Algoritmer och Datastrukturer

Sammanfattning

ADT

- Abstract Data Type
- Mängd objekt som är samlade tillsammans.
- En mängd operationer som kan utföras på objekten.
- Implementeras till olika datastrukturer. Såsom Listor, Stackar och Köer.

Lista

- En samling element i en ordnad struktur.
- Det finns en ordning mellan elementen.
- Element har oftast ett visst index som representerar deras plats.
- Vanliga Operationer
    - Lägga till element.
    - Ta bort element.
    - Hämta ett element för ett visst index.
    - Hitta ett specifik element.
    - Tömma.
- Array-baserad implementation
    - Använder en array för att hålla alla elementen.
    - Börjar med en fixerad längd/kapacitet.
    - Ifall max kapaciteten nås ökas den. Oftast genom att dubblera storleken i hopp om att denna operation inte behöver göras igen.
        - Det är en krävande process att skapa en ny array som är dubbelt så stor och kopiera över alla elementen från den gamla till den nya.
        - Minneskrävande eftersom dubbleringen, eller annan utökning, kanske inte används. Detta leder till minne som tas upp med tomma array platser.
    - Snabb tillgång till alla element i listan, sker via ett index. Det går att direkt hämta ut elementet eftersom man vet dess index.
    - Snabb tillägg och borttagning i slutet av arrayen.
        - Ifall data läggs in eller tas bort i slutet så går det snabbt eftersom det är bara att lägga till eller ta bort elementet.
        - Ifall arrayen blir full så måste den utökas. Tidskrävande.
    - Långsam tillägg och borttagning överallt förutom slutet.

        - Att lägga till eller ta bort element som inte är i slutet kräver att alla element som finns till höger om den valda platsen måste flyttas ett steg till vänster.
        - Förflyttningen av element tar tid.
        - Ifall arrayen blir full så måste den utökas. Tidskrävande.
    - Bästa användningsområde ifall man bygger listan endast från slutet och sedan läser man från den. Dessa operationer går snabbt att utföra vid en array-baserad implementation.
    - Dåligt ifall operationer behöver ske på områden som inte är slutet.
- Länknings-baserad implementation
    - Interna noder som länkas ihop med varandra, där varje nod innehåller elementet och en referens till nästa nod (enkel länkning) och referens till föregående nod (dubbel länkning).
        - Första noden och sista noden kommer vara specialfall och ha null referenser för föregående- respektive nästa nod.
        - Specialfall kan undvikas genom att använda Sentinel noder / vakt noder.
    - Enkellänkad
        - Kan endast gå framåt.
        - Varje nod känner endast till nästa nod.
    - Dubbellänkad
        - Kan gå fram och bakåt.
        - Varje nod känner till nästa nod och föregående nod.
    - Startnod: Head eller First.
        - Referens till vilken nod som är starten.
        - Om denna inte finns kan man inte gå igenom listan eftersom det finns ingenstans att börja.
    - Slutnod: Tail eller Last.
        - Referens till vilken nod som är i slutet.
        - Inte nödvändig för att få en länkad struktur att fungera. Men väldigt praktiskt vid insättning och ifall en dubbellänkad struktur används.
    - Mycket effektiv insättning i början och i slutet, endast i slutet ifall man har en referens till slutet.
        - Referenser behöver endast ändras på så en ny nod finns i början eller i slutet.
    - Effektiv borttagning.
        - Framförallt i början. Endast länka om det första noden till noden efter den första.

        - I slutet om dubbellänkad struktur. Endast backa ett steg med sista nod referensen och sätta den nya sista nästa till null och den gamla sista förra till null.
    - Mindre effektiv att lägga till eller ta bort data mot mitten.
        - Iterering genom alla element för att hitta rätt plats tar tid eftersom man måste följa varje referens efter referens tills man kommer rätt.
    - Ineffektiv att hitta ett specifikt element, som via index.
        - Iterering genom alla noder för att hitta rätt element tar tid eftersom man måste följa varje referens efter referens tills man kommer rätt.
    - För att ändra på listan måste man oftast ha flera referenser genom iterationen över listan. Skiljer sig åt ifall det är en enkellänkad eller dubbellänkad lista. För att ta bort eller sätta in ett element i mitten av en enkellänkad lista måste man hålla koll på nuvarande element och föregående element eftersom det finns ingen föregående nod referens för noderna. Detta är inte ett problem med dubbellänkad lista eftersom
    - Minneseffekt. Det är direkt proportionerligt med datan som ligger lagrad.
        - Dock så kommer varje element ta mer data eftersom de ska sparas inom en nod. Varje nod per element tar lite plats.
        - DOCK! En array med lika många platser som listan tar mindre minne eftersom i en länkad lista så kommer varje element ha ett objekt som tar minne.

Stack

- Last In First Out (LIFO) struktur.
- Vanliga Operationer
    - Lägga till element.
    - Ta bort element.
    - Kolla på det översta elementet.
    - Tömma.
- Begränsad lista där man endast:
    - Sätt in element överst i listan.
    - Ta bort element överst i listan.
    - Begränsningarna tillåter att datan som hanteras av denna datastruktur måste arbetas på ett specifikt sätt, vilket är bra för dem användningsområdena som de används till.
- Array-baserad implementation.
    - Array där alla operationer sker i slutet av arrayen.
    - Toppen av stacken representeras i slutet av arrayen.

    - Heltals Index som håller koll på toppen av stacken så att värden läggs in och tas ut på rätt ställen.
    - Snabb borttagning och tillägg eftersom man har en index plats för vart dessa ska ske.
    - Dåligt ifall vi når slutet av arrayen och måste öka den. Kostsam process som tidigare beskrivits med att öka antalet array platser och kopiera över.
    - Dåligt ifall arrayen är lång och datan aldrig används. Oanvända platser i arrayen.
- Länknings-baserad implementation.
    - Enkelläkning räcker.
    - Referens till första noden. Där första noden representerar toppen av stacken.
    - Tillägg läggs till på förstaplatsen, billigt eftersom noders referenser endast behöver länkas om.
    - Borttagning billig eftersom man har en referens till första noden och sedan länkar man bara om.
    - Operationer ske endast i början vilket är optimalt för en länknings-baserad implementation.
    - Ifall stacken inte ska vara så stor, man vet en maxstorlek och det är inte så stor så kommer en full stack ta mer plats i minnet jämfört med en array eftersom nod objekten tar plats.
- Stack är en mer specialiserad datastruktur och är bättre att använda än en Lista för dess användningsområden. En list datastruktur har mycket mer funktionalitet vilket betyder att det objektet kommer ta mer plats för att göra sådant vi inte behöver. Genom att använda en stack så kommer det objektet endast innehålla det som är möjligt för vårt användningsområde. Det blir inte bara ett objekt som innehåller mindre kod. Men kod som är mer specialiserad på att hantera just sitt område. Det är också riskabelt att använda en List eftersom vi som programmerare kan manipulera listan på sådana sätt som vi inte vill, vilket tar bort hela konceptet med att vi ska dela upp i abstraktioner.
- Användningsområden.
    - Balansering av symboler.
    - Postfix uttryck.
    - Konvertera infix till postfix.
    - Iterativ lösning till rekursiva problem.
        - Man kan själv skapa en stack som används istället för att använda datorns stack för programmets exekvering när man lägger stack-frames efter varje metodanrop på stacken.
    - För att hantera UNDO funktionalitet.
        - Dock så är oftast det ingen ren stack, men har liknande struktur.

    - mm.

Kö

- First In First Out (FIFO) struktur.
- Vanliga Operationer
    - Lägga till element.
    - Ta bort element.
    - Kolla på det första elementet.
    - Tömma.
- Begränsad lista där man kan:
    - Sätta in element sist i listan.
    - Hämta ut element först i listan.
    - Ingen index är intressant, antingen är du först eller sist.
    - Begränsningarna tillåter att datan som hanteras av denna datastruktur måste arbetas på ett specifikt sätt, vilket är bra för dem användningsområdena som de används till.
- Array-baserad implementation.
    - Arrayen används som en logisk cirkel. Om man är vid sista elementet och går ett steg till höger kommer man till första.
    - Två heltal index som pekar ut vart början av kön finns och vart slutet av kön finns.
    - Logisk så att kön faktiskt implementera en logisk cirkel måste implementeras och kräver då lite resurser.
    - Ifall maxstorleken nås, och man inte har några begränsningar, så måste den kostsamma processen att utöka en array genomföras.
    - Ifall den logiska cirkeln upprätthålls så är det snabb tillägg och borttagning eftersom man har index till vart i arrayen man ska jobba. Dock så blir det långsammare ifall logiken för att upprätthålla cirkeln måste köras.
- Länknings-baserad implementation.
    - Referenser till första noden och sista noden i kön.
    - Tilläggning sker i slutet av kön, går snabbt eftersom man har en referens till sista noden. Sedan behöver man endast länka om referenser.
    - Borttagning sker i början av kön, går snabbt eftersom man har en referens till första noden. Sedan behöver man endast länka om referenser.
    - Operationerna sker endast i början och i slutet, vilket är optimalt för en länkad-struktur.

    - Ifall kö inte ska vara så stor, man vet en maxstorlek och det är inte så stor så kommer en full kö ta mer plats i minnet jämfört med en array eftersom nod objekten tar plats.
- Kö är en mer specialiserad datastruktur och är bättre att använda än en Lista för dess användningsområden. En list datastruktur har mycket mer funktionalitet vilket betyder att det objektet kommer ta mer plats för att göra sådant vi inte behöver. Genom att använda en kö så kommer det objektet endast innehålla det som är möjligt för vårt användningsområde. Det blir inte bara ett objekt som innehåller mindre kod. Men kod som är mer specialiserad på att hantera just sitt område. Det är också riskabelt att använda en List eftersom vi som programmerare kan manipulera listan på sådana sätt som vi inte vill, vilket tar bort hela konceptet med att vi ska dela upp i abstraktioner.
- Användningsområden:
    - Buffrar. Brukar då ha en maxstorlek, vilket löser utökning problemet med arrayer. Dock så kostar det fortfarande konstant använt minne.
    - Kö till en skrivare med jobb som ska utföras.
    - Kö för anslutningen till en server.
    - Inom grafteori.
    - mm.

Sentinel node

- Noder som placeras längst fram, som en första nod och alternativt längst bak, som en sista nod.
- Dessa noder är tomma och en länkar till varandra ifall listan är tom.
- Fördelar
    - Gör så att alla specialfall för en enkellänkad eller dubbellänkad struktur försvinner eftersom första och sista riktiga noderna kommer alltid länka till någon och/eller bli länkade till via sentinel noderna.
    - Detta ökar hastigheten någorlunda eftersom inga kontroller för NULL specialfall behövs i koden. (ENLIGT NÅGRA MEN INTE ENLIGT ANDRA)
- Nackdelar
    - Dessa noder öka minnet som behövs för att spara en datastruktur någorlunda.
    - Passar inte bra ifall man vill spara en datastruktur på en hårddisk eftersom sentinel noderna bör inte sparas, mer komplex sparning och inläsning. (LITE OSÄKERT OM DETTA STÄMMER 100%)

Iteratorer

- Ett objekt som vet hur den ska ta sig igenom en datastruktur och vet vart den befinner sig i den.

- Kan ses som att den sitter mellan det föregående element i datastrukturen och nästa element i datastrukturen.
- Kan hämta ut element och kontrollera ifall det finns nya element att hämta ut.
- Den behöver inte gå igenom en datastruktur direkt när den skapas. Kan börja gå igenom, vänta lite och sedan fortsätta gå igenom. Har en egen känsla för strukturen.
- För vissa iteratorer kan även den ta bort element som den är vid.
- För vissa iteratorer kan man lägga till element som den är vid.
- Blir invalida ifall strukturen förändras. Detta kan vara ifall man sätter in data i listan innan iteratorn är tas bort. Att ta bort data via remove gör den inte invalid, men ta bort data utan iteratorn kan göra den invalid.
- Effektiv genomgång av strukturer, den kan själv hålla koll vart den befinner sig. Framförallt för länkade strukturer där den kan hålla kolla på vart i länken den befinner sig.
    - Att hämta ut ett element i en länkad struktur via .get() exempelvis leder till att man alltid måste börja från början, alternativt slutet ifall man har dubbellänkning och det är kortare från det hållet. Denna genomgång sker då vid varje .get(). Dvs, iterera via index. Fungerar för en array-implementation där man har index men inte för en länkad-implementation där man måste räkna elementen för att komma till rätt index.
    - Via en iterator så kan man först hämta ut första elementet, sedan gå vidare ett steg och hämta ut nästa. Man kan iterera på ett mer effektivt sätt.
    - Ifall borttagning eller tillägg ska ske på flera element utöver i listan där man måste iterera över listan blir en iterator mer effektiv.

Intern Klass

- Klasser som endast är användbara inom en annan klass, såsom en nod till en lista, kan implementeras direkt i klassen som en; nested class eller inner class.
- Konventioner behöver inte följas med private instansvariabler eftersom den klassen finns inom en annan klass endast.

