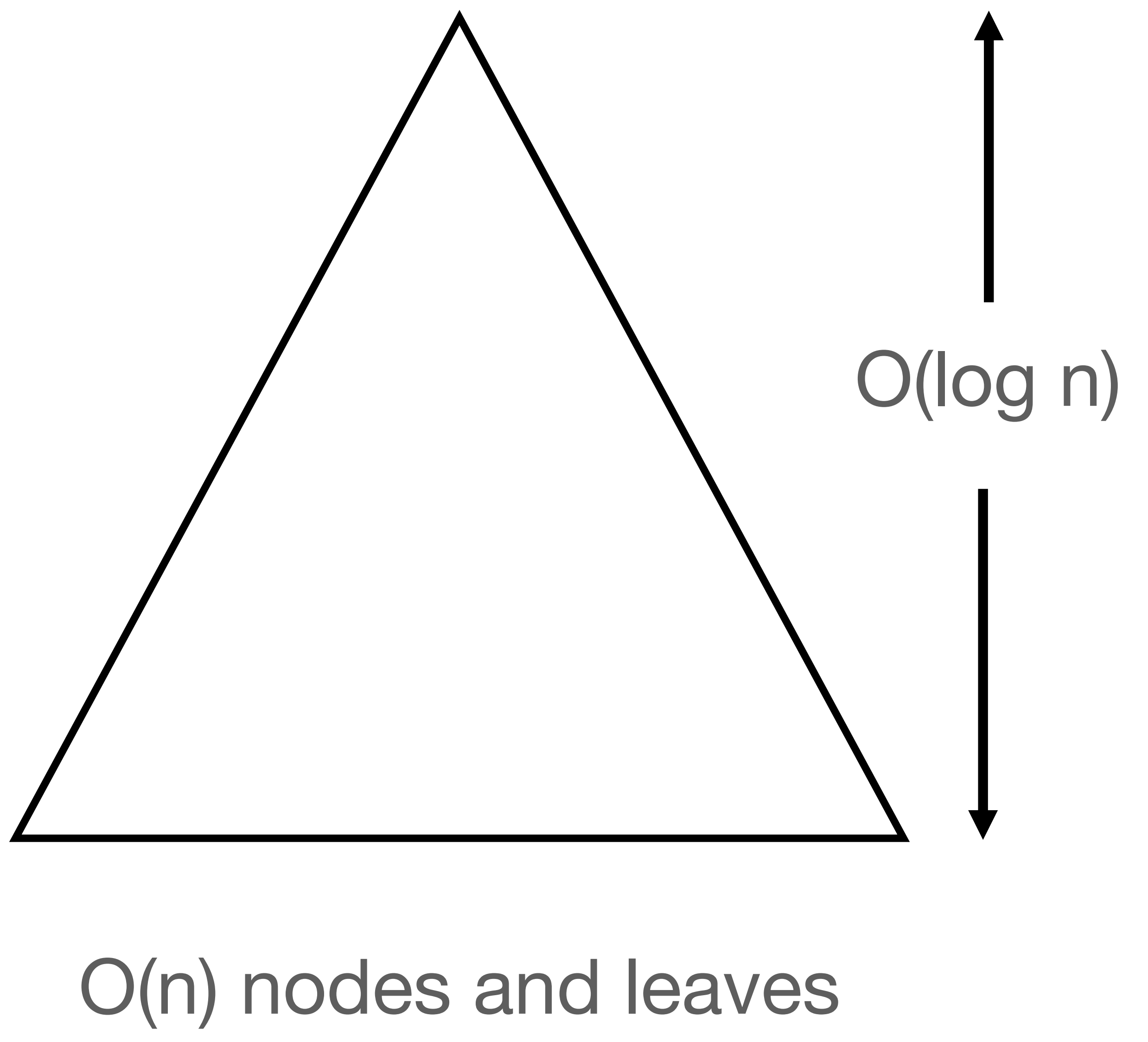
balanced trees 1

2-3-trees

balanced trees: idea

- maintain ordered set S
- #S = n
- operations find, insert delete,…
- store elements in nodes of tree with
  - O(n) nodes
  - depth O(log n)
  - ‚from left to right‘

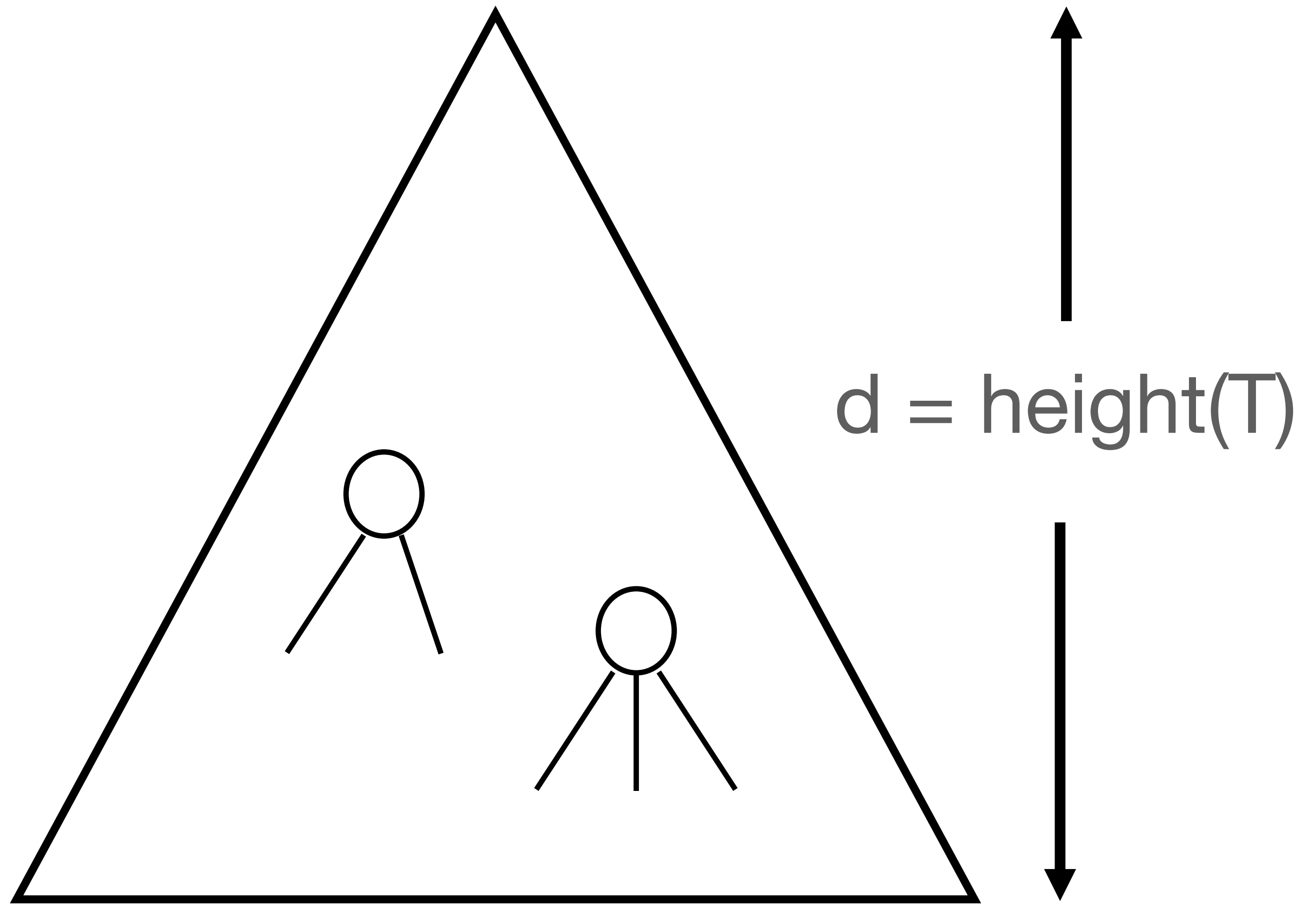
balanced trees: idea



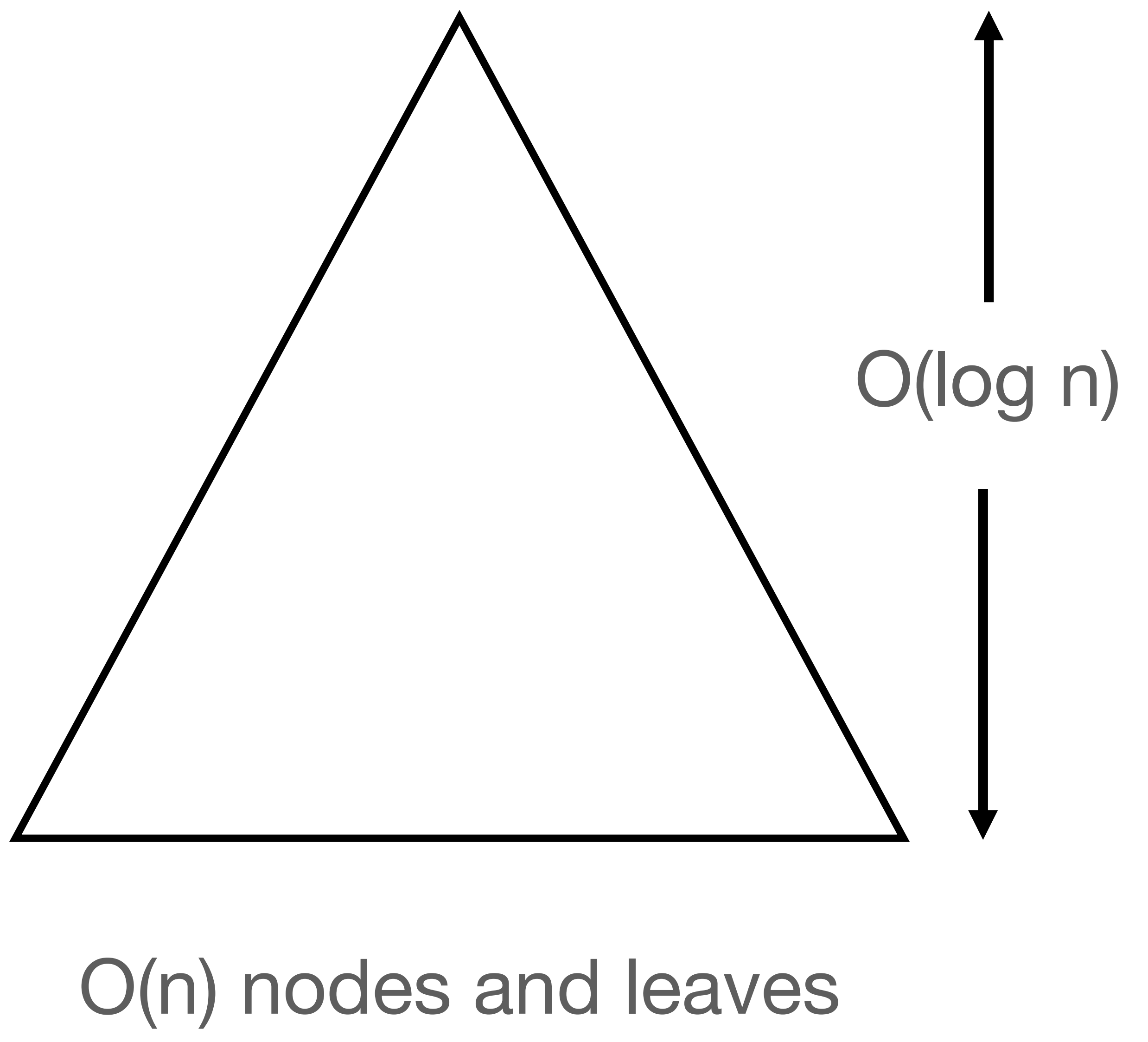
- maintain ordered set S
- #S = n
- operations find, insert delete,…
- store elements in nodes of tree with
  - O(n) nodes
  - depth O(log n)
  - ‚from left to right‘

- maintain something close to a complete binary tree
- rebalance after insert or delete

2-3-trees T: definition



- every interior node has 2 or 3 sons
- all leaves have the same depth d
- d = height(T)

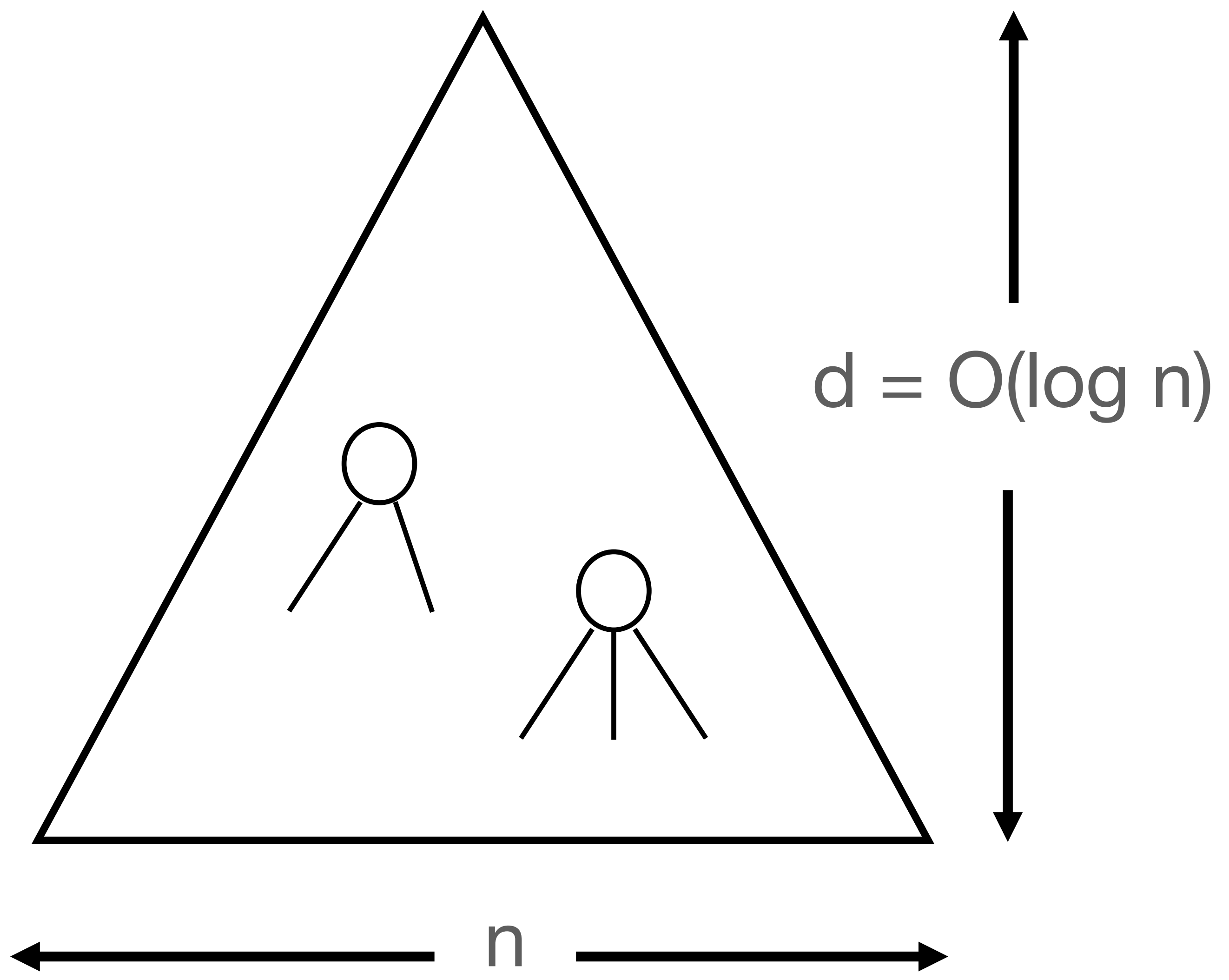


balanced trees: idea

- maintain ordered set S
- #S = n
- operations find, insert delete,…
- store elements in nodes of tree with
  - O(n) nodes
  - depth O(log n)
  - ‚from left to right‘

- maintain something close to a complete binary tree
- rebalance after insert or delete

