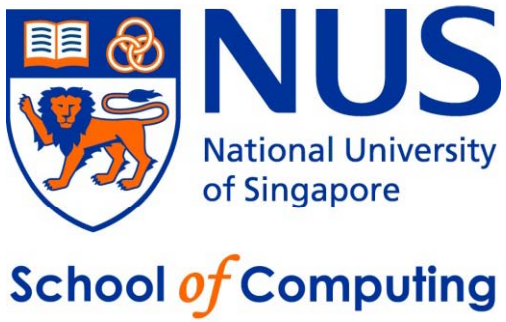
CS2010 – Data Structures and Algorithms II

Lecture 02 – Census Problem

stevenhalim@gmail.com



Admin Slide

- This slide will be updated on Tue, 21 Aug 2012
  - You can choose not to print this slide

- CP2.5 Book Sales (Part 2)
- CS2010 Game System
- Tutorial and Lab Bidding Status
- Some more advertisements

Outline

- What are we going to learn in this lecture?
  - Motivation: Census Problem
    - Abstract Data Type (ADT) Table
    - Solving Census Problem with CS1020 Knowledge
      - Using unsorted array versus sorted array
    - The “performance issue”
  - Binary Search Tree (BST)
    - Definition
    - Search, Insert, FindMin/FindMax, complexity analysis
    - Inorder traversal, Successor/Predecessor, analysis
    - Deletion, analysis
  - Relation with CS2010 PS1: “The Baby Names Problem”

Governments/Companies/This module do Census (Survey) for many purposes

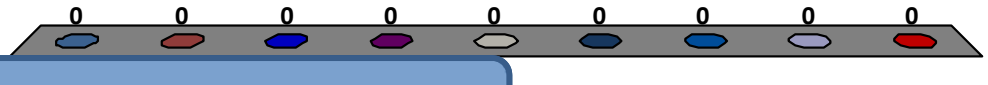
- Interesting statistics about Singapore (FYI)
  - http://www.singstat.gov.sg
- Sun Tzu's Art of War Ch 1 "The Calculations"
  - If you know your enemies and know yourself, you will not be imperiled in a hundred battles
- Census is important!
- Let's do one such census in CS2010 ☺

Your Age:

'[' (or ']') means that endpoint is included (closed)

'(' (or ')') means that endpoint is **not** included (open)

1. [24 … ∞)
2. [23 … 24)
3. [22 … 23)
4. [21 … 22)
5. [20 … 21)
6. [19 … 20)
7. [18 … 19)
8. [17 … 18)
9. [0 … 17)

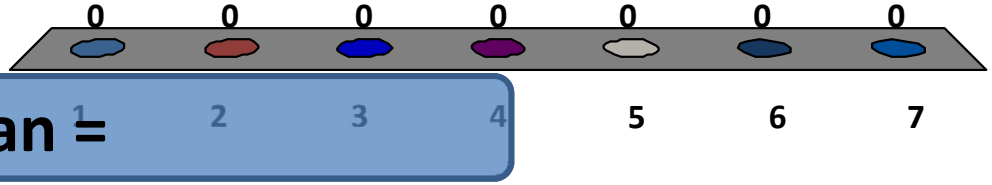


0 of 120

Mean =

Your Major:

1. Computer Science (CS)
2. Communications and Media (C&M)
3. Computer Engineering (CEG/CEC)
4. Comp. Biology (CB)
5. Information System (IS)
6. Science Maths (SCI)
7. None of the above :O

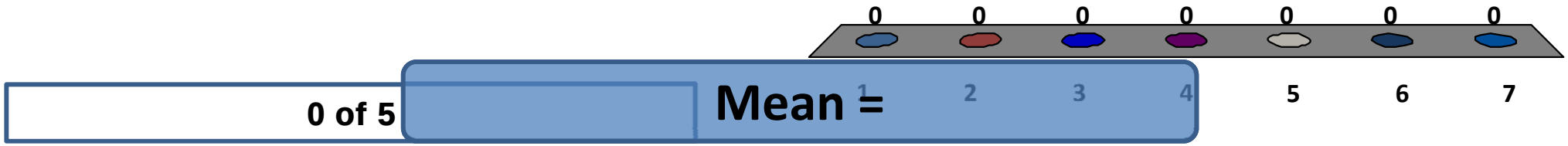


0 of 120

Mean =

Your Nationality:

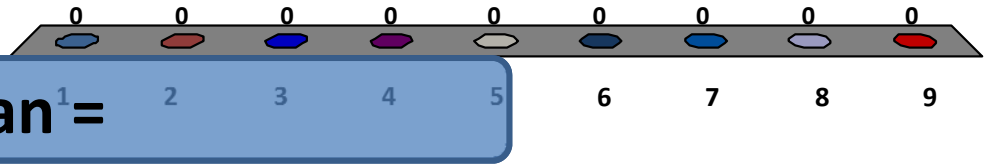
1. Singaporean
2. Chinese
3. Indonesian
4. Indian
5. Vietnamese
6. Malaysian
7. None of the above (tell me)



Your CAP:

(remember, the clicker system is anonymous!)

1. [4.5 … 5.0]
2. [4.25 … 4.5)
3. [4.0 … 4.25)
4. [3.75 … 4.0)
5. [3.5 … 3.75)
6. [3.25 … 3.5)
7. [3.0 … 3.25)
8. [0.0 … 3.00)
9. I do not want to tell



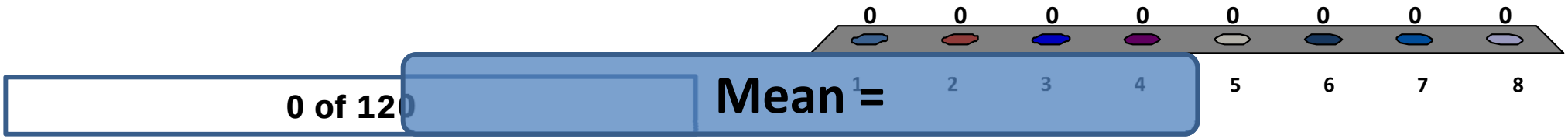
0 of 120

Mean =

My CS1020 Grade

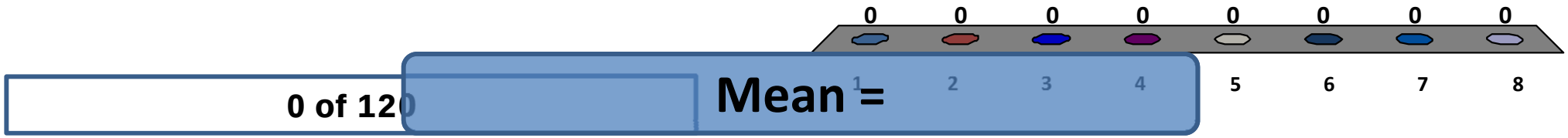
(I already know your CS1231 profile last week)

1. A+
2. A
3. A-
4. B+
5. B
6. B-
7. Just pass (> F)
8. I do not want to tell



Your Grade Expectation for CS2010

1. A+
2. A
3. A-
4. B+
5. B
6. B-
7. Just pass (> F)
8. I do not want to tell



What Happen After Census?

- Data Mining!
- Statistical Analysis!

- What do you need to do that?
  - An efficient data structure!

Abstract Data Type (ADT) **Table**

- Let's deal with one aspect of our census: **Age**
  - And to simplify this lecture, we assume that students' age ranges from [0 … 100), all integers, and distinct
- Required operations:
  1. Search whether there is a student with a certain age?
  2. Insert a new student (that is, insert his/her age)
  3. Determine the youngest and oldest student
  4. List down the ages of students in sorted order
  5. Find a student slightly older than a certain age!
  6. Delete existing student (that is, remove his/her age)
  7. Determine the median age of students

CS1020 Knowledge (1)

- If we use an **unsorted array** of size **n…**, e.g.
  - A = {5, 7, 71, 50, 23, 4, 6, 15} → Ok, the values are 'random'
- Required operations:
  1. Search whether there is a student with a certain age?
     - O(n) – go through the entire array until you reach the last item or until you reach an empty cell
  2. Insert a new student (that is, insert his/her age)
     - O(1) – simply insert at the very back
  3. Determine the youngest and oldest student
     - O(n) – go through the entire array and keep track of the smallest and largest integer value

CS1020 Knowledge (2)

– A = {5, 7, 71, 50, 23, 4, 6, 15}

4. List down the ages of students in sorted order
   - O(n log n) – we have to call an efficient **sorting** algorithm
5. Find a student slightly older than a certain age!
   - O(n) – we have to go through the entire array
6. Delete existing student (that is, remove his/her age)
   - O(n) – after removing a certain age, we have to ensure that there is no gap in our unsorted array, otherwise our search will not be O(n) if there are empty cells scattered inside our array
7. Determine the median age of students
   - O(n log n) – sort and then determine the median age
   - Note: Doable in O(n) if you know the correct algorithm (tutorial 1)

CS1020 Knowledge (3)

- If we use a **sorted array** of size **n…**, e.g.
  - A = {4, 5, 6, 7, 15, 23, 50, 71}
- Required operations:
  1. Search whether there is a student with a certain age?
     - O(log n) – we can use **binary search**
  2. Insert a new student (that is, insert his/her age)
     - O(n) – after finding the correct insertion point, we still have to make gap to maintain the sorted order
  3. Determine the youngest and oldest student
     - O(1) – the smallest/largest is at the front/back of the sorted array (assuming we sort the integers in ascending order)

CS1020 Knowledge (4)

– A = {4, 5, 6, 7, 15, 23, 50, 71}

4. List down the ages of students in sorted order
   - O(n) – go through the sorted array once from left to right
5. Find a student slightly older than a certain age!
   - O(log n) – **binary search** the given age, then the answer is just one index on the right of the student with the certain age
6. Delete existing student (that is, remove his/her age)
   - O(n) – after removing a certain age, we have to ensure that there is no gap in our unsorted array, otherwise our binary search will be confused when it hits an empty cell in the middle of our array
7. Determine the median age of students
   - O(1) – we can immediately determine the median age as the array is already sorted

Comparison

- This is what we can do with just CS1020 knowledge

Operation	Unsorted Array	Sorted Array
1. Search(age)	**O(n)**	O(log n)
2. Insert(age)	O(1)	**O(n)**
3. FindMin()/FindMax()	**O(n)**	O(1)
4. List ages in sorted order	**O(n log n)**	O(n)
5. Find older(age)	**O(n)**	O(log n)
6. Delete(age)	**O(n)**	**O(n)**
7. Compute median age	**O(n log n)** or **O(n)**	O(1)

  - If n = 1000000 students, our queries are slow…

O(n) versus O(log n): A Perspective

- n = 8

- $\log_2 n = 3$

- n = 16

- $\log_2 n = 4$

- n = 32

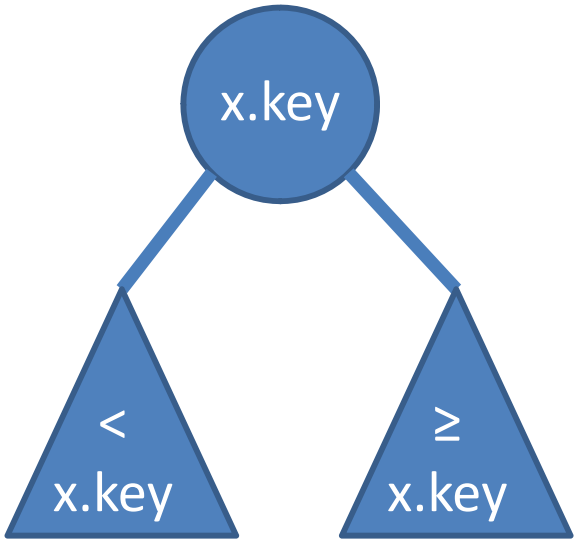
- $\log_2 n = 5$

A Versatile, Non-Linear Data Structure

BINARY SEARCH TREE (BST)

