Algoritmer og datastrukturer
forelesningsnotater og oppgaver

Dag Nylund

16. november 2020

Innhold

Kapittel 1

Lineære datastrukturer

1.1 Introduksjon

1.1.1 Praktisk informasjon

Vi har ikke satt opp noen spesiell pensumbok i år. Dere får notater, eksempler og oppgaver for hvert tema. Notatene blir lagt ut på pdf-format uke for uke, og samlet til en fil på slutten av semesteret. Referanser til bøker og andre kilder blir gitt. Boka [2] anbefales, men er vanskelig å få tak i. Et alternativ er å kjøpe Java-utgaven. Det fins også gode lett tilgjengelige ressurser på internett.

- Notater og eksempler skal leses før hver forelesning.
- Forelesning mandag 9.15 - ca 11. Forelesningene blir digitale, på Zoom. Rom 209 har 20 plasser, og det betyr at nesten halvparten må følge forelesningene digitalt.
- Første forelesning 24/8 blir kun på Zoom. Senere er det planlagt kombinasjon av fysisk/digital forelesning, altså at jeg noen ganger tar opptak fra 209.
- Lab-oppgaver mandag 11.15-12 og 13.15-16. Dere blir delt inn i grupper. All veiledning blir digital. Zoom brukes til deling av skjerm.
- Obligatoriske oppgaver: Se frister på Canvas. Gruppe-oppgaver. Det kan hende at det gis ulike oppgaver til gruppene.
- Mål: Bli god til å programmere. Kjenne til, og få erfaring med, en verktøykasse.

1.1.2 Viktig å tenke på

- time management
- memory management
- stack og heap - statisk og dymanisk minne
- referanser og pekere

- new og delete
- smarte pekere
- Kan man bli rik på søking?
- Hvorfor sortere?

1.1.3 Algoritme

algoritme clearly specified set of simple instructions to be followed to solve a problem ([2], kapittel 2).

datastruktur En måte å organisere og lagre data på. (Klart definert plassering av data).

data

spill lek/underholdning med mål (vinn/tap betingelse) og regler.

ADT (abstrakt datatype) består av en mengde objekter med en tilhørende mengde operasjoner ([2], 3.1).

ADT ([1]): verktøy for å spesifisere de logiske egenskapene til en datatype. Datatype = mengde av verdier og en mengde av operasjoner på disse. ADT refererer til de matematiske begrepene som definerer datatypen. Når vi definerer en ADT som et matematisk begrep, bryr vi oss ikke om tids- og plass-effektivitet. Implementeringen er ikke tema.

C++ template

1.1.4 eksempler på abstrakt datatype

I dette kapitlet skal vi se eksempler på noen abstrakte datatyper og C++ template klasser for disse.

- array
- std::array
- std::vector
- std::stack
- std::queue
- std::list

En felles betegnelse for disse er C++ containere. Mange av funksjonsnavnene går igjen for hver template klasse. Eksempler på funksjonsnavn er **begin(), end(), size()**.

1.2 Array-lignende datastrukturer

1.2.1 Array

En array er en datastruktur hvor data ligger etter hverandre i minneområdet (statisk minne). Linjene nedenfor er et eksempel på bruk av array. Dette er repetisjon, og er ment som et selvstudie-eksempel.

```
//C-style array
int a[10];
a[1]=12 ;
a[5]=6;
a[8]=9;
a[11]=4; // Går dette bra?
//skrive alle
for (auto i=0; i<10; i++)
std::cout << a[i] << std::endl; // Går dette bra?
```

1.2.2 std::array

std::array anbefales i stedet for array-typen ovenfor. Eksemplet nedenfor viser hvordan man oppretter en slik array-variabel. Legg merke til hvordan arrayen traverseres (hvordan vi blar gjennom arrayen) i for-løkka. Vi bruker en såkalt **iterator** i stedset for en indeksvariabel som ovenfor. En slik iterator er en slags peker, og vi henter ut innholdet som den peker på ved notasjonen ***i**.

```
    std::array<double, 5> a;
    a[0] = 4*atan(1.0);
    a[1] = exp(1.0);
    a[2] = 9.81;
    a[3] = sqrt(3);
    a[4] = sqrt(2);
    std::cout << std::fixed << std::setprecision(7) << std::setw(10);
    for (auto i=a.begin(); i!=a.end(); i++)
        std::cout << *i << std::endl;
```

1.2.3 std::vector

std::vector er den mest anbefalte containeren i C++.

```
    std::vector<int> b(20);
    for (auto i=0; i<20; i++) b[i] = std::rand()%20;
    for (auto i=b.begin(); i!=b.end(); i++) std::cout << *i << " ";
    std::cout << std::endl;
```

1.3 Lenket liste

I dette avsnittet skal vi studere et eksempel på en dynamisk lenket liste. I dynamisk lenket liste bruker vi dynamisk minneallokering. En lenket liste har noder. Dette er navnløse minneområder på dynamisk minne.

1. Vi lager først en enkel lenket liste hvor datatypen er den minst tenkelige, en char. Se CharNode.h og CharNode.cpp.

2. Vi tester innsetting foran i lista, sletting foran og traversering av lista. Se og CharNodemain.cpp. Innsetting og sletting av en vilkårlig node er vanskeligere, og det skal vi ikke gjøre her.

3. Neste steg er å gjøre om denne CharNode klassen til en template klasse. Sammenlign kildekoden for de to klassene.

4. Videre lager vi en template kø-implementering hvor vi benytter Node<T> klassen. Dennee kø-implementeringen kan brukes som eksempel når dere skal lage en egen stakk (for char og template).

En lenket liste skiller seg fra array-lignende typer. Den viktigste forskjellen er at i en liste har vi ikke såkalt random access. Vi kan ikke aksessere et bestemt, indeksert objekt slik som a[1] og a[2] ovenfor. std::vector har funksjonen **at()** for random access, std::list mangler denne.

En liste har to ender og vanligvis tilgang til begge. Vi skiller mellom enkel lenket liste og dobbelt lenket liste. I en enkel lenket liste har hver node en peker til neste node i lista. I en dobbelt lenket liste har hver node både en peker til neste node og forrige node i lista.

Vi skal se på et eksempel på enkel lenket liste. Når vi senere skal bruke denne lista til å implementere stakk og kø, vil ikke den statiske klassevariabelen s_antall kunne brukes til å telle antallet hvis vi har mer enn en stakk eller kø.

Listing 1.1: charnode.h

```
#ifndef CHARNODE_H
#define CHARNODE_H

#include <string>

namespace ADS101 {
    class CharNode
    {
    public:
        CharNode(char tegn='0', CharNode* neste=nullptr);
        std::string toString() const;
        CharNode* hentNeste() const;
        void skrivBaklengs() const;
        static int hentAntall();
        char hentData() const;
        ~CharNode();
    private:
        char m_tegn;              // Data-del
        static int s_antall;
        CharNode* m_neste;        // Datastruktur-del
    };
}
#endif // CHARNODE_H
```

Listing 1.2: charnode.cpp

```
#include <string>
#include <sstream>
#include <iostream>
#include "charnode.h"

namespace ADS101 {
int CharNode::s_antall;

CharNode::CharNode(char tegn, CharNode* neste)
    : m_tegn(tegn), m_neste(neste)
{
    //m_tegn = tegn;
    s_antall ++;
}

std::string CharNode::toString() const
{
    std::ostringstream oss;
    oss << m_tegn;
    return oss.str();
}

CharNode* CharNode::hentNeste() const
{
    return m_neste;
}
void CharNode::skrivBaklengs() const
{
    if (m_neste)
        m_neste->skrivBaklengs();            // Main: liste->skrivBaklengs();
    std::cout << m_tegn;
}
int CharNode::hentAntall()
{
    return s_antall;
}

CharNode::~CharNode()
{
    s_antall --;
}

char CharNode::hentData() const
{
    return m_tegn;
}
}
```

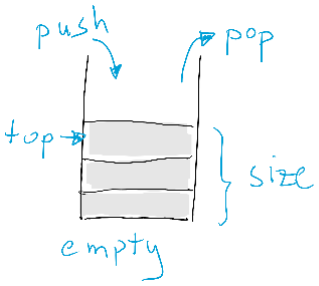
Listing 1.3: charnode_main.cpp

```
#include <iostream>
#include "charnode.h"
#include <stack>
#include <memory>


//using namespace ADS101;
int main()
{
    ADS101::CharNode* liste = new ADS101::CharNode('a');
    liste = new ADS101::CharNode('b', liste);
    liste = new ADS101::CharNode('c', liste);
    liste = new ADS101::CharNode('d', liste);

    //
    // smarte pekere kommer her
    //

    for (ADS101::CharNode* p=liste; p!=nullptr; p=p->hentNeste())
        std::cout << p->hentData();

    std::cout << std::endl << "static antall: " << liste->hentAntall();
    std::cout << std::endl;

    std::cout << "Skriver baklengs; ";
    liste->skrivBaklengs();

    // Slette den forste i lista
    ADS101::CharNode* ut = liste;
    liste = liste->hentNeste();
    delete ut;


    std::cout << std::endl << "static antall: " << liste->hentAntall() << std::endl;

    return 0;
}
```

1.3.1 Smarte pekere

Listing 1.4: tillegg til charnode_main.cpp

```
    std::shared_ptr<ADS101::CharNode> shp(liste);
    std::cout << "shp er unik: " << shp.unique() << std::endl;
    std::unique_ptr<ADS101::CharNode> up1(liste); // burde ikke ga?
    std::cout << "shp er unik: " << shp.unique() << std::endl;

    std::unique_ptr<ADS101::CharNode> up(liste->hentNeste());
    auto tmp = shp.get();
    std::cout << "shared pointer data " << tmp->hentData() << std::endl;
    std::cout << "shared pointer data " << shp->hentData() << std::endl;
    tmp = up.get();
    std::cout << "unique pointer data " << tmp->hentData() << std::endl;
    std::cout << "unique pointer data " << up->hentData() << std::endl;
```



Figur 1.1: stakk figur

1.4 Stakk

En stakk er en datastruktur hvor vi bare kan sette inn og ta ut elementer på toppen. Det er en lineær datastruktur med to ender, men vi har altså bare tilgang til den ene enden. Det siste elementet som settes inn er det første som kan tas ut, og et annet navn på stakk er LIFO. En stakk har operasjonene

- push
- pop
- top
- size
- empty

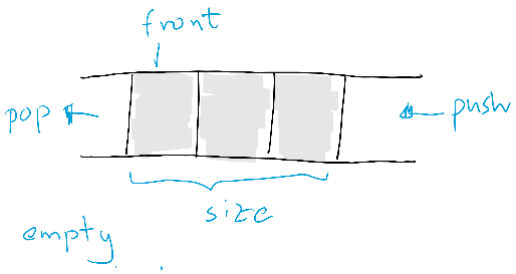
1.5 Kø

En kø er en datastruktur hvor vi setter inn elementer bak og fjerner elementer foran. Det første elementet som blir satt inn, er det første som blir tatt ut. Et annet navn på kø er FIFO. En kø har operasjonene

- push
- pop
- front
- size
- empty

Kø-implementeringen nedenfor er basert på CharNode klassen. Den er endret slik at vi kan lage en kø med hvilken som helst datatype, ikke bare char. CharNode klassen er skrevet om til en template klasse, og i tillegg har vi en template Queue<T> klasse. Når vi skriver en egen template klasse, skriver vi både klassedefinisjonen og implementeringen i h-fil.

I main-funksjonen nedefor ser vi at vår egen template kø har samme funksjonalitet som std::queue.



Figur 1.2: kø figur

1.5.1 template klasser

Her har vi skrevet om CharNode klassen til en template klasse. Når vi sammenligner kildekoden, ser vi først og fremst at alle forekomster av **char** er endret til **T**. Funksjonen hentTegn() er endret til hentData() for å ha et logisk navn, og returtypen er endret tilsvarende. For å oppnaå denne fleksibiliteten, må vi akseptere litt vanskeligere syntaks: Klassendefinisjonen må ha prefikset **template <class T>** og i implementeringen er CharNode erstattet med **Node<T>**.

Listing 1.5: node.h

```
#ifndef NODE_H
#define NODE_H

namespace ADS101 {

template <class T>
class Node
{
public:
    Node(T tegn, Node* neste=nullptr);
    Node<T>* hentNeste() const;
    void settNeste(Node<T>* neste);
    static int hentAntall();
    T hentData() const;
    ~Node();
private:
    T m_tegn;              // Data-del
    static int s_antall;
    Node<T>* m_neste;       // Datastruktur-del
};

template <class T>
int Node<T>::s_antall;

template <class T>
Node<T>::Node(T tegn, Node<T>* neste)
    : m_tegn(tegn), m_neste(neste)
{
    s_antall ++;
}

template <class T>
Node<T>* Node<T>::hentNeste() const { return m_neste; }

template <class T>
void Node<T>::settNeste(Node<T>* neste) { m_neste = neste; }
```

```
template <class T>
int Node<T>::hentAntall() { return s_antall; }

template <class T>
Node<T>::~Node() { s_antall --; }

template <class T>
T Node<T>::hentData() const { return m_tegn; }
}

#endif // TNODE_H
```

1.5.2 // Implementering av kø med enkel lenket liste

I dette eksemplet skal vi bruke den egne template klassen Node<T> til å lage queue<T> og deretter lage og teste en kø av char. En vanlig navnekonvensjon er å bruke store bokstaver på klassenavn. Her bryter vi dette prinsippet for å få nøyaktig samme navn som std sin queue. Eksemplet viser at vi har laget en kø klasse med samme funksjonalitet som std::queue.

Listing 1.6: queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include "node.h"

namespace ADS101 {

template <class T>
class queue
{
private:
    Node<T>* m_front;
    Node<T>* m_bak;

public:
    queue();
    void push(T data);
    T front() const;
    void pop();
    int size();
};

template <class T>
queue<T>::queue()
{
    m_front = m_bak = nullptr;
}

template <class T>
void queue<T>::push(T data)
{
    Node<T>* ny = new Node<T>(data);
    if (m_bak)
        m_bak->settNeste(ny);
    m_bak = ny;

    if (m_front == nullptr)
        m_front = m_bak;
```

```
}

template <class T>
void queue<T>::pop()
{
    if (m_front != nullptr)
    {
        Node<T>* ut = m_front;
        m_front = m_front->hentNeste();
        delete ut;
    }
    // Hvis vi sletter siste node, blir
    // m_bak en dingle-peker :-(
    if (m_front == nullptr)
        m_bak = nullptr;
}

template <class T>
T queue<T>::front() const
{
    return m_front->hentData();
}

template <class T>
int queue<T>::size() { return m_front->hentAntall(); }
}
#endif // QUEUE_H
```

Listing 1.7: queue_test.cpp

```
#include <iostream>
#include "queue.h"
#include <queue>

int main()
{
    std::cout << "std::queue har push(), pop(), front(), size()\n";
    std::cout << "\nTester egen queue\n";

    ADS101::queue<char> queue1;
    queue1.push('a');
    queue1.push('b');
    queue1.push('c');
    auto ch = queue1.front();
    std::cout << "queue front = " << ch << std::endl;
    queue1.pop();
    queue1.pop();
    queue1.push('a');
    std::cout << "queue size = " << queue1.size() << std::endl;

    std::cout << "\nTester std::queue\n";

    std::queue<char> queue2;
    queue2.push('a');
    queue2.push('b');
    queue2.push('c');
    ch = queue2.front();
    std::cout << "queue front = " << ch << std::endl;
    queue2.pop();
    queue2.pop();
    queue2.push('a');
    std::cout << "queue size = " << queue2.size() << std::endl;
```

```
    std::cin >> ch;
    return 0;
}
```

1.6 stl vector og list

Her følger eksempler på bruk av stl klassene vector, list og string.

Listing 1.8: vector_and_list.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

int main()
{
    std::string s1("vector");
    std::string s2("og");
    std::string s3("list");
    std::string s4("er");
    std::string s5("containere");

    std::vector<std::string> v;
    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);
    v.push_back(s4);
    v.push_back(s5);

    std::cout << "v.size() = " << v.size() << std::endl;
    std::cout << "v.capacity() = " << v.capacity() << std::endl;
    std::cout << "v.empty() = " << v.empty() << std::endl;

    // iteratorer
    for (auto it=v.begin(); it!=v.end(); it++)
        std::cout << *it << " ";
    std::cout << std::endl;

    // vector elementer er aksesserbare som i array
    std::cout << v.at(3) << " " << v[3] << std::endl;
    v.clear();
    std::cout << "v.empty() etter v.clear() = " << v.empty() << std::endl;

    // std::list er ogsa en container og har i tillegg
    // push_front() og pop_front()
    std::list<std::string> l;
    l.push_front(s1);
    l.push_front(s2);
    l.push_front(s3);
    l.push_back(s4);
    l.push_back(s5);

    for (auto it=l.begin(); it!=l.end(); it++)
        std::cout << *it << " ";
    std::cout << std::endl;
    return 0;
}
```

1.7 string

Vi så ovenfor et eksempel på bruk av string. std::string klassen har mange av de samme funksjonene som std::vector.

```
std::string s("Dette er en string.");
cout << s << endl << s.c_str() << endl;
cout << "Den har " << s.size() << " tegn." << endl;
```

1.8 Lenker til videoforklaringer

- Lenket liste `https://youtu.be/8ejk0_-_ME0`: Her går jeg gjennom hvordan vi lager en lenket liste klasse detaljert. Denne klassen er enda mindre enn CharNode klassen ovenfor. Videoen ble altfor lang, ca 28 minutter.
- Debugging `https://youtu.be/tGmS9o_6eZo`. Denne innspillingen er en fortsettelse av forrige. Den er en kort demonstrasjon i debugging i Qt.
- Smarte pekere i C++ `https://youtu.be/by61Eftf3xM`
- Smarte pekere del 2-3 `https://youtu.be/PdgwA7uqJdw`
- Litt om Qt GUI: `https://youtu.be/Q3Dm1WUnxqw`.

1.9 Oppgaver

(Oppgaveteksten er litt endret fra sist).

1.9.1

Implementer en char stakk ved å bruke CharNode klassen og lenket liste. Lag et testeksempel.

1.9.2

Implementer en template stakk ved å bruke CharNode klassen og lenket liste. Lag et testeksempel.

1.9.3

Implementer en char kø og deretter en template kø ved å bruke en array til intern lagring. Lag et testeksempel.

1.9.4

Implementer en char stakk og deretter en template stakk ved å bruke en array til intern lagring. Lag et testeksempel.

Kapittel 2

Algoritme-analyse og søking

2.1 Algoritme-analyse, søking og *O(n)*-notasjon

I forrige kapittel studerte vi ulike lineære datastrukturer. Det fins std::array, std::vector og std::list, likevel har vi noen ganger bruk for restriksjonene til stakk og kø. Videre lærte vi definisjonen på en algoritme. I dette kapitlet går vi et steg videre og studerer effektiviteten til en algoritme, og hvordan dette kan måles. Vi skal

- gå gjennom/repetere selectionsort og regne ut hvor effektiv denne sorteringsalgoritmen er.
- gå gjennom binærsøk og regne ut hvor effektiv denne søkealgoritmen er.
- sammenligne selectionsort og std::sort ved å gjøre målinger, og finne ut hvor effektive de to algoritmene er på bakgrunn av måledata.
- se på flere sorteringsalgoritmer senere i kurset.

Vi vet at det fins en motivasjon for effektiv søking. Vi skal først prøve å finne en tilsvarende motivasjon for sortering. Tabellen nedenfor består av 20 tilfeldige usorterte tall i andre rad, og de samme tallene sortert i tredje rad.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
7	11	1	5	3	13	31	2	17	7	19	23	53	37	29	97	7	61	47	59
1	2	3	5	7	7	7	11	13	17	19	23	29	31	37	47	53	59	61	97

1. Hvor mange sammenligninger trengs for å søke etter et tall som fins i arrayen?
2. Hvor mange sammenligninger trengs for å søke etter et tall som fins i arrayen?

Arrayen består av $n = 20$ tall. Anta at vi søker fra begynnelsen og ser på ett og ett tall. Vi ser i det første tilfellet at vi må gjøre i gjennomsnitt ≈ 10 sammenligninger for et tall som fins og 20 sammenligninger for et tall som ikke fins.

I det andre tilfellet (sortert array) trengs i gjennomsnitt ≈ 10 sammenligninger

enten tallet vi søker etter fins eller ikke. Men både $\frac{n}{2} = 10$ og $n = 20$ øker proporsjonal med n, og vi sier at vi må gjøre i størrelsesorden n sammenligninger. Dette har egen egen notasjon, $O(n)$. Vi kommer borti $O(n^2)$, $O(\log n)$, $O(n \log n)$ for å ta med noen eksempler, og alt dette er skrevet på formen $O(n)$.

Vi kan i det minste konkludere med at det er dobbelt så raskt å søke etter et tall som ikke fins i arrayen når tallene er sortert. Sånn sett har vi en motivasjon for sortering.

2.1.1 Selection sort

En av de enkleste og mest kjente sorteringsalgoritmene kalles selection sort. Vi velger ut ett og ett element i arrayen fra begynnelsen og sammenligner med høyere indekserte elementer. Hver gang vi finner et element som er mindre enn a[0], byttes de to. Etter en gjennomløping av arrayen har vi plassert det minste elementet på første plass.

Listing 2.1: selection sort

```
// implementering av selection sort for std::array
void selection_sort(std::array<int,20>& a) // husk referanse
{
    auto n = a.size();
    for (auto i=0; i<n-1; i++)
        for (auto j=i+1; j<n; j++)
            if (a[i] > a[j])
                std::swap(a[i], a[j]);
}
```

2.1.2 Effektivitet

Hvor rask er selectionsort? Vi kan regne ut dette analytisk, eller vi kan måle hvor lang tid det tar. Altså,

- ved å telle antall kritiske operasjoner, eller
- ved å kjøre algoritmen/programmet og ta tiden for forskjellige testdata.

Vanligvis er sorteringstiden til en algoritme avhengig av rekkefølgen på inndata (f.eks. binært søketre sortering). I sorteringsfunksjoner er det vanlig å forsake lesbarhet i koden til fordel for høyest mulig hastighet.

Utregning av effektivitet

Vi teller opp de kritiske operasjonene i algoritmen, det vil si de operasjonene som tar mest tid. I tilfellet her er det sammenligningen og swap i indre løkke. I numeriske sammenhenger kan det være multiplikasjoner og divisjoner (som tar mye mer tid enn addisjoner og subtraksjoner).

Antall kritiske operasjoner i en enkel for-løkke fra 1 til n eller fra 0 til n-1 med en kritisk operasjon for hver gjennomløpning er $\Sigma_{i=1}^{n}\ 1 = n$

Hvis vi har en dobbel for-løkke fra 0 til n-1 med en kritisk operasjon for hver gjenomløpning,

```
for (auto i=0; i<n; i++)
   for (auto j=0; j<n; j++)
      a[i] = i*j;
```

regner vi ut en dobbel sum. Antall multiplikasjoner her er

$$\Sigma_{i=1}^{n}\ \Sigma_{j=1}^{n}\ 1 = n\ \Sigma_{i=1}^{n}\ 1 = n^2$$

som selvfølgelig er $O(n^2)$. Selv om vi ikke gjør multiplikasjoner i selectionsort, blir summeringen den samme. Selectionsort er altså en $O(n^2)$ algoritme.

O(n) notasjonen

Når vi har mange elementer å sortere, altså når n er stor, er n^2 mye større enn n. Herav følger O(n) notasjonen. Når vi teller opp antall kritiske operasjoner i en algoritme, er det høyeste potens av n som betyr mest for hvor rask algoritmen er. O(n) notasjon kan altså være $O(n),\ O(n^2), O(\log n), O(n \log n), \ldots$.

Måling av effektivitet

Vi vil gjøre målinger på en rask prosessor, og det kan hende vi må bruke nanosekunder for å få registert verdier. Eksemplet nedenfor viser hvordan dette kan gjøres med datatypen chrono::nanoseconds. Nedenfor gjøres kun en tidtaking, ingen sortering av noe som helst. Men det burde være greit å forstå hvordan man kan erstatte noe kode nedenfor med aktuell kode.

Listing 2.2: chrono_eks.cpp

```
// Tidtaking med chrono

#include <iostream>
#include <chrono>
using namespace std;

void sort_tid_0(unsigned long ul) // kun tidtakings-kode
{
    chrono::nanoseconds total_tid{0};

    // tilordne random verdier
    auto start = std::chrono::high_resolution_clock::now();

    // plassholder for sorteringsfunksjon
    for (auto i=0; i<ul; i++)
        double x = 3.14 * 2.51;

    auto slutt = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> varighet = slutt-start;
    std::chrono::nanoseconds varighet_nano =
            std::chrono::duration_cast<std::chrono::nanoseconds >(varighet );
    total_tid += varighet_nano;

    // utskrift
    cout << total_tid.count() << endl;
}

int main()
{
    sort_tid_0(100000);
}
```

2.2 Binært søk

I en sortert array kan vi søke effektivt med binært søk. Gitt et tall x vi søker etter, begynner vi ikke på indeks 0 og sammenligner ett og ett tall med x. Vi sammenligner først med midterste element *midt*. Hvis $x < midt$, gjentar vi for venstre halvdel. Ellers gjentar vi for høyre halvdel. På den måten får vi sub-arrayer av lengde $n, \frac{n}{2}, \frac{n}{4}, \ldots$. Hvis for eksempel opprinnelig arraylengde er $n = 32$, blir sub-arrayenes lengde 16, 8, 4, 2. Vi trenger da kun å gjøre $\log_2 32 = 5$ gjentakelser, så binærsøk er $O(\log n)$.

