CS 577 - Union-Find & Binary Heaps

Manolis Vlatakis

Department of Computer Sciences
University of Wisconsin – Madison

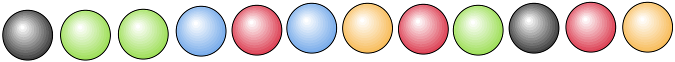
Fall 2024

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Union-Find

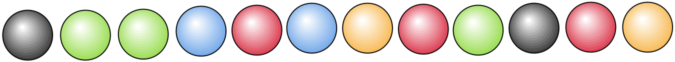
Disjoint Sets

Imagine you have a set of objects, and you want to group them based on some criteria.

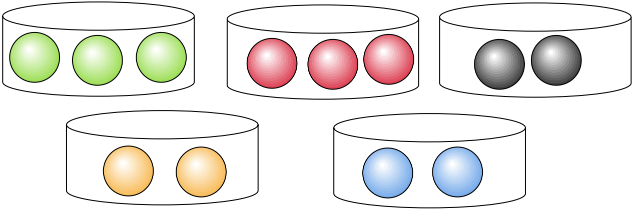


Disjoint Sets

Imagine you have a set of objects, and you want to group them based on some criteria.



One possible grouping might look like this:



Queries on Disjoint Sets

While grouping the objects, you may ask:

- How can I merge more groups?
- How many distinct groups are there?
- Do two objects belong to the same group?

Queries on Disjoint Sets

While grouping the objects, you may ask:

- How can I merge more groups?
- How many distinct groups are there?
- Do two objects belong to the same group?

Union-Find Operations

The Union-Find data structure supports three operations:

Queries on Disjoint Sets

While grouping the objects, you may ask:

- How can I merge more groups?
- How many distinct groups are there?
- Do two objects belong to the same group?

Union-Find Operations

The Union-Find data structure supports three operations:

- Init: Takes a set of elements $S = \{x_1, \cdots, x_\ell\}$ and constructs ℓ singleton sets $S_i = \{x_i\}$ for all $i = 1, \cdots, \ell$.

Queries on Disjoint Sets

While grouping the objects, you may ask:

- How can I merge more groups?
- How many distinct groups are there?
- Do two objects belong to the same group?

Union-Find Operations

The Union-Find data structure supports three operations:

- Init: Takes a set of elements $S = \{x_1, \cdots, x_\ell\}$ and constructs ℓ singleton sets $S_i = \{x_i\}$ for all $i = 1, \cdots, \ell$.
- Union: Takes two elements x and y and merges the groups containing x and y.

Queries on Disjoint Sets

While grouping the objects, you may ask:

- How can I merge more groups?
- How many distinct groups are there?
- Do two objects belong to the same group?

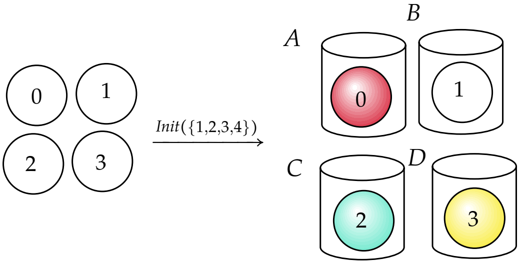
Union-Find Operations

The Union-Find data structure supports three operations:

- Init: Takes a set of elements $S = \{x_1, \cdots, x_\ell\}$ and constructs ℓ singleton sets $S_i = \{x_i\}$ for all $i = 1, \cdots, \ell$.
- Union: Takes two elements x and y and merges the groups containing x and y.
- Find: Takes an element x and identifies the group containing x.

Union-Find Data Structure

Init-Example

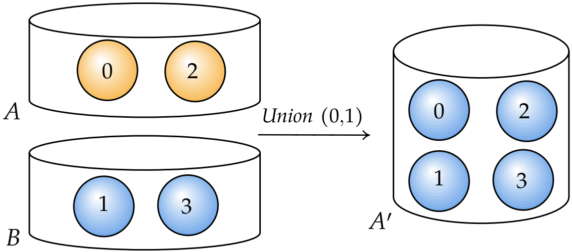


Explanation

Initially, each element starts in its own singleton set. This is the result of calling `Init`, where we have disjoint sets for each individual element.

Union-Find Data Structure

Union-Example

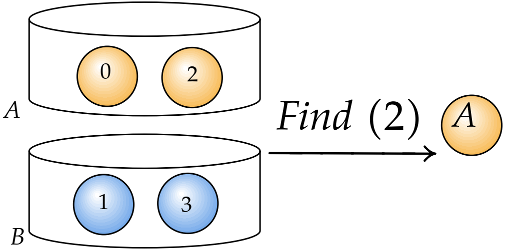


Explanation

For example, initially, 0 and 2 belong to the same group, and 1 and 3 belong to another group. When we call the function `union(0, 1)`, it merges the groups containing 0 and 1.

Union-Find Data Structure

Find-Example

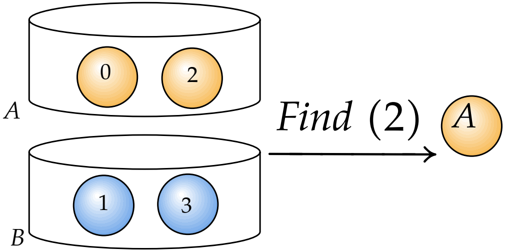


Explanation

For example, when we call `find(2)`, it returns some signal for the set containing 2.

Union-Find Data Structure

Find-Example



"Leader Element"

Each set should have a unique 'leader' element, which identifies the set.

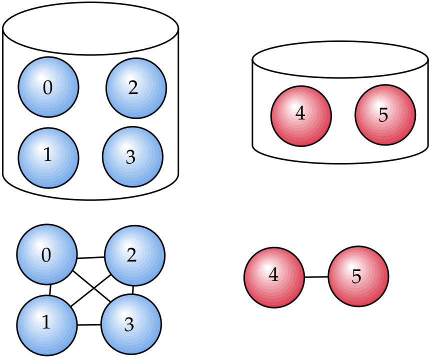
Since the sets are always disjoint, the same object cannot be the leader of more than one set.

Union-Find Data Structure

High-Level Representation

One of the most easiest ways to represent sets is through graphs:

- Same set=Same connected component. Connect all objects that belong to the same set with edges, like this:

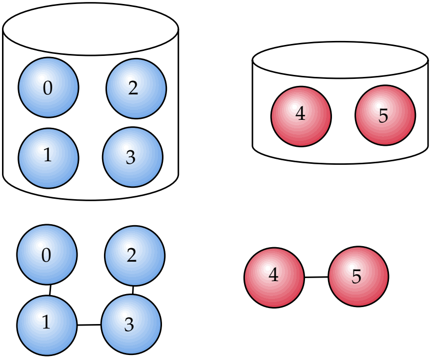


Union-Find Data Structure

High-Level Representation

One of the most easiest ways to represent sets is through graphs:

- Same set=Same tree/path:

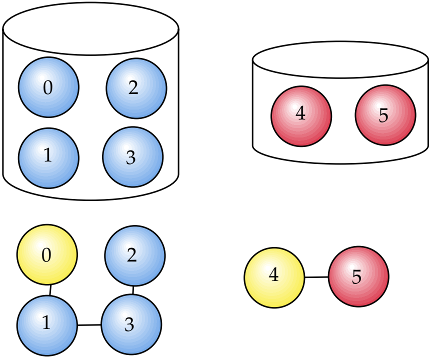


Union-Find Data Structure

High-Level Representation

One of the most easiest ways to represent sets is through graphs:

- To determine the group an element belongs to, we designate a representative element for each group.
  This representative acts as the identifier for the entire group:



Find Operations

- $\textsc{Find}(0) = 0$
- $\textsc{Find}(1) = 0$
- $\textsc{Find}(2) = 0$
- $\textsc{Find}(3) = 0$
- $\textsc{Find}(4) = 4$
- $\textsc{Find}(5) = 4$

Union-Find Data Structure

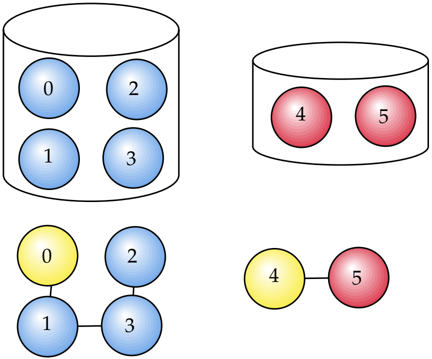
High-Level Representation

One of the most easiest ways to represent sets is through graphs:

- To determine the group an element belongs to, we designate a

SameSet(x,y)=True if Find(x)=Find(y) otherwise False

This representative acts as the identifier for the entire group:



Find Operations

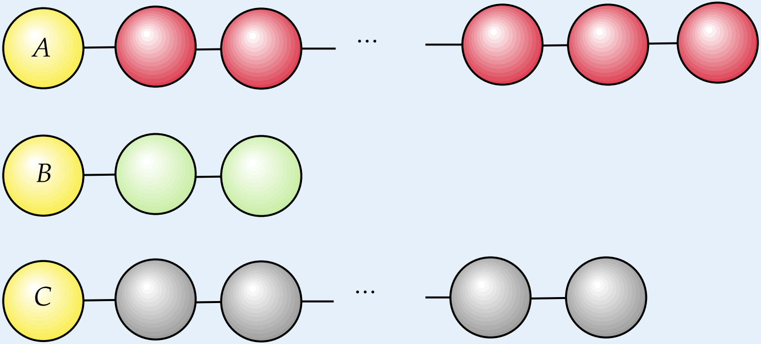
- $\textsc{Find}(0) = 0$
- $\textsc{Find}(1) = 0$
- $\textsc{Find}(2) = 0$
- $\textsc{Find}(3) = 0$
- $\textsc{Find}(4) = 4$
- $\textsc{Find}(5) = 4$

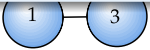
Union-Find Data Structure

High-Level Representation

One of the most easiest ways to represent sets is through graphs:

What is the worst-case complexity of Find if sets are represented by paths/chains for a data structure of n elements?