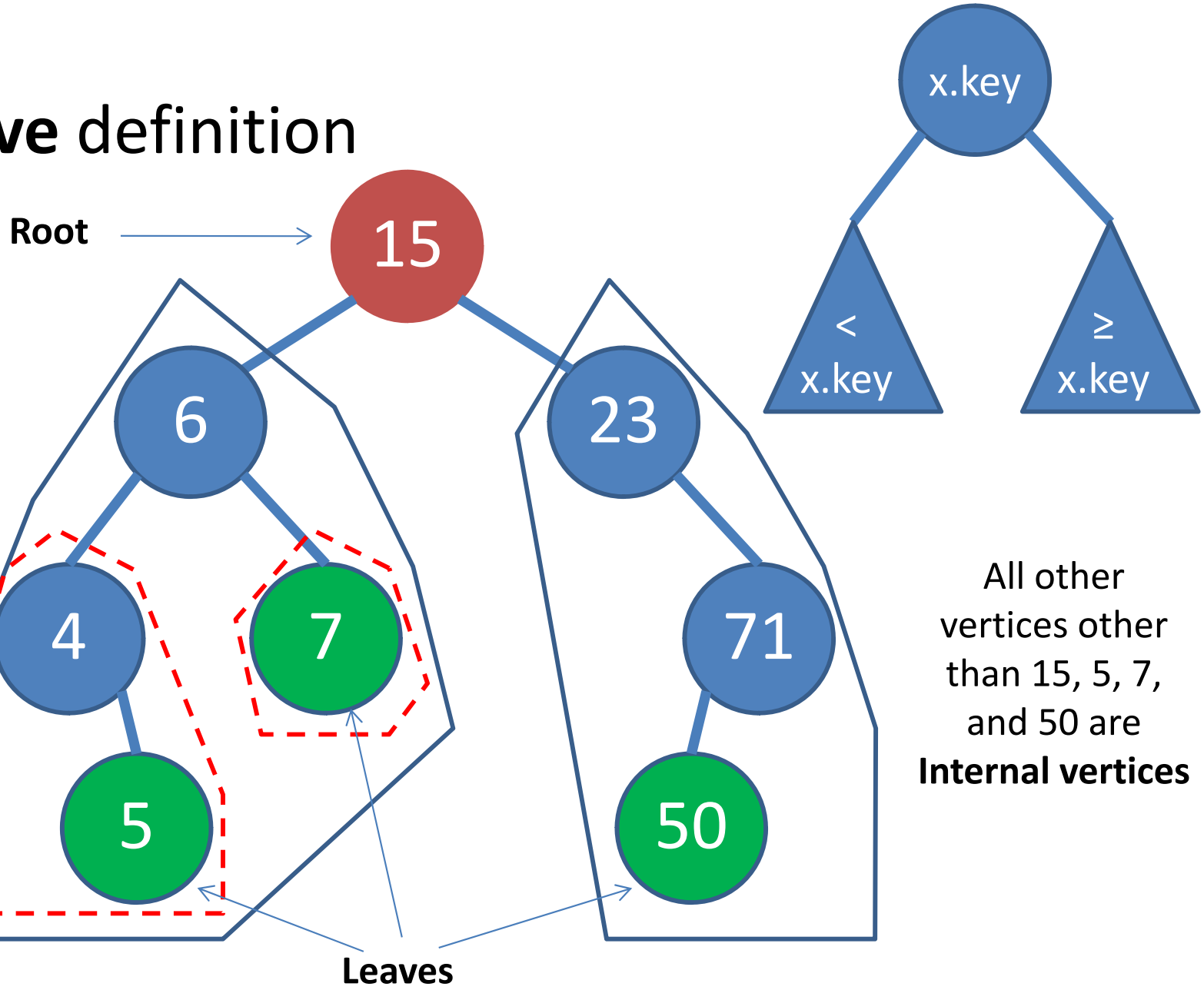
Binary Search Tree (BST) Vertex

- For every vertex x, we define:
  - x.left = the left child of x
  - x.right = the right child of x
  - x.parent = the parent of x
  - x.key (or x.value, x.data) = the value stored at x
- BST Property:
  - x.left.key < x.key ≤ x.right.key
  - For simplicity, we assume that the keys are unique so that we can change ≥ to >



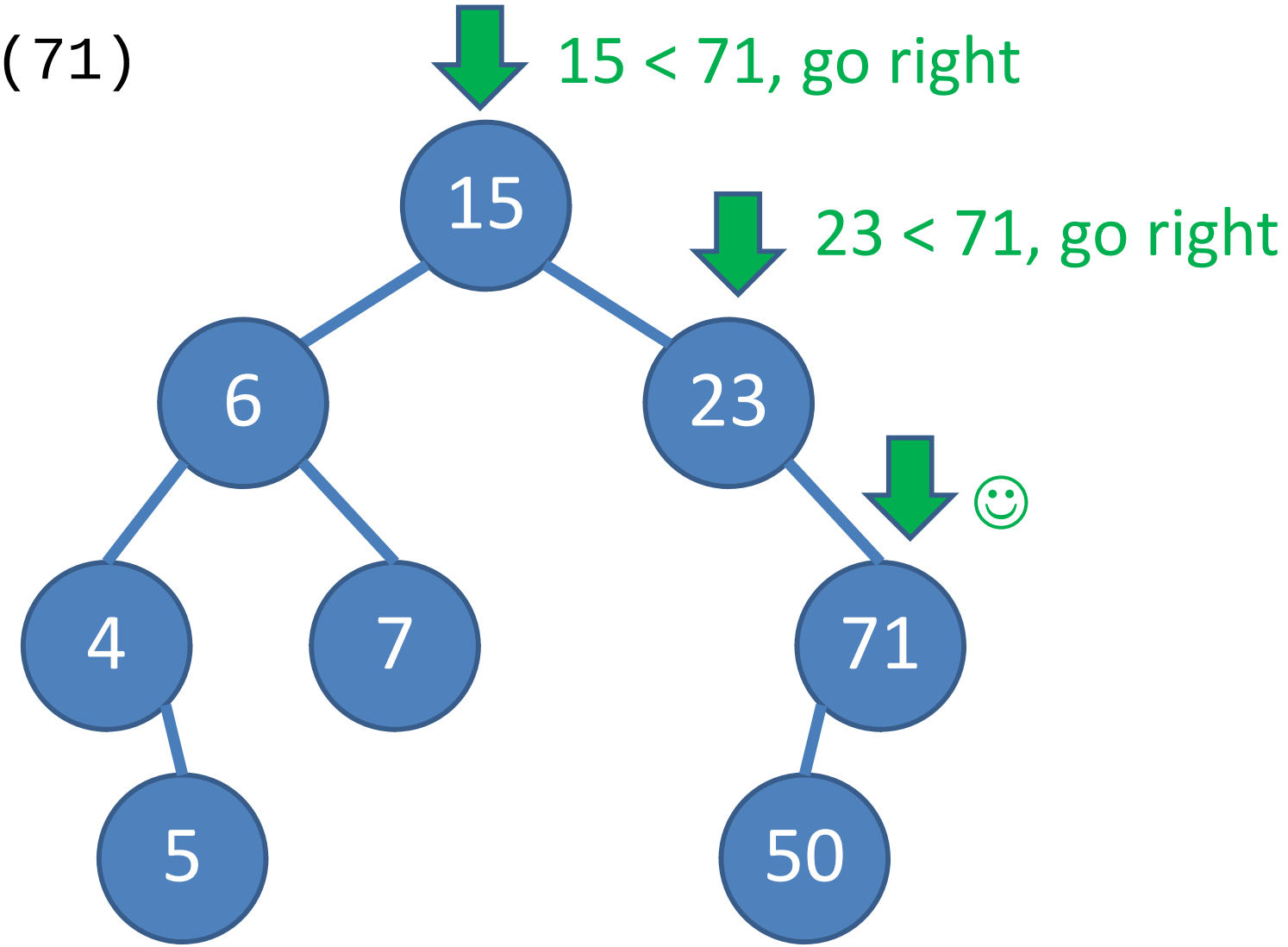
BST: An Example, Keys = Ages

- **Recursive** definition



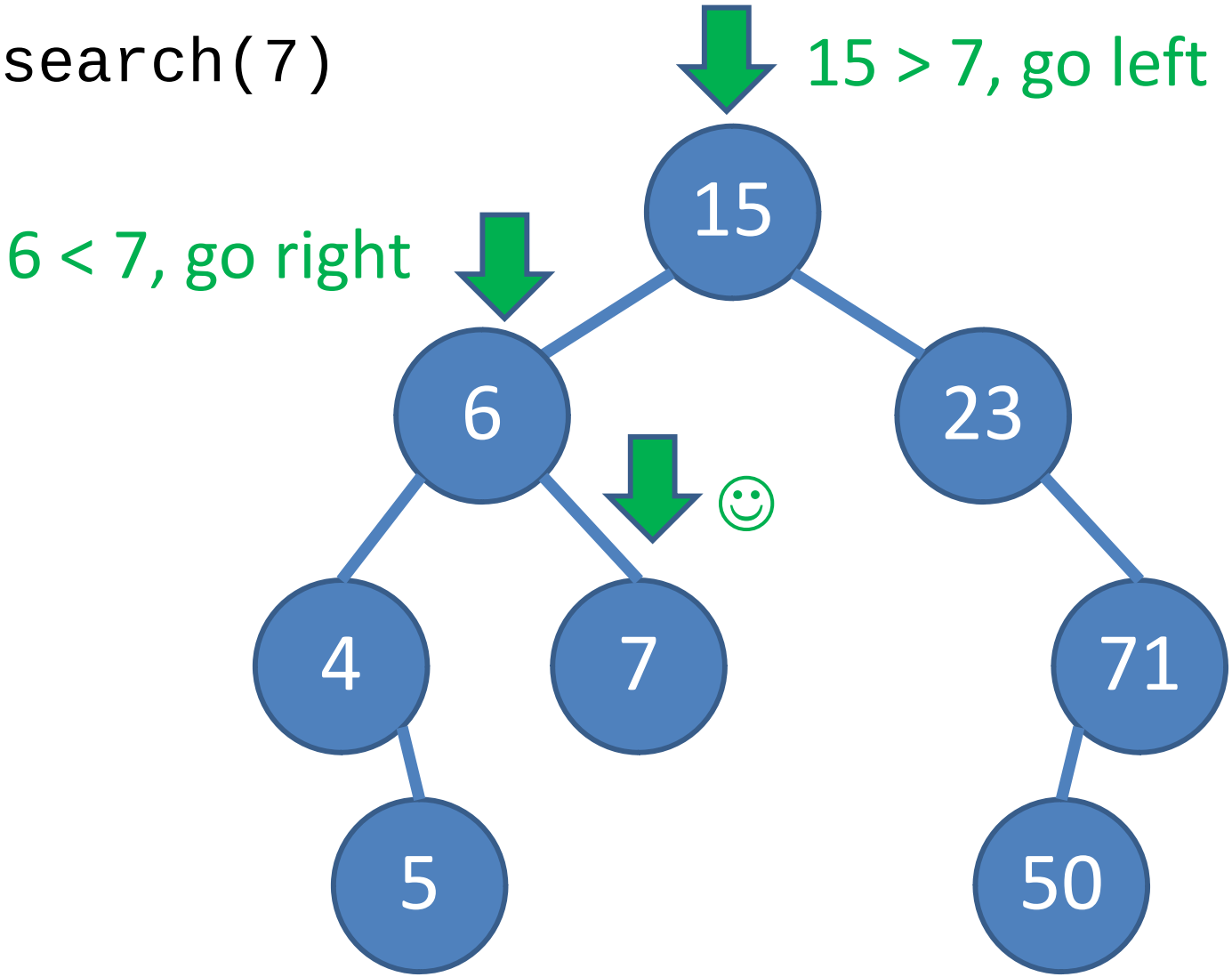
BST: Search Example (1)

- `search(71)`



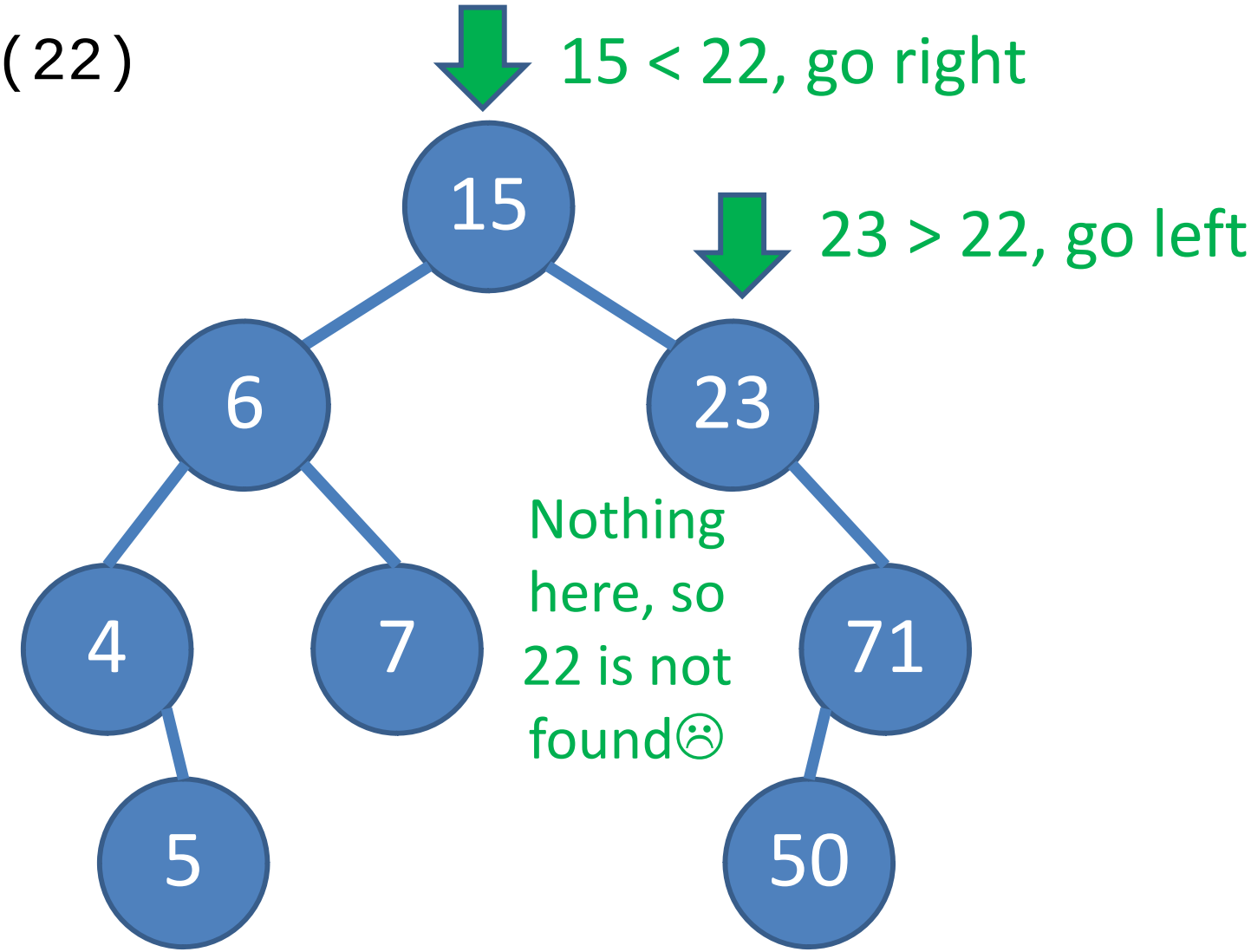
BST: Search Example (2)

