CS2040S

Recitation 5
AY22/23S2

BST and AVL recap

If it’s helpful to you, here is a set of Jin’s old slides for BST and AVL recap: BST and AVL slides

Recitation goal

Show how augmentation is a powerful means for trees to "summarize" data of its subtrees.

Problem 1

Heights and Grades

- Given a set of students with heights and grades
- Implement an ADT to efficiently answer the question: “What is the average grade of all students taller than ___?'”
- For instance, the average CAP of all students taller than John is (4.2＋4.5＋3.6＋5.0＋3.9)/5＝4.24

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

ADT

Operation	Behaviour
`insert(name, height, grade)`	Inserts student into the dataset.
`findAverageGrade(name)`	Returns the average grade among all the students that are taller than the given student.

Problem 1.b.

How do you design a Data Structure (DS) that serves as an efficient implementation of the given ADT?

You may assume that name and height are unique.

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

Operation	Behaviour
`insert(name, height, grade)`	Inserts student into the dataset.
`findAverageGrade(name)`	Returns the average grade among all the students that are taller than the given student.

Naive Solution

Keep the data as it is.

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

Preprocessing Time:

Query Time:

Insertion Time:

Naive Solution

Keep the data as it is.

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

Preprocessing Time: O(1)

Query Time: O(n)

Insertion Time: O(1)

Naive Solution

Preprocessing Time:

Query Time:

Insertion Time:

Precompute Everything

Name	Height (cm)	Grade (CAP)	FindAverage (Name)
Charles	176	4.2	4.30
Bob	162	4.5	4.17
Mary	180	3.6	5.00
John	155	4.1	4.24
Wick	186	5.0	0
Alice	170	3.9	4.26

Naive Solution

Precompute Everything

Preprocessing Time: $O(n^2)$

Query Time: $O(1)$

Insertion Time: $O(n)$

Name	Height (cm)	Grade (CAP)	FindAverage (Name)
Charles	176	4.2	4.30
Bob	162	4.5	4.17
Mary	180	3.6	5.00
John	155	4.1	4.24
Wick	186	5.0	0
Alice	170	3.9	4.26

Somewhere in between

We want our solution somewhere in between the two extremes with

Preprocessing Time: O(n log n)

Query Time: O(log n)

Insertion Time: O(log n)

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

Operation	Behaviour
`insert(name, height, grade)`	Inserts student into the dataset.
`findAverageGrade(name)`	Returns the average grade among all the students that are taller than the given student.

Topics Covered in Lecture

What are the topics covered in your lecture related to augmented BST?

Topics Covered in Lecture

What are the topics covered in your lecture related to augmented BST?

(1) Order Statistics
(2) Interval Trees
(3) Orthogonal Range Searching

(1) Order Statistics
(2) Interval Trees
(3) Orthogonal Range Searching

Which one of these is similar to our current problem?

Name	Height (cm)	Grade (CAP)
Charles	176	4.2
Bob	162	4.5
Mary	180	3.6
John	155	4.1
Wick	186	5.0
Alice	170	3.9

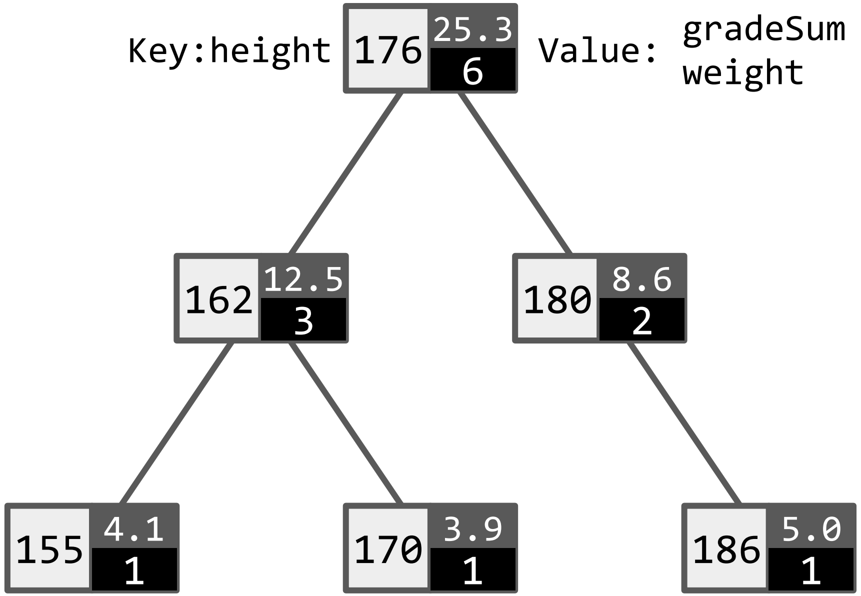
Operation	Behaviour
`insert(name, height, grade)`	Inserts student into the dataset.
`findAverageGrade(name)`	Returns the average grade among all the students that are taller than the given student.

2 stage query

nameTable

Key	Value
"Charles"	176
"Bob"	162
"Mary"	180
"John"	155
"Wick"	186
"Alice"	170

heightTree



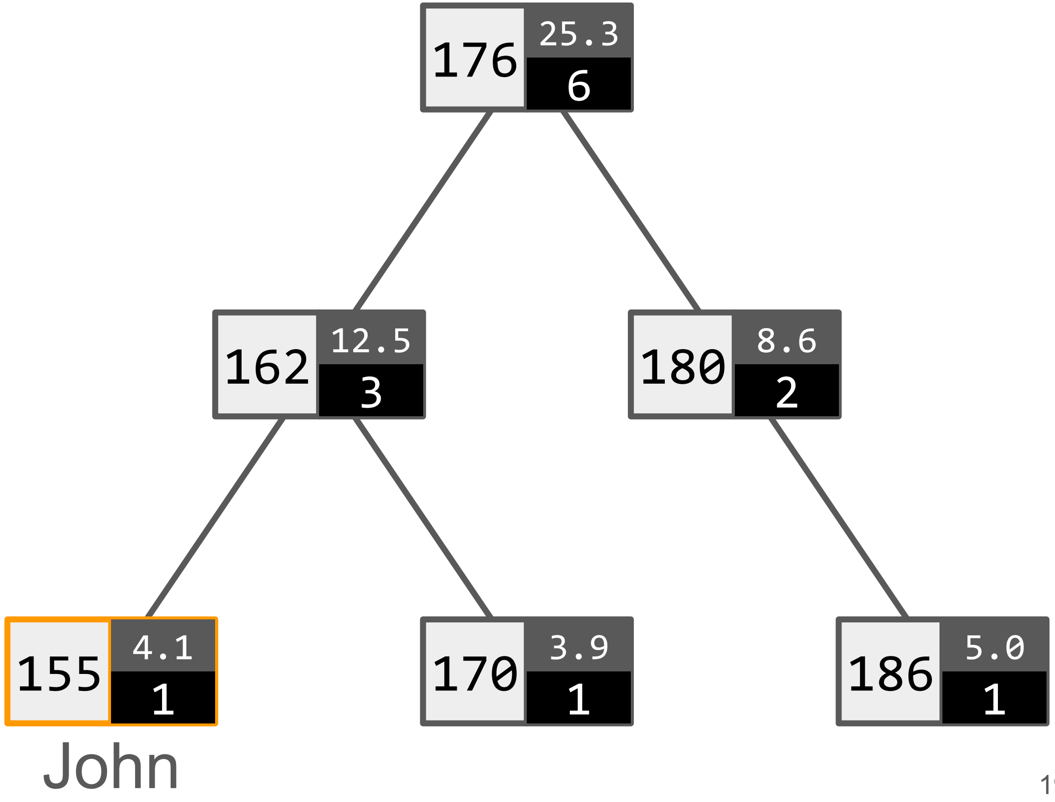
`findAverageGrade("John")`

Finding average grade now is simple for this case:

In `heightTree`, remove John's `gradeSum` and `weight` from root, then divide like so:

$$(25.3 - 4.1)/(6 - 1) = 4.24$$

Is this strategy always correct? I.e. Do we simply deduct target node's value away from root?

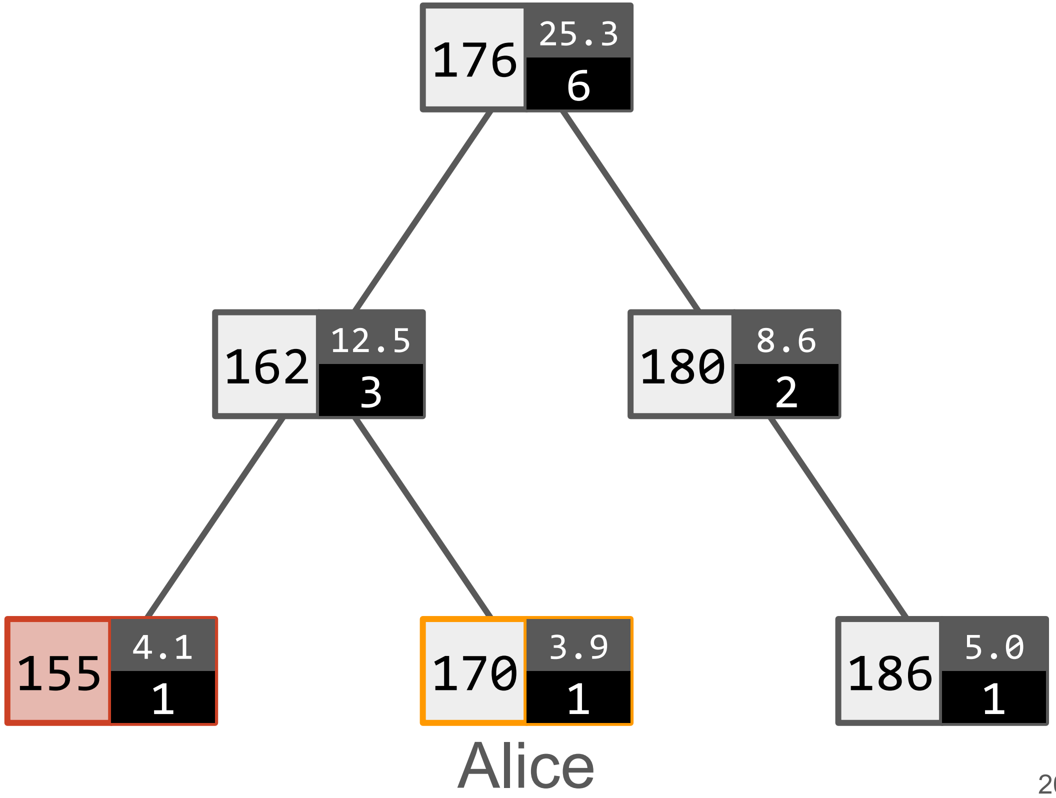


`findAverageGrade(“Alice”)`

What happens in this case?

If we follow the same naive strategy as before, we would have erroneously also counted in the people who are shorter than us!

We must therefore *traverse the tree* and figure out what to add/subtract!

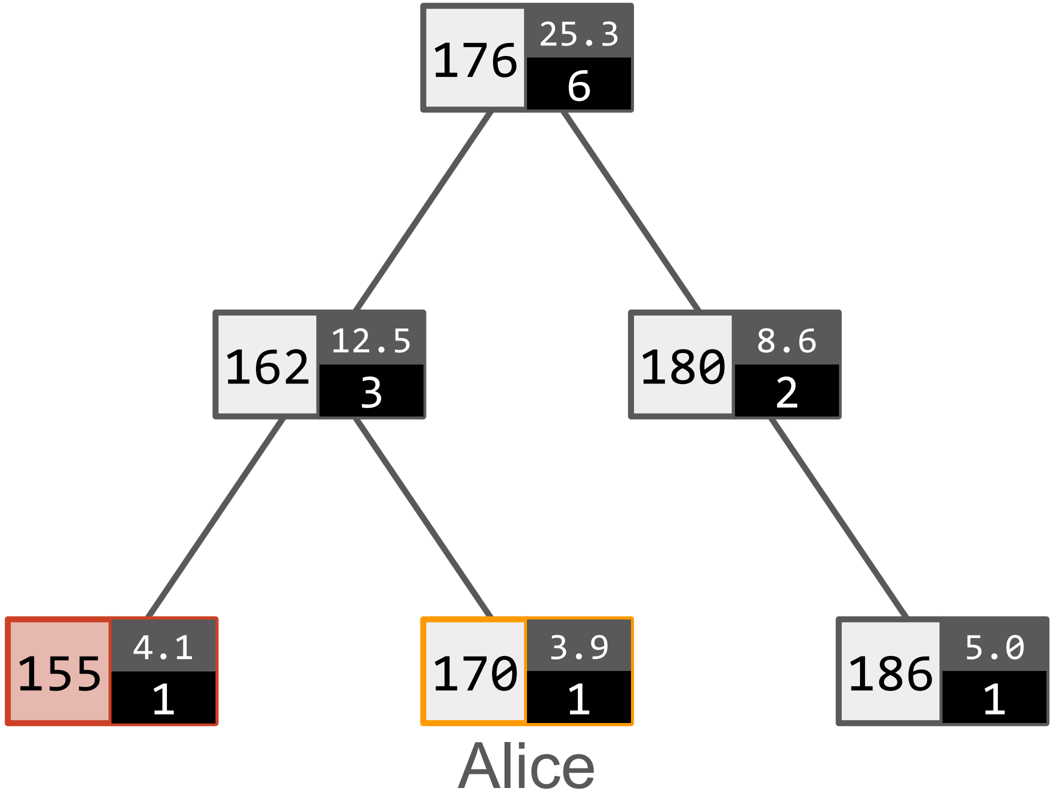


`findAverageGrade` strategy

- In `heightTree`, the root node's value stores the total number of students (`root.weight`) as well as their combined grade (`root.gradeSum`)
- We just need to deduct from `root`'s value the value of our query student (`stuH`), **as well as** all those values of students who are shorter than `stuH`
- Doing so will leave us with the total number of students taller than `stuH` as well as their combined grade

