Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda: Kürzeste Wege, Heaps, Hashing

- **Heute:**
  - Kürzeste Wege: Dijkstra
  - Heaps: Binäre Min-Heaps
  - Hashing: Overflow und Open

- Nächste Woche: Vorrechnen (first-come first-served)
  - Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U4/
  - Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U4/

Übung: https://hu.berlin/algodat17

Vorlesung: https://hu.berlin/vl_algodat17

Dijkstra (Single Source Shortest Paths)

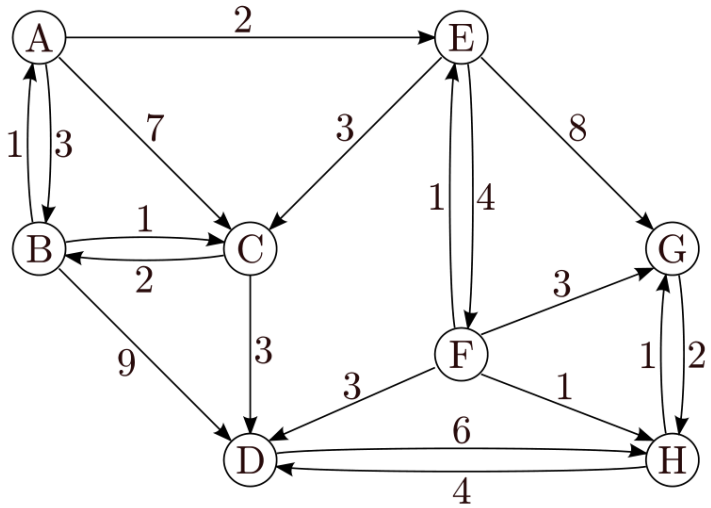
- Der Dijkstra Algorithmus löst das Problem der kürzesten Pfade bei einem einzigen Startknoten s auf einem gewichteten, gerichteten Graphen G=(V,E) für den Fall, dass alle Kantengewichte nichtnegativ sind.

- Idee: Greedy-Strategie
  Iterativ wird der am nächsten gelegene Knoten aufgenommen und aus der Menge der noch zu besuchenden Knoten L entfernt.

```
1.  G = (V, E);
2.  x : start_node;     # x∈V
3.  A : array_of_distances;
4.  ∀i: A[i]:= ∞;
5.  L := V;
6.  A[x] := 0;
7.  while L≠∅
8.    k := L.get_closest_node();
9.    L := L \ k;
10.   forall (k,f,w)∈E do
11.     if f∈L then
12.       new_dist := A[k]+w;
13.       if new_dist < A[f] then
14.         A[f] := new_dist;
15.       end if;
16.     end if;
17.   end for;
18. end while;
```

Dijkstra-Algorithmus

Gegeben sei ein gerichteter Graph mit 8 Knoten:



Bestimmen Sie den Abstand aller Knoten zum Knoten A. Führen Sie dafür den Dijkstra-Algorithmus aus der Vorlesung mit dem Startknoten A aus. Geben Sie die Zwischenergebnisse (d.h., die derzeit ermittelten Distanzen aller Knoten zu A) nach jedem Schritt an. Markieren Sie den aktuell betrachteten Knoten in jedem Schritt durch Umkreisen. Geben Sie außerdem den Inhalt der Priority Queue an, in der die als nächstes zu betrachtenden Knoten gespeichert werden.

