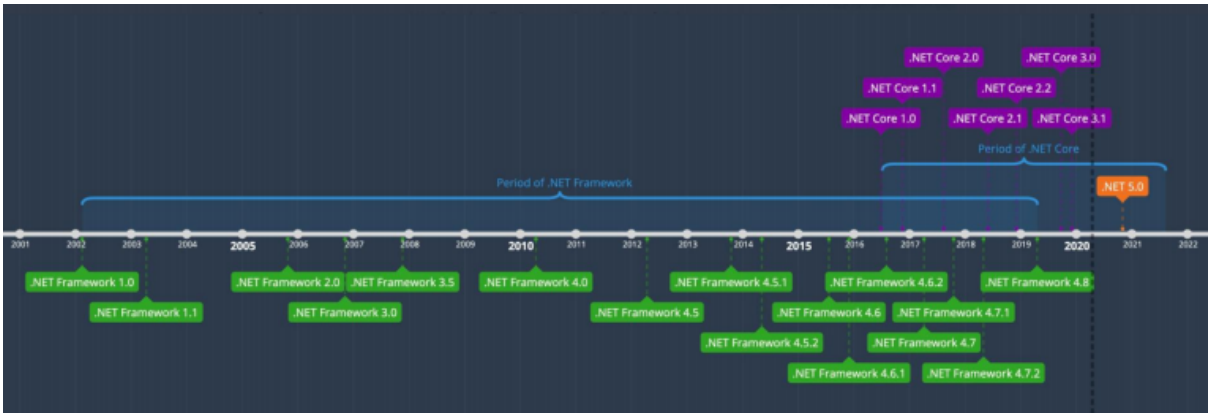
1 WPF

Wat is wpf?

- CUI vs GUI
- Evolutie van (Microsoft) ontwikkeltools:
  - WinForms
    - Eerste versie bij .NET platform
  - WPF
    - Windows Presentation Foundation
    - Meer mogelijkheden dan Winforms
  - UWP, WinUI, …
    - “Beperkt” tot Windows 10 (PC, tablet, Hololens, …)

Evolutie van .NET



- .NET core is dus de nieuwe “standaard”, deze gebruiken ipv .NET framework

Vensters

- Een WPF applicatie wordt opgebouwd met 1 (of meerdere) “window(s)”
- Elk window bestaat uit 2 bestanden:
  - **XAML** (= inhoud, layout & opmaak)
  - **cs** (= c# code die zal worden uitgevoerd)
- Design view wordt gegenereerd adhv xaml

XAML

- e**X**tensible **A**pplication **M**arkup **L**anguage
- “Beschrijft” de UI
- Gebaseerd op XML
- Eerste element (root) = “window”
  - Daaronder een “layout” element (bv “grid)
    - Daaronder 1 of meerdere “controls” (text box, btn, …)

Controls en hun eigenschappen

- Elke control heeft een aantal instelbare eigenschappen (properties)
- Kunnen worden ingesteld in de XAML en / of via de properties view.
- In XAML worden enkel eigenschappen bewaard die niet de “default” value hebben
- Stel best een “Name” in voor de control aan te roepen in code

Event driven programming

- In een console application gebeurt alle invoer (ReadLine) sequentieel en bepaalt de developer de volgorde
- In een WPF application kan de gebruiker verschillende volgordes hanteren en eerder zelf bepalen hoe hij/zij de gegevens ingeeft
- Als je in een WPF application op de hoogte wil gesteld worden van een gebeurtenis moet je inschrijven op “events”
  -> **Event driven programming**

Console.Readline() & Console.Writeline() ??

- Er is geen “Readline” dus we **lezen** de **“Text” property** van de **Textbox uit**
- Er is geen “Writeline” dus we **stellen** de **“Text” property** van de **Textbox in**

Array, Stack & Queue

Arrays

- Lijst met getallen, woorden, …
- Snel element zoeken adhv zijn ‘index’
- 1-dimensie, 2-dimensies, ...
- Volgorde van de elementen blijft behouden
- jagged-array

Stack

- Een **“stack”** is een lineaire datastructuur
- 1 dimensie
- Wordt ook wel een **LIFO** lijst genoemd
  - Last-In, First-out
- Een element toevoegen: **Push**
- Een element afnemen: **Pop**
- Het bovenste element bekijken: **Peek**
- We kunnen een stack implementeren adhv een **1D-array**

Stack implementatie

- Array + **“Top” variabele** die de index aangeeft van het bovenste element
- Deze variabele zal mee verhogen / verlagen bij een push / pop
- Check: Array Leeg / Volzet !