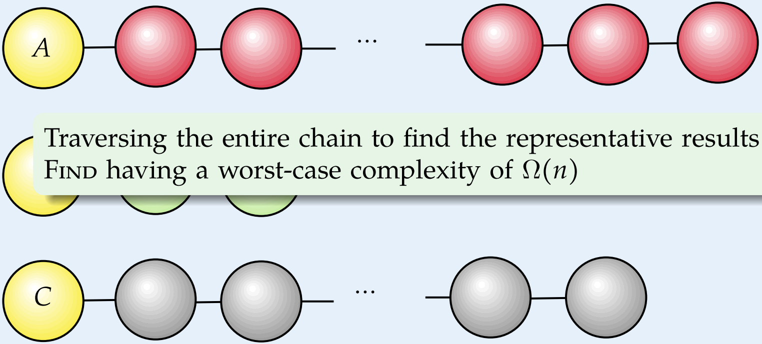
- Find(5) = 4

UNION-FIND DATA STRUCTURE

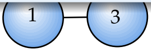
HIGH-LEVEL REPRESENTATION

One of the most easiest ways to represent sets is through graphs:

What is the worst-case complexity of FIND if sets are represented by paths/chains for a data structure of n elements?



Traversing the entire chain to find the representative results in FIND having a worst-case complexity of $\Omega(n)$



- FIND(5) = 4

Reversed Trees

To make paths shorter, it's beneficial to represent them with trees.

- Each node points to another node, called its parent, except for the leader of each set, which points to itself and thus is the root of the tree.

Init(x)

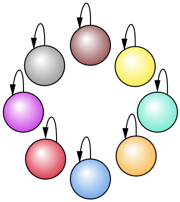
```
parent(x) ← x
```

Find(x)

```
while (x ≠ parent(x))
  x ← parent(x)
return x
```

Union(x, y)

```
x ← Find(x)
y ← Find(y)
parent(y) ← x
```



- Init is trivial. *Cost*[Init] = $\Theta(n)$.

Reversed Trees

To make paths shorter, it's beneficial to represent them with trees.

- Each node points to another node, called its parent, except for the leader of each set, which points to itself and thus is the root of the tree.

Init(x)

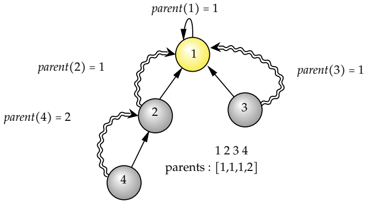
```
parent(x) ← x
```

Find(x)

```
while (x ≠ parent(x))
  x ← parent(x)
return x
```

Union(x, y)

```
x ← Find(x)
y ← Find(y)
parent(y) ← x
```



•Find traverses the tree until the root is found. Worst-case cost equals the height of tree.

Reversed Trees

To make paths shorter, it's beneficial to represent them with trees.

- Each node points to another node, called its parent, except for the leader of each set, which points to itself and thus is the root of the tree.

Init(x)

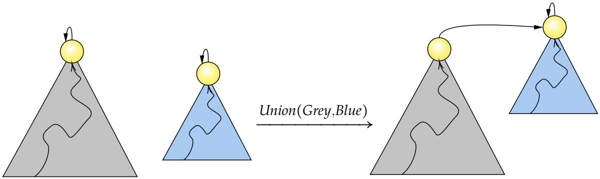
```
parent(x) ← x
```

Find(x)

```
while (x ≠ parent(x))
  x ← parent(x)
return x
```

Union(x, y)

```
x ← Find(x)
y ← Find(y)
parent(y) ← x
```



•Union just redirects the parent pointer of one leader to the other. *Cost*[Union] = $\Theta(1)$.

Reversed Trees

To make paths shorter, it's beneficial to represent them with trees.

- Each node points to another node, called its parent, except for the leader of each set, which points to itself and thus is the root of the tree.

Init(x)

```
parent(x) ← x
```

Find(x)

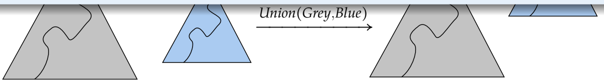
```
while (x ≠ parent(x))
  x ← parent(x)
return x
```

Union(x, y)

```
x ← Find(x)
y ← Find(y)
parent(y) ← x
```

How do we avoid creating very tall sparse trees?

Union is straightforward, but if we're not careful, Find can become costly.



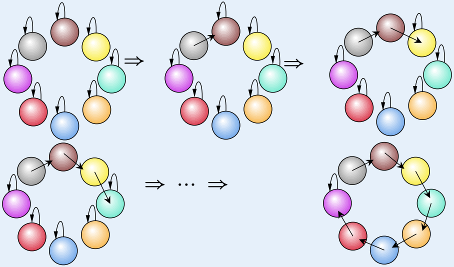
- Union just redirects the parent pointer of one leader to the other. $Cost[\textsc{Union}] = \Theta(1)$.

Union by Depth

Making Find Operation Efficient

🤔 If we always attach the taller tree to the shorter one, we might inadvertently create a long path-tree:

$$\text{UNION}(1,2), \text{UNION}(1,3), \cdots, \text{UNION}(1,n)$$



But what if we apply the inverse rule?

Union by Depth

Making Find Operation Efficient

Whenever we need to merge two trees, we always make the root of the shallower tree a child of the deeper one. This requires us to also maintain the depth of each tree, but this is quite easy.

MakeSet(x)

```
parent(x) ← x
depth(x) ← 0
```

Find(x)

```
while x ≠ parent(x)
  x ← parent(x)
return x
```

Union(x, y)

```
x̄ ← Find(x)
ȳ ← Find(y)
if depth(x̄) > depth(ȳ)
  parent(ȳ) ← x̄
else
  parent(x̄) ← ȳ
  if depth(x̄) = depth(ȳ)
    depth(ȳ) ← depth(ȳ) + 1
```

Union by Depth

Making Find Operation Efficient

Whenever we need to merge two trees, we always make the root of the shallower tree a child of the deeper one. This requires us to also maintain the depth of each tree, but this is quite easy.

MakeSet(x)

```
parent(x) ← x
depth(x) ← 0
```

Find(x)

```
while x ≠ parent(x)
  x ← parent(x)
return x
```

Union(x, y)

```
x̄ ← Find(x)
ȳ ← Find(y)
if depth(x̄) > depth(ȳ)
  parent(ȳ) ← x̄
else
  parent(x̄) ← ȳ
  if depth(x̄) = depth(ȳ)
    depth(ȳ) ← depth(ȳ) + 1
```

When depth($\bar{x}$) reaches d for the first time, $\bar{x}$ becomes the leader of two merged sets, each with leaders of depth $d-1$.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\bar{x}$, the size of the tree of $\bar{x}$ is at least $2^{\text{depth}(\bar{x})}$.

Proof by Induction:

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\bar{x}$, the size of the tree of $\bar{x}$ is at least $2^{\text{depth}(\bar{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\bar{x}) = 0$, then $\bar{x}$ is the leader of a singleton set, so the size of $\bar{x}$'s set is $2^0 = 1$.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\overline{x}$, the size of the tree of $\overline{x}$ is at least $2^{\text{depth}(\overline{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set, so the size of $\overline{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\overline{y}$ with depth $d - 1$, the size of $\overline{y}$'s set is at least 2^{d-1}.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\overline{x}$, the size of the tree of $\overline{x}$ is at least $2^{\text{depth}(\overline{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set, so the size of $\overline{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\overline{y}$ with depth $d - 1$, the size of $\overline{y}$'s set is at least 2^{d-1}.
  - When $\text{depth}(\overline{x})$ becomes d for the first time, $\overline{x}$ is the leader of the union of two sets, both of whose leaders had depth $d - 1$.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\overline{x}$, the size of the tree of $\overline{x}$ is at least $2^{\text{depth}(\overline{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set, so the size of $\overline{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\overline{y}$ with depth $d - 1$, the size of $\overline{y}$'s set is at least 2^{d-1}.
  - When $\text{depth}(\overline{x})$ becomes d for the first time, $\overline{x}$ is the leader of the union of two sets, both of whose leaders had depth $d - 1$.
  - By the inductive hypothesis, both component sets had at least 2^{d-1} elements.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\overline{x}$, the size of the tree of $\overline{x}$ is at least $2^{\text{depth}(\overline{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set, so the size of $\overline{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\overline{y}$ with depth $d - 1$, the size of $\overline{y}$'s set is at least 2^{d-1}.
  - When $\text{depth}(\overline{x})$ becomes d for the first time, $\overline{x}$ is the leader of the union of two sets, both of whose leaders had depth $d - 1$.
  - By the inductive hypothesis, both component sets had at least 2^{d-1} elements.
  - Therefore, the new set has at least 2^d elements.

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\overline{x}$, the size of the tree of $\overline{x}$ is at least $2^{\text{depth}(\overline{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set, so the size of $\overline{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\overline{y}$ with depth $d-1$, the size of $\overline{y}$'s set is at least 2^{d-1}.
  - When $\text{depth}(\overline{x})$ becomes d for the first time, $\overline{x}$ is the leader of the union of two sets, both of whose leaders had depth $d-1$.
  - By the inductive hypothesis, both component sets had at least 2^{d-1} elements.
  - Therefore, the new set has at least 2^d elements.