- examples here
  - **2-3-trees**
    - easy rebalancing scheme
    - constant factor is an issue
  - AVL trees
    - analysis is more ‚advanced‘



2-3-trees T: definition

- every interior node has 2 or 3 sons
- all leaves have the same depth d
- d = height(T)

Lemma 1. *Let L be the number of leaves of a 2-3-tree of height d. Then*

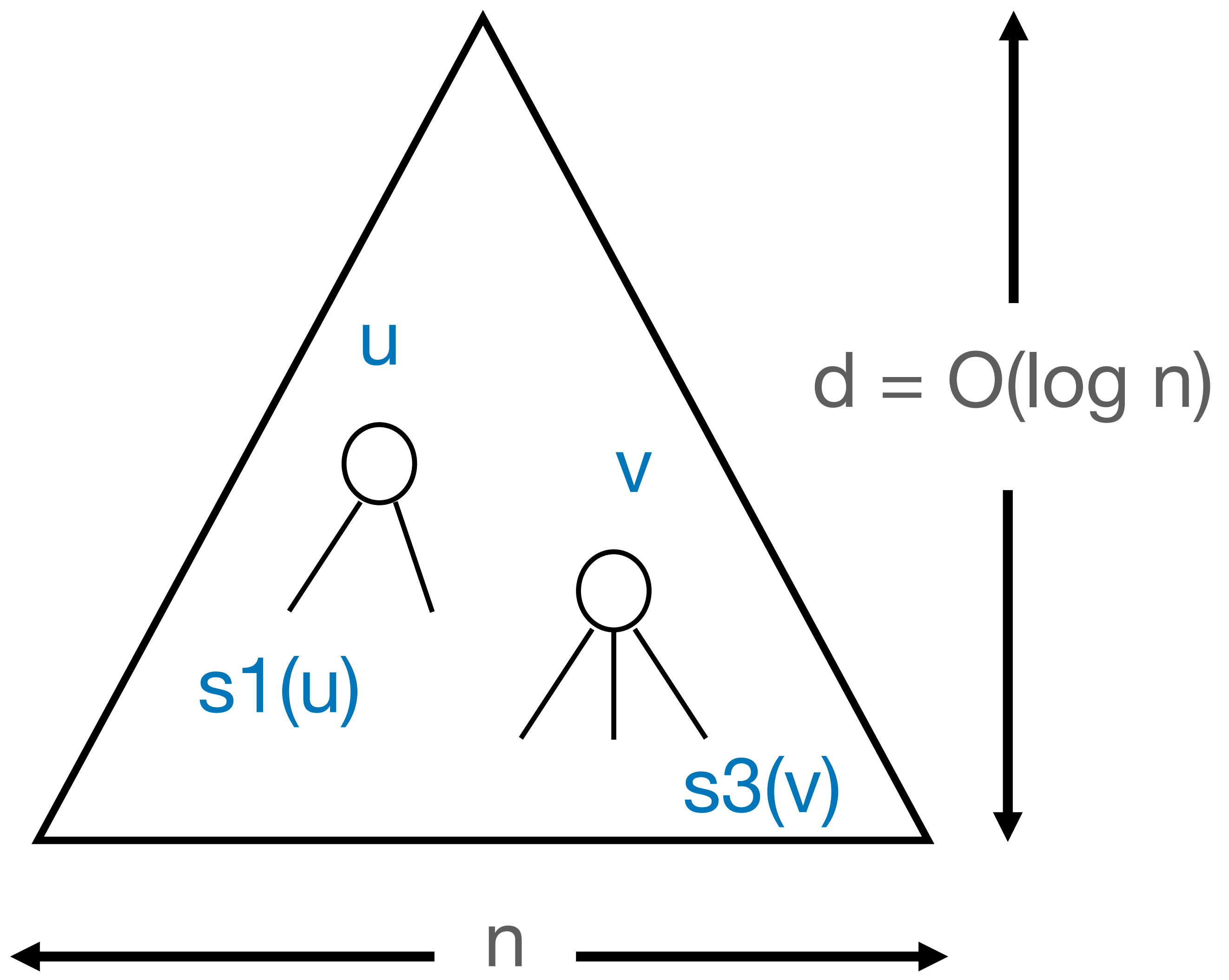
$$2^d \leq L \leq 3^d$$

Proof. induction on d

store elements $s \in S$ in leaves from left to right

$$\#L = n$$
$$d = O(\log n)$$

2-3-trees T: implementation in C0

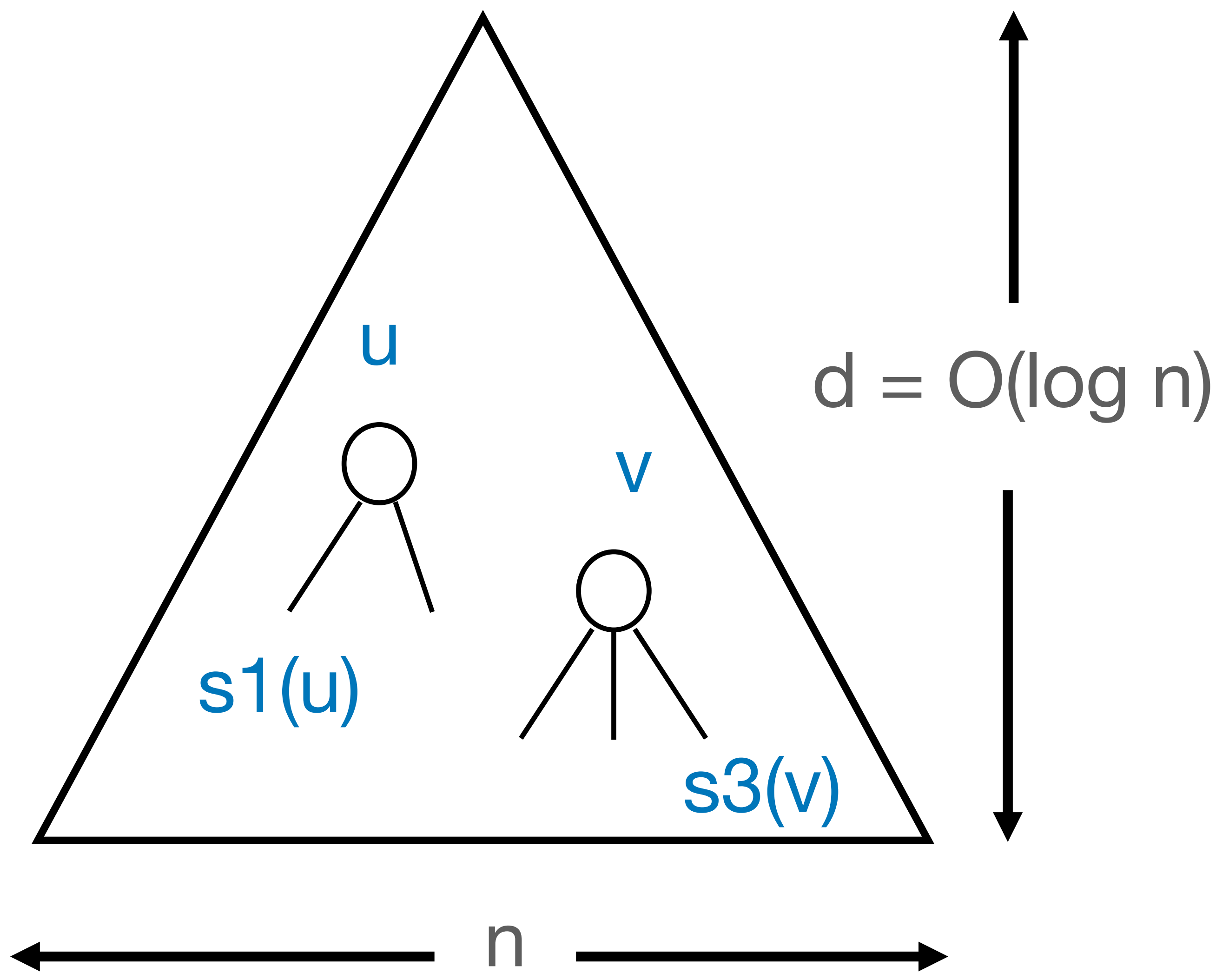


u.s3 = null

nodes u are structs with components

- p: parent
- s1, s2, s3: sons
  - u.sx= null: son not present
  - u.sx = null for all x: leaf
- key: for elements $s \in S$
- max: maximal key stored in T(u)

2-3-trees T: implementation in C0



u.s3 = null

nodes u are structs with components

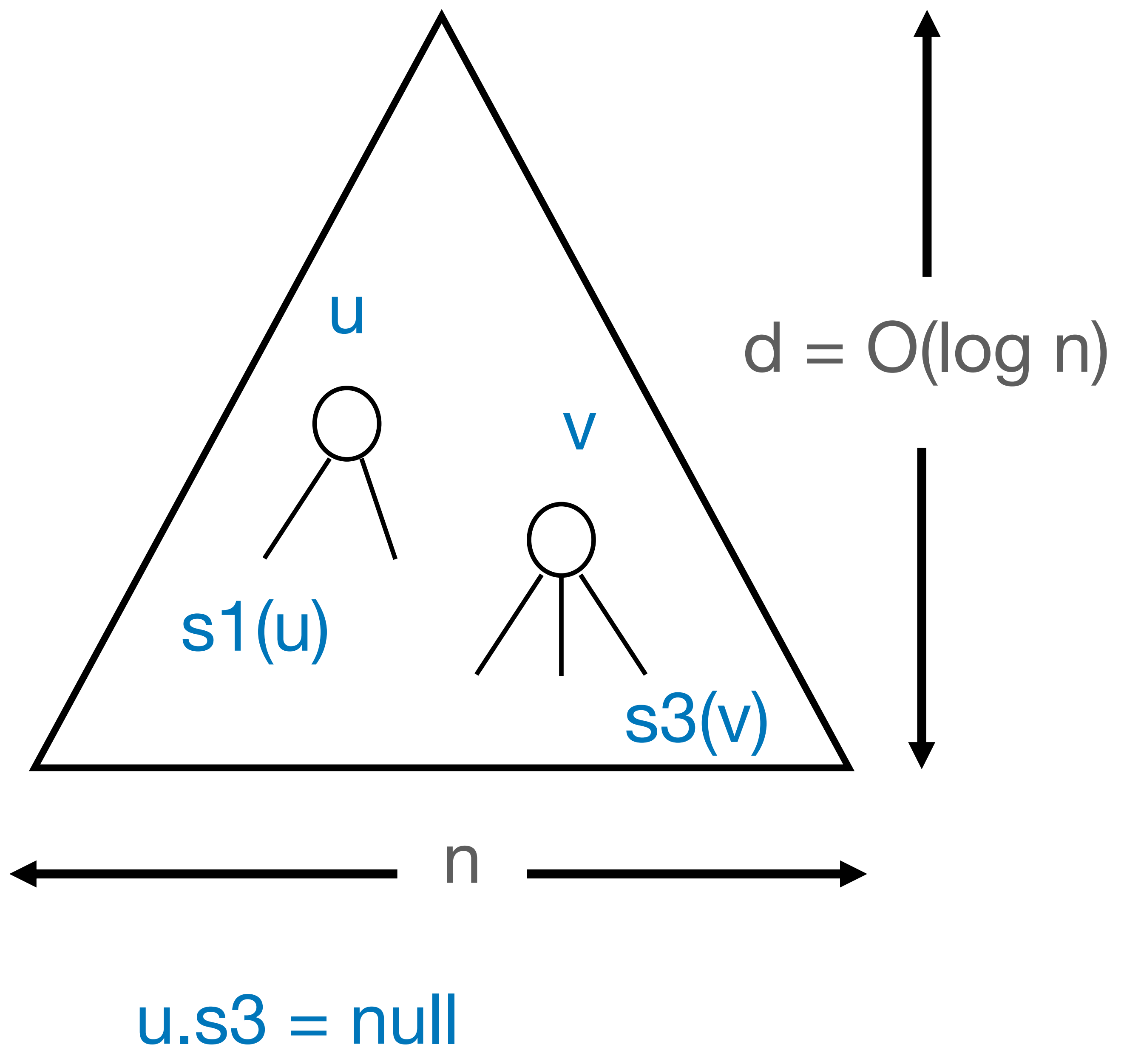
- p: parent
- s1, s2, s3: sons
  - u.sx= null: son not present
  - u.sx = null for all x: leaf
- key: for elements $s \in S$
- max: maximal key stored in T(u)

Notation (Java):

u reference to object

$$\begin{array}{rcl} u.y & & \text{dereference, then take attribute } y) \\ sx(u) & = & u.sx \quad (\text{son } x \text{ of } u) \\ p(u) & = & u.p \quad (\text{parent of } u) \\ key(u) & = & u.key \\ max(u) & = & u.max \end{array}$$

2-3-trees T: locate (x,u) and find(x)



$locate(u,x)$: locate position of x in $T(u)$

- $\ell_1, \ldots, \ell_n$ leaves of $T(u)$ from left to right with

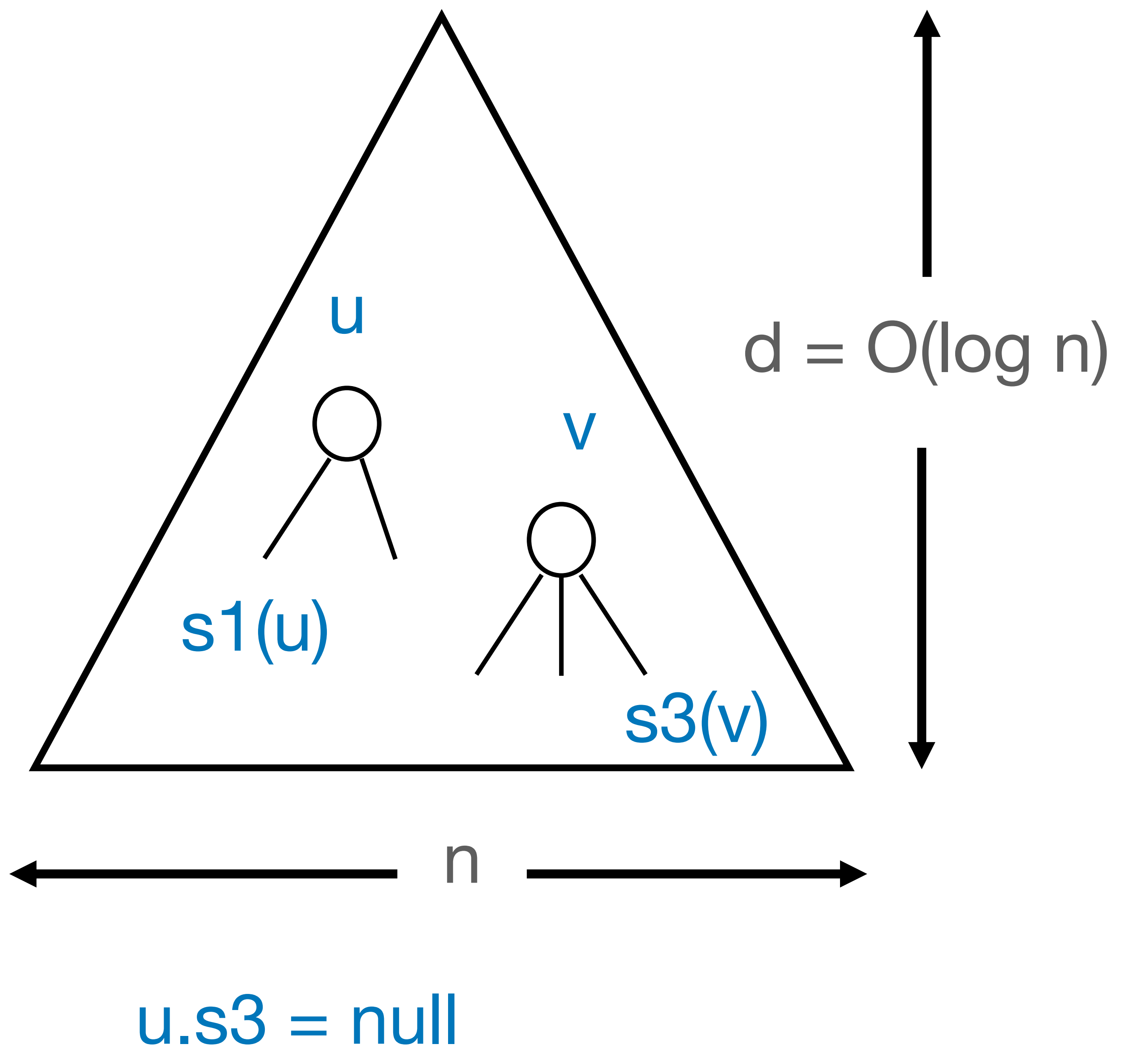
$$key(\ell_1) \leq \ldots \leq key(\ell_n)$$

