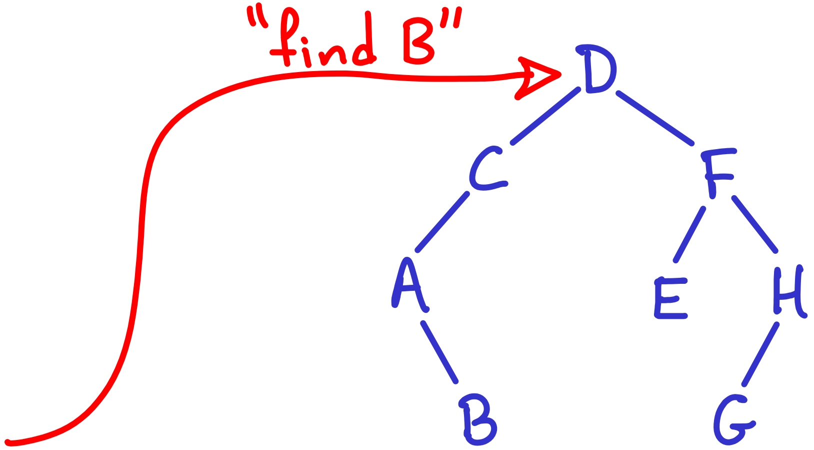
GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

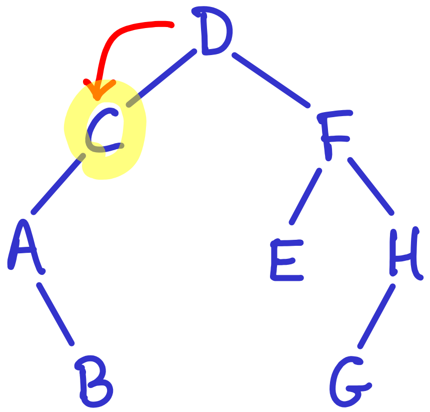


Model:

- every operation starts at the root

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

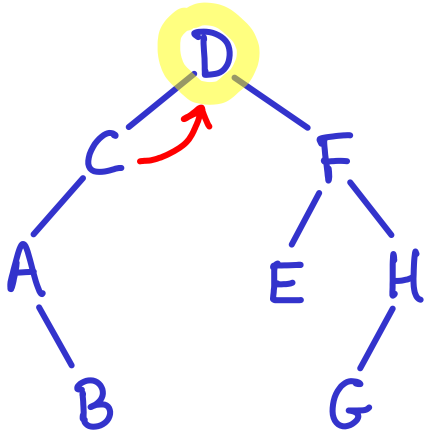


Model:

- every operation starts at the root
- at each step we may move between parent ↔ children

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

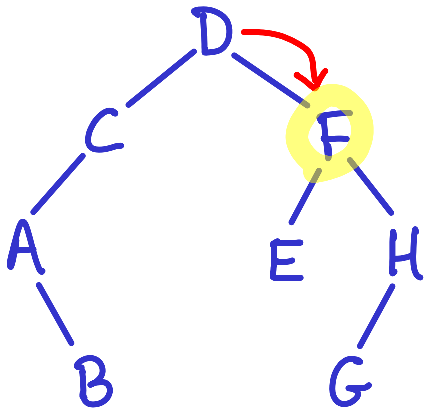


Model:

- every operation starts at the root
- at each step we may move between parent $\leftrightarrow$ children

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

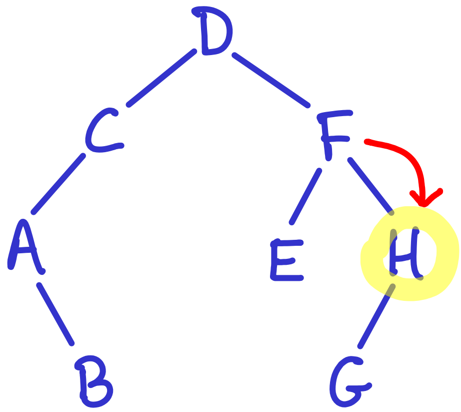


Model:

- every operation starts at the root
- at each step we may move between parent ↔ children

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

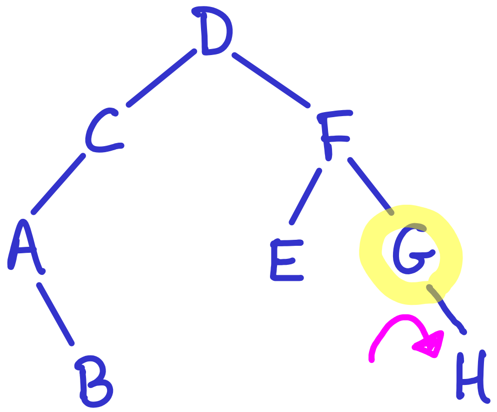
AND BST OPTIMALITY



Model:

- every operation starts at the root
- at each step we may move between parent $\leftrightarrow$ children

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS) AND BST OPTIMALITY

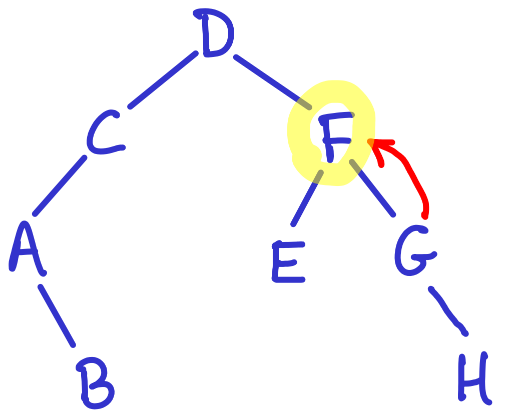


Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

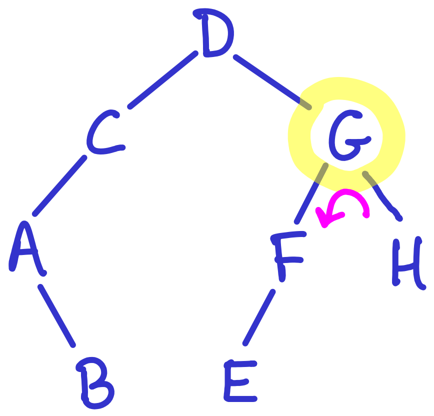
AND BST OPTIMALITY



Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS) AND BST OPTIMALITY

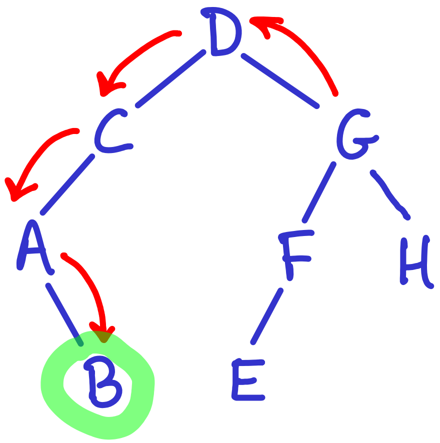


Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

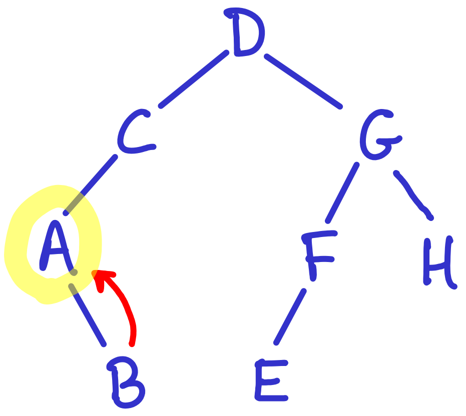


Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position
- at some point during operation we must access (find/insert/delete) given target

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

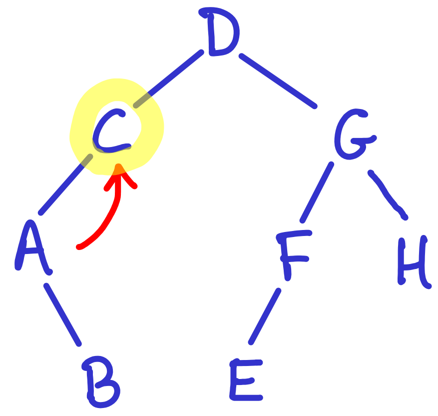


Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position
- at some point during operation we must access (find/insert/delete) given target

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY

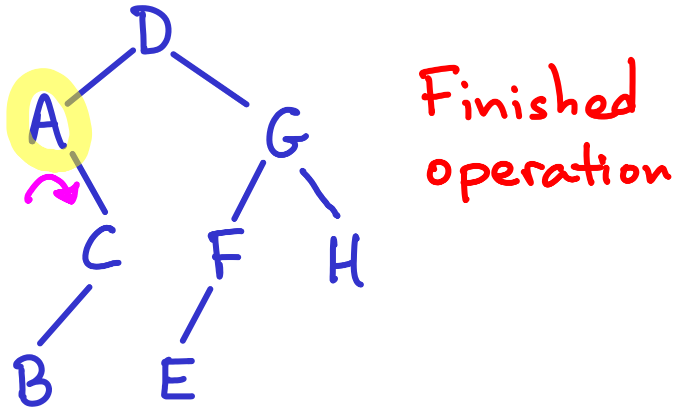


Model:

- every operation starts at the root
- at each step we may
  - move between parent $\leftrightarrow$ children
  - perform a rotation at current position
- at some point during operation we must access (find/insert/delete) given target

GEOMETRY OF BINARY SEARCH TREES (& ALGORITHMS)

AND BST OPTIMALITY



Model:

- every operation starts at the root
- at each step we may
  - move between parent ↔ children
  - perform a rotation at current position
- at some point during operation we must access (find/insert/delete) given target

Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5)

t_3 : insert(1)

t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)

Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5)

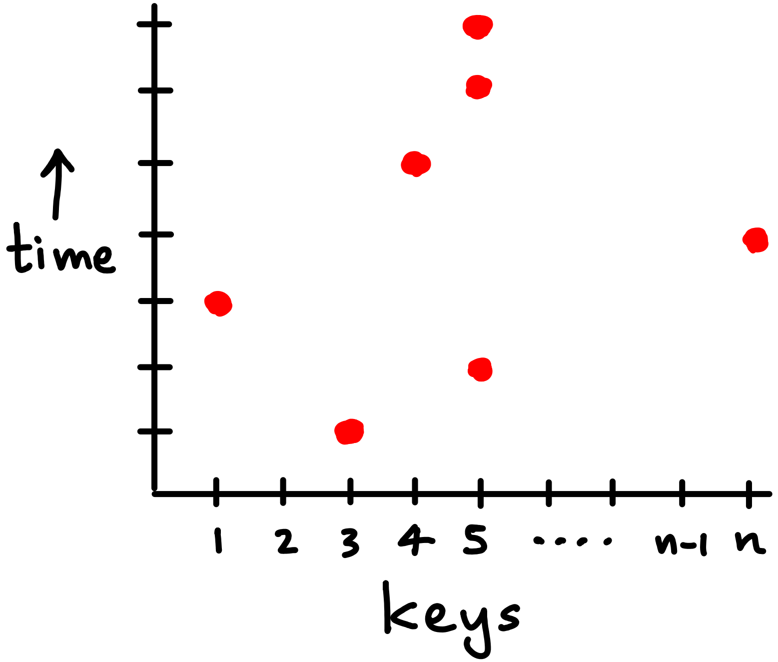
t_3 : insert(1)

t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)



Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5)

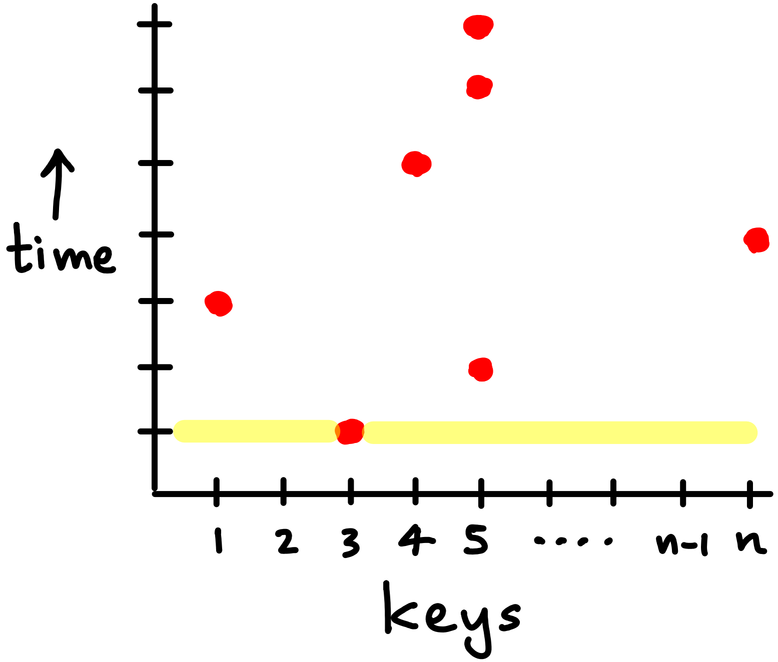
t_3 : insert(1)

t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)



Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5)

t_3 : insert(1)

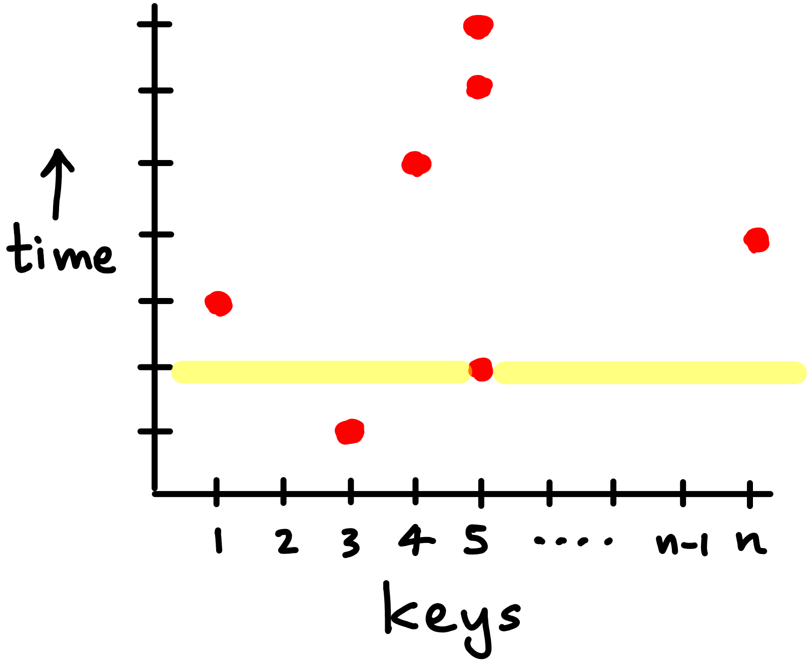
t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)

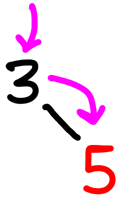
3



Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5) Must access 3 first



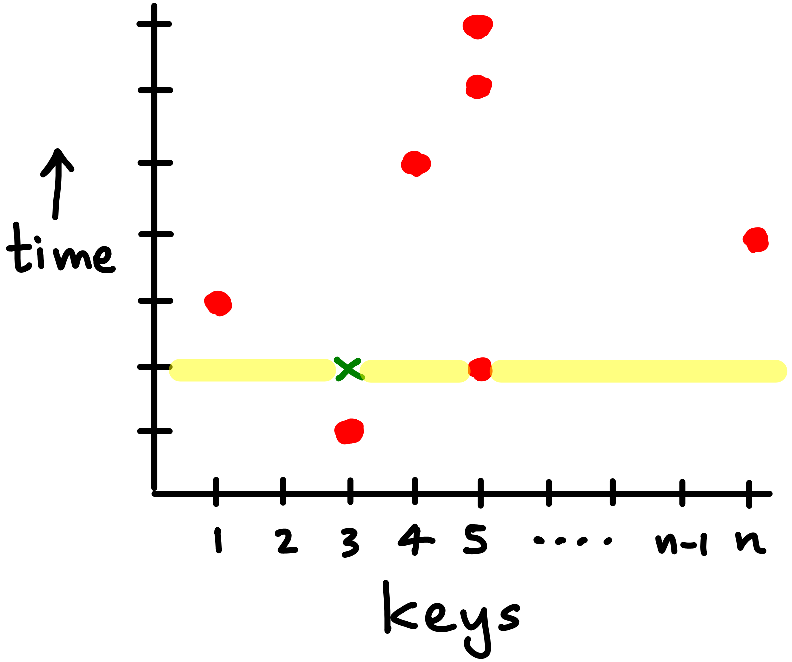
t_3 : insert(1)

t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

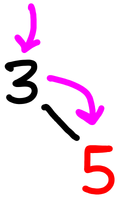
t_7 : delete(5)

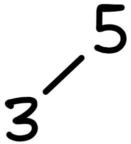


Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5) Must access 3 first





could rotate

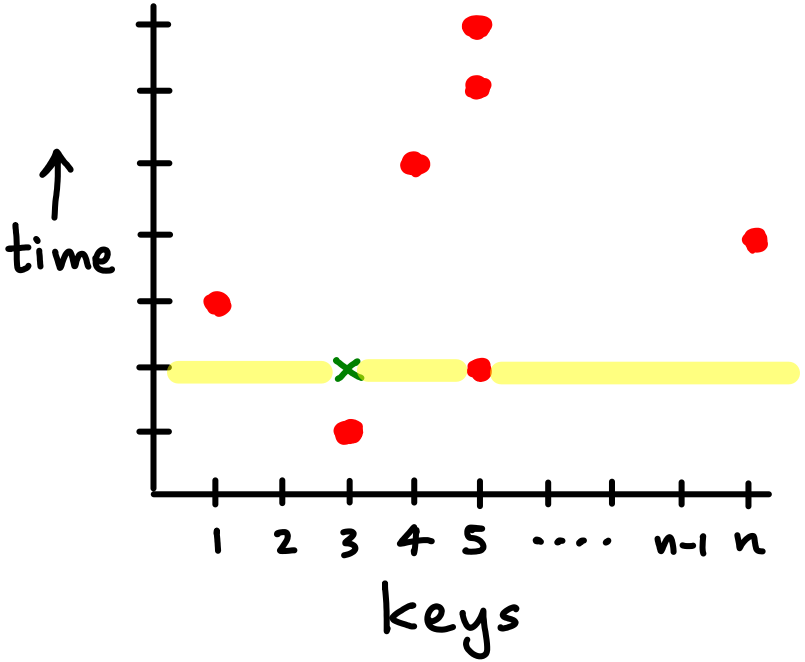
t_3 : insert(1)

t_4 : insert(n)

t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)



Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5) Must access 3 first

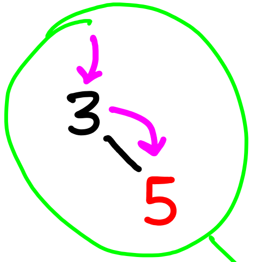
t_3 : insert(1) Must access 3 first

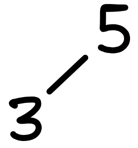
t_4 : insert(n)

t_5 : insert(4)

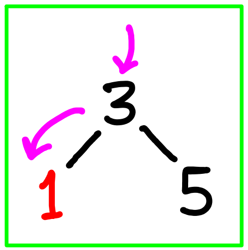
t_6 : search(5)

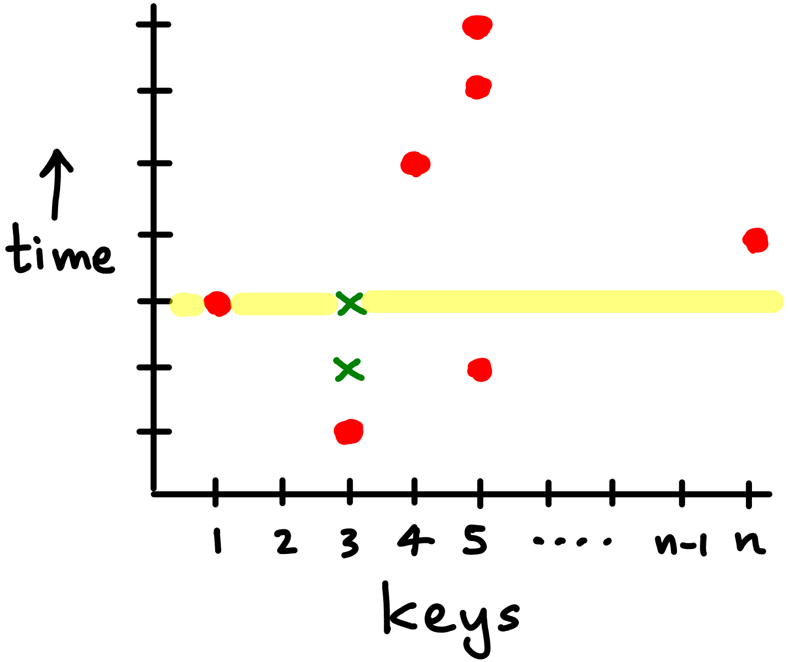
t_7 : delete(5)





could rotate





Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5) Must access 3 first

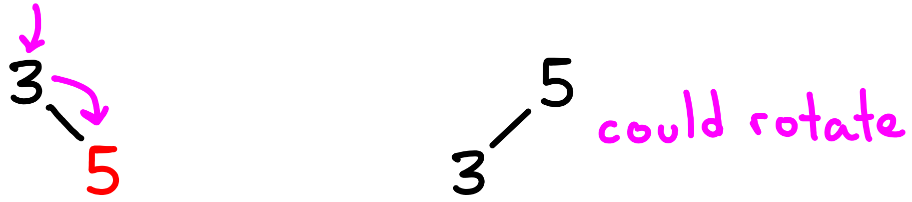
t_3 : insert(1) Must access 3 or 5 first (actually, if 5 then also 3)

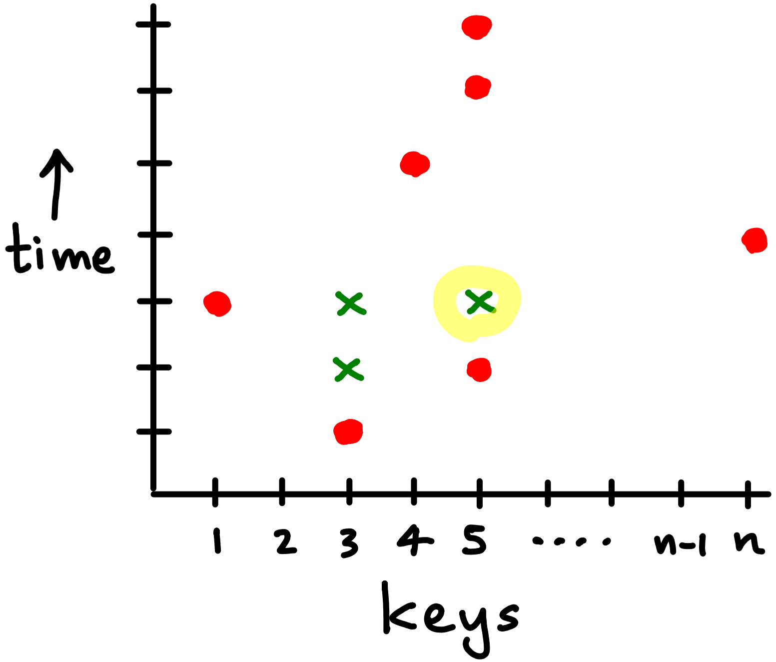
t_4 : insert(n)

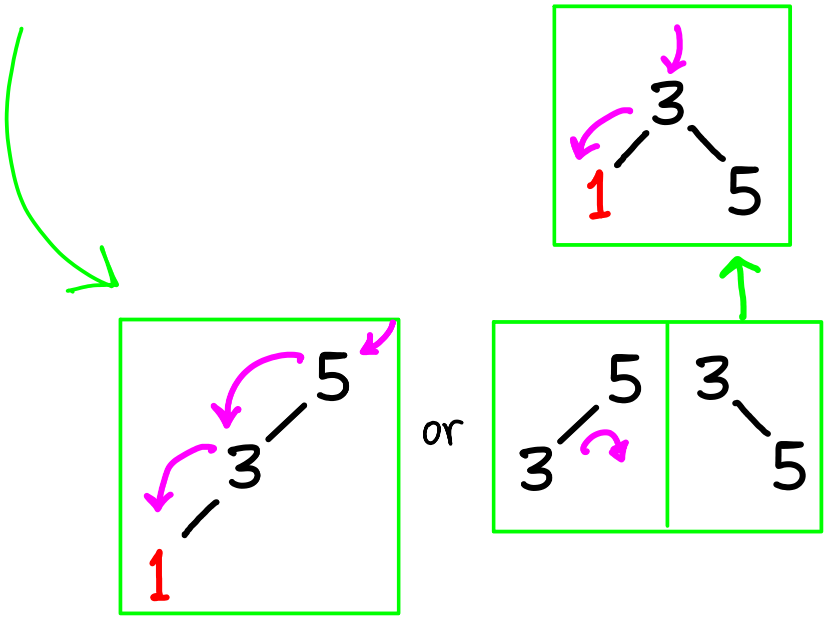
t_5 : insert(4)

t_6 : search(5)

t_7 : delete(5)