Contrapositive Trick: If $A \Rightarrow B$, then $\neg B \Rightarrow \neg A$.

For any leader $\overline{x}$, if the size of the tree of $\overline{x}$ is strictly less than 2^k, then $\text{depth}(\overline{x}) < k$.

Explanation: *If $\text{depth}(\overline{x}) \geq k$, then by above theorem, the tree would have more than 2^k nodes.* ***Contradiction.***

Union by Depth

Making Find Operation Efficient

Theorem. For any leader $\bar{x}$, the size of the tree of $\bar{x}$ is at least $2^{\text{depth}(\bar{x})}$.

Proof by Induction:

- **Base Case:** If $\text{depth}(\bar{x}) = 0$, then $\bar{x}$ is the leader of a singleton set, so the size of $\bar{x}$'s set is $2^0 = 1$.
- **Inductive Step:** Assume that for any set leader $\bar{y}$ with depth $d - 1$, the size of $\bar{y}$'s set is at least 2^{d-1}.
  - When $\text{depth}(\bar{x})$ becomes d for the first time, $\bar{x}$ is the leader of the union of two sets, both of whose leaders had depth $d - 1$.
  - By the inductive hypothesis, both component sets had at least 2^{d-1} elements.
  - Therefore, the new set has at least 2^d elements.

Conclusion

Since there are at most n elements in total, the maximum depth of any set is $\log n$. Therefore, both Find and Union run in $\Theta(\log n)$ time in the worst case.

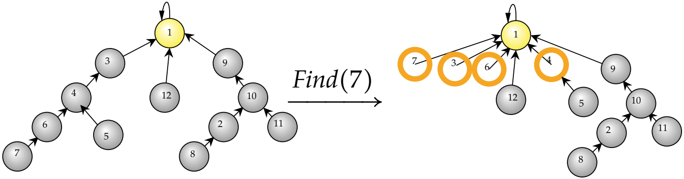
Path Compression

Optimizing Find Operation

- Path compression flattens the structure, making future Find operations quicker.
- Path compression makes every node on the Find path from x to the root point directly to the root.

Find(x)

```
if x ≠ parent(x)
  parent(x) ←
Find(parent(x))
return parent(x)
```



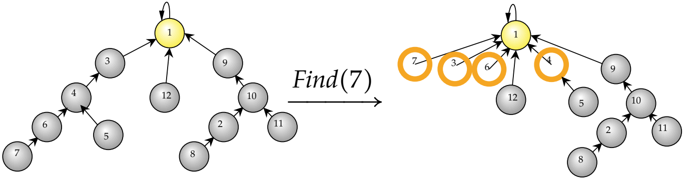
PATH COMPRESSION

OPTIMIZING FIND OPERATION

- Path compression flattens the structure, making future FIND operations quicker.
- Path compression makes every node on the FIND path from x to the root point directly to the root.

FIND(x)

```
if x ≠ parent(x)
  parent(x) ←
FIND(parent(x))
return parent(x)
```



Nodes on the FIND path (highlighted in orange) are directly attached to the root.

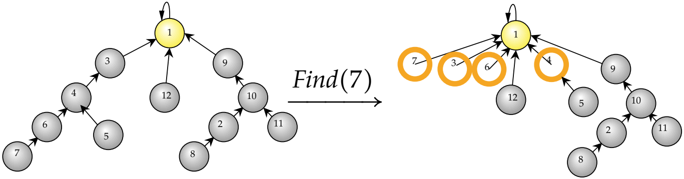
Path Compression

Optimizing Find Operation

- Path compression flattens the structure, making future Find operations quicker.
- Path compression makes every node on the Find path from x to the root point directly to the root.

Find(x)

```
if x ≠ parent(x)
  parent(x) ←
Find(parent(x))
return parent(x)
```



Teaser for Discussion Section - A $\log^*(n)$ Analysis

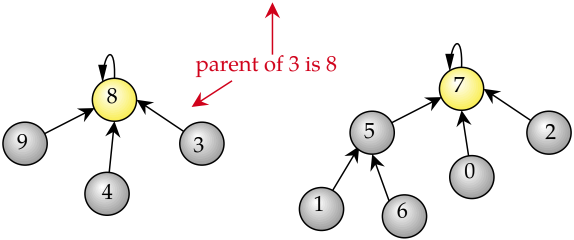
We will perform an amortized analysis showing how path compression can provide a super-exponential improvement, making the Find operation almost constant time.

Union-Find

A Final Thought on Array Representation

- Union-Find can be viewed as a reversed search tree.
- Efficiency comes from the simplicity of tracking roots with only parent pointers.
    - Use an array `parent[]` of length n.
    - `parent[i] = j` means the parent of element i is element j.

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



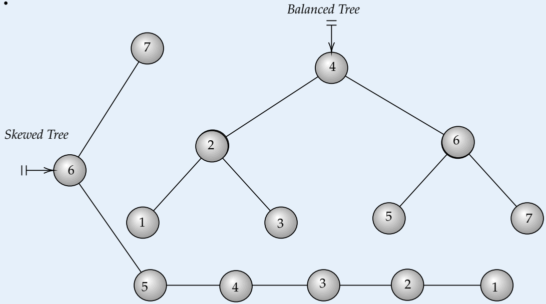
Demystify Binary Search Trees

Introduction to Binary Search Trees

Why Balanced Trees?

- We can construct any tree we want, but in this course, our focus is on efficiency.
- From the union-find example, we learned that it's better to have a balanced tree rather than a skewed one.

Can you find the number 7 if you start searching from the root of the tree?

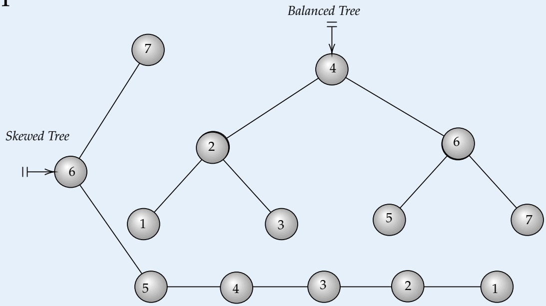


Introduction to Binary Search Trees

Why Balanced Trees?

- We can construct any tree we want, but in this course, our focus is on efficiency.
- From the union-find example, we learned that it's better to have a balanced tree rather than a skewed one.

Why do we prefer balanced trees over skewed ones?



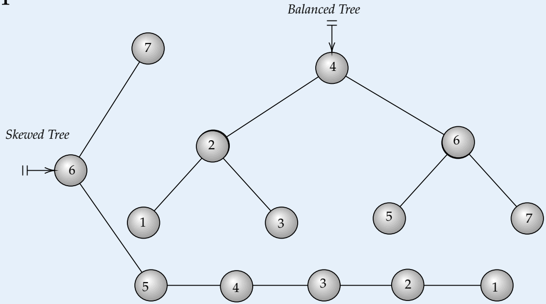
Introduction to Binary Search Trees

Why Balanced Trees?

- We can construct any tree we want, but in this course, our focus is on efficiency.

The complexity of an efficient traversal is proportional to the height!

Why do we prefer balanced trees over skewed ones?

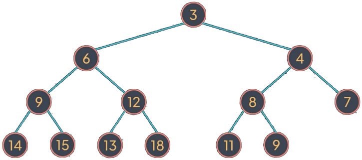


Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.



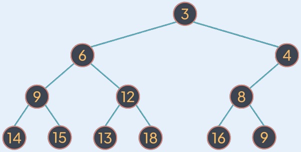
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

🤔Which level is not completely filled in this tree?



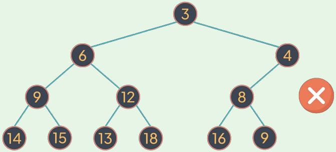
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

Which level is not completely filled in this tree?



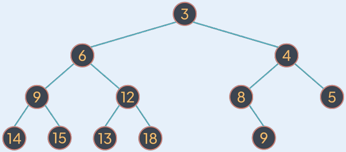
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

🤔Which rule did we violate when filling the last level of this tree?



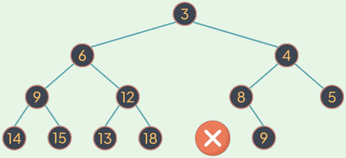
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

Which rule did we violate when filling the last level of this tree?



A Note for Coders

- Constructing trees in programming often requires manipulating pointers and structures.

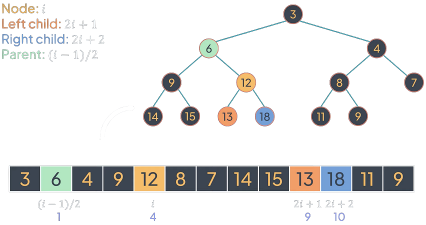
```
typedef struct treenode {           /* tree node struct */
    int data;                       /* integer field */
    struct treenode *left;          /* pointer to the left child */
    struct treenode *right;         /* pointer to the right child */
} node;
```

A NOTE FOR CODERS

- Constructing trees in programming often requires manipulating pointers and structures.

```
typedef struct treenode {              /* tree node struct */
    int data;                          /* integer field */
    struct treenode *left;             /* pointer to the left child */
    struct treenode *right;            /* pointer to the right child */
} node;
```

- For balanced trees, we can represent them simply using an array:
  - The children of node i are at positions $2i + 1$ and $2i + 2$.
  - The parent of node i is at position $(i - 1) \div 2$.



Example of Array Representation

Rule of Thumb

- The children of node i are at positions $2i+1$ and $2i+2$.
- The parent of node i is at position $(i-1) \div 2$.

