Datastrukturer

Sammanfattning och Referenshäfte

Guldbrand, Eric
Johansson, Algot

Juni 2018

Sammanfattning

Algoritmiska problem underlättas ofta genom kännedom av vanliga datastrukturer, deras styrkor och svagheter. Kursen Datatstrukturer introducerar ämnet och förhoppningen är att detta dokument kan stå någorlunda självständigt och samtidigt fungera som referens för de många olika detaljer som ingår i kursmaterialet. För djupare teori hänvisas till annan litteratur. Dokumentet är baserat på det material Fredrik Lindblad redogjort för under föreläsningar 2016 och igen 2017.

Innehåll

Kapitel 1

Grundläggande begrepp

Här presenteras några grundläggande begrepp som man bör känna till och som är centrala för diskussion av olika datastrukturer.

1.1 Mängd *(Set)*

En mängd är en samling element där inga element är lika. Det kan till exempel vara en lista som bara innehåller varje element en gång.

1.2 Multimängd

En mängd som kan innehålla ett element flera gånger.

1.3 Avbildning *(Map)*

En avbildning är mängd där varje element definierar en funktion $x \to y$.

Varje element i en avbildning är alltså ett par av element. Inom datavetenskap benämns dessa som key-value par och används för att associera en nyckel med ett värde eller objekt. Nyckeln kan sedan användas som index för att hitta objektet.

1.4 Induktionsbevis

Induktionsbevis kan utföras enligt följande procedur:

1. Visa att tesen gäller för basfallet $n = n_0$.
2. Antag att tesen gäller för $n = x$. Med detta antagande, visa att tesen även gäller för $n = x + 1$.
3. Om detta gäller är tesen bevisad enligt induktionssatsen.

Exempel: *Bevisa att*

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

genom att göra på följande sätt:

1. Sätt n till ett.
2. Båda sidorna blir ett så basfallet stämmer.
3. Utgå ifrån att $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ stämmer.
4. Visa att $\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}$ då också stämmer.
5. $\sum_{k=1}^{n+1} k = (n+1) + \sum_{k=1}^{n} k = (n+1) + \frac{n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$

1.5 Rekursion

En metod som anropar sig själv är rekursiv.

Om en rekursiv metod bara gör ett anrop till sig själv kan den antagligen lätt ersättas av en iterativ metod, det vill säga en loop.

Exempel: *Rekursiv funktion i Java som beräknar fakulteten av ett tal.*

```
public static int factorial(int n){
    if (n == 0)
        return 1;
    else
        return(n * factorial(n-1));
}
```

1.6 Tidskomplexitet

Tidskomplexiteten (eller den asymptotiska tillväxttakten) $T(n)$ beskriver hur mycket tid det tar att köra en algoritm i relation till indatans storlek n.

Exempel:

Att hitta det första elementet i en lista tar alltid lika lång tid oberoende på hur stor listan är. Vi säger att denna operation tar konstant tid.

Om vi istället vill skapa en kopia av listan måste vi kopiera alla element ett i taget. Tiden för denna operation beror då linjärt på arrayens storlek (dvs. det kommer ta dubbelt så lång tid att kopiera en dubbelt så stor lista).

Det finns några olika notationer för tidskomplexitet. Dessa är:

- **Ordo**, $T_{max}(n) = \mathcal{O}(f(n))$. Övre tidsgräns, värstafallskomplexiteten.
- **Omega**, $T_{min}(n) = \Omega(f(n))$. Undre tidsgräns, bästafallskomplexiteten.

– **Theta**, $T(n) = \Theta(f(n))$. Precis beskrivning av tidskomplexiteten eller normalfallskomplexiteten.

Notera att $T(n) = \Theta(f(n))$ i vanliga fall endast gäller om $\mathcal{O}(f(n)) = \Omega(f(n))$. Vid sorteringsalgoritmer och kortaste-vägen algoritmer används $\Theta(f(n))$ istället som normalfallskomplexiteten. Normalfallskomplexiteten är den uppmätta tidskomplexiteten vid slumpvis vald indata.

Den konstanta funktionen från föregående exempel kan då beskrivas som $\Theta(1)$ och den linjära funktionen som $\Theta(n)$.

När man räknar på tidskomplexiteten spelar koefficienterna ingen roll och endast den snabbast växande termen är betydelsefull. Det vill säga att om en algoritm är dubbelt så snabb som en annan har de ändå samma tidskomplexitet. Exempelvis gäller det alltså att

$$\mathcal{O}(a + bn + cn^2) = \mathcal{O}(n^2),$$

där a, b och c är konstanter.

1.6.1 Komplexitetsklasser

Algoritmers tidskomplexitet kan, som vi har sett, växa som polynom. Men de kan även växa på andra sätt. Den hastighet de växer med kallas komplexitetsklass.

Några av de komplexitetsklasser som finns är

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^k) \subset \mathcal{O}(l^n) \subset \mathcal{O}(n!)$$

där $k > 1$ och $l > 1$.

1.6.2 Amorterad tidskomplexitet

Ibland ger en tidskomplexitetsanalys som bara kollar på de värsta fallen en alltför pessimistisk uppskattning av den faktiska komplexiteten. Då kan en bättre uppskattning ges av den amorterade tidskomplexiteten $\mathcal{O}_{am}$.

Exempel:

> *En loop gör $n - 1$ billiga $\mathcal{O}(1)$ anrop och 1 dyrt anrop med komplexiteten ($\mathcal{O}(n)$). Om man då bara tittade på "värstafallet" med n stycken $\mathcal{O}(n)$ anrop är det tydligt att man skulle få en för pessimistisk uppskattning.*
>
> Den amorterade tidskomplexiteten kan bland annat beräknas med hjälp av **bokföringsmetoden**. Vi låtsas då att varje konstant operation kostar en konstant faktor mer än vad de gör (exempelvis 3 per operation istället för 1). Vid varje operationsanrop ger vi oss själva kredit för det belopp som ej egentligen gått åt (2 i detta fall). Vid varje dyr operation drar vi bort kredit från vårt konto motsvarande kostnaden för den operationen. Vi får i detta fall
>
> $$(n - 1) \cdot 2 - n = n - 2$$

> krediter kvar på vårt konto efter att kostnaden n dragits bort. Om det finns en konstant faktor så att vi alltid har mer än 0 krediter på vårt konto så kan vi säga att $\mathcal{O}_{am}(1)$ gäller.

Notera att den amorterade tidskomplexiteten inte är samma sak som normalfallskomplexiteten Θ eftersom att Θ bestäms med hjälp slumpvis valda element.

1.6.3 Tidsanalys

?? Att undersöka en kod eller algoritm och beräkna dess tidskomplexitet kallas tidsanalys.

