variera vilka positioner som besöks i sonderingen, vilket är vad kollisionshanteringsmetoden *dubbelhashning* gör.

3.2.1 Linjär sondering

Linjär sondering (linear probing) innebär att linjärsökning görs från hemadressen och framåt i tabellen. Om slutet av tabellen nås fortsätter sökningen från början och kan till slut återkomma till hemadressen. Vid insättning skulle detta innebära att tabellen är full men så långt ska det aldrig gå i realiteten. Följande formler definierar beräkningen av hemadressen h_0 och, om kollision uppstår, den linjära sonderingen. Hashtabellens storlek är T och adresserna går från 0 till T–1:

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



Till metodens fördelar hör dess enkelhet och att den ger fullständig tabelltäckning. Till dess nackdelar hör att det finns benägenhet till klungbildning. Hamnar man i en klunga vid insättning kommer man inte ur klungan, utan den växer garanterat till på slutet. Kodningsmässigt kan det vara enklare att använda föregående adress och öka den med 1:

$h_i = (h_{i-1} + 1)$ **mod** T $i = 1, 2, \ldots, T$.

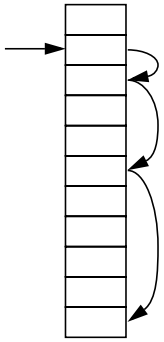
3.2.2 Kvadratisk sondering

Kvadratisk sondering (quadratic probing) innebär att alternativa platser i tabellen söks på ett avstånd från hemadressen som ökar med kvadraten på antalet försök. Första försöket görs en (1^2) position framåt i tabellen från hemadressen. Är inte den platsen ledig görs det andra försöket fyra (2^2) positioner framåt i tabellen från hemadressen, tredje försöket görs nio (3^2) steg framåt från hemadressen, osv.

I detta fall är det inte uppenbart att tabelltäckningen är fullständig och man kan dessutom visa att efter *T*/2 söksteg kan adresser som redan undersökts komma att återbesökas, vilket om inte annat innebär att sökningen blir ineffektiv. Därför bör sonderingen avbrytas efter *T*/2 steg, avrundat nedåt. I praktiken ska sonderingen normalt inte behöva fortgå så långt.

Hemadressberäkning och kvadratisk sondering beskrivs av följande formler för en tabell med storlek T och med adresser från 0 till *T*–1:

$h_0 = \text{hash}(nyckel)$ hemadressen
$h_i = (h_0 + i^2)$ **mod** T $i = 1, 2, \ldots, \lfloor T/2 \rfloor$.



Idéen är att när en ledig adress hittats innebär detta också att vi hoppat ur klungan, dvs inte hamnade i första lediga positionen efter klingan, som i fallet linjär sondering, utan en bit ifrån. Kvadreringen av *i* i formeln ovan kan undvikas med följande alternativa formel:

$h_i = (h_{i-1} + i)$ **mod** T $i = 1, 2, \ldots, \lfloor T/2 \rfloor$.

Detta ger visserligen inte exakt samma positioner som vid kvadrering av *i* (avstånd 1, 4, 9, 16, 25,… från hemadressen) men något liknande (1, 3, 6, 10, 15, …).

3.2.3 Dubbelhashning

Dubbelhashning är i princip linjär sondering men att det steg som används i sonderingen slumpas ur den aktuella nyckeln. I linjär sondering används alltid steget 1. Idéen med dubbelhashningen är att olika nycklar ska ge olika steg och därmed att olika positioner i tabellen besöks i sonderingen. Om en nyckel ger steget 2 kommer adresserna h_0+2, h_0+4, h_0+6, o.s.v. att besökas. Om en annan nyckel med samma hemadress ger steget 5 kommer positionerna h_0+5, h_0+10, h_0+15, och så vidare att besökas. Ibland kommer samma position att undersökas, till exempel h_0+10 i detta fall, men i princip anses sekundär klungbildning ändå vara eliminerad.

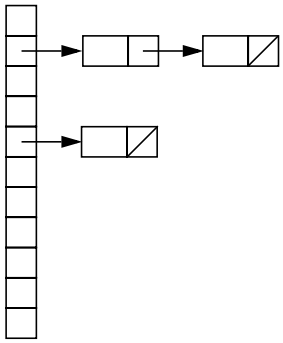
Hemadressberäkning och dubbelhashning beskrivs av följande formler för en tabell med storlek T, som adresseras från 0 till T-1:

$h_0 = \text{hash}(nyckel)$ hemadressen
$c = R - (nyckel$ **mod** $R)$ R är ett primtal, sådant att $R < T$
$h_i = (h_0 + i \cdot c)$ **mod** T $i = 1, 2, \ldots, T$.

I formeln för *c* ovan förutsätts *nyckel* vara ett heltal, vilket kan innebära att den måste förarbetas. Om den hashfunktion som används är divisionsmetoden med inbyggd förbearbetning kan den användas med i så fall *R* i stället för *T* som divisor. Sonderingssteget *c* kommer då att hamna mellan 1 och *R*. För att erhålla maximal spridning av sonderingssteget *c*, väljs *R* som det största primtalet som är strängt mindre än tabellstorleken *T*. Även i detta fall kan en alternativ formel som undviker multiplikationen användas:

$h_i = (h_{i-1} + c)$ **mod** T $i = 1, 2, \ldots$

3.2.4 Separat länkning



Ett sätt att hantera kolliderade poster är att varje tabellingång får vara en länkad lista. Poster med samma hemadress läggs in i en länkad lista, vilken kommer att utgöra en klunga på den adressen.

Idéen med separat länkning (separate chaining) är att även om sökning i de länkade listorna är en linjär operation kommer listorna att vara korta och sökningen tillräckligt effektiv.

Fyllnadsgraden 100% inte är en övre gräns för en hashtabell med separat länkning, utan snarare en lämplig fyllnadsgrad. En lägre fyllnadsgrad förbättrar inte påtagligt prestanda och en, inom rimliga gränser, högre fyllnadsgrad är acceptabel och kan innebära att utrymme sparas, jämfört med om man måste överdimensionera tabellen.

3.3 Omhashning

Omhashning (*rehashing*) innebär att en tabell utökas om fyllnadsgraden överskrider en viss nivå, till exempel 70%. En ny, större tabell skapas och alla poster i den gamla tabellen flyttas över. Överflyttningen görs med en ny hashfunktion som är anpassad till den nya tabellens storlek, därav namnet omhashning. Om divisionsmetoden används är omhashningen enkel, eftersom den använder just tabellstorleken som divisor. Den nya tabellens storlek kan till exempel bestämmas genom att dubblera den gamla tabellens storlek och sedan välja det primtal som är närmast större.

3.4 Linjärhashning

Utmärkande för linjärhashning (*linear hashing*) är att hashtabellens storlek ökas och minskas dynamiskt och att flera hashfunktioner används samtidigt. I samband med att tabellens storlek förändras anpassas hashfunktionerna så att hela tabellen alltid adresseras. Linjär hashning utformades ursprungligen för externa hashtabeller men har senare modifierats och visat sig användbar för även interna tabeller.

Hashtabellen organiseras i hinkar *(buckets)*, vilka kan ha variabel eller fix storlek. Initialt finns *M* hinkar i den tomma tabellen och de numreras 0, 1, …, *M*-1, där $M \geq 2$.

Olika kriterier kan användas för att bestämma då storleken hos hashtabellen ska ökas eller minskas, till exempel *fyllnadsgraden*. Ökning av antalet hinkar sker då ett bestämt tröskelvärde för fyllnadsgraden överskrids i samband med insättning, minskning sker då ett annat tröskelvärde för fyllnadsgraden underskrids efter att en post har tagits bort.

Om hashtabellen har fix hinkstorlek kan hinkdelning initieras även vid *överskott*, det vill säga då en ny post hamnar i en hink som redan är full. Hinkdelning sker i en bestämd ordning därför är det vanligtvis inte den hink där ett överskottet uppstår som delas.

Utökning av en hashtabell sker genom *delning* av hinkar enligt ett förutbestämt schema. I det enkla fallet delas en hink i taget, i lite mer komplicerade strategier delas flera hinkar då tabellen växer. Nedan beskrivs endast tillväxt med en hink i taget.

Hinkdelning kan göras enligt följande schema för en tabell med initial storlek *M* hinkar:

- Först delas hink nummer 0 och posterna i hink 0 fördelas med en annan hashfunktion mellan hink 0 och den nya hinken, som får nummer *M*.
- Nästa gång delas hink nummer 1 och de poster som finns i den fördelas mellan hink 1 och en ny hink M+1.

- Delningen fortskrider i hinkordning till dess hink M-1 delats och givit upphov till hink 2M-1. Därmed har samtliga ursprungliga hinkar delats och tabellens storleken har fördubblats till 2M.

Vid fortsatt hinkdelning börjar man om med hink 0 och delar sedan successivt hinkarna i ordning till dess hink 2M-1 delats och tabellens storlek ökat till 4M hinkar. Då börjar man åter om hinkdelningsproceduren med hink 0, osv.

Efter borttagning kontrolleras om fyllnadsgraden underskrider ett bestämt tröskelvärde. Om så är fallet minskas tabellens storlek genom att två hinkar slås ihop. Om hink *i* var den senaste delade hinken kommer den och den motsvarande hink som skapades vid delningen att slås ihop.

Då nya hinkar läggs till måste sökproceduren modifieras. I sökproceduren används därför en följd av hashfunktioner, $hash_0$, $hash_1$, $hash_2$, …, sådana att

$$hash_i(k) = k \bmod (2^i M) \quad i = 0, 1, 2...$$

Dessa hashfunktioner motsvarar tabellstorlekarna M, 2M, 4M och så vidare, för växande *i*. Initialt innehåller hashtabellen M hinkar och då används enbart funktionen $hash_0$. Då hink 0 delats och en ny hink M lagts till på slutet av tabellen kan den nya hinken inte adresseras av $hash_0$ men väl av $hash_1$. Funktionen $hash_1$ adresserar en tabell som innehåller 2M hinkar, vilket ännu inte är fallet, och $hash_1$ får därför endast användas på nycklar som först placeras i hink 0 av $hash_0$. Funktionen $hash_0$ gäller fortfarande för poster som finns i de odelade hinkarna, liksom för nya poster med nycklar som placeras i en odelad hink av $hash_0$. För varje delad hink gäller alltså, för val av hashfunktion, följande princip:

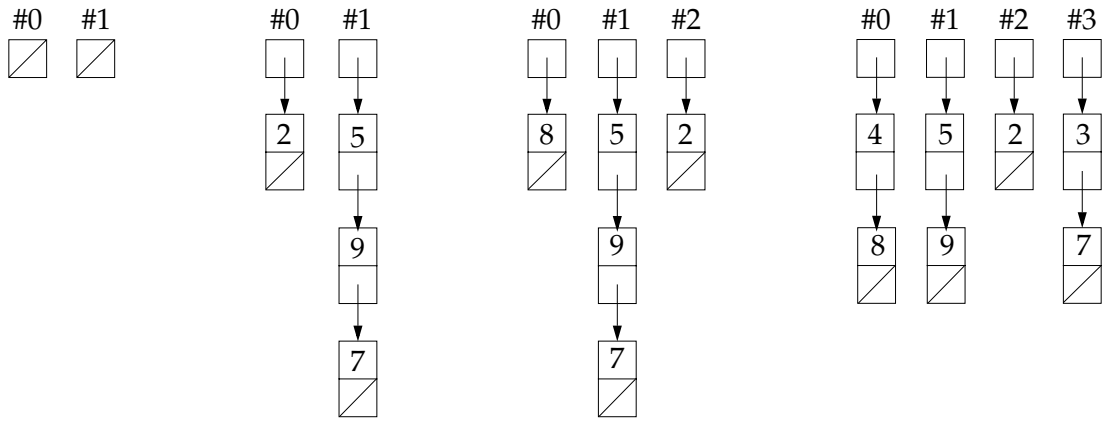
1. om *ingen* hink har delats, sök i den hink som $hash_0$ anger
2. om hinkar har delats, och $hash_0$ ger adressen för en odelad hink, sök i den hinken
3. i annat fall, sök i den hink som $hash_1$ anger.

Då samtliga *M* hinkar i den ursprungliga tabellen har delats innehåller tabellen 2M hinkar, 0 till 2M-1, och samtliga poster har fördelats med funktionen $hash_1$. Det innebär att $hash_0$ inte längre är av intresse. Då tabellen behöver utökas ytterligare startar hinkdelningsproceduren om från hink 0 och fortsätter upp till hink 2M-1. För odelade hinkar används funktionen $hash_1$ och $hash_2$ används för delade hinkar. Då samtliga dessa 2M hinkar också har delats har hashtabellen utökats till 4M hinkar, numrerade 0 till 4M-1, och samtliga poster har fördelats med funktionen $hash_2$. Delningsproceduren startar om från hink 0, denna gång med 4M hinkar och med hashfunktionerna $hash_2$ och efter delning $hash_3$.

Sökproceduren kan beskrivas med följande algoritm, där k är söknyckeln, *n* anger antalet delade hinkar och *m* den hink som *k* ska sökas i.

$m = \text{hash}_i(k)$
om $n > 0$ **och** $m < n$ ($m<n$ innebär att *m* är en delad hink)
  $m = \text{hash}_{i+1}(k)$
Sök nyckeln *k* i hink nummer *m*.

I figur 43 visas fyra situationer för en linjär hashtabell. Längst till vänster visas en tom hashtabell med initialt M=2 hinkar. Hinkstorleken förutsätts vara variabel och i varje hink lagras posterna i en länkad lista. Övriga tre delfigurer visar hur nycklar lagras och hinkar delas för indatasekvensen 7, 2, 9, 5, 8, 4, 3. Hinkdelning har gjorts då fyllnadsgraden överskridit 2 poster per hink i genomsnitt.



Figur 43. Linjärhashningstabell med M=2.

3.5 Översikt av hashfunktioner

3.5.1 Sifferurvalsmetoden, sifferanalysmetoden

Sifferurval kan användas för sådana nycklar som utgörs av siffersträngar, till exempel personnummer. Idén är att välja ut ett antal siffror ur nyckeln, vilka tillsammans får utgöra en adress. Hur många siffror som väljs beror på adressrymden och siffrorna väljs med tanke på spridningen av adresserna. Om populationen är känd i förväg kan en *sifferanalys* göras för att hitta de siffror som ger den bästa spridningen. Sifferurvalsmetoden kallas också sifferanalysmetoden.

Antag att högst 10.000 personer ska kunna lagras i en hashtabell med adresserna 0-9999 och att personnummer används som nyckel. En enkel transformation vore att välja de fyra första siffrorna i personnumret. Ett uppenbart problem att det inte kan ge alla adresser i hashtabellen, utan endast 1200 av de 10.000, nämligen 1-12, 101-112, …, 9901-9912. Detta ger att en dålig spridning och hög kollisionsfrekvens redan vid låg fyllnadsgrad.