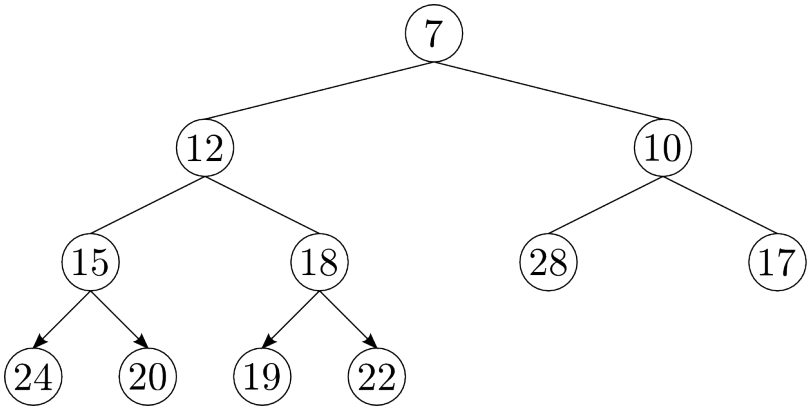
Schritt	A	B	C	D	E	F	G	H	Priority Queue
0	0	∞	∞	∞	∞	∞	∞	∞	(A)
1	**0**	3	7	∞	2	∞	∞	∞	(E),B,C
2	**0**	3	5	∞	**2**	6	10	∞	(B),C,F,G
3	**0**	**3**	4	12	**2**	6	10	∞	(C),F,G,D
4	**0**	**3**	**4**	7	**2**	6	10	∞	(F),D,G
5	**0**	**3**	**4**	7	**2**	**6**	9	7	(D),(H),G
6	**0**	**3**	**4**	**7**	**2**	**6**	9	7	(H),G
7	**0**	**3**	**4**	**7**	**2**	**6**	8	**7**	(G)
8	**0**	**3**	**4**	**7**	**2**	**6**	**8**	**7**	

Agenda: Kürzeste Wege, Heaps, Hashing

- **Heute:**
  - Kürzeste Wege: Dijkstra
  - **Heaps: Binäre Min-Heaps**
  - Hashing: Overflow und Open

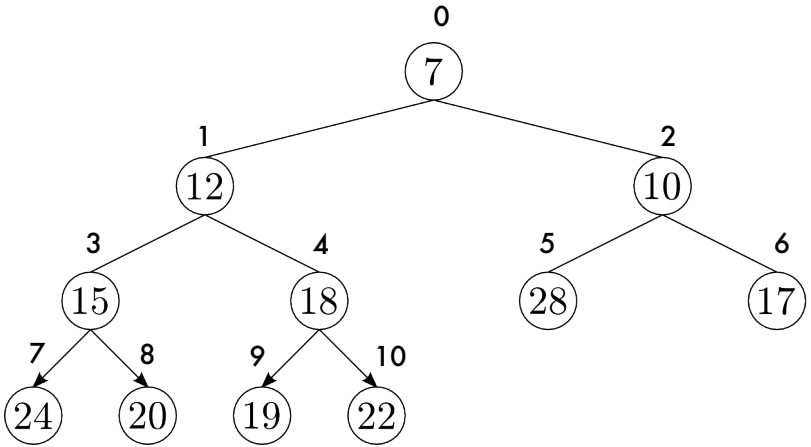
Heaps

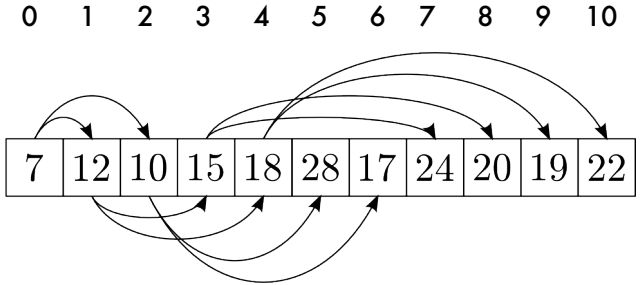
- Ein Heap ist eine auf Bäumen basierende Datenstruktur zum Speichern von Elementen, über deren Schlüssel eine totale Ordnung definiert ist
- Form Constraint: Der Baum ist fast vollständig
  - Alle Ebenen außer der untersten müssen vollständig gefüllt sein
  - In der letzten Ebene werden Elemente von links nach rechts aufgefüllt
- Heap Constraint: Bei einem Min-Heap (Max-Heap) sind die Schlüssel jedes Knotens kleiner (größer) als die Schlüssel seiner Kinder



Headgeordnete Arrays

- Heaps lassen sich als heapgeordnete Arrays repräsentieren.