Toepassingen stack

- Undo / Redo in paint, office
- Back / Forward in browser

Beperking van een stack op basis van Array

- **Grootte van de array** moet bij de start worden gekozen
  - Array **te groot** in verhouding tot het gebruik => verspilling van geheugen
  - Array **te klein** => stackoverflow
- **Oplossing**
  - Stack volzet ? -> Nieuwe array aanmaken en alle data over kopiëren

Queue

- Een **“queue”** is een lijst datastructuur
- Heeft 1 dimensie
- Wordt ook wel een **FIFO** lijst genoemd
    - First-In, First-Out
- Toevoegen van een element: **Enqueue**
- Verwijderen van een element: **Dequeue**
- We kunnen een queue ook bouwen adhv een **1D-array**

Queue implementatie

- Lineair queue
- Circular queue

Toepassing van een queue

- Print queue
- Tickets kopen
- Email, SMS versturen

Beperking van een queue op basis van Array

- **Grootte van de array** moet bij start worden gekozen
    - Array **te groot** in verhouding tot het gebruik => verspilling van geheugen
    - Array **te klein** => queue full
- **Oplossing**
    - Queue volzet ? -> Nieuwe array aanmaken en alle data over kopiëren

Algoritmes & Complexity

Algoritme

- **Een algoritme is een stappenplan om een bepaald probleem op te lossen.**
- Een **algoritme**
  - Is een ondubbelzinnig beschreven stappenplan dat duidelijk maakt wat er moet gebeuren.
  - Verwacht 1 of meerdere inputs (parameters), bv 2 getallen, een lijst van woorden, …
  - Produceert een set van outputs (resultaten), bv het grootste getal, gesorteerde lijst, …
  - Het zal gegarandeerd na een bepaalde tijd eindigen en een resultaat teruggeven.

Vb van een algoritme

Zoek de hoogste waarde in een lijst van getallen

- **Inputs**: een lijst van (minstens 1) getallen
- **Outputs**: een getal, dewelke het hoogste in de lijst voorstelt
- **Algoritme**
  - max = 0
  - voor elk getal **x** in de lijst, vergelijk met max. indien x > max, stel max = x
  - max is het hoogste in de lijst

Welke?

- **Zoekalgoritmes**
  - Zoeken naar een bepaald element in een set van data
- **Sorteeralgoritmes**
  - De volgorde van de elementen in een bepaalde volgorde plaatsen
- **Berekeningsalgoritmes**
  - Uit een set van gegevens een andere set van gegevens afleiden
- **Filteralgoritmes**
  - Uit een set van gegevens een subset afleiden

Beoordelen

- **Time-complexity**: hoeveel **tijd** neemt een algoritme in beslag in **functie van de input**
- **Space-complexity**: hoeveel **geheugen** neemt een algoritme in beslag in **functie van de input**

Time-complexity

```
/// <summary>
/// Zoek het element met de hoogste waarde in de gegeven lijst
/// </summary>
/// <param name="lijst">een lijst met enkel positieve getallen</param>
/// <returns>de hoogste waarde</returns>
1 reference
public static int ZoekMax(int[] lijst)
{
    var max = 0;
    for (long f = 0; f < lijst.Length; f++)
    {
        if (lijst[f] > max)
            max = lijst[f];
    }
    return max;
}
```

- Ik wens de **time-complexity** van bovenstaande algoritme te onderzoeken
- Ik meet de tijd nodig om het maximum te zoeken in een lijst van
    - **100.000** elementen = **0.68 ms**
    - **1.000.000** elementen = **6.9 ms**
    - **10.000.000** elementen = **66 ms**
    - **100.000.000** elementen = **616 ms**
- Wat zeggen deze metingen nu eigenlijk?
- Kan ik iets doen met de exacte waarden?
- Zijn de metingen altijd hetzelfde? **Nee**
    - Hoe krachtig is de PC
    - OS
    - Programmeertaal
    - Welke applicaties draaien er nog
    - …
- Maar in grafiek geeft dit wel een lineair verloop
    - Voor elk bijkomend element is er telkens evenveel extra tijd nodig

  -> Voor een andere pc kunnen de waarden verschillen, maar zal nog steeds lineair zijn.

