Karlsruhe Institute of Technology

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Competitive Programming

Winter Term 23/24

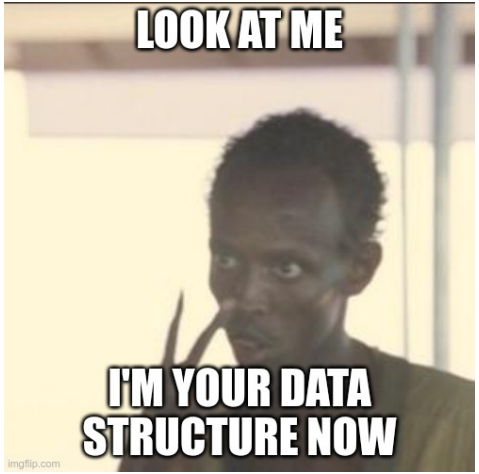
Treaps: the only data structure you'll ever need

Kirill Simonov
Algorithm Engineering Group
HPI

Christopher Weyand
Scalable Algorithms Group
KIT

Overview for Today

- Treaps
- ...
- profit



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

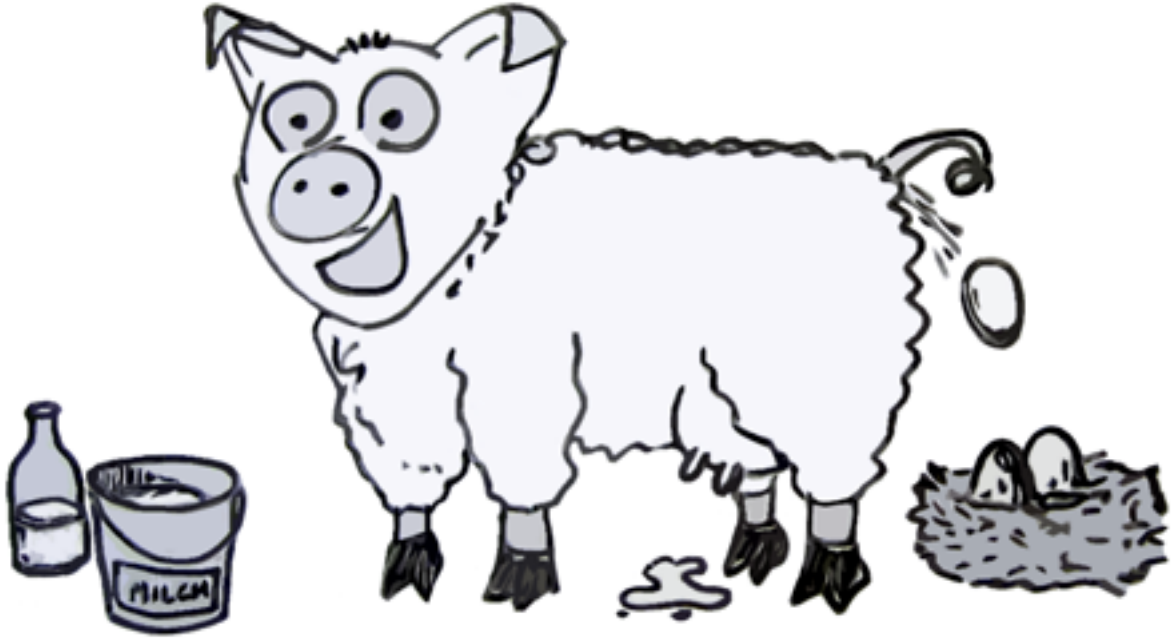
”Would recommend.”

— 9/10 Dentists

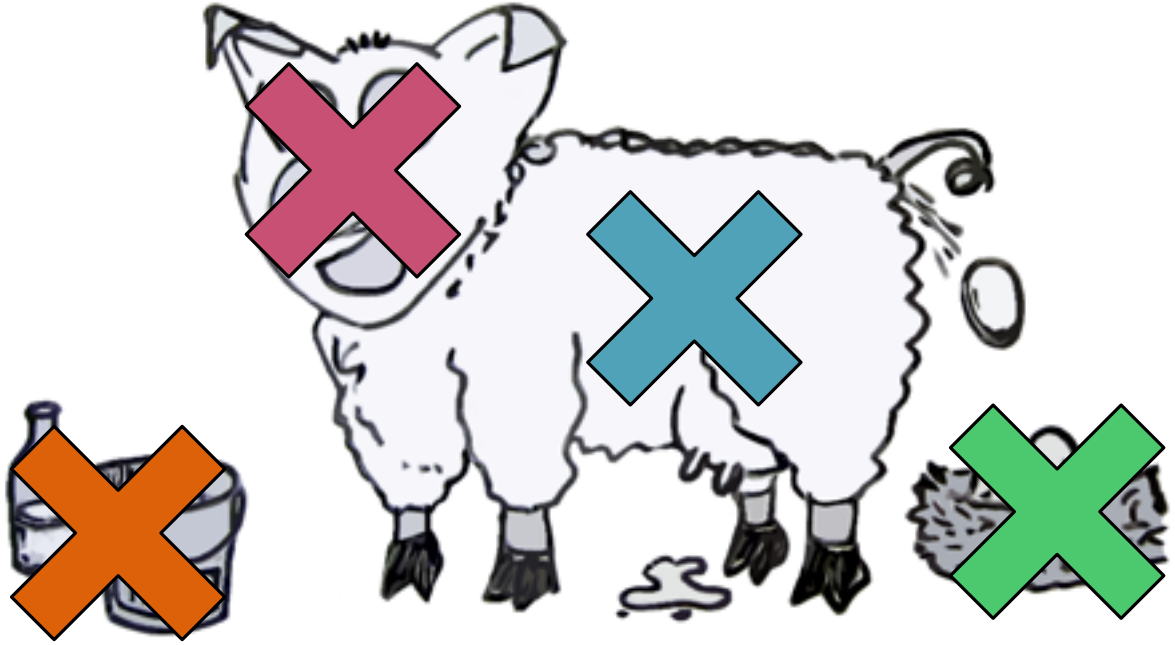
”First implementation was broken, but worked fine after refund.”

— Me

Jack of all Trades



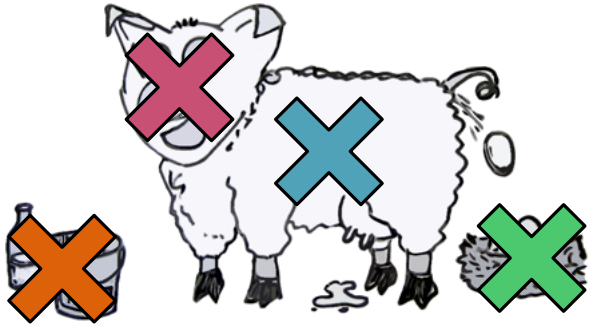
Jack of all Trades



What a Treap can do

Self balancing BSTs
(like a set)

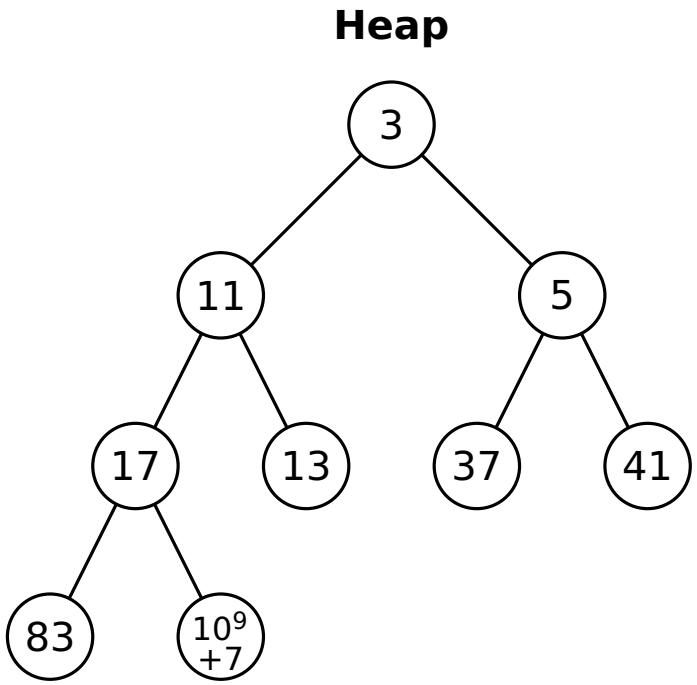
- find value
- insert value
- delete value



binary search tree

Treap = BS**T** + Heap

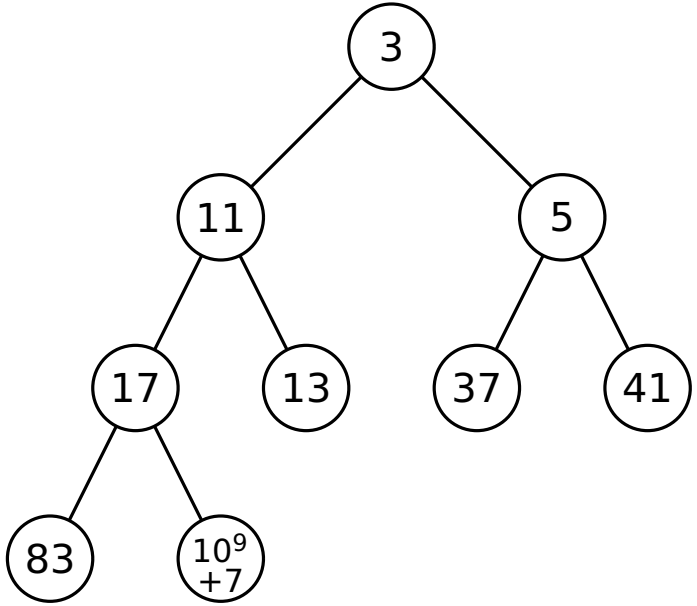
Treap = BS**T** + Heap



parent smaller than children

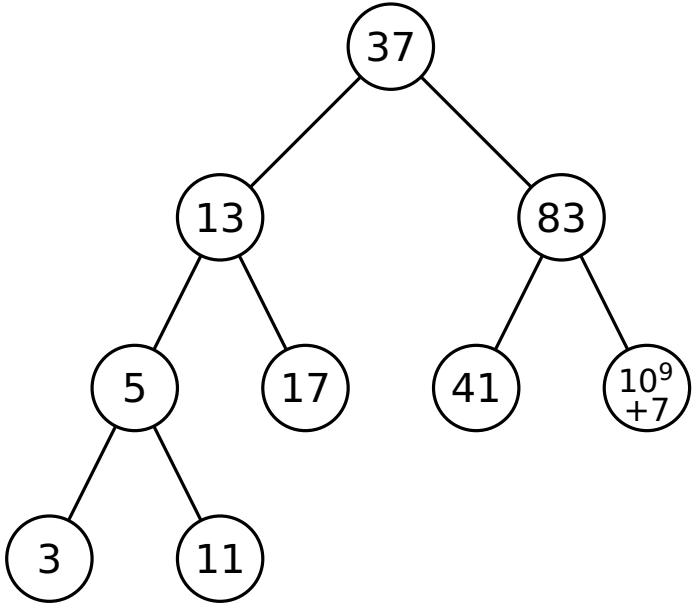
Treap = BS**T** + Heap

Heap



parent smaller than children

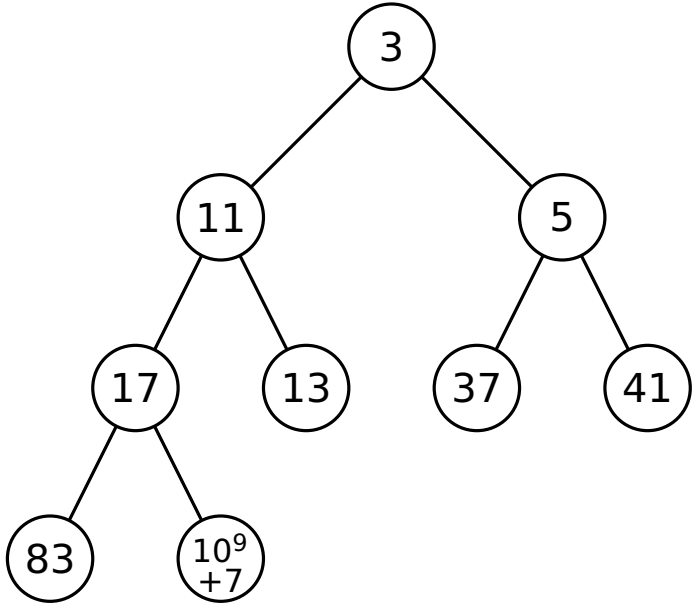
BST



left child $<$ parent $<$ right child

Treap = BS**T** + Heap

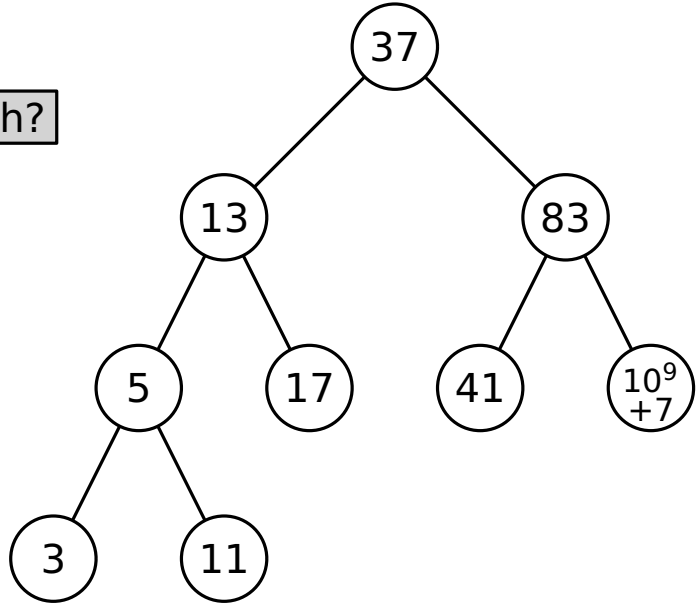
Heap



parent smaller than children

Can we do both?

BST



left child $<$ parent $<$ right child

Treap = BS**T** + Heap

- yes, we can do both, but ...

