Samenvatting Algoritmiek en Datastructuren

Cheat Sheet

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Datastructuren
- Array
- Gelinkte lijst
- Hashtabel

ADT's
- Stack
- Queue
- Dictionary

Algoritmen
- Bubble sort
- Selection sort
- Binary search

In dit vak staan drie concepten centraal

Datastructuren bepalen hoe data wordt opgeslagen in een computer. Bieden daarnaast bijpassende operaties aan.

Abstracte datatypes zijn een aantal samenhangende operaties die los staan van de manier waarop data wordt opgeslagen.

Algoritmen beschrijven stapsgewijs hoe een probleem opgelost wordt.

$O(n)$ tijdscomplexiteit: Voor de constatering dat de methode *** in het slechtste geval alle elementen moet afgaan. En voor de vermelding dat de looptijd daardoor rechteven-redig is het aantal elementen in de lijst.

$O(n^2)$ tijdscomplexiteit: kwadratisch

Week 1 - De basis.

Datastructuren.
Wijze van indeling of ordening van gegevens (data) in het (werk)geheugen van een computer. Bepalen hoe data wordt opgeslagen in een computer

Algoritmes.
Algoritmen manipuleren gegevens in datastructuren volgens een vast van tevoren bepaald stappenplan.

Standaard algoritmes voor alle datastructuren:
Invoegen
Wissen
Vervangen
Zoeken
Doorloop (iets doen met alle elementen)

Het toevoegen van een item aan een geordende array:

```
public virtual void Add(NAW item)
{
    if (_used == 0)
    {
        _nawArray[0] = item;
        _used++;
    }
    else if (_used < _size)
    {
        bool inserted = false;

        for (int i = 0; !inserted && (i < _used); i++)
        {
            if (_nawArray[i].CompareTo(item) >= 0)
            {
                for (int j = _used; j > i; j--)
                {
                    _nawArray[j] = _nawArray[j - 1];
                }
                _nawArray[i] = item;
                _used++;
                inserted = true;
            }
        }

        if (!inserted)
        {
            _nawArray[_used] = item;
            _used++;
        }
    }
    else
    {
        throw new NawArrayOrderedOutOfBoundsException();
    }
}
```

Het verwijderen van een item uit een geordende array:

```
public void RemoveAtIndex(int index)
{
    if (index >= _used || index < 0)
    {
        throw new NawArrayOrderedOutOfBoundsException();
    }

    for (int j = index + 1; j < _used; j++)
    {
        _nawArray[index] = _nawArray[j];
        index++;
    }
    _used--;
    _nawArray[_used] = null;
}
```

Sequential access (Linked List)

- Je kunt in $\mathcal{O}(1)$ naar het begin van de lijst
  - (en bij een tweezijdige ook naar het eind).
- Vanuit daar kun je in $\mathcal{O}(1)$ naar het volgende element
  - (en bij een tweerichtingslijst ook naar het vorige).
- Gelinkte lijsten zijn dus geschikt om elementen één voor één af te gaan
- Deze manier van benaderen heet *sequential access*.

Week 2 - Binair zoeken, grote-O-notatie en bubble sort.

Binair zoeken.

Tijdscomplexiteit: $O(log(n))$.

Operatie	Ongeordende array	Geordende array
Invoegen	O(1)	O(n)
Verwijderen	O(n)	O(n)
Zoeken	O(n)	O(log n)

Code:

```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // found it
      else if(lowerBound > upperBound)
         return nElems;             // can't find it
      else                          // divide range
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1; // it's in upper half
         else
            upperBound = curIn - 1; // it's in lower half
         }  // end else divide range
      }  // end while
   }  // end find()
```

