Data Structures
Heaps

Andres Mendez-Vazquez

November 12, 2016

Outline

1. Heaps
   - Definitions
   - Finding Parents and Children
   - Max-Heapify
   - Build Max Heap: Using Max-Heapify

2. Applications of Heap Data Structure
   - Heap Sort
   - Priority Queues

Outline

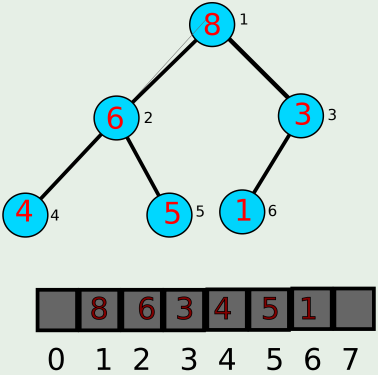
1 Heaps
- Definitions
- Finding Parents and Children
- Max-Heapify
- Build Max Heap: Using Max-Heapify

2 Applications of Heap Data Structure
- Heap Sort
- Priority Queues

Definition of a Heap

Definition

A heap is an array object that can be viewed as a nearly complete binary tree.



Heap: Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap-size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap-size[A] \leq length[A] \quad (1)$$

Heap: Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap - size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap - size[A] \leq length[A] \tag{1}$$

Heap: Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap - size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap - size[A] \leq length[A] \tag{1}$$

Outline

1. Heaps
   - Definitions
   - Finding Parents and Children
   - Max-Heapify
   - Build Max Heap: Using Max-Heapify
2. Applications of Heap Data Structure
   - Heap Sort
   - Priority Queues

Finding Parent and Children given a Node i in the heap

$Parent(i)$ - Parent Node

$Parent(i) = \lfloor \frac{i}{2} \rfloor$

Left Node Child: $Left(i)$

$Left(i) = 2i$

Right Node Child: $Right(i)$

$Right(i) = 2i + 1$

Finding Parent and Children given a Node i in the heap

$Parent(i)$ - Parent Node

$Parent(i) = \lfloor \frac{i}{2} \rfloor$

Left Node Child: $Left(i)$

$Left(i) = 2i$

Right Node Child: $Right(i)$

$Right(i) = 2i + 1$

Finding Parent and Children given a Node i in the heap

$Parent(i)$ - Parent Node

$Parent(i) = \lfloor \frac{i}{2} \rfloor$

Left Node Child: $Left(i)$

$Left(i) = 2i$

Right Node Child: $Right(i)$

$Right(i) = 2i + 1$

Heap's Properties

Given that

$A[i]$ returns the value of the key, we have that

Max heap property

$A[Parent(i)] \geq A[i]$

Min heap property

$A[Parent(i)] \leq A[i]$

Heap's Properties

Given that

$A\left[i\right]$ returns the value of the key, we have that

Max heap property

$A[Parent(i)] \geq A[i]$

Min heap property

$A[Parent(i)] \leq A[i]$

Heap's Properties

Given that

$A\left[i\right]$ returns the value of the key, we have that

Max heap property

$A[Parent(i)] \geq A[i]$

Min heap property

$A[Parent(i)] \leq A[i]$

The ADT Heap

Interface

```
public interface MaxHeapInterface<T extends Comparable<? super T
{
  public void add(T newEntry);
  public T removeMax();
  public T getMax();
  public boolean isEmpty();
  public int getSize();
  public void clear();
} // end MaxHeapInterface
```

What is "?"?

This is coming from the idea of wildcards

For example if we have:

- void printCollection(Collection<Object> c) { for (Object e : c) { System.out.println(e); } }

We do not have a generic Collection!!!

So we write Collection<?>

Collection of unknowns...

What is “?”?

This is coming from the idea of wildcards

For example if we have:

- void printCollection(Collection<Object> c) { for (Object e : c) { System.out.println(e); } }

We do not have a generic Collection!!!

So we write Collection<?>

Collection of unknowns...

Outline

1. Heaps
   - Definitions
   - Finding Parents and Children
   - Max-Heapify
   - Build Max Heap: Using Max-Heapify

2. Applications of Heap Data Structure
   - Heap Sort
   - Priority Queues

What we want!!!

A function to keep the property of max or min heap

After all, remembering Kolmogorov, we are acting in a part of the array trying to keep certain properties

- Which ONE?

Important

What we want!!!

A function to keep the property of max or min heap

After all, remembering Kolmogorov, we are acting in a part of the array trying to keep certain properties

- Which ONE?