Listing 2.3: binary_search.cpp

```
#include <iostream>
#include <iomanip>
#include <array>
#include <cmath>
#include <algorithm>

// implementering av selection sort for std::array
void selection_sort(std::array<int,20>& a) // husk referanse
{
    auto n = a.size();
    for (auto i=0; i<n-1; i++)
        for (auto j=i+1; j<n; j++)
            if (a[i] > a[j])
                std::swap(a[i], a[j]);
}

// implementering av binary search for std::array
int binary_search(std::array<int, 20>& a, int x)
{
    int indeks = -1;
    int n = static_cast<int>(a.size());
    int v{0}; int h{n-1};
    while (v<=h && indeks==-1)
    {
        auto midt = (v+h)/2;
        if (x == a[midt])
            indeks = midt;
        else if (x < a[midt])
            h = midt-1;
        else // x > a[midt]
            v = midt+1;
    }
    return indeks;
}

// eksempel paa binaersoek
int main()
{
    // Lager objekt med 20 random verdier
    std::array<int, 20> c;
    for (auto i=0; i<20; i++) c[i] = std::rand()%20;

    // Utskrift av usortert array
    std::cout << std::endl << "usortert array" << std::endl;
    for (auto i=0; i<c.size(); i++) std::cout << c[i] << " ";
    std::cout << std::endl;

    selection_sort(c);
    // Utskrift av sortert array
```

```
    std::cout << std::endl << "sortert array" << std::endl;
    for (auto i=0; i<c.size(); i++) std::cout << c[i] << " ";
    std::cout << std::endl;

    // Test

    // sortert array
    // 1 2 2 4 6 7 7 9 11 11 12 12 13 14 15 15 16 18 18 19
    // Soeke etter: 8

    for (;;)
    {
        std::cout << "Soeke etter: ";
        int tall;
        std::cin >> tall;
        int indeks = binary_search(c, tall);
        std::cout << indeks << std::endl;
        if (indeks == -1)
            break;
    }
    return 0;
}
```

2.2.1 Tidtaking av binærsøk

I eksemplet nedenfor har jeg kombinert binærsøkalgoritmen og tidtaking i et nytt eksempel. Dette eksemplet kan brukes til oppgaven nedenfor med små endringer (blant annet erstatte binary_search() med en sorteringsfunksjon).

Listing 2.4: binary_search2.cpp

```
// implementering av binary search for std::array
#include <iostream>
#include <iomanip>
//#include <array>
#include <cmath>
#include <algorithm>
#include <chrono>

// Oppretter en global konstant siden std::array krever konstant lengde
const unsigned long N{1000};
const unsigned long LOOPS{10000};

// For arrayer med store tall
unsigned long binary_search(std::array<unsigned long, N>& a, unsigned long x)
{
    unsigned long indeks = -1;
    unsigned long n = static_cast<unsigned long>(a.size());
    unsigned long v{0}; unsigned long h{n-1};
    while (v<=h && indeks==-1)
    {
        auto midt = (v+h)/2;
        if (x == a[midt])
            indeks = midt;
        else if (x < a[midt])
            h = midt-1;
        else // x > a[midt]
            v = midt+1;
    }
    return indeks;
}
```

```
int main ()
{
    std::array<unsigned long, N> a;
    for (unsigned long i=0; i<N; i++) a.at(i)=i;

    auto start = std::chrono::high_resolution_clock::now(); // Starter klokka

    for (auto i=0; i<LOOPS; i++)
    {
        unsigned long tall = std::rand()%N;
        auto indeks = binary_search(a, tall);
    }
    auto slutt = std::chrono::high_resolution_clock::now(); // Stopper klokka

    std::chrono::duration<double> varighet = slutt-start;
    std::chrono::nanoseconds varighet_nano =
            std::chrono::duration_cast<std::chrono::nanoseconds >(varighet );

    // utskrift
    std::cout << "n = " << N << ", repetisjoner= " << LOOPS << std::endl;
    std::cout << "varighet i nanosekunder: " << varighet_nano.count()/LOOPS << std::endl;

    return 0;
}
```

Her er en videoforklaring om tidtaking med chrono `https://youtu.be/PdgwA7uqJdw`

2.3 Oppgaver

Bruk kodeeksemplene fra dette kapitlet.

2.3.1

1. Lag noen arrayer med n tilfeldige tall, for eksempel n = 10, 100, 1000, 10000, 100 000.
2. Ta tiden på sortering av alle arrayene med selectionsort og noter.
3. Ta tiden på sortering av alle arrayene med std::sort og noter.
4. Gjenta punkt 1-3 10 ganger og regn ut gjennomsnitt for hver arraystørrelse og hver sorteringsfunksjon. Sett opp en tabell med ulike verdier av n og sorteringstid.
5. Marker verdiene fra tabellen på et ruteark (eller GeoGebra) og prøv å tilpasse en glatt kurve til dataene i hvert tilfelle.
6. Presenter resultatene i en LaTeXrapport.

Kapittel 3

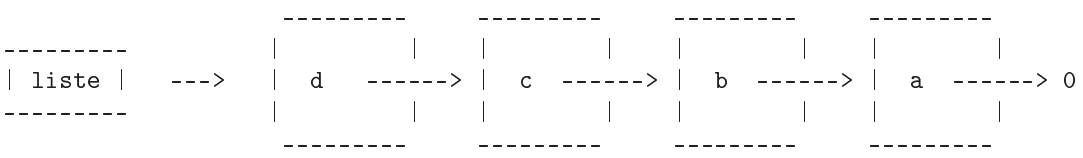
Rekursjon

3.1 Introduksjon

- rekursjon i liste
- $n!$ - fakultet
- binomialkoeffisientene
- Rekursiv forfining av polygoner

3.1.1 Rekursjon i liste

Vi husker fra CharNode klassen at vi lagde en liste med fire push, slik at den interne representasjonen ble slik:



Funksjonen

Listing 3.1: rekursiv postorder traversering

```
void CharNode::skrivBaklengs() const
{
    if (m_neste)
        m_neste->skrivBaklengs();
    std::cout << m_tegn;
}
```

skriver da ut tegnene i baklengs rekkefølge **abcd** når den kalles av første node

```
liste->skrivBaklengs();
```

Funksjonen er rekursiv. Måten den skriver ut på er et eksempel på **postorder** traversering. Vi kan skrive ut lista rekursivt forfra ved rekursiv **preorder** traversering:

Listing 3.2: rekursiv preorder traversering

```
void CharNode::skrivPreOrder() const
{
    std::cout << m_tegn;
    if (m_neste)
        m_neste->skrivBaklengs();
}
```

Men hvordan kan vi gjøre dette ikke-rekursivt?

Listing 3.3: postorder traversering med stakk

```
// rekursiv - iterativ
// Hvordan skrive enkel lenket liste baklengs uten rekursjon?
void CharNode*::skrivMedStakk()
{
    CharNode* neste = this;
    while (neste != nullptr)
    {
        m_stakk.push(neste->m_tegn);
        neste = neste->hentNeste();

    }
    while (!m_stakk.empty())
    {
        std::cout << m_stakk.top();
        m_stakk.pop();
    }
}
```

3.1.2 ! - fakultet

Vi tar med litt repetisjon om fakultetsfunksjonen, som er definert ved

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

med $0! = 1$ som et spesialtilfelle. Disse tallene dukker opp i mange sammenhenger. En funksjon som beregner $n!$ er funksjonen *fakultet()* nedenfor.

Vi ser av definisjonen at $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) = (n-1)!$, og det leder oss til en rekursiv definisjon:

$$n! = n \cdot (n-1)!$$

fortsatt med $0! = 1$ som et spesialtilfelle.Funksjonen fakrek nedenfor beregner $n!$ rekursivt.

Binomialkoeffisientene er kanskje ikke like kjent som fakultet. Binomialkoeffisientene er definert ved

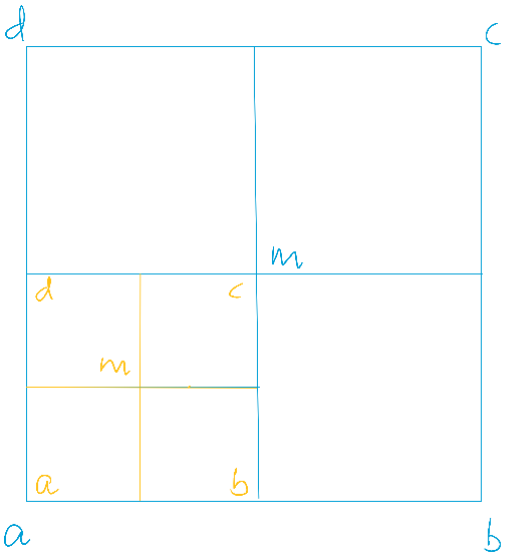
$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{1 \cdot 2 \ldots \cdot n}{1 \cdot 2 \ldots \cdot r \cdot 1 \cdot 2 \ldots \cdot (n-r)} = \frac{(n-r+1) \ldots \cdot n}{1 \cdot 2 \ldots \cdot r}$$

Binomialkoeffisientene kan regnes ut ved å bruke fakultetsfunksjonen som vist nedenfor.

Listing 3.4: fakultet.cpp

```
#include <iostream>
using namespace std;

// Iterativ funksjon
int fakultet(int n)
{
    int nfak=1;
    for (auto i=1;i<=n;i++)
        nfak = nfak*i;
    return nfak;
}

// Rekursiv funksjon
int fakrek(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakrek(n-1);
}

// Binomialkoeffisienrte
int binom(int n, int r)
{
    return fakrek(n)/(fakrek(r)*fakrek(n-r));
}

int main()
{
    cout << "n: ";
    int n; cin >> n;
    cout << n << "! = " << fak(n) << endl;
    int r;
    cout << "n r: ";
    cin >> n >> r;
    cout << "binom(" << n << " " << r << ") = " << binom(n, r) << endl;
    return 0;
}
```

Bruk heller unsigned long!



Figur 3.1: kvadrat forfining

3.1.3 Polygon forfining

I spill-scener kan vi ha behov for å dele opp et rektangulært eller triangulært område i mindre deler, og organisere objektene i et spill etter geometrisk posisjon. Quadtrær er mye brukt til dette, se for eksempel
`http://gameprogrammingpatterns.com/spatial-partition.html`
og `http://gameprogrammingpatterns.com/spatial-partition.html#design-decisions`.
Vi har ikke studert tre-strukturer ennå, men vi skal se hvordan rekursjon kan brukes til oppdeling av et rektangulært område.

Rektangel forfining

Et rektangel i 2d har fire hjørner, så det er naturlig å lage en klasse med fire hjørner som objektvariable. Vi trenger en konstruktør, vi skal ha en rekursiv funksjon **subDivide()** for å dele opp i mindre rektangler , og så må vi ha en funksjon for å lagre koordinatene til de rektanglene vi ender opp med. Vi lager også en funksjon for å skrive ut koordinatene. Det er enda morsommere å lage en grafisk løsning, men det må vi vente litt med.

Listing 3.5: PlainRectangle

```
#ifndef PLAINRECTANGLE_H
#define PLAINRECTANGLE_H
#include <vector>

class PlainRectangle
{
private:
    Vector2d m_v1;
    Vector2d m_v2;
    Vector2d m_v3;
    Vector2d m_v4;
```

```
    static std::vector<PlainRectangle> m_rectangles;
public:
    PlainRectangle();
    PlainRectangle(const Vector2d &v1, const Vector2d &v2, const Vector2d &v3, const Vector2d
    void subDivide(const Vector2d &a, const Vector2d &b, const Vector2d &c, const Vector2d& d
    void makeRectangle(const Vector2d& v1, const Vector2d& v2, const Vector2d& v3, const Vect
    void print() const;
};
#endif // PLAINRECTANGLE_H
```

Hvert rektangel skal deles opp i fire like store rektangler. Dette er vist på figur 3.1. Vi finner midtpunktet på hver side, og midtpunktet i rektanglet. Av disse punktene lager vi fire nye rektangler av samme størrelse. Vi bruker hjelpeklassen **Vector2d** til dette. Matematikken kommer vi tilbake til senere.

Listing 3.6: Vector2d

```
struct Vector2d {
    double x;
    double y;
    Vector2d operator + (const Vector2d& v) const
    {
            Vector2d u;
            u.x = x + v.x;
            u.y = y + v.y;
            return u;
    }
    Vector2d operator / (int d)
    {
        Vector2d u;
        u.x = x/d;
        u.y = y/d;
        return u;
    }
};
```

Implementering av rekursiv forfining

Nedenfor er hele implementeringen for klassen *PlainRectangle*. Det er den rekursive funksjonen *subDivide()* som er mest interessant her. Den har en parameter n som initielt sier hvor mange nivåer vi skal ha på oppdelingen. Denne parameteren dekrementeres i hvert rekursjonssteg, og når den når 0 kalles *makeRectangle()*.

Hvis $n > 0$ skal rektanglet deles opp i henhold til figuren. Vi lager fem nye punkter v1, v2, v3, v4, m i henhold til figuren. Vi bruker positiv omløpsretning på opplistingen av disse punktene i nytt rekursivt kall. Kontroller at dette stemmer med figur 3.1.

Listing 3.7: PlainRectangle.cpp

```
#include <iostream>
#include "plainrectangle.h"

std::vector<PlainRectangle> PlainRectangle::m_rectangles;

PlainRectangle::PlainRectangle() //: PlainTriangle ()
{

}
```

```
PlainRectangle::PlainRectangle(const Vector2d &v1, const Vector2d &v2, const Vector2d &v3, co
    : m_v1{v1}, m_v2{v2}, m_v3{v3}, m_v4{v4}
{

}

void PlainRectangle::subDivide(const Vector2d &a, const Vector2d &b, const Vector2d &c, const
{
    if (n>0) {
        Vector2d v1 = (a+b)/2;
        Vector2d v2 = (b+c)/2;
        Vector2d v3 = (c+d)/2;
        Vector2d v4 = (d+a)/2;
        Vector2d m = (a+c)/2;
        subDivide(a, v1, m, v4, n-1);
        subDivide(v1, b, v2, m, n-1);
        subDivide(m, v2, c, v3, n-1);
        subDivide(v4, m, v3, d, n-1);
    } else {
        makeRectangle(a, b, c, d);
    }
}

void PlainRectangle::makeRectangle(const Vector2d &v1, const Vector2d &v2, const Vector2d &v3
{
    m_rectangles.push_back(PlainRectangle(v1, v2, v3, v4));

}

void PlainRectangle::print() const
{
    std::cout << "n = " << m_rectangles.size() << std::endl;
    for (auto it=m_rectangles.begin(); it!=m_rectangles.end(); it++)
    {
        auto t = *it;
        std::cout << "(" << t.m_v1.x << ", " << t.m_v1.y << "), ";
        std::cout << "(" << t.m_v2.x << ", " << t.m_v2.y << "), ";
        std::cout << "(" << t.m_v3.x << ", " << t.m_v3.y << ") ";
        std::cout << "(" << t.m_v4.x << ", " << t.m_v4.y << ") ";
        std::cout << std::endl;
    }
}
```

Til slutt kan vi jo lage et kvadrat, og teste at vi får delt det opp i to nivåer:

Listing 3.8: PlainRectangle test

```
int main()
{
    Vector2d a{0, 0};
    Vector2d b{1, 0};
    Vector2d c{1, 1};
    Vector2d d{0, 1};

    PlainRectangle rect{a, b, c, d};
    rect.subDivide(a, b, c, d, 2);
    rect.print();
    return 0;
}
```

3.2 Oppgaver

3.2.1

Modifiser eksemplet ovenfor slik at opprinnelig rektangel/kvadrat blir lagret først i std::vector objektet.

3.2.2

Det er noen ganger interessant å ha en finere oppdeling i enkelte områder. For eksempel kan man tenke seg at man fortsetter oppdelingen av et rektangel som inneholder et gitt punkt P_0 helt til arealet av (det oppdelte) rektangelet som inneholder P_0 er mindre enn et forhåndsbestemt areal A_0. Implementer en slik løsning.

3.2.3

Implementer et tilsvarende prosjekt som deler opp trekanter i stedet for rektangler.

Kapittel 4

Binære trær

4.1 Egenskaper for et binært tre

4.1.1 Rekursivitet

Rekursivitet: Et binært tre er en datastruktur som er av rekursiv natur. Et binært tre er enten en tom mengde eller det består av tre deler (delmengder). Roten (her: toppen) er første del. Venstre og høyre subtre, som selv er binære trær, utgjør andre og tredje del. Vi kaller elementene i treet for noder, og skiller mellom indre noder og ytre noder. En indre node har ett eller to subtrær (barn). En ytre node har ingen subtrær og kalles et blad. Bladene eller løvet er altså nodene nederst i treet. Veien oppover i treet fra en node (til moren, faren) er entydig.

Enhver node har et nivå l, definert ved:

- Roten har nivå $l = 0$.
- Nivået til alle andre noder er $l = 1 +$ nivået til moren.

Et binært tre har en dybde d, definert ved $d =$ maks nivå for et blad i treet.

Dybden til et binært tre er lengste vei fra rot til blad. Videre, siden hver node kan ha to barn, har vi

- m noder på nivå $l \Longrightarrow$ maks $2m$ noder på nivå l+1
- Maks 2^l noder på nivå l

4.1.2 Komplett binært tre

Et komplett binært tre av dybde d er strengt binært med alt løvet (og bare løv) på nivå d. Et komplett binært tre av dybde d har

- 2^l noder på nivå l, $0 \leq l \leq d$
- Antall noder totalt $N = 2^{d+1} - 1$
- Herav 2^d blader og $2^d - 1$ indre noder

Sammenlignet med en lineær liste kan et binært tre ha mange noder, men kort vei fra rot til blad. (Et blad i et binært tre kan sammenlignes med siste node i en lineær liste.)

4.1.3 Nesten komplett binært tre

Et binært tre av dybde d er nesten komplett hvis

- Enhver node på nivå $l < d - 1$ har to barn, og
- Alt løvet ligger på nivå d eller $d - 1$, fylt opp fra venstre

(se [1] s.252 for en presis def. Definisjonene kan variere i ulike bøker).

4.1.4 Binært tre eksempel

Vi skal først se på en klasse som minner om CharNode klassen.

Listing 4.1: Binarynode.h

```
class BinaryNode
{
public:
    BinaryNode();
    BinaryNode(char data, BinaryNode* left=nullptr,
               BinaryNode* right=nullptr);
    char getData();
    void print();
    BinaryNode *find(char data);
    void insert(char data);
    void intrav();
private:
    char m_data;
    BinaryNode* m_left;
    BinaryNode* m_right;
};
```

Datatypene til objektvariablene her er som for en dobbelt-lenket liste, men vi skal bruke denne klassen til en helt annen datastruktur. Nedenfor følger implementeringen. Vi skal se på detaljene senere.

Listing 4.2: Binarynode.cpp

```
#include <iostream>
#include "binarynode.h"

BinaryNode::BinaryNode() { }
BinaryNode::BinaryNode(char data, BinaryNode* left,
            BinaryNode* right) :
    m_data{data}, m_left{left}, m_right{right}
{ }

char BinaryNode::getData() { return m_data; }

BinaryNode *BinaryNode::find(char data)
```

```
{
    //std::cout << m_data;
    if (m_data == data)
        return this;
    else if (data < m_data && m_left!=nullptr)
        return m_left->find(data);
    else if (m_right)  // m_data <= data
        return m_right->find(data);
    // kun hvis noden ikke finnes, kommer vi hit
    return nullptr;
}
void BinaryNode::insert(char data)
{
    if (data < m_data) {
        if (m_left)
            m_left->insert(data);
        else
            m_left = new BinaryNode(data);
    }
    else if (data > m_data) {
        if (m_right)
            m_right->insert(data);
        else
            m_right = new BinaryNode(data);
    }
}
void BinaryNode::intrav()
{
   if (m_left)
       m_left->intrav();
   std::cout << m_data;
   if (m_right)
       m_right->intrav();
}
```

I eksemplet nedenfor bygger vi et binært tre manuelt, kun ved å bruke parametrisk konstruktør. Utskriften viser at dette faktisk blir et binært søketre. Tegn steg for steg hvordan treet bygges!

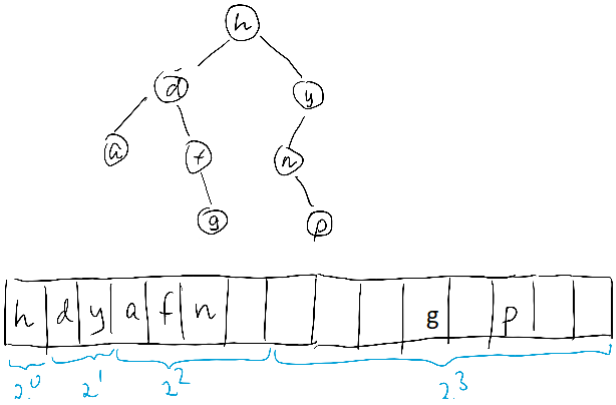
Listing 4.3: Binært tre testprogram

```
int main(int argc, char *argv[])
{
    BinaryNode* h = new BinaryNode('d');
    BinaryNode* v = new BinaryNode('a');
    v = new BinaryNode('b', v, h);
    // peker h er ledig
    BinaryNode* btre = new BinaryNode('g');
    h = new BinaryNode('p');
    btre = new BinaryNode('i', btre, h);
    btre = new BinaryNode('f', v, btre);

    btre->intrav();
}
```

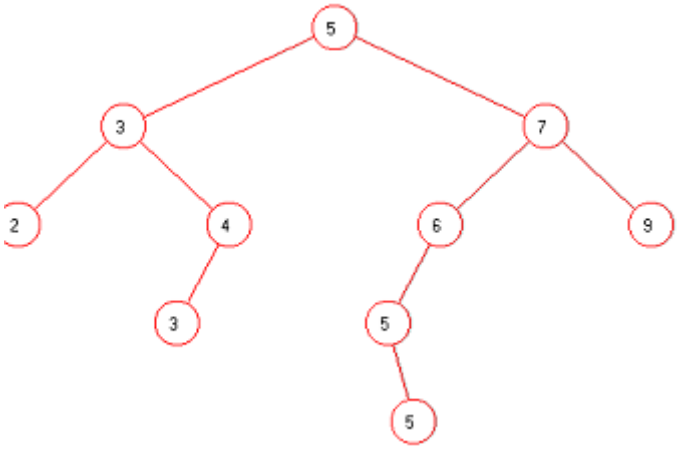
4.1.5 Definisjon av binært søketre

Et binært søketre er et binært tre hvor

- Alle nodene i det venstre subtreet til en node n er mindre enn innholdet av n



Figur 4.1:



Figur 4.2:

- Alle nodene i det høyre subtreet til en node n er større eller lik innholdet av n

Her må vi også ha et kriterium for sammenligning av noder. Dette er enkelt hvis noden inneholder data av typen int, char, double etc. Når vi lager et tre med en egendefinert datatype, må vi overlaste sammenligningsoperatoren(e) for denne datatypen, eller legge til en ekstra template parameter.

4.1.6 Dynamisk representasjon

Et binært tre kan lagres i statisk eller dynamisk minne, som andre datastrukturer. Vi skal først studere dynamisk representasjon.

4.1.7 Datastruktur

Vi benytter en enklest mulig klasse BinaryNode for datatypen char. Det er fint mulig å gjøre om denne til en template klasse, som for lenket liste eksemplet. I C++ standard template library fins det selvsagt template klasser som vi kan bruke. Hvis vi velger å bruke dynamisk minneallokering, kan en template klasse se slik ut:

```
class BinaryNode
{
private:
    char m_data;
    BinaryNode* m_left;
    BinaryNode* m_right;
};
```

4.1.8 Søking i binært søketre

figur 4.1 viser et binært søketre tegnet både som et tre (dynamisk representasjon) og array representasjonen av det samme treet. Søking gjøres ved hjelp av definisjonen avsnitt 4.1.5. Vi sammenligner først roten med søkeverdien, deretter gjentas sammenligningen for subtrær helt til vi eventuelt har funnet noden vi søker etter. Funksjonen returnerer en peker til den aktuelle noden dersom den fins, ellers en nullptr. Funksjonen har en ut-parameter til over-node for bruk ved innsetting. En ikke-rekursiv algoritme er beskrevet i [1].

```
BinaryNode *BinaryNode::find(char data)
{
    //std::cout << m_data;
    if (m_data == data)
        return this;
    else if (data < m_data && m_left!=nullptr)
        return m_left->find(data);
    else if (m_right)  // m_data <= data
        return m_right->find(data);
    // kun hvis noden ikke finnes, kommer vi hit
    return nullptr;
}
```

Dersom vi søker etter en verdi som fins flere ganger i treet, hvilken nodepeker returneres da? Vi sammenligner algoritmen med figur 4.2 og tenker oss at vi søker etter en node med verdien 3. Algoritmen vil da returnere en peker til den første/øverste 3-noden i treet. Riktignok er koden ovenfor for et tre med char data (key/nøkkel), mens treet på figuren har int data (key/nøkkel). Men det er enkelt å bytte ut char med int eller å gjøre om til template klasse). Med denne algoritmen krever søk i treet O (log n) operasjoner. O-notasjonen betyr i størrelsesorden. Hvor bra er dette sammenlignet med søking i usortert og sortert array?

4.1.9 Innsetting

En node som settes inn i et binært søketre, settes alltid inn som et blad. Dette gjelder også ved innsetting i et tomt tre, da blir inn-noden rot og den eneste noden. Innsettingen gjøres ved hjelp av avsnitt 4.1.5. Vi sammenligner først roten med inn-noden, deretter gjentas sammenligningen for subtrær helt til vi har funnet riktig plass. Legg merke til i denne implementeringen at noden som skal settes må være opprettet, og at funksjonen må kalles av roten. Vi får

```
void BinaryNode::insert(char data)
{
    if (data < m_data) {
        if (m_left)
            m_left->insert(data);
        else
            m_left = new BinaryNode(data);
    }
    else if (data > m_data) {
        if (m_right)
            m_right->insert(data);
        else
            m_right = new BinaryNode(data);
    }
}
```

Vi ser på to innsettings-eksempler.

Eksempel

Vi skal sette inn en node med en verdi som ikke fins i treet (eks. 1 i treet på figur 4.2). Vi ser at første **if** blir true for roten 5, 3 og 2. Andre **if** blir true for 5 og 3. Noden 2 har ikke noe venstrebarn, og else-delen utføres - den nye noden settes inn.

