Data Structure

http://smarlead.hallym.ac.kr

Instructor: Jin Kim
010-6267-8189(033-248-2318)
jinkim@hallym.ac.kr

Office Hours:

Puppy(귀여운 강아지)

늠름한 허스키

Non Linear Data Structure

- Data structure we will consider this semister:
  - Tree
  - Binary Search Tree
  - Graph
  - Weighted Graph
  - Sorting
  - Balanced Search Tree

Binary Search Trees: Outline

- Binary Search Tree (이진 탐색 트리)
- Heap(히프)
- Selection Tree(선택 트리)

Binary Search Trees (1/8)

- Why do binary search trees need?이진탐색트리가 필요한 이유
  - Searching an element is O(logn) 탐색시간이 O(logn) 빠르다
- **Definition** of binary search tree:이진트리정의
  - Every element has a unique key모든원소는 유일한 키
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree왼쪽서브트리는 루트보다작고, 오른쪽서브트리는 루트보다크다.
  - The left and right subtrees are also binary search trees왼쪽서브트리와 오른쪽 서브트리역시이진탐색트리

Binary Search Trees (2/8)

- Example: (b) and (c) are binary search trees

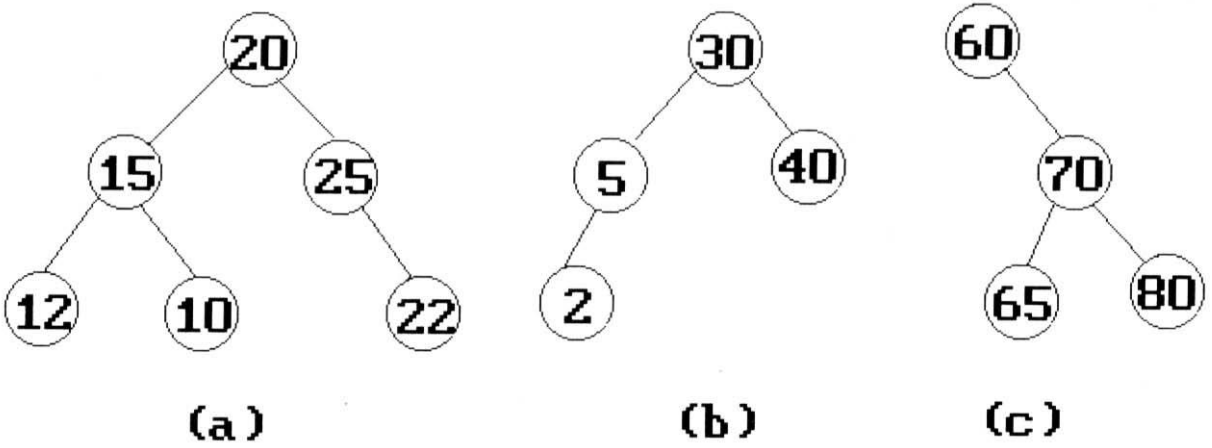
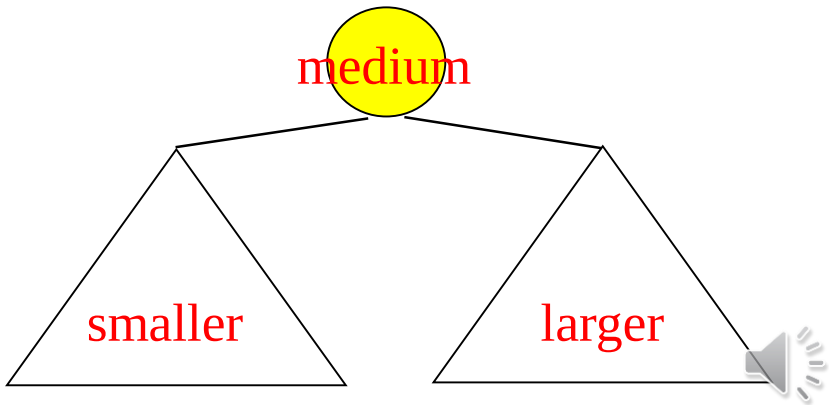
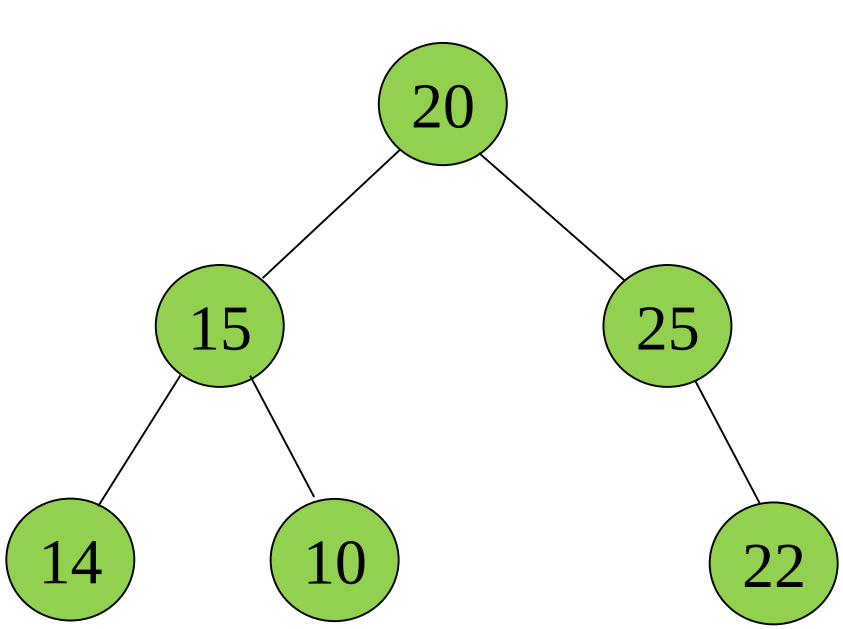
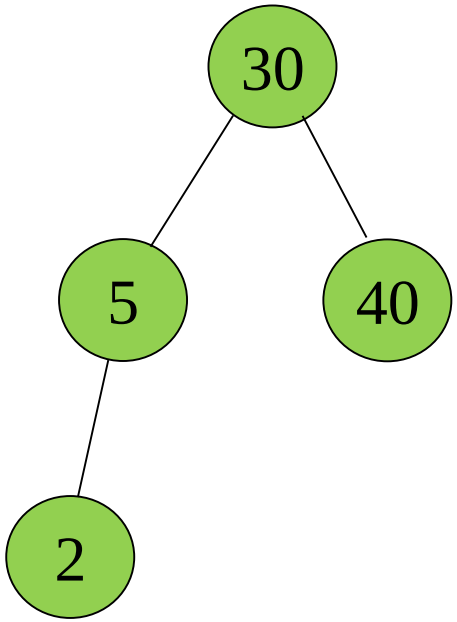


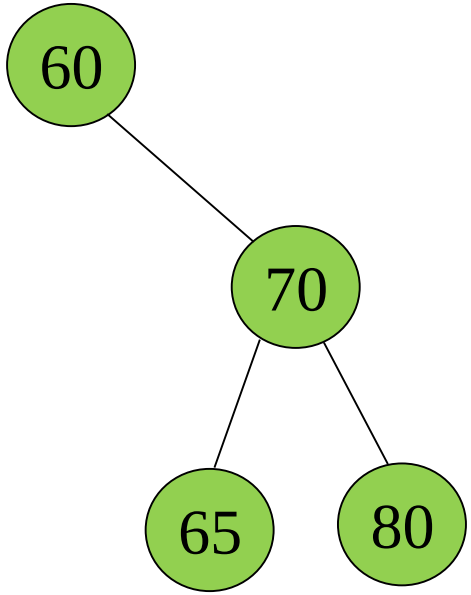
Figure 5.30: Binary trees



Binary Search Trees



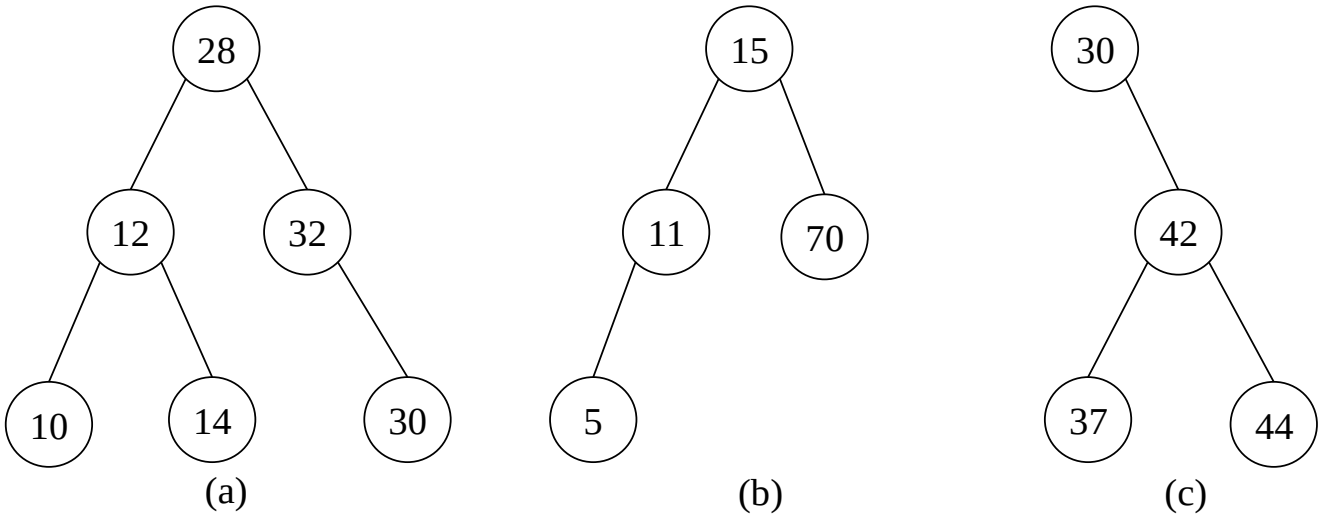




Not binary search tree이진탐색트리아님

Binary search trees 이진탐색트리임

- (a) : not bt아님
- (b),(c) : binary search tree 맞음



Deleting from a Binary Search Tree

(BST)원 소 탐 색

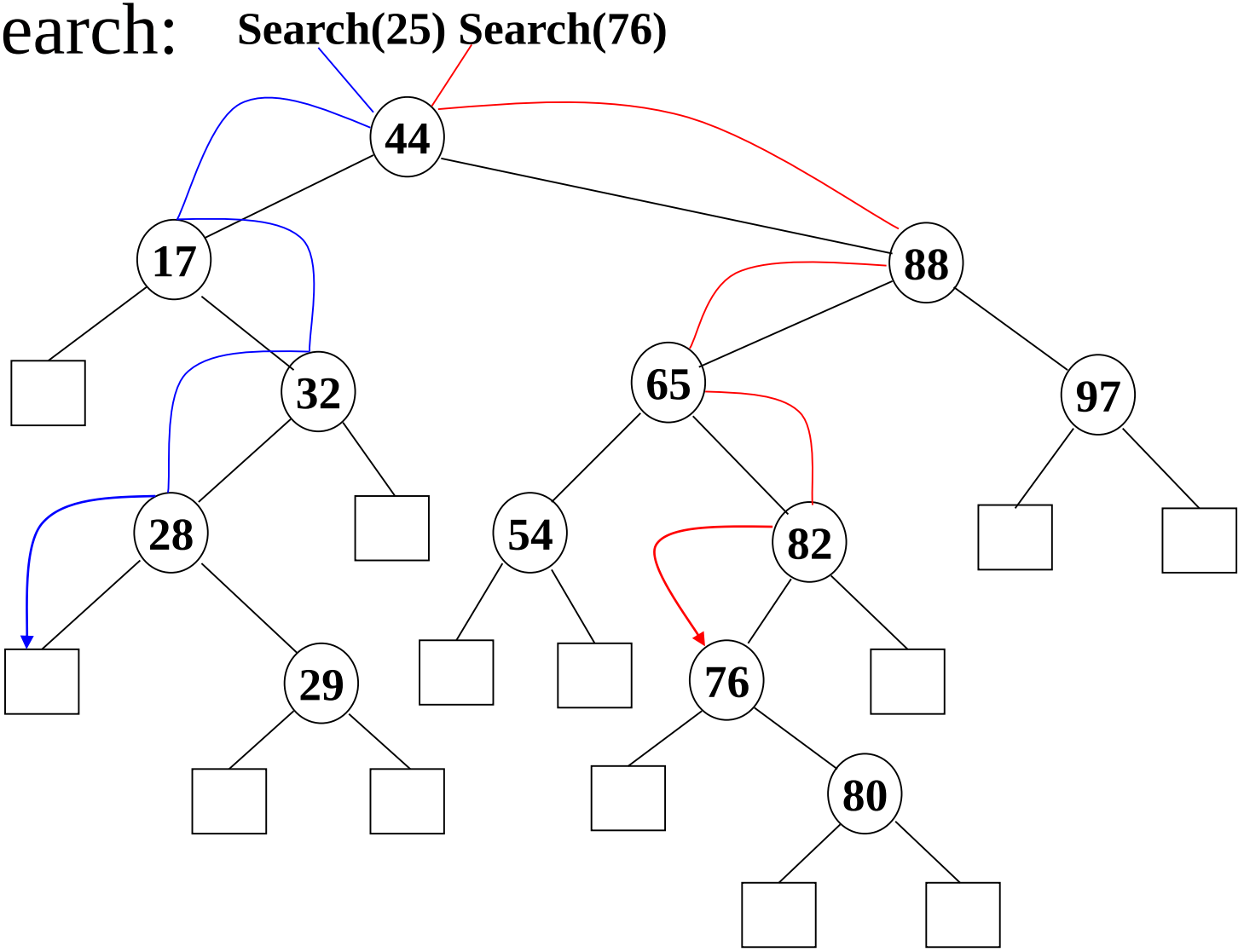
Searching A Binary Search Tree

이진탐색트리 탐색

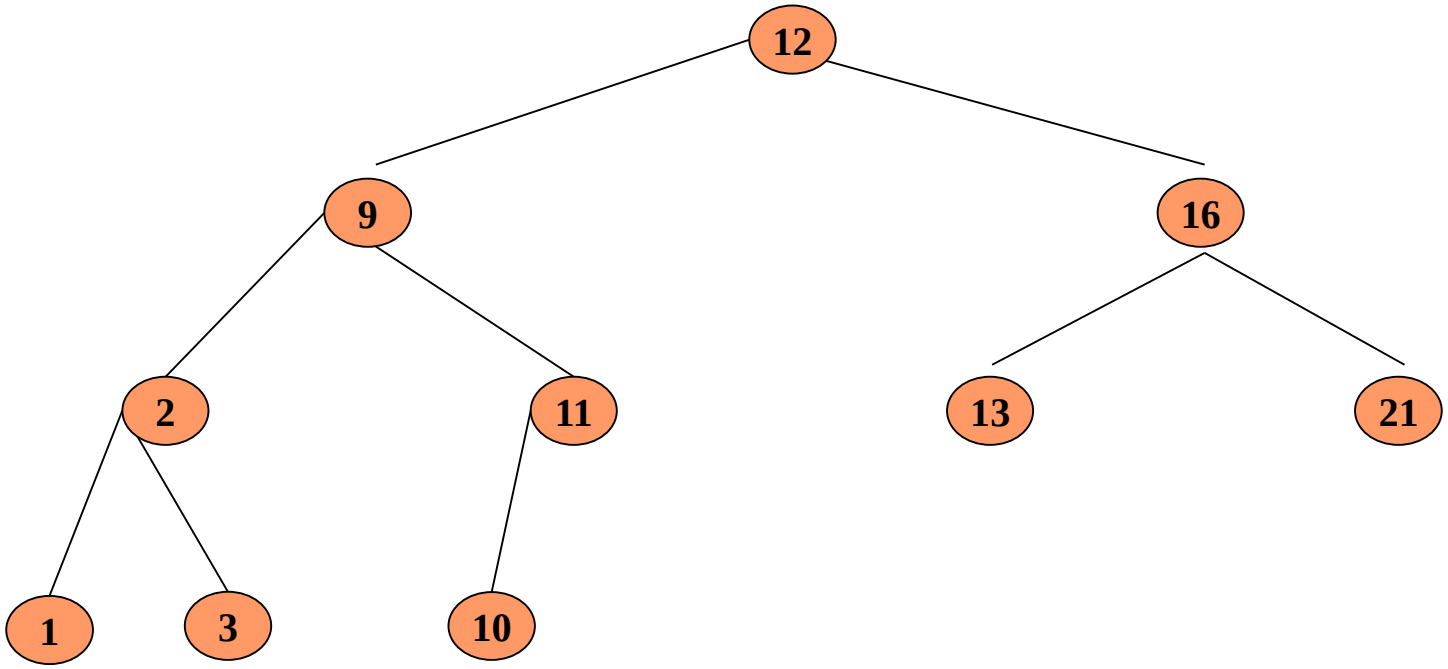
- Always search starts from the root.(트리는 항상루트부터 시작)
- If the root is null, then this is an empty tree. No search is needed.
- If the root is not null, compare the x(탐색원소) with the key of root.
  - If x equals to the key of the root, then it's done. 키가발견되면 탐색끝
  - If x is less than the key of the root, then no elements in the right subtree can have key value x. We only need to search the left tree. 탐색원소가 루트의 키보다 작으면 왼쪽서브트리로
  - If x larger than the key of the root, only the right subtree is to be searched. 탐색원소가 루트의키보다크면, 오른쪽서브트리로이동

Binary Search Trees (3/8)

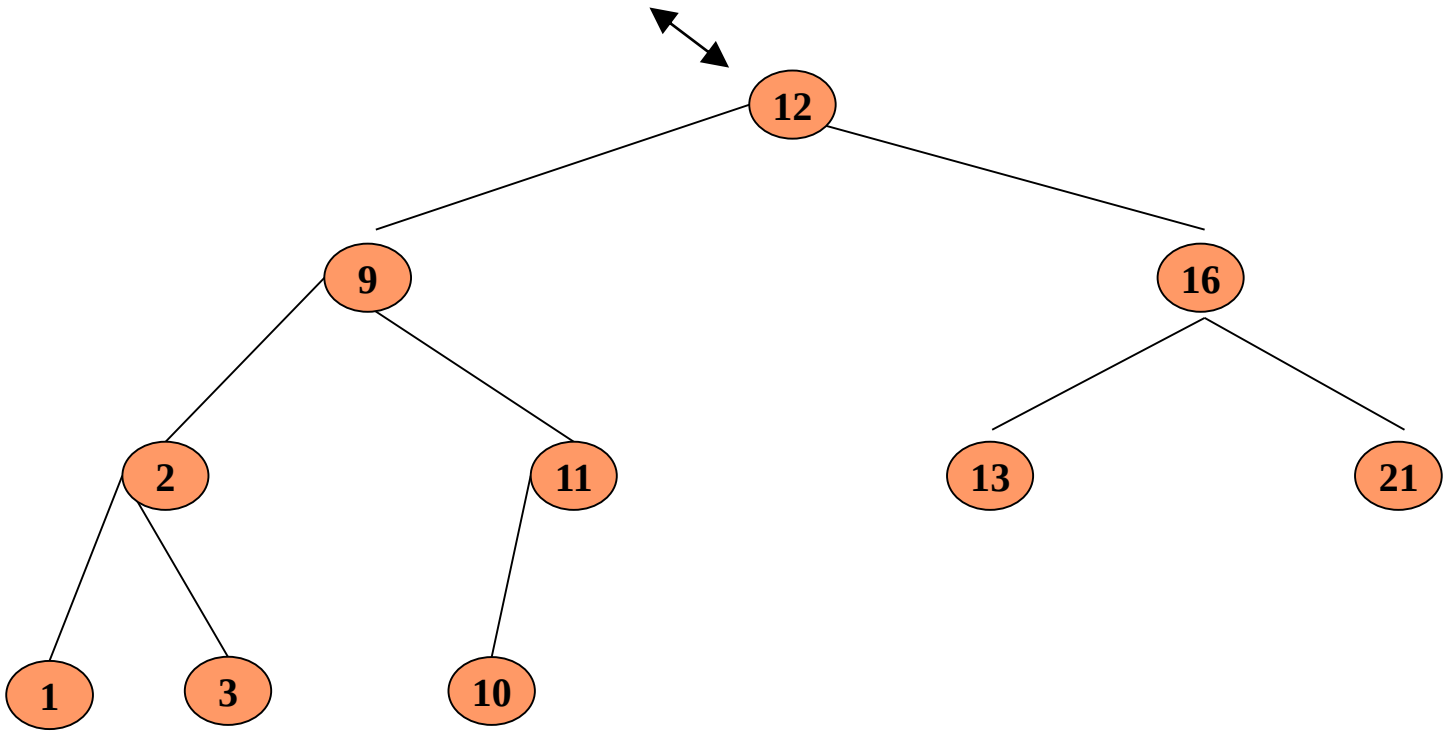
- Search:



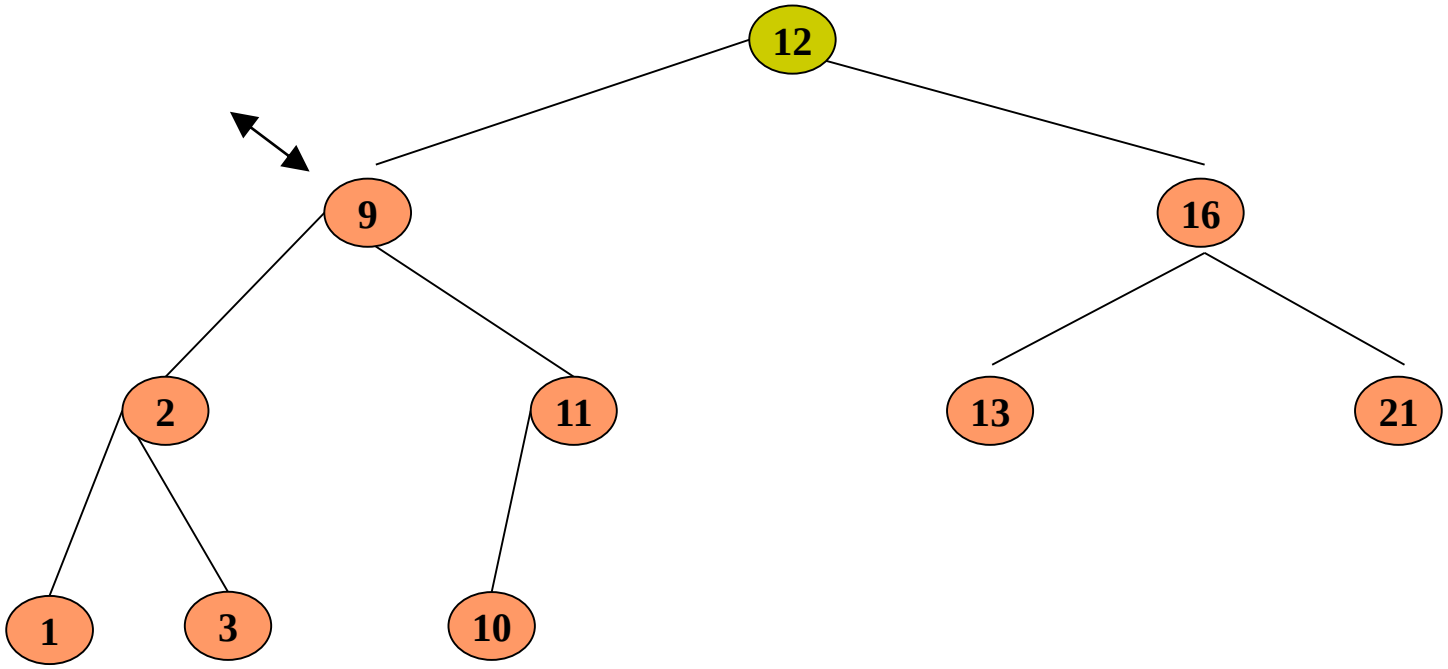
search(10)



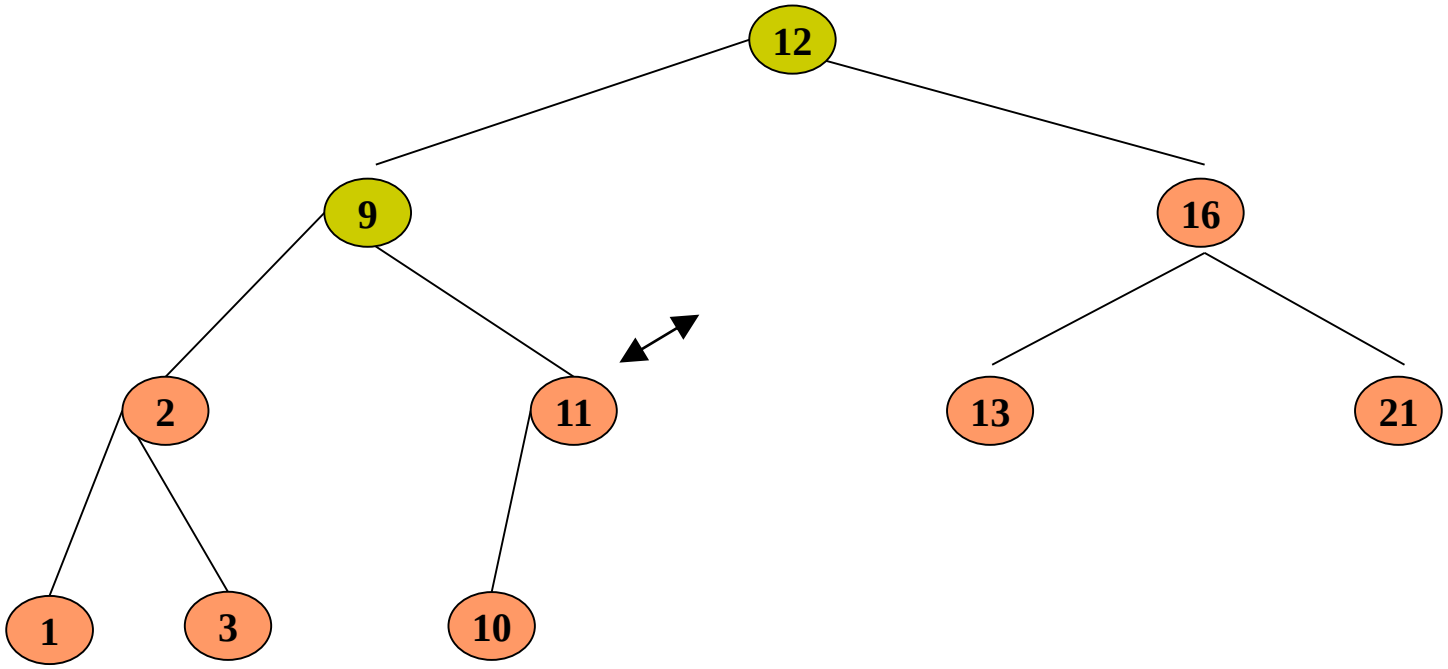
search(10)



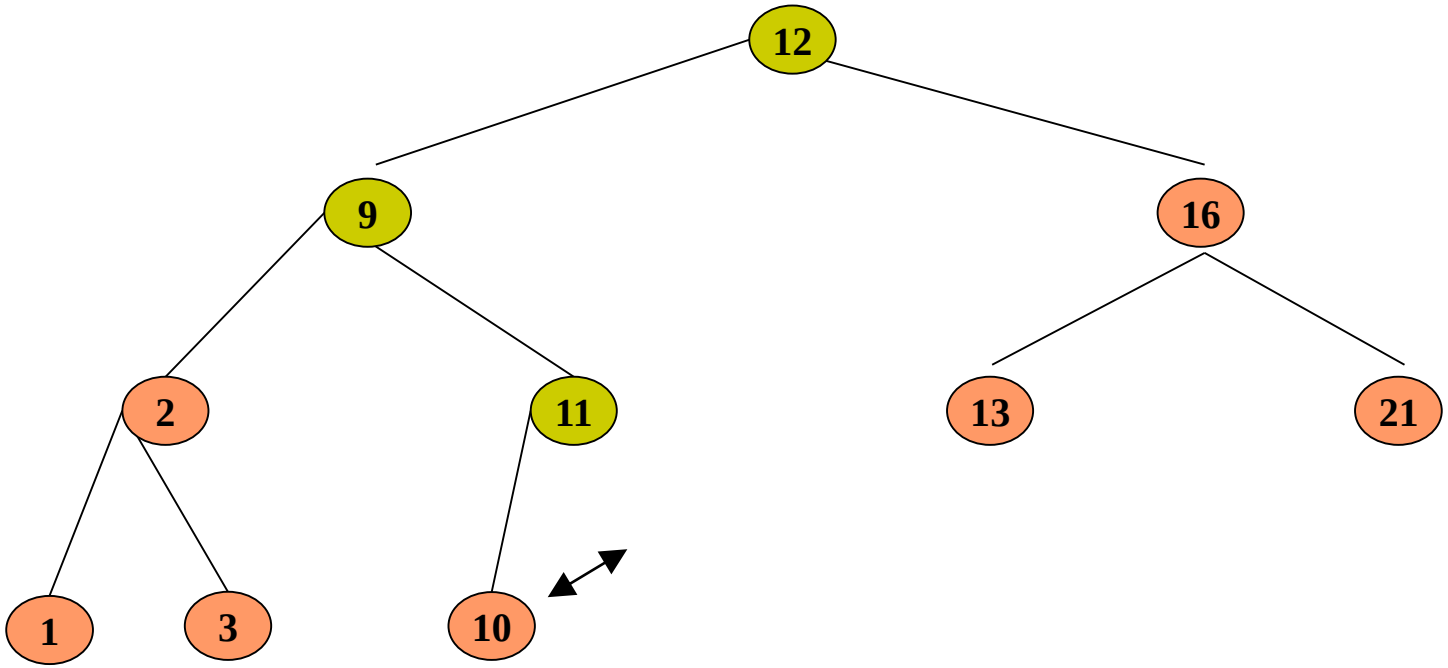
search(10)



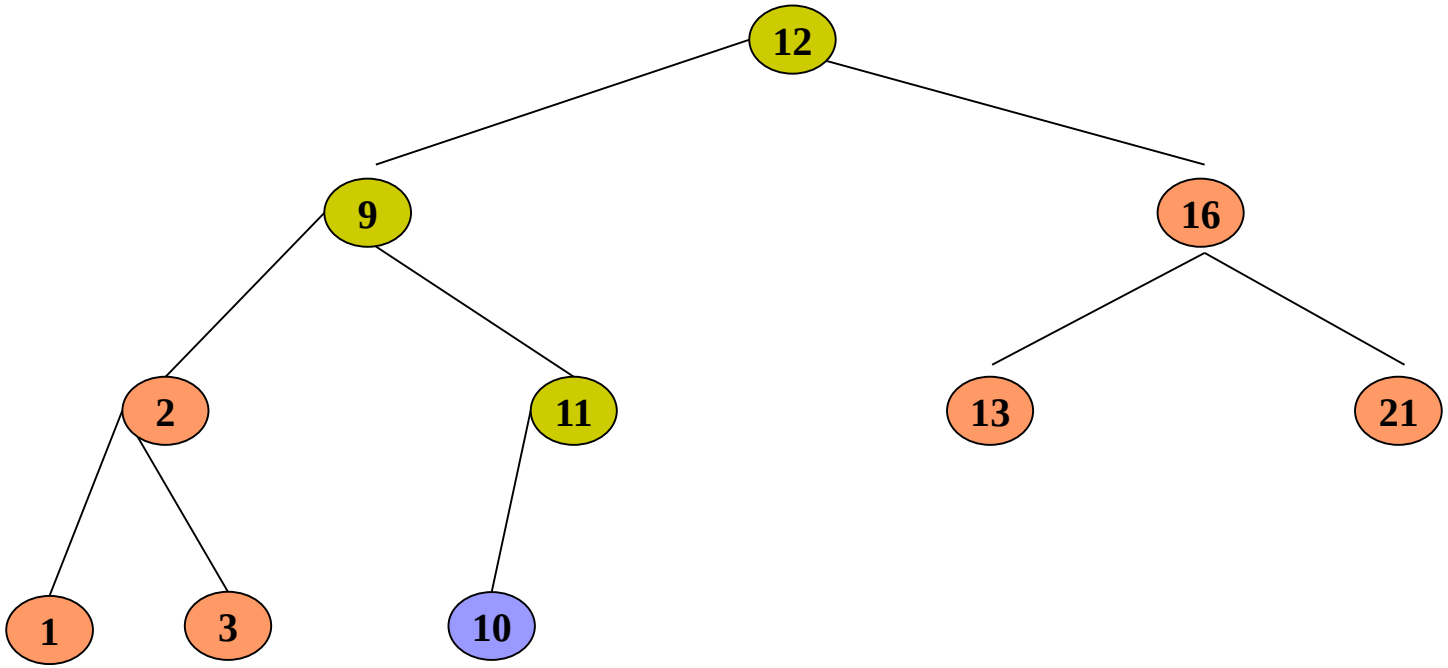
search(10)



search(10)



search(10)



Binary Search Trees (4/8)

- Searching a binary search tree

Node structure

left	key	right

```
searchBST(B, x)
        // B는 이원 탐색 트리
        // x는 탐색 키 값
      p ← B;
      if (p = null) then  // 공백 이진 트리로 실패
          return null;
      if (p.key = x) then            // 탐색 성공
          return p;
      if (p.key < x) then            //  오른쪽 서브트리 탐색
          return searchBST(p.right, x);
      else return searchBST(p.left, x);     // 왼쪽 서브트리 탐색
end searchBST()
```

Deleting from a Binary Search Tree

(BST)원소삽입

Insertion To A Binary Search Tree

이진탐색트리에 원소 삽입

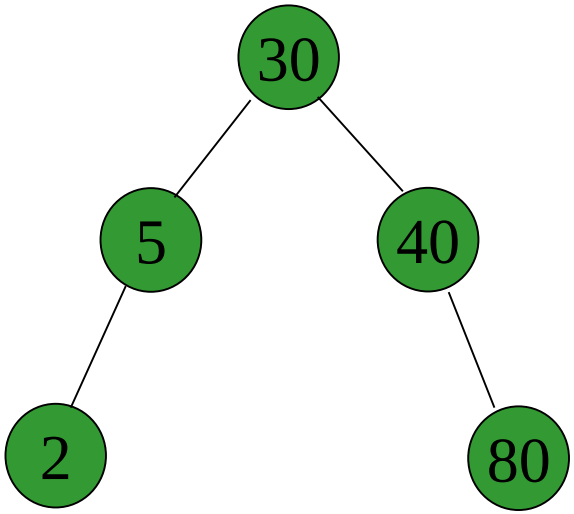
- Before insertion is performed, a search must be done to make sure that the value to be inserted is not already in the tree. (삽입하기 위해서는 탐색먼저수행)
- If the search fails, then we know the value is not in the tree. So it can be inserted into the tree. 키탐색에 실패하면 실패한 곳에 원소삽입
- It takes O(h) to insert a node to a binary search tree. h is the height of bst. 원소삽입에 소요되는시간은 탐색트리의 높이에 비례

Binary Search Trees 키 삽입

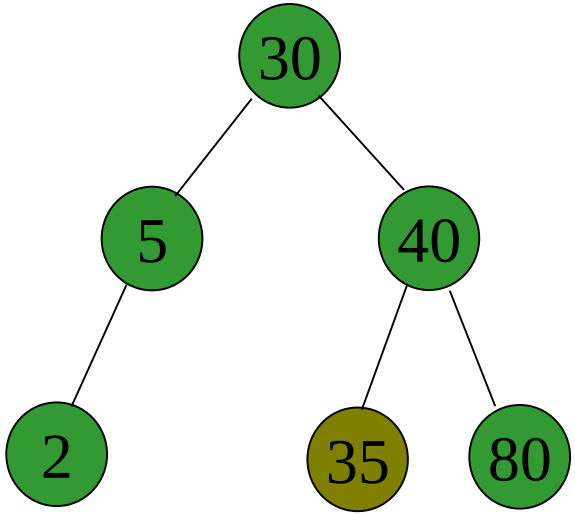
- Inserting into a binary search tree

```
insertBST(B, x)
      // B는 이원 탐색 트리, x는 삽입할 원소 키 값
    p ← B;
    while (p ≠ null) do {
                  // 삽입하려는 키 값을 가진 노드가 이미 있는지 검사
       if (x = p.key) then return;   //키가 있으면 끝
       q ← p;  // q는 p의 부모 노드를 지시
       if (x < p.key) then p ← p.left;
       else p ← p.right;
    }
    newNode ← getNode();          // 삽입할 노드를 만듦
    newNode.key ← x;
    newNode.right ← null;
    newNode.left ← null;
    if (B = null) then B ← newNode;          // 공백 이원 탐색 트리인 경우
    else if (x < q.key) then          // q는 탐색이 실패로 종료하게 된 원소
         q.left ← newNode;
    else
       q.right ← newNode;
    return;
end insertBST()
```

Inserting Into A Binary Search Tree

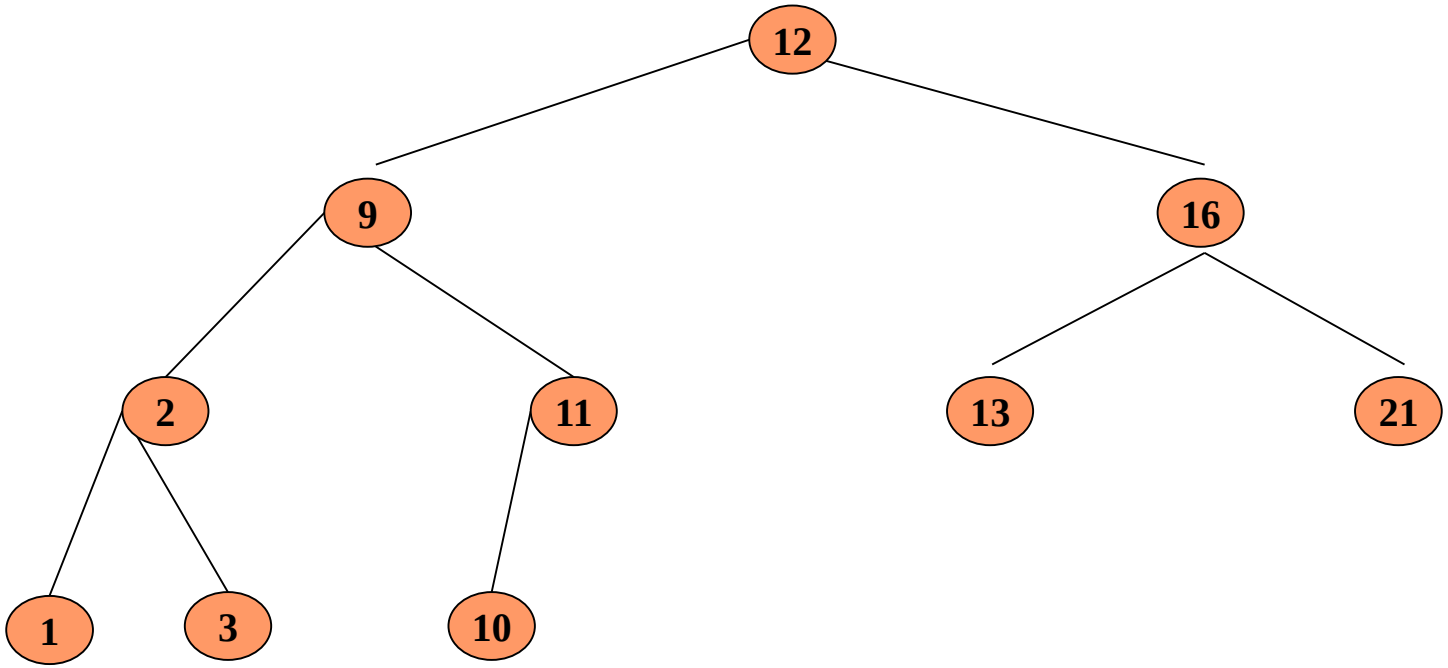


Insert(80)

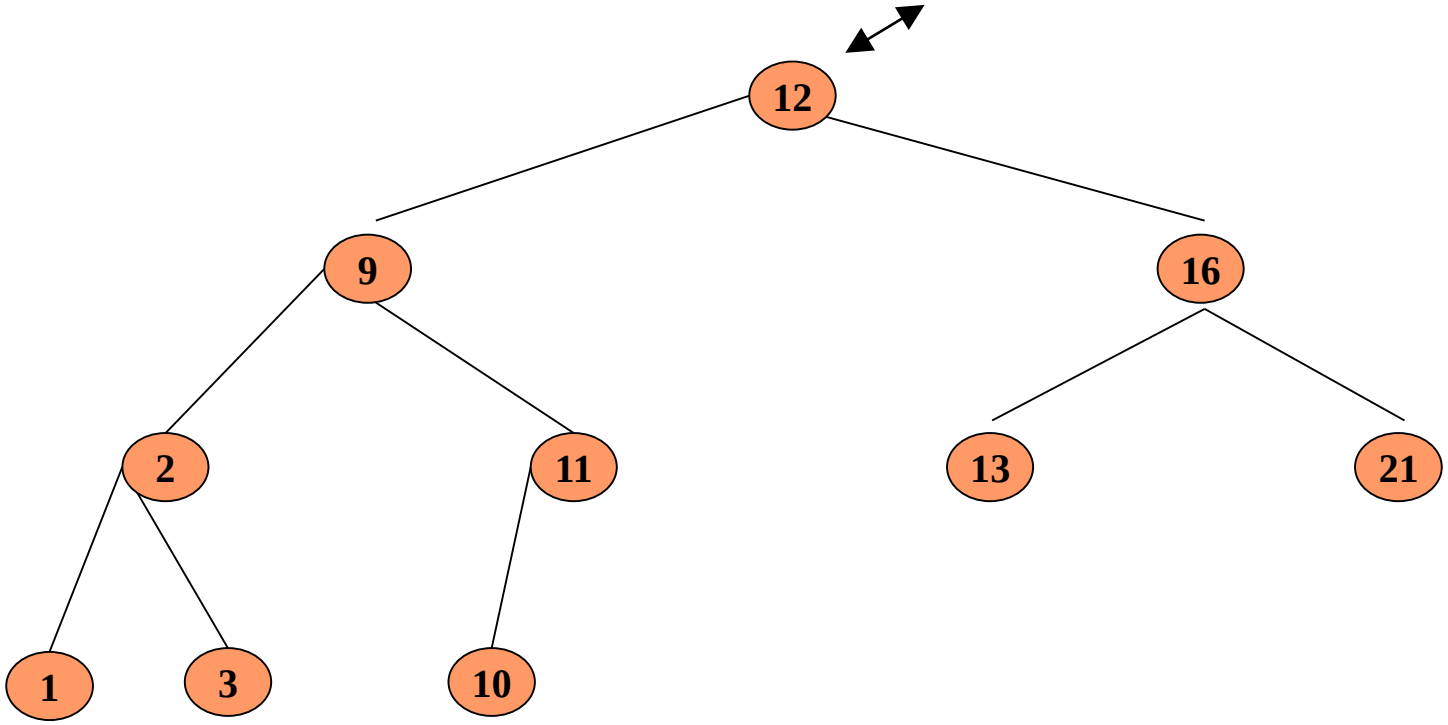


Insert(35)

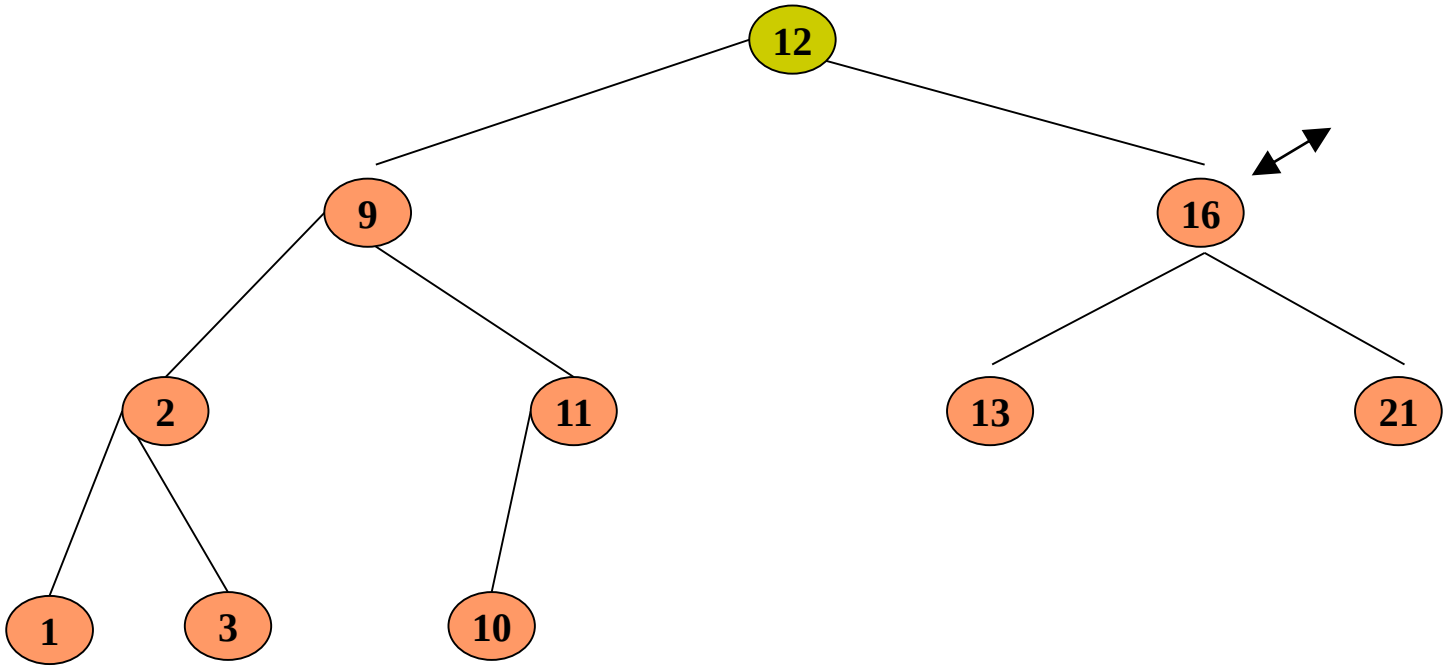
insert(14)



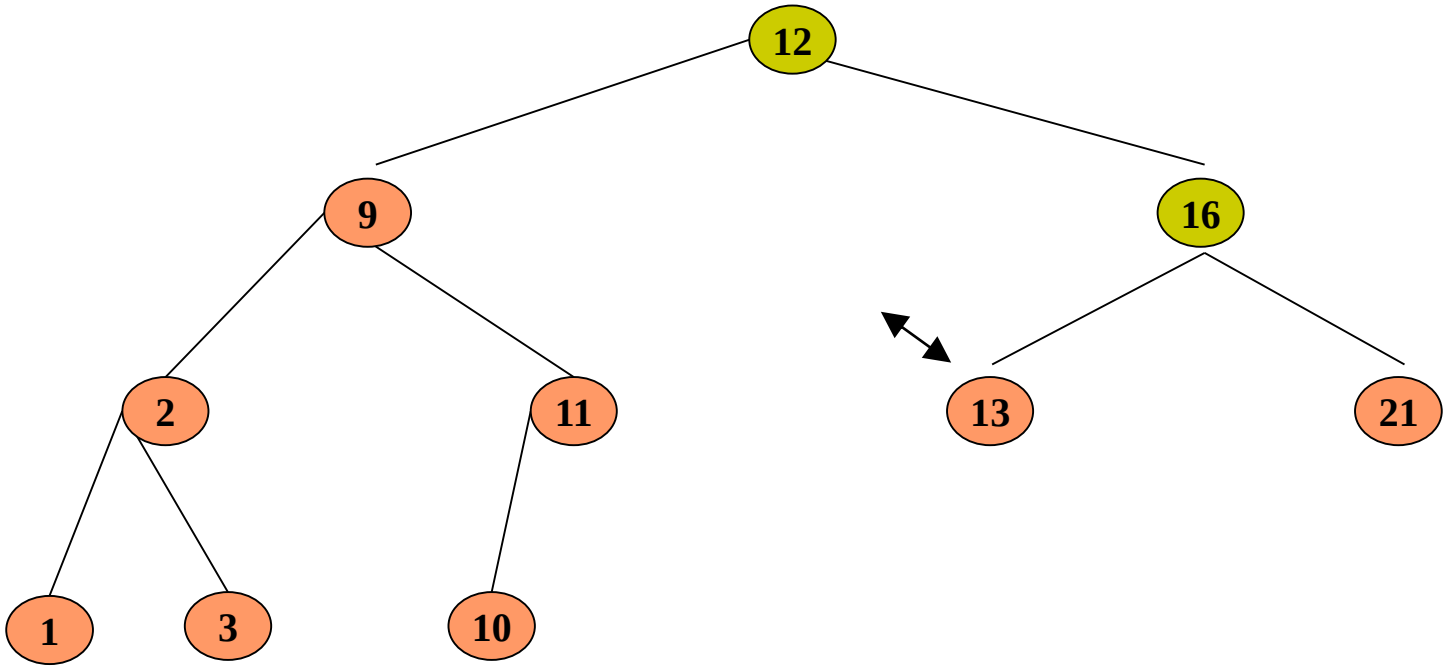
insert(14)



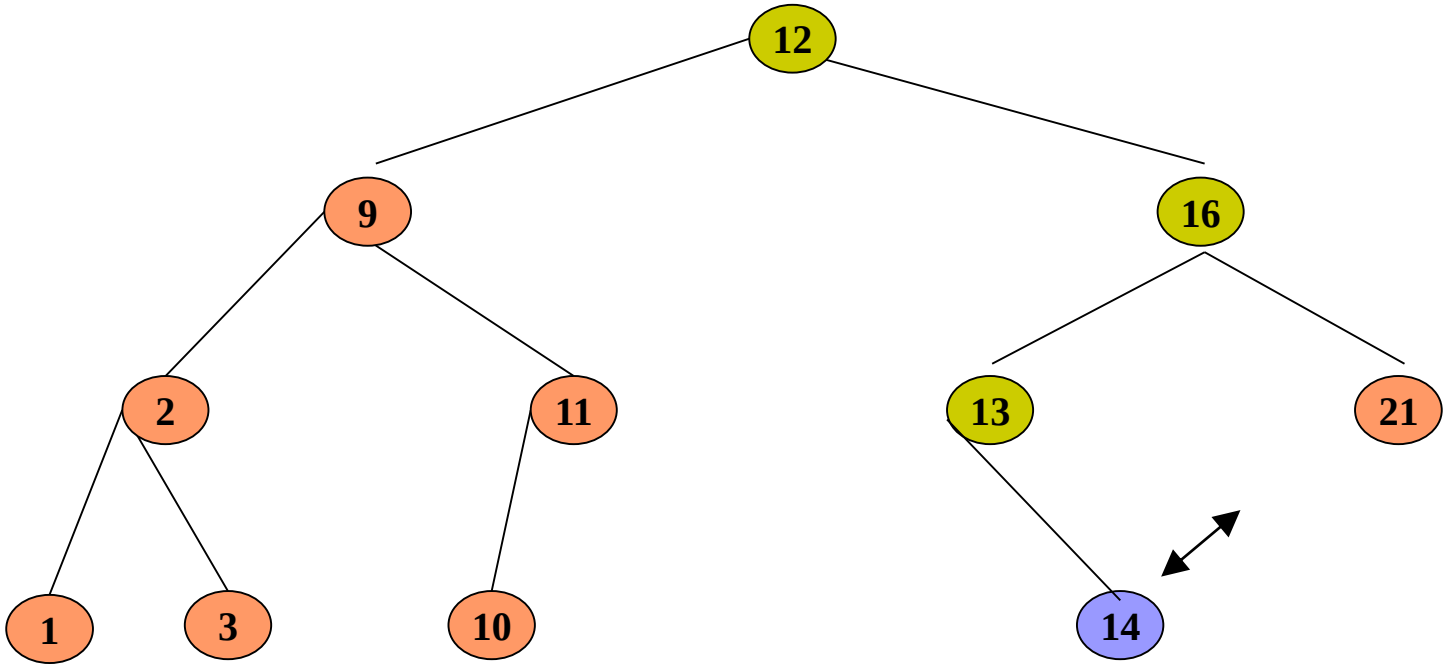
insert(14)



insert(14)



insert(14)

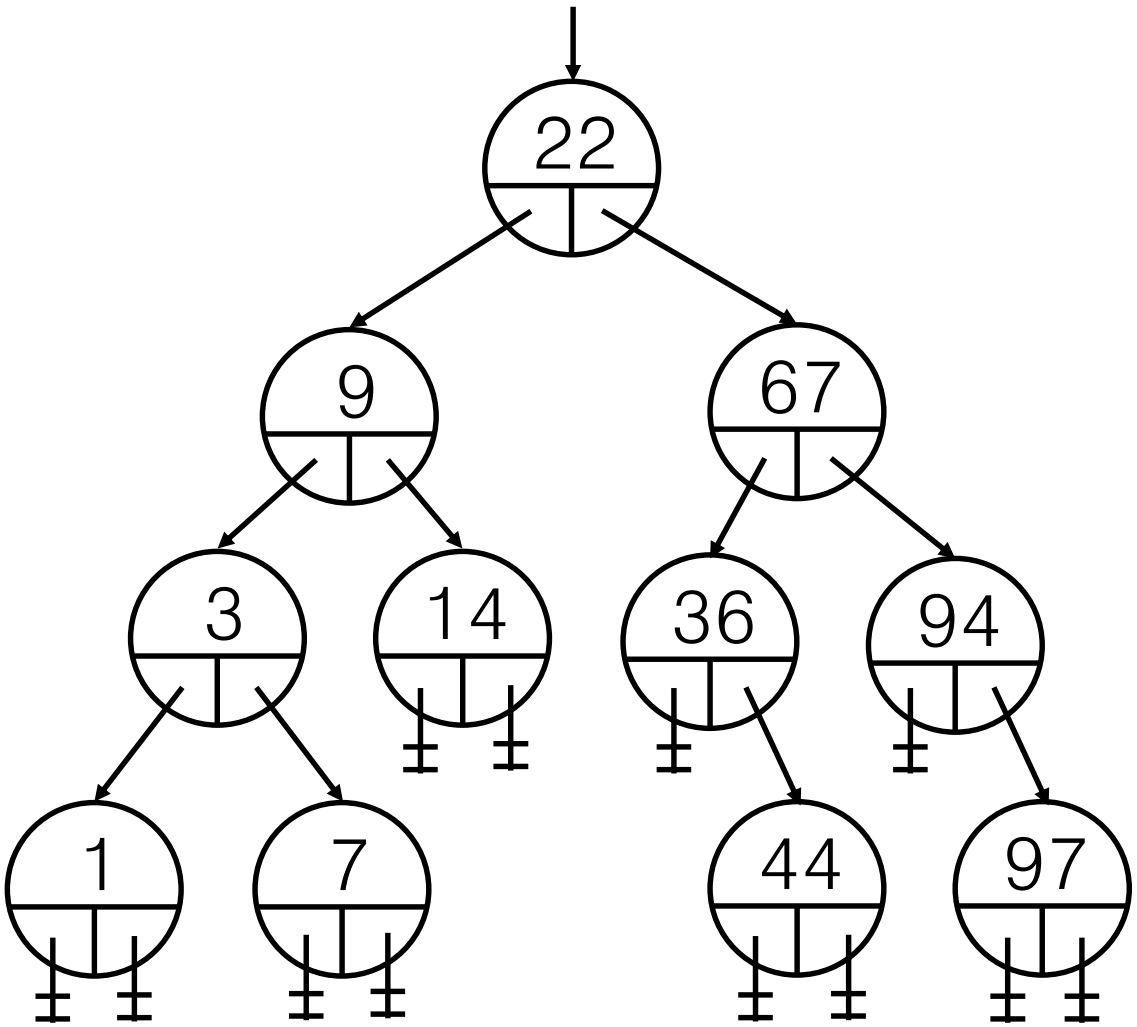


노드 삽입

(Insert Node)

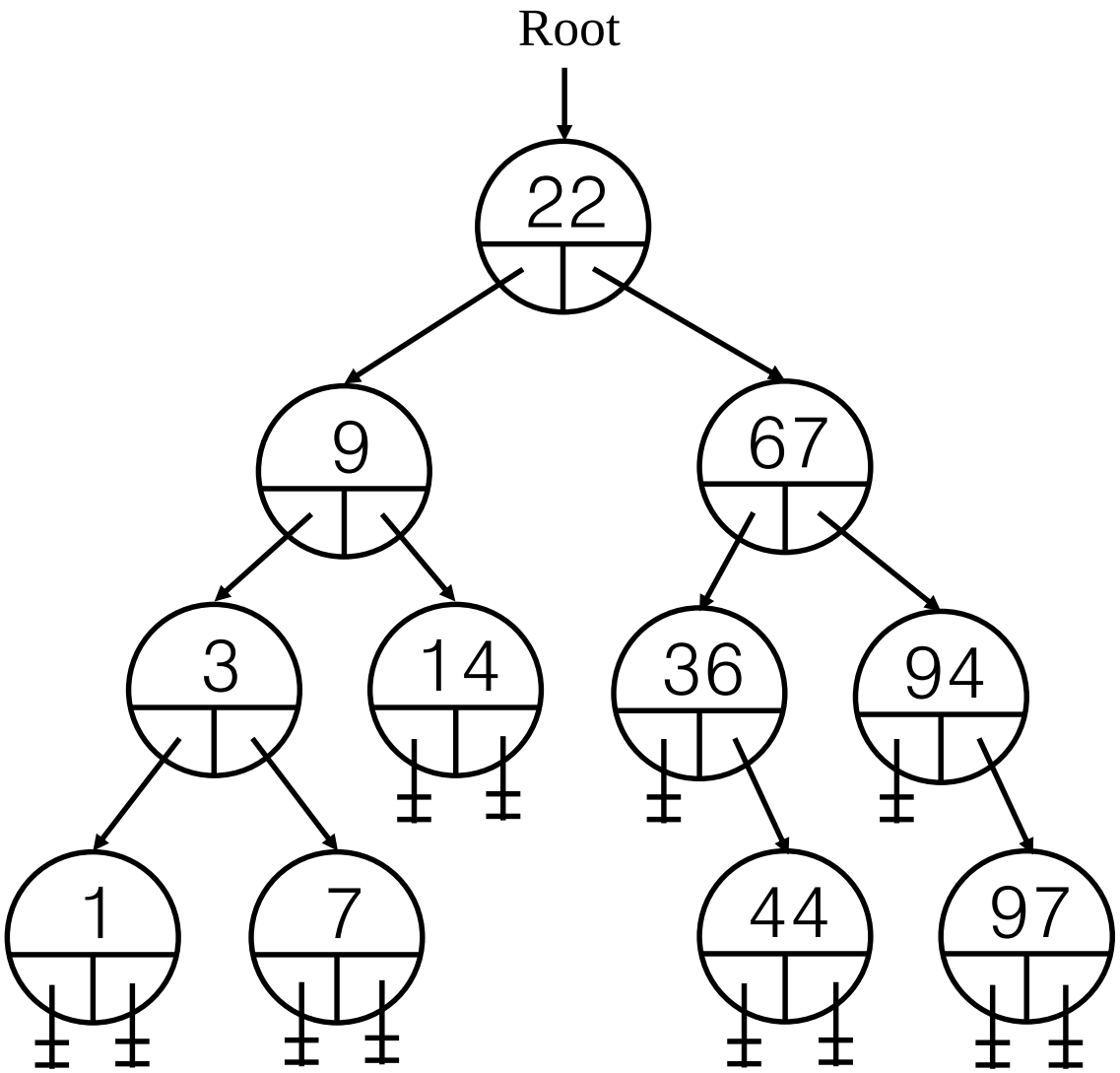
Insert Node

Root(Always search starts from the root)



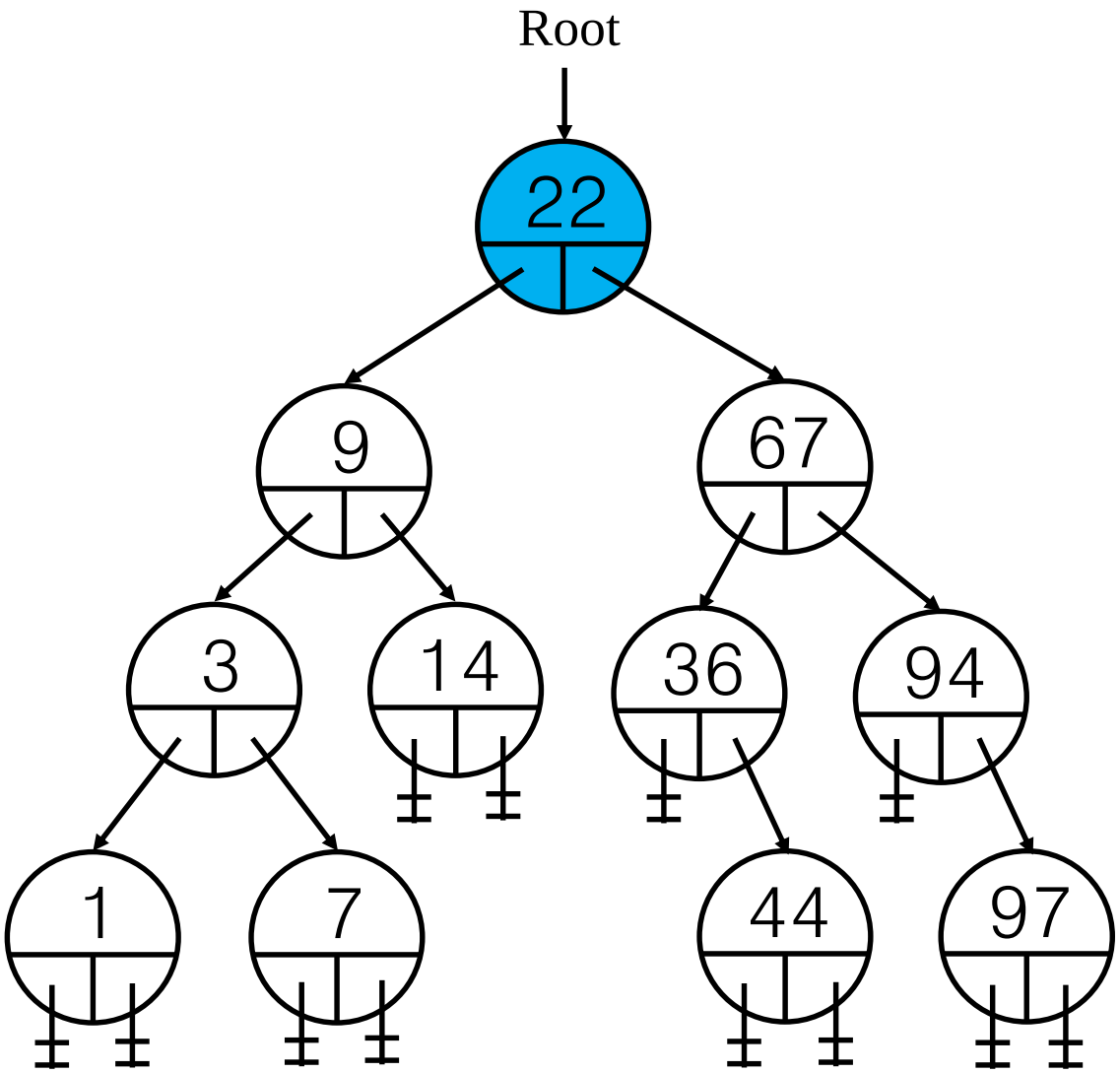
Insert Node

Let's Insert 34



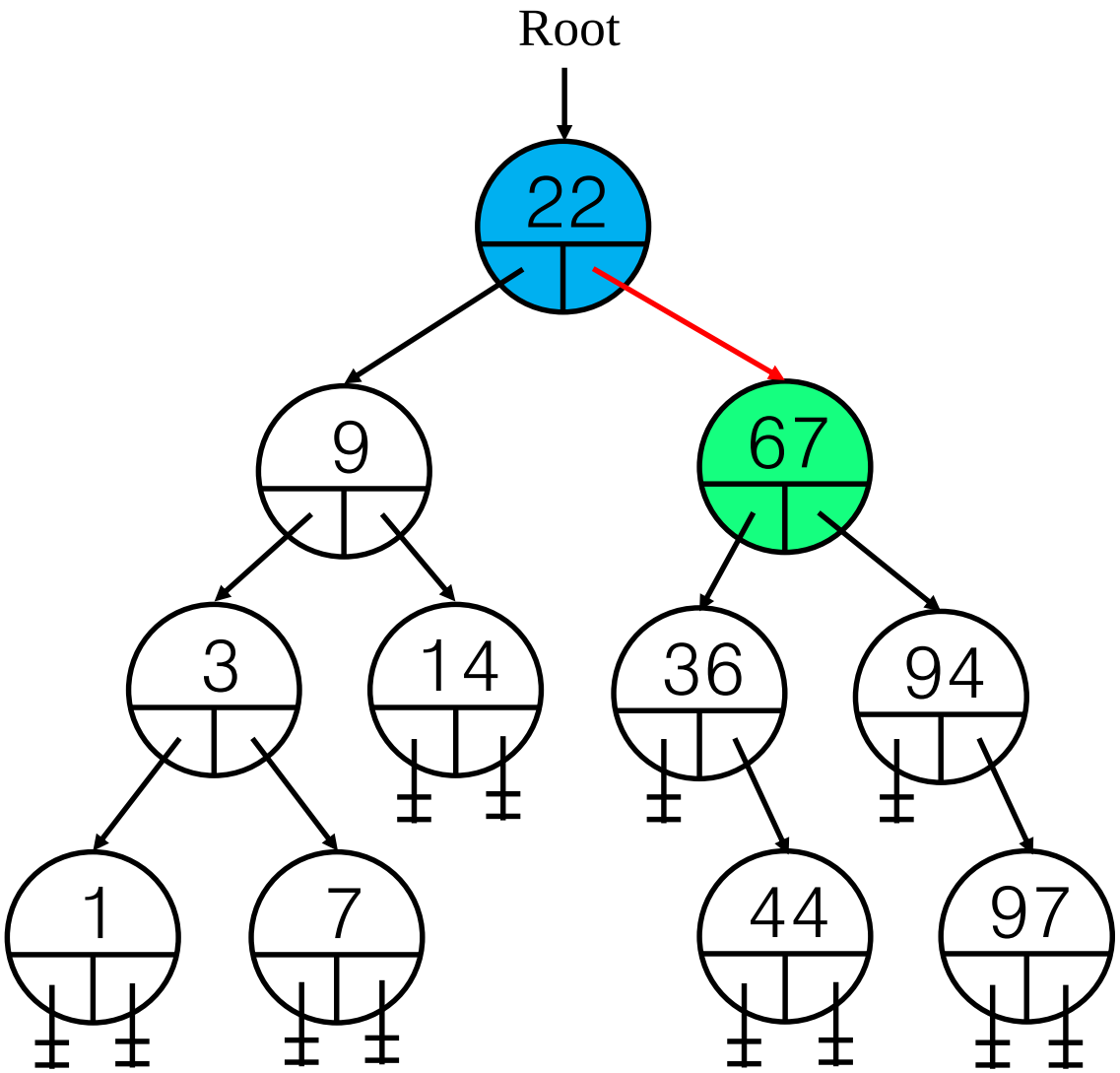
Insert Node

Let's Insert 34
34탐색부터하자



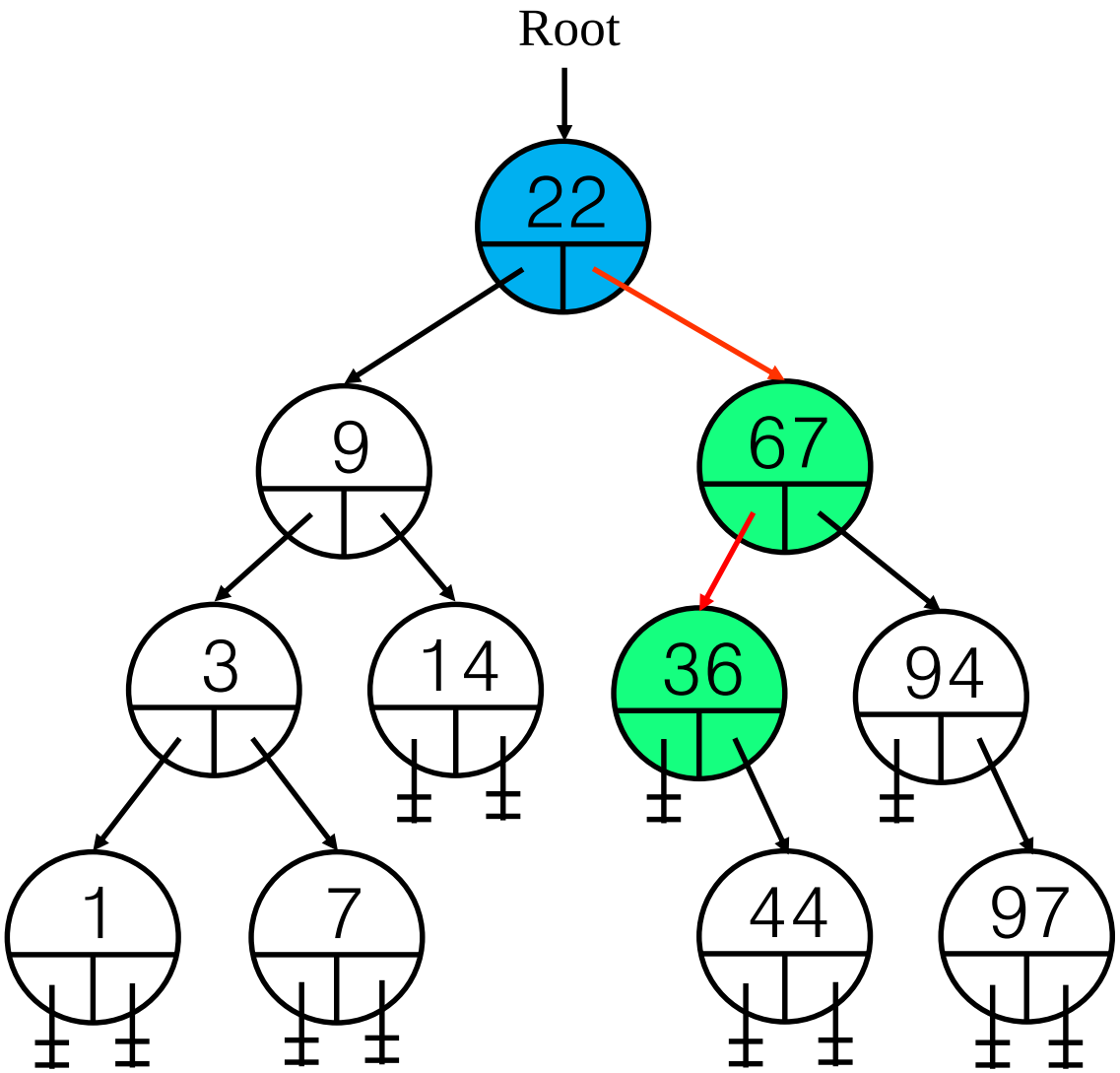
Insert Node

Let's Insert 34



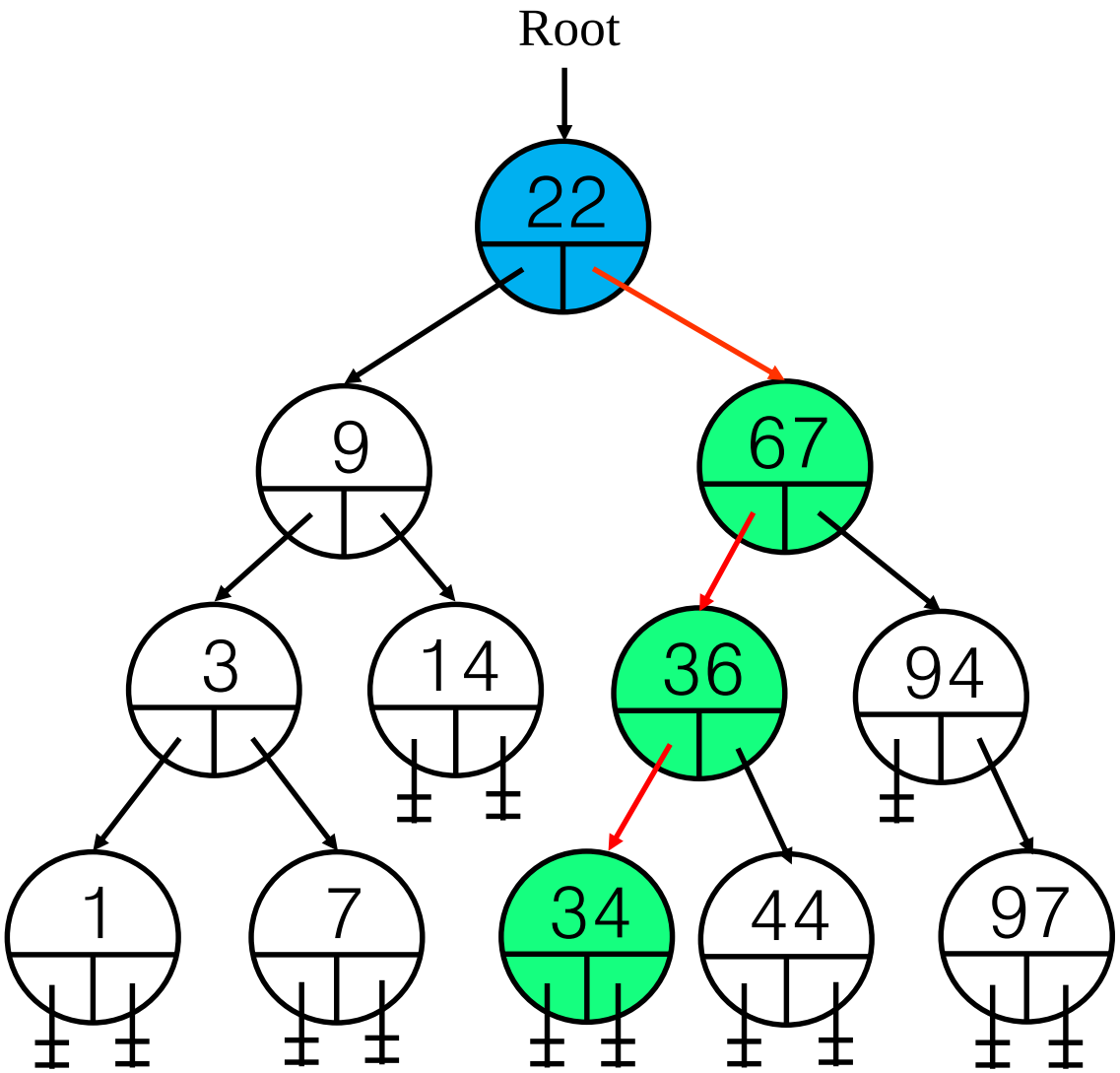
Insert Node

Let's Insert 34



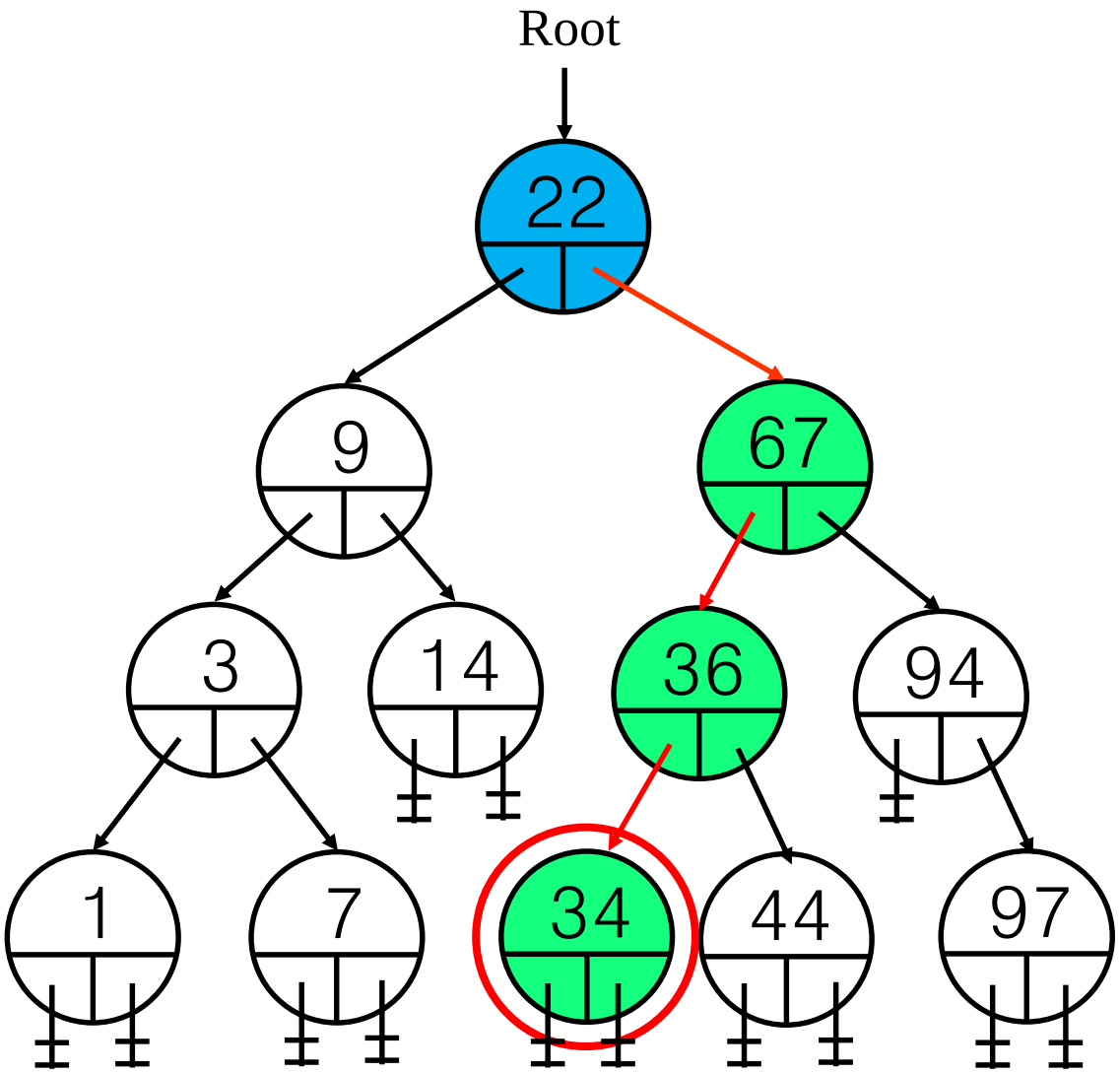
Insert Node

Let's Insert 34



Insert Node

Let's Insert 34



Deleting from a Binary Search Tree (BST)원소삭제

Binary Search Trees (6/8)원소삭제

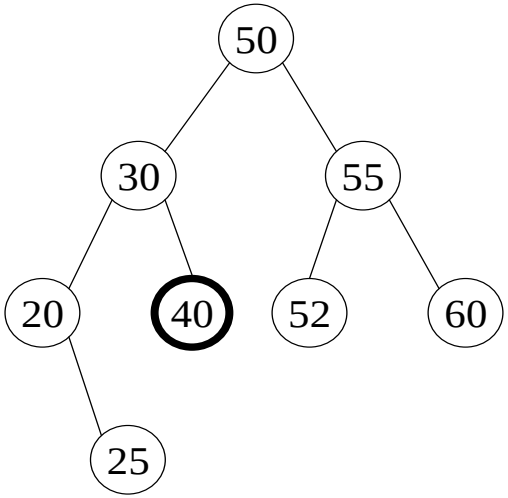
- Deletion from a binary search tree
  - Three cases should be considered 3가지경우가 있음
  - case 1. leaf → delete 나뭇잎이면 그냥삭제
  - case 2.
    one child → delete and change the pointer to this child
    삭제하려는 원소가 자식이 하나있으면, 삭제하고 자식이 그 위치
  - case 3. two child → either the smallest element in the right subtree or the largest element in the left subtree 자식이 둘 있으면 왼쪽서브트리에서가장큰원소 혹은 오른쪽 서브트리에서가장작은원소를 대치

Deletion algorithm

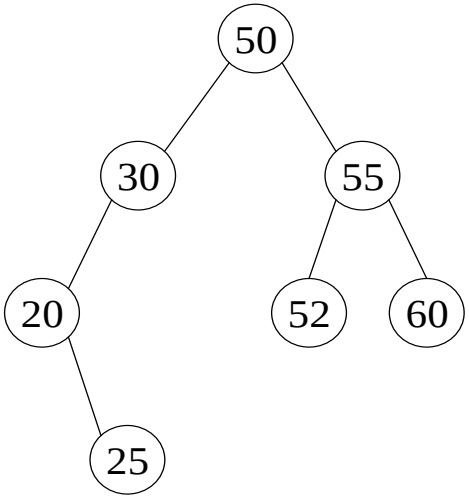
```
deleteBST(B, x)
    p ← the node to be deleted;                  //주어진 키값 x를 가진 노드
    parent ← the parent node of p;               // 삭제할 노드의 부모 노드
    if p = null then return false;     // 삭제할 원소가 없음
    case {
        p.left = null and p.right = null :    // 삭제할 노드가 리프 노드인 경우
                if parent.left = p then parent.right ← null;
                                else parent.right ← null;
        p.left = null or p.right = null :          // 삭제할 노드의 차수가 1인 경우
                if p.left ≠ null then {
                    if parent.left = p then parent.left ← p.left;
                                    else parent.right ← p.left;
                } else {
                    if parent.left = p then parent.left ← p.right;
                                    else parent.right ← p.right;
                                }
        p.left ≠ null and p.right ≠ null :        // 삭제할 노드의 차수가 2인 경우
                                q ← maxNode(p.left);                    // 왼쪽 서브트리에서
                                                                        // 최대 키값을 가진 원소를 탐색
                                p.key ← q.key;
                deleteBST(p.left, p.key);
    }
end deleteBST()
```

BST 삭제(자식없음)

- Case1 : leaf node deletion



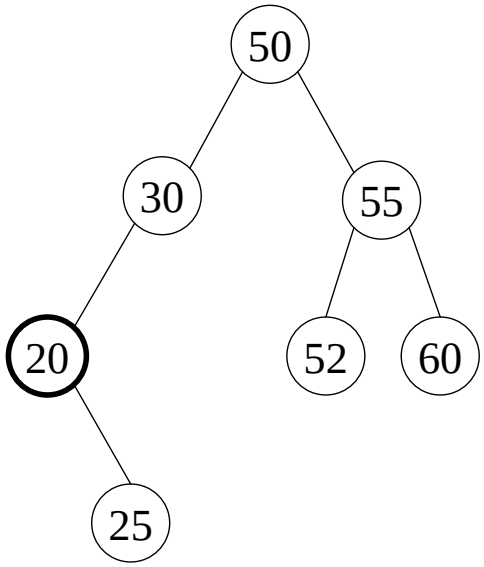
(a) 삭제전



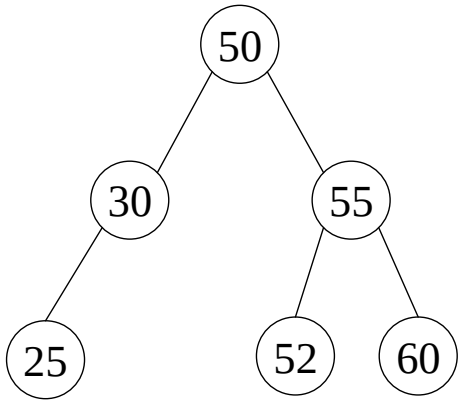
(b) 삭제후

BST삭제(자식하나있음)

- case 2.
  one child → delete and change the pointer to this child



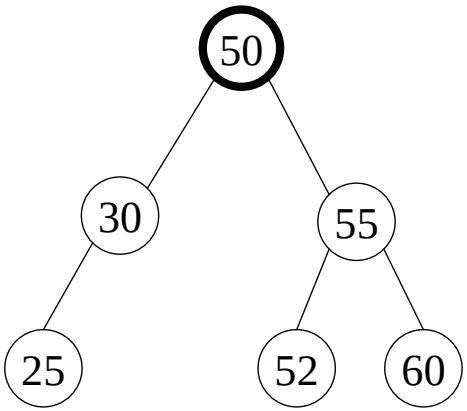
(a) 삭제전



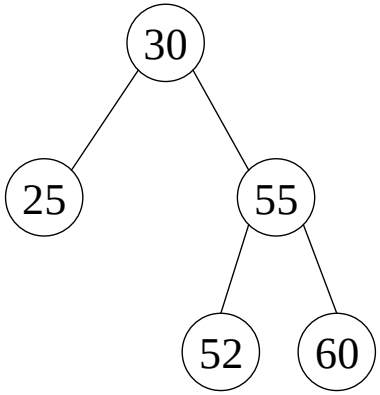
(b) 삭제후

BST삭제(자식둘있음)

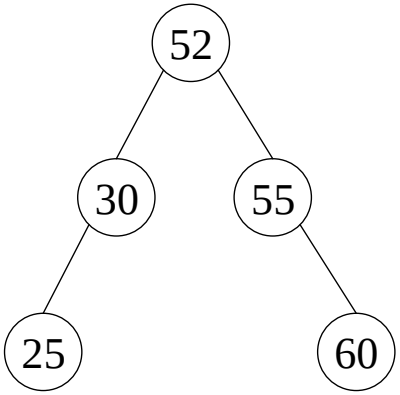
- case 3. two child → either the smallest element in the right subtree or the largest element in the left subtree



(a) 삭제전



(b) 왼쪽 서브트리의
최대 원소로 대체

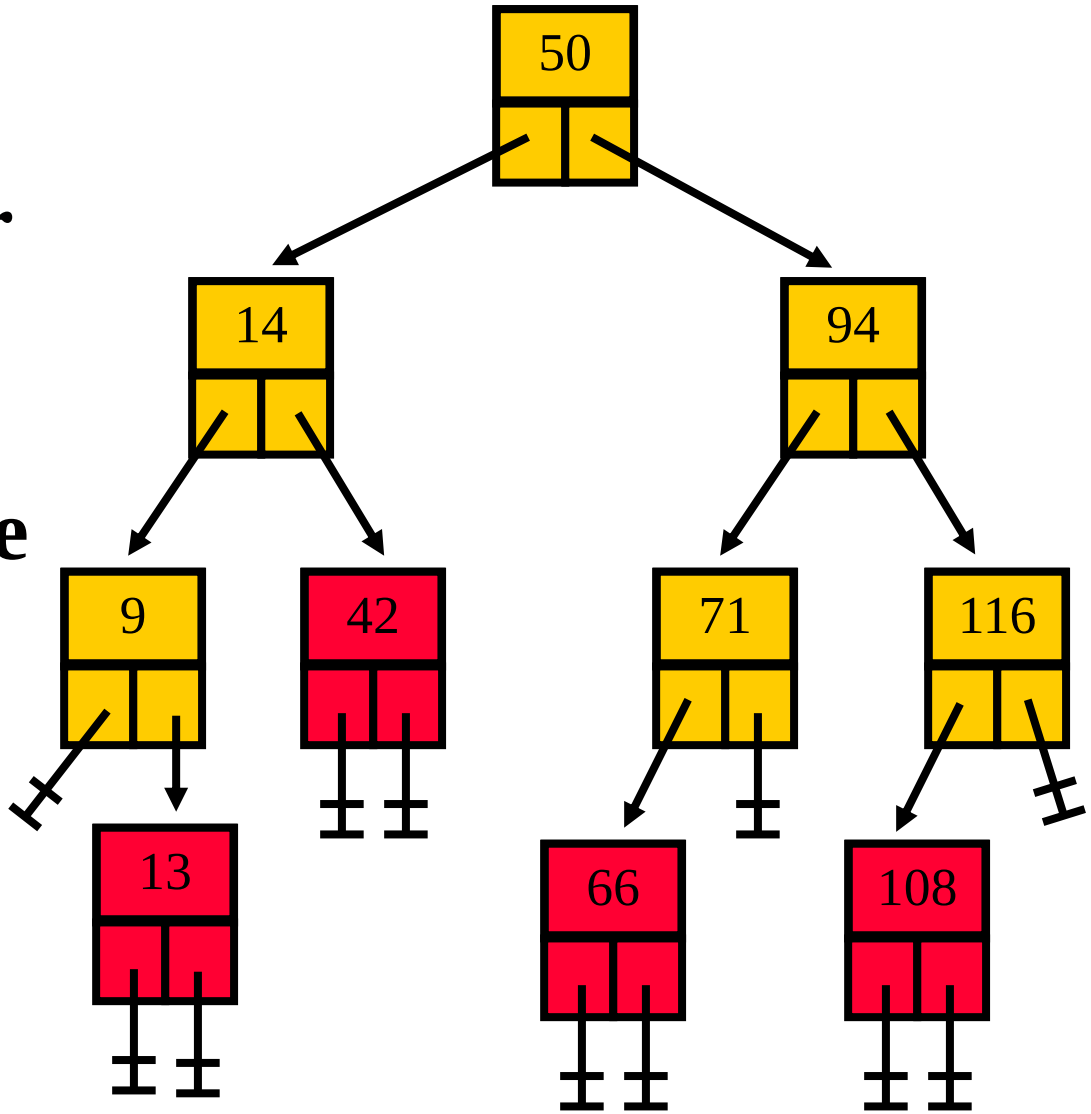


(c) 오른쪽 서브트리의
최소 원소로 대체

Delete a Leaf Node(나뭇잎삭제)

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

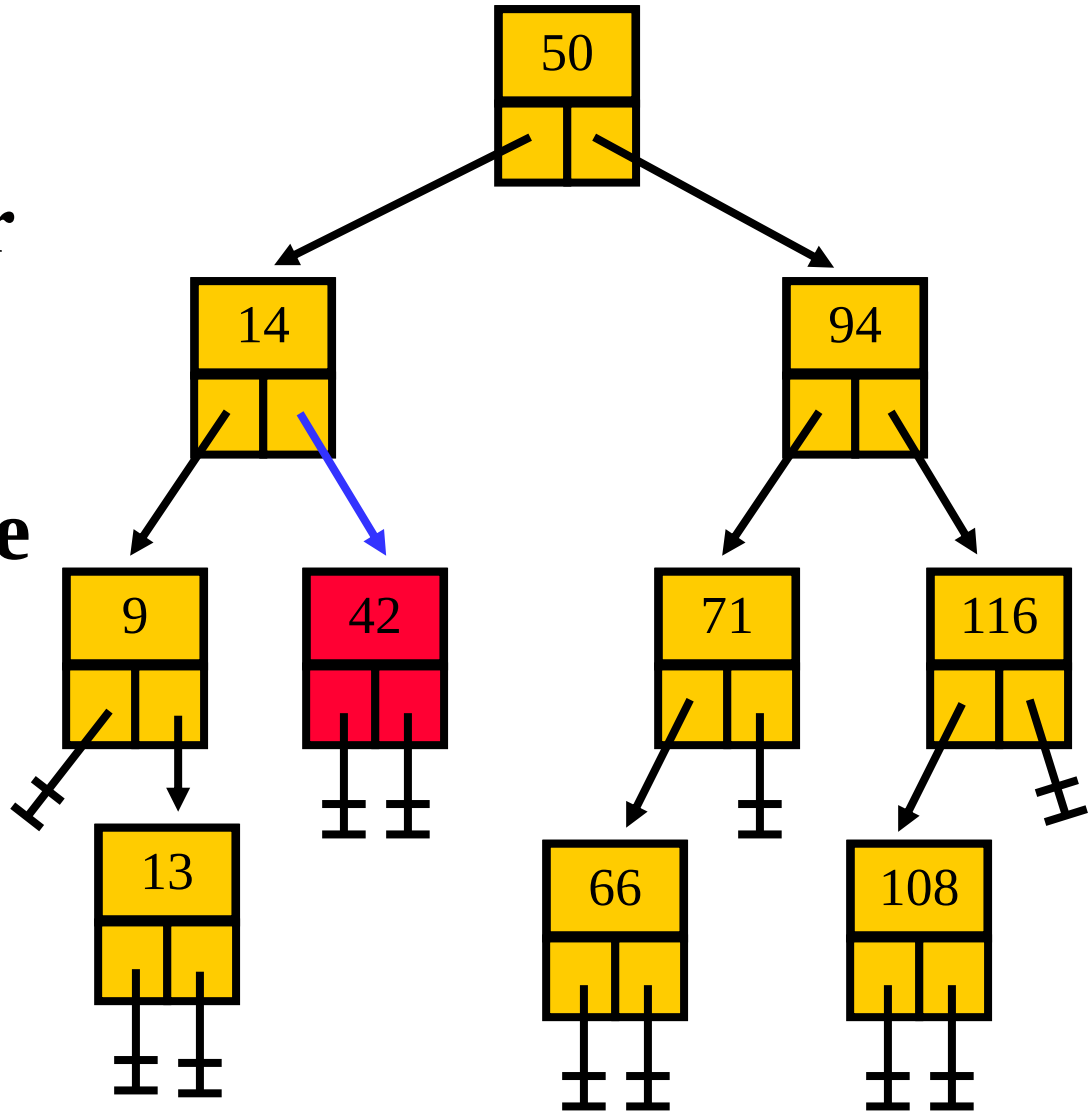


Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

Let's delete 42.

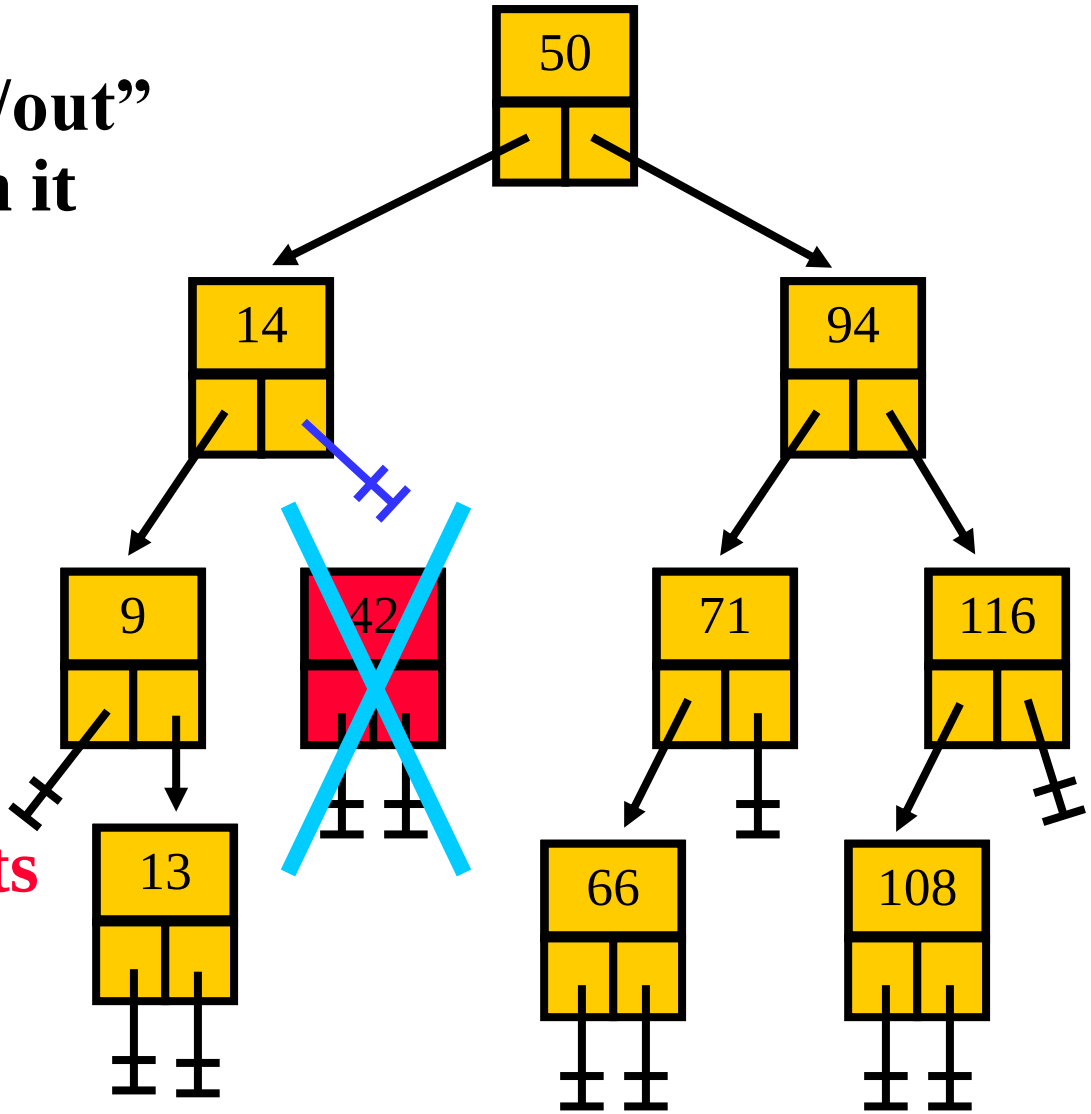


Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

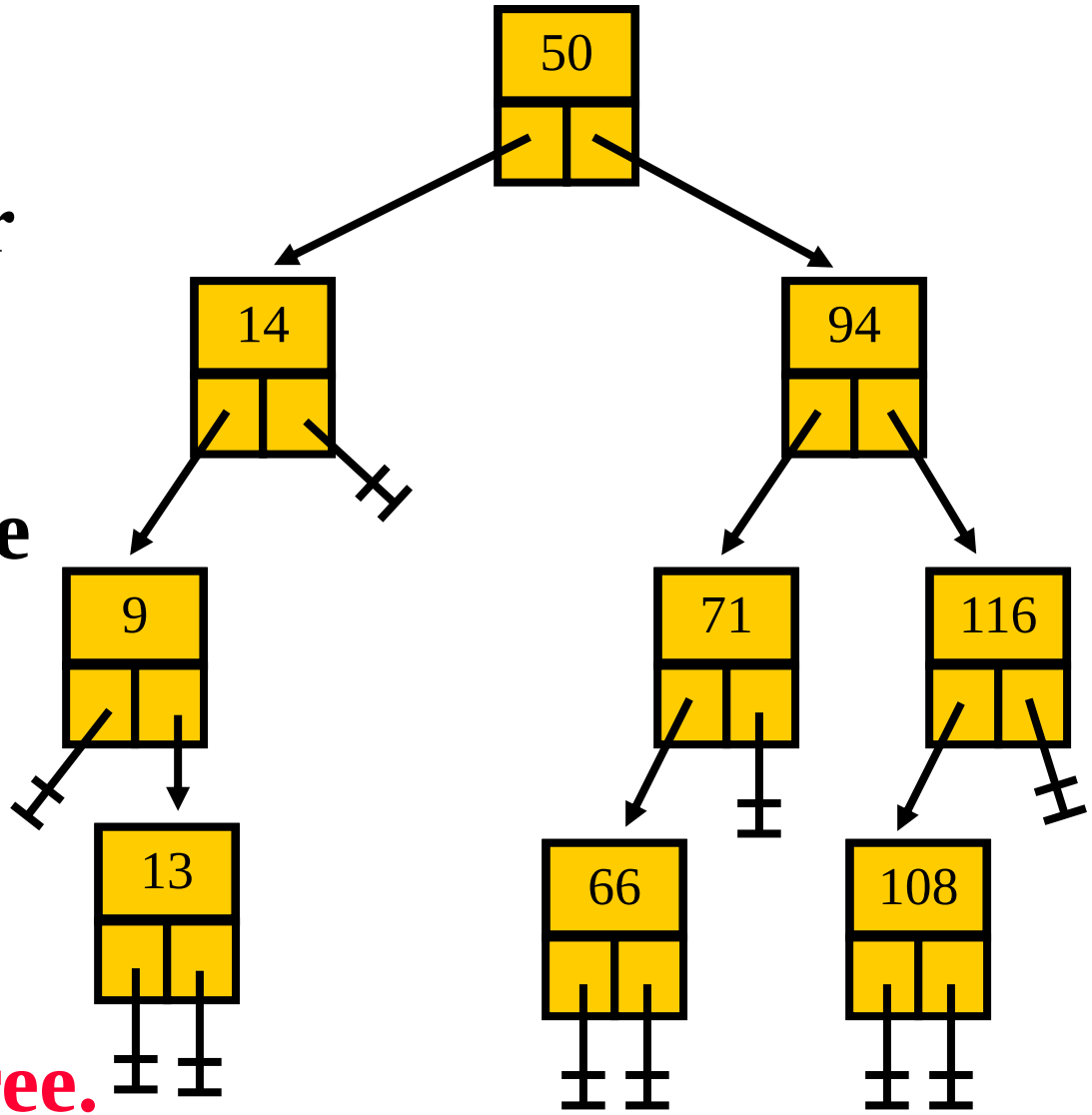
Move the pointer; now nothing points to the node.



Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```



The resulting tree.

Delete a Node with One Child

Use an "in/out" pointer.

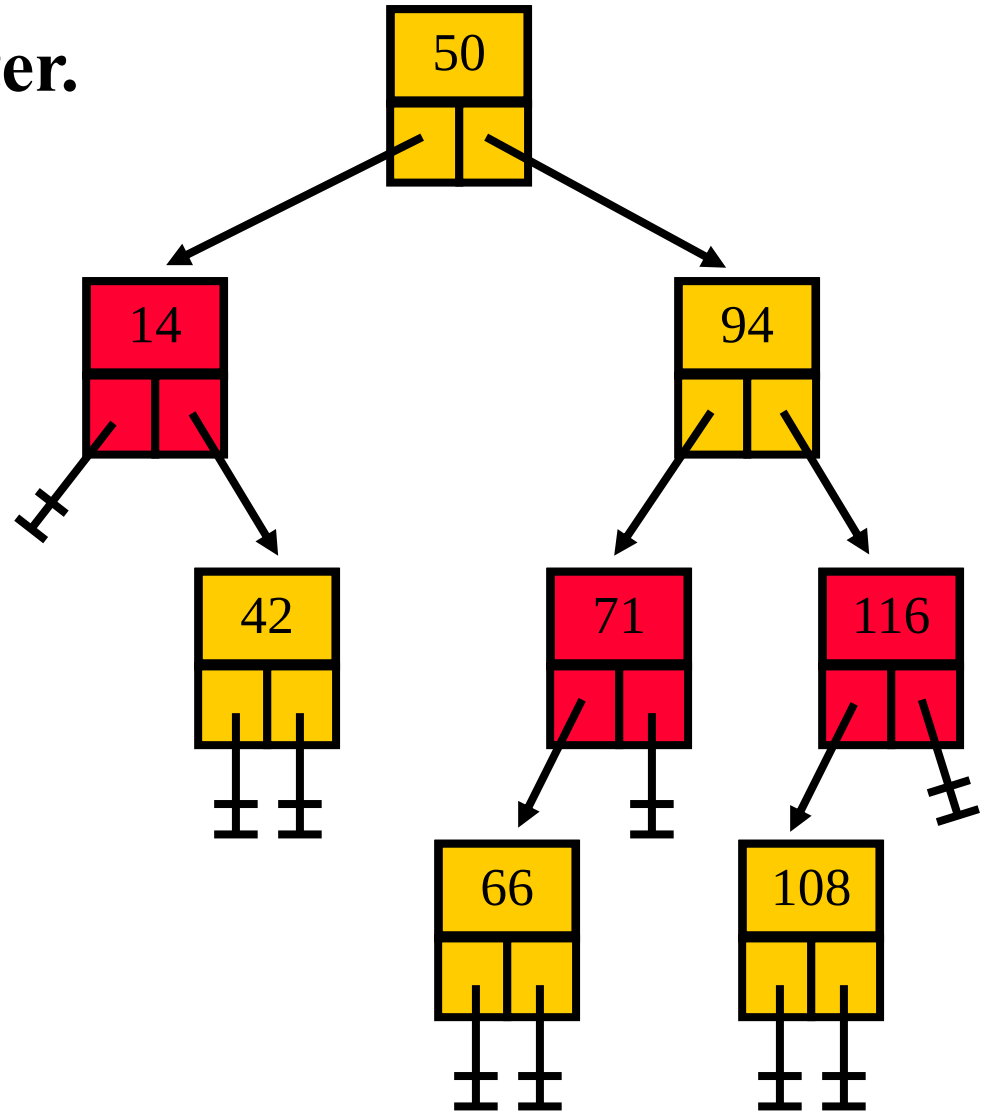
Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```



Delete a Node with One Child

Use an "in/out" pointer.

Determine if it has a left or a right child.

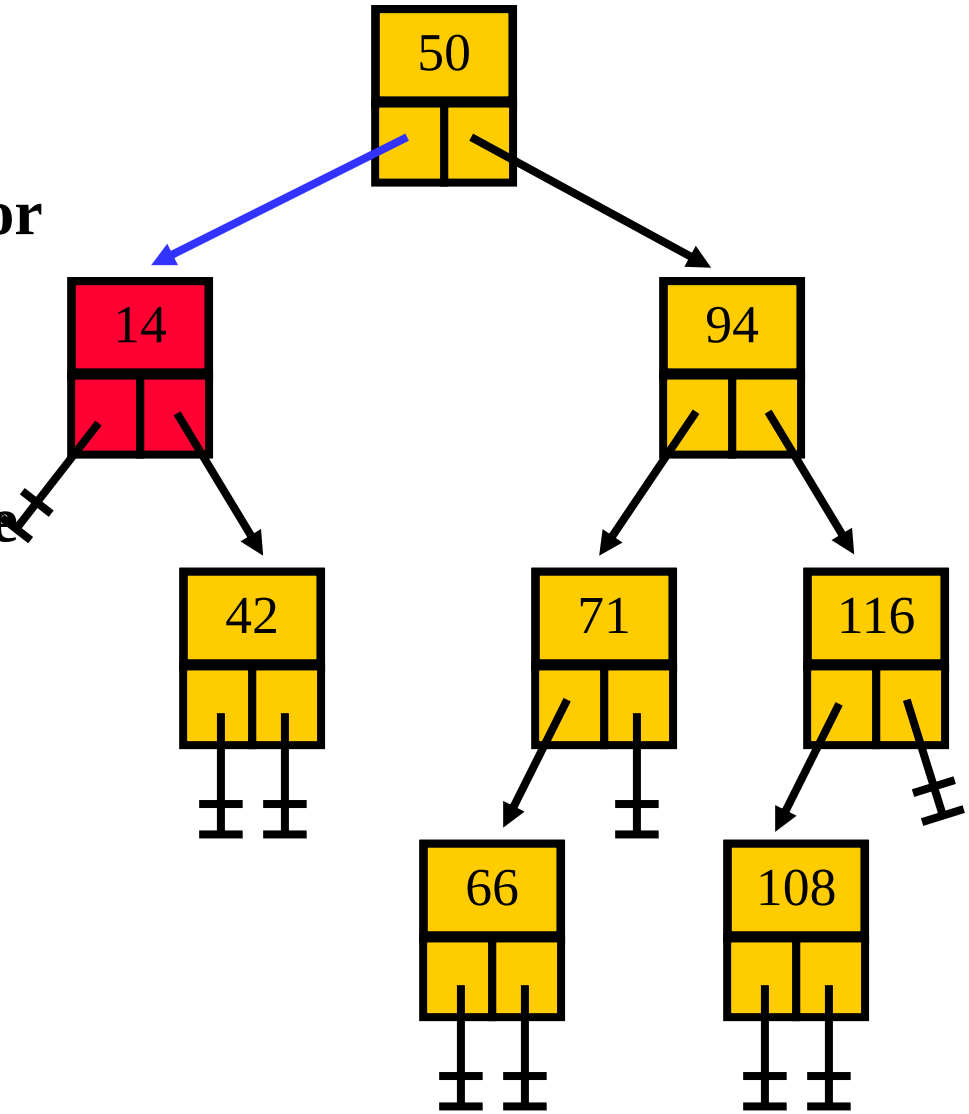
Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```

Let's delete 14.



Delete a Node with One Child

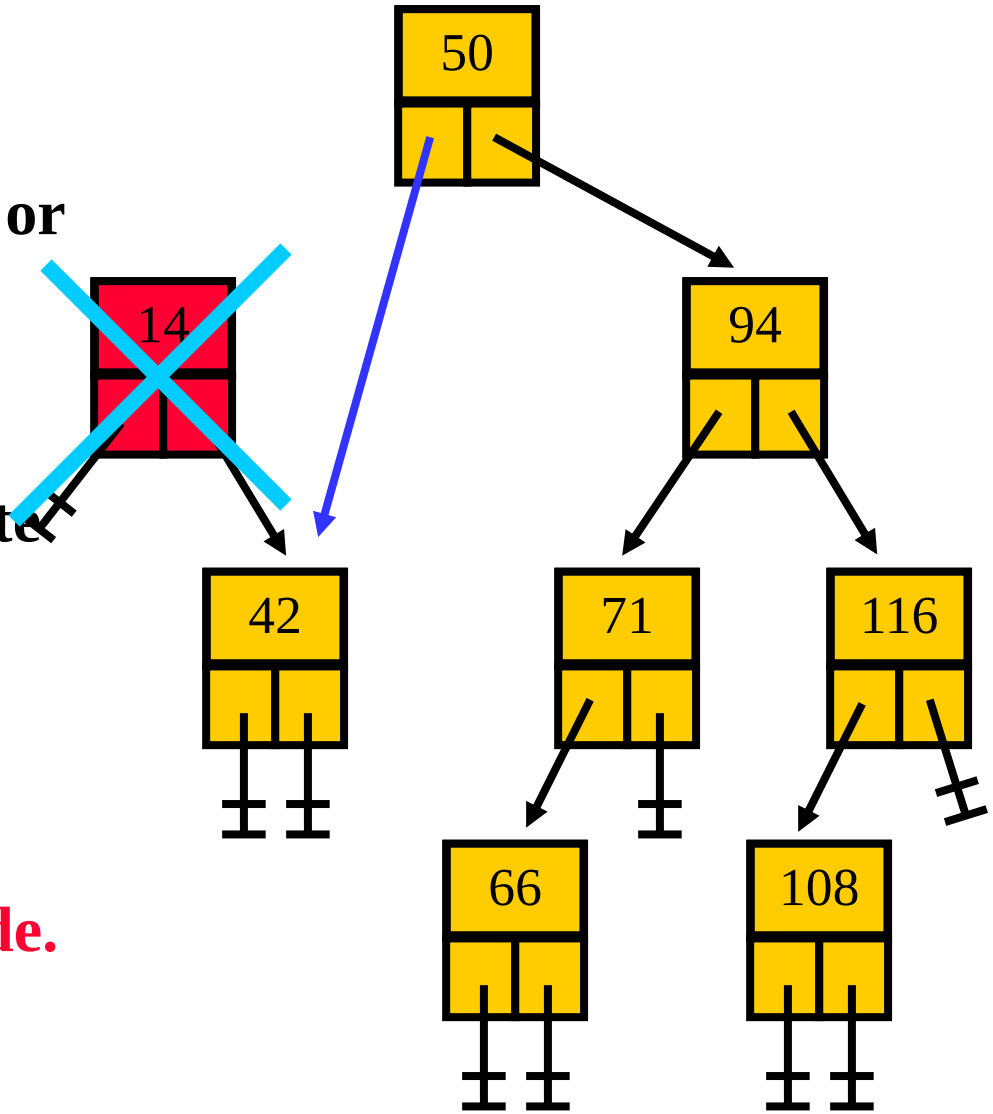
Use an "in/out" pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.right_child
```

Move the pointer; now nothing points to the node.



Delete a Node with One Child

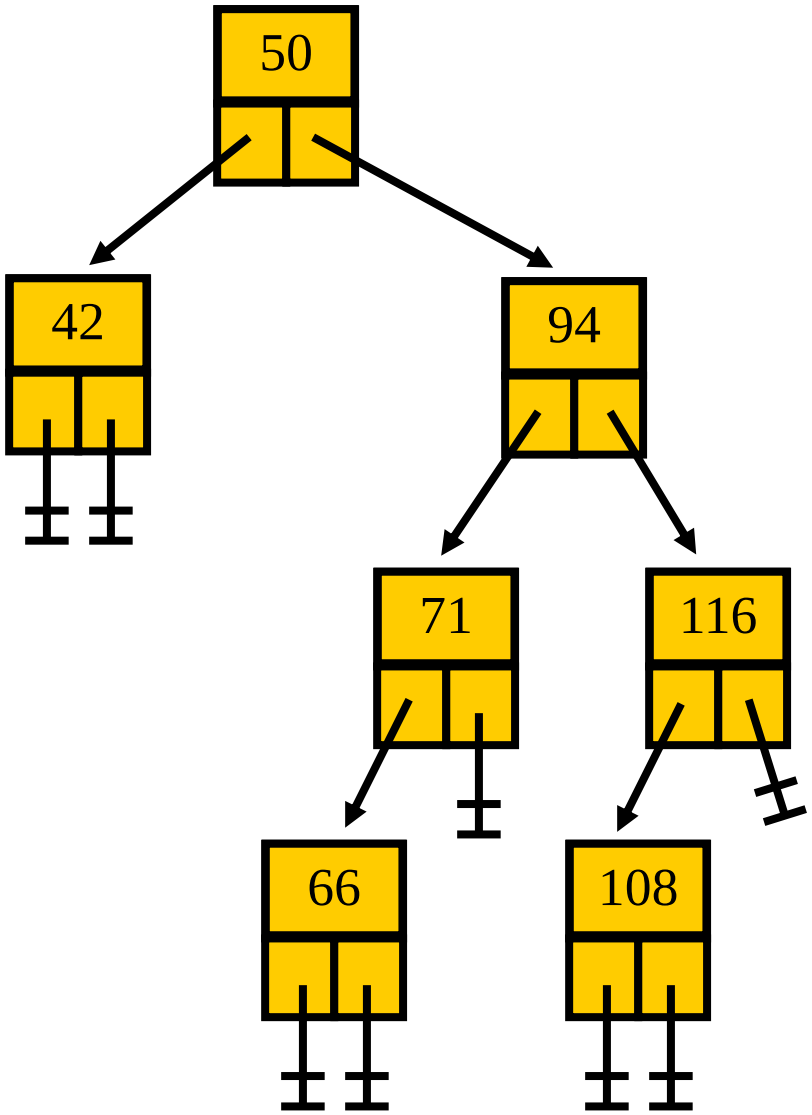
Use an "in/out" pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.right_child
```

The resulting tree.



Delete a Node with One Child

Use an "in/out" pointer.

Determine if it has a left or a right child.

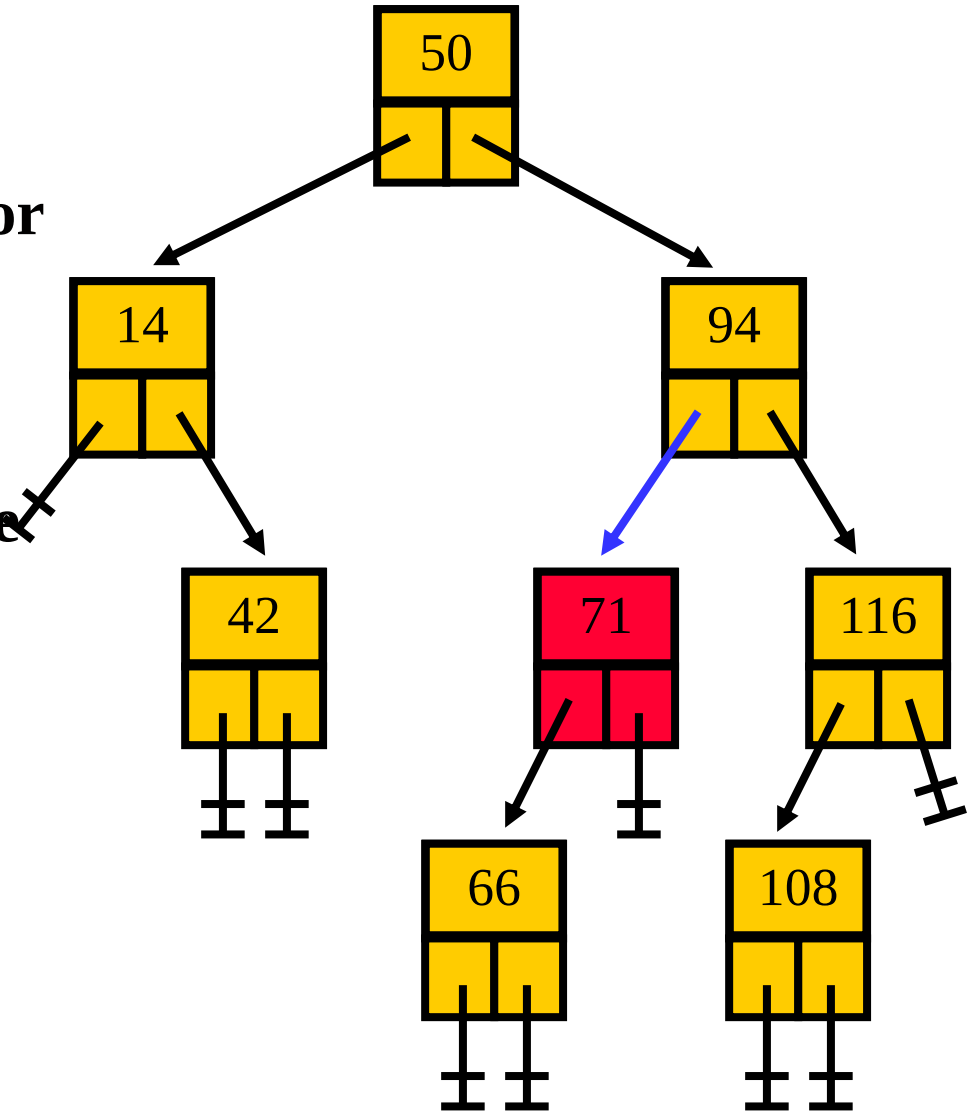
Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```

Let's delete 71.



Delete a Node with One Child

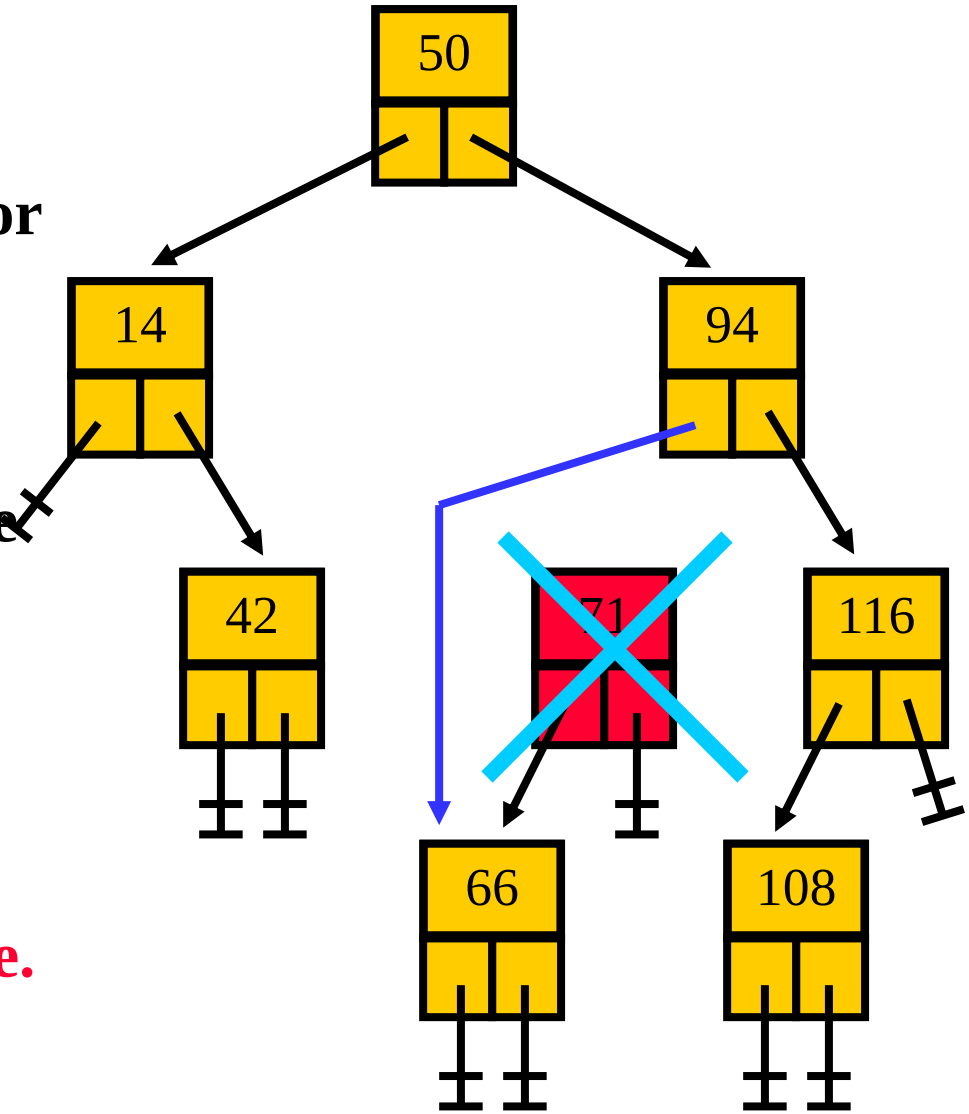
Use an "in/out" pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

Move the pointer; now nothing points to the node.



Delete a Node with One Child

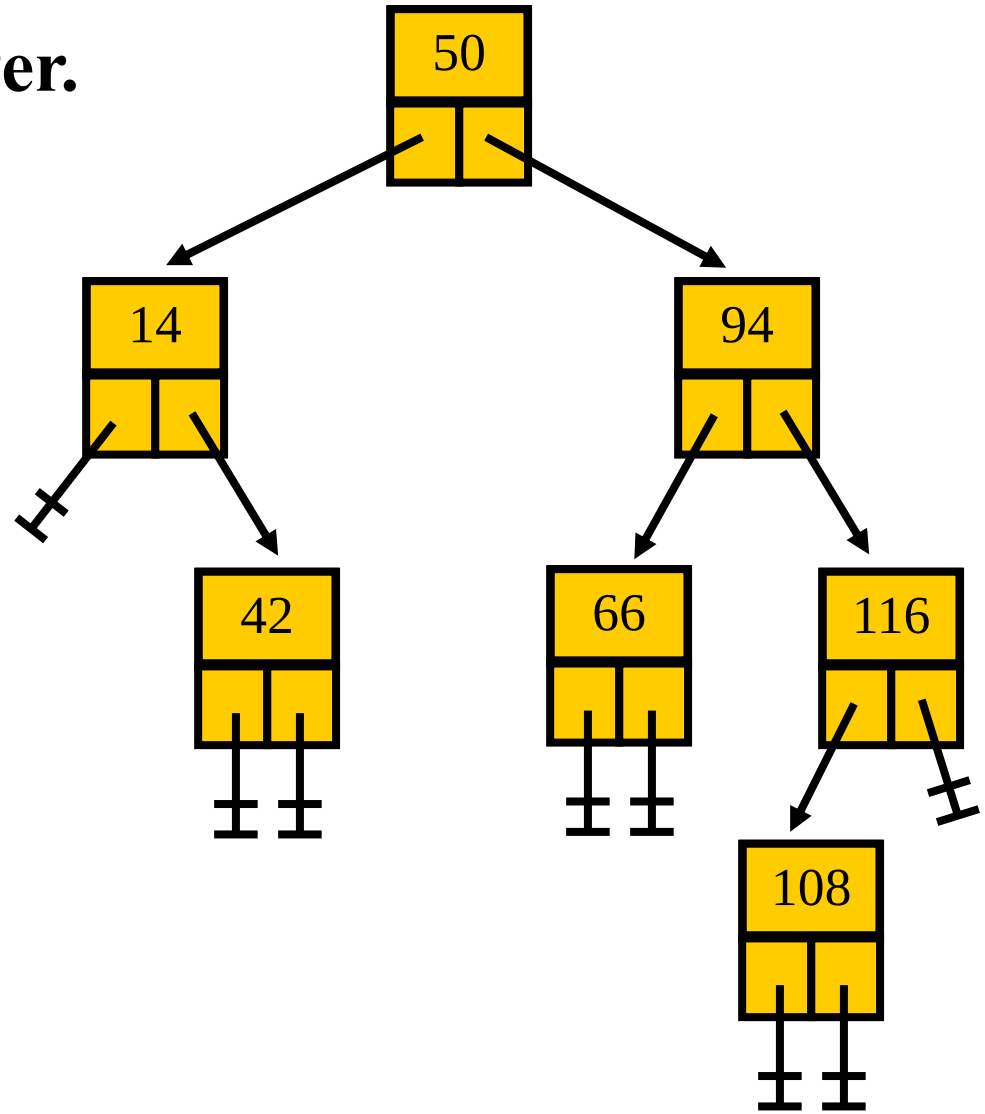
Use an "in/out" pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

The resulting tree.



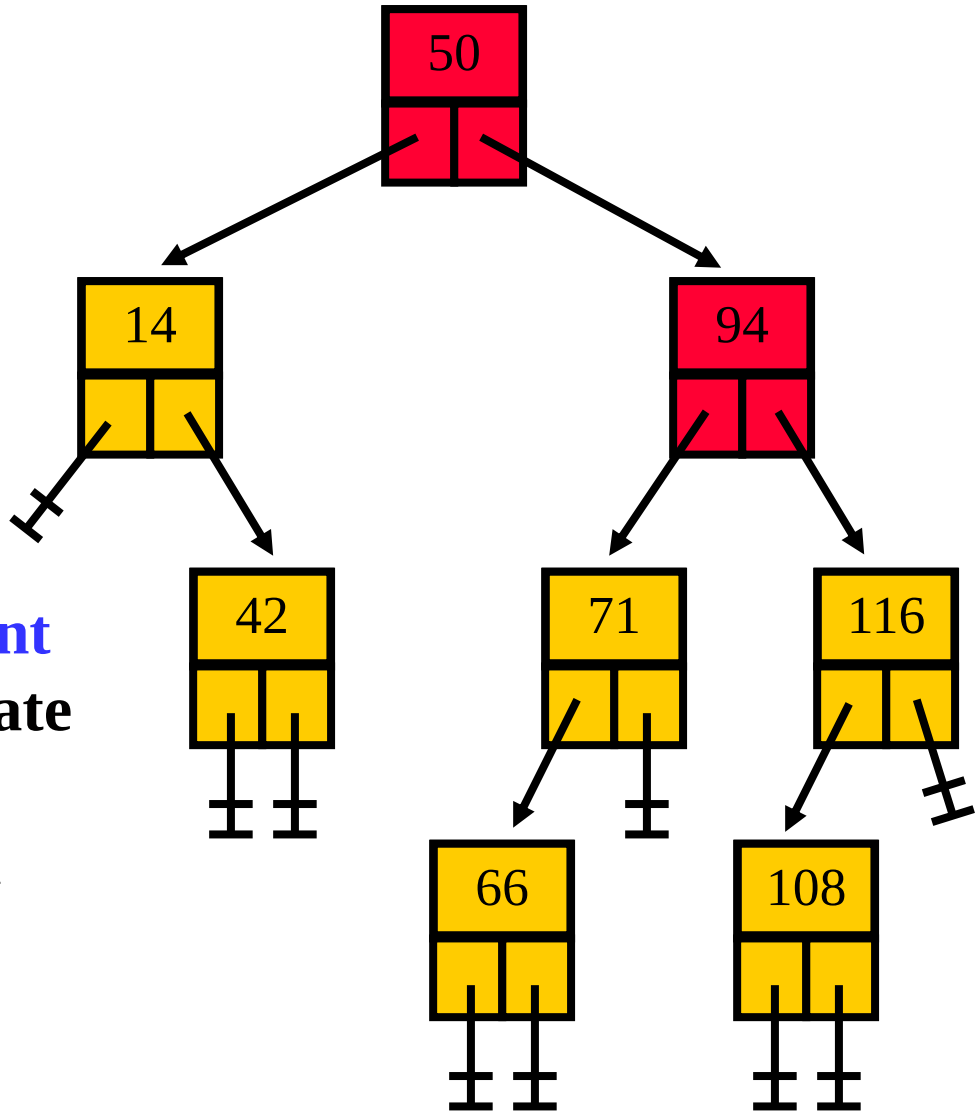
Delete a Node with Two Children

Copy a replacement value from a descendant node.

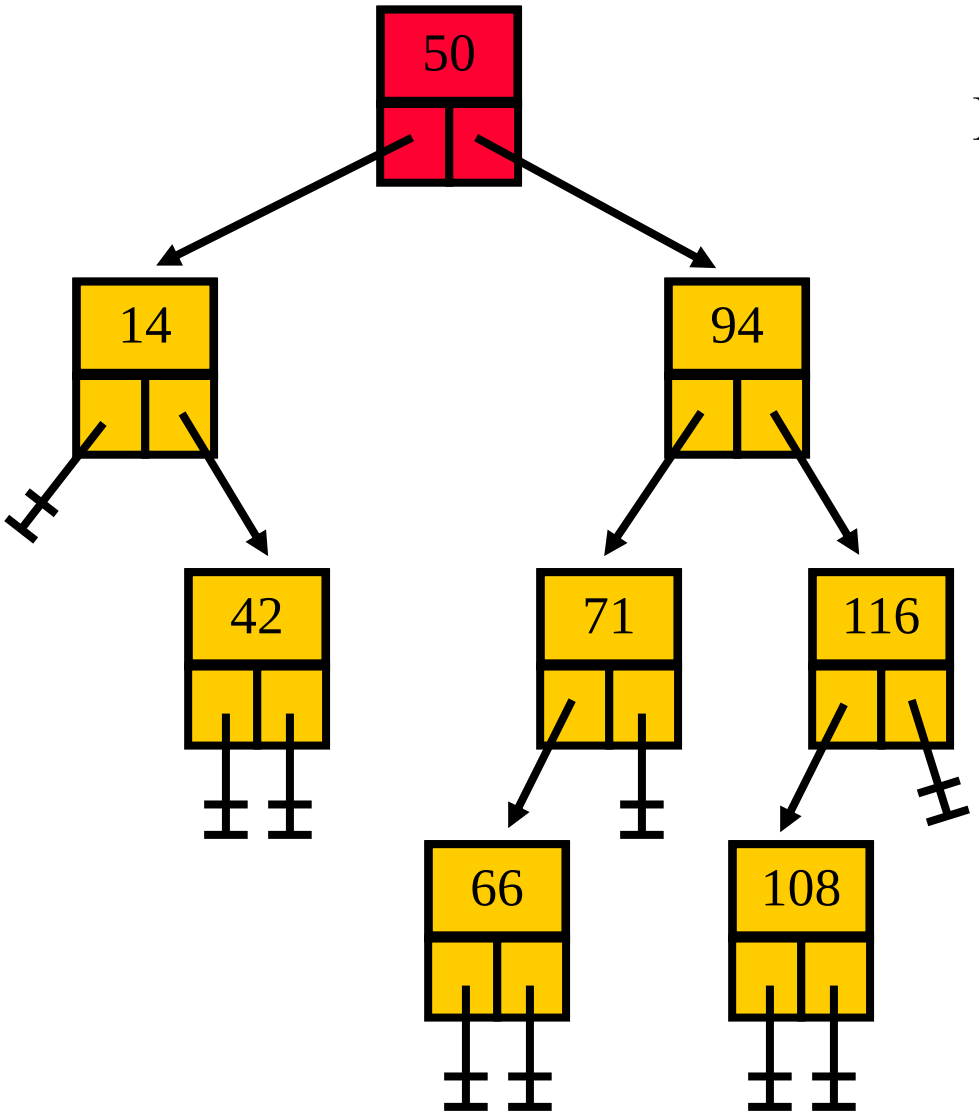
- **Largest from left**
- **Smallest from right**

Then delete that descendant node to remove the duplicate value.

- **We know this will be an easier case.**

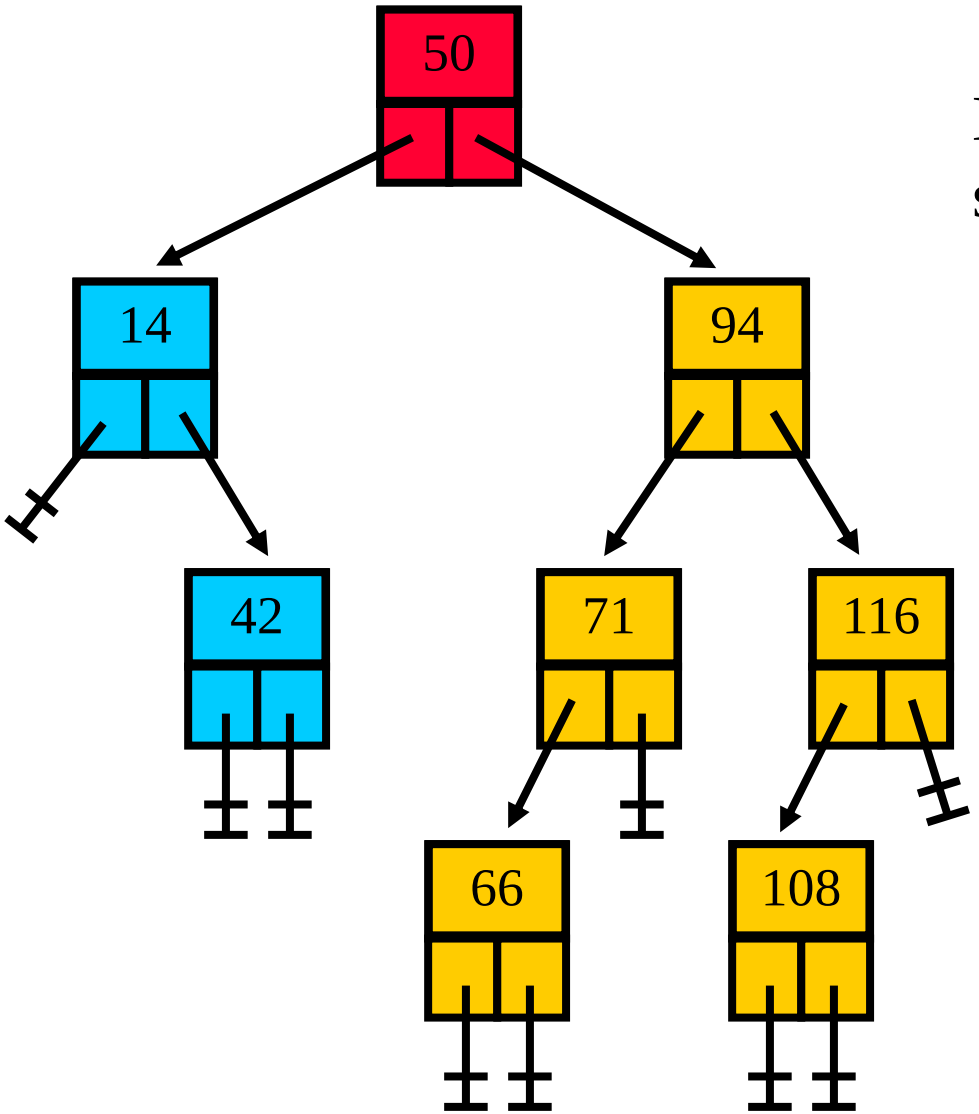


Delete a Node with Two Children



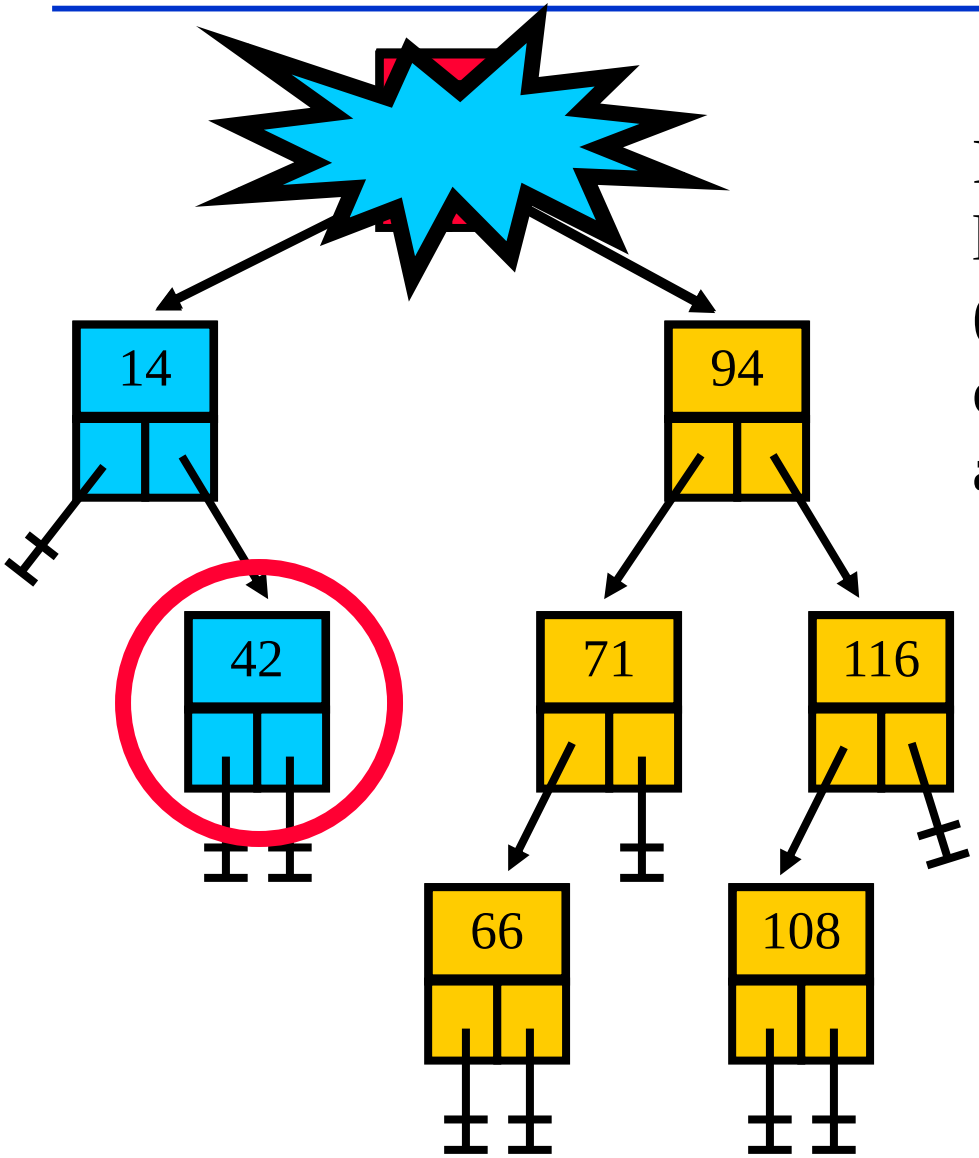
Let’s delete 50.

