Heaps

CS 624 — Analysis of Algorithms

September 18, 2024

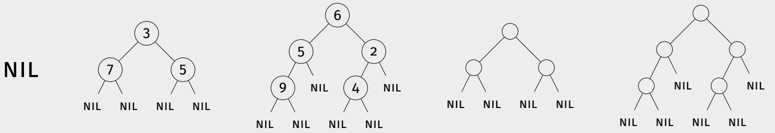
Binary Trees

Definition (Binary Tree)

A **binary tree** is either

- a **node** with two children, called *left* and *right*, which are also binary trees, and optionally a *data* field; or
- NIL, representing the empty tree

Examples (Binary trees with and without data)

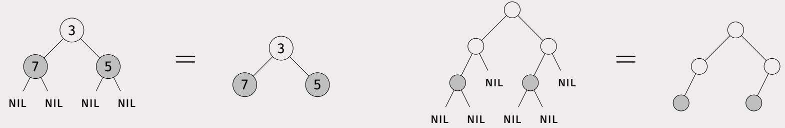


Binary Trees

Definition (Leaf Node)

A **leaf node** is a node whose children are both NIL.

Examples



NIL is often omitted from tree drawings, unless the tree is empty.

Node Height

Definition (Height)

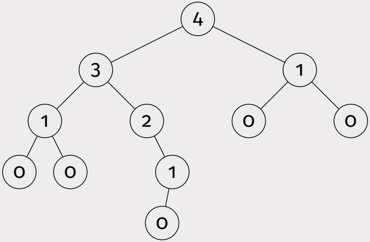
The **height** of a node in a binary tree is the number of edges on the longest path from the node to a leaf node.

The height of a binary tree is the height of its root node.

Example

In the tree to the right, each node is labeled with its height.

The tree has height 4.



Node Level

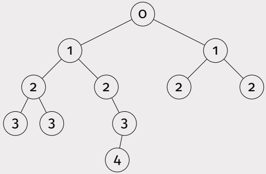
Definition (Level)

The **level** of a node is the number of edges from it to the root node.

In general, if a node has level k, its children both have level $k+1$.

Example

In the tree to the right, each node is labeled with its level.



There are at most 2^k nodes at level k.
If the highest level is completely filled in, that level contains 2^H nodes, and the tree contains $1+2+4+\cdots+2^H = 2^{H+1}-1$ nodes.

Pre-Heaps

Definition (Pre-heap)

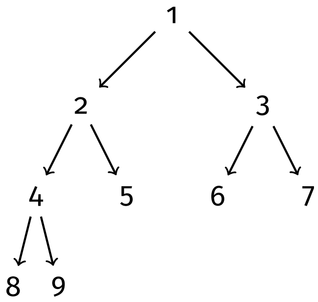
A **pre-heap** is a binary tree with the following properties:

- All leaves are on at most two adjacent levels.
- All levels, except maybe the lowest, are completely filled.
- The lowest level is filled without gaps, from the left.

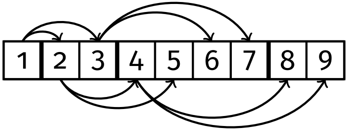
A pre-heap can be efficiently, compactly represented using an array.

Pre-Heap Representation

Logical view



Efficient representation



$$\text{Left}(k) = 2k$$
$$\text{Right}(k) = 2k + 1$$
$$\text{Parent}(k) = \lfloor k/2 \rfloor \quad \text{if } k > 1$$

Levels 0, 1, and 2 are completely filled in and contain $2^3 - 1$ nodes.
Level 3 is partly filled in from the left.

Pre-Heap Properties

Observation

Suppose we have a pre-heap with n nodes and height H.
Then $2^H \leq n \leq 2^{H+1} - 1 < 2^{H+1}$. So $H = \lfloor \log_2 n \rfloor$.