- `search(7)`



BST: Search Example (3)

- `search(22)`

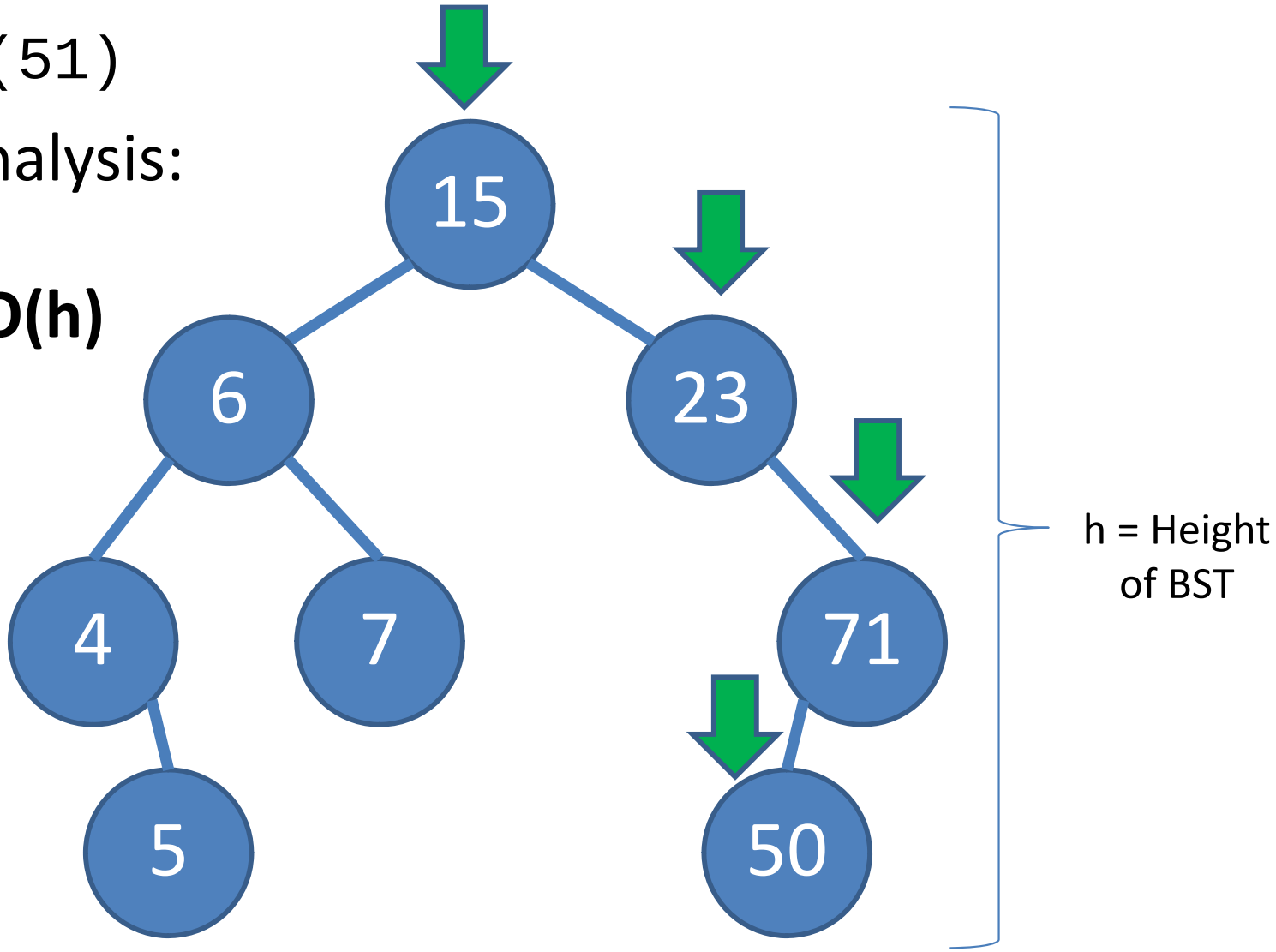


BST: Search (Recursive Pseudocode)

```
BSTVertex search(BSTVertex T, int v)
  if (T == null)
    return null
  else if (T.key == v)
    return T
  else if (T.key < v)
    return search(T.right, v)
  else // if (T.key > v)
    return search(T.left, v)
```

BST: Search Example (4)

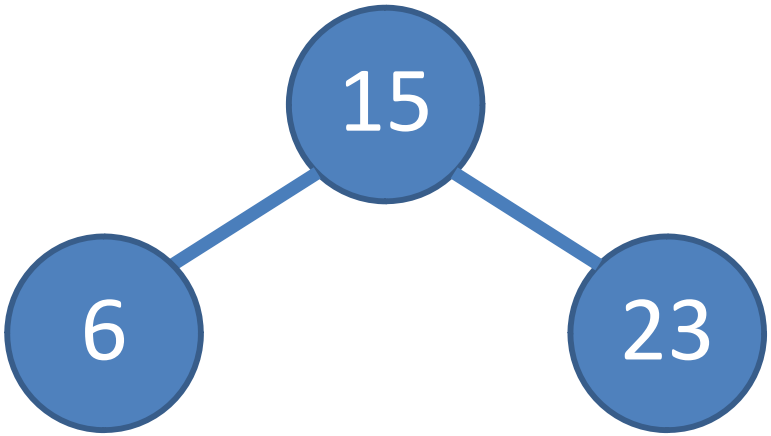
- `search(51)`
- Quick analysis: `search` runs in **O(h)**



51 is not found ☹

BST: Insertion Example (1)

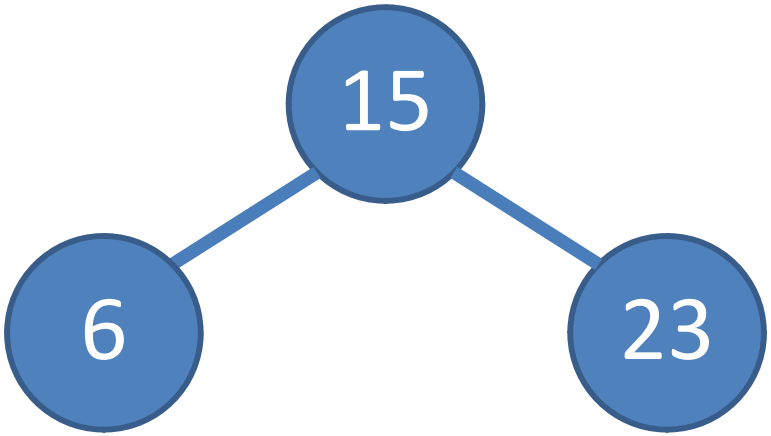
- `insert(15)`
- `insert(23)`
- `insert(6)`

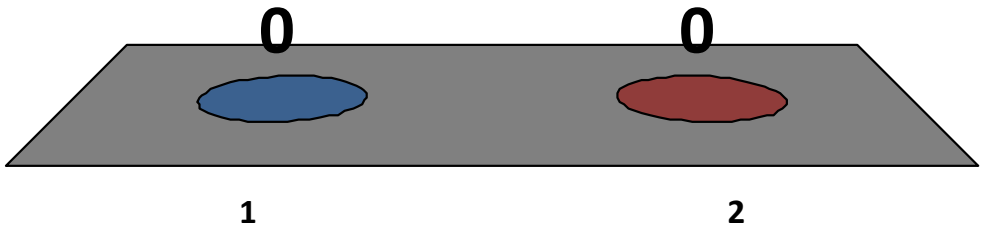


If we insert **15, 6, 23**, in that order, will we get the same BST?

1. Yes, same BST
2. No, different BST

Recall:
Inserting 15, 23, and then 6 give us:



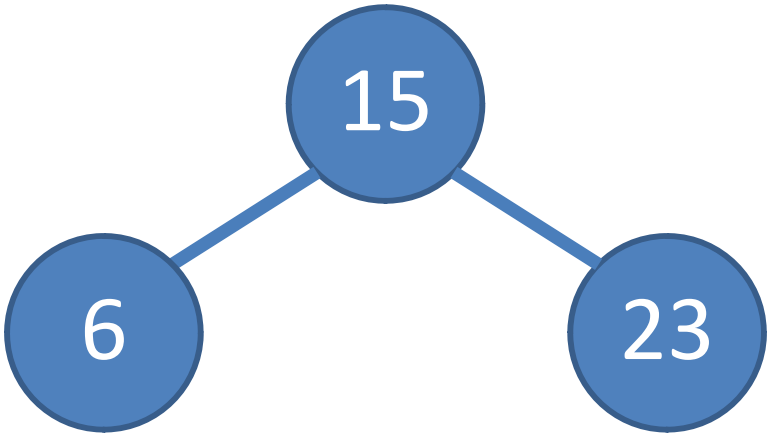


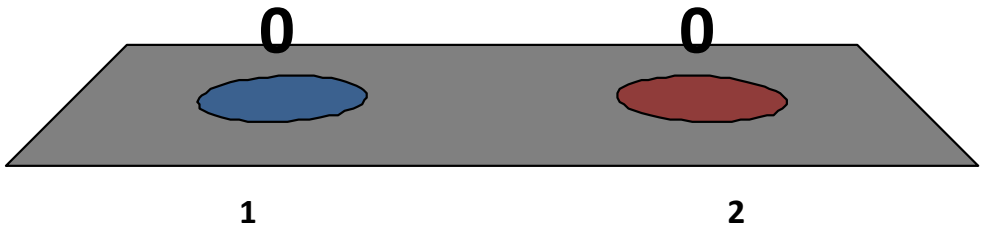
0 of 120

If we insert **6, 15, 23**, in that order, will we get the same BST?

1. Yes, same BST
2. No, different BST

Recall:
Inserting 15, 23, and then 6 give us:

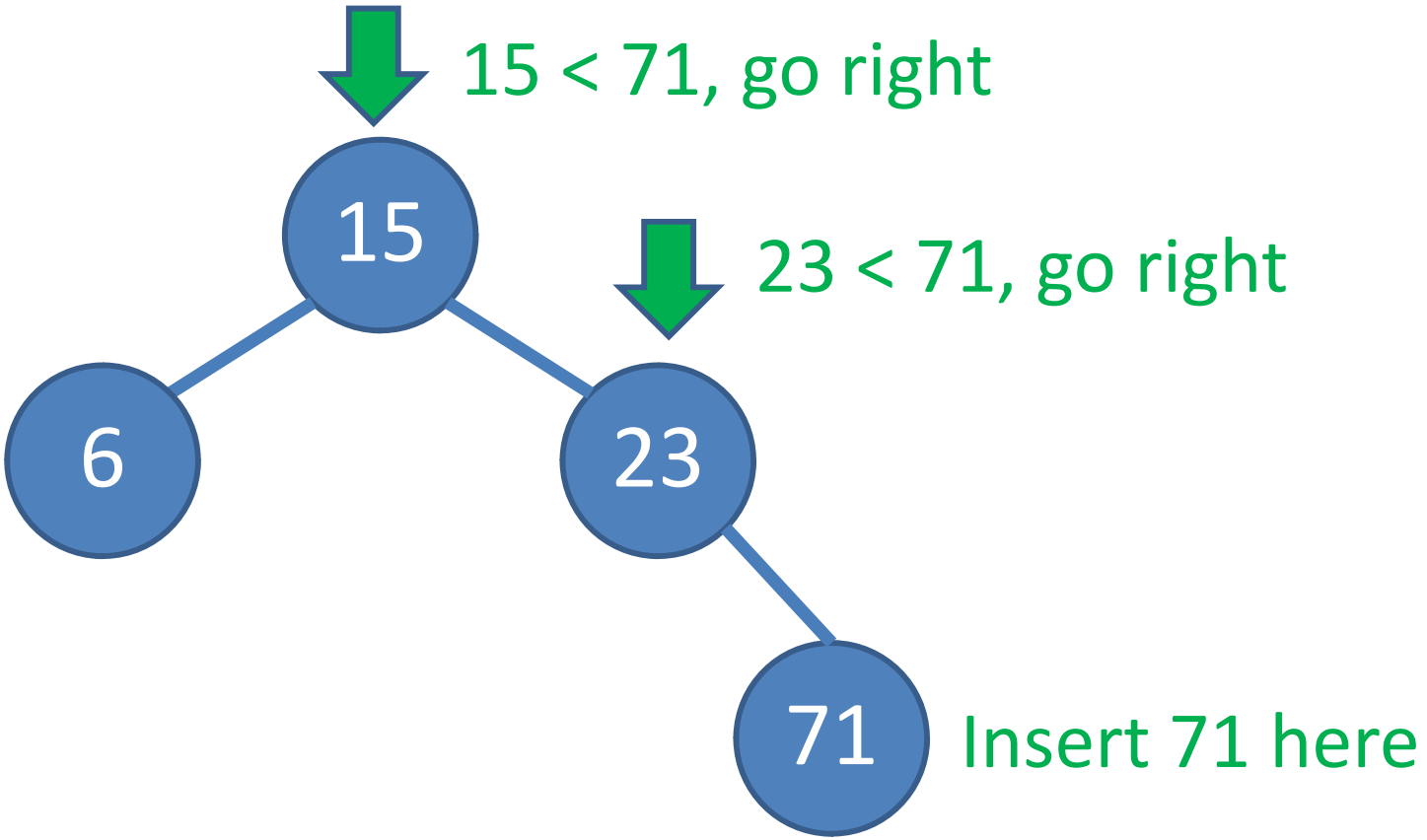




0 of 120

BST: Insertion Example (2)

