Programmering i C++
EDAF30
Dynamiska datastrukturer

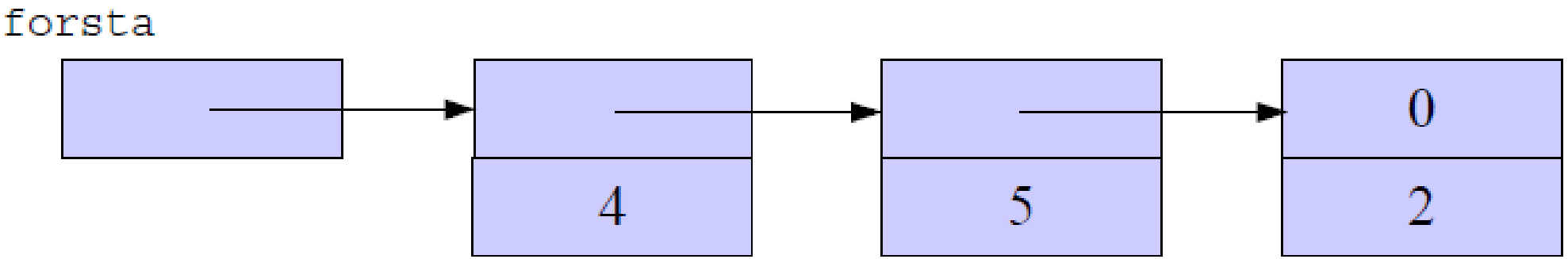
Dynamiska datastrukturer

Innehåll

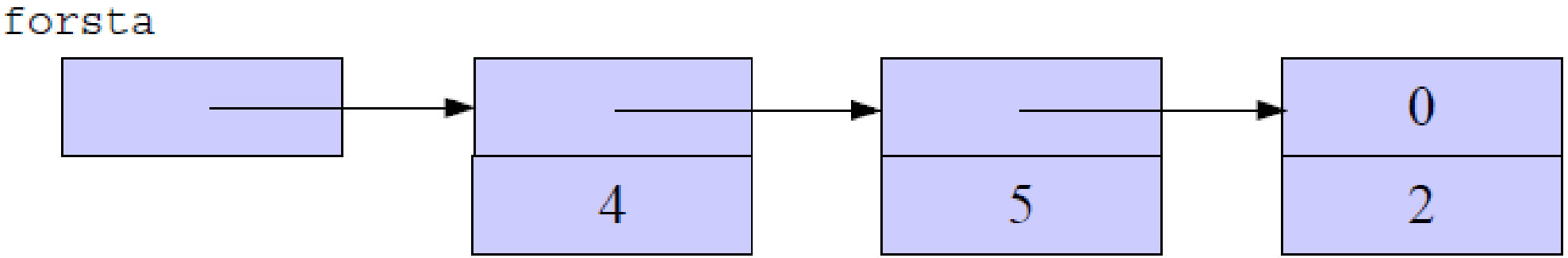
- Länkade listor
- Stackar
- Köer
- Träd
- Säkrare minneshantering (`shared_ptr` och `unique_ptr`)

Enkellänkad lista

```
class Element {
public:
    Element *nasta;
    int data;
    Element(Element *n, int d) : nasta(n), data(d) {}
};
```



Uppbyggande av enkellänkad lista




```
Element *forsta=nullptr; // Börja med tom lista
forsta = new Element(forsta, 2); // 1:a elem
forsta = new Element(forsta, 5); // 2:a elem
forsta = new Element(forsta, 4); // 3:e elem
// forsta pekar på sist inlagda elem.(3:e)
```

Länkade listor

Sökning i enkellänkad lista

```
Element *sok(Element *forsta, int sokt) {
    Element *p;
    for (p = forsta; p && p->data!=sokt; p = p->nasta) {}
    return p;
}
```

Insättning i början på enkellänkad lista

```
void lagg_forst(Element *& forsta, int d) {
    forsta = new Element(forsta, d);
}
```

