CS 2500: Algorithms

Lecture 12: Heap Sort

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

September 26, 2024

Introduction to Heap Sort

- Heap sort is a comparison-based sorting algorithm.
- It builds a binary heap and repeatedly extracts the maximum element.
- Time complexity: $O(n \log n)$ in the worst, average, and best cases.
- It is not a stable sort but is in-place.

Heapsort: Combining the Best Attributes

- Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \log n)$.
- Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.
- Heapsort combines the best attributes of merge sort and insertion sort.
- Heapsort introduces a new algorithm design technique: using the **heap** data structure to manage information.
- The heap is not to be confused with garbage-collected storage (e.g., in Java or Python). In this context, a heap is a specific data structure.
- The heap is also useful for implementing priority queues.

Binary Heap

- A binary heap is a complete binary tree.
- Two types:
  - A max-heap satisfies the property that every parent node is greater than or equal to its children:

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

  - A min-heap satisfies the opposite property: every parent node is less than or equal to its children:

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

  - In a max-heap, the largest element is stored at the root.
  - In a min-heap, the smallest element is at the root.
- Heap is typically represented as an array.

Array Representation of Heap

- The heap data structure is an array object that can be viewed as a nearly complete binary tree.
- Each node corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from left to right.
- The array `A[1...n]` represents the heap, with an attribute `A.heap_size` to track the number of elements in the heap.
- If `A.heap_size = 0`, the heap is empty.
- The root of the tree is `A[1]`.

Array Representation of Heap

For any element at index i:

- Parent: $i/2$
- Left Child: $2i$
- Right Child: $2i + 1$

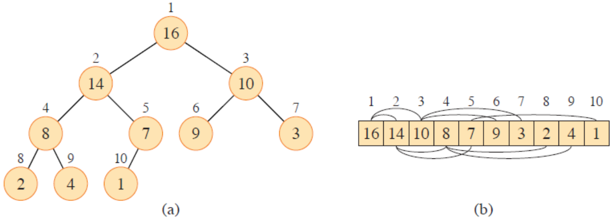


Figure: A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

Problem 1: Min and Max Elements in a Heap of Height h

Problem: What are the minimum and maximum numbers of elements in a heap of height h?

Solution:

- **Maximum number of elements:**
  - A heap is a complete binary tree, so the maximum number of nodes occurs when all levels are fully filled.
  - The number of elements in a complete binary tree of height h is:

$$\text{Max elements} = 2^{h+1} - 1$$

- **Minimum number of elements:**
  - The minimum number of nodes occurs when all levels except the last are fully filled, and the last level has at least one node.
  - The minimum number of elements is:

$$\text{Min elements} = 2^{h}$$

Problem 2: Height of an n-Element Heap

Problem: Show that an n-element heap has height $\lfloor \log n \rfloor$.
Solution:

- A heap is a complete binary tree, and the height of a complete binary tree with n elements can be derived as follows:
- The height is determined by the number of edges from the root to the deepest node.
- The number of elements at each level of a complete binary tree:
$$\text{Level 0: } 1, \quad \text{Level 1: } 2, \quad \text{Level 2: } 4, \ldots$$
- The total number of elements up to height h is:
$$2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$$
- Solving $n = 2^{h+1} - 1$ gives $h = \lfloor \log n \rfloor$.

Problem 3: Largest Value in a Max-Heap Subtree

Problem: Show that in any subtree of a max-heap, the root of the subtree contains the largest value.

Solution:

- By the max-heap property, for any node i, we have:

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

- This holds recursively for all nodes in the heap. Hence, in any subtree, the root node (which is a parent) will always contain the largest value.
- The subtree rooted at node i contains values no larger than $A[i]$.
- This property ensures that the largest element in any subtree is stored at its root.

Problem 4: Smallest Element in a Max-Heap

Problem: Where in a max-heap might the smallest element reside, assuming all elements are distinct?
Solution:

- In a max-heap, the smallest element is likely to be found in the last level (the leaves).
- Since the heap property ensures that parents are larger than children, the smallest element cannot reside in the upper levels.
- The leaf nodes are the elements furthest down in the tree, so the smallest element will always be one of the leaves.

Problem 5: Levels of the k-th Largest Element in Max-Heap

Problem: At which levels in a max-heap might the k-th largest element reside, for $2 \leq k \leq \lfloor n/2 \rfloor$, assuming all elements are distinct?

Solution:

- The root contains the largest element.
- The next largest elements (for $k \geq 2$) will be in the top levels.
- For $k = 2$, the second largest element must be one of the children of the root.
- For larger k, the k-th largest element can appear in deeper levels but will still be close to the root in terms of level.
- Generally, the k-th largest element will appear in the top few levels, but never as deep as the leaves.

Problem 6: Is a Sorted Array a Min-Heap?

