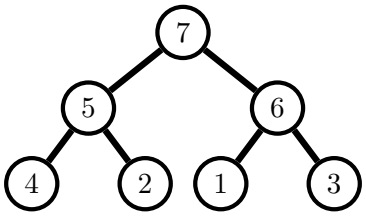
Technische Universität Berlin **SoSe 2024**
Fakultät II, Institut für Mathematik
Sekretariat MA 6–2, Dorothea Kiefer-Hoeft
Prof. Dr. Max Klimm
Dr. Tobias Hofmann, Martin Knaack, Marcel Wack

3. Programmieraufgabe Computerorientierte Mathematik II

Abgabe: 17.05.2024 über den Comajudge bis 17:00 Uhr



Der MaxHeap `[7,5,6,4,2,1,3]`.

Binäre Heaps

Implementieren Sie in `Julia` einen binären Heap. Ein binärer Heap ist ein vollständiger binärer Baum, der eine (Max-)Heap-Bedingung erfüllt: Für jeden Knoten i gilt, dass der Wert des Elternknotens von i größer oder gleich dem Wert von i ist. Sie sollen aber die Heap-Bedingung für einen allgemeinen `comparator` implementieren. D. h definieren Sie ein `mutable struct Heap{T <: Real}` mit den folgenden Feldern:

1. `data::Vector{T}`: Ein Vektor, der die Werte des Heaps speichert.
2. `comparator::Function`: Eine Funktion, die zwei Werte vergleicht und `true` zurückgibt, wenn der erste Wert größer oder gleich dem zweiten Wert ist. Größer hier im Sinne einer beliebigen Ordnung.

```
julia> Beispielcomparator(x,y) = x >= y;
julia> Beispielcomparator(1,2) # false
```

Hinweis: Das `T` in `Heap{T <: Real}` ist ein Typvariable, das bedeutet, dass beim erstellen eines Objektes vom Typ `Heap` ein konkreter Typ angegeben werden muss, der ein `subtype` von `Real` ist. Wir benötigen dies hier, um später zu gewährleiseten, dass der data-Vektor nur Werte vom Typ `T` enthält und die Funktion `comparator` Werte vom Typ `T` vergleicht.

```
julia> mutable struct IchBinEinStruct{T <:Real}
           field::T
       end
julia> IchBinEinStruct{Int}(1)
IchBinEinStruct{Int64}(1)
```

Schreiben sie anschließend zwei (innere) Konstruktoren für den Typ `Heap`:

1. `Heap(data::Vector{T}; comparator::Function)::Heap{T} where T <: Real`
   Erzeugt einen Heap mit den Werten `data` und dem Vergleichsoperator `comparator`.
2. `Heap(data::Vector{T})`: Erzeugt einen Heap mit den Werten `data` und dem Standard-Vergleichsoperator $\geq$.

Die vom inneren Konstruktor erzeugten Heaps müssen die Heapbedingung noch nicht erfüllen, und sollten die Positionen der Elemente des Vektors nicht verändern. Sie müssen jedoch einen `AssertionError` auslösen, wenn die Funktion `comparator` nicht für den Typ `T` definiert ist oder keinen Wert vom Typ `Bool` zurückgibt.

Hinweis: Mit `hasmethod` kann geprüft werden, ob die Funktion `comparator` für den Typ `T` definiert ist. Um den Typ des Rückgabewerts zu prüfen, kann `Base.return_types` verwendet werden. Gibt einen Vektor aller möglichen Rückgabetypen für eine gegebene Eingabe zurück.

```
julia> @assert hasmethod(+, Tuple{Int, String}) "Cant add Int and String"
ERROR: AssertionError: Cant add Int and String

julia> Base.return_types(+, Tuple{Int, Int})
1-element Vector{Any}:
 Int64
```

Heapify

Implementieren Sie die Funktion `is__Heap(heap::Heap{T}) ::Bool where T <: Real`, die prüft, ob der Heap die Heapbedingung erfüllt, wobei ein Key A größer oder gleich einem Key B ist, wenn `comparator(A,B) true` zurückgibt. `is__Heap` sollte `true` zurückgeben, wenn der Heap die Heap-Bedingung erfüllt, und `false` andernfalls.