Algoritmanalys

- Ordo (Big O)
    - Notation för att beskriva hur en algoritm växer.
    - Möjliggör jämförlig i tillväxthastighet.
    - Kan vara för tid eller minne eller annan mätbar resurs.
- Anger endast den dominerande faktorn. Konstanter och koefficienter tas oftast bort, för detaljerat.

- Jobbar med en enkel modell.
    - Alla operationer tar en tidsenhet att genomföra. Har oändligt med minne.
    - Nackdelar att jobba med en enkel modell.
        - I verkligheten kommer inte allt ta 1 tidsenhet. I verkligheten kan det då ta längre tid.
        - I verkligheten har vi inte oändligt minne. Ifall vi fyller RAM minnet i verkligheten kommer vi börja använda hårddisk minne, vilket kommer göra att algoritmen förmodligen körs långsammare eftersom läsning och skrivning till hårddisk minne tar lång tid.
- Språk specifikt och ospecifikt.
    - Ordo kan användas för alla språk och alla algoritmer för varje språk. För de flesta fallen så kommer tidskomplexiteten vara samma mellan olika språk.
    - Men, det finns situationer då olika språk kan vara implementerade på olika sätt vilket leder till att man måste fortfarande ha kunskapen om ett språk innan man försöker analysera det.
- Det som kommer påverka komplexiteten är de termer som varierar. Detta gäller både för minne och för tid. Kolla noggrant efter vad som varierar.
- Utgå alltid från det sämsta fallet. Ifall något är öppet för diskussion, tänk alltid att implementationen är den sämsta möjliga.
    - Ifall det står List<> så kan det antingen vara ArrayList<> eller LinkedList<>. Operationer för dessa kan vara olika beroende på vad man gör.
- Använda flera variabler ifall det är nödvändigt. Allt måste inte vara n, och använd inte n om det inte är n. Det skulle kunna vara n och k så svaret blir n*k.
- Tumregler för tidskomplexitet
    - Vanliga programsatser
        - O(1)
        - Konstant tid.
    - Enkla loopar
        - O(n).
        - Linjär tid.
        - Kolla allt i loop huvudet. Beroende värden kan loopen variera i komplexitet.
        - De kontroller och ökningar som finns i huvudet kommer att ske för varje loop varv, det är som om de sker i loop kroppen.
    - Logaritmiska algoritmer
        - För att det ska vara logaritmiskt så måste en representativ del av problemet försvinna vid varje loop eller varv.
        - Detta kan vara att problemet halveras. Såsom binarysearch.

        - Ifall att problemet blir mindre med en konstant, såsom 1 mindre, för varje element så kommer det fortfarande vara O(n).
    - Satser i följd
        - Addera deras tidskomplexitet. Vilket kommer endast resultera i att den satsen som tar längre tid är den vi använder eftersom vi använder endast den mest dominerande termen.
    - Kroppen i loopar / nästlade loopar
        - Loop kroppens komplexitet multipliceras med loopens komplexitet.
        - Glöm inte att vissa saker i huvudet sker i kroppen. I vissa taskiga fall kan det vara nödvändigt.
    - If / else
        - Ta testets komplexitet och addera det med den grenen som har värst komplexitet.
- Vanligaste Ordos.
    - O(1)
    - O(log n)
    - O(n)
    - O(n log n)
    - $O(n^2)$
    - $O(2^n)$
    - Finns fler, solklart, men dessa är de vanligaste.

Träd

- Har en root nod som allt börjar vid.
- Förgrenar sig med grenar/edges till andra noder.
- Noder utan barn är lövnoder.
- Subträd
    - En sammangänhande del av trädet.
    - En rot nod för ett subträd är noden högst upp i trädet.
- Relationer
    - Varje nod har en förälder nod, den som finns över noden. Detta gäller inte rot noden för hela trädet.
    - Noder kan ha barn, de noders som de relaterar till.
    - Noder som har samma förälder nod är syskon noder.
    - Man kan även definiera farföräldrar och barnbarn noder om man vill.
    - Noder som är ihopkopplade kallas ibland för ancestor och descendant mellan varandra.
- Hierarkisk

    - Trädet och dess norder går att rita upp i nivåer, som en hierarkisk struktur.
- Icke-linjär struktur
    - Man kan gå olika håll i trädet och de olika vägarna som man kan gå hör inte ihop.
- Träd lämpas sig för rekursiva lösningar.
    - Alla operationer och liknande görs rekursivt. Vi delar upp trädet i subträd för varje anrop.
    - Enkelt att förstå och implementera. Ifall vi har ett balanserat träd så kommer vi ha extremt liten risk att få Stack Overflow eftersom vi behöver endast lägga till log n stycken stack frames på stacken.
    - Kan lösas med loopar. För vissa operationer kan det vara mer eller mindre lätt. Men för andra kan det vara en väldigt komplicerad struktur. Hastighets vinsten som man vinner genom att implementera en loop och inte använda rekursion är oftast inte tillräckligt, eftersom man får en mycket mer avancerad och komplicerad lösning som exempelvis kan ta mer minne eller vara svårare att förändra och förbättra senare. Men, för att detta ska vara sant så gäller det att trädet är balanserat så det faktiskt har operationer på O(log n) och inte O(n).
- Djup / Depth
    - Avståndet från root noden till längsta barnnoden räknat i nivåer.
- Höjd / Height
    - Antalet bågar upp från en viss nivå till root noden.
- Traversering
    - Inorder
        - Går igenom trädet och processar det i ordningen som noderna ligger
        - Rekursivt anrop till vänstra subträdet.
        - Gör något själv.
        - Rekursivt anrop till högra subträdet.
        - Ifall trädet har en viss struktur som vi vill platta till så används denna oftast.
    - Postorder
        - Processering av båda subträdet krävs för att nuvarande nod ska kunna göra något:
        - Rekursivt anrop till båda barnens subträd.
        - Gör något själv.
        - Används ifall måste få reda på information från barnen innan man kan göra något med roten. Såsom att räkna ut djupet eller höjden.
    - Preorder

        - Processering sker i nuvarande nod, innan man skickar processeringen vidare längre ner:
        - Gör något själv.
        - Rekursivt anrop till båda barnens subträd.
        - Används ifall man måste jobba med roten för varje subträd innan man kan gå vidare till barnen. Såsom att skriva ut den hierarkiska strukturen för ett träd.
    - Level-order traversal
        - Alla noder på en och samma depth körs innan noder på nästa depth körs.
        - Detta kan inte göras med rekursion utan detta görs med användningen av en queue.
- Implementationer
    - Länkad struktur med nod klasser.
        - En nod klass har referenser till alla andra barn som den kan ha.
        - Alt. En nod klass har referenser till ett barn och nästa syskon.
            - Lämpar sig bra ifall man har många olika syskon eftersom man inte slösar plats per nod objekt med referenser som kanske sitter null.
            - Lättare att också komma åt syskonen ifall det är vad användningen av just detta träd kräver.
        - Oftast enkellänkad struktur. Man kan endast nå barnen men inte sin förälder. Dubbelläkning kan användas ifall man vill kunna nå föräldern.
        - Trädets storlek är proportionerligt med antalet noder.
        - Det är lätt att ändra strukturen ifall det skulle behövas eftersom noder är endast referenser till varandra.
        - Det är billigt att lägga in och ta bort ifall det är ett balanserat sökträd eftersom man kan snabbt hitta och ändra referenser.
    - Datastruktur baserad med nod klasser.
        - Varje nod har en vald datastruktur som dem använder för att referera till alla dess barn. Såsom en länkad lista som håller alla barnen.
        - För och nackdelar kommer från dem datastrukturerna som de använder sig av.
    - Arraybaserad.
        - En array som håller hela trädet.
        - Första noden sätts in på index 1, eftersom man använder matte för att räkna ut vart deras barn ligger.

        - Man sparar utrymme per nod eftersom man inte behöver någon nod klass.
        - Fungerar inte bra ifall antalet barn kan variera mycket. Fungerar bra med få barn, såsom för ett binärt träd.
        - Vi får hål i arrayen, vilket leder till att vi slösar minnesplatser. Ju fler barn som kan finnas, ju fler hål får vi.
        - Det är lätt att hämta data och flytta runt data eftersom deras plats beräknas av en förbestämd formell.
        - Andra för och nackdelar som man får av arrayer kan dras in här lite.
- Sökträd
    - Ett träd som har dess noder sorterade på ett sätt som gör det möjligt att söka på ett effektivt sätt. Givet att trädet är balanserat.
    - Trädet har strukturen så att noder till vänster för en given nod är alltid mindre och noder till höger från en given nod är alltid större. Detta ställer kravet att datan är comparable eller att det skickas med en comparator.
- Binära Träd
    - Ett träd där noderna endast har två stycken barn.
    - Implementeras oftast med en nod klass som har två referenser, en till vänstra barnet och en till högra barnet.
- Vanliga Operationer
    - Add
    - Remove
    - Contains
    - Size
    - ToString
    - Implementation (Utgår från Binärt Sökträd)
        - Rekursiv struktur.
        - Tillägg och Borttagning går ut på att man börjar i rot noden. Sedan använder man trädets egenskaper, såsom det till vänster är mindre och det till höger är större och går igenom trädet. Vid varje rekursivt anrop så säger man att man ska sätta sin högra eller vänstra nod till resultatet av nästa rekursiva anrop. Så varje rekursiv anrop är som att man sätter sin högra eller vänstra gren till det nya sub trädet som uppstår vid borttagning eller tillägg.
        - Vid tillägg så letar man bara efter den den null plats genom att jämföra nuvarande nod mot den värdet.
        - Vid borttagning så letar man efter noden som ska tas bort. Ifall man hittar noden och det är en barn nod är det bara att returnera subträdet null, dvs som inte innehåller barnet. Ifall det finns ett

barn så är det bara att returnera barnets subträd som det nya subträdet. Annars så letar man reda på det högra subträdets minsta värde. Kopierar upp det och börjar en borttagning av det minsta värdet på det högra subträdet som startar om borttagnings processen.
        - Vid contains, size, depth calculation är det bara att rekursivt gå igenom och antingen leta, eller räkna mitt subträds värde på +1 osv. När man kommer till löv noderna så returnerar de bara 1 eller något och så bubblar return värdena upp genom trädet.
- Struntar oftast i dubbletter.
    - Antingen helt och hållet.
    - För primitiva typer kan varje nod ha en räknare som uppdateras för att reflektera hår många av ett en typ har satts in. Så ifall det finns typ 10 st insättningar av siffran 5.
    - Alternativt kan dubbletten läggas in som vänster eller som höger barn. Detta skapar en svårare logik eftersom platsen till vänster eller höger kan ha barn, vi får ett extra speciall fall att hantera detta med. Dessutom så blir det svårare logik med rotationerna eftersom den bygger på att de noder till vänster respektive höger alltid är mindre eller större. Med dubbletter är det inte så och balanseringen skulle kunna bli dålig.
- Lazy Deletion
    - En strategi där man har en boolean eller liknande för varje nod som håller reda på ifall den noden finns i trädet eller inte. Ifall man hanterar dubbletter med att använda en räknare fungerar detta bra eftersom då kan man bara subtrahera räknaren, ifall den blir 0 så är den borttagen, fast med lazy deletion så finns den kvar i trädet med räknare på 0.
    - Ifall man tar bort en nod så markeras endast noden att den tas bort. Detta gör att inga resurser behövs för att ta bort noden och kanske då balansera om trädet så den hamnar i balans.
    - Ifall man sedan lägger in värdet igen så är det bara att avmarkera att den är borttagen och den sitter redan balanserat i trädet.
    - Nackdelen med detta är att trädet storlek kommer inte vara representativt med vad som faktiskt finns logiskt. Om många noder tas bort kommer trädet fortfarande använda lika mycket minne som det skulle göra som när det var som störst. Dessutom kommer det slösa tid ifall det blir för många noder som man måste gå igenom för att hitta rätt. Den interna stigen mellan olika noder blir längre och längre ju fler noder som finns i trädet.
    - Lite mer komplex kod måste skapas för att hantera dem fallen då en nod är borttagen.

Binärt Sökträd

- Ett träd som uppfyller kravet av att vara binärt, endast ha två barn, och vara ett sökträd, det ska vara effektivt att söka i det.
- Operationerna går OFTAST O(log n) för ett sådant sökträd eftersom problemet kan teoretisk halveras för varje nod man går till. Ska man gå till vänster eller höger från nuvarande nod?
    - Det finns ett stort problem! Binärt Sökträd kan få ett träd som är obalanserat. Om vi sätter in en mängd siffror som är sorterade kommer trädet mer eller mindre se ut som en enkellänkad lista. Vilket kommer ha O(n) tidskomplexitet för operationerna.
    - För att fixa detta måste man har ett krav på balansering, vilket vissa olika träd har.
- Andra krav från Sökträd och Binärt träd ska såklart finnas med här.

