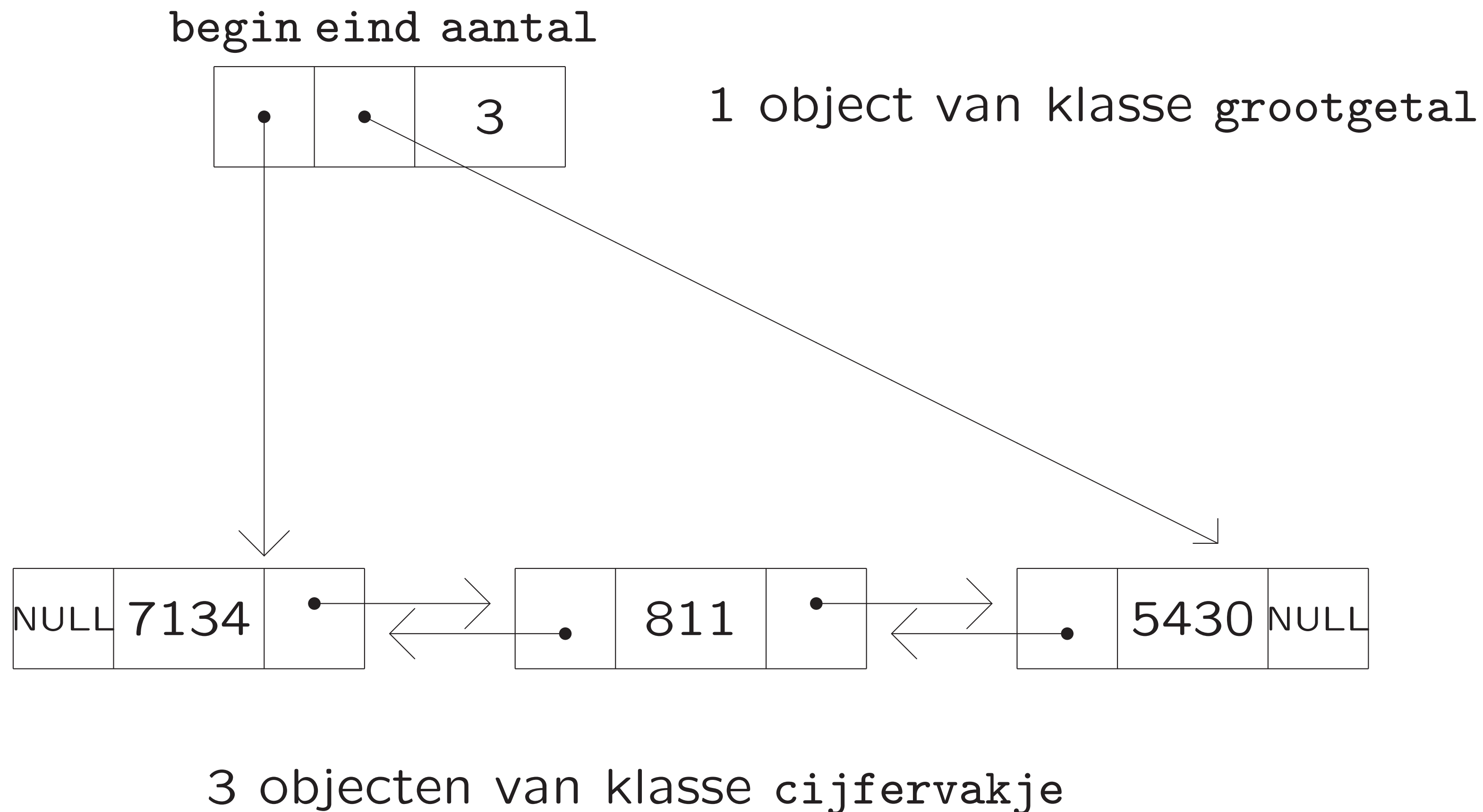
Programmeermethoden

Datastructuren: stapels, rijen en binaire bomen

week 12: 28 november–2 december 2011

http://www.liacs.nl/home/kosters/pm/

Het “grote getal” 713408115430 wordt (met $k = 4$) zo gerepresenteerd:



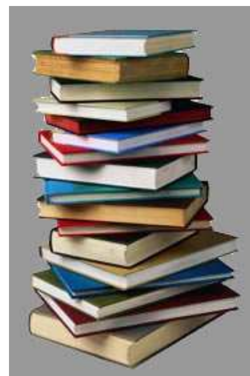
```
class cijfervakje {
  public:
    int getal;
    cijfervakje* vorige;
    cijfervakje* volgende;
};//cijfervakje
class grootgetal {
  private:
    int aantal;
    cijfervakje* begin;
    cijfervakje* einde;
  public:
    ...
};//grootgetal
```

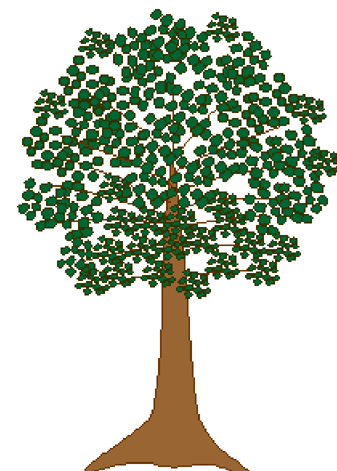
```
class grootgetal {
  ...
  public:
    void telop (grootgetal & gg1, grootgetal & gg2);
    ...
};//grootgetal

void grootgetal::telop (grootgetal & gg1, grootgetal & gg2){
  ...
  gebruik gg1.einde, gg2.einde
  tel getallen op modulo k-de macht van 10
  doe dat in "ik" (*this)
  denk aan de overdracht = carry
  ...
}//grootgetal::telop
```

In de informatica worden Abstracte Data Typen (ADT's) zoals stapels, rijen en bomen veelvuldig gebruikt.

Bij de afwikkeling van recursie komen bijvoorbeeld stapels goed van pas.

De OOP-filosofie sluit hier mooi op aan.