Grote-O-Notatie

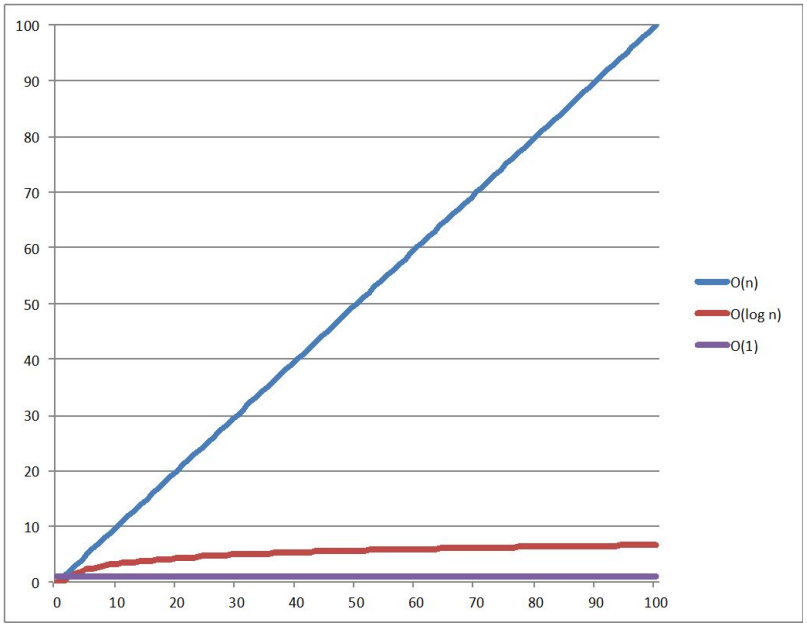
•De schaalbaarheid van een algoritme wordt de tijdscomplexiteit genoemd.

•De grote-O-notatie is een manier om de complexiteitsgraad van een algoritme uit te drukken

•Hoeveel tijd kost het algoritme bij een datastructuur met omvang ***n***

•Lineair zoeken: $O(n)$, algoritme duur schaalt lineair mee met omvang.

•Binair zoeken: $O(log(n))$, algoritme duur is ^{2}log van omvang.



Tijdscomplexiteit

- De tijdscomplexiteitsgraad van een algoritme zegt iets over de schaalbaarheid.
- We kijken hierbij naar het slechtste geval voor een algoritme.
  Voorbeeld: Bubble sort op een array die aflopend is gesorteerd

Geamortiseerde tijdscomplexiteit.

- Tijdscomplexiteit gaat standaard over het slechtste geval.
- Soms treedt dat slechtste geval slechts eens per *n* keer op.
- Dan kun je de de worst-case complexiteit “uitsmeren” over *n* aanroepen.
- Dat heet de geamortiseerde complexiteit.

Bubble Sort.

1. Kijk steeds naar 2 ‘buren’
2. Als de linkerbuur groter is dan de rechterbuur, verwissel je ze.
3. Daarna kijk je een index verder (de index wordt één groter)
4. Als je alle indexen gehad hebt, begin je opnieuw…

“D”	“B”	“A”	“S”	“L”	“N”	null	…	null	null
0	1	2	3	4	5	6	…	18	19

“B”	“D”	“A”	“S”	“L”	“N”	null	…	null	null
0	1	2	3	4	5	6	…	18	19

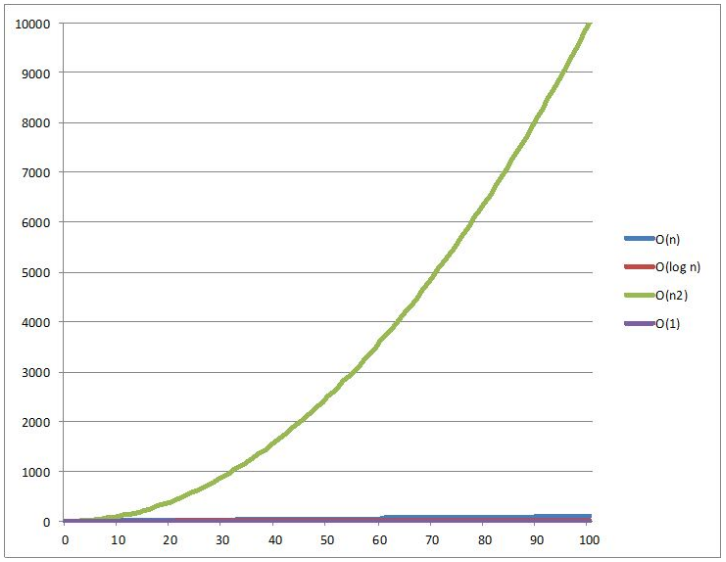
“B”	“A”	“D”	“S”	“L”	“N”	null	…	null	null
0	1	2	3	4	5	6	…	18	19

etc…

“B”	“A”	“D”	“L”	“N”	“S”	null	…	null	null
0	1	2	3	4	5	6	…	18	19

Na één doorloop staat het grootste item achteraan. Dus na elke iteratie hoef je nog maar een kleiner deel van de array te sorteren.