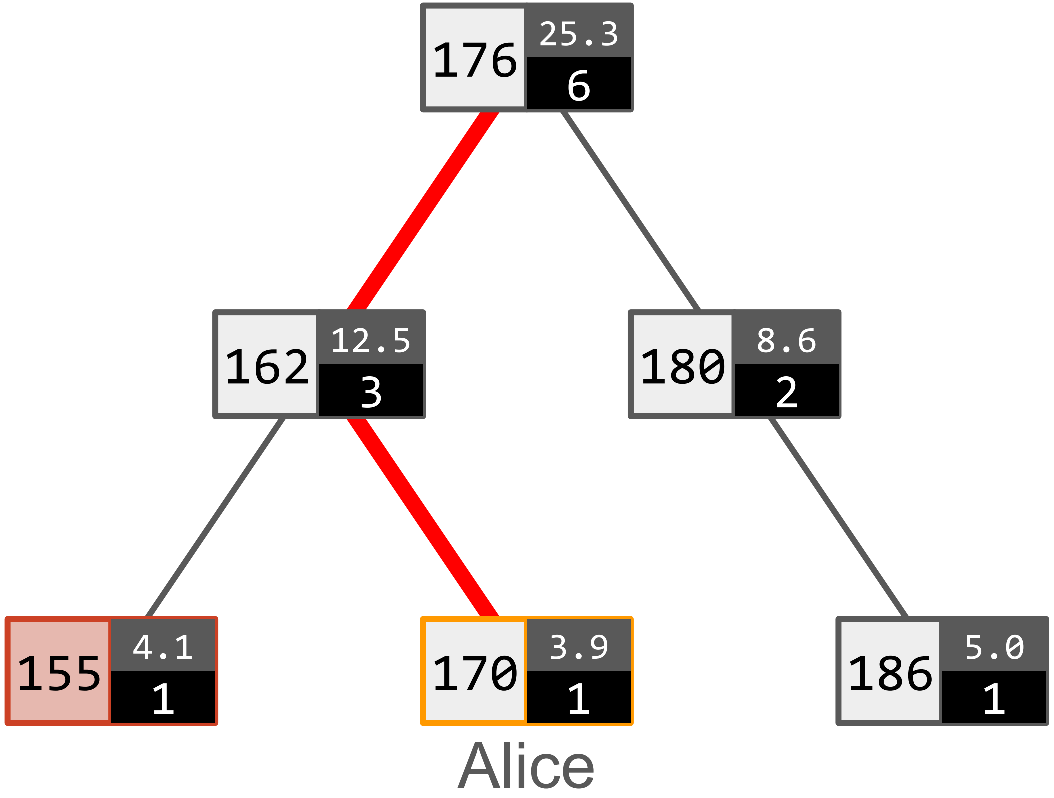
Tallergradesum = 25.3

Tallerweight = 6



Tallergradesum = 25.3

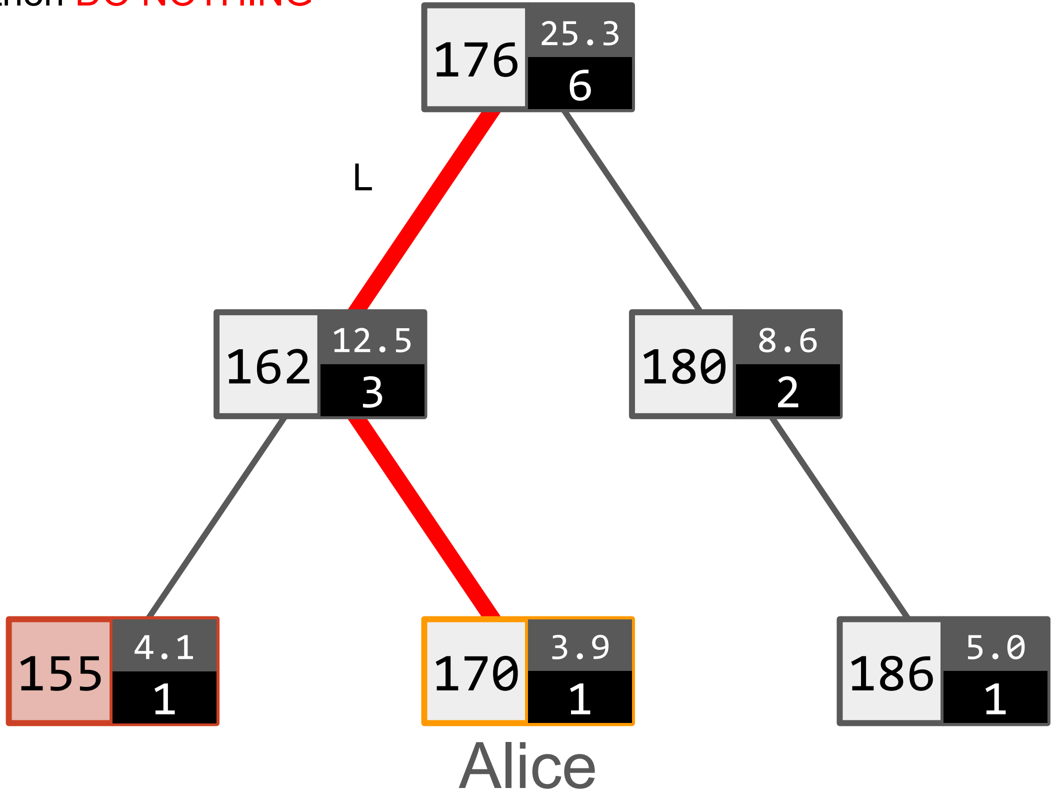
Tallerweight = 6



If the path goes left then DO NOTHING

Tallergradesum = 25.3

Tallerweight = 6

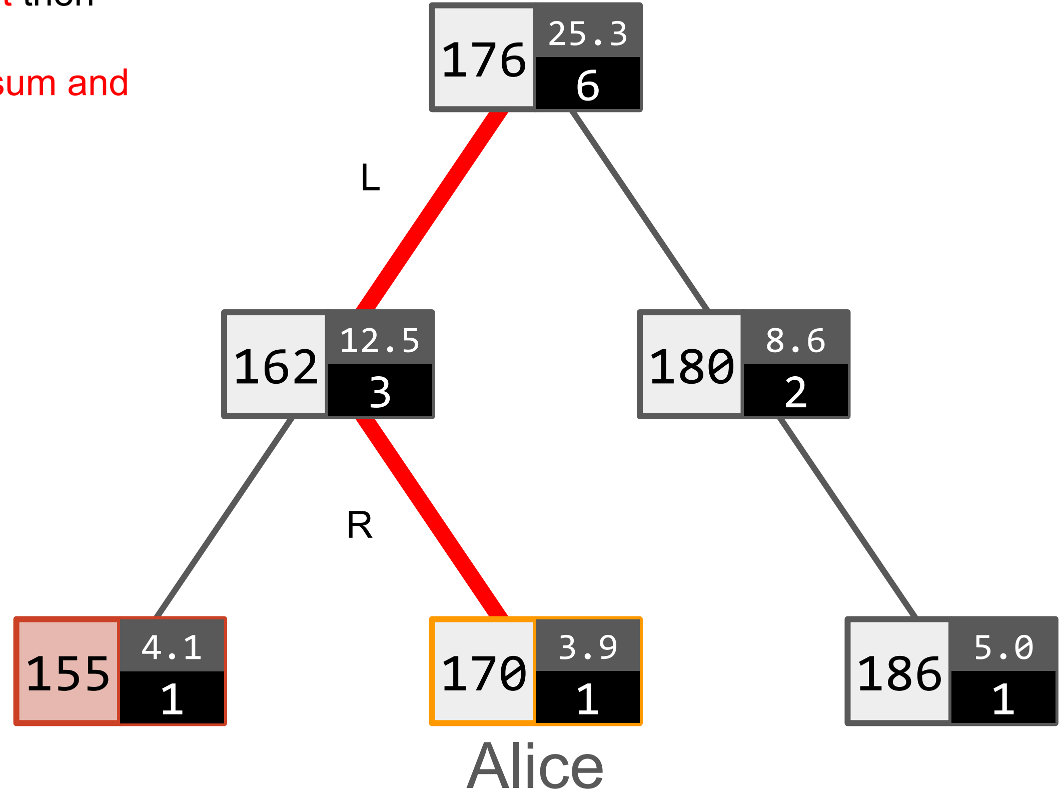


If the path goes right then

Update Tallergradesum and Tallerweight

Tallergradesum = 25.3

Tallerweight = 6

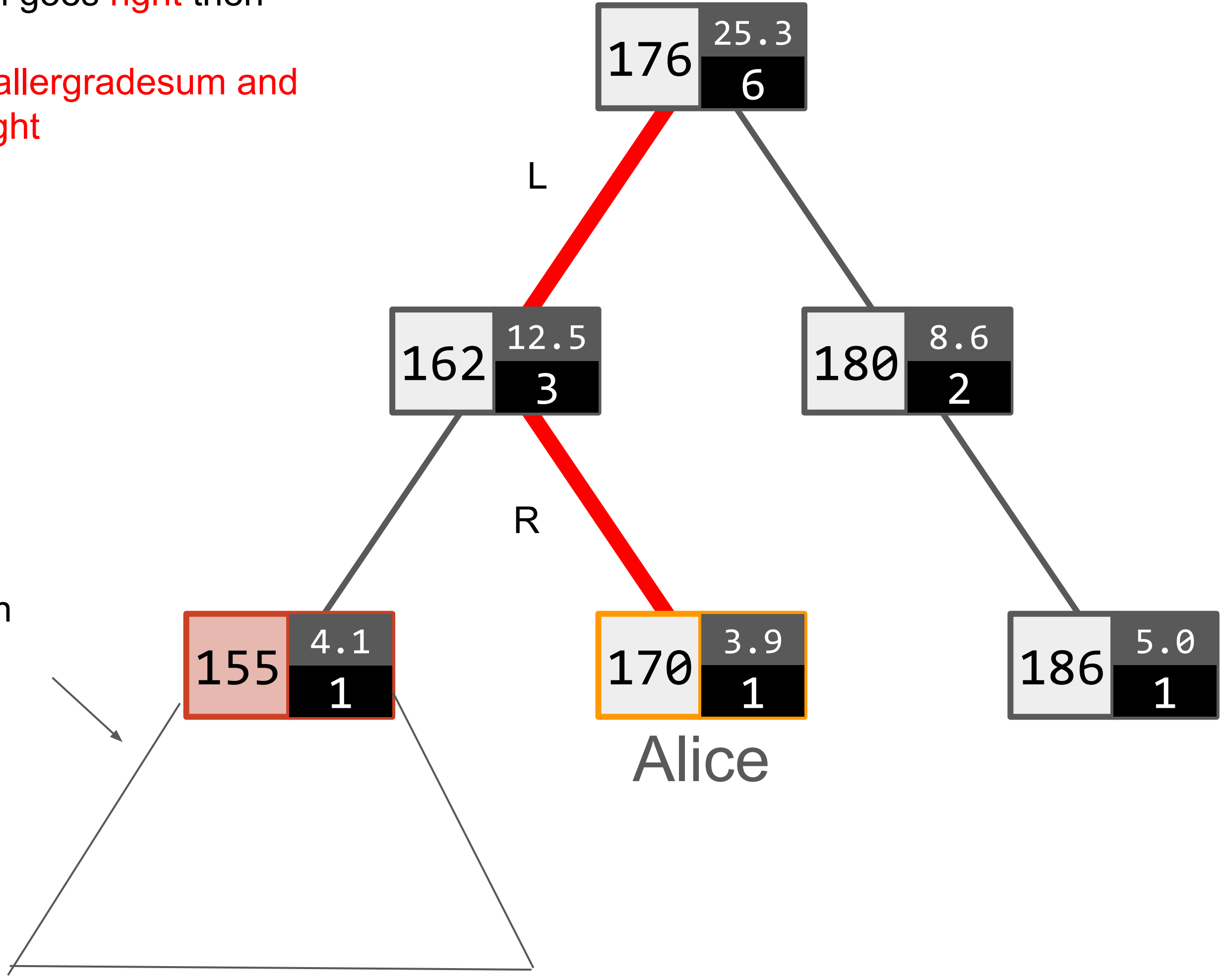


If the path goes right then

Update Tallergradesum and Tallerweight

Tallergradesum = 25.3

Tallerweight = 6



All the heights in this subtree are less than 170

If the path goes right then

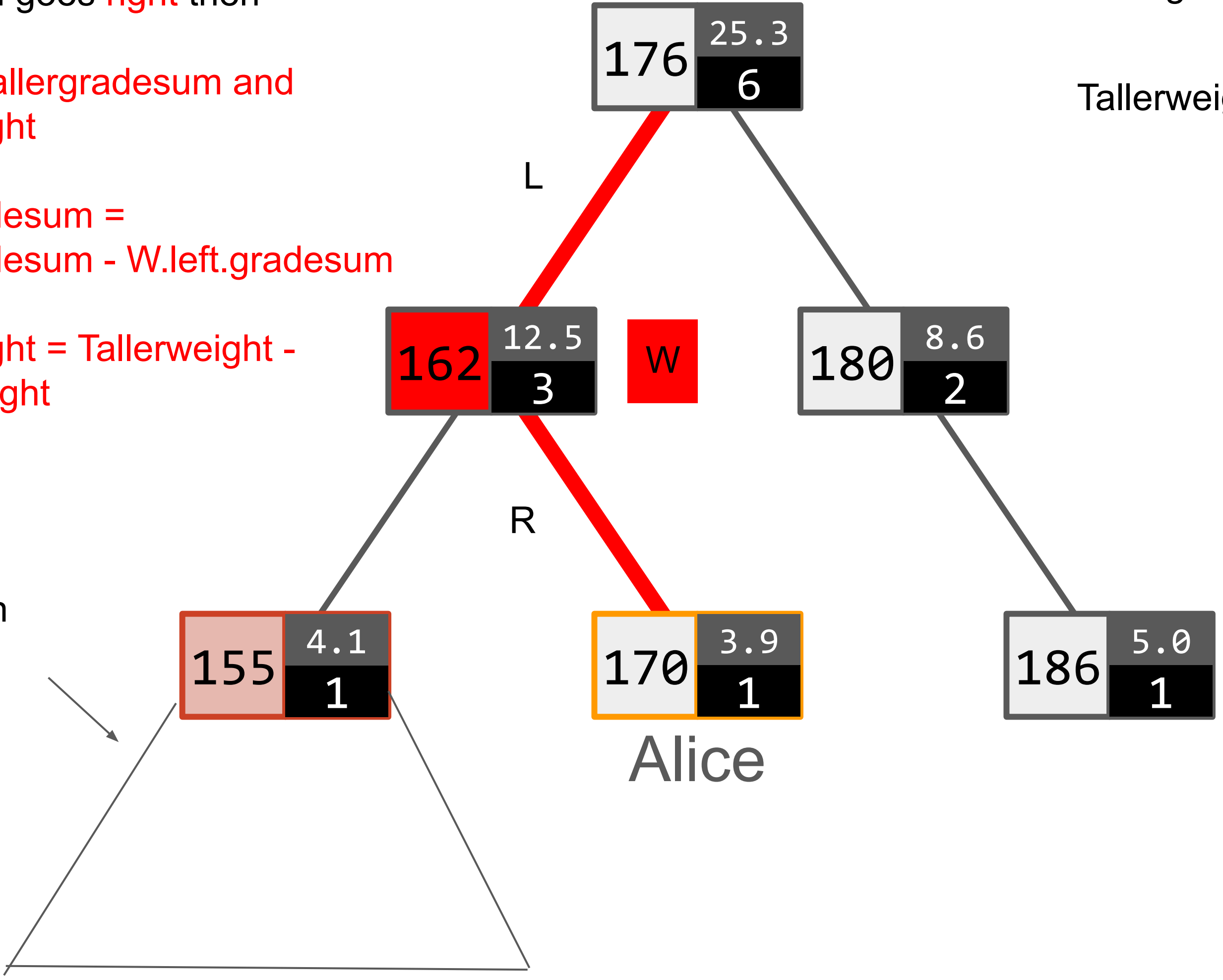
Update Tallergradesum and Tallerweight

Tallergradesum = Tallergradesum - W.left.gradesum

Tallerweight = Tallerweight - W.left.weight

Tallergradesum = 25.3 - 4.1 = 21.2

Tallerweight = 6 - 1 = 5



All the heights in this subtree are less than 170

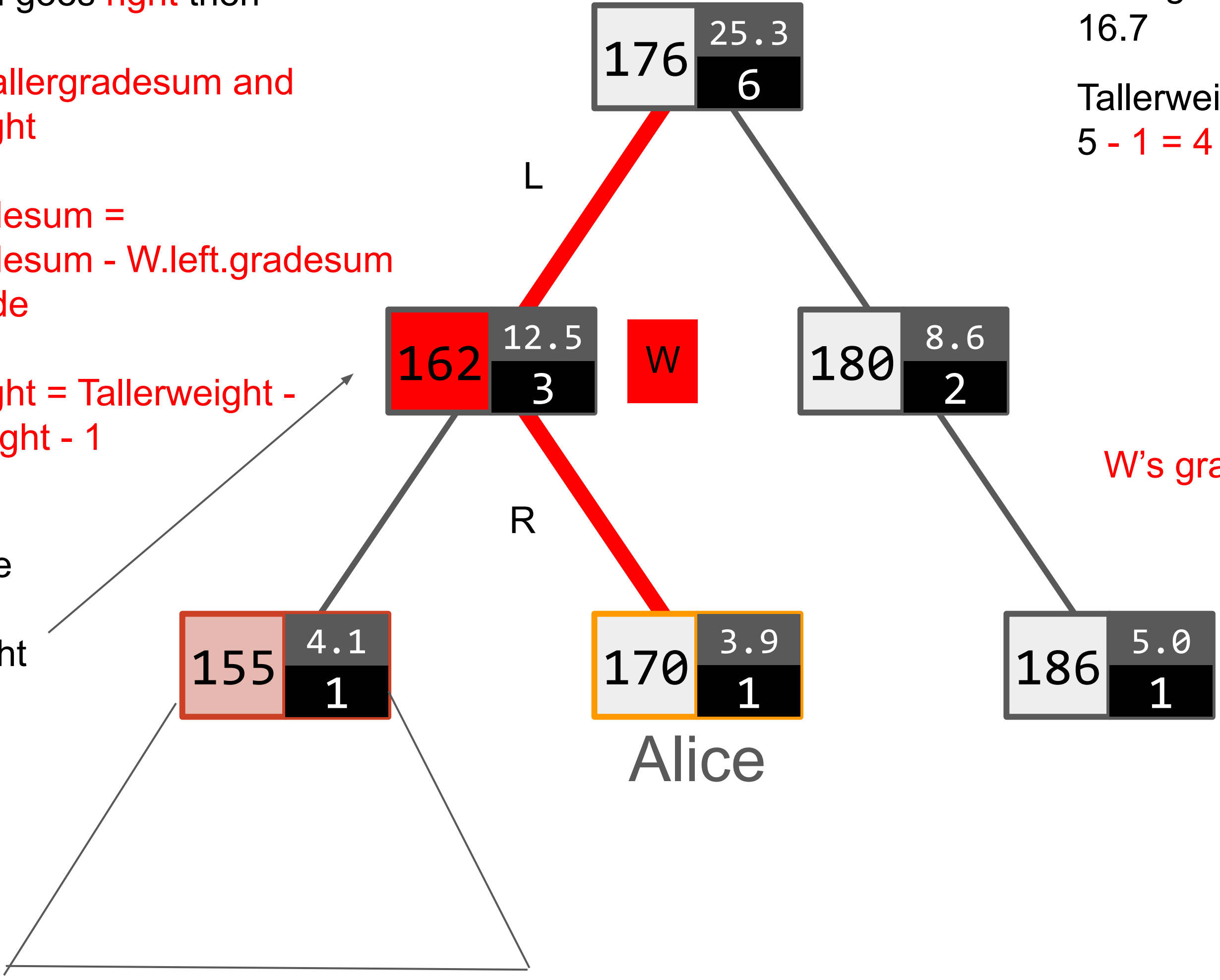
If the path goes right then

Update Tallergradesum and Tallerweight

Tallergradesum = Tallergradesum - W.left.gradesum - W's grade

Tallerweight = Tallerweight - W.left.weight - 1

This node also has less height than 170



Tallergradesum = 25.3 - 4.1 - 4.5 = 16.7

Tallerweight = 6 - 1 - 1 = 5 - 1 = 4

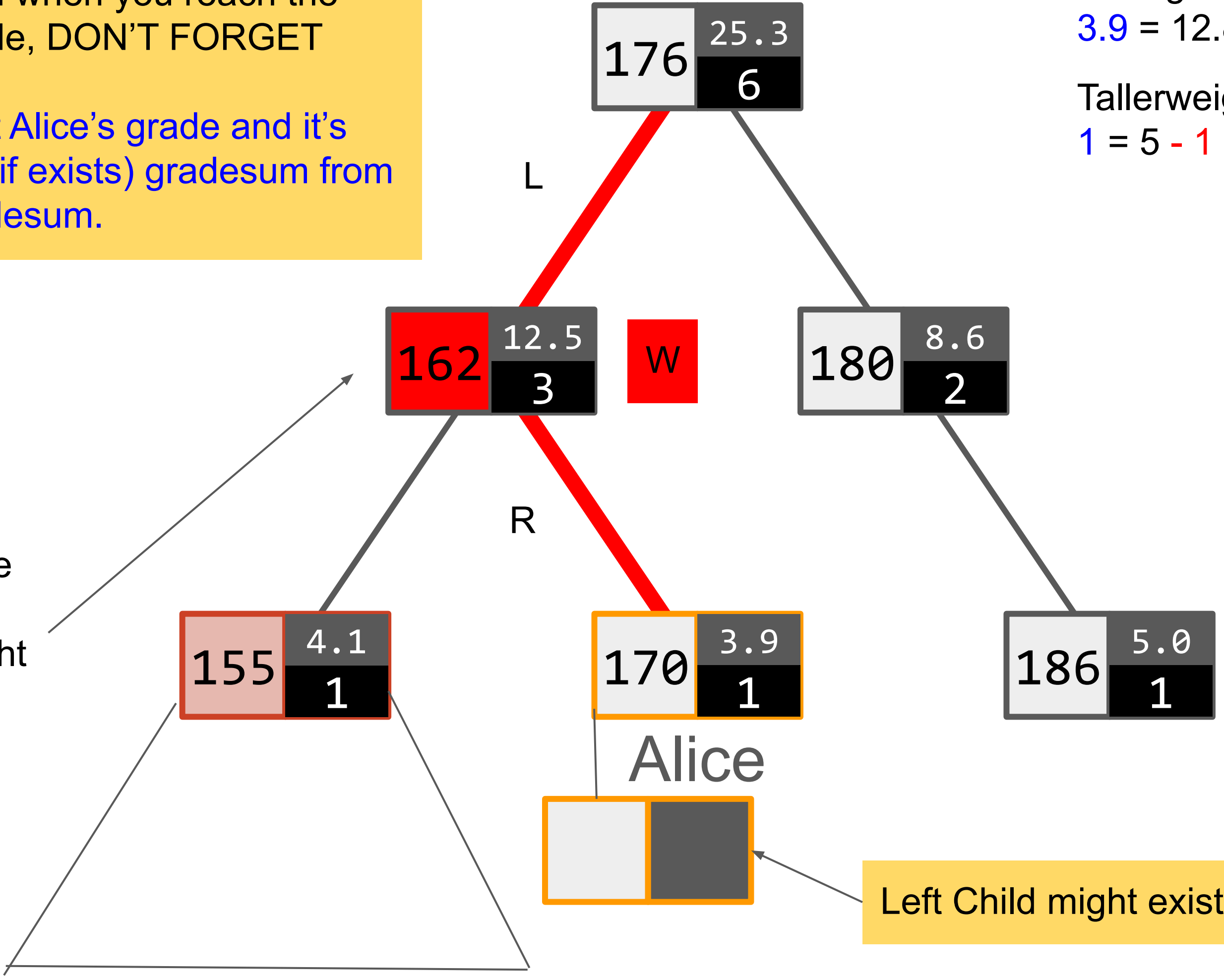
W's grade = 12.5 - (4.1 + 3.9) = 4.5

In the end when you reach the query node, DON'T FORGET

To deduct Alice's grade and it's left child (if exists) gradesum from Tallergradesum.

Tallergradesum = 25.3 - 4.1 - 4.5 - 3.9 = 12.8

Tallerweight = 6 - 1 - 1 - 1 = 5 - 1 - 1 = 3

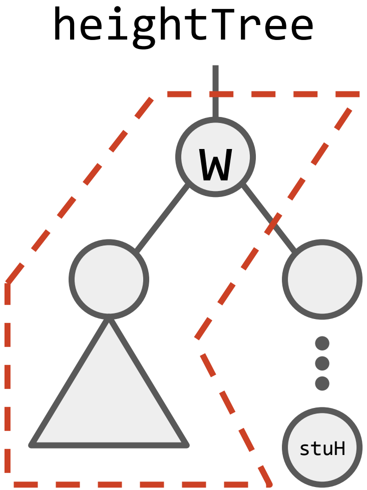


This node also has less height than 170

Left Child might exist

`findAverageGrade` algorithm

1. Initialize variables for decumulation
   - `tallerGradeSum = root.gradeSum`
   - `tallerWeight = root.weight`
2. We traverse from `root` down to `stuH`, suppose current node is `w`
   - Once we have to recurse down to right child **in the next step**
     - From `tallerGradeSum` and `tallerWeight` respectively, we deduct
       - `w`'s `grade` and `1`
       - `w.leftChild`'s `gradeSum` and `weight`
     - Realize this is nothing but first deducting `w`'s values then adding by `w.rightSubtree`'s values (because we over-deducted)
   - Not forgetting to also finally deduct `stuH` and its left child (if exist)
3. Return `tallerGradeSum/tallerWeight`



These correspond to students shorter than `stuH`

`findAverageGrade` strategy

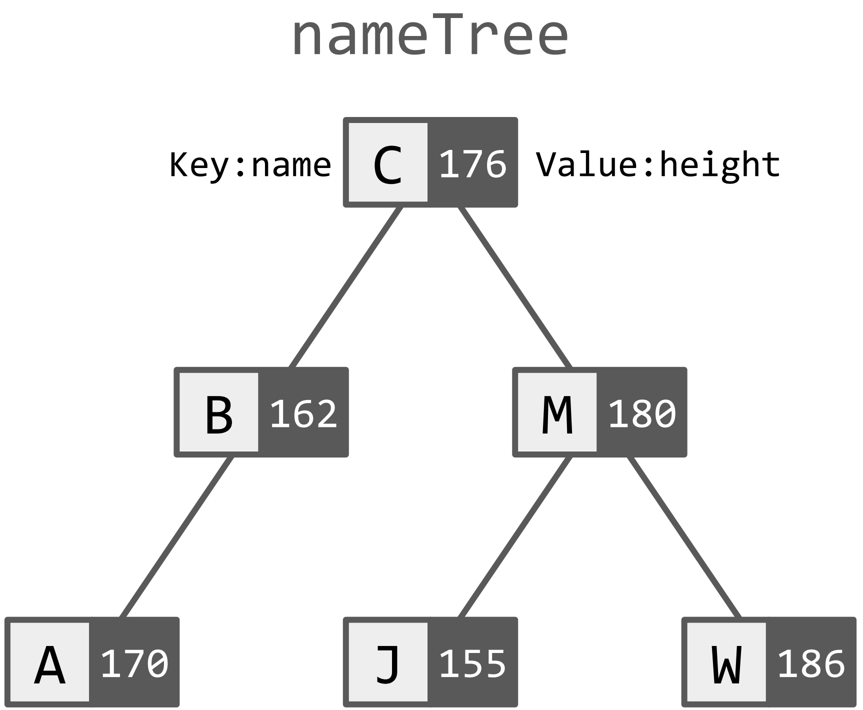
- Here I only presented one possible solution
- There are obviously various ways to solve this problem
- Some of you have proposed variations of the following:
    - Instead of traversing downwards from `root` to `stuH`, traverse upwards from `stuH` to `root`
    - Instead of decumulation, accumulate the values of students taller than `stuH`
    - Design each node in `heightTree` to only store `gradeSum` and `weight` of its right child
- All of them are potentially workable solutions! Try implementing them in pseudocode and trace them out!
- Key thing here is that you are using data-summarisation ability of augmented trees to help you compute the answer so you don’t have to visit every node in the tree

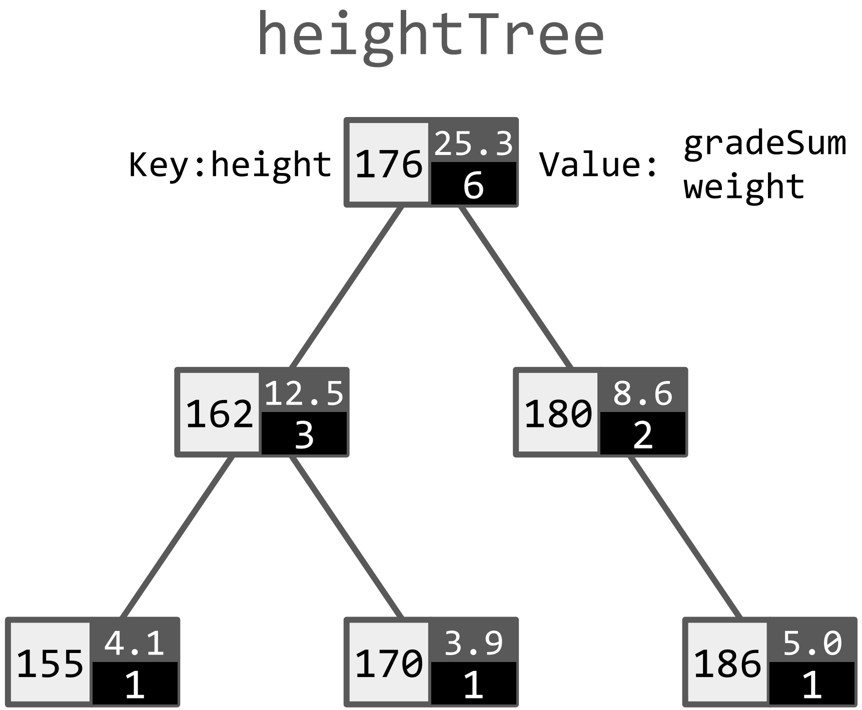
Test yourself!

What if we disallow hashtables?

Can you come up with an alternative for `nameTable`?

2 BSTs!


nameTree
Key:name
C
176
Value:height
B
162
M
180
A
170
J
155
W
186


heightTree
Key:height
176
25.3
6
Value:
gradeSum
weight
162
12.5
3
180
8.6
2
155
4.1
1
170
3.9
1
186
5.0
1

Problem 1.c.

What if height is now not unique?

What issue(s) will arise from this?

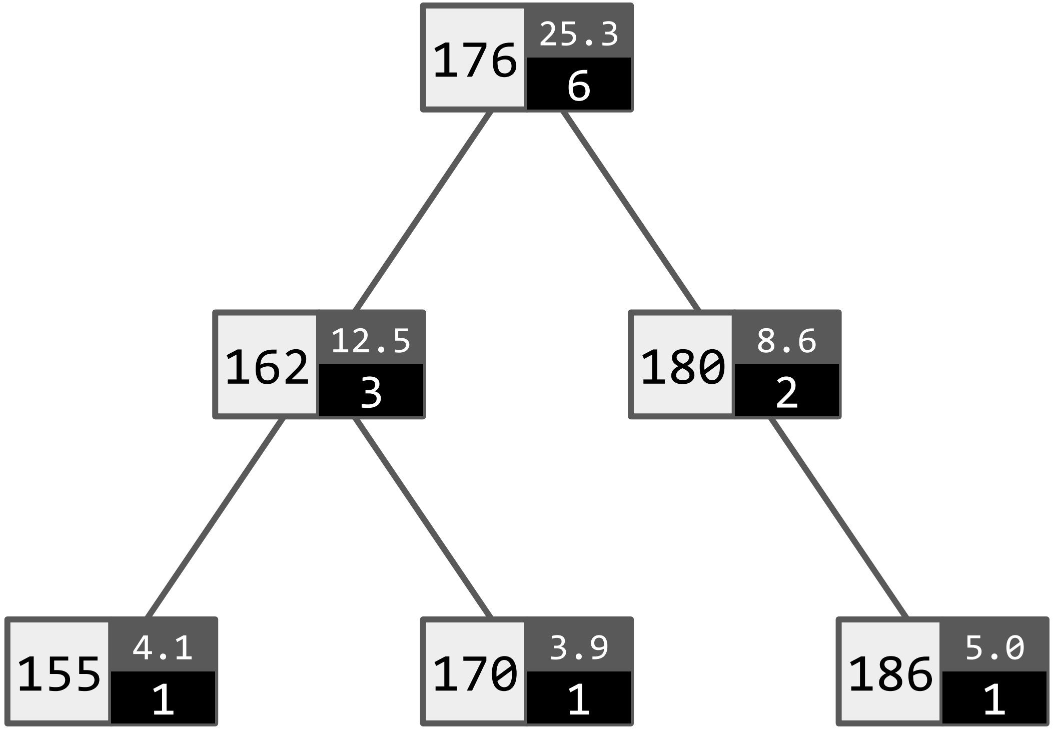
How might you modify your solution in the previous part to resolve the issue(s)?

`insert(“Peter”, 176, 4.8)`

So now we have a new student Peter who is 176cm tall.

We already have a student (Charles) with height 176.

What do we do now???



Key duplicates

Realistically speaking, the data might contain certain keys with high duplicate count. This is especially true for something like height in a large group of people.

