|  |  |  |  |
| --- | --- | --- | --- |
| **Allmänna träd**  - Ex. Filsystem, databaser för att få snabb sökning, schackprogram för att gå igenom resultaten av möjliga drag, vid datakomprimering för att få fram en optimal kod (Huffmanträd på komprimeringsdelen).  **Syskon** har samma förälder.  **Löv** är en nod som endast pekar på None.  **Nivå** = antal steg från roten. Roten är på nivå noll.  **Höjd** är maximala nivån.  **Fullt** träd om alla noder utom löven har två barn o alla barn är på samma nivå.  **Binärträd**  - Vänster barn till föräldern på nod *p* finns på positionen 2*p* i listan. Höger barn på position 2*p*+1. Hitta en förälder till en nod i position *n* så är föräldern på position *n*/2.  - Fullt träd med *k* nivåer innehåller 2^k-1 noder.  - Balanserat träd med *n* noder har ca log*n* nivåer. | **Binärt sökträd**  - Algoritmer som går igenom varje nod i trädet (t.ex. utskrift) har tidskomplexitet O(*n*).  - Söka O(log*n*) om balanserat träd.  - Insättning O(log*n*)  Def inorder(p):  If p != None:  inorder(p.left)  Print(p.value)  Inorder(p.right)  Print(p.value)  preorder(p.left)  preorder(p.right)  postorder(p.left)  postorder(p.right)  print(p.value) | **Heap – Trappa**  - Bästa implementationen av en prioritetskö.  - Roten är tab[1] dess båda barn är tab[2] och tab[3] osv. Vi använder inte tab[0]. Allmänt gäller att tab[i] har barnen tab[2\*i] och tab[2\*i+1].  - Trappvillkoret = *föräldern är bäst*.  - Insert och delMax har komplexitet O(log*N*) där trappan har N element.  **Vektor**  - Åtkomst = O(1)  - Insättning och borttagning = O(*n*)  **Länkad lista**  - Åtkomst = O(*n*)  - Insättning och borttagning = O(1) | **Prioritetskö (datastruktur)**  - Implemented med en binary heap.  - Används vid bästaförstsökning.  - Anrop: insert(p,x) stoppa in x med påstämplad prioritet p(oftast ett heltal), x = delMax() hämta det som har högst prioritet, isEmpty() undersök om prioritetskön är tom.  **Bloomfilter**  - Har flera hashfunktioner som ger index i samma booleska hashtabell tab. Lagrar 1-True eller 0-False.  - Om tabellen har *n* hashfunktioner och är fylld till *k* stor del, är sannolikheten *k^n* % att ett felstavat ord ska godkännas som rätt.  - Operationer: insert(x), isln(x) som stoppar in i datastrukturen/kollar om x finns i datastrukturen. |
| **Breddenförstsökning (BFS)**  - Implementeras med en **kö**.  Algoritm:  1. Lägg stamfadern i kön.  2. Ta ut det första objektet ur kön.  3. Skapa alla dess barn och lägg in dom i kön.  4. Om någon av barnen är lösningen så är vi klara. Annars – upprepa från punkt 2 tills kön blir tom.  5. När lösningen hittats, följ föräldrapekarna och skriv ut kedjan.  - Undvik att skapa barn som är kopior av tidigare släktingar (kallar vi *dumbarn*)  - Ger alltid den *kortaste* lösningen. Ofta är det den man är ute efter.  **Djupetförstsökning (DFS)**  - Den första lösning man hittar är inte nödvändigtvis den kortaste.  - DFS fungerar inte om trädet har oändligt djup.  - Implementeras med en **stack.**  - Kollar en hel gren innan den går till nästa gren.  Jämförelse:  - I *breddenförst* går vi först igenom alla hörn som ligger en kant från starthörnet, sen alla hörn som ligger två kanter bort osv. Om det finns flera lösningar till problemet stöter vi på den närmaste först (men om vi inte bryter där går vi igenom alla hörnen).  - I *djupetförst* följer vi istället en stig så långt det går via första kanten från starthörnet. När det tar stopp backar vi ett steg (flera vid behov) tills det går att fortsätta framåt igen.  **Bästaförstsökning (Dijkstra’s)**  - Billigaste vägen från en startnod till en slutnod i en graf med kostnader. | **Hashning**  - Pythons dictionary använder hashning.  - O(1), drygt. Krockning kan påverka.  - Undvik krockning:  *Hashtabellens storlek = 2(antal inlagda objekt)*  - Hashfunktion: skai många fall först göra om en sträng till ett stort tal. Funktionen *ord(tkn)* i Python konverterar ett tecken till motsvarande ASCII-nummer.  - **Krocklista**, lägger alla namn med samma hashvärde i en lista. Om man har femtio procents luft i sin hashtabell blir krocklistorna i allmänhet väldigt korta. Krocklistorna hanteras enklast som stackar. Hashtabellen innehåller endast pekarna till stackarna.  - **Linjär probning.** Fördel: slipper pekare. Nackdel: om det börjat klumpa ihop sig någonstans har klumpen en tendens att bli större. Kallas *klustring*.  - Kvadratisk probning. Varierar hopplängden. Om hashfunktionen gav värdet *h* tittar man i ordning på platserna: h+1, h+4, h+9, … Om värdena överstiger hashtabellens storlek använder man resten vid heltalsdivision precis som vid beräkningen av h.  - Dubbelhashning. Nästa värde räknas fram med en annan hashfunktion som tar som indata den första hashfunktionens värde. Nackdel med kvadratisk probning och denna: Kan inte enkelt ta bort noder utan att förstöra hela systemet.  - Tabellstorleken ska vara ett primtal.  - Metoder: put(key, data), data = get(key), f = hashfunction(key), exist(key) | **Binärsökning**  - Snabbaste sökningen vid sorterad lista.  Algoritm:  1. Beräkna intervallets mittpunkt.  2. Är det nyckeln? **Avbryt**, det sökta elementet är funnet.  3. Annars: Avgör om nyckeln finns i första eller andra halvan och fortsätt söka där (upprepa från punkt 1).  4. Om halvan du söker i krympt till ingenting: **Avbryt**. Det sökta elementet fanns inte med.  - *O*(log*n*)  - Antal element att söka bland halveras i varje varv så första varvet får vi *n*/2 andra varvet *n*/4 osv. Vi är klara när det bara finns ett element kvar att titta på och då är *n*/2^x = 1, där x är antal varv. Vi två-logaritmerar bägge led och får att x=log2*n*  Testning:  1. normalfallet  2. Att söka efter ett tal som *inte* finns med  3. Söka i en tom lista.  4. Söka efter elementet längst till vänster/höger.  6. Söka efter ett element som inte finns inom intervallet.  **Linjärsökning (Sequential search)**  - Enklaste sökningen: bryter så fort den hittar det sökta elementet.  Algoritm:  1. Gå igenom varje element i listan:  - Jämför elementet med nyckeln, och  - …returnera True om dom är lika.  2. Returnera False om hela listan gåtts igenom utan att elementet hittats.  - *O*(*n*) (i värsta fallet måste vi titta på alla *n* elementen) | **Grafer**  - Består av en uppsättning hörn (*vertices*) med kanter (*edges*) emellan.  - **Riktad** graf om kanterna är riktade (pilar).  - Nummer kanter från ett hörn är hörnets **grad** (*degree*).  - **Stig** (*path*) är en väg från ett hörn till ett annat.  - **Cykel** är en stig där man startar och slutar i ett och samma hörn.  - **Acyklisk** om grafen är utan cykler.  - **Viktad** graf, viktade kanter.  - **Grannmatris**. Gles grannmatris ger ineffektiv lagring.  - **Grannlista**.    **Problemträd**  - Många praktiska problem kan beskrivas med *problemträd* och sedan lösas med trädgenomgång: bredden först eller djupet först.  - Ett problemträd är en riktad, acyklisk graf.  - Problemträd är ett sätt att tänka på problem för att kunna lösa dom med grafalgoritmer. Do not mix problemträd med den abstrakta datastrukturen binärträd. |