Ett bättre val av siffror vore att välja de två första, som varierar mellan 0-99, den fjärde, som varierar mellan 0-9, och den sjätte, som varierar mellan 0-9. Detta kommer att kunna ge samtliga adresser 0-9999. Om en analys görs upptäcker man dock att spridningen inte blir likformig. Detta beror på att siffrorna 1 och 2 förekommer mer frekvent i månadsnumrens sista siffra och att siffran 1 förekommer något mer frekvent i dagnumrens sista siffra.

Man måste också tänka på är hur de verkligt förekommande personnumren kan förväntas vara fördelade. Det kan ju vara så att materialet endast utgör ett speciellt urval av befolkningen.

Ytterligare en sak att tänka på är möjligheten att det kan finnas beroenden mellan siffror. Det kan vara så att vissa sifferkombinationer är vanligare än andra, vilket kan störa fördelningen. Korrelationsanalys mellan olika siffror är därför lämpligt att utföra.

3.5.2 Divisionsmetoden

En av de mer effektiva nyckeltransformationsmetoderna är division. Nyckelvärdet divideras, efter eventuell förbearbetning för att erhålla ett heltalsvärde, med tabellstorleken och resten vid divisionen får utgöra adressen.

Då nycklar transformeras med divisionsmetoden bibehålls till viss del regelbundenheter som finns i nyckelmängden. Närliggande nyckelvärden kommer att ge närliggande adresser. Om man dividerar med till exempel 31 kommer nycklarna 993, 994 och 1000 att ge adresserna 1, 2 respektive 8, liksom även nycklarna 838, 839 respektive 845. Denna egenskap är negativ och kan medföra klungbildning på vissa adresser.

Tabellstorleken bör väljas med omsorg, till exempel ett stort primtal, vilket gör det osannolikt att olika nycklar har detta tal som gemensam faktor. I praktiken har det visat sig fungera bra med udda divisorer utan gemensamma faktorer mindre än 20. Jämna divisorer ska undvikas eftersom de systematiskt transformerar udda och jämna nycklar till udda respektive jämna adresser.

3.5.3 Mittkvadratmetoden

I mittkvadratmetoden multipliceras nyckeln med sig själv, kvadreras, och därefter väljs ett lämpligt antal siffror eller, om på binär nivå, bitar ur resultatets mitt.

Nyckeln 123456 kvadrerad ger 15241383963 och de tre mittersta siffrorna, 138, kan exempelvis väljas som adress. Varför det är viktigt att välja värdet mitt i resultatet, ser man av följande:

```
    123456
  × 123456
    740736
   617280
  493824
 370368
246912
+ 123456
 1524138 3963
```

Antag att de tre siffrorna längst till höger, 963, väljs som adress. Om man studerar hur detta värde uppkommit ser man att det skett genom multiplikationer som endast berör de tre längst till höger stående siffrorna i nyckel. Alla nycklar som slutar på 456 kommer att transformeras till adressen 963, vilket är något som bör undvikas. För de tre mittersta siffrorna däremot gäller att de tillkommit med hela nyckelvärdet inblandat. En ändring av någon siffra i nyckeln ger med stor sannolikhet en annan adress. Det finns fler metoder som bygger på multiplikation.

3.5.4 Folding-metoder

I foldingmetoderna (vika, lägga ihop) delas nyckelvärdet först upp i ett antal delar, vilka var och en har samma längd som den önskade adressen, med undantag för ena eller båda änddelarna om nyckellängden inte är jämnt delbar med den önskade adresslängden. Adressen erhålls genom att adderas delarna, eventuellt spill som uppstår vid addition av de mest signifikanta siffrorna ignoreras. Addition kan ersättas med exklusivt-eller-operationer om beräkningar görs binärt.

Om en fyrsiffrig adress ska beräknas ur nyckeln 4869273510, kan detta göras genom att dela upp nyckeln i tre delar, 486, 9273 och 510, och addera dessa:

```
  486
 9273
+ 510
 1269
```

Ettan till vänster i resultatet ignoreras, vilket ger adressen 269. Detta sätt att addera delarna brukar kallas *fold-shift-metoden*. En variant är att ta siffrorna i första och sista delen i omvänd ordning vid additionen, vilket kallas *fold-boundary”-metoden*:

```
  684
 9273
+ 015
 9972
```

Folding gör det möjlighet att transformera långa nyckelvärden till kortare adresser och där samtliga siffror i nyckeln kommer att ingå i beräkningen. Ofta används folding i kombination med andra

metoder. Folding används då först för att erhålla ett värde som exempelvis ryms i ett maskinord, och sedan appliceras ytterligare någon hashfunktion för att erhålla den önskade adressen.

3.5.5 Längdberoendemetoder

I längdberoendemetoden ingår nyckelvärdets längd som en faktor i beräkningen, tillsammans med nyckelvärdet. Detta kan vara speciellt intressant när nycklarnas längd inte är fix. Längdberoendemetoder används till exempel för nycklar som utgörs av strängar med variabel längd.

En enkel beräkning som visat goda resultat innebär att addera teckenkoderna för det första och det sista tecknet i en sträng med värdet för strängens längd skiftad vänster fyra binära steg (det kan göras genom att multiplicera med 16). Strängen "PASCAL" ger med denna beräkning adressen 252 (80 + 76 + 16 × 6).

Längdberoendemetoder kan användas för förbearbetning av nycklar, till exempel kan divisionsmetoden appliceras på det heltalsvärde som en längdberoendemetod ger.

3.5.6 Slumpmetoden

En möjlighet att generera adresser är att använda en slumptalsgenerator, det vill säga en funktion som genererar pseudoslumptal utifrån ett initialvärde, ett så kallat frö (*seed*). För ett givet frö kommer slumptalsgeneratorn alltid att ge samma följd av slumptal.

Om nyckelvärden efter lämplig förbearbetning får utgöra frön, levererar slumptalsfunktionen en hemadress för varje nyckel. Olika nycklar kan ge samma första slumptal, så kollisioner kan uppstå. I sådana fall anropas slumptalsfunktionen igen, men nu utan frö för att ge en ny adress. Skulle även denna position i hashtabellen vara upptagen anropas slumptalsfunktionen igen, och så vidare. I och med att slumptalsfunktionen ger samma slumptalsföljd för ett givet frö, kommer alltid en viss post att kunna återfinnas.

3.5.7 Radixmetoden

Radixmetoden bygger på att nyckelvärden transformerar genom att arbeta med olika radix. Antag att nycklar består av strängar av oktala siffror. Radix är alltså 8. Siffrorna i strängen betraktas dock som om de hade en annan bas, säg 11. Önskas en decimal adress omvandlas talet i bas 11 till bas 10. Detta kommer att ”röra om” bland bitarna/siffrorna och sedan kan, på samma sätt som i mitt-kvadratmetoden (se ovan), ett lämpligt antal siffror i resultatets mitt väljas som adress.

3.5.8 Perfekta hashfunktioner

Perfekta hashfunktioner kan endast erhållas under speciella omständigheter. Ett villkor är att alla nyckelvärden som kan uppträda är kända på förhand. I kompilatorer kan man tänka sig använda en sådan hashtabell för att avgöra om en inläst sträng är ett reserverat ord eller inte. Det inlästa strängen transformeras och innehållet i hashtabellen på den erhållna adressen jämförs med den inlästa strängen.

Ett problem är att beräkningsarbetet för att erhålla hashfunktionen kan bli så stort att det i praktiken är omöjligt att göra detta, även om det endast behöver göras en gång för en viss tillämpning. Beräkningsarbetet växer exponentiellt med antalet nycklar och kan bli oacceptabelt stort redan när antalet nyckelvärden överstiger några hundratal.

Ett exempel på en perfekt hashfunktion för de reserverade orden i Modula-2 är[1]:

$$\text{hash}(w) = \textit{length} + g(w[1]) + g(w[\textit{length}]) + a$$

1. Sebesta och Taylor, J*ournal of Pascal, Ada & Modula-2*, mars/april 1986.

där *w* är ett ord, *length* är ordets längd och *a* anger det alfabetiska ordningstalet (där bokstaven A har ordningstalet 0) för den *näst sista* bokstaven i ordet. Funktionen *g* (som kan vara mycket kostsam att ta fram) associerar ett heltal med varje bokstav enligt följande:

tecken	g	*tecken*	g	*tecken*	g	*tecken*	g
'A'	18	'H'	14	'O'	7	'V'	22
'B'	16	'I'	15	'P'	1	'W'	-2
'C'	3	'J'	–	'Q'	22	'X'	–
'D'	-3	'K'	–	'R'	-1	'Y'	14
'E'	-11	'L'	15	'S'	3	'Z'	–
'F'	4	'M'	1	'T'	0		
'G'	–	N'	-4	'U'	8		

För det reserverade ordet MODULE erhålls således adress 7 i tabellen enligt följande:

hash("MODULE") = 6 + g('M') + g('E') + 11 = 7

Den fullständiga hashtabellen för Modula-2 är:

[0] ELSE	[15] MOD	[30] DIV
[1] EXIT	[16] PROCEDURE	[31] AND
[2] END	[17] DEFINITION	[32] QUALIFIED
[3] WHILE	[18] RETURN	[33] BY
[4] THEN	[19] RECORD	[34] LOOP
[5] REPEAT	[20] FOR	[35] WITH
[6] ELSIF	[21] IN	[36] UNTIL
[7] MODULE	[22] OR	[37] ARRAY
[8] TYPE	[23] FROM	[38] IMPORT
[9] DO	[24] VAR	[39] IMPLEMENTATION
[10] SET	[25] BEGIN	
[11] POINTER	[26] CONST	
[12] EXPORT	[27] OF	
[13] NOT	[28] TO	
[14] CASE	[29] IF	

4 Prioritetskö och heap

En prioritetskö ger åtkomst till data på ett ordnat men begränsat sätt. I en prioritetskö rangordnas data och endast det minsta (alternativt största) värdet är åtkomligt. För att komma åt nästa värde i ordning måste det minsta värdet först tas bort. När ett värde läggs till i en prioritetskö ordnas det in i efter rang och kan sedan inte kommas åt förrän det hamnat först i kön.

I en prioritetskö kan det finnas flera element med samma rang och då gäller ingen absolut ordning mellan sådana värden. Det innebär att värden med lika rang inte behöver komma ut ur prioritetskön i samma inbördes ordning som de sattes in.

Prioritetsköer kan implementeras på olika sätt. Logiskt sett är en prioritetskö en rangordnad lista och en möjlighet är att använda ett en fältliknande struktur. En fullständig rangordning krävs dock inte i en prioritetskö, utan det räcker att komma åt det minsta värdet medan övriga värden kan vara förhållandevis ”löst” ordnade. En datastruktur som har denna egenskap är *heap*, se nedan.

De tre grundläggande operationerna på en prioritetskö är att sätta in ett nytt värde i rangordning (*insert*), att visa det minsta värdet i prioritetskön utan att det tas bort (*find-min*) och att ta bort det minsta värdet utan att returnera det (*delete-min*). För övrigt det finnas en operation för att undersöka om en prioritetskö är tom (*empty*), kanske även för att ta reda på hur många värden som finns i en prioritetskö (*length*), samt för att tömma en prioritetskö (*clear*).

4.1 Heap

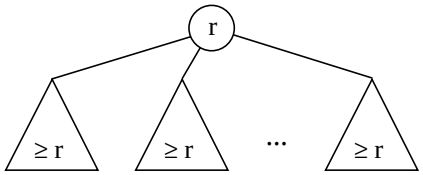
En heap är en trädstruktur där data är ordnade enligt *heapordning*. Ett sätt att definiera en *heap* är:

En *min-heap* är ett träd vars värden finns i stigande ordning utmed varje väg från roten till löven.

Mellan syskon och subträd finns däremot ingen ordning. Ett alternativt sätt att definiera en *heap* är:

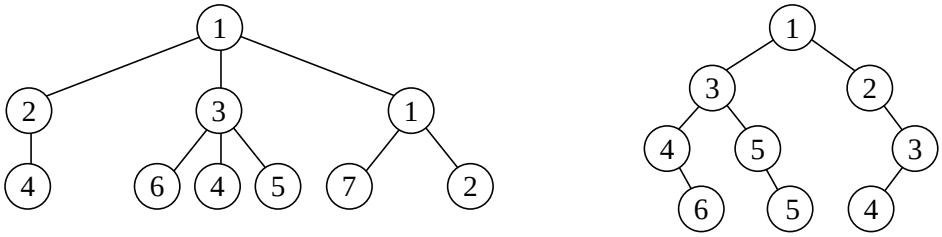
En *min-heap* är ett träd där roten innehåller det minsta värdet och vars subträd är *min-heapar*.

Analoga definitioner kan göras för heapar där värdena är lagrade i sjunkande ordning från rot till löv, *max-heapar*. I figur 44 illustreras min-heapordning med ett symboliserat träd, där *r* står för *rang*.



Figur 44. Min-heap.

I figur 45 visas två exempel på *min-heapar*, till vänsater en min-heap med grad 3, till höger en min-heap med grad 2.

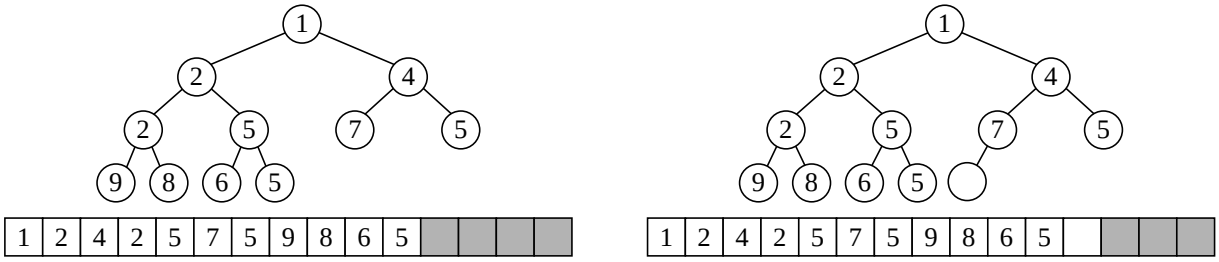


Figur 45. Exempel på min-heapar.

En heap kallas en *komplett heap* om trädet som lagrar heapens värden är ett komplett träd. En komplett heap kan med fördel lagras i ett fält.

4.1.1 Binär heap

En *binär heap* är ett *heapordnat*, komplett binärt träd. Att trädet är komplett innebär att en binär heap kan lagras i ett fält, se figur 46 a. Detta kallas *implicit representation* för ett träd.