Insättning i slutet av enkellänkad lista

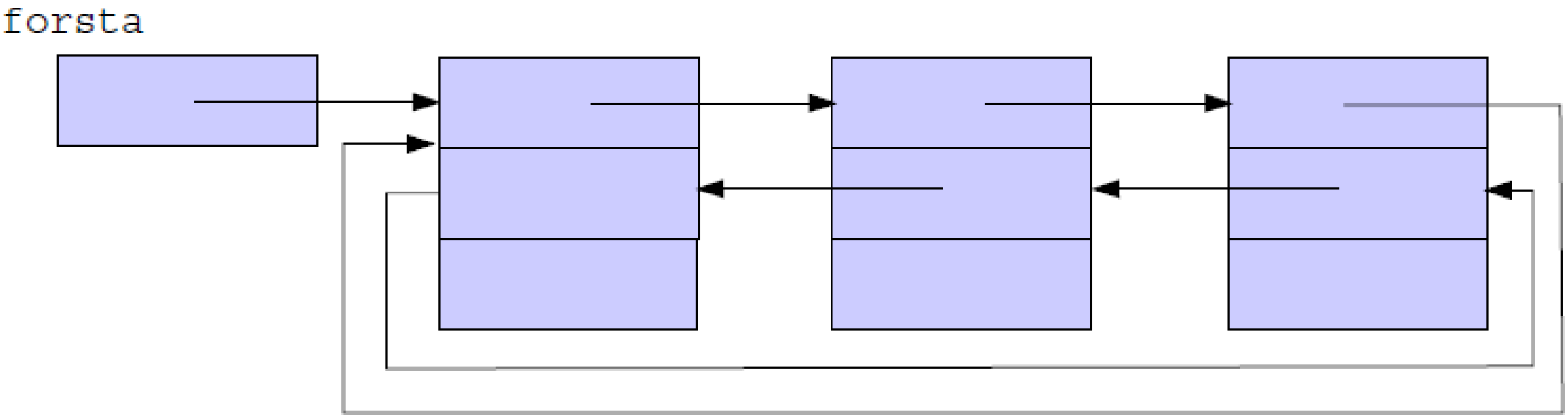
```
void lagg_sist(Element *& forsta, int d) {
    if (!forsta) // tom lista
        forsta = new Element(nullptr, d);
    else {
        Element *p;
        for (p=forsta; p->nasta; p=p->nasta)
            ;
        p->nasta = new Element(nullptr, d);
    }
}
```

Länkade listor och rekursion

```
//Skriva ut lista baklänges mha rekursion
void skriv_baklanges(Element *forsta) {
    if (forsta) {
        skriv_baklanges(forsta->nasta);
        cout << forsta->data << ' ';
    }
}

// Lägg in sist i lista mha rekursion
void lagg_sist_rek(Element *& forsta, int d) {
    if (!forsta)
        forsta = new Element(nullptr, d);
    else
        lagg_sist_rek(forsta->nasta, d);
}
```

Dubbellänkad lista (med huvudnod)

```
class Element {
public:
    Element *fram, *bak;
    int data;
    Element(Element *f=nullptr, Element *b=nullptr, int d=0)
        : fram(f), bak(b), data(d)
        { if (f == nullptr) fram = bak = this; }
};
Element *forsta = new Element;  // skapa tom lista
```



Länkade listor

Ta bort och lägga till i dubbellänkad lista

```
void ta_bort(Element *e) {
    e->bak->fram = e->fram;
    e->fram->bak = e->bak;
    delete e;
}

void lagg_fore(Element *e, int d) {
    Element *pny = new Element(e, e->bak, d);
    e->bak->fram = pny;
    e->bak = pny;
}
```

Lägga in på olika ställen i dubbellänkad lista

```
// Lägga in först i listan
void lagg_forst(Element *forsta, int d) {
    lagg_fore(forsta->fram, d);
}

// Lägga in efter
void lagg_efter(Element *e, int d) {
    lagg_fore(e->fram, d);
}

// Lägg in sist
void lagg_sist(Element *forsta, int d) {
    lagg_efter(forsta->bak, d);
}
```

Utskrift av cirkulär dubbellänkad lista

```
// framlänges
void skriv(Element *forsta) {
    Element *p;
    for (p=forsta->fram; p!=forsta; p=p->fram)
        cout << p->data << ' ';
}

// ... och baklänges
void skriv_baklanges(Element *forsta) {
    Element *p;
    for (p=forsta->bak; p!=forsta; p=p->bak)
        cout << p->data << ' ';
}
```

Stackar

Implementering av en stack mha enkellänkad lista

```
// stack.h
#include <stdexcept>
class Element;
class Stack {
public:
    Stack() : forsta(nullptr) {}
    ~Stack();
    void push(int d);
    void pop(); // throw(length_error)
    int top(); // throw(length_error)
    bool empty();
private:
    Element *forsta;
    Stack(const Stack&) {};
    Stack& operator= (const Stack&) {return *this;}
};
```

