FHV

Algorithmen & Datenstrukturen

Stack, Queue und Priority Queue

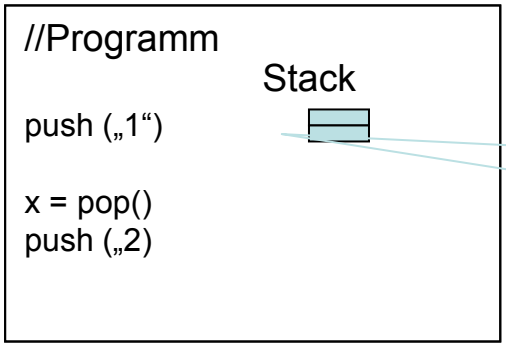
Wolfgang Auer

Motivation

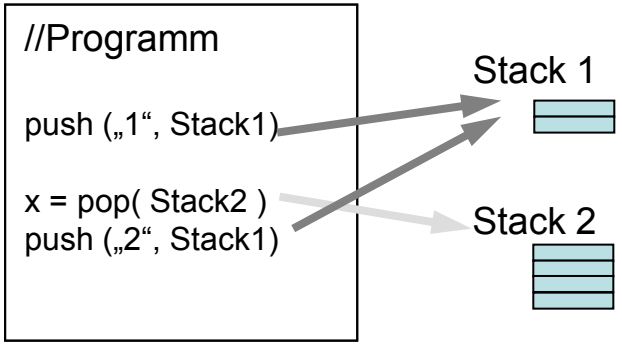
FHV

- Um eine Softwarekomponente (Modul) verwenden zu können, ist es nicht notwendig, deren interne Strukturen zu kennen
- Die Beschreibung der Funktion des Moduls erfolgt ausschließlich durch die **Beschreibung der Operationen**
- Operationen sind die einzige Möglichkeit, auf die Funktionalität zuzugreifen
- Das Verbergen des internen Aufbaus wird als das ***Geheimnisprinzip (Information-Hiding)*** bezeichnet

Abstrakte Datenstruktur / Abstrakter Datentyp



Es gibt eine genau einen Stack, auf dem alle Operationen ausgeführt werden



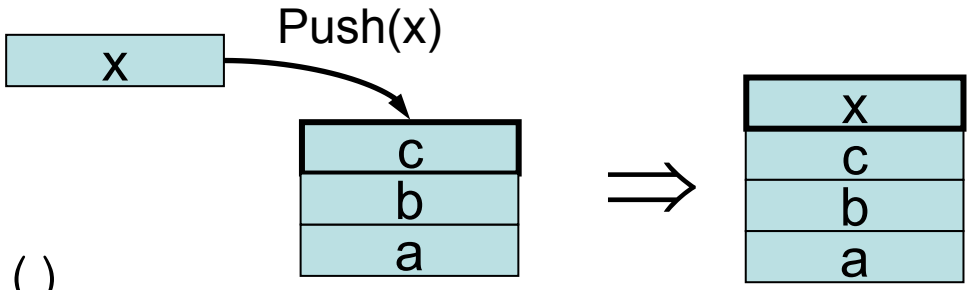
Stack (ADS)

