Competitive Programming
Winter Term 23/24

Treaps: the only data structure you’ll ever need

Kirill Simonov
Algorithm Engineering Group
HPI

Christopher Weyand
Scalable Algorithms Group
KIT
Overview for Today

- Treaps
- ...
- profit

LOOK AT ME

I'M YOUR DATA STRUCTURE NOW
Impressions

“D. A. Heger (2004) presented a performance comparison of binary search trees. Treap was found to have the best average performance”

— Wikipedia

“Treaps are hella slow, but easy to implement and customize.”

— some russian dude
Gleb Evstropov
ICPC 2013, 14, 15 Gold Medalist

“Would recommend.”

— 9/10 Dentists

“First implementation was broken, but worked fine after refund.”

— Me
Jack of all Trades
Jack of all Trades
What a Treap can do

Self balancing BSTs
(like a set)

- find value
- insert value
- delete value

binary search tree
Treap = BST + Heap
Treap = BST + Heap

Heap

3

11

5

17

13

37

41

83

10^9 + 7

parent smaller than children
Treap = BST + Heap

- **Heap**
  - Parent smaller than children
  - Example: 3 (parent), 11, 5 (children)

- **BST**
  - Left child < parent < right child
  - Example: 3 (parent), 11 (left child), 5 (right child)

- **Example Trees**
  - **Heap**:
    - 3 (root)
    - 11 (left child)
    - 5 (right child)
    - 17, 13, 37, 41 (additional nodes)
    - 83, \(10^9 + 7\) (additional nodes)
  - **BST**:
    - 37 (root)
    - 13 (left child)
    - 83, \(10^9 + 7\) (right child)
    - 5, 17, 41 (additional nodes)
**Treap = BST + Heap**

- **Heap**
  - parent smaller than children
  - 3
     - 11
        - 17
           - 83
           - $10^9 + 7$
     - 13
     - 5
        - 37
        - 41

- **BST**
  - left child < parent < right child
  - 37
     - 13
        - 5
           - 3
     - 83
        - 10^9 + 7

Can we do both?
Treap = BST + Heap

- yes, we can do both, but ...
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
  → there cannot be a left child
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
  → there cannot be a left child
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
  → there cannot be a left child
- let’s give nodes a random second value (Priority)
- heap by priority; BST by key
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
  \[\rightarrow \text{there cannot be a left child}\]
- let’s give nodes a random second value (Priority)
- heap by priority; BST by key
Treap = BST + Heap

- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)
  → there cannot be a left child
- let’s give nodes a random second value (Priority)
- heap by priority; BST by key

can be shown that:

- expected depth of a node is $\log n$
- expected depth of a treap is $\log n$
- treap has $\log n$ depth w.h.p.

Probability that a treap is deeper than $10 \log n$ is smaller than $1/n$
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

**BST by key:**
left child < parent < right child

**Heap by Prio:**
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children

for now assume unique values
Unique?

BST by key:
left child < parent < right child

Heap by Prio:
parent < children

- for now assume unique values
- keys/prios uniquely determine a treap
Implementation

we break down everything to two operations
Implementation

we break down everything to two operations

split
Implementation

we break down everything to two operations

<table>
<thead>
<tr>
<th>split</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
</tr>
<tr>
<td>(preserves order)</td>
</tr>
</tbody>
</table>
Implementation

we break down everything to two operations

\[ \text{split} \quad \text{merge} \]
(preserves order)

E.g. \textbf{delete} x:
Implementation

we break down everything to two operations

E.g. **delete** $x$:

\[
\text{split } x
\]

split (preserves order)
Implementation

we break down everything to two operations

E.g. delete x:

split x

merge (preserves order)
Implementation

We break down everything to two operations: `split` and `merge` (preserves order).

E.g. **delete** $x$:

- **split** $x$
- **split** $x+1$

$\begin{align*}
\text{split } x & \leq x \\
\text{split } x+1 & > x
\end{align*}$
Implementation

we break down everything to two operations

E.g. **delete** x:
Implementation

we break down everything to two operations

E.g. delete x:

split x

split x+1

remove x, merge

merge (preserves order)
Implementation

we break down everything to two operations

split
merge
(preserves order)