0	1	2				
★	C	C				

- The root is at position 0, with children at positions 1 and 2.

$$1 = 2 * 0 + 1 \quad 2 = 2 * 0 + 2 \quad \text{No parent since } (0-1) \div 2 < 0$$

Example of Array Representation

Rule of Thumb

- The children of node i are at positions $2i + 1$ and $2i + 2$.
- The parent of node i is at position $(i - 1) \div 2$.

0	1		3	4		
P	★		C	C		

- The node at position 1 has its parent at position 0 and children at positions 3 and 4.

Example of Array Representation

Rule of Thumb

- The children of node i are at positions $2i + 1$ and $2i + 2$.
- The parent of node i is at position $(i - 1) \div 2$.

0		2			5	6
P		$\star$			C	C

A Note for Math Enthusiasts

Binary Balanced Trees: Height → Size

Question 1: If I give you a complete tree of height h, how many nodes does it have?

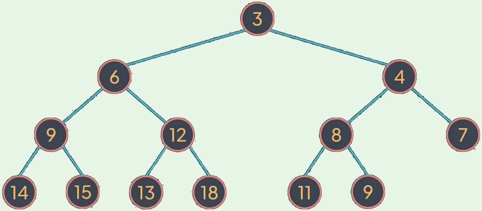
Definition of Height

- The height of a tree is the length of the path from the root to its farthest leaf node.
- A tree with only a root node has a height of 0, while an empty tree has a height of -1.

A Note for Math Enthusiasts

Binary Balanced Trees: Height → Size

Question 1: If I give you a complete tree of height *h*, how many nodes does it have?



The total number of nodes is:

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

A Note for Math Enthusiasts

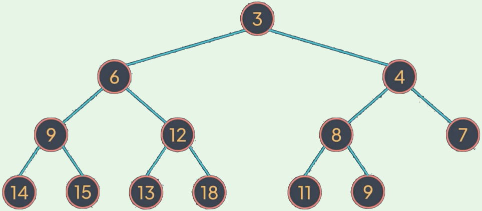
Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n - 1$ total nodes, how many nodes are at each level?

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

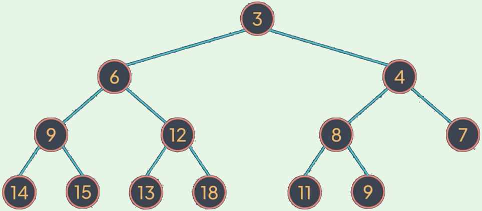
$$1 + 2 + 4 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

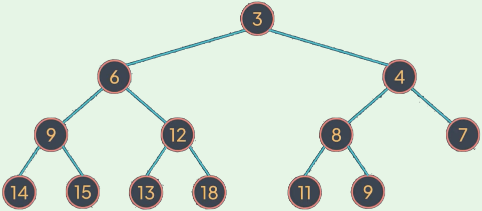
$$1 + 2 + 4 + \cdots + 2^{h-1} + 2^{h} = \underbrace{2^{h+1} - 1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

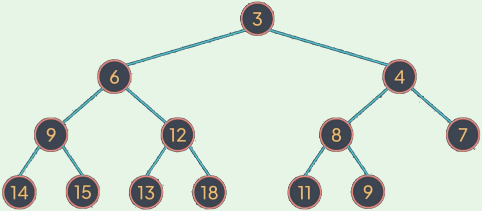
$$1 + 2 + 4 + \cdots + 2^{h-1} + \underbrace{2^h}_{\frac{n}{2}} = \underbrace{2^{h+1} - 1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

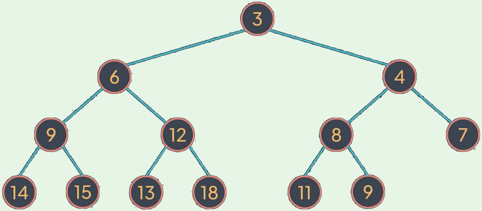
$$1 + 2 + 4 + \cdots + \underbrace{2^{h-1}}_{\frac{n}{4}} + \underbrace{2^{h}}_{\frac{n}{2}} = \underbrace{2^{h+1} - 1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

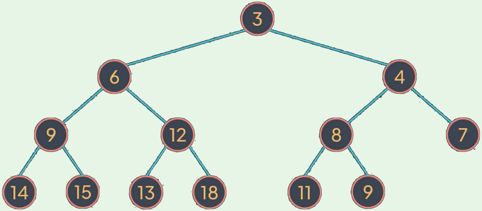
$$1+2+\underbrace{4}_{\frac{n}{2^{h-1}}}+\cdots+\underbrace{2^{h-1}}_{\frac{n}{4}}+\underbrace{2^{h}}_{\frac{n}{2}}=\underbrace{2^{h+1}-1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n-1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

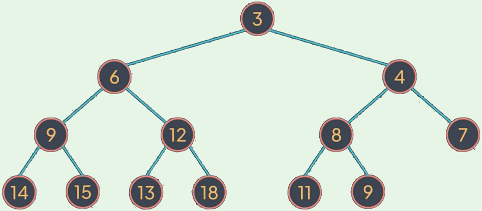
$$1 + \underbrace{2}_{\frac{n}{2^h}} + \underbrace{4}_{\frac{n}{2^{h-1}}} + \cdots + \underbrace{2^{h-1}}_{\frac{n}{4}} + \underbrace{2^h}_{\frac{n}{2}} = \underbrace{2^{h+1} - 1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Number of Nodes Per Level

Question 2: If I give you a complete tree with $n - 1$ total nodes, how many nodes are at each level?



- Reverse engineering the previous equation from bottom to top:

$$\underbrace{1}_{\frac{n}{2^{h+1}}} + \underbrace{2}_{\frac{n}{2^{h}}} + \underbrace{4}_{\frac{n}{2^{h-1}}} + \cdots + \underbrace{2^{h-1}}_{\frac{n}{4}} + \underbrace{2^{h}}_{\frac{n}{2}} = \underbrace{2^{h+1} - 1}_{n-1}$$

- **Remark:** 50% of nodes are leaves ☺

A Note for Math Enthusiasts

Binary Balanced Trees: Size → Height

Question 3: If I give you a complete tree with n nodes, what is its height?

A Note for Math Enthusiasts

Binary Balanced Trees: Size → Height

Question 3: If I give you a complete tree with n nodes, what is its height?

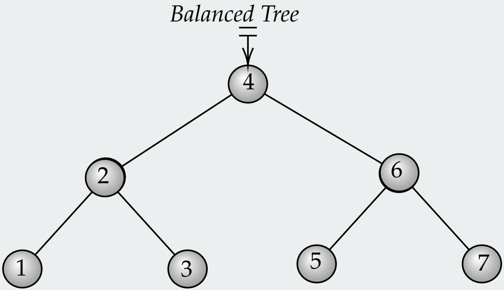
- We know that a tree of height h has $2^{h+1} - 1$ nodes.
- Therefore:

$$n = 2^{h+1} - 1 \quad \Rightarrow \quad h = \log_2 n - 1 = \Theta(\log n)$$

A Note for Math Enthusiasts

Search Efficiency in a Balanced Binary Search Tree

Question 4: In a well-balanced and sorted binary search tree, how efficient is your algorithm in finding a number?



- Given the logarithmic height of the tree, the search operation will take $\Theta(\log n)$ time.

Binary Heaps

Earliest Deadline First

A Classical Problem from Operating Systems

- Imagine you have multiple processes to be executed.
- Each process comes with a deadline.
- New processes keep arriving at the operating system.

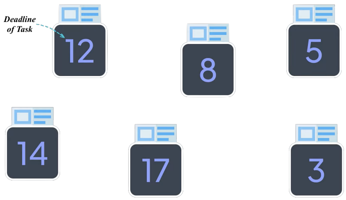
Earliest Deadline First

A Classical Problem from Operating Systems

- Imagine you have multiple processes to be executed.
- Each process comes with a deadline.
- New processes keep arriving at the operating system.

EDF Task

The CPU or server always executes the process with the earliest deadline.



Earliest Deadline First

Transforming a Dynamic Problem into a Data Structure Problem

Intuitive Goal

- We need to efficiently manage processes as they arrive and meet their deadlines.
- The challenge is to design a data structure that supports all required operations in the best possible time.

Earliest Deadline First

Transforming a Dynamic Problem into a Data Structure Problem

Intuitive Goal

- We need to efficiently manage processes as they arrive and meet their deadlines.
- The challenge is to design a data structure that supports all required operations in the best possible time.

Objective. Design an efficient data structure for managing processes.

Let's consider a set of processes $\mathcal{P} := \{p \mid \text{set of } n \text{ processes}\}$.

- **FindMin**$(\mathcal{P})$**:** Identify the process with the earliest deadline.
- **DeleteMin**$(\mathcal{P})$**:** Remove the process with the earliest deadline.

Earliest Deadline First

Transforming a Dynamic Problem into a Data Structure Problem

Intuitive Goal

- We need to efficiently manage processes as they arrive and meet their deadlines.
- The challenge is to design a data structure that supports all required operations in the best possible time.

Objective. Design an efficient data structure for managing processes.

Let's consider a set of processes $\mathcal{P} := \{p \mid \text{set of } n \text{ processes}\}$.

- **FindMin**($\mathcal{P}$)**:** Identify the process with the earliest deadline.
- **DeleteMin**($\mathcal{P}$)**:** Remove the process with the earliest deadline.
- **DeleteProcess**($\mathcal{P}, p$)**:** Remove a specific process p.