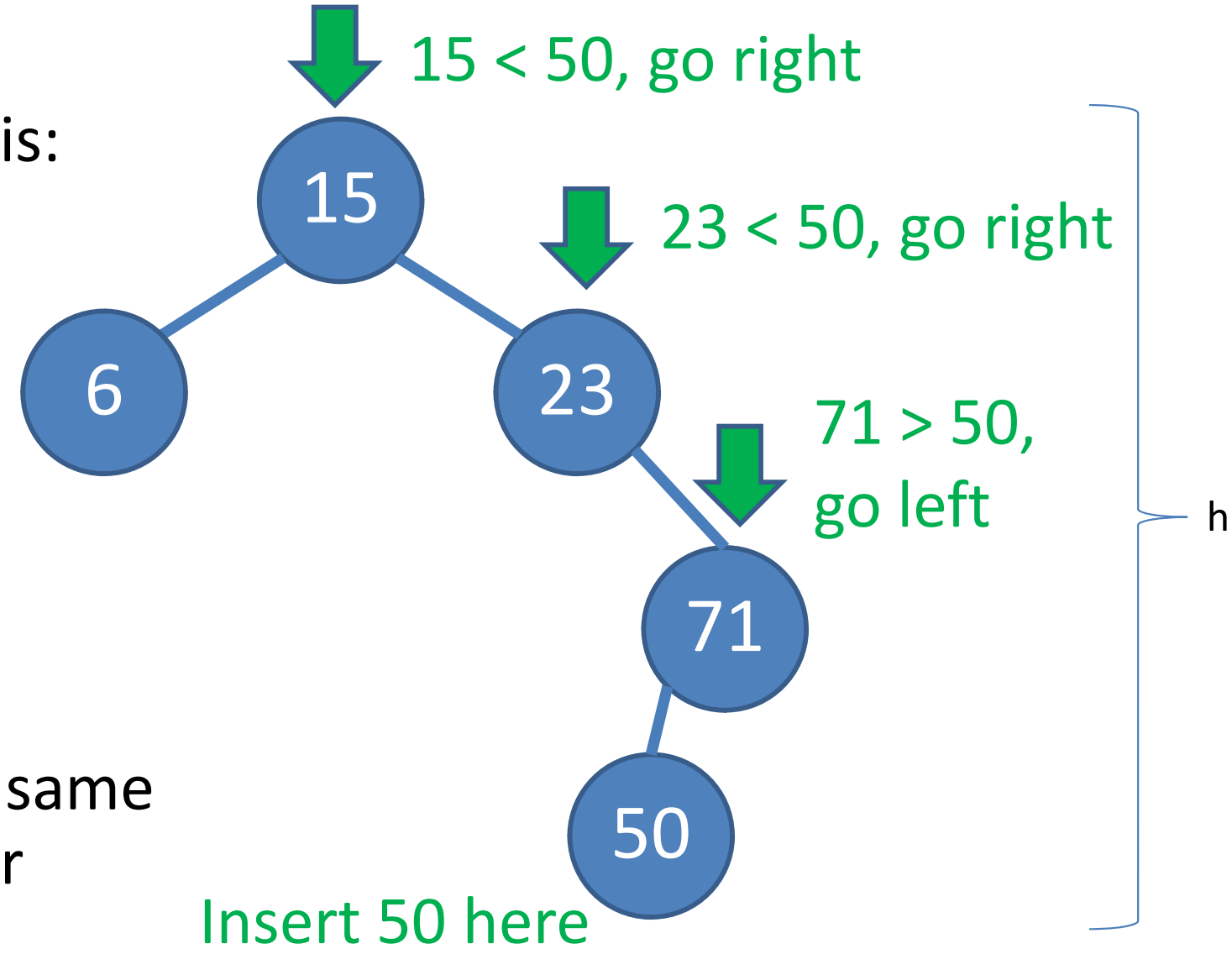
- `insert(71)`



BST: Insertion Example (3)

- `insert(50)`
- Quick analysis: `insert` also runs in **O(h)**

- After `insert(4)`, `insert(7)`, `insert(5)` we have the same BST as earlier



BST: Insert (Recursive Pseudocode)

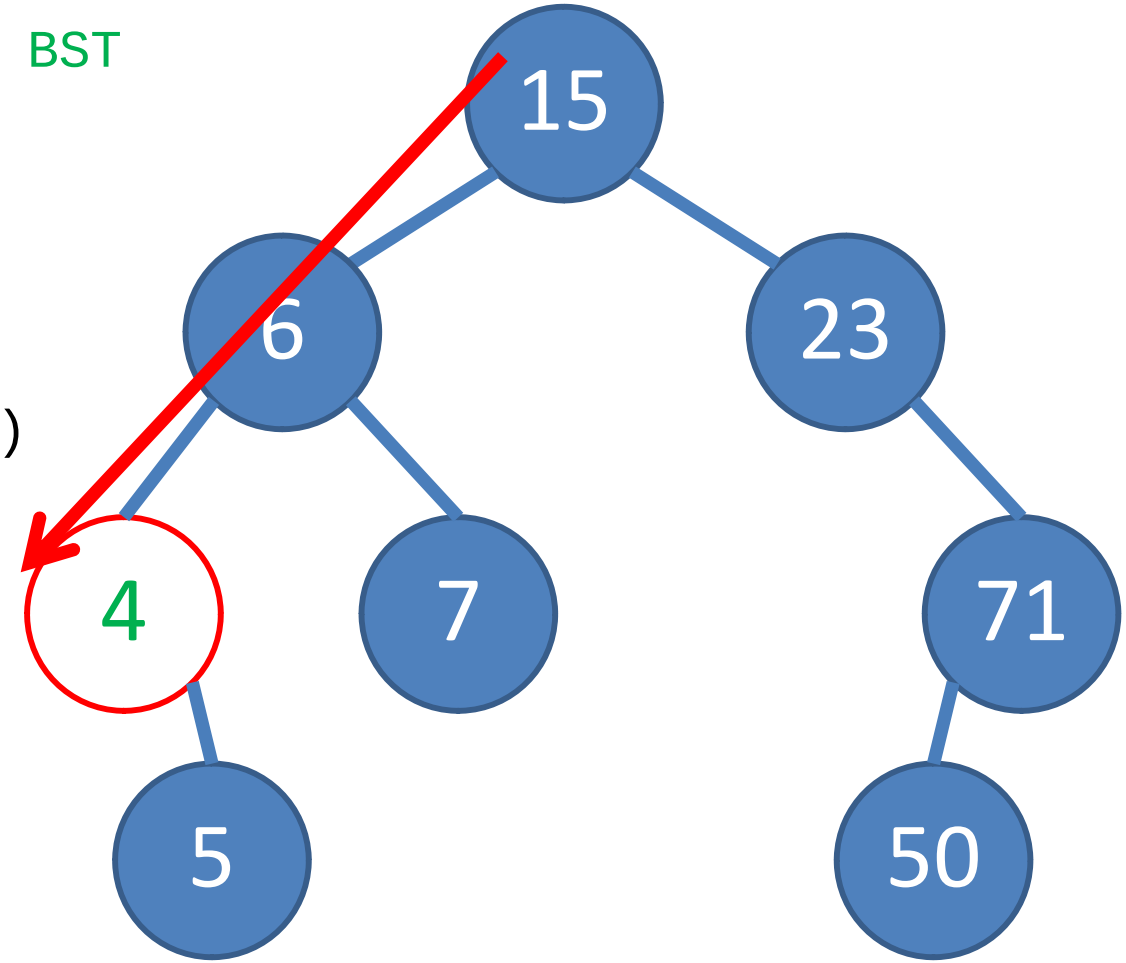
```
BSTVertex insert(BSTVertex T, int v)
  if (T == null) return a new BSTVertex(v)

  if (T.key < v)
    T.right = insert(T.right, v)
    T.right.parent = T
  else // if (T.key > v)
    T.left = insert(T.left, v)
    T.left.parent = T

  return T
```

BST: Find Minimum (Also Recursive)

```
int findMin(BSTVertex T)
  if (T == null) // empty BST
    return -1
  if (T.left == null)
    return T.key
  else
    return findMin(T.left)
```

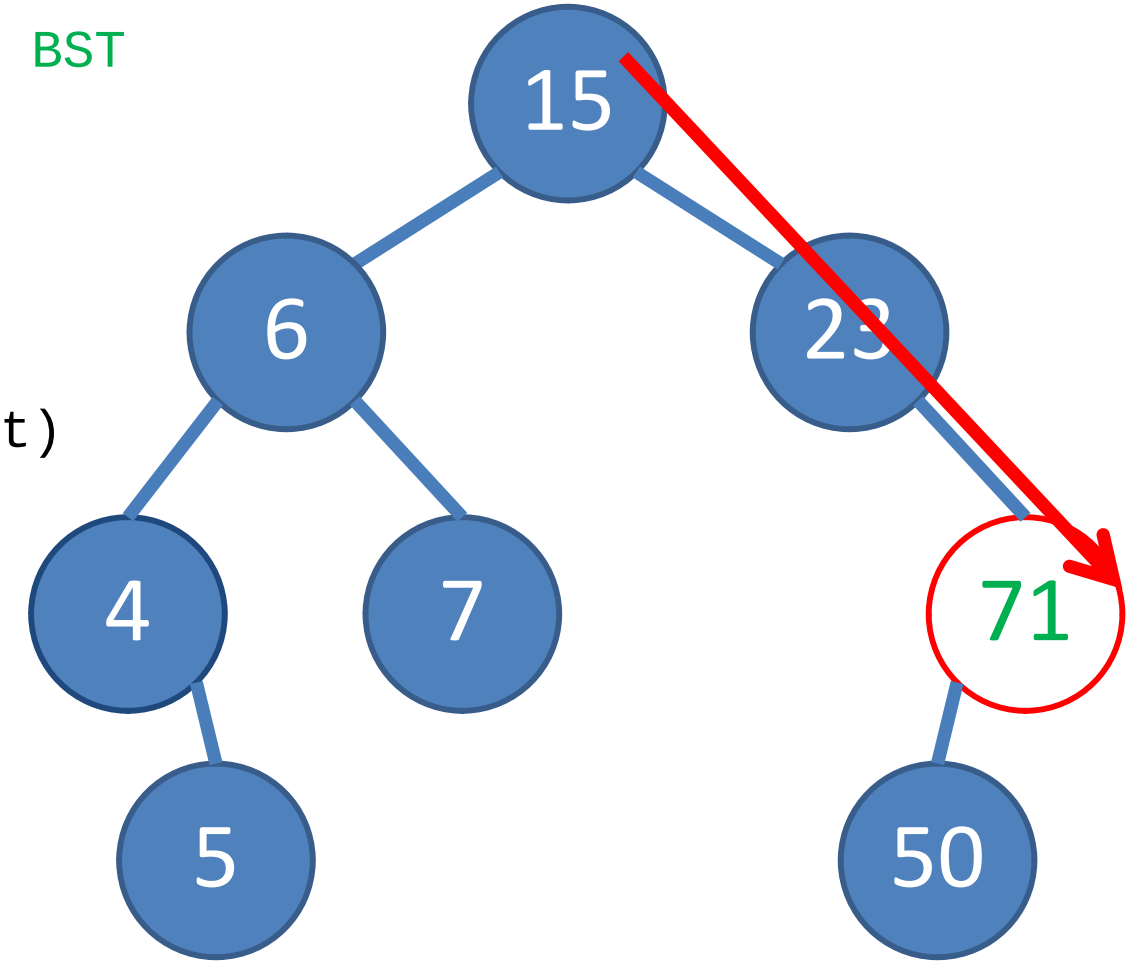


- Quick analysis: `findMin` also runs in **O(h)**

BST: Find Maximum (Similar)

```
int findMax(BSTVertex T)
  if (T == null) // empty BST
    return -1
  if (T.right == null)
    return T.key
  else
    return findMax(T.right)
```

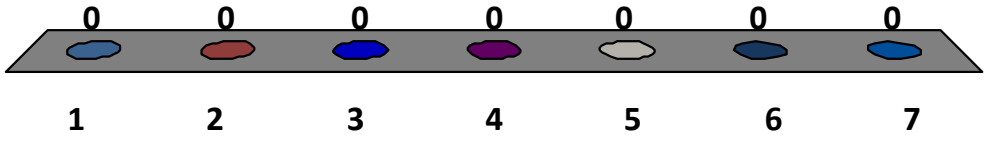
- Quick analysis: `findMax` also runs in **O(h)**