Delete a Node with Two Children



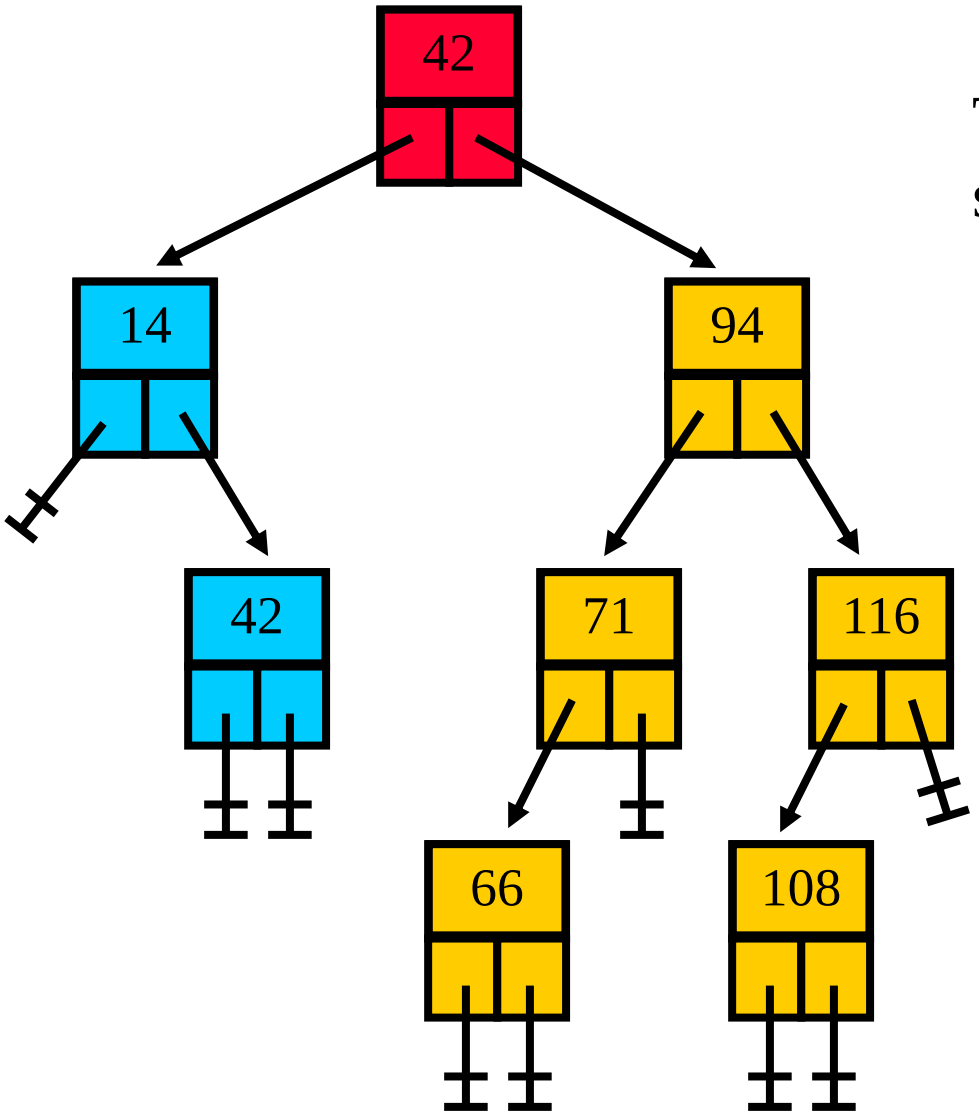
Look to the left sub-tree.

Delete a Node with Two Children



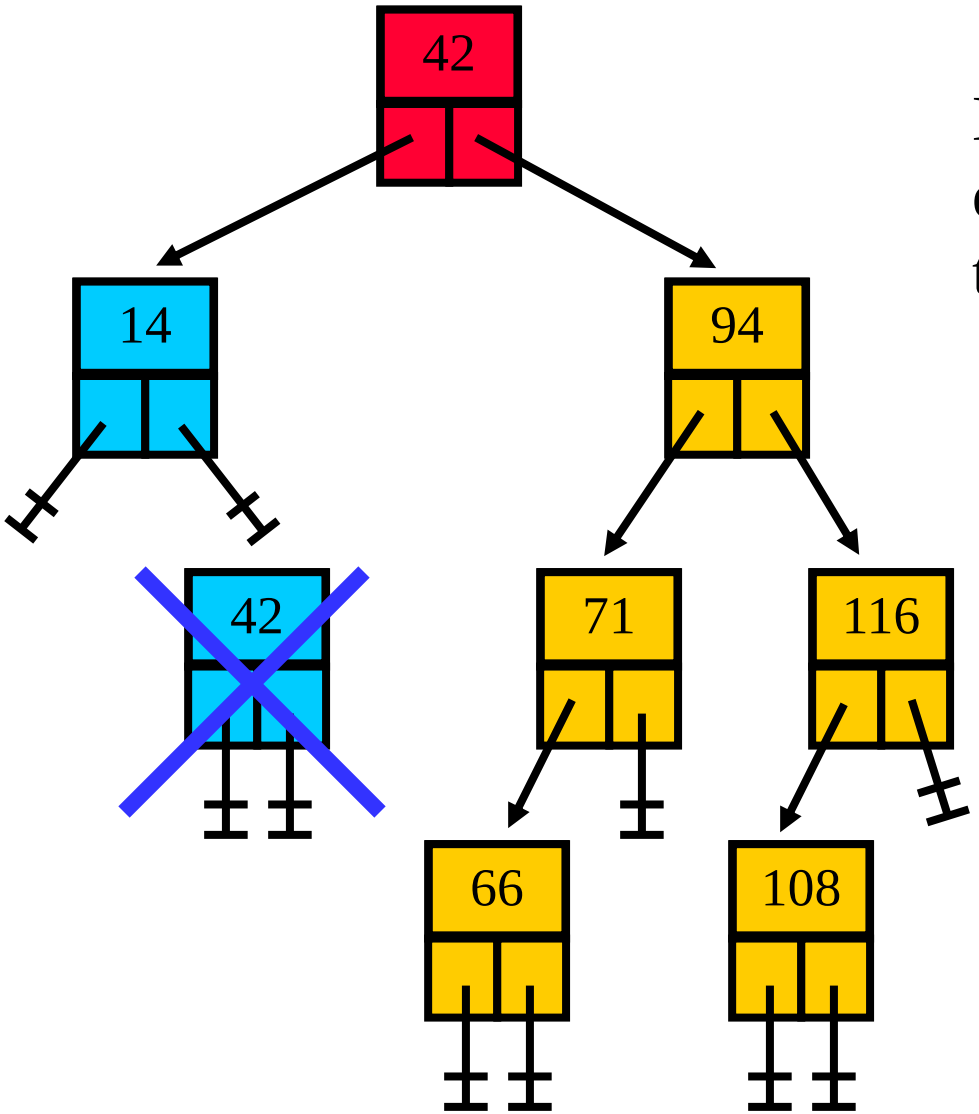
Find and copy the largest value (this will erase the old value but creates a duplicate).

Delete a Node with Two Children



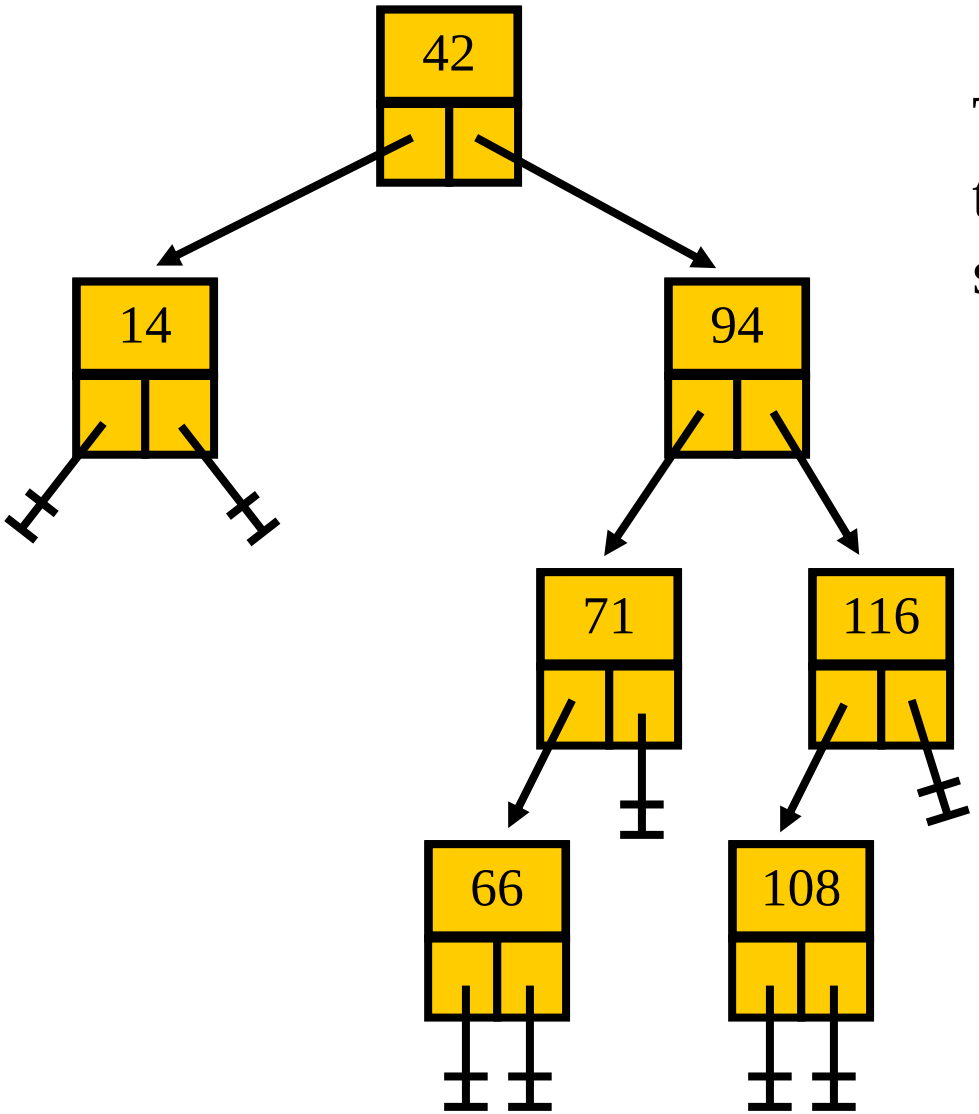
The resulting tree so far.

Delete a Node with Two Children



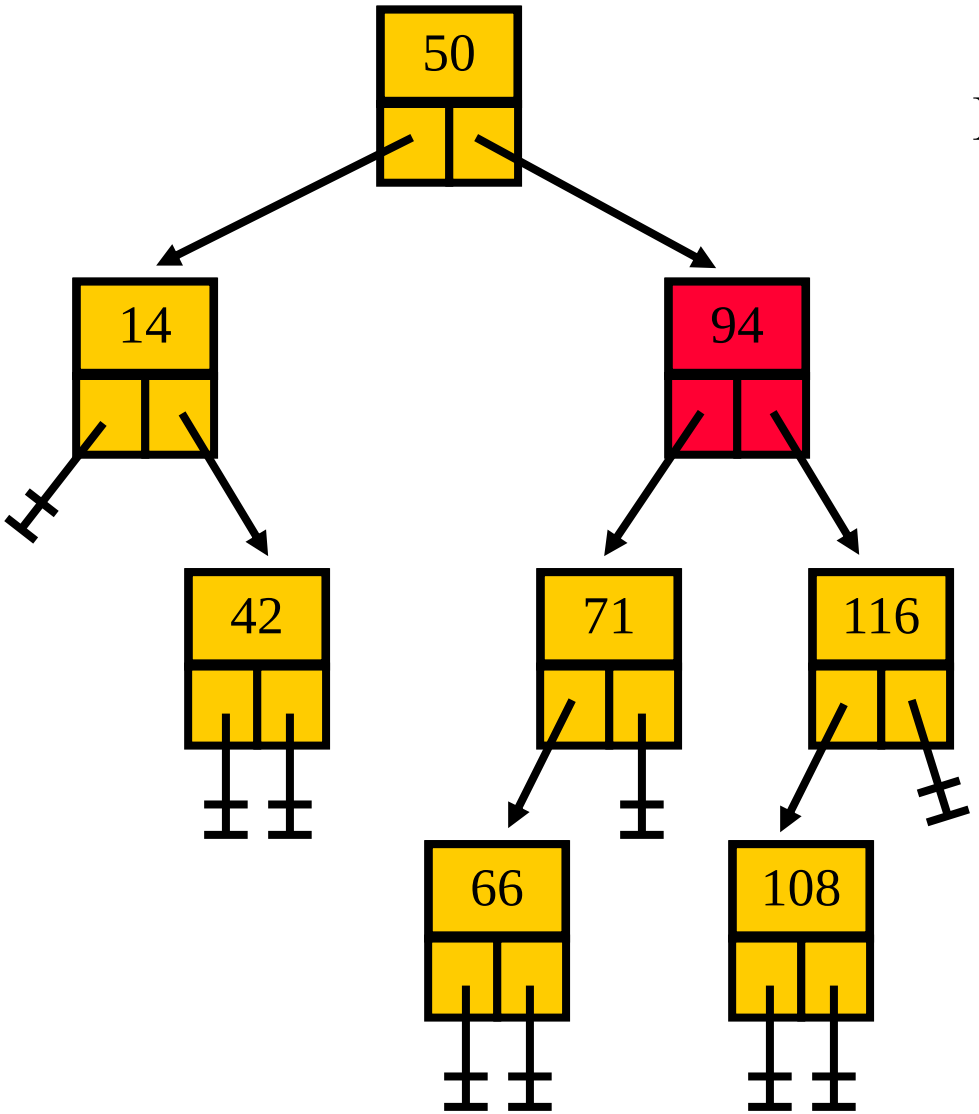
Now delete the duplicate from the left sub-tree.

Delete a Node with Two Children



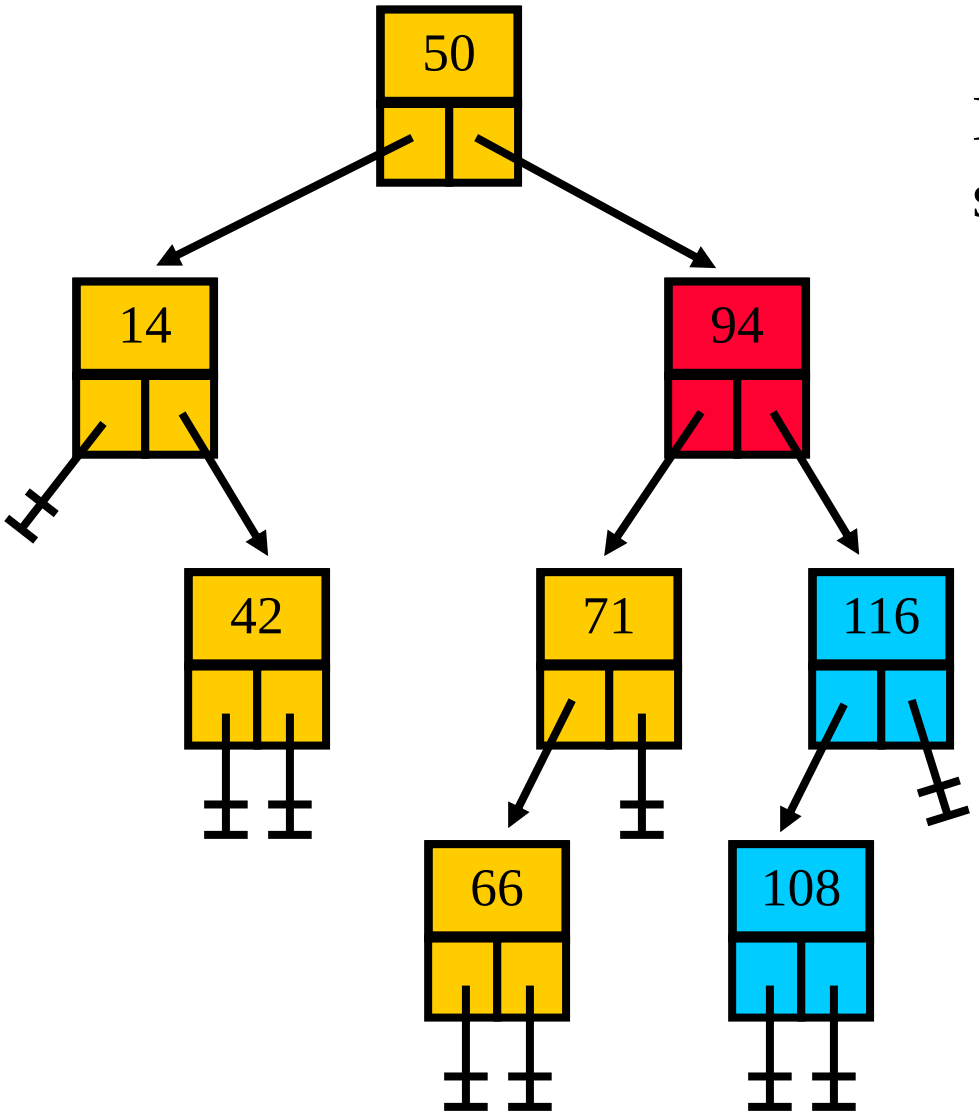
The final resulting tree – still has search structure.

Delete a Node with Two Children



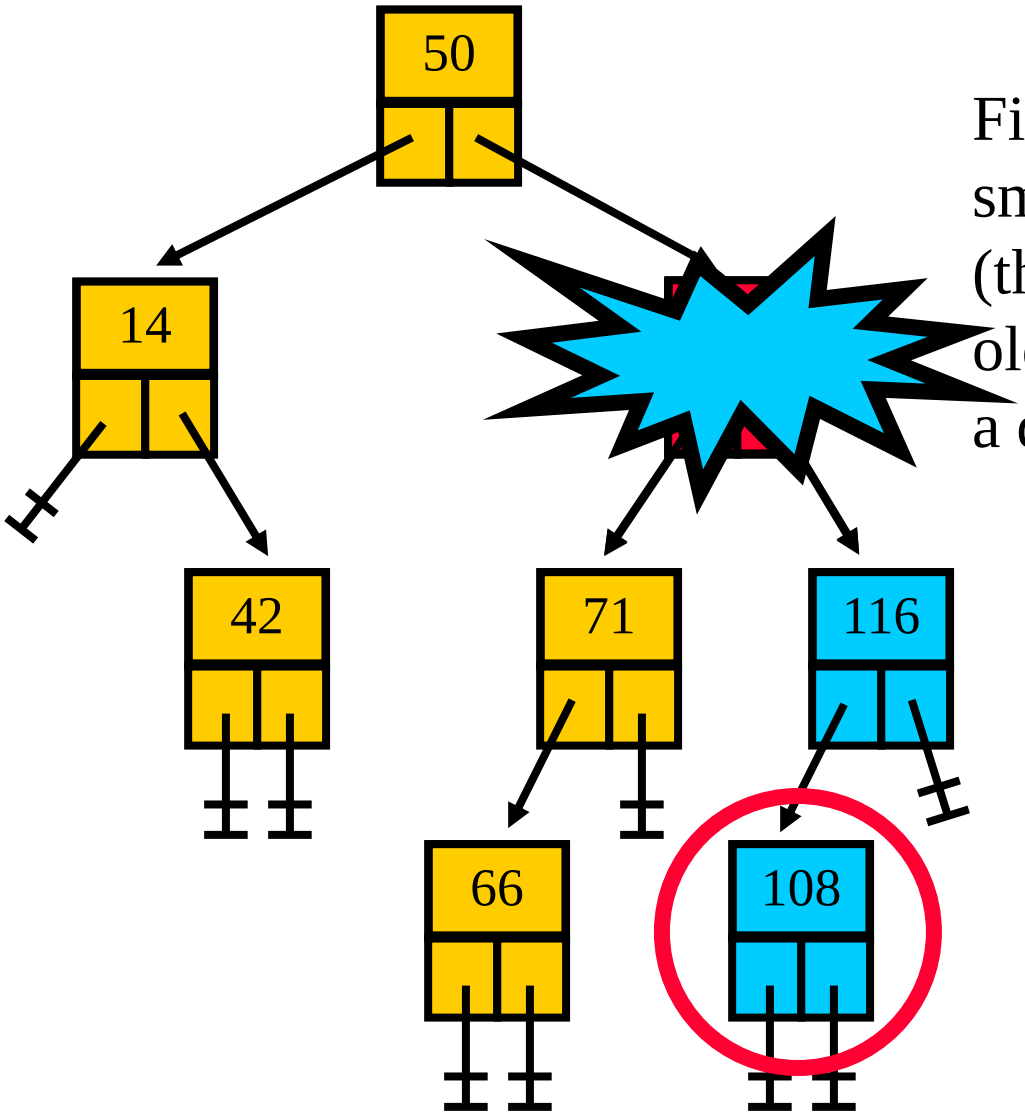
Let’s delete 94.

Delete a Node with Two Children



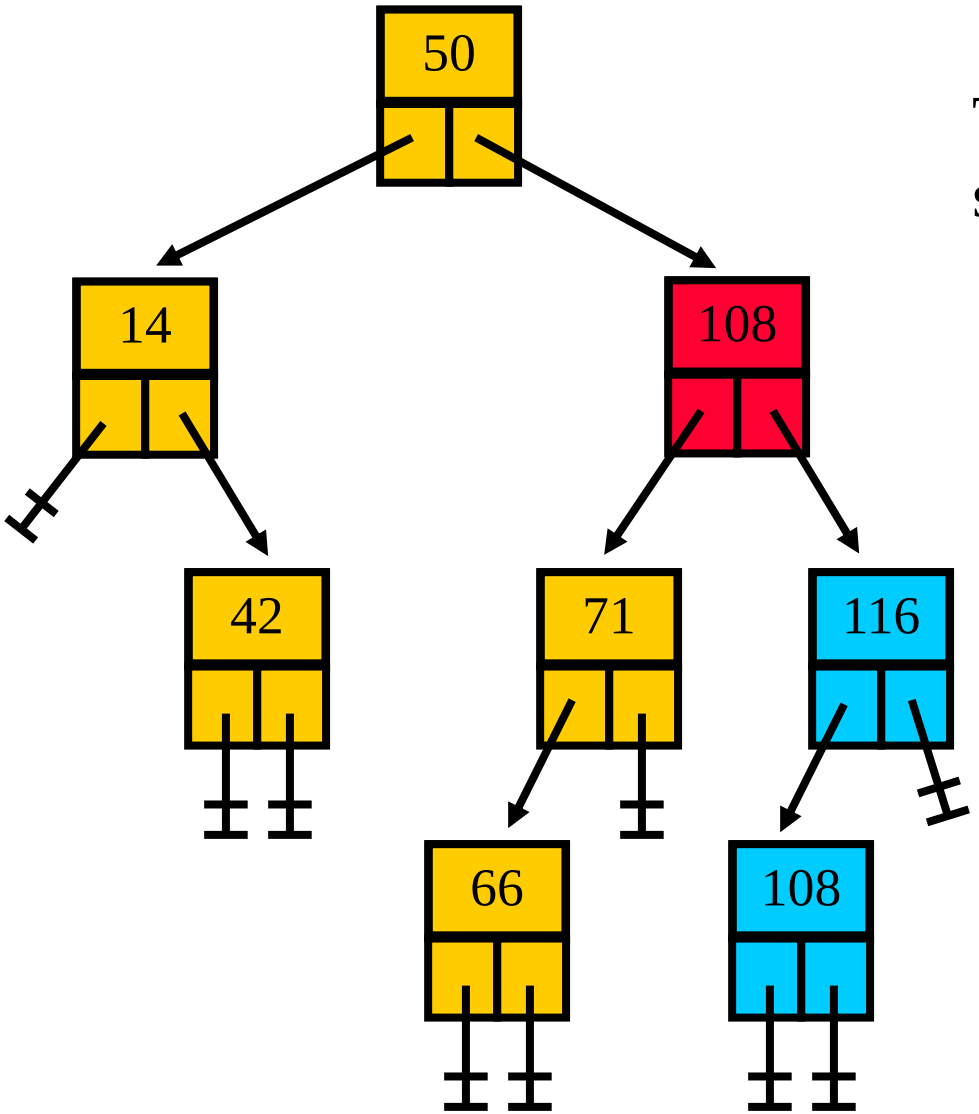
Look to the right sub-tree.

Delete a Node with Two Children



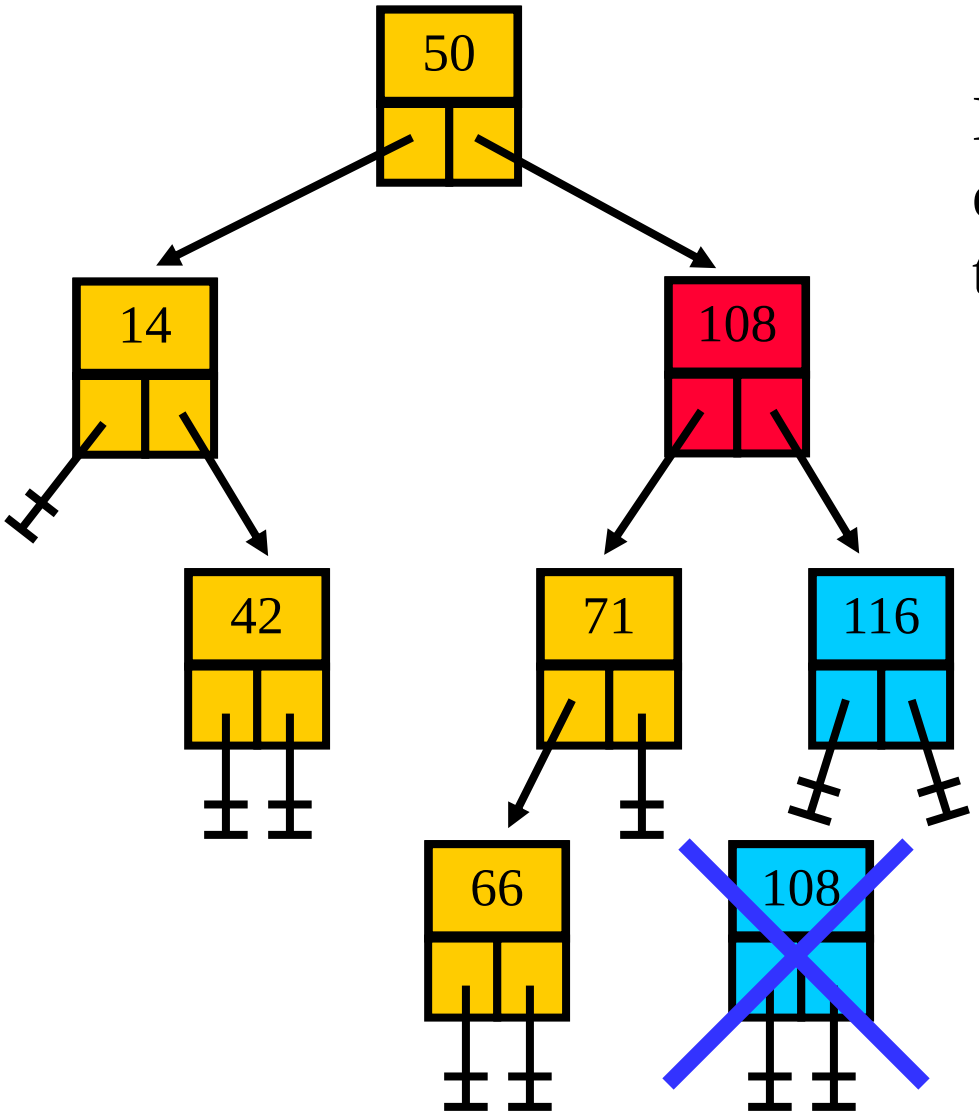
Find and copy the smallest value (this will erase the old value but creates a duplicate).

Delete a Node with Two Children



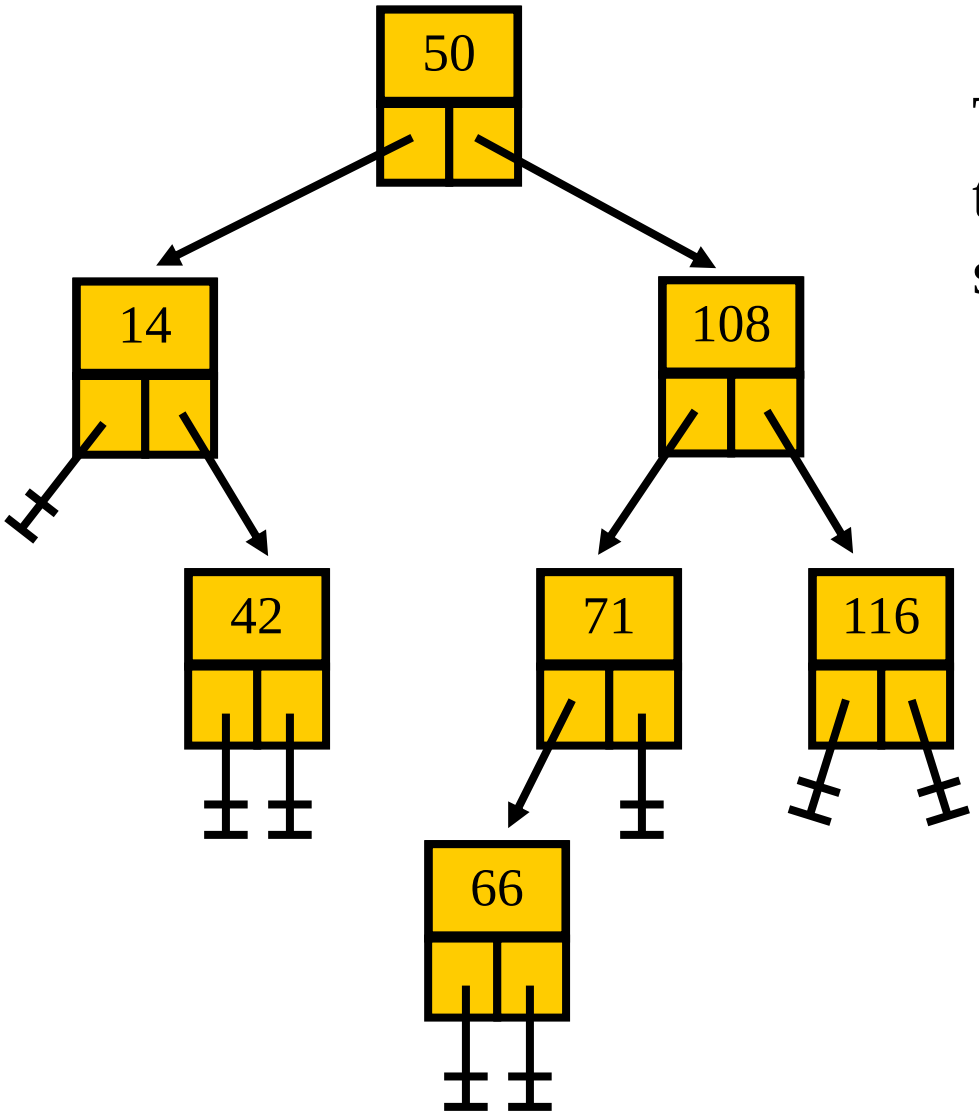
The resulting tree so far.

Delete a Node with Two Children



Now delete the duplicate from the left sub-tree.

Delete a Node with Two Children



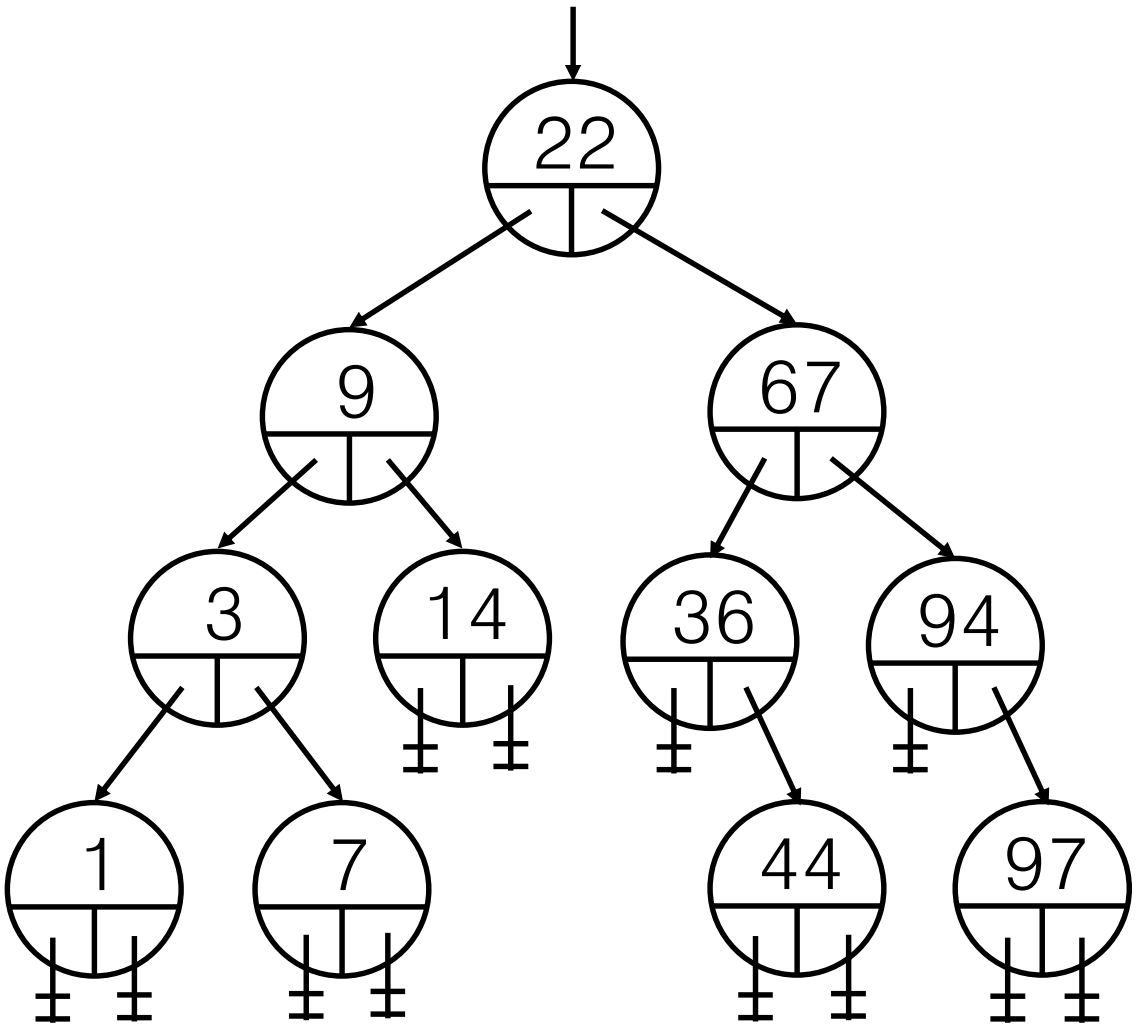
The final resulting tree – still has search structure.

노드 삭제(Delete Node)

A leaf node(자식이 없을 때)

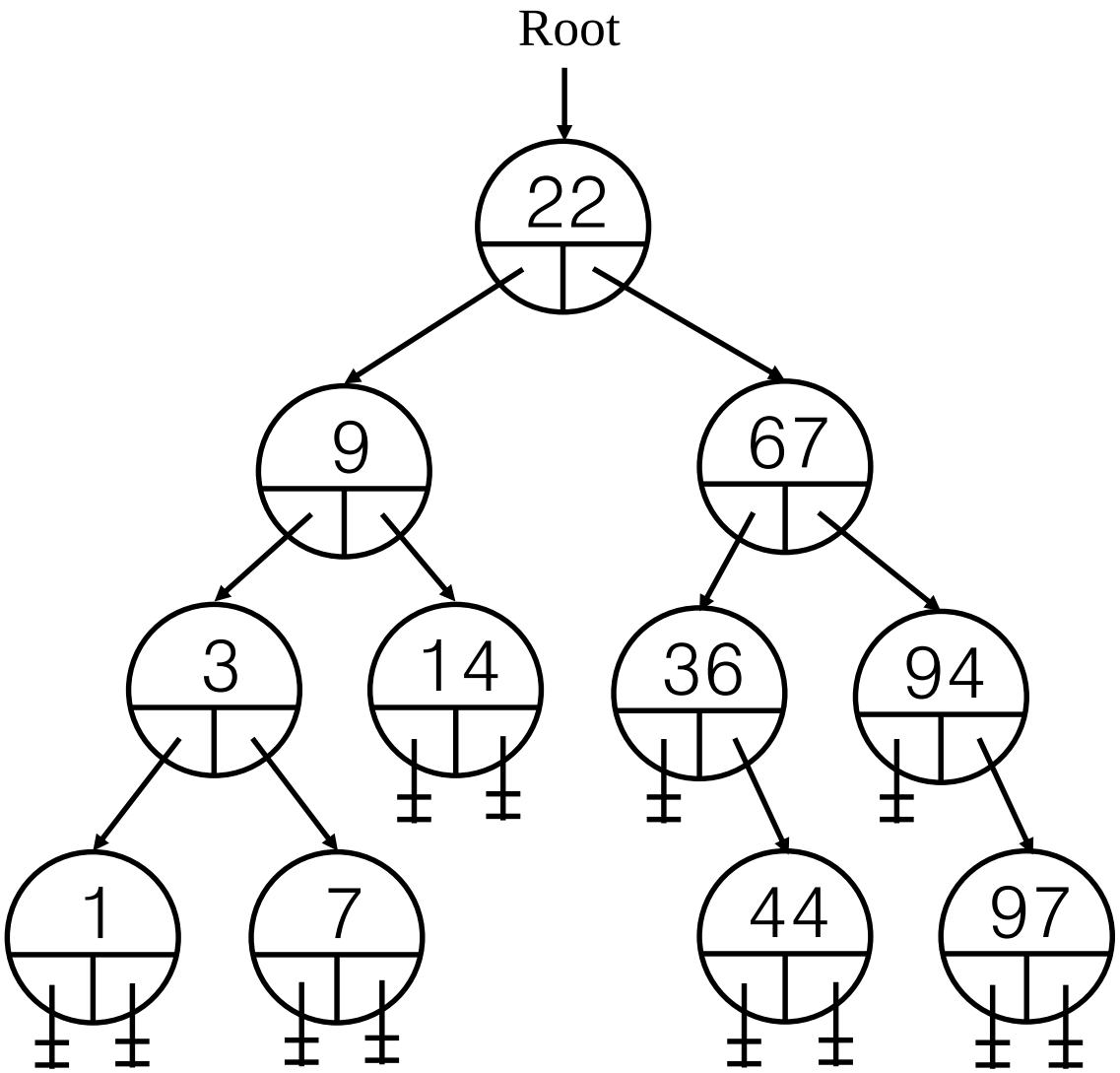
Delete a Leaf Node.

Root(Always search starts from the root)



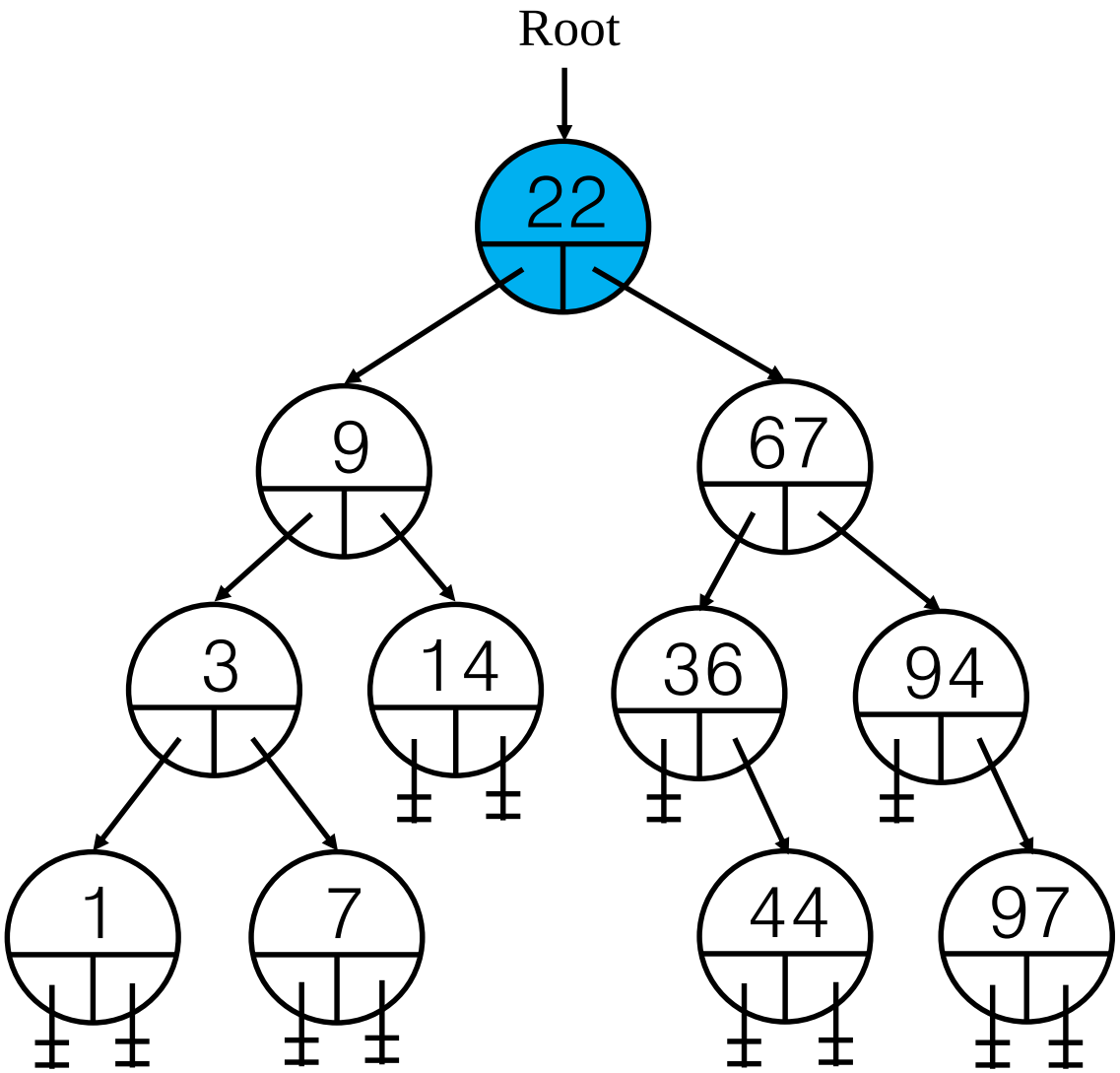
Delete a Leaf Node.

Let's Delete 14



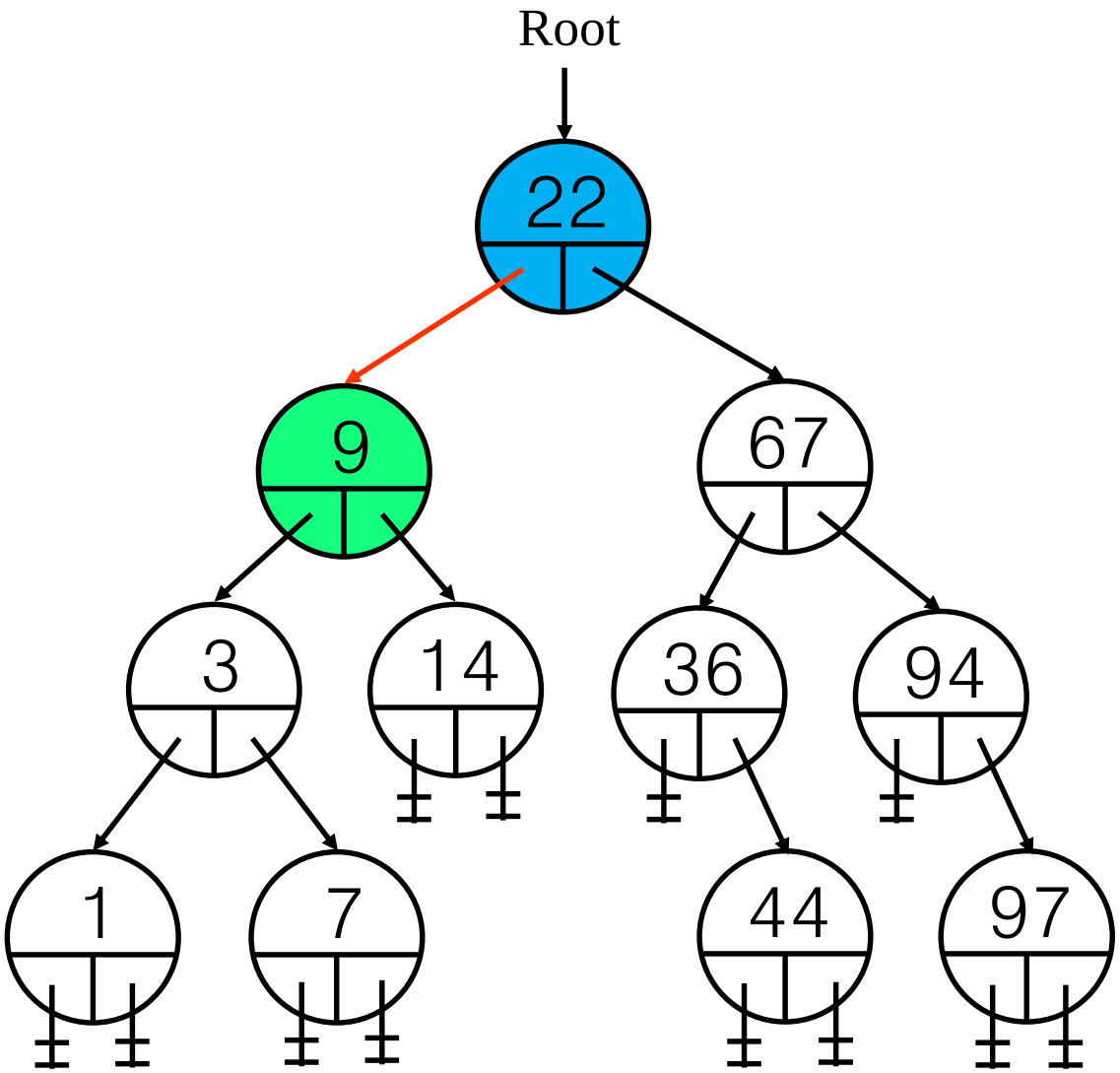
Delete a Leaf Node.

Let's Delete 14



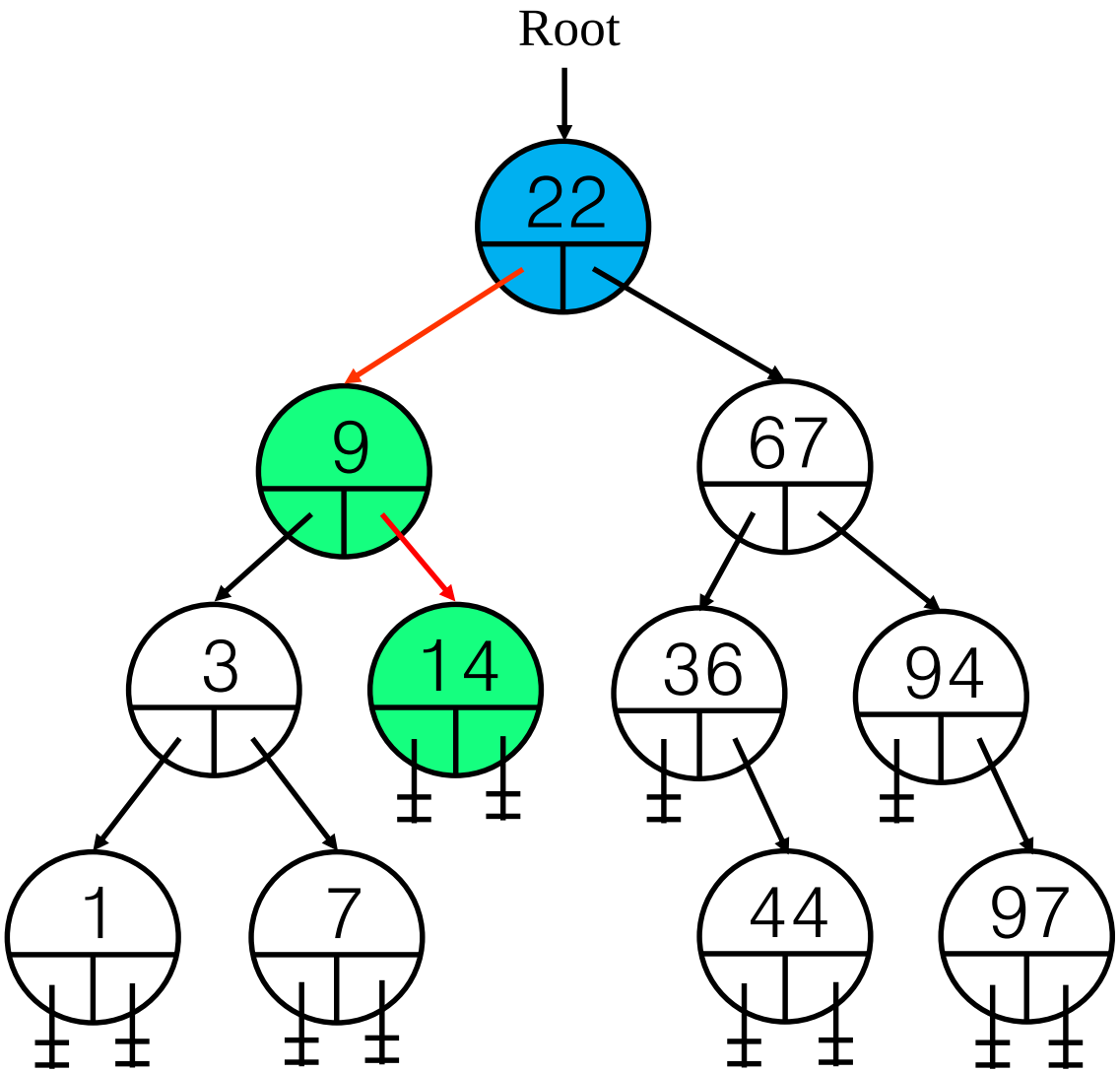
Delete a Leaf Node.

Let's Delete 14



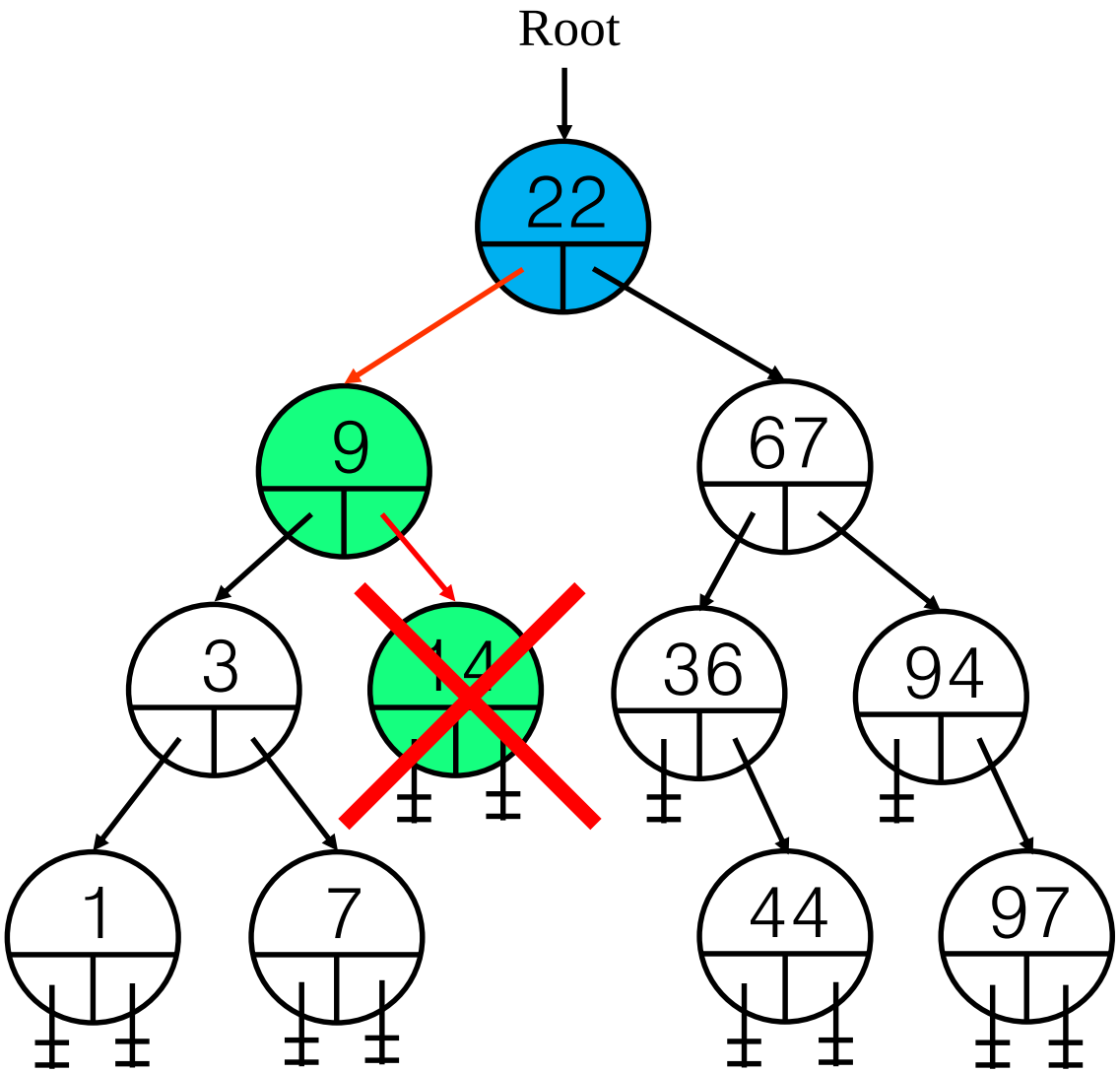
Delete a Leaf Node.

Let's Delete 14



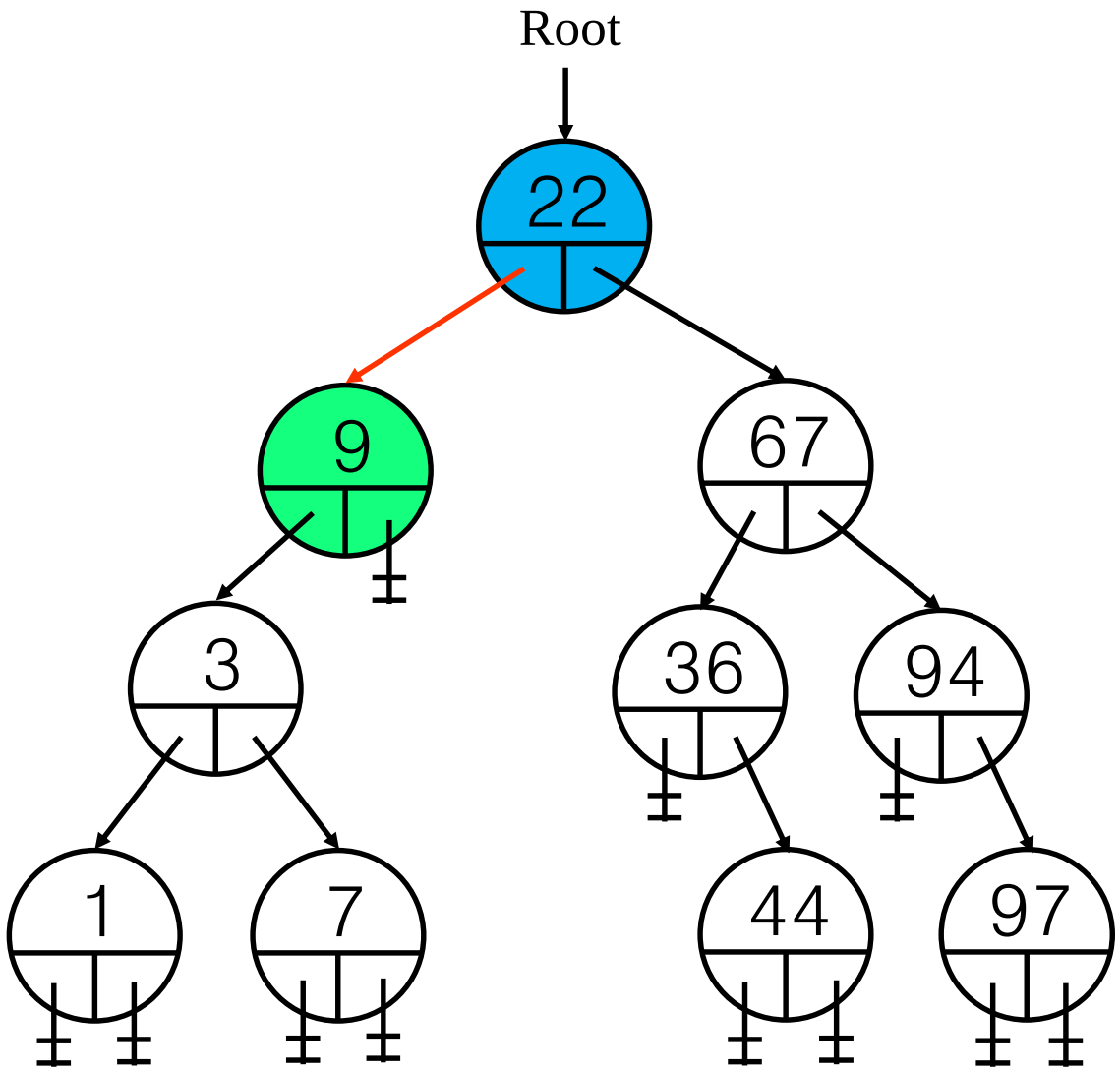
Delete a Leaf Node.

Let's Delete 14



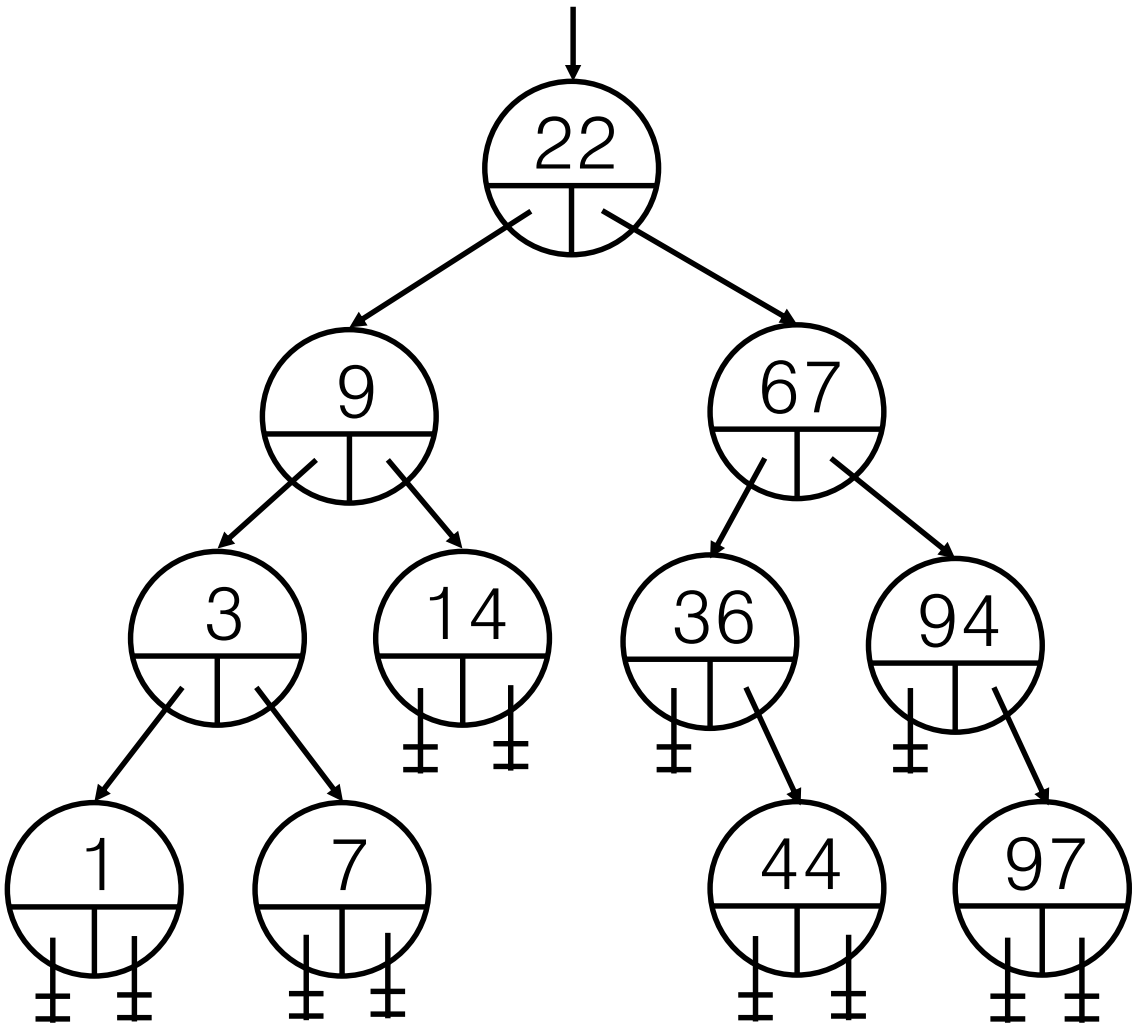
Delete a Leaf Node.

Let's Delete 14



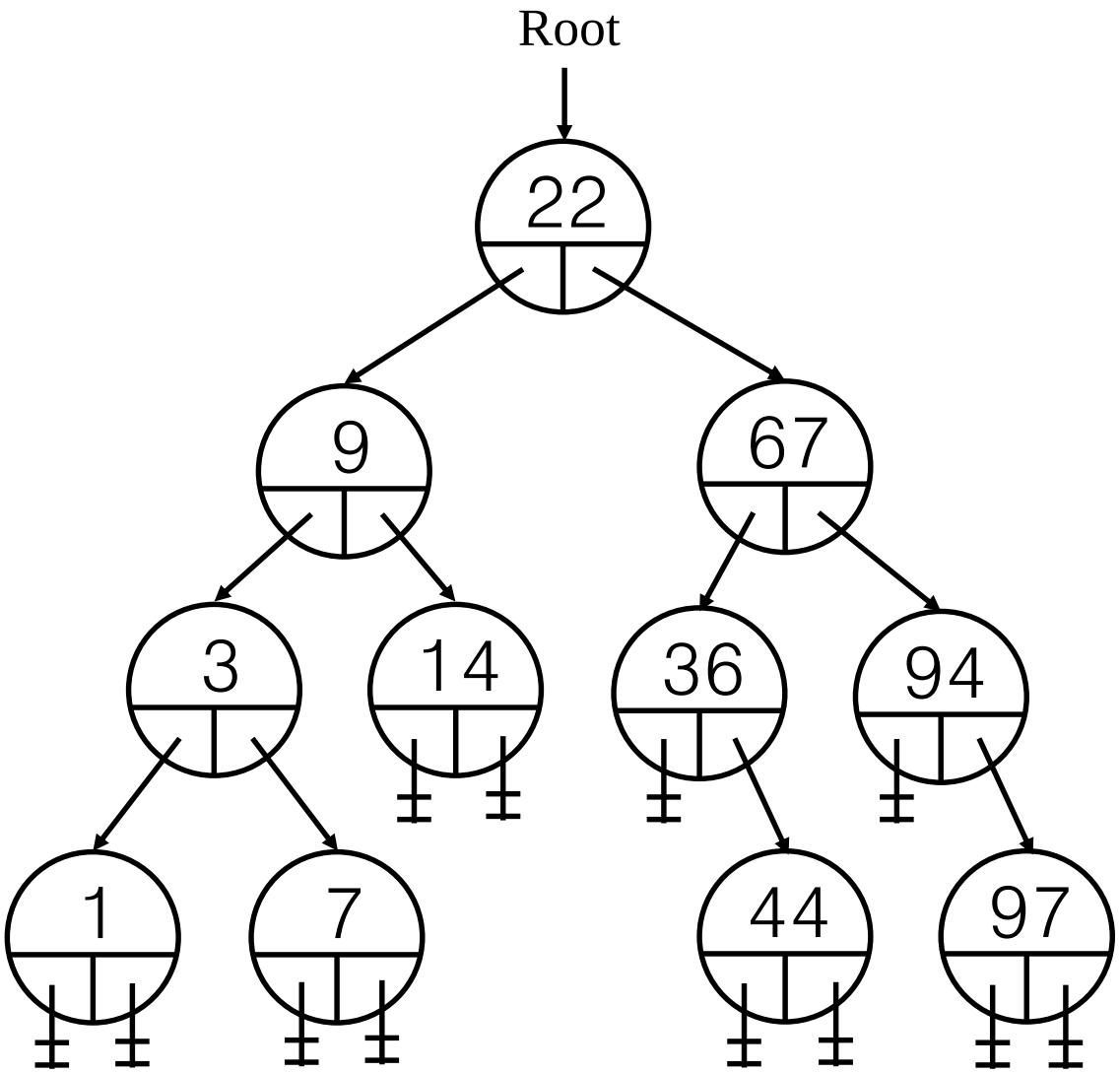
Delete a Leaf Node.

Root(Always search starts from the root)



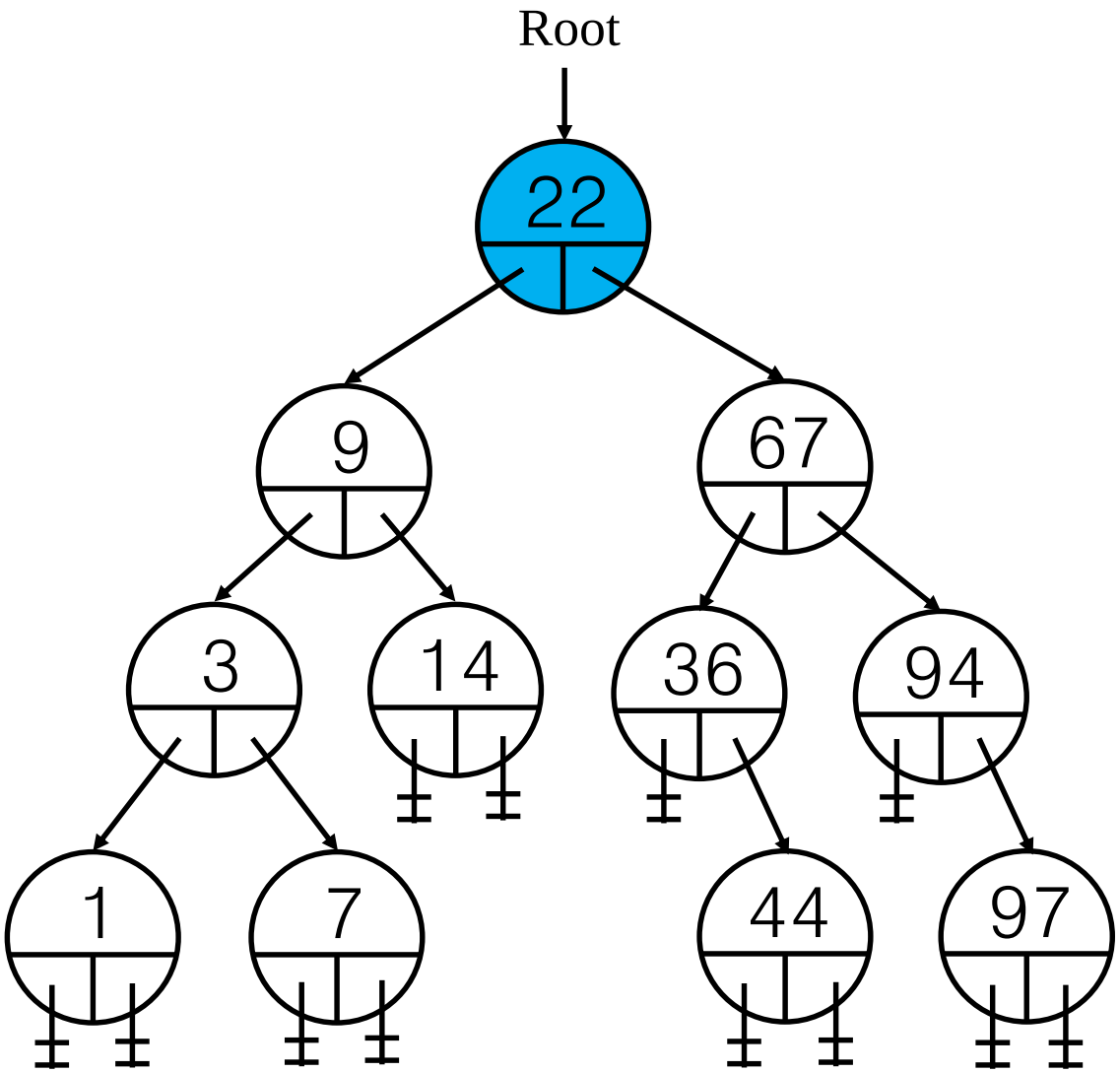
Delete a Leaf Node.

Let's Delete 44



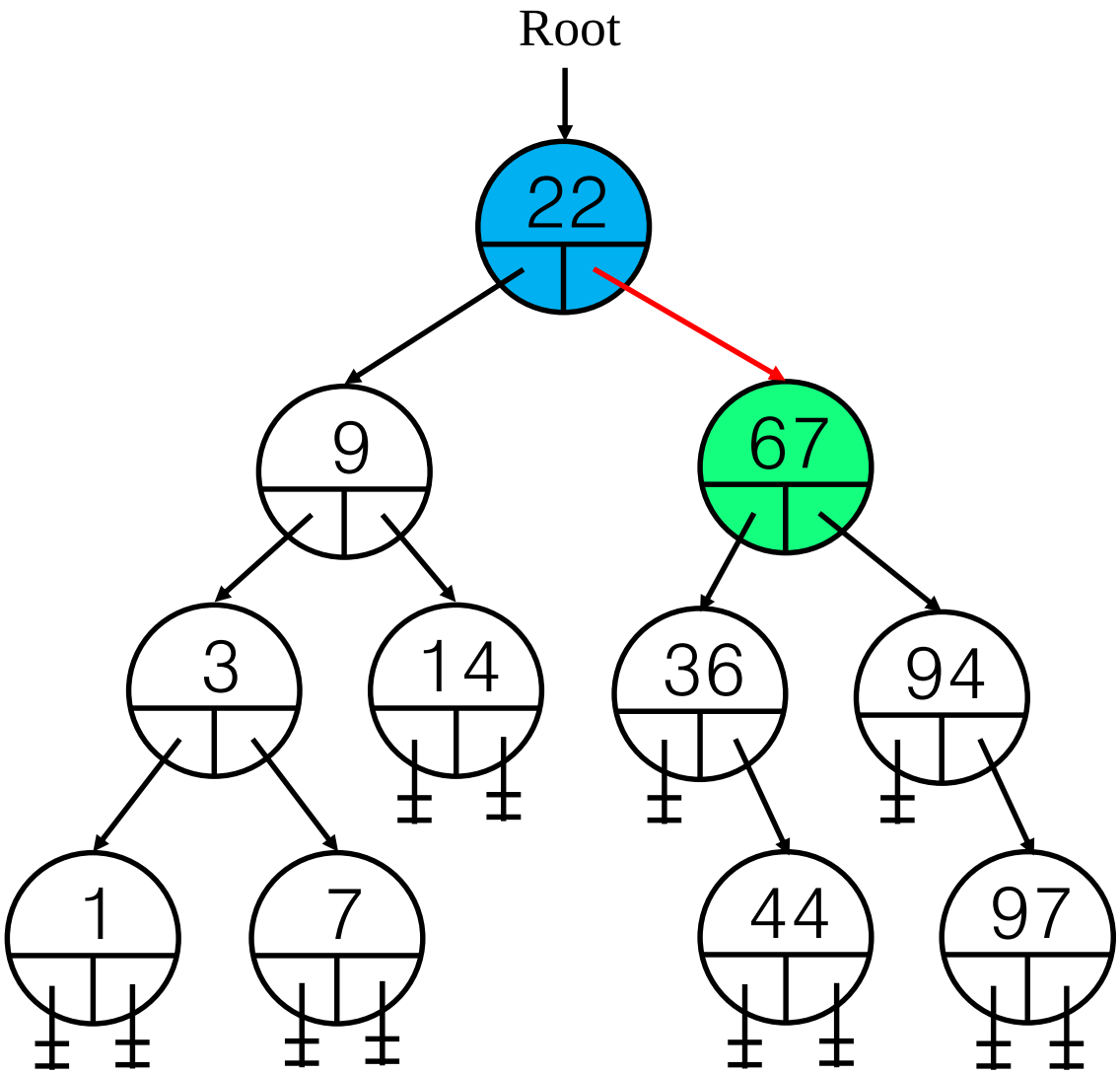
Delete a Leaf Node.

Let's Delete 44



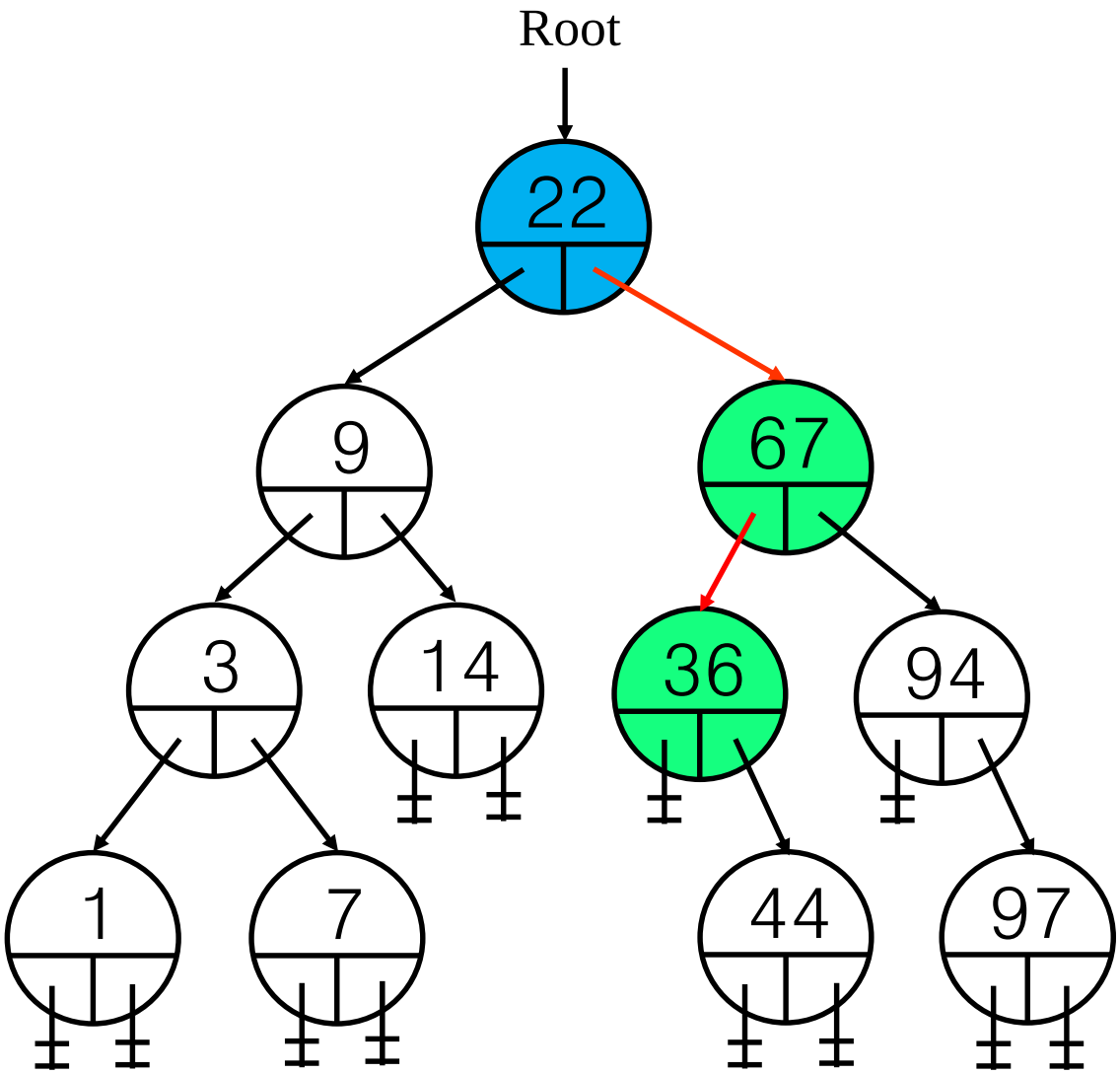
Delete a Leaf Node.

Let's Delete 44



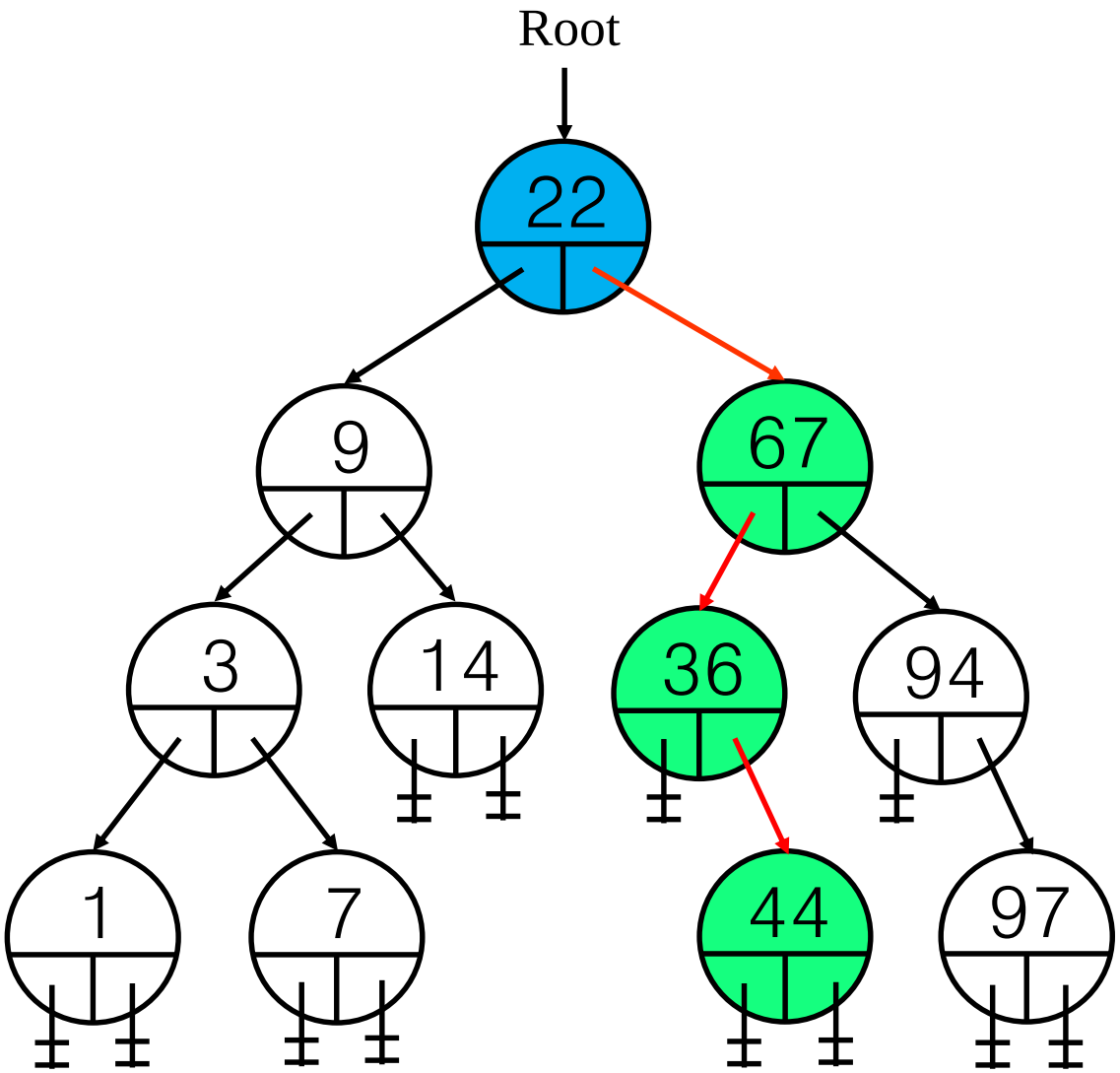
Delete a Leaf Node.

Let's Delete 44



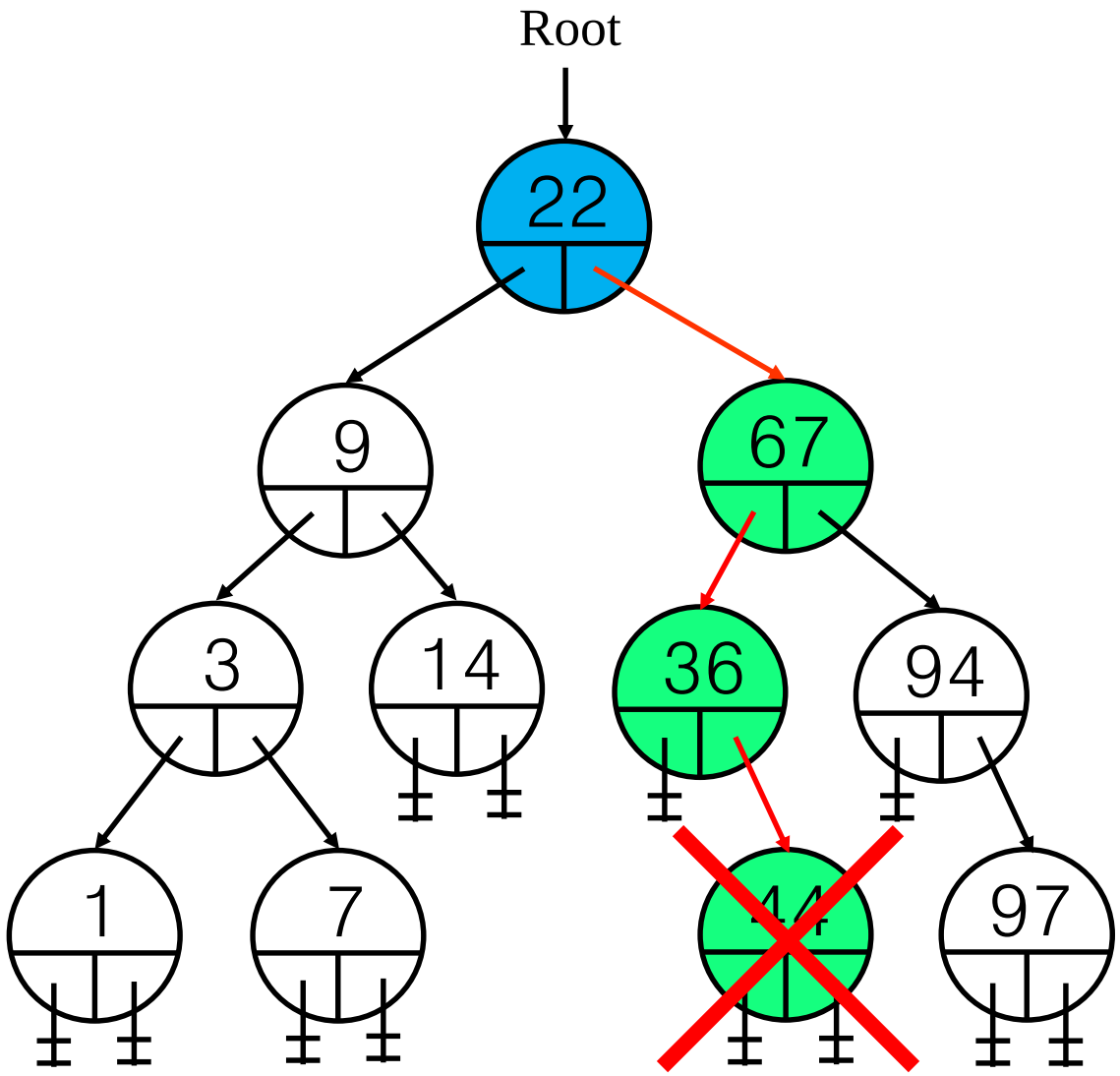
Delete a Leaf Node.

Let's Delete 44



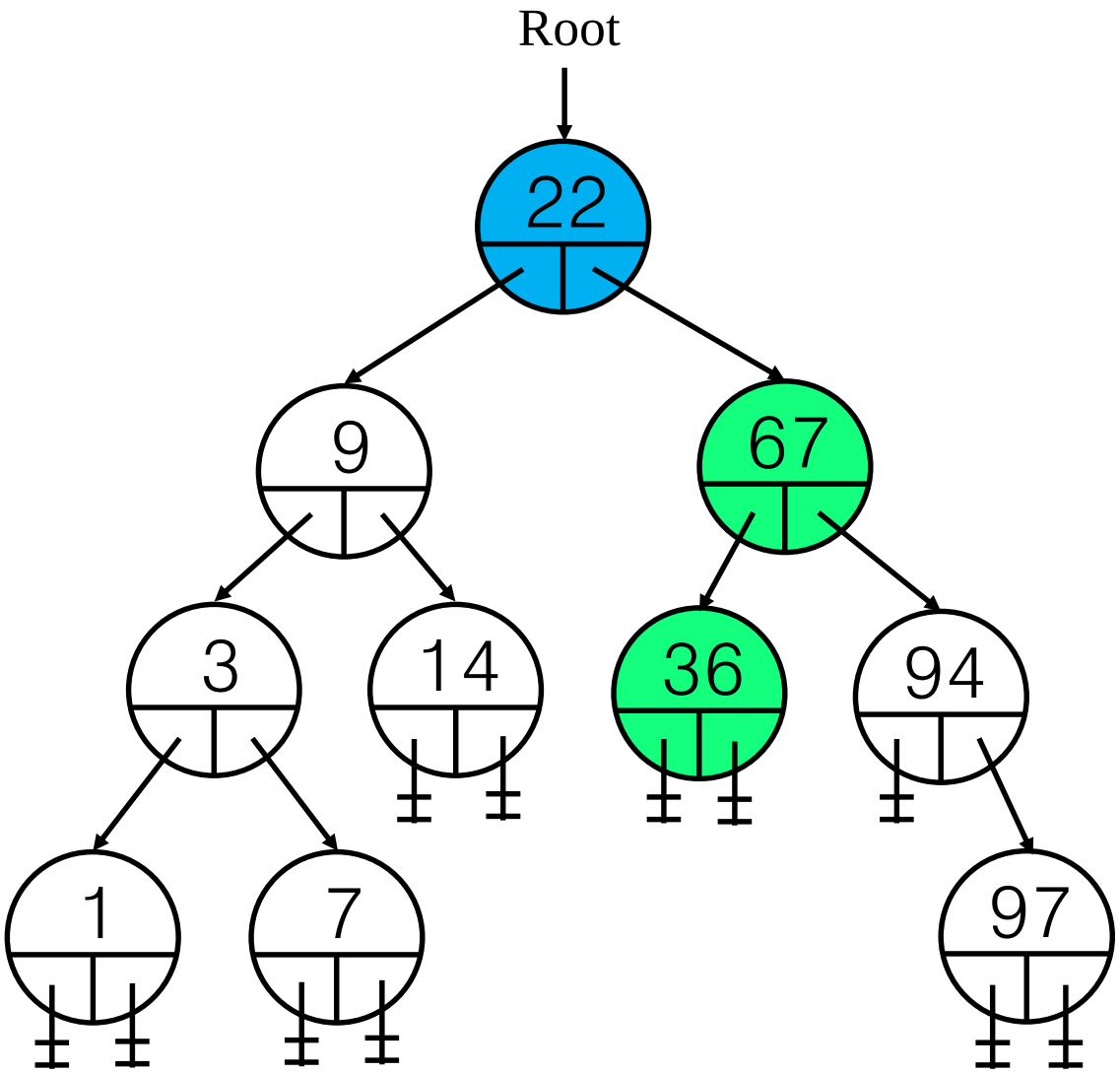
Delete a Leaf Node.

Let's Delete 44



Delete a Leaf Node.

Let's Delete 44

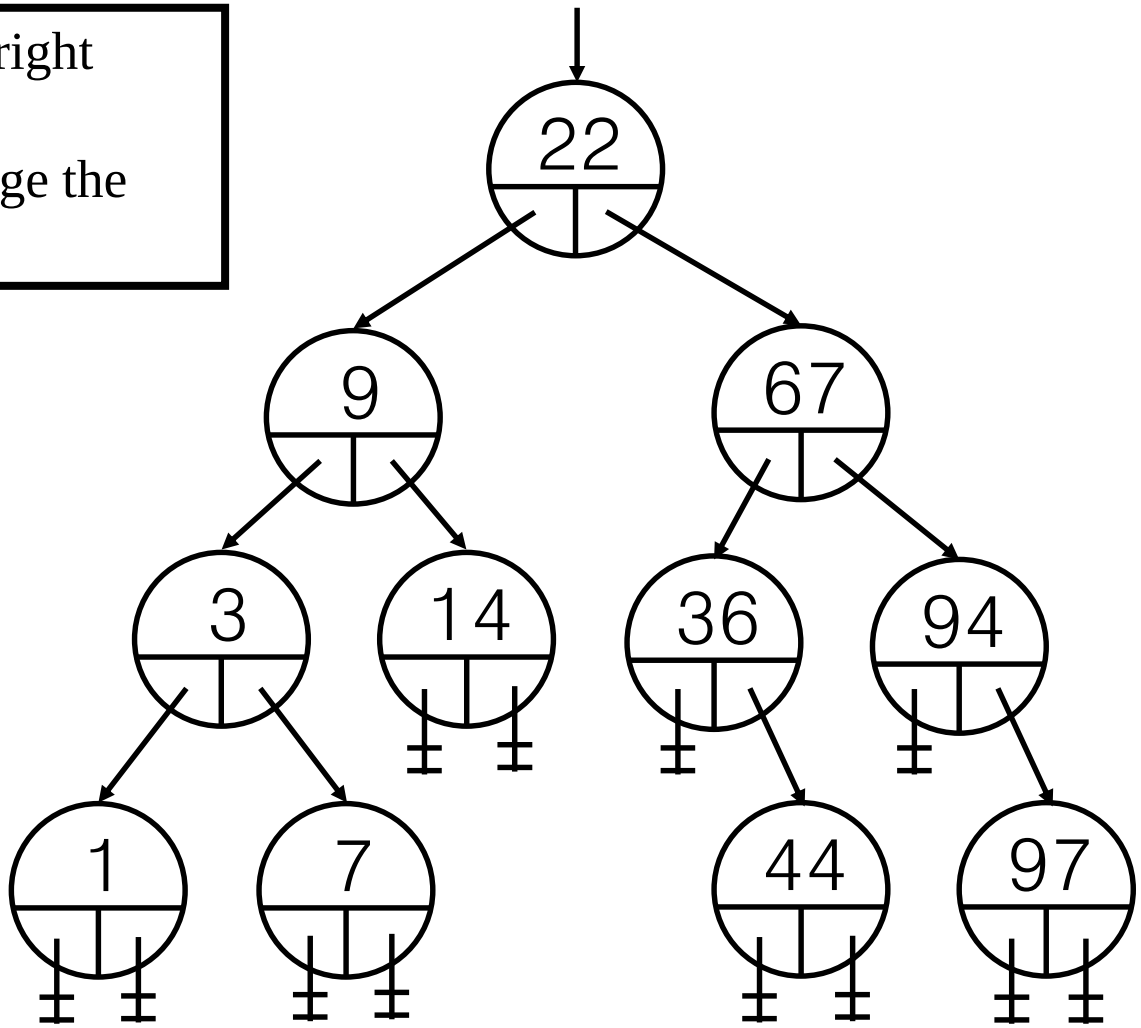


노드 삭제(Delete Node)

One child(자식이 하나일 때)

Root(Always search starts from the root)

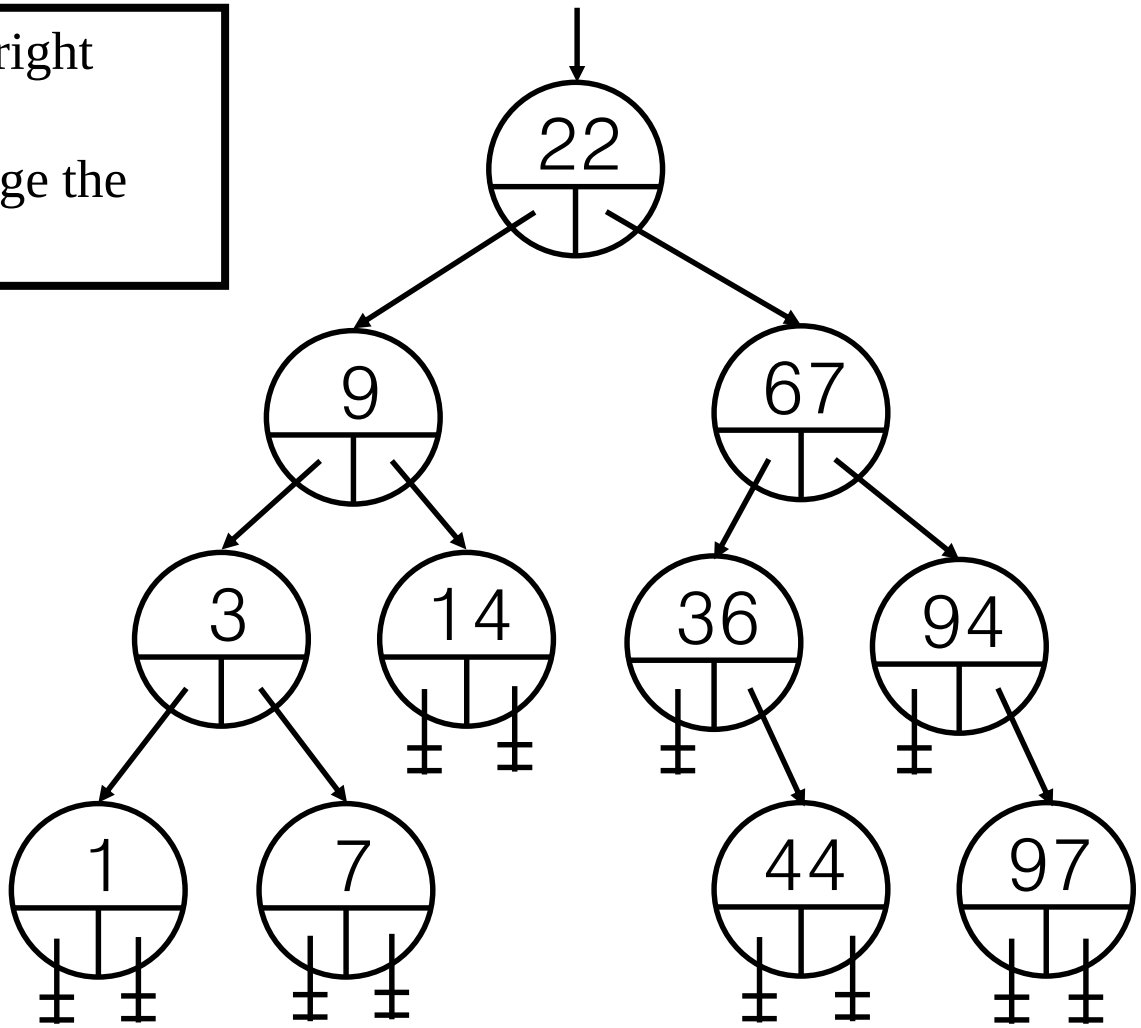
Determine if it has a left or a right child.
One child → delete and change the pointer to this child



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

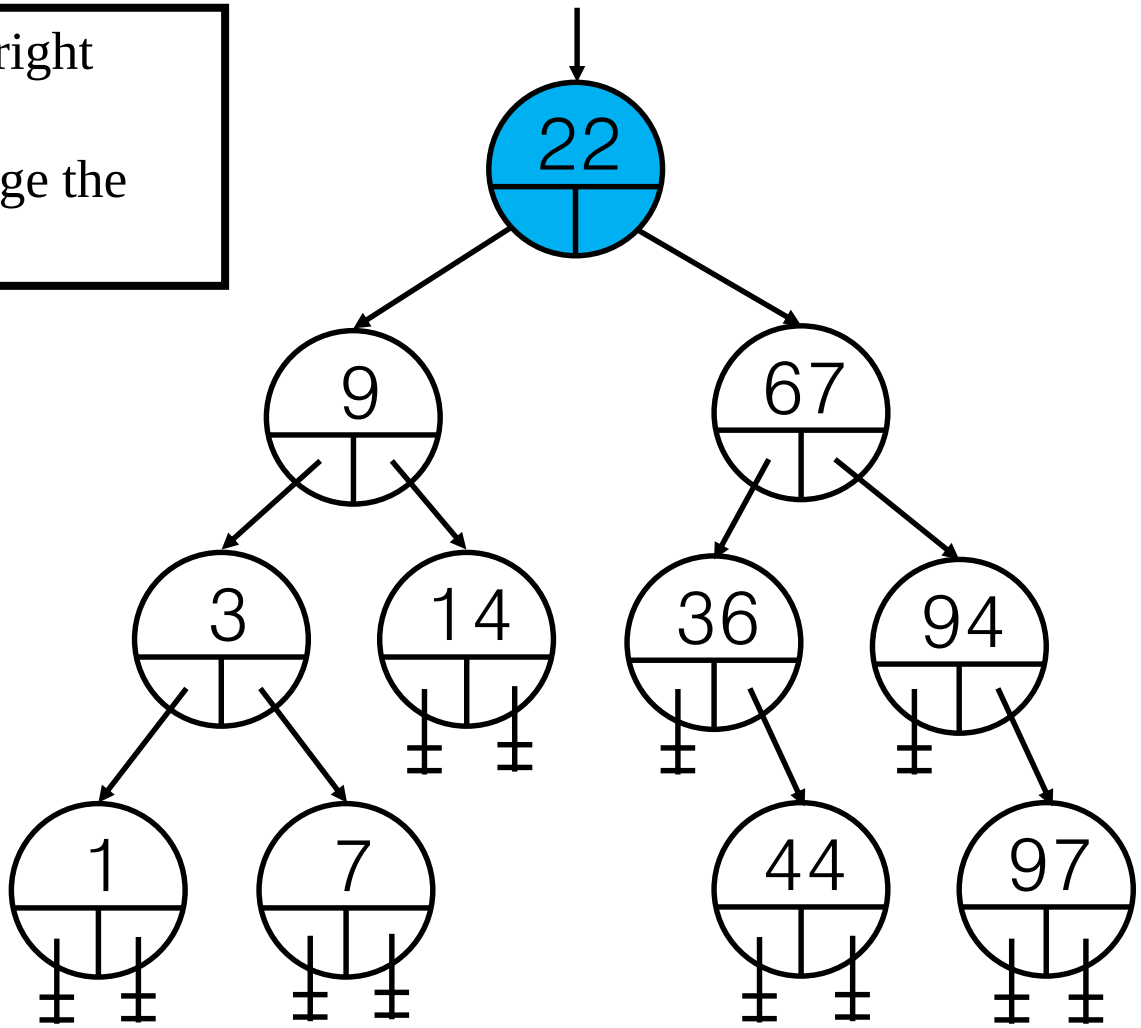
Let's Delete 36



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

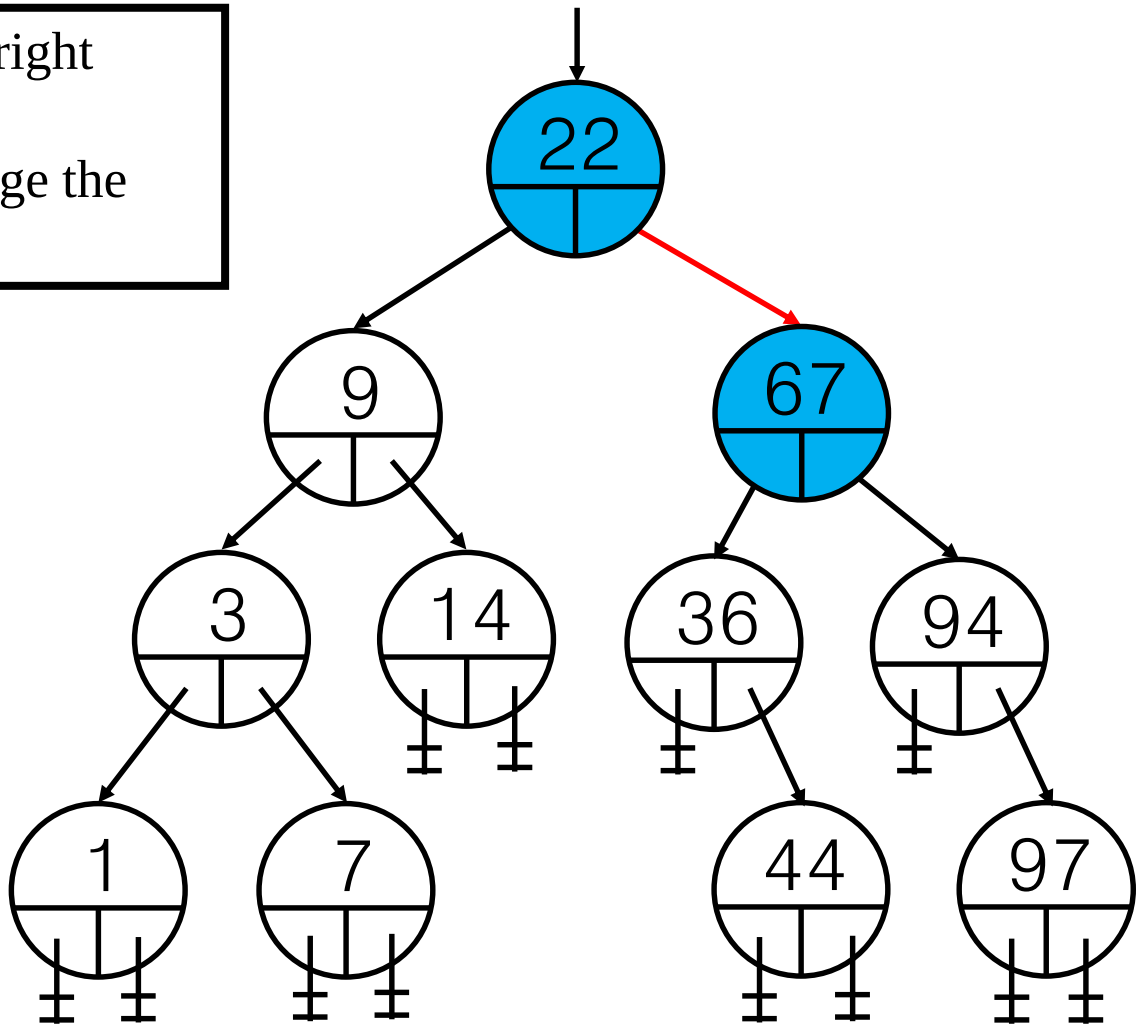
Let's Delete 36



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

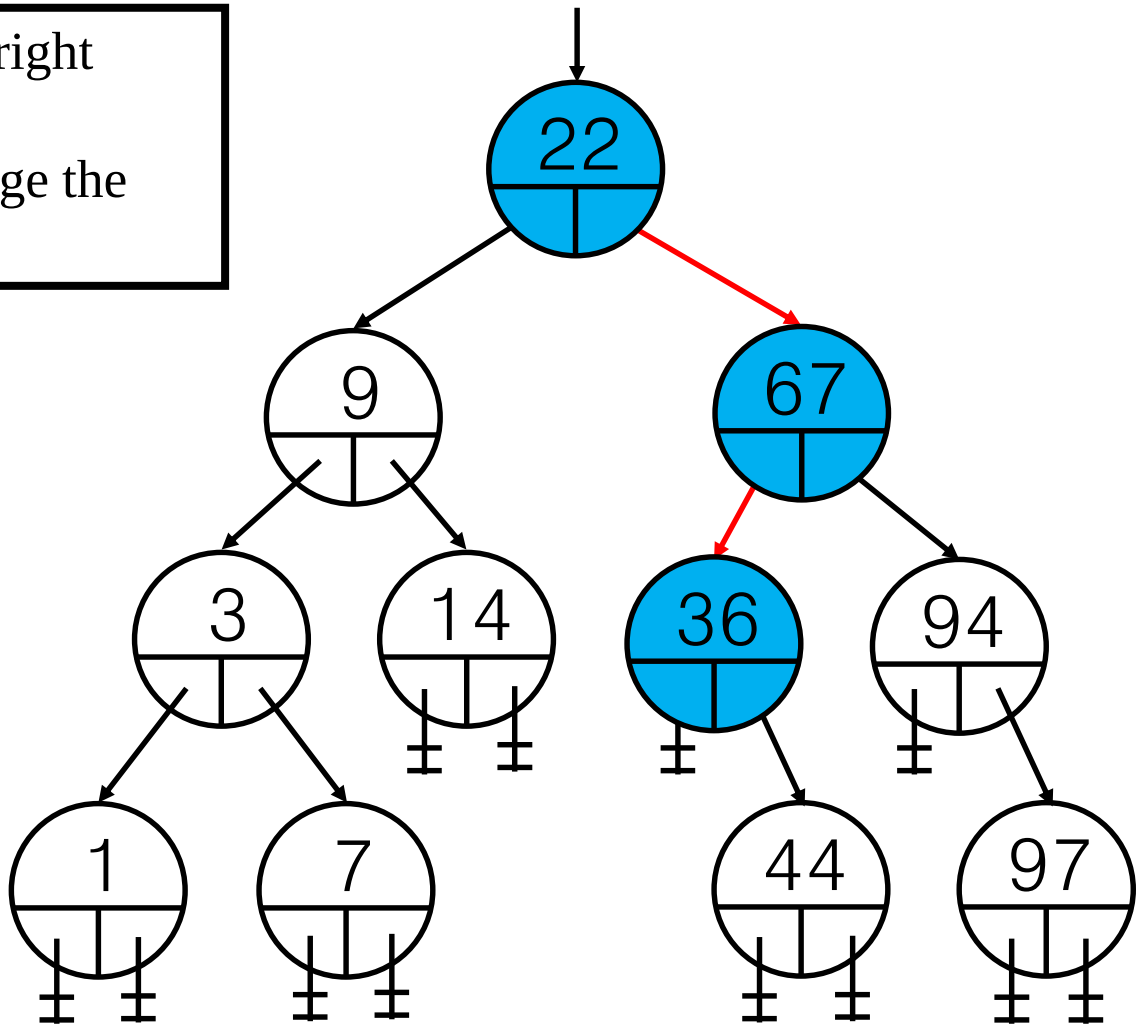
Let's Delete 36



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

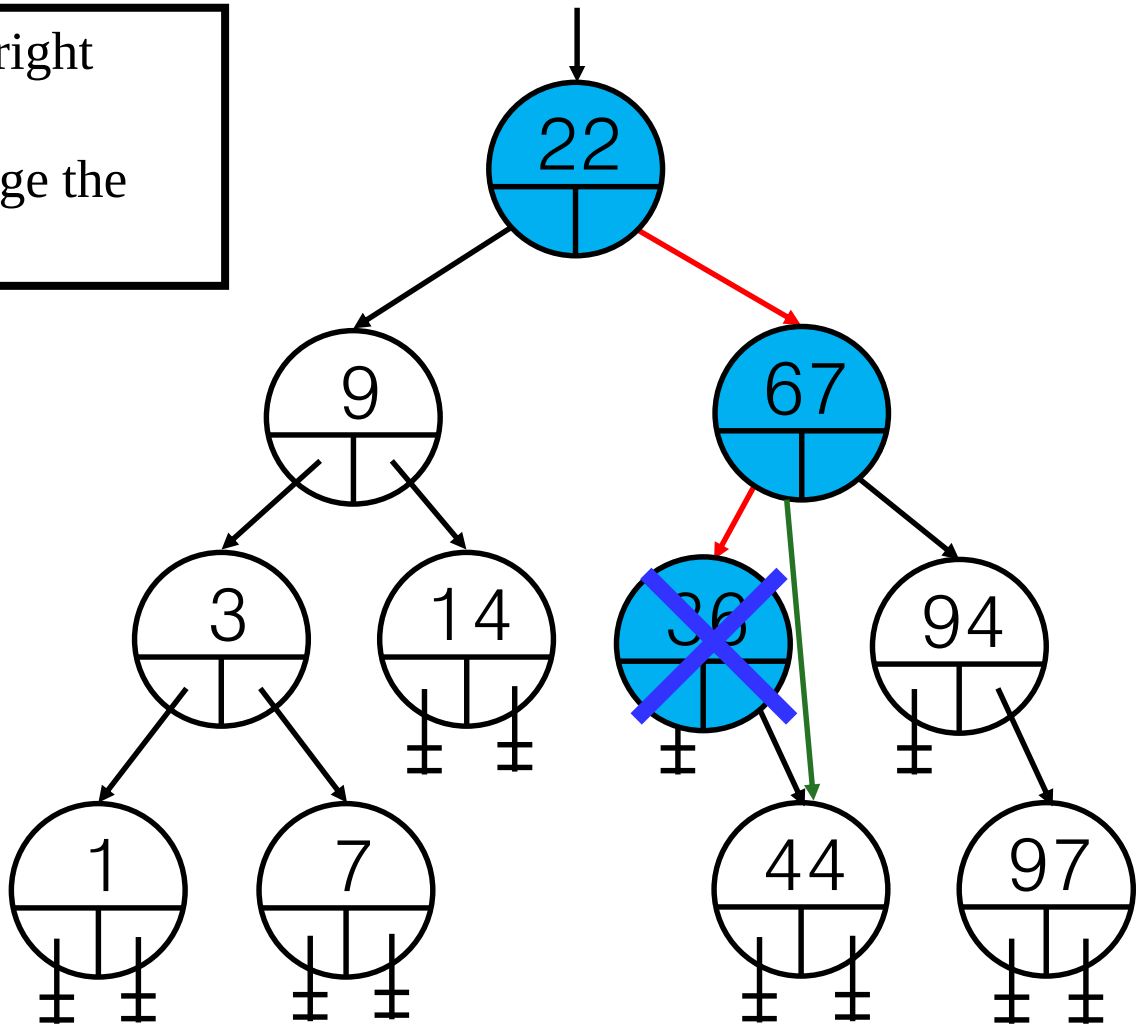
Let's Delete 36



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

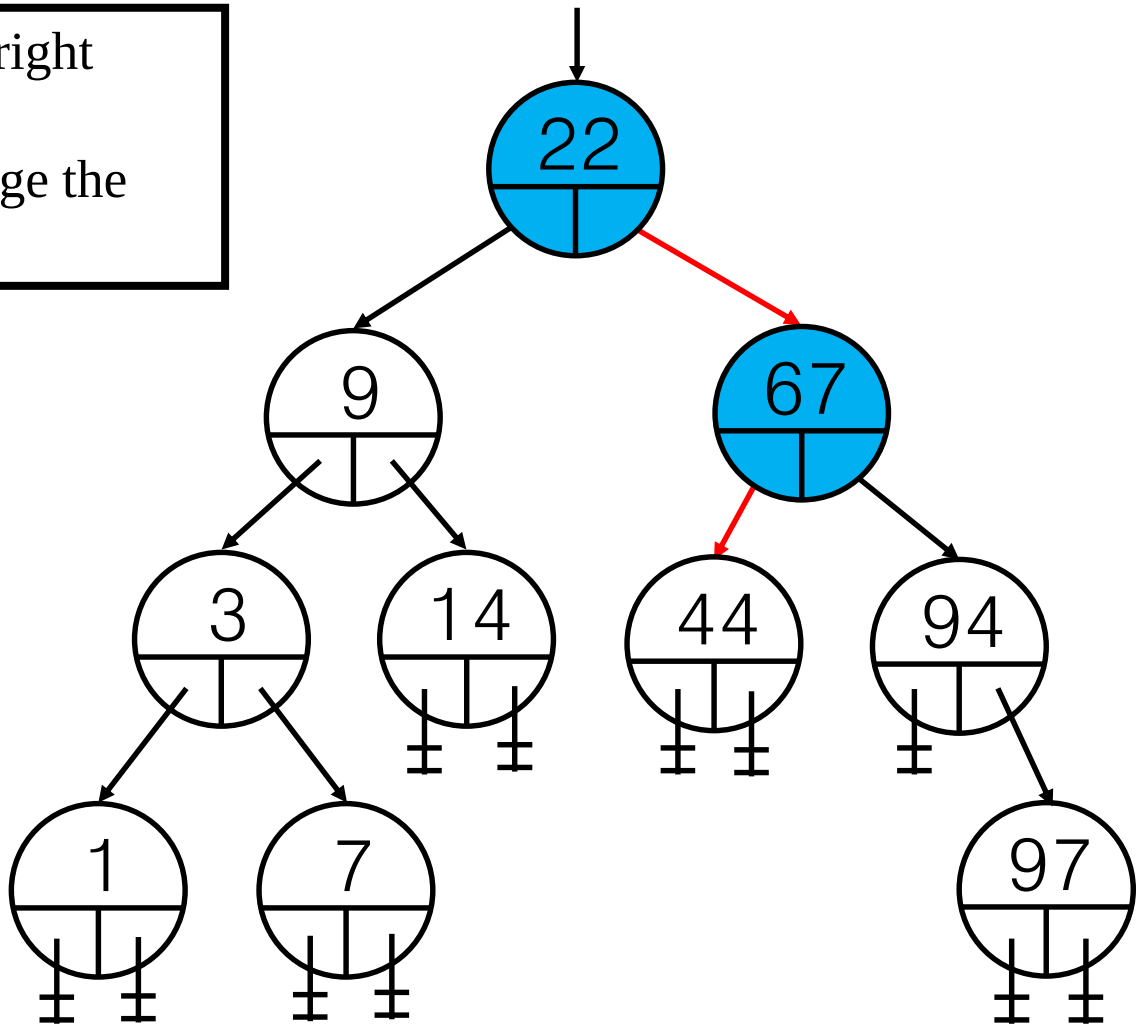
Let's Delete 36



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

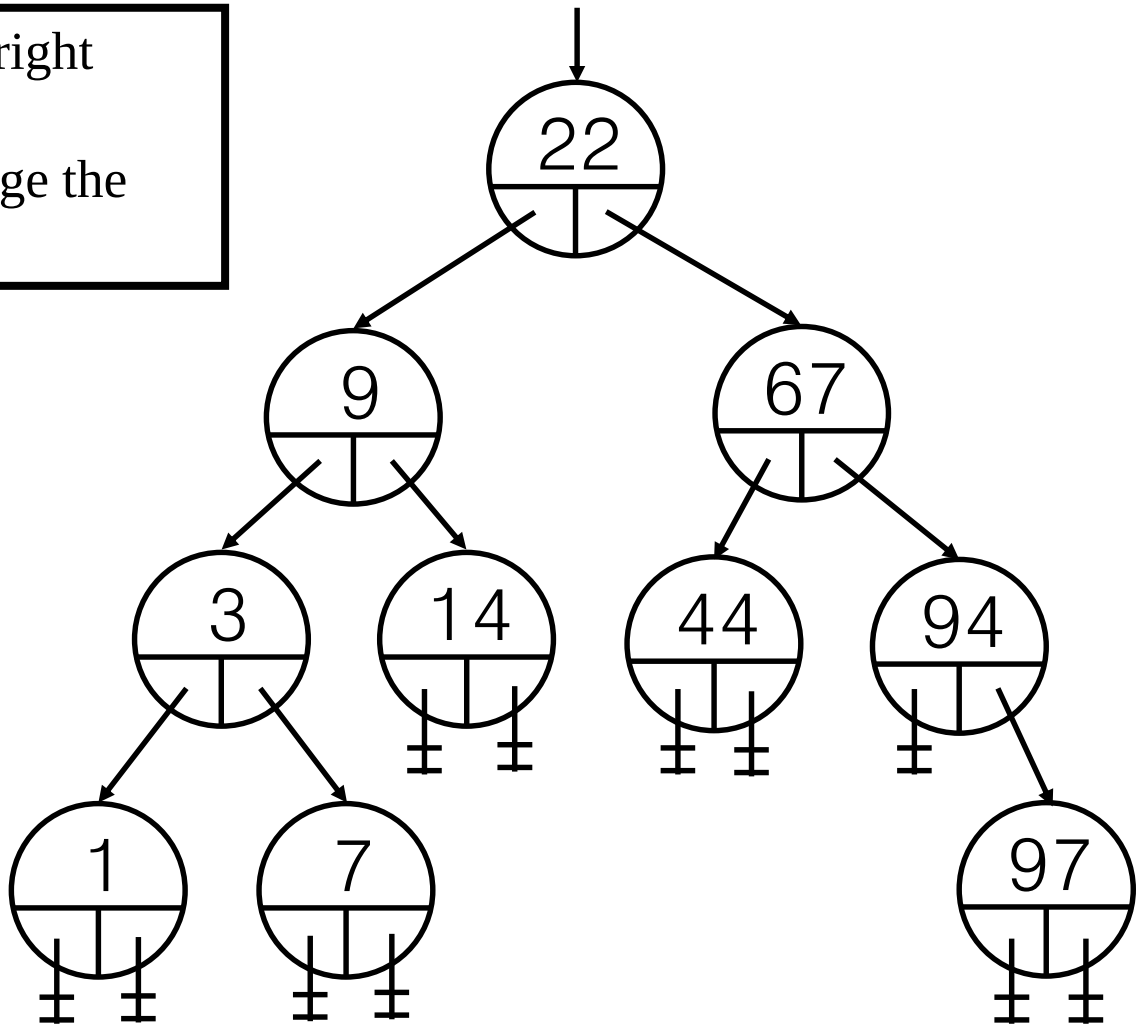
Let's Delete 36



Root(Always search starts from the root)

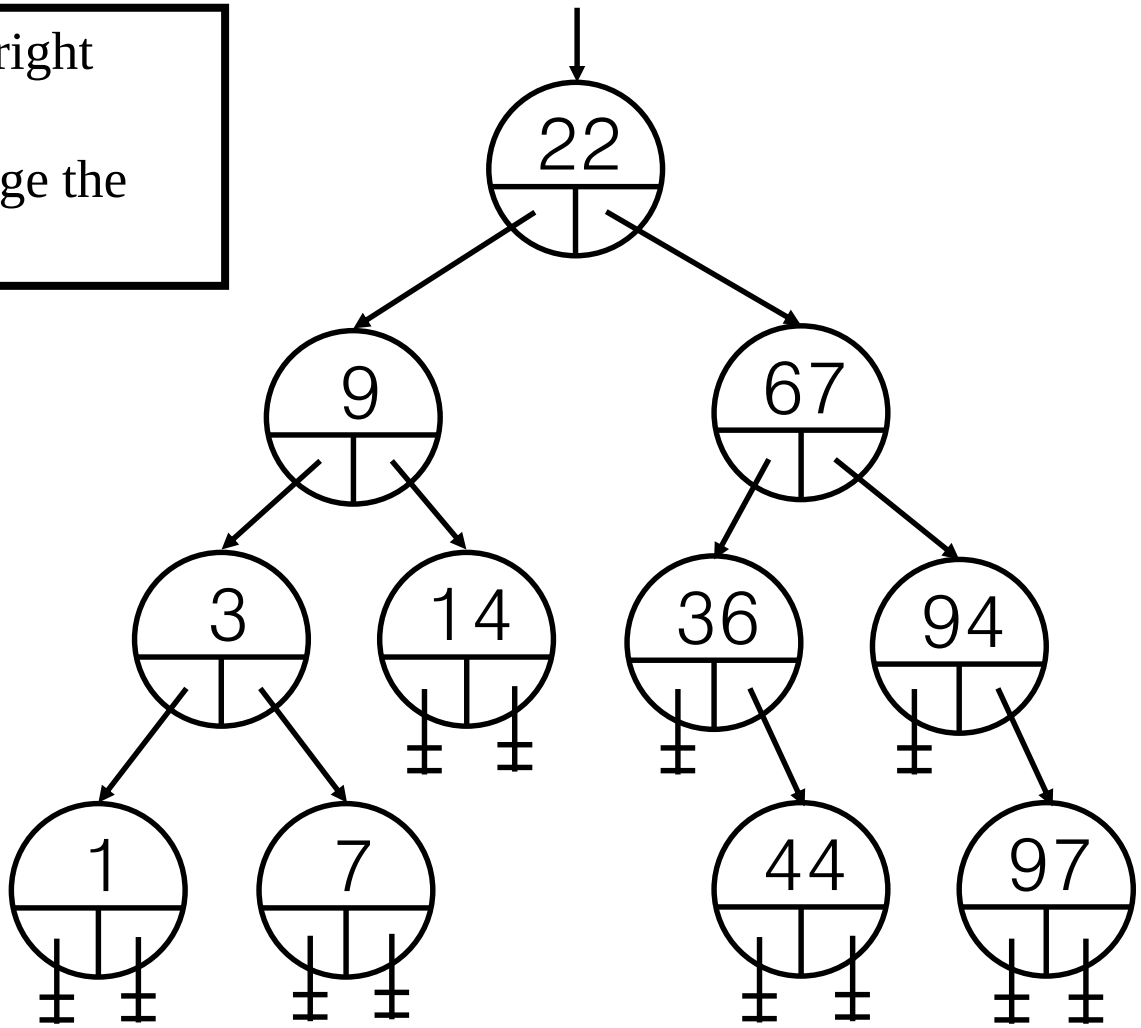
Determine if it has a left or a right child.
One child → delete and change the pointer to this child

The final resulting tree (최종트리)– still has search structure.(탐색구조)



Root(Always search starts from the root)

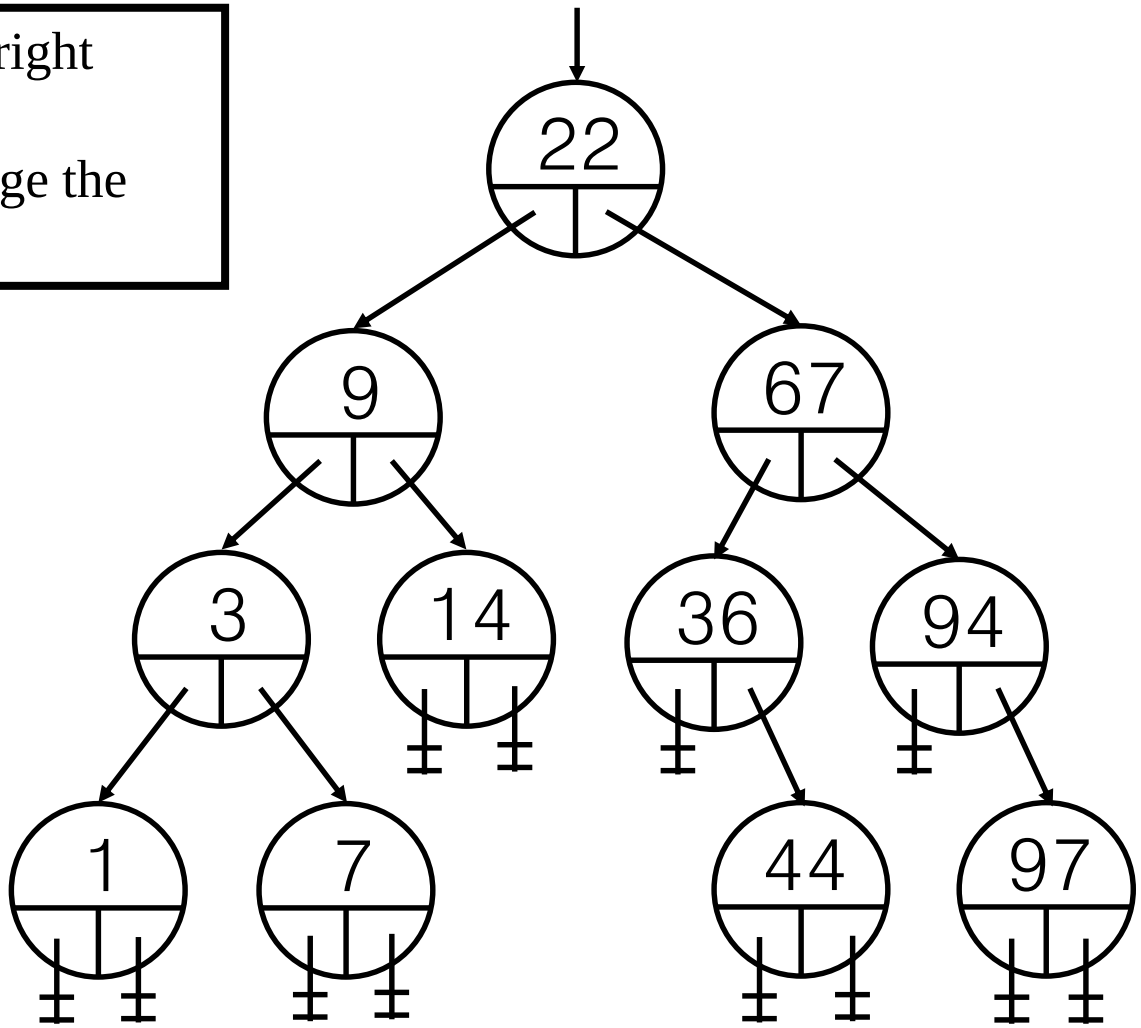
Determine if it has a left or a right child.
One child → delete and change the pointer to this child



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

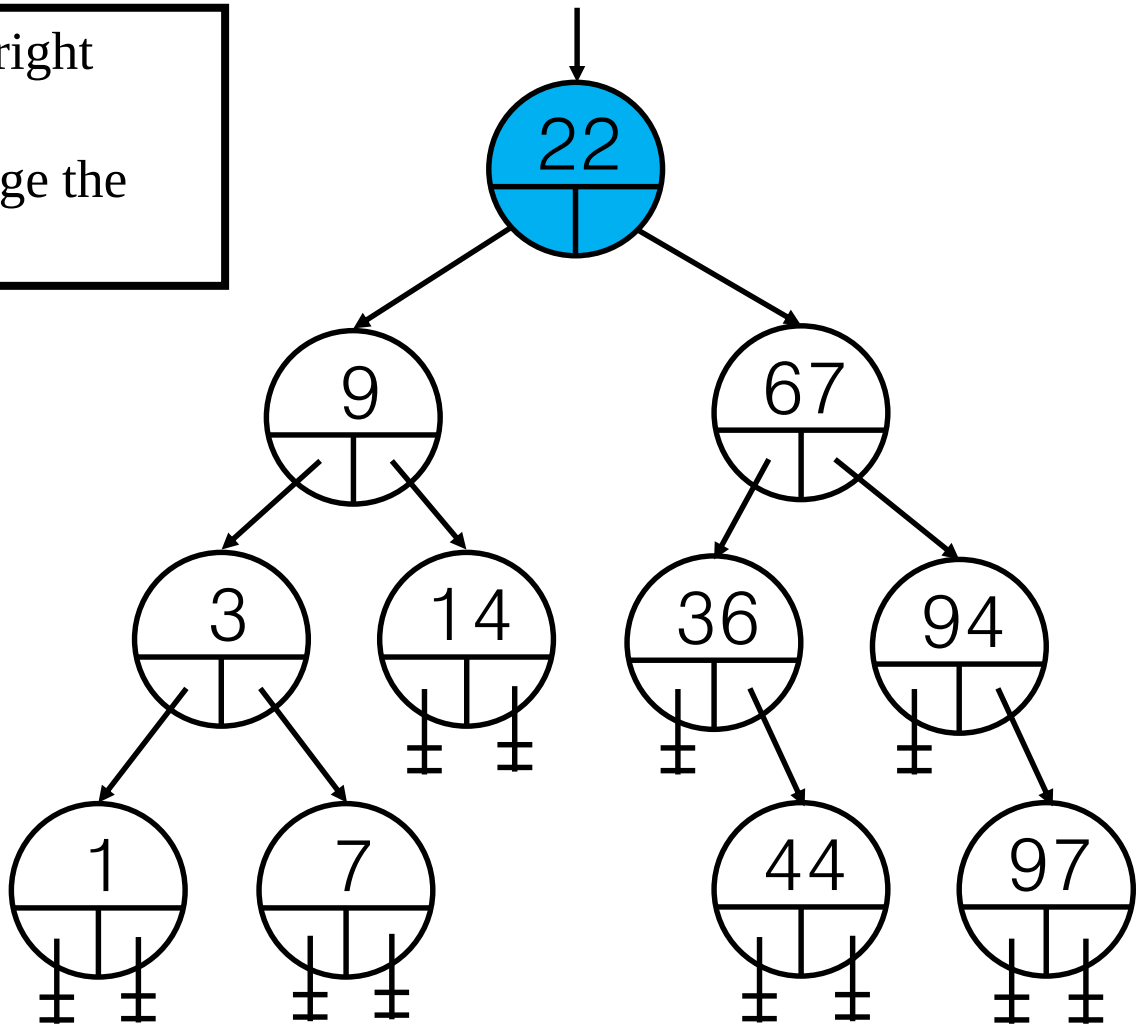
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

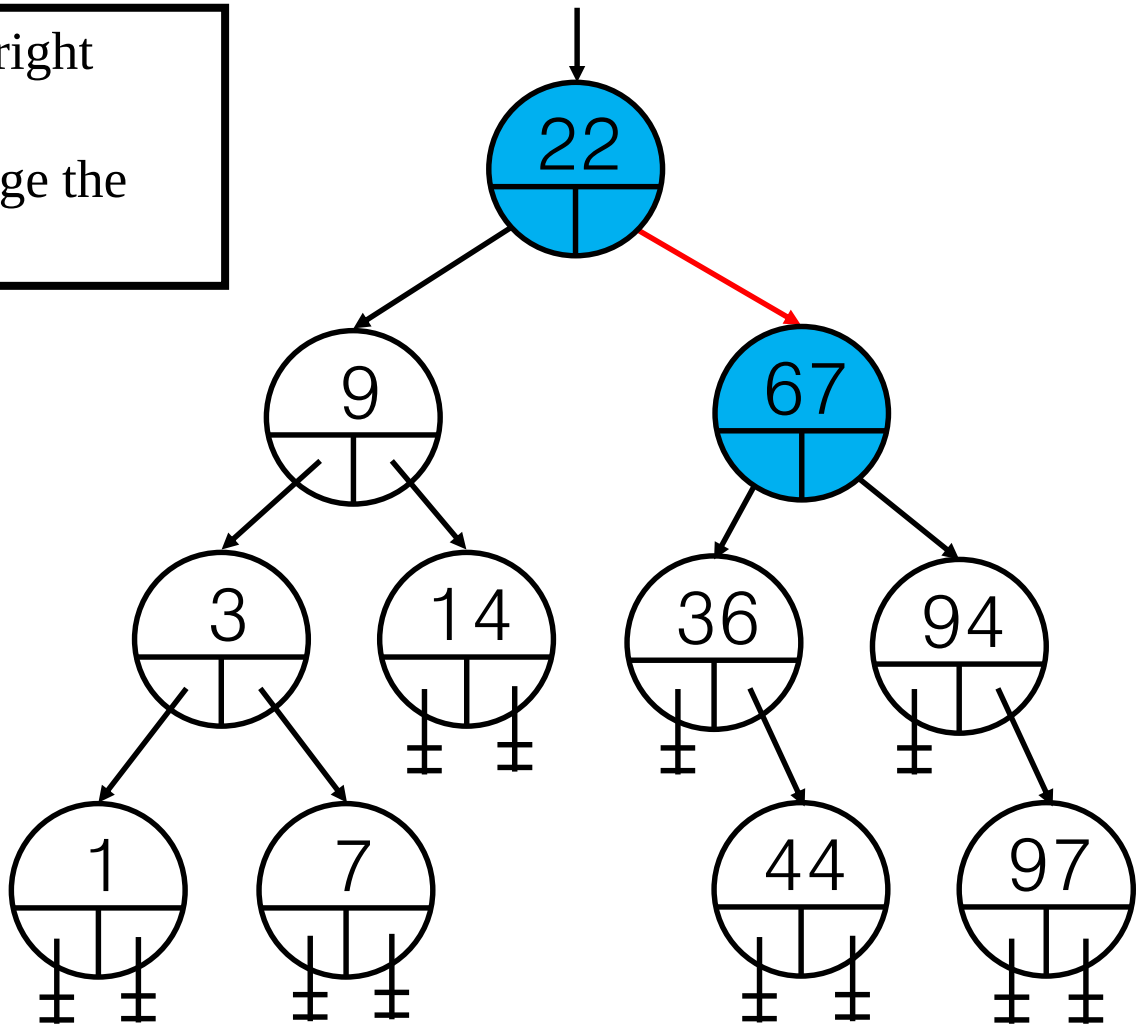
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

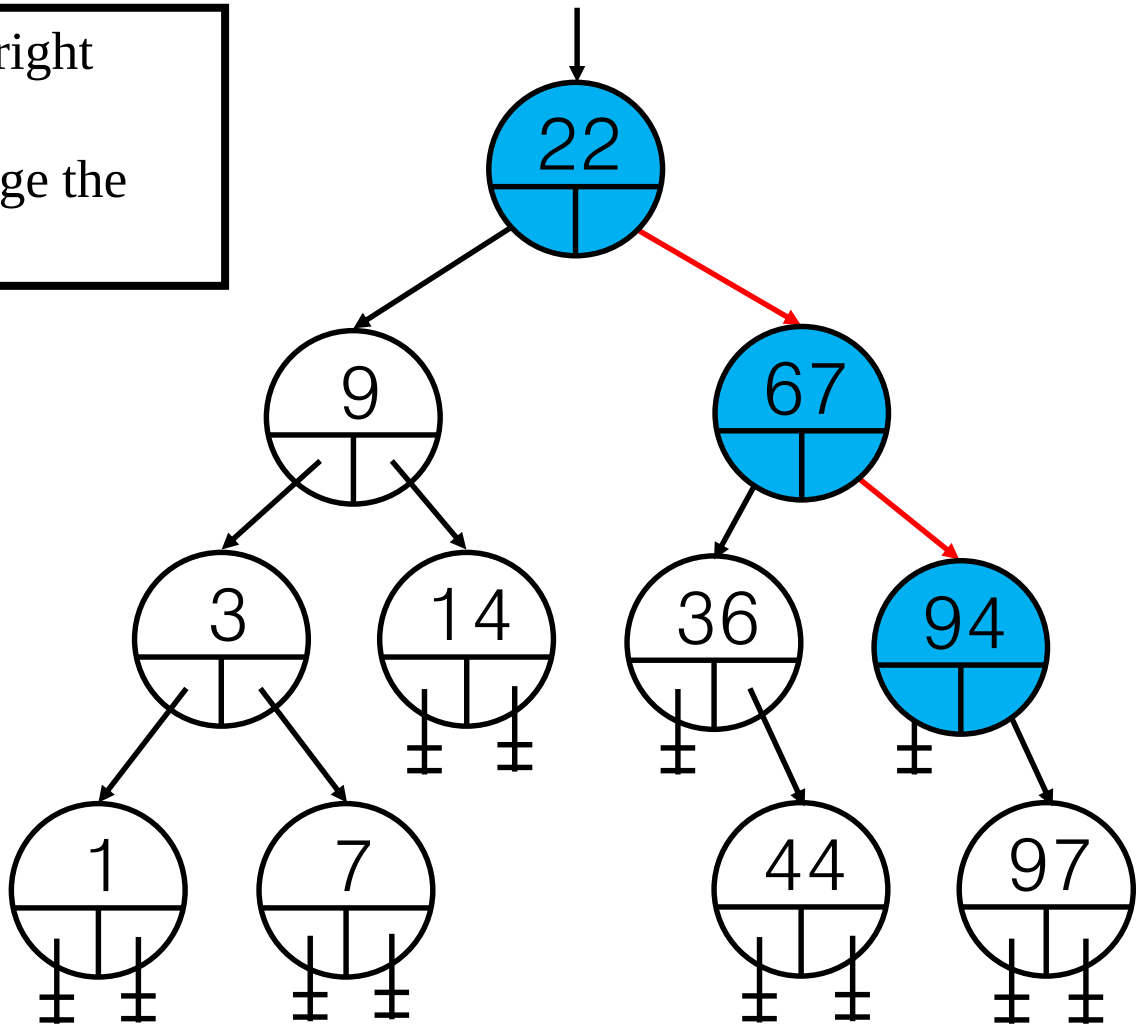
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

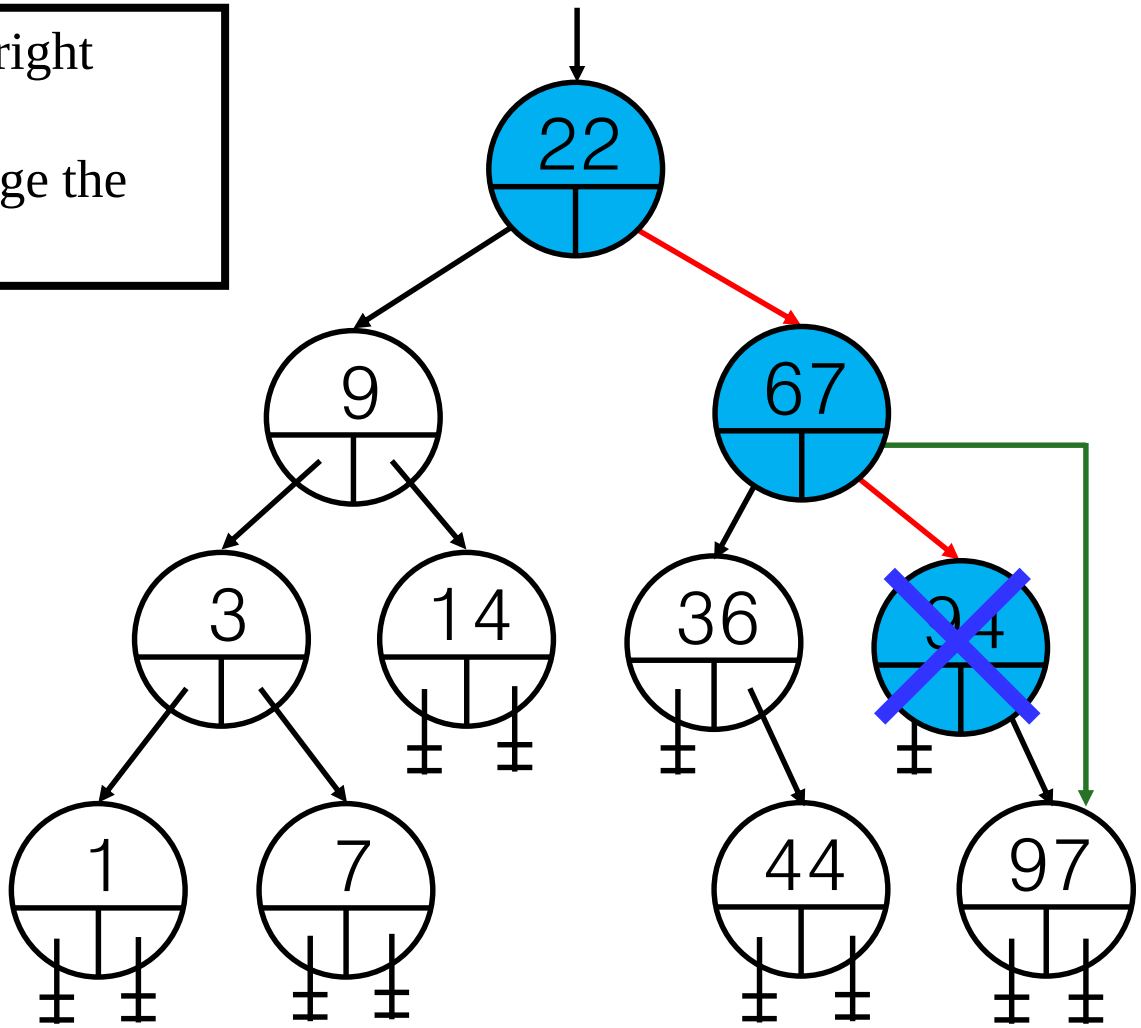
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

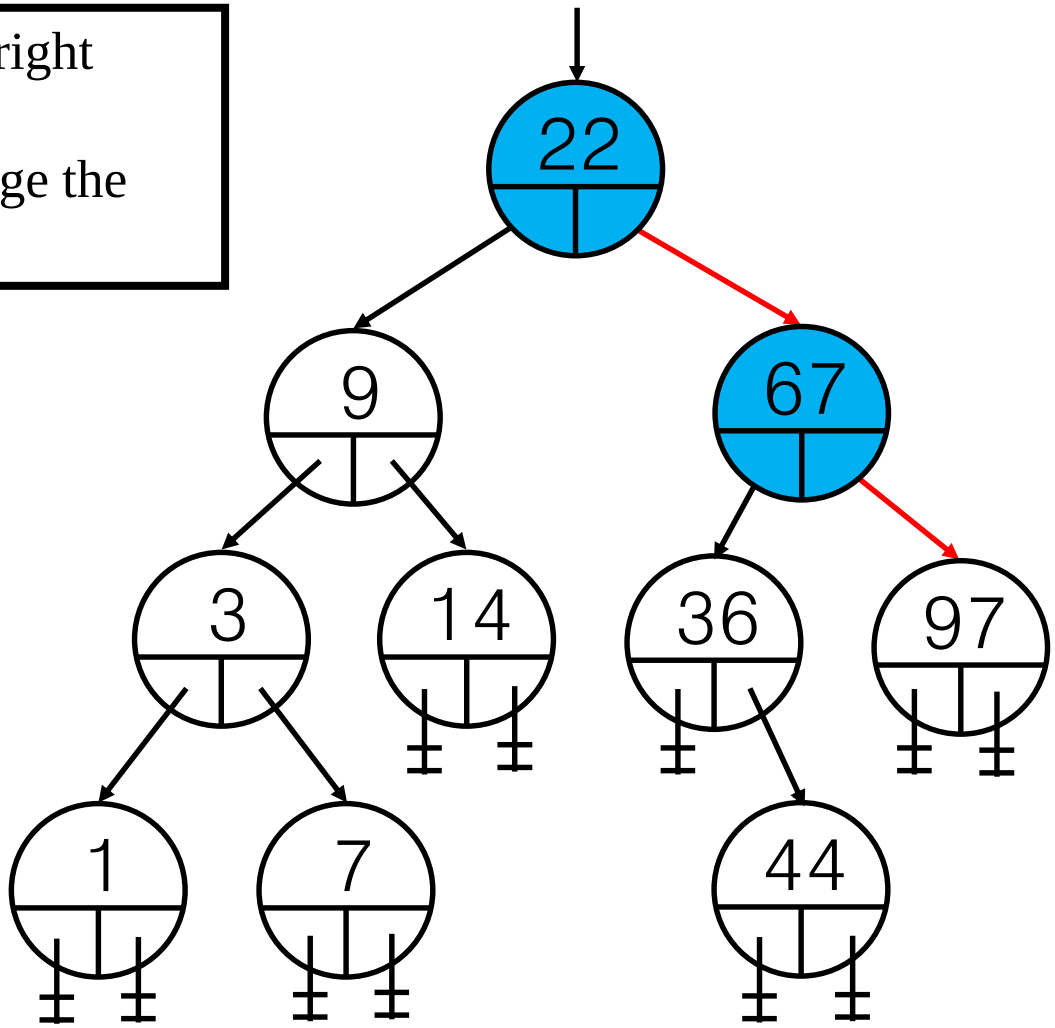
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

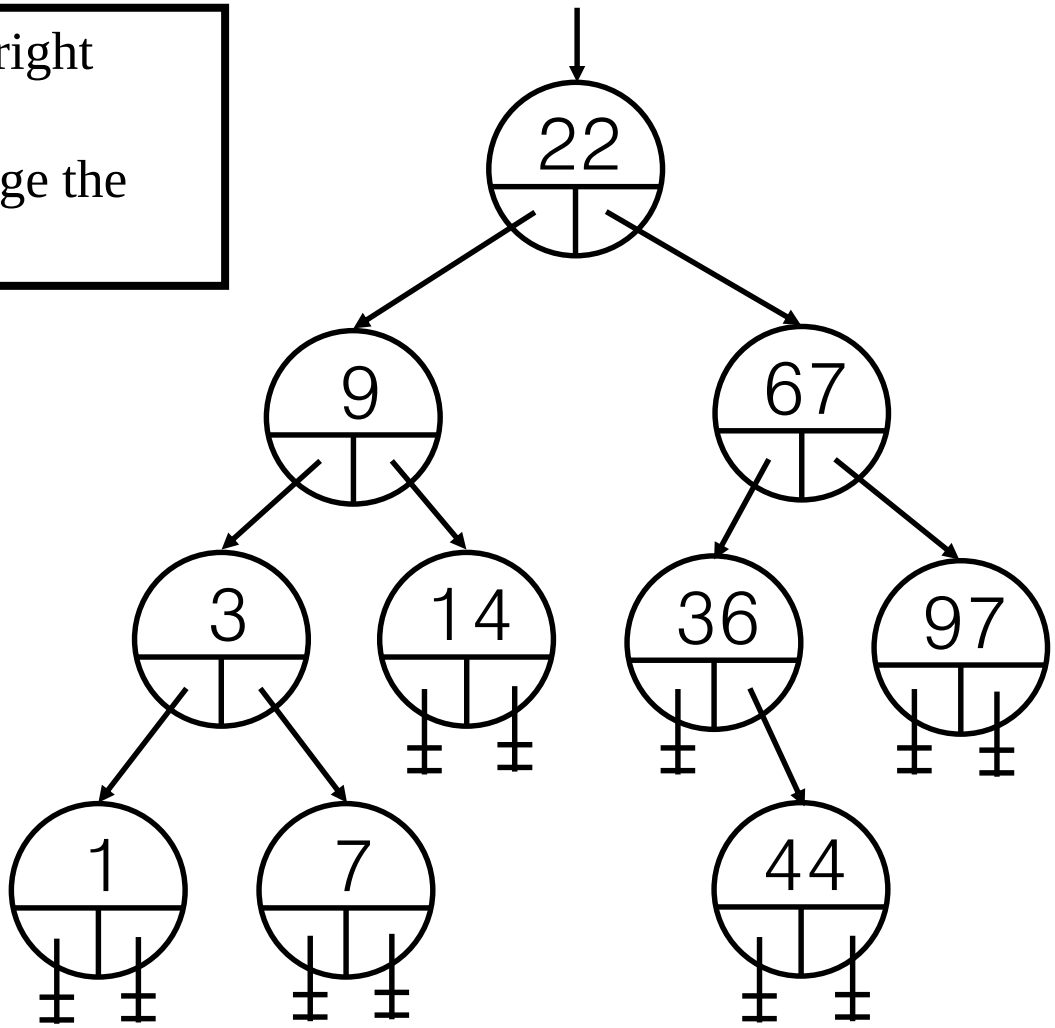
Let's Delete 94



Root(Always search starts from the root)

Determine if it has a left or a right child.
One child → delete and change the pointer to this child

The final resulting tree – still has search structure.