- Wir nummerieren alle Knoten Ebenen-weise von links nach rechts. Die Zahl über dem Knoten entspricht dem Index im heapgeordneten Array.
- Ein Knoten hat die Kinder $left(i) = 2i + 1$ und $right(i) = 2i + 2$
- Der Vater eines Knotens steht an Index $\text{parent(i)} = \left\lfloor \frac{i-1}{2} \right\rfloor$

Programmieraufgabe: Min-Heap

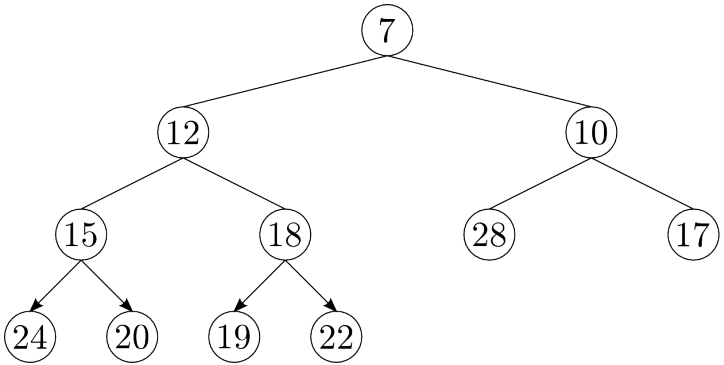
- Ein binärer Min-Heap der Größe n wird durch ein heapgeordnetes Array implementiert, welches maximal n Tupel *(key,value)* speichert. In jedem solchen Tupel ist *key* eine beliebige natürliche Zahl und *value* eine natürliche Zahl aus der Menge {0, . . . , n−1}.

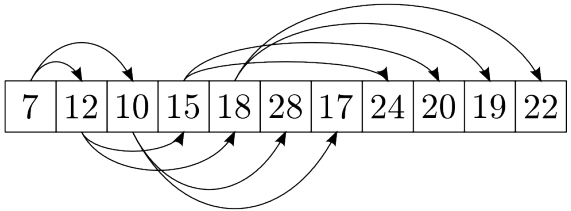
- Der *key* jedes Tupels soll dabei höchstens so groß sein wie der *key* seines linken und seines rechten Kindes (wenn vorhanden).

```
/// Das heapgeordnete Array
Entry[] heap;

public static class Entry {
        /// Schlüssel des Eintrags
        public int key;
        /// Wert des Eintrags
        public int value;
}


public void checkHeapConstraint(int i) {
    int left = 2*(i+1) - 1;
    int right = 2*(i+1);
    if (left < used) {
        if (heap[left].getKey() < heap[i].getKey()) {
            throw new TestFailedException();
        } else {
            checkHeapConstraint(left);
        }
    }
    if (right < used) {
        if (heap[right].getKey() < heap[i].getKey()) {
            throw new TestFailedException();
        } else {
            checkHeapConstraint(right);
        }
 ...
```

Aufgaben zu Min-Heaps

1. Geben Sie alle möglichen Min-Heaps zu den Zahlen 1, 2, 3, 4 und 5 an.
2. Es sei der folgende Min-Heap als heapgeordnetes Array gegeben:

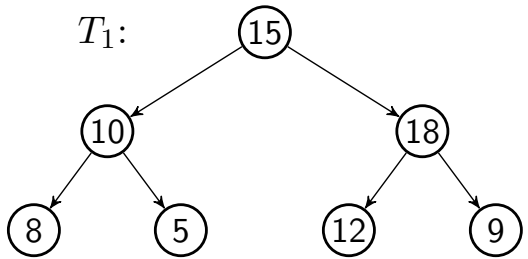




Wie sehen Heap und Array nach Anwendung der folgenden Operationen (in der gegebenen Reihenfolge) aus?
`deleteMin()`, `deleteMin()`, `add(14)`, `add(8)`

Aufgaben zu Heaps

a) In dieser Aufgabe soll ein Min-Heap mit Hilfe der *Bottom-Up-Sift-Down*-Methode aufgebaut werden. Gegeben sei die unsortierte Liste $[15, 10, 18, 8, 5, 12, 9]$, aus welcher im ersten Schritt der Methode der nachfolgende Baum T_1 generiert wird. Setzen Sie die Methode fort und zeichnen Sie nach jedem Sift-Down den jeweils entstehenden Baum.