Tijd complexiteit bubblesort: $O(n^2)$
-Bij een 2x zo grote omvang van de datastructuur duurt het algoritme 4x zo lang
-Bij een 10x zo grote omvang al 100x zo lang



Code:

```
    public void BubbleSort()
    {
        int output, input;

        for (output = _used - 1; output >= 1; output--)
        {
            for (input = 0; input < output; input++)
            {
                if (_nawArray[input].CompareTo(_nawArray[input + 1]) == 1)
                {
                    _nawArray.Swap(input, input + 1);
                }
            }
        }
    }

    public void BubbleSortInverted()
    {
        for (int i = 0; i <= _used - 2; i++)
        {
            for (int j = _used - 2; j >= i; j--)
            {
                if (_nawArray[j].CompareTo(_nawArray[j + 1]) == 1)
                {
                    _nawArray.Swap(j, j + 1);
                }
            }
        }
    }
}
```

Week 3 - Stabiliteit, selection sort en insertion sort.

Stabiliteit.

Stabiliteit is een belangrijke eigenschap van een sorteeralgoritme:

- Een efficiënt sorteeralgoritme (quicksort) heeft die eigenschap bijvoorbeeld niet. Dat is een negatief gevolg van het efficiënte algoritme.

Stabiel is gelijke items qua sortering gegarandeerd niet wisselen.

"Donald", "Duck"	"Dagobert", "Duck"	"Guust", "Flater"	"Katrien", "Duck"
0	1	2	3

"Dagobert", "Duck"	"Donald", "Duck"	"Katrien", "Duck"	"Guust", Flater"
0	1	2	3

Niet stabiel want Dagobert en Donald zijn (onnodig) gewisseld.

Selection Sort.

1. Je zoekt de kleinste waarde die in de datastructuur voorkomt
2. Die verwissel je met de eerste waarde
3. Daarna zoek je de kleinste waarde op de eerste na.
4. Die verwissel je met de tweede waarde
5. Totdat je de laatste waarde op zijn plek hebt gezet.

Tijdscomplexiteit: $O(n^2)$

Code:

```
public void selectionSort()
   {
   int out, in, min;

   for(out=0; out<nElems-1; out++)   // outer loop
      {
      min = out;                     // minimum
      for(in=out+1; in<nElems; in++) // inner loop
         if(a[in] < a[min] )         // if min greater,
             min = in;               // we have a new min
      swap(out, min);                // swap them
      }  // end for(out)
   }  // end selectionSort()
```

Insertion Sort.

1. Het eerste element in de datastructuur is het gesorteerde deel.
2. Kijk naar het element na het gesorteerde deel en plaats deze op de juiste plek in het gesorteerde deel.
3. Totdat het gesorteerde deel alle elementen van de datastructuur bevat.

Tijdscomplexiteit: $O(n^2)$

Code:
Onstabiel:

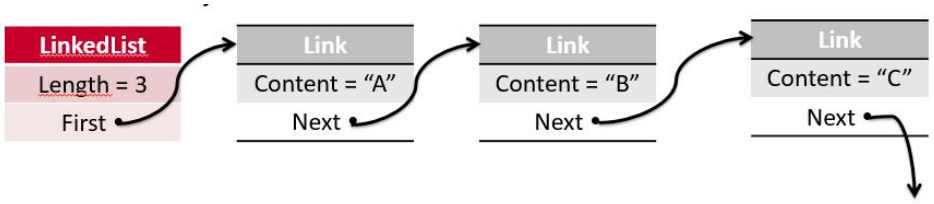
```
public void insertionSort()
   {
   int in, out;

   for(out=1; out<nElems; out++)     // out is dividing line
      {
      long temp = a[out];        // remove marked item
      in = out;                       // start shifts at out
      while(in>0 && a[in-1] >= temp) // until one is smaller,
         {
         a[in] = a[in-1];            // shift item right,
         --in;                       // go left one position
         }
      a[in] = temp;                  // insert marked item
      }  // end for
   }  // end insertionSort()
```

Stabiel:

```
public void insertionSort() {
	for (int out = 1; out < nElems; ++out) {
		long temp = a[out];
		int in = out;
		while (in > 0 && a[in - 1] > temp) {
			a[in] = a[in - 1];
			--in;
		}
		if (in != out)
			a[in] = temp;
	}
}
```

Week 4 - Gelinkte lijsten, Undo & Redo.