Earliest Deadline First

Transforming a Dynamic Problem into a Data Structure Problem

Intuitive Goal

- We need to efficiently manage processes as they arrive and meet their deadlines.
- The challenge is to design a data structure that supports all required operations in the best possible time.

Objective. Design an efficient data structure for managing processes.

Let's consider a set of processes $\mathcal{P} := \{p \mid \text{set of } n \text{ processes}\}$.

- **FindMin**$(\mathcal{P})$**:** Identify the process with the earliest deadline.
- **DeleteMin**$(\mathcal{P})$**:** Remove the process with the earliest deadline.
- **DeleteProcess**$(\mathcal{P}, p)$**:** Remove a specific process p.
- **Update**$(\mathcal{P}, p,$ **NewDeadline**$)$**:** Update the deadline of process p to a new value.

Earliest Deadline First

Transforming a Dynamic Problem into a Data Structure Problem

Intuitive Goal

- We need to efficiently manage processes as they arrive and meet their deadlines.
- The challenge is to design a data structure that supports all required operations in the best possible time.

Objective. Design an efficient data structure for managing processes.

Let's consider a set of processes $\mathcal{P} := \{p \mid \text{set of } n \text{ processes}\}$.

- **FindMin**$(\mathcal{P})$**:** Identify the process with the earliest deadline.
- **DeleteMin**$(\mathcal{P})$**:** Remove the process with the earliest deadline.
- **DeleteProcess**$(\mathcal{P}, p)$**:** Remove a specific process p.
- **Update**$(\mathcal{P}, p, \textbf{NewDeadline})$**:** Update the deadline of process p to a new value.
- **Insert**$(\mathcal{P}, p_{\textbf{new}}, \textbf{Deadline})$**:** Insert a new process with a specified deadline.
- **Build**$(\mathcal{P})$**:** Construct the data structure with the set of processes $\mathcal{P}$.

Earliest Deadline First

Bad Solution #1

🤔If we store processes in an unsorted array, what is the cost of **FindMin**($\mathcal{P}$)?

Earliest Deadline First

Bad Solution #1

🤔If we store processes in an unsorted array, what is the cost of **FindMin**$(\mathcal{P})$?

Computing the minimum in an unsorted array costs $\Omega(n)$.

Earliest Deadline First

Bad Solution #2

🤔If we store processes in a sorted array, what is the cost of **FindMin**($\mathcal{P}$)?

Earliest Deadline First

Bad Solution #2

🤔If we store processes in a sorted array, what is the cost of **FindMin**$(\mathcal{P})$?

Computing the minimum in a sorted array costs $\Theta(1)$.

Earliest Deadline First

Bad Solution #2

If we store processes in a sorted array, what is the cost of **Insert**$(\mathcal{P}, p_{\textbf{new}}, \textbf{Deadline})$?

Earliest Deadline First

Bad Solution #2

🤔If we store processes in a sorted array, what is the cost of **Insert**$(\mathcal{P}, p_{\mathbf{new}}, \mathbf{Deadline})$?

Inserting into and expanding a sorted array costs $\Omega(n)$.

Remember: Insertion Sort

Earliest Deadline First

🤔Does anyone have an idea for a better solution?

Earliest Deadline First

Priority Queue as a Tree

🤔Does anyone have an idea for a better solution?

Eureka: What if we use a tree structure with the minimum element at the root?

Earliest Deadline First

Priority Queue as a Tree

🤔Does anyone have an idea for a better solution?

Eureka: What if we use a tree structure with the minimum element at the root?

🤔How should we order the remaining elements?

Earliest Deadline First

Priority Queue as a Tree

🤔Does anyone have an idea for a better solution?

Eureka: What if we use a tree structure with the minimum element at the root?

🤔How should we order the remaining elements?

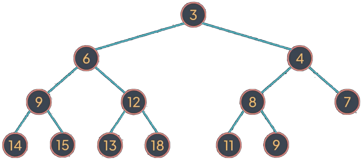
1) Divide in the middle to create a balanced tree.
2) Apply the same logic recursively to each partition.

Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.



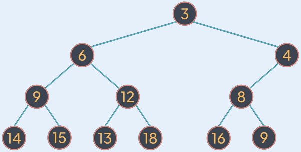
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

🤔Which level is not completely filled in this tree?



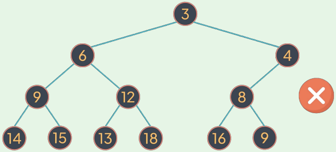
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

Which level is not completely filled in this tree?



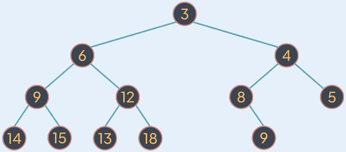
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

🤔Which rule did we violate when filling the last level of this tree?



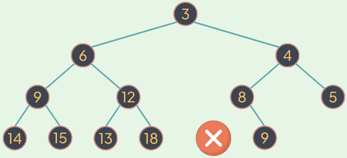
Binary Balanced Tree

Definition

A binary tree is fully balanced if and only if all levels, except possibly the last, are completely filled from left to right.
Convention: The last level is filled with priority from left to right.

Imagine that in place of any missing nodes, we insert a dummy value, such as $+\infty$ or $-\infty$.

Which rule did we violate when filling the last level of this tree?



A Note for Coders

- Constructing trees in programming often requires manipulating pointers and structures.

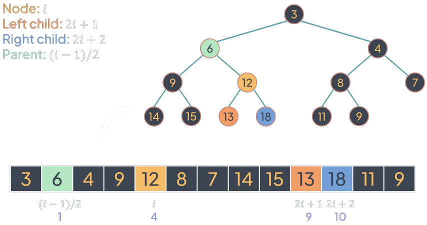
```
typedef struct treenode {            /* tree node struct */
    int data;                        /* integer field */
    struct treenode *left;           /* pointer to the left child */
    struct treenode *right;          /* pointer to the right child */
} node;
```

A Note for Coders

- Constructing trees in programming often requires manipulating pointers and structures.

```
typedef struct treenode {               /* tree node struct */
    int data;                           /* integer field */
    struct treenode *left;              /* pointer to the left child */
    struct treenode *right;             /* pointer to the right child */
} node;
```

- For balanced trees, we can represent them simply using an array:
    - The children of node i are at positions $2i + 1$ and $2i + 2$.
    - The parent of node i is at position $(i - 1) \div 2$.

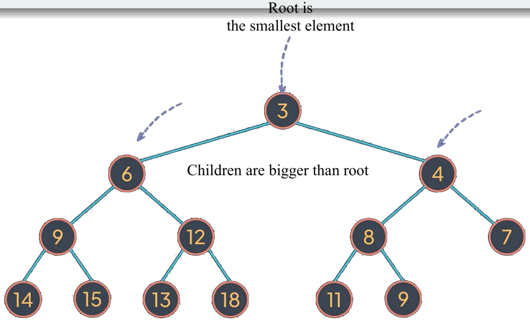


Binary Heap

A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

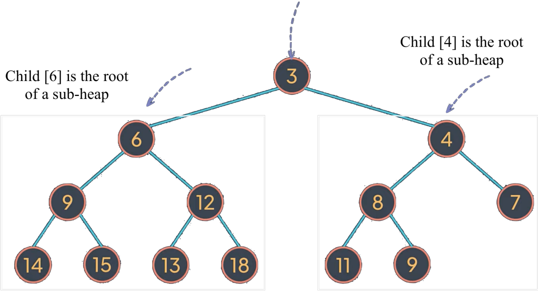


Binary Heap

A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

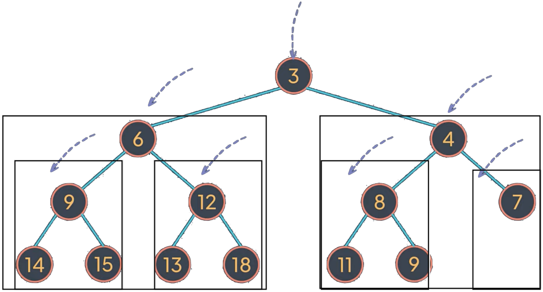


Binary Heap

A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

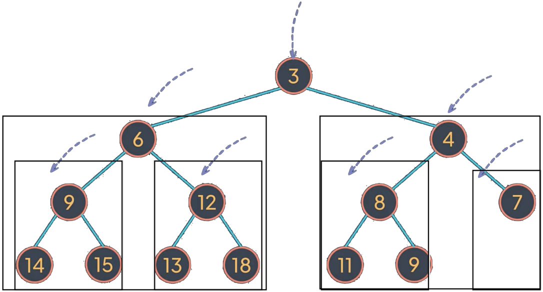


Binary Heap

A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.



🤔How do we store such a tree efficiently?

Binary Heap

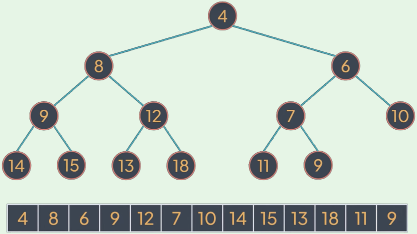
A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

🤔How do we store such a tree efficiently?

Since it is a balanced binary tree, we can use an array representation.



Binary Heap

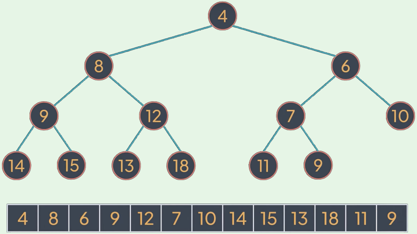
A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

But inserting into an array always costs at least $\Omega(n)$, right?

Since it is a balanced binary tree, we can use an array representation.