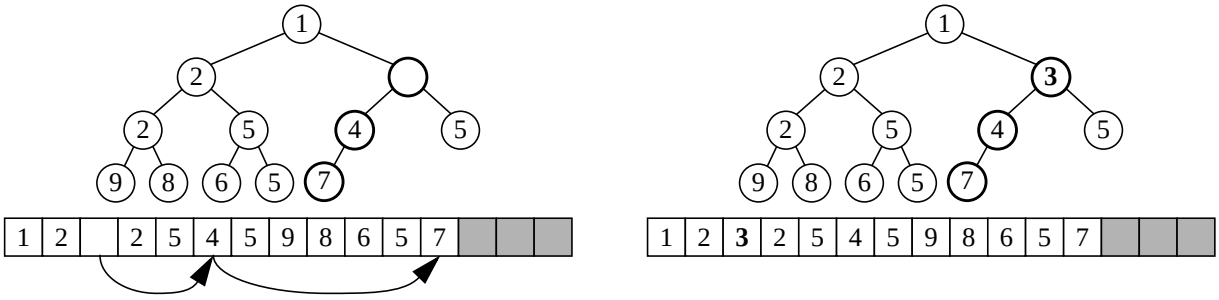


a) binär heap b) en tom nod har satts in sist

Figur 46. Binär min-heap.

4.1.1.1 Insättning i binär min-heap

När ett värde ska sättas in i en binär heap, *insert*, läggs först en ny tom nod till i trädet, ett *hål*. Det enda stället en ny nod kan sättas in i en heap är nästa lediga position i fältet, annars blir trädet inte komplett. Se figur 46 b ovan.



a) hålet har vandrat upp så att 3 kan sättas in b) 3 har satts in

Figur 47. Insättning i binär min-heap.

Så länge värdet i föräldranoden är större än värdet som ska sättas in, flyttas värdet i föräldranoden ner till hålet och hålet vandrar upp en nivå. Detta upprepas till dess, antingen värdet i föräldranoden till hålet är mindre än eller lika med värdet som ska sättas in, eller att hålet har vandrat ända upp till roten. Det nya värdet sätts in hålet. Se figur 47. Man kan se detta som om det nya värdet direkt sattes in i det nya lövet och att sedan värdet vandrar uppåt i heapen genom att successivt byta plats med värdet i föräldranoden till dess heapordning erhålls, vilket skulle fungera men vara mindre effektivt. Att låta ett hål, eller ett värde, vandra uppåt i en heap på detta sätt kallas på engelska för *percolate-up*, *walk-up* eller *sift-up*.

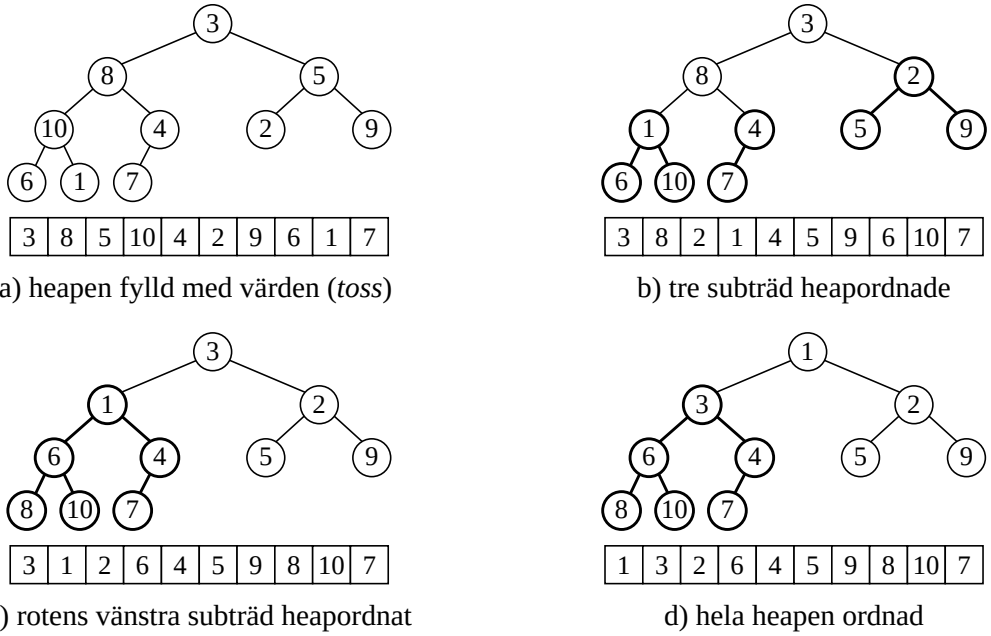
En analys av *insert* ger att insättning av n värden har tidskomplexiteten $O(n \cdot \log n)$. Ett nytt värde kan direkt sättas in som ett nytt sista löv, dvs i konstant tid, $O(1)$. Eftersom trädet är komplett, är väglängden upp till roten garanterat $\lfloor \log n \rfloor$, vilket innebär att *percolate-up* har tidskomplexiteten $O(\log n)$.

I speciella fall kan ett alternativt sätt att sätta in i en binär heap användas. Om alla värden som ska sättas in i en heap finns tillgängliga kan värdena först placeras i heapen, utan tanke på heap-ordningen, genom att sätta in dem uppifrån-och-ner, bredden-först. Den operationen kallas *toss*.

Därefter ordnas värdena i heapen med en operation som kallas *fix-heap* eller *build-heap*. I figur 48 nedan visas hur följande 10 värden sätts in i en min-heap med *toss* och sedan ordnas med *fix-heap*.

```
3  8  5  10  4  2  9  6  1  7
```

Först placeras de 10 värdena (*n*=10) i heapen med *toss*, se figur 48 a.



a) heapen fylld med värden (*toss*)

b) tre subträd heapordnade

c) rotens vänstra subträd heapordnat

d) hela heapen ordnad

Figur 48. Generering av min-heap med toss och fix-heap.

Med början i *sista icke-lövet* vandrar man sedan nod för nod, nivåvis från höger till vänster, upp till roten. Eftersom heapen är lagrad i ett fält är detta enkelt:

- *Sista icke-lövet* hittas i fältet i position *n*/2, d.v.s. i detta fall position 5 (där värdet 4 finns).
- Att vandra nod för nod till roten innebär att successivt stega ner positionen till 1.

För varje nod på vägen till roten heapordnas det subträd som noden är rotnod i och till slut heapordnas hela heapen. För att heapordna ett subträd görs *percolate-down* (*walk-down, sift-down*) på subträdets rotvärde till dess heapordning erhålls.

Arbetet med att ordna heapen i figur 48 a inleds i noden med värdet 4. I detta fall finns bara ett barn och dess värde (7) är större än 4, så detta subträd är redan en min-heap.

Nästa subträd att behandla är det som har 10 i roten. Här finns två barn och både är mindre än 10 (6 och 1). Värdet i det högra subträdet (1) är minst och får byta plats med 10 (*percolate-down* utförs på 10). Den nod som 10 flyttas ner i har inga barn och därmed är detta subträd ordnat.

Därefter står subträdet med 5 i roten på tur. Det finns två barn men det är bara värdet i det vänstra subträdet (2) som är mindre än 5 och kan byta plats med 5.Nu ser heapen ut som i figur 48b.

Det subträd som ska heapordnas härnäst är det som har sin rot i noden med värdet 8, se figur 48b. Det finns två barn och i detta fall är det subträd som redan heapordnats i tidigare steg. Båda subträden har ett värde i sin rot som är mindre än 8 (1 resp. 4). Värdet i det vänstra subträdet (1) är minst och det får byta plats med 8. Den nod som 8 hamnar i efter bytet är i detta fall inget löv, och proceduren upprepas för detta subträd. Det finns två barn med värdena 6 resp. 10, 6 är minst och mindre än 8, så 6 och 8 byter plats. Nu hamnar 8 i ett löv och subträdet är heapordnat, se figur 48c.

Vi har nu kommit till heapens rot (3). Den har två barn och båda har värden i sin rot som är mindre än 3 (1 och 2) men värdet i det vänstra subträdet (1) är minst och får byta plats med 3. Efter detta

byte ser heapen ut som i figur 48d. Värdena i de båda subträden till den nod som 3 hamnade i är i detta fall större än 3 (6 resp. 4) och inget byte ska göras. Hela heapen är nu ordnad.

Även i detta fall får ett hål vandra ner då *percolate-down* utförs, i stället för att successivt byta plats på två värden. Det värde som finns i roten av det subträd som behandlas tas först ut ur trädet och det hål som då uppstår i noden får vandrar ner i subträdet till dess värdet kan sättas in i heapordning.

Fördelen med att utföra *toss* och *fix-heap* i stället för *insert* är att heapen kan genereras på ett effektivare sätt. En analys visar att *toss* och *fix-heap* kan göras i linjär tid, dvs $O(n)$. *Toss* innebär att stega igenom fältet från början och placera in de n värdena, vilket uppenbart är $O(n)$. Värdena ska sedan heapordnas enligt beskrivningen ovan, vilket inte lika uppenbart är $O(n)$.

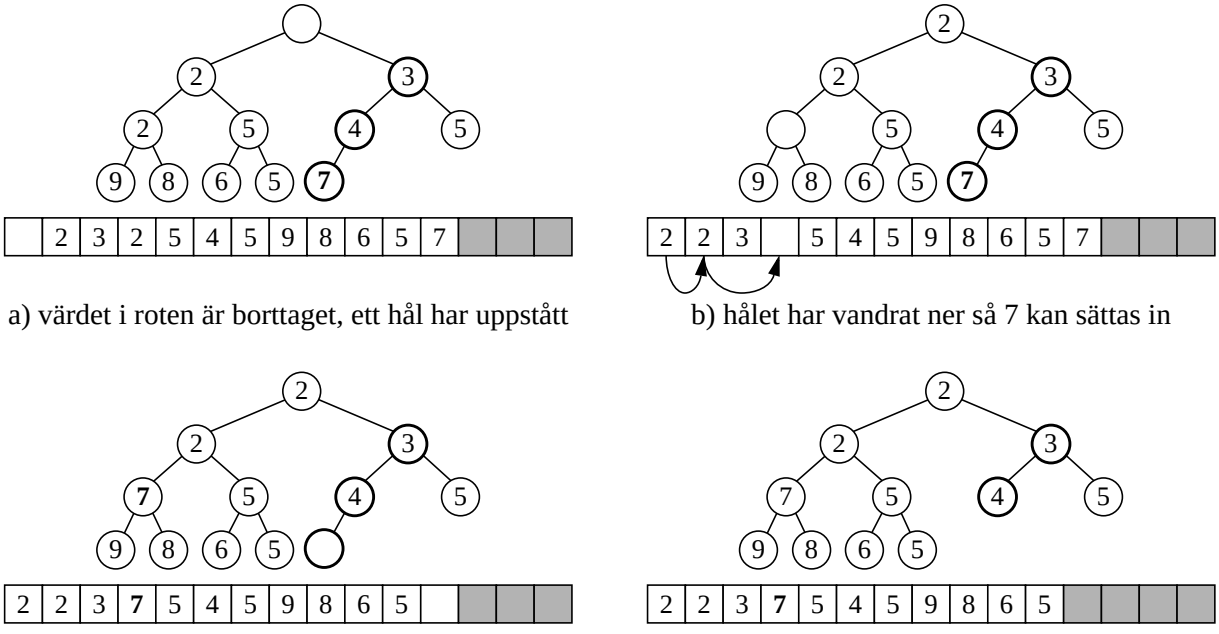
4.1.1.2 Borttagning ur binär min-heap

Borttagning ur en binär min-heap innebär att värdet i roten ska tas bort, *delete-min*. Ett hål uppstår då i roten och om det finns barn ska det med lägst rang eller något av barnen om de har lika rang, flyttas upp till roten. Detta upprepas för det subträd dit hålet flyttas. Samtidigt måste man tänka på att heapens storlek ska minskas med ett och att den nod som ska tas bort är det *sista lövet*, annars kommer inte heapen vara komplett efter borttagningen.

Efter att värdet i roten tagits bort flyttar man därför upp värdet i sista lövet (x) till hålet i roten och sedan upprepas följande.

- Om det finns två barn byter x och barnet med lägst rang plats, om detta har lägre rang än x. Har båda barnen samma rang och lägre rang än x byter något av barnen plats med x.
- Om det bara finns ett barn och det har lägre rang är än x, byter x och barnet plats.
- Om noden med x inte har några barn, har x hamnat i ett löv.

I praktikenkan man vänta med att flytta värdet i sista lövet till dess man hittat rätt plats för det i övriga heapen. I stället får hålet i roten vandra nedåt i heapen till dess värdet i sista lövet kan flyttas till hålet med bibehållen heapordning och först därefter tas noden som utgör sista lövet bort. I figur 49 visas *delete-min* med utgångspunkt från den heap som visas i figur 47 b.



Figur 49. Borttagning ur binär min-heap (delete-min).

5 Sortering

Sortering är vanligt i samband med lagring, bearbetning och presentation av data. Det finns många olika sorteringsmetoder och de kan kategoriseras på olika sätt.

Interna metoder sorterar i arbetsminnet och det är vanligtvis fråga om att sortera data som är lagrade i fält. Varje elementen i ett fält kan direkt kommas åt via ett index och det ger stora möjligheter att konstruera sorteringsmetoder på olika sätt. *Externa metoder* sorterar data som är lagrade på sekundärminnesfiler. Filerna läses sekventiellt och möjligheterna att variera konstruktionen av externa sorteringsmetoder är begränsad och det är i princip så kallad samsorteringsteknik som externa metoder bygger på.

I *jämförande metoder* jämförs två eller flera element i taget. Antingen flyttas ett eller flera element till en annan sekvens där de placeras i ordning eller så får elementen byta plats så att de komma i inbördes ordning. I *distributiva metoder* undersöks varje enskilt element för sig. och placeras med utgångspunkt från något kriterium i en av flera möjliga utdatasekvenser. Distributiva metoder har begränsningar men kan vara mycket snabba.

Sorteringsmetoder bygger vanligtvis på någon av de grundläggande teknikerna *urval* (*selection*), *insättning* (*insertion*), *utbyte* (*exchange*), *samsortering* (*merge*) *och distribution*.

En *stabil* sorteringsmetod bibehåller inbördes ordning på lika element, element med lika sorteringsnycklar, vilket kan vara viktigt då det är fråga om att sortera annat än enkla element och att sortera i olika avseenden vid olika tillfällen.

En *naturlig* metod lägger ner ett arbete som står i proportion till initialordningen hos indata, ju mer ordnade indata desto mindre arbete.

En *enkel metod* bygger på någon enkel, rättfram metod för att ordna data. Sådana metoder är enkla att implementera men ofta inte så effektiva. *Avancerade metoder* innebär att man använder mer komplicerade algoritmer, som är svårare att implementera men normalt betydligt effektivare, speciellt för större datamängder.