노드 삭제(Delete Node)

Two child(자식이 2개일 때)

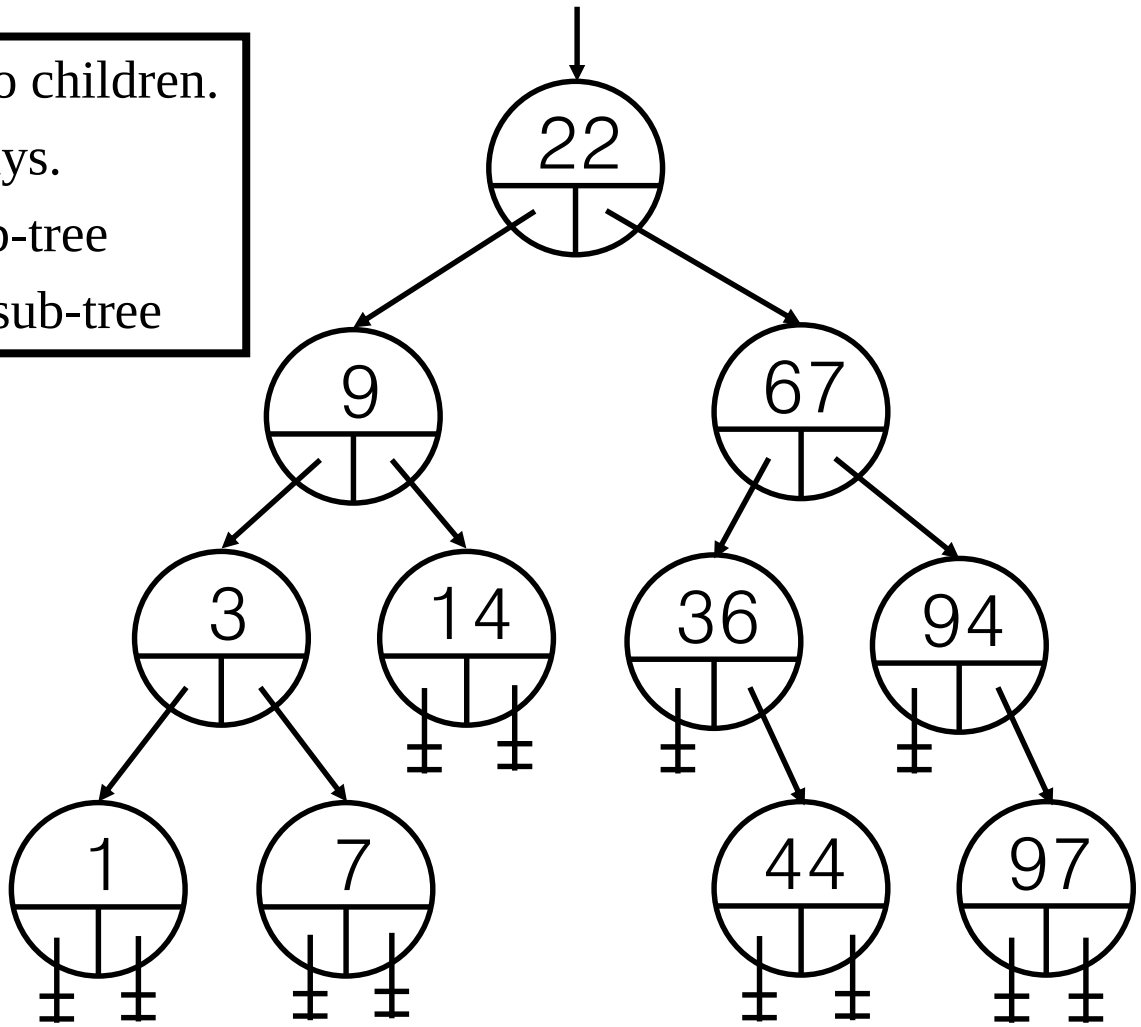
Root(Always search starts from the root)

Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree



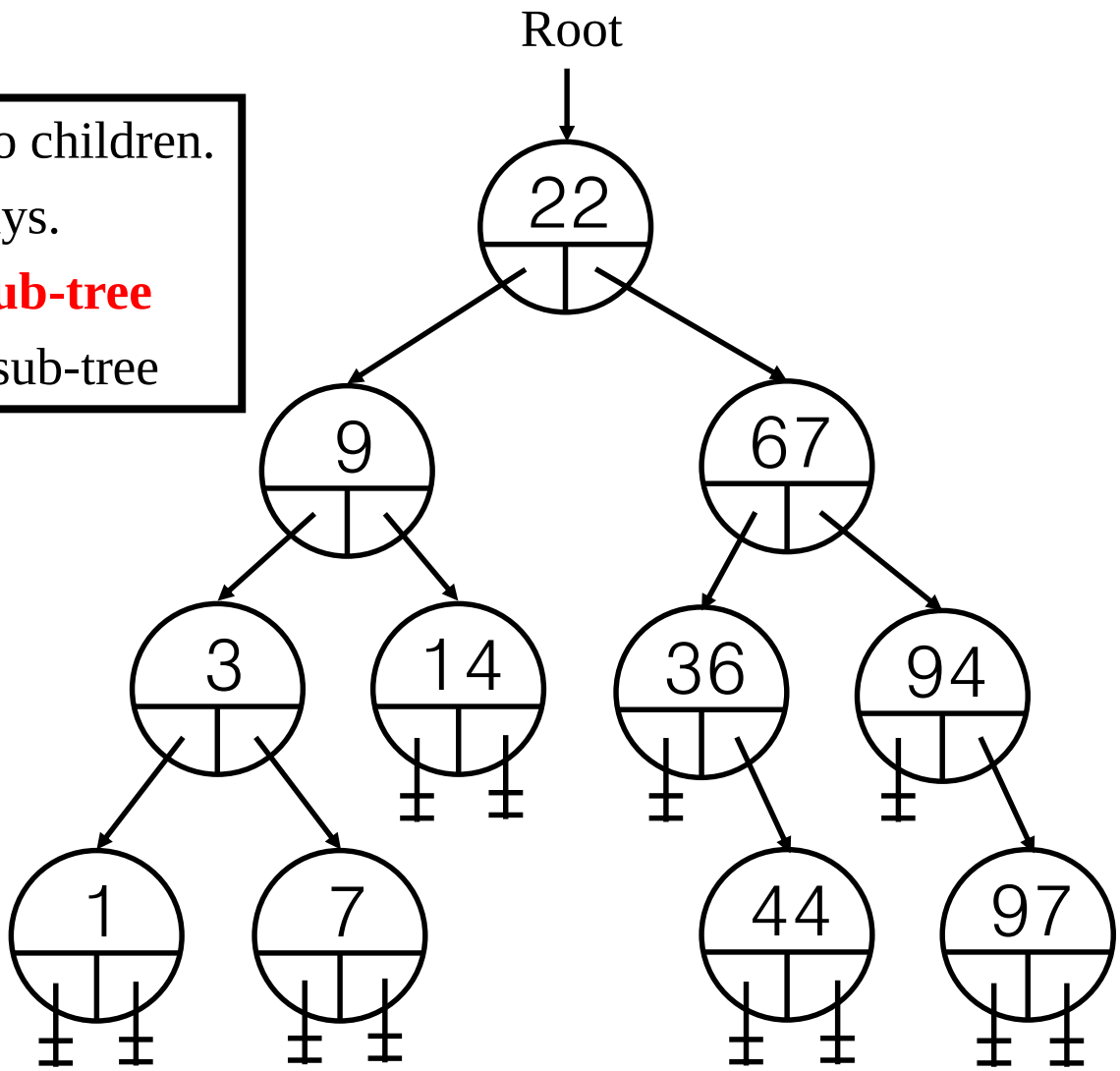
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Let's Delete 22

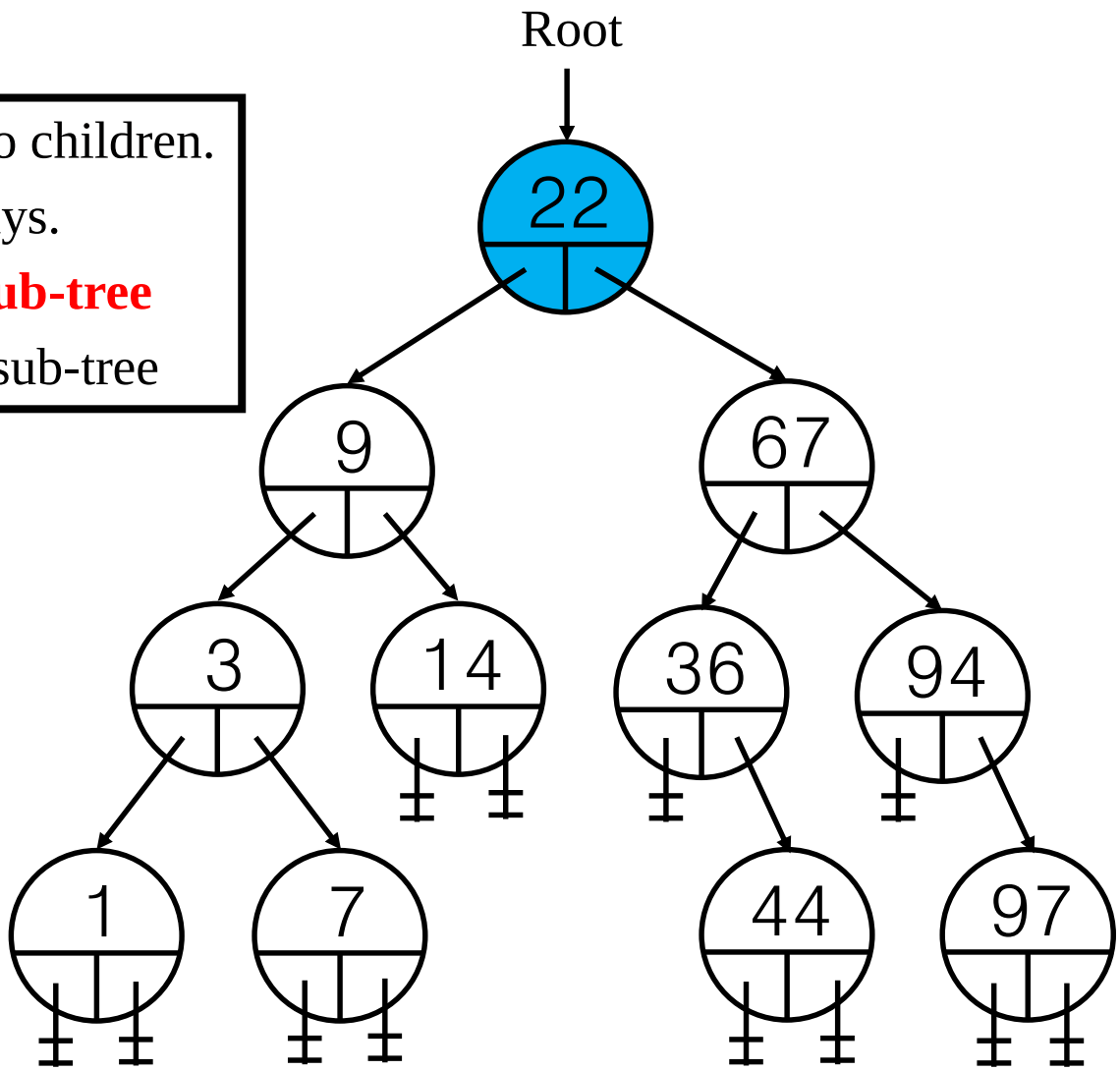


Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

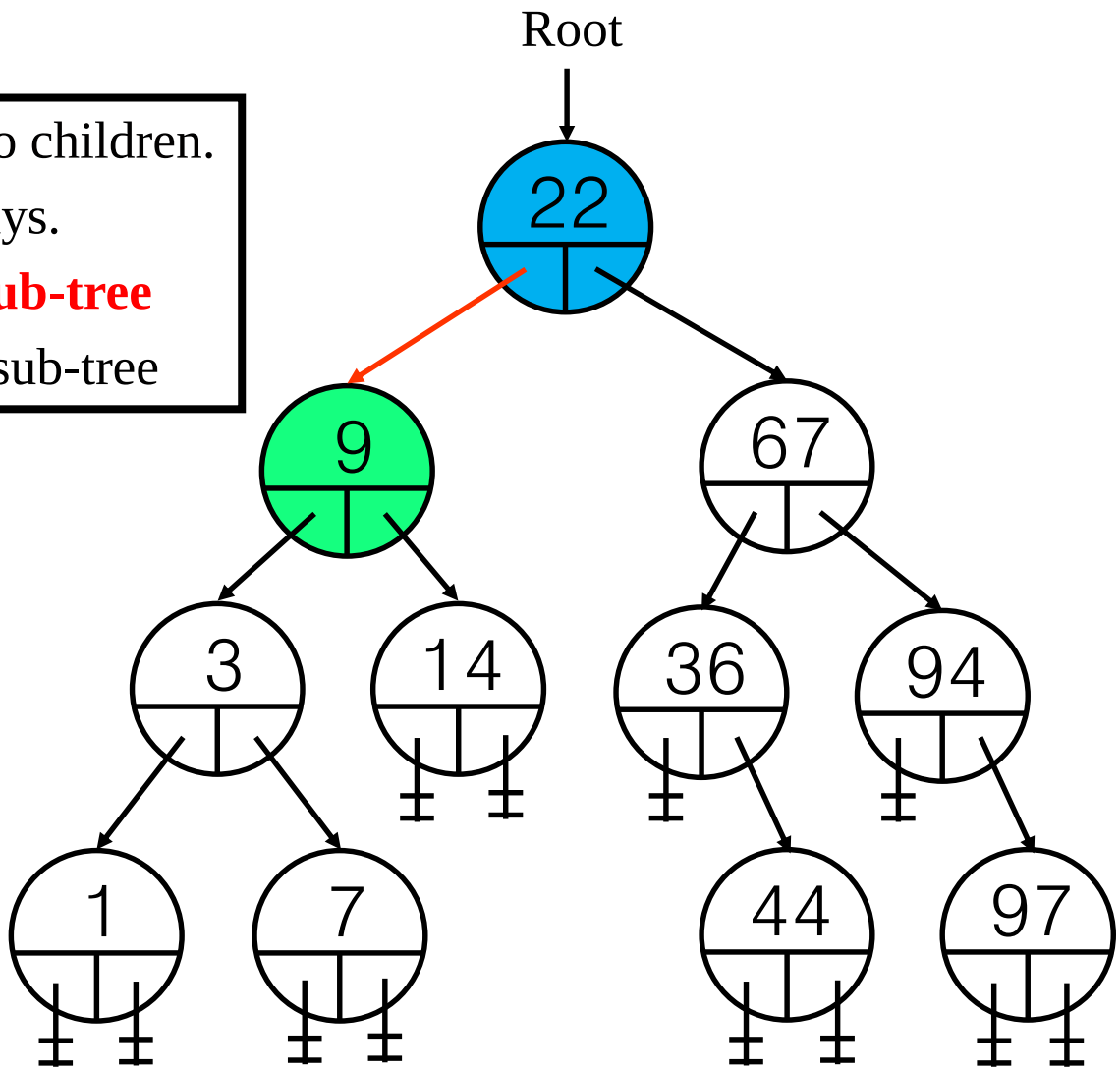


Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree



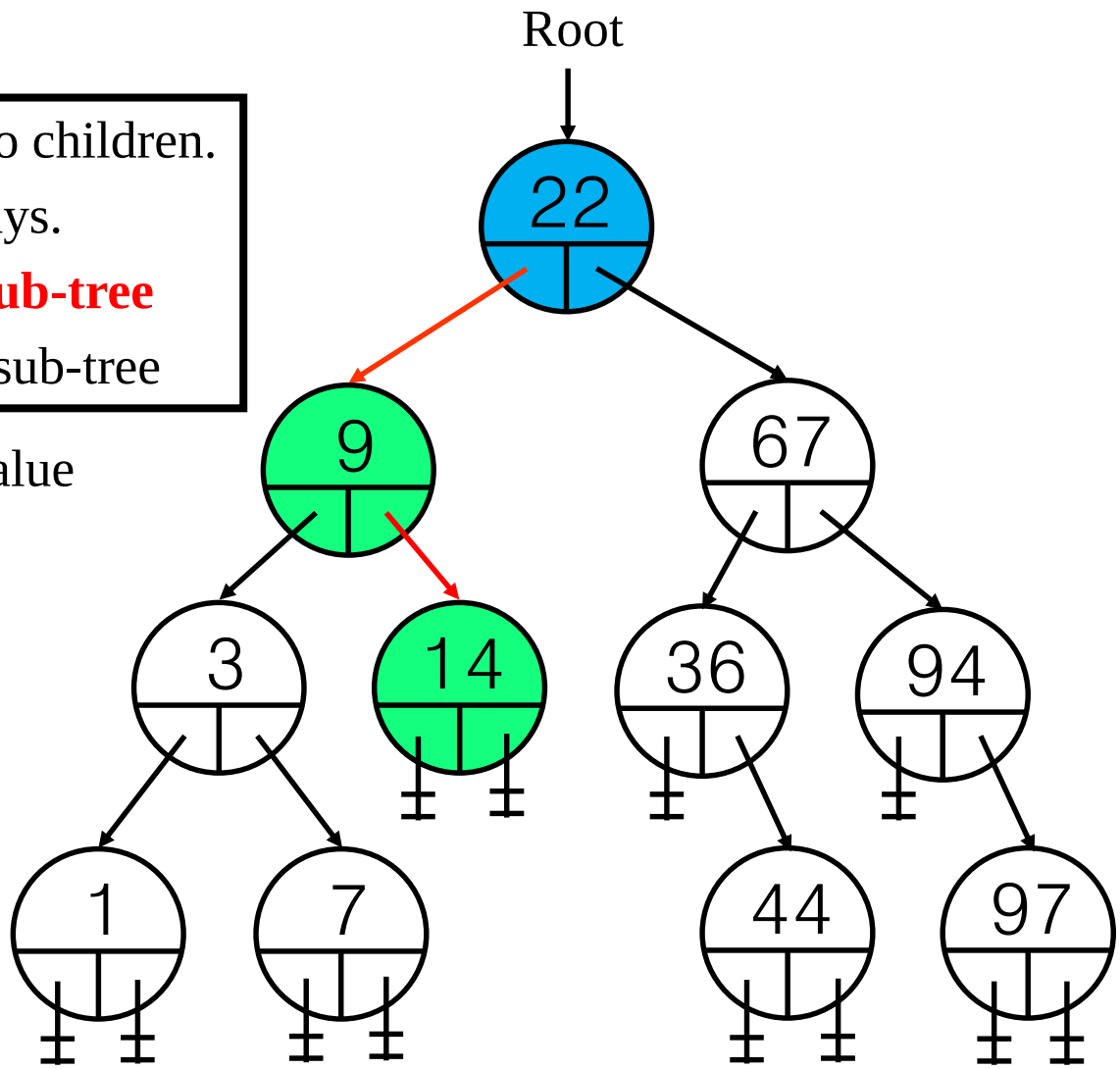
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Find and copy the largest value in the left sub-tree.



Node to be removed has two children.

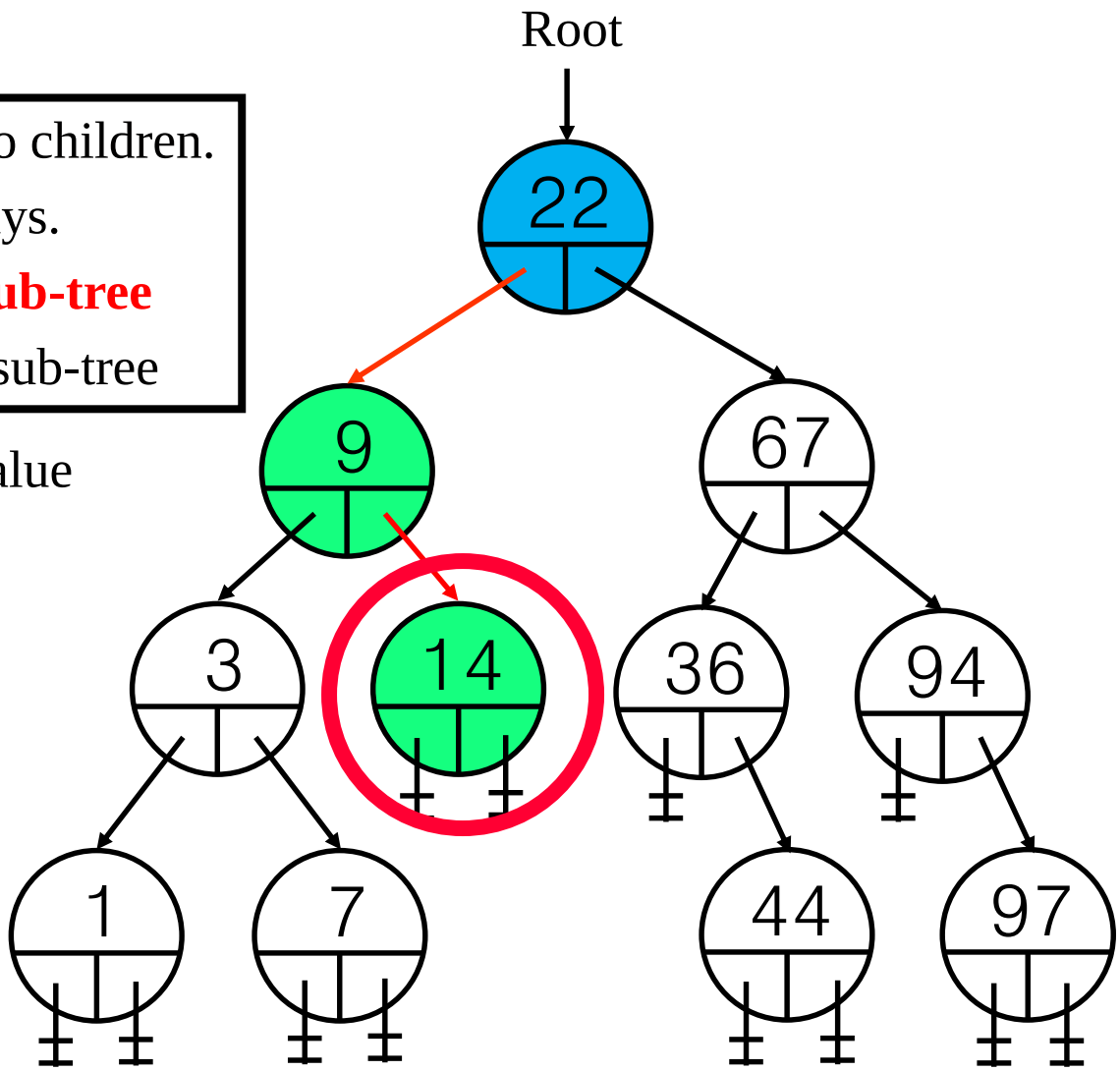
We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Find and copy the largest value in the left sub-tree.

In current example it is 14.



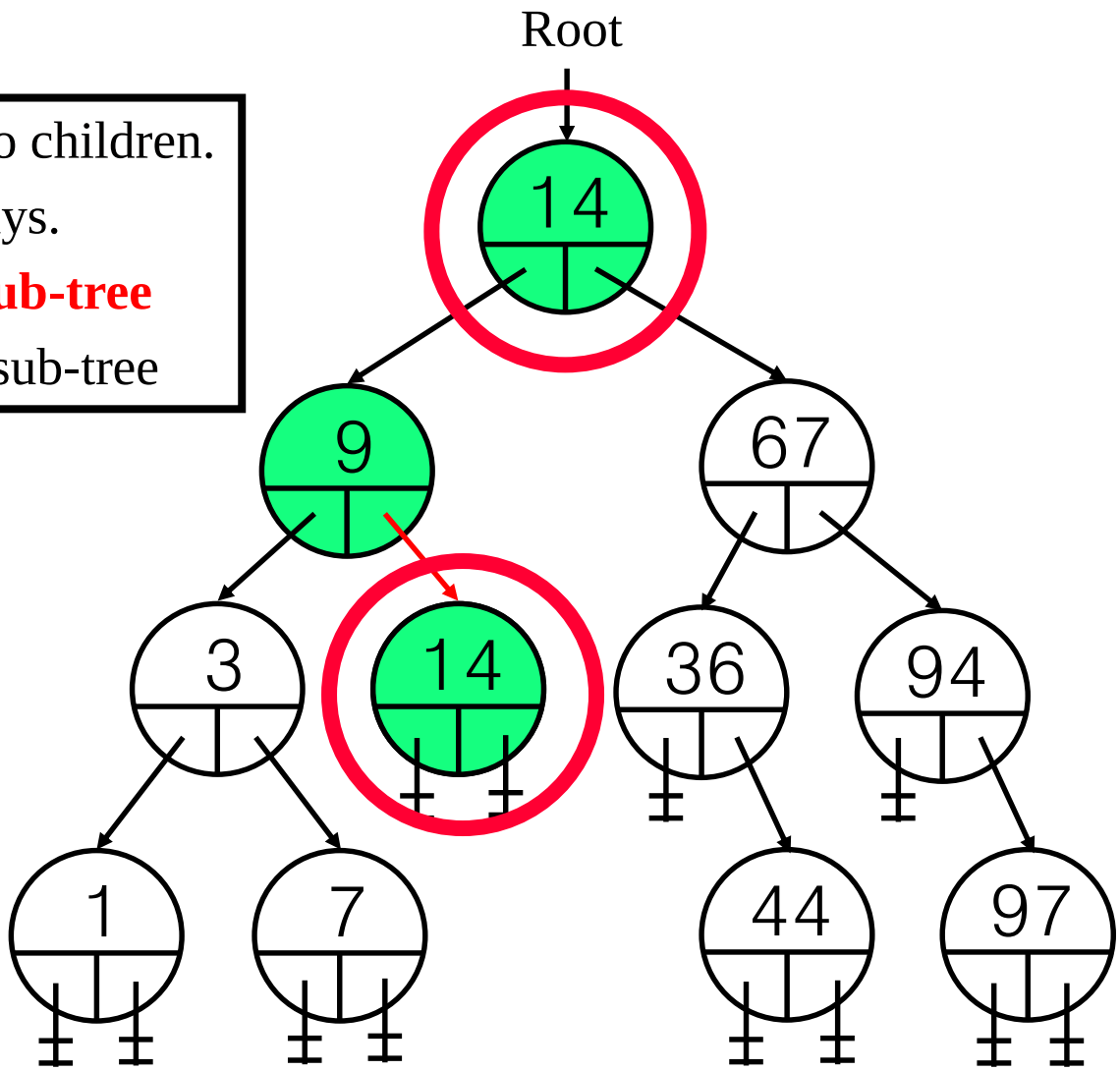
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Replace 22 with 14.



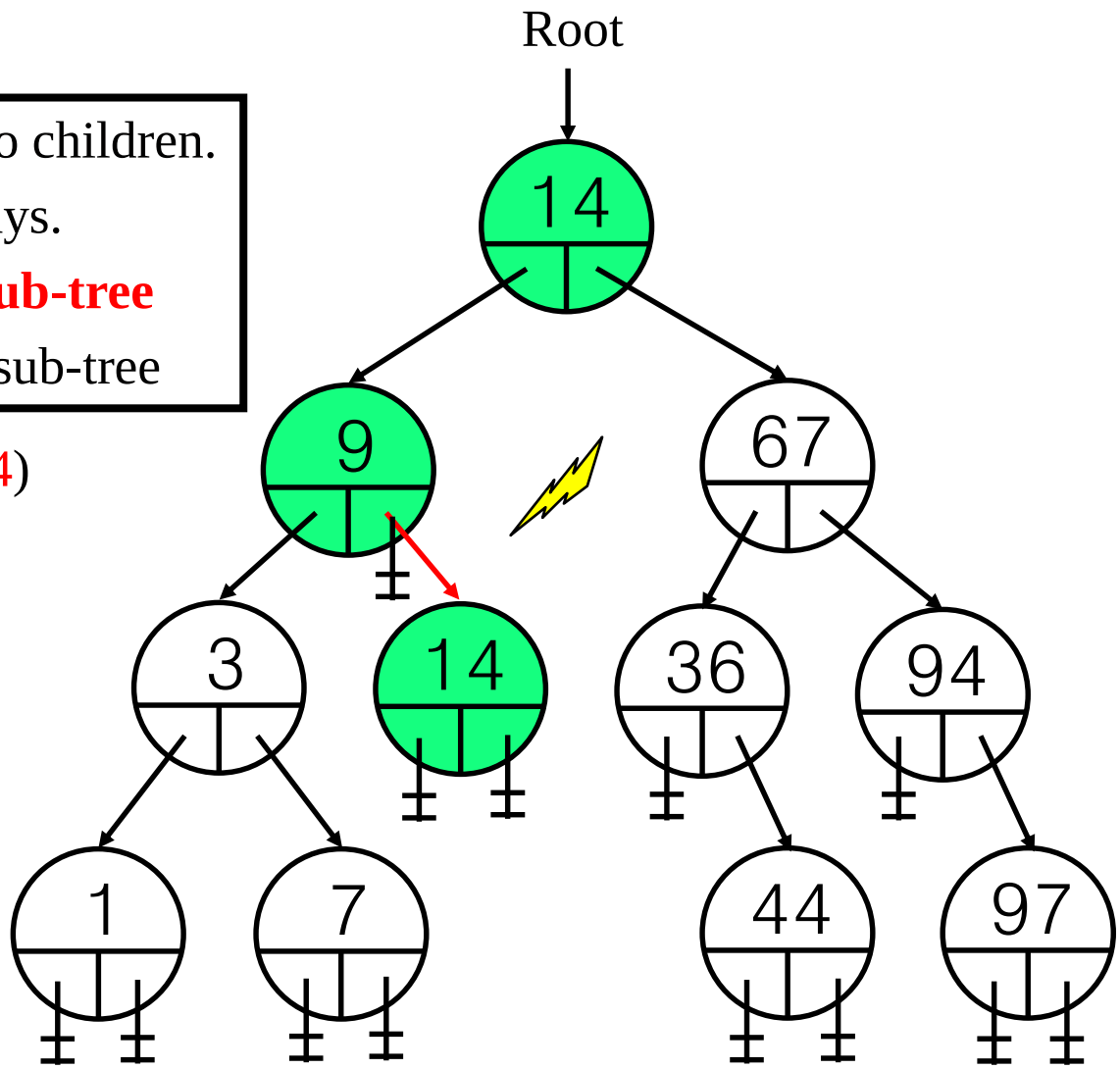
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Now delete the duplicate(14) from the left sub-tree.



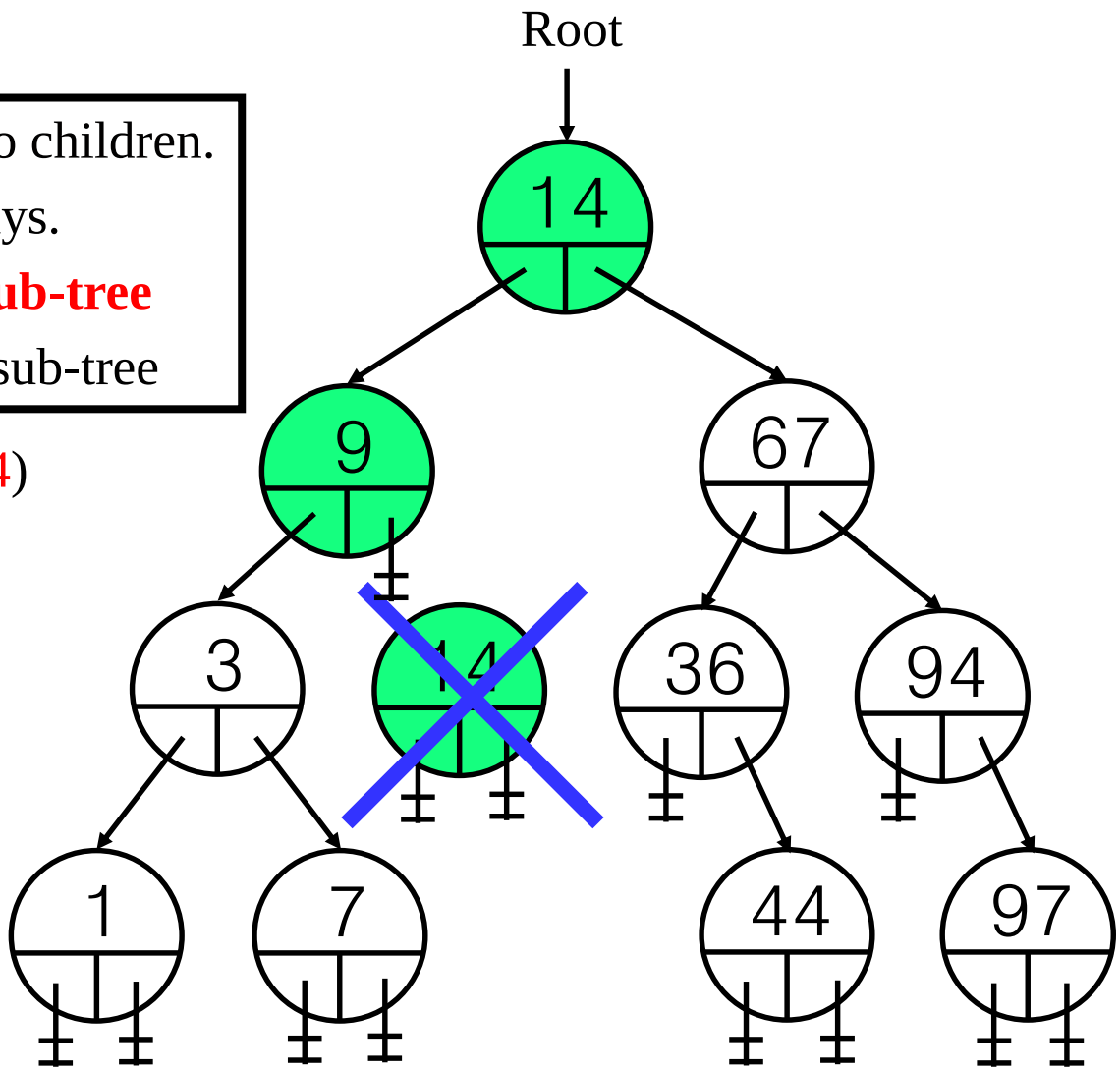
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Now delete the duplicate(14) from the left sub-tree.



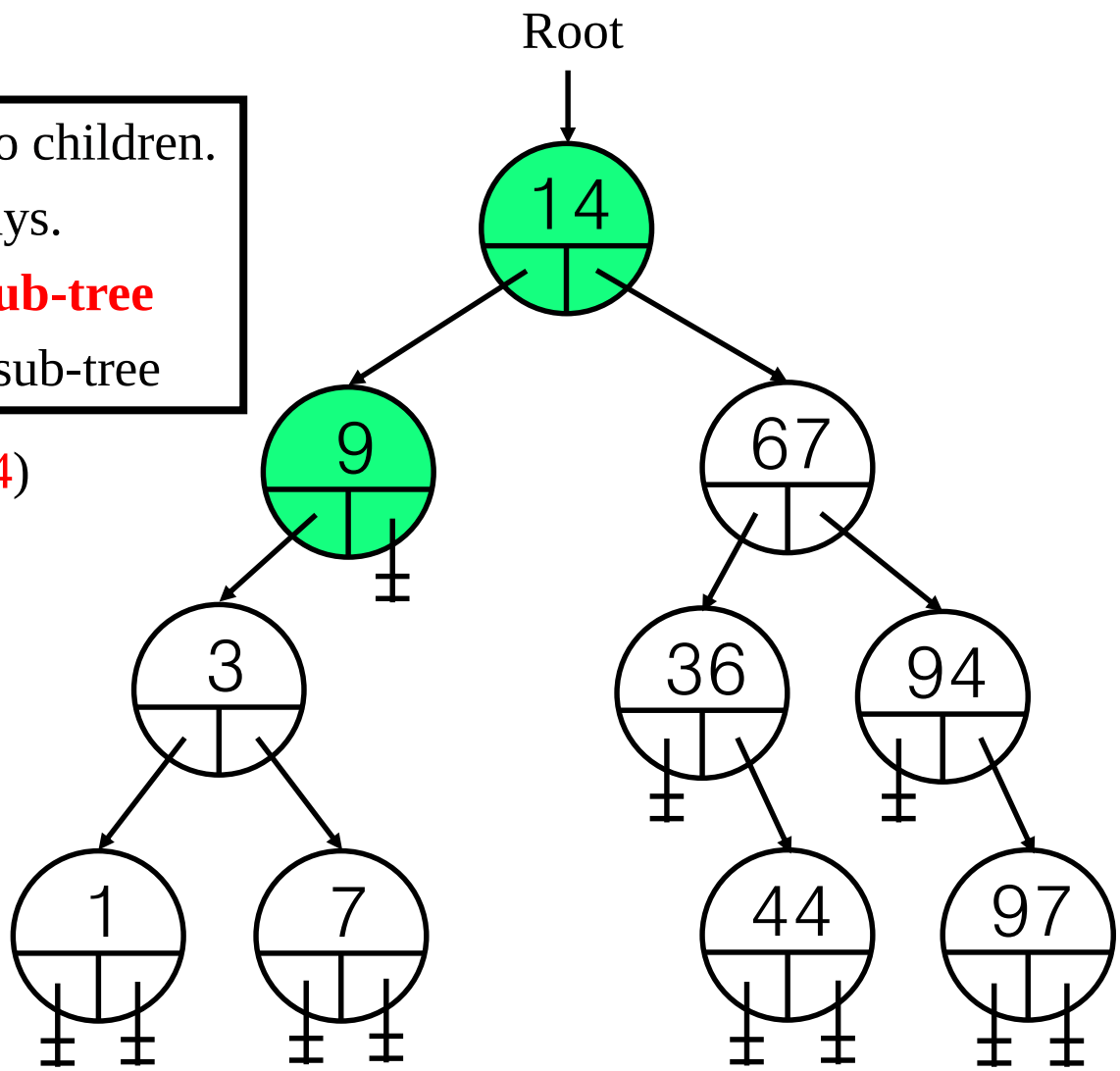
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Now delete the duplicate(14) from the left sub-tree.



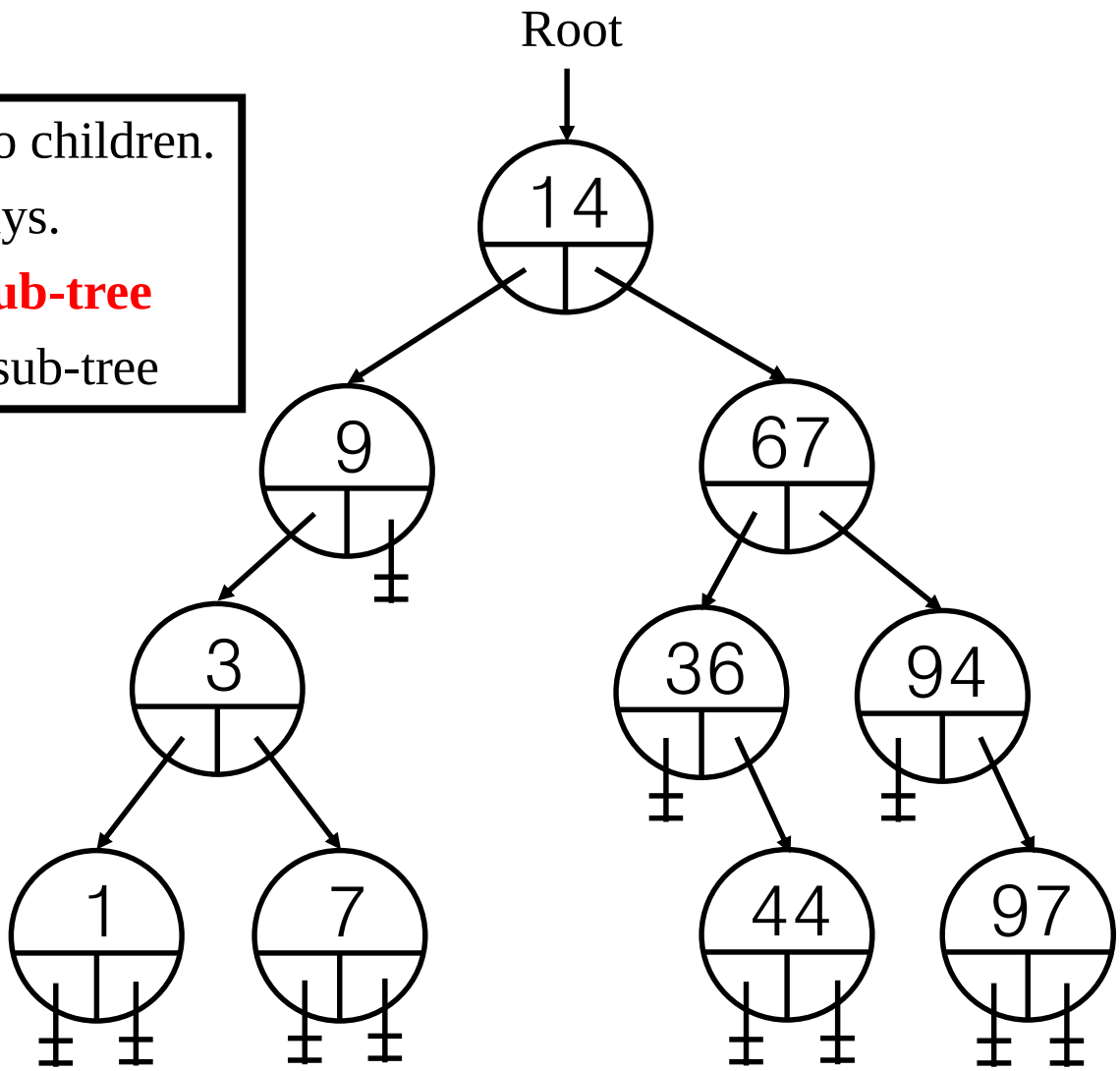
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

The final resulting tree –
still has search structure.



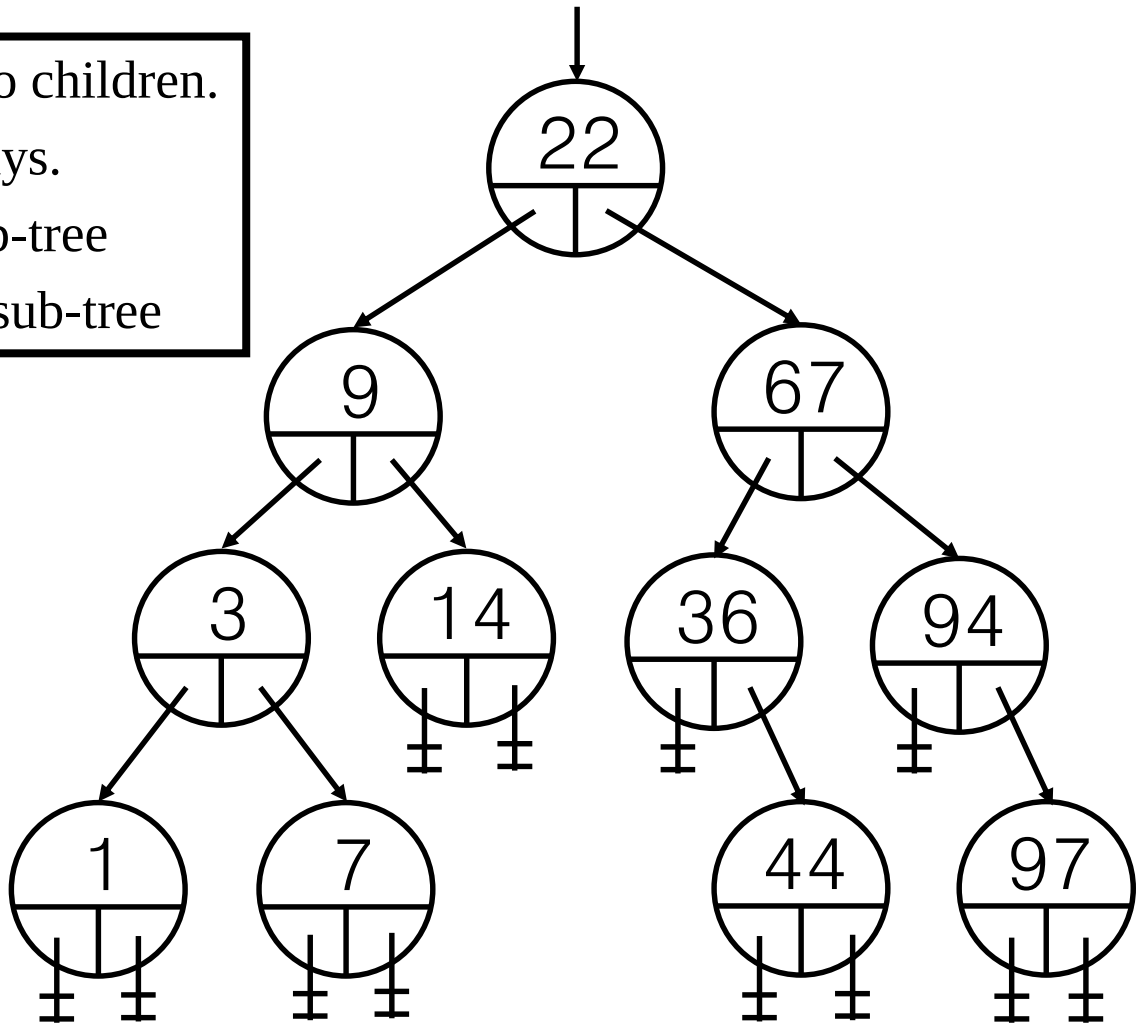
Root(Always search starts from the root)

Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree



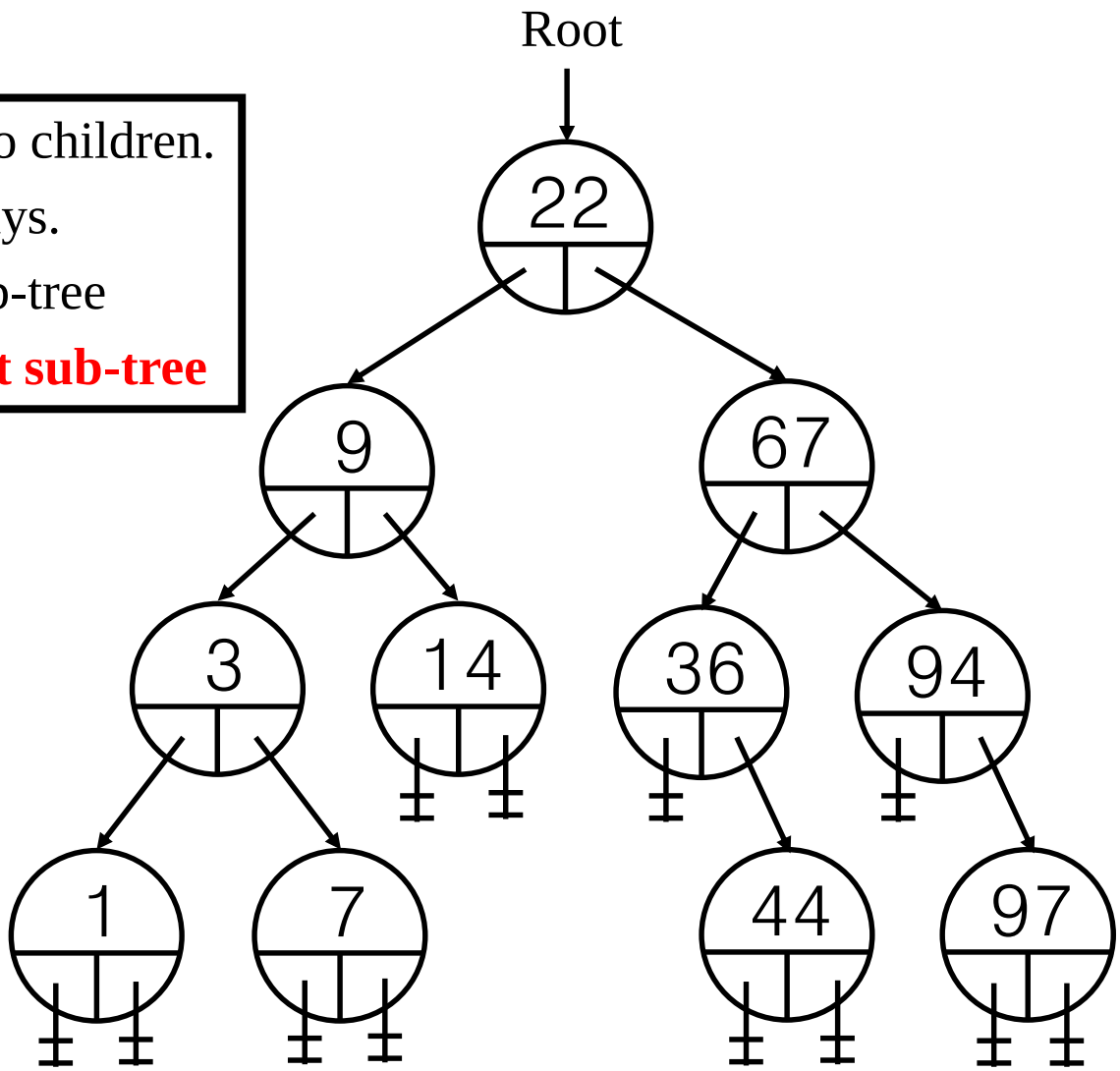
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Let's Delete 22

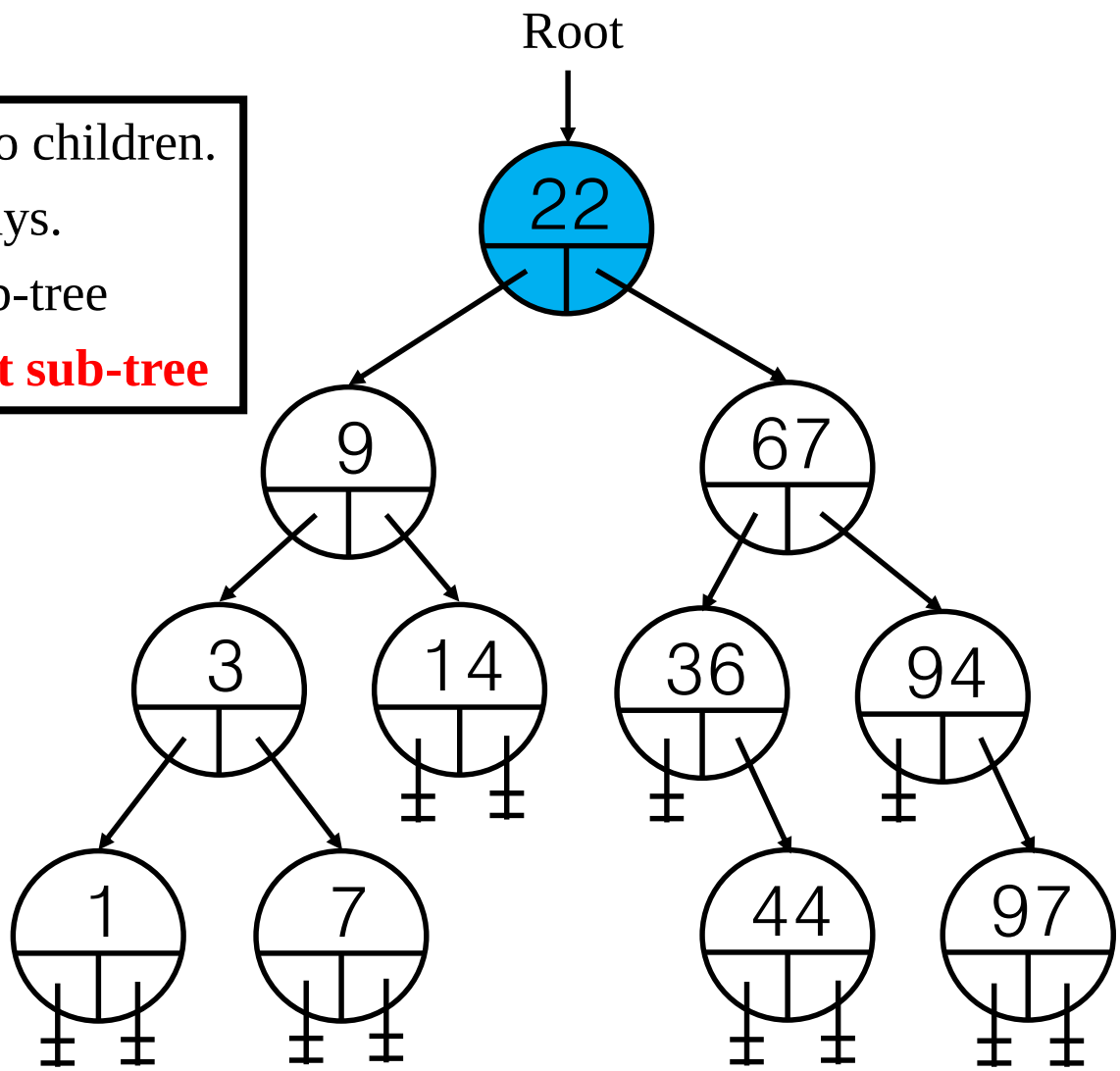


Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

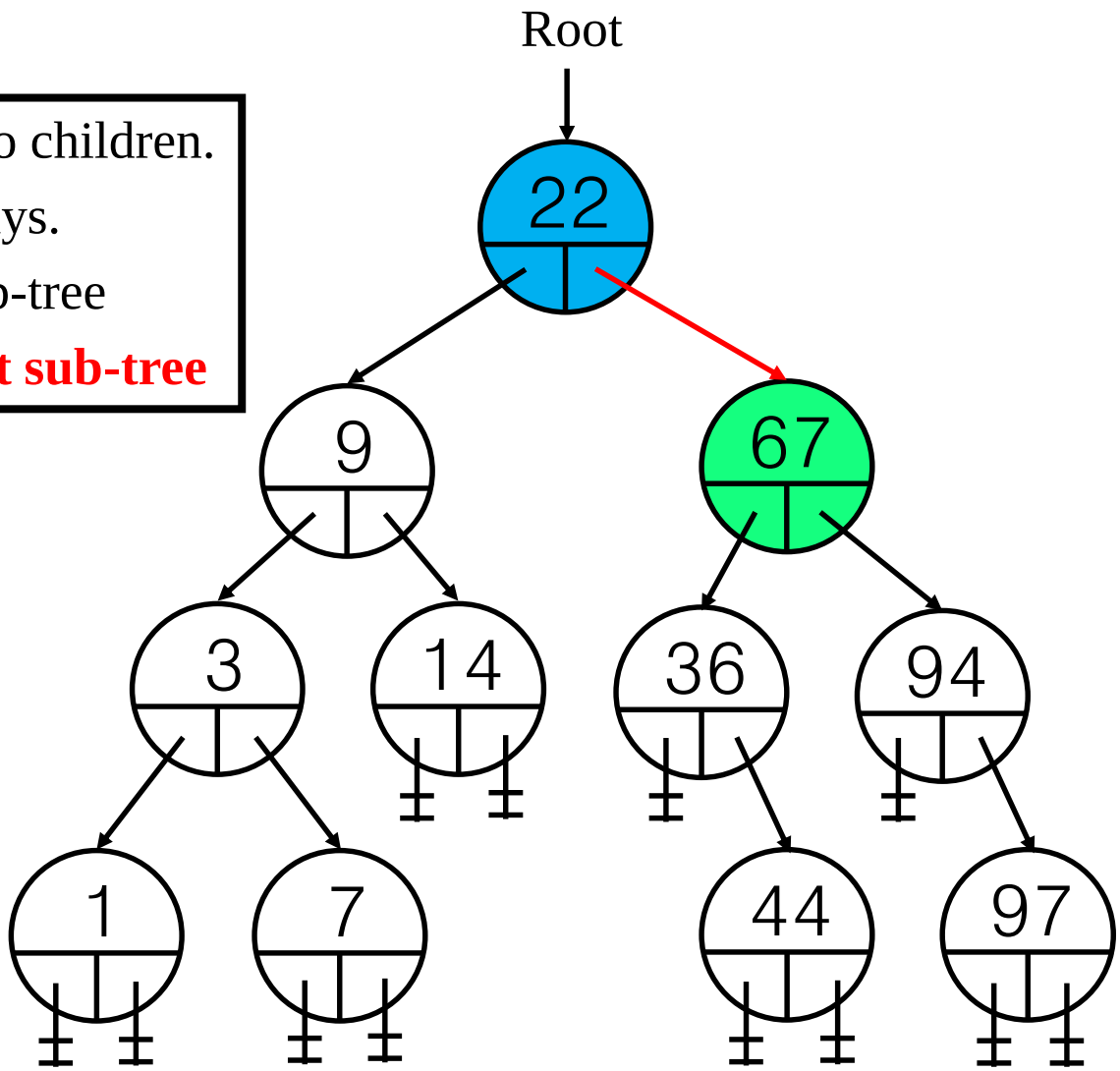


Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree



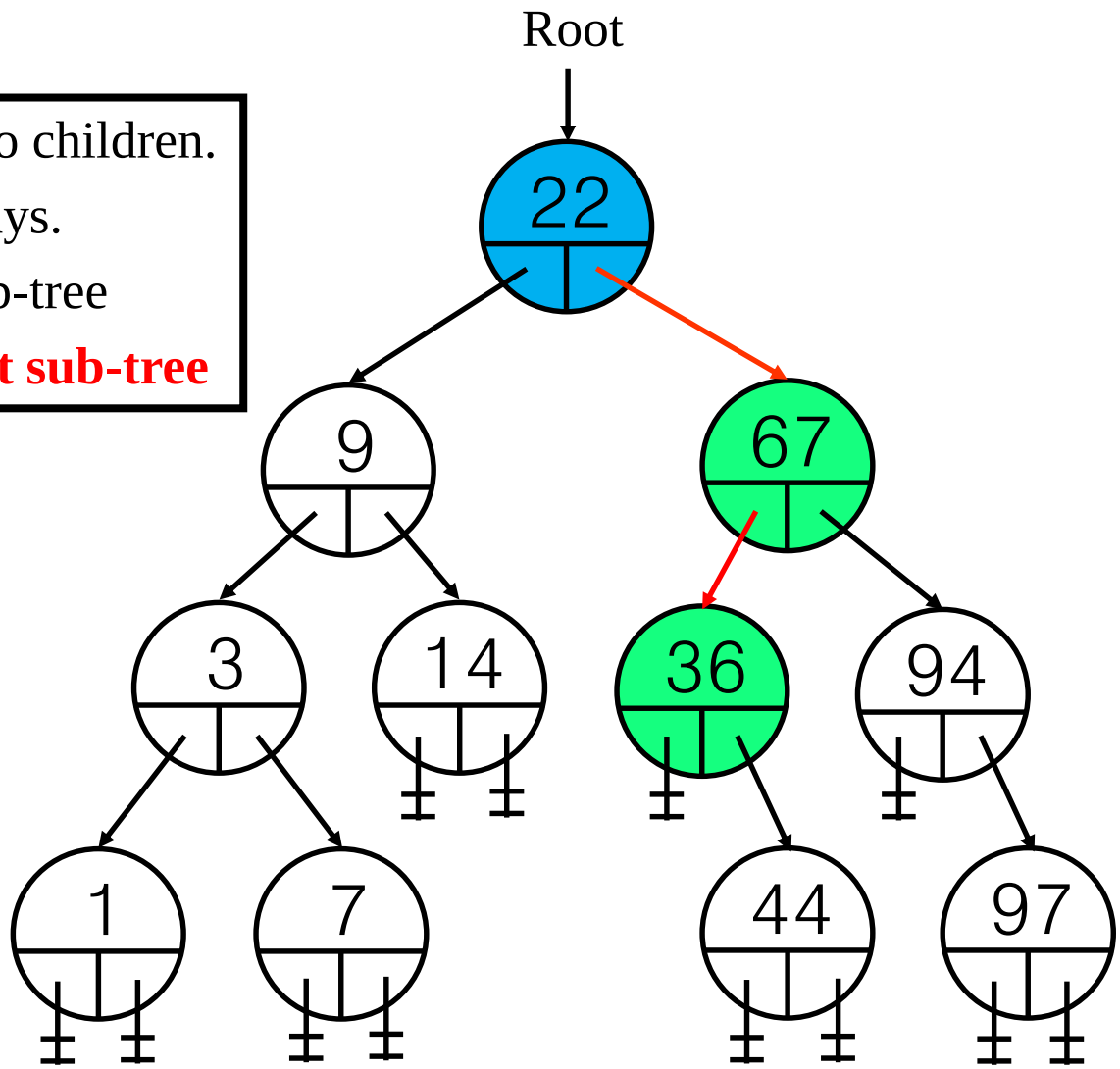
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Find and copy the smallest value in the right sub-tree.



Node to be removed has two children.

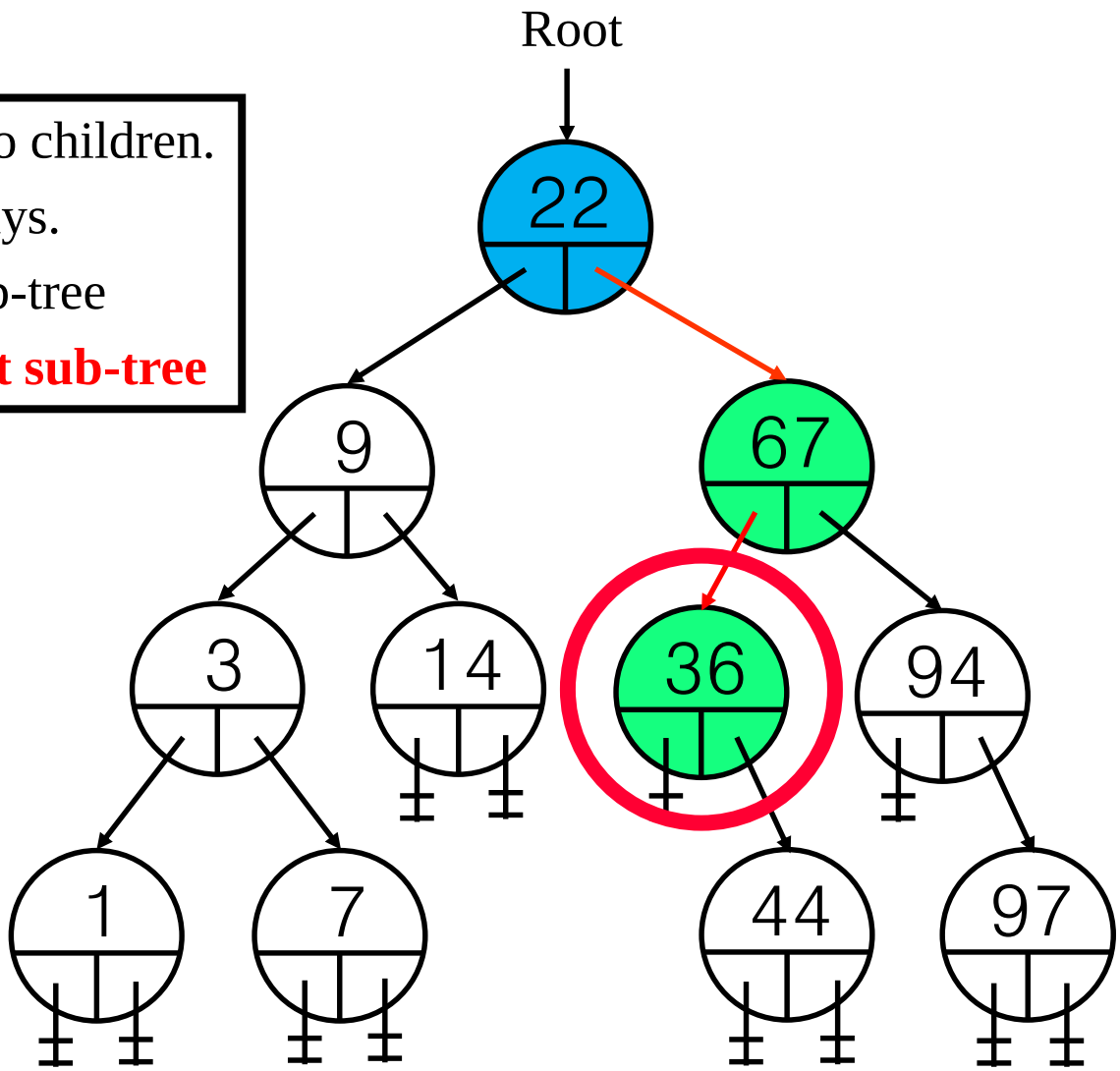
We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Find and copy the smallest value in the right sub-tree.

In current example it is 36.



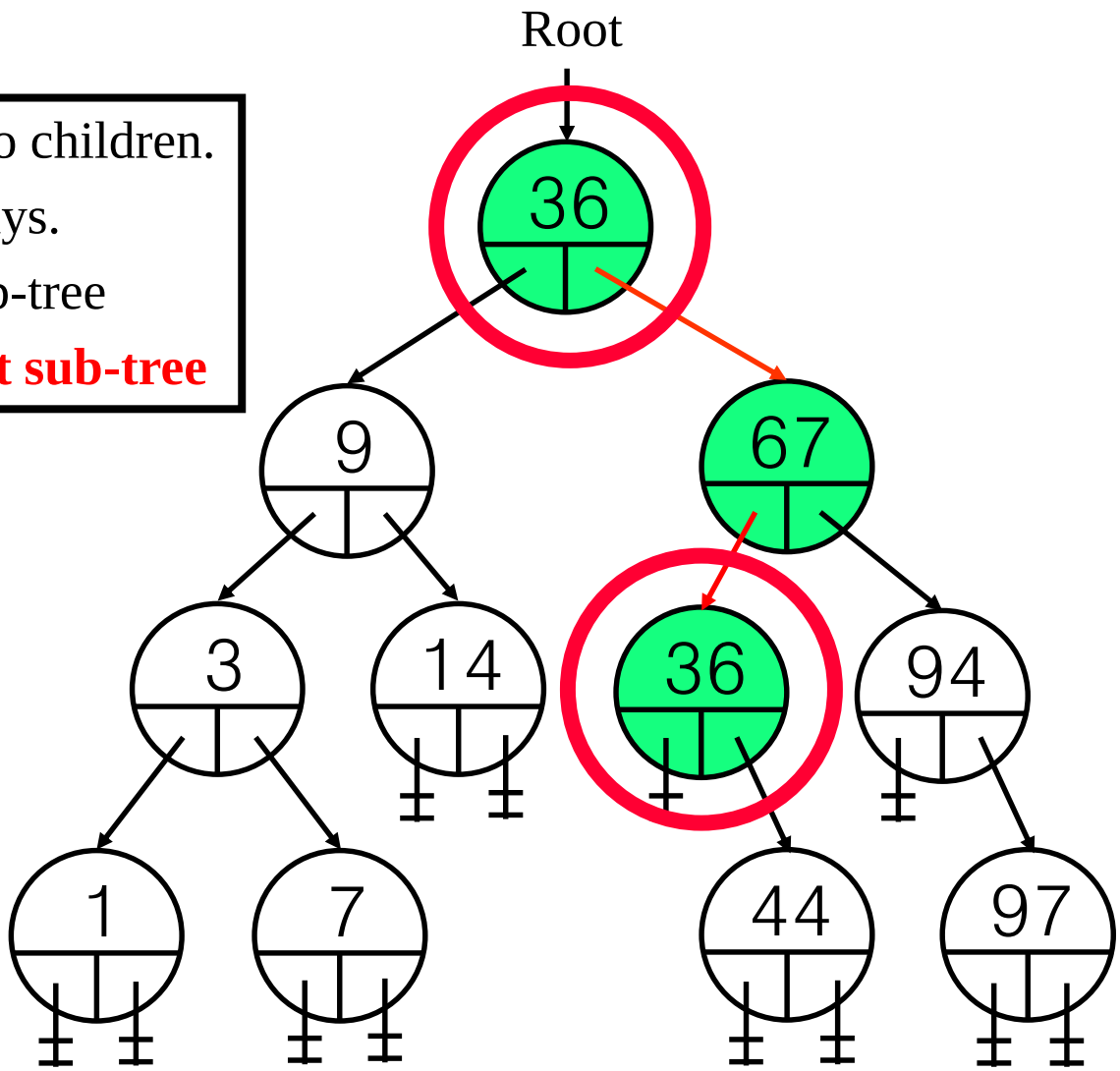
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Replace 22 with 36.



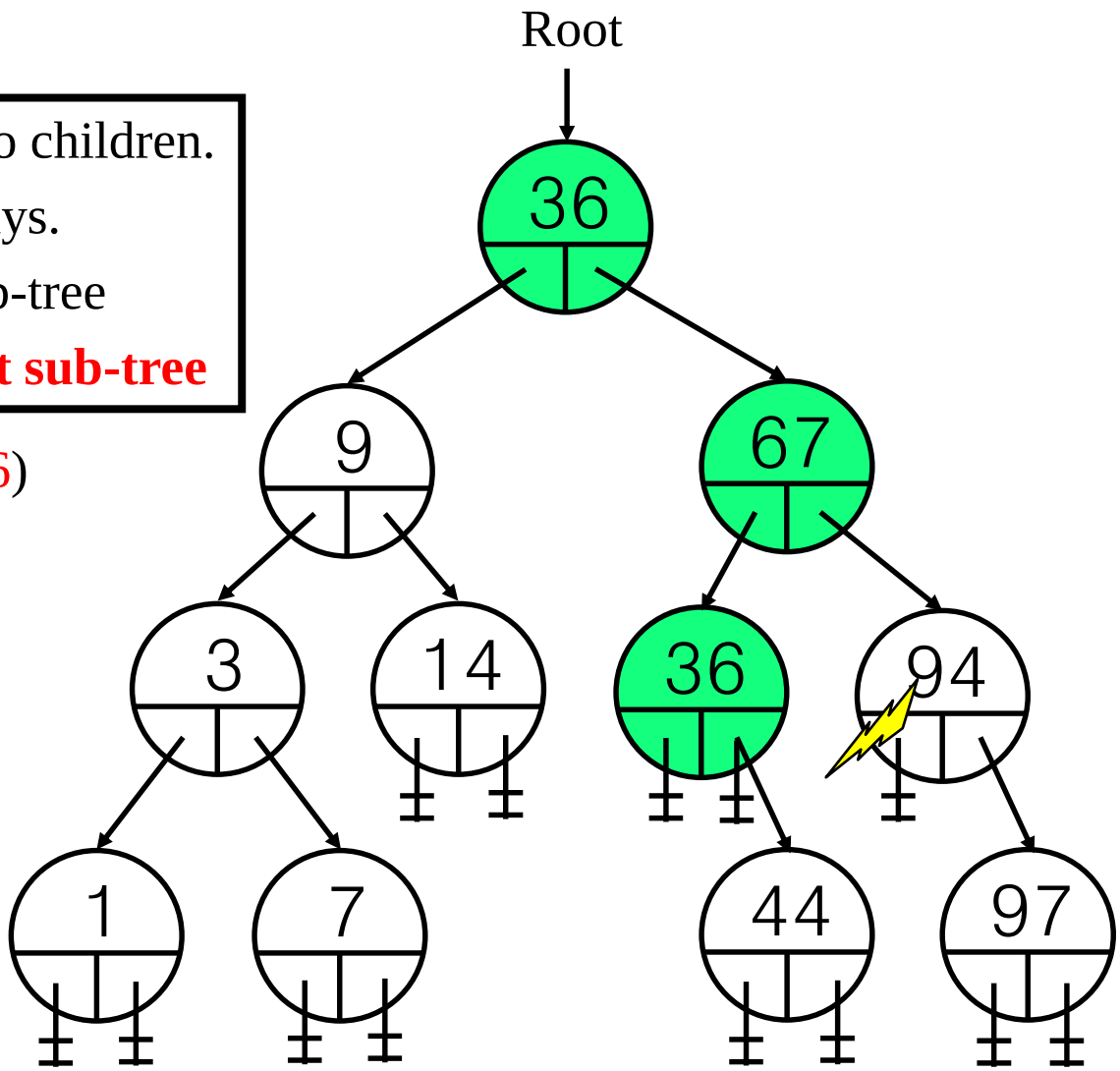
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Now delete the duplicate(36) from the right sub-tree.



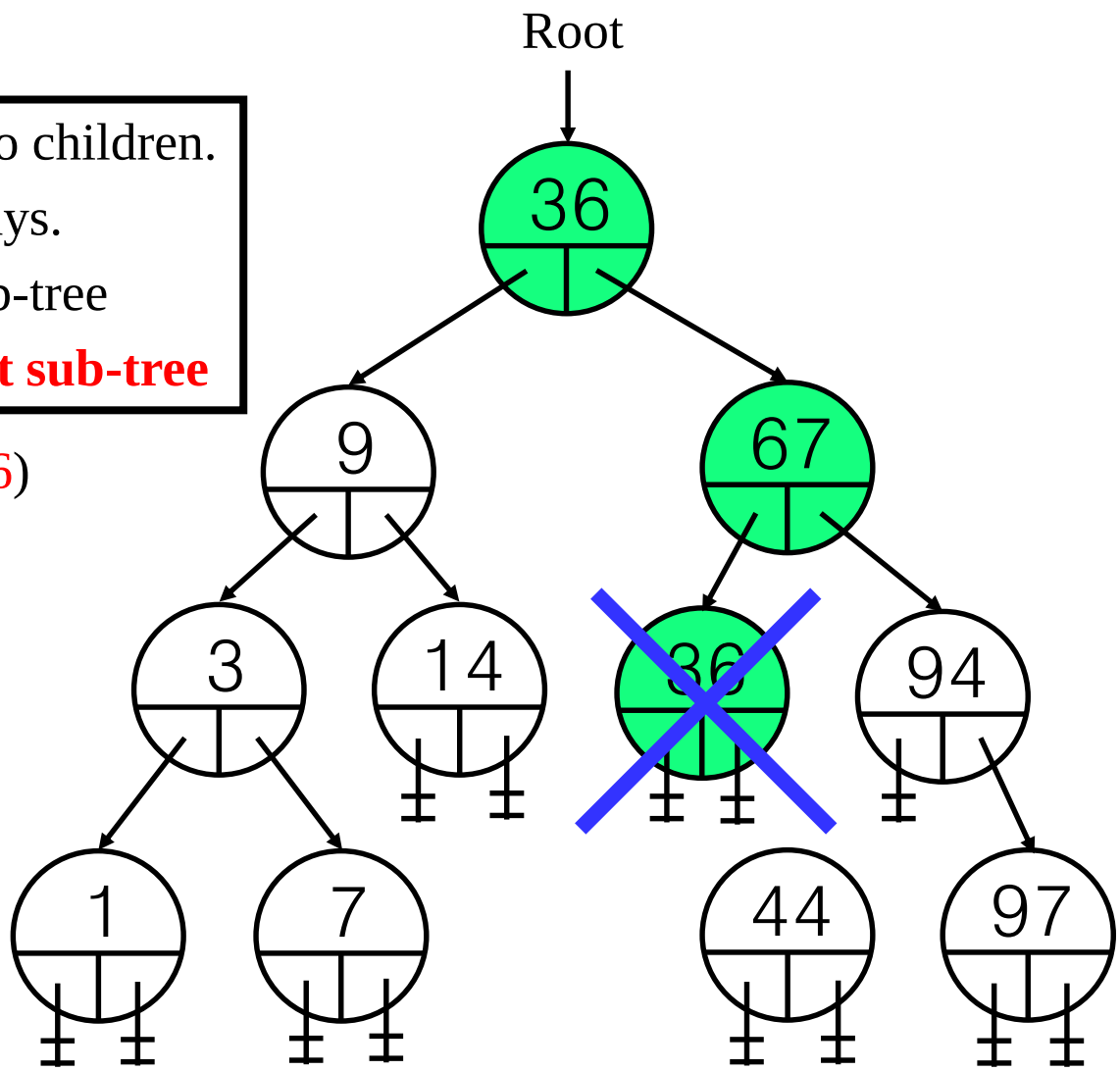
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Now delete the duplicate(36) from the right sub-tree.



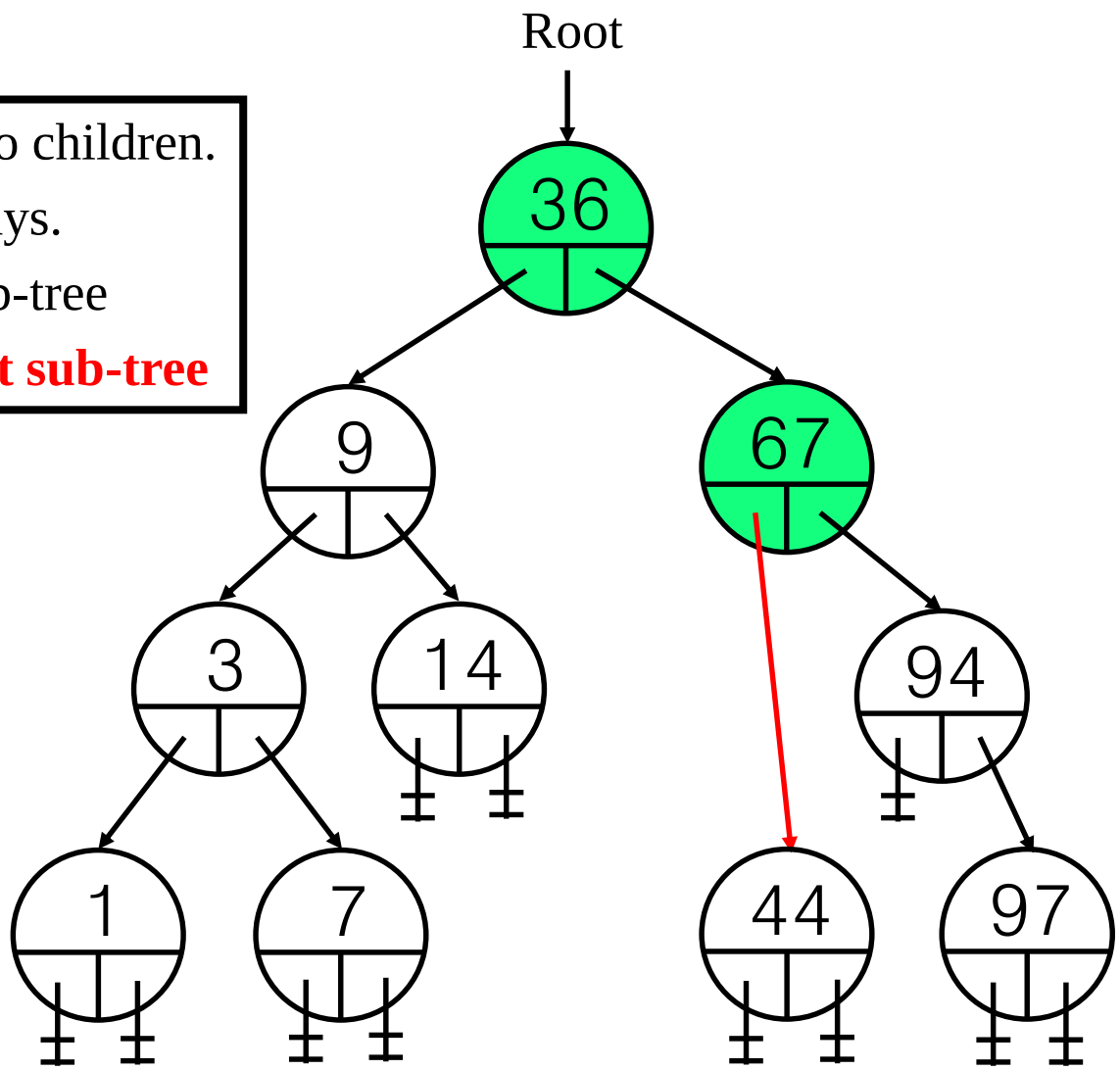
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Move the pointer.



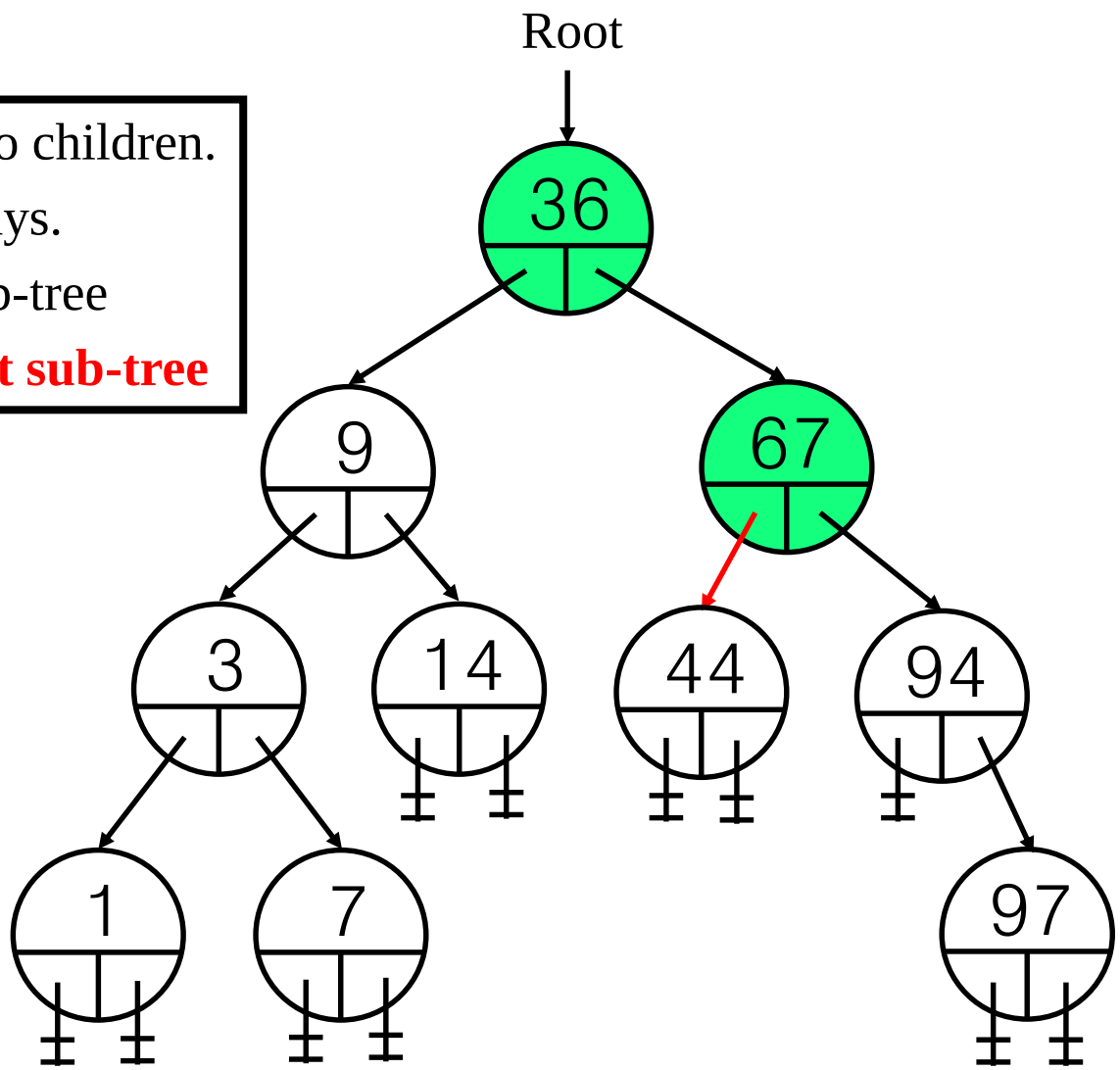
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

Move the pointer.



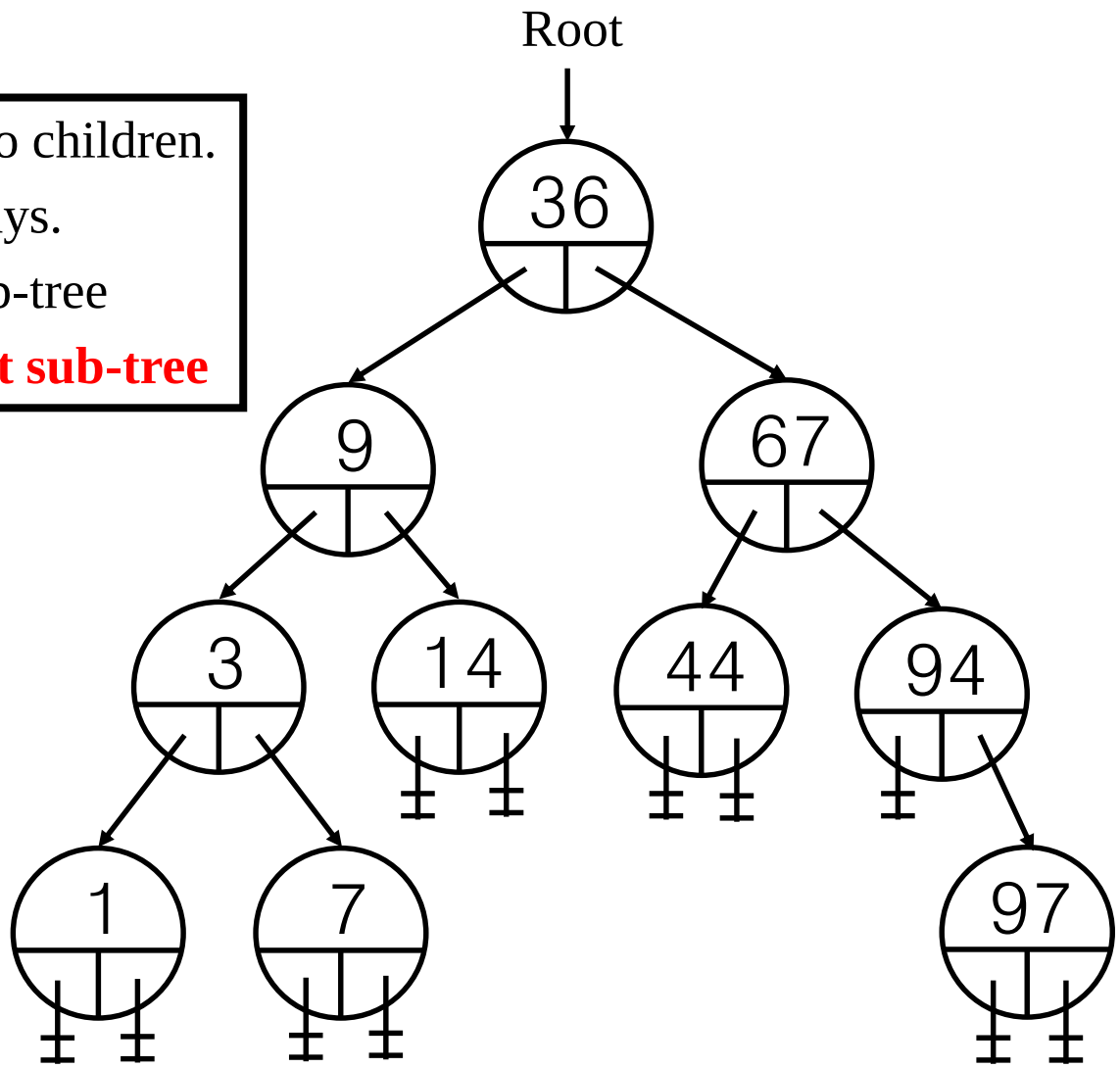
Node to be removed has two children.

We must choose the two ways.

Case1. Largest from left sub-tree

Case2. Smallest from right sub-tree

The final resulting tree –
still has search structure.

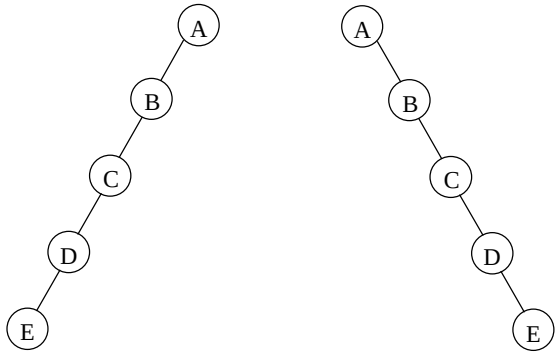


Binary Search Trees (7/8)

- Height of a binary search tree
  - The height of a binary search tree with n elements can become as large as n. (최악의 경우 높이 n)
  - It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$ on the average. (랜덤하게 만들면)
  - Search trees with a worst-case height of $O(\log_2 n)$ are called *balance search trees(균형 탐색 트리)* –높이가 $O(\log_2 n)$ *을 보장*

Binary Search Trees (8/8)

- Time Complexity
  - Searching, insertion, removal
    - O(*h*), where h is the height of the tree
  - Worst case - skewed binary tree경사이진트리 (최악의 이진탐색트리)
    - O(*n*), where n is the # of internal nodes
- Prevent worst case //최악의 경우를 회피방법
  - rebalancing scheme
  - AVL, 2-3, 2-3-4, and Red-black tree



(a) (b)

Find Maximum(최대값찾기)

- Find maximum is another common operation in binary search trees and is helpful when implementing deletion.
- Again the recursive thinking makes this implementation much easier to understand
- The idea is simple. If a node has a non-null pointer to the right this node cannot be the maximum as the element in the right must be greater. //루트에서 출발, 오른쪽으로 한없이 내려간다.

findMax ()

1. If head of the tree is null return null;
2. If the right pointer from the head is null head is the maximum
3. Else (it is not null) return the result of findMax in the right subtree

Find Minimum(최소값찾기)

- The idea of finding the minimum value in a binary search tree will also help the delete operation
- Its idea is very much like the maximum
  //루트에서출발, 왼쪽으로 한없이 내려간다

findMin ()

1. If head of the tree is null return null;
2. If the left pointer from the head is null head is the minimum
3. Else (it is not null) return the result of findMin in the left subtree

이원 탐색 트리의 Java 구현 및 검색 (2)

- 이원 탐색 트리 구축 및 탐색 프로그램

```
class TreeNode {
     String key;
     TreeNode left;
     TreeNode right;
}


class BinarySearchTree {
    private TreeNode rootNode; ;
```

이원 탐색 트리의 Java 구현 및 검색 (3)

```
private TreeNode insertKey(TreeNode T, String x) {
        // insert() 메소드에 의해 사용되는 보조 순환 메소드
  if (T==null) {
      TreeNode newNode = new TreeNode();
      newNode.key = x;
      return newNode;
  } else if (x.compareTo(T.key) < 0) {    // x < T.key이면 x를 T의 왼쪽
      T.left = insertKey(T.left, x);          // 서브트리에 삽입
      return T;
  } else if (x.compareTo(T.key) > 0) {     // x > T.key이면 x를 T의 오른쪽
      T.right = insertKey(T.right, x);         // 서브트리에 삽입
      return T;
  } else {                              // key값 x가 이미 T에 있는 경우
      return T;
  }
} // end insertKey()
```

이원 탐색 트리의 Java 구현 및 검색 (4)

```
void insert(String x) {
    rootNode = insertKey(rootNode, x);
} // end insert()

TreeNode find(String x) {      // 키값 x를 가지고 있는 TreeNode의
    TreeNode T = rootNode;     // 포인터를 반환
    int result;
    while (T != null) {
        if ((result = x.compareTo(T.key)) < 0) {
            T = T.left;
        } else if (result == 0) {
            return T;
        } else {
            T = T.right;
        } // end if
    }
    return T;
} // end find()
```

이원 탐색 트리의 Java 구현 및 검색 (5)

```
  private void printNode(TreeNode N) {   // print() 메소드에 의해
       if (N != null) {                  // 사용되는 순환 메소드
          System.out.print("(");
          printNode(N.left);
          System.out.print(N.key);
          printNode(N.right);
          System.out.print(")");
       }
   } // end printNode()

   void printBST() {   // 서브트리 구조를 표현하는 괄호 형태로 트리를 프린트
       printNode(rootNode);
       System.out.println();
   }  // end printBST()
}   // end BinarySearchTree class
```

이원 탐색 트리의 Java 구현 및 검색 (6)

```
class BinarySearchTreeTest {
   public static void main(String args[]) {
      BinarySearchTree T = new BinarySearchTree();
      // 그림 8.6의 BST를 구축
      T.insert("S");
      T.insert("J");
      T.insert("B");
      T.insert("D");
      T.insert("U");
      T.insert("M");
      T.insert("R");
      T.insert("Q");
      T.insert("A");
      T.insert("G");
      T.insert("E");
      // 구축된 BST를 프린트
      System.out.println(" The Tree is:");
      T.printBST();
      System.out.println();
      // 스트링 "R"을 탐색하고 프린트
```

이원 탐색 트리의 Java 구현 및 검색 (7)

```
        System.out.println(" Search For ₩"R₩"");
        TreeNode N = T.find("R");
        System.out.println("Key of node found = " + N.key);
        System.out.println();
        // 스트링 "C"를 탐색하고 프린트
        System.out.println(" Search For ₩"C₩"");
        TreeNode P = T.find("C");
        if (P != null) {
           System.out.println("Key of node found = " + P.key);
        } else {
           System.out.println("Node that was found = null");
        }
        System.out.println();
    } // end main()
} // end BinarySearchTreeTest class
```

Heaps(히프)

Heaps (1/6)

- The heap abstract data type
  - **Definition:** A *max*(*min*) *tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children. A *max* (*min*) *heap* is a complete binary tree (완전이진트리)that is also a *max* (*min*) *tree*
  - **Basic Operations:**
    - creation of an empty heap
    - insertion of a new element into a heap
    - deletion of the largest element from the heap

Heaps (2/6)

- The examples of max heaps and min heaps루트가 가장큰 맥스히프,루트가 가장 작은 민히프
  - Property: The root of max heap (min heap) contains the

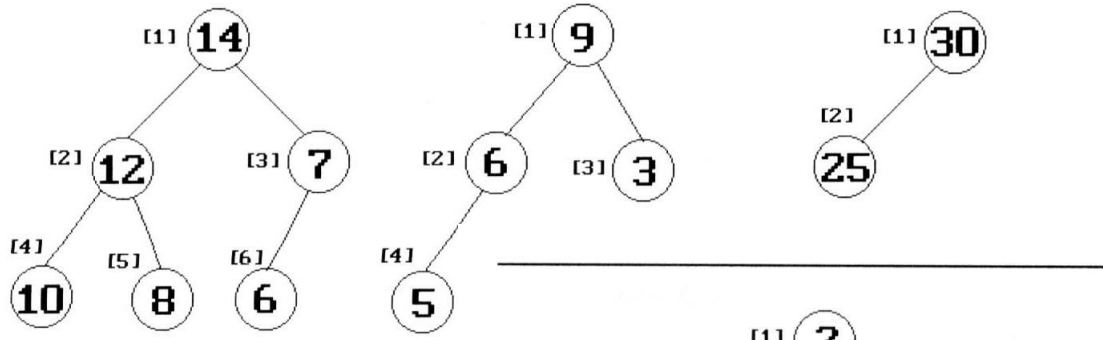


Figure 5.25: Sample max heaps

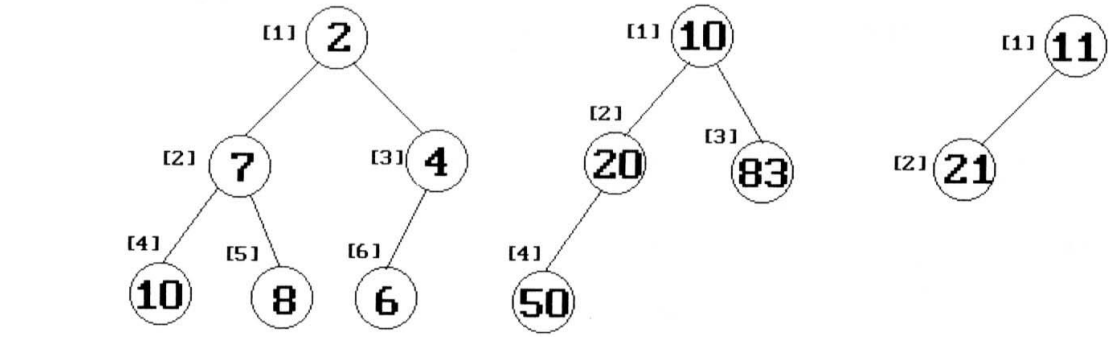


Figure 5.26: Sample min heaps

Heaps (3/6)

- Abstract data type of Max Heap

ADT Heap

Data : n>0 원소로 구성된 완전 이진 트리로 각 노드의 키값은 그의 자식 노드의 키값보다 작지 않다.

Operators :

H∈Heap; e ∈Element;
createHeap() := create an empty heap;
insertHeap(H,e) := insert a new item e into H
isEmpty(H) := if the heap H is empty
then return true
else return false
deleteHeap(H) := if isEmpty(H) then null
else {
e ← the largest element in H;
remove the largest element in H;
return e;
}

End Heap

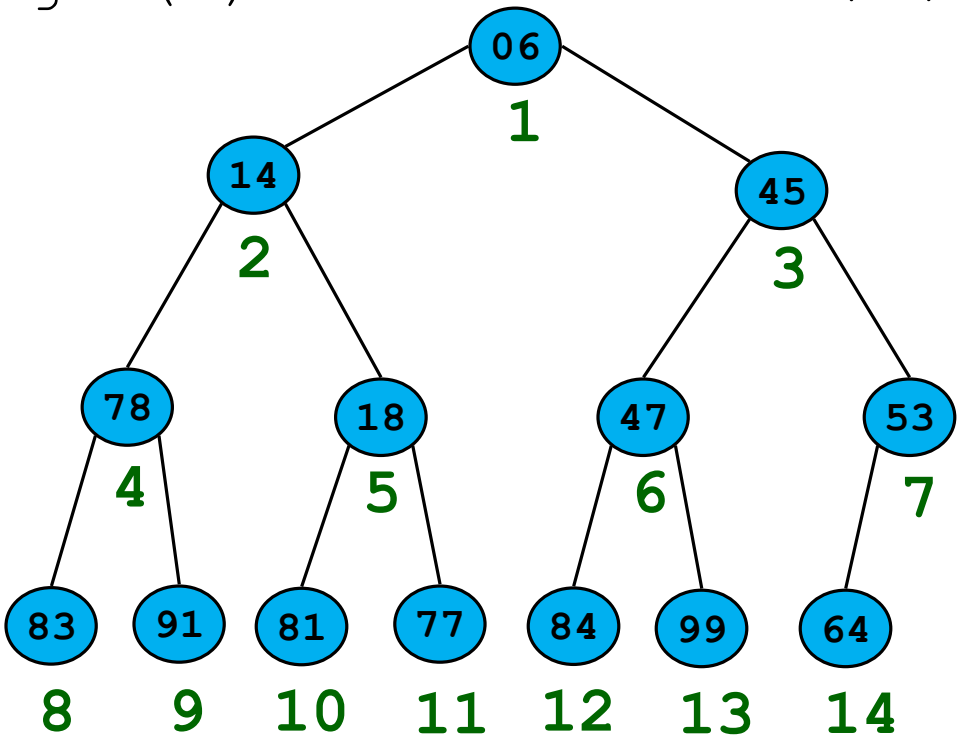
Heaps (4/6)

- Queue : FIFO 큐에 응응
- Priority queues
  - Heaps are frequently used to implement *priority queues* *(히프는 우선순위큐 구현에 최적)*
  - delete the element with highest (lowest) priority
  - insert the element with arbitrary priority
  - Heaps is the only way to implement priority queue(우선순위큐)

Binary Heaps: Array Implementation

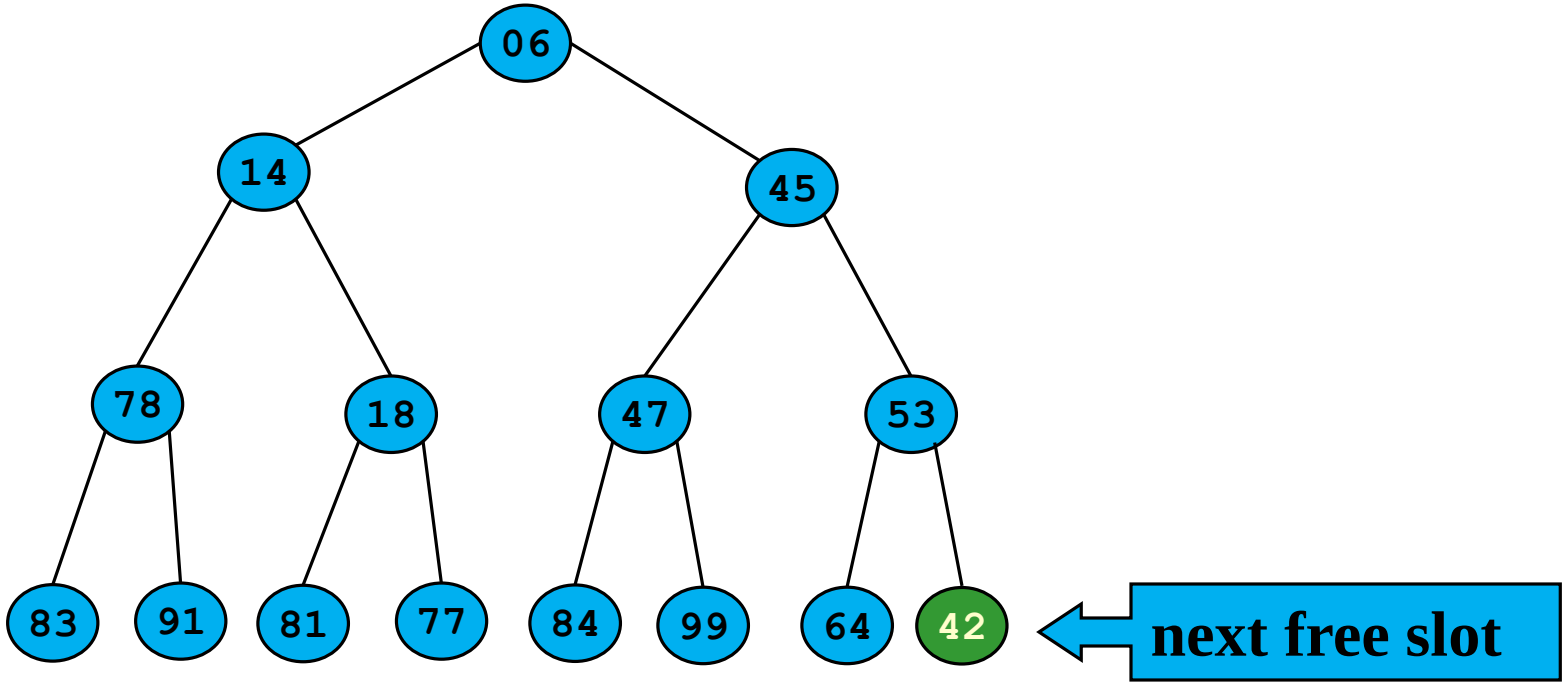
배열로구현

- Implementing binary heaps.
  - Use an array: no need for explicit parent or child pointers.
    - `Parent(i) = ⌊i/2⌋` 아버지의 위치
    - `Left(i) = 2i` 왼쪽 자식의 위치
    - `Right(i) = 2i + 1` 오른쪽 자식의 위치



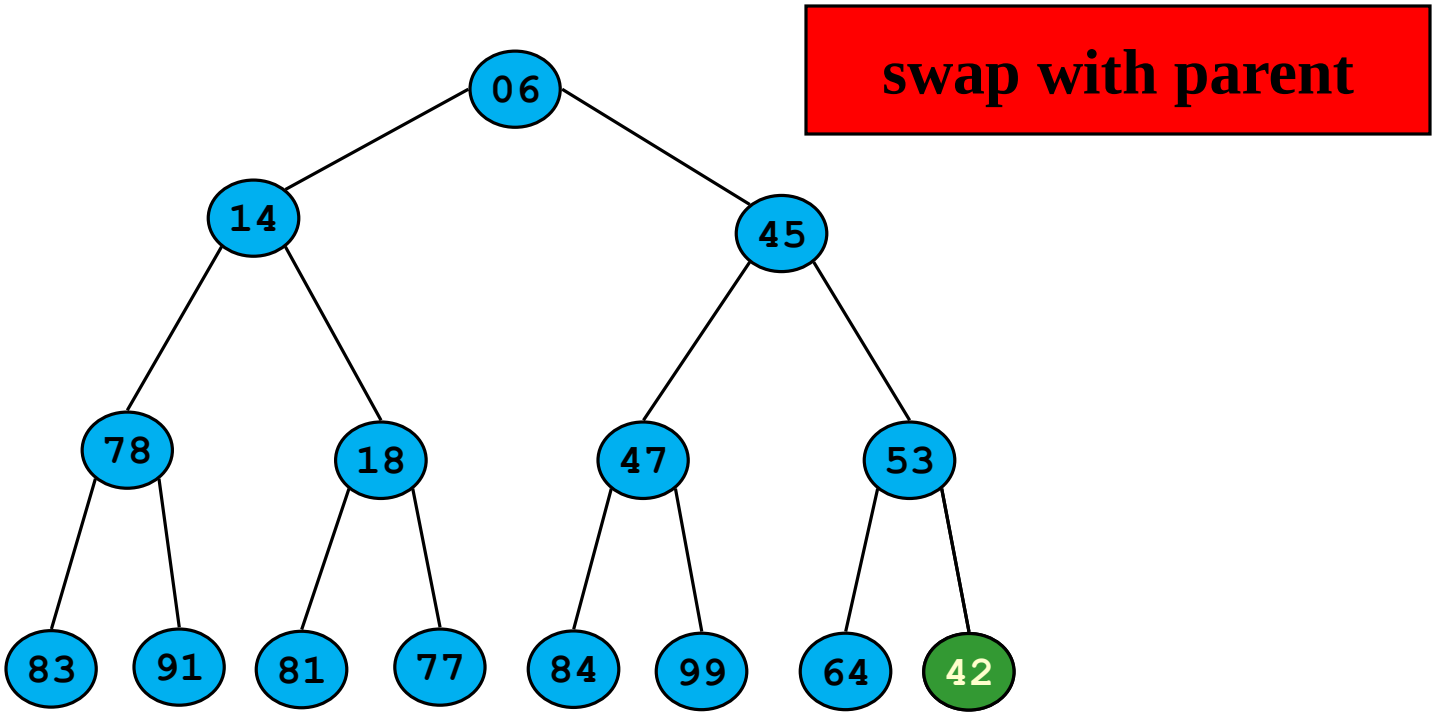
Binary Heap: Insertion(노드삽입)

- Insert element x into heap.히프에 x삽입
  - Insert into next available slot. 마지막+1번에추가
  - Bubble up until it's heap ordered. 아버지와비교
    - Peter principle: nodes rise to level of incompetence



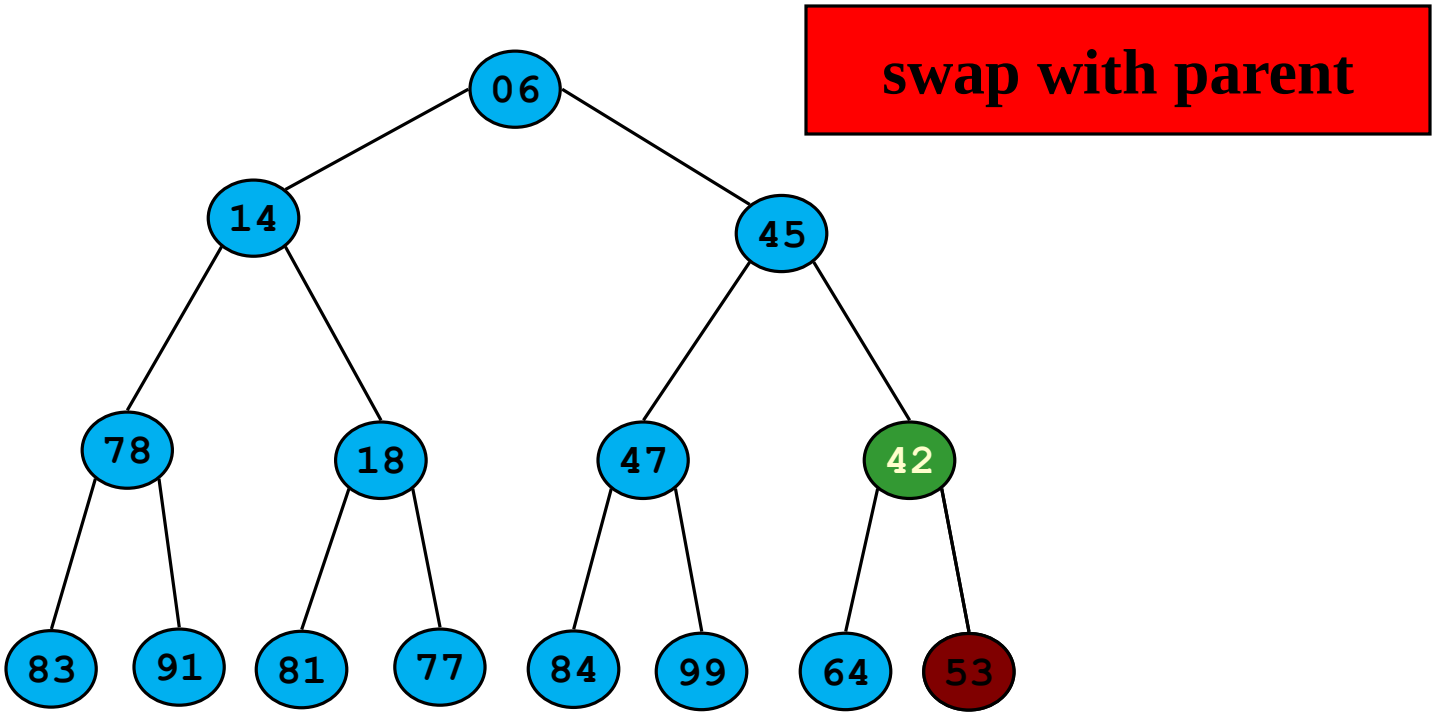
Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence



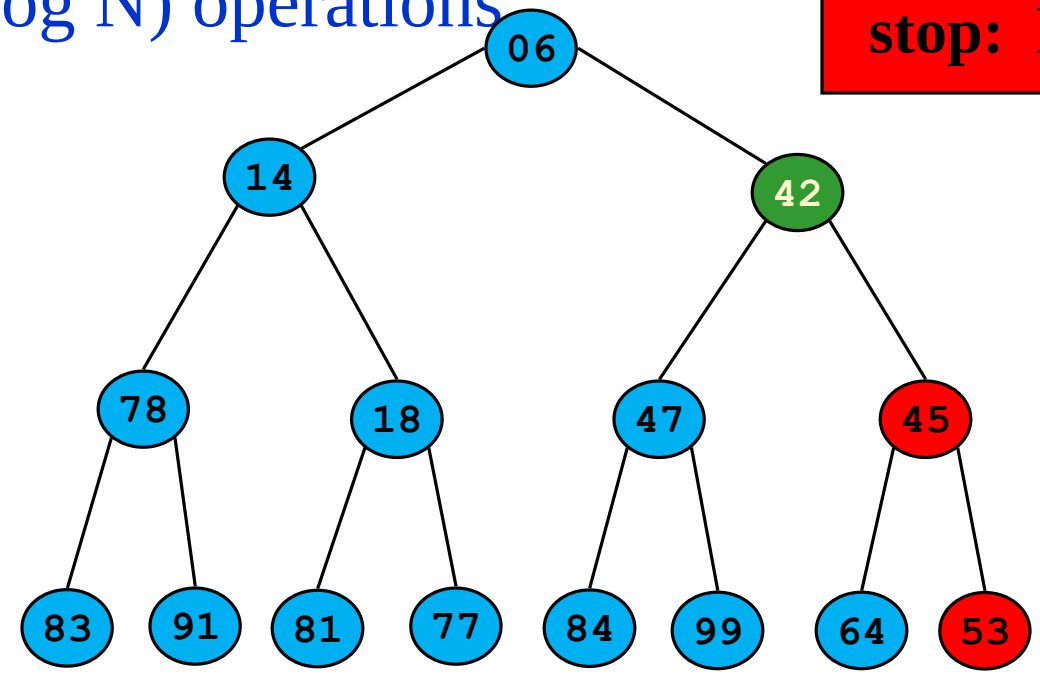
Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

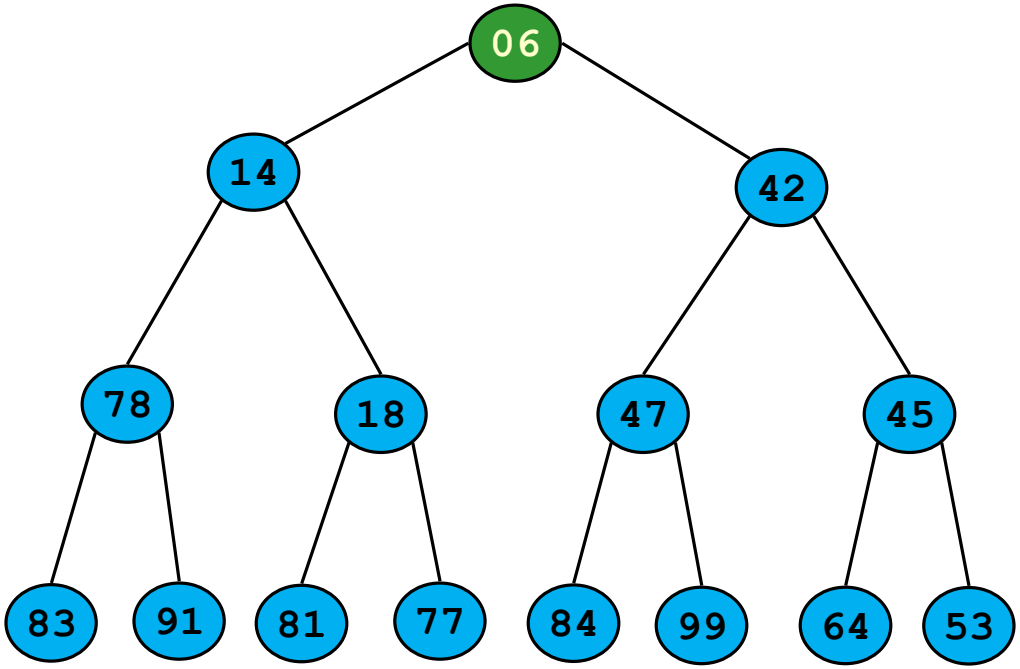
- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence
  - O(log N) operations



stop: heap ordered

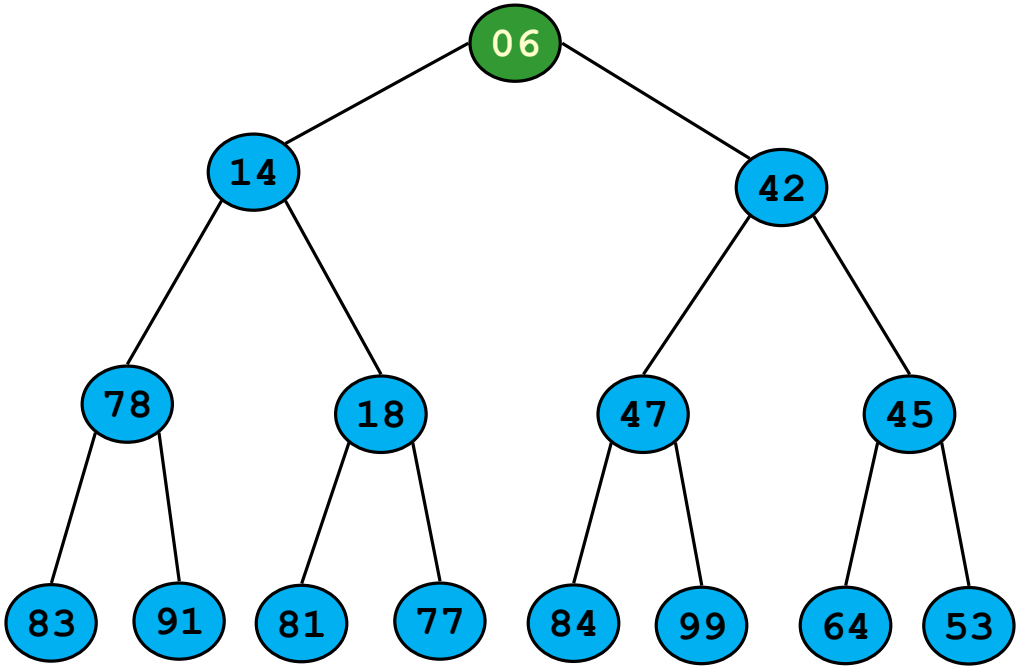
Binary Heap: Delete Min삭제

- 히프에서 삭제는 루트만을 삭제함.



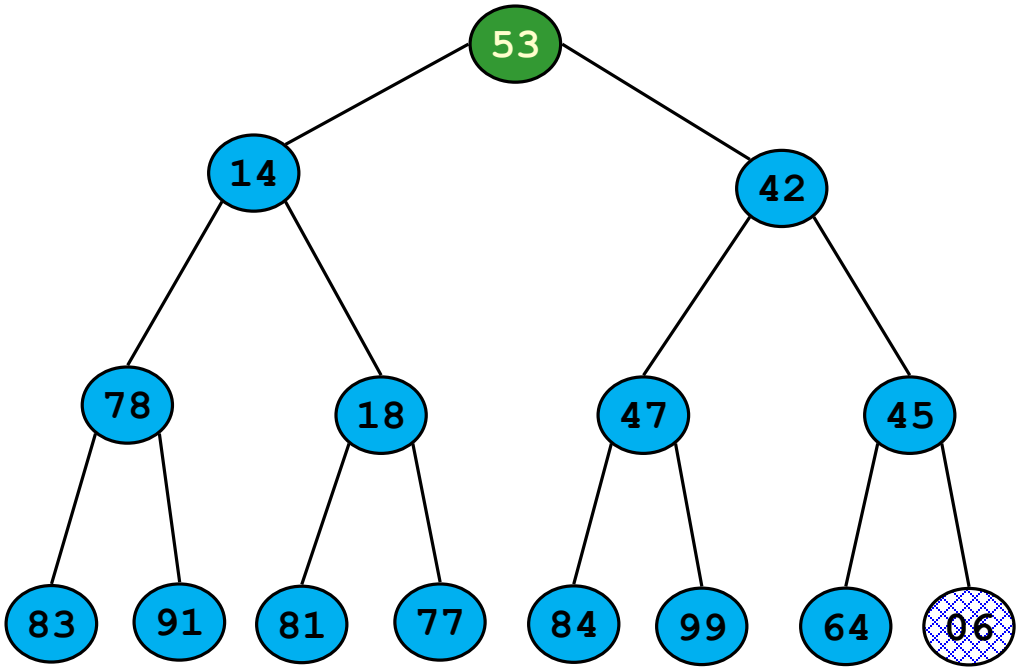
Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



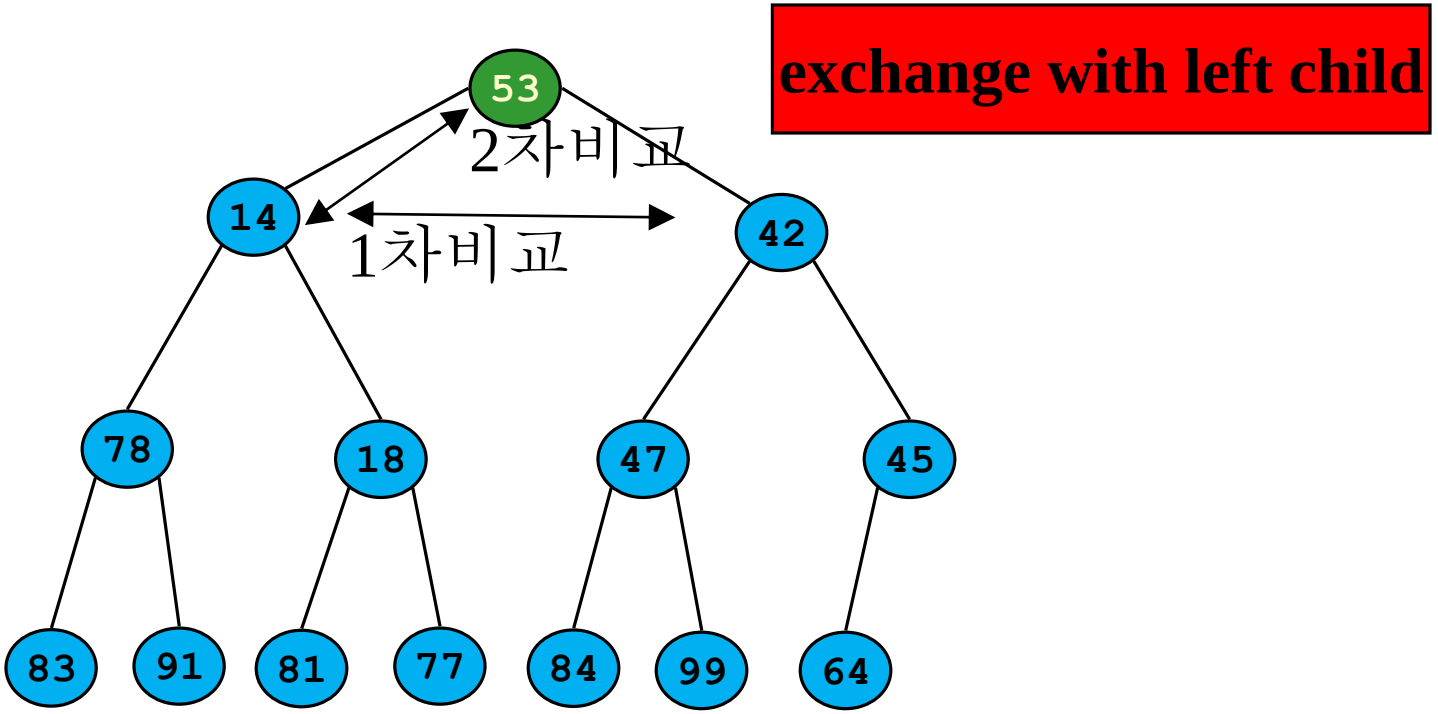
Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf. 루트와 가장오른쪽원소를 교환
  - Bubble root down until it's heap ordered. 비교하며내려옴
    - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

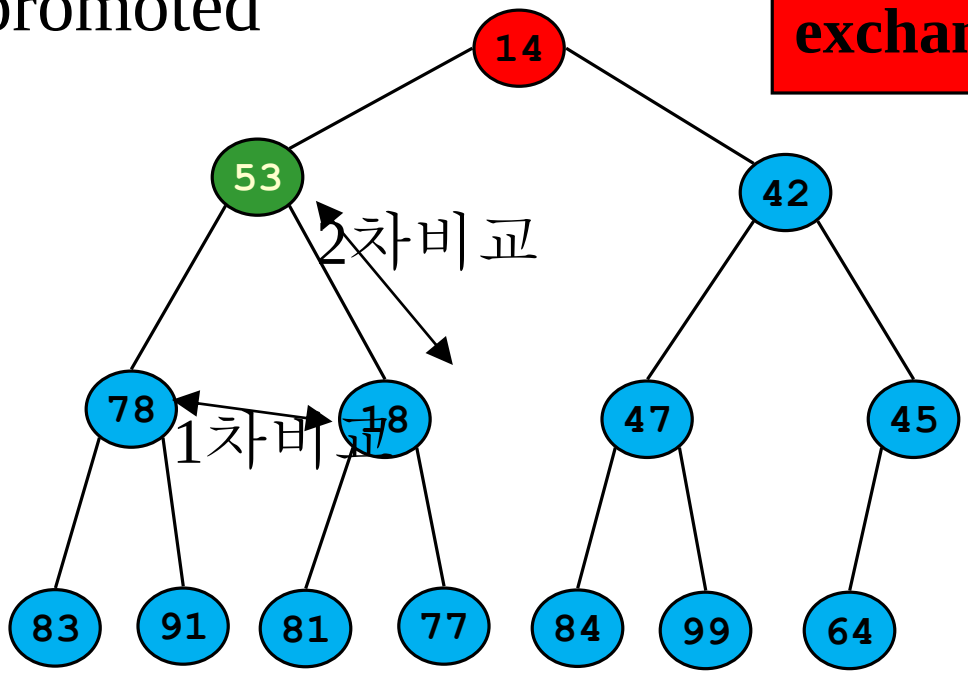
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

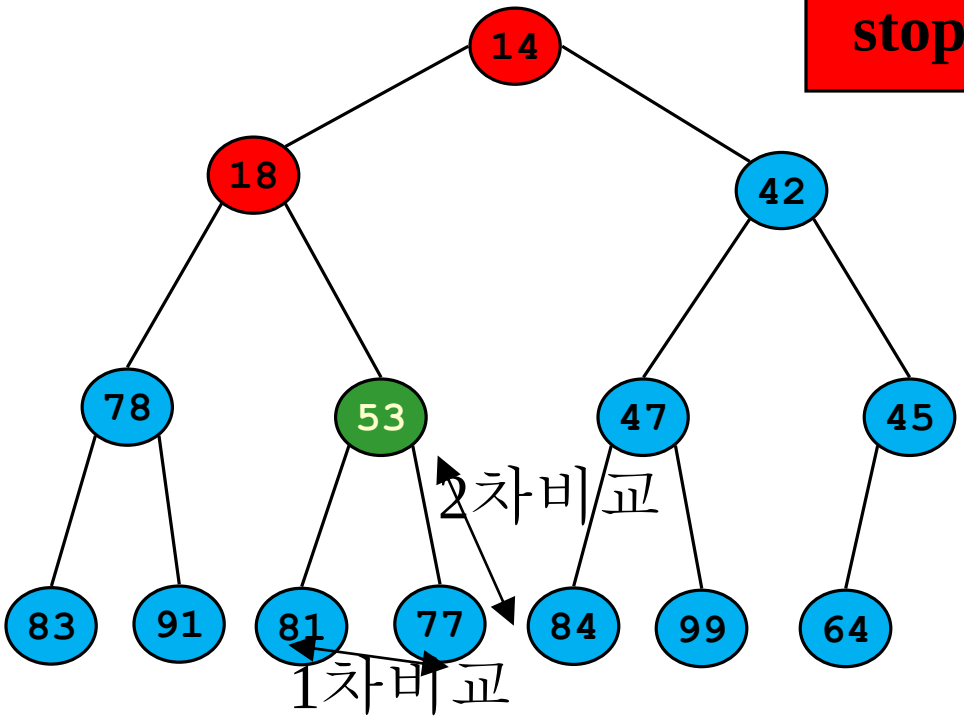
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted

exchange with right child



Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted
  - O(log N) operations.



stop: heap ordered

Insertion algorithm삽입알고리즘

```
insertHeap(Heap,e)
        // 순차 표현으로 구현된 최대 히프
        // 원소 e를 히프 Heap에 삽입, n은 현재 히프의 크기(원소 수)
   if (n = maxSize) then heapFull;   // 히프가 만원이면 히프 크기를 확장
   n←n+1;   // 새로 첨가될 노드 위치
   for (i←n; ; ) do {
      if (i = 1) then exit;   // 루트에 삽입
      if(e.key ≤ Heap[  i/2  ].key) then exit;   // 삽입할 노드의 키값과
                                 // 부모 노드 키값을 비교
      Heap[i] ← Heap[  i/2  ];   // 부모 노드 키값을 자식노드로 이동
      i ←   i/2  ;
   }
   Heap[i] ← e;
end insertHeap()
```

Deletion algorithm

```
deleteHeap(heap)
        // 히프로부터 원소 삭제, n은 현재의 히프 크기(원소 수)
    if (n=0) then return error;   // 공백 히프
    item ← heap[1];    // 삭제할 원소
    temp ← heap[n];    // 이동시킬 원소
    n ← n-1;       // 히프 크기(원소 수)를 하나 감소
    i ← 1;
    j ← 2;   // j는 i의 왼쪽 자식 노드
    while (j ≤ n) do {
        if (j < n) then if (heap[j] < heap[j+1])
            then j ← j+1;   // j는 값이 큰 자식을 가리킨다.
        if (temp ≥ heap[j]) then exit;
        heap[i] ← heap[j];   // 자식을 한 레벨 위로 이동
        i ← j;
        j ← j*2;        // i와 j를 한 레벨 아래로 이동
    }
    heap[i] ← temp;
    return item;
end deleteHeap()
```

Building a Heap(히프만들기)

Build - Max - Heap (A)

$heap\text{-}size[A] \leftarrow length[A]$;

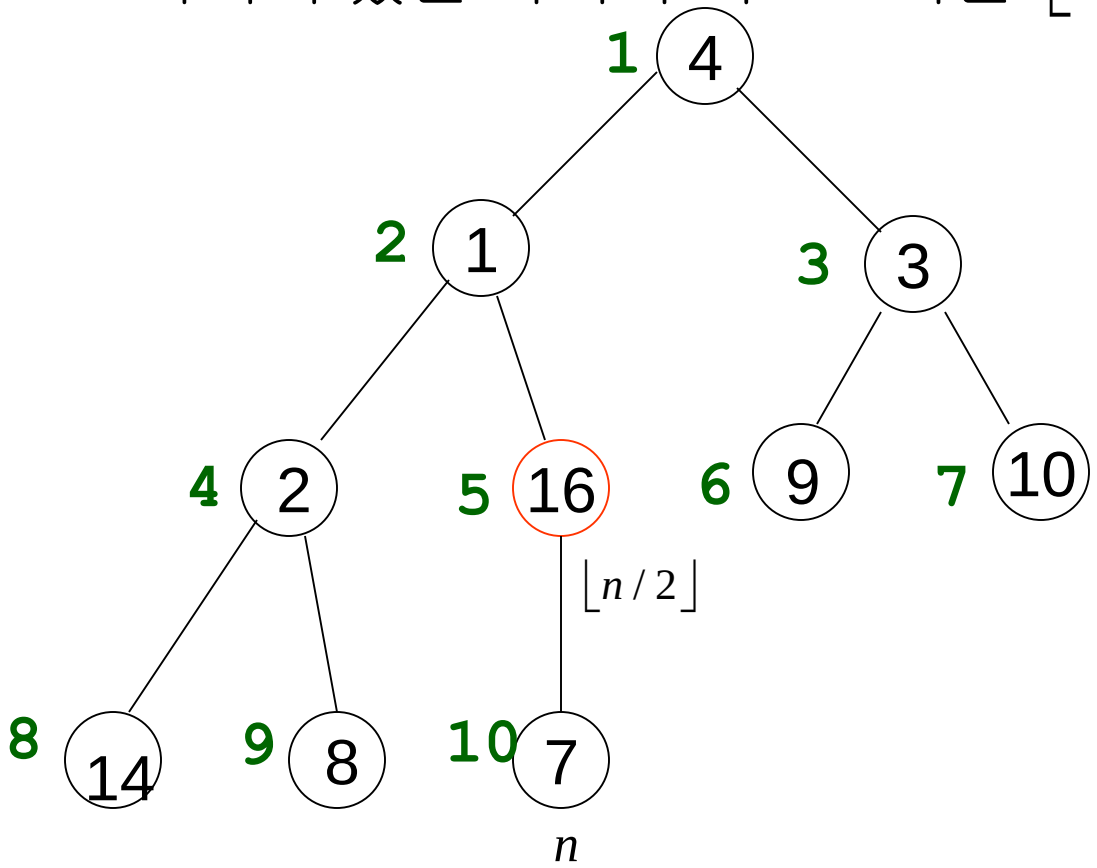
for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1

do Max - Heapify (A, i);

e.g., 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

The last location who has a child is $\lfloor n/2 \rfloor$.