Eksempel

Hva med innsetting av en node med en verdi som forekommer i treet? Vi kan kreve at et binært søketre har unike nøkkelverdier (data), slik som i std::set. Eller vi kan tillate duplikater som i std::multiset. Vi illustrerer også dette med et eksempel fra figur 4.2). Anta at vi skal bruke funksjonen insert() til å sette inn en enda node med verdien 3.

Det er fortsatt slik at roten i treet kaller insert(). Vi ser if (data < m_data) blir true første gangen, men false andre gang (steg nummer to i rekursjonen). Hva skjer videre? else if - delen blir ikke utført. Funksjonen gjør ingenting. Vi kan altså ikke bruke denne insettingsfunksjonen til et tre hvor duplikater er tillatt, uten å gjøre endringer.

4.1.10 Traversering

Vi skiller mellom dybde-først traversering og bredde-først traversering. Dybde-først traverseringer er preorder, inorder og postorder. En rekursiv algoritme for preorder traversering er

Algoritme preorder rekursiv traversering

1. Besøk noden
2. Traverser venstre subtre preorder
3. Traverser høyre subtre preorder

Å besøke er her et abstrakt begrep. Det kan være noe så enkelt som å skrive ut noden, eller vi kan være interessert i å gjøre noe annet når vi besøker den. Så, i hvilken rekkefølge vil nodene bli besøkt ved preorder traversering?

Vi ser av algoritmen og figur 4.2 at de første tre nodene som blir besøkt er 5, 3, 2. De to første linjene i algoritmen gjentas helt til vi når et blad – dybde-først traversering. Når vi når bladet 2, er de to første linjene utført for nodene 5, 3 og 2. Det neste som skjer, er traversering av høyre subtre til noden 3 (siden 2-er noden er et blad). Deretter traverseres noden 4 preorder, før høyre subtre til roten traverseres preorder.

Vi har også

Algoritme inorder rekursiv traversering

1. Traverser venstre subtre preorder
2. Besøk noden
3. Traverser høyre subtre preorder

Algoritme postorder rekursiv traversering

1. Traverser venstre subtre preorder
2. Traverser høyre subtre preorder
3. Besøk noden

4.1.11 Traversering anvendelser

Det er lett å se at treet på figur 4.2 gir en sortering av nodene ved inorder traversering. Dette gjelder generelt, og sortering ved hjelp av binært søketre er vanligvis meget effektivt.

4.1.12 Ikke-rekursiv traversering

Vi skal se på preorder traversering av treet på figur 4.2. Av rekkefølgen for de tre første nodene (5, 3, 2) kan vi foreslå følgende start på algoritmen (vi bruker en nodepeker p til traverseringen):

```
p = rot;
while (p != nullptr) {
   besøk p;
   p = p- >vsub;
}
```

Men dette blir jo som traverseringen av en enkel lenket liste, og vi får kun traversert en del av treet. Den fjerde noden som skal besøkes er 4 (hsub til 3). Når while-løkka terminerer må vi derfor gå oppover i treet igjen. Men da kan vi jo bruke en over-peker? Her løser ikke det problemet. Når løkka terminerer, er p lik nullptr, og ingen nullptr har noen over-peker. Legg merke til at vi i denne situasjonen har utført det samme som de to første linjene i rekursiv preorder traversering.

Vi kan prøve å bruke en ekstra hjelpepeker som ligger ett steg bak og gå oppover (mot roten) ved hjelp av en over-peker. Dette er en kjent metode fra traversering av lineære lister. Her skal vi se hvordan vi kan løse problemet ved å bruke en stakk.

Når vi besøker 5, 3, 2 på vei fra roten og nedover i treet, kan vi legge en peker til den aktuelle noden på toppen av en stakk. Når while-løkka terminerer, kan vi hente tilbake pekeren til den sist besøkte noden (et blad) fra stakken. Vi har igjen det som svarer til siste linje i den rekursive traverseringsalgoritmen, å traversere høyre subtre preorder. Men vi har jo klart å traversere de tre første nodene preorder! Altså skal vi gjenta while-løkka for det første høyre subtre vi finner på vei opp mot roten. Gitt en stakk med push(), pop() og empty() og nodepekere til de nodene vi har besøkt, gjør vi følgende etter while-løkka:

```
if (!stakk.empty())  {
   p = stakk.top();
   stakk.pop();
   p = p->hsub;
}
```

Det gjenstår nå bare å finne en stopp-betingelse. På vei ned fra roten (første to linjer i rekursiv algoritme) legges nodepekere på stakken, og while-løkka terminerer når vi har nådd et blad. På tilbakeveien tar vi en nodepeker fra stakken så lenge den ikke er tom og gjentar while-løkka for hsub til denne. På vei tilbake til roten fra det nederste høyre bladet poppes nodepekere fra stakken – men siden ingen av disse nodene har noen hsub, utføres ikke while-løkka (se figuren). Vi får følgende algoritme:

Algoritme ikke-rekursiv preorder traversering

```
p =rot;

do {
   while (p != nullptr) {
      // besøk p;
      // push p på stakk;
      p = p->vsub;
   }
   if (!stakk.empty()) {
      // pop p fra stakk;
      p = p->hsub;
   }
} while (!stakk.empty() || p != nullptr);
```

4.2 Bredde-først traversering

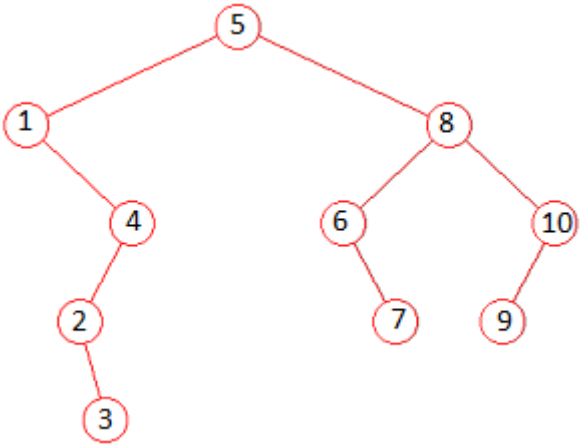
Ved bredde-først traversering skal vi besøke nodene nivåvis – først roten (nivå 0), deretter nodene på nivå 1, nodene på nivå 2 osv. Vi velger å besøke nodene på et gitt nivå fra venstre til høyre. Til denne traverseringen bruker vi en kø. Vi bruker figur 4.2 til hjelp igjen, og her er det ikke vanskelig å skrive ned traverseringsrekkefølgen.

Vi tenker oss at vi har besøkt noden 5 og setter vsub (3) og hsub(7) inn i en kø. Da vil fronten i køa være 3 – neste node som skal besøkes. Når vi har tatt ut 3 fra køa, vil fronten (og eneste element) i køa være 7 – neste node som skal besøkes. Nodene som skal besøkes etter 7, er vsub(2) og hsub(4) til 3. Hvis vi setter inn barna til en node bakerst i køa etter at vi har besøkt noden, så vil køa inneholde en riktig rekkefølge av de neste nodene som skal besøkes. Vi skal altså ta ut fronten til køa (en nodepeker), besøke noden, sette inn vsub og hsub bakerst i køa. Algoritmen blir:

Algoritme bredde-først traversering i binært tre

```
 p =rot;
sett inn p i kø;
while (!q.empty())  {
   ta ut p fra kø;
   besøk p;
   if (p->vsub != nullptr)
      sett inn p->vsub i kø;
   if (p->hsub != nullptr)
      sett inn p->hsub i kø;
}
```

Det første vi gjør, er altså å initialisere køa med en peker til roten.



Figur 4.3:

4.2.1 Sletting av en node

Innsetting er relativt enkelt, vi kan raskt søke etter riktig plass og sette inn en ny node som et blad. Sletting er litt mer komplisert. Når en node i treet slettes, må treet reorganiseres slik at det oppfyller Definisjon 1. Samtidig er det ønskelig at reorganiseringen av treet (dvs. å flytte pekere) gjøres enklest og raskest mulig. Vi bruker igjen figur 4.2 som eksempel.

Slette et blad

Et blad (eks. noden 2) kan slettes enkelt. Det har ingen subtrær, og alt vi trenger å gjøre er å slette noden og sette over-nodens vsub eller hsub lik nullptr. Når en indre node slettes, må en annen node overta den ledige plassen inkludert vsub-, hsub- og over-pekere.

Slette en indre node som ikke har et høyre subtre

Et eksempel på dette på figuren er å slette noden 6. Den er venstrebarn til 7, altså ekte mindre enn 7. Den har et venstre subtre (med 5 som rot), men ikke noe høyre subtre. Det fins altså ingen noder i venstre subtre til 7 som er større enn 5. Da kan vi slette noden og la venstre subtre (5) til ut-noden (6) bli nytt venstre subtre til ut-nodens mor (7).

Slette en indre node som har et høyre subtre

For å illustrere dette, bruker vi et nytt eksempel. Anta at vi skal slette den noden 8 på figur 4.3. Hvilken node skal vi erstatte denne med? I følge definisjonen på binært søketre trenger vi en node som er ekte større enn alle nodene i venstre subtre til ut-noden (8), og mindre eller lik nodene i høyre subtre. Det første kravet utelukker noder oppover i treet i forhold til ut-noden (for eksempel roten (5)) og noder i venstre subtre til ut-nodens mor. Det er lett å se ved hjelp av figur 4.3. Vi må altså lete etter en erstatter i ut-nodens venstre eller høyre subtre.

1. Venstrebarn (6) til 8 er utelukket. Hvorfor?
2. Høyrebarn (10) til 8 er utelukket. Hvorfor?
3. Høyrebarn (7) til venstrebarnet (6) er utelukket. Hvorfor?

Den eneste gjenstående kandidaten til å ta ut-noden (8) sin plass, er node 9. Og det går bra! (Prøv). Men hva med det generelle tilfellet? Har denne noden en spesiell egenskap slik at dette alltid går?

Svaret er heldigvis ja. Node 9 er ut-nodens inorder etterfølger, det vil si den etterfølgende noden til ut ved inorder traversering. Vi trenger en metode for å finne denne inorder etterfølgeren i det generelle tilfellet. For å illustrere dette, ser vi hva som skjer når vi skal slette 1-noden i treet.

Vi leter etter inorder etterfølger til 1 i denne nodens høyre subtre. Fra det høyre subtreet (4) går vi nedover til vsub gjentatte ganger helt til vi kommer til en node uten vsub. Dette svarer til første linje i inorder rekursiv traversering, og noden vi finner er 2. 3 er ekte mindre enn 2 sin mor (4), slik at den nye vsub til 4 kan bli 3.

Med den datastrukturen vi benytter her, får vi følgende algoritme for å finne inorder etterfølger og slette en node. Begge funksjonene kalles av noden som skal slettes.

Algoritme: finne inorder etterfølger

```
BinaryNode* BinaryNode::finnErstatter() {
   BinaryNode* etterfolger = nullptr;
   if (m_right) {
      etterfolger = m_right;
      while (etterfolger->m_left)
         etterfolger = etterfolger->m_left;
   }
   else
      etterfolger = m_left;
   return etterfolger;
}
```

Algoritme: Slette node i binært søketre

Algoritmen nedenfor forutsetter at hver node har en peker til moren/faren.

```
BinaryNode* BinaryNode::erstatt(BinaryNode* etterfolger) {
   if (m_right) {
      if (etterfolger->m_right)
         if (etterfolger != m_right)  // ut sin hsub.
            etterfolger->m_right->m_over = etterfolger->m_over;
         else
            etterfolger->m_right->m_over = m_over;
      if (etterfolger != m_right)  {
         etterfolger->m_over->m_left = etterfolger->m_right;
         etterfolger->m_right = m_right;
```

```
            m_right->m_over = etterfolger;
         }
         etterfolger->m_left = m_left;
         if (m_left)
            m_left->m_over = etterfolger;
      }
      if (etterfolger)
         etterfolger->m_over = m_over;
      if (m_over) {
         if (is_left())
            m_over->m_left = etterfolger;
         else
            m_over->m_right = etterfolger;
      }
      return this;
   }
```

4.2.2 Array representasjon

Vi har sett at vi kan implementere en stakk og en kø enten som lenket liste eller ved å bruke array. Slik er det også med binære trær. Dette står omtalt i [1] side 265-270. Vi skal se på hovedideene i denne implementeringen. Husk at nå snakker vi ikke nødvendigvis om binært søketre!

Vi utvider treet til et nesten komplett binært tre, og bruker formlene for antall noder på hvert nivå til å finne vsub, hsub og over til en aktuell node. Vi bruker igjen figur 4.2 som eksempel. Dette treet kan lagres i en array slik:

Nivå																									
0	***1***		***2***				***3***								***4***										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20					
5	3	7	2	4	**6**	9	-1	-1	3	-1	**5**	**-1**	-1	-1	-1	-1	-1	-1	-1	5					

Her svarer verdien –1 til en nullptr i dynamisk representasjon. Treet forutsettes altså ikke å inneholde noder med verdien –1. En slik utenkelig verdi kalles gjerne en sentinell verdi.

Vi observerer følgende:

- roten (5) med indeks 0 har vsub (3) på indeks 2 * 0 + 1 = 1 og hsub (7) på indeks 2 * 0 + 2 = 2.
- noden (3) med indeks 1 har vsub (2) på indeks 2 * 1 + 1 = 3 og hsub (4) på indeks 2 * 1 + 2 = 4.
- noden (6) med indeks 5 har vsub (5) på indeks 2 * 5 + 1 = 11 og ingen hsub på indeks 2 * 5 + 2 = 12.

Formlene for å finne indeksene til vsub og hsub til en gitt node med indeks i er altså

$$vsub(i) = 2 \cdot i + 1 \; og \; hsub(i) = 2 \cdot i + 2$$

Hvordan finne indeksen til moren for en gitt node? Vi ser på de uthevete nodene på figuren og løser ligningen over med hensyn på *i*. vsub(i) og hsub(i) er i dette tilfellet kjente indekser (11 og 12).

For noden 5 på indeks 11 blir i = (11– 1)/2 = 5. Siden vi bruker heltall til indekser, avrundes dette til 5. Dersom det var en hsub på indeks 12, ville vi få i = (12– 2)/2 = 5. Dette stemmer jo. Men hvordan kan vi vite om en node er vsub eller hsub?

Det trenger vi ikke vite! Siden vi bruker heltall til indekser, blir (12– 1)/2 = 5.5 avrundet til 5. Formelen blir altså lik enten noden er vsub eller hsub:

$$over(i) = \frac{i-1}{2}$$

Vi skal ikke implementere binære søketrær ved hjelp av arrayer her. men vi har faktisk allerede gjort noe som ligner veldig i avsnitt 3.1.3.

4.2.3 Binærsøk i array

Søking i binære søketrær har mye til felles med binærsøk i en sortert tabell/array. Vi nevner det her, slik at ingen skal tro at man må lage kompliserte dynamiske trestrukturer for å søke effektivt. Nedenfor er en sortert tabell med indeksene i midterste rekke. Anta at vi her søker etter verdien 8 i den sorterte

					↓2		↓3			↓1										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	2	3	3	5	6	7	**8**	8	9	11	12	12	13	15	17	19	21	21	22	23

arrayen. Binærsøk foregår ved at man først sammenligner midterste element i arrayen med søkeverdien. Er søkeverdien lik verdien av dette elementet, er man ferdig. Er søkeverdien mindre enn dette elementet, gjentas prosessen for venstre subarray, ellers for høyre subarray. Dette fortsetter helt til man har funnet søkeverdien eller det ikke er igjen flere elementer å undersøke. Siden vi kan halvere antall elementer i subarrayen for hvert steg, er denne metoden like effektiv som binærtre søk.

4.3 Kort sammendrag

- Søking og innsetting av en node krever $O(\log n)$ operasjoner for et tre med n noder. Å konstruere hele treet krever $O(n \log n)$ operasjoner.
- En node som settes inn i et binært søketre, blir alltid et blad.
- Et binært søketre kan brukes til effektiv sortering ved hjelp av inorder traversering.
- En node som slettes fra et binært søketre, erstattes av inorder etterfølger.
- Vi har to hovedtyper traversering: Dybde først og bredde først. Dette gjelder også for mer kompliserte datastrukturer enn binære søketrær.
- Et binært søketre bør være mest mulig balansert. Hva betyr det?
- Noen ganger kan binære søketrær bli veldig dype (ha mange nivåer i forhold til antall noder). Et worst case er å konstruere et binært søketre av en sortert liste. Da får vi et tre tilsvarende en lineær liste!
- `https://en.wikipedia.org/wiki/Binary_tree`

Balansering

Det fins ulike måter å balansere et binært søketre. Noen av de mest kjente er AVL-tre (se [2], 4.4) og red-black trees ([2], 12.2). Vi skal se litt på dette i kapittel 5. Her er noen ideer: `https://youtu.be/_4eQs8fsb5I`.

4.4 Oppgaver

4.4.1

List opp minst to ulike innsettings-rekkefølger for de binære søketrærne på figur 4.1 og figur 4.2.

4.4.2

Skriv opp rekkefølgen ved preorder traversering av de binære søketrærne på figur 4.1 og figur 4.2.

4.4.3

Hvilke noder i treet på figur 4.1 er indre noder, og hvilke er blader? Hvor mange noder er det på hvert nivå? Hva er dybden d til treet? Hvor mange noder er det plass til i et binært tre med dybde d?

4.4.4

Skriv opp rekkefølgen ved postorder traversering av de binære søketrærne på figur 4.1 og figur 4.2.

4.4.5

Skriv og test en ikke-rekursiv inorder traversering for binært tre.

4.4.6

Skriv og test en ikke-rekursiv postorder traversering for binært tre.

4.4.7

Skriv en funksjon som returnerer antall noder i et binært søketre.

4.4.8

Skriv en funksjon som returnerer antall nivåer i et binært søketre.

4.4.9

Et binært søketre kan sies å være balansert dersom det ikke er større nivåforskjell enn 1 mellom venstre og høyre subtre. Skriv en funksjon som finner ut om et binært søketre balansert.

4.4.10

Lag et testprogram for oppgavene ovenfor.

4.4.11

Endre insert i avsnitt 4.1.9 slik at det går an å sette inn duplikater.

Kapittel 5

Trær II

5.1 Introduksjon

Vi skal se litt på noen andre trestrukturer, men også mer på binære trær. Det fins ulike måter å balansere et binært søketre. Noen av de mest kjente er AVL-tre (se [2], 4.4) og red-black trees ([2], 12.2). Se også [3] kapittel 15 og Wikipedia lenker nedenfor. Vi nøyer oss her med praktiske eksempler på innsetting i Red-black tre og 2-3-4 tre og går ikke inn på implementering av innsetting og sletting av noder i disse trærne.

5.2 Avl-tre

Et AVL-tre er et binært søketre med balansering ([2], kapittel 4.4). Kravet til balansering er at differansen mellom høyden på venstre og høyre subtre er maksimalt 1. Dette må man ta hensyn til ved innsetting og sletting. For å reorganisere treet ved innsetting brukes rotasjoner. Et fornuftig kriterium for å balansere binært søketre er at venstre og høyre subtre skal ha en nivåforskjell på maksimalt 1. Et binært søketre som tilfredsstiller dette kriteriet, kalles et AVL-tre. Høyden til et AVL-tre er $\approx O(\log N)$.

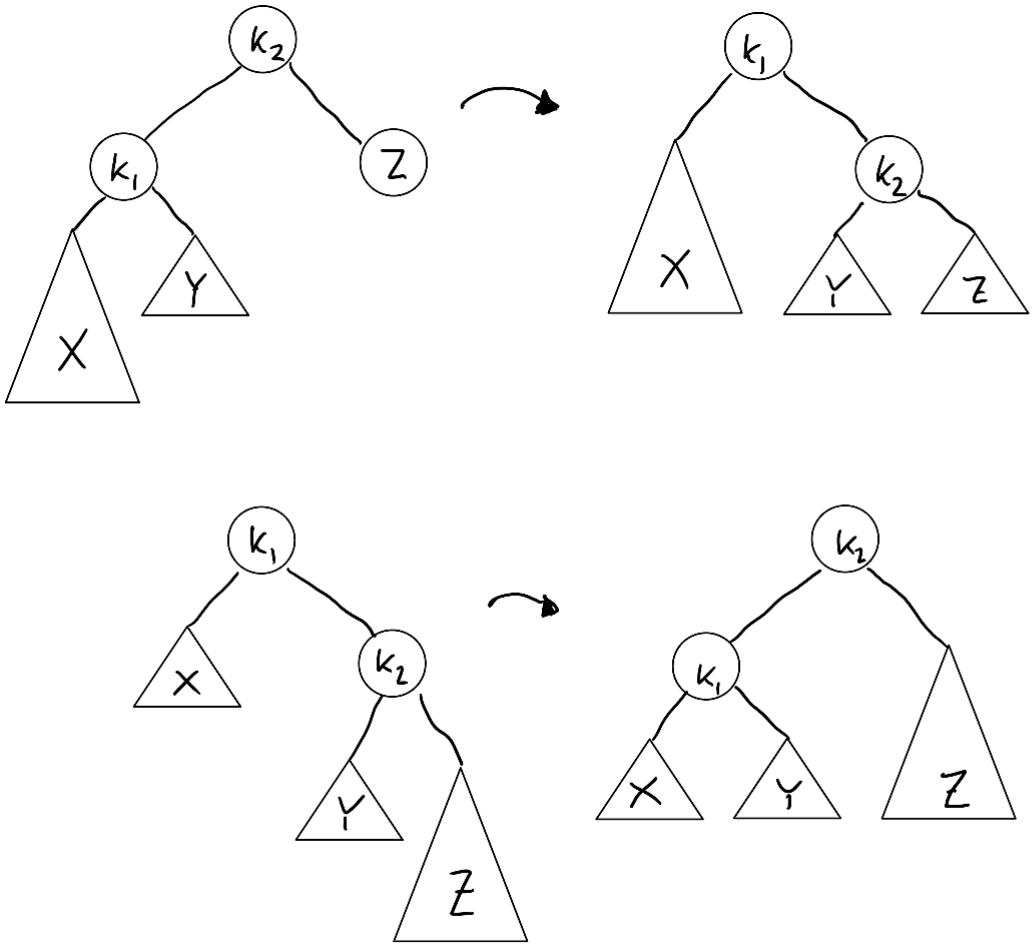
Det vi må gjøre for å holde på et AVL-tre, er å balansere treet etter hver innsetting. Hvis nødvendig. Dette gjøres ved rotasjoner, og vi skiller mellom enkel og dobbel rotasjon. Her følger to konkrete eksempler.

Hvis vi ser på 5.3, er subtreet med rot 3 ikke et AVL-tre etter at 4 er satt inn. Nodene 3, 6 og 4 må reorganiseres slik at vil får et AVL-(sub)tre av disse nodene. Hvordan kan vi gjøre dette (med rotasjoner)?

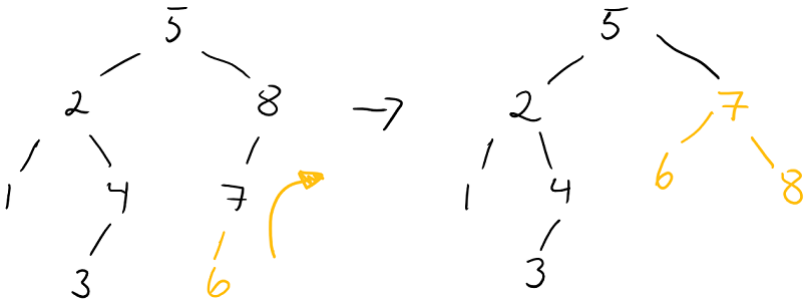
Før vi svarer på spørsmålet, la oss liste opp de fire tilfellene vi har ([2], 4.4):

1. innsetting i venstre subtre til venstre barn
2. innsetting i høyre subtre til venstre barn
3. innsetting i venstre subtre til høyre barn
4. innsetting i høyre subtre til høyre barn

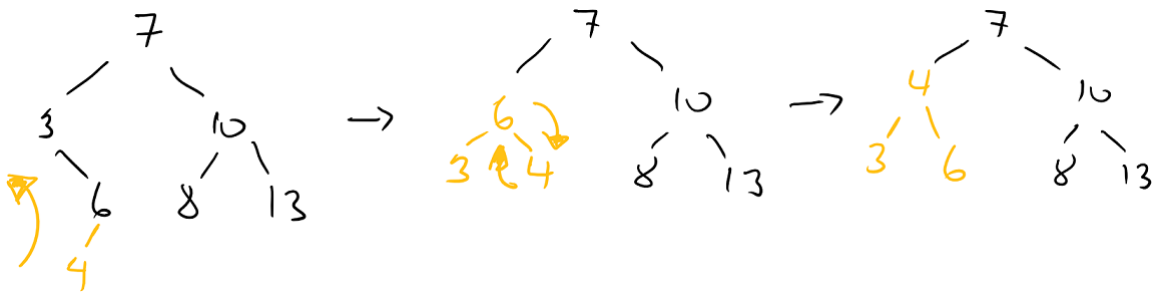
Tilfelle 1 og 4 er symmetriske og løses med enkel rotasjon. Tilfelle 2 og 3 er også symmetriske og løses med dobbelt rotasjon.



Figur 5.1: Singel rotasjon (fra [2], kapittel 4.4).



Figur 5.2: Binært søketre, ikke balansert. Enkel rotasjon.



Figur 5.3: Dobbelt rotasjon.

5.2.1 Red-black tree

Mye av framstillingen her er basert på og hentet fra [2], 12.2 og [3], kapittel 15. Et red-black tre er et binært søketre med følgende tilleggsegenskaper:

1. Hver node er farget enten rød eller svart.
2. Roten er svart.
3. Hvis en node er rød, må barna være svarte.
4. Hver sti fra en node til en nullptr må inneholde samme antall svarte noder.

Disse egenskapene tilsammen skal gi et balansert tre. Prisen vi må betale, er som for AVL-tre en mer komplisert innsetting og sletting enn for et enkelt binært søketre. En ny node settes som vanlig inn som et blad, men vi reorganiserer etterpå treet i hovedsak ved hjelp av rotasjoner og farge-endringer.