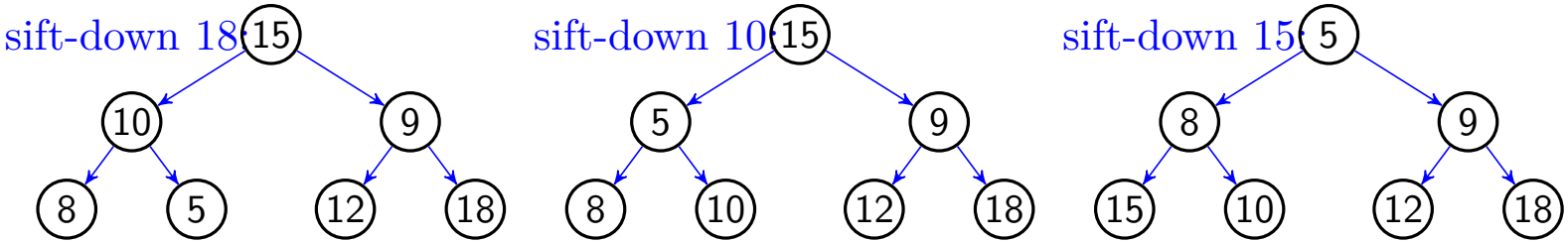


b) Begründen Sie oder widerlegen Sie per Gegenbeispiel:

1) Ein aufsteigend sortiertes Array (ohne Duplikate) ist immer auch ein binärer Min-Heap (in Form eines heapgeordneten Arrays).

2) Ein binärer Min-Heap (in Form eines heapgeordneten Arrays) ist immer auch ein aufsteigend sortiertes Array.

Lösung



- Es müssen Form-Constraint und Heap-Constraint gezeigt werden.
  - Form-Constraint: In einem aufsteigend sortierten Array sind alle Positionen besetzt. Das entspricht einem vollständigen Baum, der bis auf die letzte Ebene gefüllt ist. Die letzte Ebene ist von links nach rechts gefüllt.
  - Heap-Constraint: In einem heapgeordneten Array befinden sich die Kinder am Index 2*pos+1 und 2*pos+2. Da das Array aufsteigend sortiert ist, sind die Elemente an dieser Position immer größer(gleich) dem Vater.
- Zeigen per Gegenbeispiel:
  - Der MinHeap [11, 80, 25] ist nicht aufsteigend sortiert.

Agenda: Kürzeste Wege, Heaps, Hashing

- **Heute:**
  - Kürzeste Wege: Dijkstra
  - Heaps: Binäre Min-Heaps
  - **Hashing: Overflow und Open**

Wörterbuchoperationen

	Searching by Key	Inserting	Pre-processing
Unsorted array	O(n)	O(1)	0
Sorted array	O(log(n))	O(n)	O(n*log(n))
Sorted linked list	O(n)	O(n)	O(n*log(n))
Priority Queue	O(1) for min	O(log(n))	O(n)
Our dream	O(1)	O(1)	0

Wichtige Datenstrukturen

Datentyp	Wichtige Operationen	Besonderheit	Java-Klassen (Auswahl)
Array	A[i] |A|	**Indexbasierter Zugriff** in $O(1)$, Feste Länge.	[]
Liste	add, contains, delete, length	**Dynamische Datenstruktur**, Hohe Kosten für indexbasierten Zugriff / Suche	LinkedList, ArrayList
Stack	push, pop, top, isEmpty	Zugriff auf zuletzt eingefügtes Element in O(1), **keine Längenoperation**	Stack
Queue	enqueue, dequeue, head, isEmpty	Zugriff auf zuerst eingefügtes Element in O(1), **keine Längenoperation**	LinkedList
Heap	add, getMin, deleteMin, merge, create, size	Zugriff auf **Min / Max** in O(1), Entfernen von Min / Max in O(log n)	PriorityQueue
Hash-Tabelle	put, get, contains, size	**Konstante Laufzeit** nur im Average Case	HashMap, HashSet

