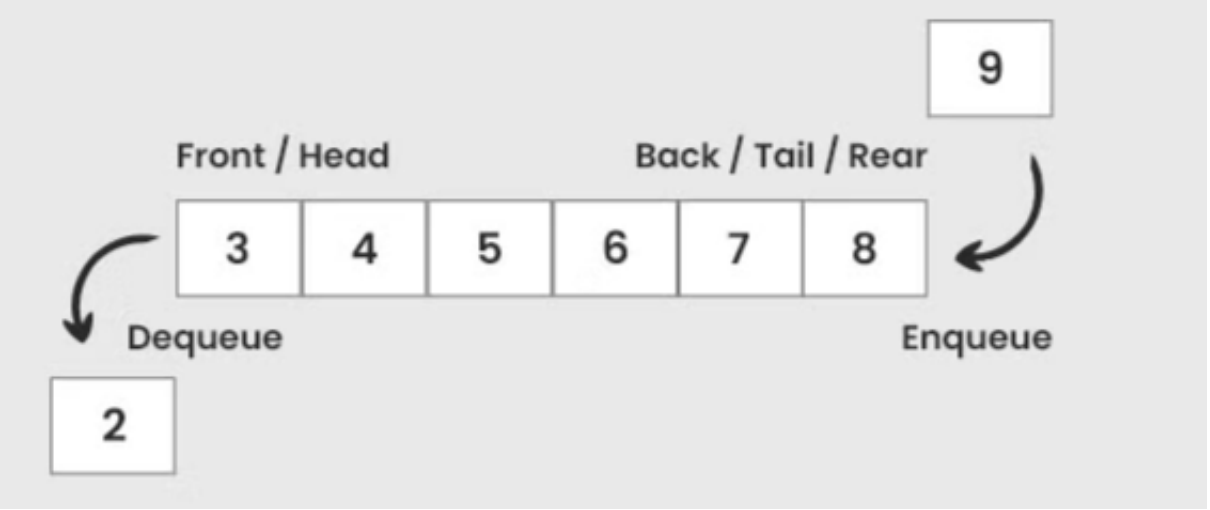
Vorlesung 28

Queues: Warteschlangen

Abstrakte Datenstruktur, die dem Prinzip „ **First in, First out** “ **(FIFO)** , folgt, wobei das erste zur Warteschlange hinzugefügte Element auch das erste ist, das entfernt wird.




```
import scala.reflect.ClassTag

trait MyQueue[A]:
    //Objekte eines beliebig aber festen Typs A sind gespeich

    //Vor: Keine
    //Erg: Keins
    //Effekt: x ist nun das aktuellste Element in der Queue
    def enqueue(x: A) : Unit

    //Vor: Der Stack ist nicht leer
    //Erg: Das älteste Element der Queue ist geliefert
    //Eff: Das älteste Element aus der Queue entfernt
    def dequeue(): A
```

```
    //Vor: Keine
    //Erg: Es ist true geliefert, genau dann wenn die Queue l
    //Eff: Keiner
    def isEmpty : Boolean

    //Vor: Keine
    //Erg: Die Anzahl der Elemente in der Queue ist geliefert
    //Eff: Keiner
    def size : Int
```

Queue Implementierung mit Linked List

Vorteile einer Implementierung durch eine verkettete Liste ist, dass diese wächst und schrumpft dynamisch zur Laufzeit, da Speicherplatz nur für tatsächlich gespeicherte Elemente benötigt wird. Außerdem können `enqueue(x)` und `dequeue()` in konstanter Laufzeit $O(1)$ durchgeführt werden, wenn Referenzen auf das erste und letzte Element gespeichert werden. Nachteile sind der Overhead wegen dem zusätzlichen Speicher der für die Pointer benötigt wird, schlechtere Cache Effizienz, da Speicheradressen an unterschiedlichen Orten verteilt sein kann, sowie die Ineffizienz beim Zugriff auf mittlere Elemente, bei denen die Liste mit $O(n)$ durchlaufen werden muss.

```
class LinkedNodeQueue[A] extends MyQueue[A]:
    private class Node(val item:A, var next: Node)

    private var _size : Int = 0
    private var frontNode : Node = null
    private var backNode :  Node = null

    def isEmpty: Boolean = frontNode == null
    def size: Int = _size
    def dequeue(): A =
        if isEmpty then
            throw new Exception("Emtpy Queue")
        else
            val n : Node = frontNode
```

```
            frontNode = frontNode.next
            _size -=1
            n.item

    def enqueue(x: A): Unit =
        val n = new Node(x, null)
        if !isEmpty then
            backNode.next = n
        else
            backNode = n
            frontNode = n
        _size += 1

    override def toString(): String =
        var s : String  = "front<"
        var n : Node = frontNode
        while n != null do
            s += " |" + n.item
            n=n.next
        s+="<back"
        s
```