Een Data Type bestaat uit een domein (een collectie "waarden", al dan niet met structuur), in combinatie met een aantal basisoperaties die op dit domein gedefinieerd zijn.

We spreken van een **Abstract** Data Type als de implementatie van de operaties is afgeschermd van de gebruiker.

Voorbeeld 1: De gehele getallen (`int` in C++), met basisoperaties zoals $+$, $-$ en $*$.
De gebruiker kan deze operaties wel gebruiken, maar weet niet (en hoeft ook niet te weten) hoe deze precies in C++ zijn geïmplementeerd.

Voorbeeld 2: Verzamelingen, zoals verzamelingen gehele getallen tussen 0 en n: $\{3, 6, 10\}$.

Bekijk het data type Verzameling (Set) waarvan het domein bestaat uit verzamelingen gehele getallen tussen 0 en n. Een verzameling is ongeordend en bevat allemaal verschillende elementen. Als basisoperaties op deze verzamelingen definiëren we:

- een lege verzameling aanmaken (of een bestaande leeg maken),
- kijken of de verzameling leeg is,
- testen of een gegeven getal erin zit,
- een getal eraan toevoegen,
- een getal eruit halen.

```
class verzameling {
  public:
    verzameling ( );            // constructor
    bool isleeg ( );            // is V leeg?
    bool ziterin (int i);       // zit i in V ?
    void erbij (int i);         // stop i in V
    void eruit (int i);         // haal i uit V
    ...
  private:        // de implementatie wordt afgeschermd
    bool inhoud[n];             // n een constante
};//verzameling
```

We implementeren een verzameling V dus met behulp van een array `inhoud`, waarbij `inhoud[i] == true` $\Longleftrightarrow i \in V$.

Met behulp van de basisoperaties kunnen we andere functies schrijven, zoals `void verzameling::doorsnede (A,B)`.

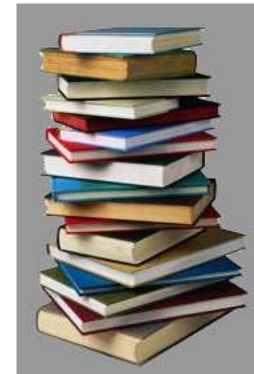
```
verzameling::verzameling ( ) {
  for ( int i = 0; i < n; i++ ) inhoud[i] = false;
}//verzameling::verzameling
bool verzameling::isleeg ( ) {
  for ( int i = 0; i < n; i++ )
    if ( inhoud[i] ) return false;
  return true;
}//verzameling::isleeg
bool verzameling::ziterin (int i) {
  return inhoud[i];
}//verzameling::ziterin
void verzameling::erbij (int i) {
  inhoud[i] = true;
}//verzameling::erbij
void verzameling::eruit (int i) {
  inhoud[i] = false;
}//verzameling::eruit
```

```
// de verzameling *this wordt gelijkgemaakt aan A door B
void verzameling::doorsnede (verzameling & A,
                             verzameling & B) {
  for ( int i = 0; i < n; i++ ) {
    if ( A.ziterin (i) && B.ziterin (i) )
      erbij (i);  // oftewel this->erbij (i);
  }//for
}//doorsnede
```

In main (voor $\{6,10\} \cap \{3,6\} = \{6\}$):

```
  verzameling een, twee, drie;
  een.erbij (10); een.erbij (6);  // vul een = {6,10}
  twee.erbij (3); twee.erbij (6); // vul twee = {3,6}
  drie.doorsnede (een, twee);     // drie wordt de doorsnede
                                  // van een en twee: {6}
```

Een stapel (stack; denk aan een stapel boeken) is een reeks elementen van hetzelfde type, bijvoorbeeld gehele getallen, met de volgende toegestane operaties:

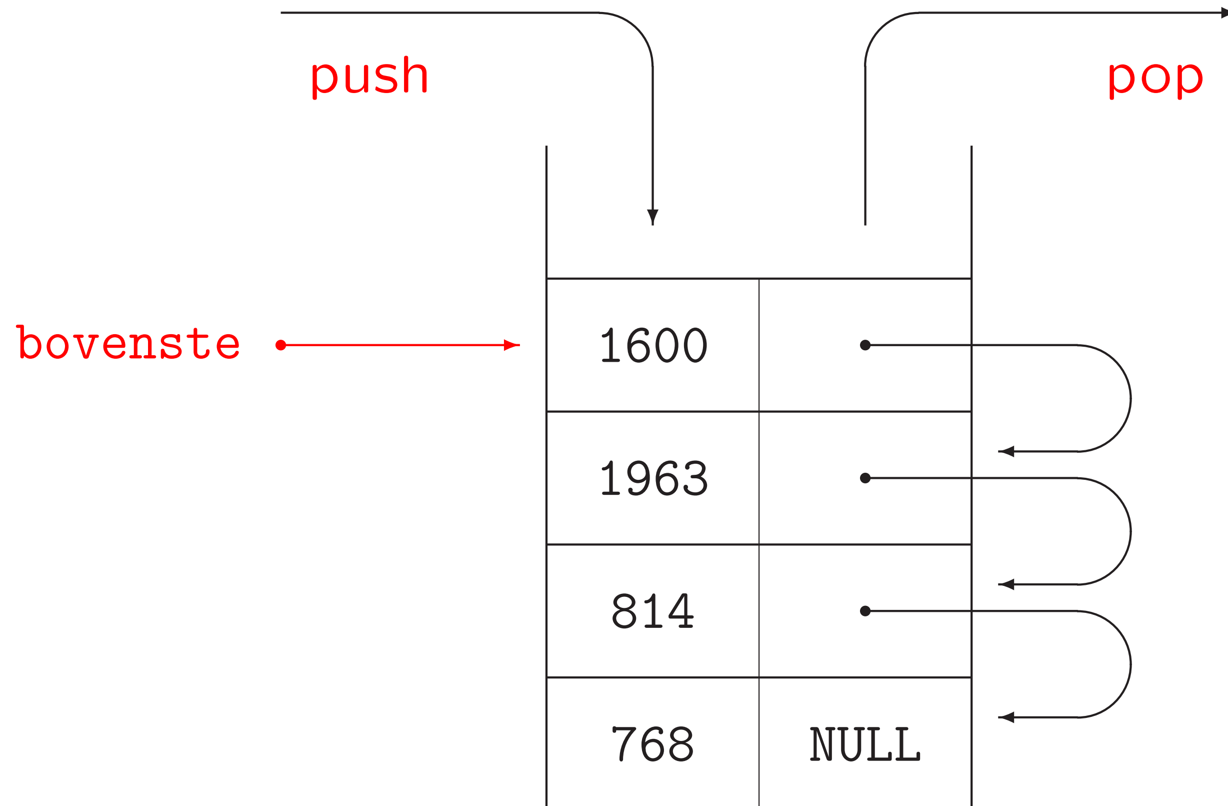


- een lege stapel aanmaken,
- testen of de stapel leeg is,
- een element toevoegen (*push*),
- het laatst-toegevoegde element eruithalen (*pop*),
- soms: kijken of de stapel al vol is.

Een stapel heeft dus de *LIFO-eigenschap*: LIFO = Last In First Out. Toevoegen en verwijderen gebeurt derhalve aan dezelfde kant: de *bovenkant*.

```
class stapel {  // stapel die gehele getallen bevat
  public:
    stapel ( );
    bool isstapelleeg ( );
    void zetopstapel (int);        // push
    void haalvanstapel (int&);     // pop
    ...
  private:
    // implementatie met pointers of array
};//stapel
```

C++ aanroep:	abstracte notatie:
`S.zetopstapel (768);`	S $\Leftarrow$ 768;
`S.haalvanstapel (jaartal);`	jaartal $\Leftarrow$ S;


push
pop
bovenste
1600
1963
814
768
NULL

We hebben een extra type nodig voor de vakjes waaruit de pointerlijst bestaat. De vakjes zijn opgebouwd uit een veld `info` voor een geheel getal en een veld `volgende` voor de rest van de stapel.

```
class vakje {  // een struct mag ook
  public:
    // constructor (een destructor hoeft misschien niet)
    vakje ( ) {
      info = 0; volgende = NULL; }// constructor vakje
    int info;
    vakje* volgende;
};//vakje
```

De stapel als enkelverbonden lijst:

```
class stapel {   // de stapel zelf
  public:
    stapel ( ) {
      bovenste = NULL; }// maak lege stapel
    ~stapel ( );   // destructor
    void zetopstapel (int);       // push
    void haalvanstapel (int&);    // pop
    bool isstapelleeg ( ) {       // is stapel leeg?
      return ( ( bovenste == NULL ) ? true : false );
      // of: if ( bovenste == NULL ) ...
    }//isstapelleeg
    ...
  private:                 // het begin van de lijst is
    vakje* bovenste;       // de bovenkant van de stapel
};//stapel
```

```
void stapel::zetopstapel (int getal) {       // push
  vakje* temp = new vakje;
  temp->info = getal;
  temp->volgende = bovenste;
  bovenste = temp;
}//stapel::zetopstapel

void stapel::haalvanstapel (int & getal) {   // pop
  vakje* temp = bovenste;
  getal = bovenste->info;
  bovenste = bovenste->volgende;
  delete temp;
}//stapel::haalvanstapel
```

NB Bij deze `haalvanstapel` hoef je er niet op te letten of de stapel leeg is, dat moet de gebruiker via `isstapelleeg` zelf maar doen . . .

En de destructor die de pointerlijst netjes afbreekt:

```
stapel::~stapel ( ) {
  int getal;
  while ( ! isstapelleeg ( ) )
    haalvanstapel (getal);
}//stapel::~stapel
```

Deze destructor wordt "vanzelf" aangeroepen als de betreffende variabele ophoudt te bestaan, dus aan het eind van de functie waarin de variabele gedeclareerd is.

Stapel via een array — 1

```
const int MAX = 100;
class stapel { // voor maximaal MAX integers
  public:
    stapel ( ) { bovenste = -1; }  // constructor
    void zetopstapel (int);
    void haalvanstapel (int&);
    bool isstapelleeg ( ) {
      return ( bovenste == -1 ); }
    ...
  private:
    int inhoud[MAX];
    int bovenste;      // index bovenste waarde
};//stapel
```

```
void stapel::zetopstapel (int getal) {
  bovenste++;
  inhoud[bovenste] = getal;
}//stapel::zetopstapel

void stapel::haalvanstapel (int & getal) {
  getal = inhoud[bovenste];
  bovenste--;
}//stapel::haalvanstapel
```

Er is eigenlijk ook een memberfunctie `vol` nodig, bijvoorbeeld in het `private`-gedeelte gedefinieerd. Deze functie wordt dan in `zetopstapel` aangeroepen.

```
bool stapel::vol ( ) {
  return ( bovenste == MAX - 1 );
}//stapel::vol
```

```
void haalgrootstegetaluitstapel (stapel & S, int & grootste) {
  stapel hulp;
  int x;
  if ( ! S.isstapelleeg ( ) ) {
    S.haalvanstapel (grootste);
    hulp.zetopstapel (grootste);
    while ( ! S.isstapelleeg ( ) ) {
      S.haalvanstapel (x);
      if ( x > grootste )
        grootste = x;
      hulp.zetopstapel (x);
    }//while
    while ( ! hulp.isstapelleeg ( ) ) {
      hulp.haalvanstapel (x);
      if ( x != grootste )
        S.zetopstapel (x);
    }//while
  }//if
}//haalgrootstegetaluitstapel
```

Merk op dat de precieze implementatie van de stapel er niet toe doet, evenmin als in het volgende voorbeeld.

```
int main ( ) {   // een main die de stapel gebruikt
  stapel S;
  int getal = 0;
  while ( getal >= 0 ) {   // zet getallen > 0 op stapel
    S.drukaf ( );   // nog te schrijven memberfunctie
    cout << "getal > 0: push; = 0: pop; < 0 stop" << endl;
    cin >> getal;
    if ( getal > 0 )
      S.zetopstapel (getal);
    else
      if ( ( getal == 0 ) && ( ! S.isstapelleeg ( ) ) ) {
        S.haalvanstapel (getal);
        cout << getal << " van stapel gehaald " << endl;
      }//if
  }//while
  return 0;
}//main
```

In de Standard Template Library (STL) zitten ook al complete stapels ("stacks"):

```
#include <stack>

stack<int> S;
S.push (1963); S.push (2011);
while ( ! S.empty ( ) ) {
  cout << S.top ( ) << endl; S.pop ( ); }//while
```

Tussen < > staat het soort elementen dat op de stapel komt.