Pre-Heap Properties

Lemma

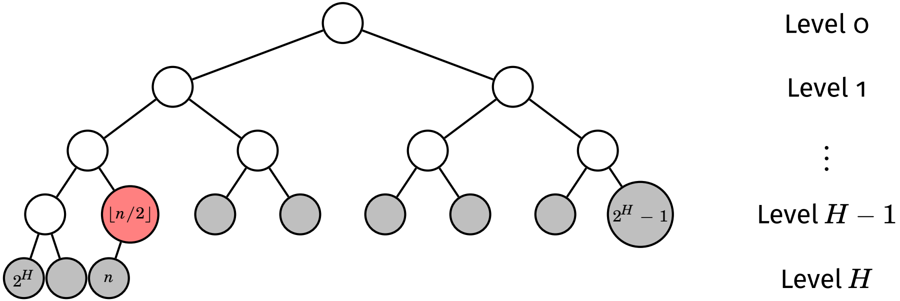
In a pre-heap with n elements, there are $\lceil \frac{n}{2} \rceil$ leaves.

Proof.

- Let H be the height. All nodes on level H are leaves.
- Some of the rightmost nodes on level $H - 1$ may be leaves.
- The parent of node n is $\lfloor \frac{n}{2} \rfloor$, and that node is the last node of height 1 on level $H - 1$, since level H is filled from the left.
- That is, all nodes after $\lfloor \frac{n}{2} \rfloor$ on level $H - 1$ are leaves, and all nodes on level H are leaves.
- So there are $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$ leaves.

□

Pre-Heap Properties



The parent of node n is $\left\lfloor \frac{n}{2} \right\rfloor$, and that node is the last node of height 1 on level $H - 1$, since level H is filled from the left.

Pre-Heap Properties

Corollary

In a pre-heap with height H, there are at most 2^H leaves.

Proof.

If n is the number of elements in the pre-heap, we know that

$$2^H \leq n \leq 2^{H+1} - 1 < 2^{H+1}$$

Then by the Lemma, the number of leaves is

$$\left\lceil \frac{n}{2} \right\rceil \leq \frac{2^{H+1}}{2} = 2^H$$

□

Pre-Heap Properties

Theorem

In a pre-heap with n elements, there are at most $\frac{n}{2^h}$ nodes at height h.

Proof.

- We have just seen that there are at most 2^H leaves in such a tree, and the leaves are just the nodes at height 0.
- If we take away the leaves, we have a smaller pre-heap with at most 2^{H-1} leaves, and these leaves are exactly the nodes at height 1 in the original tree.
- Continuing, we see that there are at most 2^{H-h} nodes at height h in the original tree, therefore $2^{H-h} = \frac{2^H}{2^h} \leq \frac{n}{2^h}$

□

Heaps

Definition (Heap)

A **heap** is a pre-heap where each node contains a key, the keys are comparable, and each node satisfies the heap properties:

1. The node's key is greater than or equal to the keys of its children.
2. The node's left and right subtrees are also heaps.

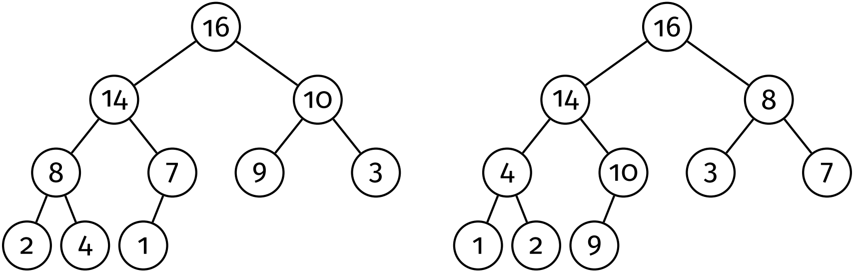
Specifically, this is called a **max-heap**; the root has the maximum key.

Another way of phrasing the heap conditions would be:

- The key at each node is greater than or equal to the key in any descendant of that node.

Example: Two Heaps With the Same Set of Keys

The shape of a heap with n elements is uniquely determined, since it is a pre-heap, but the arrangement of the elements is not.

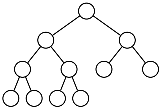


Building a Heap

How can we (efficiently) build a heap?

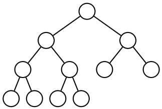
Input: A pre-heap represented by an array.

Strategy:



Building a Heap

How can we (efficiently) build a heap?



Input: A pre-heap represented by an array.

Strategy: Let's try divide and conquer:

- **Sub-problems:** Turn the left and right children of the into heaps.
- **Combine:** Given a pre-heap whose left and right children are heaps, convert the whole thing into one heap.

Building a Heap, Recursively

Initial call: $\text{BuildHeapRec}(A, 1)$, with $\text{Heapsize}(A) \leftarrow \text{Length}(A)$.

