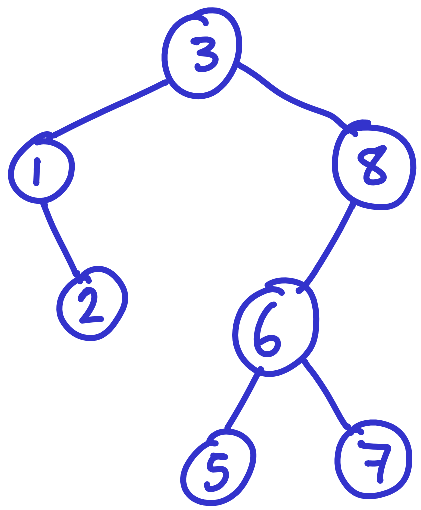
BINARY SEARCH TREES – BUILT RANDOMLY



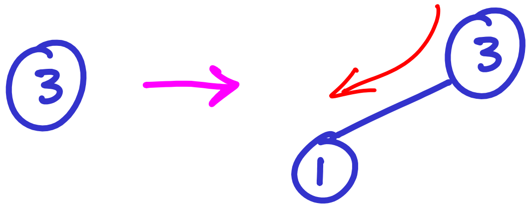
Insert n elements into a BST

in the order that they're given.

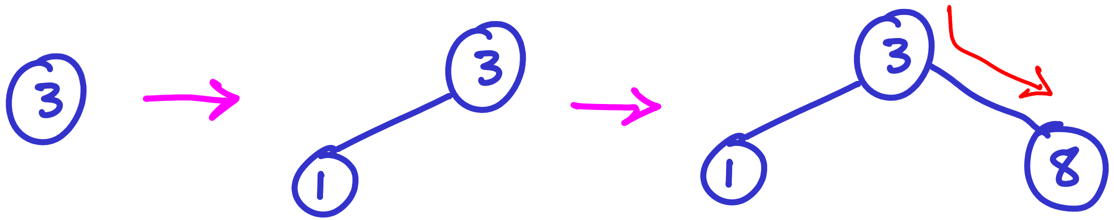
Given array of elements : 3 1 8 2 6 7 5