- input x possibly in S

- output is a leaf

$$locate(u,x) = \begin{cases} \ell_{\min\{i \mid key(\ell_i) \geq x\}} & \text{if it exists} \\ \ell_n & x > key(\ell_n) \end{cases}$$

2-3-trees T: locate (x,u) and find(x)



$locate(u,x)$: locate position of x in $T(u)$

- $\ell_1, \ldots, \ell_n$ leaves of $T(u)$ from left to right with

$$key(\ell_1) \leq \ldots \leq key(\ell_n)$$

- input x possibly in S

- output is a leaf

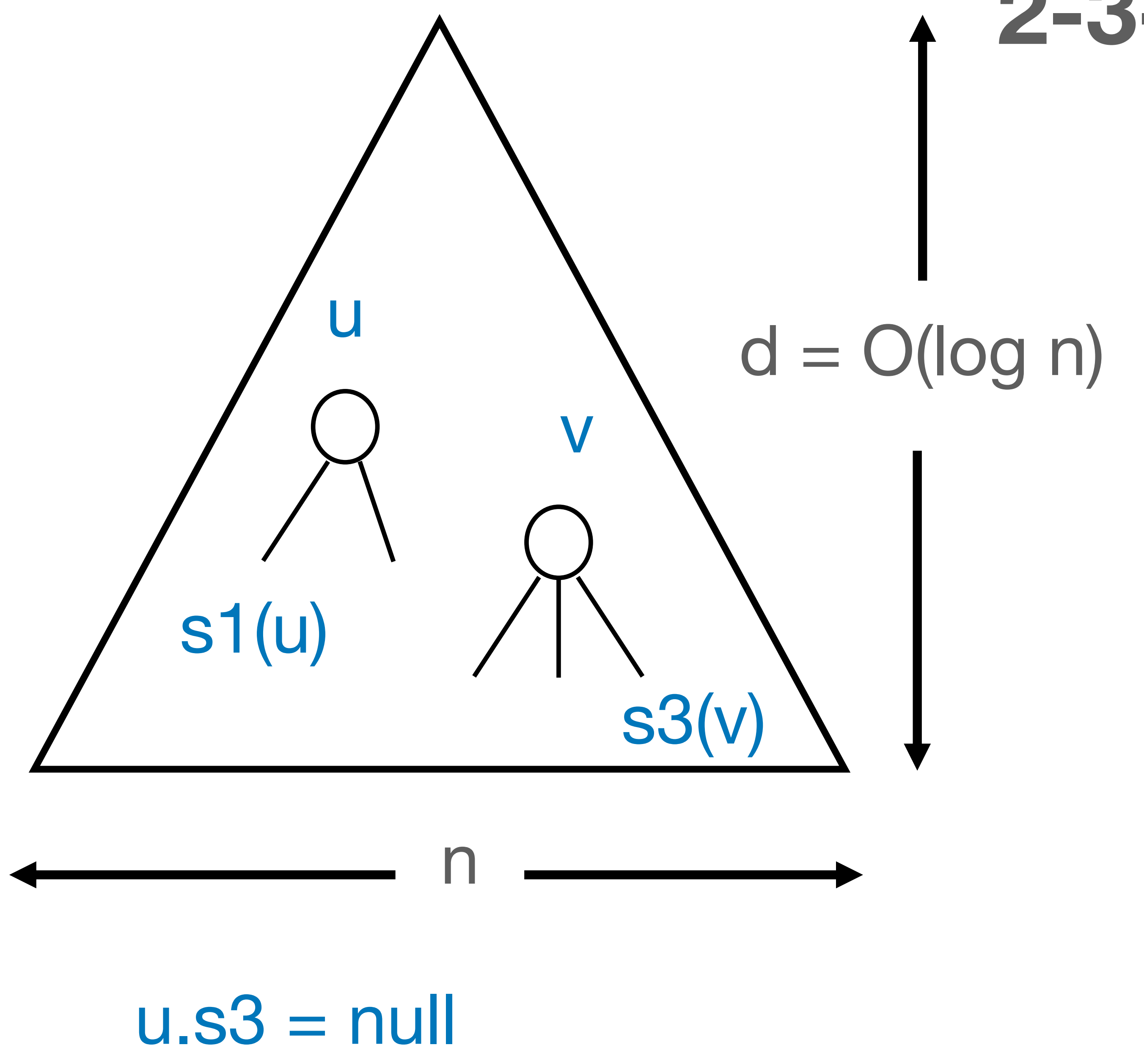
$$locate(u,x) = \begin{cases} \ell_{\min\{i \mid key(\ell_i) \geq x\}} & \text{if it exists} \\ \ell_n & x > key(\ell_n) \end{cases}$$

$find(x)$: determine if $x \in S$

$$find(x) = \begin{cases} 1 & x \in S \\ 0 & x \notin S \end{cases}$$

```
find(x) = if key(locate(root, x)) = x {1} else {0}
```

2-3-trees T: implementation of locate(x,u)



$locate(u,x)$: locate position of x in $T(u)$

- $\ell_1, \ldots, \ell_n$ leaves of $T(u)$ from left to right with

$$key(\ell_1) \leq \ldots \leq key(\ell_n)$$

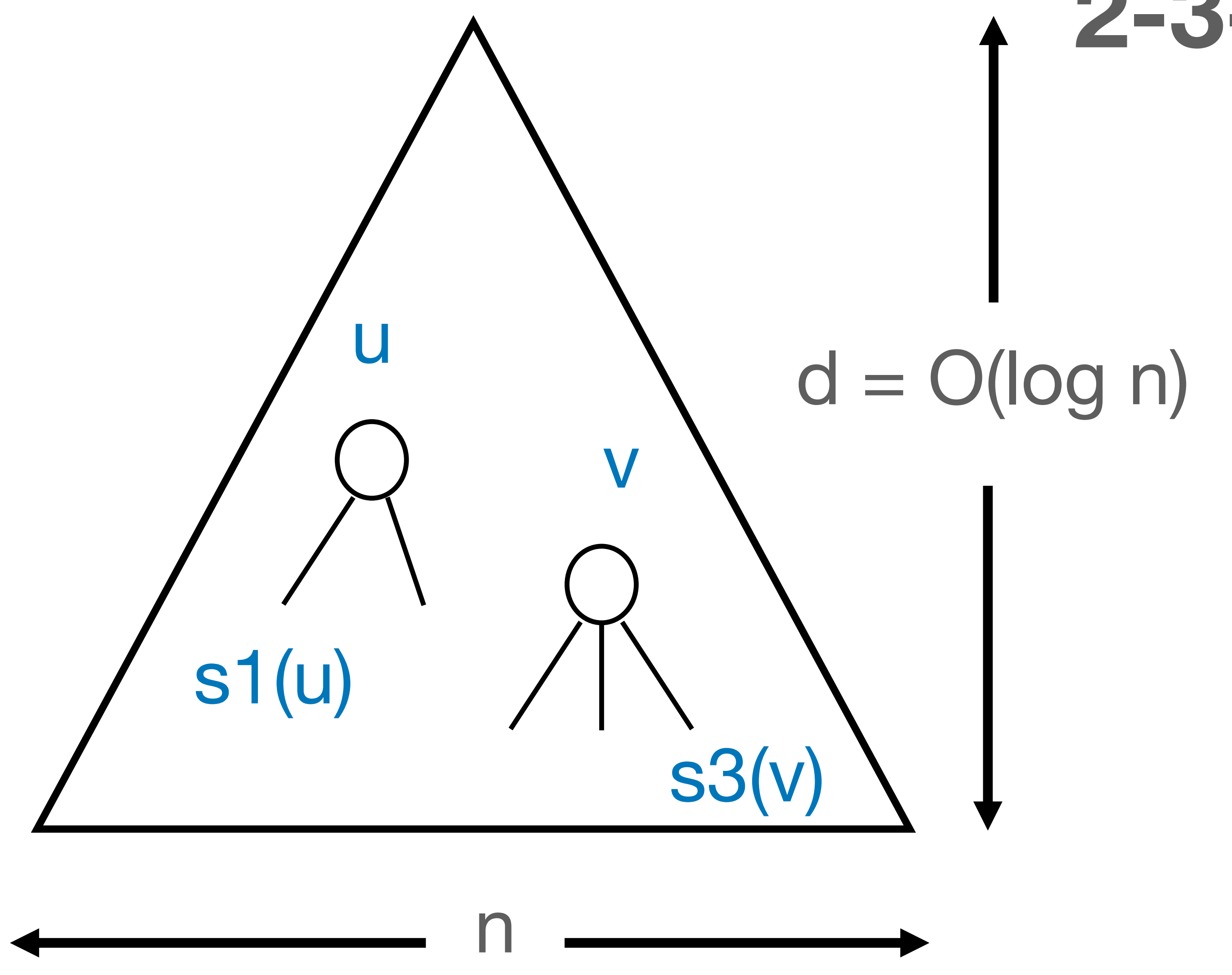
- input x possibly in S

- output is a leaf

$$locate(u,x) = \begin{cases} \ell_{\min\{i \mid key(\ell_i) \geq x\}} & \text{if it exists} \\ \ell_n & x > key(\ell_n) \end{cases}$$

```
base case u is leaf: return u

u is interior node with 2 sons /* u.s3=null */

if key(max(s1(u)) >= x {locate(x, s1(u)} else {locate(x,s2(u)}

u is interior node with 3 sons

if key(max(s1(u)) >= x {locate(x, s1(u)} else
{if key(max(s2(u) >= x >= {locate(x,s2)}  else {locate(x, s3(u))}}}
```

2-3-trees T: implementation of locate(x,u)



u.s3 = null

$\boxed{\text{run time } O(\log n)}$

$locate(u,x)$: locate position of x in $T(u)$

- $\ell_1, \ldots, \ell_n$ leaves of $T(u)$ from left to right with

$$key(\ell_1) \leq \ldots key(\ell_n)$$

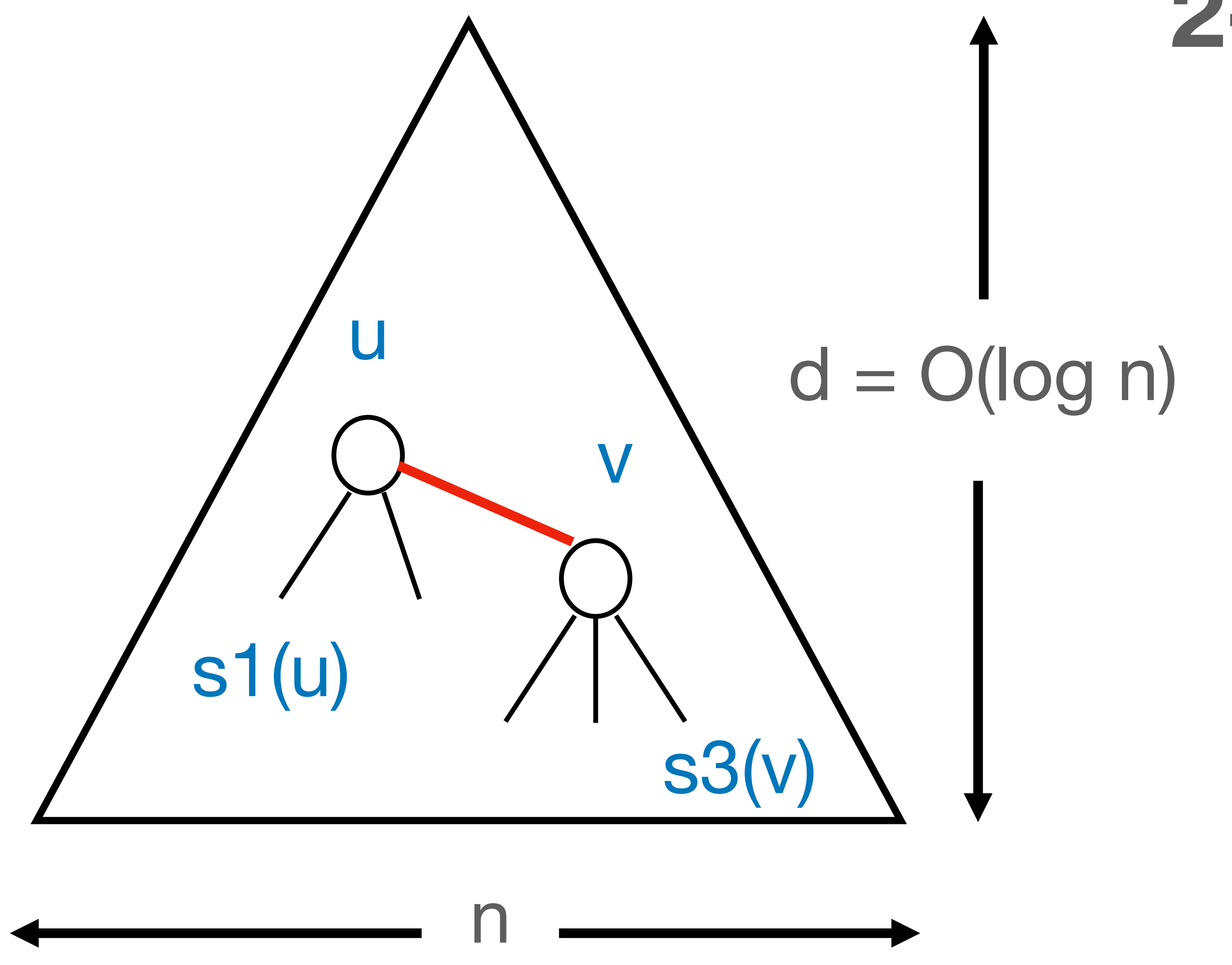
- input x possibly in S

- output

$$locate(u,x) = \begin{cases} \min\{i \mid key(\ell_i) \geq x\} & \text{if it exists} \\ 0 & x < key(\ell_1) \end{cases}$$

```
base case u is leaf: return u

u is interior node with 2 sons /* u.s3=null */

if key(max(s1(u)) >= x {locate(x, s1(u)} else {locate(x,s2(u)}

u is interior node with 3 sons

if key(max(s1(u)) >= x {locate(x, s1(u)} else
{if key(max(s2(u) >= x >= {locate(x,s2)}  else {locate(x, s3(u))}}}
```

2-3-trees T: addson(v,u) and insert(x)