| **Urvalssortering (selection sort)**  1. Sök igenom listan efter minsta värdet (n-1 jämförelser)  2. Flytta det till första positionen. (Ett byte)  3. Sök efter näst minsta värdet. (n-2 jämförelser)  4. Flytta det till andra positionen. (Ett byte)  - Totalt n(n-1)/2 jämförelser och n-1 byten, oberoende av hur osorterad listan är från början.  - O(*n*^2) | **Bubbelsortering (Bubble sort)**  1. Byt första och andra om dom står i fel ording (En jämförelse, ett byte)  2. Byt andra och tredje om dom står i fel ordning. (En jämförelse, ett byte)  3. Bubbla igenom listan gång på gång tills inga byten sker.  - Totalt krävs i värsta fall n(n-1)/2 jämförelser och lika många byten. Men om listan är nästan sorterad från början räcker det med några få genomgångar och då blir bubbel snabbare än urval.  - O(*n*^2) | **Insättningssortering - (Insertion sort)**  - Bra val om man läser in värden ett i taget.  1. Jämför med varje tidigare värde i listan.  2. Om det nya värdet är mindre gör vi plats genom att flytta det tidigare värdet ett snäpp åt höger.  3. Flytta fram så många värden som behövs för att sätta in nya värdet på rätt plats.  4. Stoppa in det nya värdet.  - O(*n*^2). Lite snabbare än urvalssortering i praktiken eftersom en flytt är snabbare än ett byte.  - Bästa valet för att sortera in ett nytt värde i en redan sorterad lista. | **Quick sort**  - O(*n*log*n*)  - Om man tar första värdet i listan till sitt pivot värde blir den långsam för listor som rean nästan är sorterade.  - Bättre: ta ut tre värden, första, sista och något i mitten – låt det mellersta värdet vara pivot värdet. Kallas *median-of-three*.  - Detta ger komplexiteten O(1.4*n* log*n*)  **Devide and conquer**  - Delar upp ett problem i två eller flera delproblem och löser dessa var för sig. Lättast med rekursion.  - T.ex. quicksort och mergesort. |
| **Räknesortering (Distribution count)**  - Bra vid få antal nyckelvärden. Ex. sortering av Sveriges befolkning efter födelseår.  - Supersnabb vid dessa tillämpningar.  - Kräver att talen som sorteras in i listan hämtas från en anna lista eller fil.  - O(*n*).  1. Läs igenom filen och räkna hur många det finns av varje nyckelvärde. Lagra värdena i en mellanvektor.  2. Dela in listan i lagom stora segment.  3. Läs filen igen och lägg in varje värde i sitt segment. Väljer om man vill fylla på bakifrån eller framifrån. | **Heap sort**  - Sätt in *n* tal i en heap och ta ut dom -> sorterade.  - O(*n*log*n*) (samma som Quicksort) men inte lika snabb.  - Har dock inte Quicksorts värstafallbeteende. | **Mergesort (Samsortering)**  - Om man har flera sorterade småfiler är det lätt att *samsortera* dom till en fil. Rekursiv implementation.  1. Dela listan i två hälften så långa listor. Tills den har längden 1.  2. Sortera varje halva för sig.  3. Samsortera till ursprungliga listan.  - O(*n*log*N*)  - Lika snabb som quicksort – kräver dock extra minnesutrymme. |  |
| **Transpositionschiffer (Kryptering)**  - Dekrypteringspinne för avkodning av meddelanden skrivna på en remsa skinn.  - mxn-matris och bilda det kodade meddelandet genom att ta varje kolumn i texten.  - Behöver matrisens storlek. Enkelt isf.  **Ceasarchiffer (Kryptering)**  - A byts mot D  - B byts mot E …. Osv.  - Kan variera hur stor förskjutning. T.ex. 13 steg.  - Kan slumpa fram en mappning istället för ett bestämt antal steg. Lätt att knäcka om man har statistik på hur mkt varje bokstav används i svenska språket.  **Bokchiffer (Kryptering)**  - Bok som nyckel. Mappar varje ord med nummer på sidan och nummer som ordet har på den sidan.  - Måste i förväg bestämma bok. Och kommunicera detta så att Eve inte hör. | **One-time pad (Kryptering)**  - Oknäckbar. Använder sig av en slumpad nyckel.  - Har ett meddelande på binär form:  - Slumpar en nyckel med samma bitlängd som meddelandet.  - Bitvis XOR mellan meddelandet & nyckeln.  - För dekryptering måste man ha nyckeln.  - Bob & Alice har varsit tex av en ”pad”/block med slumpade siffror.  - XOR nyckel genereras snabbt, snabbare än tex RSA.  - Neg: Engångsnyckel och måste genereras ny varje gång.  - Neg: Nyckeln måste distribueras på ett säkert sätt.  - Neg: Garanterar inte autentisering. Kan skicka falska meddelanden. | **RSA – Assymetrisk kryptering**  - Förutsätter nyckelpar. Krypterar med Bobs publika nyckel, signerar med Alice privata nyckel.  - Du har en *offentilig* och en *privat* nyckel. *Offentliga* ges till alla för kryptering av meddelanden som ska skickas till dig. Bara du kan dekryptera mha den *privata* nyckeln.  - Pos: Behöver bara generera nycklarna en gång.  - Pos: Publika nycklarna behöver inte distribueras på ett säkert sätt, då de är publika.  - Pos: Ger autentisering och kryptering. (Säker på att det verkligen är Alice som har skickat meddelandet)  **Symmetrisk kryptering**  - Sändare och mottagare har samma krypteringsnyckel. Går oftast mkt snabbare än asymmetrisk kryptering. | **Kryptering**  - Ex. överföring av kontokortsnummer över internet eller när man vill skicka lägesrapporter från diktaturer.s  **Datasäkerhet – Lösenord**  - Lösenord borde sparas saltade och hashade.  - En hashfunktion kan ge samma värde för två helt olika indata och det går därmed inte att, givet hashvärdet, veta vilket ursprungsvärdet var. |
| **Automater**  Kan användas till:  - KMP-automat, söka efter ord i en text.  - Tolka *reguljära* *uttryck*.  - Beskriva grafiska gränssnitt.  - Kompilatorns/interpretatorns analys av ditt program (se syntax)  - Komprimering.  **Boyer - Moore (textsökning)**  - Texten man söker i ligger i en lista.  - Hela texten = *n*  - Matchar först sista bokstaven (*m*) i söksträngen.  - Om det tecknet från söktexten inte finns i söksträngen hoppar den fram *m* steg.  - Annars flyttar den fram så att tecknet i texten passar ihop med sista förekomsten i söksträngen.  - O(*n*+*m*) i värsta fall  - ca O(*n*/*m*) steg om söktexten består av många fler täcken än vad som ingår i söksträngen. Då kan vi ofta kan hoppa fram *m* steg. | **Rabin-Karp (textsökning)**  - Beräknar en hashfunktion för söksträngen och jämför med hashfunktionen beräknad för alla avsnitt av längden *m* i texten. Låt oss söka efter ”TILDA” i texten ”MEN MILDA MATILDA”. Med pythons hashfunktion hash() % 17 får vi hash(TILDA)=16  - För att snabba upp beräkningarna använder man hela tiden förra hashvärdet.  - O(*nm*) i värsta fallet men i praktiken bara O(*n*+*m*) | **Reguljära uttryck (textsökning)**  - Om man vill söka efter t.ex. lab1, Lab2 eller labb3 används det för att beskriva söksträngen.  - a\* matchar noll eller flera a:n  - a+ matchar ett eller flera a:n  - a+? Matchar ett eller flera a:n men försöker med så få a:n som möjligt  - {n} matchar n gånger  - a? matchar ett eller flera a:n men försöker med så få a:n som möjligt.  - . matchar alla tecken utom radslut.  - [a-zA-Z] matchar alla engelska bokstäver.  - [abc] matchar a, b eller c  - [^abc] matchar vilket tecken som helst utom a, b eller c.  - X|Y matchar uttrycket X eller uttrycker Y.  - \. matchar en punkt. Tecknet ”\” används för att rädda det efterföljande tecknet från att tolkas som ett metatecken.  - (ab) skapar en grupp. T.ex. matchar (ab)+ ett eller flera ab:n.  - Se RLE, Huffmankodning, Lempel-Ziv, Entropi, Komprimering av bilder, Komprimering av rörliga bilder, Komprimering av ljud, Felkorrektion. |  |