3

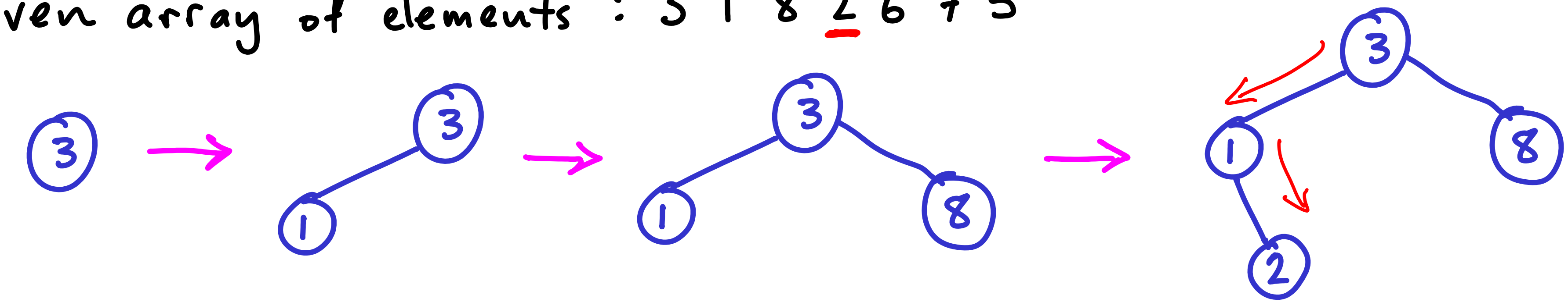
Given array of elements : 3 1 8 2 6 7 5



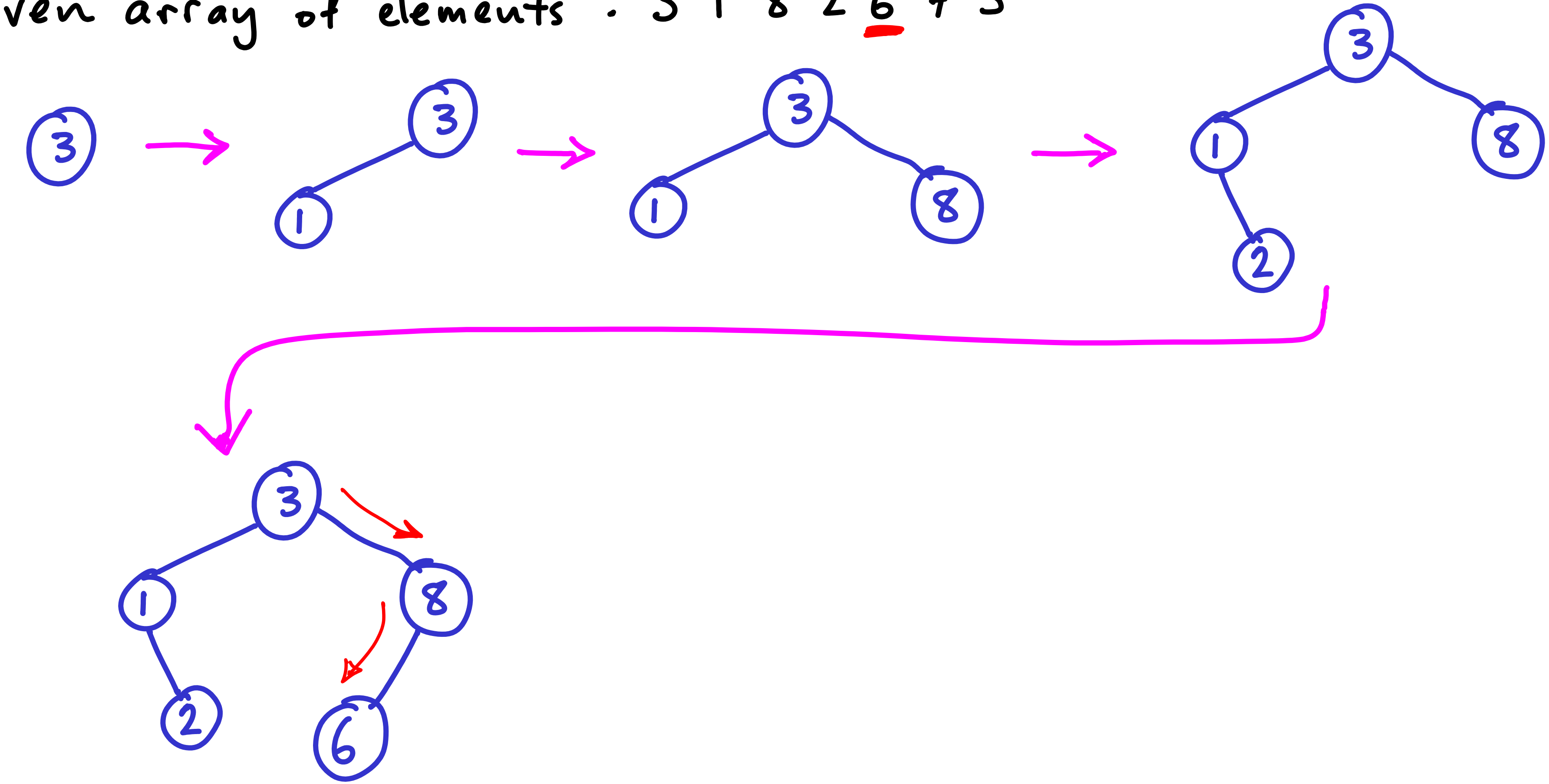
Given array of elements : 3 1 8 2 6 7 5



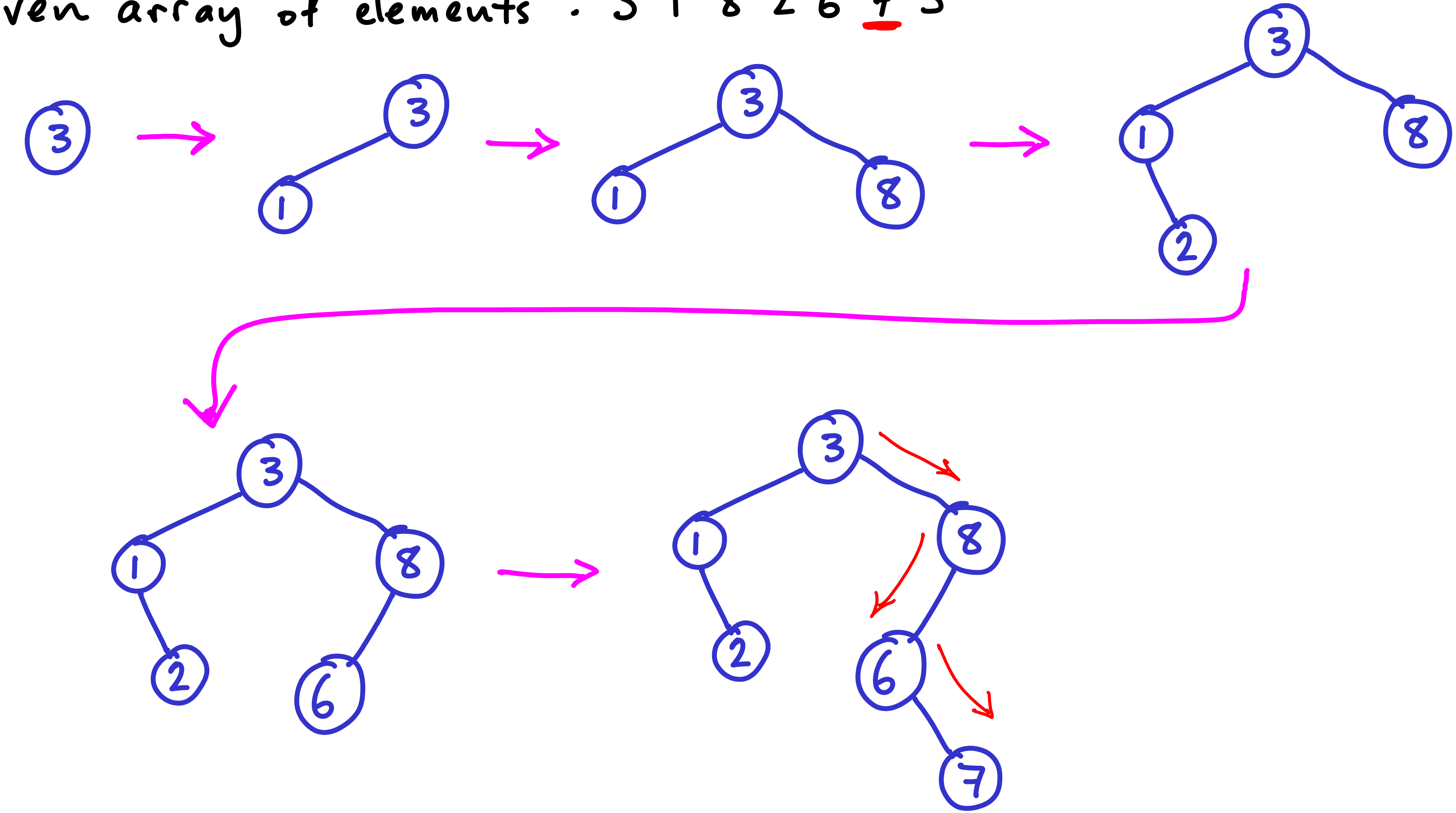