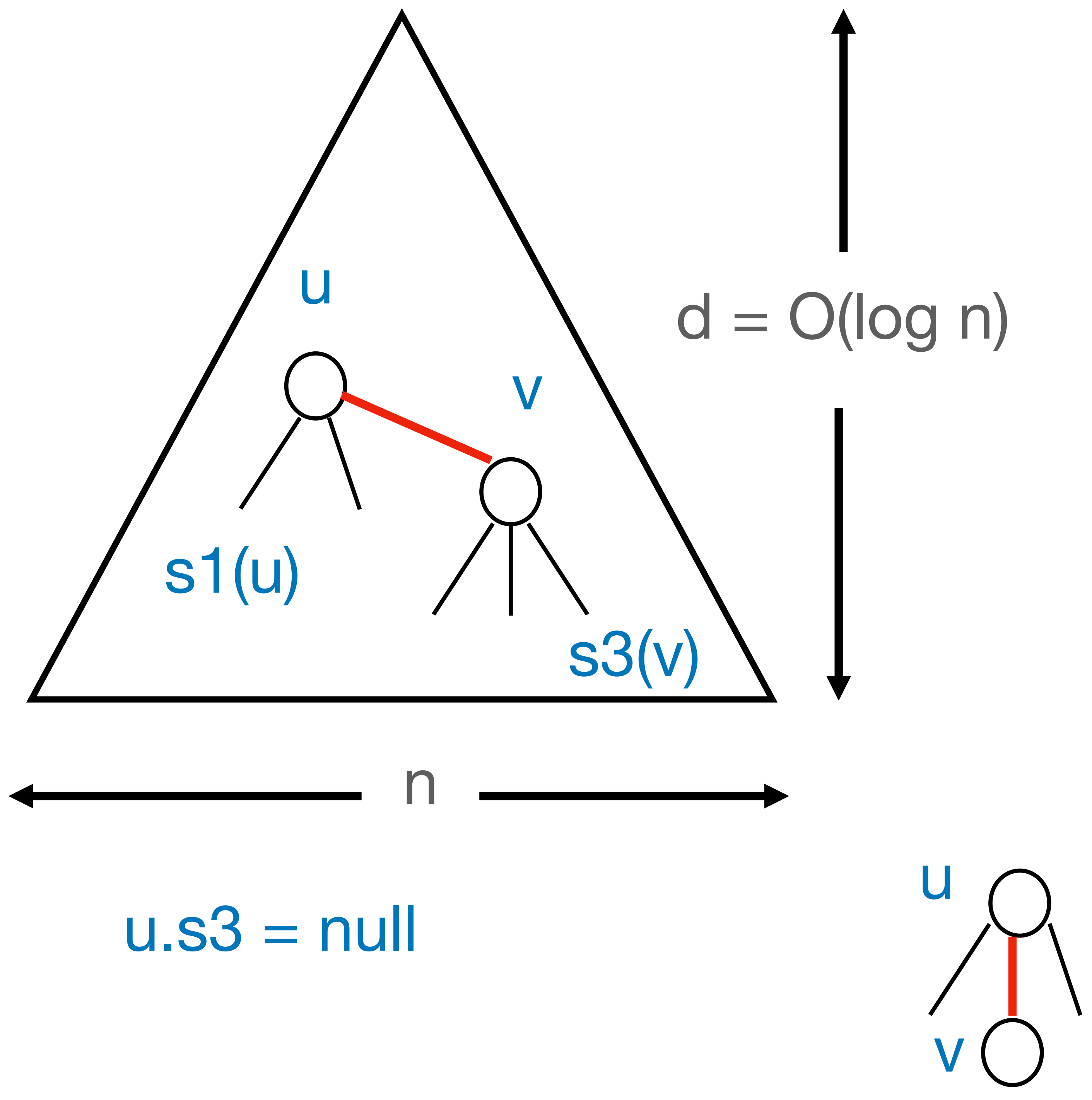
u.s3 = null

$addson(v,u)$: makes node v son of node u and rebalances tree.
$insert(x)$: adds x to S

$$S' = S \cup \{x\}$$

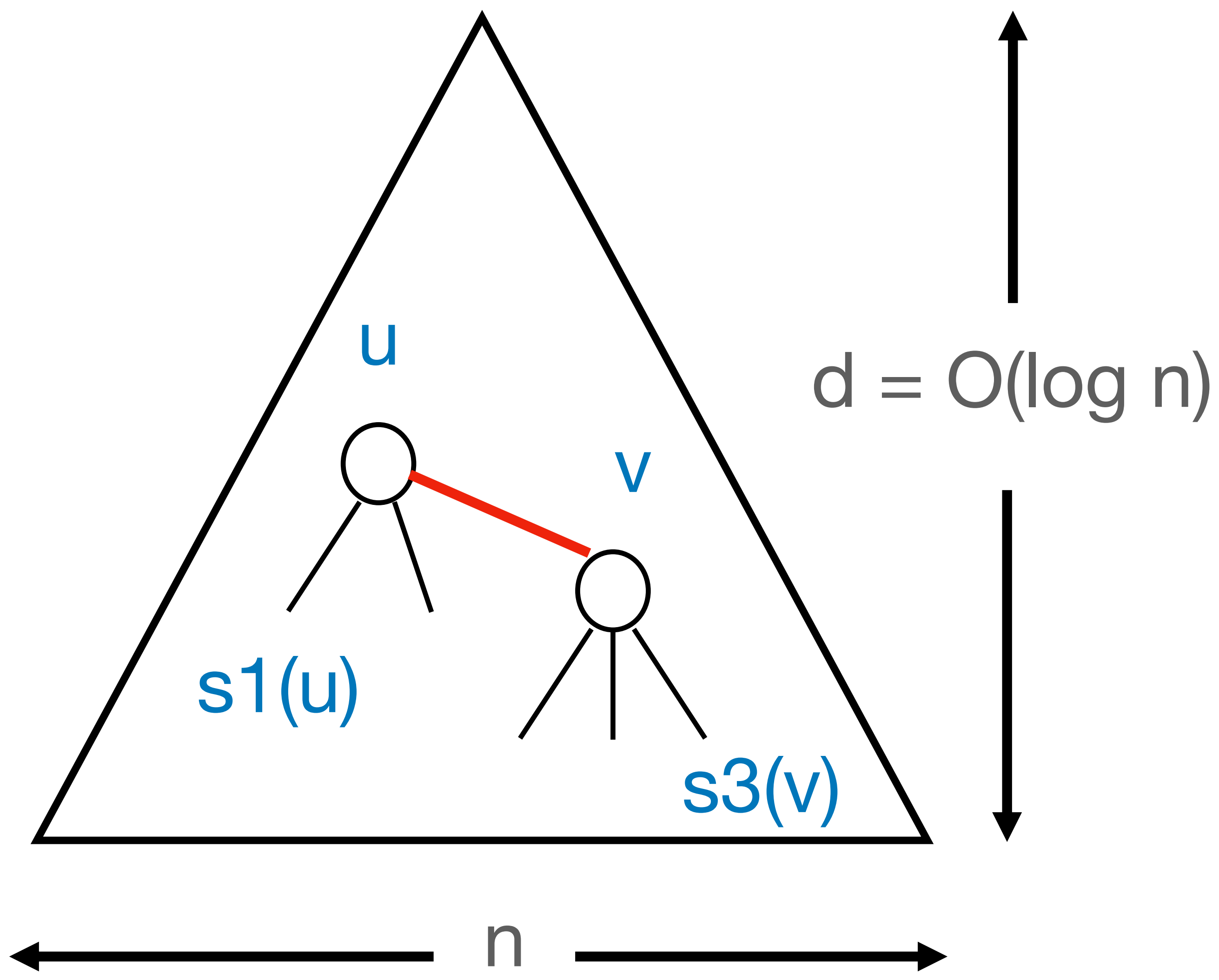
```
w= locate(x, root);
u=p(w)
create new leaf v;
key(v) = max(v) = x;
addson(v, u)
```

2-3-trees T: addson(v,u)

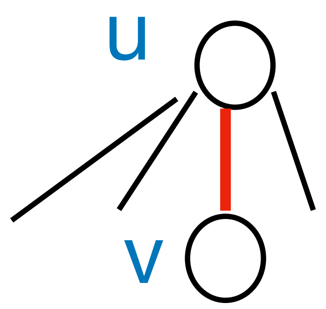



```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

2-3-trees T: addson(v,u)

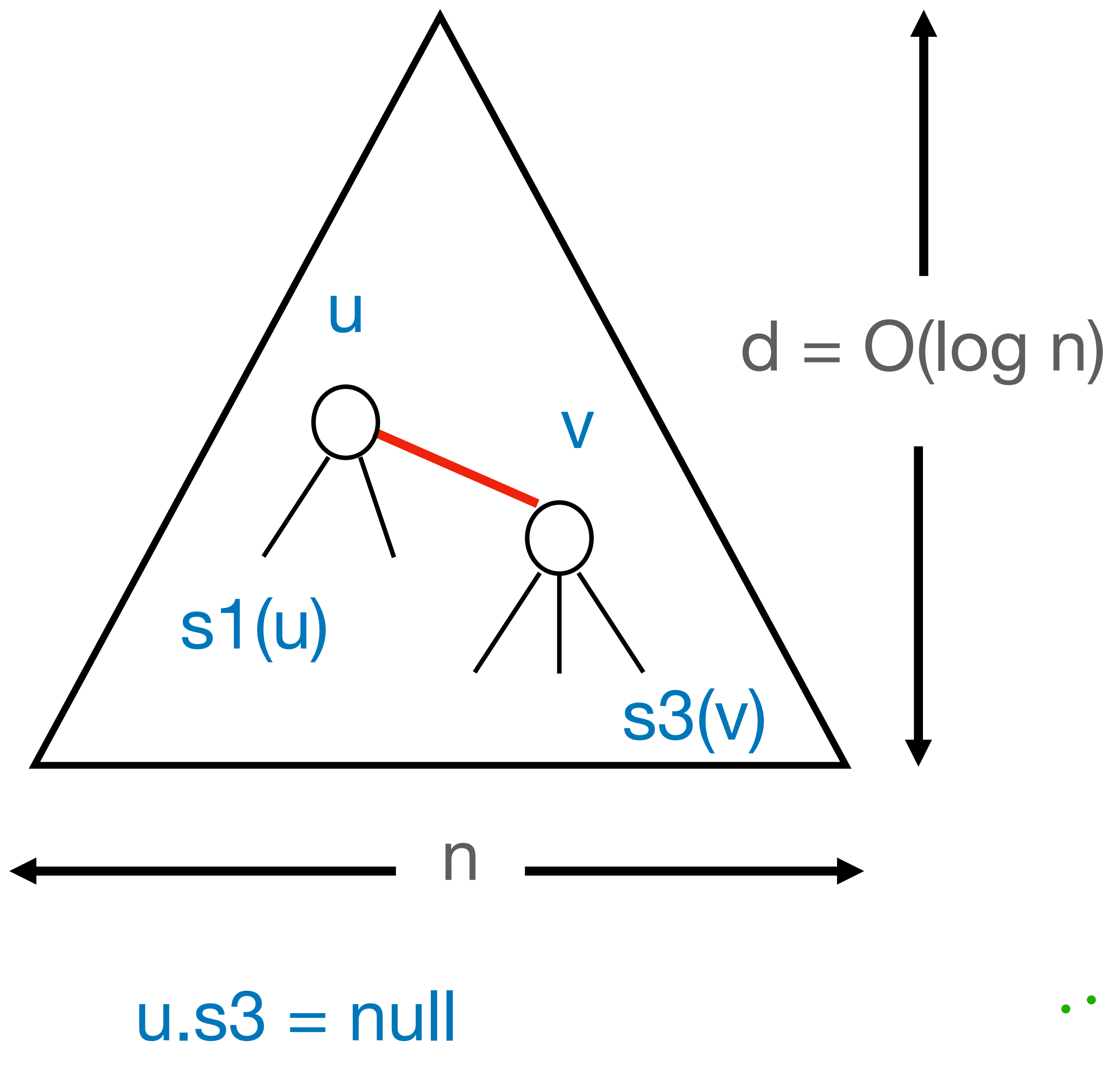


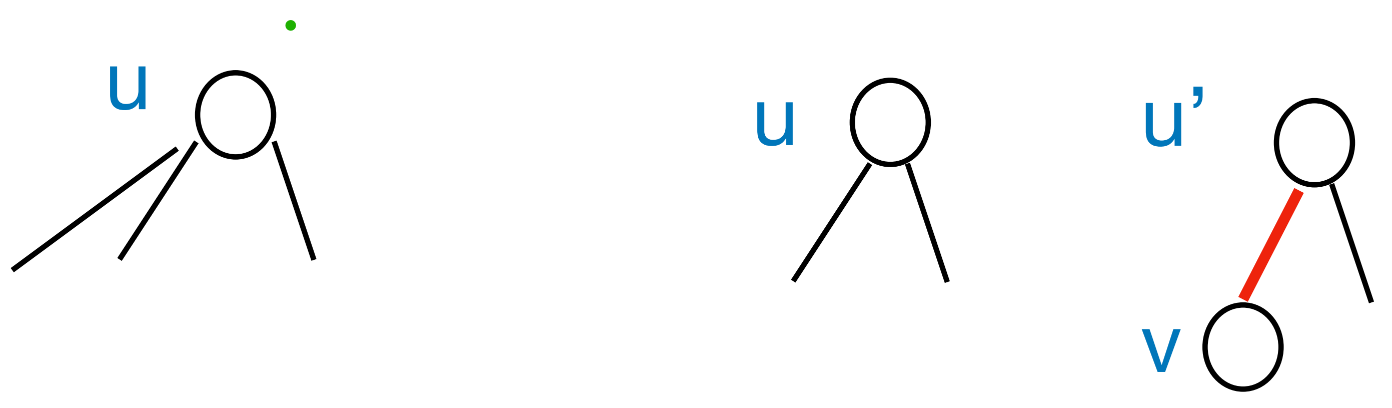
u.s3 = null




```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

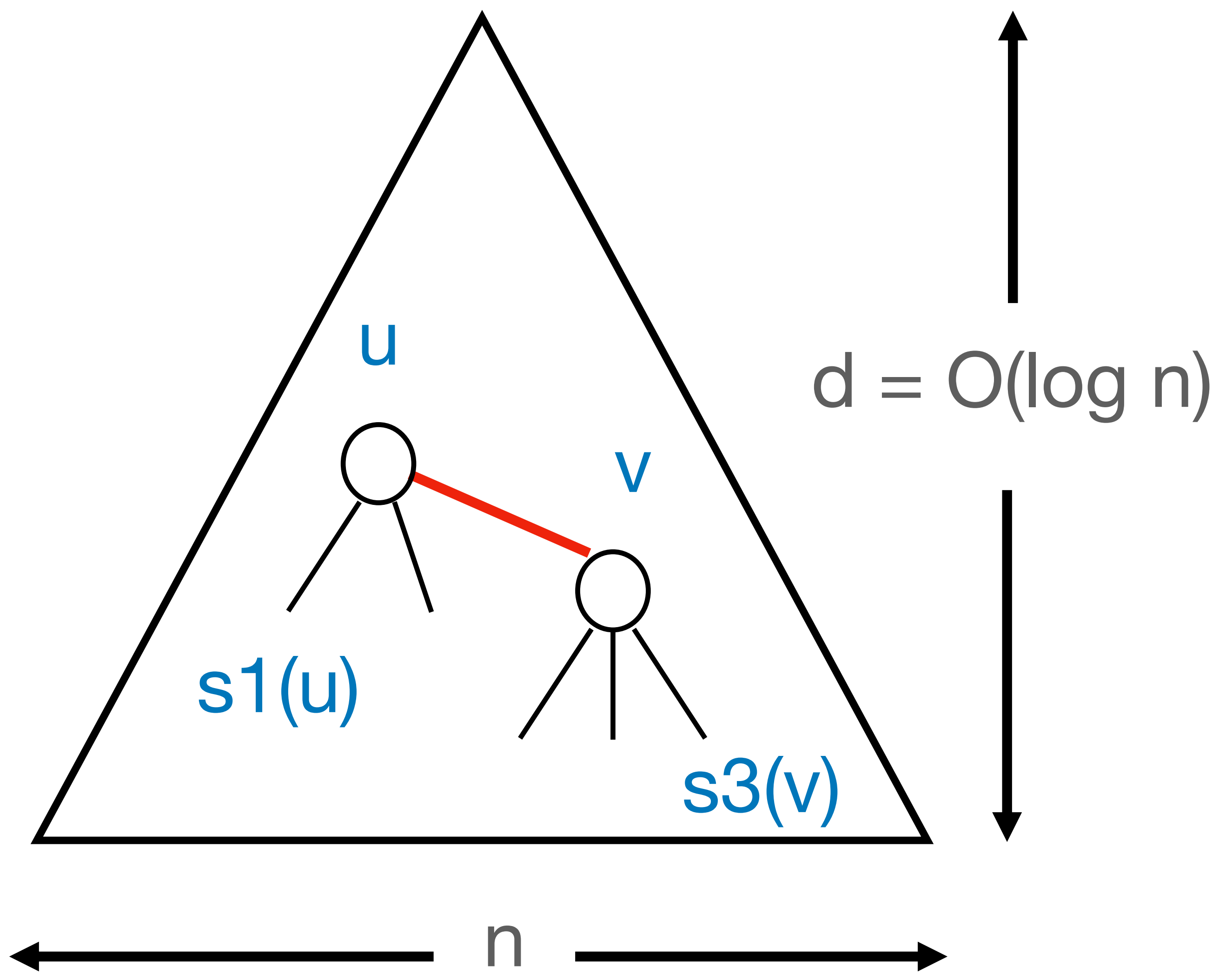
2-3-trees T: addson(v,u)