Queue Implementierung durch Array statische Größe

Das Problem mit der Implementierung von Queues in Arrays statischer Größe ist, dass die Größe statisch ist, die Elemente aber von vorne entfernt werden. Das heißt, dass vorne im Array eventuell teilweise voll ist, aber hinten 'voll', also `back == array.length-1`. Als Lösung dafür nuten wir eine wrap-around Methode, in der wenn dieser Fall eintritt Elemente vorne angefügt werden.

Im folgenden Code sorgen die folgenden Zeilen für den wrap-around Effekt:

`front = (front+1) % n`, `back = (back+1) % n`

```
class ArrayQUeue[A:ClassTag](private val capacity:Int) extend

    private val n : Int = if capacity < 1 then 1 else capacit
    private val array:Array[A] = new Array[A](n)
    private var front = 0
```

```
    private var back = 0

    def isEmpty: Boolean = front == back

    def size: Int = (back-front+n)%n

    def enqueue(x: A): Unit =
        if size >= n-1 then throw new Exception("Queue full")
        array(back) = x
        back = (back+1) % n

    def dequeue(): A =
        if isEmpty then throw new Exception("Empty Queue")
        val result: A = array(front)
        array(front) = null.asInstanceOf[A]
        front = (front+1) % n
        result
```

Immutable Queue

Eine I**mmutable Queue** ist eine Warteschlange, bei der nach jeder Operation (z. B. Einfügen oder Entfernen eines Elements) eine **neue Queue** erstellt wird, anstatt die bestehende zu ändern. Das bedeutet, dass die Queue unveränderlich ist: Einmal erstellt, kann sie nicht verändert werden.

Immutable Datenstrukturen bieten ***referentielle Transparenz***. Das bedeutet, dass eine Methode, die eine neue Queue erzeugt, bei gleichem Input immer die gleiche Queue zurückgibt, *ohne Nebeneffekte.* Dies erleichtert das Testen, Debuggen und Verständnis des Programms, da der Zustand der Queue zu einem bestimmten Zeitpunkt klar und vorhersagbar ist.

💡 ***Concurrency:***
Immutable Datenstrukturen sind **thread-safe**, weil sie nicht verändert werden können. Mehrere Threads können gleichzeitig auf die gleiche Queue zugreifen, ohne sich um Synchronisierung oder Locking zu sorgen, da es keine Möglichkeit gibt, den Zustand der Datenstruktur zu ändern.

💡 ***Strukturelle Teilung*** *(Structural Sharing)*: Eine neue Queue nutzt viele Teile der alten Queue, um Speicher zu sparen und unnötige Kopieroperationen zu vermeiden. Beispielsweise wird bei einer `enqueue`-Operation nur das neue Element und die neuen Zeiger erstellt, während die bereits existierenden Elemente unverändert bleiben und von der neuen Queue referenziert werden.

Im folgenden wollen wir diese Vorteile einer Immutable Queue nutzen. Dafür definieren wir sie zuerst als Trait.

```
// ADT ImmutableQueue

trait ImmutableQueue[A]:
    //Gespeichert werden beliebige Daten von einem festen Typ
    //Vor: Keine
    //Erg: Die Anzahl der Elemente in der Queue ist geliefert
    //Eff: Keiner
    def size:Int

    //Vor: Keine
    //Erg: Es ist true geliefert, genau dann wenn die Queue l
    //Eff: Keiner
    def isEmpty:Boolean

    //Vor: keine
    //Erg: eine neue ImmutableQueue mit elem als neustem Elem
```

```
    def enqueue(elem:A) : ImmutableQueue[A]


    //Vor: Die Queue ist nicht leer
    //Erg: ein Tupel aus dem älesten Element und der Queue oh
    def dequeue:(A, ImmutableQueue[A])
```

