Chapter 6. Heapsort

Joon Soo Yoo

April 3, 2025

Assignment

- Read §6.3, §6.4, §6.5
- Problems
  - §6.1 - #2, 8
  - §6.2 - #2
  - §6.3 - #3, 4
  - §6.4 - #2, 4
  - §6.5 - #3, 7

Chapter 6: Review

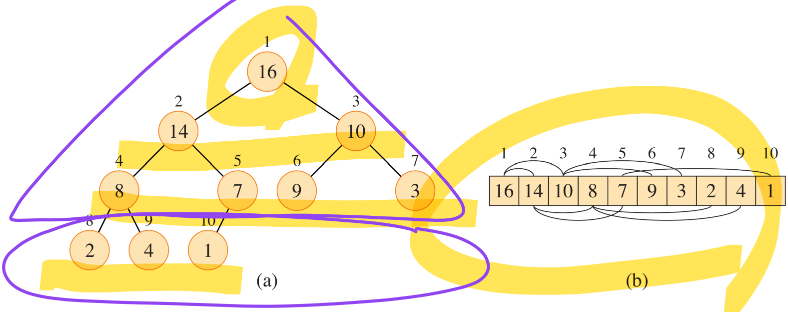
- **Chapter 6.1: Heaps**
- Chapter 6.2: Maintaining the heap property
- Chapter 6.3: Building a heap
- Chapter 6.4: The heapsort algorithm
- Chapter 6.5: Priority queues

Heapsort: Overview

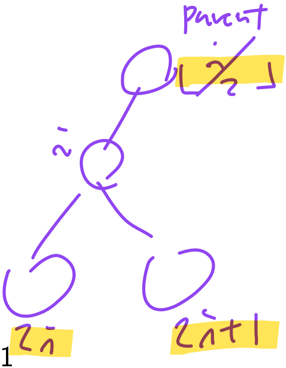
- Heapsort is a sorting algorithm introduced in this chapter.
- Like **Merge Sort**:
  - Runs in $O(n \log n)$ time
- Like **Insertion Sort**:
  - Sorts **in place**, using only constant extra space
- Combines the best features of both: fast and space-efficient
- Introduces a new algorithmic idea: using a **data structure** called a **heap** to manage information

What is a Binary Heap?

- A (binary) heap is a **nearly complete binary tree**
- Represented as an array $A[1:n]$
- `A.heap_size` denotes the number of elements currently in the heap
- Tree is filled top to bottom, left to right



Heap Index Formulas



Given a node at index i:

$$\mathtt{PARENT}(i) = \lfloor i/2 \rfloor$$
$$\mathtt{LEFT}(i) = 2i$$
$$\mathtt{RIGHT}(i) = 2i + 1$$

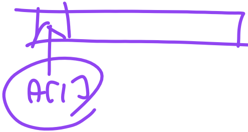
- These are computed efficiently using bit shifts
- Commonly implemented as inline functions or macros

Heap Properties

- **Max-Heap:**
  - For all nodes i (except the root), the following holds:

$$A[\text{parent}(i)] \geq A[i]$$

  - So the largest element is at the root: $A[1]$

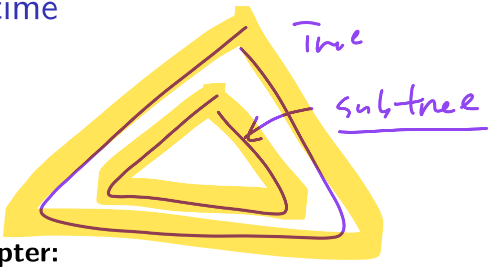


- **Min-Heap:**
  - For all nodes i (except the root), the following holds:

$$A[\text{parent}(i)] \leq A[i]$$

  - So the smallest element is at the root: $A[1]$
- Heapsort uses a **max-heap**
- Priority queues often use a **min-heap**

Heap Height and Runtime



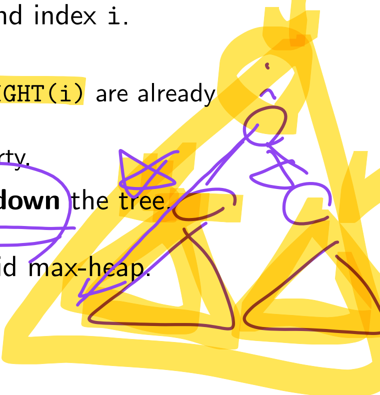
Coming up in this chapter:

- `MAX-HEAPIFY` – maintain heap property ($O(\log n)$)
- `BUILD-MAX-HEAP` – make a heap from unordered array ($\Theta(n)$)
- `HEAPSORT` – sorting in-place using a heap ($O(n \log n)$)

Chapter 6: Heapsort

Maintaining the Heap Property

- Core idea: `MAX-HEAPIFY` ensures the **max-heap property** is preserved.
- Input: Array `A` with `A.heap-size`, and index `i`.
- Assumes:
  - Subtrees rooted at `LEFT(i)` and `RIGHT(i)` are already max-heaps.
  - Node `i` may violate the heap property.
- Action: Let the value at `A[i]` **float down** the tree.



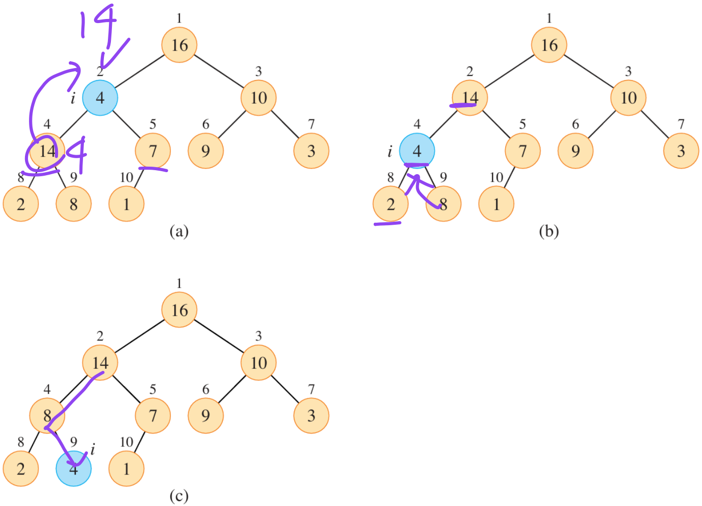
Goal: Make the subtree rooted at `i` a valid max-heap.

MAX-HEAPIFY in Action (Figure 6.2)

- Step 1: Compare `A[i]`, `A[LEFT(i)]`, and `A[RIGHT(i)]`
- Step 2: Find the largest and set index in `largest`
- Step 3: If largest $\neq$ i:
  - Swap `A[i]` and `A[largest]`
  - Recursively call `MAX-HEAPIFY` on index `largest`

Fixing violations from top-down.

MAX-HEAPIFY in Action (Figure 6.2)



Fixing violations from top-down.