Important

Single nodes are always min heaps or max heaps

Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

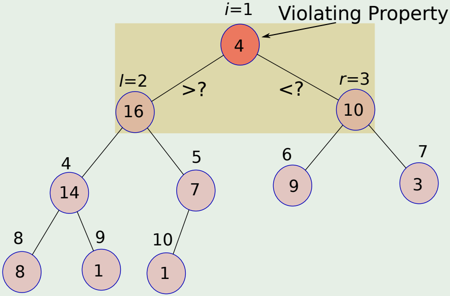
1. $l = Left(i)$
2. $r = Right(\text{i})$
3. If $l \leq heap - size\,[A]$ and $A\,[l] > A\,[i]$
4. $\quad\quad largest = l$
5. else $largest = i$
6. If $r \leq heap - size\,[A]$ and $A\,[r] > A\,[largest]$
7. $\quad\quad largest = r$
8. if $largest \neq i$
9. $\quad\quad$ exchange $A[i]$ with $A[largest]$
10. $\quad\quad$ **Max-Heapify**$(A, largest)$

Figure: A trickle down algorithm

Example keeping the heap property starting at $i = 1$

Here, you could imagine that somebody inserted a node at $i = 1$

3. If $l \leq heap - size\left[A\right]$ and $A\left[l\right] > A\left[i\right]$
4. $\quad largest = l$
5. else $largest = i$
6. If $r \leq heap - size\left[A\right]$ and $A\left[r\right] > A\left[largest\right]$
7. $\quad largest = r$

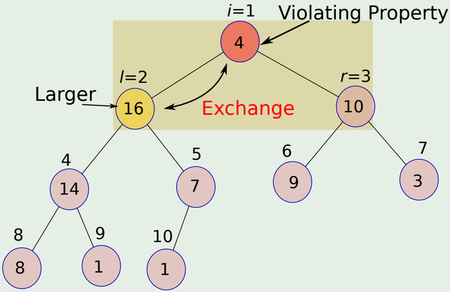


	4	16	10	14	7	9	3	8	1	1
0	1	2	3	4	5	6	7	8	9	10

Example keeping the heap property starting at $i = 1$

One of the children is chosen to be exchanged

8. if $largest \neq i$
9. exchange $A[i]$ with $A[largest]$

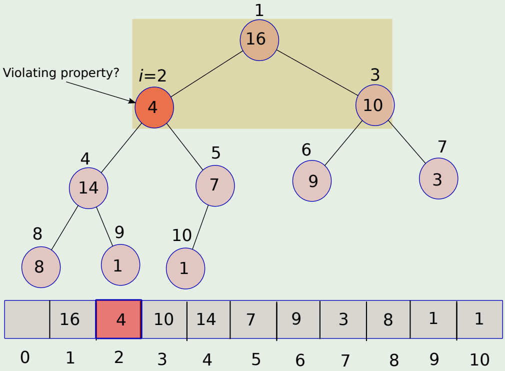


	4	16	10	14	7	9	3	8	1	1
0	1	2	3	4	5	6	7	8	9	10

Example: Now $i = largest$

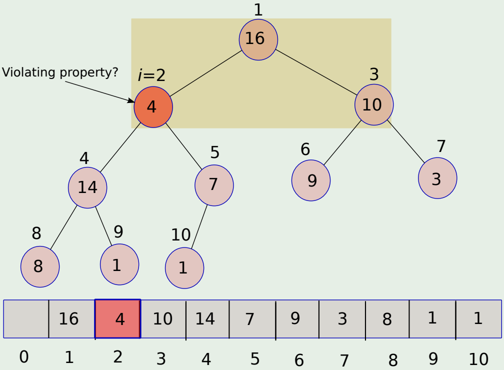
Make the excahnge and call the **Max-Heapify**

