CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DATA STRUCTURES – PART II

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe binary search trees (BSTs) and heaps
- Explain the algorithms for *insert*, *search* and *delete* operations in BSTs, and derive their time complexities
- Explain the algorithms of the *delete-min* and *insert* operation in heaps, and prove their logarithmic time complexity
- Step through a comprehensive, non-trivial data-structure design process (for Union-Find), along with progressive enhancements
- Distinguish yet relate between *conceptual* and *physical* implementations

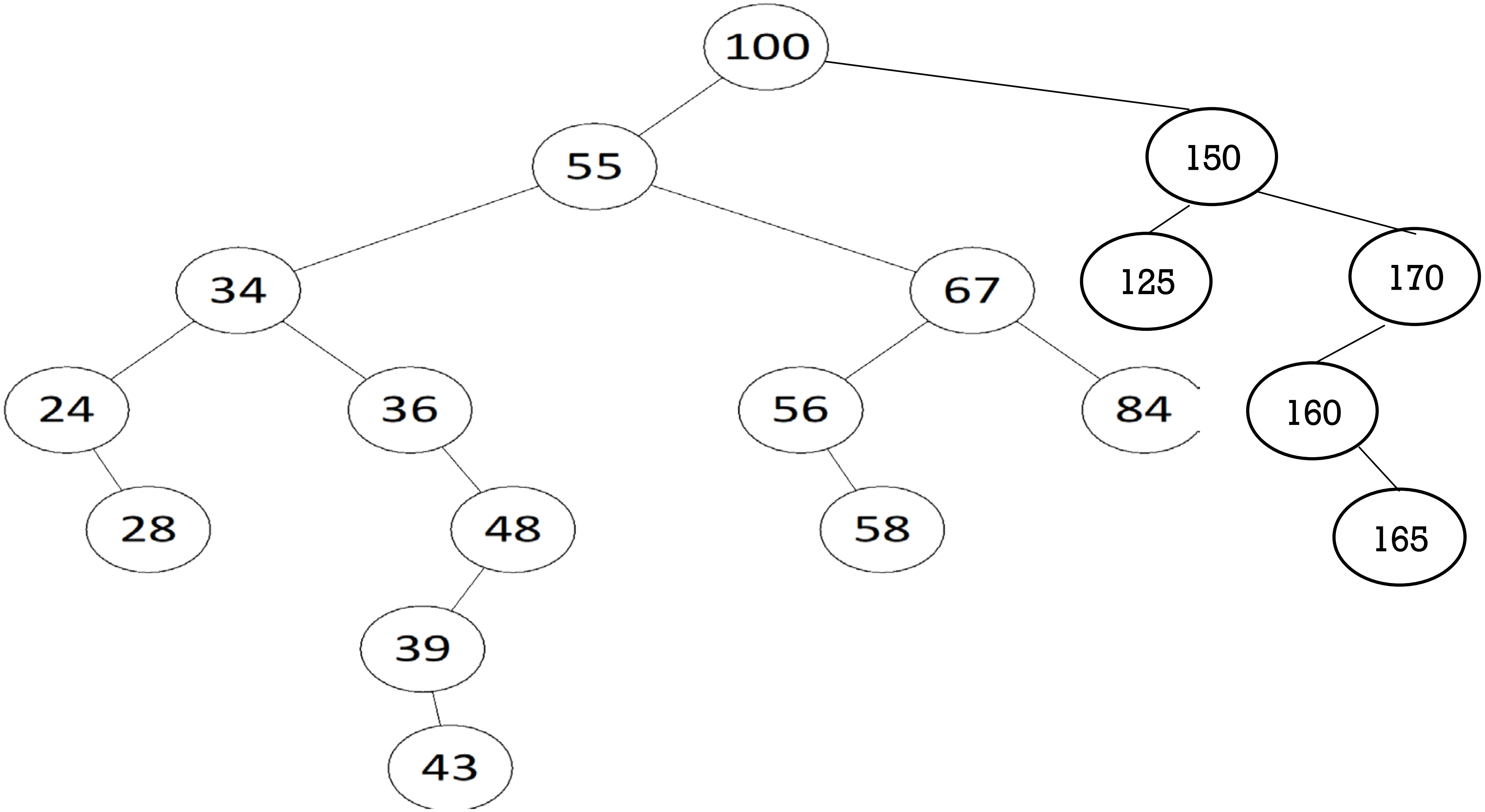
OUTLINE

- Binary Search Trees: Structure, operations, and time complexities
- Heaps: Structure, operations, array implementations, and time complexities
- Union-Find Data Structure:
  - Specs
  - Conceptual and physical implementations
  - Three successively better implementations and their time analysis

BINARY SEARCH TREES
-- DEFINITION --

- **Definition**: A binary search trees (BST) T is data structure with a built-in organization where
  - The data is of any kind that has a comparator like ≤ (e.g., int, real, String)
  - The organization is a binary tree where for every node x:
    - x holds (among its data) a data field called key
    - all the nodes in the left subtree of x have keys that are ≤ the key of x, and
    - all the nodes in the right subtree of x have keys that are > the key of x.
  - The operations supported are: *search* (a), *insert* (a), *delete* (a)

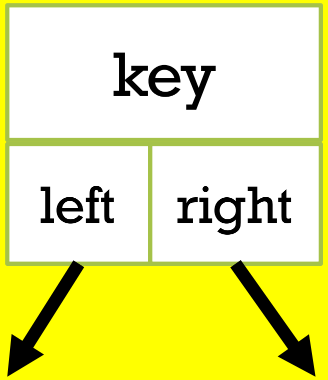
BINARY SEARCH TREES
-- EXAMPLE--



BINARY SEARCH TREES
-- SEARCH --

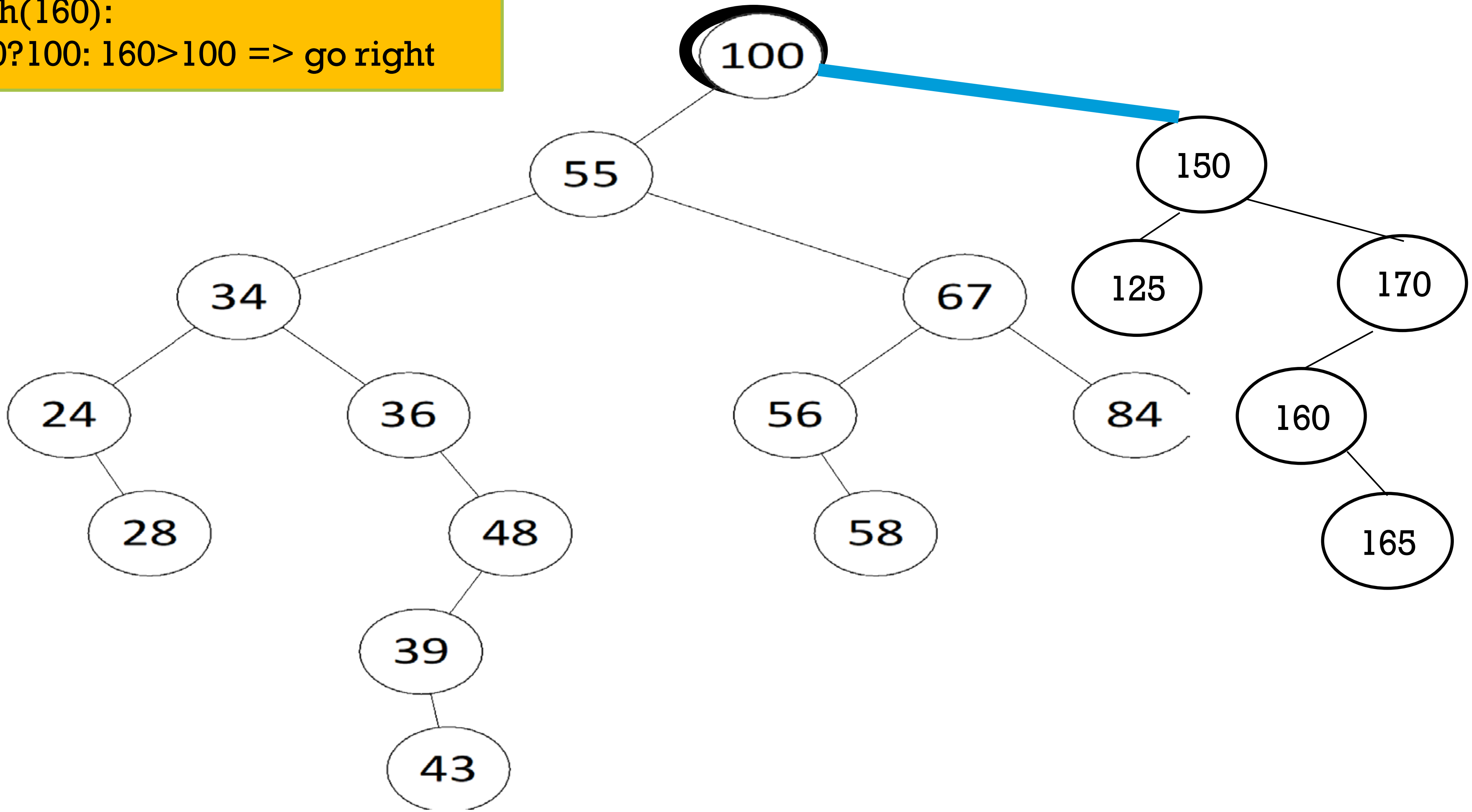
```
Function search(T,a)  // T is a nodeptr to the root node record
begin
    nodeptr p;
    p=T;
    while (p != null and  p.key != a) do
        if  a < p.key  then
            p := p.left;
        else
            p := p.right;
        endif
    endwhile
    return (p);
end search
```

```
record node
begin
    generic key;
    nodeptr left;
    nodeptr right;
end
```



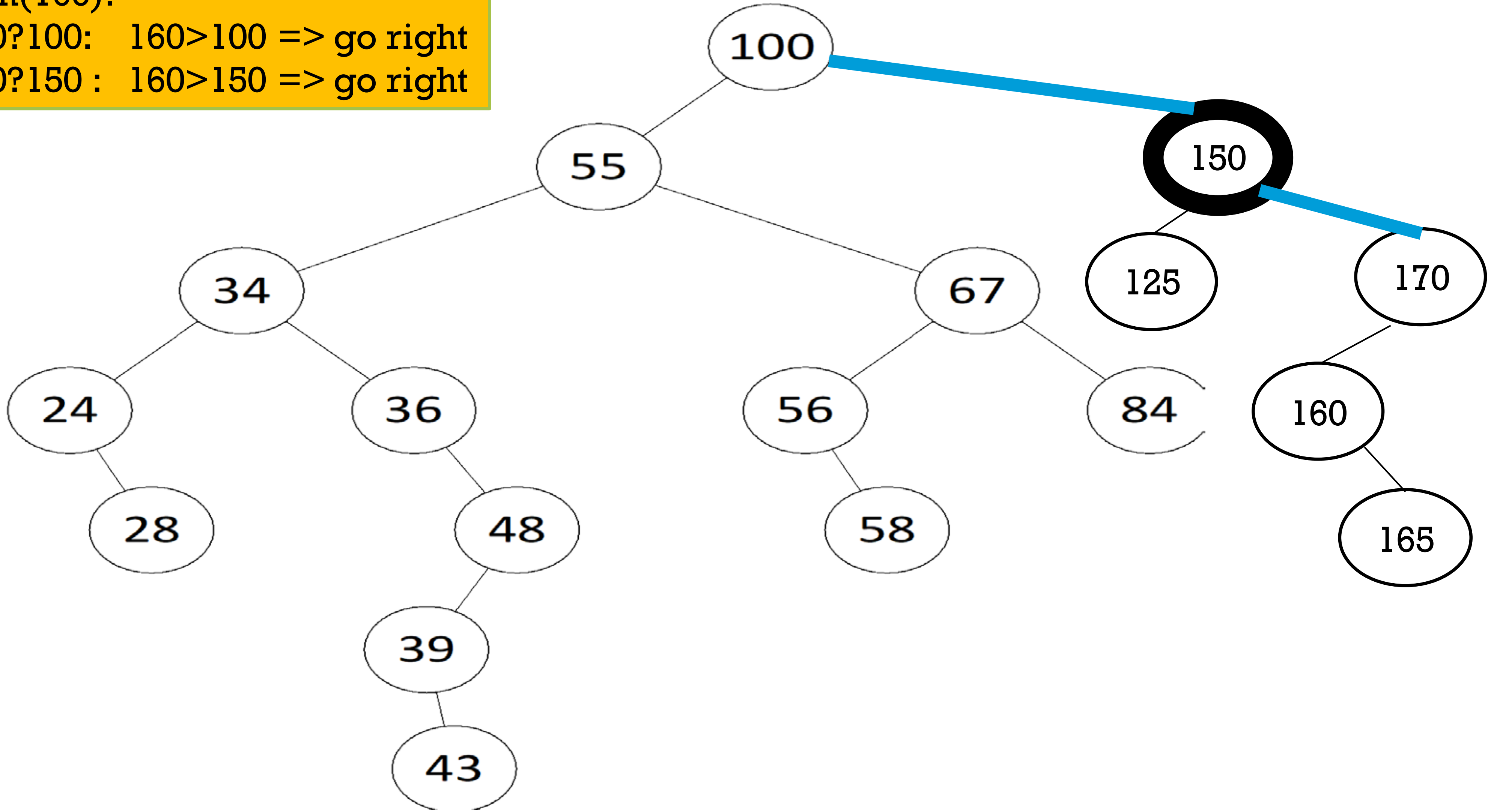
BINARY SEARCH TREES
-- EXAMPLE--

Search(160):
- 160?100: 160>100 => go right



BINARY SEARCH TREES
-- EXAMPLE--

Search(160):
- 160?100: 160>100 => go right
- 160?150 : 160>150 => go right

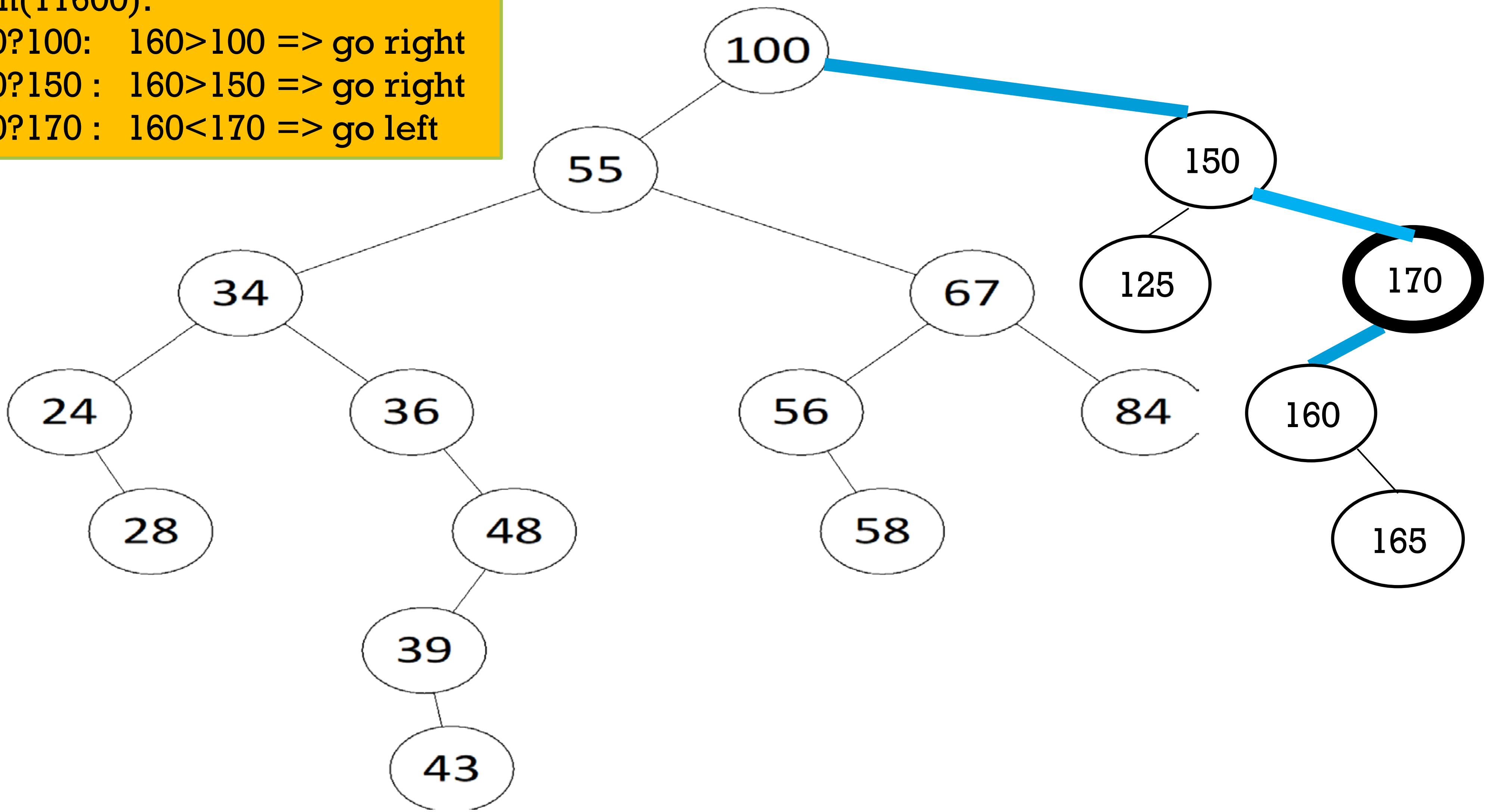


BINARY SEARCH TREES

-- EXAMPLE--

Search(11600):

