



Universiteit
Antwerpen

Inleiding Programmeren Rekursieve datastructuren

Toon Calders

Overzicht

- Korte herhaling
 - Opgepast met new en delete!
- const variabelen
- recursieve datastructuren

Wat zagen we vorige keer ?

- Dynamic memory allocation:
 - Reserveer geheugen in “free space”
 - Blijft behouden tot expliciet vrijgegeven
 - Toegang tot het geheugen via een pointer
- “struct” om groepjes attributen en functies die samen horen te groeperen
 - Nieuw data type
 - Kan net zoals elk ander datatype via new in de vrije ruimte aangemaakt worden

Vlugge vraag

- Bekijk volgend stukje code; welke van de volgende rvalues staan op de stack, welke in de vrije ruimte?
- matrix
- matrix2
- matrix[1]
- matrix2[0][1]

```
int main() {  
    double** matrix;  
    matrix=new double*[2];  
  
    for(int i=0;i<2;i++) {  
        matrix[i]=new double[4];  
    }  
  
    double** matrix2=matrix;  
}
```

Vlugge vraag

- Bekijk volgend stukje code; welke van de volgende rvalues staan op de **stack**, welke in de **vrije ruimte**?
- **matrix**
- **matrix2**
- **matrix[1]**
- **matrix2[0][1]**

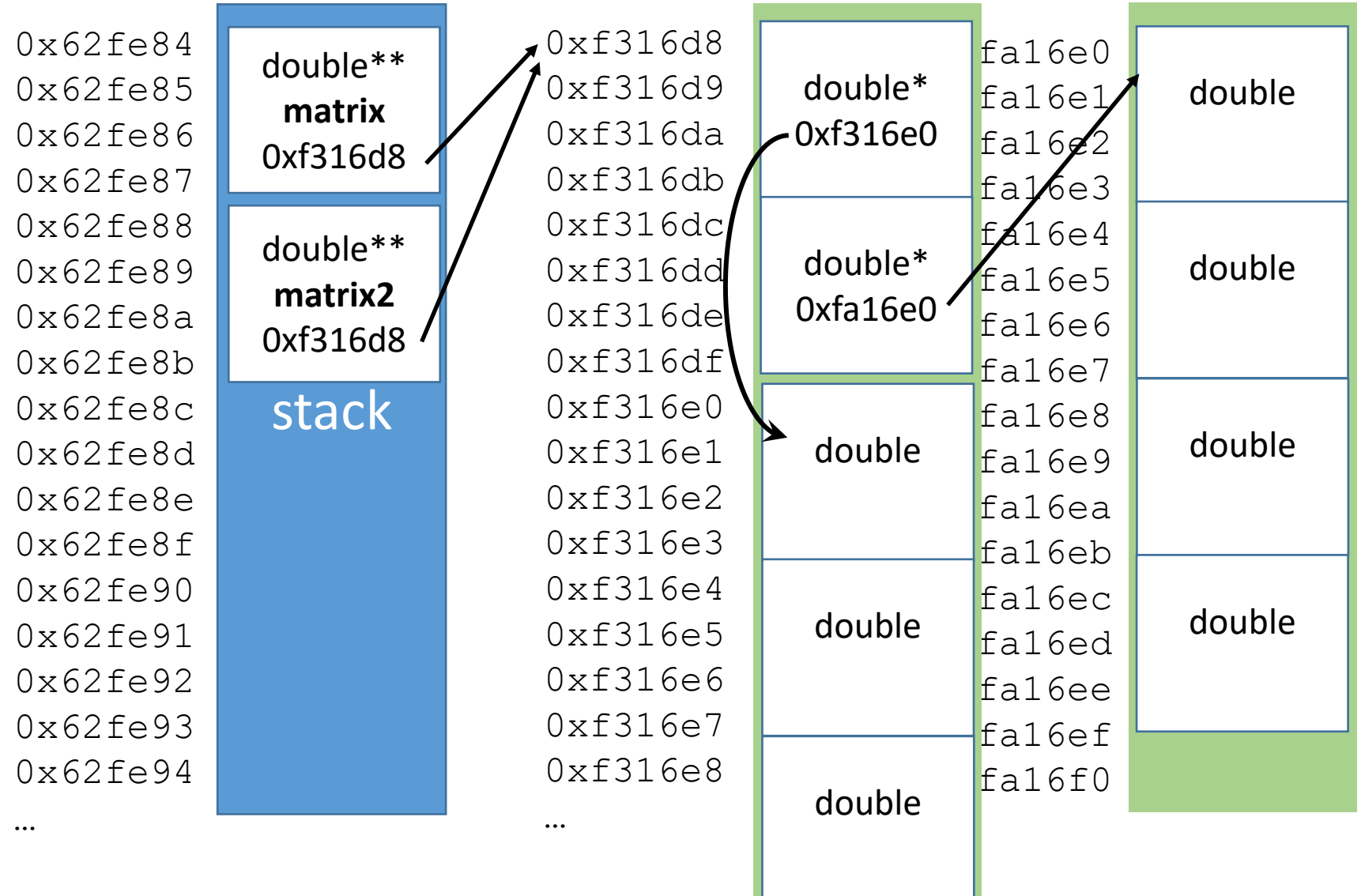
```
int main() {  
    double** matrix;  
    matrix=new double*[2];  
  
    for(int i=0;i<2;i++) {  
        matrix[i]=new double[4];  
    }  
  
    double** matrix2=matrix;  
}
```

Vlugge vraag

```
int main() {
    double** matrix;
    matrix=new double*[2];

    for(int i=0;i<2;i++) {
        matrix[i]=new double[4];
    }

    double** matrix2=matrix;
}
```



Vlugge vraag

- Welke commando's geven het geheugen gereserveerd voor matrix correct terug vrij?

```
int main() {  
    double** matrix;  
    matrix=new double*[2];  
  
    for(int i=0;i<2;i++) {  
        matrix[i]=new double[4];  
    }  
  
    double** matrix2=matrix;  
}
```

A

```
delete[] matrix;
```

B

```
for(int i=0;i<2;i++) {  
    delete[] matrix[i];  
}  
delete[] matrix;
```

C

```
delete[] matrix;  
for(int i=0;i<2;i++) {  
    delete[] matrix[i];  
}
```

Vlugge vraag

- Welke commando's geven het geheugen gereserveerd voor matrix correct terug vrij?

```
int main() {  
    double** matrix;  
    matrix=new double*[2];  
  
    for(int i=0;i<2;i++) {  
        matrix[i]=new double[4];  
    }  
  
    double** matrix2=matrix;  
}
```

memory leak

A

```
delete[] matrix;
```

B

```
for(int i=0;i<2;i++) {  
    delete[] matrix[i];  
}  
delete[] matrix;
```

C

```
delete[] matrix;  
for(int i=0;i<2;i++) {  
    delete[] matrix[i];  
}
```

access violation

New en delete: let op!

- Delete enkel geheugen dat je met new reserveerde!

```
int* grootste(int a[], int size) {  
    if (size==0) return nullptr;  
    int max=a[0];  
    int maxidx=0;  
    for (int i=1;i<size;i++) {  
        if (a[i]>max) {  
            max=a[i];  
            maxidx=i;  
        }  
    }  
    return &a[maxidx];  
}
```

```
int main() {  
    int a[5]={3,2,7,8,9};  
    int* m=grootste(a,5);  
    delete m;  
    delete a;  
}
```

New en delete: let op!

- Delete nooit iets 2 maal!

```
int* a=new int[5];  
int* b=a;  
delete[] b;  
delete[] a;  
delete[] a;
```

Struct

- Met struct kunnen we nieuwe datatypes maken door bestaande datatypes te combineren

```
struct staticList {  
    int* v;  
    int size;  
};
```

```
staticList(int n) {  
    v=new int[n];  
    size=n;  
}
```

Constructor; wordt uitgevoerd telkens een nieuwe staticList wordt gemaakt.

```
};
```

Vlugge vraag

- Bekijk volgend stukje code; wat is de output?

```
struct staticList {  
    int *v;  
    int size;  
  
    staticList(int n) {  
        v = new int[n];  
        size = n;  
    };  
};
```

```
void doubleList(staticList sl) {  
    int* newv=new int[2*sl.size];  
    for (int i=0;i<sl.size;i++) {  
        newv[i]=sl.v[i];  
    }  
    sl.size=2*sl.size;  
    delete[] sl.v;  
    sl.v=newv;  
}
```

```
int main() {  
    staticList sl(3);  
    doubleList(sl);  
    sl.v[1] = 3;  
    cout << sl.size;  
}
```

Overzicht

- Korte herhaling
 - Opgepast met new en delete!
- **const variabelen**
- recursieve datastructuren

Const variabelen

- In C++ kunnen we expliciet aangeven dat een variabele niet van waarde mag veranderen
- Waarom willen we dat?
 - Compiler kan deze informatie gebruiken om performantere code te schrijven
 - Documentatie van code (“Deze waarde verandert niet”)
 - Vermijden van fouten: soms geven we een variabele door *by reference* omwille van efficiëntie redenen en niet om waarden aan te passen; const parameters laten de compiler toe om te waarschuwen als dit toch fout loopt!
- Hoe doen we dit?
const int i=5;

Const variabelen

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

```
const int WIT=0;
const int ZWART=1;
```

Constant global variables

```
int main() {
    int a=3;
    const int b=4;
```

Constant local variable

Voorbeeld (cplusplus.org)

```
1 #include <iostream>
2 using namespace std;
3
4 const double pi = 3.14159;
5 const char newline = '\\n';
6
7 int main ()
8 {
9     double r=5.0;           // radius
10    double circle;
11
12    circle = 2 * pi * r;
13    cout << circle;
14    cout << newline;
15 }
```


Const parameters

type safety

```
int largest(const vector<int> &v)
{
    int largest=v[0];
    for(int x:v) {
        if (x>largest) {
            largest=x;
        }
    }
    return largest;
}
```

efficiency

```
int main() {
    vector<int> v={1,5,7,9,3};
    cout << largest(v) << endl;
}
```

Const parameters

- Const is “besmettelijk”; als een variabele const is, dan moeten alle functie-parameters waar die variabele *by reference* aan doorgegeven wordt, const zijn.

```
../les15-const.cpp:51:21:  
error: binding 'const  
matrix' to reference of  
type 'matrix&' discards  
qualifiers
```

```
void printMatrix(matrix m);  
matrix newMatrix(int num_r, int num_c);  
matrix multiply(matrix &A, matrix &B);  
  
matrix square(const matrix &A) {  
    return multiply(A,A);  
}  
  
int main() {  
    matrix A=newMatrix(3,3);  
    matrix B=square(A);  
}
```

Hoe lossen we dit op?

```
void printMatrix(matrix m);  
matrix newMatrix(int num_r, int num_c);  
matrix multiply(const matrix &A, const matrix &B);  
  
matrix square(const matrix &A) {  
    return multiply(A,A);  
}  
  
int main() {  
    matrix A=newMatrix(3,3);  
    matrix B=square(A);  
}
```

Voorbeeld: Const parameters

```
bool deler(int deler, int x) {  
    return (x%delel)==0;  
}  
  
bool isPriem(const int x) {  
    for (int i=2;i<x;i++) {  
        if (delel(i,x)) {  
            return false;  
        }  
    }  
    return true;  
}  
  
int main() {  
    cout << isPriem(13) << endl;
```

OK!

Const pointers en pointers naar const

- We lezen types van links naar rechts:
 - `const int * p;` // p is een pointer naar een constante integer
`int a=5; p=&a;` // toegestaan
`p=new int;` // toegestaan
`*p=3;` // **ERROR**
 - `int a=5;`
`int * const p=&a;` // p is een constante pointer naar een integer
`p=&a;` // **ERROR**
`*p=3;` // toegestaan
 - `int a=5;`
`const int * const p=&a;` // p is een constante pointer naar een constante int
`int b=3; p=&a;` // **ERROR**
`*p=3;` // **ERROR**

Vlugge vraag

- Beschouw volgende declaraties:

```
const int a=3;  
int b=5;  
vector<int> v={1, 2, 3};  
const int* p=nullptr;  
const int* const p2=&a;
```

- Welke van volgende 6 assingments zijn toegestaan?

```
*p2=4;  
p=&b;
```

```
delete p2;  
b=6;
```

```
p2=nullptr;  
*p=3;
```

Overzicht

- Korte herhaling
 - Opgepast met new en delete!
- const variabelen
- recursieve datastructuren

Noot: Pointer naar struct

- Als we een pointer naar een struct hebben, kunnen we -> gebruiken om de componenten van de struct rechtstreeks te benaderen.
 - p->x is synoniem van (*p).x

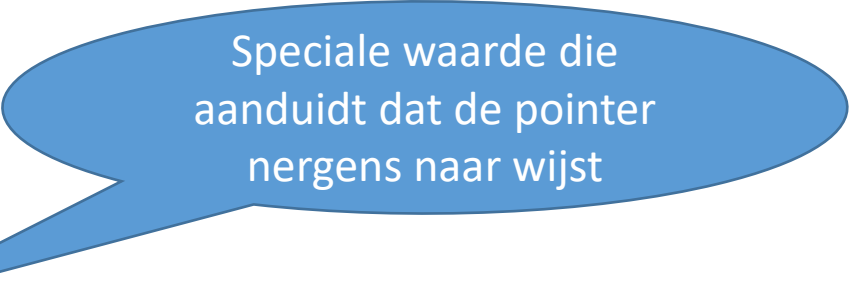
```
struct coordinaat {  
    double x;  
    double y;  
};  
  
int main() {  
    coordinaat* p = new coordinaat;  
    p->x=0.5;  
    p->y=0.7;  
    return 0;  
}
```


Rekursieve structs

- Een struct kan ook een referentie naar een object van hetzelfde struct type bevatten:

```
struct wagon {  
    int zitplaatsen;  
    wagon* volgende;  
};
```

```
wagon* wagon1=new wagon;  
wagon* wagon2=new wagon;  
wagon1->zitplaatsen=20;  
wagon2->zitplaatsen=20;  
wagon1->volgende=wagon2;  
wagon2->volgende=nullptr;
```

