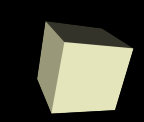
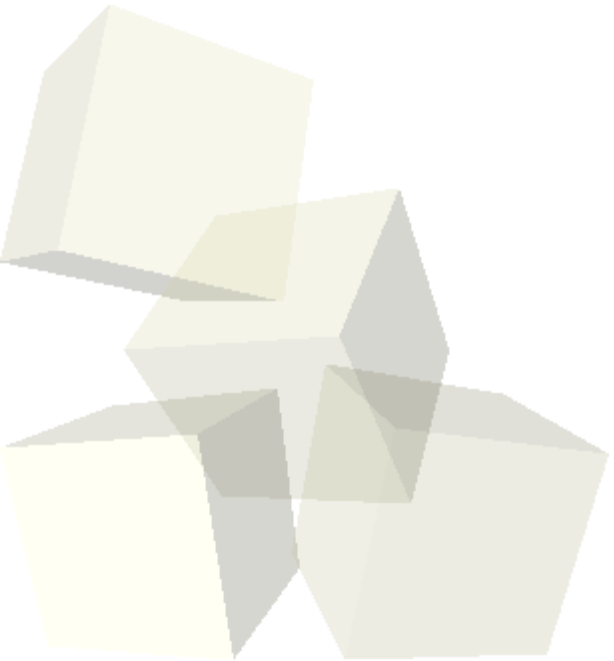


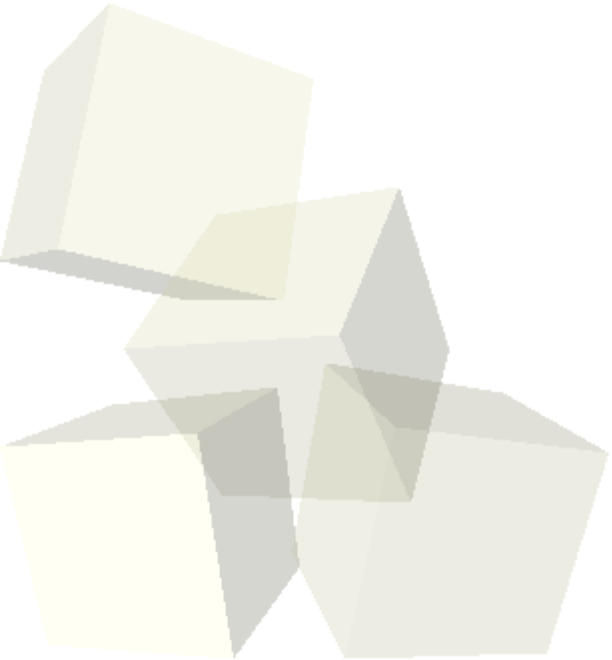
C-Schulung Tag 8a

Herzlich willkommen



Agenda

- Prioritätswarteschlangen
- Vektoren
- Zeitkomplexität
- Heaps
- Heapsort



Prioritätswarteschlangen

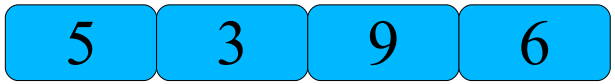
- Prioritätswartschlangen spielen dann eine Rolle, wenn man Objekte nach Priorität geordnet bearbeiten möchte:
  - Prozess-Scheduling in Betriebssystemen
  - Besuchen des nächsten Nachbarn bei Dijkstra
- Es gibt nur zwei zentrale Operationen:
  - Neues Element in die Warteschlange einreihen
  - Kleinstes Element aus der Warteschlange entfernen
- Beide Operationen sind gleich wichtig:
  - #Eingereihte Elemente $\approx$ #Entfernte Elemente

Vektor als Warteschlange

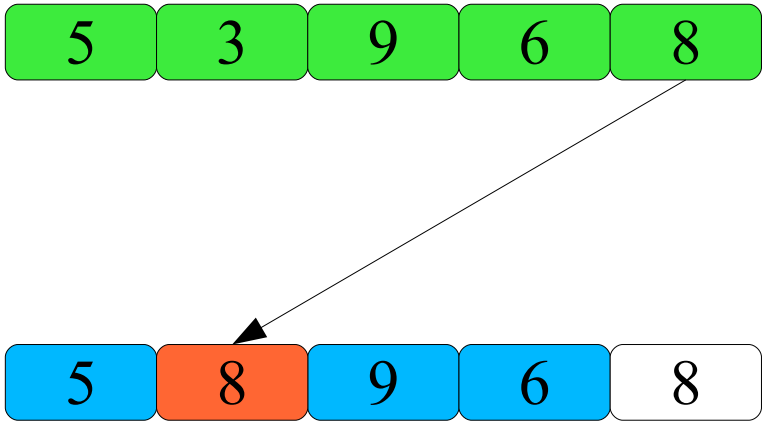
```
struct vector
{
    int * data;
    int capacity;
    int used;
};

void vector__init(struct vector * self, int capacity)
{
    self->data = calloc(capacity, sizeof(int));
    self->capacity = capacity;
    self->used = 0;
}

void vector__done(struct vector * self)
{
    assert(self->data);
    free(self->data);
    self->data = NULL;
}
```

unsortierter Vektor

```
/* Einfügen am Ende ist günstig */
void vector__insert(struct vector * self, int element)
{
    assert(self->used < self->capacity);
    self->data[self->used++] = element;
}
```