Given array of elements : 3 1 8 2 6 7 5



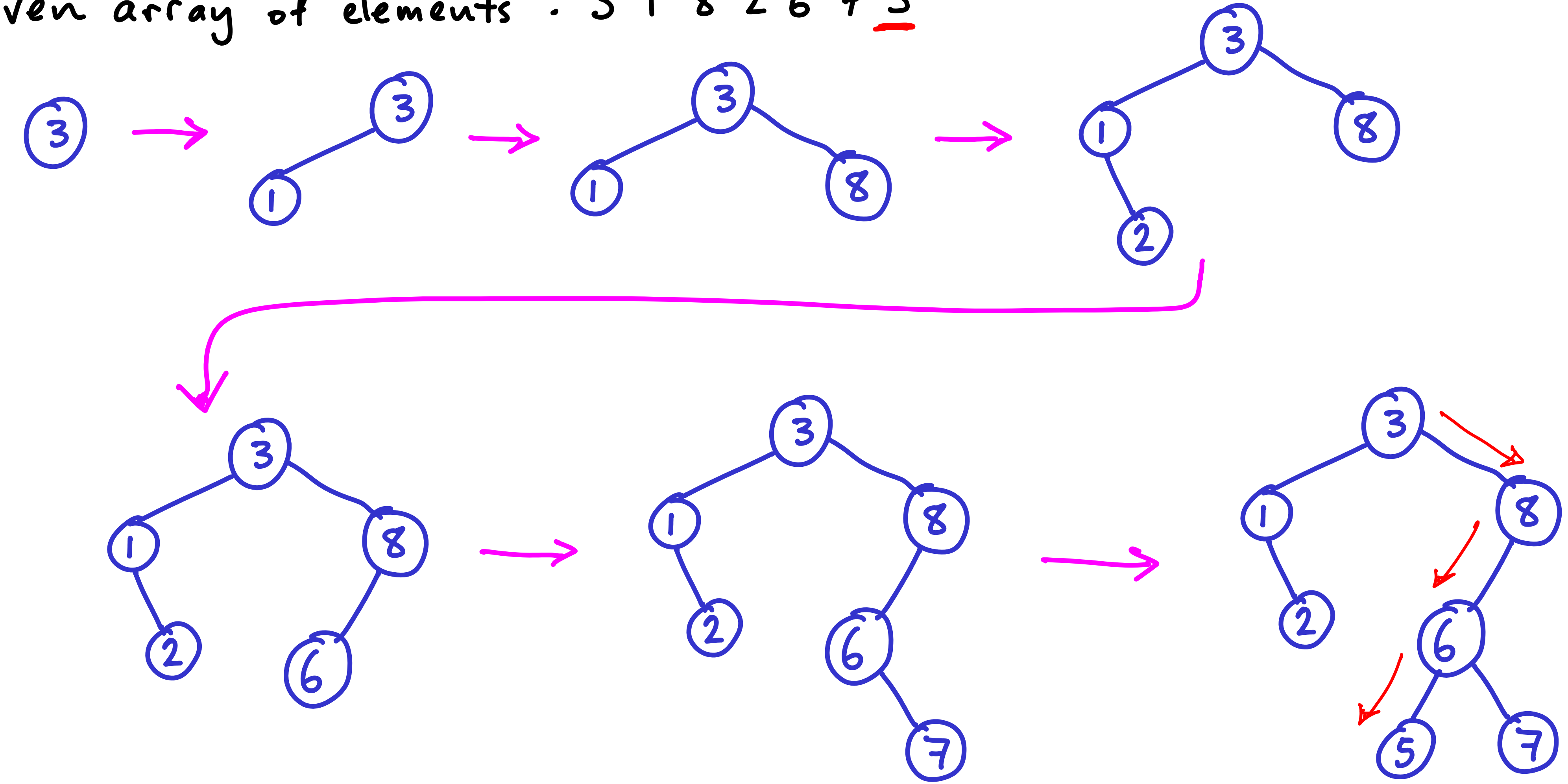
Given array of elements : 3 1 8 2 6 7 5



Given array of elements : 3 1 8 2 6 7 5



Given array of elements : 3 1 8 2 6 7 5

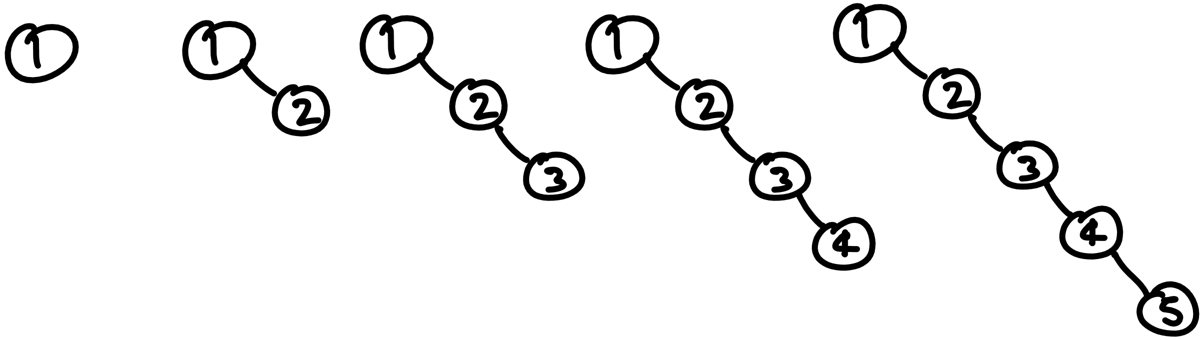


- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

↳ already sorted input, reverse-sorted, nearly sorted...

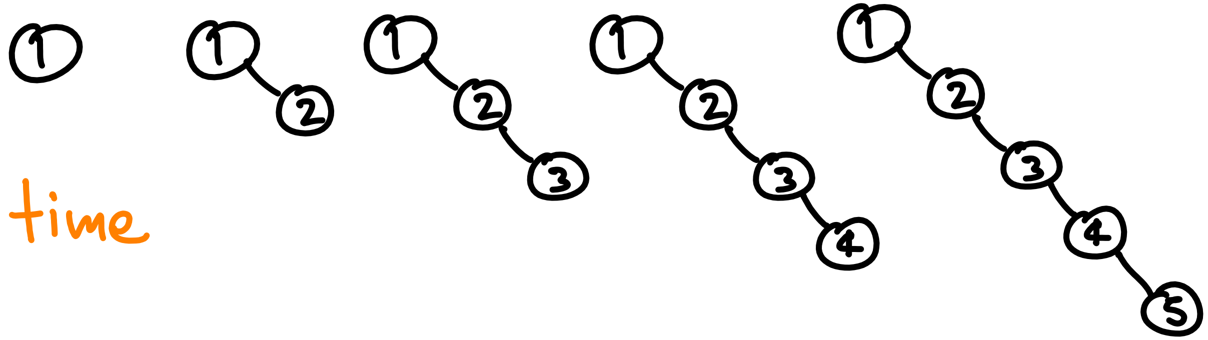
1 2 3 4 5 →



- What is the worst-case time complexity, and why?
- How unbalanced could the tree be?