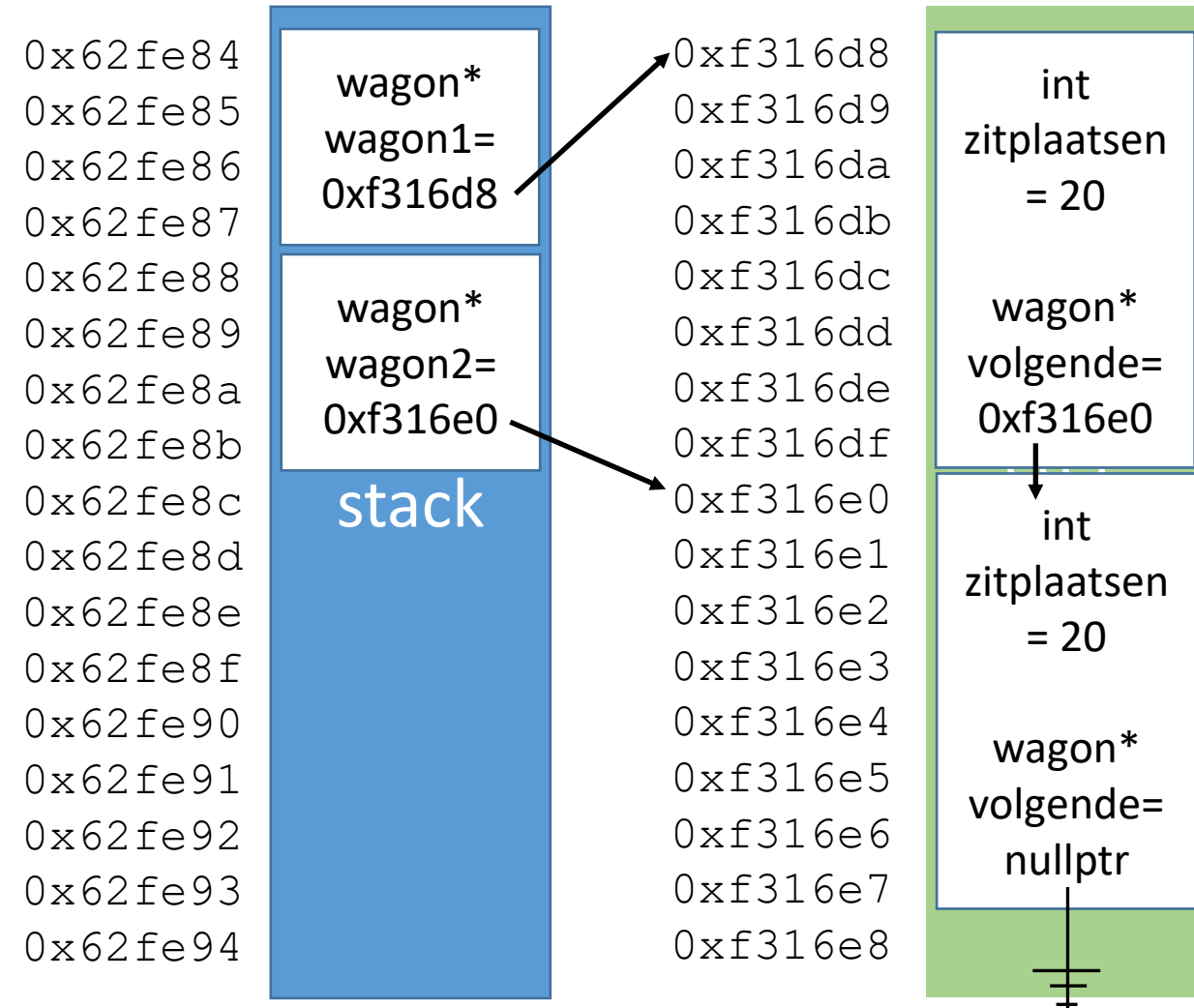


Speciale waarde die
aanduidt dat de pointer
nergens naar wijst

Rekursieve structs

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```

```
wagon* wagon1=new wagon;
wagon* wagon2=new wagon;
wagon1->zitplaatsen=20;
wagon2->zitplaatsen=30;
wagon1->volgende=wagon2;
wagon2->volgende=nullptr;
```



Recursieve structs

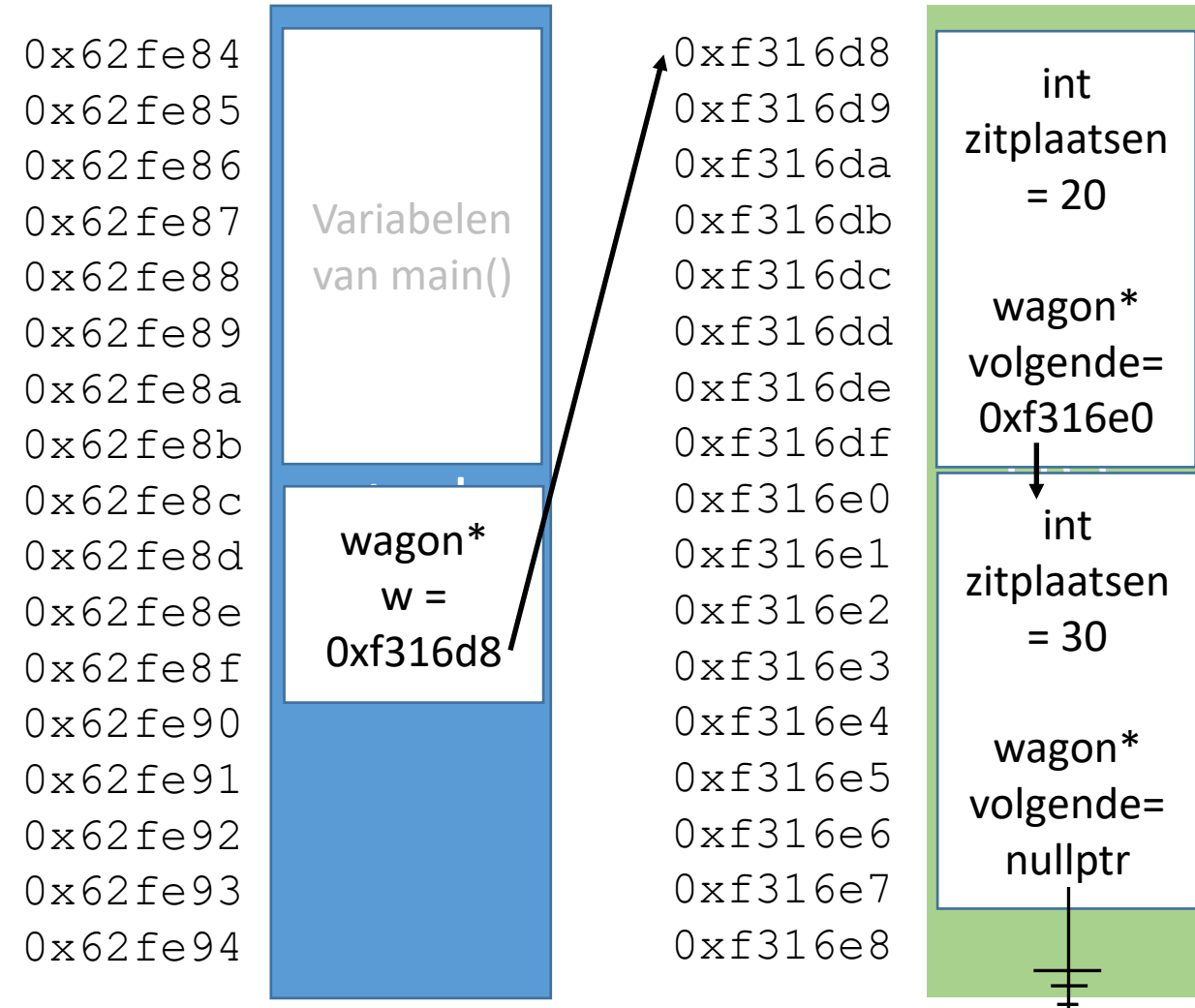
- We kunnen naar de volledige trein verwijzen door enkel een pointer naar de eerste wagon door te geven:

```
void printTrein(const wagon* w) {  
    while (w!=nullptr) {  
        cout << "Wagon met "  
              << w->zitplaatsen  
              << " plaatsen" << endl;  
        w=w->volgende;  
    }  
}
```

```
struct wagon {  
    int zitplaatsen;  
    wagon* volgende;  
};
```

Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```



Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

wagon*
w =
0xf316d8

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0

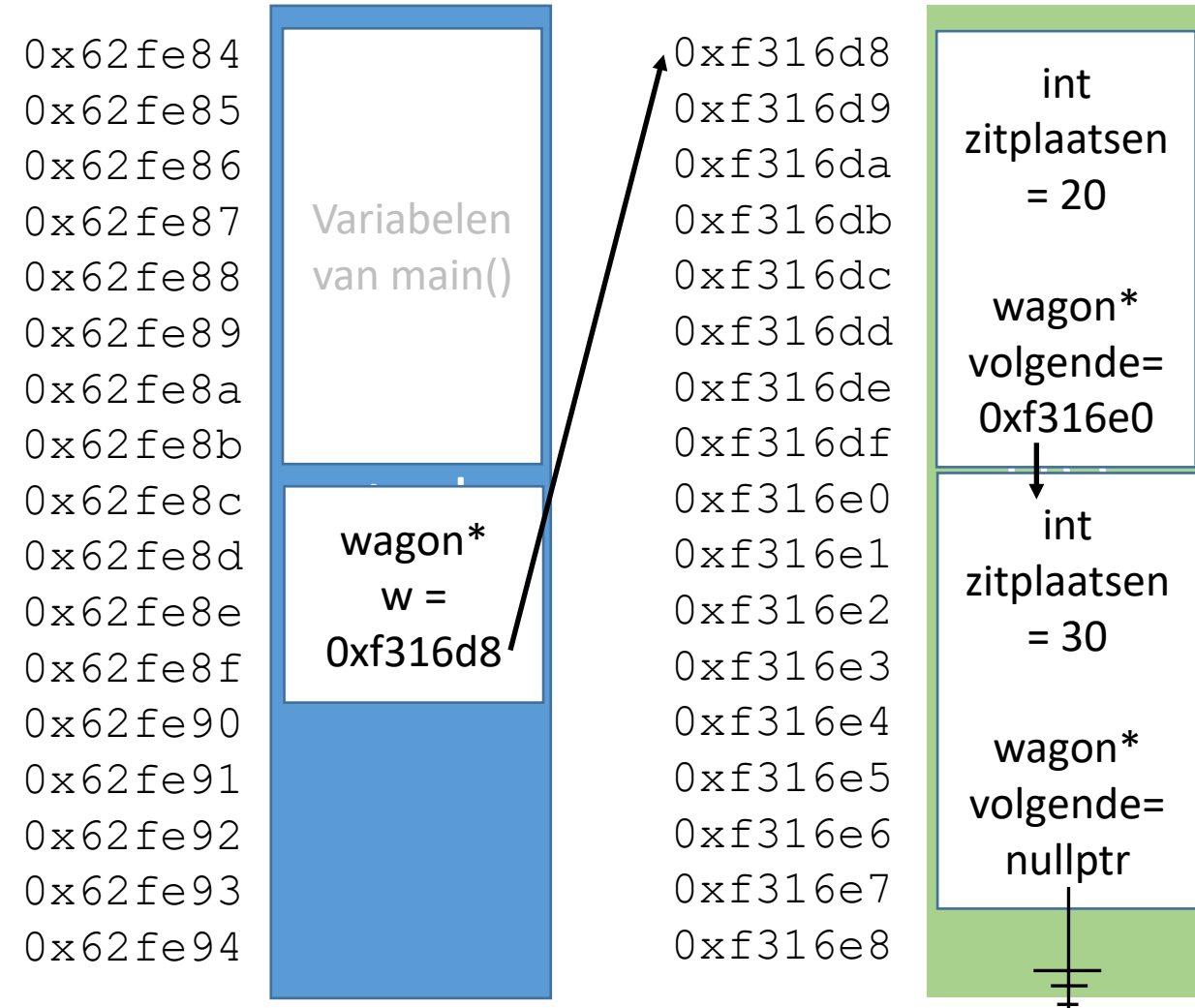
int
zitplaatsen
= 30

wagon*
volgende=
nullptr

Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

Wagon met 20 plaatsen



Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

Wagon met 20 plaatsen

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

wagon*
w =
0xf316d8

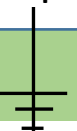
0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0