Implementering av en stack mha enkellänkad lista

```
// stack.cpp
#include "stack.h"
class Element {
    friend class Stack;
    Element *nasta;
    int data;
    Element(Element *n, int d):nasta(n), data(d) {}
};

void Stack::push(int d) {
    forsta = new Element(forsta, d);
}
```

Stackar

Implementering av en stack mha enkellänkad lista

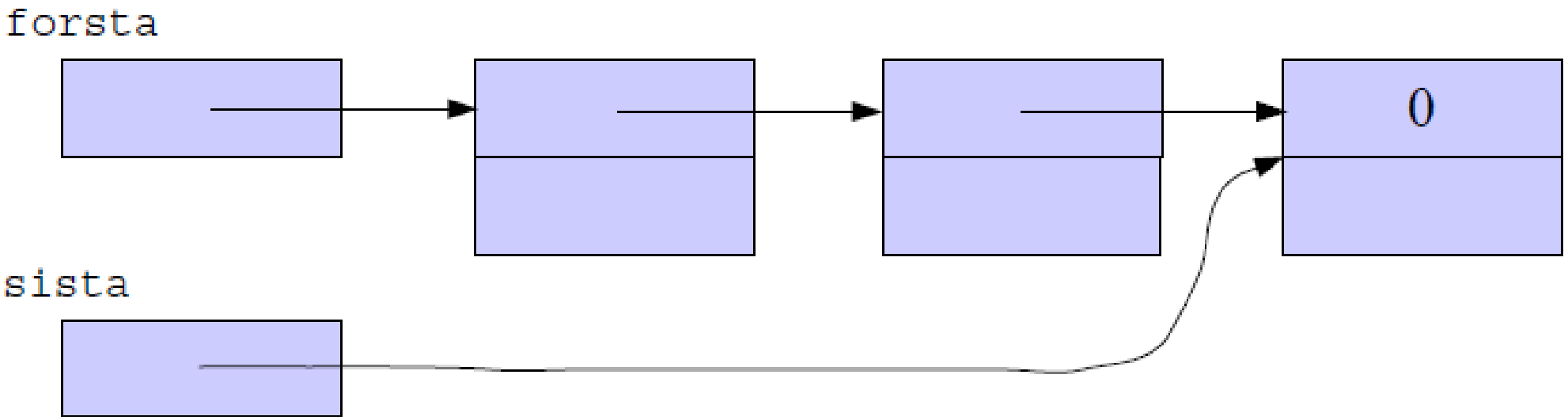
```
// stack.cpp (forts.)
void Stack::pop() /*throw(length_error)*/ {
    if (empty())
        throw length_error("Stack::pop");
    Element *p = forsta;
    int d = p->data;
    forsta = forsta->nasta;
    delete p;
}

int Stack::top() /*throw(length_error)*/ {
    if (empty())
        throw length_error("Stack::top");
    return forsta->data;
}
```

Implementering av en stack mha enkellänkad lista

```
// stack.cpp (forts.)
bool Stack::empty() {
    return !forsta;
}

Stack::~Stack() {
    while (!empty()) {// Gå igenom och ta bort
        Element *p = forsta;
        forsta = forsta->nasta;
        delete p;
    }
}
```

Köer

Implementering av kö mha enkellänkad lista + extra pekare



Köer

Implementering av kö mha enkellänkad lista + extra pekare

```
// queue.h
#include <stdexcept>
class Element;
class Queue {
public:
    Queue() : forsta(nullptr), sista(nullptr) {}
    ~Queue();
    void push(int d);
    void pop(); // throw(std::length_error)
    int front(); // throw(std::length_error)
    int back(); // throw(std::length_error)
    bool empty();
private:
    Element *forsta, *sista;
    Queue(const Queue&) {}
    Queue& operator= (const Queue&) {}
}
```

Köer

Implementering av kö mha enkellänkad lista + extra pekare

```
// queue.cpp
#include "queue.h"
using namespace std;
class Element {
    friend class Queue;
    Element *nasta;
    int data;
    Element(Element *n, int d) : nasta(n), data(d) {}
};
void Queue::push(int d) {
    Element *pny = new Element(0, d);
    if (empty())
        forsta = pny;
    else
        sista->nasta = pny;
    sista = pny;
}
```