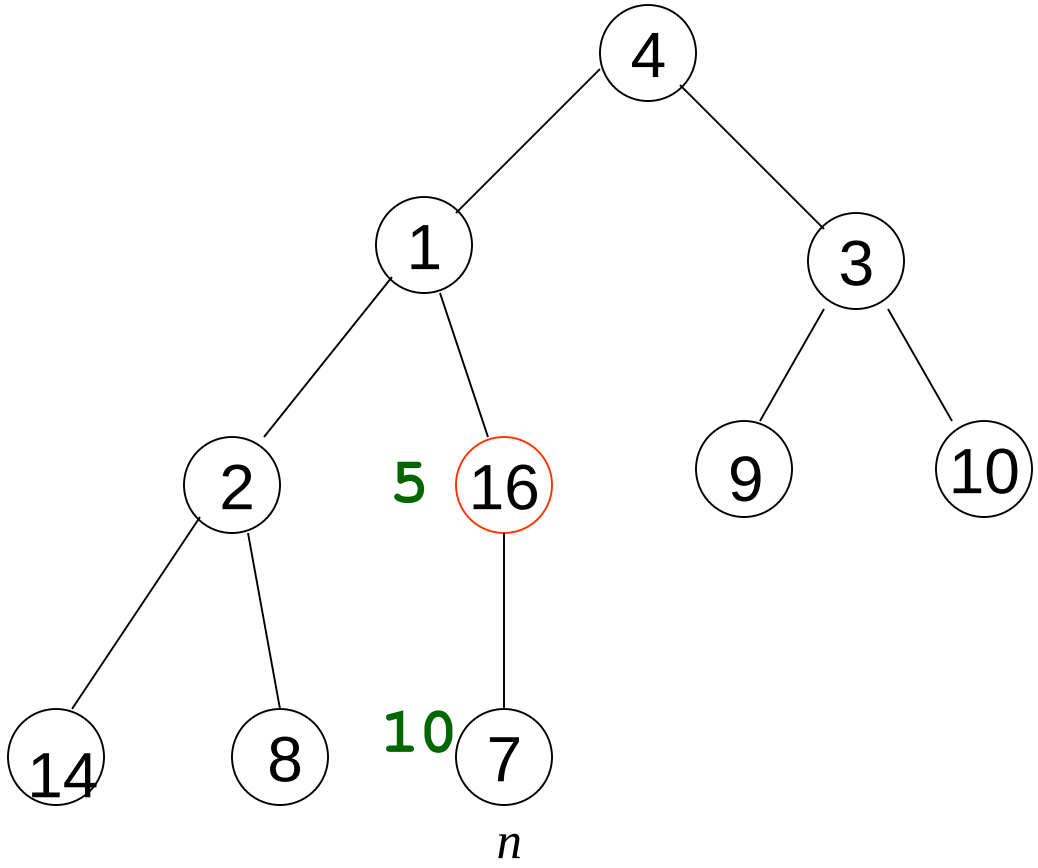
자식이 있는 마지막 서브트리는 $\lfloor n/2 \rfloor$에 위치



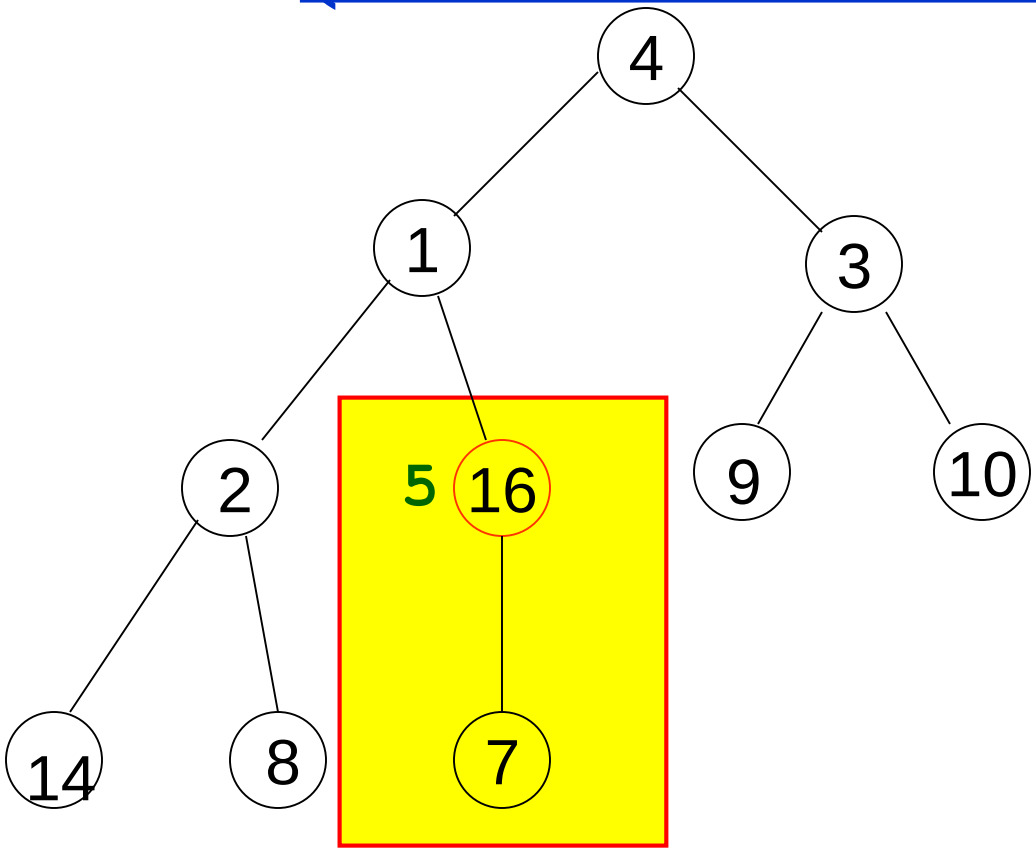
e.g., 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

그냥 평범한 이진 트리임. 히프도 아니고, 이진탐색도 아님.

1. 첫스탭 : 제일 마지막 10번방에 원소가존재.
그러면 아버지는 5번방. 5번방을 루트로 하는 서브트리부터시작

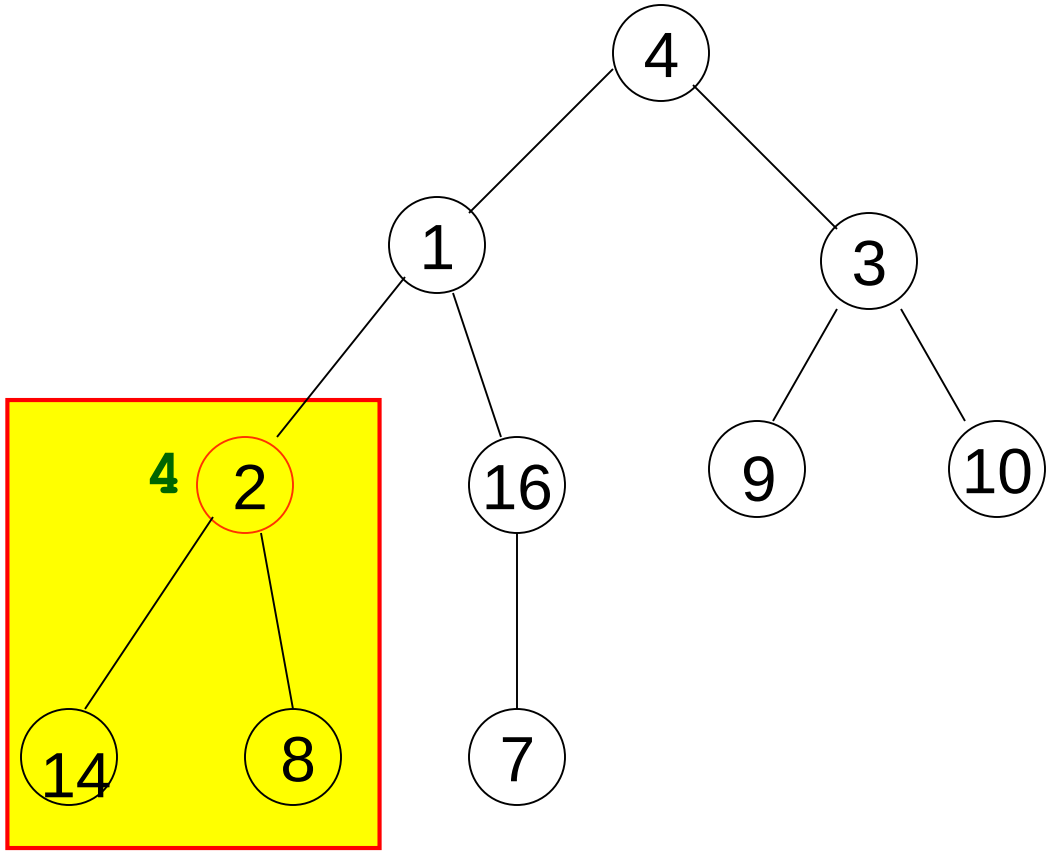


Max-Heapify(node5)
(최대히프만들기)



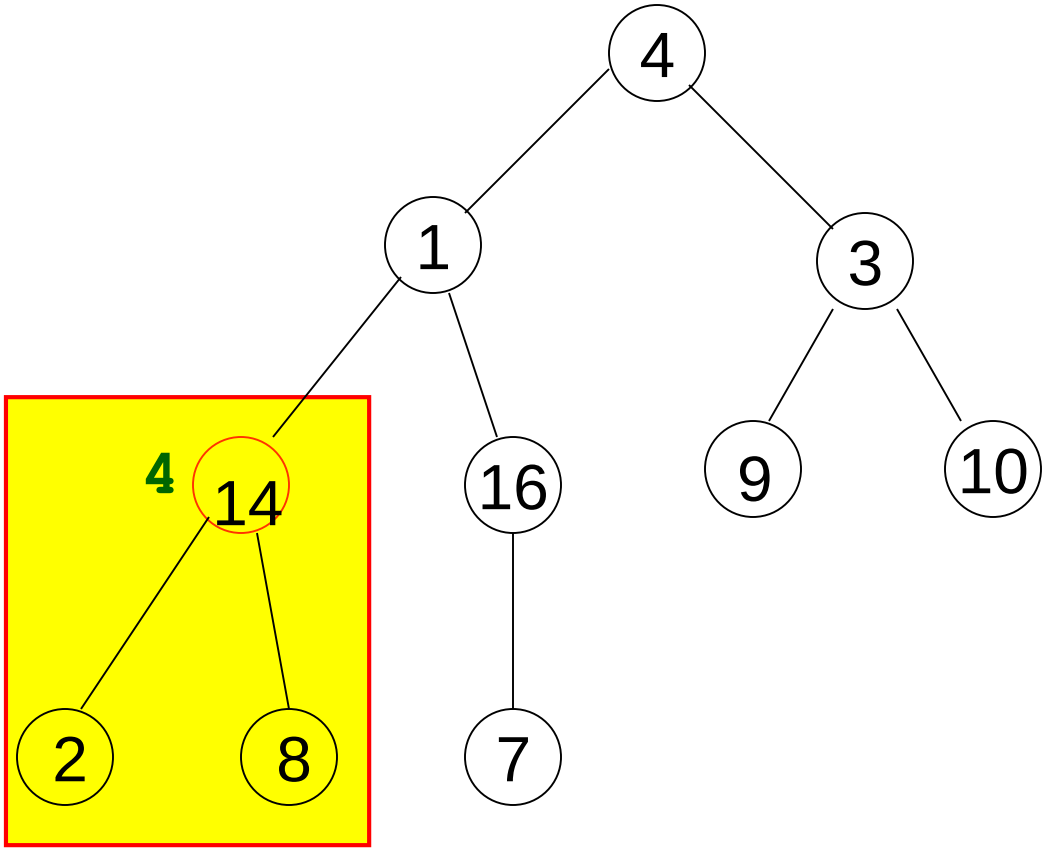
5를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node 4)

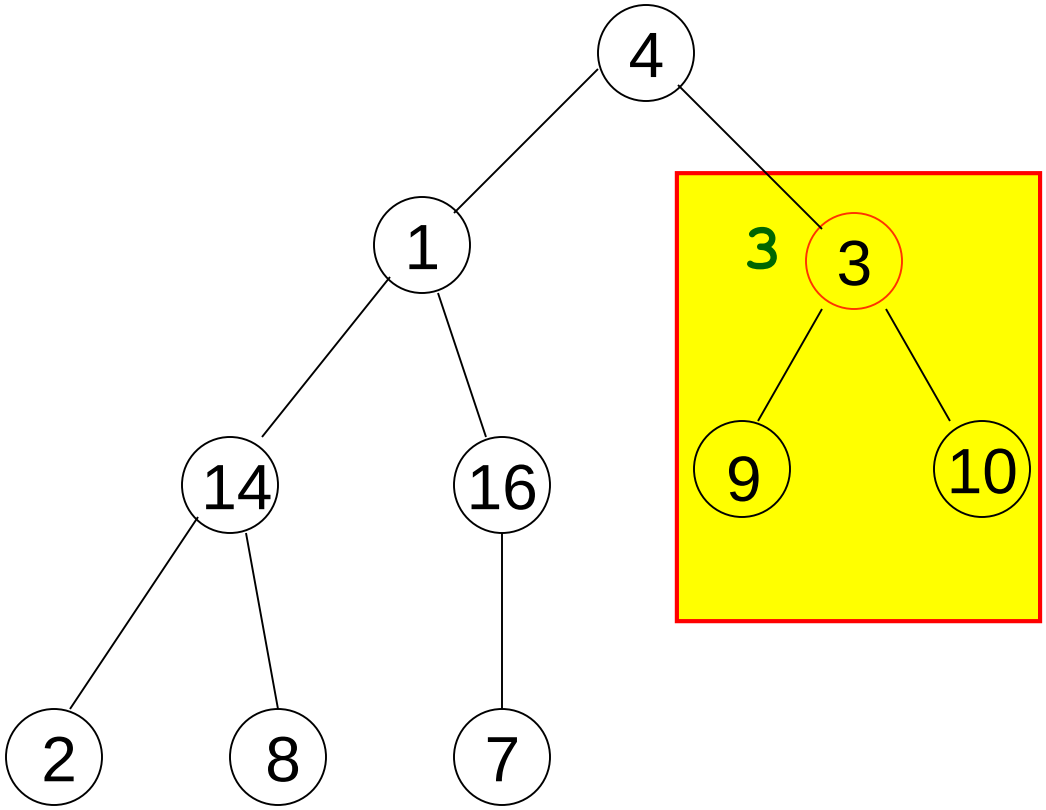


4를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node4)

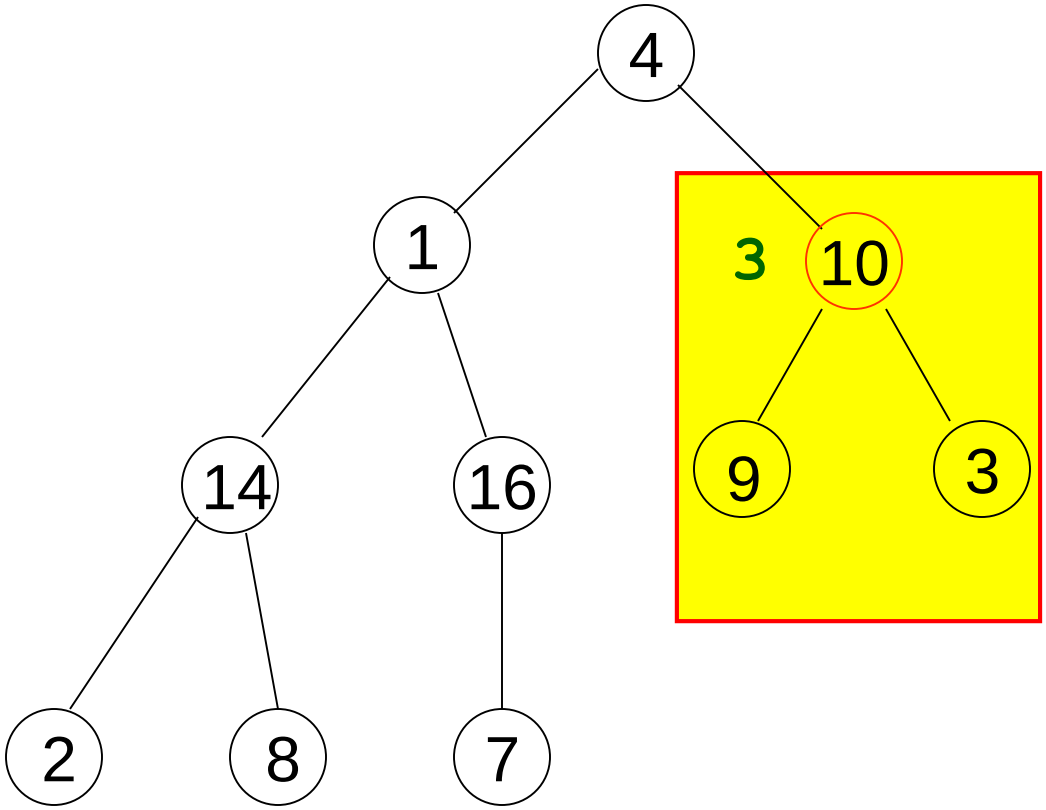


Max-Heapify(node3)

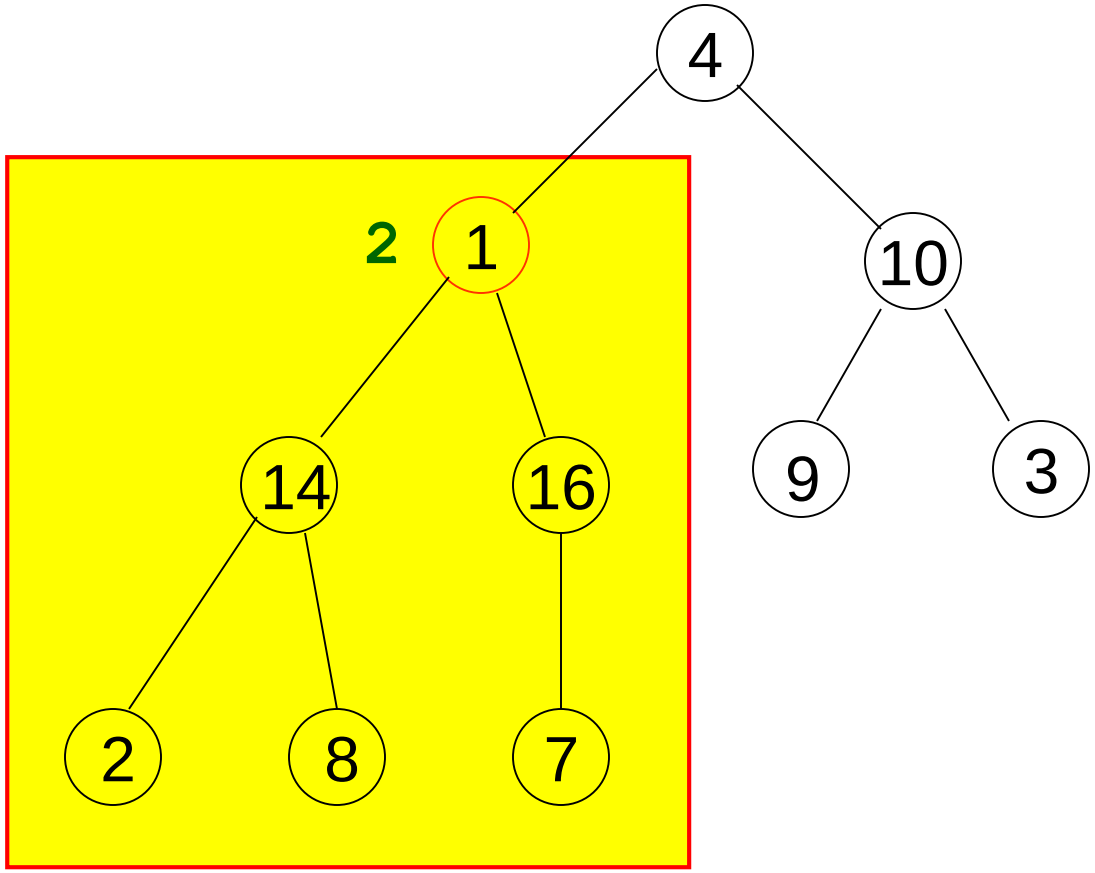


3을 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node3)

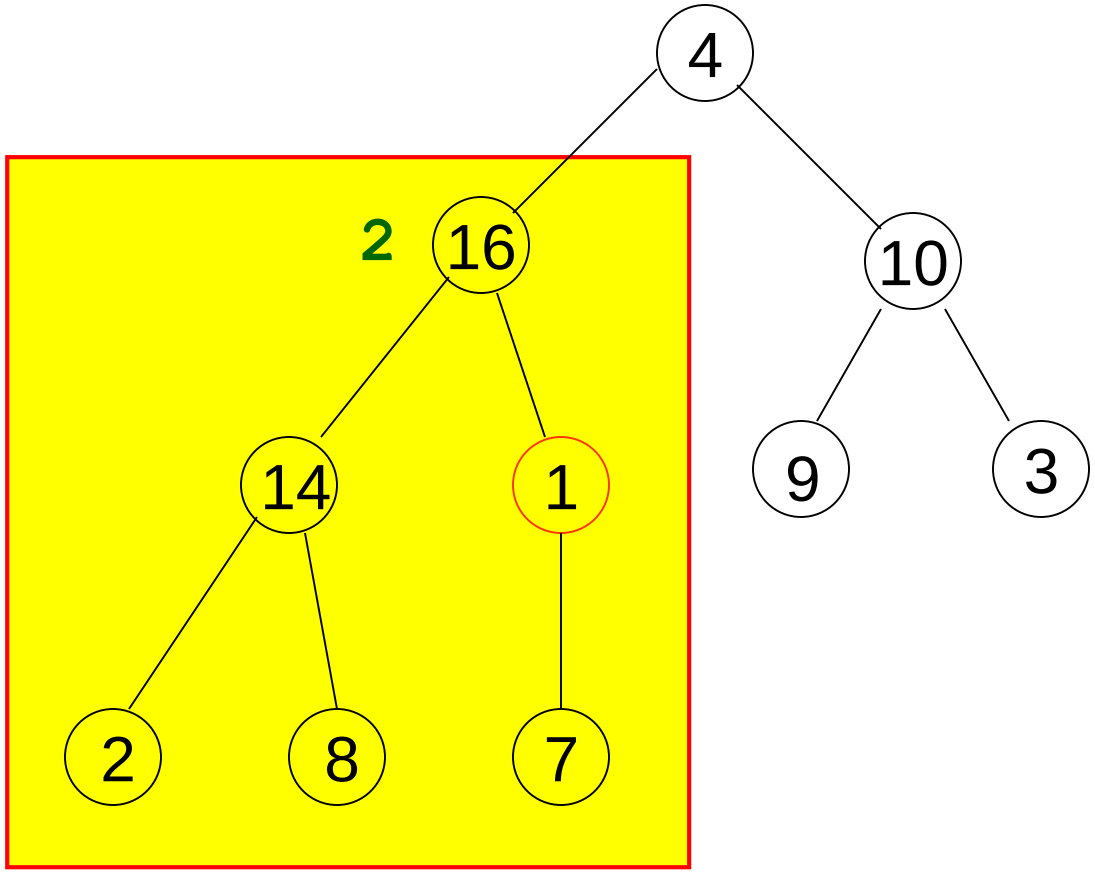


Max-Heapify(node2)

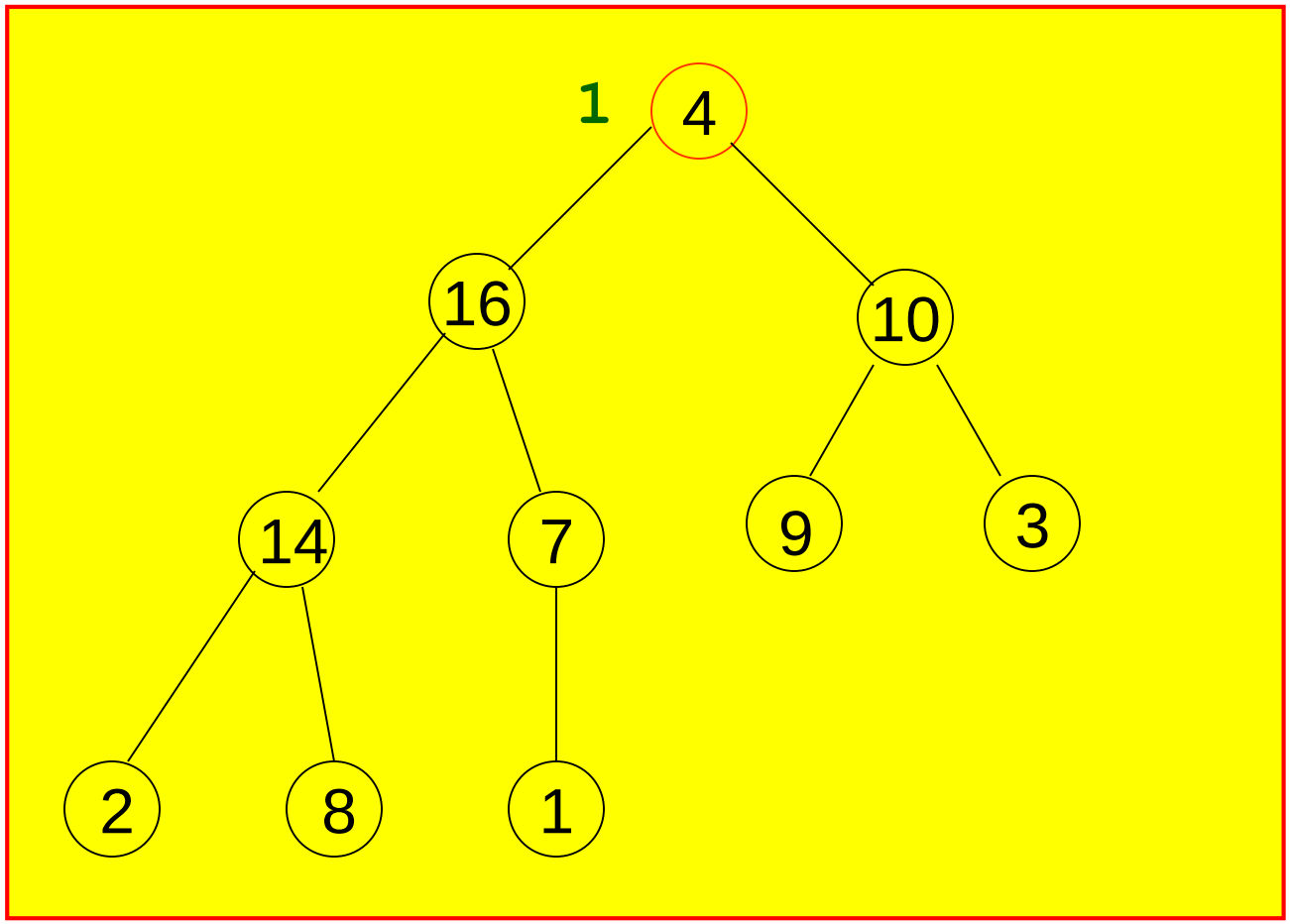


2를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node2)

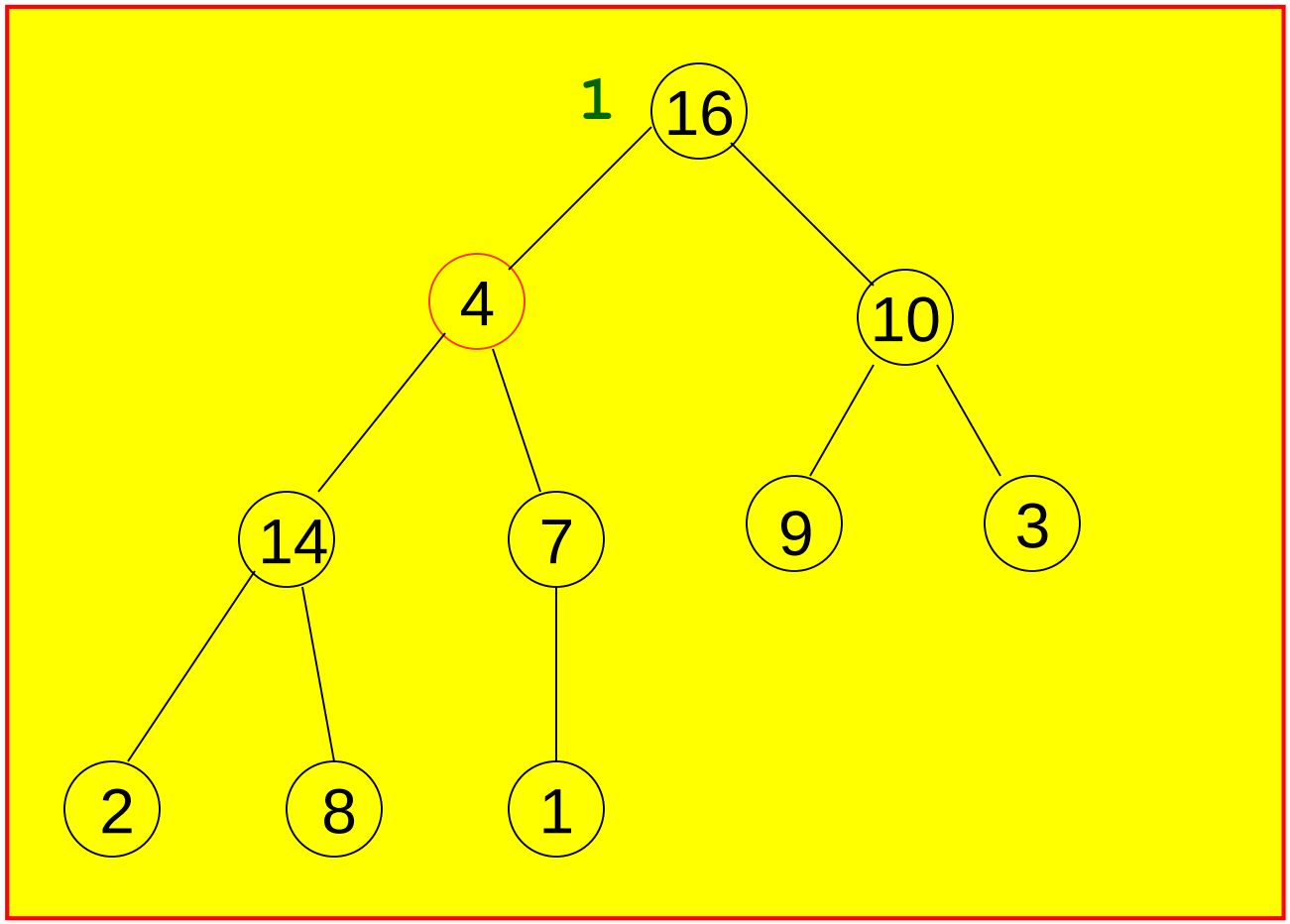


Max-Heapify(node1)

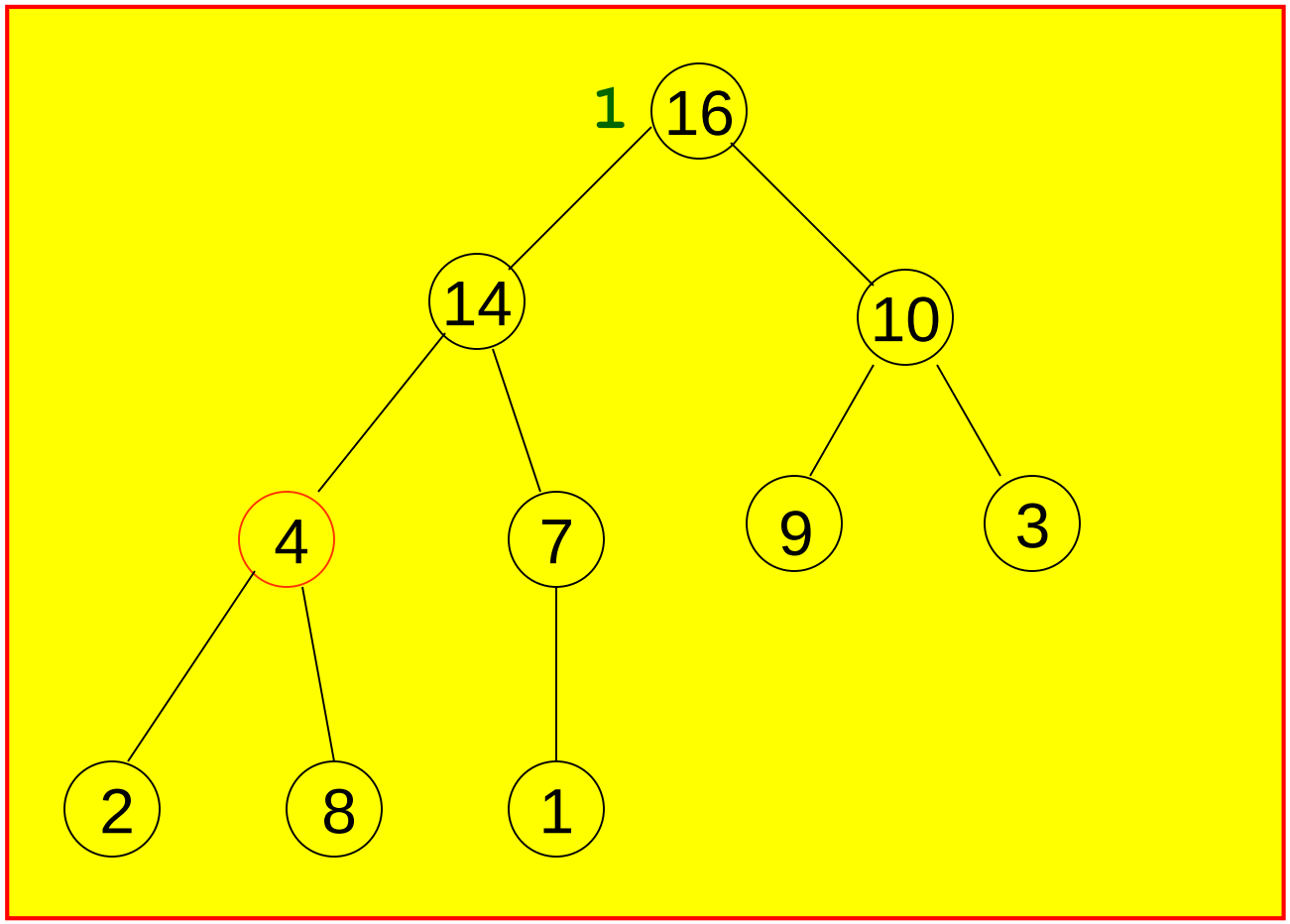


1을 루트로 하는 서브트리를 히프로 만든다

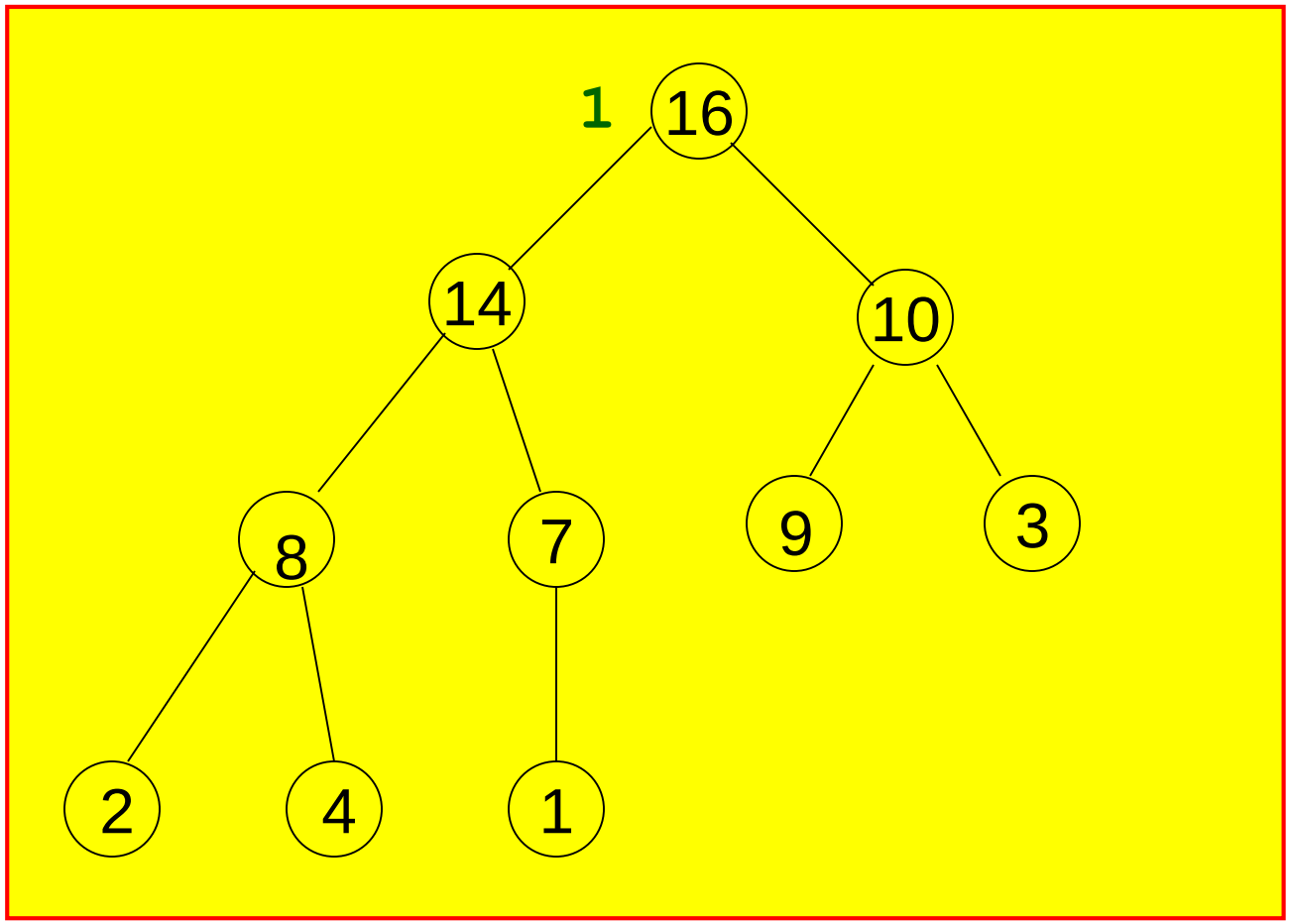
Max-Heapify(node1)



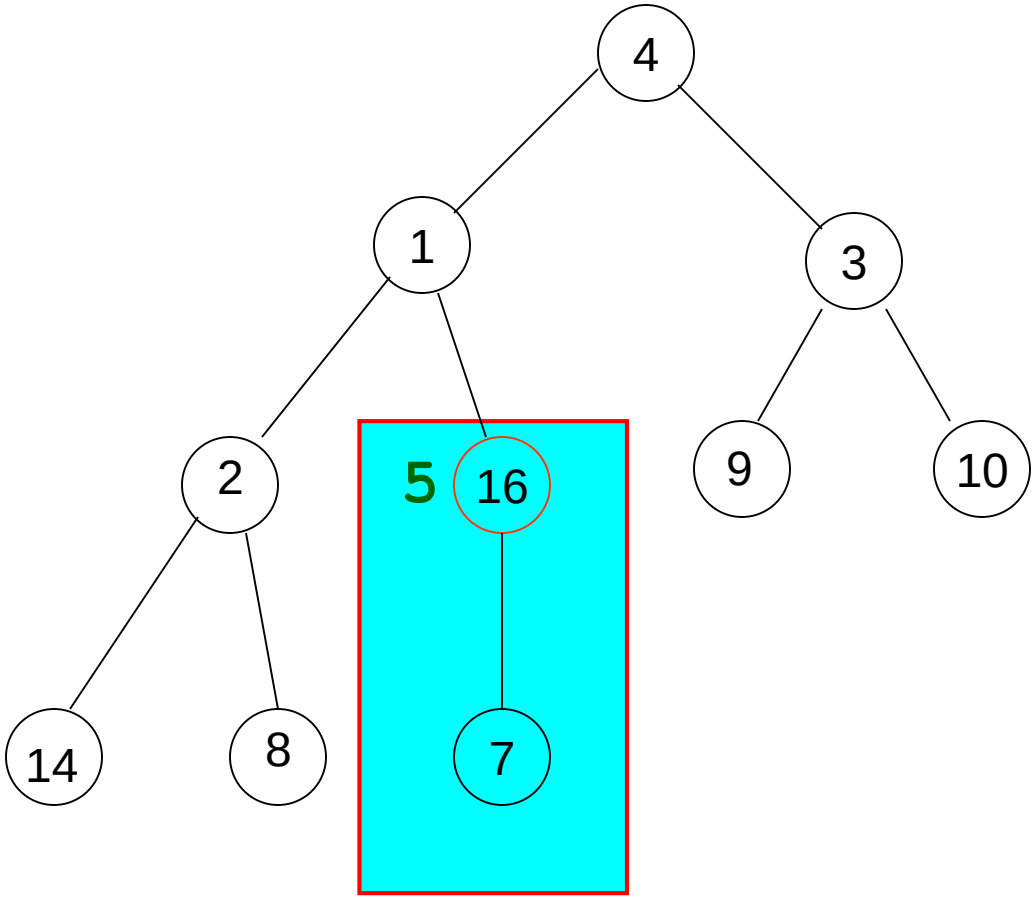
Max-Heapify(node1)



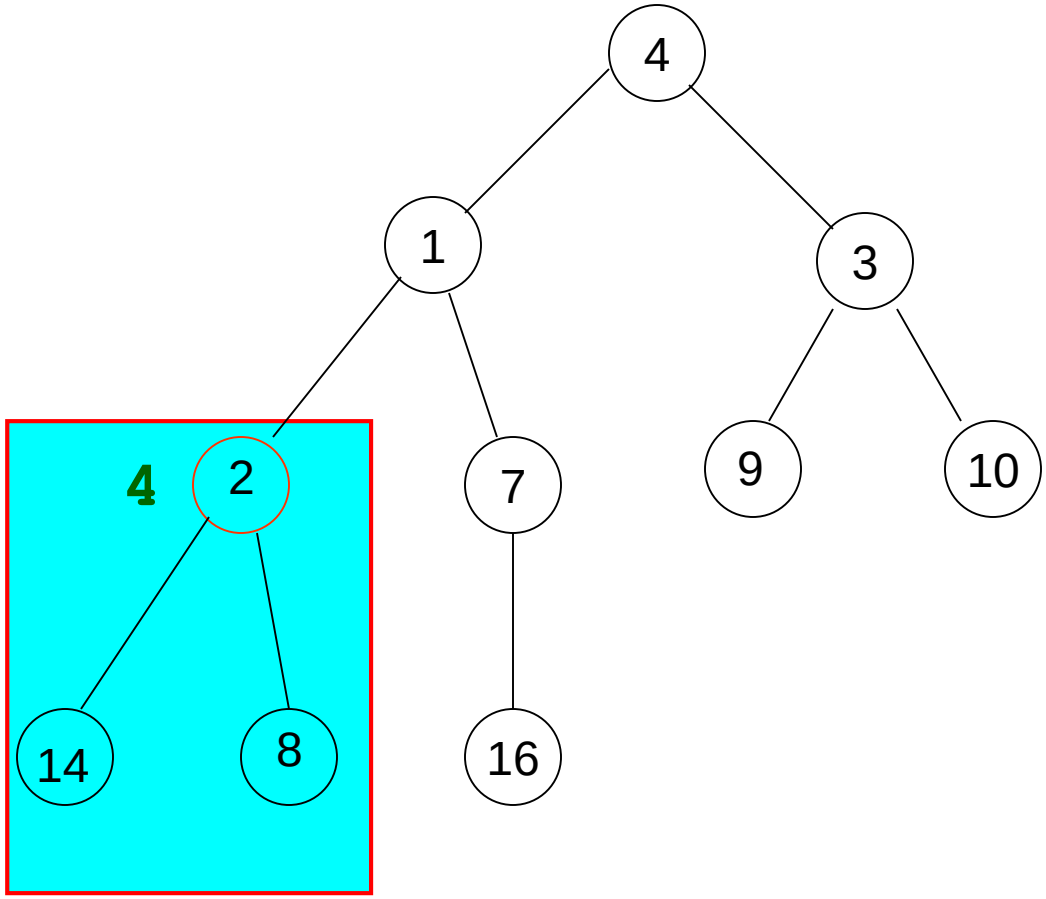
Max-Heapify(node1)



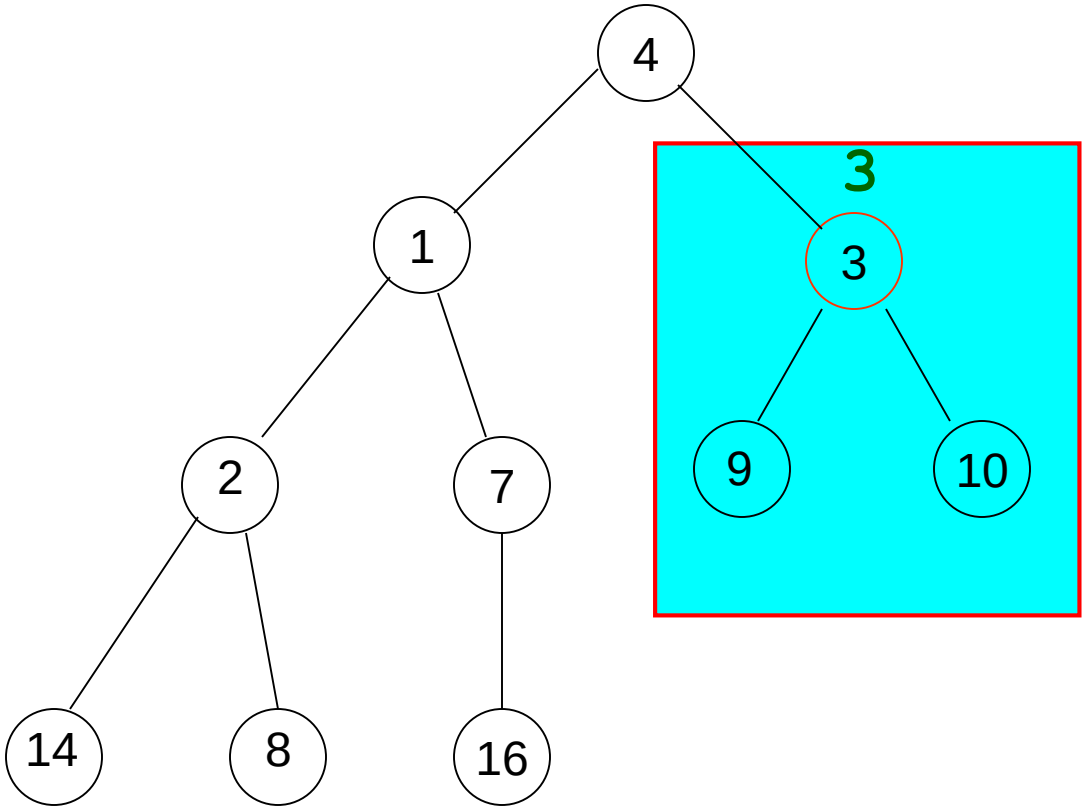
Min-Heapify(노드5)(최소히프만들기)



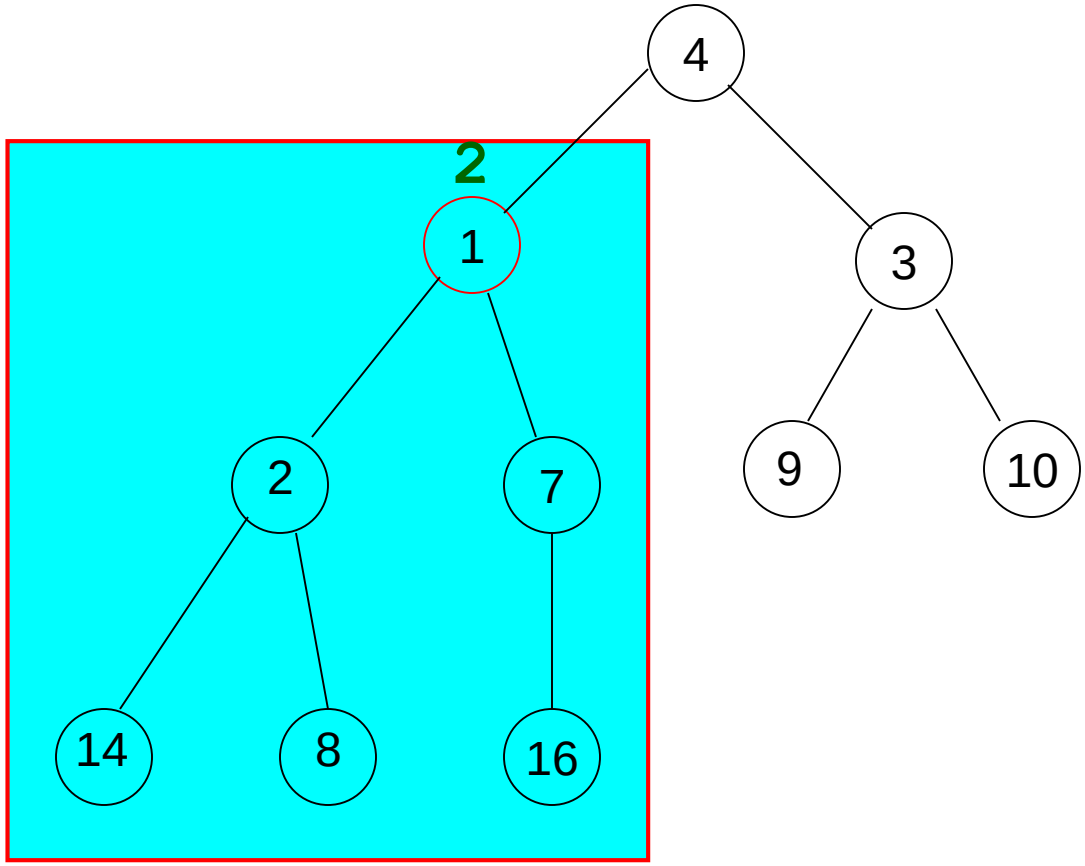
Min-Heapify(노드4)



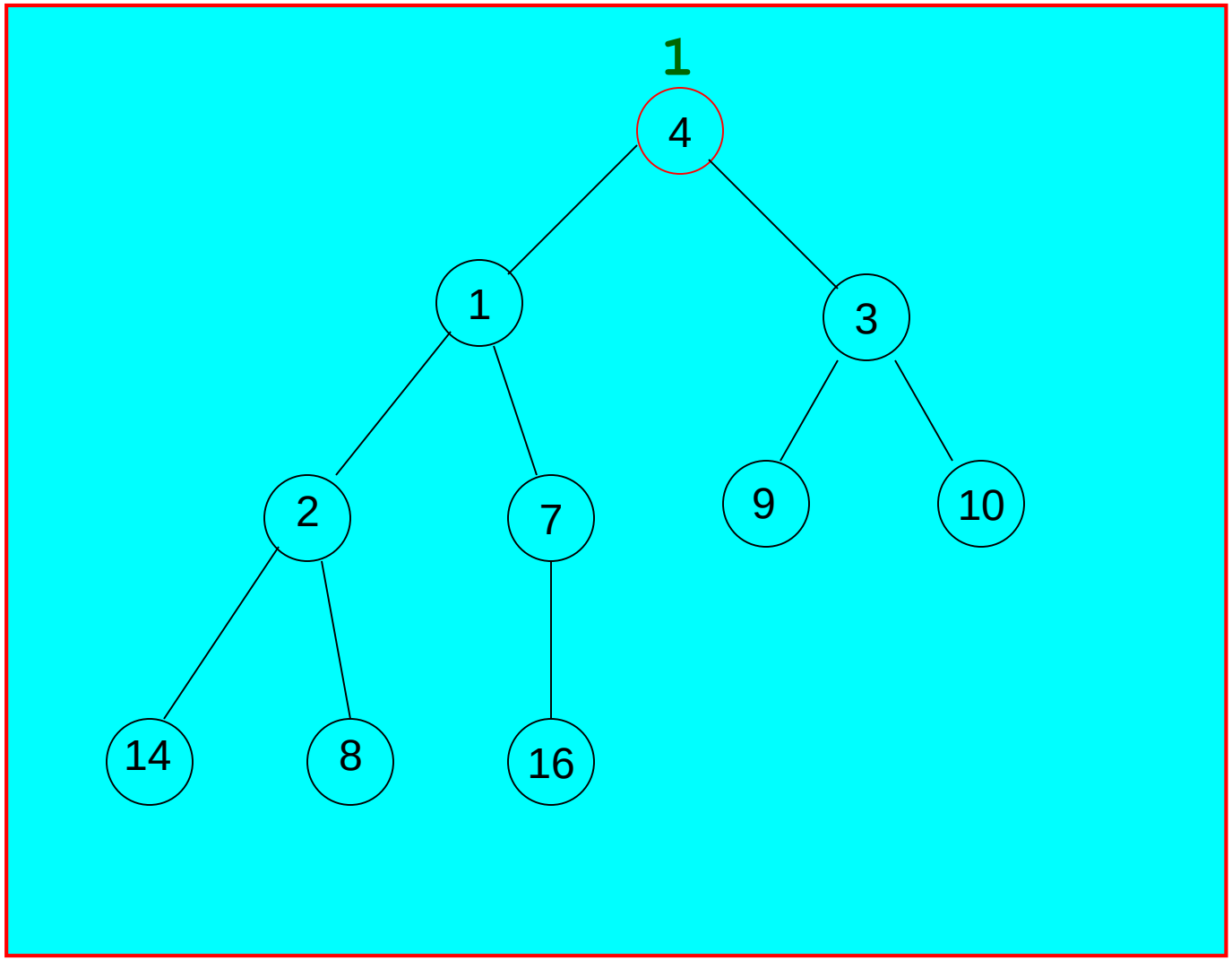
Min-Heapify(노드3)



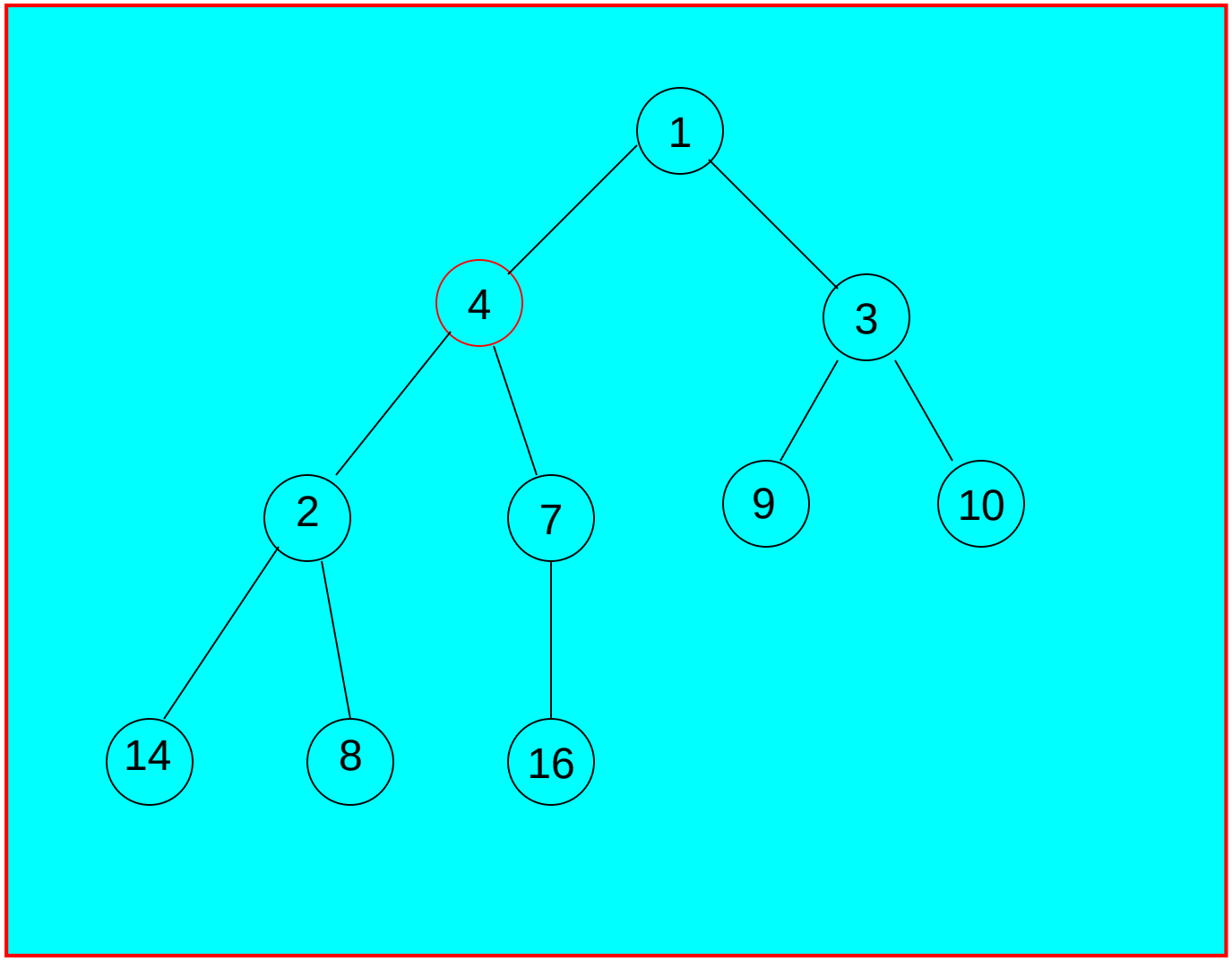
Min-Heapify(노드2)



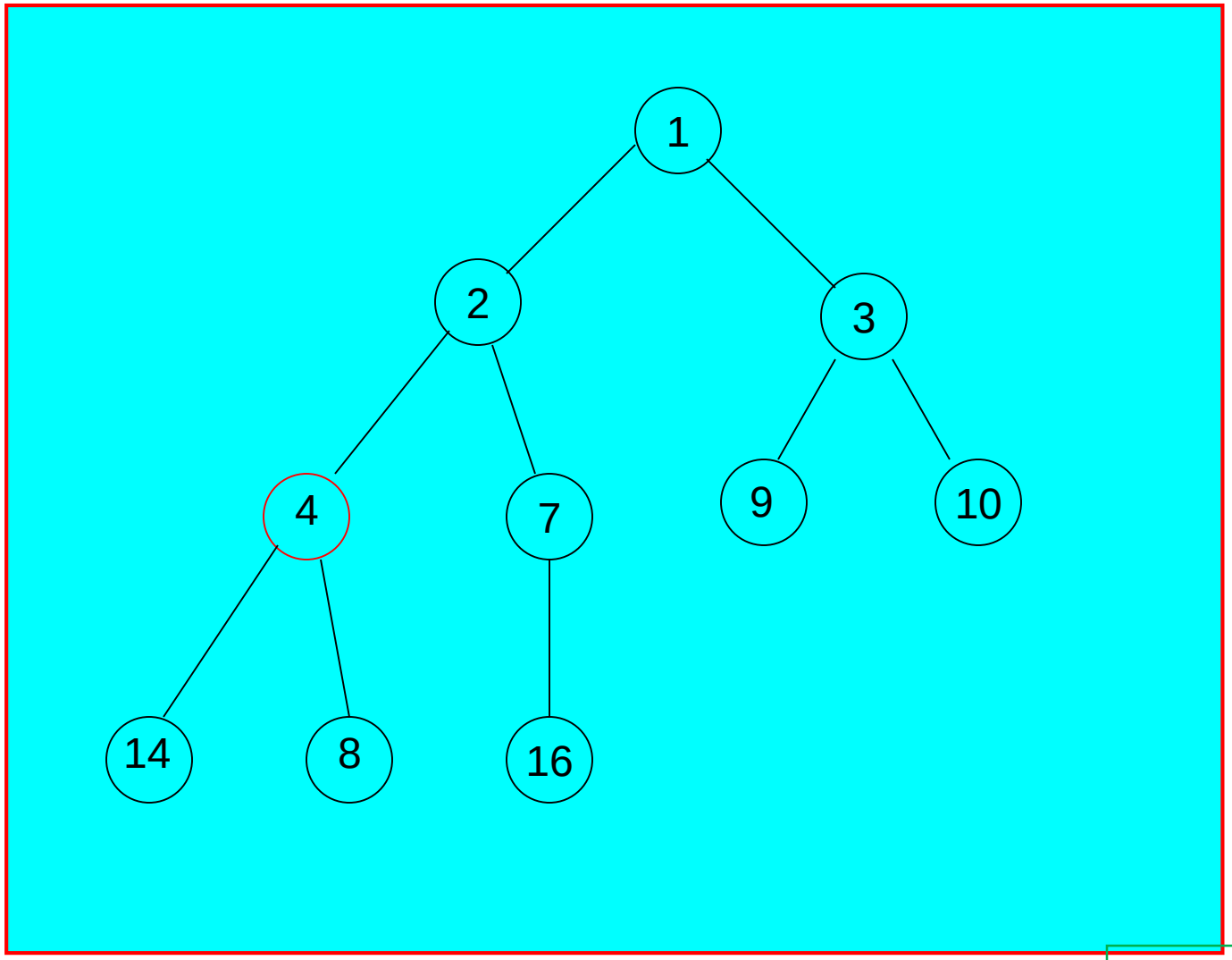
Min-Heapify(노드1)



Min-Heapify(노드1)



Min-Heapify(노드1)



$O(n)$ time

Priority Queue using Heap
히프를 사용한 우선순위

- 우선순위큐 프로그램
  - 사실상 히프에서 삽입, 삭제 작업의 다른 표현

```
class PriorityQueue{
    private int count;                          // 우선순위 큐의 현재 원소수
    private int size;                           // 배열의 크기
    private int increment;                      // 배열 확장 단위
    private PrioityElement[] itemArray;  // 우선순위 큐 원소를 저장하는 배열

    public PriorityQueue(){
        count = 0;   // itemArray[0]는 실제로 사용하지 않음
        size = 16;   //실제 최대 원소 수는 size - 1
        increment = 8;
        itemArray = new PrioityElement[size];
    }

    public int currentSize(){                   // 우선순위 큐의 현재 원소수
        return count;
    }
```

```
    public void insert(PrioityElement newKey){
                        // 우선순위 큐에 원소 삽입
        if (count == size-1) PQFull();
        count++;   // 삽입 공간을 확보하고 원소의 삽입 위치를 밑에서부터 찾아
올라감
        int childLoc = count;
        int parentLoc = childLoc/2;
        while (parentLoc != 0) {
            if (newKey.compareTo(itemArray[parentLoc]) <= 0) {
                            // 위치가 올바른 경우
                itemArray[childLoc] = newKey;              // 원소 삽입
                return;
            } else {   // 한 레벨 위의 위치로 이동
                itemArray[childLoc] = itemArray[parentLoc];
                childLoc = parentLoc;
                parentLoc = childLoc/2 ;
            }
        }
        itemArray[childLoc] = newKey;        // 최종 위치에 원소 삽입
    } // end insert()
```

```
public void PQFull() {
            // itemArray가 만원이면
   size += increment;     // increment만큼 더 크게 확장
   PriorityElement[] tempArray = new PriorityElement[size];
   for (int i = 1; i < count; i++) {
         tempArray[i] = itemArray[i];
   }
   itemArray= tempArray;
} // end PQFull()

public PriorityElement delete() {
             // 우선순위 큐로부터 원소 삭제
   if (count == 0) {              // 우선순위 큐가 공백인 경우
       return null;
   }
   else {
          // 변수 선언
       int currentLoc;
       int childLoc;
       PriorityElement itemToMove;   // 이동시킬 원소
       PriorityElement deletedItem;    // 삭제한 원소
      deletedItem = itemArray[1];    // 삭제하여 반환할 원소
```

```
            itemToMove = itemArray[count--];   // 이동시킬 원소
            currentLoc = 1;
            childLoc = 2*currentLoc;
            while (childLoc <= count) {            // 이동시킬 원소의 탐색
                if (childLoc < count) {
                    if (itemArray[childLoc+1].compareTo(itemArray[childLoc]) > 0)
                        childLoc ++;
                }
                if (itemArray[childLoc].compareTo(itemToMove) > 0) {
                    itemArray[currentLoc]=itemArray[childLoc]; // 원소를 한 레벨
                    currentLoc = childLoc;                  // 위로 이동
                    childLoc = 2*currentLoc;
                } else {
                    itemArray[currentLoc]=itemToMove;  // 이동시킬 원소 저장
                    return deletedItem;
                }
            } // end while
            itemArray[currentLoc] = itemToMove;     // 최종 위치에 원소 저장
            return deletedItem;
        } // end if
    } // end delete()
} // end PriorityQueue class
```

Selection Trees(선택 트리)
완전이진트리 응용의예

Selection Trees선택트리 (1/6)

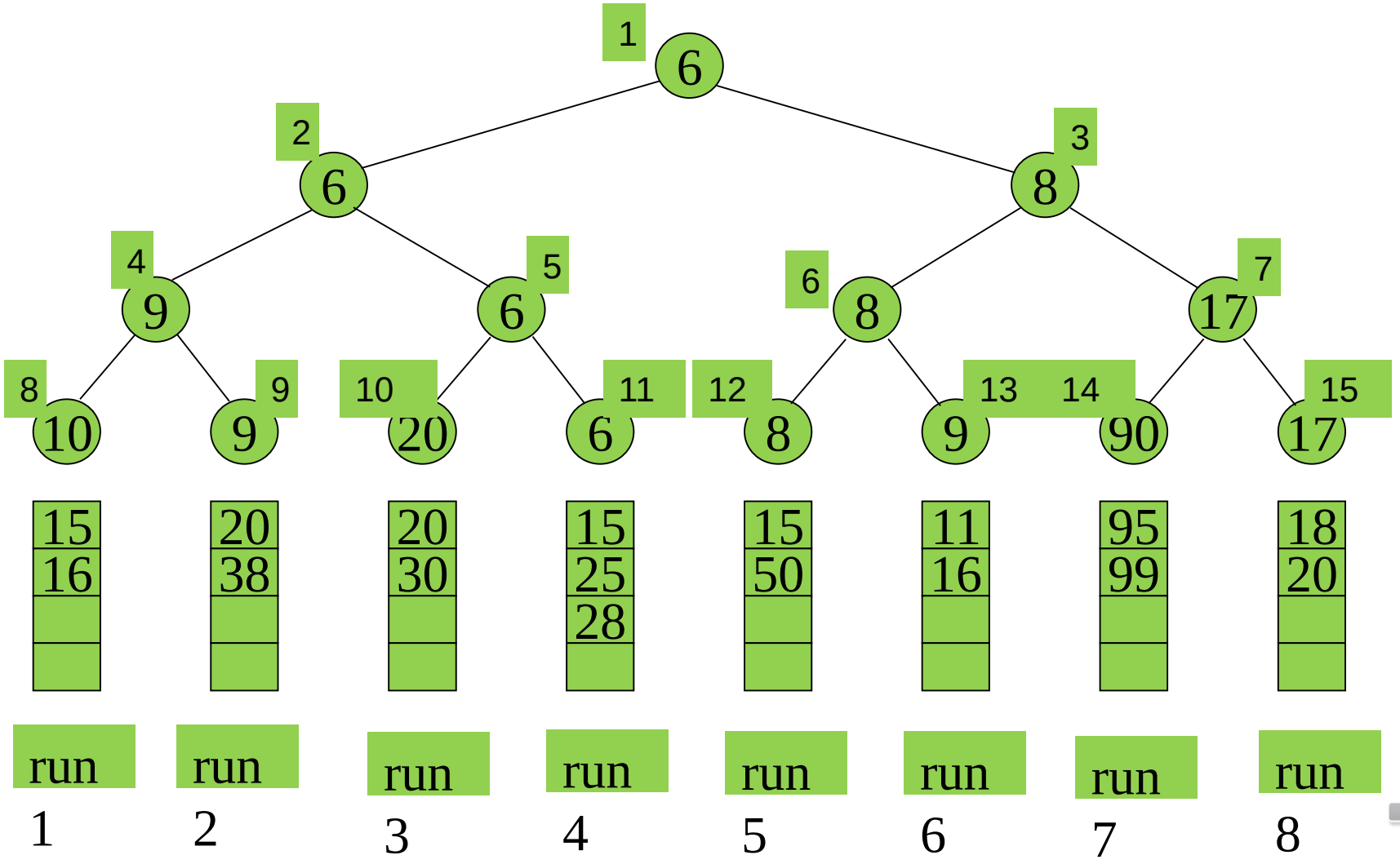
- Problem:
  - suppose we have *k* order sequences, called runs, that are to be merged into a single ordered sequence(런은 정렬된 순서)
- Solution: (k-1회비교할 상황에서 $\lceil \log_2 k \rceil$+1회비교. 좋음)
  - straightforward : *k*-1 comparison
  - selection tree : $\lceil \log_2 k \rceil$+1
- There are two kinds of selection trees: 두 종류의 선택트리.
  winner trees and *loser trees* *승자트리, 패자트리*

Winner Tree

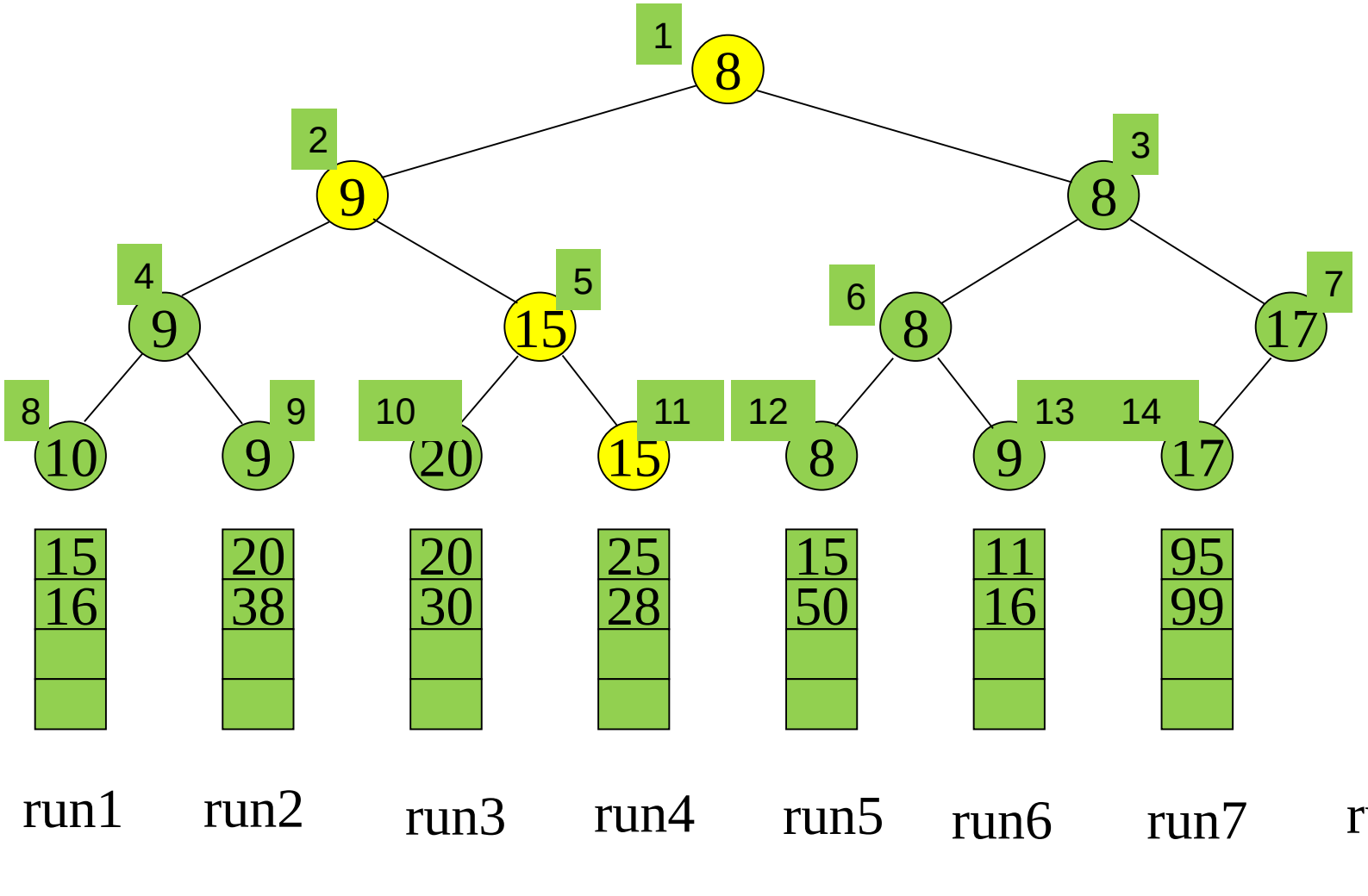
- A winner tree is a complete binary tree(완전이진트리) in which each node represents the smaller of its two children. Thus the root represents the smallest node in the tree.
- Each leaf node represents the first record in the corresponding run.
- Each non-leaf node in the tree represents the winner of its right and left subtrees.

Winner Tree For k = 8 예

완전이진트리

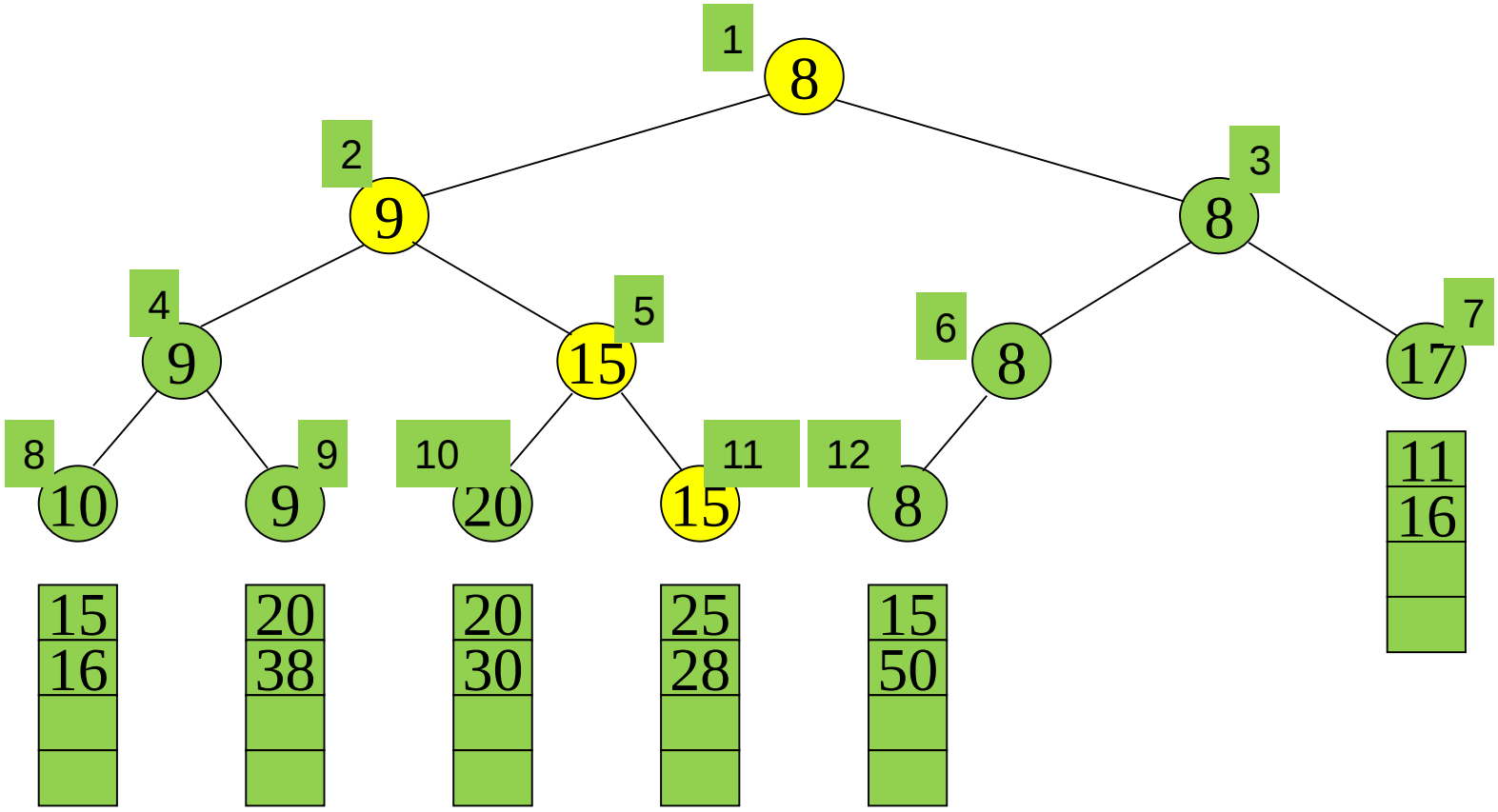


Winner Tree For k = 7
완전이진트리



Winner Tree For k = 6

완전이진트리 (run이 6개일때)



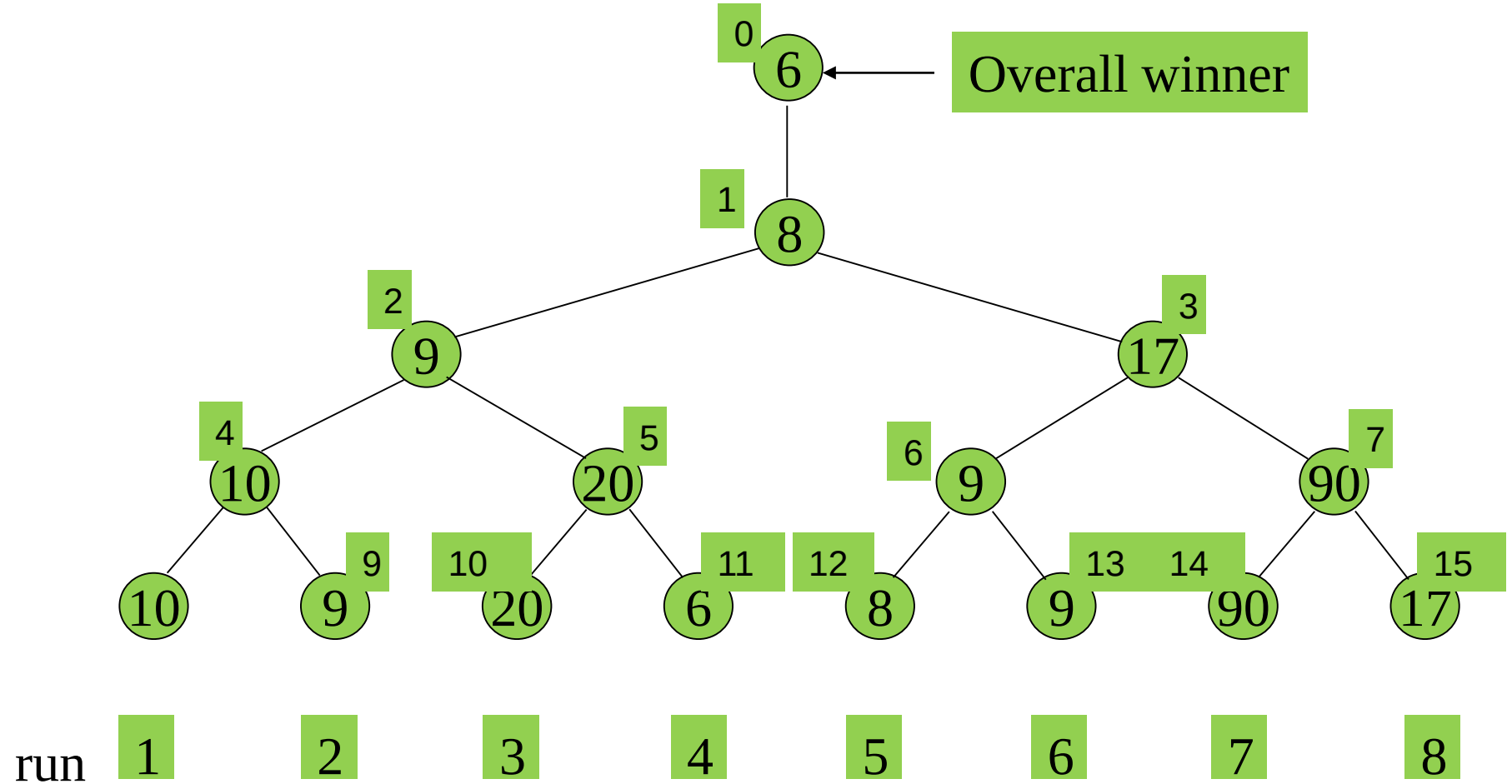
Analysis of Winner Tree

- The number of levels in the tree is $\lceil \log_2(k+1) \rceil$
  - The time to restructure the winner tree is O($log_2 k$).
- Since the tree has to be restructured each time a number is output, the time to merge all n records is O($n\ log_2 k$).
- The time required to setup the selection tree for the first time is O(k).
- Total time needed to merge the k runs is O($n\ log_2 k$). K개의 런에 있는 n개의 원소들을 다 합병하는데 O($n\ log_2 k$).

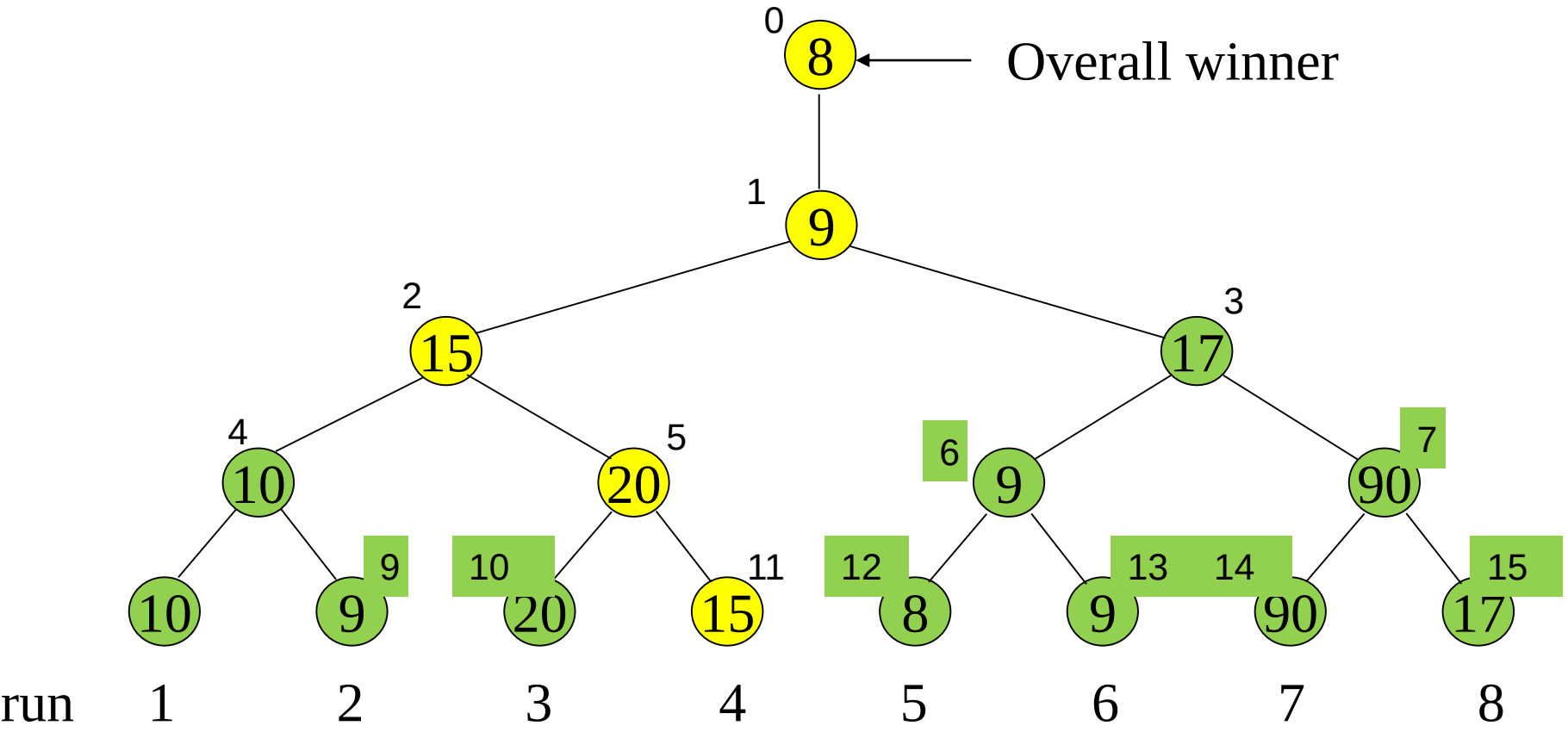
Loser Tree패자트리

- A selection tree in which each nonleaf node retains a pointer to the loser is called a loser tree.진사람은 머물고, 이긴사람은 연필로 옆에적어놓고 있음
- Again, each leaf node represents the first record of each run. 단말노드는 각 런의 선두원소
- An additional node, node 0, has been added to represent the overall winner of the tournament.

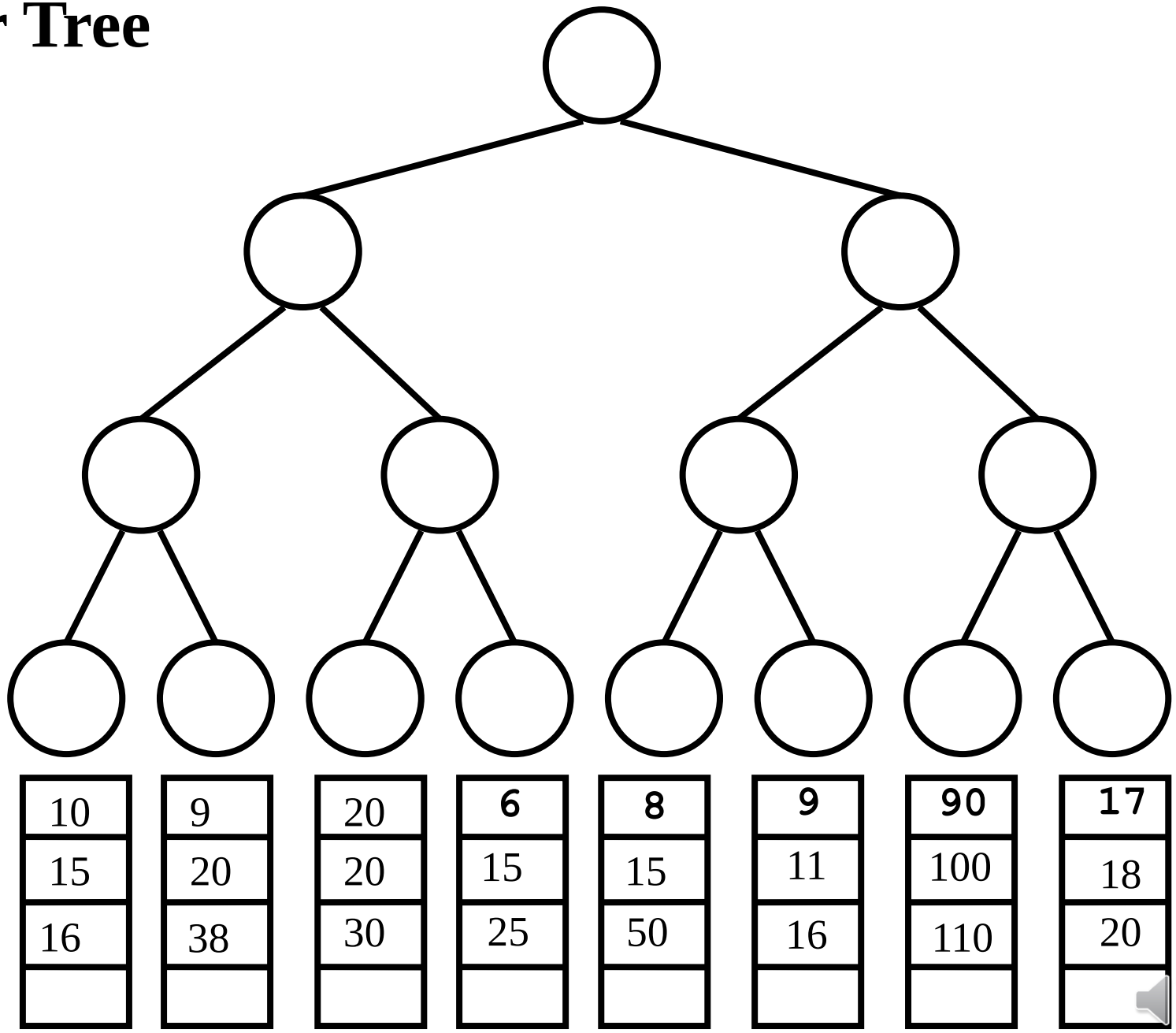
Loser Tree 역시 완전이진트리



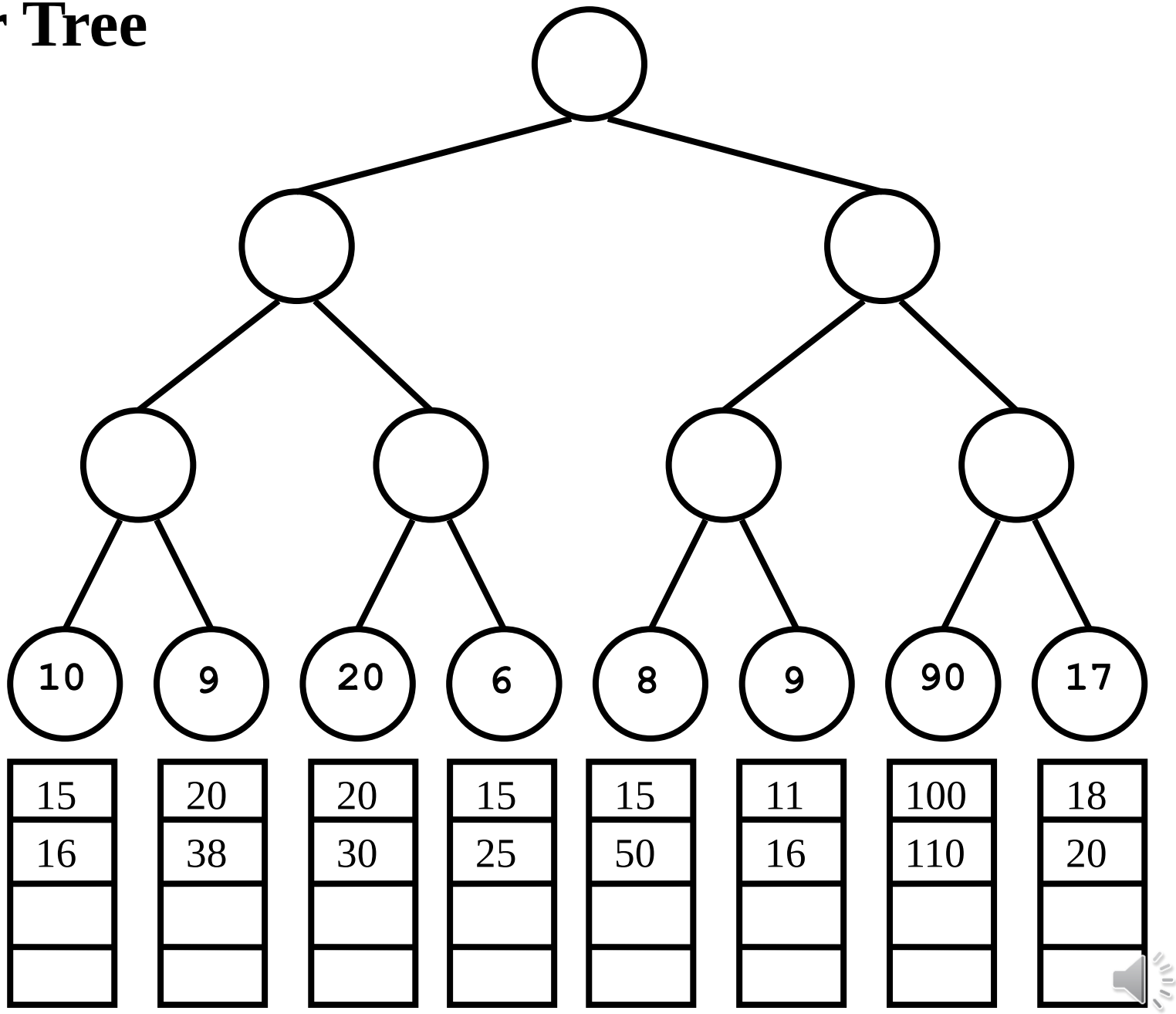
Loser Tree패자트리



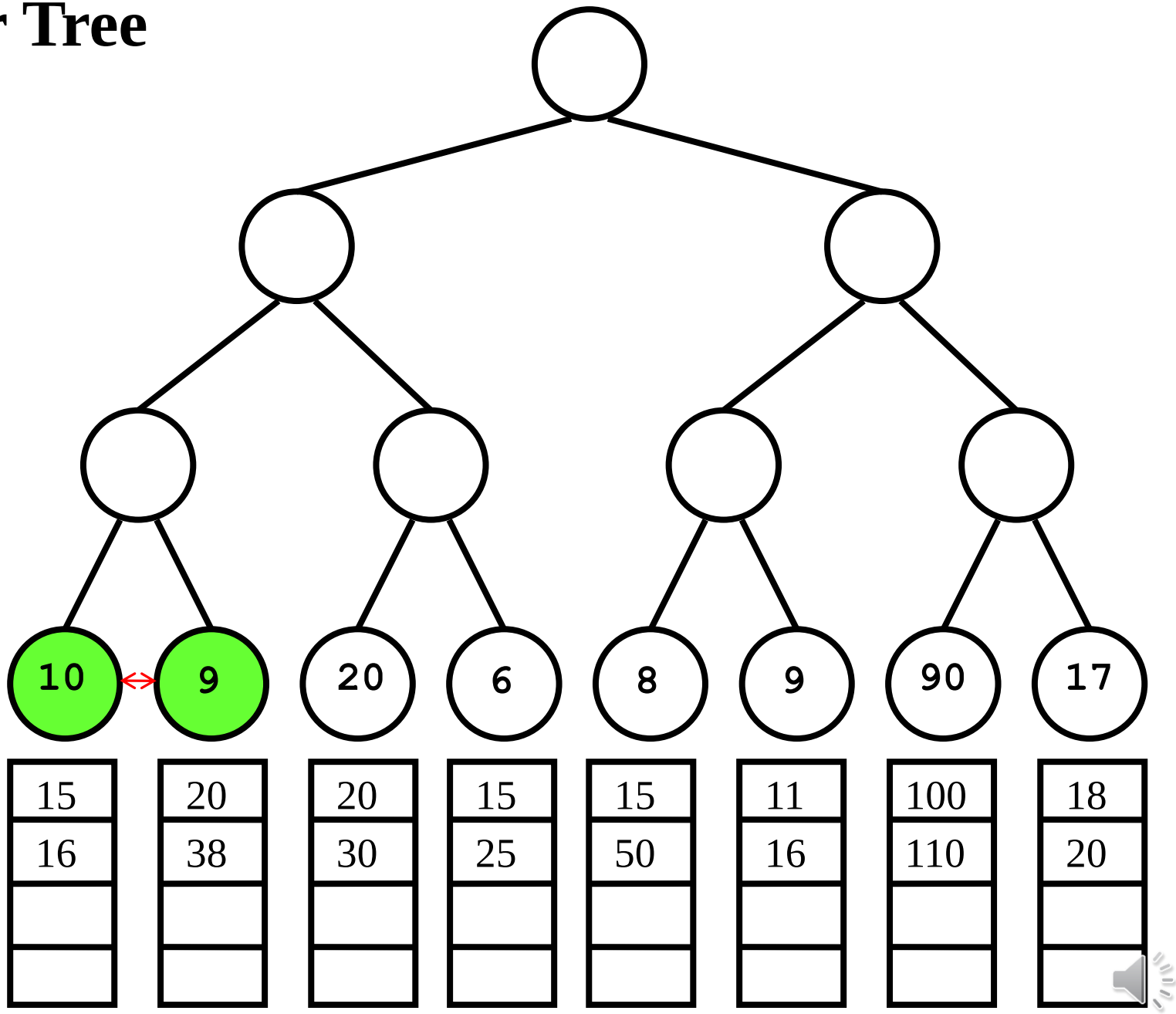
Winner Tree



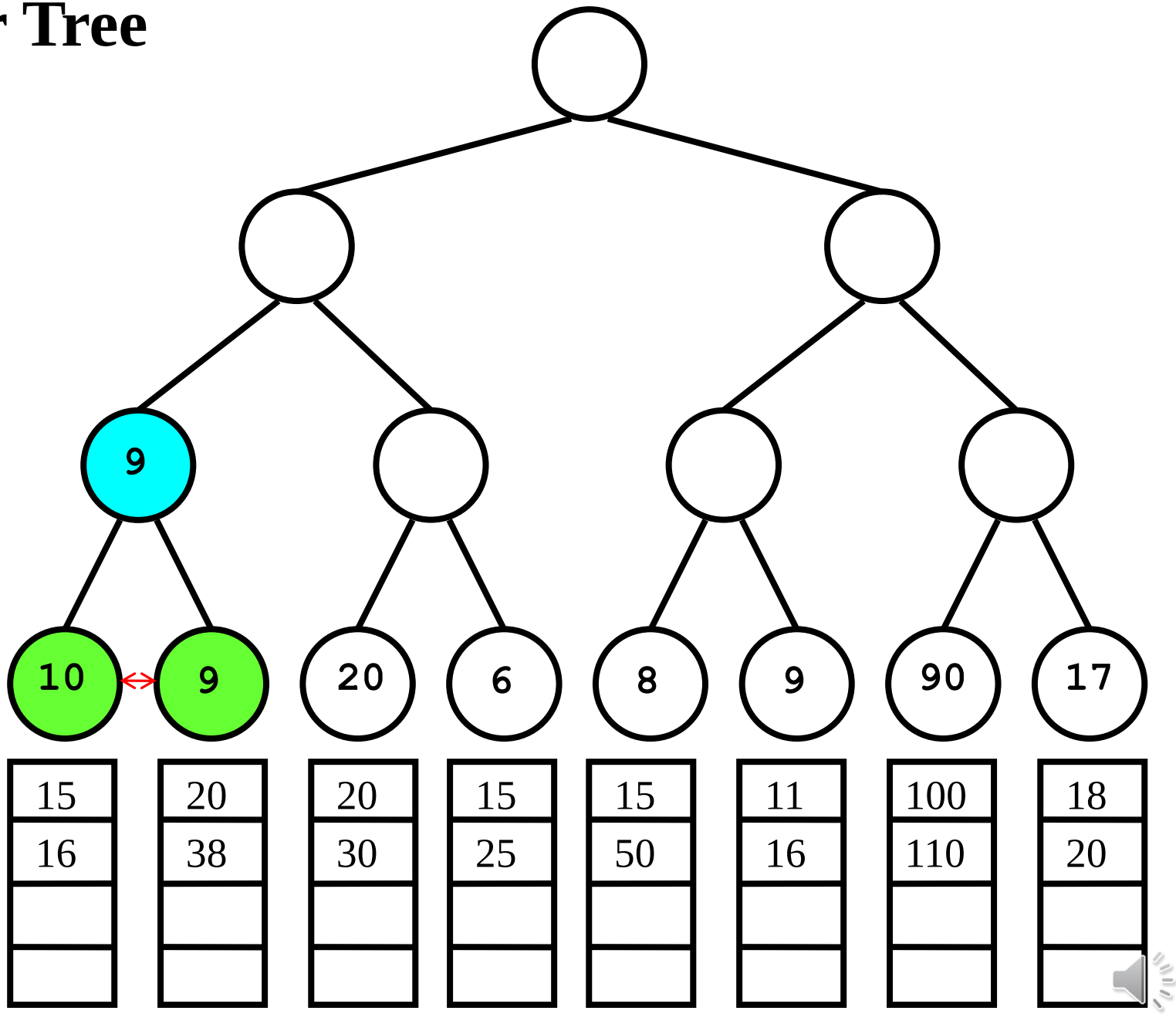
Winner Tree



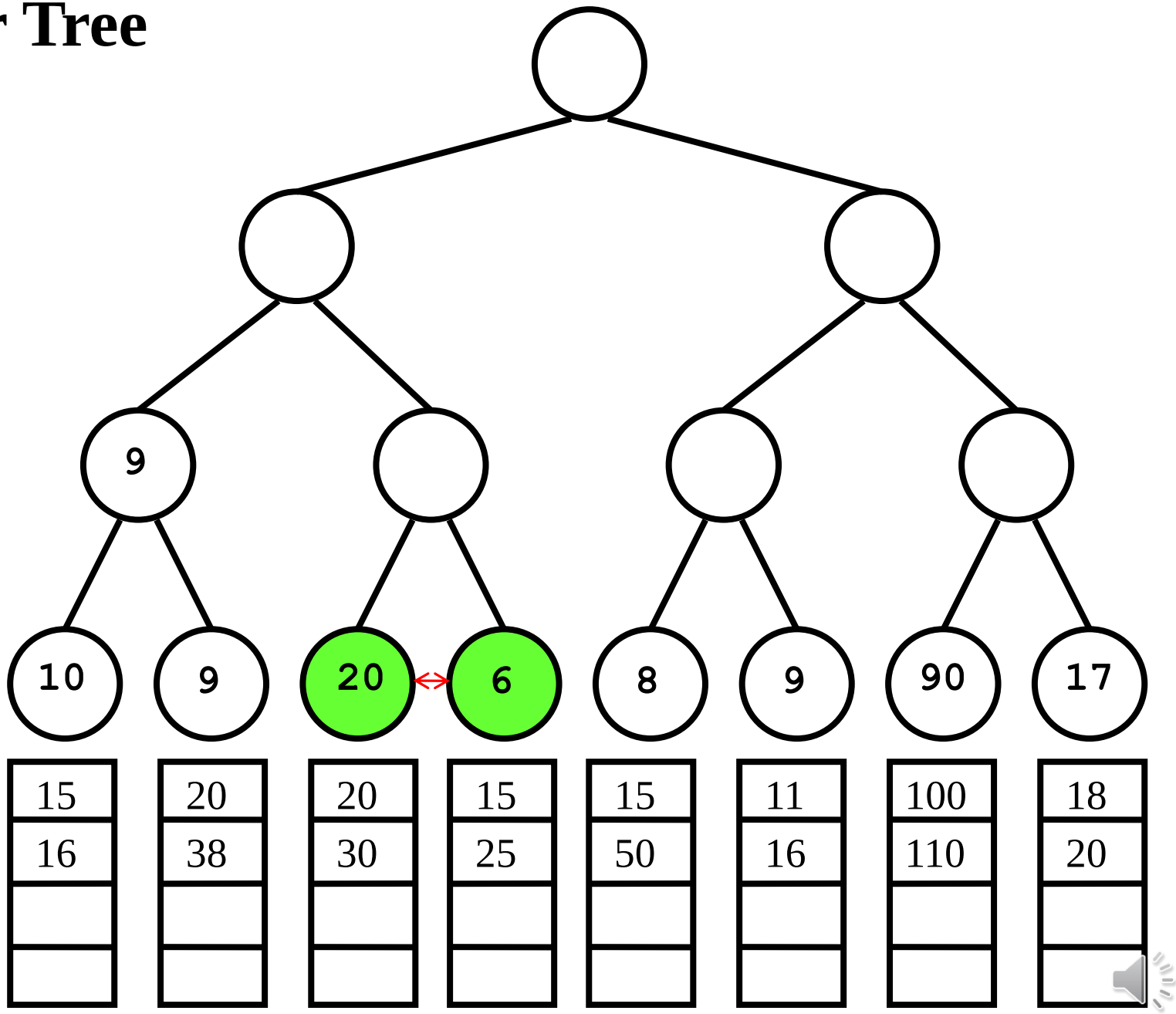
Winner Tree



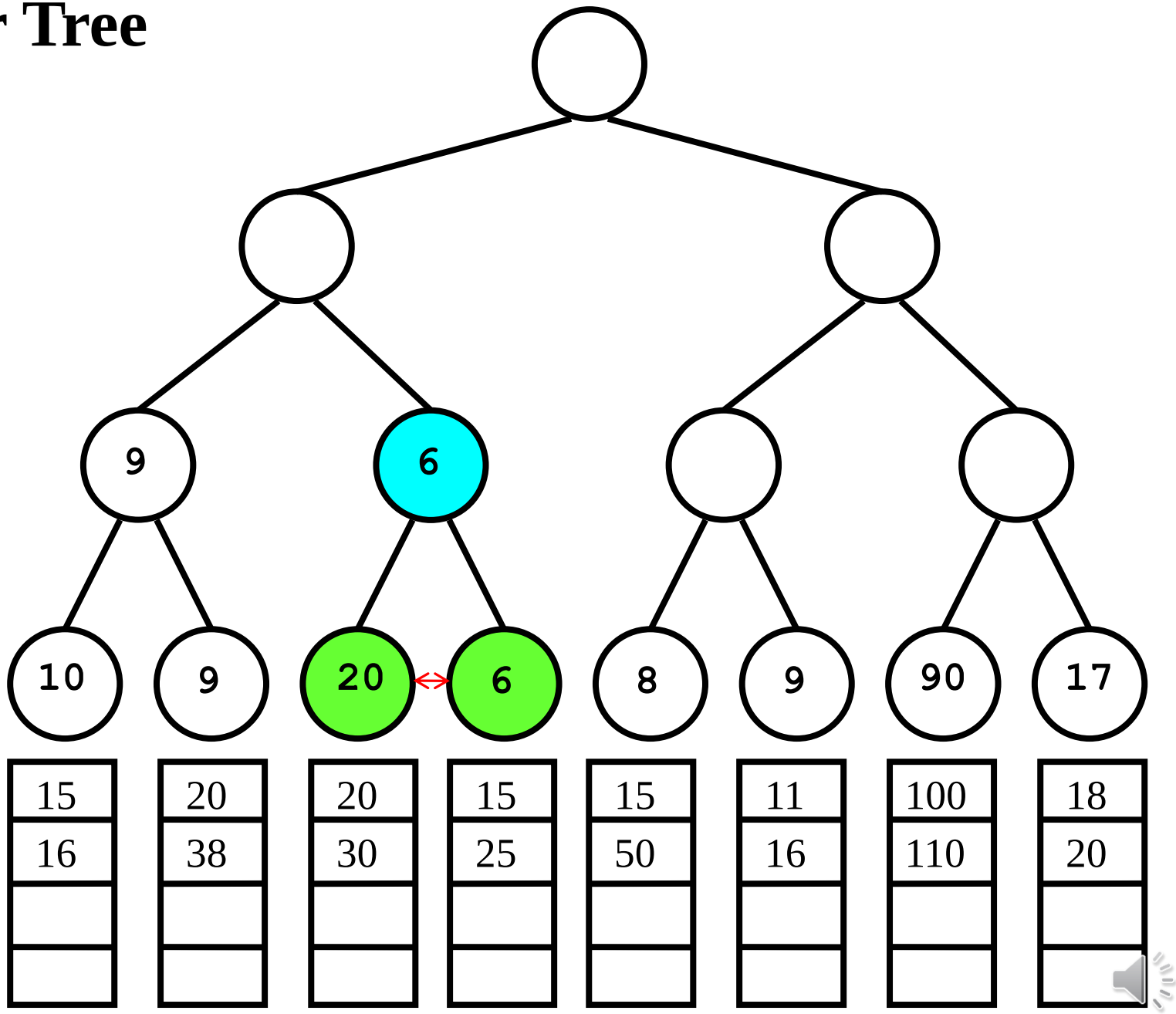
Winner Tree



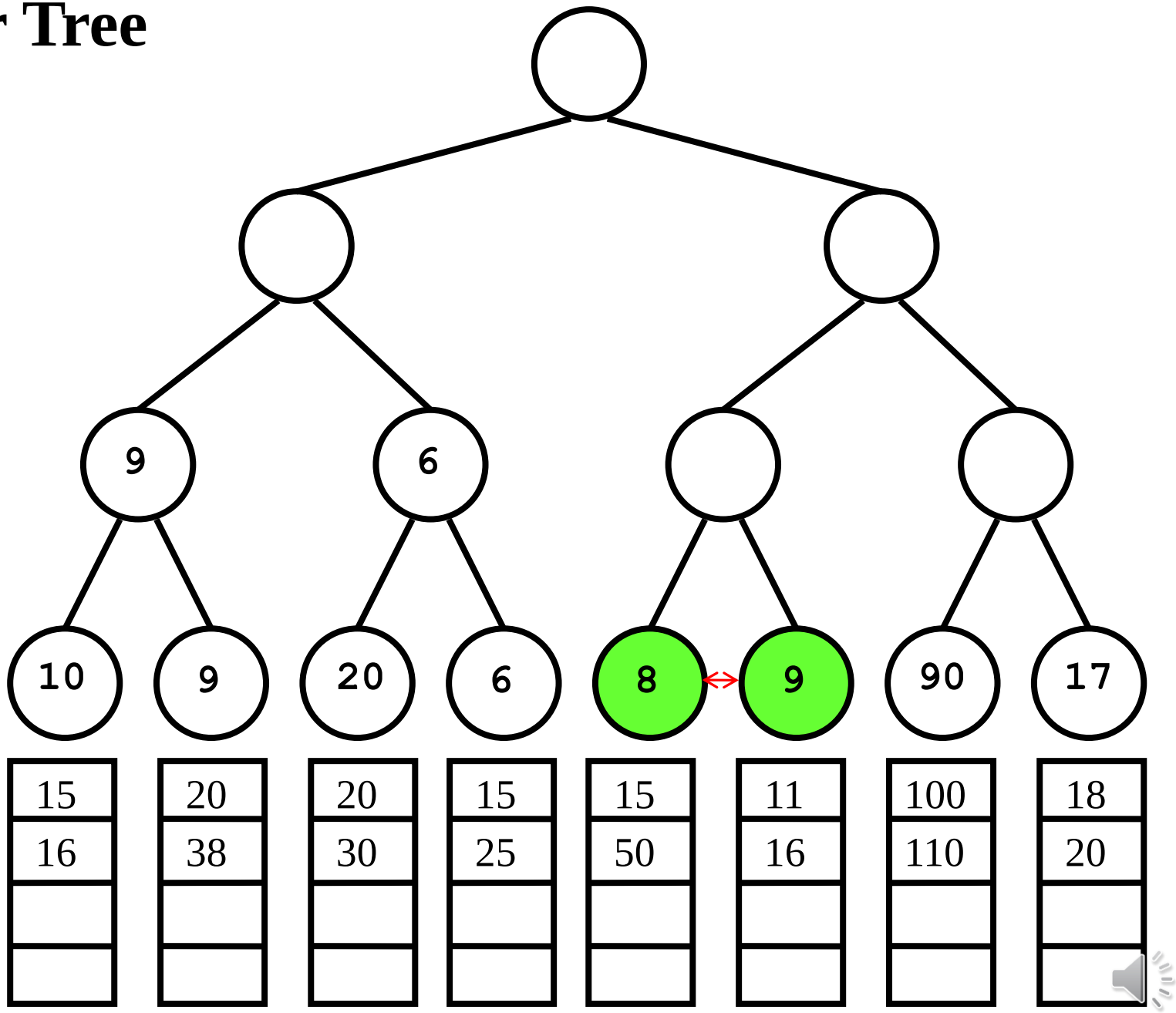
Winner Tree



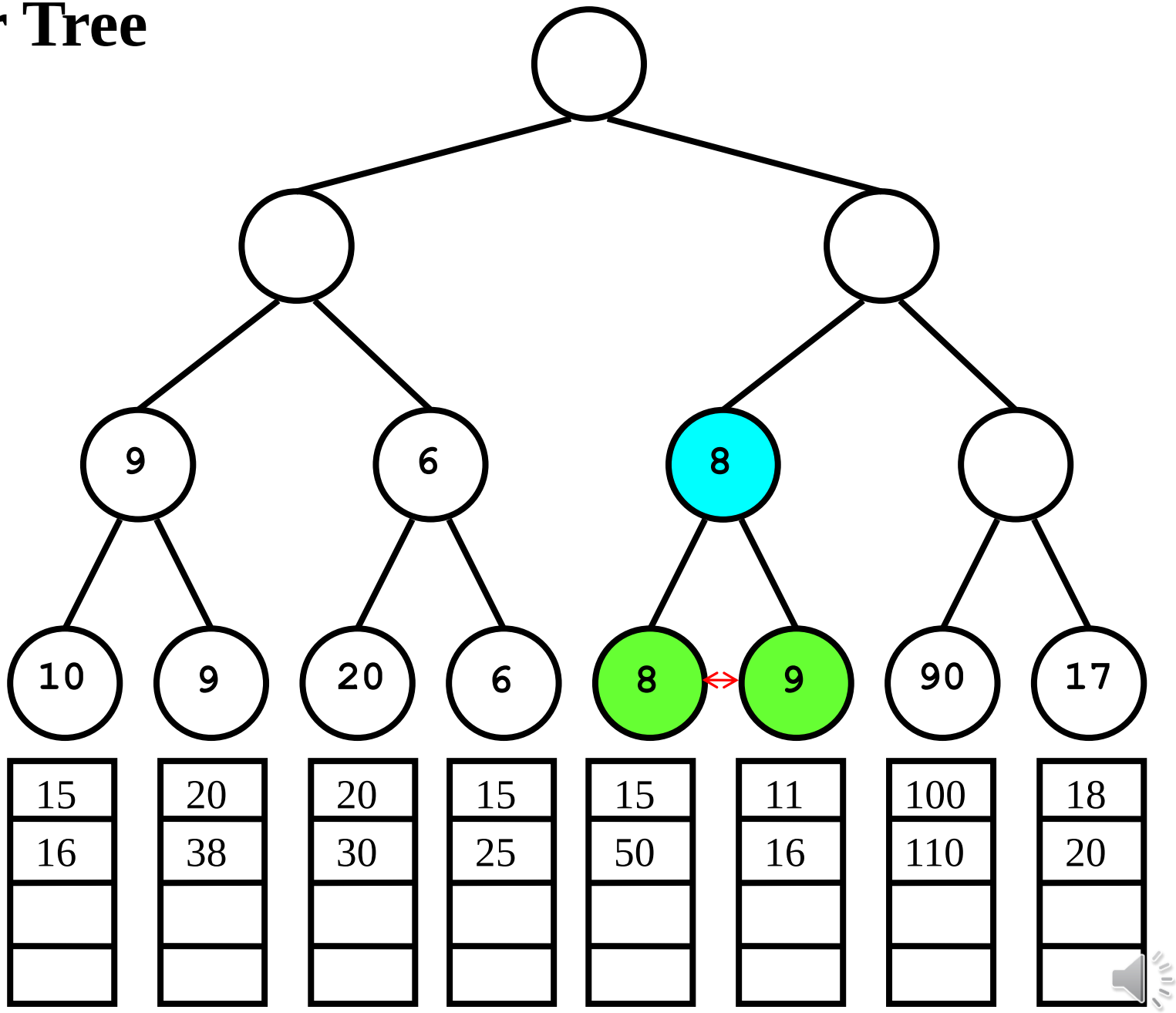
Winner Tree



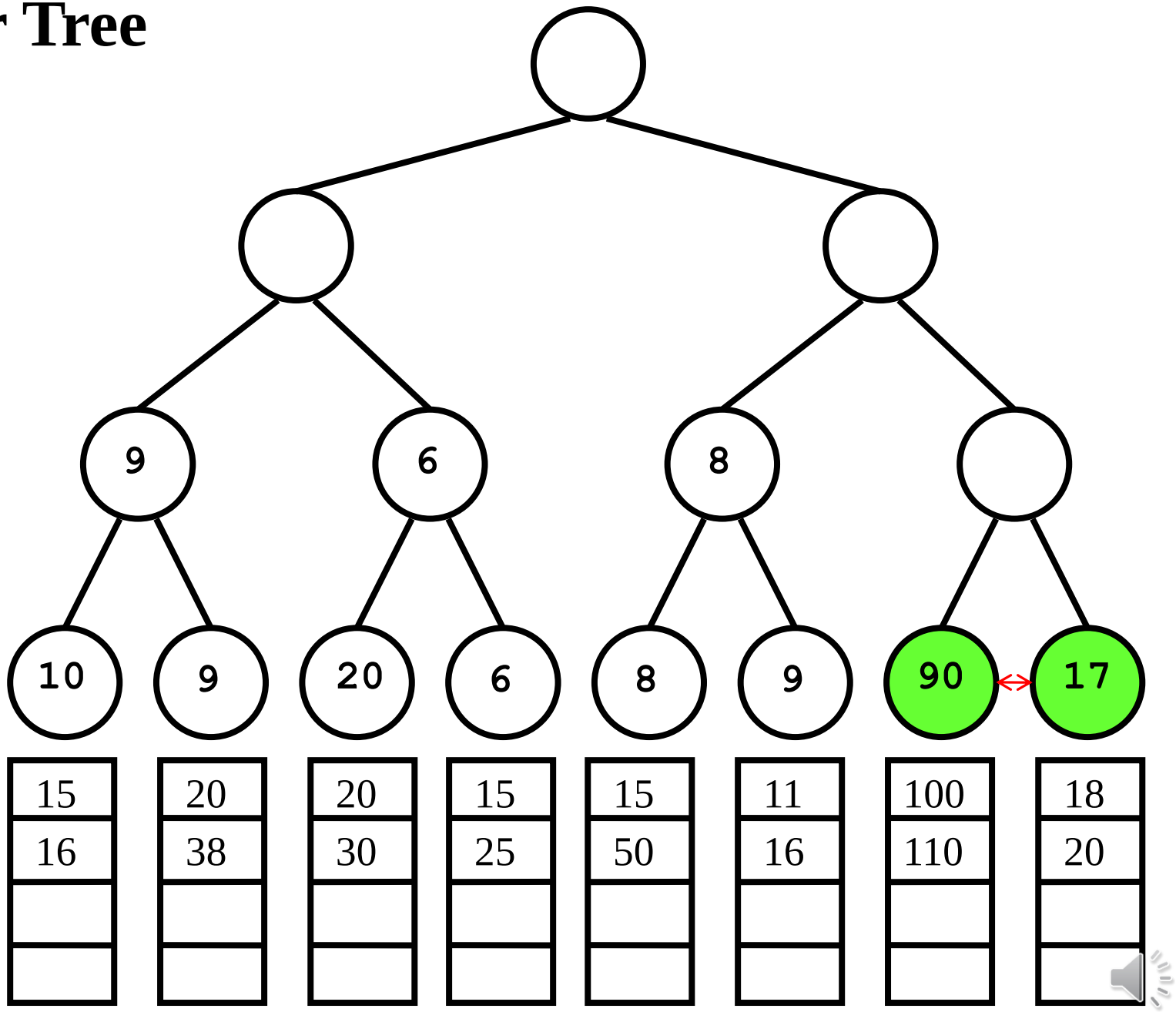
Winner Tree



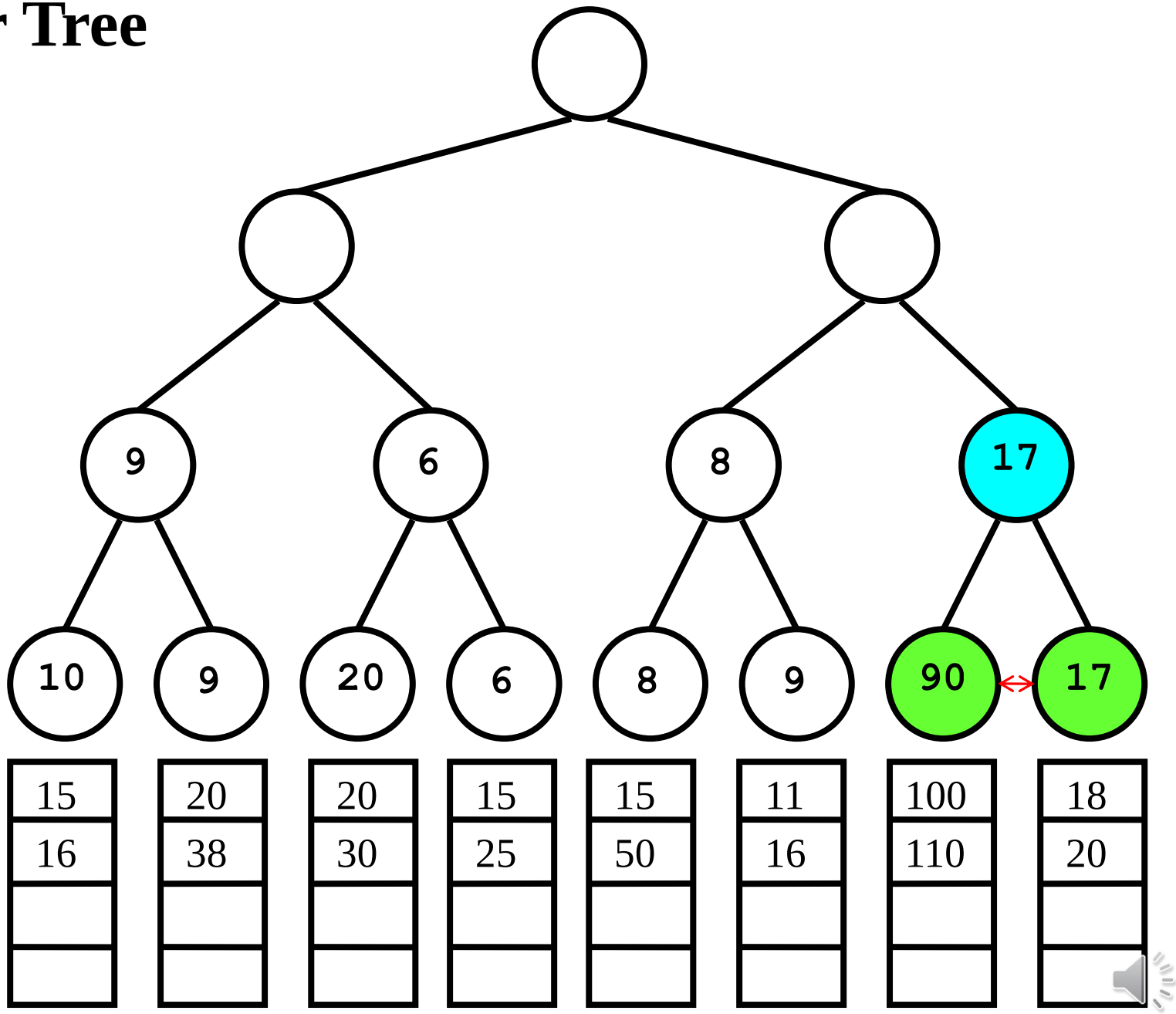
Winner Tree



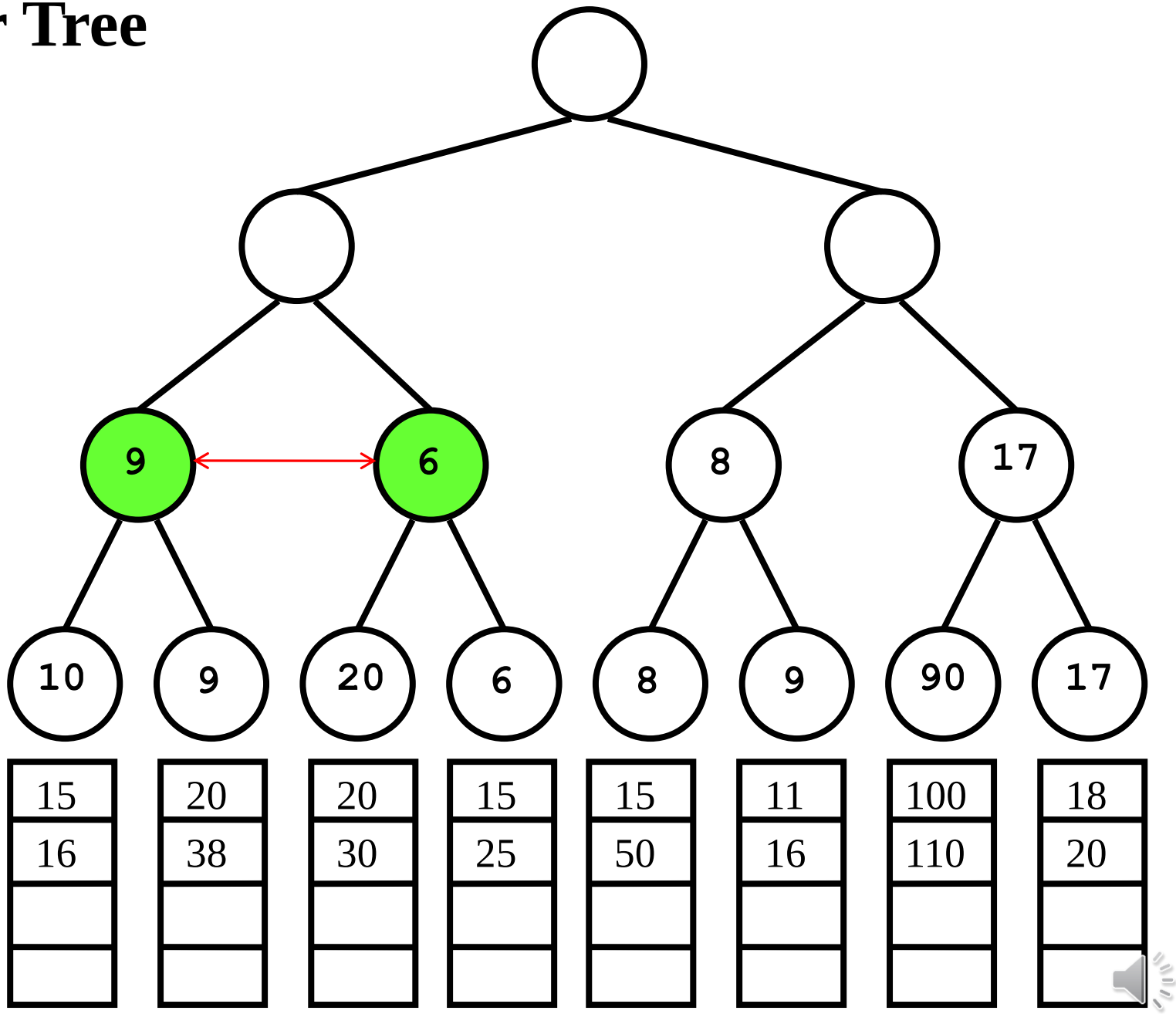
Winner Tree



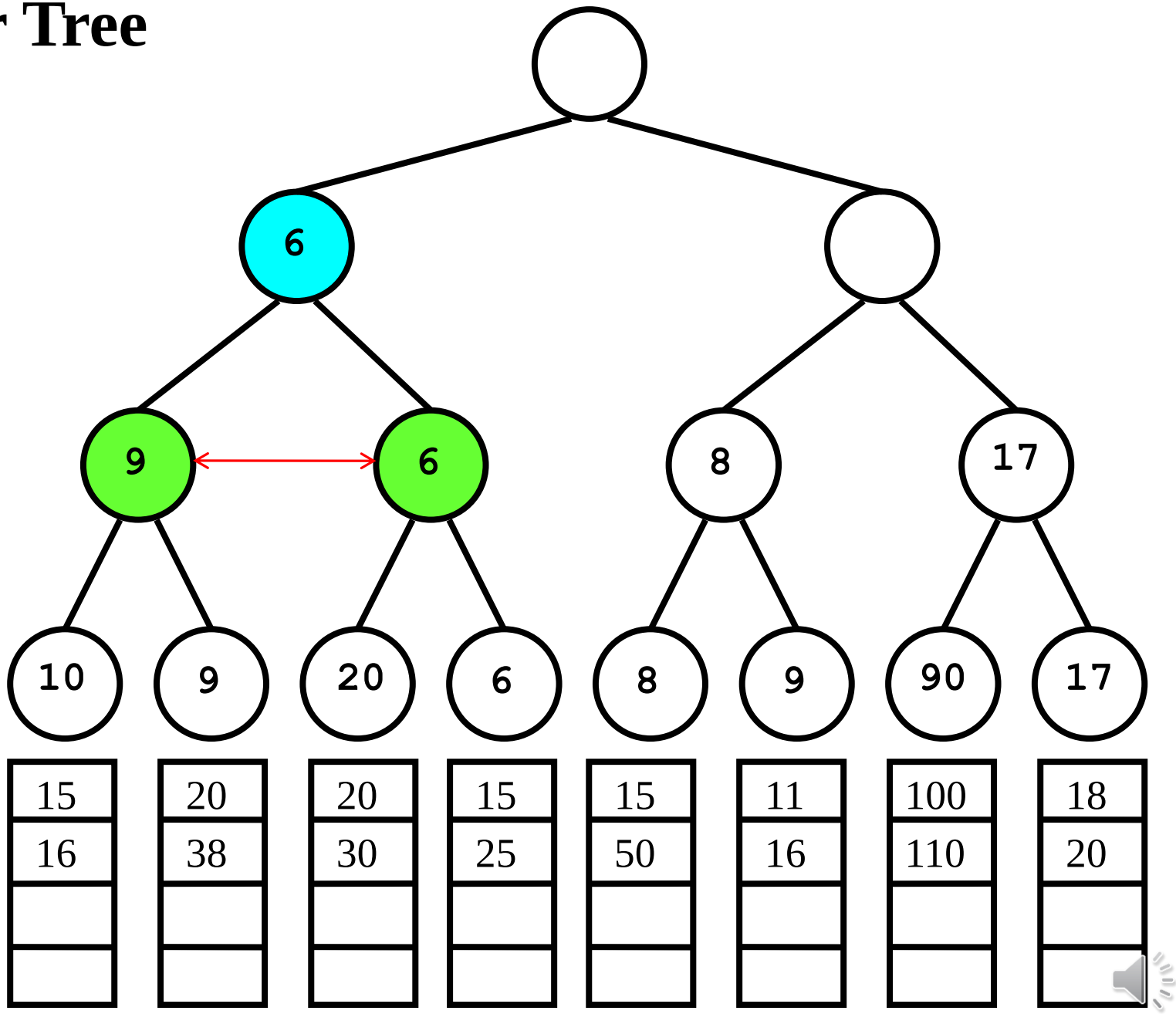
Winner Tree


9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

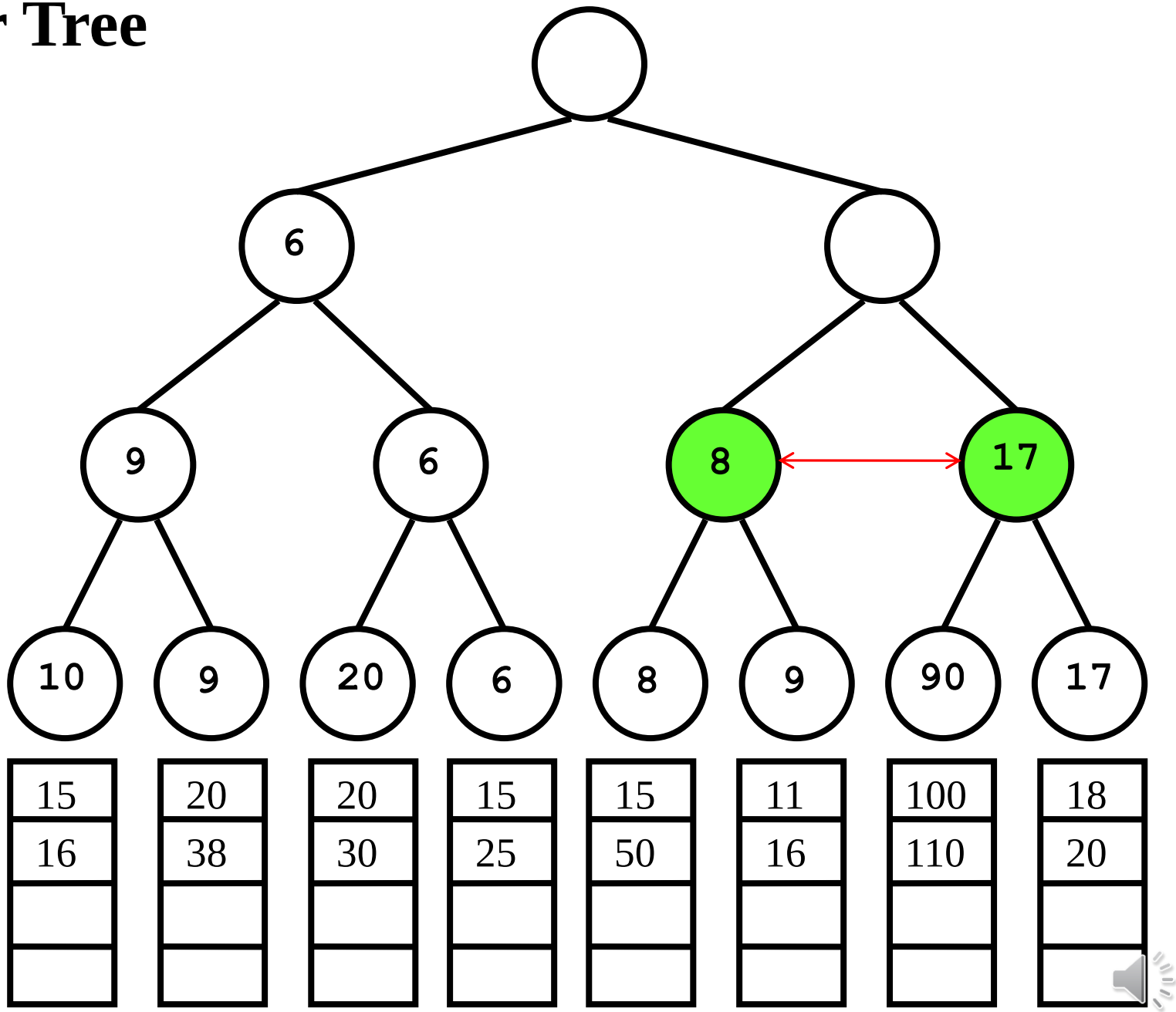
Winner Tree


9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

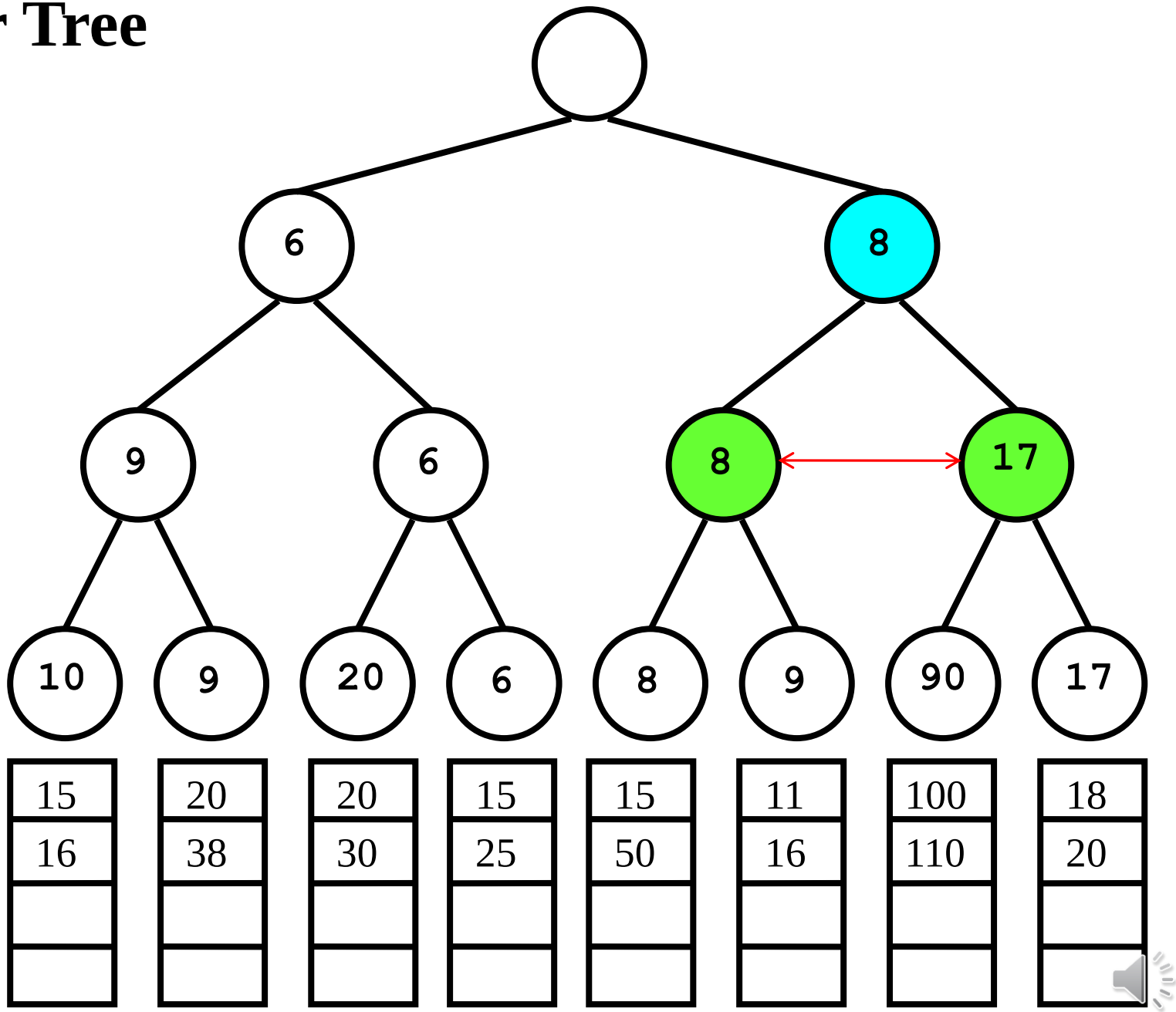
Winner Tree


6
9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

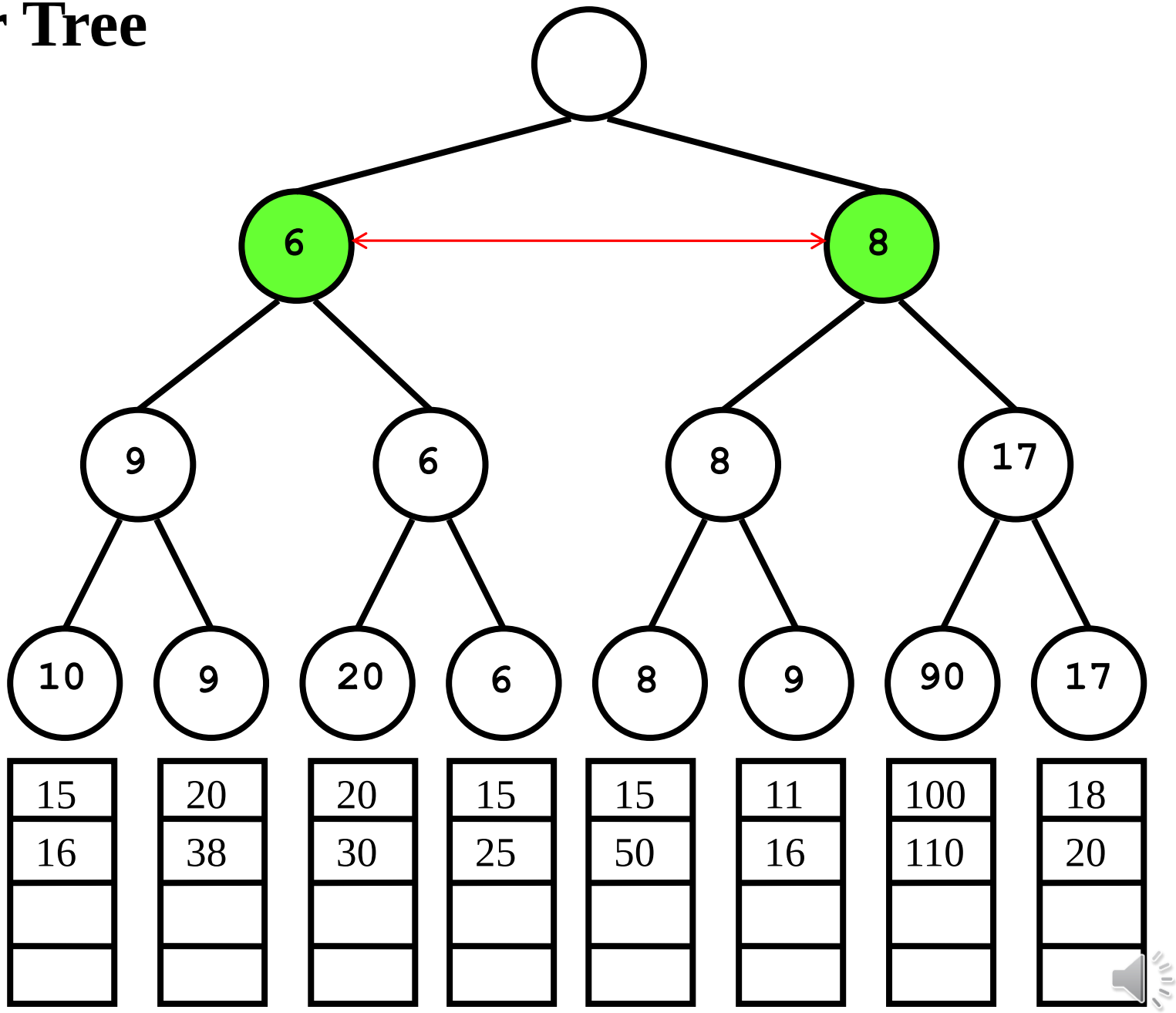
Winner Tree


6
9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

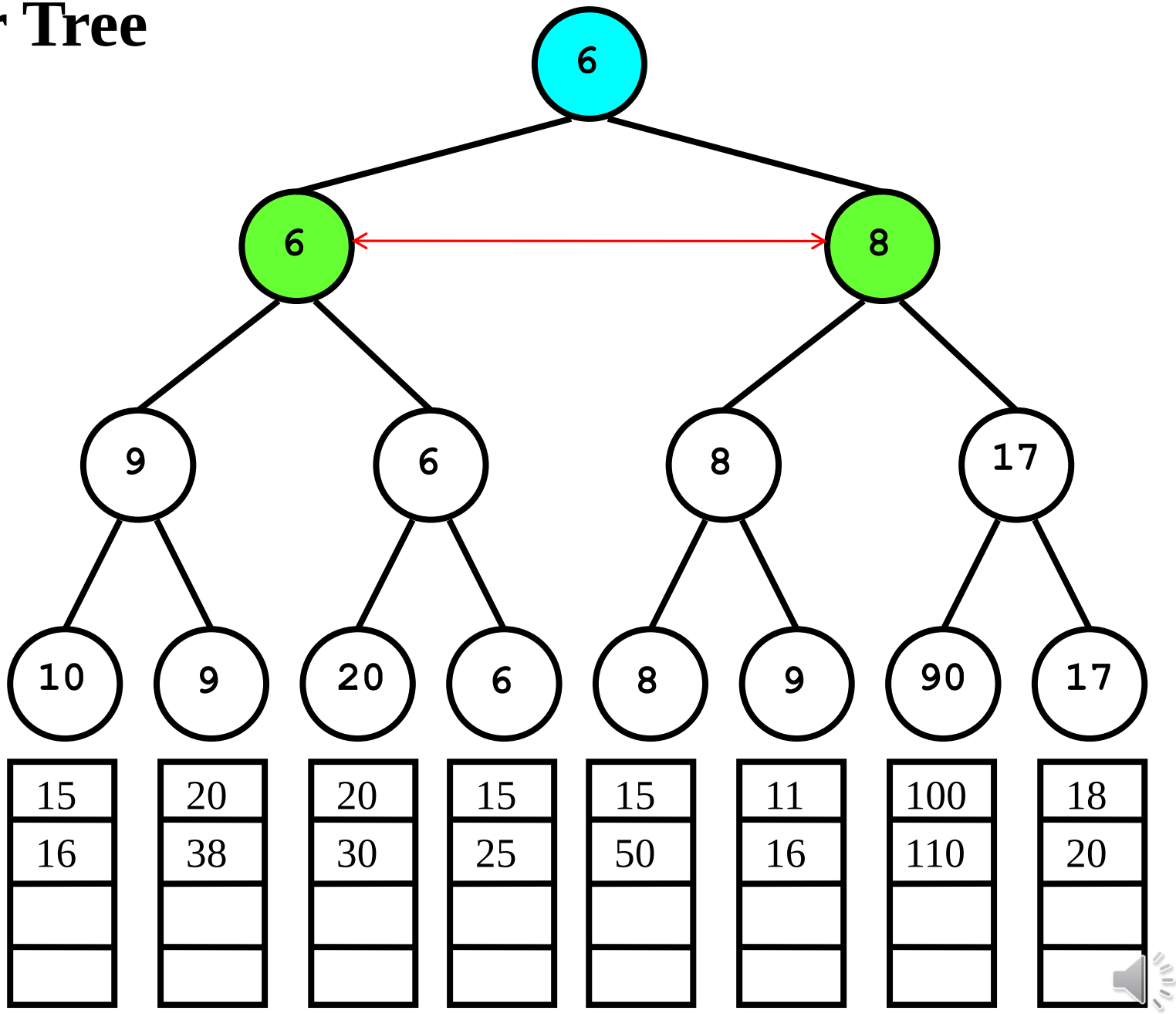
Winner Tree


6
8
9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

Winner Tree

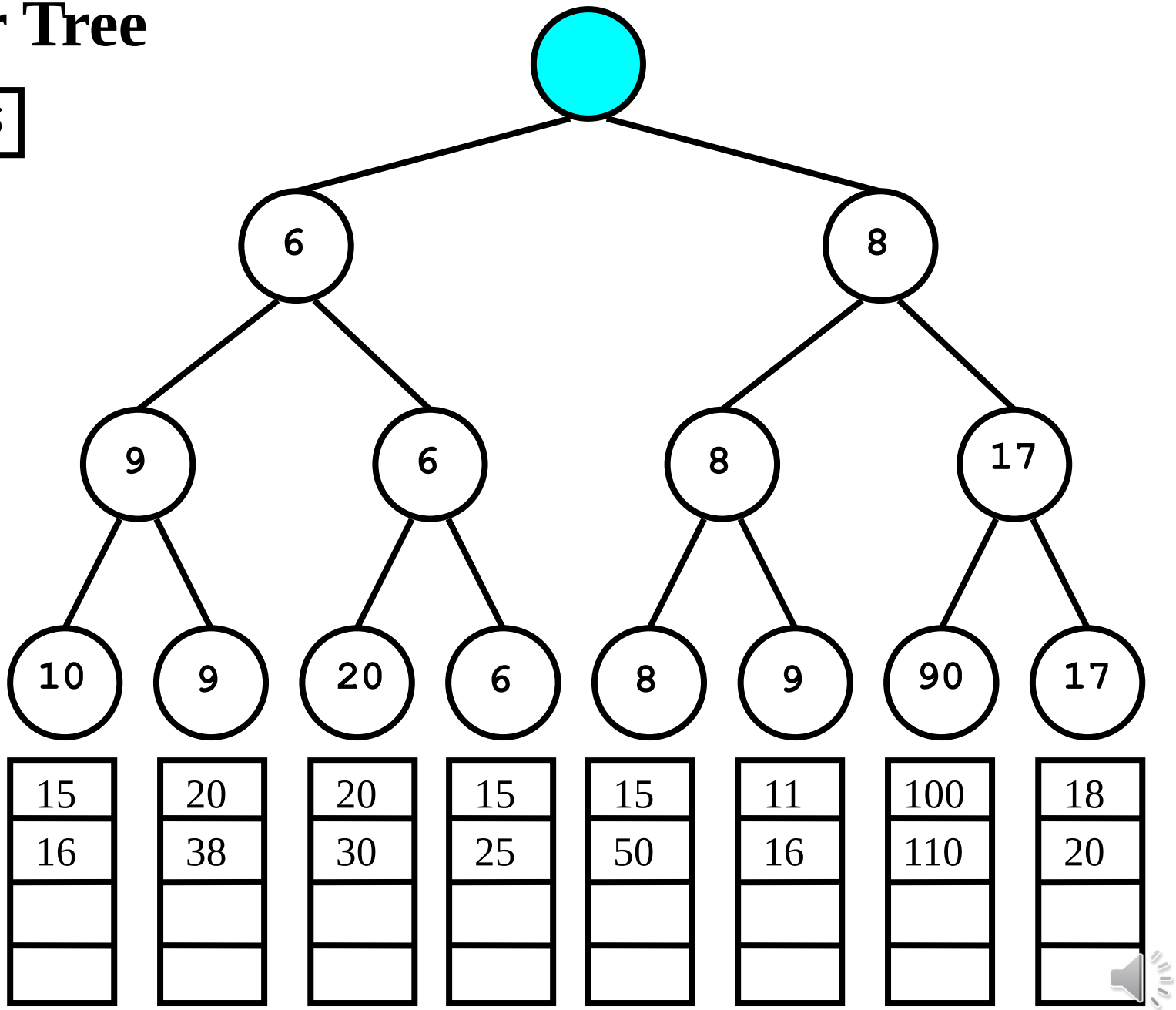

6
8
9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

Winner Tree


6
6
8
9
6
8
17
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
100
110
18
20