int
zitplaatsen
= 30

wagon*
volgende=
nullptr



Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

Wagon met 20 plaatsen

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

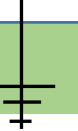
wagon*
w =
0xf316d8

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0
↓
int
zitplaatsen
= 30

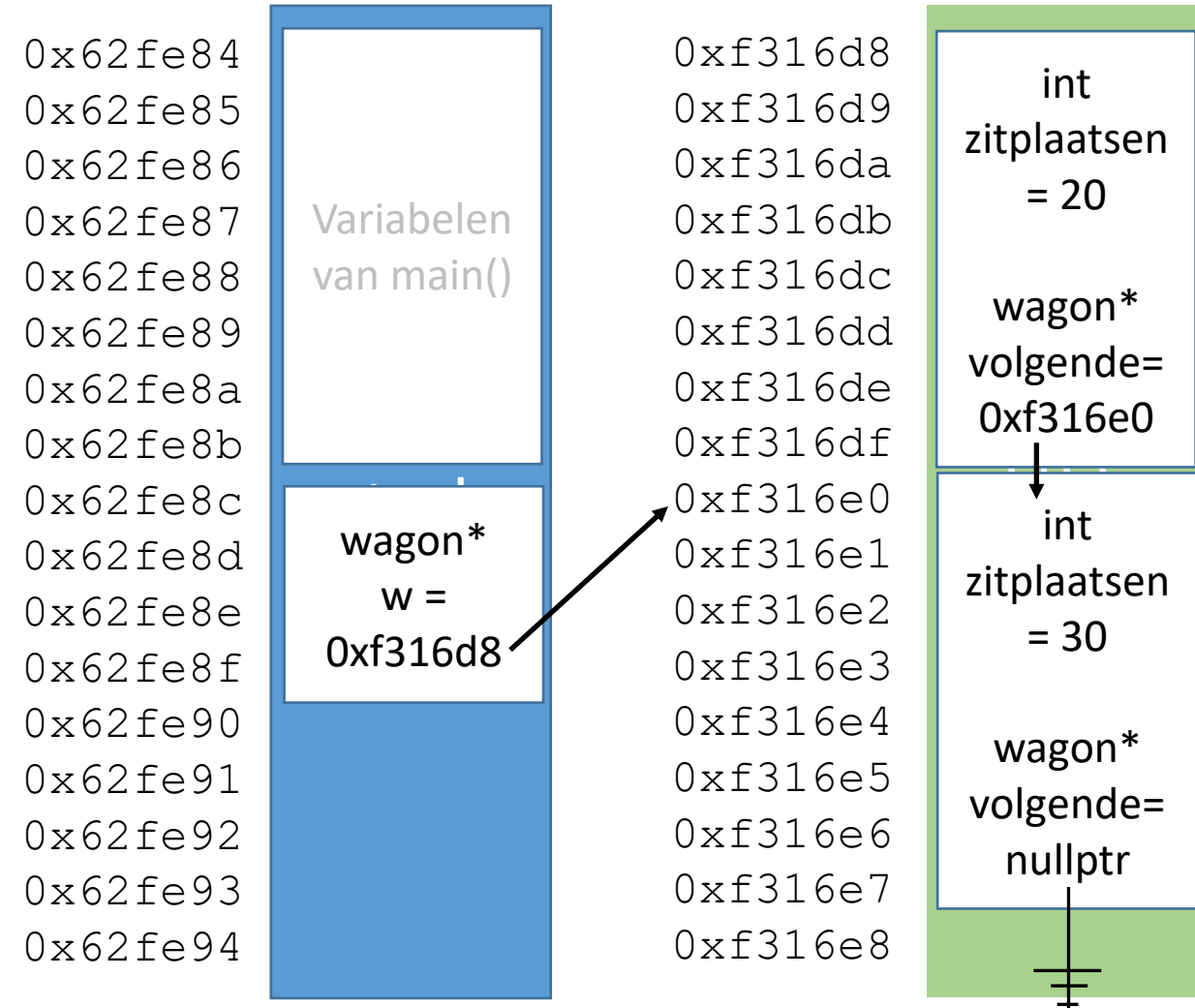
wagon*
volgende=
nullptr



Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

Wagon met 20 plaatsen
Wagon met 30 plaatsen



Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

Wagon met 20 plaatsen
Wagon met 30 plaatsen

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

wagon*
w =
nullptr

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0

int
zitplaatsen
= 30

wagon*
volgende=
nullptr

Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w != nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w = w->volgende;
    }
}
```

Wagon met 20 plaatsen
Wagon met 30 plaatsen

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

wagon*
w =
nullptr

0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0

int
zitplaatsen
= 30

wagon*
volgende=
nullptr

Rekursieve structs

```
void printTrein(const wagon* w) {
    while (w!=nullptr) {
        cout << "Wagon met "
              << w->zitplaatsen
              << " plaatsen" << endl;
        w=w->volgende;
    }
}
```

}

Wagon met 20 plaatsen
Wagon met 30 plaatsen

0x62fe84
0x62fe85
0x62fe86
0x62fe87
0x62fe88
0x62fe89
0x62fe8a
0x62fe8b
0x62fe8c
0x62fe8d
0x62fe8e
0x62fe8f
0x62fe90
0x62fe91
0x62fe92
0x62fe93
0x62fe94

Variabelen
van main()

wagon*
w =
nullptr



0xf316d8
0xf316d9
0xf316da
0xf316db
0xf316dc
0xf316dd
0xf316de
0xf316df
0xf316e0
0xf316e1
0xf316e2
0xf316e3
0xf316e4
0xf316e5
0xf316e6
0xf316e7
0xf316e8

int
zitplaatsen
= 20

wagon*
volgende=
0xf316e0




int
zitplaatsen
= 30

wagon*
volgende=
nullptr



Rekursieve structs

- We kunnen de trein uitbreiden (vooraan):



```
wagon* addWagon(wagon* eerste, int z)
{
    wagon* w=new wagon;
    w->zitplaatsen=z;
    w->volgende=eerste;

    return w;
}
```

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```

Rekursieve structs

- We kunnen de trein uitbreiden (achteraan):

```
void addWagonAchteraan(wagon* trein, int z) {  
    // Zoek de laatste wagon  
    while (trein->volgende != nullptr) {  
        trein=trein->volgende;  
    }  
  
    // Voeg de wagon toe  
    trein->volgende=new wagon;  
    trein->volgende->volgende=nullptr;  
    trein->volgende->zitplaatsen=z;  
}
```

```
struct wagon {  
    int zitplaatsen;  
    wagon* volgende;  
};
```

Rekursieve structs

- Let op! Wat als we oproepen met een lege trein?

```
void addWagonAchteraan(wagon* trein, int z) {  
    // Zoek de laatste wagon  
    while (trein->volgende != nullptr) {  
        trein=trein->volgende;  
    }  
  
    // Voeg de wagon toe  
    trein->volgende=new wagon;  
    trein->volgende->volgende=nullptr;  
    trein->volgende->zitplaatsen=z;  
}
```

```
struct wagon {  
    int zitplaatsen;  
    wagon* volgende;  
};
```

Recursieve structs

- Let op! Wat als we oproepen met een `nullptr`?

ERROR !!!

nullptr

```

struct wagon {
    int zitplaatsen;
    wagon* volgende;
};

void addWagonAan(trein* trein, int z) {
    // Zoek de laatste wagon
    while (trein->volgende != nullptr) {
        trein=trein->volgende;
    }

    // Voeg de wagon toe
    trein->volgende=new wagon;
    trein->volgende->volgende=nullptr;
    trein->volgende->zitplaatsen=z;
}
  
```


Rekursieve structs

- Let op! Wat als we oproepen met een lege trein?

```
struct wagon {  
    int zitplaatsen;  
    wagon* volgende;  
};
```

```
void addWagonAchteraan(wagon* &trein, int z) {  
    if (trein==nullptr) {  
        trein=new wagon;  
        trein->volgende=nullptr;  
        trein->zitplaatsen=z;  
    }
```

```
    wagon* laatste=trein;
```

```
    // Zoek de laatste wagon
```

```
    while (laatste->volgende != nullptr) {  
        laatste=laatste->volgende;  
    }
```

“Trein” aanmaken

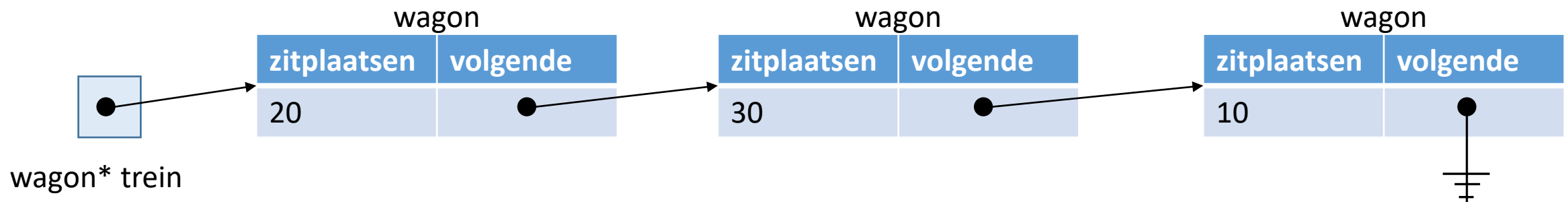
- Gegeven een vector met aantal zitplaatsen, maak een trein aan met exact die aantallen zitplaatsen:

```
wagon* maakTrein(vector<int> v) {  
    wagon* result = new wagon;  
    wagon* current = result;  
    result->zitplaatsen=v[0];  
    for (int i=1;i<v.size();i++) {  
        current->volgende=new wagon;  
        current=current->volgende;  
        current->zitplaatsen=v[i];  
    }  
    current->volgende=nullptr;  
    return result;  
}
```

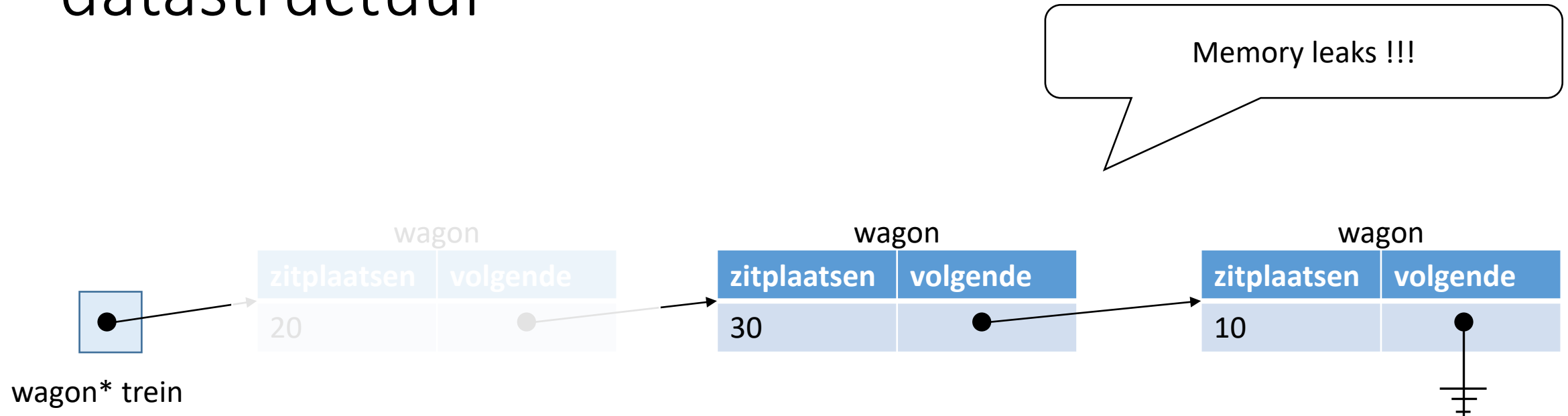
Delete

- Schrijf een functie die een trein, gegeven door middel van een pointer naar de eerste wagon, volledig terug verwijderd. Dat is, alle objecten die met new werden aangemaakt in `maakTrein(vector<int> v)` moeten terug verwijderd worden door middel van `delete`. Schrijf een functie `void ontmantel_trein(wagon* trein)` die dit doet.

Conceptuele voorstelling van onze datastructuur

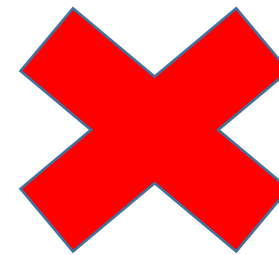


Conceptuele voorstelling van onze datastructuur



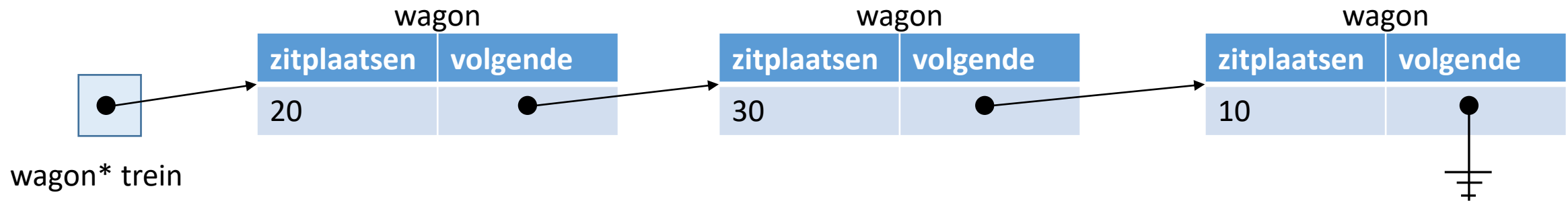
```