Implementering av kö mha enkellänkad lista + extra pekare

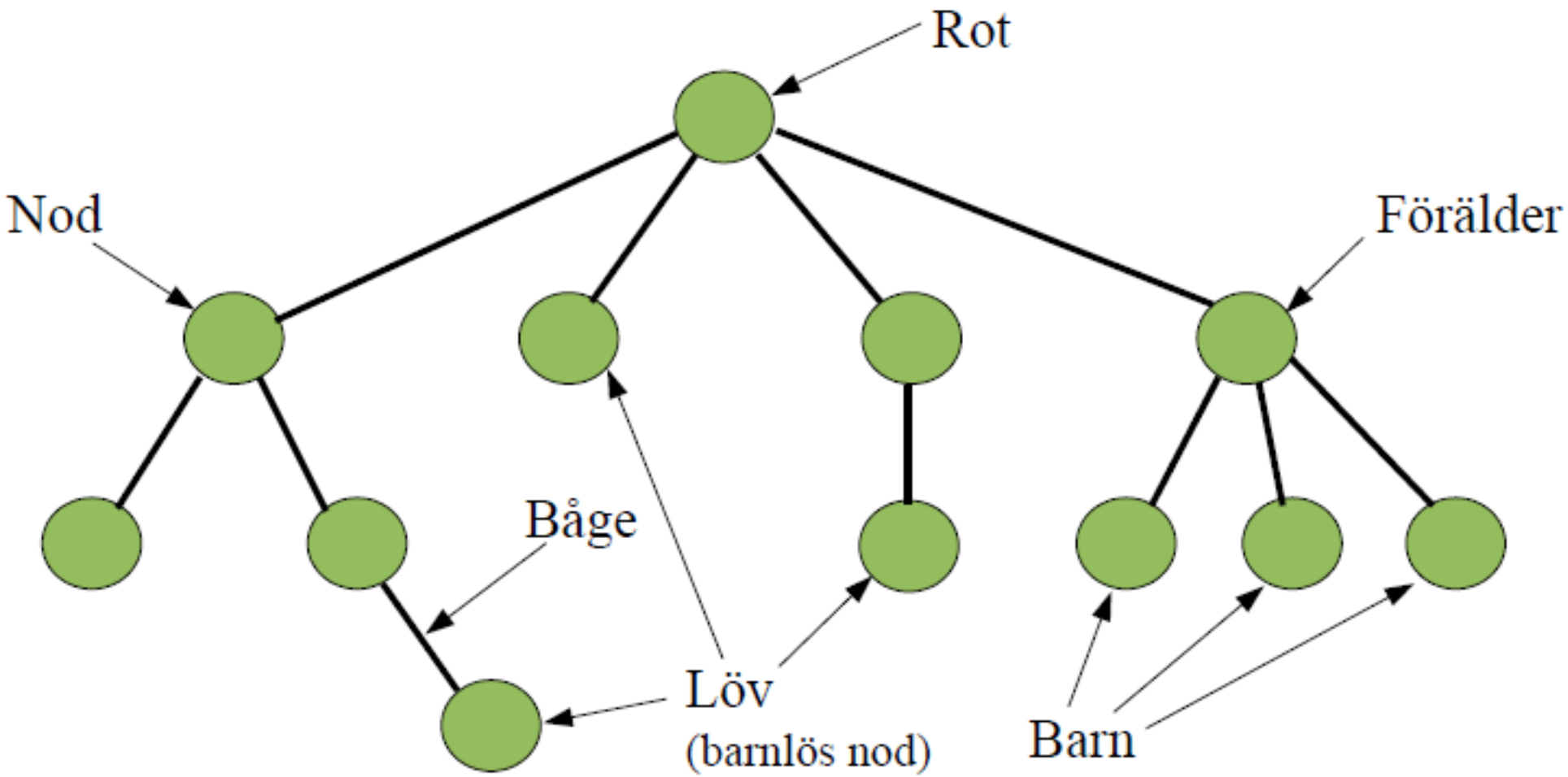
```
// queue.cpp (forts.)
void Queue::pop() /*throw(length_error)*/ {
    if (empty())
        throw length_error("Queue::pop");
    Element *p=forsta;
    forsta = forsta->nasta;
    if (empty())
        sista = 0;
    delete p;
}

int Queue::front() /*throw(length_error)*/ {
    if (empty())
        throw length_error("Queue::front");
    return forsta->data;
}
```