AVL Träd

- Ett Binärt Sökträd som har krav på att vara auto balanserat.
- För varje nod kollar dem om dess subträds längsta depth skiljer sig mer än 2. Dvs, nivåerna för subträden får max skiljas med 1 nivå. Ifall det skiljer sig med 2 nivåer så måste denna nod balansera om sig.
    - Detta kan lösas genom att varje nod har information om på vilken höjd den ligger på. Då kan subträden effektivt ta fram högsta djupet och rapportera till den givna noden. Den noden kan då avgöra ifall den måste balansera om sig eller inte.
- Balansen är inte perfekt. Men, det skulle vara för kostsamt att hålla det perfekt. Balansen är tillräckligt bra som gör att sökning fortfarande kan gå fort.
- Kontrollerna sker vid operationerna. Insättning och Borttagning.
    - När noder sätts in eller tas bort så måste varje nod, från noden som togs bort eller sattes in balans kontrolleras. Dvs, innan man returnerar ett subträd där det lagts till eller tagits bort noder måste en balansering köras. Det finns ingen riktig skillnad mellan insättning och borttagning.
    - Man börjar från den förändrade noden och jobbar sig uppåt och balanserar varje nod och dess träd tills man kommer till rot noden.
    - Balanseringen sker ju endast ifall för en given nod subträden skiljer sig åt med 2.
    - Detta implementeras som en rekursiv struktur där retur värdet för varje subträd först balanseras innan det returneras till noden ovan som kallade på metoden för den givna noden. Eftersom den rekursiva strukturen är byggt på sättet att en nod sätter sitt högra subträd lika med resultatet från en rekursivt anrop kan man i rekursionen balansera varje träd innan

det returneras. På så sätt får man strukturen att alla noder från root noden och den påverkade noden kontrolleras och balanseras.

- När en balansering sker görs en så kallad rotation. Ifall vägen från den obalanserade noden och påverkade noden. Ifall de 3 första noderna har en linjär struktur kan man göra en enkel rotation (single rotation). Ifall den har en zig zag struktur görs en dubbel rotation (double rotation).
- Enkel rotation (Single rotation)
    - Sker vid en linjär struktur av noder som måste justeras.
    - Principen går ut på att flytta upp en nod och ner med en annan. Det sker en omlänkning så vi får en ny root nod.
- Dubbel rotation (Double rotation)
    - Sker vid en zig zag struktur av noder som måste justeras.
    - Principen går ut på att först sker en enkel rotation så vi får en linjär struktur. Sedan kan man köra en till enkel rotation på den linjära strukturen.
- Rotation med barn
    - Ifall det finns barn så länkas dem också om, vi vet att noder som finns till höger och vänster om en annan nod har vissa egenskaper. Så vi kan med säkerhet flytta över dem till rätt ställen ifall vi följer egenskaperna.

Alternativa träd

- Splayträd
    - En annan typ av binärt sökträd. Har en effektivitet mellan O(log n) och O(n), men som oftast är O(log n).
    - Bygger på principen att data som nyligen används kommer förmodligen användas igen snart.
    - När en nod ska hämtas så flyttar man upp och balanserar så att den noden är roten för trädet. Ingen information om height behöver sparas vid noderna eftersom den balanserar sig vid varje uttagning då en nod ska flyttas upp till rot platsen. Enkel rotationer och Dubbel rotationer sker för att flytta upp noden.
    - Datan som flyttas upp, och närliggande data kommer vara väldigt snabbt tillgängligt.
    - Från början kan operationerna vara O(n), men i och med att man hämtar data några gånger som flyttas upp så blir det en sidoeffekt att trädet blir mer balanserat vilket leder till att operationerna tillslut blir O(log n).
- M-Way träd / B-Tree
    - Ett träd där man har M stycken barn.
    - Har en balansering så att trädet har snabbt åtkomst, fast med flera barn.

    - En implementation av M-Way träd är ett B-Tree. Vilket jag kommer skriva om nedan från och med nu:
    - Datan delas in block. Där varje block har då en gräns som beskriver vilka element som får plats. Dessa beskrivningar läggs in i ett block ovanför. Detta block är då parent noden. Sedan så har det blocket en parent nod som har beskrivningar till vilket spann som dess barn block ligger inom. Förenklat så har rot noden M stycken spann som beskriver vilken gren man ska gå för ett visst värde. Om man följer den grenen så kommer man till ett barn som också har M stycken spann som beskriver vilken gren man ska gå för ett visst värde. Den grenen kanske går till ett block av data där man hittar alla element som ligger i ett visst span.
    - Är balanserat på det sättet att spannen är designade att inte innehålla för lite och inte för mycket data. Ifall det fylls så skapas flera spann så man kan dela upp barnen mer. Ifall det försvinner data kan man slå ihop spann så att varje block av data inte har för få element i sig.

TreeSet

- En samling av unika element. Det förekommer inga dubbletter.
- Är själv sorterande då den är implementerad med ett sökträd.
- Är en Collection, så det som går att göra med List går även med Set.
- Operationer
    - Lägg till element.
    - Ta bort element.
    - Söka efter element på ett effektivt sätt.
- Sökning sker på O(log n) i värsta fall.
    - Balanserat Binärt sökträd används i Java.
    - Top-down red-black tree används oftast.
- Behöver att elementen är Comparable eller att det skickas med en Comparator.
- Svårt stöd av iterator klassen, s.174 för olika möjliga sätta att göra det på.

TreeMap

- En samling av nycklar som kan associeras med värden.
- Nycklarna måste vara unika och ligger sorterade i trädet.
- Operationer
    - Kolla om en nyckel existerar.
    - Hämta värdet för en nyckel.
    - Lägg in ett nyckel-värde par.
- Sökning sker på O(log n) i värsta fall.
    - Balanserat Binärt sökträd används i Java.
    - Top-down red-black tree används oftast.

- Iteration
    - Har ingen iterator
    - Man kan hämta ut en Set med alla Nycklar.
    - Man kan hämta ut en Collection med alla Values.
    - Man kan hämta ut en Set med nyckel-värde paren.
- Svårt stöd av iterator klassen, s.174 för olika möjliga sätta att göra det på.

Hashtabeller

- Datastruktur med fokus på konstant operationstid för borttagning, tillägg och hämtning.
    - Många saker påverkar operationstiden och kan försämra den. Såsom dåliga hashfunktioner.
- Elementen lagras i en array men det finns ingen ordning bland elementen. Lämpar sig inte för applikationer med sortering.
- Använder sig av en hashkod som de sparade objekten kan räkna fram för att avgöra vart i arrayen objektet ska sparas.
    - Objektets hashkod modulus storleken på arrayen är den vanligaste interna hashfunktionen.
    - Det är viktigt med en hashfunktion som ger en spridning i tabellen. Även ifall tabellen är väldigt stor.
    - Om hashfunktionen är snabb så får man fram indexvärdet för ett objekt väldigt fort, konstant fort. Man kan då direkt gå till det index och hämta u elementet. Beroende på kollisionshantering så kan man behöva söka igenom några element, men dessa är oftast få, inte N i storlek.
- Load Factor
    - Hur stor del av tabellen som är full. En load factor på 0.5 betyder att halva arrayen är full.
- Vid en specifik Load Factor utökar tabellen sin arrays storlek. På samma sätt som en ArrayList skulle göra det på.
- Rehash
    - Vid en utökning av den interna arrayen måste objekten hashas på nytt med den ny tabellens storlek.
- Primtal
    - Storleken på arrayen är oftast ett primtal. När man ökar storleken så ökar man till nästa primtal som är ungefär dubbelt så stor.
    - Använder primtal i Hashfunktionerna för att räkna fram hashkoderna.
    - Primtal är ej delbart med andra tal vilket ger dem en unik egenskap som man hoppas ska ge bättre spridning på hashfunktionen.
    - Ifall man inte använder ett primtal som hashtabell storlek så finns det risk vid open adress kollisionshantering. Ifall vi förflyttar oss med en steglängd

som är en faktor av tabell längden kan vi få loopar som endast går igenom vissa platser och då loopar för alltid om alla dem platserna är upptagna.
    - Svårt att bevisa att primtal är bättre i alla situationer, men det är åtminstone inte sämre, så man har fortsatt traditionen.
- För att objekt ska kunna sparas i en hashtabell måste dem i Java implementera hashCode funktionen och equals funktionen. HashCode för att få ut objektets hashkod. Equals för att kunna jämföra ifall man sätter in dubbletter eller inte.
- Några fördelar
    - Med förutsättningarna att man har en bra hashfunktion så kommer man få extrem snabb operationstid.
    - Konstant insättning, sökning och borttagning.
- Några nackdelar
    - Beroende på hur stor arrayen är så kommer den ta upp platser som kanske inte används. Andra datastrukturer som Träd eller liknande använder bara de platser som de behöver för elementen.
    - Man måste designa en bra hashfunktion för att hashtabellen ska fungera bra.

Hashfunktioner

- Används internt i en hashtabell för att få fram ett index där ett objekt ska sparas.
    - Oftast är det hashkoden från objektet modulus tabellens storlek.
- Varje objekt har en hashfunktion, hashCode, för att kunna räkna ut sin egna hashkod.
- Ska ge bra spridning så att objekt får egna index platser. Mindre kollision leder till bättre prestation och bättre användning av tabellens platser.
    - Viktigt att tänka på att tabeller kan vara stora. Så det är viktigt att hashkoden blir stor så att elementen kan spridas ut på alla tabellens platser.
- Ska gå snabbt att räkna fram.
    - Hashkoden kan cashas ifall det inte går snabbt att räkna fram den.
    - Detta är ett exempel på tid-minne avvägning.
- Ska vara baserade på objektets interna data. Datan den baserar sin uträkning på kan inte vara mutable eftersom då kan man få nya hashkoder vid ett senare tillfälle om man ändrar något i objektet. Vilket leder till att ett objekt inte då längre kan hittas om det sparats i hashtabellen.
- Det är viktigt att analysera det man använder för att skapa hashkoden. Hur kommer dessa datafält att se ut? Vilka mönster har den. Detta är viktigt för att man ska få en så bra spridning som möjligt.
    - Baserar man det på en faktura vars siffra alltid ökar med ett uppåt?

    - Eller baserar man det på ett datum där det bara finns vissa intervall av siffror?
    - Text är också oftast lite begränsat eftersom ord tenderar att ge upprepningar för vissa kombinationer. Vilket skulle kunna påverka vilken spridning vi får.
    - En bra hashfunktion ser till så vi får en spridning för dessa. Men att använda bra datafält gör jobbet lättare.
- En hashfunktion blir sällan perfekt och man måste förmodligen testa sig fram lite inom sitt område för att få en tillräckligt bra spridning.
- Universella Hashfunktioner
    - Hashfunktioner som är designade för att fungera med det mesta. Fungerar oftast ganska bra.
    - Mer vanligt att specialisera hashfunktioner för sin applikation.

Kollisioner

- Sker ifall två objekt hashar till samma plats i tabellen. Detta kommer alltid att ske.
- Kan undvikas mer ifall man har en bra hashfunktion som ger en bra spridning.
- Load Factor
    - N / TabellStorlek.
    - Hur full tabellen är.
    - Olika kollision strategier har olika optimala Load Factor gränser för att de ska fungera snabbt. Ifall Load Factor gränsen når sitt max värde vill man utföra en rehash så att man får en större tabell och lägre Load Factor. Detta påverkar prestandan på hur snabbt de olika kollisions strategierna körs, men även ifall de ens kan köra som det står mer om längre ner.
- Open-address
    - Vid kollision söker man framåt i arrayen för en ny ledig plats. Använder sig av olika matematiska formler för att ta steg framåt.
    - Ifall man kommer till slutet av arrayen så börjar man om med att söka från början, en rundgång sker.
    - Clustering
        - Nackdel vid Open-addressing.
        - Element tenderar att samlas ihop och bilda kluster av upptagna platser.
        - Försämrar hastigheten eftersom man måste gå igenom hela klustret för att hitta en ledig plats.
        - Finns Primary Clustering och Secondary Clustering.
    - Påverkar tid av operationer eftersom det finns en risk att man måste gå flera steg framåt för att hitta en ledig plats eller rätt objekt.

- Lazy Deletion måste användas.
    - Man kan inte ta bort ett objekt. Endast markera dem som borttagna.
    - Ifall man skulle ta bort objekt skulle man inte kunna hitta andra objekt som har hashats till samma index men var tvungen att sökas framåt.
- Har en Load Factor på 0.5. Ifall man kommer upp till den Load Factorn måste man göra tabellen större och rehasha.
- Det är viktigt att tabellen är ett primtal så att vi kan rotera runt med quadratic och double hashing på ett effektivt sätt, annars riskerar vi att hamna i en oändlig loop om vi har en för hög Load Factor och en tabell storlek som inte är ett primtal.
- Linera Probing
    - Vid kollision så går man 1 steg framåt i arrayen och försöker sätta in. Detta upprepas med +1 tills man hittar en ledig plats.
    - Skapar Primary Clustering.
        - Stora kluster block av data. Man kommer förmodligen behöva gå många steg för att hitta en ledig plats eller rätt data.
- Quadratic Probing
    - Vid kollision så går man i^2 steg framåt i arrayen vid varje försök tills man hittar en ledig plats. Så +1 först, +4 sen, +9 sen osv.
    - Insättnings försöken blir mer utspritt vilket löser Primary Clustering.
    - Tabellens storlek måste vara ett primtal och vi vill ha en Load Factor på mindre än 0.5, annars finns ingen garanti att vi inte hamnar i en oändlig loop. Ifall de kraven möts så kommer vi oftast hitta en plats snabbt.
    - Skapar Secondary Clustering
        - Objekt som hashas till samma värde kommer förmodligen behöva gå många steg för att hitta en ledig plats eller rätt data.
- Double Hashing
    - Vid kollisions används en ny hashfunktion för att avgöra vilket steg som ska tas. Varje kollision eller icke hittat objekt går man i * hash2(hashkod) framåt. Hash2 funktionen blir steglängden.
    - Kostar mer att använda en annan hash funktion för att räkna ut steglängden. Tabellens storlek måste vara ett primtal för att double hashing inte ska ge några problem med clustering.