Algorithm 1 $\text{BuildHeapRec}(A, i)$

Ensure: The subtree rooted at $A[i]$ is a heap.

```
if i > Heapsize(A) then
    // Then i represents NIL
    return
else
    l ← Left(i)
    r ← Right(i)
    BuildHeapRec(A, l)
    BuildHeapRec(A, r)
    Heapify(A, i)
end if
```

The Heapify Procedure

Algorithm 2 Heapify(A, i)

Require: $1 \leq i \leq \text{Heapsize}(A)$, and the sub-trees rooted at $A[\text{Left}(i)]$ and $A[\text{Right}(i)]$ (if they exist) are heaps.

Ensure: The tree rooted at $A[i]$ is a heap.

```
l ← Left(i)
r ← Right(i)
largest ← i
if l ≤ Heapsize[A] and A[l] > A[i] then
    largest ← l
end if
if r ≤ Heapsize[A] and A[r] > A[largest] then
    largest ← r
end if
if largest ≠ i then
    exchange A[i] ↔ A[largest]
    Heapify(A, largest)
end if
```

The Heapify Procedure

Comments on Heapify:

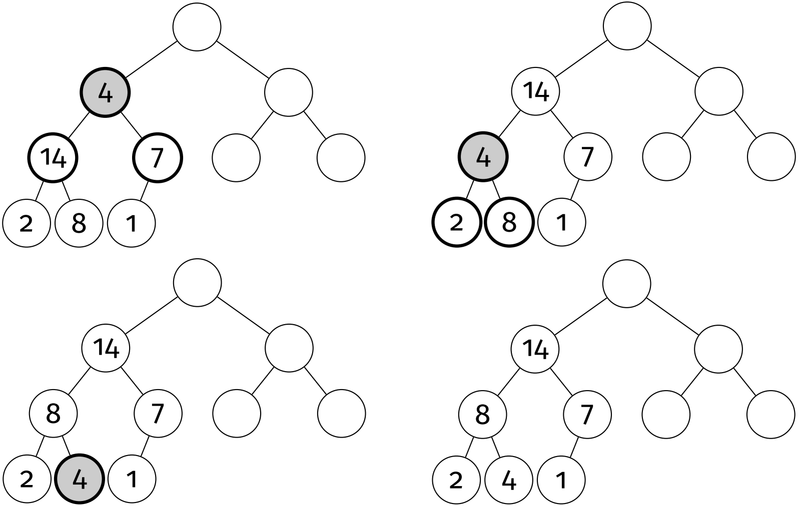
- $largest$ is the index of node with the largest key, out of i, its left child (if it exists), and its right child (if it exists).

  There are three cases:

  1. $largest = i$. Then $A[i]$ satisfies the heap properties. Done.
  2. $largest = \text{Left}(i)$. Then $A[i]$ does not satisfy heap property #1. By exchanging $A[i] \leftrightarrow A[largest]$, we fix heap property #1, but we may have broken heap property #2: the left sub-tree may no longer be a heap. So we repair it by calling Heapify recursively. (?)
  3. $largest = \text{Right}(i)$. Similar to previous case.

- The algorithm works by letting the value $A[i]$ "float down" to its proper position in the heap.

Example: Heapify



Running Time of Heapify

The time needed to run Heapify on a subtree of size n rooted at a given node i is (worst case)

- time $\Theta(1)$ to fix up the relationships among the elements $A[i]$, $A[\text{Left}(i)]$, and $A[\text{Right}(i)]$, plus
- time to run Heapify on a subtree rooted at one i's children

  That subtree has size at most $2n/3$ — the worst case occurs when the last row of the tree is exactly half full. (?)

So the running time $T(n)$ can be characterized by the recurrence

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

Running Time of Heapify

The time needed to run Heapify on a subtree of size n rooted at a given node i is (worst case)

- time $\Theta(1)$ to fix up the relationships among the elements $A[i]$, $A[\text{Left}(i)]$, and $A[\text{Right}(i)]$, plus
- time to run Heapify on a subtree rooted at one i's children

  That subtree has size at most $2n/3$ — the worst case occurs when the last row of the tree is exactly half full. (?)

So the running time $T(n)$ can be characterized by the recurrence

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

This falls into case 2 of the master theorem, and so we must have $T(n) = O(\log n)$.

