Programmeermethoden

Datastructuren: stapels, rijen en binaire bomen

Walter Kosters en Jonathan Vis

week 12: 27 november–1 december 2023

www.liacs.leidenuniv.nl/~kosterswa/pm/

Gomoku programmeren we als volgt:

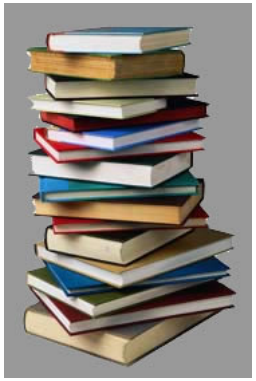
- week 1 (“10”): pointerpracticum, opgave lezen
- week 2 (“11”): klassen, pointerbord, meerdere files, ruw spelen
- week 3 (“12”): spel helemaal in orde maken, stapel
- week 4 (“13”): (vervolgpartijen), experiment (`gnuplot`), verslag

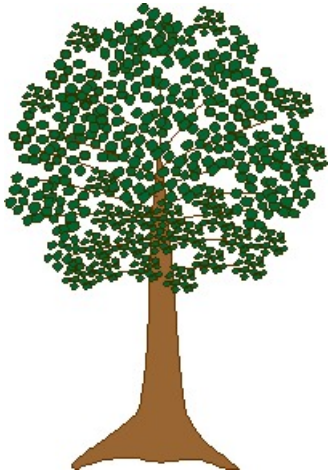
`www.liacs.leidenuniv.nl/~kosterswa/pm/op4pm.php`

In de informatica worden Abstracte DataTypen (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 DataType 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 DataType 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 datatype 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)`.

Klasse verzameling — 2

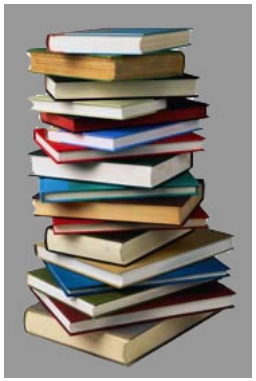
```
verzameling::verzameling ( ) {
  for ( int i = 0; i < n; i++ ) inhoud[i] = false;
}//verzameling::verzameling
bool verzameling::isleeg ( ) {
  for ( int i = 0; i < n; i++ )
    if ( inhoud[i] ) return false;
  return true;
}//verzameling::isleeg
bool verzameling::ziterin (int i) {
  return inhoud[i];
}//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 ("emuleert" {6,10} ∩ {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:



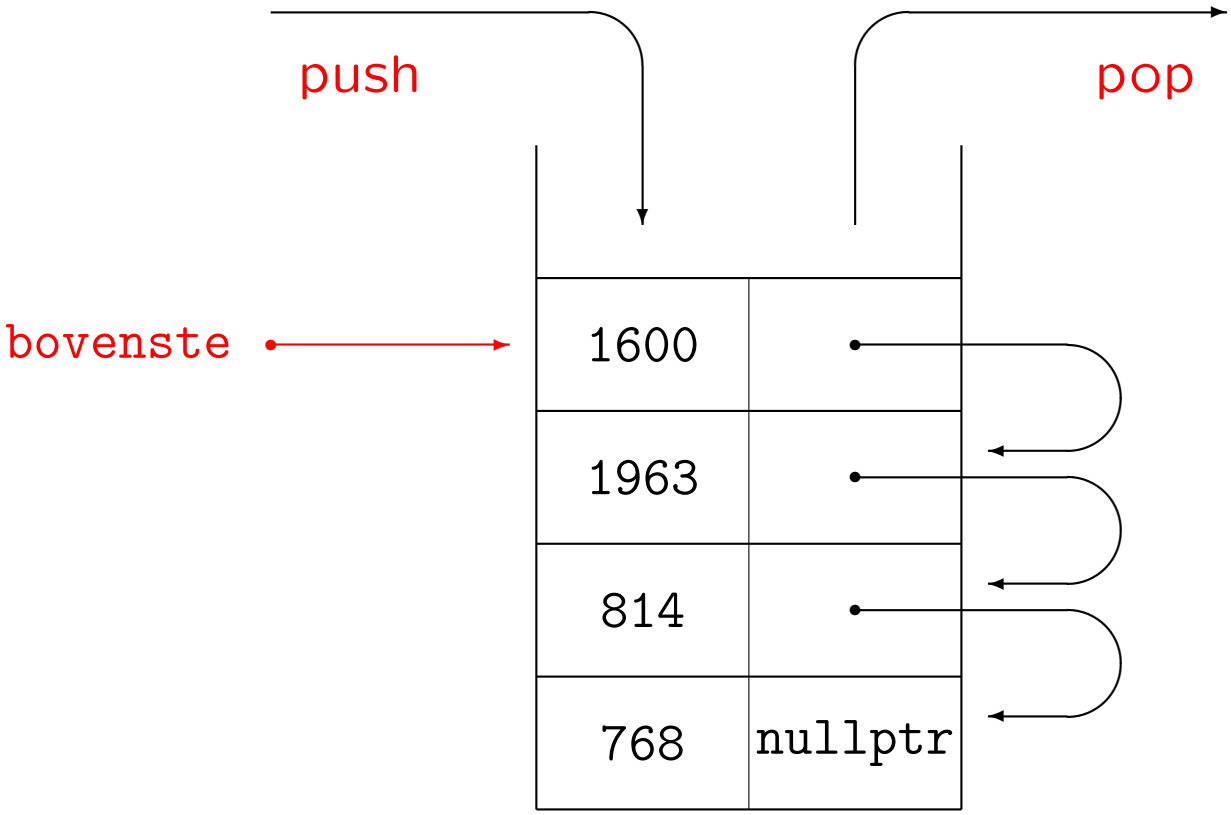
- een lege stapel aanmaken,
- testen of de stapel leeg is,
- een element toevoegen (*push*),
- het laatst-toegevoegde element eruithalen (*pop*),
- soms: kijken of de stapel al vol is.

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, let op &
    ...
  private:
    // implementatie met pointers of array
};//stapel
```