Köer

Implementering av kö mha enkellänkad lista + extra pekare

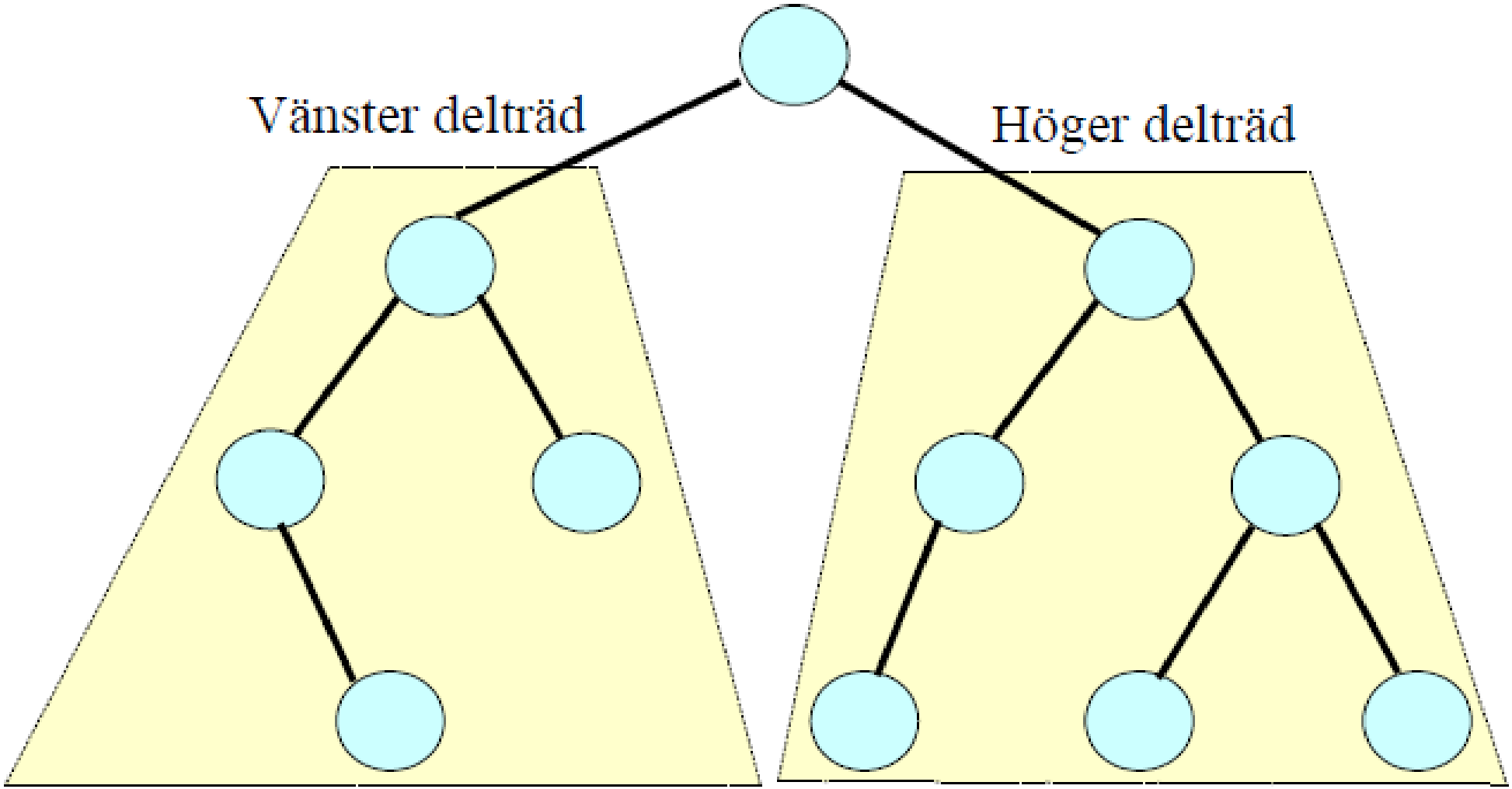
```
// queue.cpp (forts.)
int Queue::back() /*throw(length_error)*/ {
    if (empty())
        throw length_error("Queue::back");
    return sista->data;
}
bool Queue::empty() {
    return !forsta;
}
// Destruktorn
Queue::~Queue() {
    while (!empty()) {
        Element *p = forsta;
        forsta = forsta->nasta;
        delete p;
    }
}
```