FHV

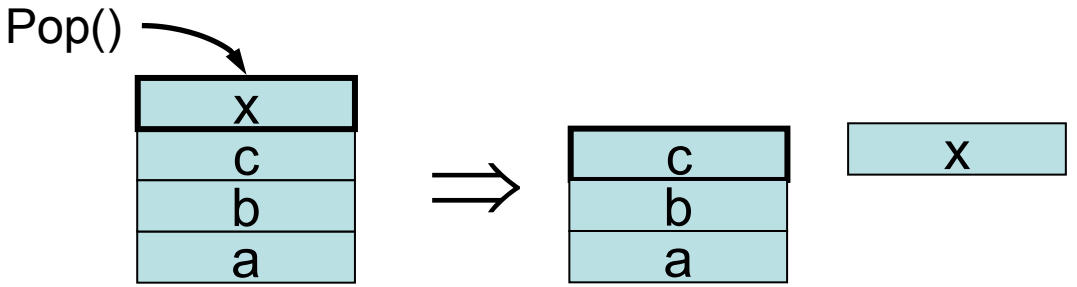
- Der *Stapel* (Stack) eine Liste von Elementen, die nach dem *Last-In-First-Out*-Prinzip organisiert ist und die folgenden Operationen zur Verfügung stellt:
  - `Push(x)`
    - Element *x* auf den Stapel legen
  - `Pop()`
    - Oberstes Element von Stapel entfernen
  - `IsEmpty()`
    - Testen auf leeren Stack
- Zugriff nur auf das oberste Element

Operationen

- `Push(x)`
  - Das Element *x* wird oben auf dem Stack gelegt. D.h. das Element wird zum obersten Element

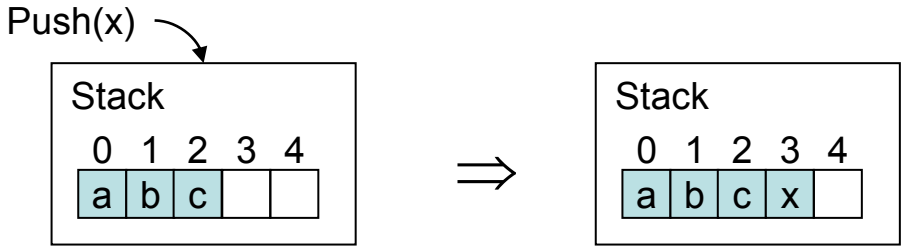


- `Pop()`
  - Pop entfernt das oberste Element vom Stack

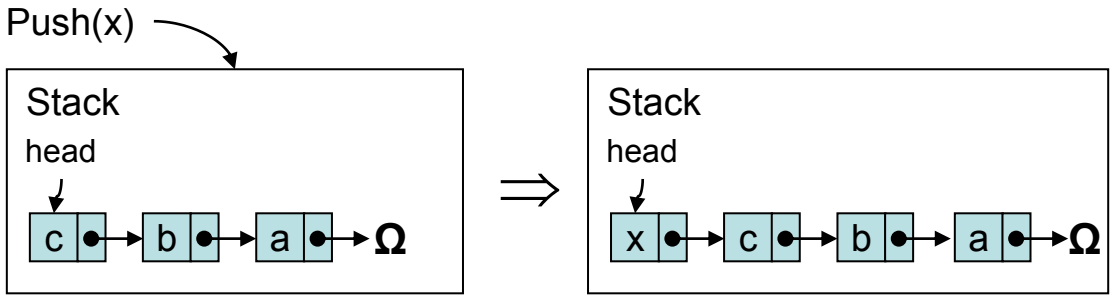


Realisierung

- Eine ADS sagt nichts über die technische Realisierung aus!
- Repräsentation eines Stacks
  - Feld (Array)
    - Eine sequentielle Repräsentation mit Hilfe eines Feldes wird als *Bound Stack* bezeichnet, da die Kapazität durch die Größe des Feldes begrenzt wird



  - Einfach Verkette Liste



    - Neue Elemente werden am Kopf eingefügt

Bewertung Stack

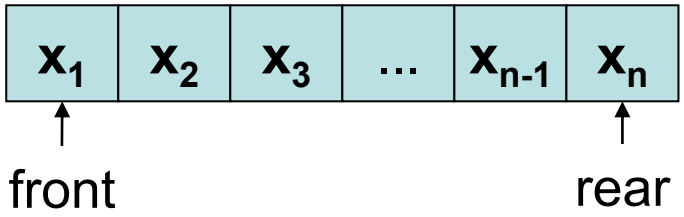
FHV

- Vorteile
  - ▲ Sehr nützlich in vielen Anwendungen zur Abarbeitung von Abläufen
  - ▲ Sehr effizienter Zugriff, egal ob durch Liste oder Array realisiert: O(1)
  - ▲ Dynamische Länge (nur bei Verwendung einer Liste)
  - ▲ Elementare, einfache Einfüge- (push) und Löschoperationen (pop): alle O(1)
- Nachteile
  - ▼ Nur für LIFO-Anwendungen nutzbar (Last In, First Out)
  - ▼ (Kein wahlfreier Zugriff auf Elemente)
  - ▼ (Keine Navigation über die Elemente)

