Heaps

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

“Heap” as a data structure

- In dictionary:

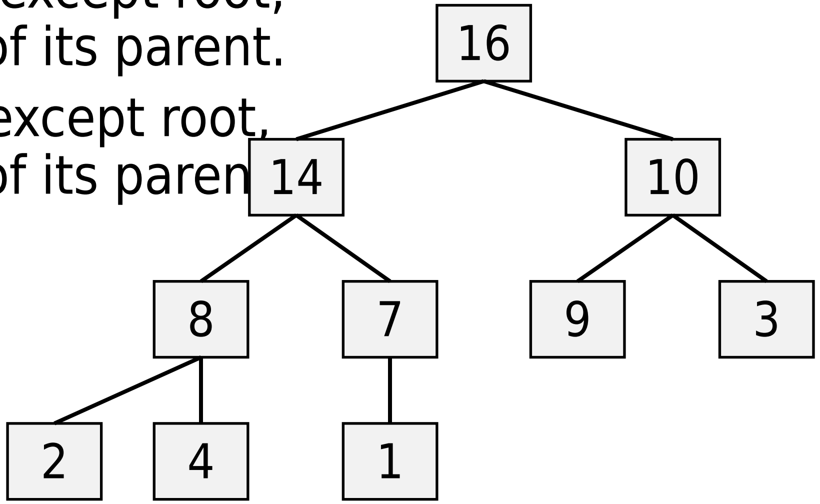
heap[1] /hiːp/ *noun* [countable]
Word origin
1 a large untidy pile of things:
a rubbish heap

- In computer science, a *heap* is a data structure that is used to represent a collection of “*somewhat organized*” items.
  - In fact, the word has other meanings in computer science…

Data structure

Binary Heap

- A binary heap is a **complete binary tree**, in which each node represents an item.
  - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Values in the nodes satisfy **heap-property**.
  - **Max-heap:** for each node except root, value of that node ≤ value of its parent.
  - **Min-heap:** for each node except root, value of that node ≥ value of its parent.



Data structure

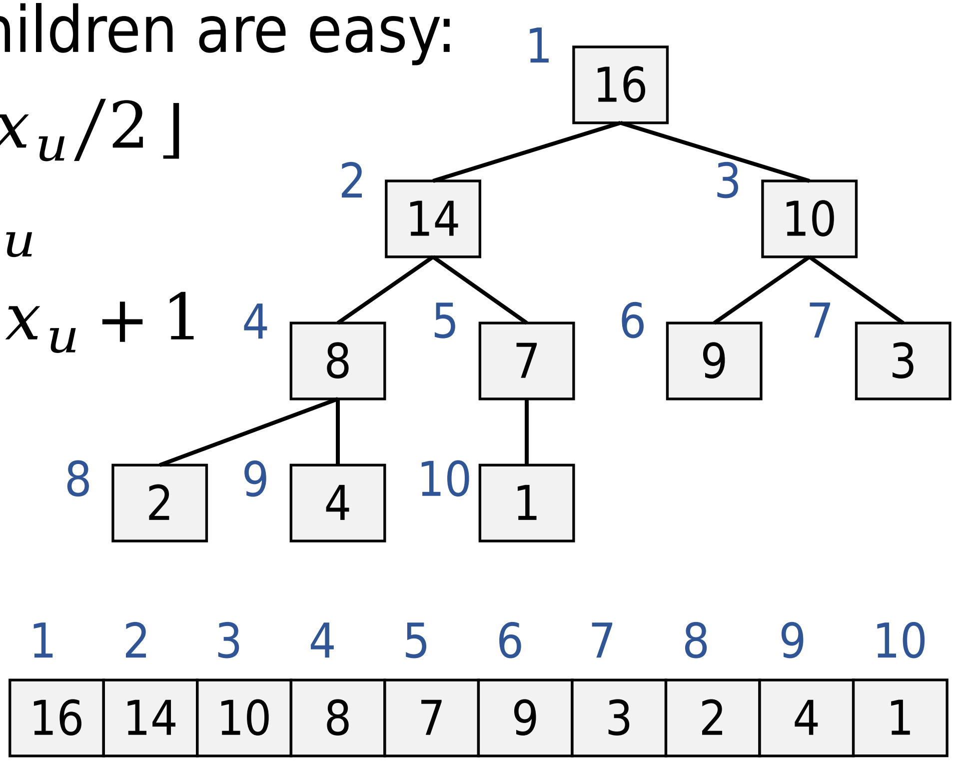
Binary Heap

- We can use an array to represent a binary heap.

Obtaining parent and children are easy:

- Parent of node u: $\lfloor idx_u/2 \rfloor$
- Left child of u: $2 \cdot idx_u$
- Right child of u: $2 \cdot idx_u + 1$

All in $O(1)$ time!