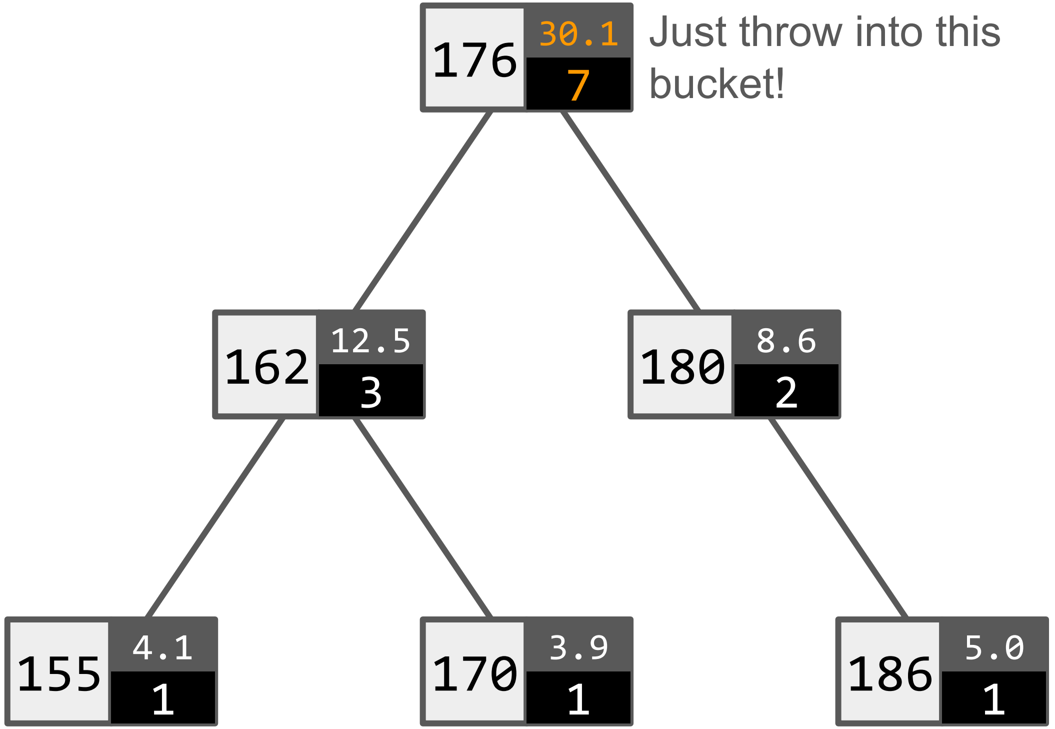
Having too many duplicate values in a BST can lead to non-optimal insertion and a potentially complicated (read: bug-prone) querying process.

There exists an easy and optimal solution to handle duplicate height in this problem :)

```
insert("Peter", 176, 4.8)
```

We can simply treat each node as a bucket of students!

You should convince yourself that doing so wouldn't affect any operation.



Test yourself!

If there will be *no further insertions* after initially adding in all students in the roster, how might you achieve $O(n \log n)$ preprocessing and $O(1)$ query time?

Challenge yourself!

We have conveniently avoided mentioning about tree rebalancing in an augmented BST tree.

How would rebalancing work with augmented nodes in the tree here?

Problem 2

A game of cards

Suppose you have a deck of n cards.

They are spread out in front of you on the table from left to right.

Each card indexed from 1 to n.

Each card can either be facing up or down.

ADT

Operation	Behaviour
`query(i)`	Return whether card at index `i` is facing up or down.
`turnOver(i,j)`	Turn over all cards in the subsequence specified by the index range [`i`, `j`].

Problem 2.a.

Given n cards already laid out on the table, how do you design a DS that implements such an ADT?

Can you achieve `turnOver(i,j)` in less than $O(n)$ time? Just like magic!

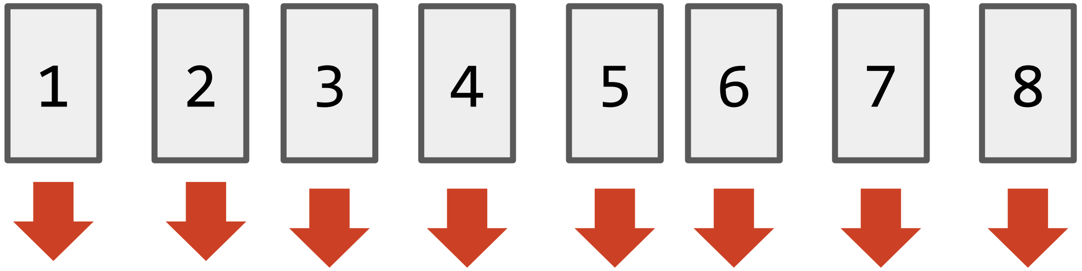
Problem 2.a.ー Approach

Perhaps let’s start with a simpler problem.

Suppose all cards (say we have 8) are initially facing downwards.

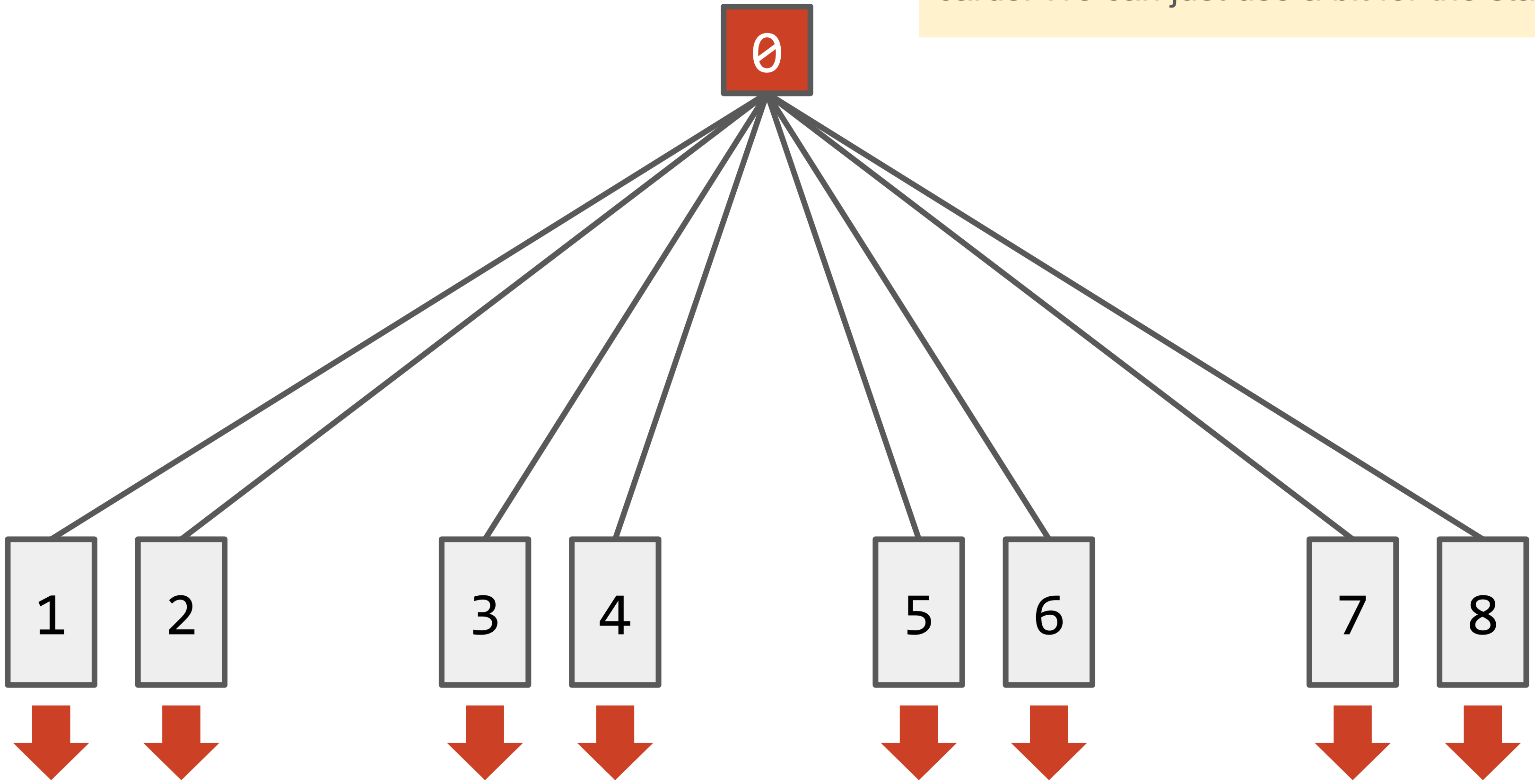
How can we use augmented trees to turn over all cards at once?

Problem 2.a. - Approach



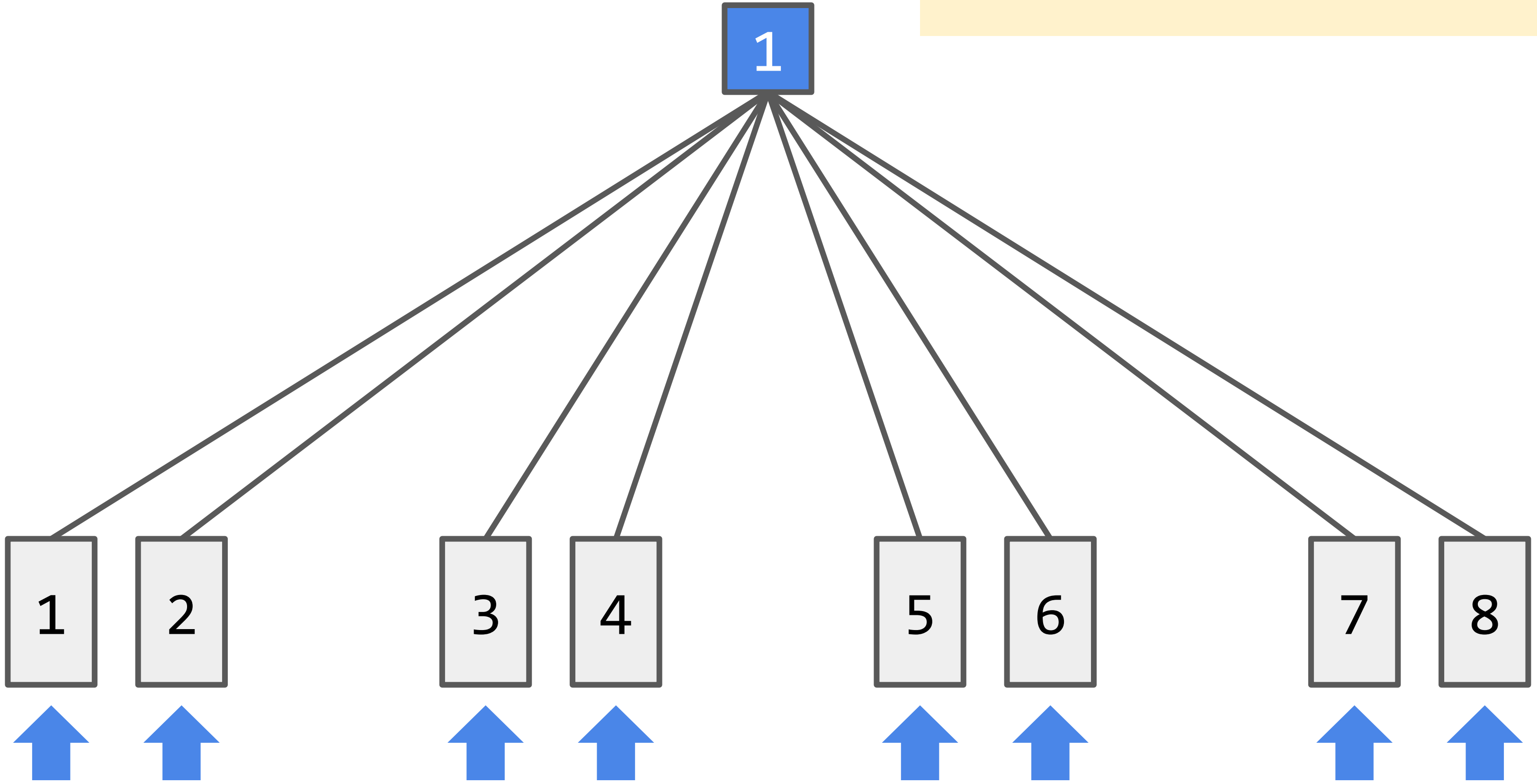
Toggling all cards

Use an augmented tree with root summarising face direction for all the cards! We can just use a bit for the state.



Toggling all cards

Toggle state bit at root node to turn over all the cards



Problem 2.a. — Approach

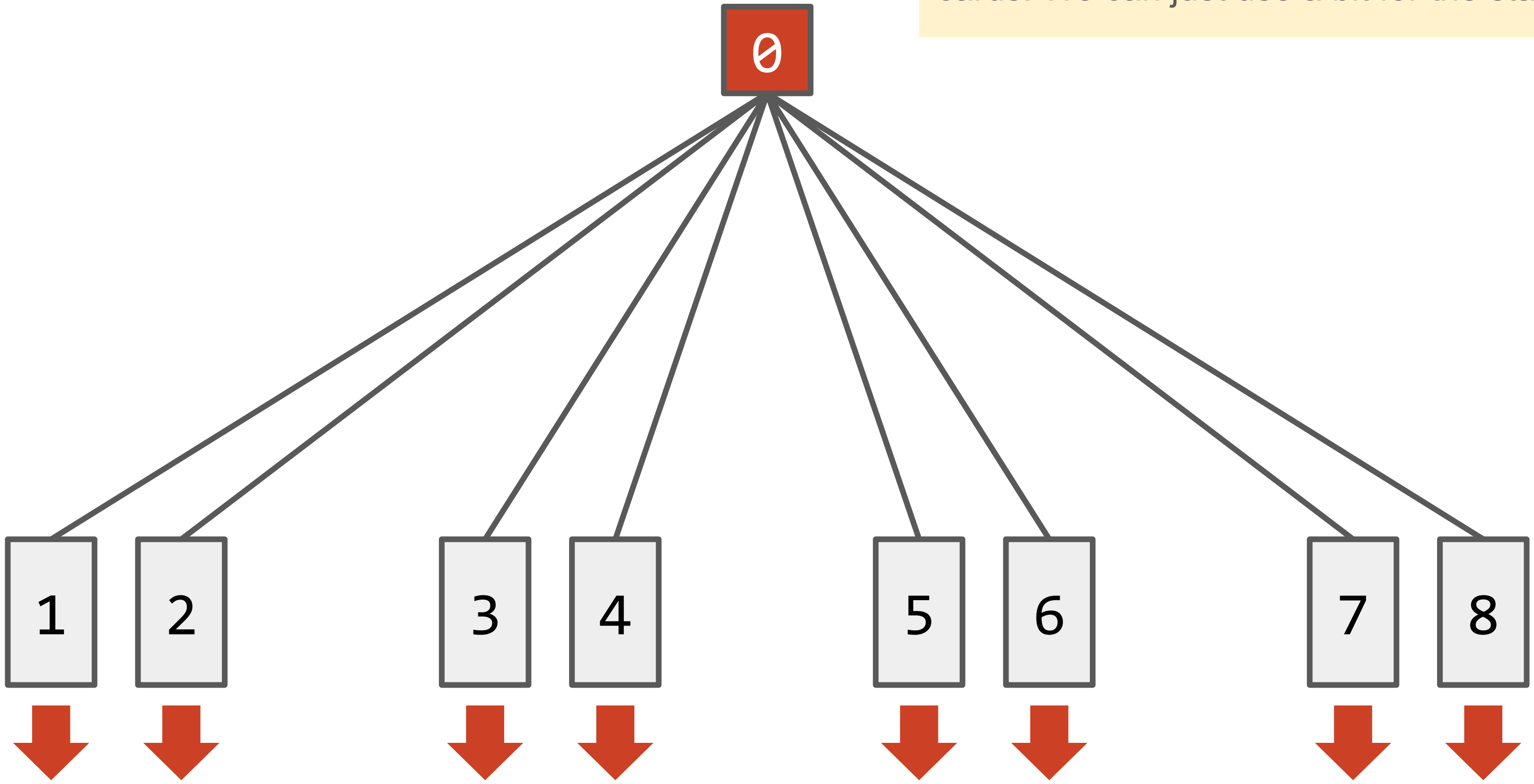
Ok that was easy!

Now let’s solve a *slightly* harder problem.

What if, in addition to having the ability of turning over all cards at once, we can *also* choose to turn over half of them?

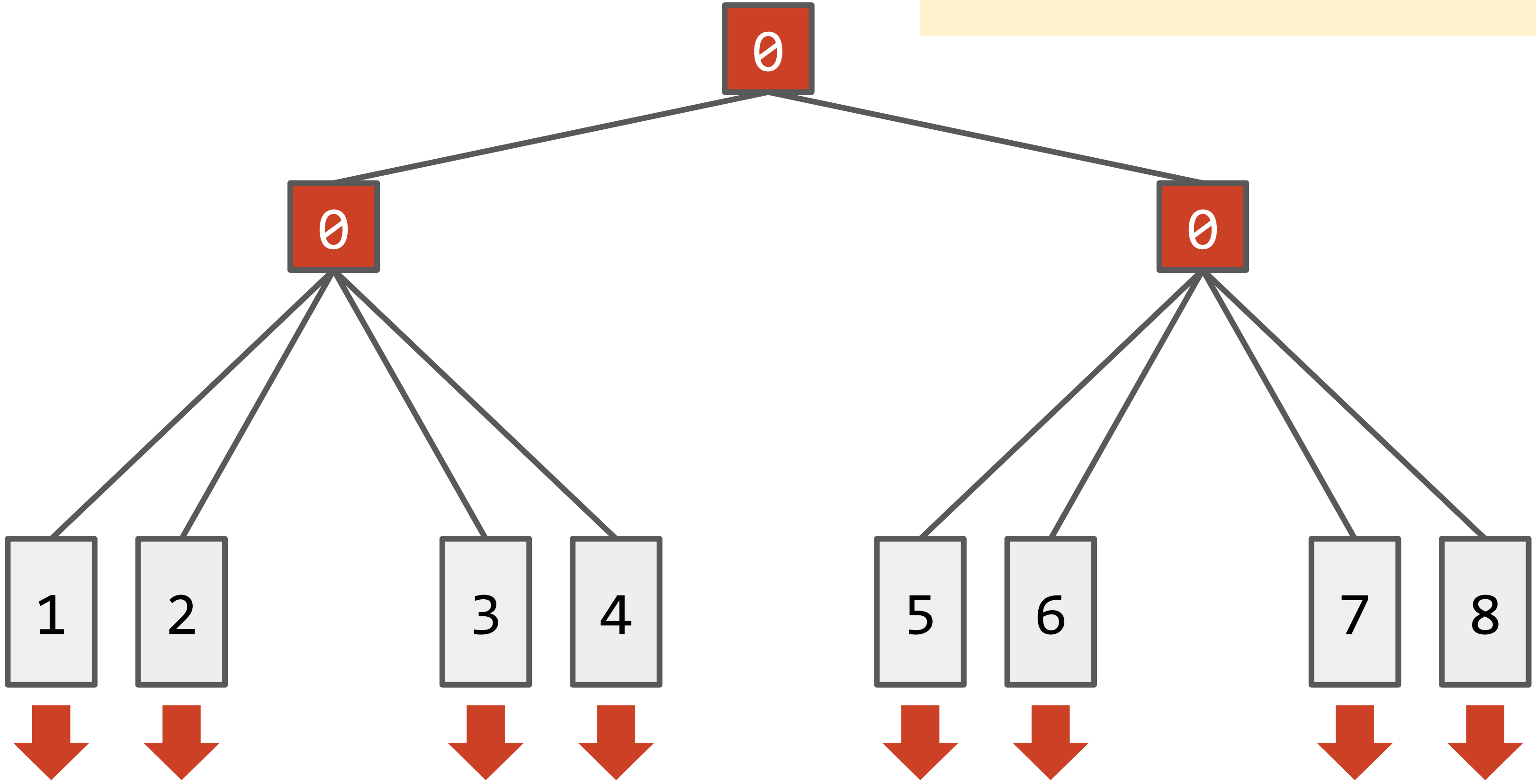
Toggling all cards

Use an augmented tree with root summarising face direction for all the cards! We can just use a bit for the state.



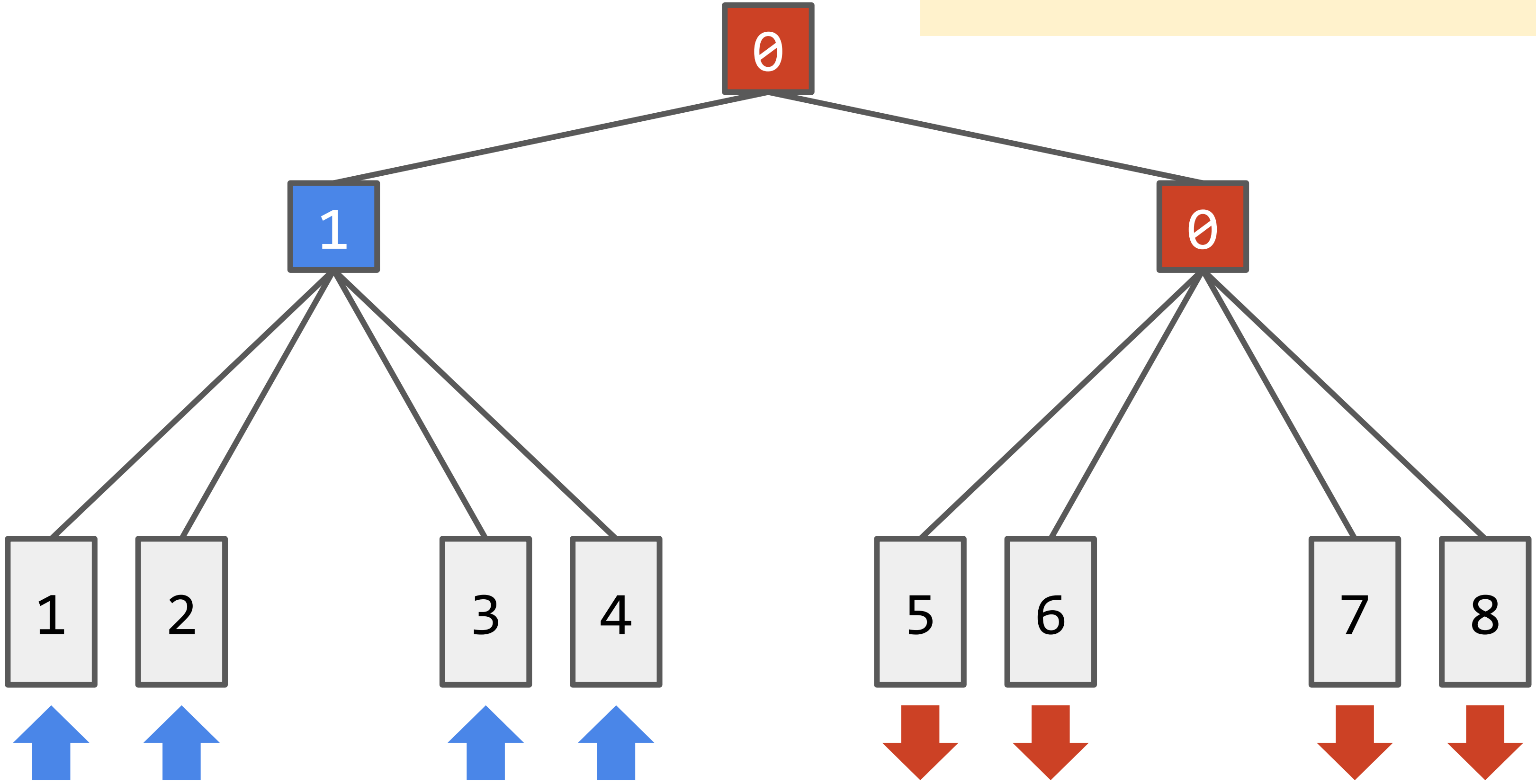
Also toggling ½ the cards

We just add another layer after the root node. Simple!