I vissa metoder byter element plats med varandra, vilket motsvarar tre förflyttningar. I vissa metoder flyttas element från en sekvens till en annan. När man analyserar metoder beräknar man typiskt antal *byten*/*jämförelser* och antal *förflyttningar* som behövs för att ordna data.

Avancerade metoder har typiskt tidskomplexiteten $O(n \cdot \log n)$, vilket också är en undre gräns för jämförande metoder, medan enkla metoder typiskt har tidskomplexiteten $O(n^2)$. Distributiva metoder kan i speciell fall ha tidskomplexiteten $O(n)$. För små datamängder kan enkla metoder vara mer effektiva än avancerade på grund av att de senare har en högre kostnad per delmoment, per *pass*. Avancerade metoder lönar sig vanligtvis genom att färre pass behöver utföras, till exempel ett, sett till datamängdens storlek, logaritmiskt i stället för linjärt antal gånger.

När man analyserar sorteringsmetoder brukar man studera tre fall avseende ordningen hos indata, *ordnade*, *omvänt ordande* och *slumpmässigt ordnade*. Dessa representerar vanligtvis också *bästa fallet*, *värsta fallet* och *allmäna fallet* för metoderna. Vilket som gäller för en specifik metod kan variera, till exempel kan ordnade indata vara värsta fallet för en viss metod.

En *inversion* är ett två godtyckliga element som är inbördes oordnade. Antalet inversioner är ett mått på graden av oordning i indata. Sekvensen [5, 3, 6, 1, 4, 2] har 10 inversioner, (5, 3), (5, 1), (5, 4), (5, 2), (3, 1), (3, 2), (6, 1), (6, 4), (6, 2) och (4, 2).

I många enkla interna metoder delas datamängden upp i två delar. I den ena finns de ännu oordnade elementen, *källsekvens*. I den andra delen finns element som antingen är slutgiltigt sorterade eller sorterade inbördes men ännu inte slutgiltigt, *destinationssekvens*.

Många interna metoder behöver inte mer minne än antalet element som ska sorteras, plus en hjälpvariabel för att kunna byta plats på element. Det finns dock metoder som kräver dubbelt eller tre gånger så mycket minne som det finns element.

När man ska välja metod får man ta hänsyn till exempelvis antalet element som ska sorteras, om det kan förväntas en viss ordning i indata, om sorteringen måste vara stabil, etc.

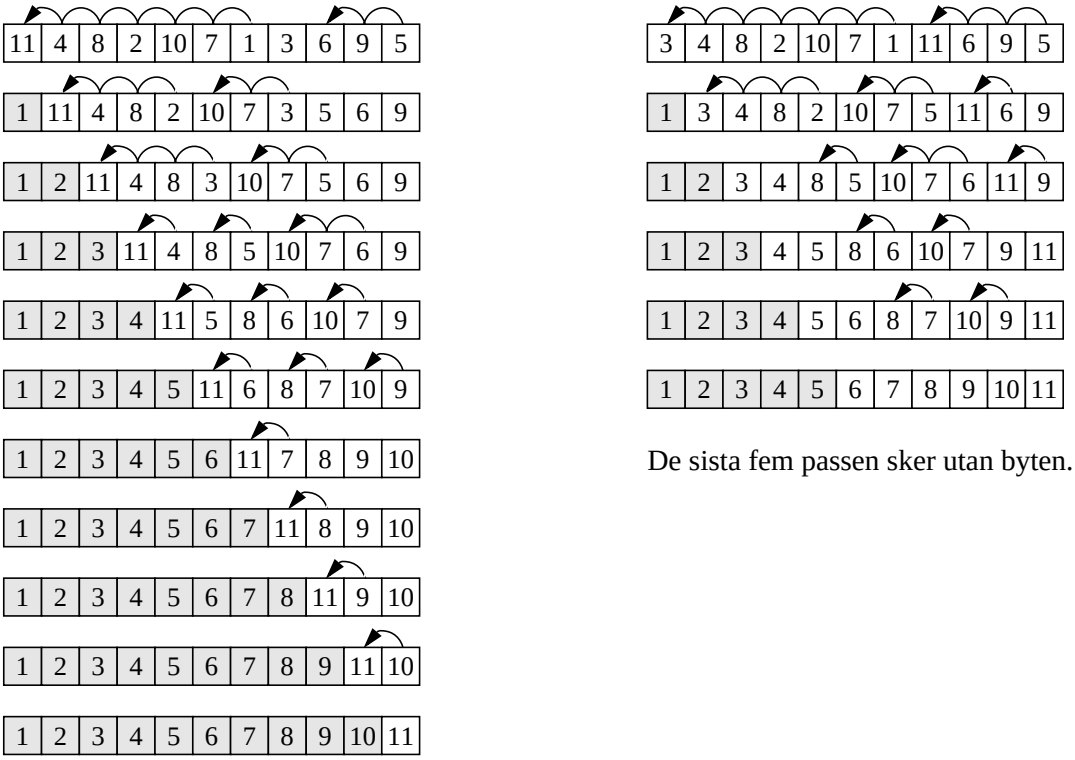
5.1 Interna sorteringsmetoder

De metoder som vanligtvis används för att sortera data internt är jämförande metoder. Det finns ett stort antal sådana metoder som bygger på olika tekniker där det kan finnas både enkla och avancerade variationer på en viss teknik. Även distributiva metoder kan dock komma ifråga. De element som ska sorteras lagras vanligtvis i fält.

5.1.1 Bubblesort

Bubblesort är en *enkel utbytesmetod*. Den bygger på att *n*-1 pass utförs på de *n* elementen i ett fält. Element på direkt intilliggande positioner jämförs parvis och får byta plats om de inte ligger i ordning. Under det första passet kommer det minsta elementet att flyttas, ”bubbla”, till sin slutgiltiga plats och behöver sedan inte behandlas mer. I det andra passet kommer det näst minsta elementet att hamna på sin plats. Efter *n*-1 pass är de *n* elementen i fältet sorterade.

I figur 50 visas två exempel på hur bubblesort arbetar. Den första raden visar elementens initialordning och vilka byten som sker i första passet. Den andra raden visar ordningen efter första passet och vilka byten som görs i det andra passet, osv. I det vänstra exemplet krävs samtliga *n*-1 pass innan alla element kommit på plats. Det största elementet, 11, flyttas bara en position i taget mot sin slutgiltiga plats. I det högra exemplet är samtliga element är på plats redan efter fem pass och återstående fem pass genomförs utan att några byten sker, enbart jämförelser.



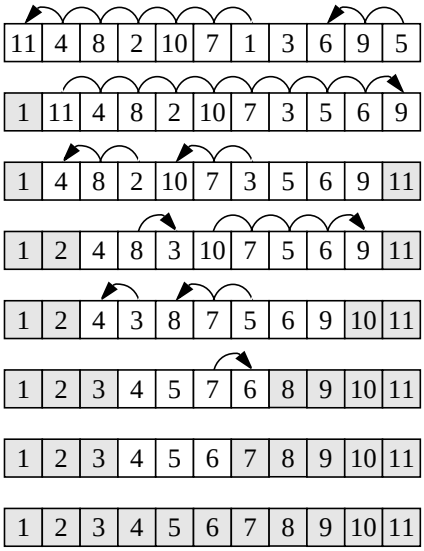
Figur 50. Två exempel på hur Bubblesort arbetar.

En analys av Bubblesort ger att samtliga *n*-1 pass alltid genomförs. I det *i*:te passet utförs *n*-i jämförelser, i bästa fall inga byten, i värsta fall *n*-i byten. Detta ger en tidskomplexitet som är $O(n^2)$, Bubblesort är en kvadratisk metod. Ett värsta fall är om data ligger i helt omvänd ordning, vilket innebär att det i varje pass kommer att utföras maximalt antalet byten.

Man skulle kunna upptäcka om ett pass har genomförts utan byten och avbryta. Det skulle reducera antalet pass och därmed antalet jämförelser men däremot inte antalet byten. Man skulle också kunna notera var sista bytet i ett pass gjordes och flytta gränsen för den färdigsorterade delen med dit direkt. Det skulle reducera antalet jämförelser i ett pass men inte påverka antalet byten.

5.1.2 Shakersort

Som framgår av exemplet i figur 50 finns en asymmetri i Bubblesort. Ett litet element ”bubblar” direkt till sin slutgiltiga plats, medan ett stort element bara flyttas ett steg per pass. Genom att köra varannat pass i motsatt riktning skulle man kunna åtgärda den asymmetrin. Denna variant kallas Shakersort. I Shakersort brukar man också noterar var det sista bytet i ett pass görs och flytta gränsen mellan färdigsorterade och osorterade element dit direkt.

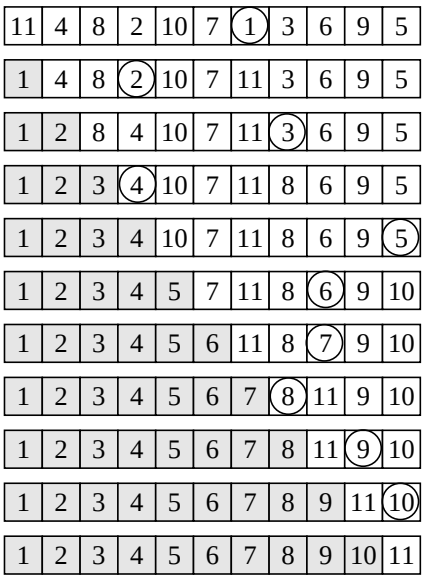


Figur 51. Shakersort.

5.1.3 Selectionsort

Selectionsort är en *enkel urvalsmetod* och representerar det kanske mest intuitiva sättet att sortera. I selectionsort utförs *n*-1 pass om antal element som ska sorteras är *n*. I det första passet genomsöks hela fältet och positionen för det minsta elementet sparas. Det minsta elementet får sedan byta plats med det element som låg först. Destinationssekvensen utgörs nu av detta minsta element. I det andra passet söks det näst minsta elementet i återstoden av fältet och placeras näst först i fältet genom att det får byta plats med det element som låg där, osv. I figur 52 visas hur detta successivt går till för de *n*-1 passen.

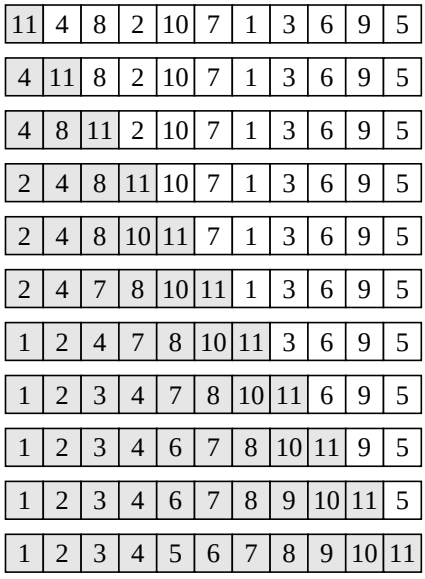
En tilltalande egenskap hos selectionsort är att det i varje pass bara görs ett byte, dvs antalet byten totalt är $O(n)$. Antalet jämförelser i det *i*:te passet är däremot alltid maximala *n*-*i*. Med avseende på jämförelser är selectionsort en $O(n^2)$-metod och klassificeras därmed som en kvadratisk metod.



Figur 52. Selectionsort.

5.1.4 Insertionsort

Insertionsort är en *enkel insättningsmetod*. När sorteringen börjar utgör det första elementet destinationssekvens och övriga källsekvens. Sedan utförs *n*-1 pass, där i varje pass det första elementet i källsekvensen ordnas in bland elementen i destinationssekvensen. Destinationssekvensen är alltså i detta fall inte slutgiltigt ordnad, som i bubblesort och selectionsort.



Figur 53. Insertionsort.

En variation på insertionsort är *linjär insättning*. Man flyttar först elementet som ska ordnas in till en hjälpvariabel. Sedan gör man en linjärsökning i destinationssekvensen från höger och varje element som är större än det som ska sättas in flyttas en position åt höger. Man flyttar alltså ett ”hål” till den position där elementet som ska ordnas in ska finnas ”hålet”.

I det *i*:te passet i linjär insättning görs mellan 1 och *i* stycken jämförelser. Att flytta elementet som ska sättas in till hjälpvariabeln och sedan till ”hålet” innebär två förflyttningar. Om elementet ifråga är större än eller lika med det sista i destinationssekvensen, dvs det ska inte flyttas, görs inga fler förflyttningar. I annat fall görs mellan 1 och *i* förflyttningar av ”hålet” in i destinations-

sekvensen. Den förväntade sökvägen är halva destinationssekvensen, dvs det förväntade antalet jämförelser och förflyttningar i det *i*:te passet vid linjär insättning är $O(n)$. Antalet pass är $O(n)$ och linjär insättning följaktligen en $O(n^2)$-metod.

Eftersom destinationssekvensen är ordnad kan man använda binärsökning för att hitta den plats där nästa element ska placeras in. Denna variant av insertionsort kallas *binär insättning*. Detta innebär att sökningen i destinationssekvensen förbättras till $O(^2\log n)$, i stället för O(n). Antalet förflyttningar påverkas inte, så även binär insättning är en $O(n^2)$-metod. Man kan notera att för ordnade eller nästan ordnade indata är linjär insättning ett bättre val bättre än binär insättning.

5.1.5 Shellsort

Shellsort har fått sitt namn efter sin upphovsman, Donald Shell. Det är en avancerad insättningsmetod som bygger på idén att det borde vara effektivare att flytta element över länge avstånd än ett för att elementen snabbare ska komma till sin slutliga position.

Shellsort har subkvadratisk tidskomplexitet. Den finns i ett antal varianter som är mer eller mindre svåra eller rent av omöjliga att analysera formellt. Empiriskt beter sig vanligtvis Shellsort som om den har en tidskomplexitet som är $O(n\cdot^2\log n)$ eller $O(n^{3/2})$.

Implementeringen av Shellsort omfattar bara ett par rader kod mer än insertionsort. Shellsort är dessutom iterativ, till skillnad från många andra avancerade metoder, vilka är rekursiva.

I Shellsort kommer ett antal pass att genomföras. I varje sådant pass delas elementen upp i delsekvenser vars element finns på ett visst inbördes avstånd, *gap*. I första passet används ett stort gap och för varje efterföljande pass minskas gapet successivt för att i sista passet alltid vara 1. Valet av gap är väsentligt och det är detta val som skiljer de olika varianterna av Shellsort åt. Ett annat namn på Shellsort på engelska är *diminishing gap sort*.