10. **Max-Heapify**$(A, largest)$



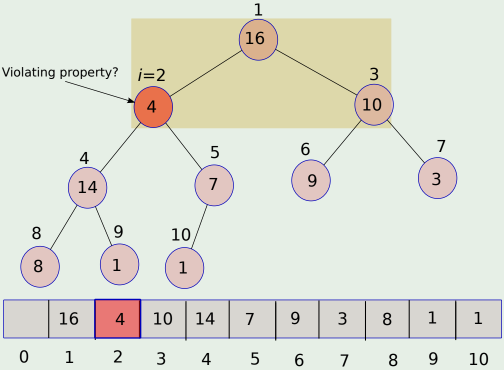
Example: Now $i = largest$

Keep going



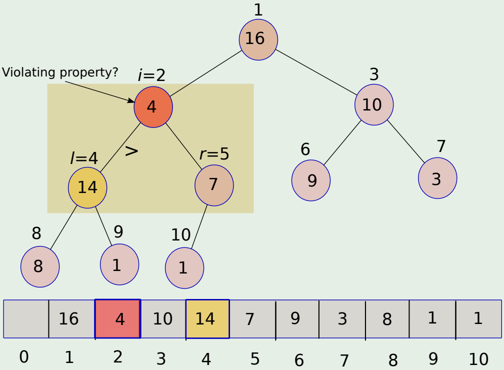
Example: Now $i = largest$

Keep going



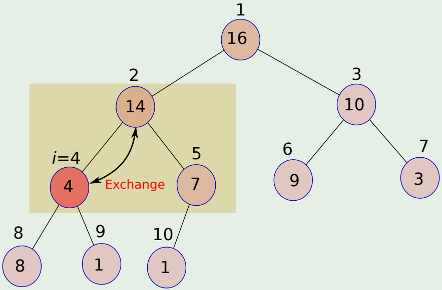
Example: Now $i = largest$

Keep going



Example: Now $i = largest$

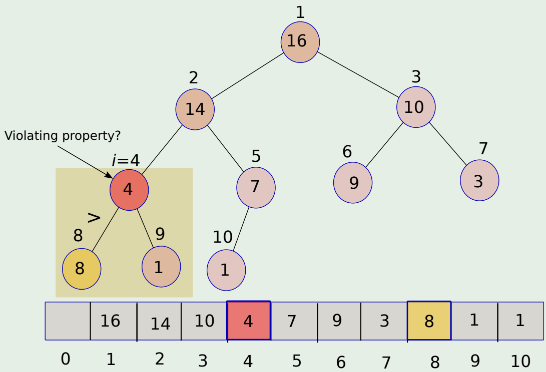
Keep going



	16	14	10	4	7	9	3	8	1	1
0	1	2	3	4	5	6	7	8	9	10

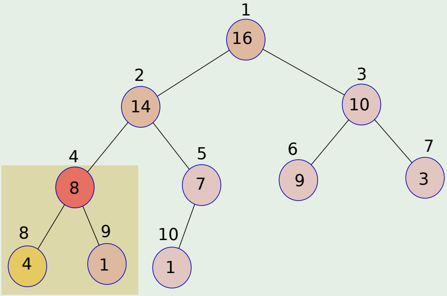
Example: Now $i = largest$

Keep going



Example: Now $i = largest$

Keep going



0	1	2	3	4	5	6	7	8	9	10
	16	14	10	8	7	9	3	4	1	1

Complexity of Max-Heapify

Algorithm Complexity

$O(\log n)$.

Outline

1. Heaps
   - Definitions
   - Finding Parents and Children
   - Max-Heapify
   - Build Max Heap: Using Max-Heapify

2. Applications of Heap Data Structure
   - Heap Sort
   - Priority Queues

Example: Using Max-Heapify

Algorithm Build-Max-Heap

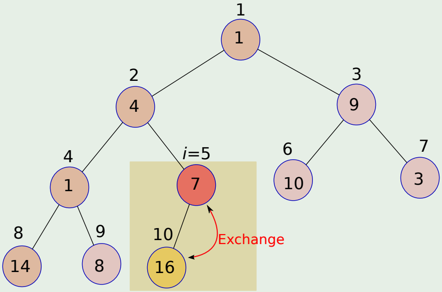
Build-Max-Heap(A, i)

1. $heap - size[A] = length[A]$
2. for $i = \lfloor {}^{length[A]}/2 \rfloor$ downto 1
3. **Max-Heapify**(A, i)

Figure: Building a Heap

Build Max Heap: Using Max-Heapify

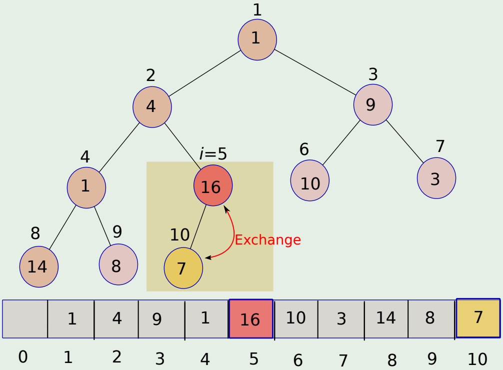
Example



	1	4	9	1	7	10	3	14	8	16
0	1	2	3	4	5	6	7	8	9	10

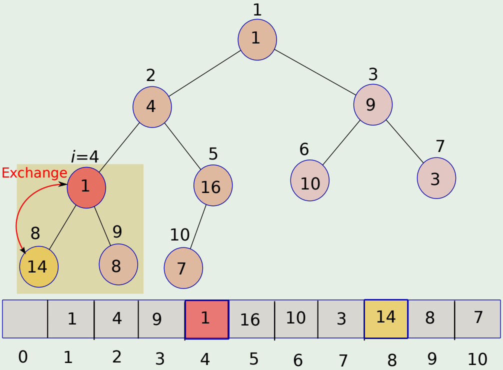
Build Max Heap: Using Max-Heapify

Example



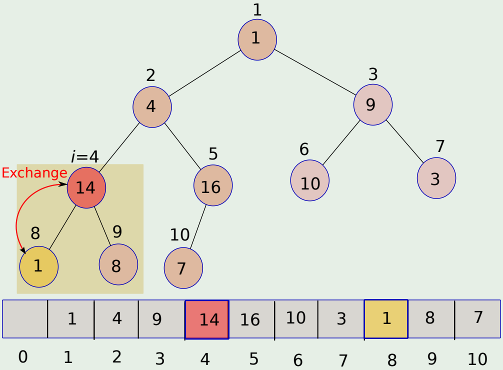
Build Max Heap: Using Max-Heapify

Example



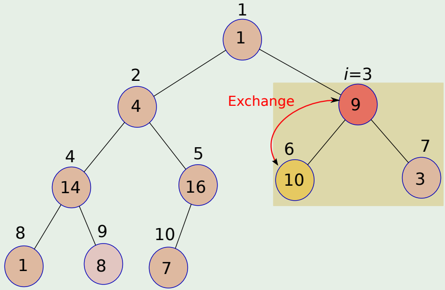
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