Rotasjoner

Rotasjoner kan være komplisert. På figur 5.1 er det vist to singelrotasjoner. Nodene som inngår i rotasjonene er kalt k_1 og k_2 som i [2], kapittel 4.4. X, Y og Z representerer subtrær. Høyden på trekantene iillustrerer at et subtre har fått økt dybden etter innsetting av ny node. Formålet med rotasjonene er å balansere treet igjen. I andre tilfeller er det nødvendig med dobbelt rotasjoner.

Legg merke til at $k_1 < Y < k_2$ i begge rotasjonene. Det er de ytre nodene som roterer, og Y skifter side i forhold til roten.

5.2.2 Eksempel - innsetting i red-black tre.

Vi skal se på et praktisk eksempel, uten å gå gjennom hele teorien for red-black trees (se for eksempel [2], kapittel 12). Det binære søketreet på figur 4.3 kan vel sies å være noe glissent, på godt norsk. Det har fem nivåer, og plass til

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 = \Sigma_{i=0}^{4}\, 2^i = 31$$

noder. Siden det inneholder 10 noder, blir fyllingsgraden $\frac{10}{31} = 0.32$. Det har omtrent like mange noder på hver side av roten, men for eksempel subtreet med rot 1 er ekstremt ubalansert med 0 nivåer i venstre subtre og tre nivåer i høyre

subtre. Med balansering prøver vi å oppnå like lang sti til hvert blad, eler heller like kort sti. Dette er for at søketiden skal være $O(\log n)$.

En innsettingsrekkefølge for treet på figur 4.3 kan for eksempel ha vært

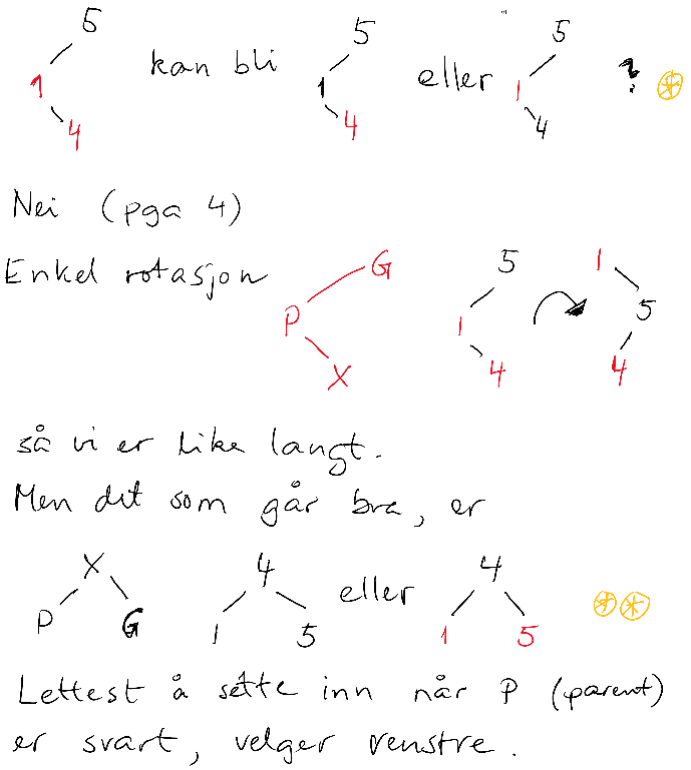
5, 1, 4, 8, 6, 10, 2, 7, 9, 3.

(Hvis rekkefølgen hadde vært 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ville vi ha fått en lenket liste av m_right pekerne).

Når vi setter inn en ny node, må den farges rød. Hvis ikke, bryter vi egenskap 4 (siden treet vi setter inn i allerede skal være et red-black tre). Hvis moren er svart, er innsettingen ok. Hvis moren er rød, brytes egenskap 3 og vi må foreta oss noe. Så la oss se hvordan vi kan sette inn noen noder i et red-black tre.

5,1,4

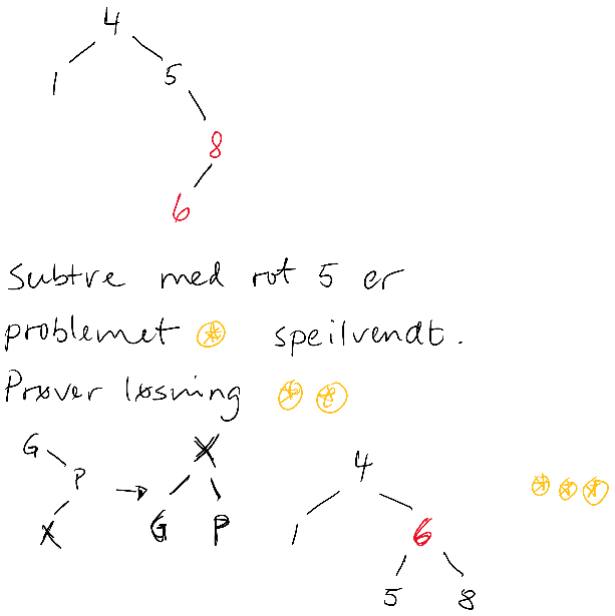
5 settes inn som bladnode og farges dermed rød. Siden treet er tomt, blir 5 en rot og vi må endre farge til svart. 1 settes inn som bladnode og farges rød. Siden moren er svart, er vi ferdige. 4 settes inn som bladnode til 1 og farges rød. Men moren er rød, så den må bli svart. Dette bryter egenskap 4. Vi kan prøve med en enkel rotasjon, som gir rot 1 (rød), høyre 5 (svart), venstre 4 (rød). Dette bryter med egenskap 2. Det eneste som er mulig, er å la 4 bli rot (svart). Da kan 1 venstre (rød) og 5 høyre (rød), eller vi kan farge bladene svarte. Siden det er enklest å sette inn når P (parent) er svart, velger vi dette (se figur 5.4).



Figur 5.4:

8,6

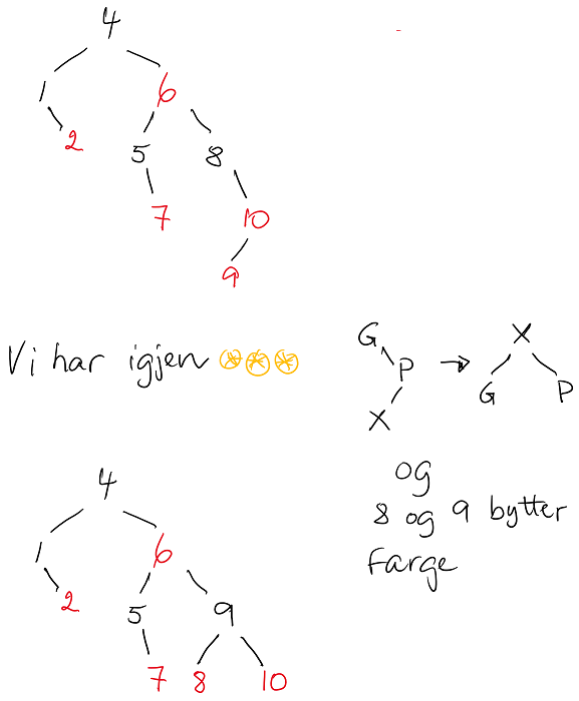
Den neste som settes inn er 8. Den blir høyre til 5 (som er svart), og rød fordi den er et blad. Deretter setter vi inn 6 som blad til 8, og vi har to røde. (se figur 5.5).



Figur 5.5:

10,2,7,9

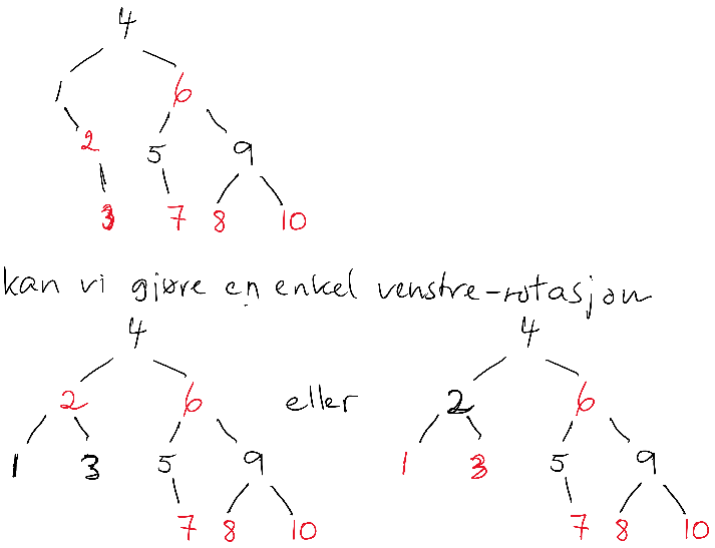
De neste tre (10, 2, 7) settes inn enkelt. Så følger 9. Vi har igjen situasjonen (***) som kan løses med enkel venstrerotasjon og fargeendring på 8 og 9.



Figur 5.6:

3

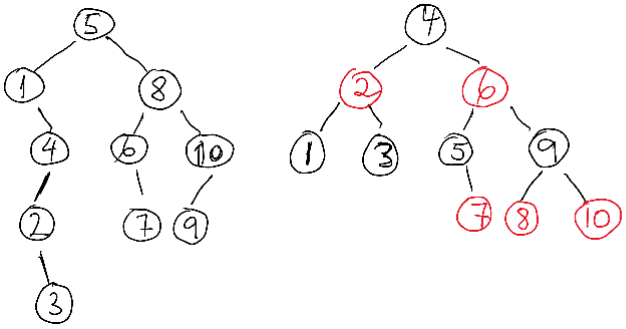
Når vi setter inn 3, kan vi gjøre en enkel venstrerotasjon og farge om 3.



Figur 5.7:

Resultat

Treet til venstre er det binære søketreet fra figur 4.3 og treet til høyre er red-black treet som vi nettopp har laget. I dette tilfellet klarte vi å lage treet uten kompliserte rotasjoner, og vi har ikke sett på sletting av noder. Se også `https:`



Figur 5.8:

`//en.wikipedia.org/wiki/Red%E2%80%93black_tree`.

5.3 stl set

Red-black tre er brukt til å implementere **stl::set**. I **stl::multiset** er duplikater tillatt. **stl::map** har to template parametre og svaret ellers til stl::set. På samme måte svarer **stl::multimap** til stl::set. Nedenfor er et eksempel på hvordan vi bruker sistnevnte.

Listing 5.1: set_eks.cpp

```
#include <iostream>
#include <set>
#include <map>

using namespace std;

// set - ordnet container uten duplikater, altsa sortert
// begin end size empty insert
// siden duplikater ikke er tillatt, er det logisk a
// returnere en bool ved insert
// insert returnerer i tillegg en iterator til innsatt element
// returverdien er altsa et par:

//set<double>::iterator it;
typedef set<double>::iterator setIt;

void setEks()
{
    pair<setIt, bool> par;
    set<double> ds;

    par = ds.insert(3.14);
    if (par.second) cout << "insert ok" << endl;
    ds.insert(9.81);
    if (par.second) cout << "insert ok" << endl;
    ds.insert(1.0);
    if (par.second) cout << "insert ok" << endl;
    for (auto it=ds.begin(); it!=ds.end(); it++)
        cout << *it << ", ";
    cout << endl;
}
// map - ordnet container med (key, value) par
// key - entydig
// flere keys kan mappe til samme value
// map virker som set med sammenligning kun pa key
// begin end size empty insert find erase
// ValueType& operator[] (const KeyType& key);
```

5.4 B-trær

Et binært tre er som vi nå vet et tre hvor hver node kan ha to subtrær. Et B-tre ([2], 4.7) har en orden m som sier hvor mange grener/subtrær/barn treet kan ha. Et B-tre har en orden m som angir antall mulige subtrær for hver node. I et binært søketre har vi *en* nøkkel som brukes til å avgjøre om vi skal søke videre i venstre eller høyre subtre. Med to nøkler kan vi ha tre subtrær for hver node, med tre nøkler kan vi ha fire subtrær osv. I et B-tre brukes alle de indre nodene til søk, og data ligger lagret i bladene. Egenskapene ([2], 4.7) er:

1. De indre nodene kan ha inntil m-1 nøkler.
2. Data lagres kun i bladene.
3. Roten er enten et blad eller har mellom $m/2$ og m barn (subtrær eller blader)
4. De indre nodene bortsett fra roten har mellom $m/2$ og m barn.
5. Alle bladene ligger på samme nivå og har mellom L og L/2 dataverdier. Vi kan ha L=M, men ikke nødvendigvis.

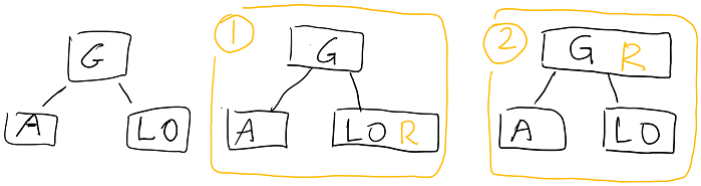
Et B+ -tre er i hovedsak et B-tre hvor i tillegg bladene er lenket sammen (`https://en.wikipedia.org/wiki/B%2B_tree`). Disse tretypene kan brukes som indeksfiler for databaser, og størrelsen på bladene bør velges i samsvar med en disk blokk.

5.5 2-3-4 trær

Et 2-3-4 tre er et B-tre av orden 4. En node i et 2-3-4-tre kan være en 2-node, 3-node eller 4-node. Husk at en m-node kan ha m barn og m-*1* keys(nøkler). Dette gjelder både indre noder og bladnoder. Et 2-3-4-tre er som både binært søketre og B-trær generelt et søketre.

5.5.1 Eksempel

La oss begynne med å se på et eksempel hvor vi setter inn **ALGORITME** i et 2-3-4 tre. Det er altså tillatt med inntil 3 keys og 4 barn for hver node.



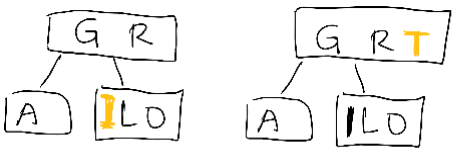
Figur 5.9:

1. De tre første bokstavene kan settes inn i roten (som til å begynne med kan være et blad). **A - AL - AGL**

2. Insetting av O krever en splitt og innføring av ett ekstra nivå. Hvordan? Vi kan la den midterste verdien G bli en ny rot med A som venstrebarn og L som høyrebarn. Deretter starter vi i roten og setter inn den nye verdien i et blad (som for binært søketre). Vi får da ett nivå mer, som på figur 5.9 (venstre).

3. R kan settes inn som vist på figur 5.9 (midtre eller høyre). Hvilket tre skal vi velge? La oss velge høyre og prøve oss fram litt til.

4. Hvis vi velger høyre alternativ, blir det lett å sette inn de to neste bokstavene I og T (figur 5.10.



Figur 5.10:

5.5.2 Splitt

M skal settes inn mellom G og R, men i blad ILO er det fullt. Vi må gjøre en splitt. Det gjøres ved at midterste verdi L går til moren, og vi lager to nye 1-noder av I og O. Dette er måten å gjøre det på generelt.

Men vi har et problem, siden det allerede er fullt i roten. Løsningen kan være å skyve høyeste verdi T i roten nedover, som på figur 5.12. Til sist kan vi sette inn siste bokstav E enkelt i blad med A.

Figur 5.11:

5.5.3 Alternativ innsetting

Hva skjer hvis vi i stedet går videre fra alternativ 1, figur 5.9? I er neste bokstav som skal settes inn. Det vil kreve en vanlig splitt av LOR-noden. Dette og påfølgende innsetting er vist på ??. Resultatet blir anderledes - vi trenger en algoritme her.

Figur 5.12:

5.5.4 Lenker

Dette var som tidligere nevnt et prøve-og-feile eksempel. Er det samsvar mellom dette eksemplet og fungerende algoritmer? Se `https://en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree`
`https://www.cs.princeton.edu/courses/archive/fall06/cos226/lectures/balanced.pdf`
Det er en interessant sammenligning mellom 2-3-4 trær og Red-black trær, se `https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf`.

5.6 Quadtrær

Et quadtre er et tre hvor hver node har fire barn eller er et blad. Et quad er et engelsk navn på rektangel. Det fins ulike typer quadtrær, se for eksempel `https://en.wikipedia.org/wiki/Quadtree`. Vi har allerede sett litt på quadtre-relaterte tema i avsnitt 3.1.3. Et quadtre kan brukes til å organisere objektene i et spill og gjøre rendring og kollisjonsdetektering mer effektivt. Gitt koordinatene til et bevegelig objekt X i spillet, kan man søke seg fram til hvilket quad objektet befinner seg i på $O(\log n)$ tid. Dette blir nesten som å bruke et binært søketre, men her må vi sammenligne xy-koordinater i stedet for nøkkelverdier.

Vi bruker samme notasjon som i avsnitt 3.1.3 og benevner hjørnene A, B, C, D med A som nederste venstre hjørne og positiv omløpsretning, og midtpunktet M. Vi bruker pekere, referanser eller indekser til subtrær med vanlige navn **sv, so, no, nv** (**sw, se, ne, nw**). Et quadtre for oss skal altså både være en datastruktur og et søketre. Vi kan få bruk for noen hjelpeklasser:

5.6.1 Vector2d

Vector2d er en hjelpeklasse som vi skal implementere mer av senere. Her er følgende tilstrekkelig:

Listing 5.2: Vector2d

```
struct Vector2d {
    double x;
    double y;
    Vector2d operator + (const Vector2d& v) const
    {
            Vector2d u;
            u.x = x + v.x;
            u.y = y + v.y;
            return u;
    }
    Vector2d operator / (int d)
    {
        Vector2d u;
        u.x = x/d;
        u.y = y/d;
        return u;
    }
};
```

5.6.2 GameObject

Vi skal her bruke kun posisjon og navn på spillobjekter, og følgende klasse kan benyttes:

Listing 5.3: GameObject

```
struct GameObject {
    Vector2d m_position;
    std::string m_navn;
    GameObject() { /* to do */ }
    GameObject(const Vector2d& position, std::string navn)
        : m_position{position}, m_navn{navn} { }
```

```
    Vector2d getPosition() const { return m_position; }
};
```

5.6.3 Quadtre klasse

Så følger et forslag til dynamisk implementering av quadtre.

Listing 5.4: Quadtre

```
class QuadTre
{
private:
    Vector2d m_a;
    Vector2d m_b;
    Vector2d m_c;
    Vector2d m_d;
    QuadTre* m_sv;
    QuadTre* m_so;
    QuadTre* m_no;
    QuadTre* m_nv;
    std::vector<GameObject> m_gameObjects;
    bool isLeaf() const;
public:
    QuadTre();
    QuadTre(const Vector2d &v1, const Vector2d &v2, const Vector2d &v3, const Vector2d & v4);
    void subDivide(int n);
    void print() const;
    QuadTre* insert(const GameObject& gameObject);
    QuadTre* find(const Vector2d& p);
};
```

5.6.4 Quadtre søkefunksjon

Søking i et quadtre er ganske likt søking i et binært søketre. Men for hver indre node har vi fire barn og fire muligheter til å gå nedover i treet, i stedet for bare to. En rekursiv søkefunksjon kan implementeres som nedenfor.

Listing 5.5: Quadtre

```
QuadTre* QuadTre::find(const Vector2d& p)
{
   if (isLeaf()) {
       return this;
   }
   else {
       Vector2d m = (m_a+m_c)/2;
       if (p.y < m.y)
       {
           if (p.x < m.x)
               m_sv->find(p);
           else
               m_so->find(p);
       }
       else {
           if (p.x < m.x)
               m_nv->find(p);
           else
               m_no->find(p);
       }
   }
}
```

5.7 Oppgaver

5.7.1

Lag et quadtre hvor bladene ligger på minst to ulike nivåer.

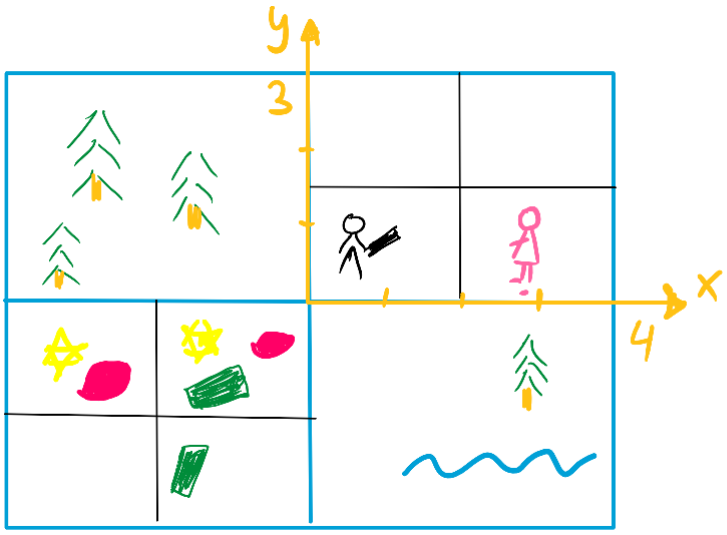
5.7.2

Lag en funksjon som skriver ut (hjørnene til) alle quad som er blader. Lag et testprogram.

5.7.3

En skjematisk scene fra et spill kan være som på figur 5.13. I øverste venstre kvadrant i xy-planet er det noen spredte trær. I nederste venstre kvadrant er det noen items (gule stjerner, røde mynter og grønn seddel). I nederste høyre kvadrant er et vann og et tre, og i øverste høyre kvadrant er en svart NPC med balltre og en rosa player. Alle disse objektene har en xy-koordinat som kan leses av i koordinatsystemet.

Alle objektene kan være (avledet) av typen GameObject, hvor en tilstrekkelig GameObject klasse kan være som i avsnitt 5.6.2. (Husk at en struct i C++ er en klasse hvor alt er public).



Figur 5.13:

- I denne oppgaven skal vi tenke oss at alle objektene i spillet (alle GameObject instansene) er laget, og at de ligger i en std::vector<GameObject*>. De er altså lagret med posisjon og navn.
- Videre tenker vi oss at navnet skal skrives ut som en erstatning for rendring av et 3d objekt.

- Vi skal så opprette et quadtre som vi kan bruke til å behandle deler av scenen separat. For eksempel kan vi tenke oss at det stilles større krav til nøyaktighet ved kollisjonsbehandling i nederste venstre hjørne, hvor spilleren skal plukke items, enn i øverste venstre og nederste høyre hjørne.
- Det er også relativt mange spillobjekter i nederste venstre hjørne. Vi kan derfor tenke oss at vi deler opp hele scenen i fire rektangler, og deretter deler vi opp to rektangler enda en gang. Se figur. Alt dette er selvsagt skjematisk og forenklet, men prinsippene blir som i et dataspill.

Oppgaven blir da:

1. Lag en funksjon hvor du oppretter en GameObject instans for hvert objekt på scenen, leser av koordinatene manuelt og legger inn en peker til objektet i en std::vector<GameObject*>. Dette kan gjøres ved gjentatte kall på parametrisk konstruktør.
2. Lag et quadtre som vist på figuren. Både i dette og forrige punkt må du bruke flere lignende kodesetninger - det er ikke meningen å finne på noen algoritme for dette.
3. Lag så en innsettingsfunksjon (insert) for QuadTre klassen som skal brukes på alle objektene i scenen, slik at hvert quadtre objekt får sine egne pekere (se objektvariabelen gameObjects) til de objektene som hører til det aktuelle rektanglet. (ikke bruk unique_ptr her).
4. Lag en testfunksjon hvor du kan oppgi koordinater og få skrevet ut navn på alle objekter som ligger i rektanglet med de angitte koordinatene.

Eksempel: roten skal ha fire pekere som peker til hvert sitt QuadTre objekt. **nv** objektet skal ha pekere til tre objekter i sin gameObject vektor. **se** objektet skal ha pekere til fire mindre Quadtre-objekter, men gameObject vektoren skal ikke inneholde noen pekere her.

5.8 Løsningsforslag

Vi ser først på konstruktør, funksjonen subDivide() som tilsvarer subDivide() fra rektangel-eksemplet, print() og isLeaf():

Listing 5.6: Quadtre

```
QuadTre::QuadTre(const Vector2d &v1, const Vector2d &v2, const Vector2d &v3, const Vector2d &
    : m_a{v1}, m_b{v2}, m_c{v3}, m_d{v4},
      m_sv{nullptr}, m_so{nullptr}, m_no{nullptr}, m_nv{nullptr}
{
   //
}

void QuadTre::subDivide(int n) // uniform rekursiv subdivisjon, fyller alle levels
{
    if (n>0) {
        Vector2d v1 = (m_a+m_b)/2;
        Vector2d v2 = (m_b+m_c)/2;
        Vector2d v3 = (m_c+m_d)/2;
        Vector2d v4 = (m_d+m_a)/2;
        Vector2d m = (m_a+m_c)/2;
        m_sv = new QuadTre(m_a, v1, m, v4);
        m_sv->subDivide(n-1);
        m_so = new QuadTre(v1, m_b, v2, m);
        m_so->subDivide(n-1);
        m_no = new QuadTre(m, v2, m_c, v3);
        m_no->subDivide(n-1);
        m_nv = new QuadTre(v4, m, v3, m_d);
        m_nv->subDivide(n-1);
    }
}

void QuadTre::print() const
{
    Vector2d c = (m_a+m_c)/2;
    std::cout << "senter = (" << c.x << ", " << c.y << ")" << std::endl;
    for (auto it=m_gameObjects.begin(); it!=m_gameObjects.end(); it++)
        std::cout << it->m_navn << std::endl;
}

bool QuadTre::isLeaf() const
{
   return m_sv == nullptr && m_so == nullptr && m_no == nullptr && m_nv == nullptr;
}
```

insert() funksjonen er rekursiv, og ligner på find():

Listing 5.7: Quadtre::insert()

```
QuadTre *QuadTre::insert(const GameObject &gameObject)
{
    if (isLeaf()) {
        m_gameObjects.push_back(gameObject);
        return this;
    }
    else {
        Vector2d m = (m_a + m_c)/2;
        if (gameObject.getPosition().y < m.y)
        {
            if (gameObject.getPosition().x < m.x)
                m_sv->insert(gameObject);
            else
                m_so->insert(gameObject);
        }
        else {
            if (gameObject.getPosition().x < m.x)
                m_nv->insert(gameObject);
            else
                m_no->insert(gameObject);
        }
    }
}
```

Til slutt kan vi teste klassen med et eksempel. Her har vi brukt eksemplet på figur 5.13.