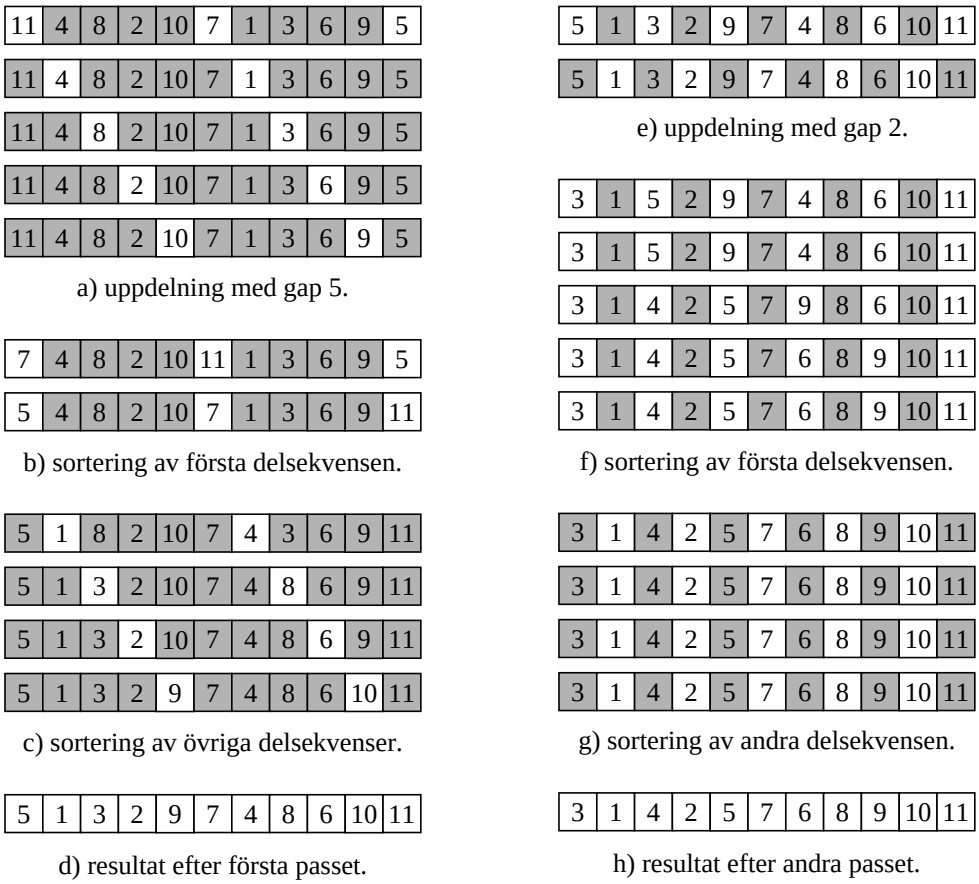
I varje pass utförs insättningssortering på varje delsekvens. Principen är densamma som för enkel insättningssortering (gap 1). Elementen i källsekvensen ordnas i tur och ordning in i destinationssekvensen men det sker alltså på ett avstånd som är lika med gapet för passet i fråga.

I den ursprungliga varianten av Shellsort väljs det första gapet som $n/2$. Sedan minskas gapet successivt i de efterföljande passen, genom att gapet för föregående pass halveras. Detta brukar kallas *Shells gapsekvens*.

I figur 54 har vi 11 element som ska sorteras. Med Shells variant blir första gapet 5, det efterföljande 2 och sist 1. Det innebär att det blir tre pass men i figuren visas endast de två första, det sista passet är en vanlig Insertionsort.

Figur 54a visar de fem delsekvenser som uppstår för gapet 5. Figur 54b visar hur den första delsekvensen av dessa successivt sorteras med insertionsort. Tre element ger två pass, där först 7 ordnas med 11 och sedan 5 med 7 och 11. De övriga fyra delsekvenserna omfattar endast två element vardera och det medför endast ett pass i insättningen, vilket visas i figur 54c. Resultatet efter att de fem delsekvenserna för gapet 5 sorterats inbördes visas i figur 54d.

I figur 54e visas de två delsekvenser som uppstår för gapet 2. Den första innehåller 6 element och det ger de fem insättningspass som visas i figur 54f. Den andra delsekvensen innehåller fem element och för dessa genomförs de fyra insättningspass som visas i figur 54g (elementen råkade ligga i ordning, så inget förändrades). Resultatet efter andra huvudpasset för gapet 2 visas i figur 54h. Efter detta genomförs ett tredje och sista huvudpass med gapet 1, men det är ju en vanlig Insertionsort så det visar vi ej här.



Figur 54. Shellsort med gapen n {1, 2, 5}. Sista passet med gap 1 visas ej (vanlig insertionsort).

För att Shellsort ska fungera väl är det väsentligt att gapsekvensen gör att elementen från olika delsekvenser i ett pass blandas så mycket som möjligt i delsekvenserna i efterföljande pass, för att minimera jämförelse av element som redan har jämförts.

Man kan visa att i värsta fall kan Shellsort vara $O(n^2)$. Detta inträffar för Shells gap om antalet element som ska sorteras är en jämn multipel av två, alla stora element finns på jämna index och alla små element finns på udda index. I det allmäna fallet kan man visa att om antalet element är en multipel av 2 är tidskomplexiteten $O(n^{3/2})$, vilket är en avsevärd förbättring jämfört med Insertionsort.

En mindre modifiering av gapsekvensen kan förhindra det kvadratiska beteendet. Om gapet efter division med 2 blir ett jämnt värde, adderas 1 för att göra det udda. Man kan då visa att tidskomplexiteten i värsta fallet är $O(n^{3/2})$. Det allmäna fallet är okänt men vid experiment tycks det vara $O(n^{5/4})$.

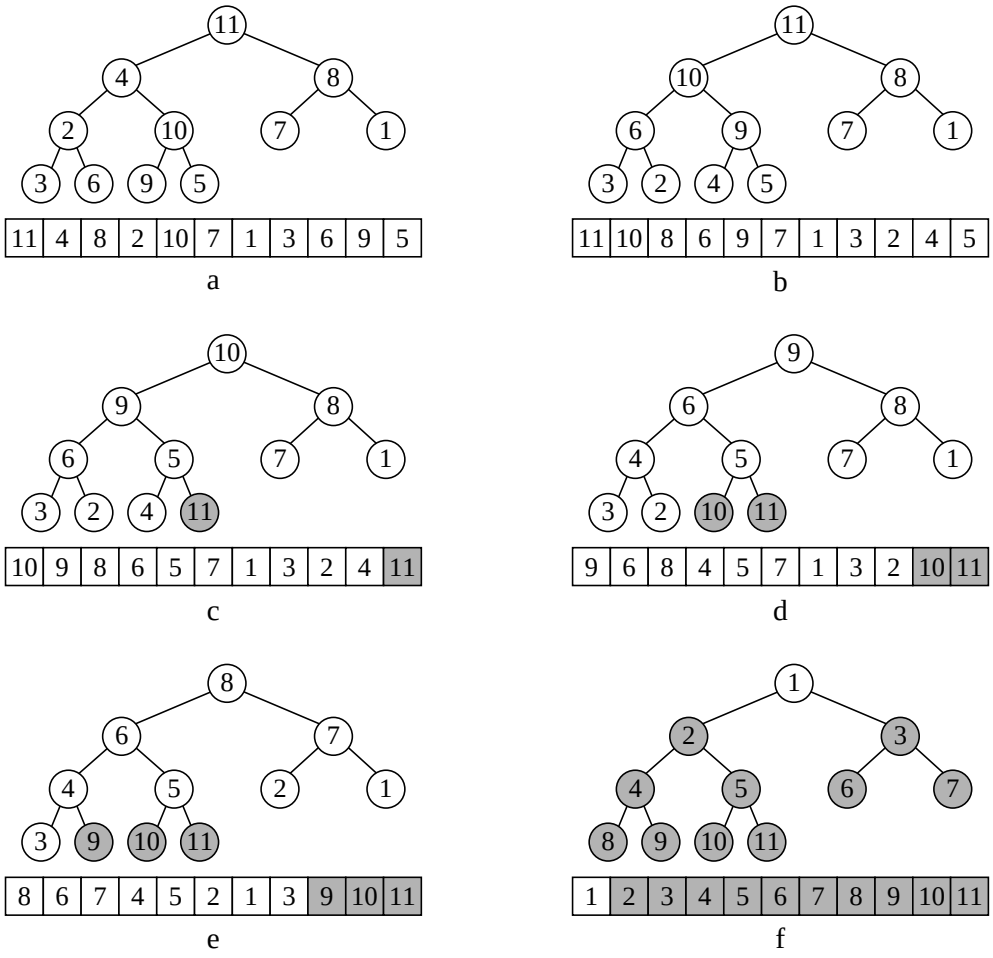
En annan modifiering är att dividera föregående gap med 2,2 i stället för 2. Det finns ingen teoretisk grund för detta men det fungerar väl i praktiken. Tidskomplexiteten förefaller hamna under $O(n^{5/4})$, eventuellt $O(n^{7/6})$. I detta fall måste man se upp med är att gapet 1 inte missas. Detta kallas för *Gonnets increment*, efter upphovsmannen.

Andra välkända gapsekvenser för Shellsort är *Hibbards* {1, 3, 7…, 2^k-1}, *Knuths* {1, 4, 13,…, $(3^k-1)/2$} och *Sedgewicks* {1, 5, 19, 41, 109,…}, där varje term är på formen $9 \cdot 4^k - 9 \cdot 2^k + 1$ eller $4^k - 3 \cdot 2^k + 1$.

5.1.6 Heapsort

Heapsort är en avancerad intern metod som utnyttjar en prioritetskö. Genom att placera elementen som ska ordnas i en binär min-heap och sedan utföra delete-min *n* gånger erhålls elementen i stigande ordning. Detta skulle kräva dubbelt så mycket minne som antalet element om heapen är en separat struktur men det kan undvikas.

Genom att heapstorleken minskar varje gång ett element tas ut och det friställda utrymmet alltid är sist i heapen kan man sortera på plats. I detta fall används en max-heap. Elementet som ska tas ut får byta plats med elementet i sista lövet, heapstorleken minskas med ett och heapordningen återställs. Detta upprepas till dess heapen har minskats i storlek till ett. I figur 55 visas delar av detta förlopp.



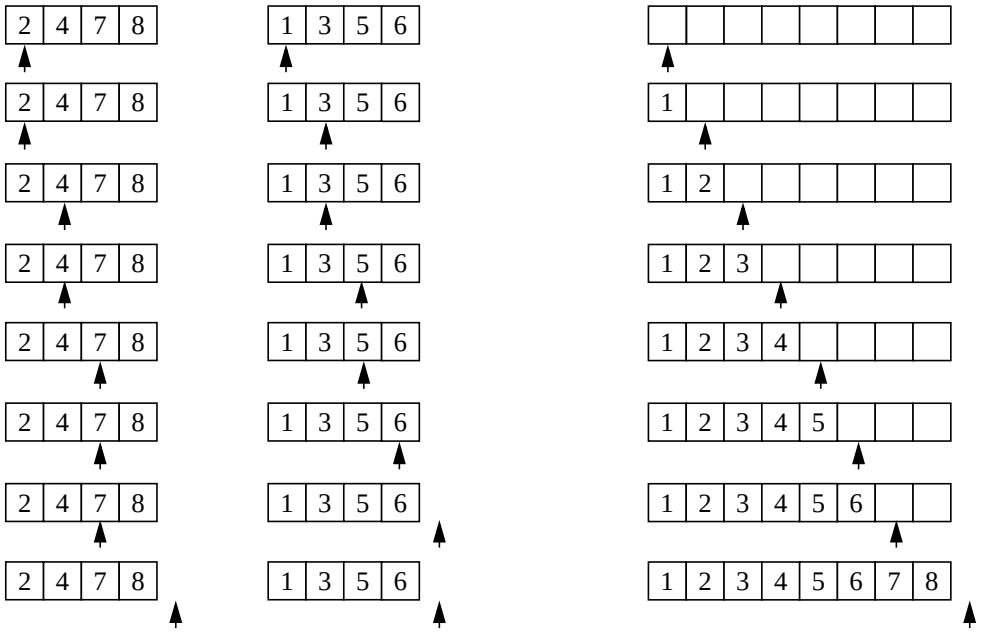
Figur 55. Heapsort.

Figur 55a visar de osorterade elementen efter att de placerats i heapen, dels hur vi ser på heapen logiskt, som ett träd, dels hur elementen lagras fysiskt, i ett fält. I figur 55b visas läget efter att heapen ordnats som en max-heap. Figur 55c-e visar tre successiva pass, där i respektive pass rotelementet och elementet i sista lövet byter plats (11 och 5 i figur 55b), heapens storlek minskas med ett (noden/elementet skuggas) och till slut hur heapordningen återställs (i figur 55c har elementet 5 flyttats ner till position 5, varvid elementet 10 och 9 flyttas upp). Figur 55f visar läget efter ytterligare sex pass, då heapstorleken har reducerats till ett och sorteringen därmed är klar.

5.1.7 Mergesort

Mergesort är en intern sorteringsmetod som bygger på *samsortering*, en teknik som normalt används för extern sortering. Mergesort tillhör en kategori av metoder som betecknas som söndra-och-härska-metoder, för vilka man kan visa att de har en tidskomplexitet som är $O(n \cdot \log n)$. Åtminstone teoretiskt tillhör därmed Mergesort de snabbare metoderna.

I figur 56 visas principen för samsortering av två redan sorterade sekvenser, med fyra element vardera. I källsekvenserna markerar pilarna positionen för det första av de återstående elementen i respektive sekvens. I utdatasekvensen markerar pilen den position där nästa element ska placeras.



Figur 56. Samsortering av två sorterade fält.

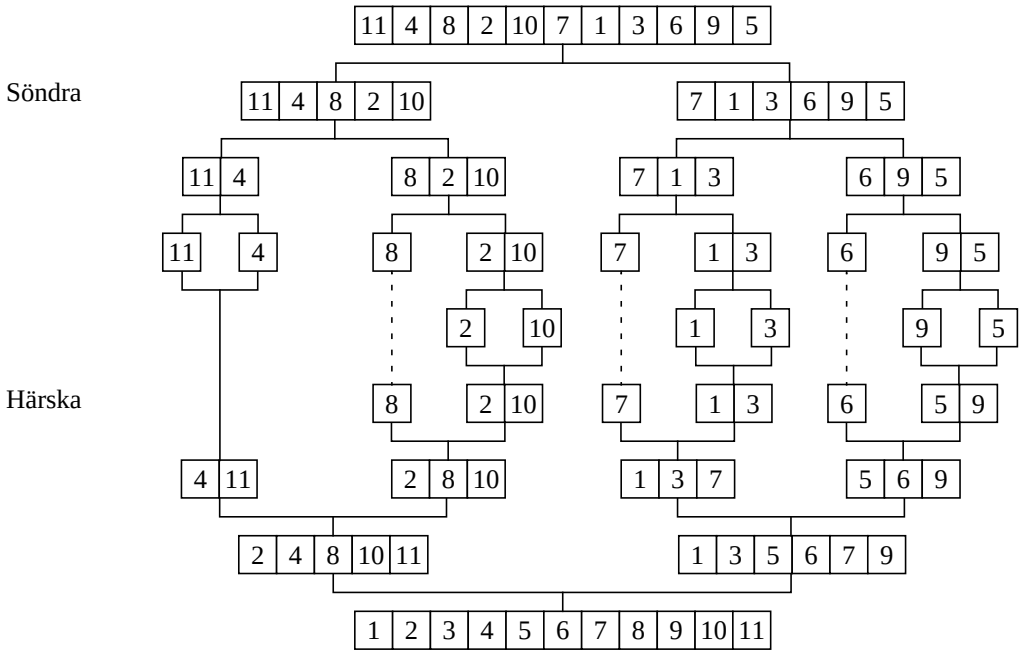
Den första raden i figur 56 visar utgångsläget. De första elementen i de två källsekvenserna ska jämföras och det minsta flyttas till destinationssekvensen. Den andra raden visar läget efter att det första elementet, 1, har flyttats till utdatafältet.