MAX-HEAPIFY Algorithm

Algorithm 1 MAX-HEAPIFY(A, i) $T(n)$ $T(x)$

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i] then
    largest ← l
else
    largest ← i
end if
if r ≤ A.heap-size and A[r] > A[largest] then
    largest ← r
end if
if largest ≠ i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
end if
```

$\Theta(1)$

Note: `LEFT(i) = 2i`, `RIGHT(i) = 2i + 1`

Running Time of MAX-HEAPIFY

- Let $T(n)$ be the worst-case time for a subtree with at most n nodes.
- Time to compare and swap at each step is $O(1)$.
- Recursive call occurs on a subtree of size at most $\frac{2n}{3}$.

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

- Using the Master Theorem (Case 2):

$$T(n) = O(\log n)$$

Alternatively: Run time is $O(h)$ where h is the height of node i.

Think About It

Question 1:
Why does each child subtree in MAX-HEAPIFY have size at most $2n/3$?

Hint: Consider the structure of a complete binary tree.

Question 2:
Given the recurrence:

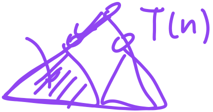
$$T(n) \leq T(2n/3) + \Theta(1)$$

use the **Master Theorem** to find the time complexity of `MAX-HEAPIFY`.

Answer: Why Subtree Size Is at Most $2n/3$

To analyze the worst-case size of a child subtree in a binary heap:

- Assume a heap with n nodes, rooted at index i.
- We want to find the maximum number of nodes in either the left or right subtree.
- The heap is a complete binary tree: all levels full, except possibly the last, which is filled left-to-right.
- To maximize the size of the left subtree, it should include as many nodes as possible in the last level.



Let k be the height of the last level, with:

$$\text{Left subtree size} = 2^k, \quad \text{Right subtree size} = 2^{k-1}$$

So:

$$2^k + 2^{k-1} = 3 \cdot 2^{k-1} \leq n \Rightarrow 2^{k-1} \leq \frac{n}{3} \Rightarrow \text{Max subtree size} \leq \frac{2n}{3}$$

Conclusion: Each recursive call to `MAX-HEAPIFY` acts on a subtree of size at most $\boxed{2n/3}$.

Answer: Time Complexity via Master Theorem

We solve the recurrence:

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

Step 1: Match to the Master Theorem form:

$$T(n) = T(n/b) + f(n)$$

Here, $a = 1$, $b = 3/2$ (since $n/b = 2n/3$), and $f(n) = \Theta(1)$

Step 2: Compare $f(n)$ to $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Conclusion: This is Case 2 of the Master Theorem:

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(\log n)$$

Final Result: `MAX-HEAPIFY` runs in $\boxed{O(\log n)}$ time.

Chapter 6: Heapsort

- Chapter 6.1: Heaps
- Chapter 6.2: Maintaining the heap property
- **Chapter 6.3: Building a heap**
- Chapter 6.4: The heapsort algorithm
- Chapter 6.5: Priority queues

Exercise 6.1-8: Identifying Leaves in a Heap

MH

Statement: In an n-element heap stored as an array $A[1:n]$, the leaves are the nodes with indices:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$$

$\lfloor n/2 \rfloor + 1$

Proof:

- In a heap, the children of a node i are at $2i$ and $2i+1$.
- If $i > \left\lfloor \frac{n}{2} \right\rfloor$, then $2i > n \rightarrow i$ has no children $\rightarrow$ it is a leaf.
- If $i \leq \left\lfloor \frac{n}{2} \right\rfloor$, then $2i \leq n \rightarrow$ it has at least a left child $\rightarrow$ it is internal.

Conclusion: Leaves are exactly those with indices from $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n.

BUILD-MAX-HEAP Algorithm

Goal: Convert an array $A[1:n]$ into a max-heap.

Procedure:

```
procedure BUILD-MAX-HEAP(A, n)
    A.heap-size ← n
    for i ← ⌊n/2⌋ downto 1 do
        MAX-HEAPIFY(A, i)
    end for
end procedure
```

$\boxed{\lfloor n/2 \rfloor \sim 1}$

$A[1] \leftarrow \max$

Key Idea: Internal nodes are heapified from bottom-up. Leaves are already heaps.

Why BUILD-MAX-HEAP Works (Loop Invariant)

Loop Invariant: At the start of each iteration, nodes $i+1$ to n are all roots of valid max-heaps.

$n/2 \longrightarrow$ leaf $\Rightarrow$ max-heap

Initialization:

- $i = \lfloor n/2 \rfloor$
- All nodes from $i+1$ to n are leaves $\rightarrow$ trivially max-heaps.

Maintenance:

i ... $i+1$ .. $i+n$

$2i$ $2i+1$

- Children of node i are greater than i
- `MAX-HEAPIFY(A, i)` ensures subtree rooted at i becomes a max heap

Termination:

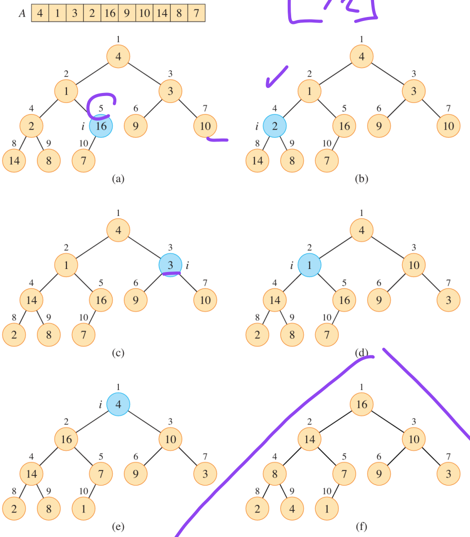
$n+1$

- After loop, all nodes 1 through n are max-heap roots
- Especially $A[1]$ — the root of the final heap

Figure: BUILD-MAX-HEAP in Action

$\lfloor n/2 \rfloor + 1 \rightarrow \underline{\underline{6}}$

$n/2 \rightarrow 1$



Naïve Runtime Analysis of BUILD-MAX-HEAP

MH → O(log n)

- The procedure BUILD-MAX-HEAP(A) runs a loop:

$$\texttt{for } i = \lfloor n/2 \rfloor \texttt{ downto } 1: \quad \texttt{MAX-HEAPIFY(A, i)}$$

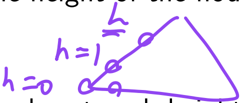
- Number of iterations: $\lfloor n/2 \rfloor \approx O(n)$
- Each call to MAX-HEAPIFY can take up to $O(\log n)$ time in the worst case (if the node is near the root)
- Therefore, total worst-case time is:

$$O(n) \cdot O(\log n) = O(n \log n)$$

- This is a correct upper bound — but not the tightest possible.

Tighter Runtime Analysis of `BUILD-MAX-HEAP`

- The simple bound $O(n \log n)$ assumes all calls to `MAX-HEAPIFY` take $O(\log n)$ time
- In reality, many nodes are near the bottom of the heap and have small height
- `MAX-HEAPIFY` runs in $O(h)$ where h is the height of the node



- Idea: sum the work done at each height
- Using known bounds on the number of nodes at each height:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O(n)$$

$\cancel{\boxed{n \log n}}$

- Tighter bound: $O(n)$

Key Questions for the Tighter Bound

To justify the tight $O(n)$ analysis, we ask:

1. Why does an n-element heap have height $\lfloor \log n \rfloor$?
2. Why are there at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h?

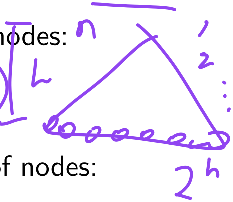
These structural properties of binary heaps allow us to bound the total work done by `BUILD-MAX-HEAP`.

Why is Heap Height $\lfloor \log_2 n \rfloor$?

Goal: Show that a binary heap with n elements has height $\lfloor \log_2 n \rfloor$

- In a perfect binary tree of height h, number of nodes:

$$1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$$



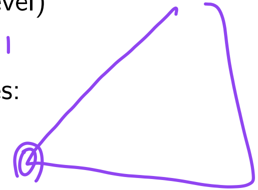
- In a binary tree of height h, minimum number of nodes:

$$2^h \text{ (only one node at bottom level)}$$

$1 + 2 + \cdots + 2^{h-1} + 1$

So the number of nodes n in a binary heap satisfies:

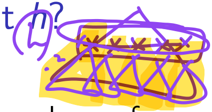
$$2^h \leq n < 2^{h+1}$$



Taking $\log_2$:

$$h \leq \log_2 n < h + 1 \Rightarrow h = \lfloor \log_2 n \rfloor$$

Why are there at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h?



Goal: Bound the number of nodes at height h in a binary heap of size n

- A node at height h must be the root of a subtree of height h
- A full binary subtree of height h has:

$$\text{Total nodes} = 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$$

- So each such node **uses up** at least $2^{h+1} - 1$ positions in the heap
- Total number of such nodes is at most:

$$\left\lfloor \frac{n}{2^{h+1} - 1} \right\rfloor \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

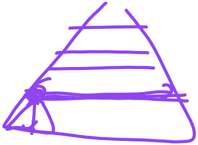
Tighter Runtime of `BUILD-MAX-HEAP`: Setup

Let:

- c: constant factor in the $O(h)$ cost of `MAX-HEAPIFY`
- Nodes at height h: at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

Total cost:

h



$$T(n) \le \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot ch$$

max-heapify

O(h)

Apply inequality: $\lceil x \rceil \le 2x$ for all $x \ge \frac{1}{2}$

$$\Rightarrow \left\lceil \frac{n}{2^{h+1}} \right\rceil \le \frac{n}{2^h} \quad \text{(CLRS Exercise 6.3-2)}$$

So:

$$T(n) \le cn \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

