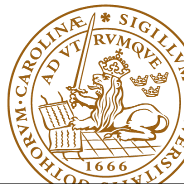
EDAF30 – Programmering i C++

Dynamiska datastrukturer

Sven Gestegård Robertz

Datavetenskap, LTH

2015



Innehåll

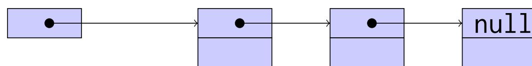
1. Länkade strukturer
   - Länkade listor
     - Stackar
     - Köer
   - Dubbellänkade listor
   - Träd
2. Smarta pekare (shared_ptr och unique_ptr)

Länkade listor

Enkellänkad lista

```
struct Element {
    Element* next;
    int data;
    Element(Element* n, int d) : next(n), data(d) {}
};
```

huvud:



Länkade listor

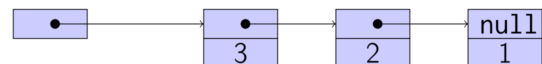
Uppbyggande av enkellänkad lista

```
Element* head=nullptr; // Börja med tom lista

head = new Element(head, 1);
head = new Element(head, 2);
head = new Element(head, 3);

// head pekar på sist inlagda elementet
```

head:



Kom ihåg **delete**

Länkade listor
Insättning först respektive sist

Insättning i början på enkellänkad lista

```
void insert_first(Element*& head, int d) {
    head = new Element(head, d);
}
```

Insättning i slutet av enkellänkad lista

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


Länkade listor

Insättning efter ett element e

```
void insert_after(Element* e, int d)
{
    if(e) {
        e->next = new Element(e->next, d);
    }
}
```

Insättning före ett element e

```
Element* prepend(const Element* const e, int d)
{
    return new Element(e, d);
}
```

Elementet känner inte till föregående element, användaren får göra rätt tilldelning, t ex `head = prepend(head, 10);`

Länkade listor

Sökning i enkellänkad lista

```
Element* find(Element* head, int val) {
    Element* p;

    for (p = head; p && p->data!=val; p = p->next) ;

    return p;
}
```

Tydligare(?) variant med while-sats

```
Element* find(Element* head, int val) {
    Element* p=head;

    while(p && p->data!=val){
        p = p->next;
    }
    return p;
}
```


Länkade listor

Länkade listor och rekursion

Skriva ut lista baklänges mha rekursion:

```
void print_reverse(Element* head) {
    if (head) {
        print_reverse(head->next);
        cout << head->data << ' ';
    }
}
```

Lägg in sist i lista mha rekursion:

```
void insert_last_rec(Element* & head, int d) {
    if (!head)
        head = new Element(nullptr, d);
    else
        insert_last_rec(head->next, d);
}
```


Enkellänkade listor
Kommentarer

- Insättning “mitt i” är dyrt
- Länkade listor med “nakna” element är känsliga
  - Används ofta inuti implementatationen av en annan datastruktur
  - Tänk på minneshantering

Stackar
Implementering av en stack mha enkellänkad lista

stack.h

```
#include <stdexcept>
class Element;  // forward declaration

class Stack{
public:
    Stack() :head{nullptr} {}
    ~Stack();
    void push(int d);
    void pop();
    int top();
    bool empty() {return head==nullptr;}
private:
    Element* head;
    Stack(const Stack&) =delete; // eller  {} om inte C++11
    Stack& operator=(const Stack&) =delete; //{return *this;}
};
```


Stackar
Implementering av en stack mha enkellänkad lista

stack.cpp

```
#include "stack.h"

class Element {
    friend class Stack;
    Element* next;
    int data;
    Element(Element* n, int d) :next{n},data{d} {}
};

Stack::~Stack()
{
    while(!empty()) {
        pop();
    }
}
```

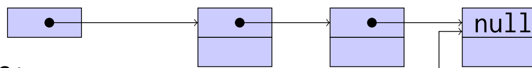

Stackar
Implementering av en stack mha enkellänkad lista

stack.cpp (forts.)

```
void Stack::push(int d)
{
    head = new Element(head, d);
}
void Stack::pop()
{
    if(empty()) {
        throw std::length_error("Stack::pop");
    }
    auto h = head->next;
    delete head;
    head = h;
}
int Stack::top()
{
    if(empty()) {
        throw std::length_error("Stack::top");
    }
    return head->data;
}
```