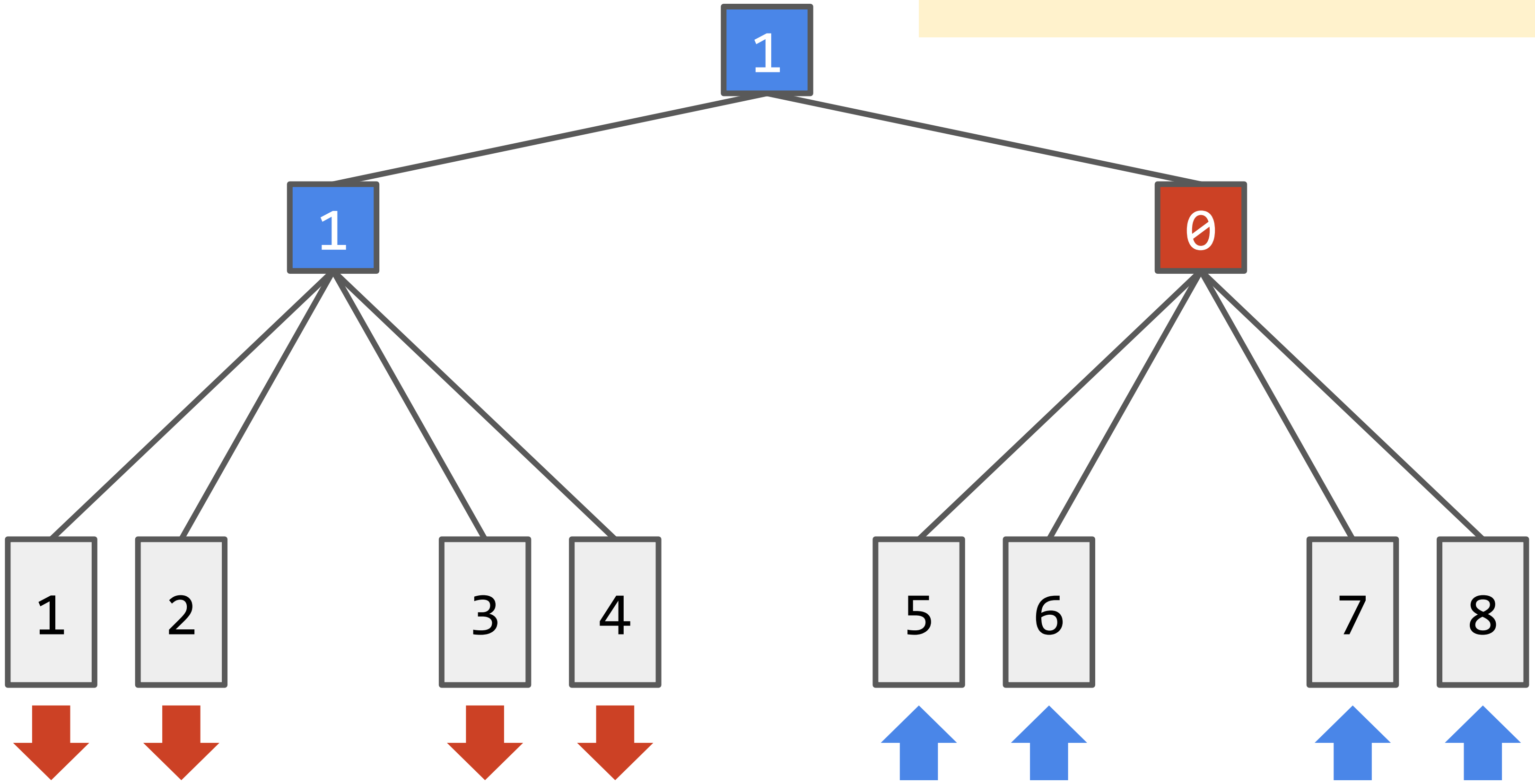
Also toggling ½ the cards

Toggling that new node now turns over half the cards!



Also toggling ½ the cards

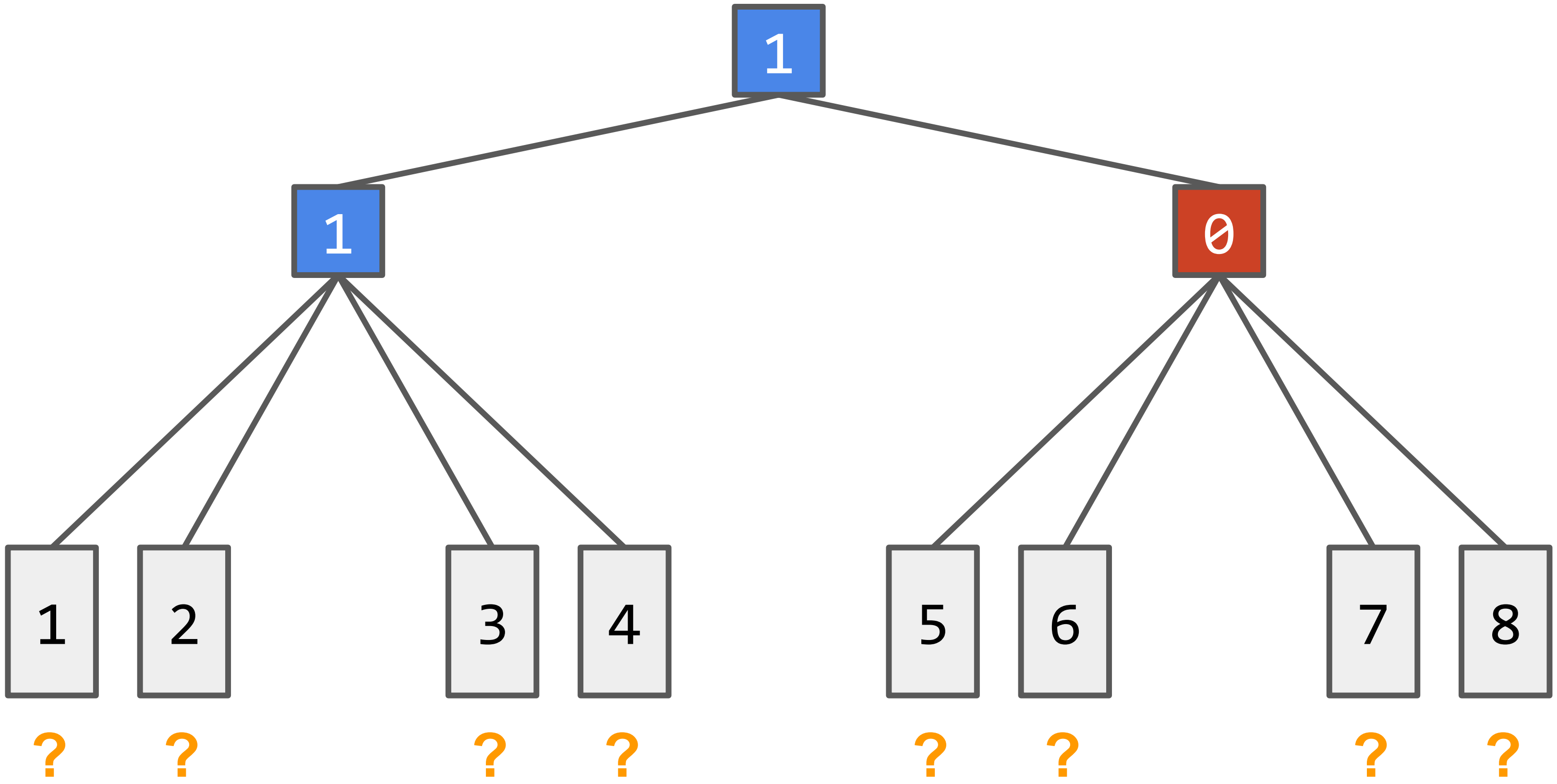
Toggling the root node still turns over all the cards.



Guiding question

How do we determine the face direction of a card at this point?

How do we determine face directions?



Guiding question

How do we determine the face direction of a card at this point?

Answer: Bits in the tree store all state changes of the cards, we just need to figure out which of them affect our card of interest.

Therefore to determine the face direction of a card, we just need to look along the path from the card (leaf) to the root node.

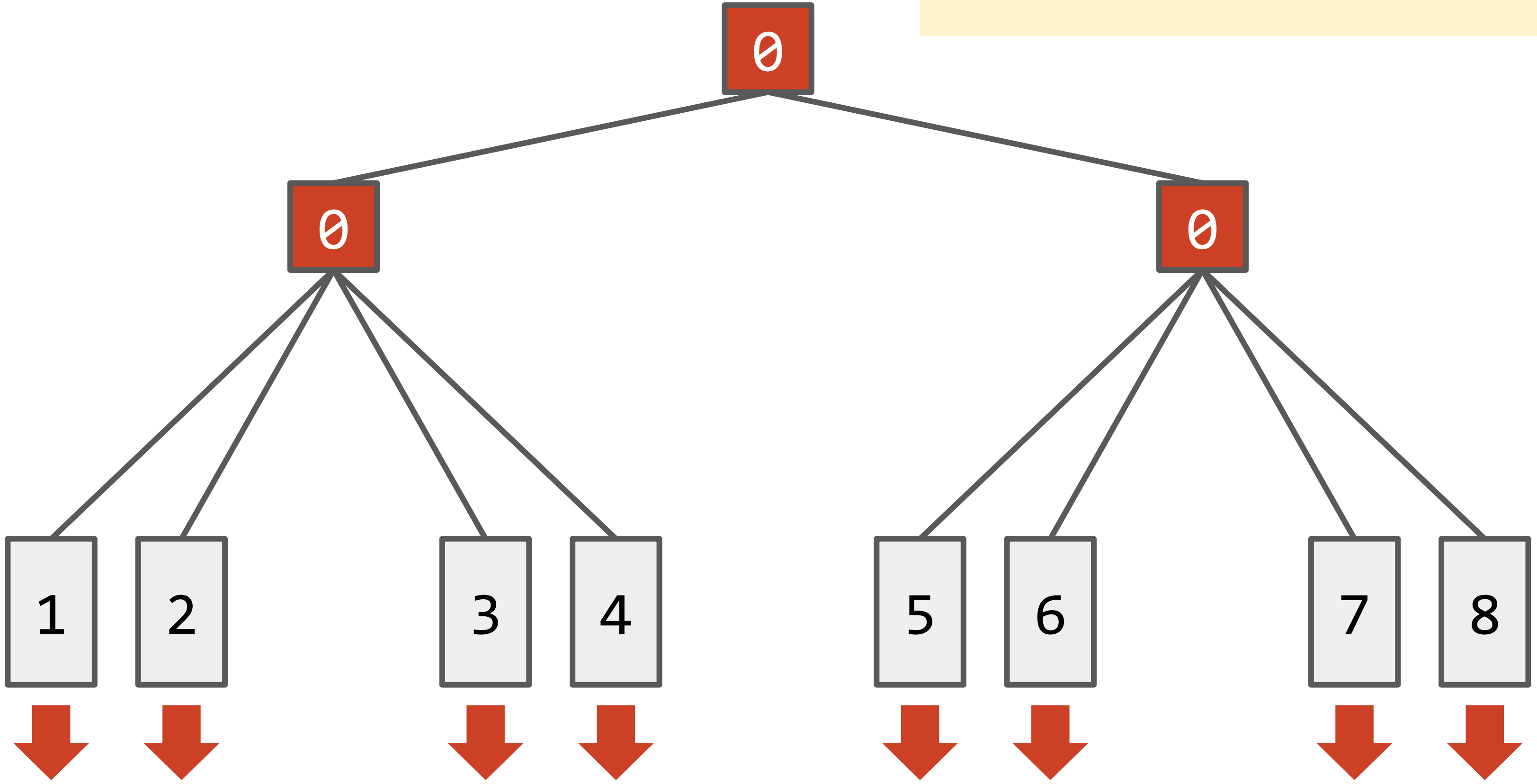
Problem 2.a. ─ Approach

Ok that was easy too!

Now what about *also* being able to turn over a quarter of the cards?

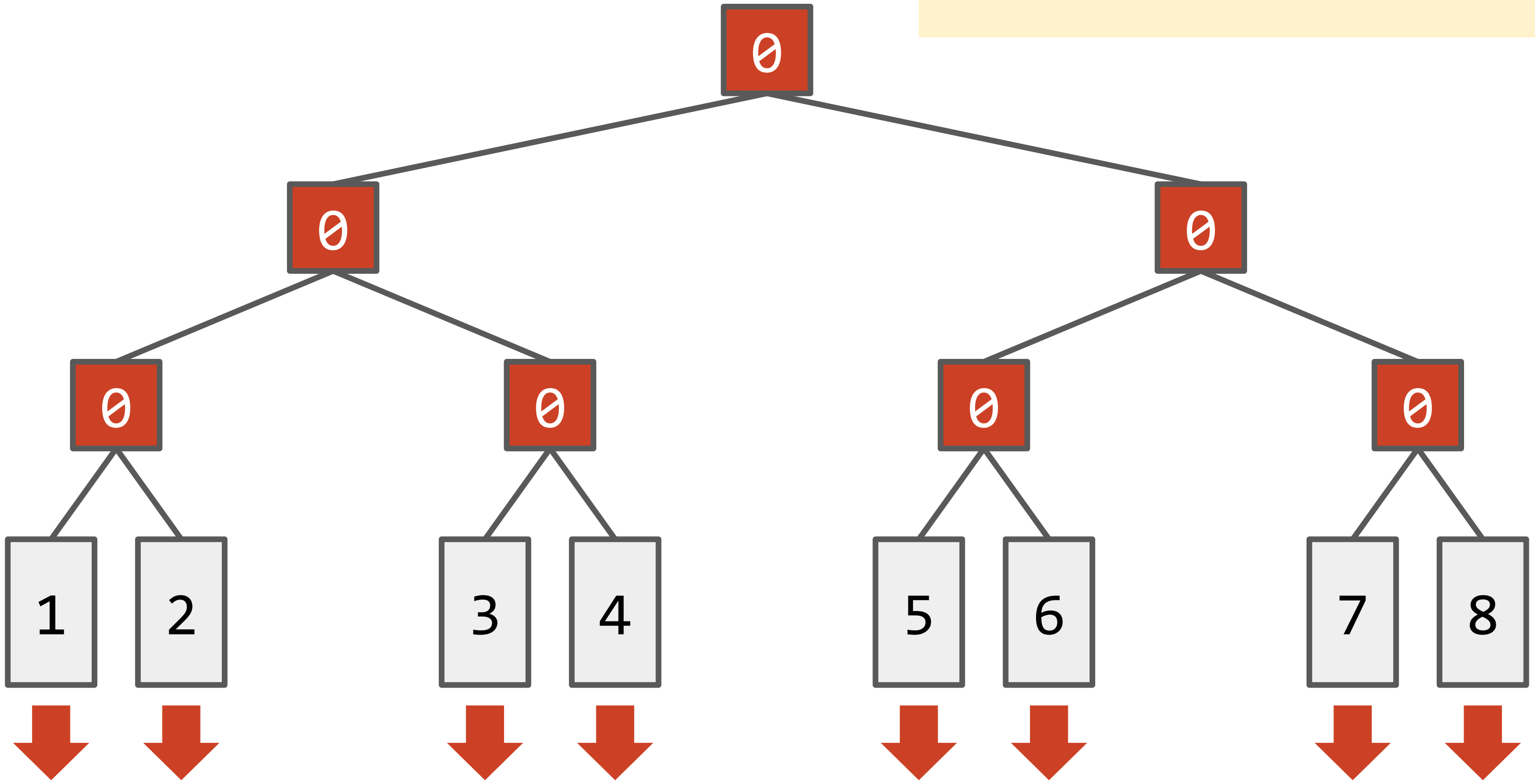
Also toggling ½ the cards

We just add another layer after the root node. Simple!



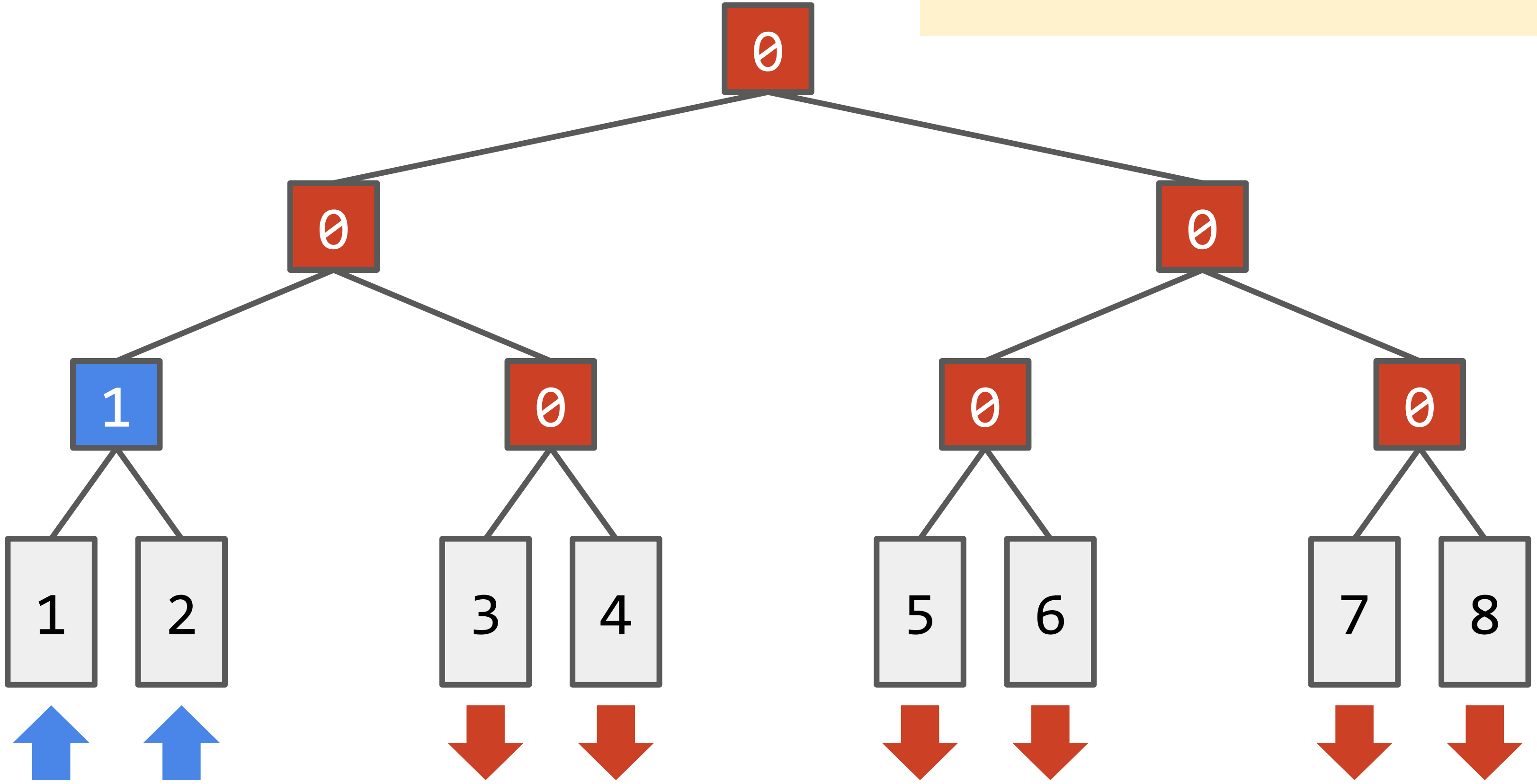
Also toggling ¼ of the cards

We just add another layer after the root node. Simple!



Also toggling ¼ of the cards

Toggling that new node now turns over a quarter of the cards!

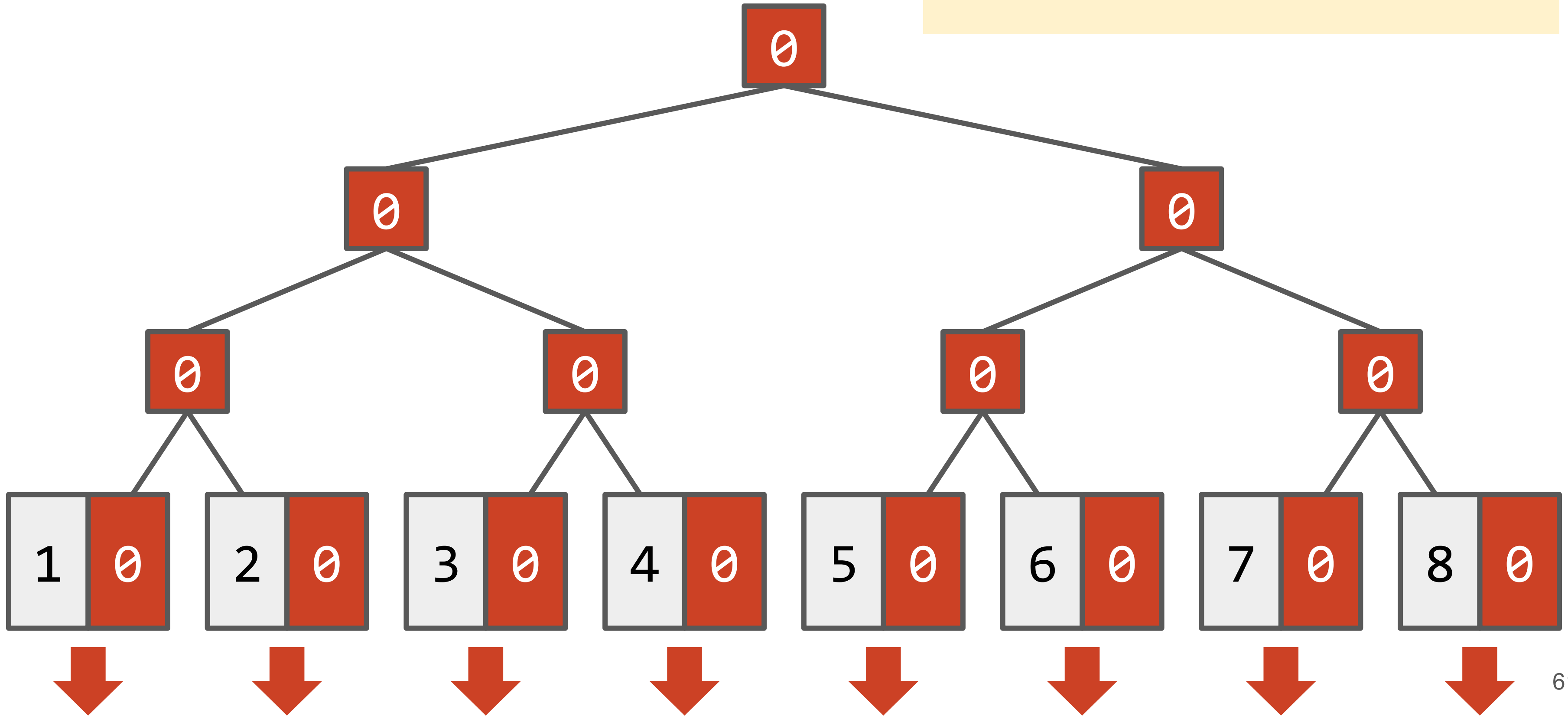


Problem 2.a.ー Approach

Now what about also being able to turn over every single card individually?