Hoe beoordelen van complexity

- De absolute tijdmetingen zijn van meerdere factoren afhankelijk
    - Input
    - Hardware, OS, …
- We gaan bijgevolg geen rekening houden met absolute waarden, maar wel **met het verloop in functie van de invoer**
    - Hiervoor bestaat **een specifieke notatie**
        - Big O -> Worst case
        - Big Ω (Omega) -> Best case
        - Big Θ (Theta) -> Average case

Big-O Notation

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

- Afhankelijk van het verloop van de benodigde tijd van een algoritme wordt een andere notatie gebruikt
- Bijvoorbeeld: indien de tijd **lineair** verhoogt in functie van de input (n), dan duiden we dit aan met **O(n)** (wordt uitgesproken als: **"Big o of n"**)

Space Complexity

- Gebruiken we ook de **Big-O** notatie
- We kijken nu echter naar het **geheugenverbruik** van het algoritme.
- Maw. **welke variabelen** zijn er nodig binnen het algoritme
- We kijken wederom **niet** naar de **absolute waarden**
- Wel: Wat is het **verband** tussen de **input en deze variabelen**
- Maw. heeft de input een invloed op de grootte van deze variabelen

Recursion

- Recursie is een probleem kunnen opsplitsen in een eenvoudigere versie van zichzelf
- Bij recursie roept een methode zichzelf aan
- Alternatief voor “iteratie”

Recursie = eindig

- Een functie die zichzelf aanroept zou in theorie leiden tot een oneindige “loop”
  - In de praktijk -> stackoverflow exception
- Daarom dat elke recursieve functie **minstens 1 conditie** moet bevatten die deze recursieve lus doorbreekt.
  - Dit noemt men de **“base case”**

Biertoren

- Hoeveel blikjes hebben we nodig om een toren te bouwen met basis 4
  - $Aantal_4 = 4 + 3 + 2 + 1$
- Of met recursie
  - $Aantal_4 = 4 + Aantal_3$
  - $Aantal_n = n + Aantal_{n-1}$
- “Base case”
  - Aantal van 1 = 1

Complexity

Driehoeksgetal berekenen	Time complexity	Space complexity
Iteratief algoritme	O(n)	O(1)
Recursief algoritme	O(n)	O(n)
Algoritme met formule	O(1)	O(1)

Directe vs Indirecte recursie

- **Directe** recursie:
  - Een methode roept **zichzelf** aan
- **Indirecte** recursie:
  - Methode A roept methode B aan, die op zijn beurt terug methode A aanroept

Sorteeralgoritmen

Wat en waarom?

- Wat?
  - Een **sorteeralgoritme** is een algoritme om elementen in een lijst in een bepaalde volgorde te zetten.
- Waarom?
  - We willen een gesorteerde lijst op het scherm, …
  - We willen de mediaan vinden van een lijst getallen
  - Sommige **zoek**algoritmen verwachten dat de elementen reeds gesorteerd zijn.
  - …

Bubble sort

1. Loop door de te sorteren rij van n elementen
   a. Vergelijk daarbij elk element met het volgende
   b. Verwissel beide als ze in de verkeerde volgorde staan.
   c. Schuif dan een stapje op tot het einde is bereikt
2. Loop opnieuw door de rij, maar ga nu door tot het voorlaatste element, omdat het laatste element reeds het grootste in de rij is.
3. Nog een keer, maar negeer de twee laatste elementen, …
4. Ga zo door tot de hele reeks gesorteerd is

Time complexity O(n2) Space complexity O(1)

Selection sort

1. Zoek eerst de kleinste waarde in de lijst.
2. Verwissel het met de “eerste” waarde in de lijst. De eerste waarde staat nu op zijn plaats
3. Herhaal de bovenstaande stappen met de rest van de lijst.