void deleteTrein(wagon* eerste) {
    delete eerste;
}
    
```



Trein verwijderen

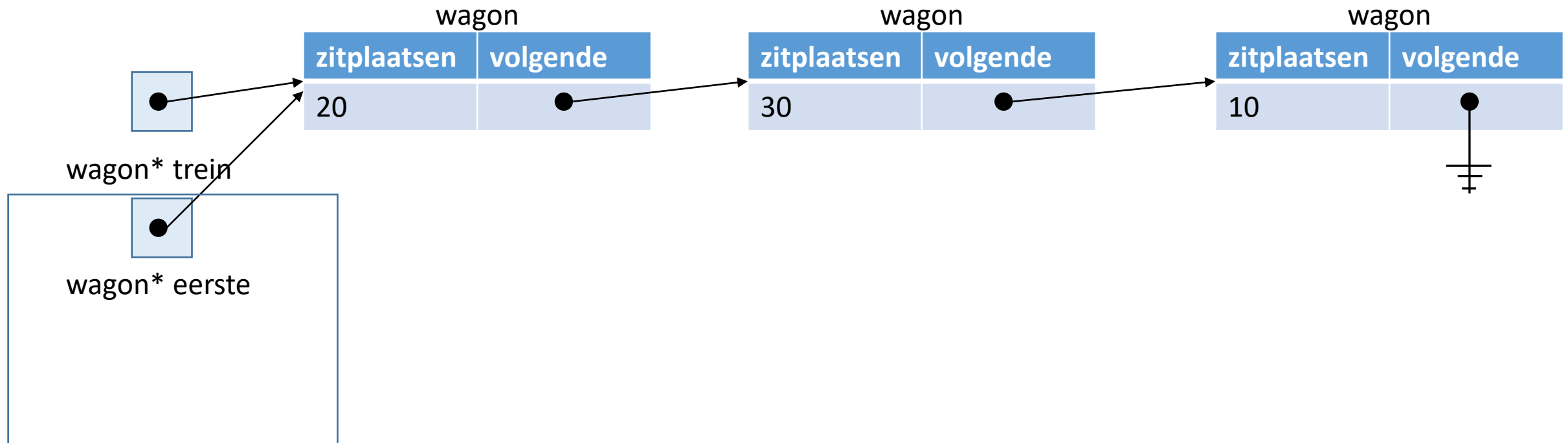
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

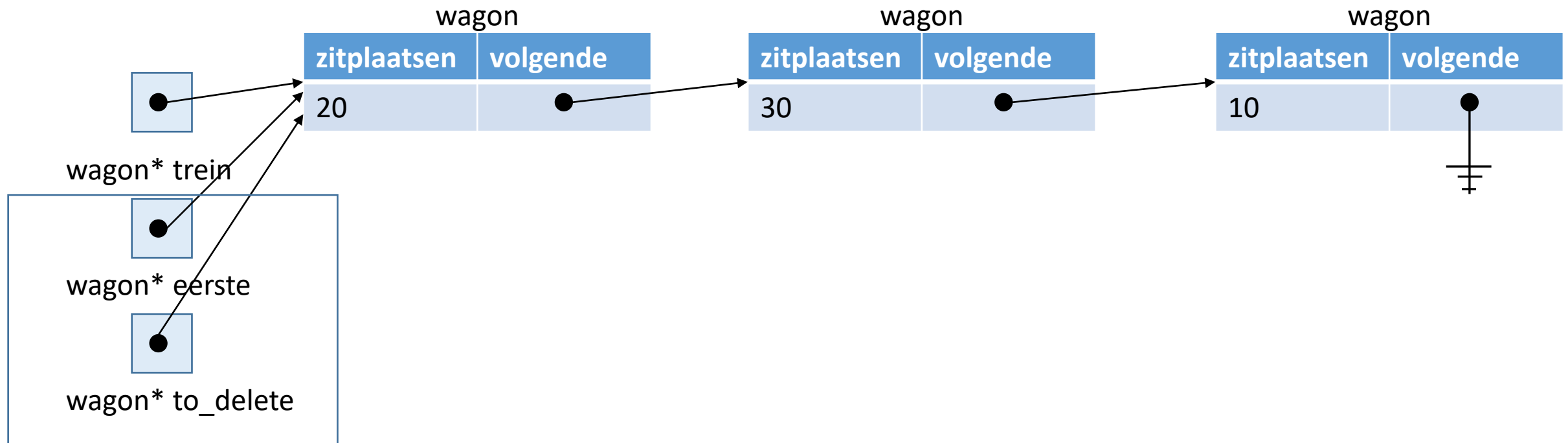
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

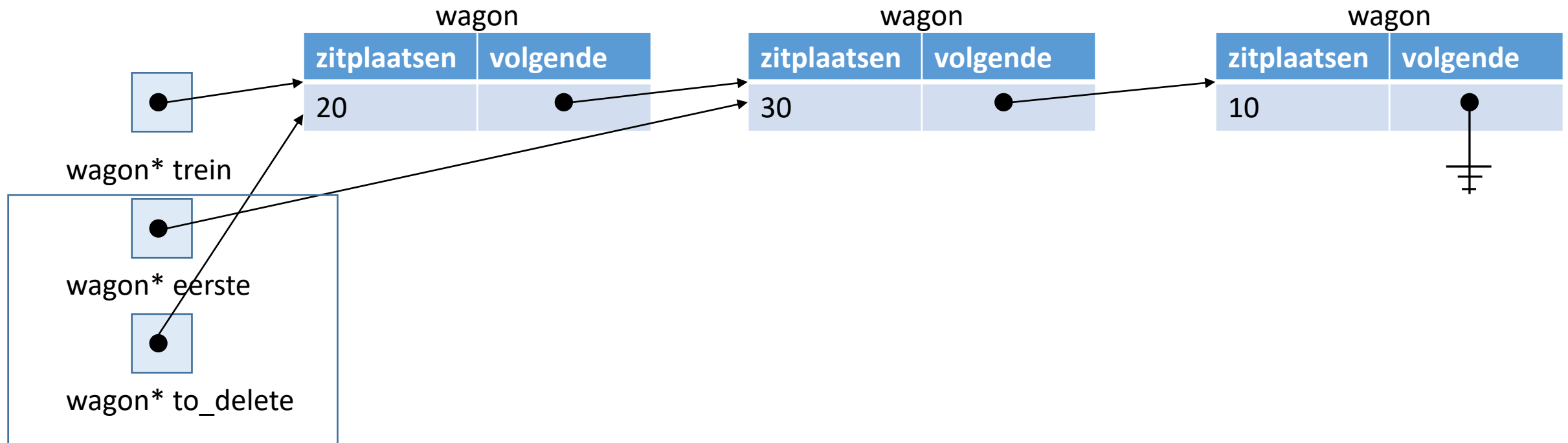
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```


Trein verwijderen

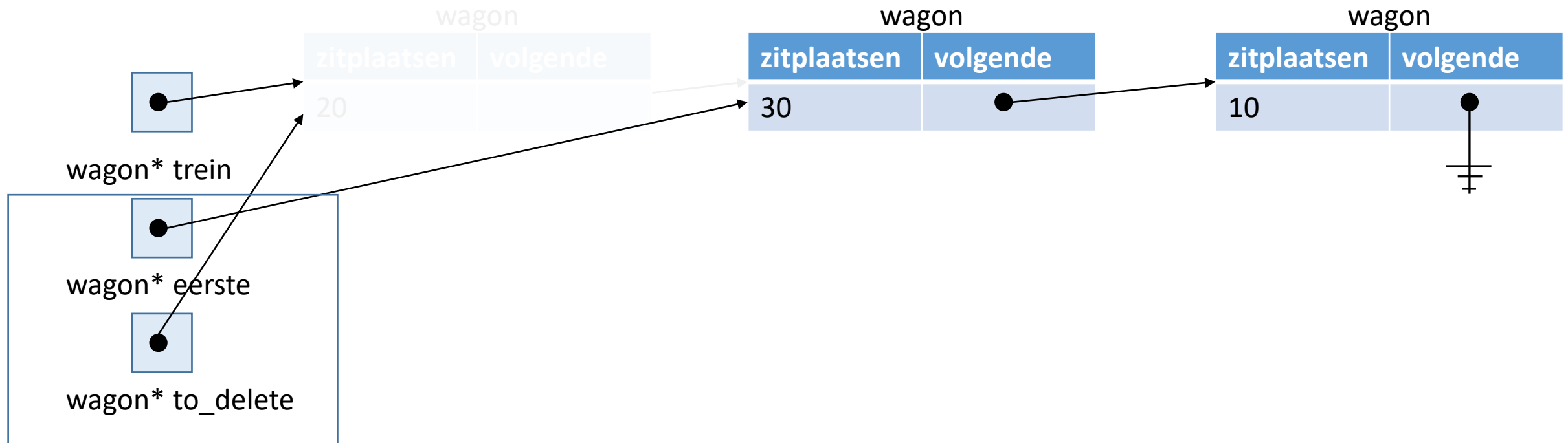
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

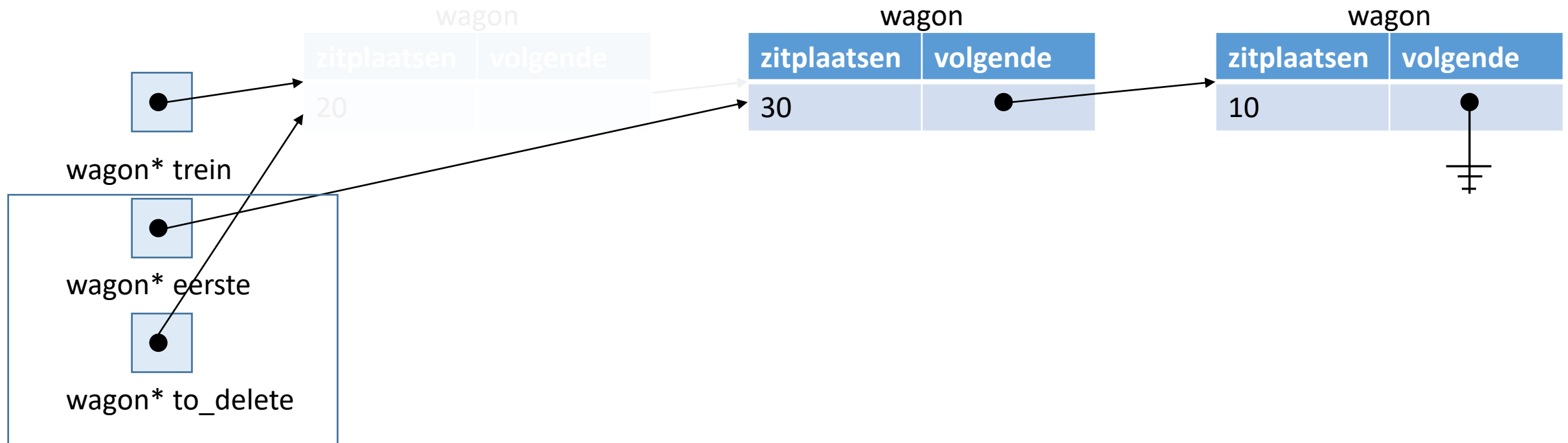
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

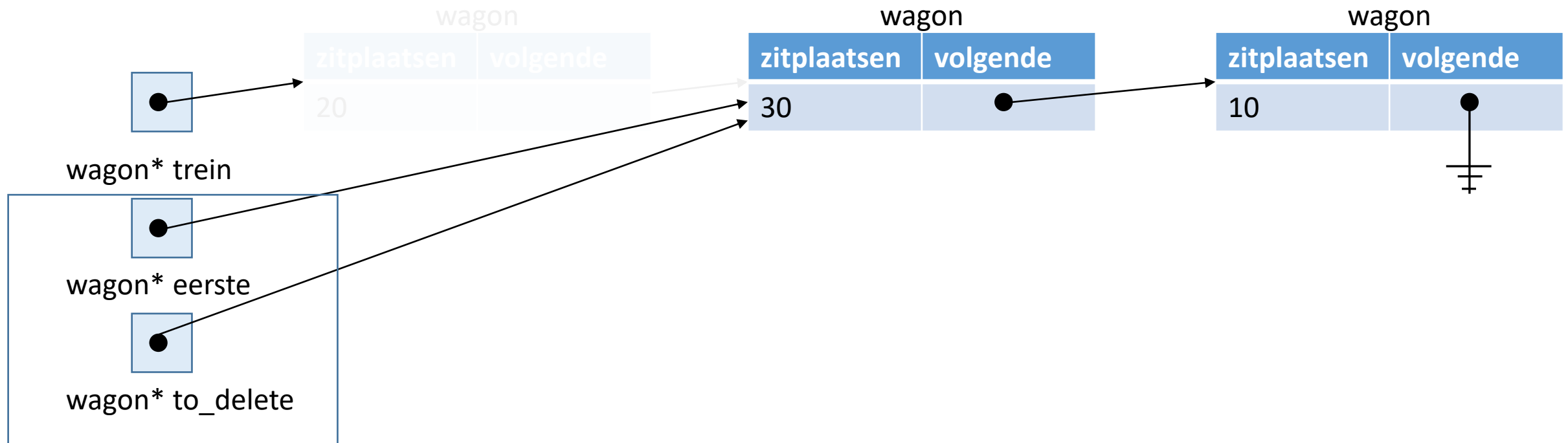
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

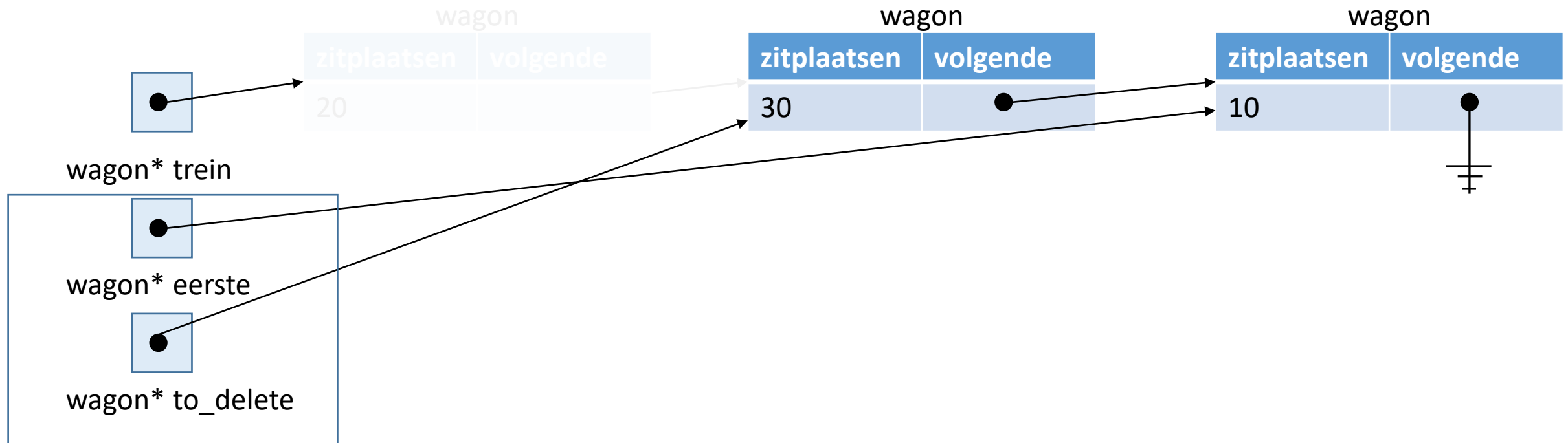
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

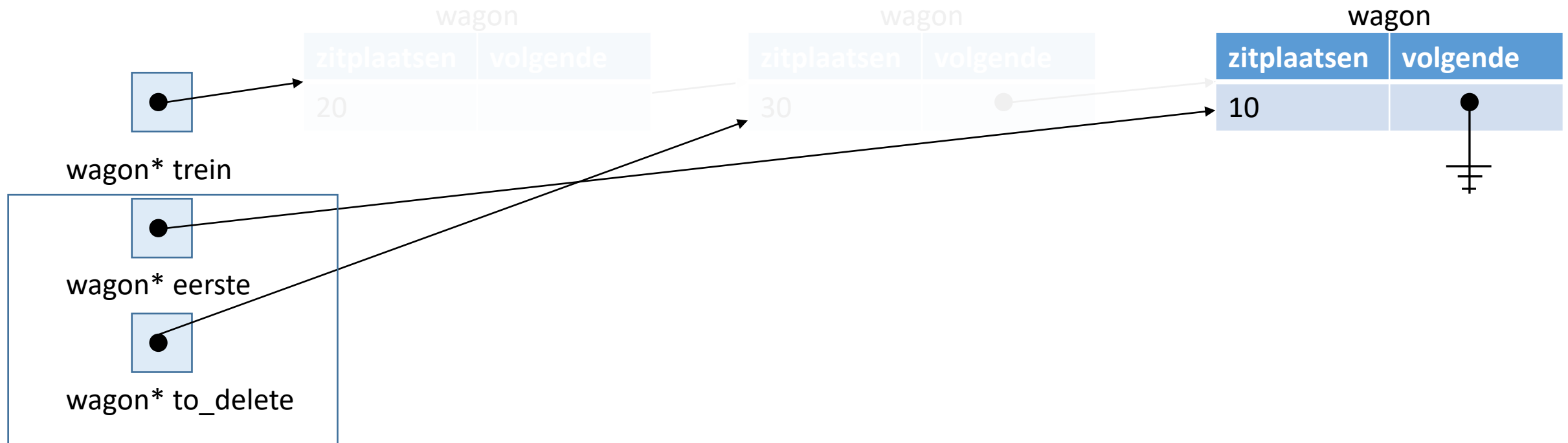
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

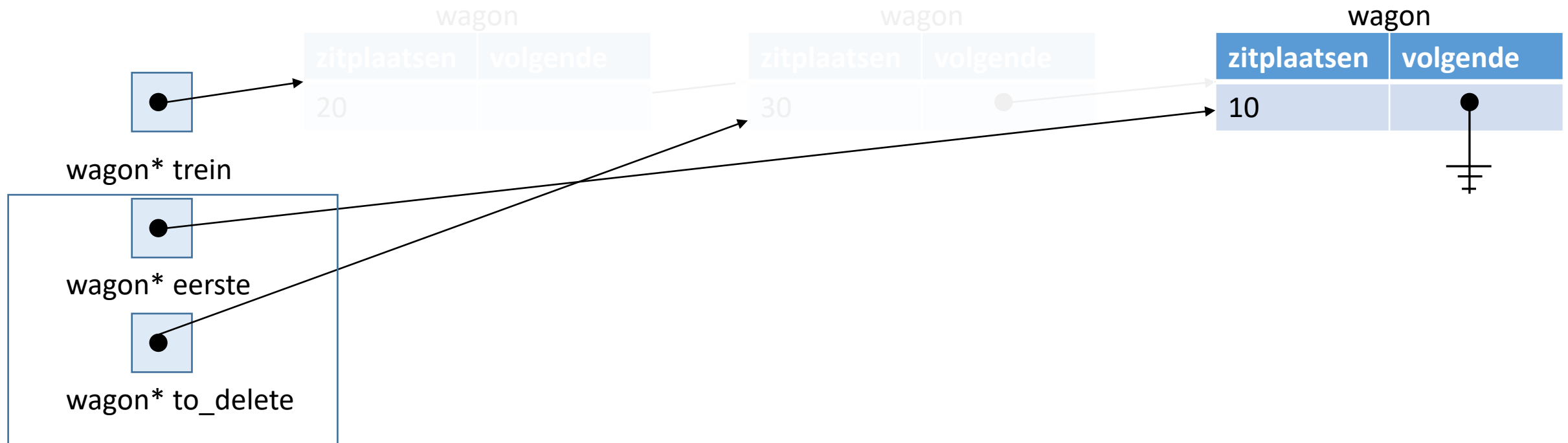