- 160?100: 160>100 => go right
- 160?150 : 160>150 => go right
- 160?170 : 160<170 => go left

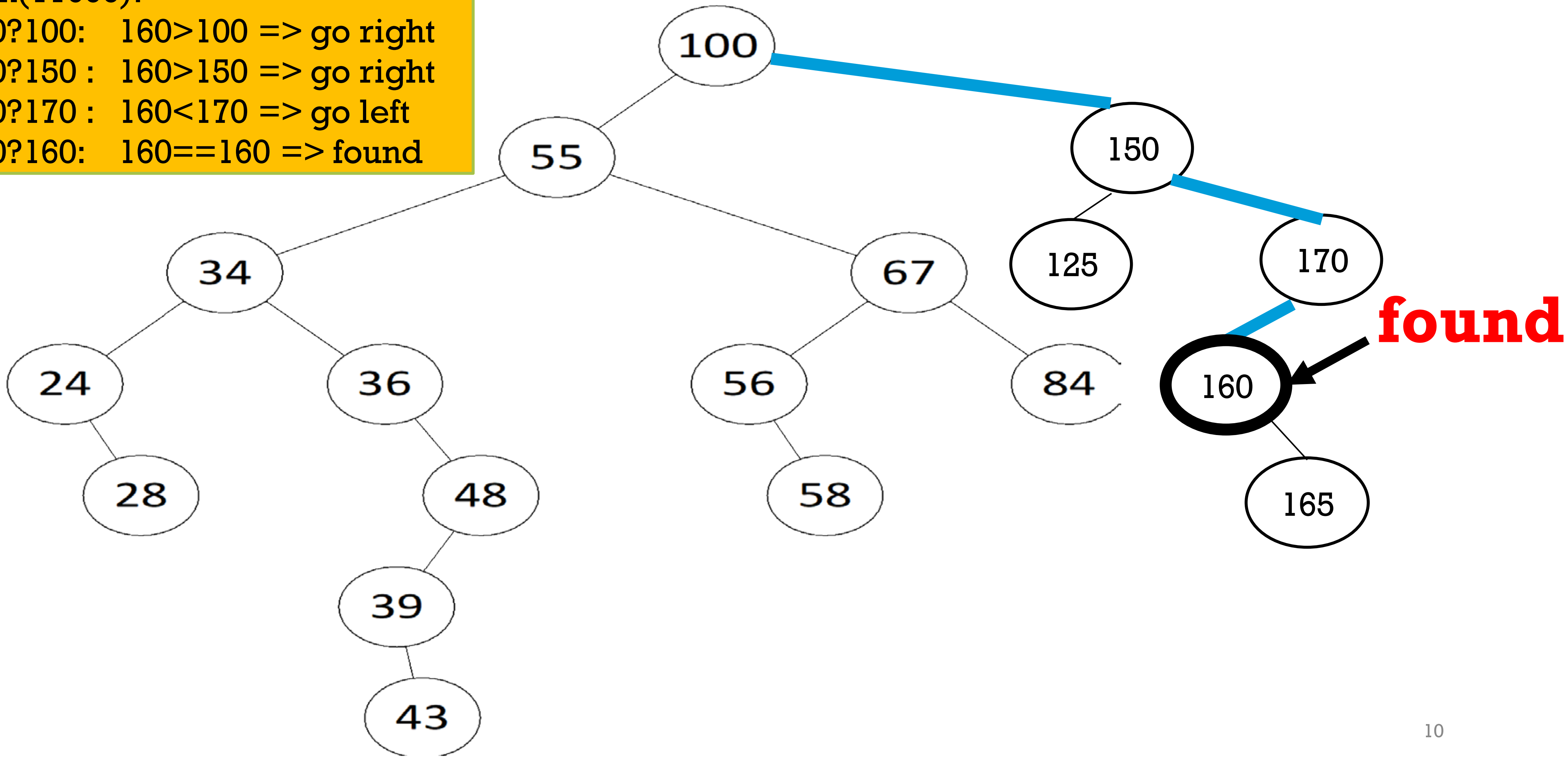


BINARY SEARCH TREES

-- EXAMPLE--

Search(11600):

- 160?100: 160>100 => go right
- 160?150 : 160>150 => go right
- 160?170 : 160<170 => go left
- 160?160: 160==160 => found



BINARY SEARCH TREES
-- SEARCH TIME COMPLEXITY--

- Search(T,a) takes as many comparisons as
  $depth_T(a) + 1 = O(depth(T) + 1) = O(height(T) + 1) = O(h + 1) = O(h)$

- Therefore, search takes $O(h)$ time

BINARY SEARCH TREES
-- INSERT --

- Insert(T,a) method
  1. Search for the **<u>missing</u>** node containing a
     a. keep record of the parent p
     b. Keep record whether the missing node is a left child or a right child of p
  2. Create a new node q, and put a in it
     a. Have p point to q

BINARY SEARCH TREES

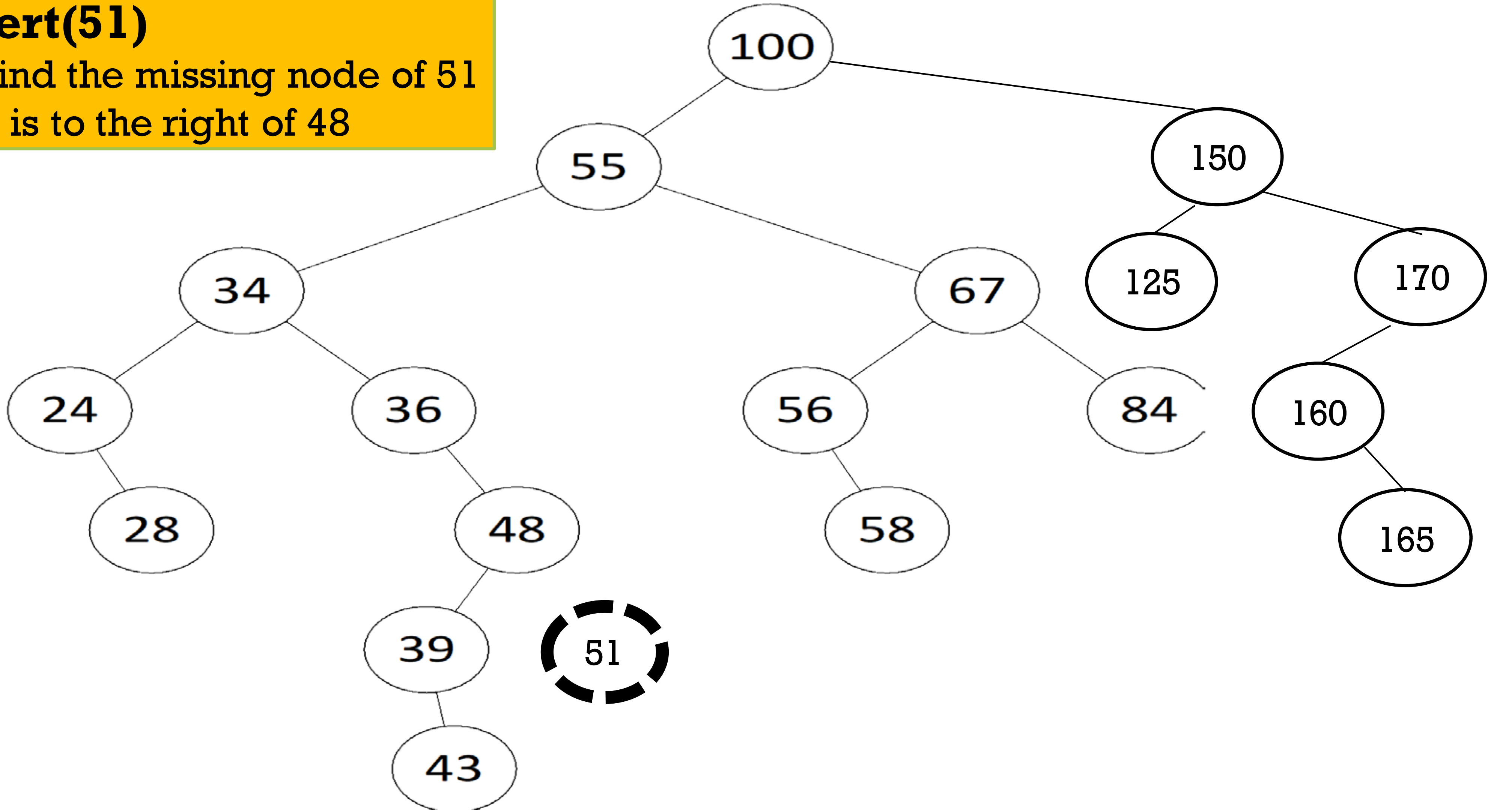
-- INSERT --

```
procedure insert(T,a)
begin
    nodeptr p;
    p=T;
    Bool done = false;
    while not done do
      if a <= p.key then
          if p.left != null then
              p = p.left;
          else
              p.left = new (node);
              p.left.key = a;
              done =true;
          endif
```

```
// Continue insert here
    else
          if p.right != null then
              p = p.right;
          else
              p.right = new (node);
              p.right.key := a;
              done = true;
          endif
       endif
    endwhile
end insert
```

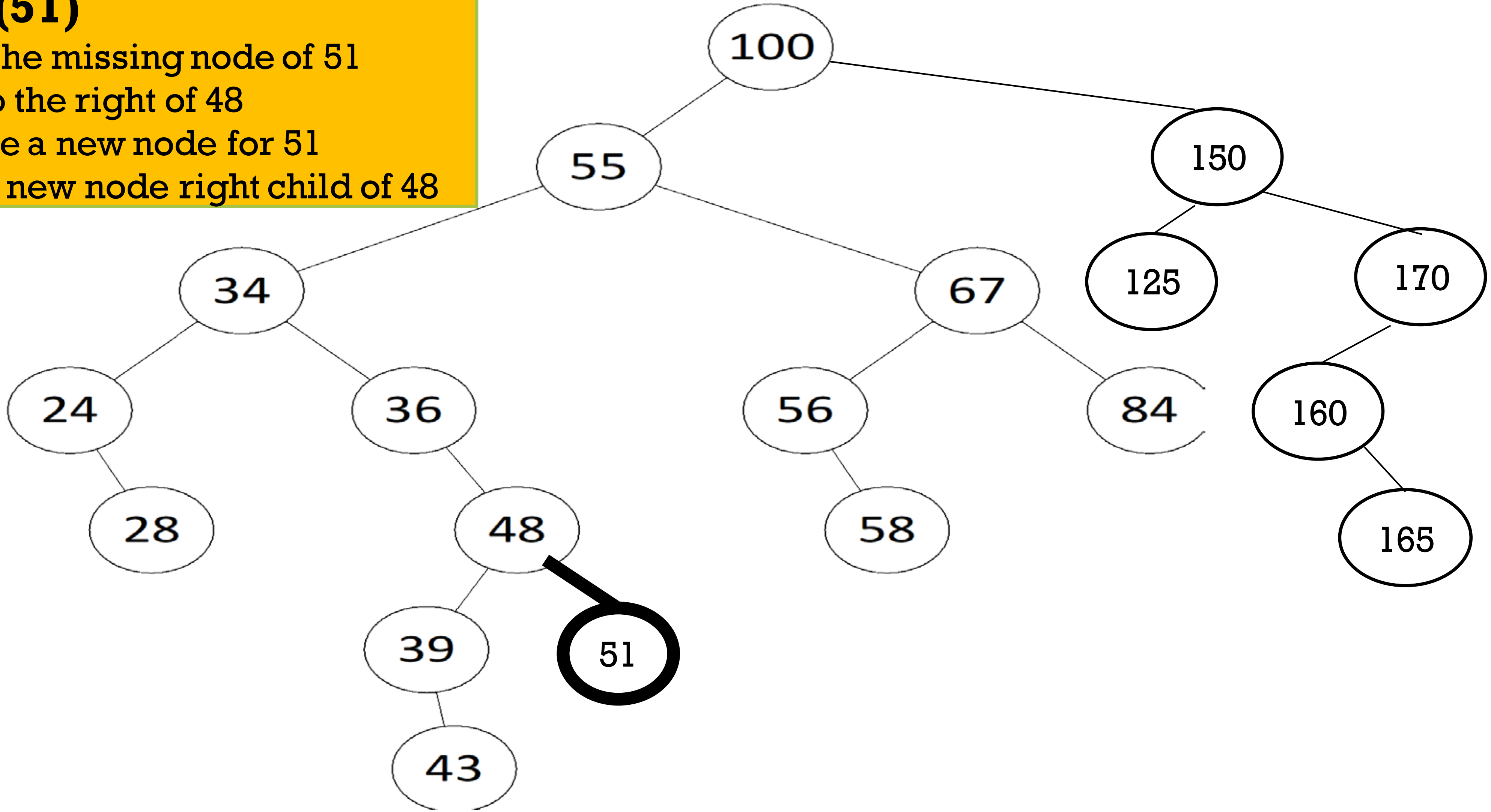
BINARY SEARCH TREES
-- INSERT EXAMPLE (INSERT(51)) --