Lijst en schakel: twee aparte klassen.



Eigenschappen

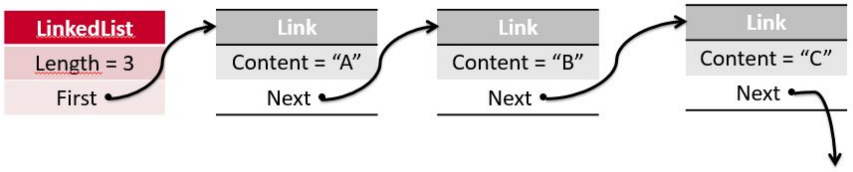
- Elementen los van elkaar opgeslagen
- Dynamische capaciteit, kan groeien/krimpen
- Element ophalen via index gaat in O(n)
- Homogeen (alle elementen hebben hetzelfde type)

Eigenschappen gelinkte lijsten:

- Één- of tweerichtings lijst
- Enkel- of tweezijdig

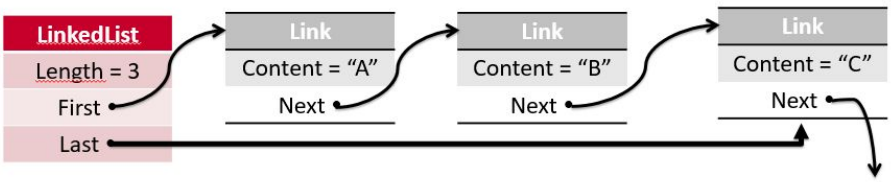
Enkelzijdige éénrichtings lijst

- Elke schakel heeft een verwijzing naar de volgende schakel (hier genaamd Next)
- De lijst heeft een verwijzing naar de eerste schakel van de lijst (hier genaamd First)
- Zoeken in O(n)



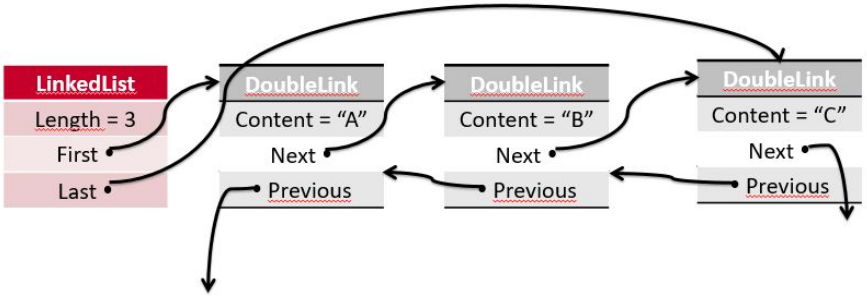
Tweezijdige éénrichtings lijst

- Elke schakel heeft een verwijzing naar de volgende schakel (hier genaamd Next)
- De lijst heeft een verwijzing naar de eerste èn laatste schakel van de lijst (hier genaamd First en Last)



Tweezijdige tweerichtings lijst

- Elke schakel heeft een verwijzing naar de volgende èn vorige schakel (hier genaamd Next en Previous)
- De lijst heeft een verwijzing naar de eerste èn laatste schakel van de lijst (hier genaamd First en Last)



Code:

```
public class NawDoublyLinkedList
{
   public DoubleLink First { get; set; }
   public DoubleLink Last { get; set; }


   public void InsertHead(NAW naw)
   {
      DoubleLink newItem = new DoubleLink { Naw = naw, Next = First, Previous = null };
      if (First == null)
      {
         Last = newItem;
      }
      else
      {
         First.Previous = newItem;
      }
      First = newItem;
   }
```

```
public DoubleLink SwapLinkWithNext(DoubleLink link)
{
    if (link.Next == null)
    {
        return null;
    }

    DoubleLink baselink = link;
    DoubleLink nextBaselink = link.Next;
    DoubleLink leftBaselinkNeedsAdjustment = baselink.Previous;
    DoubleLink rightBaselinkNeedsAdjustment = nextBaselink.Next;

    if (First == baselink)
    {
        First = nextBaselink;
    }
    if (leftBaselinkNeedsAdjustment != null)
    {
        leftBaselinkNeedsAdjustment.Next = nextBaselink;
    }
    nextBaselink.Previous = leftBaselinkNeedsAdjustment;
    nextBaselink.Next = baselink;
    baselink.Previous = nextBaselink;
    baselink.Next = rightBaselinkNeedsAdjustment;
    if (rightBaselinkNeedsAdjustment != null)
    {
        rightBaselinkNeedsAdjustment.Previous = baselink;
    }
    if (Last == nextBaselink)
    {
        Last = baselink;
    }
    return nextBaselink;
}
```

```
public void BubbleSort()
{

    bool isSorted = false;

    while (!isSorted)
    {
        isSorted = true;
        DoubleLink current = First;
        while (current != null && current.Next != null)
        {
            if (current.Naw.CompareTo(current.Next.Naw) > 0)
            {
                SwapLinkWithNext(current);
                isSorted = false;
            }
            current = current.Next;
        }
    }
}
```