Time complexity O(n2) Space complexity O(1)

Insertion sort

Werkt zoals het sorteren van een kaartspel in je hand

1. Zet de 2de kaart tov kaart 1 op de juiste plaats.
2. Kijk naar de volgende kaart en zet deze in de reeds gesorteerde rij links op de juiste plaats
3. Doe dit tot het einde van de reeks

Time complexity O(n2) Space complexity O(1)

- Is **“adaptief”**
- Is een **“stabiel**” algoritme

“Stabiele” algoritmen

- Bij een **“stabiel”** algoritme blijven gegevens met eenzelfde waarde in dezelfde volgorde staan nadat de lijst gesorteerd werd.
- Bij een **“niet-stabiel”** algoritme wordt er hiermee geen rekening gehouden en zou het dus kunnen dat de volgorde na sortering niet meer dezelfde is.

“In place” algoritmen

- Er is geen extra geheugen nodig om te sorteren
    - Voorbeeld: omkeren van een array
- Not in place: extra geheugen gebruiken

Quick sort

- **“Divide and conquer”** algoritme
    - Het probleem opdelen in 1 of meerdere deelproblemen van hetzelfde type, tot ze eenvoudig genoeg zijn.

1. Pivot selectie: Selecteer een element uit de reeks. Dit wordt het **pivot** element.
2. Partitioneren: Alle elementen **kleiner dan de pivot** worden aan de **ene zijde** bij elkaar geplaatst, **groter dan de pivot** aan de **andere zijde**
3. Herhaal dit **recursief**

- **Pivot selectie** kan op verschillende manieren, neem steeds:
    - Het eerste element
    - Het laatste element
    - Het middelste element
    - Een **willekeurig** element
    - De **mediaan**
- Een goede keuze van pivot kan het algoritme versnellen

- **Partitioneren** / opdelen in 2 groepen (kleiner / groter dan pivot)
    - Kan op verschillende manieren gebeuren
        - Van buiten naar binnen
        - Van links naar rechts
        - …
    - Zorgt ervoor dat de pivot zelf reeds op de juiste plaats komt te staan

Time complexity O(n2) Space complexity O(n)

“niet-stabiel” algoritme

Vergelijking

Name	Best	Average	Worst	Memory
Quicksort	n log n	n log n	n^2	log n (average)
Mergesort	n log n	n log n	n log n	n (worst case)
Heapsort	n log n	n log n	n log n	1
Timsort	n	n log n	n log n	n
Bubble sort	n	n^2	n^2	1
Selection sort	n^2	n^2	n^2	1
Insertion sort	n	n^2	n^2	1

Merge sort

1. Deel de reeks op in 2 groepen
2. Herhaal dit tot je groepjes van max 2 elementen overhoudt
3. Sorteer elk groepje
4. Combineer 2 groepjes terug en sorteer ze tegelijkertijd
5. Herhaal dit tot de volledige reeks terug is opgebouwd

Time complexity O(n.logn) Space complexity O(n)

Niet **“in place”** algoritme

OO-Datastructuren

Even terugblikken

- Tot nu toe hebben we gewerkt met array's
  - = Aaneengesloten blok geheugen
  - Voordelen
    - **Snelle lees / schrijftoegang** aan de hand van de **index**
  - Nadelen
    - **Op voorhand geheugen reserveren**, wat misschien niet volledig gebruikt wordt en de **grootte ligt** dan ook vast.
    - Array toch groter maken => nieuwe array aanmaken en **alle elementen overkopiëren**
    - Een element tussenvoegen, ook dan moeten **alle volgende elementen 1 plaats opschuiven**

Linked list

Losse elementen hier en daar in het geheugen die naar elkaar verwijzen

Node

- Elk element in een "linked list" wordt een **"Node"** genoemd
- Een node heeft steeds
  - Een **Value** (getal, string, …)
  - Een **Verwijzing** (referentie) naar de **eerstvolgende** Node is de lijst.