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

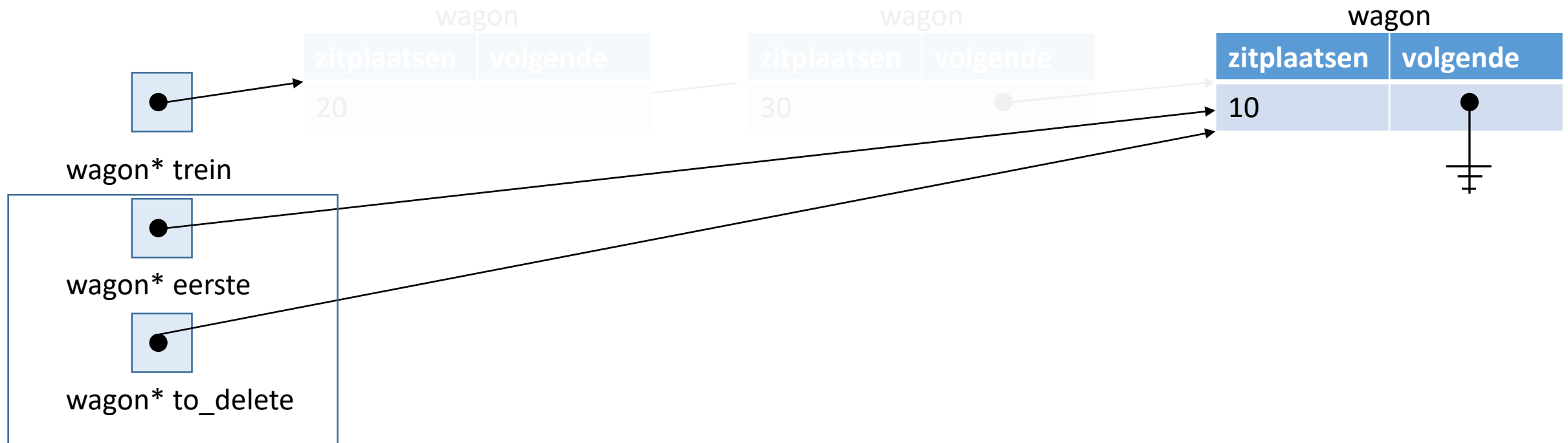
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste!=nullptr) {
        wagon* to_delete=eerste;
        eerste=eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

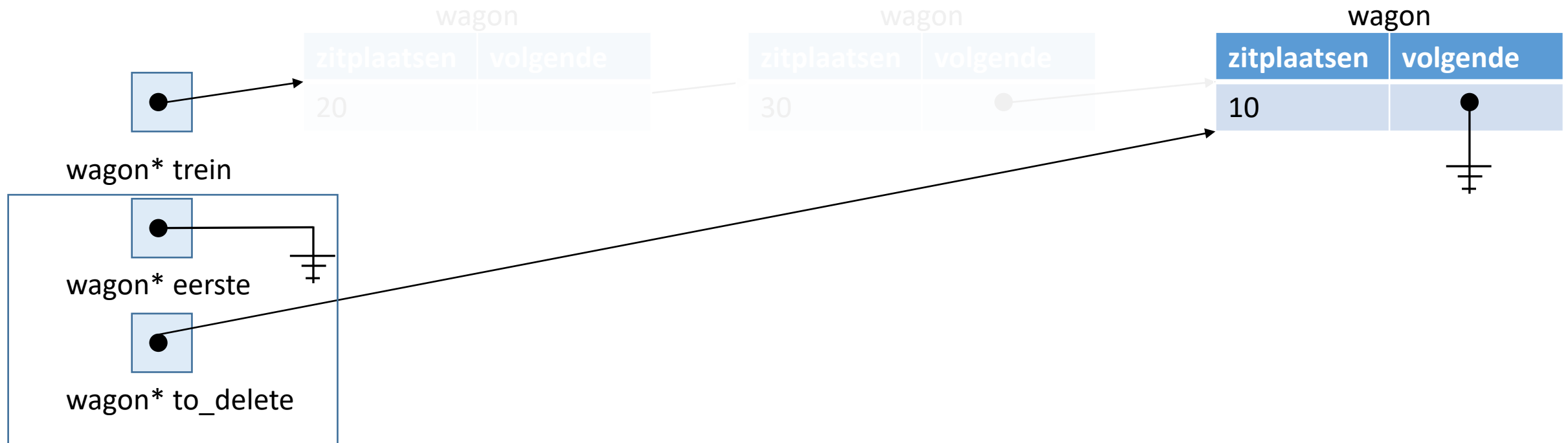
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```


Trein verwijderen

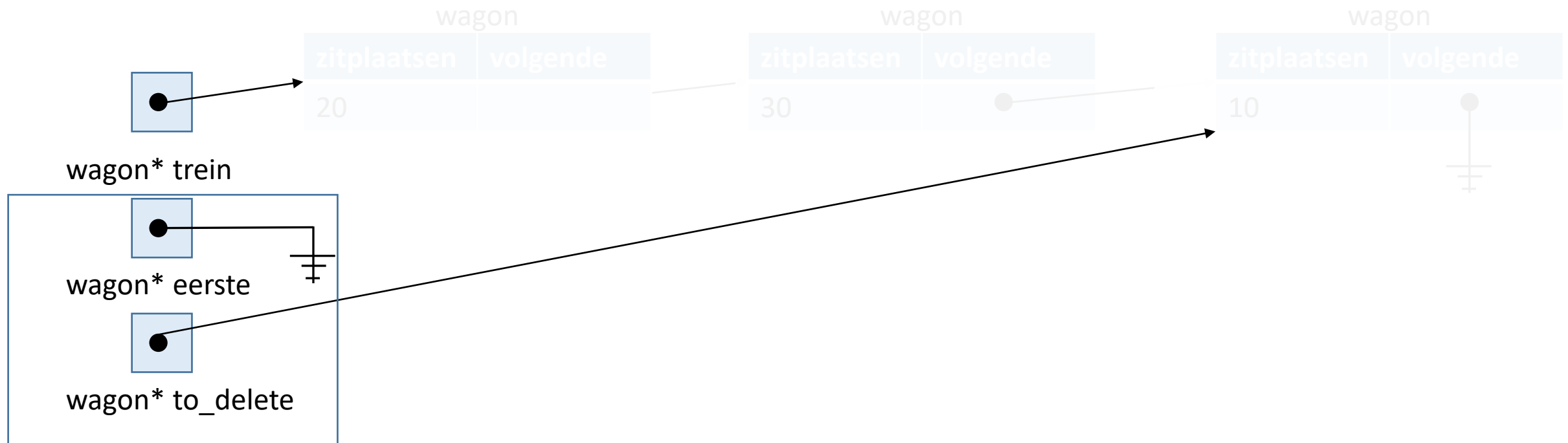
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

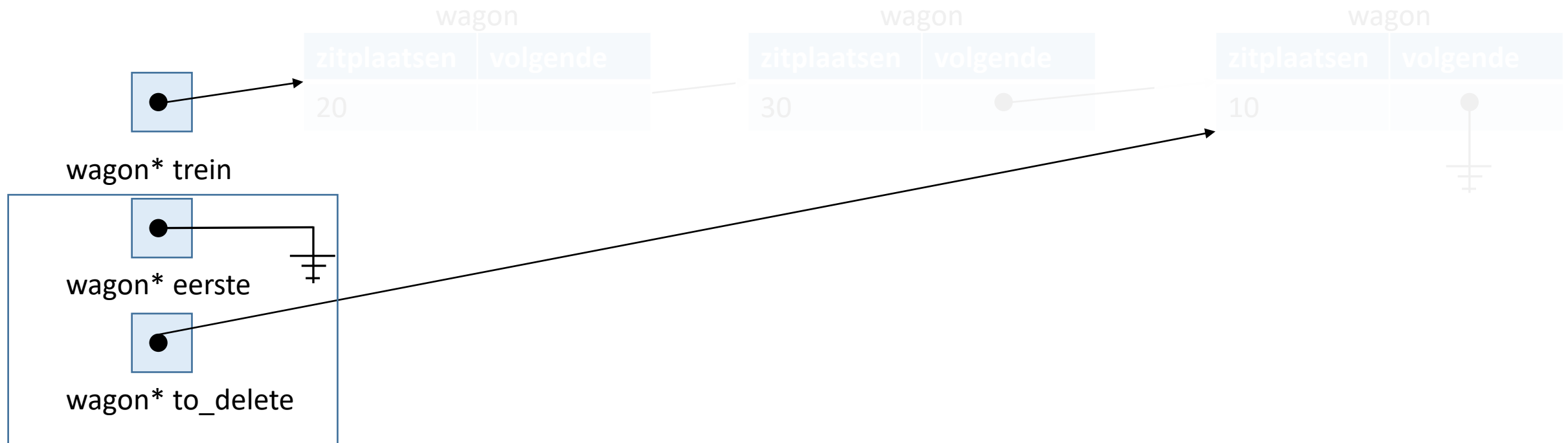
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

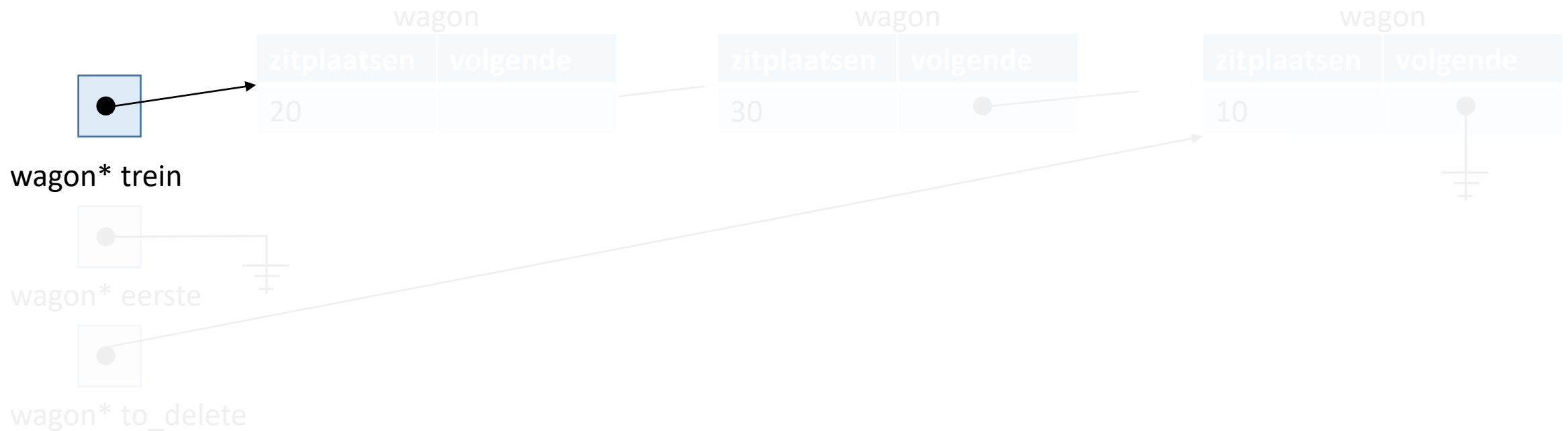
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen

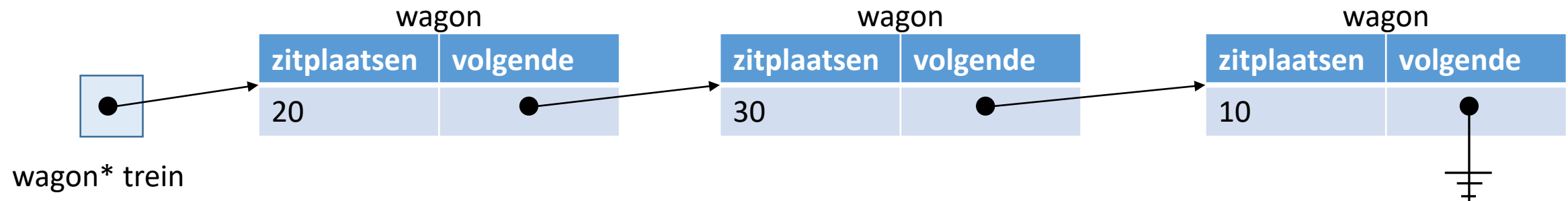
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void delete_trein(wagon* eerste) {
    while (eerste != nullptr) {
        wagon* to_delete = eerste;
        eerste = eerste->volgende;
        delete to_delete;
    }
}
```