Exempel: *Tidsanalys av loop med funktionsanrop*

> *En funktion f med tidskomplexiteten $\Theta(\log n)$ och en funktion g med tidskomplexiteten $\Theta(1)$ körs efter varandra i en loop som loopar n gånger.*
>
> Eftersom funktionerna körs efter varandra kan vi addera tidskomplexiteterna för f och g vilket ger oss $\Theta(\log n + 1)$. Eftersom de körs n gånger behöver vi sedan multiplicera allt med n och vi får $\Theta(n(\log n + 1)) = \Theta(n \log n)$.

Notera att om en funktions utdata är indata till en annan funktion (som för $f(g(x))$) så adderar man fortfarande tidskomplexiteterna.

1.7 Kostnadsmodeller

En tidsanalys kräver vissa antaganden om hur lång tid varje operation tar. Dessa antaganden sammanfattas av en kostnadsmodell. Två kostnadsmodeller som ofta används är den uniforma och den logaritmiska.

Exempel: *Motivering för kostnadsmodeller*

> Om all data är 32-bitars siffror är det rimligt att addition kan antas vara en konstant operation. Om data däremot kan bestå av godtyckligt stora tal är dock detta inte längre ett rimligt antagande.

1.7.1 Uniform modell

Tidsåtgången per operation antas vara konstant, även för oändligt stora data.

Tillgängligt minne antas vara oändligt.

1.7.2 Logaritmisk modell

Tidsåtgången per operation antas vara linjär i antalet bitar som krävs för att representera varje element.

Tillgängligt minne antas vara oändligt.

1.8 Java Collections Framework *(JCF)*

Java Collections Framework består av en mängd gränssnitt och klasser som beskriver och implementerar datastrukturer för samlingar av olika slag. Vanliga exempel är gränssnitten `List` och `Queue`.

1.9 Generiska typer (i Java)

Generiska typer underlättar skapandet av klasser, metoder och gränssnitt som fungerar för många olika typer av objekt. Speciellt så blir typfel lättare att upptäcka redan vid kompilering. Här följer några exempel.

1.9.1 Klass

En klass som beskriver ett multiset där elementen kan vara av vilken given typ som helst kan specificeras på följande sätt.

Exempel:

```
class MultiSet<E> {
    E[] array;
    void add (E x) {...}
}
```

Här säger <E> att klassen kommer att innehålla någon generisk typ E som måste definieras när ett objekt av klassen skapas. Notera att alla element i samma MultiSet måste vara av samma typ, men olika instanser av MultiSet kan innehålla objekt av olika typ.

1.9.2 Metod

En metod som verkar på en array där objekten har exakt en men godtycklig typ kan se ut på följande sätt:

Exempel:

```
public static <E> void reverse(E[] arr) {
    E tmp;
    for (int i = 0; i < arr.length / 2; i++) {
        tmp = arr[i];
        arr[i] = arr[arr.length-1-i];
        arr[arr.length-1-i] = tmp;
    }
}
```

Om vi istället har en generisk funktion findMin som hittar det minsta elementet i en generisk array kommer E behöva vara jämförbart med andra E. Då måste E implementera Comparable. Vilket ser ut såhär:

Exempel:

```
public static <E extends Comparable<? super E>>
E findMin(E[] arr) {
    if (arr.length == 0) return null;
    E min = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i].compareTo(min) < 0) min = arr[i];
    }
    return min;
}
```

Kapitel 2

Testning

För att effektivt testa och specificera korrekthet hos kod bör man känna till följande begrepp.

2.1 Invariant

En egenskap eller förutsättning som alltid måste gälla.

Exempel:

```
//Invariant: List sorted
public boolean find(A element, List<A> list)
```

Kommentaren beskriver att listan måste vara sorterad. Annars är det inte säkert att metoden fungerar.

Invarianter får brytas inuti metoder sålänge som de återupprättats innan metoden slutförs.

2.2 Precondition

Krav som förväntas vara uppfyllt då metod anropas.

Exempel:

> Stack får inte vara tom då `pop()` anropas.

2.3 Postcondition

Egenskap som förväntas vara uppfyllt då metod avslutas.

Exempel:

> Stack får inte vara tom då `push(x)` anropats.

2.4 Assertion

Assertion är att testa så att invarianter, preconditions och postconditions håller.

I Java kan man göra assertions med nyckelordet `assert`.

Exempel: *Listan args måste innehålla exakt 2 element.*

```
assert args.length == 2
```

Notera att `assert` i Java bara används i debugläge för att testa korrekthet. För att göra assertion i funktioner under riktigt körning (t.ex. för att kontrollera indata från användaren) kan man istället använda if-satser som kastar exceptions om en condition bryts.

För att köra Java-applikationer i ett sådant läge att `assert` gör något måste Java anropas med flaggan `-ea`.

Exempel:

```
java -ea programnamn arg1 arg2
```

2.5 Korrekthet

För att förvissa sig om att program fungerar korrekt kan man använda sig av:

Bevis: Kan ta lång tid och vara svårt, men ger försäkring om att inga fel finns.

Testning: Ofta mycket snabbare, men ger inga garantier på korrekthet.

Kapitel 3

Datatyper

En datatyp är ett visst sätt att strukturera data på. Olika datatyper har olika användningsområden, stödjer olika operationer och har olika tidskomplexitet för de operationer som stöds. Detta kapitel beskriver ett antal viktiga datatyper och några av deras operationer. En sammanfattning av operationernas tidskomplexiteter finns i tabell A.3.

3.1 Abstrakta datatyper *(ADT:s)*

Logiska datatyper med vissa krav på metoder och tidskomplexitet.

Representeras i Java med hjälp av gränssnitt (interface) som sedan kan implementeras konkret av klasser.

Några vanliga abstrakta datatyper och metoder som dessa implementerar är:

- **Stack**: `push(x)`, `pop()`, ...
- **Kö**: `enqueue(x)`, `dequeue()`, ...
- **Lista**: `add(x)`, `add(x, i)`, `remove(i)`, `get(i)`, ...
- **Iterator**: `hasNext()`, `next()`, `nextInt()`, ...
- **Träd**: `add(x)`, `size()`, ...

3.2 Dynamisk array

En förallokerad array med en extra variabel som minns arrayens nuvarande logiska storlek.

När arrayen blir full och man försöker lägga till fler element allokeras en ny, dubbelt så stor array. Eftersom man då måste kopiera över alla element från den gamla arrayen har detta steget en tidskomplexitet på $\mathcal{O}(n)$.

Exempel: *Beräkning av amorterad tidskomplexitet.*

> Vi vill beräkna den amorterade tidskomplexiteten $\mathcal{O}_{am}$ för insättning av n element i en tom lista enligt bokföringsmetoden. Vår lista är en dynamisk array med förallokerad storlek 1. Varje gång den blir full multipliceras storleken med en faktor

2.