Nachteile treten bei den Anwendungsgebieten des Stacks prinzipiell nicht auf. Sie sind daher als Hinweis zu sehen, dass der Stack die falsche Datenstruktur für die vorliegende Aufgabenstellung ist.

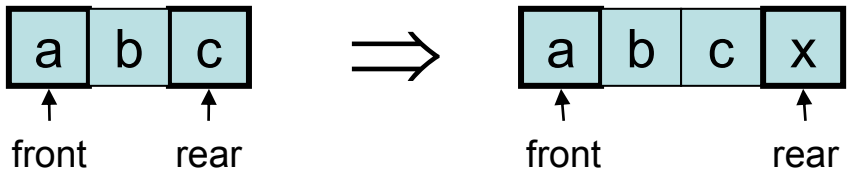
Queue (ADS)

- Die *Schlange* (Queue) eine Liste von Elementen, die nach dem *First-In-First-Out*-Prinzip organisiert ist und die folgenden Operationen zur Verfügung stellt:
  - `Enqueue(x)`
    - Element *x* an die “Warteschlange” stellen
  - `Dequeue()`
    - Erstes Element aus der Schlange entfernen
  - `IsEmpty()`
    - Testen auf leere Schlange
- Das Hinzufügen nur am Ende (rear) und das Entfernen am Anfang (front) erlaubt

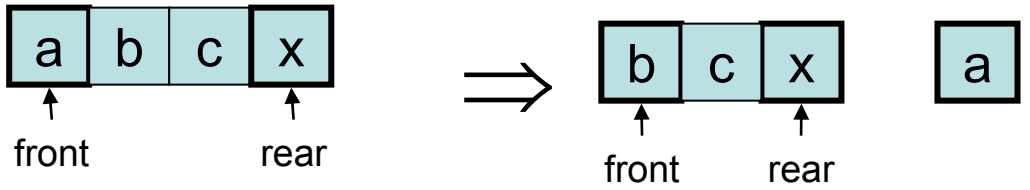


Operationen

- `Enqueue(x)`
  - Das Element *x* wird am Ende der Schlange angefügt



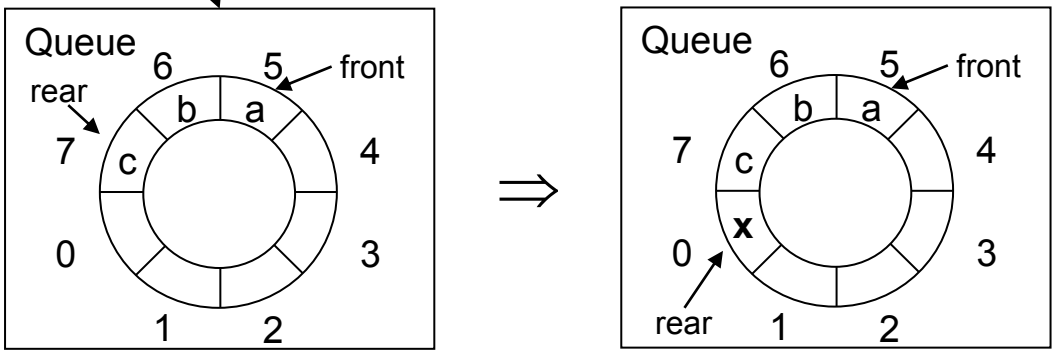
- `Dequeue()`
  - Entfernen des ersten Elements in der Schlange d.h. jenes Element, das sich am längsten in der Queue befindet