Binary Heap

A Picture is Worth a Thousand Words

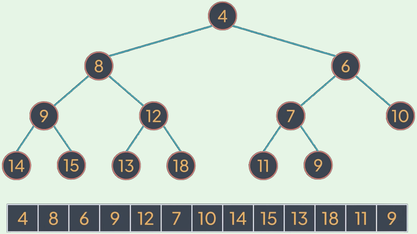
Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

We can always insert an element at the end of array at $\Theta(1)$.

Remember ***Realloc*** *in C, C++, etc.*

Since it is a balanced binary tree, we can use an array representation.



Binary Heap

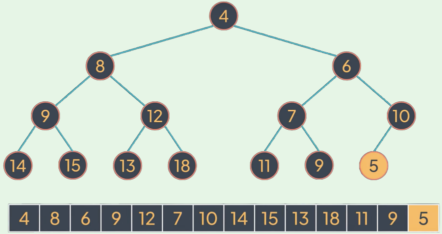
A Picture is Worth a Thousand Words

Definition of a Binary Heap

A binary heap is a balanced binary tree where the root of every (sub)tree is its minimum element.

🤔How do we maintain the tree as a binary heap?

Since it is a balanced binary tree, we can use an array representation.

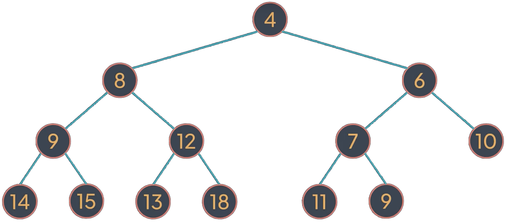


Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



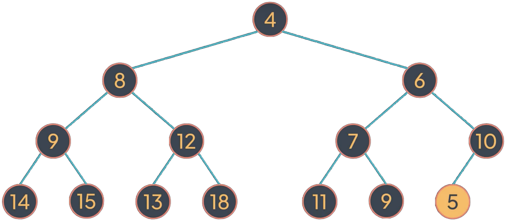
4	8	6	9	12	7	10	14	15	13	18	11	9

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	6	9	12	7	10	14	15	13	18	11	9	5

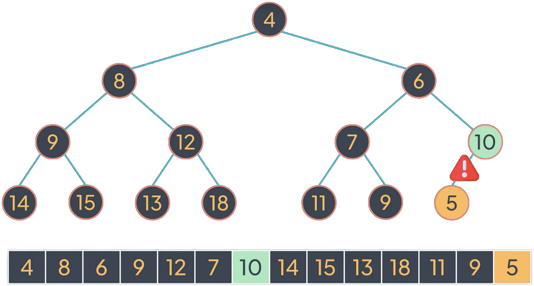
🤔What operation do we perform after adding a new element to maintain the heap property?

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



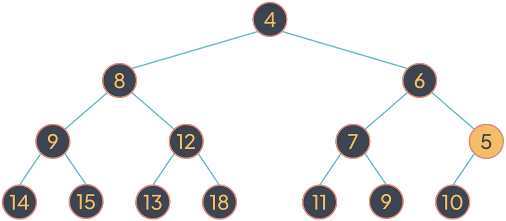
We perform a "shift up" operation to place the new element and correct the subtree $[10, 5, null]$.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	6	9	12	7	5	14	15	13	18	11	9	10

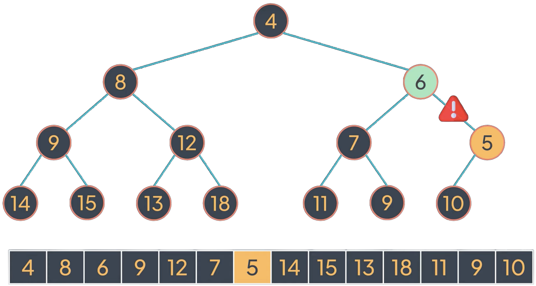
🤔Should we continue? · Do we need to check the entire heap?

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



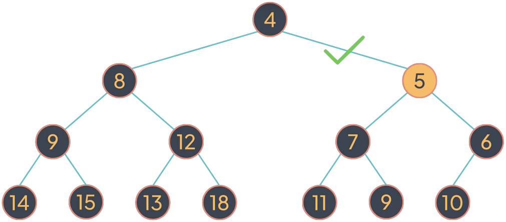
Yes! · All the subtrees that do not include the new element are already correct.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



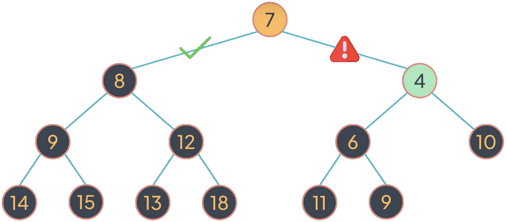
4	8	5	9	12	7	6	14	15	13	18	11	9	10

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



7	8	4	9	12	6	10	14	15	13	18	11	9

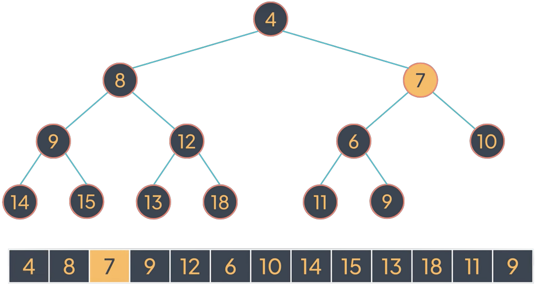
🤔If we change an element in the min-heap (w.l.o.g., increase), what do we have to do?

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



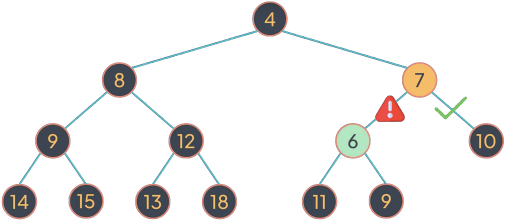
Shift down until the heap property is restored.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	7	9	12	6	10	14	15	13	18	11	9

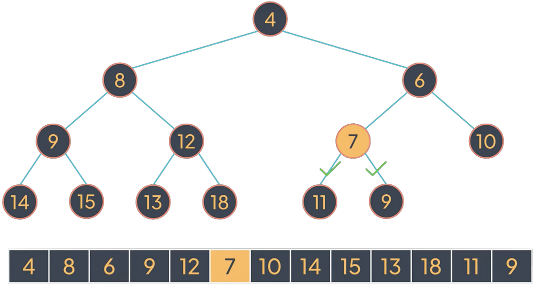
Shift down until the heap property is restored.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



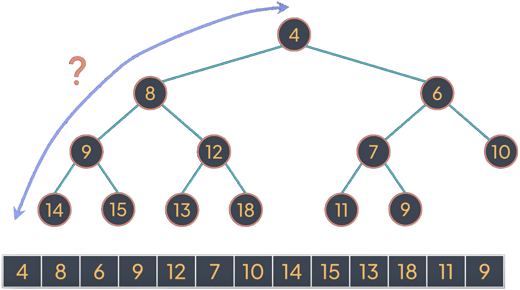
Shift down until the heap property is restored.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



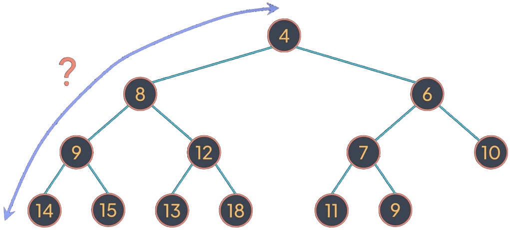
- ⋆ What is the complexity of any shift up & shift down?
- ⋆ Can the update process create a cycle?
- ⋆ What if we change an element in the middle?

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	6	9	12	7	10	14	15	13	18	11	9

Shift-Up	Shift-Down
$O(\log n)$	$O(\log n)$

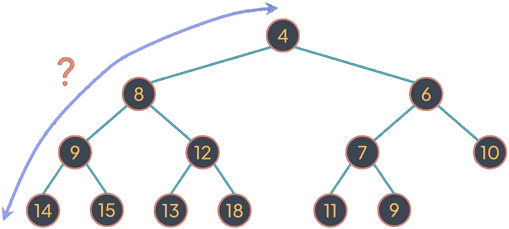
⋆ In the worst case, we move from the root to the leaf.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	6	9	12	7	10	14	15	13	18	11	9

Rule of Thumb:

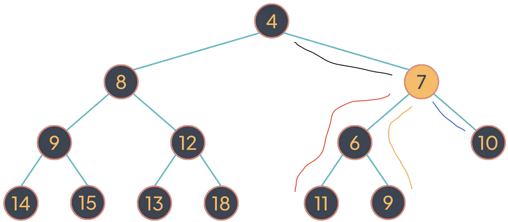
$$\begin{cases} \text{If priority } \uparrow \Rightarrow \text{Shift down} \\ \text{If priority } \downarrow \Rightarrow \text{Shift up} \end{cases}$$

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



4	8	7	9	12	6	10	14	15	13	18	11	9

The subtree structure remains valid except for the affected branch after an update.

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."

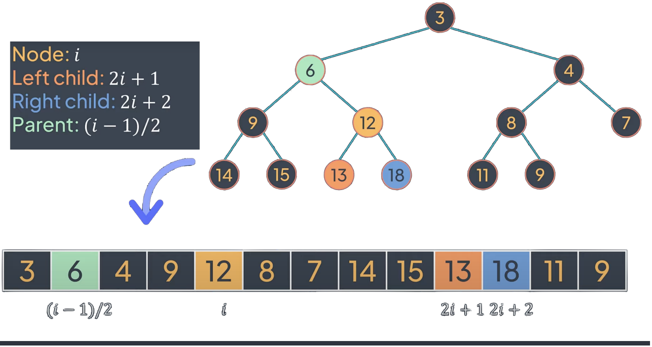
🤔How do we know the parent & children indices are in $O(1)$?