Köer
Implementering av kö mha enkellänkad lista + extra pekare

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() :head{nullptr},last{nullptr} {}
    ~Queue();
    void push_back(int d);
    void pop_front();
    int front();
    int back();
    bool empty() {return head==nullptr;}
private:
    Element* head;
    Element* last;
    Queue(const Queue&) =delete;
    Queue& operator=(const Queue&) =delete;
};
```


Köer
Implementering av kö mha enkellänkad lista + extra pekare

queue.cpp

```
#include "queue.h"
Queue::~Queue()
{
    while(!empty())
        pop_front();
}

void Queue::push_back(int d)
{
    auto tmp = new Element(nullptr, d);
    if(empty()) {
        head = tmp;
    } else {
        last->next = tmp;
    }
    last = tmp;
}
```


Köer
Implementering av kö mha enkellänkad lista + extra pekare

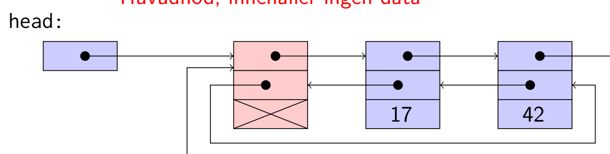
queue.cpp (forts.)

```
void Queue::pop_front()
{
    if(empty()) {
        throw std::length_error("Queue::pop_front");
    }
    auto h = head->next;
    delete head;
    head = h;
    if(head == nullptr)
        last=head;
}
int Queue::front()
{
    if(empty()) {
        throw std::length_error("Queue::front");
    }
    return head->data;
}
int Queue::back() // analogt med front(), fast last->data
```


Länkade listor
Dubbellänkad lista (med huvudnod)

```
struct Element {
  Element* next;
  Element* prev;
  int data;
  Element() :next(this), prev(this), data(0) {}
  Element(Element* n, Element* p, int d=0)
    : next(n), prev(p), data(d)
    { if (p == nullptr) throw std::invalid_argument("prev is null");
      if (n == nullptr) throw std::invalid_argument("next is null");}
};
```

Huvudnod, innehåller ingen data



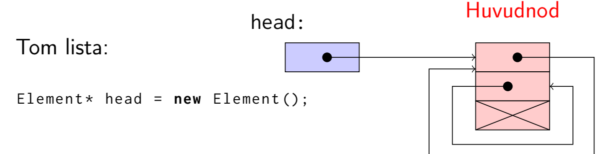
Länkade listor
Dubbellänkad lista (med huvudnod)

- *dubbellänkad*: varje element har pekare till elementen före och efter
- *cirkulär lista*: sista elementet länkar tillbaka till huvudet och vice versa.
- Implementation med *huvudnod*: en nod som representerar början av listan (ett "tomt" element före första elementet). Kostar ett element, gör implementationen enklare (t ex inga specialfall för tomma listan).

Tom lista:

```
Element* head = new Element();
```

Huvudnod



Länkade listor

Ta bort och lägga till i dubbellänkad lista

```
void remove(Element* e) {
    e->prev->next = e->next;
    e->next->prev = e->prev;
    delete e;
}

void insert_before(Element* e, int d) {
    Element* ne = new Element(e, e->prev, d);
    e->prev->next = ne;
    e->prev = ne;
}
```


Länkade listor
Lägga in på olika ställen i dubbellänkad lista

använd `insert_before`

```
// Lägga in efter
void insert_after(Element* e, int d) {
    insert_before(e->next, d);
}

// Lägga in först i listan
void insert_first(Element* head, int d) {
    insert_before(head->next, d);
}

// Lägg in sist
void insert_last(Element* head, int d) {
    insert_after(head->prev, d); // eller insert_before(head, d);
}
```

*Notera att pekaren till huvudnoden inte ändras (*head *är ingen utparameter)*

Länkade listor

Utskrift av cirkulär dubbellänkad lista

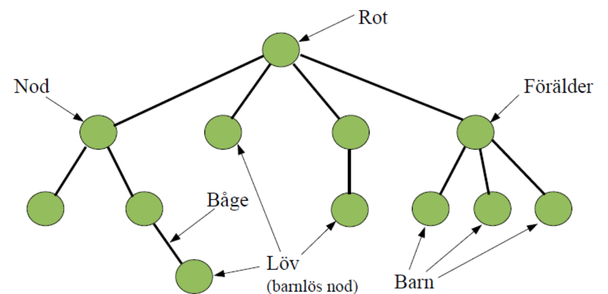
```
// framlänges
void print_list(const Element * const  head) {
    Element* p;
    for (p=head->next; p!=head; p=p->next)
        std::cout << p->data << ' ';
}