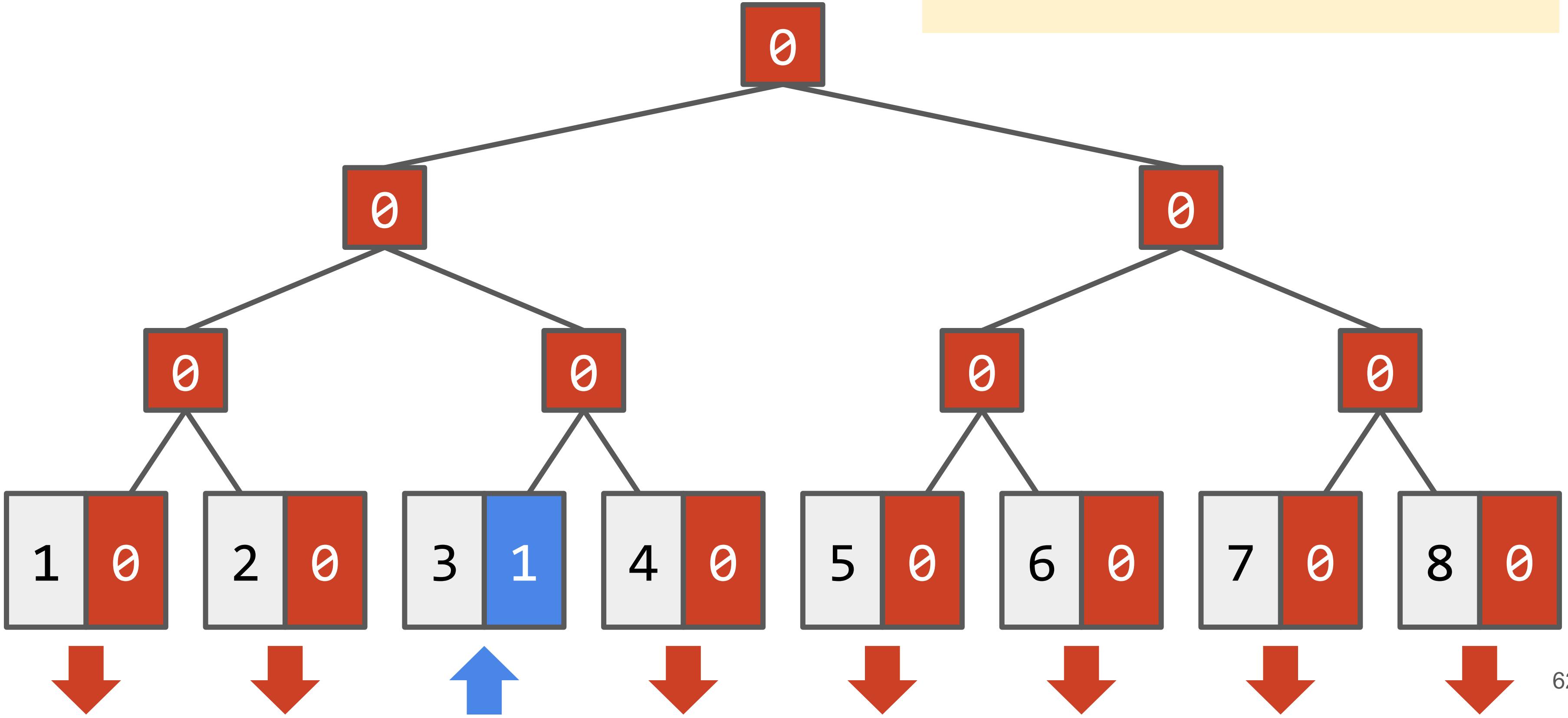
Also toggling every single card

We also attach a bit to every card!



Also toggling every single card

So turning over card 3 is as simple as this!



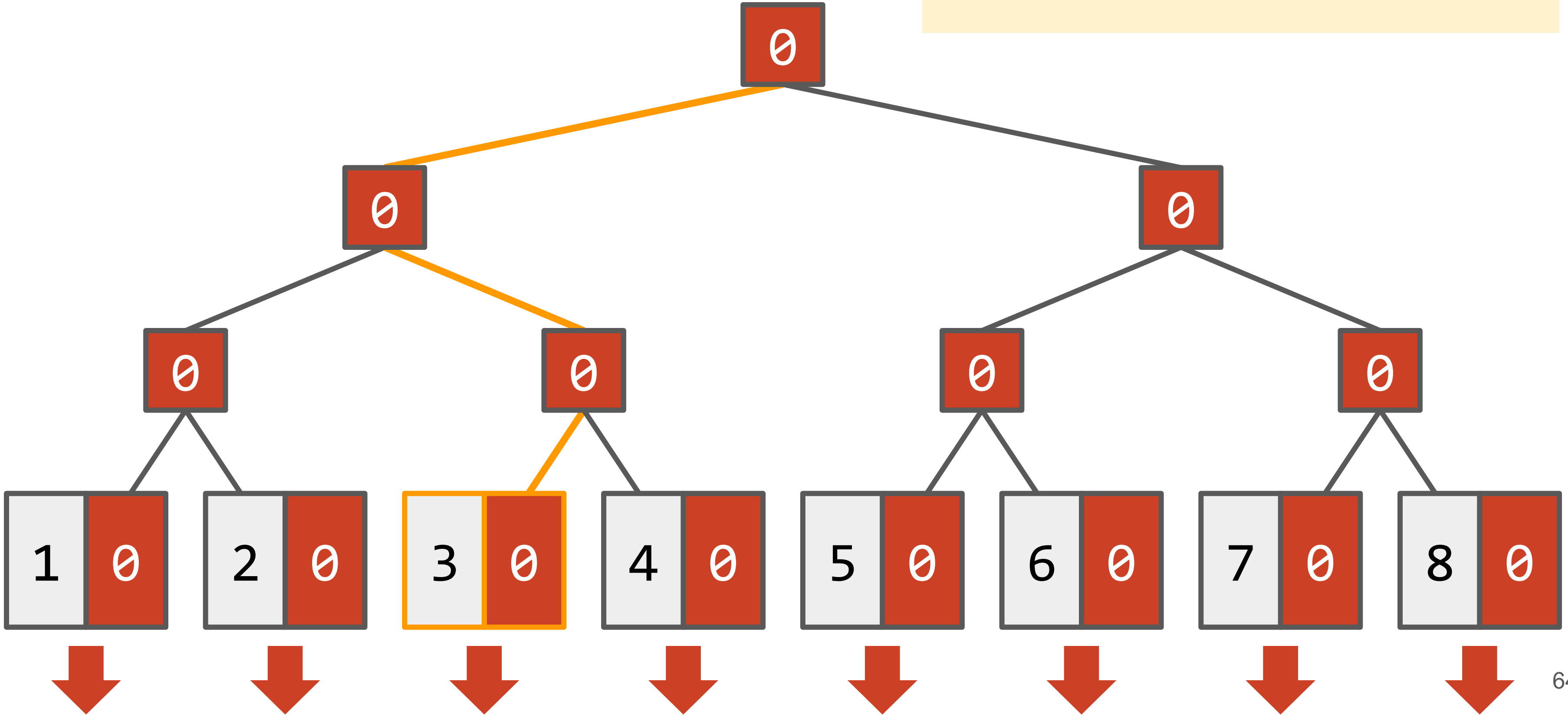
Problem 2.a.ー Approach

But wait a minute!

How do we even find the card we want to flip in the first place?

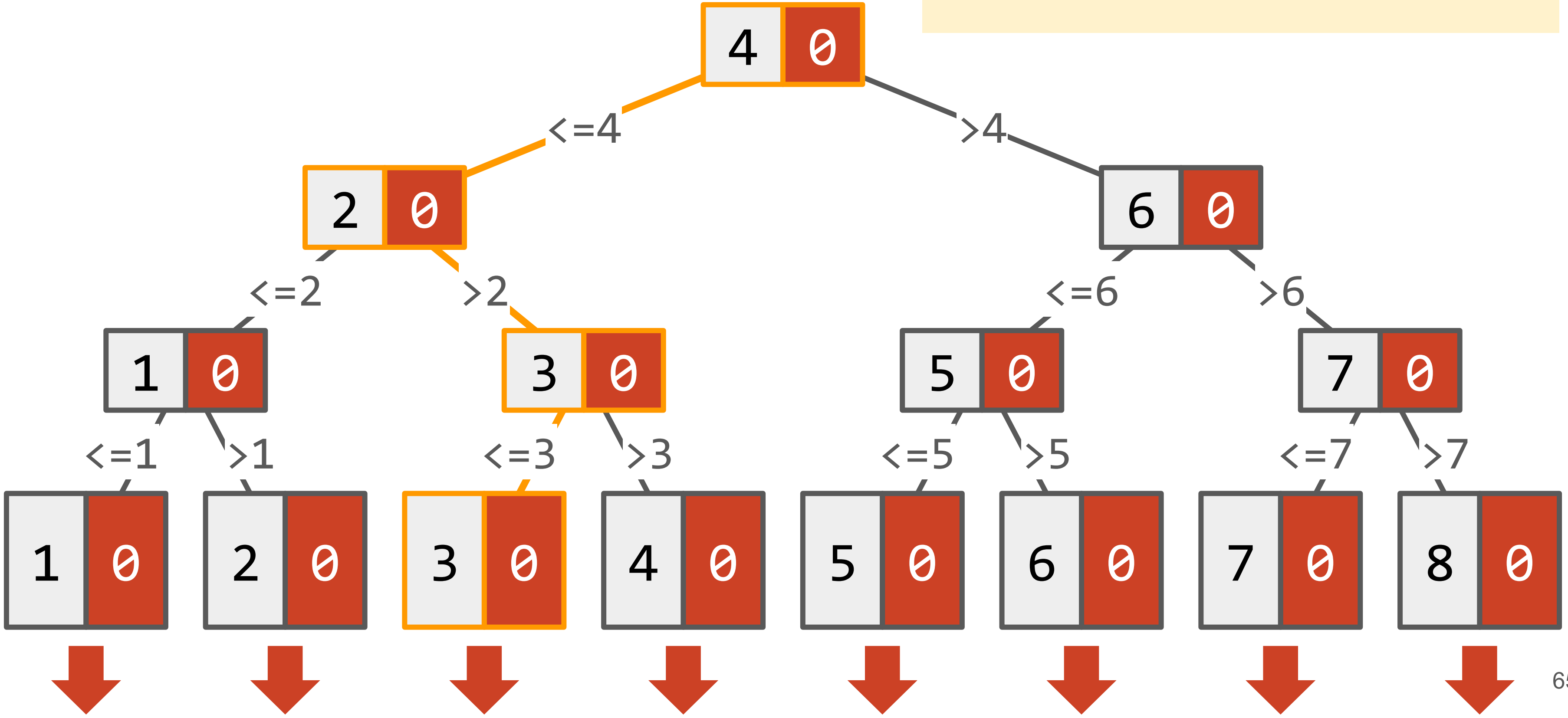
How do we `query(3)`?

How do we even navigate such a tree to do search?

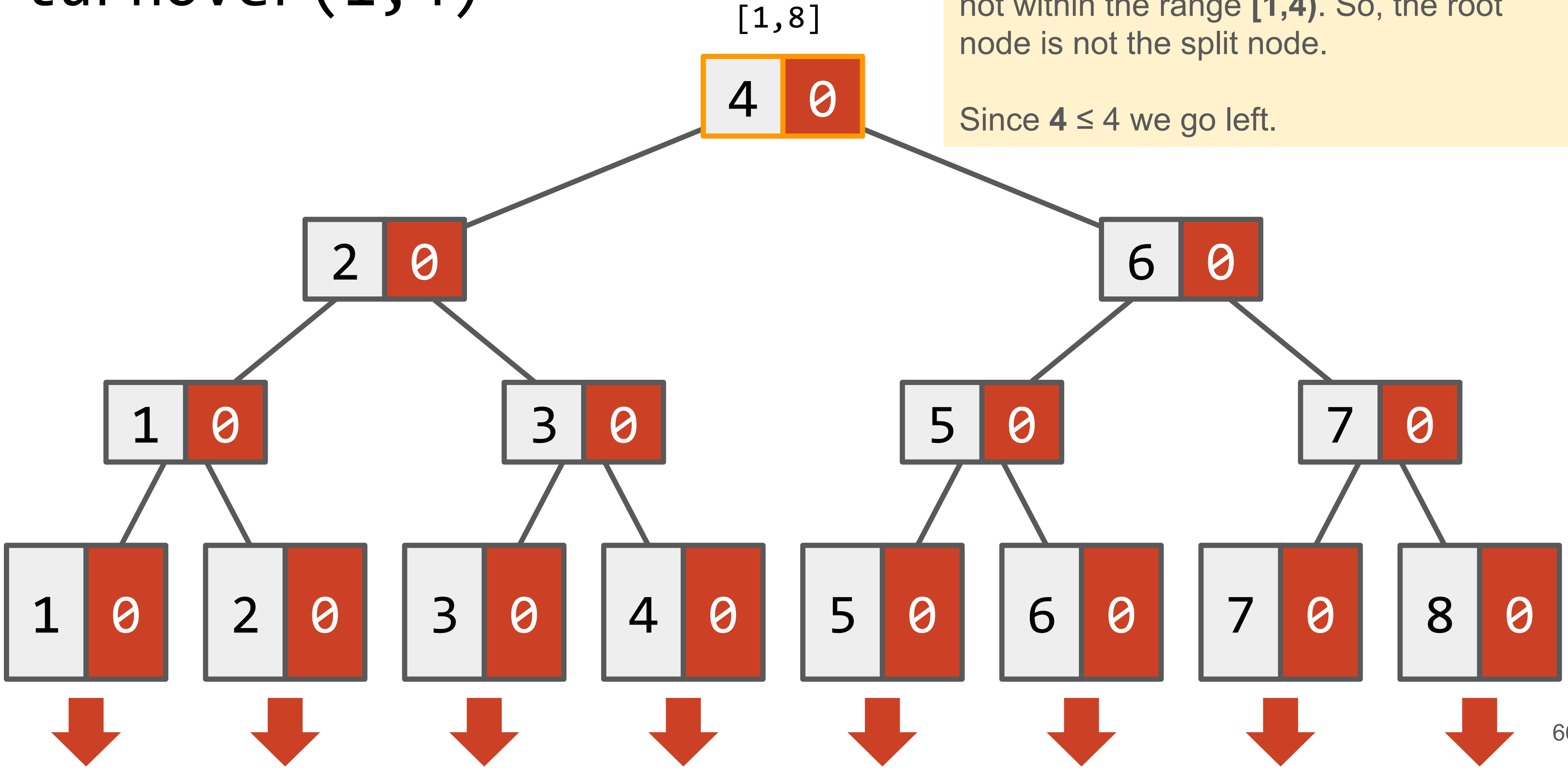


How do we `query(3)`?

Simple! We, turn it into a BST!
Keys of each node here is simply the max element in the left subtree.



```
turnOver(1,4)
```

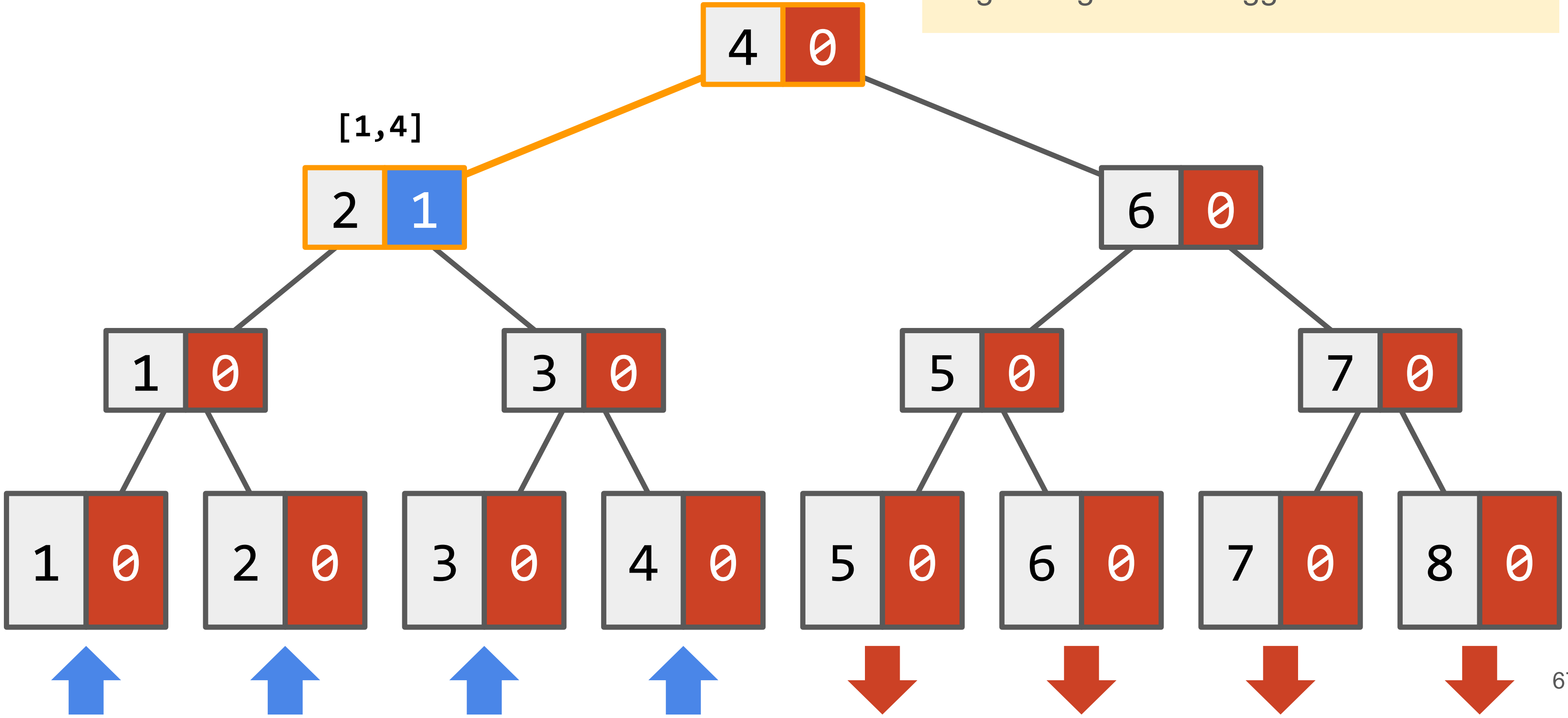


Here, our target range is **[1,4]** is contained within the root range **[1,8]** Again 4 (augmented data in the node) is not within the range **[1,4)**. So, the root node is not the split node.

Since **4** ≤ 4 we go left.

`turnOver(1,4)`

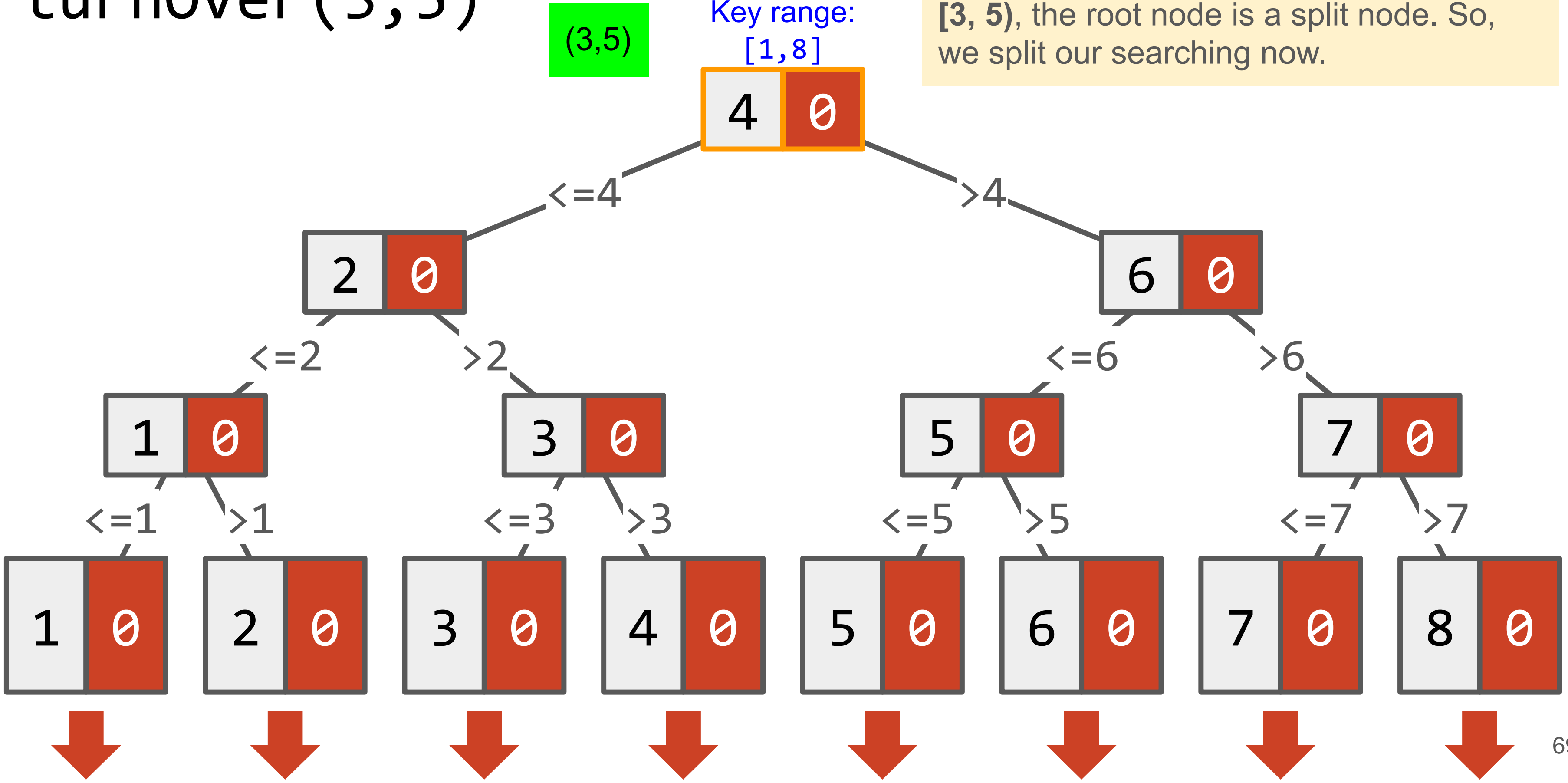
Current node has key range **`[1,4]`**. `which` completely matches with our target range so we toggle its state bit.



Problem 2.a. ー Approach

Can we now turn over cards in any subsequence?

`turnOver(3,5)`

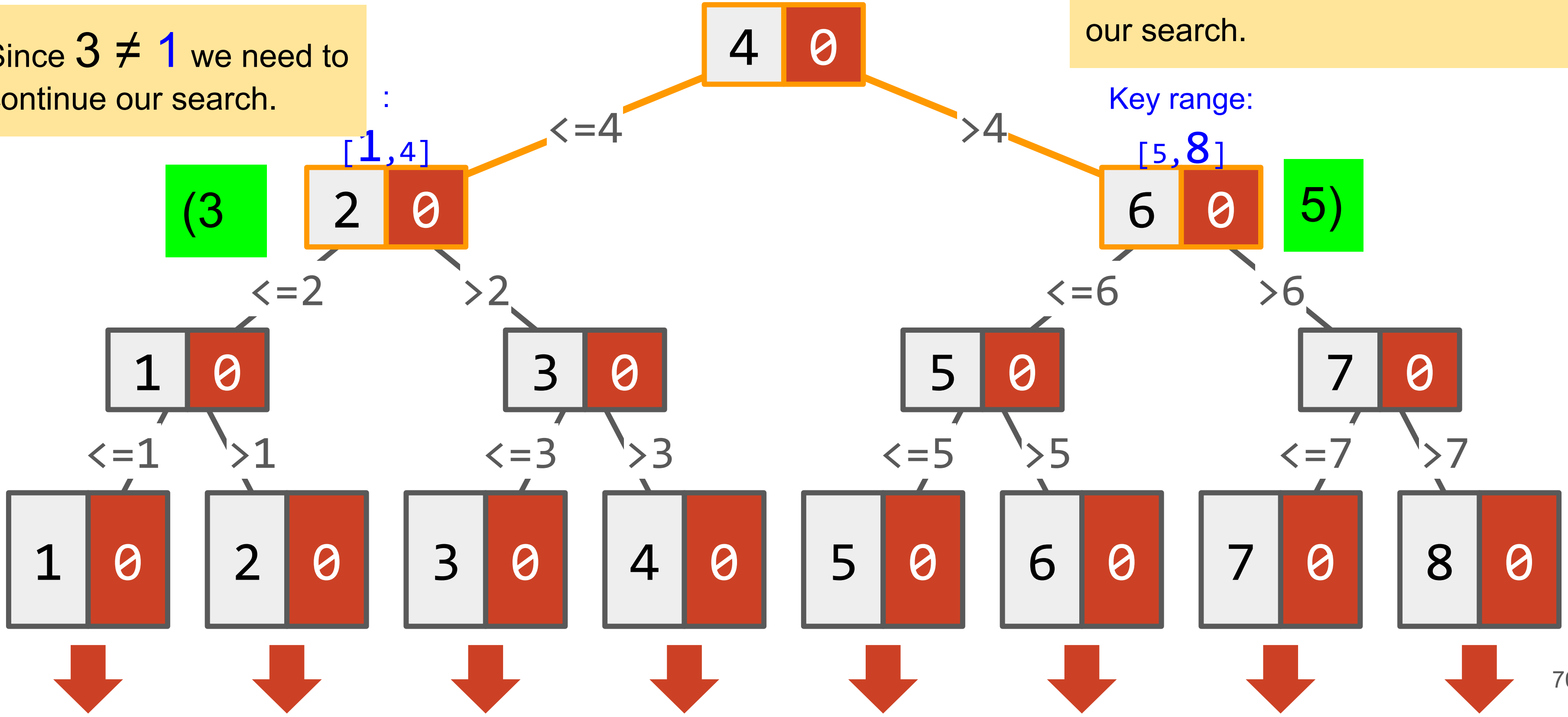


Target range is **`[3,5]`**. Since 4 is within **[3, 5)**, the root node is a split node. So, we split our searching now.

turnOver(3,5)

Since 3 ≠ 1 we need to continue our search.

