Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2013 / 2014

Vorlesung 6, Donnerstag 28. November 2013
(Hashing Kollisionsbehandlung, Prioritätswarteschlangen)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

[basiert auf der letztjährigen Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2" von Prof. Dr. Matthias Teschner]

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü4 (GeoNames)
- Hashing
  - Nochmal die Pointe von universellem Hashing
  - Behandlung von Kollisionen
- Prioritätswarteschlangen (Englisch: priority queues)
  - Ebenfalls eine Datenstruktur, die man sehr häufig braucht
  - Operationen: insert, getMin, deleteMin, changeKey
  - Anwendungsbeispiel
  - Benutzung in C++ und in Java
  - Implementierung mittels eines binären Heaps
  - Übungsblatt: Implementierung einer PriorityQuery mit den genannten Methoden

Erfahrungen mit dem Ü4 (GeoNames)

UNI FREIBURG

- Zusammenfassung / Auszüge   Stand 27. November
  - Programmieren hat Spaß gemacht
  - Jenkins ist einfach nur frustrierend, weil es nie funktioniert.

    … Bitte im Forum fragen
  - Mindestens 1 Stunde nur für Stylechecker

    … Probieren Sie mal „astyle“ http://astyle.sourceforge.net/

    astyle --style=google --indent=spaces=2 myUglyCode.cpp

    (Google style seit Version 2.04 (November 2013) verfügbar)
  - Beide Verfahren nahezu gleichschnell, teilweise sortieren sogar schneller … siehe nächste Folie
  - Viel zu viel Zeit verbraucht für: Einlesen, Operator überladen, vector, map, iterators, etc. … einiges davon wurde in Vorlesung erklärt
  - „Tests vorgeben“ … Tests entwerfen ist ein wichtiger Teil vom Verstehen des Problems

C++ Musterlösung auf unseren Servern

UNI FREIBURG

	Luckyluke Xeon E5-2665 2.40GHz 32GB RAM	**Ororea** Xeon X7460 2.66GHz 256GB RAM	**Marsupilami** Xeon X5365 3.00GHz 32GB RAM	**Joe** Xeon X5690 3.47GHz 24GB RAM
198 936 cities				
std::sort	163 ms	**235 ms**	**215 ms**	120 ms
std::unordered_map	**135 ms**	266 ms	255 ms	**118 ms**
std::map	205 ms	271 ms	**216 ms**	166 ms
3 239 960 cities				
std::sort	3693.28 ms	7275 ms	5660 ms	2778 ms
std::unordered_map	**3073.16 ms**	**6494 ms**	**4328 ms**	**2376 ms**
std::map	4352.3 ms	**6461 ms**	4659 ms	3396 ms

Universelles Hashing Reprise 1/2

UNI FREIBURG

- Nochmal die Pointe
  - Keine Hashfunktion ist gut für **alle** Schlüsselmengen
    - das kann gar nicht gehen, weil ein großes Universum auf einen kleinen Bereich abgebildet wird
  - Für **zufällige** Schlüsselmengen tun es auch einfache Hashfunktionen wie $h(x) = x \bmod m$
    - dann sorgen die zufälligen Schlüssel dafür, dass es sich gut verteilt
  - Wenn man für **jede** Schlüsselmenge gute Hashfunktionen finden will, braucht man universelles Hashing
    - dann ist aber, für eine feste Schlüsselmenge, nicht jede Hashfunktion gut, sondern nur viele / die meisten

Universelles Hashing Reprise 2/2

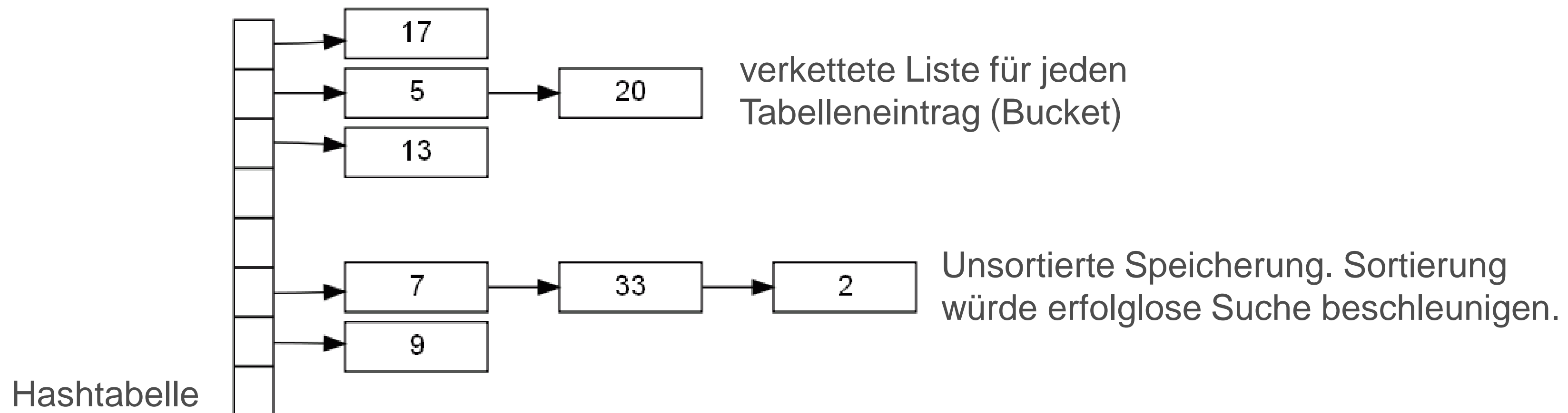
- Rehash
  - Auch mit universellem Hashing kann man mal eine schlechte Hashfunktion erwischen, wenn auch unwahrscheinlich
  - Das kann man aber leicht feststellen, in dem man die maximale Bucketgröße misst
  - Wenn die einen vorgegeben Wert überschreit macht man einen sogenannten **Rehash**
    - Neue Hashtabelle mit neuer zufälliger Hashfunktion
    - Elemente von der alten in die neue Tabelle kopieren
    - Das ist teuer, wird aber selten passieren
    - Deshalb durschnittliche Kosten gering
    - Siehe Thema amortisierte Analye in der nächsten Vorlesung

Behandlung von Hashkollisionen

Verkettung

UNI FREIBURG

- Hashtabelleneintrag ist Kopf einer verketteten Liste
- kollidierende Schlüssel werden gemeinsam in die Liste eines Eintrags (Bucket) sortiert oder am Ende eingefügt



- Operationen in O(1) bei sinnvoller Wahl von Tabellengröße und Hashfunktion (wenig Elemente pro Bucket)
- schlechtester Fall O(n), z. B. bei Tabellengröße 1
- dynamisch, variable Zahl von Elementen möglich

Offene Hashverfahren

UNI FREIBURG

- Für kollidierende Schlüssel (Überläufer) wird ein freier Eintrag in der Tabelle gesucht.
- statisch, Zahl der Elemente fest
- Sondierungsreihenfolge bestimmt für jeden Schlüssel, in welcher Reihenfolge alle Hashtabelleneinträge auf einen freien Platz durchsucht werden.
  - Ist ein Eintrag belegt, kann beispielsweise iterativ der nächstfolgende Tabelleneintrag geprüft werden. Wird ein freier Eintrag gefunden, wird das Element eingetragen.
  - Wird bei der Suche ein Element nicht am entsprechenden Tabelleneintrag gefunden, obwohl der Eintrag belegt ist, muss ebenfalls die definierte Sondierungsreihenfolge abgearbeitet werden, bis das Element oder ein freier Eintrag gefunden wurde.