In de STL zitten overigens bijvoorbeeld ook vectoren, verzamelingen ("sets") en rijen ("queues").

Een rij (queue; denk aan een rij voor een kassa) is een reeks elementen van hetzelfde type, bijvoorbeeld karakters, met de volgende toegestane operaties:

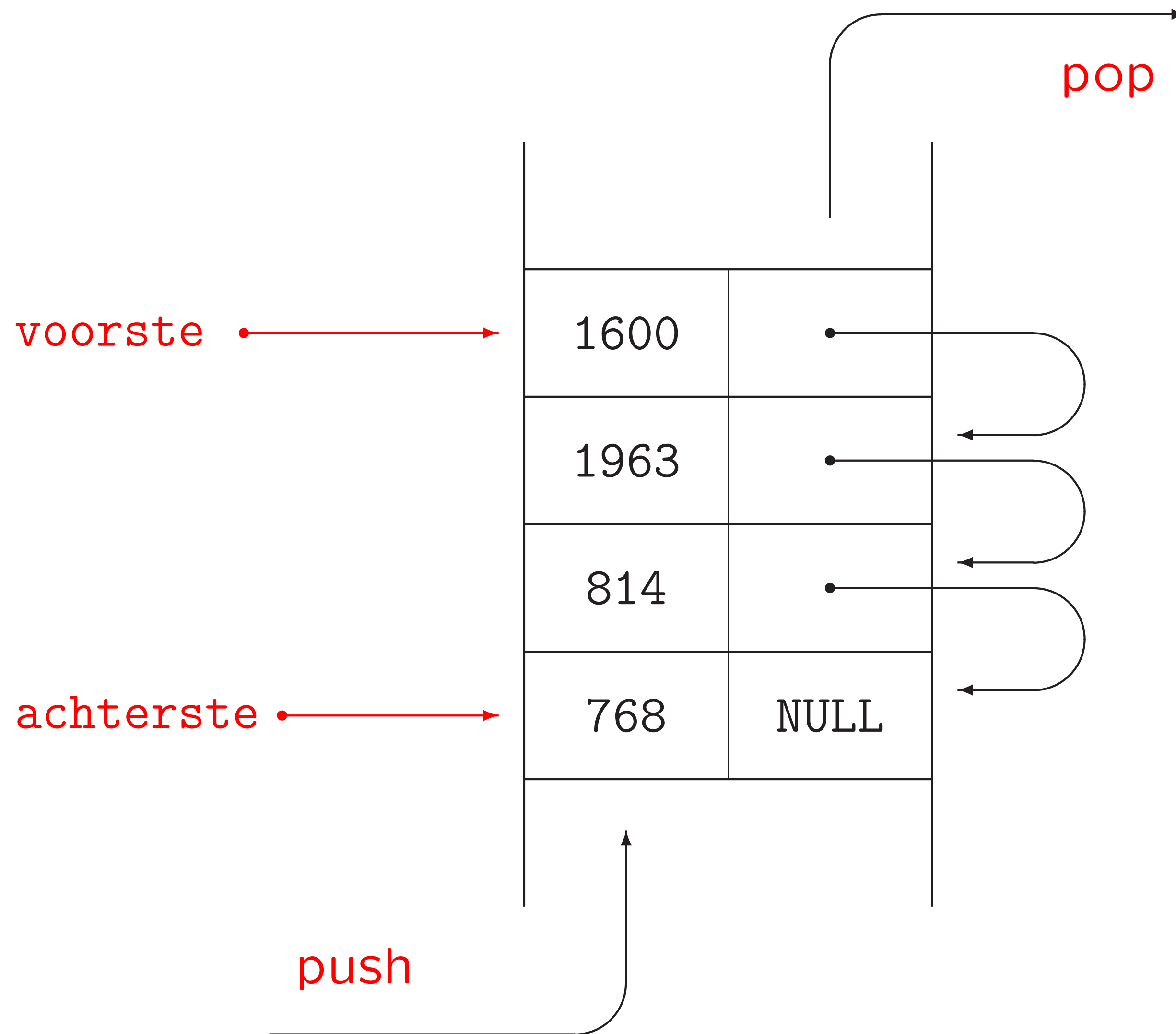
- een lege rij aanmaken,
- testen of de rij leeg is,
- een element toevoegen (*push*),
- het eerst-toegevoegde element eruithalen (*pop*),
- soms: kijken of de rij al vol is.

Een rij heeft dus de *FIFO-eigenschap*: FIFO = First In First Out. Toevoegen en verwijderen gebeurt derhalve aan verschillende kanten: *achteraan* respectievelijk *vooraan*.

```
class rij {  // rij die gehele getallen bevat
  public:
    rij ( );
    bool isrijleeg ( );
    void zetinrij (int);        // push
    void haaluitrij (int&);     // pop
    ...
  private:
    // implementatie met pointers of array
};//rij
```

C++ aanroep:	abstracte notatie:
`R.zetinrij (768);`	R $\Leftarrow$ 768;
`R.haaluitrij (jaartal);`	jaartal $\Leftarrow$ R;

Rij via pointers — 1



De rij als enkelverbonden lijst:

```
class rij {
  public:
    rij ( ) {  // constructor: maak lege rij
      voorste = NULL; achterste = NULL; }
    ~rij ( );  // destructor
    bool isrijleeg ( ) { return ( voorste == NULL ); }
    void zetinrij (int getal);
    void haaluitrij (int & getal);
    ...
  private:
    // int aantal; is niet nodig, misschien wel handig
    // dan moet in rij ( ) erbij: aantal = 0;
    vakje* voorste;
    vakje* achterste;
};//rij
```