↳ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 $\longrightarrow$

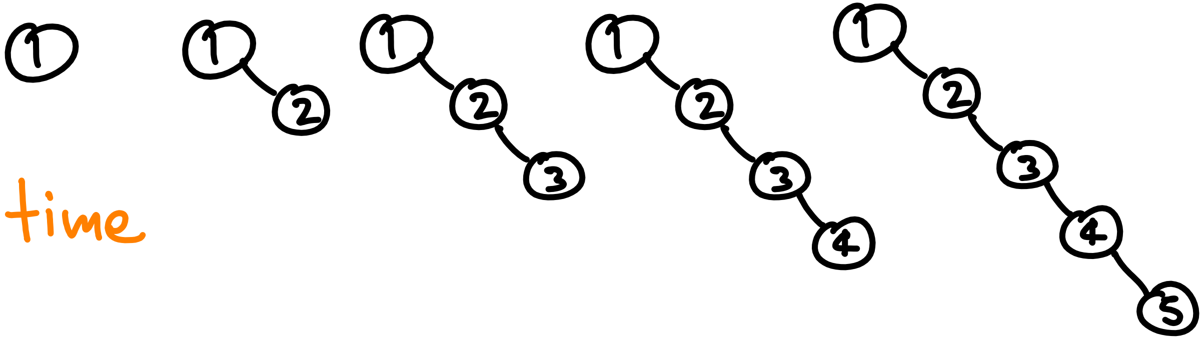


$O(n)$ depth, $O(n^2)$ time

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be?

↳ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 →



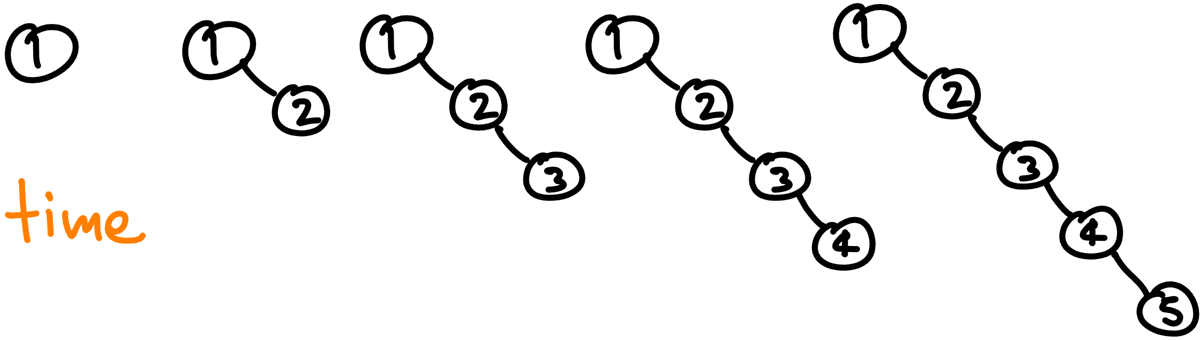
$O(n)$ depth, $O(n^2)$ time

- What would be ideal?

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

↳ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 →



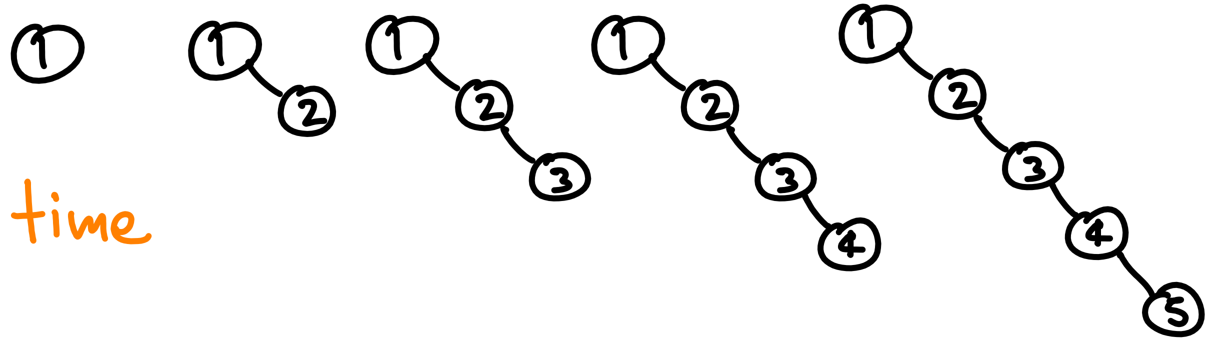
$O(n)$ depth, $O(n^2)$ time

- What would be ideal ? → balanced partition every time
- Worst-case time complexity $= \Omega(?)$

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

↳ already sorted input, reverse-sorted, nearly sorted...

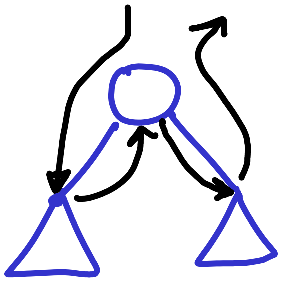
1 2 3 4 5 →



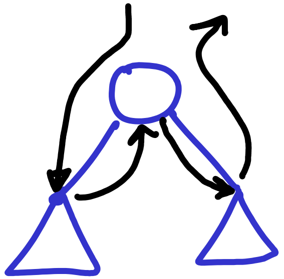
$O(n)$ depth, $O(n^2)$ time

- What would be ideal ? → balanced partition every time
- Worst-case time complexity $= \Omega(?)$

Hint: after constructing the BST, what does an in-order traversal give?