Singly Linked List (SLL)

- Een node verwijst telkens naar de volgende node in de lijst, dit noemt een **"Singly linked list"**
- Er **moet** ook een **referentie** zijn naar de **1e node** ("First of "Head")
- Soms houdt men ook een **referentie** naar de **laatste** node ("Last" of "Tail")
- In tegenstelling tot bij een array
  - Zitten de nodes dus verspreid in het geheugen
  - De enige manier om een bepaalde node "te vinden", is **de lijst vanaf de "Head" 1 voor 1 aflopen**

Element opzoeken

- Zoek het element met een bepaalde "value"
  - Beginnen bij "First"
  - Heeft deze node de gezochte value?
  - Ga verder via Next tot de te zoeken value
  - Of stop bij het einde van de list

Element achteraan toevoegen

- Beginnen bij "Last"
- Maak een nieuwe Node aan
- Laat de "Last" node verwijzen naar de nieuwe Node
- Laat de "Last" pointer verwijzen naar de nieuwe Node

Element tussenvoegen

- **Na** een bepaalde node (value = x)
  - Maak een nieuwe Node aan
  - Laat de nieuwe node verwijzen naar de “Next” node
  - Laat de Node ‘x’ verwijzen naar de nieuwe Node
- **Voor** een bepaalde node (value = x)
  - Maak een nieuwe Node aan
  - Laat de nieuwe node verwijzen naar node ‘x’
  - Laat de Node daarvoor verwijzen naar de nieuwe Node

Element verwijderen

- Een node verwijderen (value = x)
  - Laat de node ervoor verwijzen naar de “Next” node
  - De node ‘x’ kan verwijderd worden (gebeurt automatisch in .NET)

Doubly Linked List (DLL)

- Bij een doubly linked list verwijst elke node zowel naar
  - de **volgende node** (Next)
  - de **vorige node** (Previous)
  - Dit betekent dat we deze lijst in **2 richtingen** kunnen doorlopen
  - Of, als we een bepaalde node hebben dat we nu zowel de volgende als de voorgaande node kennen.

Node

- Een Node in een “DLL” heeft steeds
  - Een **Value** (getal, string, …)
  - Een **Referentie** naar de **eerstvolgende** Node in de lijst (Next).
  - Een **Referentie** naar de **eerste voorgaande** Node in de lijst (Prev).

Circulair

Er bestaan voor beide SLL en DLL ook circulaire versies

Time complexity

Operatie	Array	Singly Linked List	Doubly Linked List
Opzoeken “by index”	O(1)	O(n)	O(n)
Opzoeken ”by value”	O(n) (*)	O(n)	O(n)
Element achteraan toevoegen	O(1)	O(1)	O(1)
Element vooraan invoegen	O(n)	O(1)	O(1)
Element invoegen na een bepaalde index/node	O(n)	O(1)	O(1)
Element invoegen voor een bepaalde index/node	O(n)	O(n)	O(1)
Element verwijderen	O(n)	O(n)	O(1)

Memory usage

	Array	Singly Linked List	Doubly Linked List
Lege lijst	Vast aantal elementen op voorhand te reserveren. Zolang de array niet volzet is wordt er geheugen gereserveerd dat niet wordt gebruikt.	Geen reservatie op voorhand nodig. Geheugen wordt gebruikt naarmate er nodes bijkomen of verdwijnen.	
Wat "kost" 1 element	Enkel geheugen nodig om 1 "value" in te kunnen bewaren per element. (dus bv. 4 bytes voor een int,..)	Elke node heeft geheugen nodig voor de "value" + referentie naar de next node	Elke node heeft geheugen nodig voor de"value" + 2 referenties (next en previous)
Element toevoegen of invoegen	Geen extra geheugen nodig, tenzij de array volzet is. Dan moet er een nieuwe array worden aangemaakt en moet alle data worden over gekopieerd.	Extra geheugen nodig telkens voor 1 "node"	
Element verwijderen	Er komt geen geheugen vrij, tenzij de array zou verkleind worden als deze bv. achteraan leeg is, maar dat betekent ook weer de data over kopiëren.	Geheugen van 1 node wordt telkens terug vrijgegeven	

Wanneer wat gebruiken

Wat	Array	Singly Linked List	Doubly Linked List
Queue	✓	✓	✓
Stack	✓	✓	✓
Bubble sort	✓	✓	✓
Selection sort	✓	✓	✓
Insertion sort	✓	✓	✓
Quick sort	✓	✓	✓
Merge sort	✓	✓	✓

Binary tree

- Is een boomstructuur -> hiërarchische datastructuur bestaande uit "nodes"
- Bij een binaire boom heeft een "parent" max 2 "children"
- Men spreekt dan over de
  - "Left child"
  - "Right child"