Since 5 ≠ 8 we need to continue our search.



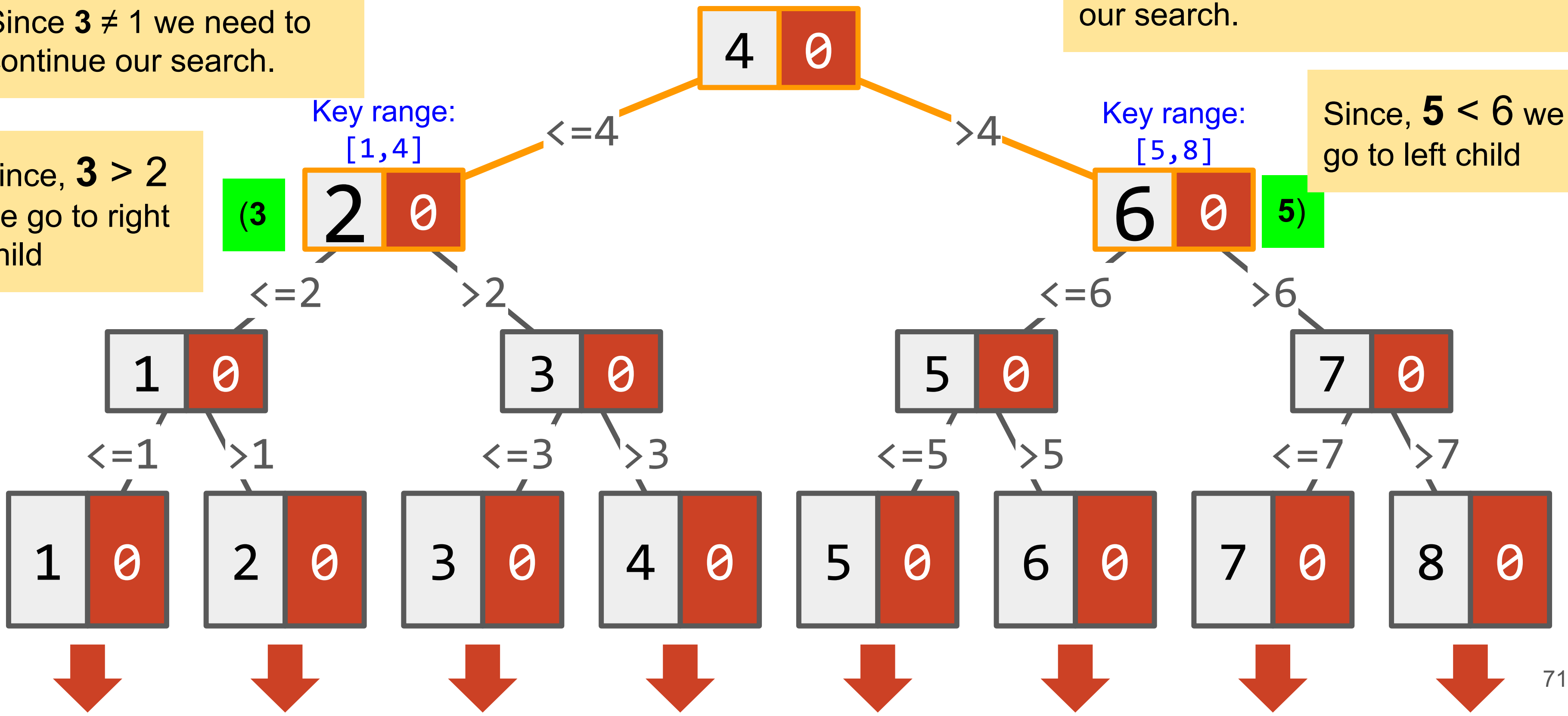
`turnOver(3,5)`

Since **3** ≠ 1 we need to continue our search.

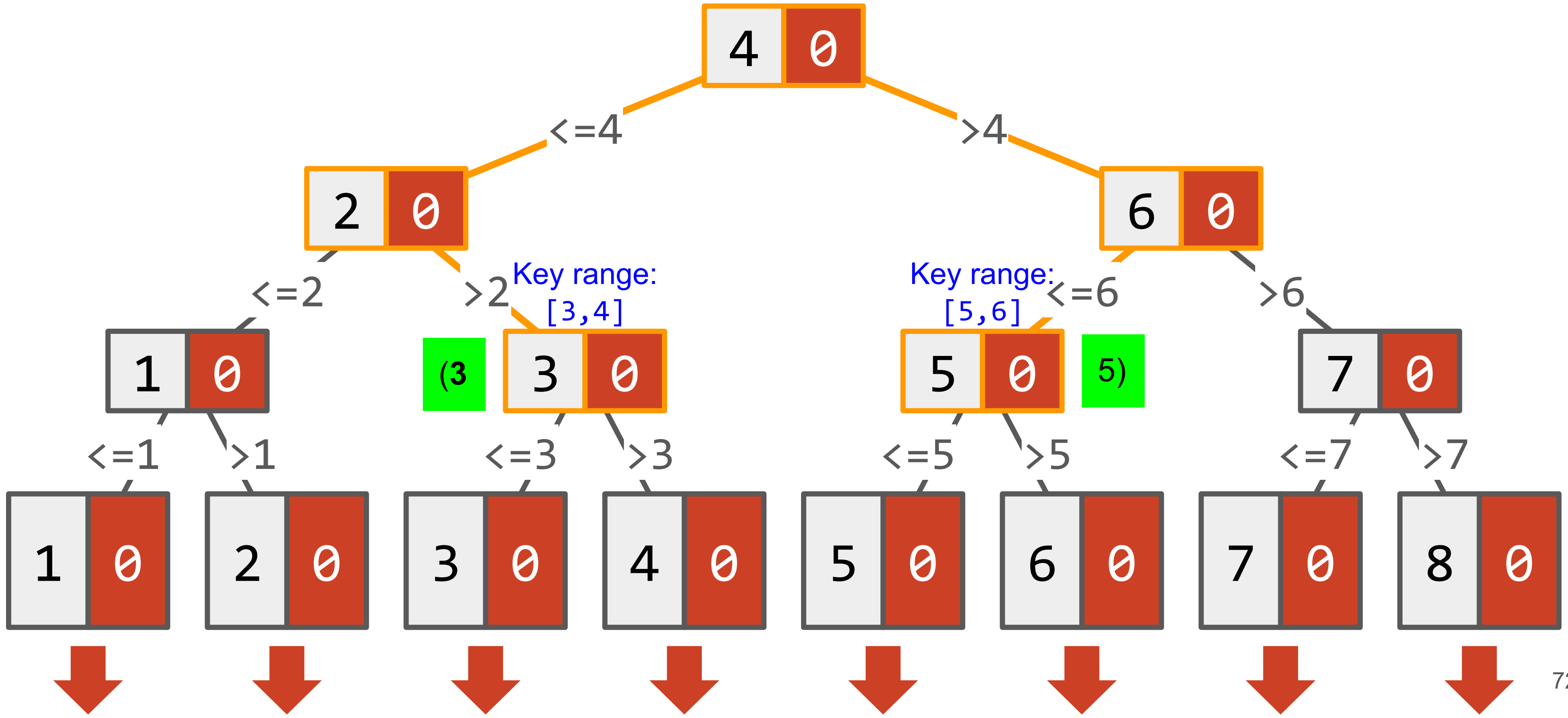
Since, **3** > 2 we go to right child

Since **5** ≠ 8 we need to continue our search.

Since, **5** < 6 we go to left child



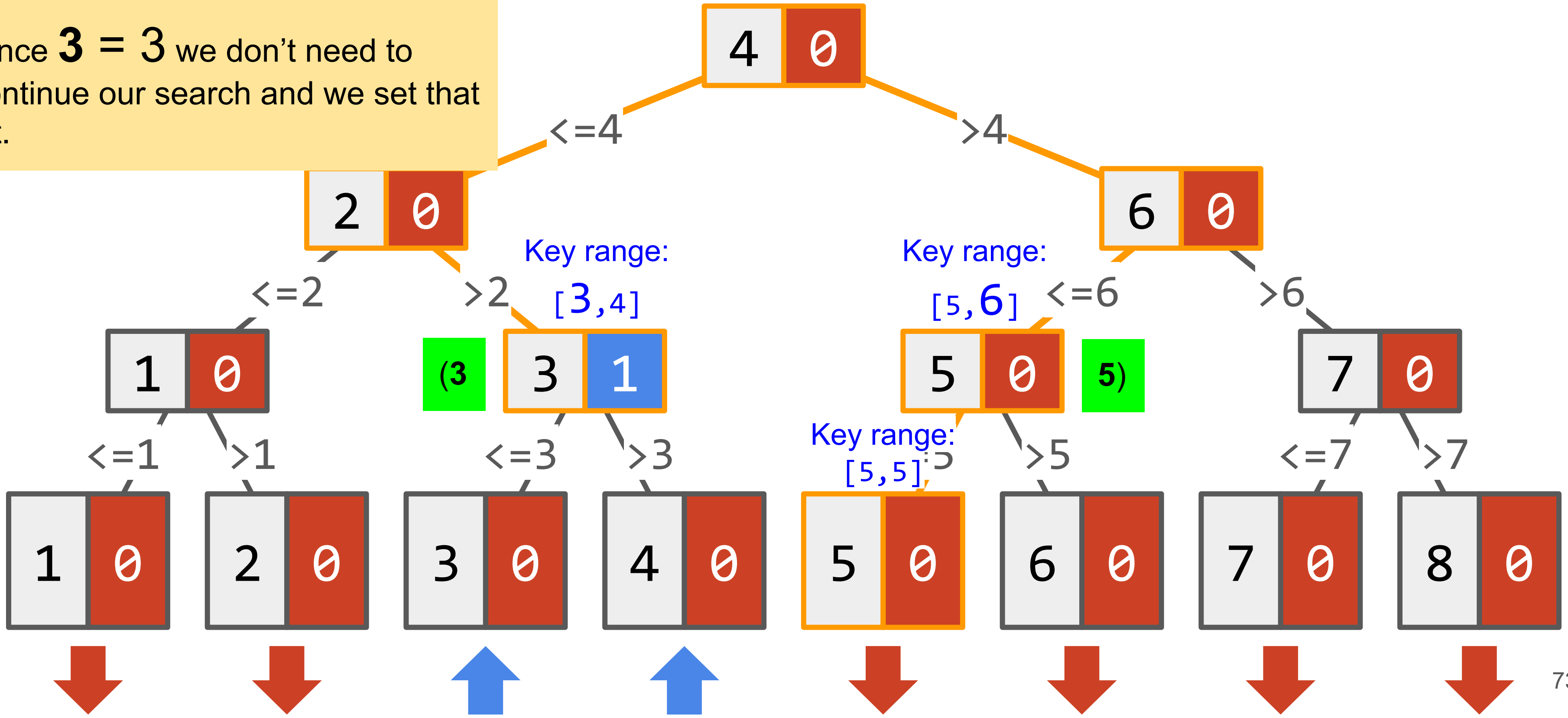
turnOver(3,5)



4 0
<=4
>4
2 0
6 0
<=2
>2
Key range: [3,4]
Key range: [5,6]
<=6
>6
1 0
(3
3 0
5 0
5)
7 0
<=1
>1
<=3
>3
<=5
>5
<=7
>7
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0

turnOver(3,5)

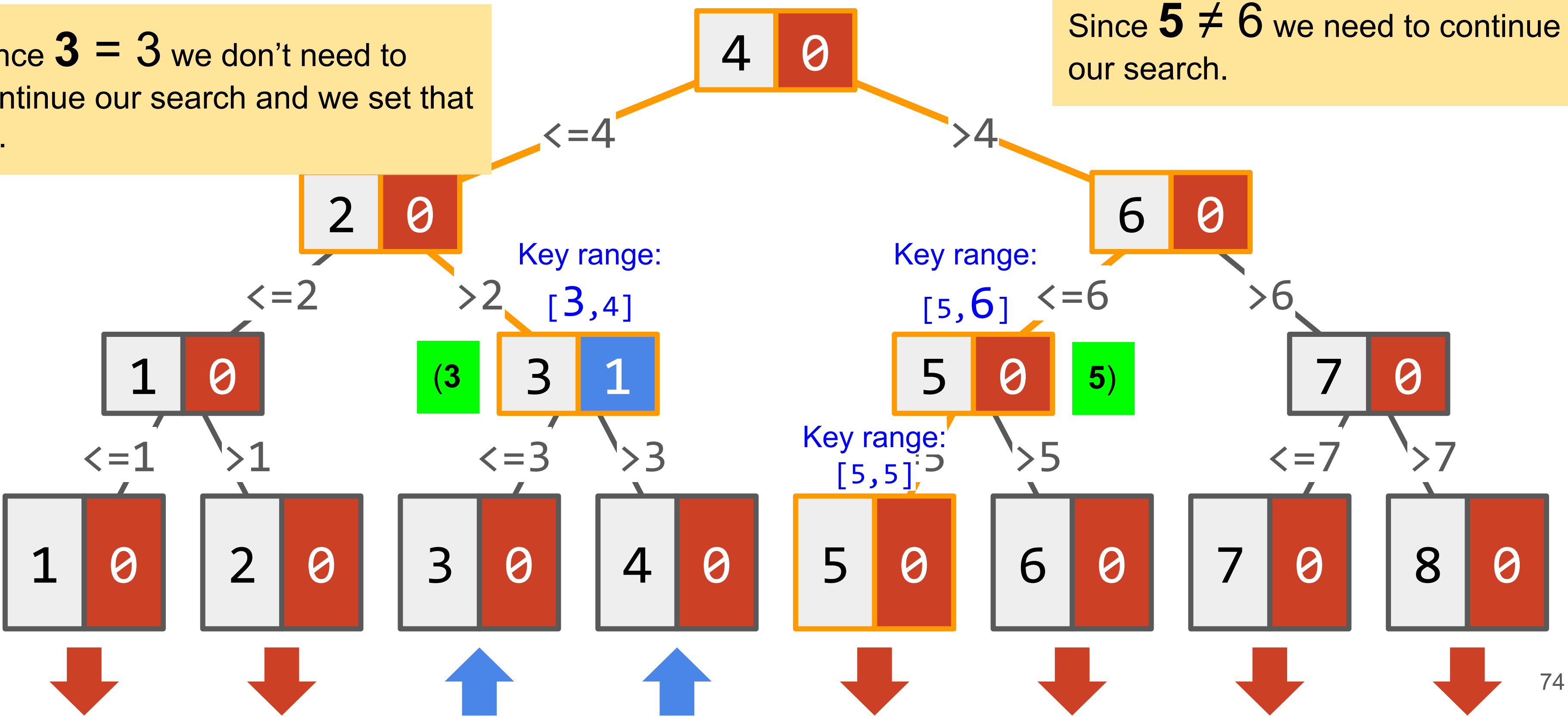
Since **3** = 3 we don't need to continue our search and we set that bit.



`turnOver(3,5)`

Since **3** = 3 we don't need to continue our search and we set that bit.

Since **5** ≠ 6 we need to continue our search.

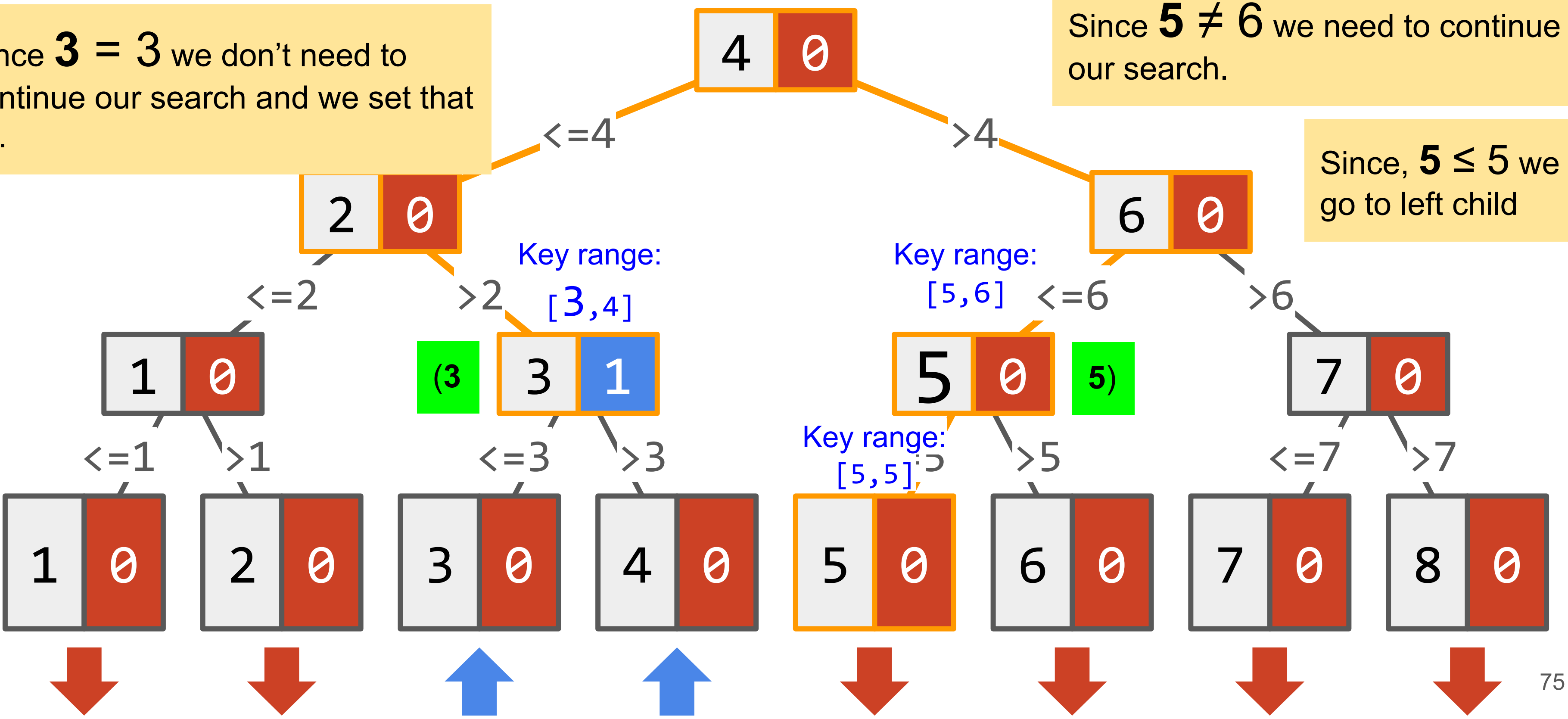


turnOver(3,5)

Since **3** = 3 we don't need to continue our search and we set that bit.

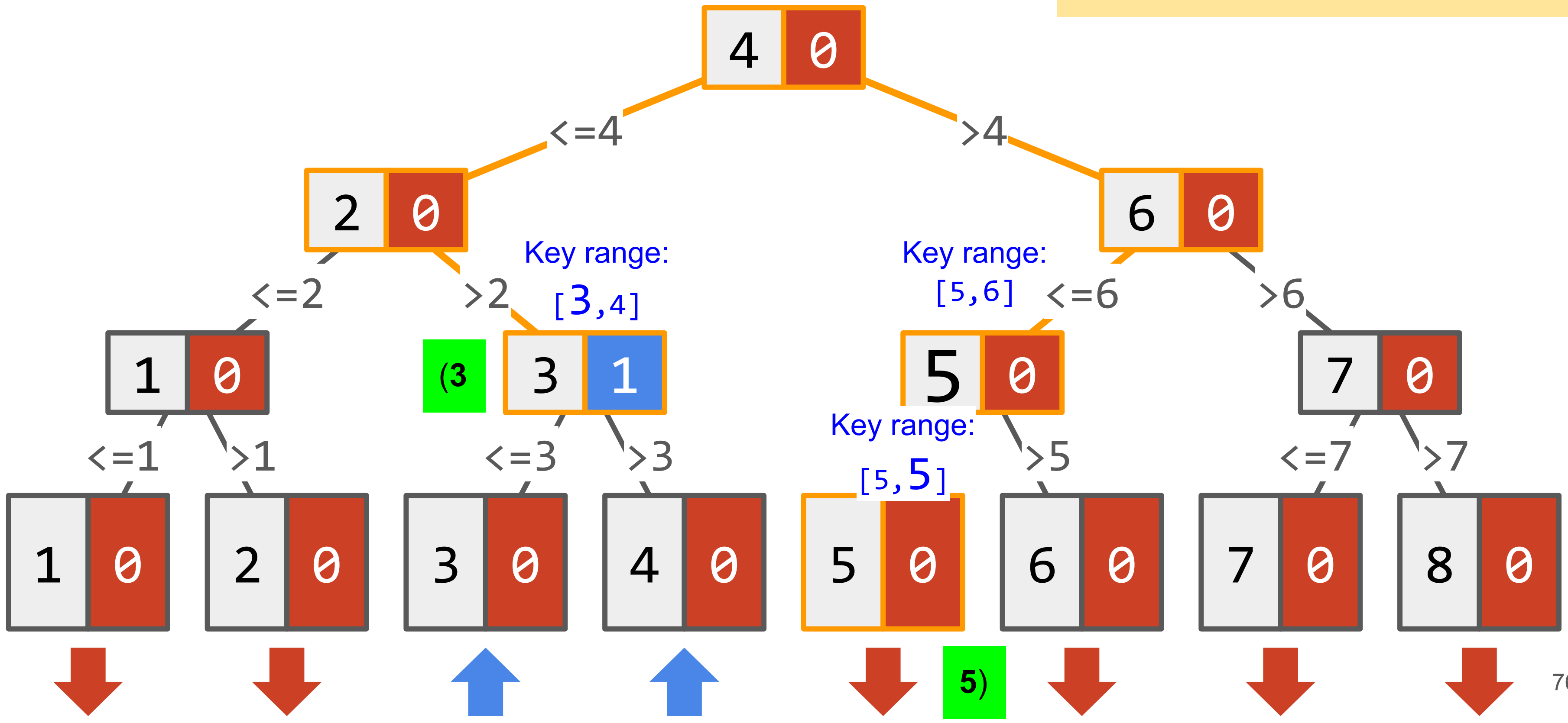
Since **5** ≠ 6 we need to continue our search.