Treap = BS**T** + Heap

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

Treap = BS**T** + Heap

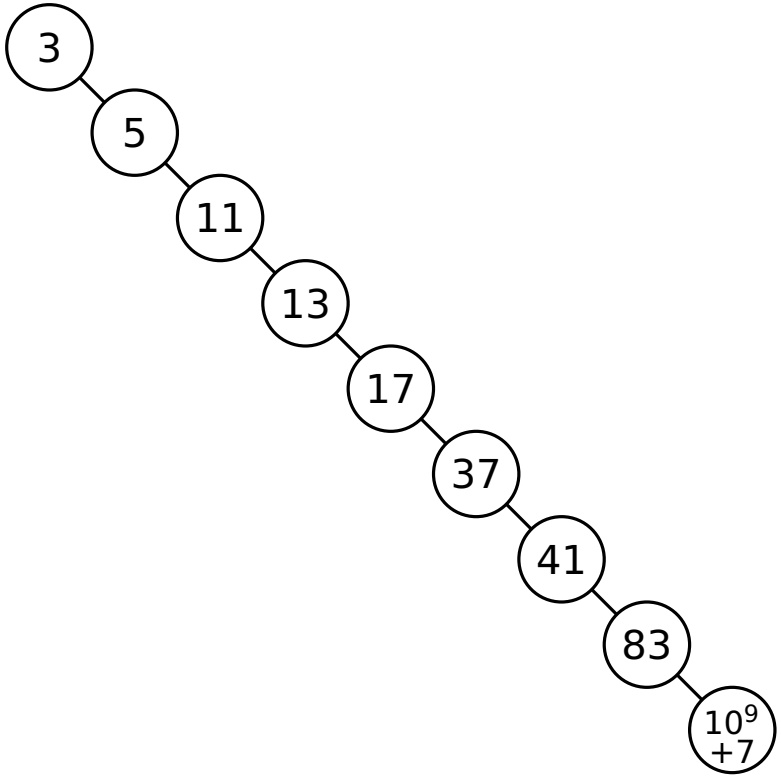
- yes, we can do both, but ...
- parent must be smaller than children (heap)
- parent must be larger than left child (BST)

  $\rightarrow$ there cannot be a left child

Treap = BS**T** + Heap

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

  $\rightarrow$ there cannot be a left child

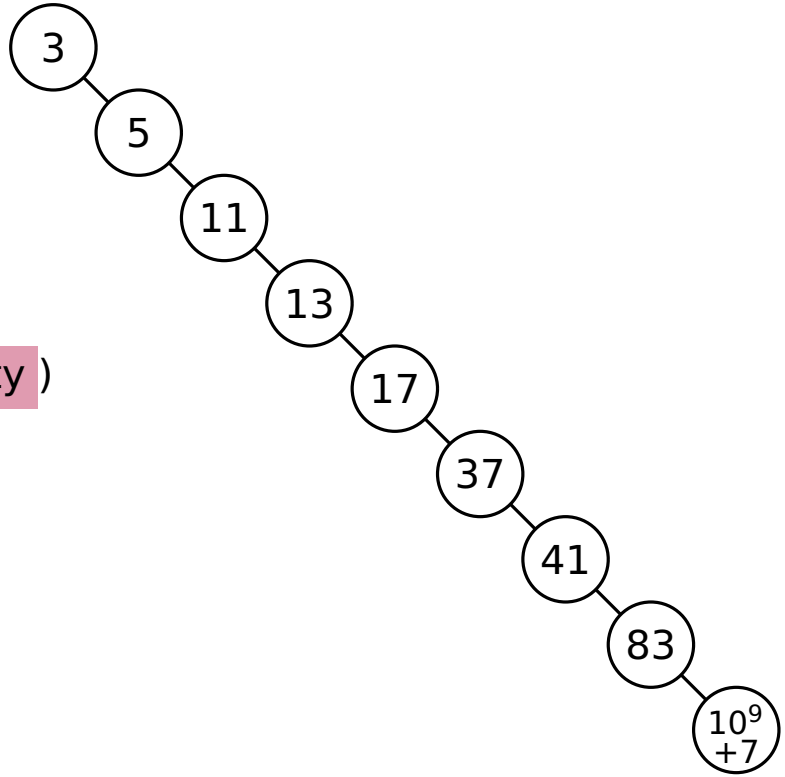


Treap = BS**T** + Heap

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

  $\rightarrow$ there cannot be a left child

- let's give nodes a random second value (Priority)
- heap by priority; BST by key

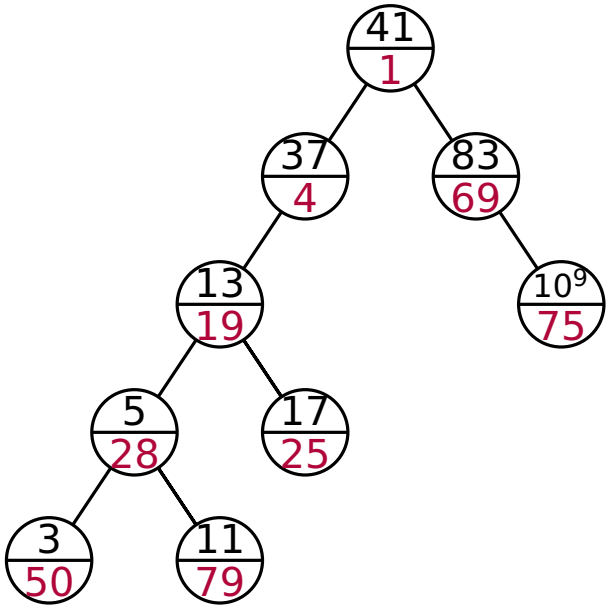


Treap = BS**T** + Heap

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

  $\rightarrow$ there cannot be a left child

- let's give nodes a random second value (Priority)
- heap by priority; BST by key



Treap = BS**T** + Heap

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

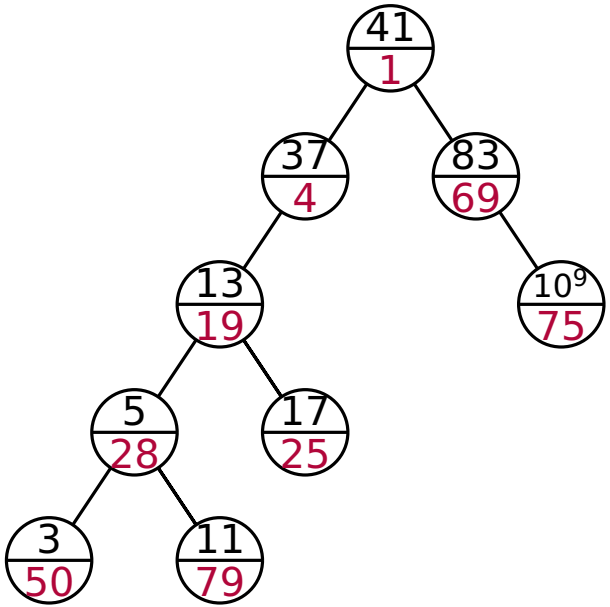
  $\rightarrow$ there cannot be a left child

- let's give nodes a random second value (Priority)
- heap by priority; BST by key

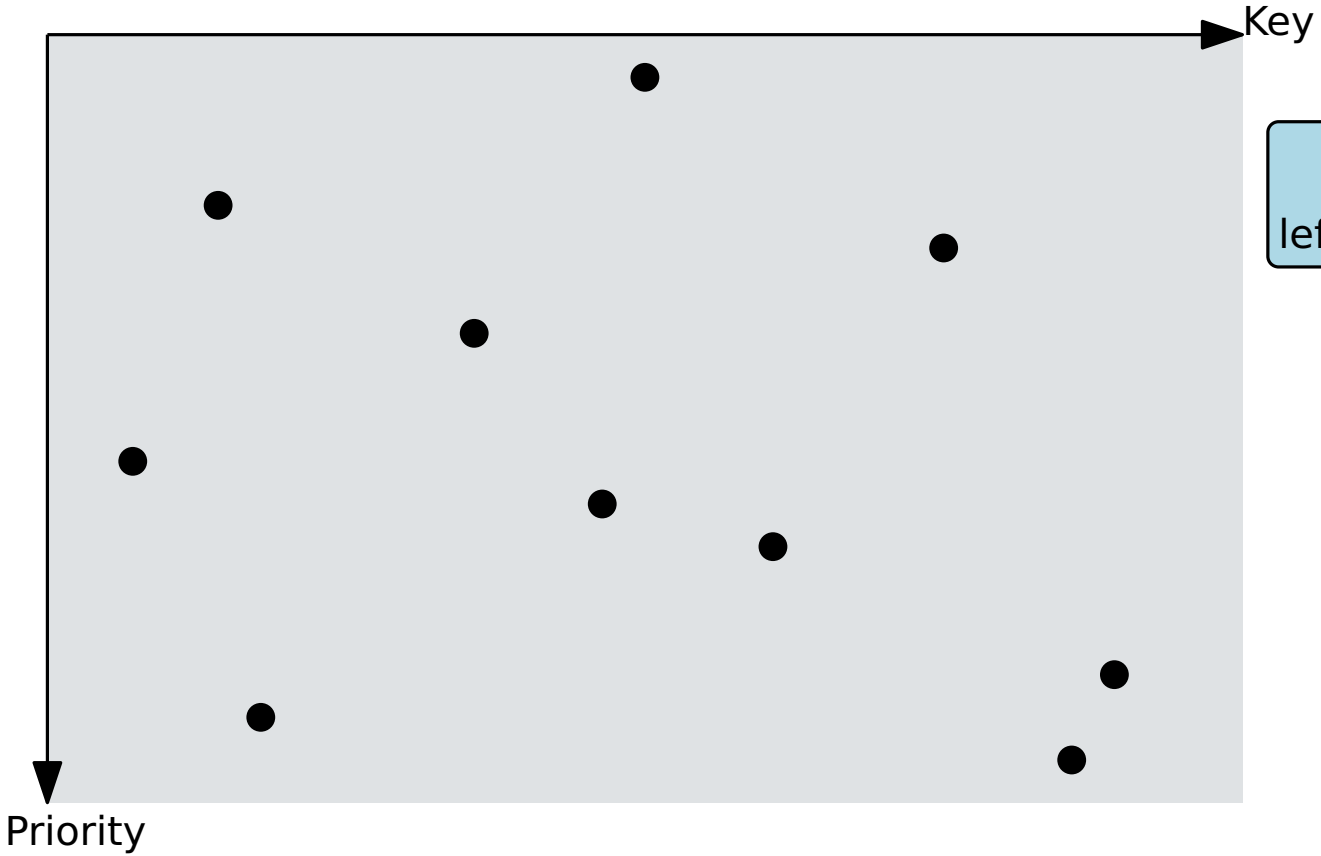
can be shown that:

- 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?



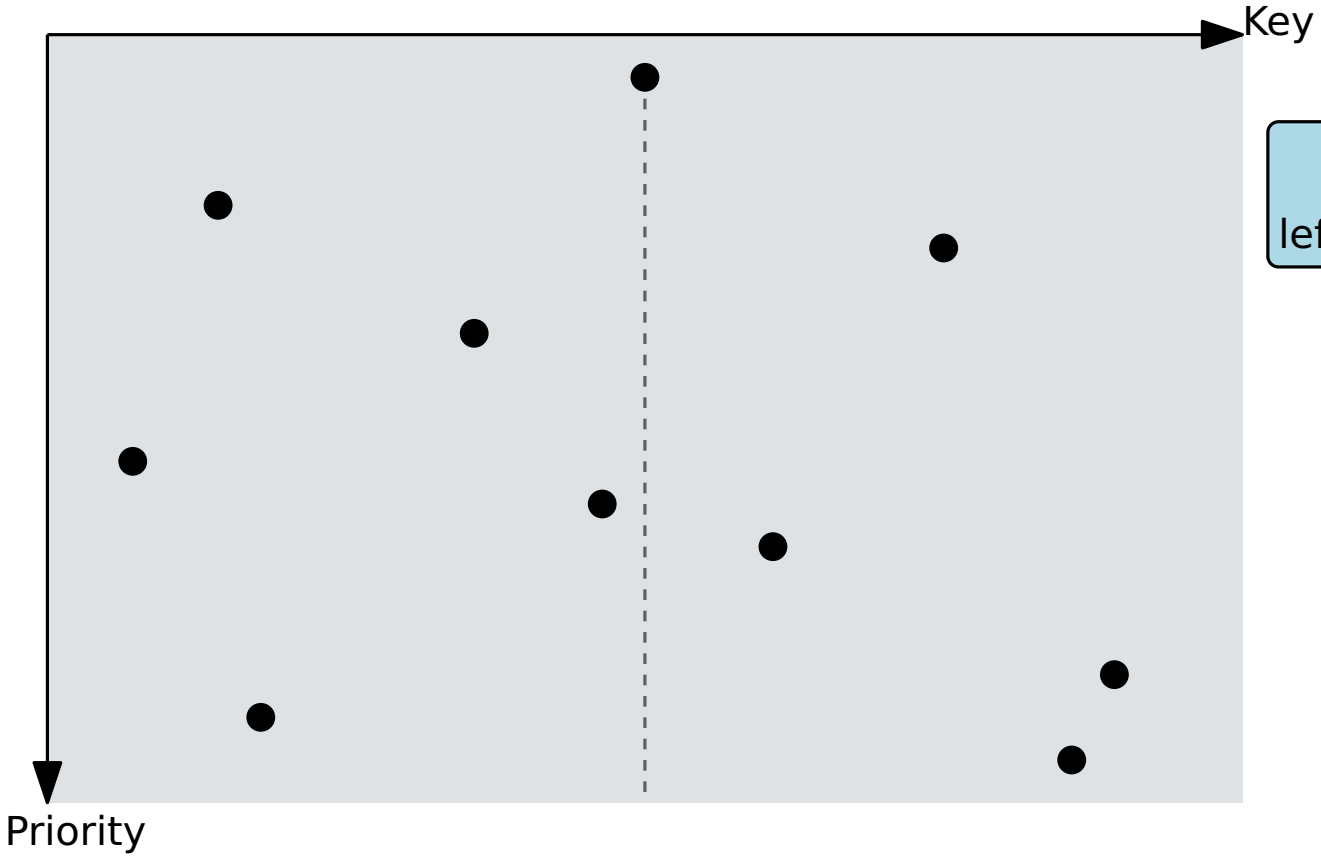
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



