Algoritmer og Datastrukturer

Written in December 2024

Authors:

Rikke Aaling Boysen

Simone Ingvild Lebech

Contents

1 Bibliography

- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++

2 Rekursion

En rekursiv funktion er en funktion der enten kalder sig selv eller en del af sig selv. De er smarte fordi man slipper for at holde styr på de rekursive skridt, da det håndteres af runtime-stakken vha. activation records. Man taler om dem, som værende sjældne eksempler på, at det kan bevises matematisk at et program er fejlfrit grundet deres induktive natur.

Regler for rekusion

- Base Case: Der skal altid være en base case som kan løses uden Rekusion
- Making progress: Hvis kaldet skal håndteres rekursivt skal det altid bevæge sig mod base case.

2.1 Eksempel 1 på rekursion

```
1  int f( int x )
2  {
3     if( x == 0 )
4         return 0;
5     else
6         return 2 * f( x-1)+ x*x;
7  }
```

2.2 Eksempel 2 på rekursion

```
1  public static int sumNaturalNumbers( int n )
2  {
3     if(n < 1)
4         return 0;
5     return n + sumNaturalNumbers(n-1);
6  }
```

2.3 Eksempel 3 på rekursion

```
1  public void walk(String path)
2  {
3      File root = new File(path);
4      File[] list = root.listFiles();
5
6      if(list ==nuul) return;
7
```

```
8       for(File f: list){
9           if(f.isDirectory())
10          {
11              walk(f.getAbsolutePath());
12              maps++;
13          }
14          else
15              files++;
16      }
17  }
```

3 Store O

Tidskompleksitet måler mængden af tid, der kræves af en algoritme for at løse et problem som funktion af inputstørrelsen. Det er en overslags-beregning af det værste tilfælde af inddata. Den højeste potens er den afgørende faktor for algoritmens køretidskompleksitet fordi alt andet, for store værdier af N, er irrelevant.

Der findes fem typiske tidskompleksiteter:

O(1)	Kontant tid. Varigheden er uafhængig af mængden af input
— O(N)	Linær tid. Tiden vokser linært med mængden af input
O(N^2)	Kvadratisk tid. Tiden vokser kvadratisk med mængden af input.
O($log_2 N$)	Logaritmisk tid. Tidsforbruget stiger logaritmisk med mængden af input.
O(2^N)	Eksponentiel tid. Tidforbruget stiger ekponentielt med mængden af input.

Table 1: Almindelige tidskompleksiteter

3.1 Store-O Analyse

For at analysere en fuktion med henblik på at finde tidskompleksitet, gennemgåes koden linje for linje. Med udgangspunkt i nedestående kode, kan det ses at for-loop løber fra $0 \rightarrow N$. Dermed vil funktionen have en tidskomplekitet på O(N). Store O kan ikke indeholde konstanter.

```
1   int sum( int n )
2   {
3       int partialSum;
4
5       partialSum = 0;
6       for( int i = 1; i <= n; ++i )
7           partialSum += i * i * i;
8       return partialSum;
```

```
9   }
```

Hvis der optræder flere forløkker i samme metode, så vil tidskompleksiten blive sat efter "worst case" secenariet hvor man vil sætte store O efter den med mest indflydelse.

Der tages igen udgangspunkt i nedenstående kode. Lægges de tre første løkker sammen får man $LogN * LogN * N * \sqrt{N} = Log(N)^2 * N * \sqrt{N} \Longrightarrow O(Log(N)^2 * N * \sqrt{N})$. Kigger man efterefter på den forløkke der kommer efter de 3 andre, så løber den fra 0 til N*N. Dette giver den en store O på $O(N^2)$

For at afgøre hvilken store O der skal bruges gåes der efter worst case scenario. Eftersom N stiger mere ved N^2 end ved $Log(N)^2 * N * \sqrt{N}$. Så vil store-O for algoritmen være $O(N^2)$

```
1    public static int myMethod( int N )
2     {
3        int x = 0; int y = 0;
4        for (int i = 0; i < N; i++)
5        {
6             for (int j = 0; j < N; j++)
7             {
8                 for (int k = 0; k < N*Math.sqrt(N); k++)
9                 //square root; C++: #include <math.h>
10                {
11                    x++;
12                }
13                j *= 2;
14            }
15            i += i;
16       }
17       for (int i = 0; i < N*N; i++)
18            y++;
19       return x+y;
20    }
```

3.2 Tommelfingerregler

Basis tommelfinger regler til estimering af Store-O

- Halvering af problem → O($LogN$)
- En for løkke → O(N)
- To forløkker i sekvens → O(N)

- For løkke med indbygget halvering → O($N * logN$)
- To nestede forløkker → O(N^2)
- Tre nestede forløkker → O(N^3)
- Tjek alle kombinationer → O(2^N)

4 Liste

En abstrakt datastruktur med en generel liste af formen A0, A1, ..., AN-1, når listens størrelse er N. Hvis listen har størrelsen 0 siges den at være tom. For enhver liste, undtagen den tomme, følger Ai (eller efterfølger) Ai-1 (i¡N), og Ai-1 foregår Ai(i¿0). Det første element i listen er A0, og det sidste er AN-1. Det antages at listeelementerne er heltal, og følgende operationer bruges.

4.1 Operationer på liste

- printList: Udskriver listen
- makeEmpty: tømmer listen
- find: Returnerer positionen af den første forekomst af et givent element.
- insert: Indsætter et element på en given position i listen.
- remove: Fjerner et element fra en given position i listen
- findKth: Returnerer elementet på en given position.

4.2 Implementering af liste

Den nemmeste måde at implementere på er ved at bruge et array. Selvom man opretter arrays med fast kapacitet, tillader vektor-klassen, som internt gemmer et array, at arrayet vokser ved at fordoble sin kapacitet, når det er nødvendigt. Dette løser det mest alvorlige problem ved arrays, som er at man ellers bliver nødt til at estimere størrelsen på forhånd. Ved at bruge arrays kan man nemlig udføre printList på lineær tid og findKth på konstant tid, dog bliver insert og remove dyre alt efter hvor i arrayet det foretages.

Worst Case: Insert på position 0, ender med at skubbe hele arrayet ned i stedet for at gøre plads, og remove på position 0 kræver at alle elementerne flyttes en plads op. Det giver en køretid på O(N).

I gennemsnit skal halvdelen af lsiten flyttes for hver operation, så der kræves stadig lineær tid. På den anden side, hvis alle operationer forekommer i den høje ende af listen, behøver man ikke flytte elementer, og derfor tager insert og remove kun O(1) tid.

4.3 Simple Linked Lists

For at undgå de lineære omkostninger ved insert og remove, skal det sikres at listen ikke gemmes sammenhængende, da man så kommer ud i, at hele dele af listen flyttes rundt. Den sammenkædede liste består af en række noder, som ikke nødvendigvis er tilstødende i hukommelsen. Hver node indeholder elementet og et link til en node, der indeholder dets efterfølger. Vi kalder dette det næste link. Det sidste cells næste link peger til nullptr.

For at udføre printList() eller find(x) starter vi ved den første node i listen og gennemløber listen ved at følge næste-links. Denne operation er lineær-tid, som i array-implementeringen, selvom konstanten sandsynligvis vil være større end hvis en array-implementering blev brugt. findKth-operationen er ikke længere lige så effektiv som en array-implementering; findKth(i) tager O(i) tid og virker ved at gennemløbe listen ned på den åbenlyse måde. I praksis er denne grænse pessimistisk, fordi opkald til findKth ofte er i sorteret rækkefølge (efter i). Som et eksempel kan findKth(2), findKth(3), findKth(4) og findKth(6) alle udføres i én scanning ned gennem listen.

Fjernelsesmetoden kan udføres med én ændring af næste-pegeren. Figur 3.2 viser resultatet af at slette det tredje element i den oprindelige liste.

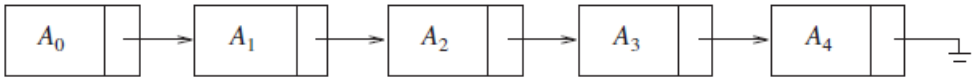


Figure 3.1 A linked list

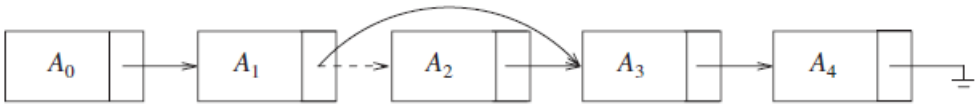


Figure 3.2 Deletion from a linked list

Indsættelsesmetoden kræver at indhente en ny node fra systemet ved at bruge et nyt kald og derefter udføre to næste-peger-manøvrer. Den generelle idé er vist i Figur 3.3. Den stiplede linje repræsenterer den gamle pointer.

Det ses at indsættelse eller fjernelse af et element fra en sammenkædet liste i princippet ikke kræver at flytte mange elementer, hvis vi ved, hvor en ændring skal foretages, og involverer i stedet kun et konstant antal ændringer af node-links. Særtilfældet med at tilføje til fronten eller fjerne det første element er derfor en konstant-tids-operation, forudsat at der opretholdes et link til fronten af den sammenkædede liste. Særtilfældet med at tilføje i slutningen (dvs. gøre det nye element til det sidste element) kan være konstant-tid, så længe vi bibeholder et link til den sidste node. En typisk sammenkædet liste opbevarer altså links til begge ender af listen. Fjernelse af det sidste element er mere kompliceret, fordi vi skal finde det næstsidste element, ændre dets næste-link til nullptr og derefter opdatere det link, der opretholder den sidste node. I den klassiske sammenkædede liste,

hvor hver node gemmer et link til sin næste node, giver det at have et link til den sidste node ingen information om den næstsidste node.

Man kunne tro, at man bare kunne bibeholde et tredje link til den næstsidste node, men det virker ikke fordi den også ville skulle opdateres under fjernelse. I stedet får vi hver node til at opretholde et link til sin foregående node i listen. Dette er vist i Figur 3.4 og er kendt som en dobbelt-sammenkædet liste.

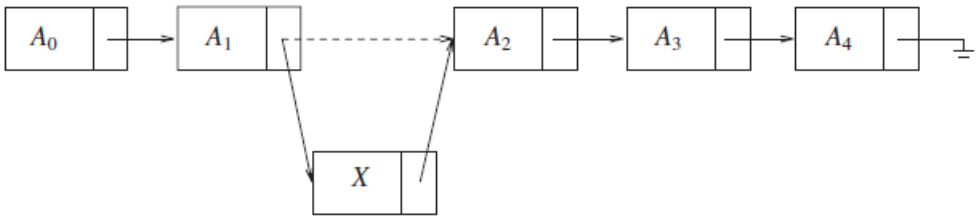


Figure 3.3 Insertion into a linked list

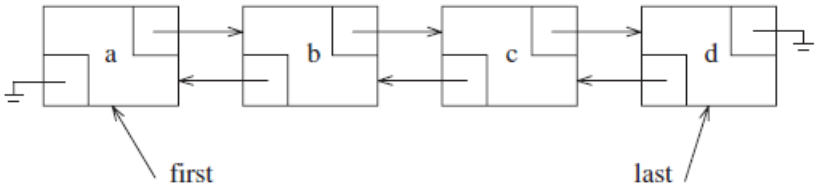


Figure 3.4 A doubly linked list

4.4 Dobblet-samenkædet Liste

Fungerer ligesom sammenkædede lister, men har en ekstra tilføjelse af link til den foregående node i hver celle, hvad giver en mere fleksibel datastruktur. Fordelene herved er en mere effektiv indsættelse og fjernelse: ikke kun ved start og slut, men også midt i listen, da man kan opdatere både det foregående og efterfølgende link. Listen bliver også to-vejs traversal da vi både kan køre forlæns og baglæns. Det gør det også nemt at implementere stak og kø. Nedenstående er strukturen af en node i en dobbelt-sammenkædet liste:

```
class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
```

```
    }
}
```

Man kan indsætte en node efter en given node, ses her i java:

```
    public void insertAfter(Node node, int data) {
    Node newNode = new Node(data);
    newNode.prev = node;
    newNode.next = node.next;

    if (node.next != null) {
        node.next.prev = newNode;
    }
    node.next = newNode;
}
```

5 Stakken

Stakken er en del af de abstrakte datatyper hvor der er et fastlagt type operationer. Grundlæggende er stakken en liste hvor man kun kan indsætte eller fjerne ting fra toppen (LIFO: Last-In, First-Out).

Figure 3.33 Stack model: input to a stack is by push, output is by pop and top

De fundementale operationer for stakken er "push", hvilket indsætter et element i toppen af stakken. Den ande operation er "pop" hvilket retunere det første element fra toppen. Hvis man forsøger at benytte "pop" på en tom stak er det en fejl.