Problem: Is an array that is in sorted order a min-heap?
Solution:

- Yes, an array in ascending sorted order is a min-heap.
- In a min-heap, each parent must be smaller than or equal to its children.
- Since the array is sorted, every parent element will always be smaller than or equal to its children, satisfying the min-heap property.

Problem 7: Is the Given Array a Max-Heap?

Problem: Is the array $[33, 19, 20, 15, 13, 10, 2, 13, 16, 12]$ a max-heap?

Solution:

- Check the max-heap property for each element:
  - $A[1] = 33$, which is larger than its children $A[2] = 19$ and $A[3] = 20$.
  - $A[2] = 19$, which is larger than its children $A[4] = 15$ and $A[5] = 13$.
  - $A[3] = 20$, which is larger than its children $A[6] = 10$ and $A[7] = 2$.
  - Other children also satisfy the max-heap property.
- Since all nodes satisfy the max-heap property, the array is a max-heap.

Problem 8: Indices of Leaf Nodes in an n-Element Heap

Problem: Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Solution:

- In a binary heap, a node at index i has children at indices $2i$ and $2i + 1$.
- Therefore, the nodes that have children must satisfy $i \leq \lfloor n/2 \rfloor$.
- Any node with index greater than $\lfloor n/2 \rfloor$ cannot have children, meaning these nodes are the leaf nodes.
- Hence, the leaf nodes are indexed by:

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$$

- Example: If $n = 10$, the leaves are at indices $\lfloor 10/2 \rfloor + 1 = 6, 7, 8, 9, 10$.

Maintaining the Heap Property

Introduction to `Max-Heapify`

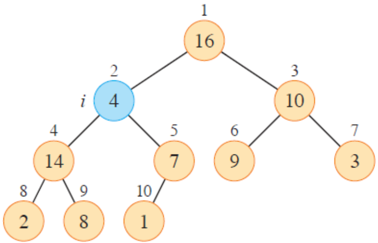
- `Max-Heapify` is an algorithm that maintains the max-heap property of a binary heap.
- Inputs:
  - An array A that represents the heap.
  - An index i into the array.
- The algorithm assumes:
  - The subtrees rooted at $Left(i)$ and $Right(i)$ are max-heaps.
  - The element $A[i]$ may violate the max-heap property.

Maintaining the Heap Property

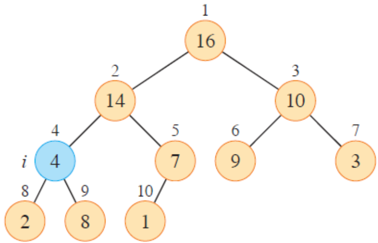
How `Max-Heapify` Works:

- The goal of `Max-Heapify` is to "float down" the value at $A[i]$ if it is smaller than one of its children.
- This ensures that the subtree rooted at index i becomes a valid max-heap.
- The algorithm compares $A[i]$ with $A[Left(i)]$ and $A[Right(i)]$.
- The largest value among these three is assigned to the variable *largest*.
- If $largest \neq i$, the values at $A[i]$ and $A[largest]$ are swapped, and the process repeats for the subtree rooted at *largest*.

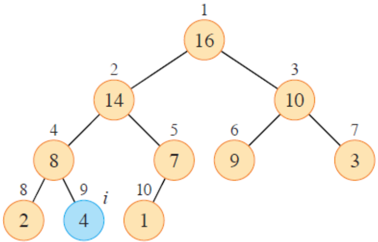
Maintaining the Heap Property



(a)



(b)



(c)

Maintaining the Heap Property

Algorithm Max-Heapify(A, i)

```
l ← Left(i)
r ← Right(i)
largest ← i
if l ≤ A.heap_size and A[l] > A[largest] then
    largest ← l
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
```

Analysis of `Max-Heapify`

- We analyze the time complexity of the `Max-Heapify` algorithm.
- Our goal is to:
  - Derive the recurrence relation for `Max-Heapify`'s running time.
  - Solve the recurrence.

Analysis of `Max-Heapify`

- **Question:** When does the worst-case scenario for a subtree rooted at a child of the root in a binary heap occur?
- **Answer:** When the bottom level is exactly half full.

Analysis of `Max-Heapify`

What Does Half-Full Mean?

- A binary heap is a complete binary tree where levels are filled from left to right.
- When the bottom level is half full, only the left half of the bottom level has nodes.
- If the bottom level has a maximum capacity of 2^h nodes, a half-full bottom level has:

$$\frac{2^h}{2} = 2^{h-1} \text{ nodes}$$

Analysis of `Max-Heapify`

What Does Half-Full Mean?

- When the bottom level is half full:
  - The left subtree contains all the nodes on the left side of the bottom level.
  - The right subtree contains no nodes from the bottom level.
- This configuration maximizes the size of the left subtree rooted at the left child of the root.