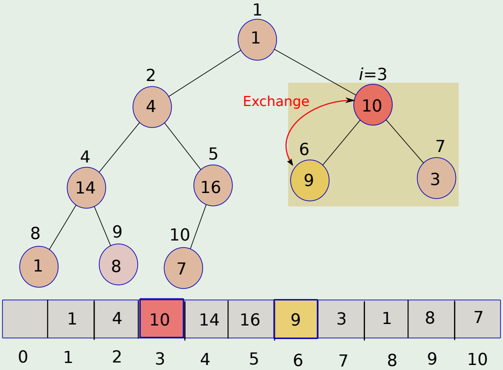
Example



0	1	2	3	4	5	6	7	8	9	10
	1	4	9	14	16	10	3	1	8	7

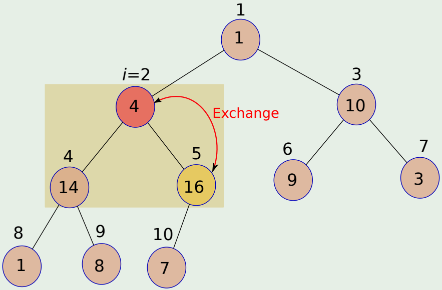
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

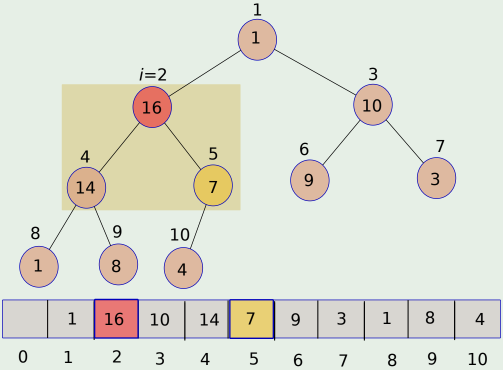
Example



0	1	2	3	4	5	6	7	8	9	10
	1	4	10	14	16	9	3	1	8	7

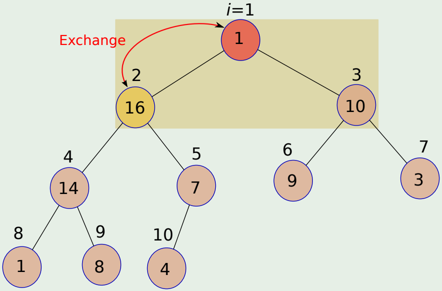
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

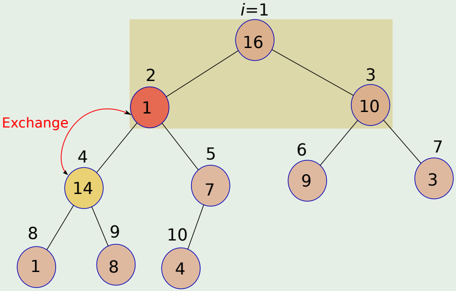
Example



	1	16	10	14	7	9	3	1	8	4
0	1	2	3	4	5	6	7	8	9	10

Build Max Heap: Using Max-Heapify

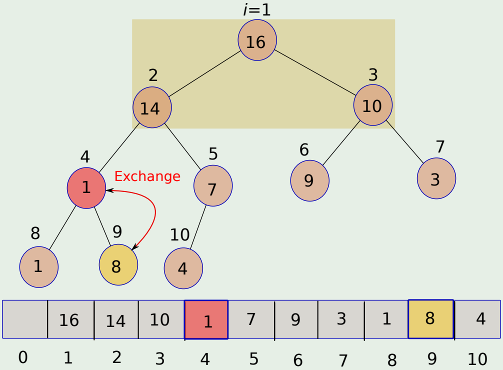
Example



	16	1	10	14	7	9	3	1	8	4
0	1	2	3	4	5	6	7	8	9	10

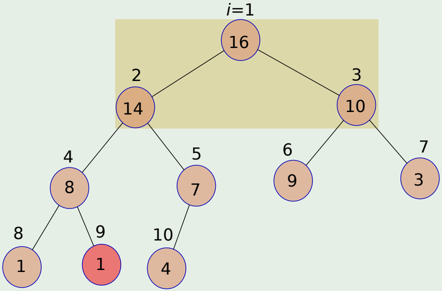
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

Example

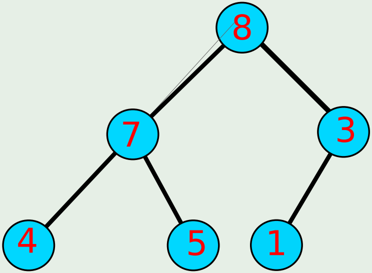


0	1	2	3	4	5	6	7	8	9	10
	16	14	10	8	7	9	3	1	1	4

Height h of the Heap for Complexity of Build-Max-Heap

We can use the height of a three to derive a tight bound

- The height h is the number of edges on the longest simple downward path from the node to a leaf.
- You have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at any height, where n is the total number of nodes.



h=2 $\left\lceil \frac{6}{2^{2+1}} \right\rceil = 1$

h=1 $\left\lceil \frac{6}{2^{1+1}} \right\rceil = 2$

h=0 $\left\lceil \frac{6}{2^{0+1}} \right\rceil = 3$

Cost of Building the Build-Max-Heap

Cost

$O(n)$

Applications of Heap Data Structure

Heap Sort of Arrays

Clearly, if the list of numbers is stored in an array!!!

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
   1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
   1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Heap Sort of Arrays

Clearly, if the list of numbers is stored in an array!!!

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
   1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
   1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Heap Sort of Arrays

Clearly, if the list of numbers is stored in an array!!!