Shift Up & Shift Down

It's all about swaps

Rule of Thumb

Whether you Insert or Update in the heap, always perform swaps to maintain:
The heap rule: "In a binary heap, the root of every (sub)tree is its minimum element."



Delete from Heap

Swap with Last Element

🤔Deleting an element from an array typically costs at least $\Omega(n)$, right?

Delete from Heap

Swap with Last Element

🤔Deleting an element from an array typically costs at least $\Omega(n)$, right?

We can delete the last element of the array in $\Theta(1)$ time.

*Remember **Realloc** in C, C++, etc.*

Delete from Heap

Swap with Last Element

🤔Deleting an element from an array typically costs at least $\Omega(n)$, right?

We can delete the last element of the array in $\Theta(1)$ time.
*Remember **Realloc** in C, C++, etc.*

Deletion Strategy

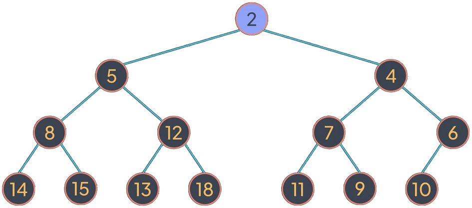
1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.

Delete from Heap

Example

Short Strategy

1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.



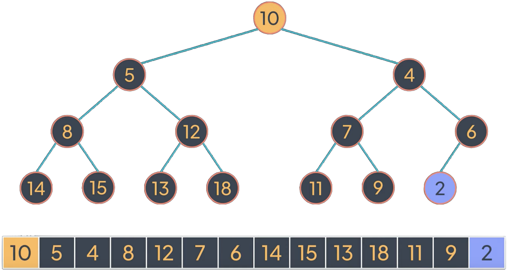
2	5	4	8	12	7	6	14	15	13	18	11	9	10

Delete from Heap

Example

Short Strategy

1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.

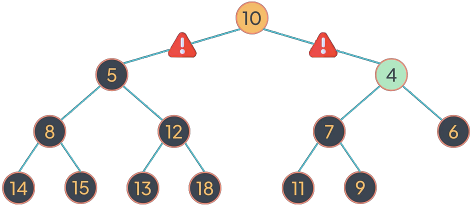


Delete from Heap

Example

Short Strategy

1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.



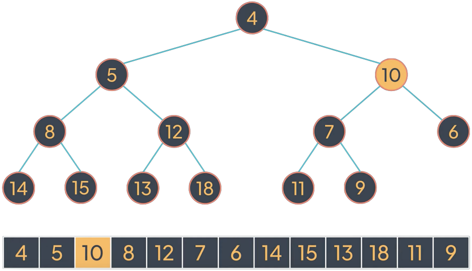
10	5	4	8	12	7	6	14	15	13	18	11	9

Delete from Heap

Example

Short Strategy

1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.

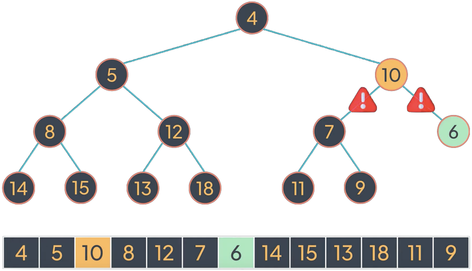


Delete from Heap

Example

Short Strategy

1. Swap the element to be deleted with the last element in the heap.
2. Remove the last element from the array (reallocate).
3. Perform a **shift down** to restore the heap property.



Building a Heap

Build($\mathcal{P}$):Trivial Way

- Assume you have a set of processes $\mathcal{P} = \{p_1, \cdots, p_n\}$ with priorities/deadlines $(v_1, \cdots, v_n)$.

Building a Heap

Build($\mathcal{P}$):Trivial Way

- Assume you have a set of processes $\mathcal{P} = \{p_1, \cdots, p_n\}$ with priorities/deadlines $(v_1, \cdots, v_n)$.
- Start with an empty heap $H = \emptyset$ and insert each element one by one using **Insert**(H, p_i, v_i).

$$\text{Runtime: } n \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$$

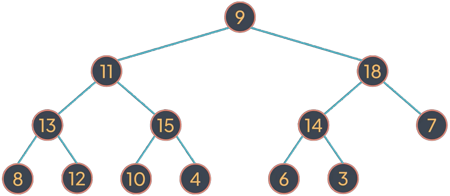
Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

Instead of inserting elements one by one, we can take the entire array and turn it into a heap in a more efficient way.
This process is called **heapify**.

- Start by treating the array as a binary tree.

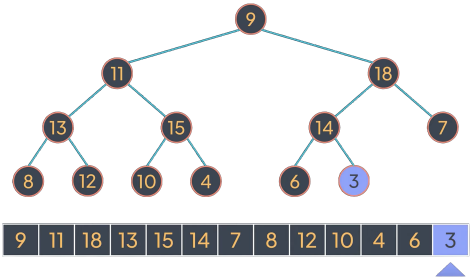


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.

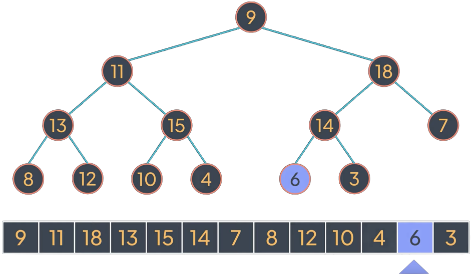


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

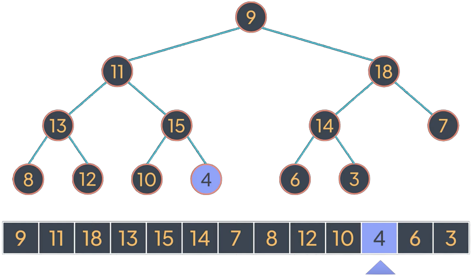


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

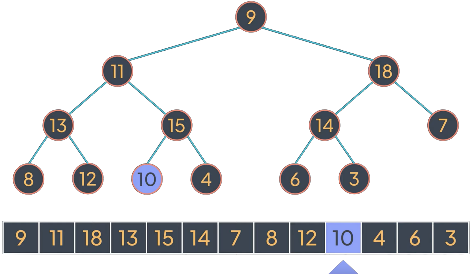


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

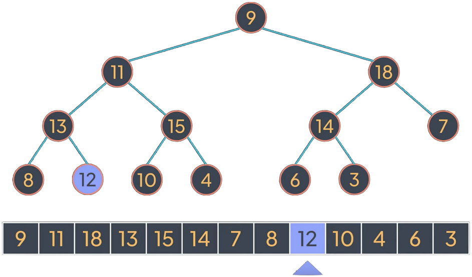


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

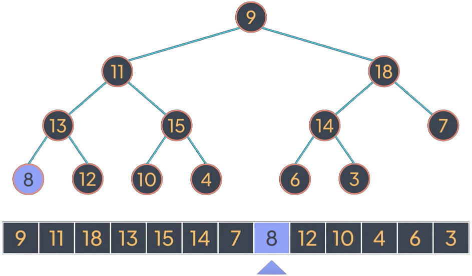


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

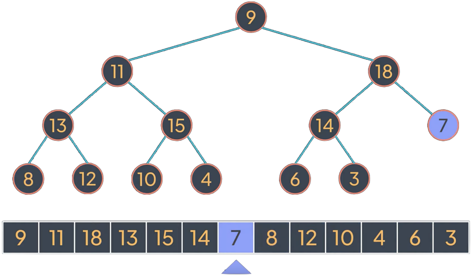


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

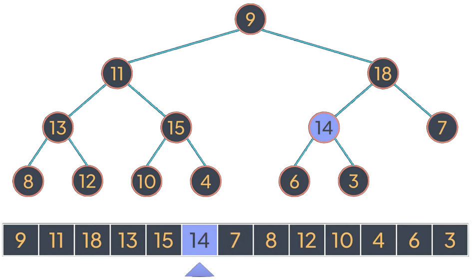


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

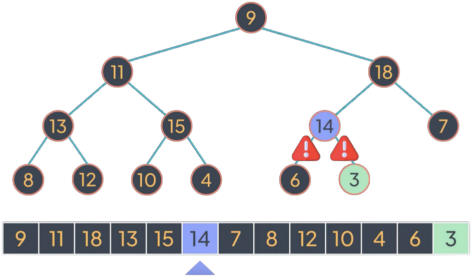


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

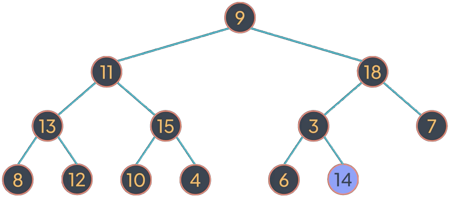


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



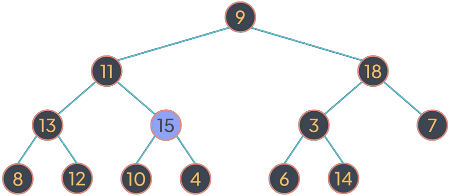
9	11	18	13	15	3	7	8	12	10	4	6	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



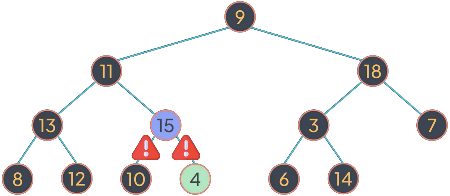
9	11	18	13	15	3	7	8	12	10	4	6	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