Constructing a BST essentially sorts data so the sorting lower bound applies



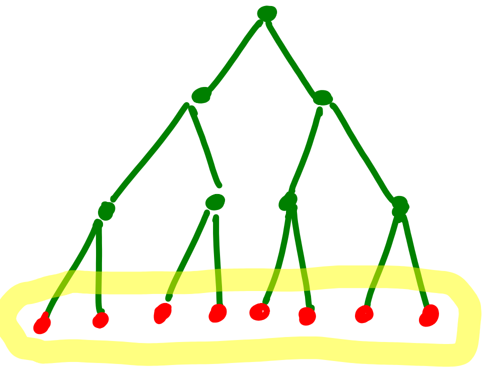
Constructing a BST essentially sorts data
so the sorting lower bound applies

worst case $\Omega(n \log n)$

Constructing a BST essentially sorts data so the sorting lower bound applies } worst case $\Omega(n \log n)$

But we can make an even stronger claim

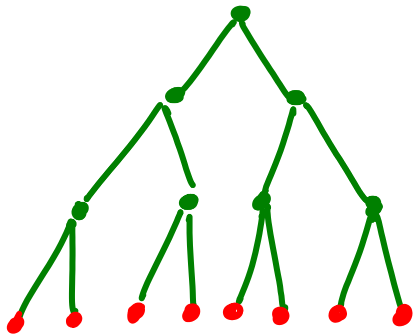
Constructing a BST essentially sorts data so the sorting lower bound applies $\Big\}$ worst case $\Omega(n\log n)$



In a balanced tree, $\sim\frac{n}{2}$ nodes have depth $\sim\log n$

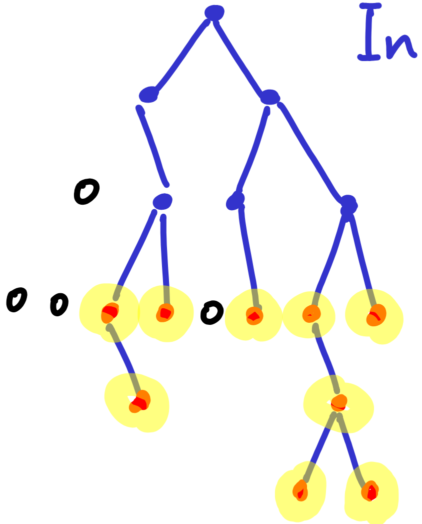
so they take $\Omega(n\log n)$ time to insert.

Constructing a BST essentially sorts data so the sorting lower bound applies } worst case $\Omega(n\log n)$



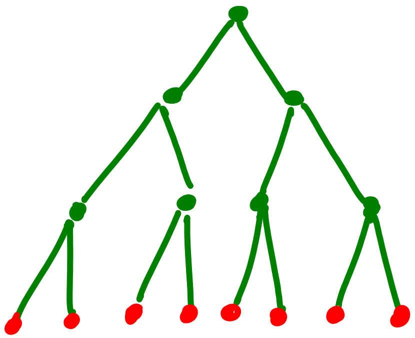
In a balanced tree, $\sim\frac{n}{2}$ nodes have depth $\sim\log n$

so they take $\Omega(n\log n)$ time to insert.



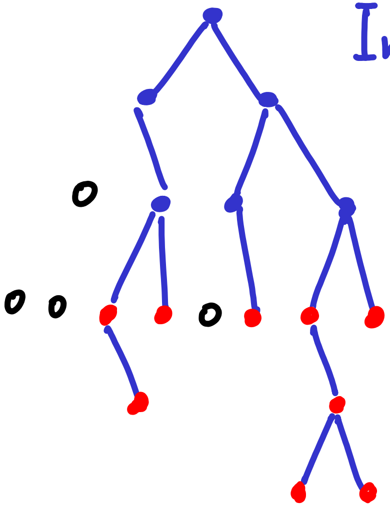
In any other tree we can find $\geq\frac{n}{2}$ nodes with depth $\geq\log n$

Constructing a BST essentially sorts data so the sorting lower bound applies $\Big\}$ worst case $\Omega(n \log n)$



In a balanced tree, $\sim\frac{n}{2}$ nodes have depth $\sim\log n$

so they take $\Omega(n\log n)$ time to insert.