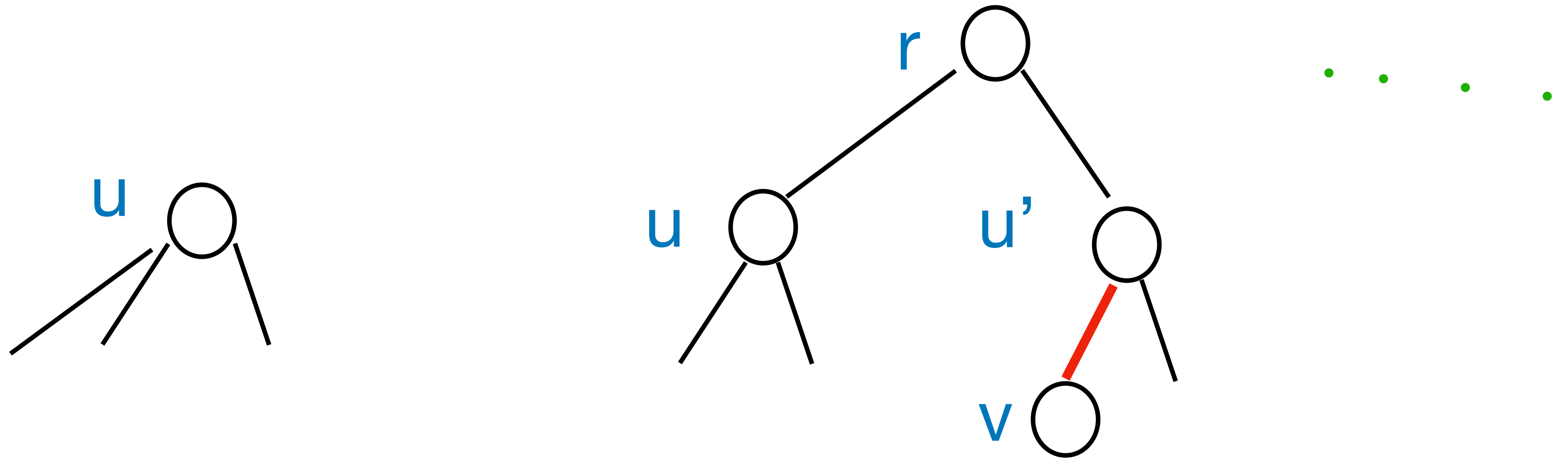



```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

2-3-trees T: addson(v,u)

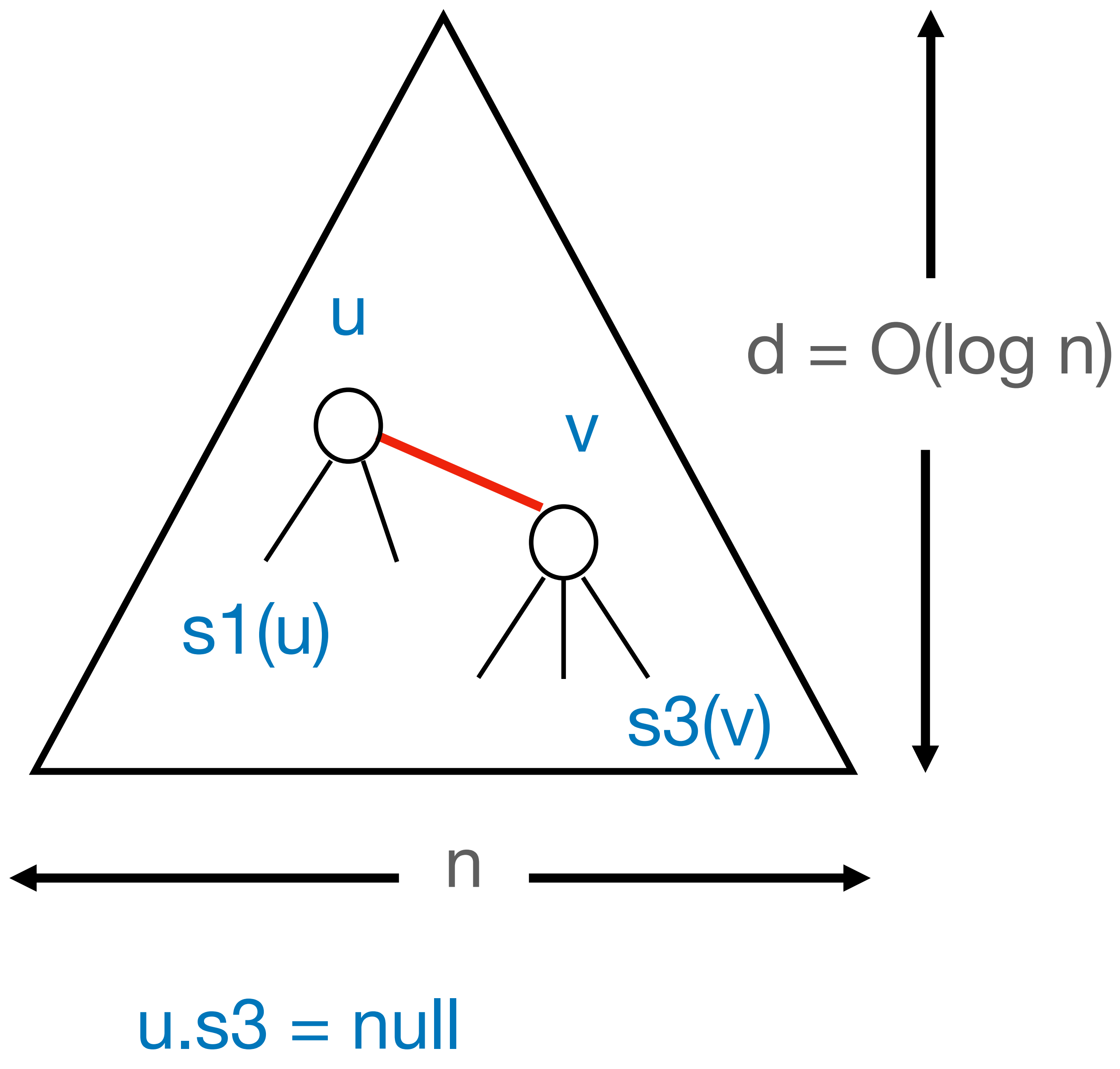


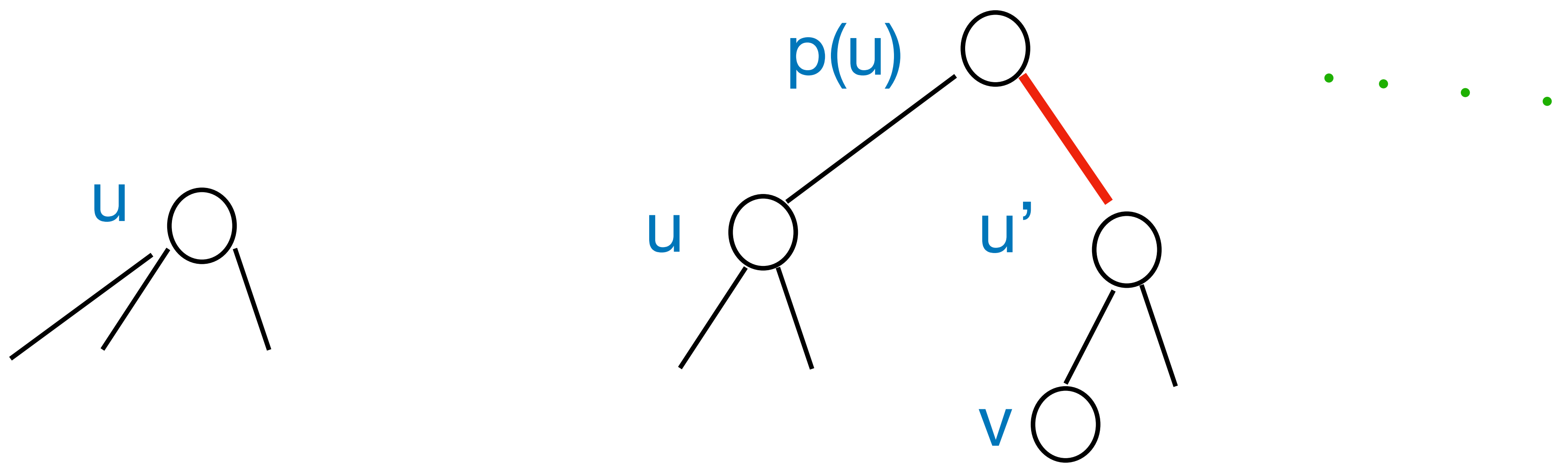
u.s3 = null




```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

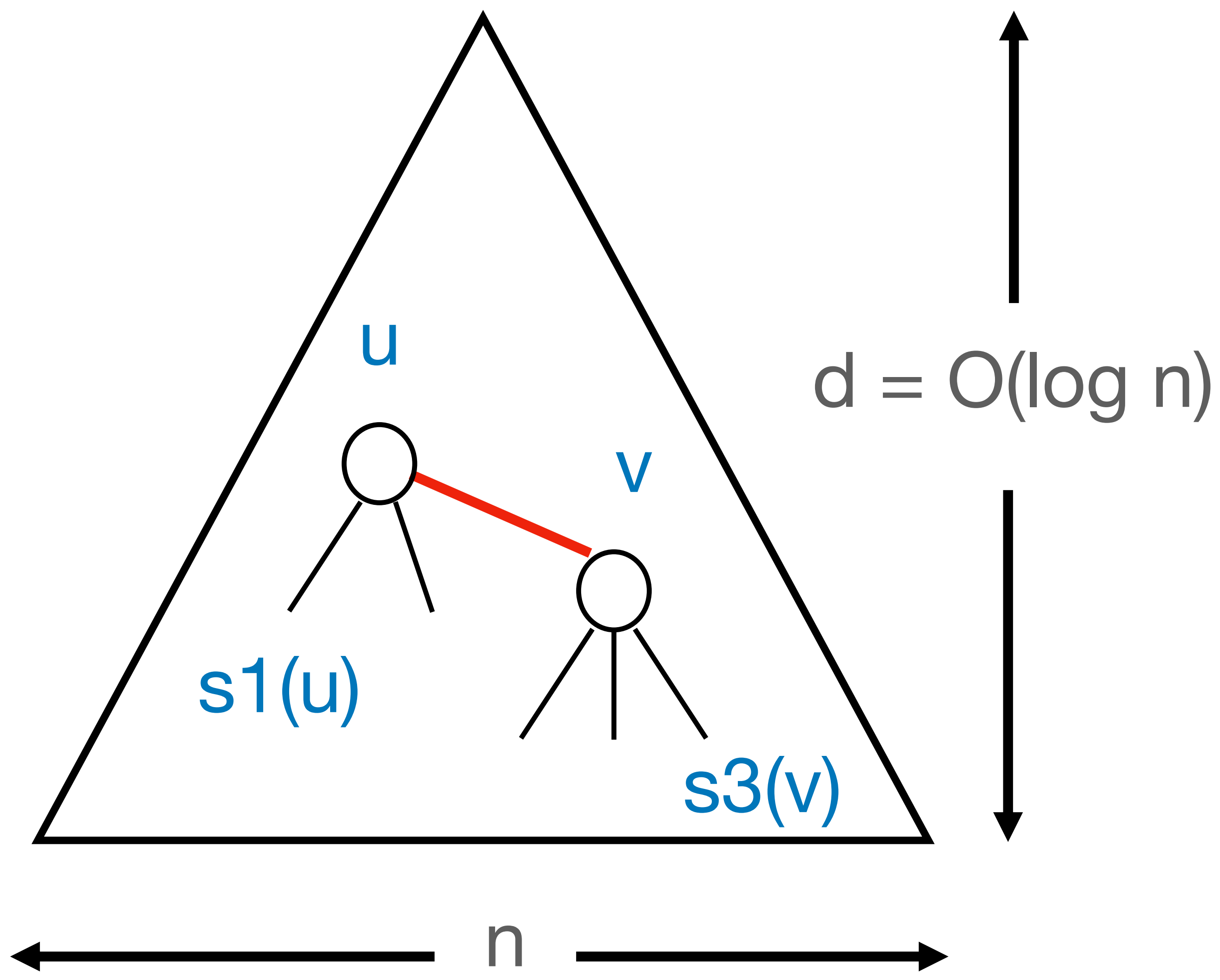
2-3-trees T: addson(v,u)