Allmän trädstruktur



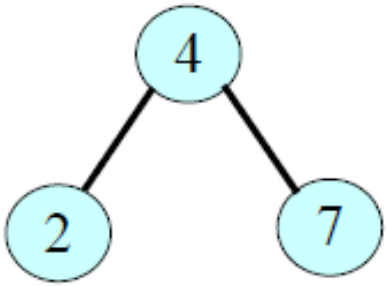
Träd

Binärt träd: Högst två barn per förälder



En klass för noder i ett binärt träd

```
class Nod {
public:
    int data;
    Nod *vanster, *hoger;
    Nod(int d=0, Nod *v=nullptr, Nod *h=nullptr)
        : data(d), vanster(v), hoger(h) {}
};

// ... skapa ett litet binärträd
Nod *rot = new Nod(4);
rot->vanster = new Nod(2);
rot->hoger = new Nod(7);
```



Träd

Gå igenom (traversera) ett träd i *in-order*:
vänster delträd – rot – höger delträd

```
// Utskrift av delträd med rot p i in-order
void inorder(Nod *p) {
    if (p != nullptr) {
        inorder(p->vanster);
        cout << p->data << ' ';
        inorder(p->hoger);
    }
}
```

Träd

Gå igenom (traversera) ett träd i *pre-order*:
rot – vänster delträd – höger delträd

```
// Utskrift av delträd med rot p i pre-order
void preorder(Nod *p) {
    if (p != nullptr) {
        cout << p->data << ' ';
        preorder(p->vanster);
        preorder(p->hoger);
    }
}
```