Vraag: waar wordt toegevoegd en waar wordt verwijderd?

```
void rij::zetinrij (int getal) {   // push
// toevoegen achteraan de lijst
  vakje* temp;
  temp = new vakje;  // constructor vakje wordt aangeroepen!
  temp->info = getal;
  temp->volgende = NULL;  // doet constructor ook al
  if ( achterste != NULL ) { // of: if ( ! isrijleeg ( ) )
    achterste->volgende = temp;  // achteraan plaatsen
  else   // rij was leeg
    voorste = temp;
  achterste = temp;
}//rij::zetinrij
```

```
// aanname: de rij is niet leeg
void rij::haaluitrij (int & getal) {   // pop
// verwijderen vooraan de lijst
  vakje* temp = voorste;
  getal = voorste->info;
  voorste = voorste->volgende;  // nieuwe voorste
  if ( achterste == temp )  // rij bestond uit 1 element
    achterste = NULL;
  delete temp;
}//rij::haaluitrij

rij::~rij ( ) {  // destructor
  int getal;
  while ( ! isrijleeg ( ) )
    haaluitrij (getal);
}//rij::~rij
```

Rij via een array — 1

```
const int MAX = 100;
class rij { // voor maximaal MAX-1 (!!!) integers
  public:
    rij ( ) { voor = 1; achter = 0; }
    void zetinrij (int);
    void haaluitrij (int&);
    bool isrijleeg ( ) {
      return ( (achter + 1) % MAX == voor ); }
    ...
  private:
    int inhoud[MAX];
    int voor, achter;    // voor is de array-index van het
    // voorste getal uit de rij, achter die van het laatste
};//rij
```

Om een volle rij van een lege rij te kunnen onderscheiden, blijft er altijd minstens één array-element ongebruikt, namelijk het vakje `inhoud[voor-1]`. Bij aanvang begint de rij bij index `1`.

```
void rij::zetinrij (int getal) {
// achteraan toevoegen
  achter = (achter + 1) % MAX;  // circulair array
  inhoud[achter] = getal;
}//rij::zetinrij

void rij::haaluitrij (int & getal) {
// vooraan verwijderen
  getal = inhoud[voor];
  voor = (voor + 1) % MAX;      // circulair array
}//rij::haaluitrij
```

Er is eigenlijk ook een memberfunctie `vol` nodig:

```
bool rij::vol ( ) {
  return ( (achter + 2) % MAX == voor );       (*)
}//rij::vol
```

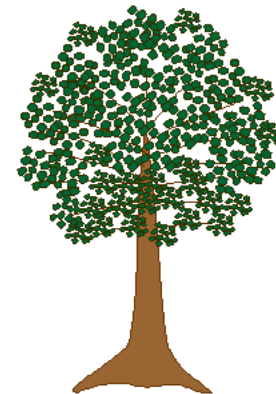
Deze functie moet je dan aanroepen voordat je `zetinrij` aanroept.

In plaats van standaard één vakje leeg te laten kunnen we ook een membervariabele `aantal` opnemen die het actuele aantal rijelementen bijhoudt. Bijvoorbeeld `(*)` wordt dan: `return ( aantal == MAX );`.

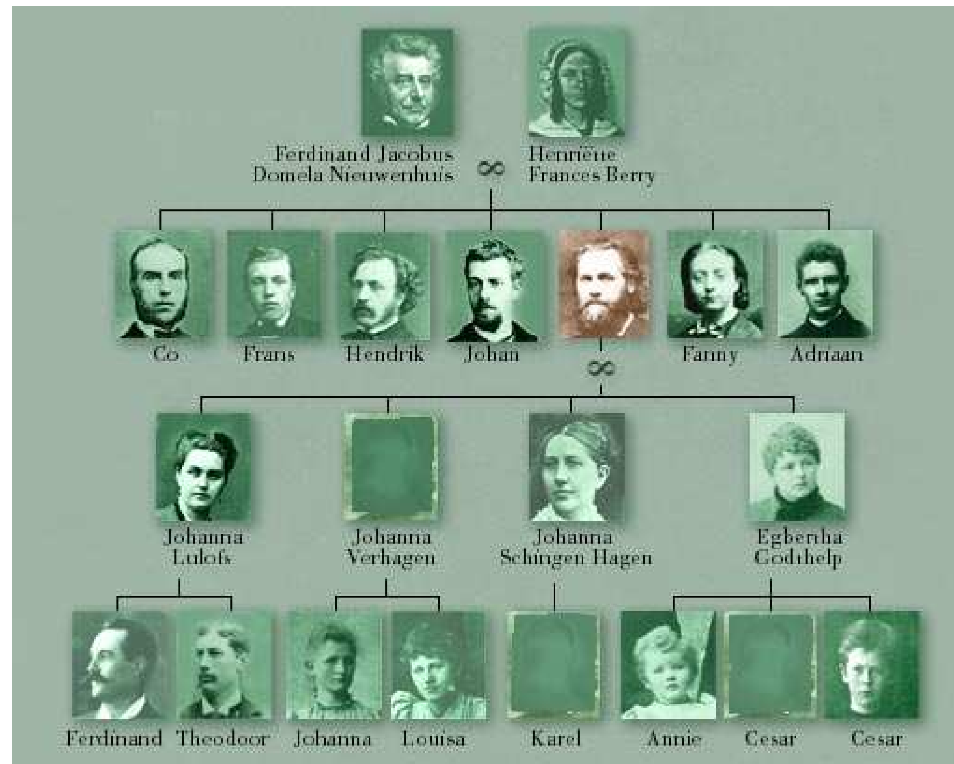
Definitie: een boom is een samenhangende (= uit één stuk bestaande) ongerichte graaf zonder cykels (= kringen).

Wijs een speciale knoop aan, de *wortel*. Teken de wortel bovenaan en alle paden vanuit de wortel naar beneden: dit geeft een hiërarchische structuur die lijkt op een stamboom. Dit heet ook wel een georiënteerde boom.