1. Antag att det tar 3 gånger längre tid att lägga till ett element i listan än vad det faktiskt gör.
2. Det första elementet läggs till i arrayen. Vi får då den antagna kostnaden minus den faktiska kostnaden som krediter på vårt konto. I detta fall $3-1=2$ krediter.
3. Innan nästa element läggs till måste arrayen dubbleras i storlek. Då måste alla element i arrayen kopieras. Arrayen innehåller ännu bara 1 element och eftersom att kopiering är linjär med antalet element kostar kopieringen bara 1 kredit. Vårt konto innehåller nu $2-1=1$ kredit.
4. Ytterligare 1 element läggs till i arrayen och vi får därmed 2 till krediter till vårt konto. $(1+2=3)$
5. Arrayen måste kopieras igen och detta kostar nu 2 krediter eftersom att den innehåller 2 element. Detta har vi råd med. $(3-2=1$
6. Fyll igen och dubblera storleken ger oss $1+4-4=1$. En gång till och vi har $1+8-8=1$. Varje gång vi fyller arrayen kommer vi alltså ha råd att göra en kopiering om vi antagit att insättning av element tar tre gånger så lång tid som det egentligen gör.
7. Tidskomplexiteten för att lägga till n element till en tom lista är alltså $\mathcal{O}(3n)=\mathcal{O}(n)$ och därmed blir den amorterade värstafallskomplexiteten för att lägga till *ett* element $\mathcal{O}_{am}(1)$.

3.3 Länkad lista

Har pekare till första och eventuellt sista element.

Varje element representeras av en intern nodklass med pekare till efterföljande element.

Kan vara enkel- eller dubbellänkad. (Dubbellänkad har pekare även till föregående element.)

Första och sista element kan vara vaktposter (sentinels). Dessa kan bidra till att färre specialfall behöver skrivas för första och sista element.

3.4 Skipplista

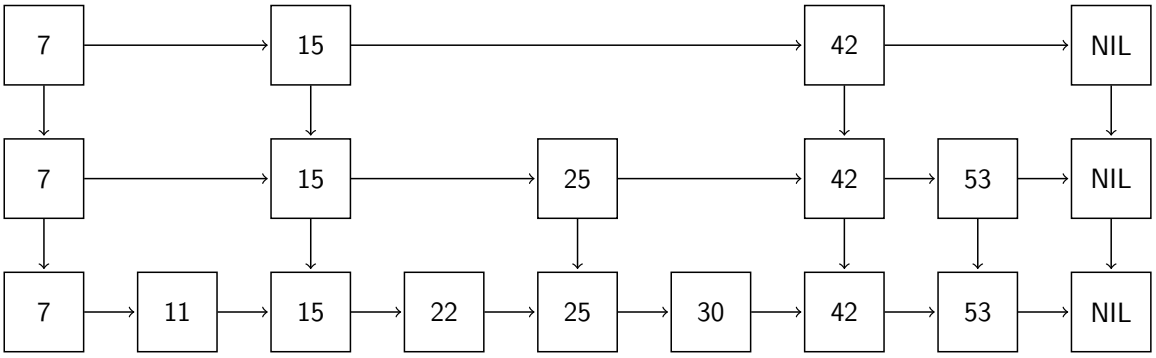
En skipplista är en länkad lista där varje element minst har en länk till nästa element men också kan ha länkar till andra element längre fram. Detta brukar representeras som att varje element har en viss höjd och alla element på samma höjd länkar till ett annat element på den höjden. Figur 3.1 visar en skipplista där horisontella pilar motsvarar länkar mellan element.

Ett elements höjd bestäms slumpmässigt med en viss sannolikhet när de läggs till i listan genom att det alltid är $(1-p)$ sannolikhet för ett element att nå till ytterligare en nivå. Ofta är p likamed $1/4$ eller $1/2$.

Först och sist i listan måste det finnas element med maximal höjd. Dessa kan vara riktiga

element i listan eller vaktpost-element med null-värde som bara visar var listan startar eller slutar.

Skipplistor tillåter binär sökning i länkade listor som annars bara kan söka i $\mathcal{O}(n)$.



Figur 3.1: *Skipplista med vaktpost på slutet. De horisontella pilarna är länkar mellan element, de vertikala visar bara nivåskillnaden. För att hitta ett element x, börja högst upp på det minsta elementet i listan. Om nästa element på samma rad är mindre än eller likamed x, följ pilen till höger, om inte, följ pilen neråt. Repetera tills elementet du är på är x eller större än x.*

3.5 Träd

Träd finns ej som gränssnitt i Java Collections Framework men är en mycket viktig typ av datastruktur. För mer om träd, se sektion 5.

3.6 Cirkulär array

En array som liksom en vanlig lista försöker lägga till nya element på slutet.

Om arrayen är full och element läggs till på slutet hamnar det i början på arrayen om det finns plats där. Om inte fördubblas arrayens storlek och alla element hamnar i början av den nya arrayen.

Har pekare på första och sista element.

3.7 Prioritetskö

Kö där varje element har viss prioritet. Ofta pratar man om prioritetsköer som att det element som ska vara längst fram i kön är det ”minsta”.

Metoder: `insert(x)`, `find-min()`, `delete-min()`, `merge(xs)`, ...

Implementation av prioritetskö med en vanlig lista ger funktionerna tidskomplexiteten $\mathcal{O}(n)$. Implementation med binär heap ger däremot $\mathcal{O}(\log n)$.

3.8 Hashtabell

En hashtabell kan antingen implementeras som en avbildning (Java `map`) eller mängd (Java `set`).

Data lagras i array.

Både nyckel och värde kan vara vilken typ som helst.

Varje tänkbar nyckel kan inte ha en reserverad plats i arrayen. Därför bestämmer en hashfunktion vilken plats ett element ska läggas på genom att omvandla nyckeln till ett index som finns i arrayen. Se avsnitt 3.9.

Om varje plats i arrayen innehåller en lista som så att många element kan ligga på samma index kallas detta för "chaining".

Ett alternativ till chaining är öppen adressering. Om en plats i listan redan är upptagen då ett element ska läggas in läggs detta då istället på nästa lediga plats.

3.9 Hashfunktion

En hashfunktion tillhör en hashtabell och omvandlar objekt till index i hashtabellens interna lista. Detta för att det i en hashtabell ska vara möjligt att använda vilken typ av objekt som helst som nyckel.

I Javas `Object()` finns en funktion `hashCode()` som blir override:ad av typer som ska kunna hashas. Denna funktion genererar ett numeriskt värde så att två objekt som `equals(..)` tycker är lika alltid har samma hashkod samtidigt som alla objekt som är olika har olika hashkod.

Hashtabellens hashfunktion använder sig sedan av hashkoden vid insättning/lookup/etc. av element för att generera ett index. Detta index genereras genom att beräkna modulus av hashkoden och den underliggande arrayens längd.