In any other tree we can find $\geq\frac{n}{2}$ nodes with depth $\geq \log n$

$\hookrightarrow$ Every algorithm requires $\Omega(n\log n)$ time to construct any BST

Conclusion so far:

If lucky, $\Theta(n\log n)$ time

If unlucky, $\Theta(n^2)$ time

Conclusion so far:

If lucky, $\Theta(n \log n)$ time

If unlucky, $\Theta(n^2)$ time

Sounds like ...

Conclusion so far:

If lucky, $\Theta(n \log n)$ time

If unlucky, $\Theta(n^2)$ time

Sounds like ... Quicksort

Stable quicksort

3 1 8 2 6 7 5

- use first elt to partition → 1 2 3 8 6 7 5

Stable quicksort

3 1 8 2 6 7 5

- use first elt to partition → 1 2 3 8 6 7 5
- repeat on each side

1 2 3 8 6 7 5

1 2 675 8

Stable quicksort

3 1 8 2 6 7 5

- use first elt to partition
- repeat on each side
- 3rd round

1 2 3 8 6 7 5

1 2 3 8 6 7 5

1 2 6 7 5 8

1 2 3 8 6 7 5

1 2 6 7 5 8

2 5 6 7

Stable quicksort

3 1 8 2 6 7 5

- use first elt to partition → 1 2 3 8 6 7 5

- repeat on each side

  1 2 3 8 6 7 5

  1 2 6 7 5 8

- 3rd round

  1 2 3 8 6 7 5

  1 2 6 7 5 8

  2 5 6 7

- 4th round

  1 2 3 8 6 7 5

  1 2 6 7 5 8

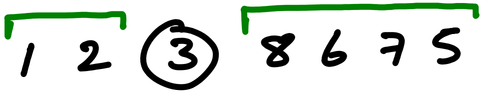
  2 5 6 7

  5 7

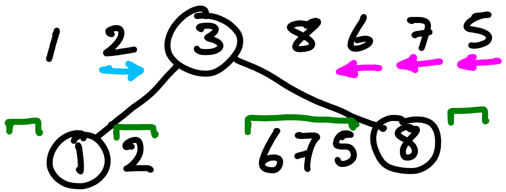
Stable quicksort

3 1 8 2 6 7 5

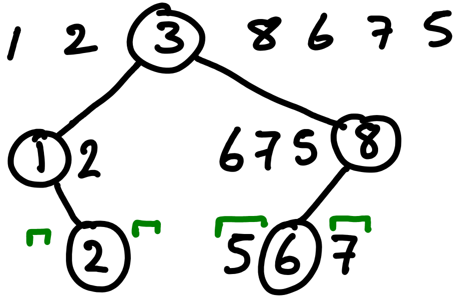
- use first elt to partition



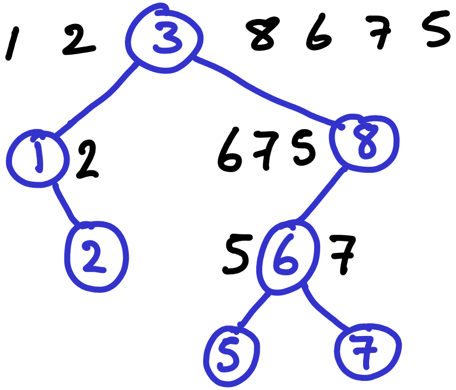
- repeat on each side



- 3rd round



- 4th round

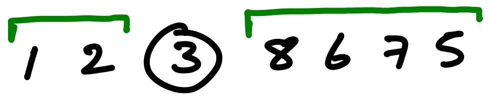


same tree as BST

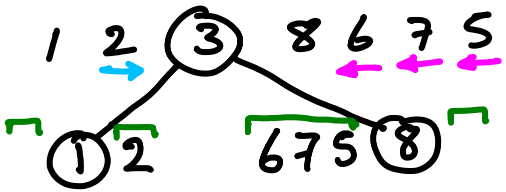
Stable quicksort

3 1 8 2 6 7 5

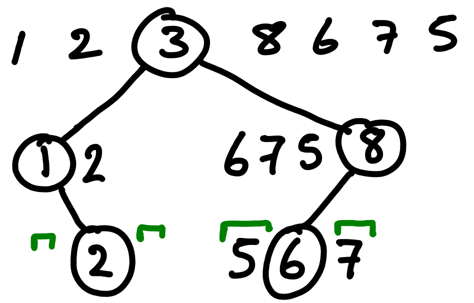