Tijd complexiteit bubblesort: $O(n^2)$

Undo & Redo met een tweerichtings lijst.

```
public void Undo()
{
    if (Current == null)
    {
        return;
    }
    else if (Current.Previous == null)
    {
        ReverseOperation(Current);
        Current = null;
        First = null;
    }
    else
    {
        ReverseOperation(Current);
        Current = Current.Previous;
    }

}


public void Redo()
{
    if (Current == null && First == null)
    {
        return;
    }
    else if (Current == null && First != null)
    {
        Current = First;
    }
    else if (Current.Next == null)
    {
        return;
    }
    else
    {
        Current = Current.Next;
    }
    ApplyOperation(Current);
}
```

Week 5 - Abstracte datatypes (ADT's) en stacks.

Abstracte datatypen.

Er wordt gezegd wat ze kunnen, niet hoe ze dat voor elkaar krijgen.

- Een ADT is een soort interface.
- Neem een queue als voorbeeld:

```
interface IStringQueue
{
    void Enqueue(String queue);
    String Dequeue();
    bool IsEmpty();
}
```

- Je kunt deze queue implementeren met een array of gelinkte lijst
- **Keuzes maken betekent afwegen!**

ADT VS een datastructuur.

- Een ADT sluit eerder aan bij *functionele eisen*
  - Een ADT bepaalt wat een applicatie kan
- Een datastructuur sluit aan bij *niet-functionele eisen*
  - Een datastructuur bepaalt de schaalbaarheid, zoals
    - de tijdscomplexiteit van operaties en algoritmen,
    - en het geheugengebruik.

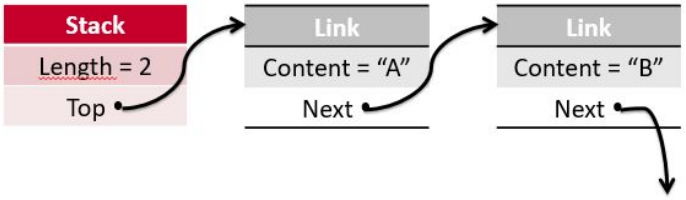
Stack.

Een stack (stapel) kent vier methodes:

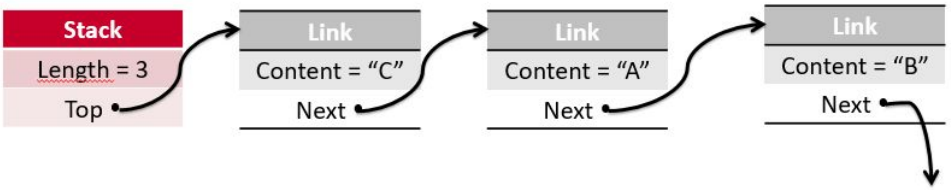
- push voeg item toe boven op de stapel
- pop haal bovenste item van de stapel
- peek kijk naar bovenste item
- isEmpty is de stapel leeg?
- isFull is de stapel vol? (alleen middels gebruik van een array)

LIFO - Last in, First out.

Stack kan geïmplementeerd worden in een eenvoudige enkel gelinkte lijst.



Pushen en poppen gebeurt aan de kop!