Transforming the Summation

We now simplify:

∞

$$\boxed{T(n) \leq cn \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}}$$

This summation is known to converge:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

From CLRS Equation A.11 (p.1142):

$$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2} \quad \text{for } 0 < x < 1$$

$$\Rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = \frac{1/2}{1/4} = 2$$

So we are safe to approximate:

$$\boxed{T(n) \leq cn \cdot 2}$$

Final Result: `BUILD-MAX-HEAP` is Linear Time

~~$O(h \log h) = O(n)$~~ ~~$O(\log n)$~~

Final bound:

$$T(n) \le cn \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = cn \cdot 2 = O(n)$$

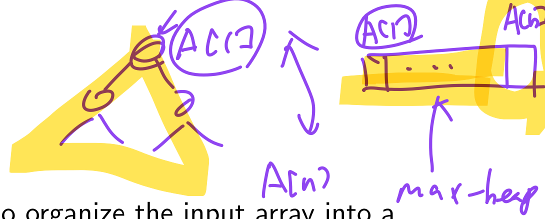
Interpretation:

- While a single `MAX-HEAPIFY` may cost $O(\log n)$, most nodes lie near the bottom of the heap
- Their small height makes their cost low
- So the total cost of building a heap is dominated by these shallow calls
- **Hence, BUILD-MAX-HEAP runs in linear time.**

Chapter 6: Heapsort

- Chapter 6.1: Heaps
- Chapter 6.2: Maintaining the heap property
- Chapter 6.3: Building a heap
- **Chapter 6.4: The heapsort algorithm**
- Chapter 6.5: Priority queues

Overview of HEAPSORT



HEAPSORT Steps:

1. Call `BUILD-MAX-HEAP` to organize the input array into a max-heap.
2. The largest element is now at the root $A[1]$.
3. Swap $A[1]$ with $A[n]$ to move the max to its final sorted position.
4. Reduce the heap size by 1.
5. Call `MAX-HEAPIFY` on $A[1]$ to restore the max-heap.
6. Repeat until heap size is 1.

Each iteration places the next-largest element in its final position.

HEAPSORT Pseudocode

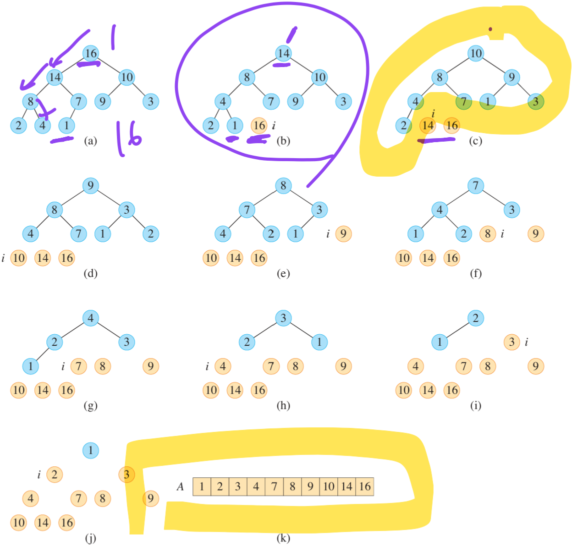
```
HEAPSORT(A, n):
  1. BUILD-MAX-HEAP(A, n)
  2. for i = n downto 2:
       ▶ exchange A[1] with A[i]
       ▶ A.heap-size = A.heap-size - 1
       ▶ MAX-HEAPIFY(A, 1)
```

$O(n)$
$+ (n-1) \times O(\log n)$

$O(n \log n)$

Key Idea: Each MAX-HEAPIFY fixes the heap with one fewer element.

Visual Example of HEAPSORT



Time Complexity of HEAPSORT

Step-by-step costs:

- `BUILD-MAX-HEAP` takes $O(n)$ time
- The loop runs $n-1$ times
- Each `MAX-HEAPIFY` call takes $O(\log n)$

$$\Rightarrow T(n) = O(n) + (n-1) \cdot O(\log n) = O(n \log n)$$

HEAPSORT is an in-place, comparison-based sort with worst-case $O(n \log n)$ **time.**

Chapter 6: Heapsort

What is a Priority Queue?

heap → "priority queue"

~~sorting~~

Definition: A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called a **key**.

Operations Supported by a Max-Priority Queue:

- `INSERT(S, x, k)`: Inserts element x with key k into S
- `MAXIMUM(S)`: Returns the element with the largest key
- `EXTRACT-MAX(S)`: Removes and returns the element with the largest key
- `INCREASE-KEY(S, x, k)`: Increases the key of x to new value k, where k is at least as large as the current key

Application of Max-Priority Queues

Use Case: Job Scheduling

- A computer shared by multiple users runs a scheduler.
- Jobs have priorities, and the scheduler selects the job with the highest priority.
- `EXTRACT-MAX` is used to select the next job to run.
- `INSERT` is used to add a new job at any time.

Takeaway: Max-priority queues are effective when the **most important task must be served first**.

Application of Min-Priority Queues

Min-Priority Queue Operations:

- `INSERT(S, x, k)`
- `MINIMUM(S)`
- `EXTRACT-MIN(S)`
- `DECREASE-KEY(S, x, k)`

Use Case: Event-Driven Simulation

- Each event has a time of occurrence as its key.
- Events are simulated in chronological order.
- `EXTRACT-MIN` selects the next event to simulate.
- `INSERT` adds new future events during simulation.

We will revisit min-priority queues in Chapters 21 and 22.

Objects and Keys in Priority Queues

Recall:

- In sorting (e.g., heapsort), the array stores **keys** directly.
- Satellite data (extra information) is moved implicitly with keys.

In Priority Queues:

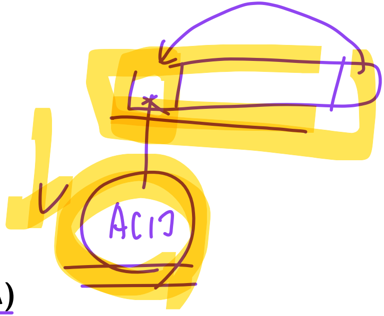
- The heap array stores **pointers to objects**.
- Each object has a `key` attribute (e.g., `object.key`).
- Heap operations compare objects by their keys.

Analogy: Object = satellite data. Key = determines priority.

MAX-HEAP-MAXIMUM and EXTRACT-MAX Code

MAX-HEAP-MAXIMUM(A)