Sedan visas, ner till näst sista raden, hur ett element i taget flyttas från någon av källsekvenserna till destinationssekvensen. Den näst sista raden visar läget efter att elementet 6 har flyttats till destinationssekvensen och den högra källsekvensen därmed har tömts. De två återstående elementen i den vänstra källsekvensen, 7 och 8, kan nu flyttas till destinationssekvensen utan att några jämförelser görs.

I figur 57 visas principen för Mergesort. Datamängden halveras rekursivt till dess endast ett element återstår i varje delsekvens. Därefter samsorteras delsekvenserna parvis, enligt den princip som visats i figur 56 ovan, till dess destinationssekvensen utgörs av alla element.

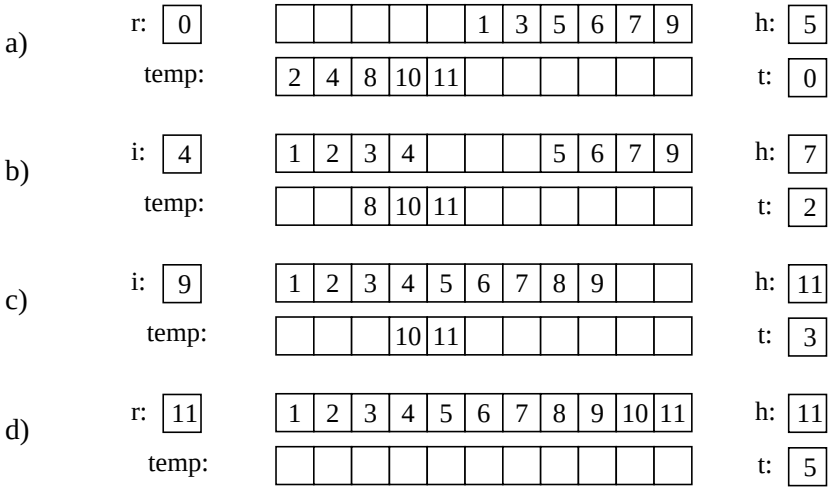
Djupet i delningsförfarandet bestäms av hur många gånger som n kan delas med 2 och fortfarande vara större än eller lika med 1, dvs $^{2}\log n$. På varje delningsnivå måste varje element samsorteras i sitt delfält, vilket innebär O(n) jämförelser och $O(n)$ förflyttningar. Mergesort är alltså en $O(n \cdot \log n)$-metod.

En nackdel med Mergesort är att sorteringen inte kan utföras ”på plats”, utan det krävs dubbelt så mycket minne som det finns element som ska sorteras. Detta minne kan skapas då sorteringen inleds och återlämnas då sorteringen är klar, se figur 58 nedan. En fördel med Mergesort är att den är enkelt och naturligt kan göras stabil. Det är vanligt att Mergesort används i programbibliotek som bibliotekens stabila sorteringsalgoritm, under exempelvis namn som ”stable_sort”.



Figur 57. Principen för Mergesort.

I figur 58 visas några steg då de två sista delsekvenserna i figur 57 ovan samsorteras. temp är ett hjälpfält, r är index till nästa lediga position i destinationssekvensen, h är index för nästa element i den högra källsekvensen, t är index för nästa element i temp, den vänstra källsekvensen.



Figur 58. Fyra steg i samsortering av två sorterade delsekvenser.

I figur 51a visas situationen efter att den vänstra källsekvensens element har flyttats till temp. Figur 51b visar läget efter att fyra element har samsorterats. I figur 51c har nio element samsorterats och den högra källsekvensens element har tagit slut. Figur 51d visar slutresultatet, efter att de återstående elementen i temp flyttats till destinationssekvensen.

5.1.8 Quicksort

Quicksort är en avancerad utbytesmetod. Den bygger på idén att, eftersom sortering går ut på att flytta varje element till sin slutliga position, vore ett steg i rätt riktning att flytta *ett* element, *x*, till sin slutliga position. Ett sätt att hitta slutpositionen för *x* vore att flytta om elementen, så att alla element som är mindre än *x* ska hittas till vänster om *x* och alla elementet som är större än *x* ska hittas till höger om *x*. När det är gjort kan förfarandet upprepas rekursivt på elementen till vänster respektive till höger om *x*, till dess endast ett eller inget element kvarstår i varje del. Det element som bestämmer uppdelningen, *x*, kallas i fortsättningen för *pivot* och de delar som uppstår för *partitioner*.

Quicksort är en söndra-och-härska-metod. Den delar successivt delar upp elementen i partitioner som sedan behandlas var för sig. Till skillnad från Mergesort utförs förflyttningarna i samband med uppdelningen, så när uppdelningen är klar är också sorteringen klar. En annan skillnad jämfört med Mergesort är att delning inte görs exakt i mitten, utan det beror på *pivot*. I värsta fall kan delningsdjupet blir linjärt och inte som önskat logaritmiskt. Detta inträffar om pivot i varje delning skulle råka väljas som det mista eller största elementet. Tidskomplexiteten för Quicksort blir i så fall $O(n^2)$. I allmänhet är dock Quicksort en $O(n \cdot^2 \log n)$-metod och en mycket snabb sådan.

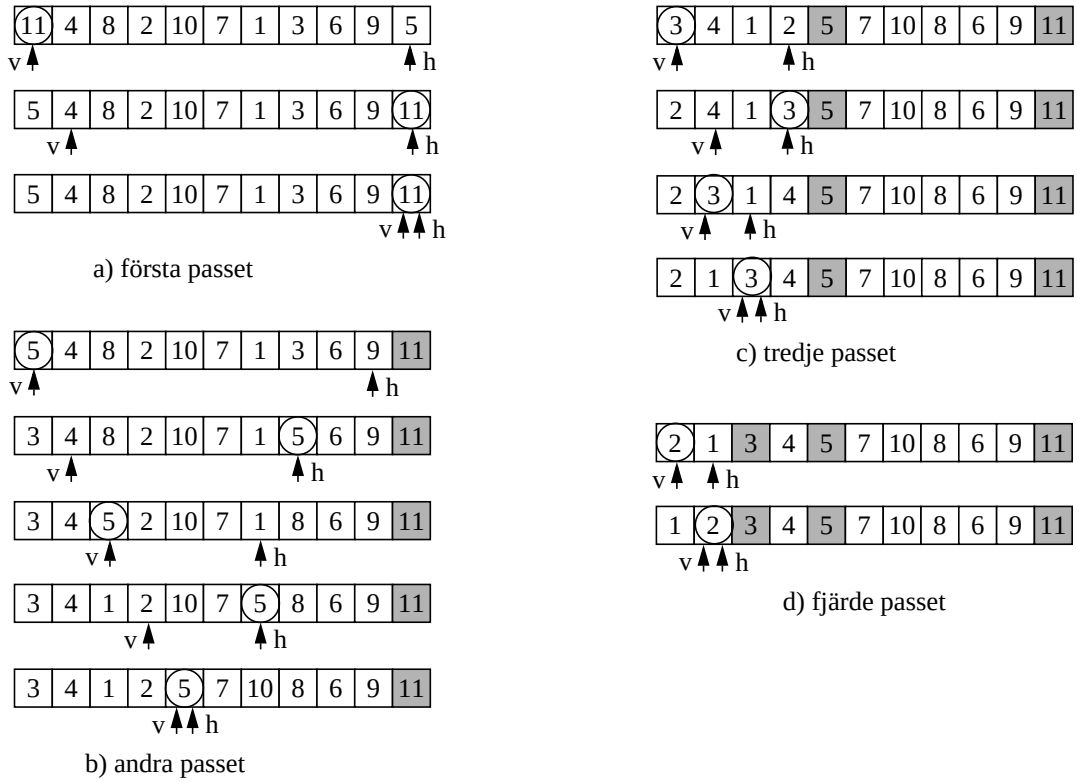
Quicksort kan varieras på flera sätt. Gemensamt är att två index används för att stega igenom och dela upp elementen i en partition. Ett index stegas från vänster till höger och ett stegas från höger till vänster. När indexen möts är uppdelningen klar, positionen för *pivot* bestämd och därmed också uppdelningen inför nästa steg.

Valet av *pivot* är väsentligt. Ett möjlighet är att välja det första elementet i den aktuella partitionen. Om data redan är ordnade eller har en förhållandevis hög grad av ordning kan dock detta val innebära att uppdelningen urartar eller blir påtagligt skev. Ett bättre val skulle vara att välja mitt-elementet, för att eliminera risken att drabbas av ett degenererat beteende. För varje deterministisk delningsteknik finns dock alltid ett degenererat fall. Det optimala valet av *pivot* vore *medianen* men då måste elementen först sorteras. En mer rimlig lösning är att välja det första, det mittersta och det sista elementet i partitionen, beräkna medianen av dessa tre och använda som pivot. Hur man än gör ska pivot vara ett element som ingår i partitionen.

I figur figur 59 visas ett exempel på sortering med en enkel variant av Quicksort. *Pivot* väljs i detta fall som det första elementet i den aktuella partitionen och får vara kvar bland övriga element då uppdelningen görs. Delfigur, a-d, i figur 59 visar de fyra första passen, vilket leder fram till att rekursionen bottnar för första gången, i den vänstraste partitionen. Algoritmen för partitioneringen är följande.

1. det första elementet i partitionen väljs som *pivot*
2. v sätts till det första elementet i partitionen och h till det sista elementet
3. så länge v är mindre än h och *pivot* är mindre än elementet i h, ska h stegas ner.
4. om v är mindre än h ska elementen i h och v byta plats, v ska stegas upp med 1, fortsätt sedan med punkt 5, annars är uppdelningen klar och *pivot* på plats (i v)
5. så länge v är mindre än h och elementet i v är mindre än *pivot* ska v stegas upp.
6. om v är mindre än h ska elementen i h och v byta plats, h ska stegas ner med 1, fortsätt sedan med punkt 1, annars är uppdelningen klar och *pivot* på plats (i h)

Ovanstående upprepas så länge det finns fler partitioner att behandla med fler än ett element.



Figur 59. Quicksort.

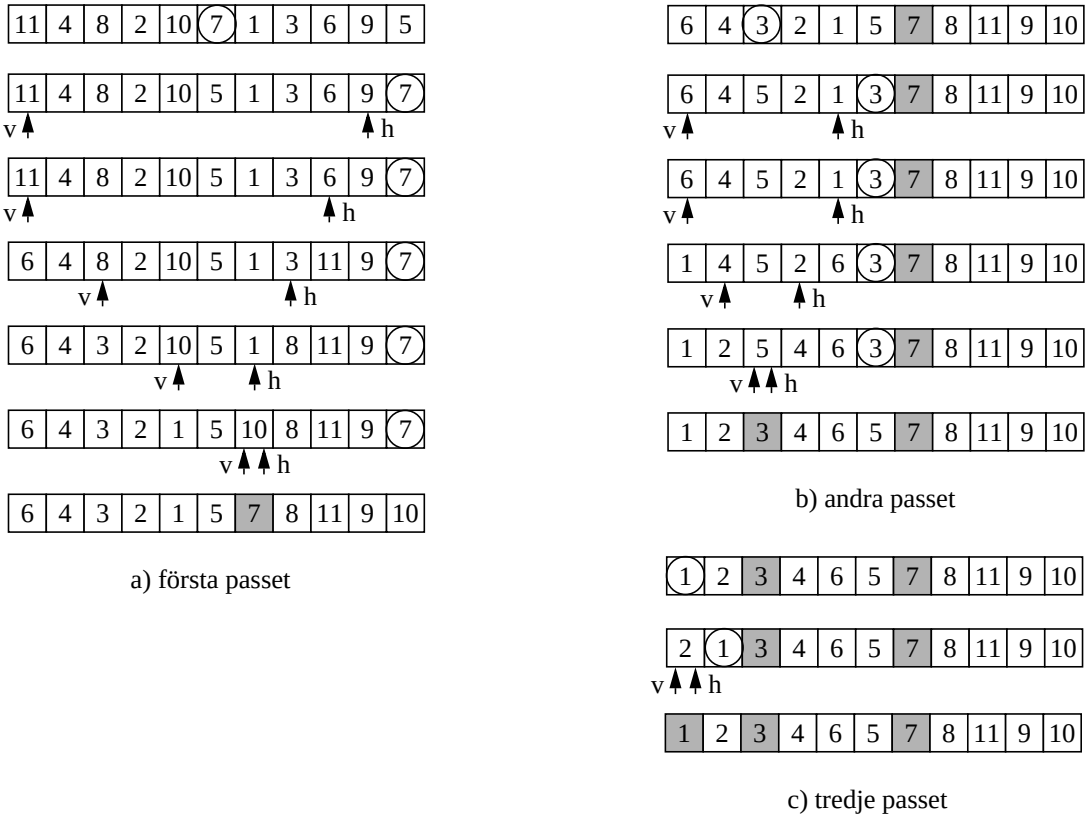
I figur 59a visas det första passet i tre delfigurer. Efter det första passet har alltså alla element utom *pivot* hamnar i den vänstra partitionen, den högra blev tom och vi fick ett degenererat fall. I figur 59b visas hur den vänstra partitionen från figur 59a delas upp. Elementet 5 blir *pivot* i detta fall och efter fyra erhålls en vänsterpartition med fyra element, {3, 4, 1, 2}, och en högerpartition med 5 element, {7, 10, 8, 6, 9}. I figur 59c behandlas vänsterpartitionen från figur 59 b, vilket ger en vänsterpartition med 2 element, {2, 1}, och en högerpartition med endast ett element, {4}. I figur 59d visas behandlingen av den vänstra partitionen från figur 59c. Vänsterpartition innehåller bara ett element, vilket kommer att avbryta rekursionen för denna del i nästa steg. Ännu obehandlade partitioner till vänster i den andra uppdelningen (5 som pivot) är antingen tomma eller innehåller endast ett element. Därmed är denna del klar och elementen ordnade, {1, 2, 3, 4}. Samma procedur ska nu utföras på den högra partitionen i den andra delningen, dvs för {7, 10, 8, 6, 9}.