På grund av användningen av modulus kan kollisioner uppstå (flera element hamnar på samma plats i arrayen). Därför krävs ibland re-hashing. Då skapas en ny dubbelt så stor array och alla index beräknas på nytt, vilket sprider ut dem över hela den nya arrayen.

Kapitel 4

Grafer

4.1 Terminologi

En graf kan beskrivas av $G = (V, E)$ där G är grafen, V är mängden av noder (vertices) och E är mängden av kanter (edges).

Mängden kanter för en graf kan beskrivas av $E = (v, w)$ där alla $v, w \in V$. Det vill säga, alla kanter går mellan par av grafens noder.

I en riktad graf är $(v, w) \neq (w, v)$. I en oriktad graf gäller istället $(v, w) = (w, v)$.

4.1.1 Stig

En stig eller väg beskrivs av noderna $w_1, w_2, ..., w_n$ där $(w_i, w_{i+1}) \in E$, dvs. den beskrivs av en följd noder där kanten från en nod till nästa alltid är en av grafens kanter.

4.1.2 Loop

En kant (w, w) vars båda ändnoder alltså är densamma.

4.1.3 Enkel stig

En stig där samma nod bara finns med högst en gång med undantaget att samma nod kan vara både först och sist.

4.1.4 Cykel / Acyklisk

En cykel är en stig där den första noden också är den sista. En acyklisk graf är en graf som inte har några cykler.

4.1.5 Viktad graf

I en viktad graf representeras varje kant av (v, w, x) där x är en vikt.

4.1.6 Direkt föregångare/efterföljare

En direkt föregångare till noden w är en nod v så att kanten (v, w) är en kant i grafen. Det vill säga $\{v|(v, w) \in E\}$. En direkt efterföljare till noden v är en nod w så att kanten (v, w) är en kant i grafen. Det vill säga $\{w|(v, w) \in E\}$.

4.1.7 Ingrad/Utgrad

En nods ingrad/utgrad är likamed antalet direkta efterföljare/föregångare som noden har.

4.2 Multigraf

En graf som får ha parallella kanter. Två kanter är parallella om de har samma slutnoder.

4.3 Grannmatris

En matris som beskriver en graf genom att varje element e_{ij} i grafen har värdet 1 om det finns en kant mellan nod i och nod j. Om kant ej finns har elementet värdet 0.

Innehåller $|V|^2$ element.

Att gå igenom en nods direkta efterföljare är $\Theta(|V|)$.

Att gå igenom alla noders direkta efterföljare är $\Theta(|V|^2)$.

Att avgöra om kant finns från v till w är $\Theta(1)$.

4.4 Grannlista

En lista av listor som beskriver en graf. Den yttre listan har lika många element som antal noder i grafen. Varje inre lista e_i beskriver nod i genom att innehålla referenser eller index till alla nodens grannar.

Att gå igenom en nods direkta efterföljare är $\Theta(|V|)$.

Att gå igenom alla noders direkta efterföljare är $\Theta(|V| + |E|)$.

Att avgöra om kant finns från v till w är $\Theta(|V|)$.

4.5 Minsta uppspännande träd *(MST)*

Ett fritt träd (ett träd utan bestämd rot) som är en delmängd av någon graf. Trädet inkluderar alla noder hos sin graf, med bara vissa kanter.

Summan av alla valda kanters vikter är minimal.

En graf kan ha flera minsta uppspännande träd.

4.6 Total Ordning och Topologisk Sortering

En topologisk sortering bestämmer en total ordning för en graf G.

I en total ordning gäller att om det finns en väg från v till w så är $v < w$.

Det kan finnas flera totala ordningar för en graf.

En total ordning kan endast existera om grafen ej innehåller några cykler.

Algoritm, med kö, för att hitta en topologisk ordning:

1. Lägg alla noder med ingrad 0 i en kö (eller stack).
2. Ta bort en nod från kön. Ta bort denna nod och alla dess (utgående) kanter från grafen.
3. Lägg till alla direkta efterföljare till noden som nu har ingrad 0 i kön.
4. Upprepa steg 2 och 3 tills kön är tom.

Exempel på total ordning: Kunskapskrav för kurser. För kursen *Avancerad matematik* krävs att man läst kursen *Inledande matematik*. Då är den inledande kursen mindre än den avancerade .

4.7 Starkt Sammanhängande Komponent *(SCC)*

En starkt sammanhängande graf är en graf där det finns minst en väg mellan varje par av noder. (Det behöver dock ej finnas en kant mellan varje par av noder.) Riktade vägar räknas på samma sätt som oriktade, alltså behöver det inte finnas riktade vägar åt båda håll.

En starkt sammanhängande komponent är ett antal noder och kanter som är starkt sammanhängande och är del av en större graf.

Om varje SCC i en graf ersätts med en enda nod blir grafen en acyklisk (multi-)graf.

Starkt sammanhängande komponenter kan hittas mha DFS.

Kapitel 5

Träd

Ett träd är en graf utan cykler. Detta kapitel beskriver ett antal begrepp och trädoperationer samt listar några olika typer av träd och heaper.

5.1 Begrepp och operationer

För träd gäller följande terminologi:

- **Rotnod**: Nod högst upp i trädet. Har inga föräldrar.
- **Förälder**: Nod över en annan nod.
- **Barn**: Noder under annan nod.
- **Syskon**: Barn till samma förälder.
- **Löv**: Noder utan barn.
- **Delträd**: En nod i ett träd och alla dess barn, barnbarn osv.

En mängd träd kallas en **skog**.

Längden av en väg mellan noder är antalet passerade kanter emellan dem.

En nods **djup** är längden av vägen från den noden till roten.

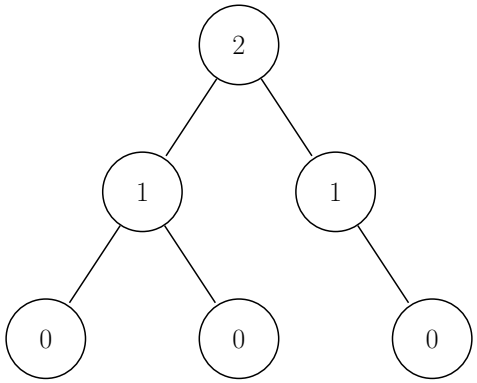
Trädets **höjd** är det maximala djupet för någon nod.

Träd kan implementeras med hjälp av:

- Pekare: Fungerar för alla träd. Likt implementeringen för länkad lista.
- Array: Fungerar enbart för vissa träd, men är då särskilt effektivt. Det går att hitta varje nods förälders/barns index m.h.a. enkel matematisk formel.

5.1.1 Null Path Length

Null Path Length (NPL) för en nod är längden av den kortaste vägen till en nod som inte har barn. Figur 5.1 visar en graf där varje nod är märkt med sitt NPL värde.



Figur 5.1: *Ett träd med noder, alla märkta med sitt Null Path Length värde.*

5.1.2 Trädgenomlöpning

Ett träds noder kan genomlöpas på flera olika sätt.