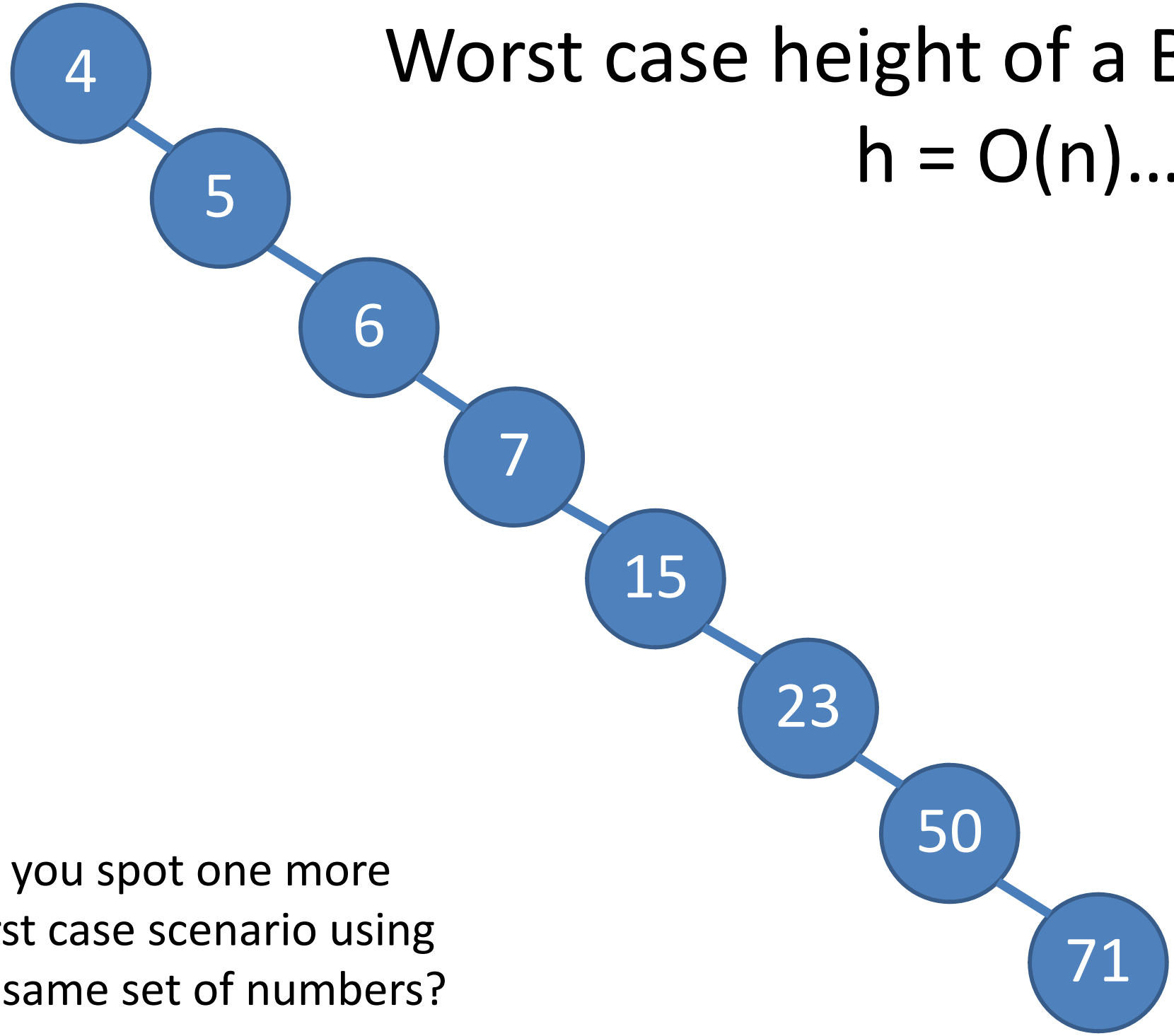
So, the time complexity of search/insert/findMin/ findMax all depends on '**h**', but how high can '**h**' be?

Hint: We divide the search range by half at each step?

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. $O(n^2)$
6. $O(n^3)$
7. $O(2^n)$



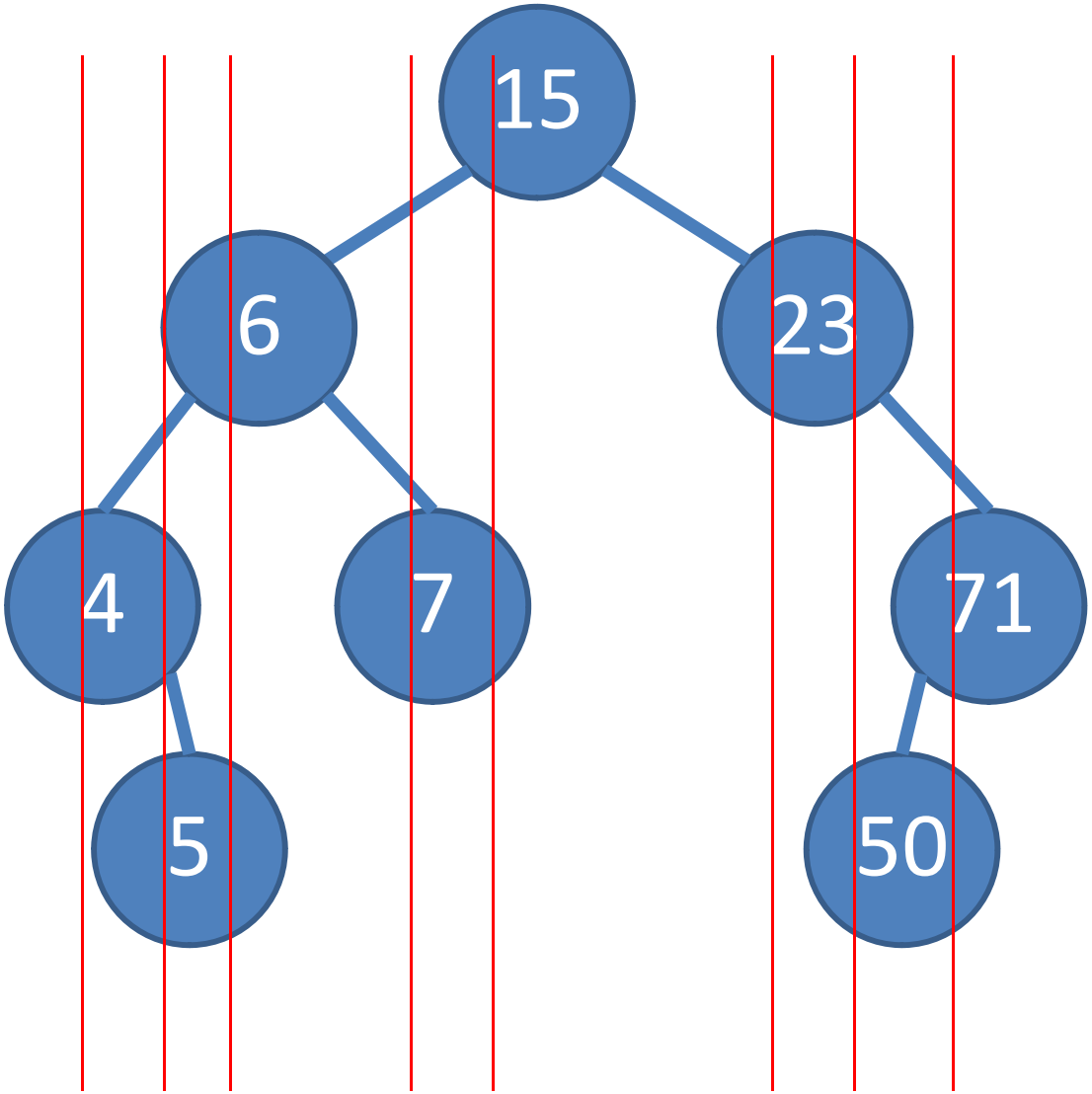
Worst case height of a BST
h = O(n)… ☹



Can you spot one more worst case scenario using the same set of numbers?

BST: Inorder Traversal (Recursive)

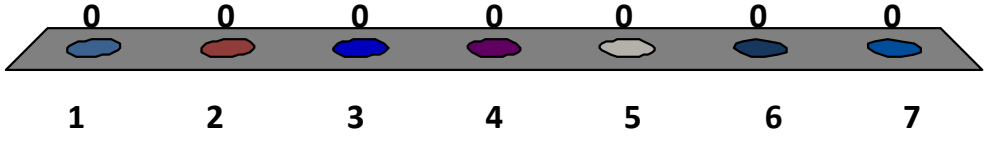
```
inorder(BSTVertex T)
   if (T == null) return
   inorder(T.left)
   // visit T
   inorder(T.right)
```



- Analysis?
  - See next slide

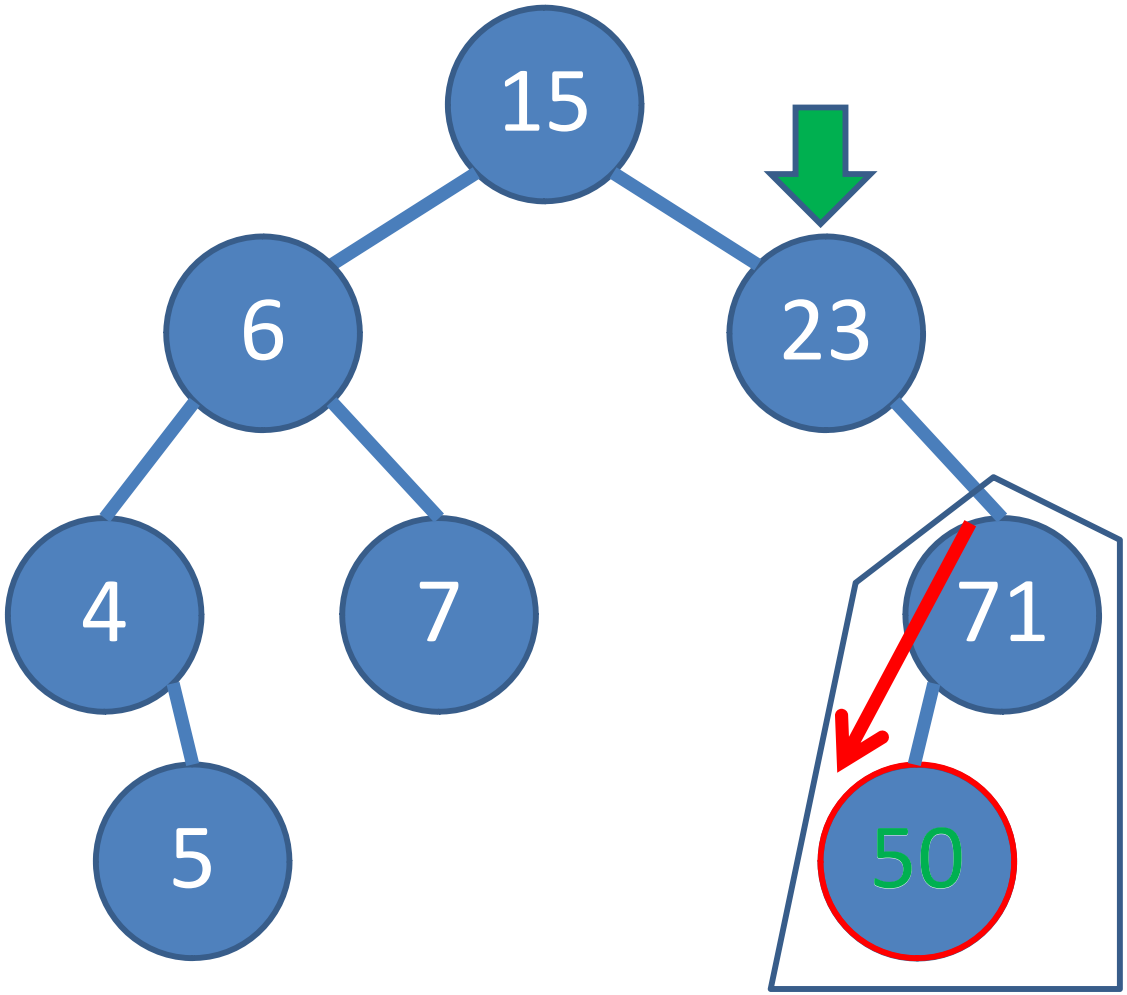
What is the worst-case running time of an **inorder traversal** in a standard BST?