Running Time of BuildHeapRec

We can express the running time for BuildHeapRec with this recurrence:

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

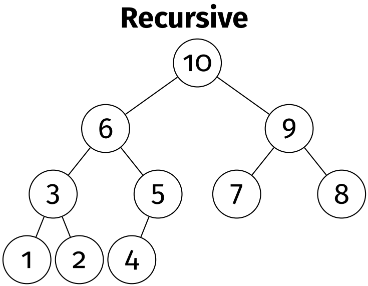
By the master theorem (case 1), that gives us $\Theta(n^{\log_{3/2} 2}) = O(n^{1.71})$.

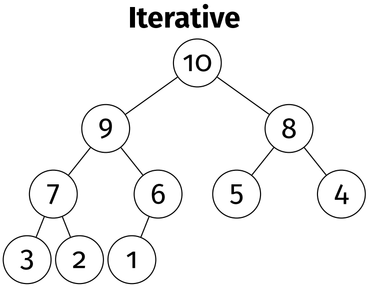
This recurrence is a coarse bound, though; it fails to capture the fact that the sum of the subproblem sizes is less than n.

We can change the order that we tackle the subproblems.
That leads to an iterative algorithm and a better analysis.

Changing the Order of Subproblems

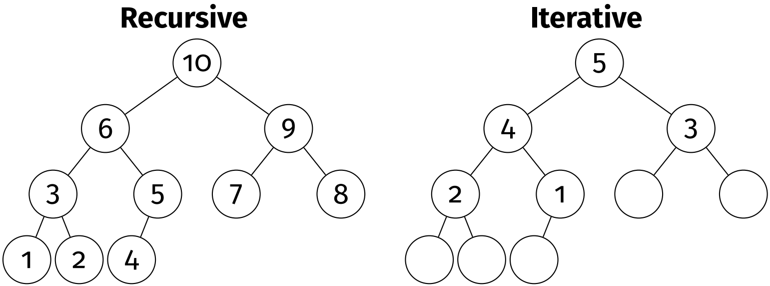
The *real work* happens in the combine step: Heapify.
What order is Heapify first called on a node?





Changing the Order of Subproblems

The *real work* happens in the combine step: $\mathrm{Heapify}$.
What order is $\mathrm{Heapify}$ first called on a node?



Optimization: There's no need to call $\mathrm{Heapify}$ on a leaf node.

Building a Heap

Iterative algorithm: The heap is built from the bottom up, starting at the first non-leaf node.

Algorithm 3 $\text{BuildHeap}(A)$

Ensure: The tree rooted at $A[1]$ is a heap.

```
1: Heapsize[A] ← Length[A]
2: for i ← ⌊Length[A]/2⌋ to 1 do
3:     Heapify(A, i)
4: end for
```

To prove that this is correct we use the following loop invariant:

Lemma (Loop Invariant)

*Let $n = \text{Length}(A)$. At the start of each iteration of the **for** loop, each node $i+1, i+2, \ldots, n$ is the root of a heap.*

Proof of Correctness

Proof.

Initialization: On the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial heap.

Maintenance:

- Assume it is true for loop i: each node $i+1, \ldots, n$ is the root of a heap.
- Goal: show that at the end of iteration i, the invariant is true for $i-1$.
- By the LI, both children of i, namely $2i$ and $2i+1$ (if they exist) are heaps. That satisfies the precondition for $\mathrm{Heapify}$.
- The call to $\mathrm{Heapify}(A, i)$ makes i a heap (postcondition).
- Furthermore, all nodes which are not descendants of i are untouched by the call to $\mathrm{Heapify}(A, i)$, and so we can conclude that each node $i, i+1, \ldots, n$ is now the root of a heap.

Termination: The loop exits when $i = 0$, and the loop invariant implies that node 1 is the root of a heap. □

Running Time of BuildHeap

- The number of elements of the heap at height h is $\leq \frac{n}{2^h}$, and the cost of running Heapify on a node of height h is $O(h)$.
- The root of a heap of n elements has height $\lfloor \log_2 n \rfloor$.
- Therefore the worst-case cost of running BuildHeap on a heap of n elements is bounded by

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = O(n)$$

Since the sum converges, so we don't care what the upper bound of the summation is.