Insert(51)
- Find the missing node of 51
- It is to the right of 48



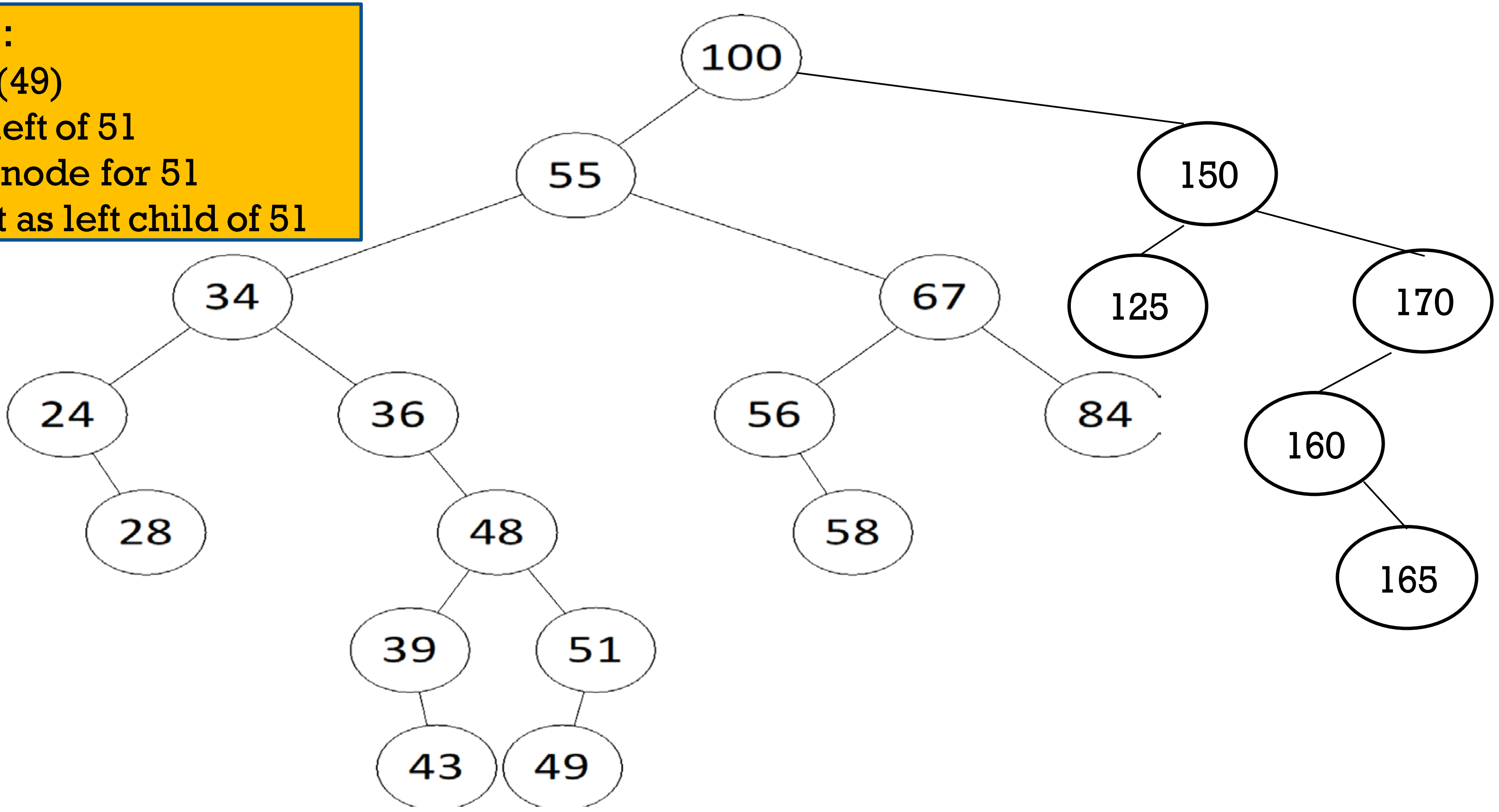
BINARY SEARCH TREES
-- INSERT EXAMPLE (INSERT(51)) --

Insert(51)
- Find the missing node of 51
- It is to the right of 48
- Create a new node for 51
- Make new node right child of 48



BINARY SEARCH TREES
-- INSERT EXAMPLE (INSERT(49)) --

Insert(49):
- Search(49)
- Found left of 51
- Create node for 51
- Insert it as left child of 51



BINARY SEARCH TREES
-- COMPLEXITY OF INSERT --

Recall that:

Insert(T,a) method

1. Search for the **missing** node containing a ← Time: O(h)
   a. keep record of the parent p
   b. Keep record whether the missing node is a left child or a right child of p
2. Create a new node q, and put a in it ← Time: O(1)
   a. Have p point to q ← Time: O(1)

Therefore, time of Insert is: O(h)+O(1) = O(h)

BINARY SEARCH TREES
-- DELETE --

Procedure delete(T,a)

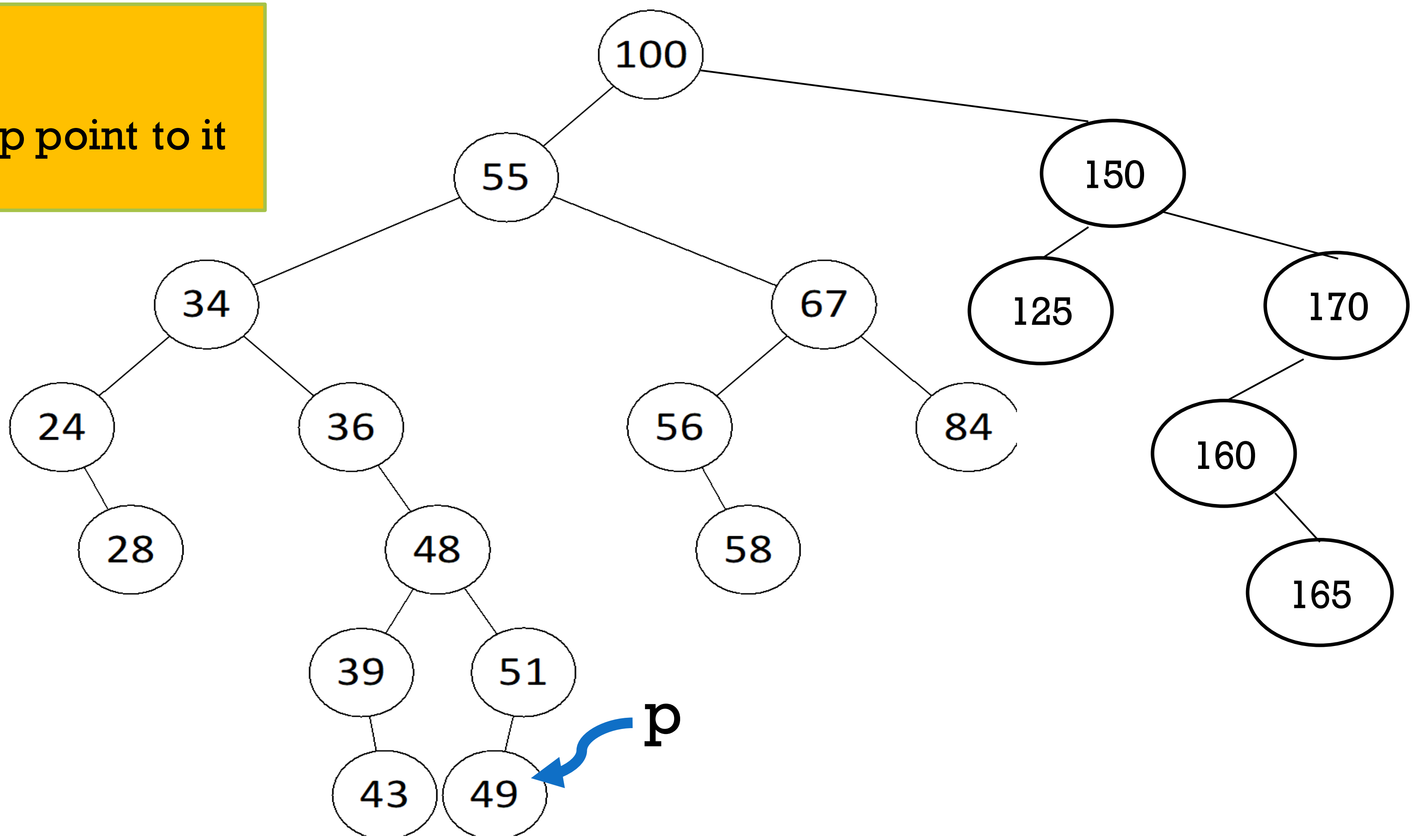
1. Search for a; if not found, return;
2. Let p be the pointer pointing to the node containing a;
3. If p is a leaf, remove it (making its parent's corresponding pointer null), and return;
4. If p has one child, make that child take the place of node p, and return;
5. If p has two children:
   a. Search for the largest (rightmost) node in the left subtree of p, and call it q;
   b. Move the key of q to node p; // now q is an empty node
   c. If q is a leaf, delete and return;
   d. Else, q has a left child only: bypass it as in step 4, and return;

BINARY SEARCH TREES

-- DELETE EXAMPLE (DELETE(49)) --

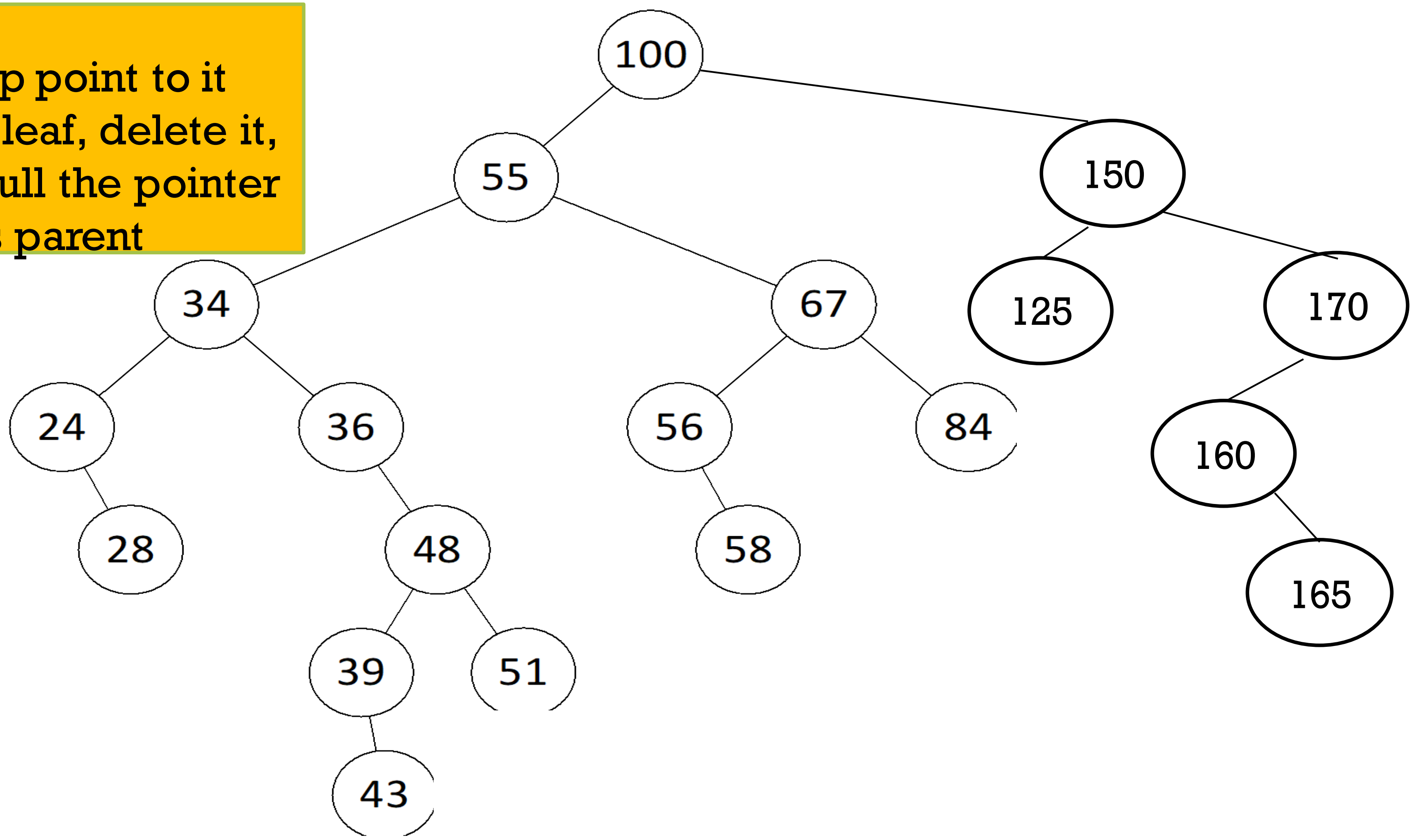
Delete (49):

- Find 49; let p point to it



BINARY SEARCH TREES
-- DELETE EXAMPLE (DELETE(49)) --

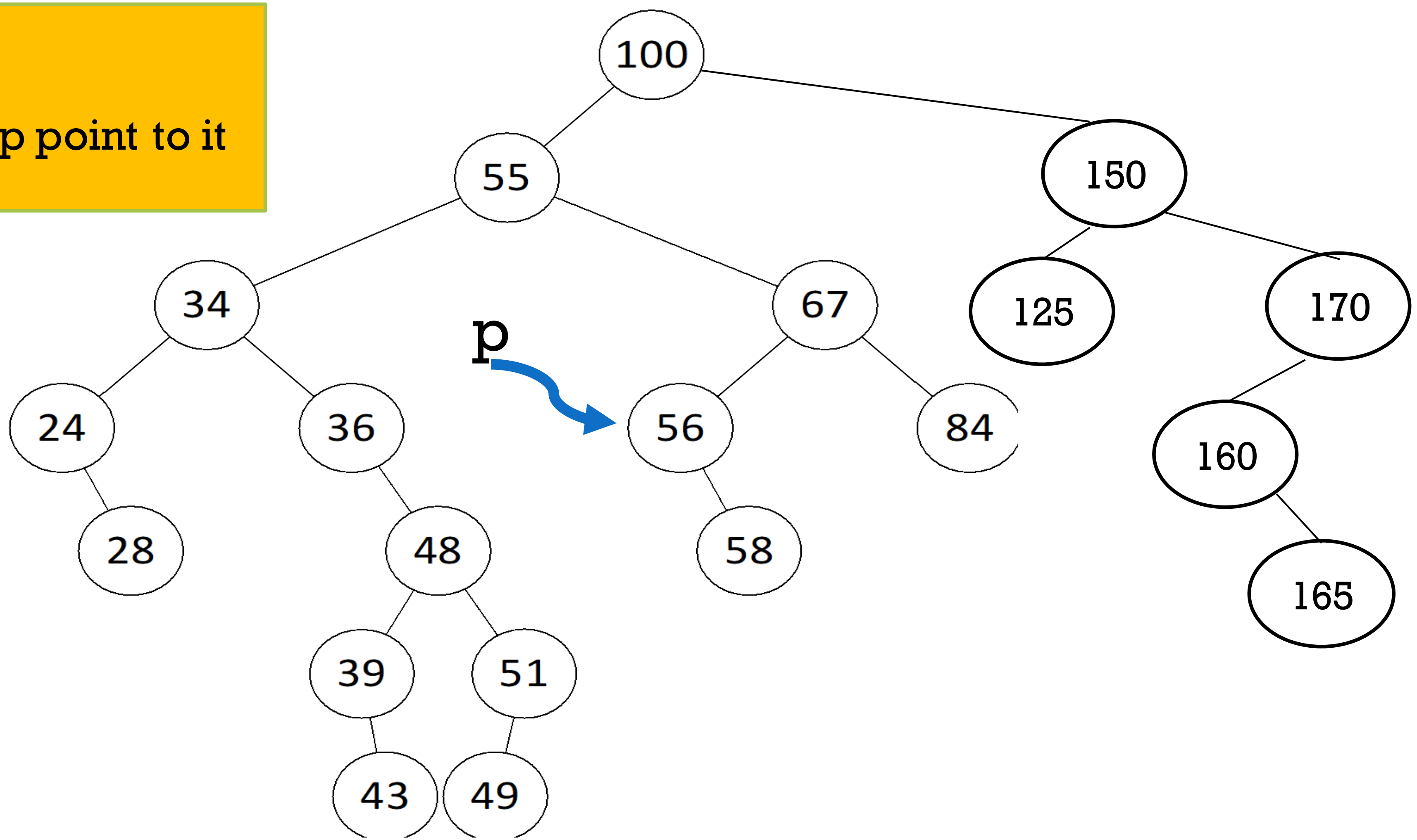
Delete (49):
- Find 49; let p point to it
- Since it is a leaf, delete it, and set to null the pointer to it from its parent