Analysis of Max-Heapify

Total Nodes in the Heap

- Total number of nodes in a binary heap = nodes above the last level + nodes at the last level.
- Number of nodes at level $i = 2^i$.
- **Nodes above the last level (fully filled levels 0 to $h-1$):**

$$\text{Nodes above last level} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

- **Nodes at the last level (half full):**

$$\text{Nodes at last level} = 2^{h-1}$$

- Therefore, the total number of nodes n is:

$$n = (2^h - 1) + 2^{h-1} = 2^h + 2^{h-1} - 1$$

Analysis of `Max-Heapify`

Nodes in the Left Subtree

- **Nodes in the left subtree above the last level:**
  - The left subtree occupies half of the nodes at each level of the entire tree beyond the root.
  - Thus, at level l, the left subtree has half the nodes of the entire tree at that level.
    - Nodes at level$l$in left subtree $= \frac{2^l}{2} = 2^{l-1}$
  - Therefore,
    number of nodes in the left subtree *above the last level* $= \sum_{l=1}^{h-1} 2^{l-1} = 2^{h-1} - 1$
- **Nodes in the left subtree at the last level (all nodes in the left half of the bottom level):**

$$\text{Left at last level} = 2^{h-1}$$

- Therefore, the total number of nodes in the left subtree is:

$$L(n) = (2^{h-1} - 1) + 2^{h-1} = 2^h - 1$$

Calculating the Ratio $\frac{L(n)}{n}$

- We now calculate the ratio of the number of nodes in the left subtree to the total number of nodes:

$$\frac{L(n)}{n} = \frac{2^h - 1}{2^h + 2^{h-1} - 1}$$

- For large h, the -1 terms become negligible.
- To simplify, divide both the numerator and denominator by 2^{h-1}:

$$\frac{L(n)}{n} \approx \frac{2}{2+1} = \frac{2}{3}$$

Analysis of `Max-Heapify`

Result

- The left subtree contains up to $\frac{2}{3}$ of the total nodes.
- This is the maximum possible ratio for a subtree rooted at a child of the root.
- This configuration maximizes the size of one subtree, making it the worst-case scenario for algorithms that depend on the balance of the tree.

Deriving the Recurrence for `Max-Heapify`

- Let $T(n)$ be the worst-case time complexity of `Max-Heapify` for a heap of size n.
- The root node compares itself with its children and may perform a swap $O(1)$.
- Then, `Max-Heapify` is recursively called on one of the subtrees, which has size at most $\frac{2n}{3}$.
- This leads to the recurrence relation:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Solving the recurrence: Applying the Master Theorem

- The recurrence $T(n) = T\left(\frac{2n}{3}\right) + O(1)$ fits the form of the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

- For our recurrence:
  - $a = 1$ (one recursive call),
  - $b = \frac{2}{3}$ (subproblem size reduces by $\frac{2}{3}$),
  - $d = 0$ (constant work outside the recursive call).
- To apply the Master Theorem, we calculate $\log_b a$:

$$\log_{\frac{2}{3}} 1 = 0$$

Analysis of `Max-Heapify`

- Using the Master Theorem, we check the case:
- Case 2 applies when $\log_b a = d$, i.e., $0 = 0$.
- Thus, the solution to the recurrence is:

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

- This means the `Max-Heapify` procedure runs in $O(\log n)$ time in the worst case.

Analysis of `Max-Heapify`

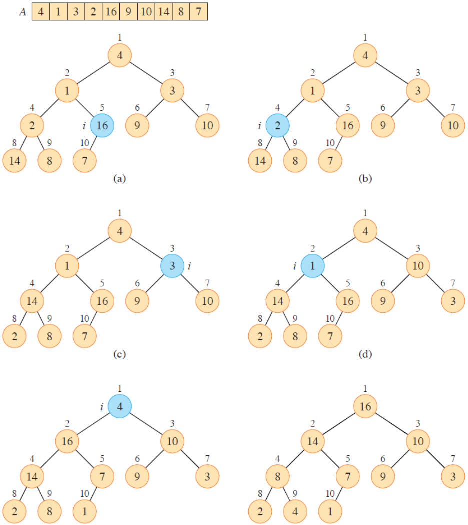
Conclusion:

- We proved that each child of the root in a binary heap is the root of a subtree containing at most $\frac{2n}{3}$ nodes.
- We derived the recurrence relation for `Max-Heapify`:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

- Using the Master Theorem, we solved the recurrence and found that `Max-Heapify` runs in $O(\log n)$ time.

Building a Heap



(a) (b) (c) (d) (e) (f)

Building a Heap

- `Build-Max-Heap` is an algorithm that converts an unordered array $A[1 \dots n]$ into a max-heap.
- It works by calling the `Max-Heapify` algorithm in a bottom-up manner, ensuring that each node satisfies the max-heap property.
- The algorithm builds the heap starting from the non-leaf nodes, which are located in the first half of the array.
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves and are already valid 1-element heaps.