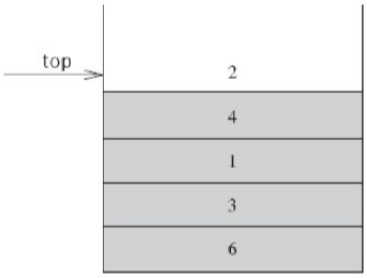


Figure 3.34 Stack model: Only the top element is accessible

5.1 Implementering af stack

Ved implementering af stak bruger man linkede lister hvor hvert element peger til det næste, hvad muliggør effektiv indsættelse og sletning i toppen af stakken, ellers bruges arrays som er lidt simplere hvor man bruger et array til at lagre elementer. Man anvender det til afbalancering af symboler, fx. korrekt matchning af parentser, klammer og bøjler i kode, til postfix-udtryk og til funktionskald så man kan håndtere deres tilknyttede data under kørsel.

6 Køen

Køen er også en form for liste der dog fungerer via FIFO princippet. Der er to hovedoperationer når det kommer til køen. "Enqueue" hvor man tilføjer et element til den bagerste del af køen og "dequeue" hvor man returner det forreste element af køen.

6.1 Anvendelse af Kø

Anvendelsen er primært til scheduling hvor køer bruges til at styre rækkefølgen af opgaver f.eks når flere programmer eller processer skal køre på en enhed. Simulering når køer kan bruges til at modellere virkelige situationer, hvor der er ventetid f.eks ved trafik, og databehandling hvor køer bruges til at organisere data på en måde så det behandles i den modtagede rækkefølge.

6.2 Implementering af kø

Implementering kan man bruge linkede lister og cirkulære arrays, ligesom ved stak. Fordelen ved at bruge og implementere kø er at det er enkelt at forstå og implementere, det er effektivt og meget alsidigt. Man bruger dem særligt til printerkø, på netværk hvor datapakker står i kø inden de sendes afsted og i operativsystemer.

7 Hashing

Den generelle ide bag hasning er at have en tabel med en størrelse der typisk går fra 0 til tablesize -1. Hver Key er derefter tildelt en plads i taballen, udregnet via en hash funktion. Denne funktion skulle ideelt sørge for at at to key's ikke bliver tildelt samme plads. Siden der er et begrænset antal pladser vil dette dog være umuligt og der vil opstå kollisioner. Derfor kan man benytte sig af forskellige strategier for at undgå dette.

Hashing er en abstrakt datatype, hvor objekter kan hentes og gemmes i konstant tid O(1). Det kræver at hvert objekt har en éntydig nøgle, da genfinding ellers bliver umulig. I traditionelle løsninger er objekterne gemt i et array hvis størrelse er et primtal. High performance kræver at objekterne er ensartet distribueret i arrayet. Loadfaktoren som er betegnet λ, er lig antal

elementer i tabellen divideret med tabellens størrelse, og derfor altid ≤ 1.00. I de fleste løsninger bør loadfaktoren være < 0.50. Hashing er et eksempel på tradeoff, hvilket betyder, at man får noget på bekostning af noget andet, i dette tilfælde opnår man fart ved at ofre plads. De fleste hashing implementeringer kræver, som nævnt, dobbelt så meget plads, som data for at opnå nogenlunde konstant tid ved indsættelse og læsning af objekter.

7.1 Hash Function

Den mekanisme, som placerer et objekt i arrayet, kaldes en hash function. Der findes både gode og dårlige funktion, herunder ses en simpel og effektiv hash funktion:

```
public static int hash (String key, int tableSize)
{
        int hashVal = 0;

        for(int i = 0; i < key.length(); i++)
            hashVal += 37 * hashVal + key.charAt(i);

        hashVal %= tableSize;
        if(hashVal <0)
            hashVal += tableSize;

        return hashVal;
}
```

7.2 Problemer ved Hashing

Uanset hvor god en hashfunktion man vælger, vil det altid forekomme, at to eller flere objekter hasher til det samme indeks i tabellen. Dette kaldes kollision, og kan løses ved forskellige metoder:

- Separate chaining
- Linear probing
- Quadratic probing
- Double hashing

Disse er relativt simple strategier.

7.3 Separate Chaining

Ideen her, er at implementere arrayet som en linked list. Det der så sker, når flere elementer hasher til samme indeks, er at elementerne indsættes i en linked list, altså en kæde.

7.4 Linear Probing

Ideen er, at når der konstateres en kollision, så placeres objektet på den næste ledige plads. Problemer med denne løsning er, at der vil opstå klyngedannelse (clustering) som giver nogle meget lange læse- og skrive-til tider afhængigt af loadfaktoren. I linear probing undersøges hele tabellen i en sekvens, startende fra den originale lokation og indtil der kan findes en tom celle i tabellen.

Funktionen der bruges til rehashing er: $rehash(key) = (n + 1)\%tablesize$.

7.4.1 Eksempel på linear probing

Placering af D $(11 * 4)mod16 = 12$

Placering af E $(11 * 5)mod16 = 7$

Placering af M $(11 * 13)mod16 = 15$

Placering af O $(11 * 15)mod16 = 5$

Placering af C $(11 * 3)mod16 = 1$

Placering af R $(11 * 18)mod16 = 6$

Placering af A $(11 * 1)mod16 = 11$

Placering af T $(11 * 20)mod16 = 12$, og da dette giver en kollison benyttes linear probing

$(12 + 1)mod16 = 13$

0	
1	C
2	
3	
4	
5	O
6	R
7	E
8	
9	
10	
11	A
12	D
13	T
14	
15	M

Table 2: Linear probing

7.5 Quadratic probing

Quadratic probing er med til at eliminere primary clustering. Her vil man benytte sig af en kvadratisk kollisions funktion oftes $f(i) = i^2$

Funktionen der bruges til rehashing er: $rehash(key) = (n + i^2)\%tablesize$.

7.5.1 Eksempel på quadratic probing

Først indsættes 16 på plads 5, her opstår der en kollisoon

$16 mod 11 = 5$

Med quadritc probing bliver det

$(5 + 1^2) mod 11 = 6$

27 indsættes og der opstår igen en kollison

$27 mod 11 = 5$

$(5 + 1^2) mod 11 = 6$

$(5 + 2^2) mod 11 = 9$

1 indsættes

$1 mod 11 = 1$

12 indsættes og der opstår en kollison

$12 mod 11 = 1$

$(1 + 1^2) mod 11 = 2$

23 indsættes og der opstår en kollison

$23 mod 11 = 1$

$(1 + 1^2) mod 11 = 2$

$(1 + 2^2) mod 11 = 5$

$(1 + 3^2) mod 11 = 10$

0	22
1	1
2	12
3	
4	22
5	5
6	16
7	6
8	
9	27
10	23

Table 3: Quadratic probing

7.6 Double Hashing

Double hashing går ud på at man andvender en anden has fuktion når kollisionen opstår. Funktionen der bruges til rehashing er: $rehash(key) = (hash_1 + i * hash_2)\%tablesize$.

7.7 Cuckoo Hashing

Cuckoo Hashing er en effektiv metode til at løse kollisioner i hash-tabeller. Den bruger to hashfunktioner og tilknyttede tabeller, hvor mindst 50% skal være tomt. De to hashfunktioner kan tildele hver element en plads i begge tabeller. Hvis pladsen i tabel 1 er optaget, vil algortiemn forsøge at "skubbe" den nuværende key over til dens plads i tabel 2.

Fordelen ved Cuckoo Hashing er, at det garanterer hurtig opslagstid på O(1).

7.7.1 Eksempel

A: 0,2

B: 0,0

C: 1,4

D: 1,0

E: 3, 2

F: 3,4

0	B
1	C
2	
3	E
4	

Table 4: Tabel 1

0	D
1	
2	A
3	
4	F

Table 5: Tabel 2

7.8 Hopscotch hashing

Hopscoths hashing er en forbedring af linear probing. Hopscotch Hashing begrænser længden af probesekvensen til en fast grænse, MAX_DIST. Hvis en ny indsættelse placerer et element for langt fra dets hash-position, flyttes andre elementer baglæns mod deres egne hash-positioner. Hopscotch Hashing bruger også en bit-arraystruktur (Hop-array) til at spore, hvilke positioner der er besat af elementer med en given hashværdi. For eksempel viser bit-arrayet for en given position, hvilke af de næste MAX_DIST-positioner, der indeholder elementer, der hashes til denne position.

7.8.1 Eksempel

Hopscotch-hashingtabel. "Hop"-arrayet angiver, hvilke af positionerne i blokken der er optaget af celler, som indeholder denne hashværdi. For eksempel angiver Hop[8] = 0010, at kun position 10 i øjeblikket indeholder elementer, hvis hashværdi er 8, mens positionerne 8, 9 og 11 ikke gør. A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

	Item	Hop
...	...	...
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000

Table 6: Hopschots hashing tabel

7.9 Rehashing

Når loadfaktoren bliver for stor, skal tabellen udvides, det kaldes rehashing. Dette kan gøres på forskellige måder, enten ved loadfaktor større end 0.5 eller hvis en indsættelse fejler.

8 Prioritetskøer

Prioriteteskøer kommer i spil når man har behov for at kunne afvikle køer ud fra et andet princip end FIFO, eksempelvis ved printjobs, CPU-schedulering eller hasteordrer. Det er en data struktur der bruges til at afgøre hvilke "tasks" der skal køres først. Som minimum skal den have to operationer "insert" og "deleteMin", som fjerner det mindste objekt.

Prioritetskøer kan implementeres via binære søgetræer

8.1 Binary Heap

Binary heap er den mest almindelig implementering af prioritets køer. Binary heap er implementeret som et komplet binært træ hvilket vil sige det er fyld fra venstre mod højre med udtagelse af bunden.

8.2 Basic operationer

Som nævnt tidligere havde prioritetskøer som minimum to operationer, insert, and DeleteMin.

8.2.1 Insert

Ved at indsætte et element i heapen laves der et hul i en tilgængelig position. Hvis elementet kan placeres uden at ændre orden, placeres det direkte eller rykkes det op mod roden indtil ordnen passer. Dette kan ses i nedenstående fig. 1.

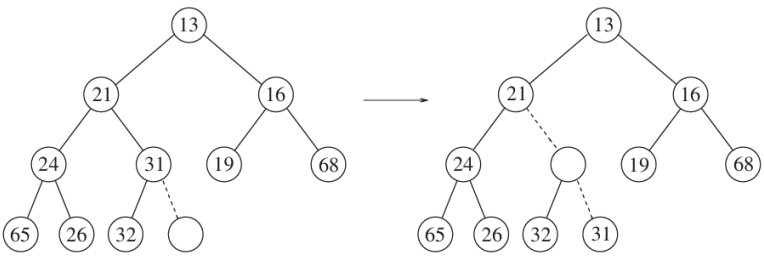


Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

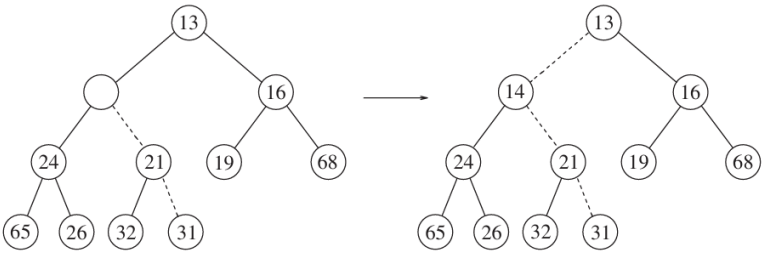


Figure 6.7 The remaining two steps to insert 14 in previous heap

Figure 1: Insert operation

8.2.2 DeleteMin

Når det mindste element fjernes, opstår der et hul ved roden, lignende ved indsættelse, skal hullet udfyldelses igen. Det sidste element skal derfor rykkes op ad i heapen uden at ødelægge ordnen. Dette kan ses i nedenstående fig. 2.