Listing 5.8: Quadtre::insert()

```
int main()
{
    vector<GameObject> gameObjects;

    // Lager rektangel
    Vector2d a{-4, -3};
    Vector2d b{ 4, -3};
    Vector2d c{ 4,  3};
    Vector2d d{ -4, 3};

    // Legger inn objektene i gameObjects
    // nv
    Vector2d p{-3.25, 0.25};
    gameObjects.push_back(GameObject{p, "grantre1"});
    p.x = -3.25; p.y = 1.5;
    gameObjects.push_back(GameObject{p, "grantre2"});
    p.x = -1.5; p.y = 1.0;
    gameObjects.push_back(GameObject{p, "grantre3"});
    // sv - nv
    p.x = -3.25; p.y = -0.75;
    gameObjects.push_back(GameObject{p, "stjerne1"});
    p.x = -2.75; p.y = -1.0;
    gameObjects.push_back(GameObject{p, "mynt1"});
    // sv - no
    p.x = -1.5; p.y = -0.5;
    gameObjects.push_back(GameObject{p, "stjerne2"});
    p.x = -0.5; p.y = -0.5;
    gameObjects.push_back(GameObject{p, "mynt2"});
    p.x = -1.5; p.y = -1.0;
    gameObjects.push_back(GameObject{p, "seddel1"});
    // sv - so
    p.x = -1.75; p.y = -2.0;
    gameObjects.push_back(GameObject{p, "seddel2"});
    // so
    p.x = 2.75; p.y = -1.0;
    gameObjects.push_back(GameObject{p, "grantre4"});
    p.x = 2.0; p.y = -2.0;
    gameObjects.push_back(GameObject{p, "vann"});
    // no - sv
    p.x = 0.5; p.y = 0.5;
    gameObjects.push_back(GameObject{p, "fiende"});
    // no - so
    p.x = 3.0; p.y = 0.25;
    gameObjects.push_back(GameObject{p, "spiller"});

    // Lager quadtre med subdivisjon
    QuadTre root{a, b, c, d};
    root.subDivide(1);
    QuadTre* subtre = root.find(Vector2d{-1,-1});
    subtre->subDivide(1);
    subtre = root.find(Vector2d{1,1});
    subtre->subDivide(1);

    cout << "----------------------------" << endl;
    cout << "Setter inn og skriver ut " << endl;
    cout << "----------------------------" << endl;
    for (auto go : gameObjects)
    {
```

```
        auto p = root.insert(go);
        p->print();
    }
    cout << "-------------------------" << endl;
    cout << "Skriver ut objektene " << endl;
    cout << "-------------------------" << endl;
    subtre = root.find(Vector2d{-1, 1});
    subtre->print();
    subtre = root.find(Vector2d{-2.5, -1});
    subtre->print();
    subtre = root.find(Vector2d{-1.5, -2});
    subtre->print();
    subtre = root.find(Vector2d{-1.5, -1});
    subtre->print();
    subtre = root.find(Vector2d{ 1, -1});
    subtre->print();
    subtre = root.find(Vector2d{ 1, 1});
    subtre->print();
    subtre = root.find(Vector2d{ 3, 1});
    subtre->print();
    return 0;
}
```

Kapittel 6

Heap

6.1 Introduksjon

En prioritetskø er en kø hvor elementene holdes stigende eller avtakende sortert. En heap er et nesten komplett binært tre (avsnitt 4.1.3 og kan brukes til å implementere en prioritetskø. En max heap er synonymt med en avtakende prioritetskø, og er standard implementeringen i stl. I en max heap ligger største element først. En min heap er det samme som en stigende prioritetskø, og her ligger det minste elementet først. Vi skal vise eksempler på begge.

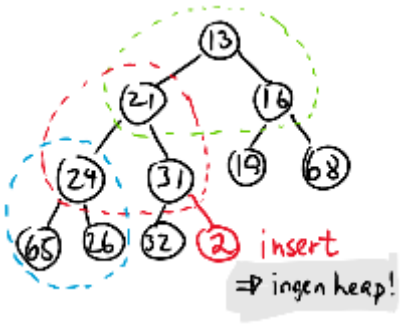
En heap er et nesten komplett binært tre (avsnitt 4.1.3. Vi kan implementere en heap ved hjelp av et dynamisk binært tre, eller ved å bruke en array. Det er enklere å forestille seg operasjonene som må gjøres hvis vi tegner opp heapen som et dynamisk binært tre. En int heap kan for eksempel bli noe som dette:

6.1.1 Datastruktur

```
struct BinaryNode
{
   int data;
   BinaryNode* left;
   BinaryNode* right;
   void push(int i);
   void pop();
   int top();
   int size();
   bool empty();
};
```

Vi har brukt dette klassenavnet tidligere. Det er implementeringen av push() og pop() som skal gjøre dette til en heap og ikke et binært søketre.

La os se på et eksempel på en min heap (stigende prioritetskø), hvor det minste elementet skal ligge øverst ([2], fig. 6.5). Vi ser her at en min heap har følgende



Figur 6.1: Eksempel på stigende prioritetskø

egenskaper:

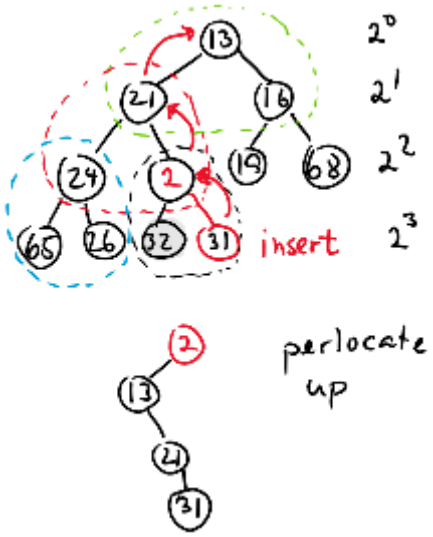
- Roten er minst.
- Dette gjelder for hvert subtre.
- Det har ingen ting å si hvem som er størst av venstre og høyre barn.
- Vi kan ha venstre<høyre eller venstre>høyre, begge er tillatt.
- Rekursiv struktur.
- pop() betyr å fjerne roten. Treet må reorganiseres slik at roten forblir det minste element etter pop.
- push() betyr (som vanlig) innsetting av nytt blad. Her settes blad inn på første ledige posisjon i et nesten komplett binært tre. Treet må reorganiseres.

6.2 push()

Reorganiseringen til en heap etter innsetting kalles å perlokere opp. Se figur 6.2. Noden som er satt inn, byttes oppover så lenge moren er større. Dette gjøres på $O(\log n)$ operasjoner.

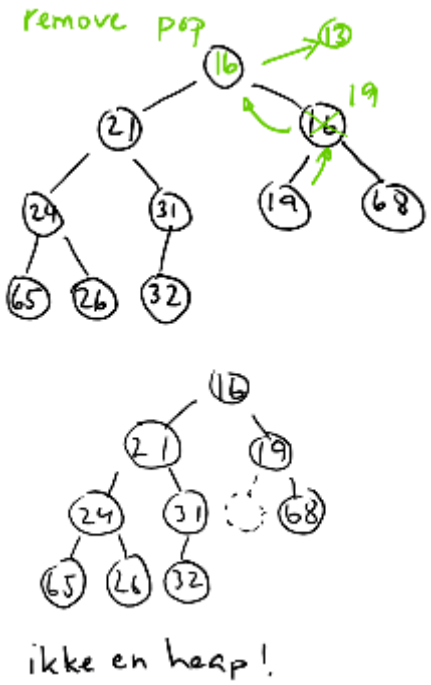
6.3 pop()

Anta nå at vi skal slette minste element (13) i stredet for å legge inn 2. Vi kan tenke oss at vi prøver med en slags perlokering, se figur 6.3. Men dette er ikke en heap. På figur 6.4 ser vi hva som må gjøres.



Figur 6.2: Innsetting

1. Vi finner minste barn til roten (16) og setter inn denne noden på rotens plass. Denne operasjonen er jo å poppe fra høyre subtre.
2. Vi finner minste barn til 16 (19) og setter inn denne noden på 16 sin plass (som ble ledig, akkurat som roten i hele treet ble ledig etter pop).
3. Dette gjentas til vi når et blad.
4. Hvis vi ender opp med å flytte opp et høyrebarn, er vi ferdige.
5. Hvis vi ender opp med å flytte opp et venstrebarn, som her (32), må vi flytte høyrebarnet på nederste nivå til dette venstrebarnet sin plass. (Hvis vi ikke hadde hatt det nederste nivået her, med 65, 26, 32), måtte vi endret på 68 til å bli venstrebarn.



Figur 6.3: Sletting (ufullstendig)

6.4 stl priority_queue

Nedenfor er et eksempel på bruk priority_queue i stl. Legg merke til syntaksen for å opprette en min heap.

Listing 6.1: heap_eks.cpp

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    cout << "max heap (descending) = default" << endl;
    priority_queue<double> max_heap;
    max_heap.push(3.14);
    max_heap.push(9.81);
    max_heap.push(2.71828);

    auto rot = max_heap.top();
    cout << "Foer pop: heap rot = heap top = " << rot << endl;

    max_heap.pop();
    rot = max_heap.top();
    cout << "Etter pop: heap rot = heap top = " << rot << endl;

    cout << "min heap (ascending)" << endl;
    priority_queue<double, vector<double>, greater<double>> min_heap;

    min_heap.push(3.14);
```

```
    min_heap.push(9.81);
    min_heap.push(2.71828);

    auto min_rot = min_heap.top();
    cout << "Foer pop: heap rot = heap top = " << min_rot << endl;

    min_heap.pop();
    min_rot = min_heap.top();
    cout << "Etter pop: heap rot = heap top = " << min_rot << endl;
    cout << "Etter pop: heap size = " << min_heap.size() << endl;
    cout << "Etter pop: heap empty? " << boolalpha << min_heap.empty() << endl;
    return 0;
}
```

6.5 Oppgaver

6.5.1

Sett inn tallene 9, 17, 2, 7, 5, 8, 1, 22, 4, 11, 6 i en stigende prioritetskø.

6.5.2

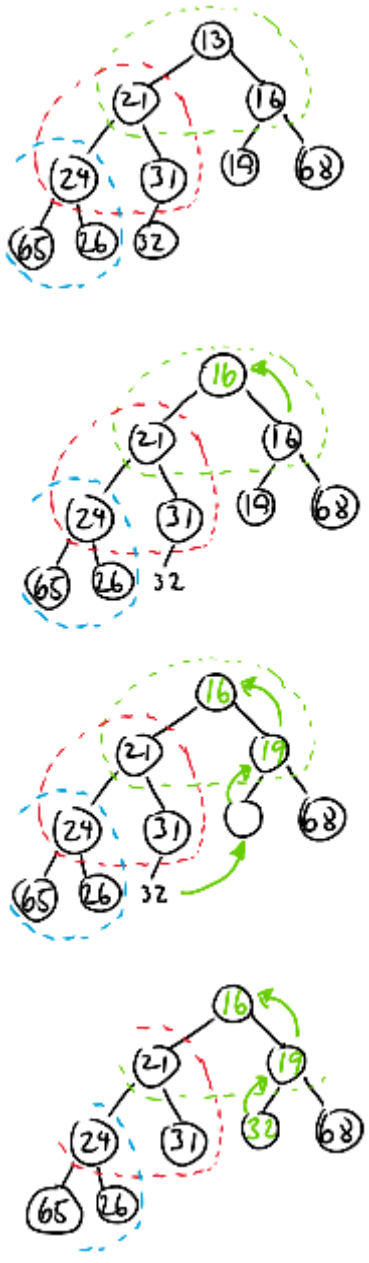
Skriv et lite program som sorterer de samme tallene ved hjelp av stl prioritetskø.

6.5.3

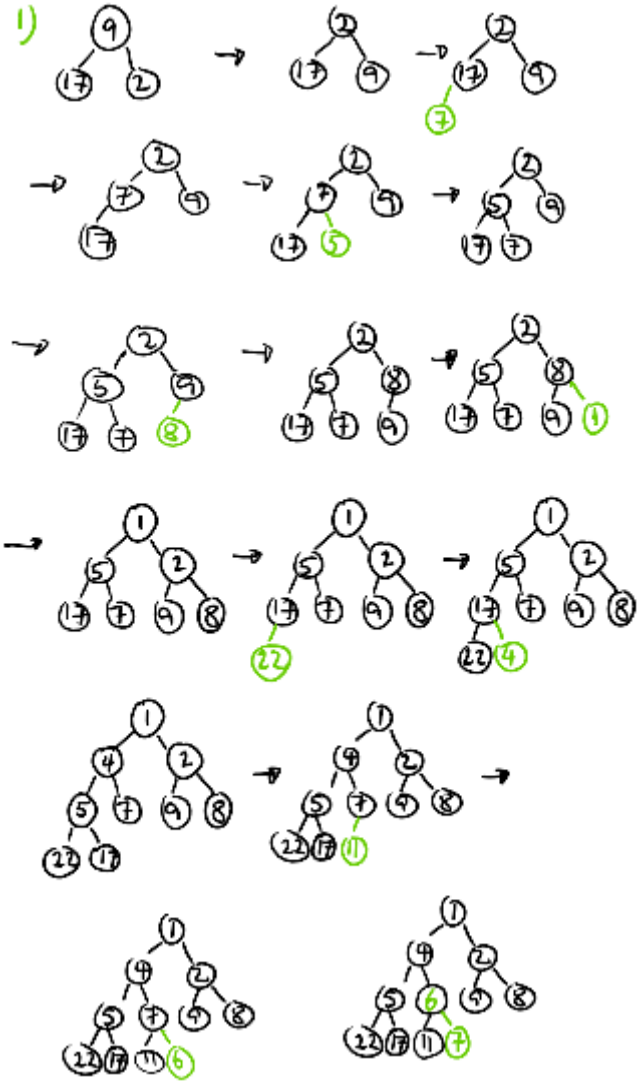
Blir det alltid riktig å reorganisere heapen (ved å flytte et blad over i høyre subtre) som på figur 6.4?

6.5.4

Lag en egen heap implementering, enten ved å bruke en dynamisk struktur eller array til å lagre heapen.



Figur 6.4: Sletting



Figur 6.5:

Kapittel 7

Sortering

7.1 Sortering introduksjon

Dette kapitlet består av sammendrag om sortering. Vi skal bruke litt tid på

- å lære om ulike klassiske sorteringsalgoritmer og fordeler og ulemper ved disse i ulike anvendelser, og bli i stand til å vurdere disse.
- Sammenhengen mellom sortering og søking, spesielt relatert til søking og indeksering i relasjonsdatabaser.
- Bredde - Sammenlikning av ulike algoritmer, effektivitet, O(n) notasjon
- Implementering og bruk av stl

Mål

Lære om ulike klassiske sorteringsalgoritmer og fordeler og ulemper ved disse i ulike anvendelser, og bli i stand til å vurdere disse. Sammenhengen mellom sortering og søking, spesielt relatert til søking og indeksering i relasjonsdatabaser.

7.1.1 Repetisjon

- Utvalgssortering (selection sort) $O(n^2)$
- Boblesortering (bubble sort) $O(n^2)$
- Innstikksortering (insewrtion sort) $O(n^2)$
- Binært søketre sortering $O(n \log n)$
- Dybde - Grundigere studium av de enkelte algoritmer

Nytt:

- Shellsort - første algoritme som var raskere enn $O(n^2)$
- Qiucksort $O(n \log n)$
- Heapsort $O(n \log n)$

- Mergesort $O(n \log n)$

Viktig: Husk sammenhengen mellom sortering og søking - motivasjonen for å holde en array eller fil sortert, hva er det? Hva er den minst tenkelige søketiden vi kan forestille oss? Husk dette spørsmålet til vi kommer til hashing.

7.1.2 Velge algoritme

Noen ganger kan en enkel sorteringsalgoritme være bra, til og med bedre enn en mer avansert algoritme. Eksempel på dette er når vi har små filer. Når vi skal velge en algoritme, trenger vi et beslutningsgrunnlag. Vi vil finne den best mulige algoritmen. Vi skal se litt på kriterier som vi kan bruke til å sammenlikne algoritmer.

- Hvordan måler vi effektiviteten til en algoritme?
- Kritiske operasjoner: Sammenlikne nøkler, flytte poster eller pekere, etc.
- Numeriske algoritmer: Multiplikasjon/divisjon kritiske i forhold til addisjon/subtraksjon.
- Teller opp kritiske operasjoner i løkker.

Hva regner vi ut? Vi kan telle antall tilordninger, sammenligninger, addisjoner, multiplikasjoner, funksjonskall. Vi er interessert i å telle det som bidrar til CPU tiden. Algoritmene omfatter vanligvis ulike typer operasjoner, og vi kan forenkle ved å utelate operasjoner som relativt sett bidrar lite til CPU tiden. Et eksempel: Addisjoner tar vesentlig mindre tid enn multiplikasjoner, så når man sammenlikner numeriske algoritmer, telles multiplikasjoner/divisjoner, men ikke addisjoner/subtraksjoner. Erfaringsmessig er det i størrelsesorden like mange addisjoner som multiplikasjoner i en numerisk algoritme.

7.1.3 Eksempel

Antall kritiske operasjoner i en enkel for-løkke fra 1 til n eller fra 0 til n-1 med en kritisk operasjon for hver gjennomløpning er $\Sigma_{i=1}^{n}\ 1 = n$

Hvis vi har en dobbel for-løkke fra 0 til n-1 med en kritisk operasjon for hver gjenomløpning,

```
for (auto i=0; i<n; i++)
   for (auto j=0; j<n; j++)
      a[i] = i*j;
```

regner vi ut en dobbel sum. Antall multiplikasjoner her er

$$\Sigma_{i=1}^{n}\ \Sigma_{j=1}^{n}\ 1 = n\ \Sigma_{i=1}^{n}\ 1 = n^2$$

som selvfølgelig er $O(n^2)$.

7.1.4 O(n) notasjonen

Når vi har mange elementer å sortere, altså når n er stor, er n^2 mye større enn n. Herav følger O(n) notasjonen. Når vi teller opp antall kritiske operasjoner i en algoritme, er det høyeste potens av n som betyr mest for hvor rask algoritmen er.

7.1.5 Hvordan måle effektivitet?

- Ved å telle antall kritiske operasjoner, eller
- ved å kjøre algoritmen/programmet og ta tiden for forskjellige eksempelfiler.

Vanligvis er sorteringstiden til en algoritme avhengig av rekkefølgen på inndata (f.eks. binært søketre sortering). I sorteringsfunksjoner er det vanlig å forsake lesbarhet i koden til fordel for høyest mulig hastighet.

Eksempel

Listing 7.1: chrono_eks.cpp

```
// Tidtaking med chrono

#include <iostream>
#include <chrono>
using namespace std;

void sort_tid_0(unsigned long ul) // kun tidtakings-kode
{
    chrono::nanoseconds total_tid{0};

    // tilordne random verdier
    auto start = std::chrono::high_resolution_clock::now();

    // plassholder for sorteringsfunksjon
    for (auto i=0; i<ul; i++)
        double x = 3.14 * 2.51;

    auto slutt = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> varighet = slutt-start;
    std::chrono::nanoseconds varighet_nano =
            std::chrono::duration_cast<std::chrono::nanoseconds >(varighet );
    total_tid += varighet_nano;

    // utskrift
    cout << total_tid.count() << endl;
}

int main()
{
    sort_tid_0(100000);
}
```

7.2 $O(n^2)$ sorteringsalgoritmer

7.2.1 Boblesortering

En såkalt exchange sorts algoritme som er enkel å forstå, og enkel å programmere.

```
for (auto i=0; i<n-1; i++) {
   for (auto j=0; j<n-1; j++) {
      if (a[j] < a[j+1])
```

```
      {
         auto hjelp = a[j];
         a[j] = a[j+1];
         a[j+1] = hjelp;
      }
   }
}
```

Men den er treg, den bruker $(n-1)\cdot(n-1) = n^2 - 2n + 1$ sammenlikninger og er $O(n^2)$. Det er enkelt å gjøre forbedringer på denne ([1]), men den vil alltid være $O(n^2)$ for et generelt tilfelle. Boblesorterings-algoritmen har imidlertid en veldig god egenskap: Den er O(n) for et datasett som er (nesten) sortert på forhånd.

7.2.2 Utvalgssortering

Ideen bak utvalgssortering er: Plasser minste element i posisjon 0, deretter nest minste element i posisjon 1 osv. Dette er kjent stoff. Den enkleste utvagssorterings-algoritmen kan implementeres slik:

```
for (auto i=0; i<n-1; i++)
{
   for (auto j=i+1; j<n; j++)
   {
      if (a[j] < a[i])
      {
         auto hjelp = a[i];
         a[i] = a[j];
         a[j] = hjelp;
      }
   }
}
```

Denne algoritmen er $O(n^2)$ - alltid, men vanligvis raskere enn boblesortering som også er $O(n^2)$.

Generell algoritme

Den generelle algoritmen for utvalgssortering: Gitt en array v med n elementer og en avtakende prioritetskø pq.

```
for (i=0; i<n; i++)
   sett inn v[i] i pq
for (i=n-1; i>=0; i--)
 v[i] = største element i pq
```

Dette gir en avtakende sortert array. Hvis vi i stedet ønsker en stigende sortering, brukes en stigende priorotetskø. I denne algoritmen implementeres prioritetskøa som uordnet array, og den originale arrayen brukes til å holde køen.

7.2.3 Binærtre sortering

Sortering ved hjelp av binært søketre og inorder traversering er ren repetisjon for oss. Denne algoritmen er $O(n \log n)$ for et balansert tre. Som quicksort er den dårligst på en ferdig sortert fil, da er den$O(n^2)$. Vi husker at dette skyldes at treet reduseres til en lineær liste. Legg merke til at denne algoritmen kategoriseres som utvalgssortering, og at et binært søketre kan representere en prioritetskø.

7.2.4 Innstikksortering

Innstikk-sortering (insertion sort) er også en $O(n^2)$ algoritme. Ideen her er å flytte det minste (eller største) elementet fremst, gjentatte ganger. Innstikksortering kan betraktes som en utvalgssortering hvor vi implementerer prioritetskøen som en ordnet array. Når preprosesseringen er ferdig, dvs. når elementene er satt inn i prioritetskøa, er sorteringen ferdig! Implementeringen:

```
for (auto i=1; i<n; i++)
{
   auto hjelp = a[i];
   for (auto j=i-1; j>=0 && a[j] > hjelp; j--)
      a[j+1] = a[j];
   a[j+1] = hjelp;
}
```

7.3 Fletting

Ideen bak algoritmen er å starte med å dele et datasett med n elementer (en array med n elementer) i n subarrayer med kun ett element i hver. Så flettes to og to av disse, til ca n/2 arrayer med to elementer. Slik fortsetter prosessen helt til vi har **en** sortert array med **n** elementer. Studer algoritmen:

Listing 7.2: mergesort.cpp

```
void mergeSort(std::array<int,n> a)
{
    int i, j, k, lower1, lower2, size, upper1, upper2;
    std::array<int,n> hjelp;

    size = 1;
    while (size < n)
    {
           lower1 = 0;
           k = 0;
           while (lower1+size < n)
           {
              upper1 = lower1+size-1;
              lower2 = upper1+1;
              upper2 = (lower2+size-1 < n) ? lower2+size-1 : n-1;
              for (i=lower1, j=lower2; i<=upper1 && j<=upper2; k++)
                 if (a[i] < a[j])
                    hjelp[k]=a[i++];
                 else
                    hjelp[k] = a[j++];
```