- **Pre-order**: Nod, vänster delträd, höger delträd.
- **In-order**: Vänster delträd, nod, höger delträd.
- **Post-order**: Vänster delträd, höger delträd, nod.
- **Bredden-först**: Nod, nodens barn, nodens barnbarn etc. (En nivå i taget.)

Notera att när man nämner ett delträd i listan ovan betyder det att man applicerar samma genomlöpning rekursivt på det delträdet.

5.1.3 Trädrotationer

För att träd ska kunna upprätthålla sin balans krävs det ibland att man använder trädrotationer.

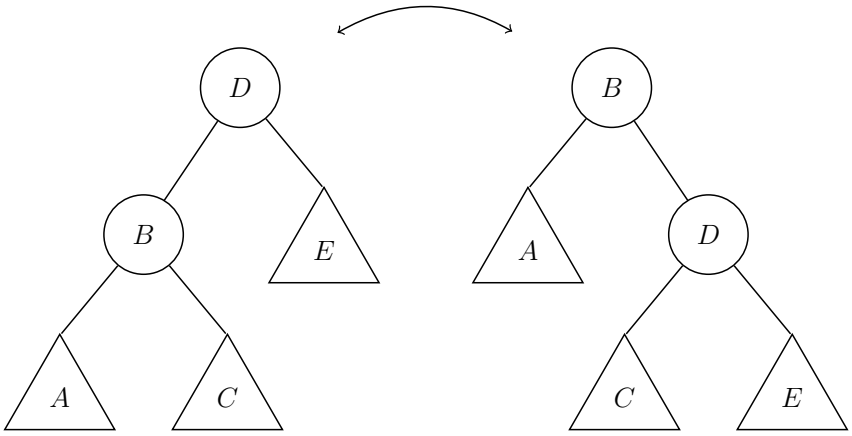
En trädrotation ändrar höjdskillnaden mellan delträd men bevarar ordningen. I figur 5.2 ser vi en **enkelrotation**. Efter en högerrotation har B tagit platsen som rotnod och D blivit dess högra barn. Delträd C har fått en ny förälder vilket är möjligt då alla dess noder är större än nod B och mindre än nod D.

Om höjden på A vore 1 nivå högre än C och E så hade höjdskillnaden mellan rotnodens barn i det vänstra delträdet varit 2. Efter rotationen skulle höjdskillnaden istället vara 0 och trädet alltså ha balanserats.

Notera att man inte måste rotera ett helt träd. Om höjdskillnad upptäcks inom ett delträd räcker det att rotera delträdet.

Ibland räcker inte enkelrotation. Om höjden på C i figur 5.2 vore 1 nivå högre än A och E hade trädet efter rotationen fortfarande varit obalanserat. Då hade en dubbelrotation varit nödvändig.

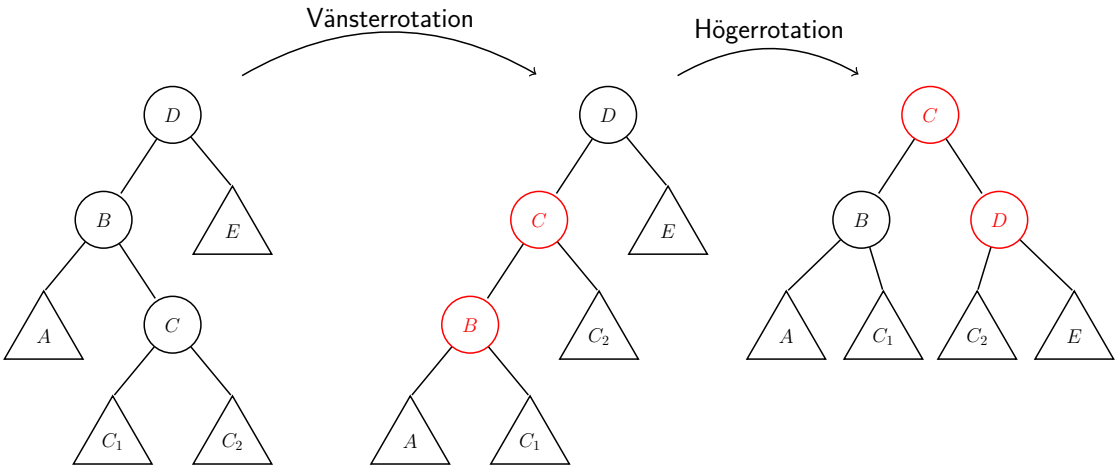
I en **dubbelrotation** roterar man först inom det delträd som är för högt. Sedan roterar man vid noden där höjdskillnaden upptäckts. Är det i vänster delträd roterar man först åt vänster sedan åt höger. I höger delträd gör man tvärtom.



Figur 5.2: *Exempel på enkel trädrotation. Efter rotation åt höger har B tagit platsen som rotnod och D blivit dess högra barn. Delträd C har bytt förälder och blivit D's barn. Noder representeras av cirklar och delträd av trianglar. Bokstavsordningen för varje komponents namn motsvarar storleken på elementen, dvs alla noder $a \in A < B < c \in C$ osv. Notera att ordningen från vänster till höger behålls genom rotationen.*

I fallet då delträd C i figur 5.2 var för högt skulle man behöva rotera det västra delträdet åt vänster och sedan hela trädet åt höger som i figur 5.3. Vilket delträd till C som var för högt spelar ingen roll.

Notera att figur 5.3 vänstra träd är detsamma som det vänstra i figur 5.2 men delträd C har expanderats för att visa höjdskillnaden.



Figur 5.3: *Exempel på dubbel trädrotation. Det vänstra trädet är detsamma som i figur 5.2 men med delträd C expanderat för att visa höjdskillnaden. I första steget roteras det delträd som har B som rotnod åt vänster, i andra steget roteras hela trädet åt höger.*

5.2 Typer av träd

Det finns många olika typer av träd. Vissa egenskaper (binär, ordnad etc.) återkommer ofta, andra träd är mer specifika. Denna sektion listar ett antal av dessa.

5.2.1 Ordnade träd

Ett ordnat träd är ett träd där, för varje nod i trädet, alla element i dess vänstra delträd är mindre än noden. Noden är i sin tur mindre än alla element i höger delträd. Alltså att:

- Det gäller att $v < r < h$ för alla noder $v \in V$ och för alla noder $h \in H$ där V och H är vänster och höger delträd och r är rotnoden.
- Alla delträd är ordnade.

Ett träd som är ordnat sägs uppfylla **sökträdsegenskapen**.

5.2.2 Kompletta träd

Ett komplett träd är ett träd med så låg höjd som möjligt. Detta innebär att endast sista och näst sista raden innehåller löv. Den sista raden fylls från vänster. Det innebär att det inte får finnas några hål mellan två noder på den sista raden.