Trein verwijderen – v2

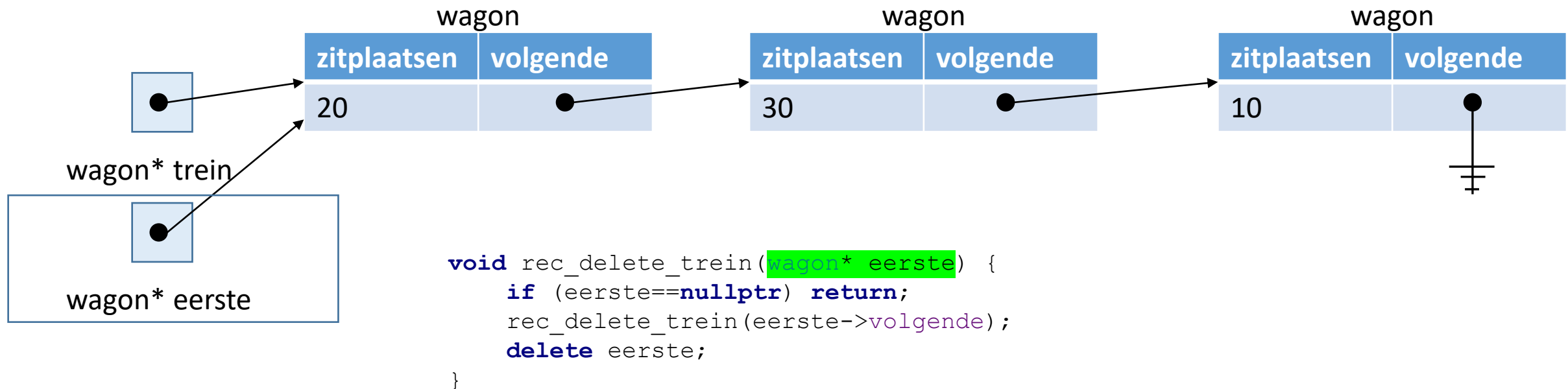
```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void rec_delete_trein(wagon* eerste) {
    if (eerste==nullptr) return;
    rec_delete_trein(eerste->volgende);
    delete eerste;
}
```

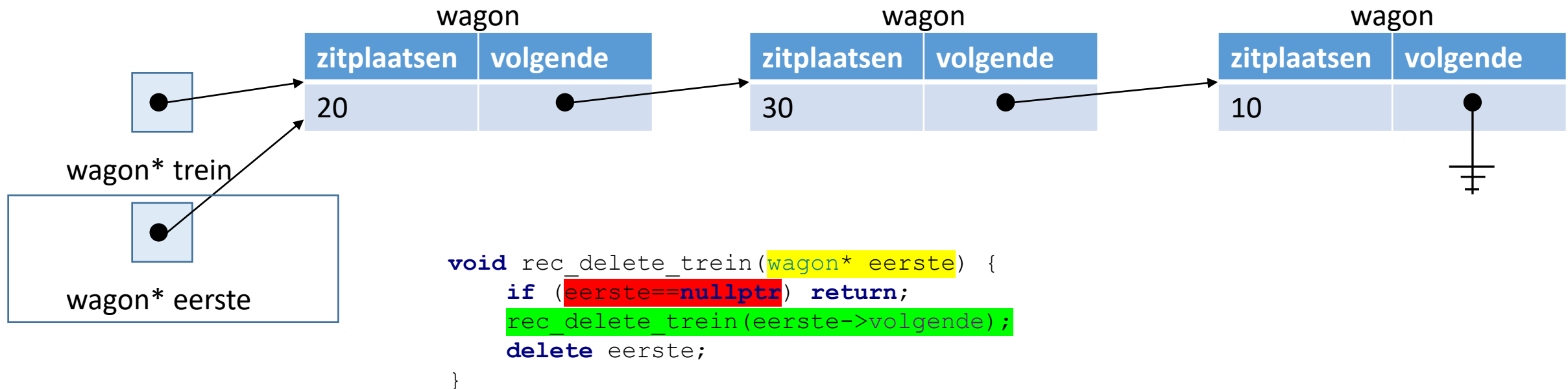
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



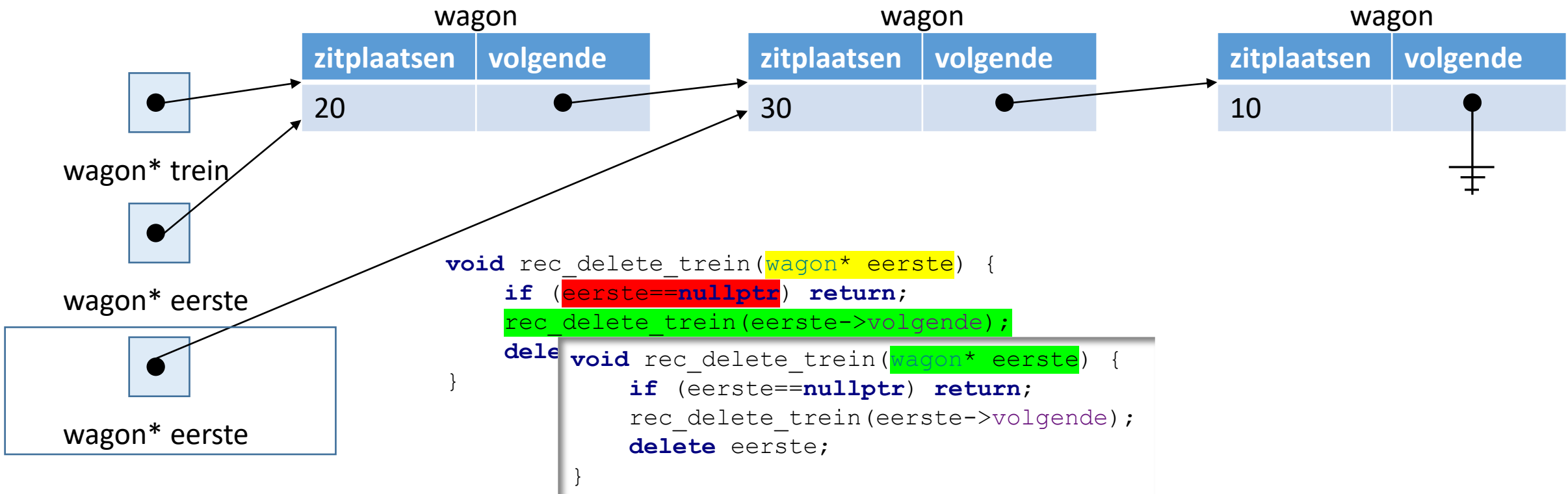
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



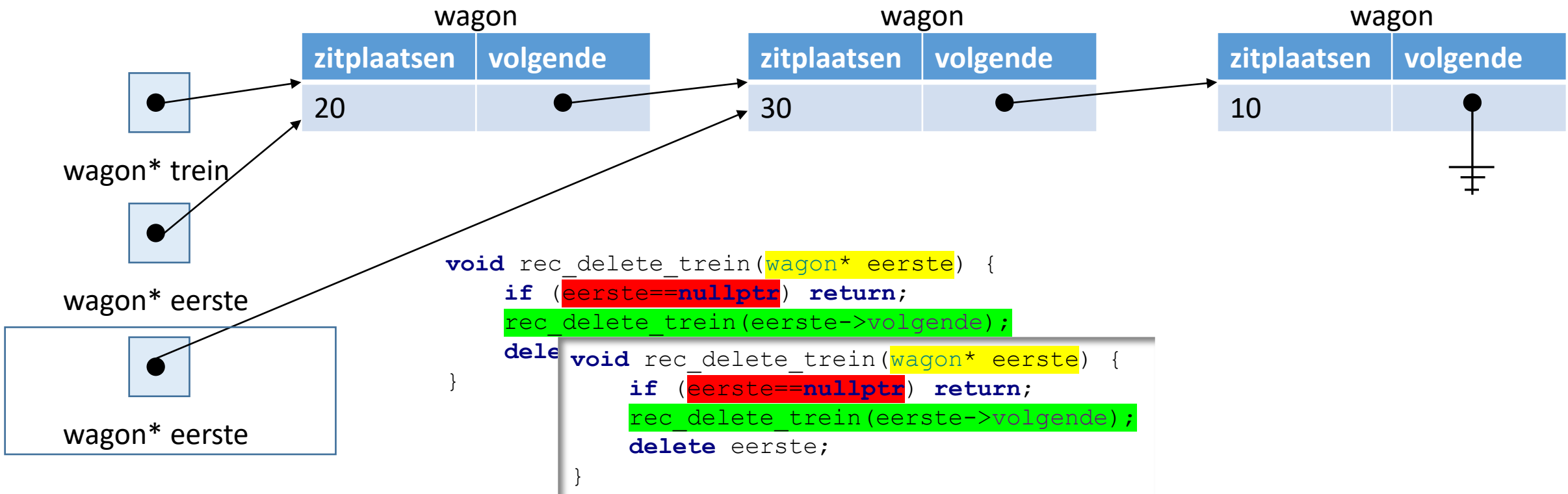
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



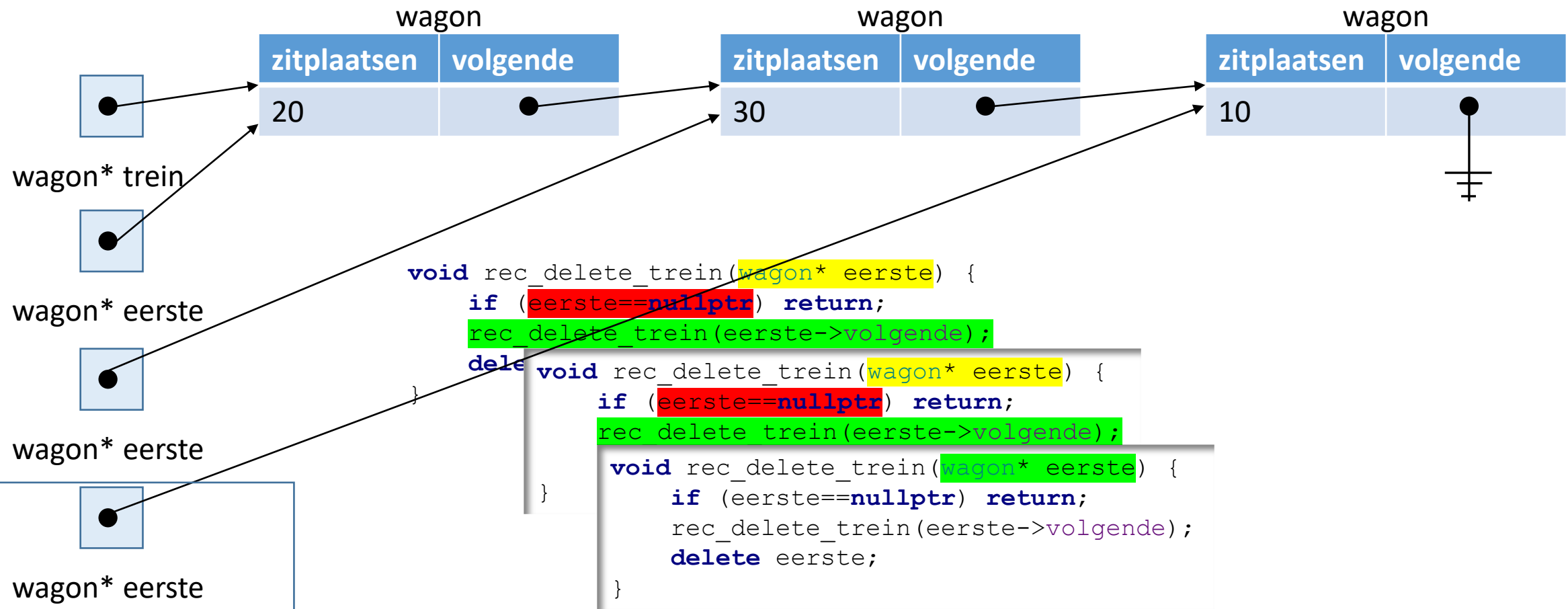
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