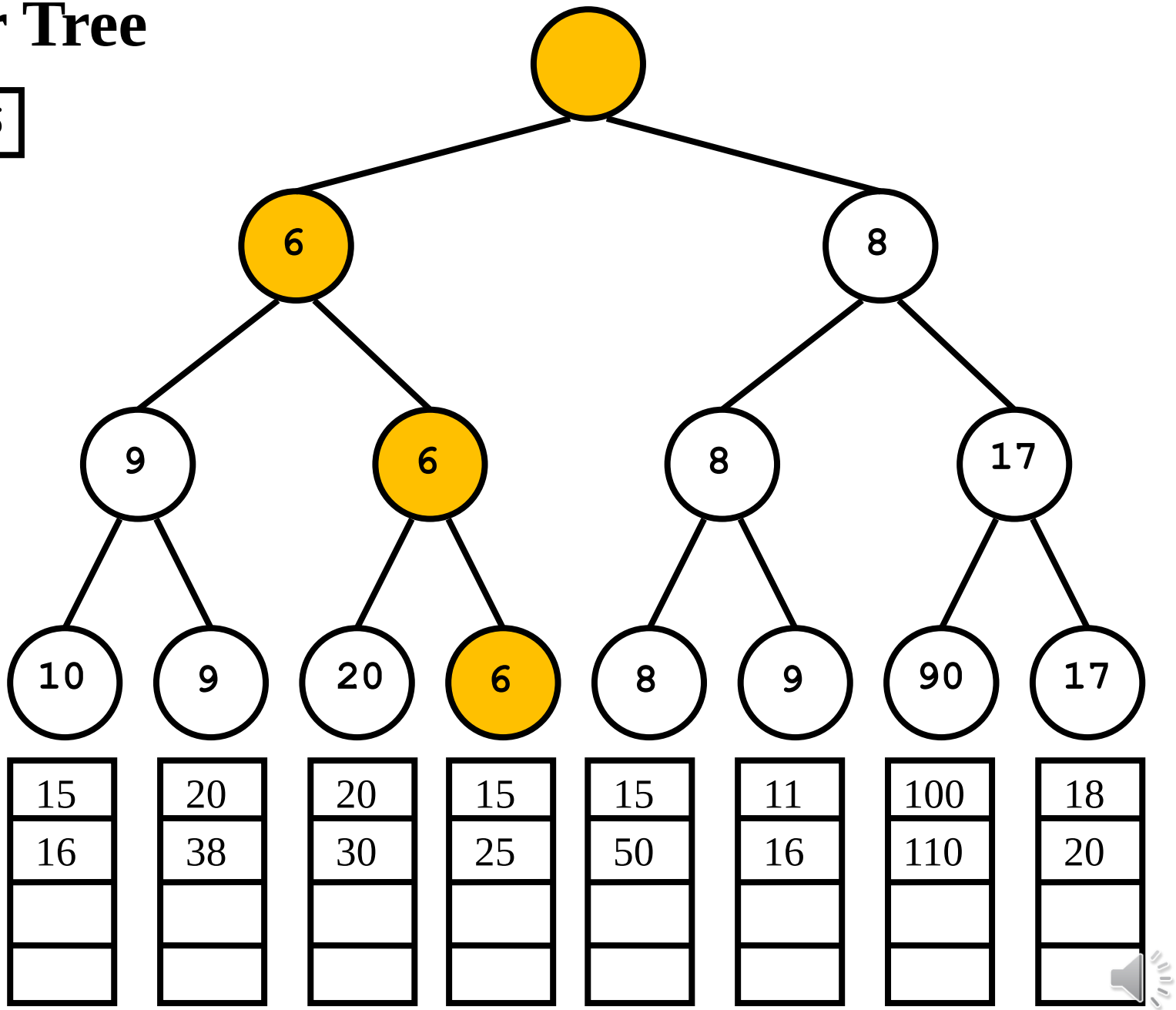
Winner Tree

```
Output: 6
```



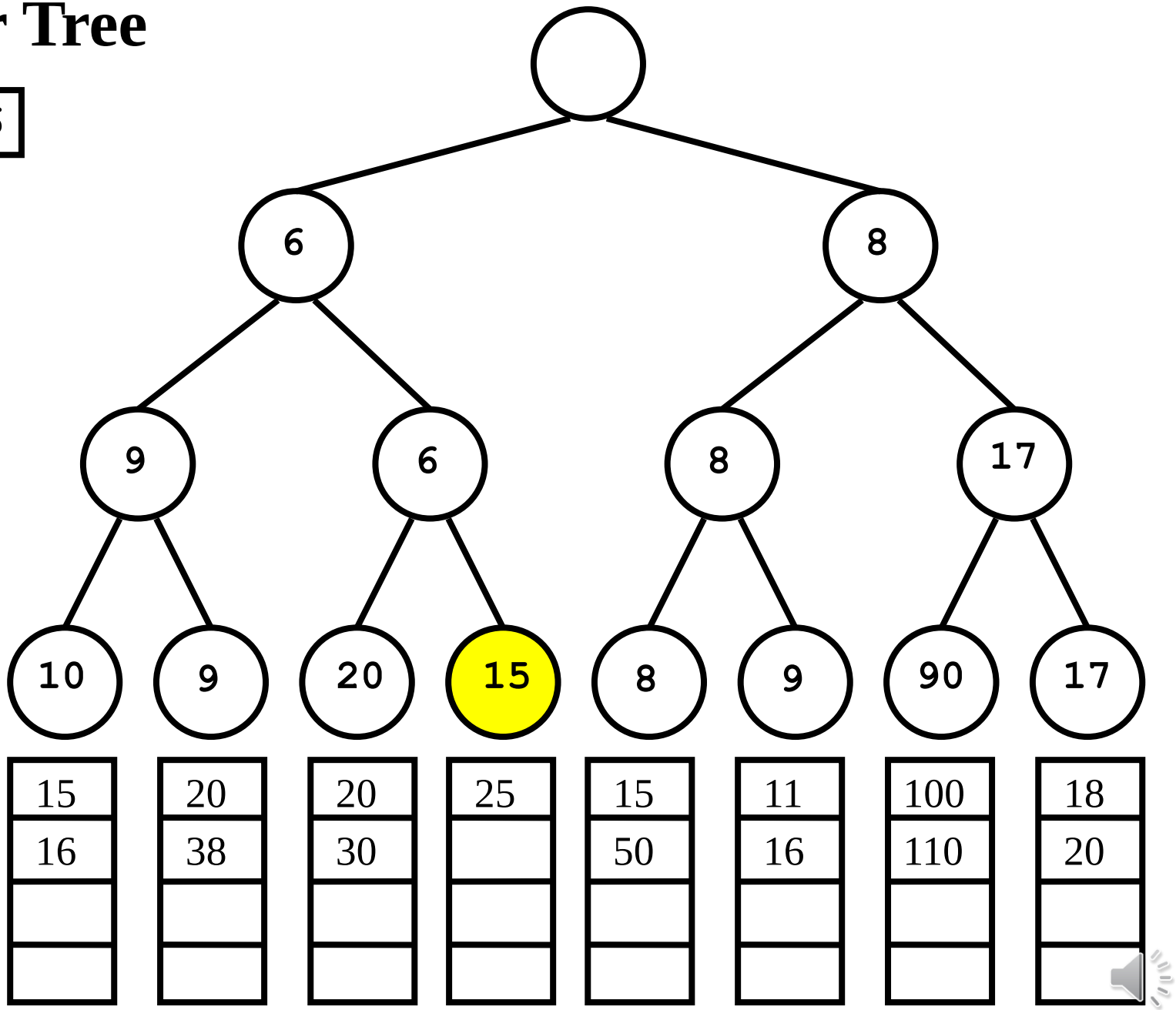
Winner Tree

```
Output: 6
```



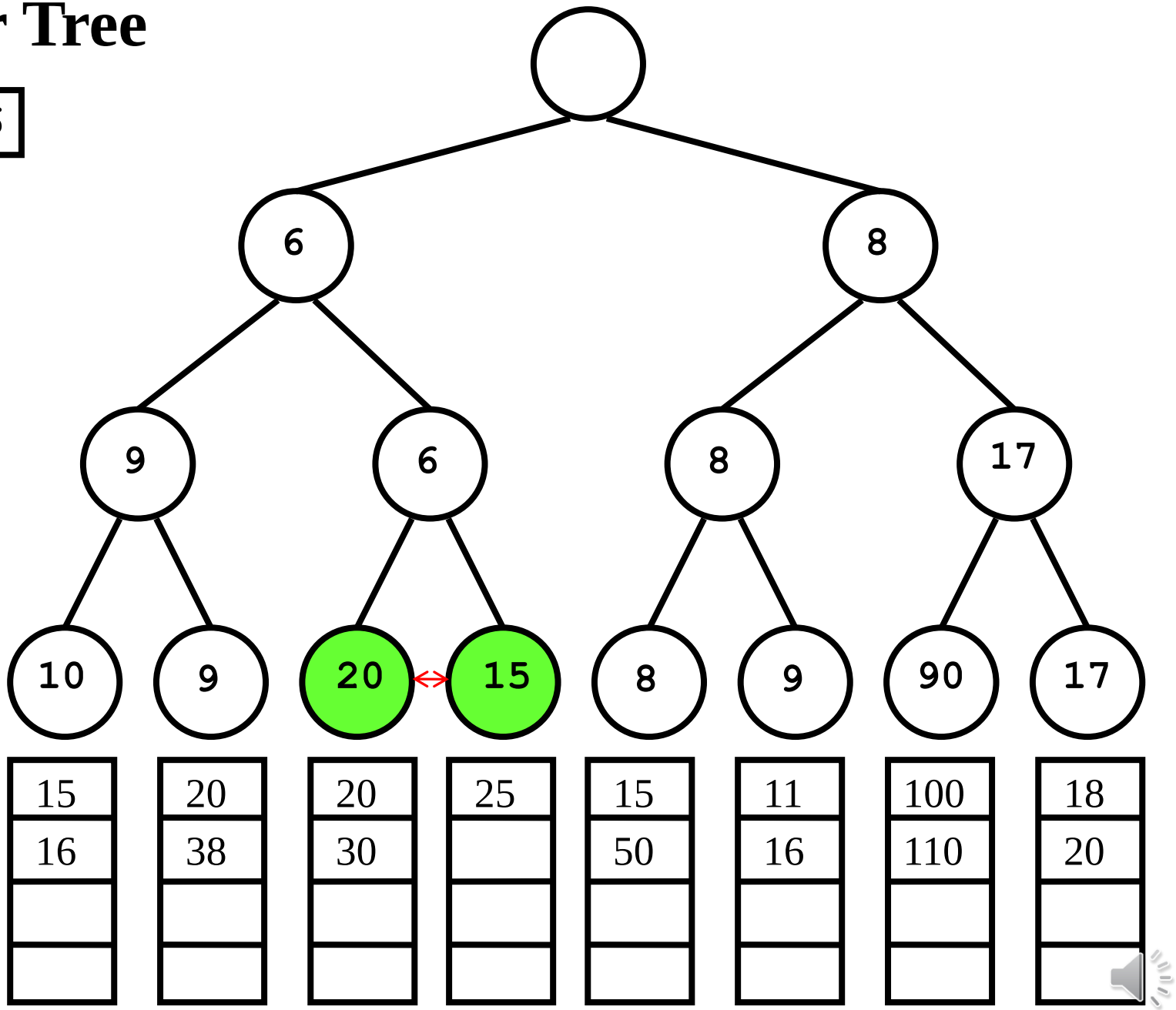
Winner Tree

```
Output: 6
```



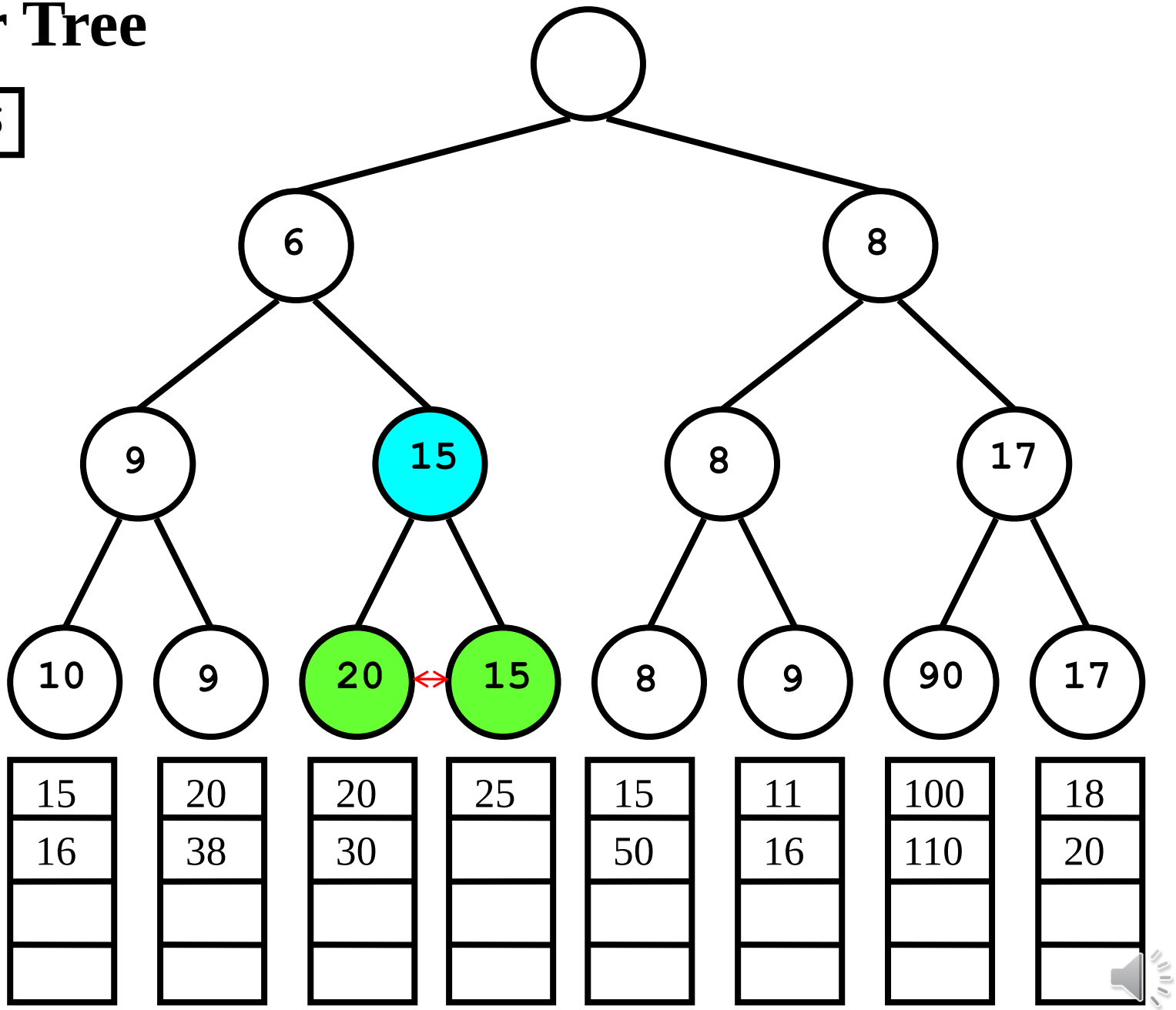
Winner Tree

```
Output: 6
```



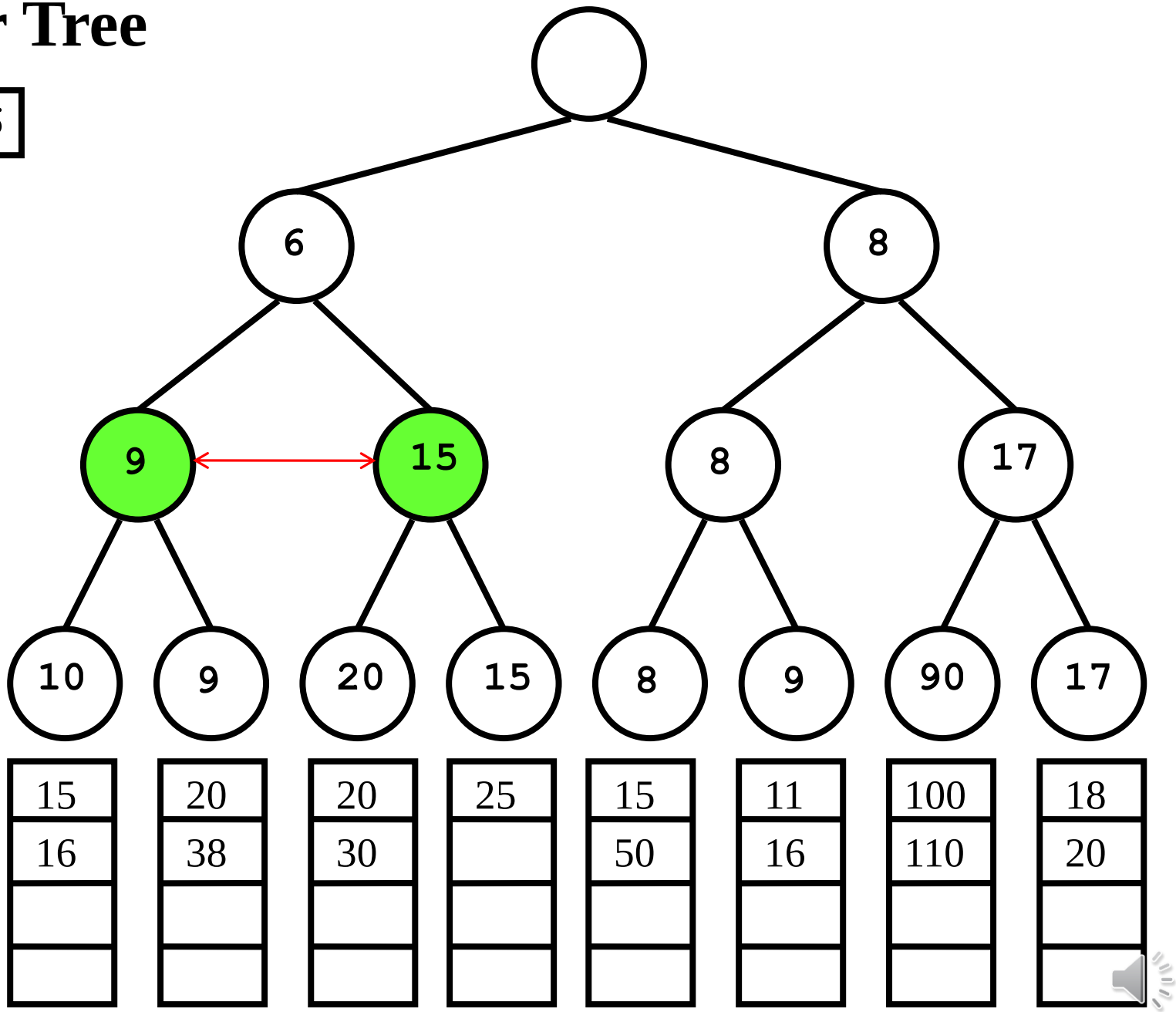
Winner Tree

```
Output: 6
```



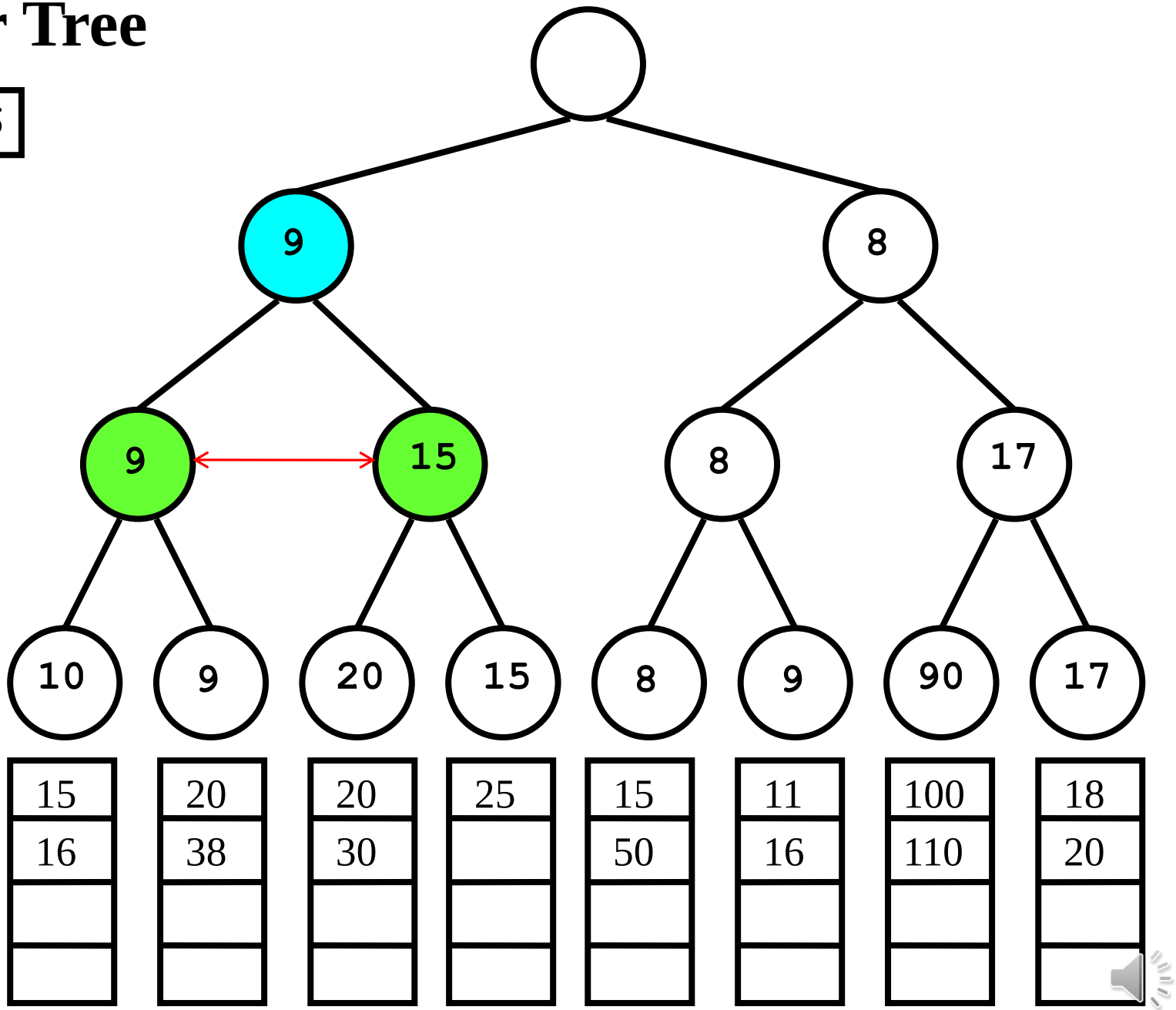
Winner Tree

Output: 6



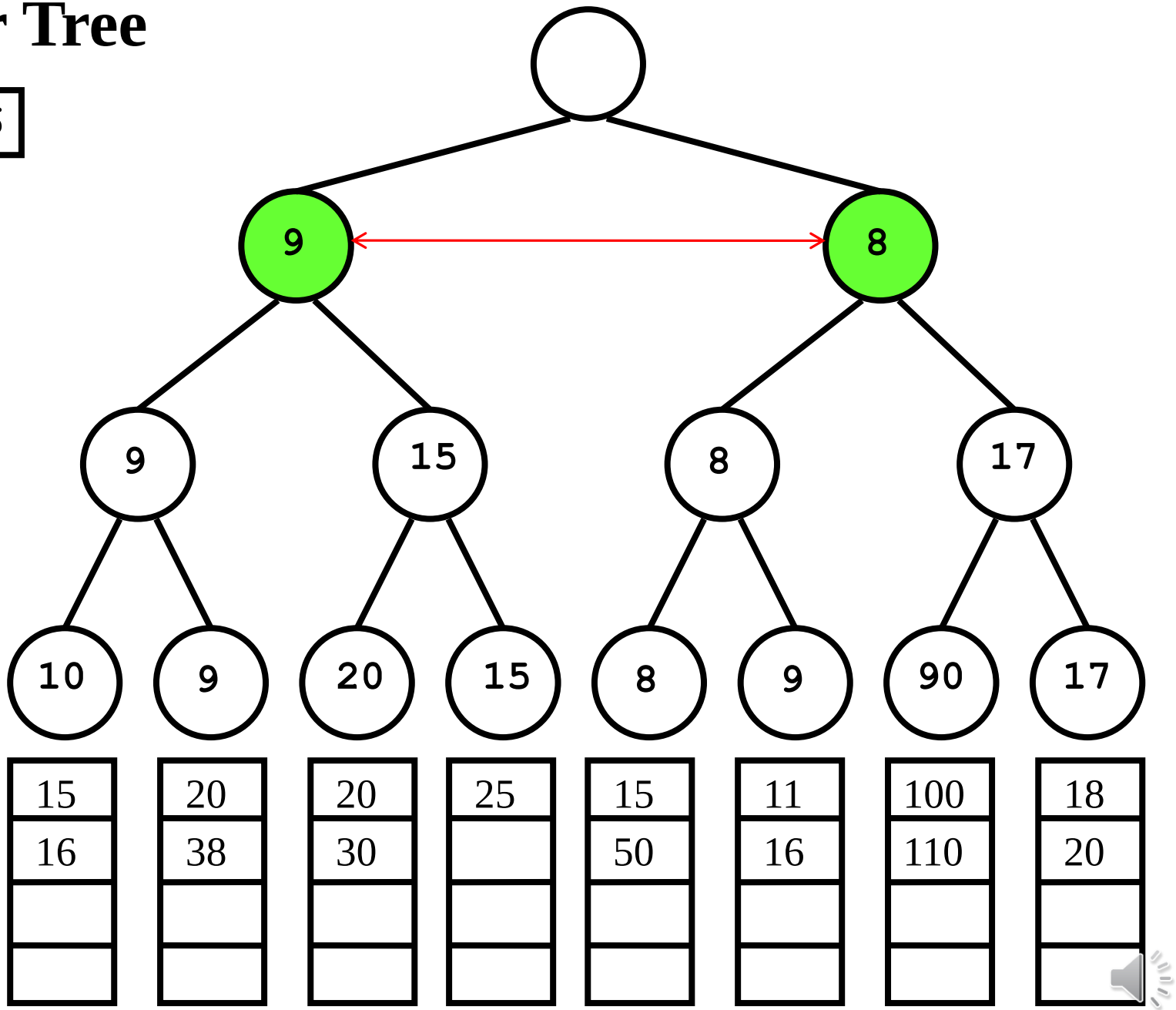
Winner Tree

```
Output: 6
```



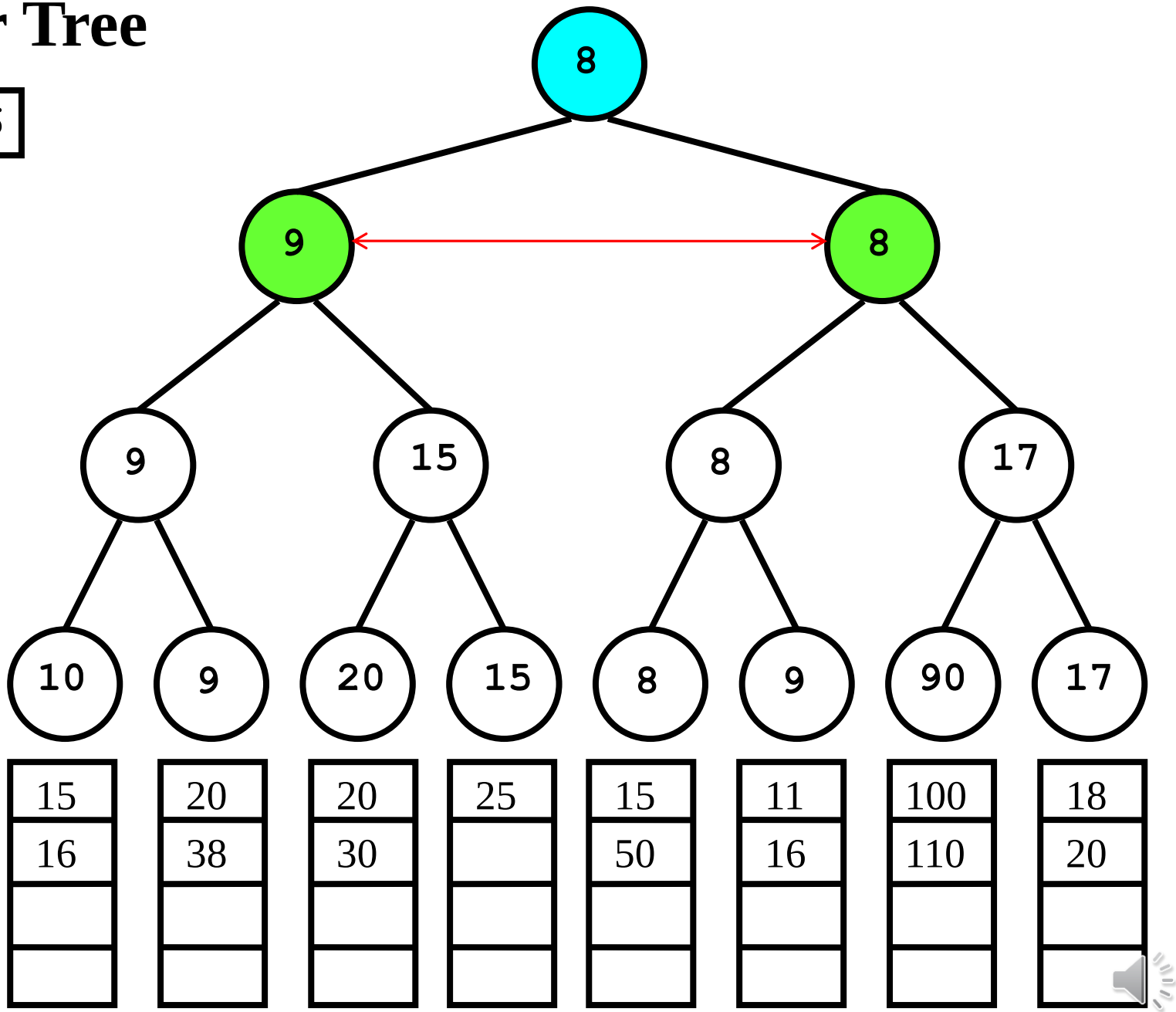
Winner Tree

```
Output: 6
```



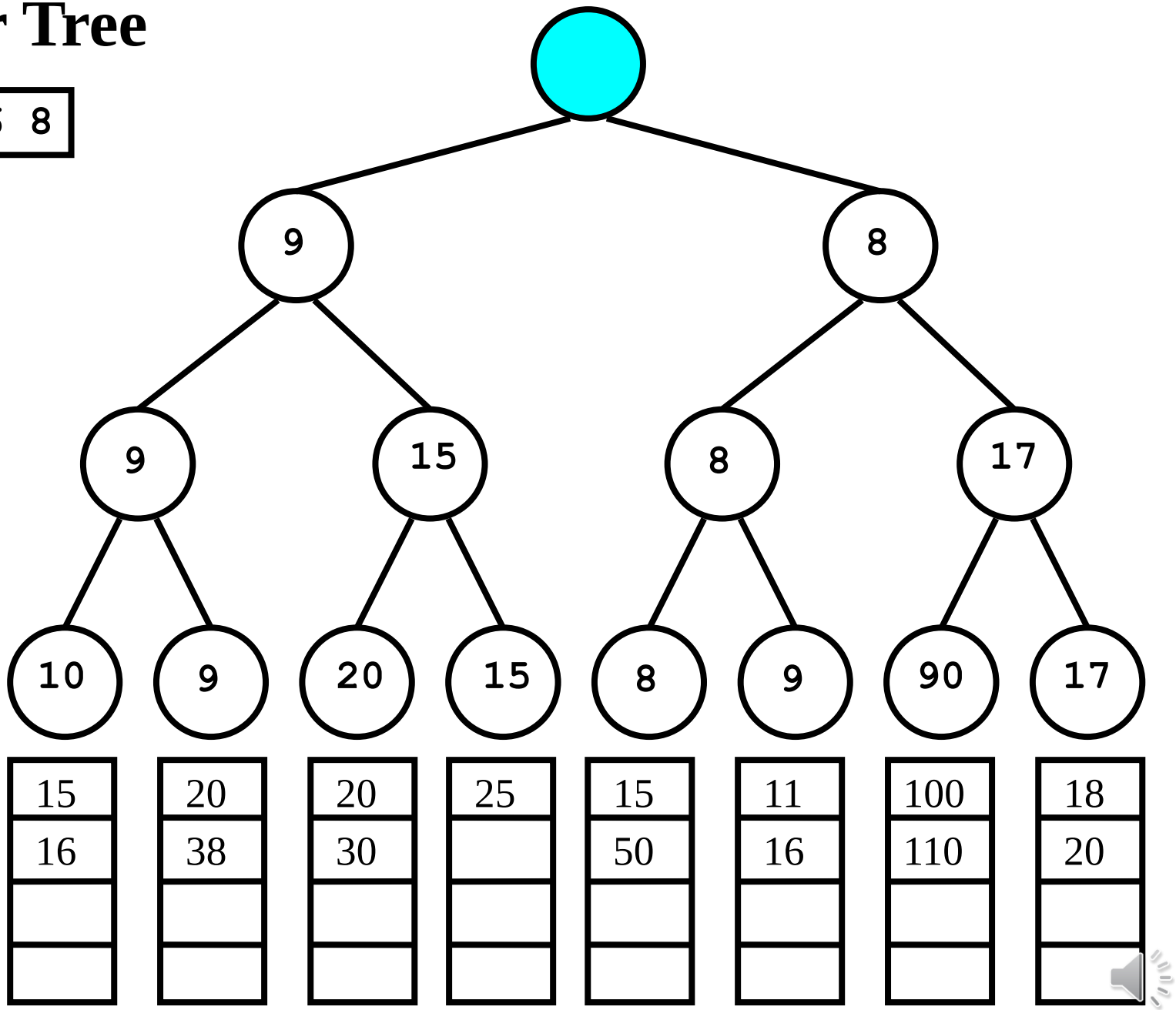
Winner Tree

Output: 6



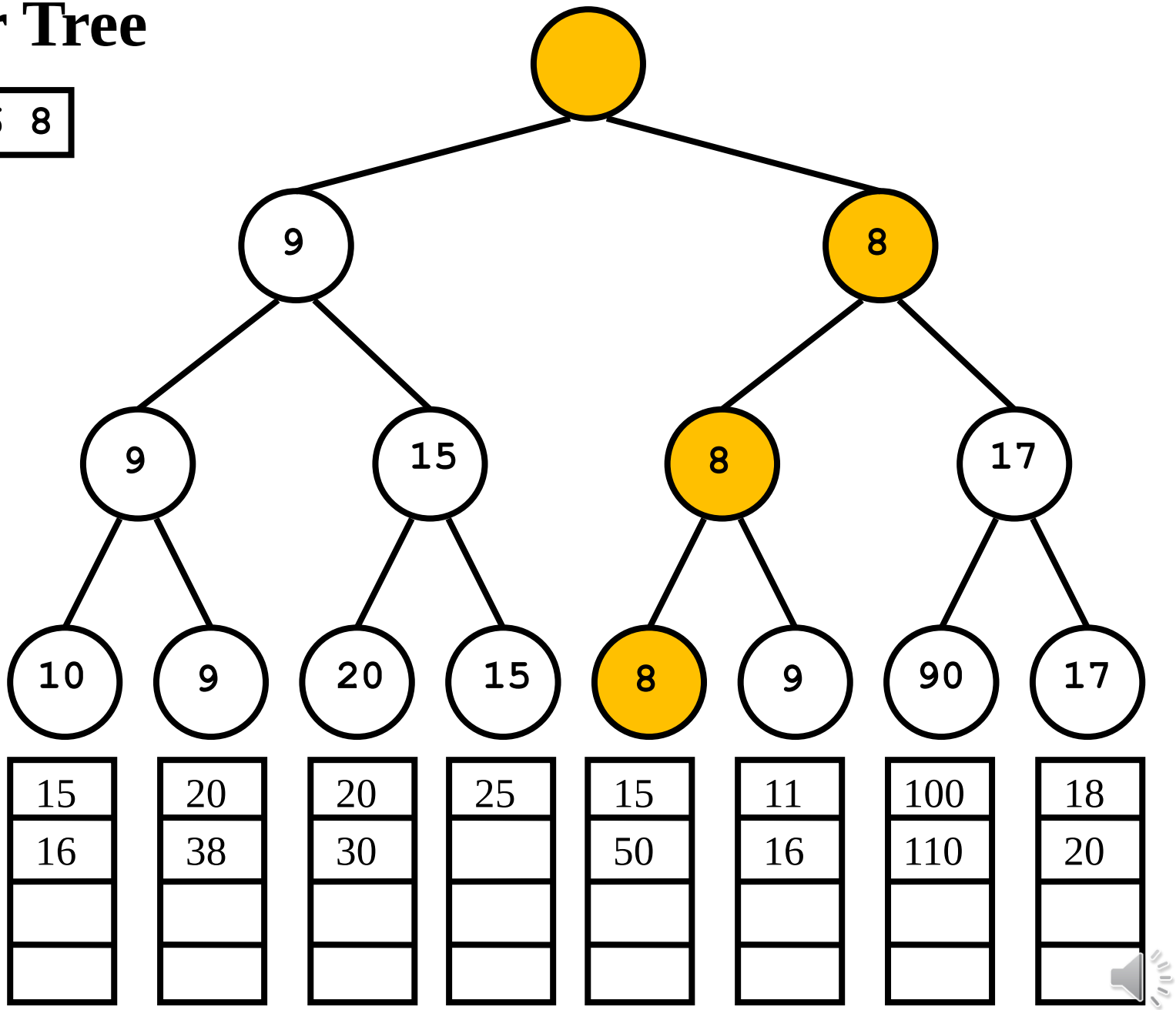
Winner Tree

```
Output: 6 8
```



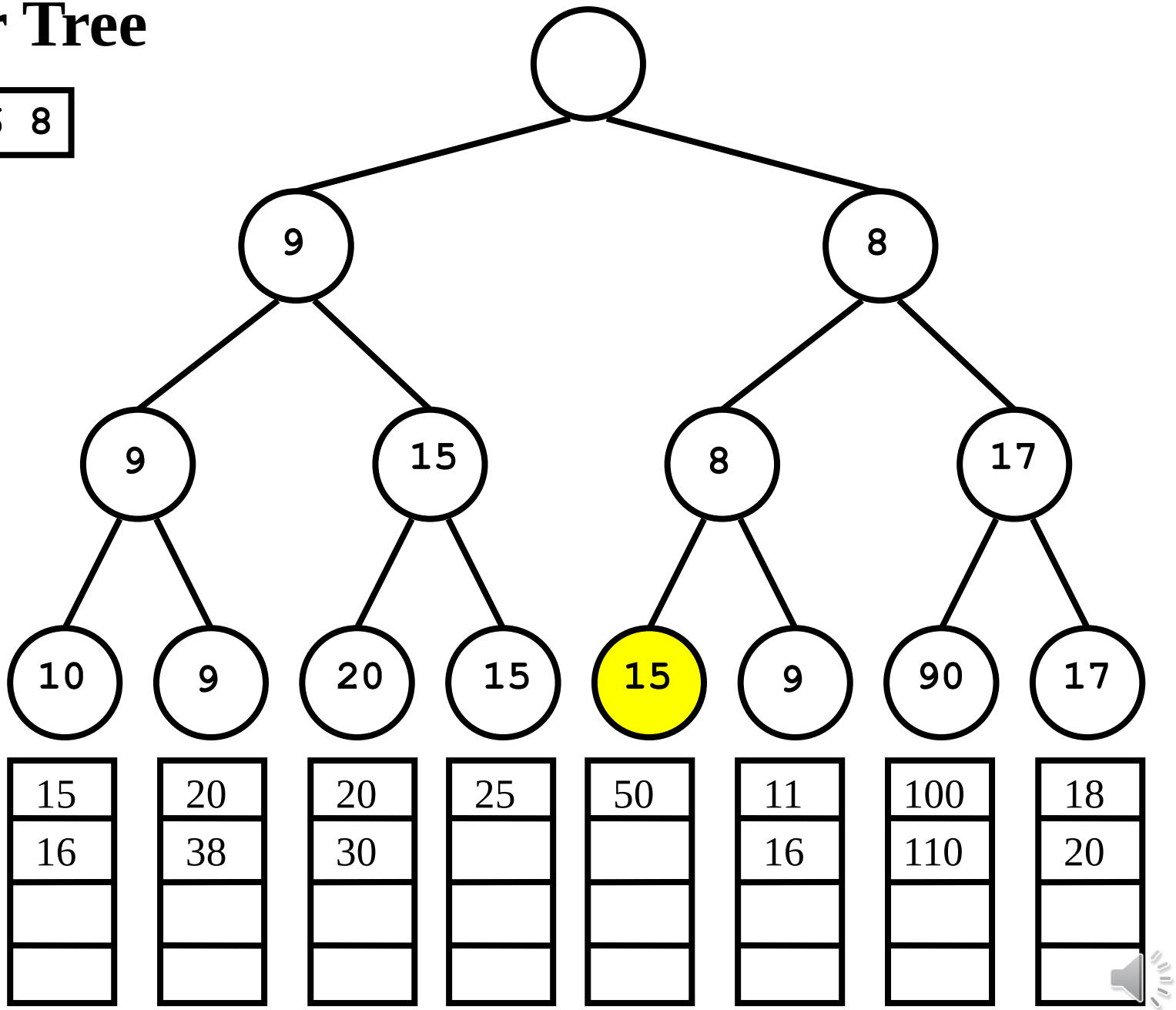
Winner Tree

```
Output: 6 8
```



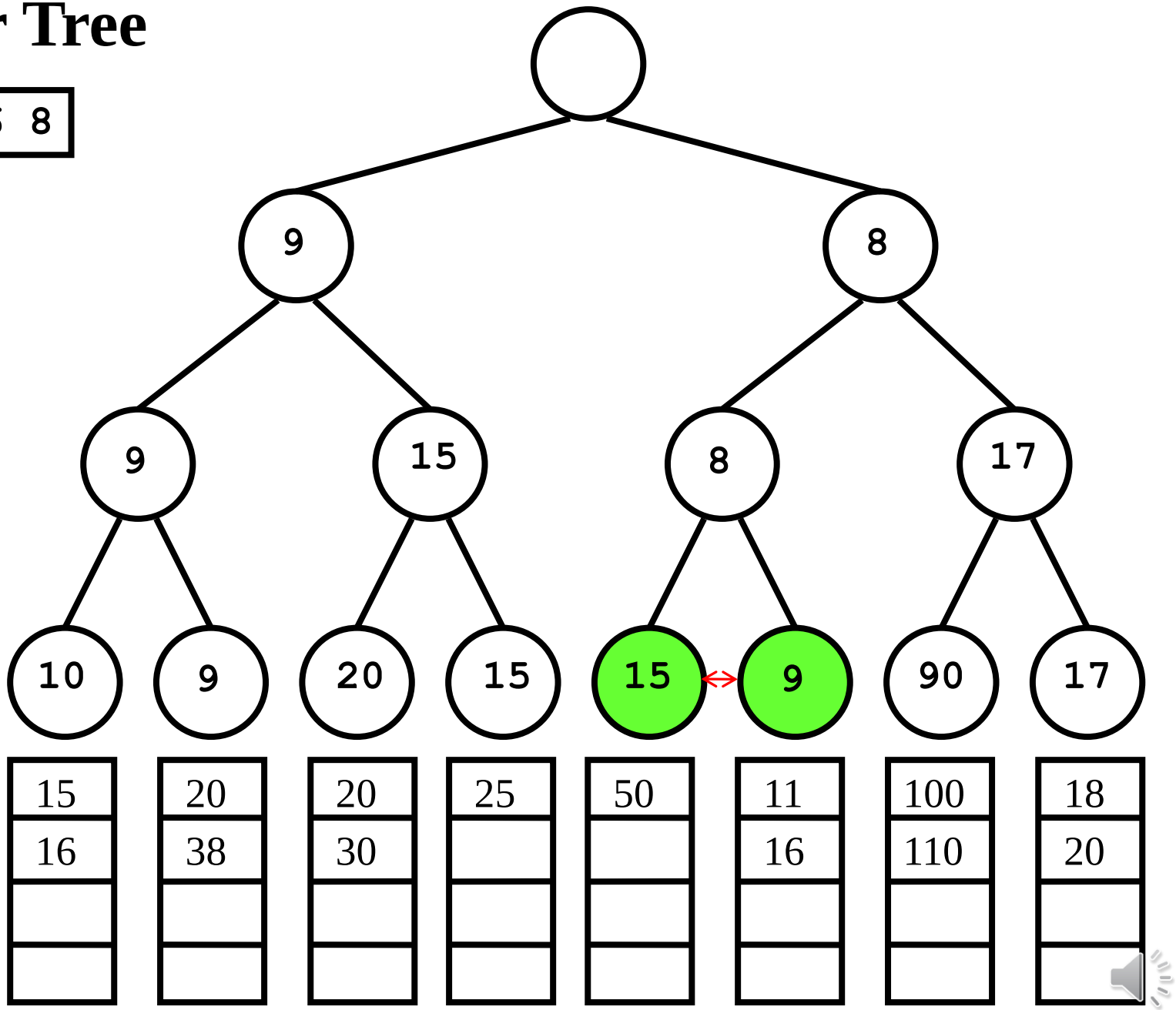
Winner Tree

```
Output: 6 8
```



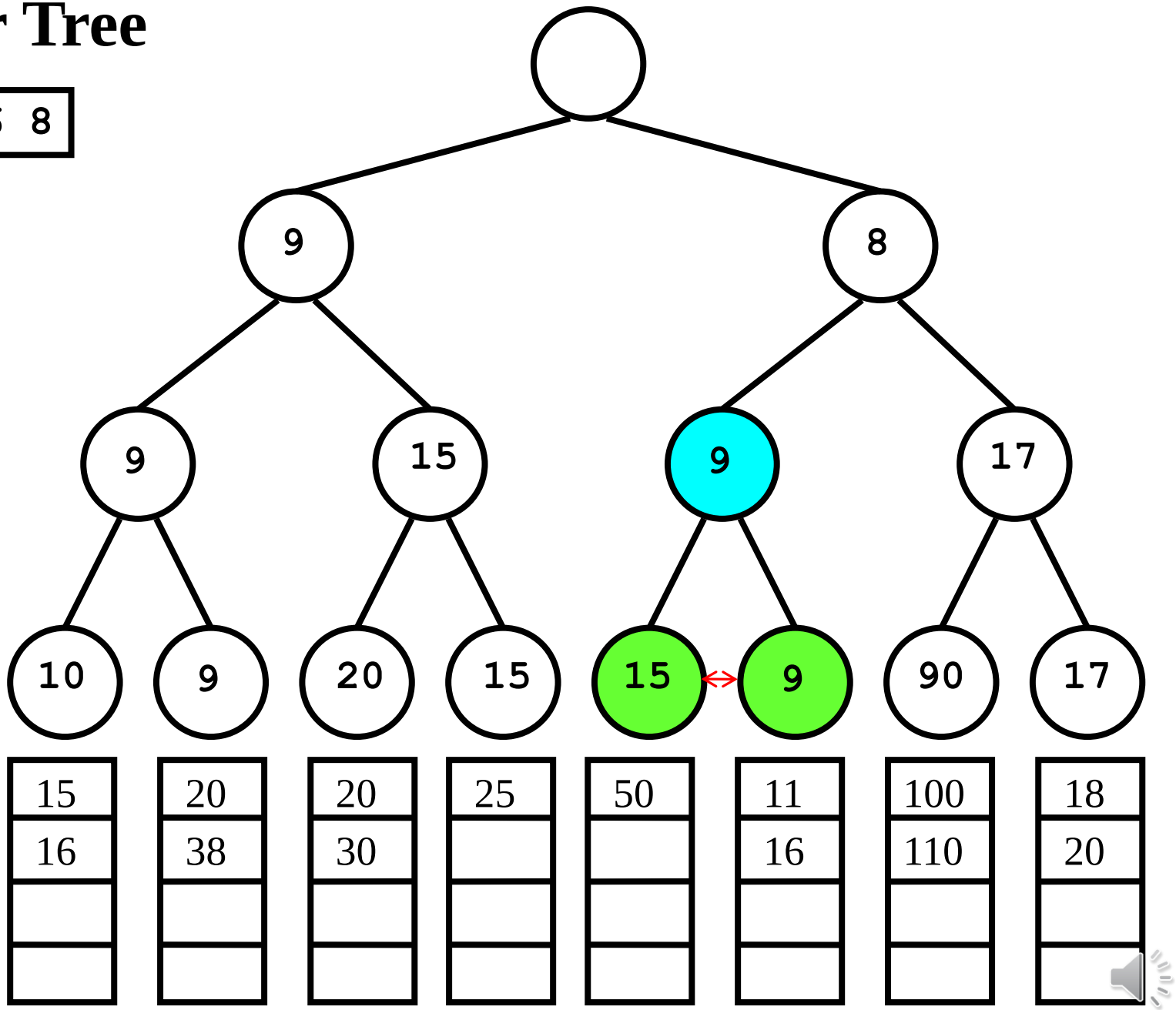
Winner Tree

```
Output: 6 8
```



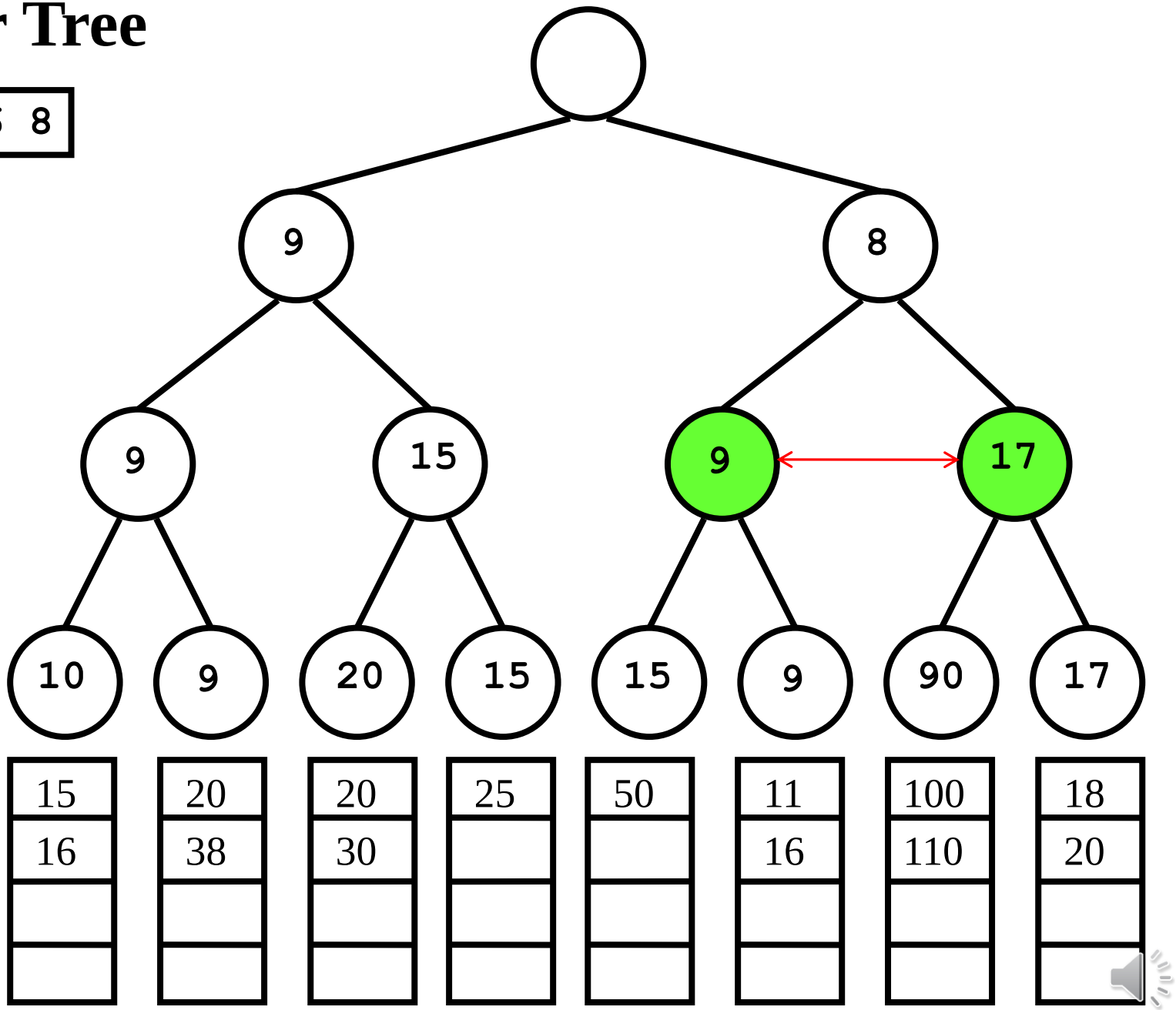
Winner Tree

```
Output: 6 8
```



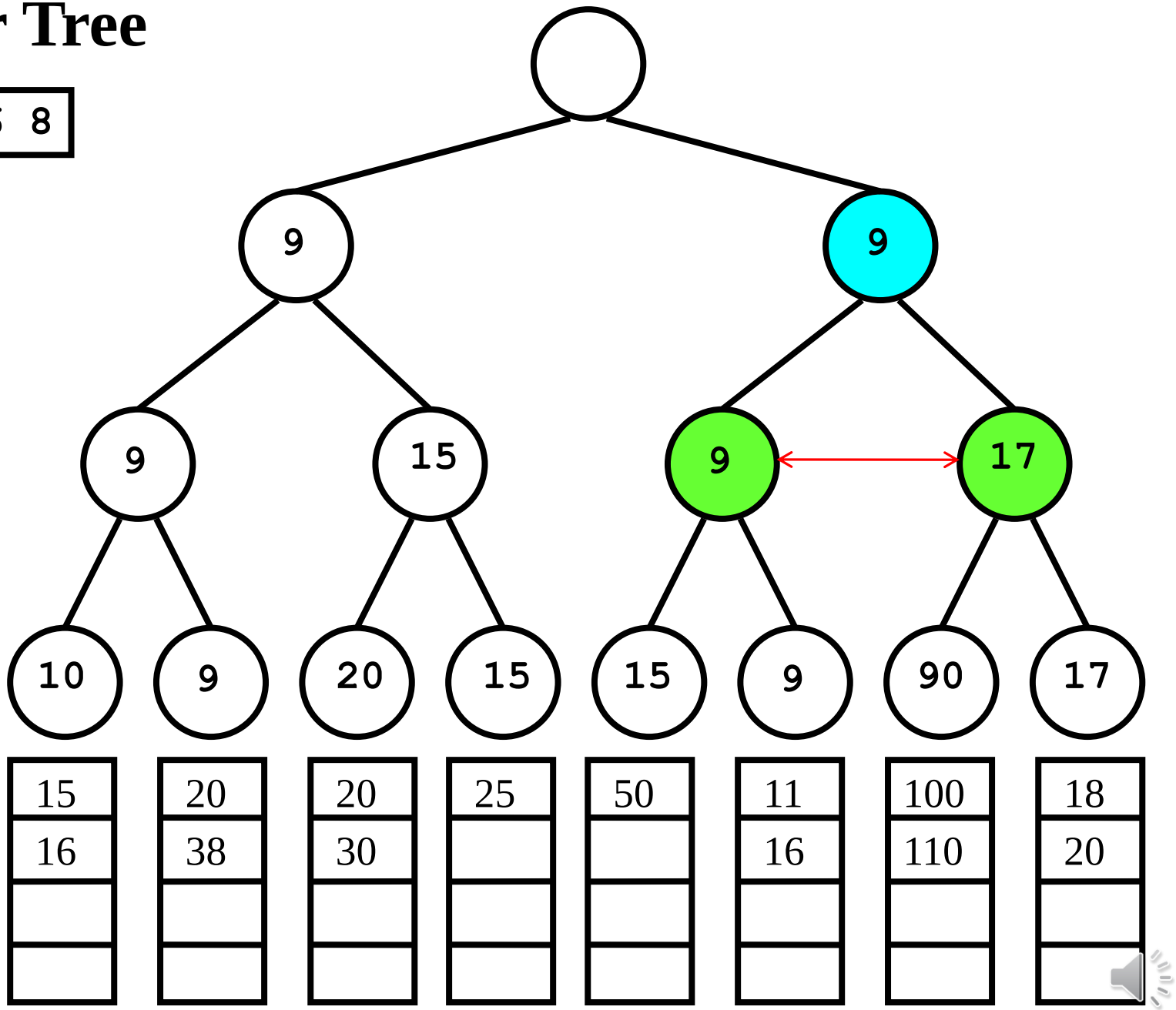
Winner Tree

```
Output: 6 8
```



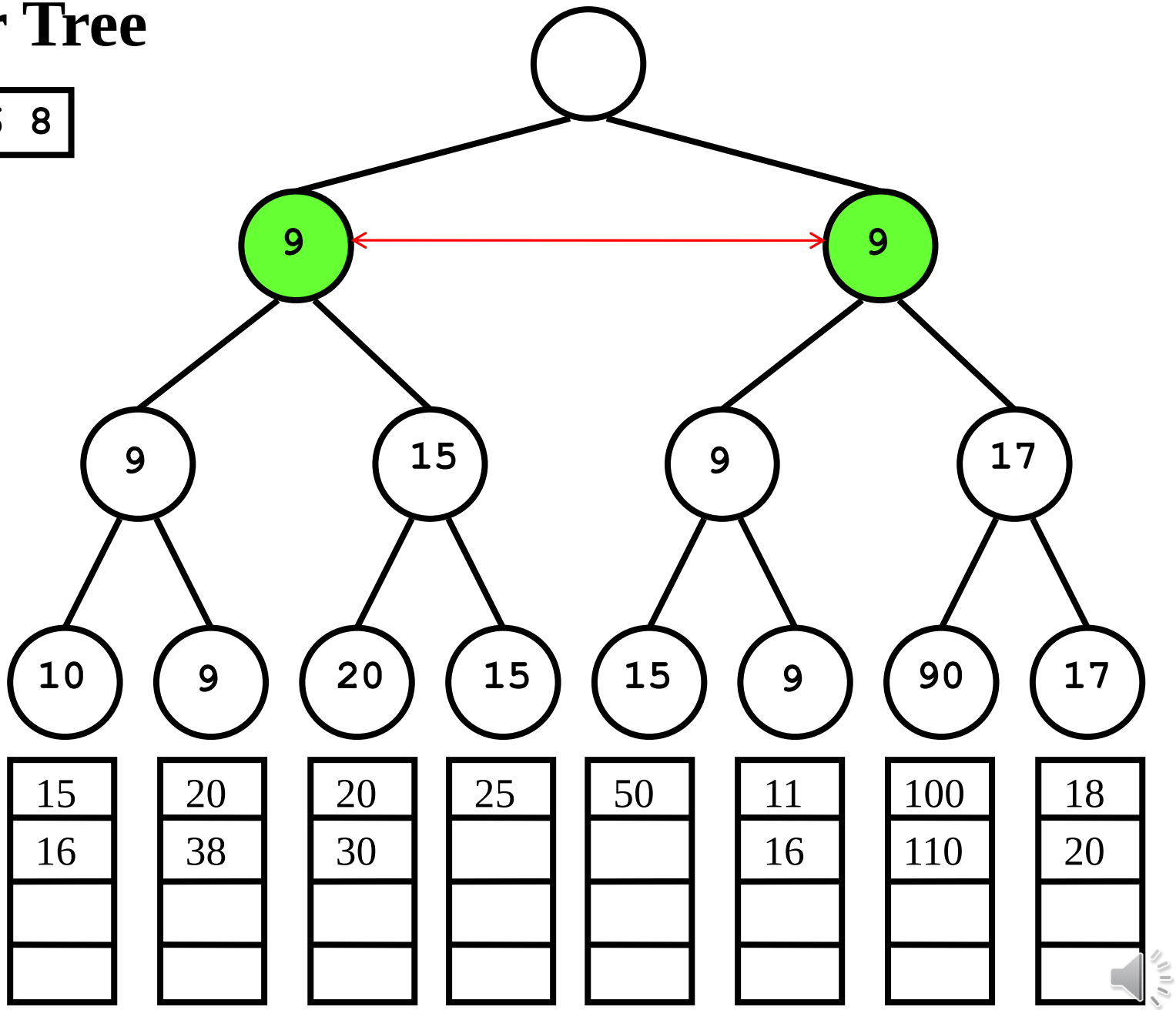
Winner Tree

```
Output: 6 8
```



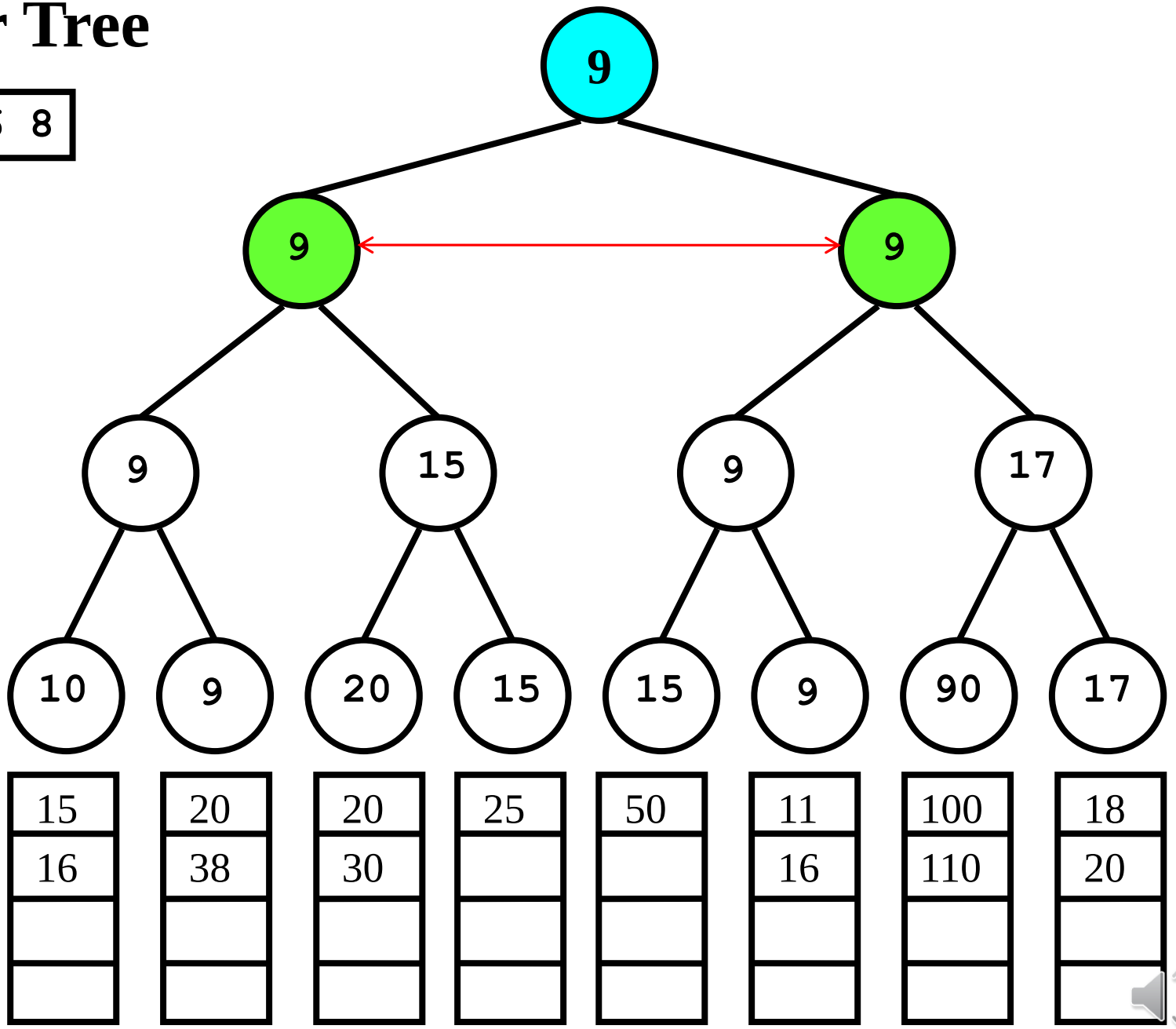
Winner Tree

```
Output: 6 8
```



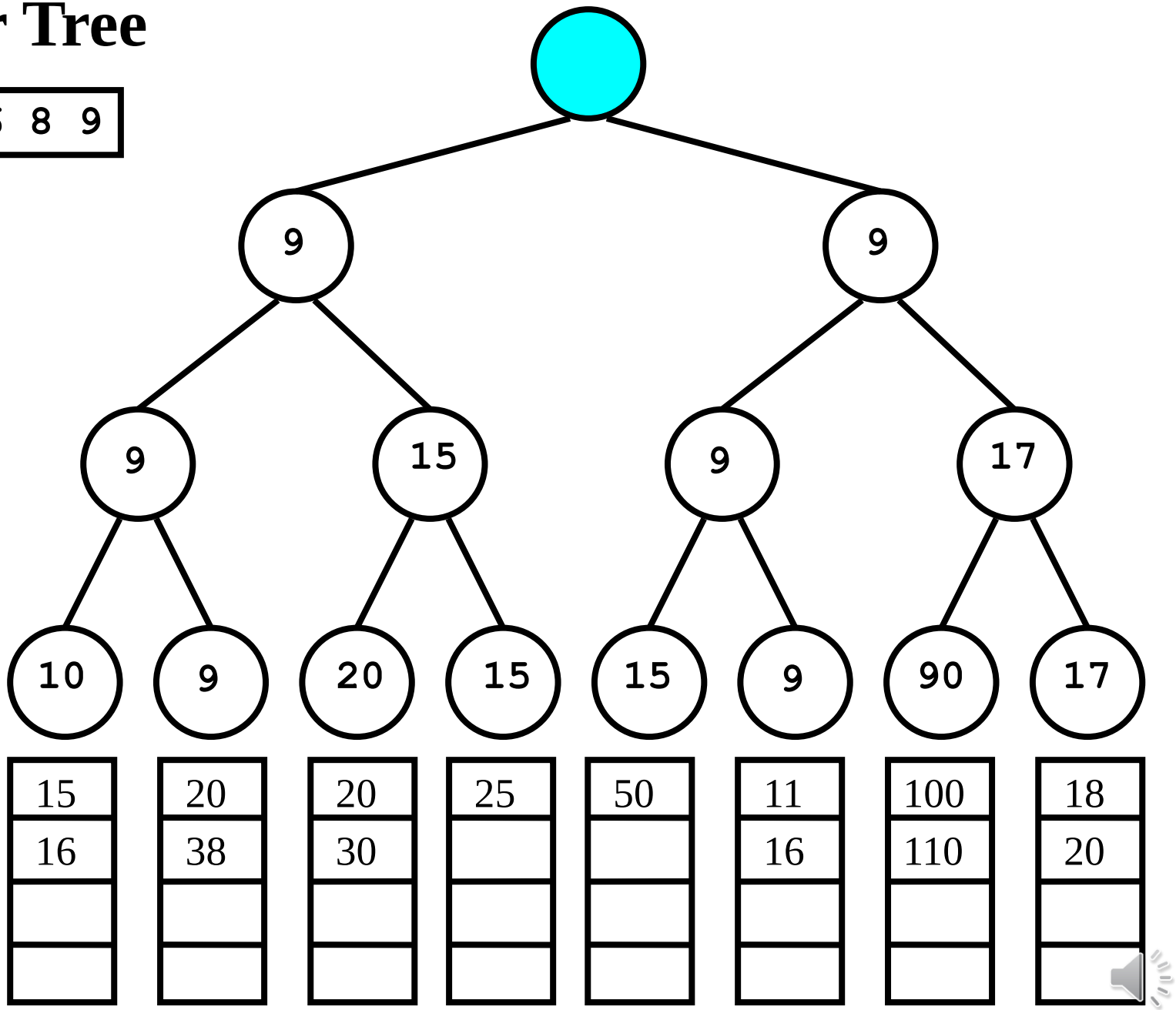
Winner Tree

```
Output: 6 8
```



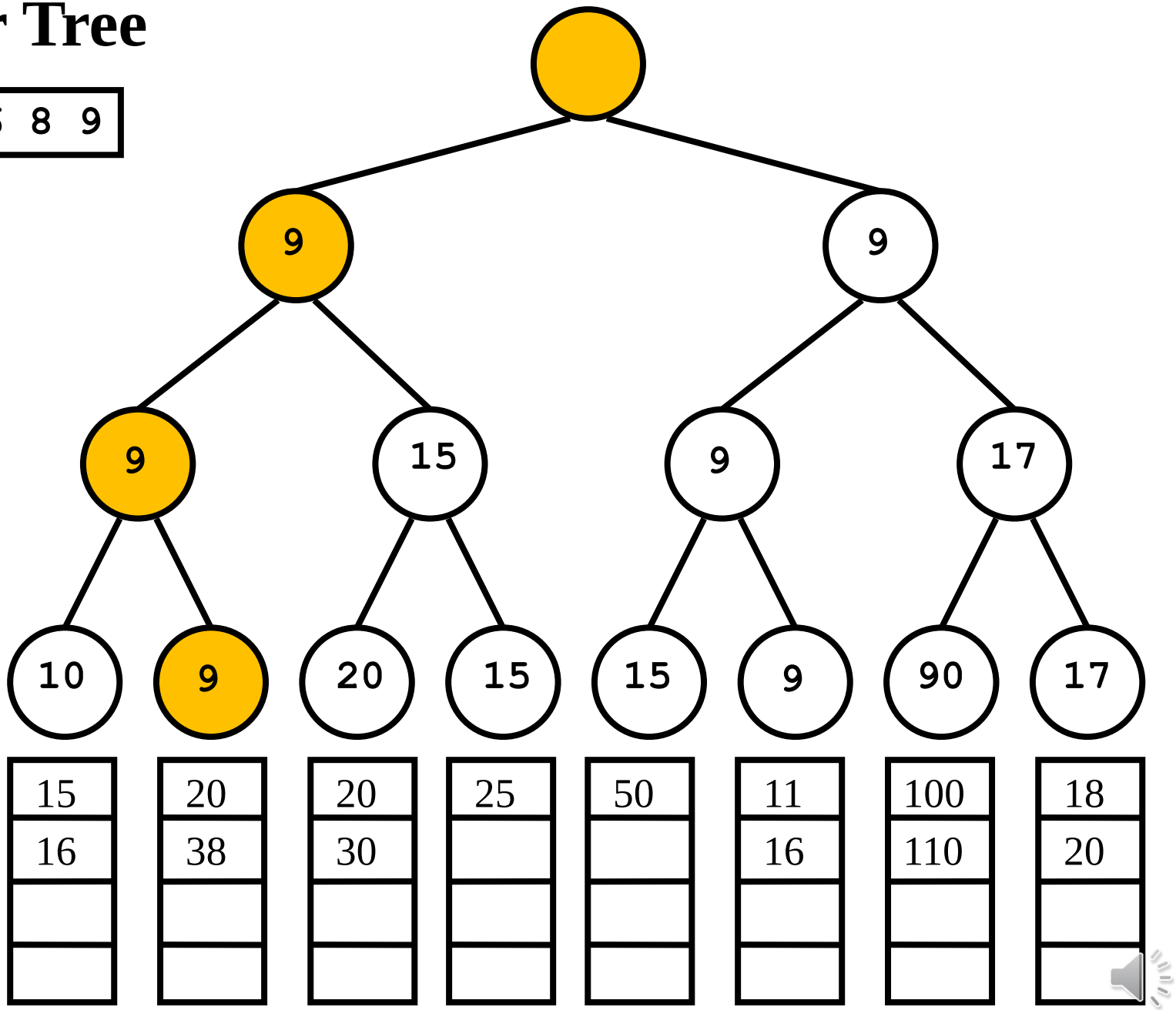
Winner Tree

```
Output: 6 8 9
```



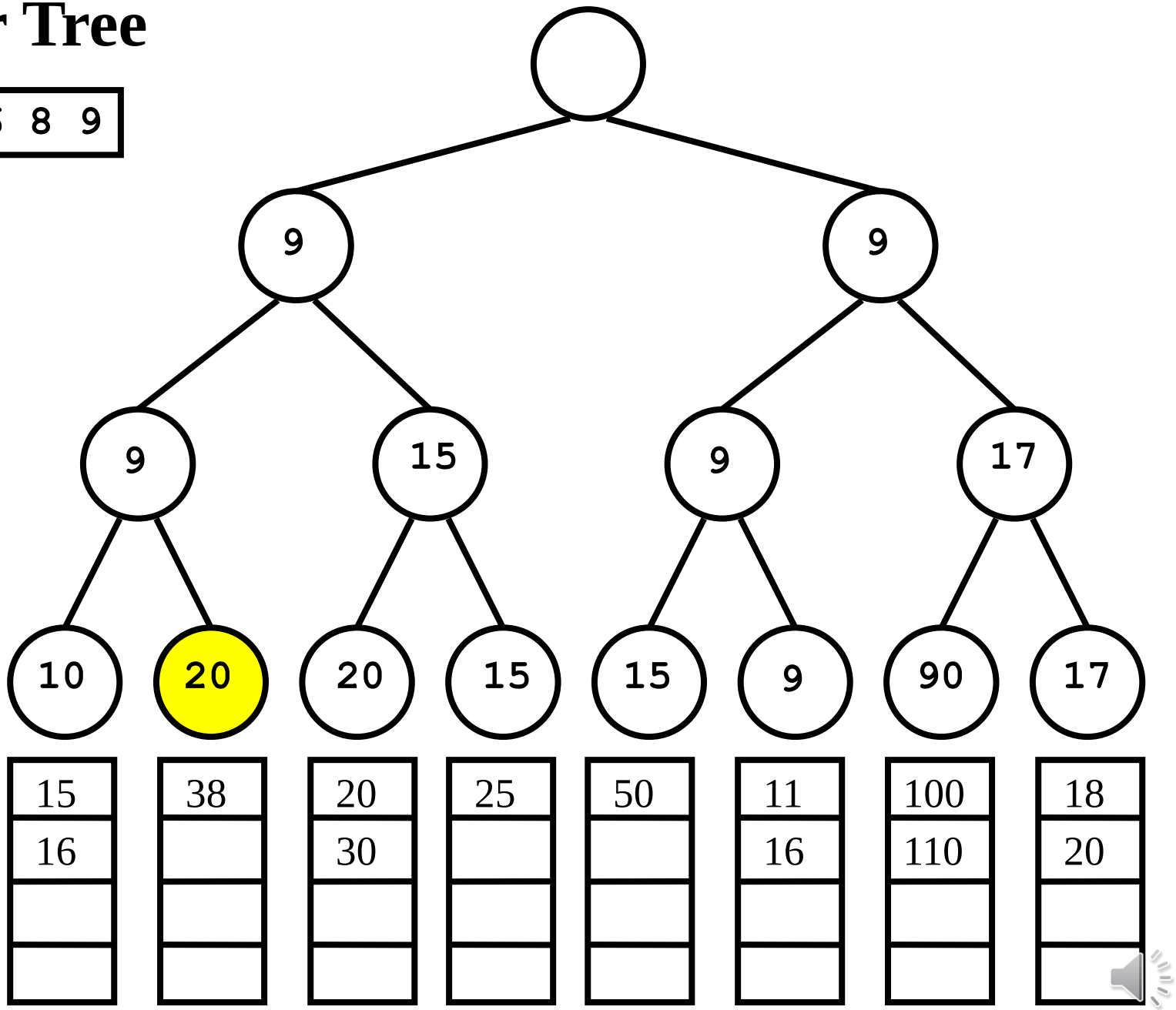
Winner Tree

```
Output: 6 8 9
```



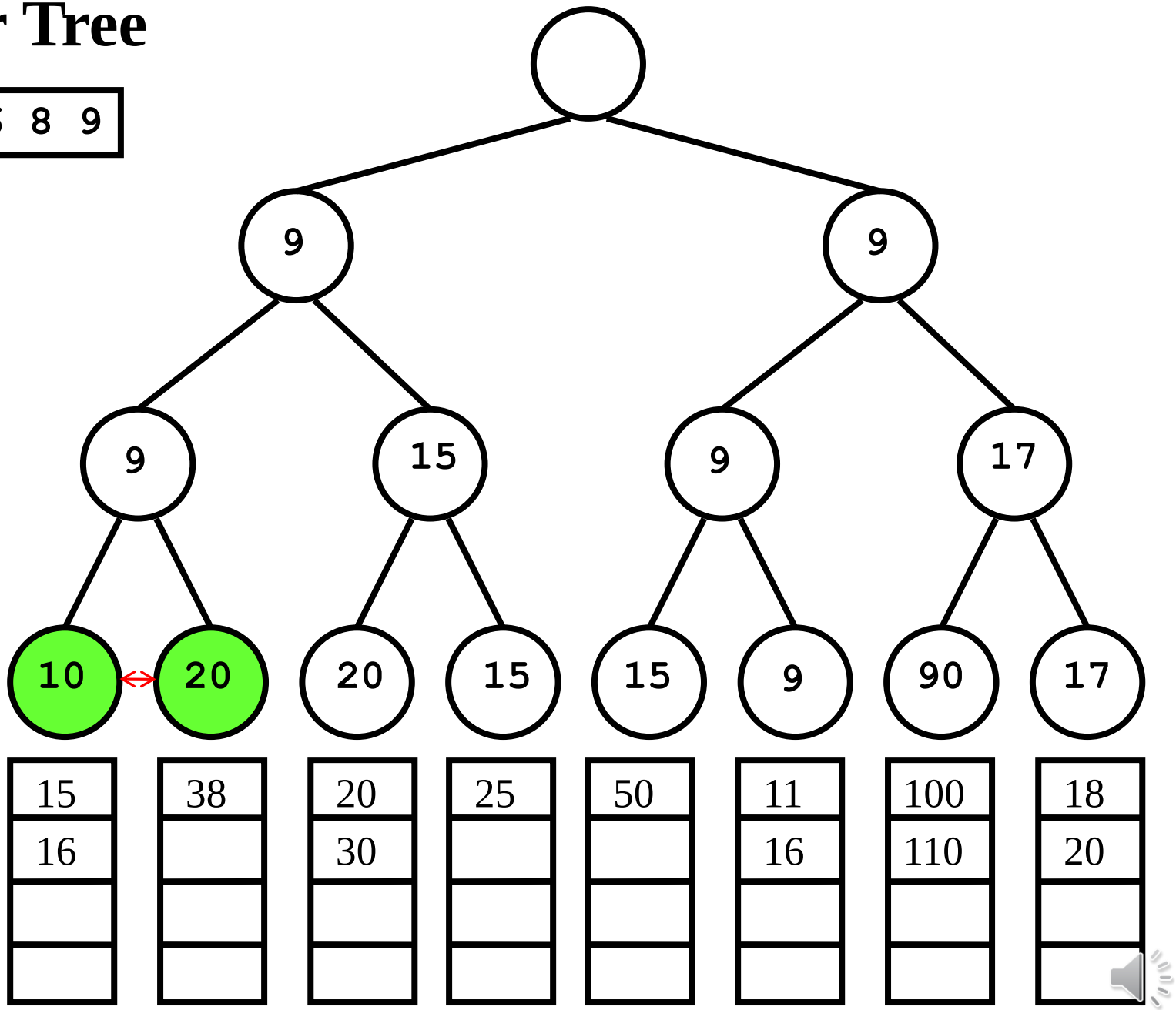
Winner Tree

```
Output: 6 8 9
```



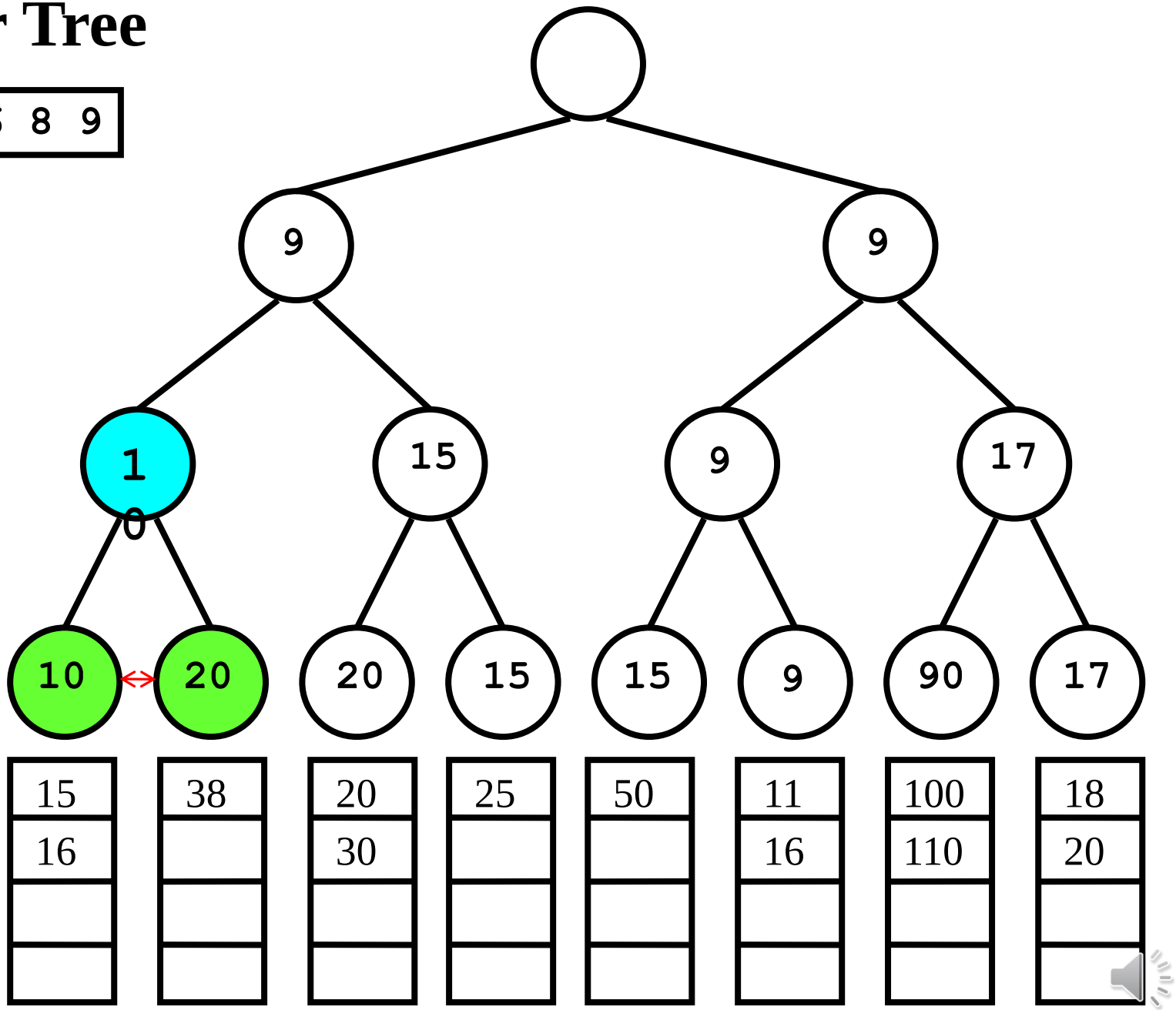
Winner Tree

```
Output: 6 8 9
```



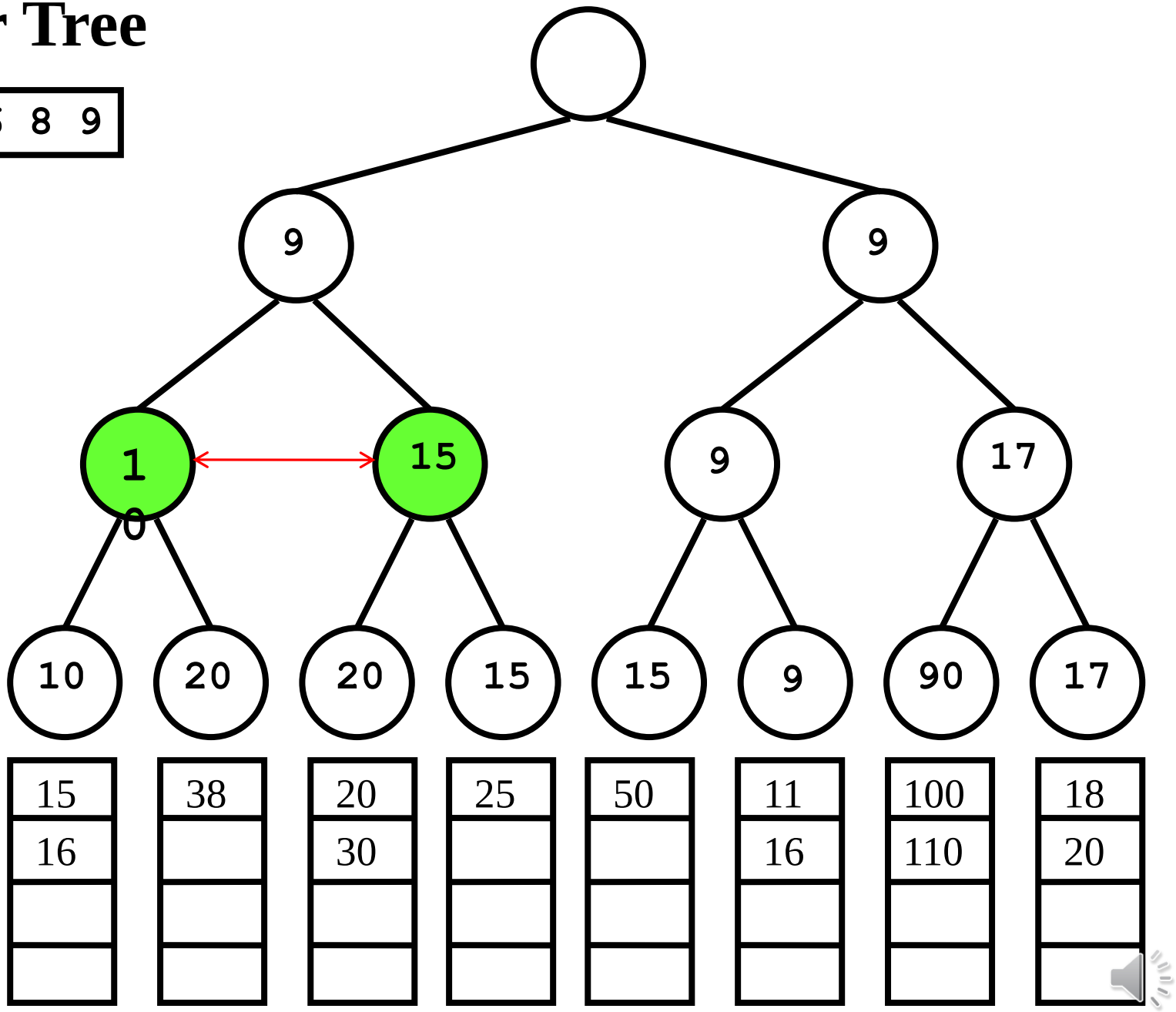
Winner Tree

```
Output: 6 8 9
```



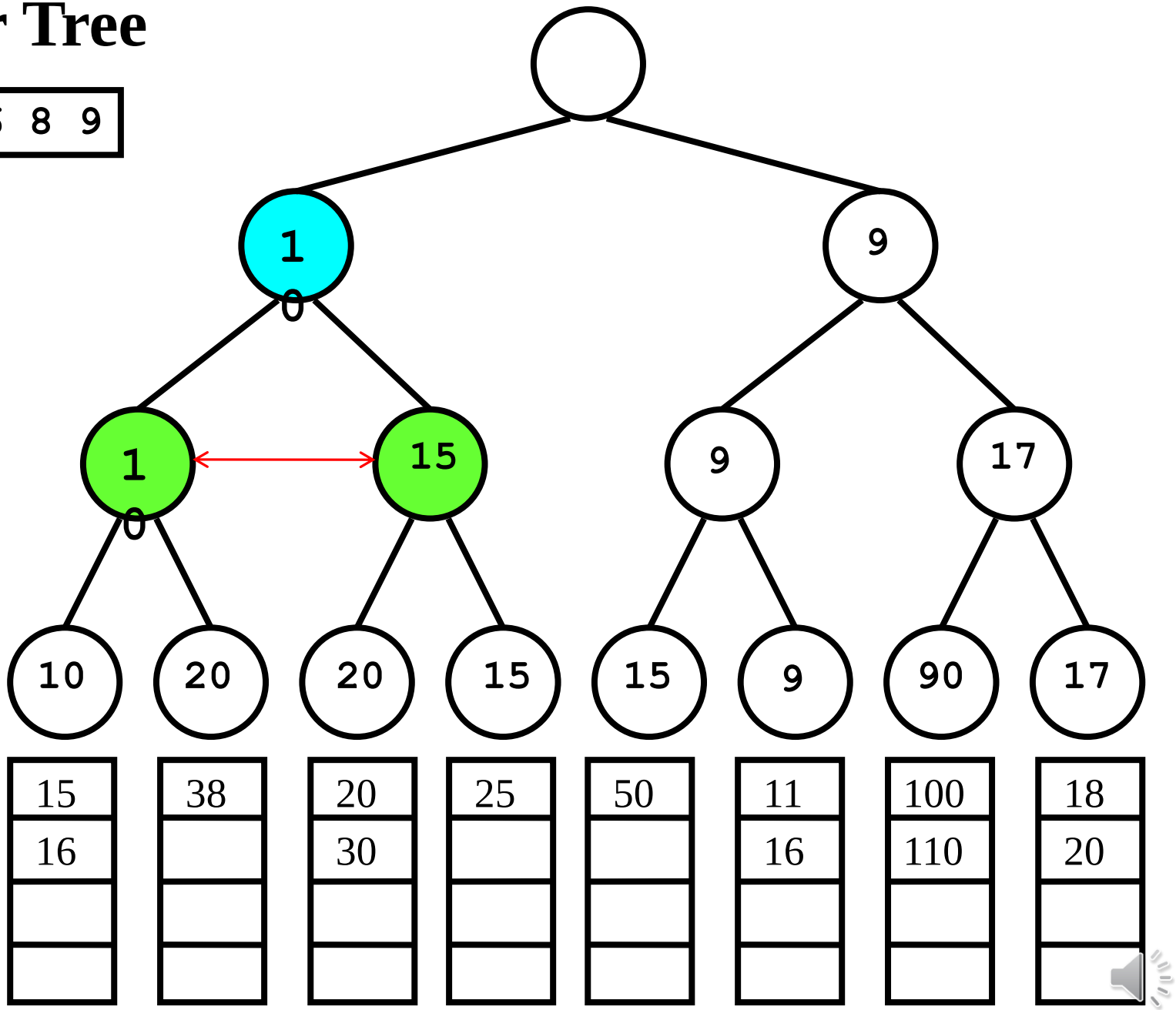
Winner Tree

```
Output: 6 8 9
```



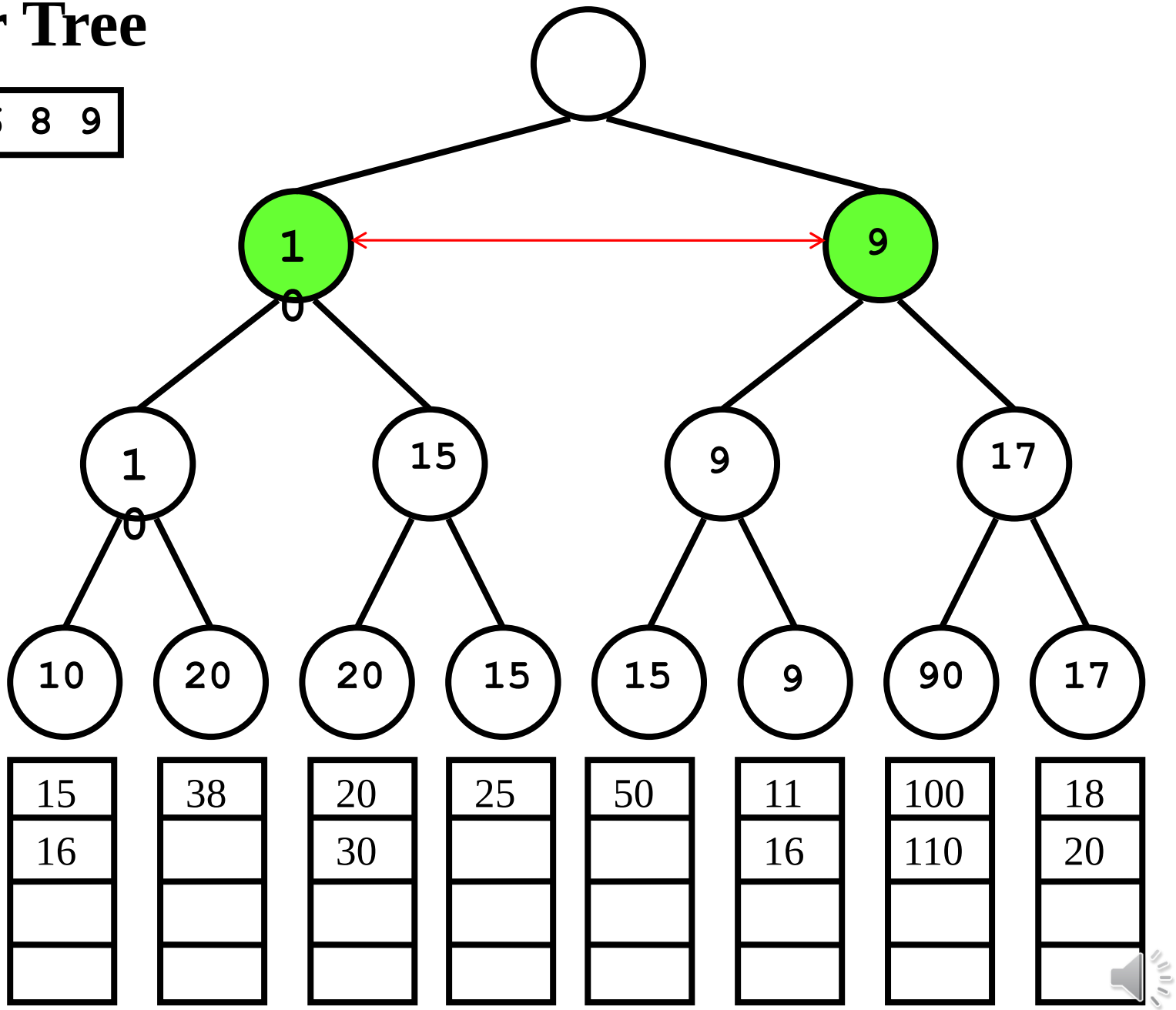
Winner Tree

```
Output: 6 8 9
```



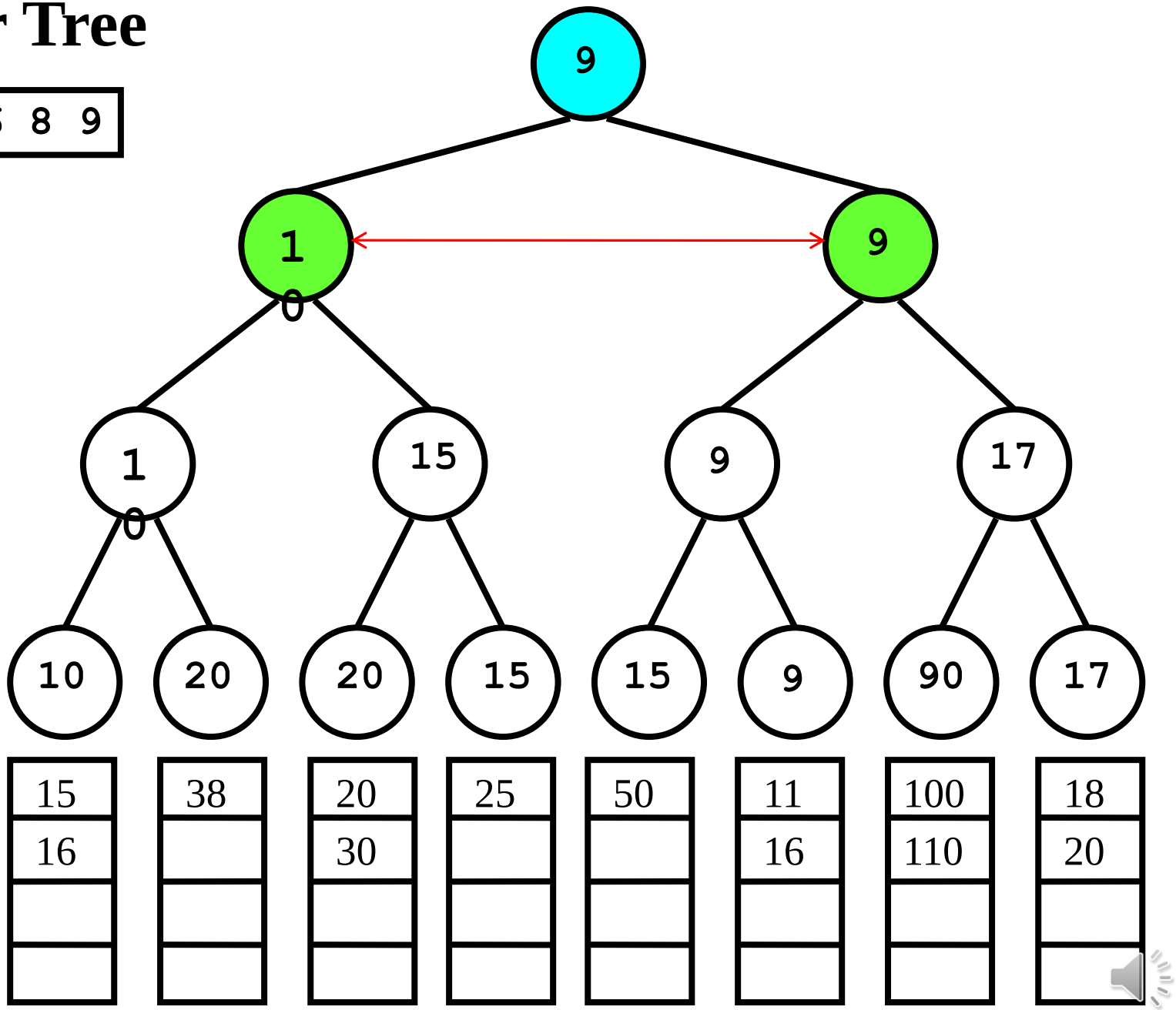
Winner Tree

```
Output: 6 8 9
```



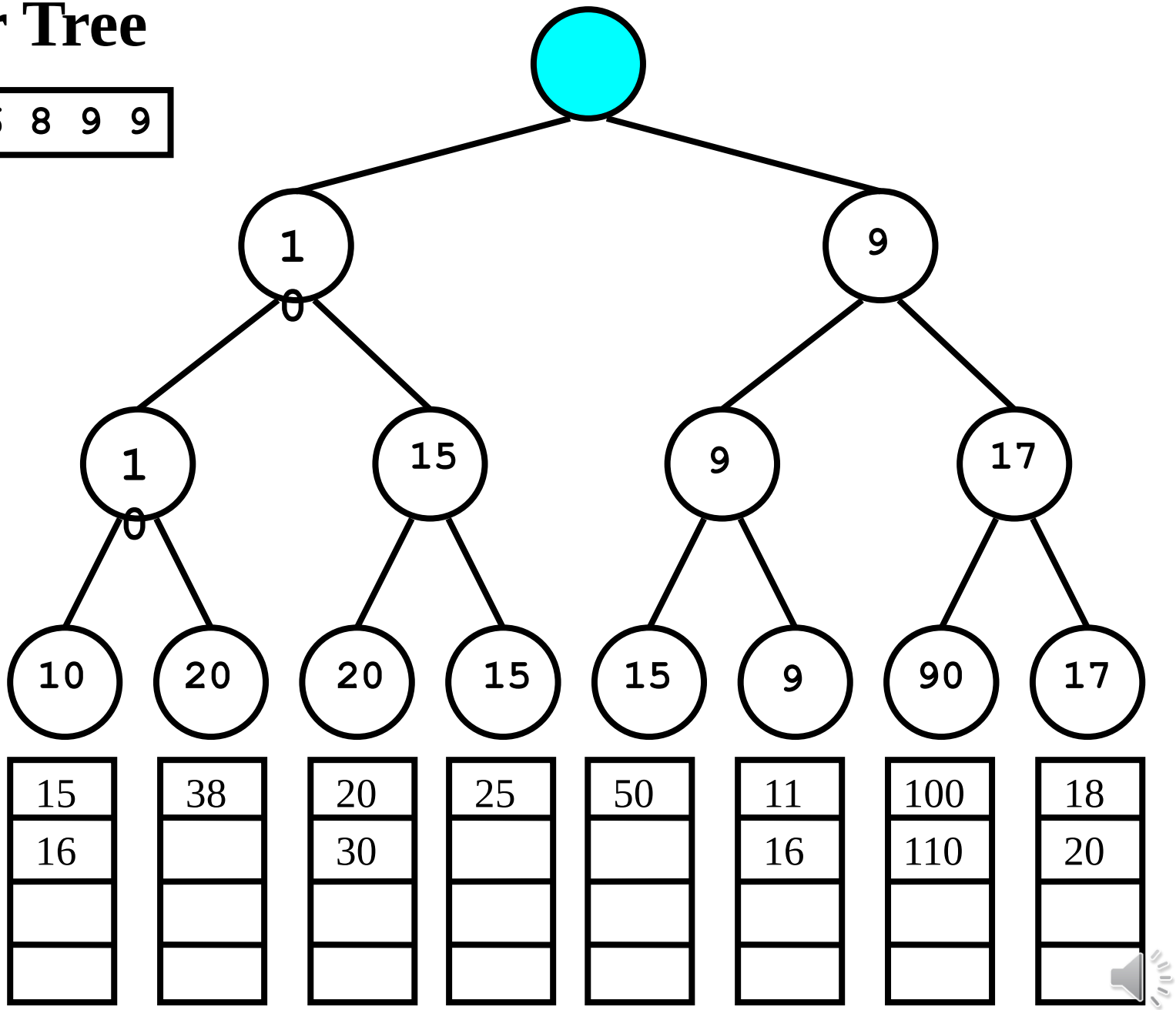
Winner Tree

```
Output: 6 8 9
```



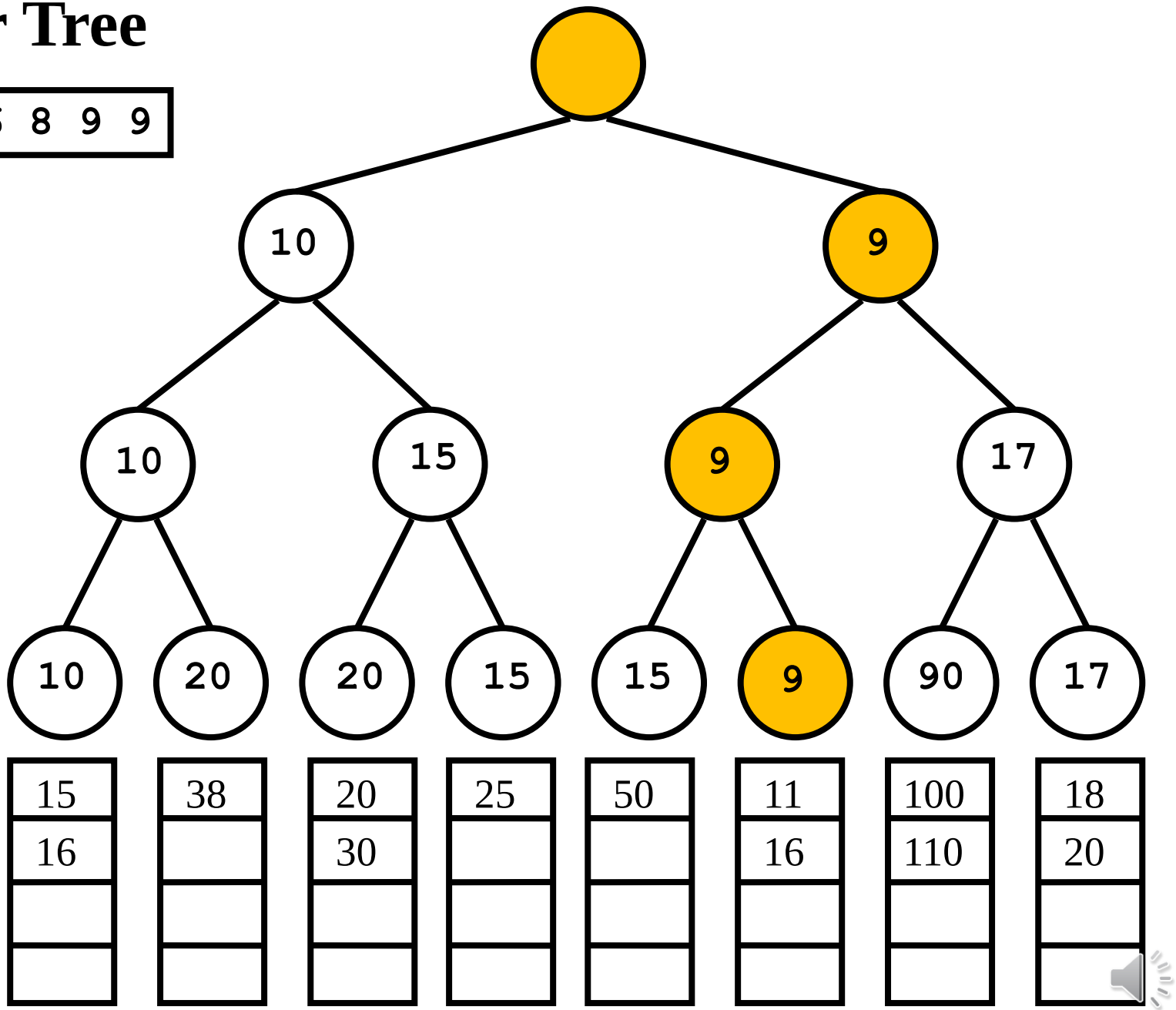
Winner Tree

```
Output: 6 8 9 9
```



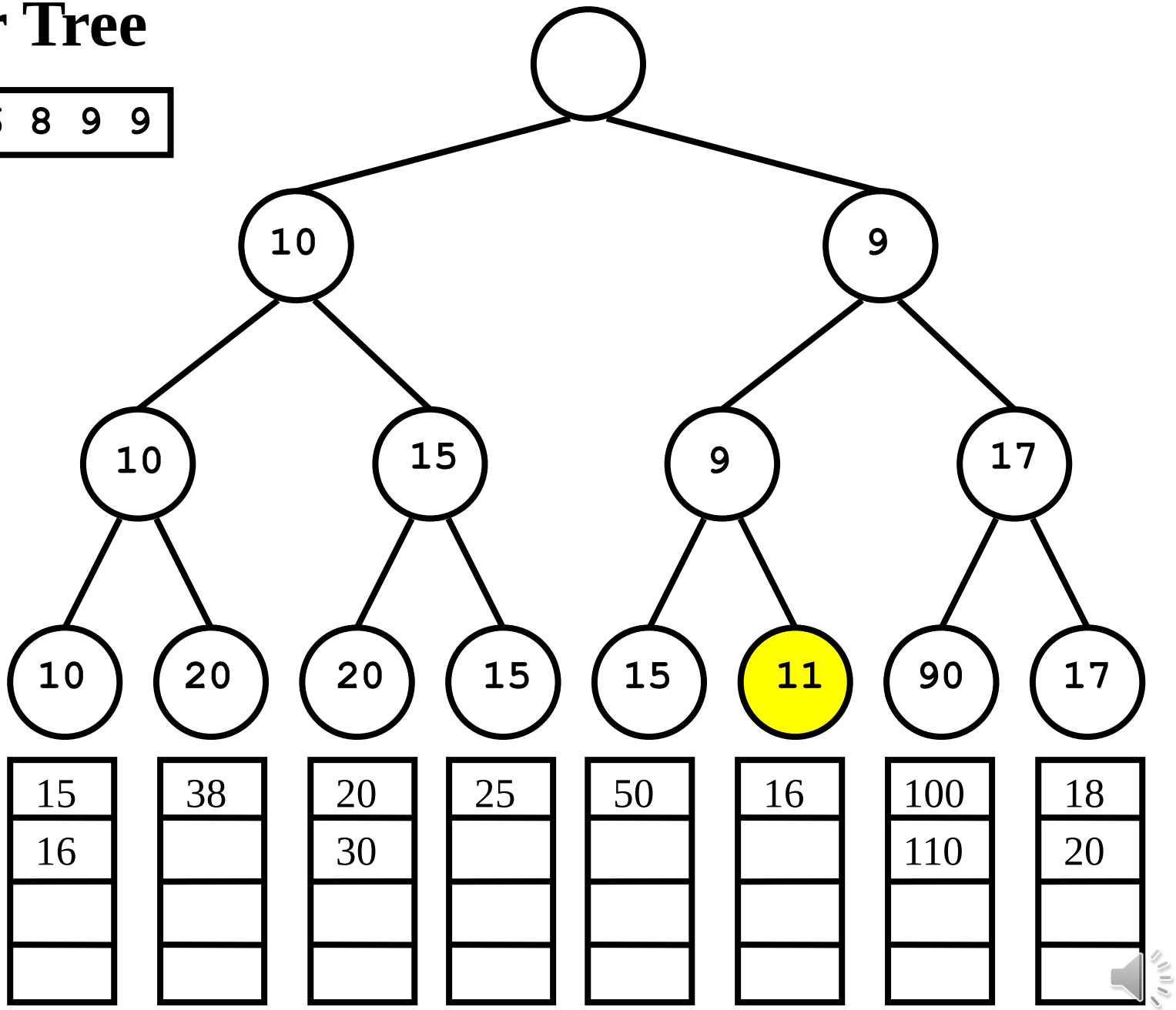
Winner Tree

```
Output: 6 8 9 9
```



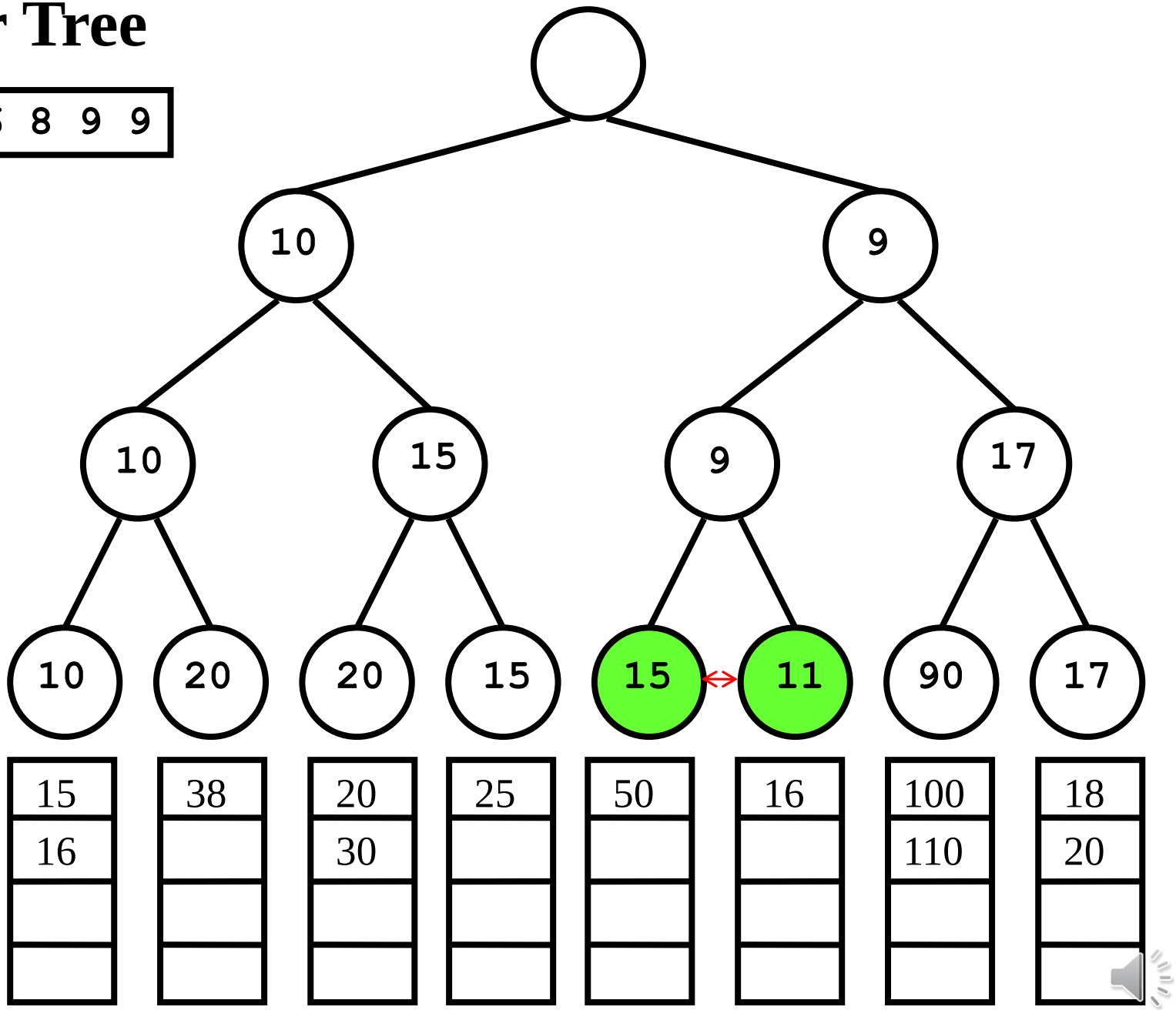
Winner Tree

```
Output: 6 8 9 9
```



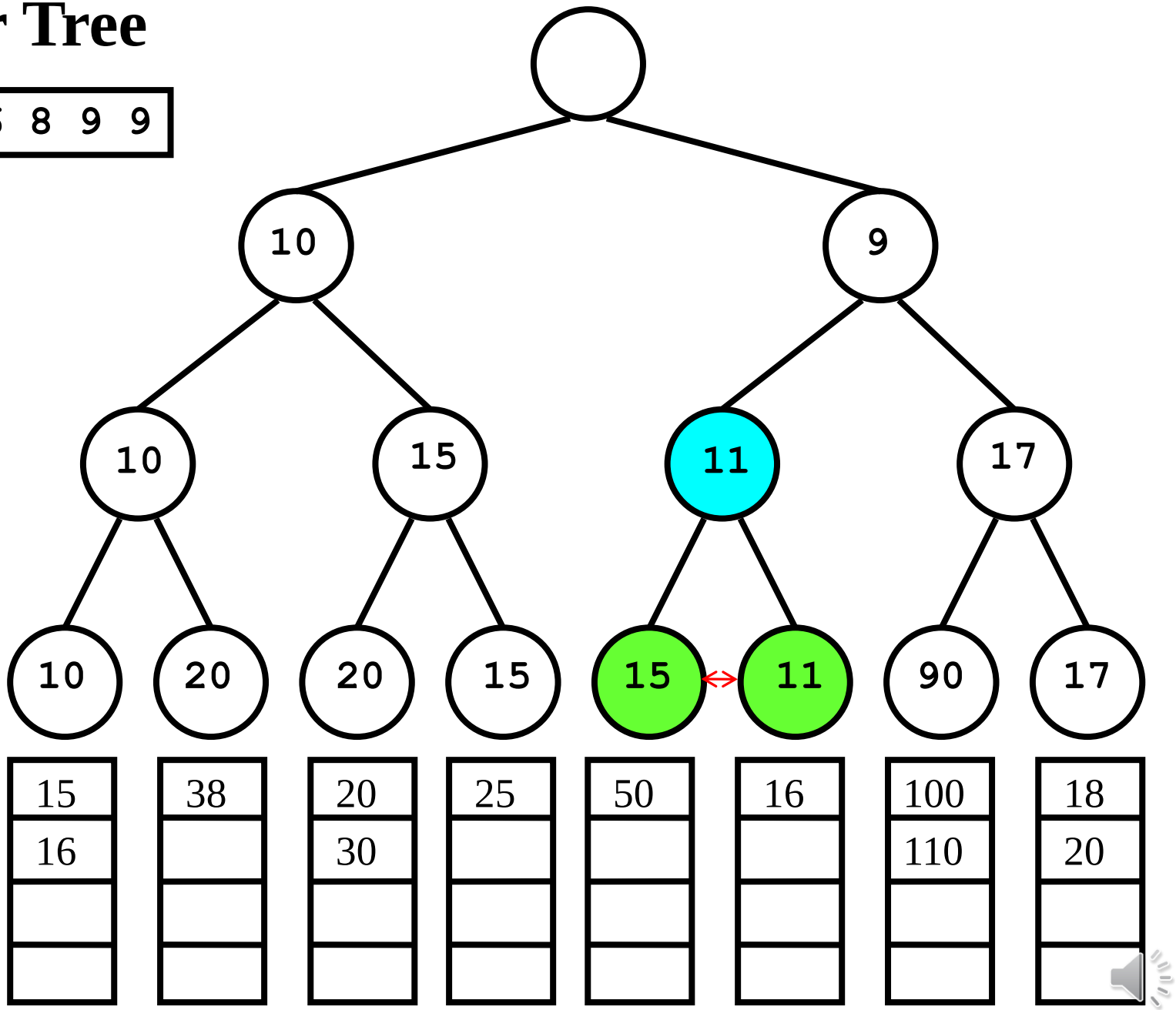
Winner Tree

```
Output: 6 8 9 9
```



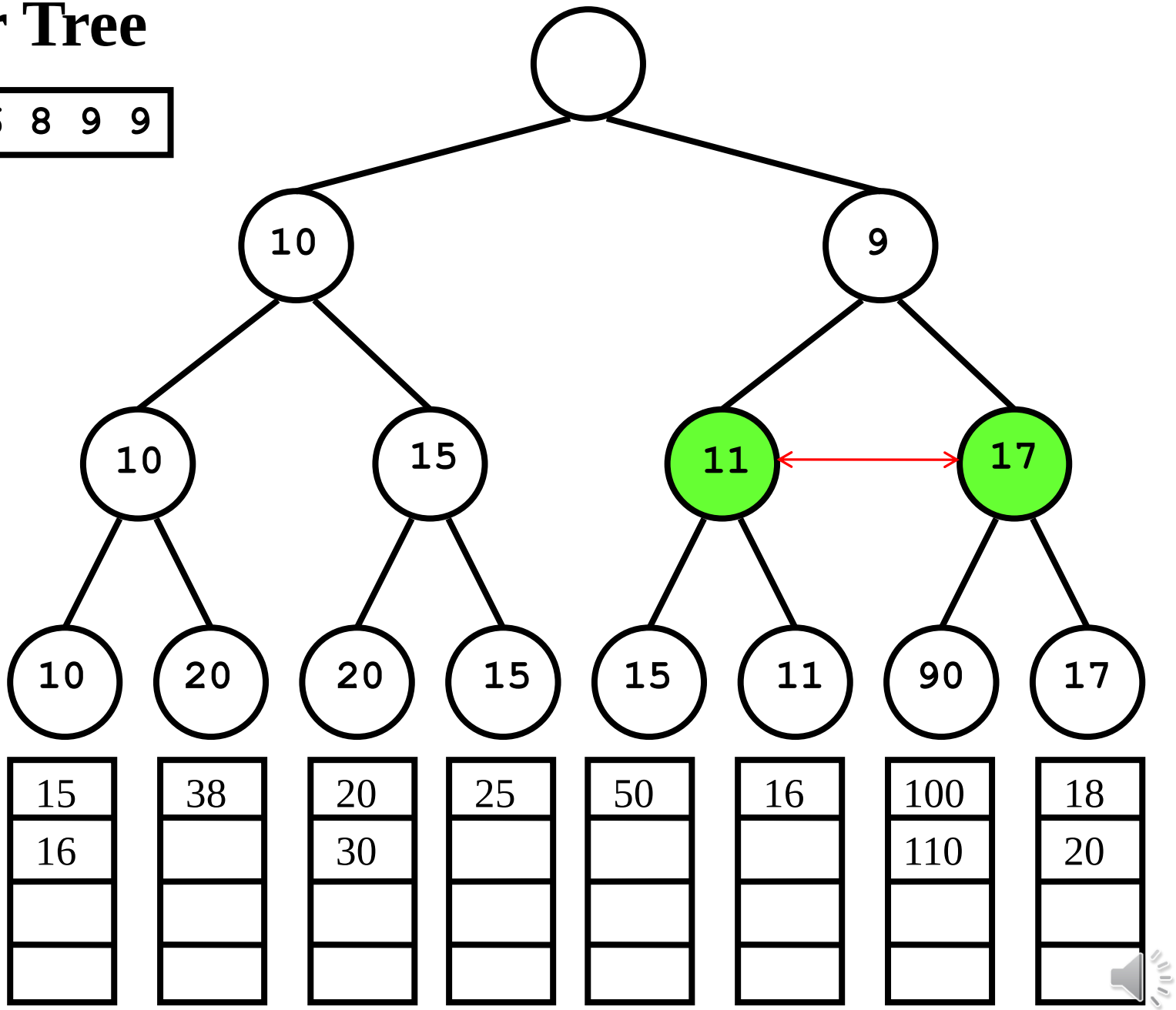
Winner Tree

```
Output: 6 8 9 9
```



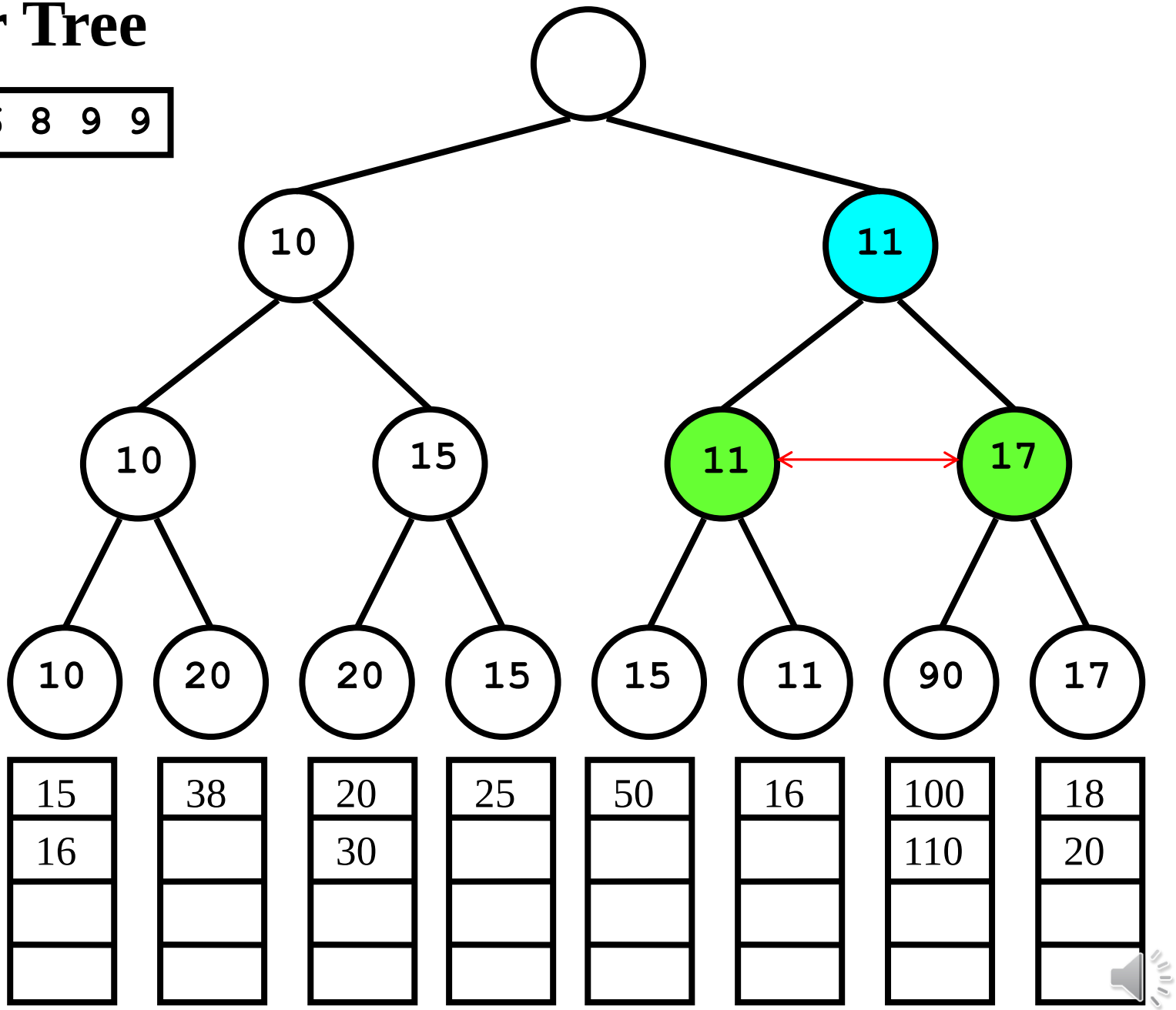
Winner Tree

```
Output: 6 8 9 9
```



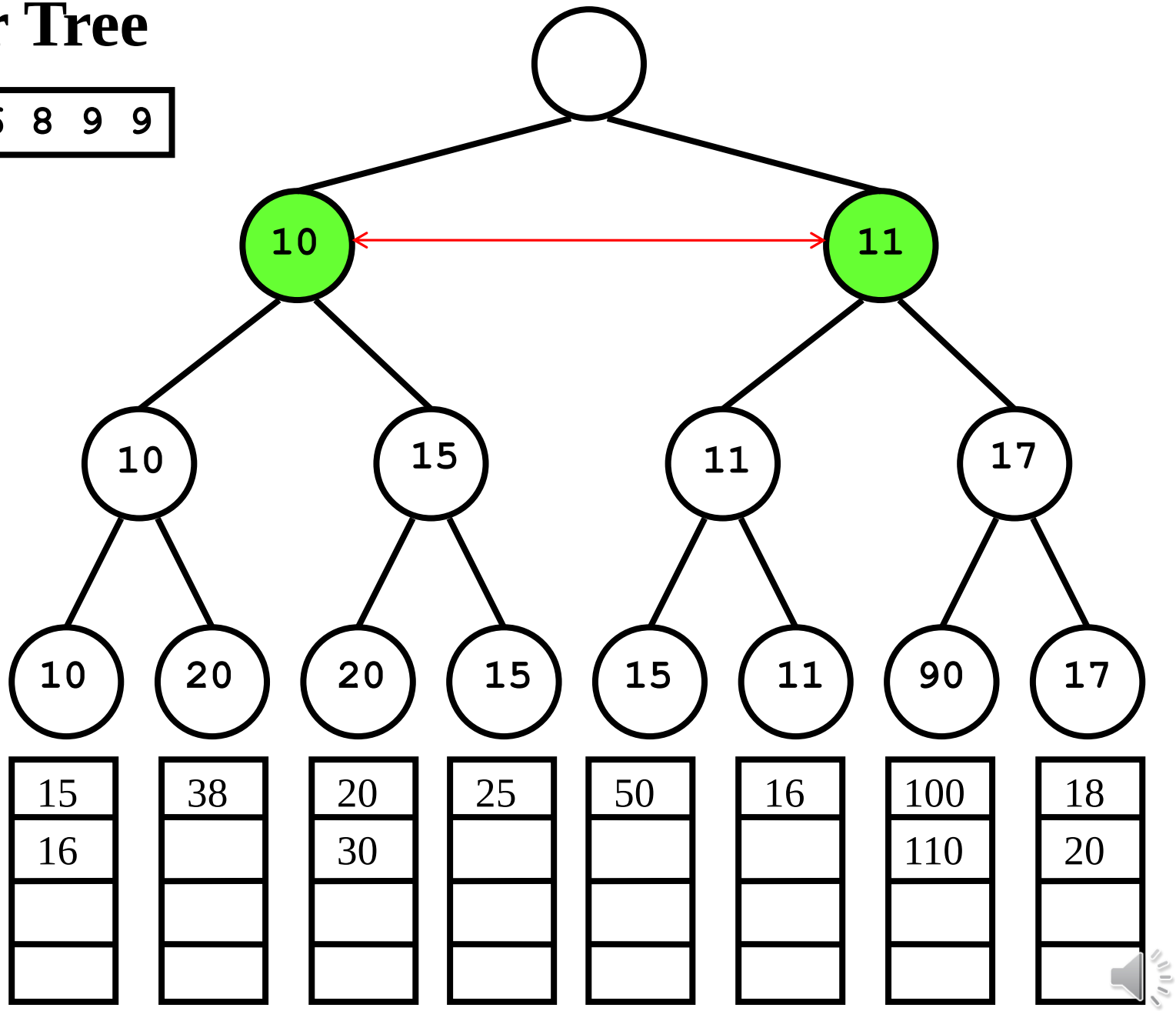
Winner Tree

```
Output: 6 8 9 9
```



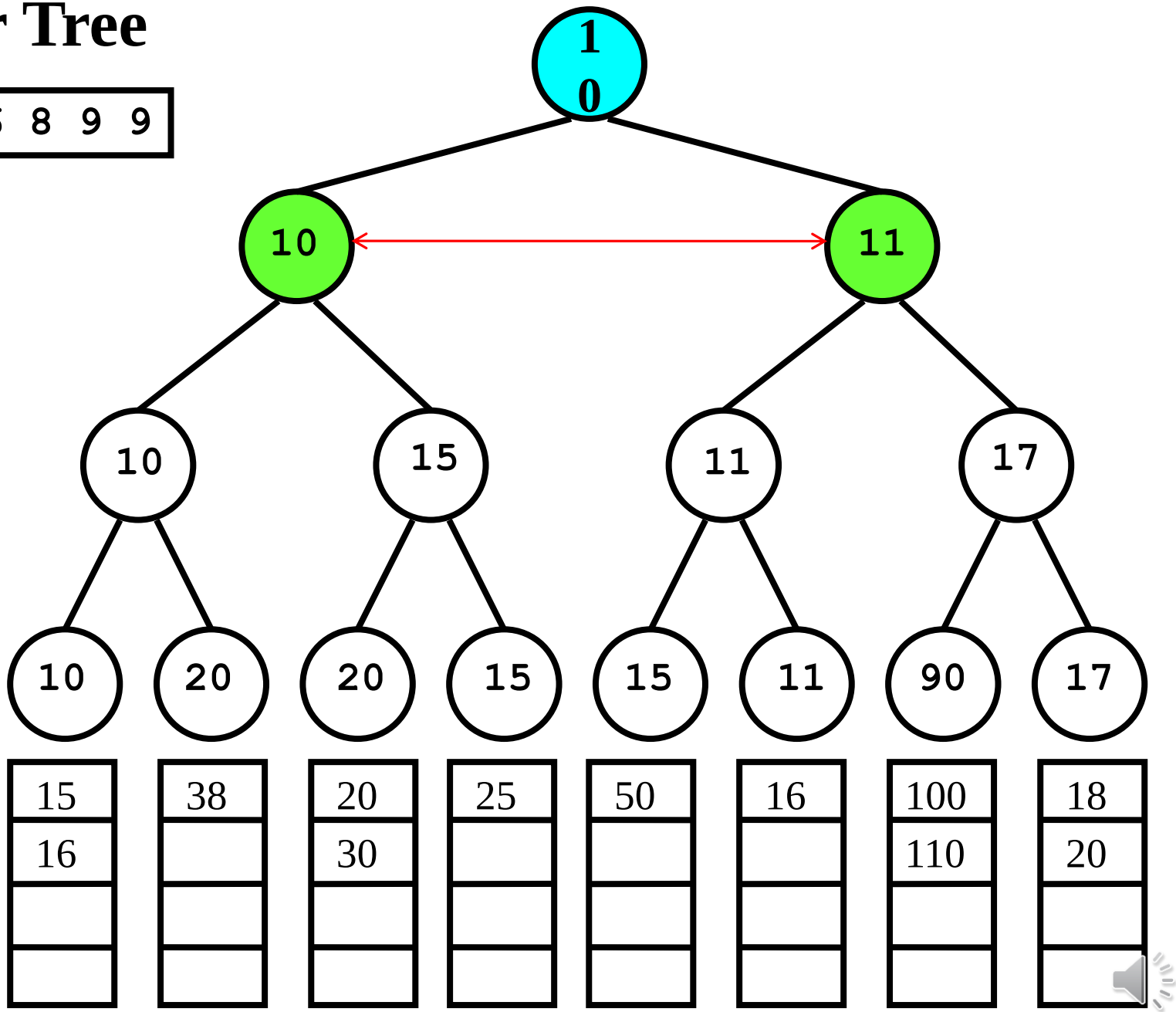
Winner Tree

```
Output: 6 8 9 9
```



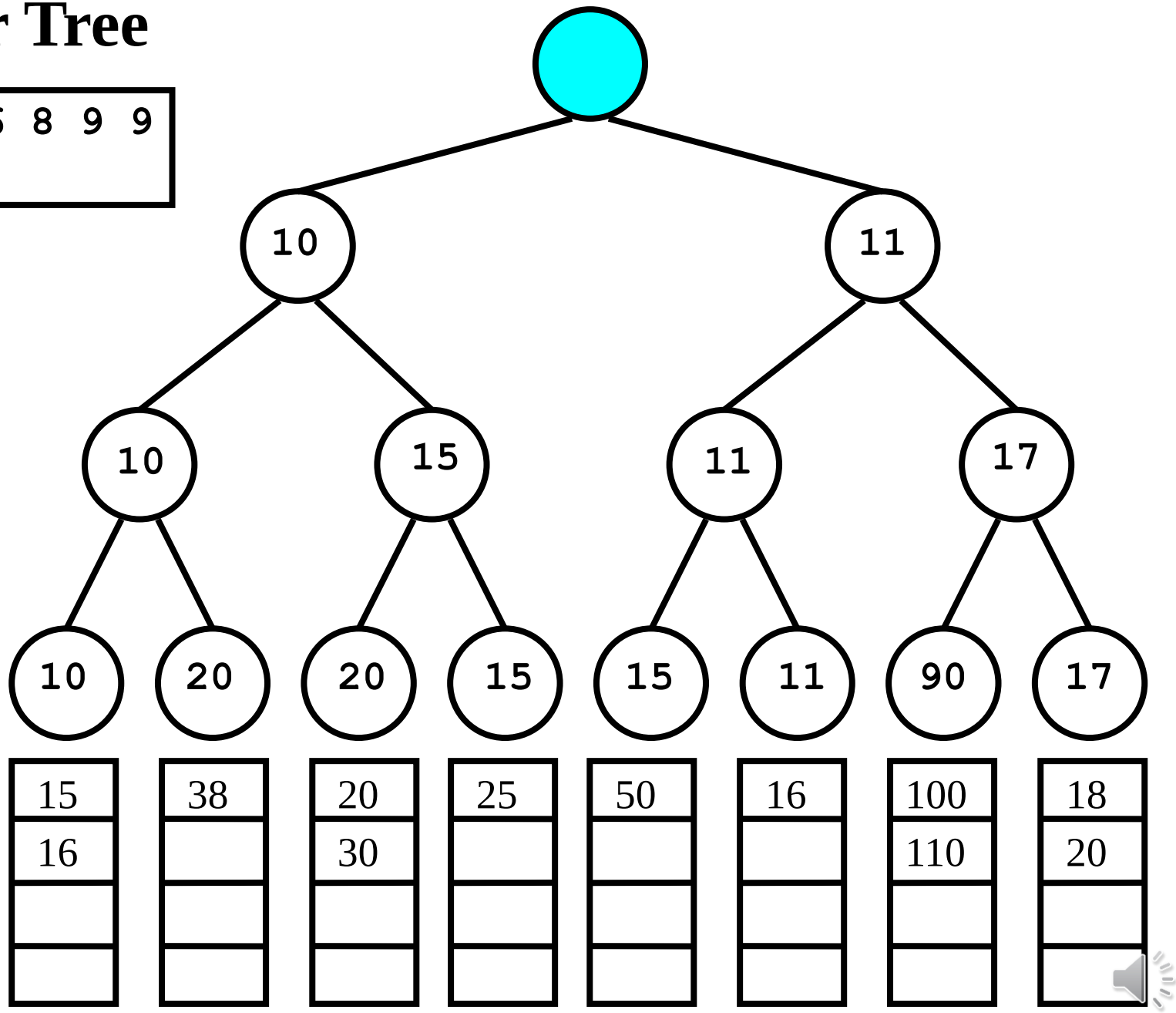
Winner Tree

```
Output: 6 8 9 9
```



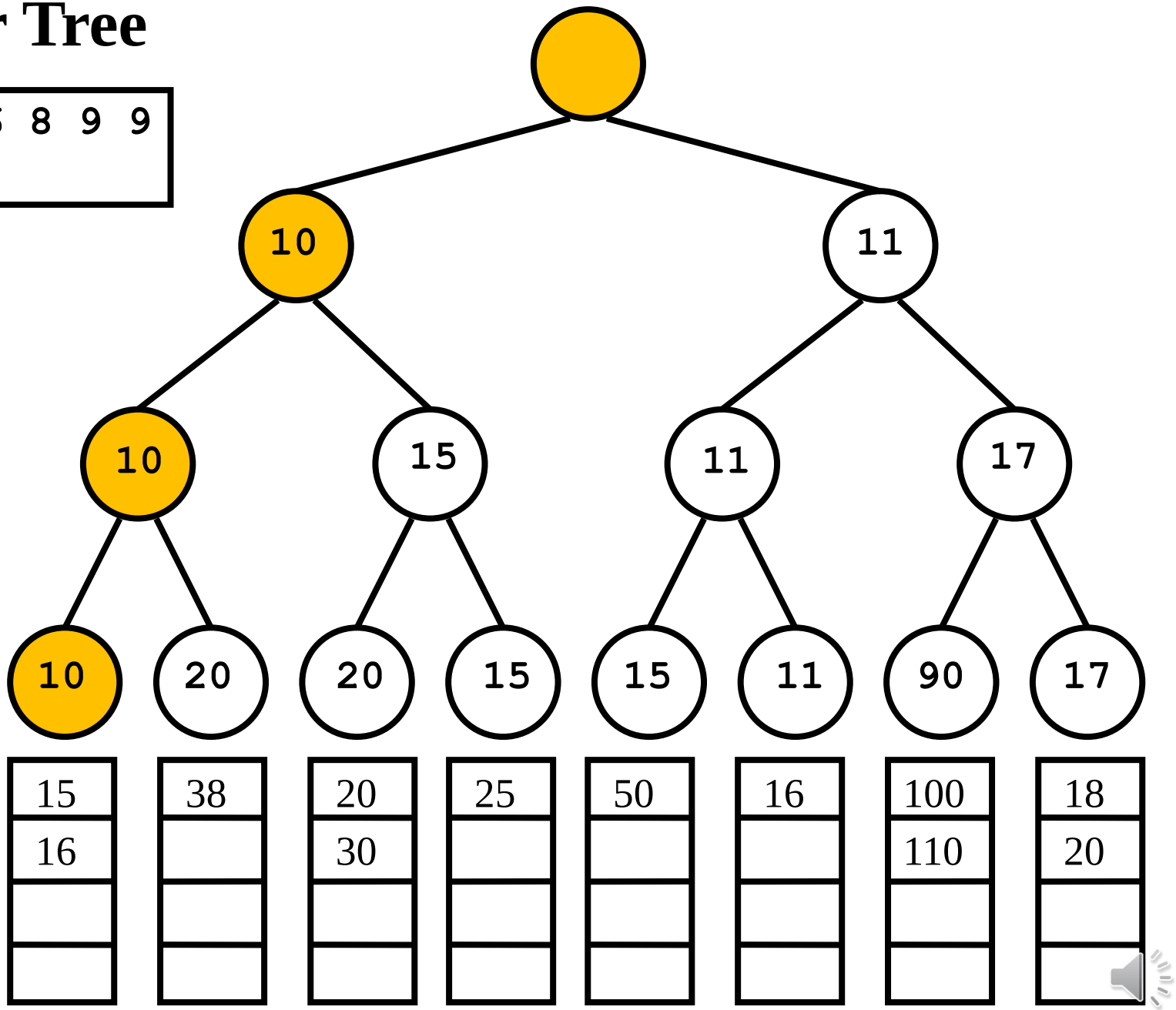
Winner Tree

```
Output: 6 8 9 9
    10
```