Realisierung

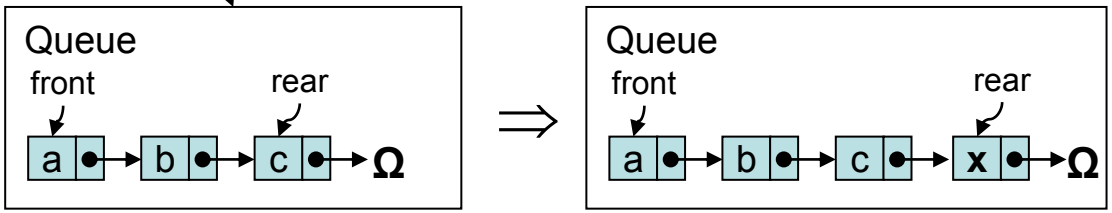
- Repräsentation einer Queue
  - Feld (Array)
    - Analog zum Stack spricht man hier von einer *Bound Queue*

Enqueue(x)



  - Einfach Verkette Liste

Enqueue(x)



    - Neue Elemente werden am Ende eingefügt

Bewertung

- Vorteile
  - Sehr nützlich in vielen Anwendungen zur Abarbeitung von Abläufen
  - Sehr effizienter Zugriff, egal ob durch Liste oder Array realisiert: O(1)
  - Dynamische Länge (nur bei Verwendung einer Liste)
  - Elementare, einfache Einfüge- (enqueue) und Löschoperationen (dequeue)
- Nachteile
  - Nur für FIFO-Anwendungen nutzbar (First In, First Out)
  - (Kein wahlfreier Zugriff auf Elemente)
  - (Keine Navigation über die Elemente)

Nachteile treten nur bei verkehrter Anwendung auf
⇒ Wähle eine andere Datenstruktur

Priority Queue (ADT)

FHV

- Eine *Prioritätswarteschlange* ist eine Abstrakte Datenstruktur zur Verwaltung von Datensätzen, die mit einer Priorität (Gewichtung) versehen sind.
  - Datensätze können wahllos in die *Priority Queue* eingefügt werden
  - Beim Entfernen von Elementen wird das Element mit der höchsten bzw. niedrigsten Priorität entfernt.

- Anwendung von *Priority Queues*
  - Simulationssysteme (Schlüsselwerte repräsentieren Ausführungszeiten von Ereignissen)
  - Prozessverwaltung (Prioritäten der Prozesse)
  - Numerische Berechnungen (Schlüsselwerte entsprechen den Rechenfehlern, wobei zuerst die großen beseitigt werden)
  - Grundlage für eine Reihe komplexer Algorithmen (Graphentheorie, Filekompression, etc.)

Priority Queue - Operationen

FHV

- Create
  - Erzeugen einer leeren Priority Queue
- Destroy
  - Zerstören einer Prioritiy Queue. D.h. Freigabe des gesamten belegten Speicherplatzes
- IsEmpty
  - Abfrage auf leere Priority Queue
- Insert
  - Einfügen eines Elements mit beliebiger Priorität
- Remove
  - Entfernen des Elements mit der höchsten Priorität