Common operations of

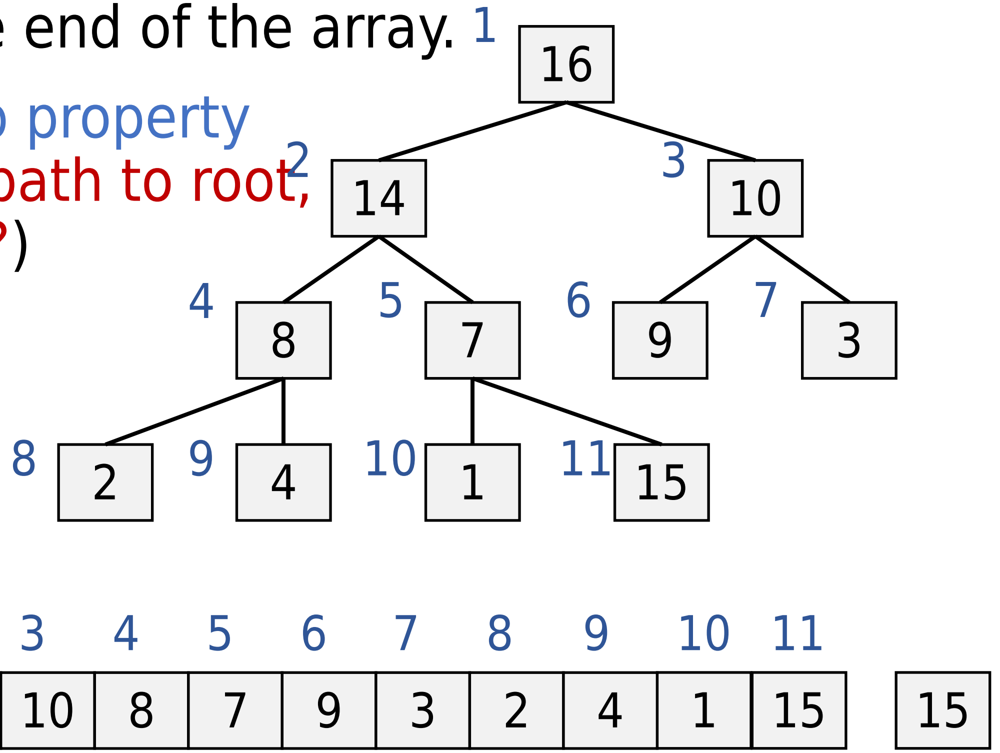
Binary Max-Heap

- Consider max-heap as an example. (Min-heap is similar.)
- Most common operations:
  - **HeapInsert:** insert an element into the heap. Runtime is $O(1)$
  - **HeapGetMax:** return the item with maximum value.
  - **HeapExtractMax:** remove the item with maximum value from the heap and return it.
- Other operations (which we'll see later)…

Operations of binary max-heap
HeapInsert

Insert an item into a binary max-heap represented by an array.

- Simply put the item to the end of the array.
- We need to maintain heap property after insertion: along the path to root, compare and swap. (Why?)

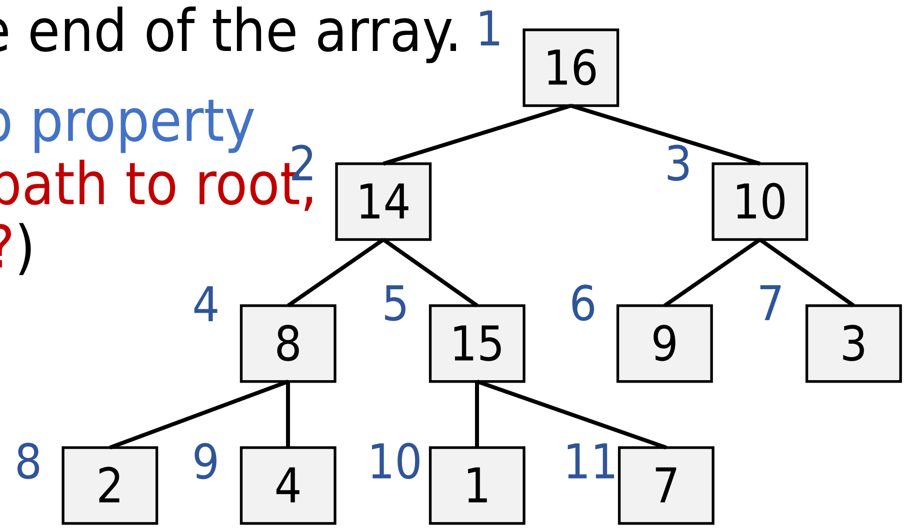


Operations of binary max-heap

HeapInsert

Insert an item into a binary max-heap represented by an array.

- Simply put the item to the end of the array.
- We need to maintain heap property after insertion: along the path to root, compare and swap. (Why?)



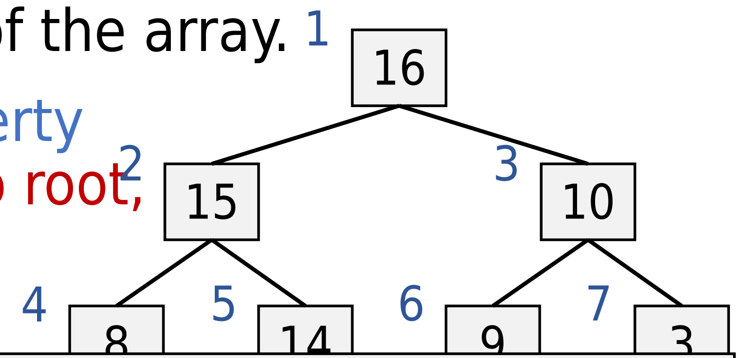
1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	15	9	3	2	4	1	7

Operations of binary max-heap

HeapInsert

Insert an item into a binary max-heap represented by an array.