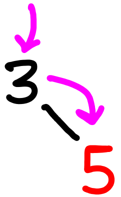


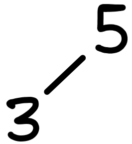


Task: sequence of operations on n keys (one at a time)

t_1 : insert(3)

t_2 : insert(5) Must access 3 first





could rotate

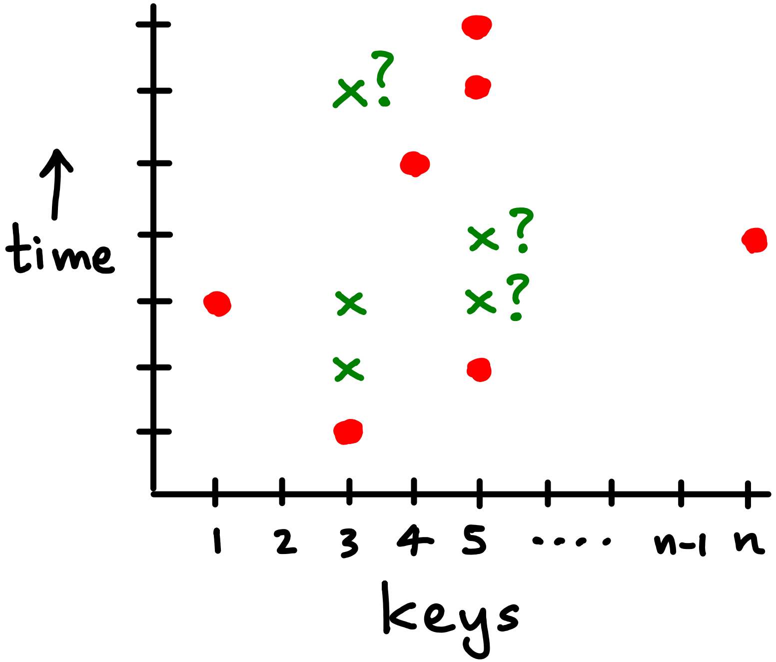
t_3 : insert(1) Must access 3 or 5 first (actually, if 5 then also 3)

t_4 : insert(n)

t_5 : insert(4)

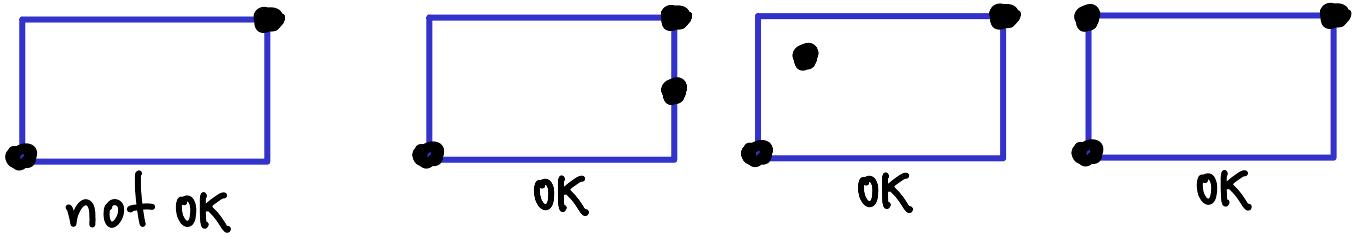
t_6 : search(5)

t_7 : delete(5)

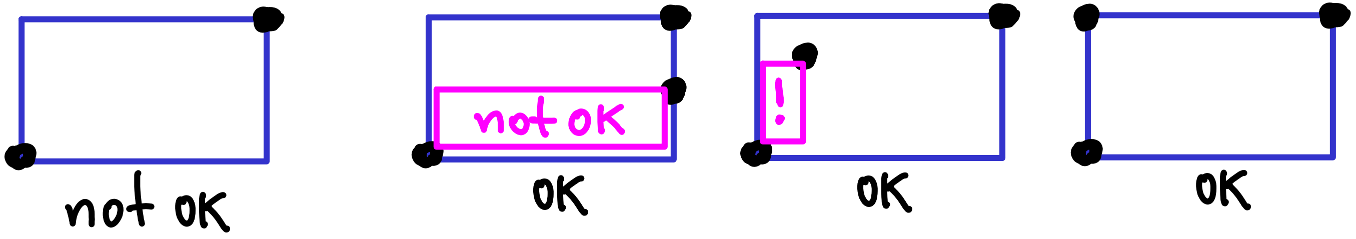


Besides the required ops there are other keys that we may need to go through

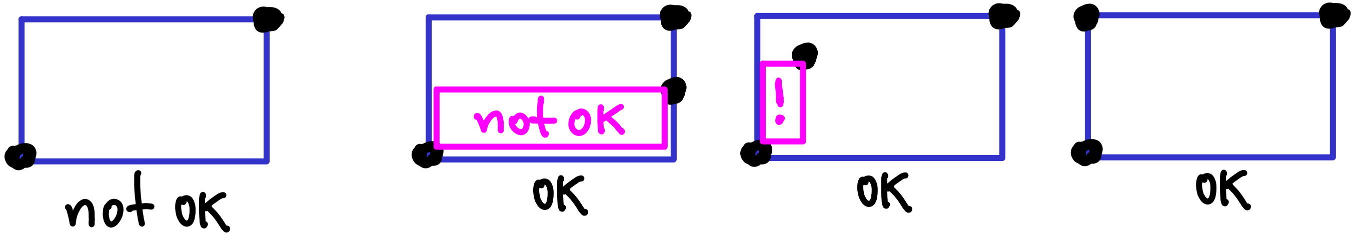
Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

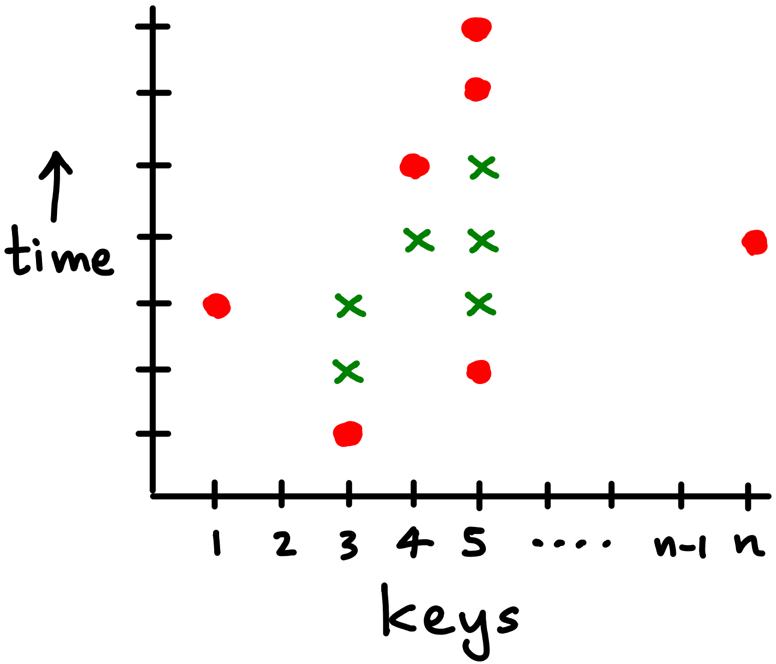


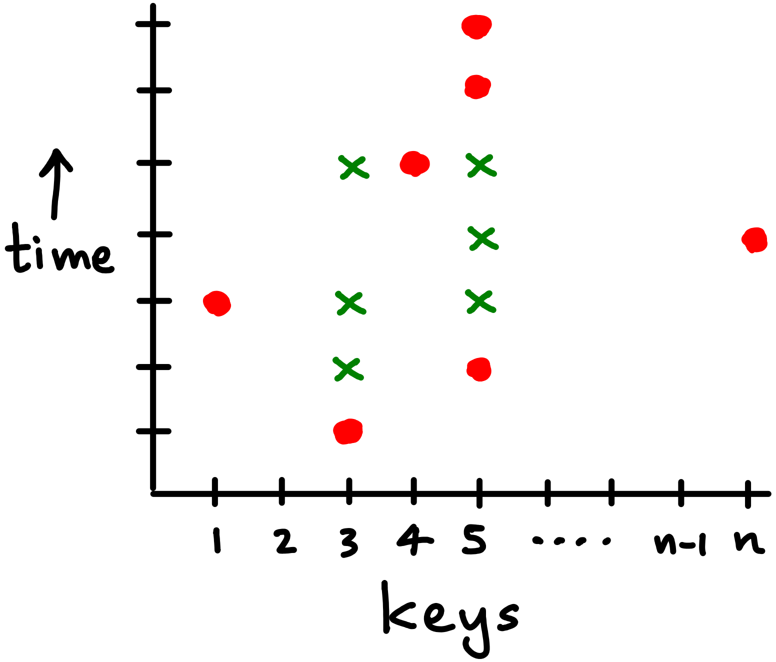
Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points



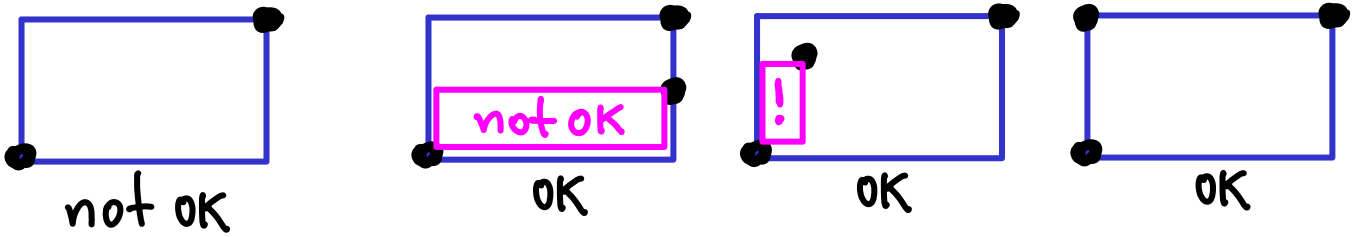
Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points







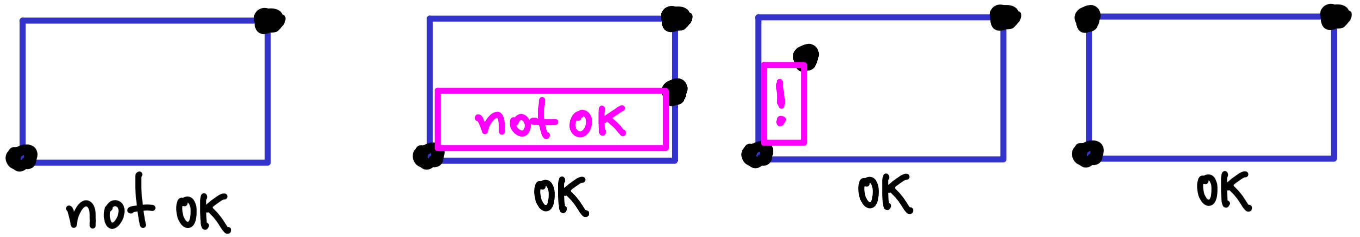
Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

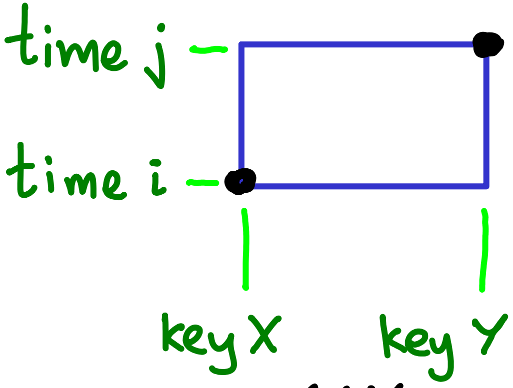


PROOF

(suppose all ops = search)

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

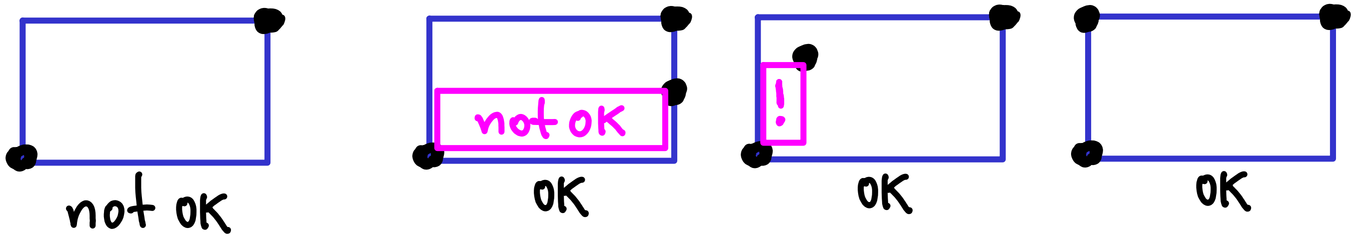


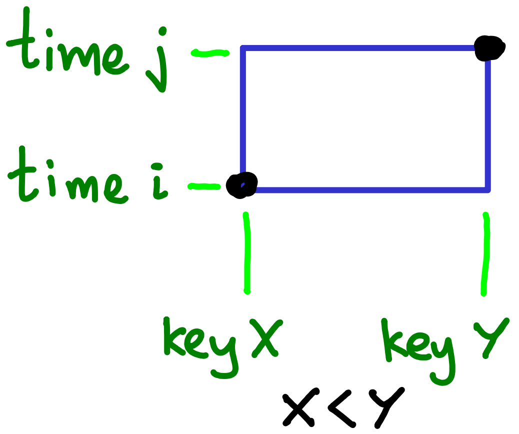


X<Y
(w.l.o.g.)

can this happen?

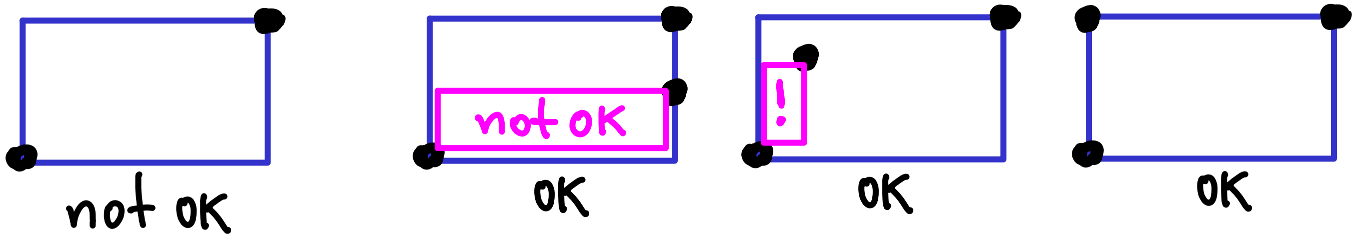
Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

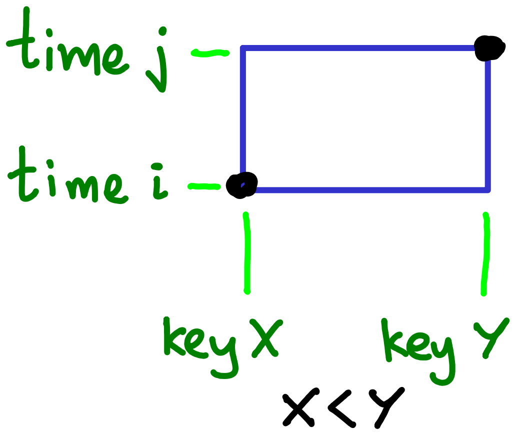




$LCA_i(X,Y)$ = lowest common ancestor at time i

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

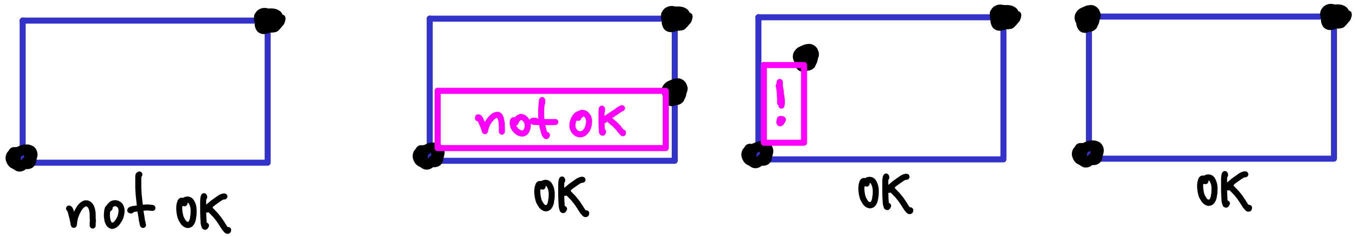


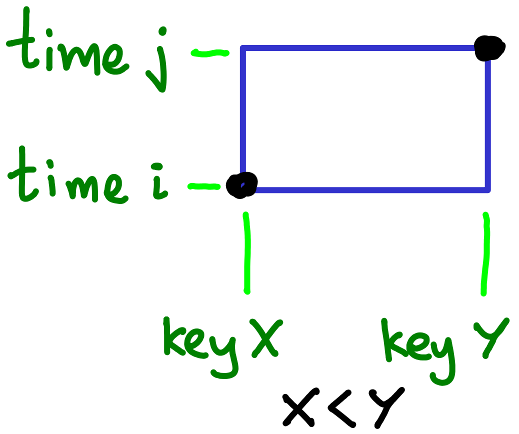


$LCA_i(X,Y)$ = lowest common ancestor at time i

if $LCA_i(X,Y) \neq X$ then ?

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points



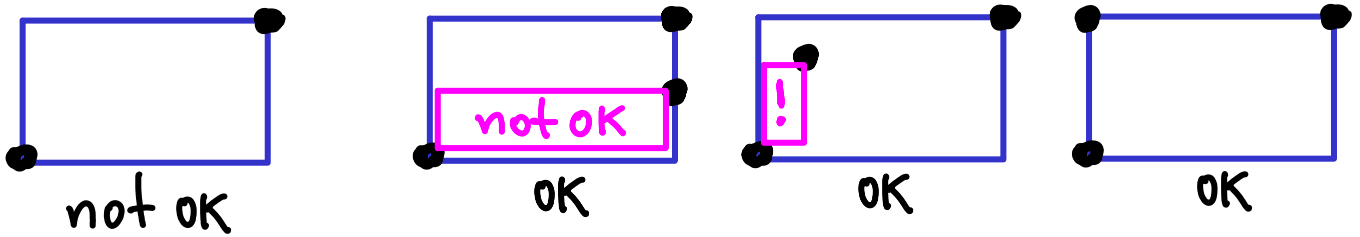


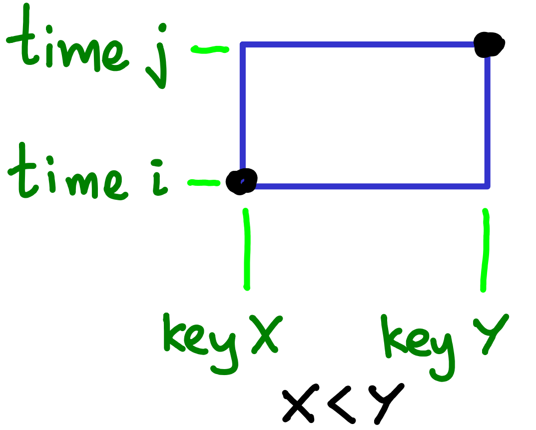
$LCA_i(X,Y)$ = lowest common ancestor at time i

if $LCA_i(X,Y) \neq X$ then: at time i, to access X...

?

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points



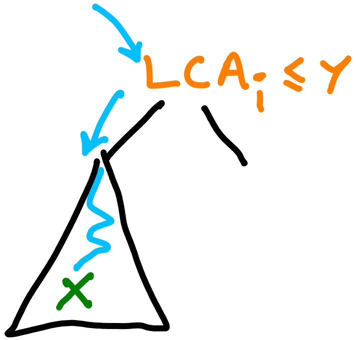


$\mathrm{LCA}_i(X,Y)$ = lowest common ancestor at time i

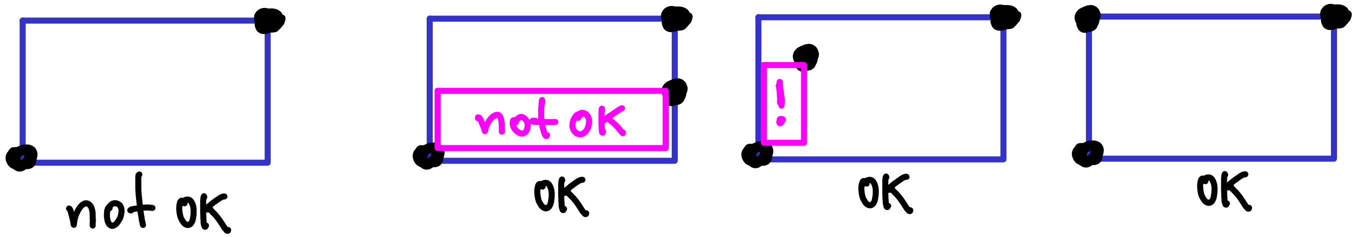
if $\mathrm{LCA}_i(X,Y) \neq X$ then: at time i, to access X...

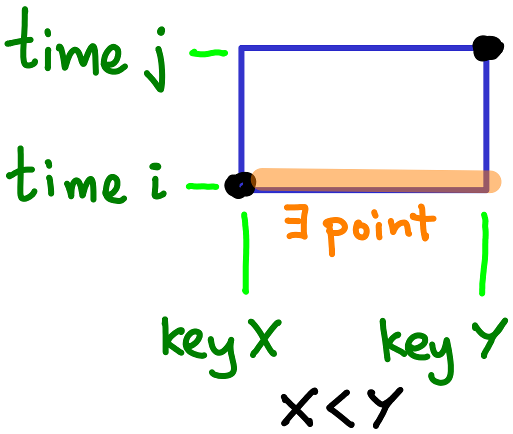
...we must go through $\mathrm{LCA}_i(X,Y)$

So? ←



Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

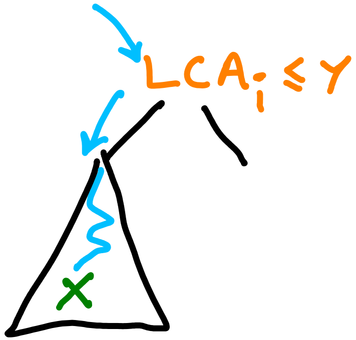




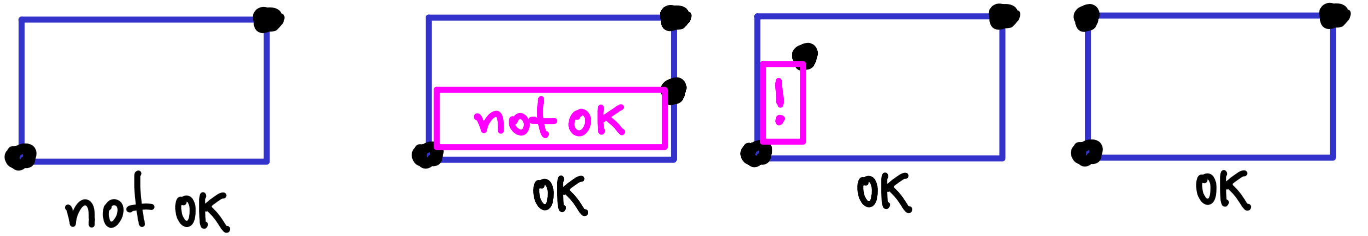
$LCA_i(X,Y)$ = lowest common ancestor at time i

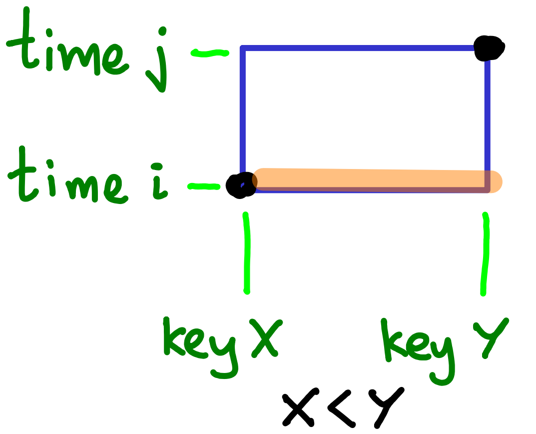
if $LCA_i(X,Y) \neq X$ then: at time i, to access X...

...we must go through $LCA_i(X,Y)$



Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

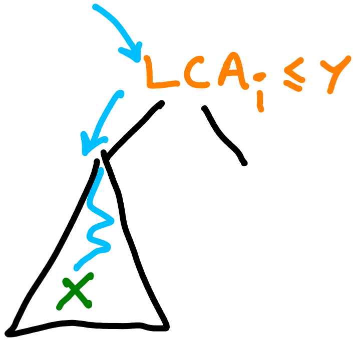




$LCA_i(X,Y)$ = lowest common ancestor at time i

if $LCA_i(X,Y) \neq X$ then: at time i, to access X...

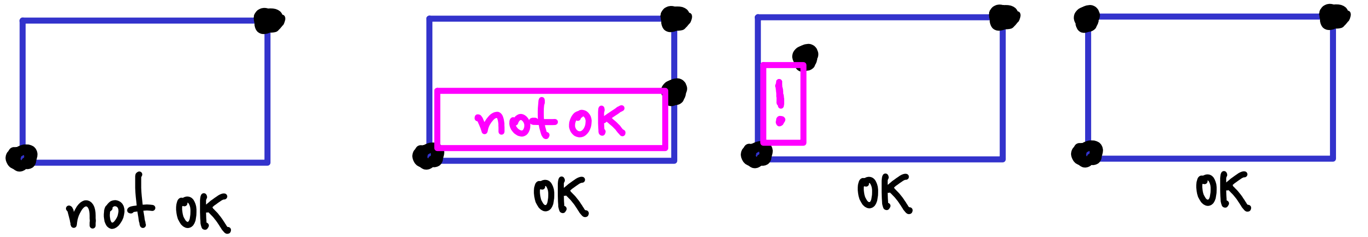
...we must go through $LCA_i(X,Y)$

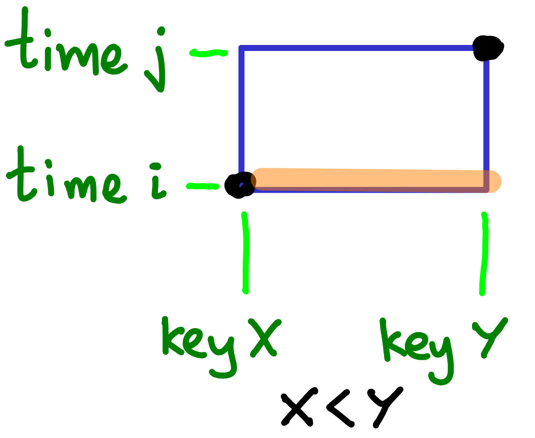


else

$X = LCA_i$

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

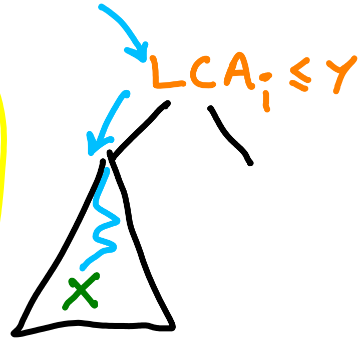




$LCA_i(X,Y)$ = lowest common ancestor at time i

if $LCA_i(X,Y) \neq X$ then: at time i, to access X...