```
if A.heap-size < 1 then
    error "heap underflow"
end if
return A[1]
```



MAX-HEAP-EXTRACT-MAX(A)

```
max ← MAX-HEAP-MAXIMUM(A)
A[1] ← A[A.heap-size]
A.heap-size ← A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

Time Complexity: $\mathcal{O}(\log n)$

MAXIMUM and EXTRACT-MAX Operations

MAXIMUM:

- Returns the object with the largest key.
- Implemented as: `return A[1]`
- Runs in $\mathcal{O}(1)$ time.

EXTRACT-MAX:

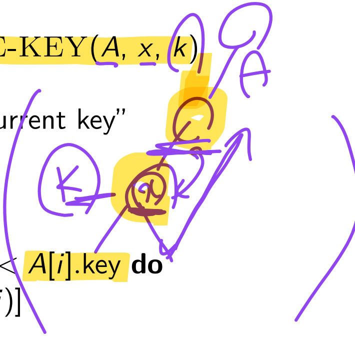
- Removes and returns the maximum element.
- Replaces root with last element, reduces heap size, calls `MAX-HEAPIFY`.
- Runs in $\mathcal{O}(\log n)$ time.
- Exchanges pointers (not raw data) and updates object-index mapping.

MAX-HEAP-INCREASE-KEY Procedure

Goal: Increase the key of object x to a new value k in the heap A.

Procedure:

```
procedure MAX-HEAP-INCREASE-KEY(A, x, k)
    if k < x.key then
        error "new key is smaller than current key"
    end if
    x.key ← k
    Find index i such that A[i] = x
    while i > 1 and A[PARENT(i)].key < A[i].key do
        Exchange A[i] with A[PARENT(i)]
        i ← PARENT(i)
    end while
end procedure
```



max-heap

Time Complexity: $\mathcal{O}(\log n)$

MAX-HEAP-INSERT Procedure

Goal: Insert object x into heap A with n slots.

Procedure:

```
procedure MAX-HEAP-INSERT(A, x, n)
    if A.heap-size == n then
        error "heap overflow"
    end if
    A.heap-size ← A.heap-size + 1
    k ← x.key
    x.key ← −∞
    A[A.heap-size] ← x
    Map x to index A.heap-size
    MAX-HEAP-INCREASE-KEY(A, x, k)
end procedure
```

heap

n

A

x $-\infty$

k

Time Complexity: $\mathcal{O}(\log n)$

Summary: Max-Priority Queue with Max-Heap

All operations are supported in $\mathcal{O}(\log n)$ time:

- `MAXIMUM(A)` $\rightarrow \mathcal{O}(1)$
- `EXTRACT-MAX(A)` $\rightarrow \mathcal{O}(\log n)$
- `INCREASE-KEY(A, x, k)` $\rightarrow \mathcal{O}(\log n)$
- `INSERT(A, x)` $\rightarrow \mathcal{O}(\log n)$

$\mathcal{O}(n)$

Overhead:

- Additional cost for maintaining object-to-index mapping (handles or hash tables)
- Typically $\mathcal{O}(1)$ per update (expected)

Question?