E.g. **delete** x:

split x

split x+1

remove x, merge

\{< x\} \cup \{> x\}
Split

returns two treaps containing all nodes < and ≥ the given key respectively
Split

returns two treaps containing all nodes < and ≥ the given key respectively

\[
\text{split}(\text{Node } v, \text{Key } x)
\]

\[
\begin{align*}
\text{if } v.\text{key} &< x: \\
\text{else} &
\end{align*}
\]

\[
\ldots
\]
Split

returns two treaps containing all nodes < and ≥ the given key respectively

\[
\text{split}(\text{Node } v, \text{ Key } x)
\]

\[
\begin{align*}
\text{if } v.\text{key} &< x: \\
\text{rl, rr} &\text{ = split}(v.\text{right, } x)
\end{align*}
\]

\[
\text{else} \\
\ldots
\]
Split

returns two treaps containing all nodes < and ≥ the given key respectively

```plaintext
split(Node v, Key x)
if v.key < x:
    rl, rr = split(v.right, x)
    v.right = rl
else
    ...
```

Split

returns two treaps containing all nodes < and ≥ the given key respectively

```plaintext
split(Node v, Key x)
    if v.key < x:
        rl, rr = split(v.right, x)
        v.right = rl
        return {v, rr}
    else
        ...
```
Merge

merge two treaps into one preserving horizontal ordering (left, right)

merge(Node l, Node r)
merge two treaps into one preserving horizontal ordering (left, right)

```
merge(Node l, Node r)
    if l.prio < r.prio:
        ...
    else:
        ...
```
Merge

merge two treaps into one preserving horizontal ordering (left, right)

merge(Node l, Node r)
  if l.prio < r.prio:
    ...
  else
merge two treaps into one preserving horizontal ordering (left, right)

merge(Node l, Node r)

if l.prio < r.prio:
    tmp = merge(l.right, r)
else
    ...

merge two treaps into one preserving horizontal ordering (left, right)

```
merge(Node l, Node r)
    if l.prio < r.prio:
        tmp = merge(l.right, r)
        l.right = tmp
    else
        ...
```
merge two treaps into one preserving horizontal ordering (left, right)

merge(Node l, Node r)

if l.prio < r.prio:
    tmp = merge(l.right, r)
    l.right = tmp
    return l
else
    ...

What a Treap can do

Self balancing BSTs
(like a set)

- find value
- insert value
- delete value

binary search tree
What a Treap can do

Self balancing BSTs
(like a set)

- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values

(binary search tree)
What a Treap can do

Self balancing BSTs
(like a set)

- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values

binary search tree
aggregates
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
- recompute each time we modify children of a node
Aggregates

we now want to answer queries about our data
- what is the gcd of values in interval?
- sum of values in interval?
- each node maintains aggregates about its subtree
- recompute each time we modify children of a node

update(Node v)

\[
\begin{align*}
v.\text{gcd} &= \text{gcd}(v.\text{key}, \\
gcd(v.\text{left}), \\
gcd(v.\text{right})) \\
v.\text{sum} &= v.\text{key} + \text{sum}(v.\text{left}) \\
&\quad + \text{sum}(v.\text{right})
\end{align*}
\]
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
- recompute each time we modify children of a node
- requires aggregate to be combinable in $O(1)$

```
update(Node v)
    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))
    v.sum = v.key + sum(v.left) + sum(v.right)
```
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
- recompute each time we modify children of a node
- requires aggregate to be combinable in $O(1)$
- to answer questions about any interval $[l, r]$

```
update(Node v)
    v.gcd = gcd( v.key, 
gcd(v.left), 
gcd(v.right))

v.sum = v.key + sum(v.left) + sum(v.right)
```
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
- recompute each time we modify children of a node
- requires aggregate to be combinable in $O(1)$
- to answer questions about any interval $[l,r]$
  - split treap into 3 treaps: $< l, [l,r], > r$

**update**(Node $v$)

$$
\begin{align*}
  v.gcd &= \gcd(v.key, \gcd(v.left), \gcd(v.right)) \\
  v.sum &= v.key + \text{sum}(v.left) + \text{sum}(v.right)
\end{align*}
$$
Aggregates

we now want to answer queries about our data

■ what is the gcd of values in interval?
■ sum of values in interval?

■ each node maintains aggregates about its subtree
■ recompute each time we modify children of a node
■ requires aggregate to be combinable in $O(1)$
■ to answer questions about any interval $[l, r]$
  ■ split treap into 3 treaps: $< l$, $[l, r]$, $> r$
  ■ read aggregates from the middle treap

update(Node v)

$$v.gcd = \gcd(v.key, \gcd(v.left), \gcd(v.right))$$
$$v.sum = v.key + \text{sum}(v.left) + \text{sum}(v.right)$$

Competitive Programming | Algorithm Engineering Group (HPI) & Scalable Algorithms Group (KIT)
Aggregates

we now want to answer queries about our data

- what is the gcd of values in interval?
- sum of values in interval?