| **Komprimering**  - Komprimering innebär att man använder någon metod för att minska storleken på en fil. *Förlustfri komprimering* (non-lossy compression) betyder att det går att dekomprimera och sedan få tillbaka filen i ursprungligt skick. *Förstörande komprimering* (lossy compession) innebär att man tar bort data. Principen att komprimera utan att förstöra en fil grundrar sig i att filer oftast har redundans, alltså innehåller mer än nödvändigt.  **Felkorrektion (Komprimering)**  - Motsatsen till komprimering för att motverka fel. Lägger till redundans.  Exempel:  - Kontrollsiffra.  - Skicka kopior av hela meddelandet, minst tre är nödvändigt om man ska kunna korrigera.  - Paritetsbitar, att man lägger till en etta eller nolla till ett binärt tal för att göra det udda. Ett jämnt tal innebär att någon bit är fel.  - Hammingavstånd: Lägg till så många extrabitar till koden så att varje enbitsfel ger ett kodord som skiljer sig en bit från det förstörda kodordet, men i flera bitar från alla övriga kodord. | **Komprimering av bilder**  - En del färger kan vara vanligare, kan använda Huffmankodning för att få kortare koder för dessa.  - I foton är närliggande pixlar ofta lika. Kan använda RLE. Kan även använda varianter av LZW – då innehåller tabellen pixelinfo istället för strängar! Kan även använda *förstörande komprimering* och ta bort information som ögat inte kan se ändå.  - GIF, filformat för bilder där färgkodningen görs med 8 bitar, man får 2^8=256 färger. Använder sedan LZW för att komprimera. Komprimeringen är förlustfri och storleken minskas ungefär med en faktor 4.  - JPEG Bra för foton där närliggande pixlar har liknande färger. Delar upp färgbilder i en belysningsdel och en färgdel, sedan komprimeras färgdelen med förstörande komprimering då ögat inte är så känsligt för färgförändringar. Man använder sedan en kombo av RLE och Huffmankodning för att koda grupper av pixlar. Har *komprimeringsgraden* som en parameter till algoritmen, på så sätt kan man bestämma själv hur hårt man vill komprimera. Färgkodningen görs med 24 bitar, dvs 2^24 (nästan 17 miljoner) färger.  **Huffmankodning (Komprimering)**  - Vet man sannolikheten för att ett visst tecken ska dyka upp = > ställ upp en tabell.  - Statistisk metod. | **Entropi (Komprimering)**  - Hur mycket kan man komprimera utan att förlora information? Finns en undre gräns för hur kompakt en fil kan fås med förlustfri komprimering. Om man vet sannolikheten för varje tecken som ska kodas kan man beräkna *entropin* som ger en undre gräns för medellängden hos en kod.  - Antag att vi har en teckenmängd m1, m2, …m mn (t.ex. alfabetet) och att sannolikheten för att tecknet mi ska förekomma är P(mi).  - Då är L(mi)=-log(P(mi)) minimilängden för ett kodord för tecknet mi och L\_medel = P(m1)\*L(m1) + … + P(mn)\*L(mn) medellängden för koderna (entropin).  **Följdlängdskodning–RLE (Komprimering)**  - Utnyttjar att man har en följd av likadana tecken.  - ÅÅÅÅH! JAAAAAAA! AAAAAAAAAAAAH.  - Vi ersätter följderna av Å och A med antalet följt av det upprepade tecknet:  - 4ÅH! J7A! 12AH.  - Om dock grundtexten innehåller siffror blir det svårtolkat. Inför bryttecken, t.ex. $, som vi är säkra på inte kommer att förekomma i texten.  - $4ÅH! J$7A! $12AH.  - Algoritmen blir enkel, men tyvärr inte så användbar för textkomprimering eftersom de flesta texter inte innehåller längre följder av samma tecken. | **Lempel-Ziv (Komprimering)**  - Alla texter följer inte statistiken.  - Förutsättningslös metod som anpassar sig till indata.  Algoritm:  - Läs in tecken för tecken och slå ihop till en sträng s.  - Fortsätt på det viset så länge som strängen redan finns med i ordlistan.  - Så småningom får vi en sträng som inte finns i ordlistan (s finns med men inte s+c)  - Skriv då ut koden för strängen s, skriv ut tecknet c, och lägg in s+c i ordlistan.  Exempel: NÄSSNUVSNORSNOK  **code:** 0 1 2 3 4 5  **sträng:** ”N” ”Ä” ”S” ”SN” ”U” ”V” ”SNO”  **code:** 6 7 8  **sträng:** ”SNO” ”R” ”SNOK”  NÄS2NUV3OR6K |