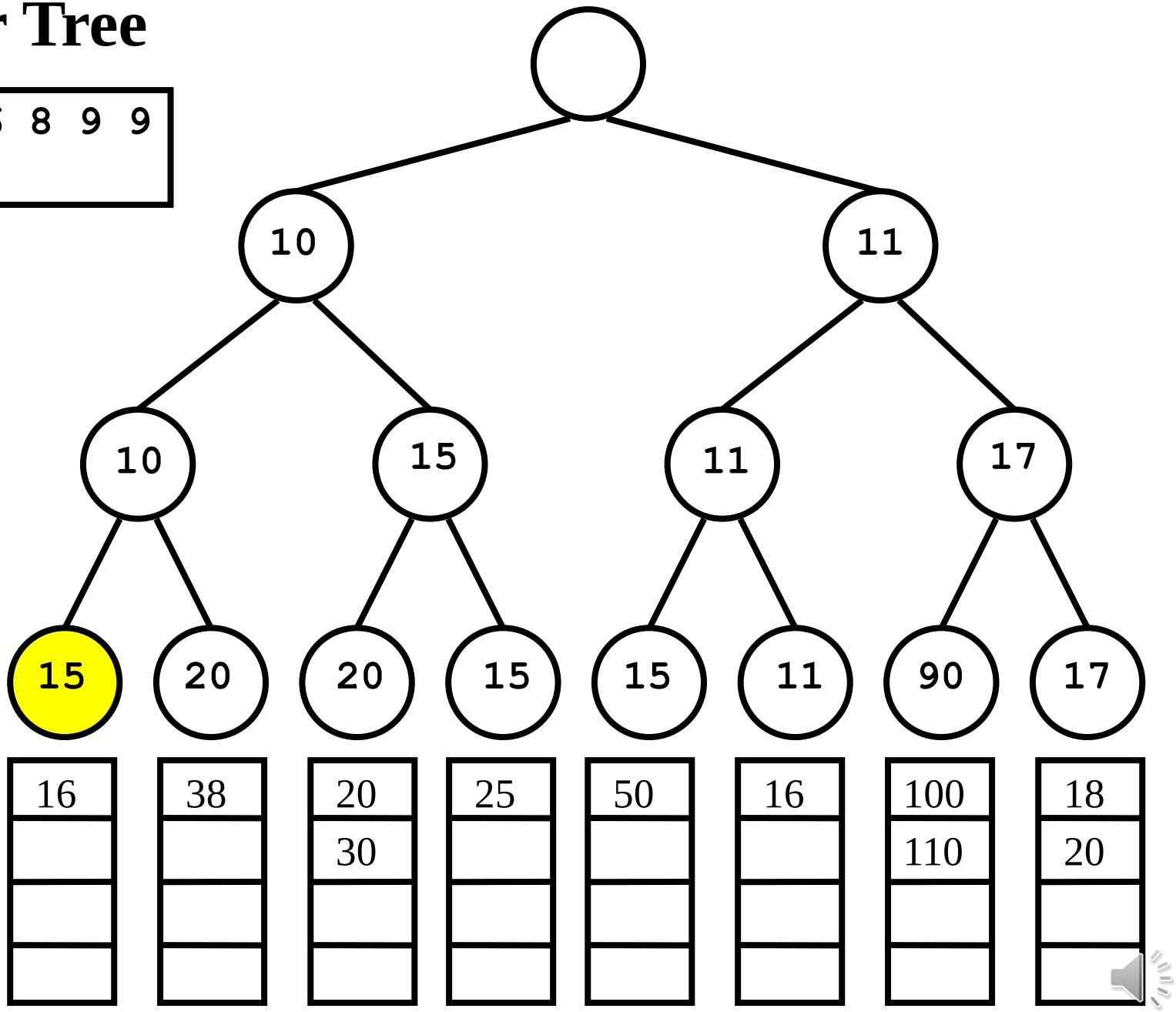
Winner Tree

```
Output: 6 8 9 9
    10
```



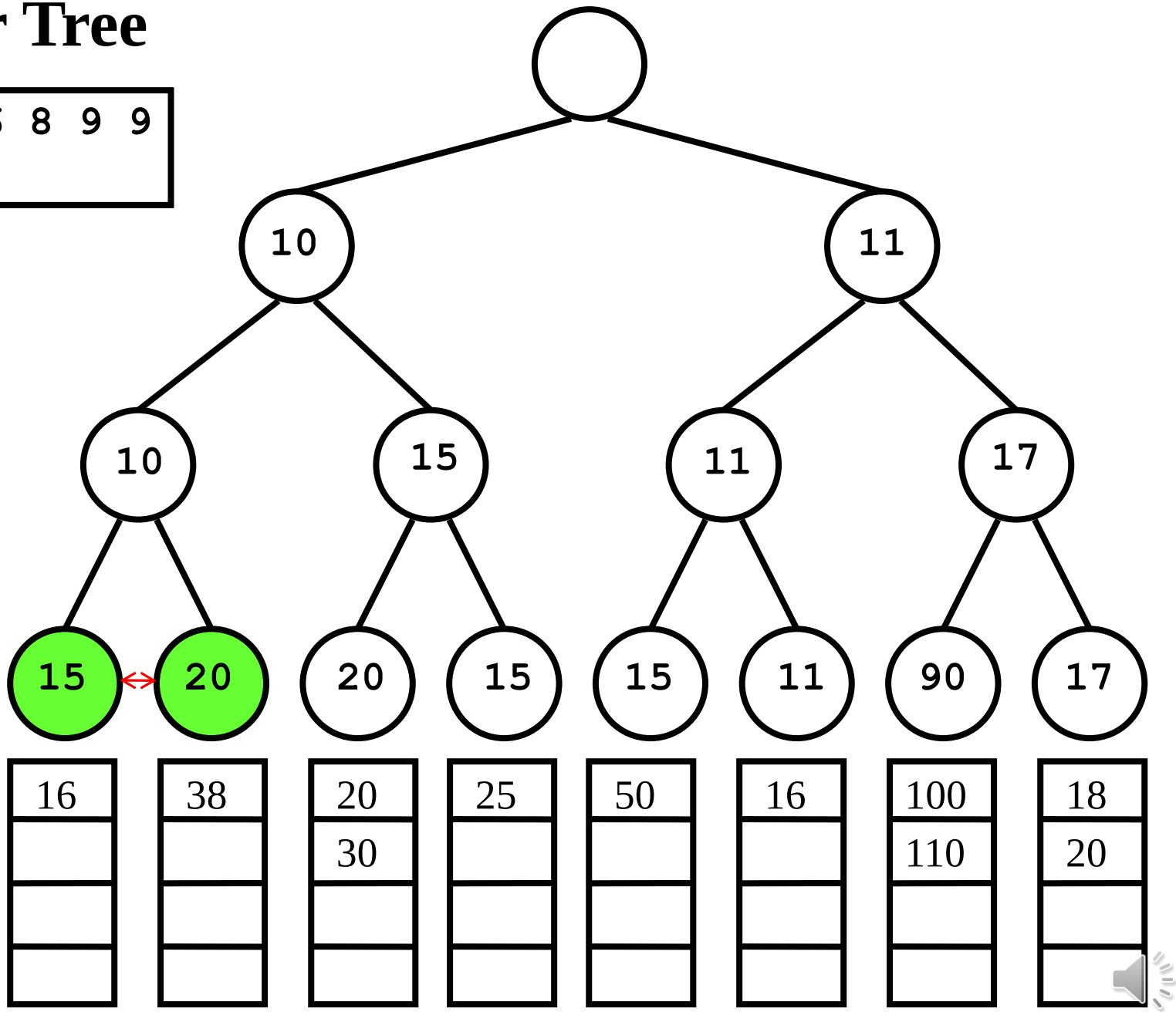
Winner Tree

```
Output: 6 8 9 9
    10
```



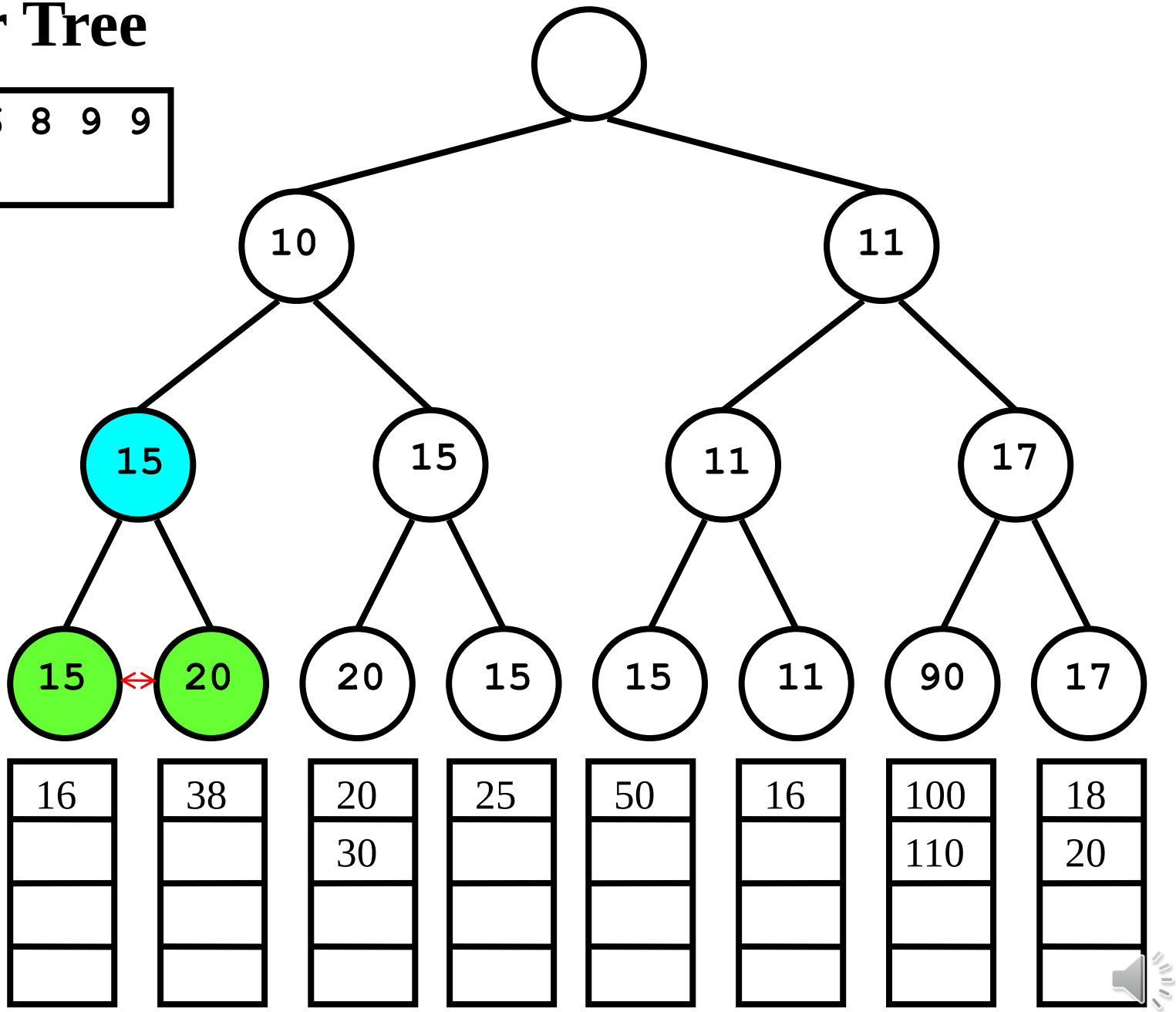
Winner Tree

```
Output: 6 8 9 9
    10
```



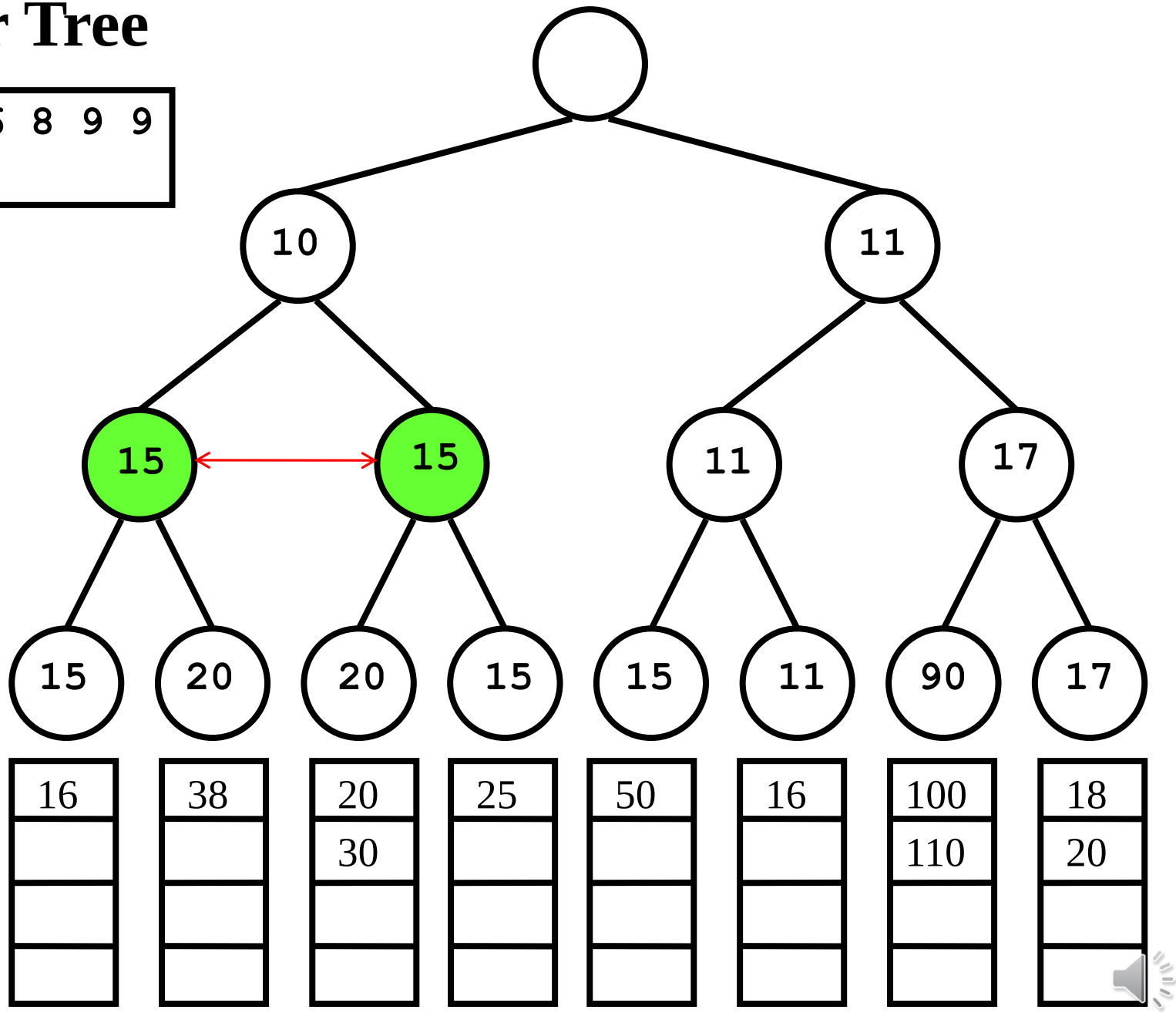
Winner Tree

```
Output: 6 8 9 9
    10
```



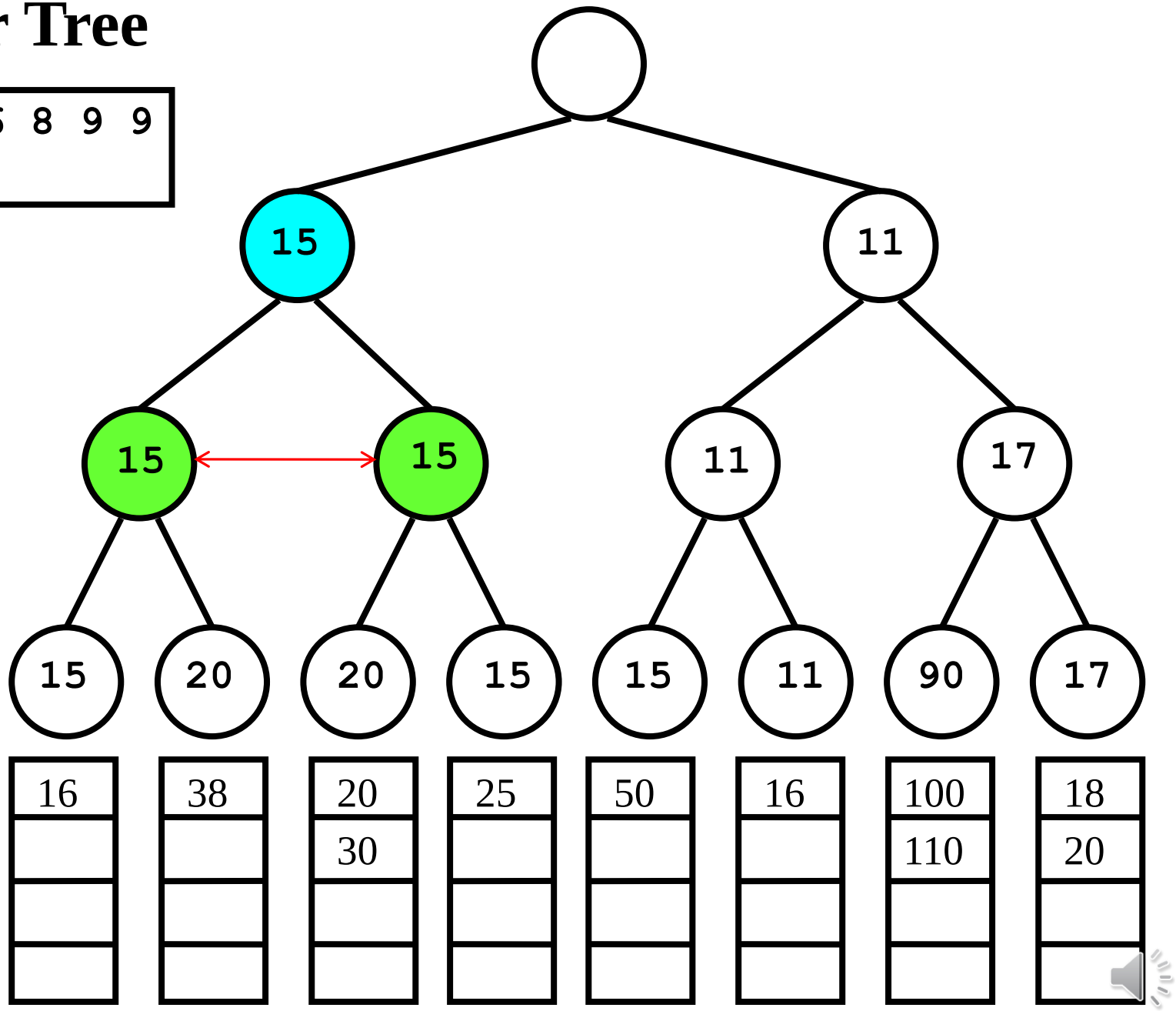
Winner Tree

```
Output: 6 8 9 9
    10
```



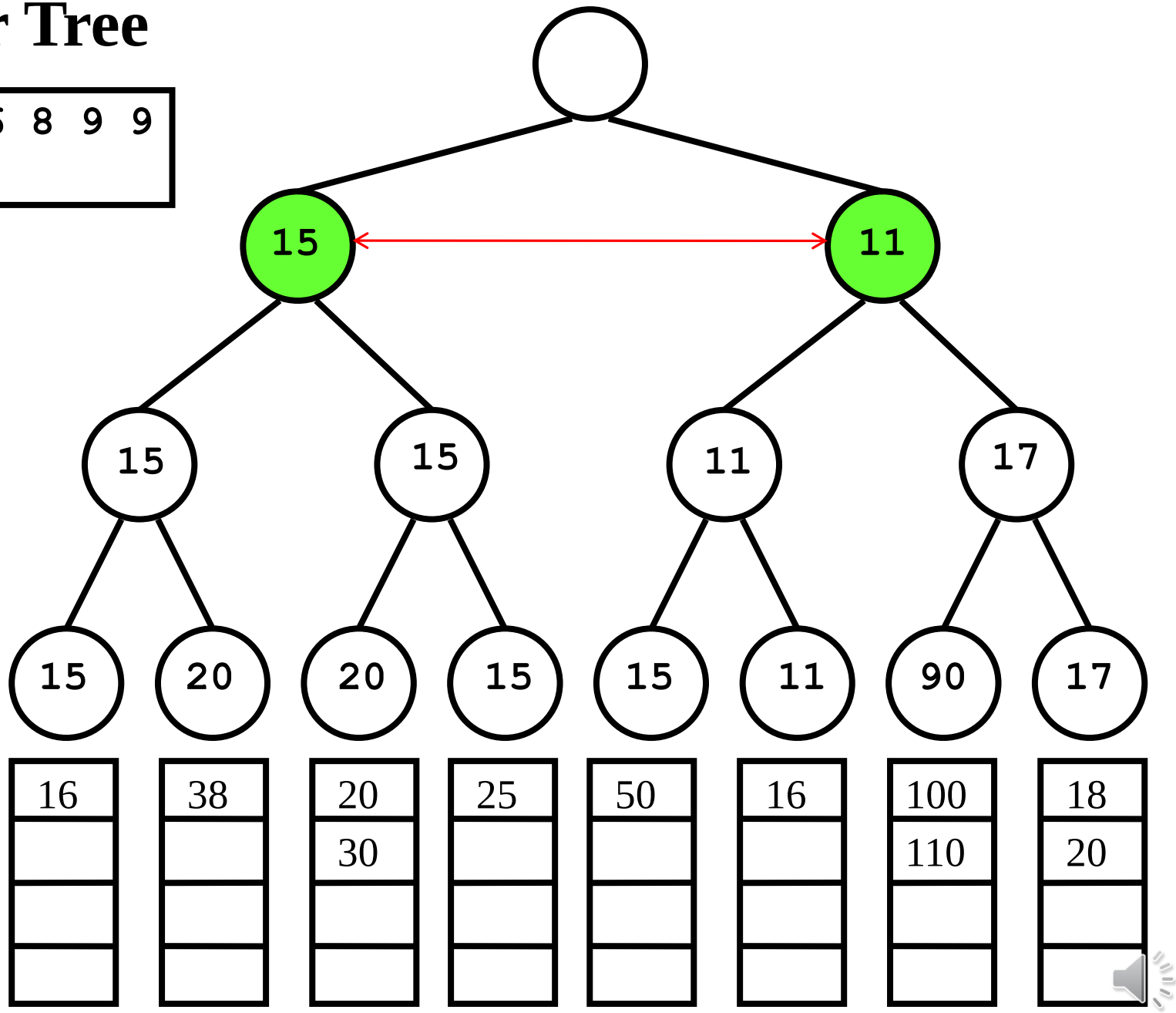
Winner Tree

```
Output: 6 8 9 9
    10
```



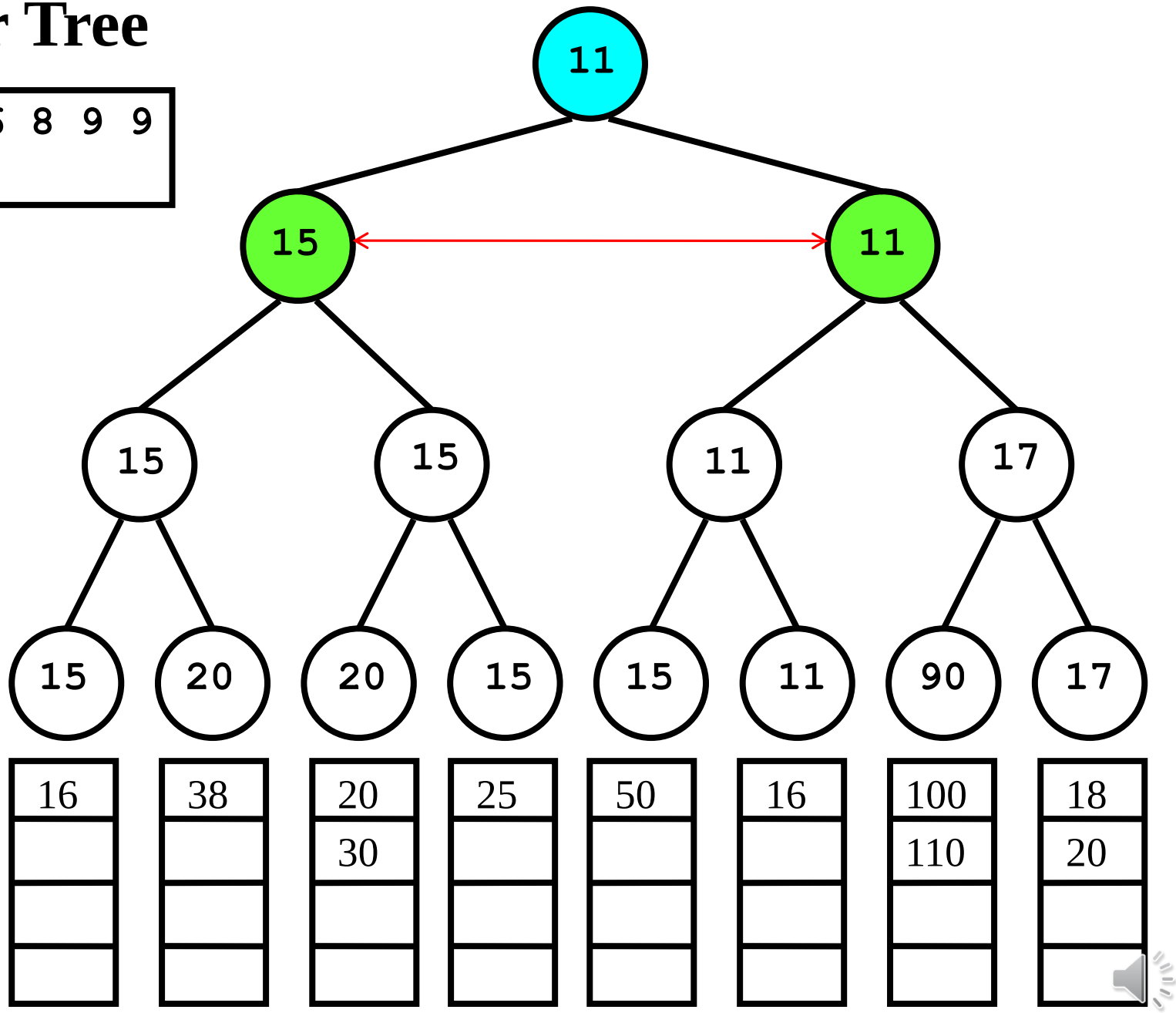
Winner Tree

```
Output: 6 8 9 9
    10
```



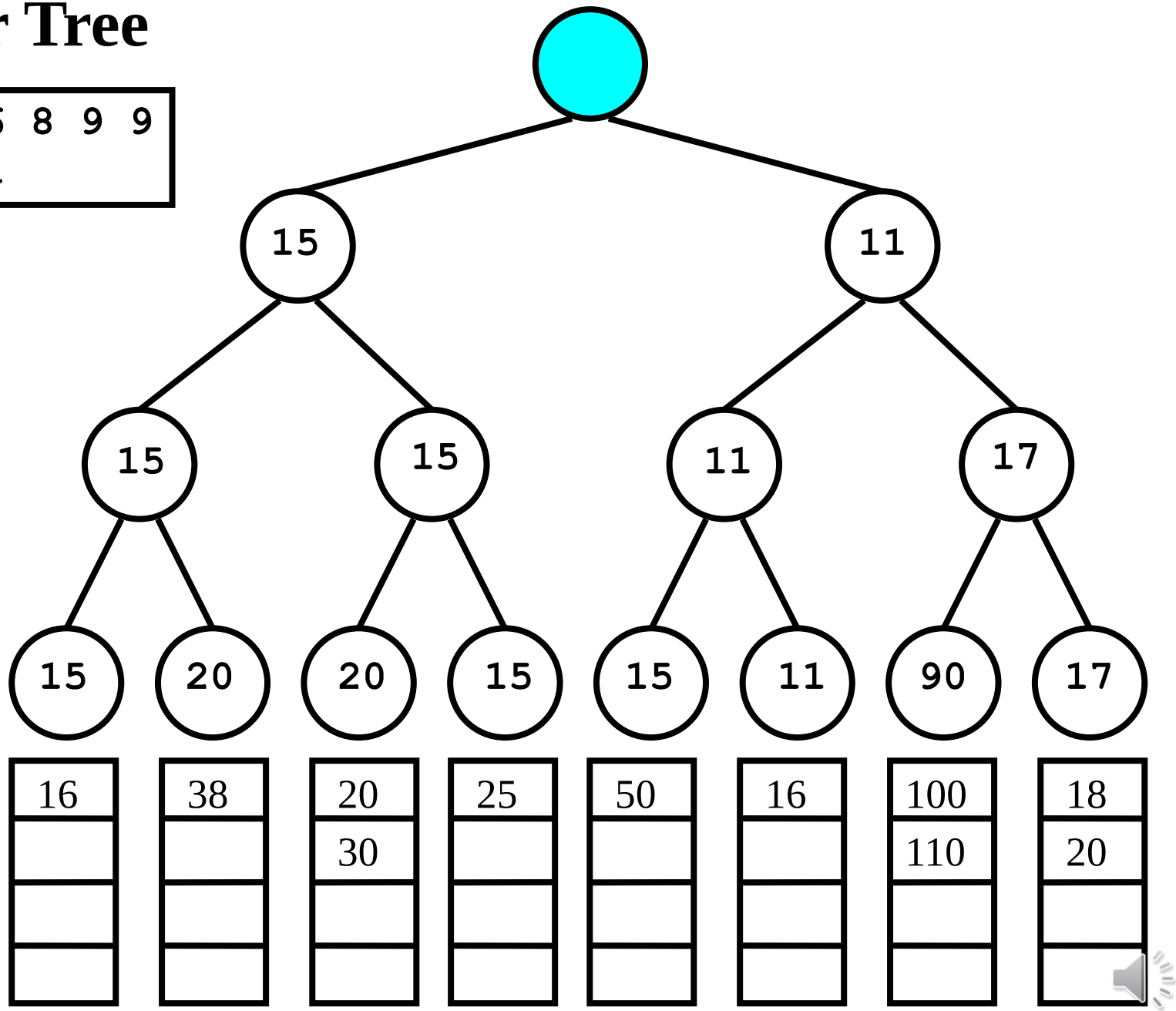
Winner Tree

```
Output: 6 8 9 9
    10
```



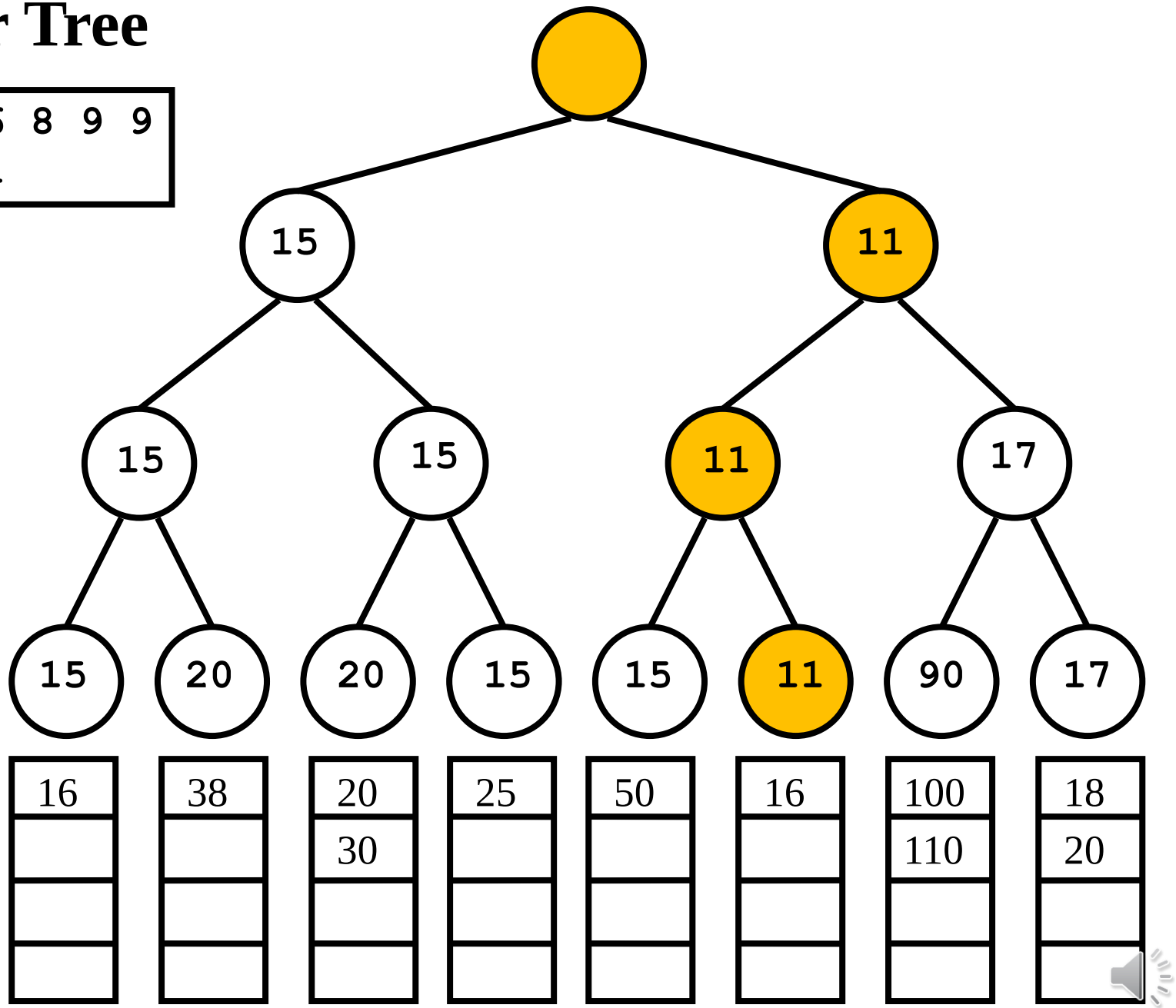
Winner Tree

```
Output: 6 8 9 9
    10 11
```



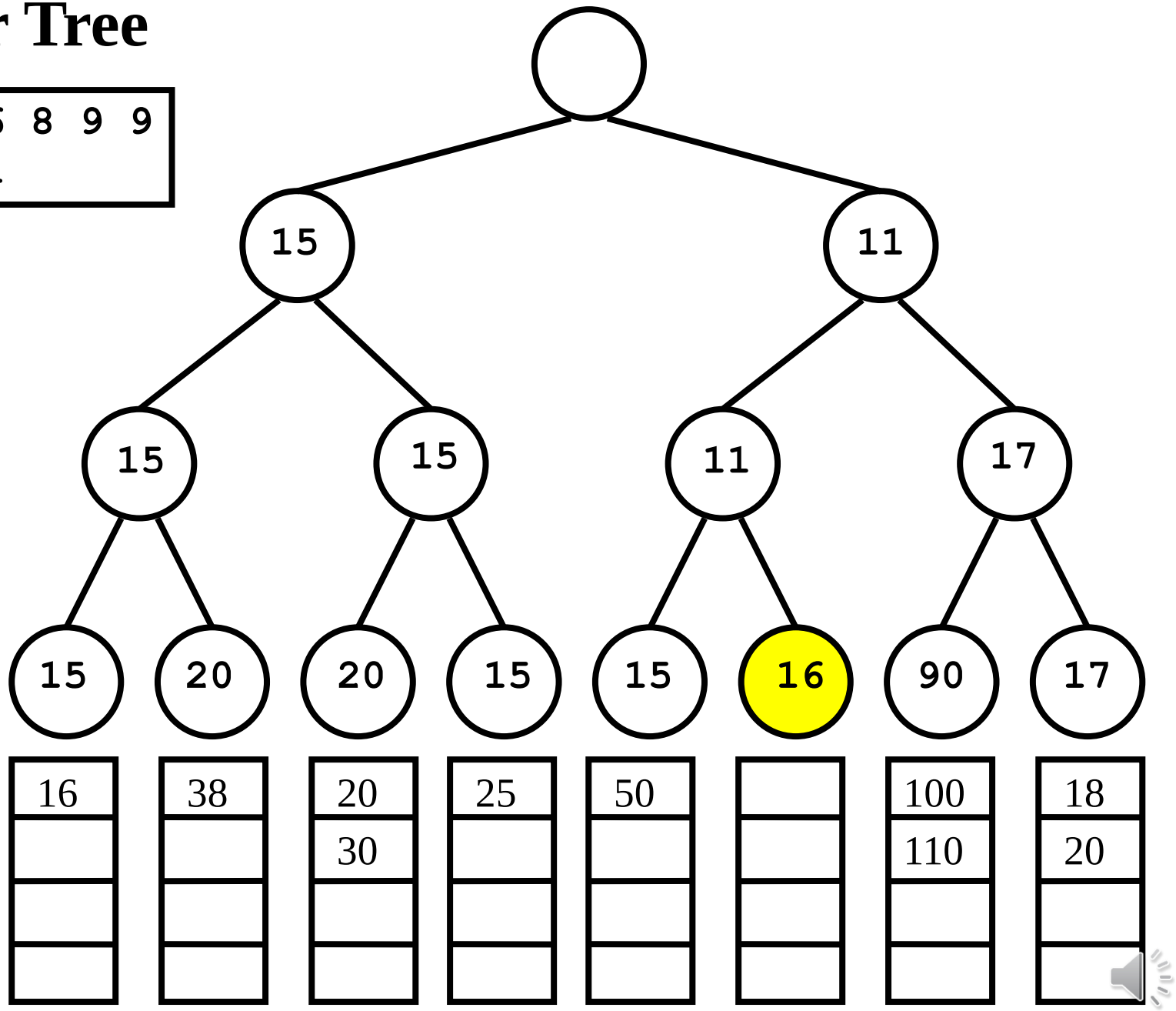
Winner Tree

```
Output: 6 8 9 9
    10 11
```



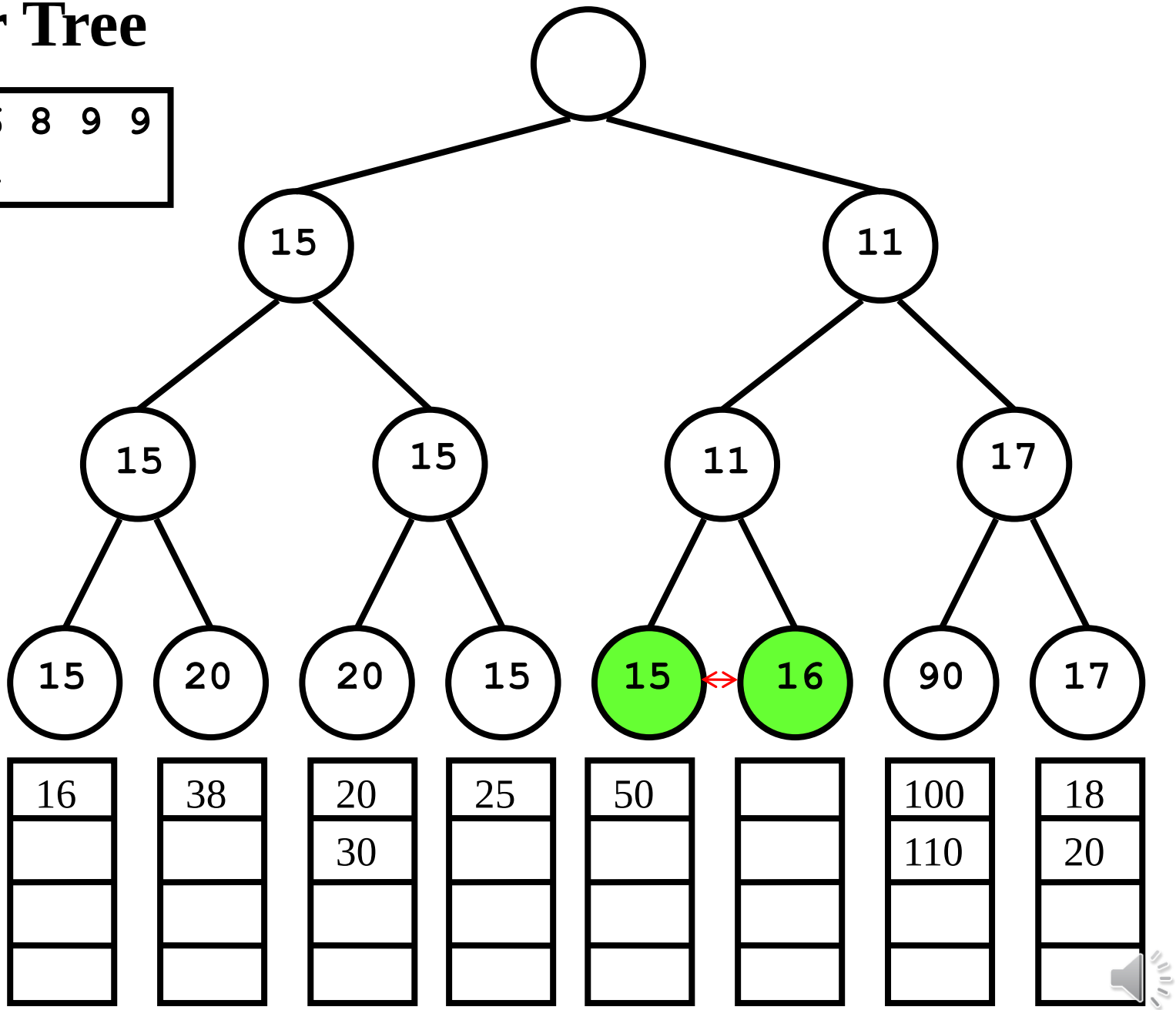
Winner Tree

```
Output: 6 8 9 9
    10 11
```



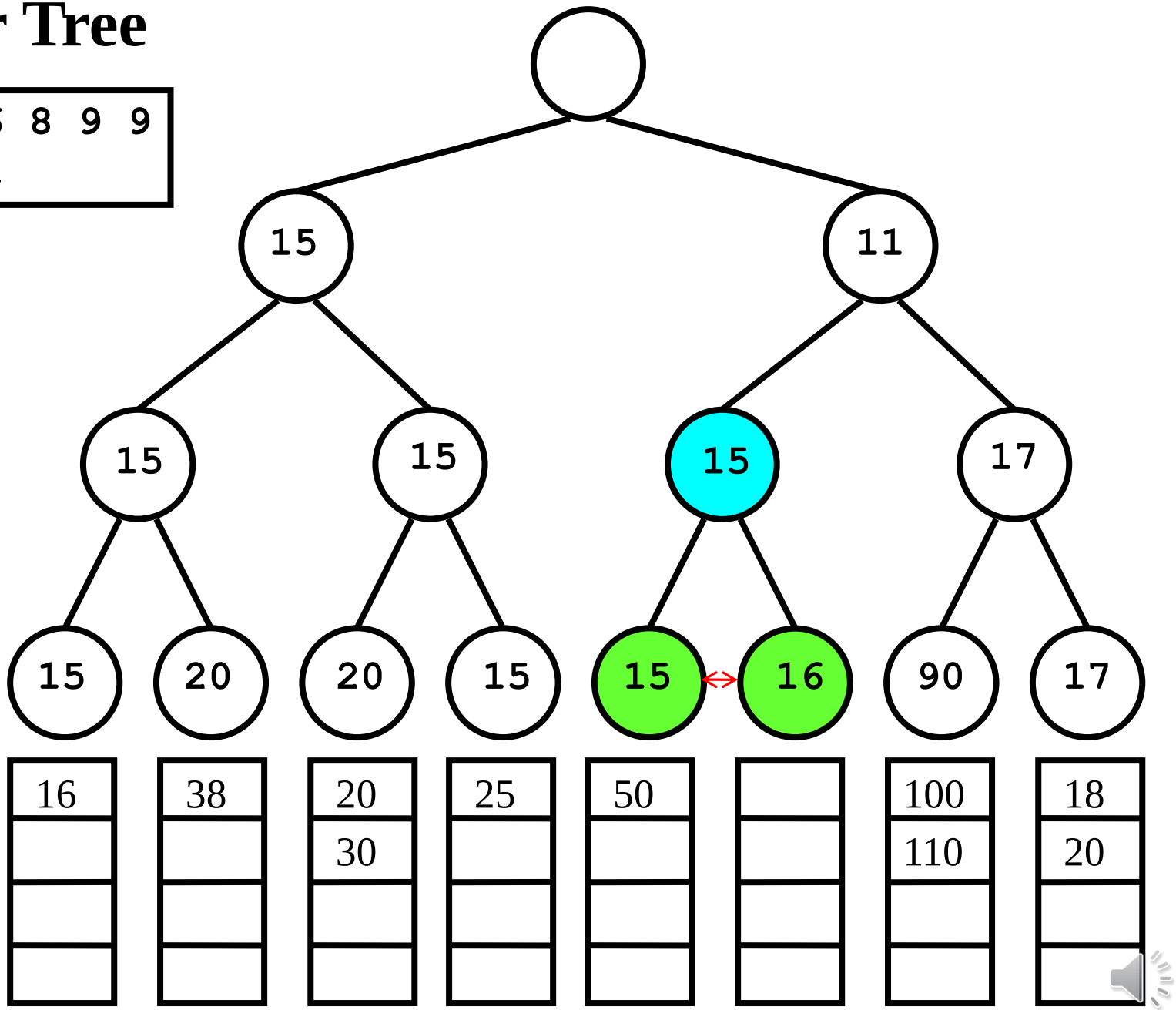
Winner Tree

```
Output: 6 8 9 9
    10 11
```



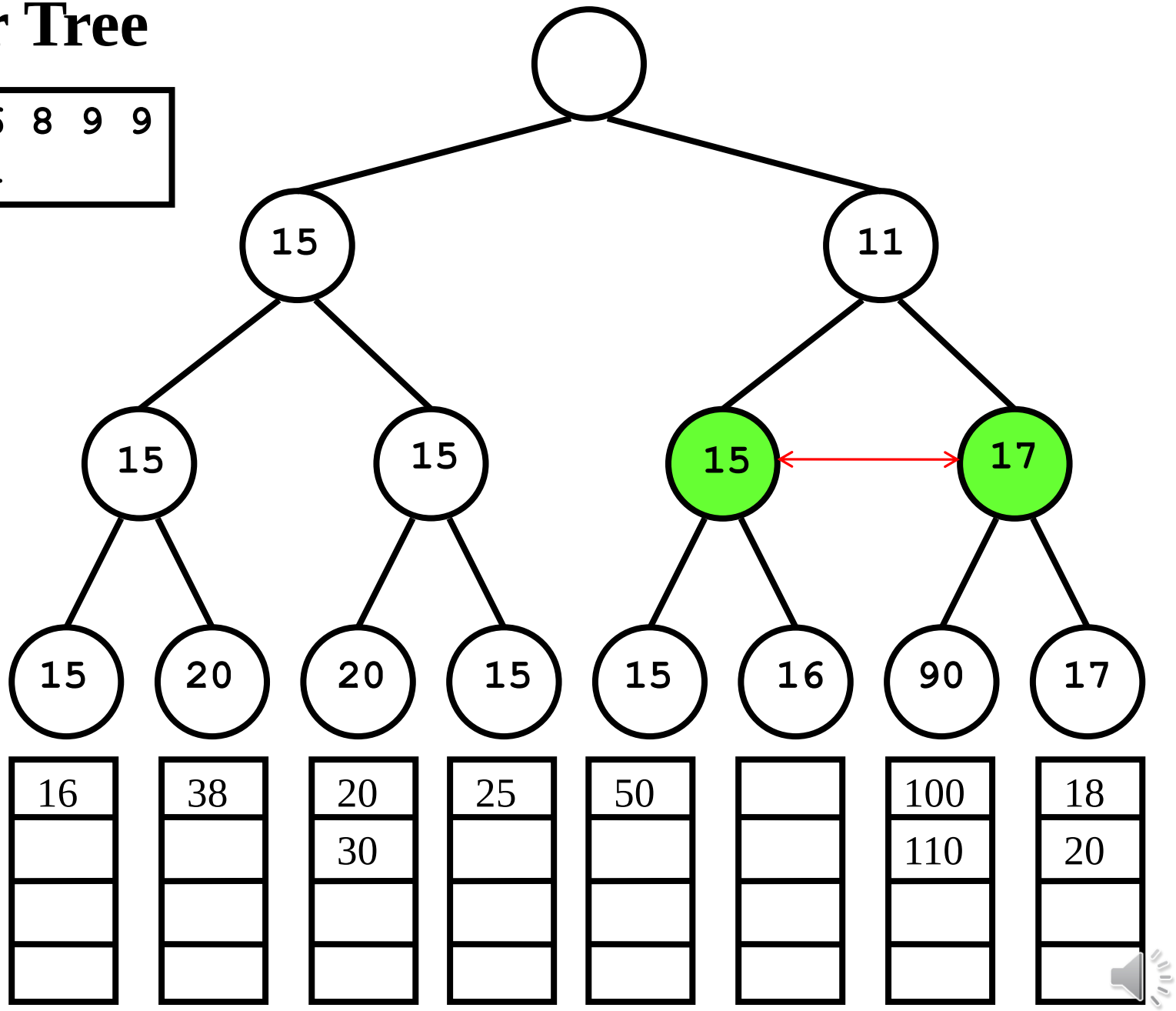
Winner Tree

```
Output: 6 8 9 9
    10 11
```



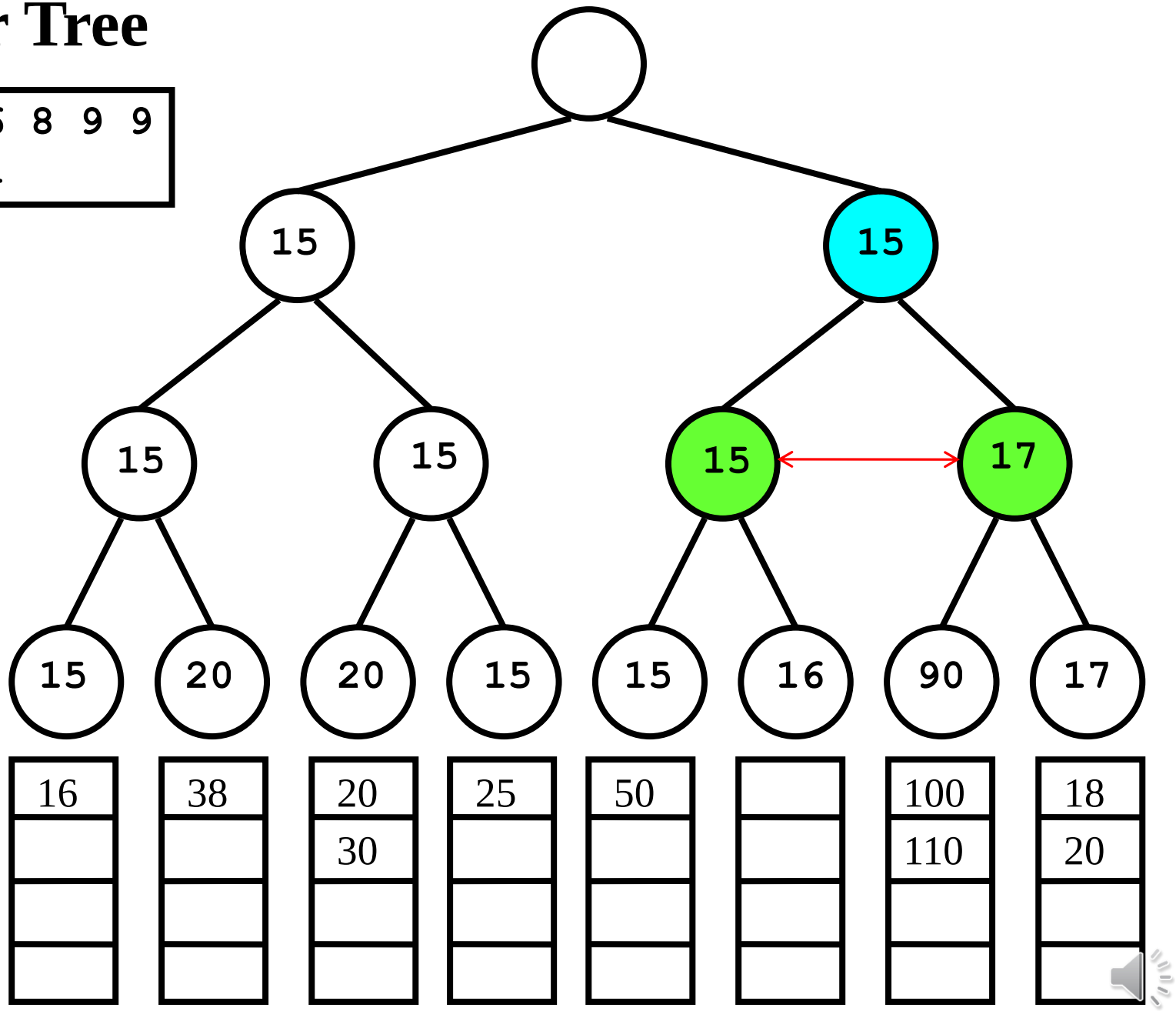
Winner Tree

```
Output: 6 8 9 9
    10 11
```



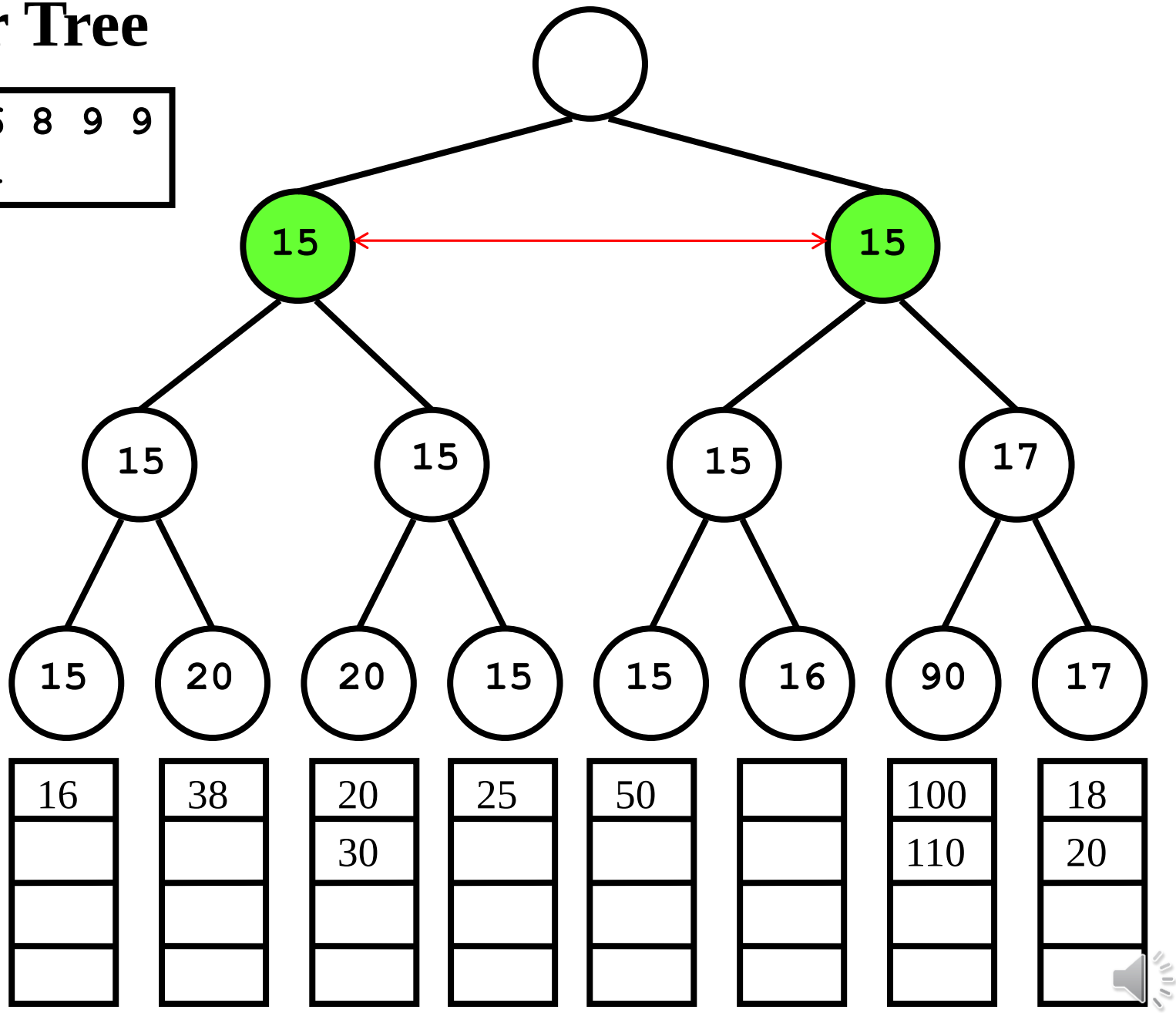
Winner Tree

```
Output: 6 8 9 9
    10 11
```



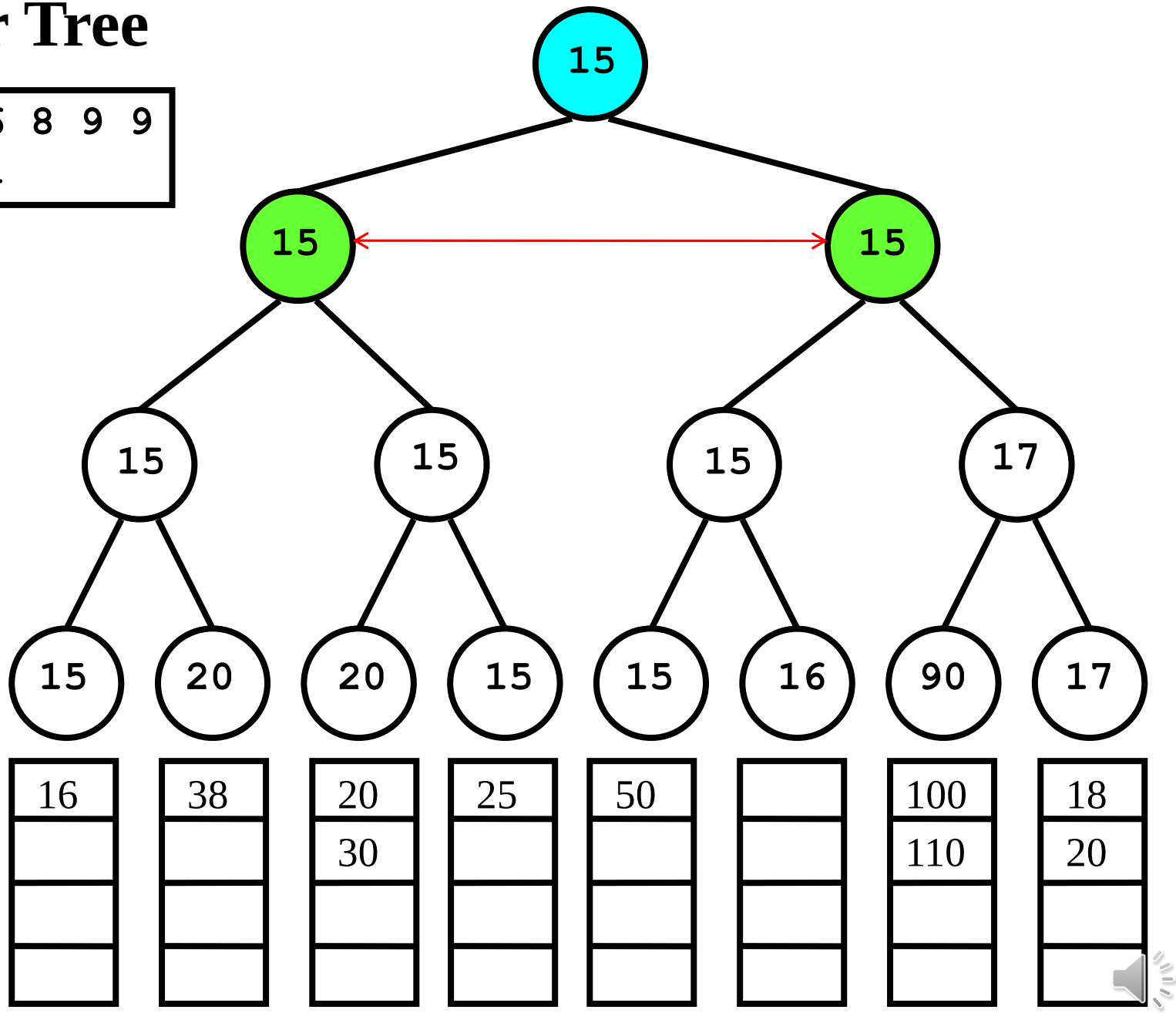
Winner Tree

```
Output: 6 8 9 9
    10 11
```



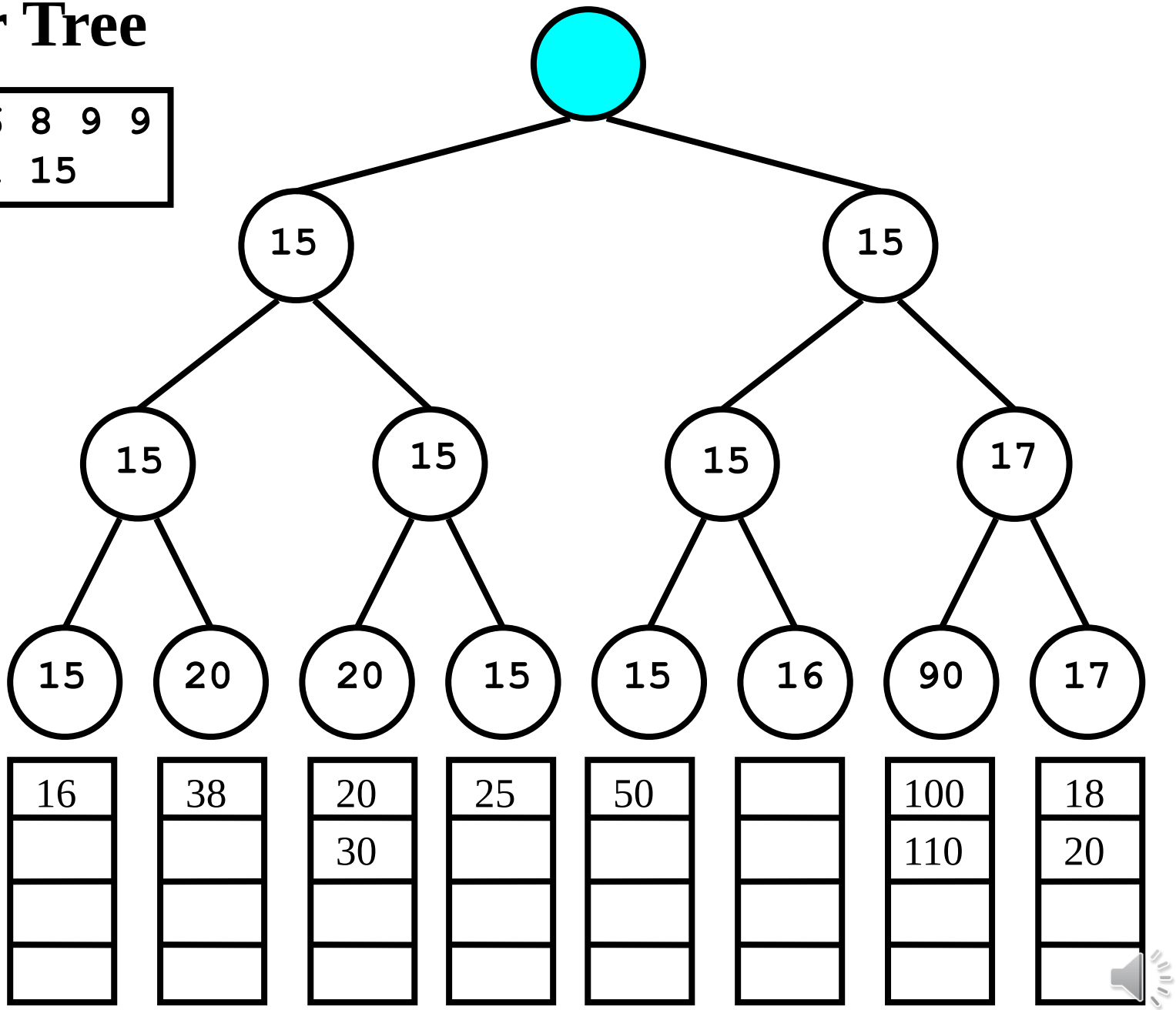
Winner Tree

```
Output: 6 8 9 9
    10 11
```



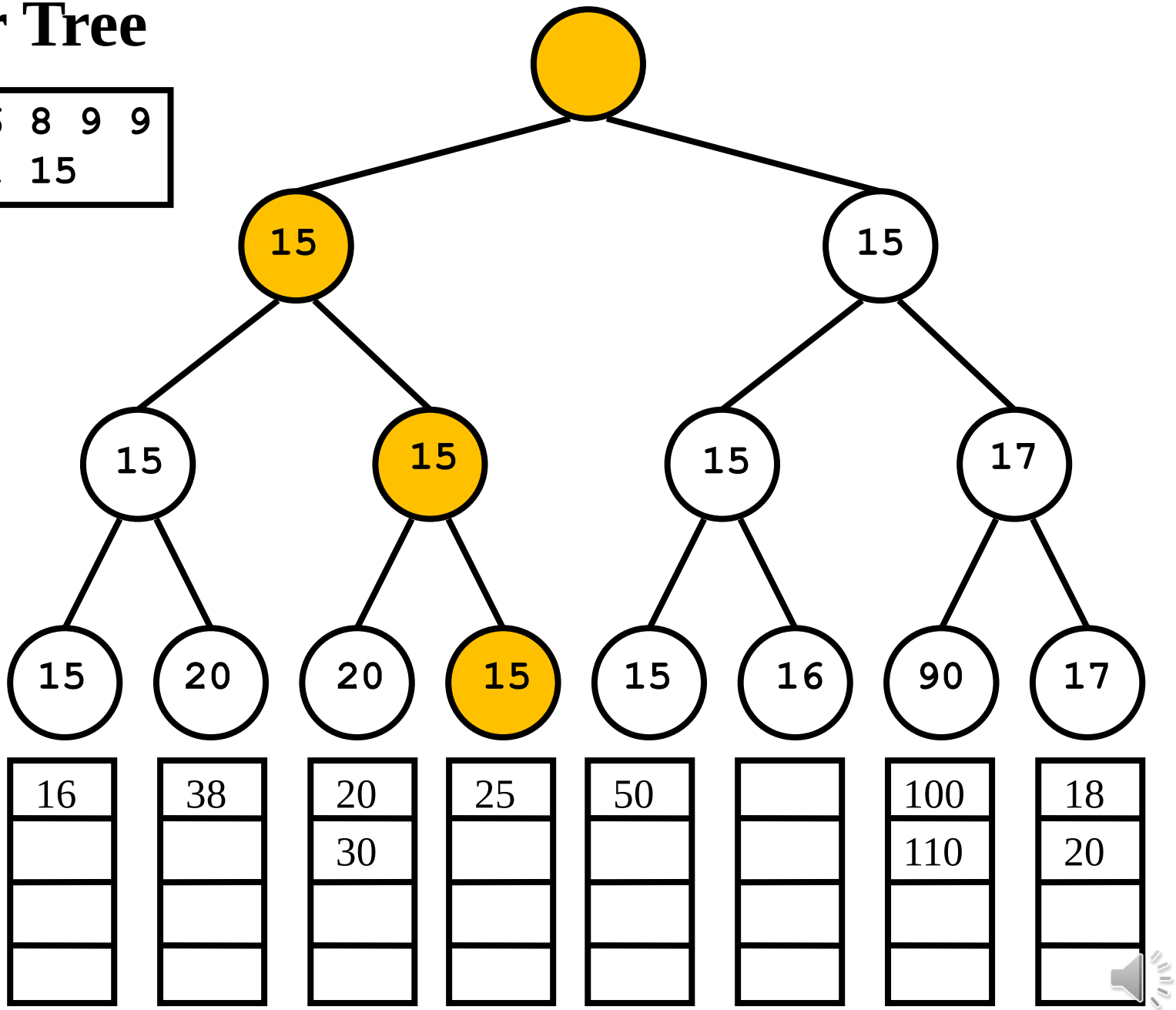
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



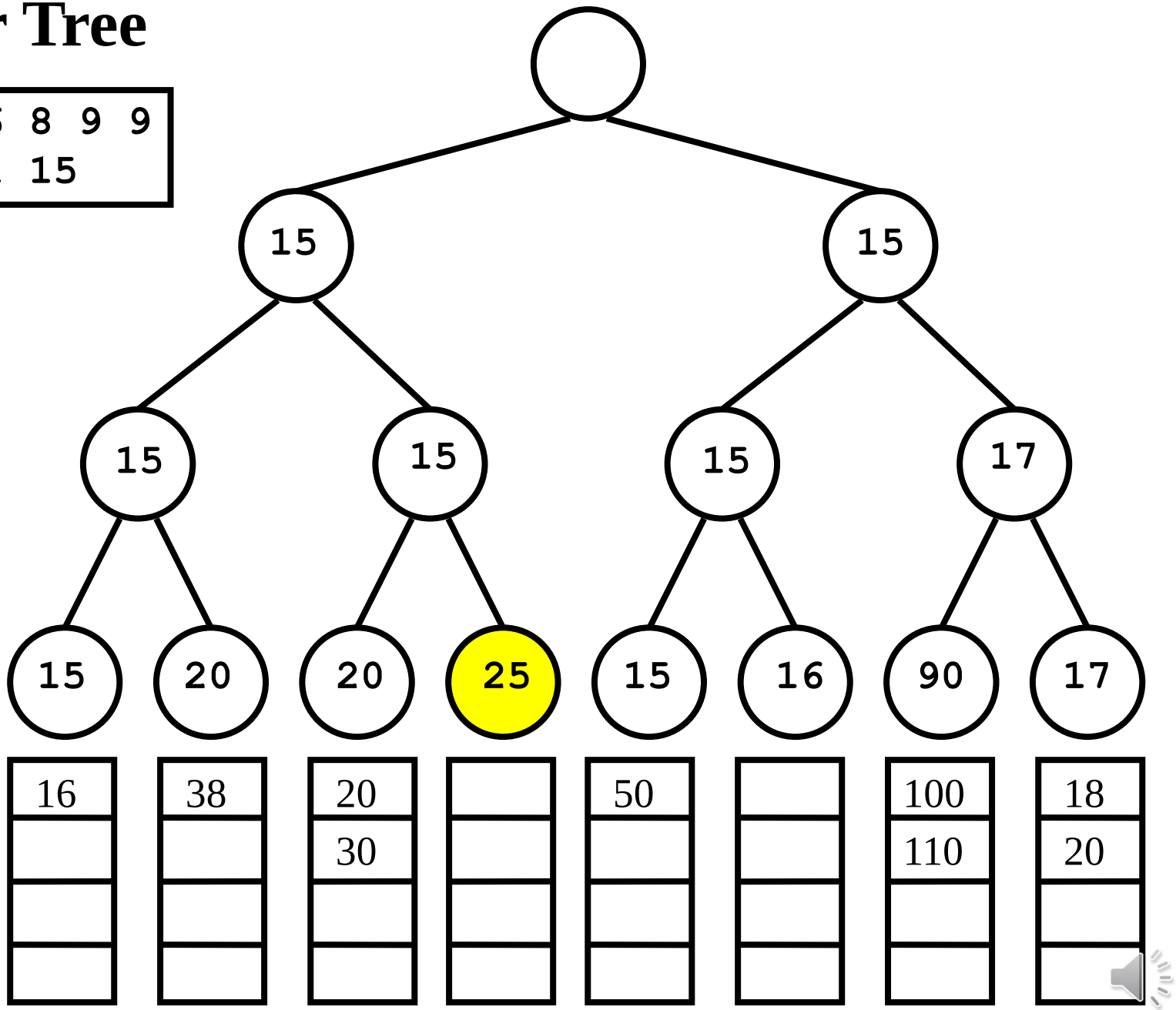
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



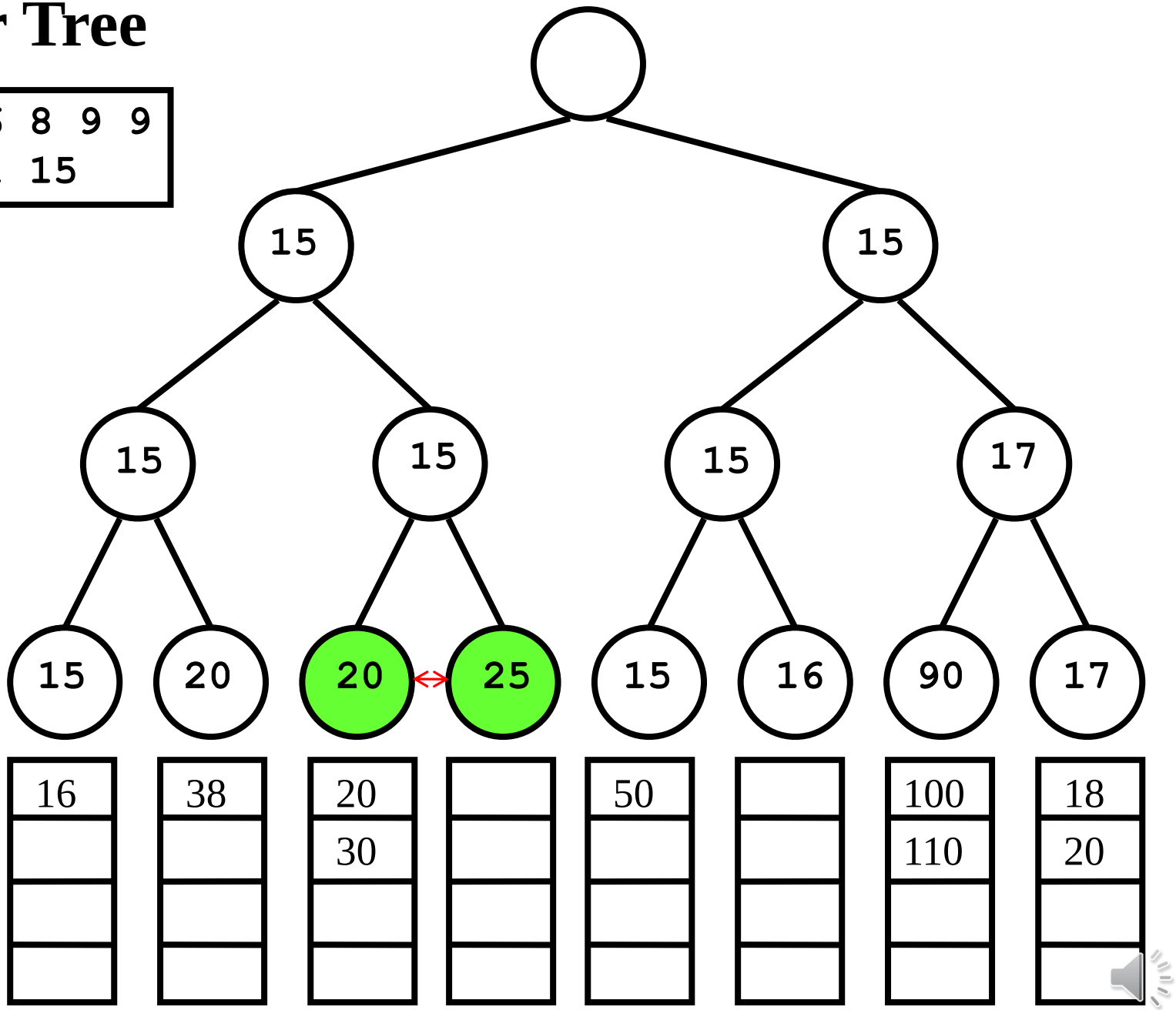
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



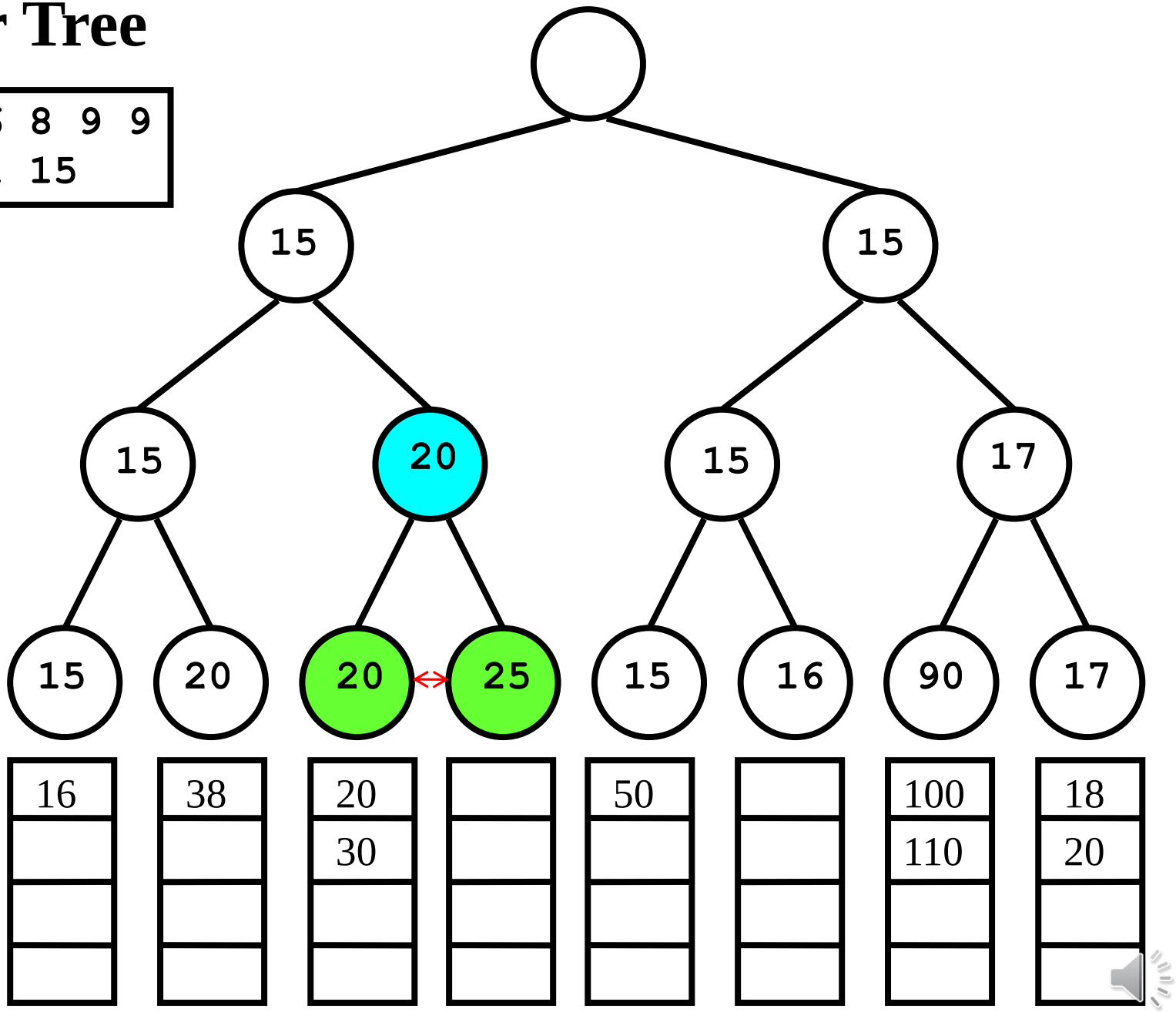
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



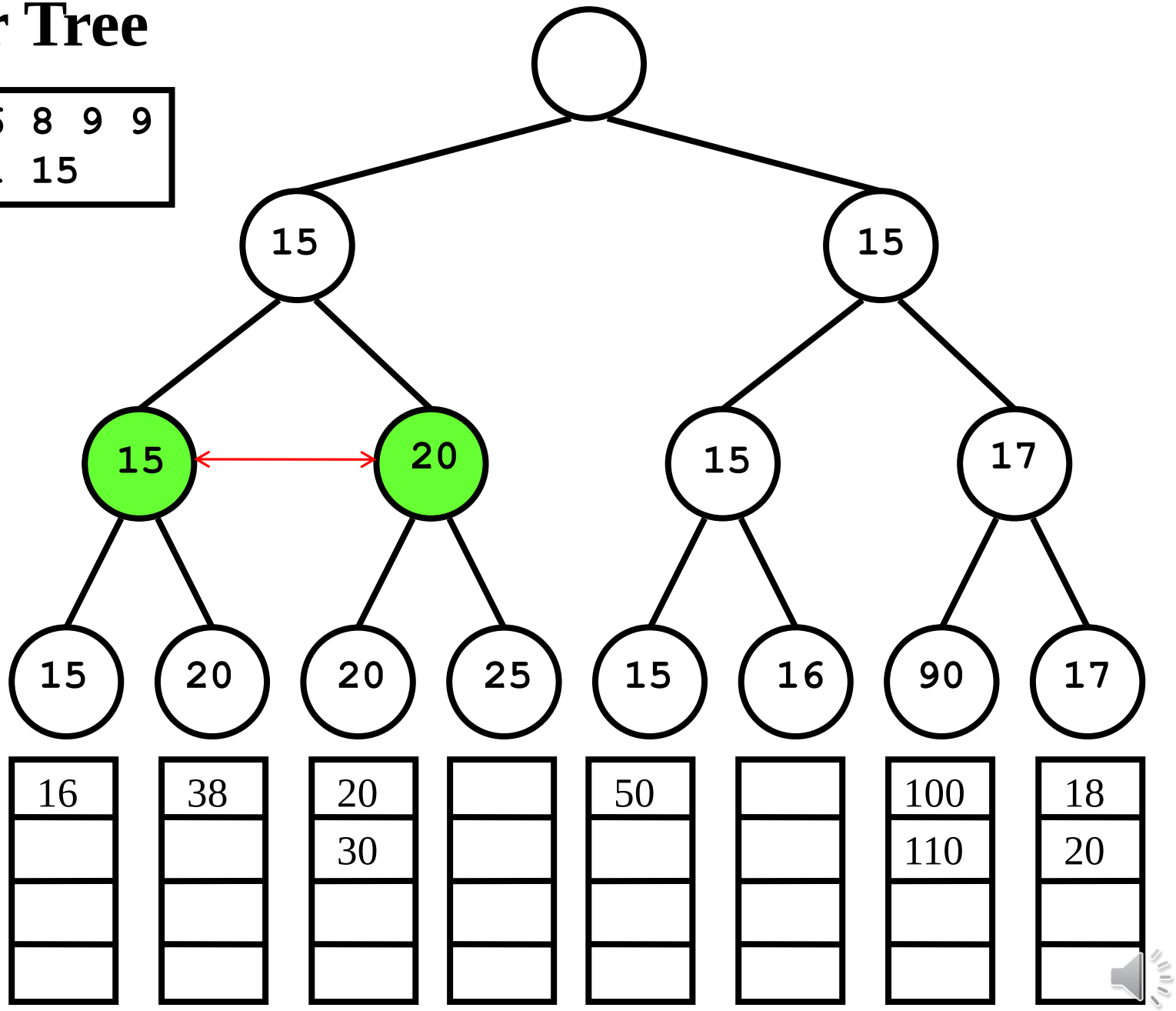
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



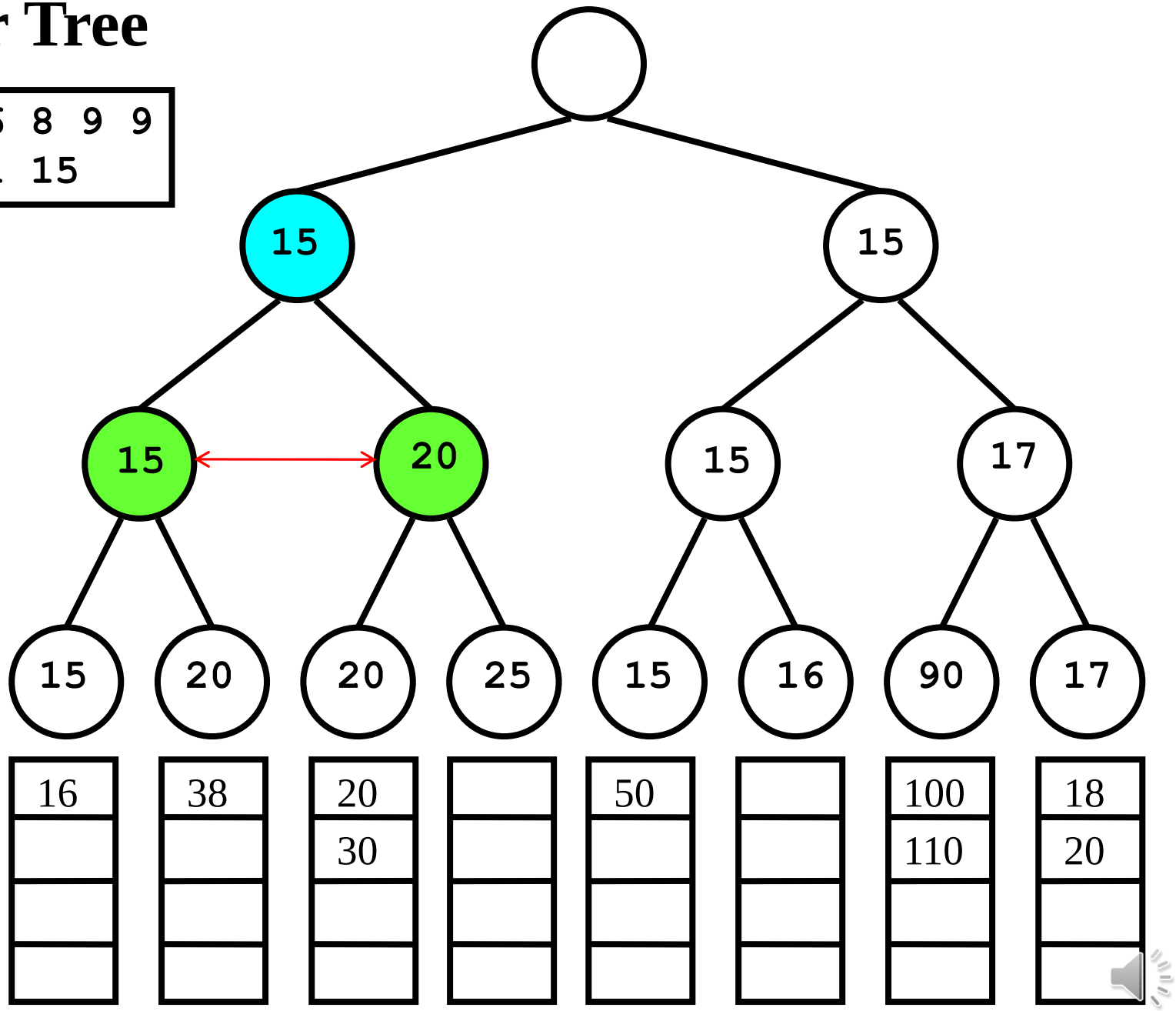
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



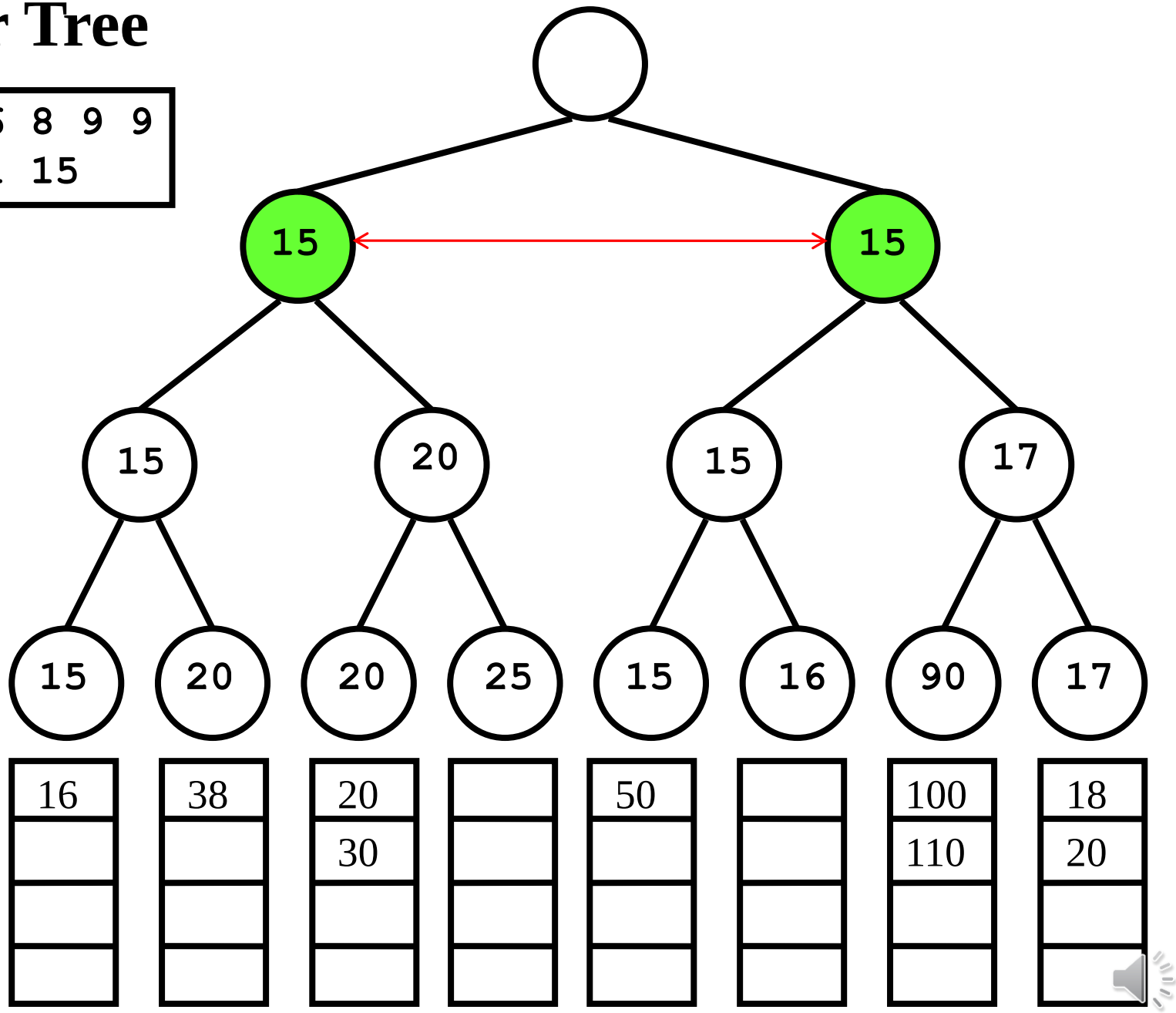
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



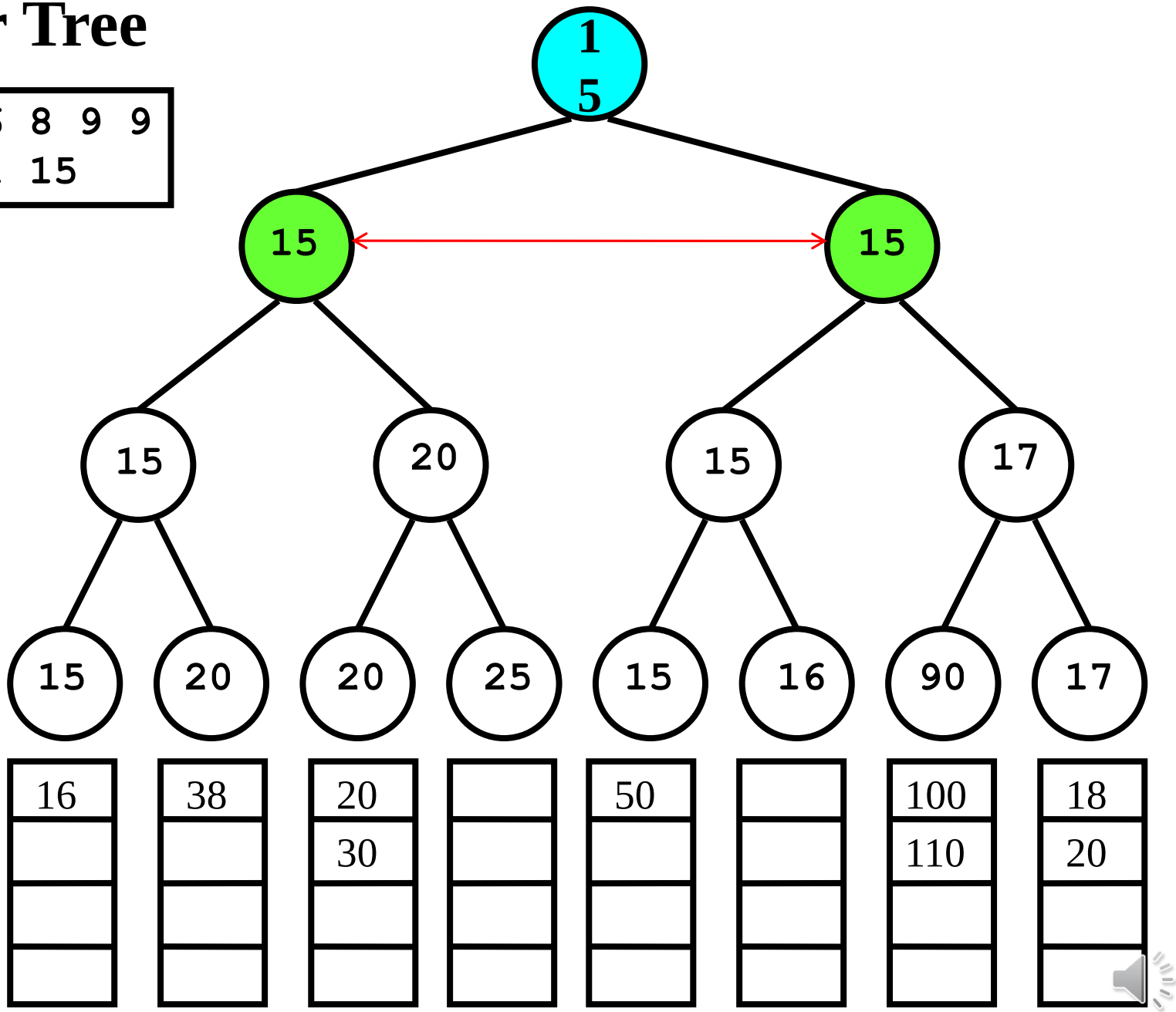
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



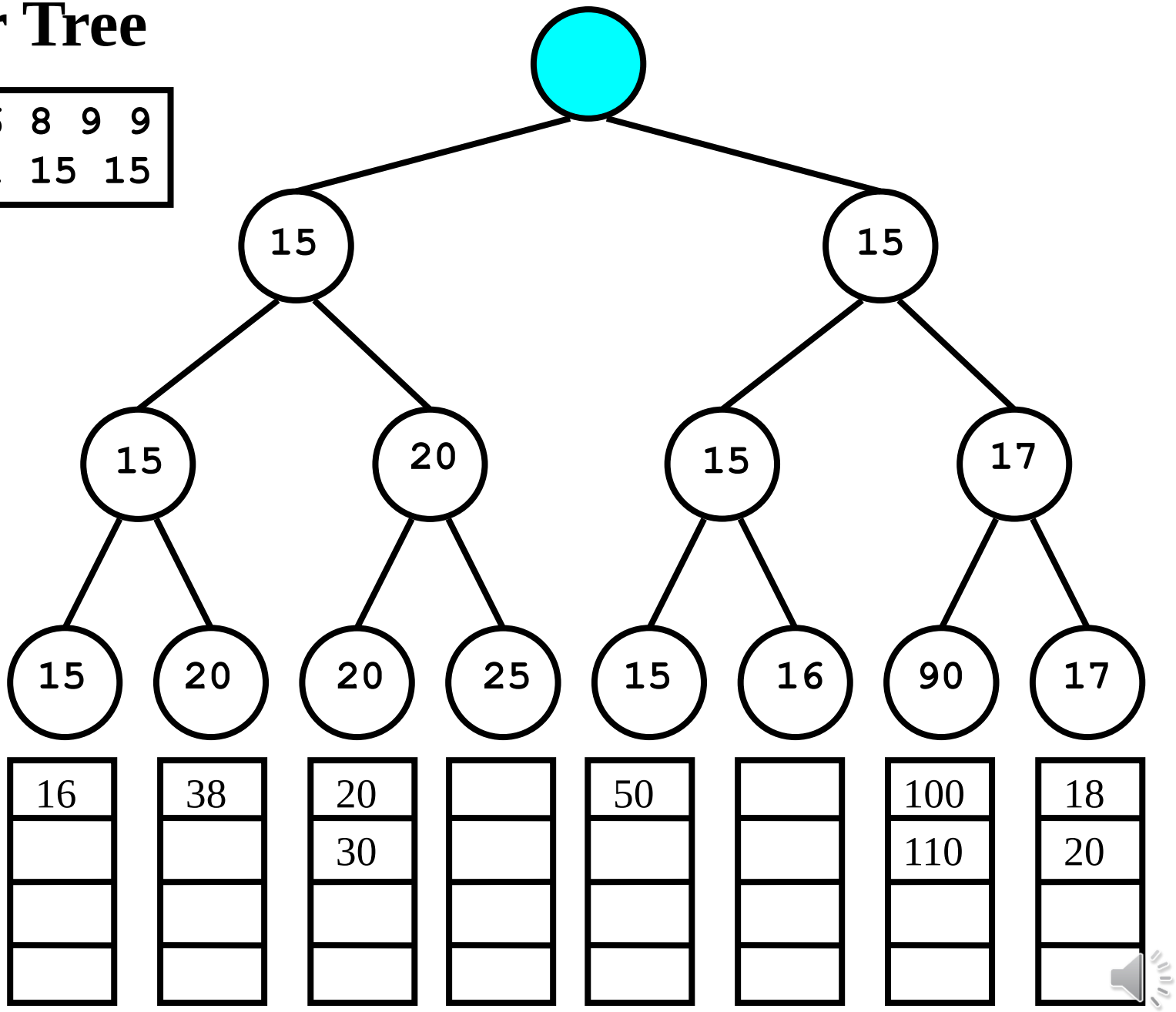
Winner Tree

```
Output: 6 8 9 9
    10 11 15
```



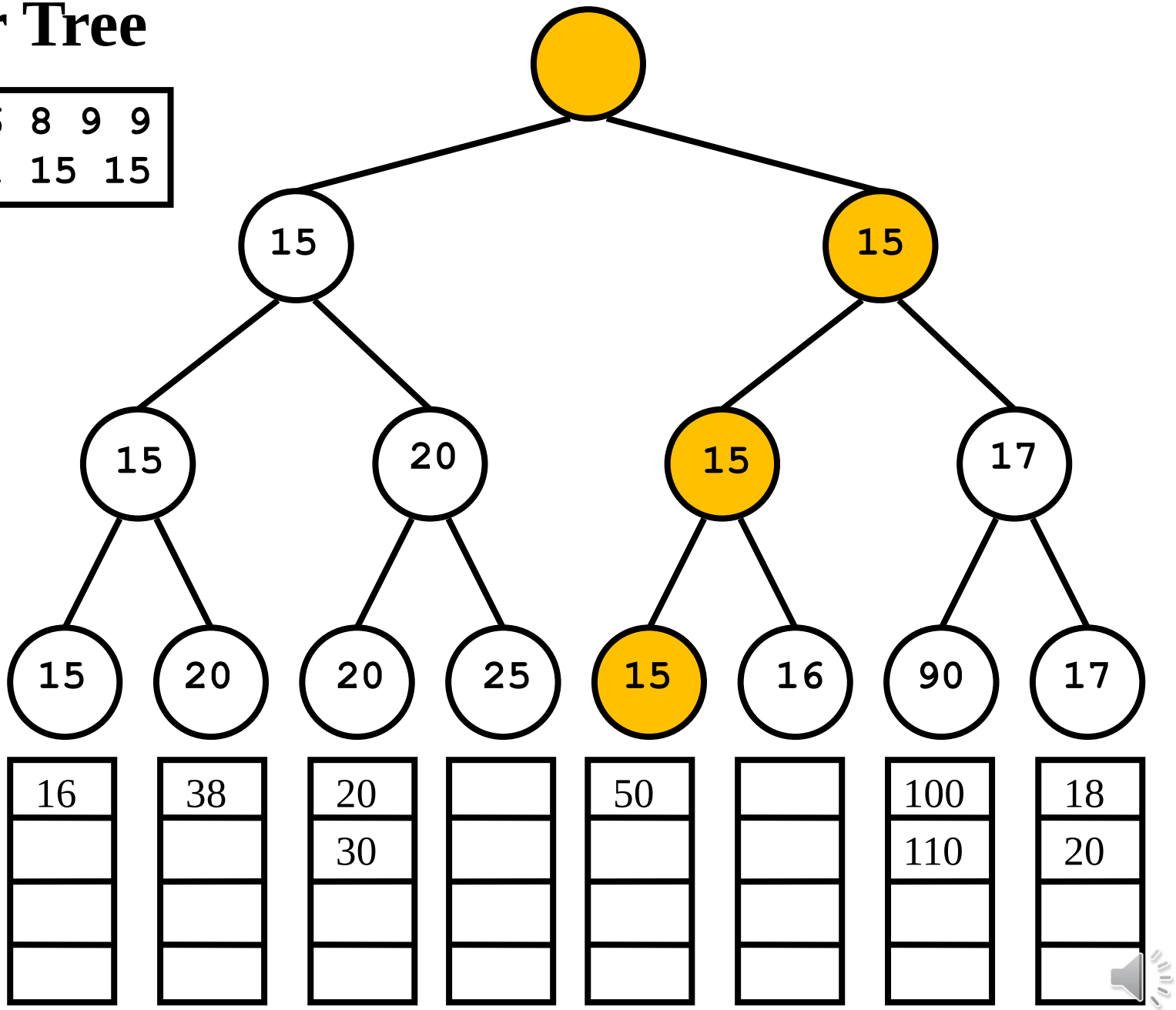
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



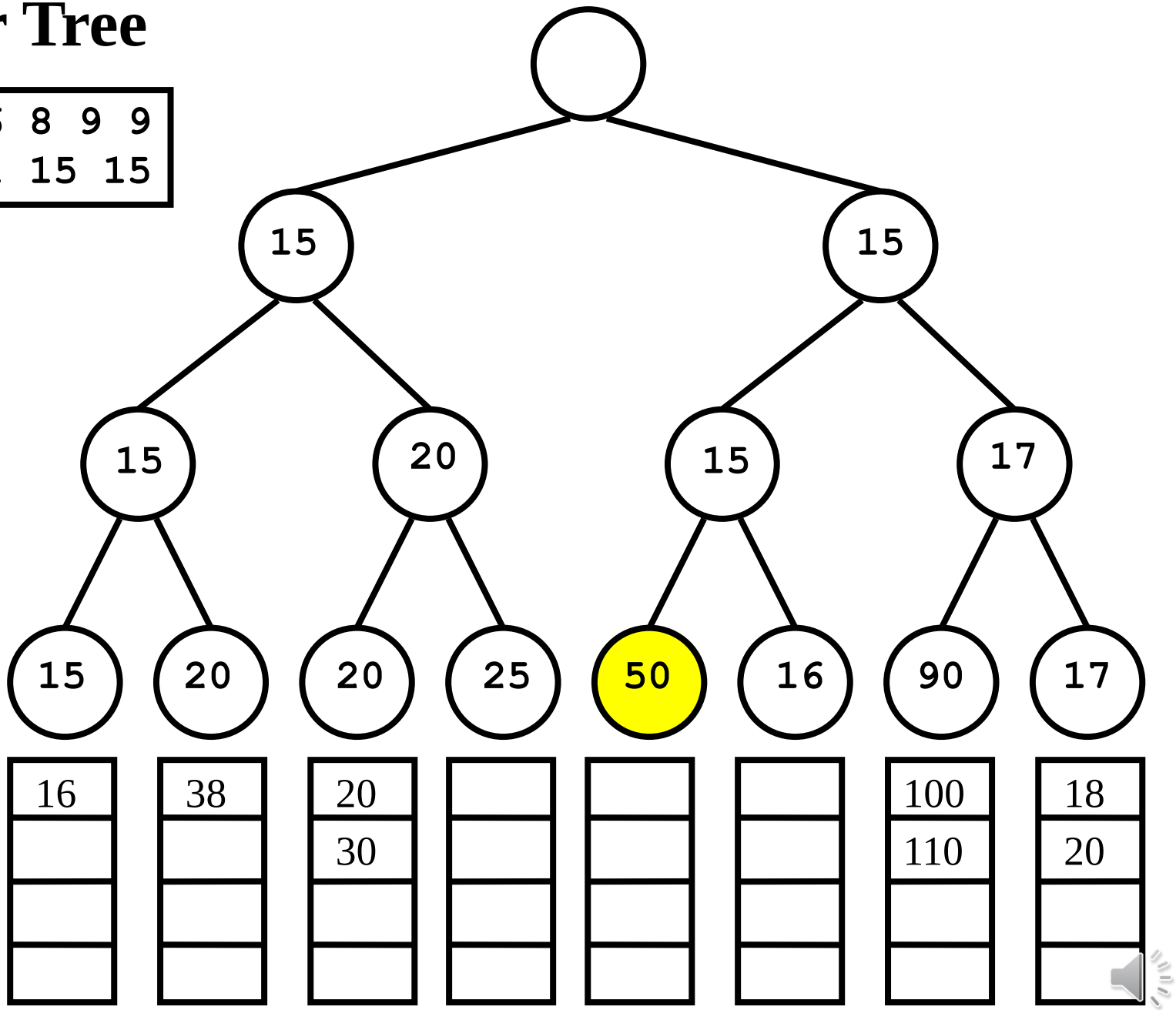
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



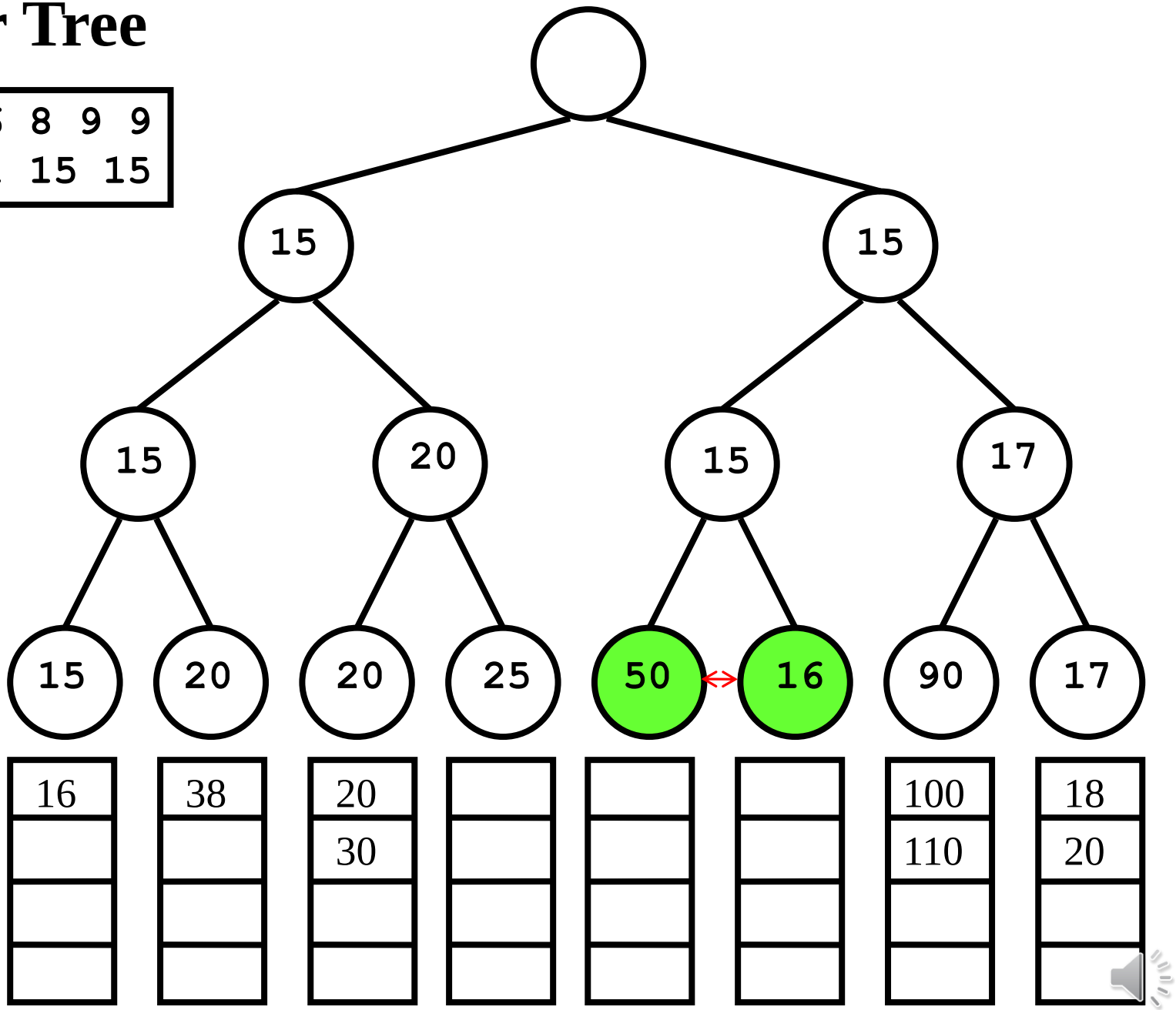
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



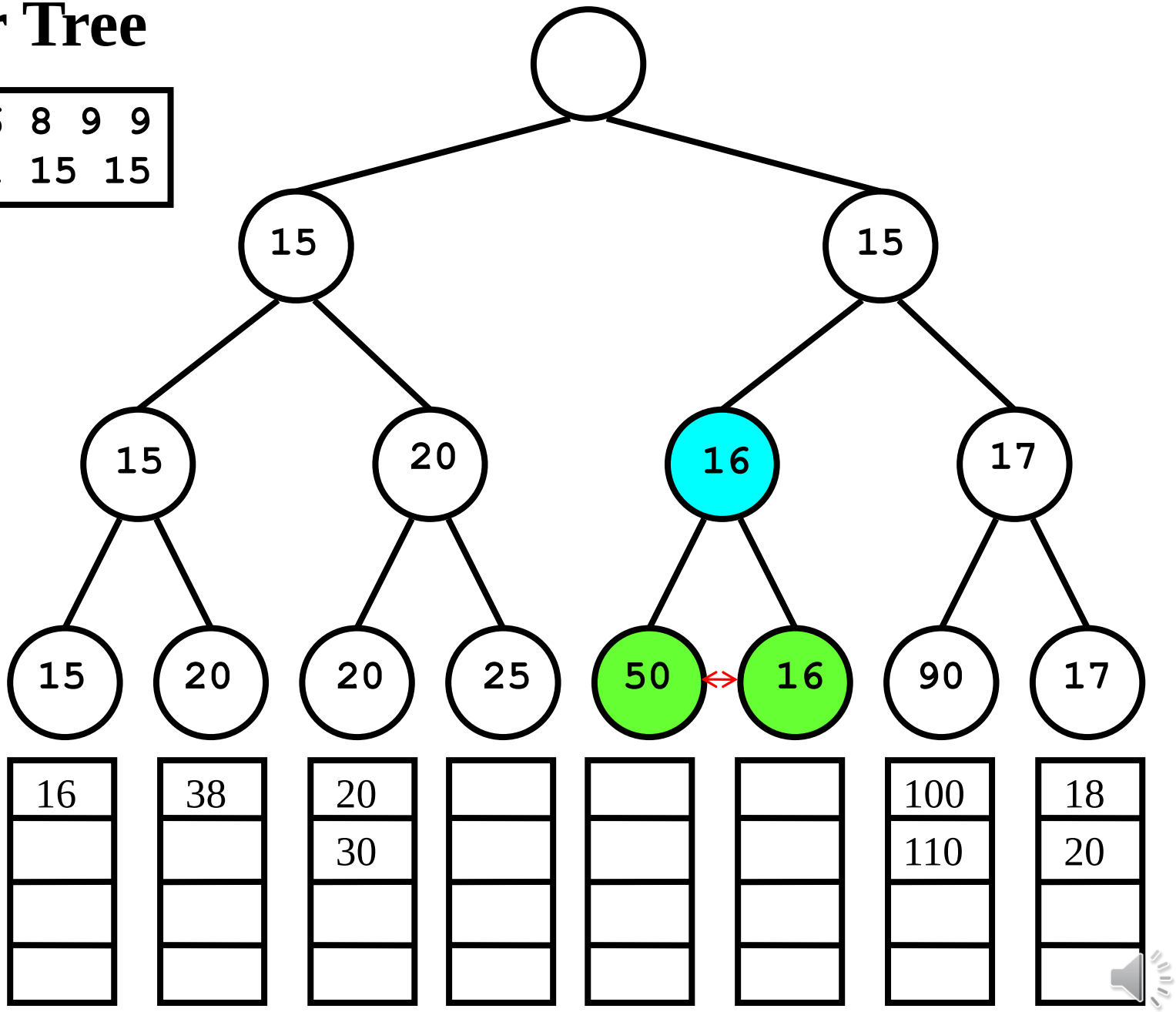
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



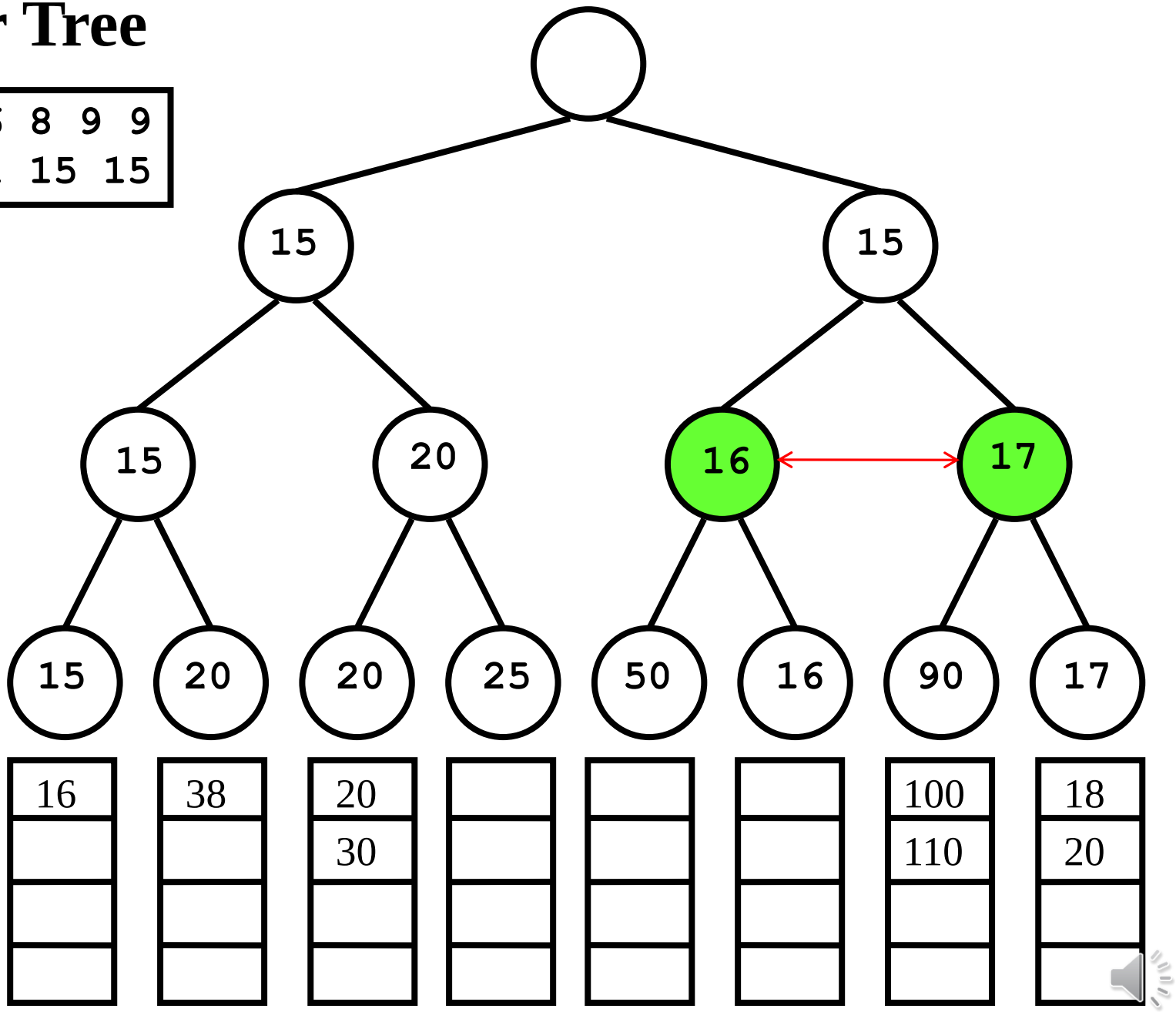
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



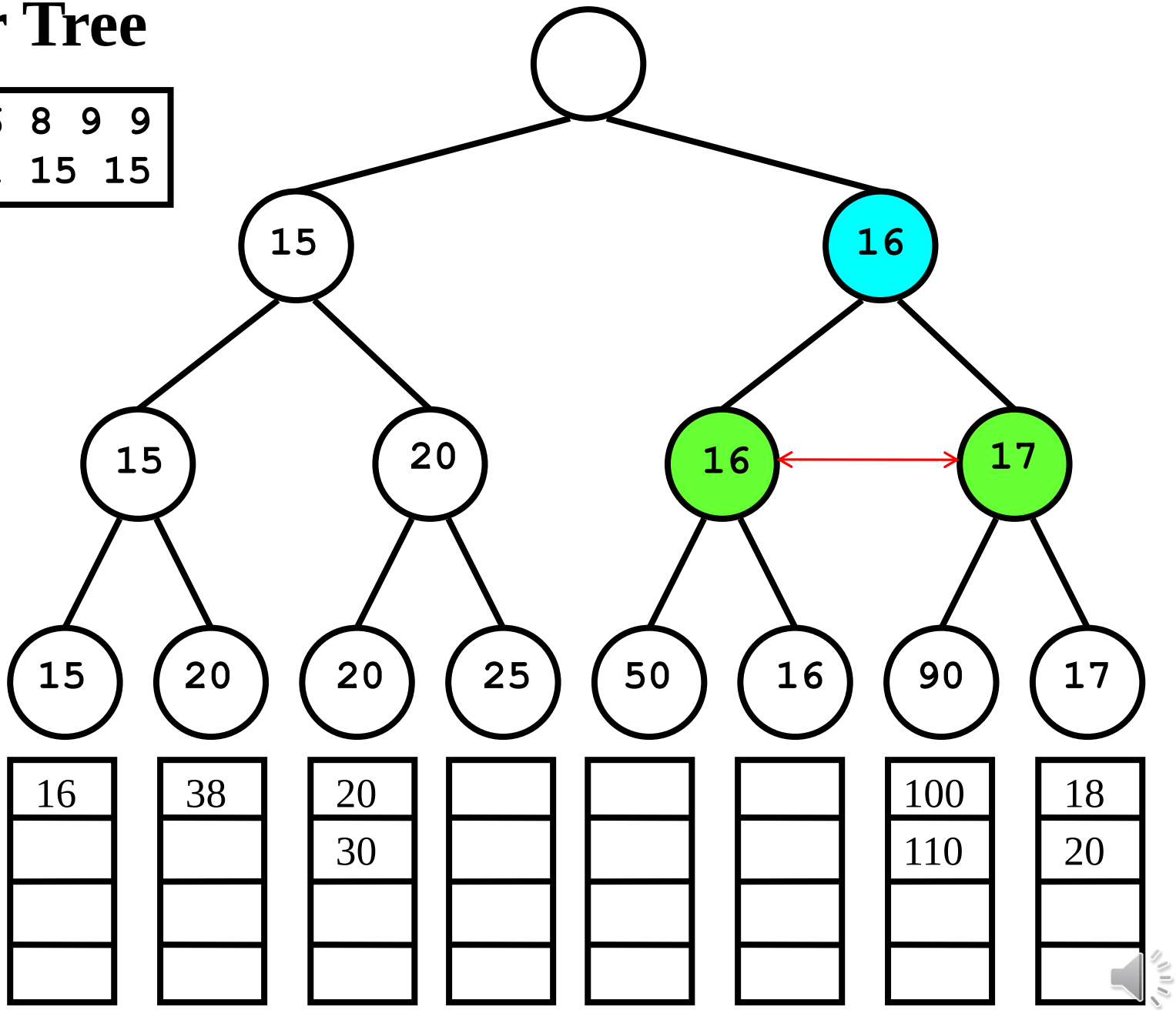
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