- Simply put the item to the end of the array.
- We need to maintain heap property after insertion: along the path to root, compare and swap. (Why?)
- Runtime is $O(\lg n)$.



HeapInsert(x):

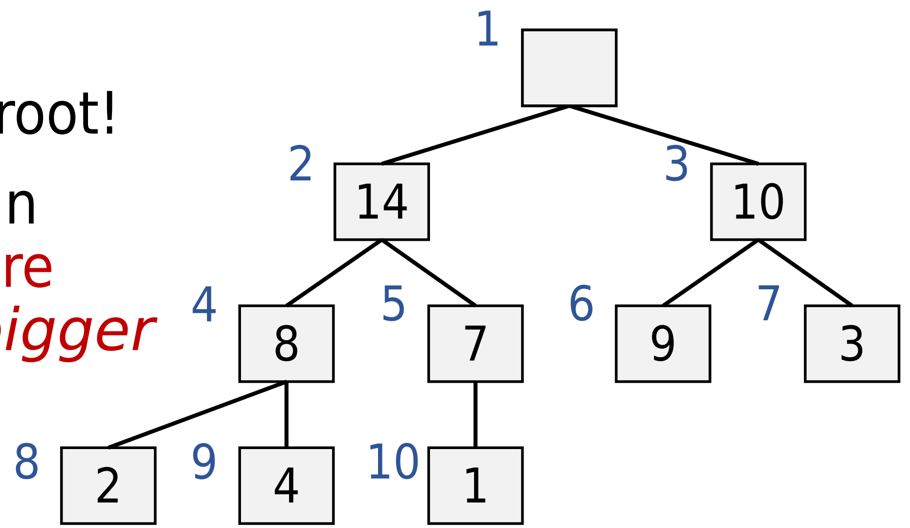
```
heap_size++
data[heap_size] = x
idx = heap_size
while (idx>1 and data[Floor(idx/2)]<data[idx])
  Swap(data[Floor(idx/2)], data[idx])
  idx = Floor(idx/2)
```

Operations of binary max-heap

HeapExtractMax

Remove the maximum item from the heap and return it.

- Remove and return root is simple, but then what to do?!
- Move the last item to the root!
- Again, we need to maintain the heap property: compare with children, swap with *bigger* one; do this recursively.



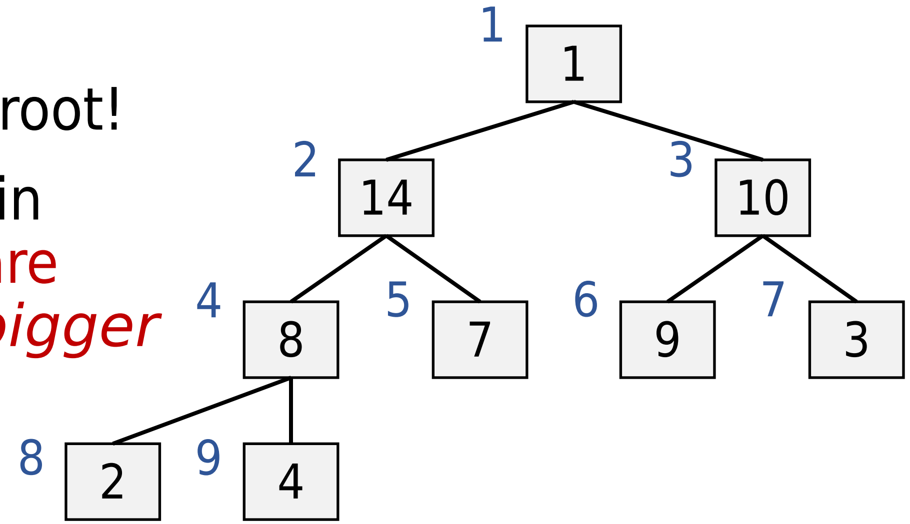
1	2	3	4	5	6	7	8	9	10
	14	10	8	7	9	3	2	4	1

Operations of binary max-heap

HeapExtractMax

Remove the maximum item from the heap and return it.

- Remove and return root is simple, but then what to do?!
- Move the last item to the root!
- Again, we need to maintain the heap property: compare with children, swap with *bigger* one; do this recursively.