Code:

```
public class Stack : IStack
{
    protected StackLink First { get; set; }

    public void Push(string value)
    {
        if (value != null)
        {
            StackLink temp = new StackLink { String = value, Next = First };
            First = temp;
        }
    }

    public string Pop()
    {
        if (IsEmpty())
        {
            return null;
        }
        string temp = First.String;
        First = First.Next;
        return temp;
    }

    public string Peek()
    {
        if (!IsEmpty())
        {
            return First.String;
        }
        else
        {
            return null;
        }
    }
```

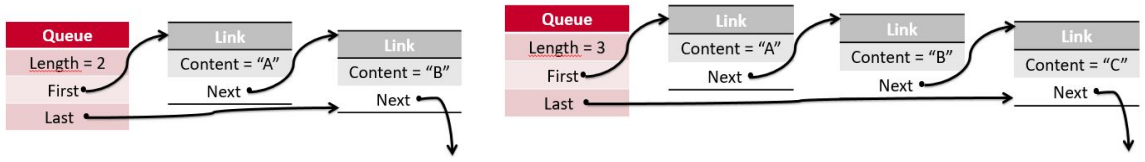
Week 6 - De ADT's: queues en priority queue.

Queue (wachtrij).

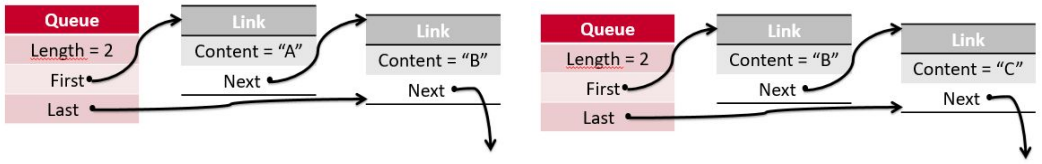
Enqueue / Insert	voeg item toe aan einde van wachtrij
Dequeue / Remove	haal item uit begin van de wachtrij
peek	kijk naar het voorste item
isEmpty	is de wachtrij leeg?

Een Queue is een **First In First Out** (FIFO) systeem. Ofwel wat er als eerste in geplaatst is komt er ook weer als eerste uit.

Queue implementatie (Gelinkte lijst): Enqueue



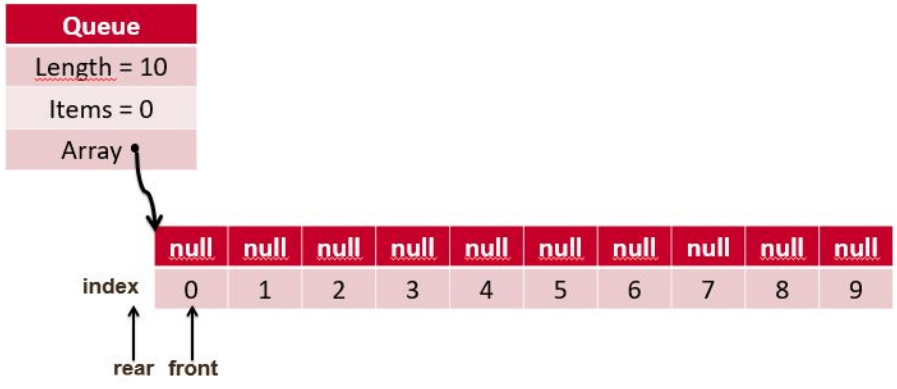
Queue implementatie (Gelinkte lijst): Dequeue