        - Ifall Double Hashing implementeras korrekt får vi en mycket bättre spridning vid kollision och undviker både Primary och Secondary Clusters. Dock så liknar detta Quadratic Probing ganska mycket med resultatet vi får, fast är långsammare eftersom vi har två hashfunktioner.
    - Fördelar
        - Man jobbar direkt mot elementen och behöver inte gå via en annan datastruktur. Vid små mängder data är denna form mer effektiv. Men så fort vi börjar öka storleken för att hålla en Load Factor på 0.5 blir det inte lika minnes effektivt.
    - Nackdelar
        - Man behöver utföra rehashing oftare eftersom man endast vill ha en Load Factor på 0.5 för att garantera att probing teknikerna inte tar för lång tid på sig.
        - Beroende på probing teknik kan clustering vara ett problem som gör att det tar längre tid att hämta fram element.
        - Lazy Deletion gör att datan inte försvinner och tar upp plats i minnet. Dessutom ger detta en lite mer complex logik eftersom det kan var svårt att veta
- Separate chaining
    - Vid varje array plats finns det en datastruktur, exempelvis en lista.
        - Ex: en enkellänkad lista. Tar lite mindre utrymme eftersom den är enkellänkad och uppfyller alla kraven vi behöver. Snabb insättning i början och sökningen igenom listan.
    - Vid en kollision så sparar man elementen den listan.
    - Oftast så sparas elementet först i datastrukturen så det går snabbt att hämta den efter insättning. Oftast så används element som man precis har satt in.
    - Fördelar
        - Fördelarna är att man alltid garanterat kommer att ha plats vid kollision. Och rehashing behöver endast ske när man får en Load Factor på 1, vilket är betydligt senare än för en Load Factor på 0.5.
        - Man utökar arrayen vid en Load Factor på 1. Så att man inte fyller listorna alltför mycket.
    - Nackdelar
        - Nackdelen är att varje datastruktur kommer ta plats. Det är mer minneskrävande med denna lösning.
        - För vissa språk så kan det vara tidskrävande använda andra datastrukturer beroende på språk och dess implementation. Alla språk är inte lika snabba som Java på vissa saker exempelvis.

        - Nackdelar som den valda datastrukturen tar.

Prioritetskö

- En speciell typ av kö där element läggs in i kön beroende på dess prioritet och tar ut den som har högst prioritet.
- Används inom specialiserade situationer.
    - Operativsystem process köer.
    - Graf algoritmer.
    - Grigia algoritmer.
    - Hitta k:th elementet.
    - Simulationer.
- Operationer
    - Insättning
        - Lägger in ett element på den plats i kön som den har prioritet för.
    - Uttag
        - Tar ut det element som har högst prioritet.
    - Unika jämfört mot en vanlig kö eftersom dem jobbar på elementens prioritet.
- Implementationer
    - Länkad Struktur / Array
        - Sorterad
            - Insättning blir O(n) eftersom man måste leta reda på rätt plats att lägga in elementet.
            - Borttagning blir O(1) eftersom det alltid är första, eller sista, elementet.
        - Osorterad
            - Insättning blir O(1) eftersom vi kan sätta in längst fram, eller längst bak, i listan vid varje insättning.
            - Borttagning blir O(n) eftersom vi måste leta igenom listan för att hitta elementet med högst prioritet.
        - Generellt sätt så skulle man säga att den osorterade varianten är bättre eftersom man alltid kommer göra fler insättningar än borttagningar, om inte lika många.
    - Binärt träd
        - Binärt träd kommer hålla våra element i en sorterad struktur. Alla med högst prioritet längst till vänster och lägst prioritet till höger.
        - Operationer för insättning och borttagning kommer att ha en O(log n), i förutsättningen att trädet är balanserat.

        - Fungerar inte lika bra med prioritetsköer eftersom vi hela tiden kommer ta bort från vänster, vilket kommer leda till högerbalanserat träd, eller att rotationer sker ständigt.
        - Varje nod tar minne jämfört med ett annat sätt att representerad det på, såsom en heap.
        - Har mycket mer funktioner som vi inte behöver.
    - Heap - standard.
        - Det sättet som man använder för att implementera en prioritetskö.
        - Mer om det under rubriken Heap.

Heap

- En trädstruktur som implementeras med hjälp av en array och matematiska formler.
    - Arrayen använder inte plats 0, eftersom det blir konstigare matte oftast.
    - För en nod på plats i gäller för en binär heap att:
        - i * 2 är första barnet.
        - i * 2 + 1 är andra barnet.
        - i / 2 är föräldern.
- Oftast är det en Binär Heap, vilket är ett Binärt träd. Men det kan också vara d-heap, där varje nod har d barn.
    - Med D stycken barn har en nod plats i följande:
        - d * (i - 1) + 2 för första barnet.
        - (i - 2) / d + 1 för föräldern.
    - Ju större värde på D, ju bredare kommer trädet att vara, dvs flera barn per nod. Detta kommer gör att sökningen för vilket barn som är mindre bli längre, men kommer också göra att mängden hopp nedåt kommer sjunka.
    - Ju bredare ju snabbare går insättning. Vi behöver göra färre jämförelser uppåt för att komma till föräldrar noden.
    - Ju högre ju snabbare går borttagning. Vi behöver göra färre jämförelser med barnen och kan snabbare hitta det minsta barnet.
    - I teorin brukar man säga att en 4-heap är bättre än en 2-heap
- Använder sig av prioritetsköns egenskaper.
    - Det enda viktiga värdet att kunna hitta är det enda värdet som har högst prioritet.
    - Strukturen är på ett sådant sätt att värdet med högst prioritet ligger först i listan.
- Rotationerna i detta trädet är enklare och färre en om man skulle använda ett balanserat AVL träd.

- Har O(log n) för sina operationer. I och med att det är ett perfekt balanserat träd. Komplett Binärt Träd.
- Är ett Komplett Binärt Träd, dvs att det finns inga hål i trädet och alla noder fylls på från vänster till höger på barn raden längst ner. Detta har effekten att insättning går snabbt eftersom vi vet alltid vart vi ska lägga in.
    - Med ett komplett binärt träd är vi säkra på att operationerna tar O(log n) eftersom vi inte har några hål i trädet att förhålla oss till.
- Använder mindre minne än de andra alternativen eftersom sparar elementen direkt i en array, inte i omslukande klasser som en nod.
    - Med matten så går det snabbt att hoppa fram och tillbaka mellan barn noder och föräldrar noder, vilket annars skulle behöva vara länkar som tar upp minne.
- Varje nod i trädet har mindre, eller lika prioritet som barn noderna. Det är det enda kravet som måste uppfyllas.
- Insättning
    - Vi lägger in elementet på första lediga plats.
    - Ifall det inlagda elementet har lägre prioritet än förälder elementet är vi klara.
    - Ifall det inlagda elementet har högre prioritet än förälder elementet så måste vi byta plats på dem. Detta steg upprepas tills förälder noden har högre prioritet än det inlagda elementet, eller att vi kommer till rooten.
    - Det blir som att värdet vi sätter in bubblar upp, vilket kallas för percolate up.
    - I värsta fall tar det O(log n), men oftast blir den klar lite tidigare än så.
- Borttagning
    - Vi tar bort elementet som finns på root platsen eftersom det är värdet med högst prioritet. Nå har vi ett hål högst upp som måste rättas till.
    - Börja med att kolla om det senaste tillagda elementet kan ligga på hålets plats, om det går så är vi klara. Annars, kolla bland hålets barn och hitta barnet som har högst prioritet. Byt plats på hålet och barnet med högst prioritet. Detta steg upprepas tills vi hållet kommer så långt ner att det inte finns några barn.
    - Det senaste inlagda elementet flyttas då till hålets plats eftersom den kan vara där, det finns inga barn som kan ha högre prioritet.
    - Det blir som att värdet hålet bubblar ner, vilket kallas för percolate down.
    - I värsta fall har borttagning O(log n).

Sortering

- När sorteringsalgoritm ska väljas är det viktigt att tänka på en mängd olika saker. Eftersom varje algoritm har dess för och nackdelar som gör att vissa av dem passar bättre in än andra för dessa olika påverkningar:
- Påverkningar
    - Typ av data
        - Kan datan jämföras snabbt?
        - Kan datan flyttas runt snabbt?
    - Typ av datastruktur
        - Fungerar sorteringen bra ifall man skickar in en annan datastruktur än en specifik array.
    - Ordningen på data
        - Kommer algoritmen att fungera bra ifall datan är sorterad eller osorterad?
    - Mängd data
        - Hur fungerar algoritmen med små mängder data?
        - Hur fungerar algoritmen med stora mängder data?
        - Små mängder data - jämförelse med loopar oftast mer effektivt.
        - Stora mängder data - rekursivt divide-and-conquer oftast mer effektivt.
    - Mängd tillgängligt minne
        - Finns det minne i main memory att jobba med?
        - Det som påverkar här är framför allt ifall algoritmen är in-place eller inte. Ifall den är in-place så tar den mindre minne och fungerar ifall vi har lite minne.
        - Man vill undvika att spara data på sekundärminnet eftersom då kommer sorteringen utan tvekan ta långtid.
        - Om man har lite minne blir in-place viktigt.
    - Stabila eller inte
        - Kommer datan med samma storlek att ligga i samma ordning efter sorteringen?
        - Används för multi-kriterier sortering, så ifall man har flera krav på sortering. Då är det viktigt att de som är lika ligger bredvid varandra i korrekt ordning när man sorterar på ett annat kriterier.
        - Framförallt intressant för key-value sortering där vi har dubbleringar för key:na.
    - Språk implementation
        - Olika språk har olika implementationer som påverkar vilken algoritm man vill använda. Hur bra språket fungerar med jämförelse och byten etc.

        - I Java används referenser vilket gör att alla förflyttningar går snabbt. Medans i C++ kan man flytta pekare eller hela objekt.
        - I Java tar det längre tid att jämföra objekt med varandra än primitiva typer, vilket påverkar algoritmen. Medans i C++ så går oftast snabbare att jämföra objekt eftersom jämförelse operationerna kan optimeras för inline.
- Stack minnet som finns är redan allokerat innan programmet. Så minnet som tas upp av rekursion räknas inte mot heap minnet.
- Man måste tänka olika för en enkellänkad lista och en dubbellänkad lista.
- Viktigt att tänka på ifall algoritmen som man använder jobbar med random access till datan eller inte. Ifall det är random access så kommer länkade listor passa sämre eftersom då måste man gå långa steg, som att välja mittersta pivo värdet. En array passar mycket bättre med random access. Så ifall vi har en länkad lista är det bättre att använda en array som inte använder random access. Ifall den använder det eller inte måste man fundera. Dvs, kan vi implementera algoritmen på ett sådant sätt att vi inte behöver köra O(n) varje gång ifall vi har en länkad lista.

Selection Sort

- Implementation
    - Hittar det minsta värdet som och sätter in det på förstaplatsen beroende på passering.
    - För varje passering söker man igen 1 mindre element, eftersom alla placerade värden till vänster är sorterade.
- In-place algoritm då den jobbar inom en och samma array.
- Är inte stable per automatik, men man kan justera så den blir stable.
- Typ av datastruktur
    - Fungerar bra med andra typer av datastrukturer som Linked List.
- Ordningen på data
    - Kommer alltid köra $O(n^2)$ på både sorterad som osorterad data.
- Mängd data
    - Snabb på att sortera liten mängd data i jämförelse med Rekursion. ingen overhead kostnad.
- Jämförelser
    - $O(n^2)$
    - För varje passering sker många jämförelser, eftersom man måste jämföra med alla resterande värden för att hitta det minsta värdet.
- Byten
    - O(n)

    - Byten sker endast en gång per passering, i slutet för att flytta elementet till början av listan.
- Övre Gräns
    - $O(n^2)$
- Undre Gräns
    - $O(n^2)$

Insertion Sort

- Implementation
    - Värden väljs en efter en och så letar man reda på rätt plats där värdet ska sitta för att arrayen ska vara sorterad.
    - Börjar block från i = 1 och så ökar man i++ per passering.
- In-place algoritm då den jobbar inom en och samma array.
- Är stable per automatik.
- Typ av datastruktur
    - Fungerar bra med andra typer av datastrukturer som Linked List
- Ordningen på datan
    - Ifall den är osorterad kommer den köra på $O(n^2)$.
    - Ifall den är sorterad kommer den endast köra O(n).
- Mängd data
    - Snabb på att sortera liten mängd data i jämförelse med Rekursion. ingen overhead kostnad.
- Jämförelser
    - $O(n^2)$
    - Behöver bara jämföra fram tills den hittar en plats som den ska sitta på.
    - Ifall listan är sorterad kommer det gå fort att hitta vilken plats som varje värde ska sitta på, de platser de redan har.
- Byten
    - $O(n^2)$
    - Byten kan ske flera gånger varje passering, den inre while loopen kommer att flytta på alla värden som finns till höger för insatt plats. Kostsamt att göra många byten.
    - Med en sorterad lista kommer det inte behöva ske några byten och det kommer gå fort.
- Analys
    - Övre Gräns
        - $O(n^2)$
    - Undre Gräns
        - O(n)

    - I och med att den har en bättre körtid mot redan sorterad data så är denna implementation förmodligen bättre än selection sort.
    - När det finns delar som är sorterade eller ifall hela listan är sorterad så kommer det gå fort att sortera. Eftersom den snabbare hittar rätt plats att sätta.
    - Detta väger upp att Insertion Sort kan ha många Byten.