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. $O(n^2)$
6. $O(n^3)$
7. $O(2^n)$



BST: Successor Example (1)

- `successor(23)`

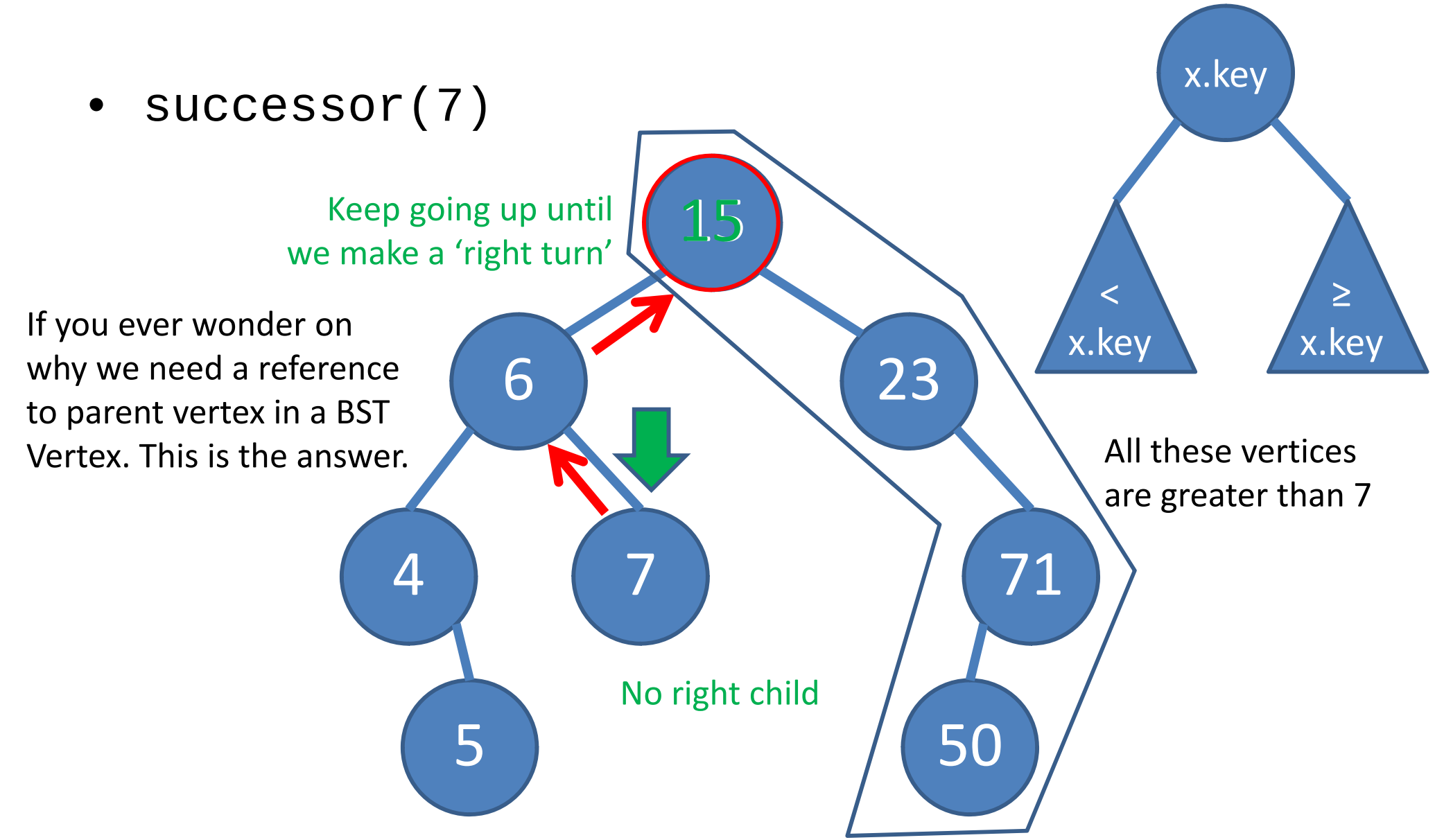


Definitely bigger than 23

Get the minimum in this subtree

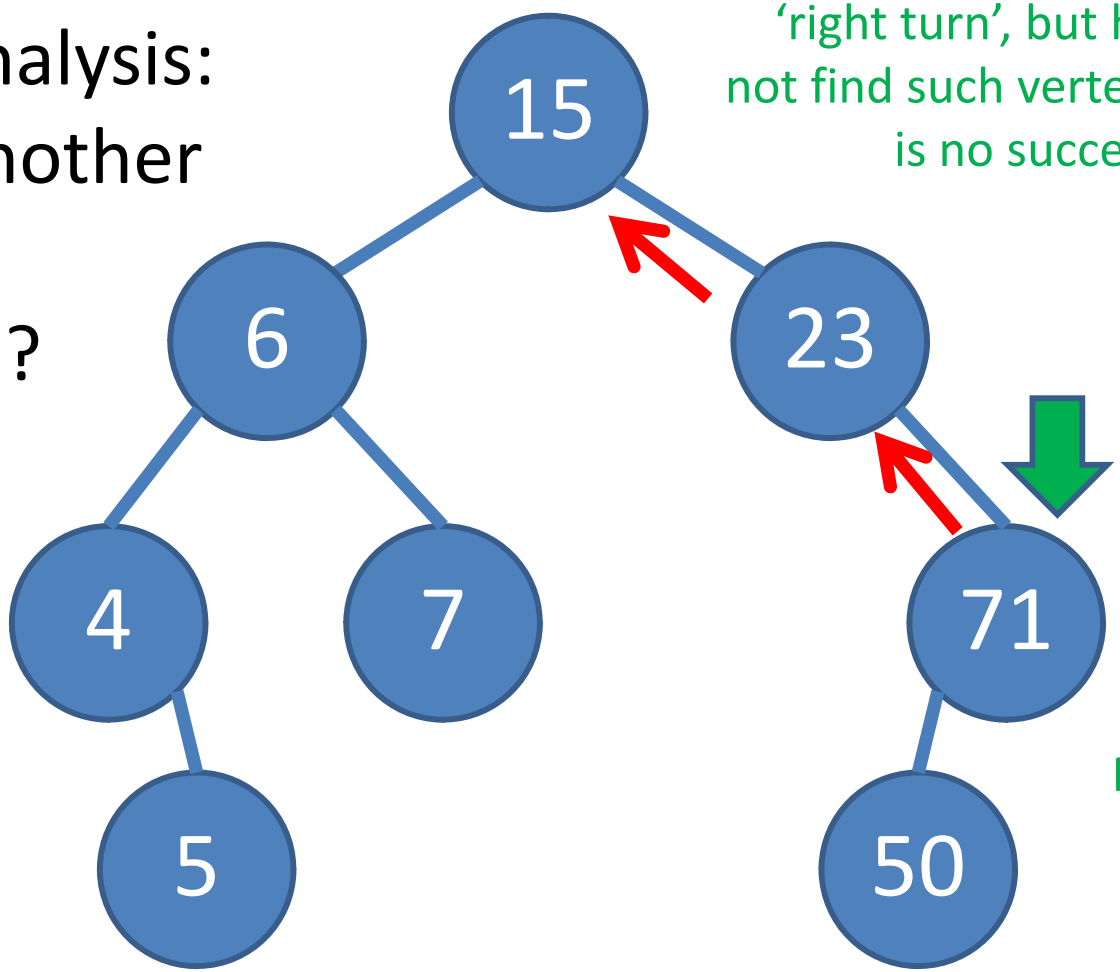
BST: Successor Example (2)

- `successor(7)`



BST: Successor Example (3)

- `successor(71)`
- Quick analysis:
  Is this another **O(h)** function?



Keep going up until we make a 'right turn', but here we do not find such vertex, so there is no successor for 71

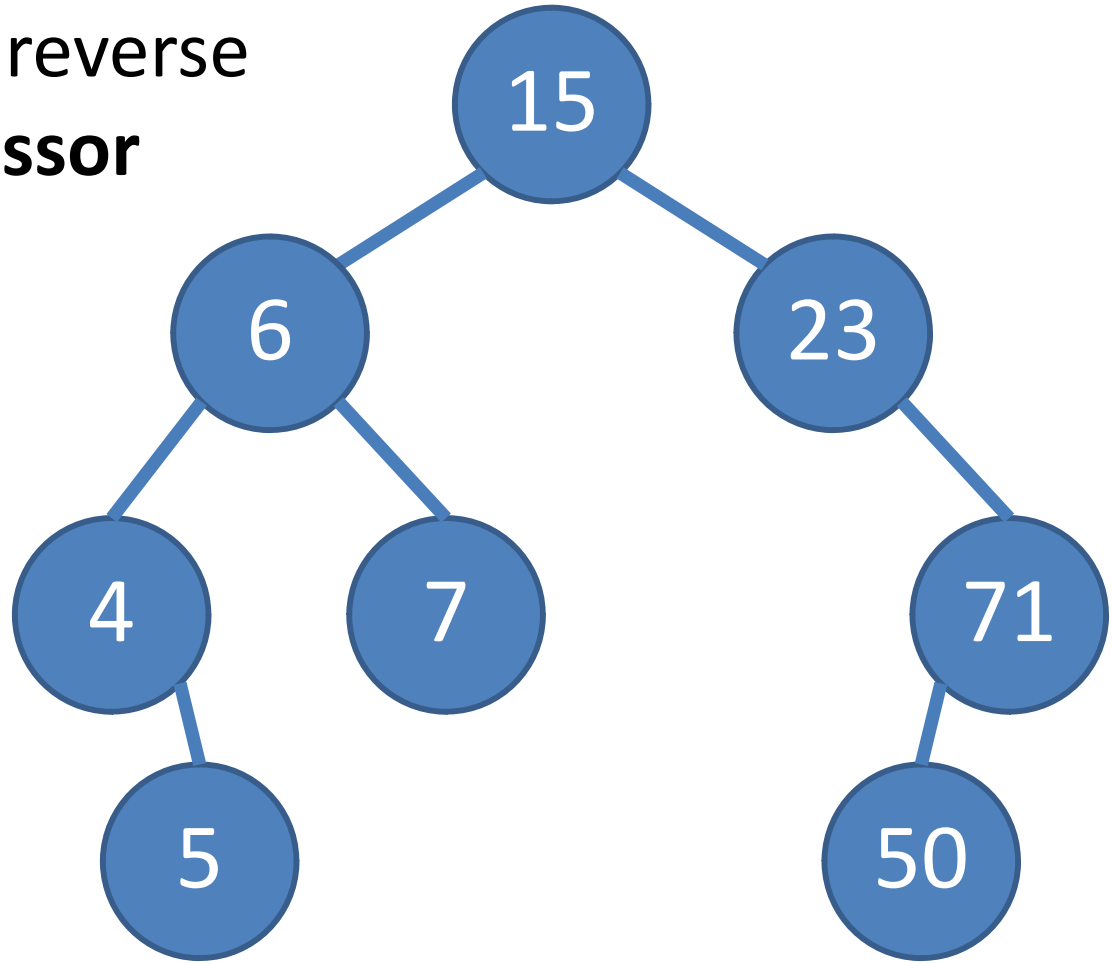
No right child

BST: Successor (Pseudocode)

```
int successor(BSTVertex T)
  if (T.right != null) // the easy case
    return findMin(T.right)
  else // the slightly harder case
    par = T.parent, cur = T
    while ((par != null) & (cur == par.right))
      cur = par
      par = cur.parent
    if (par == null) return -1 // special case
    else                return par.key
```