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
   1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
   1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Outline

1 Heaps
- Definitions
- Finding Parents and Children
- Max-Heapify
- Build Max Heap: Using Max-Heapify

2 Applications of Heap Data Structure
- Heap Sort
- Priority Queues

Sorting: Using Max-Heapify

Heapsort Algorithm

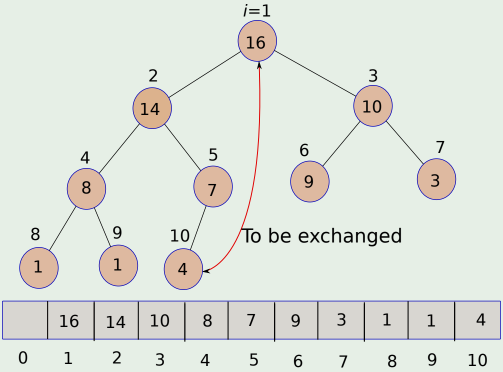
Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5. **Max-Heapify**$(A, 1)$

Figure: Heapsort

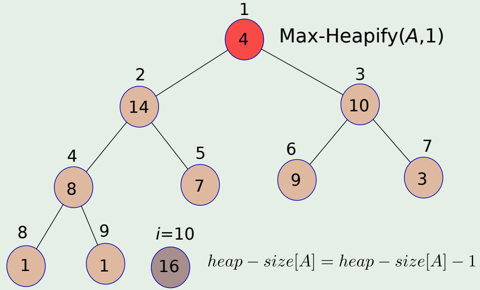
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

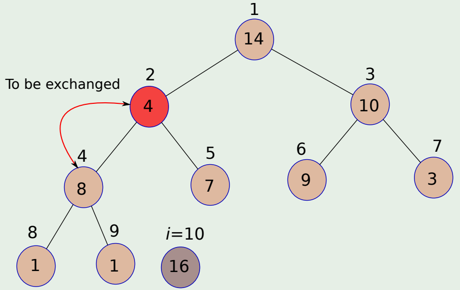
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	4	14	10	8	7	9	3	1	1	16
0	1	2	3	4	5	6	7	8	9	10

Sorting: Using Max-Heapify

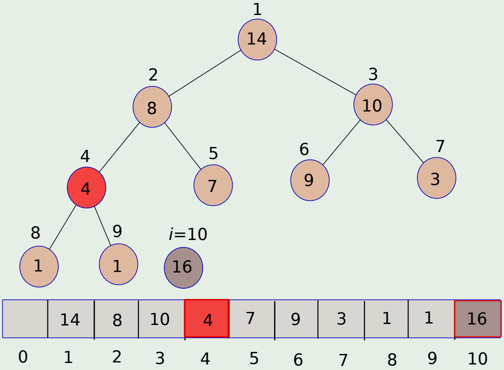
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	14	4	10	8	7	9	3	1	1	16
0	1	2	3	4	5	6	7	8	9	10

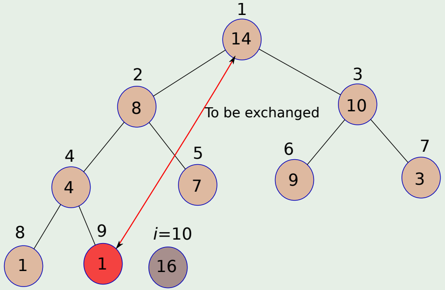
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

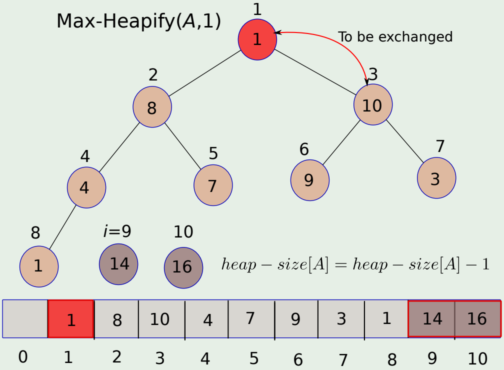
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	14	8	10	4	7	9	3	1	1	16
0	1	2	3	4	5	6	7	8	9	10

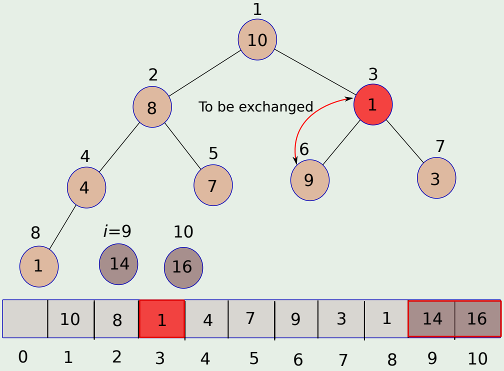
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



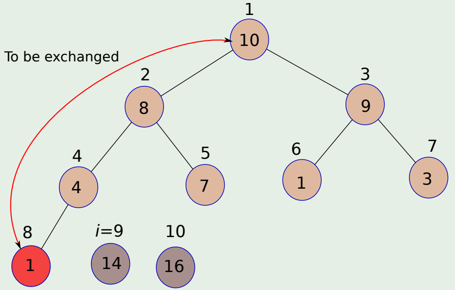
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