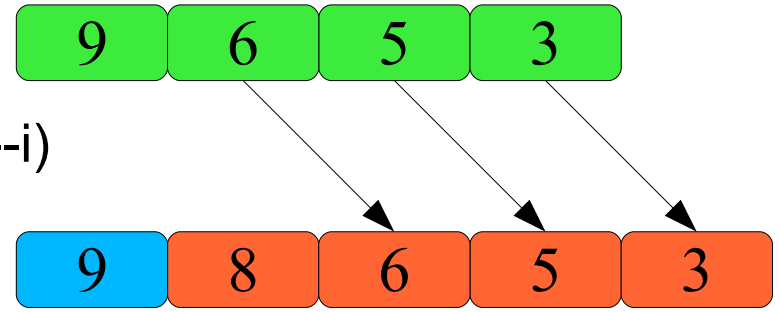



```
/* Niedrigstes Element muss gesucht werden, teuer */
int vector__remove_minimum(struct vector * self)
{
    int * a = self->data, index_of_minimum = 0, i, result;
    assert(self->used > 0);
    for (i = 1; i < self->used; ++i)
    {
        if (a[i] < a[index_of_minimum])
            index_of_minimum = i;
    }
    result = a[index_of_minimum];
    a[index_of_minimum] = a[--self->used];
    return element;
}
```

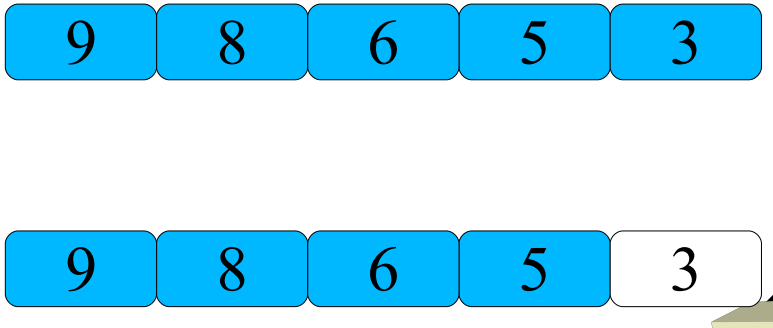


absteigend sortierter Vektor

```
/* Korrekte Position muss bestimmt und Einträge verschoben werden, teuer */
void sorted_vector__insert(struct vector * self, int element)
{
    int * a = self->data, i;
    assert(self->used < self->capacity);
    for (i = self->used; (i > 0) && (a[i-1] < element); --i)
    {
        a[i] = a[i-1];
    }
    a[i] = element;
    self->used++;
}
```




```
/* Entfernen am Ende ist günstig */
int sorted_vector__remove_minimum(struct vector * self)
{
    assert(self->used > 0);
    return self->data[--self->used];
}
```