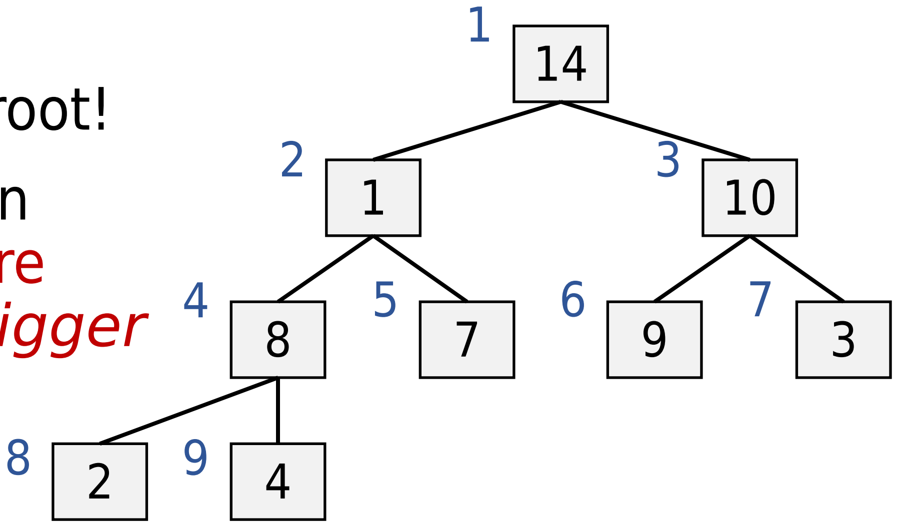
1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4

Operations of binary max-heap

HeapExtractMax

Remove the maximum item from the heap and return it.

- Remove and return root is simple, but then what to do?!
- Move the last item to the root!
- Again, we need to maintain the heap property: compare with children, swap with *bigger* one; do this recursively.



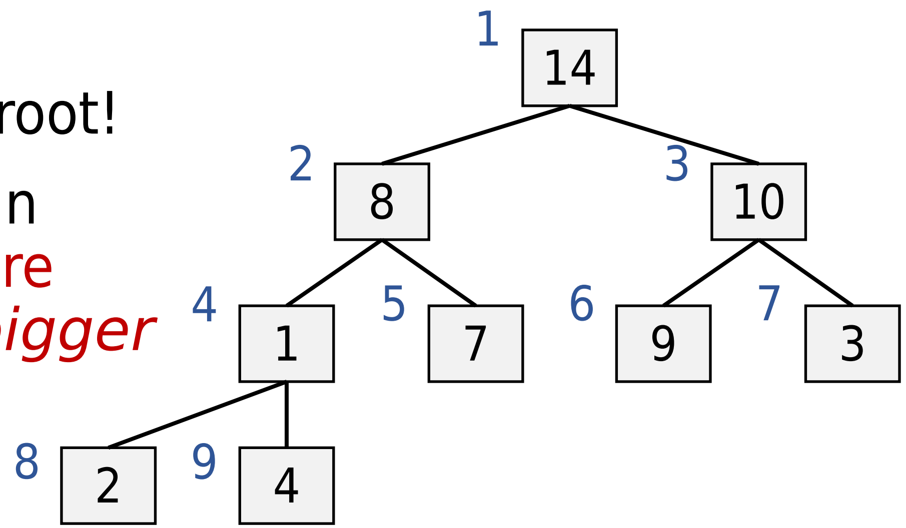
1	2	3	4	5	6	7	8	9
14	1	10	8	7	9	3	2	4

Operations of binary max-heap

HeapExtractMax

Remove the maximum item from the heap and return it.

- Remove and return root is simple, but then what to do?!
- Move the last item to the root!
- Again, we need to maintain the heap property: compare with children, swap with *bigger* one; do this recursively.



1	2	3	4	5	6	7	8	9
14	8	10	1	7	9	3	2	4

Operations of binary max-heap

HeapExtractMax

Remove the maximum item from the heap and return it.

Runtime is $O(\lg n)$

```
HeapExtractMax():
max_item = data[1]
data[1] = data[heap_size--]
MaxHeapify(1)
return max_item

MaxHeapify(idx):
idx_l = 2*i, idx_r = 2*i+1
idx_max = (idx_l<=heap_size && data[idx_l]>data[idx])?
            idx_l:idx
idx_max = (idx_r<=heap_size && data[idx_r]>data[idx_max])?
            idx_r:idx_max
if (idx_max != idx)
  Swap(data[idx_max], data[idx])
  MaxHeapify(idx_max)
```

Applications of heaps

Priority Queue

Recall the Queue ADT represents a collection of items to which we can **add** items and **remove** the next item.

- Add(item): add $item$ to the queue.
- Remove(): remove the next item y from queue, return y.

The *queuing discipline* decides which item to be removed.

- First-in-first-out queue (FIFO Queue)
- Last-in-first-out queue (LIFO Queue, Stack)
- **Priority queue**: each item associated with a priority, Remove always deletes the item with max (or min) priority.

Applications of heaps
Priority Queue

- Use binary heap to implement priority queue
  - Add(item): HeapInsert(item)
  - Remove(): HeapExtractMax()
  - Other operations: GetMax(), UpdatePriority(item,val)
  - All these operations finish within $O(\lg n)$ time

- Applications of priority queues
  - Event simulation, scheduling, …
  - Used in more sophisticated algorithms (and we'll see some of them)