Die Überlegung bei unserer Implementierung ist, dass wir versuchen die Ineffizienz von Linked Lists in Scala zu umgehen indem wir statt einer, zwei Listen Nutzen:

```
| x1 x2 x3 | x4 x5 x6 | ->  | x1 |   und   | x6 |
        front         back  | x2 |         | x5 |
                            | x3 |         | x4 |
                             ------          ------
                              aus             ein
```

Die Methoden funktionieren dann wie folgt:

- `enqueue(x)`: Füge x vorne an ein hinzu und gebe neue Queue zurück → Laufzeit $O(1)$
- `dequeue()`: Wenn aus nicht leer: gebe Tupel aus `.head` und neuer Queue aus `tail.ein` zurück → $O(1)$
- Wenn aus leer ist, ein aber nicht: drehe ein um und mache zum neuen aus, ein ist dann leer. Dann wie oben. → Laufzeit $O(n)$, jedoch ist diese Laufzeit amortisiert

```
// Mache den primären Konstruktor private, damit eine leere Q
class FunctionalListQueue[A] private
    (private val input:List[A],
    private val output:List[A])
    extends ImmutableQueue[A]:

    def this() = this(Nil, Nil)

    private def reverse : FunctionalListQueue[A] =
        if input.isEmpty then FunctionalListQueue[A](input.re
```

```
    def size : Int = input.length + output.length
    def isEmpty : Boolean = size == 0 // input.isEmpty && out

    def enqueue(elem:A):FunctionalListQueue[A] =
        new FunctionalListQueue[A](elem::input, output)

    def dequeue:(A, FunctionalListQueue[A])=
        if isEmpty then throw new Exception("Empty Queue")
        val q = this.reverse
        (q.output.head, new FunctionalListQueue[A](q.input, q
```

Prioritätswarteschlange

Ähnlich zu normaler Queue, aber jedes Element hat eine Priorität aus einem total geordnetem Universum. Elemente mit kleinerer Priorität müssen vor Elementen mit größerer Priorität
ausgegeben werden. Zur Übersichtlichkeit ignorieren wir die Elemente und betrachten nur die
Prioritäten.

Prioritätswarteschlangen haben vielseite Anwendungsbereiche wie:

- Prozessverwaltung im Betriebssystem
- Datenübertragung im Netzwerk
- Datenbankmanagementsysteme
- Algorithmen, z. B. Dijkstra für kürzeste Wege in Graphen
- Druckerwarteschlangen

```
// ADT PrioQueue
import scala.reflect.ClassTag

trait PrioQueue[K:Ordering]:
    //In einer Prioritätswarteschlage werden Elemente aus ein

    //Vor: keine
    //Eff: Das Element key ist zur Prioritätswarteschlange hi
    //Erg: kein
```

```
    def insert(key:K):Unit

    //Vor: Die Prioritätswarteschlange ist nicht leer
    //Eff: Das kleinste Element ist aus Prioritätswarteschlan
    //Erg: Das kleinste Element ist geliefert
    def extractMin():K

    //Vor: keine
    //Eff: keiner
    //Erg: true ist geliefert, genau dann, wenn die Priorität
    def isEmpty: Boolean
```

Implementierung mit Verketteten Knoten

Variante 1: Speichere Elemente aufsteigend, also $x_i \leq x_{i+1}$ für $1 \leq i \leq n$

- `extractMin()` → gebe Eintrag vom ersten Knoten aus und lösche diesen, entspricht `pop()` oder `dequeue()` → hat Laufzeit von $O(1)$
- `insert(elem)` → finde die Stelle an der `elem` in sortierter Reihenfolge steht und füge dort neuen Knoten ein → hat worst case Laufzeit von $O(n)$

Dummy Knoten: Statt bei leeren `PrioQueue` auf null zu zeigen, zeige auf Dummy Knoten mit Eintrag null. Diesen nennen wir `anchor`, und sorgen damit für weniger Sonderfälle

```
class SortedNodesPrioQueue[K:Ordering] extends PrioQueue[K]:
    private val ord = summon[Ordering[K]]
    import ord.mkOrderingOps

    private class Node(val item:K, var next:Node)

    private var anchor: Node = Node(null.asInstanceOf[K], nul

    def isEmpty: Boolean = anchor.next == null

    def extractMin():K =
```