Prinzip

UNI FREIBURG

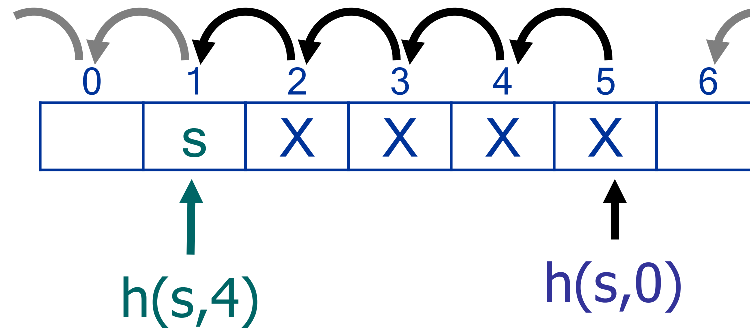
- h(s) – Hashfunktion für Schlüssel s
- g(s,j) – Sondierungsfunktion für Schlüssel s mit $0 \leq j \leq m-1$
- (h(s) – g(s,j)) mod m – Sondierungsreihenfolge, d. h. Permutation von < 0, 1, ..., m-1 >
- Einfügen

```
j:=0;
while ( t[(h(s) - g(s,j)) mod m] != frei ) j++;
t[(h(s) - g(s,j)) mod m] := s;
```

- Suchen

```
j:=0;
while (t[(h(s) - g(s,j)) mod m] != frei &&
          t[(h(s) - g(s,j)) mod m] != s) j++;
if (t[(h(s) - g(s,j)) mod m] == s) return true;
   else return false;
```

Lineares Sondieren



- g(s,j) = j → Hashfunktion (h(s) – j) mod m
- führt zu Sondierungsreihenfolge
  h(s), h(s)-1, h(s)-2, …, 0, m-1, …, h(s)+1
- einfach
- führt aber zu primärer Häufung (primary clustering)
- Behandlung einer Hashkollision erhöht die Wahrscheinlichkeit einer Hashkollision in benachbarten Tabelleneinträgen

Beispiel

UNI FREIBURG

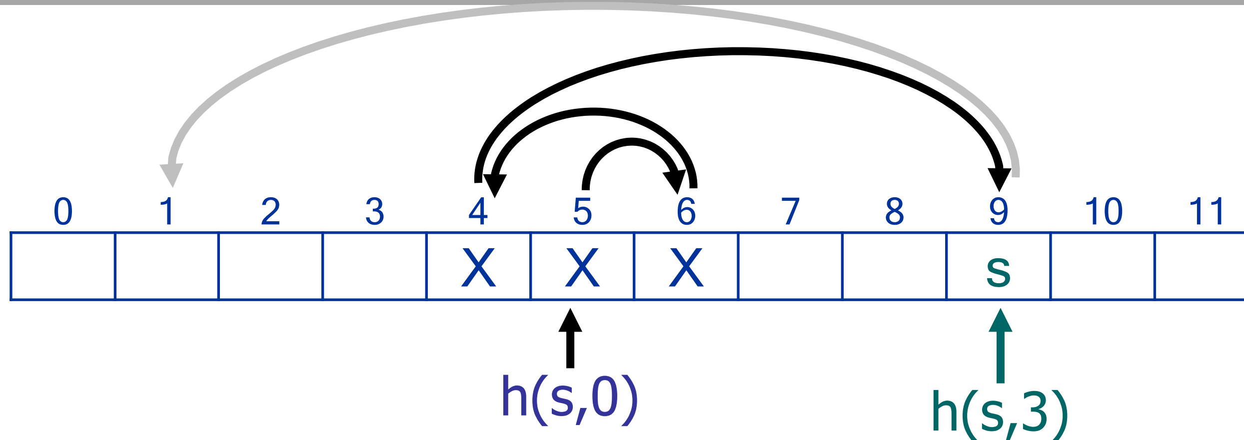
- Schlüssel: {12, 53, 5, 15, 2, 19},
  Hashfunktion: $h(s,j) = (s \bmod 7 - j) \bmod 7$

- t.insert(12); h(12,0) → 5;
- t.insert(53); h(53,0) → 4;
- t.insert(5); h(5,0) → 5;
  h(5,1) → 4; h(5,2) → 3;
- t.insert(15); h(15,0) → 1;
- t.insert(2); h(2,0) → 2;
- t.insert(19);
  h(19,0) → 5; h(19,1) → 4; h(19,2) → 3;
  h(19,3) → 2; h(19,4) → 1; h(19,5) → 0;

0	1	2	3	4	5	6
					12	
				53	12	
			5	53	12	
	15		5	53	12	
	15	2	5	53	12	
19	15	2	5	53	12	

Quadratisches Sondieren

UNI FREIBURG



- Motivation: Vermeidung von lokalen Häufungen
- $g(s,j) = (-1)^j \lceil j / 2 \rceil^2$
- führt zu Sondierungsreihenfolge
  h(s), h(s)+1, h(s)-1, h(s)+4, h(s)-4, h(s)+9, h(s)-9, …

  $h(s,j) = h(s) - g(s,j)$
- Wenn m eine Primzahl der Form $4 \cdot k+3$ ist, ist die Sondierungsreihenfolge eine Permutation der Indizes der Hashtabelle
- alternativ auch $h(s,j) = (h(s) - c1 \cdot j - c2 \cdot j^2) \bmod m$