Julia

```
julia> is__Heap(Heap([1,2,3,4,5,6,7], comparator=(x,y)->x>=y))
false

julia> is__Heap(Heap([1,2,3,4,5,6,7], comparator=(x,y)->x<=y))
true
```

Darauf aufbauend schreiben sie eine Funktion `heapify!(heap::Heap{T})::Heap{T} where T <: Real`, die die Einträge des Heaps so umsortiert, dass die Heapbedingung erfüllt ist.

Julia

```
julia> heapify!(Heap([1,2,3,4,5,6,7], comparator=(x,y)->x>=y))
[7, 5, 6, 4, 2, 1, 3] # This is a Heap!!

julia> heapify!(Heap([1,2,3,4,5,6,7], comparator=(x,y)->x<=y))
[1, 2, 3, 4, 5, 6, 7] # You don't need to overload Base.show
```

Abschließend können Sie nun einen äußeren Konstruktor für den Typ `Heap` implementieren:

1. `heap(data::Vector{T}; comparator::Function)::Heap{T} where T <: Real`: Erzeugt einen Heap mit den Werten `data` und dem Vergleichsoperator `comparator`. Der Heap sollte die Heapbedingung erfüllen, wenn dieser Konstruktor aufgerufen wird.

Julia

```
julia> heap([1,2,3,4,5,6,7], comparator=(x,y)->x>=y)
[7, 5, 6, 4, 2, 1, 3] # This is a Heap!!
```

Heap-Sort

Implementieren Sie die Funktion `heapSort!`, die den Heap sortiert. Für diese Funktion sind zwei Methoden zu implementieren:

1. `heapSort!(heap::Heap{T})::Heap{T} where T <: Real`: Sortiert den Heap `heap` in aufsteigender Reihenfolge und gibt ihn zurück.

heapSort!

```
julia> h1 = heap([1,2,3,4,5,6,7], comparator=(x,y)->x>=y)
[7, 5, 6, 4, 2, 1, 3]

julia> heapSort!(h1)
[1, 2, 3, 4, 5, 6, 7] # This is a Heap!!
```

2. `heapSort!(data::Vector{T}; comparator::Function)::Vector{T} where T <: Real`: Erzeugt einen Heap aus den Werten `data` und dem Vergleichsoperator `comparator`. Sortiert den Heap in aufsteigender Reihenfolge und gibt die sortierten Werte zurück.

height

```
julia> heapSort!([1,7,3,5,6,4,2], comparator=(x,y)->x>=y)
7-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
 7
```

Schließlich implementieren Sie die Funktion `maximum(heap::Heap{T})::T where T <: Real`, die das Maximum eines Heaps zurückgibt, der die Heapbedingung erfüllt. Die Funktion sollte `nothing` zurückgeben, wenn der Heap leer ist.

```
julia> maximum(heap([1,7,3,5,6,4,2], comparator=(x,y)->x>=y))
7
```

Die Verwendung von bereits in `Julia` vorhandenen Sortierfunktionen ist nicht erlaubt, wir behalten uns vor, Programme, die diese verwenden, als ungültig zu bewerten.

Zusammengefasst

Implementieren Sie die folgenden Funktionen und `structs` in `Julia`:

(a) `mutable struct Heap{T <: Real}`: Ein `struct` für einen binären Heap.

(b) `Heap(data::Vector{T}; comparator::Function)::Heap{T} where T <: Real`:
Ein Konstruktor für den Typ `Heap`.

(c) `Heap(data::Vector{T})`: Ein Konstruktor für den Typ `Heap`.

(d) `is__Heap(heap::Heap{T})::Bool where T <: Real`: Überprüft, ob der Heap die Heap-Bedingung erfüllt.

(e) `heapify!(heap::Heap{T})::Heap{T} where T <: Real`: Sortiert den Heap so um, dass die Heap-Bedingung erfüllt ist.

(f) `heap(data::Vector{T}; comparator::Function)::Heap{T} where T <: Real`:
Ein (äußerer) Konstruktor für den Typ `Heap`.

(g) `heapSort!(heap::Heap{T})::Heap{T} where T <: Real`: Sortiert den Heap in aufsteigender Reihenfolge.

(h) `heapSort!(data::Vector{T}, comparator::Function)::Vector{T} where T <: Real`:
Sortiert die Werte `data` in aufsteigender Reihenfolge.

(i) `maximum(heap::Heap{T})::T where T <: Real`: Gibt das Maximum des Heaps zurück.