Bubble Sort

- Implementation
    - Antar att listan är sorterad.
    - Börjar gå igenom listan och om man hittar två element som står bredvid varandra och är i fel ordning, byt plats på dem. Markera att listan inte är sorterad.
    - Ifall listan inte är sorterad. Antag att den nu är sorterad och kör igenom en gång till.
    - Effekten blir att rätt värden bubblar upp till sin rätta plats.
- In-place algoritm då den jobbar inom en och samma array.
- Är stable per automatik.
- Typ av datastrukturer
    - Fungerar bra med andra typer av datastruktur som Linked List.
- Ordningen på datan
    - Ifall den är osorterad kommer den köra på $O(n^2)$.
    - Ifall den är sorterad kommer den endast köra O(n).
- Mängd data
    - Snabb på att sortera liten mängd data i jämförelse med Rekursion. ingen overhead kostnad.
- Jämförelser
    - $O(n^2)$
    - Mellan alla element.
    - Många jämförelser kommer ske eftersom ett värde som står på första platsen och ska till sista måste jämföras med alla andra element ett steg i taget tills den kommer upp till sista platsen.
- Byten
    - $O(n^2)$
    - Varje gång det är två värden som har fel ordning.
    - Många byten kommer ske eftersom ett värde som står på första platsen och ska till sista måste bytas med alla andra element ett steg i taget tills den kommer upp till sista platsen.
- Analys
    - Övre Gräns

        - $O(n^2)$
    - Undre Gräns
        - O(n)
    - Är generellt sett väldigt ineffektivt eftersom det sker väldigt många byten och jämförelser.
    - Det tar lång tid för ett element som finns längst ner att bubbla upp, innan den kommer till rätta plats.
    - Enda fördelen den har, är ifall listan redan är sorterad går det fort. Men, detta gör redan Insertion Sort, så det är inte en direkt vinst, eftersom Insertion Sort blir även bättre ifall delar av listan är sorterad.

Shellsort

- Första algoritmen att bryta $O(n^2)$ körtid.
- $O(n^{3/2})$ beroende på vilken variant man använder.

Heapsort

- Implementation
    - En heap används för att sortera. Min-heap eller Max-heap beroende ifall man vill ha fallande eller stigande ordning.
    - Är perfekt balanserat, inga hål, så operationer går på O(log n).
    - Enkel implementation skulle använda en ny array som man fyller genom att köra deleteMin på heapen.
    - Mer effektiv operation så återanvänder man arrayen.
    - Man börjar med att köra perlocade upp på alla element, dvs, simulerar att man lägger in alla element i listan.
    - Sedan när heap strukturen är skapad kör man deleteMin så att första elementet plockas bort. Detta gör så att heapen måste sorterar om sig själv, vilket leder till att platsen längst bak blir ledig, där placerar man elementet från deleteMin. Sedan kör man deleteMin igen och så ges en ny tom plats näst längst bak där man kan placera elementet.
    - Detta upprepas om och om igen.
- In-place algoritm då den jobbar inom en och samma array.
- Är inte stable per automatik. Två element med samma prioritet har ingen ordning.
- Typ av datastrukturer
    - Fungerar inte så bra med andra datastruktur som Linked List. Bättre att skapa en array med alla platser i så fall.
- Ordningen på datan

    - Har alltid O(n log n) eftersom den inte bryr sig om ordningen på något sätt, följer endast heap strukturering. Så sämre ifall datan redan är sorterad.
- Mängd data
    - Snabb på att sortera liten mängd data i jämförelse med Rekursion. ingen overhead kostnad.
    - Inte lika snabb att sortera större mängder data som Merge och Quick, men bättre än d e andra icke-rekursiva alternativen.
- Jämförelser
    - Inget speciellt. Följer heap strukturen endast.
- Byten
    - Inget speciellt. Följer heap strukturen endast.
- Analys
    - Övre Gräns
        - O(n log n)
    - Undre Gräns
        - O(n log n)
    - Minneseffektiv eftersom den jobbar inom sig själv, in-place.
    - Effektiv sortering och har alltid en säker prestanda, men den presterar inte så bra ifall datan är sorterad som andra algoritmer.
    - Enkel icke-rekursiv kod. Det finns ingen overhead information som vi rekursion.
    - Har dålig locality of reference. Dvs, en hårdvara optimering där det är enkelt att förutspå hur data kommer hämtas och då utföra operationerna parallellt. I en heap liknar det mer som en slumpad arbete.
    - Inte jättevanlig i praktiken.

Mergesort

- Implementation
    - Jobbar rekursivt genom att dela upp problemet i mindre problem som ska lösas. Sedan sätter man ihop dem två mindre lösta problemen.
    - Bygger på principen att kunna väldigt snabbt och effektivt sätta ihop två sorterade listor.
        - Använder en extra array för att spara den nya ihopsatta listan.
    - Så genom att dela upp problemet i två delar och be dem delarna att sortera sig och sedan slå ihop två sorterade listor kommer gen en konstant körtid för hopslagning och logaritmisk tid för att problemet halveras varje rekursivt anrop.
        - O(n) för att sätta ihop två sorterade listor och O(log n) för att dela upp problemet i mindre delar.

    - När listan blir tillräckligt liten, mellan 10-20 element så försöker man inte dela upp listan logaritmiskt mer, utan då använder man en annan algoritm för att sortera den lilla listan.
- Inte en in-place algoritm per automatik. Den använder en extra array vid ihop merging av sublistor.
    - Ifall vi använder en Länkad Lista kan vi skriva om algoritmen så att den blir in-place. (?)
- Är stable per automatik.
- Typ av datastrukturer
    - Fungerar bra med andra typer av datastruktur som Linked List. Är förmodligen den bästa metoden att sortera en Linked List.
    - Eftersom Linked List data bygger på länkar, dessa länkar kommer då vara utspridda i minnet vilket gör att hårdvara optimeringar inte kan användas och att i Mergesort jobbar mycket med att slänga runt och ändra vilket går bra när vi använder länkar. I quicksort måste vi göra partitions stegen vilket tar längre tid för en Linked List. I Mergesort har vi inte några sådana operationer som tar extra lång tid för just Linked list.
- Ordningen på datan
    - Spelar ingen roll eftersom den kommer ändå dela upp datan.
    - Samma körtid för sorterad som osorterad data.
    - Fungerar dock bra med sorterad data och kan justeras så det går snabbare.
- Mängd data
    - Snabb på att sortera väldigt stora mängder data. Halverar problemet varje gång genom att rekursion.
    - När listan blir för liten, 10-20 element, så använder man en icke-rekursiv variant. Det är mer effektivt eftersom varje rekursivt anrop är lite kostsamt, och med dessa små värden vinner man inte så mycket av att dela upp det i två delar, som man gör med riktigt stora värden.
    - Ex: Insertion Sort.
- Jämförelser
    - Optimalt antal jämförelser / färre jämförelser.
- Byten
    - Lite mer byten av arrayer, för att merga två sorterade listor exempelvis.
- Analys
    - Övre Gräns
        - O(n log n)
    - Undre Gräns
        - O(n log n)

    - Är helt beroende på datan som ska sorteras ifall denna metod är bättre än quicksort.
    - I och med denna algoritm jobbar mer med byten så måste datan gå snabbt att flytta runt.
    - Används i situationer där datan går snabbare att flytta runt än att jämföra.
    - I Java går det snabbt att flytta runt värde, eftersom objekt är endast referenser. De är alltid referenser om de är objekt, inte som i C++ där användaren skulle kunna använda ett objekt och inte pekare.
    - I C++ kan det gå snabbt att flytta runt värden, ifall man använder pekare. Men om man inte gör det så kommer det inte gå snabbt.
    - I och med att man använder Optimalt antal jämförelser är mergesort bättre ifall datan där jämförelser är mer kostsam. Att jämföra två objekt i Java är mer kostsamt eftersom man måste anropa en metod. Jämförelsen kan inte ske inline. Objekten som jämförs kan dessutom ha långa och komplexa jämförelser metoder som tar tid.
        - Till skillnad från C++ där överlagra jämförelse operationer och skriva dem som inline för att öka prestandan så man inte behöver anropa en ny metod.
    - För Primitiva typer i Java kan man använda en annan metod som Quicksort eftersom jämförelse kan ske inline där och inga externa anrop behöver ske. Det är inbyggt i hårdvaran hur man kan effektivt jämföra värden.
    - Fungerar bättre i Java för Generiska typer eftersom den generiska typen är vi säkra på alltid kommer vara en referens och kommer gå snabbt att byta runt men inte lika snabbt att jämföra beroende på objektets jämförelse metod.

Quicksort

- Implementation
    - Jobbar rekursivt genom att dela upp problemet i mindre problem som ska lösas genom att välja ett pivot-värde, jämförelse värde. Sedan sätter man ihop dem två mindre lösta problemen till vänster och höger om pivot-värdet.
    - Börja med att välja ett Pivot-värde.
        - Första elementet (sista elementet har samma)
            - Bra ifall man använder Linked List, snabb åtkomst till första värdet.

                - Hemskt dåligt ifall datan är sorterad eftersom då kommer hela listan hamna i ena gruppen för varje rekursivt anrop, vilket resulterar i att man får en $O(n^2)$ körtid.
            - Mittenvärdet
                - Fungerar bättre ifall datan är redan sorterad och ifall det är en väldigt slumpad sortering.
                - Fungerar inte bra för länkade listor, eftersom det tar tid att hämta mitten värdet.
            - Slumpmässigt
                - Bra i praktiken. Man slipper troligtvis att man får det största eller minsta värdet för varje nivå.
                - Problem med länkade listor igen…
            - Median-of-three
                - Man väljer ut 3 element, första, mitte och sista.
                - Sedan väljer man det mittersta värdet av dem.
                - Fungerar extremt bra.
                - De andra två värdena kan sorteras direkt vid utval av pivot.
            - Ifall länkade lista så kommer pivot värdet vara dyrare att hämta ut vilket kan påverka valet.
        - Sedan dela upp arrayen i två delar, en del för värden mindre än Pivot-värdet och en del som är större än Pivot-värdet. Anropa rekursivt quicksort för dessa två grupperingar.
        - När listan blir tillräckligt liten, mellan 10-20 element så försöker man inte dela upp listan logaritmiskt mer, utan då använder man en annan algoritm för att sortera den lilla listan. Exempelvis kan Insertion Sort användas.
        - Det finns varianter där man har en tredje grupp för att sortera in värden som är lika med pivot. Detta är bra ifall man har många dubbletter. Annars vill man försöka dela upp dubletter så det finns lika många dubletter i båda grupperingarna.
            - Om denna grupp inte finns och det finns många dubletter så kommer quicksort att prestera sämre.
- In-place algoritm då den jobbar inom en och samma array.
- Är inte stable per automatik. Kan göras, men ger sämre effektivitet.
- Typ av datastrukturer
    - Fungerar bra med andra datastrukturer som Linked List, MEN beror på pivot värdet som kan vara krävande att hämta fram och försämra prestandan.
    - Eftersom Linked List data bygger på länkar, dessa länkar kommer då vara utspridda i minnet vilket gör att hårdvara optimeringar inte kan användas

och att i Mergesort jobbar mycket med att slänga runt och ändra vilket går snabbt ifall vi använder länkar. I quicksort måste vi göra partitions stegen vilket tar längre tid för en Linked List. I Mergesort har vi inte några sådana operationer som tar extra lång tid för just Linked list.