"Sorted binary tree" of "Binary search tree"

- Bij een "sorted binary tree" is **ten allen tijde:**
  - De **waarde** van de **"left child" kleiner** dan die van de **"parent"**
  - De **waarde** van de **"right child" groter** dan die van de **"parent"**

Verwijderen van elementen uit een BST

- Indien de te verwijderen node
  - Een “parent” is **zonder “children”**, dan kan de node gewoon verwijderd worden
  - Een “parent” is **met maar “1 child”**, dan kan de node gewoon verwijderd worden
- Indien de te verwijderen node een “parent” is **met 2 “children”**
  - Dan gaan we eerst op zoek naar het **kleinste** element bij de **“right children**” van die node
  - Dit kleinste element zal de node vervangen.

Time complexity BST

- Worst case **O(n)**
  - Totaal ongebalanceerde BST
  - De BST is herleid tot een “linked list”
- Average case **O(log n) / O(h)**
  - Bij een gebalanceerde BST

Zoekalgoritmes

Waarom

- We willen nagaan of een element aanwezig is in een lijst
- We willen de plaats kennen van een bepaald element in een lijst
- We willen nagaan of een bepaald element meermaals aanwezig is in de lijst
- ..

Lineair search

- Ook wel “sequential” search genoemd.
- Algoritme
    - Start bij het begin van de lijst
    - Vergelijk met de te zoeken waarde
    - Ga naar het volgende element en herhaal ofwel tot
        - De waarde gevonden werd
        - Het einde van de lijst is
- Dit algoritme hadden we ook reeds gebruikt bij de linked list om een element op te zoeken
- Return waarde
    - **True / false**: het element is aanwezig / niet aanwezig
    - **Index**: plaats van het element in de lijst
    - **Node**: in geval van een linked list
- Voordeel
    - Eenvoudig algoritme
    - De lijst hoeft niet gesorteerd te zijn
- Nadeel
    - In het slechtste geval moet de volledige lijst doorlopen worden
      -> time complexity: **O(n)**

Binary search

- Andere benamingen: “half-interval search”, “logarithmic search”
- Voorwaarde: de lijst **moet** reeds gesorteerd zijn!
- Algoritme:
    - Neem het **middelste** element in de lijst
    - **Vergelijk** met de **te zoeken waarde**
    - Indien de **gevonden waarde < te zoeken:**
        - Neem de **rechter helft** en start opnieuw
    - Indien de **gevonden waarde > te zoeken:**
        - Neem de **linker helft** en start opnieuw
- “Divide and conquer” principe
    - Steeds de lijst in de helft delen
    - Werkt dus enkel met arrays
    - Recursie!
- Voordeel
    - Het is een snel algoritme: **O(log n)**
- Nadeel
    - Lijst moet reeds gesorteerd zijn
    - Niet bruikbaar met linked lists

Binary search / BST

- Werken op dezelfde manier
- Hebben dus ook dezelfde efficiëntie
- Beiden **O(log n)** time complexity

N	Max. aantal compares
7	3
15	4
512	9
1.000.000	20

Nog beter?

- Wat is de snelste manier om in een array te zoeken?
- Stel een lijst met volgende elementen
  - 2, 5, 6, 11, 13, 14, 15, 18
- Wat als mijn array zou opgevuld zijn met nullen

0	0	2	0	0	5	6	0	0	0	0	11	0	13	14	15	0	0	18	0

- Wat is de performantie dan om een element op te zoeken
  - Time complexity **O(1)**
  - Maar… Space complexity **O(max)**

Hashing

- Wat is hashing
  - een input naar een key of value van een vaste lengte
  - Hallo -> 287
- “One way”
- Gekende algoritmes
  - SHA
  - MD
- Toepassingen:
  - **Datastructuren:** Hashtable, Dictionary, …
  - **Authenticatie:** wachtwoorden worden nooit als “plain tekst” bewaard in de databank
  - **Patronen** zoeken in teksten (plagiaatdetectie)
  - …

Zoeken mbv “hashing”

- Hoe kunnen we de hash value nu concreet gebruiken
  - We zouden de hash value als index kunnen nemen van onze array
  - Dit geeft echter nog steeds veel lege ruimte in de array
- Extra toevoeging
  - Gebruik van module operator (%)