| **Komprimering av rörliga bilder**  - Videofiler innehåller en hel del bilder och även ljud, det är därmed extra viktigt att kunna komprimera dem. Dekomprimeringen ska gå snabbt om i fall där man ska kunna se filmen i realtid. Mest kända formatet för rörliga bilder är MPEG.  - MPEG är en samling standarder för kombinationer av ljud och video.  - Komprimeringen av video-delen kan delas upp i *bildkomprimering* av varje enskild bildruta och *tidskomprimering* där man utnyttjar likhet mellan på varandra följande bilder.  - För bildkomprimeringen används i regel JPEG. För tidskomprimeringen finns ett antal olika metoder:  - Koda *likheter* (utnyttjar att delar av bilden ser likadan ut som i förra rutan).  - Koda *förskjutningar* (att en del av bilden har förskjutits sen förra rutan).  - Koda *skillnaden* mellan två bildrutor.  - Koda *förväntad* rörelse.  - Tidskomprimeringen kan göra det knepigare att redigera filmen. | **Komprimering av ljud**  - Samplat en analog ljudkurva i ett ändligt antal punkter, detta medför automatiskt komprimering då vi har digital lagring av ljud.  - Komprimering av digitala ljudfiler kan göras med RLE eller Huffmankodning.  LZ-metoderna fungerar inte särskilt bra, då de bygger på att man hittar upprepningar. Och trots att t.ex. en låt har upprepningar behöver inte samma upprepningar återfinnas i ljudfilen efter samplingen.  - Kan använda *förstörande metoder*.  Exempel på sådana:  1. *Tystnadskomprimering*, man ersätter väldigt svaga ljud med tystnad  2. *Companding*, man minskar ordlängden för varje ljudpunkt (t.ex. från 16 till 12 bitar).  - MP3 använder en kombo av tekniker, man utnyttjar en modell av den mänskliga hörseln och Huffmankodning. | **Syntax**  - Testfall: Ett som är komplicerat men följer syntaxen och ett som är enkelt men inte följer syntaxen. Andra bra testfall är tomma meddelanden och långa meddelanden, eftersom dessa ofta skapas av misstag.    **Syntaxkontroll med stack**  - Kan använda en stack för att se om en inmatning fljer syntaxen. Exempelvis omatchande paranteser.  1. Skapa en tom stack.  2. Slinga som läser symboler (här:tecken) tills inmatningen tar slut.  - Om symbolen är en startsymbol (tex[), lägg den på stacken.  - Om symbolen är en slutsymbol (t.ex. ]), titta på stacken. Om stacken är tom eller om den symbol som poppar ut inte matchar slutsymbolen har vi ett syntaxfel.  3. När inmatningen tar slut – kolla om stacken är tom. Om den inte är tom har vi fått ett syntaxfel.  - Men vad händer om man lagt in parenteser inuti en kommentar? Vi vill att alla tecken inuti kommentaren ska läsas bort. Lös detta genom att låta programmet bete sig som en automat med flera tillstånd.  1. Leta parenteser att lägga på stacken. Övergår till tillstånd 2 om den upptäcker en kommentar, till tillstånd 3 om inmatningen tar slut.  2. Inuti kommentaren – läser bort tecken. Återgår till tillstånd 1 om kommentaren tar slut.  3. När inmatningen tar slut – kollar om stacken är tom. Om den inte är tom har vi fått ett syntaxfel.  - Den som vill skriva ett eget programmeringsspråk måste först skriva en syntax för språket, och sedan ett program som kan tolka språket. | **Rekursion**  - Rekursiv tanke: reducerar problemet till ett enklare problem med samma struktur.  - Basfall: ett fall som inte leder till rekursivt anrop.  - Ex. Triangeltalet S(N) är summan av de N första heltalen  - Ex. Vilken siffersumma har heltalet n?  - Hur programspråken hanterar rekursiva anrop.  1. För varje anrop skapas en *aktiveringspost* som innehåller data för anropet, t.ex. parametrar, lokala variabler och anropspunkt.  2. Aktiveringsposten pushas på en stack.  3. När det rekursiva anropet är klart poppas aktiveringsposten från stacken, varefter föregående anrop ligger överst på stacken. |