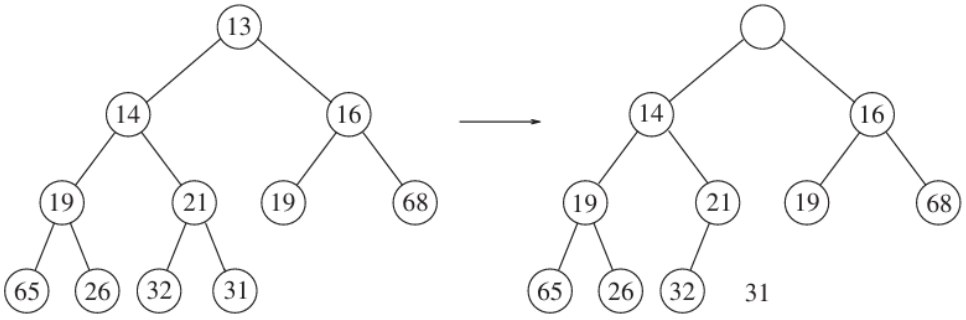


Figure 6.9 Creation of the hole at the root

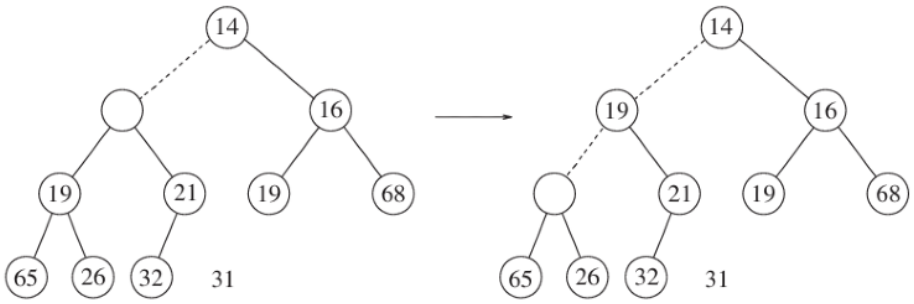


Figure 6.10 Next two steps in `deleteMin`

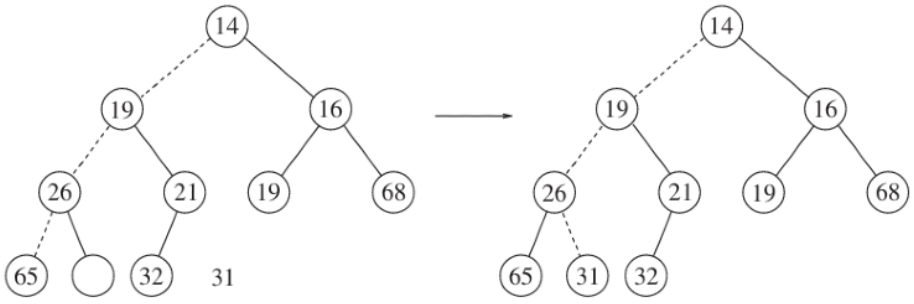


Figure 6.11 Last two steps in `deleteMin`

Figure 2: DeleteMin operation

9 Sortering

Sortering i denne forbindelse går ud på at reducere antallet af inversions skal være 0. En inversion er et talpar hvor $i < j$ men $a[i] > a[j]$. Jo flere inversions en liste indeholder, desto mere usorteret

er den. Sorteringsdisciplinen går ud på at reducere antallet af inversions i en liste til nul. Den nedre grænse for sortering, hvis der kun anvendes sammenligninger er $\Omega(Nlog)$.

9.1 Inversions

Det gennemsnitslige antal inversions i en liste bestående af N forskellige heltal er $\frac{N(N-1)}{4}$. Hvis en operation kan eliminere en inversion, så må den slags sortering have kvadratisk tidskompleksitet $O(N^2)$. Hvis mere end én inversion kan elimineres af én operation, kan vi opnå bedre resultater end kvadratisk tidskompleksitet.

9.2 Sorteringsalgoritmer

Der er nogle specifikke sorteringsalgoritmer som skal kendes og forstås. Når man taler om dem som falder indenfor $O(N^2)$ er de mest brugte insertionsort, selectionsort og selvfølgelig bubblesort. Når man taler om $O(NlogN)$ er det derimod mergesort, heapsort og quicksort som er mest almindelige.

9.3 Bubblesort

Bubblesort er en af de mest grundlæggende sorteringsalgoritmer, og er ikke særligt effektiv. Den fungerer ved at sammenligne parvise elementer i en liste og bytte dem om, hvis de står i den forkerte rækkefølge. Denne process gentages så indtil listen er sorteret. Fordelen er, at den er nem at forstå og implementere, mens ulempen ligger i at den er ineffektiv for store lister da den har en køretid på $O(N^2)$.

```
public void bubbleSort(int array[])
{
    for(int i = 0; i < array.length -1; i++)
    {
        boolean swapped = false;
        for(int j = 0; j < array.length -i-1; j++)
        {
            if(array[j] > array[j+1])
            {
                swap(array,j,j+1);
                swapped = true;
            }
        }
        if(!swapped)
            return;
    }
```

```
}
```

9.3.1 Eksempel på Bubblesort

Hvis vi gives listen [5,2,4,6,1,3] køres bubble sort således: 5 og 2 byttes, da 5 ¿ 2. 4 og 6 forbliver, da 4 ¡ 6. 6 og 1 byttes, da 6 ¿ 1. 6 og 3 byttes, da 6 ¿ 3. Nu haves listen [2,4,1,3,5,6] Herefter køres andet pas, hvor man igen sammenligner. Og dette forsætter indtil listen er sorteret.

9.4 MergeSort

Mergesort er noget kraftigere end bubblesort og baserer sig på princippet "divide and conquer". I stedet for at lave en direkte sammenligning og bytte elementer som i bubblesort, deler mergesort problemet op i mindre dele, og løser dem hver for sig, hvorefter den kombinerer dem på en sorteret måde. Der startes altså med en opdeling hvor listen deles rekursivt i to halvdele, indtil hver del kun indeholder ét element. Derefter laves en sammenføjning (merging) hvor de to sorterede halvdele sammenføjes i en ny sorteret liste. Dette gøres ved at sammenligne det første element i hver halvliste og placere det mindste i den nye liste. Denne process fortsætter så indtil alle elementer er blevet kopieret til den nye liste.

```
private static <AnyType extends Comparable <? super AnyType >>
void mergeSort(AnyType[] a, AnyType [] tmpArray, int left, int right)
{
    if(left < right)
    {
        int center = (left + right) / 2;
        mergeSort(a, tmpArray, left, center);
        mergeSort(a, tmpArray, center+1, right);
        merge(a, tmpArray, left, center+1, right);
    }
}
/**
* Mergesort algorithm
* @param a an array of Comparable items.
*/

public static <AnyType extens Comparable <? super AnyType>>
void mergeSort(anyType[] a)
{
    AnyType [] tmpArray = (AnyType[]) new Comparable [a.length];
    mergeSort(a, tmpArray, 0, a.length-1);
```

```
}
```

Merge algoritmen:

```
// Precondition: no duplicates
public static int[] merge(int [] a, int[] b)
{
    int [] m = new int[a.length+b,length];
    int pointerA = 0; int pointerB = 0; int pointerM = 0;

    while (pointerA < a.length && pointerB < b.length)
        if (a[pointerA] < b[pointerB])
            m[pointerM++] = a[pointerA++];
        else
            m[pointerM++] = b[pointerB++];
    while (pointerA < a.length)
        m[pointerM++] = a[pointerA++];

    while (pointerB < b.length)
        m[pointerM++] = b[pointerB++];

    return m;
}
```

9.4.1 Eksempel på Mergesort

Hvis vi gives listen [5,2,4,6,1,3], laves først en opdeling af listen i to [5,2,4] og [6,1,3], derefter deles disse igen indtil vi har lister som kun indeholder et element hver. Derefter laves sammenføjningen hvor de mindste elementer fra hver del sammenlignes og placeres i den nye liste. Processen fortsætter til alle elementer er blevet kopieret. Fordelen her er at effektiviteten er bedre og køretiden hurtigere end bubblesort, mens ulempen er at der kræves ekstra plads til at opbevare de midlertidige lister under sammenføjningen.

9.5 QuickSort

Quicksort er en ret effektiv sorteringsalgoritme der benytter samme princip som MergeSort. Her vælges et pivotelement som kommer til at fungere som grænsen mellem de mindre og større elementer. Derefter foretages en partitionering hvor listen opdeles i to med elementer der er større end eller lige med pivot, og en hvor de er mindre. Pivoten placeres på sin korrekte position mellem de to dele. Derefter laves rekursion på de nye dellister, indtil hver del kun indeholder ét element, og

altså per definition er sorteret. Fordelen her er hastigheden og behovet for minimal hukommelse, dog kan man ende med en kvaratisk køretid hvis man vælger sin pivotværdi uhensigtsmæssgit og samtidigt er det ikke en stabil algoritme.

```
public void quickSort(int array[], int left, int right)
{
    if(left + CUTOFF < right)
    {
        int pivot = median3(array, left, right);
        int i = left, j = right-1;
        for(;;)
        {
            while(array[i]<pivot)i++;
            while(array[j]>pivot)j--;
            if(i < j)
                    swap(array, i, j);
            else
                    break;
        }
        swap(array, i, right-1);

        quickSort(array, left, i-1);
        quickSort(array,i+1,right);
    }
    else
        insertionSort(array, left, right);
}
```

9.5.1 Eksempel QuickSort

Hvis vi gives listen [5,2,4,6,1,3], vælges 5 som pivot. Først partitioneres så der er en liste med elementer mindre end 5: [2,4,1,3] og en liste med elementer som er lig eller større end 5: [6]. Processen foretages nu for de to dellister.

9.6 ShellSort

Shellsort er en 'multi-pass' algoritme. Hvert pass er en insertion sort af sekvenser bestående af hvert h.element for et fast (fikseret) mellemrum h (increment). Dette kaldes for h-sortering. Analysen her er besværlig og simuleringer antyder at best case er $O(N^{\frac{5}{4}})$, men det er ikke bevist.

9.7 BucketSort

Sorteringsalgoritme som fungerer ved at fordele elementerne i en liste med flere buckets. Hver spand svarer til et bestemt interval af værdier. Når alle elementerne er fordelt, sorteres elementerne i hver spand individuelt (ofte ved hjælp af en anden algoritme såsom insertionSort) og til sidst samles de sorterede spande igen. Tidskompleksiteten her er $O(M + N) \rightarrow O(N)$.

Hvilken betydning har buildHeap()? Sortering er en meget vigtig øvelse, og de triviellle sorteringsalgoritmer har kvadratisks tidskompleksitet. Med buildHeap() kan man opnå $O(NlogN)$, ved først at bygge en heap (prioritetskø) af sine usorterede tal, som kræver N operationer. Derefter kan man kalde deleteMin() på alle elementer, hvad tager N log N operationer. Dermed er tallene sorterede. Altså får man en tidskompleksitet på $N + Nlog(N) \rightarrow O(NlogN)$

10 Træer

Træer består af noder. Den øverste node kaldes roden (root) og er forbundet gennem kanter (edges) til en eller flere subtræer. Roden af hvert subtræ er et barn (child) af roden som dermed kaldes forældre (parent). Noder uden børn kaldes for blade (leaves), noder med den samme forældre kaldes søskende (siblings).

Længden (lenght) kan defineres som antallet af kanter i en sekvens. Dybden (depth) er afstanden af en node til roden. Højden (height) er den længeste distance fra node til et blad. Man taler om at træer kan være komplette, perfekte og balancerede.

10.1 Binære træer

Binære træer defineres som at en node kun har en eller to børn. Hvilket gør at et binær træ kun kan have to subtræer.

Der er tre typer af binære træer.

10.1.1 Komplet binært træ

Komplette binære træer har alle niveauer fra venstre mod højre fyldt.

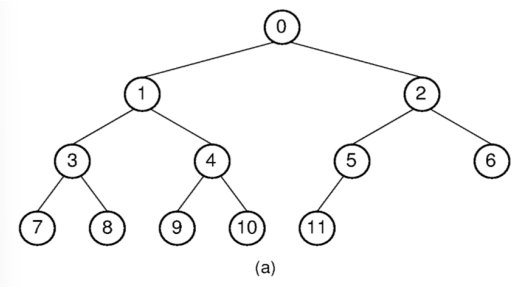


Figure 3: Komplet binært træ

10.1.2 Perfekt binært træ

Et perfekt binært træ har alle niveauer fyldt fra venstre til højre.

$$N = 2^{h+1} - 1$$

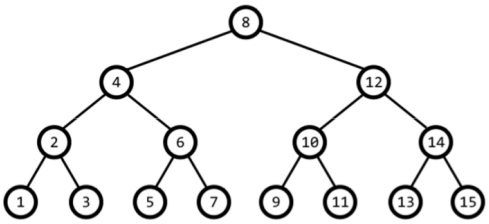


Figure 4: Perfekt binært træ

10.1.3 Balanceret Binært træ

Balancerede binære træer må ikke have en højdeforskel mellem subtræer på mere end en.

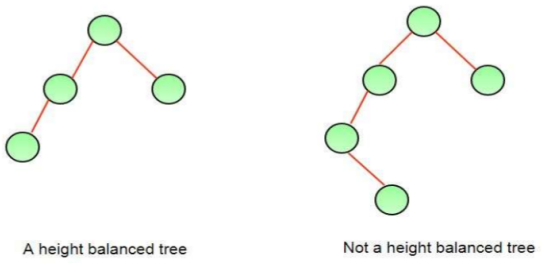


Figure 5: Balanceret binært træ

10.2 Heap order priority

Roden i et binært træ indeholder det mindste element. Enhver node er mindre end alle dens efterkommere, og en prioritetskø kaldes sommetider for en binary heap. Man bruger kommandoen buildHeap() til at konstruere et binært træ, og kalder percolateDown() for alle noder under roden- Man begynder med rightmost deepest som ikke er et blad. BuildHeap() erO(N) og dette bevises ved at finde den øvre grænse som er lig med summen af højderne af samtlige af træet noder- Et perfekt binært træ med højden h har $2^{h+1} - 1$ noder hvad medfører at $N = 2^{h+1} - 1$. Summen af højderne er mindre end N. Buildheap() er vigtigt fordi trivielle sorteringsalgoritmer har kvadratisk tidskompleksitet mens man ved buildheap kan opnå O(N log N) ved først at bygge en heap af sine usorterede tal. Derefter kaldes deleteMin() på alle elementer for at sortere alle tallene.

