

Algorithms and Data Structures

B3. Heaps, Priority Queues and Heapsort

Gabriele Röger

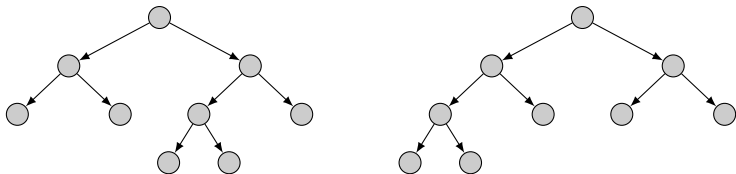
University of Basel

April 4, 2024

Our Plan for Today

- Data structure **heap**
- Algorithm **heapsort** that uses a heap.
- Abstract data type **priority queue**,
that can be implemented with a heap.

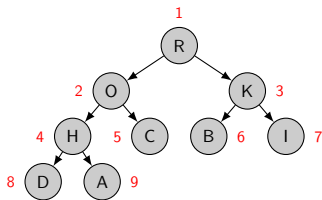
Binary Trees



- **Binary tree**: each node has at most two successor nodes.
- We distinguish the **left** and the **right child** of a node.
- A single child can be the left or the right child.
- A **nearly complete binary tree** is completely filled on all levels except possibly the lowest, which is filled from left to right.

Nearly Complete Binary Trees as Arrays

- Consider 1-indexed arrays.
- Every such array can be interpreted as a nearly complete binary tree and vice versa.
 - Assign numbers $1, 2, \dots$ to nodes in tree from root to leaves and left to right on each level.
 - The number is the index in the array.
 - The left child of node i gets $2i$ and the right child $2i + 1$.



Helper Functions

```
def left(i):  
    return 2 * i
```

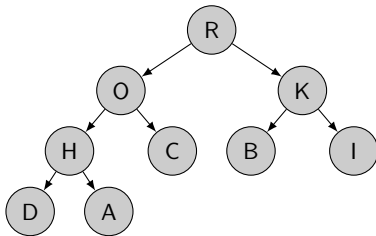
```
def right(i):  
    return 2 * i + 1
```

```
def parent(i):  
    return i // 2
```

Heap: Max-Heap

Definition: Max-Heap

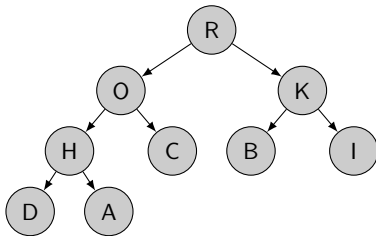
A nearly complete binary tree is a max-heap if the key stored in each node is greater or equal to the keys of each of its children.



Heap: Max-Heap

Definition: Max-Heap

A nearly complete binary tree is a max-heap if the key stored in each node is greater or equal to the keys of each of its children.

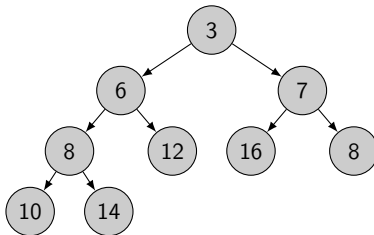


The largest key in a max-heap is at the root.

Heap: Min-Heap

Definition: Min-Heap

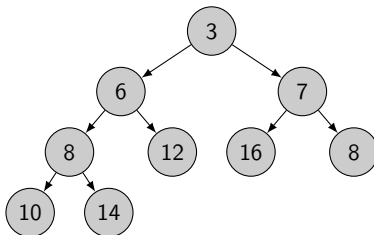
A nearly complete binary tree is a min-heap if the key stored in each node is smaller or equal to the keys of each of its children.



Heap: Min-Heap

Definition: Min-Heap

A nearly complete binary tree is a min-heap if the key stored in each node is smaller or equal to the keys of each of its children.



The smallest key in a min-heap is at the root.

We will focus on max-heaps. Min-heaps are implemented analogously.

Max-heaps: Operations

We will implement the following operations:

- `build_max_heap` transforms an array into a max-heap.
- `max_heap_maximum` returns the largest element.
- `max_heap_extract_max` removes and returns the largest element.
- `max_heap_insert` add an item to the heap.