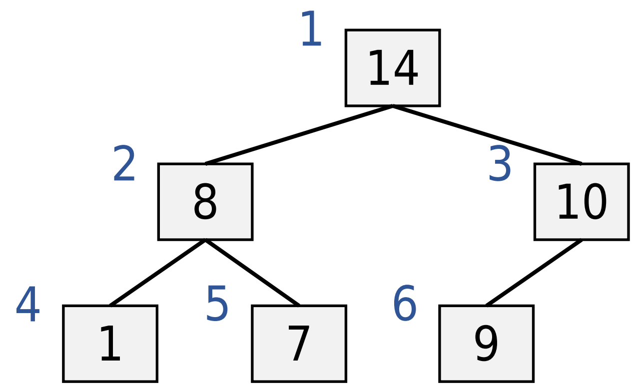
Applications of heaps

HeapSort

```
HeapSort(data[1…n]):
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

Take an array and make it a max-heap.

1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item



8	1	9	14	7	10
1	2	3	4	5	6
14	8	10	1	7	9

In each iteration:

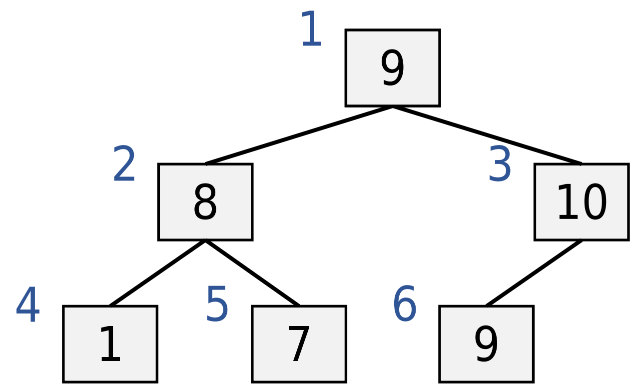
- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

HeapSort

```
HeapSort(data[1…n]):
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item



$i = 6$

1	2	3	4	5	6
9	8	10	1	7	9

14

In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

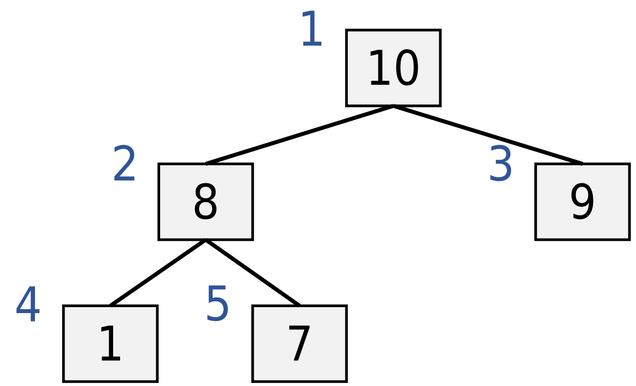
HeapSort

HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

In each iteration:
- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.



$i = 6$

1	2	3	4	5	6
10	8	9	1	7	14

14

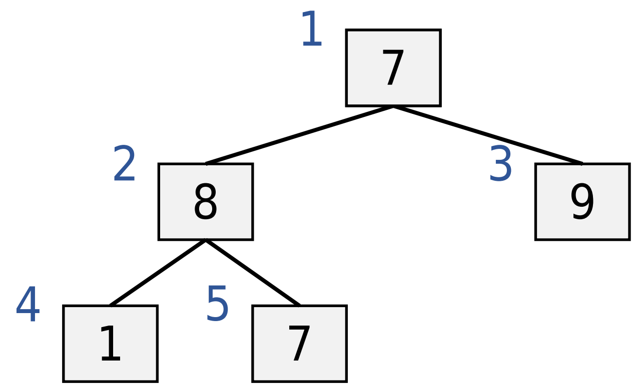
Applications of heaps

HeapSort

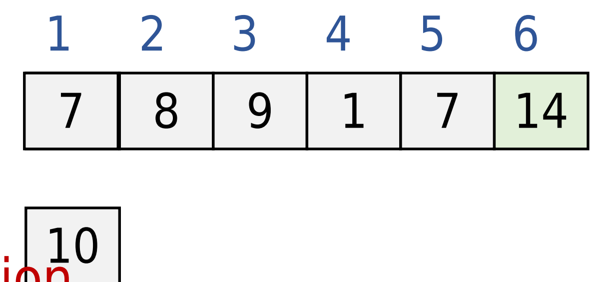
HeapSort(data[1...n]):

```
heap = BuildMaxHeap(data[1...n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item



$i = 5$



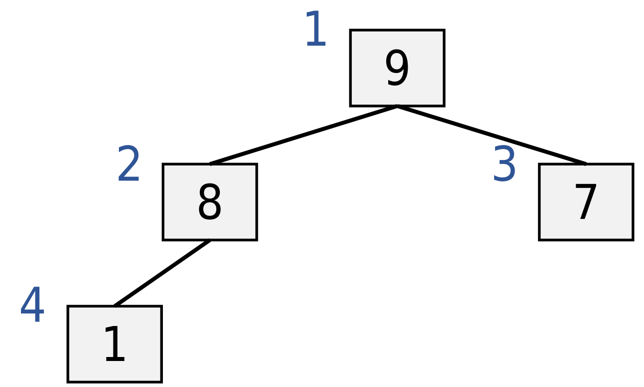
In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

HeapSort

```
HeapSort(data[1…n]):
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```