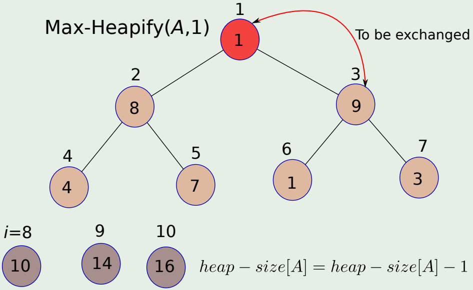
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	10	8	1	4	7	9	3	1	14	16
0	1	2	3	4	5	6	7	8	9	10

Sorting: Using Max-Heapify

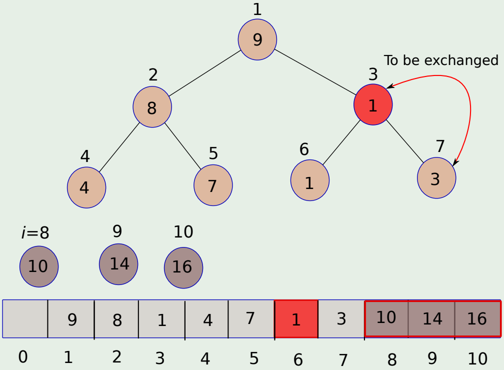
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	1	8	1	4	7	9	3	10	14	16
0	1	2	3	4	5	6	7	8	9	10

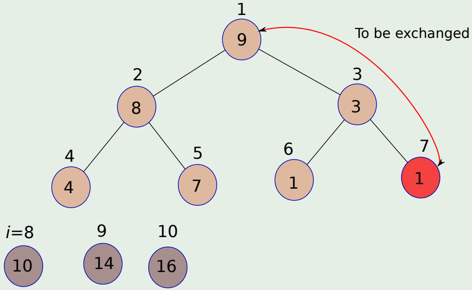
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

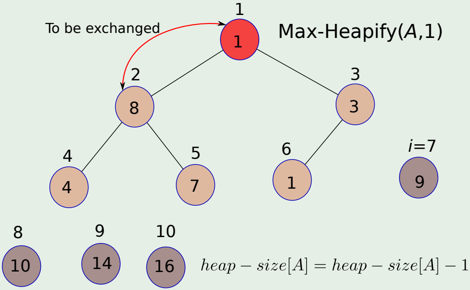
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	9	8	1	4	7	3	1	10	14	16
0	1	2	3	4	5	6	7	8	9	10

Sorting: Using Max-Heapify

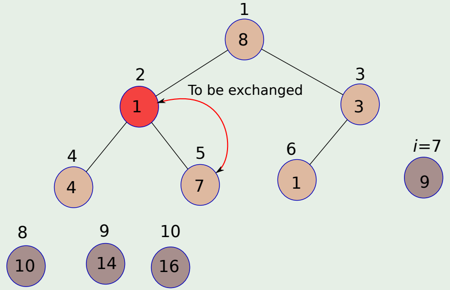
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	1	8	1	4	7	3	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

Sorting: Using Max-Heapify

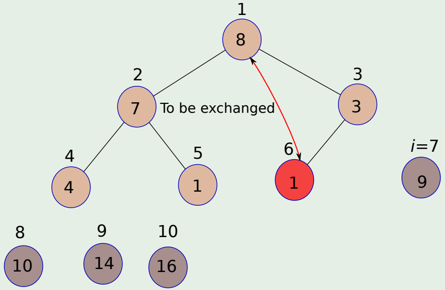
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	8	1	3	4	7	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

Sorting: Using Max-Heapify

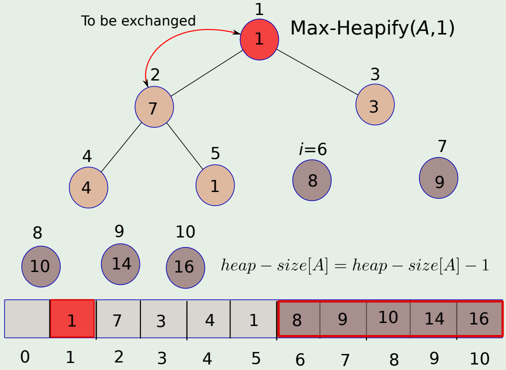
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	8	7	3	4	1	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

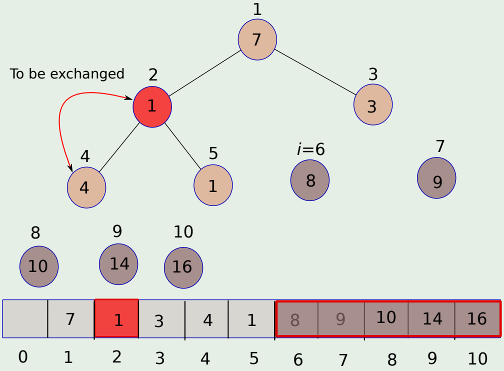
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



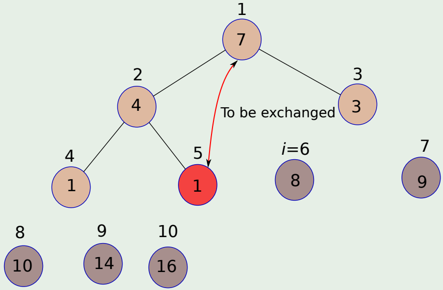
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