5.2.3 Binära träd

Ett träd där ingen nod har fler än 2 barn.

En nod med binärt delträd till vänster är inte nödvändigtvis detsamma som en likadan nod med samma delträd till höger.

5.2.4 Binära sökträd

Ett binärt sökträd är ett ordnat binärt träd.

Lägg till en nod n i ett binärt sökträd genom att jämföra n med rotnoden. Om n är störst, lägg till n i höger delträd, annars i vänster. När man lägger till noder i delträd använder man samma algoritm rekursivt.

Ta bort en nod utan barn genom att bara plocka bort den. Ta bort en nod med ett barn genom att ersätta den med sitt barn. Ta bort en nod med två barn genom att ersätta den med sitt högra delträds minsta element eller sitt vänstra delträds största element.

Sökträd kan ha samma funktioner som både mängd och prioritetskö. De kan även implementera in-order operator (att gå igenom alla element i ordning). In-order genomlöpning returnerar noder i växande storlek.

5.2.5 Balanserade träd

Träd där höjdskillnaden mellan delträd (för varje nod) alltid är mindre än något visst värde.

Trädet har höjden $\Theta(\log n)$. Detta är bra för sökträd då stora höjdskillnader kan resultera i längre söktid. (Ett sökträd med n noder och höjd n är bara en länkad lista!)

Självbalanserade träd är träd vars operationer alltid lämnar trädet i ett balanserat läge.

5.2.6 Röd-Svarta Träd

Röd-svarta träd är självbalanserande binära sökträd där varje nod antingen är svart eller röd.

Rotnoden är svart. Röda noder har svarta barn. Alla enkla vägar från roten till noder med max ett barn innehåller lika många svarta noder.

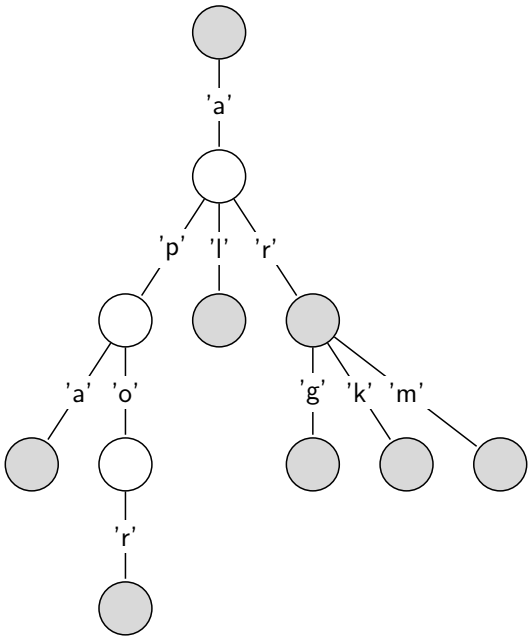
Röd-svarta träd används i Java's `TreeSet` och `TreeMap`.

5.2.7 Prefixträd

Ett prefixträd är ett träd för att implementera mängder eller avbildningar där elementen/nycklarna ofta är strängar.

För ett exempel, se figur 5.4 där ett prefixträd representerar ett mängd ord. Gråa noder markerar slutet på ord. Notera att alla löv måste vara gråa, men alla gråa noder måste inte vara löv.

Att undersöka om ett prefixträd innehåller en viss sträng har en tidskomplexitet som endast beror endast på hur lång strängen är, antal noder i trädet spelar ingen roll. Tidskomplexiteten blir $\mathcal{O}(l)$ där l är längden på ordet.



Figur 5.4: *Prefixträd för mängden { "", "apa", "apor", "al", "ar", "arg", "ark", "arm" }*

5.2.8 Splay-träd

Självbalanserande binärt sökträd. Vid åtkomst splayas noder upp till roten vilket gör nyligen använda noder snabbare att komma åt igen. Detta ger i praktiken ofta splay-träd en fördel över andra träd.

För att splaya en nod utförs rotationer på delträdet tills noden är rot. Det finns tre olika rotationer.

– **Zig**: En enkelrotation. Utförs då nodens förälder är rot.
– **Zig-zig**: Två enkelrotationer åt samma håll. Utförs om nod och förälder är barn på samma sida (dvs. båda är vänsterbarn eller båda är högerbarn).
– **Zig-zag**: Två enkelrotationer åt olika håll. Utförs om nod och förälder är barn på olika sidor.

5.2.9 AVL-träd

Själv-balanserande binärt sökträd. Höjdskillnaden får max vara 1. Använder sig av trädrotationer för att upprätthålla balansen.

5.3 Heapar

En heap är ett träd som uppfyller **heapordningsegenskapen**: Varje nod är mindre än eller lika med sina barn. (Detta för en min-heap där minsta elementet alltså är roten. Det motvända gäller för max-heap.)

5.3.1 Binär heap

En binär heap är ett komplett binärt träd.

En binär heap kan användas för att skapa en prioritetskö. Prioritetsköns operationer kan implementeras på följande sätt:

– `Insert(x)`: Lägg till nytt element sist. Bubbla upp.
– `Delete-min()`: Ta bort roten. Ersätt detta med sista elementet. Bubbla ner detta.
– `Bubble(x)`: (Up/Down). Jämför elementet med förälder/barn. Byt plats om elementen ej uppfyller heapordningsegenskapen. Fortsätt bubbla elementet tills inget byte sker. Vid nedåtbubbling, om byte ska ske, byt plats med minsta barnet.
– `Build-heap(..)`: Stoppa in alla element i godtycklig ordning. Bubbla ner alla icke-löv med början på nedre högra hörnet. Tidskomplexitet $\Theta(n)$.

5.3.2 Leftistheap

En leftistheap (eller leftistträd) är en sorts binärheap som inte behöver vara komplett.

Leftistheapen uppfyller förutom heapordningsegenskapen även **leftistträdsegenskapen**: För varje nod gäller att $NPL(l) \geq NPL(r)$ där l är vänster barn och r är höger barn.

De flesta funktioner hos leftistheapen är baserade på operationen `merge`.

`Merge(`h_1`,` h_2`)`: Slår ihop två träd. Om leftistegenskapen är uppfylld är tidskomplexiteten $\mathcal{O}(\log n)$ där n är antalet element i det största av de två träden h_1 och h_2. En rekursiv variant av operationen kan implementeras på följande sätt:

1. Låt a vara h_1:s vänstra delträd och b vara h_1:s högra delträd.

2. Om h_1 är tom, returnera h_2 och vice versa.
3. Om $\text{rot}_{h_1} > \text{rot}_{h_2}$, byt namn på träden (så att h_2 alltid är trädet med störst rot).
4. Låt $d = merge(b, h_2)$.
5. Låt e vara trädet med roten rot_{h_1}, vänster delträd a och höger delträd d.
6. Om a är tomt, byt plats på a och d i e. I detta fall är nu e det mergade trädet.
7. Annars, om $NPL(a) < NPL(d)$ byt plats på a och d. Nu är e det mergade trädet.