- Ordningen på datan
    - Pivot värdet har all makt här. Dålig strategi kommer ge en dålig körtid för sorterade listor. Men, ifall en bra strategi används gäller detta:
    - Spelar ingen roll eftersom den kommer ändå dela upp datan.
    - Samma körtid för sorterad som osorterad data.
    - Men, ifall vi vet datan är osorterad så är quicksort bra för då kan vi välja första elementet för pivot värde.
- Mängd data
    - Snabb på att sortera väldigt stora mängder data. Halverar problemet varje gång genom att rekursion.
    - När listan blir för liten, 10-20 element , så använder man en icke-rekursiv variant. Det är mer effektivt eftersom varje rekursivt anrop är lite kostsamt, och med dessa små värden vinner man inte så mycket av att dela upp det i två delar, som man gör med riktigt stora värden.
    - Ex: Insertion Sort.
- Jämförelser
    - Fler jämförelser.
- Byten
    - Färre byten.
- Analys
    - Övre Gräns
        - O(n log n)
    - Undre Gräns
        - O(n log n)
    - Gränsvärdena är beroende på vilken pivot strategi vi använder.
    - Används i situationer där datan är snabbare att jämföra en att exempelvis byta på. Men ifall datan går snabbt att jämföra vill man använda denna form oftast.
    - I Java för primitiva värden så använder man hellre quicksort eftersom det går fort att jämföra dem typerna av data. De kan göras inline och är oftast optimerade för hårdvaran.
    - I C++ brukar man också använda quicksort eftersom det går fort att jämföra i C++ då objekt, som primitiva typer, kan jämföras inline. Men detta är situationsspecifikt.
    - Det sker också färre byten så det är bättre i C++ där användaren skulle kunna använda objekt och inte pekare.

    - Fungerar bättre i C++ för generiska värden eftersom Objekt jämfört oftast snabbare i C++ än Java och det finns en risk att Objekt används och inte pekare då användaren bestämmer den generiska typen.
    - Dålig pivot strategi kan försämra allt hos Quicksort, det kommer påverka hur många varv vi måste köra.

Bucket Sort / Counting Sort

- Implementation
    - För att kunna använda denna så måste datan som ska sorteras kunna omvandlas till en positivt siffra. Denna siffra kommer användas för att placera ut i arrayen och räkna hur många sådana som finns. Därför fungerar detta bäst för siffror.
    - Ta reda på det största möjliga siffran M, skapa en ny array med M platser. Ifall M är stort så är detta inte en minnes effektiv lösning.
    - Gå igenom listan sedan och läsa av dess siffervärde. Använd detta siffervärde som index plats i den nya arrayen, array[indexSiffra], och öka med 1. Antalet som i slutet finns i array[indexSiffra] är antalet element med den indexSiffra värde.
- Är inte in-place eftersom den skapar en ny array för att spara alla värden i som kan vara stor, minnes ineffektivt.
- Är stable per automatik.
- Typ av datastrukturer
    - Fungerar bra med andra typer av datastruktur som Linked List eftersom den skapar ändå en ny array.
- Ordningen på datan
    - Bryr sig inte eftersom den skapar en ny array där den håller data.
- Mängd data
    - Snabb på att sortera liten mängd data i jämförelse med Rekursion. ingen overhead kostnad.
- Jämförelser
    - Sker inga jämförelser.
- Byten
    - SKer inga byten.
- Analys
    - Övre Gräns
        - O(n)
    - Undre Gräns
        - O(n)
    - För att ens kunna använda denna metod så måste siffer kravet fyllas.

    - Ifall den största siffran är väldigt stor så är denna metod väldigt minnes ineffektiv.
    - Det finns mer utvecklade metoder som bygger på Bucket Sort som är mer minneseffekt framförallt.
    - Bucket sort används för att lägga distinkta objekt i högar så att vi kan skilja på dem, buckets.
    - Counting sort används för att lägga in värden som är samma, såsom primitiva värden. Så ifall vi har många dubbletter och få värden fungerar denna bra.

Grafer

- Ickelinjär datastruktur som består av noder och kanter.
- G = (V, E)
    - V - En mängd av noder.
    - E - En mängd av kanter mellan noderna.
- Det kan finnas en hierarkisk struktur, att en nod kommer före en annan. Men det behöver inte finnas det.
- Träd är en specialform av en graf.
    - Grafen måste vara sammanhängande.
    - Det får inte förekomma några cyklar.
- Terminologi
    - Kant
        - En förbindelse mellan två noder.
        - Kan vara riktad eller oriktad.
        - Kan vara värderad / viktad eller ovärdera / oviktad.
    - Komplett / Fullständig
        - Varje nod är sammankopplad till alla andra noder.
        - Ganska ovanliga men förekommer.
        - Dessa är oftast viktade för annars finns det ingen direkt mening med dem.
    - Angränsande
        - Två noder som har en kant mellan sig.
    - Väg / Stig
        - Sekvens av kanter mellan noder.
        - Kanterna utgör en väg / stig man kan gå.
    - Oriktad
        - En kant gäller åt båda hållen.
    - Riktad
        - En kant gäller endast åt ett håll.
    - Viktad / värderad

        - Varje kant har ett värde till sig.
        - Kan vara kostnad, vikt eller kapacitet som är associerad med en kant.
        - En oviktad graf kan ses som en graf där alla kanter har värdet 1.
        - Kan ha negativa värden, men detta gör så att algoritmerna här inte fungerar och man måste ha mer avancerade algoritmer för att lösa problemen. Oftast ovanligt.
    - Cykler
        - Man börjar och slutar på samma punkt för en väg / stig.
        - Man går som i en cykel i grafen, detta tillåter loopande vägar inuti grafen.
        - En kant gås endast en gång.
    - Loop
        - En kant som går till sig själv.
        - En kant från A till A.
    - Multigraf
        - Det kan finnas flera kanter mellan en och samma nod.
- Implementation
    - Nodklass / Adjacency List
        - Varje nod som finns i grafen kan vara sin egna klass.
        - Noderna har en datastruktur, såsom Lista, för att hålla reda på vilka andra noder som denna nod sitter ihop med. Dvs kanterna.
        - Ifall relationerna är viktade måste värdena finnas med. Detta kan göras med att ha en kant klass som läggs in i datastrukturen.
        - Hashmap gör att det går fort att hitta en nod i grafen och få ut dess relationer.
        - Fördelar
            - Bra ifall man jobbar med noden i sin graf.
            - Exempelvis undersöka vilka noder som sitter ihop med en viss nod. Dvs, vilka kanter går ut från mig.
            - Minnes effektiv då man endast sparar information om vad som faktiskt finns. Framförallt ifall grafen är gles, det saknas många kanter. I jämförelse med Adjacency matrix framförallt.
        - Nackdelar
            - Inte effektivt ifall man jobbar med att hitta något i grafen om man inte har en nod att utgå ifrån och måste då söka igenom alla noder.
            - Man måste alltså ha en nod att utgå ifrån för att det ska vara effektivt.

            - Hitta alla billigaste kanterna är ineffektivt eftersom man måste söka igenom alla noder.
    - Samling av kanter
        - Grafen har en datastruktur som samlar alla kanter. En kant har två noder som den finns emellan.
        - Noderna fås ut genom att undersöka kanterna.
        - Lätt att hitta kanterna för grafen, dem finns i en datastruktur.
        - Fördelar
            - Bra ifall man vill jobba med kanterna för grafen.
            - Ifall man vill kunna snabbt hämta ut dem och sortera dem, såsom Kruskal algoritm för minimalt uppsänt träd.
        - Nackdelar
            - Inte effektivt ifall man vill jobba med noderna.
            - Exempelvis att hitta alla kanter som går ut från en nod ger en ineffektiv genomsökning av kanterna.
            - Eller att undersöka om en relation finns mellan två noder.
    - Matris / Adjacency Matrix
        - Tvådimensionell array av alla noder. $|V|^2$ stor array.
        - I cellen där X och Y möts läggs informationen in. Position (X, Y) kan läsas som X → Y och (Y, X) som Y → X.
        - Fördelar
            - Det går snabbt att undersöka ifall det finns en relation mellan två noder som man vill undersöka. Endast att utföra en (X, Y) eller (Y, X) konstant tid lookup i arrayen.
        - Nackdelar
            - Minnes Ineffektiv ifall det inte finns massvis med kanter mellan noderna. Dvs om det fattas många kanter blir det många hål. Om det är en komplett graf fungerar det bra.
            - Blir inte så effektiv ifall man jobbar med noderna eftersom man måste hitta dem genom deras relationer. Hitta alla kanter från en nod kräver genomsökning av en hel rad/kolumn.
            - Blir inte effektiv ifall man jobbar mot alla kanter heller utan noderna mellan en kant.

Depth-First Search

- Hitta en väg från en nod till en annan nod.
- Man går en väg så långt man kan tills man gått fast, då får man börja backa. Som preorder traversal i ett träd.
- Behöver inte vara den kortaste vägen.

- Går snabbare att hitta en väg eftersom man inte försöker hitta den kortaste. Tar mindre minne och är mer effektiv. Så ifall det är mer intressant att undersöka om det finns en väg kan denna metod vara mer effektiv.
- Då man jobbar med Noder och undersöker vilka Noder som den är ihopkopplad med är Adjacency List ett bra val. Man kan utifrån en nod då snabbt undersöka vilka valmöjligheter man har.
- Algoritmbeskrivning
    - Fokus ligger på att gå en väg genom att följa kanterna tills man hittar slutet. Ifall man fastnar och inte kan gå vidare går man tillbaka ett steg, backtrack, tills man hittar en ny väg att gå.
    - Noder som har blivit undersökta / passerade markeras så att man inte går till en sådan nod igen.
    - Börjar med att lägga till första Noden på stacken, markerar den som besökt och väljer en av kanterna utåt och går till den.
    - När man kommer till en ny nod, är det slutnoden? I så fall är vi klara.
    - Annars, markera noden som besökt, lägg till den i stacken och gå till nästa nod.
    - Ifall man fastnar vid en nod och inte kan gå vidare så tar man det senaste noden från stacken och testar om man kan gå därifrån. Ifall man kan det så kan man fortsätta, annars poppar man från stacken igen.
    - För varje nod så måste man hålla information om vilken nod man kom ifrån. Så noderna kan ha ett fält för föregående noden så man kan backtraka från slutnoden och hitta rätt väg.
- Algoritmen bryr sig inte något om ifall grafen är riktad eller viktad eller dess motsatser. Det enda den gör är att följa de kanter som den kan för att hitta en väg. Den kommer inte hitta den kortaste. Så det spelar inte så stor roll ifall den har vikter eller inte.
- Är mer resurseffektiv eftersom man endast sparar noder för den vägen som för tillfället fungerar. Inte som Breadth-First Search där man sparar hela nivåer noder.

Breadth-First Search

- Hitta den kortaste vägen från en nod till en annan. (Eller en av de kortaste vägarna ifall det finns flera)
- Fungerar lite olika ifall grafen är viktad eller oviktad. Dijkstras Algoritm används ifall den är viktad, vilket endast är en variant på Breadth-First konceptet.
- Är mer resurskrävande eftersom man söker efter den kortaste vägen.
- Man jobbar i lager, där varje lager definieras utifrån längden från start noden. Så alla med 1 i längd från start noden tillhör samma lager.

- En kö används för att spara varje lager. Så alla noder för samma lager läggs in samtidigt i kön. Dessa kan sedan plockas ut och processeras lagervis.
- Då man jobbar med Noder och undersöker vilka Noder som den är ihopkopplad med är Adjacency List ett bra val. Man kan utifrån en nod då snabbt undersöka vilka valmöjligheter man har.
- Algoritmbeskrivning
    - Börja med att lägga in noden som vi börjar med, dvs alla noder med avstånd 0.
    - Plocka ut en nod ur kön, vilket kommer vara vår start nod. Markera den som besökt och lägg till alla dess barn i kön. Barnen kommer vara noderna med avstånd 1. När barnen läggs till måste backtrack information sparas om vilken nod som la till den noden.
    - Sedan upprepas denna process med att plocka ut, markera, lägga in, tills man har nått alla noder i grafen. När alla noder är markerade som besökta kommer algoritmen tekniskt bara tömma kön och sedan inse att den är klar.
    - Noder som redan är besökta läggs inte till.
    - Noderna i grafen kan ha datafält för ifall dem är besökta och vilken deras tidigare nod var, samt antalet noder som man har gått för att avgöra den kortaste vägen.
    - I och med att varje nod har backtrack info kan man välja noden som man ville hitta och kolla vilken nod den har som backtrack info, sedan tar man den noden och kollar dennes backtrack info och upprepar tills man kommer till första noden. Då har man vägen man ska gå.
- Nackdel är att kön kan växa väldigt fort vilket är resurskrävande. Både i tid det tar att utöka kön (om det är en array baserad implementation) men också för att spara alla noder. Man sparar ju noder för varje nivå vilket gör att man lägger in många noder i taget, vilket man inte gör med exempelvis Depth-First Search där man endast sparar den nuvarande fungerande vägen. Flera noder som läggs in kommer inte behöva processas eftersom man kanske redan har hittat den kortaste vägen.

Dijkstra

- En typ av Breadth-First Search algoritm som hittar den billigaste vägen i en viktad graf.
- Hitta den billigaste vägen från en nod till en annan i en viktad graf. (Eller en av de billigaste vägarna ifall det finns flera)
- Använder sig av en prioritetskö istället för en vanlig kö så att man undersöker barnen för en nivå i ordning på billigaste väg.
- Annars jobbar man på samma sätt som för Bredden-Först genom att jobba i lager.