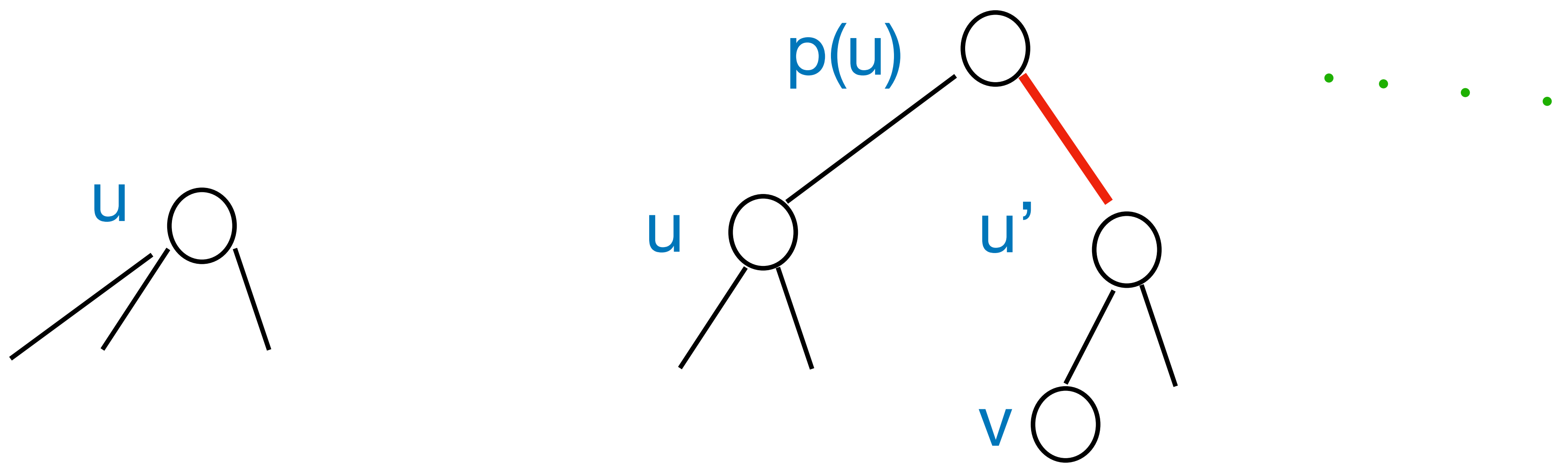



```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

2-3-trees T: addson(v,u)

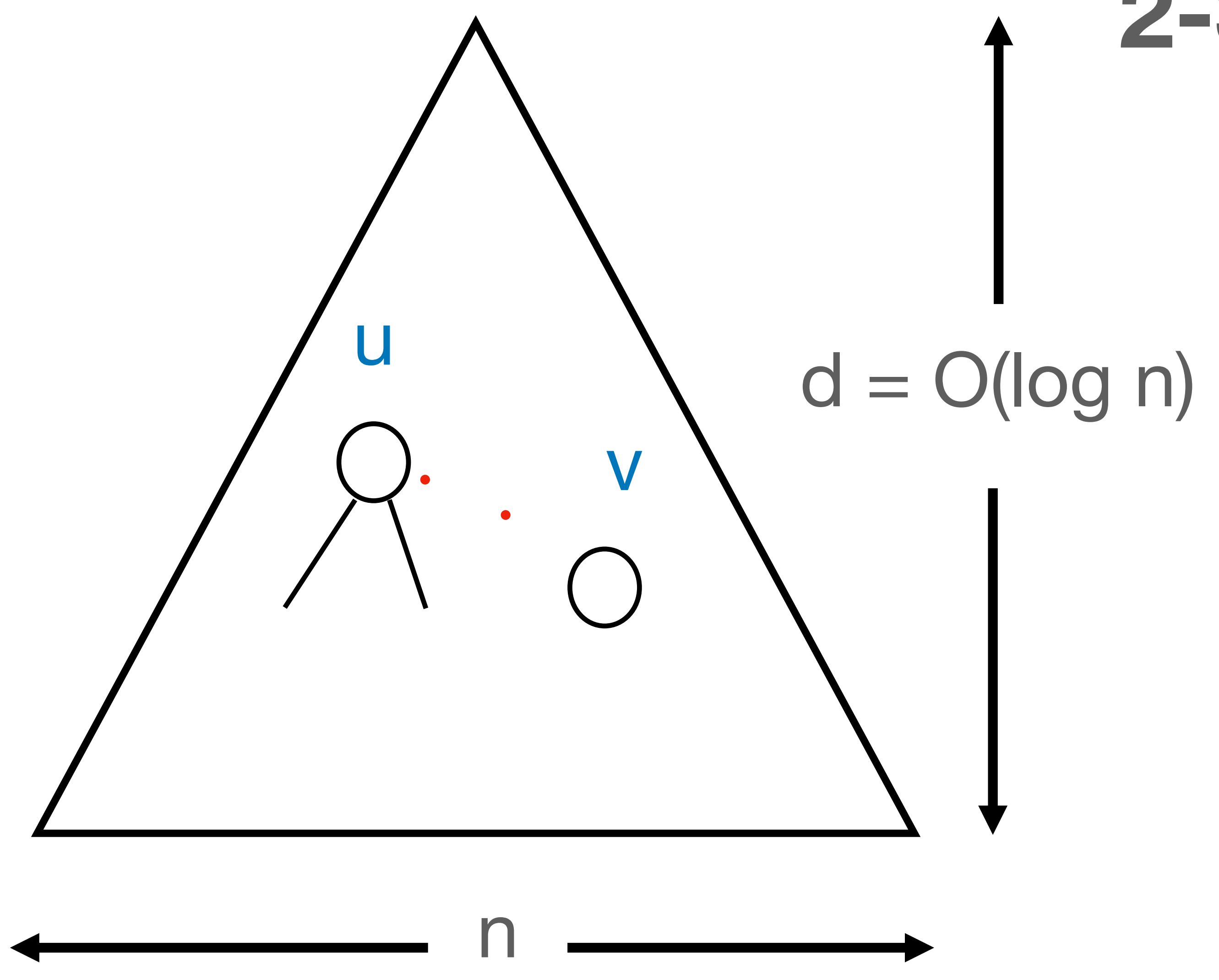


u.s3 = null




```
1. u has 2 sons:

make v son at appropriate place; /*case split*/
max(u) = max{ max(u), max(v)}; done

2. u has 3 sons:

make v son of u at appropriate place
/* u has now 4 sons s1(u);...,s4(u)*/
create new node u'; make s3(u) and s4(u) sons of u':
compute max for u and u';

2a. u was root:

create new root r with sons u and u'; max(r) = max{max(u), max(v))}

2b: u hat parent p(u):

addson(u', p(u))
```

run time O(log n)



2-3-trees T: deleteson(v,u) and delete(x)

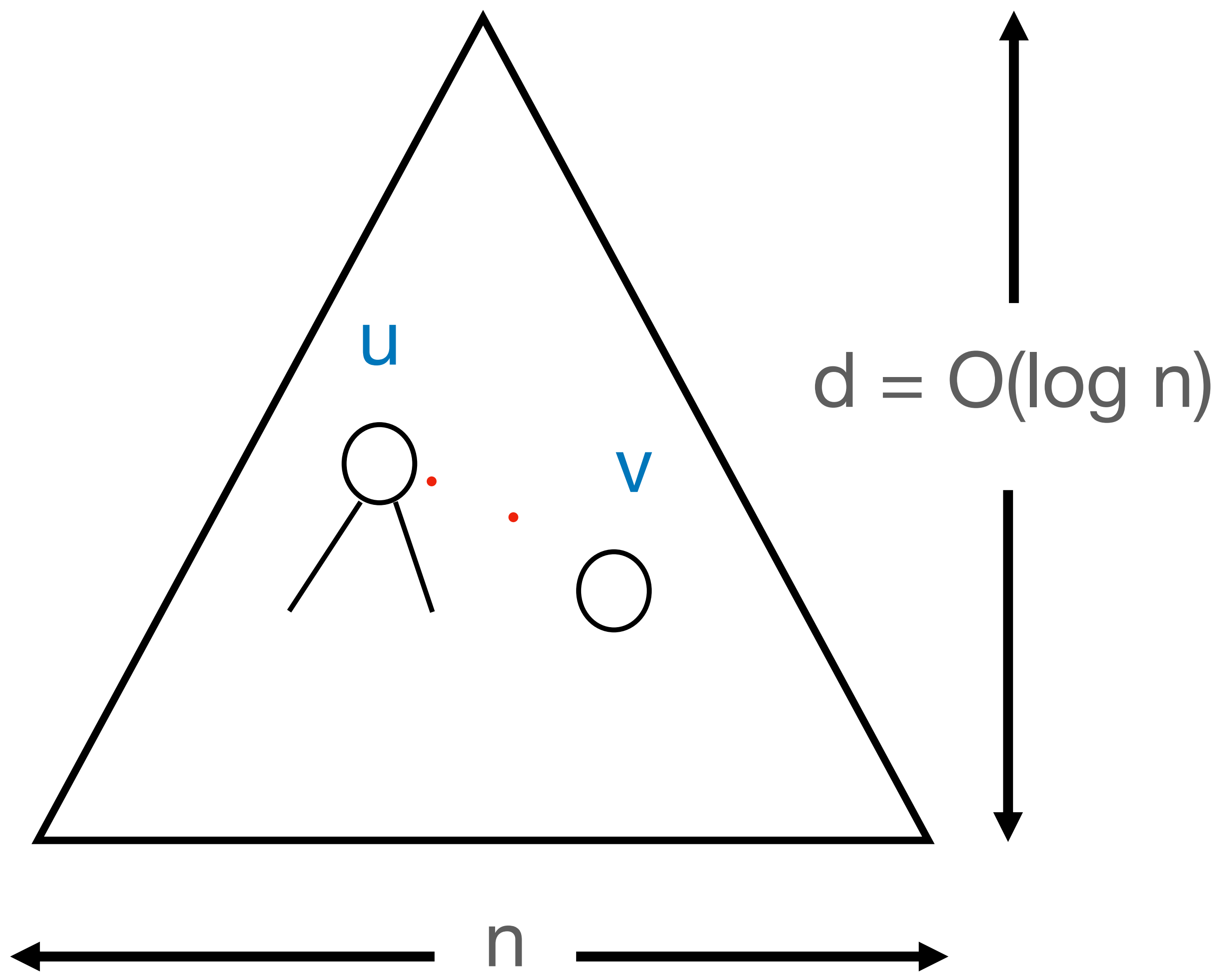
$deleteson(v,u)$: deletes son v from parent $u = p(v)$ and rebalances tree.

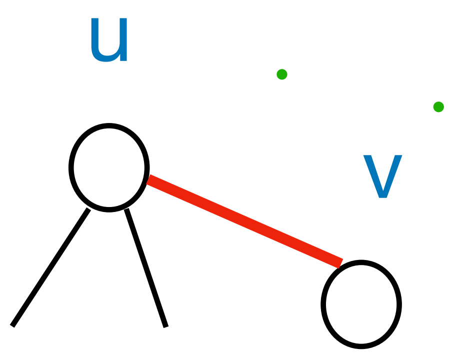
$delete(x)$: deletes x from S

$$S' = S \setminus \{x\}$$

```
v= locate(x,root); deleteson(v, p(v))
```

2-3-trees T: deleteson(v,u)

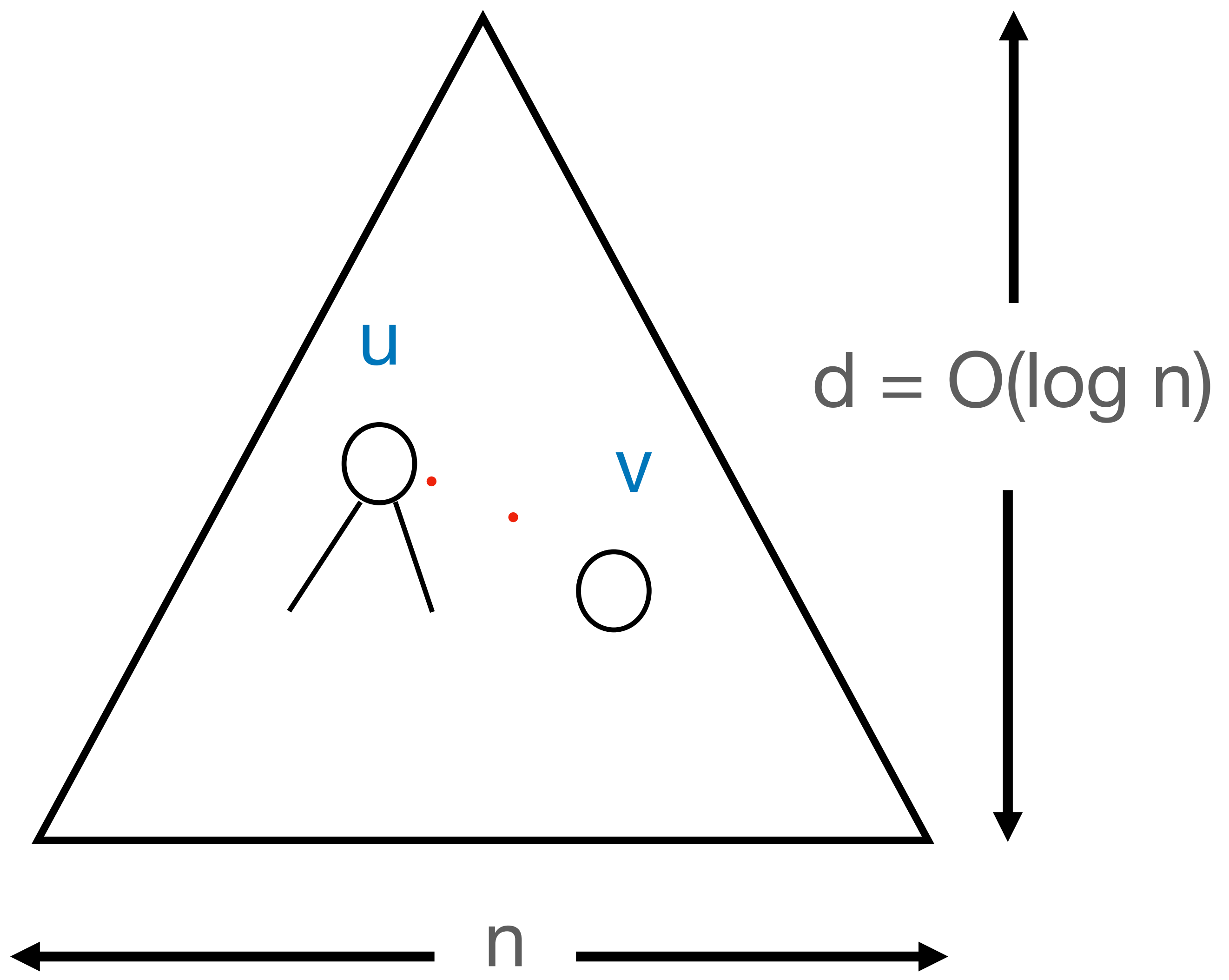


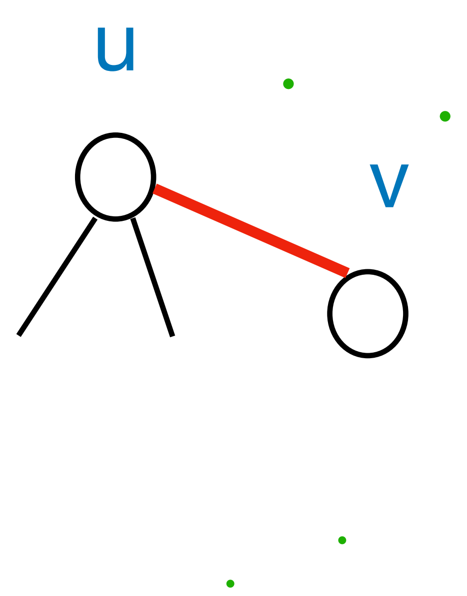


def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

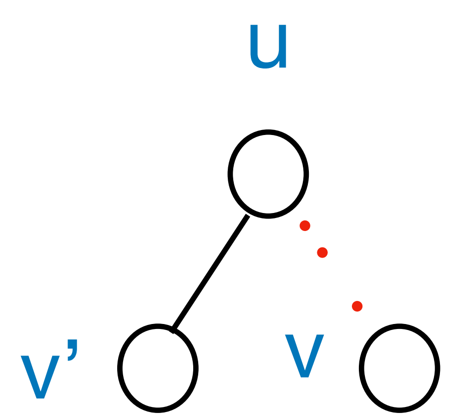
```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v′= brother(v)

2.a u is root

    delete u and v; v′ is new root

2.b u has brother u′ with 3 sons

delete v and move 1 son of u′ to u; done

2.c u has brother u′ with 2 sons

make v′ son of u′;
deleteson(u; p(u))
```

2-3-trees T: deleteson(v,u)



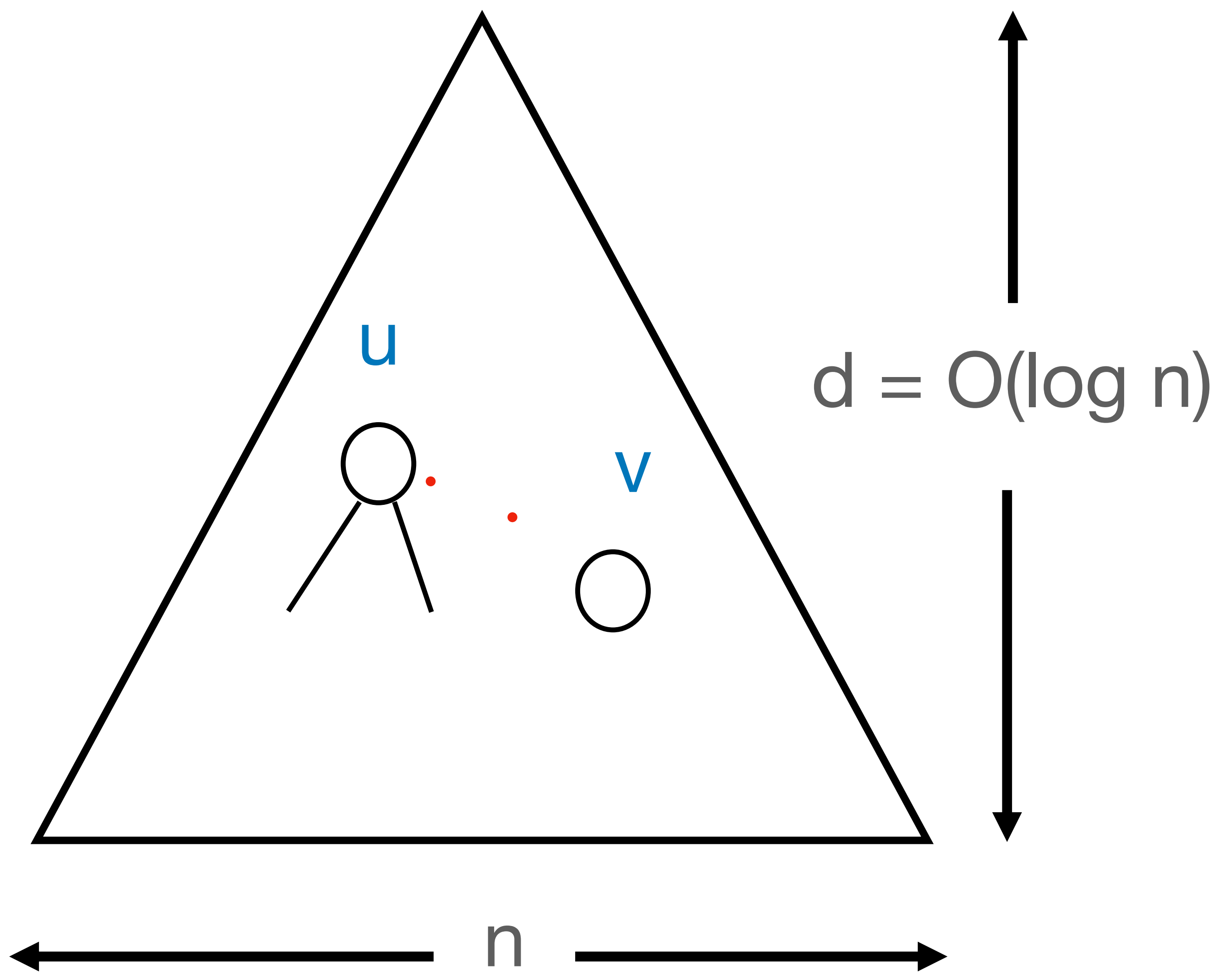


def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

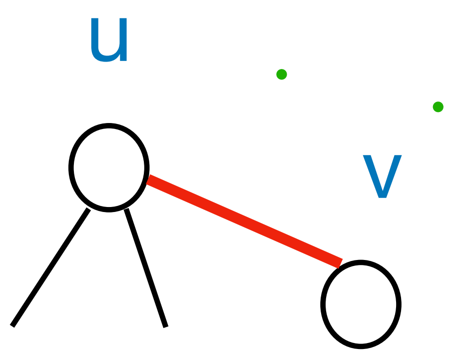