1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

$i = 5$

1	2	3	4	5	6
9	8	7	1	10	14

10

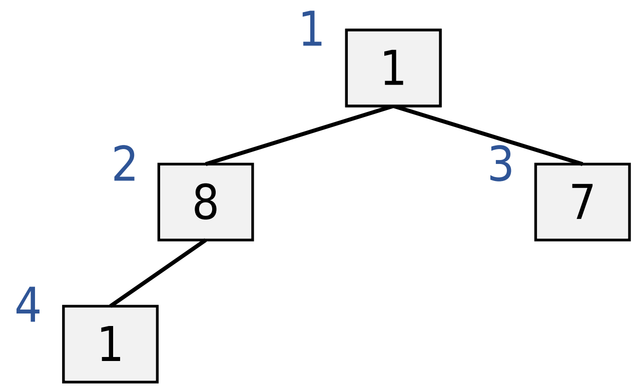
In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

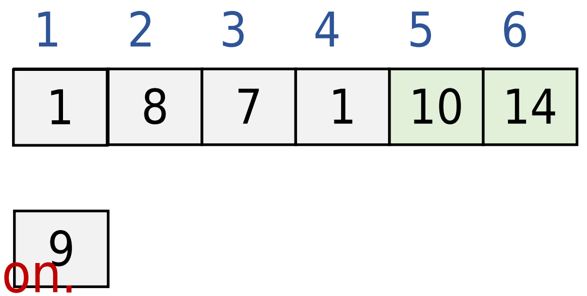
Applications of heaps
HeapSort

HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```



$i = 4$



1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

In each iteration:

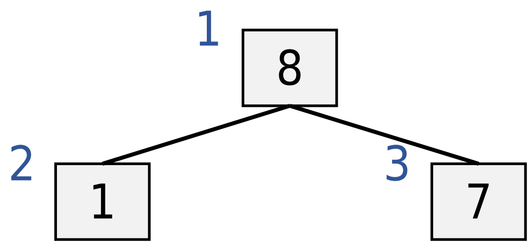
- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

HeapSort

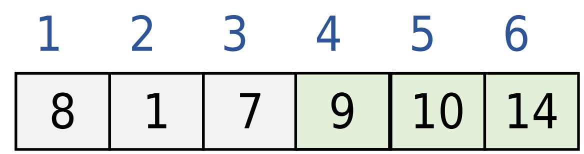
HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```



1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

$i = 4$



In each iteration:

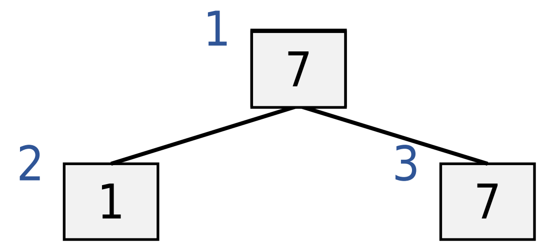
- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

HeapSort

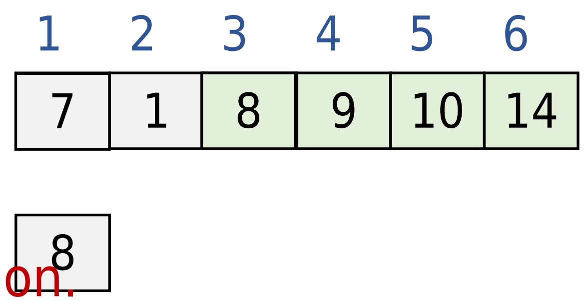
HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```



1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

$i = 3$



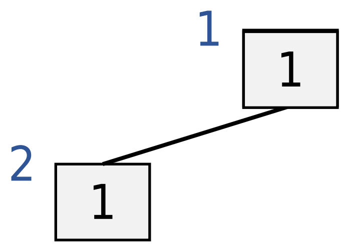
In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

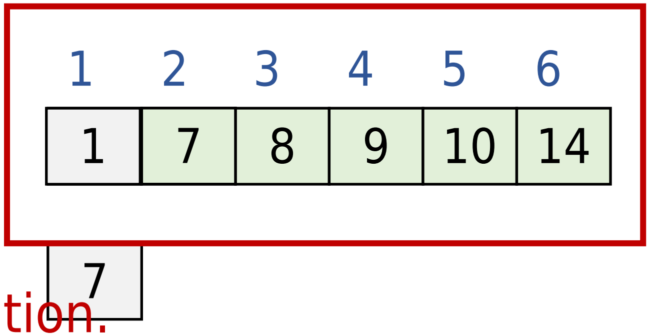
HeapSort

```
HeapSort(data[1…n]):
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```



1. Keep a copy of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the copy of the root item

$i = 2$



In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Applications of heaps

HeapSort

HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

In each iteration:

- Place one item in the array to its final position.
- Place max item in current heap to its final position.
- Place i^{th} biggest item to position $n - i + 1$.

Total runtime of these iterations:

$$\sum_{i=2}^{n} O(\lg i) = O(\lg(n!)) = O(n \lg n)$$

Applications of heaps

HeapSort

HeapSort(data[1…n]):

```
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

Runtime of for-loop is $O(n \lg n)$.

Given an array $data[1 \ldots n]$, how to build a max-heap?