```
            for (; i<=upper1; k++)
                hjelp[k] = a[i++];
            for (; j<=upper2; k++)
                hjelp[k] = a[j++];

            lower1 = upper2+1;
        } // endwhile

        for (i=lower1; k<n; i++)
            hjelp[k++] = a[i];
        for (i=0; i<n; i++)
            a[i] = hjelp[i];

        size = size*2;
    } //endwhile
}
```

Bruk programmet. Flettealgoritmen (merge sort) blir vesentlig enklere å forstå dersom dere prøver den på ulike små, oversiktlige arrayer. Et eksempel:

9	7	1	5	6	4	9	8	7
7	9	1	5	4	6	8	9	7
1	5	7	9	4	6	8	9	7
1	4	5	6	7	8	9	9	7
1	4	5	6	7	7	8	9	9

Her har vi en liten array med 9 heltall (øverste linje). I linje to har vi fire arrayer med to elementer (det siste elementet blir hengende igjen, siden det er en oddetallsarray). To og to tall har blitt flettet $9, 7 \rightarrow 7, 9, \ldots$. Fra andre til tredje linje flettes arrayer av lengde to til en array av lengde fire $7, 9, 15 \rightarrow 1, 5, 7, 9$. De fire neste elementene er allerede sortert. Fra tredje til fjerde linje flettes to arrayer av lengde fire til en array av lengde åtte. Til slutt flettes arrayen med de åtte første tallene med den neste arrayen (her har den kun ett tall (7) - men det kunne ha vært inntil åtte).

Flettealgoritmen har en stor fordel i det at den aldri bruker mer enn $n \log_2 n$ sammenligninger! Dette er mye bedre enn worst case for quicksort og binærtresortering. I gjennomsnitt bruker flettesortering $n \log_2 n - n + 1$ sammenlikninger. Dette er bedre enn quicksort. Til gjengjeld bruker flettesortering omtrent dobbelt så mange tilordninger som quicksort. Den trenger også O(n) ekstra plass under sorteringen.

Flettealgoritmen er vanligvis foretrukket algoritme ved fletting av filer.

7.4 Heapsort

Heapsortering er en $O(n \log n)$ algoritme uavhengig av inndata. Det er bedre enn binærtre sortering for veldig ubalanserte tre. I heapsortering bruker vi også et binært tre. Dette treet er ikke et binært søketre, men en heap.

Vi vet fra tidligere at vi kan ha en stigende eler avtakende heap. En avtakende

heap (descending heap, max heap, descending partially ordered tree) av størrelse n er et nesten komplett binært tre med n noder, hvor hver nodes nøkkel er mindre eller lik over-nodens nøkkel. Følgelig vil treets rot inneholde det største elementet.

En heap er et nesten komplett binært tre. Både innsetting og sletting kan da gjøres ved $O(\log n)$ operasjoner. For n elementer tar da sorteringen $O(n \log n)$ operasjoner.

I heapsortering bruker vi en heap til å implementere en prioritetskø. Vi husker at i en prioritetskø setter vi inn elemeter bakerst og tar ut største element, etter den formelle definisjonen. Når vi bruker en heap som prioritetskø, gjør vi en liten endring i forhold til dette. Vi setter inn det nye elementet bakerst, men i innsettingsprosessen reorganiserer vi i tillegg binærtreet slik at det får tilbake heap egenskapen.

Heapsortering er som nevnt en $O(n \log n)$ algoritme. Den er bedre enn quicksort i worst case, men i gjennomsnitt krever den dobbelt så lang tid som quicksort. Den er heller ikke effektiv for små n.

7.5 Quicksort

Quicksort er også en exchange sorts algoritme, basert på en rekursiv observasjon/ide. Vi tenker oss følgende (vanlige) problem:

Gitt et datasett v med n elementer. Sorter v.

Velg et element a fra arrayen, for eksempel det første, slik at a = v[0] . Anta at arrayen er partisjonert slik at elementet a nå er plassert i posisjon j, a = v[j] og at følgende tilstand er oppfylt:

1. Hvert av elementene i posisjon 0 til $j-1$ er mindre eller lik a. 2) Hvert av elementene i posisjon $j+1$ til $n-1$ er større eller lik a.

En skjematisk figur av arrayen:

0	1	...	$j-1$	j	$j+1$		...					$n-1$
				$v[j]$								

Da er elementet a (også kalt et pivot) på den plassen det vil ha i en ferdig sortert array, og det opprinnelige problemet er redusert til to mindre delproblemer – å sortere venstre og høyre subarray, dvs. elementene i posisjon 0 til j-1 og posisjon j+1 til n-1. Hvis vi gjentatte ganger bruker strategien ovenfor på stadig mindre subarrayer, vil vi til slutt ha en sortert array.

Partisjonering

Vanskeligheten i quicksort ligger i partisjoneringen. I [2], figur 7.14 er partisjoneringen forklart med et eksempel. Algoritmen (figur 7.17) bruker to indekser eller pekere:

1. *i* som starter nederst i arrayen og søker oppover helt til den finner et tall større enn pivot. Altså økes i med en plass helt til a[i] > pivot.

2. j som starter øverst i arrayen og søker nedover helt til den finner et tall mindre enn eller lik pivot. Altså reduseres j helt til a[j] $\leq$ pivot.

3. Dersom $j > i$ byttes a[i] og a[j].

Hva skjer i punkt 3? a[i] er det første tallet nedenfra/fra venstre som er større enn pivot. a[j] er det første tallet ovenfra/fra høyre som er mindre enn eller lik a. Altså er a[j] $>$ a[i]. Hvis disse to indeksene ikke har møttes ennå, fører byttet til at det minste tallet flyttes ned i arrayen og det største tallet flyttes opp. Begge disse får dermed en riktigere posisjon med hensyn til sorteringen. Dette byttet er i henhold til betingelsene ovenfor arrayfiguren (1-2).

Når vi fortsetter prosessen 1-2, partisjonerer vi stadig større deler av venstre og høyre subarray i henhold til Equation 1. Prosessen 1-2 gjentas helt til betingelsen 3 er brutt. Det vil si at $i \leq j$.

i brukes for å søke etter et tall mindre eller lik pivot. Byttene har ført til at alle tallene til venstre for i er mindre eller lik pivot. Når betingelse 3 er brutt, er a[i] $>$ a[j]. Disse to tallene er da innbyrdes sortert. Nå byttes pivot=a[0] med a[i], og pivot er på rett plass.

Denne prosessen gjentas først for venstre og høyre subarray (som nå er å betrakte som a) og deretter for stadig flere og mindre subarrayer inntil hele arrayen er ferdig sortert.

Eksempel

Gitt arrayen

```
int a { 17, 14, 5, 7, 12, 1, 16, 29, 13, 4, 8, 18, 22, 2 } ;
```

med $n = 14$ elementer. Første kall på rekursiv quicksort funksjon blir da

```
quick(a, 0, 13);
```

Med noen utskrifter underveis får vi

```
pivot: a[6]=16
bytter a[13]=2 og a[0]=17
2, 14, 5, 7, 12, 1, 16, 29, 13, 4, 8, 18, 22,
bytter a[10]=8 og a[6]=16
2, 14, 5, 7, 12, 1, 8, 29, 13, 4, 16, 18, 22,
bytter a[10]=16 og a[7]=29
2, 14, 5, 7, 12, 1, 8, 16, 13, 4, 29, 18, 22,
bytter a[9]=4 og a[7]=16
2, 14, 5, 7, 12, 1, 8, 4, 13, 16, 29, 18, 22,
2, 14, 5, 7, 12, 1, 8, 4, 13, 16, 29, 18, 22, 17, i=9, j=9
```

Dette er altså et resultat (arrayen) etter første rekursive quicksort kall. Vi analyserer dette, og setter opp en tabell som viser hvordan i, j og *pivot* endrer seg:

```
0    1   2   3    4    5    6    7    8   9   10   11   12   13
-------------------------------------------------------------
17  14   5   7   12    1  |16|  29   13   4    8   18   22   2
 i                                                          j
 2  14   5   7   12    1  |16|  29   13   4    8   18   22   17
                          i                  j
 2  14   5   7   12    1   8    29   13   4  |16|  18   22   17
                           i                 j
 2  14   5   7   12    1   8   |16|  13   4   29   18   22   17
                            i         j
 2  14   5   7   12    1   8    4    13  |16|  29   18   22   17
```

Vi vet nå at pivotelementet (16) nå er på rett plass i det som skal bli den ferdig sorterte arrayen.

Vurdering av quicksort

Quicksort er en $O(n \log n)$ algoritme. Dette er lett å se hvis man antar at filstørrelsen er en potens av 2, og at pivotelementet alltid er det midterste i subarrayene og telle opp antall sammenlikninger i hver (sub)array multiplisert med arraystørrelsen, som i [1], side 348.

Quicksort uten modifikasjoner kan være så dårlig som $Q(n^2)$ for eksempel dersom den opprinnelige arrayen allerede er sortert.

Det kan vises ([?]) at den umodifiserte quicksort algoritmen i gjennomsnitt bruker $1.386 \cdot n \log n$ sammenlikninger.

Eksempel med enkelt pivoteringsvalg

Nedenfor er et eksempel på implementering, med heltallsarray.

Listing 7.3: quick_rec.cpp

```
void quick_rec(int a[], int left, int right)
{
    if (left < right) // left+10 <=right i boka
    {
        int pivot = a[(left+right)/2]; // enklere pivoteringsvalg
        int i=left, j=right;
        for (;;)
        {
            while (a[i] < pivot) {i++; } // funnet 1. fra venstre >= pivot
            while (pivot < a[j]) {j--; } // funnet 1. fra hoyre <= pivot
            if (i<j)
                std::swap(a[i], a[j]);   // pa riktig side av pivot
            else
                break;
        }
        quick_rec(a, left, i-1);
        quick_rec(a, i+1, right);
    }
}
```

Detaljene med denne implementeringen blir:

```
17, 14, 5, 7, 12, 1, 16, 29, 13, 4, 8, 18, 22, 2, before quick
quick(a, 0, 13)
pivot: a[6]=16
bytter a[13]=17 og a[0]=2
bytter a[10]=16 og a[6]=8
bytter a[10]=29 og a[7]=16
bytter a[9]=16 og a[7]=4
2, 14, 5, 7, 12, 1, 8, 4, 13, 16, 29, 18, 22, 17, i=9, j=9
```

7.6 Sammendrag

- $O(n^2)$ algoritmer
  - Enkle å programmere
  - For mindre sorteringsoppgaver
- $O(n \log n)$ algoritmer
  - Også noen av disse kan ha worst case
  - Nødvendig ved store datamengder
- Quicksort average case er $O(n \log n)$. Dette er lett å se hvis man antar at filstørrelsen er en potens av 2, og at pivotelementet alltid er det midterste i subarrayene. En kan da telle opp antall sammenlikninger i hver (sub)array multiplisert med arraystørrelsen.
- std::sort *is generally quicksort* ([2], side 310).
- For n<20 er quicksort dårligere enn insertion sort (som jo er $O(n^2)$)

7.7 Oppgaver

1. Lag et C++ prosjekt
2. Opprett en vektor eller array med n=20-30 heltall
3. Sorter tallene med std::sort
4. Sorter tallene med heapsort:
   - Bruk en std::priority_queue
   - Sett inn tallene i denne
   - Skriv ut det minste/største n ganger
5. Sorter tallene ved å bruke et binært søketre
6. Sorter tallene med mergesort

Kapittel 8

Hashing

8.1 Hashing Introduksjon

- Hvorfor hashing? O(1) søk
- Hashing metoder
- Hashing kodeeksempel med stl
- set eller unordered_set (map eller unordered map)?
- Ide
- Definisjoner - hva er hashing
- Hashfunksjoner
- Anvendelser
- Eksempler
  - når vi har en ideell situasjon
  - når vi har en virkelig situasjon

8.1.1 Ide

Et optimalt søk er et søk hvor vi finner igjen det vi leter etter uten å lete noe i det hele tatt. Vi kan tenke oss situasjoner som å lete etter et navn i telefonkatalogen, en vare på et lager eller en post på en fil. Det beste vi kan oppnå er direkte oppslag, altså et O(1) søk. Det beste vi har sett hittil er et O(log n) søk.

Vi vet nå at sortering og søking henger nøye sammen - sortering er viktig fordi søking på en sortert tabell/fil går vesentlig raskere enn søking på en usortert tabell/fil. Og søking er en operasjon som forekommer ekstremt ofte.

Paradokset med hashing er at søking etter en gitt post går fortest på en tabell/fil som tilsynelatende ser ut som et kaos.

0	1	2	3	4	5	6	7	8	9	10
11	1	2		15	5		7			10
				4			40			
				26			18			

Figur 8.1: Eksempel på hashfil. Kun nøkkelverdiene til de ulike postene er markert. Hvordan kan vi finne igjen en post på denne filen ved første forsøk?

8.1.2 Definisjoner - hva er hashing

Hashing er søking, men har flere anvendelser enn bare søking. Hashing brukes både til gjenfinning, innsetting og sletting av poster (data). Hashing er en felles betegnelse for en type tabell/filstrukturer (nesten kompakte filer), innsetting og sletting. Et sentralt begrep er en hashfunksjon.

Hashfunksjon

En hashfunksjon er en ideell søkefunksjon som i de fleste tilfeller gir direkte oppslag på en gitt nøkkel på en hashtabell. Dvs. at den kan finne igjen et gitt element (nøkkel) i en eneste operasjon. Hashfunksjonen krever at postene lagres i array eller på fil, med direkte sammenheng mellom array-indeksen og nøkkelen.

Men er ikke dette praktisk umulig? Tenk bare på problemene med innsetting i en array. Joda, men noen ganger er det nesten mulig! La oss nå se på definisjonen av en hashfunksjon.

En hashfunksjon er en funksjon som transformerer en nøkkel til en indeks i en tabell eller på fil. En perfekt hashfunksjon er en hashfunksjon som ikke gir kollisjoner.

Et praktisk eksempel kan være en hashfunksjon som transformerer et delenummer til en bestemt plass på lageret.

Ideelt skal det ikke finnes to nøkkelverdier som gir samme indeks (sml. entydig funksjon i funksjonslære). Strategien blir at vi prøver å lage en best mulig hash-funksjon og behandler unntakene etterpå.

Hashtabell og hashfil

En hashfil er en nesten kompakt direkte fil. Den har en bestemt størrelse. Fyllingsgraden kan variere, men bør være noenlunde konstant. En hashfil er organisert i et bestemt antall hjemmeadresser eller blokker, og med et fast antall poster i hver blokk. Tilsvarende er en hashtabell en tabell eller array med bestemt størrelse: Et antall blokker/kolonner/skuffer/buckets multiplisert med antall plasser i hver skuff. Det er lett å se at hashtabellen på figur 8.1 har 11 skuffer med plass til 3, total størrelse 11*3 og fyllingsgrad 1/3.

Sammen med denne hashtabellen har vi benyttet hashfunksjonen

```
int Hash(long nokkel)
```

```
{
   return (nokkel % antalladresser);
}
```

Vi ser blant annet av denne hashtabellen at det er mange tomme plasser, og at flere poster med ulik nøkkelverdi har havnet i samme blokk. Alle postplasseringene er beregnet ut fra hashfunksjonen ovenfor. Den beregner altså adressen til en skuff i en tabell. Legg merke til at antalladresser benyttes i hashfunksjonen ovenfor. Vi må derfor ha en fast størrelse på denne tabellen.

8.2 Krav til en god hashfunksjon

Følgende tre krav er fundamentale for en god hashfunksjon. Den skal

1. returnere et heltall mellom 0 og antalladresser-1
2. være rask
3. spre verdiene jevnt utover

Vi studerer disse kravene på noen hashfunksjoner hvor nøkkelen er et 6-sifret studentnummer på formatet ÅÅxxxx, eks. 97xxxx, 98xxxx.

```
long hash2(long key) { return key/10000; } // Dårlig
long hash3(long key) { return key%10000; } // Bedre
long hash4(long key) { return key%1000; }  // Bedre?
```

Alle disse eksemplene er tabell-eksempler, men filer er like relevant. Vi returnerer til figur 8.1 og spør: Hva skjer når vi skal sette inn en post med nøkkelverdien 37 på denne hashfila? Situasjoner som dette kan inntreffe, og er bakgrunnen for kollisjonsbehandling.

8.3 Kollisjonsbehandling

En kollisjon innebærer at to poster får samme hjemmeadresse. Kollisjonsbehandling omfatter strategier for å takle kollisjoner. I hovedsak har vi to metoder: Rehashing og kjeding. Vi kommer tilbake til dette ganske snart. I denne omgang er det nok å legge merke til at problemet først oppstår når hjemmeadressen er full.

Ved hashing på fil beregner vi altså hjemmeadressen til en blokk i stedet for adressen direkte i en tabell. Vi fortsetter med samme hashtabell og samme hashfunksjon som ovenfor.

Øving

Som en liten repetisjon av modulo-funksjonen og hashfunksjonen ovenfor kan dere gjøre følgende øving: Hvordan blir tabellen etter at postene med disse nøklene er satt inn: 16, 28, 39, 17, 49.

0	1	2	3	4	5	6	7	8	9	10
11	1	2		15	5		7			10
				4			40			
				26			18			

8.3.1 Kollisjon - eksempel

Etter at disse postene er satt inn, er noen blokker fulle. Vi har 16%11=5, 28%11=6, 39%11=6, 17%11=6, 49%11=5.

0	1	2	3	4	5	6	7	8	9	10
11	1	2		15	5	28	7			10
				4	16	39	40			
				26	49	17	18			

Hvis vi nå ønsker å sette inn poster med nøklene 27, 6 eller 29 får vi et problem. Dette problemet er en kollisjon, og disse postene kalles overløpsposter. Hvordan løser vi det?

Rehashing

En rehash funksjon er en hashfunksjon som brukes på en indeks/hjemmeadresse og returnerer en annen indeks. Vi husker at en hash funksjon brukes på en nøkkel og returnerer en indeks. En god hashfunksjon skal spre postene jevnt utover. En god rehash funksjon skal gi færrest mulig iterasjoner (når hjemmeadressen er full) og gjentatt bruk av rehash funksjonen skal føre til at flest mulig hjemmeadresser dekkes. En rehash funksjon som har denne egenskapen er

```
int rehash1(int i)
{
   return (i+c) % antalladresser ;
}
```

hvor c er en konstant som er primisk med antalladresser.

Åpen adressering

Den enkleste løsningen er å søke etter nærmeste blokk med ledig plass. I dette tilfellet er det den niende blokken (hjemmeadresse 8) for alle tre nøklene. Selv om hashfunksjonen beregner ulik hjemmeadresse for disse tre nøklene, fører åpen adressering til at de plasseres i samme blokk. Dette kalles primær klasing. Primær klasing er altså definert ved at poster med ulik hjemmeadresse får samme overløpsbane. Vi kan unngå åpen adressering ved å bruke en annen rehash funksjon, for eksempel

```
int rehash2(int i, int j)
{
   return (i+j) % antalladresser ;
}
```

hvor j er antall ganger rehash-funksjonen er brukt. Denne metoden gir imidlertid sekundær klasing - det vil si at poster med samme hjemmeadresse får samme overløpsbane. Dette kan unngås med dobbelthashing.

Dobbelthashing

Problemet er altså kollisjon når hjemmeadressen er full. Vi trenger da en algoritme for å finne en annen plass ved innsetting, og en samsvarende algoritme til

bruk ved søking (vi har ikke lenger en ideell situasjon med direkte oppslag). La oss for anledningen kalle funksjonen finnplass().

En annen og bedre løsning enn de som er skissert ovenfor, er å søke med forskjellig steglengde etter ledig blokk, slik at overløpspostene blir spredd. Dette kan gjøres ved å beregne steglengde ved funksjonen

```
int dobbelthash(long nokkel)
{
   return (1 + nokkel%(antalladresser-1));
}
```

i den samme finn plass-funksjonen. Denne metoden løser problemene med klasing. Vi skal studere hvordan denne algoritmen plasserer postene ovenfor (27, 6, 29) på figuren før vi går over på selve algoritmen.

0	1	2	3	4	5	6	7	8	9	10
11	1	2		15	5	28	7			10
				4	16	39	40			
				26	49	17	18			

Vi har hash(27) = 27%11 = 5 og dobbelthash(27) = 1 + 27%10 = 8. Det er fullt i skuff 5, og vi skal lete etter en ledig plass på en smartere måte enn de påfølgende skuffene i tur og orden. Skuffene 6 og 7 er også fulle, så hvis vi gjorde det så enkelt, ville nøkler hjemmehørende både i skuff 4, 5, 6 og 7 hope seg opp i skuff 8 (vi indekserer skuffene fra 0 og oppover).

Vi søker heller med en steglengde fra dobbelthashfunksjonen. Den første skuffen vi sjekker, blir da
(hash(27)+dobbelthash(27))%11 = (5+8)%11 = 2
og her er det ledig.

Videre: hash(6)=5, altså den samme hjemmeadressen som 27. Men dobbelthash(6) = 1 + 6%10 = 5. Den første skuffen vi sjekker, blir da skuff 10, og her er det ledig.

hash(29)=7 og dobbelthash(29)=10. Det er fullt i skuff 7, og når vi går 10 plasser videre (med modulo) kommer vi til skuff 6. Når vi går ytterligere 10 plasser videre, kommer vi til skuff 5. Slik fortsetter det. Overløpsbanen til 29 blir altså (inkludert hjemmeadressen) 7, 6, 5, 4, 3.

Dobbelthash-funksjonen skal altså ikke brukes direkte på nøkkelen, den finner ikke en ny blokk for overløpsposten direkte. Den brukes derimot for å beregne steglengden i letingen etter en ledig blokk.

8.3.2 Kort sammendrag

- Åpen adressering er veldig dårlig for hashfiler med fyllingsgrad større enn 0.75, spesielt for søk som resulterer i at man ikke finner posten man søker etter.

- Primær klasing gir vesentlig større bidrag til søketiden enn sekundær klasing.
- For en full tabell med n elementer tar suksess-søk i gjennomsnitt ca ln (n+1) - 0.5 oppslag,
- og fiasko-søk tar i gjennomsnitt ca (n+1) / 2 oppslag ([1]).

8.4 Kjeding

Når vi løser kollisjonsbehandling med rehashing, åpen adressering eller dobbelthashing, brukes hovedområdet på tabellen/filen til overløpsposter. I stedet for å bruke hovedområdet på hashfilen til overløp, kan vi benytte en peker i hver blokk. Ved overløp får vi en pekerkjede av blokker (buckets).

8.5 std::unordered_set

I standard template library er kjeding benyttet. Eksemplet nedenfor (hashtest.cpp) viser bruk av std::unordered_set og std::unordered_multiset.

Listing 8.1: hashtest.cpp

```
#include <string>
#include <hash\_map>
#include <unordered\_set>
#include <iostream>
using namespace std;

// Lager en egen klasse for bruk med std::hash
struct Test {
    int key;
    std::string s;
    // Overlaster operator == for a lage unordered_set<Test>
    bool operator == (const Test& t2) const { return key == t2.key; }
};
```

Denne delen av eksemplet viser hvordan vi kan lage en egen hashfunksjon. Vi utvider hash til å gjelde typen Test. Siden hash allerede fins i std, må vi angi namespace for å kunne gjøre noe med den. Vi definerer nå standard hashfunksjon for Test objekter i stl.

Vi lager et funksjonsobjekt (se [2] §1.6.4). Funksjonstemplate krever operator == for de aktuelle objektene. Vi kan alternativt sende inn objekter og sammenligningsfunksjon.

Vi lager en klasse uten data men med en medlemsfunksjon og sender inn en instans av denne som parameter. Da får vi sendt inn funksjonen som parameter implisitt. Dette er et funksjonsobjekt. Vi benytter altså operatoroverlasting i stedet for funksjonsnavn.

```
namespace std {
    template<>
    class hash<Test>
    {
    public:
```

```
        size_t operator() (const Test& t) const
        {
            return t.key % 11; // 11 er et selvvalgt primtall
        }
        bool operator() (const Test& t1, const Test& t2)
        {
            return t1.key == t2.key;
        }
    };
}
void test_hash()
{
    // Lager et hashobjekt. Men hva er det? Det ser ut som en funksjon
    // implementert for int, string, ..
    // parameteren er det vi oppfatter som en key
    hash<int> h;

    size_t i = h(1113335557);
    cout << "hash(1113335557) = " << i << endl;

    hash<Test> ht;
    Test t{1113335557, " blabla"};
    i = ht(t);
    cout << "egen hash verdi " << i << endl;
}
```

Listing 8.2: unordered set

```
void test_unordered_set()
{
    hash<Test> ht;
    unordered_set<Test> us;
    unordered_set<Test>::iterator usit;
    pair<unordered_set<Test>::iterator, bool> par;

    Test t;
    // Tilordner verdiene 0...15 til key og tester
    // den egne hashfunksjonen
    cout << "Tester egen hashfunksjon" << endl;
    for (auto i=0; i<15; i++) {
        t.key = i;
        cout << "hash ht(" << i << ") = " << ht(t) << endl;
    }

    // Setter inn noen verdier i unordered set
    t.key = 12; t.s = "tolv";
    us.insert(t);
    t.key = 14; t.s = "fjorten";
    us.insert(t);

    // Duplikater gar ikke i unordered_set, vi ma bruke
    // unordered_multiset
    t.key = 14; t.s = "fjorten_b";
    int m = us.size();
    //
    par = us.insert(t);
    int n = us.size();
    cout << t.s << " satt inn? " << boolalpha << (n-m==1) << endl;
    // enklere med par:
    cout << t.s << " satt inn? " << boolalpha << par.second << endl;

    // Vi skal finne igjen objektet med key 12
    // Det er da ikke interessant hva objektets string verdi
```

```
    // faktisk er under soket, siden 12 er key
    t.key = 14;
    usit = us.find(t);
    t = *usit;
    cout << "post funnet: " << t.key << " " << t.s << endl;
}
```

Listing 8.3: unordered multiset

```
void test_unordered_multiset()
{
    unordered_multiset<Test> ums;
    unordered_multiset<Test>::iterator umsit;
    pair<unordered_multiset<Test>::iterator, bool> par;

    Test t1, t2;
    // setter inn noen verdier i unordered_multiset
    t1.key = 12; t1.s = "tolv";
    ums.insert(t1);
    t2.key = 14; t2.s = "fjorten";
    ums.insert(t2);

    // Duplikater gar i unordered_multiset
    int m = ums.size();
    t2.key = 14; t2.s = "fjorten_b";
    ums.insert(t2);
    int n = ums.size();
    cout << t2.s << " satt inn? " << boolalpha << (n-m==1) << endl;

    // Vi skal finne igjen objektet med key 12
    // Det er da ikke interessant hva objektets string verdi
    // faktisk er, siden 12 er key
    t2.key = 14;
    umsit = ums.find(t2);

    t2 = *umsit;
    cout << t2.key << " " << t2.s << endl;
    cout << "Hvor mange keys = " << t2.key << "? count = " << ums.count(t2) << endl;

    cout << "load_factor " << ums.load_factor() << endl;;
    cout << "size " << ums.size() << endl;;
    cout << "bucket_count " << ums.bucket_count() << endl;

    cout << endl << "tenk litt over implementeringen i stl:" << endl;
    cout << "bucket_size for key " << t1.key << " = " << ums.bucket(t1) << endl;
    cout << "bucket_size for key " << t2.key << " = " << ums.bucket(t2) << endl;

    cout << "setter inn 17 poster" << endl;
    for (int i=13; i<30; i++)
        ums.insert(Test{i, "blabla"});
    cout << "load_factor = " << ums.size() << "/" <<ums.bucket_count() << " ="
<< ums.load_factor() << endl;;
    cout << "size " << ums.size() << endl;;
    cout << "bucket_count " << ums.bucket_count() << endl;
}
int main(int argc, char *argv[])
{
    cout << "********** tester hash for int **********" << endl;
    test_hash();
    cout << endl << "********** test unordered set  **********" << endl;
    test_unordered_set();
    cout << endl << "********** test unordered multiset **********" << endl;
    test_unordered_multiset();
```