10.3 Internal path lenght

Den "internal path lenght" er summen af alle dybder af alle noder.

10.3.1 Eksempel

Den internal path lenght af det nedenstående træ i fig. 6 vil være: $2 * 1(B, C) + 4 * 2(D, E, F, G) + 3 * 3(H, I, J) = 19$

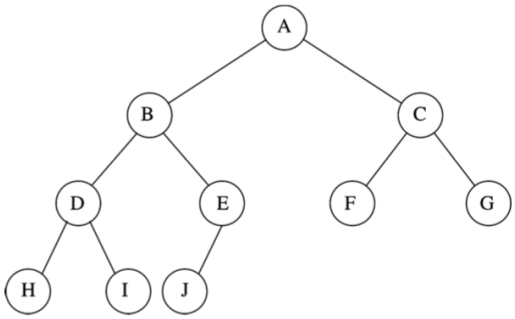


Figure 6: Binært træ

10.4 Traversering

Traversering går ud på at finde en metode til at besøge alle noder i træet. Træer kan besøges på flerre forskellige måder.

10.4.1 Inorder Traversal

Ved inorder travesering besøges træet i følgende rækkefølge. Først besøges venstre subtræ, herefter roden og til sidst højre subtræ, dvs venstre → Rod → højre.

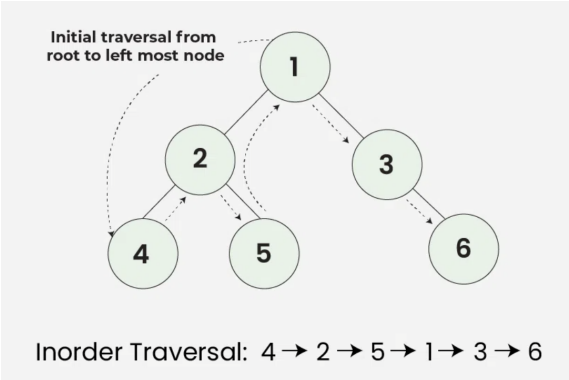


Figure 7: Inorder traversal

10.4.2 Preorder Traversal

Ved preorder travesering besøger man først roden, herefter det venstre subtræ og til sidst det høre subtræ, dvs rod → venstre → højre.

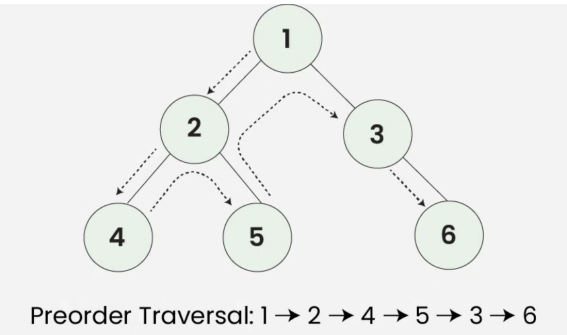


Figure 8: Preorder traversal

10.4.3 Postorder traversal

Postorder travesering besøger først venstre subtræ, herefter højre subtræ og til sidst roden, dvs venstre → højre → rod.

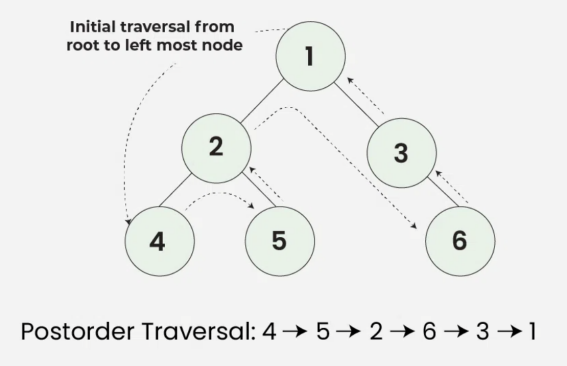


Figure 9: Postorder traversal

10.4.4 Level order Traversal

Ved level order besøges alle noder i det samme niveau før noderne i næste niveau besøges.

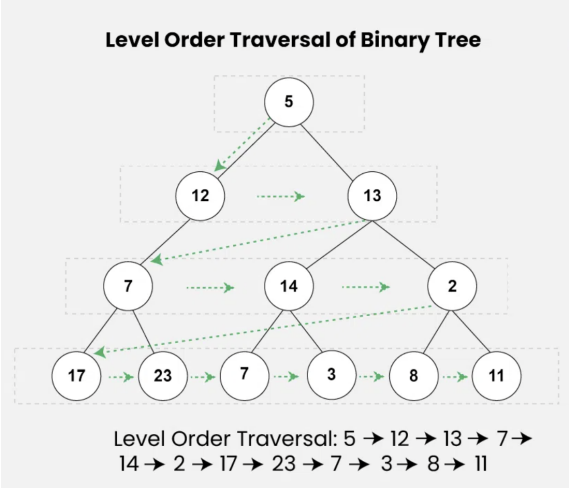


Figure 10: Level order traversal

10.5 AVL-træ

AVL træet er et binært søge trø med en balance condition. For alle noder i træet må variatione i højden af nodens subtræer højest være 1.

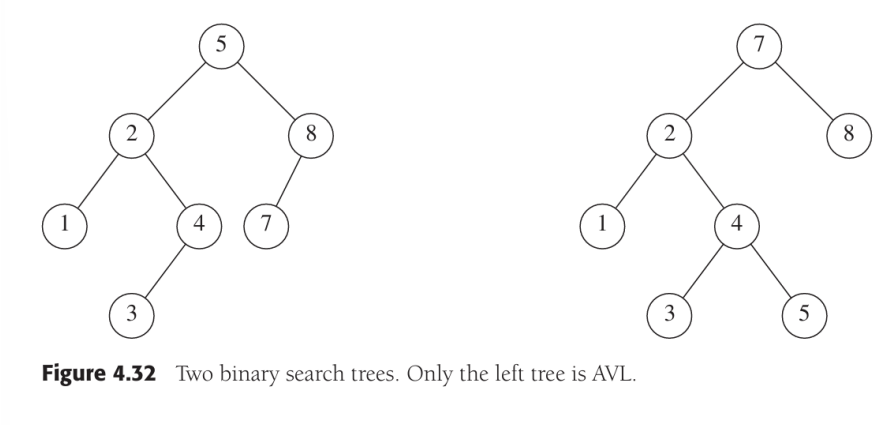


Figure 4.32 Two binary search trees. Only the left tree is AVL.

Figure 11: AVL træ

Ubalancer i træet kan oftest løses med rotationer. Disse opstår typisk på grund af indsættelse eller fjernelse af en node.

10.5.1 Rotationer

En rotation kan enten fortages som enten en venstre eller højre rotation. Der kan fortages flere rotationer på et enkelt træ. I nedenstående fig. 12 er der fortaget en dobbelt rotation. k2 er roteret mod højre og k1 er roteret mod venstre.

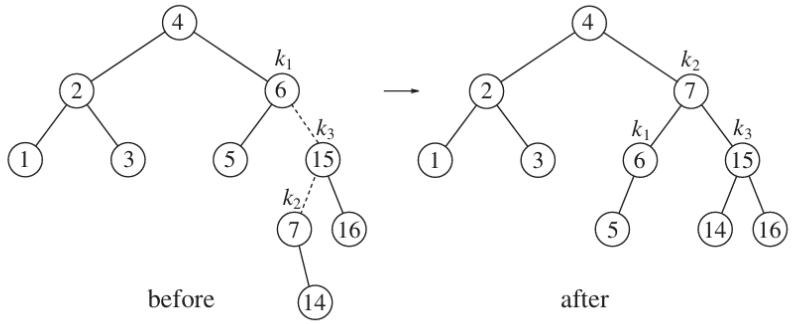


Figure 12: Dobbelt rotation

10.6 buildHeap()

Algoritme der bruges til at omdanne en usorteret liste af tal til en special datastruktur kaldet heap. En heap er som et træ, hvor hver knude har en værdi, og hvor der er bestemte regler for, hvordan værdierne skal være ordnet. Man bruger buildHeap() fordi den er en vigtig del af heap sort algoritmen, og derfor bruges den ofte til at implementere prioriteteskøer hvor det element med den højeste (eller laveste) prioritet altid er øverst.

buildHeap() virker således:

- Start fra bunden: Starter ved det sidste ikke-blad i træet. Et blad er en knude uden børn.
- Siker heap-egenskaber: For hver knude sammenlignes nodens værdi med børnenes værdier. Hvis noden ikke opfylder heap-egenskaben byttes der om på værdierne.
- Gentag: Dette gentages for alle ikke-blad knuder, indtil hele træet opfylder heap-egenskaben.

11 Grafer

En graf (G) består af hjørner (Vertices) og kanter (edges), som også bliver kaldt arcs. Hver kant er en del af et par. Hvis kanterne er sorterer er grafen rettet (directed) og kaldt digraphs. Kanter kan have en tredje komponent kaldet vægt (weight) eller kost (cost). En sti (path) er en sekvens af hjørner forbundet af kanter. Længden er antallet af kanter. En sti fra et hjørne til sig selv har længden 0, hvis der ikke er kanter. En løkke (loop) er en kant fra et hjørne til sig selv. En simpel sti har forskellige hjørner, undtagen start og slut. En cyklus (cycle) i en digraf er en sti, hvor start og slut er ens. I urettet grafer skal kanterne være forskellige. En digraf uden cyklusser kaldes en acyklisk graf (DAG).

11.1 Korteste sti

Der findes forskellige mulighed for at finde den korteste sti mellem en node og en anden node. De 4 typiske muligheder er:

- Unweighted shortest path $O(|E| + |V|)$
- Weighted shortest path $O(|E|log|V|)$
- Weighted shortest path with negative edges $O(|E| * |V|)$
- Weighted acyclic graph special case O(N)

11.1.1 Unweighted shortest path (USP)

Ved USP benytter man sig af breadth-first search hvor man besøger man de noder med den korteste afstand først og derefter besøger man dem længere afstand.

11.1.2 Weighted shortest path (WSP) - Dijkstra

Denne er en grådig algoritme der vælger hvad der virker til at være den bedste løsning i hvert stadie. I hvert stadie vælger algoritmen den node med den med korteste distance, dette gentager den til at alle muligheder for en node er undersøgt. Herefter går den videre til den næste node.

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Figure 9.27 After v_6 is declared *known* and algorithm terminates

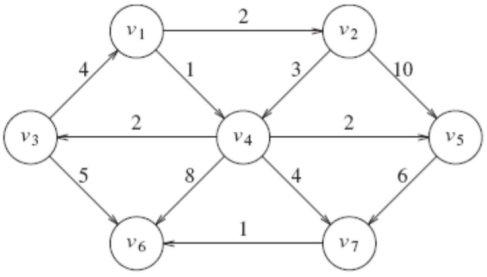


Figure 9.20 The directed graph G (again)

Figure 13: Dijkstra algoritme

11.1.3 Weighted shortest path with negative edges

Hvis grafen har negative kanter virker Dijkstras algoritme ikke, hvis grafen har negative kantvægte. Problemet er, at en kortere sti kan findes tilbage til et kendt hjørne fra et ukendt hjørne. Dette kan resultere i en bedre sti, hvilket bryder algoritmens logik. I stedet for at markere hjørner som kendte, tillader algoritmen at genoverveje beslutninger. Startpunktet s sættes i en kø, og ved hver iteration tages et hjørne v ud. For alle naboer w til v opdateres d_w og hvis $d_w > d_v + c_{v,w} * w$ tilføjes det til køen.

11.1.4 Weighted acyclic graph special case

Hvis grafen er acyklisk, kan Dijkstras algoritme forbedres ved at ændre rækkefølgen af hjørneudvælgelse til topologisk orden. Algoritmen kan udføres i ét gennemløb, hvor opdateringer sker under den topologiske sortering.

11.2 Minimum spanning tree

Mimimum spanning tree er en måde at forbinde alle noder med minimal omkostning, dette er noget der kan gøres med directed graphs. Der findes flere forskellige algoritmer til at gøre dette.

11.2.1 Prim's Algoritmer

Prim's algoritme finder Minimum Spanning Tree (MST) i en vægtet graf ved at tilføje den billigste kant, der forbinder et valgt hjørne med et uvalgt hjørne. Den starter fra et vilkårligt hjørne og fortsætter, indtil alle hjørner er forbundet. Algoritmen sikrer, at grafen forbliver sammenhængende uden at danne cyklusser, og den minimerer den samlede vægt af kanterne i træet.