I en del partitioneringstekniker flyttas *pivot* ut ur partitionen (göms), sedan delas övriga element upp och slutligen flyttas *pivot* in till sin rätta plats. *Pivot* flyttas exempelvis längst ut till höger i den aktuella partitionen genom att byta plats med det elementet. Delfigur a-c i figur 60 visar de tre första passen i en sådan Quicksort. Algoritmen för partitioneringen är följande:

1. välj mittelementet som *pivot*
2. byt plats med det högraste elementet, v sätts till det första elementet i partitionen och h till det näst sista
3. så länge v är mindre än h och elementet i v är mindre än *pivot* stegas v upp
4. så länge v är större än h och elementet i h är större än *pivot* stegas h ner
5. om v < h ska elementen i v och h byta plats och proceduren upprepas från punkt 3, annars är
6. partitioneringen klar, *pivot* och elementet i v ska byta plats

Ovanstående upprepas så länge det finns fler partitioner att behandla med fler än ett element.

I figur 60a visas i första delfiguren att mittelementet, 7, väljs som pivot. I nästa delfigur har pivot bytt plats med elementet längst till höger och indexen *v* och *h* har givits sina startpositioner. Tredje delfiguren visar vilka positioner v och h anger efter att både v och h har stegats första gången, 11 ska byta plats med 6. I fjärde delfiguren har 11 och 6 bytt plats och v och h har stegats till nästa par som ska byta plats, 8 och 3. Femte delfiguren visar läget efter att 8 och 3 har bytt plats och indexen stegats. I sjätte delfiguren har 10 och 1 bytt plats och indexen stegats. I och med det är uppdelningen klar och pivot byter plats med elementet i position v.



Figur 60. Quicksort med undangömd pivot.

Ett lite mer avancerat sätt att bestämma *pivot* är att välja medianen av det första, mittersta och sista elementet i partitionen. Genom att ordna dessa tre element inbördes i fältet, får man medianen i mitten och de andra elementen hamnar i den halva de ska vara även efter uppdelningen och de behöver alltså inte ingå i själva uppdelningen. Pivot göms undan genom att byta plats med elementet *näst längst till höger* i partitionen. Detta kallas *medianen-av-tre-uppdelning*.

Dessutom visas i detta exempel att rekursionen inte alltid körs i botten, utan avbryts då en viss minimistorlek på partitionerna uppnås, så kallad *cutoff*. Partitionerna slutsorteras med någon enkel metod, till exempel Insertionsort: Detta utnyttjar förhållandet att enkla metoder är effektivare än avancerade för små datamängder. Empiriska försök har visat att *cutoff* upp till 20 element ger bra resultat. Använder man *medianen-av -tre-uppdelning* är det minsta meningsfulla antalet element i en partition fyra, så *cutoff* lika med tre är i det fallet det minsta tänkbara.

Algoritmen för partitioneringen är i grunden densamma som föregående men med tillägg för cutoff-hantering och medianen-av-tre-beräkning av *pivot*:

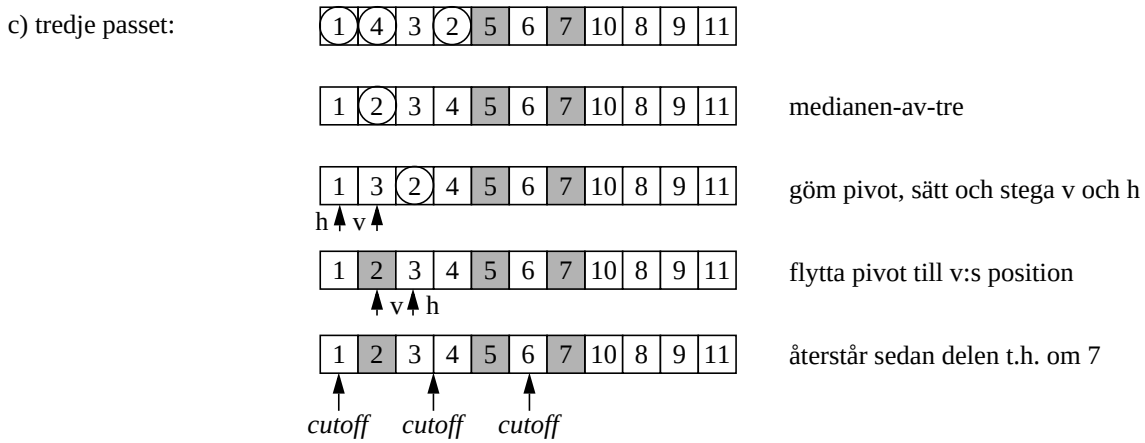
1. om antalet element är mindre än eller lika med *cutoff* ska partitionen inte delas upp ytterligare, gå annars till punkt 2
2. ordna elementen i första, mittersta och sista positionen i partitionen inbördes, medianen av dessa hamnar i mittpositionen och väljs som *pivot*

3. byt plats på *pivot* och det andra elementet från höger i partitionen, v sätts till det första elementet i partitionen och h till det tredje elementet från höger i partitionen
4. så länge elementet i v är mindre än *pivot* stegas v upp (*pivot* fungerar som vaktpost)
5. så länge elementet i h är större än *pivot* stegas h ner (elementet längst till vänster är vaktpost)
6. om v < h ska elementen i v och h byta plats och proceduren upprepas från punkt 4, annars är
7. partitioneringen klar, *pivot* och elementet i v ska byta plats

Upprepa från punkt 1 ovan så länge det finns obehandlade partitioner.

I figur 61 nedan visas ett exempel på Quicksort med medianen-av-tre-uppdelning och *cutoff* vid 3. *Cutoff* vid 3 innebär att det ska finnas minst fyra element kvar i en partition för att ytterligare en partitionering ska genomföras.

Figur 61. Quicksort med medianen-av-3-partitionering och cutoff vid 3.



Figur 61 (forts). Quicksort med medianen-av-3-partitionering och cutoff vid 3.

Efter tredje uppdelningen i figur 61 erhålls *cutoff* i partitionerna till vänster och till höger om 2 och sedan i den högra partitionen till den delning som hade 5 som pivot. Återstår behandling av partitionen till höger om den första delningen vid 7, {10, 8, 9, 11}.

5.2 Distributiva sorteringsmetoder

Den grundläggande idén bakom distributiva metoder är att data som ska sorteras kan placeras olika destinationssekvenser med hjälp av indexering, utan att det förekommer några jämförelser nycklar emellan. En förutsättning för att detta slags metoder ska kunna användas är att nycklar antingen direkt eller genom någon bearbetning kan användas som index. Är detta möjligt behöver indata endast läsas igenom för att utföra sorteringen och tidskomplexiteten blir alltså $O(n)$.

Att data på något vis ska kunna användas som index, innebär naturligtvis en påtaglig begränsning och detta slags metoder är inte så vanliga men värda att känna till.

5.2.1 Count Sort

Count Sort, även kallad *Pigeonhole Sort* eller *Bucket Sort*. Genom att ha räknare, där varje värde som kan förekomma motsvaras av en egen räknare, kan de sorterade talen genereras ur räknarna.

Om heltal ska sortera heltal läses talen igenom och respektive tal används som index i en frekvenstabell där det i varje positionen finns en räknare som stegas upp. Sedan går man igenom tabellen och för varje position återskapas talet lika många gånger som räknaren anger.

Arbetet med att läsa igenom indata är proportionellt mot antalet element, så tidskomplexiteten för detta är $O(n)$. Att gå igenom frekvenstabellen och generera varje tal lika många gånger som dess räknare anger är också en aktivitet med tidskomplexiteten $O(n)$, och *Count Sort* är följaktligen en $O(n)$-metod. Detta är bättre än vad någon jämförande metod kan åstadkomma i normalfallet, $O(n \cdot \log n)$ men det finns påtagliga begränsningar.

En begränsning för Count Sort som redan nämnts är att de element som ska sortera måste kunna användas som index, direkt eller genom någon bearbetning. Om värdeintervallet för de data som ska sorteras är mycket stort krävs mycket minne för att lagra frekvenstabellen och om antalet element som ska sorteras påtagligt underskrider storleken på frekvenstabellen, innebär det dåligt minnesutnyttjande. Om värdemängden är stor och följaktligen även frekvenstabellen men de element som faktiskt förekommer är en liten delmängd, blir utnyttjandet av frekvenstabellen dålig genom att endast ett litet antal av räknarna kommer att användas. Ett omfattande initieringsarbete läggs ner på att nollställa räknare där de flesta aldrig används och en stor frekvenstabell där de flesta räknarna är 0, måste sedan gås igenom då utdata ska genereras.

5.2.2 Radix Sort

Radix Sort, även kallad *Digit Sort*, bygger på principen att element som ska sorteras har nycklar som kan behandlas som om varje element bestod av en följd av siffror, därav namnet *Digit Sort*. Decimala heltal, där varje element består av en följd de decimala siffrorna 0-9 är ett exempel. Varje siffra har en vikt, beroende på dess position i talet, entalssiffran har vikten 1 (10^0), tiotalssiffran har vikten 10 (10^1), osv. Det finns olika varianter av Radix Sort, här beskrivs LSD (Least Sigificant Digit) Radix Sort, vilken börjar distribuera med avseende på den minst signifikanta siffran. Radix-sortering användas för alla slags element som består av en följd av symboler ur ett ordnat alfabet.

För att genomföra sorteringen används köer, lika många som det finns symboler i alfabetet. De element som ska sorteras gås igenom lika många gånger som det maximala antalet symboler som kan förekomma i ett element. I det första passet betraktas symbolerna i den minst signifikanta positionen och symbolen avgör i vilken kö som elementet ska placeras. När alla element är lästa och distribuerade över köerna, slås köerna ihop i alfabetisk ordning. I det andra passet tas den näst minst signifikanta symbolen och elementen fördelas på nytt över köerna, vilka sedan slås ihop. I det sista passet behandlas den mest signifikanta positionen och när köerna slås ihop för sista gången är elementen sorterade.

Som exempel sorteras tresiffriga decimala heltal i intervallet 0..999. Antalet köer som behövs är alltså 10 och tre pass behöver genomföras. Följande element ska sorteras:

```
189   203   305   099   858   974   056   008   273   021
```

I det första passet fördelas elementen med avseende på den minst signifikanta, sista siffran. De tio köerna innehåller följande efter detta:

```
 0    1    2    3    4    5    6    7    8    9
     021       203  974  305  056       858  189
               273                      008  099
```

Köerna slås ihop, från vänster, och vi får följande resultat av det första passet:

```
021   203   273   974   305   056   858   008   189   099
```

Elementen är nu ordnade med avseende på den minst signifikanta siffran. Det andra passet genom-för och elementen fördelas nu med avseende på den andra siffran. Köerna innehåller efter detta:

```
 0    1    2    3    4    5    6    7    8    9
203       021            056       273  189  099
305                      858       974
008
```

Sammanslagning av köerna ger följande, talen är ordnade efter de två minst signifikanta siffrorna:

```
203   305   008   021   056   858   273   974   189   099
```

I det tredje och sista passet fördelas elementen med avseende på den mest signifikanta, första siffran,. Detta leder till följande innehåll i köerna:

```
 0    1    2   3    4    5    6    7    8    9
008  189  203  305                     858  974
021       273
056
099
```

Sista sammanslagningen av köerna görs och ger slutresultatet:

```
008   021   056   099   189   203   273   305   858   974
```

Om man försöker uppskatta tidskomplexiteten hos radixsortering, kan man konstatera att det bör vara en funktion av antalet element *n* som ska sorteras, radix *r* och antalet symboler *d* i nycklarna. Momentet att sprida elementen från en (sammanslagen) lista till dellistorna innebär *n* förflyttningar. Operationen att slå ihop listorna beror på hur listan och dellistorna är realiserade,. Om man antar att de är länkade listor innebär sammanslagning att länka ihop de *r* dellistorna. Antalet pass beror av antalet siffror, dvs *d*. Man får alltså en tidskomplexitet som är $O(d \cdot (n+r))$. För fixa element på *d* och *r* är alltså radixsortering *O(n)* men observera att då kan som mest $n = r^d$ distinkta nyckelelement hanteras. Då *n* ökar måste vid vissa gränser antingen *r* eller *d* ökas, vilket för *d* innebär att $d \geq {}^{r}\log n$. Komplexiteten för radixsortering kan i sken av detta sägas vara $O({}^{r}\log n \cdot (n+r))$.

5.3 Externa sorteringsmetoder

Externa sorteringsmetoder avser sortering av sekundärminneslagrade data, dvs sortering av data på filer. Anledningen till att använda en extern metod kan vara att datamängden är så stor att alla data inte kan tas in i primärminnet på en gång eller att det av andra skäl inte är önskvärt eller nödvändigt att tillgripa en intern metod.

Externa metoder bygger vanligtvis på samsorteringsteknik, dvs att ur korta sorterade delsekvenser successivt generera allt längre sorterade delsekvenser till dess en enda sorterad sekvens återstår. Man kan urskilja två kategorier av samsortering, balanserad och naturlig. *Balanserad samsortering* bygger på att man arbetar med bestämda, successivt ökande delsekvenslängder, medan *naturlig samsortering* innebär att man i varje pass försöker generera så långa sorterade delsekvenser som möjligt. Som exempel på dessa två grundläggande varianter tas *balanserad tvåvägs samsortering* och *naturlig tvåvägs samsortering* upp nedan.

Vid samsortering genererar man ibland sorterade delsekvenser (*runs*) innan själva samsorteringen startar. Det har fördelen att antalet pass i själva samsorteringen reduceras påtagligt och att sådana initiala delsekvenser kan genereras effektivt genom intern sortering. Ska delsekvenser av längd *m* genereras, läses *m* element i taget in från indatafilen och sorteras. Varje sorterad delsekvens skrivs på någon av två eller flera utdatafiler. Hur många filer och hur delsekvenserna distribueras på filerna beror på samsorteringsmetoden ifråga.