```
}
```

8.6 Sammendrag

Vi har sett på to hovedmetoder for hashing og implementering i C++ stl, std::unordered_set og std::unordered_multiset.

Kapittel 9

Grafer

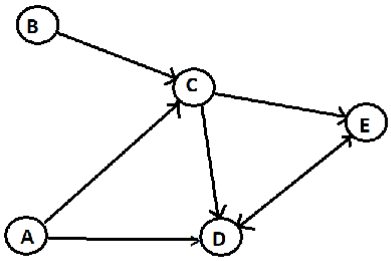
9.1 Grafer Introduksjon

Vi skal begynne med noen definisjoner, og se litt på implementering. Eksemplene er til dels uferdige, men gir noen ideer om hva som er likheter og forskjeller mellom trær og grafer generelt, spesielt når det kommer til traversering.

- Definisjoner
- Graf som lenket liste
- Node og Kant (Edge) klasser
- Traversering
  - Dybde først traversering
  - Bredde først traversering
- Dijkstras algoritme for en dynamisk graf
- A* algoritmen
- Minimum spenntre
- **Implementering!**

9.1.1 Definisjoner

- graf G
- node v_i eller n_i, $V = \{v_i\}$
- En kant (edge) e_i med vekt eller kostnad c_i er en forbindelse mellom to noder. $E = \{e_i\}$
- $G = \{V, E\}$. En graf er en datastruktur som består av
  - V - en mengde av noder (hjørner/vertices)
  - E - en mengde av kanter



Figur 9.1: En rettet graf med noder og kanter.

- En node kan representeres ved et punkt i planet - (x,y). Nodene kan f.eks. nummereres A, B, C, ...
- En kant er definert ved et par av noder. Nummerering f.eks: (A,B), (A,C)
- Nodene kan også ha strenger og tall som navn

Rettet graf

En rettet graf (digraph) er en graf hvor kantene er retningsbestemt av notasjonen - sml. med en vektor. (A,B) eller <A,B> er en kant fra A til B og kan tegnes som en linje med pil på B. Vi skriver altså kantene slik: (A,B), (A,C).

Eksempel: Polygonet ABC er en rettet graf som består av en mengde noder, $V = \{A, B, C\}$ og en mengde kanter, $E = \{(A, B), (B, C), (C, A)\}$. Flere begreper:

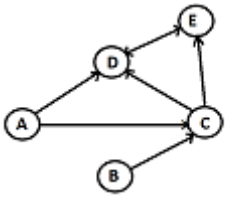
- graden til en node
- inngrad, utgrad
- rettet og urettet graf
- sykluser (cycles)
- asyklisk graf
- DAG - directed acyclic graph
- path, lengde

Eksempel på rettet graf:

- $G = \{V, E\}$
- $V = \{A, B, C, D, E\}$.
- $V = \{(A, C),\ (A, D),\ (B, C),\ (C, D),\ (C, E),\ (D, E),\ (E, D)\}$.
- Tegn opp nodene som sirkler med navn inni og kantene som linjer med pil mellom nodene.

Topologi $\neq$ geometri

Det er flere likeverdige måter å tegne opp en graf på. For eksempel er grafen på figur 9.2 den samme som grafen på figur 9.1.



Figur 9.2: En rettet graf med noder og kanter.

- Er dette en rettet asyklisk graf?
- Hva med lengste path her?

9.1.2 Matriserepresentasjon

Vi kan representere den rettede grafen fra figur 9.1 ved hjelp av en nabomatrise **M**:

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

Her er $\mathbf{M}_{ij} = 1$ dersom det er en kant fra node i til node j, og 0 ellers.

Boolsk matrisemultiplikasjon

Vi kan finne ut om en node kan nås fra en annen node via 2 kanter ved følgende resonnement: Dersom det fins en vei av lengde 2 fra A til E, må det finnes kanter - veier av lengde 1 - fra A til B og fra B til E, og/eller fra A til C og fra C til E, og/eller fra A til D og fra D til E. Eventuelt de to spesialtilfellene AA og AE og AE og EE. Dette kan vi sette opp med boolske operatorer slik:

$$(AA\&\&AE) \,||\, (AB\&\&BE) \,||\, (AC\&\&CE) \,||\, (AD\&\&DE) \,||\, (AE\&\&EE) \quad (9.1)$$

Gjør vi denne operasjonen på alle elementene i matrisen, får vi en matrise med alle forbindelser av 2 kanter.

Hvis vi sammenligner med vanlig matrisemultiplikasjon $m_{ij} = \Sigma_{k=1}^{n} m_{ik} \cdot m_{kj}$, ser vi at multiplikasjon er erstattet med boolsk AND og summering er erstattet

med boolsk OR.

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

Man kan fortsette å gjøre samme operasjon med denne og opprinnelig nabomatrise, og få forbindelser over 3 kanter. Se Warshall's algoritme for transitive closure.

Naboliste representasjon

En annen naboliste representasjon for grafen fra figur 9.1 er

A	CD
B	C
C	DE
D	E
E	D

Vi kan navngi nodene med tall som starter på 0:

0	2, 3
1	2
2	3, 4
3	4
4	3

Da trenger vi bare tallene (referansene, indeksene, pekerne) i høyre kolonne.

9.1.3 Node, kant og graf klasser

En graf er en datastruktur som består av en mengde noder og en mengde kanter. Enkle klasser for dynamisk representasjon:

Listing 9.1: Node, Kant og Graf

```
struct Kant;
struct Node {
    char m_navn;
    bool m_besokt;
    std::list<Kant> m_kanter;

    Node(char navn) : m_navn(navn), m_besokt(false) { }
    //Node* finn_node(char navn);
    void settinn_kant(const Kant &kant);
    void dybdeforst();
};

struct Kant {
    float m_vekt;
    Node* m_tilnode;
    Kant(float vekt, Node* tilnode) : m_vekt(vekt), m_tilnode(tilnode) { }
    bool operator > (const Kant& k) const { return m_vekt > k.m_vekt; }
};
```

```
struct Graf {
    std::list<Node*> noder;
    Graf() { }
    Node* finn_node(char navn);
    void settinn_node(char navn);
    void settinn_kant(char fra_navn, char til_navn, float vekt);
    void dybdeforst(char navn);
    float mst();
};
```

Vi bruker std::vector til å lagre noder og kanter.

9.1.4 Traversering

Å traversere en datastruktur betyr å besøke alle nodene systematisk. Vi ønsker å traversere datastrukturen på en måte som er avhengig av eller viser selve strukturen. Vi skiller mellom to hovedtyper av traversering: Bredde-først og dybde-først traversering. Vi husker for binære søketrær at vi har lært tre dybde-først traverseringer og en bredde-først traversering (oppgave). Vi husker også at at vi har benyttet rekursjon til dybde-først traversering.

Traversering i grafer er vanskeligere enn for eksempel i binære søketrær. Det er ingen selvsagt første node i grafen (selv om det er en i implementeringen presentert ovenfor). Videre er det heller ikke selvsagt hvilken node N2 som er etterfølgeren til en node N1. Til sist kan en node ha flere enn en forløper, i motsetning til en node i et tre.

Ved traversering i grafer er det ofte lønnsomt å ha et ekstra logisk felt, m_besoekt, i hver node.

Dybde-først traversering

- Generalisering av preorder traversering
- Trenger bool m_besoekt; // hjelpevariabel til traversering
- Besøke = visit = color
- oppgave

La oss først se hva som skjer med følgende kode:

Listing 9.2: Dybde-først traversering som ikke alltid virker

```
void Node::dybdeforst()
{
      for (auto kant:m_kanter)
      {
         auto nodepeker = kant.m_tilnode;
         nodepeker->dybdeforst();
      }
   }
}
```

Dette kan gi en fornuftig traversering så lenge vi har en asyklisk graf. I en syklisk graf vil vi kunne oppleve evig løkke (prøv)! Dybde-først traversering for

en graf er generelt vanskeligere enn for et tre, og vi trenger å holde styr på hvilke noder vi allerede har besøkt under traverseringen. En foreløpig algoritme for dybde-først traversering blir:

Listing 9.3: Dybde-først traversering

```
void Node::dybdeforst()
{
    if (!m_besokt)
    {
        cout << m_navn;
        m_besokt = true;
        for (auto kant:m_kanter)
        {
            auto nodepeker = kant.m_tilnode;

            if (!nodepeker->m_besokt) // Alle kanter har en tilnode
                nodepeker->dybdeforst();
        }
        m_besokt = false;
    }
}
```

Prøv denne også!

Øving

Følgende er hentet fra eksamen høsten 2019, oppgave 4b:

```
// Lage en testgraf G={V,E}
// hvor V={A, B, C, D, E}
// og E = {AB(1.0), AC(2.0), BC(2.0), CD(3.0), DE(1.0), AE(5.0), CE(4.0)
```

Når vi kaller dybdeførst-funksjonen fra **a**, blir utskriften **abcdeecdeee**, også hvis vi legger til en kant **ea** som gir en syklisk graf. Vi ser at de første fem bokstavene representerer en sti så langt det er mulig å komme fra a, men resten av utskriften kan være vanskelig å tyde.

9.1.5 Topologisk sortering

En topologisk sortering er en ordning av nodene i en rettet graf som tar hensyn til retningen på kantene. Hvis det er en kant AB i en graf, kommer node A før node B i en topologisk sortering.

Øving

Skriv en funksjon som gjør en topologisk sortering for denne grafstrukturen.

9.1.6 Bredde-først traversering

Nedenfor presenteres en enkel (ikke helt ferdig) algoritme som bruker en kø. breddeforst() besøker alle nodene som kan nås fra den kallende noden. Den etterlater alle nodene som besøkt uten endringer, og en må derfor traversere hele den interne strukturen og tilbakestille besøkt.

Listing 9.4: Bredde-først traversering

```
void Node::breddeforst()
{
   std::queue<Node*> q;

   q.push(this);
   do {
      auto* ut = q.front();
      q.pop();
      if (!ut->m_besokt)
      {
         ut->m_besokt = true;
         std::cout << ut->m_navn;

         for (auto kant:ut->m_kanter)
         {
            auto nodepeker = kant.m_tilnode;
            q.push(nodepeker);
         }
      }
   } while (!q.empty());
}
```

- Også her blir utskriften noe mangelfull uten endringer i algoritmen.
- Debugging av funksjonen er nyttig og viktig.

9.2 Dijkstra's algoritme for en lenket graf

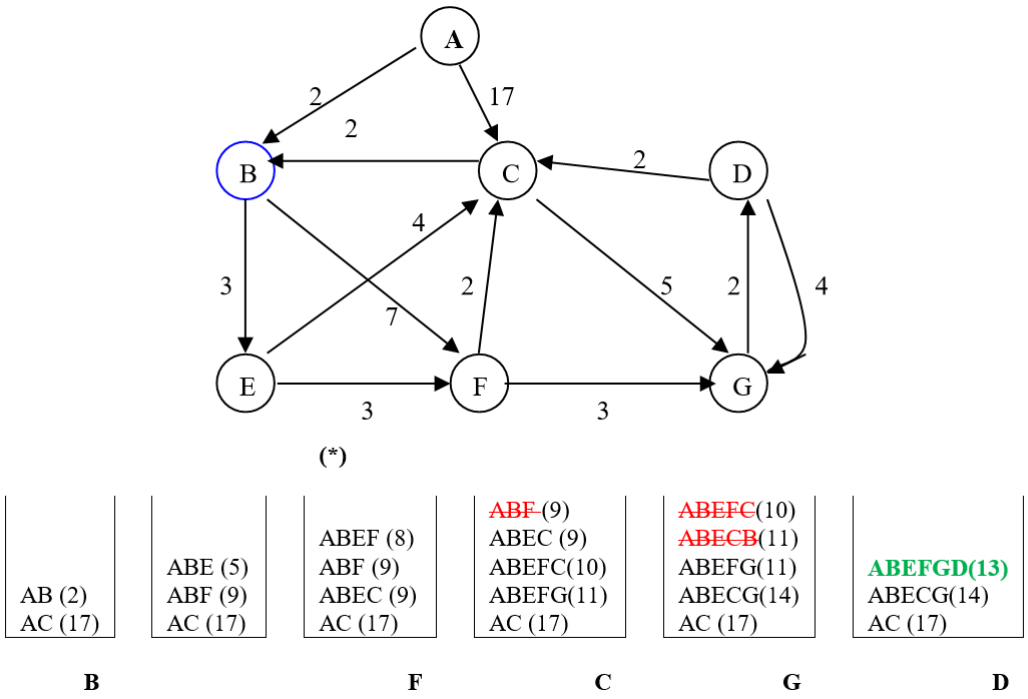
Ideene i Dijkstras algoritme er:

- Vi skal finne korteste vei fra node **s** til node **t** i en vektet graf.
- Vi benytter en klasse Veg som er en samling av kanter.
- Initielt kjenner vi kun korteste avstand fra **s** til **s** , nemlig avstanden 0, veien AA. Fra den til enhver tid minste veien som går ut fra A, lager vi nye veier ved å skjøte på kantene fra sluttnoden til den minste veien.
- I algoritmen benytter vi en prioritetskø av veier med kostnad.

Som et eksempel ser vi på hvordan vi finner korteste vei fra A til D - kantene AB(2) og AC(17). Vi

- setter disse veiene inn i en prioritetskø,
- tar ut hittil korteste vei fra A - AB(2),
- skjøter på med BE og BF og setter veiene ABE(5) og ABF(9) inn igjen i prioritetskøa.
- Fortsetter prosessen.

Prioritetskø underveis er vist på figur 9.3.



Figur 9.3: Dijkstra's algoritme med prioritetskø underveis.

Situasjonen ved (*)

Vi har to mulige veier til F: ABF (9) og ABEF(8). Vi vet her at ABEF(8) er korteste vei fra A til F.
La AF_k betegne korteste vei fra A til F, og anta at $AF_k \neq ABEF$. Med andre ord antar vi at det finnes en annen korteste vei enn ABEF(8). Men da må den korteste veien gå om ABF(9), ABEC(9) eller AC(17) og bestå av en eller flere kanter i tillegg. Siden vi kun har positive vekter, er dette umulig. Alle andre veier fra A til F må være lengre enn ABEF(8), og dermed er AF_k = ABEF(8) den korteste vei.

Når vi har funnet korteste veien til en node, markeres denne noden som besøkt. Når vi senere får med veier å gjøre som slutter på denne noden, vet vi ikke trenger behandle disse. Derfor vrakes

- ABF (etter at vi har satt inn C) fordi ABEF(8) er kortere,
- ABEFC(10) etter at vi har satt inn G fordi ABEC(9) er kortere,
- ABECB(11) etter at vi har satt inn G fordi AB(2) opplagt er kortere (og B er besøkt).

Det samme gjelder for C. For å komme *fra* A, må vi gå en av de påbegynte veiene. For å komme til C på en annen måte enn ABEC(9), må vi først gå en vei med lengde større enn 9, og i tillegg en eller flere kanter med positiv vekt.

Altså er ABEC(9) den korteste veien fra A til C. Når vi tar ut en vei som ender på D fra prioritetskøa, er vi ferdige.

Dijkstra's algoritme er et eksempel på en **grådig algoritme**. Se [2], figur 9.29-9.30-9.31 for pseudokode til Dijkstra's algoritme.

9.3 A* algoritmen

A* algoritmen ligner Dijkstra's algoritme. Den brukes også for å finne korteste sti fra start til mål. Den brukes gjerne på grafer hvor nodene har koordinater i 2d og vekten på kantene er vanlig Euklidsk distanse. I tillegg brukes en såkalt heuristikk (som måler avstanden fra en node til sluttnoden). Det fins som vanlig en del ressurser på internett:

- `https://no.wikipedia.org/wiki/A*`
- `http://qiao.github.io/PathFinding.js/visual/`
- `http://ashblue.github.io/javascript-pathfinding/`
- `https://www.youtube.com/watch?v=KNXfSOx4eEE`
- `https://www.youtube.com/watch?v=-L-WgKMFuhE`

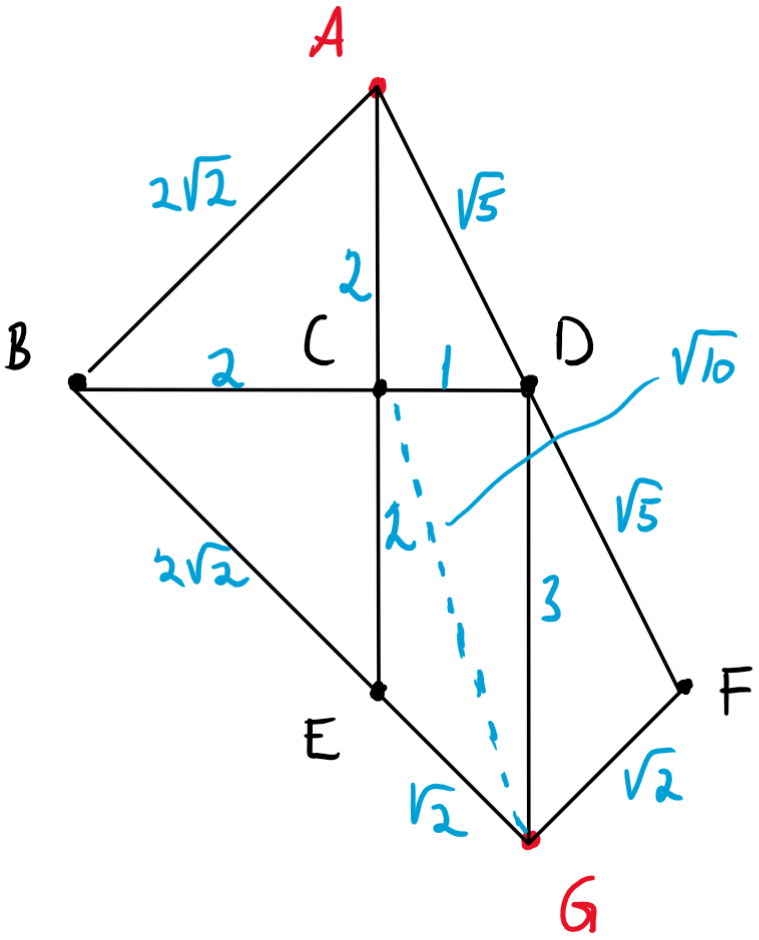
Vi skal her studere et lite eksempel i detalj. På figur 9.4 har vi en graf med sju noder A-G og hvor kantenes vekt er avstanden mellom nodene. Vi skal først se hvordan Dijkstra's algoritme virker på denne grafen. Deretter skal vi bruke A* algoritmen. Vi lar A=start og G=slutt og starter først med Dijkstra's algoritme.

9.3.1 Eksempel, Dijkstra's algoritme

Vi bruker to desimaler, slik at $\sqrt{2} = 1.41$, $\sqrt{5} = 2.24$, $2\sqrt{2} = 2.82$, $\sqrt{10} = 3.16$, $3\sqrt{2} = 4.24$.

1. Dijkstra's algoritme starter fra A, og legger til stier fra A (første gang stier som kun består av *en* kant) fra A i en prioritetskø. Når dette er gjort, blir prioritetskøa som i første kolonne i tabellen nedenfor.
2. Korteste sti (AC, 2) tas ut fra prioritetskøa. Vi legger til kanter fra C: CB(2), CD(1), CE(2). Dette gir nye stier ACB(4), ACD(3), ACE(4). Disse legges på prioritetskøa, se kolonne 2. Hvilken sti som kommer først og sist av ACB(4) og ACE(4) kommer an på implementeringen.
3. AD tas ut (kolonne 2), og ADF og ADE legges til. Her kunne også ADC vært lagt til, men dette fører bare til en lengre stil fra A til C (C er allerede besøkt).
4. Fra fjerde kolonne tas ACE(4) ut. ACEG legges til (men ikke ACEB, av samme grunn som i forrige punkt).
5. Fra femte kolonne tas ACB ut. Siden E allerede er besøkt (algoritmen har funnet korteste sti fra A til E), legges ingen ny sti til prioritetskøa.
6. Fra sjette til sjuende kolonne tas ADF ut, og ADFG legges til.
7. Når ADG tas ut terminerer algoritmen. Denne stien er den første stien til sluttnoden som tas ut fra prioritetskøa.

	AD(2.24)	AB(2.82)	ACE(4)	ACB(4)		
	ACD(3)	ACE(4)	ACB(4)	ADF(4.46)	ADF(4.46)	ADG(5.23)
AC(2)	AB(2.82)	ACB(4)	ADF(4.46)	ADG(5.23)	ADG(5.23)	ACEG(5.41)
AD(2.24)	ACE(4)	ADF(4.46)	ADG(5.23)	ACEG(5.41)	ACEG(5.41)	ABE(5.64)
AB(2.82)	ACB(4)	ADG(5.23)	ABE(5.64)	ABE(5.64)	ABE(5.64)	ADFG(5.87)



Figur 9.4: A* algoritmen.

9.3.2 Eksempel 1, A* algoritmen

Vi bruker en enkel heuristisk funksjon $h(y) = |y - s|$ hvor $|y - s|$ er avstanden mellom node y og startnoden s. I stedet for kun å legge til en ny kant xy og denne kantens vekt i hvert steg, legger vi også til $h(y) = |y - s|$. Vi bruker notasjonen $|\mathbf{xy}|$ når den heuristiske funksjonen returnerer lengden av en kant, og når vi setter inn denne beregnede avstanden.

Prioritetskøa blir nå

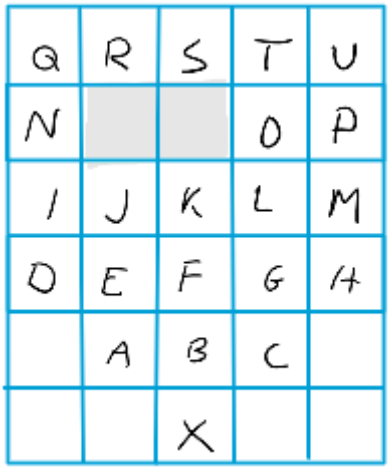
		ADG+|GG| =5.24
	AD+|DG|=5.24	ACE+|EG|=5.41
	ACE+|EG|=5.41	ADF+|FG|=5.89
AC+|CG|=5.16	ACD+|DG|=6.00	ACD+|DG|=6.00
AD+|DG|=5.24	AB+|BG|=7.06	AB+|BG|=7.06
AB+|BG|=7.06	ACB+|BG|= 8.28	ACB + |BG| = 8.28

- Det er lett å se forskjellen på Dijkstra's algoritme og A* algoritmen når vi sammenligner første kolonne i tabellene. A* algoritmen vil her velge ut samme kant, AC, som tas ut.
- Legg merke til at det kun er AC og ikke AC+|CG| som brukes til å bygge nye stier.
- A* algoritmen bygger altså nye stier ACB(4), ACD(3), ACE(4) på samme måte som Dijkstra's algoritme. Deretter brukes den heuristiske funksjonen for hver av de nye nodene B, D, E. I kolonne to settes inn ACB+|BG|=8.28, ACD+|DG|=6, ACE+|EG|=5.41
- Stien som tas ut fra prioritetskøa i kolonne to er AD. Nye stier blir ADG, ADC og ADF. Men ADC settes ikke inn, siden C allerede er besøkt. ADG settes inn med 0 i tillegg fra den heuristiske funksjonen.
- Når korteste sti tas ut fra prioritetskøa i kolonne tre, er sluttnoden nådd.

I dette eksemplet gjør altså A* algoritmen færre iterasjoner enn Dijkstra's algoritme. Den ekstra kostnaden er å beregne *h(y)* for hver ny sti som settes inn i prioritetskøa.

9.3.3 Eksempel 2, A* algoritmen

på figur 9.5 nedenfor viser et annet eksempel.



Figur 9.5: Finne korteste vei fra X til S med A* algoritmen. Skraverte ruter representerer hindringer.

Nodene er her markert med ruter og navn. Alle rutene har lengde 1, og vi antar at nodenes koordinater er midt i hver rute. Vi kan måle avstanden mellom nodene på flere måter.

1. Vanlig Euklidsk avstand mellom midtpunktene: |XA|=$\sqrt{2}$, |XB|=1
2. Samme lengde mellom alle nabonoder: |XA| = |XB| = 1
3. Ved å bruke *Manhattan distance* |XA| = |XB| = 2 (det er kun lov til å gå horisontalt og vertikalt)

Her skal vi måle avstand som i punkt 2, og det er ikke lov til å gå diagonalt. Vi får bruk for
$\sqrt{17} \approx 4.12,\ \sqrt{10} \approx 3.16,\ \sqrt{5} \approx 2.24.$

						XBFG(6.16)
			XBA(6.12)	XBC(6.12)	XBFE(6.16)	XBAE(6.16)
		XBFK(5.00)	XBC(6.12)	XBFE(6.16)	XBFG(6.16)	XBCG(6.16)
		XBA(6.12)	XBFE(6.16)	XBFG(6.16)	XBAE(6.16)	XBFKJ(6.24)
	XBF(5.00)	XBC(6.12)	XBFG(6.16)	XBAE(6.16)	XBCG(6.16)	XBFKL(6.24)
	XBA(6.12)	XBFE(6.16)	XBFKJ(6.24)	XBFKJ(6.24)	XBFKJ(6.24)	XBFEJ(6.24)
XB(5.00)	XBC(6.12)	XBFG(6.16)	XBFKL(6.24)	XBFKL(6.24)	XBFKL(6.24)	XBFED(7.16)

Når dette er situasjonen, går XBFG ut av prioritetskøa. XBFGL(6.24) og XBFGH($4 + \sqrt{13} \approx 7.61$) settes inn i kø. Den neste som går ut XBAE vil på samme måte erstattes av XBAED(7.61) og XBAEJ(6.24). Tilsvarende nye veier får vi når XBCG går ut av køa. XBFKJ(6.24) erstattes av XBFKJI(7.83). XBFKL(6.24) erstattes av XBFKLO(6.41) og XBFKLM(7.82).

Øving

Tegn opp prioritetskøa videre og konkluder med at XBFKLOTS(7) er korteste vei fra X til S.

Listing 9.5: Pseudokode for Dijkstra's algoritme

```
struct Sti{
    std::vector<Kant> kanter;
    double totalkostnad;
    // overlaste < eller >
};

void Graf::Dijkstra(Node* start, Node* end)
{
    std::priority_queue<Kant> pq; // default: max-heap
    // ascending priority queue
    //std::priority_queue<Kant, std::vector<Kant>, std::greater<Kant> > apq;
    // trenger en prioritetsko, hvor elementene er
    // FLERE kanter + total lengde
    // kan lage en hjelpestruktur Sti
    std::priority_queue<Sti, std::vector<Sti>, std::greater<Sti> > apq;