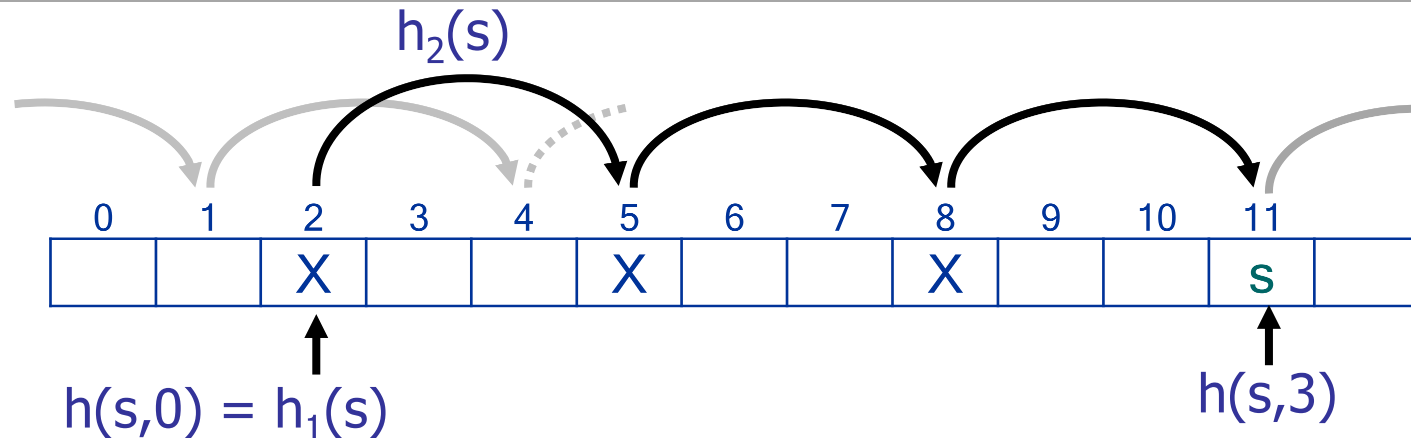
  c1, c2 – Konstanten
- Problem der sekundären Häufung
  - keine lokale Häufung mehr, allerdings durchlaufen Schlüssel mit gleichem Hashwert immer die gleiche Ausweichsequenz

Uniformes Sondieren

UNI FREIBURG

- Motivation: Funktion g(s,j) berücksichtigt beim linearen und quadratischen Sondieren lediglich den Schritt j. Die Sondierungsreihenfolge ist vom Schlüssel unabhängig.
- Uniformes Sondieren berechnet die Folge g(s,j) von Permutationen aller möglichen Indizes in Abhängigkeit vom Schlüssel s
- Vorteil: Häufung wird vermieden, da unterschiedliche Schlüssel mit gleichem Hashwert zu unterschiedlichen Sondierungsreihenfolgen führen
- Nachteil: schwierige praktische Realisierung

Double Hashing



- Motivation: Berücksichtung des Schlüssels s in der Sondierungsreihenfolge
- Verwendung zweier unabhängiger Hashfunktionen h_1, h_2
- $h(s,j) = (h_1(s) + j \cdot h_2(s)) \bmod m$
- Folge: $h_1(s)$, $h_1(s) + 1h_2(s)$, $h_1(s) + 2h_2(s)$, $h_1(s) + 3h_2(s)$, …
- funktioniert praktisch sehr gut
- Approximation des uniformen Sondierens

Beispiel

- $h_1(s) = s \bmod 7$
- $h_2(s) = 1 + (s \bmod 5)$
- $h(s,j) = (h_1(s) + j \cdot h_2(s)) \bmod 7$

s	10	19	31	22	14	16
h_1 (s)	3	5	3	1	0	2
h_2 (s)	1	5	2	3	5	2

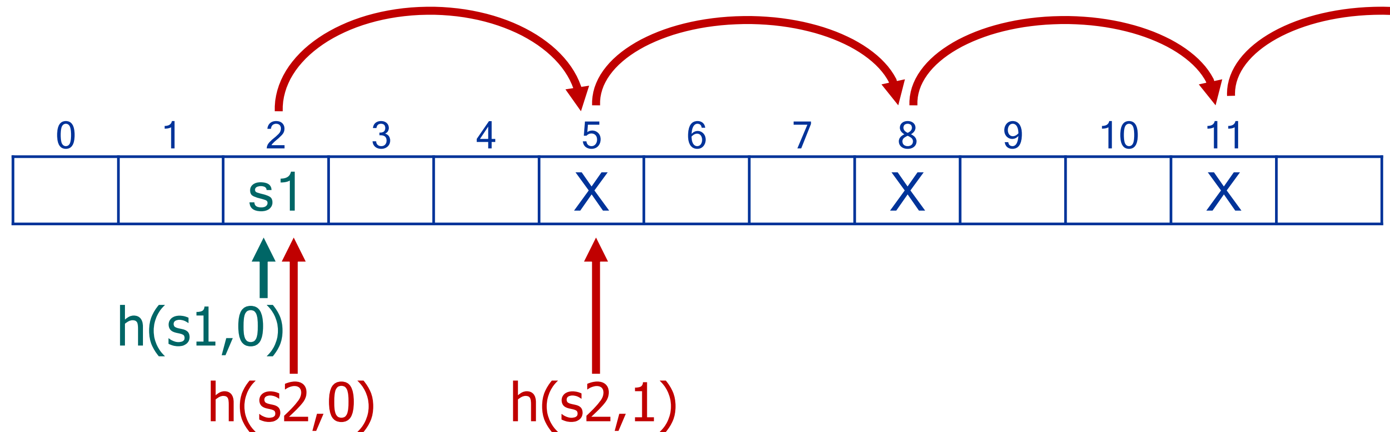
Effizienz beruht auf
h1 (s) != h2 (s) → Sondierungs-
reihenfolge abhängig von s

Verbesserung des Double Hashing nach Brent

UNI FREIBURG

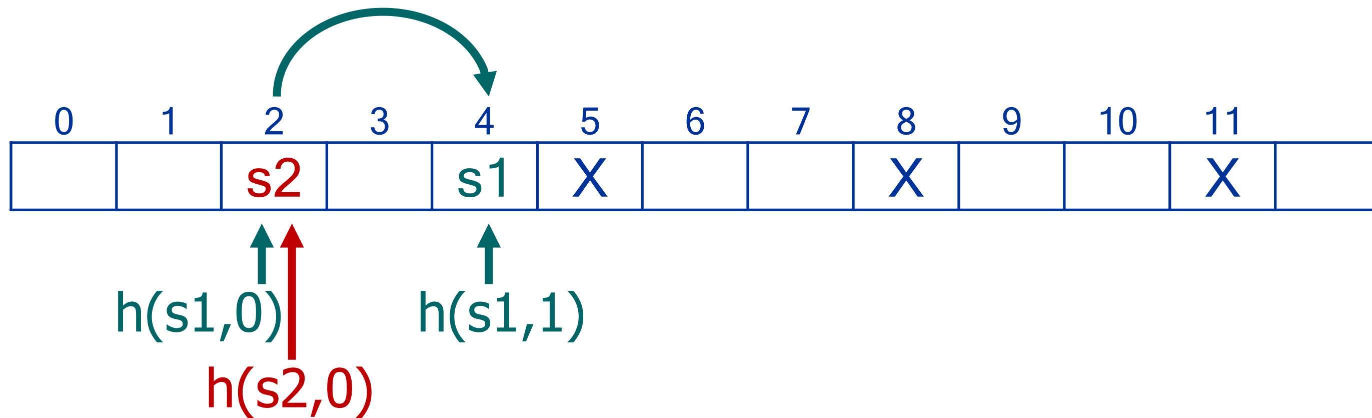
- Motivation: Da unterschiedliche Schlüssel unterschiedliche Sondierungsreihenfolgen haben, hat die Reihenfolge des Einfügens der Schlüssel Einfluss auf die Effizienz der erfolgreichen Suche

Verbesserung des Double Hashing nach Brent



- Beispiel:
  - s1 wird an Position p1 = h (s1, 0) eingefügt.
  - s2 liefert ebenfalls p1 = h (s2, 0).
  - h (s2, 1..n) sind ebenfalls belegt.
  - s2 wird an Position h (s2, n+1) eingetragen, was bei der Suche recht ineffizient ist.