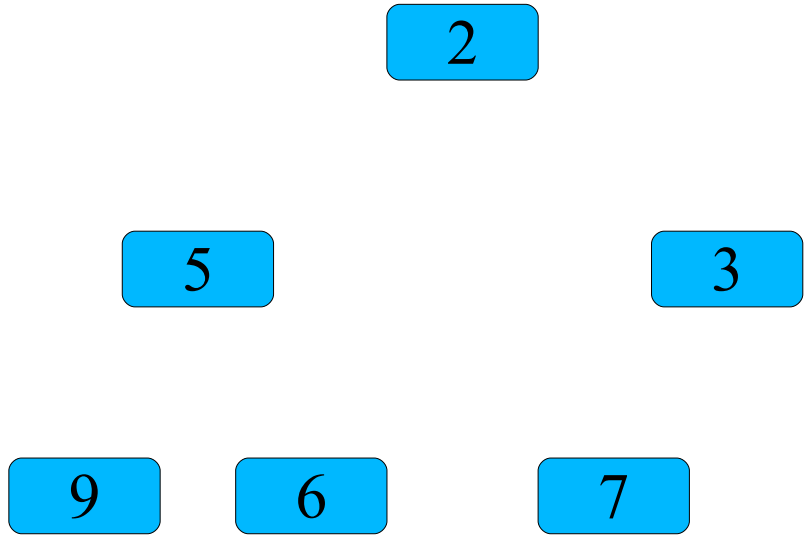


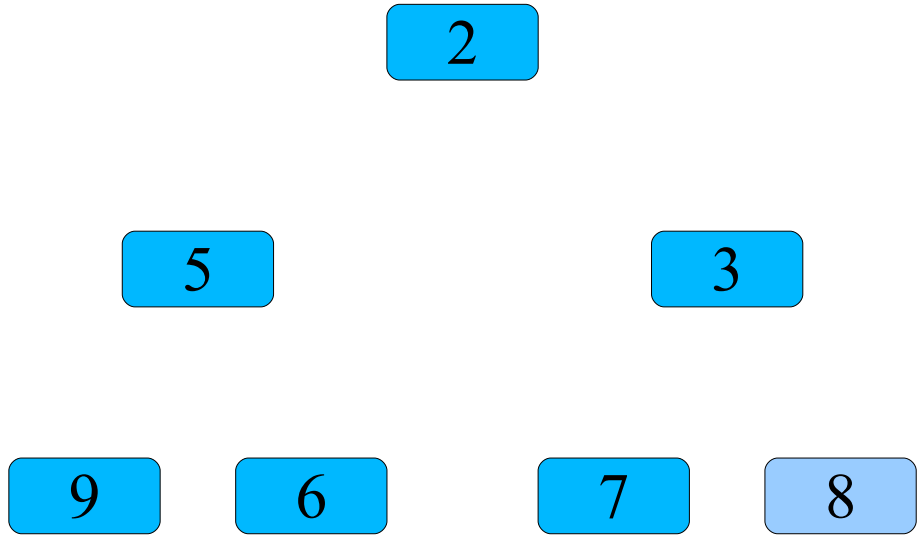
Zeitkomplexität

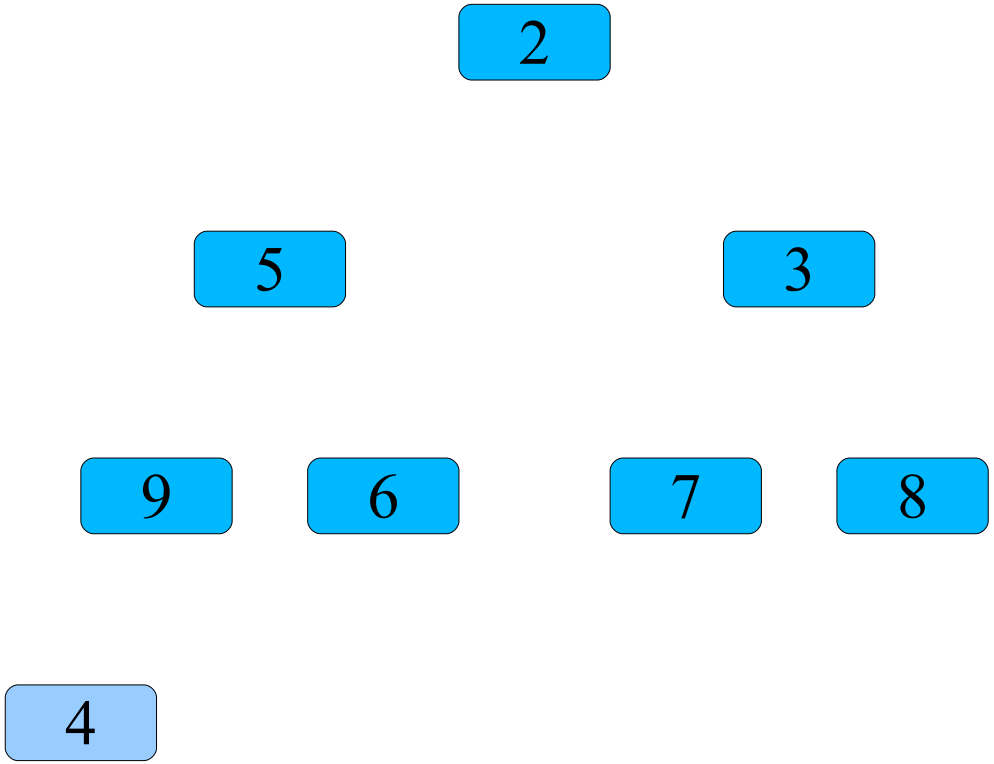
Datenstruktur	Einreihen	Entfernen	n Operationen
unsortierter Vektor	O(1)	O(n)	$O(n^2)$
sortierter Vektor	O(n)	O(1)	$O(n^2)$
Heap	O(log n)	O(log n)	O(n log n)