Repräsentationen einer Priority Queue

FHV

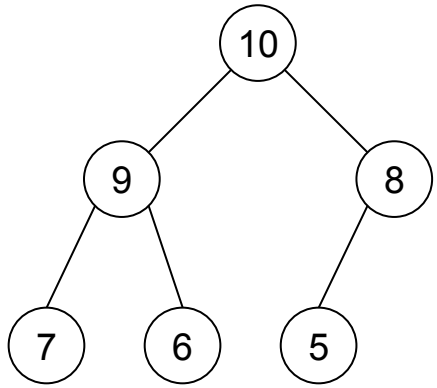
- Geordnete verkettete Liste
  - Elemente werden der Priorität entsprechend in die Liste eingefügt
    - Einfügen in O(n)
    - Löschen in O(1)
- Ungeordnetes Feld
  - Elemente werden unsortiert eingefügt d.h. erst beim Entfernen wird das Element mit der höchsten Priorität bestimmt
    - Einfügen in O(1)
    - Löschen in O(n)
- Heap
  - Binärer Baum mit der Eigenschaft, dass kein Wert eines Nachfolgers eines Knotens *K* größer sein darf als der Wert des Knotens *K*
    - Einfügen in O(log n)
    - Löschen in O(log n) (Zugriff in O(1) + Umorganisieren)

Heap

- Motivation: Für viele Anwendungen ist eine partielle Ordnung ausreichend.
- Ein Heap ist ein binärer Baum, in dem
  - jedem Knoten ein Schlüsselwert zugeordnet wird
  - der Schlüsselwert jedes Vorgängers eines Knotens größer oder gleich dem Schlüsselwert des Knotens ist (*Heap-Bedingung)*
  - die Wurzel den größten Schlüsselwert enthält

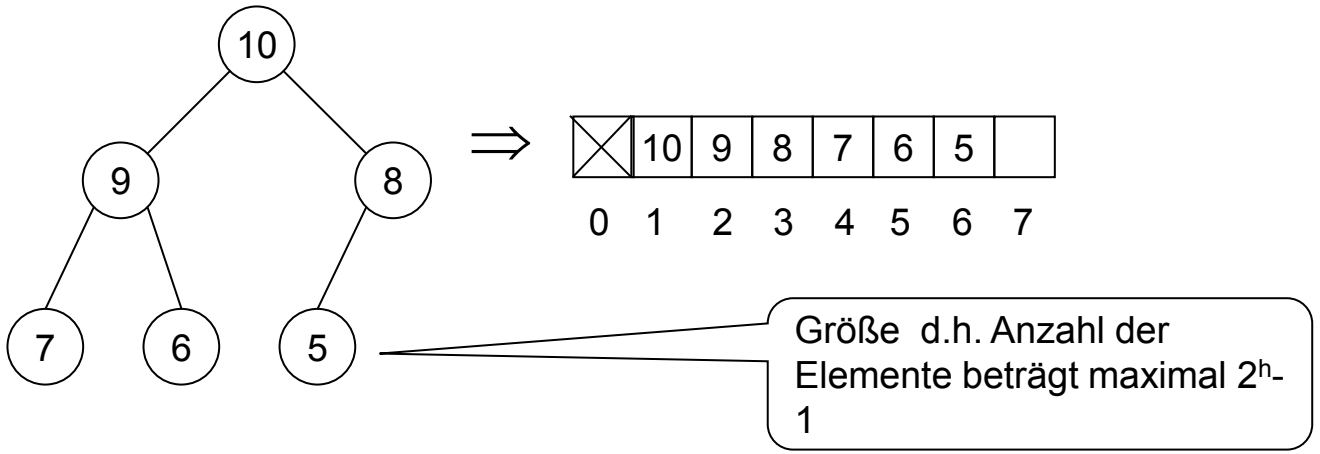
  Ein Heap verfügt über eine partielle Ordnung. Betrachtet man jeden Pfad isoliert, ist dieser absteigend sortiert.