Heap Properties

Heaps give us partial information about the order of their elements.

- We can tell immediately what the largest element is.
- They are really cheap to build.
- They are reasonably cheap to update.
- They can be stored in a simple array.

This makes them very useful for various applications.

Heapsort

Algorithm 4 Heapsort(A)

Ensure: A is sorted

```
BuildHeap(A)
for i ← Length[A] to 2 do
    exchange A[1] ↔ A[i]
    Heapsize[A] ← Heapsize[A] − 1
    Heapify(A, 1)
end for
```

The call to BuildHeap takes time $O(n)$. Each of the $n-1$ calls to Heapify takes time $O(\log n)$. Hence the total running time (in the worst case) is $O(n \log n)$.

Priority Queues

Definition

A **priority queue** is a data structure that maintains a set S of elements, each with an associated value called a *key*. (As usual, the keys must be comparable.)

The priority queue supports the following operations:

- $\text{Insert}(S, x)$ inserts the element x into the set S.
- $\text{Maximum}(S)$ returns the element of S with the largest key.
- $\text{ExtractMax}(S)$ removes and returns the element of S with the largest key.
- $\text{IncreaseKey}(S, x, k)$ increases the value of element x's key to the new value k, which must be at least as large as x's current key value.

A priority queue can be implemented using a heap.

Priority Queue Operations

Algorithm 5 HeapMaximum(A)

Require: Heapsize(A) ≥ 1
1: **return** $A[1]$

Obviously, the run time is $O(1)$.

Algorithm 6 HeapExtractMax(A)

Require: Heapsize(A) ≥ 1
1: $maxx \leftarrow A[1]$
2: $A[1] \leftarrow A[\text{Heapsize}[A]]$
3: $\text{Heapsize}[A] \leftarrow \text{Heapsize}[A] - 1$
4: Heapify(A, 1)
5: **return** $maxx$

Here the running time is dominated by the call to Heapify, so it is $O(\log n)$.

Priority Queue Operations

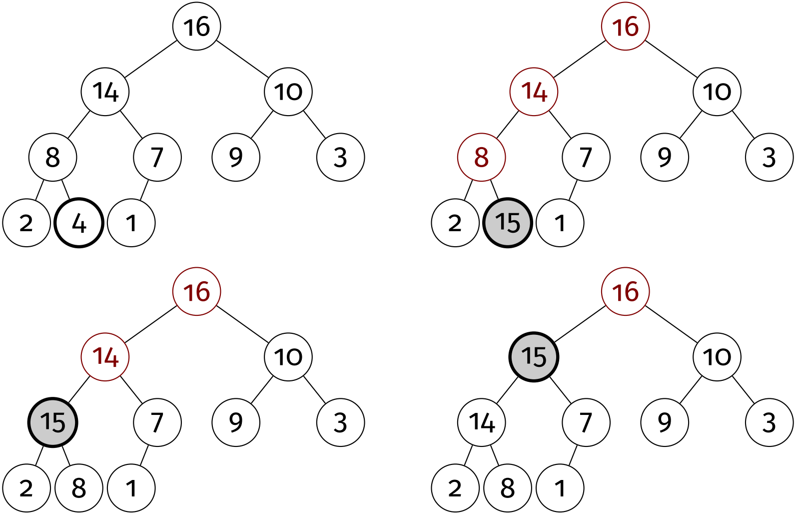
Algorithm 7 HeapIncreaseKey(A, i, key)

Require: $key \geq A[i]$

```
A[i] ← key
while i > 1 and A[Parent(i)] < A[i] do
    exchange A[i] ↔ A[Parent(i)]
    i ← Parent(i)
end while
```

We just increase the key of $A[i]$, and then let that node “float up” to its proper position.

Example: HeapIncreaseKey



HeapInsert

Algorithm 8 HeapInsert(A, key)

Require: $\text{Heapsize}(A) < \text{Length}(A)$, or A can grow

1: $\text{Heapsize}[A] \leftarrow \text{Heapsize}[A] + 1$
2: $A[\text{Heapsize}[A]] \leftarrow -\infty$
3: $\text{HeapIncreaseKey}(A, \text{Heapsize}[A], key)$

The running time here is again $O(\log_2 n)$.

Thus, a heap supports any priority queue operation on a set of size n in $O(\log n)$ time.