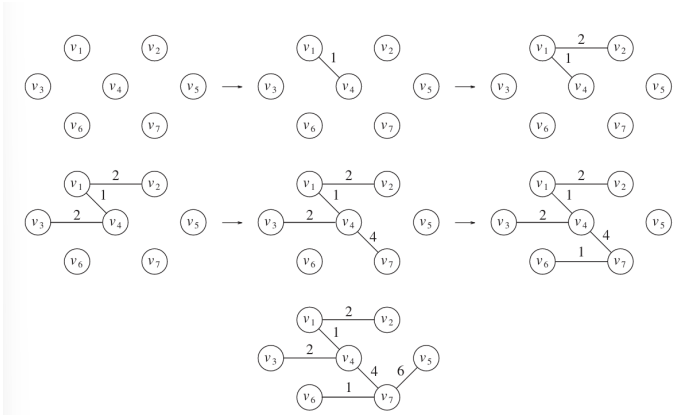


Figure 14: Prim'salgoritme

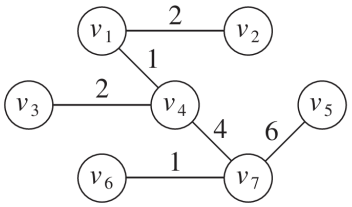
11.2.2 Kruskal's Algoritme

Kruskal's algoritme finder Minimum Spanning Tree (MST) ved at sortere alle kanter efter vægt og tilføje dem én ad gangen, startende med den mindste. Kun kanter, der forbinder uafhængige

komponenter, tilføjes for at undgå cyklusser. Algoritmen fortsætter, indtil alle hjørner er forbundet. Kruskal's metode er effektiv og nem at implementere, især i grafer med få kanter.

Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

(a) Kruskal tabel



(b) Minimum spanning tree

Figure 15: Kruskals algoritme

11.3 Repræsentation af grafer

Adjacency matrix og adjacency list er to forskellige måder at repræsetnere graer på i programmering. Graferne bruges til at modellere relationer mellem forskellige punkter.

11.3.1 Adjacency Matrix

En 2D matrix, hvor rækkerne og kolonnerne repræsenterer noderne i en graf. Et element i matrixen (fx. elementet i række 3, kolonne 5) angiver om der er en kant mellem to noder (mellem node 3 og 5). Hvis der er en kant mellem to noder sættes det tilsvarende element i matrix ti 1, og hvis der ikke er en kant sættes elementet til 0. Fordelen ved denne tilgang er at den er nem at implementere og at det er hurtigt at tjekke om der er en kant mellem to noder, dog er det meget pladskrævende for store grader da alle mulige kanter skal repræsenteres selvom de ikke eksisterer.

11.3.2 Adjacency list

En datastruktur hvor hver node i grafen er tilknyttet en liste af sine naboer. FOr hver node oprettes en liste, der indeholder alle de noder, som den er forbundet med. Det er en effektiv måde at repræsentere grafer med få kanter ift antal noder, og man sparer noget plads sammenlignet med matrix. Dog er det mere komplekst at impleementer og det tager længere tid at tjekke om der er en kant mellem to vilkårlige noder.

11.3.3 Topologisk sortering

En ordning af noder i en directed, acyklisk graf, sådan at hvis der er en sti fra vi til vj, så skal vj komme efter vi i sorteringen. De cykliske grafer kan ikke sorteres topologisk og der kan være mere

end en løsning, alle noder skal besøges.

12 Design

Algoritmedesign er "hjertet" i programmering og databehandling. En algoritme er en præcis rækkefølge af instrukser som løser et specifikt problem. De vigtigste aspekter ved design:

- Effektivitet: Tidskompleksitet (hvor lang tid tager det algoritmen at køre ift inputstørrelse), rumkompleksitet (hvor meget hukommelse er der brug for).
- Korrekthed: Algoritmen skal give korrekte resultater for alle mulige inputs.
- Læsbarhed: Koden skal være let forståelig
- Robusthed: Der skal kunne håndteres uventede inputs og fejl
- Anvendelighed: Kan der generaliseres så man kan løse andre lignede problemer?

12.1 Schedulering

Handler om at tildele ressourcer (fx. processorer, maskiner) til opgaver på en optimal måde. Målet er ofte at minimere ventetid, maksimere gennemstrømning eller minimere energiforbrug. Eksempler på scheduleringsproblematikker inkluderer: Job-shop scheduling: Tildeling af jobs til maskiner på en fabrik, Process scheduling: tildeling af processorer til opgaver i et operativsystem.

12.2 Multiprocessor-systemer

I en multiprocessor arbejder flere processor sammen for at løse en opgave. Algoritmer til multiprocessor-systemet skal tage højde for:

- Parallelisering: Hvordan kan opgaven deles i mindre dele som kan udføres parallelt.
- Kommunikation: Hvordan taler de forskellige processor med hinanden
- Synkronisering: Hvordan sikrerer man, at de forskellige processor udfører deres opgaver i den rigtige rækkefølge.

12.3 Datakomprimering

Handler om at reducere størrelsen af datafiler uden at miste for meget information. Algoritmer til data komprimeren kan være: Lossless hvor alle informationer bevares, og lossy hvor en del af informationen tabes men filstørrelsen reduceres betydeligt. Huffman-kode er et eksempel på lossless komprimering, hvor der tideles korte koder til hyppige sysmboler og lange koder til sjældne symboler.

12.4 Binpacking

Handler om at pakke et sæt genstande med forskellige størrelser i et minimum antal beholdere. Der findes forskellige varianter:

- Optimal: Findes den optimale løsning, dvs. det mindste antal beholdere?
- Next fit: Pak den næste genstand i den første beholder der har plads
- First fit: Pak den næste genstand i den første beholder, der kan rumme den.

12.5 NP-komplette problemer

NP-komplette problemer er en klasse af problemer hvor det er svært at finde en effektiv algoritme til at finde den optimale løsning. Mange vigtige problemer såsom rejsemændenes problem og knapsack problemet er NP-komplette. Knapsack problemet handler om at maksimere værdien af de genstande man kan putte i en rygsæk og samtidigt overholde vægtbegrænsningen.

13 Opgaver

Opgaver tilhørende lektionerne.

13.1 Opgaver i induktionsbeviser

13.1.1 Summen af de første n ulige naturlige tal er n^2

- Basisstegnet: Når n = 1, har vi kun det første ulige tal, nemlig 1. Derfor $1^2 = 1$, altså holder påstanden for n =1.
- Induktionsantagelse: Antag at påstanden holder for et vilkårligt naturligt tal k, dvs. at summen af de første k ulige tal er k^2.
- Induktionsskridtet: Vise at det også holder for $n = k + 1$.
  - Summen for de første k+1 ulige tal er: (summen af de første k ulige tal) + (det (k+1)-te ulige tal). Altså med k^2 hvor det $k + 1$-te ulige tal er $2k + 1$.
  - Summen af de første $k + 1$ ulige tal: $k^2 + (2k + 1)$
  - Faktoriserer: $k^2 + 2k + 1$ og så $(k + 1)^2$

Nu har vi altså vist at hvis påstanden holder for n=k, så holder den også for n=k+1. Visuelt, for n=3, 1+3+5 = 9 = 3^2 eller n=4, 1+3+5+7 = 16 = 4^2.

13.1.2 Summen af de første n kubiktal er $n^2(\frac{(n+1)^2}{4})$

- Induktionsbevis: For n = 1, venstresiden: $1^3 = 1$ og højresiden: $1^2 \cdot \frac{(1+1)^2}{4} = \frac{4}{4} = 1$. Da venstre og højre er ens for n=1, holder påstanden for grundlaget.
- Induktionsantagelse: Antager at påstanden holder for et vilkårligt naturligt tal k, altså at $1^3 + 2^3 + ... + k^3 = k^2 \cdot \frac{(k+1)^2}{4}$
- Induktionsskridt: Viser at påstand holder for $n = k + 1$.
  - Summen for de første k+1 kubiktal er: $1^3 + 2^3 + ... + k^3 + (+1)^3$
  - Erstatter summen af de første k kubiktal: $k^2 \cdot \frac{k+1^2}{4} + (k+1)^3$
  - Vi finder fællesnævner og samler led: $(k+1)^2 \cdot \frac{(k^2+4(k+1))}{4}$
  - Udvider parenteserne i tælleren: $\frac{(k+1)^2 \cdot (k^2+4k+4)}{4} = \frac{(k+1)^2 \cdot (k+2)^2}{4}$

Dermed er det altså vist, at hvis påstanden holder for $n = k$ holder den også for $n = k + 1$. Den gælder altså for alle naturlige tal n.

13.1.3 $\sum_1^n \frac{1}{(2n-1)(2n+1)} = \frac{n}{2n+1}$

- Induktionsbevis: For n = 1, venstresiden: $\frac{1}{((2\cdot1-1)(2\cdot1+1))} = \frac{1}{3}$, højresiden: $\frac{1}{2\cdot1+1} = \frac{1}{3}$. Da siderne er ens for n=1, holder påstanden grund.
- Induktionsantagelse: Antag påstanden gælder for et vilkårligt naturligt tal k, altså $\sum_{n=1}^{k} \frac{1}{((2n-1)(2n+1))} = \frac{k}{(2k+1)}$
- Induktionsskridt: Viser påstand holder for $n = k + 1$
  - Summen af de første $k + 1$ led: $\sum_{n=1}^{k+1} \frac{1}{((2n-1)(2n+1))}$
  - Omskriver: $\sum_{n=1}^{k+1} \frac{1}{((2n-1)(2n+1))} + \frac{1}{((2(k+1)-1)(2(k+1)+1)}$
  - Erstatter første sum: $\frac{k}{(2k+1)} + \frac{1}{((2k+1)(2k+3))}$
  - Finder fællesnævner: $\frac{(k(2k+3)+1)}{((2k+1)(2k+3))}$
  - Samler led: $\frac{(2k^2+3k+1)}{((2k+1)(2k+3))}$
  - Tælleren kan faktoriseres: $\frac{((2k+1)(k+1))}{((2k+1)(2k+3))}$
  - Forkort med $(2k + 1)$: $\frac{(k+1)}{(2k+3)}$

Dermed er det vist at at påstanden holder for alle naturlige tal n.

13.1.4 $n = x \cdot 4 + y \cdot 5 (n \geq 12; y \geq 0$

For denne giver det ikke mening at bruge et klassisk induktionsbevis, men man kan i stedet anvende matematiske teknikker til at analysere og bevise forskellige egenskaber ved løsningsmængden. For

at bevise at der altid findes en løsning for $n \geq 12$ kunne man bruge:

- Induktionsbevis: For $n = 12$, findes $x = 3$ og $y = 0$
- Induktionsantagelse: Antager at der findes en løsning for et vilkårligt $n \geq 12$.
- Induktionsskridt: For $n + 1$ kan vi finde en løsning ved at:
  - Hvis den tidligere løsning for n havde x ¿ 0, så kan vi reducere x med 1 og øge y med 1 for at få en løsning for n+1.
  - Hvis den tidligere løsning for n havde x = 0, så må y være mindst 2. Vi kan så reducere y med 2 og øge med 5 for at få en løsning for n+1.

Dermed er det altså vist, at hvis der er en løsning for n, så er der også en løsning for n+1. Da vi har vist, at der er en løsning for n = 12, følger det ved deduktion, at der er en løsning for alle $n \geq 12$.

13.1.5 $7^n - 1$ er dividerbar med 6 for n > 0

- Induktionsbevis: For n=1 fås $7^1 - 1 = 6$, og da 6 er delelig med 6 holder påstanden for n =1.
- Induktionsantagelse: Antager at påstanden holder for vilkårlige naturlige tal k, altså at $7^k - 1$ er delelig med 6. Altså om der findes et heltal m sådan at $7^k - 1 = 6m$.
- Induktionsskridt: Viser at påstanden holder for $n = k + 1$
  - Summen: $7^{k+1} - 1$
  - Omskriver: $7 \cdot 7^k - 1$
  - Ud fra induktionsantagelsen vides at $7^k = 6m + 1$ som indættes til $7 \cdot (6m + 1) - 1$.
  - Udregner: $42m + 1 - 1$
  - Faktoriserer 6 ud: $6 \cdot (7m + 1)$

Det er altså vist, at hvis påstanden holder for $n = k$, holder den også for $n = k + 1$, og dermed alle naturlige tal som er større end 0.

13.2 Opgaver i rekursion - Java

Opgaverne skal løses med rekursion.