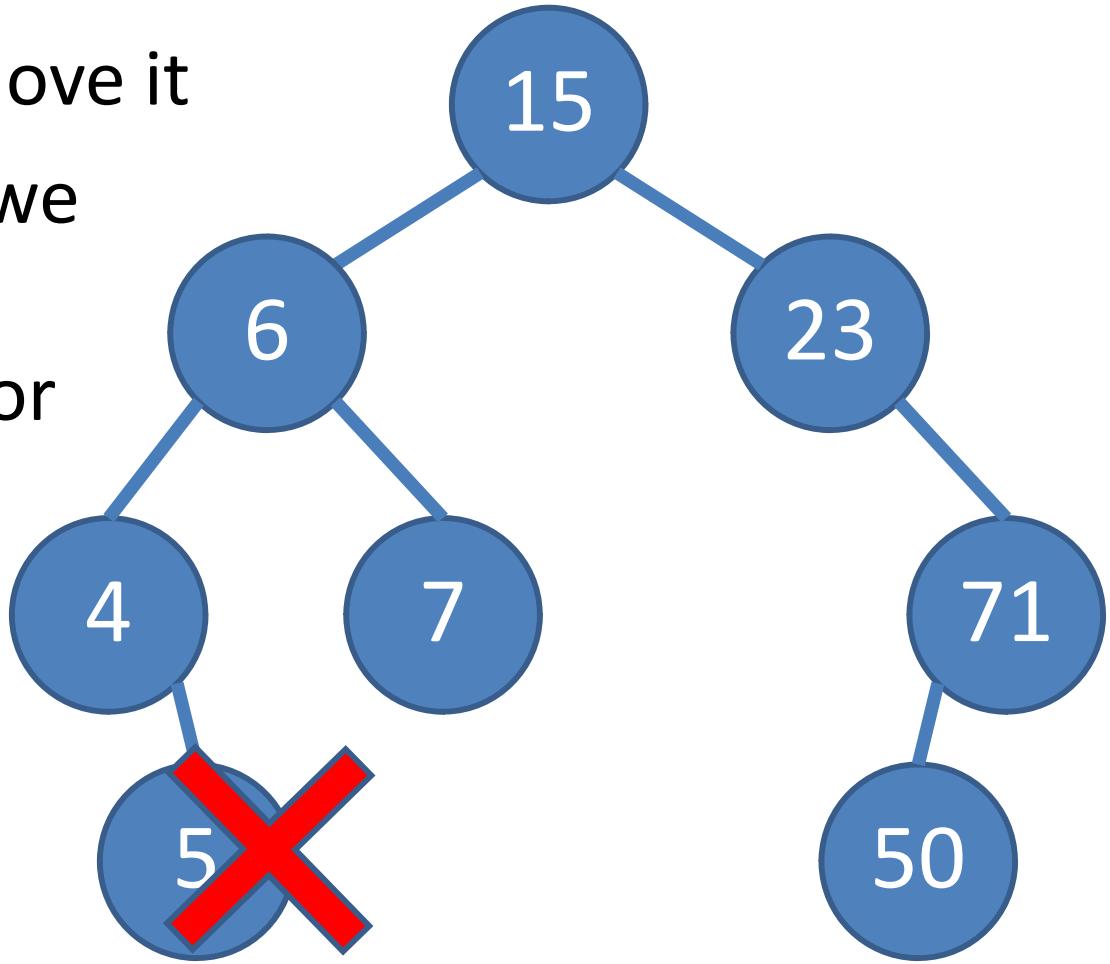
BST: Predecessor Example

- `predecessor(23), (7), (71)?`
- Just the reverse of **successor**



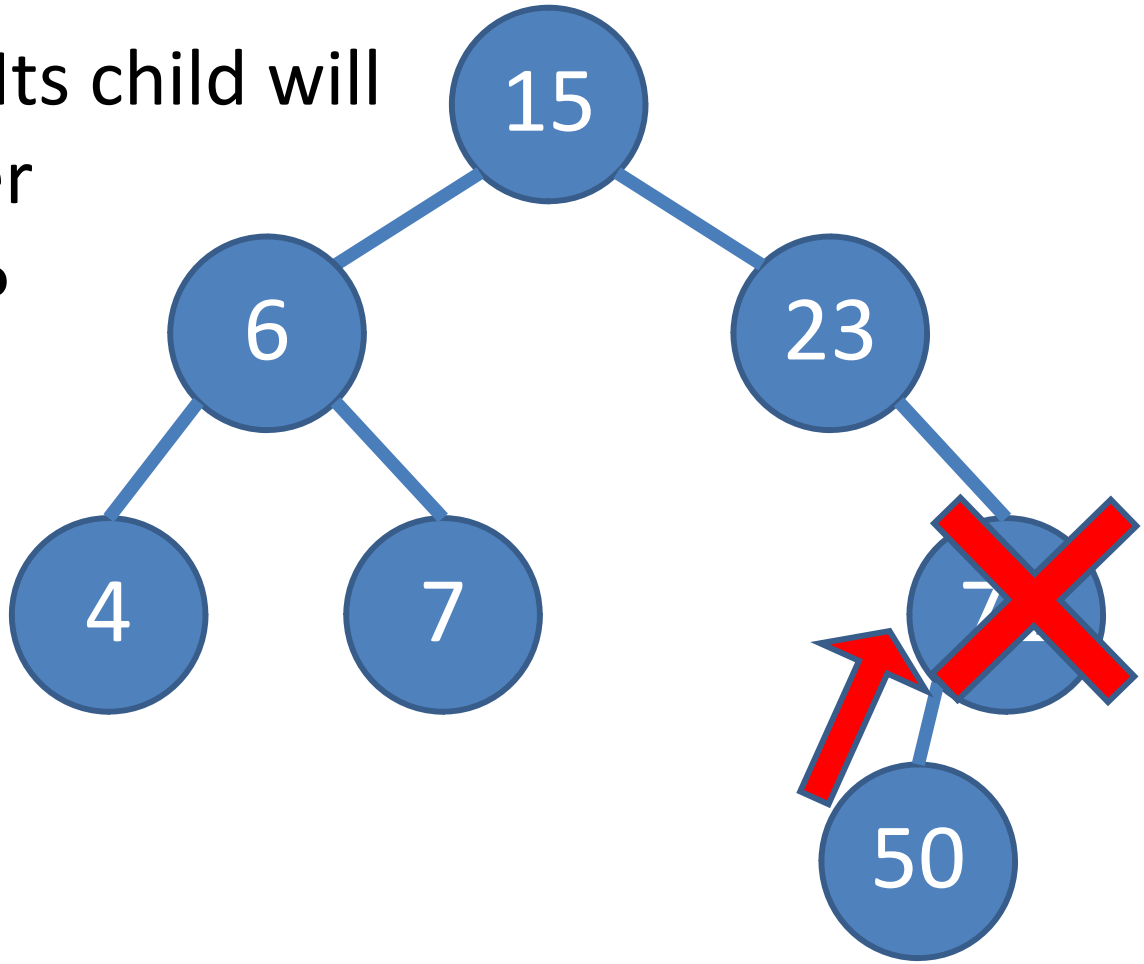
BST: Delete Example (1)

- `delete(5)` – a leaf
- Just remove it
- O(h) as we need to search for it first



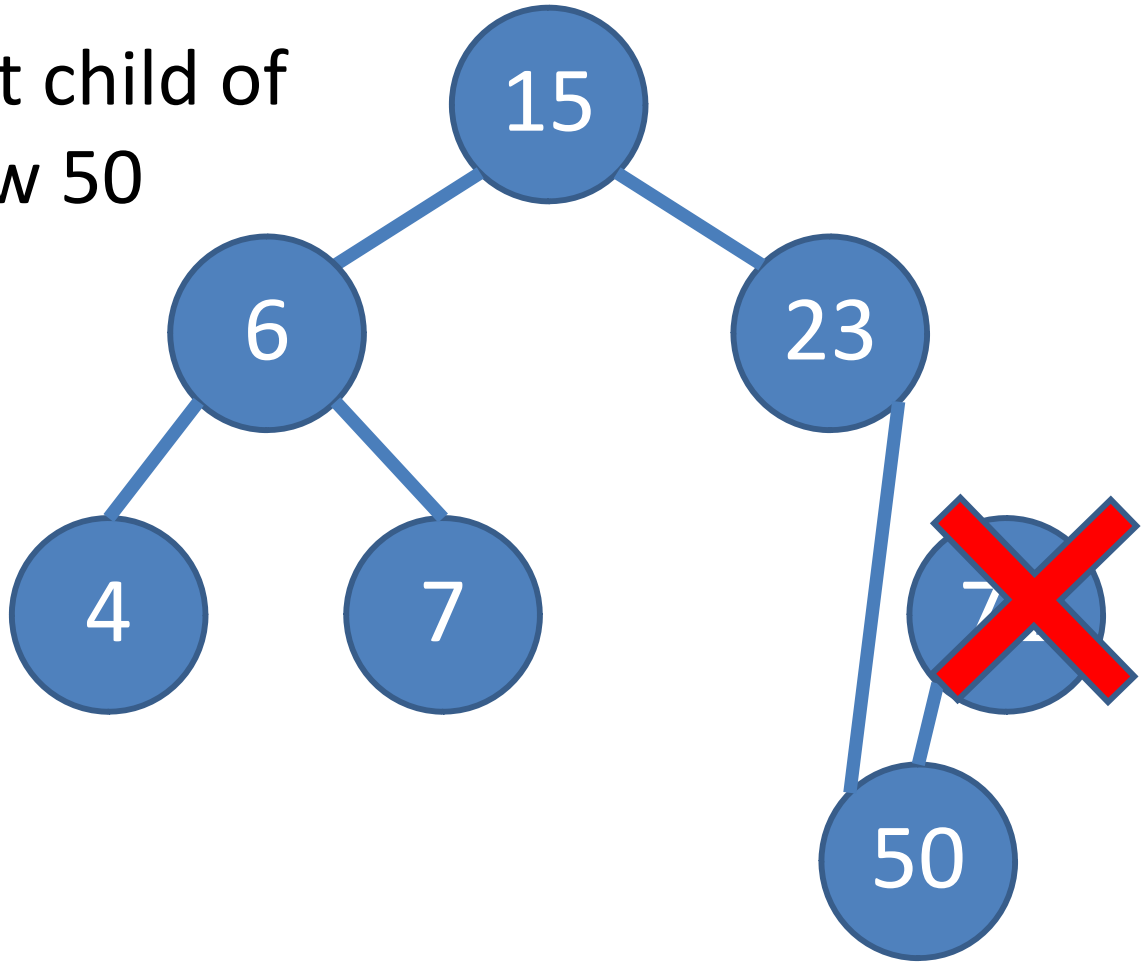
BST: Delete Example (2a)

- Now `delete(71)` – it has one left child
- Simple: Its child will take over
- Q: Why?



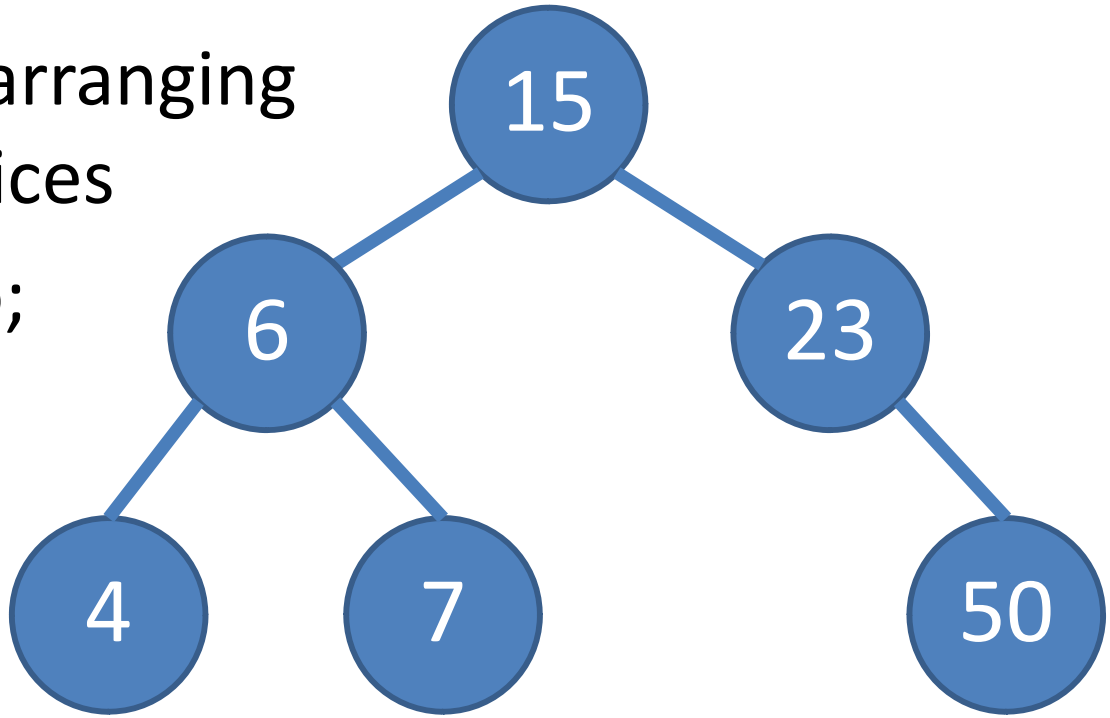
BST: Delete Example (2b)

- Now `delete(71)` – it has one left child
- The right child of 23 is now 50



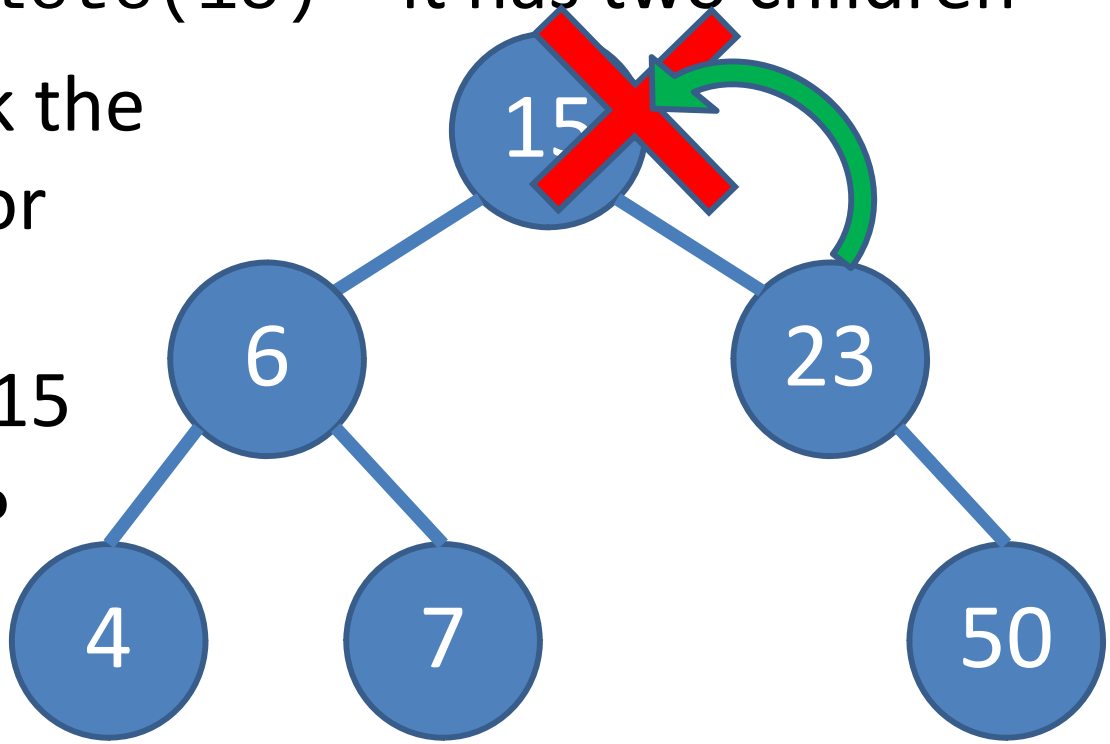
BST: Delete Example (2b)