- Då man jobbar med Noder och undersöker vilka Noder som den är ihopkopplad med är Adjacency List ett bra val. Man kan utifrån en nod då snabbt undersöka vilka valmöjligheter man har.
- Algoritmbeskrivning
    - Börja med att lägga in första noden i prioritetskön så att processen kan börja.
    - Vid uttag så markeras noden som besökt och vi gör precis som i den ovanstående Bredden-Först beskrivningen och lägger till barnen i kön.
    - Vid varje inlägg i kön ska den totala kostnaden användas. Så ifall det man lägger in är en nod som det kostade 3 + 4 att komma till ska 7 vara dess kostnad.
    - Ifall man stöter på en nod som man reda lagt i kön (fast från en annan väg) kollar man ifall den nuvarande vägen har ett lägre värde än värdet som noden ligger med i prioritetskön. Ifall den nuvarande vägen har en lägre kostnad ska nodens värde i kön justeras.
    - Genom att nodernas köplats justeras med uppdatering av väg kostnader ger det ingen risk att en nod som blir markerad har en billigare väg.
    - Noder som redan markerats som besökt struntas i.
    - Noderna i grafen kan ha datafält för ifall dem är besökta och vilken deras tidigare nod var, samt totala kostnaden som man har gått för att avgöra den kortaste vägen.
- Nackdel är att kön kan växa väldigt fort vilket är resurskrävande. Både i tid det tar att utöka kön (om det är en array baserad implementation) men också för att spara alla noder. Man sparar ju noder för varje nivå vilket gör att man lägger in många noder i taget, vilket man inte gör med exempelvis Depth-First Search där man endast sparar den nuvarande fungerande vägen. Flera noder som läggs in kommer inte behöva processas eftersom man kanske redan har hittat den kortaste vägen.
- Fungerar INTE med negativa värden som kostnad. Då blir det att man vill loopa runt i med den negativa kanten så att man för varje varv sänker den totala kostnaden.

Minimum Spanning Tree

- Det billigaste trädet, utifrån kant kostnader, som man kan bygga från en graf. Blir en subgraf av grafen.
- Trädet måste vara sammanhängande, viktad och vara oriktad.
- Prim
    - Jobbar med Noderna för att hitta ett Minimum Spanning Tree.
    - Antar att grafen är tom och består endast av noder utan ihopkoppling.

    - Välj sedan en nod och koppla ihop den på det billigaste sättet med en annan nod.
    - Vid varje steg av algoritmen kommer vi ha en mängd noder ihopkopplade och en mängd noder som inte är det. Sedan letar man efter den kortaste vägen att koppla ihop en ny nod till mängden av dem noder som är ihopkopplade. Den kortaste valda vägen får inte bilda cyklar.
    - Upprepa med att välja billigaste kanten tills alla noder är ihopkopplade.
    - Liknar Dijkstras Algoritm mycket när man ska hitta den kortaste vägen mellan noderna för att koppla ihop.
    - Fungerar bättre ifall grafen sparar noderna eftersom man kan då välja en nod och effektivt kolla dess relationer och välja den billigaste av dem. Vi kan snabbt utifrån en nod undersöka billiga relationer utåt.
- Kruskal
    - Jobbar med Kanterna för att hitta ett Minimum Spanning Tree.
    - Antar att grafen är tom och består endast av noder utan ihopkoppling.
    - Välj en av de billigaste kanterna och koppla ihop dem. Ifall kanten som valdes kommer bilda en cykel ska den inte läggas till.
    - Upprepa med att välja billigaste kanten tills alla noder är ihopkopplade.
    - Fungerar bättre ifall grafen sparar kanter eftersom man kan då sortera kanterna utifrån storleksordning och enkelt plocka dem en efter en. Vi kan snabbt komma åt dem billigaste kanterna.

Giriga Algoritmer

- Algoritmdesignteknik där man jobbar med att göra val som verkar bäst nu och håller sig sedan till det valet.
- Jobbar i faser där man i varje fas försöker avgöra vad det bästa resultatet är nu, man jobbar med ett lokalt optimum och hoppas att det stämmer överens med det globala optimum.
    - Oftast så följs någon Heuristik som avgör vad som lokalt är optimum.
    - Oftast så kan ett och samma problem både har en slutgiltig global optimum eller lokal optimum beroende på hur problemet utformas. Exempelvis dijkstras kommer alltid ge den kortaste vägen, globalt optimum. Men detta är ifall det inte finns några negativa vägar, för då kommer de inte ge den kortaste vägen, suboptimal lösning.
- Efter att ett val har gjorts håller algoritmen till det och ändrar sig inte. Valet var ju lokalt optimalt.
    - Detta är det som gör den girig, att den inte ändrar sitt val. För alla algoritmer gör ju det den anser är bäst nu, men dem kan oftast ändra sig eller liknande.

- Ifall att de lokala optimum inte stämmer överens med det globala optimum i slutet av algoritmen har den genererat ett suboptimalt resultat.
- I vissa fall så stämmer det lokala optimum överens med det globala optimum och då fungerar algoritmen för det problemet, annars gör den inte det
- Ett suboptimalt resultat kan fortfarande vara av intresse.
    - Vi kanske inte behöver ha ett perfekt svar, utan är bara intresserade att det finns ett svar eller något svar.
    - Det är mindre resurskrävande att använda en girig algoritm som ger ett bra svar, även fast det inte är det bästa. Det kan alltså vara för resurskrävande att räkna fram det perfekta svaret.
    - Vissa problem kanske inte går att lösa på en rimlig tid för det globala optimum. Exempelvis ifall det finns för många kombinationer att testa.
- Olika typer av giriga algoritmer
    - Dijkstras
        - Slutgiltiga lösningen är globalt optimum.
    - Prims
        - Slutgiltiga lösningen är globalt optimum.
    - Kruskals
        - Slutgiltiga lösningen är globalt optimum.
    - Beräkna växel för pengar.
        - För våra valutor fungerar det bra eftersom siffrorna vi använder kommer alltid leda till att de lokala optimum stämmer överens med de globala optimum.
        - Men, det är lätt att problemet slutar fungera ifall vi introducerar en 12 krona. Då kommer få algoritm för 15 ge växeln 12 1 1 1, vilket inte är optimalt (10 5 är optimalt). Algoritmen gör ett val, att 12 får plats i 15 och håller sig till det, eftersom lokalt sätt så känns det bäst och då håller den till det.
    - Schemaläggning
    - Huffmankodning
    - Bin-Packing
        - Problem med att få in så mycket som möjligt på en begränsas yta, såsom en låda.
        - Varje Item har en storlek och ska packas ner i lådorna.
        - Målet är att packa ner alla Items med så få lådor som möjligt.
        - Alternativ
            - Det behöver inte vara en 1-dimensionell låda som man ska stapla items i.
            - Alla problem där man måste få ut så mycket av en yta som möjligt.

        - Kretskortsdesign där man ska få plats med olika komponenter som är olika stora på ett kort.
    - On-line
        - Packar varje item när dem får dem
        - Next Fit
            - Om item får plats i den första binen, lägg den där.
            - Annars skapa en ny bin.
        - First Fit
            - Leta efter en första möjliga platsen att lägga ner itemet på.
            - Annars, skapa en ny bin.
        - Best Fit
            - Leta efter den lediga plats som är minst där detta item får plats i. Man vill välja platsen där detta item fyller ut så mycket plats som möjligt.
            - Annars, skapa en ny bin.
    - Off-line
        - Packar varje item efter att den har läst in alla items.
        - Sorterar oftast dem på storleksordning. Tanken är att man kan packa mer kompakt ifall man lägger ner dem största items först.
        - First Fit Decreasing
            - First Fit fast man börjar med det största item, sedan tar man det näst största item osv.
        - Best Fit Decreasing
            - Best Fit fast man börjar med det största item, sedan tar man det näst största item osv.

Divide-and-Conquer Algoritmer

- Algoritmsdesigntekink där man delar upp problemet i mindre delar och löser dem mindre delarna först rekursivt för att kunna sedan slå ihop och lösa det större problemet.
- Man delar upp problemet i mindre delar och löser dem delarna, sedan sätter man ihop resultatet till det slutgiltiga svaret.
- Divide
    - Man delar upp problemet i en mängd olika delar, där varje del är disjunkta.
    - Uppdelningen sker genom ett rekursivt anrop till samma problem.
    - Varje uppdelning måste vara märkbar, såsom halvering av problemet. Att bara minska med ett element per rekursivt anrop räcker inte.

    - I och med att man delar upp det i märkbara delar måste flera rekursiva anrop göras, så om man halverar problemet kommer två rekursiva anrop att göras.
    - Man delar upp tills man har en tillräckligt liten mängd data som kan lösas på ett effektivt sätt, såsom 20 element i sortering kan sorteras med Insertion Sort istället för att dela upp ännu mera.
- Conquer
    - Lösningen till de mindre problemen används för att skapa en lösning till det större problemet.
- Closest Points
    - Ett klassiskt problem där Divide-and-Conquer används.
    - Hitta i ett plan av punkter de två punkterna som har kortaste avståndet.
    - För att lösa denna brukar man börja med att pre-processa datan, såsom att sortera alla punkter på dess X värde.
    - Dela upp planet i två delar, en vänster halva och en högre halva.
    - Det minsta avståndet som finns är att fråga den vänstra halvan vilket det minsta punkt avståndet är och sedan fråga den högra halvan vilket den minsta punkt avståndet är, detta är divide delen eftersom vi delar upp problemet i två mindre halvor och rekursivt löser dem.
    - Det finns ett tredje avstånd, där en punkt finns i end a halvan och en i andra som också kan vara minsta och måste undersökas.
    - När man har alla tre minsta avstånd så kan man undersöka vilket av dem tre som är minst och då ge tillbaka svaret, vilket är conquer delen.

Dynamic Programming

- Algortimdesignteknik där man delar upp problemet i mindre delar och sparar lösningarna för senare användning.
- Man delar upp problemet i mindre delar. Precis som Divide-and-Conquer. Men, dessa algoritmer används ifall man kan spara lösningar till subproblemenen och dem är optimala, och att subprolemen överlappar varandra så att man får användning av att man sparar undan resultatet.
- Uppdelningen av problemet kan ske rekursivt eller inte.
- Använder sig av en tabell för att spara tidigare lösta resultat.
- Det gäller att svaret som man sparar är optimalt. Ifall svaret man kommer fram till som subproblem inte är optimalt kan vi inte spara det för senare användning.
- Det gäller att problemet som delas upp har överlappningar mellan sig så att man använder sig av tabell lookups för att hitta den optimala lösningen istället för att räkna ut svaret om och om igen.
    - Detta kommer att ske eftersom uppdelningarna ska överlappa, vilket betyder att samma sub del av problemet kommer anropas flera gånger.

- All pair-shortest path
    - Man vill hitta den kortaste vägen från en punkt till en annan.
    - Man sparar resultatet för varje kortaste väg hittat.
    - Man kan då återanvända resultatet av att komma till en nod för att undersöka vilka andra noder man kan komma till och vad den totala kostnaden kommer vara.

Randomized Algoritmer

- Algoritmdesignteknik där man använder sig av slumptal för att göra något beslut under algoritmens gång.
- Används eftersom man vill inte att algoritmens körtid ska påverkas negativt beroende på indatan. Det är de slumpade talen som avgör hur bra algoritmen är, inte indatan.
- En nackdel är att man är beroende av slump genereringen av tal. Oftast så fungerar det bra med den som finns i språket, men för vissa applikationer är det viktigt att man har en slumpgenerator som kan genererar dem talen man behöver för att sin slumpade algoritm fungerar som den ska. Annars kan den bli opålitlig och inte alls fungera bra.
- Exempelvis slumpad pivo värde val för Quicksort
    - Ifall slumpad pivo värde väljs spelar det ingen roll ifall indatan är sorterad eller inte, det kommer att fungera bra ifall vi får bra slumpade tal, vilket vi i de flesta fallen kommer vi. Det är otrolig liten risk att första eller sista elementet i den inskickade sorterade listan väljs varje gång.
- Skip Lists
    - Variant på Länkade Listor som har Binär Söknings egenskaper.
    - Datan ligger i sorterad ordning.
    - Noder kan har flera referenser till noder längre fram. Hälften av alla noder har endast en länk framåt. Hälften av dem kvarvarande har sedan två länkar, så var fjärde. Hälften av dem har sedan tre länkar, så var åttonde osv osv. På så sätt kan man hoppa fram i listan och halvera problemet vid varje sökning, så O(log n).
    - Att hålla noder med olika länkar i perfekt balans är svårt att uppnå och dyrt. Vi måste bygga om större delar av listan för att se till så att den perfekta balansen hålls.
    - För att göra det mer effektivt så använder man slumpen för att bestämma vilken nivå varje insättning ska ha, dvs antalet länkar. Vi får inte en perfekt balans och det kan vara så att vi får flera med samma antal länkar i följd, men det är liten sannolikhet.
    - I genomsnitt så blir det bra.
    - Blir enklare att implementera eftersom dem inte behöver balansera om.