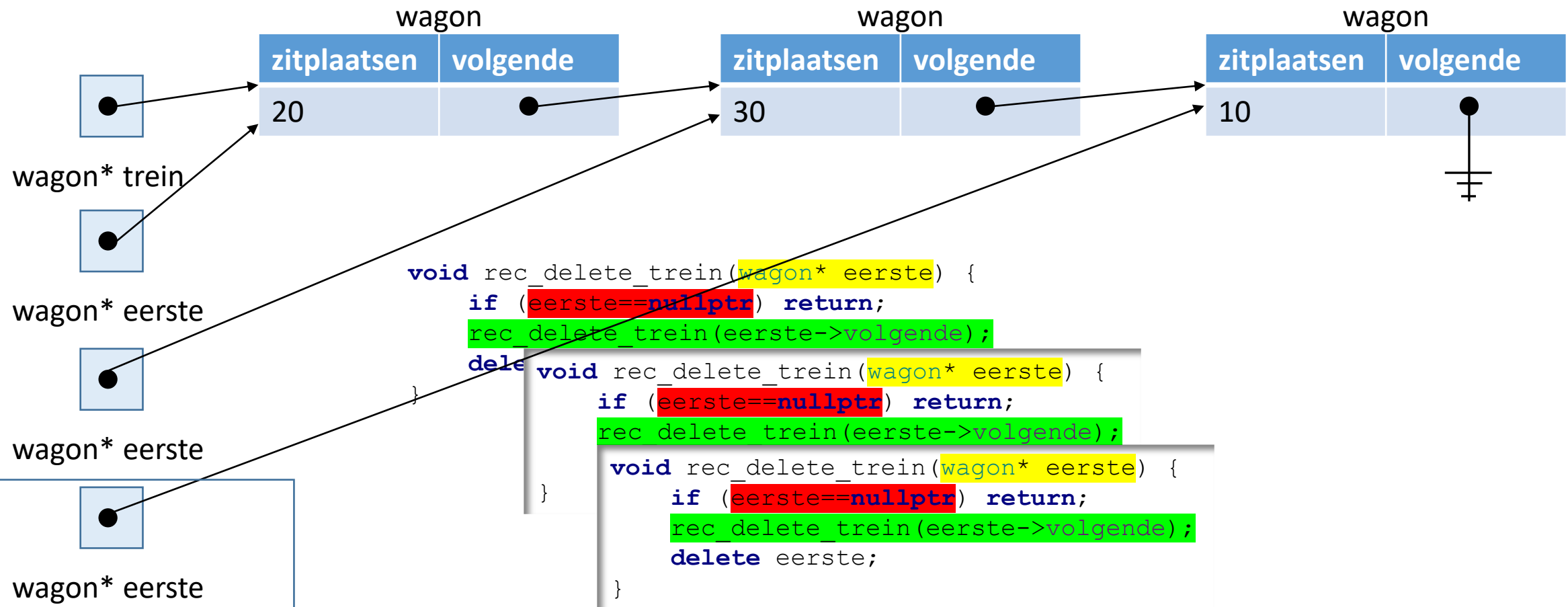
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



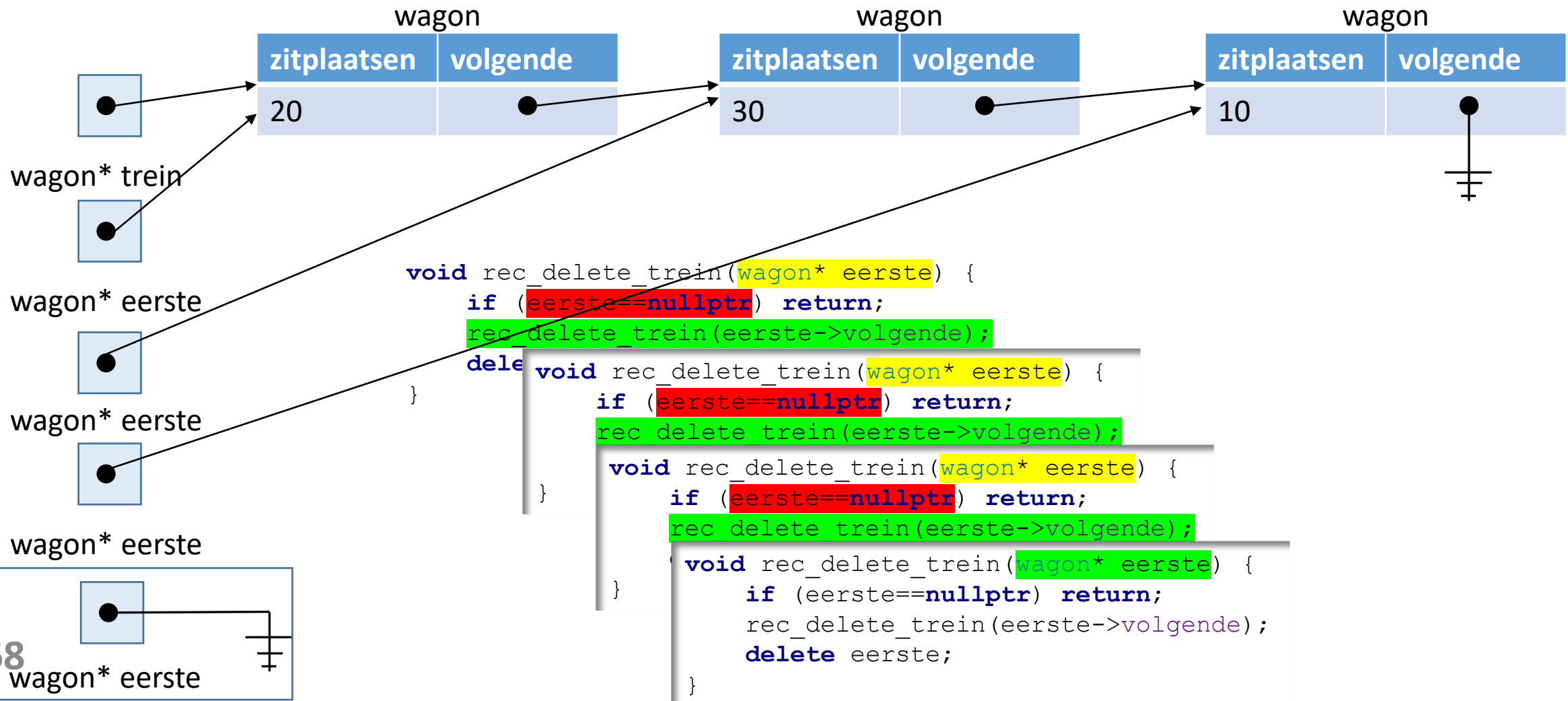
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



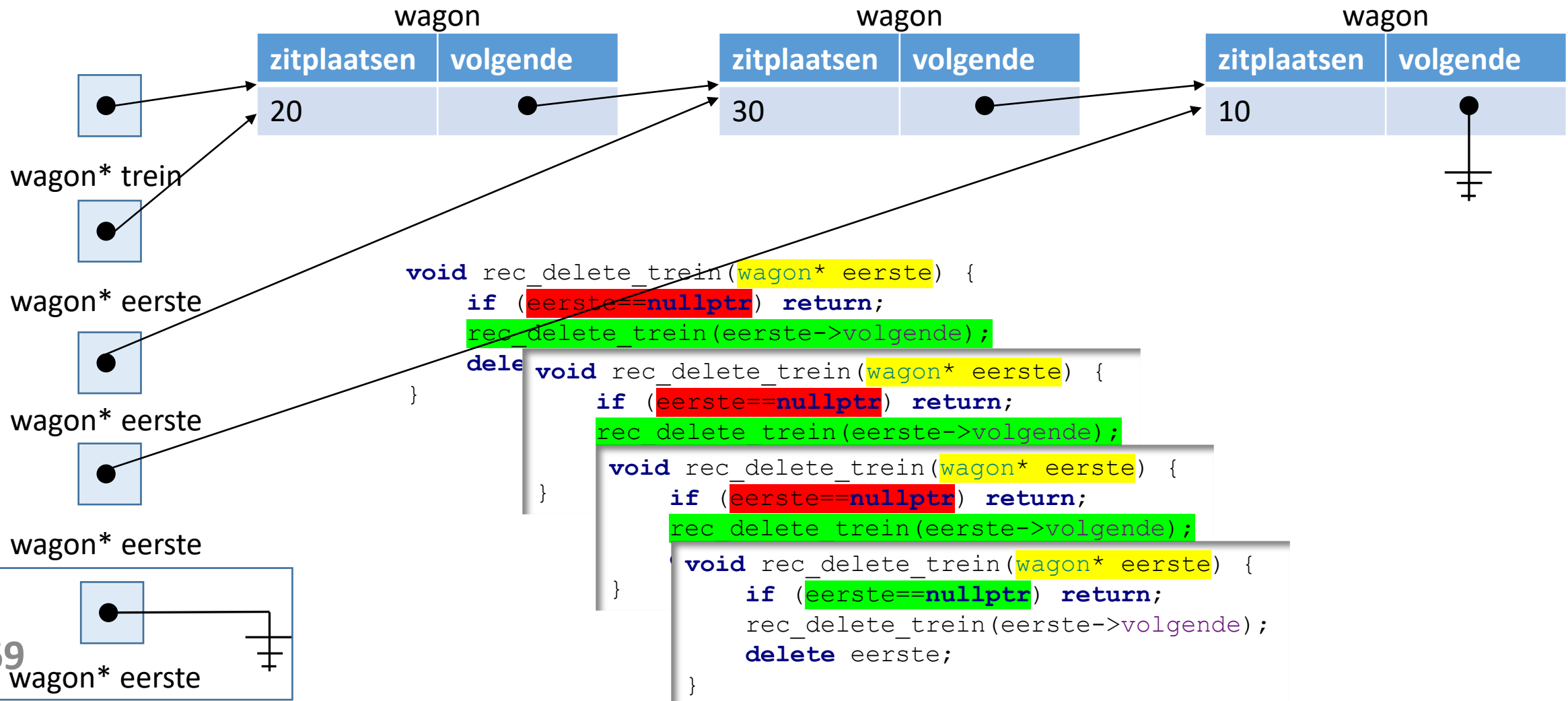
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



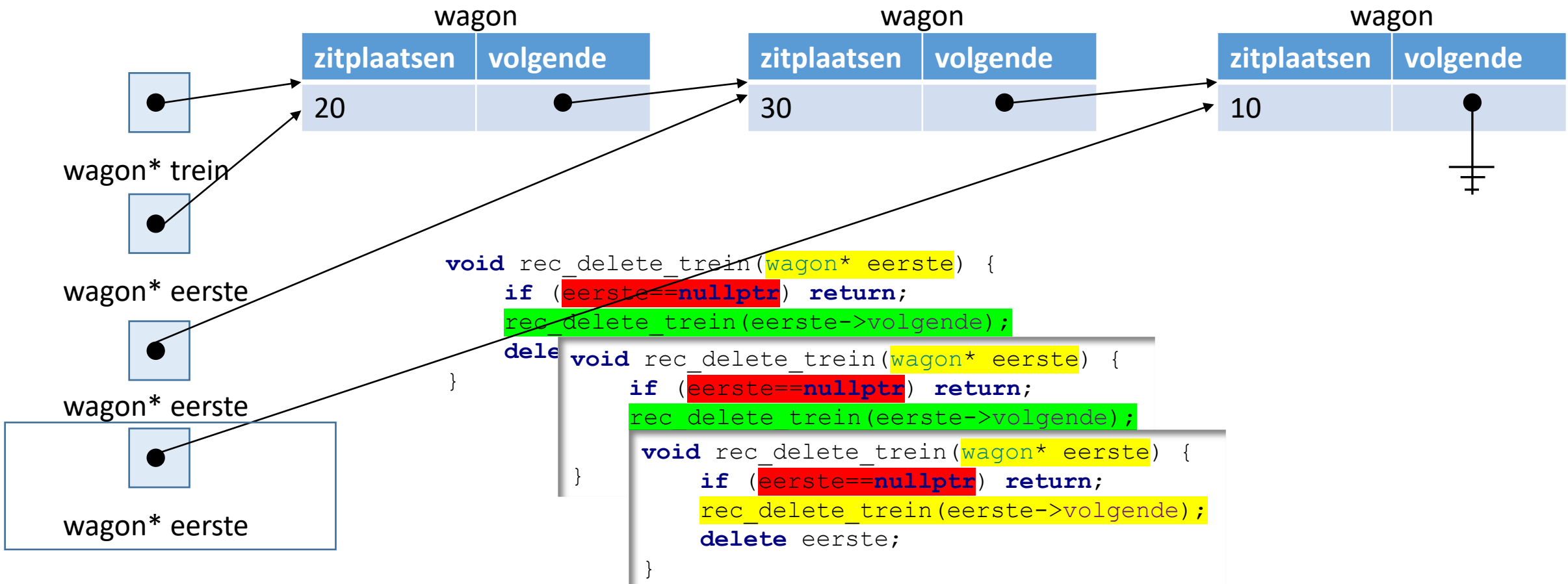
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