Verbesserung des Double Hashing nach Brent



- Brents Idee: Teste, ob h (s1, 1) frei ist.
- Falls ja, wird s1 von Position h (s1,0) nach Position h (s1,1) verschoben und s2 an Position h (s1, 0) eingetragen.

Ordered Hashing

UNI FREIBURG

- Motivation: Kollidierende Elemente werden in sortierter Reihenfolge in der Hashtabelle abgelegt.
  - Dadurch kann bei erfolgloser Suche von Elementen in Kombination mit lin. Sondierung oder bei double hashing früher abgebrochen werden, da hier einzelne Sondierungsschritte feste Länge haben.
- Realisierung:
  - Bei einer Kollision werden beide Schlüssel verglichen.
  - Der kleinere Schlüssel wird abgelegt.
  - Für den größeren Schlüssel wird eine neue Position gemäß Sondierungsreihenfolge gesucht.
- Beispiel:
  - 12 ist an Position $p1 = h(12, 0)$ gespeichert.
  - 5 liefert $p1 = h(5, 0)$. $5 < 12 \rightarrow 5$ wird an Position $p1$ eingetragen.
  - Für 12 werden die Positionen $h(12, 1 \ldots)$ weiter getestet.

Robin-Hood-Hashing

UNI FREIBURG

- Motivation: Angleichung der Länge der Sondierungsfolgen für alle Elemente. Gesamtkosten bleiben gleich, aber gerecht verteilt. Führt zu annähernd gleichen Suchzeiten für alle Elemente.
- Realisierung:
  - Bei einer Kollision zweier Schlüssel s1 und s2 mit $p1 = h(s1, j1) = h(s2, j2)$ werden j1 und j2 verglichen.
  - Schlüssel mit größerer Länge der Sondierungsfolge wird an Position p1 gespeichert. Der andere Schlüssel erhält neue Position.
- Beispiel:
  - 12 ist an Position $p1 = h(12, 7)$ gespeichert.
  - 5 liefert $p1 = h(5, 0)$. $0<7 \rightarrow$ 12 bleibt an Position p1.
  - Für 5 werden die Positionen $h(5, 1 \ldots)$ weiter getestet.

Implementierung Einfügen / Entfernen

- Entfernen von Elementen kann problematisch sein.
  - Schlüssel s1 wird an Position p1 eingefügt.
  - Schlüssel s2 liefert den gleichen Hashwert, wird aber durch das Sondieren an Position p2 eingefügt.
  - Wird Schlüssel s1 entfernt, kann s2 nicht wiedergefunden werden.
- Lösung.
  - Entfernen: Elemente werden nicht entfernt, sondern als gelöscht markiert.
  - Einfügen: Als gelöscht markierte Einträge werden überschrieben.

Zusammenfassung Kollisionsbehandlung

UNI FREIBURG

- Verkettung (dynamisch, Zahl der Elemente variabel)
  - kollidierende Schlüssel werden in Liste gespeichert
- Offene Hashverfahren (statisch, Zahl der Elemente fest)
  - Bestimmung einer Ausweichsequenz (Sondierungsreihenfolge), Permutation aller Hashwerte (Indizes der Hashtabelle)
  - lineares, quadratisches Sondieren: einfach, führt zu Häufungen, da Sondierungsreihenfolge vom Schlüssel unabhängig
  - uniformes Sondieren, double hashing: unterschiedliche Sondierungsreihenfolgen für unterschiedliche Schlüssel, vermeidet Häufungen von Elementen
- Effizienzsteigerung der Suche durch Umsortieren von Elementen beim Einfügen (Brent, Ordered Hashing)

Zusammenfassung Hashverfahren

UNI FREIBURG

- effiziente Wörterbuchoperationen: Einfügen, Suchen, Entfernen
- direkter Zugriff auf Elemente einer Hashtabelle
- Berechnung der Position in der Hashtabelle durch Hashfunktion (Hashwert)
- gleiche Hashwerte für unterschiedliche Schlüssel führen zu Hashkollisionen
- Hashfunktion, Größe der Hashtabelle und Strategie zur Vermeidung von Hashkollisionen beeinflussen die Effizienz der Datenstruktur.

Prioritätswarteschlangen

Prioritätswarteschlangen 1/4

UNI FREIBURG

- Definition
  - Eine **Prioritätswarteschlange** (PW) speichert eine Menge von Elementen, von denen jedes einen Schlüssel hat.
    - Also Key-Value Paare, wie in einer Map auch
  - Es gibt eine totale Ordnung ≤ auf den Keys
  - Die PW unterstützt auf dieser Menge folgende Operationen
    - insert(key, value): füge das gegebene Element ein
    - getMin(): liefert das Element mit dem kleinsten Key
    - deleteMin(): entferne das Element mit dem kleinsten Key
  - Und manchmal auch noch
    - changeKey(item, key): ändere Key des gegebenen Elementes
    - remove(item): entferne das gegebene Element

Prioritätswarteschlangen 2/4

UNI FREIBURG

- Mehrere Elemente mit dem gleichen Key
  - Kein Problem, und für viele Anwendungen nötig
  - Falls es mehrere Elemente mit dem kleinsten Key gibt:
    - gibt getMin irgend eines davon zurück
    - und deleteMin löscht eben dieses
- Argument der Operationen changeKey und remove
  - Eine PW erlaubt **keinen schnellen** Zugriff auf ein beliebiges Element
  - Deshalb geben (bei unserer Implementierung) insert und getMin eine Referenz auf das entsprechende Element zurück
  - Mit dieser Referenz kann man dann später über changeKey bzw. remove den Schlüssel ändern / das Element entfernen
  - Dafür muss jedes Element intern seine aktuelle Position im Heap speichern.