Beispiel: { 5, 6, 7, 8, 9, 10 }



Realisierung mittels Feld

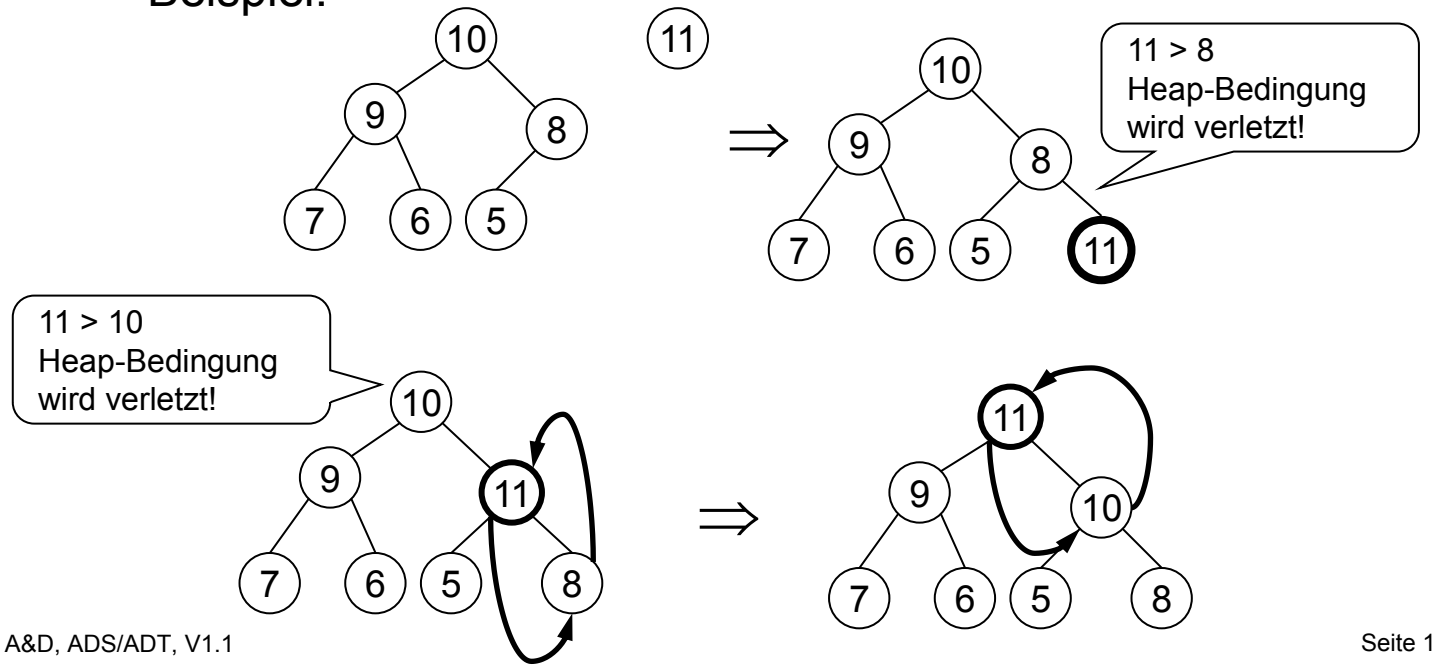
- Effiziente Darstellung eines vollständigen binären Baumes mit Hilfe eines Feldes. (Level-Order)



Wurzel	A[1]	
Linkes Kind von A[i]	A[2i]	für 2i ≤ n
Rechtes Kind von A[i]	A[2i + 1]	für 2i +1≤ n
Vater von A[i]	A[i/2]	für i > 1
A[i] ist Blatt	wahr	für 2i > n