Since, **5** ≤ 5 we go to left child



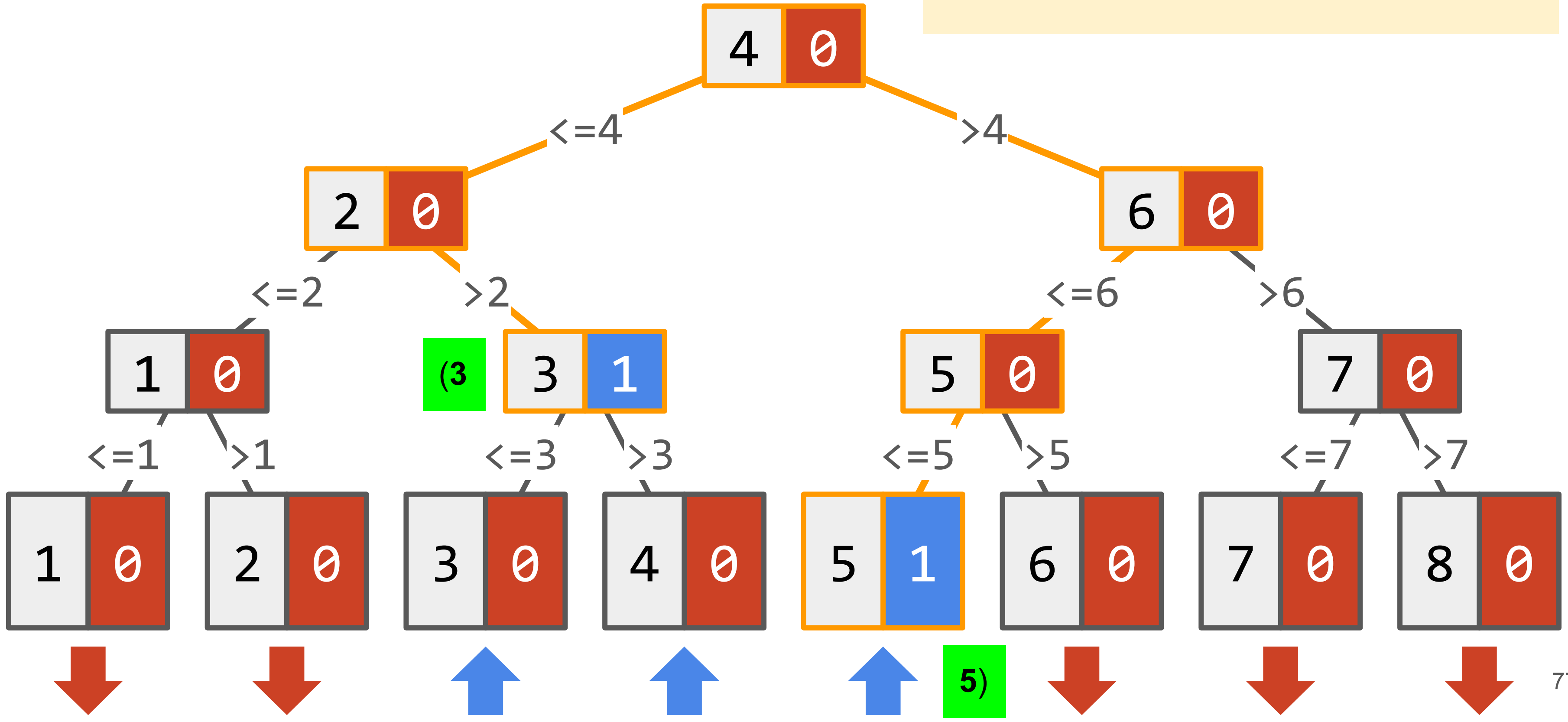
`turnOver(3,5)`

Since **5** = 5 we don't need to continue our search and we set that bit.



```
turnOver(3,5)
```

We are done!



Pseudocode for turnover

```
turnover(low, high):
    v = FindSplit(low, high);
    LeftTraversal (v, low, high);
    RightTraversal (v, low, high);
```

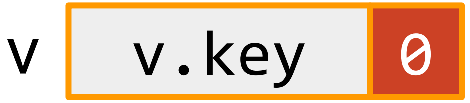
Target Range = (low, high)

This part of pseudocode is exactly same as the “1-D range search” pseudocode covered in lectures.

Look at the next slides for the subroutine calls - FindSplit, LeftTraversal and RightTraversal.

Pseudocode for turnover

```
FindSplit (low, high)
    Done = False;
    v = root;
    while (Done != True)
        if  $(high \le v.key)$  then v = v.left;
        else if  $(low > v.key)$ then v = v.right;
        else  Done = True;
    Return v
```

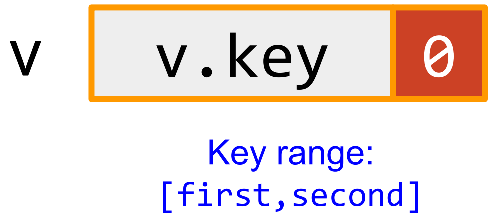


Target Range = (low, high)

This part of pseudocode is also exactly same as the FindSplit subroutine call done in “1-D range search” pseudocode covered in lectures.

Pseudocode for turnover

Target Range = (low, high)



```
LeftTraversal(v, low, high)
    If ( low != v.keyrange.first)

        then we need to continue our search

    else
    {
    set the bit of v from 0 to 1
    }
```

Pseudocode for turnover

Target Range = (low, high)

Key range:
`[first,second]`

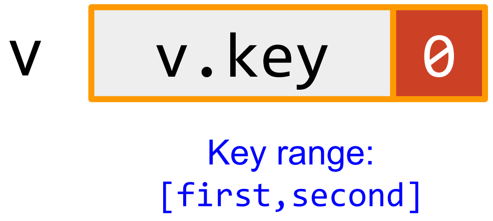
```
LeftTraversal(v, low, high)
    If ( low != v.keyrange.first)
    {
        If ( low ≤ v.key) then
            Set the bit of v.right from 0 to 1
            LeftTraversal (v.left, low, high)
        else  LeftTraversal (v.right, low, high)
    }
    else  {
    set the bit of v from 0 to 1
    }
```

This part of the pseudocode is not captured in the example “turnover(3,5)”. Try to dry run “turnover(2,7)”

Pseudocode for turnover

```
RightTraversal(v, low, high)
    If ( high != v.keyrange.second)
    {
        If ( high > v.key) then
            Set the bit of v.left from 0 to 1
            RightTraversal (v.right, low, high)
        else  RightTraversal (v.left, low, high)
    }
    else  {
    set the bit of v from 0 to 1
    }
```

Target Range = (low, high)



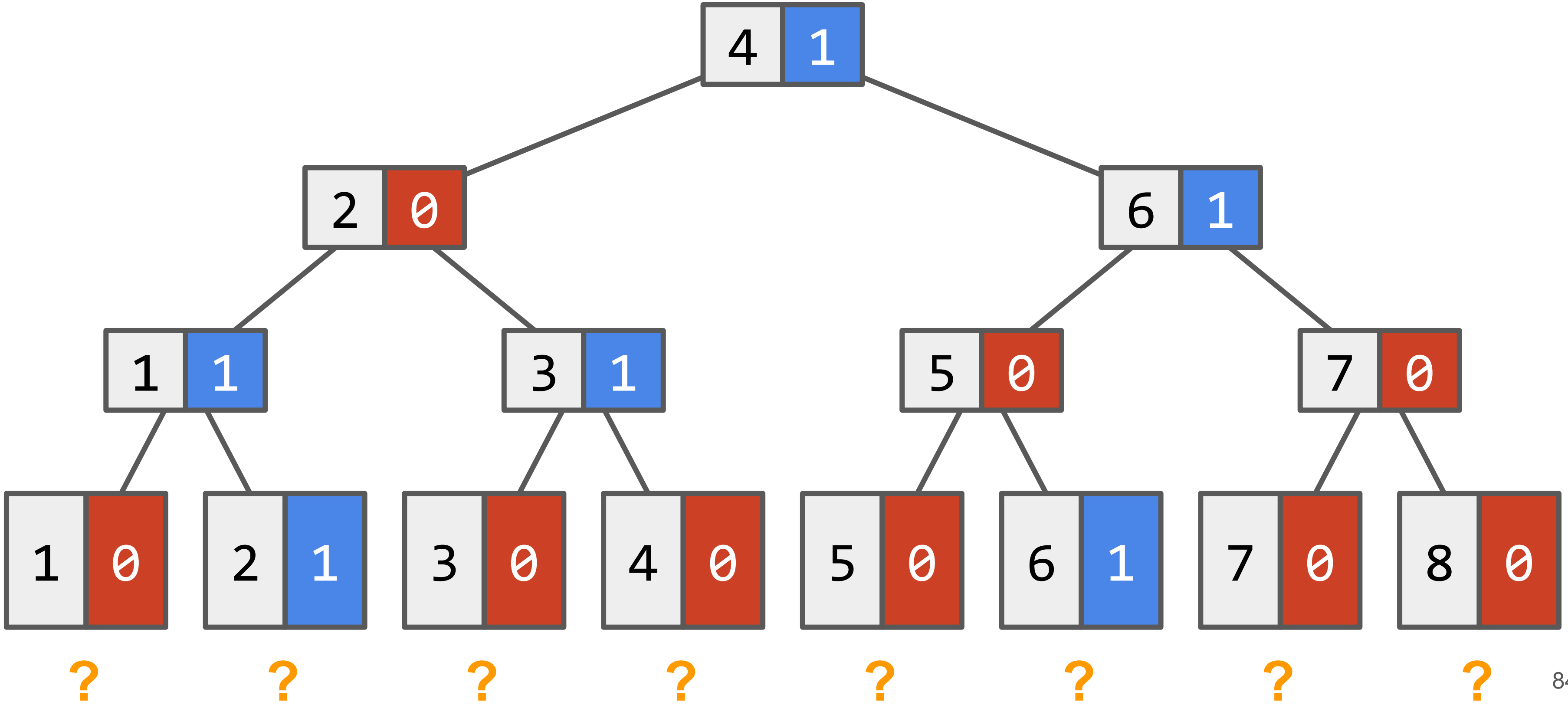
This part of the pseudocode is not captured in the example “turnover(3,5)”. Try to dry run “turnover(2,7)”

Problem 2.a.

But wait a minute!

How do we determine the face direction of a card?

What are the face directions?



Guiding question

How do we implement `query(i)`?

Guiding question

How do we implement `query(i)`?

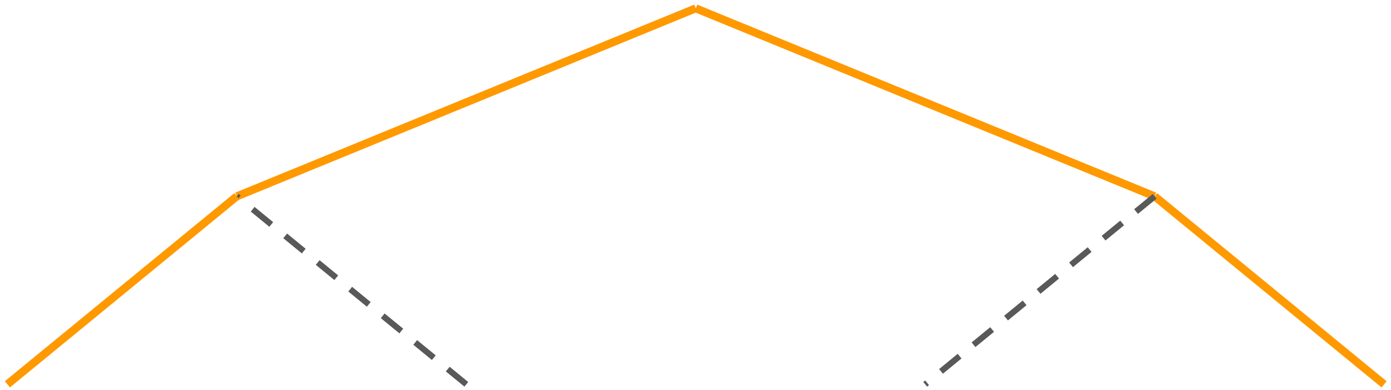
Answer:

3 equivalent methods:

1. Count the number of 1’s in the root-to-leaf path, even number means card is same as initial state, odd means otherwise
2. `XOR` all the bits encountered in the root-to-leaf path, result of 0 means card is same as initial state, 1 means otherwise
3. Initialize a variable direction `d` with initial face direction of cards; traverse the tree from root to leaf, every time we encounter a 1, we toggle the direction of `d`; at the end of the traversal, `d` will be the card’s final face direction

Test yourself!

Once there is a left-right split in the search path, can the left branch have a further left split and the right branch have a further right split later on? E.g.



Test yourself!

So what are some bad cases for `turnOver`?

Test yourself!

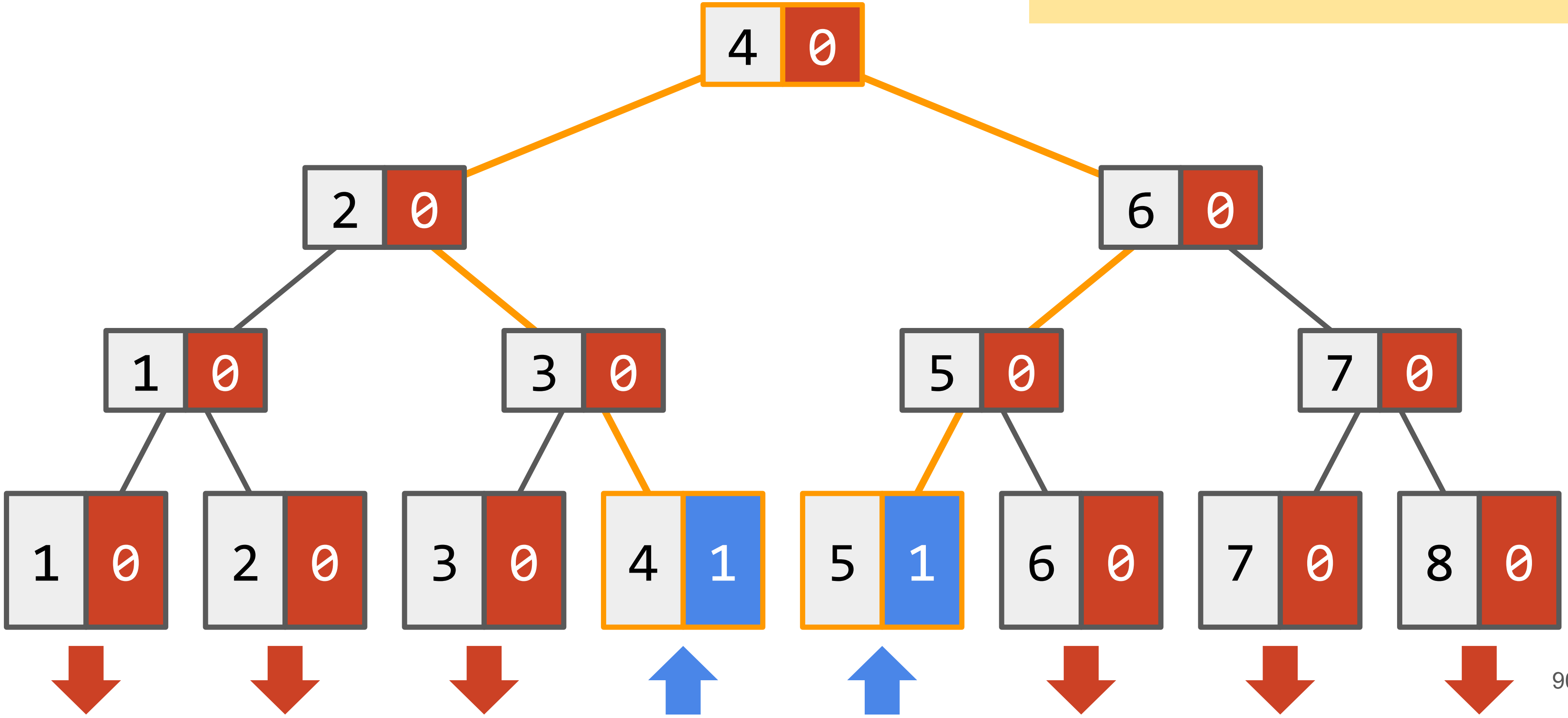
So what are some bad cases for `turnOver`?

Answer:

- `turnOver(`$\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1$`)`
- `turnOver(`$2, n-1$`)`

turnOver(4,5)

Target range is **[4,5]**. Since 4 is within **[4, 5)**, the root node is a split node. So, we split our searching now.



Test yourself!

What is the time complexity of `turnOver`?

Test yourself!

What is the time complexity of `turnOver`?

Answer: Worst case takes $2 \log n$

Therefore time complexity is $O(\log n)$.

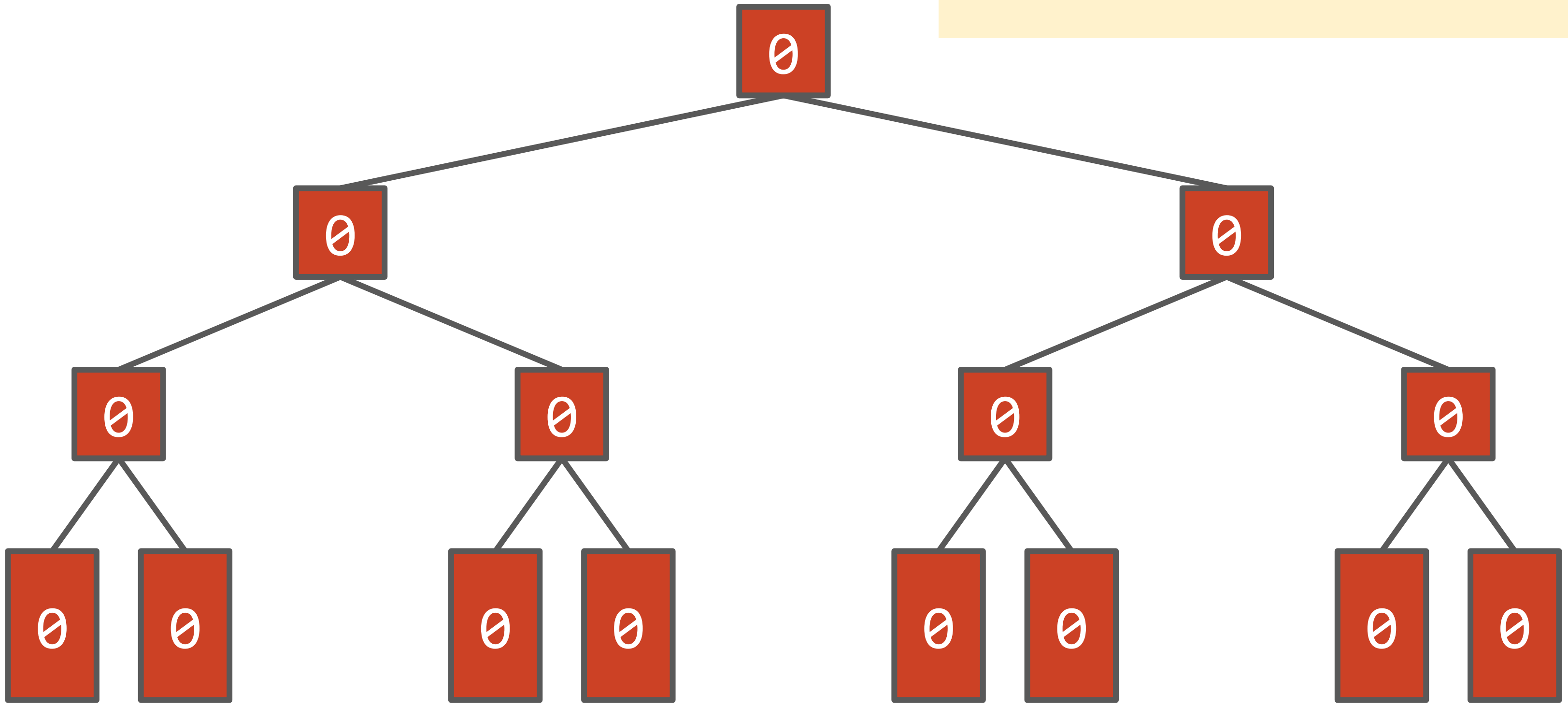
Challenge yourself!

Do we always need a BST to help us navigate the search?

Can we just store the state at each node and do away with the key?

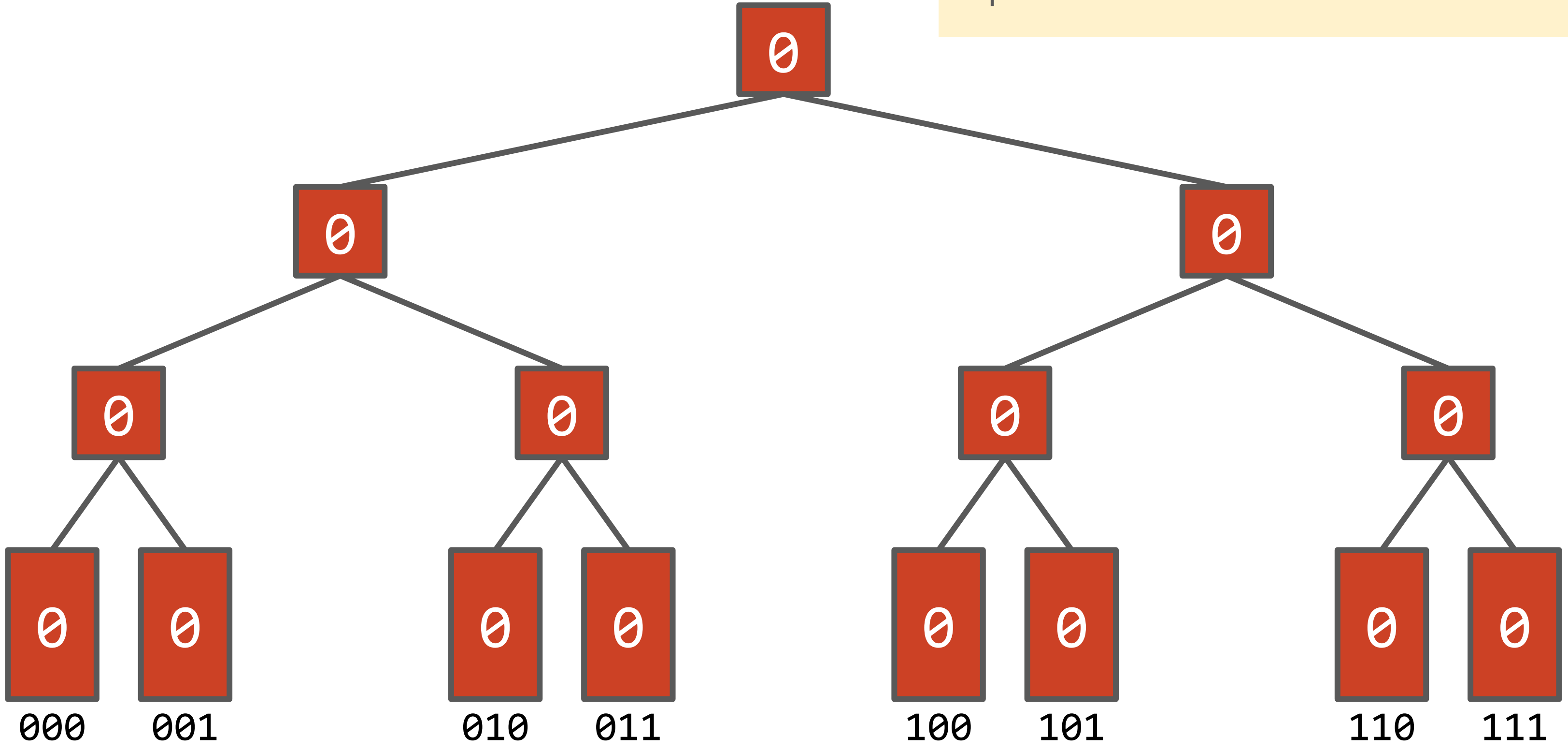
Challenge yourself!

In other words, can we just leave it like this?