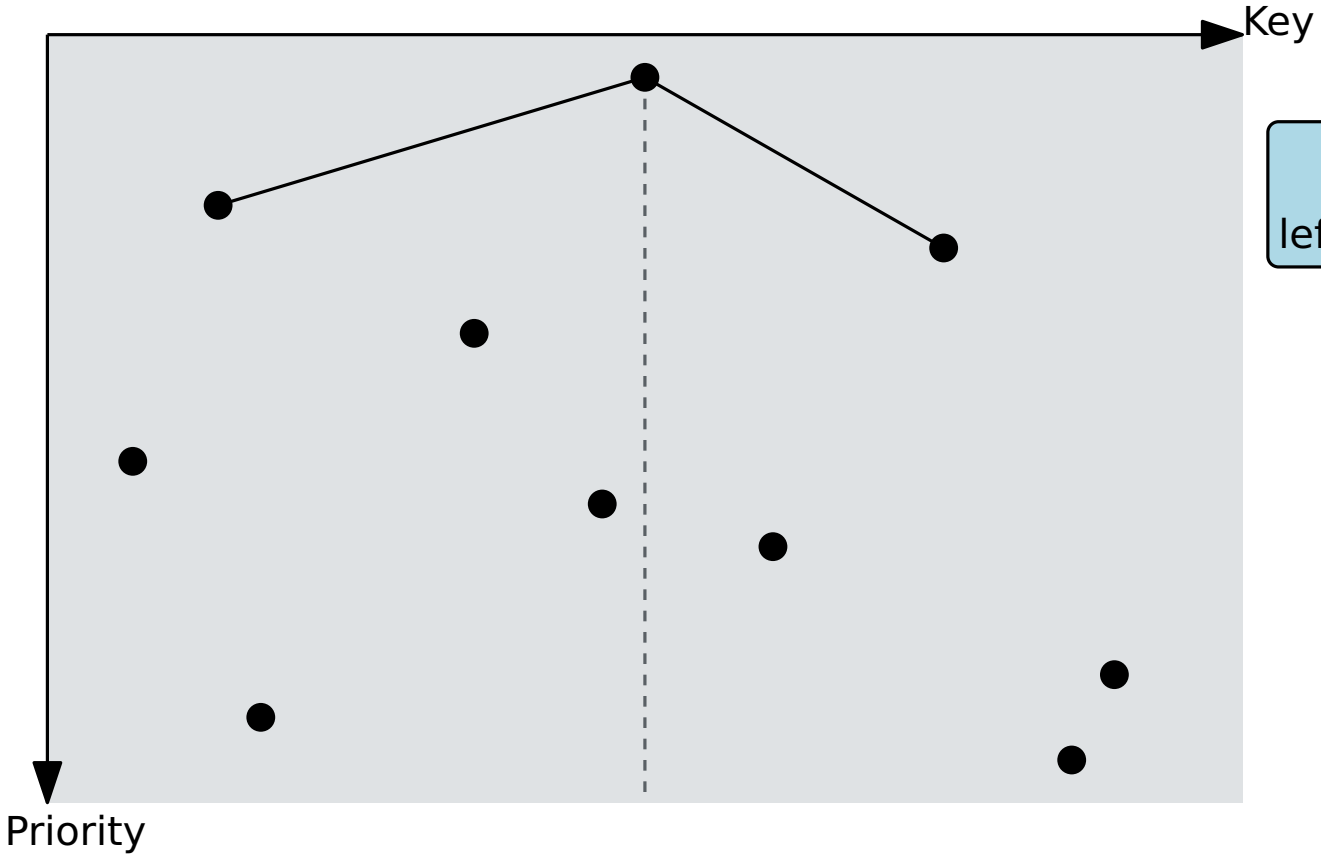
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



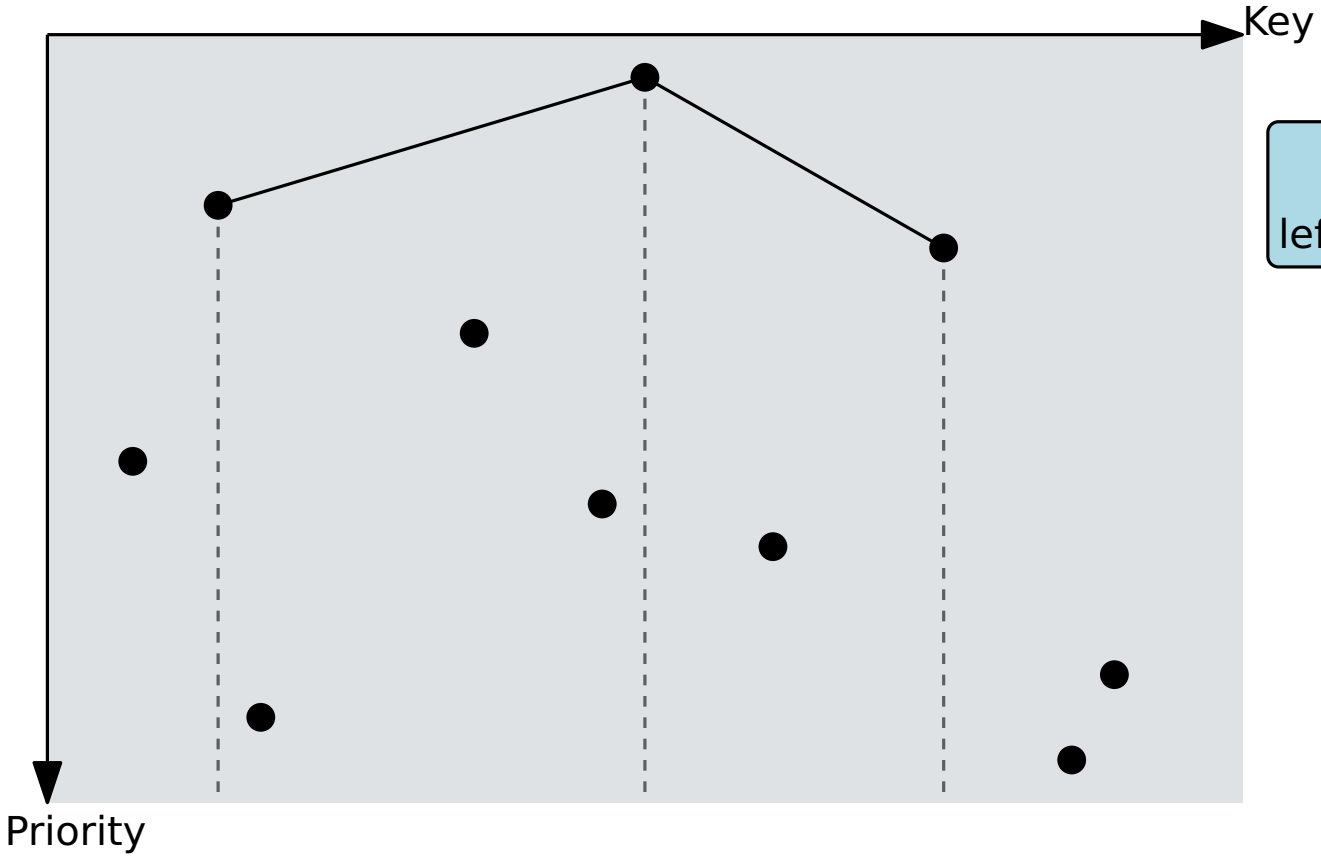
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



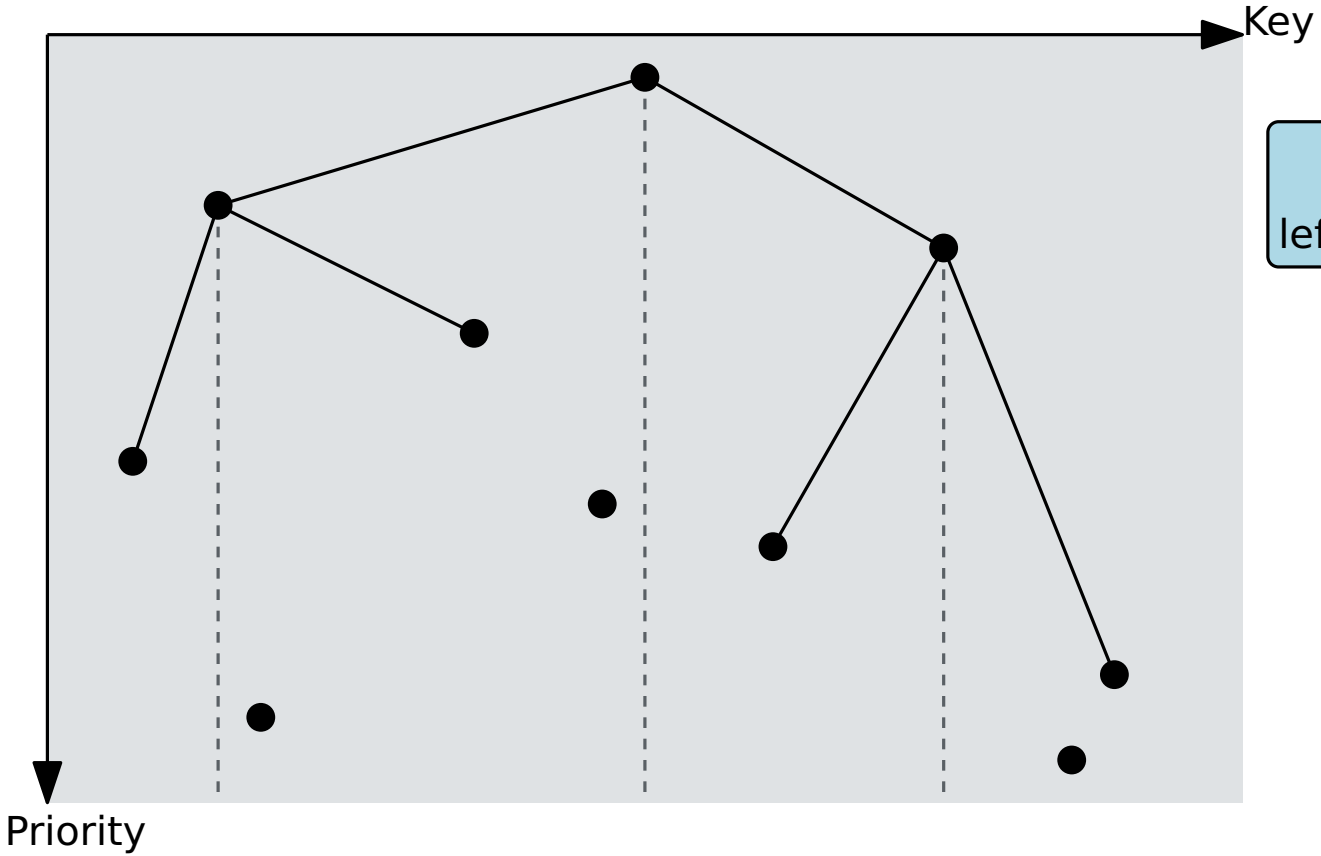
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



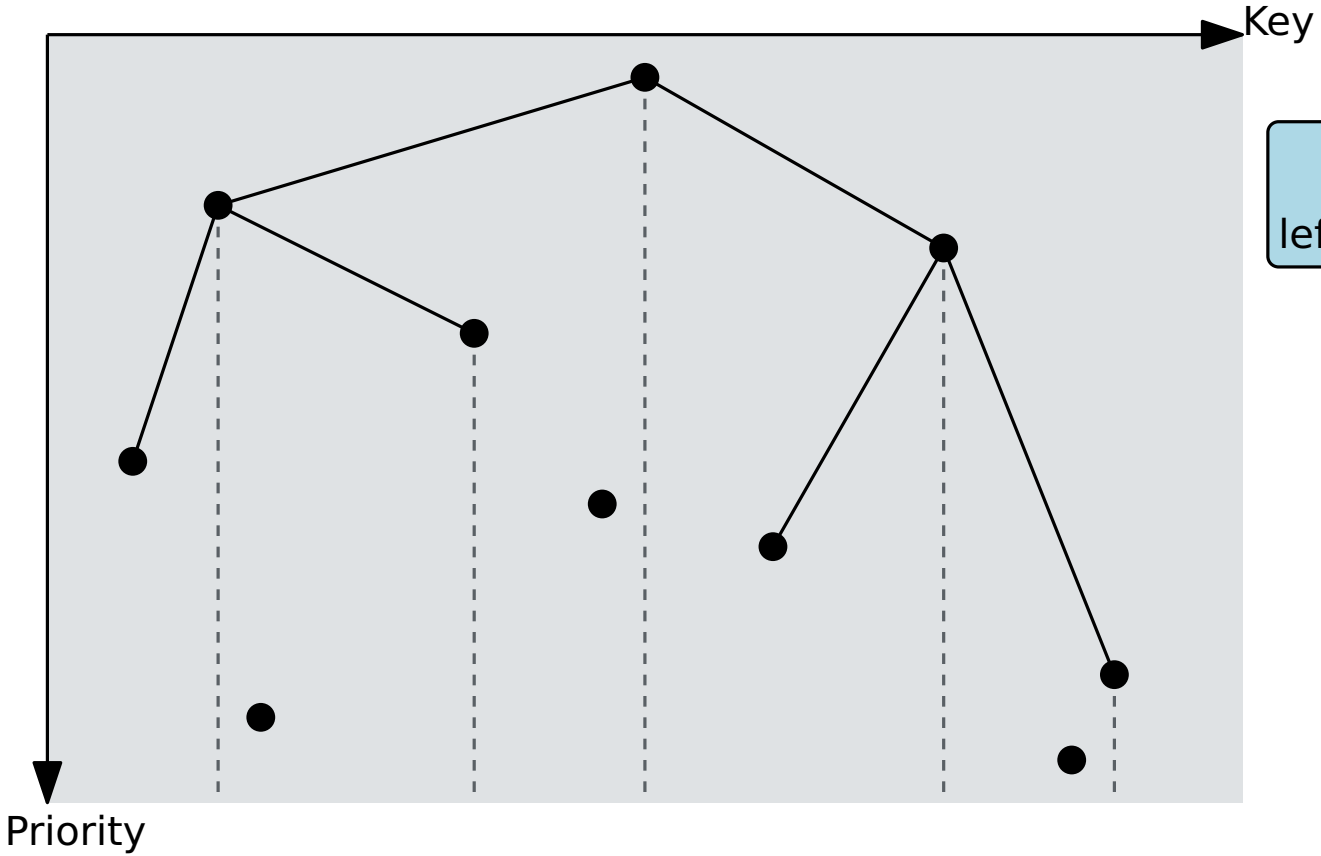
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