Building a Heap

Algorithm Build-Max-Heap(A, n)

```
A.heap_size ← n
for i = ⌊n/2⌋ to 1 do
    Max-Heapify(A, i)
end for
```

Analysis of `Build-Max-Heap`

- A simple upper bound on the running time of `Build-Max-Heap`:
  - Each call to `Max-Heapify` takes $O(\log n)$ time.
  - `Build-Max-Heap` makes $O(n)$ calls to `Max-Heapify`.
- Thus, the running time is $O(n \log n)$.
- However, this upper bound is not as tight as it can be.

Analysis of `Build-Max-Heap`

Tighter Asymptotic Bound for `Build-Max-Heap`

- The time for `Max-Heapify` to run at a node depends on the height of the node.
- Most nodes in a heap are at lower heights, so they require less time.
  - Let this constant time be c.
  - So time for node at height $h = c \cdot h$.
- The maximum height of an n-element heap is $\lfloor \log n \rfloor$.
- At most $N_h = \lceil \frac{n}{2^{h+1}} \rceil$ nodes have a height h.
- So we get: $T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} N_h \cdot c \cdot h$
- We need to approximate $\lceil x \rceil$(where $x = N_h$) in a way that allows us to simplify the summation.

Analysis of `Build-Max-Heap`

Simplifying the Bound

- Observation:
  - $\lceil x \rceil \le 2x$ for any $x \ge \frac{1}{2}$
  - To use this inequality, we need to ensure that: $x = \frac{n}{2^{h+1}} \ge \frac{1}{2}$.
  - For the range of h we are considering $(0 \le h \le \lfloor \log n \rfloor)$, $x \ge \frac{1}{2}$ is true.
    - At $h = 0$:
$$\frac{n}{2^{0+1}} = \frac{n}{2} \ge \frac{1}{2} \quad \text{since} \quad n \ge 1$$
    - At $h = \lfloor \log_2 n \rfloor$:
$$\frac{n}{2^{\lfloor \log_2 n \rfloor + 1}} \approx \frac{n}{2^{\log_2 n}} = \frac{n}{n} = 1 \ge \frac{1}{2}$$
    - Therefore, the approximation holds throughout the range of h we are considering.
  - Why is this important?
    - Ensures Applicability: It allows us to safely apply the inequality $\lceil x \rceil \le 2x$ because x meets the necessary condition.
- Applying the approximation: $\frac{n}{2^{h+1}} \le 2\left(\frac{n}{2^{h+1}}\right) = \frac{n}{2^h}$.

Simplifying the Bound

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

Analysis of `Build-Max-Heap`

Key Takeaways

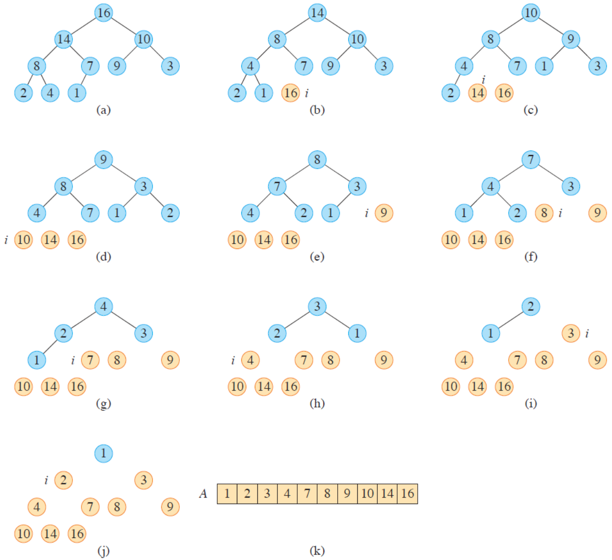
- `Build-Max-Heap` constructs a max-heap from an unordered array in linear time.
- Although a naive analysis suggests $O(n \log n)$, a tighter analysis shows that the actual running time is $O(n)$.
- The tighter bound is achieved by considering the number of nodes at each height and the time required for `Max-Heapify` at each height.

Heap Sort: Algorithm

Algorithm Heap-Sort(A, n)

```
Build-Max-Heap(A, n)
for i = n to 2 do
    Exchange A[1] with A[i]
    A.heap-size ← A.heap-size − 1
    Max-Heapify(A, 1)
end for
```

Heap Sort: Example



Heap Sort: Analysis

- The `Heap-Sort` algorithm takes $O(n \log n)$ time, since the call to `Build-Max-Heap` takes $O(n)$ time and each of the $n-1$ calls to `Max-Heapify` takes $O(\log n)$ time.