Hashing + Modulo operator = “Hash table”

- vb: verwacht max **10** elementen in de lijst -> **%10**

Original string	“Hash value”	%10
“hallo”	287	7
“dit is een tekst”	498	8
“hello”	189	9
“AP hogeschool”	344	4

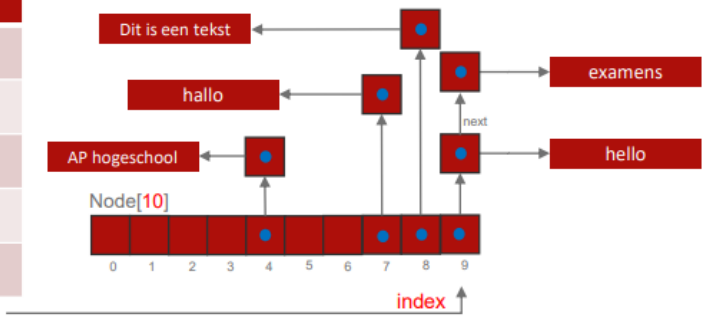
Hashing collisions

- Het zou kunnen dat het hashing algoritme dezelfde “hash value” geeft voor verschillende “input”
- Dit noemt men **“collisions”**

Collisions vermijden / oplossen

- Een goed hashing algoritme zorgt voor weinig of geen collisions
- Toch collisions
  - Werken met **array** + bijkomende **linked list**

Original string	“Hash value”	%10
“hallo”	287	7
“dit is een tekst”	498	8
“hello”	189	9
“AP hogeschool”	344	4
“examens”	189	9



Vergelijking zoekalgoritmes

Zoekalgoritme	Time Complexity	Array	Linked List
Lineair / Sequential search	O(n)	✓	✓
Binary Search	O(log n)	✓	X
Hash table (*)	O(1)	✓	X

C# - GetHashCode()

- Op de base class “object”
  - GetHashCode() zal een ‘default’ hashcode geven voor eender welk object of type in C#

Re-usability

Compiler?

- **Compiler errors** brengen een probleem **onmiddellijk** aan het licht tegenover **run-time exceptions** die pas optreden **als de code** daadwerkelijk **wordt uitgevoerd**
- Beeld je in dat je een applicatie hebt met 10.000 lijnen broncode, dwz. dat je elke lijn steeds moet testen als je iets hebt aangepast in de code.

Typesystems

- C# werkt
  - met een **"static type system"**
  - dwz dat **elke variabele of parameter** een **vast type** heeft
  - Dat type ligt vast bij de **declaratie**
- js werkt
  - met een **"dynamic type system"**
  - dwz dat het **type** van een variabele **kan wijzigen** tijdens de levensduur
  - Hey **type** kan zelfs **"undefined"** zijn

Static type system

- Werken met static types
  - **Intellisense:** Je weet "at development time" exact welke properties en methodes je kan aanroepen
  - **Type safety:** Compiler zal fouten geven als je de applicatie compileert

Generics

- We kunnen het Type dan een functie / klasse zal gebruiken meegeven
- IComparable
  - Er moet een **systeem** bedacht worden waardoor we 2 objecten van **eender welk type** met elkaar kunnen vergelijken
  - -> Interface **IComparable**
  - **Alle c# types** implementeren reeds deze interface
  - Dit is een **uniforme** manier om 2 elementen van eender welk type met elkaar te kunnen vergelijken
  - Vergelijken met **CompareTo()**

Sorteren op meerdere manieren ?

- Ik wil graag 2 lijsten van Auto's maken waarbij:
  - De ene lijst is gesorteerd op **'Aantalkm'**
  - De andere is gesorteerd op **'Bouwjaar'**
- Hoe kan ik dit realiseren als de klasse **zelf** de 'Compare' doet
- Niet: ik moet **een ander partij** de 'Compare' laten doen
- -> **IComparer**

IComparer

- Een ‘Comparer’ klasse doet niet meer dan 2 objecten van hetzelfde type met elkaar vergelijken
- Dit gebeurt adhv de Compare() methode

IComparable vs IComparer

- Gebruik **IComparable** als de ‘default’ comparer
- Gebruik **IComparer** om bijkomende Comparers te bouwen