We will use two helper functions that fix local violations of the heap property:

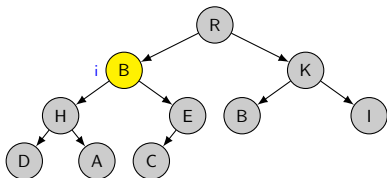
- `sink` moves an element with a too small key downwards.
- `swim` moves an element with a too large key upwards.

Helper Function: Sink

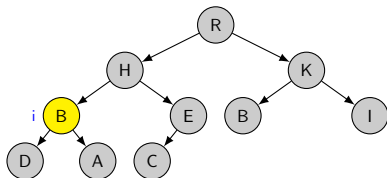
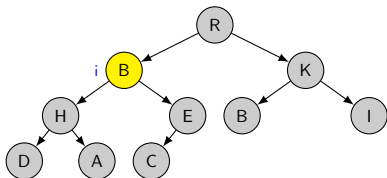
- **Sink** assumes that the left and right subtree of node i are max-heaps but the key at i might be smaller than the keys at $2i$ or $2i + 1$ (root of left and right sub-tree), violating the heap property.
- **Idea**: Let the entry recursively “float down” into the subtree with the larger key at its root.

In the book by Cormen et al. the function is called `max_heapify`.

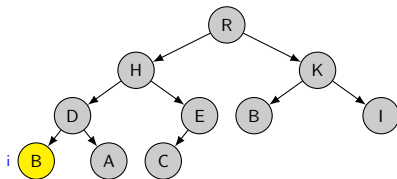
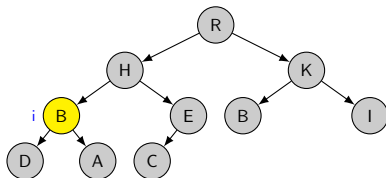
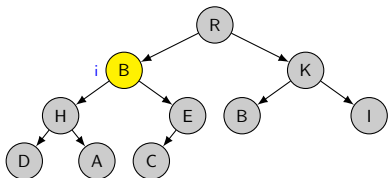
Sink: Example



Sink: Example



Sink: Example



Jupyter Notebook



Jupyter notebook: `heaps.ipynb`

Sink: Implementation

```
def sink(heap, i, heap_size=None):
    if heap_size is None:
        heap_size = len(heap) - 1

    l = left(i)
    r = right(i)
    if l <= heap_size and heap[l] > heap[i]:
        largest = l
    else:
        largest = i
    if r <= heap_size and heap[r] > heap[largest]:
        largest = r
    if largest != i:
        heap[i], heap[largest] = heap[largest], heap[i]
        sink(heap, largest, heap_size)
```

Parameter `heap_size` can be used to exclude some entries at the end of the array from the heap (these positions will be ignored).

Sink: Running time

Simple insight:

- Let h be the height of the subtree rooted at position i .
- Then the worst-case running time of sink is $O(h)$.

Sink: Running time

Simple insight:

- Let h be the height of the subtree rooted at position i .
- Then the worst-case running time of `sink` is $O(h)$.

Full story:

- Let n be the number of nodes of the subtree rooted at position i .
- Determining the final value of `largest` is $\Theta(1)$.
- Each subtree has size at most $2n/3$, so for the worst-case running time T of `sink`, we have

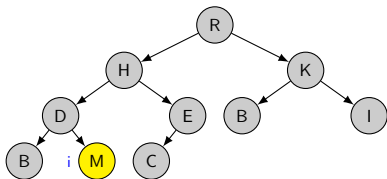
$$T(n) \leq T(2n/3) + \Theta(1).$$

- By master theorem (case 2), $T(n) \in O(\log_2 n)$.

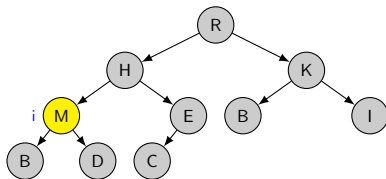
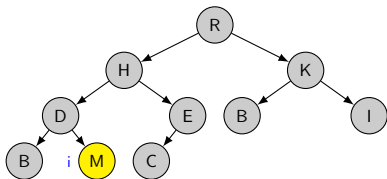
Helper Function Swim

- **Sink** lets an entry with a too small key recursively “float down” into the subtree (a heap) with the larger key at its root.
- We now consider the counterpart **swim**: let an entry with a too large key float up in a tree that is otherwise a heap.