```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v′= brother(v)

2.a u is root

    delete u and v; v′ is new root

2.b u has brother u′ with 3 sons

delete v and move 1 son of u′ to u; done

2.c u has brother u′ with 2 sons

make v′ son of u′;
deleteson(u; p(u))
```

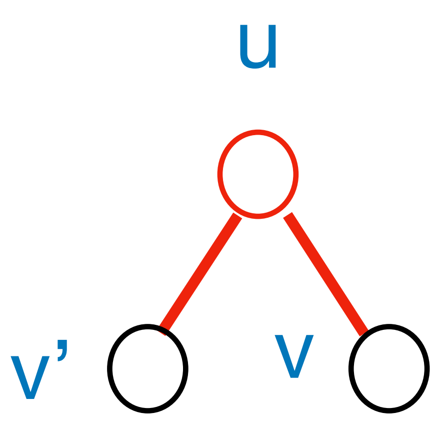
2-3-trees T: deleteson(v,u)



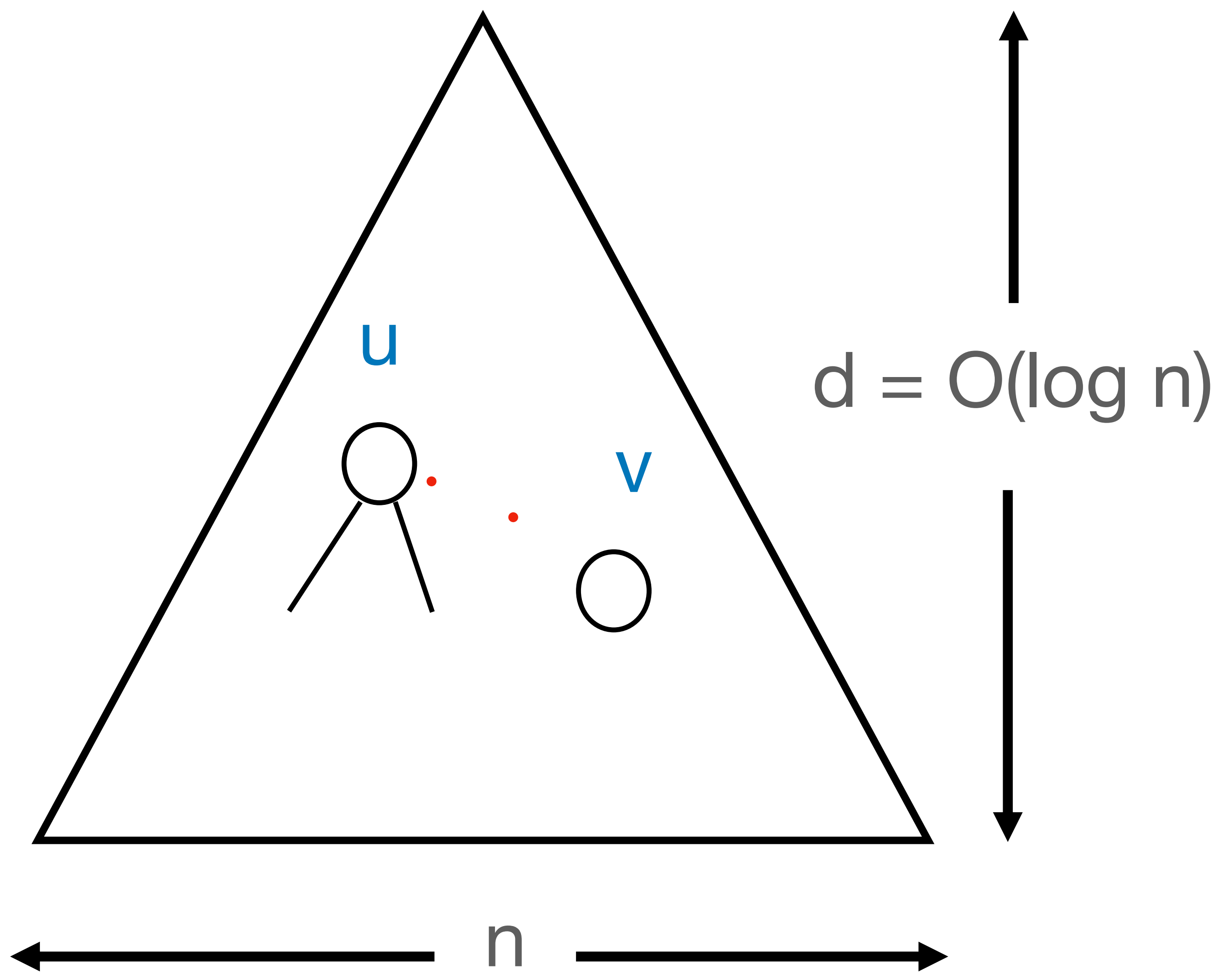
def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$




```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v′= brother(v)

2.a u is root

    delete u and v; v′ is new root

2.b u has brother u′ with 3 sons

delete v and move 1 son of u′ to u; done

2.c u has brother u′ with 2 sons

make v′ son of u′;
deleteson(u; p(u))
```

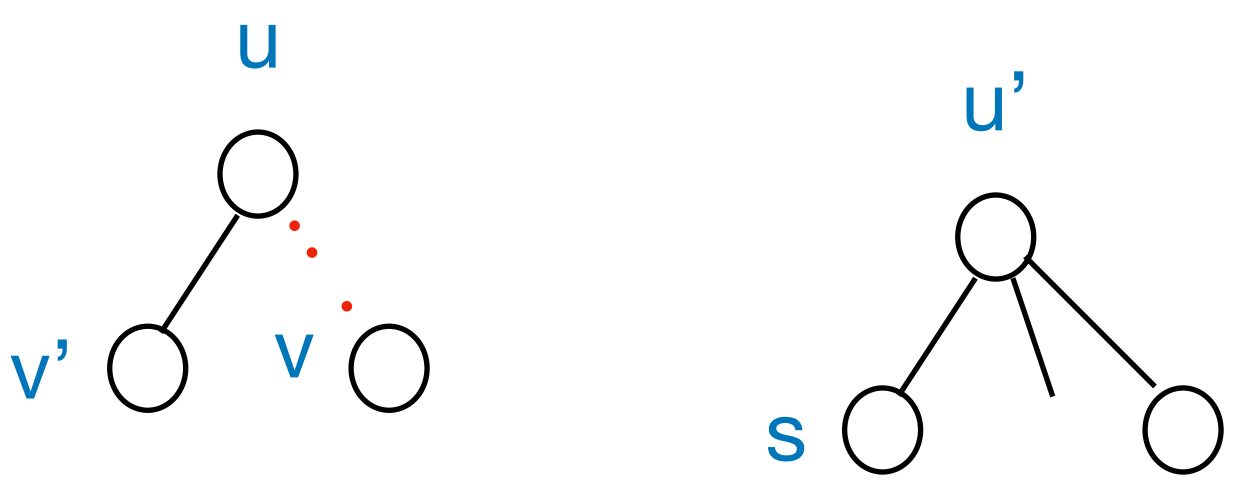


2-3-trees T: deleteson(v,u)

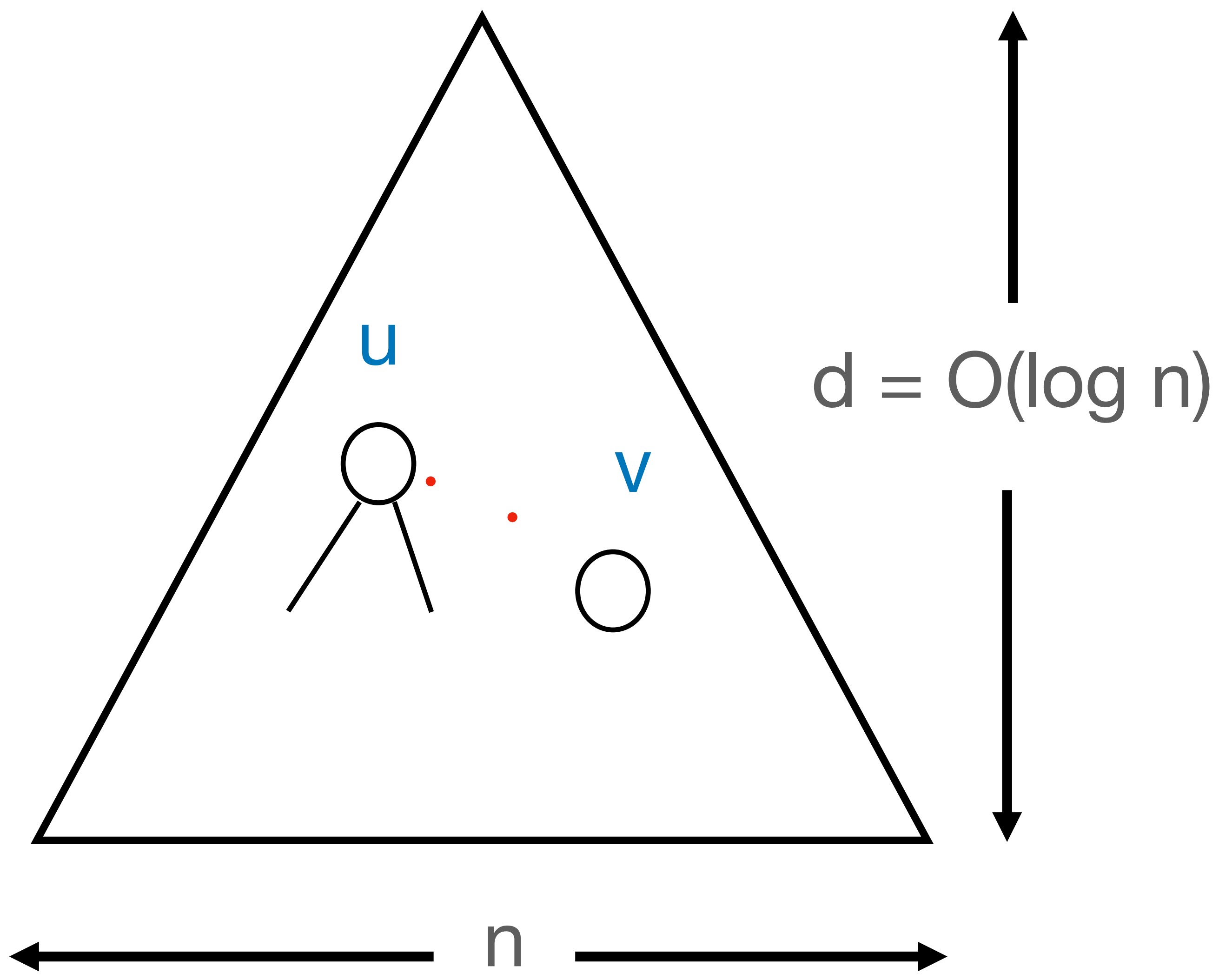


def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

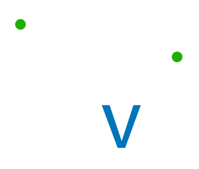
```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v'= brother(v)