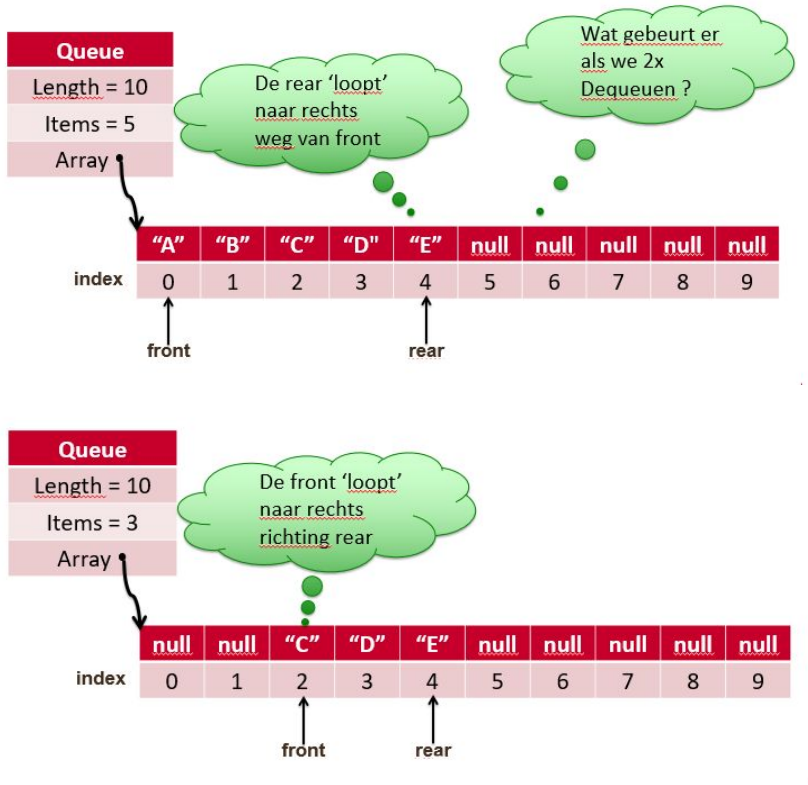


Queue in een array.

- De queue houdt 2 indexen bij front en rear
- We moeten het volgende implementeren:

Insert	voeg item in op index rear+1
Remove	haal item uit index front en verhoog front
peek	retourneer item op index front
isEmpty	Items = 0

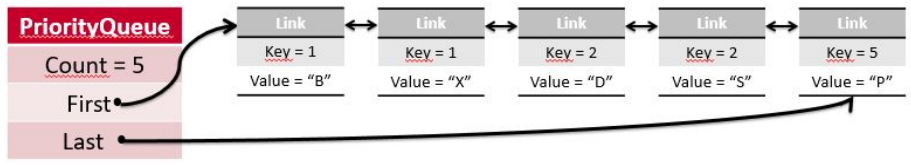




Priority Queue.

PriorityQueue is een wachtrij waarbij items met de hoogste prioriteit altijd als eerste worden gedequeuet.

Enqueue / Insert	voeg item toe afhankelijk van prioriteit
Dequeue / Remove	haal item uit begin van de wachtrij
peek	kijk naar het voorste item
isEmpty	is de wachtrij leeg?



Code:

Queue LinkedList

```
public void Enqueue(NAW naw)
{
    Link newLink = new Link { Naw = naw };

    if (First == null)
    {
        First = newLink;
        Last = First;
    }
    else
    {
        Last.Next = newLink;
        Last = Last.Next;
    }
    _count++;
}

public NAW Dequeue()
{
    if (First == null)
    {
        return null;
    }

    NAW result = First.Naw;
    First = First.Next;
    _count--;
    return result;
}
```

Queue array

```
public void Enqueue(NAW naw)
{
    if (_count.Equals(_size))
    {
        throw new NawQueueArrayOutOfBoundsException();
    }
    if (Rear == _size - 1)
    {
        Rear = -1;
    }
    _array[++Rear] = naw;
    _count++;

}

public NAW Dequeue()
{
    if (_count == 0)
    {
        return null;
    }

    NAW temp = _array[Front++];
    if (Front == _size)
    {
        Front = 0;
    }
    _count--;
    return temp;
}
```

Priority queue

```
public void Enqueue(int priority, NAW naw)
{
    if (priority < 0)
    {
        return;
    }

    NawQueueLinkedList list;

    if (_priorityQueue.ContainsKey(priority))
    {
        list = new NawQueueLinkedList();
        _priorityQueue.TryGetValue(priority, out list);
        list.Enqueue(naw);
    }
    else
    {
        list = new NawQueueLinkedList();
        list.Enqueue(naw);

        _priorityQueue.Add(priority, list);
    }
}
```

```
public NAW Dequeue()
{
    if (_priorityQueue.Count == 0)
    {
        return null;
    }
    var min = _priorityQueue.Keys[0];
    NawQueueLinkedList temp;
    _priorityQueue.TryGetValue(min, out temp);
    NAW tempNaw = temp.Dequeue();
    if (temp.Count() == 0)
    {
        _priorityQueue.Remove(min);
    }
    return tempNaw;

}

public int Count()
{
    int total = 0;
    IList<int> keys = _priorityQueue.Keys;
    foreach (var t in keys)
    {
        total += _priorityQueue[t].Count();
    }
    return total;
}
```

Week 7 - Het dictionary-ADT en de datastructuur hashtabel.