13.2.1 int sum(int n)

Skal returnere summen af de første n naturlige tal. Basistilfældet: når n er 1, er summen 1. Rekursive tilfælde: når summen af de første n tal er n plus summen af de første n-1 tal.

```java
public static int sum(int n){
    if (n == 1){
        return 0;
    } else {
        return n + sum(n-1);
    }
}
```

13.2.2 int evenSquares(int n)

Skal returnere summen af kvadraterne af de første n lige tal. Basistilfældet: når n er 1, er summen 4 (2^2). Rekursive tilfælde: Summen af kvadraterne af de første n lige tal er kvadratet af det n'te tal plus summen af kvadraterne af de første n-1 lige tal.

```java
public static int evenSquares(int n){
    if (n == 1) {
        return 4; //2^2
    } else {
        return (2*n)*(2*n)+evenSquares(n-1);
    }
}
```

13.2.3 int fib(int n)

Det n'te Fibonacci tal. Basistilfælde: Det 0. og det 1. Fibonacci tal er begge 1. Rekursiv tilfælde: Det n'te Fibonacci-tal er summen af de to foregående Fibonacci-tal.

```java
public static int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

13.2.4 bool linear(string s, char c, int l)

Tjekker om en karakter findes i en streng. Basistilfælde: Hvis slutningen af strengen nås uden der er fundet en karakter, returneres false. Rekursiv tilfælde: Tjekker om den aktuelle karakter er den vi ledte efter, og hvis ikke kaldes metoden rekursivt på resten af strengen.

```java
public static boolean linearSearch(String s, char c, int i) {
    if (i == s.length()) {
        return false; // Karakteren er ikke fundet
    } else if (s.charAt(i) == c) {
        return true; // Karakteren er fundet
    } else {
        return linearSearch(s, c, i + 1); // Fortsæt søgningen
    }
}
```

13.2.5 bool binarySearch(int arr[], int value)

Basistilfælde: Hvis søgeområdet er tomt, findes værdien ikke. Rekursiv tilfælde: Finder midtpunkt, hvis midtpunktsværdien er lig søgefærdigen er vi færge, hvis det er større end søgesøgeværdien søges i venstre halvdel, ellers søges i højre.

```java
public static boolean binarySearch(int[] arr, int low, int high, int value) {
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == value) {
            return true;
        }
        if (arr[mid] > value) {
            return binarySearch(arr, low, mid - 1, value);
        }
        return binarySearch(arr, mid + 1, high, value);
    }
    return false;
}
```

13.3 Opgaver i rekursion - C++

13.3.1 int sum(int n)

Opgaven går ud på at returnere summen af de første n tal, indtil n er 0. Basis tilfælde: Hvis n er = med 0 retuneres 0.

```cpp
int sum (int n) //keeps adding until n becomes 0
{
    if(n != 0)
    {
```

```
5           return n + sum(n - 1);
6       }
7       return 0;
8   }
9
```

13.3.2 int evenSquares(int n)

Skal returnere summen af kvadraterne af de første n lige tal. Basistilfældet: når n er ligmed 0 retuneres 0.

```
1   int evenSquares(int n) {
2    //Check if n is not 0
3       if(n != 0){
4           //Check if n is an even number
5           if (n % 2 == 0)
6           {
7               //Calculate power and return the number + substract one from n
8               return n*n + UnevenSquares(n-1);
9           }
10          else
11          {
12              //If not odd subtract 1 from n
13              return (UnevenSquares(n - 1));
14          }
15      }
16      return 0;
17
```

13.3.3 int fib(int f)

Skal returnere det n'te fibonacci tal. Basistilfældet: hvis f er ligmed 1 eller 0 så retuneres f.

```
1   // opg 3 return the n'the fib number
2   // Virker med 0 indeksering
3
4   int fib(int f) {
5      if((f==1)||(f==0)) {
6         return(f);
7      }else {
8         return(fib(f-1) + fib(f-2));
```

```
    }
}

```

13.3.4 bool linear(string s, char c, int l)

Skal retunere "true" hvis strengen indeholder karakteren c, ellers retuneres "false". Basecase: Hvis længden er ligmed 0 så retuneres "false", da enden på strengen er noget uden at finde karakteren.

```
bool linear(string s, char c, int l) {
    // Base case: if length is zero, we've reached the beginning without finding the character
    if (l == 0) {
        return false;
    }
    // Check if the last character in the current substring matches
    if (s[l - 1] == c) {
        return true;
    }
    // Recursive call, reducing the length by 1
    return linear(s, c, l - 1);
}
```

13.3.5 bool binarySearch(int arr[], int left, int right, int value)

Skal retunere "true" hvis værdien bliver fundet i array ellers skal der retuneres "false" Basecase: Der retuneres false hvis værdien ikke er fundet

```
bool binarySearch(int arr[], int left, int right, int value) {
    if (left > right) {
        return false; // Base case: not found
    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == value) {
        return true; // Value found
    }
    else if (arr[mid] > value) {
        return binarySearch(arr, left, mid - 1, value); // Search left half
    }
    else {
```

```
15          return binarySearch(arr, mid + 1, right, value); // Search right half
16      }
17  }
```

13.4 Opgaver i Tidskompleksitet

13.4.1 Store O tidskompleksitet i givet kode

```
1   public static int myMethod(int[] arr) {
2       int x = 0;
3       for (int i = 0; i < arr.length; i++) {
4           for (int j = 0; j < arr.length / 2; j++) {
5               for (int k = 0; k < arr.length; k++) {
6                   x++;
7                   if (k == 1) {
8                       break;
9                   }
10              }
11          }
12      }
13      return x;
14  }
```

Der laves en tidskompleksitetsanalyse:

- Yderste løkke (i): Løkken itererer arr.length gange
- Miderste løkke(j): Itererer arr.length/2 gange indeni hver iteration af den yderste løkke.
- Inderste løkke(k): Itererer arr.length gange i hver iteration af den miderste løkke. Break statementen efter hver iteration af k løkken begrænser dog execution til kun en iteration.

Samlet: $arr.length \cdot \frac{arr.length}{2} \cdot 1 = O(n^2)$, det er altså tidskompleksiteten hvor n er længden af input-arrayet arr.

13.4.2 Store O tidskompleksitet i givet kode

```
1   public static int myMethod(int[] arr)
2   {
3       int x = 0;
4       for(int i = 0; i < arr.length/2; i++)
5           for (int j = 0; j < arr.length; j++)
6               for(int k = 0; k < arr.length; k++)
```

```
7               {
8                   x++;
9                   if(k == arr.length/2)
10                      break;
11              }
12      return x;
13  }
```

Der laves en tidskompleksitetsanalyse:

- Yderste løkke (i): $\frac{arr.length}{2}$
- Miderste løkke(j): $arr.length$ for hver iteration af i
- Inderste løkke(k): $arr.length$ for hver iteration af j, men brea-sætningen afbryder efter den første iteration. Effektiv køres derfor kun 1 gang for hver iteration af de ydre løkker.

Samlet: Antalleat af iterationer af den inderste løkke hvor x inkremeenteres, er:

$$\frac{arr.length}{2} \cdot arr.length \cdot 1 = \frac{(arr.length)^2}{2}$$

. Da vi er interesserede i den asymptotiske adfærd ses bort fra konstanten 1/2. Derfor er tidskompleksiteten her: $O(n^2)$, hvor n er længden af arrayet arr.

13.4.3 Store O tidskompleksitet i givet kode

```
1   public static int func2(int N)
2   {
3       int res = 0;
4       for (int i = 0; < N; i++)
5           res = res + 1;
6
7       return res;
8   }
9
10  public static int func1(intN)
11  {
12      int x = 0;
13      for (int i = 0; i < N; i++)
14          x = x + func2(N);
15      return x;
16  }
```

Der laves en tidskompleksitetsanalyse af func1:

- Yderste løkke (i): Udføres N gange, for hver iteration af i, udføres func2, hvad tager O(N) tid.
- Inderste løkke(k): Udføres N gange for hver iteration af i, og har selv en tidskompleksitet på O(N).

Samlet: For func1 bliver den da $O(N^2)$, altså vokser udførelsestiden af func1 kvadratisk med inputstørrelsen N. Hvis man fordobler inputstørrelsen vil udførelsestiden altså firedobles.

13.5 Implementering af stak - Java

Implementer din egen stak og skriv en metode. Der må ikke bruges tællere men stakken skal bruges. Metoden skal tjekke om parenteser() i parameteren er balanceret. En udvidet version kan omfatte og [].

```
1  import java.util.Stack;
2
3  public class ParenthesesChecker {
4
5      public static boolean balPar(String text) {
6          Stack<Character> stack = new Stack<>();
7
8          for (char c : text.toCharArray()) {
9              switch (c) {
10                 case '(', '{', '[':
11                     stack.push(c);
12                     break;
13                 case ')':
14                     if (stack.isEmpty() || stack.pop() != '(') {
15                         return false;
16                     }
17                     break;
18                 case '}':
19                     if (stack.isEmpty() || stack.pop() != '{') {
20                         return false;
21                     }
22                     break;
23                 case ']':
24                     if (stack.isEmpty() || stack.pop() != '[') {
```

```
                        return false;
                    }
                    break;
                }
            }

            return stack.isEmpty();
        }

        public static void main(String[] args) {
            String[] expressions = {"()", "()[]{}", "{[()]}", "([)]", "{[}"};
            for (String expression : expressions) {
                System.out.println(expression + ": " + balPar(expression));
            }
        }
}
```

Forklaringen hertil:

- Stack: Der bruges en stack til at holde styr på åbningsparenteserne. Når vi møder en åbningsparentes, lægges den i stakken.
- Iterering over strengen: Vi gennemgår hver karakter i strengen. Hvis karakteren er en åbningsparentes pushes den på stakken. Hvis stakken er tom, er der en lukkeparentes uden en tilsvarende åbningsparentes. Hvis den øverste karakter på stakken ikke matcher den aktuelle lukkeparentes, er der ikke balance.
- Efter løkken: Hvis stakken er tom efter gennemløbning af hele strengen, er alle parenteser balancerede, ellers er der for mange åbningsparenteser.

13.6 Implementering af stak - C++

Implementer din egen stak og skriv en metode. Der må ikke bruges tællere men stakken skal bruges. Metoden skal tjekke om parenteser() i parameteren er balanceret. En udvidet version kan omfatte {} og [].

```
// Define Stack class
class Stack {
private:
    int top;
    char arr[100]; // Store characters for parentheses
```

```
6
7   public:
8       Stack() { top = -1; }
9
10      void push(char x) {
11          if (top >= 99) {
12              cout << "Stack overflow" << endl;
13              return;
14          }
15          arr[++top] = x;
16      }
17
18      char pop() {
19          if (top < 0) {
20              cout << "Stack underflow" << endl;
21              return 0;
22          }
23          return arr[top--];
24      }
25
26      char peek() {
27          if (top < 0) {
28              return 0;
29          }
30          return arr[top];
31      }
32
33      bool isEmpty() {
34          return (top < 0);
35      }
36  };
37
38  bool balPar(string text) {
39      Stack s;
40
41      for (char ch : text) {
42          if (ch == '(' || ch == '{' || ch == '[') {
```

```
43              s.push(ch);
44          } else if (ch == ')' || ch == '}' || ch == ']') {
45              if (s.isEmpty()) {
46                  return false;
47              }
48              char top = s.peek();
49              if ((ch == ')' && top == '(') ||
50                  (ch == '}' && top == '{') ||
51                  (ch == ']' && top == '[')) {
52                  s.pop();
53              } else {
54                  return false;
55              }
56          }
57      }
58      return s.isEmpty();
59  }
60
61  int main() {
62      string test = "{[()]}";
63      string test2 = "{[(((((())))]}";
64      if (balPar(test)) {
65          cout << "Balanced" << endl;
66      } else {
67          cout << "Not Balanced" << endl;
68      }
69
70      if (balPar(test2)) {
71          cout << "Balanced" << endl;
72      } else {
73          cout << "Not Balanced" << endl;
74      }
75      return 0;
76  }
77
```

Forklaringen hertil:

- Et stack object benyttes til at holde styr på åbnings parenteser

- Funktionen gennemløber hele strengen, hvis den støder på en åbnings paratens bliver denne pushet til stakken. Hvis den derimod finder en lukke parentes tjekker dem om der ligger en åbnings paratens, hvis dette er tilfældet popper den stakken, hvis ikke så retunere den "false".
- Når hele strengen er gennemløbet tjekkes der om stakken er tom. Er dette tilfældet er strengen balanceret, hvis ikke er den ubalanceret.

13.7 Implementering af cirkulær kø - Java

En cirkulær kø er en datastruktur, hvor elementer tilføjes og fjernes fra en fast størrelse array. Når vi når enden af arrayet, "cirkler" vi tilbage til begyndelsen. Dette giver en mere effektiv brug af hukommelsen sammenlignet med en traditionel kø, hvor vi ville skulle flytte alle elementer en plads hver gang vi fjernede et element.