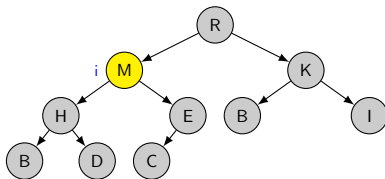
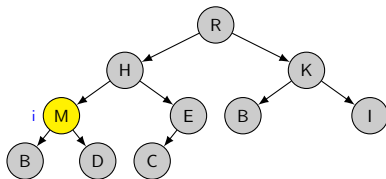
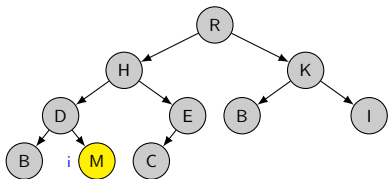
Swim: Example



Swim: Example



Swim: Example



Swim: Implementation

```
def swim(heap, i):  
    parent_index = parent(i)  
    # as long as i is not the root and the parent  
    # of i has a smaller key than i  
    while i > 1 and heap[parent_index] < heap[i]:  
        # swap the entries of nodes i and its parent  
        heap[parent_index], heap[i] = heap[i], heap[parent_index]  
  
        # continue floating up the entry from the parent  
        i = parent_index  
        parent_index = parent(i)
```

Running time:

Swim: Implementation

```
def swim(heap, i):  
    parent_index = parent(i)  
    # as long as i is not the root and the parent  
    # of i has a smaller key than i  
    while i > 1 and heap[parent_index] < heap[i]:  
        # swap the entries of nodes i and its parent  
        heap[parent_index], heap[i] = heap[i], heap[parent_index]  
  
        # continue floating up the entry from the parent  
        i = parent_index  
        parent_index = parent(i)
```

Running time: $O(\log_2 n)$

(height of a nearly complete binary tree with n nodes is $\lceil \log_2 n \rceil$)

Build_max_heap

We can use sink to transform any array into a max-heap in a bottom-up fashion, processing all nodes from the second-lowest layer up to the root.

```
def build_max_heap(array):  
    heap_size = len(array) - 1  
  
    # all elements from positions heap_size//2 + 1  
    # to heap_size are leaves of the tree.  
    for i in range(heap_size//2, 0, -1):  
        sink(array, i, heap_size)
```

Running Time of `build_max_heap`

- Heap with n elements has height $\lfloor \log_2 n \rfloor$.
- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes rooting subtrees of height h .
 - The call of `sink` for each such node is $O(h)$.
 - Use c for the constant hidden in the asymptotic notation.

Running Time of `build_max_heap`

- Heap with n elements has height $\lfloor \log_2 n \rfloor$.
- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes rooting subtrees of height h .
 - The call of `sink` for each such node is $O(h)$.
 - Use c for the constant hidden in the asymptotic notation.

$$\begin{aligned} 3T(n) &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\ &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} ch = nc \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\ &\leq nc \sum_{h=0}^{\infty} \frac{h}{2^h} \leq nc \frac{1/2}{(1 - 1/2)^2} \in O(n) \end{aligned}$$

(cf. Cormen et al., p. 169 for reasons for inequalities; you may ignore the math.)

Running Time of `build_max_heap`

- Heap with n elements has height $\lfloor \log_2 n \rfloor$.
- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes rooting subtrees of height h .
 - The call of `sink` for each such node is $O(h)$.
 - Use c for the constant hidden in the asymptotic notation.

$$\begin{aligned} 3T(n) &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\ &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} ch = nc \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\ &\leq nc \sum_{h=0}^{\infty} \frac{h}{2^h} \leq nc \frac{1/2}{(1 - 1/2)^2} \in O(n) \end{aligned}$$

(cf. Cormen et al., p. 169 for reasons for inequalities; you may ignore the math.)

We can create a heap in linear time in the number of entries.