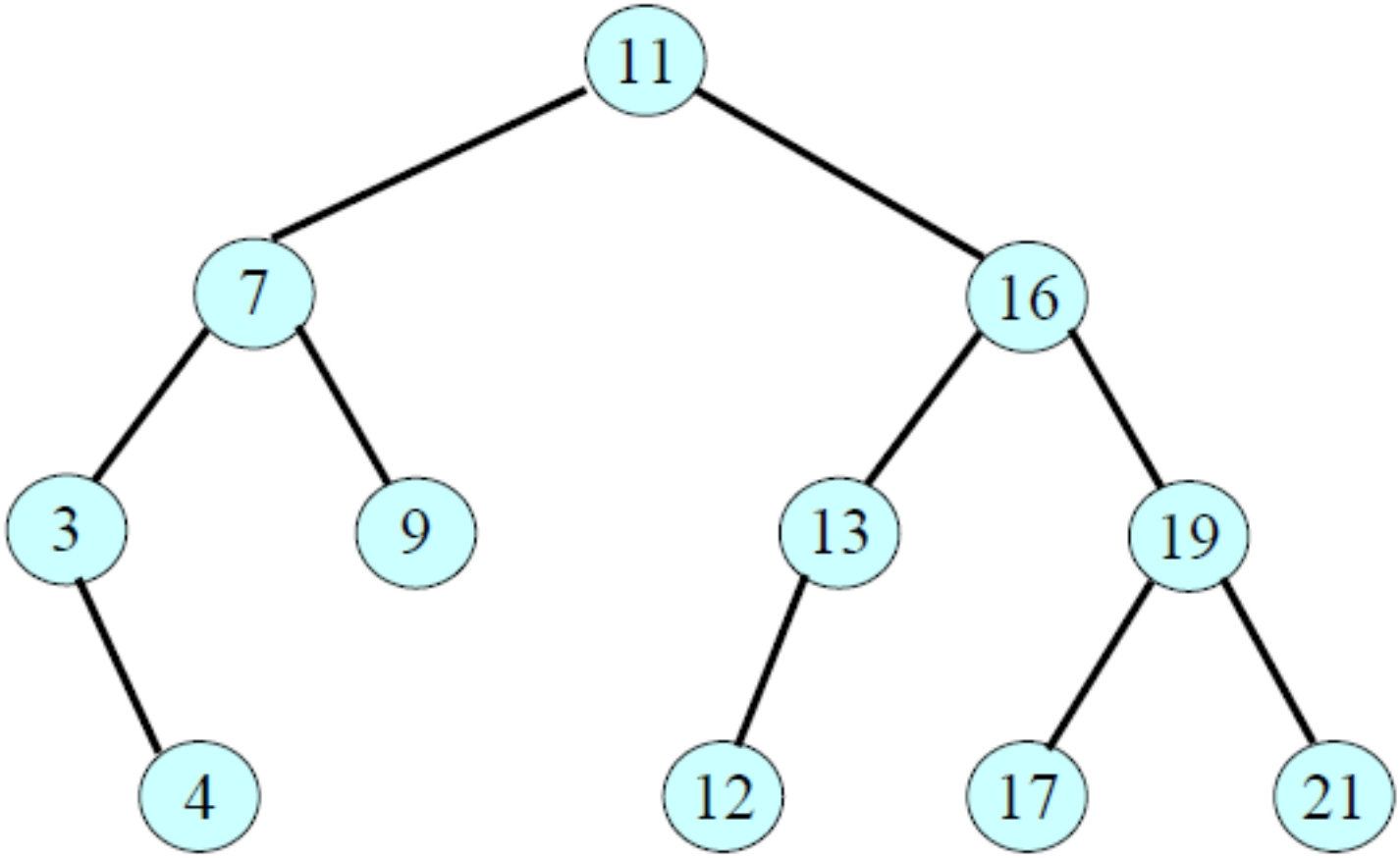
Träd

Gå igenom (traversera) ett träd i *post-order*:
vänster delträd – höger delträd – rot

```
// Utskrift av delträd med rot p i post-order
void postorder(Nod *p) {
    if (p != nullptr) {
        postorder(p->vanster);
        postorder(p->hoger);
        cout << p->data << ' ';
    }
}
```