```
1  public class CircularQueue<T> {
2      private T[] data; //[] giver mulighed for forskellige datatyper
3      private int front;
4      private int rear;
5      private int size;
6
7      @SuppressWarnings("unchecked")
8      public CircularQueue() {
9          data = (T[]) new Object[10]; // Start med en kapacitet på 10
10         front = 0;
11         rear = -1;
12         size = 0;
13     }
14
15     public void enqueue(T item) {
16         if (isFull()) {
17             // Fordobl størrelsen af arrayet
18             T[] temp = (T[]) new Object[data.length * 2];
19             System.arraycopy(data, front, temp, 0, size);
20             data = temp;
21             front = 0;
22             rear = size - 1;
23         }
24         rear = (rear + 1) % data.length;
```

```
25          data[rear] = item;
26          size++;
27      }
28
29      public T dequeue() {
30          if (isEmpty()) {
31              throw new NoSuchElementException("Queue is empty");
32          }
33          T item = data[front];
34          front = (front + 1) % data.length;
35          size--;
36          return item;
37      }
38
39      public boolean isFull() {
40          return size == data.length;
41      }
42
43      public boolean isEmpty() {
44          return size == 0;
45      }
46
47      public T peek() {
48          if (isEmpty()) {
49              throw new NoSuchElementException("Queue is empty");
50          }
51          return data[front];
52      }
53  }
```

Koden kan nu testes med:

```
1   public class CircularQueueTest {
2       public static void main(String[] args) {
3           CircularQueue<Integer> queue = new CircularQueue<>();
4
5           for (int i = 0; i < 15; i++) {
6               queue.enqueue(i);
7           }
```

```
8
9           while (!queue.isEmpty()) {
10              System.out.println(queue.dequeue());
11          }
12      }
13  }
```

Forklaring af koden:

- Definitioner: public class CircularQueue$< T >$: Dette deklarerer en generisk klasse kaldet CircularQueue. Generics ($< T >$) gør det muligt at oprette objekter af denne klasse med forskellige datatyper (f.eks. $CircularQueue < Integer >$, CircularQueue $< String >$).
- Attributer:
  - private T[] data: Et array til at lagre elementerne i køen. Typen T[] angiver, at arrayet kan indeholde objekter af den type, der er specificeret ved oprettelsen af objektet.
  - private int front: Indeholder indekset for det første element i køen.
  - private int rear: Indeholder indeks for det sidste element i køen.
  - private int size: repræsenterer det aktuelle antal elementer i køen.
- Konstruktør: initialiserer objektet, her public CircularQueue():
  - data = (T[]) new Object[10];: Opretter et nyt array af objekter med en initial kapacitet på 10. Bemærk brugen af @SuppressWarnings("unchecked") for at undertrykke en advarsel, da type-casting af arrays kan være risikabelt.
  - front = 0;: Sætter front til 0, hvilket er indekset for det første element i arrayet.
  - rear = -1;: Sætter rear til -1, hvilket indikerer, at køen er tom.
  - size = 0;: Sætter size til 0, da der ikke er nogen elementer i køen endnu.
- Metoder:
  - enqueue(T item): Tilføjer et element (item) til bagenden af køen.
  - dequeue(): Fjerner og returnerer elementet fra fronten af køen.
  - isFull(): Tjekker om køen er fuld ved at sammenligne size med data.length.
  - isEmpty(): Tjekker om køen er tom ved at kontrollere om size er 0.
  - peek(): Returnerer elementet i fronten af køen uden at fjerne det.

13.8 Opgaver til Hashing

Der gives en opgave med 3 forskellige udgaver af hashtabeller.

13.8.1 Quadratic probing

En hashtabel har 16 elementer, der indsættes 5 som alle hasher til samme position i tabellen med quadratic probing.

```
public class HashTable {
    private int[] data;
    private int size;

    public HashTable(int capacity) {
        data = new int[capacity];
        size = 0;
    }

    public int hash(int key) {
        // En simpel hashfunktion, der altid returnerer 0 for at simulere kollisioner
        return 0; // Alle elementer vil hash til indeks 0
    }

    public void insert(int key) {
        int index = hash(key);
        int i = 0;
        while (data[index] != 0 && i < data.length) {
            index = (index + i * i) % data.length;
            i++;
        }
        if (i < data.length) {
            data[index] = key;
            size++;
        } else {
            System.out.println("Hash table is full");
        }
    }

    public static void main(String[] args) {
        HashTable ht = new HashTable(16);
```

```
        for (int i = 0; i < 5; i++) {
            ht.insert(i);
        }
    }
}
```

Alle elementerne forsøges indsat på indeks 0. Da dette indeks allerede er optaget, vil de efterfølgende elementer blive indsat på indeks 1,4,9,16 (modulo 16 giver 0). Hvis vi derefter vil indsætte et 6. element er det ikke muligt, da alle pladserne er optaget.

13.8.2 Quadratic probing

En hashtabel har plads til 11 elementer. Der indsættes 6 elementer, som alle hasher til samme position i tabellen med quadratic probing

```
public class HashTable {
    private int[] data;
    private int size;

    public HashTable(int capacity) {
        data = new int[capacity];
        size = 0;
    }

    public int hash(int key) {
        // En simpel hashfunktion, der altid returnerer 0 for at simulere kollisioner
        return 0;
    }

    public void insert(int key) {
        int index = hash(key);
        int i = 0;
        while (data[index] != 0 && i < data.length) {
            index = (index + i * i) % data.length;
            i++;
        }
        if (i < data.length) {
            data[index] = key;
            size++;
        } else {
```

```
            System.out.println("Hash table is full");
        }
    }

    public static void main(String[] args) {
        HashTable ht = new HashTable(11);
        for (int i = 0; i < 6; i++) {
            ht.insert(i);
        }
    }
}
```

Alle 6 elementer vil forsøge at blive indsat på indeks 0. Da dette indeks allerede er optaget, vil de efterfølgende elementer blive indsat på indeks 1, 4, 9, 4 (mod 11 giver 4) og 9 (mod 11 giver 9) igen. Hvis vi forsøgte at indsætte et syvende element, ville det ikke være muligt, da alle pladser ville være optaget.

13.8.3 Sletning

Håndtering af sletninger hvis der anvendes probing/open addressing. Når man bruger probing i en hashtabel, opstår der en særlig udfordring ved sletning af elementer. Hvis man blot fjerner et element fra tabellen, kan det forstyrre søgeprocessen for andre elementer, der måtte være blevet indsat på grund af kollisioner og placeret i efterfølgende celler.

Problemet med simpel sletning: Hvis vi har et eksempel med kvadratisk probing; Hvis vi har indsat elementerne A, B, og C, hvor B og C er blevet placeret efter A på grund af kollisioner, og vi så sletter A, vil der være et hul i tabellen. Hvis vi nu søger efter C; vil søgningen stoppe, når den finder det tomme felt, hvor A tidligere lå, selvom C faktisk er i tabellen.

Problemet kan løses ved at man vælger en af følgende fremgangsmåder:

- Markér som slettet: I stedet for at fjerne elementet helt markeres cellen som slettet, og når vi søger efter et element skal vi fortsætte søgningen indtil vi enten finder elementet eller når en tom celle. Det giver os dog den ulempe at søgetiden bliver længere.
- Lazy deletion: Her markerer vi kun elementer som slettede når vi ved at der ikke bliver søgt efter dem igen.
- Rehashing: Når antal slettede elementer overstiger en tærskel, oprettes en ny hashtabel og alle de eksisterende elementer indsættes igen.
- Erstatning: Når et element slettes, flyttes det sidste element i den aktuelle klynge til den slettede position.

13.9 Opgaver i sortering - Java

13.9.1 Sortering af heltal

En tabel indeholder N sorterede heltal. Afgør om tabellen indeholder to tal, hvis sum er lig parameteren X. Metoden skal skrives i to udgaver: én med kvadratisk tidskompleksitet og en med lineær.

Ved kvaratisk løsning gennemgås alle mulige par af tal i tabellen, og der tjekkes om deres sum er lig med X. Tidskompleksiteten er $O(N^2)$ da vi har to indlejrede løkker, som begge kører N gange i værste tilfælde.

```
public boolean findSumQuadratic(int[] arr, int X) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] + arr[j] == X) {
                return true;
            }
        }
    }
    return false;
}
```

Ved lineære løsning udnytter vi at tabellen er sorteret. Vi starter med to pointers, en ved begyndelse og en ved slutning af tabellen. Vi sammenligner summen af tallene på de to positioner med X. Hvis summen er for lille, flytter vi venstre pointer mod højre, og hvis den er for stor, flytter vi højre pointer mod venstre. Denne process fortsætter indtil vi finder et match eller når pointersne krydser hinanden. Tidskompleksiteten er $O(N)$ da listen kun gennemgår listen en gang.

```
public boolean findSumLinear(int[] arr, int X) {
    int left = 0;
    int right = arr.length - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == X) {
            return true;
        } else if (sum < X) {
            left++;
        } else {
            right--;
```

```
        }
    }

    return false;
}
```

13.9.2 Sortering i string

Ordene stale og least er anagrammer, dvs. de indeholder nøjagtigt samme karakterer. Skriv en metode der kan afgøre om de to ord (strings) er anagrammer.

Denne opgave kan løses ved at bruge et hashap(counting chars). Som tæller forekomensten af hvert bogstav i begge ord og sammenligner tællingerne med hinanden.

```
public static boolean areAnagrams(String str1, String str2) {
    if (str1.length() != str2.length()) {
        return false;
    }

    int[] charCounts = new int[26]; // Antager kun små bogstaver
    for (char c : str1.toCharArray()) {
        charCounts[c - 'a']++;
    }
    for (char c : str2.toCharArray()) {
        charCounts[c - 'a']--;
        if (charCounts[c - 'a'] < 0) {
            return false;
        }
    }
    return true;
}
```

Her oprettes et array med 26 elementer hvor hvert alement repræsenterer et bogstav i det engelske alfabet. Derefter tælles forekomsten af hvert bogstav i det første ord, og tællingen gemmes i arrayet. Nu trækkes 1 fra tællingen for hvert bogstav i det andet ord. Hvis tællingen bliver negativ betyder det, at der er flere af et bestemt bogstav i det andet ord end i det første, og dermed ville der ikke være tale om anagrammer. Hvis vi når til slutten af det andet ord uden at finde en uoverensstemmelse, så må de to ord være anagrammer.

Man kunne også vælge at løse ogpaven ved sortering; i så fald ville begge ord sorteres og man ville sammenligne om de sorterede strenge var ens.

```java
public static boolean areAnagrams(String str1, String str2) {
    char[] charArray1 = str1.toCharArray();
    char[] charArray2 = str2.toCharArray();
    Arrays.sort(charArray1);
    Arrays.sort(charArray2);
    return Arrays.equals(charArray1, charArray2);
}
```

Her omdannes begge strenge til char arrays, som derefter sorteres. Arraysne sammenlignes da, og hvis de er ens må de oprindelige strenge være anagrammer.

13.10 Opgaver i sortering - C++

13.10.1 Sortering af heltal

En tabel indeholder N sorterede heltal. Afgør om tabellen indeholder to tal, hvis sum er lig parameteren X. Metoden skal skrives i to udgaver: én med kvadratisk tidskompleksitet og en med lineær.

Ved kvaratisk løsning gennemgås alle mulige par af tal i tabellen, og der tjekkes om deres sum er lig med X. Tidskompleksiteten er $O(N^2)$ da vi har to indlejrede løkker, som begge kører N gange i værste tilfælde.

Ved lineære løsning udnytter vi at tabellen er sorteret. Vi starter med to pointers, en ved begyndelse og en ved slutning af tabellen. Vi sammenligner summen af tallene på de to positioner med X. Hvis summen er for lille, flytter vi venstre pointer mod højre, og hvis den er for stor, flytter vi højre pointer mod venstre. Denne process fortsætter indtil vi finder et match eller når pointersne krydser hinanden. Tidskompleksiteten er $O(N)$ da listen kun gennemgår listen en gang.

```
1  //kvadratisk tidskompleksitet
2
3  bool sum1 (vector<int> &N, int X)
4  {
5      // Iterate through each element in the array
6      for (int i = 0; i < N.size(); i++) {
7
8          // For each element arr[i], check every
9          // other element arr[j] that comes after it
10         for (int j = i + 1; j < N.size(); j++) {
11
12             // Check if the sum of the current pair
```

```
13              // equals the target
14              if (N[i] + N[j] == X) {
15                  return true;
16              }
17          }
18      }
19
20      return false;
21
22  }
23
24  //Linær tidskompleksitet
25  bool sum2(vector<int>& N, int X) {
26      int left = 0;
27      int right = N.size() - 1;
28
29      while (left < right) {
30          int sum = N[left] + N[right];
31          if (sum == X) {
32              return true;
33          } else if (sum < X) {
34              ++left;  // Move the left pointer to increase the sum
35          } else {
36              --right; // Move the right pointer to decrease the sum
37          }
38      }
39      return false;
40  }
41
```

13.10.2 Sortering i string

Ordene stale og least er anagrammer, dvs. de indeholder nøjagtigt samme karakterer. Skriv en metode der kan afgøre om de to ord (strings) er anagrammer.

Opgaven løses ved først at sortere de to strenge og herefter tjekke om de er ens.