`Insert(x)`: Låt elementen `x` som ska sättas in vara rot i ett annars tomt träd. Gör merge mellan detta träd och trädet som ska få ett element insatt.

`Delete-min()`: Ta bort rotnoden. Merga delträden.

Kapitel 6

Algoritmer

Det finns många olika algoritmer för att operera på listor, grafer och andra datastrukturer. I detta kapitel listas ett antal olika algoritmer för sortering och sökning av olika slag. I bilaga A sammanfattas dessa algoritmers tidskomplexiteter i tabellform.

6.1 Sorteringsalgoritmer

I denna sektion listas ett antal sorteringsalgoritmer.

6.1.1 Urvalssortering

Följande metod används i urvalssortering:

1. Hitta det minsta elementet i listan.
2. Placera det först.
3. Det första elementet är nu sorterat.
4. Hitta det minsta elementet i den osorterade delen av listan.
5. Placera det som andra element i listan.
6. De första två elementen är nu sorterade.
7. Fortsätt med samma metod till hela listan är sorterad.

Tidskomplexiteterna för urvalssortering är $\Omega(n^2)$, $\Theta(n^2)$, $\mathcal{O}(n^2)$.

6.1.2 Insättningssortering

Följande metod används i insättningssortering:

1. Jämför de två första elementen i listan.
2. Byt plats på dem om det andra elementet är mindre än det första.

3. De två första elementen är nu sorterade.
4. I varje följande iteration välj det första elementet i den osorterade delen av listan.
5. Byt plats på det valda och det tidigare elementet så länge som det tidigare är större än det valda.

Tidskomplexiteterna för insättningssortering är $\Omega(n)$, $\Theta(n^2)$, $\mathcal{O}(n^2)$.

6.1.3 Bucket sort

I bucket sort används listor som hinkar eftersom de har obegränsad storlek.

För att förstå bucket sort kollar vi först på specialfallet då vi har 10 tal som ska sorteras. Alla ska även ha ungefär samma tiopotens med låt säga n siffror, annars blir hinkarna obalanserade. Inget tal får ha mer än n siffror.

1. Skapa en ny array av tomma listor. Den ska ha lika många element som det finns element att sortera, 10 i detta fall.
2. Ta den n:te siffran i det första talet. Kalla den s. (Om talet har mindre än n siffror, låt $s = 0$.)
3. Placera talet i hinken på plats s i arrayen.
4. Upprepa de tidigare två stegen till alla tal har hamnat i en hink.
5. Nu är alla siffrorna ordnade i arrayen efter deras mest signifikanta tal.
6. Sortera varje lista i arrayen för sig med någon annan sorteringsalgoritm.
7. Sätt ihop listorna i arrayen.

I mer generella fall måste man använda någon funktion som omvandlar ett tal till ett index av storlek 0 till N-1 där N är antalet listor. Funktionen måste också se till att ett större tal inte får ett mindre index än ett annat.

Tidskomplexiteterna för bucket sort är $\Omega(n+k)$, $\Theta(n+k)$, $\mathcal{O}(n^2)$. Där n är antalet element och k är antalet hinkar.

6.1.4 Merge sort

Merge sort är en sorteringsalgoritm av typen söndra och härska. Den utnyttjar det faktum att det är billigt att sammanfoga två redan sorterade listor till en ny sorterad lista.

För att sammanfoga två redan sorterade listor använder man följande metod:

1. Skapa en ny lista med plats för båda de gamla listorna.
2. Skapa två pekare till de första(minsta) elementen i de gamla listorna.
3. Välj ut det minsta elementet bland de som pekarna pekar på.
4. Lägg till elementet på nästa tomma plats i den nya listan.

5. Flytta pekaren som pekade på det minsta element till nästa element i listan och upprepa från 3.

För själva merge sort används sedan följande metod:

1. Skicka in den osorterade listan till sorteringsfunktionen.
2. Dela listan på hälften till två nya listor.
3. Skicka in de två halva listorna rekursivt till funktionen. Detta sorterar halvorna.
4. När de två halva listorna är sorterade sammanfogar man bara dem till en ny sorterad lista och returnerar den.
5. I basfallet, om funktionen får in en lista med endast ett element så returnerar den bara listan.

Tidskomplexiteterna för merge sort är $\Omega(n \log n)$, $\Theta(n \log n)$, $\mathcal{O}(n \log n)$.

6.1.5 Quicksort

Quicksort är också en sorteringsalgoritm av typen söndra och härska.

Följande metod används i quicksort:

1. Skicka in den osorterade listan till sorteringsfunktionen.
2. Välj ett pivotelement ur listan.
3. Placera alla element som är mindre än pivotelementet innan det och alla element som är större än det efter det.
4. Kalla funktionen rekursivt på de båda halvorna.

Tidskomplexiteterna för Quicksort är $\Omega(n \log n)$, $\Theta(n \log n)$, $\mathcal{O}(n^2)$.

6.2 Algoritmer för grafer

Det finns en mängd saker man kan göra med grafer. Här beskrivs några av de vanligaste.

6.2.1 Bredden-först sökning

Kan användas för att hitta en specifik nod eller kortaste vägen till den.

Fungerar endast för oviktade grafer. Detta beror på att i en viktad graf kan en nod som är en direkt efterföljare ändå ligga längre bort än en nod som inte är det.

Algoritm:

1. Skapa en tom kö. Denna håller reda på vilka noder man härnäst kommer utgå ifrån.
2. Börja på någon nod v.

3. Markera alla v:s obesökta direkta efterföljare som besökta och lägg till dessa i kön.

4. Gå till noden överst i kön och låt nu v peka på denna nod. Repetera från steg 2 tills sökt nod hittats eller kön är tom.

Notera att om kortaste väg ska hittas så måste grafen antingen vara ett träd (då kan man bara gå uppåt efter att mål-noden hittats) eller så måste man för varje nod hålla koll på från vilken nod man först besökte den.

Om grafen man utgick ifrån är ett träd och noden man börjat på varit rotnoden så hade bredden-först sökningen besökt en nivå i taget.

Tidskomplexitet $\mathcal{O}(|V| + |E|)$.

6.2.2 Djupet-först sökning *(DFS)*

Kan användas för att hitta en specifik nod eller kortaste vägen till den.

Fungerar endast för oviktade grafer och liknar bredden-först algoritmen, men går istället först på djupet.

Algoritm:

1. Börja på någon nod.

2. Gå till nodens första obesökta barn och markera detta som besökt.

3. Repetera steg 2 tills du hittar en nod som inte längre har något obesökt barn.

4. Gå upp tills du når en nod med obesökta barn och repetera från steg 2.

Tidskomplexitet: $\Theta(|V| + |E|)$

Om DFS används för att hitta en uppspännande skog skapas ett träd för varje toppnivå-anrop till DFS.