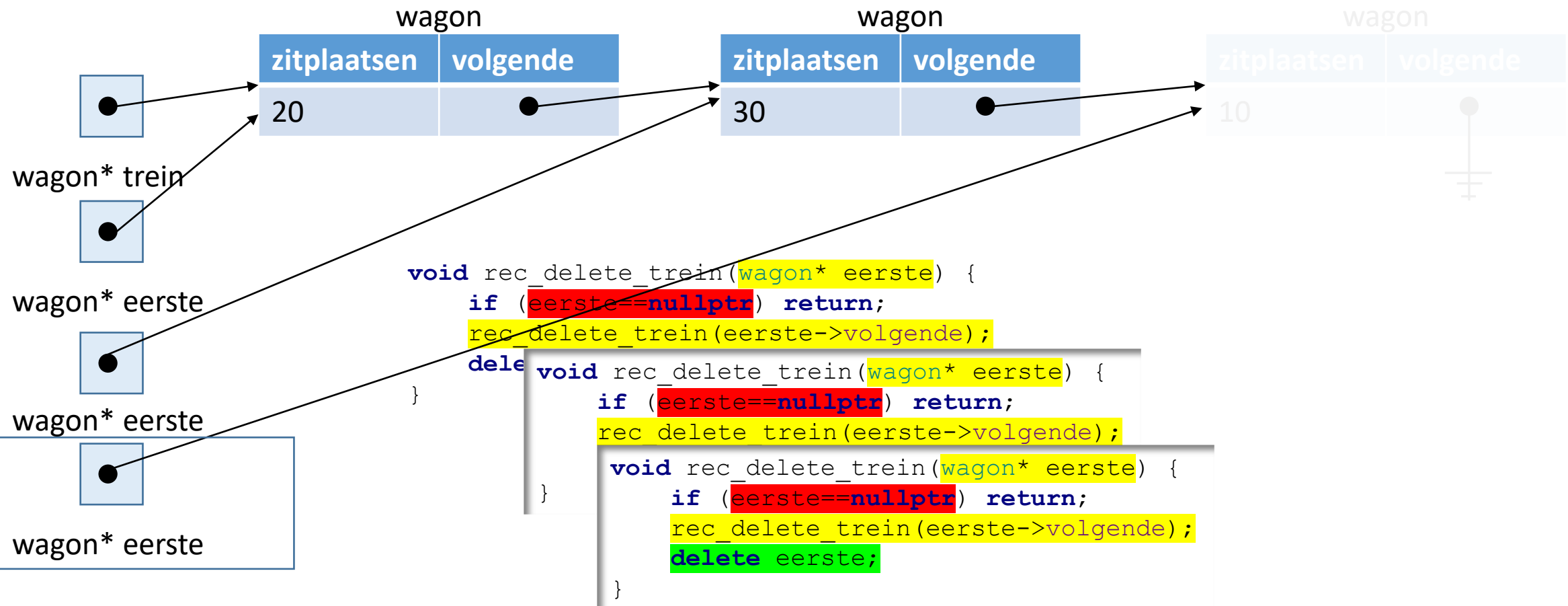
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



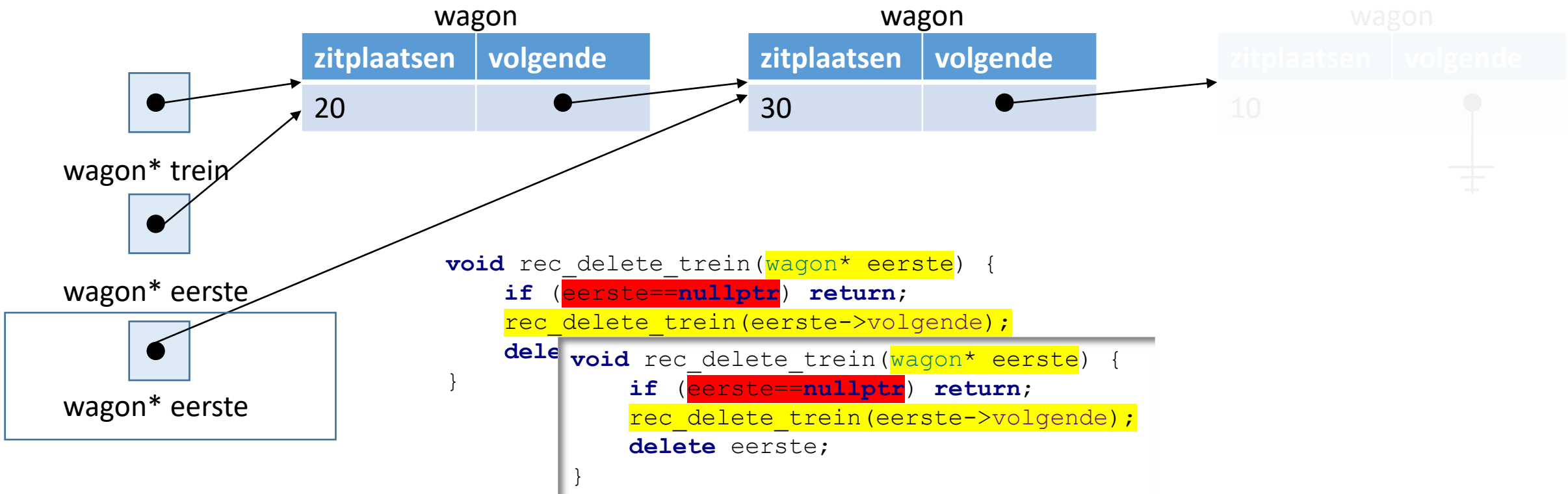
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



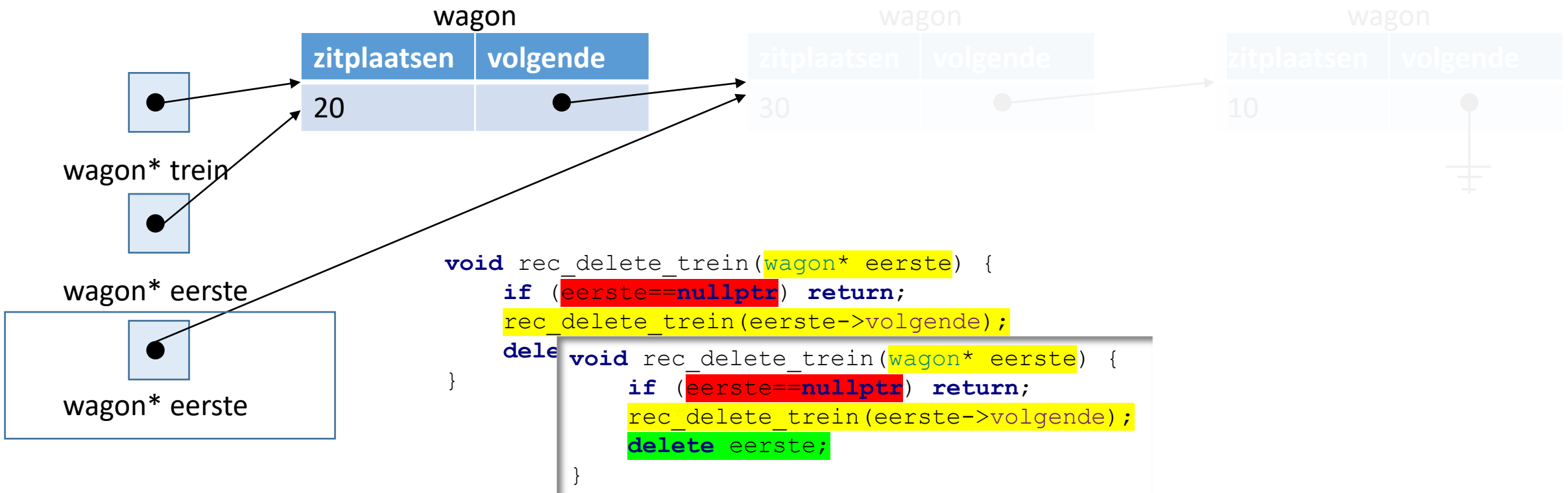
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



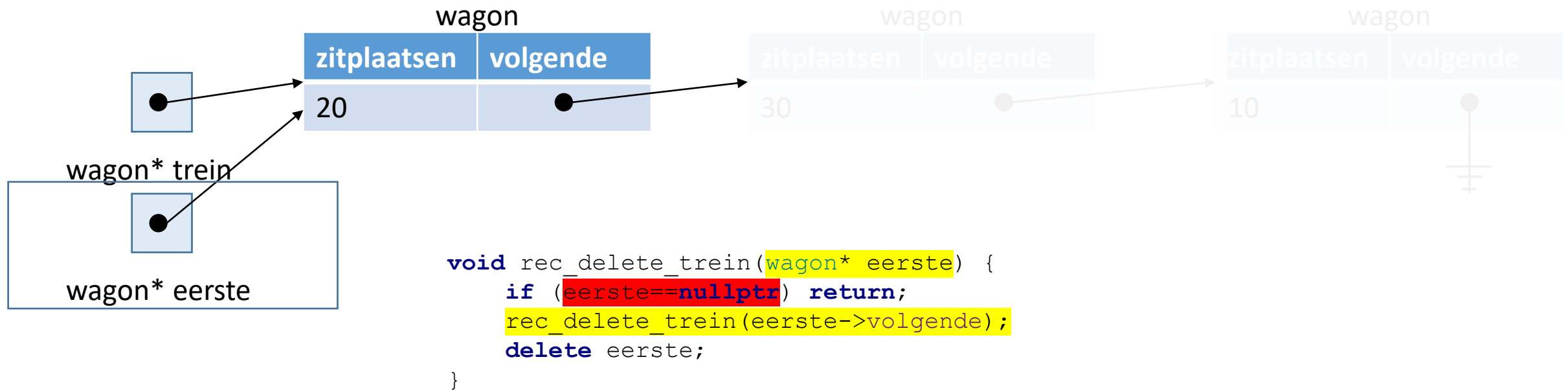
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



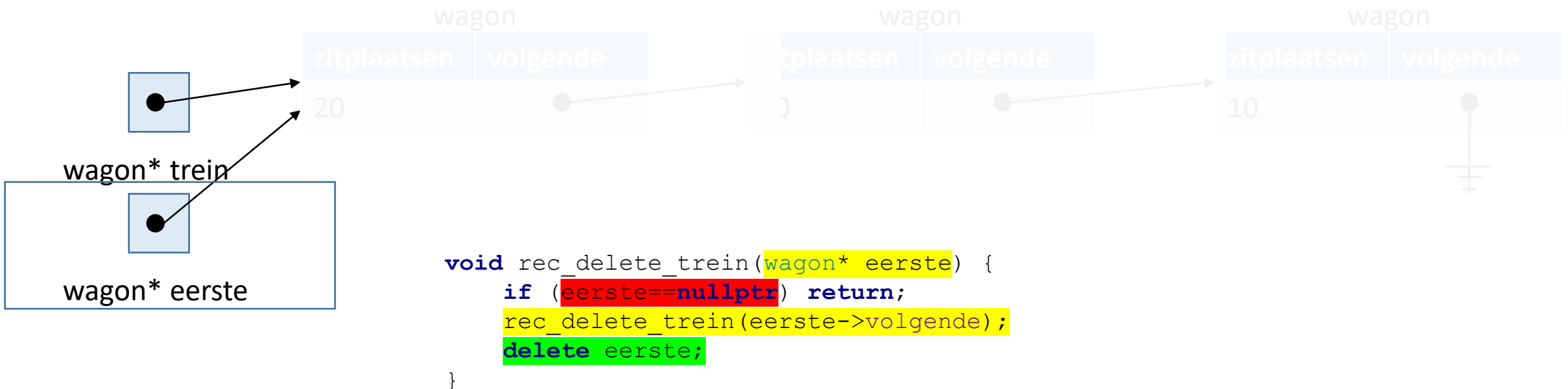
Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



Trein verwijderen – v2

```
struct wagon {
    int zitplaatsen;
    wagon* volgende;
};
```



```
void rec_delete_trein(wagon* eerste) {
    if (eerste==nullptr) return;
    rec_delete_trein(eerste->volgende);
    delete eerste;
}
```

Stack implementatie

- Interface:

```
typedef StackNode* Stack;  
  
Stack newIntStack();  
void push(Stack &S, int c);  
bool empty(Stack S);  
int pop(Stack &S);
```

- Om het programma overzichtelijk te houden kunnen we een *forward declaration* van onze functies doen, en pas na de functie main() de *definitie* van de functie geven.

Forward declaration

```
struct StackNode {
    StackNode* next=nullptr;
    int content;
};

typedef StackNode* Stack;

Stack newIntStack();
void push(Stack &S, int c);
bool empty(Stack S);
int pop(Stack &S);

int main() {
    Stack S=newIntStack();
    for (int i=0;i<10;i++) {
        push(S,i);
    }

    for (int i=0;i<12;i++) {
        cout << pop(S) << " ";
    }
    cout << endl;

    return 0;
}
```

```
Stack newIntStack() {
    ...
}

void push(Stack &S, int c) {
    ...
}

bool empty(Stack S) {
    ...
}

int pop(Stack &S) {
    ...
}
```

Graaf implementatie

- Graaf wordt bewaard in een bestand van de volgende vorm:

#nodes

#buren node 0	buur 0	buur 1	buur 2	...
---------------	--------	--------	--------	-----

#buren node 1	buur 0	buur 1	buur 2	...
---------------	--------	--------	--------	-----

#buren node 2	buur 0	buur 1	buur 2	...
---------------	--------	--------	--------	-----

Graaf implementatie

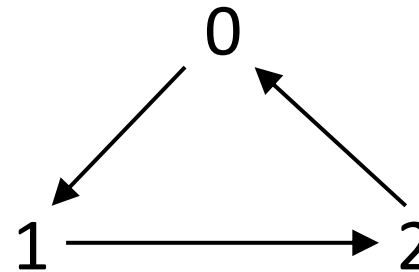
- Graaf wordt bewaard in een bestand van de volgende vorm:

3

1 1

1 2

1 0



- We willen de graaf van disk laden en zo compact mogelijk opslaan in het geheugen

Graaf implementatie

```
struct Node {  
    int degree=0;  
    int* neighbors=nullptr;  
};
```

```
struct Graph {  
    int n=0;  
    Node* nodes=nullptr;  
};
```

```
Graph readGraphFromDisk(string name);
```

Depth-first search in de graaf

- Vind een doelnode startende van een gegeven node

```
bool reachable(Graph G, int sID, int tID);
```

- We maken gebruik van depth-first search