```
1  bool anagram (string &s1, string &s2)
2  {
3      // Sort both strings
```

```
    sort(s1.begin(), s1.end());
    sort(s2.begin(), s2.end());

    // Compare sorted strings
    return s1 == s2;

}

```

13.10.3 Sortering af 2 elementer

Antag, at du har en matrix af N elementer, der kun indeholder to forskellige nøgler, sandt og falsk. Giv en O(N)-algoritme til at omarrangere listen, så alle falske elementer går foran de sande elementer. Du må kun bruge konstant ekstra plads.

Opgaven løses ved at benytte sig af to vektorer en i bunden og en i toppen. Disse flytter på elementerne alt om hvilken key de peger på.

```
void rearrange(vector<bool>& arr) {
    int left = 0, right = arr.size() - 1;

    while (left < right) {
        // Move left pointer forward if the current element is already false
        while (left < right && arr[left] == false) {
            ++left;
        }

        // Move right pointer backward if the current element is already true
        while (left < right && arr[right] == true) {
            --right;
        }

        // Swap elements at left and right pointers
        if (left < right) {
            swap(arr[left], arr[right]);
            ++left;
            --right;
        }
    }
}
```

13.11 Opgaver i Træer - C++

13.11.1 Traversering af træer

Denne opgave går ud på at skrive metoder til traversering af et binært træ. Først oprettes en instans af en node i træet. Dette gøres via to pointere.

```
struct Node {
    int data;
    struct Node *left, *right;
    Node(int v)
    {
        data = v;
        left = right = nullptr;
    }
};
```

Herefter oprettes en metode til at traversere træet via Preorder. Dette gøre ved at først gå igennem venstre side af træet og derefter højre og udskrive noden.

```
void Preorder(struct Node* node)
{
    if (node == nullptr)
        return;

    // Deal with the node
    cout << node->data << " ";

    // Recur on left subtree
    Preorder(node->left);

    // Recur on right subtree
    Preorder(node->right);
}
```

Herefter laves en metode til at gå igennem træet via in-order. Denne går først gennem det vesntre træ og outputter en node og derefter det højre subtræ.

```
void Inorder(struct Node* node)
{
```

```
    if (node == nullptr)
        return;

    // First recur on left subtree
    Inorder(node->left);

    // Now deal with the node
    cout << node->data << " ";

    // Then recur on right subtree
    Inorder(node->right);
}
```

Til sidst laves en metode til at gå igennem træet via post-order. Denne går først gennem det vesntre træ, derefter det højre subtræ og outputter til sidst en node.

```
void postorder(Node* node) {

    // Base case
    if (node == nullptr)
        return;

    // Recur on the left subtree
    postorder(node->left);

    // Recur on the right subtree
    postorder(node->right);

    // Visit the current node
    cout << node->data << " ";
}

```

13.12 Opgave i QuickSelect

Selection-problemet er et meget brugt eksempel, opgaven er ret simpel: Man har en liste af usorterede elementer, i dette tilfælde integers, og skal finde det k'ne mindste element i listen. Den åbenlyse løsning er at sortere listen i stigende orden og returnere det k'ne element. Ved at bruge en triviel sorteringsalgoritme fås en kvadratisk tidskompleksitet, og bruges en mere avanceret

kan man opnå $O(N \cdot log(N))$. Der er dog, en måde at opnå lineær tidskompleksitet, nemlig ved at bruge quickSelect. Når der her foretages eksperimenter, skal det gøre på varierende tal og relativt mange af dem (gerne 1000), og der skal findes en måde til at tælle antal instruktioner som metoden udfører, sådan at den kan relateres til N, hvormed tidskompleksiteten kan estimeres.

13.12.1 Metode med priorityQueue

Der skal skrives en metode der løser selection problemet med prioritetskø, og foretag en serie af eksperimenter som indikerer at tidskompleksiteten af metoden er $O(N \cdot log(N)$.

```
import java.util.PriorityQueue;

public class SelectionProblemPQ {

    public static int findKthSmallest(int[] arr, int k) {
        // Create a min-heap (Priority Queue)
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Insert all elements into the min-heap
        for (int num : arr) {
            minHeap.offer(num);
        }

        // Extract the k smallest elements
        for (int i = 0; i < k - 1; i++) {
            minHeap.poll();
        }

        // The top of the min-heap now holds the kth smallest element
        return minHeap.peek();
    }

    public static void main(String[] args) {
        int[] arr = {7, 4, 6, 3, 9, 1};
        int k = 3;
        int kthSmallest = findKthSmallest(arr, k);
        System.out.println("The " + k + "th smallest element is: " + kthSmallest);
    }
}
```

Der laves nu en tidskompleksitetsanalyse:

- Indsætter N elementer i min-heap: $O(NlogN)$
- Udtrækker k-1 elementer fra min-heap: $O(klogN)$
- Samlet: $O(NlogN) + O(klogN) = O(NlogN)$ hvis det antages at $k << N$

13.12.2 Implementering quickSelect

quickSelect skal implementeres og der skal laves en serie af eksperimenter som indikerer at tidskompleksiteten er $O(N)$.

```
public class QuickSelect {

public static int quickSelect(int[] arr, int left, int right, int k) {
    if (left == right) {
        return arr[left];
    }

    int pivotIndex = partition(arr, left, right);

    if (pivotIndex == k - 1) {
        return arr[pivotIndex];
    } else if (pivotIndex > k - 1) {
        return quickSelect(arr, left, pivotIndex - 1, k);
    } else {
        return quickSelect(arr, pivotIndex + 1, right, k);
    }
}

private static int partition(int[] arr, int left, int right) {
    int pivot = arr[right];
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr, i, j);
        }
    }
```

```
        swap(arr, i + 1, right);
        return i + 1;
    }

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] arr = {7, 4, 6, 3, 9, 1};
        int k = 3;
        int kthSmallest = quickSelect(arr, 0, arr.length - 1, k);
        System.out.println("The " + k + "th smallest element is: " + kthSmallest);
    }
}
```

Der laves nu en tidskompleksitetsanalyse:

- Gennemsnits case: $O(N)$
- Worst case: $O(N^2)$ men dette kan afhjælpes ved at bruge randomiseret pivot selection.

13.12.3 Rekursion for quickSelect

Den gennemsnitlige case rekursionsligning for quickSelect skal findes. $T(N) = T(pN) + T((1-p)N) + O(N)$. Her er T(N) tiden det tager at finde det k'ne mindste element i et array med størrelsen N. p er sandsynligheden for at pivot vælges sådan at arrayet er opdelt i to under-arrays, hvoraf det ene har størrelsen pN og det andet har (1-p)N.

13.13 Opgave i grafer og aktivitetsplanlægning

I denne opgave skal der arbejdes med filer, klasser, objekter og vektorer/ArrayLists. Lav en tekstfil ved navn data.txt med følgende indhold:1;A;3 1;B;6 1;C;4 2;D;5 3;E;4 3;F;1 4;G;2 4;H;7 4;I;1 5;J;4. Den type af filer kaldes kommaseparerede filer og bruges ofte til logning af måledata og andre strukturerede data. Her er der tale om data til en aktivitetsplan for et udviklingsprojekt med tre informationer pr aktivitet. Den første information kalder vi event, og den angiver rækkefølgen af aktiviteterne; den anden kalder vi task, og den angiver navnet på aktiviteten; den sidste hedder duration og indeholder aktivitetens varighed i dage.

Opgaven går ud på følgende:

- Lav en klasse ved navn Aktivitet, som indeholder attributterne event (int), task (string) og duration (int) med de sædvanlige metoder (constructors, get-metoder).
- Indlæs aktiviteterne fra filenog isoler attributterne.
- Hver aktivitet transformeres til et Aktivitet-objekt, og objekterne placeres i en vector//ArrayList ved navn tabel.
- Herefter udskrives antallet af aktiviteter (10) og den gennemsnitlige varighed somdecimaltal (3,7 dage).
- Udførelsen af de enkelte aktiviteter er afhængig af attributten event. Fx kan task D(event 2) ikke udføres, før A, B og C(event 1) alle er udført, og task J (event 5) kan ikke udføres før G, H og I (event 4) er udført. Aktiviteter med samme event-nummer kan altså udføres parallelt/samtidigt.
- Der er således en række aktiviteter, der kan kaldes den kritiske vej, hvorom det gælder,at ingen af de indgående aktiviteter kan blive forsinkede, uden at det forsinker heleprojektet. Fx kan task A godt blive forsinket, forditask D alligevel ikke kan starte, førtask B er færdig. Men en forsinkelse af B, forsinker hele projektet.
- Du skal udskrive varigheden af den kritiske vej, dvs. den tid det minimum tager atudføre projektet (26 dage) samt rækkefølgen af tasks (B, D, E, H, J). Opgaven skal løsesved at gennemløbe vector/ArrayList tabel,og ikke ved at læse inputfilen igen.
- Precondition: du kan antage, at events er i stigende orden startende med 1og uden'huller'. Tasks, derimod, kommer ikke altid nødvendigvis i stigende alfabetisk ordensom i dette tilfælde.
- Endelig skal du lave en helt ny tekstfil (eller flere) med andre data, som du kan teste ditprogram yderligere med.I forhold til hvordan programmeringsopgaver løses i industrien, er d et af afgørende betydning, at du følger ovenstående specifikation til punkt og prikke. Fx ikke noget med at bruge andre navne på attributterne end de angivne eller at bruge et array i stedet for en vector/ArraList.

```
package plan;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
```

```
class Aktivitet {
    int event;
    String task;
    int duration;

    public Aktivitet(int event, String task, int duration) {
        this.event = event;
        this.task = task;
        this.duration = duration;
    }

    public int getEvent() {
        return event;
    }

    public String getTask() {
        return task;
    }

    public int getDuration() {
        return duration;
    }

    @Override
    public String toString() {
        return "Aktivitet{" +
                "event=" + event +
                ", task='" + task + '\'' +
                ", duration=" + duration +
                '}';
    }
}

public class AktivitetsPlan {
    public static List<Aktivitet> læsAktiviteterFraFil(String fileName) throws IOException {
        List<Aktivitet> aktiviteter = new ArrayList<>();
```

```
        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] data = line.split(" ");
                for (String activityData : data) {
                    String[] activityParts = activityData.split(";");
                    if (activityParts.length != 3) {
                        System.err.println("Invalid activity format: " + activityData);
                        continue;
                    }
                    try {
                        int event = Integer.parseInt(activityParts[0].trim());
                        String task = activityParts[1].trim();
                        int duration = Integer.parseInt(activityParts[2].trim());
                        aktiviteter.add(new Aktivitet(event, task, duration));
                    } catch (NumberFormatException e) {
                        System.err.println("Invalid number format in activity: " + activityData);
                    }
                }
            }
        }
        return aktiviteter;
    }

    public static void main(String[] args) {
        try {
            List<Aktivitet> tabel = læsAktiviteterFraFil("C:\\Users\\Lenovo\\OneDrive - Syddansk U

            // Print the number of activities
            System.out.println("Number of activities: " + tabel.size());

            // Calculate and print the average duration
            double totalDuration = 0;
            for (Aktivitet aktivitet : tabel) {
                totalDuration += aktivitet.getDuration();
            }
            double averageDuration = totalDuration / tabel.size();
```

```
        System.out.println("Average duration: " + averageDuration + " days");

        // Determine the critical path
        List<Aktivitet> criticalPath = new ArrayList<>();
        int currentEvent = 1;
        while (true) {
            List<Aktivitet> currentEventActivities = new ArrayList<>();
            for (Aktivitet aktivitet : tabel) {
                if (aktivitet.getEvent() == currentEvent) {
                    currentEventActivities.add(aktivitet);
                }
            }
            if (currentEventActivities.isEmpty()) {
                break;
            }
            Aktivitet longestActivity = currentEventActivities.get(0);
            for (Aktivitet aktivitet : currentEventActivities) {
                if (aktivitet.getDuration() > longestActivity.getDuration()) {
                    longestActivity = aktivitet;
                }
            }
            criticalPath.add(longestActivity);
            currentEvent++;
        }

        // Print the critical path and its duration
        int criticalPathDuration = 0;
        System.out.print("Critical path: ");
        for (Aktivitet aktivitet : criticalPath) {
            System.out.print(aktivitet.getTask() + " ");
            criticalPathDuration += aktivitet.getDuration();
        }
        System.out.println("\nCritical path duration: " + criticalPathDuration + " days");

    } catch (IOException e) {
        System.err.println("File not found: " + e.getMessage());
    }
```

```
    }
}
```