    - Effektivt eftersom vi behöver inte rotera och balansera om som binära träd.
    - Det blir billigt att söka eftersom stora delar av listan kan hoppas över eller utessluta eftersom vi har länkar som leder en längre bit framåt. Att listan är sorterad gör detta möjligt att avgöra vart i listan något finns.
- Används även för att följa ut tal i ekvationer för att exempelvis på ett effektivt sätt undersöka ifall ett tal är ett primtal eller inte.

Backtracking Algoritmer

- Algoritmdesignteknik där man sparar information om tidigare val så att man kan backa tillbaka och ändra dem ifall valen man gjorde ledde fel.
- Handlar om att söka efter ett korrekt alternativ på något sätt där man har möjligheten att backa tillbaka ifall man går en väg och upptäcker att detta är fel.
- Man söker igenom många kombinationer för att hitta den som fungerar.
- Är inte en brute-force algoritm eftersom man använder logik för att inte söka i vägar som man vet inte kommer leda någon vart. Dessa tekniker kallas för pruning, att inte söka vägar som man innan kan räkna fram eller vet på andra sätt inte kommer påverka resultatet eller bara ge sämre resultat. Det är bara slöseri med tid.
- Rekursion är ett sätt att implementera Backtracking eftersom det är väldigt lätt med att göra rekursionsanrop. Men man kan implementera det med en vanlig stack om man vill.
- Att söka igenom för att hitta rätt väg är långsamt eftersom det finns många möjliga kombinationer.
    - Detta kan lösas genom att använda sig av pruning, att minska mängden noder i sökträdet att faktiskt söka.
    - Vi kan även möjligtvis använda oss av Dynamic Programming för att spara resultat från tidigare sökningar ifall att olika sökningar kan bygga på varandra.
- Det är relativt enkelt att implementera. Rekursion ger en enkel implementation.
- Djupet-Först Sökning använder sig av backtracking.
- Minimax
    - En form av Backtracking algoritm som bygger på att söka efter vilket val man vill göra i exempelvis ett spel.
    - Varje position i ett spel kan utvärderas som att ha ett värde. Exempelvis kan en position där datorn vinner vara +1, där det blir oavgjort vara +0 och där spelaren vinner vara -1. Dessa slutpositioner kommer vara löv noderna för sökträdet.
    - En position vars värde kan simuleras utan att undersöka underliggande positioner kallas för en terminal position.

- Ifall att positionen inte är terminal så vill man då söka från det. Då skapas ett sökträd för spelet, vilket bara är en visualisering av rekursiva anrop.
- I sökträdet som bildas simulerar man för varje nivå att det antingen är spelaren som vill maximera värdet (såsom datorn) eller minimera värdet (såsom spelaren). Så för en nod som är maximerande kommer den undersöka sina barn och returnera det värdet av barnen som har max och ifall det är en minimerande kommer det vara tvärt om.
- För att sökningen inte ska bli för stor måste man begränsa antalet positioner man söker neråt och använda sig av någon pruning teknik för att välja bort positioner som är meningslösa.
- Man kan även spara redan utvärderade positioner i ett transposition table.
- Alpha Beta Pruning
    - Än av dem bästa optimeringarna för minimax algoritmen och jobbar på sökträdet för att minimera antalet noder som faktiskt måste sökas.
    - Bygger på att använda noder som redan har utvärderats för att avgöra ifall en nod ens behöver utvärderas.
    - Ifall en tidigare nod har utvärderats till ett större värde än vad denna nod kan komma fram till, och noden ovan är en maximerande nod så kommer den alltid att välja det redan större värdet, eftersom noden under är en minimerande nod och kommer välja det minsta värdet, vilket vi kanske redan vet kommer vara mindre än det större värdet. Detta kan vetas ifall man har undersökt ett av barnen för min noden. Ifall det barnet har ett värdet som är mindre än det större värdet på andra sidan i trädet så spelar det ingen roll ifall vi söker för andra noden. Den kommer endast välja det minsta av dem som kommer vara mindre än det större på andra sidan och sedan kommer max noden ändå att välja värdet som är större.
    - På så sätt kan vi undvika att slösa tid genom att strunta i att söka för ett helt subträd, vilket sparar mycket resurser.
    - Alpha Pruning kallas det när man just när man redan vet delar av noderna och avgör om det är värt att söka igenom ett subträd eller inte. Det implementeras genom att man skickar med ett Alpha värde som kan användas för att avgöra detta vid varje nod.
    - Beta Pruning är exakt samma sak och implementeras på samma sätt men hanterar ifall vi har en minimerande situation och inte en maximerande situation.

        - Alpha värdet kommer vara det bästa tidigare alternativet för den maximerande spelaren medans Betavärdet kommer vara det bästa tidigare alternativet för den minimerande spelaren.
        - Alpha och Beta Pruning blir då att hantera båda fallen, både ifall det är minimerande och ifall det är maximerande.
        - Alpha-Pruning gör att vi endast söker igenom sqrt(n) av alla noder i spelträdet. Detta gör det möjligt att låta datorn nå längre när den tänker eftersom den använder sina resurser optimalt.

Top-down Splay tree

- Vid insättning och sökning så sker en mängd splay rotationer så att elementet i fråga hamnar högst upp i trädet.
    - Om elementet man söker efter inte finns så kommer det senaste elementet man var vid att roteras upp till roten.
- Tanken är att trädet balanserar sig själv genom att göra detta.
- Vanligt att man utför en bottom-up rotation, där man först hittar noden och sedan roterar man upp den.
    - Varje nod kräver en parent link så att man kan gå upp, eller att man använder sig av en stack för att spara elementen på väg ner.
    - Man roterar upp noden från botten upp till toppen.
    - Tar mer minne än Top-down implementation och är mindre effektiv.
    - I Bottom Up söker vi först en gång och sedan göra vi splay rotationer. Vi får två passeringar.
    - I Top Down så söker vi och roterar samtidigt.
- Top-down är att man börjar med att rotera tills elementet man letar efter hamnar högst upp. Så man börjar direkt med att rotera top elementet.
- Implementeras genom att man har en root nod och två subträd om roten. Subträd L och R, där noder som är mindre än roten kan sparas i L och noder som är större än roten kan sparas i R. Från början används är L och R tomma och används vid rotationerna.
- Vid rotationerna uppåt så används L och R för att spara subträd beroende på ifall en zig, zig-zig eller zig-zag rotation ska utföras. Vid varje splay rotation så sparas de vänstra eller högra subträden i L eller R för att “gräva fram” rätt värde.
- Efter att rotationer har skett och man har delat upp subträden till L och R träden och elementet man letar efter är i roten sker en reassemble, då sätts subtrden från L och R ihop med roten.

Red-black Tree

- Alternativ till AVL träd.

- Har mindre strikta regler än AVL träd och är då sämre balanserat. Men, den har billigare och färre rotationer.
    - Det går lite snabbare att söka i ett AVL träd eftersom att den är mycket bättre balanserat.
    - Det går snabbare att sätta in och ta bort element hos ett röd-svart träd eftersom den har färre och billigare rotationer.
- Egenskaper för noderna
    - Varje nod är färgad röd eller svart.
    - Rooten är svart.
    - Ifall en nod är röd så måste dess barn vara svarta.
    - Varje stig från en nod till en null referens måste innehålla samma antal svarta noder.
- Vid modifiering av trädet så måste man kontroller att egenskaperna följs. Detta leder till att:
    - Varje nod sätts in som röd från början.
    - Sedan undersöker man om detta bryter mot egenskaperna.
    - Ifall det gör det börjar man med att undersöka om man kan färga om trädet så att egenskaperna återställs, ifall man kan det är detta en väldigt billig balansering.
    - Annars får man utföra single- och double rotations samt färga om för att återställa balansen.
- I och med dessa utspridda egenskaper och många så finns det oftast väldigt många olika sätt att lösa obalans på. Vissa lösningar kan då vara mycket mer effektiva, men det ger en mer komplicera kod för att kolla på vilken av dessa möjligheter faktiskt går att utföra.
- Man kan antingen jobba bottom-up eller top-down. Top-down använder lite mindre minne eftersom vi inte behöver ha parent-länkar eller en stack för att spåra vägen ner.
- Sentinel noder används oftast istället för att ha null länkar. Dessa har ett oändligt litet värde så att man alltid vill gå höger, höger länken kan länka tillbaka till roten.
- En null referens eller en sentinel nod anses vara svarta noder logiskt sätt.
- Med Top-down så sker omfärgning och ombalansering på vägen ner eftersom att säkerställa att man kan sätta in en röd nod längst ner. Samma sak vid borttagning, när man går ner och letar utför man omfärgning och ombalansering så att den noden som ska tas bort är röd, så att den kan tas bort utan problem.
- Vid borttagning för en nod med två barn så letar man reda på det minsta barnet i det högra subträdet, kopierar det och sedan letar man efter den noden för att ta bort den.
- Vid borttagning för en nod med ett barn så letar man efter det minsta eller största barnet i det högra respektive vänstra subträdet beroende vilken barn

nod man har. Fungerar på samma sätt som borttagning för två barn fast man gör endast på det tillgängliga subträdet.

Treaps

- Binärt Sökträd som använder sig av slump given prioritet för att ordna elementen enligt en heap ordning.
- Prioriteten ges med hjälp av slumpen.
- Prioriteten hos elementen ska ligga tillfredsställa heap order.
- Först placeras noden in genom att gå höger och vänster som ett binärt sökträd, sedan när den har placerats så flyttas den upp så att den uppfyller heap order. Föräldern ska inte ha lägre prioritet än barnen.
- Uppflyttningen görs med hjälp av rotationer, så att det fortfarande följer sökträd kraven.
- Balansens för ett Binärt Sökträd att vänstra noder har mindre värde och högra har större måste fortfarande hållas även fast man roterar noder för att hålla heap order.
- Att introducera slumpen gör att man är mindre beroende av indatan, så ifall man lägger in tal i sorterad ordning ska man inte få en länkad lista situation eftersom att den balanseras om via prioritets omskiftning.
- Borttagning fungerar genom att minska prioriteten hos elementet så att den roterar ner till en lövnod och borttagning blir trivial.
- Fördelen är att Treaps gör mycket färre rotationer i utbyte att det inte är lika balanserat. (?)
- Sedan är treaps också förhållandevis väldigt enkel att implementera.
- Har en O(log n) precis som binära sökträd, fast dock lite sämre i jämförelse eftersom det är sämre balansering.

k-d Trees

- Ett binärt sökträd som tillåter användningen av flera nycklar per nod. Antalet nycklar kan ses som antalet dimensioner.
- Ett vanligt binärt sökträd är 1-d.
- Vid varje nivå så skiftar man vilken nyckel i noden som man ska jämföra med.
    - Om vi har ett 2-d träd har vi 2 nycklar att jämföra med.
    - Vid roten jämför vi med nyckel 1, sedan går vi ner en nivå och använder nyckel 2 för jämförelse, sedan går vi ner en sväng och då använder vi nyckel 1 igen osv.
- Lazy Deletion används för att ta bort saker. Detta eftersom att man kan inte ta bort noder eftersom då skulle sökningen inte fungera eftersom en nod vars nyckel fanns inte finns längre och kommer leda en sökning åt fel håll.

- När man söker i trädet kan man leta efter en exakt matchning, dvs samma för båda nycklarna. Men man kan även söka bara efter en nyckel och få flera svar, ett spann.
    - Används ifall man vill tillåta sökning med flera kriterier och med ett spann. Alla bokningar på ett visst datum för minst ett pris.
- Har ingen balansering eftersom det går inte att balansera på flera nycklar. Så det finns ingen garanti att vi kommer söka i O(log n).
- Fungerar sämre för stora värden av k.

Pairing Heaps

- Heap variant som stödjer decreaseKey operationen på ett effektivt sätt.
- Heap strukturen är implementerad som ett träd.
- Varje nod refererar endast till sitt vänstra barn, sedan har dem en referens till sitt syskon nod. Det blir som en zig-zag struktur om man ritar ut noderna. Referenserna är dubbelsidiga, så man kan gå upp och ner i trädet.
- Den stödjer mergning. Vid mergning av två pairing heaps så kollar den vilken root som har högst prioritet. Sedan tar vi den andra trädet och sätter det trädet som vänstra barn till trädet med högst prioritets root. Den andra trädets gamla rot kommer då refererar som sitt syskon referens till den nya robotens gamla subträd.
- Insättning av ett element är en variant på mergning, fast man sammanfogar trädet med ett träd på en enda nod, den vi vill sätta in.
- När en decreaseKey operation sker så tar man ut elementet ur listan, uppdaterar dess prioritet och åter-mergar det med trädet igen.
- När en delete min sker så tar man bort roten. Efter det så tar man alla barnen till den nu borttagna roten och gör en merge med dem. Men, inte en vanlig merge utan man använder sig av en Two-pass Mergning.
    - Vi börjar med att skanna vänster till höger och mergar alla barn noderna i par.
    - Sedan gör vi en till runda där vi mergar höger till vänster. Fast, nu mergas alltid de två sista hela tiden så att vi i slutändan får bara ett enda träd.

AA-Tree

- Variant på Red-Black Tree.

Deterministic Skip List

- Använder inte slump för höjden av noderna.
- Bygger om listan vid insättning och borttagning så att man får en bättre balans.