Heapsort

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

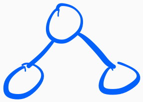
The University of Texas at Arlington

Introduction

Heaps

A **binary heap** can be represented as a binary tree, but is stored as an array.

- The root is the first element of the array.
- The left subnode for the element at index i is located at $2i$ and the right subnode is located at $2i + 1$.
- **This assumes a 1-based indexing.**

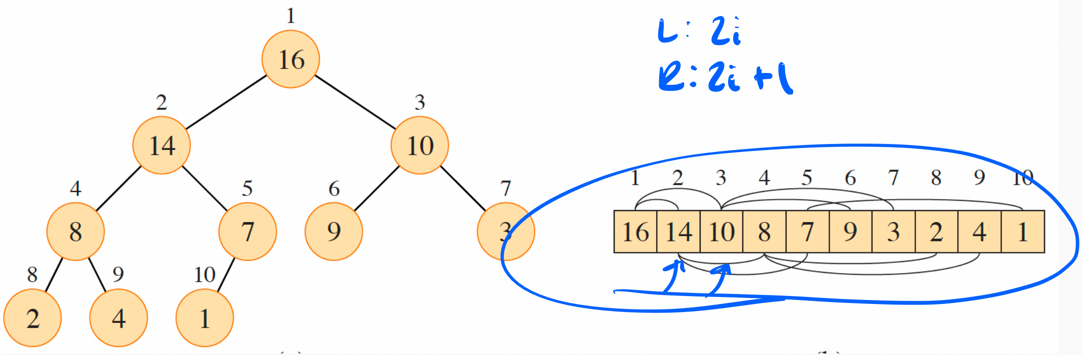


How would this change for 0-based indexing?

How would this change for 0-based indexing?

- $2i + 1$ for the left.
- $2i + 2$ for the right.
- The parent could be accessed via $\lfloor \frac{i-1}{2} \rfloor$.

Heaps



A binary tree as a heap with its array representation (Cormen et al.).

Heaps

Heaps come in two flavors: **max-heaps** and **min-heaps**.

They can be identified by satisfying a **heap property**.

- **max-heap property**: $\boxed{A[parent(i)] \geq A[i]}$
- **min-heap property**: $A[parent(i)] \leq A[i]$

Heaps

For sorting, a **max-heap** is used.

We will later study priority queues, where a **min-heap** is used.

Maintaining the Heap Property

Heap Property

The heap should always satisfy the max-heap property.

- This relies on a procedure called `max_heapify`.
- This assumes that the root element may violate the max-heap property, **but...**

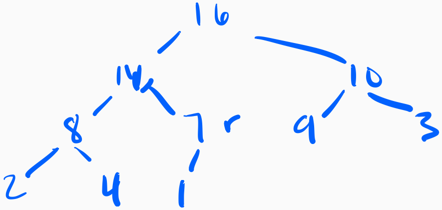
Heap Property

The heap should always satisfy the max-heap property.

- This relies on a procedure called `max_heapify`.
- This assumes that the root element may violate the max-heap property, **but...**
- Assumes subtrees rooted by its subnodes are valid max-heaps.
- Swap nodes down the tree until the misplaced element is in the correct position.

Heapify

```python
def max_heapify(A, i, heap_size):
    l = left(i)
    r = right(i)
    largest = i
    if l < heap_size and A[l] > A[i]:
        largest = l
    if r < heap_size and A[r] > A[largest]:
        largest = r
    if largest != i:
        A[i], A[largest] = A[largest], A[i]
        max_heapify(A, largest, heap_size)
```



Analysis

Given that `max_heapify` is a recursive function, we can analyze it with a recurrence.

- **Driving function:** the fix up that happens between the current node and its two subnodes: $\Theta(1)$.
- **Recurrence:** based on how many elements are in the subheap rooted at the current node.

Analysis

Recurrence

- In the worst case of a binary tree, the last level of the tree is half full.
- The left subtree has height $h + 1$ compared to the right subtree's height of h.
- For a tree of size n, the left subtree has $2^{h+2} - 1$ nodes and the right subtree has $2^{h+1} - 1$ nodes.

Analysis

The number of nodes in the tree is equal to $1 + (2^{h+2} - 1) + (2^{h+1} - 1)$.

$$
\begin{aligned}
n &= 1 + 2^{h+2} - 1 + 2^{h+1} - 1 \\
n &= 2^{h+2} + 2^{h+1} - 1 \\
n &= 2^{h+1}(2 + 1) - 1 \\
n &= 3 \cdot 2^{h+1} - 1
\end{aligned}
$$

Analysis

- This implies that $2^{h+1} = \frac{n+1}{3}$.
- In the worst case, the left subtree would have $2^{h+2} - 1 = \frac{2(n+1)}{3} - 1$ nodes which is bounded by $\frac{2n}{3}$.
- The recurrence for the worst case of `max_heapify` is $T(n) = T(\frac{2n}{3}) + O(1)$.