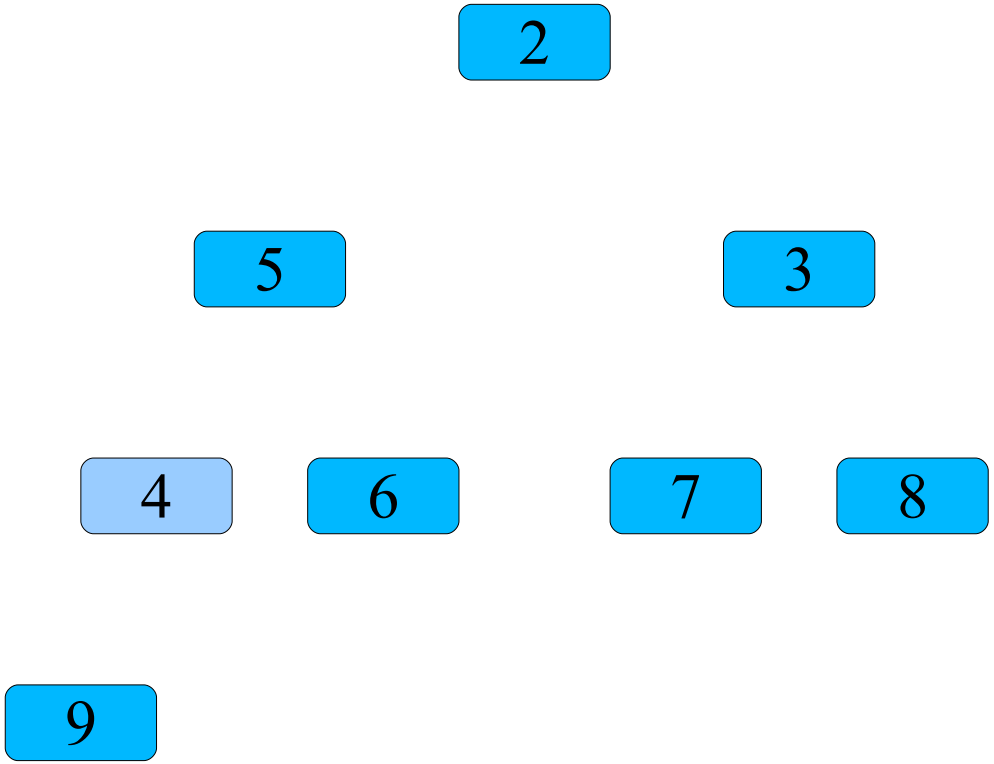
Zeitkomplexität

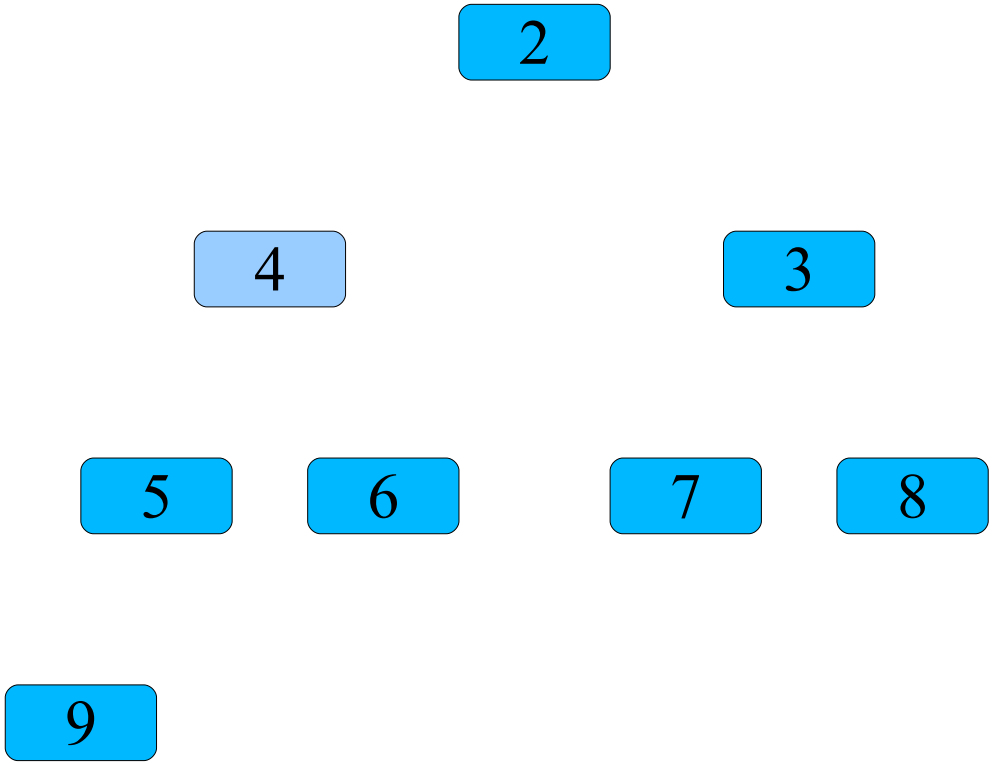
Komplexitätsklasse	1 GHz Rechner Problemgröße	2 GHz Rechner Problemgröße
O(1)	n	n
O(log n)	n	n^2
O(n)	n	2n
O(n log n)	n	<2n
O(n^2)	n	1,41n
O(2^n)	n	n+1