BINARY SEARCH TREES
-- DELETE EXAMPLE (DELETE(56)) --

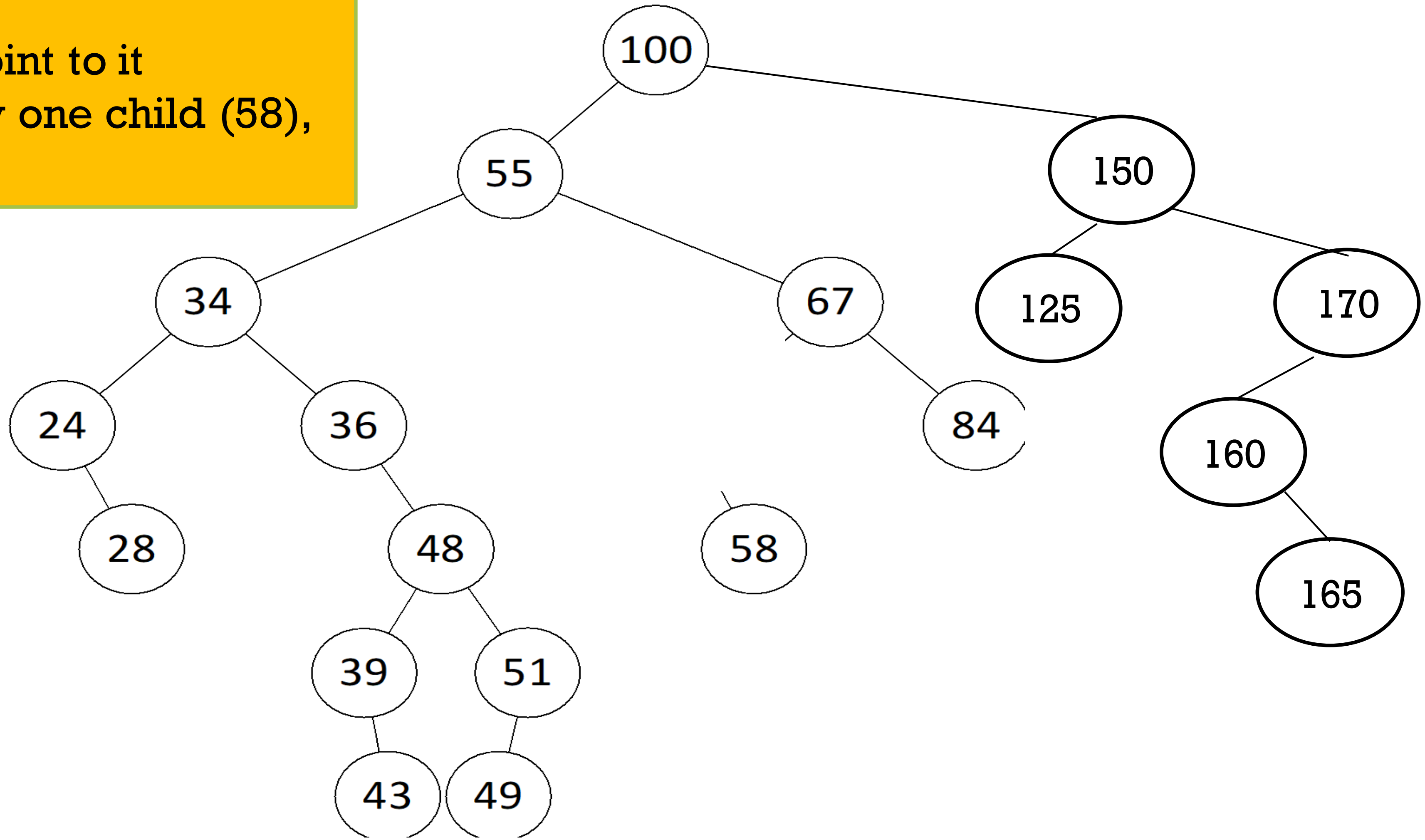
Delete (56):

- Find 56; let p point to it



BINARY SEARCH TREES
-- DELETE EXAMPLE (DELETE(56)) --

Delete (56):
- Find 56; let p point to it
- Since it has only one child (58), delete 56,

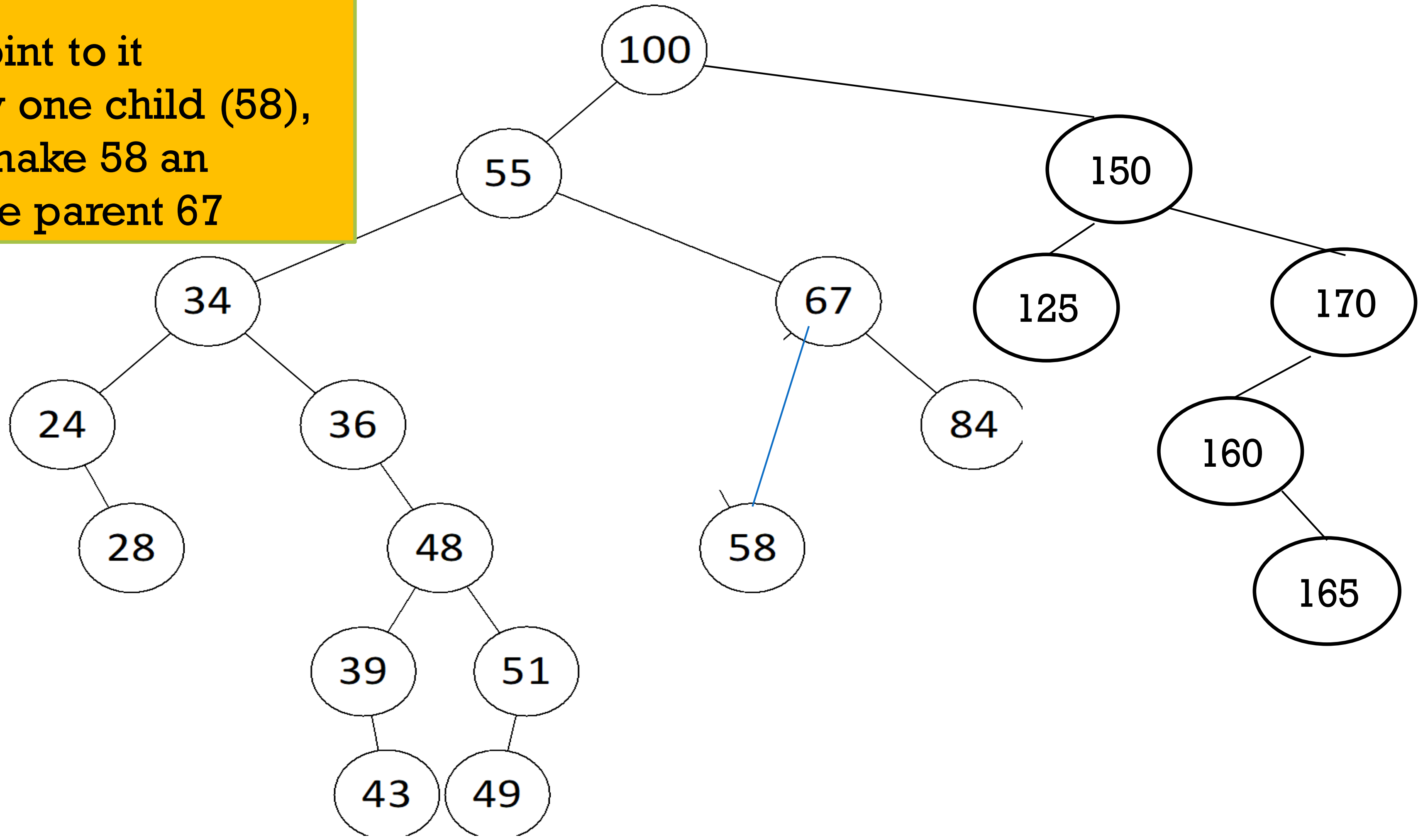


BINARY SEARCH TREES

-- DELETE EXAMPLE (DELETE(56)) --

Delete (56):

- Find 56; let p point to it
- Since it has only one child (58), delete 56, and make 58 an immediate of the parent 67

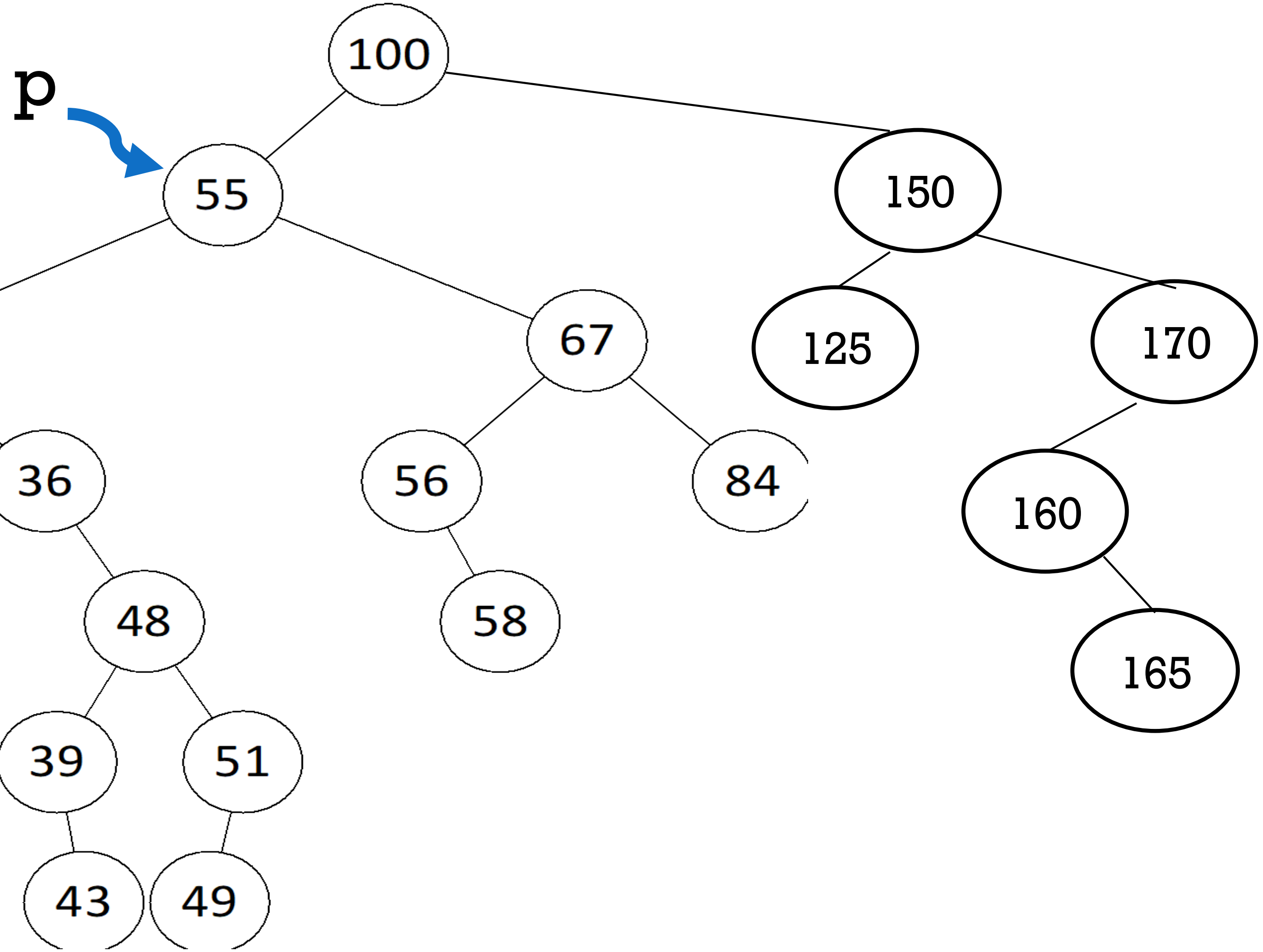


BINARY SEARCH TREES

-- DELETE EXAMPLE (DELETE(55)) --

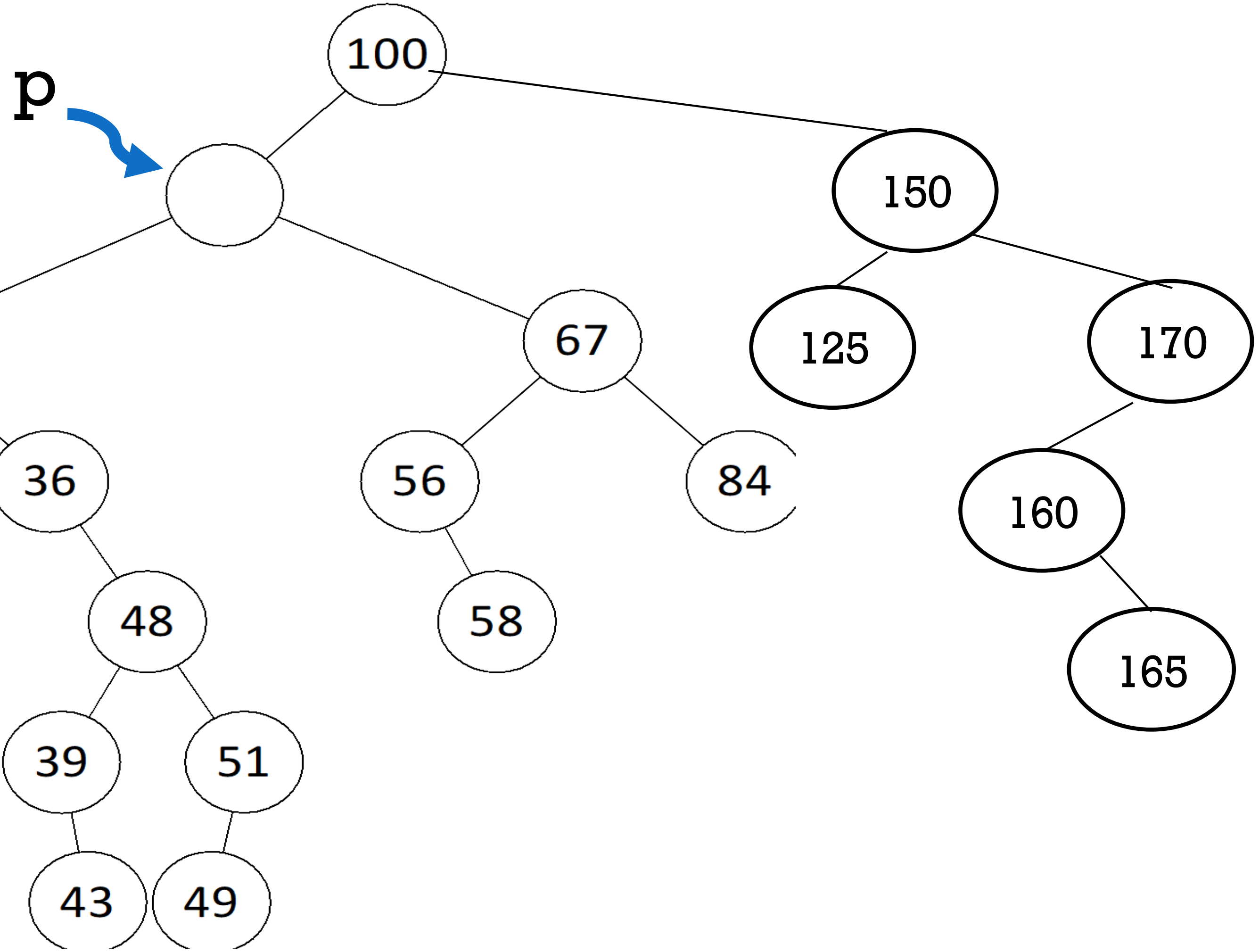
Delete (55):