Determining the Maximum Element

In a max-heap, it is trivial to determine the largest element: it is the element at the root.

```
def max_heap_maximum(heap, heap_size):  
    if heap_size < 1:  
        raise Exception("empty heap")  
    else:  
        return heap[1]
```

Running time:

Determining the Maximum Element

In a max-heap, it is trivial to determine the largest element: it is the element at the root.

```
def max_heap_maximum(heap, heap_size):  
    if heap_size < 1:  
        raise Exception("empty heap")  
    else:  
        return heap[1]
```

Running time: $\Theta(1)$

Extracting the Maximum Element

If we remove the largest element, we fill the position with the bottom-right element and restore the heap property with `sink` on position 1.

```
def max_heap_extract_max(heap, heap_size):  
    maximum = max_heap_maximum(heap, heap_size)  
    heap[1] = heap[heap_size]  
    sink(heap, 1, heap_size)  
    return maximum  
  
    # the externally handled heap_size  
    # needs to be decremented
```

Running time:

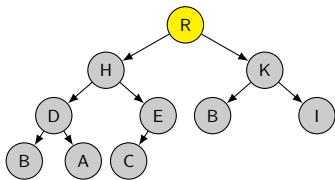
Extracting the Maximum Element

If we remove the largest element, we fill the position with the bottom-right element and restore the heap property with `sink` on position 1.

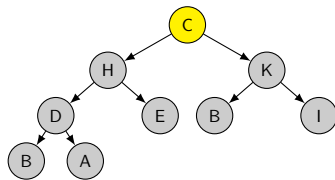
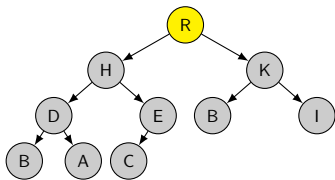
```
def max_heap_extract_max(heap, heap_size):  
    maximum = max_heap_maximum(heap, heap_size)  
    heap[1] = heap[heap_size]  
    sink(heap, 1, heap_size)  
    return maximum  
  
# the externally handled heap_size  
# needs to be decremented
```

Running time: $O(\log_2 n)$ (with n size of the heap)

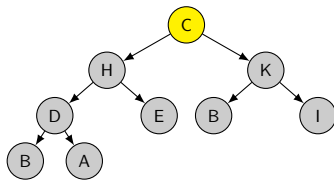
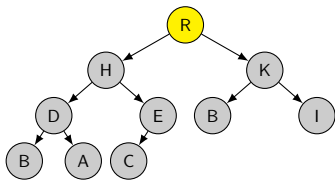
Extracting the Maximum Element: Example



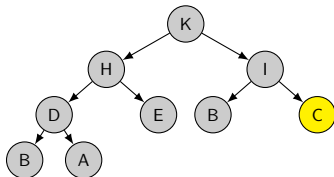
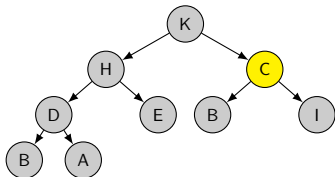
Extracting the Maximum Element: Example



Extracting the Maximum Element: Example



Let the element sink from the root to a suitable node:



Inserting an Element

We insert an element as a new leaf and let it swim to restore the heap property:

```
def max_heap_insert(heap, item, heap_size):  
    if heap_size < len(heap) - 1:  
        # we still have space in the array  
        heap[heap_size + 1] = item  
    else:  
        assert heap_size == len(heap) - 1  
        heap.append(item)  
    swim(heap, heap_size + 1)
```

Running time:

Inserting an Element

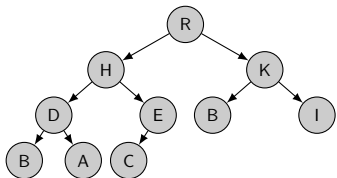
We insert an element as a new leaf and let it swim to restore the heap property:

```
def max_heap_insert(heap, item, heap_size):  
    if heap_size < len(heap) - 1:  
        # we still have space in the array  
        heap[heap_size + 1] = item  
    else:  
        assert heap_size == len(heap) - 1  
        heap.append(item)  
    swim(heap, heap_size + 1)
```

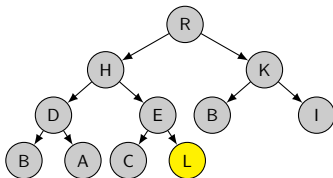
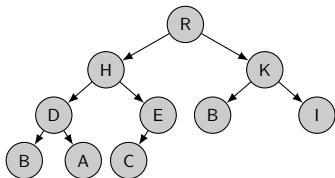
Running time: $O(\log_2 n)$ (with n size of the heap)

Only amortized if we are precise wrt. the append operation.