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

Stapel via pointers — 1



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 ( ) {                          // beter niet "inline"
      info = 0; volgende = nullptr; }// constructor vakje
    int info;
    vakje* volgende;
};//vakje
```

De stapel als enkelverbonden lijst:

```
class stapel {   // de stapel zelf
  public:
    stapel ( ) {
      bovenste = nullptr; }// maak lege stapel
    ~stapel ( );   // destructor
    void zetopstapel (int);       // push
    void haalvanstapel (int&);    // pop
    bool isstapelleeg ( ) {       // is stapel leeg?
      return ( ( bovenste == nullptr ) ? true : false );
      // of: if ( bovenste == nullptr ) ...
    }//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.

Vaak wordt hiervan een aparte `verwijder`-functie gemaakt, met destructor: `stapel::~stapel ( ) { verwijder ( ); }`

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
```

Stapel: gebruik in functies

```
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.

Stapel: gebruik in main

```
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 (2002); S.push (2023);
while ( ! S.empty ( ) ) {
  cout << S.top ( ) << endl; S.pop ( ); }//while
```

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

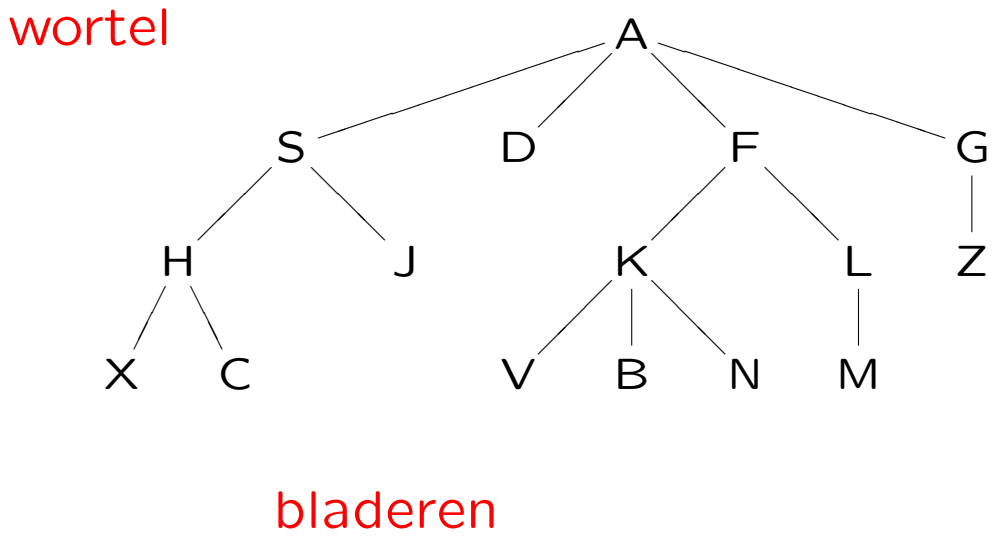
In de STL zitten overigens bijvoorbeeld ook vectoren, verzamelingen ("sets") en rijen ("queues").

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.