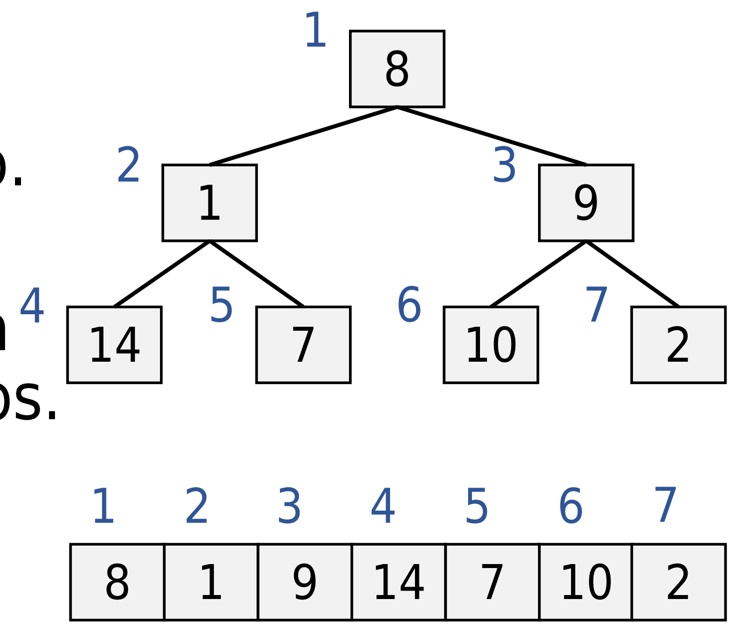
- Start with an empty heap, then call HeapInsert n times.
- Cost is $\sum_{i=1}^{n} O(\lg i) = O(n \lg n)$.
- Not bad, but we can do better…

Applications of heaps

HeapSort

- Given an array $data[1...n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.

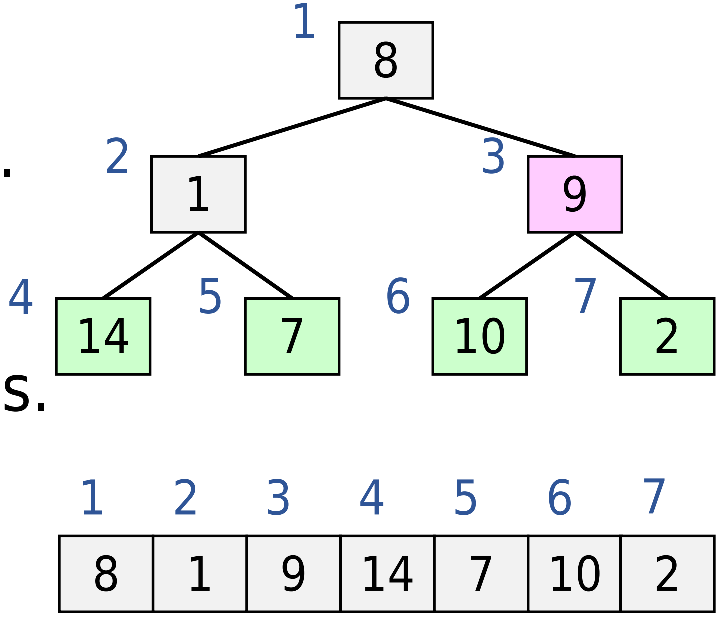
- Each leaf node is a 1-item heap.
- Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.



Applications of heaps

HeapSort

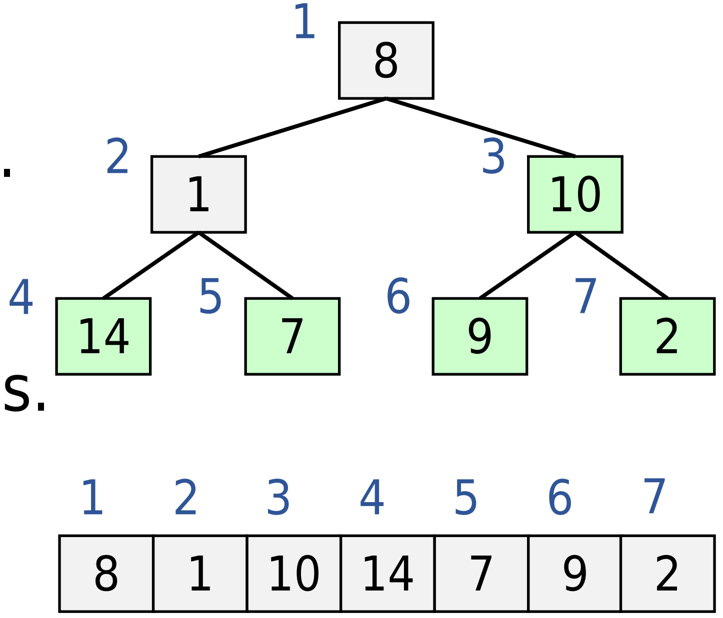
- Given an array $data[1...n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
- Each leaf node is a 1-item heap.
- Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
- Maintain heap property during merging: use MaxHeapify.



Applications of heaps

HeapSort

- Given an array $data[1...n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.