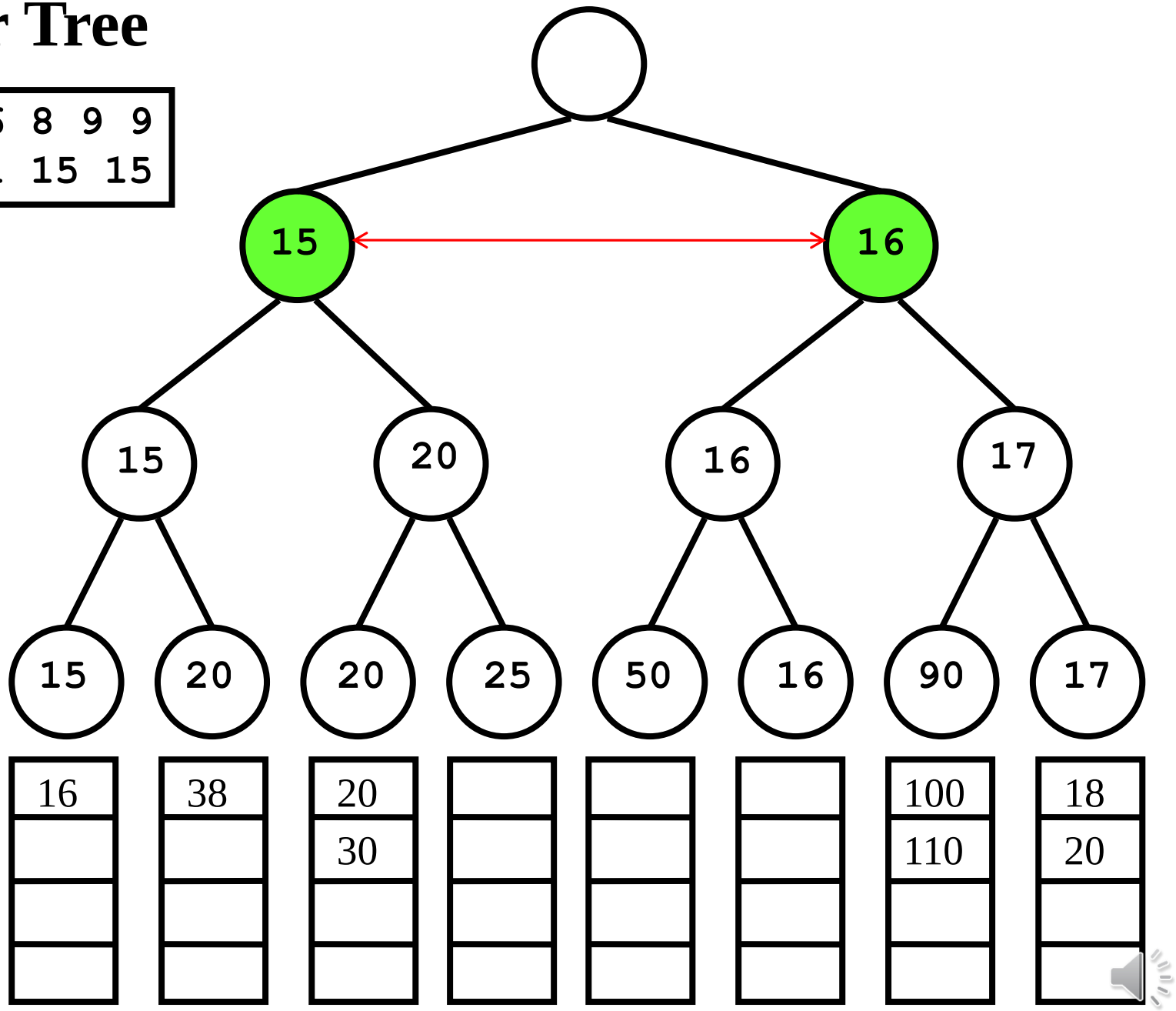
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



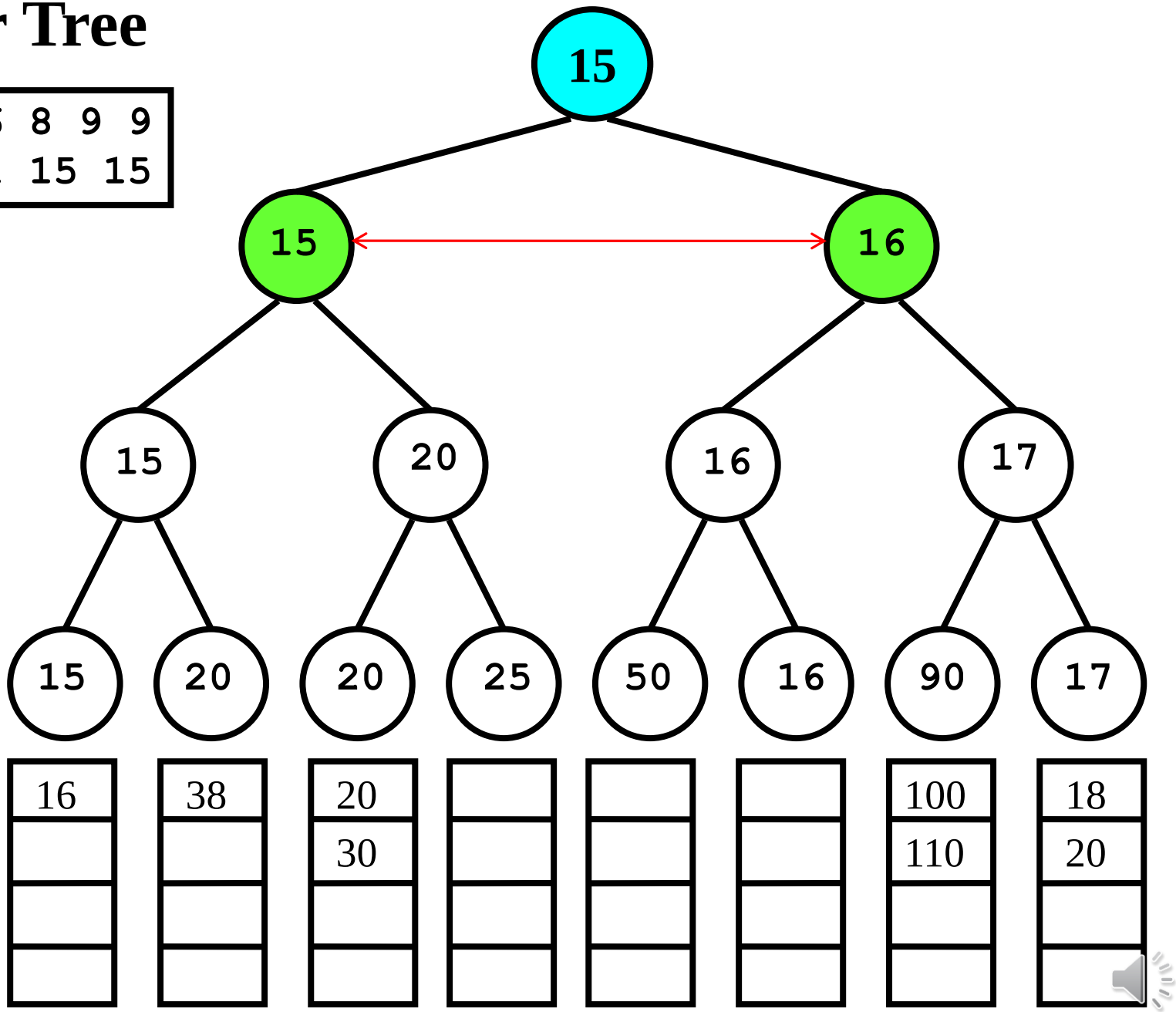
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



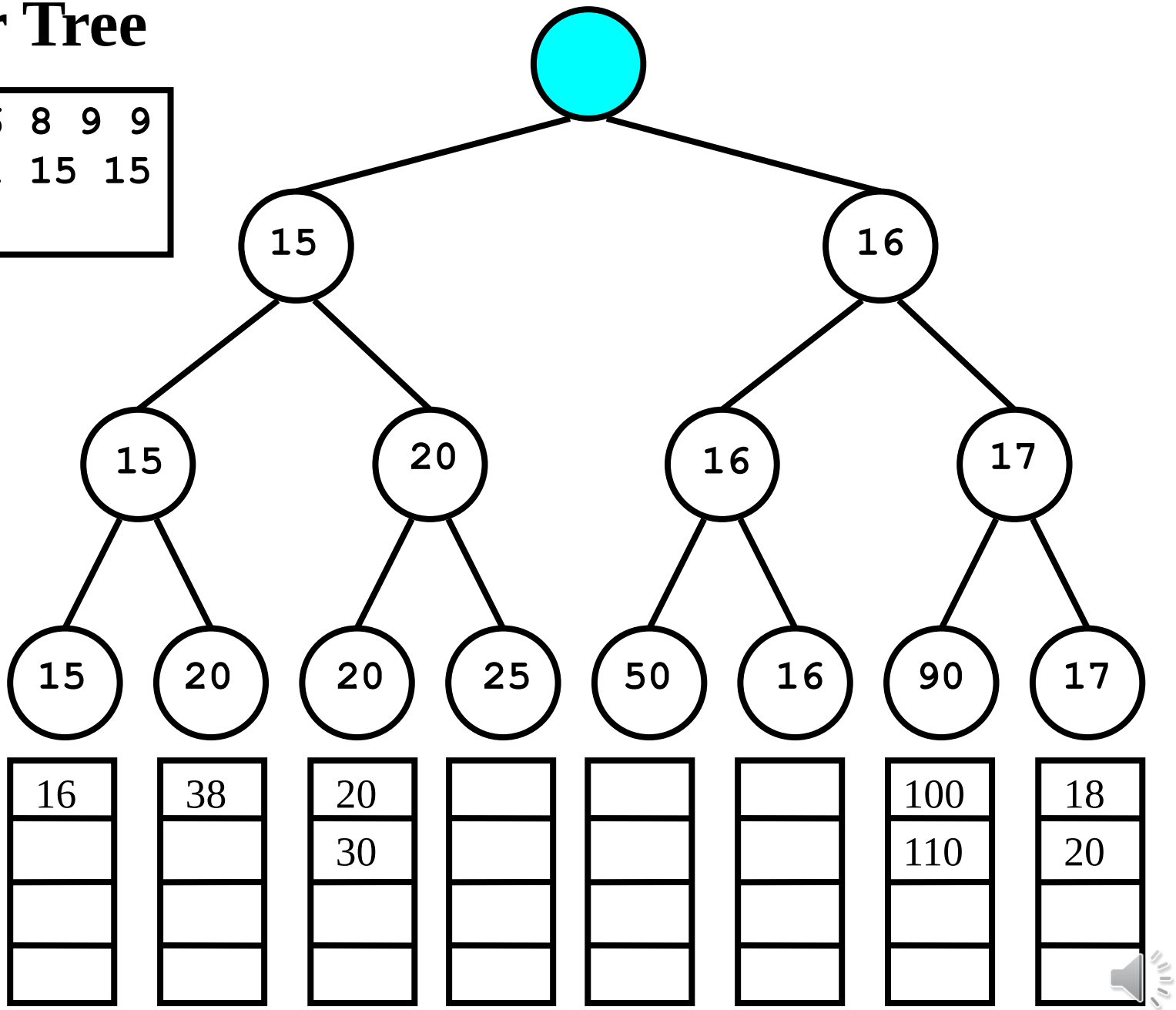
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
```



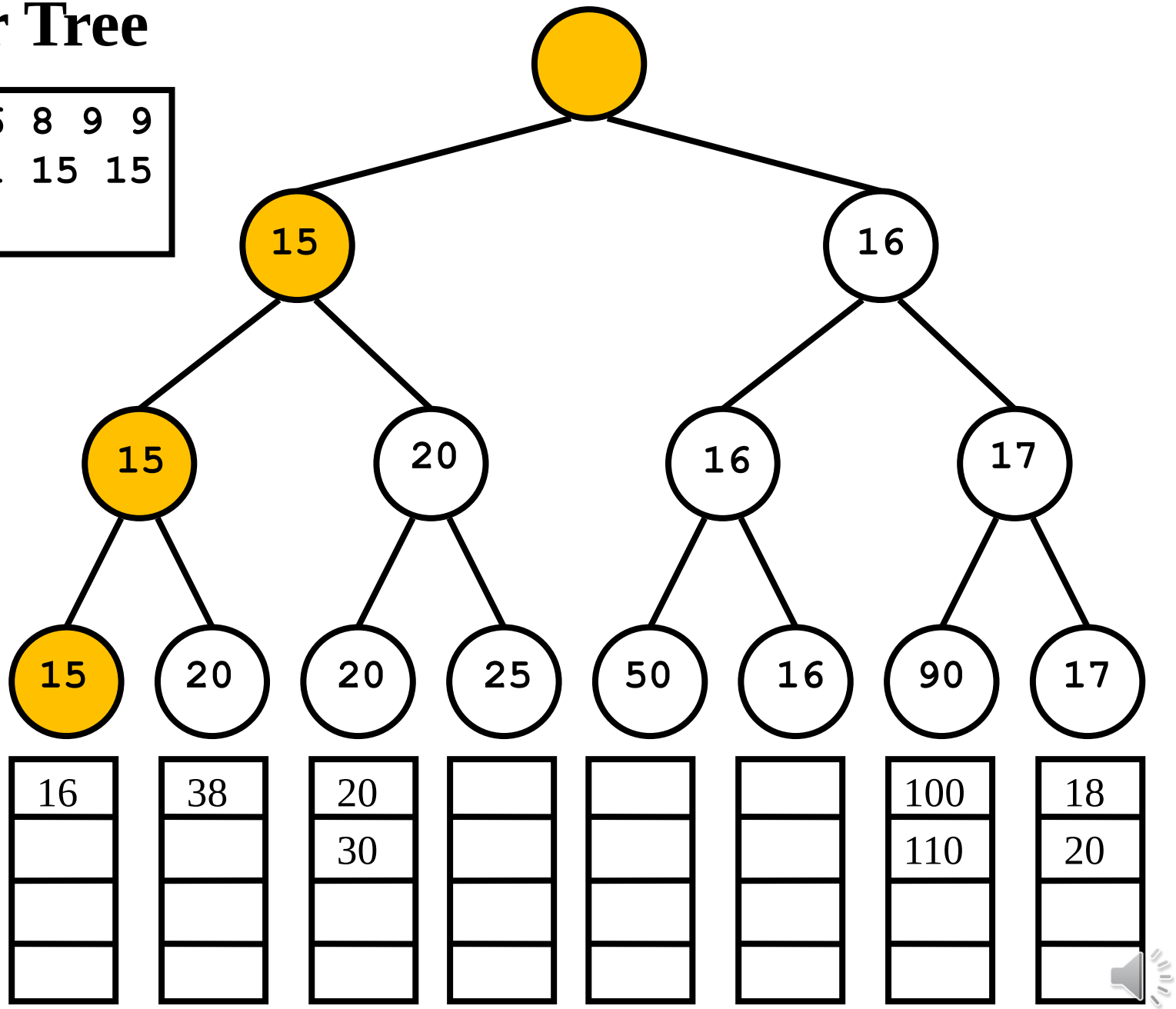
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



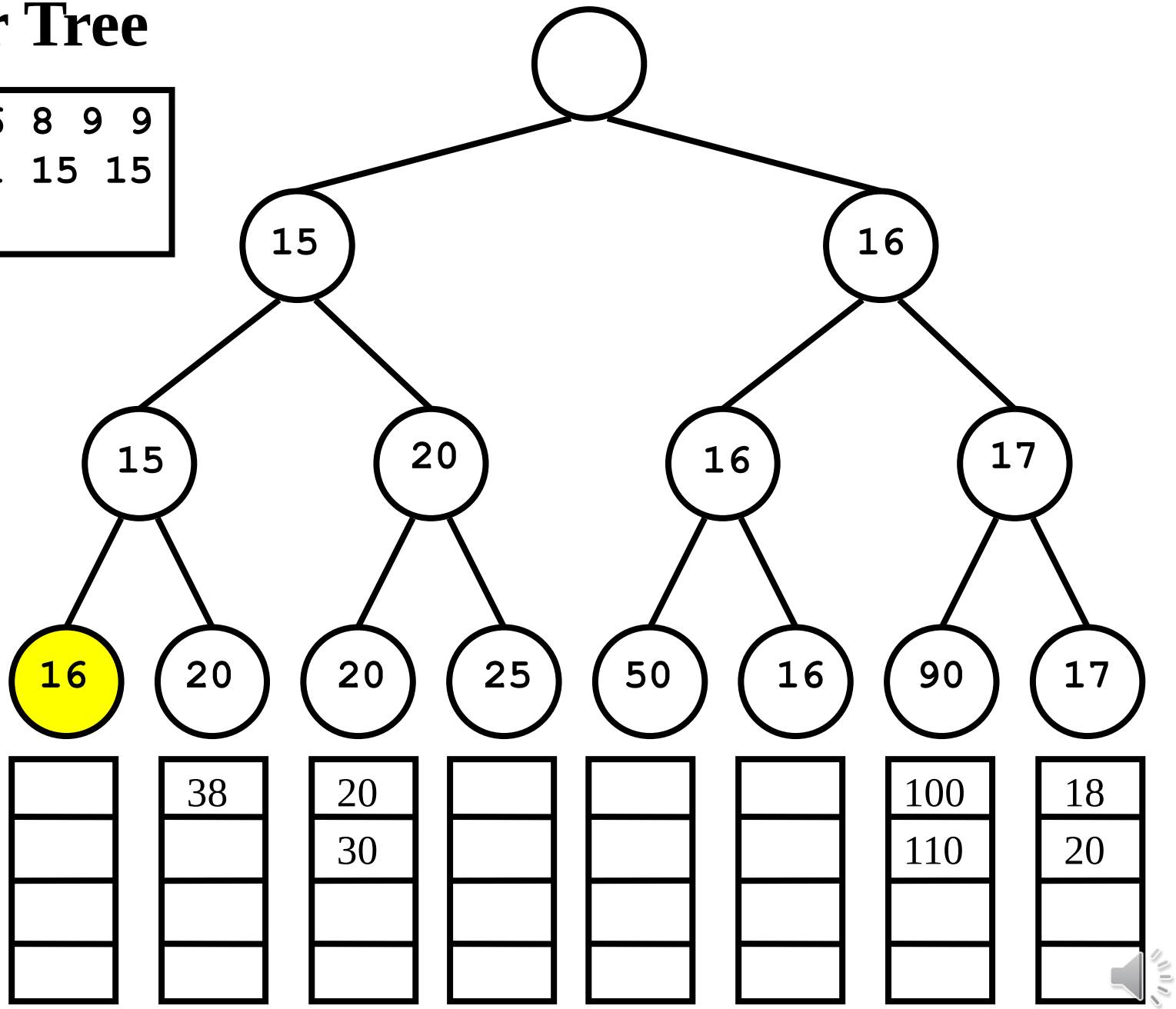
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



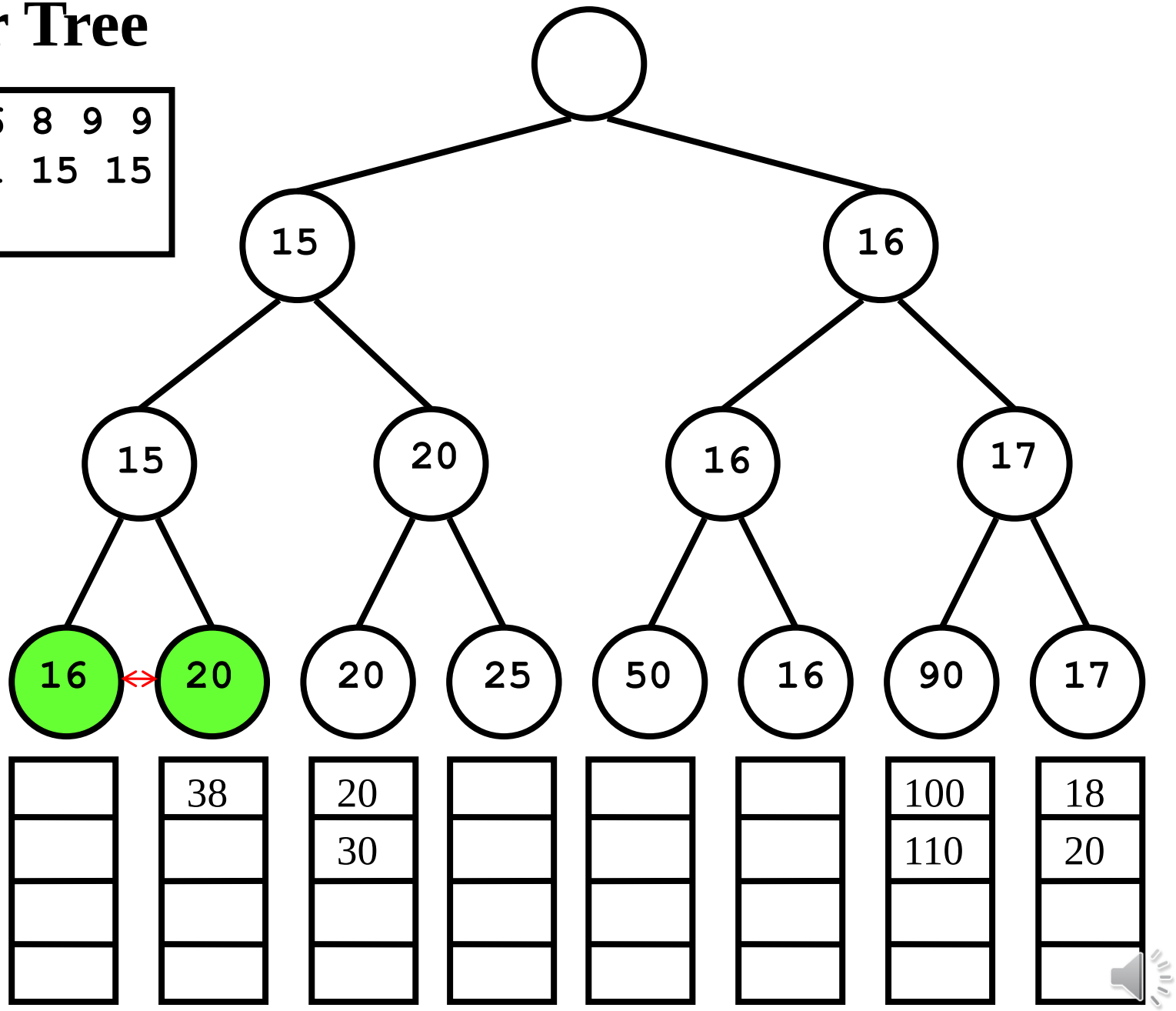
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



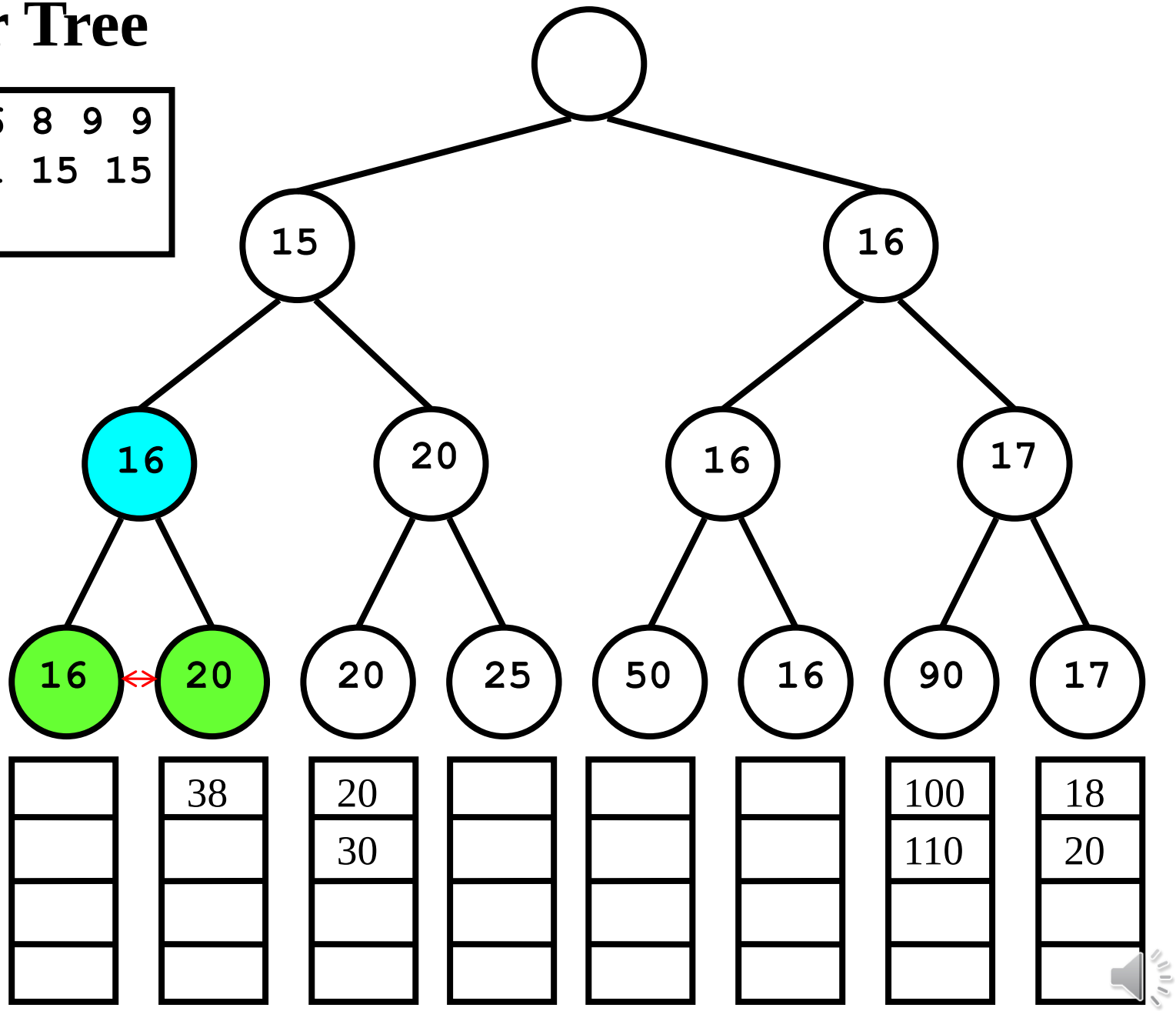
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



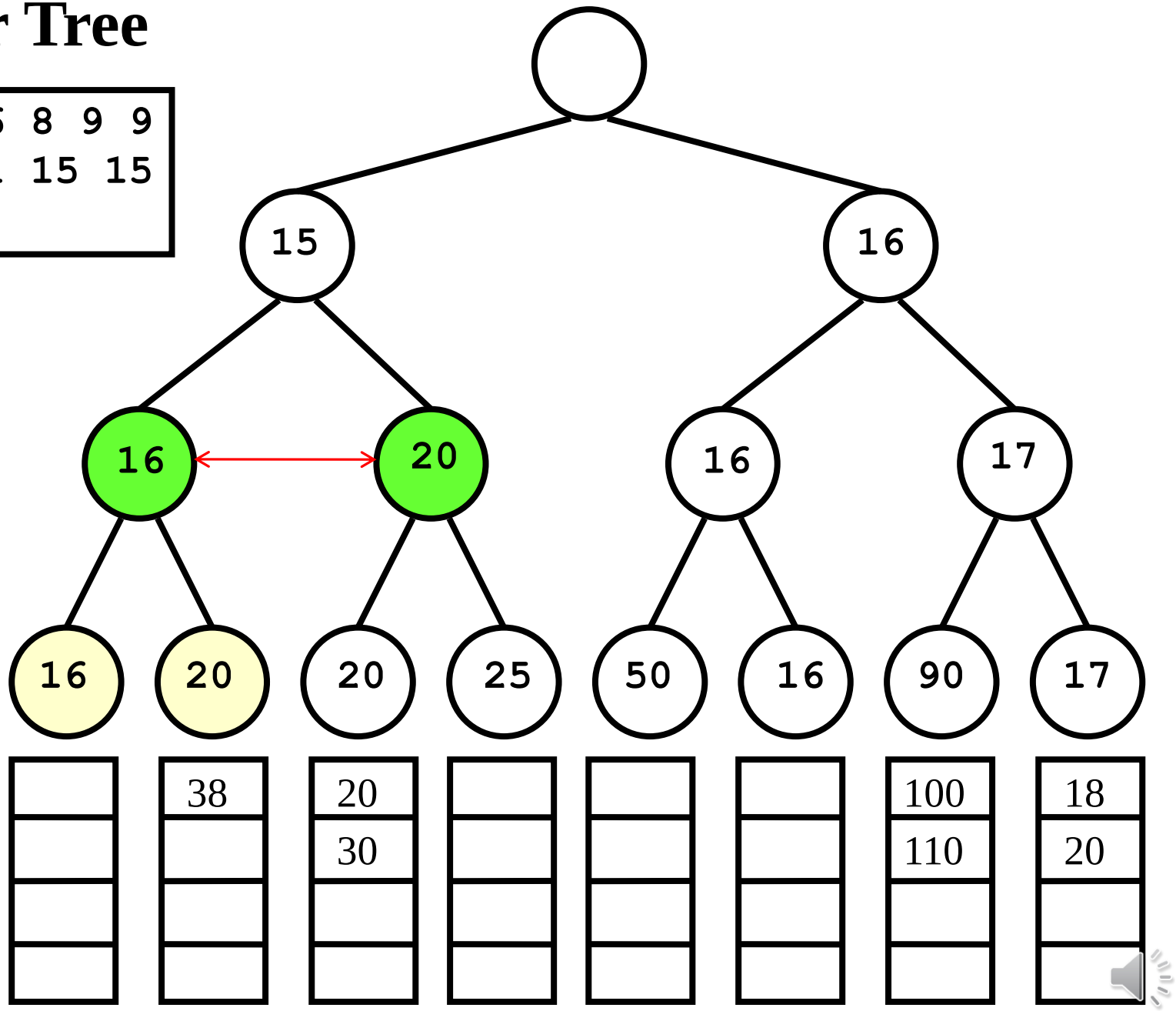
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



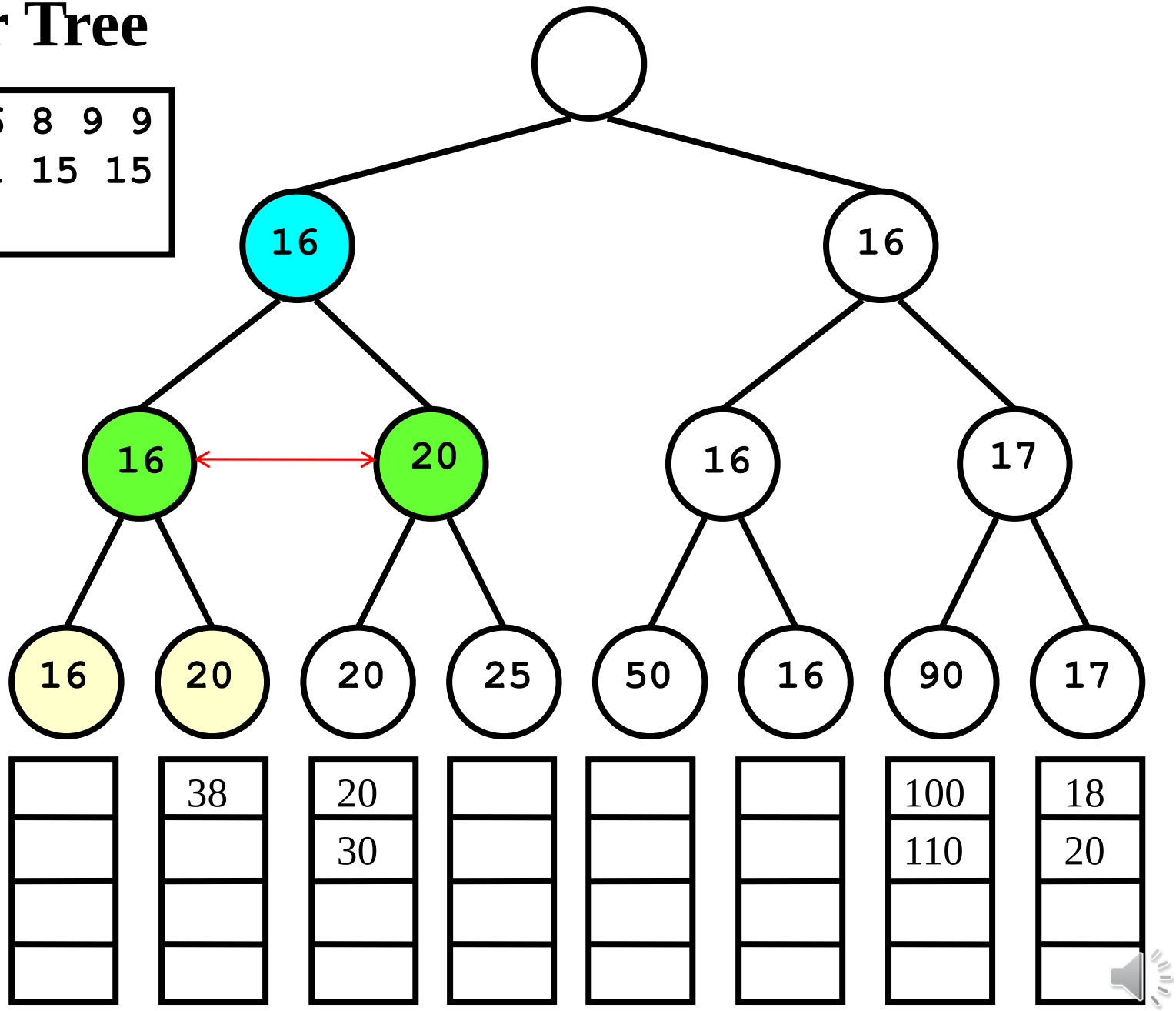
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



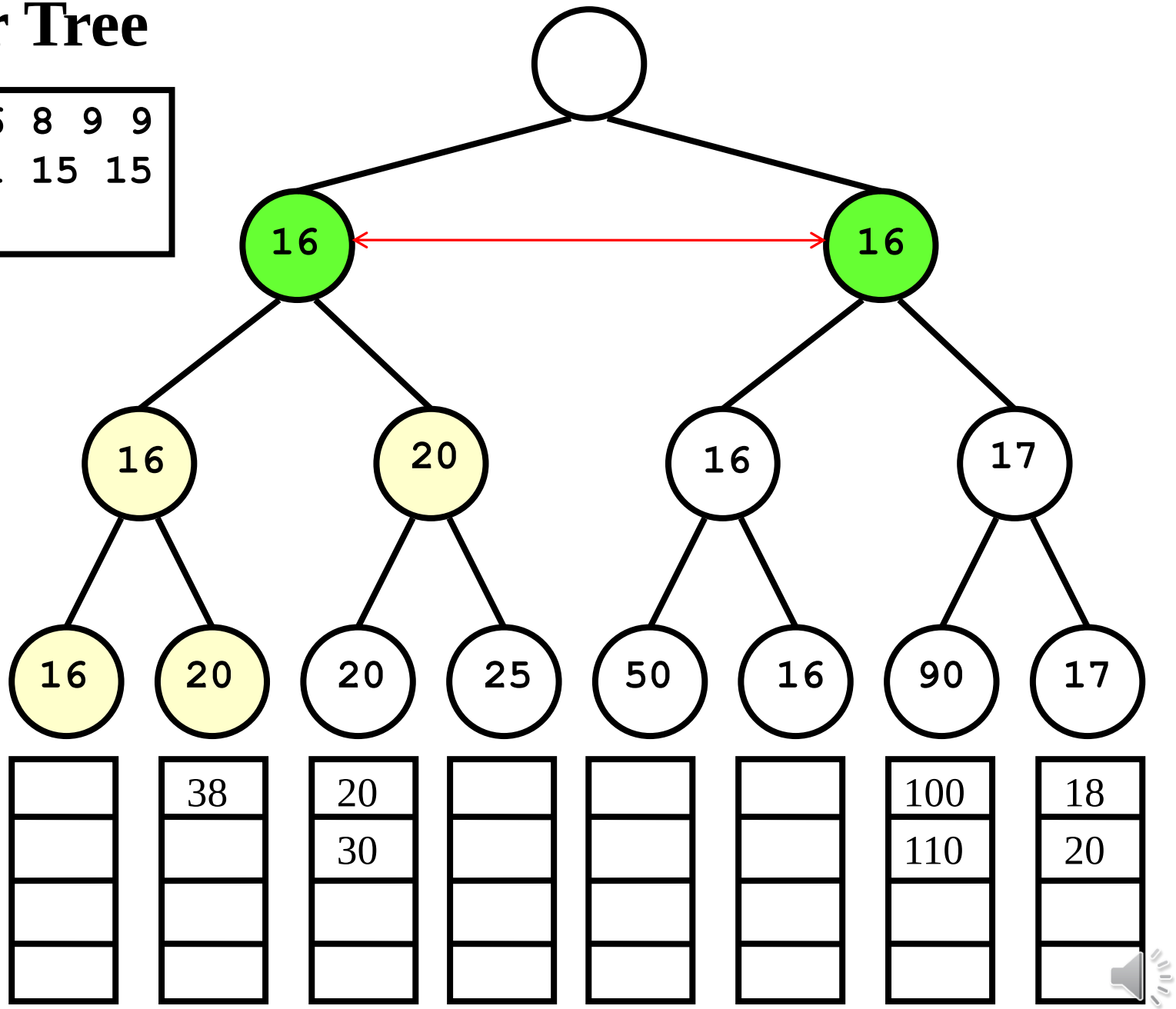
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



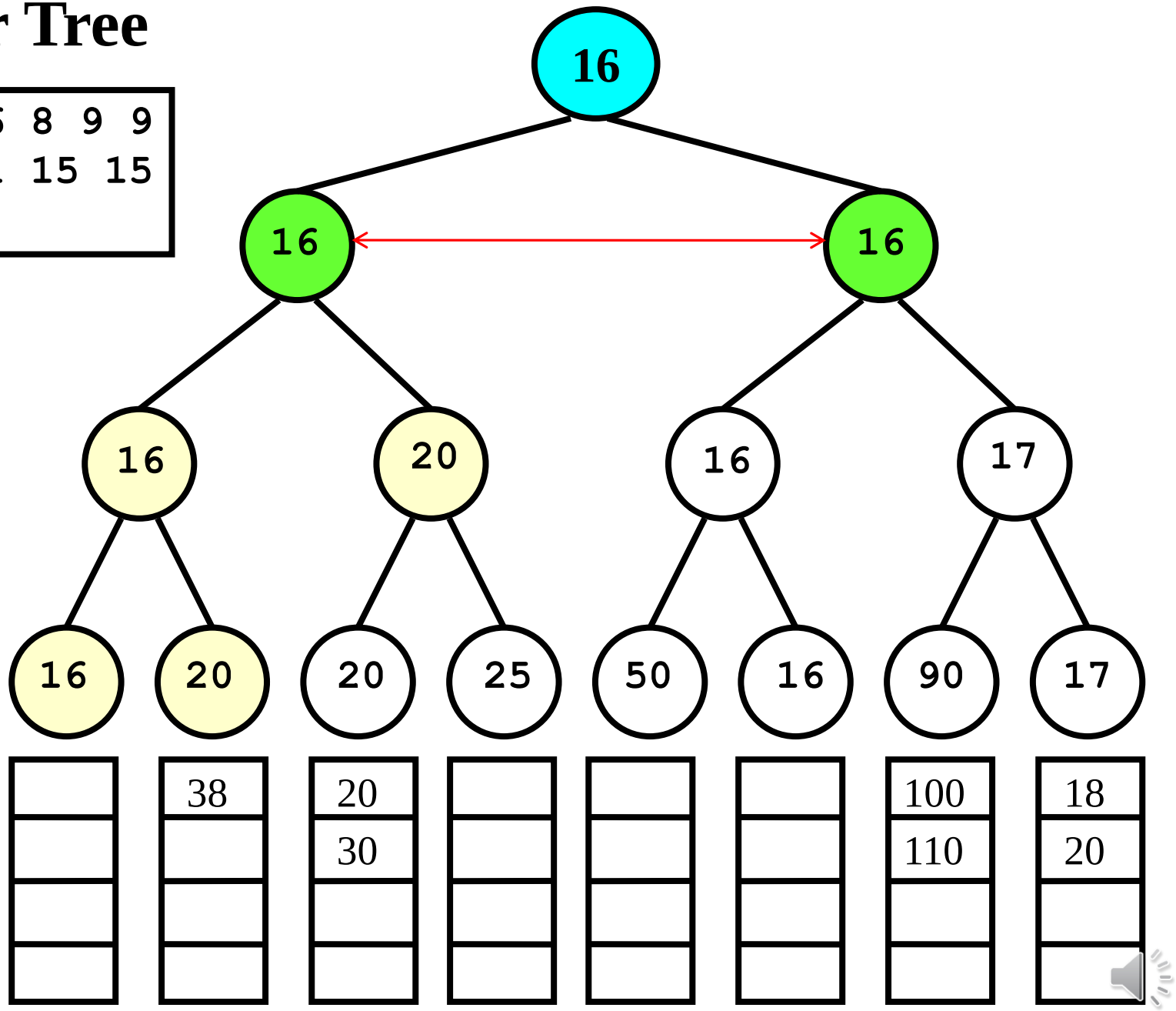
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



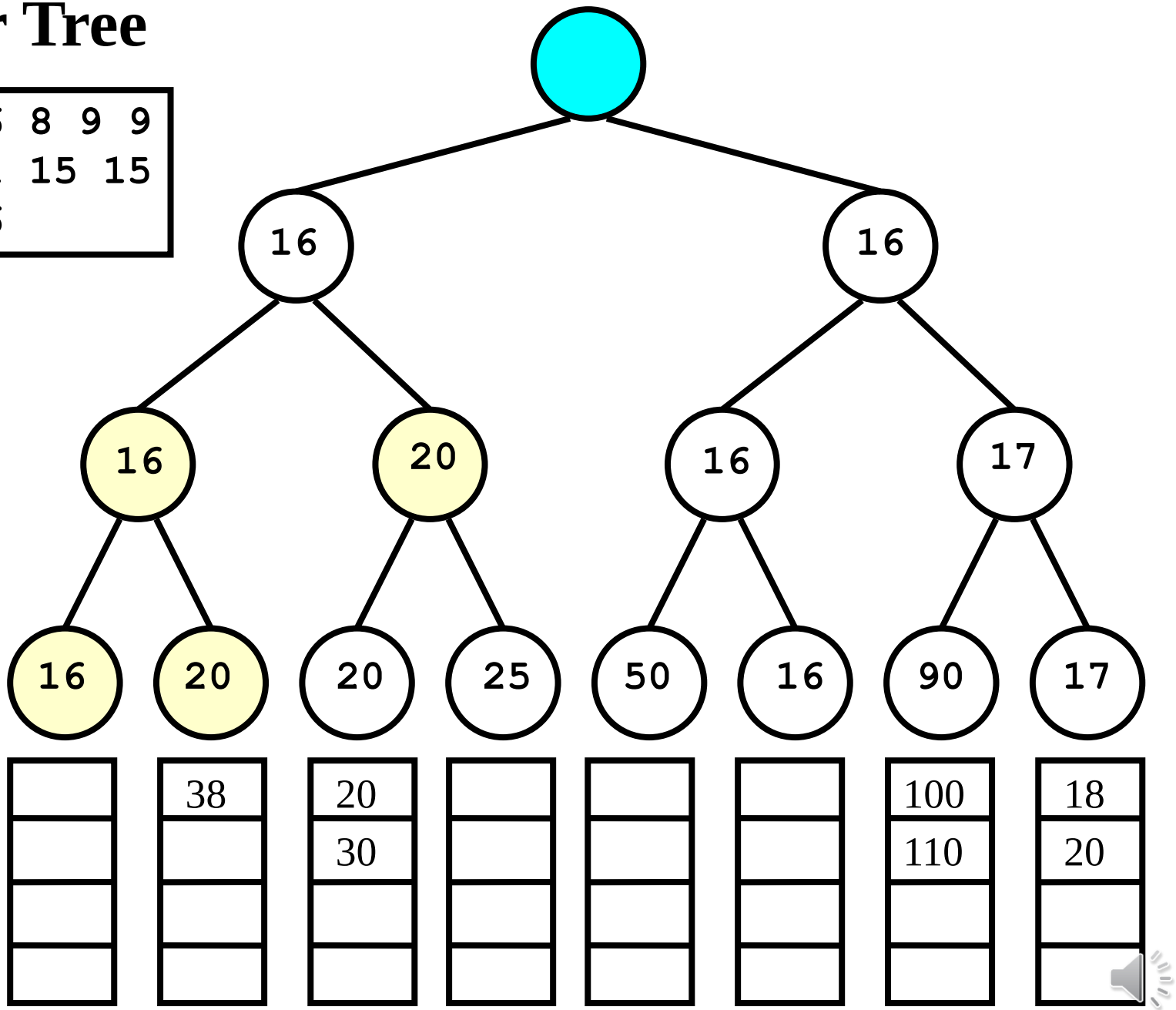
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15
```



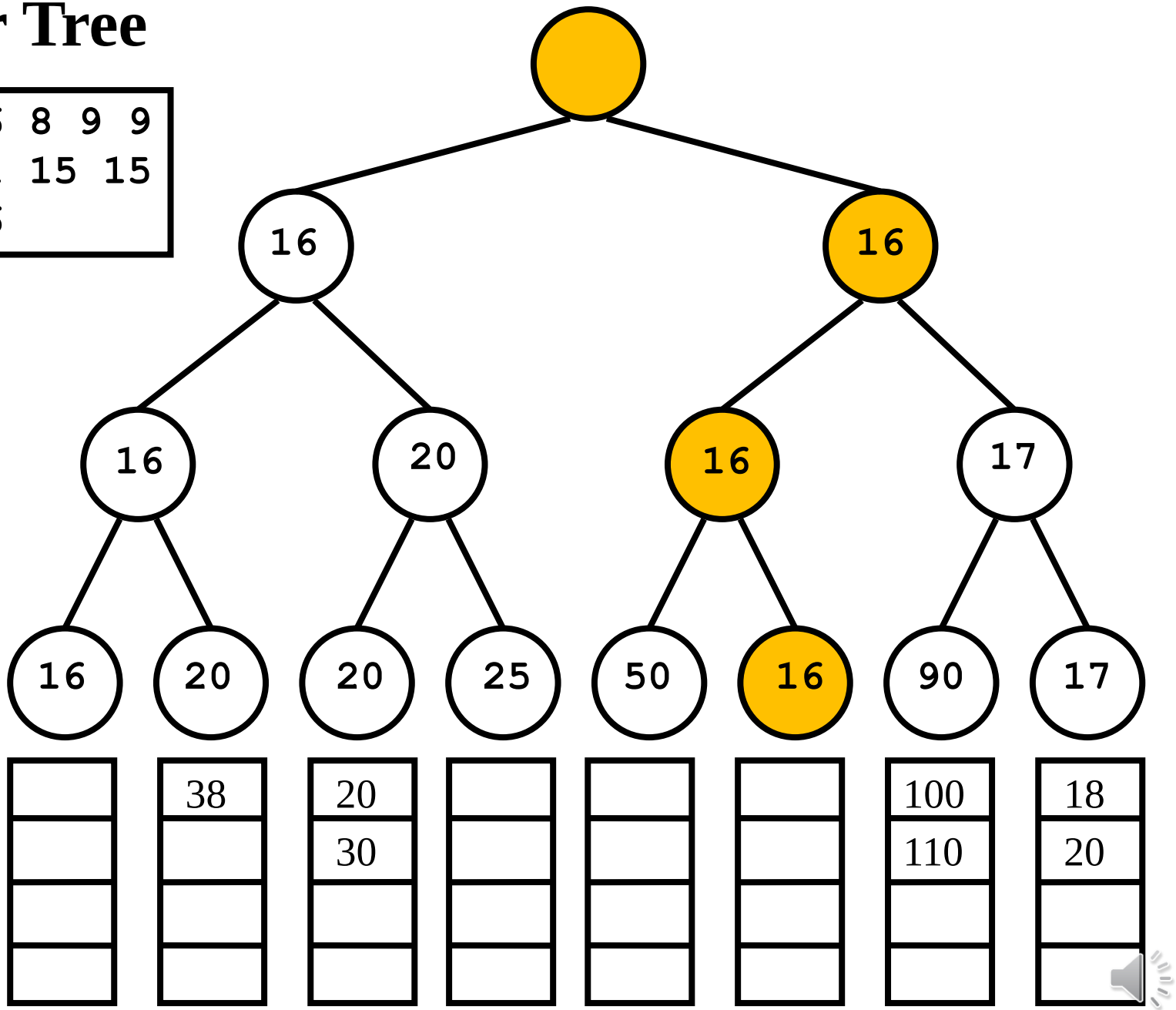
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



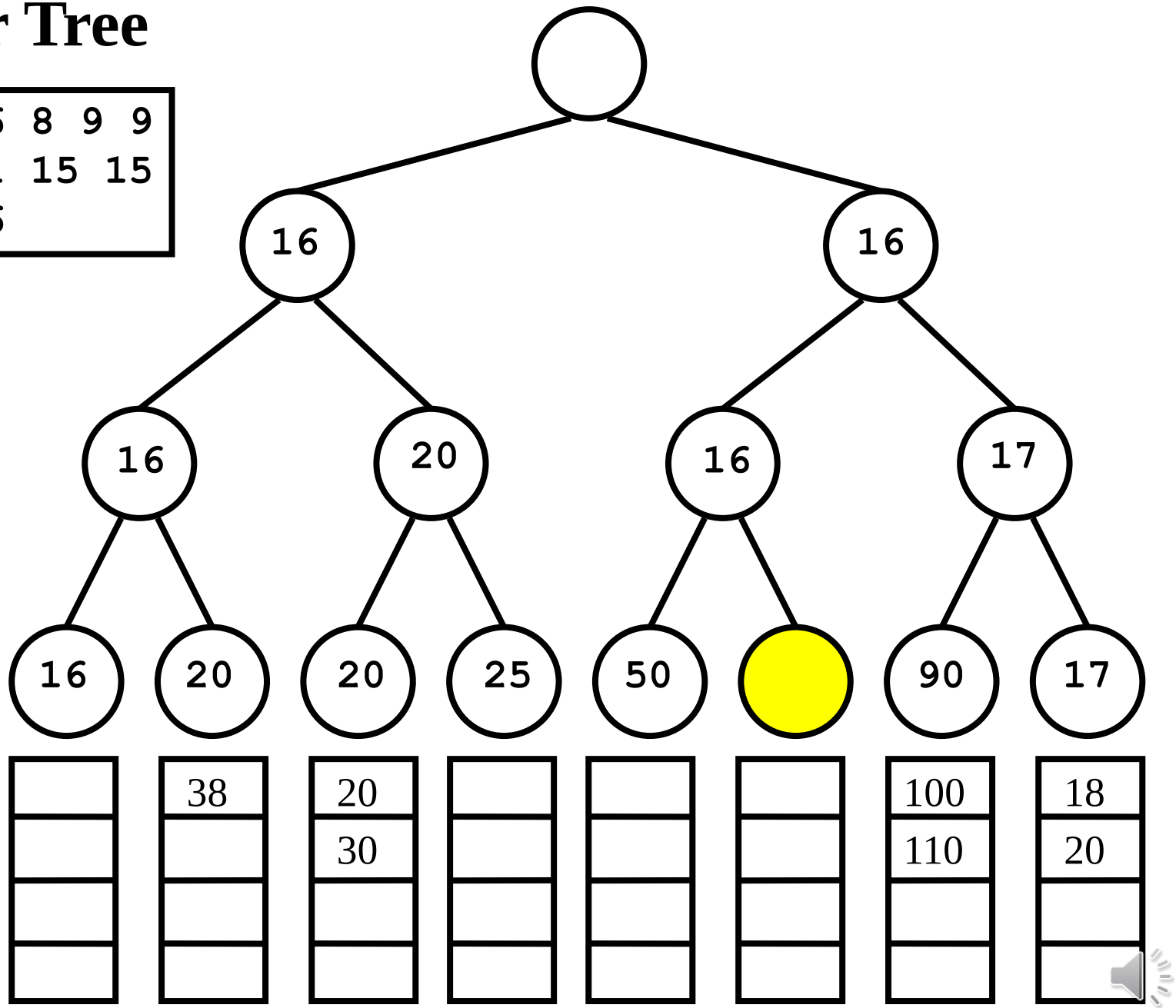
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



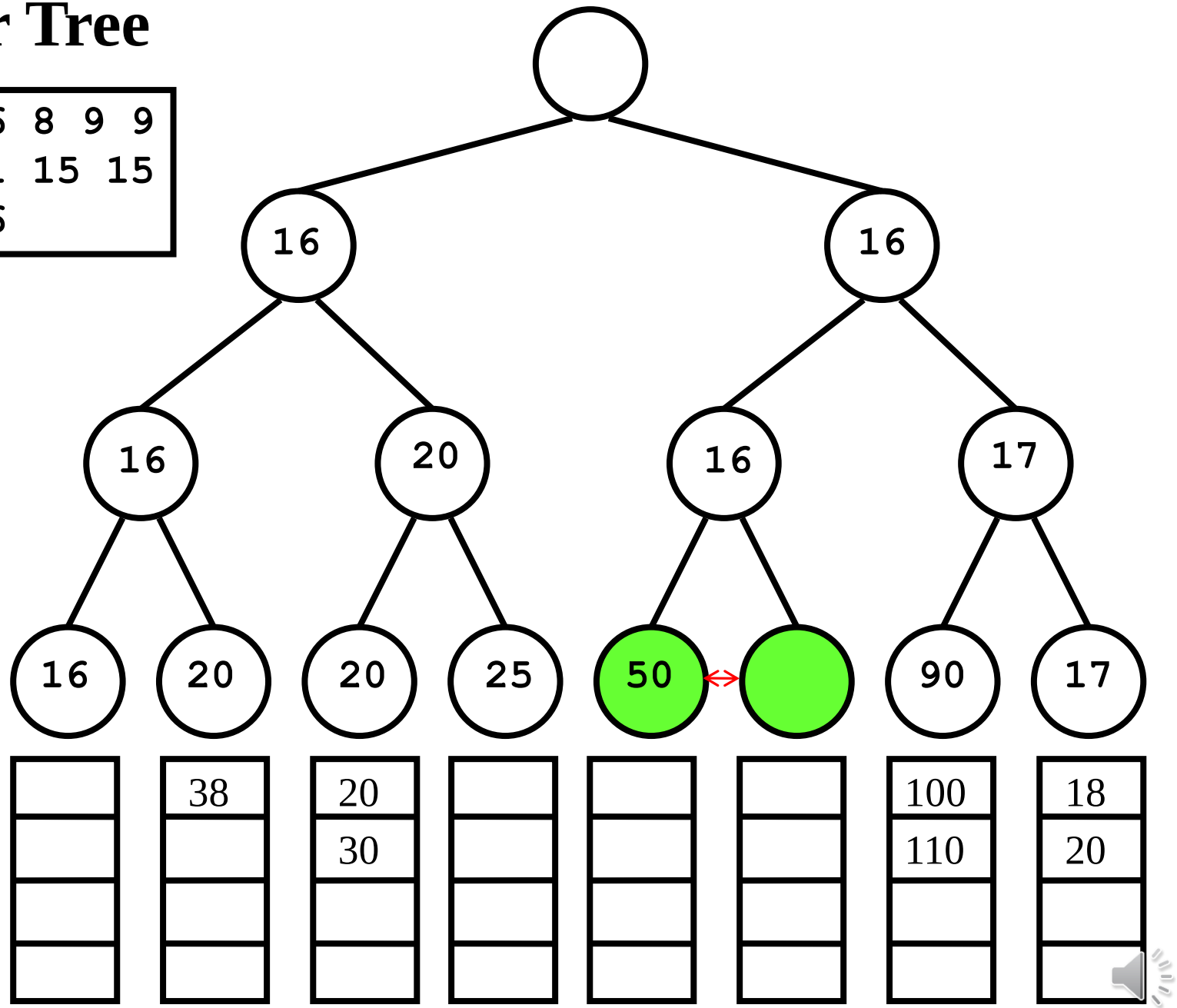
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



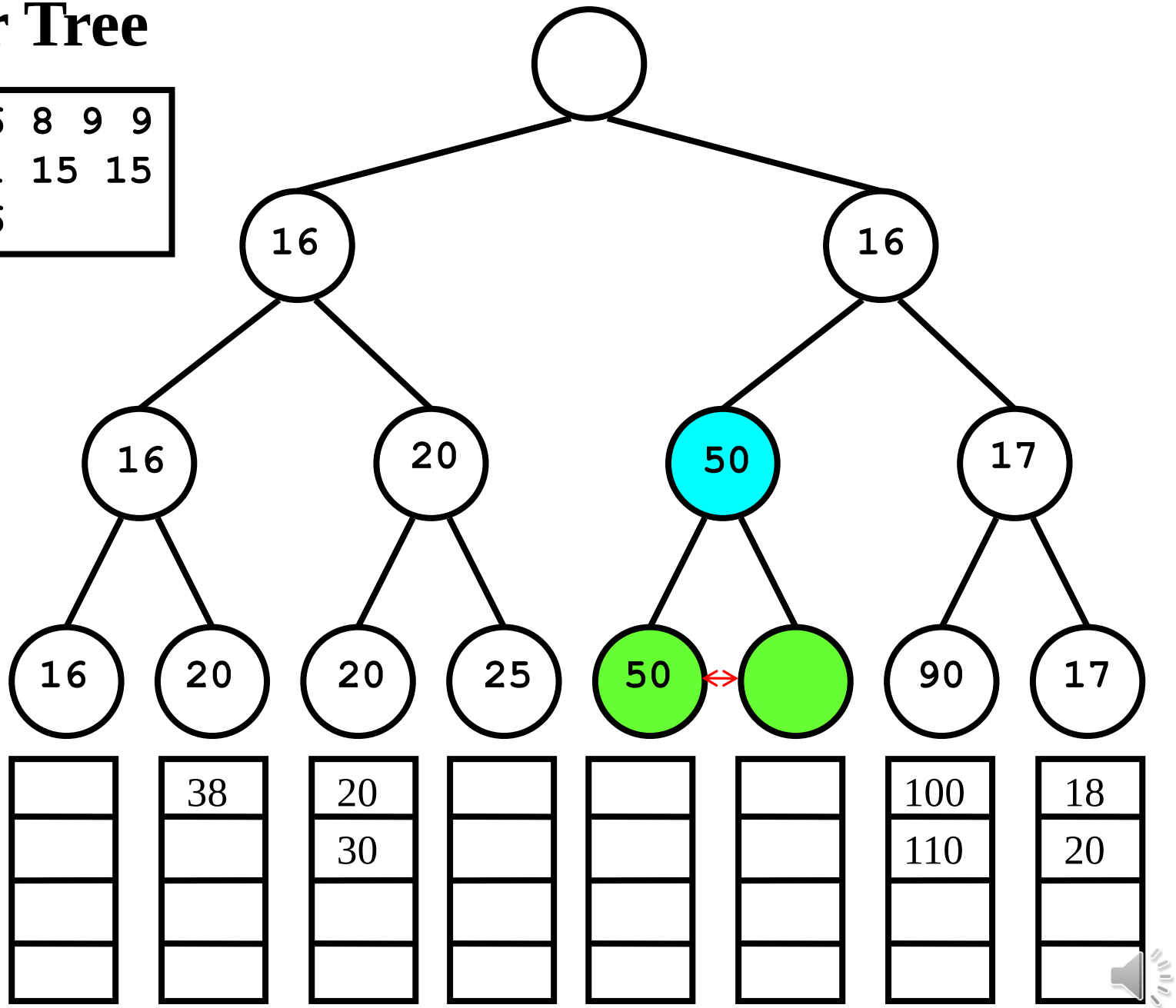
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



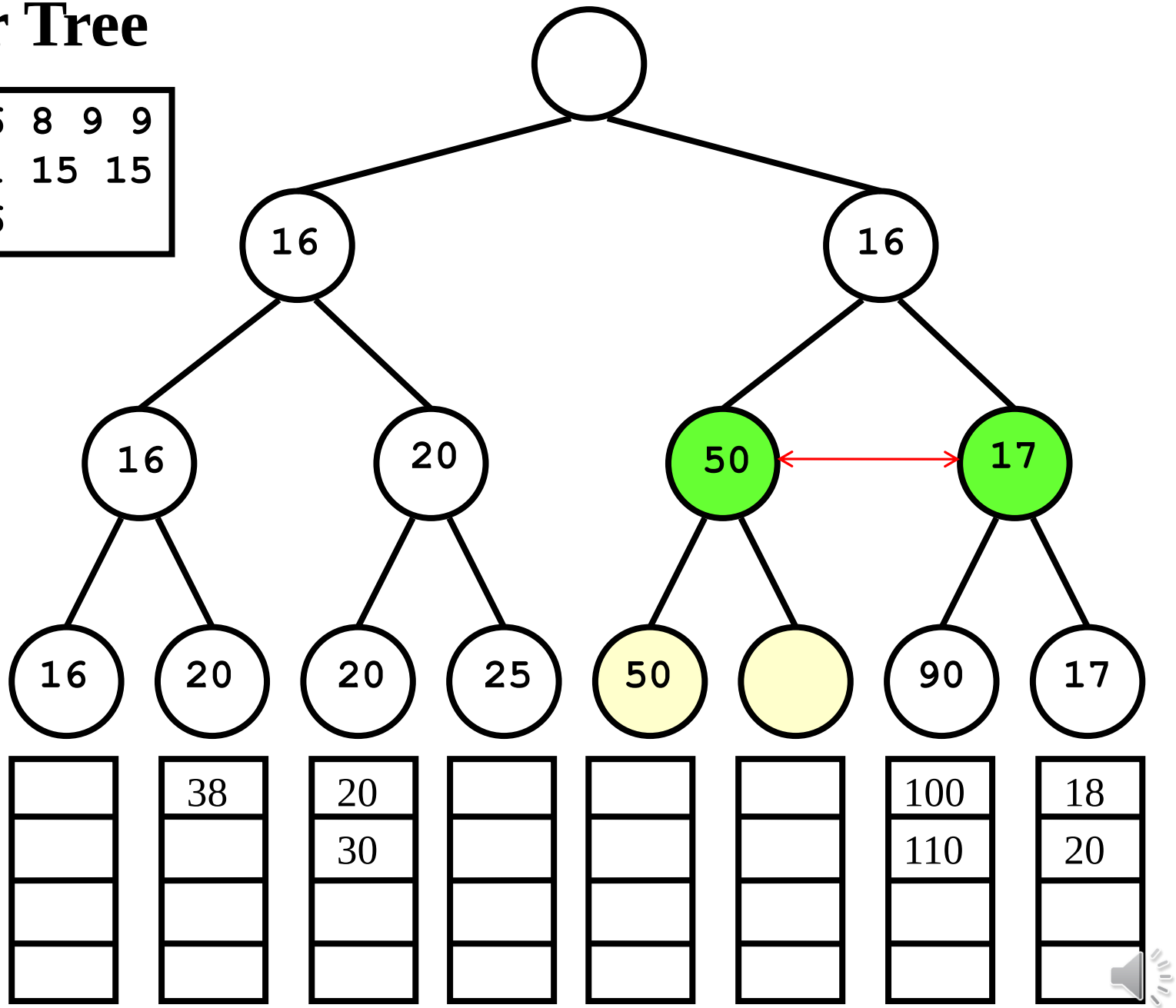
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



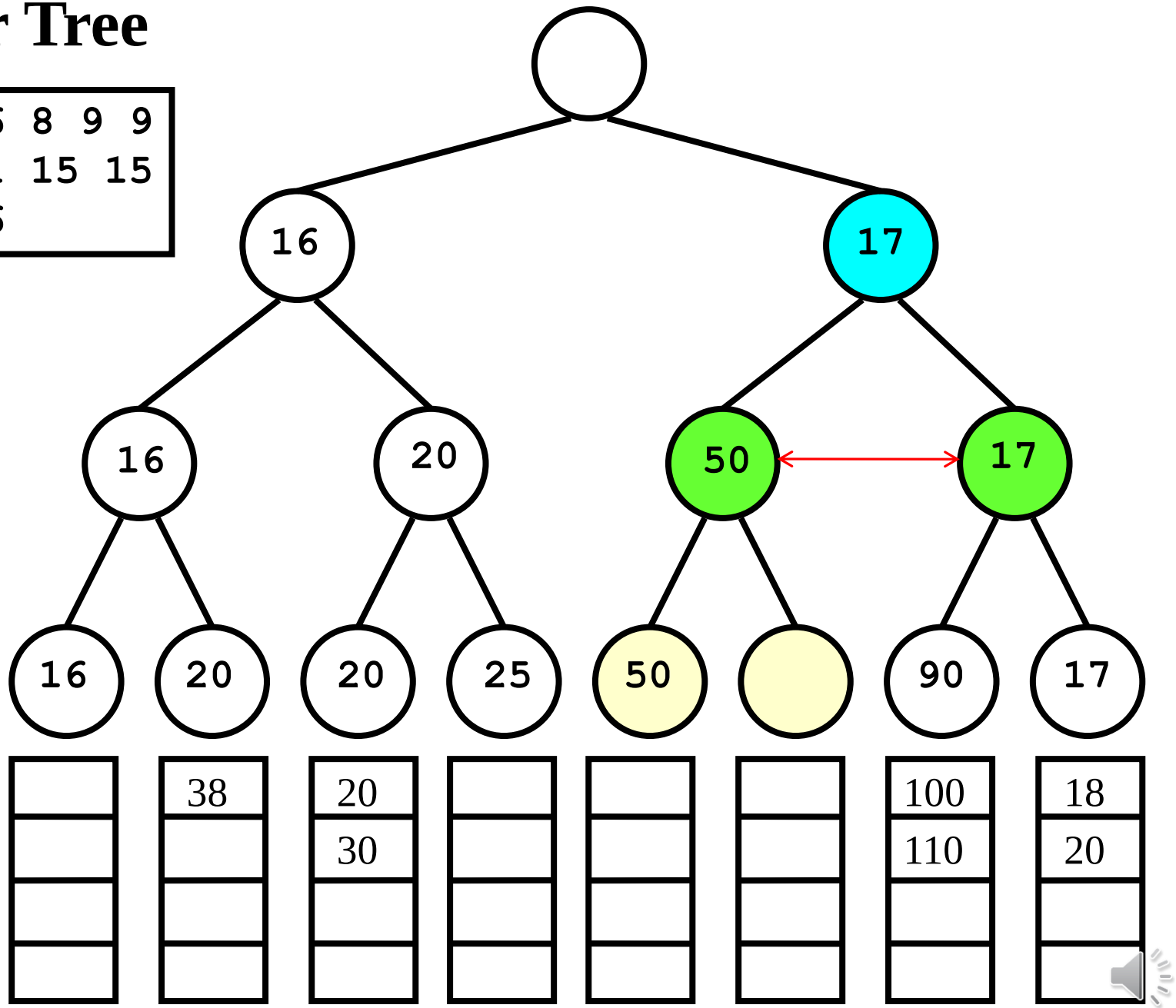
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



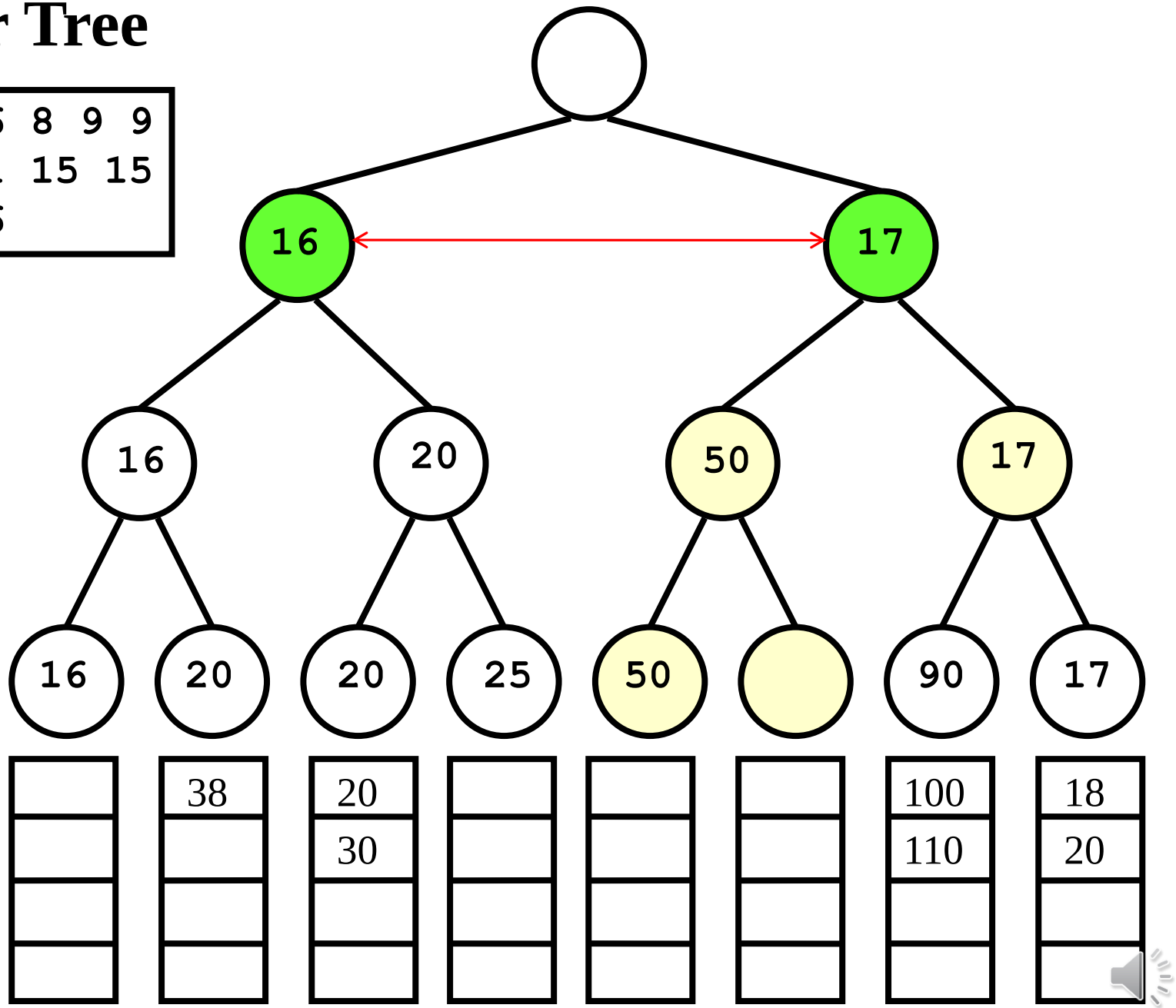
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



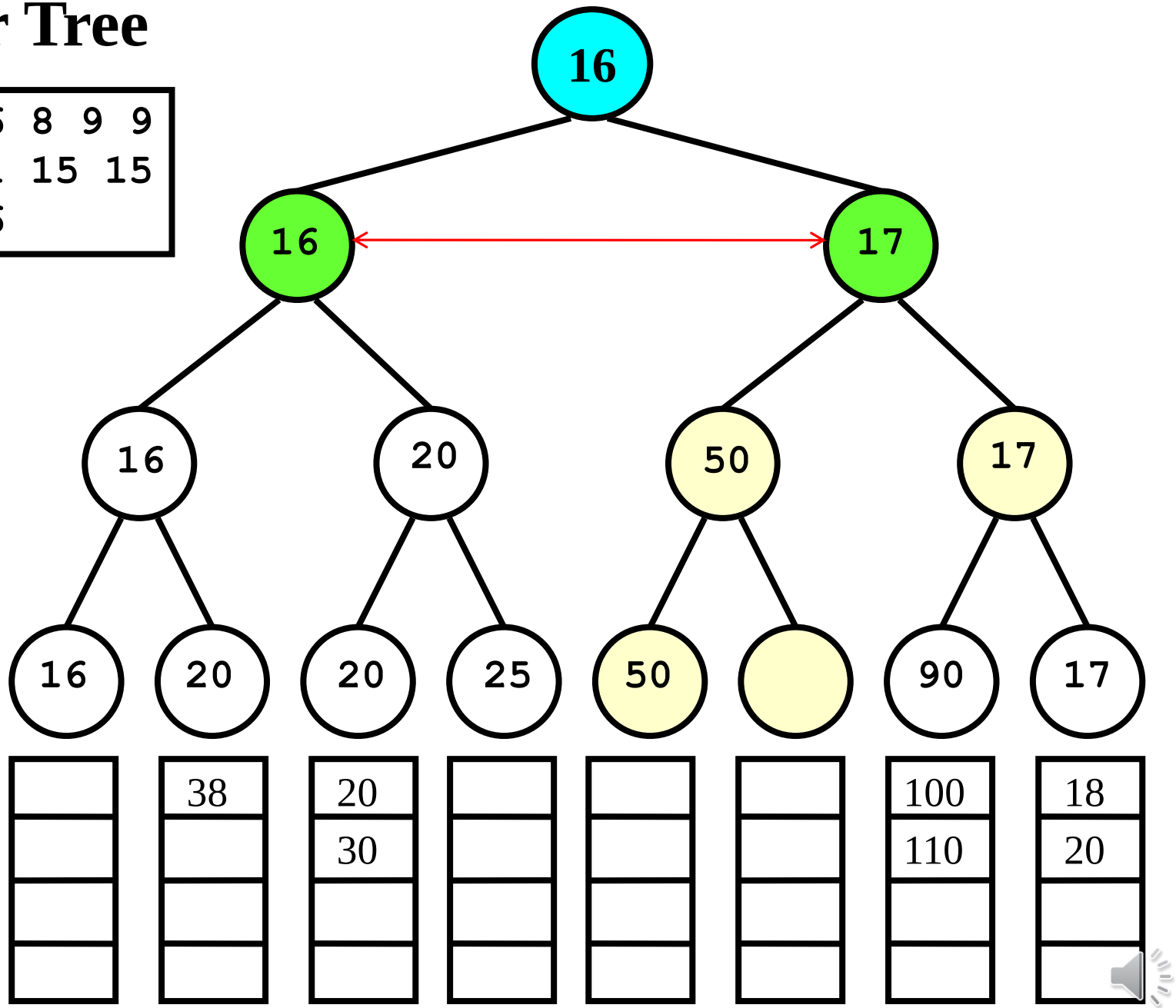
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



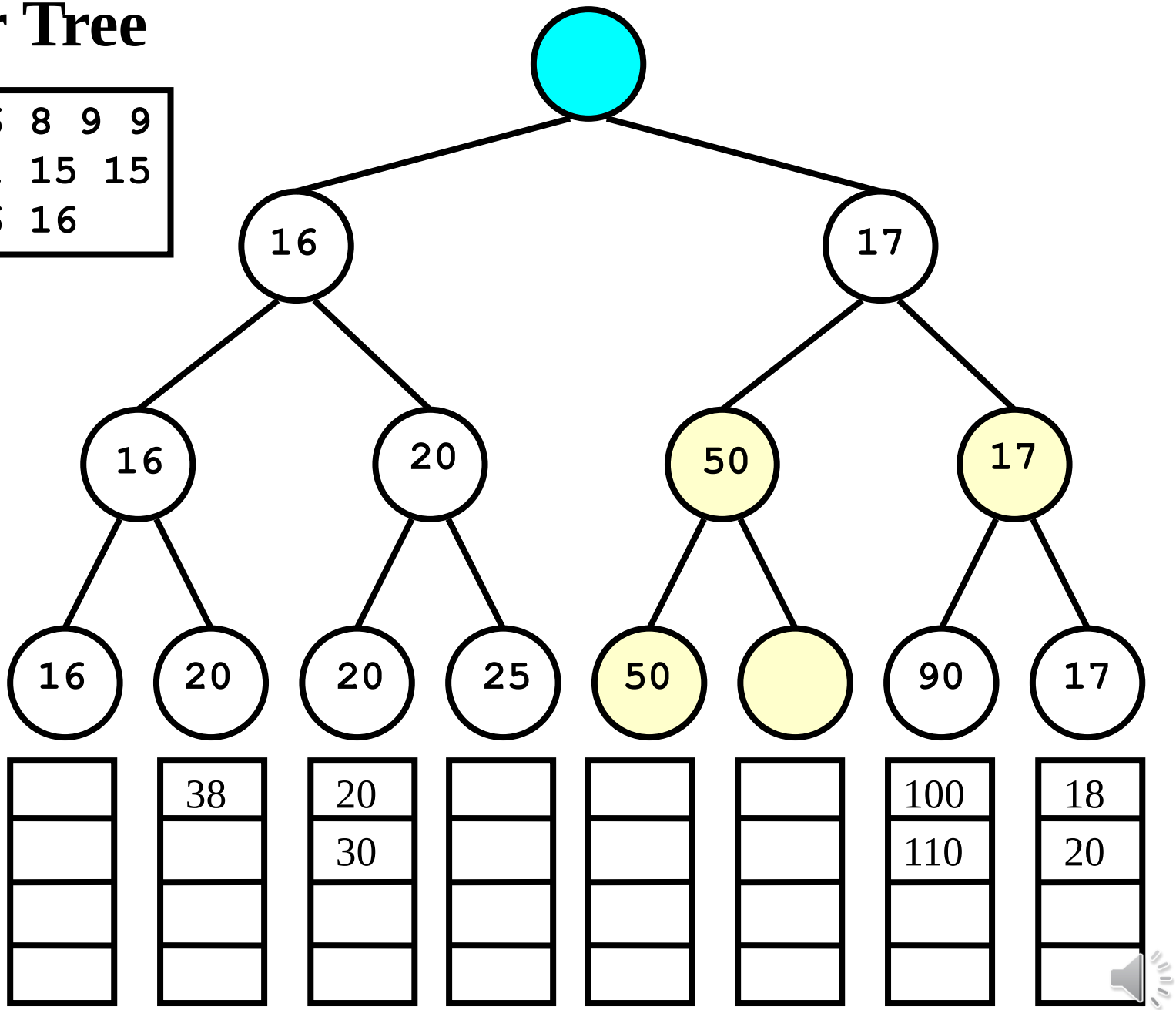
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16
```



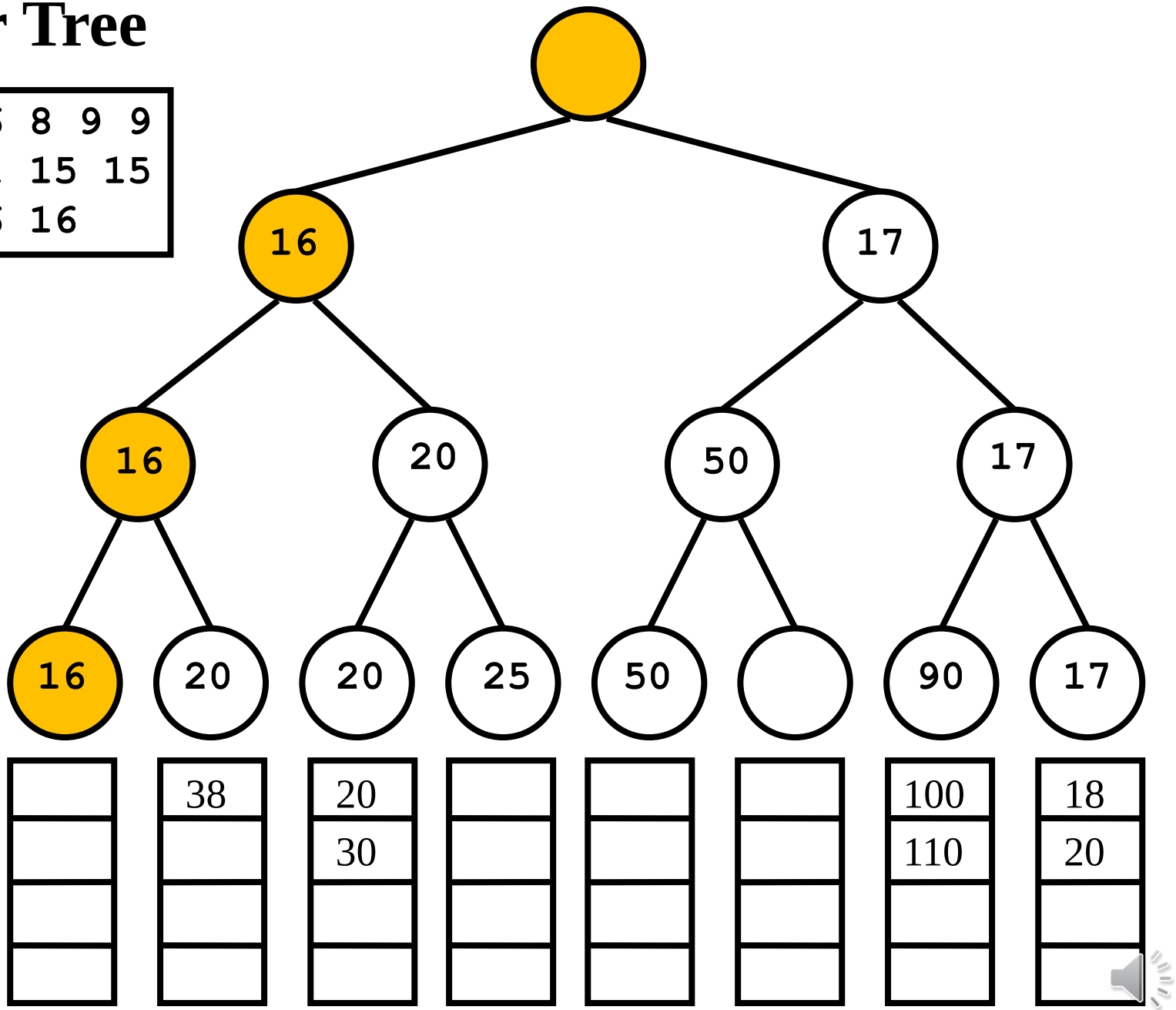
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16 16
```



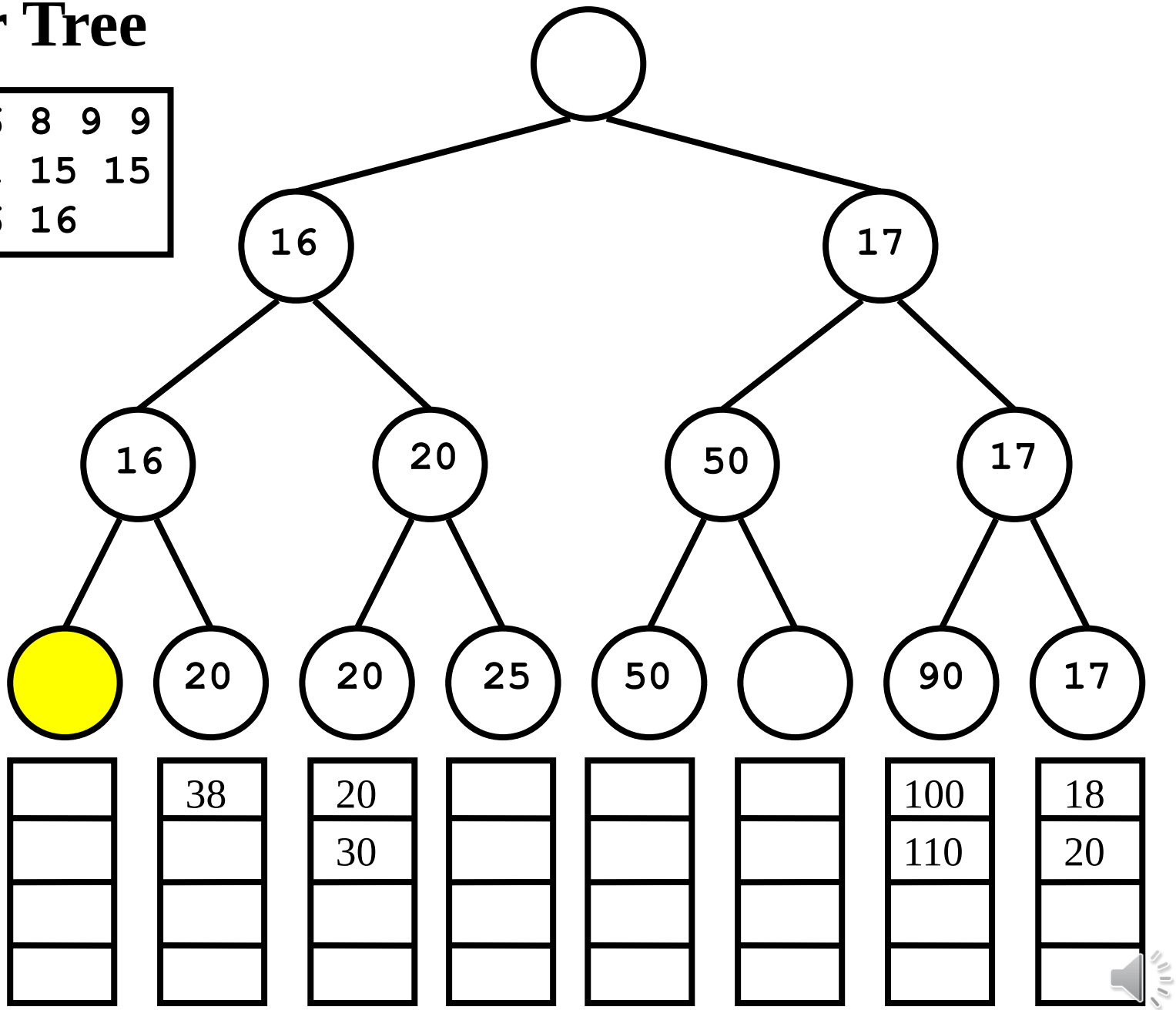
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16 16
```



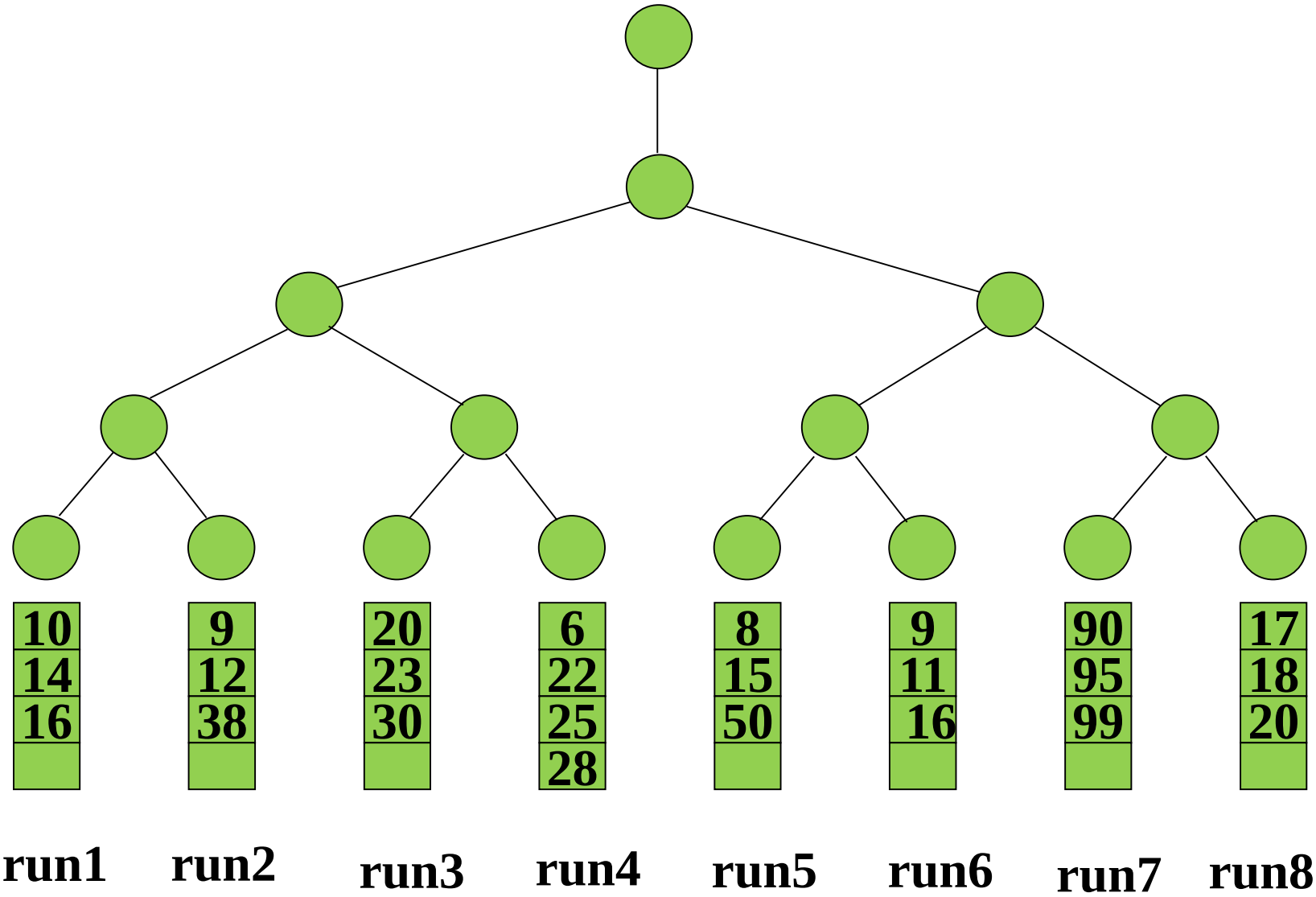
Winner Tree

```
Output: 6 8 9 9
    10 11 15 15
    15 16 16
```



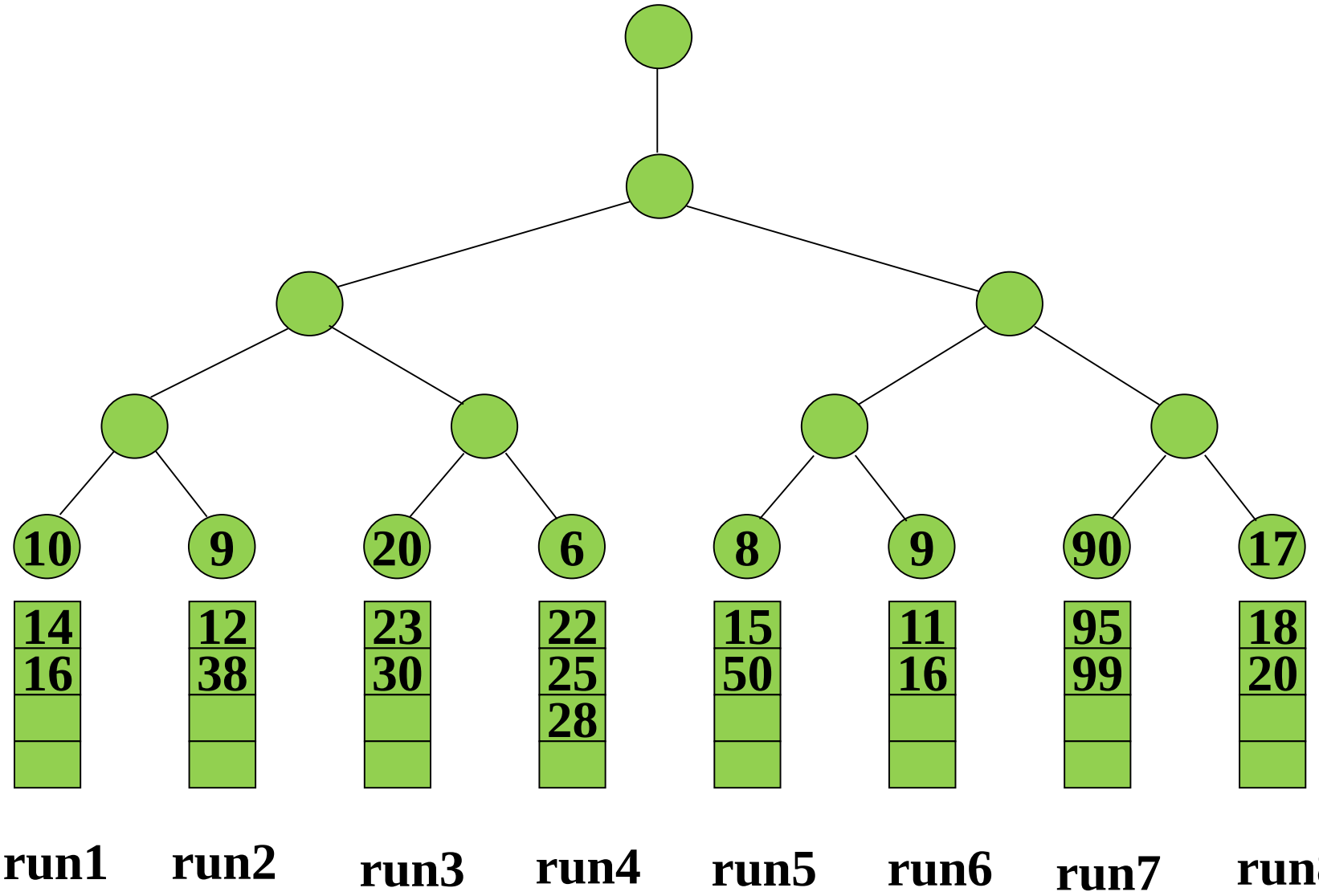
Loser Tree

Run:



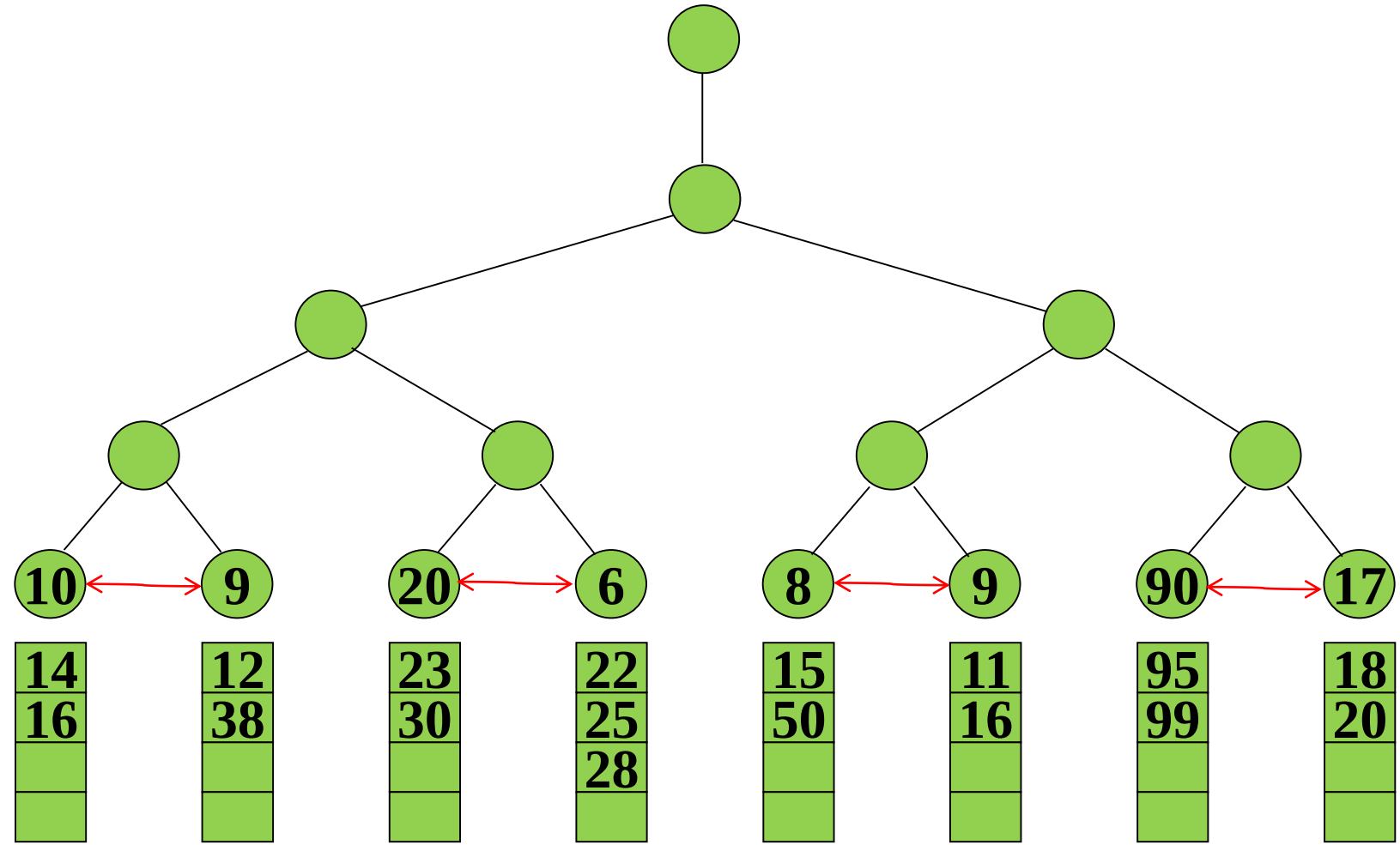
Loser Tree

Run:



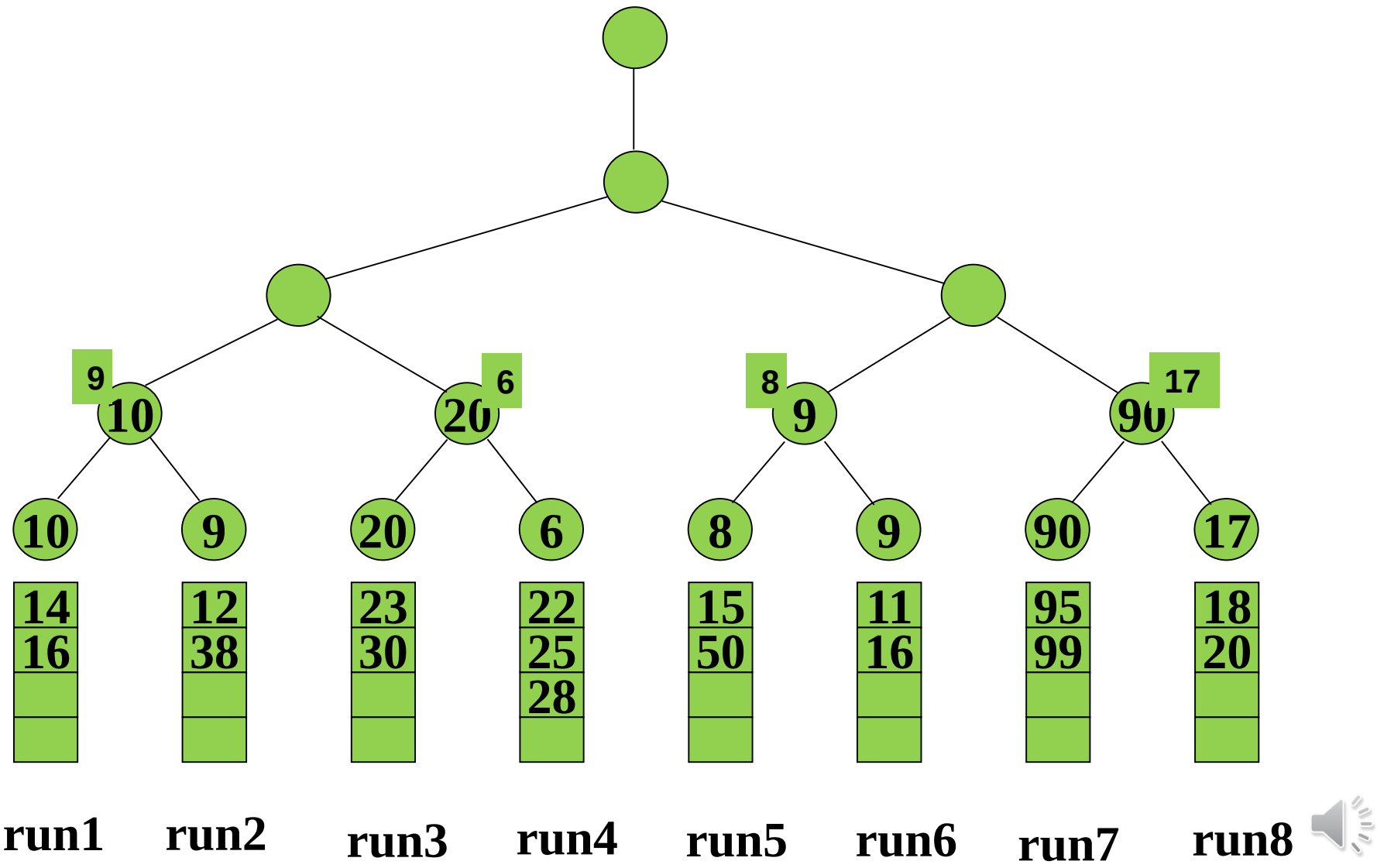
Loser Tree

Run:



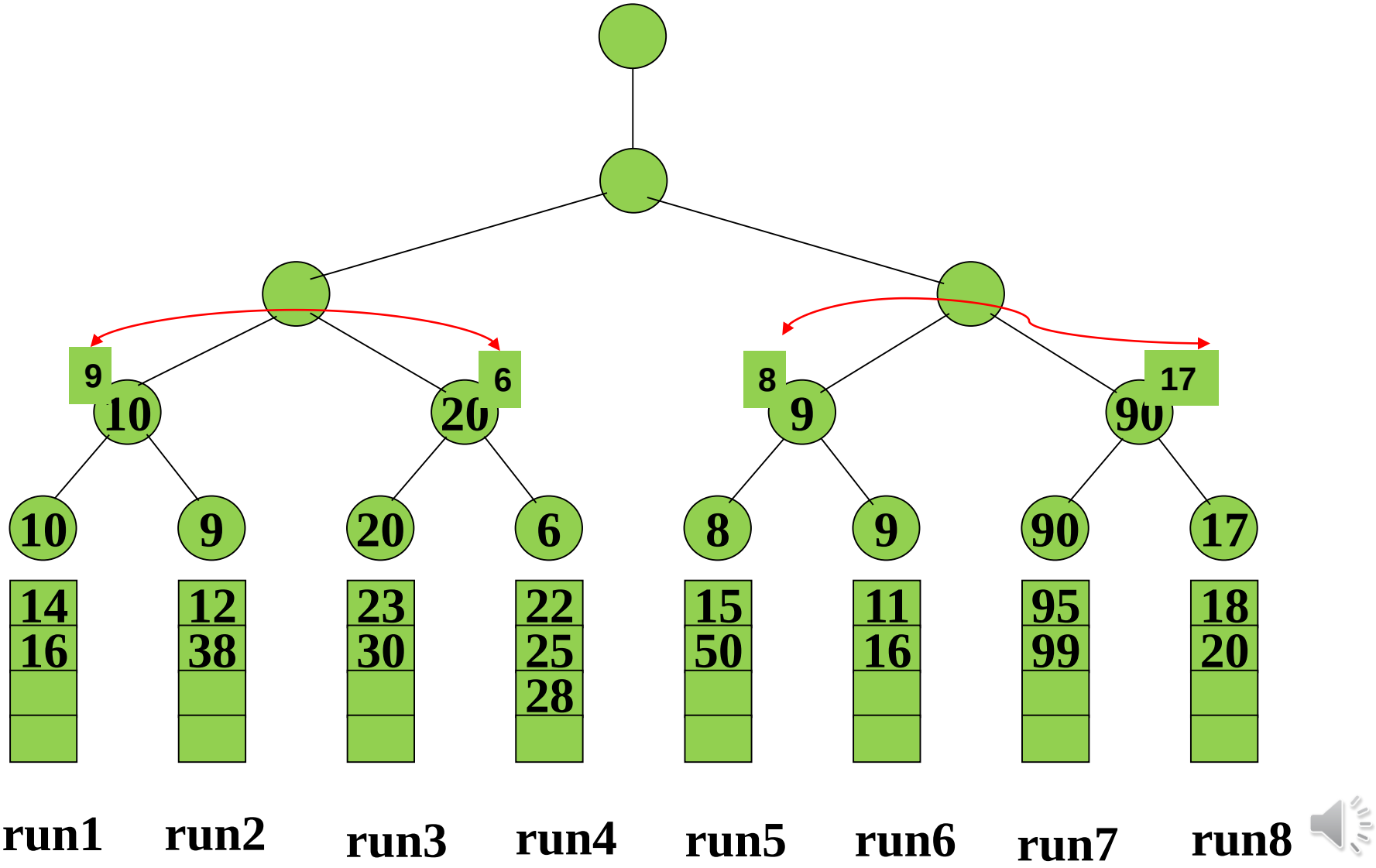
Loser Tree

Run:



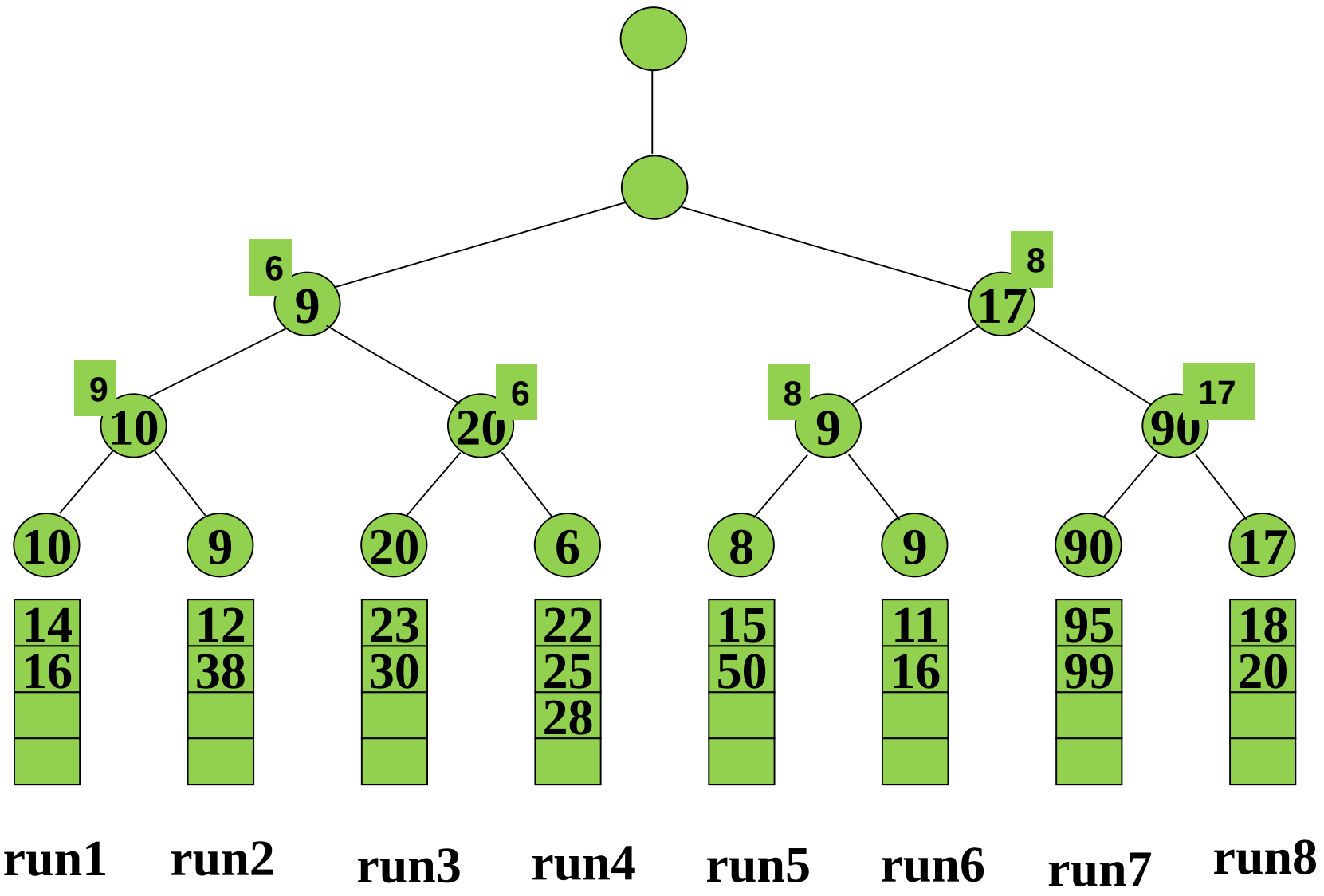
Loser Tree

Run:



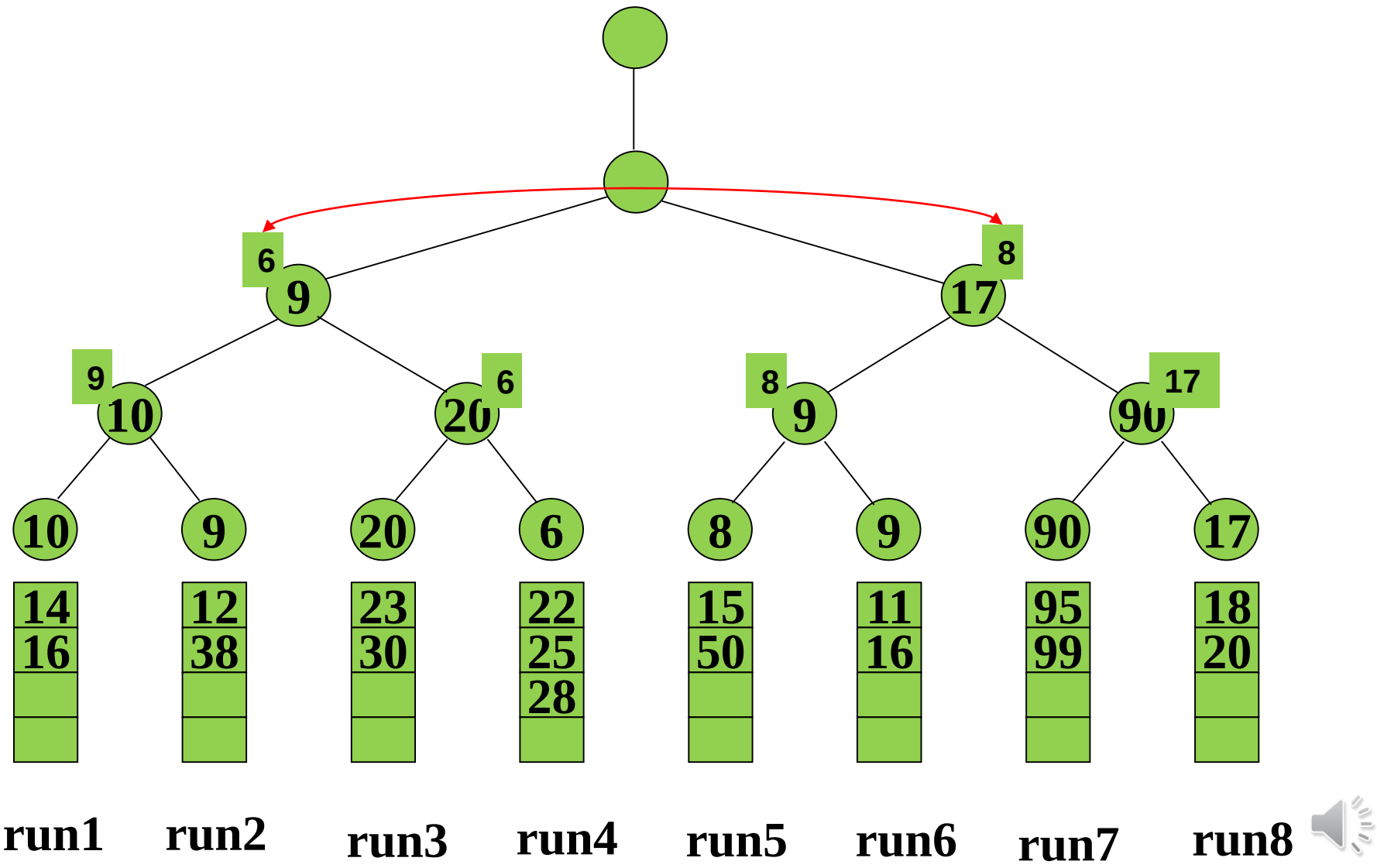
Loser Tree

Run:



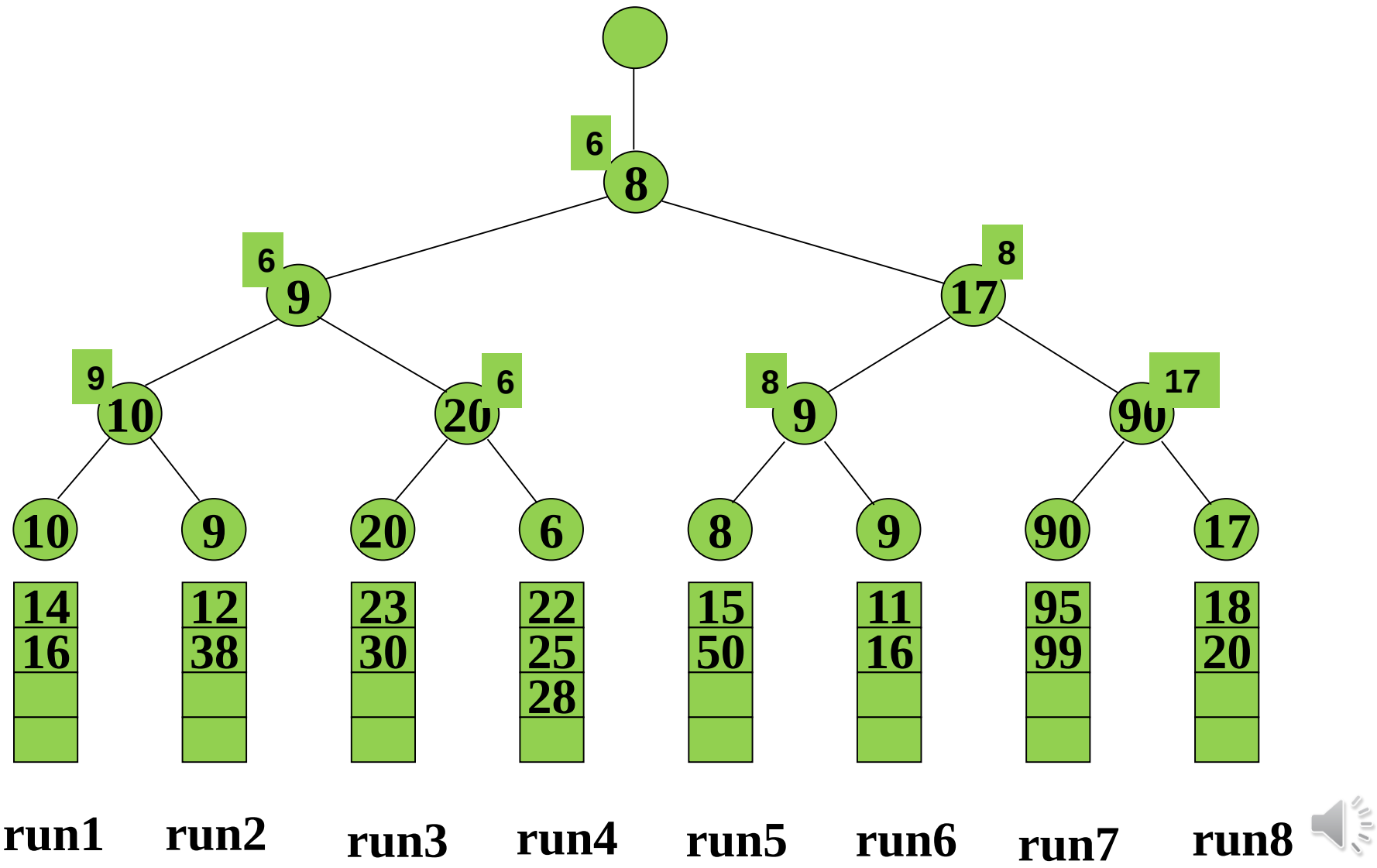
Loser Tree

Run:



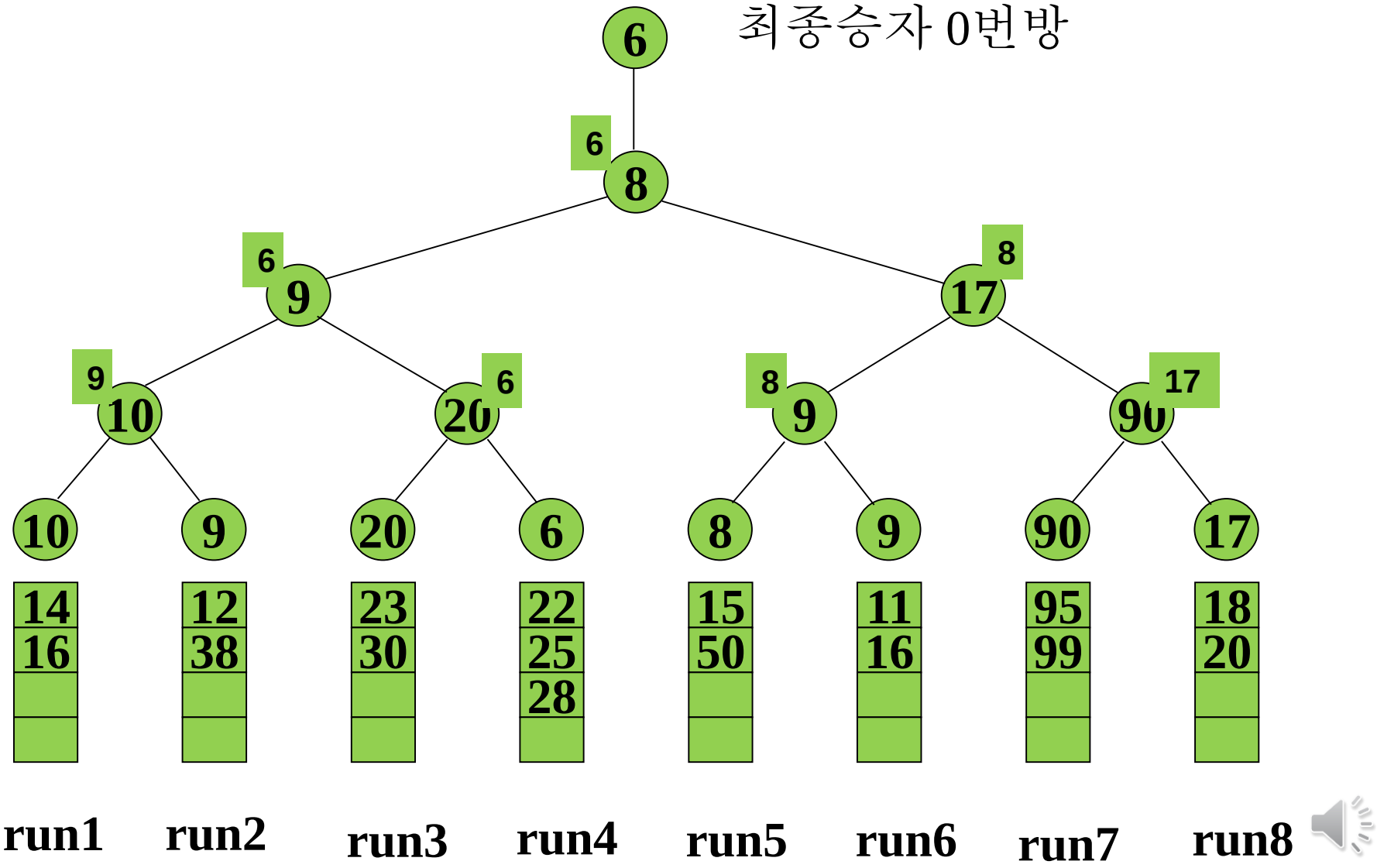
Loser Tree

Run:



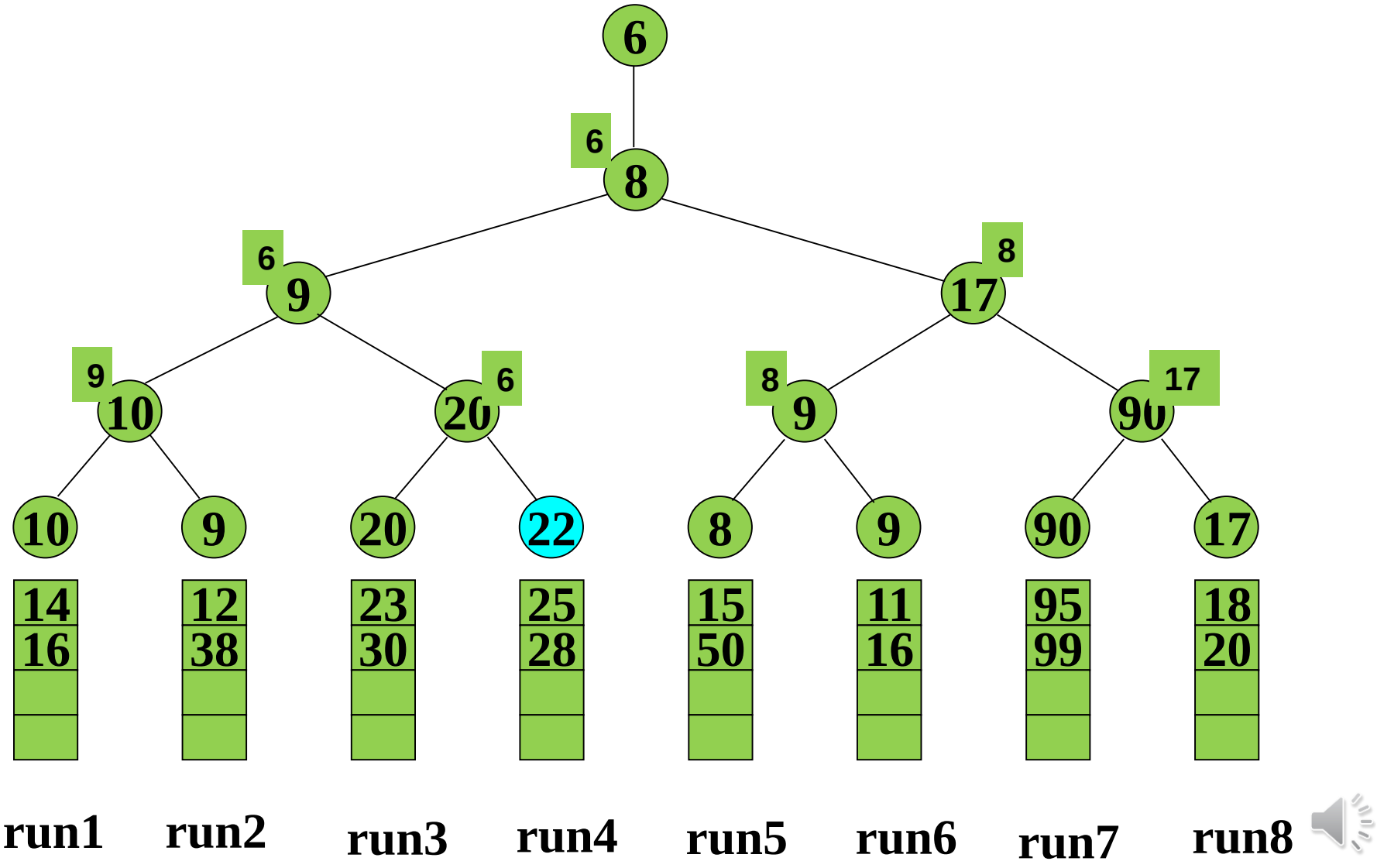
Loser Tree(트리완성)

Run:6



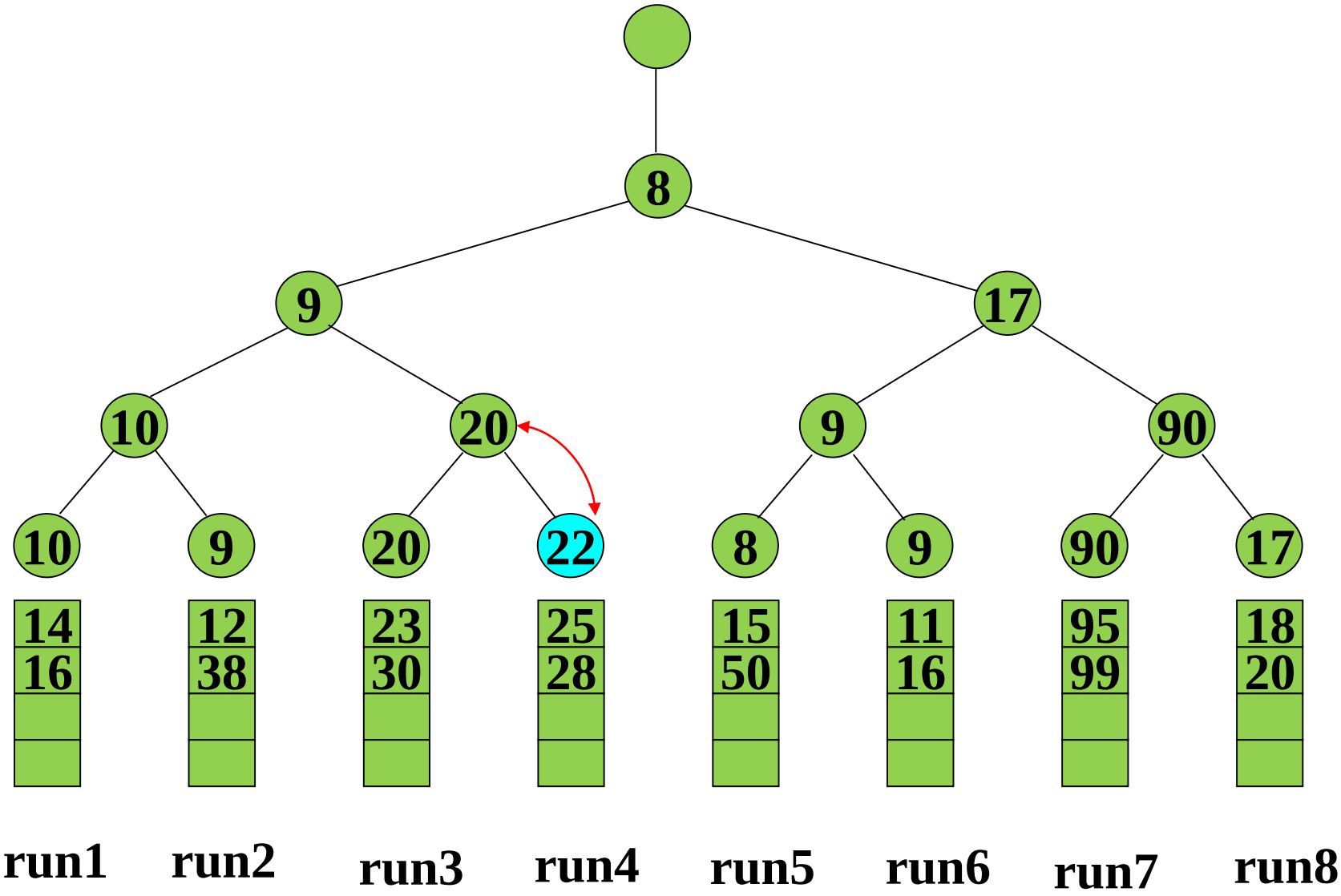
Loser Tree

Run : 6



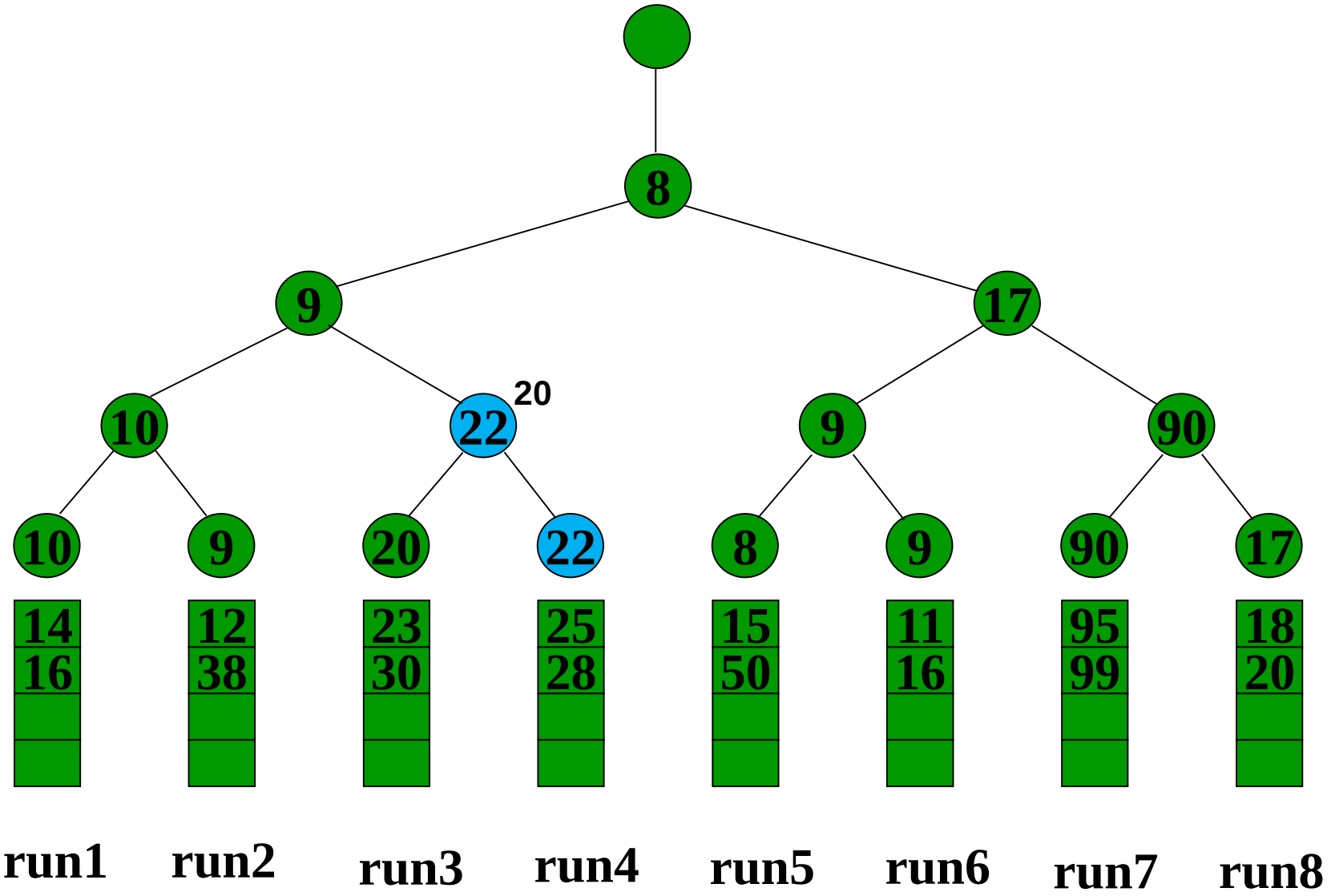
Loser Tree

Run : 6



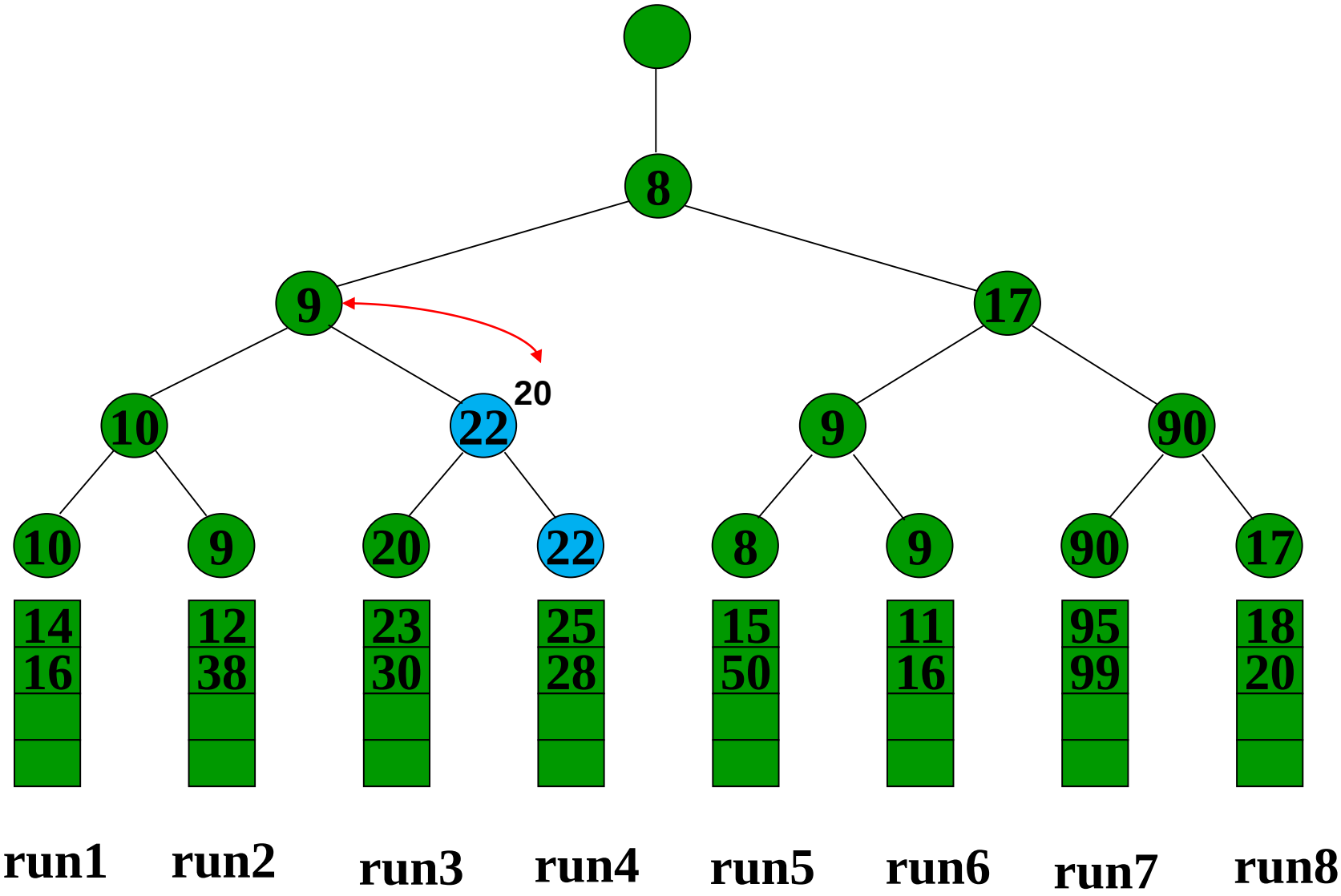
Loser Tree

Run:6



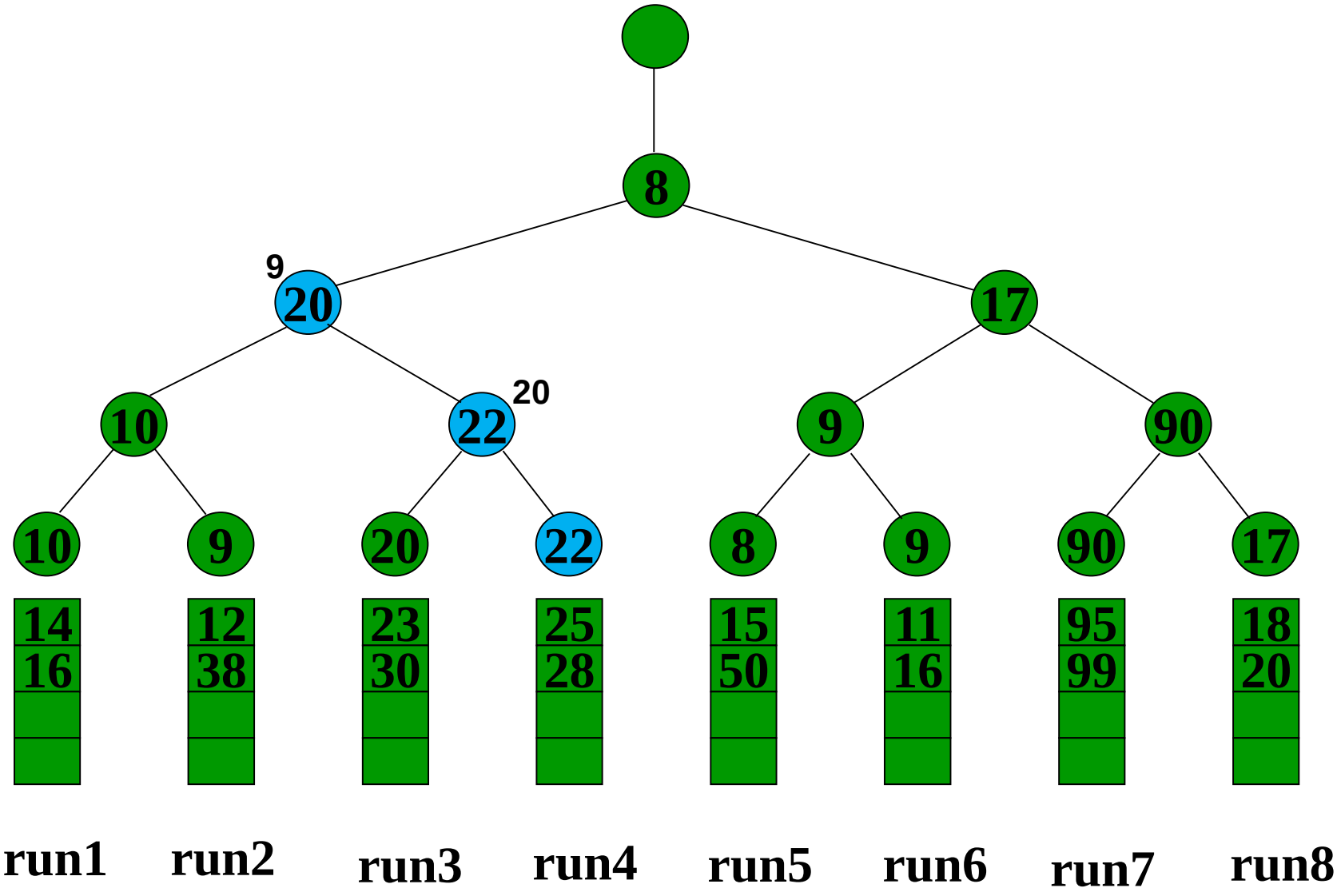
Loser Tree

Run:6



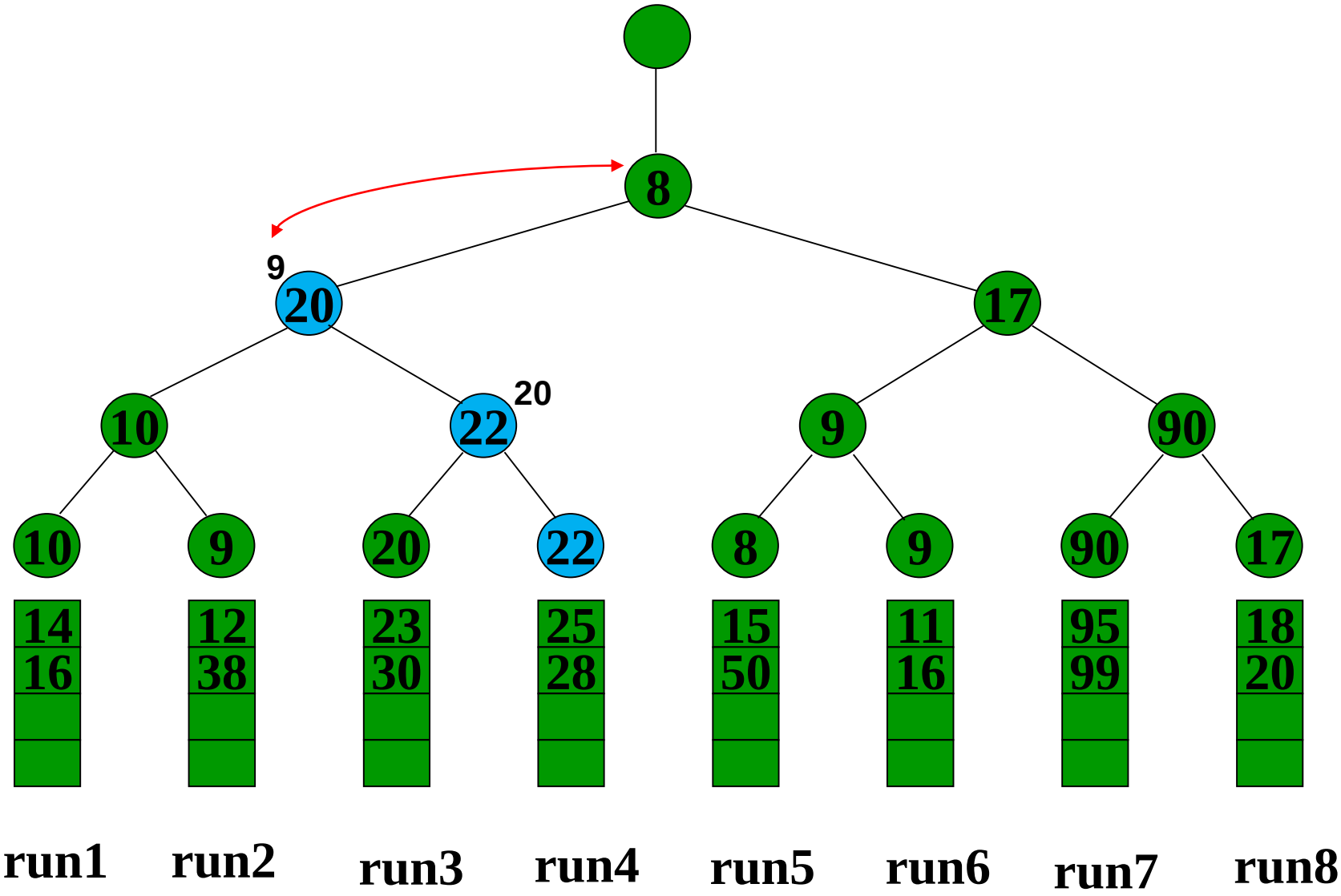
Loser Tree

Run:6



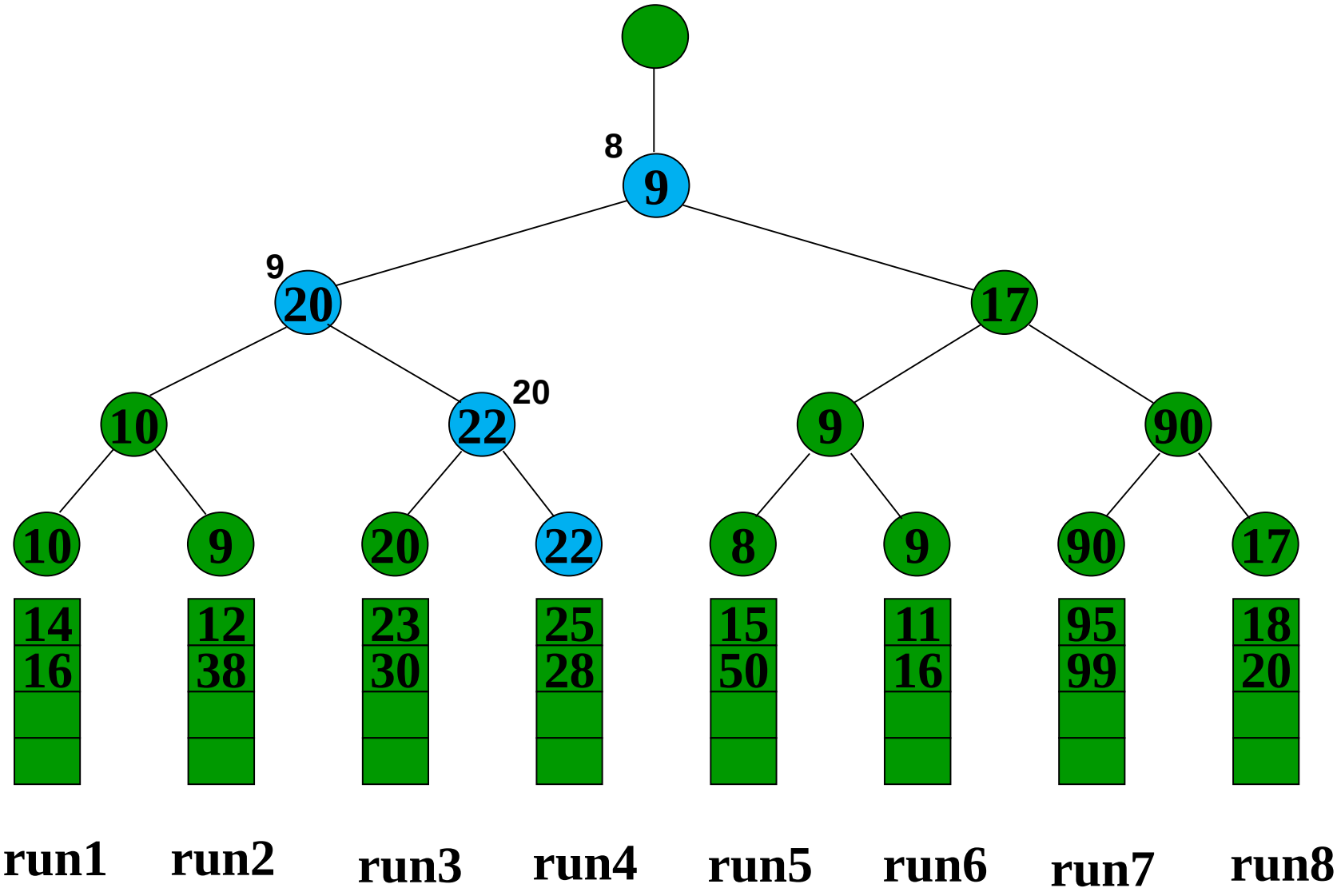
Loser Tree

Run:6



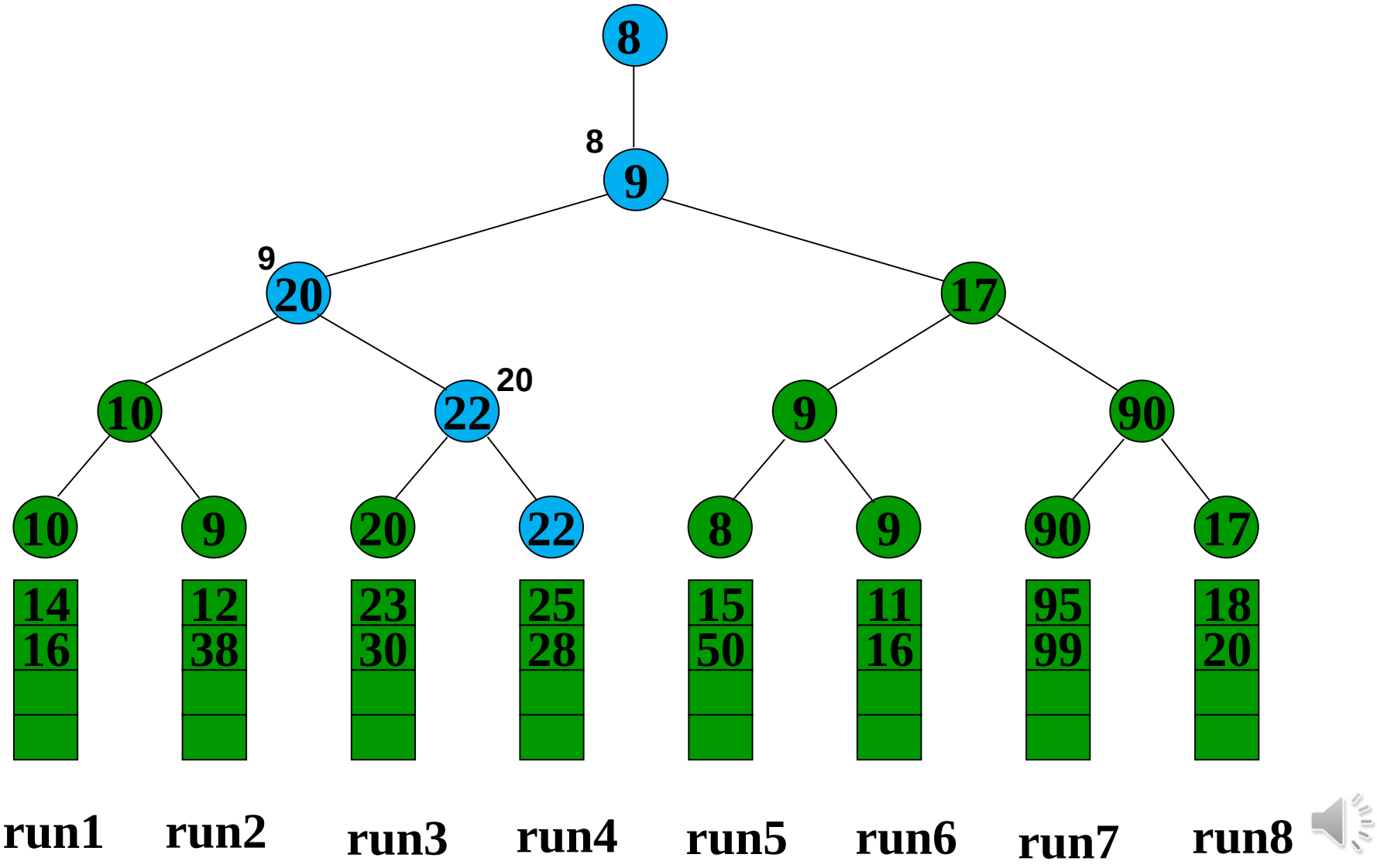
Loser Tree

Run:6



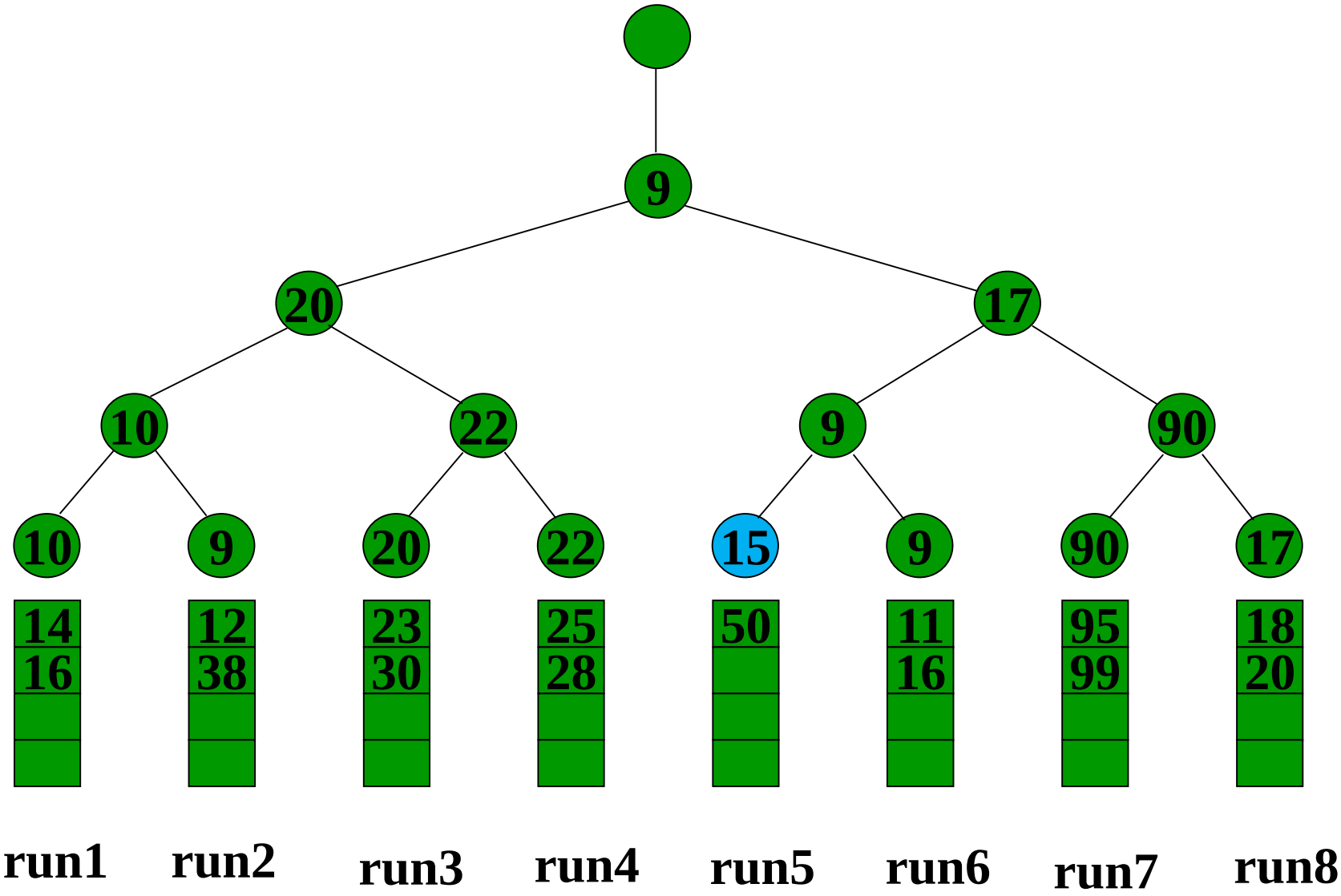
Loser Tree

Run:6 8



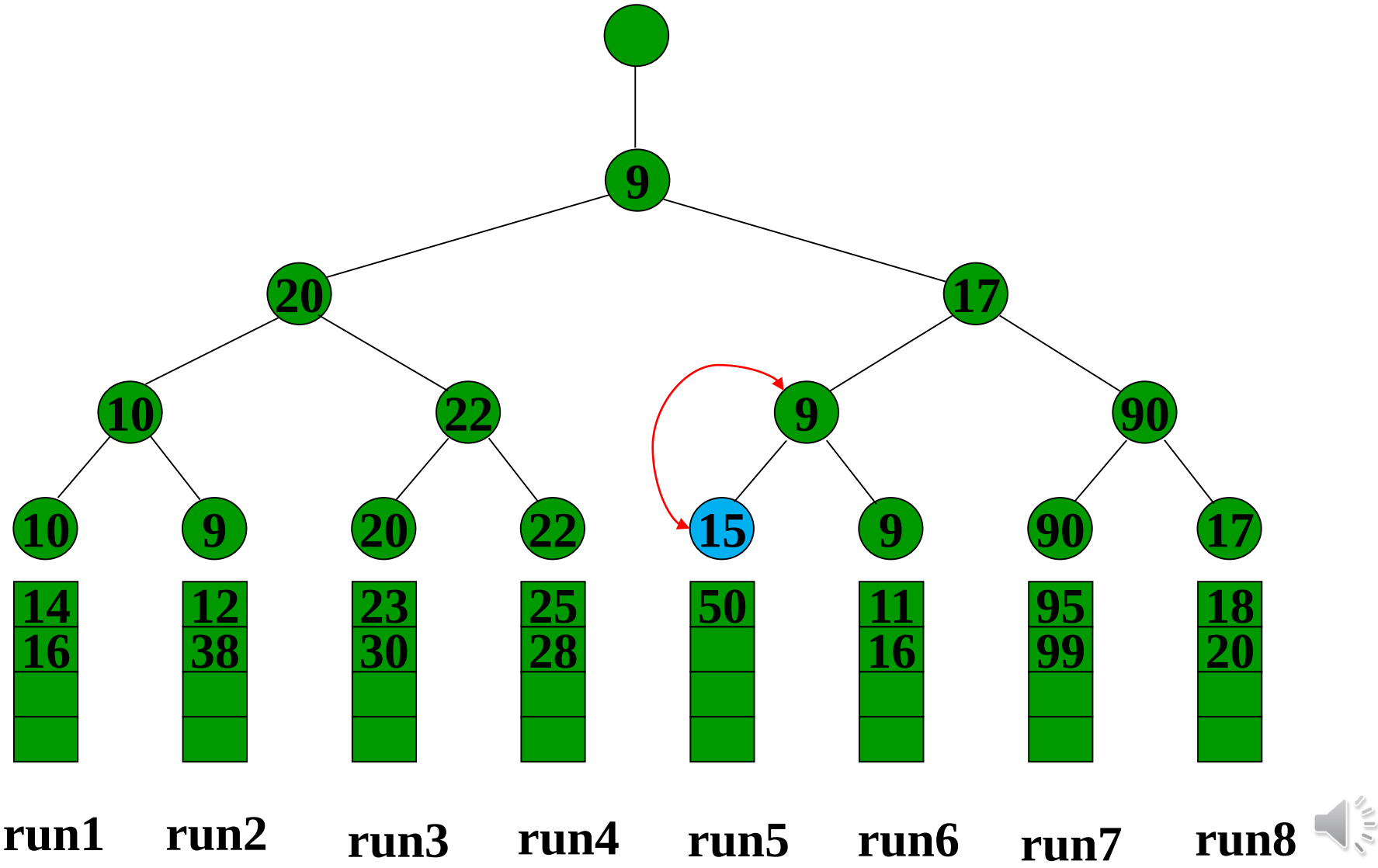
Loser Tree

Run:6 8



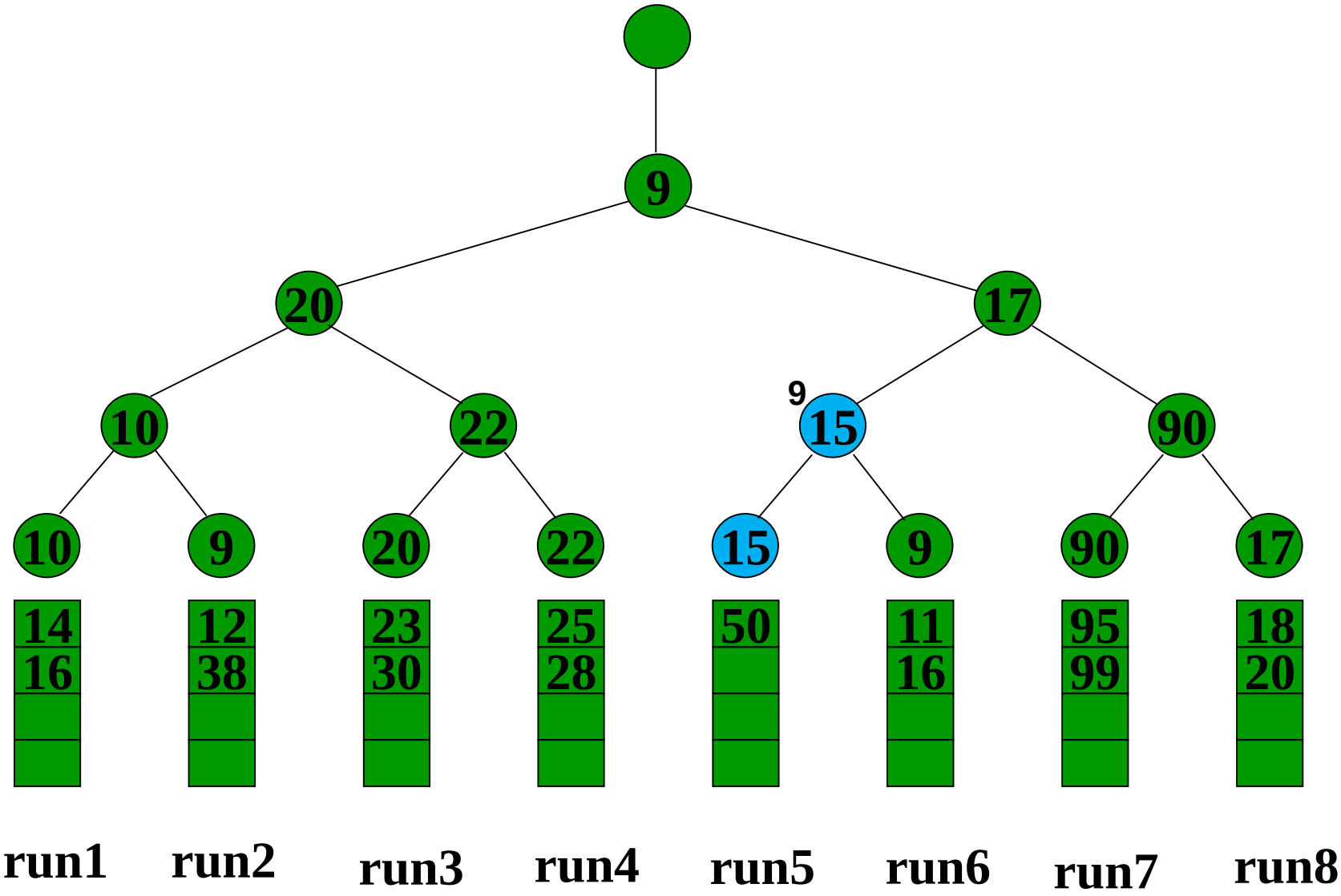
Loser Tree

Run:6 8



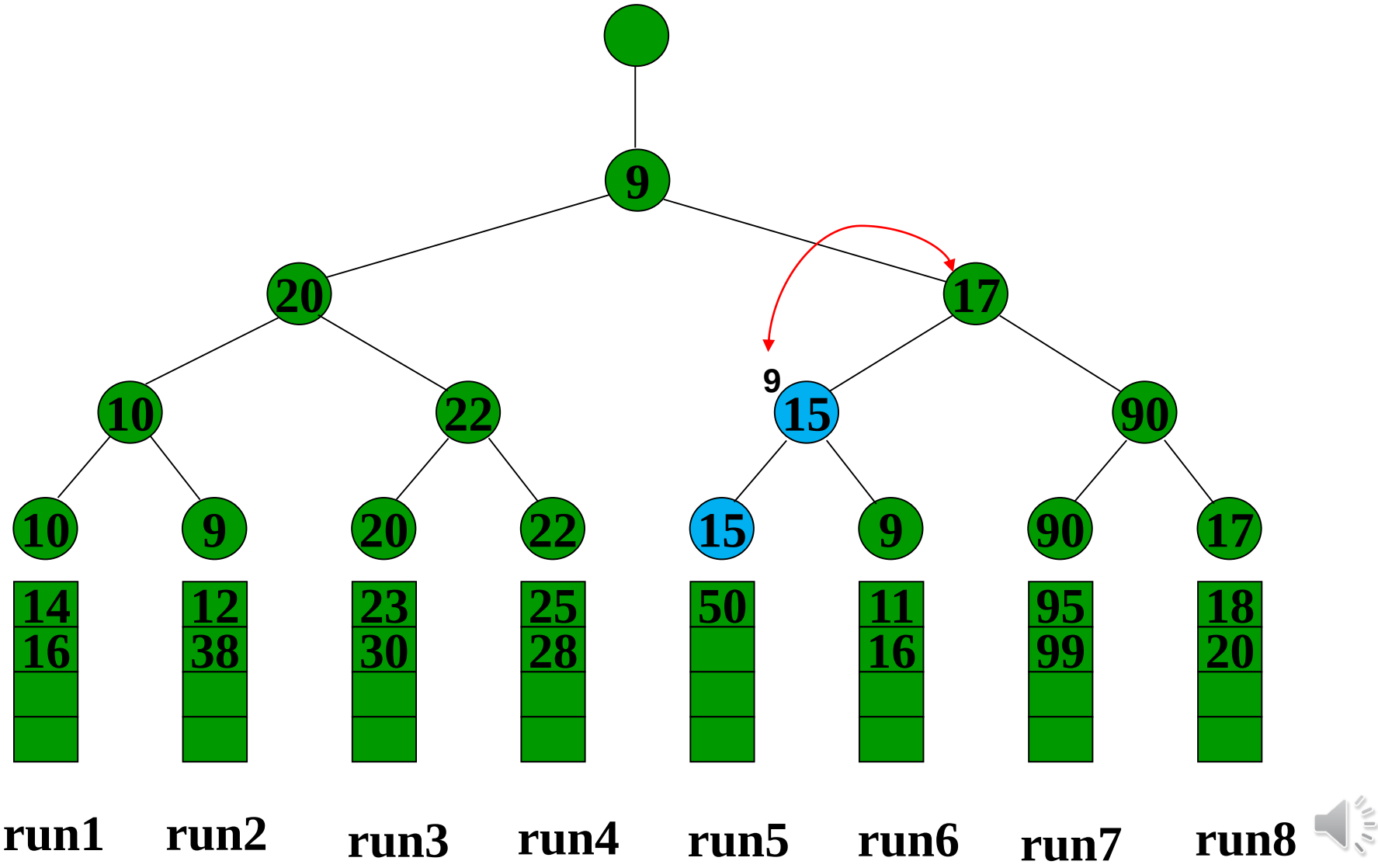
Loser Tree

Run:6 8



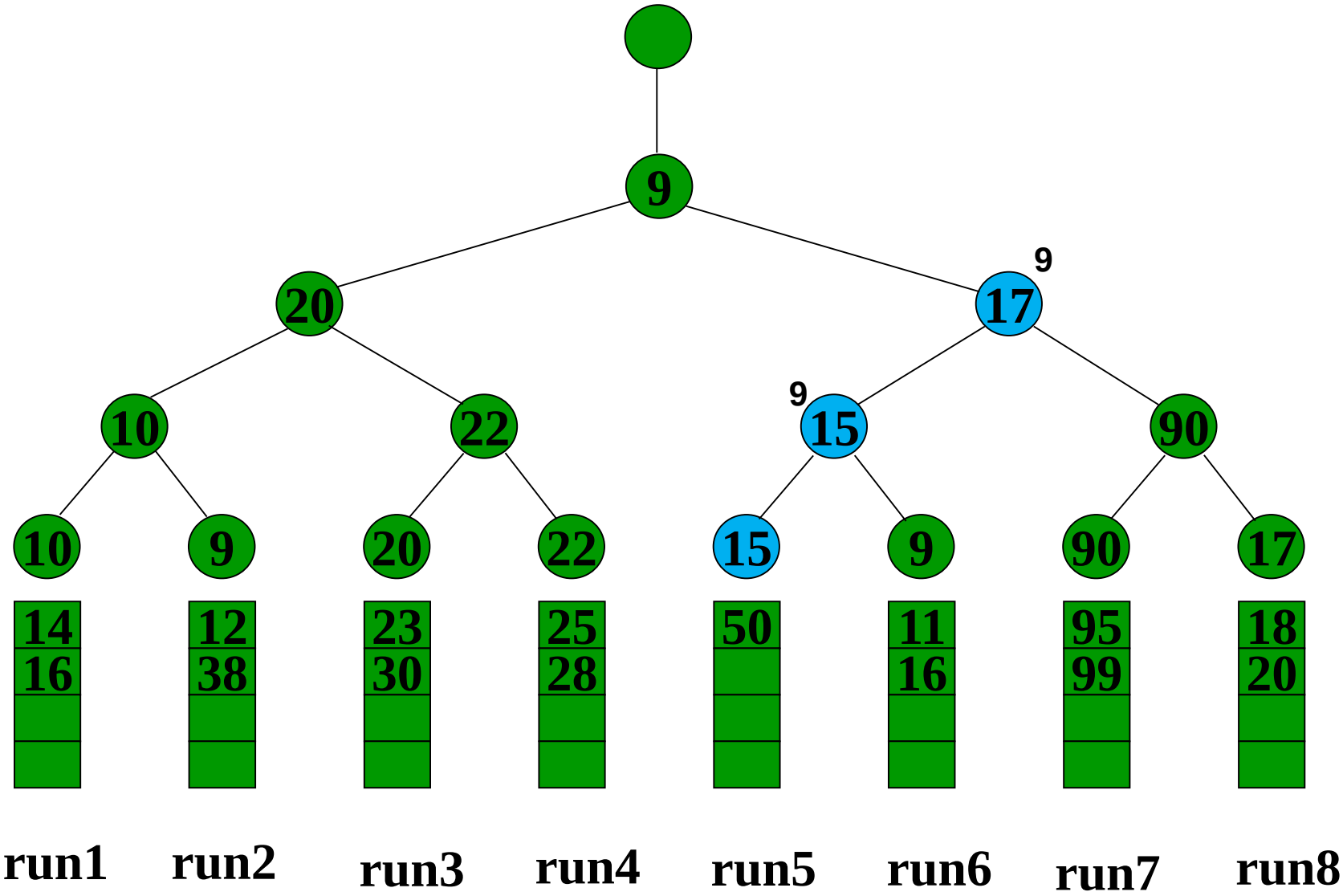
Loser Tree

Run:6 8



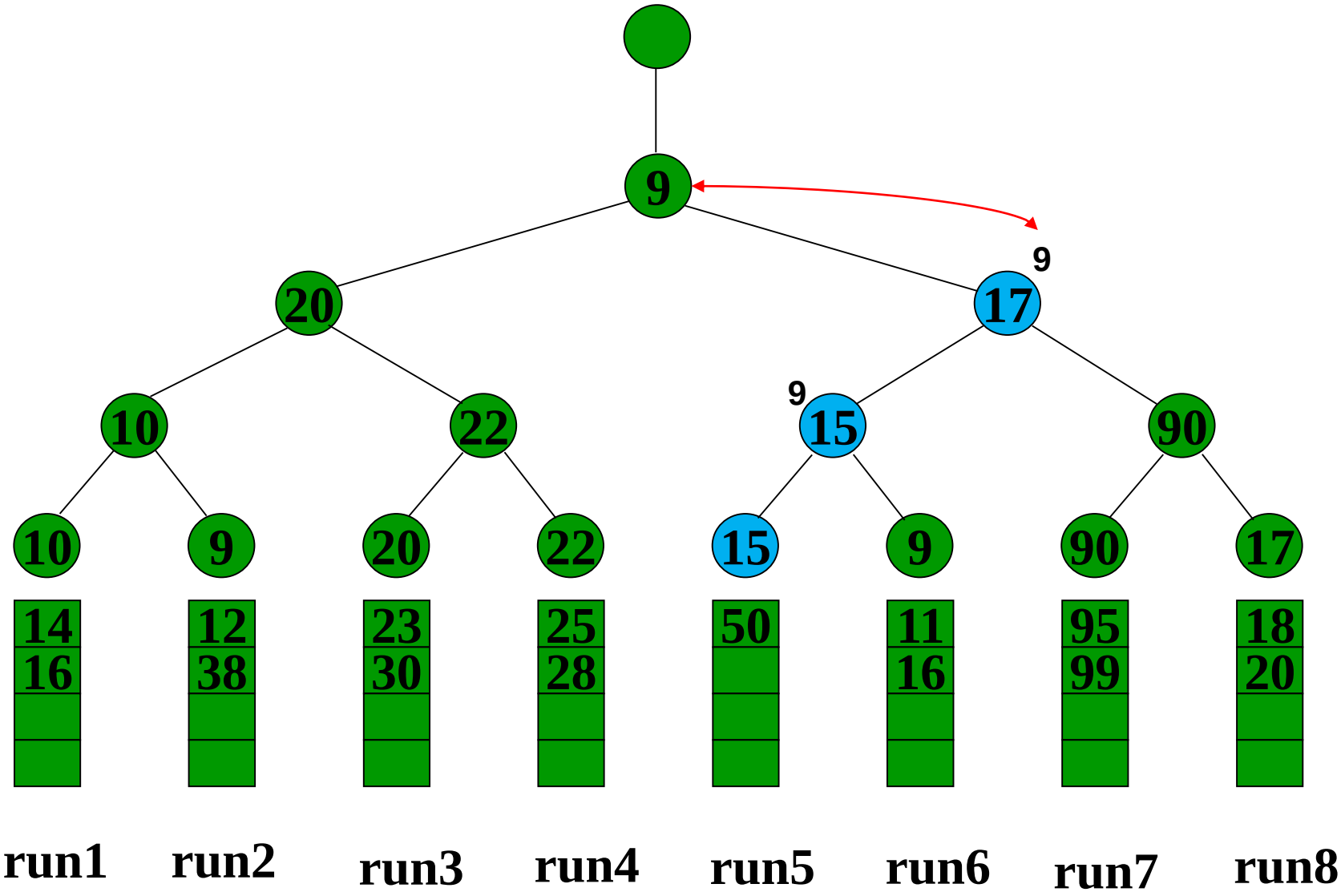
Loser Tree

Run:6 8



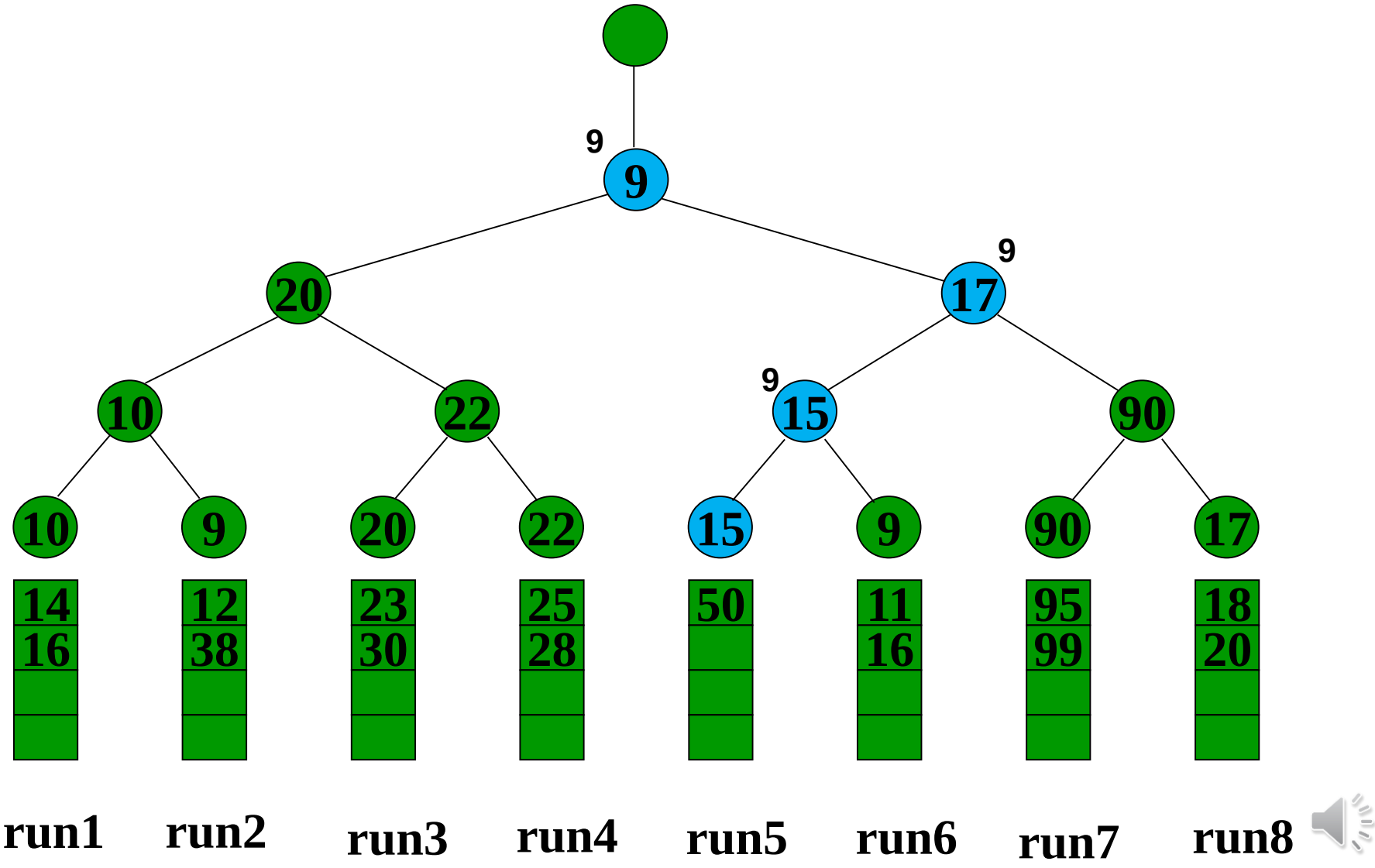
Loser Tree

Run:6 8



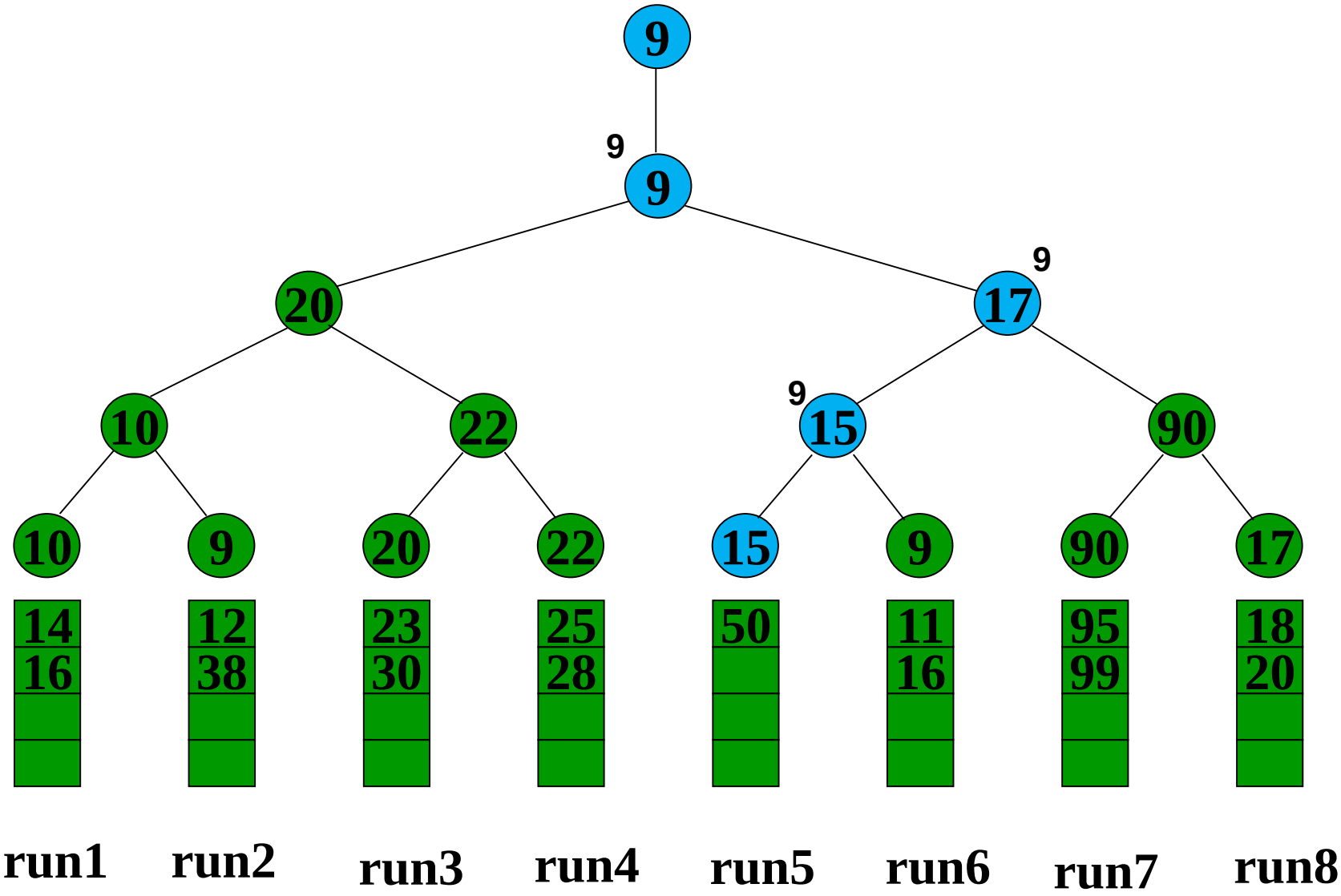
Loser Tree

Run:6 8



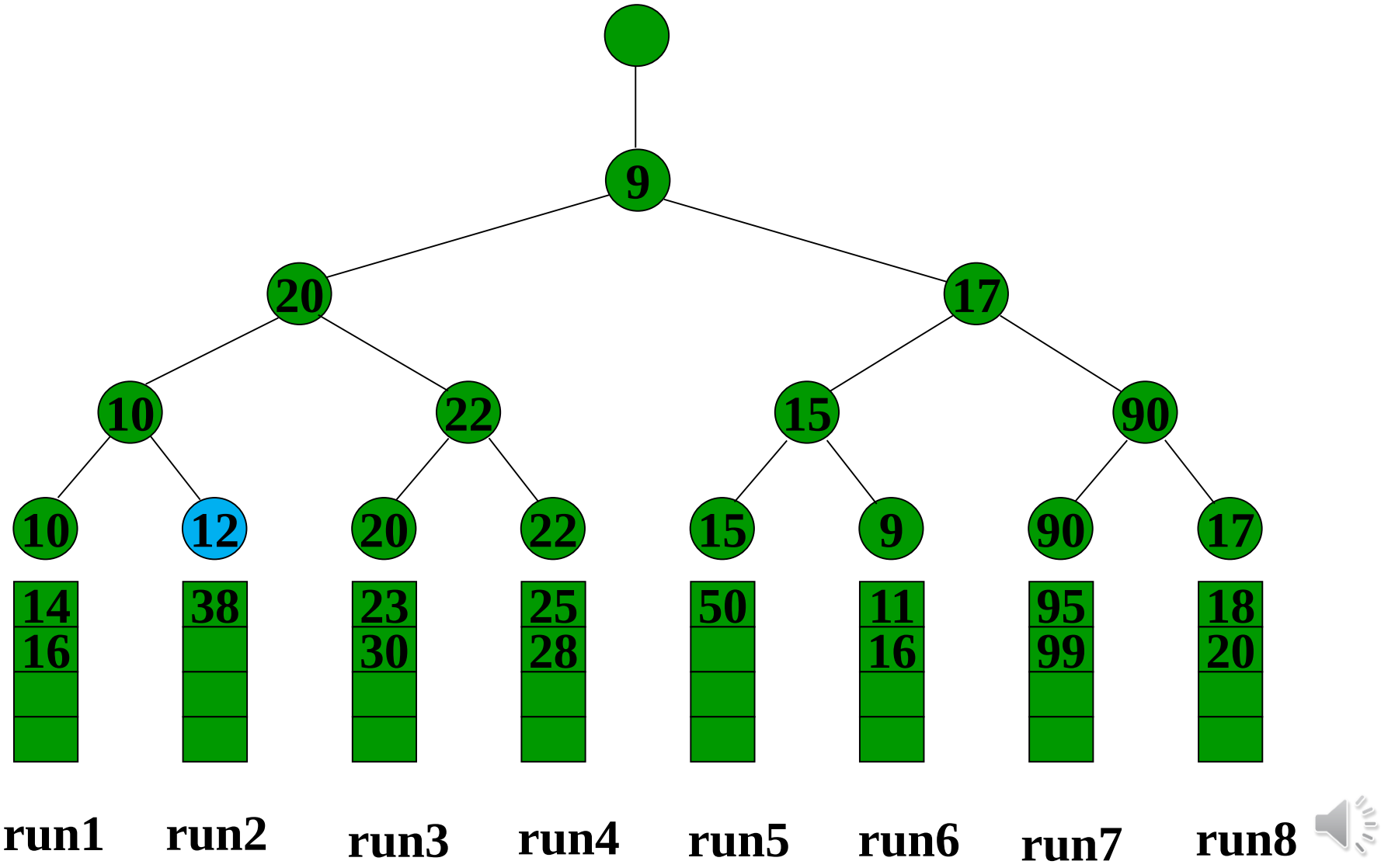
Loser Tree