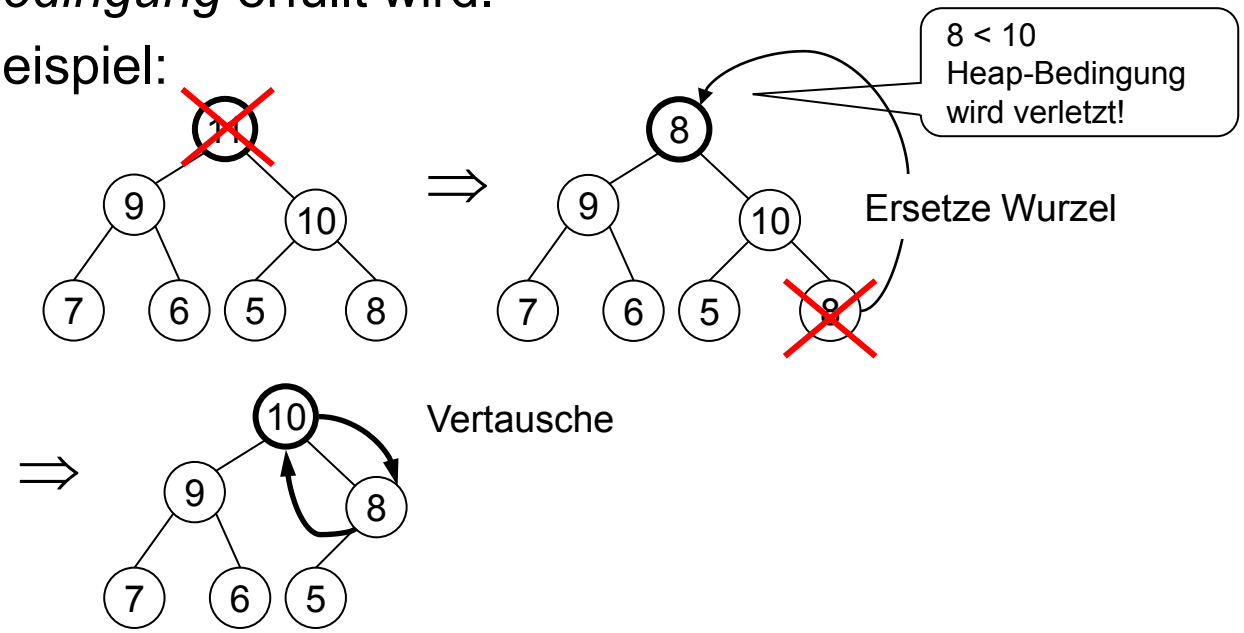
Einfügen

- Algorithmus
  - Füge neuen Knoten K_N am Ende des Feldes ein
  - Vertausche K_N solange mit den Vätern bis die *Heap-Bedingung* erfüllt ist. D.h. Der neue Knoten wandert gegebenenfalls in Richtung Wurzel
  - Beispiel:



Löschen

- Algorithmus
  - Ersetze Wurzel mit dem äußersten rechten Blatt
  - Die neue Wurzel wird solange im Baum abgesenkt (d.h. mit dem größeren Kind vertauscht) bis die *Heap-Bedingung* erfüllt wird.
  - Beispiel:



Realisierung mittels Zeiger

FHV

- Die sequentielle Realisation hat den Vorteil, dass alle Operationen sehr leicht realisierbar sind.
- Für Fälle, bei denen z.B.
  - die Anzahl der benötigten Einträge beliebig ist
  - man verschiedene PQueues zusammenfügen will

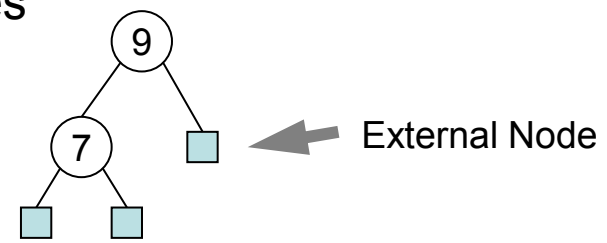
  ‚wird eine andere Datenstruktur benötigt ⇒ Verwendung von Zeiger

  Die Realisierung eines kompletten binären Baums erfordert zusätzlichen Aufwand, der für die Realisierung eines Heaps keine Vorteile bringt!

