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

4. Programmieraufgabe Computerorientierte Mathematik II

Abgabe: 24.05.2024 über den Comajudge bis 17:00 Uhr

Union-Find

In dieser Aufgabe sollen Sie eine weitere Datenstruktur implementieren, die als `UnionFind` bekannt ist. Diese Datenstruktur wird verwendet, um eine Partition einer Menge effizient zu speichern und zu verwalten. Wir implementieren sie mit Hilfe von Bäumen. Dazu definieren wir eine `struct Node`, der die folgenden Felder enthält:

1. `value::Int`: Der Wert des Knotens.
2. `parent::Union{Node, Nothing}`: Der Elternknoten des Knotens. Der Elternknoten eines Wurzelknotens ist `nothing`.

Für den Rest der Aufgabe verwenden wir nur den kanonischen Konstruktor für `Node`.

```
julia> x = Node(1, nothing)
Node(1, nothing)

julia> y = Node(2, x)
Node(2, Node(1, nothing))
```

Die Daten an sich werden jetzt in einem `Dictionary` von `Int` zu `Node` gespeichert. Schreiben Sie eine Funktion `union_find(data::Vector{Tuple{Int, Vector{Int}}})::Dict{Int, Node}`, die ein `Dictionary` von `Int` zu `Node` zurückgibt, das die Bäume der Union-Find Datenstruktur speichert.

Hinweis: `Tuple` sind in `Julia` oberflächlich betrachtet ähnlich zu `Vector`, jedoch sind sie unveränderlich. Dafür brauchen sie (meistens) weniger Speicherplatz und sind schneller zu erstellen. Sie können auf die Elemente eines `Tuple` wie auf die eines `Vector` zugreifen.

Hinweis: `Dict` ist in `Julia` ein assoziatives Array, das Schlüssel-Wert-Paare speichert. Es ist ähnlich zu einem `Dictionary` in Python. Ihr offensichtlicher Vorteil zu `Array` ist, die konstante Zeit, die benötigt wird, um auf ein Element zuzugreifen. Aber sie sind nicht immer die beste Wahl, da sie mehr Speicherplatz benötigen, und relative lange zum Erstellen brauchen.

```
julia> Partition = [(1, [1, 2, 3]) , (4, [4, 5]), (6,[6, 7, 8, 9])];

julia> nodes = union_find(Partition)
Dict{Int64, Node} with 9 entries:
  5 => Node(5, Node(4, nothing))
  4 => Node(4, nothing)
  6 => Node(6, nothing)
  7 => Node(7, Node(6, nothing))
  2 => Node(2, Node(1, nothing))
  9 => Node(9, Node(6, nothing))
  8 => Node(8, Node(6, nothing))
  3 => Node(3, Node(1, nothing))
  1 => Node(1, nothing)
```

find_set und find_set!

Implementieren Sie die Funktion `find_set(node::Node)::Node`, die den Wurzelknoten des Baumes, in dem sich der Knoten `node` befindet, zurückgibt.

Julia

```
julia> find_set(nodes[5])
Node(4, nothing)
```

Schreiben Sie nun eine Funktion `find_set!(node::Node)::Node`, die den Wurzelknoten des Baumes, in dem sich der Knoten `Node` befindet, zurückgibt. Dabei wird Pfadkompression angewendet, um die Struktur des Baumes zu optimieren. Indem Sie den gefunden Wurzelknoten als Elternknoten des Knotens `node` setzen, Sie müssen nicht den gesamten Pfad komprimieren.

union!

Implementieren Sie die Funktion `union!(node1::Node, node2::Node)::Nothing`, die die Bäume, in denen sich die Knoten `node1` und `node2` befinden, vereinigt. Dabei wird **zuerst** Pfadkompression angewendet, und der Wurzelknoten von `node2` wird optimal an den Baum von `node1` angehängt. Die Funktion gibt den Wurzelknoten des neuen Baumes zurück. Falls `node1` und `node2` gleich sind, soll die Funktion nichts tun.

Julia

```
julia> union!(nodes[1], nodes[4])

julia> nodes
Dict{Int64, Node} with 9 entries:
  5 => Node(5, Node(4, Node(1, nothing)))
  4 => Node(4, Node(1, nothing))
  6 => Node(6, nothing)
  7 => Node(7, Node(6, nothing))
  2 => Node(2, Node(1, nothing))
  9 => Node(9, Node(6, nothing))
  8 => Node(8, Node(6, nothing))
  3 => Node(3, Node(1, nothing))
  1 => Node(1, nothing)
```

add!

Schreiben Sie eine Funktion `add!(nodes::Dict{Int, Node}, value::Int)::Nothing`, die einen neuen Knoten mit dem Wert `value` zu den Bäumen hinzufügt. Der neue Knoten wird als Wurzelknoten eines neuen Baumes hinzugefügt. Die Funktion gibt `nothing` zurück. Sie sollte ein `AssertionError` werfen, wenn der Wert `value` bereits in den Bäumen enthalten ist.

heapSort!

```
julia> add!(nodes, 5)
ERROR: AssertionError: The element 5 is already in the partition

julia> add!(nodes, 11)
Node(11, nothing)

julia> nodes
Dict{Int64, Node} with 10 entries:
  11 => Node(11, nothing)
    ..
```

Zusammengefasst

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

(a) `struct Node`: Ein Struct zum Speichern der Daten eines Knotens in einem Baum.

(b) `union_find(data::Vector{Tuple{Int, Vector{Int}}})::Dict{Int, Node}`: Eine Funktion, die ein `Dictionary` von `Int` zu `Node` zurückgibt, das die Bäume der Union-Find Datenstruktur speichert.

(c) `find_set(node::Node)::Node`: Eine Funktion, die den Wurzelknoten des Baumes, in dem sich der Knoten `node` befindet, zurückgibt.

(d) `find_set!(node::Node)::Node`: Eine Funktion, die den Wurzelknoten des Baumes, in dem sich der Knoten `Node` befindet, zurückgibt. Dabei wird Pfadkompression angewendet.

(e) `union!(node1::Node, node2::Node)::Nothing`: Eine Funktion, die die Bäume, in denen sich die Knoten `node1` und `node2` befinden, vereinigt. Dabei wird Pfadkompression angewendet.

(f) `add!(nodes::Dict{Int, Node}, value::Int)::Nothing`: Eine Funktion, die einen neuen Knoten mit dem Wert `value` zu den Bäumen hinzufügt. Der neue Knoten wird als Wurzelknoten eines neuen Baumes hinzugefügt.