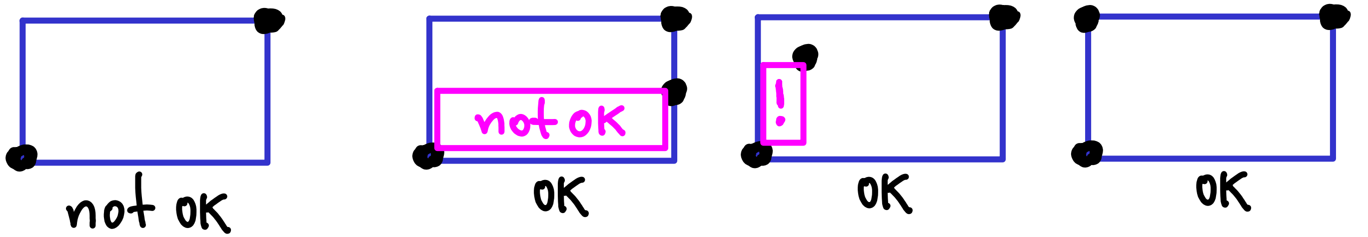
...we must go through $LCA_i(X,Y)$

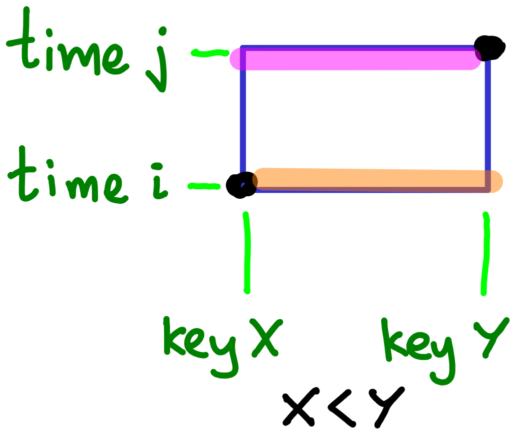


else if $LCA_j(X,Y) \neq Y$

?

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

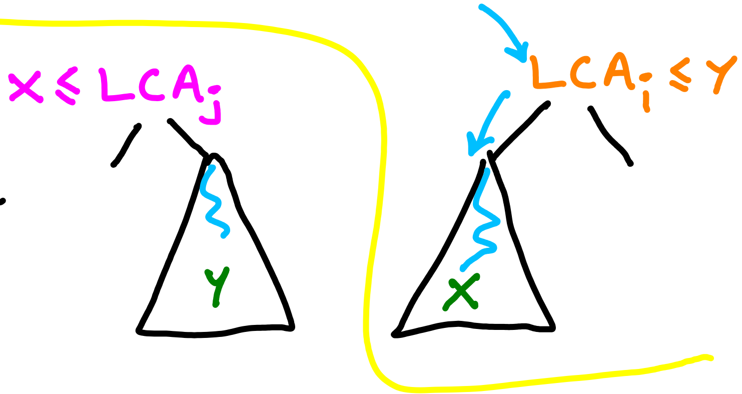




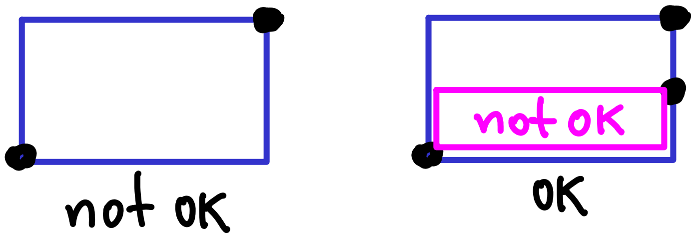
$LCA_i(X,Y)$ = lowest common ancestor at time i

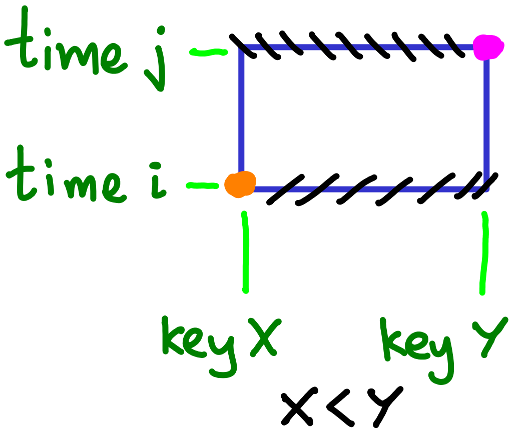
if $LCA_i(X,Y) \neq X$ then: at time i, to access X... ...we must go through $LCA_i(X,Y)$

else if $LCA_j(X,Y) \neq Y$ then we must go through $LCA_j(X,Y)$



Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points

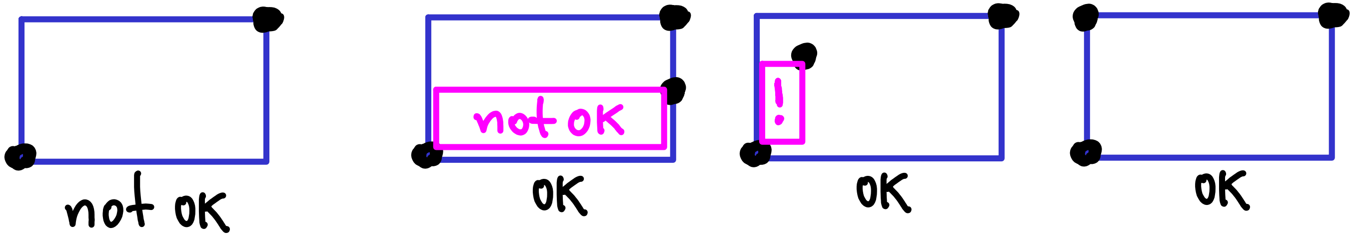


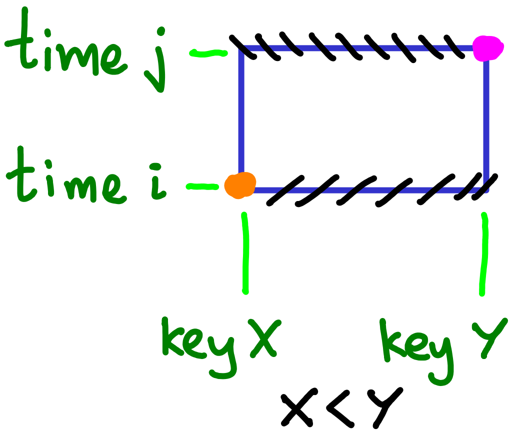


$LCA_i(X,Y)$ = lowest common ancestor at time i

So far the only hope of contradicting the theorem is

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points



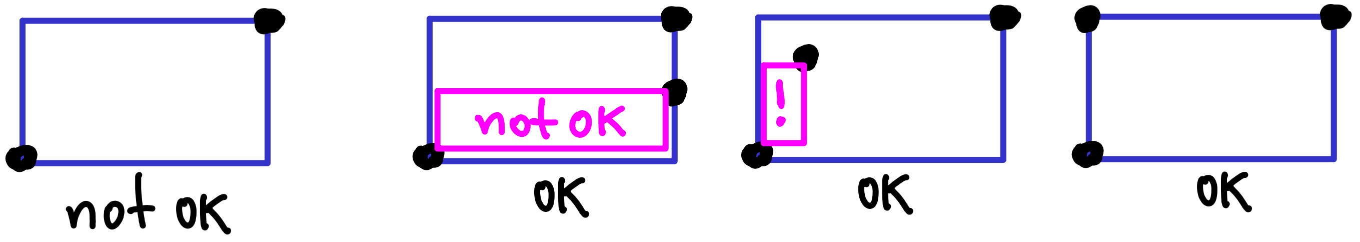


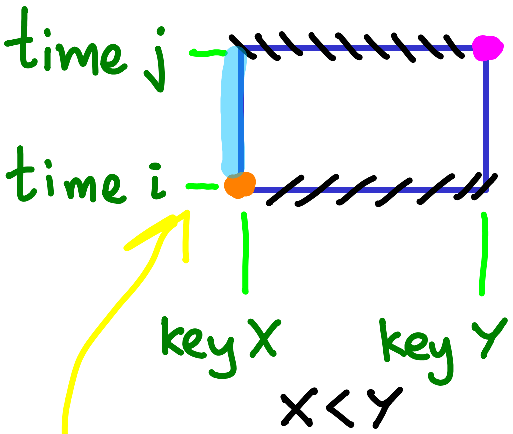
$LCA_i(X,Y)$ = lowest common ancestor at time i

So far the only hope of contradicting the theorem is

but then some time t $(i < t \leq j)$ we must rotate X

Theorem: the set of key accesses (over all ops) corresponds to a diagram where no two points form opposite corners of a closed rectangular region that is empty of other points





$LCA_i(X,Y)$ = lowest common ancestor at time i

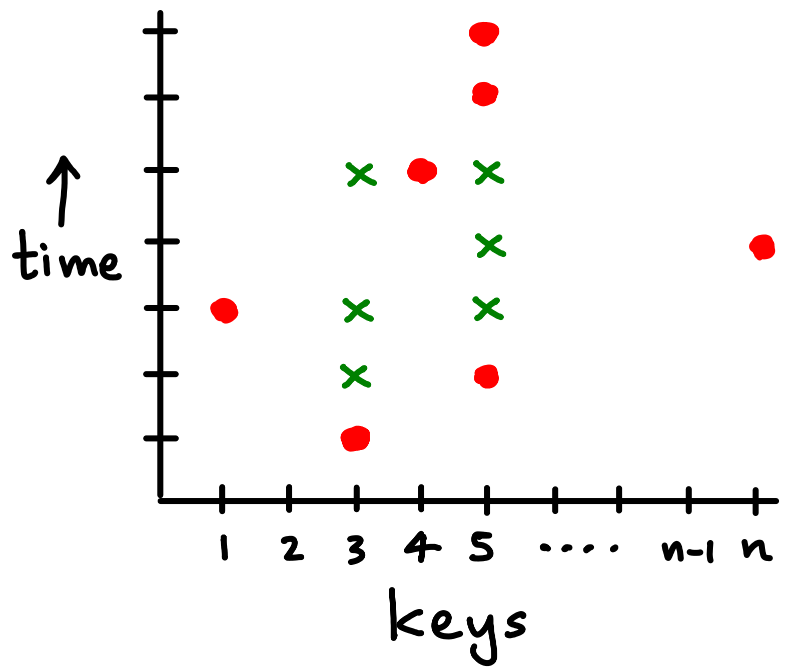
So far the only hope of contradicting the theorem is

but then some time t ($i<t\leq j$) we must rotate X

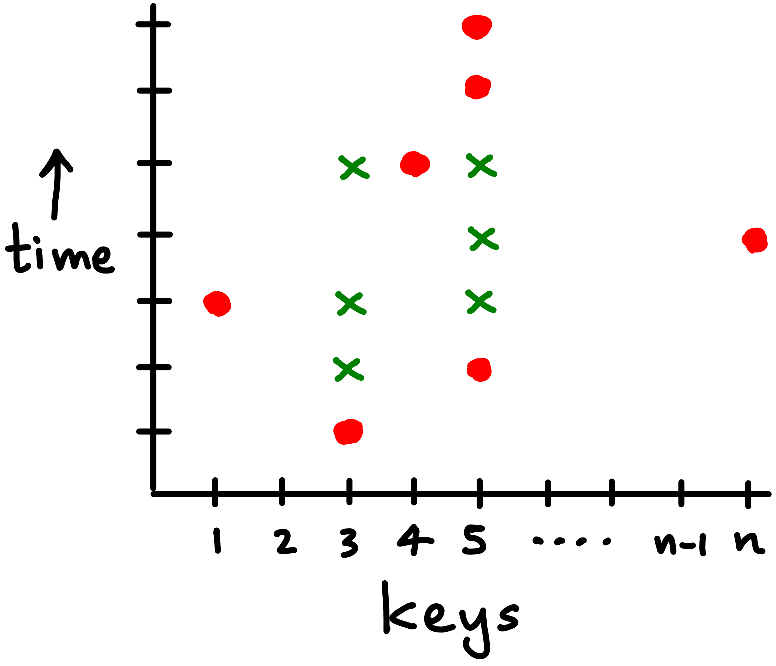
□

Theorem: any diagram where no two points form opposite corners of a closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm

(reverse)

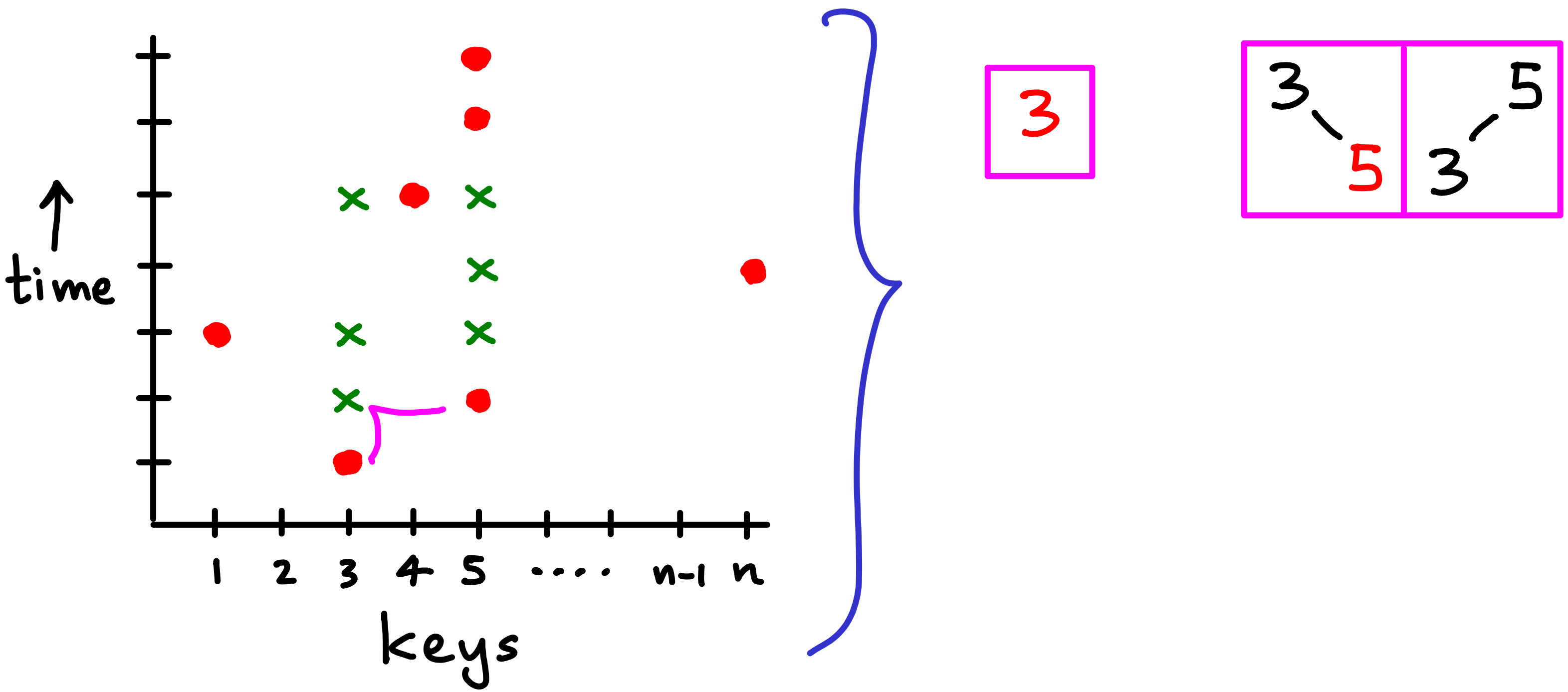


Theorem: any diagram where no two points form opposite corners of a
(reverse)
closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm

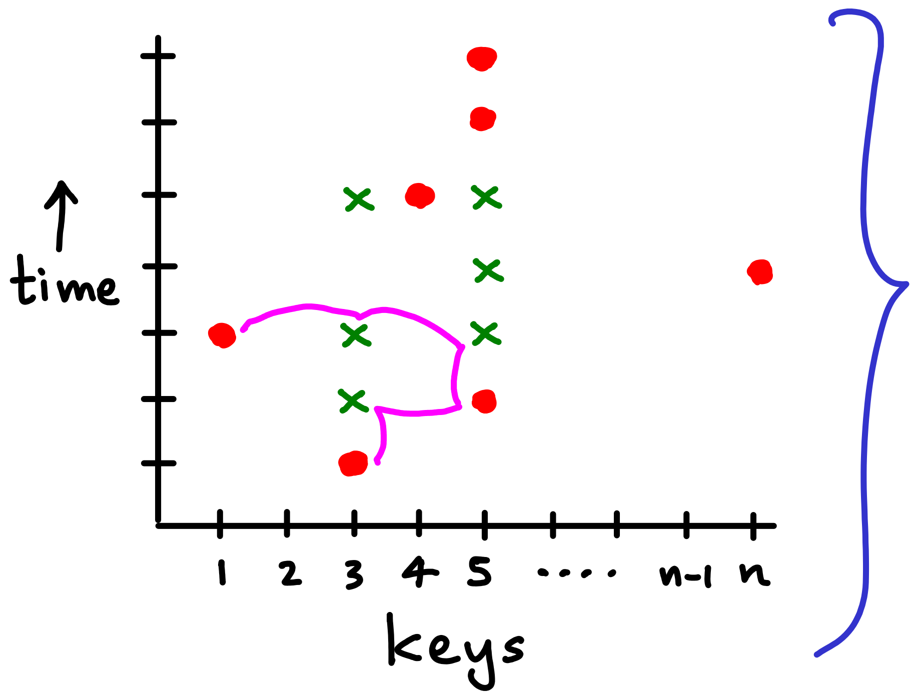


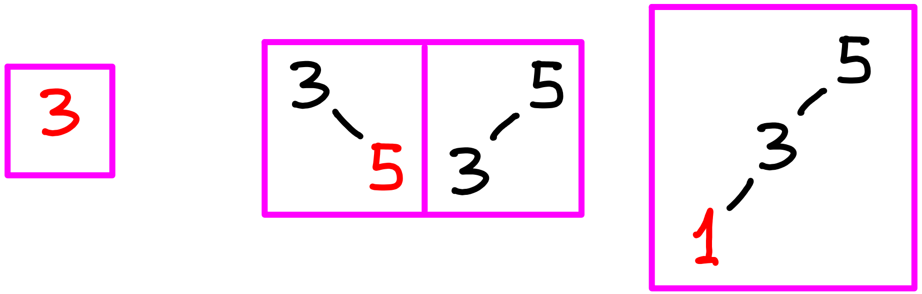
$\boxed{3}$

Theorem: any diagram where no two points form opposite corners of a
(reverse)
closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm

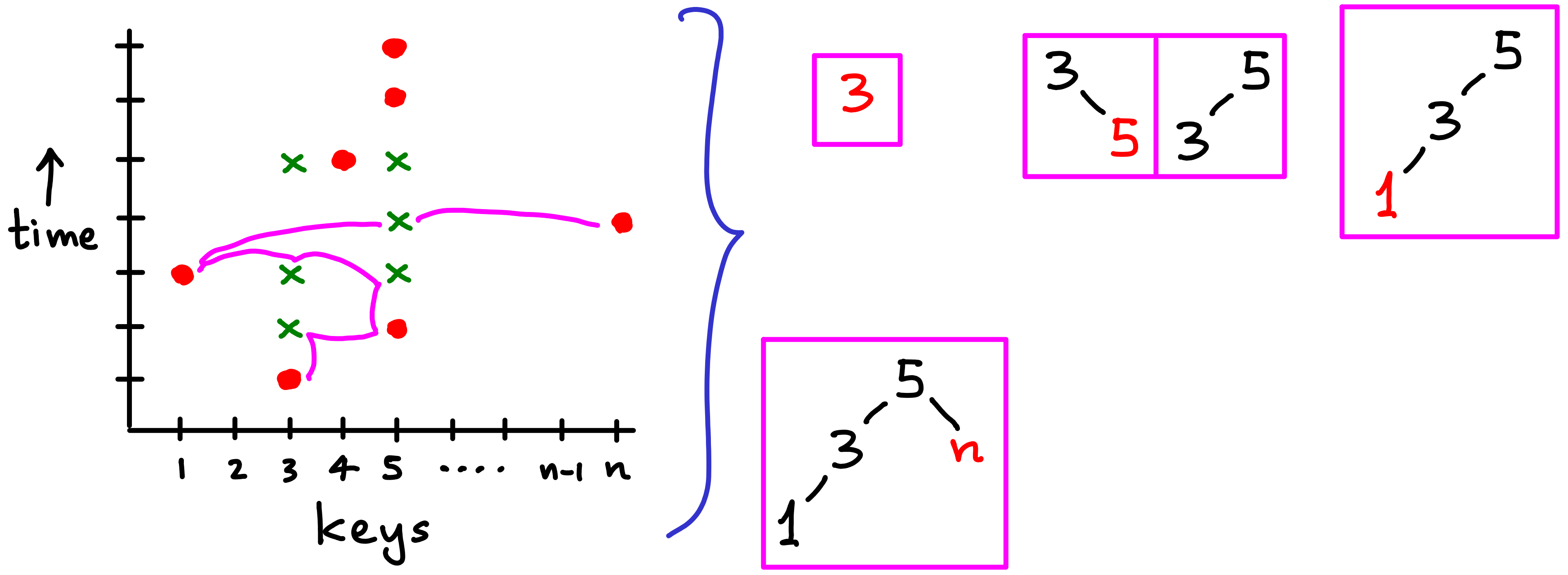


Theorem: any diagram where no two points form opposite corners of a closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm
(reverse)

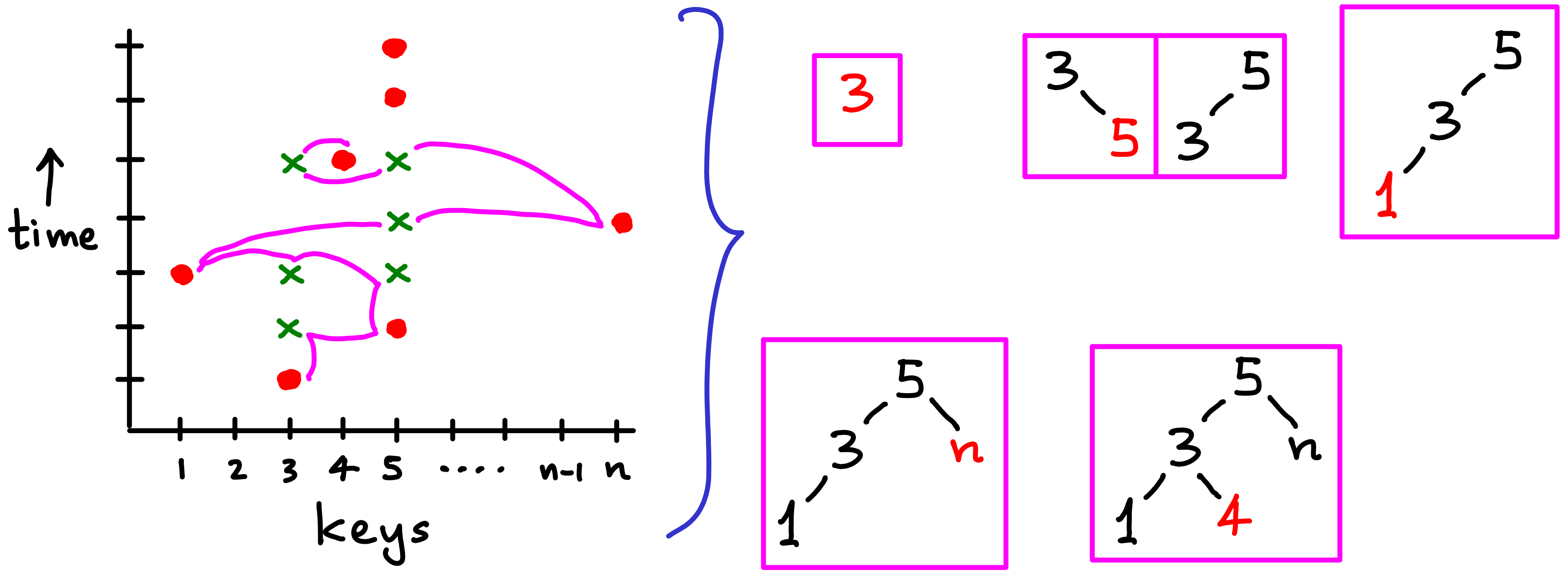




Theorem: any diagram where no two points form opposite corners of a
(reverse)
closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm

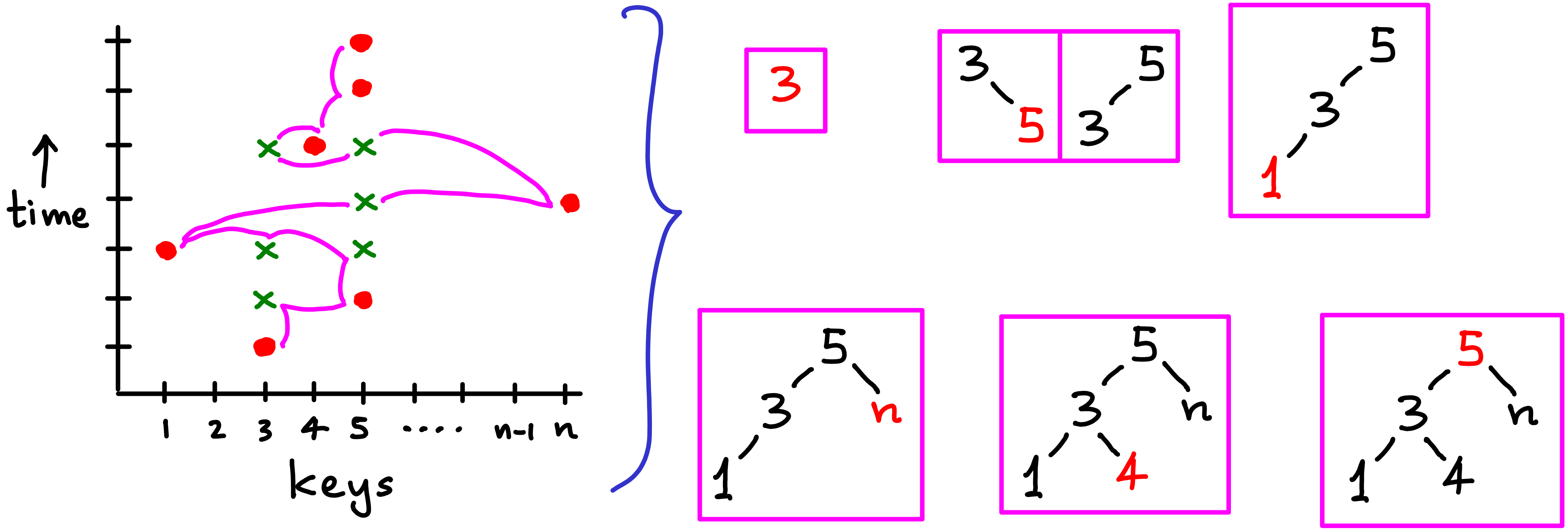


Theorem: any diagram where no two points form opposite corners of a
(reverse)
closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm



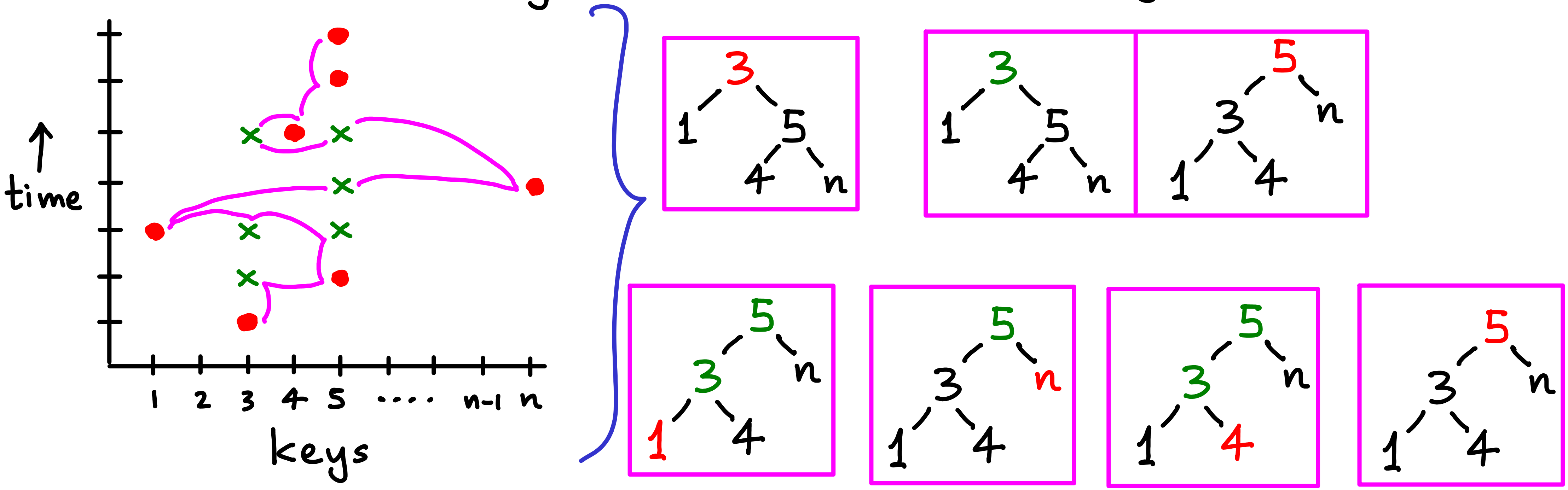
Theorem: any diagram where no two points form opposite corners of a closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm

(reverse)



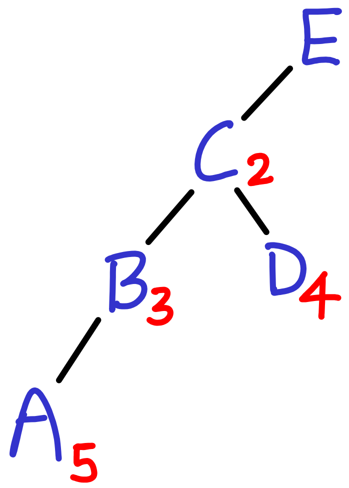
Theorem: // version for static set of keys (no insert-delete)

(reverse) any diagram where no two points form opposite corners of a closed rectangular region that is empty of other points corresponds to the set of key accesses for some BST algorithm



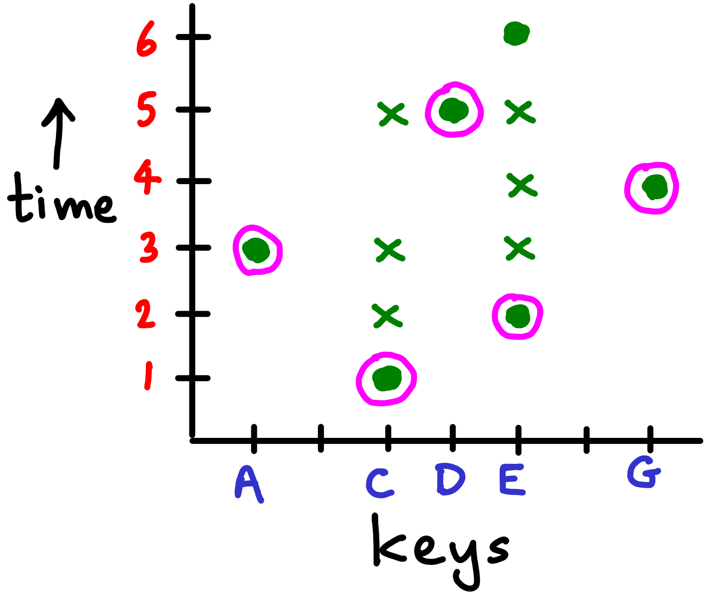
We will prove the theorem constructively, using TREAPS

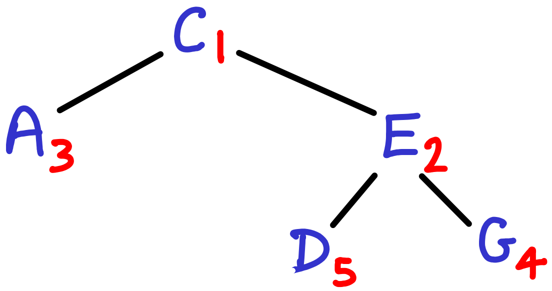
BST + HEAP



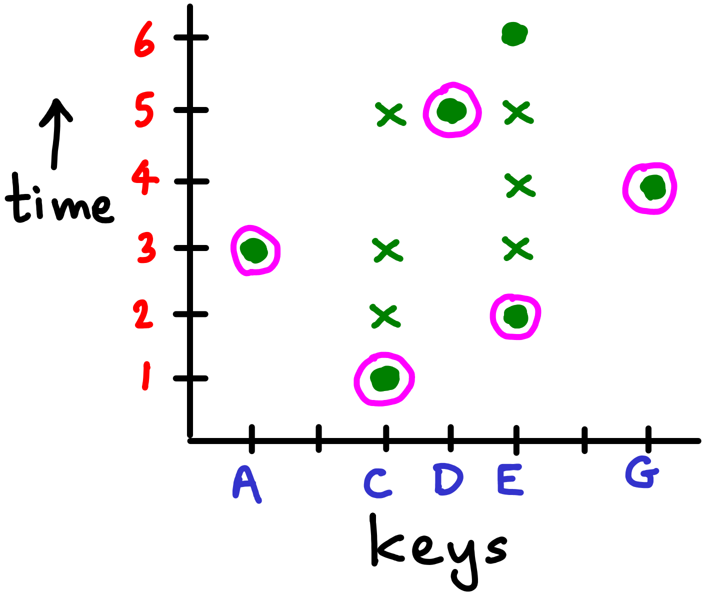
If heap values (and keys) are unique then shape is too.

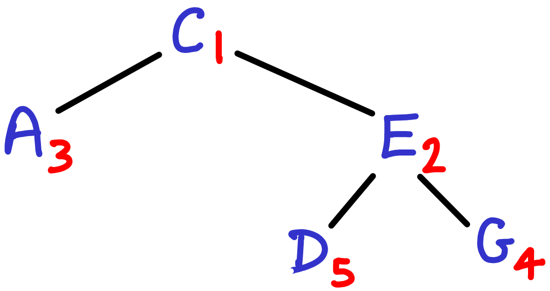
Given a diagram, let every key have treap priority = lowest access time





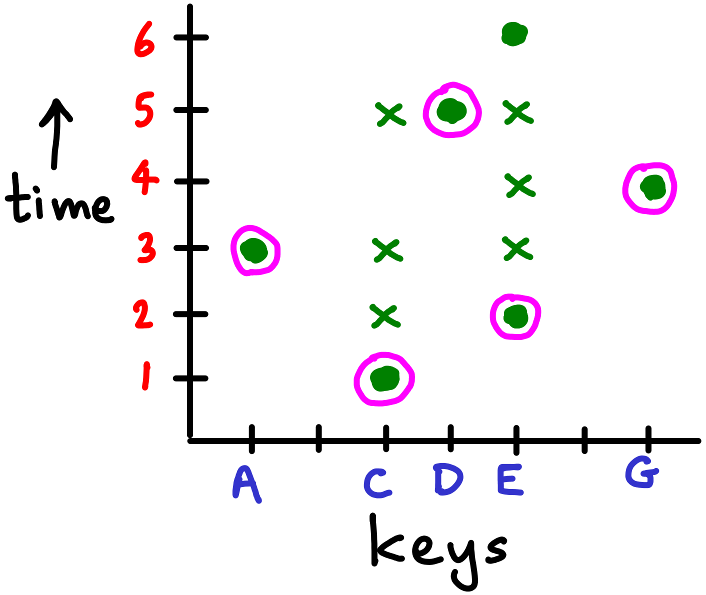
Given a diagram, let every key have treap priority = lowest access time

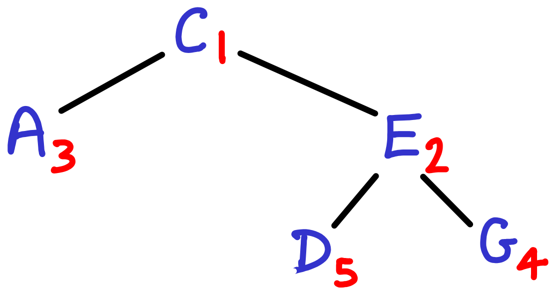




Treap priorities will increase over time

Given a diagram, let every key have treap priority = lowest access time

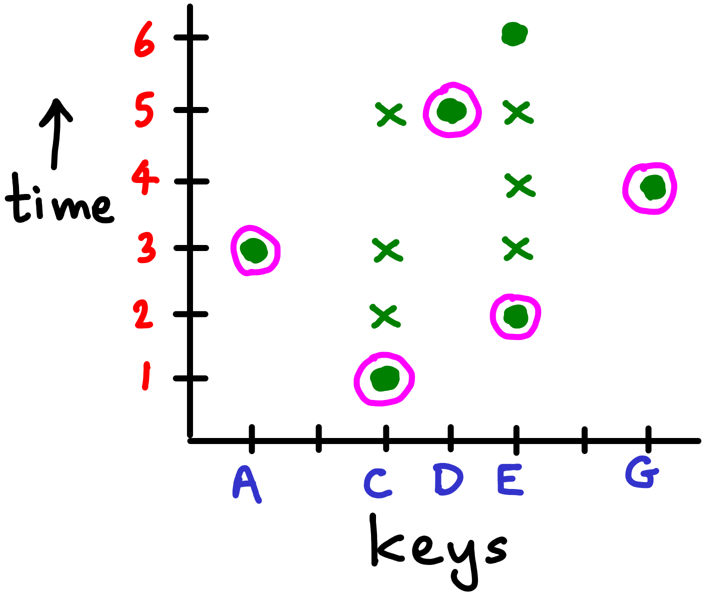


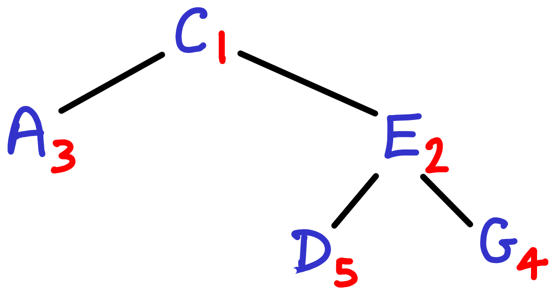


Treap priorities will increase over time

top of treap:
all nodes with
priority = root

Given a diagram, let every key have treap priority = lowest access time



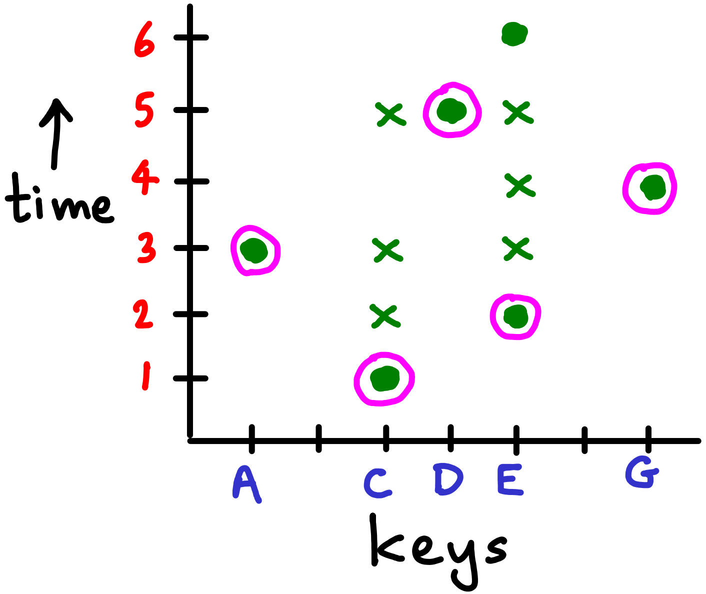


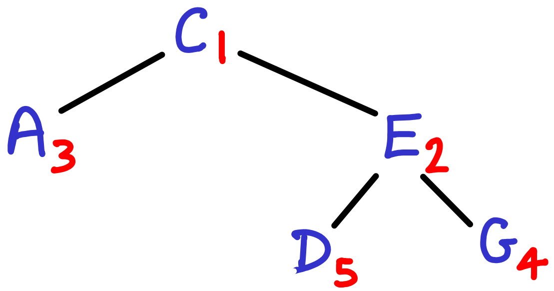
Treap priorities will increase over time

top of treap: all nodes with priority = root

At time = min priority, we must access all of top and nothing below in treap

Given a diagram, let every key have treap priority = lowest access time



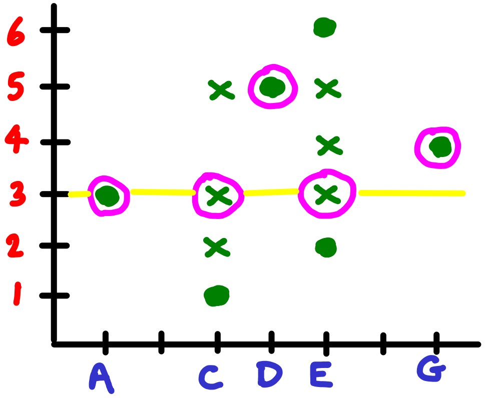


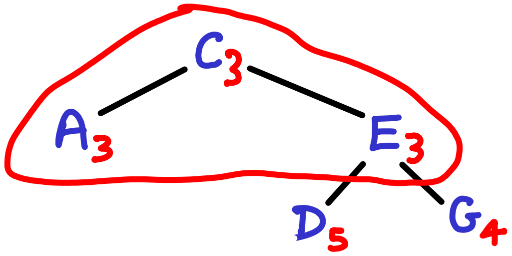
Treap priorities will increase over time

top of treap:
all nodes with
priority = root

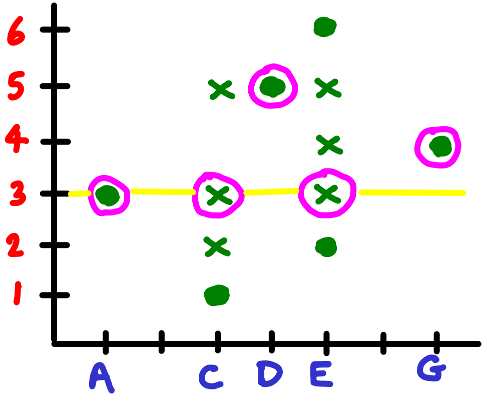
At time = min priority, we must access all of top
and nothing below in treap

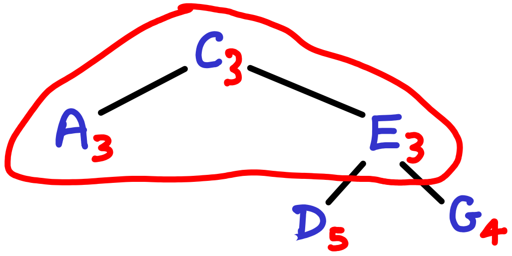
top may change shape via rotations,
and all priorities within will increase in value
(to next required access time)


6
5
4
3
2
1
A
C
D
E
G

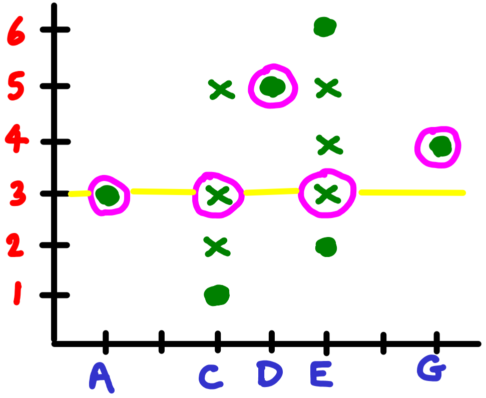

C3
A3
E3
D5
G4

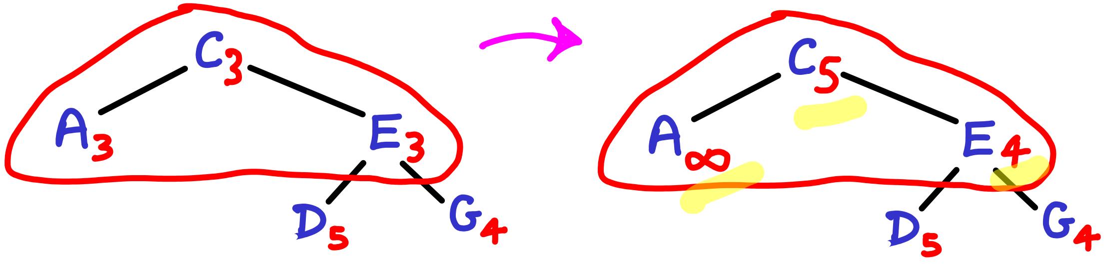
How should priorities in top change?

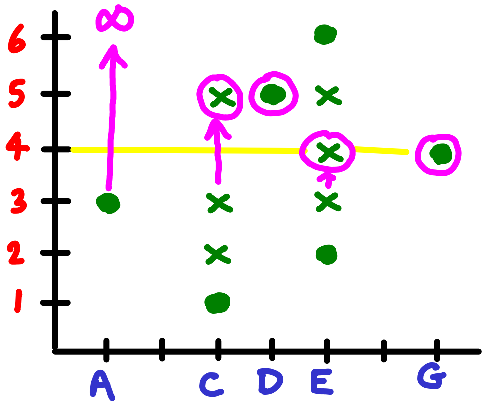




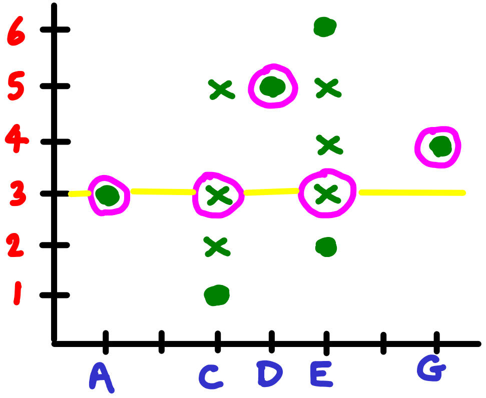
How should priorities in top change?

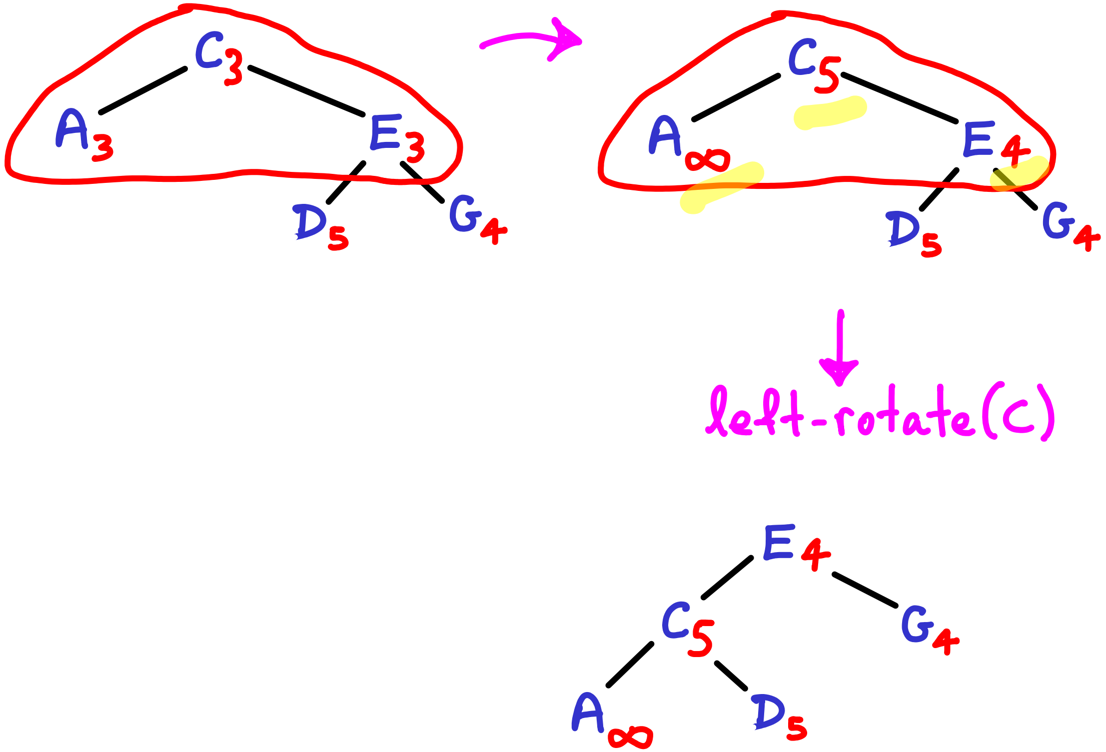


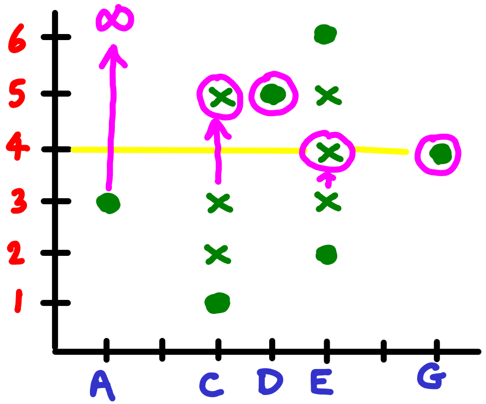




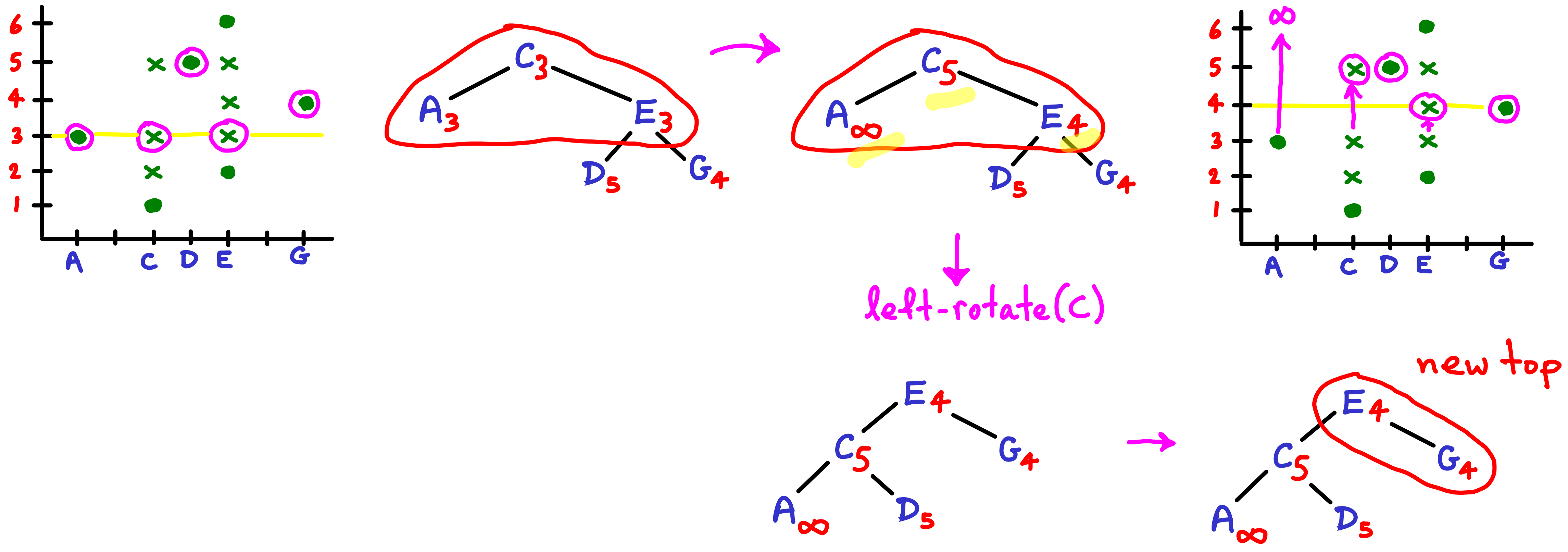
How should priorities in top change?