Prioritätswarteschlangen 3/4

UNI FREIBURG

- Benutzung in **Java**
  - Im Vorspann: `import java.util.PriorityQueue;`
  - Element-Typ unterscheidet nicht zwischen Key und Value

    `PriorityQueue<T> pq;`
  - Defaultmäßig wird die Ordnung $\leq$ auf T genommen
    - eigene Ordnung über einen Comparator, wie bei sort
    - siehe unseren Code zum Sortieren mit einer PW
  - Operationen: insert = add, getMin = peek, deleteMin = poll
  - Die Operation changeKey gibt es nicht
  - Dafür gibt es remove = entferne ein gegebenes Element (Achtung: Kosten O(N))
  - Mit remove und insert kann man ein changeKey simulieren !

Prioritätswarteschlangen 4/4

UNI FREIBURG

- Benutzung in **C++**
  - Im Vorspann: #include <queue>;
  - Element-Typ unterscheidet nicht zwischen Key und Value

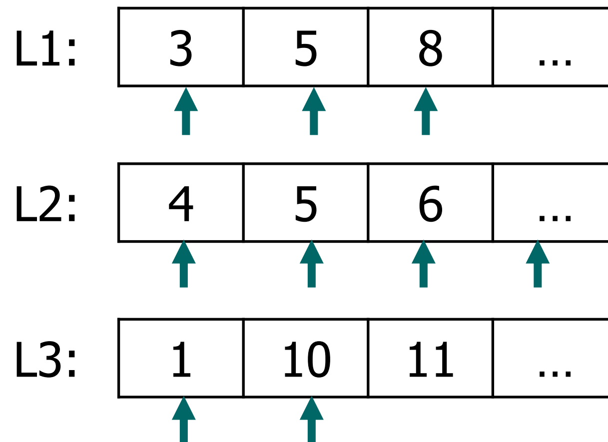
    std::priority_queue<T> pq;
  - Es wird die Ordnung ≥ auf T genommen, und nicht ≤
  - Beliebige Vergleichsfunktion wie bei std::sort
  - Operationen: insert = push, getMin = top, deleteMin = pop
  - Es gibt kein changeKey und auch kein beliebiges remove

    (aus Effizienzgründen: es macht die Implementierung komplexer, aber viele Anwendungen brauchen es nicht)

PWs — Anwendungen 1/2

UNI FREIBURG

- Anwendungsbeispiel 1
  - Berechnung der Vereinigungsmenge von k sortierten Listen (sogenannter multi-way merge oder k-way merge)
  - Beispiel k=3:



R: 1, 3, 4, 5, 5, 6, ...

  - Laufzeit (N: Summe der Listenlängen); k: Anzahl der Listen)
    - Trivial: $\Theta(N \cdot k)$, da Minmumberechnung $\Theta(k)$
    - Mit PW: $\Theta(N \cdot \log k)$, da Minmumberechnung $\Theta(\log k)$

PWs — Anwendungen 2/2

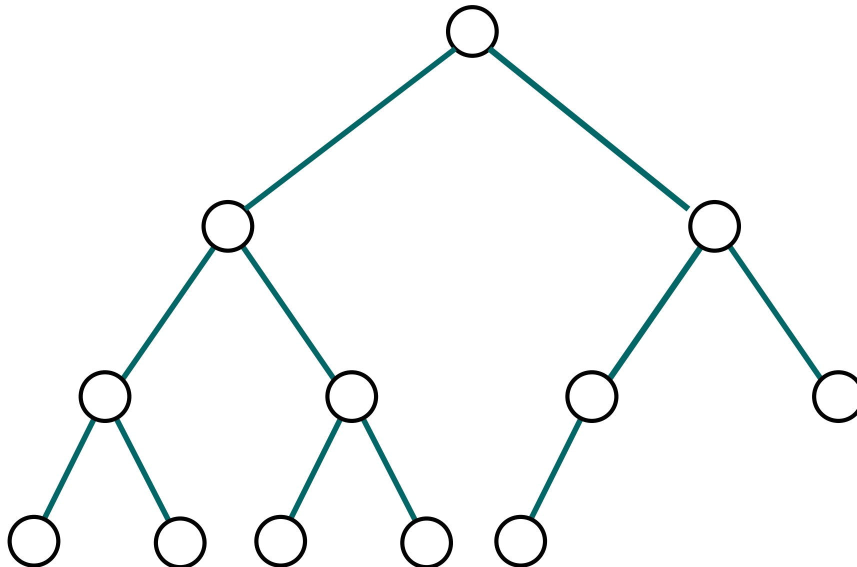
UNI FREIBURG

- Anwendungsbeispiel 2
  - Zum Beispiel für Dijkstra's Algorithmus zur Berechnung kürzester Wege → spätere Vorlesung
  - Unter anderem kann man damit auch einfach Sortieren

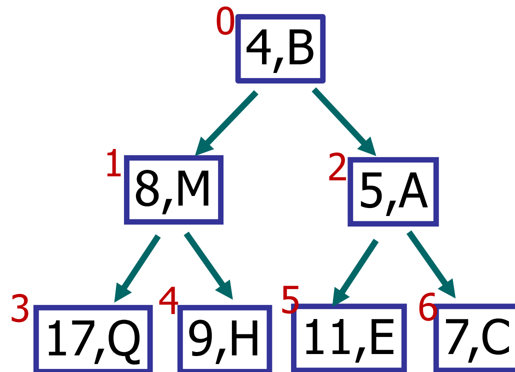
Implementierung 1/7

UNI FREIBURG

- Grundidee
  - Elemente in einem binären Heap speichern
  - Wiederholung aus der 1. Vorlesung (HeapSort):
    - vollständiger binärer Baum (bis evtl. "unten rechts")
    - es gilt die Heap-Eigenschaft = der Key jedes Knotens ist $\leq$ die Keys von den beiden Kindern