Building the Heap

Building the Heap

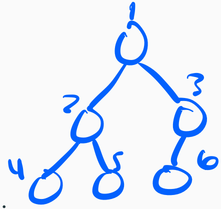
Given an array of elements, **how do we build the heap in the first place?**

Building the Heap

Given an array of elements, **how do we build the heap in the first place?**

Use a bottom-up approach from the leaves.

Building the Heap



- The elements from $\lfloor \frac{n}{2} \rfloor + 1$ to n are all leaves.
- This means that they are all 1-element heaps.
- Run `max_heapify` on the remaining elements to build the heap.

Building the Heap

```python
def build_max_heap(A):
    heap_size = len(A)
    for i in range(len(A) // 2, -1, -1):
        max_heapify(A, i, heap_size)
```

Why does this work?

- Each node starting at $\lfloor \frac{n}{2} \rfloor + 1$ is the root of a 1-element heap.

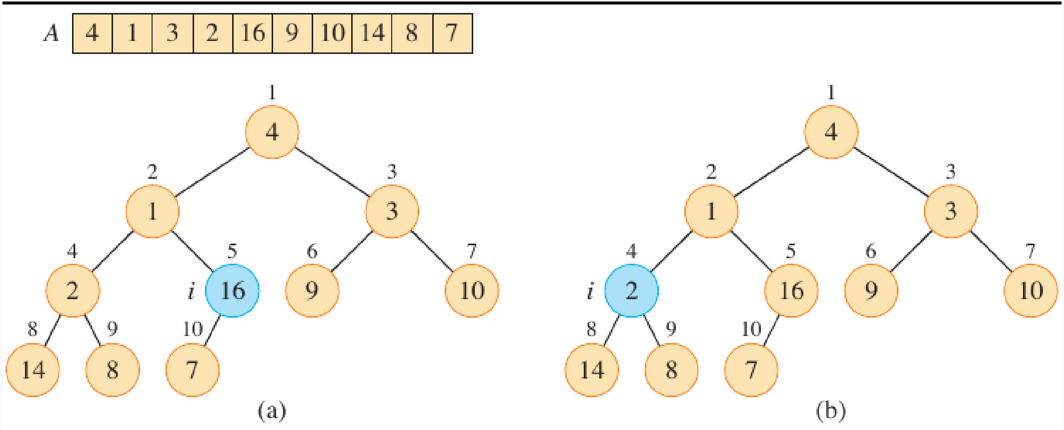
Why does this work?

- Each node starting at $\lfloor \frac{n}{2} \rfloor + 1$ is the root of a 1-element heap.
- The subnodes, which are to the right of node $\lfloor \frac{n}{2} \rfloor$, are roots of their own max-heaps.

Why does this work?

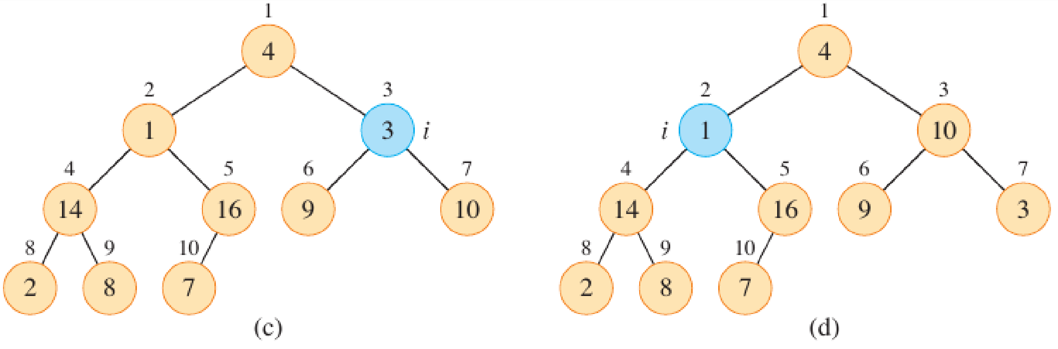
- Each node starting at $\lfloor \frac{n}{2} \rfloor + 1$ is the root of a 1-element heap.
- The subnodes, which are to the right of node $\lfloor \frac{n}{2} \rfloor$, are roots of their own max-heaps.
- The procedure loops down to the first node until all sub-heaps have been max-heapified.

Why does this work?