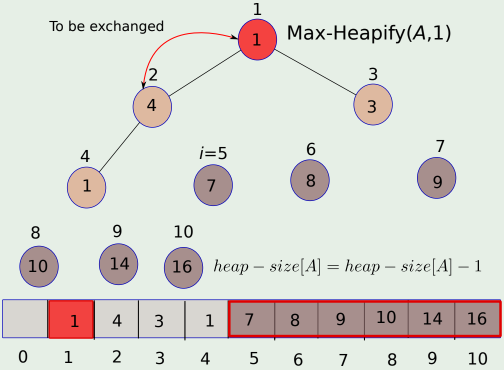
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	7	4	3	1	1	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

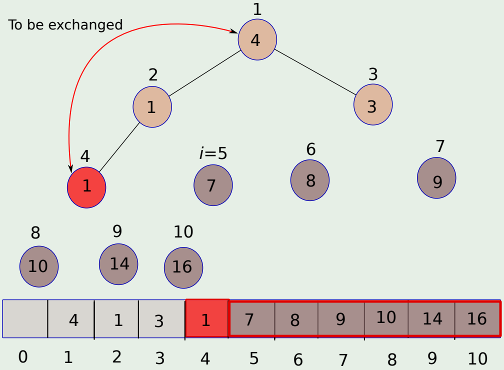
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



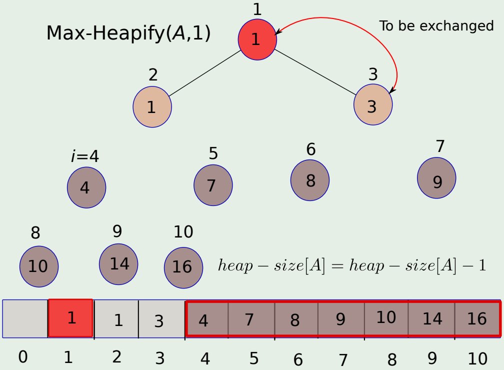
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



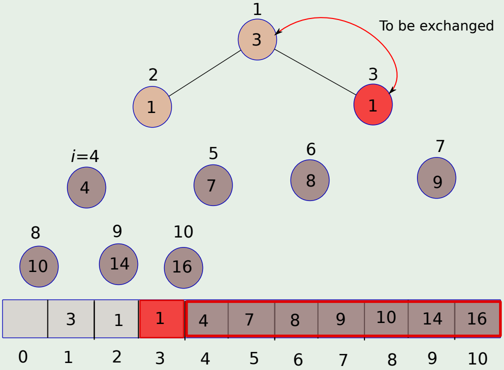
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



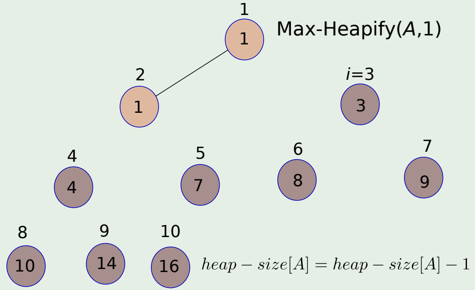
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

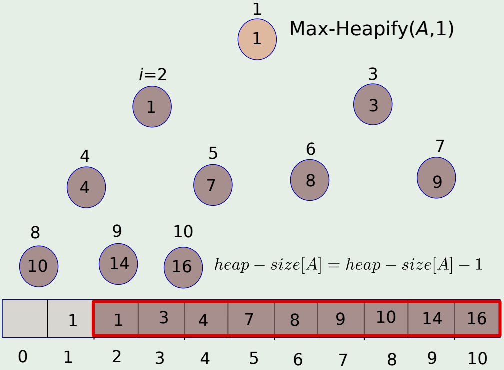
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	1	1	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

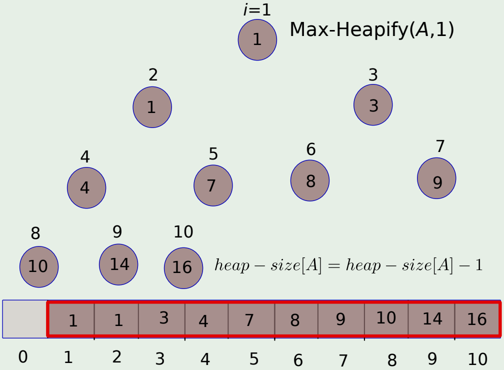
Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Sorting: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Cost of the Heapsort

Cost

$O(n \log n)$

Outline

1 Heaps
- Definitions
- Finding Parents and Children
- Max-Heapify
- Build Max Heap: Using Max-Heapify

2 Applications of Heap Data Structure
- Heap Sort
- Priority Queues

Basic Concepts

Definition

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.

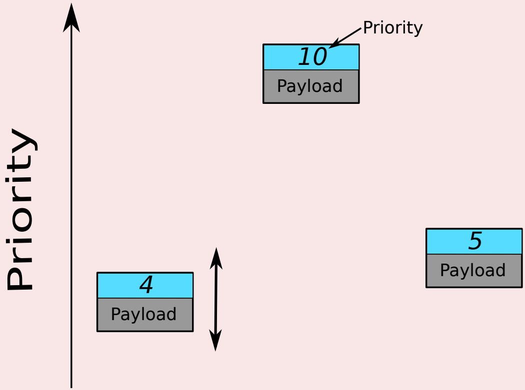
We use this priority

Basic Concepts

Definition

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.