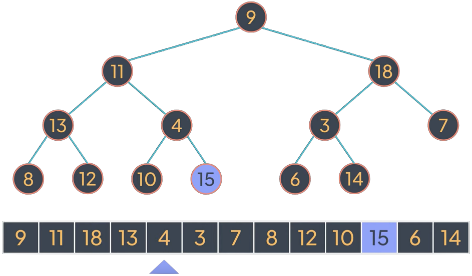
9	11	18	13	15	3	7	8	12	10	4	6	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

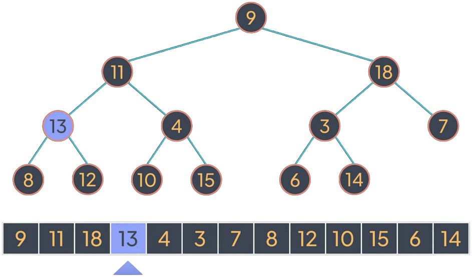


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

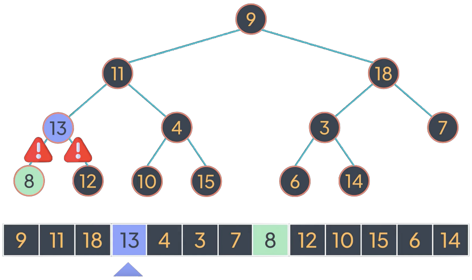


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

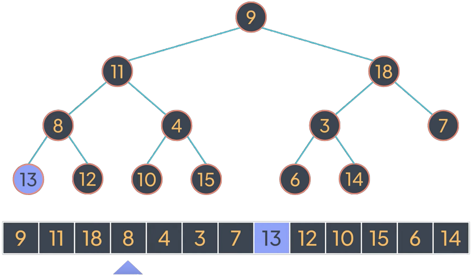


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

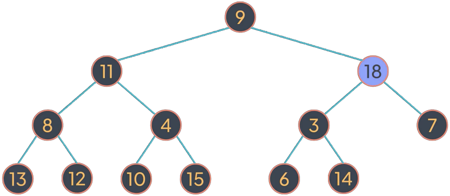


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



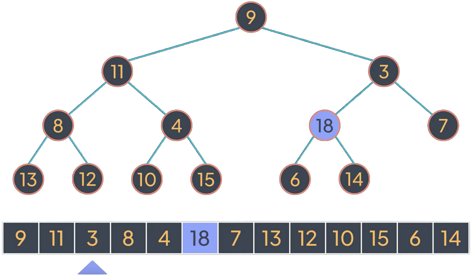
9	11	18	8	4	3	7	13	12	10	15	6	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

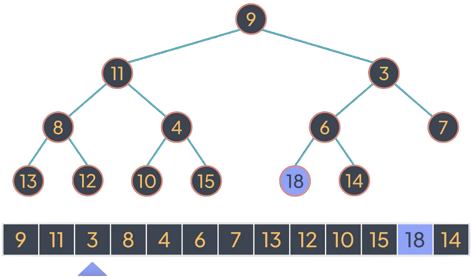


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

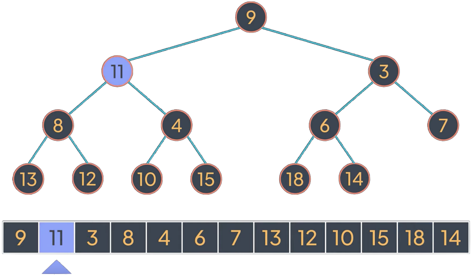


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

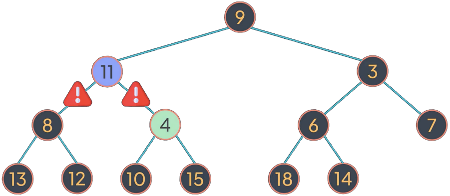


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



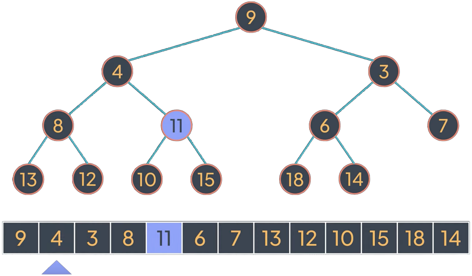
9	11	3	8	4	6	7	13	12	10	15	18	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

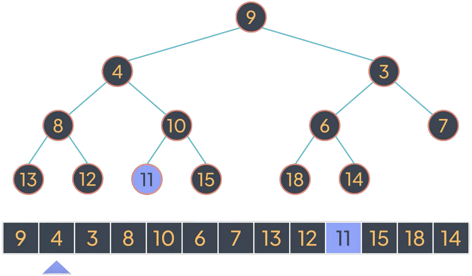


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

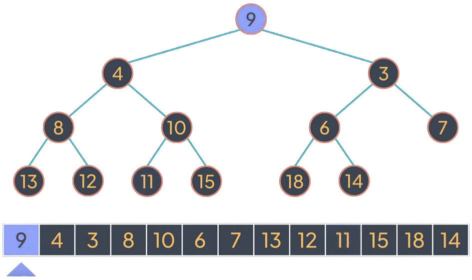


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

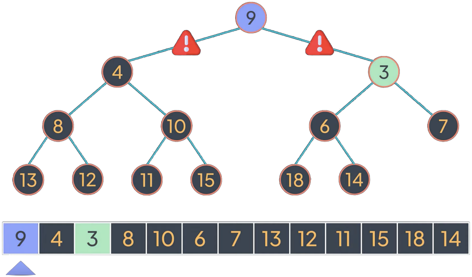


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

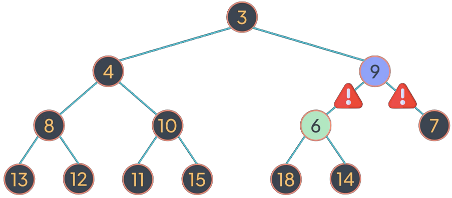


Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



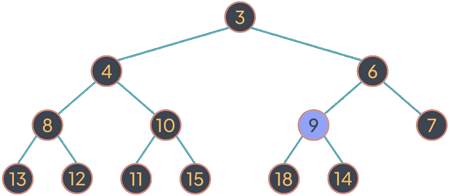
3	4	9	8	10	6	7	13	12	11	15	18	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



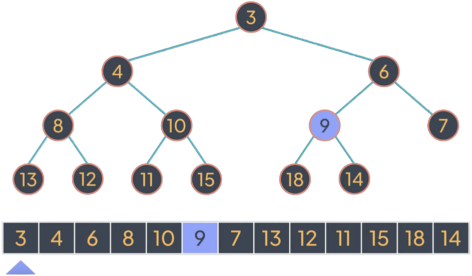
3	4	6	8	10	9	7	13	12	11	15	18	14

Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.



Building a Heap

Clever Way

Build($\mathcal{P}$): Heapify Method

- Begin from the last non-leaf node, and perform a "shift down" operation to ensure each subtree satisfies the heap property.
- Continue this process moving upward to the root.

Worst-case Scenario

Every correction takes a shift-down $\mathcal{O}(\log n) \Rightarrow$
Runtime:$\mathcal{O}(n \log n)$.

But...

Optimistic Observation

Expensive corrections are rare & cheap corrections are frequent*

* *Remember: 50% of nodes are leaves and 25% of nodes are parents of leaves.*

Total Swaps

Proof Sketch

Observation: Every level has different height. Shift-down costs actually $\mathcal{O}(\textit{height of subtree})$.

$$
\left.
\begin{array}{c}
\frac{n}{2}:\ \text{(the leaves)}\ \times 0 \\
+ \\
\frac{n}{4}:\ \text{(parents of leaves)}\ \times 1 \\
+ \\
\frac{n}{8}:\ \text{(grandparents of leaves)}\ \times 2 \\
+ \\
\cdots + \\
1:\ \text{root}\ \times \mathcal{O}(\log n)
\end{array}
\right. = \mathcal{O}(n)
$$

HeapSort

How to Sort an Array Using a Heap

🤔How we can sort an array using a min-heap?

HeapSort

How to Sort an Array Using a Heap

🤔How we can sort an array using a min-heap?

Heapsort Explained

- First, build a max heap from the array using **heapify**.
- Then, repeatedly remove the root (the minimum element) and place it at the end of the array.
- After each removal, perform a "shift down" to restore the heap property.

$$\textbf{Runtime: } \underbrace{\text{heapify}}_{\mathcal{O}(n)} + n \times \underbrace{\text{delete}}_{\mathcal{O}(\log n)} + n \times \underbrace{\text{swaps}}_{\mathcal{O}(1)}$$

Find k-th element

How to Sort an Array Using a Heap

How we can compute the k-th smallest element of an array using a min-heap?

Find k-th element

How to Sort an Array Using a Heap

How we can compute the k-th smallest element of an array using a min-heap?

Two solutions (I will give you the complexity-describe the algorithm & application)

1. $\Theta(n) + O(k \log n)$
2. $\Theta(k) + O(n \log k)$

Find k-th element

How to Sort an Array Using a Heap

How we can compute the k-th smallest element of an array using a min-heap?

Two solutions (I will give you the complexity-describe the algorithm & application)

1. $\Theta(n) + O(k \log n)$
2. $\Theta(k) + O(n \log k)$

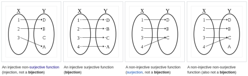
Application: First version is offline. The second is online.

Appendix

References

IMAGE SOURCES I

https://brand.wisc.edu/web/logos/



https://en.wikipedia.org/wiki/Bijection