- each node maintains aggregates about its subtree
- recompute each time we modify children of a node
- requires aggregate to be combinable in $O(1)$
- to answer questions about any interval $[l,r]$
  - split treap into 3 treaps: $< l, [l,r], > r$
  - read aggregates from the middle treap
  - merge together into one again

update(Node $v$)

$v.gcd = \gcd(v.key, \gcd(v.left), \gcd(v.right))$
$v.sum = v.key + \text{sum}(v.left) + \text{sum}(v.right)$
Aggregates

- the code for aggregates is really simple
- just call **update** after each change
Aggregates

- the code for aggregates is really simple
- just call `update` after each change

```plaintext
split(Node v, Key x)
    if v.key < x:
        rl, rr = split(v.right, x)
        v.right = rl
        update(v)
    return {v, rr}
else
    ...
```
Aggregates

- The code for aggregates is really simple
- Just call `update` after each change

```c
split(Node v, Key x)
if v.key < x:
    rl, rr = split(v.right, x)
    v.right = rl
update(v)
return {v, rr}
else
    ... 
```

```c
merge(Node l, Node r)
if l.prio < r.prio:
    tmp = merge(l.right, r)
    l.right = tmp
update(l)
return l
else
    ... 
```
an important operation on sets is called **order of key**

given a key $k$, it returns the number of elements in the set that are smaller than $k$

How would you support **order of key** in a treap?
What a Treap can do

Self balancing BSTs
(like a set)

- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values

binary search tree
aggregates
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

---

- binary search tree
- aggregates
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

binary search tree
aggregates
implicit treap
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix

splits off the \( k \) leftmost nodes

\[ \text{split-size}(\text{Node } v, \text{ int } k) \]
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

splits off the $k$ leftmost nodes

`split-size(Node v, int k)`
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```
split-size(Node v, int k)
```

splits off the \( k \) leftmost nodes

```
if 1+size(v.left) \leq k:
```

```
else
```

example: split off first 15 nodes

now that we have aggregates; we can split based on aggregates
e.g. split based on size of prefix
example: split off first 15 nodes

```
split-size(Node v, int k)
```

```
if 1+size(v.left) \leq k:
```

```
else
```

example: split off first 15 nodes

now that we have aggregates; we can split based on aggregates
e.g. split based on size of prefix
example: split off first 15 nodes
 Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```cpp
split-size(Node v, int k)

if 1+size(v.left) ≤ k:
    rl, rr = split-size(v.right, k-1-size(v.left))
else
```

splits off the $k$ leftmost nodes
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```
split-size(Node v, int k)
  if 1+size(v.left) ≤ k:
    rl, rr = split-size(v.right, k-1-size(v.left))
    v.right = rl
  else
    splits off the k leftmost nodes
```

---

SIZE

example: split off first 15 nodes

now that we have aggregates; we can split based on aggregates
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```
split-size(Node v, int k)
    if 1+size(v.left) ≤ k:
        rl, rr = split-size(v.right, k-1-size(v.left))
        v.right = rl
        update(v)
    else
        splits off the k leftmost nodes
```

example: split off first 15 nodes

now that we have aggregates; we can split based on aggregates
e.g. split based on size of prefix
e.g. split based on size of prefix

SIZE
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

splits off the $k$ leftmost nodes

```cpp
split-size(Node v, int k)
    if 1+size(v.left) ≤ k:
        rl, rr = split-size(v.right, k-1-size(v.left))
        v.right = rl
        update(v)
        return {v, rr}
    else
```

example: split off first 15 nodes
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```java
split-size(Node v, int k)
    if 1+size(v.left) ≤ k:
        rl, rr = split-size(v.right, k-1-size(v.left))
        v.right = rl
        update(v)
        return {v, rr}
    else
        ll, lr = split-size(v.left, k)
        v.left = lr
        update(v)
        return {ll, v}
```

splits off the $k$ leftmost nodes

example: split off first 15 nodes

SIZE
Implicit Treaps - Split by Size

- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix
- example: split off first 15 nodes