Implementierung 2/7

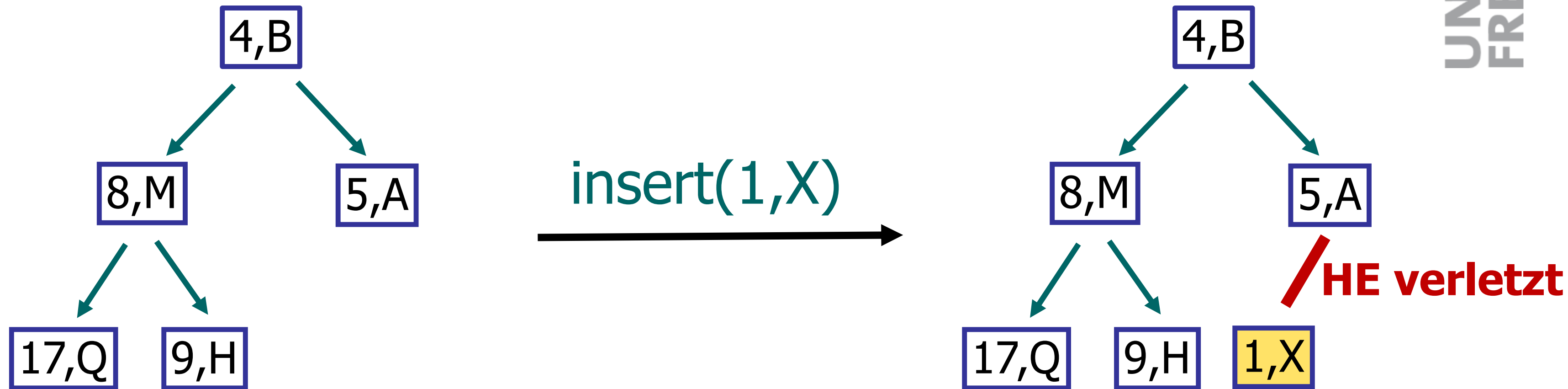


0	1	2	3	4	5	6
4,B	8,M	5,A	17,Q	9,H	11,E	7,C

- Wie speichert man einen binären Heap
  - Ebenfalls bekannt aus Vorlesung 1 – jetzt aber mit Indizierung ab 0, wie in der Musterlösung
  - Wir numerieren die Knoten von oben nach unten und links nach rechts durch, beginnend mit **0**
  - Dann sind die Kinder von Knoten i die Knoten 2i+1 und 2i+2
  - Und der Elternknoten von einem Knoten i ist floor((i-1)/2)
  - Elemente stehen dann einfach in einem Array:

```
ArrayList<PriorityQueueItem> heap;       // Java.
std::vector<PriorityQueueItem> heap;     // C++.
```

Implementierung 3/7



- Einfügen eines Elementes (insert)
  - Erstmal hinzufügen am Ende des Arrays

    heap.add(keyValuePair); // Java.

    heap.push_back(keyValuePair); // C++.
  - Danach kann die Heapeigenschaft (HE) nach oben verletzt sein

    ... aber nur genau an dieser (letzten) Position !
  - Wiederherstellung der HE → spätere Folie

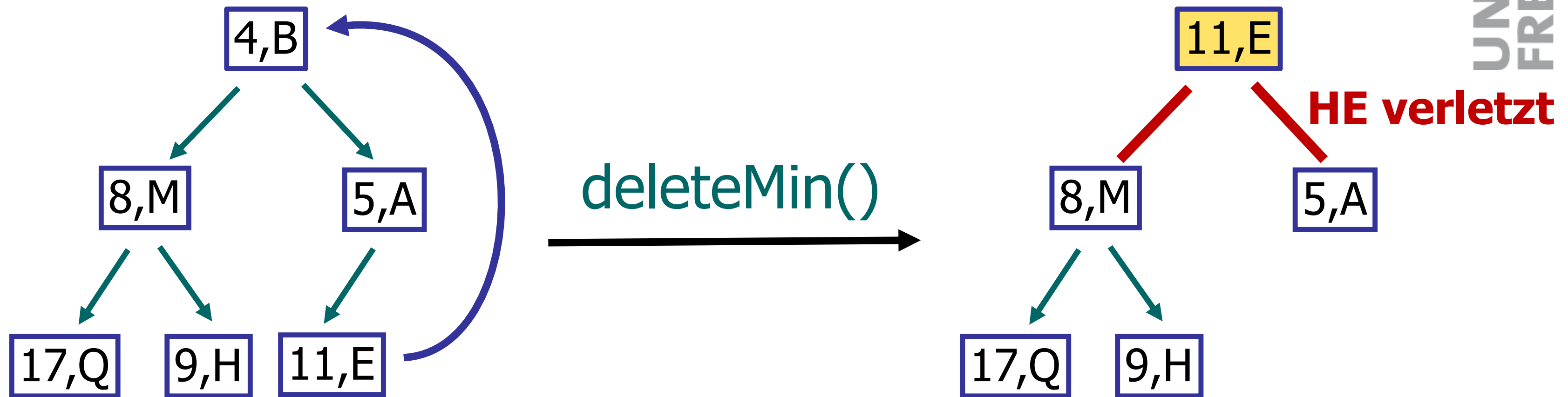
Implementierung 4/7

- Rückgabe des Elem. mit kleinstem Key (getMin)
  - Einfach das oberste Element zurückgeben

```
return heap.get(0);   // Java.
return heap[0];   // C++.
```

  - Achtung falls Heap leer, dann null zurückgeben

Implementierung 5/7



UNI FREIBURG

- Löschen des Elem. mit kleinstem Key (deleteMin)
  - Einfach das Element von der letzten Position an die erste Stelle setzen (falls heap nicht leer)

    heap.get(0) = heap.remove(heap.size() - 1); // Java.

    heap[0] = heap.back(); heap.pop_back(); // C++.
  - Danach kann die Heapeigenschaft (HE) nach unten verletzt sein

    ... aber wieder nur genau an dieser (ersten) Position !
  - Wiederherstellung der HE → spätere Folie