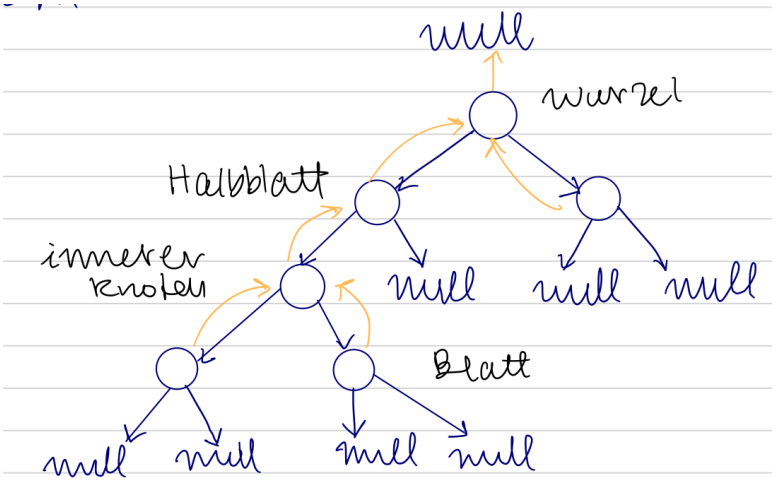
```
        if isEmpty then throw new Exception("Empty PrioQueue"
        val result:K = anchor.next.item
        anchor.next = anchor.next.next
        result

    def insert(elem:K):Unit =
        var current:Node = anchor
        while(current.next != null && current.next.item < ele
        current.next=Node(elem, current.next)

def linkedNodeSort[A:Ordering](array:Array[A]):Unit =
    val n = array.length
    val pq:PrioQueue[A] = new SortedNodesPrioQueue[A]
    for a <- array do pq.insert(a)
    for i <- 0 to n-1 do array(i) = pq.extractMin()
```

Variante 2: Speichere Elemente in beliebiger Reihenfolge

- `insert(elem)` → füge neuen Knoten vorne an, entspricht `push(x)` → hat Laufzeit von $O(1)$
- `extractMin()` → rekursive oder iterative Minimums Funktion mit temp um das Minimum zu finden. Entferne Knoten mit Minimum → hat Laufzeit von $O(n)$

Implementierung mit Binärbäumen

Für eine bessere Implementierung brauchen wir Binärbäume. Die Knoten haben dann jeweils ein linkes und/ oder ein rechtes Kind. Der erste Knoten ist die Wurzel. Die untersten Knoten heißen Blätter. Alle Knoten dazwischen können als innerer Knoten oder Halbblatt bezeichnet werden.



- Tiefe eine Knotens: Anzahl der Schritte von der Wurzel
- Höhe des Baums: Maximale Tiefe eines Knotens +1
- Größe eines Baums: Anzahl der Knoten

Binärer Heap

Ein binärer min-heap T ist ein Binärbaum mit Elementen aus einem total geordneten Universum in dem Knoten mit den folgenden Eigenschaften

a. Ordnungseigenschaft: Der Eintrag in jedem Knoten ist ≤ den beiden Einträgen in den Kinder-Knoten

b. Vollständigkeitseigenschaft: Sei h die Höhe des Baumes T. Die Ebenen 0 bis h-1 sind komplett gefüllt und die Blätter in Ebene h sind von links aufgefüllt

Eigenschaften:

- Minimum eines Min-Heaps steht in der Wurzel. Beweis über strukturelle Induktion
- Sei n die Größe des Heaps und h die Höhe des Heaps, dann ist die Höhe asymptotische $O(logn)$ durch den maximalen Verzweigungsgrad. Beweis würde ebenfalls über strukturelle Induktion erfolgen.

Repräsentation als Array mit Indexstartpunkt =0

Um vom Elternknoten zum linken Kind zu kommen: 2(x+1)

Vom linken Kind zum Elternknoten: (x-1)/2

Vom Elternknoten zum rechten Kind: 2x + 2

Vom rechten Kind zum Elternknoten: x-2/2

```
class BinaryHeap[K:Ordering:ClassTag](capacity: Int) extends
    private val ord = summon[Ordering[K]]
    import ord.mkOrderingOps

    private val array:Array[K] = new Array[K](if capacity <1
    private val last:Int = -1

    private def parent(i:Int):Int = (i-1)/2
    private def left(i:Int):Int = 2*i +1
    private def right(i:Int):Int = 2*i + 2

    private def swap(i:Int, j:Int):Unit =
        val tmp:K = array(i)
        array(i) = array(j)
        array(j) = tmp
```