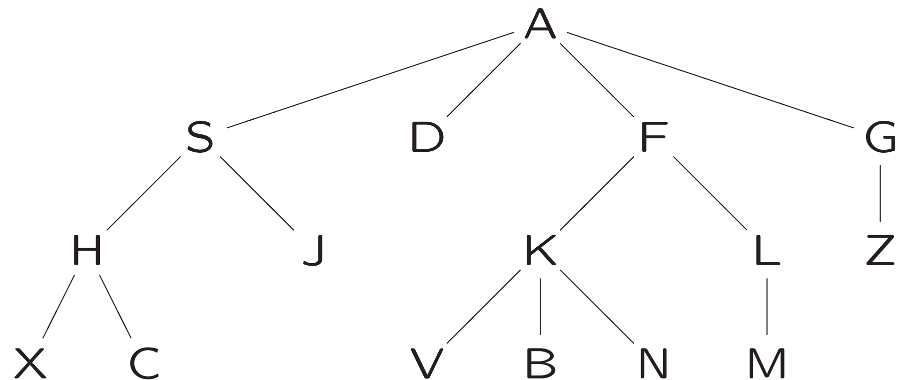
Terminologie: kind $\longleftrightarrow$ ouder,
afstammeling $\longleftrightarrow$ voorouder.



In een georiënteerde boom hebben we dus ouder-kind relaties tussen knopen.



Stambomen kunnen op vele verschillende manieren vormgegeven worden. Is dit overigens een (stam)boom?



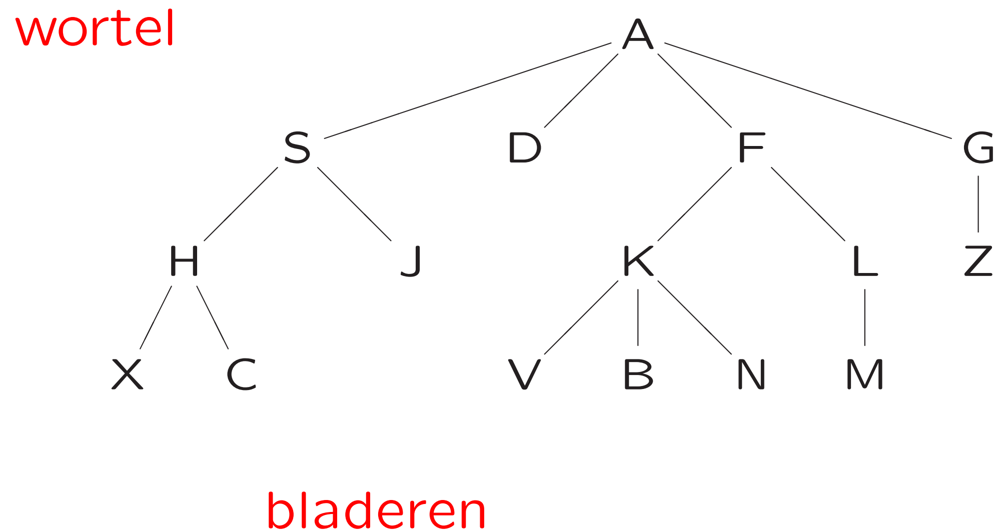
De kinderen van (de knoop met) A zijn S, D, F en G.

De ouder van J is S.

Alle afstammelingen van F zijn K, L, V, B, N en M.

Alle voorouders van X zijn H, S en A.

Terminologie: takken, knopen, nivo, hoogte, wortel, bladeren $\longleftrightarrow$ interne knopen



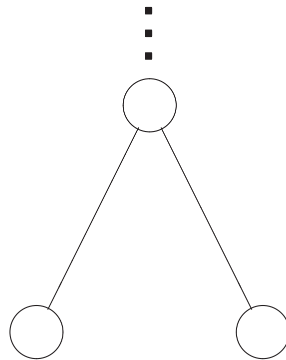
De wortel (hier A) is de *enige* ingang tot de boom.

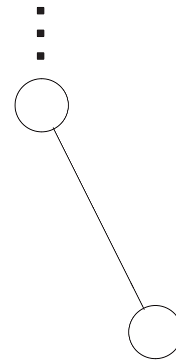
Bladeren zijn knopen die geen kinderen hebben; hier dus: X, C, J, D, V, B, N, M, Z. Knopen met een of meer kinderen noemen we interne knopen.

De wortel zit op nivo 1 (sommigen vinden dit nivo 0); op nivo 3 bevinden zich: H, J, K, L en Z.

De hoogte van een boom is het grootste nivo dat voorkomt. De hoogte van de voorbeeldboom is dus 4.

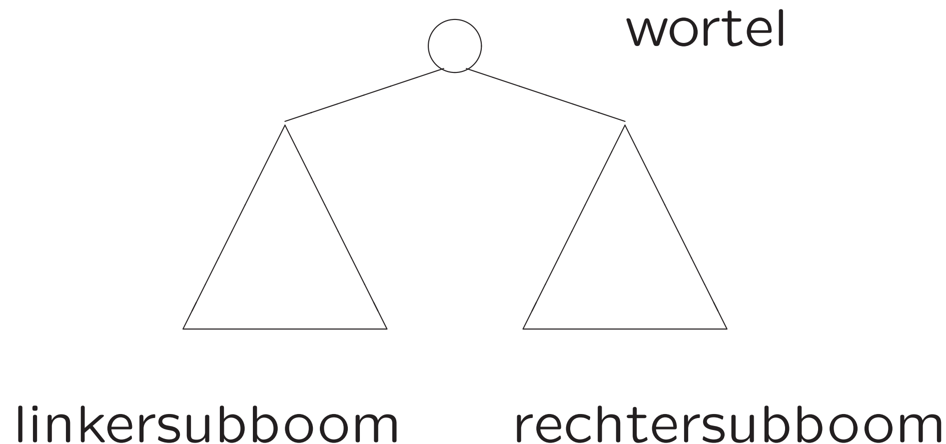
Een binaire boom is een boom waarin elke knoop ofwel nul, ofwel één ofwel twee kinderen heeft; als een knoop twee kinderen heeft dan is het ene kind het **linkerkind**, het andere het **rechterkind**; als een knoop één kind heeft, dan is dit ofwel een linkerkind, ofwel een rechterkind.