```
split-size(Node v, int k)
if 1 + size(v.left) ≤ k:
    rl, rr = split-size(v.right, k - 1 - size(v.left))
    v.right = rl
    update(v)
    return {v, rr}
else
    ll, lr = split-size(v.left, k)
    v.left = lr
    update(v)
    return {ll, v}
```

splits off the \( k \) leftmost nodes

---

SIZE

Competitive Programming | Algorithm Engineering Group (HPI) & Scalable Algorithms Group (KIT)
Implicit Treaps

this is the point were Treaps really start to lay eggs
Implicit Treaps

**this is the point were Treaps really start to lay eggs**

- let's drop the BST property of our treap
Implicit Treaps

- Let's drop the BST property of our treap.
- Conceptually, we now work like an array instead of a set.

This is the point where Treaps really start to lay eggs.
Implicit Treaps

- lets drop the BST property of our treap
- conceptually we now work like an array instead of a set
- we replace \texttt{split} with \texttt{split-size}
  
  e.g. give me the first 5 nodes

\textbf{this is the point were Treaps really start to lay eggs}
Implicit Treaps

- lets drop the BST property of our treap
- conceptually we now work like an array instead of a set
- we replace `split` with `split-size`
  e.g. give me the first 5 nodes
- positions of nodes are now implicit

this is the point were Treaps really start to lay eggs
Implicit Treaps

- lets drop the BST property of our treap
- conceptually we now work like an array instead of a set
- we replace `split` with `split-size`
  - e.g. give me the first 5 nodes
- positions of nodes are now implicit
- merge will preserve order

```
this is the point were Treaps really start to lay eggs
```
Implicit Treaps

- Let's drop the BST property of our treap
- Conceptually we now work like an array instead of a set
- We replace `split` with `split-size`
  
  *e.g. give me the first 5 nodes*

- Positions of nodes are now implicit
- Merge will preserve order

---

**This is the point where Treaps really start to lay eggs**
Implicit Treaps

- lets drop the BST property of our treap
- conceptually we now work like an array instead of a set
- we replace `split` with `split-size`
  
  e.g. give me the first 5 nodes
- positions of nodes are now implicit
- merge will preserve order

this is the point were Treaps really start to lay eggs

with great power comes great responsibility
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

- binary search tree
- aggregates
- implicit treap
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

**Implicit Treaps**
(like a really cool array)
- split by size
- access at index
- insert after index
- delete at index
- get range
- delete range
- query range for aggregates

**Binary search tree**
-aggregates
-implicit treap
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

**Implicit Treaps**
(like a really cool array)
- split by size
- access at index
- insert after index
- delete at index
- get range
- delete range
- query range for aggregates
- move range

binary search tree
aggregates
implicit treap
What a Treap can do

**Self balancing BSTs**  
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

**Implicit Treaps**  
(like a really cool array)
- split by size
- access at index
- insert after index
- delete at index
- get range
- delete range
- query range for aggregates
- move range

- binary search tree
- aggregates
- implicit treap
- lazy propagation
Treaps + Lazy Propagation

- same concept as for segment trees
Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children
Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children
- update range by splitting it out and apply change to root
Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children
- update range by splitting it out and apply change to root
- cool application: reverse order of range
Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children
- update range by splitting it out and apply change to root
- cool application: reverse order of range
  
  reverse order ≡ for all nodes swap left and right child
Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children
- update range by splitting it out and apply change to root
- cool application: reverse order of range

reverse order ≡ for all nodes swap left and right child

```python
push(Node v)
if v.reversed:
    swap(v.left, v.right)
    v.reversed = False
if v.left:
    v.left.reversed ^= 1
if v.right:
    v.right.reversed ^= 1
```

reverse order of range 

```plaintext
[l, r] < l > r
```
Treaps + Lazy Propagation

- lazy propagation again is a one-line change
- don’t forget to add it to split-size

\[
\text{split} (\text{Node } v, \text{Key } x) \\
\text{if } v\.\text{key} < x:\n\begin{align*}
&\text{push} (v) \\
&rl, rr = \text{split} (v\.\text{right}, x) \\
&v\.\text{right} = rl \\
&\text{update} (v) \\
&\text{return} \{v, rr\} \\
\text{else} &\ldots
\end{align*}
\]

\[
\text{merge} (\text{Node } l, \text{Node } r) \\
\text{if } l\.\text{prio} < r\.\text{prio}: \\
\begin{align*}
&\text{push} (l) \\
&tmp = \text{merge} (l\.\text{right}, r) \\
&l\.\text{right} = tmp \\
&\text{update} (l) \\
&\text{return} l \\
\text{else} &\ldots
\end{align*}
\]

lazy propagation again is a one-line change
don’t forget to add it to split-size
Treaps + Lazy Propagation

- lazy propagation again is a one-line change
- don’t forget to add it to split-size

### split(Node v, Key x)