// ... och baklänges
void reverse_print_list(const Element* const head) {
    Element* p;
    for (p=head->prev; p!=head; p=p->prev)
        std::cout << p->data << ' ';
}
```

Huvudnoden skrivs inte ut. Notera hur tomma listan inte behöver hanteras speciellt.

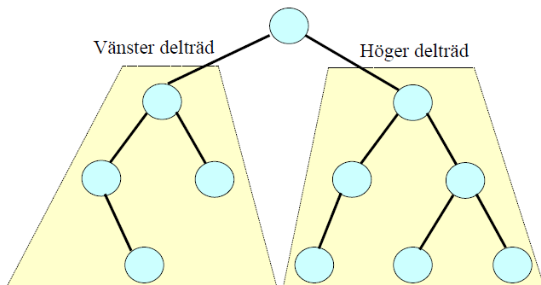
Träd

Allmän trädstruktur



Träd

Binärt träd: Högst två barn per förälder

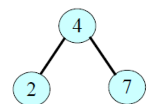


Träd

En klass för noder i ett binärt träd

```
class Nod {
public:
    int data;
    Nod* vanster;
    Nod* 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);
    }
}
```

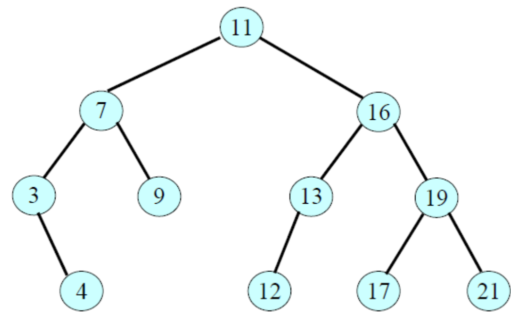

Träd

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 << ' ';
    }
}
```

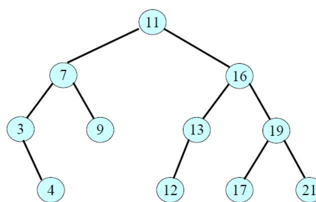

Träd

Traversering av ett träd på 3 olika sätt



Träd

Traversering av ett träd på 3 olika sätt



- inorder-utskrift
  3 4 7 9 11 12 13 16 17 19 21
- preorder-utskrift
  11 7 3 4 9 16 13 12 19 17 21
- postorder-utskrift
  4 3 9 7 12 13 17 21 19 16 11

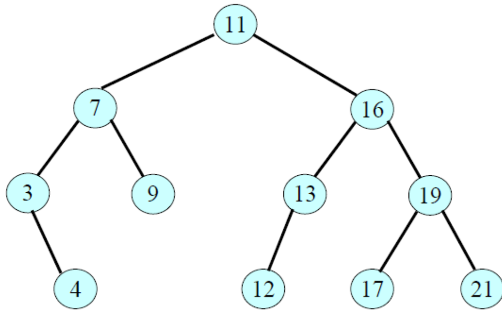
Träd

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);
}
```


Länkade strukturer

- Kan effektivt växa dynamiskt
  - Men `std::vector` är ofta effektivare i praktiken
- Rekursiva algoritmer
- Implementeras med pekare: var noga med ägarskap

Säkrare minneshantering

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)`' (C++14)

Säkrare minneshantering (exempel)

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

```
#include <memory>

using std::shared_ptr;

class Nod {
public:
    int data;
    shared_ptr<Nod> vanster;
    shared_ptr<Nod> 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...)
```

shared_ptr
Kommentarer

- `shared_ptr` använder *referensräkning*
- *cykler ⇒ minnesläckor*

- `weak_ptr` är en *icke-ägande* referens till ett objekt som ägs av en `shared_ptr`
  - används för att bryta cykler
  - måste omvandlas till en `shared_ptr` innan den kan dereferereras
- två sätt att omvandla en `weak_ptr` (ta ägarskap)
  - `std::shared_ptr<T> std::weak_ptr::lock() const;`
  - konstruktorn `shared_ptr(const std::weak_ptr<T>&);`
  - `lock` ger tom pekare, konstruktorn kastar exception

Vi har talat om

- Länkade strukturer

Nästa föreläsning:

Vi kommer att gå igenom

- Generisk programmering med templates
  - Parametriserade typer
  - Parametriserade funktioner