Implementierung 6/7

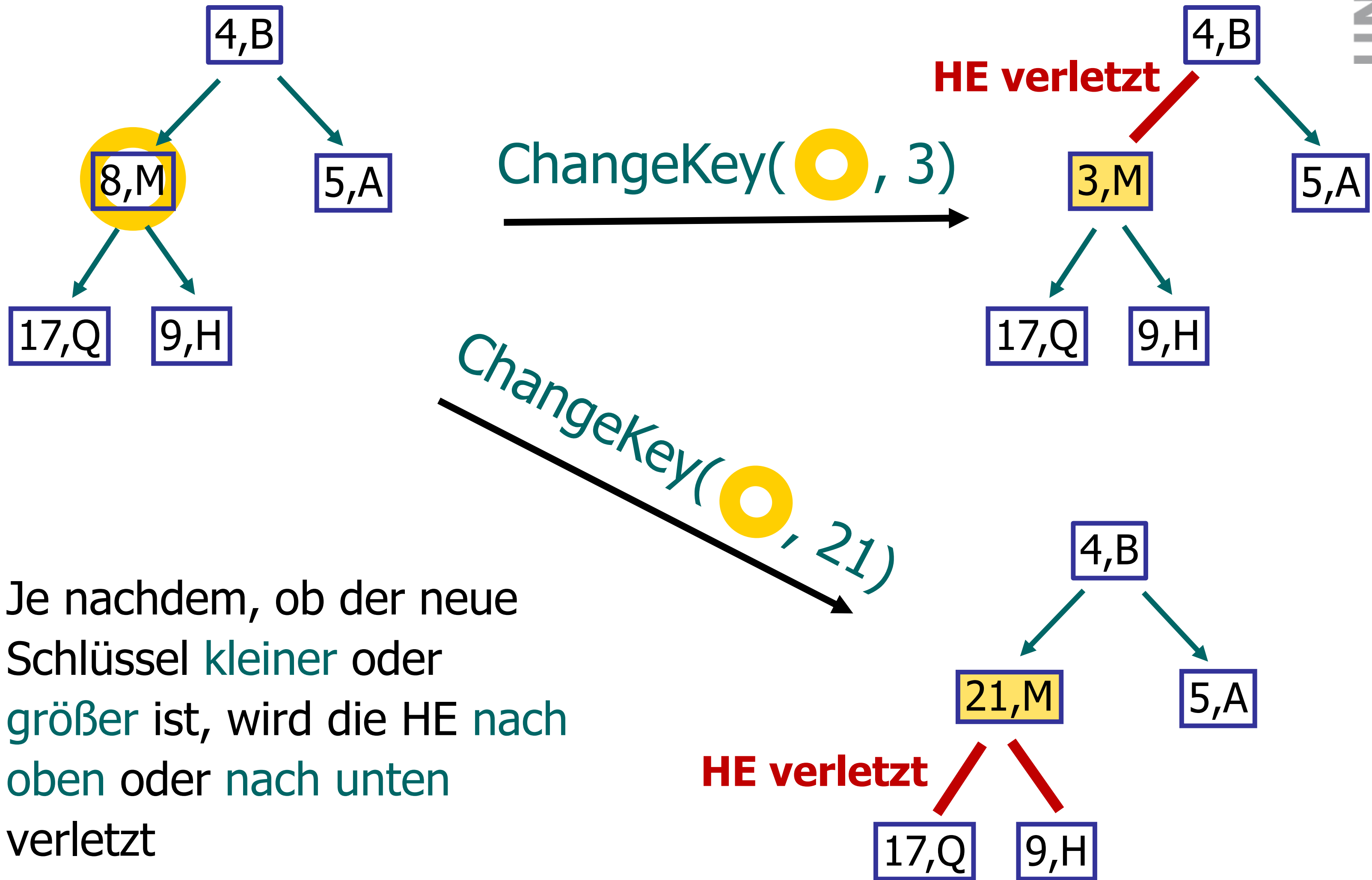
- Ändern eines Schlüssels (changeKey)
  - Element (pqItem) wurde als Argument übergeben !
  - Dann einfach den Schlüssel ändern

    pqItem.key = newKey;
  - Danach kann die Heapeigenschaft (HE) nach oben oder unten verletzt sein

    ... aber wieder nur genau an dieser Position !
  - Wiederherstellung der HE → spätere Folie
  - Jedes pqItem muss also seine Position kennen → dito

UNI FREIBURG

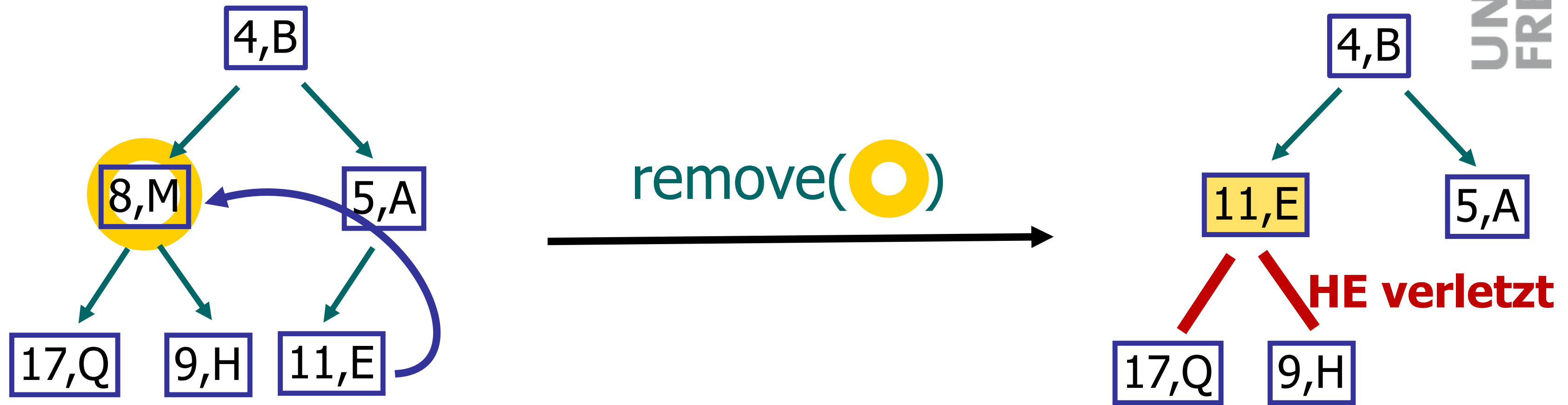
Implementierung 6/7



– Je nachdem, ob der neue Schlüssel kleiner oder größer ist, wird die HE nach oben oder nach unten verletzt

Implementierung 7/7

UNI FREIBURG



- Entfernen eines Elementes (remove)
  - Element (pqItem) wurde als Argument übergeben !
  - Dann einfach das Element von der letzten Position an diese Stelle setzen
  - Danach kann die Heapeigenschaft (HE) nach oben oder nach unten verletzt sein

    ... aber wieder nur genau an dieser Position !
  - Wiederherstellung der HE → spätere Folie
  - Jedes pqItem muss also seine Position kennen → dito

Reparieren der Heapeigenschaft 1/4

UNI FREIBURG

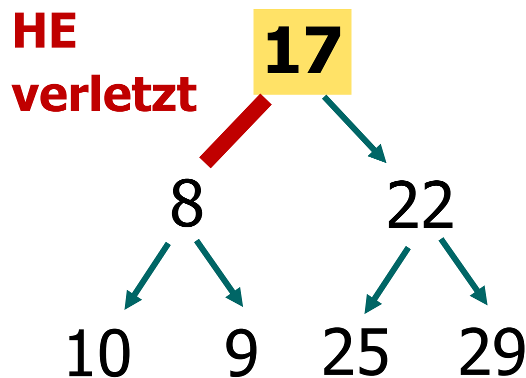
- Nach insert, deleteMin, changeKey, remove
  - ... kann die Heapeigenschaft (HE) verletzt sein
  - Aber nur an genau einer (bekannten) Position i
  - Die HE kann auf zwei Arten verletzt sein:
    - „nach unten": Schlüssel an Position i ist nicht ≤ der seiner Kinder
    - „nach oben": Schlüssel an Position i ist nicht ≥ der vom Elternkn.
  - Entsprechend brauchen wir zwei Reperaturmethoden