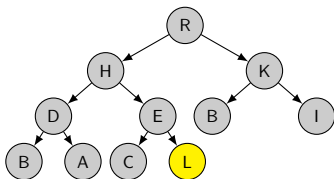
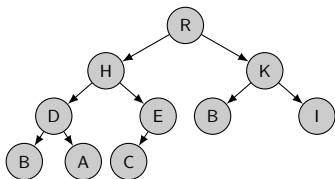
Inserting an Element: Example



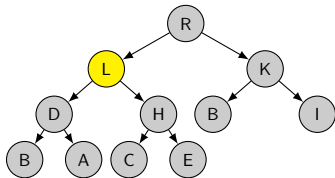
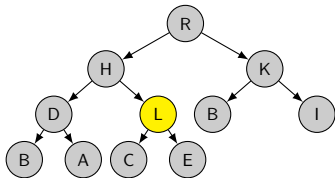
Inserting an Element: Example



Inserting an Element: Example



Let the element swim from the leaf to a suitable node:



Heapsort

Heapsort

- Basic idea as in selection sort but from right to left:
Successively swap the largest element to the end of the non-sorted range.
- We can represent the heap directly in the input sequence, so that heapsort only needs constant additional memory.

Heapsort

```
# assumes that array[0] is not part of the input sequence
def heapsort(array):
    build_max_heap(array)
    # i ranges from last position down to position 1
    for i in range(len(array) - 1, 0, -1):
        # swap largest element from heap to position i
        array[i], array[1] = array[1], array[i]
        # restore heap_property for heap (in range 1,...,i-1)
        sink(array, 1, i-1)
```

Heapsort

```
# assumes that array[0] is not part of the input sequence
def heapsort(array):
    build_max_heap(array)
    # i ranges from last position down to position 1
    for i in range(len(array) - 1, 0, -1):
        # swap largest element from heap to position i
        array[i], array[1] = array[1], array[i]
        # restore heap_property for heap (in range 1, ..., i-1)
        sink(array, 1, i-1)
```

- Building the heap takes linear time in n (length of array).
- We have a linear number of iterations of the for loop, each running in $O(\log_2 n)$.
- Overall running time $O(n \log_2 n)$.

Remarks

- Heapsort is asymptotically optimal wrt. running time and memory requirements:
 - Running time $O(n \log n)$.
 - Additional memory $O(1)$ (in-place)
- Practical disadvantage: Does not efficiently use the CPU cache because of poor locality of reference (swapping elements that do not have close storage locations)
- As an in-place approach still relevant, e.g. for embedded systems.

Priority Queue

ADT Priority Queue

A **priority queue** is an ADT for maintaining a collection of elements, each with an associated key.

A max-priority queue supports the following operations:

- `insert(x, k)` inserts element `x` with key `k`.
- `maximum()` returns the element with the largest key.
- `extract_max()` returns and removes the element with the largest key.

Min-priority queues analogously prioritize elements with small keys.

Priority Queues: Applications

- **Protocols for local area networks** use them to ensure that high-priority applications experience lower latency than other applications.
- **Prim's algorithm** for minimum spanning trees and **Dijkstra's algorithm** for finding shortest paths in graphs use them for the processing order of the nodes of the graph (Ch. C4/C6).
- **Huffman coding** for lossless data compression uses them to prioritize nodes with high probability.

Jupyter Notebook

We can implement a priority queue with a heap:



Jupyter notebook: `heaps.ipynb`

Summary

- **(Max-)Heaps** support the following operations:
 - Build heap from array: $O(n)$
 - Return largest element: $O(1)$
 - Remove largest element: $O(\log n)$
 - Insert element: $O(\log n)$

Summary

- **(Max-)Heaps** support the following operations:
 - Build heap from array: $O(n)$
 - Return largest element: $O(1)$
 - Remove largest element: $O(\log n)$
 - Insert element: $O(\log n)$
- **Heapsort** uses a heap to sort an array.
 - Can maintain the heap in the space of its input array.
 - In-place sorting algorithm.

Summary

- **(Max-)Heaps** support the following operations:
 - Build heap from array: $O(n)$
 - Return largest element: $O(1)$
 - Remove largest element: $O(\log n)$
 - Insert element: $O(\log n)$
- **Heapsort** uses a heap to sort an array.
 - Can maintain the heap in the space of its input array.
 - In-place sorting algorithm.
- A **priority queue** is an **abstract data type**.
 - Can insert items with a priority (= key).
 - Can obtain the item with the highest priority.
 - Implementation with heaps
(or AVL trees or Fibonacci heaps; not covered in this course).