- Find 55; let p point to it



BINARY SEARCH TREES
-- DELETE EXAMPLE (DELETE(55)) --

Delete (55):
- Find 55; let p point to it
- Delete the key of p

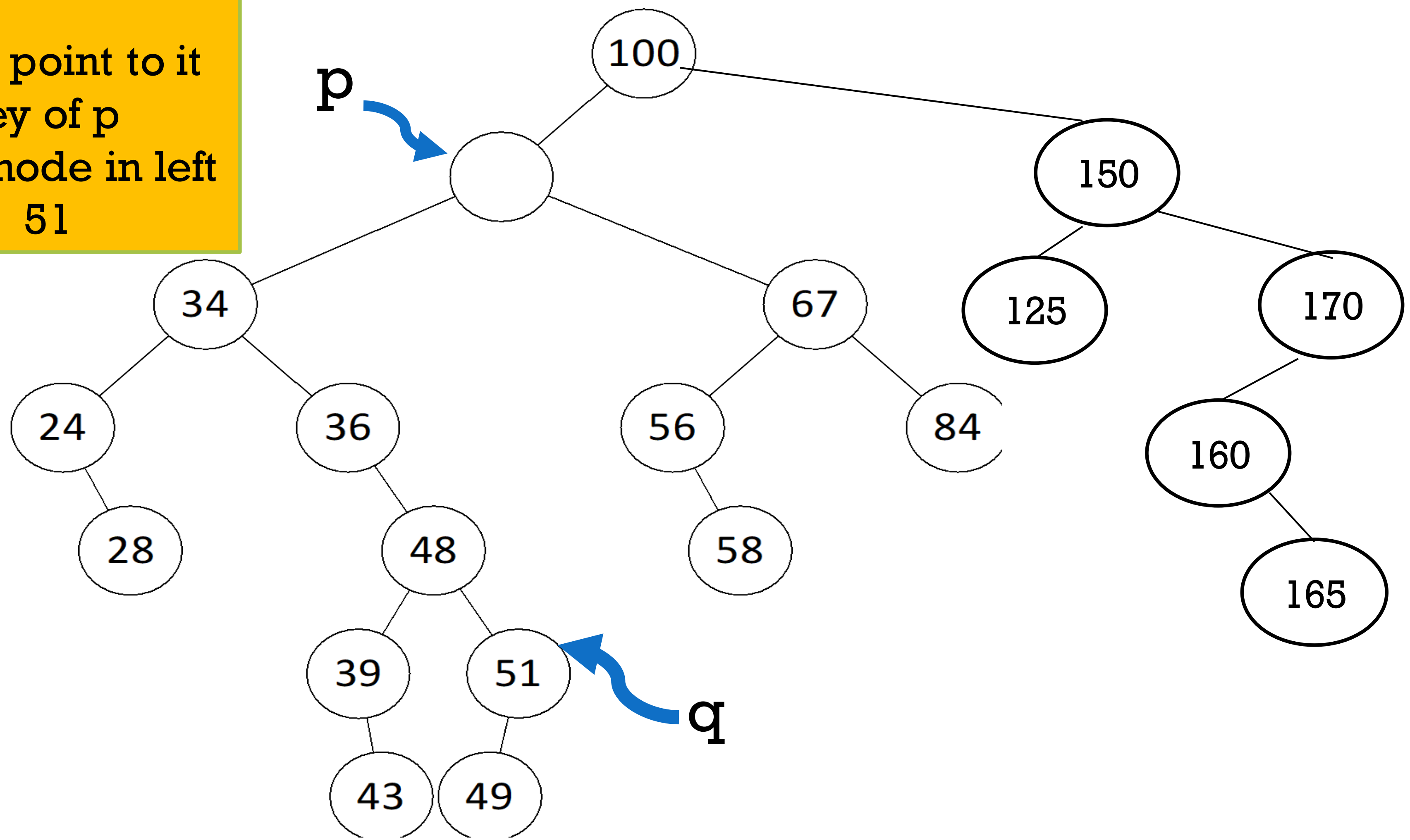


BINARY SEARCH TREES

-- DELETE EXAMPLE (DELETE(55)) --

Delete (55):

- Find 55; let p point to it
- Delete the key of p
- Find largest node in left subtree of p: 51

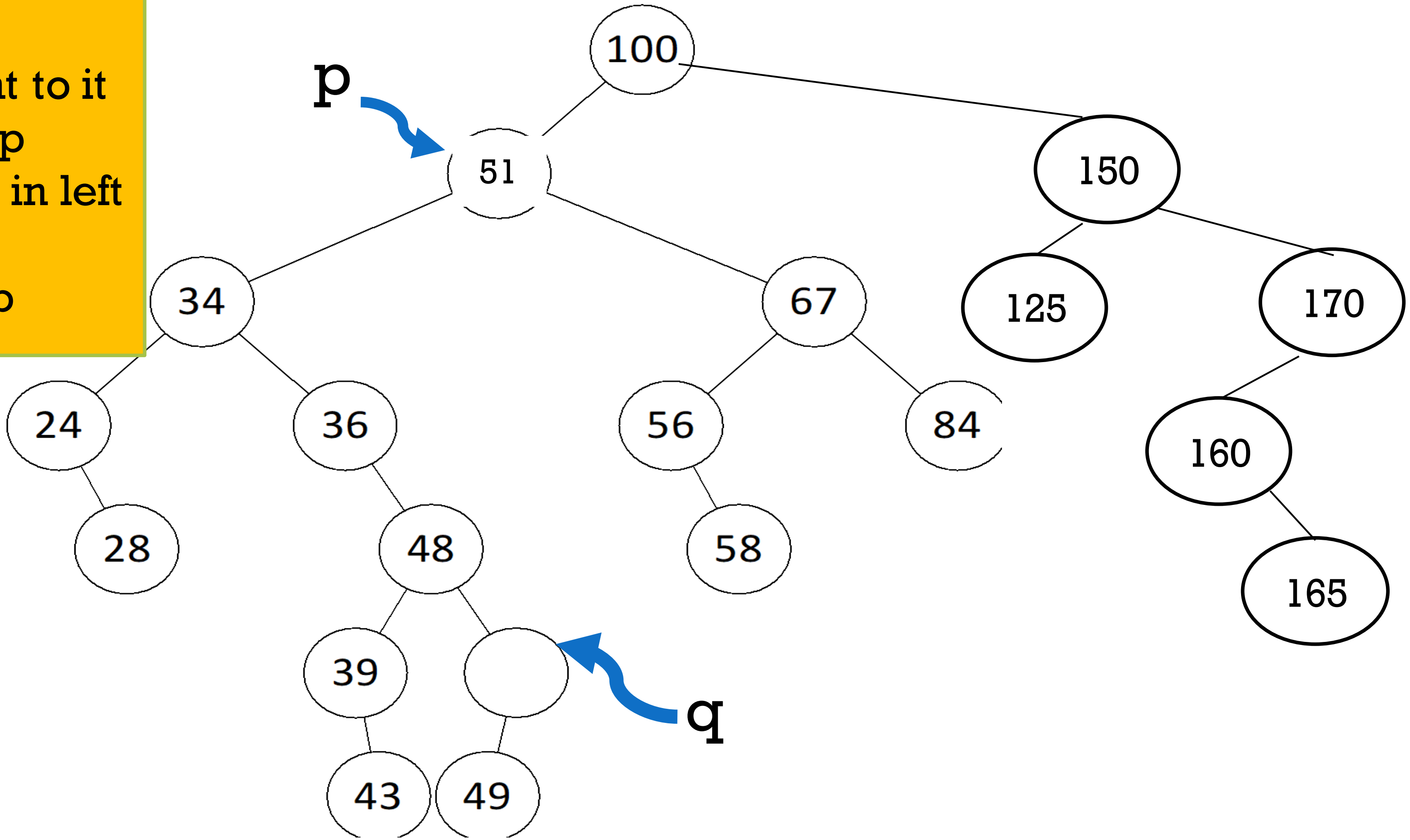


BINARY SEARCH TREES

-- DELETE EXAMPLE (DELETE(55)) --

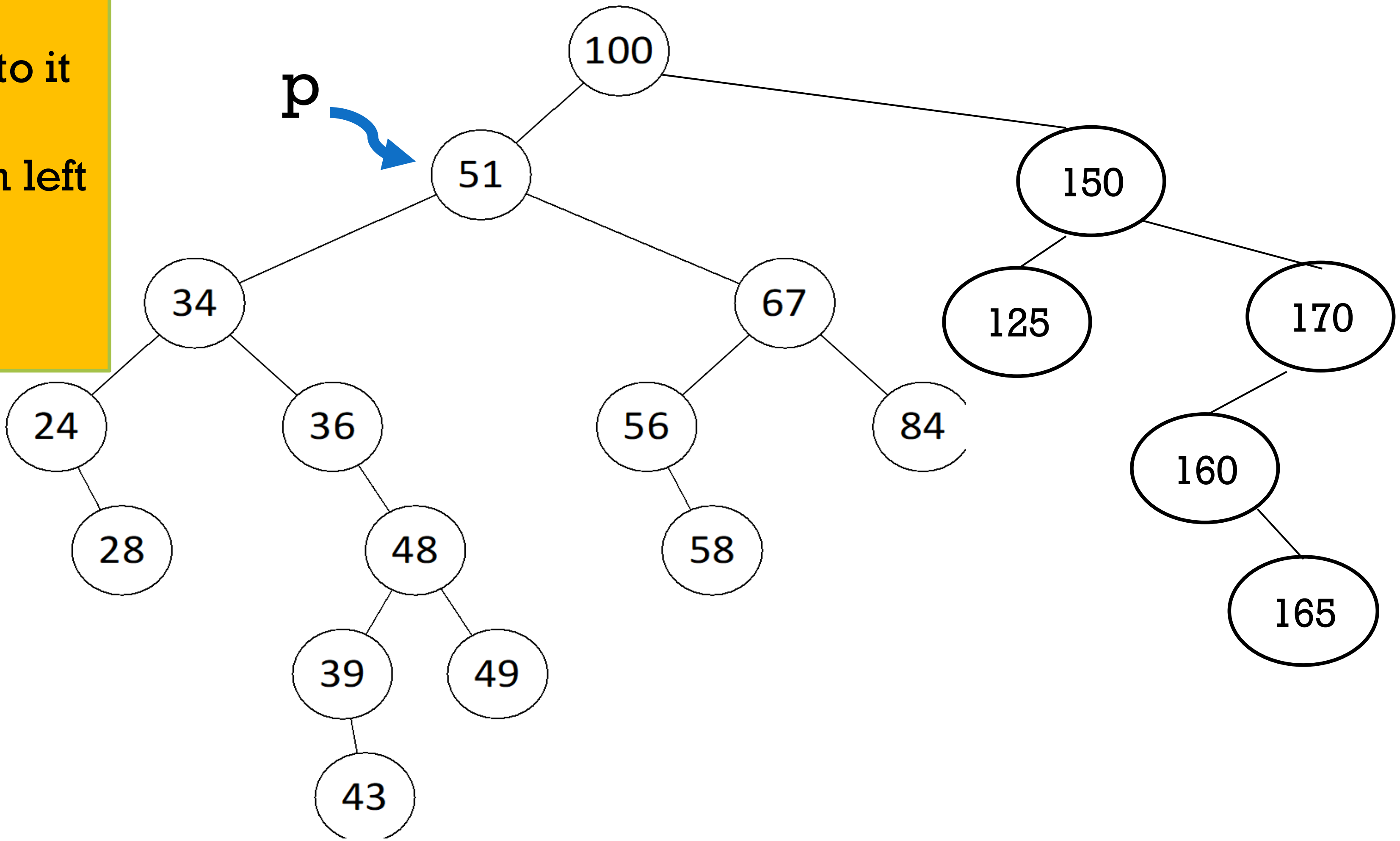
Delete (55):

- Find 55; let p point to it
- Delete the key of p
- Find largest node in left subtree of p: 51
- Move key of q to p



BINARY SEARCH TREES
-- DELETE EXAMPLE (DELETE(55)) --

Delete (55):
- Find 55; let p point to it
- Delete the key of p
- Find largest node in left subtree of p: 51
- Move key of q to p
- Bypass q



BINARY SEARCH TREES
-- DELETE PSEUDOCODE (1/3) --

```
procedure  delete(T,a)
begin
    nodeptr  p,q,r,s;
    integer  direction;
    p = T;
    while  (p != null and p.key != a) do
        if a < p.key then
            q := p;
            p := p.left;
            direction := 0;
        else
            q := p;
            p := p.right;
            direction := 1;
        endif
    endwhile
```

```
// continue delete here
    if p == null then  return;
    elseif p.left == null and p.right == null then
        // p has no children; delete that node
        if direction == 0 then q.left = null;
        else q.right = null ;
        endif
        free (p);
    elseif p.left == null then
        // p has only one child, the right one
        if direction == 0 then  q.left := p.right;
           //shortcut from parent to grandchild
        else  q.right := p.right;
        endif
```

BINARY SEARCH TREES

-- DELETE PSEUDOCODE (2/3) --

```
// continue delete here
        elseif p.right == null then
         // p has only one child, the left one
                if direction == 0 then
                        q.left := p.left;
                else
                        q.right := p.left;
                endif
        else
        // p has two children
        // find the maximum node in the
        // left subtree of p
                s := p.left;
                q := p;
```

```
// continue delete here
                // now q will be the parent of
                // s , and direction will
                // indicate the type of child s
                // is to q

                direction = 0;
                while s.right != null do
                        q := s;
                        s := s.right;
                        direction := 1;
                endwhile
                // Now s points to the maximum node
                // in the left subtree of p
                p.key := s.key;
```

BINARY SEARCH TREES

-- DELETE PSEUDOCODE (3/3) --

```
// continue delete here
        // now node s must be deleted. But since s has no right child,
        // the deletion is done by deletion or shortcutting
        if s.left == null then        // s is a leaf
            if direction == 0 then  q.left := null;
            else q.right := null;
            endif
            free (s);
            return;
        else                        // s has a left child
            if direction == 0 then q.left := s.left;
            else  q.right := s.left;
            endif
            free(s) ; return;
        endif
    endif
end delete
```

BINARY SEARCH TREES
-- COMPLEXITY OF DELETE --

Procedure delete(T,a)

1. Search for a; if not found, return;
2. Let p be the pointer pointing to the node containing a;
3. If p is a leaf, remove it (making its parent's corresponding pointer null), and return;
4. If p has one child, make that child take the place of node p, and return;
5. If p has two children:
   a. Search for the largest (rightmost) node in the left subtree of p, and call it q;
   b. Move the key of q to node p; // now q is an empty node
   c. If q is a leaf, delete and return;
   d. Else, q has a left child only: bypass it as in step 4, and return;

$O(depth_T(a))=O(depth_T(p))$ (steps 1–2)

Time: O(1) (step 3)

Time: O(1) (step 4)

Time: $O(depth_T(q)-depth_T(p))$ (step 5a)

Time: O(1) (step 5b)

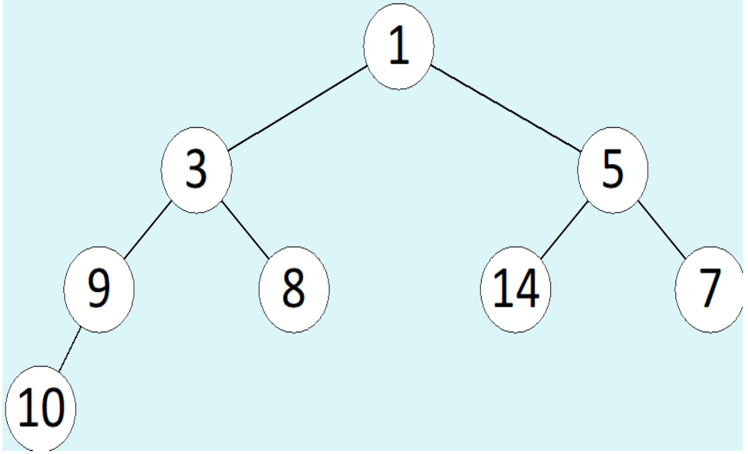
Time: O(1) (step 5c)

Time: O(1) (step 5d)

Therefore, Delete Time: $O(depth_T(p))+O(depth_T(q)-depth_T(p))+O(1)=O(depth_T(q))=O(h)$

HEAPS
-- DEFINITION --

- **Definition**: A heap H is data structure with a built-in organization where
  - The data is of any type that has a comparator like ≤ (e.g., int, real, String)
  - The organization is an **almost complete binary tree** where for all nodes x:
    - x holds (among its data) a data field called key
    - The key of x is ≤ the keys of its children
  - The operations supported are:
    - **delete-min**(): it finds & deletes the minimum value m, restores the heap, and returns m.
    - **insert**(H,a): inserts a new value a into the heap
- Notes: the minimum is at the root

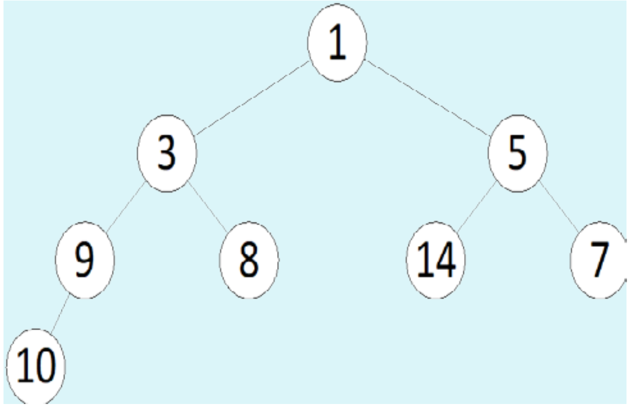


- For ≤: it is min-heap
- For ≥: it is a max-heap

HEAPS

-- USES--

- A heap implements a **priority queue**
  - Unlike the familiar queue which implements "first-come, first serve"
  - It implements "first-priority, first serve"
  - So, to select (and remove) the next item from the priority queue
    1. we look for the item of highest priority/importance (e.g., of priority 1)
    2. remove it from the waiting (priority) queue, and serve it.