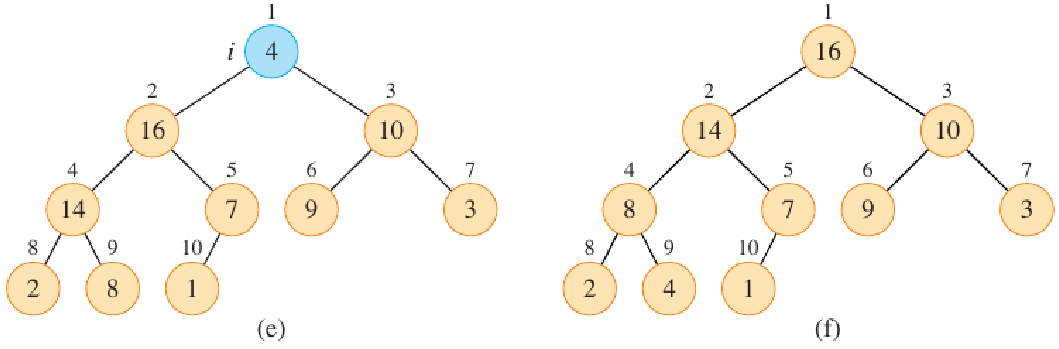
Building a heap from an array (Cormen et al.).

Why does this work?



Building a heap from an array (Cormen et al.).

Why does this work?



Building a heap from an array (Cormen et al.).

Analysis of Heapsort

The call to `max_heapify` is...

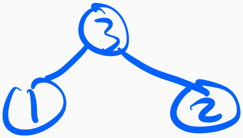
The call to `max_heapify` is... $O(\lg n)$.

The loop in `build_max_heap` runs...

The call to `max_heapify` is... $O(\lg n)$. $\Theta(n \lg n)$

The loop in `build_max_heap` runs... $O(n)$ times.

Heapsort




```python
def heapsort(A):
    build_max_heap(A)
    heap_size = len(A)
    for i in range(len(A) - 1, 0, -1):
        A[0], A[i] = A[i], A[0]
        heap_size -= 1
        max_heapify(A, 0, heap_size)
```

1	2	3	4	7

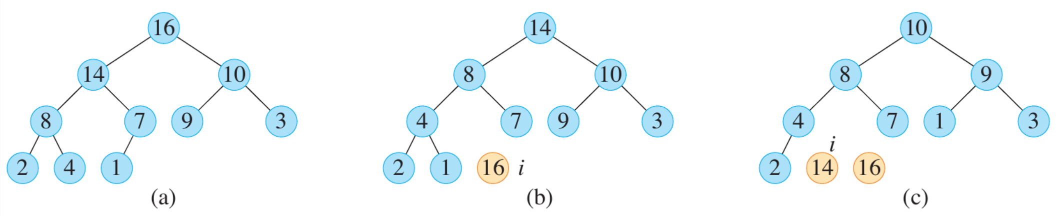
$$\Theta(n\lg n + n\lg n) = \Theta(n\lg n)$$

Analysis

- Start by building a max-heap on the input array - $O(n)$.

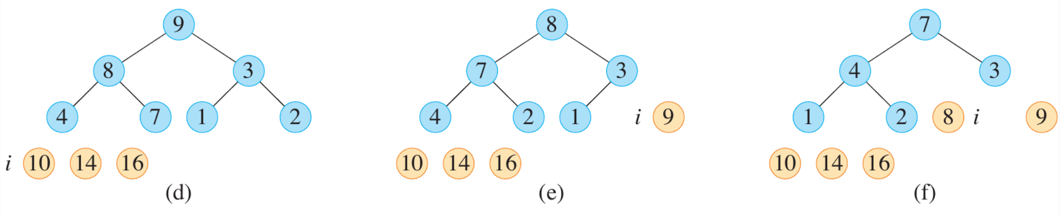
Analysis

- Start by building a max-heap on the input array - $O(n)$.
- Take the root element out of the heap and run `max_heapify` to maintain the max-heap property - $O(n \lg n)$.

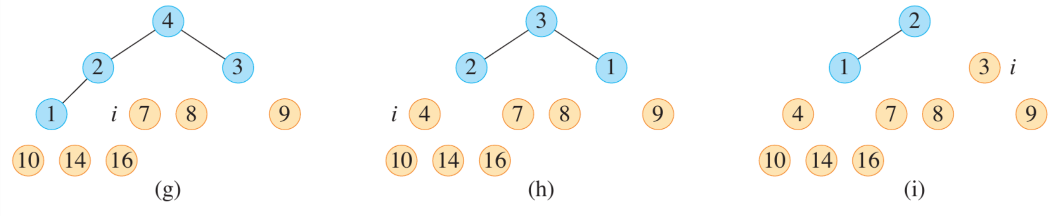


Example of Heapsort (Cormen et al.).