2.a u is root

    delete u and v; v' is new root

2.b u has brother u' with 3 sons

delete v and move 1 son of u' to u; done

2.c u has brother u' with 2 sons

make v' son of u';
deleteson(u; p(u))
```



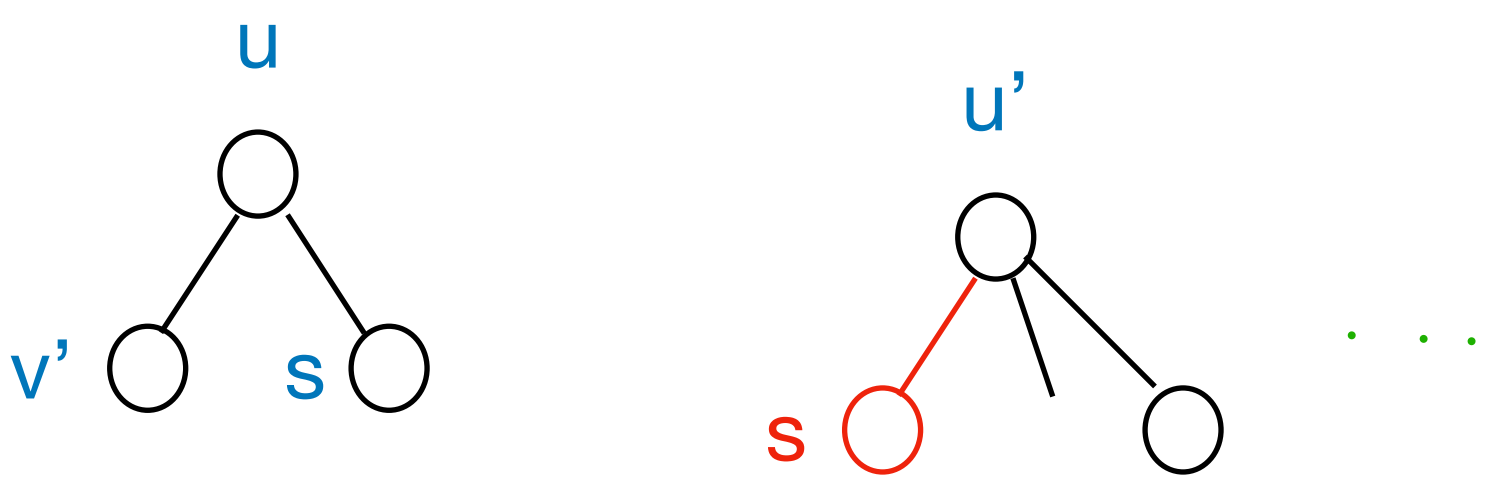
2-3-trees T: deleteson(v,u)



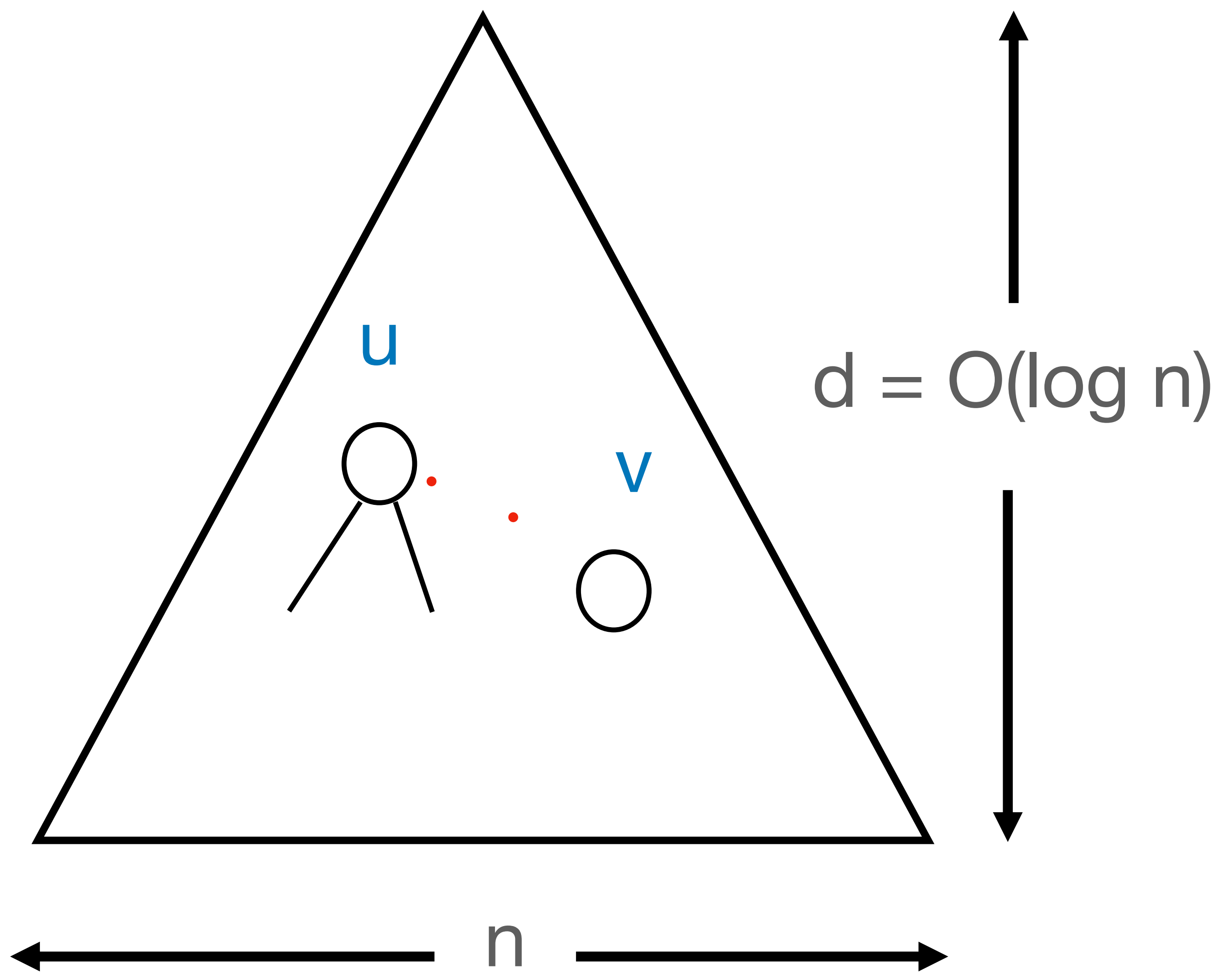
def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$




```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v'= brother(v)

2.a u is root

    delete u and v; v' is new root

2.b u has brother u' with 3 sons

delete v and move 1 son of u' to u; done

2.c u has brother u' with 2 sons

make v' son of u';
deleteson(u; p(u))
```



2-3-trees T: deleteson(v,u)

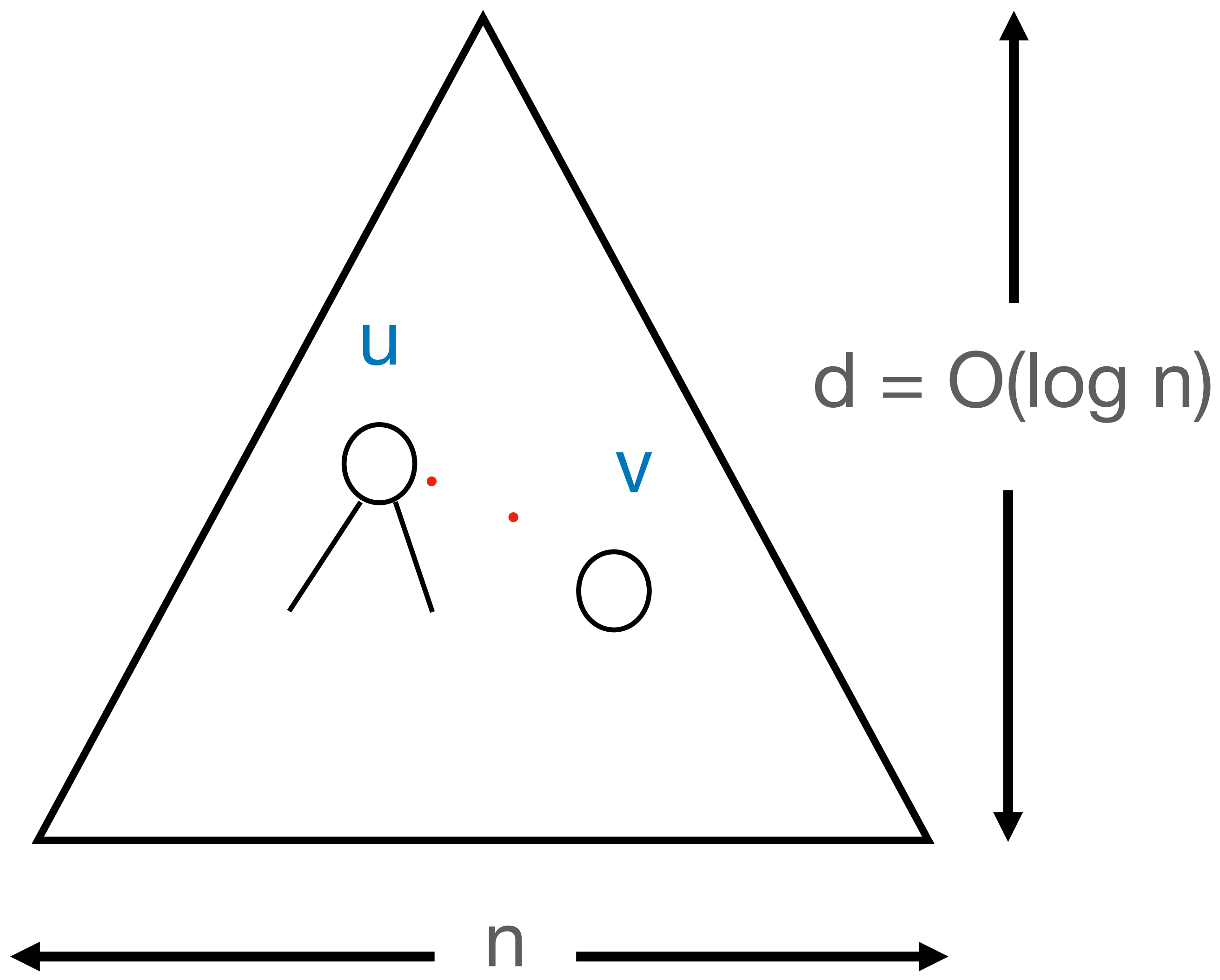


def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

v


```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v′= brother(v)

2.a u is root

    delete u and v; v′ is new root

2.b u has brother u′ with 3 sons

delete v and move 1 son of u′ to u; done

2.c u has brother u′ with 2 sons

make v′ son of u′;
deleteson(u; p(u))
```

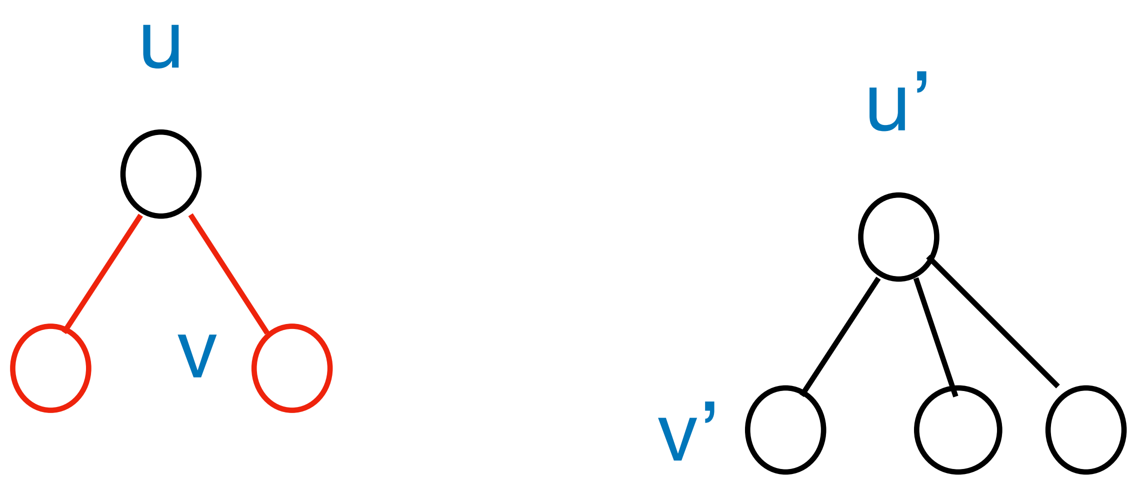
2-3-trees T: deleteson(v,u)



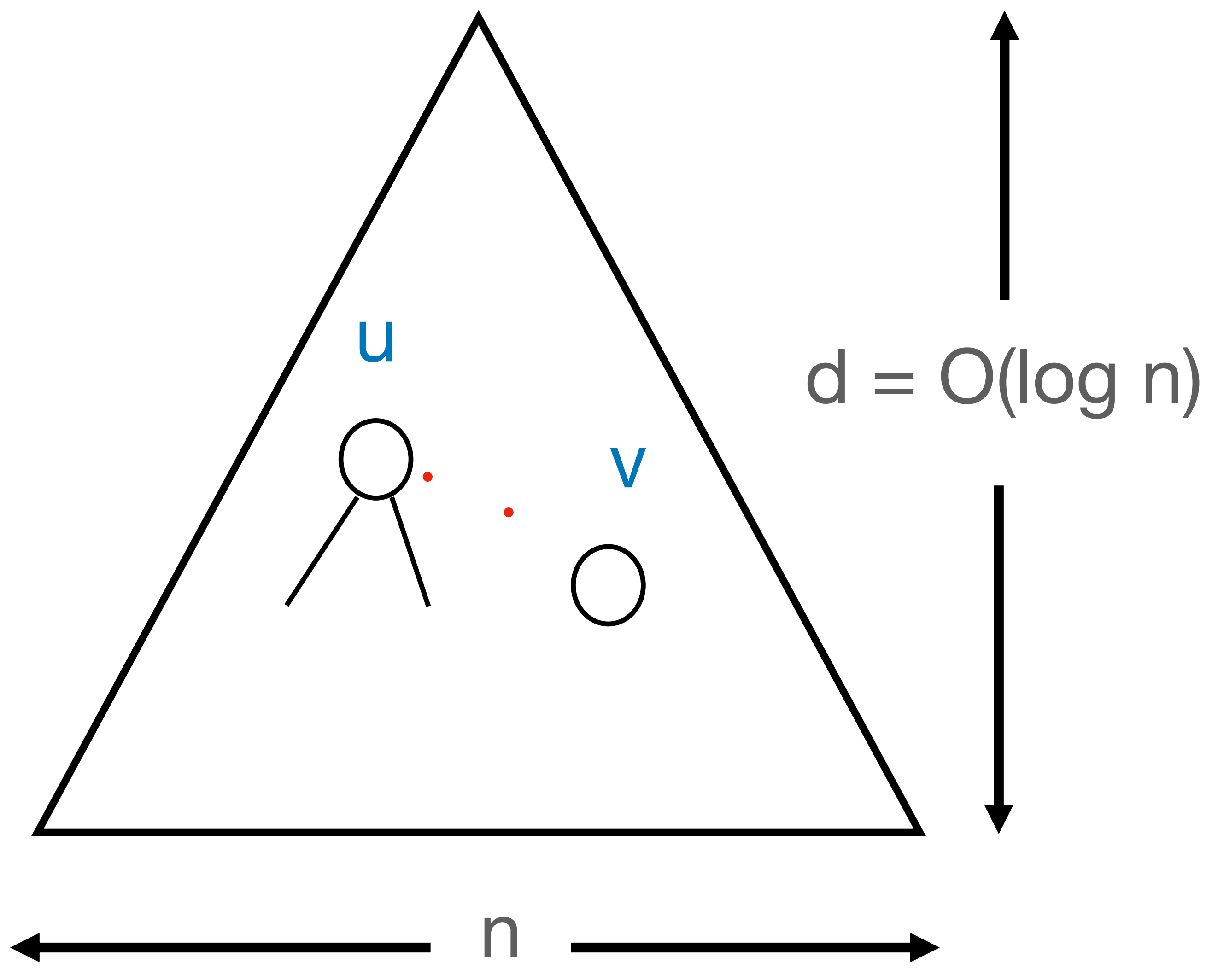
def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

v

```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v′= brother(v)

2.a u is root

    delete u and v; v′ is new root

2.b u has brother u′ with 3 sons

delete v and move 1 son of u′ to u; done

2.c u has brother u′ with 2 sons

make v′ son of u′;
deleteson(u; p(u))
```



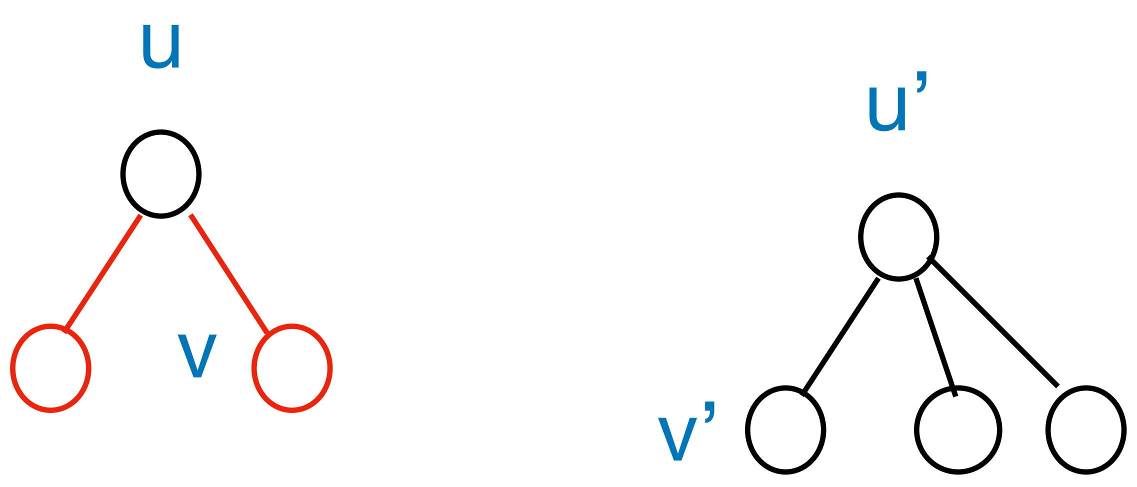
2-3-trees T: deleteson(v,u)



def: u and u' are *brothers* if $p(u) = p(u')$ and $u \neq u'$

v

```
1. u has 3 sons
delete v; done

2. u has 2 sons

let v'= brother(v)

2.a u is root

    delete u and v; v' is new root

2.b u has brother u' with 3 sons

delete v and move 1 son of u' to u; done

2.c u has brother u' with 2 sons

make v' son of u';
deleteson(u; p(u))
```



run time O(log n)