- That is accomplished using **delete-min()**
- As new items come to the waiting line,

  they have to be inserted, using **insert(...)**
- Operating systems use heaps to prioritize waiting processes

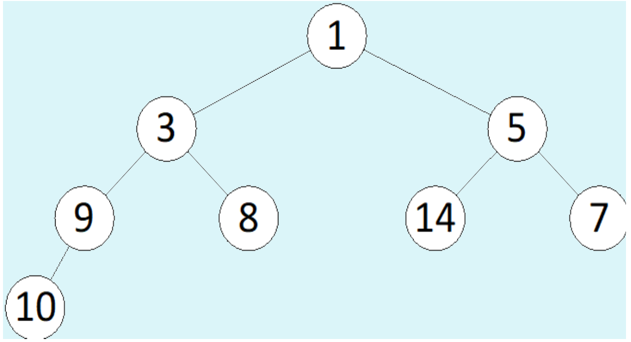
HEAPS

-- INSERT --

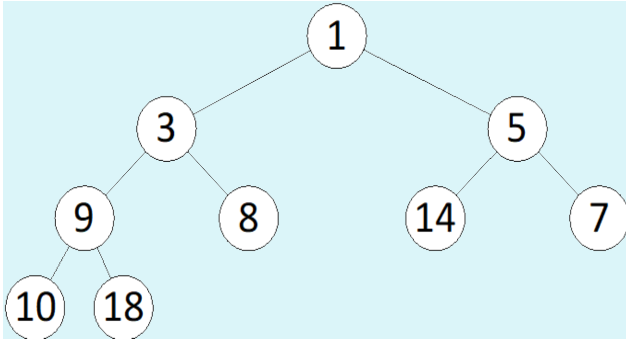
```
procedure insert(H,a)        // inserts the key value a into the heap
begin
    create a node of label n+1, and insert a into that node;
    let x point to that node;
    while (x < its parent   or   x is not root) do
        swap the key of x with key of the parent of x;
        let x point to its parent;
    endwhile
end insert
```

HEAP
-- INSERT EXAMPLE (INSERT(H,18))--

- Let H be the following heap:

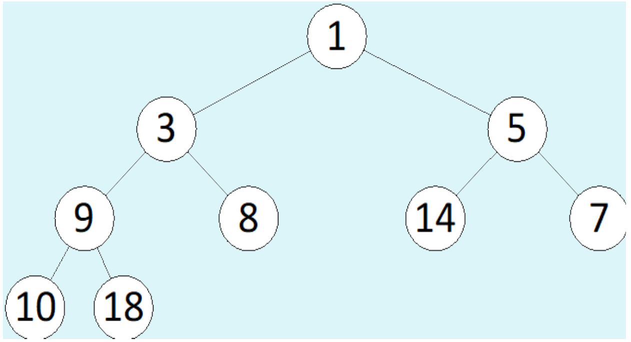


- Insert(H,18):
  - Put 18 as the next node while preserving the almost-complete structure
  - Restore heap: well, 18 is already $\geq$ its parent
  - So, no restoration is needed



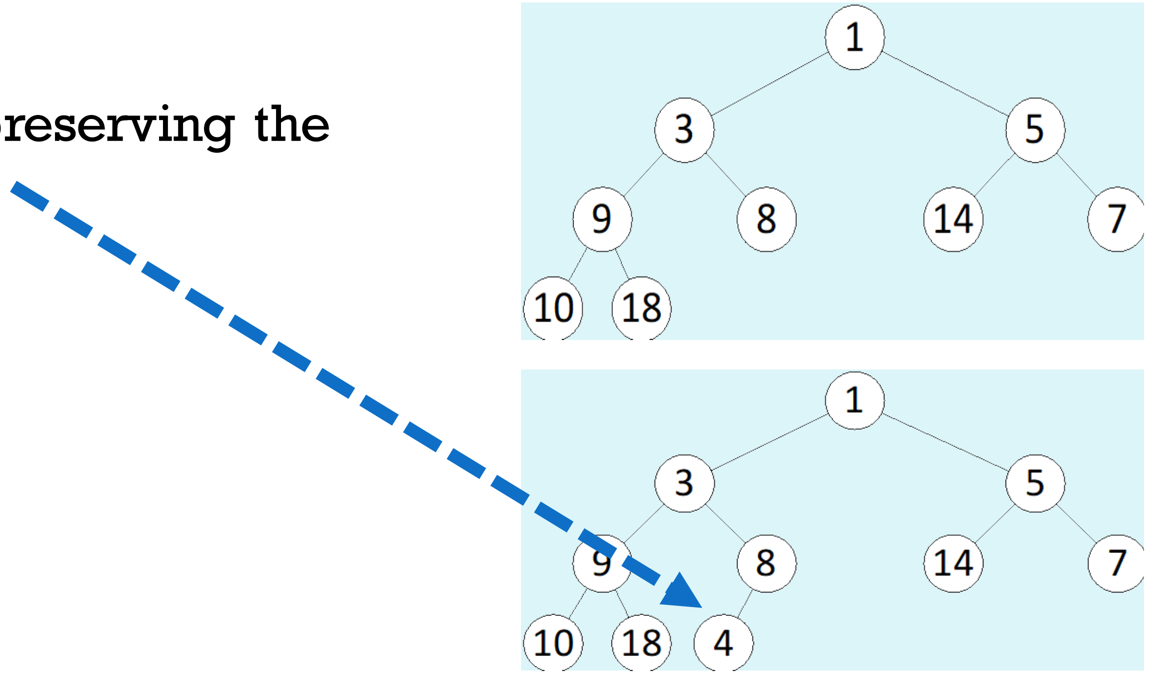
HEAP
-- INSERT EXAMPLE (INSERT(H,4))--

- Insert(H,4):



HEAP
-- INSERT EXAMPLE (INSERT(H,4))--

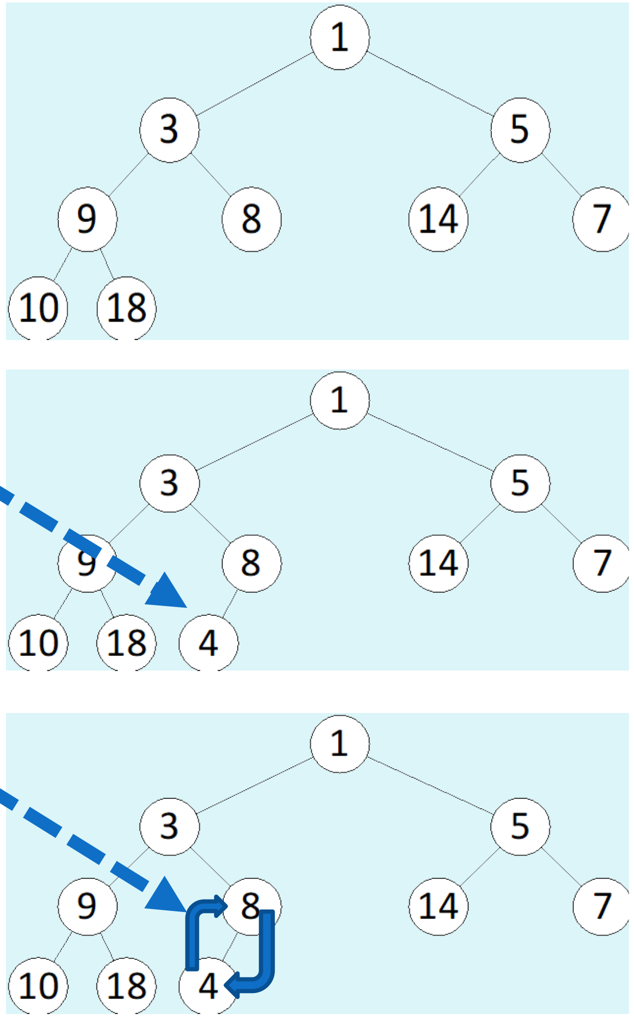
- Insert(H,4):
  - Put 4 as the next node while preserving the almost-complete structure



HEAP

-- INSERT EXAMPLE (INSERT(H,4))--

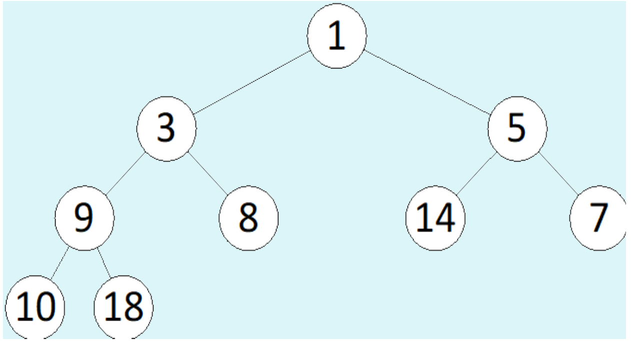
- Insert(H,4):
  - Put 4 as the next node while preserving the almost-complete structure
  - Restore heap:
    1. Since 4 < its parent 8, swap it with parent

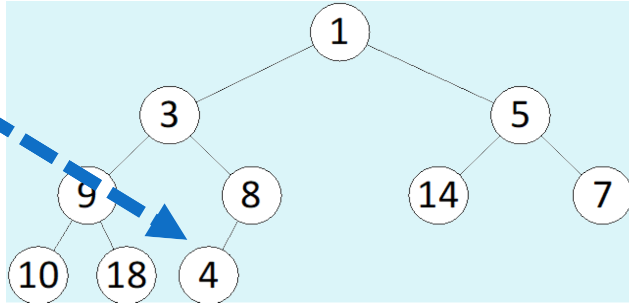


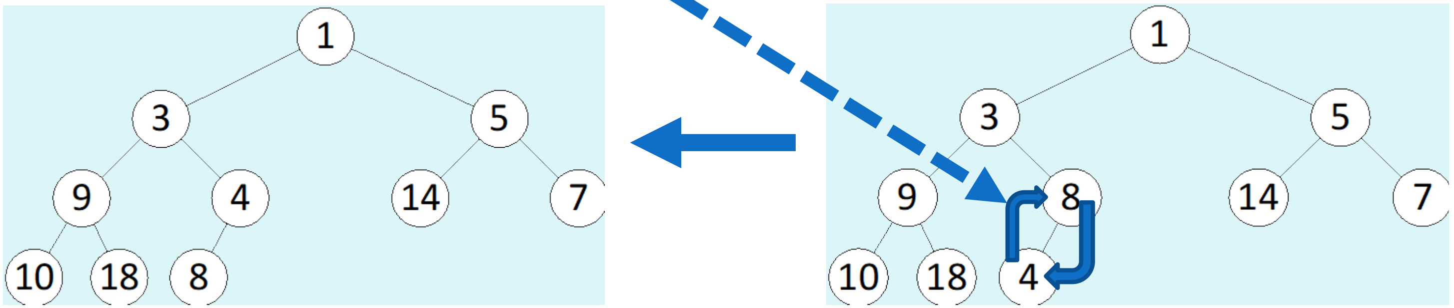
HEAP

-- INSERT EXAMPLE (INSERT(H,4))--

- Insert(H,4):
  - Put 4 as the next node while preserving the almost-complete structure
  - Restore heap:
    1. Since 4 < its parent 8, swap it with parent



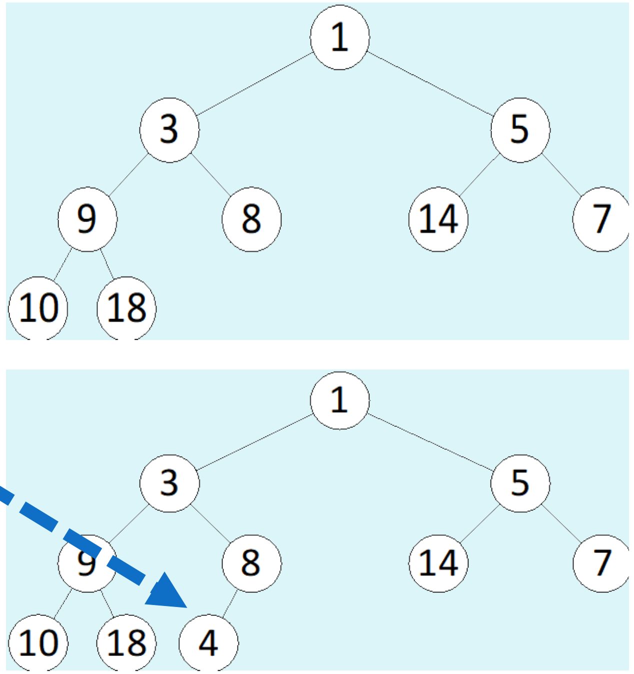


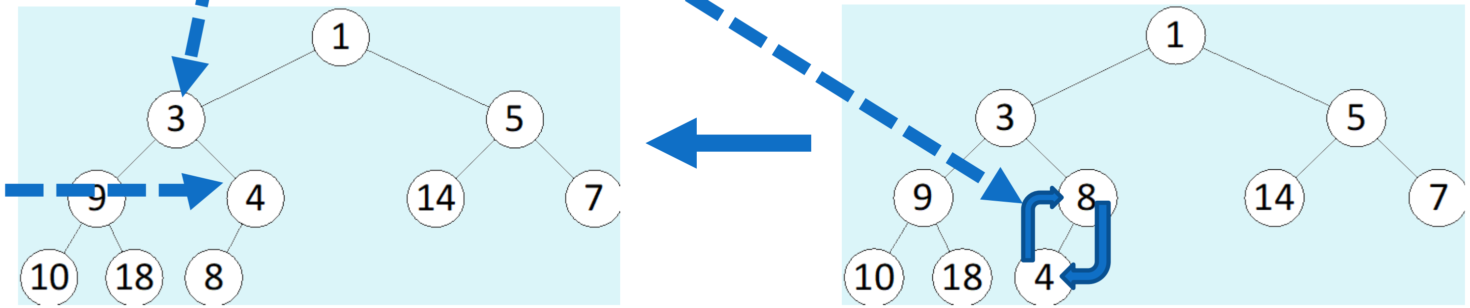


HEAP
-- INSERT EXAMPLE (INSERT(H,4))--

- Insert(H,4):
  - Put 4 as the next node while preserving the almost-complete structure
  - Restore heap:
    1. Since 4 < its parent 8, swap it with parent
    2. Now 4 ≥ its new parent 3,

       So the restoration is complete





HEAPS

-- TIME COMPLEXITY OF INSERT --

```
procedure insert(H,a) begin
    create a node of label n+1, and insert a into that node;
    let x point to that node;
    while (x < its parent   or   x is root) do
        swap the key of x with key of the parent of x;
        let x point to its parent;
    endwhile
end insert
```

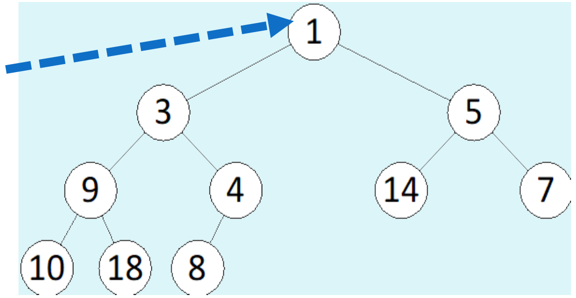
- The while-loop iterates at most the height of the tree
- Every iteration takes constant time (one comparison and one swap)
- Thus, Insert take O(h) time
- But for almost-complete trees, h=O(log n)
- Therefore, Insert takes **O(log n) time**

HEAPS
-- DELETE-MIN--

```
function delete-min(H) /* H is the heap*/
begin
    x= root of H;
    r=key of x;              // to be returned at the end
    remove r from node x;
    take the last node (node n), remove its key (call it b), and store b in the root;
    remove node n;
    // now restore the heap
    while  (x has a key bigger than one of its children) do
        swap x with the smaller child;
        make x point to that child;
    end while
    // the while loop will stop when x becomes a leaf or  ≤ both its children
    return r;
end delete-min
```

HEAPS
-- EXAMPLE OF DELETE-MIN() --

- Delete-min()
- The minimum is at the root (of value r=1)