Dictionary.

- Een dictionary stelt je in staat waardes (values) te koppelen aan sleutels (key) van een willekeurig type.
- Vergelijk met een array: daar koppel je een getal, de index aan een waarde.
- Een Dictionary is een abstract datatype!
    - Vertelt wat een dictionary kan
    - Niet hoe een dictionary dat doet
- Een Dictionary wordt ook wel eens een Map genoemd.

Operatie	Omschrijving
Insert(key, value)	Associeert de waarde *value* met de sleutel *key*.
Find(key)	Geeft de waarde terug die geassocieerd is met sleutel *key*
Delete(key)	Verwijdert de waarde die geassocieerd is met de sleutel *key*

Hashtabel.

Complexiteit van operaties op een Dictionary geïmplementeerd met een gelinkte lijst

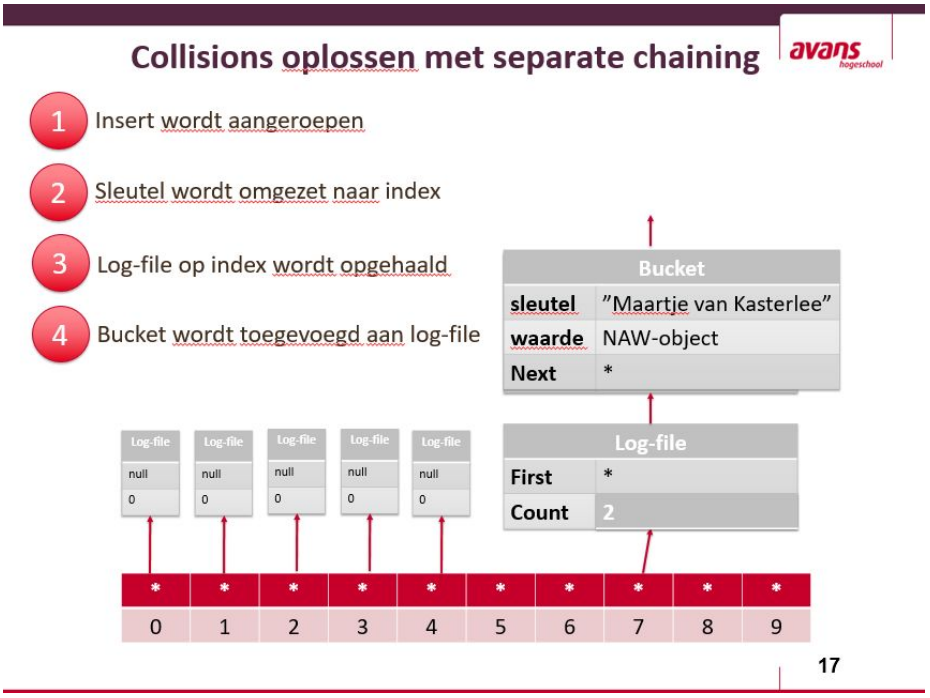
Operatie	Complexitieit
Insert(key, value)	*O(1)*
Find(key)	*O(n)*
Delete(key)	*O(n)*

- Waardes toevoegen, opvragen uit een array kost O(1) als je de index weet
- Maar een dictionary werkt met sleutels in plaats van indices
- Een hashtabel
    - slaat waardes op een array
    - en daarvoor moet deze eerst de sleutel naar een index converteren
- Een hashtabel is een datastructuur en geen abstract datatype
    - Want vertelt HOE waardes worden opgeslagen!

De weg van sleutel naar index.

1. var hashValue = “Jan Willemsen”.GetHashCode()
2. var compressedHashValue = hashValue % arraySize
3. var index= Math.Abs(compressedHashValue)

Hashtabel: collisions.
Probleem ontstaat omdat we maar 1 bucket per index kunnen opslaan
Oplossing: Plaats op elk item in de onderliggende array een linked-list van buckets



Load factor.

- Gebruik een goede hashfunctie
- Houd de load factor onder de 1/2

$$load\ factor = \frac{aantal\ elementen\ in\ de\ hashtable}{grootte\ van\ de\ onderliggende\ array}$$

- Wordt de load factor te hoog:
  - Maak een nieuwe hashtabel aan
  - Verplaats alle items van de oude naar de nieuwe hashtabel