

# Programmietechnik II

## Priority Queues and Heapsort

# Priority Queue

- Abstrakter Datentyp
- Inhalt: Elemente mit Priorität
- Operationen:
  - Einfügen: Angabe des Elements und seiner Priorität
    - Operation liefert keinen Ergebniswert
  - Entfernen: Parameterlos
    - Operation liefert das Element mit der höchsten Priorität

# Priority Queue: Implementierungsvarianten

- Speicherung in Liste/Ringpuffer
- Konstante Zeit entweder für Einfügen oder Entfernen
- Einfügen in konstanter Zeit:
  - Element wird am Ende der Queue angehängt:  $O(1)$
  - Entfernen durchsucht Liste nach Element mit höchster Priorität:  $O(n)$
- Entfernen in konstanter Zeit:
  - Elemente sind stets nach Priorität sortiert
  - Einfügen durchsucht Liste nach richtiger Position:  $O(n)$
  - Entfernen entfernt vorderstes Element
- Effiziente Implementierungen
  - Heap (Einfügen und Entfernen in  $O(\lg N)$ )
  - sortierter Baum (Komplexität hängt von Balanzierungsalgorithmus ab)

# Heaps

- Ein Baum heißt “**heap-geordnet**”, wenn der Schlüssel in jedem Knoten größer-oder-gleich den Schlüsseln in allen Kindknoten ist
  - Eigenschaft läßt sich für beliebige Bäume definieren (nicht notwendig binär)
  - Ordnung zwischen den Kindern eines Knotens ist nicht vorgeschrieben
- In einem heap-sortieren Baum ist kein Schlüssel größer als der in der Wurzel
- Ein **Heap** ist ein Menge von Schlüsseln in einem heap-geordneten (fast) vollständigen Binärbaum
  - Kindknoten von Knoten  $i$  stehen an Position  $2i$  und  $2i+1$ 
    - Annahme: Nummerierung beginnt bei 1
  - letzte Ebene des Baums wird “von links” aufgefüllt

# Erzeugung von Heaps

- Operation: Anhängen eines neuen Elements
  - auch: Erhöhen der Priorität eines Elements im Heap
  - Neues Element wird am Ende des Heaps eingefügt
  - Heap-Eigenschaft u.U. verletzt, wenn neuer Knoten größer als Elternknoten
- Wiederherstellen der Eigenschaft: Vertauschen des Knotens mit seinem Elternknoten (swim)
  - Elternknoten von Position  $N$  ist an Position  $N/2$
  - Knoten ist nun größer als seine beiden Kindknoten
    - per Definition größer als sein alter Elternknoten
    - auch größer als der andere Kindknoten, weil dieser kleiner als der alte Elternknoten
  - Operation muss u.U. wiederholt werden, evtl. bis neuer Knoten zum Wurzelknoten wird

# Erzeugen von Heaps (2)

- Operation: Senken der Priorität eines Elements
  - auch: Entfernen der Wurzel, Austausch z.B. mit letztem Element
- Wiederherstellen der Heap-Eigenschaft: Vertauschen des Elements mit dem größeren seiner Kinder (sink)
  - eventuell wiederholt, bis Element Blatt im Baum ist
- Basis der Algorithmen: `less(index1, index2)`, `exch(index1, index2)`

# Erzeugung von Heaps: swim

```
void swim(int k)
{
    while (k > 1 && less (k/2, k))
    {
        exch(k, k/2); k = k/2;
    }
}
```

# Erzeugen von Heaps: sink

```
void sink(int k, int N)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j); k = j;
    }
}
```

# Priority-Queue auf Heap-Basis

- Initialzustand: Feld ist leer
- Einfügen
  - Anfügen des neuen Elements ans Ende,  $N++$
  - swim( $N$ )
  - Maximal  $\lg N$  Vergleiche
- Entfernen des größten Elements
  - Tauschen des ersten und des letzten Elements
  - $N--$
  - sink( $1, N$ )
  - Ergebnis: Element  $N+1$
  - Maximal  $2 \cdot \lg N$  Vergleiche

# Heapsort

1. Heap-Aufbau von unten nach oben
  - Aufbau von Teilheaps, beginnend bei  $N/2 \dots 1$
  - Integration mehrerer Teilheaps in größere
2. Sortieren durch wiederholtes Entfernen des größten Elements
  - jeweils größtes Element wird an jeweils letzte Position des Felds vertauscht
  - danach Wiederherstellen der Heap-Eigenschaft

# Heapsort (2)

```
for (int k = N/2; k >= 1; k--)  
    sink(k, N);  
while (N > 1)  
    { exch(1, N); sink(1, --N); }
```

# Analyse von Heapsort

- Phase 1 (Aufbau):  $O(N)$ 
  - $N/4$  Heaps der Tiefe 1:  $N/2$  Vergleiche (2 pro sink)
  - $N/8$  Heaps der Tiefe 2:  $2 \cdot N/4$  Vergleiche
  - $N/16$  Heaps der Tiefe 3:  $3 \cdot N/8$  Vergleiche
  - $N/32$  Heaps der Tiefe 4:  $4 \cdot N/16$  Vergleiche
  - ...
  - insgesamt weniger als  $2 \cdot N$  Vergleiche
- Phase 2 (Sortieren):  $O(N \lg N)$ 
  - $N$  Elemente, pro Elemente einmal sink, jeweils weniger als  $\lg N$  Vergleiche
- Gesamtlaufzeit:  $O(N \lg N)$ 
  - Weniger als  $2 N \lg N$  Vergleiche
- Speicherverbrauch:  $O(1)$
- nicht stabil
- Worst-case-Laufzeit besser als Quicksort, Speicherverbrauch besser als Mergesort
  - aber: höhere Laufzeit für Zufallsdaten