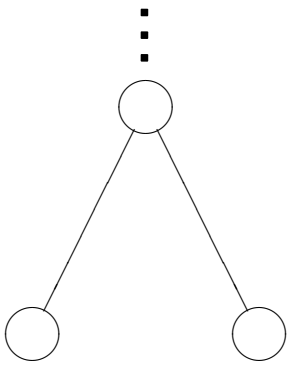
Een rij heeft dus de *FIFO-eigenschap*: FIFO = First In First Out. Toevoegen en verwijderen gebeuren dus aan verschillende kanten: *achteraan* respectievelijk *vooraan*.

Definitie: een boom is een "samenhangende ongerichte graaf zonder cykels", met één speciale knoop: de *wortel*.

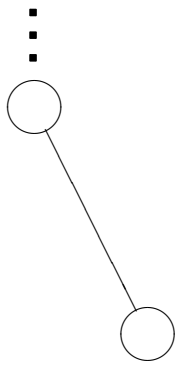


De kinderen van A zijn S, D, F en G; S is de ouder van J. Verder zijn K, L, V, B, N en M de afstammelingen van F. En H, S en A zijn de voorouders van X.

Een binaire boom is een boom waarin elke knoop ofwel nul, ofwel één, ofwel twee kinderen heeft: het *linkerkind* en het *rechterkind*. Als een knoop één kind heeft, dan is dit ofwel een linkerkind, ofwel een rechterkind.

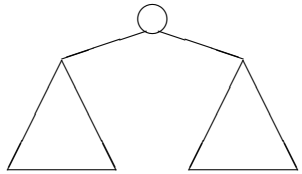


linkerkind rechterkind



één kind: rechterkind

Recursief gedefinieerd: een binaire boom
is leeg, of bestaat uit een wortel,
een linker- en een rechtersubboom.



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

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

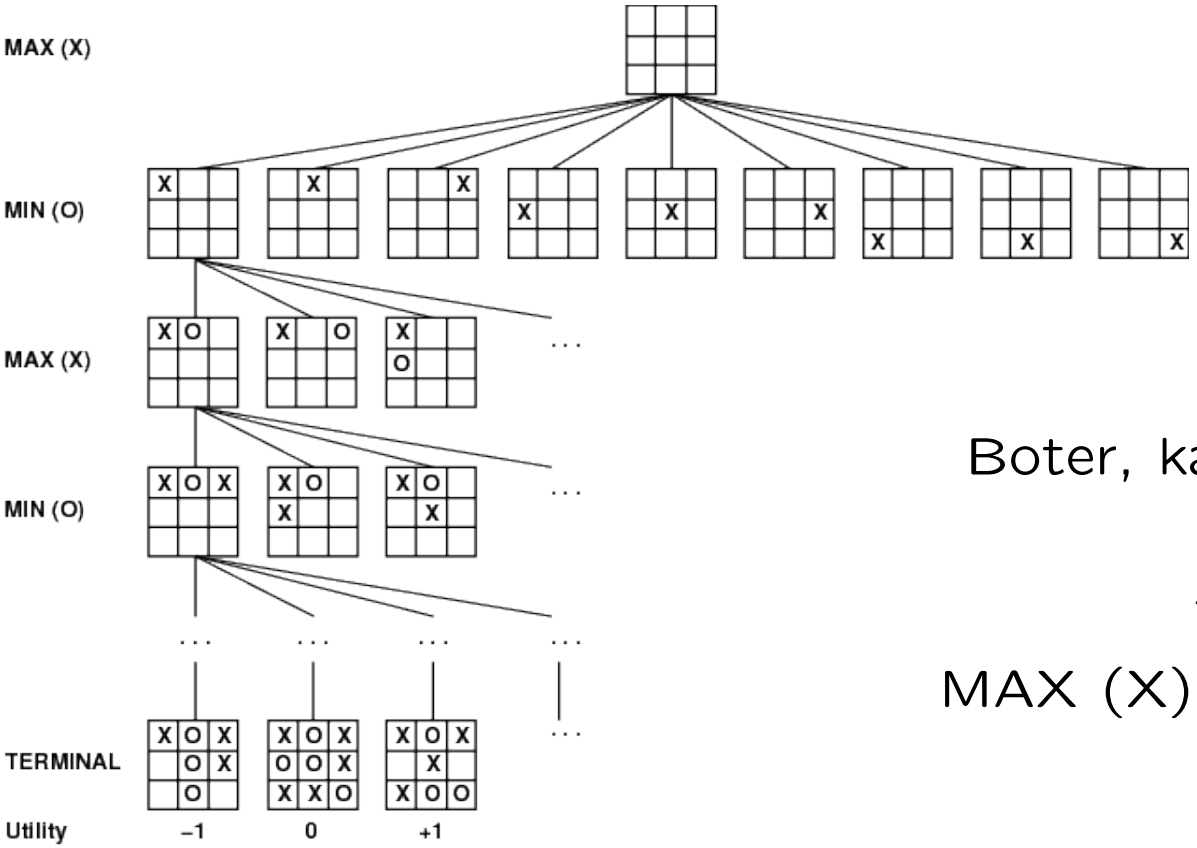
```
void binaireboom::preorde (knoop* root) {          // WLR
  if ( root != nullptr ) {          // Wortel-Links-Rechts
    cout << root->info << endl;
    preorde (root->links);
    preorde (root->rechts);
  }//if
}//preorde

void binaireboom::symmetrisch (knoop* root) {      // LWR
  if ( root != nullptr ) {          // Links-Wortel-Rechts
    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 == nullptr )  // lege boom
    return 0;
  else
    return ( 1 + aantal (root->links)
               + aantal (root->rechts) );
}//aantal
```