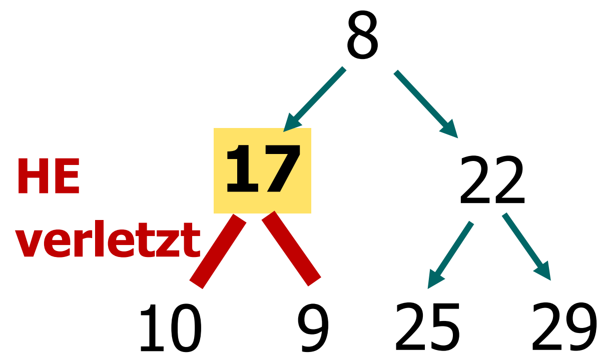
    repairHeapDownwards

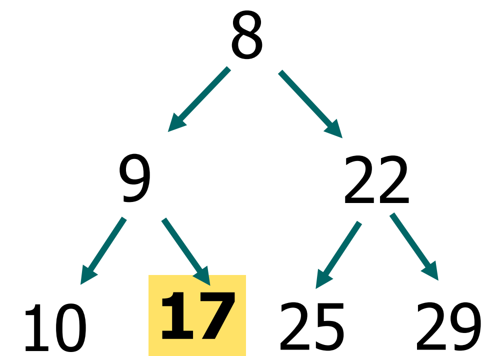
    repairHeapUpwards
  - Siehe die nächsten drei Folien ...

Reparieren der Heapeigenschaft 2/4

- Methode repairHeapDownwards
  - Nach unten „Durchsickern". Bekannt aus Vorlesung 1 und Übung 1
  - Knoten mit dem Kind tauschen, das den kleineren Key von den beiden Kindern hat
  - Jetzt ist bei diesem Kind evtl. die HE verletzt
  - In dem Fall einfach da dasselbe nochmal, usw.



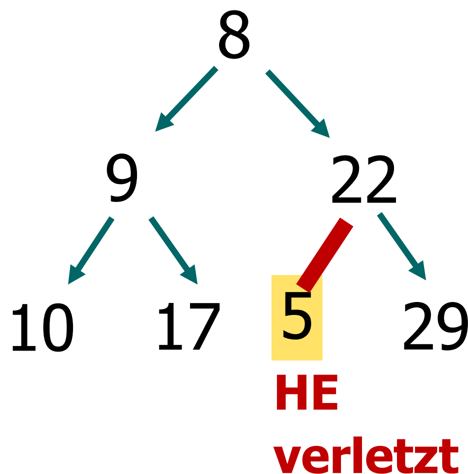


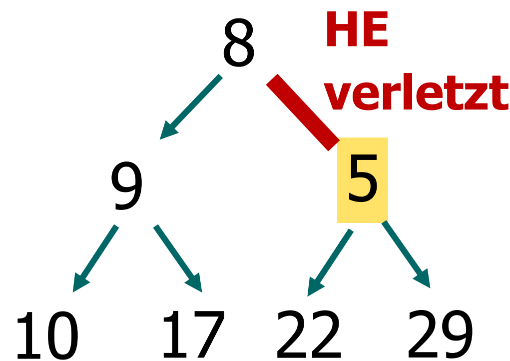


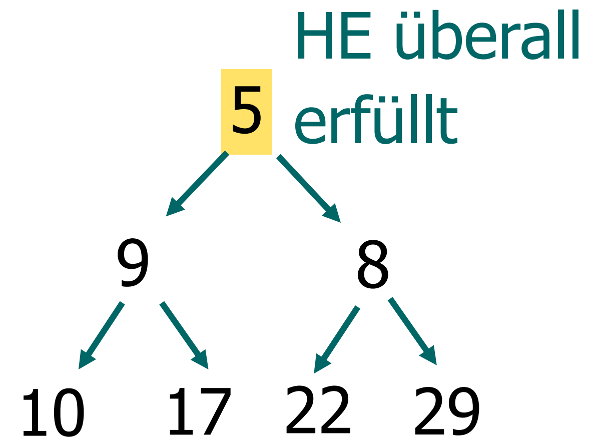
HE überall erfüllt

Reparieren der Heapeigenschaft 3/4

- Methode repairHeapUpwards
  - Knoten mit dem Elternknoten tauschen
  - Jetzt ist bei dem Elternknoten evtl. die HE verletzt
  - Wenn, dann Key < der von dessen Elternkn. ... warum?
  - In dem Fall einfach da dasselbe nochmal, usw.







Reparieren der Heapeigenschaft 4/4

UNI FREIBURG

- Index eines PriorityQueueItems
  - Achtung: für changeKey und remove muss ein PriorityQueueItem wissen, wo es im Heap steht

```
class PriorityQueueItem {
  int key;
  Object value;   // In C++, use a template T.
  int heapIndex;
}
```

  - Bei repairHeapDownwards und repairHeapUpwards beachten:

    Wann immer wir ein Element im Heap verschieben, muss der heapIndex des Elementes geupdated werden !

Komplexität 1/2

- Wiederholung Vorlesung 1
  - Ein vollständiger binärer Baum (bis evtl. "unten rechts") mit n Elementen hat Tiefe O(log n)
  - D.h. die Anzahl der Elemente auf einem Pfad von einer beliebigen Position im Heap nach oben zur Wurzel oder nach unten zu einem Blatt ist O(log n)
- Kosten (Laufzeit) für unsere diversen Methoden
  - Für repairHeapDownwards und repairHeapUpwards daher O(log n)
  - Für insert, deleteMin, changeKey, remove daher ebenfalls O(log n)
  - Für getMin offensichtlich O(1)

UNI FREIBURG

Komplexität 2/2

UNI FREIBURG

- Geht es noch besser?
  - Ja, mit sogenannten Fibonacci Heaps bekommt man

    getMin in Zeit O(1)

    insert in Zeit O(1)

    descreaseKey in amortisierter Zeit O(1)

    deleteMin in amortisierter Zeit O(log n)

    amortisiert = durchschnittlich ... nächste Vorlesung
  - In der Praxis ist der binäre heap aufgrund seiner Einfachheit aber schwer zu schlagen ... vor allem wenn die Anzahl der Elemente nicht riesig ist
  - Beachte: für $n = 2^{10} \approx 1.000$ ist $\log_2 n$ nur 10

    und selbst für $n = 2^{20} \approx 1.000.000$ ist $\log_2 n$ nur 20

Literatur / Links

UNI FREIBURG

- Hashkollisionen
  - In Ottmann/Widmeyer: 4.3 „Offene Hashverfahren“
- Prioritätswarteschlangen
  - In Mehlhorn/Sanders:

    6 Priority Queues [einfache und fortgeschrittenere Varianten]
  - In Cormen/Leiserson/Rivest

    20 Binomial Heaps [gleich die fortgeschrittenere Variante]
  - In Wikipedia

    http://de.wikipedia.org/wiki/Vorrangwarteschlange

    http://en.wikipedia.org/wiki/Priority_queue
  - In C++ und in Java

    http://www.sgi.com/tech/stl/priority_queue.html

    http://download.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html