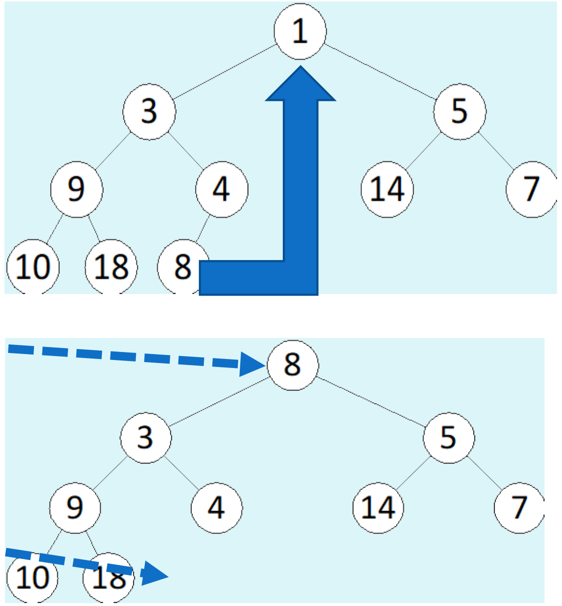


HEAPS

-- EXAMPLE OF DELETE-MIN() --

- Delete-min()
- The minimum is at the root (of value r=1)
- Replace the root value with the value of the last node

  and remove the last node

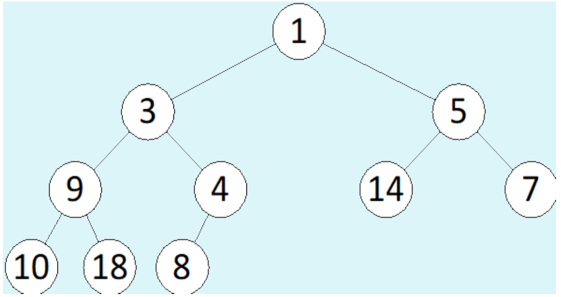


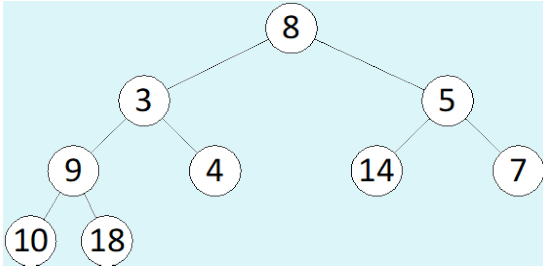
HEAPS

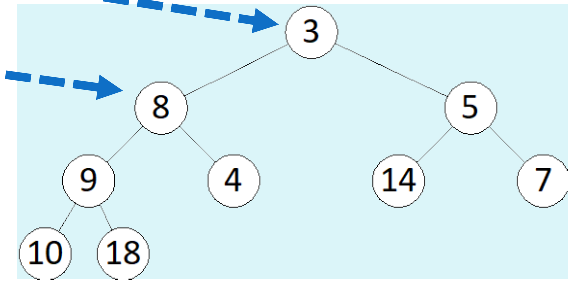
-- EXAMPLE OF DELETE-MIN() --

- Delete-min()
- The minimum is at the root (of value r=1)
- Replace the root value with the value of the last node

  and remove the last node
- Restore the heap
  - Swap 8 with its smaller child (3)



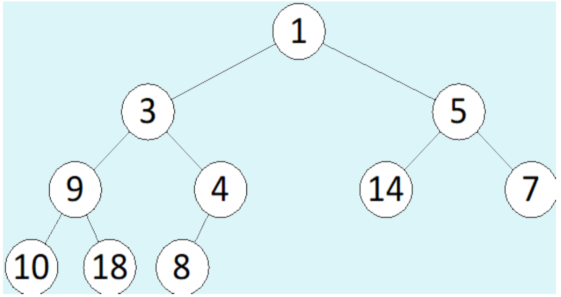


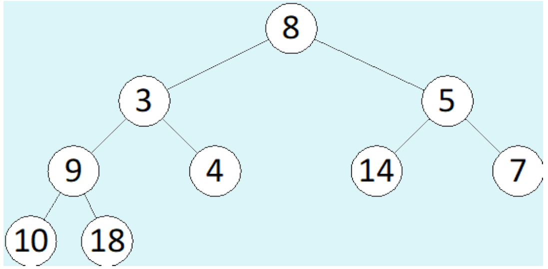


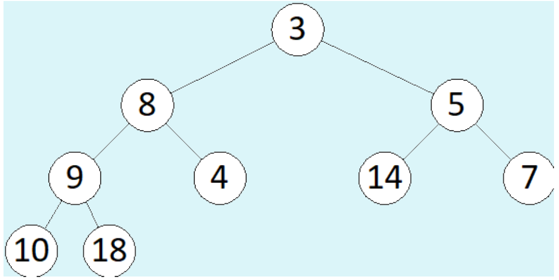
HEAPS
-- EXAMPLE OF DELETE-MIN() --

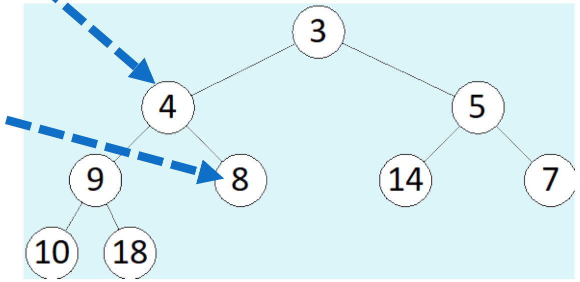
- Delete-min()
- The minimum is at the root (of value r=1)
- Replace the root value with the value of the last node

  and remove the last node
- Restore the heap
  - Swap 8 with its smaller child (3)
  - Swap 8 with its smaller child 4
  - Now 8 is a leaf: stop









HEAPS

-- TIME COMPLEXITY OF DELETE-MIN --

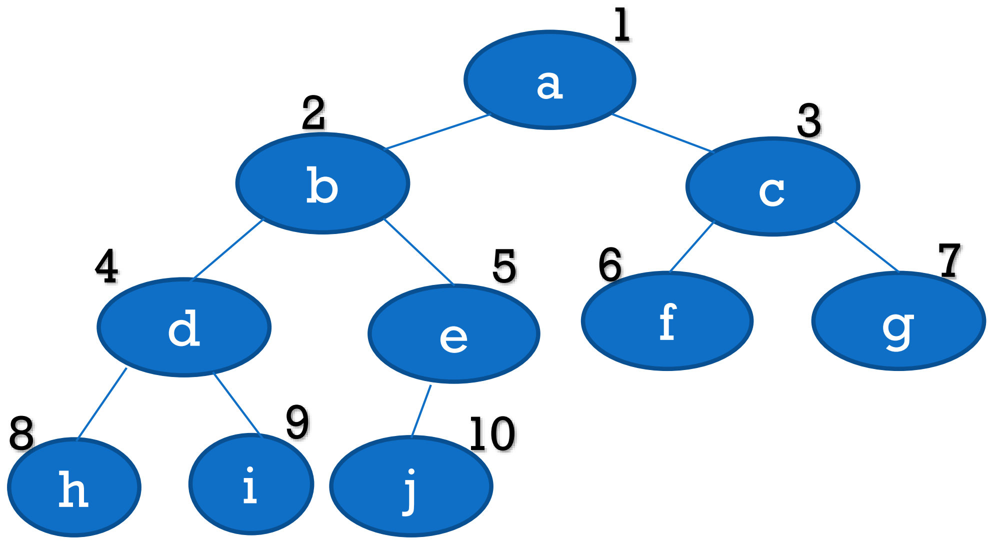
```
function delete-min(H)
begin
    x= root of H;
    r=key of x;
    remove r from node x;
    take the last node (node n), remove its key (call it b), and store b in the root;
    remove node n;
    // now restore the heap
    while (x has a key bigger than one of its children) do
        swap x with the smaller child;
        make x point to that child;
    end while
    // the while loop will stop when x becomes a leaf or ≤ both its children
    return r;
end delete-min
```

Time complexity of delete-min():

- Before the while loop, there is constant-time work;
- The while loop iterates at most the height of the tree (recall h=O(log n))
- Each iteration takes constant time (swap)
- Therefore, delete-min takes O(h)=**O(log n) time**

IMPLEMENTATION OF HEAPS WITH ARRAYS
-- STRUCTURAL CORRESPONDENCE --

- Any almost-complete trees can be stored in an array A
- Node of canonical label i is placed in entry A[i]



An almost complete binary tree;
the canonical labels are outside the nodes;
the data are inside the nodes;

i:	1	2	3	4	5	6	7	8	9	10
A[i]	a	b	c	d	e	f	g	h	i	j

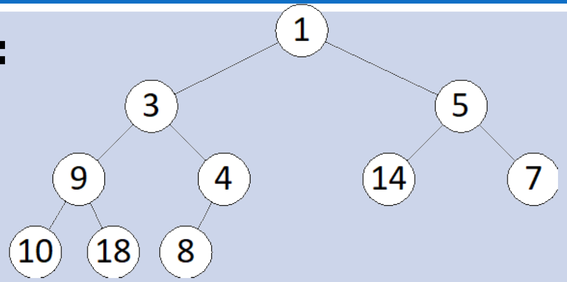
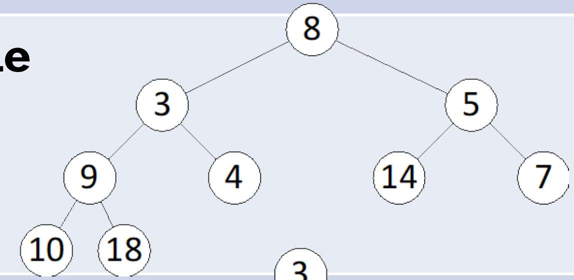
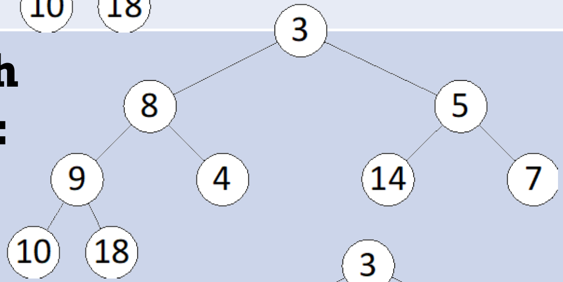
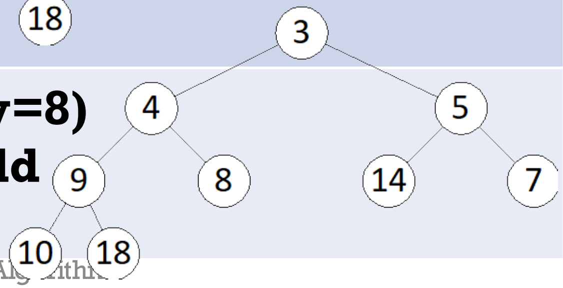
The corresponding array;
the data of node i is in A[i]

IMPLEMENTATION OF HEAPS WITH ARRAYS
-- NAVIGATION --

- Tree navigation (between parents and children, going to root, or going to last node) can be mirrored in the array
  - The left and right children of node i are $2i$ and $2i+1$, and the parent of i is $\lfloor\frac{i}{2}\rfloor$
  - Going from node i to its left/right child is like going from A[i] to A[$2i$] or A[$2i+1$]
  - Going from a node i to its parent is like going from A[i] to A[$\lfloor\frac{i}{2}\rfloor$]
  - The root is at A[1], and the last node (say node n) is at A[n]
  - Thus, for example, swapping nodes i and j is like swapping A[i] and A[j]
- Therefore, every step of the insert() and delete-min() can be expressed in terms of the array, and the time complexities stay the same, i.e., O(log n)
- So, the <u>tree</u> can be viewed as a <u>**conceptual implementation**</u>, while the <u>array</u> can be viewed as the <u>**physical implementation**</u> of the heap

HEAPS AS ARRAYS

-- ILLUSTRATION: DELETE-MIN()--

Tree View	Array View
Original Heap: 	Corresponding array:
Move last node to root: 	Move last entry to A[1]:
Swap root with smaller child: 	Swap root A[1]=8 with smaller child (A[2]=3):
Swap node 2 (key=8) with smaller child (of node label 5): 	Swap A[2]=8 with smaller child A[5]=4:

Corresponding array:

1	3	5	9	4	14	7	10	18	8

Move last entry to A[1]:

8	3	5	9	4	14	7	10	18

Swap root A[1]=8 with smaller child (A[2]=3):

3	8	5	9	4	14	7	10	18

Swap A[2]=8 with smaller child A[5]=4:

3	4	5	9	8	14	7	10	18

CREATING A HEAP FROM SCRATCH

- How long does it take to build a heap of n values from scratch:
  - One method is to call insert(…) n times on an initially empty heap
    - Time: $O(\log 1 + \log 2 + \log 3 + \cdots + \log n) = O(\log n!) = O(n \log n)$,where the last equality can be proved used Stirling's approximation

- There is an alternative (recursive) method that takes O(n) time
  - We won't cover it in this course, and so you don't need to know the algorithm for that
  - But you need to know that heaps can be constructed in O(n) time

USE OF HEAPS FOR SORTING

- You can use heaps for sorting, i.e., for re-ordering an arbitrary input array into increasing order (i.e., from the smallest to the largest)
- Method:
  1. Build the input array into a heap (in time O(n))
  2. **For** i=1 to n **do**: x=delete-min(); put x next in the output; **endfor**
- Time:
  - Step 2 takes $O(\log n + \log(n-1) + \log(n-2) \ + \cdots + \log 1) = O(\log n!) = O(n \ \log n)$
  - Therefore, total time is: $O(n) + O(n \ \log n) = O(n \log n)$

UNION-FIND DATA STRUCTURE
-- DEFINITION --

- Definition:
  - Data: n disjoint sets {1}, {2}, … , {n}, where each set has initially a single element
  - Operations:
    - **Union**: U(A,B), which unions the two input sets A and B such that after the union, the two old sets A and B are removed from the collection of sets, and replaced by the new set $C = A \cup B$.
    - **Find**: F(x), where x is an integer between 1 and n, finds the set that contains x
- Notes:
  - The unions change the collection of sets, but the find(s) do not
  - The sets in the collection are disjoint (non-overlapping) at all times

UNION-FIND DATA STRUCTURE
-- GOAL AND STRATEGY --

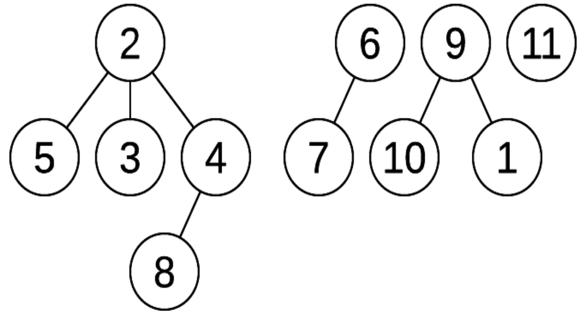
- **Goal**: to design a data structure so that O(n) calls to U and F take as little time as possible
- We will carry out the design of the data structure by having two different representations of the sets: one conceptual and one physical.
- The **conceptual representation**:
  - Each set is a rooted tree (not necessarily binary) containing the elements of that set
  - The nodes are labeled with the elements of the corresponding set
  - As new sets are born (from Union), we need an automated naming system
- The physical representation will be derived a little later

UNION-FIND DATA STRUCTURE
-- EXAMPLE OF TREE REPRESENTATION--

- Suppose we have 11 elements: 1, 2, 3, … , 11
- Suppose after a few unions, the collection of sets is:

  {2,3,4,5,8}, {6,7}, {1,9,10}, and {11}

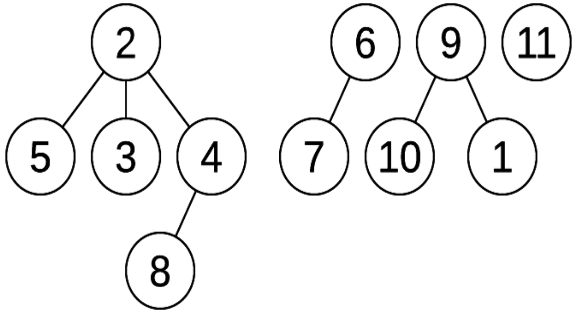
- The tree representation of the data structure can be:
  - One tree per set: the tree contains the elements of its set
  - We **<u>don't care</u>** about the structure of each tree
  - But we care what elements are in each tree



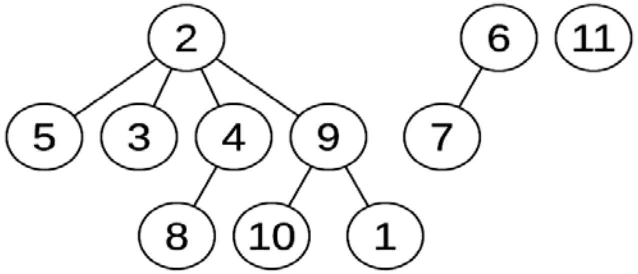
- We need a set-naming mechanism that gives a unique name to each set, including to new sets that emerge out of Union
- **Naming scheme**: Let the <u>root label</u> <u>double</u> as the <u>label for that set</u>