Hier wordt eigenlijk een preorde-wandeling gedaan.

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.

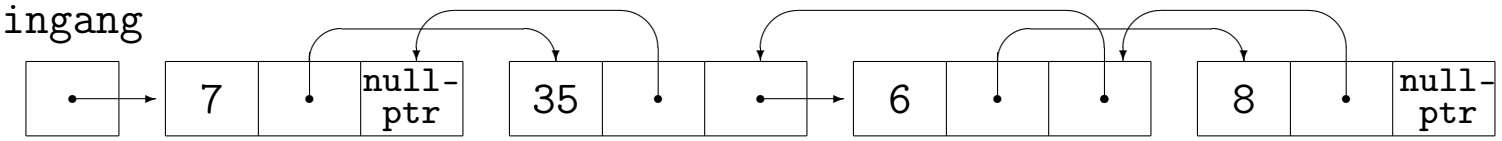
En voor zetten terugnemen kun je een stapel gebruiken: elke keer als je een zet doet, zet je de complete "oude" positie (oftewel stand) op de stapel! Soms hoef je alleen de zetten te onthouden.

Opgave 4 van het tentamen van 3 januari 2019:

Gegeven is het volgende type:

```
class object { public: int info; object* volg1; object* volg2; };
```

Met behulp hiervan kan een lijst van objecten worden opgebouwd, bestaande uit vakjes met een getal, en twee pointers. Precies een van deze twee wijst naar het volgende object, de andere naar het vorige — maar je weet niet welke. Een voorbeeld, met `ingang` van type `object*`:



a. (6) Schrijf een C++-functie `voegtoe (ingang,getal)` die een nieuw object met `getal` erin netjes vooraan de lijst met ingang `ingang` toevoegt. Je mag zelf kiezen welke van de twee pointers in het nieuwe object naar vorige en volgende object wijst. Zet in het oude eerste object (als dat bestaat) de terugwijzende pointer goed.

Opgave 4 van het tentamen van 3 januari 2019, vervolg:

b. (6) Schrijf een C++-functie `verwijder (ingang)` die het eerste object uit de lijst met ingang `ingang` netjes verwijdert, indien dit bestaat.

c. (4) Schrijf een C++-functie `hoogop (ingang)` die als er minstens twee objecten zijn en als het `info`-veld van het eerste object oneven is, dit ophoogt met het `info`-veld van het tweede object. In het voorbeeld: 7 wordt 42.

d. (3) In de functies bij **a**, **b** en **c** staat in de heading een pointer. 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? Mag het, moet het? Leg duidelijk uit.

e. (6) Schrijf een C++-functie `repareer (ingang)` die ervoor zorgt dat na afloop alle `volg1`-pointers naar het volgende, en alle `volg2`-pointers naar het vorige object wijzen.

www.liacs.leidenuniv.nl/~kosterswa/pm/tentamens.php

Uitwerking Opgave 4a van 3 januari 2019:

```
void voegtoe (object* & ingang, int getal) {
  object* nieuw = new object;
  nieuw->info = getal;
  nieuw->volg1 = nullptr; // of
  nieuw->volg2 = ingang;  // andersom

  if ( ingang != nullptr )
    if ( ingang->volg1 == nullptr )
      ingang->volg1 = nieuw;
    else
      ingang->volg2 = nieuw;
  ingang = nieuw;
}//voegtoe
```

- werk aan de vierde programmeeropgave — de deadline is op maandag 11 december 2023
- lees dictaat Hoofdstuk 5
- maak opgaven 47/51 uit het opgavendictaat
- nog twee colleges: Algoritmen, talen als Python en Oude tentamens, . . .
- www.liacs.leidenuniv.nl/~kosterswa/pm/