Hashing: Ansätze

- Direkte Adressierung: *A[e.key] = e.value*
  - Nicht praktikabel, falls Anzahl der Schlüssel klein im Vergleich zum Wertebereich
- Hash: Datenstruktur, die die Operationen Wörterbuchoperationen: Insert, Search und Delete unterstützt.
- Wir verwenden daher eine Hashfunktion *h* und speichern *e* an Stelle *h(e)*: *A[h(e)] = e.value*
- Kollision: Tritt auf, wenn zwei Schlüssel auf den gleichen Index abgebildet werden
- Overflow Hashing (Verkettung): Kollisionen werden außerhalb des Arrays in verketteten Listen gespeichert. Es wird zusätzlicher Speicherplatz benötigt.
- Open Hashing (Offene Adressierung): Alle Elemente werden in der Hashtabelle gespeichert. Kein zusätzlicher Speicherplatz notwendig.

Aufgabe: Hashing mit offener Adressierung

Beim Hashing bestimmt die Sondierungsreihenfolge für jeden Schlüssel, in welcher Reihenfolge die Einträge der Hashtabelle auf einen freien Platz hin durchsucht werden.
Gegeben sei eine Hashtabelle mit 11 Feldern für Einträge, die durch $0, \ldots, 10$ indiziert sind, und die Hashfunktion $h(k) = k \bmod 11$.
Führen Sie für die folgenden Hashverfahren einen Schreibtischtest durch, indem Sie jeweils die Elemente 54, 15, 27, 76, 22, 5 und 14 in dieser Reihenfolge in eine leere Tabelle einfügen und diese nach jeder Einfügeoperation ausgeben.

a) **Offenes Hashing mit linearem Sondieren**
Bei Kollisionen wird hier das einzufügende Element an der nächsten freien Stelle links vom berechneten Hashwert eingefügt. Das heißt, die Sondierungsreihenfolge (die Reihenfolge, in der die Plätze im Array durchgegangen werden, bis erstmals ein freier Platz angetroffen wird) ist gegeben durch $s(k, i) = (h(k) - i) \bmod 11$ für $i = 0, \ldots, 10$.

b) **Doppeltes Hashing**
Das doppelte Hashing ist ein offenes Hashing, bei dem die Sondierungsreihenfolge von einer zweiten Hashfunktion $h'(k) = 1 + (k \bmod 7)$ abhängt. Die Position für das i-te Sondieren ist bestimmt durch die Funktion $s(k, i) = (h(k) - i \cdot h'(k)) \bmod 11$ für $i = 0, \ldots, 10$.

Anmerkung:
(-1) mod 11 = (11-1) mod 11 = 10

- Offenes Hashing mit linearem Sondieren:

0	1	2	3	4	5	6	7	8	9	10	Hash-Werte
–	–	–	–	–	–	–	–	–	–	–	Start
–	–	–	–	–	–	–	–	–	–	54	Einfügen: 54
–	–	–	–	15	–	–	–	–	–	54	Einfügen: 15
–	–	–	–	15	27	–	–	–	–	54	Einfügen: 27
–	–	–	–	15	27	–	–	–	76	54	Einfügen: 76 (1× sondiert)
22	–	–	–	15	27	–	–	–	76	54	Einfügen: 22
22	–	–	5	15	27	–	–	–	76	54	Einfügen: 5 (2×)
22	–	14	5	15	27	–	–	–	76	54	Einfügen: 14 (1×)

- Doppeltes Hashing mit zweiter Hashfunktion:

0	1	2	3	4	5	6	7	8	9	10	Hash-Werte
–	–	–	–	–	–	–	–	–	–	–	Start
–	–	–	–	–	–	–	–	–	–	54	Einfügen: 54
–	–	–	–	15	–	–	–	–	–	54	Einfügen: 15
–	–	–	–	15	27	–	–	–	–	54	Einfügen: 27
–	–	–	76	15	27	–	–	–	–	54	Einfügen: 76, $h'(76) = 7$ (1× sondiert)
22	–	–	76	15	27	–	–	–	–	54	Einfügen: 22,
22	–	–	76	15	27	–	–	–	5	54	Einfügen: 5, $h'(5) = 6$ (3×)
22	–	14	76	15	27	–	–	–	5	54	Einfügen: 14, $h'(14) = 1$ (1×)