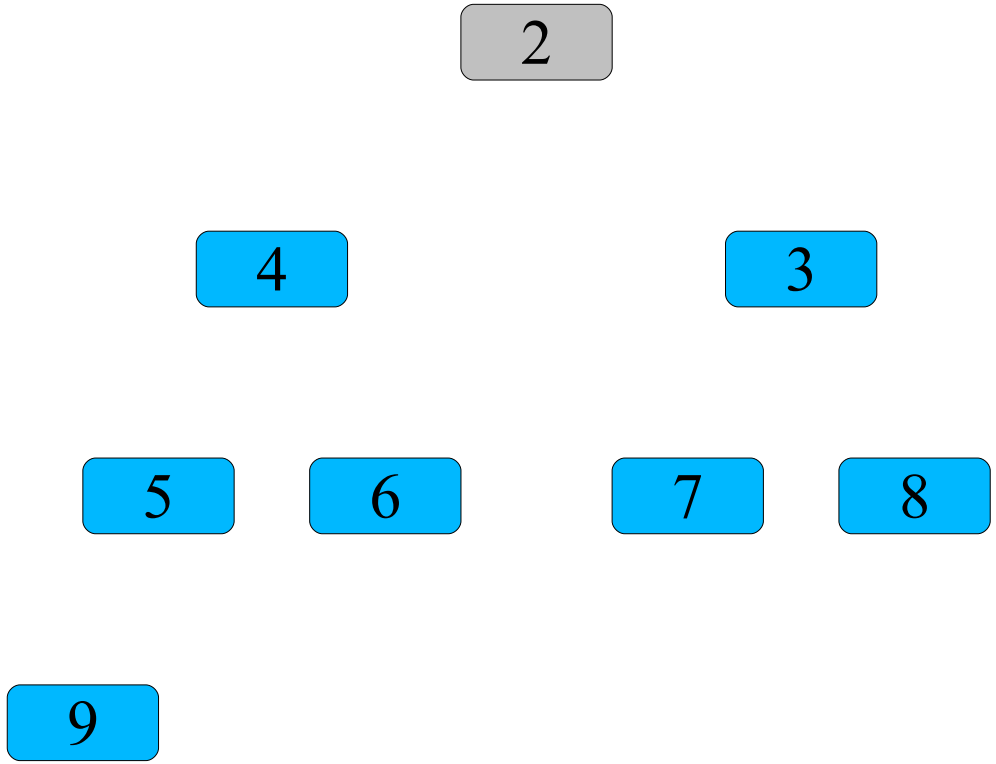

2
5
3
9
6
7

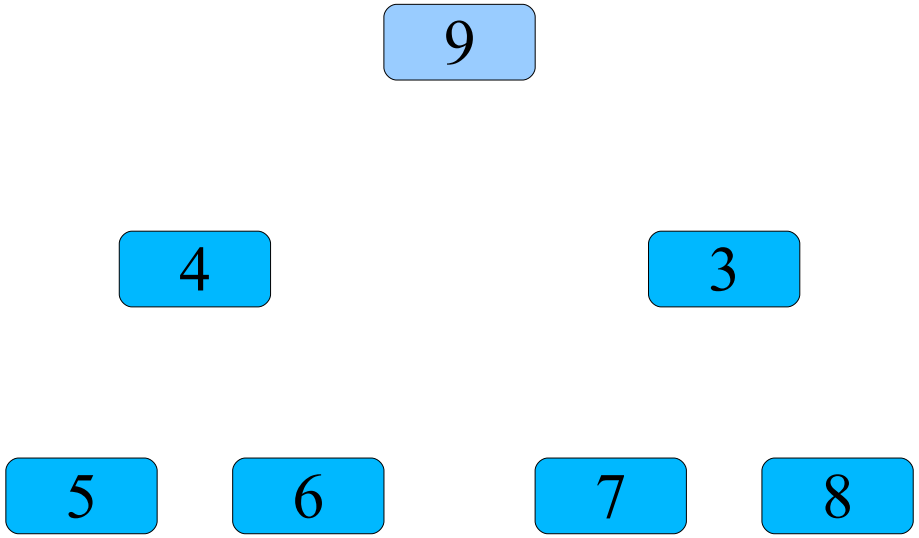

2
5
3
9
6
7
8

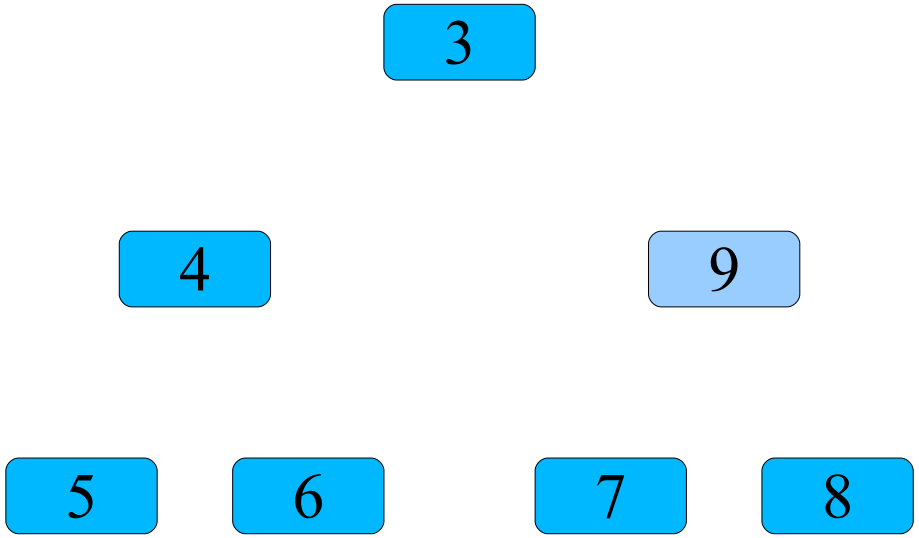

2
5
3
9
6
7
8
4

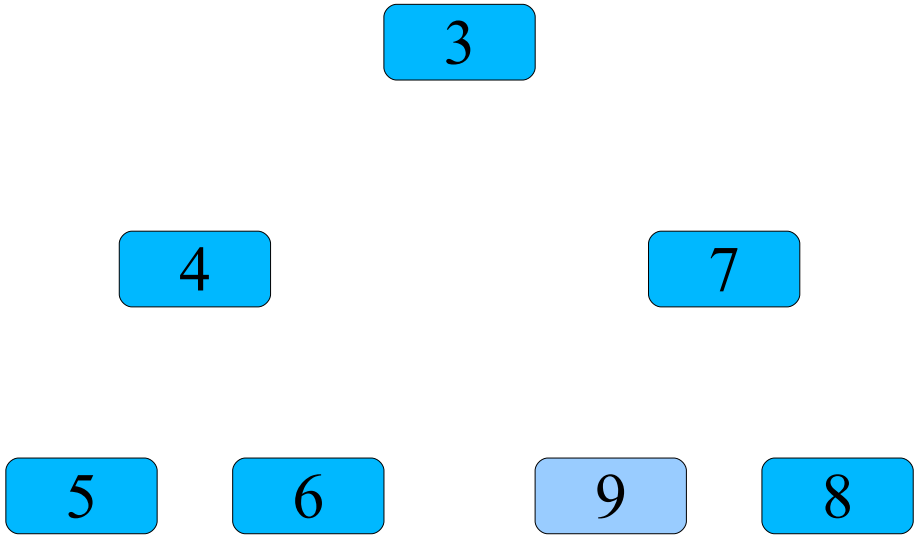

2
5
3
4
6
7
8
9

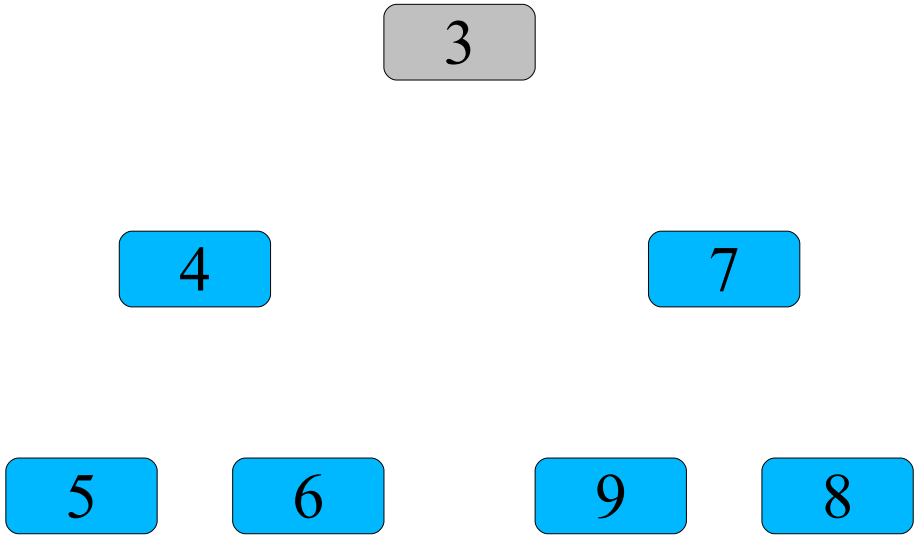

2
4
3
5
6
7
8
9

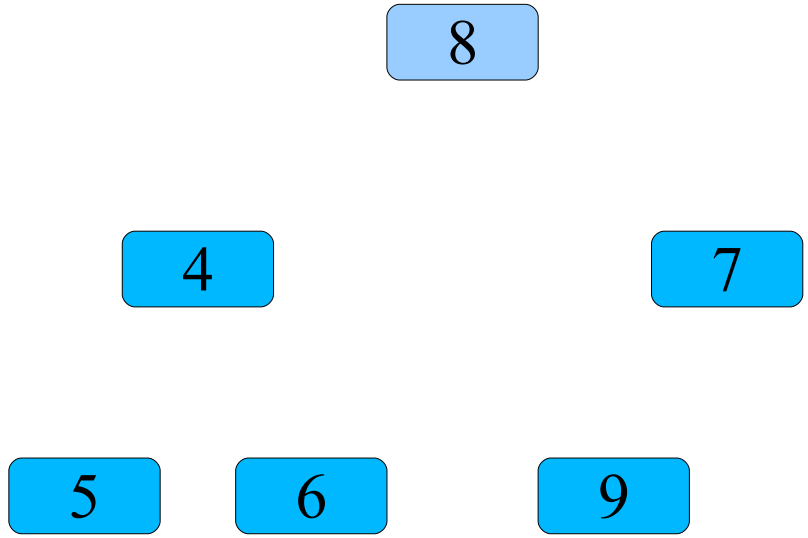

2
4
3
5
6
7
8
9

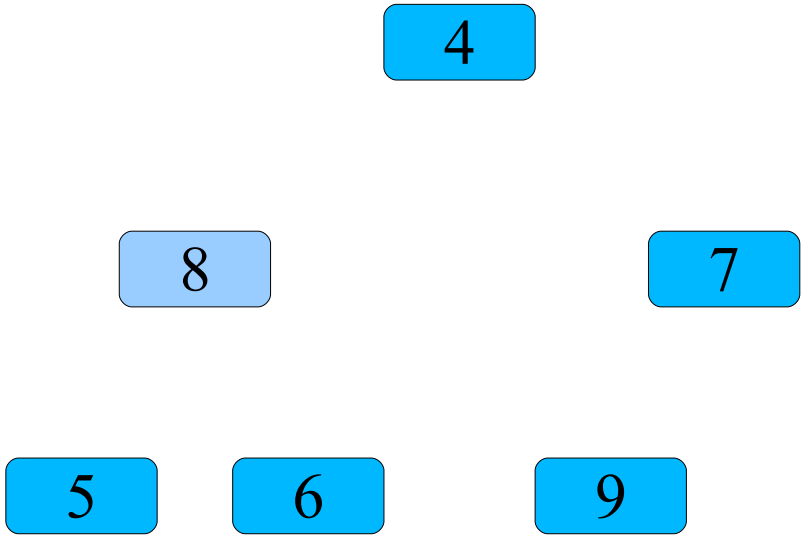

9
4
3
5
6
7
8

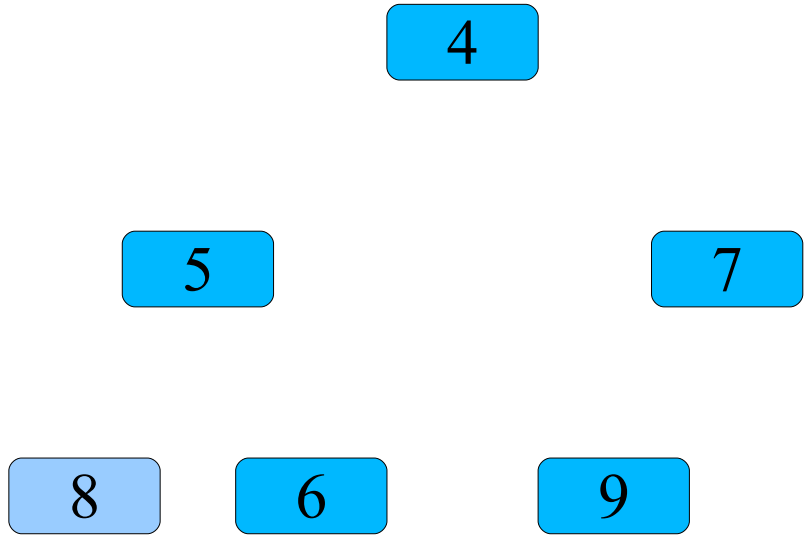

3
4
9
5
6
7
8


3
4
7
5
6
9
8


3
4
7
5
6
9
8


8
4
7