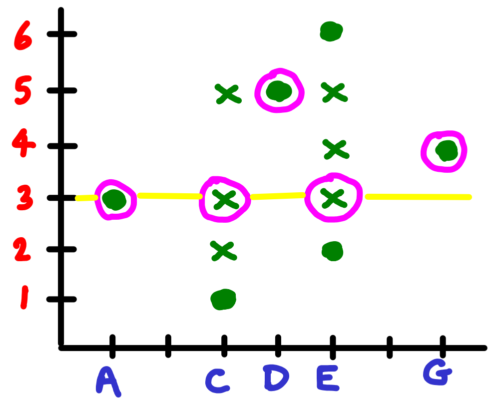


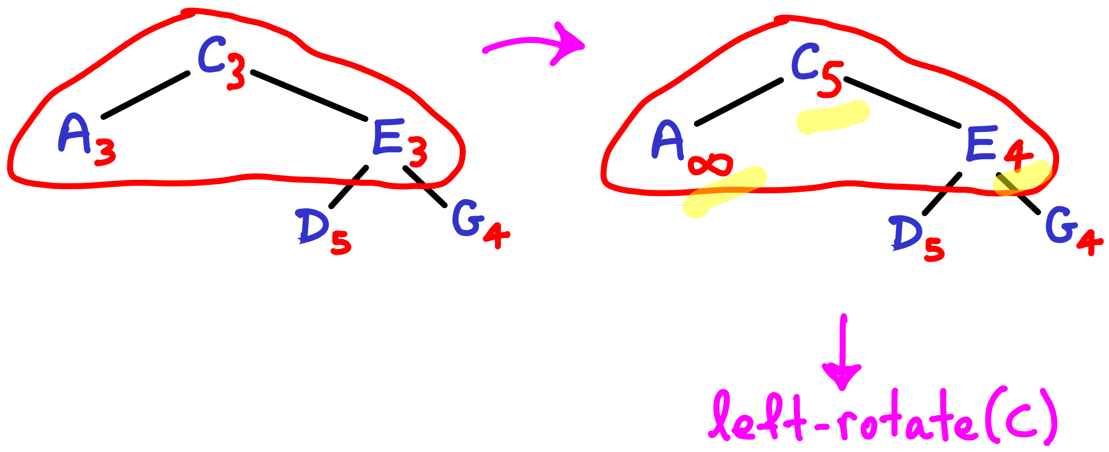


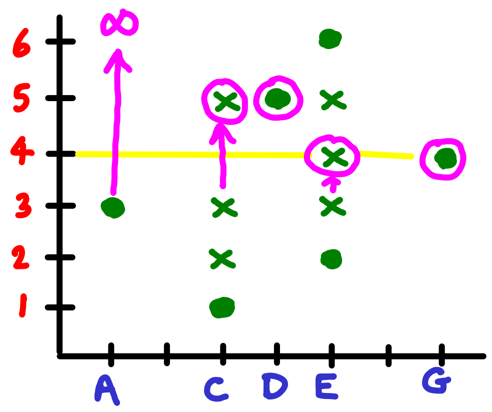
How should priorities in top change?



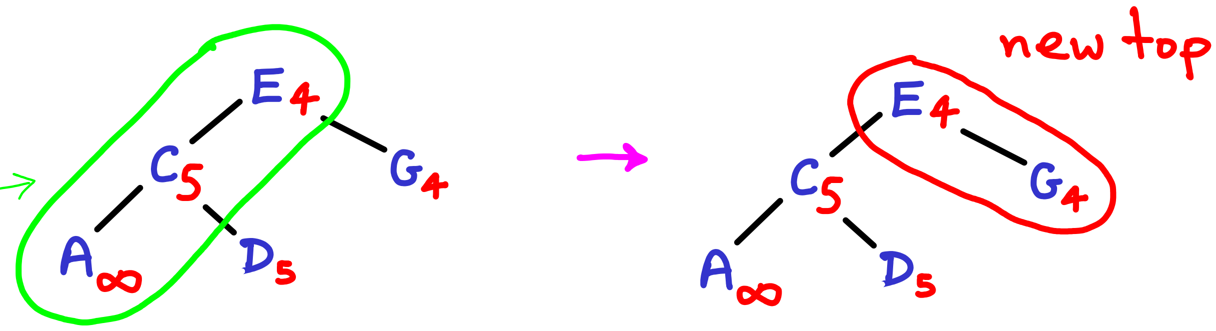
How should priorities in top change?



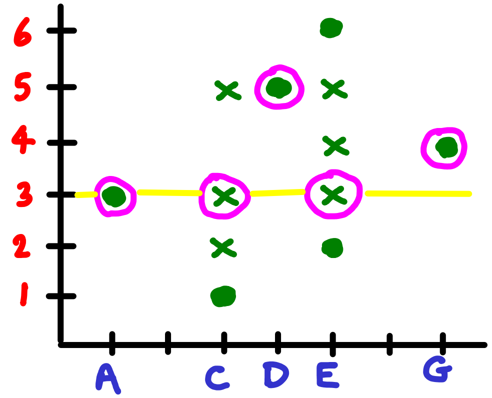


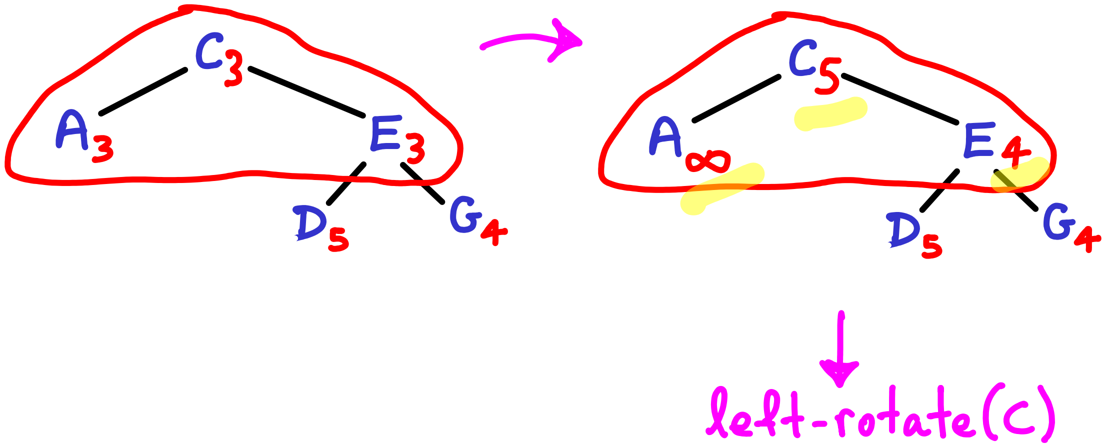


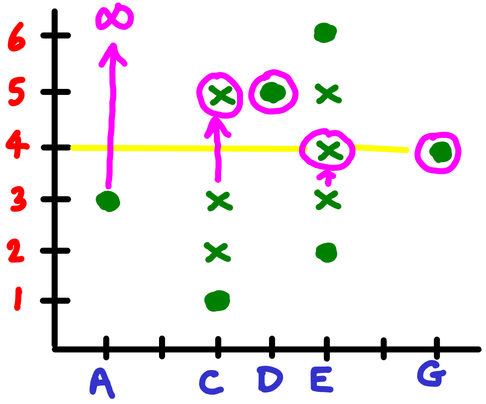
Just convert given shape of top
to whatever shape we need
s.t. heap property is restored



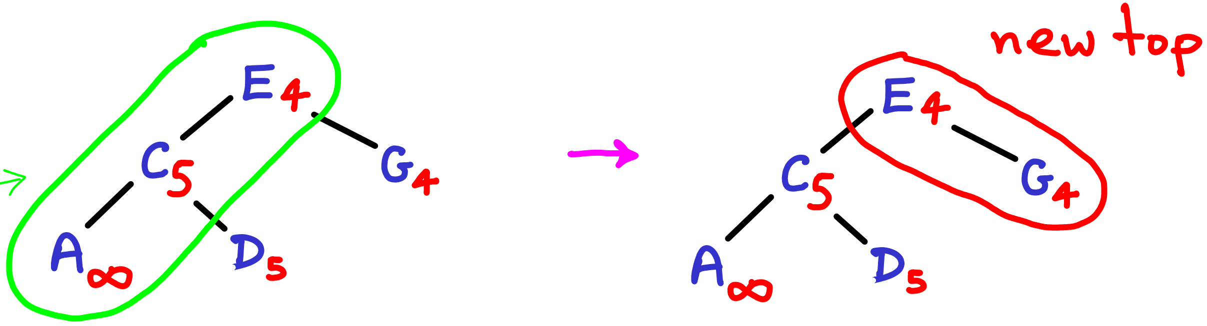
How should priorities in top change?



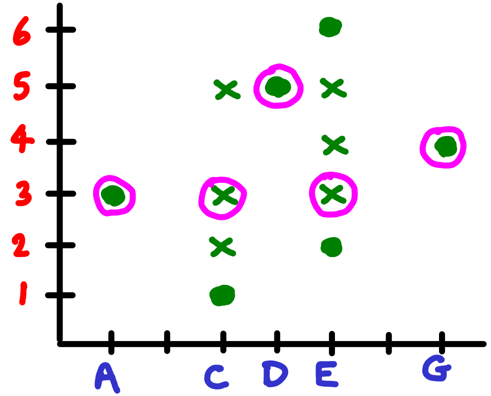


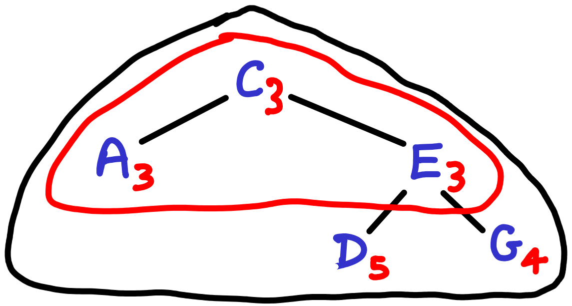


Just convert given shape of top to whatever shape we need s.t. heap property is restored

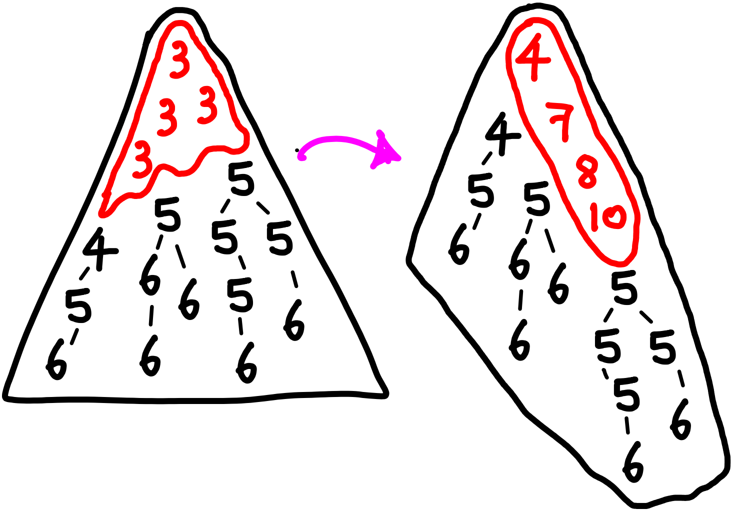


From COMP-160 we know this can always be done with <2n rotations



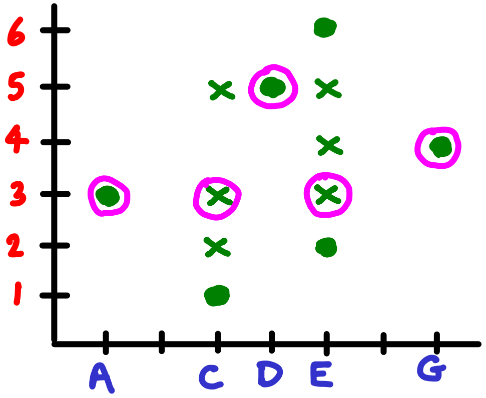


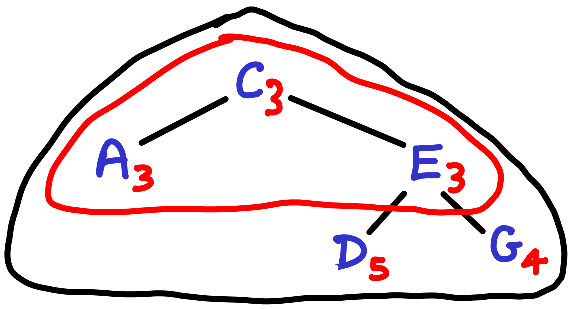
Method so far: at every time step, we will access (and possibly rotate) precisely the nodes in top, and update their priorities.



This restores the top as a treap.

All other nodes passively follow.

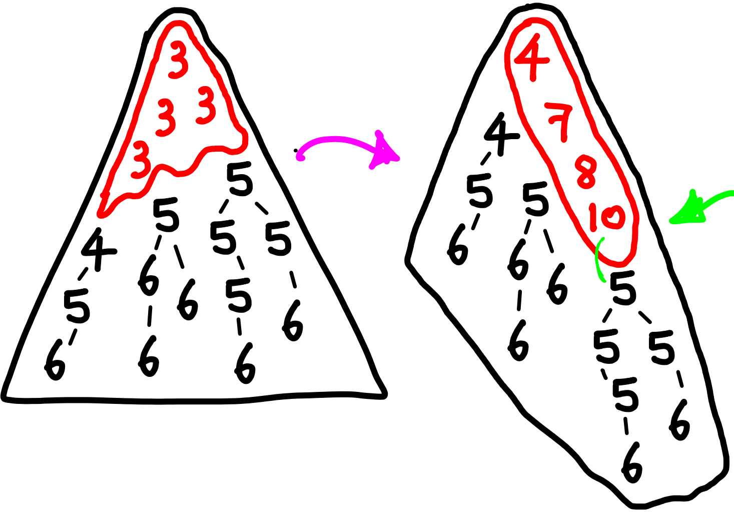




Method so far: at every time step, we will access (and possibly rotate) precisely the nodes in top, and update their priorities.

This restores the top as a treap.

All other nodes passively follow.

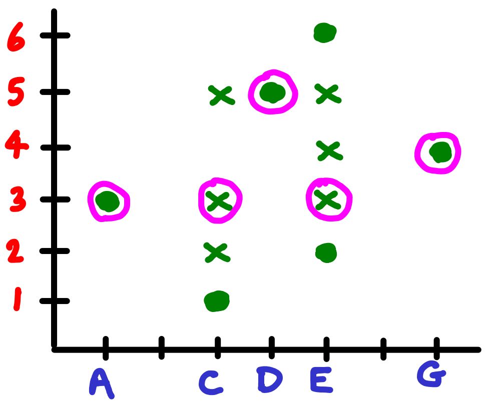


Claim:

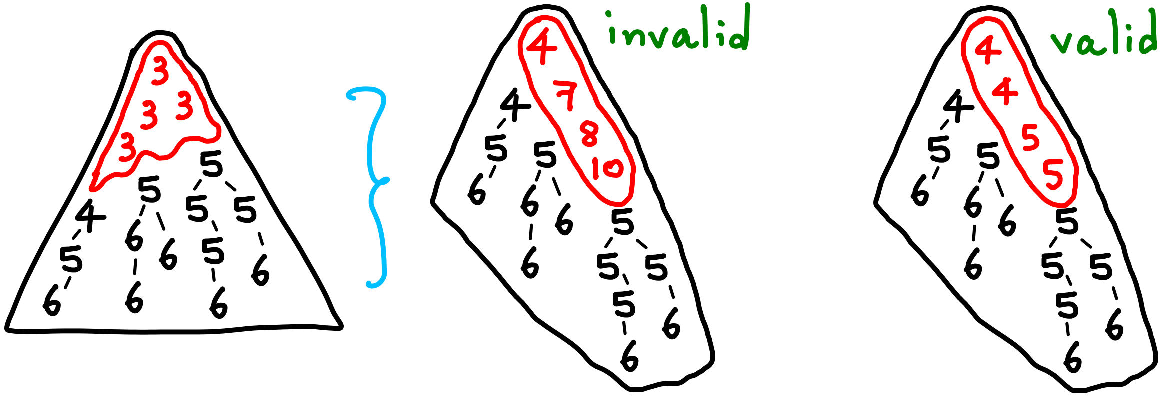
new priorities won't violate heap property globally.

(this won't happen)

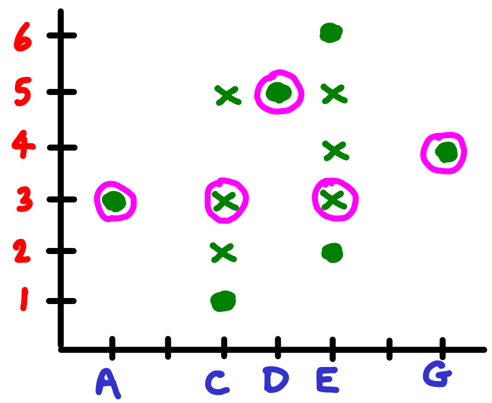
...so we always have a treap



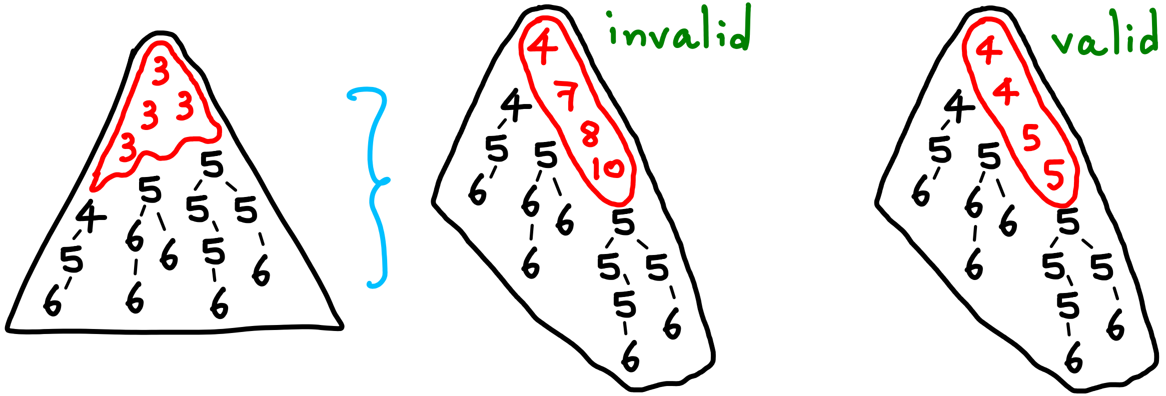
Claim:
new priorities
won't violate
heap property



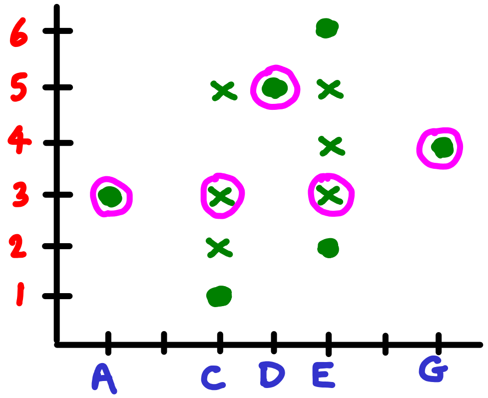
PROOF...



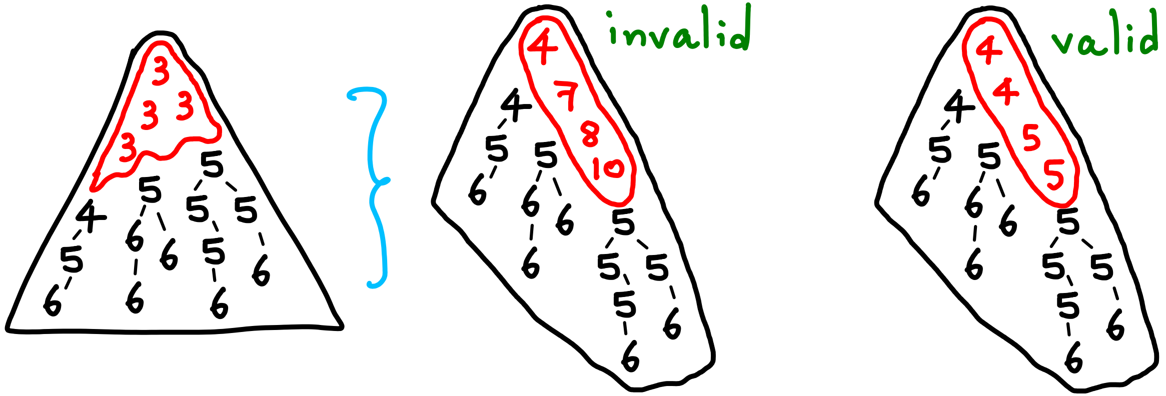
Claim:
new priorities won't violate heap property



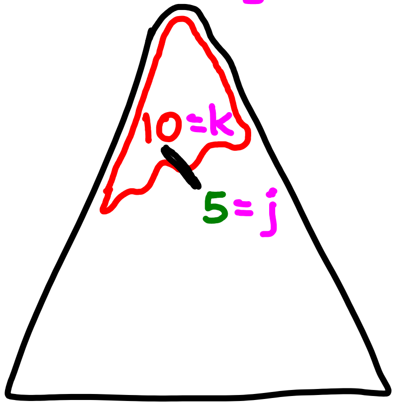
Proof by contradiction:



Claim:
new priorities won't violate heap property

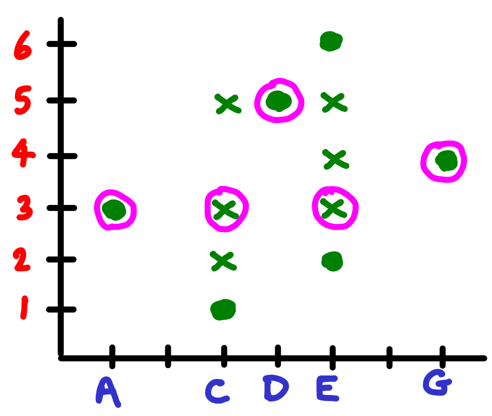


Proof by contradiction: suppose ∃ edge X — Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$

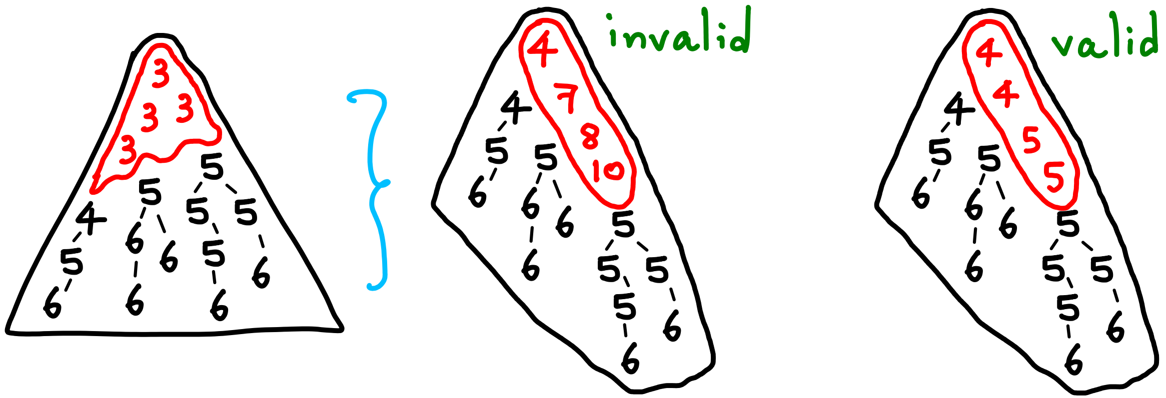


keys:

wlog X<Y



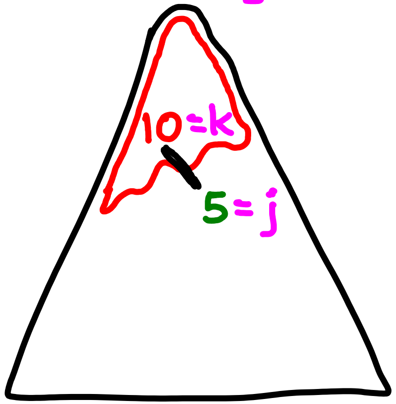
Claim:
new priorities won't violate heap property

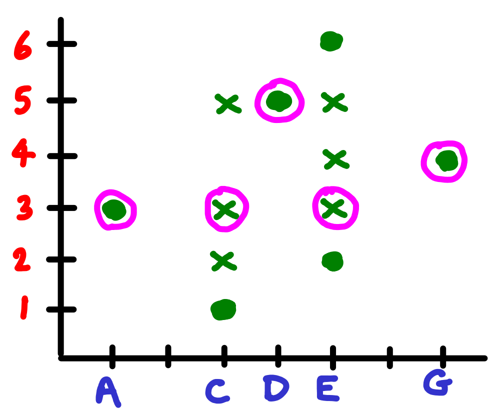


Proof by contradiction: suppose $\exists$ edge X–Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$

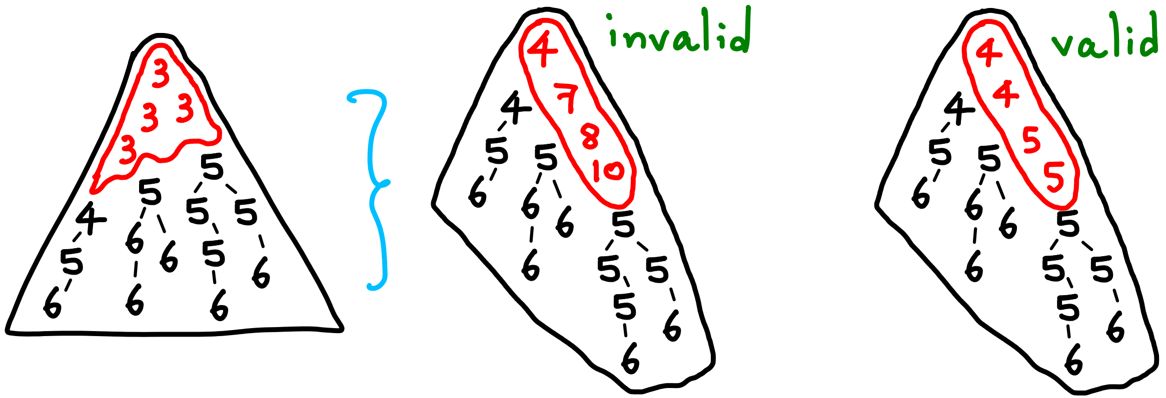
We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

WHY?