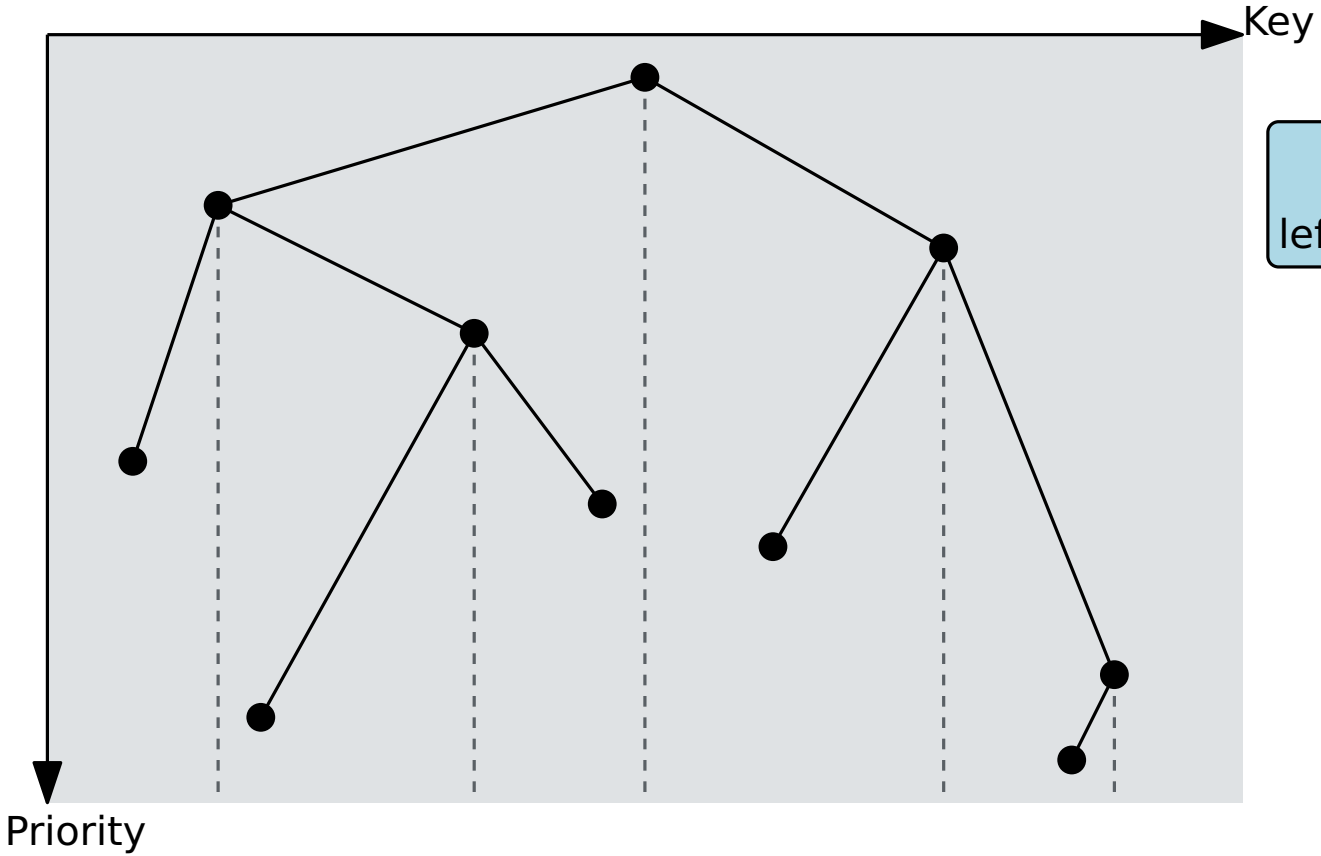
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



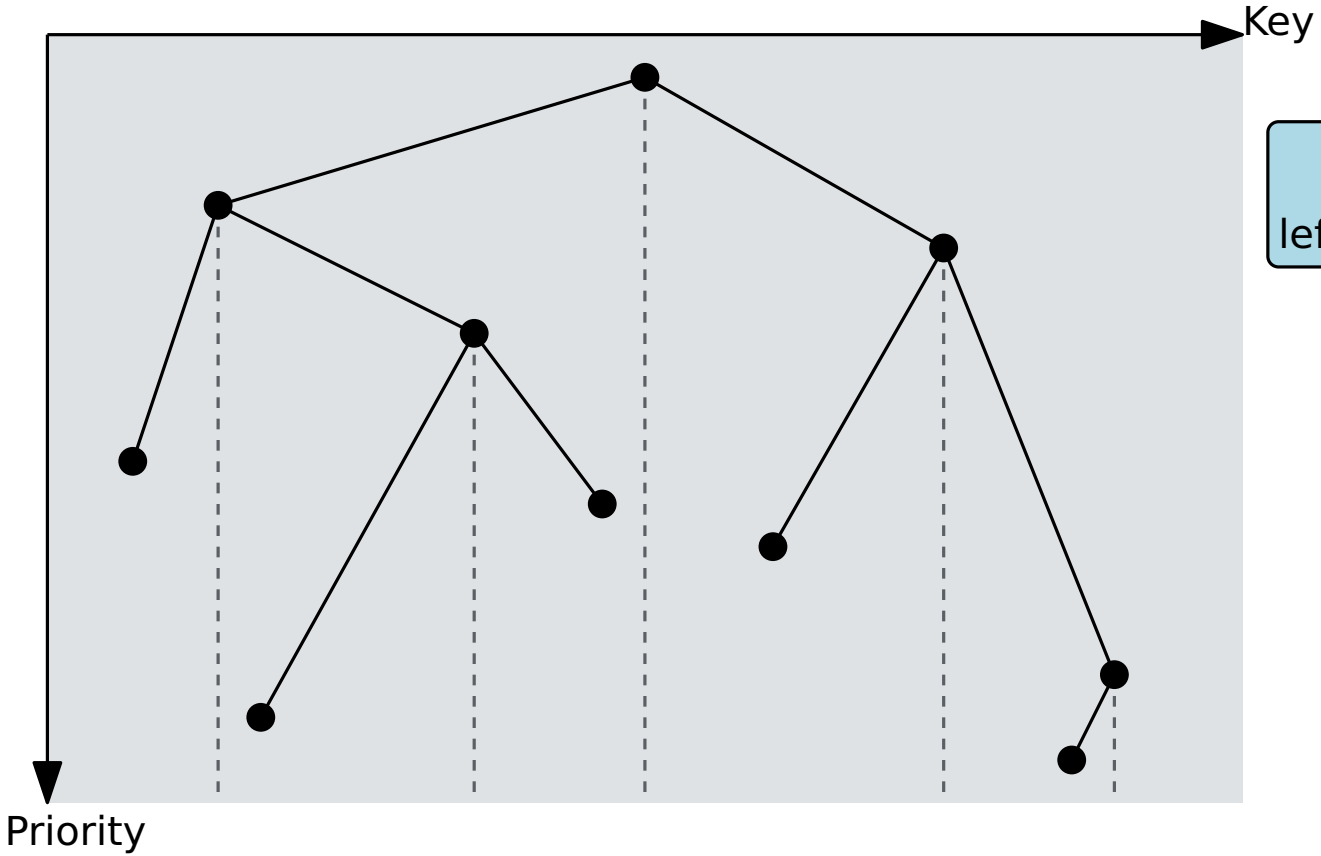
BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

Unique?



BST by key:

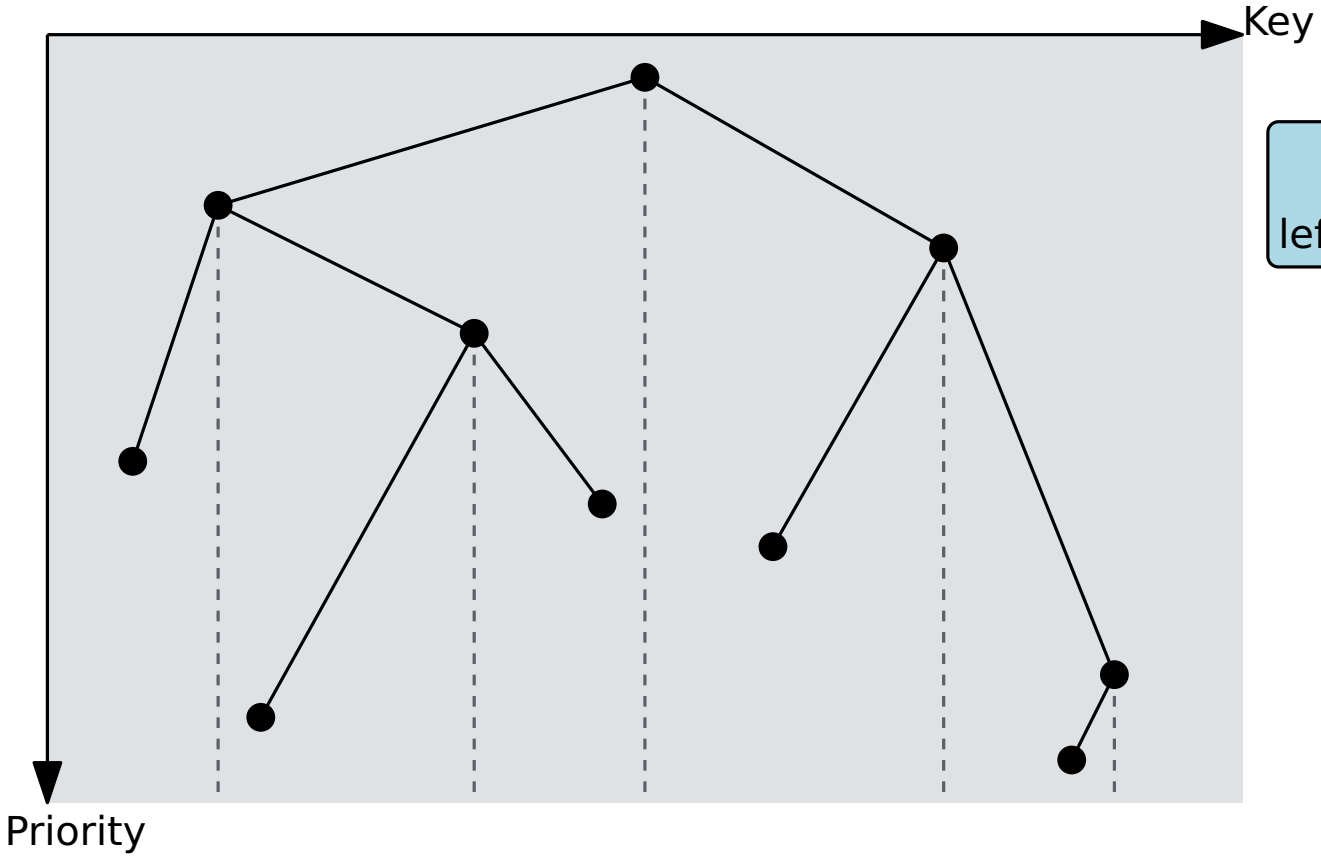
left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

- for now assume unique values

Unique?



BST by key:

left child $<$ parent $<$ right child

Heap by Prio:

parent $<$ children

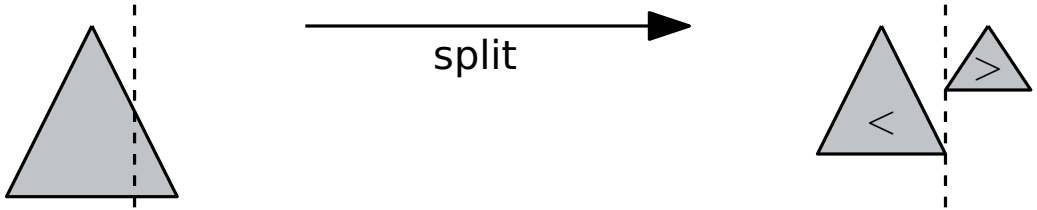
- for now assume unique values
- keys/prios uniquely determine a treap

Implementation

we break down everything to two operations

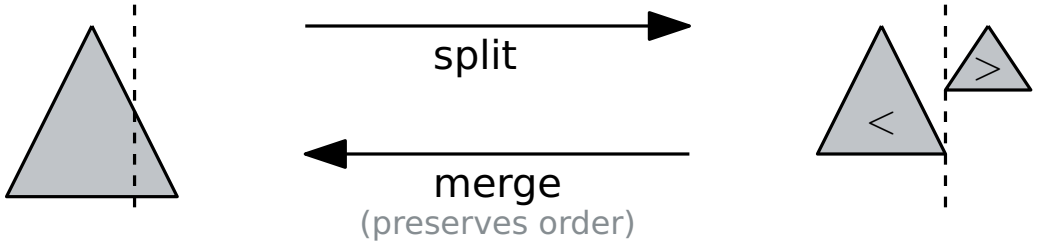
Implementation

we break down everything to two operations



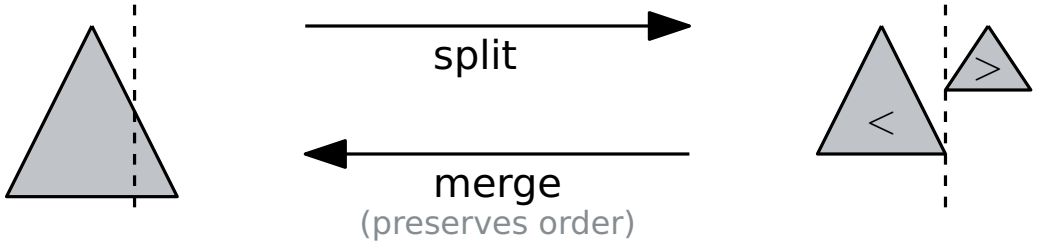
Implementation

we break down everything to two operations

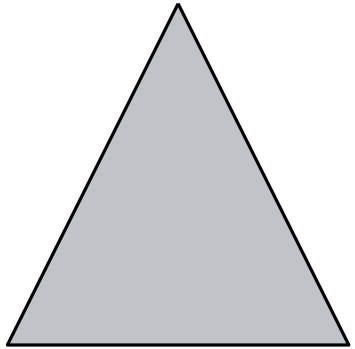


Implementation

we break down everything to two operations

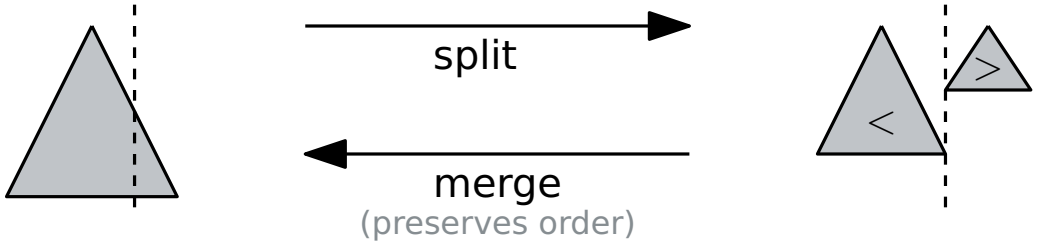


E.g. **delete** x:

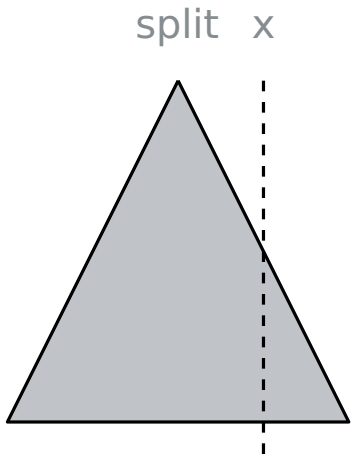


Implementation

we break down everything to two operations

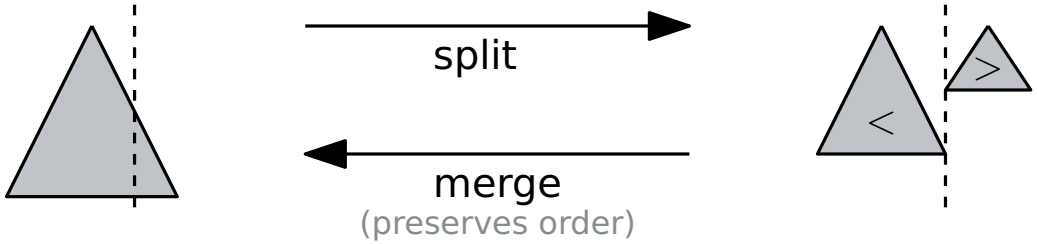


E.g. **delete** x:

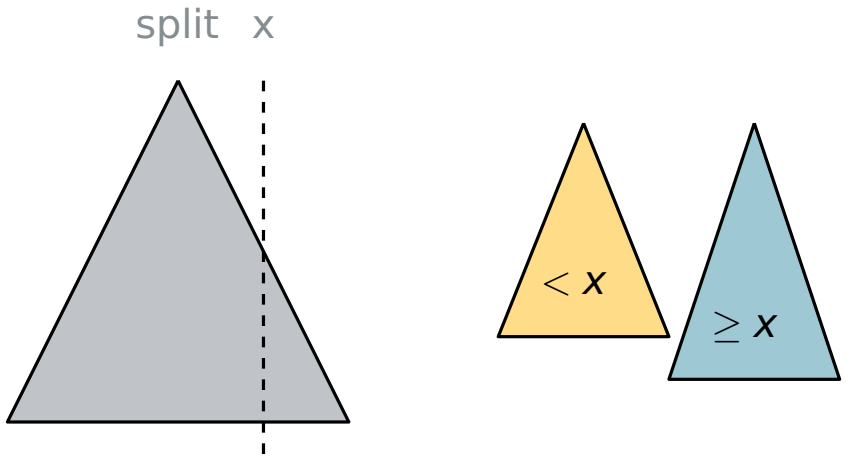


Implementation

we break down everything to two operations

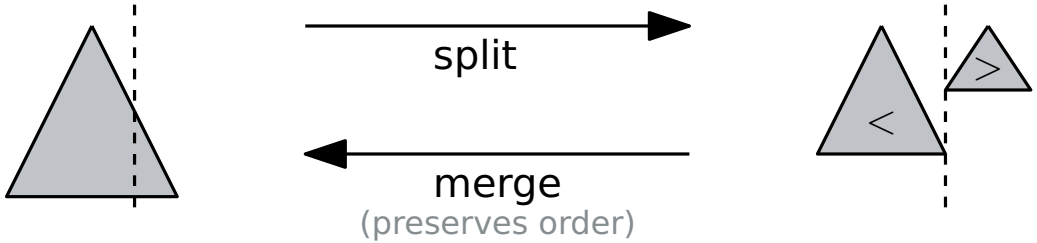


E.g. **delete** x:

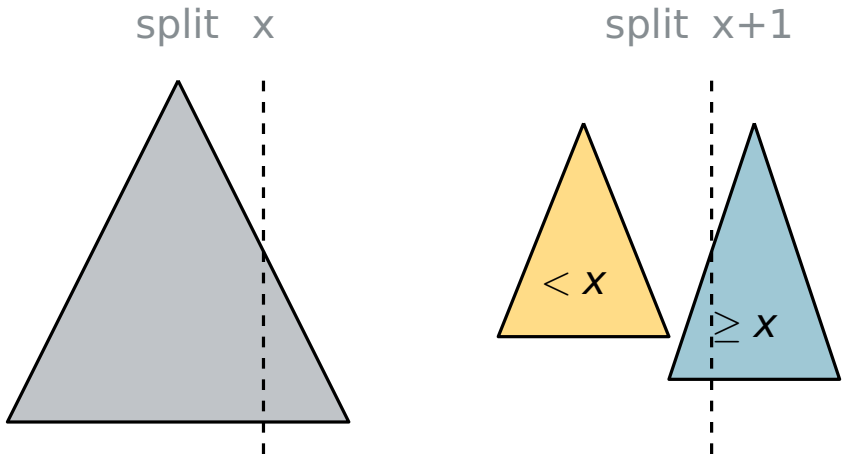


Implementation

we break down everything to two operations

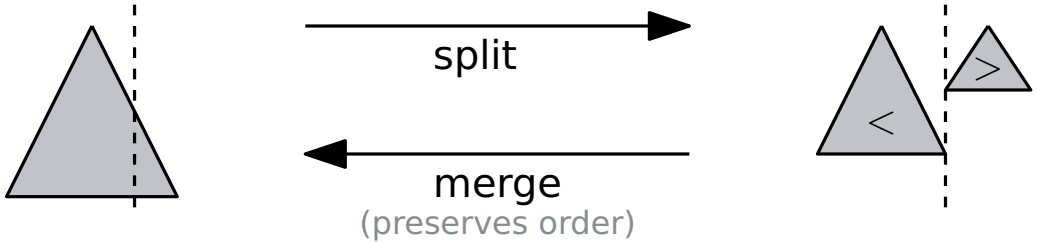


E.g. **delete** x:

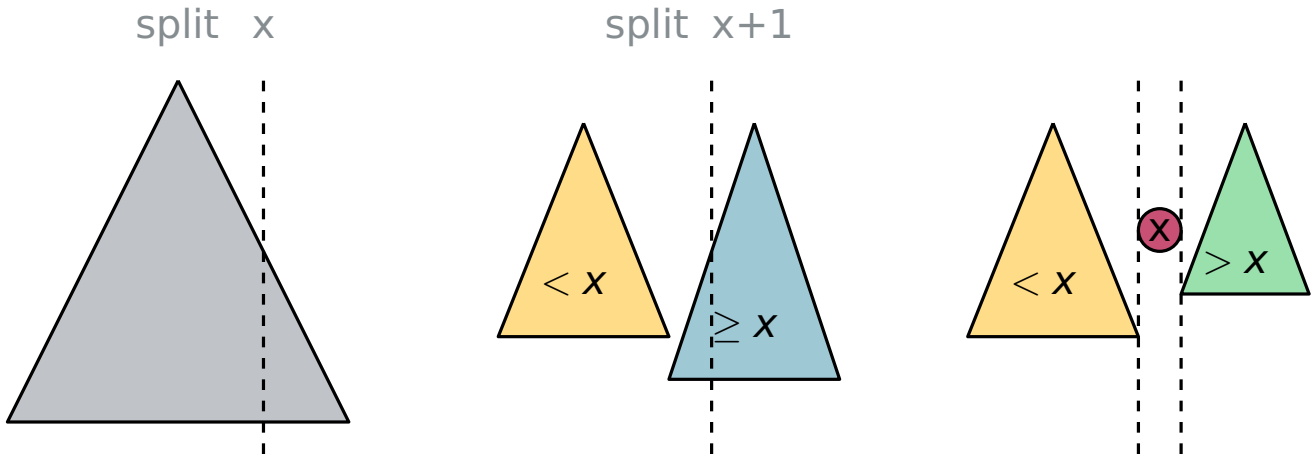


Implementation

we break down everything to two operations

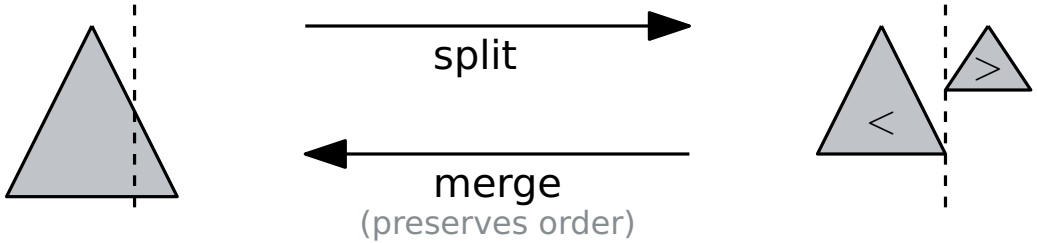


E.g. **delete** x:

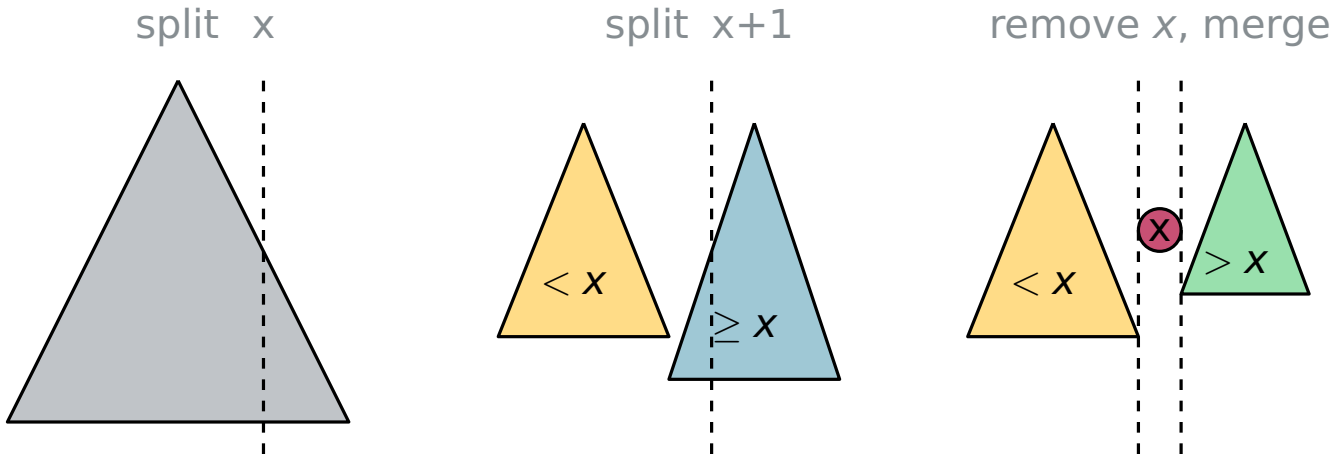


Implementation

we break down everything to two operations

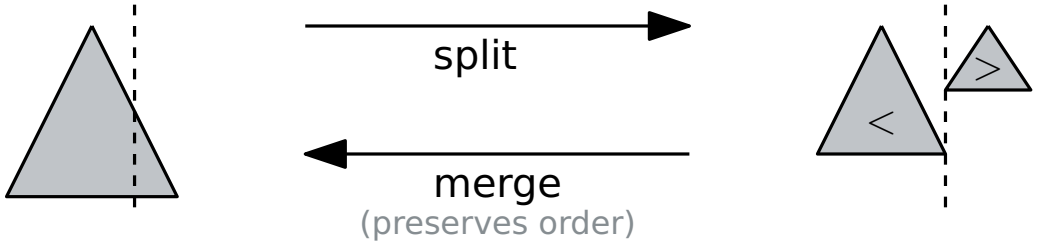


E.g. **delete** x:

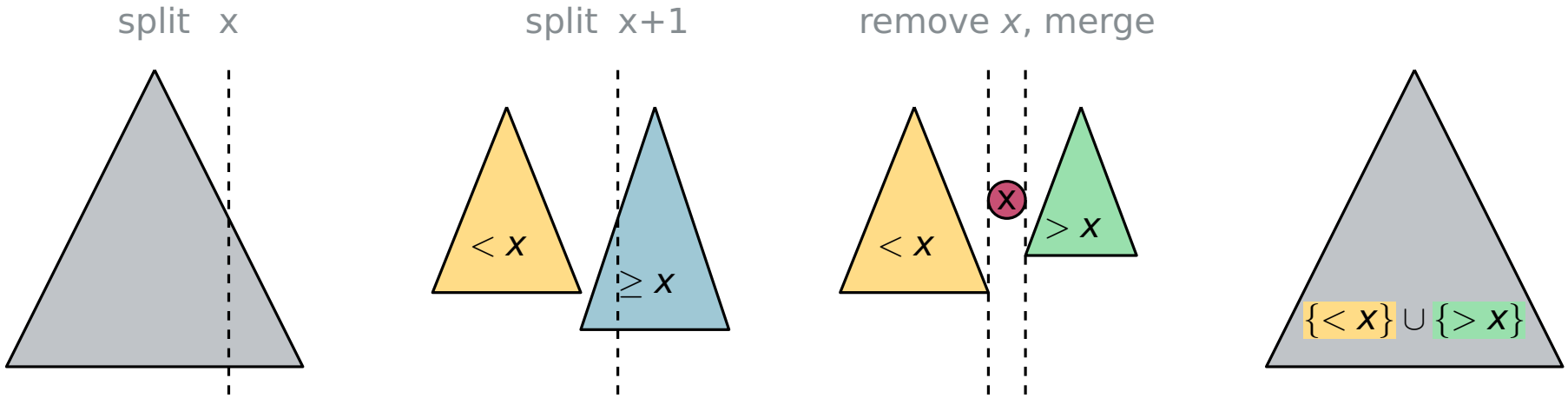


Implementation

we break down everything to two operations

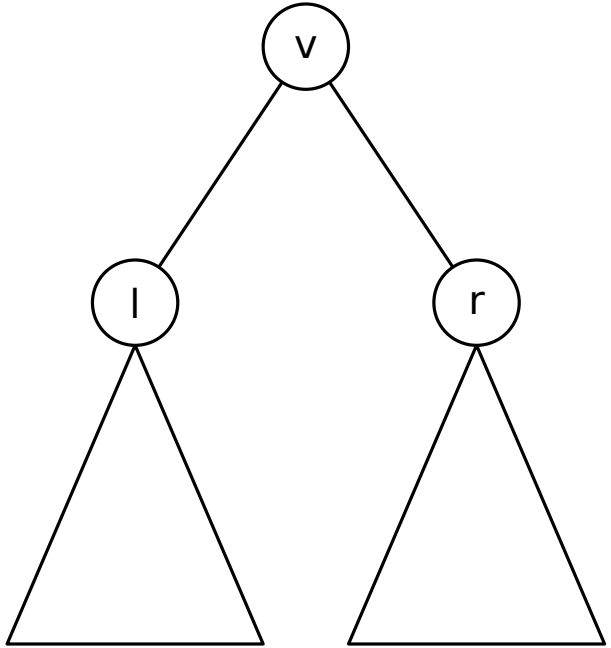


E.g. **delete** x:



Split

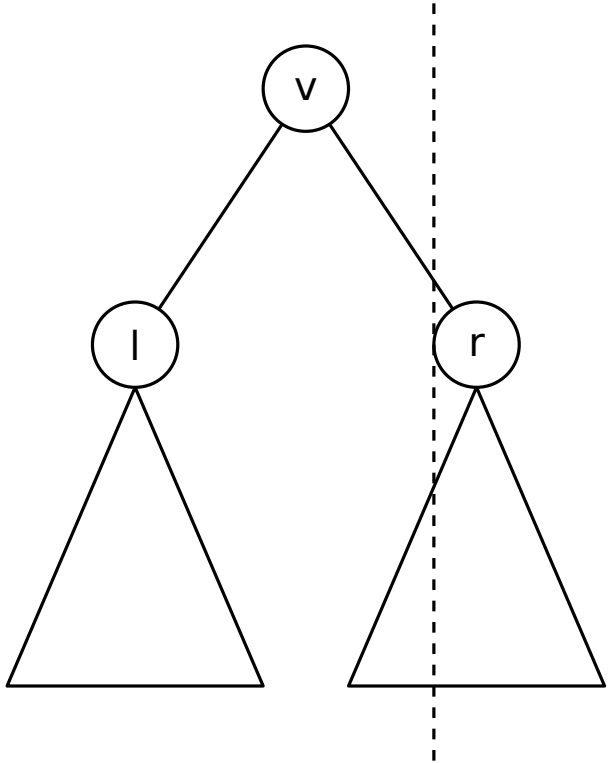
returns two treaps containing all nodes $<$ and $\geq$ the given key respectively



split(Node v, Key x)

Split

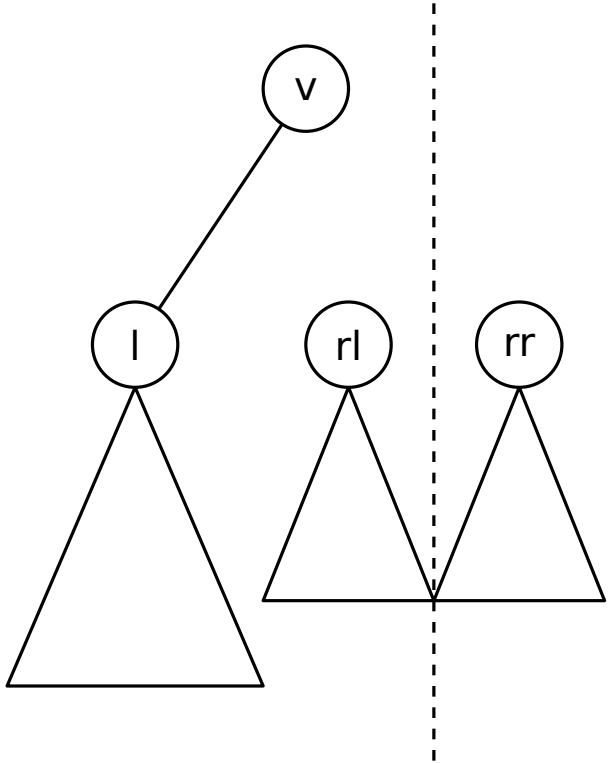
returns two treaps containing all nodes $<$ and $\geq$ the given key respectively




```
split(Node v, Key x)
    if v.key < x:
        ...
    else
        ...
```

Split

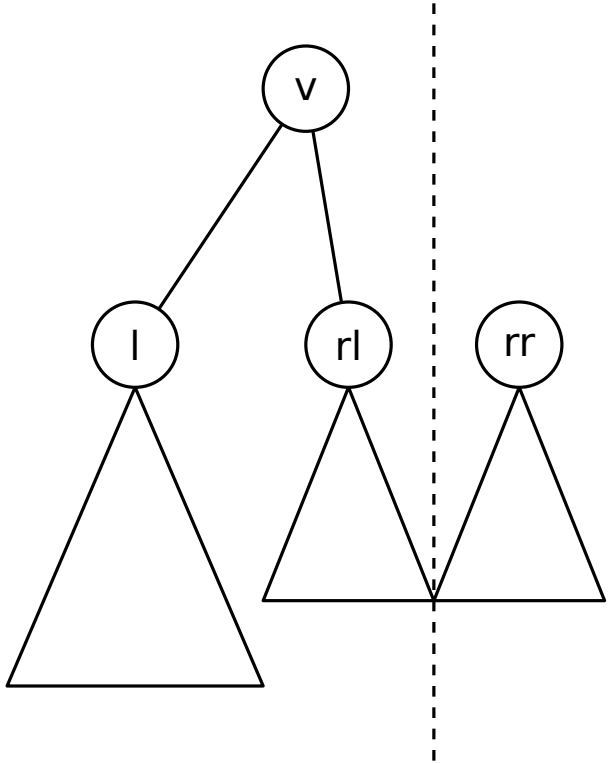
returns two treaps containing all nodes $<$ and $\geq$ the given key respectively




```
split(Node v, Key x)

    if v.key < x:
        rl, rr = split(v.right, x)



    else
            ...
```

Split

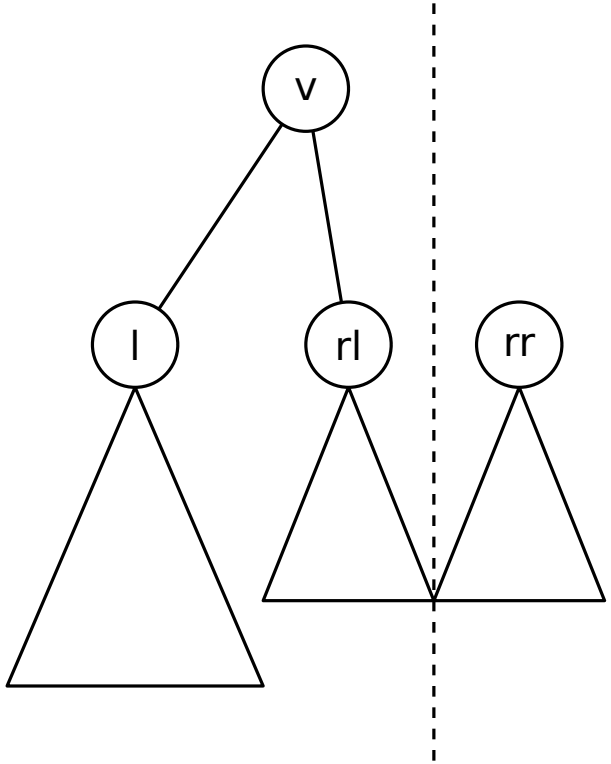
returns two treaps containing all nodes $<$ and $\geq$ the given key respectively




```
split(Node v, Key x)

    if v.key < x:
        rl, rr = split(v.right, x)
        v.right = rl


    else
            ...
```

Split

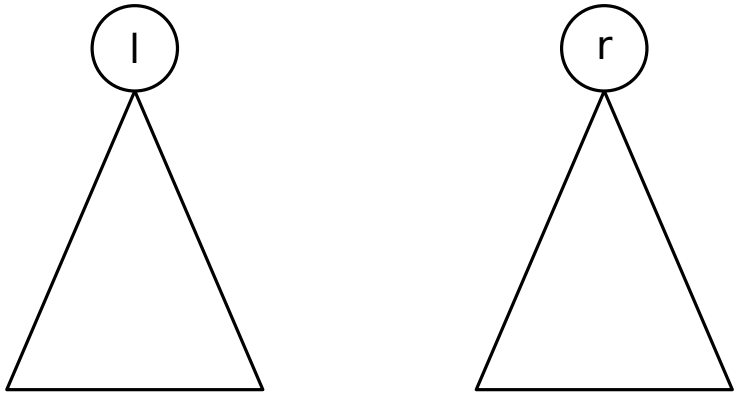
returns two treaps containing all nodes $<$ and $\geq$ the given key respectively




```
split(Node v, Key x)

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

Merge

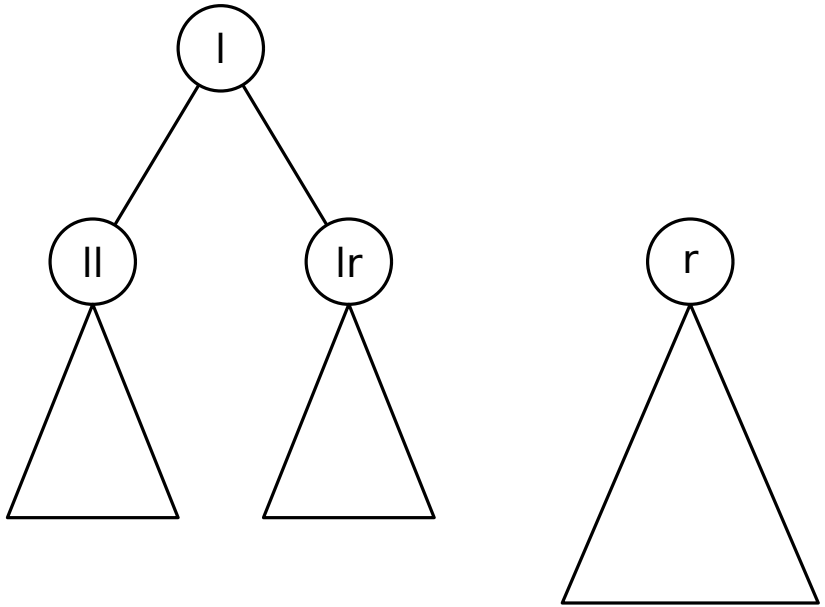
merge two treaps into one preserving horizontal ordering (left, right)



merge(Node l, Node r)

Merge

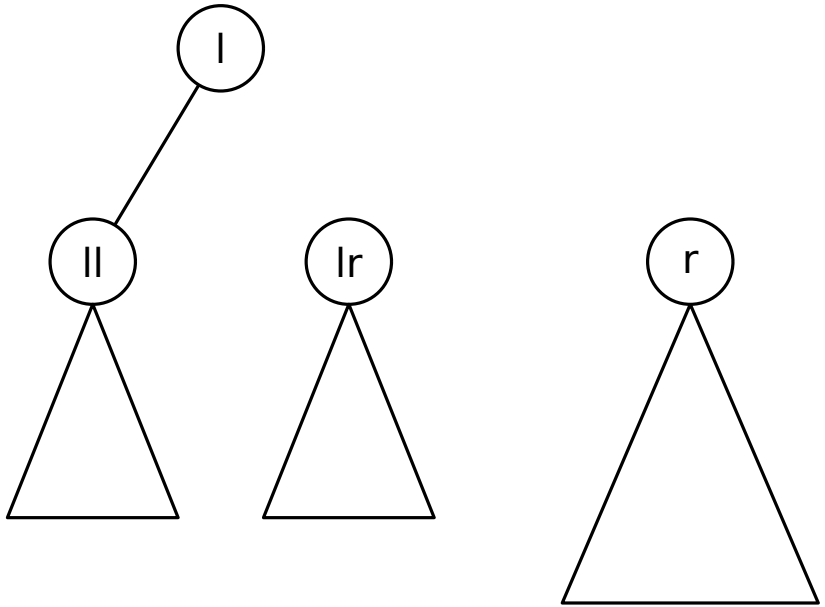
merge two treaps into one preserving horizontal ordering (left, right)




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

Merge

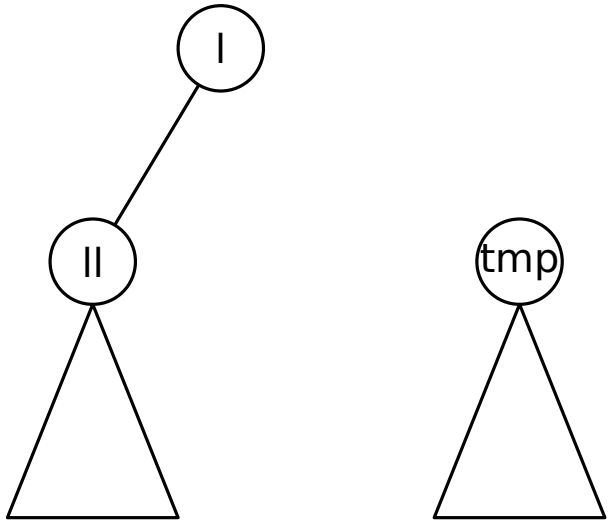
merge two treaps into one preserving horizontal ordering (left, right)




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

Merge

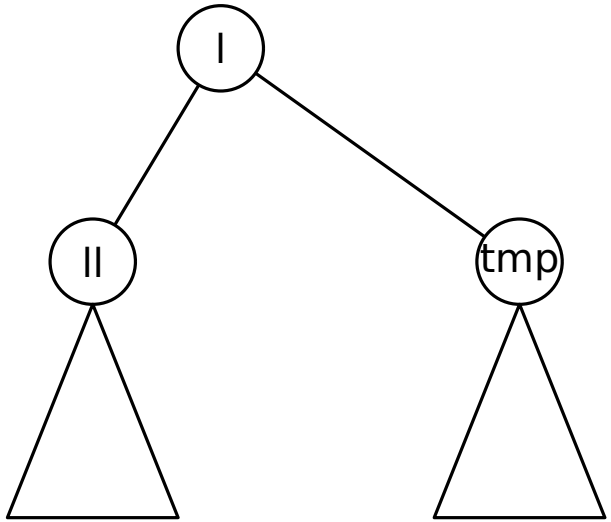
merge two treaps into one preserving horizontal ordering (left, right)