5
6
9


4
8
7
5
6
9

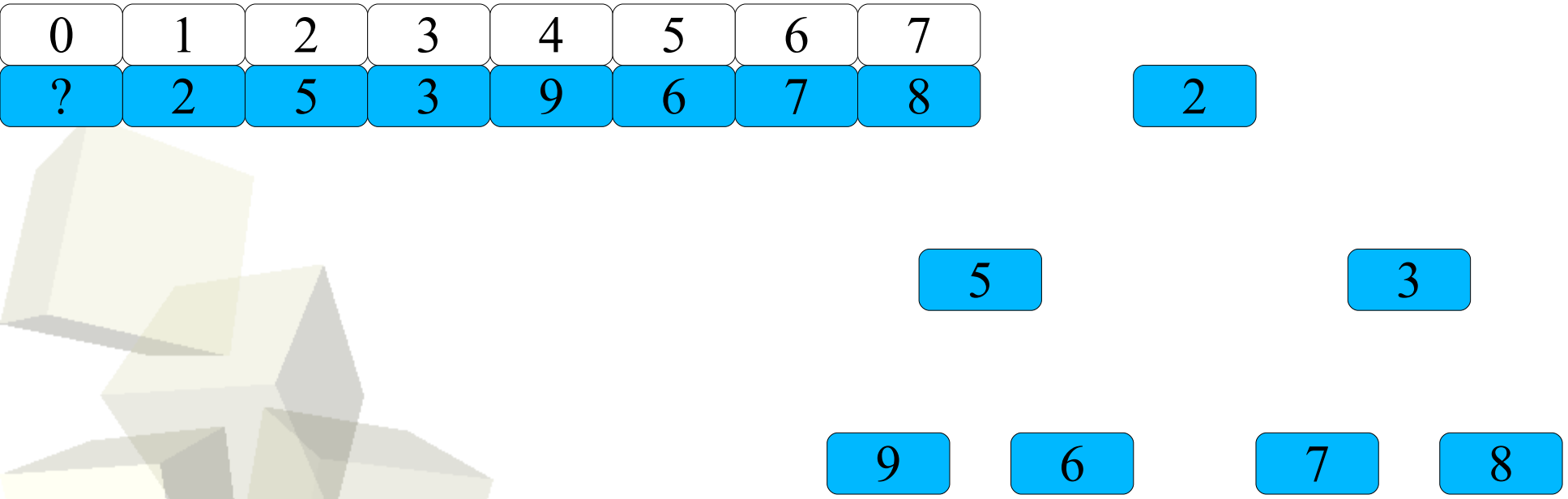

4
5
7
8
6
9

Heaps

- Das kleinste Element befindet sich in der Wurzel.
- Alle anderen Knoten haben einen Vaterknoten, der ein kleineres Element speichert.
- Die Blätter befinden sich auf maximal 2 Ebenen.
- Die unterste Ebene ist links ausgerichtet.
- Ein neues Element wird rechts unten eingefügt und dann nach oben verschoben, solange der Vaterknoten ein größeres Element speichert.