    // Initierer
    Sti startsti;
    Kant startkant{0.0, start};
    startsti.kanter.push_back(startkant);
    startsti.kanter.push_back(startkant);
    startsti.totalkostnad = 0.0;
    apq.push(startsti);

     while (!apq.empty() && !end->m_besokt)
     {
         // pop en sti fra prioritetsko
         auto sti = apq.top();
         apq.pop();
         // {{AB, BE, EF}, 8} {{AB, BE, EF}, 9} {{AB, BE, EC}, 9}, {{AC}, 17}
         // sjekk om noden som stien ender pa er besokt
         if (!end->m_besokt)
         {
             for (;;)/* end sine kanter */
             {
                // Lag ny sti av sti + kant og vekt
                // push denne pa apq
                // kode som ligner pa initieringen av prioritetskoa
             }
         }
     }
}
```

9.4 Minimum spenntre

Gitt en graf $G = \{V, E\}$ hvor V={noder} og E={kanter}. Hvordan kan vi koble sammen alle nodene ved å bruke færrest mulig kanter? Vi skal altså kunne nå enhver node fra en annen - det skal være en veg/sti/path mellom to vilkårlige noder. 9.6 viser et eksempel på en graf hvor dette problemet faktisk kan løses (øverst til venstre), og en mulig løsning (øverst til høyre). Mengden av kantene { AB, BD, BD, DE, CE, CG, CF} utgjør et **spenntre** for grafen. Det er lett å tenke seg flere mulige løsninger på dette ektuelle problemet (prøv!) Når vi setter vekt/kostnad på kantene, er vi interessert i et spenntre med minste totalkostnad. Dette kalles et minimum spenntre. Vi skal studere to klassiske algoritmer som løser dette problemet. Vi skal altså løse følgende problem:

Minumum spenntre problem

Gitt en urettet graf med vekter/kostnader. Hvordan lage et tre som knytter sammen alle nodene med minimum kostnad? La hver kant e_i ha kostnad c_i. Vi har da et minimeringsproblem: min $\Sigma_i c_i$ for alle c_i i spenntreet.

9.4.1 Prim

På figur 9.6 har vi satt på noen vekter på kantene (nederst til venstre). Prim's algoritme starter med en node, velger kanten med minst vekt, og legger til denne i spenntreet. Så gjentas dette for endenoden til denne kanten. Eksempel: La oss starte med node C på **??**. Kanter fra C er CB(17), CE(12), CG(2) og CF(4). Vi velger CG(2) og prøver å sette opp begynnelsen på algoritmen:

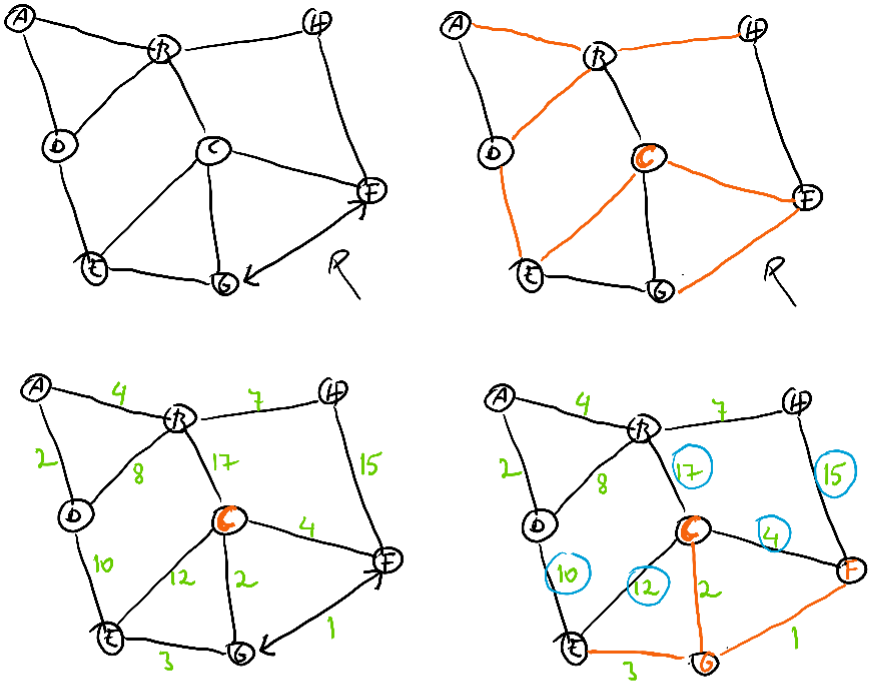
```
// start: C
   for (/*alle kanter fra C*/)
      // velg kant med minst kostnad (CG)
      // koble til tilnoden (G)
      // gjenta
```

Kanter fra G er GE(3), GC(2), GF(1), så det er opplagt at vi skal velge GF(1) og koble til F. Men her støter vi på en utfordring:

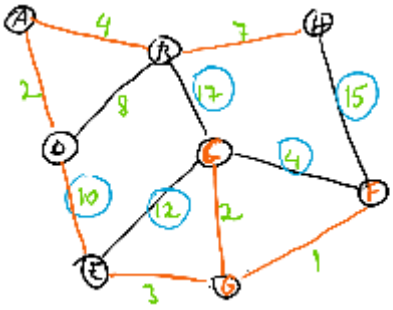
1. Kanter fra F er FC(4) og FH(15). FC(4) fører til en node som allerede er besøkt, og som kan nås fra F med en kostnad 3.
2. Kant GE(3) er nå den kanten som med minst kostnad kobler til en ny node i spenntreet.

Vi bør altså velge GE(3), og ender da opp med et foreløpig spenntre som på figur 9.6 nederst til høyre: De røde nodene er besøkt, de røde kantene er valgt ut, men node E har vi ikke gjort noe med ennå. Kantene som har blå ring rundt vekt er tidligere vurdert (men har hatt for stor kostnad). De ligger kanskje på en prioritetskø?

Prim's algoritme er også en grådig algoritme, og da bruker vi gjerne en prioritetskø til implementeringen. Hvis vi fortsetter å sette inn endenoden til minste kostnads kant og velger minste kostnads kant til ikke besøkte noder, ender vi opp med et minimum spenntre som på figur 9.7. Vi har her brukt samme ide som Prim's algoritme.



Figur 9.6: Prim's minimum spanning tree algoritme.



Figur 9.7: Minimum spenntre (kostnad=29).

Algoritme

Følgende algoritme beregner kostnad til et minimum spenntre ved Prim's algoritme. Algoritmen returnerer ikke hele spenntreet med noder og kanter, men skriver ut kantene og returnerer total kostnad. I implementeringen nedenfor er det benyttet en egen Node og Kant klasse, og en stigende prioritetskø. Nodeklassen har en boolsk objektvariabel mBesokt som initielt er false.

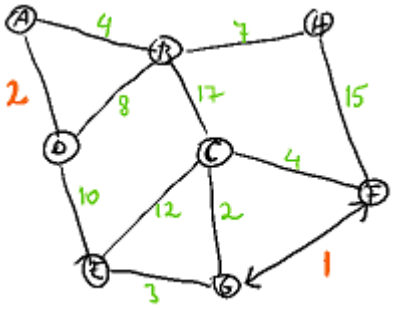
Listing 9.6: Prim's minimum spenntre algoritme

```
float Graf::mst() // Eksamen H2019 - oppgave 4c
{
    float sum = 0.0;
    std::priority_queue<Kant, std::vector<Kant>, std::greater<Kant> > apq;
    // Initialiserer
    Kant kant(0.0, noder.front());
    apq.push(kant);
    do {
        kant = apq.top();
        apq.pop();
        Node* p = kant.m_tilnode;
        if (!p->m_besokt) {
            p->m_besokt = true;
            cout << kant.m_tilnode->m_navn;
            sum += kant.m_vekt;
            for (auto it=p->m_kanter.begin(); it!= p->m_kanter.end(); it++)
            apq.push(*it);
        }
    } while (!apq.empty());
    return sum;
}
```

Legg merke til at med Prim's algoritme har vi i hvert steg et tre, og dette utvides med en kant og en node om gangen.

9.4.2 Kruskal

Kruskal's minimum spanning tree algoritme fokuserer i utgangspunktet på kantene. Vi velger kante med lavest mulig vekt og setter sammen. På figur 9.8 er de to kantene med minste kostnad markert (AD er ikke entydig). I dette steget av algoritmen har vi en skog: Nodene A, D og kant AD, og nodene G, F og kant GF. Se også figur 9.9 øverst.



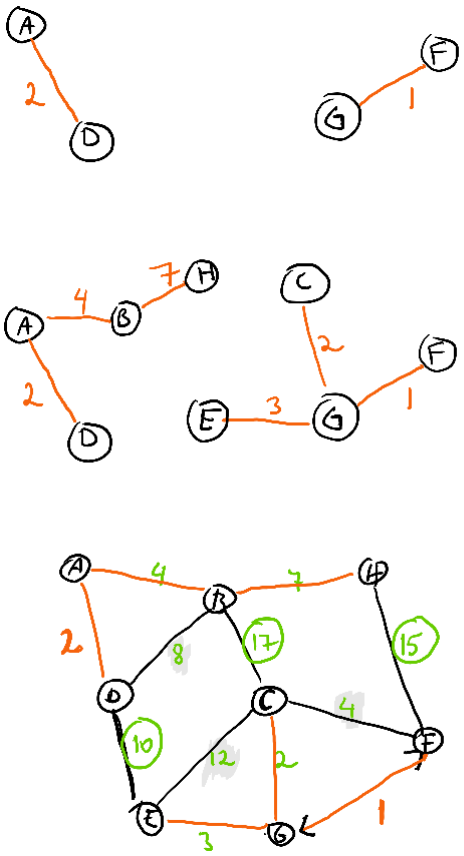
.

Figur 9.8: Kruskal's minimum spanning tree algoritme, start.

De neste stegene i algoritmen fører til innsetting av

1. CG(2) og C,
2. EG(3) og G,
3. AB(7) og B, og
4. BH(7) og H.

Vi har da kommet til situasjonen midt på figur 9.9. Vi har to trær og tre mulige kanter DE(10), BC(17) og FH(15). De andre kantene er ikke aktuelle, siden de kun fører til sykler i eksisterende trær - f.eks har BD(8) høyere kostnad enn BAD(6). Algoritmen velger da DE(10), og vi har et minimum spenntre. Kostnad totalt er 29, selvsagt det samme som for Prim's algoritme.



Figur 9.9: Kruskal's minimum spanning tree algoritme.

9.4.3 Oppgaver

(utgår, brukt i eksempler tidligere i kapitlet).

Graf representasjon

Tegn den interne representasjonen til grafen på figur 9.1 når du bruker implementeringen i avsnitt 9.1.3.

Dynamisk graf implementering

Implementer dynamisk representasjon av en rettet graf (dvs kanter både AB og BA etc). Lag Node og Kant klasser med konstruktører. Node-objektene skal ha et navn og for eksempel en std::vector<Kant*> m_kanter. Kant objektene skal representere franode og tilnode og ha en vekt (kostnad).

Lag et lite testprogram hvor du bygger en graf som i PowerPoint eksemplet og skriver ut grafen (nodene og kantene, kantene som par). Bruk parametrisk konstruktør til å bygge grafen.

Dybde-først traversering

Implementer dybde-først traversering i nodeklassen. Test dybde-først med å starte på A og B nodene.

9.4.4 Fasit

9.4.5 Dynamisk graf implementering

Listing 9.7: kant.h

```
#ifndef KANT_H
#define KANT_H

class Node;
class Kant {
public:
   Kant(Node* fra, Node* til, float vekt);
   float hentVekt() const;
   Node* til() const;
private:
   Node* m_fra;
   Node* m_til;
   float m_vekt;
};
#endif // KANT_H
```

Listing 9.8: kant.cpp

```
#include "kant.h"
#include "node.h"
Kant::Kant(Node *fra, Node *til, float vekt)
   : m_fra{fra}, m_til{til}, m_vekt{vekt} {
    //
}
float Kant::hentVekt() const{
    return m_vekt;
```

```
}
Node* Kant::til() const {
    return m_til;
}
```

Listing 9.9: node.h

```
#ifndef NODE_H
#define NODE_H
#include <vector>
class Kant;
class Node {
public:
    explicit Node(char navn);
    void push(Kant* kant);
    char hentNavn() const;
    void dybdeForst1() const;
    void dybdeForst2();
private:
    char m_navn;
    bool m_besokt;
    std::vector<Kant*> m_kanter;
};
#endif // NODE_H
```

Listing 9.10: node.cpp

```
#include "node.h"
#include "Kant.h"
#include <iostream>
void Node::push(Kant *kant) {
    m_kanter.push_back(kant);
}
char Node::hentNavn() const {
     return m_navn;
}
Node::Node(char navn): m_navn{navn} {
    //
}
void Node::dybdeForst1() const
{
    for (auto it=m_kanter.begin(); it!=m_kanter.end(); it ++)
    {
        Node* n=(*it)->til();
        std::cout << m_navn;
        std::cout << n->hentNavn() << ":";
        std::cout << (*it)->hentVekt() << ",␣";
        n->dybdeForst1();
    }
}
// dybdeforst traversering med besokt-flagg takler sykluser i grafen
void Node::dybdeForst2()
{
    if (!m_besokt)
    {
        m_besokt = true;
        //std::cout << m_navn << ", ";
        for (auto it=m_kanter.begin(); it!=m_kanter.end(); it ++)
        {
            Node* n=(*it)->til();
            std::cout << m_navn;
            std::cout << n->hentNavn() << ":";
            std::cout << (*it)->hentVekt() << ",␣";
```

```
            n->dybdeForst2();
        }
    }
    m_besokt = false;
}
```

Listing 9.11: graf_dybdeforst.cpp

```
#include <iostream>
#include <vector>
#include <kant.h>
#include <node.h>

using namespace std;

int main(int argc, char *argv[])
{
    std::vector<Node*> noder;
    for (char ch='A'; ch<'F'; ch++)
    noder.push_back(new Node(ch));

    Node* n=noder.at(0); //A
    cout << n->hentNavn() << endl;
    n->push(new Kant(n, noder.at(2), 3.14));//AC
    n->push(new Kant(n, noder.at(3), 2.71));//AD
    n=noder.at(1); //B
    cout << n->hentNavn() << endl;
    n->push(new Kant(n, noder.at(2), 9.81));//BC
    n=noder.at(2); //C
    cout << n->hentNavn() << endl;
    n->push(new Kant(n, noder.at(3), 2.71));//CD
    n->push(new Kant(n, noder.at(4), 2.71));//CE
    n=noder.at(4); //E
    cout << n->hentNavn() << endl;
    n->push(new Kant(n, noder.at(3), 2.71));//ED

    cout << "dybdeForst1" << endl;
    n = noder.at(0); //A
    n->dybdeForst1();
    cout << endl;

    // Lager en syklus og dybdeforst virker ikke mere
    // (ta bort kommentartegnene og test!)
    n = noder.at(2); //C
    n->push(new Kant(n, noder.at(0), 1.73));//CA
    //n->dybdeForst1();

    // Men med et ekstra besokt-flagg er det ok
    cout << "dybdeForst2" << endl;
    n = noder.at(0); //A
    n->dybdeForst2();
    cout << endl;

    return 0;
}
```

Kapittel 10

Huffman

10.1 Huffman Introduksjon

Huffman algoritmen er en klassisk algoritme som kan brukes til datakompresjon. Konkret brukes denne algoritmen til å generere et Huffman tre. Et Huffman tre er et binært tre og et slags skjema for kompresjon og dekompresjon.

Vi skal studere selve Huffman-algoritmen, datastrukturer som benyttes, og anvendelser av Huffman algoritmen til datakompresjon. Kapitlet om Huffman algoritmen i [2] (10.1.2, side 471-477), eller i [1] (5.3 side 283-292), er ikke stort. I [2] presenteres ikke implementering. I dette kapitlet vil jeg til en viss grad bruke dynamiske strukturer i stedet for arrayer.

10.1.1 Ide

Vi tenker oss at vi har en fil som består av en million tegn som hver tar en byte. På denne fila er det til sammen 184 ulike tegn. Hvordan kan vi lagre denne fila på mindre plass?

Vi gjør en forenkling som i [1], side 283 og tenker oss en fil som består av kun 7 tegn, BADACCA (eller ABACCDA gjør samme nytten). Her er det 4 ulike tegn på fila – alfabetet består her av ABCD. Siden hvert tegn er en char, trenger vi 7 byte for å lagre denne fila. Hvordan kan vi lagre denne fila på mindre plass?

Hvis vi sammenlikner med fysikk, kan vi si at en char eller byte (som er nesten det samme) tilsvarer et atom. Etter hvert har man funnet ut at et atom består av mindre deler – protoner, nøytroner, elektroner, kvarker. For å løse lagringsproblemet vårt må vi gå under byte-nivå og tenke bit.

Bokstavene ABCD har følgende tallverdi og bitrepresentasjon:

Tegn	*Tallverdi*	*Bitrepresentasjon*
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100

Ideen bak Huffman algoritmen er

- Å betrakte hele fila som en bitstreng.
- Velge en bit-kode for hvert tegn i alfabetet.
- hyppigst forekommende tegn får kortest bit-kode.

I fila ABACCDA kan vi på denne måten få følgende koder:

Tegn	*Frekvens*	*Kode*
A	3	0
B	1	110
C	2	10
D	1	111

Med denne kodingen trenger vi $3 \cdot 1 + 1 \cdot 3 + 2 \cdot 2 + 1 \cdot 3 = 13$ bit for å lagre fila. Den originale fila er på 7 byte og krever 56 bit. For store filer er det nærliggende å tenke seg at man kan spare en del plass. Man kan komprimere fila.

Fila skal kunne dekomprimeres, slik at koden for ett symbol kan ikke være prefiks for koden til et annet symbol.

10.1.2 Huffman algoritmen

Før vi går inn på hvordan vi faktisk kan lage en slik kodet/komprimert fil, tenker vi oss at vi allerede har en komprimert fil som vi skal dekode/dekomprimere:

0110010101110

Vi leser bitstrengen fra venstre. Første bit er 0, og vi vet at dette er koden for tegnet A. Neste bit er 1, og vi vet at dette er koden for enten B, C eller D. Vi leser neste bit som er 1 og vet at tegnet da er enten B eller D. Når vi har lest neste bit som er 0, vet vi at tegnet er B.

Anta nå at vi har en fil hvor hele alfabetet (alle ulike tegn) og frekvensen for hvert tegn er kjent. Et eksempel på dette er kolonne 1 og 2 i tabellen ovenfor. I Huffman algoritmen gjøres følgende:

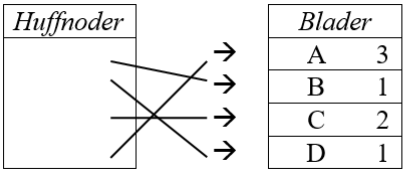
- Generer er Huffman tre. Dette er et strengt binært tre som lages nedenfra og opp, dvs. at vi starter med bladene.
- For hvert blad i Huffman treet benyttes treet til å kode en bitstreng for bladets tegn/symbol. Dette gjøres ved traversering oppover i treet.

Huffman tre generering

Huffman algoritmen genererer et Huffman tre på følgende måte:

1. Velg de to symbolene med lavest frekvens (her: B og D)
2. Kombiner disse til et nytt symbol hvor frekvensen er summert (her: BD med frekvens 2)
3. Vi har nå symbolene og frekvensene A(3), C(2) og BD(2). Vi gjentar prosessen ovenfor og får et nytt symbol CBD med frekvens 4.

Huffman treet blir i dette tilfellet som i [1], figur 5.3.1a, side 286.



Figur 10.1: Huffman noder

Dekoding

Ved dekoding går vi inn på roten i treet. Dersom vi leser et 0 bit, går vi ned til venstre. Dersom vi leser et 1 bit, går vi ned til høyre. Vi fortsetter slik helt til vi når et blad, hvor vi kan lese symbolet.

Vi velger for enkelhets skyld et dynamisk Huffman tre. En datastruktur for nodene i Huffman treet:

```
class Huffnode {
   int frekvens;
   Huffnode *over, *vsub, *hsub;
// Konstruktør etc.

// Følgende trengs kun i bladene:
   char symbol;
   int bits[MAXBITS];
   int startpos;
};
```

Med denne strukturen er det lett å se at 1. og 2. kolonne i hver linje i tabellen ovenfor er et Huffnode objekt.

Huffman algoritmen forutsetter at vi på forhånd kjenner alle symbolene på fil og deres frekvenser. Vi trenger da forut for denne selve algoritmen å gjøre en preprosessering. Preprossesseringen blir å lage et Huffnode objekt eller node for hvert av symbolene i alfabetet og lagre dem i tabell eller pekerkjede. Vær oppmerksom på at alle disse opprinnelige nodene blir blader i Huffman treet, og at symbolet lagres ikke i andre noder enn i disse bladene!

Preprosesseringen blir altså:

1. Lag en (blad-)node for hvert symbol
2. Lenk sammen disse i en liste eller array (lineær eller binær) av blader, eller la blader være en array av pekere eller referanser.
3. Sett disse pekerne inn i en stigende prioritetskø – ordnet etter frekvens.
4. Prioritetskøa er her kalt huffnoder. Initielt har vi en situasjon som på figur 10.1.

Vi forutsetter at *Huffnoder* har funksjoner som tilsvarer

- apqmindelete() som returnerer en peker til minste node i den stigende prioritetskøa (sortert etter nodenes frekvens).
- apqinsert(Huffnode* hn, int frekvens) som setter inn en node på rett plass med hensyn til sorteringen på frekvens.

Vi forutsetter også en konstruktør som ligner på denne:

```
Huffnode::Huffnode(Huffnode* h1, Huffnode* h2){
   frekvens = h1->frekvens + h2->frekvens;
   vsub = h1;
   hsub = h2;
   h1->over = h2->over = this;
   //
}
```

Med denne konstruktøren, en tabell med nodepekere (til blader med symboler og frekvenser) og en stigende prioritetskø apq (se programeksempel for detaljer), kan vi skissere del 1 av Huffman algoritmen slik:

```
// Preprosessering - konstruer bladene og sett inn i apq huffnoder
// Konstruer treet
while ( /* huffnoder inneholder mer enn en node */ ) {
   p1 = huffnoder.Apqmindelete();
   p2 = huffnoder. Apqmindelete();
   p = new Huffnode(p1, p2)
   huffnoder.Apqinsert(p, p->frekvens);
}
rot = huffnoder.Apqmindelete();
```

Dette utgjør altså del 1 av Huffman-algoritmen. Et navn på funksjon for denne delen kan være *lagtre()*. Vi kan tenke oss funksjoner Funksjonen *lagkoder()* og en medlemsfunksjon *lagkode)* til å implementere del 2 i Huffman algoritmen. *lagkoder()* kaller medlemsfunksjonen *lagkode)* for hvert blad.

```
void Lagkoder(Huffnode* bladtab[], int n) {
   for (int i=0; i<n; i++)
      bladtab[i]->lagkode();
}
```

Medlemsfunksjonen Huffnode::lagkode() kalles av et blad i treet og koder ved å klatre opp til roten mens den tester om den kommer fra venstre eller høyre:

```
void Huffnode::lagkode()
{
   // startpos er initialisert lik MAXBITS i konstruktør
   Huffnode* p=this;

   while (p->over != NULL) // dvs p != roten i treet
   {
      startpos = startpos-1;
      if (p == p->over->vsub) // isleft()
         bits[startpos] = 0;
```

```
      else
         bits[startpos] = 1;
      p = p->over;
}
```

10.1.3 Diskusjon

I [1] er en array struktur brukt i stedet. Siden Huffman treet er strengt binært, blir det enkelt å finne en mor og barn til en node, jfr. heap.

Huffman algoritmen trenger som nevnt et alfabet med frekvenser som input. I den fasen som genererer treet, brukes en stigende prioritetskø til suksessivt å ta ut de to minste nodene, kombinere disse til en ny node og sette denne inn igjen i treet. I fase 2 hvor man genererer koder, traverseres treet oppover fra hvert blad, og bitstrengen bits fylles opp bakfra. I bladene brukes altså hver int i arrayen bits til å representere ett bit.

10.1.4 Anvendelser

Fra en gitt fil kan vi lese byte for byte, avdekke et alfabet og telle frekvenser for hvert symbol. Når vi så har konstruert et Huffman tre med koder, kan dette treet brukes

- Til dekoding av en komprimert fil, ved å lese bit for bit, gå inn i roten og gå ned til riktig blad hvor det originale symbolet kan leses
- Til koding av fil. I denne fasen finner man rett blad ikke ved å starte i roten til Huffman treet, men ved å gå inn i en pekertabell eller pekerliste (ovenfor kalt blader), finne riktig symbol, og skrive koden (bitstrengen) for dette symbolet til fil.

Det er konverteringen fra koden (bitstrengen) bits til en ekte bitstreng og tilbake som er utfordringen her. Dette er overlatt til leseren i [1].

10.1.5 Oppgaver

Velg en testfil på ca 10 tegn og deretter et par-tre passende testfiler på ca 15 tegn. Gå gjennom fasene i Huffman algoritmen:

- Bestem alfabet og frekvens.
- Konstruer og tegn Huffman treet.
- Hvordan er prioritetskøen underveis?
- Bestem koden for hvert symbol.
- Hvor stor blir den kodete (komprimerte) fila i forhold til den originale?
- Hva er den teoretisk beste kompresjonen vi kan oppnå med denne algoritmen?

Bibliografi

[1] Y.Langsam, M.J.Augenstein, A.M.Tenenbaum. *Data Structures Using C and C++*. Prentice-Hall, 1996.

[2] M.A.Weiss. *Data Structures and Algorithm Analysis i C++ Fourth Edition*. Pearson, 2014.

[3] R.Sedgewick. *Algorithms in C++*. Addison Wesley, 1992.