- use first elt to partition →



- repeat on each side

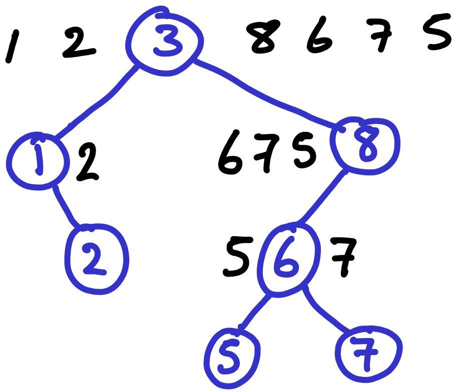


- 3rd round



- 4th round

same tree as BST



quicksort round1: compare all elts to 3

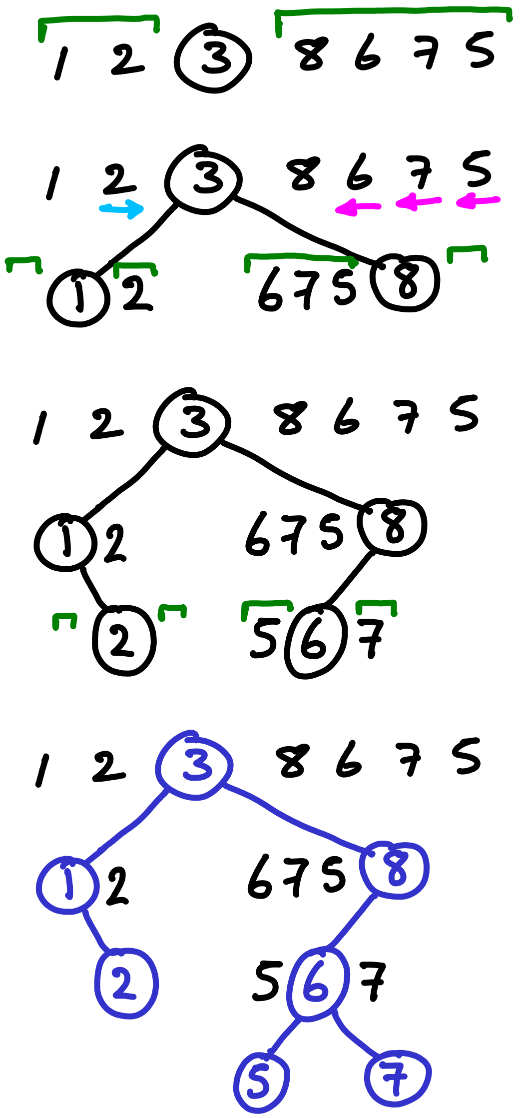
BST 3 = root; eventually all elts pass through.

Stable quicksort

③ 1 8 2 6 7 5

- use first elt to partition
- repeat on each side
- 3rd round
- 4th round

same tree as BST



quicksort round 1: compare all elts to ③

BST ③ = root; eventually all elts pass through.

quicksort: partitions into 2 groups
<③ & >③
each is independent

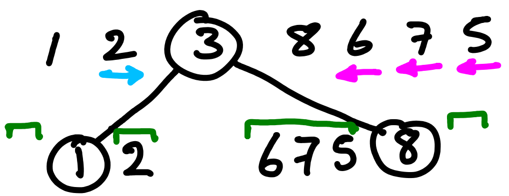
BST same

Stable quicksort

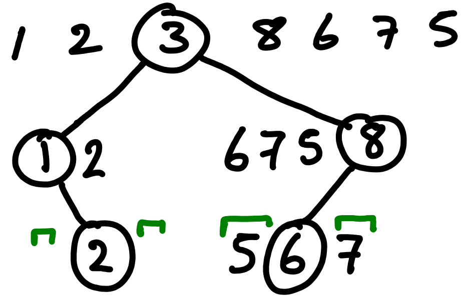
3 1 8 2 6 7 5

- use first elt to partition → 1 2 3 8 6 7 5

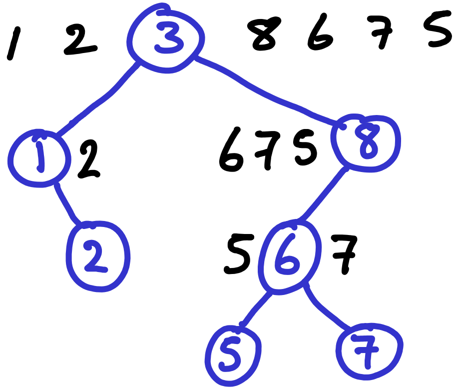
- repeat on each side



- 3rd round



- 4th round



same tree as BST

quicksort round 1: compare all elts to 3

BST 3 = root; eventually all elts pass through.

quicksort: partitions into 2 groups <3 & >3 each is independent

BST same

exactly same comparisons but in different order

Conclusion:

The expected time complexity of building a BST on random data is the same as for Quicksort: $O(n \log n)$

Conclusion:

The expected time complexity of building a BST on random data is the same as for Quicksort: O(nlogn)

Unfortunately this doesn't imply anything useful about the expected depth of a random BST.

In particular it doesn't imply expected O(logn) depth.
There exist trees that have much greater depth, but that can be constructed in O(nlogn) time.