- Beim Entfernen des kleinsten Elements wird das letzte Element an die Wurzel gesetzt und dann mit dem jeweils kleineren Kindknoten vertauscht, solange er mindestens ein Kind hat und noch ein größeres Element speichert als seine Kinder.

Implementation

- Wir speichern die Knoten in einem Array.
- Index 0 ignorieren wir einfach.
- Die Wurzel hat den Index 1.
- Die Kinder eines Knotens i sind 2i und 2i+1.
- Der Vater eines Knotens i ist $\lfloor i/2 \rfloor$.



Heapsort

- Einen Heap kann man zum Sortieren benutzen:
  - Befülle einen zunächst leeren Heap mit den Elementen eines unsortierten Arrays.
  - Entferne alle Elemente aus dem Heap – dabei kommen sie in geordneter Reihenfolge heraus.
- Dieses Verfahren nennt man Heapsort.
- Die Komplexität ist immer O(n log n).
  - 2n Operationen mit jeweils O(log n) Kosten
- Theoretisch ist Heapsort sehr schnell, das Muster der Speicherzugriffe ist aber cache-unfreundlich.
- Heapsort wird gerne als “Notbremse” verwendet, wenn schnellere Verfahren bei ungünstigen Eingaben zu O(n^2) zu entarten drohen.