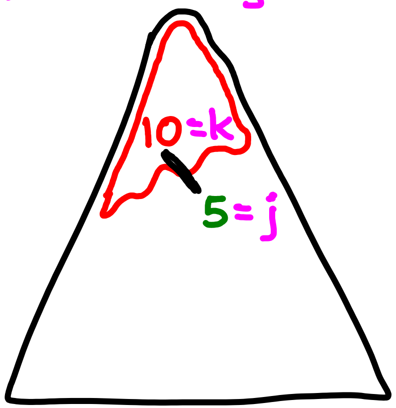


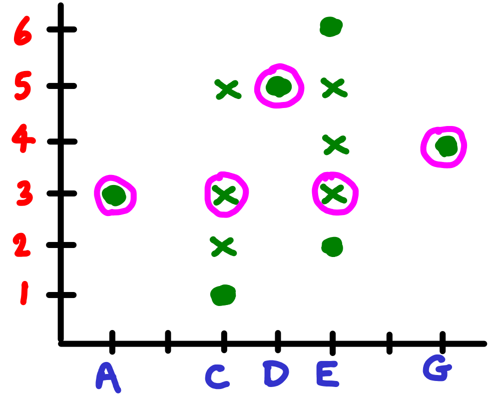
Claim:
new priorities won't violate heap property



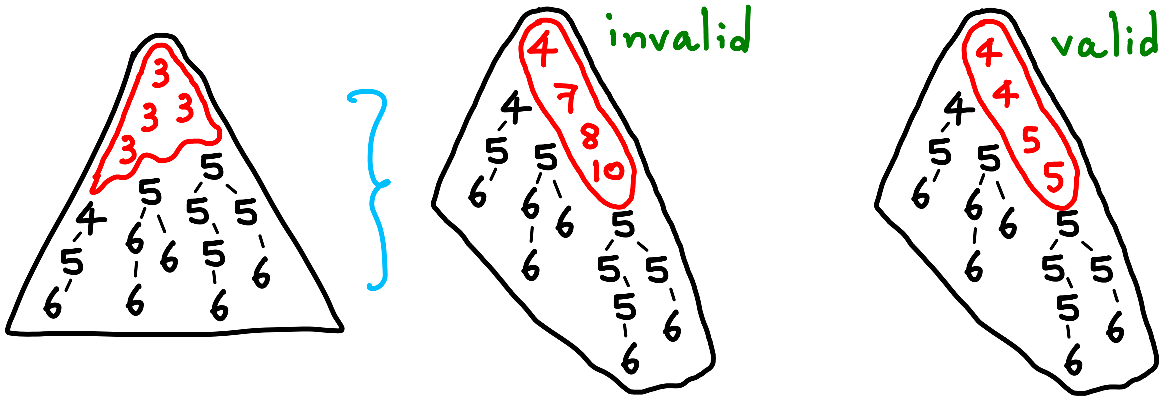
Proof by contradiction: suppose $\exists$ edge X–Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$

We know $\overbrace{\underbrace{\text{priority}(Y)}_{\text{wasn't in top}}}^{j} > \overbrace{\text{old priority}(X)}^{i}$

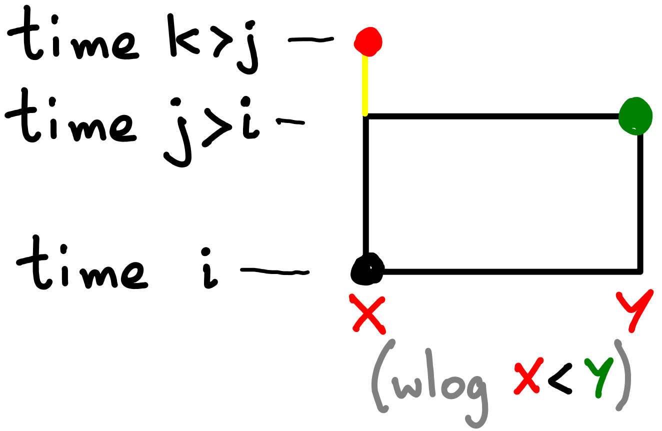




Claim: new priorities won't violate heap property

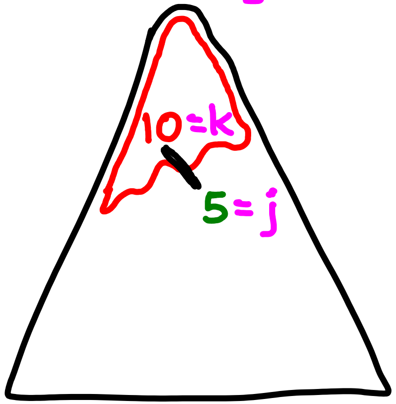


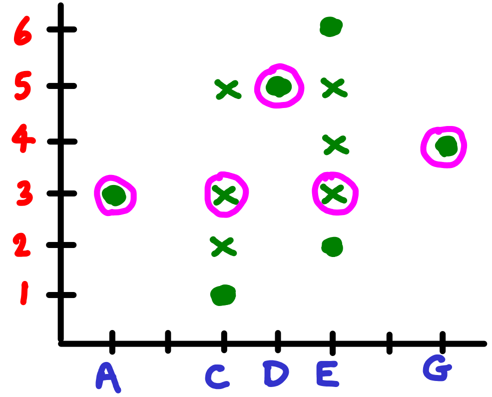
Proof by contradiction: suppose ∃ edge X–Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



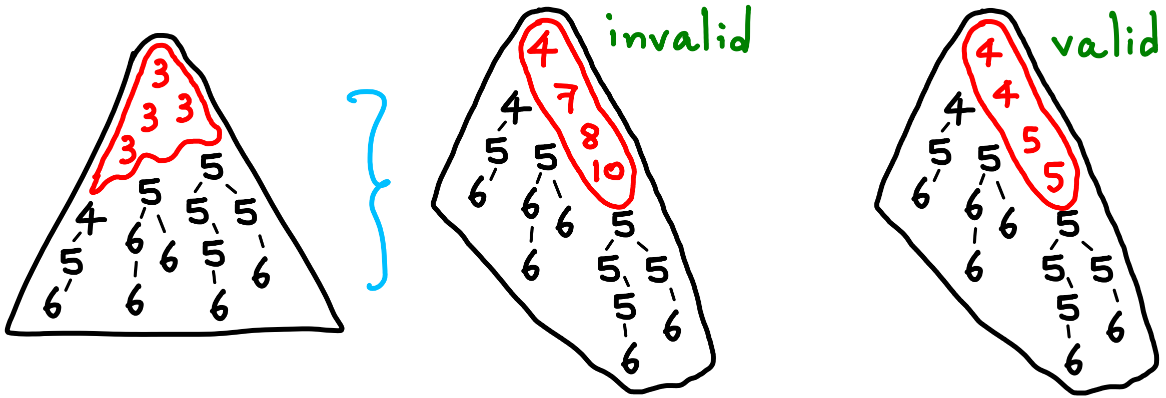
We know $\overbrace{\text{priority}(Y)}^{j} \overset{>}{>} \overbrace{\text{old priority}(X)}^{i}$

...?

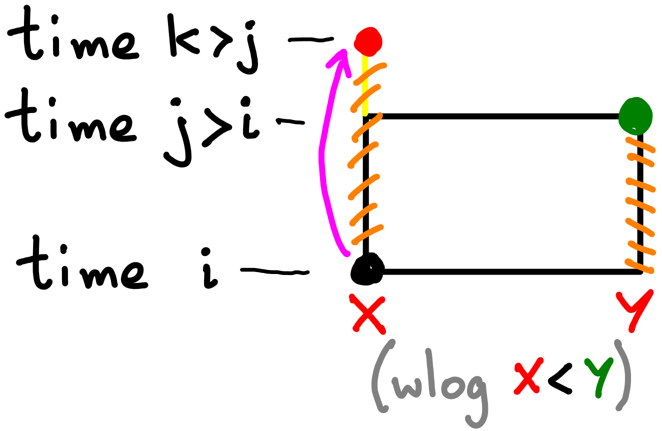




Claim:
new priorities won't violate heap property

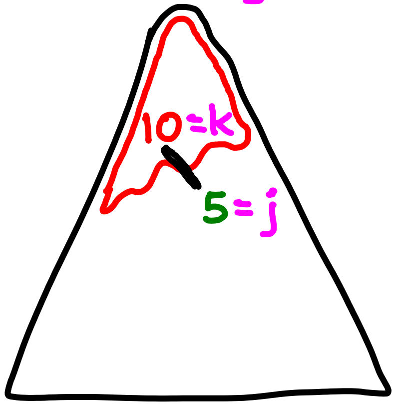


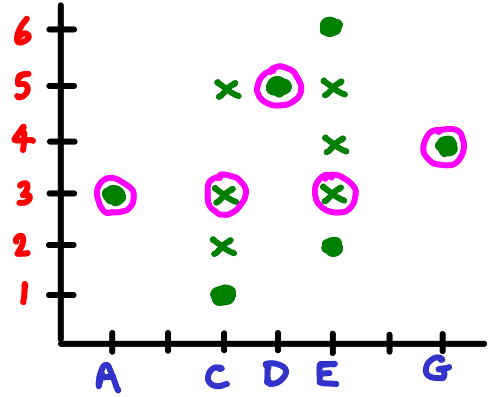
Proof by contradiction: suppose ∃ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



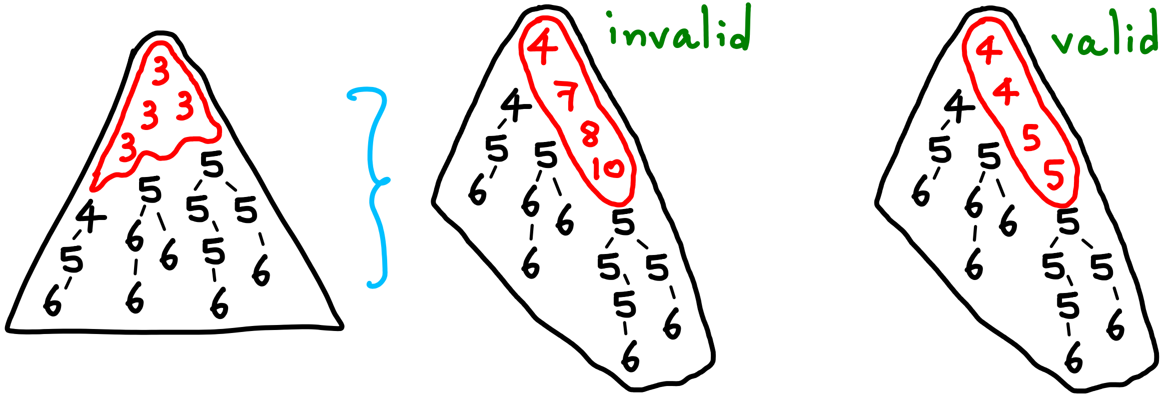
We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

↳ so left & right sides of box are empty

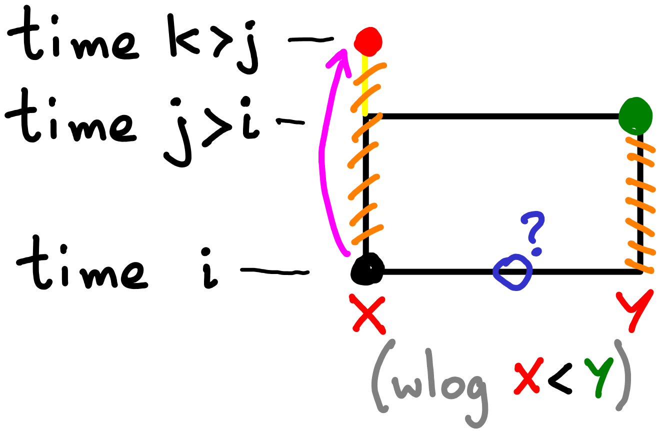




Claim:
new priorities won't violate heap property



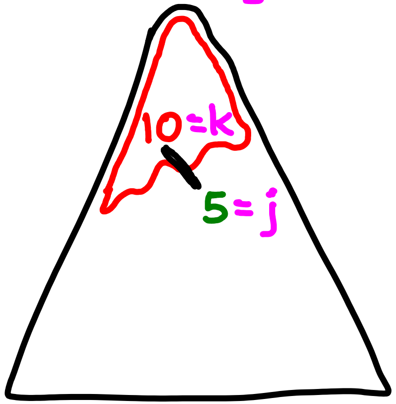
Proof by contradiction: suppose ∃ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$

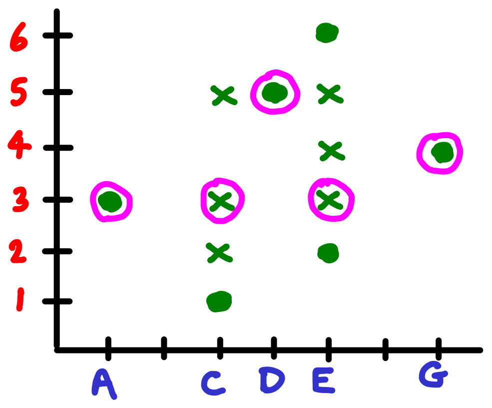


We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

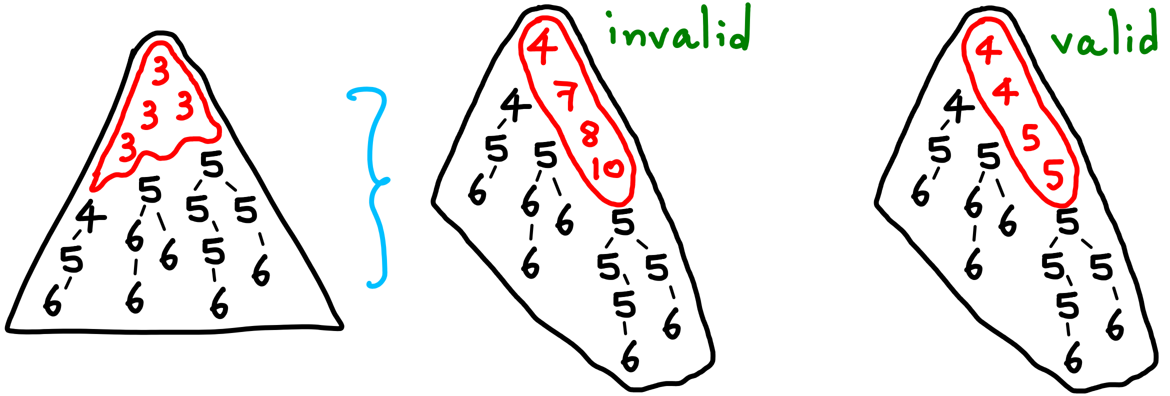
↪ so left & right sides of box are empty

What about bottom side?

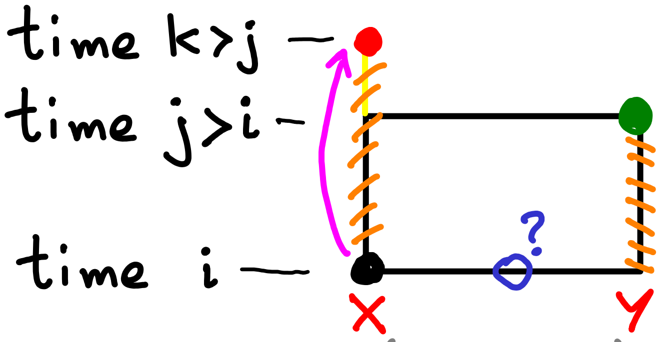




Claim:
new priorities won't violate heap property



Proof by contradiction: suppose $\exists$ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$

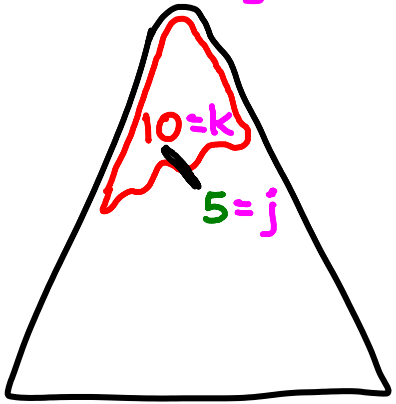


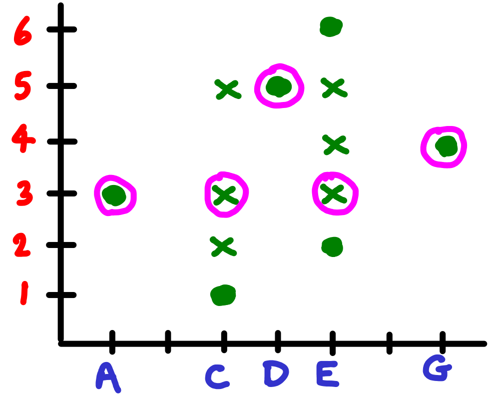
We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

↳ so left & right sides of box are empty

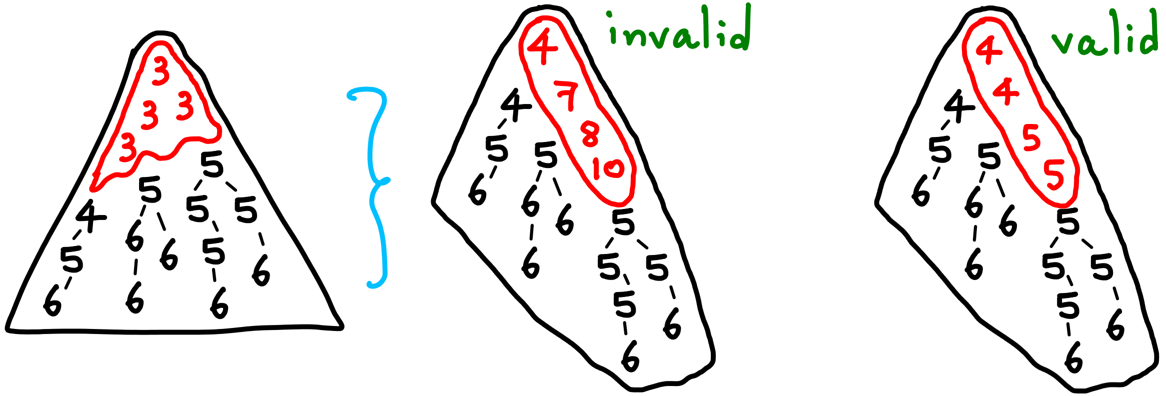
What about bottom side?

Would need X<key<Y but then...

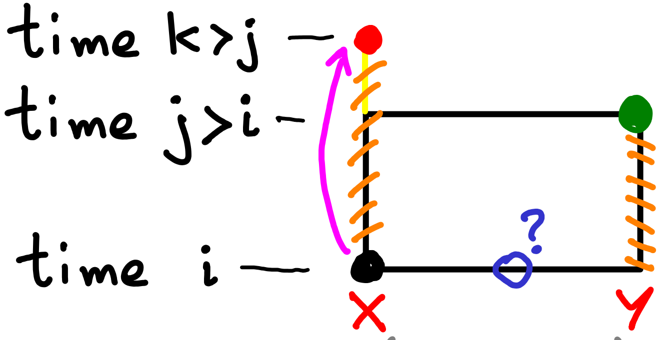




Claim:
new priorities won't violate heap property



Proof by contradiction: suppose ∃ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



(wlog X<Y)

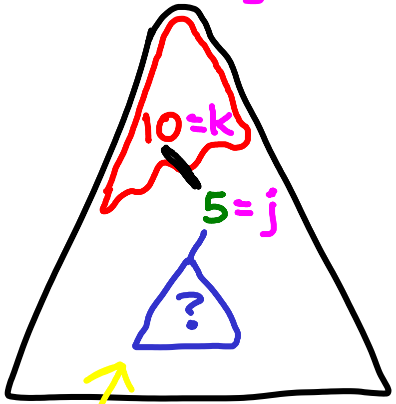
We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

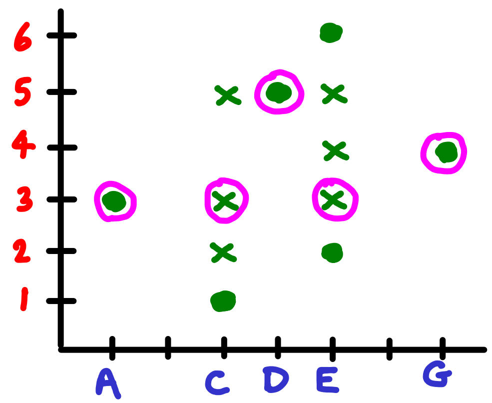
↳ so left & right sides of box are empty

What about bottom side?

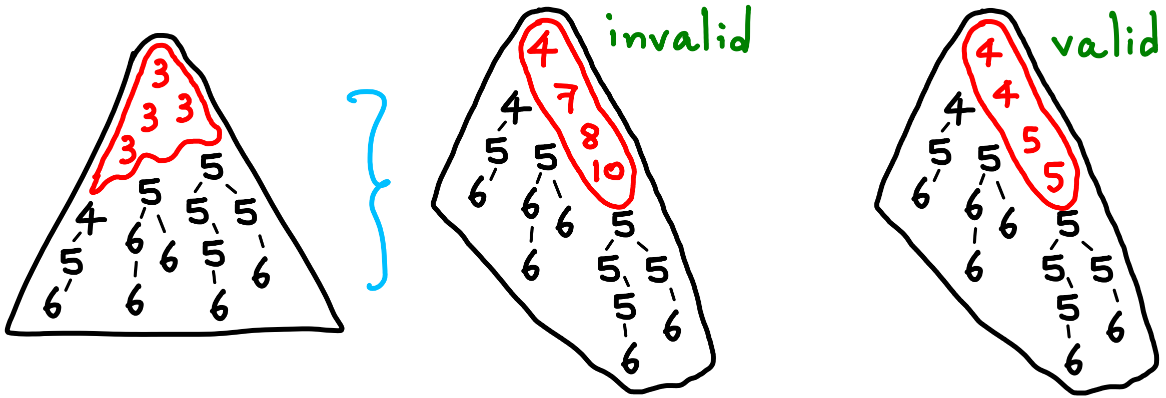
Would need X<key<Y but then...