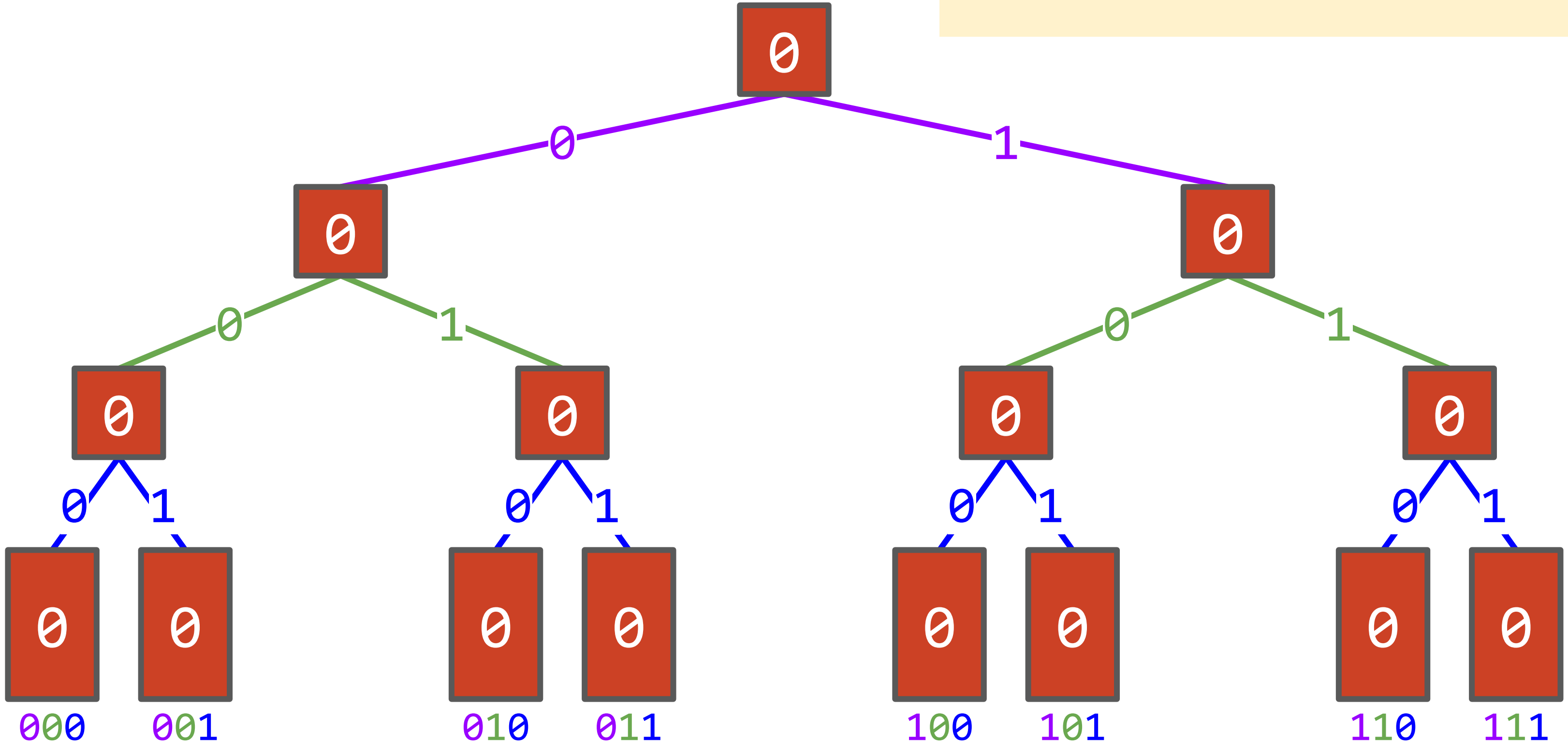
Challenge yourself!

We index cards from `0` to `7`
What do you observe from the binary representation of numbers from `0` to `7`?



Challenge yourself!

Go left: `0`
Go right: `1`

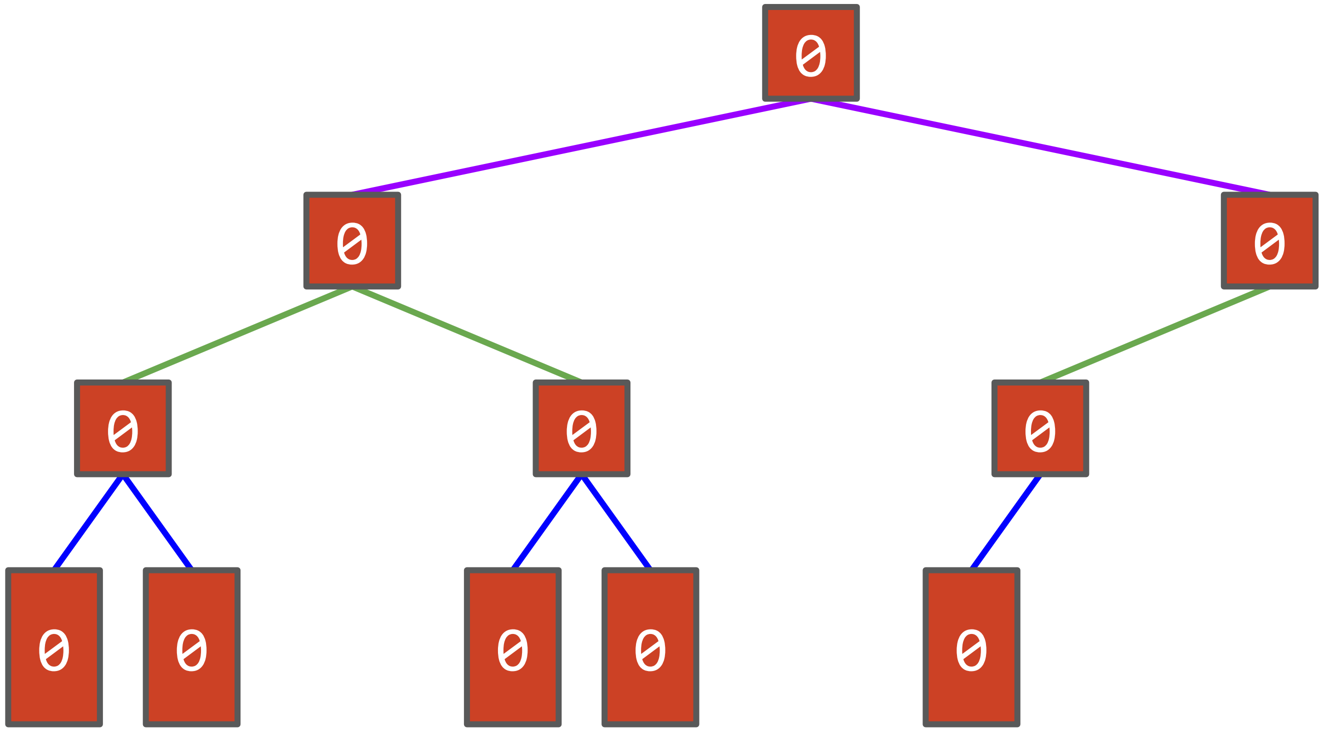


Test yourself!

What if the number of cards we have isn't a power of 2?

Tree with 5 cards

Here, levels needed to capture 5 cards is $\lceil \log_2 5 \rceil$ which is still 3



Alternative solution: Overview

- Cards are indexed from 1 to n
- Maintain a BST B (initially empty) where the keys within are card indices
- If card i is in B, it means that we performed `turnOver(`i,n`)`
- Insight: `turnOver(`i,j`)` can be decomposed into `turnOver(`i,n`)` and `turnOver(`$j+1,n$`)`, let's call them *sub-toggles*
- To encode a `turnOver(`i,j`)` in B, we want to toggle the keys i and $j+1$
    - If key is not yet in B, insert it
    - Else remove key
- To determine the face direction of a card i, we just need to check the number of *overlapping* sub-toggles affecting it
    - If the number of overlapping sub-toggles is odd then the card is flipped from its initial state, else it's even then there is no net resultant change
    - The number of overlapping toggles is the number of keys less than i in B

Test yourself!

How to obtain the number of keys in a BST less than i?

What is an efficient way to implement it?

Test yourself!

How to obtain the number of keys in a BST less than i?

What is an efficient way to implement it?

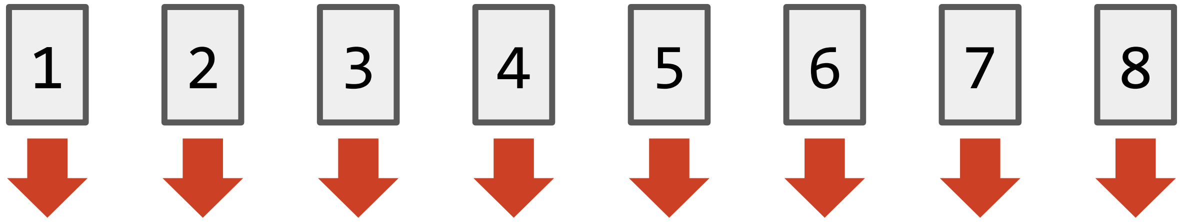
Answer: it’s the rank of i in the BST.

The rank of a key can be obtained in $O(\log n)$ if nodes in the BST (with n nodes) is augmented with weight.

Alternative solution: Example

Initial state

Cards:

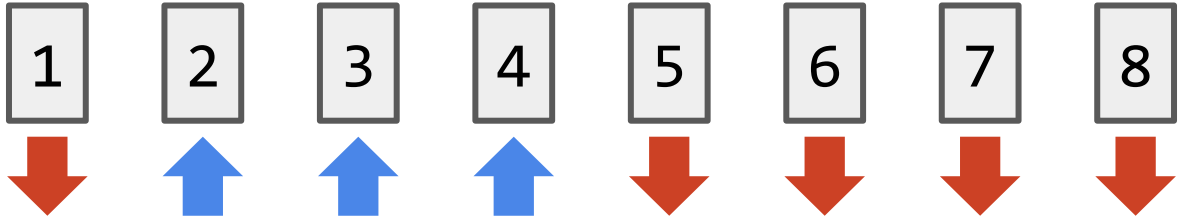


BST:

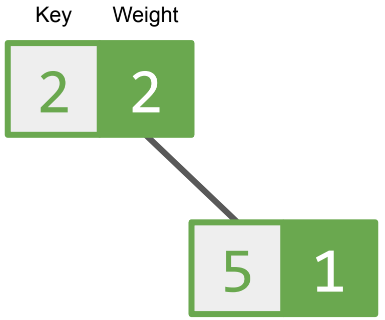
Alternative solution: Example

`turnOver(2,4)`: insert keys `2` and `5` into BST

Cards:



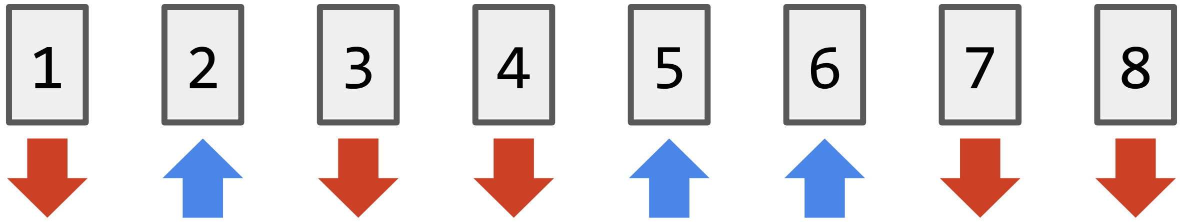
BST:



Alternative solution: Example

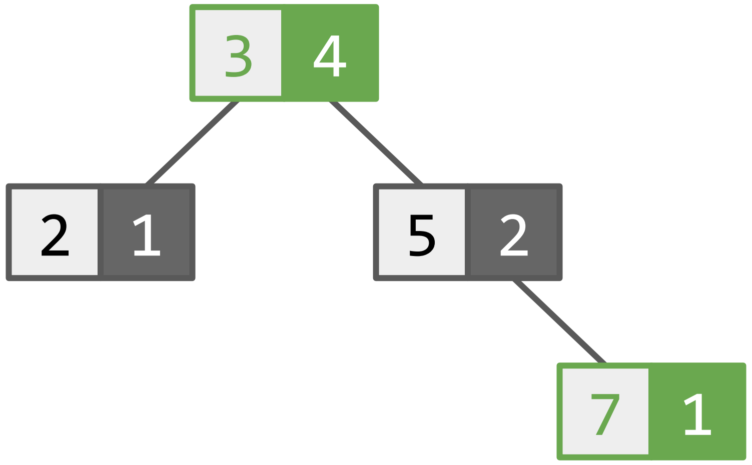
`turnOver(3,6)`: insert keys `3` and `7` into BST

Cards:



BST:

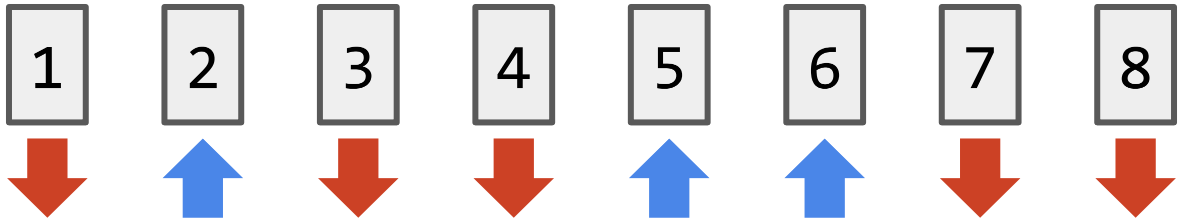
Displayed after AVL-tree rotations



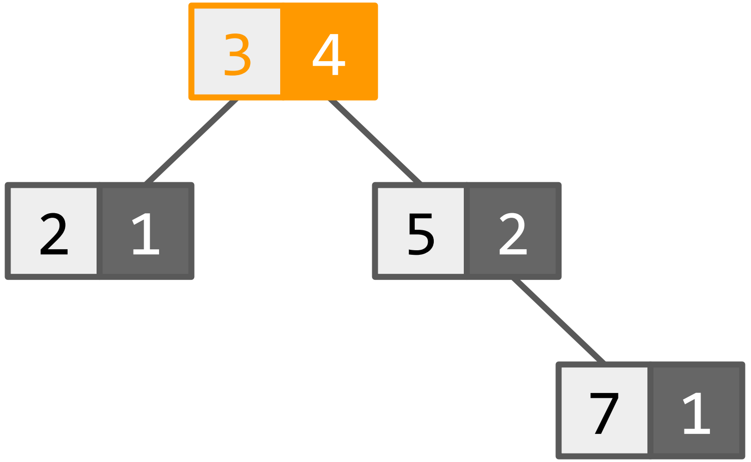
Alternative solution: Example

`query(3)`: key `3` in BST has rank 2 (even) so same as initial state.

Cards:



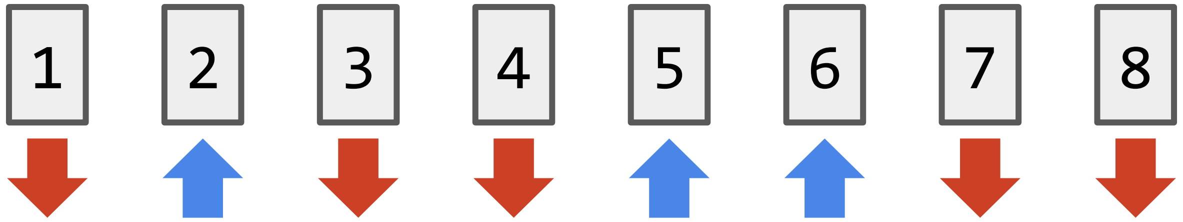
BST:



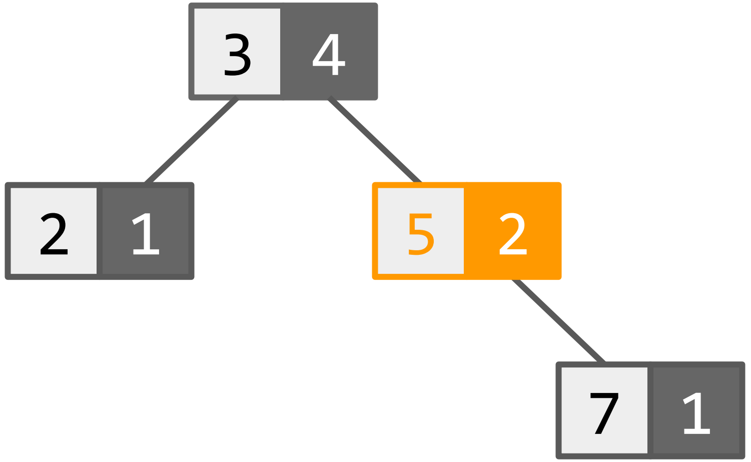
Alternative solution: Example

`query(5)`: key `5` in BST has rank 3 (odd) so opposite of initial state.

Cards:



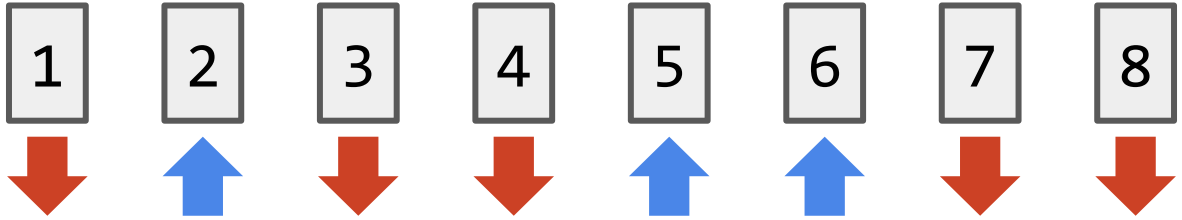
BST:



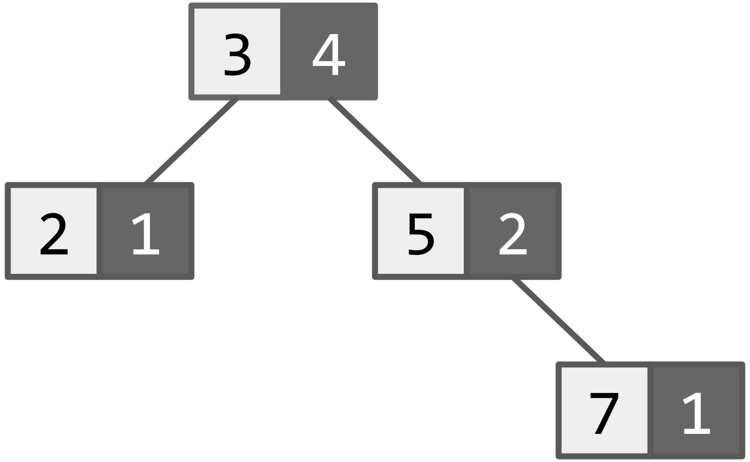
Alternative solution: Example

`query(4)`: There are 2 (even) keys less than 4 it in BST so same as initial state.

Cards:



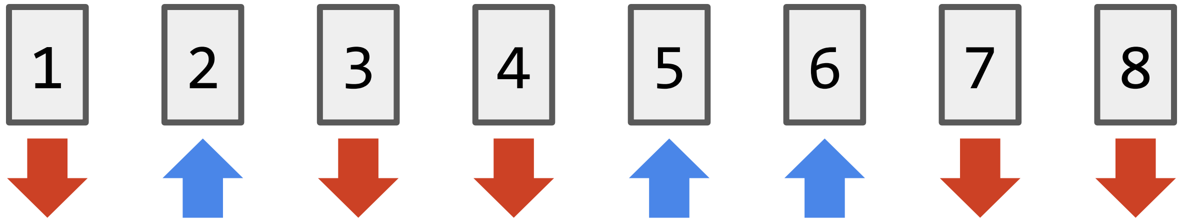
BST:



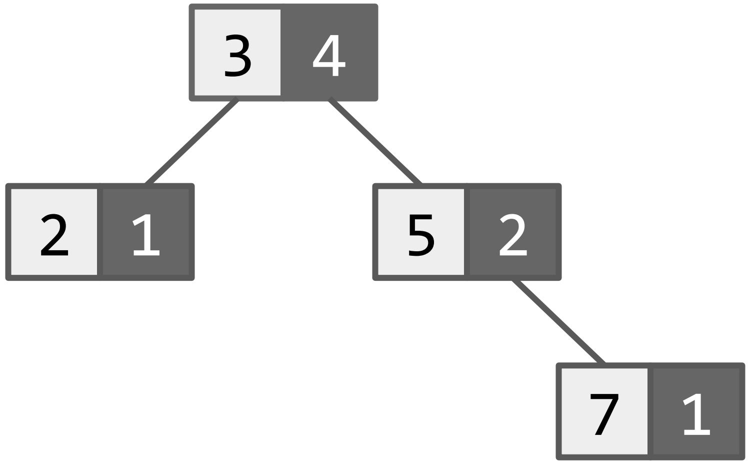
Alternative solution: Example

`query(6)`: There are 3 (odd) keys less than 6 it in BST so opposite of initial state.

Cards:



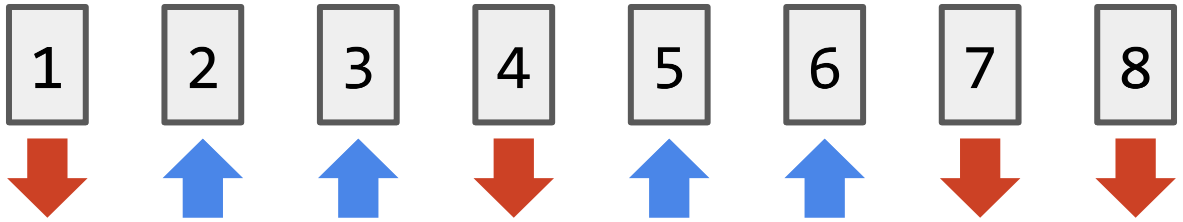
BST:



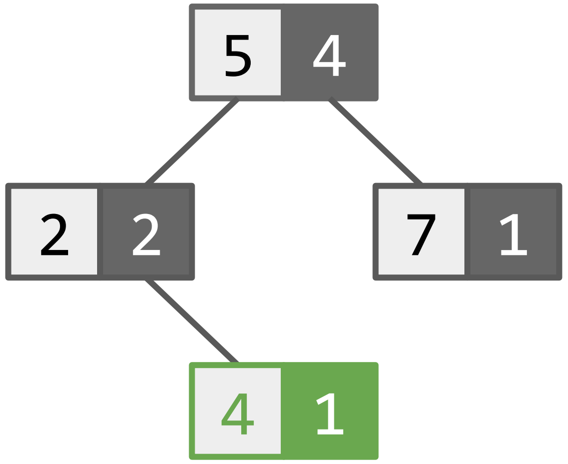
Alternative solution: Example

`turnOver(3,3)`: delete key `3`, insert key `4` into BST

Cards:



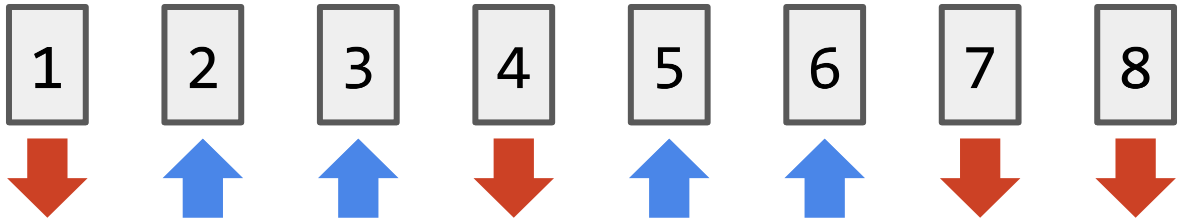
BST:



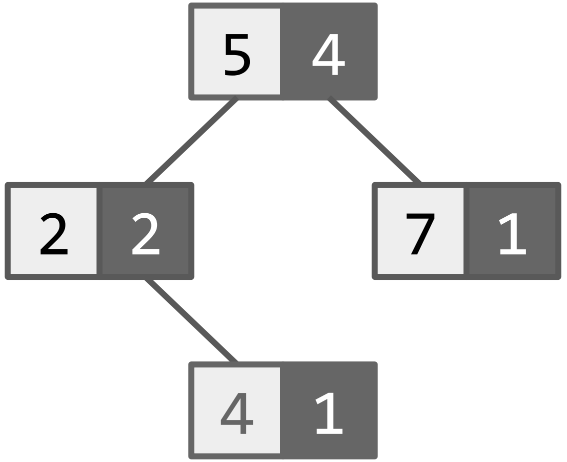
Alternative solution: Example

`query(3)`: There are 1 (odd) keys less than 3 it in BST so opposite of initial state.

Cards:



BST:



Alternative solution: Time complexity

Assuming there are k number of `turnOver` operations called and therefore $2k$ nodes in the BST in the worst case.

`query(i)`: $\mathrm{O}(\log k)$ to find out the rank of i. (What if i is not in the tree?)

`turnOver(i,j)`: $\mathrm{O}(\log k)$ since 2 key searches needed, each search will traverse the $\mathrm{O}(\log k)$ height of the tree in the worst case.

Alternative solution: Space complexity

Space complexity is simply $O(n)$ since in the worst case, all the card indices are in the tree.

Algo stonks