6.2.3 Dijkstras algoritm

Används för att hitta minsta avstånd från en startnod till en slutnod i en graf.

Fungerar för viktade grafer utan negativa vikter.

Algoritm:

1. Lägg alla noder i en lista som relaterar dem till sitt kortaste avstånd från start. I början är detta avstånd okänt för alla.

2. Börja på någon nod v. Kalla avståndet till v för S_v och sätt det till 0 här och i listan.

3. För varje efterföljande nod i kallar vi avståndet från v till i för $S_{v \to i}$. Om kortaste avståndet till efterföljaren i listan är okänt eller större än $S_v + S_{v \to i}$, uppdatera kortaste vägen i listan till $S_v + S_{v \to i}$.

4. Gå nu den billigaste av alla kända vägar i listan till en efterföljare j.

5. Låt först $S_j = S_v + S_{v \to j}$. Noden v är nu klar så vi döper om j till v.

6. Repetera från steg 2 med de nya värdena för v och S_v. Fortsätt tills kortaste väg till önskad nod har hittats.

Dijkstras algoritm får olika tidskomplexitet beroende på vilka datastrukturer som används för att implementera algoritmen. Om algoritmen implementeras med hjälp av prioritetskö (baserad på binär eller leftistheap) kan den nå $\mathcal{O}(|V| + |E| \log |V|)$.

6.2.4 Prims algoritm

Används för att hitta minsta uppspännande träd.

Algoritm:

1. Lägg alla noder i en lista så att varje nod kan relateras till sitt kortaste avstånd till trädet. I början är detta avstånd okänt för alla.

2. Börja på en slumpvis vald nod. Detta är nu början på trädet. Lägg längden till alla grannar i listan.

3. Välj den nod med minst avstånd till trädet och lägg till den i trädet.

4. För varje ny nod, lägg till alla dess grannar vars avstånd till trädet i listan är okänt eller större än avståndet till den nya noden.

Liknar Dijkstras algoritm fast att man här bara bryr sig om $S_{v \to i}$ för att avgöra vilken nod man ska gå till i varje iteration.

Korrektheten för Prims algoritm kan visas med skärningsegenskapen. Induktionsbevis kan visa att det faktiskt är tillräckligt bra att i varje steg ta kortaste sträckan till ny nod för att skapa ett MST.

6.2.5 Kruskals algoritm

Används för att hitta en skog av minsta uppspännande träd i en osammanhängande graf. (Ett MST per sammanhängande delgraf.)

Idéen är att alltid ansluta de noder i grafen med kortast kant mellan sig utan att det skapar några cykler. Till skillnad från Prims algoritm måste inte kanterna kopplas till det redan existerande trädet utan kan skapa delträd.

Algoritm:

1. Skapa en lista A där varje nod i motsvaras av ett index. Varje element i A är en lista av alla noder som i är direkt ansluten till. Till att börja med är varje nod i bara ansluten till sig själv.

2. Skapa en tom lista B. Denna kommer innehålla de kanter som ska bli del av de minsta uppspännande träden.

3. Lägg alla grafens kanter i en lista C och sortera den så att billigast kant är först.

4. Löp igenom listan C i ordning. För varje kant, titta på de element i listan A som motsvarar kantens slutnoder. Om dessa element (som pekar på listor) pekar på samma lista, ignorera kanten och gå direkt vidare till nästa kant.

5. Lägg till den nuvarande kanten (u, v) i lista B. Välj sedan den nod i kanten vars lista i A är kortast (låt denna nod vara v). Slå ihop u och v's A listor genom att låta alla element från v:s lista peka på samma lista som u pekar på.

6. När C löpts igenom innehåller B alla kanter i skogen av minst uppspännande träd och A innehåller informationen vilket träd varje nod ingår i.

6.3 Binärsökning

Binärsökning är en metod för att söka efter element i en ordnad lista. Algoritmen är som följer.

1. Titta på elementet i mitten av listan.

2. Om elementet är större än det sökta elementet, tillämpa binärsökning rekursivt på vänstra halvan av listan, annars den högra.

3. Repetera från steg 1 till elementet är funnet i mitten av någon dellista eller listan är tom.

Bilaga A

Tidskomplexiteter för algoritmer och strukturer

A.1 Sorteringsalgoritmer

Algoritm	**Bästafall** Ω	**Normalfall** Θ	**Värstafall** $\mathcal{O}$
Urvalssortering	n^2	n^2	n^2
Insättningssortering	n	n^2	n^2
Bucket sort	$n + k$	$n + k$	n^2
Merge sort	$n \log n$	$n \log n$	$n \log n$
Quick sort	$n \log n$	$n \log n$	n^2

Tabell A.1: *Tidskomplexiteter för olika sorteringsalgoritmer.*

A.2 Algoritmer för grafer

Algoritm	**Tidskomplexitet**
Bredden först	$\mathcal{O}(\|V\| + \|E\|)$
Djupet först	$\Theta(\|V\| + \|E\|)$
Dijkstras (prioritetskö)	$\mathcal{O}(\|V\| + \|E\| \log \|V\|)$
Prims (binary heap, grannlista)	$\mathcal{O}(\|V\| + \|E\| \log \|V\|)$
Prims (grannmatris)	$\mathcal{O}(\|V\|^2)$
Kruskals	$\mathcal{O}(\|E\| \log \|V\|)$

Tabell A.2: *Tidskomplexiteter för olika kortaste-vägen algoritmer. V är mängden noder (vertices) och E är mängden kanter (edges) i grafen.*

A.3 Operationer på datastrukturer

Operation	**Dynamisk array**	**Dubbel-länkad lista**	**Skipplista**	**Binär heap**	**Leftistheap**	**Hashmap**	**AVL träd**	**Röd-svarta träd**
add(x)	$\mathcal{O}_{am}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}_{am}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}_{am}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
add(x,i)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	—	—	—	—	—
remove(x)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	—	—	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
remove(i)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	—	—	—	—	—
get(i)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	—	—	—	—	—
set(i,x)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	—	—	—	—	—
find(x)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	—	—	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
size	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
find-min	—	—	—	$\mathcal{O}(1)$	$\mathcal{O}(1)$	—	—	—
delete-min	—	—	—	$\mathcal{O}_{am}(\log n)$	$\mathcal{O}(\log n)$	—	—	—
merge	—	—	—	$\Theta(n)$	$\mathcal{O}(\log n)$	—	—	—
build-heap	—	—	—	$\Theta(n)$	$\Theta(n)$	—	—	—

Tabell A.3: *Tidskomplexiteter för olika operationer på diverse datastrukturer. Vissa operationer har andra namn än de angivna för vissa datastrukturer. Markeringen (—) betyder att operationen inte är standard för datastrukturen, däremot vore det ofta möjligt att implementera den.*

Sakregister