UNION-FIND DATA STRUCTURE
-- FIRST IMPLEMENTATION (UNION) --

- U(A,B) can be done by "a single stroke"
  - Make the root of A to be the parent of the root of B
  - Note that the trees of A and B stop existing separately, and are replaced by the new tree, which is what we want
- Example:
  - Do U(2,9) when the collection is:

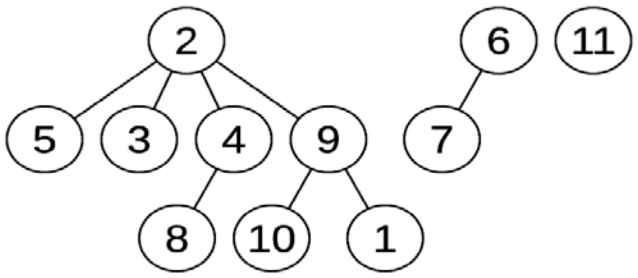


  - This unions the tree rooted at 2 with the tree rooted at 9, which is like, U({2,3,4,5,8},{1,9,10})
  - The result, derived by making 2 the parent of 9:



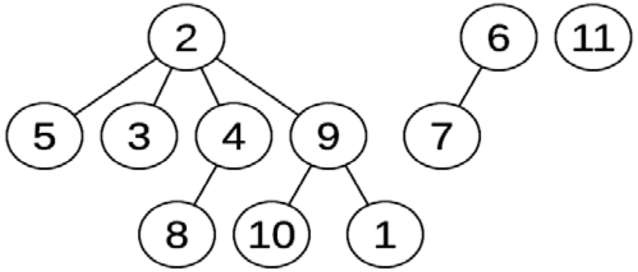
UNION-FIND DATA STRUCTURE
-- FIRST IMPLEMENTATION (FIND) --

- F(x) needs to return the name of the set containing element (int) x
- The name of that set is the label of the root of the corresponding tree
- We can find that root by:
  - Moving up from x to its parent, and from that to its parent, and so on until we get to the root, which has no parent
  - Return that root.
- Example: F(10)
  - Parent of 10 is 9
  - Parent of 9 is 2
  - Parent of 2 doesn't exit => 2 is the root => return 2 (which means that the set that contains 10 is set 2)



UNION-FIND DATA STRUCTURE
-- FIRST PHYSICAL IMPLEMENTATION --

- We can implement the collections of trees (we call it **forest**) by using general tree representations (using node records and pointers)
- But there is a better, cheaper representation, which we'll derive next
- Note that in both the Union and Find that we just did, we only needed to refer to **parents** of nodes (never to children), and to know which is root
- So, if we use a physical representation that stores the parent of each node and that signals which nodes are root, that representation is adequate for implementing U and F



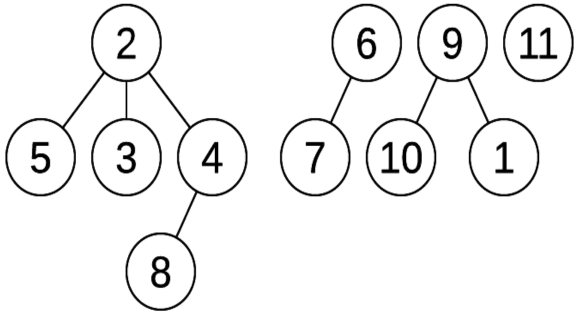
- Answer: a single array PARENT[1:n] where
  - PARENT[i] stores the parent of node i
  - If i is a root, set PARENT[i]=0 (or any number other than 1,2, …, n)

UNION-FIND DATA STRUCTURE

-- FIRST PHYSICAL IMPLEMENTATION: PARENT ARRAY --

- PARENT array of this collection:

i	1	2	3	4	5	6	7	8	9	10	11
PARENT	9	0	2	2	2	0	6	4	0	9	0



- Note: at the beginning, PARENT[i]=0 for all i, because each set is a single node, and so, that node is root.
- Implementation of U and F using PARENT:

```
Procedure U(i,j)
Begin
    PARENT[j]=i;
End U
```

- Time: O(1)

```
Function F(x)
begin
    int r=x;
    while PARENT[r] > 0) do
        r = PARENT[r];
    endwhile // now r is a root
    return (r);
end
```

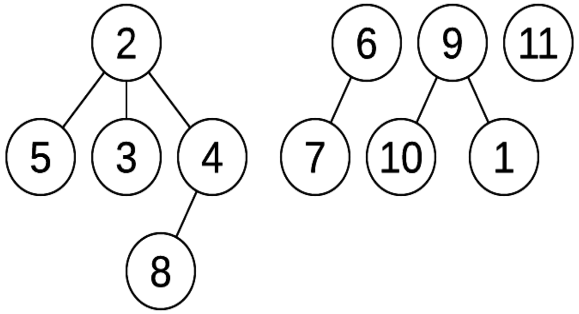
- Time: O(h)
- h= height of tree

UNION-FIND DATA STRUCTURE

-- FIRST PHYSICAL IMPLEMENTATION: U(2,9) AND F(10) --

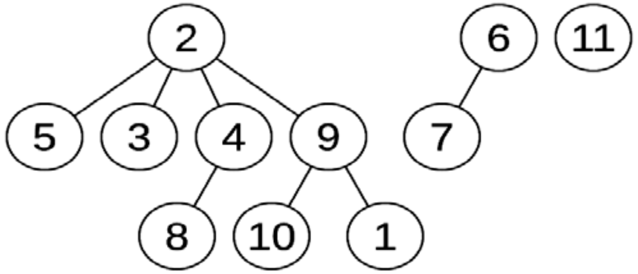
- PARENT array of this collection before U(2,9):

i	1	2	3	4	5	6	7	8	9	10	11
PARENT	9	0	2	2	2	0	6	4	0	9	0



- U(2,9): PARENT[9]=2, which results in this array:

i	1	2	3	4	5	6	7	8	9	10	11
PARENT	9	0	2	2	2	0	6	4	**2**	9	0



- F(10):
  - PARENT[10] == 9 → PARENT[9] == 2 → PARENT[2]==0 → 2 is the root
  - Therefore, the set returned by F(10) is set 2, which is correct

UNION-FIND DATA STRUCTURE

-- 1^{ST} IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--

- Each Union takes O(1) time, so O(n) U's take O(n) time
- Each Find takes O(h), but how bad can h be?
  - Answer: it can be as bad as O(n), which makes O(n) calls to F take $O(n^2)$ time
  - Proof:
    - Take this sequence of calls: U(2,1), U(3,2), U(4,3), … , U(n, n-1), F(1), F(2), … , F(n)
    - The calls to U create a single-path tree: n, n-1, n-2, … , 2, 1 (prove that to yourself)
    - The depth of node (i) is n-i, for all i
    - Thus, each F(i) takes O(n-i) time
    - Therefore, the n calls to F take: $O(1+2+\ldots+(n-1))=O(n(n-1)/2)=O(n^2)$
- $O(n^2)$ can be quite costly: check if n = 1 Mil, on a computer that executes 1MFLOP (1 million operations/second), what is $O(n^2)$ be in real time?

UNION-FIND DATA STRUCTURE
-- SECOND IMPLEMENTATION --

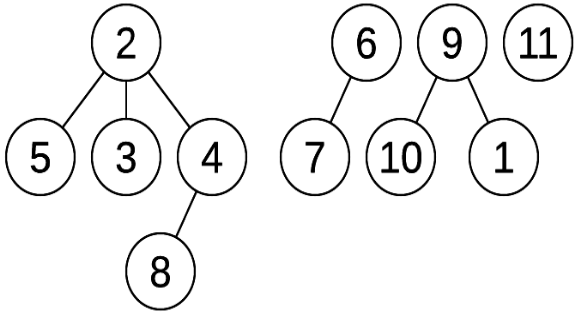
- **Issue**: the reason we could get such long thin trees is
  - U(i,j) makes i the parent of j regardless of how small tree i is
- **Remedy**: Make the **root** of the bigger tree the **parent** of the other root
- **Issue**: This requires that we compute (or keep track of) the size of each tree
- **Remedy**: If i is a root, let PARENT[i] store the number of nodes in tree rooted at i
- **Issue**: If PARENT[i]==3, Is 3 the parent of i or # nodes in tree rooted at i?
- **Remedy**: For root i, make $\text{PARENT}[i] = -(\textbf{number of nodes in tree of } \boldsymbol{i})$
- **Issue**: How to efficiently update tree size while doing unions?
- **Remedy**: When making i parent of j, the new tree of i has the sum of nodes of the two old trees: PARENT[i]:=PARENT[i]+PARENT[j], which takes O(1) time!!

UNION-FIND DATA STRUCTURE

-- 2ND IMPLEMENTATION (UNION) --

- PARENT array of this collection:

i	1	2	3	4	5	6	7	8	9	10	11
PARENT	9	**-5**	2	2	2	**-2**	6	4	**-3**	9	**-1**



- At the start, PARENT$[i] = -1\ \forall i$, why?
- Implementation of U:
- How about Find F: same as before
- Time of U: O(1)

```
Procedure U(i,j)
Begin
    if |PARENT[i]| >= |PARENT[j]| then
        PARENT[i]=PARENT[i]+PARENT[j];
        PARENT[j]=i;
    else
        PARENT[j]=PARENT[i]+PARENT[j];
        PARENT[i]=j;
    endif
End U
```

UNION-FIND DATA STRUCTURE

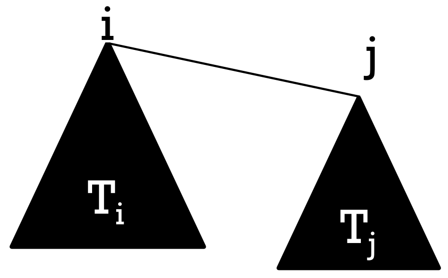
-- 2ND IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--

- Each Union takes O(1) time, so O(n) U's take O(n) time
- Each Find F(x) takes O(h), but how bad can h be?
  - **Theorem**: $h_x \leq \log N_x$ where h_x and N_x are the height and # nodes in the tree containing x
  - Proof: next slide

UNION-FIND DATA STRUCTURE

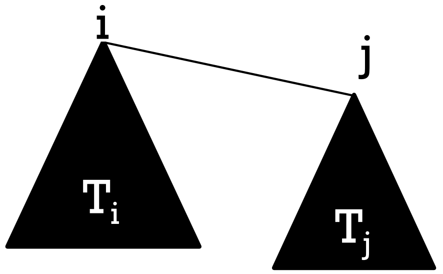
-- 2^{ND} IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--

- Theorem: $h_x \leq \log N_x$ for all x.
- Proof: By induction on the number of U's that created the tree of x (call it T_x)
  - Call m that number of calls to U
  - Basis: m=0. Then T_x is a 1-node tree, i.e., $N_x=1$ and $h_x=0$. Since $\log N_x=\log 1=0$, it follows that $h_x=\log N_x$ and thus $h_x \leq \log N_x$ in the basis case.
  - Induction: Assume that $h_y \leq \log N_y$ for all trees created after m-1 calls to U, and let T_x be in a tree created from m calls to U. Prove that $h_x \leq \log N_x$.
  - Suppose the m^{th} call to U is U(i,j), and let T_i and T_j be the trees rooted at i and j before that call to U.
  - Those two trees were created by at most m-1 calls to U, so by the induction hypothesis, $h_i \leq \log N_i$ and $h_j \leq \log N_j$



CONTINUATION OF THEOREM PROOF

-- 2ND IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--



- Proof continuation:
  - Recall that T_x is the whole tree shown to the right
  - $N_x = N_i + N_j$
  - We're assuming without loss of generality that $N_i \geq N_j$
  - $h_x = \max(h_i, 1 + h_j) \leq \max(\log N_i, 1 + \log N_j) = \max(\log N_i, \log 2 + \log N_j)$
  - $h_x \leq \max(\log N_i, \log 2 + \log N_j) = \max(\log N_i, \log(2N_j))$
  - Now, $2N_j = N_j + N_j \leq N_i + N_j = N_x \Rightarrow \log(2N_j) \leq \log N_x$
  - Also, $N_i \leq N_i + N_j = N_x \Rightarrow \log N_i \leq \log N_x$
  - From the previous 3 bullets, we get: $h_x \leq \max(\log N_i, \log(2N_j)) \leq \log N_x$
  - Therefore, $h_x \leq \log N_x$, which is what we needed to prove. Q.E.D.

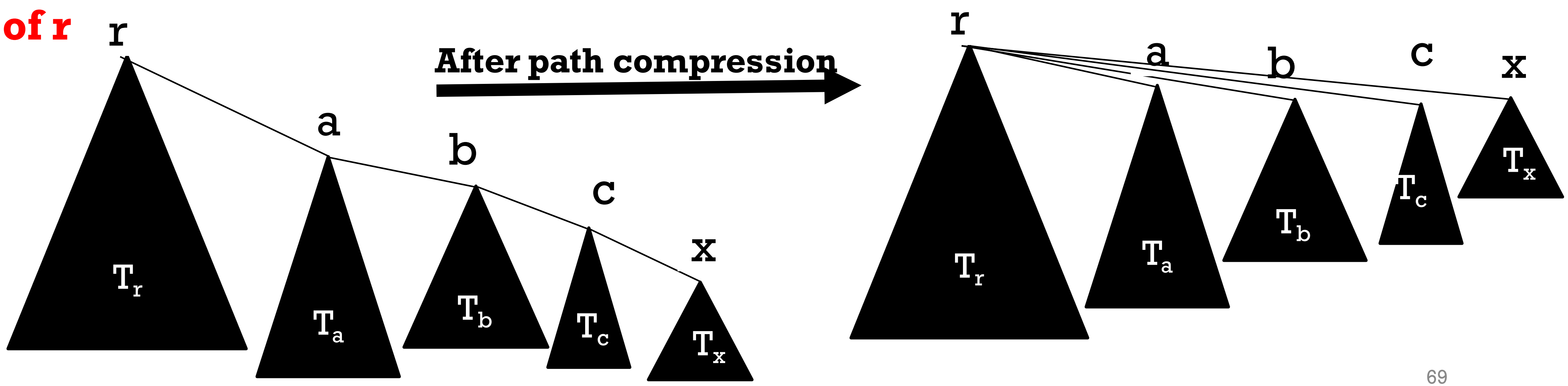
UNION-FIND DATA STRUCTURE

-- 2^{ND} IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--

- Each Union takes O(1) time, so O(n) U's take O(n) time
- Each Find F(x) takes $O(h_x) = O(\log N_x) = O(\log n)$
- Therefore, O(n) F's take $O(n \log n)$ time
- Conclusion: O(n) calls to U and F take O(n + n log n)=O(n log n) time
- That is much better than $O(n^2)$ time

UNION-FIND DATA STRUCTURE
-- THIRD IMPLEMENTATION: PATH COMPRESSION --

- Can we do better?
- Yes: Keep U as in the 2nd implementation, but speed up F
- How? Path Compression
  - Call F(x) tracing the path from x to the root (call it r)
  - Trace that path again, making each node along the way an **immediate child of r**



UNION-FIND DATA STRUCTURE

-- 3RD IMPLEMENTATION: PSEUDOCODE OF FIND --

```
Function F(x)
begin
        int r,s,t;
        r=x;
        while PARENT[r] > 0 do r = PARENT[r];endwhile
        //now r is a root
        s=x;    // s will trace the path from x to the root r
        while s != r do
                t := s;   // t records the current value of s before s steps to its parent
                s := PARENT[s];
                PARENT[t] := r;        // make t an immediate child of root r
        endwhile
        return (r);
End F
```

UNION-FIND DATA STRUCTURE

-- 3RD IMPLEMENTATION TIME COMPLEXITY OF O(N) CALLS TO U AND F--

- **Theorem**: Every O(n) sequence of calls to U's and F's take O(n G(n)) time where $G(n) \leq 5 \; \forall \; n \leq 2^{65000}$.
- We won't give a proof for that.
- So, for all practical values of n, $G(n) \leq 5$, and so practically the sequence of calls takes O(n) time.