linkerkind rechterkind één kind: rechterkind

Recursieve definitie: een binaire boom is een eindige verzameling knopen die ofwel leeg is, ofwel bestaat uit een knoop (de wortel) en twee disjuncte verzamelingen knopen die samen de rest van alle knopen vormen, die beide ook weer een binaire boom zijn: de linkersubboom en de rechtersubboom.



```
class knoop {  // een struct mag ook
  public:
    knoop ( ) {  // constructor
      info = 0; links = NULL; rechts = NULL; }
    int info;
    knoop* links;
    knoop* rechts;
};//knoop
```

De binaire boom wordt gerepresenteerd door middel van een pointer naar de wortel:

```
  knoop* wortel; // de ingang tot de binaire boom
```

Het is netter de binaire boom met een klasse te represen-teren:

```
class binaireboom {
  public:
    binaireboom ( ) { wortel = NULL; }
    void WLR ( ) { preorde (wortel); }
    void LWR ( ) { symmetrisch (wortel); }
    void LRW ( ) { postorde (wortel); }
         .
         .
  private:
    knoop* wortel;
    void preorde (knoop* root);
    void symmetrisch (knoop* root);
    void postorde (knoop* root);
         .
         .
};//binaireboom
```

WLR (preorde):

bezoek wortel
doorloop linkersubboom WLR
doorloop rechtersubboom WLR

LWR (symmetrisch):

doorloop linkersubboom LWR
bezoek wortel
doorloop rechtersubboom LWR

LRW (postorde): analoog

```
void binaireboom::preorde (knoop* root) {
  if ( root != NULL ) {
    cout << root->info << endl;
    preorde (root->links);
    preorde (root->rechts);
  }//if
}//preorde

void binaireboom::symmetrisch (knoop* root) {
  if ( root != NULL ) {
    symmetrisch (root->links);
    cout << root->info << endl;
    symmetrisch (root->rechts);
  }//if
}//symmetrisch
```

We tellen *recursief* het aantal knopen van een binaire boom met ingang wortel.
Aanroep: `int tellen = aantal (wortel);`

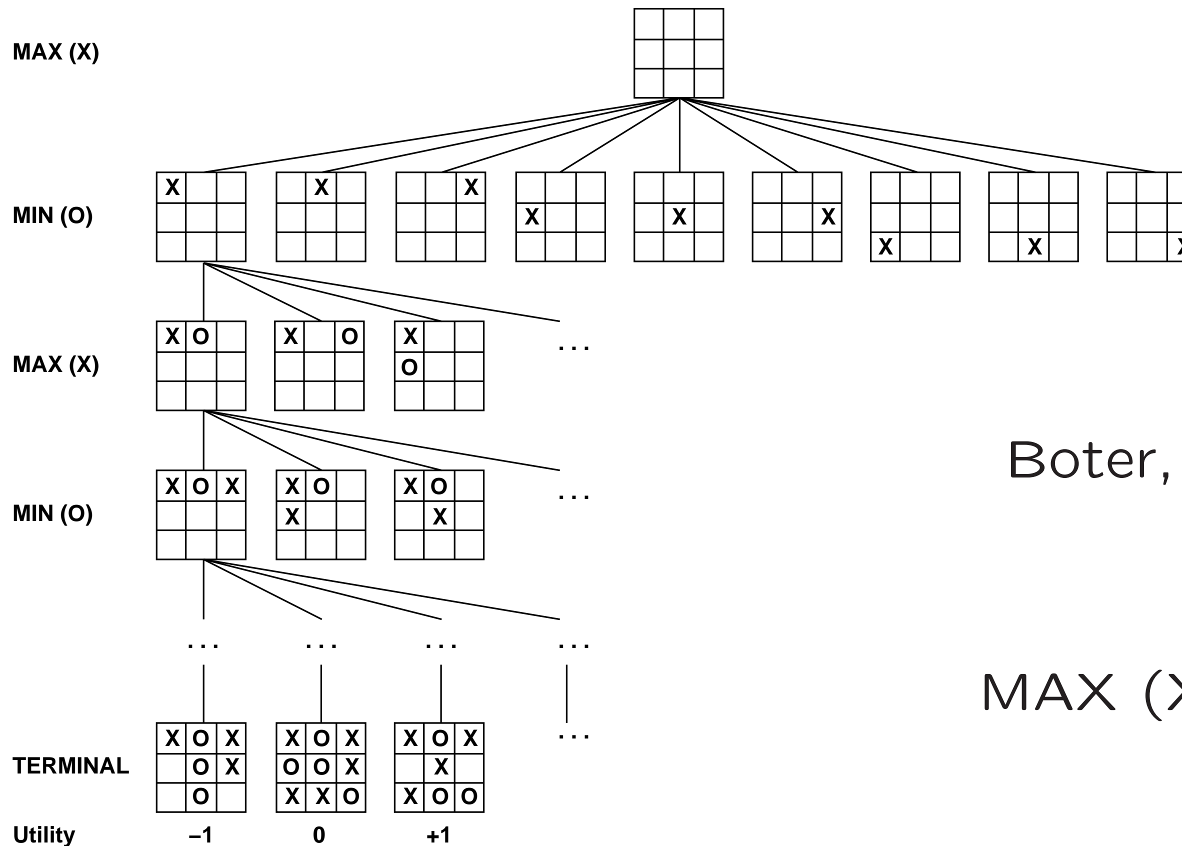
```
int aantal (knoop* root) {
  if ( root == NULL )       // lege boom
    return 0;
  else
    return ( 1 + aantal (root->links)
               + aantal (root->rechts) );
}//aantal
```

Merk op dat hier eigenlijk een preorde-wandeling wordt gedaan.