...key is in left subtree of Y (at time i)

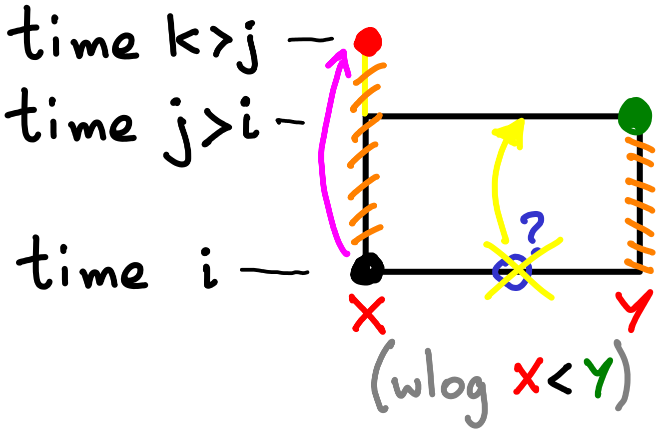




Claim:
new priorities won't violate heap property



Proof by contradiction: suppose ∃ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

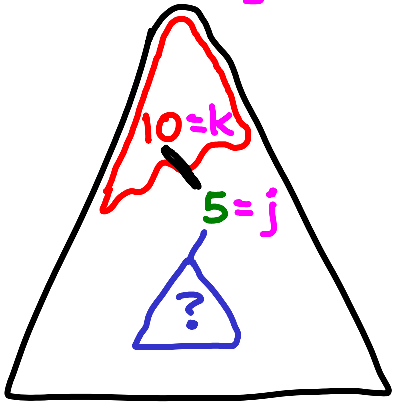
↳ so left & right sides of box are empty

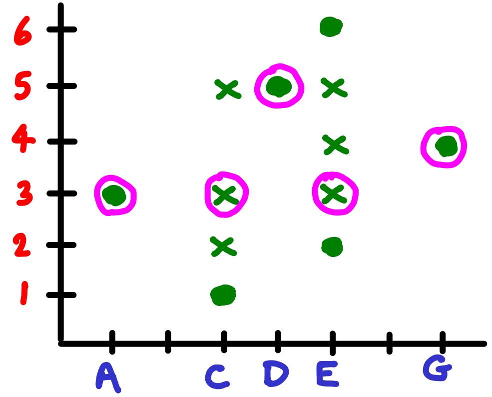
What about bottom side?

Would need X<key<Y but then...

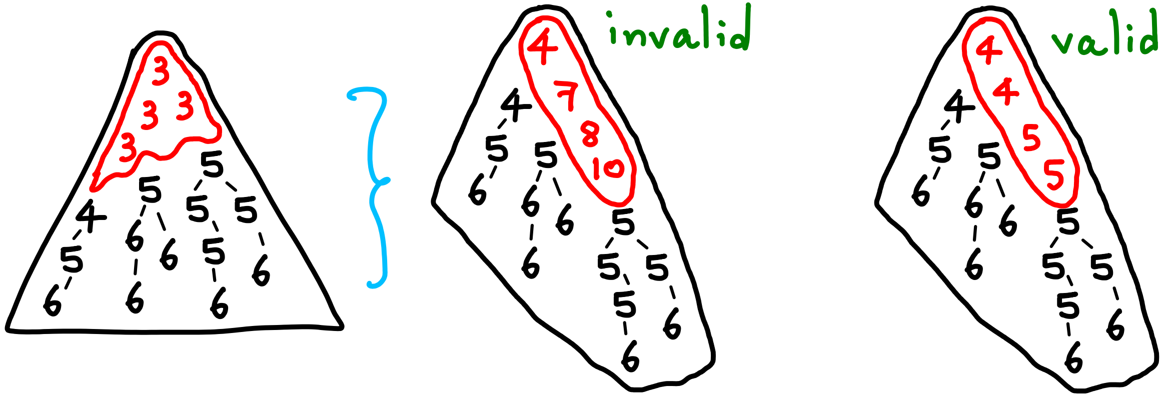
...key is in left subtree of Y, so (unchanged) priority ≥ j (> i)

CONTRADICTION

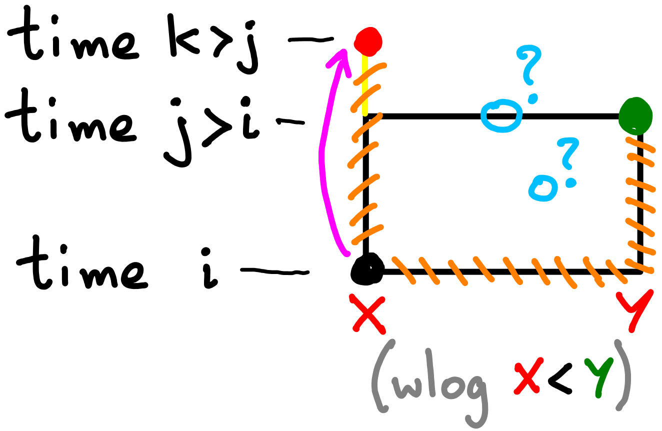




Claim:
new priorities won't violate heap property



Proof by contradiction: suppose $\exists$ edge X—Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

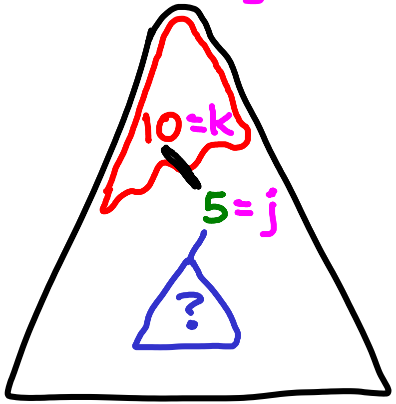
↳ so left & right sides of box are empty

What about bottom side?

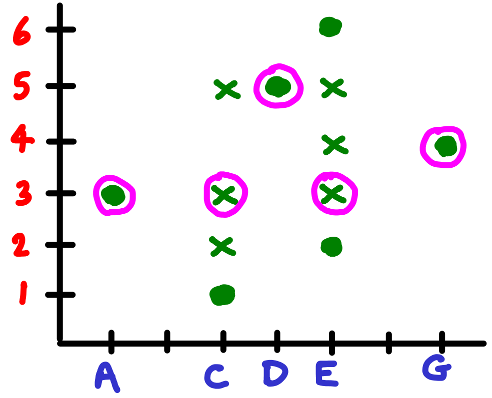
Would need X<key<Y but then...

...key is in left subtree of Y, so (unchanged) priority $\geq j$ $(> i)$

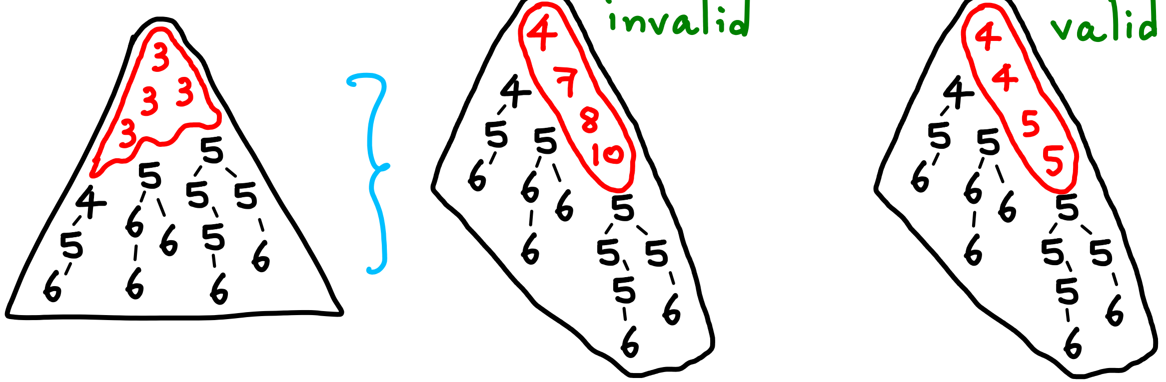
↳ so bottom side is empty.



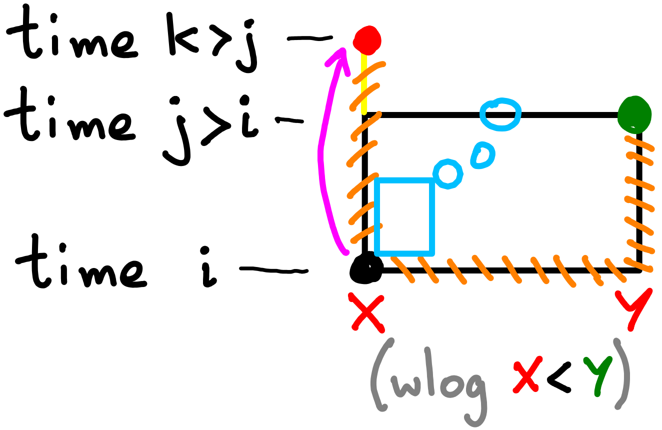
Top side? Inside?



Claim:
new priorities won't violate heap property



Proof by contradiction: suppose ∃ edge X–Y s.t. $\underbrace{\text{new priority}(X)}_{k} > \underbrace{\text{priority}(Y)}_{j}$



We know $\overbrace{\text{priority}(Y)}^{j} > \overbrace{\text{old priority}(X)}^{i}$

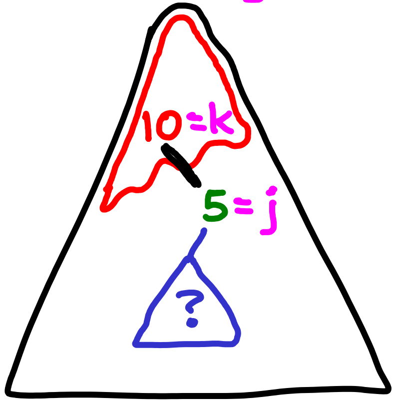
↳ so left & right sides of box are empty

What about bottom side?

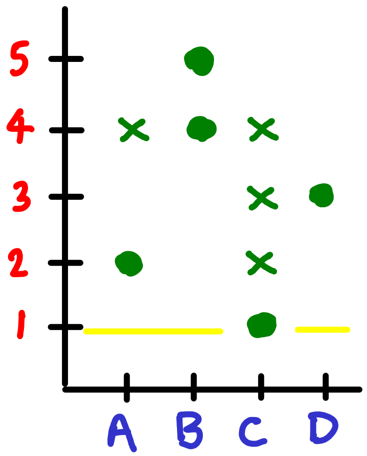
Would need X<key<Y but then...

...key is in left subtree of Y, so (unchanged) priority ≥j (> i)

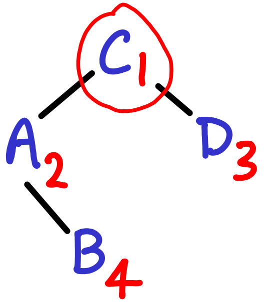
↳ so bottom side is empty.



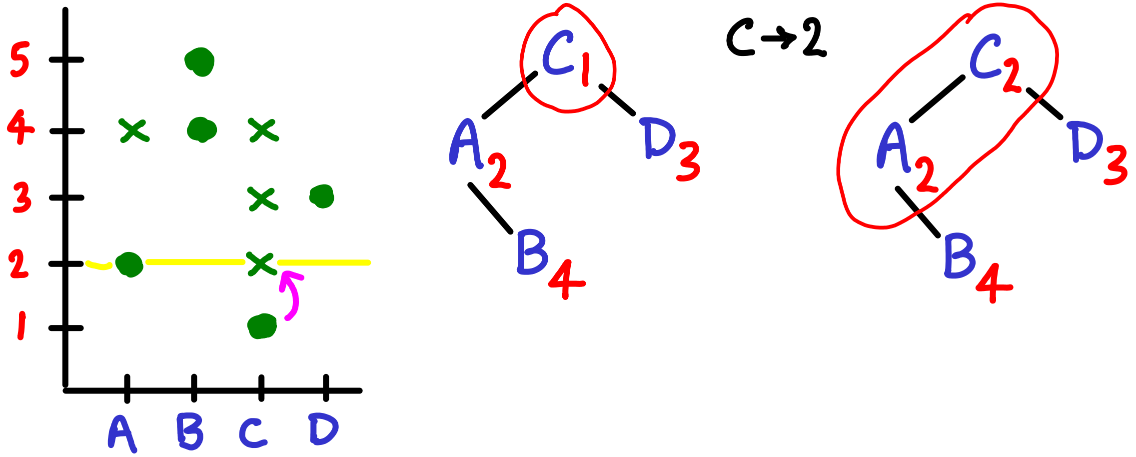
Now, if we place a point on top side, or inside, we will get an empty rectangle □