```python
if v.key < x:
    push(v)
    rl, rr = split(v.right, x)
    v.right = rl
    update(v)
    return {v, rr}
else
    ...
```

### merge(Node l, Node r)

```python
if l.prio < r.prio:
    push(l)
    tmp = merge(l.right, r)
    l.right = tmp
    update(l)
    return l
else
    ...
```
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

**Implicit Treaps**
(like a really cool array)
- split by size
- access at index
- insert after index
- delete at index
- get range
- delete range
- query range for aggregates
- move range

**Binary search tree**
-aggregates
-implicit treap
-lazy propagation
What a Treap can do

**Self balancing BSTs**
(like a set)
- split by value
- find value
- insert value
- delete value
- find all values in interval
- delete interval of values
- query interval for aggregates
- order of key

**Implicit Treaps**
(like a really cool array)
- split by size
- access at index
- insert after index
- delete at index
- get range
- delete range
- query range for aggregates
- move range
- update range (e.g. reverse)

- binary search tree
- aggregates
- implicit treap
- lazy propagation
Implementation Details

- represent a treap by a pointer to its root node
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are nullptr
- requires some nullptr checks before actual **split** and **merge** methods
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods

```cpp
if(!v) return {nullptr, nullptr};
```
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are nullptr
- requires some nullptr checks before actual split and merge methods

```c
split
if(!v) return {nullptr, nullptr};
```

```c
merge
if(!l) return r;
if(!r) return l;
```
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are nullptr
- requires some nullptr checks before actual split and merge methods
- push and update assume v is not nullptr

split
if(!v) return {nullptr, nullptr};

merge
if(!l) return r;
if(!r) return l;
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children
  (already done in pseudocode)

```
split  
if(!v) return {nullptr, nullptr};

merge  
if(!l) return r;
if(!r) return l;
```
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children (already done in pseudocode)
- `update` reads aggregates from children with free functions (already done in pseudocode)

```cpp
auto size(Treap* t) { return t ? t->size : 0; }

split
if(!v) return {nullptr, nullptr};

merge
if(!l) return r;
if(!r) return l;
```

(already done in pseudocode)
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children
  (already done in pseudocode)
- `update` reads aggregates from children with free functions
  (already done in pseudocode)
- treap struct should have no methods except constructor

```cpp
auto size(Treap* t) { return t ? t->size : 0; }
```

```cpp
split
if(!v) return {nullptr, nullptr};
```

```cpp
merge
if(!l) return r;
if(!r) return l;
```

(already done in pseudocode)
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children
  (already done in pseudocode)
- `update` reads aggregates from children with free functions
  (already done in pseudocode)
- treap struct should have no methods except constructor
- after all the split and merge, don’t forget to assign back to your root handle

```cpp
auto size(Treap* t) { return t ? t->size : 0; }

if(!v) return {nullptr, nullptr};

if(!l) return r;
if(!r) return l;

if(!l) return r;
if(!r) return l;

(already done in pseudocode)
```

```cpp
split
```
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children
  (already done in pseudocode)
- `update` reads aggregates from children with free functions
  (already done in pseudocode)
- treap struct should have no methods except constructor
- after all the split and merge, don’t forget to assign back to your root handle

```cpp
auto size(Treap* t) { return t ? t->size : 0; }
```

- it’s ok to use `rand()`
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are nullptr
- requires some nullptr checks before actual split and merge methods
- push and update assume v is not nullptr
- push has to check children
  (already done in pseudocode)
- update reads aggregates from children with free functions
  (already done in pseudocode)
- treap struct should have no methods except constructor
- after all the split and merge, don’t forget to assign back to your root handle

split
if(!v) return {nullptr, nullptr};

merge
if(!l) return r;
if(!r) return l;

(auto size(Treap* t) { return t ? t->size : 0; })

- it’s ok to use rand()
- create nodes with new
  root = merge(root, new Treap(v));
Implementation Details

- represent a treap by a pointer to its root node
- use pointer for children; empty treaps are `nullptr`
- requires some `nullptr` checks before actual `split` and `merge` methods
- `push` and `update` assume `v` is not `nullptr`
- `push` has to check children
  (already done in pseudocode)
- `update` reads aggregates from children with free functions
  (already done in pseudocode)
- treap struct should have no methods except constructor
- after all the split and merge, don’t forget to assign back to your root handle