- Now `delete(71)` – it has one left child
- After rearranging the vertices
- O(h) too; search and some O(1) pointer manipulation



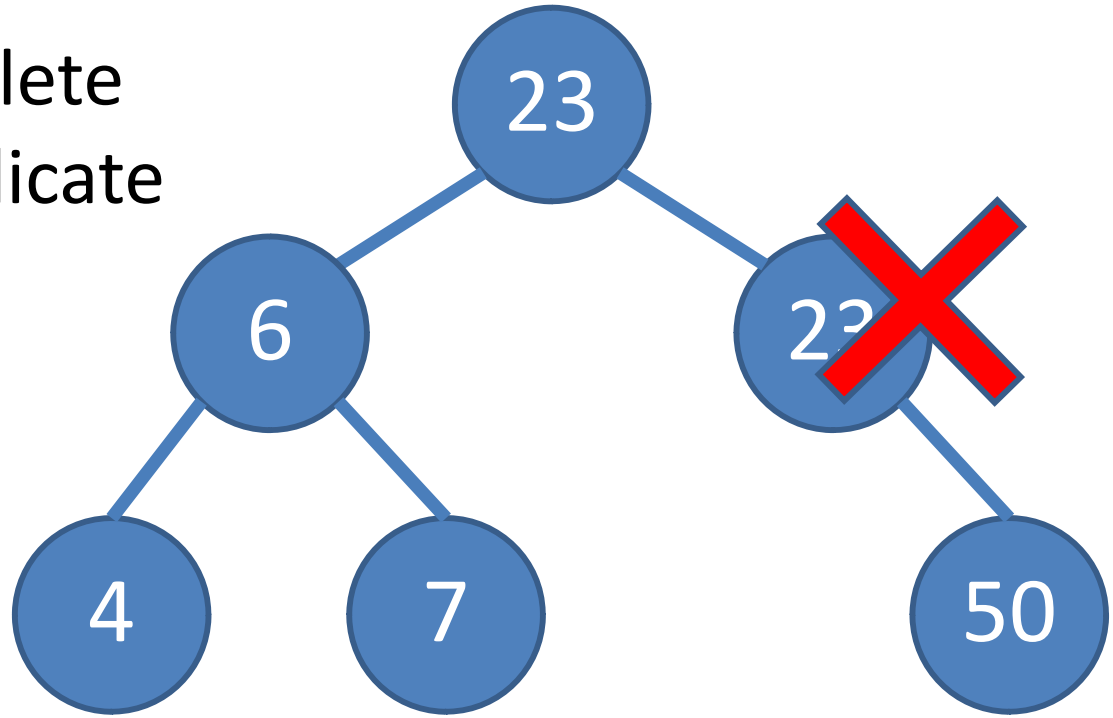
BST: Delete Example (3a)

- Now `delete(15)` – it has two children
- First, ask the successor of 15 to replace 15
- Q: Why?



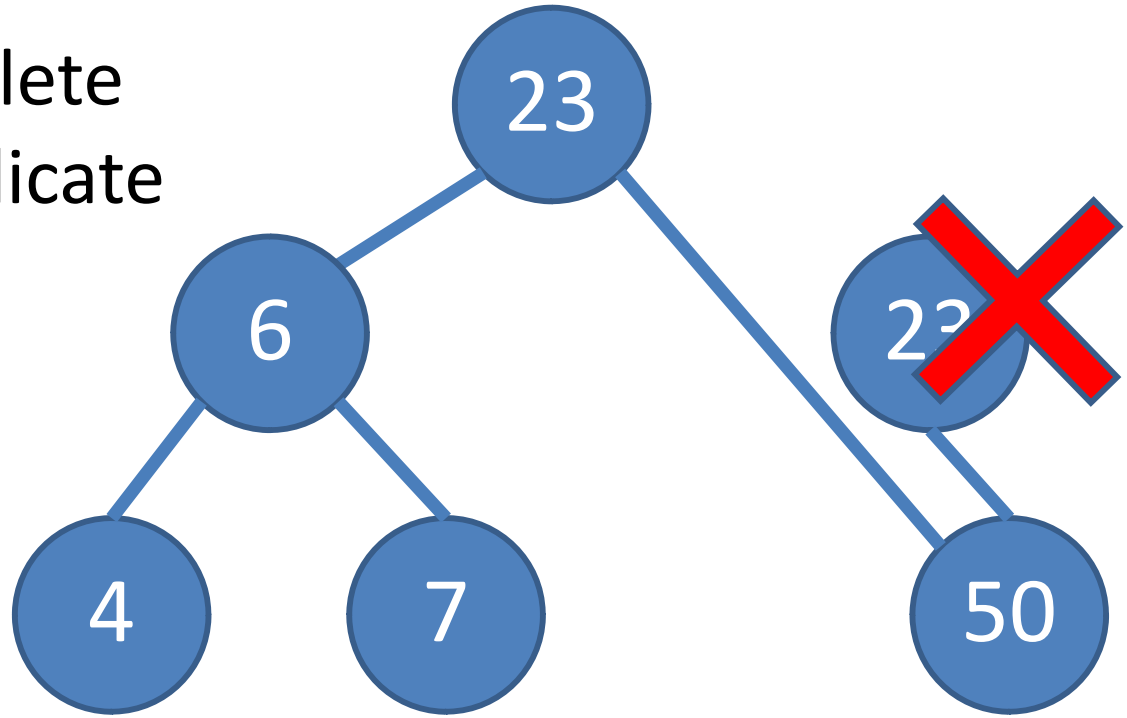
BST: Delete Example (3b)

- Now `delete(15)` – it has two children
- Then delete the duplicate
- O(h) to search the vertex, its successor, and delete the duplicate



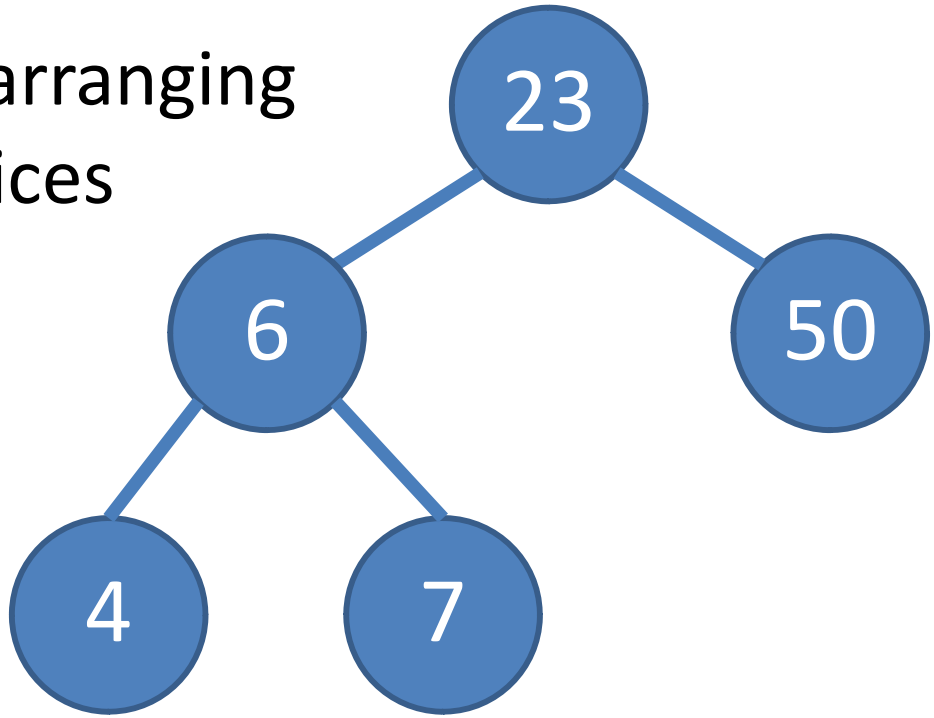
BST: Delete Example (3c)

- Now `delete(15)` – it has two children
- Then delete the duplicate



BST: Delete Example (3d)

- Now `delete(15)` – it has two children
- After rearranging the vertices



Why successor of x can be used for deletion of a BST vertex x with 2 children?

- Claim: Successor of **x** has at most 1 child!
- Proof:
  - Vertex x has two children
  - Therefore, vertex x must have **a right child**
  - Successor of x must then be the minimum of the right subtree
  - A minimum element of a BST has no left child!!
  - *So, successor of x has at most 1 child!* ☺

BST: Delete (Recursive Pseudocode)

```
BSTVertex delete(BSTVertex T, int v)
  if (T == null) return T
  if (T.key == v)
    // the three cases outlined earlier
    // Expanded in the next slide
  else if (T.key < v)
    T.right = delete(T.right, v)
  else // search to the left
    T.left = delete(T.left, v)
  return T
```

if (T.key == v)

```
if (T.left == null and T.right == null) // leaf
  // set T to null
else if (T.left == null and T.right != null)
  // one right child
  // bypass T
else if (T.left != null and T.right == null)
  // one left child
  // bypass T
else // has two children, find successor
  // replace this key with the successor's key
  T.key = successor(V)
  // then delete the old successor
  T.right = delete(T.right, T.key)
```

BST: Delete (Summary)

- Delete a BST vertex v, Three cases:
  - No children:
    - Just remove the corresponding BST vertex v
  - 1 child (either left or right):
    - Connect v.left (or v.right) to v.parent and vice versa
    - Then remove v
  - 2 children:
    - Find x = successor(v)
    - Replace v.key with x.key
    - Then delete x in v.right
- Running time: O(h)

Summary of this Lecture

- Now, after we spend one lecture learning BST…

Operation	Unsorted Array	Sorted Array	BST
1. Search(age)	**O(n)**	O(log n)	O(h) – search
2. Insert(age)	O(1)	**O(n)**	O(h) – insert
3. FindMin()/FindMax()	**O(n)**	O(1)	O(h) – findMin()/Max()
4. List ages in sorted order	**O(n log n)**	O(n)	O(n) – inorder traversal
5. Find older(age)	**O(n)**	O(log n)	O(h) – successor
6. Delete(age)	**O(n)**	**O(n)**	O(h) – delete
7. Compute median age	**O(n log n)** or **O(n)**	O(1)	O(h) – with a trick

  - It is all now depends on **'h'**… → Next lecture ☺

The Baby Names Problem (PS1)

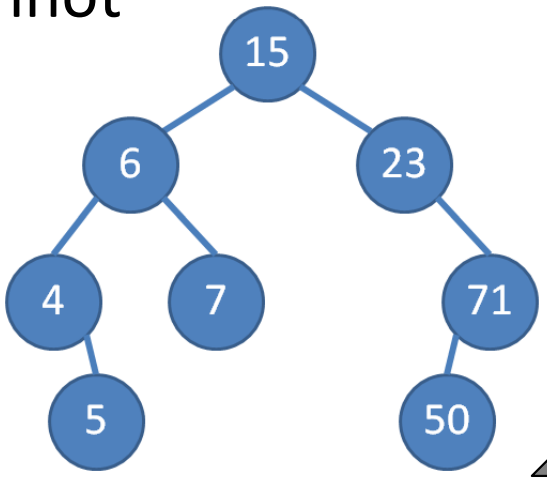
- *Always* encountered by every parents with new baby
  - Given a list of male and female baby names suggestions *(from your parents, in-laws, friends, yourself, Internet, **etc**)*
  - Answer some queries (next slide)
- Many, if not all of you, will probably encounter this 'real life problem' too if you have wife/husband and bab(ies) in a few years time

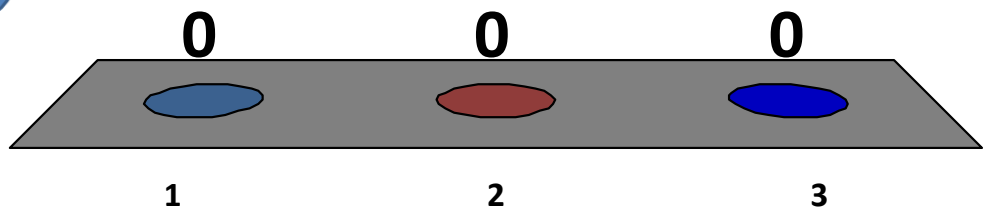
Typical Queries

- Note: The keys are strings
- Easy: How many names start with a certain letter?
- Medium: How many names start with a certain prefix?
  - A prefix of a string $T = T_0T_1…T_{n-1}$ with length n *is string* $P = T_0T_1…T_m$ *where* $m < n$.
- CS2010R: How many names has a certain substring?
  - A substring of a string $T = T_0T_1…T_{n-1}$ with length n *is string* $S = T_iT_{i+1}…T_{j-1}T_j$ *where* $0 \le i \le j < n$.
- What do you need to answer those queries?
  - An efficient data structure!

Teaser: Quiz 1 last year: Now that you have seen “inorder”, “findMin”, and “successor”, can we implement “inorder” as “findMin” one time, and then repeatedly call “successor” until we run out of successor?

1. Yes, we can, and the time complexity will be the same, O(___)
2. Yes, we can, but the time complexity increases to O(____)
3. No, we cannot





0 of 120