We use this priority



Clearly, you could sort the elements by priorities

Cost of that

$$O\left(n \log n\right) \tag{2}$$

We want something better!!!

After all that is what we do when designing data structures

Clearly, you could sort the elements by priorities

Cost of that

$$O(n \log n) \tag{2}$$

We want something better!!!

After all that is what we do when designing data structures

First, the ADT of a Max Priority Queue

ADT of a Max Priority Queue

```
public interface MaxHeapInterface<T extends Comparable<? super T
{
  public void Insert(T newEntry);
  public T Maximum();
  public T Extract-Max();
  public void Increase-Key(T, key)
  public boolean isEmpty();
  public int size();
}
```

Thus, we need to look at the implementations

First, insertion

public void Insert(T newEntry);

First, What do we do?

Second

Where is the best place to put the new key?

Thus, we need to look at the implementations

First, insertion

public void Insert(T newEntry);

First, What do we do?

See if you have enough space in the array!!!

Second

Where is the best place to put the new key?

Thus, we need to look at the implementations

First, insertion

public void Insert(T newEntry);

First, What do we do?

See if you have enough space in the array!!!

Second

Where is the best place to put the new key?

What to do?

Imagine the following $Heap$

	16	10	9	8	7	4				
0	1	2	3	4	5	6	7	8	9	10

Ideas?

What about the following... when we draw the Heap!!!

What to do?

Imagine the following $Heap$

0	1	2	3	4	5	6	7	8	9	10
	16	10	9	8	7	4				

Ideas?

What about the following... when we draw the Heap!!!

Yes

Insert at the end

	16	10	9	8	7	4	17			
0	1	2	3	4	5	6	7	8	9	10

Thus we need to move this up!!!

1. while $i > 1$ and $Heap[Parent(i)] < Heap[i]$
2. exchange $Heap[i]$ with $Heap[Parent(i)]$
3. $i = Parent(i)$

In addition

$Heap.heap-size = Heap.heap-size + 1$

Yes

Insert at the end

	16	10	9	8	7	4	17			
0	1	2	3	4	5	6	7	8	9	10

Thus we need to move this up!!!

1. while $i > 1$ and $Heap\left[Parent\left(i\right)\right] < Heap\left[i\right]$
2. exchange $Heap\left[i\right]$ with $Heap\left[Parent\left(i\right)\right]$
3. $i = Parent\left(i\right)$

In addition

$Heap.heap-size = Heap.heap-size+1$

Yes

Insert at the end

	16	10	9	8	7	4	17			
0	1	2	3	4	5	6	7	8	9	10

Thus we need to move this up!!!

1. while $i > 1$ and $Heap\left[Parent\left(i\right)\right] < Heap\left[i\right]$
2. exchange $Heap\left[i\right]$ with $Heap\left[Parent\left(i\right)\right]$
3. $i = Parent\left(i\right)$

In addition

$Heap.heap - size = Heap.heap - size + 1$

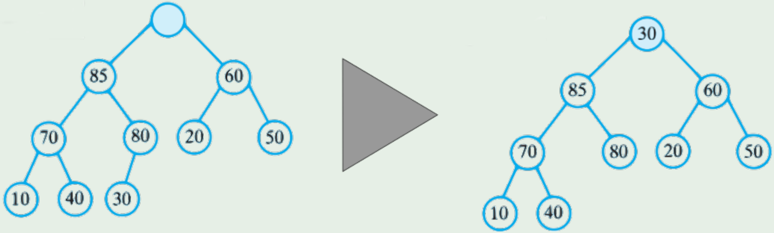
The Pseudo-Code

Insertion

```
Insertion_Max(newEntry){
  if (the  array heap is full)
     Double the size of the array

  newIndex = Heap.heap-size + 1
  parentIndex = Parent(newIndex)

   while (parentIndex > 1 and newEntry > heap[parentIndex])
   {
      heap[newIndex] = heap[parentIndex]
      newIndex = parentIndex
      parentIndex = Parent(newIndex)
   }

   heap[newIndex] = newEntry
}
```

What about Extract-Max

We remove from the top



Here, we can use Max-Heapify

To trickle down as the Max-Heap property is not working

Using the previous code...

Pseudocode

Extract-Max()

1. if $Heap.heap - size < 1$
2. error "heap underflow"
3. max $= Heap[1]$
4. $Heap[1] = Heap[Heap.heap - size]$
5. $Heap.heap - size = Heap.heap - size - 1$
6. **Max-Heapify**(1)
7. return max

What about Heap-Increase-Key?

Here, a design issue

- In a Max Priority Queue you can only increase keys
- In a Min Priority Queue you can only decrease keys

Then

Pseudo-Code

Increase-key(i, key)

1. if $key < Heap\left[i\right]$
2. error "new key is smaller than current key"
3. $Heap\left[i\right] = key$
4. while $i > 1$ and $Heap\left[Parent\left(i\right)\right] < Heap\left[i\right]$
5. exchange $Heap\left[i\right]$ with $Heap\left[Parent\left(i\right)\right]$
6. $i = Parent\left(i\right)$