5.3.1 Balanserad tvåvägs samsortering

Balanserad tvåvägs samsortering innebär att man använder två indatafiler och två utdatafiler i varje pass, och att längden på de sorterade delsekvenserna fördubblas i varje pass. De initiala delsekvenserna av längd *m*, genereras genom att läsa in *m* element i taget från den fil som ska sorteras, sortera med en intern metod och skriva ut delsekvenserna växelvis på de två filer som ska vara indatafiler i det första samsorteringspasset. Sedan öppnas de två filerna för läsning och delsekvenserna samsorteras parvis till delsekvenser av längd 2*m*, som växelvis skrivs på två utdatafiler. I nästa pass samsorteras till delsekvenser av längd 4*m*. Detta upprepas till dess en sorterad sekvens med de *n* element erhålls. Nedan visas detta med *m*=1, dvs de initiala delsekvenserna består av bara ett element. A, B, C och D representerar filer, som växelvis används som indata- eller utdatafiler. Följande data som ska sorteras:

```
11 43 82 21 10 78 16 35 69 97 54
```

Första görs uppdelning på två filer, A och B, i detta exempel genom att skriva element växelvis på filerna. Lodstrecken avgränsar delsekvenserna.

```
A: 11 | 82 | 10 | 16 | 69 | 54
B: 43 | 21 | 78 | 35 | 97
```

Fyra samsorteringspass krävs för att erhålla en sorterad sekvens. Delsekvensernas längd blir 2, 4, 8 och 16. Som framgår räcker inte alltid elementen till för att fylla den sista delsekvensen som genereras i ett pass.

```
C: 11 43 | 10 78 | 69 97
D: 21 82 | 16 35 | 54

A: 11 21 43 82 | 54 69 97
B: 10 16 35 78 |

C: 10 11 16 21 35 43 78 82 |
D: 54 69 97 |

A: 10 11 16 21 35 43 54 69 78 82
B:
```

Antalet pass är $\lceil {}^2\log(n) \rceil$ för m=1, dvs $O(\log n)$. För m>1 får vi $\lceil {}^2\log(n/m) \rceil$, dvs $\log(n/m)$ pass. För m>1 tillkommer arbetet med att generera de initiala delsekvenserna. I varje pass flyttas samtliga n element från indatafilerna till utdatafilerna, vilket ger tidskomplexiteten $O(n)$. Om den aktuella delsekvenslängden är d, så varierar antalet jämförelser mellan $d/2$ och $d–1$. Det minsta antalet jämförelser får man om samtliga element i den ena delsekvensen är mindre än eller lika med det första i den andra delsekvensen, flest jämförelser får man om det endast återstår ett element i en delsekvens när den andra tömts. Antalet jämförelser per pass är $O(n)$.

5.3.2 Naturlig tvåvägs samsortering

Naturlig tvåvägs samsortering innebär att man genererar så långa sorterade delsekvenser som möjligt i varje pass, i stället för att använda fixa delsekvenslängder. Fördelen med detta är att antalet pass kan reduceras. Med samma indata som i exemplet för balanserad tvåvägs samsortering ovan, erhålls följande förlopp.

```
A: 11  82  10  16  69  54
B: 43  21  78  35  97

C:  11  43  82 • 35  54  97
D: 10  16  21  69  78

A: 10  11  16  21  43  69  78  82
B: 35  54  97 •

C: 10  11  16  21  35  43  54  69  78  82  97
D:
```

Punkterna markerar de delsekvenser som genererats. Jämfört med exemplet på balanserad tvåvägs samsortering ovan blev det ett pass mindre men antalet jämförelser per pass blir fler i naturlig samsortering. Samtliga element i indatafilerna, utom de som återstår i den ena filen när den andra helt tömts, jämförs både med ett element i den andra indatafilen och med det sista elementet som placerades i utdatafilen, för att avgöra om det går att bygga vidare på den aktuella delsekvensen eller inte.

5.3.3 Mångvägs samsortering

Med användning av flera filer kan man förvänta att antalet pass minskar. Den balanserade tvåvägs samsorteringen, till exempel, kan generaliseras till en balanserad *k*-vägs samsortering, med *k* indatafiler och *k* utdatafiler, dvs totalt 2*k* filer. Det blir naturligtvis mer komplicerat att hitta det minsta av *k* element, ett sätt är att använda en prioritetskö. Efter att de initiala delsekvenserna med längd *m* genererats, blir antalet pass som krävs för att samsortera i detta fall $\lceil {}^{k}\log(n/m) \rceil$.

5.3.4 Polyphase Merge

Polyphase Merge är en avancerad balanserad samsorteringsmetod, som gör *k*-vägs samsortering med endast *k*+1 filer. Den bakomliggande idén är att endast en indatafil töms helt i varje pass och att fördelningen av antalet delsekvenserna på de olika filerna görs optimal med tanke på detta. Den bästa distributionen är relaterad till Fibonaccitalen, vilka definieras enligt följande.

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

De tio första Fibonaccitalen är alltså 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Fibonaccital kan användas för att bestämma distributionen för en tvåvägs Polyphase Merge enligt följande. Antag att de 21 talen nedan ska sorteras. Den initiala delsekvenslängden väljs för enkelhets skull till 1.

```
21 58 85 18 64 96 37 47 72 51 29 87 13 69 44 82 25 75 32 93 54
```

Den första uppdelningen av de 21 (F_8) elementen görs med 8 (F_6) element på fil A och 13 (F_7) element på fil B.

```
A: 21 | 58 | 85 | 18 | 64 | 96 | 37 | 47
B: 72 | 51 | 29 | 87 | 13 | 69 | 44 | 82 | 25 | 75 | 32 | 93 | 54
```

En utdatafil används i en tvåvägs Polyphase Merge. I varje pass kommer den kortaste av de två indatafilerna att tömmas. Den längre kommer att vara indatafil även i nästa pass, då den kommer att tömmas. Förloppet visas nedan och det behövs alltså 6 pass för att sortera de 21 elementen.

```
A:
B: 25 | 75 | 32 | 93 | 54
C: 21 72 | 51 58 | 29 85 | 18 87 | 13 64 | 69 96 | 37 44 | 47 82

A:  21 25 72 | 51 58 75 | 29 32 85 | 18 87 93 | 13 54 64
B:
C: 69 96 | 37 44 | 47 82

A: 18 87 93 | 13 54 64
B: 21 25 69 72 96 | 37 44 51 58 75 | 29 32 47 82 85
C:

A:
B: 29 32 47 82 85
C: 18 21 25 69 72 87 93 96 | 13 37 44 51 54 58 64 75

A: 18 21 25 29 32 47 69 72 82 85 87 93 96
B:
C: 13 37 44 51 54 58 64 75

A:
B: 13 18 21 25 29 32 37 44 47 51 54 64 69 72 75 82 85 87 93 96
C:
```

I en *k*-vägs Polyphase Merge, används *k*:te ordningens Fibonaccital för distributionen.

5.3.5 Replacement Selection

Replacement selection används för att generera initiala delsekvenser på ett sätt som kan vara bättre än att bara läsa in *m* element från en indatafil, sortera dem internt och skriva ut dem på en utdatafil. Algoritmen bygger på att det blir en intern plats ledig när ett element skrivs på utdatafilen. och att denna lediga plats kan användas för nästa element i indatafilen. Det kan då uppstå två fall:

- nästa element är större än eller lika med det utskrivna elementet och kan ingå i den delsekvens som håller på att skapas och kan ordnas in i den interna strukturen.
- nästa element är mindre än det utskrivna elementet och kan inte ingå i den aktuella delsekvensen men om den interna strukturen kan delas upp i en del med element som ingår i den aktuella delsekvensen lagras och en del med element som inte gör det, kan ett sådant element läsas och göra det möjligt att komma åt fler element i indatafilen som kan ingå i den aktuella delsekvensen.

En datastruktur som uppfyller dessa krav är en prioritetskö (binär heap). Först läses *m* element in från indatafilen och sätts in i en *min-heap*. Sedan görs *delete-min* och elementet som tas ut ur prioritetskön skrivs på utdatafilen. Nästa element läses från indata. Om det är större än eller lika med det element som just skrivits på utdatafilen kan det ingå i den aktuella delsekvensen och sätts in i priortetskön med operationen *insert*. I annat fall placeras elementet i det i det tomma utrymme som uppstod när prioritetsköns storlek minskade då *delete-min* utfördes. Detta upprepas till dess prioritetsköns storlek blir 0. Då påbörjas en ny delsekvens, genom att återställa storleken på prioritetskön och heapordna de element som finns i det tidigare döda utrymmet. Som exempel visas hur några delsekvenser genereras för följande element:

```
64 21 96 58 18 85 37 72 47 51 75 82 13 87 44 93 25 29 32 69 54
```

I figur 62 visas, steg för steg, hur den första delsekvensen genereras med en prioritetskö med storlek 3. De klamrade siffrorna 1, 2 och 3 avser positionerna i den binära heap som implementerar prioritetskön. De skuggade elementen ingår inte i den del av minnet som prioritetskön omfattar.

	prioritetskö			
operation	`[1]`	`[2]`	`[3]`	**utdata**
toss	64	21	96	
fix-heap	21	64	96	
delete-min	64	96		21
insert(58)	58	96	64	21
delete-min	64	95		21 58
18 < 58	64	96	18	21 58
delete-min	96		18	21 58 64
insert(85)	85	96	18	21 58 64
delete-min	96		18	21 58 64 85
37 < 85	96	37	18	21 58 64 85
delete-min		37	18	21 58 64 85 96
72 < 96	72	37	18	21 58 64 85 96

Figur 62. Generering av första delsekvensen.

I figur 63 visas, lite mindre detaljerat än i figur 62, hur den andra delsekvensen genereras. I detta fall finns redan tre element från föregående steg på plats, så den första åtgärden är att ordna elementen med *fix-heap*.

operation	prioritetskö [1]	[2]	[3]	utdata
fix-heap	18	37	72	
delete-min, insert(47)	37	72	47	18
delete-min, insert(51)	47	72	51	18 37
delete-min, insert(75)	51	72	75	18 37 47
delete-min, insert(82)	72	75	82	18 37 47 51
delete-min, göm 13	75	82	13	18 37 47 51 72
delete-min, insert(87)	82	87	13	18 37 47 51 72 75
delete-min, göm 44	87	44	13	18 37 47 51 72 75 82
delete-min, insert(93)	93	44	13	18 37 47 51 72 75 82 87
delete-min, göm 25	25	44	13	18 37 47 51 72 75 82 87 93

Figur 63. Generering av andra delsekvensen.

De tre element som återstår efter andra passet kommer att generera en tredje och sista delsekvens. En 3-vägs sortering skulle i detta fall bli klar i ett pass. Om vi hade läst in och sorterat tre element i taget, hade sju delsekvenser erhållits och en 3-vägs sortering hade då behövt tre pass för att bli klar.

Om indata är slumpvis ordnade kan man visa att *replacement selection* genererar delsekvenser vars längd i genomsnitt är 2*m*. Detta kan dock innebära att antalet pass inte reduceras men med lite tur kan det bli så och med tanke på att extern sortering är långsam är varje insparat pass värdefullt. I situationer där extern sortering används är det dessutom ganska vanligt att indata är nästan ordnade och då kommer *replacement selection* att producera endast ett fåtal mycket långa initiala delsekvenser och därmed få pass. Detta sammantaget gör *replacement selection* till en värdefull metod för att generera delsekvenser.

5.4 Indirekt sortering

Indirekt sortering innebär att de element som ska sorteras ligger i en struktur och i en annan struktur lagras enbart adresserna till respektive element. Sortering görs genom att ordna adresserna så att de hamnar i en ordning som vid en sekventiell genomläsning ger elementen i ordning.

6 Interpolationssökning

Interpolationssökning tillhör en grupp av sökmetoder som bygger på en teknik som kallas tudelningssökning. Mest känd bland dessa är *halveringssökning*, ofta kallad *binärsökning*. Tudelningssökning bygger på att man har en linjär sökstruktur där elementen är direktadresserbara, exempelvis ett fält, och att elementen är ordnade enligt en söknyckel.

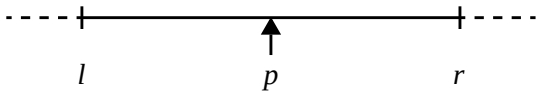
Tudelningsökning innebär att man väljer ut en position någonstans i den del av sökstrukturen man för tillfället opererar på, initialt hela strukturen. Om nyckelvärdena är unika gäller då att alla nyckelvärden till vänster om den valda positionen är mindre än nyckelvärdet i den valda positionen och alla nyckelvärden till höger är större än nyckelvärdet i den valda positionen. Man jämför nyckelvärdet i den utvalda positionen med sökt nyckelvärde. Tre fall kan då inträffa:

- nycklarna är lika, man har funnit det man söker och kan avbryta sökningen.
- söknyckeln är mindre än nyckelvärdet i den utvalda positionen och man kan eliminera den del av sökintervallet som ligger till höger därom.
- söknyckeln är större än nyckelvärdet i den utvalda positionen och man kan eliminera den del av sökintervallet som ligger till vänster därom.

Så länge man inte finner sökt värde reducerar man successivt sökintervallet genom delning och eliminering av den ena intervallhalvan, till dess man antingen finner sökt värde eller inget element återstår i sökintervallet.

I halveringssökningsfallet väljer man ut det mittersta elementet för att jämföra med. Om sökstrukturen är ett fält och l och r anger index för sökintervallets ändpunkter beräknad delningspunkten p som p = (l+r)/2, vilket är härlett ur följande uttryck.

$$p = l + \frac{1}{2} \cdot (r - l)$$

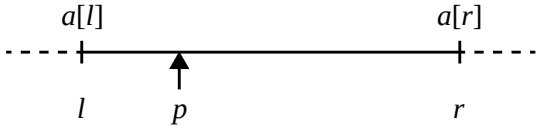


Mittpunkten i intervallet beräknas genom att lägga till halva intervalllängden till dess vänstra ändpunkt.

I varje steg i halveringssökningen eliminerar man halva antalet element i den återstående delen av sökstrukturen. Om det inte finns några statistiska faktorer, som att sökmängden har en speciell sammansättning eller att sökning normalt gäller exempelvis främst små nycklar, är halveringssökning det optimala valet av tudelningssökning, såvida man inte väljer interpolationssökning.

Interpolationssökning syftar till att försöka optimera valet av delningspunkt, genom att utifrån söknyckelns värde *x* och nyckelvärdena i sökintervallets ändpunkter, *a*[*l*].key respektive *a*[*r*].key, interpolera en lämplig delningspunkt *p*. Det är alltså faktorn 1/2 man ersätter med en uppskattning av var sökt element borde finnas, genom att interpolera.

$$p = l + \frac{x - a[l].\text{key}}{a[r].\text{key} - a[l].\text{key}} \cdot (r - l)$$



I värsta fall är interpolationssökning mycket sämre än halveringssökning men detta innebär att man skulle råka välja söknyckelsekvenser som är mycket osannolika. I normala fall, vilket avser att valet av söknycklar är likfomigt fördelat, är den förväntade tidskomplexiteten avsevärt mycket bättre än den för halveringssökning, nämligen O(log log n), jämfört med halveringssökningen O(log n).

6.1 Litteratur

Sedgewick, R. (1990), Algorithms in C, Addison-Wesley, sid.201-202.
Lewis H. R., Denenberg L. (1991), Data Structures and Their Algorithms, HarperCollins, sid 184-187.