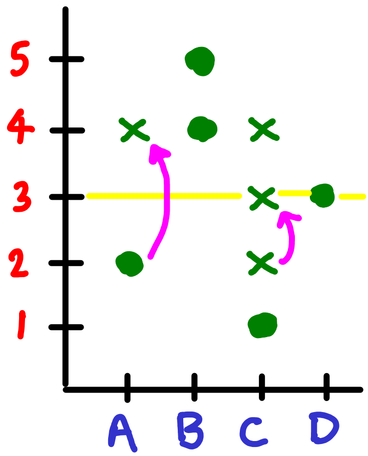
5
4
3
2
1
A
B
C
D

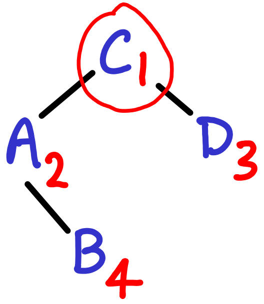


C1
A2
D3
B4

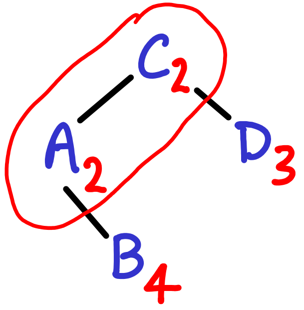


5
4
3
2
1
A B C D
C1
A2
D3
B4
C→2
C2
A2
D3
B4

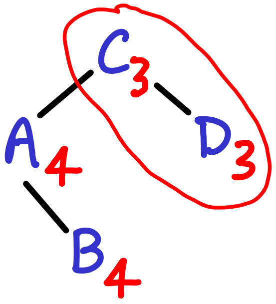


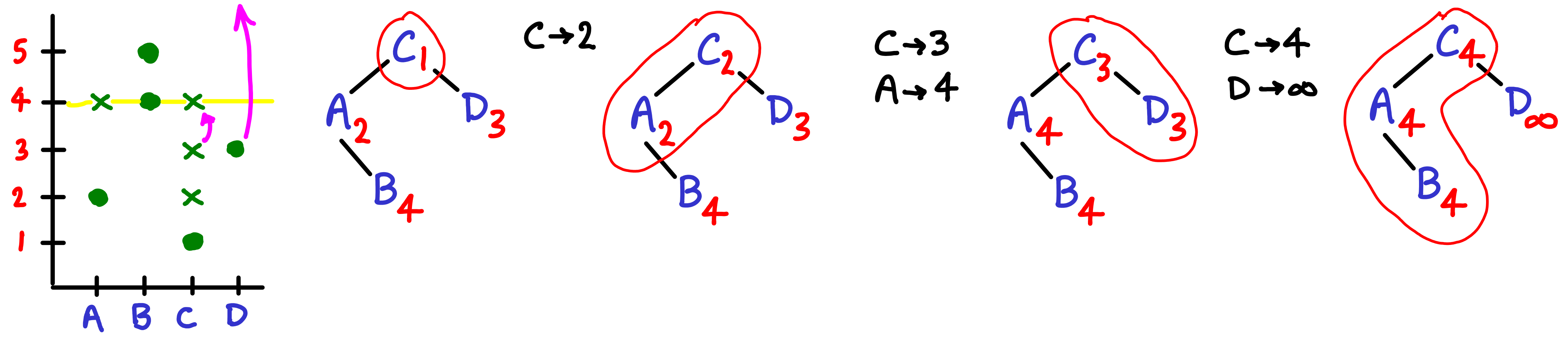


$C \to 2$

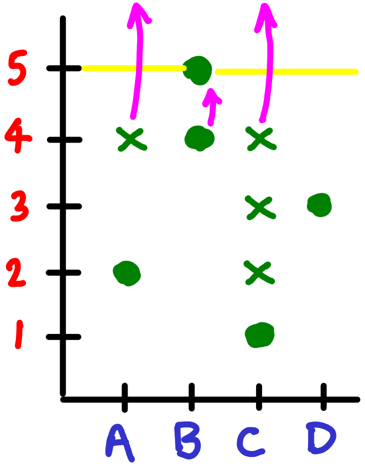


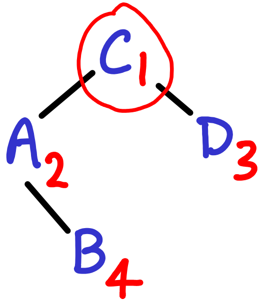
$C \to 3$
$A \to 4$



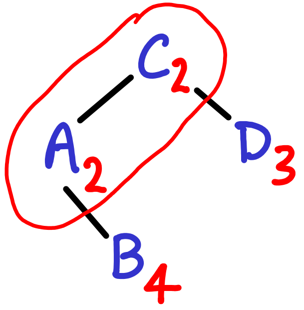


5
4
3
2
1
A B C D
C1
A2
D3
B4
C→2
C2
A2
D3
B4
C→3
A→4
C3
A4
D3
B4
C→4
D→∞
C4
A4
D∞
B4

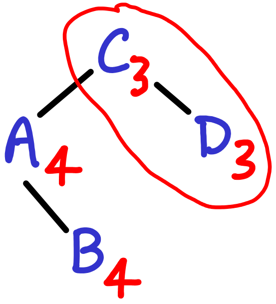




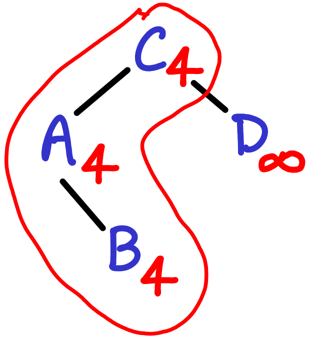
$C \to 2$



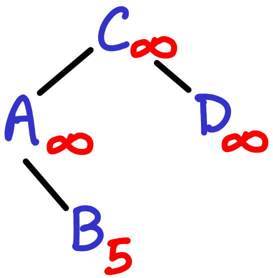
$C \to 3$
$A \to 4$

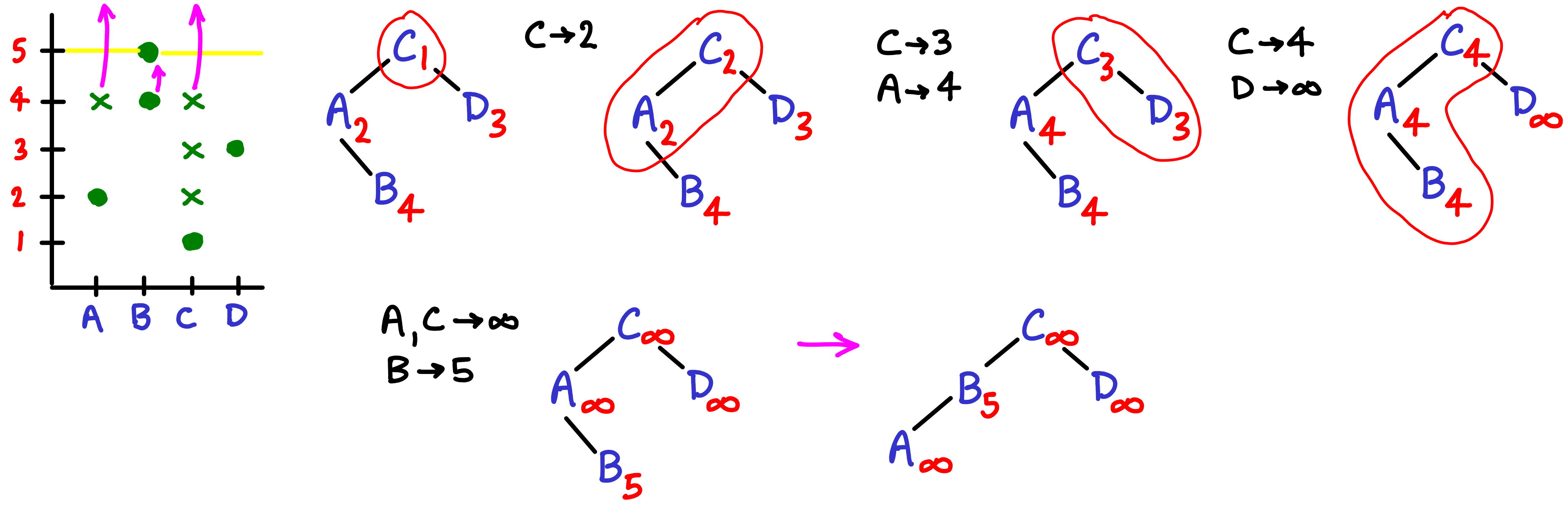


$C \to 4$
$D \to \infty$

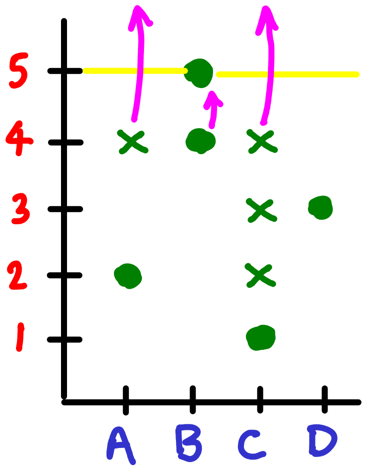


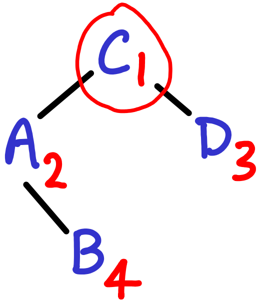
$A, C \to \infty$
$B \to 5$



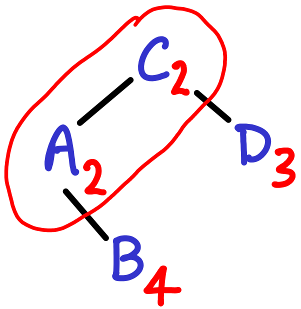


5
4
3
2
1
A B C D
C→2
C→3
A→4
C→4
D→∞
A, C→∞
B→5

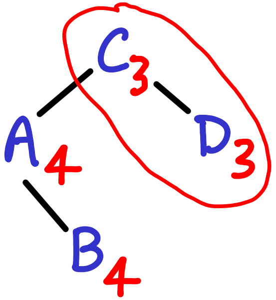




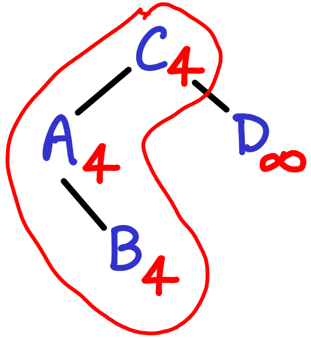
$C \to 2$



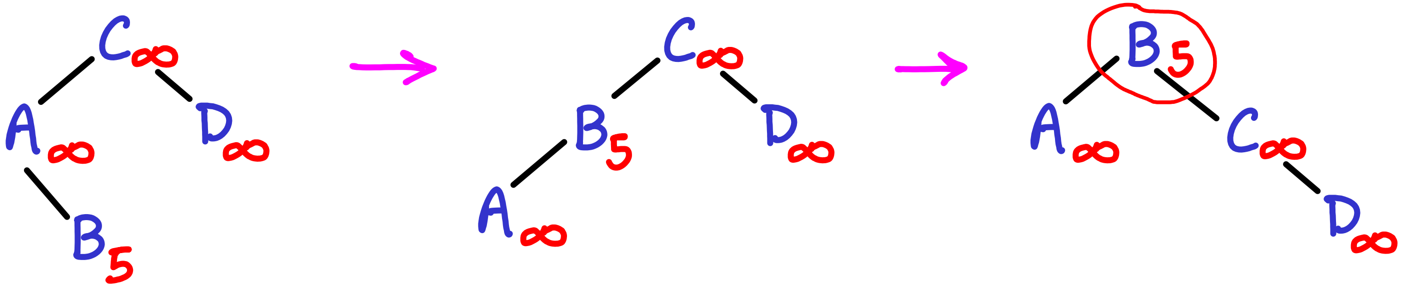
$C \to 3$
$A \to 4$

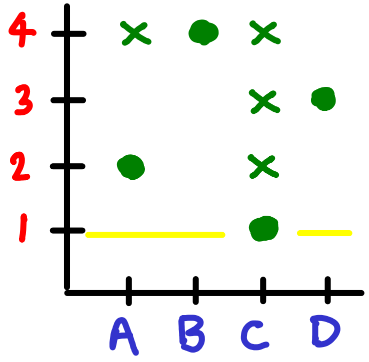


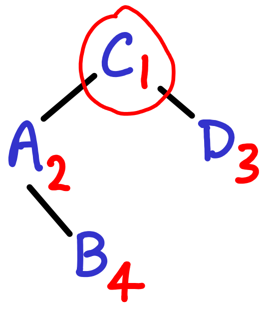
$C \to 4$
$D \to \infty$



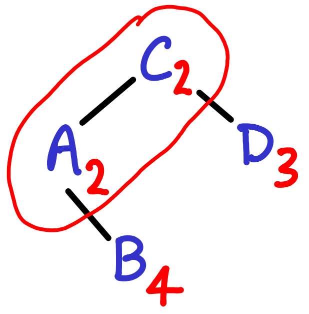
$A, C \to \infty$
$B \to 5$



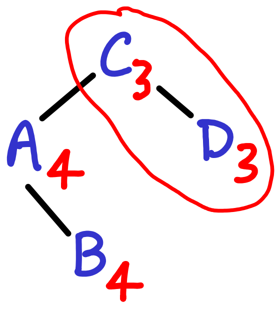




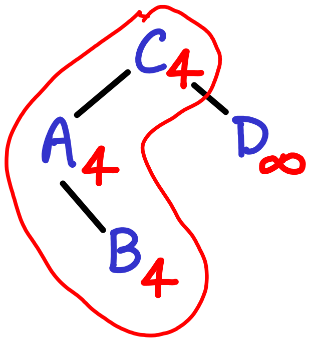
$C \to 2$

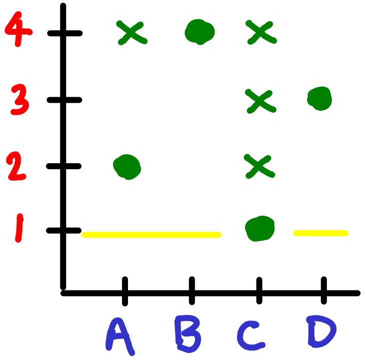


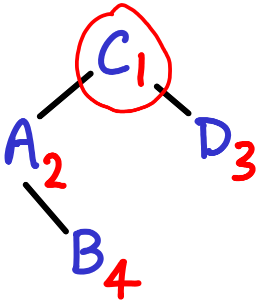
$C \to 3$
$A \to 4$



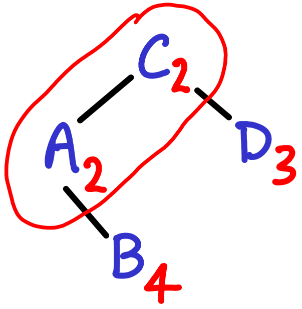
$C \to 4$
$D \to \infty$



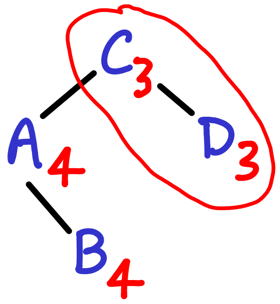




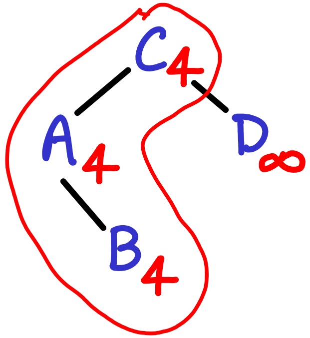
$C \to 2$

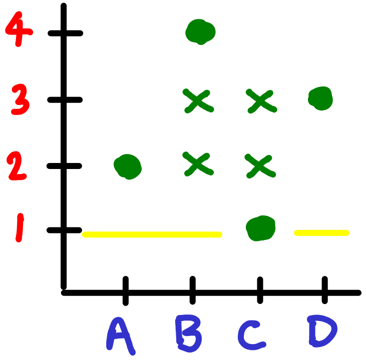


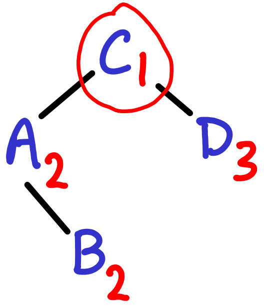
$C \to 3$
$A \to 4$

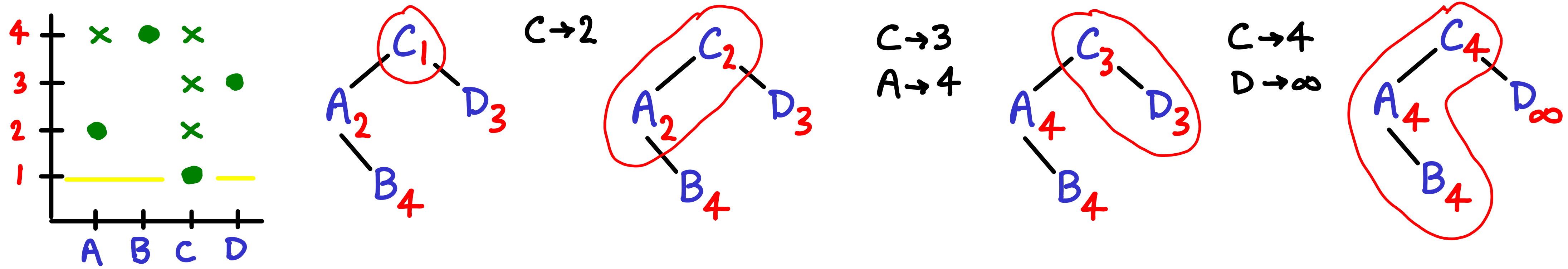


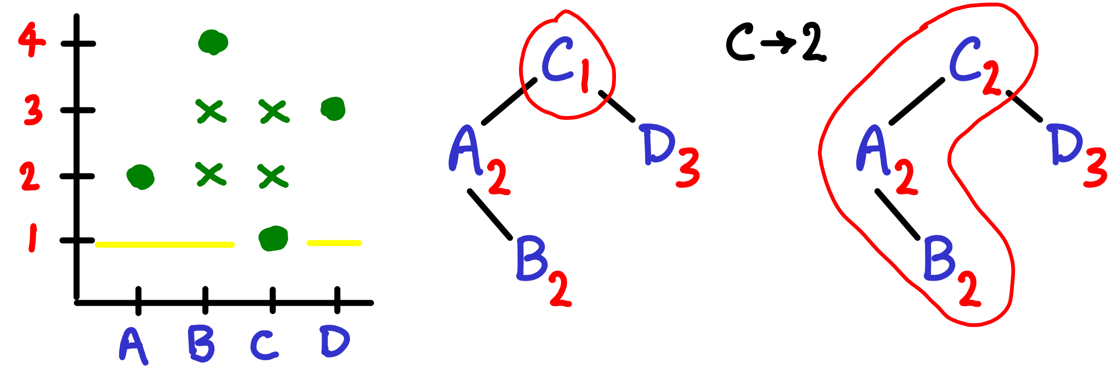
$C \to 4$
$D \to \infty$

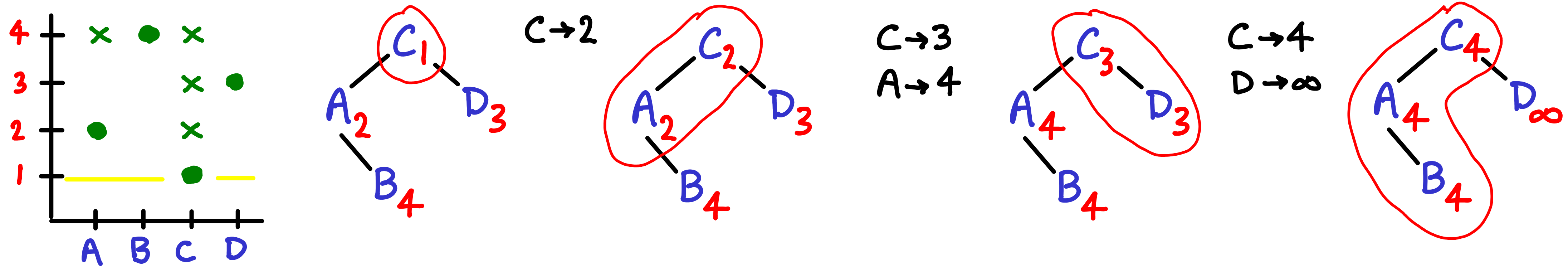


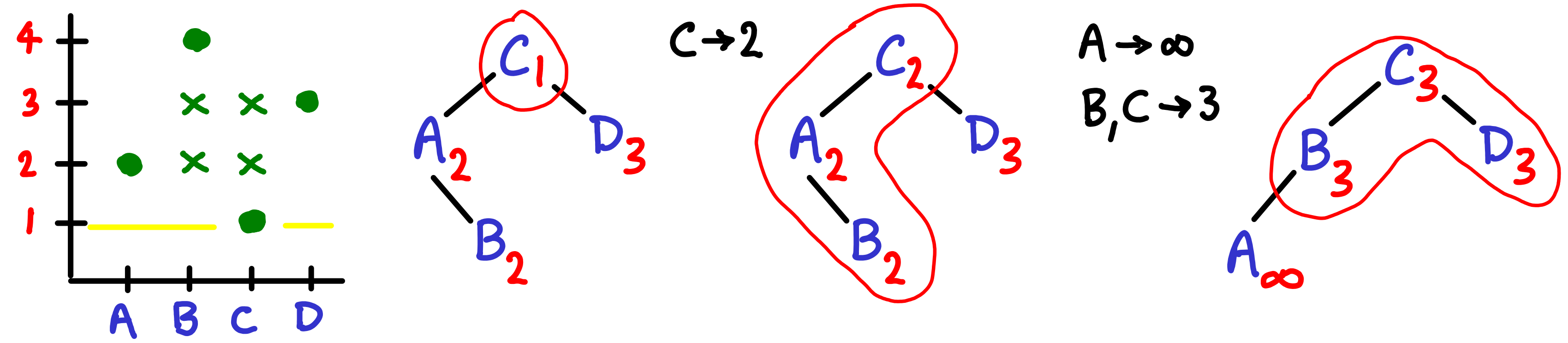


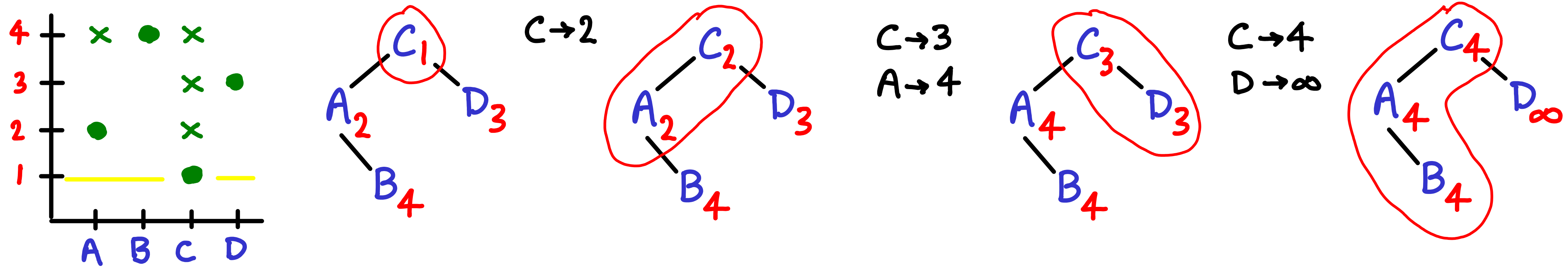



4
3
2
1
A B C D
C1
A2
D3
B4
C→2
C2
A2
D3
B4
C→3
A→4
C3
A4
D3
B4
C→4
D→∞
C4
A4
D∞
B4

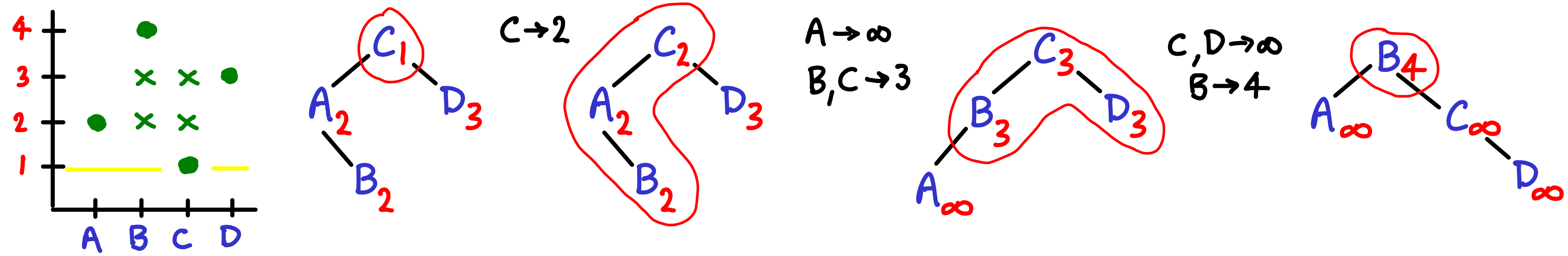

4
3
2
1
A B C D
C1
A2
D3
B2
C→2
C2
A2
D3
B2


4
3
2
1
A B C D
C1
A2
D3
B4
C→2
C2
A2
D3
B4
C→3
A→4
C3
A4
D3
B4
C→4
D→∞
C4
A4
D∞
B4

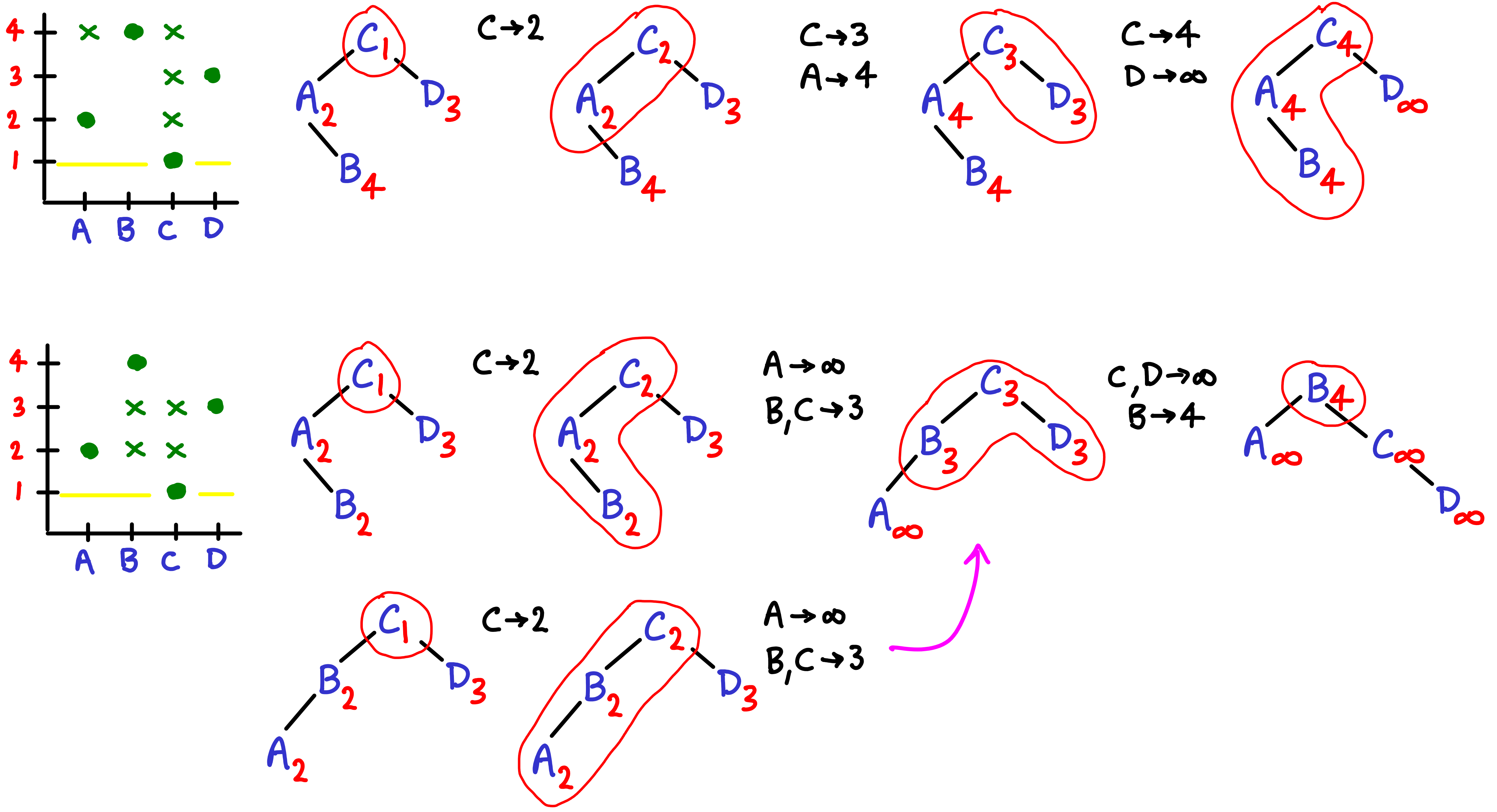

4
3
2
1
A B C D
C1
A2
D3
B2
C→2
C2
A2
D3
B2
A→∞
B,C→3
C3
B3
D3
A∞



4
3
2
1
A B C D
C1
A2
D3
B4
C→2
C2
A2
D3
B4
C→3
A→4
C3
A4
D3
B4
C→4
D→∞
C4
A4
D∞
B4



4
3
2
1
A B C D
C1
A2
D3
B2
C→2
C2
A2
D3
B2
A→∞
B,C→3
C3
B3
D3
A∞
C,D→∞
B→4
B4
A∞
C∞
D∞



4
3
2
1
A B C D
C→2
C→3
A→4
C→4
D→∞
4
3
2
1
A B C D
C→2
A→∞
B,C→3
C,D→∞
B→4
C→2
A→∞
B,C→3

Conclusion so far:

Every BST op sequence can be drawn as a special grid pattern and every valid grid pattern can be translated into a sequence of BST ops

Conclusion so far:

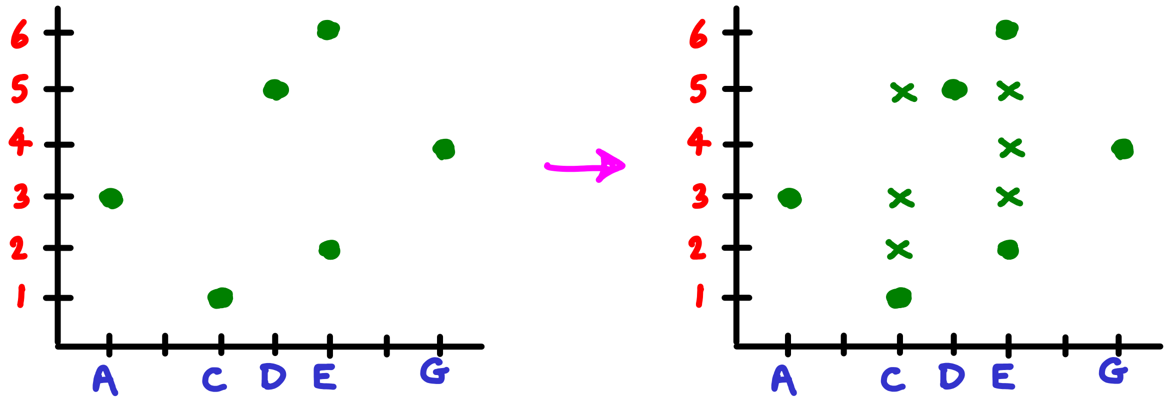
Every BST op sequence can be drawn as a special grid pattern
and every valid grid pattern can be translated into a sequence of BST ops

therefore

BST optimality (when we can modify the tree)

is related to

minimizing #added points to make a given grid pattern valid



Conclusion so far:

Every BST op sequence can be drawn as a special grid pattern

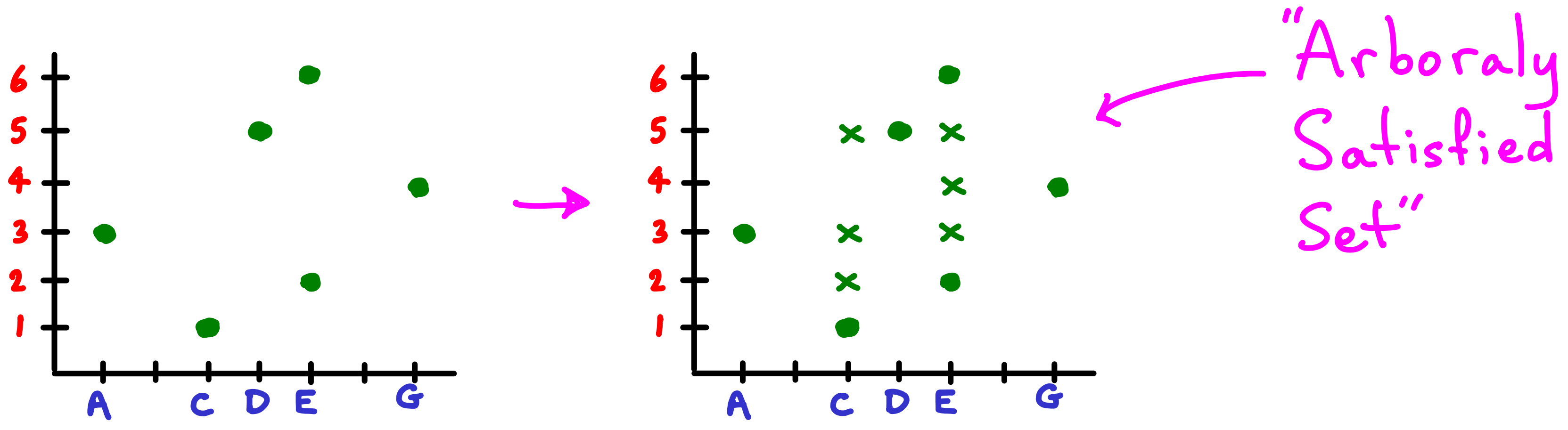
and every valid grid pattern can be translated into a sequence of BST ops

therefore

BST optimality (when we can modify the tree)

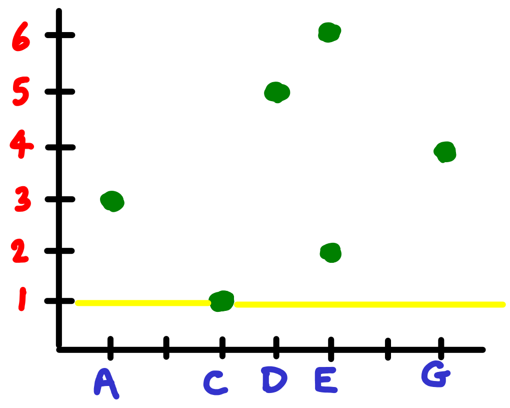
$\updownarrow$

minimizing #added points to make a given grid pattern valid



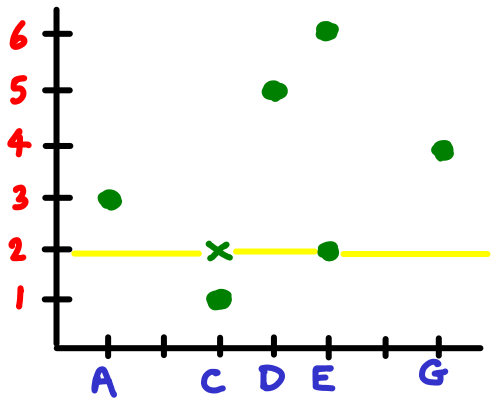
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



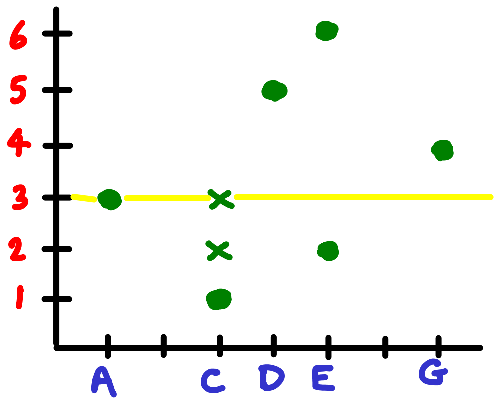
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



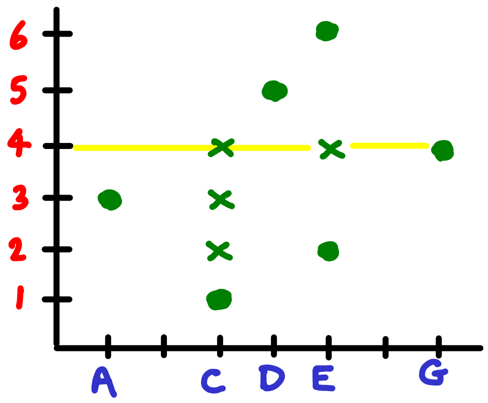
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



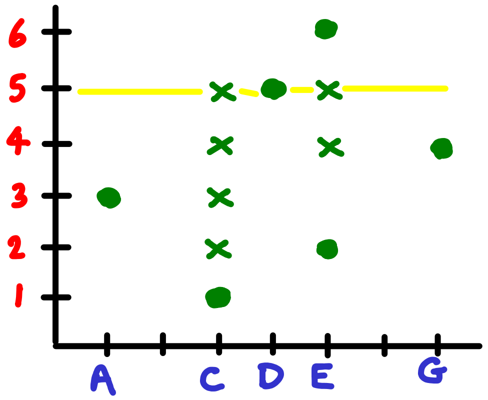
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



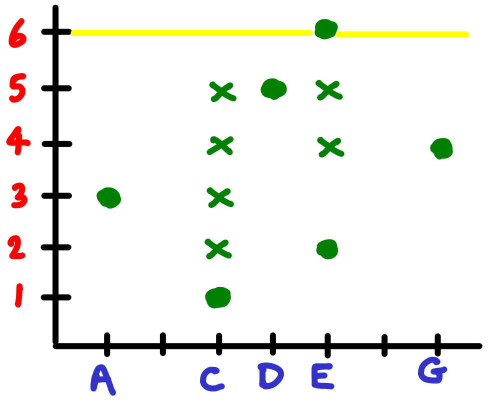
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



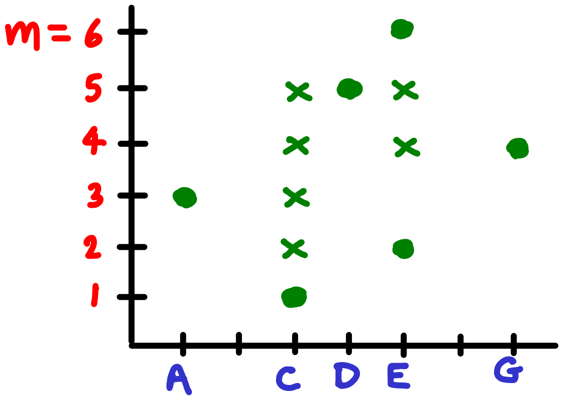
Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



Greedy algorithm (online: doesn't even need to know sequence)

Sweep up, add points only when necessary



Conjectured $O(OPT)$ or $OPT + O(m)$

OPT : min # steps possible, with knowledge of sequence