```
merge(Node l, Node r)

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



    else
            ...
```

Merge

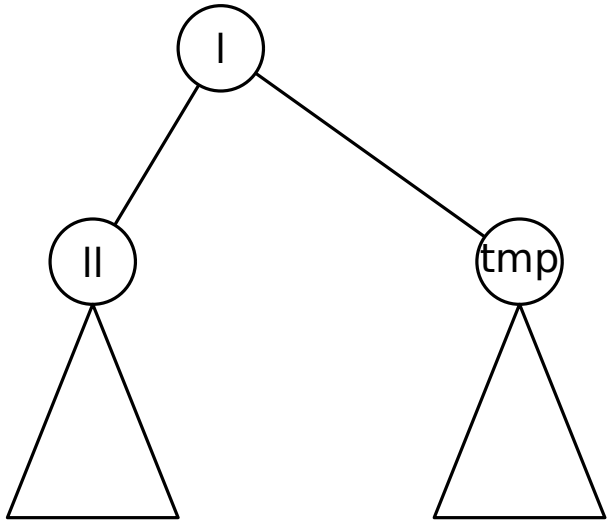
merge two treaps into one preserving horizontal ordering (left, right)




```
merge(Node l, Node r)

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

    else
            ...
```

Merge

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

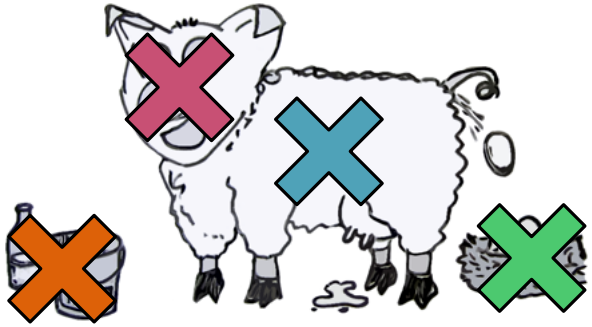



```
merge(Node l, Node r)
    if l.prio < r.prio:
        tmp = merge(l.right, r)
        l.right = tmp
        return l
    else
            ...
```

What a Treap can do

Self balancing BSTs
(like a set)

- find value
- insert value
- delete value

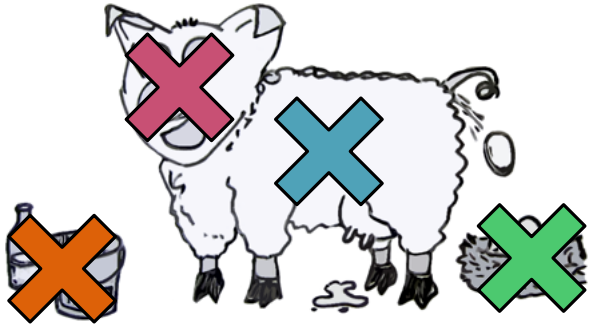


binary search tree

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

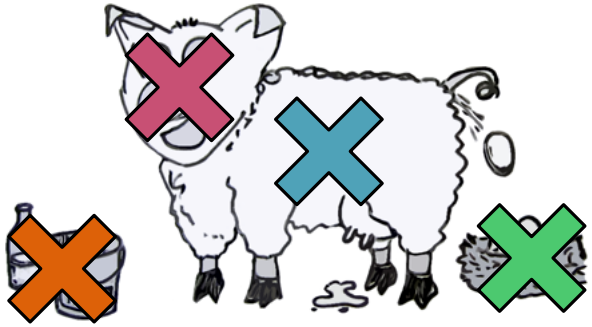


binary search tree

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)

    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?

```
update(Node v)

    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))

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

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

Aggregates

we now want to answer queries about our data

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

```
update(Node v)

    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))

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

- 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]

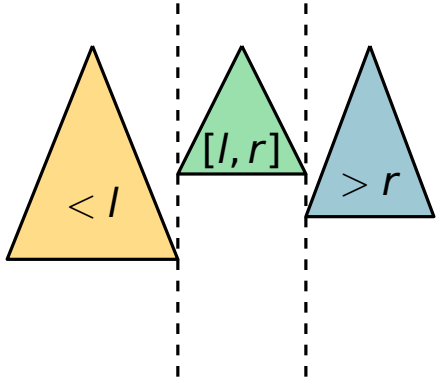
Aggregates

we now want to answer queries about our data

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

```
update(Node v)

    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))

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

- 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: $< l$, $[l, r]$, $> r$



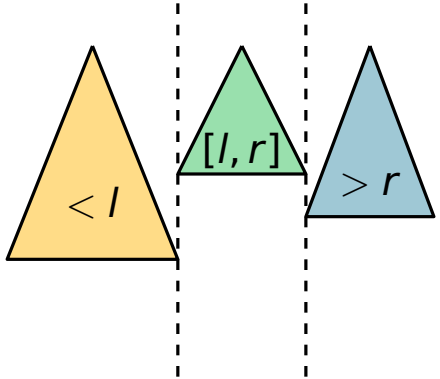
Aggregates

we now want to answer queries about our data

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

```
update(Node v)

    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))

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

- 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



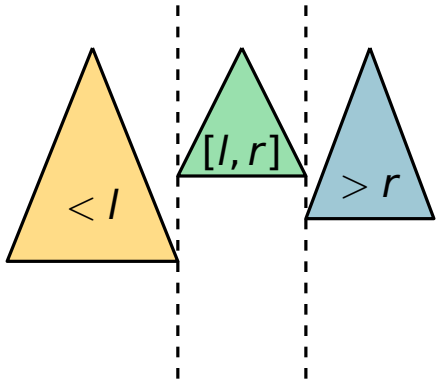
Aggregates

we now want to answer queries about our data

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

```
update(Node v)

    v.gcd = gcd( v.key,
                 gcd(v.left),
                 gcd(v.right))

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

- 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



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

```
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

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

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

5 min break

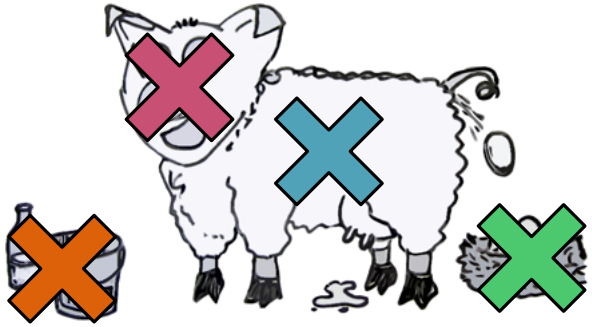
- an important operation on sets is called **order of key**
- given a key k, it returns the number of elements in the set that are smaller than k

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

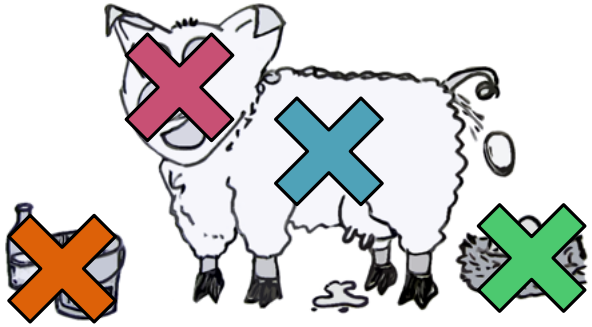
aggregates

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

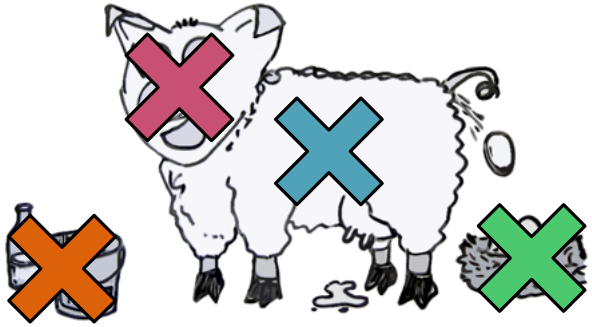
aggregates

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

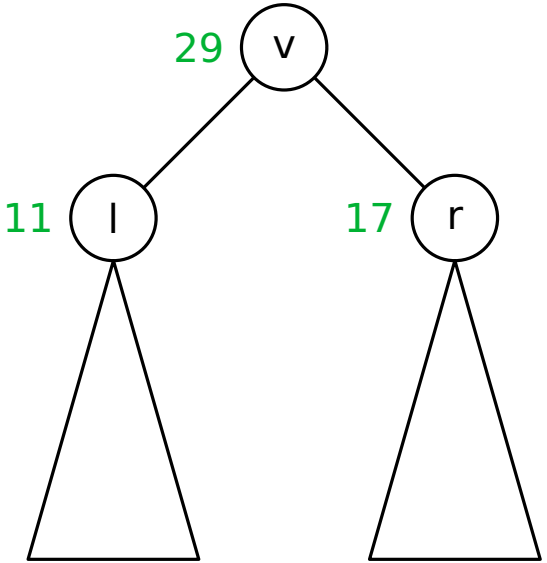
- now that we have aggregates; we can split based on aggregates
- e.g. split based on size of prefix

splits off the *k* leftmost nodes

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



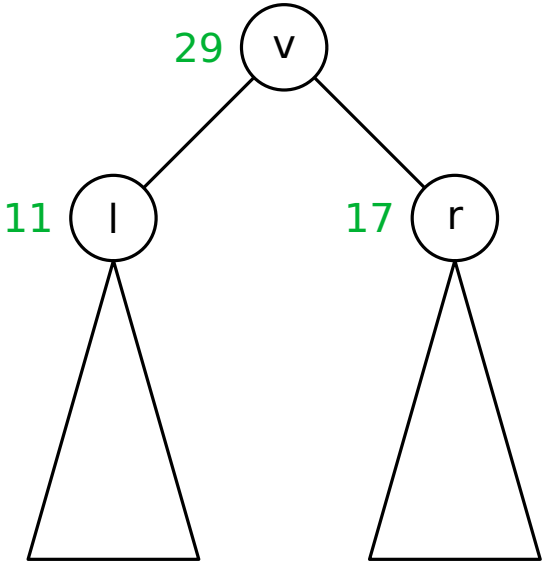
splits off the *k* leftmost nodes

split-size(Node v, int k)

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



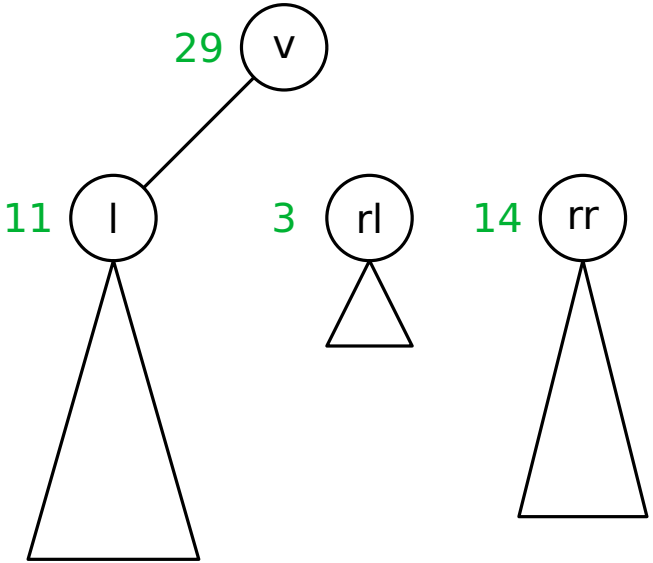
splits off the *k* leftmost nodes

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

    if 1+size(v.left) ≤ k:

    else
```

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



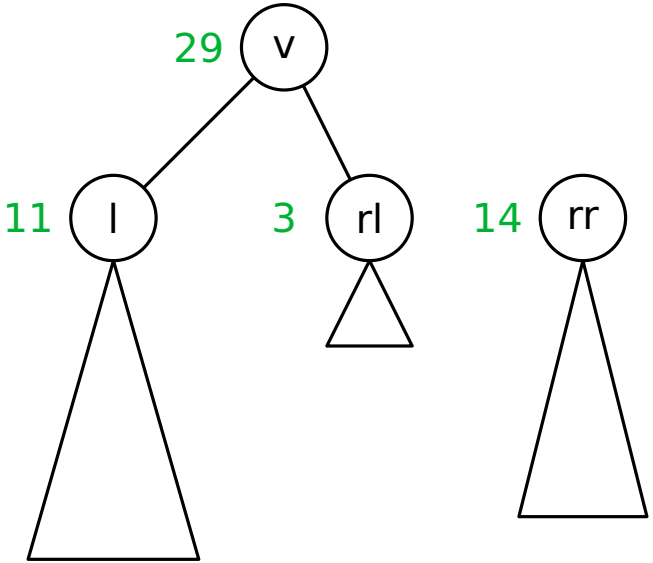
splits off the *k* leftmost nodes

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

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



    else
```

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



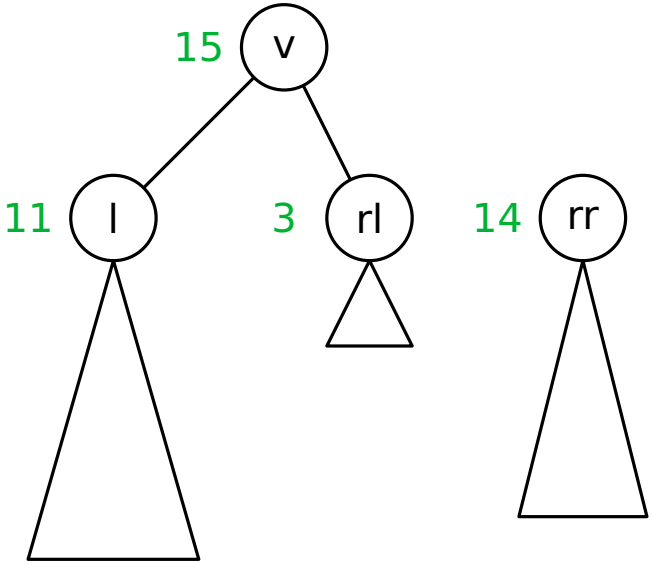
splits off the *k* leftmost nodes