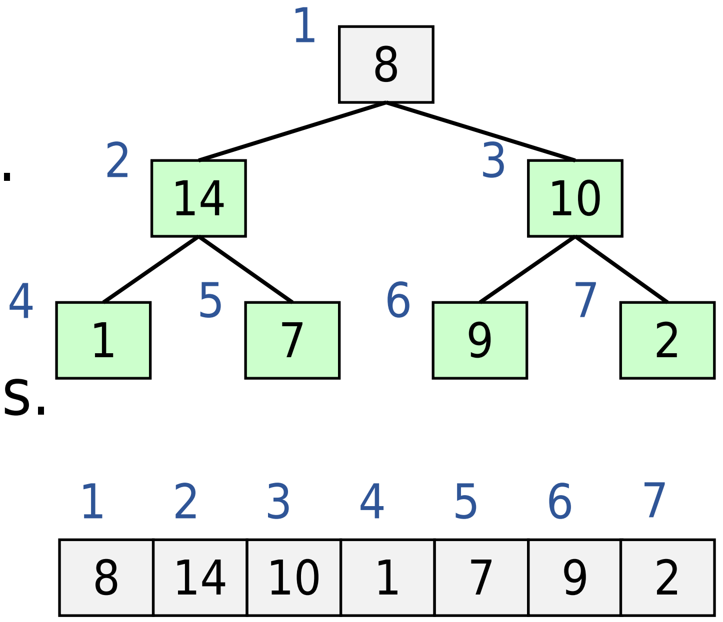


- Each leaf node is a 1-item heap.
- Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
- Maintain heap property during merging: use MaxHeapify.

Applications of heaps

HeapSort

- Given an array $data[1...n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
- Each leaf node is a 1-item heap.
- Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
- Maintain heap property during merging: use MaxHeapify.



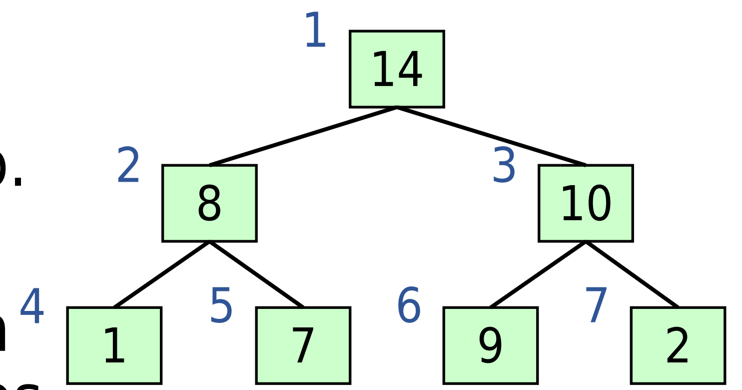
Applications of heaps

HeapSort

BuildMaxHeap(data[1…n]):

```
heap_size = n
for i=Floor(n/2) down to 1
  MaxHeapify(i)
```

- Given an array $data[1 \ldots n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
- Each leaf node is a 1-item heap.
- Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
- Maintain heap property during merging: use MaxHeapify.



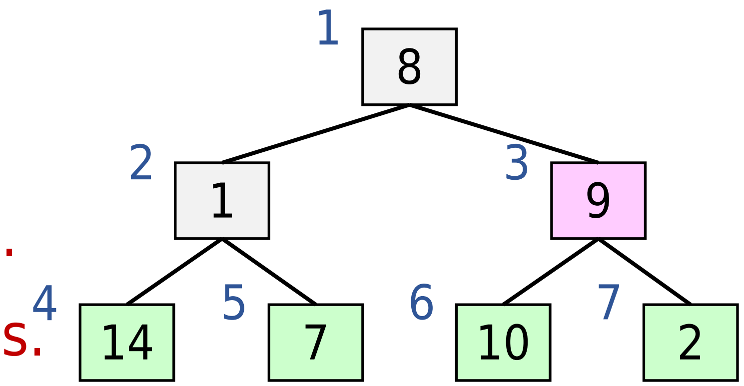
1	2	3	4	5	6	7
14	8	10	1	7	9	2

Applications of heaps
HeapSort

BuildMaxHeap(data[1…n]):

```
heap_size = n
for i=Floor(n/2) down to 1
  MaxHeapify(i)
```

- Given an array $data[1...n]$, how to build a max-heap?
- Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
- Time complexity of BuildMaxHeap?
  - $\Theta(n)$ calls to MaxHeapify, each costing $O(\lg n)$, so $O(n \lg n)$?
  - Correct but not tight…



- Height of n-items heap is $\lfloor \lg n \rfloor$.
- Any height h has $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes.
- Cost of all MaxHeapify: $\sum_{h=0}^{\lfloor \lg n \rfloor} \left(\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \right) = O\left(n \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O(n)$

Applications of heaps

HeapSort

```
BuildMaxHeap(data[1…n]):
heap_size = n
for i=Floor(n/2) down to 1
  MaxHeapify(i)
```

Runtime of BuildMaxHeap is $O(n)$.

```
HeapSort(data[1…n]):
heap = BuildMaxHeap(data[1…n])
for i=n down to 2
  cur_max = heap.HeapExtractMax()
  data[i] = cur_max
```

Runtime of for-loop is $O(n\lg n)$.

Time complexity of HeapSort is $O(n\lg n)$,
extra space required during execution is $O(1)$.

Reading

- [CLRS] Ch.6