Träd

Traversering av ett träd på 3 olika sätt



Träd

Traversering av ett träd på 3 olika sätt

- Resultat av inorder-utskrift

  `3 4 7 9 11 12 13 16 17 19 21`

- Resultat av preorder-utskrift

  `11 7 3 4 9 16 13 12 19 17 21`

- Resultat av postorder-utskrift

  `4 3 9 7 12 13 17 21 19 16 11`

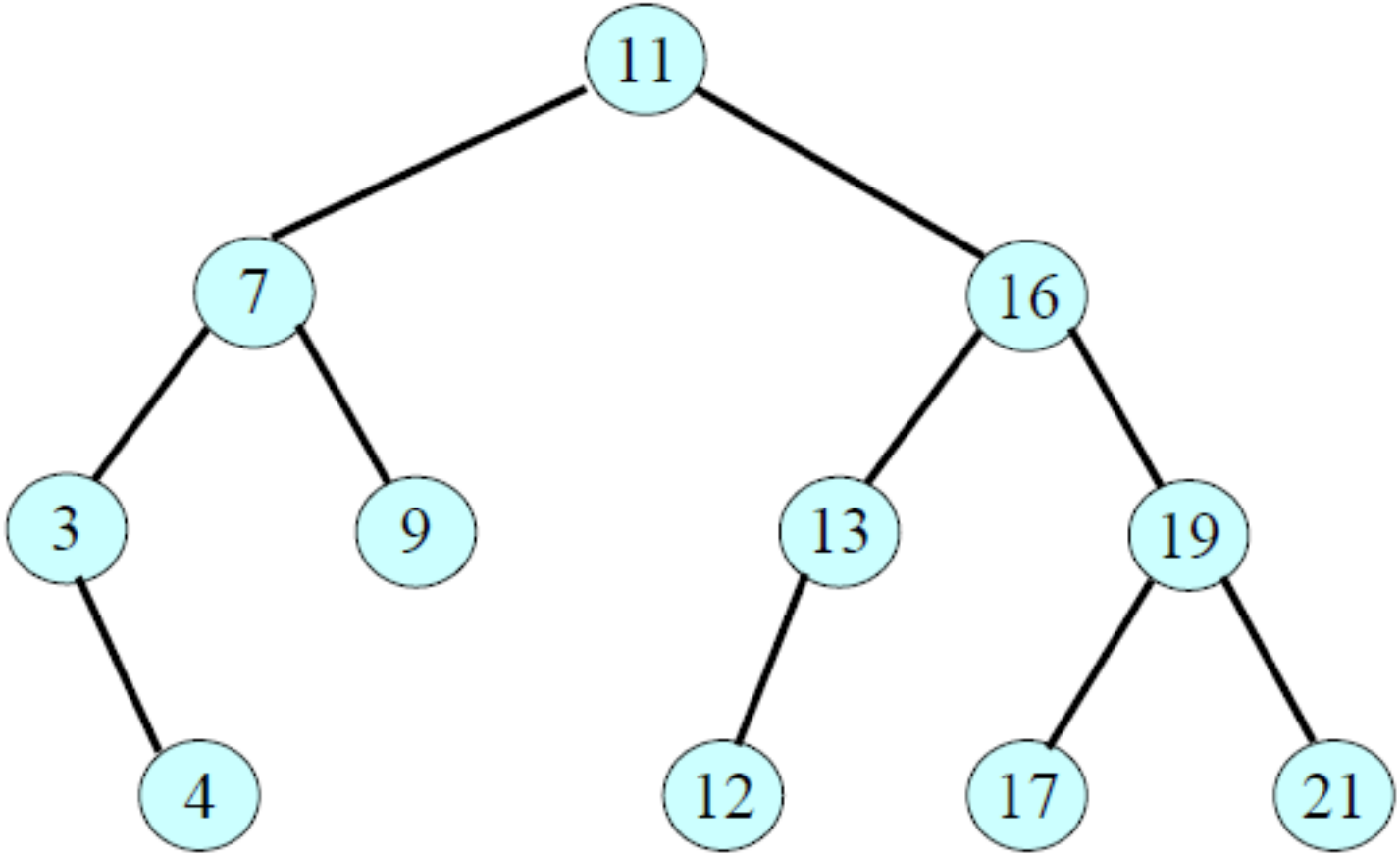
Träd

Djupet för ett träd = antalet noder på den längsta vägen från trädets rot till något av dess löv

```
int djup(Nod *r) {
    if (r == nullptr)
        return 0;
    else {
        int vdjup = djup(r->vanster);
        int hdjup = djup(r->hoger);
        if (vdjup > hdjup)
            return vdjup + 1;
        else
            return hdjup + 1;
    }
}
```

Träd

Binärt sökträd: Alla noder i vänstra delträdet har mindre nyckelvärden än roten och alla noder i högra har större värden



Träd

Egenskap hos binära sökträd: Utskrift i *inorder* ger värdena i *växande ordning* (sorterat)

3 4 7 9 11 12 13 16 17 19 21

Sökning efter visst värde

```
Nod* sok(Nod *r, int sokt) {
    if (r == nullptr)
        return nullptr;
    else if (sokt == r->data)
        return r;
    else if (sokt < r->data)
        return sok(r->vanster, sokt);
    else if (sokt > r->data)
        return sok(r->hoger, sokt);
}
```

Säkrare minneshantering (ej i kursboken – överkurs)

I biblioteket till C++11 finns tillägg för ”säkra pekare”, bl.a.:

`shared_ptr<T>`	en ”säker” pekarvariabel med *delat* ägarskap till ett objekt av typen `T` objektet tas bort när sista ”säkra” pekaren på det försvinner
`make_shared<T>(`*params*`)`	skapar och returnerar en ”säker” pekare använd i st f ’`new T(`*params*`)`’
`unique_ptr<T>`	en ”säker” pekarvariabel med *ensamt* ägarskap till ett objekt av typen `T` objektet tas bort när sista ”säkra” pekaren på det försvinner
`make_unique<T>(`*params*`)`	skapar och returnerar en ”säker” pekare använd i st f ’`new T(`*params*`)`’

Säkrare minneshantering (exempel)

En klass för noder i ett binärt träd med `shared_ptr`

```
#include <memory>

using namespace std;

class Nod {
public:
    int data;
    shared_ptr<Nod> vanster, hoger;
    Nod(int d = 0, shared_ptr<Nod> v = nullptr,
        shared_ptr<Nod> h = nullptr)
        : data(d), vanster(v), hoger(h) {}
    ~Nod() { }  // shared_ptr ser till noderna
                // i delträden tas bort när
                // inga pekare finns kvar på dem
};
```

Säkrare minneshantering (exempel – forts.)

`shared_ptr` används på samma sätt som en ”vanlig” pekare

```
// ... skapa ett litet binärträd
shared_ptr<Nod> rot = make_shared<Nod>(4);
rot->vanster = make_shared<Nod>(2);
rot->hoger = make_shared<Nod>(7);

// ta bort den enda pekaren till rotnoden
// genom att tilldela rot ett nytt värde
rot = nullptr;  // destruktorn till Nod anropas
                // som anropar destruktorerna för
                // vanster och hoger (som anropar...)
```