  ⇒ Abschwächen der geforderten Baumeigenschaften *(Height-biased leftist trees)*

Height-biased leftist tree

- Für die Definition des *HBLT* wird ein neuer Knotentyp eingeführt ⇒ Externe Knoten
  - Externe Knoten sind imaginäre Knoten an der Position eines fehlenden Kindes

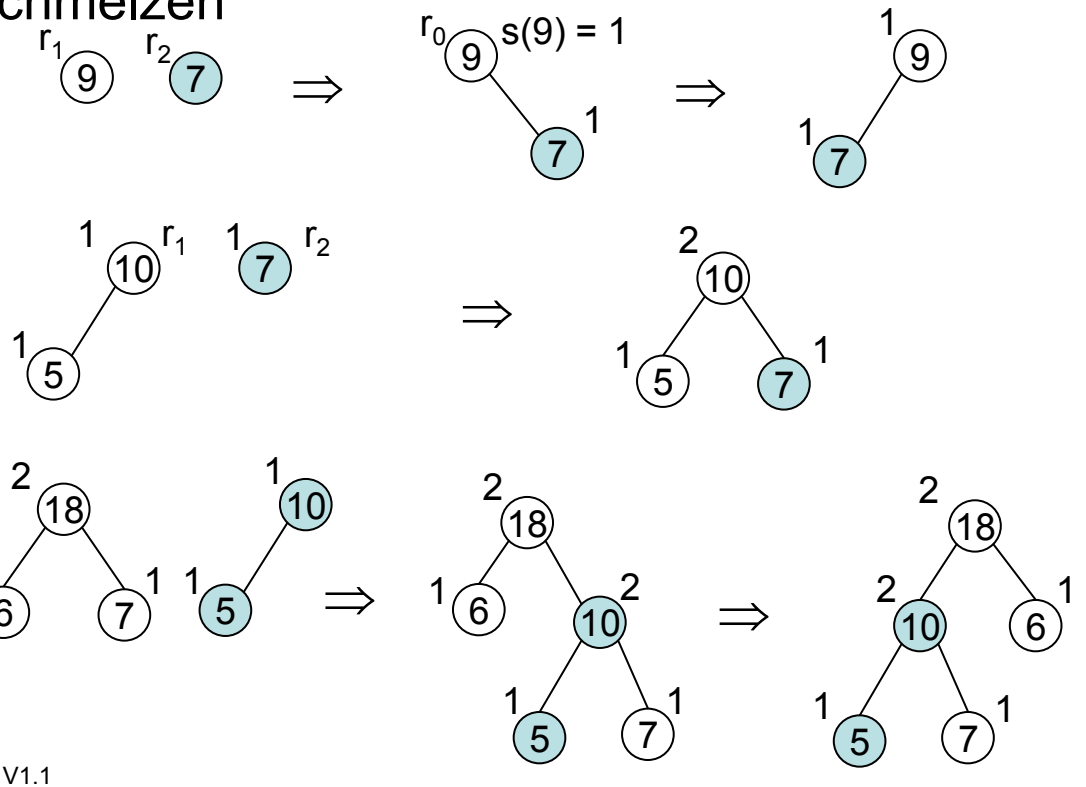


- s(x) bezeichnet die Länge des kürzesten Pfades vom Knoten X zu einem externen Knoten im Baum

  D.h. s(⑦) = 1
- Für einen Knoten ergibt sich *s(K)* aus
  min [s(K.left), s(K.right)] + 1
- Ein Binärer Baum ist ein ***HBLT*****,** genau dann wenn für jeden *internen Knoten K* **s(K.left) ≥ s(K.right)** ist.

Einfügen und Löschen

- Einfügen und Löschen in *Height-biased Leftist Trees* können auf das Verschmelzen von zwei HBLTs zurückgeführt werden.
- Verschmelzen



Verschmelzungsalgorithmus

FHV

- Wurzel *r1* mit größerem Schlüsselwert als *r2* wird die neue Wurzel *r0*
- Rechter Teilbaum von *r1* wird mit dem Baum, dessen Wurzel *r2* ist, verschmolzen
- Wird durch das Verschmelzen die HBLT-Bedingung verletzt, werden die Teilbäume vertauscht.
- Wiederhole die Schritte solange, bis der Baum wieder die HBLT Bedingung erfüllt