We breken de binaire boom met ingang wortel helemaal af: hiertoe wordt eerst de linkersubboom (recursief) weggegooid, daarna de rechtersubboom en tenslotte de wortel zelf. Aanroep: `breekaf (wortel);`

```
void breekaf (knoop* & root) {
  if ( root != NULL ) {
    breekaf (root->links);      L
    breekaf (root->rechts);     R
    delete root;                W
    root = NULL;
  }//if
}//breekaf
```

Bomen, en niet alleen binaire, worden vaak gebruikt om spellen als schaken en go te analyseren (“α-β-algoritme”).



Boter, kaas en eieren

twee spelers:

MAX (X) en MIN (O)

Het *aantal vervolgpartijen* vanuit een gegeven positie in de wortel is het aantal bladeren in deze boom. Het kan berekend worden zonder de boom "echt" te maken, zie het college over recursie!

Voor Boter, kaas en eieren is dit 255.168, waarvan overigens 131.184 gewonnen door de beginspeler, 77.904 door de ander en 46.080 remise.

Als je C++-code over meerdere files verdeelt, helpt een `makefile` bij het compileren (aanroep: `make`). Stel je hebt:

file `stapel.h`:

```
class stapel {
  ...
  void hvs (int&);
};//stapel
```

file `stapel.cc`:

```
#include "stapel.h"
// implementatie van
// prototypes uit
// stapel.h
void stapel::hvs (int& i) {
  ...
}//stapel::hvs
```

file `hoofd.cc`:

```
#include <iostream>
#include "stapel.h"
...
int main ( ) {
  stapel S;
  ...
}//main
```

De makefile ziet er dan bijvoorbeeld uit als (let op tabs!):

```
all: stapel.o hoofd.o
	g++ -Wall -o alles stapel.o hoofd.o
stapel.o: stapel.cc stapel.h
	g++ -Wall -c stapel.cc
hoofd.o: hoofd.cc stapel.h
	g++ -Wall -c hoofd.cc
```

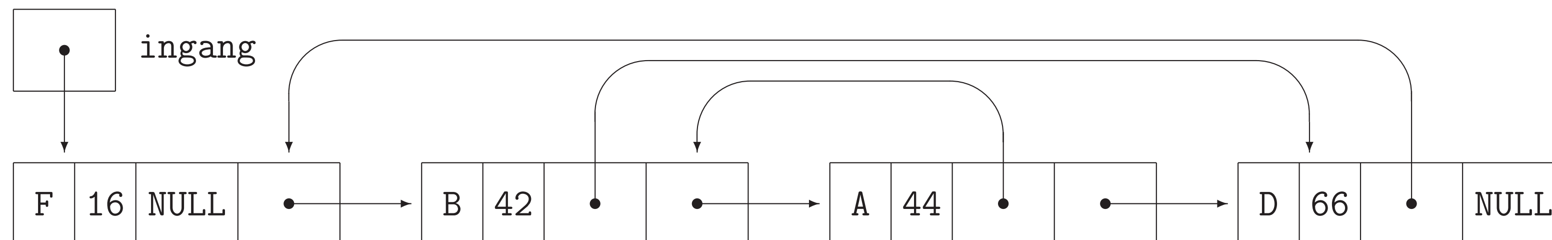
Zie make-dictaat!

Opgave 4 van het tentamen van 8 januari 2007:

Gegeven is het volgende type:

```
class info { public: info* volgl; info* volgg;
                     char letter; int getal; };
```

Met behulp hiervan worden rijtjes (lijstjes) met letter-getal combinaties opgebouwd; alle gebruikte letters en getallen verschillen onderling. Het veld `volgl` bevat een pointer naar het `info`-object met de eerstvolgende alfabetisch grotere letter (of `NULL`), het veld `volgg` een pointer naar het object met het eerstvolgende grotere getal (of `NULL`). Een voorbeeld (`ingang` van type `info*`; deze wijst steeds het object met het kleinste getal aan), waarbij `volgg` de meest rechtse pointer in ieder object is:



Het object met `B` en `42` erin heeft hier een pointer `volgl` naar het object met `D` en `66`, en een (horizontale) pointer `volgg` naar het object met `A` en `44`.

a. Schrijf een C++-functie `voegtoe (ingang,let,get)` die een nieuw object met getal `get` en letter `let` erin vooraan de structuur (met `ingang` van type `info*` als ingang) toevoegt. Neem aan dat het eerste getal uit de "oude" lijst groter is dan `get`. Maak de nieuwe `volgl`-pointer `NULL`, en verander (nog) niets aan de andere `volgl`-pointers.

Opgave 4 van het tentamen van 8 januari 2007, vervolg:

b. Schrijf een C++-functie `verwijder (ingang)` die het eerste object uit de lijst (met `ingang` van type `info*` als ingang) verwijdert indien in dat vakje *niet* de letter `D` en het getal `66` zitten. Denk aan de lege lijst. Verander niets aan `volgl`-pointers.

c. Schrijf een C++-functie `verwissel (ingang)` die eerste en tweede (via de `volgg`-pointer) object verwisselt (dus *niet* de inhouden), indien deze bestaan en het getal uit het eerste object groter is dan dat uit het tweede, en anders niets doet.

d. In de functies bij **a**, **b** en **c** staat in de heading de parameter `ingang`. Deze heb je call by value of call by reference doorgegeven (met een `&`). Maakt het voor de werking van deze functies verschil uit of die `&` erbij staat? Leg duidelijk uit.

e. Vul de functie `voegtoe` van **a** aan zodat na afloop alle `volgl`-pointers goed staan. Neem aan dat er minstens één letter $<$ `let` in de lijst voorkomt. Tip: zoek de grootste kleinere.

- college dinsdag 5 december in Gorlaeus 3
- werk aan de vierde programmeeropgave — de deadline is op 9 december 2011
- bezoek vragenuren en (werk)colleges
- lees dictaat Hoofdstuk 5
- maak opgaven 57/61 uit het opgavendictaat
- nog twee colleges: Algoritmen en Java/Qt/MATLAB
- `http://www.liacs.nl/home/kosters/pm/`