```
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
```

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



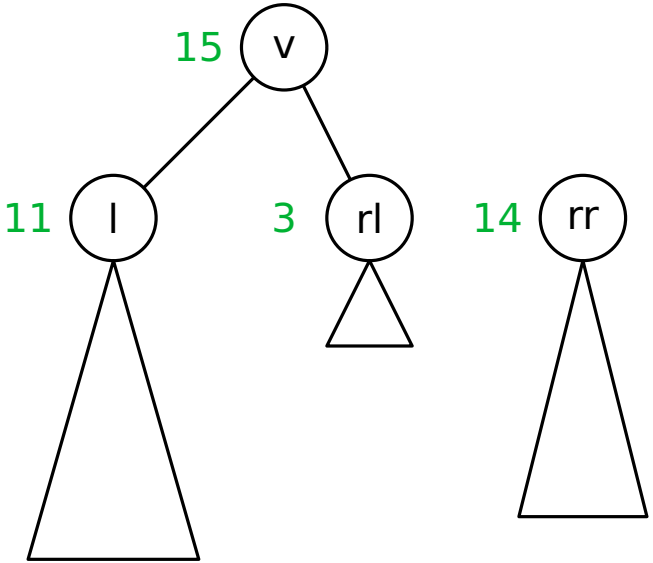
splits off the *k* leftmost nodes

```
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
```

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



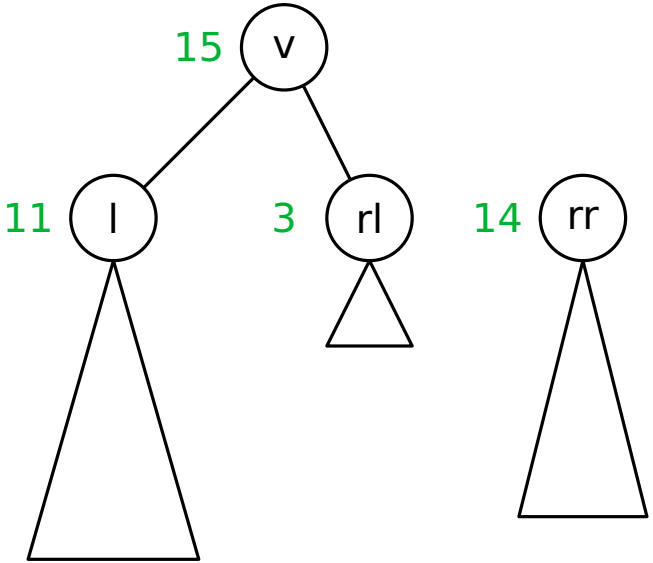
splits off the *k* leftmost nodes

```
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
```

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



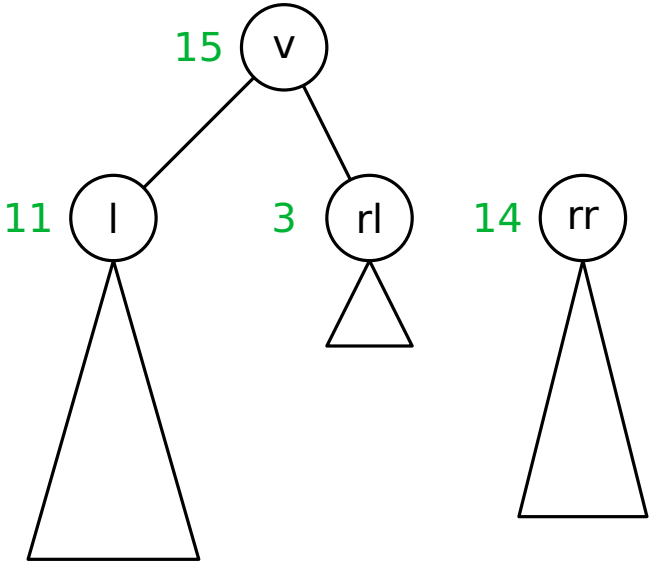
splits off the *k* leftmost nodes

```
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}
```

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

```
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}
```

SIZE

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

- lets drop the BST property of our treap

Implicit Treaps

this is the point were Treaps really start to lay eggs

- lets drop the BST property of our treap
- conceptually we now work like an array instead of a set

Implicit Treaps

this is the point were Treaps really start to lay eggs

- 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

Implicit Treaps

this is the point were Treaps really start to lay eggs

- 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

Implicit Treaps

this is the point were Treaps really start to lay eggs

- 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**

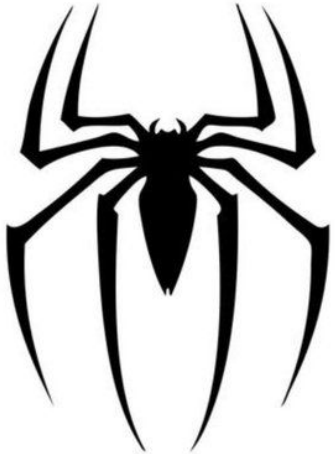
  e.g. give me the first 5 nodes
- positions of nodes are now implicit
- merge will preserve order

Implicit Treaps

this is the point were Treaps really start to lay eggs

- 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

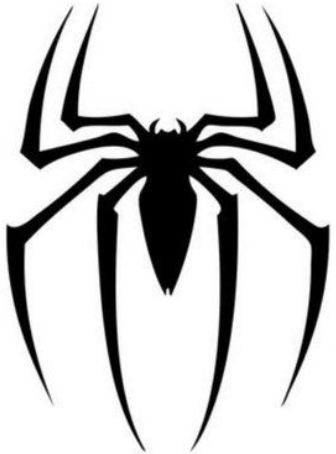


Implicit Treaps

this is the point were Treaps really start to lay eggs

- 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



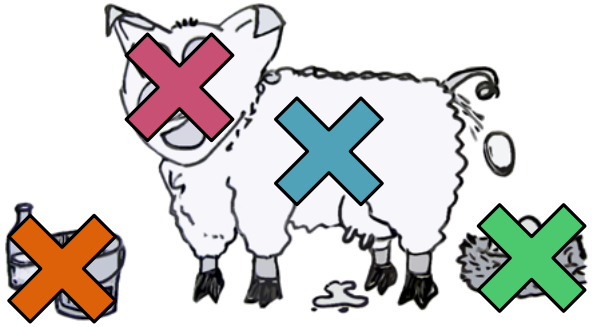
with great power comes great responsibility

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

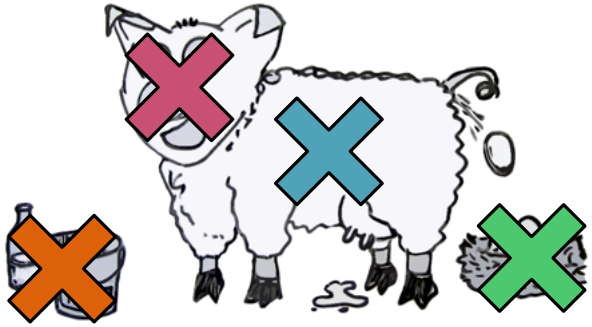
What a Treap can do

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

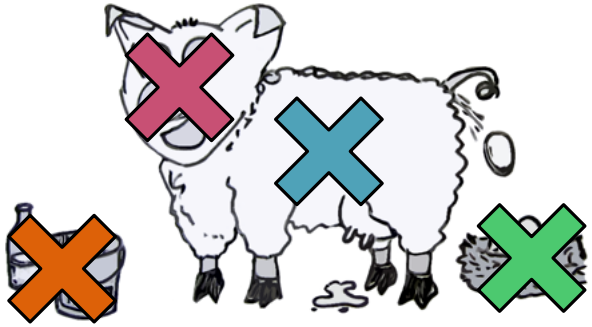
What a Treap can do

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

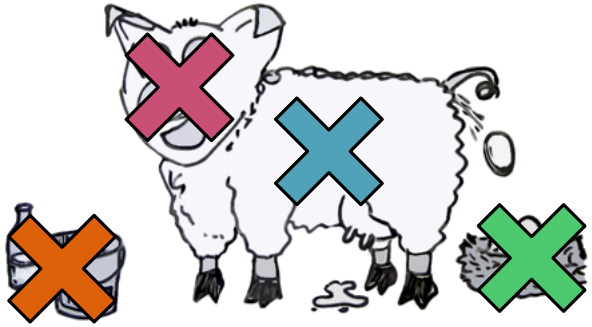
What a Treap can do

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

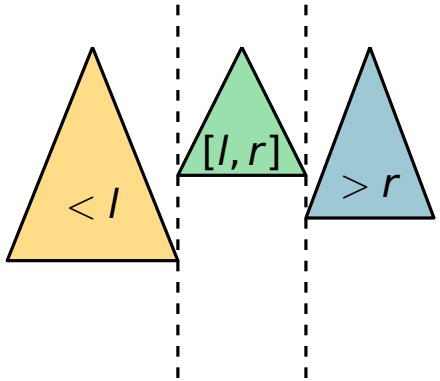
- same concept as for segment trees

Treaps + Lazy Propagation

- same concept as for segment trees
- before recursing, push pending updates to children

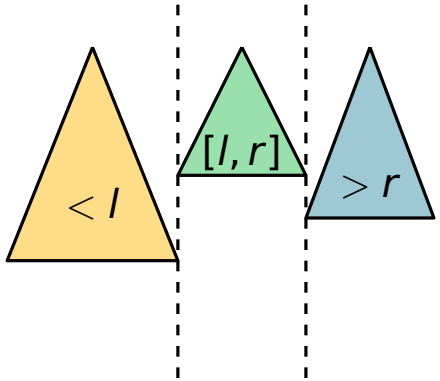
Treaps + Lazy Propagation

- 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

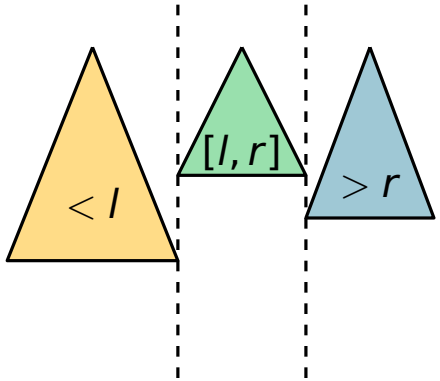
- same concept as for segment trees
- before recursing, push pending updates to children
- update range by splitting it out and apply change to root
- cool application: reverse order of range



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

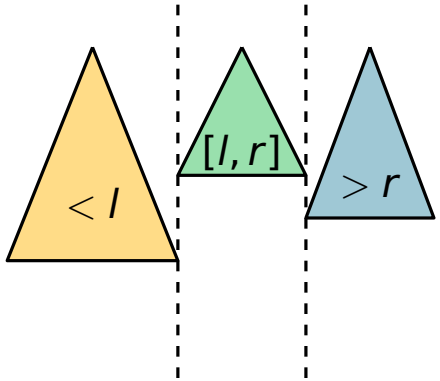
  reverse order $\equiv$ for all nodes swap left and right child



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 $\equiv$ for all nodes swap left and right child




```
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
```

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

Treaps + Lazy Propagation

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

if you read from children here (e.g. split-size), push must be earlier!!!

```
split(Node v, Key x)
    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)
    if l.prio < r.prio:
        push(l)
        tmp = merge(l.right, r)
        l.right = tmp
        update(l)
        return l
    else
            ...
```

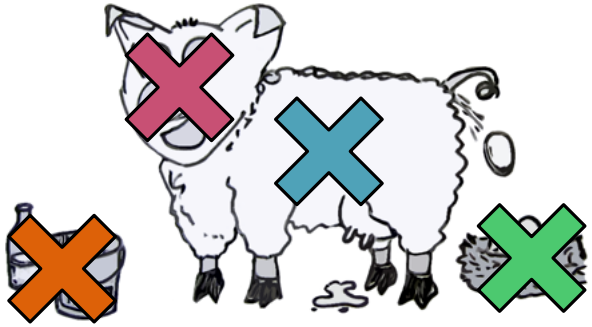
What a Treap can do

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

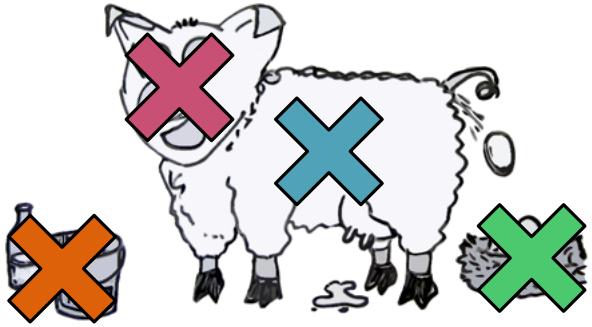
What a Treap can do

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

split

```
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

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`

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)

split

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

merge

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

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

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

split

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

merge

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

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

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; }
```

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()`

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

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));`
- ignore memory leaks

Advanced Optimizations

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

Advanced Optimizations

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

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

merge(Node l, Node r)

if l.prio $<$ r.prio:

...

else

...

→

merge(Node l, Node r)

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

...

else

...

Advanced Optimizations

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

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

merge(Node l, Node r)

if l.prio $<$ r.prio:

. . .

else

. . .

→

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

Advanced Optimizations

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

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

merge(Node l, Node r)

if l.prio $<$ r.prio:

. . .

else

. . .

→

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

- $O(n)$ construction

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

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

Advanced Optimizations

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

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

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

→

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

- $O(n)$ 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]`

Advanced Optimizations

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

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

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

→

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

- $O(n)$ 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]`
    pros: Node stores two 32-bit ints instead of two 64-bit pointers

Advanced Optimizations

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

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

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

→

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

- $O(n)$ 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]`
    pros: Node stores two 32-bit ints instead of two 64-bit pointers
    cons: ugly

It’s dangerous to go alone! Take this.

```
void debug(Treap* t) {
        if(!t) return;
        push(t);
        debug(t->left);
        cout << t->key << ' ';
        debug(t->right);
 }
```