```cpp
auto size(Treap* t) { return t ? t->size : 0; }

if(!v) return {nullptr, nullptr};

if(!l) return r;
if(!r) return l;
merge

split

update reads aggregates from children with free functions
(already done in pseudocode)

it’s ok to use `rand()`

create nodes with `new`

root = merge(root, new Treap(v));

ignore memory leaks
```
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge
Implicit priorities: instead of storing, toss a coin every merge

```cpp
merge(Node l, Node r)
if l.prio < r.prio:
    ...
else
    ...
```

```cpp
merge(Node l, Node r)
if rng() % (l.size + r.size) < l.size:
    ...
else
    ...
```

mt19937 rng(static_cast<unsigned>(chrono::steady_clock::now().time_since_epoch().count()));
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge

\[
\text{merge}(\text{Node } l, \text{Node } r)
\]

\[
\begin{align*}
\text{if } & l.\text{prio} < r.\text{prio}: \\
& \text{...} \\
\text{else} \\
& \text{...}
\end{align*}
\]

\[
\begin{align*}
\text{merge}(\text{Node } l, \text{Node } r)
\end{align*}
\]

\[
\begin{align*}
\text{if } & \text{rng()} \% (l.\text{size} + r.\text{size}) < l.\text{size}: \\
& \text{...} \\
\text{else} \\
& \text{...}
\end{align*}
\]

\[\text{mt19937 \ rng(static\_cast<unsigned>(\text{chrono::steady\_clock::now().time\_since\_epoch().count()}));}\]

could reduce depth up to x8!
also, one less field to store
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge

\[
\text{merge}(\text{Node } l, \text{Node } r)
\]

\[
\begin{align*}
\text{if } l.\text{prio} &< r.\text{prio}: \\
&\ldots \\
\text{else} &\\
&\ldots
\end{align*}
\]

- \(O(n)\) construction

\[
\text{mt19937 } \text{rng(static\_cast<unsigned>(\text{chrono::steady\_clock::now().time\_since\_epoch().count()}));}
\]

sometimes helpful if size >> no. of queries, but not too often

see, e.g., https://cp-algorithms.com/data_structures/treap.html

could reduce depth up to x8!
also, one less field to store
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge

```cpp
merge(Node l, Node r)
if l.prio < r.prio:
  ...
else
  ...
```

- \(O(n)\) construction
  - see, e.g., https://cp-algorithms.com/data_structures/treap.html
  - sometimes helpful if size >> no. of queries, but not too often

- ints vs raw pointers
  - store all nodes in a global `vector<Node> v`, pass index `i`, access by `v[i]`

```cpp
mt19937 rng(static_cast<unsigned>(chrono::steady_clock::now().time_since_epoch().count()));
mtrie::trie<
case a
```
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge

```
merge(Node l, Node r)

if l.prio < r.prio:
    ...
else
    ...
```

- O(n) construction

```
merge(Node l, Node r)

if rng() % (l.size + r.size) < l.size:
    ...
else
    ...
```

could reduce depth up to x8!
also, one less field to store

- pros: Node stores two 32-bit ints instead of two 64-bit pointers

store all nodes in a global vector<Node> v, pass index i, access by v[i]

see, e.g., https://cp-algorithms.com/data_structures/treap.html
sometimes helpful if size >> no. of queries, but not too often

- ints vs raw pointers
Advanced Optimizations

- Implicit priorities: instead of storing, toss a coin every merge

```
def merge(Node l, Node r):
    if l.prio < r.prio:
        ...
    else:
        ...
```

- **O(n)** construction

  see, e.g., [https://cp-algorithms.com/data_structures/treap.html](https://cp-algorithms.com/data_structures/treap.html)

  sometimes helpful if size >> no. of queries, but not too often

- ints vs raw pointers

  store all nodes in a global `vector<Node> v`, pass index `i`, access by `v[i]`

  pros: Node stores two 32-bit ints instead of two 64-bit pointers

  cons: ugly

```
mt19937 rng(static_cast<unsigned>(chrono::steady_clock::now().time_since_epoch().count()));
```

could reduce depth up to x8!
also, one less field to store
It’s dangerous to go alone! Take this.