Heapsort

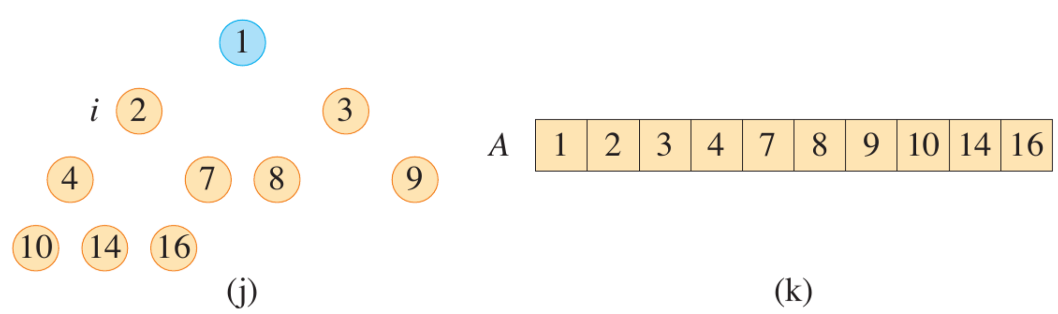


Example of Heapsort (Cormen et al.).



Example of Heapsort (Cormen et al.).

Heapsort



Example of Heapsort (Cormen et al.).

Priority Queues

What is it?

Priority Queues

What is it?

- A key-value data structure where each key has a priority.
- The elements are processed as a queue.
- Implemented with either a min-heap or max-heap (depending on the use case).

Operations

- **Insert:** Add a new element to the queue.
- **Extract:** Remove and return the element with the highest/lowest priority.
- **Max/Min:** Return the element with the highest/lowest priority without removing it.
- **Increase/Decrease Key:** Change the priority of an element.

Priority Queue: Insert

1. Add the new element to the end of the array.
2. Set the new element's priority to $-\infty$ (for max-heap) or ∞ (for min-heap).
3. Use the **Increase/Decrease Key** operation to set the correct priority.

Priority Queue: Insert

```python
def max_heap_insert(A, obj, n):
    if len(A) == n:
        raise ValueError("Heap overflow")
    key = float("-inf")
    obj.key = key
    A.append(obj)
    # map obj to the last index -- dependent on the implementation
    max_heap_increase_key(A, obj, key)
```

Priority Queue: Extract

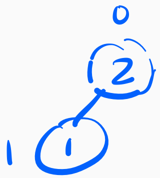
1. Grab the root element (max or min).
2. Replace the root with the last element in the array.
3. Remove the last element.
4. Call `max_heapify` or `min_heapify` on the root to maintain the heap property.

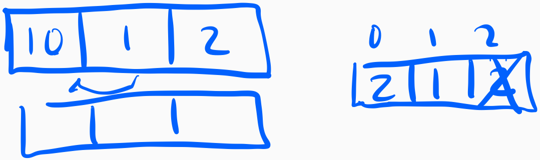
Priority Queue: Extract

```
def max_heap_maximum(A):
    if len(A) < 1:
        raise ValueError("Heap underflow")
    return A[0]

def max_heap_extract_max(A):
    max_val = max_heap_maximum(A)
    A[0] = A[-1]
    A.pop()
    max_heapify(A, 0)
    return max_val
```

max-val = 10

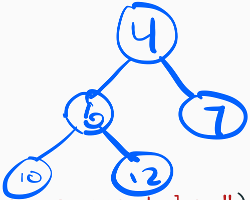




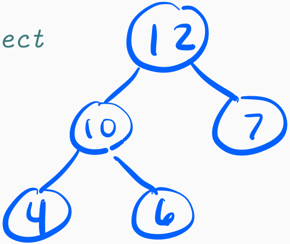
Priority Queue: Increase Key

1. Check if the new key is smaller than the current key.
2. Set the element's key to the new key.
3. While the element is not the root and its parent's key is less than the element's key:
    3.1 Swap the element with its parent.
    3.2 Move up to the parent's index.

Priority Queue: Increase Key




```python
def max_heap_increase_key(A, obj, key):
    if key < obj.key:
        raise ValueError("New key is smaller than current key")
    obj.key = key
    i = A.index(obj) # gets the index of the object
    while i > 0 and A[parent(i)].key < A[i].key:
        A[i], A[parent(i)] = A[parent(i)], A[i]
        i = parent(i)
```



Implement a minimum priority queue using a min-heap.