```
Run:6  8  9
```



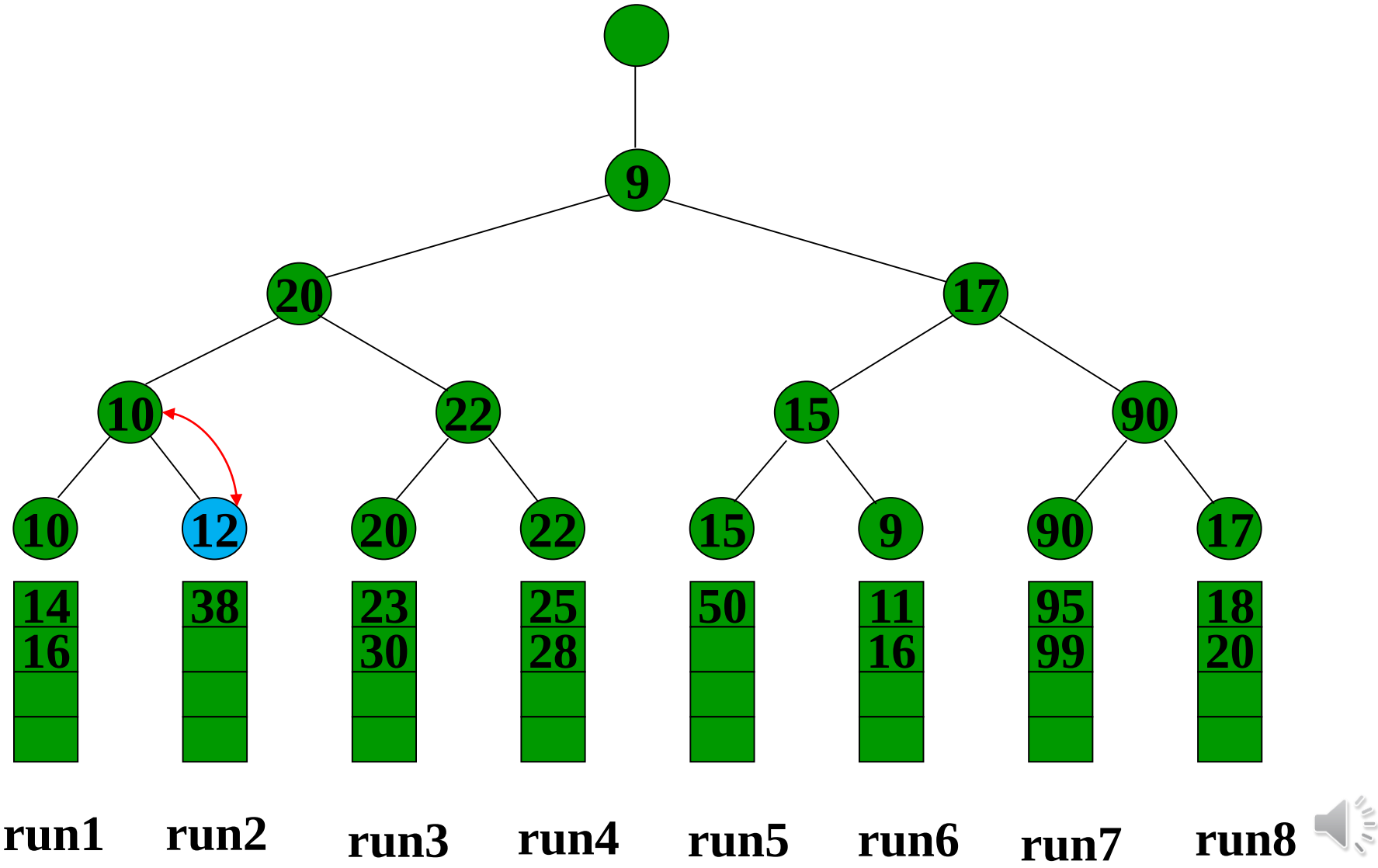
Loser Tree

Run:6 8 9



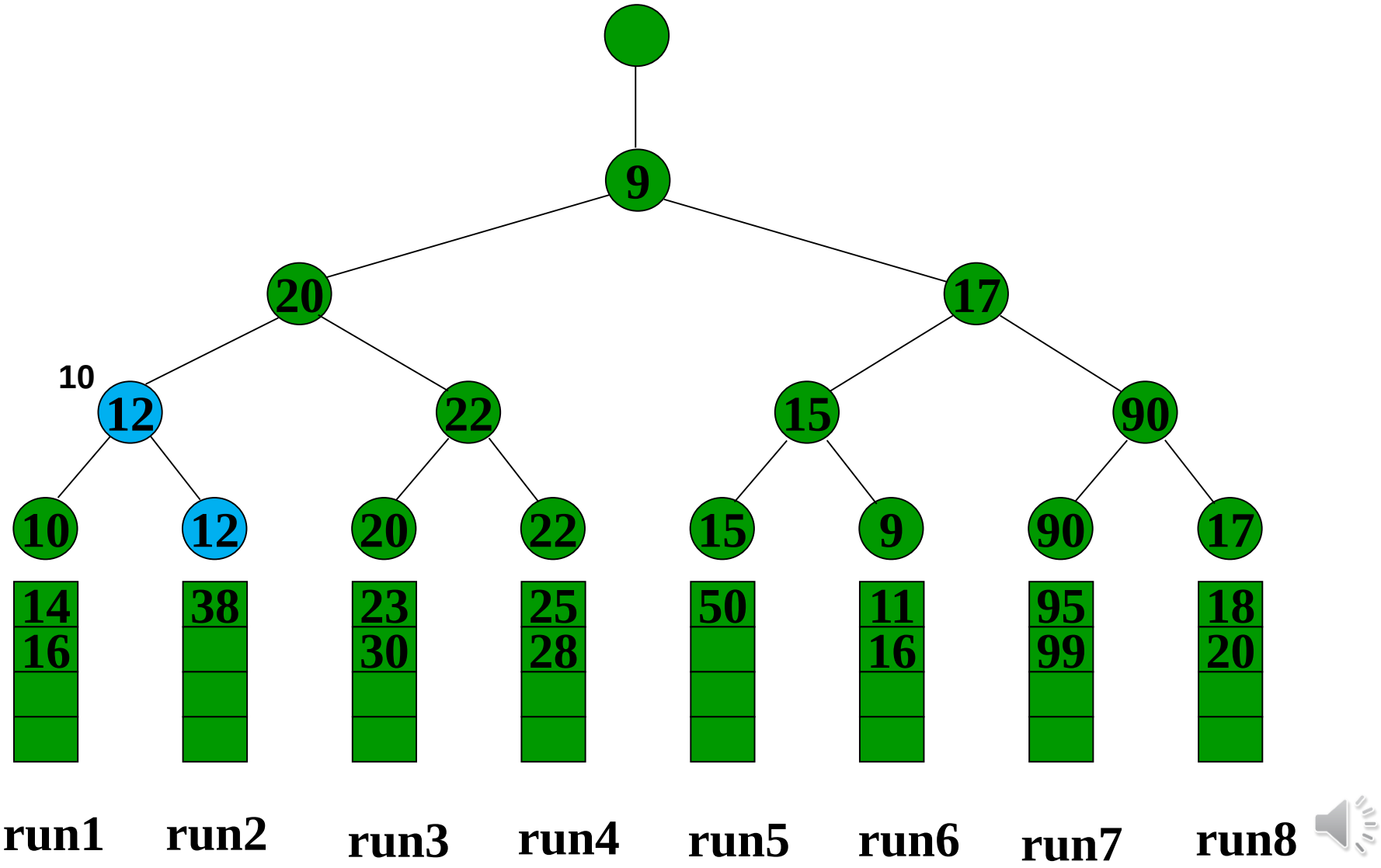
Loser Tree

Run:6 8 9



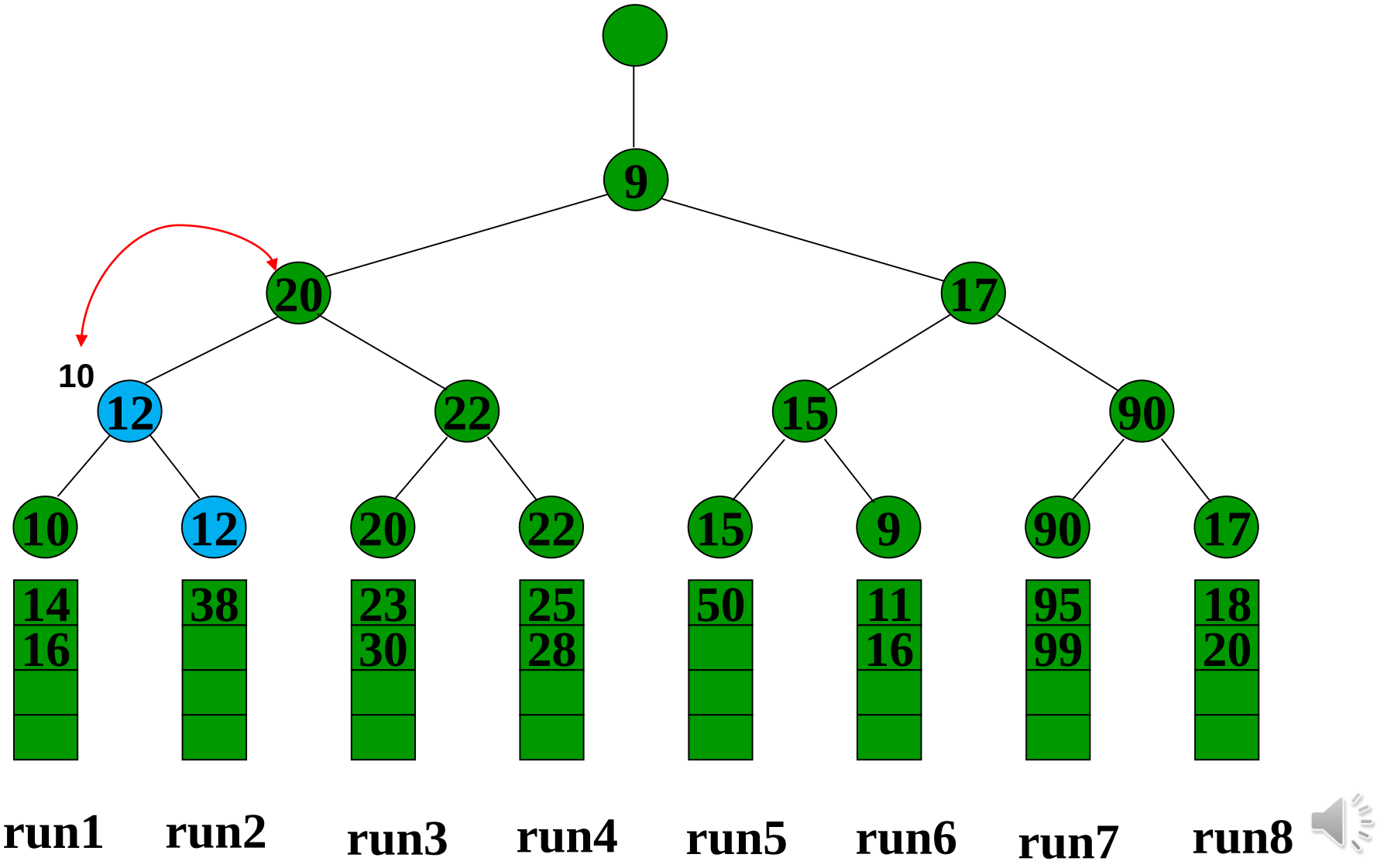
Loser Tree

Run:6 8 9



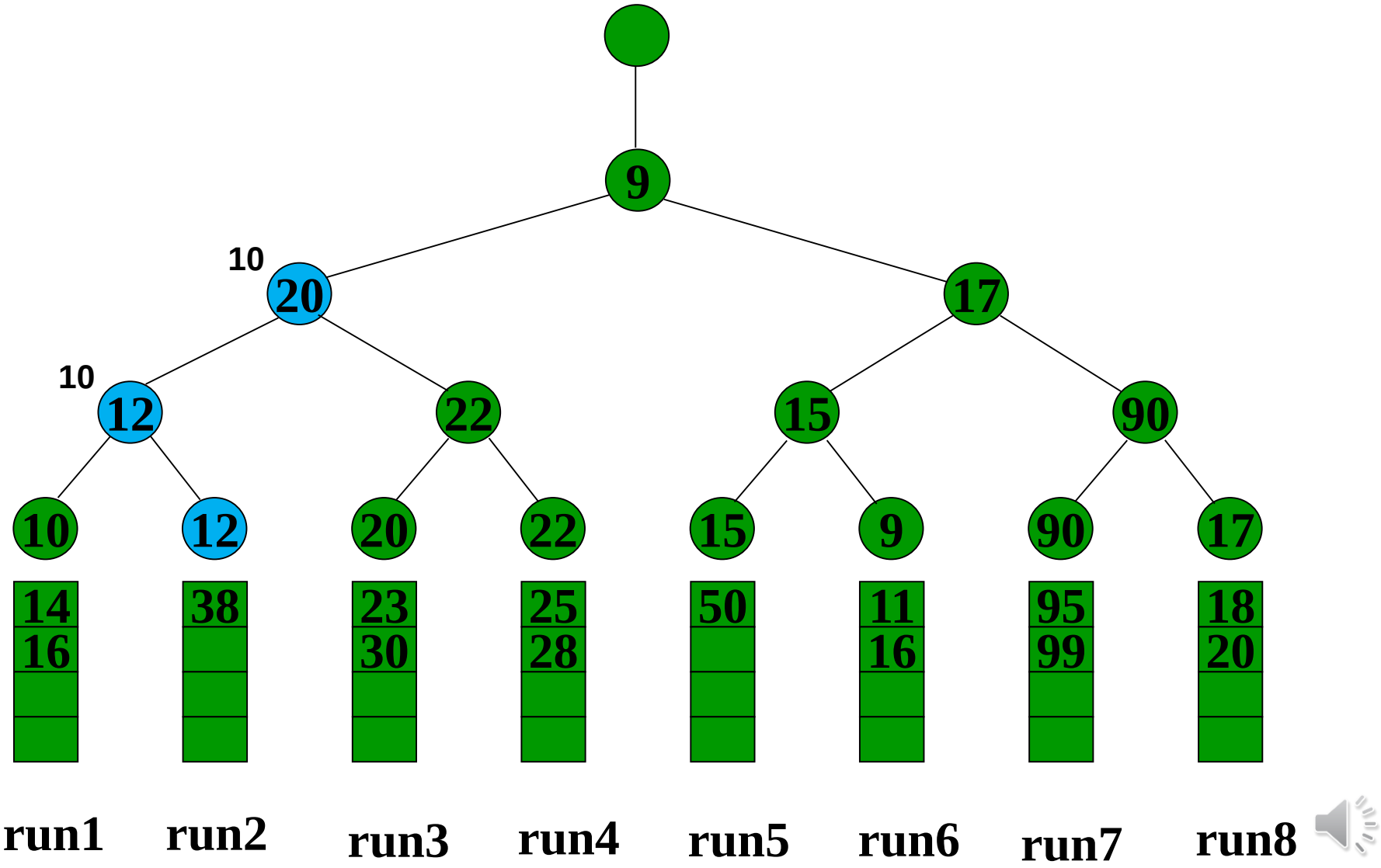
Loser Tree

Run:6 8 9



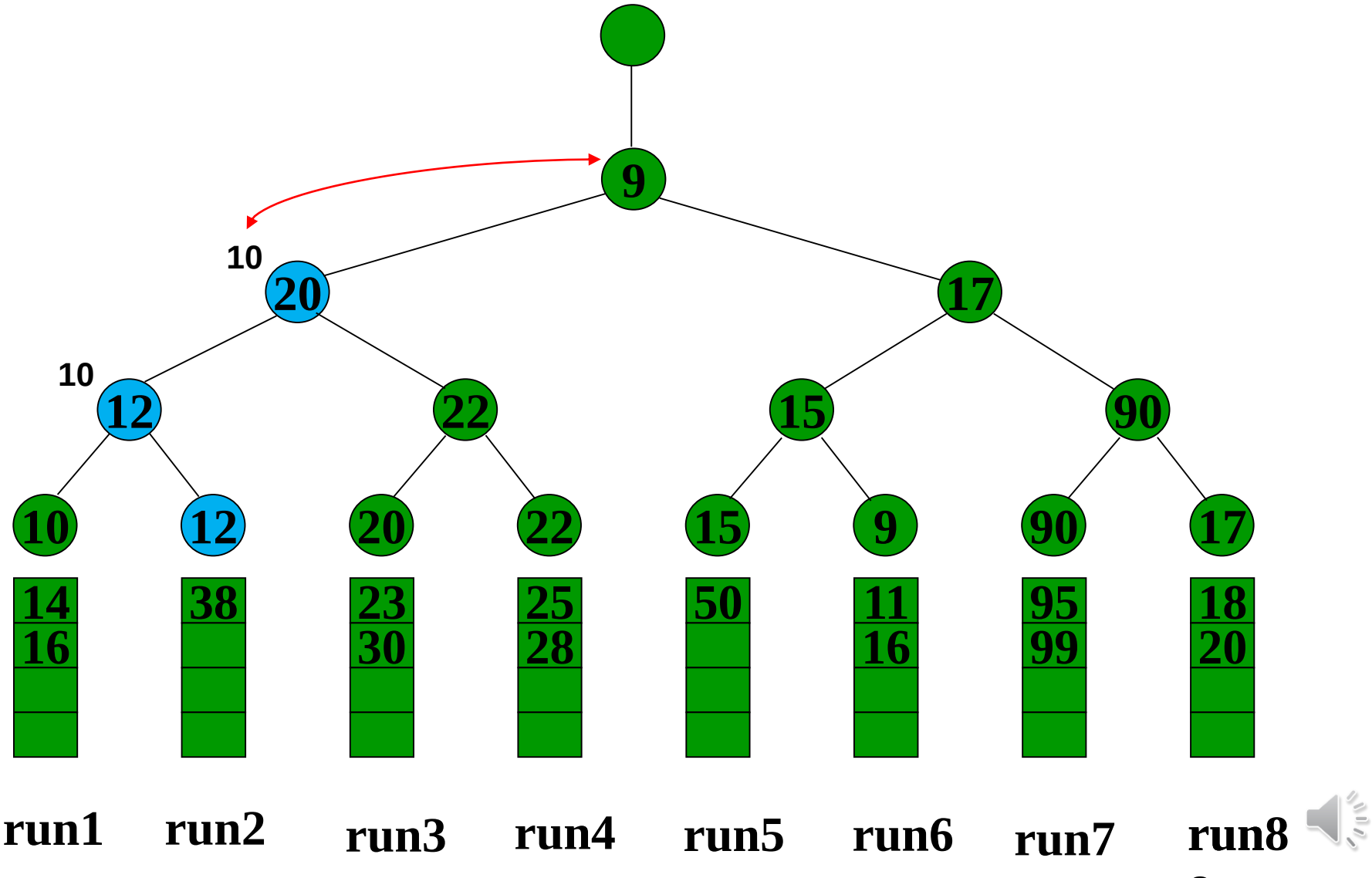
Loser Tree

Run:6 8 9



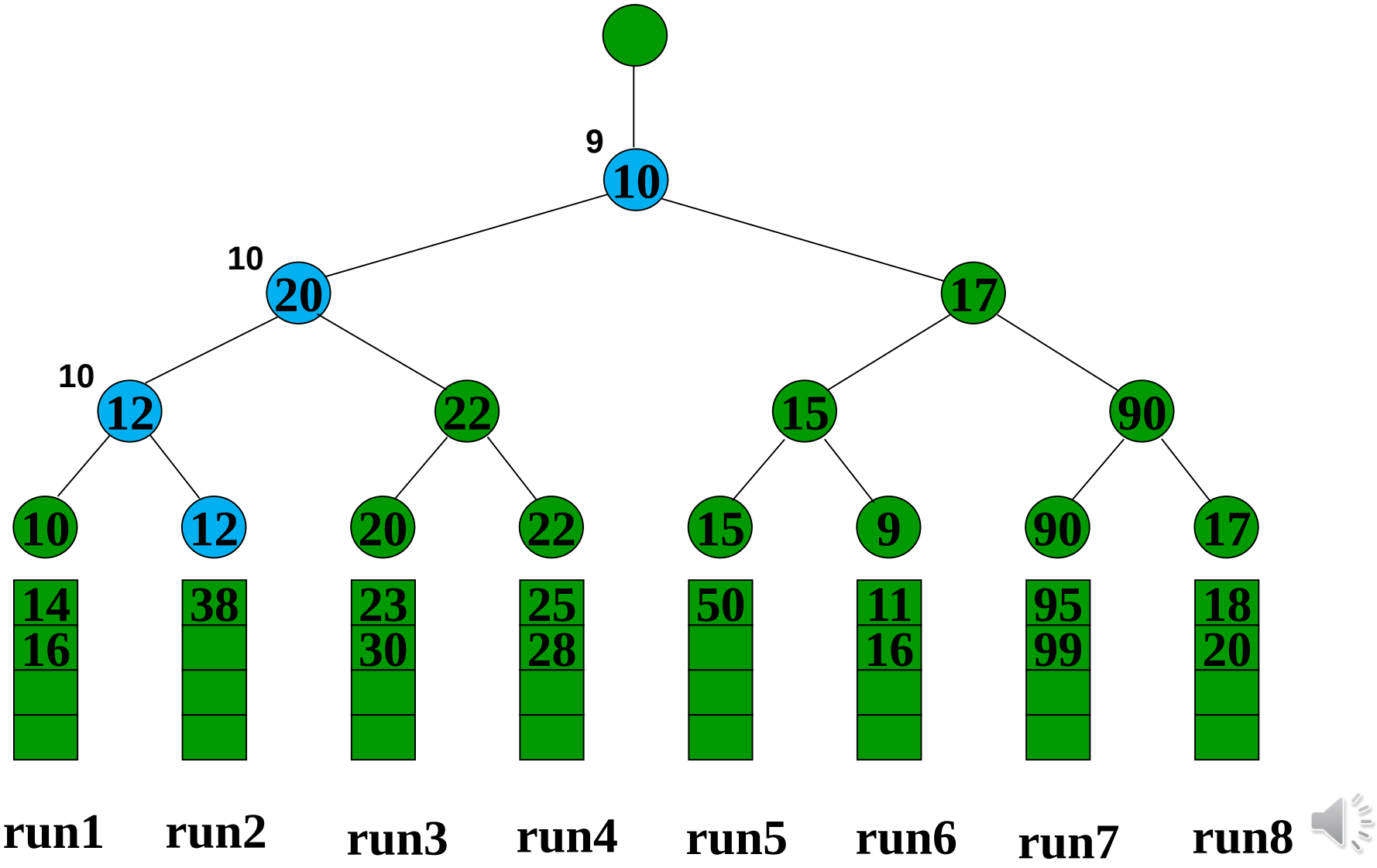
Loser Tree

```
Run:6  8  9
```



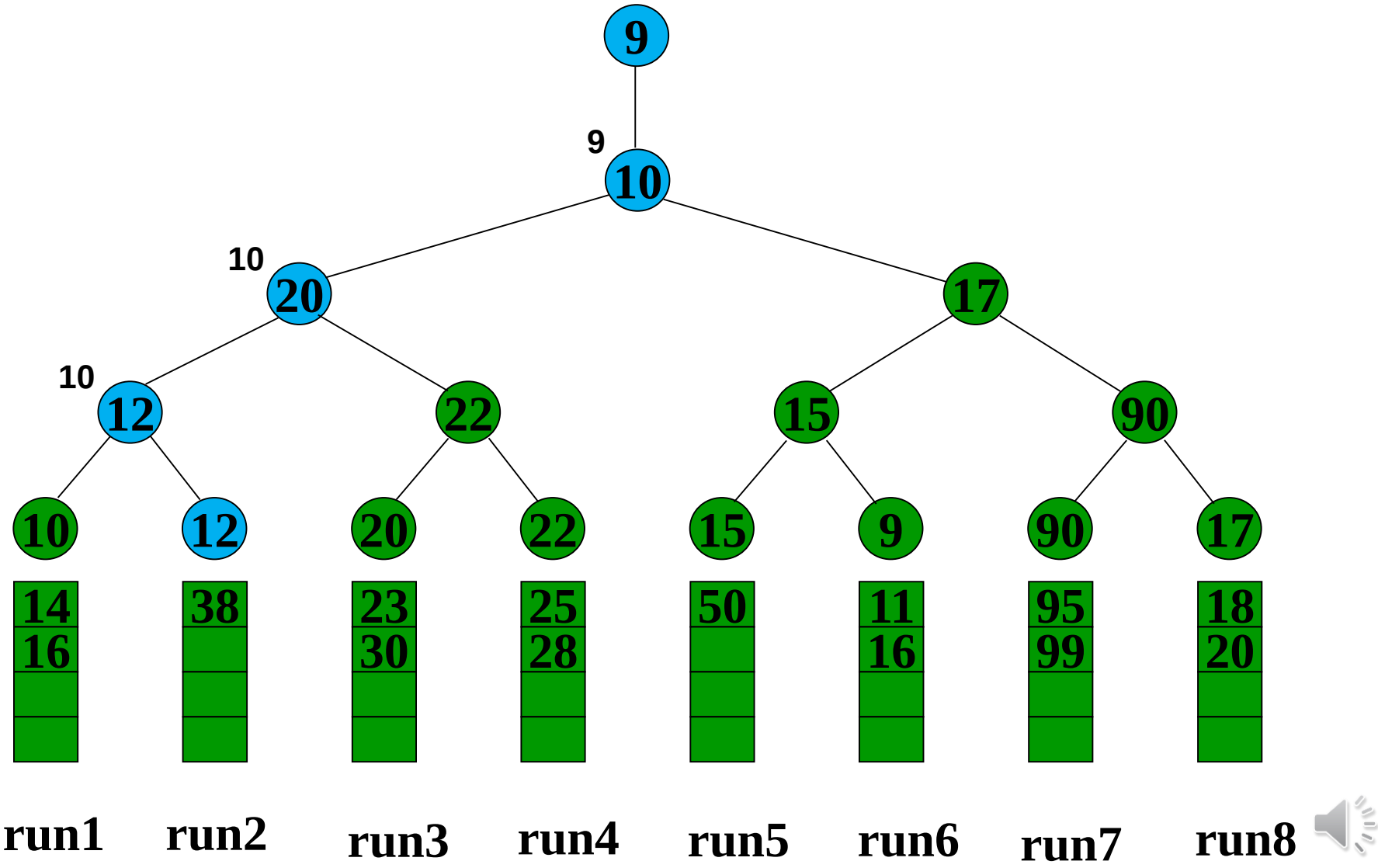
Loser Tree

Run:6 8 9



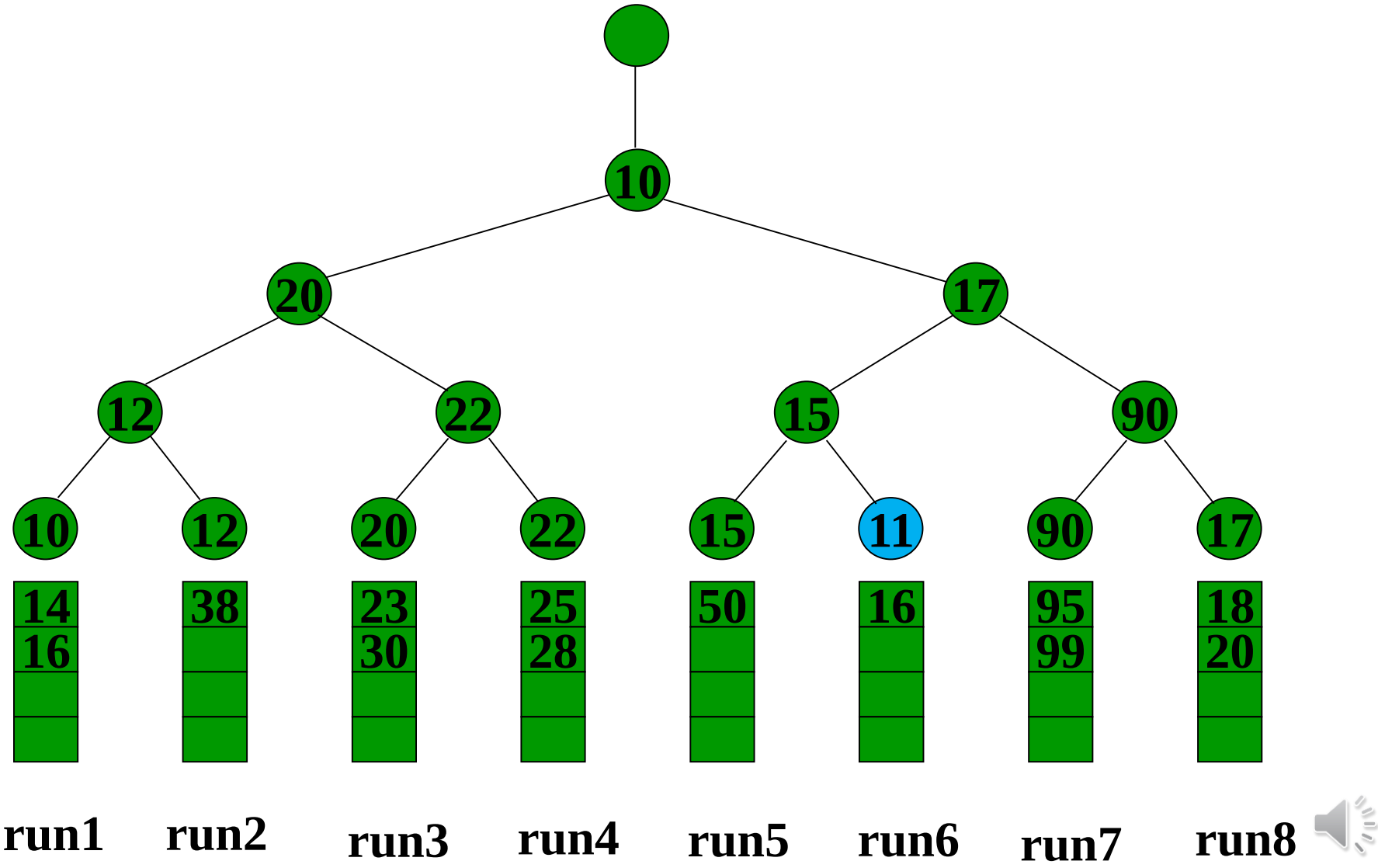
Loser Tree

Run:6 8 9 9



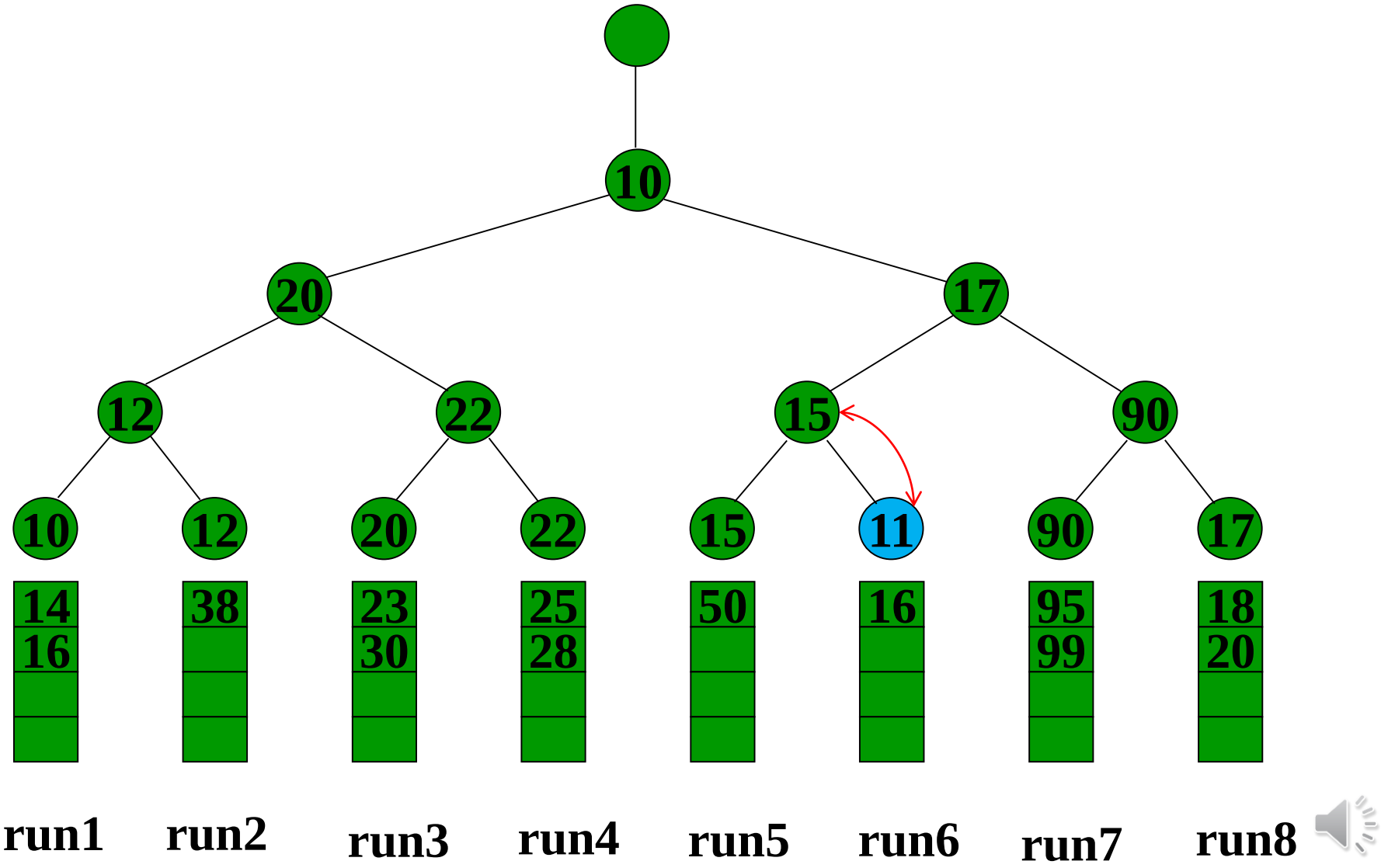
Loser Tree

Run:6 8 9 9



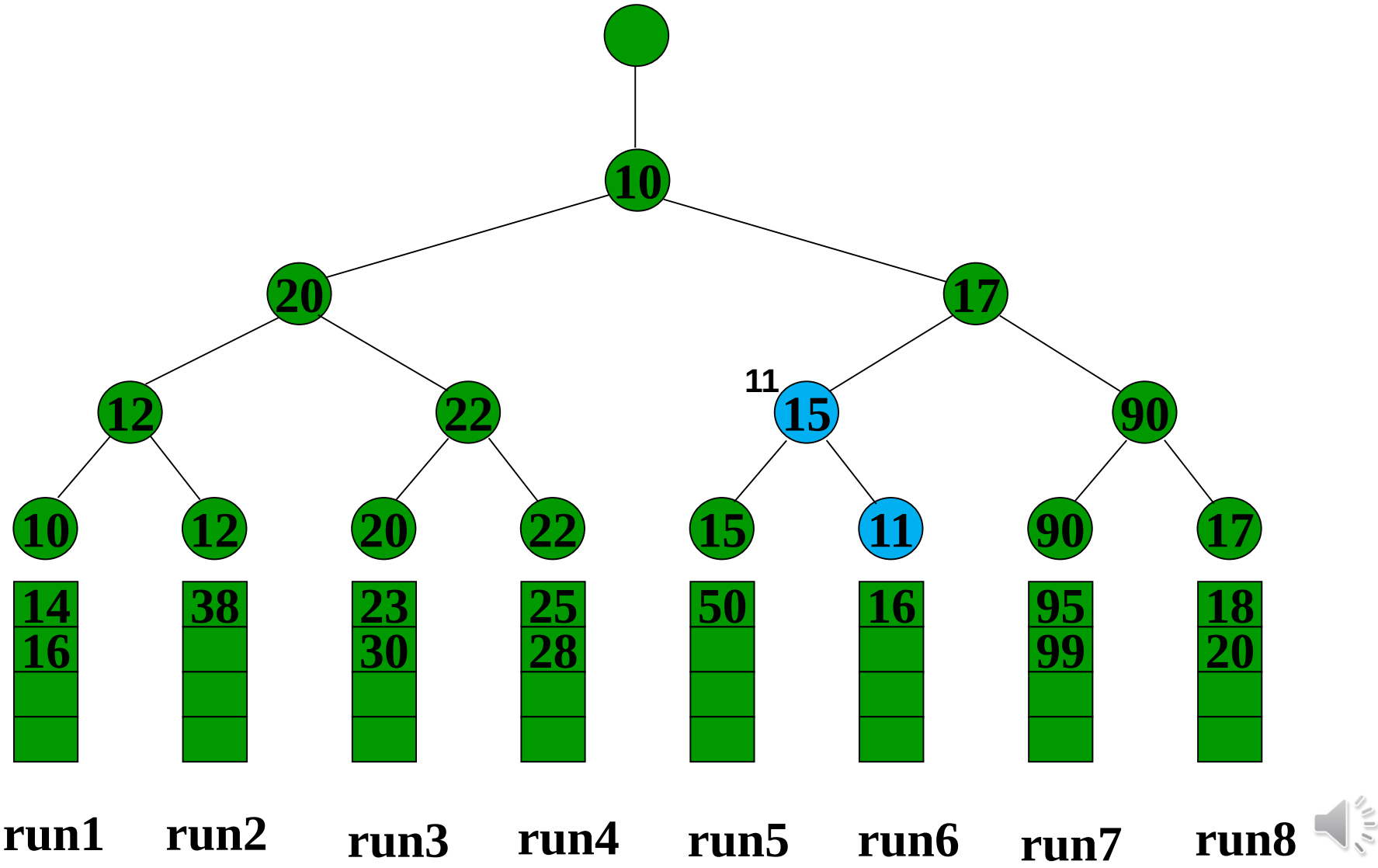
Loser Tree

Run:6 8 9 9



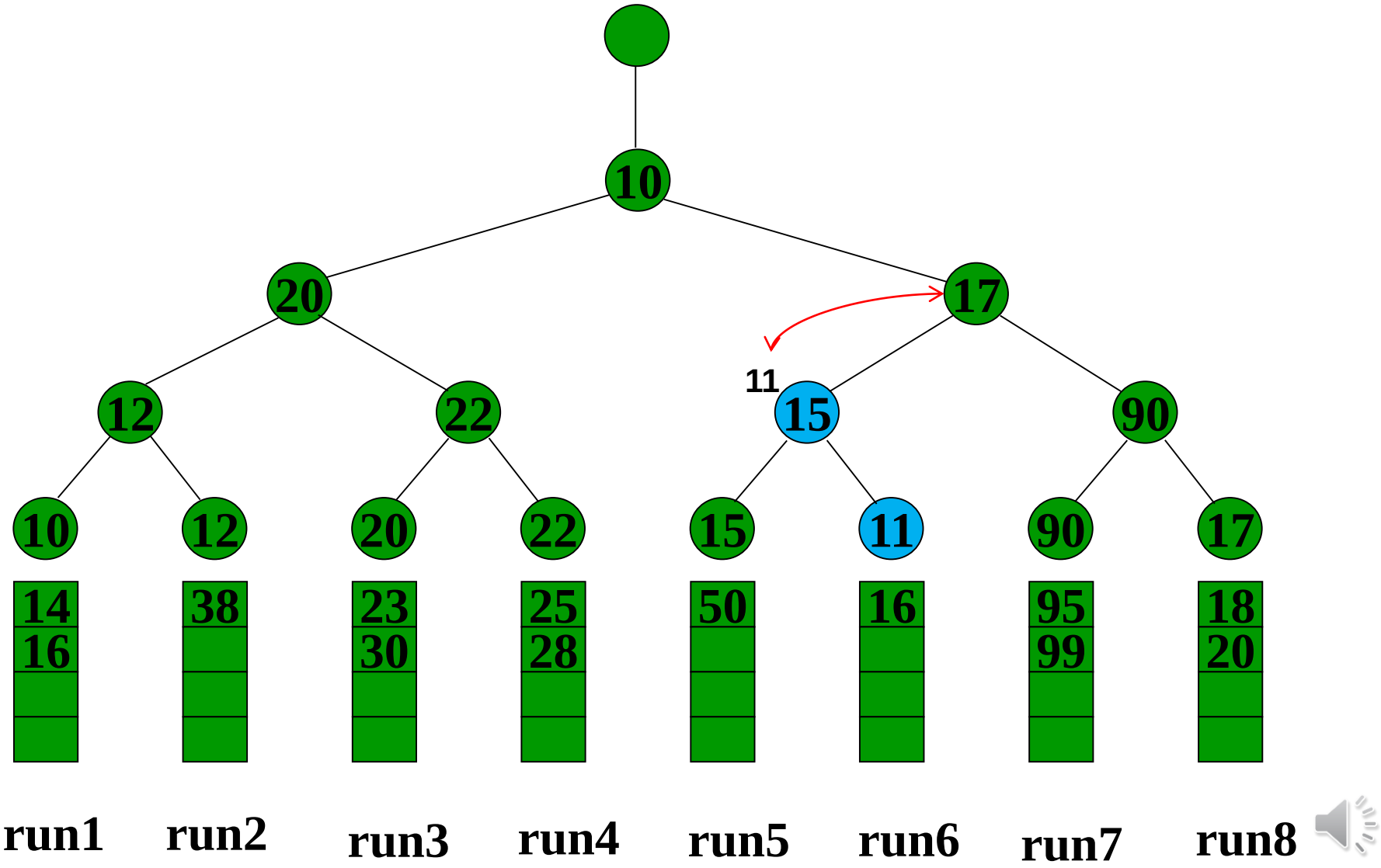
Loser Tree

Run:6 8 9 9



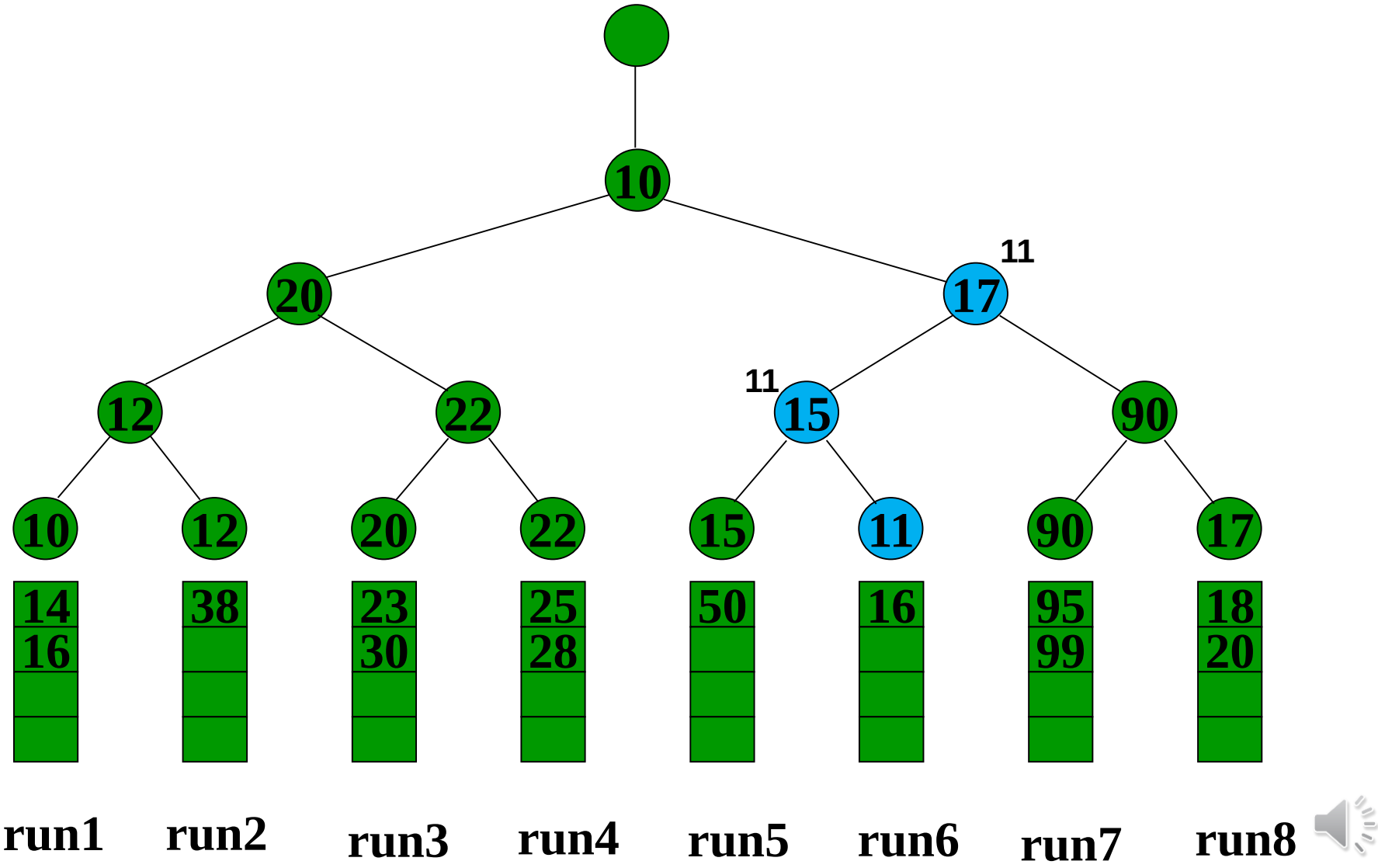
Loser Tree

Run:6 8 9 9



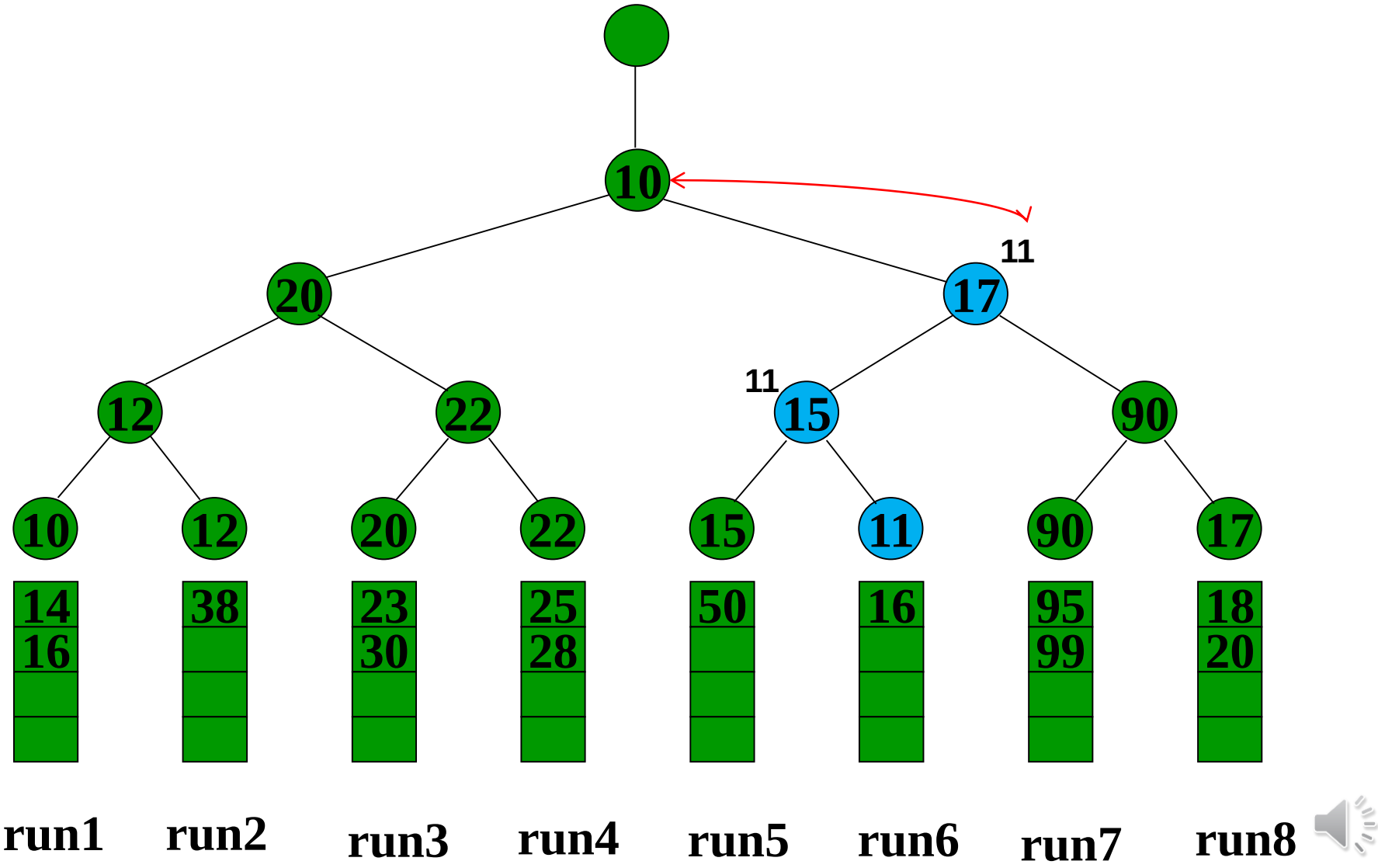
Loser Tree

Run:6 8 9 9



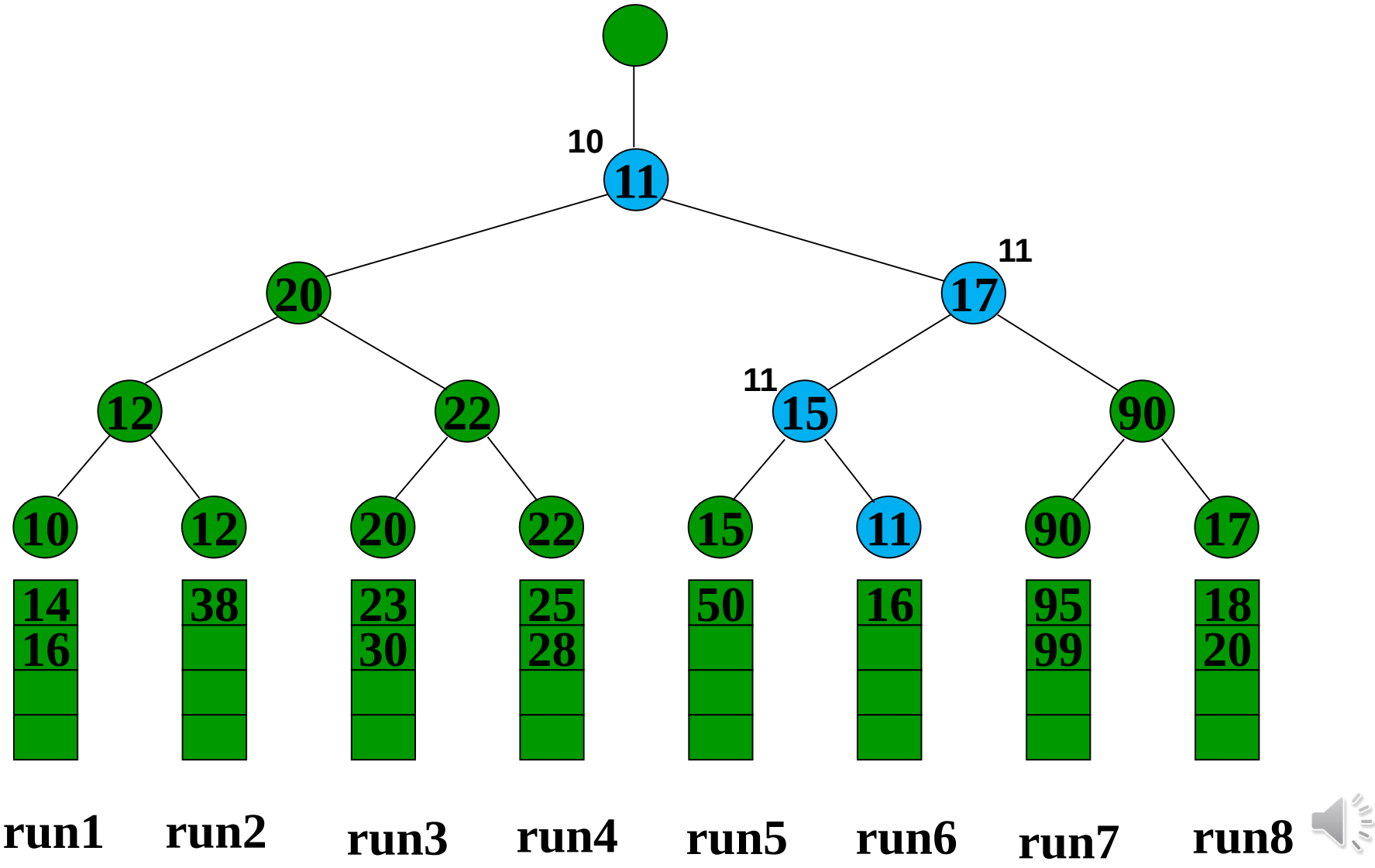
Loser Tree

Run:6 8 9 9



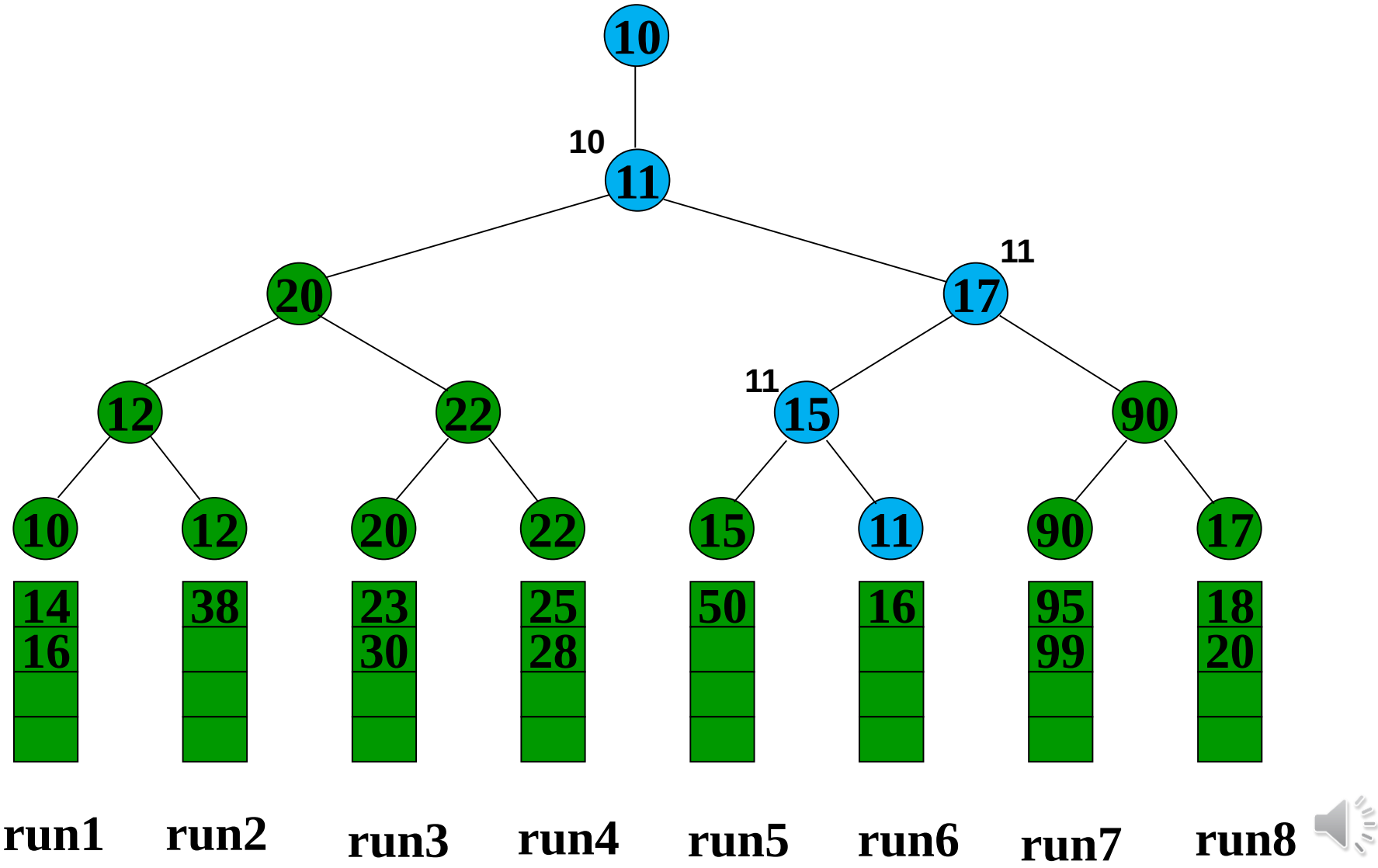
Loser Tree

Run:6 8 9 9



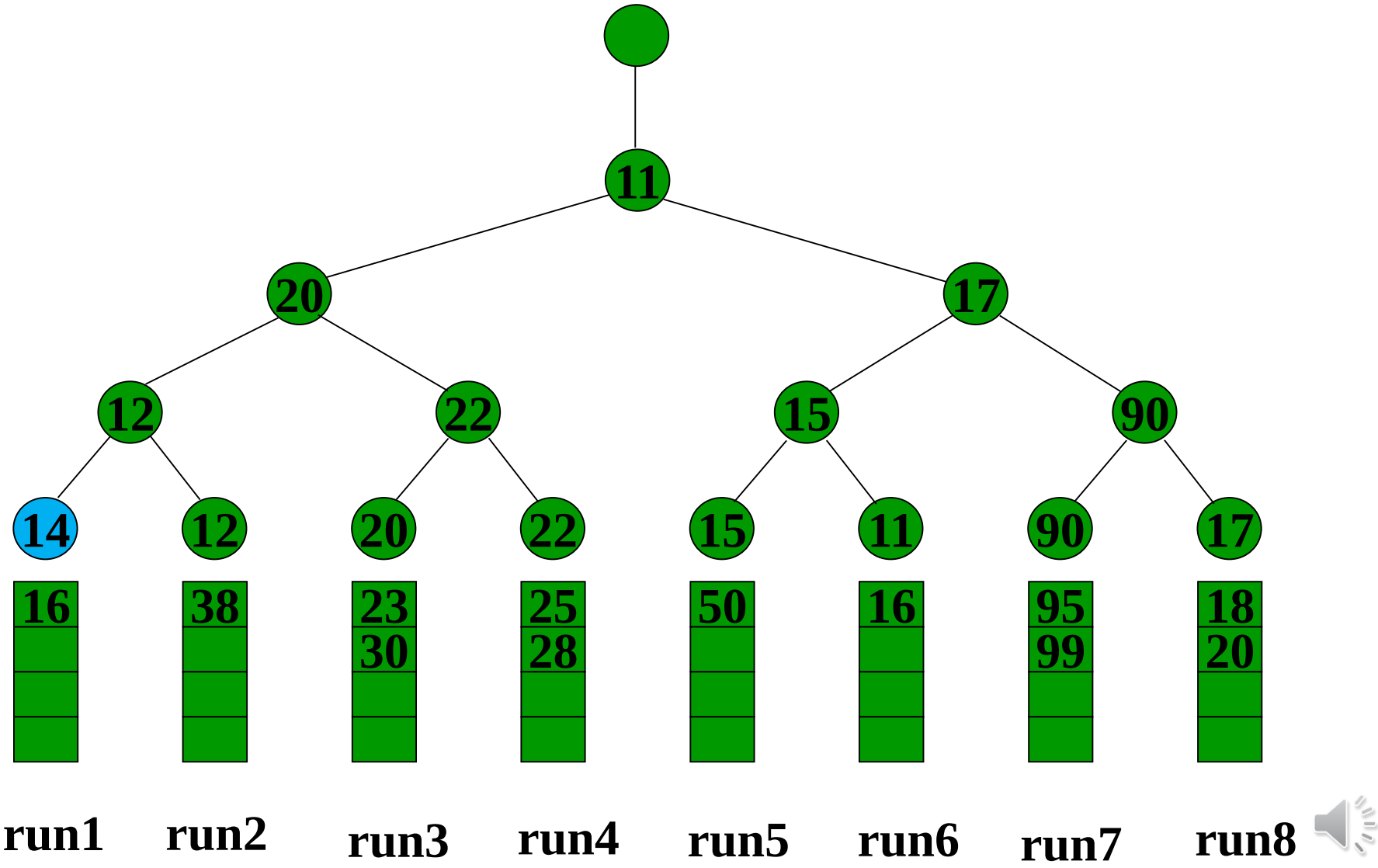
Loser Tree

```
Run:6  8  9  9  10
```



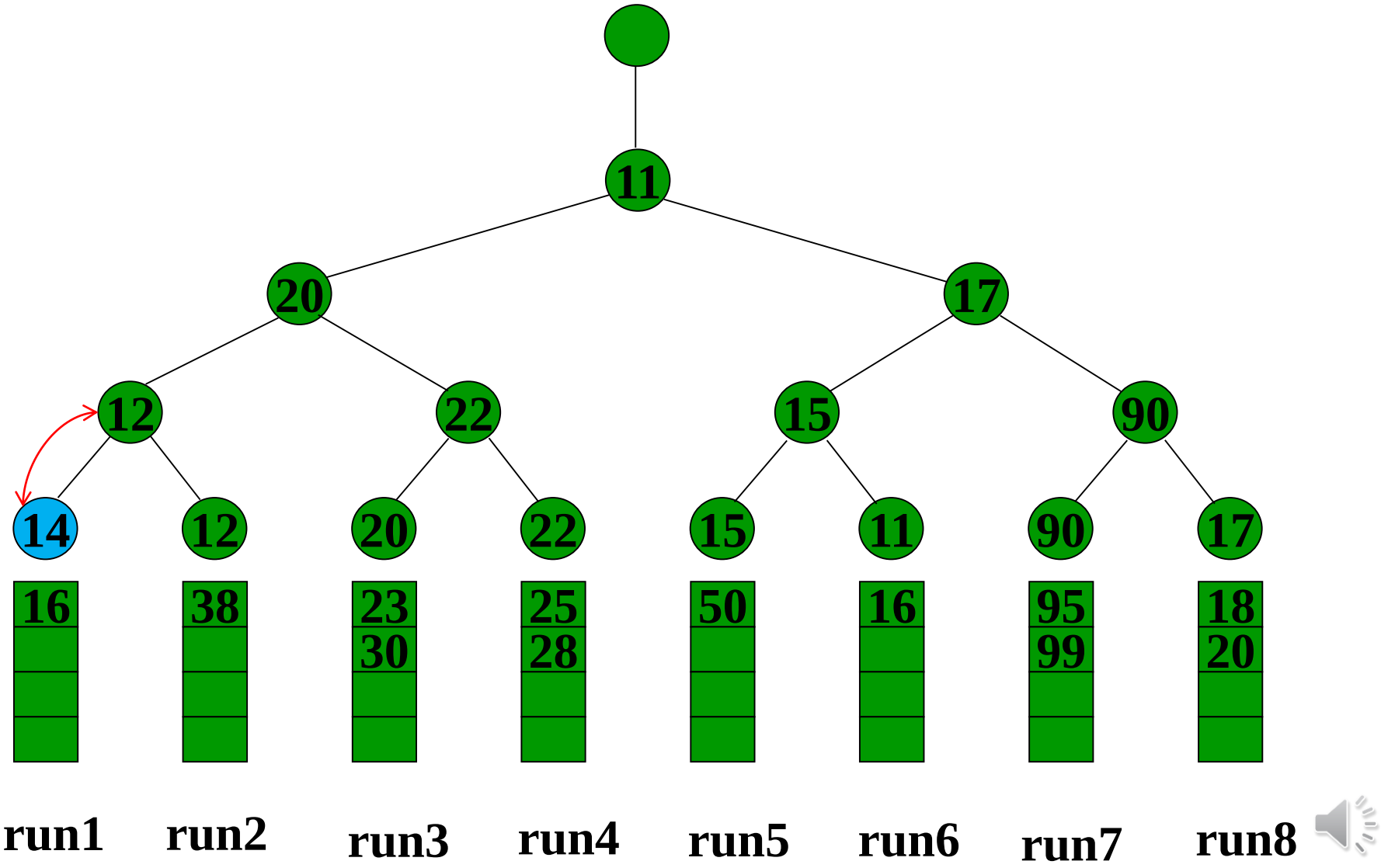
Loser Tree

Run:6 8 9 9 10



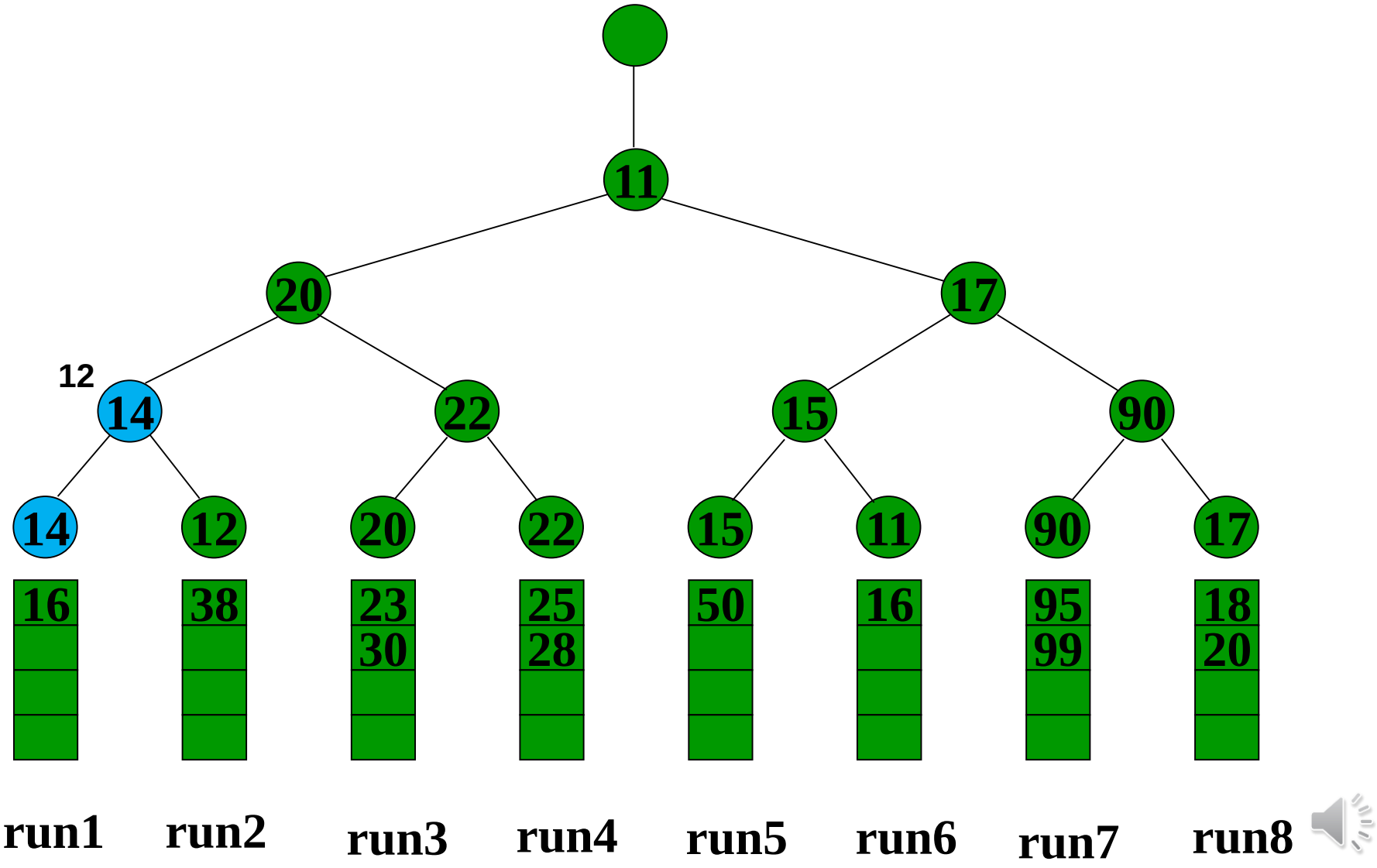
Loser Tree

```
Run:6  8  9  9  10
```



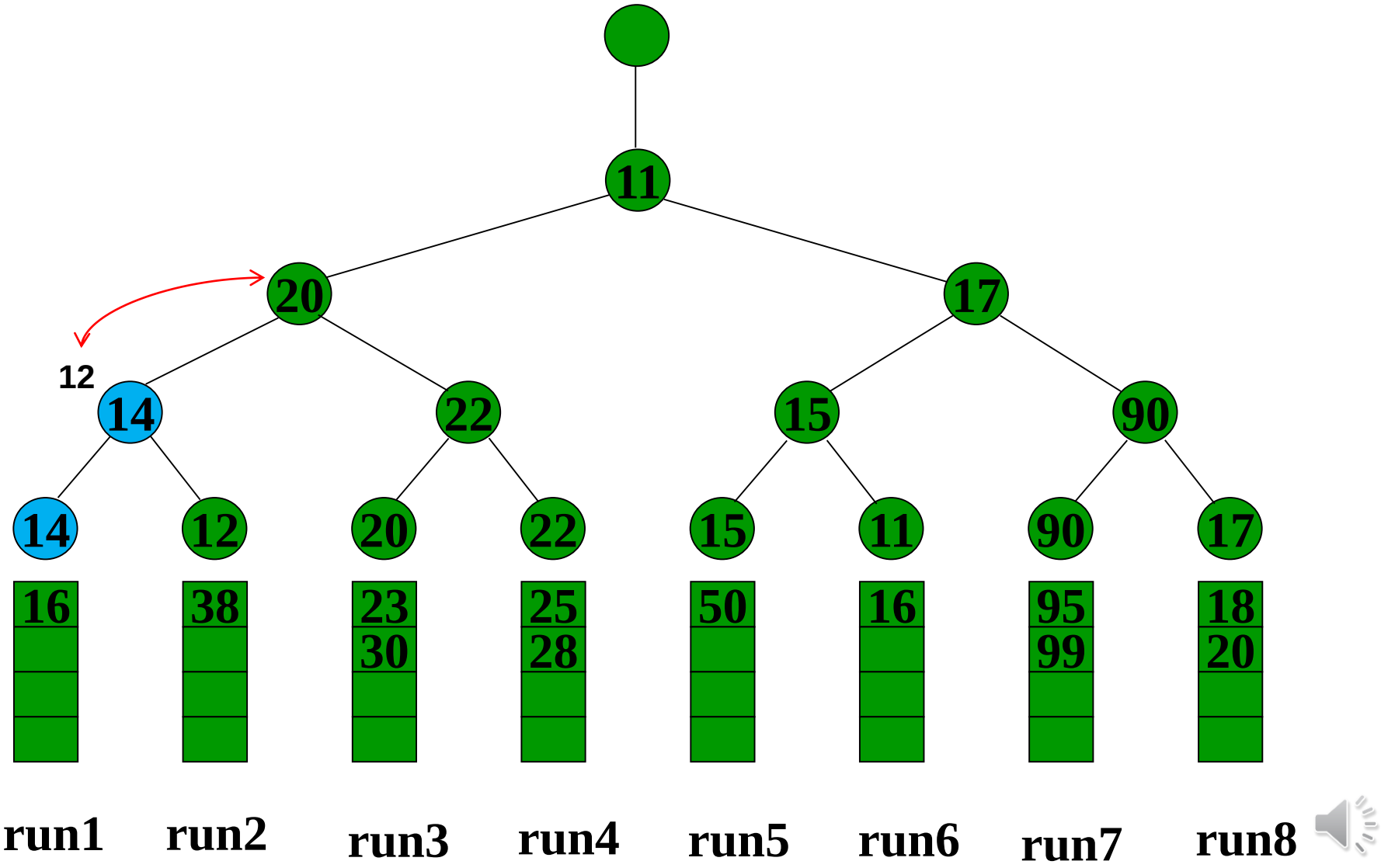
Loser Tree

Run:6 8 9 9 10



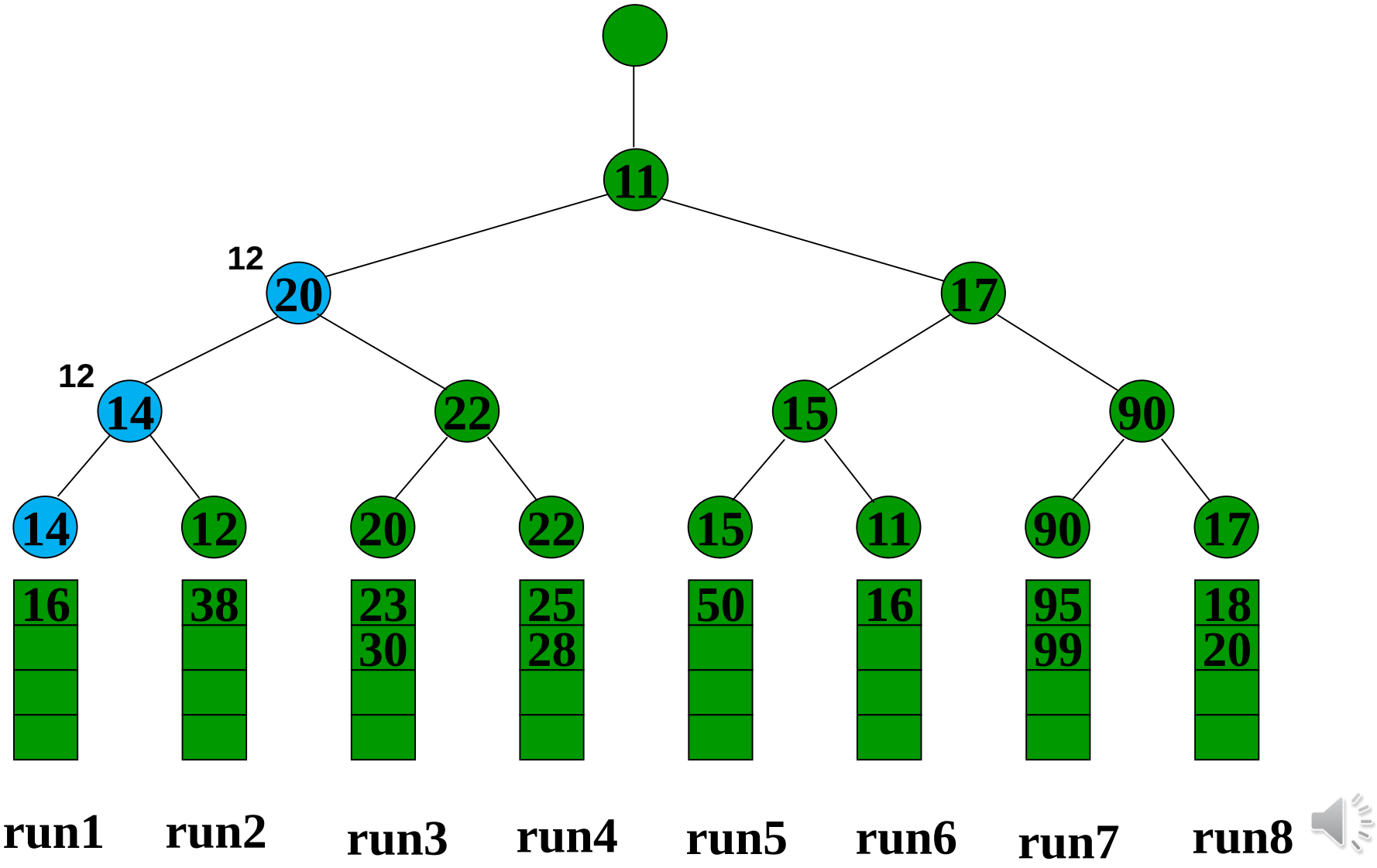
Loser Tree

```
Run:6  8  9  9  10
```



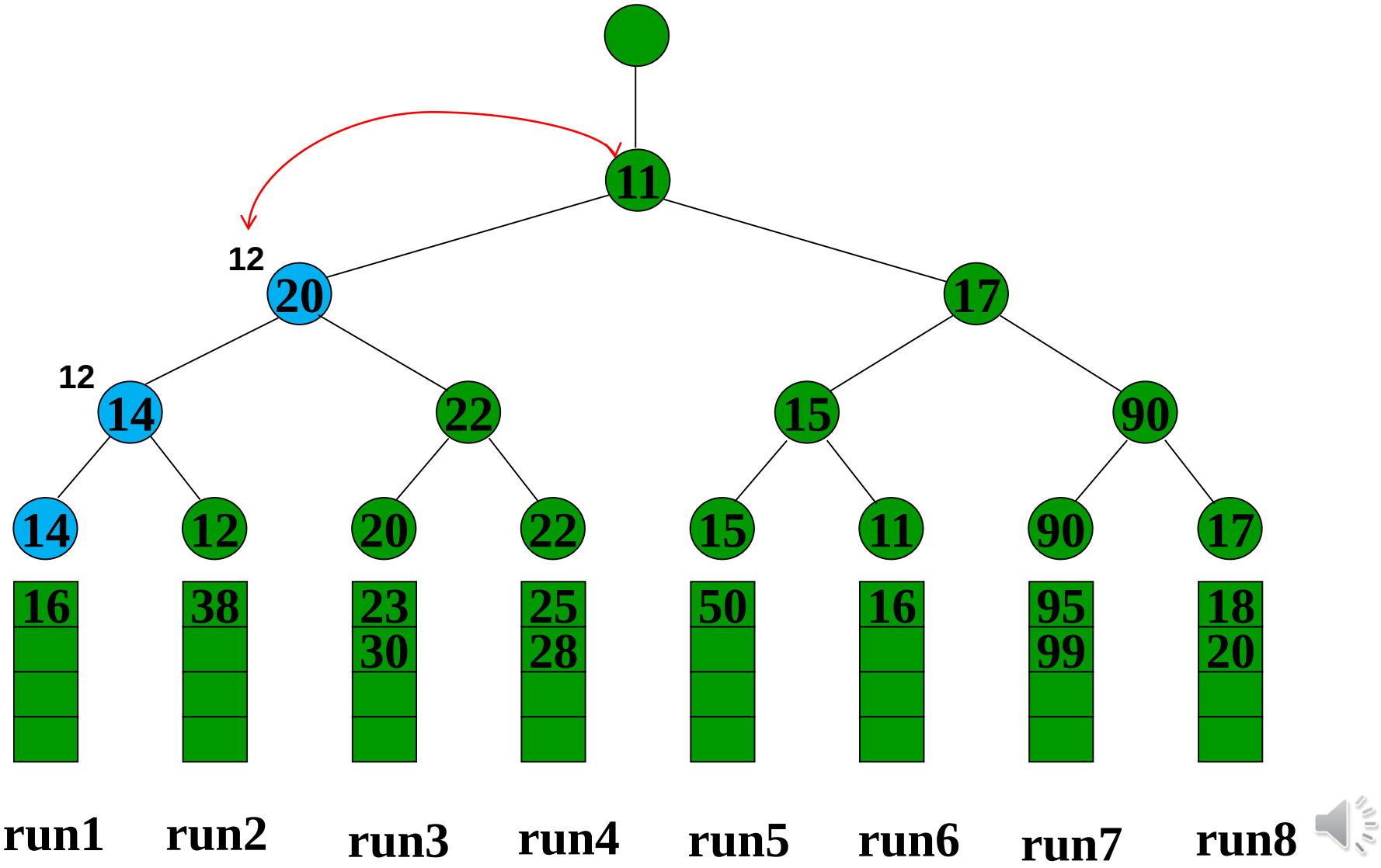
Loser Tree

Run:6 8 9 9 10



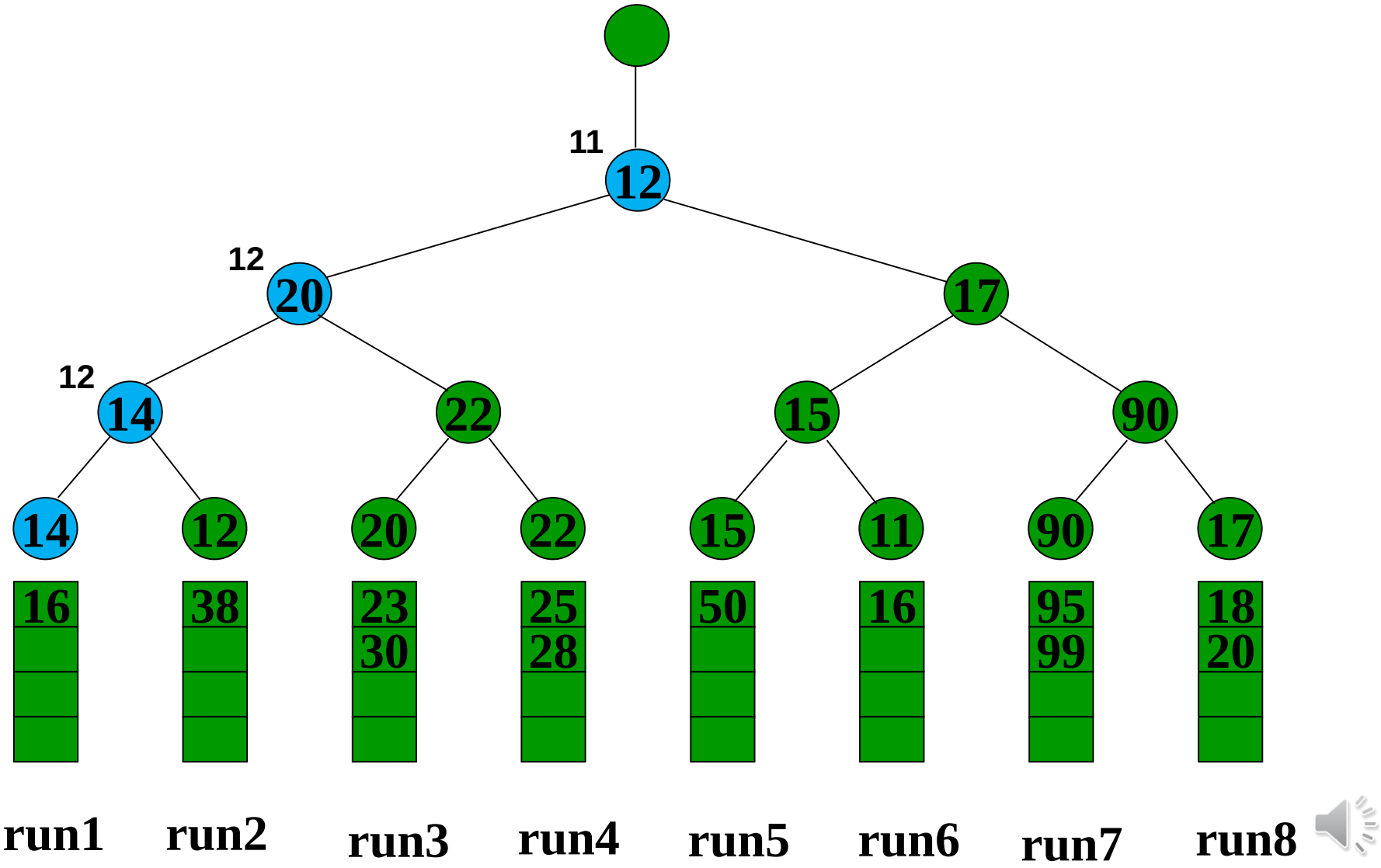
Loser Tree

Run:6 8 9 9 10



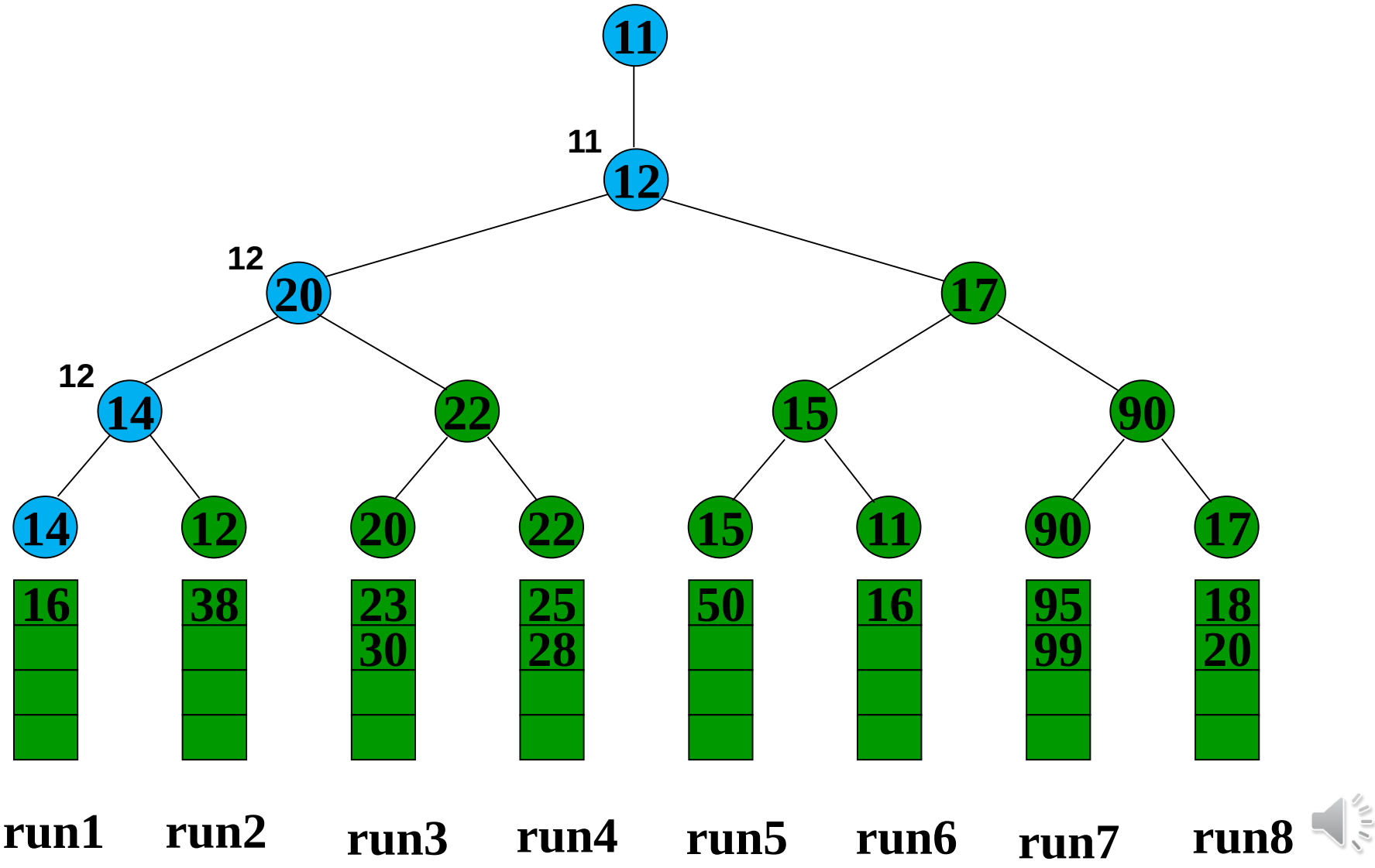
Loser Tree

Run:6 8 9 9 10



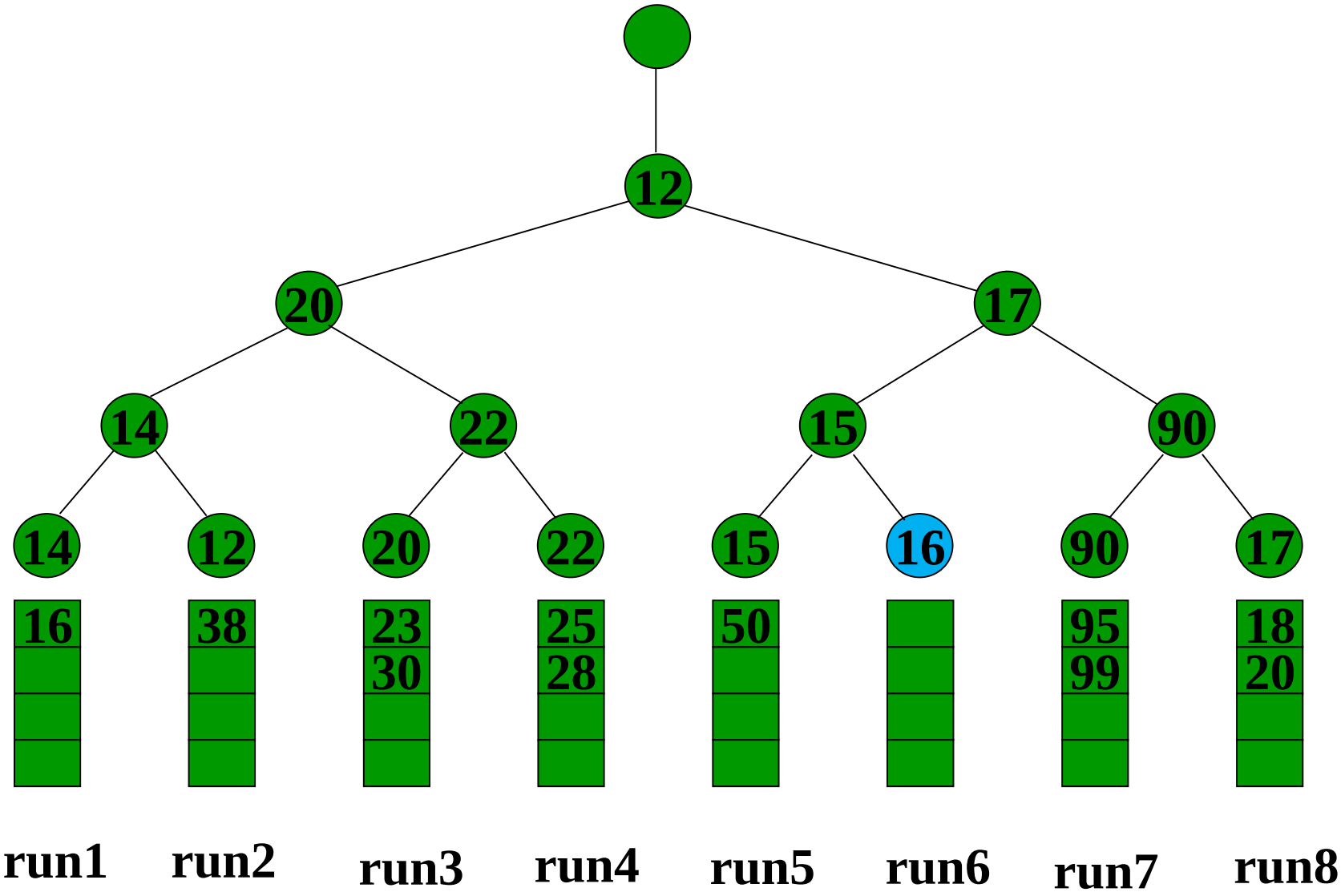
Loser Tree

Run:6 8 9 9 10 11



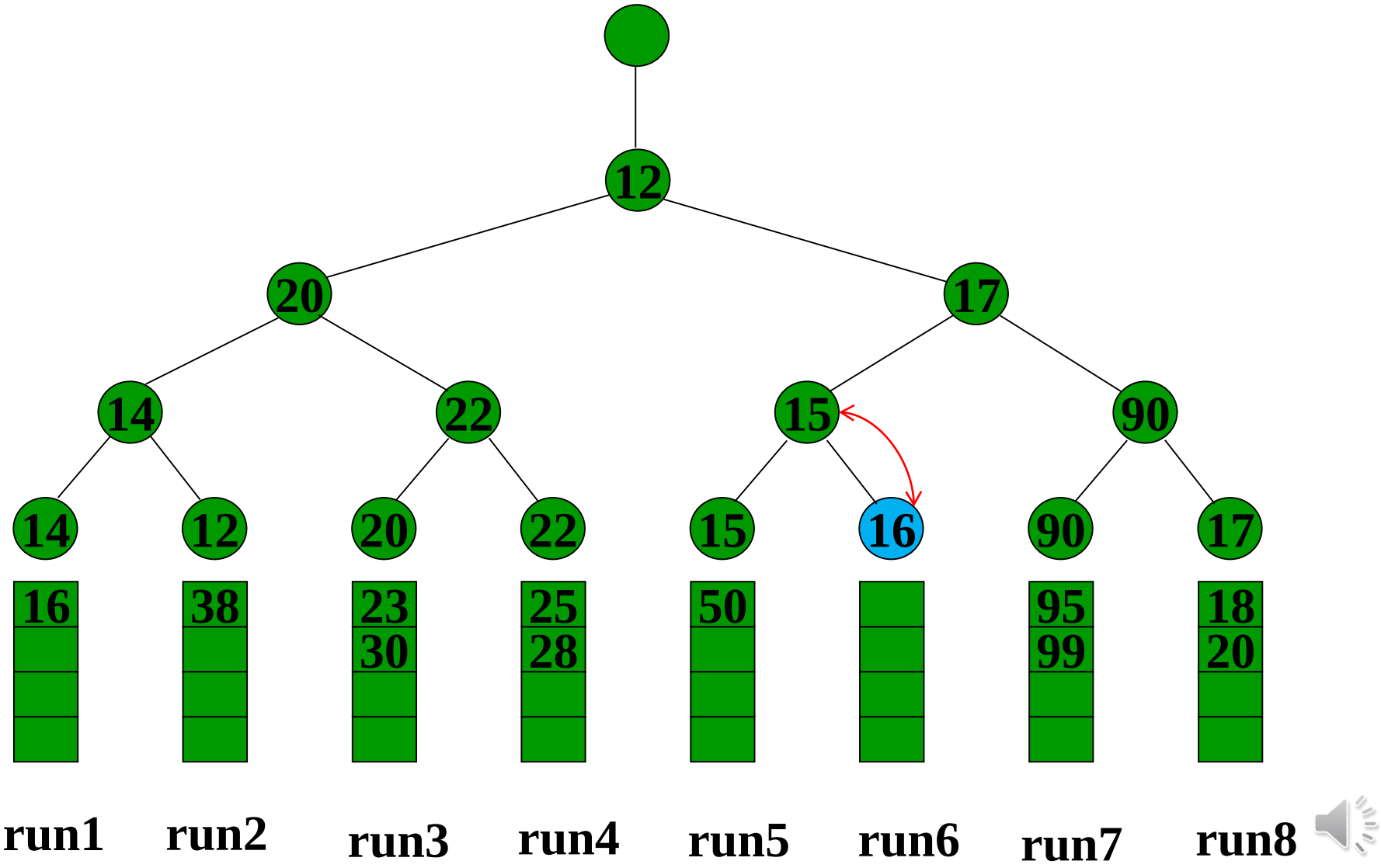
Loser Tree

```
Run:6  8  9  9  10  11
```



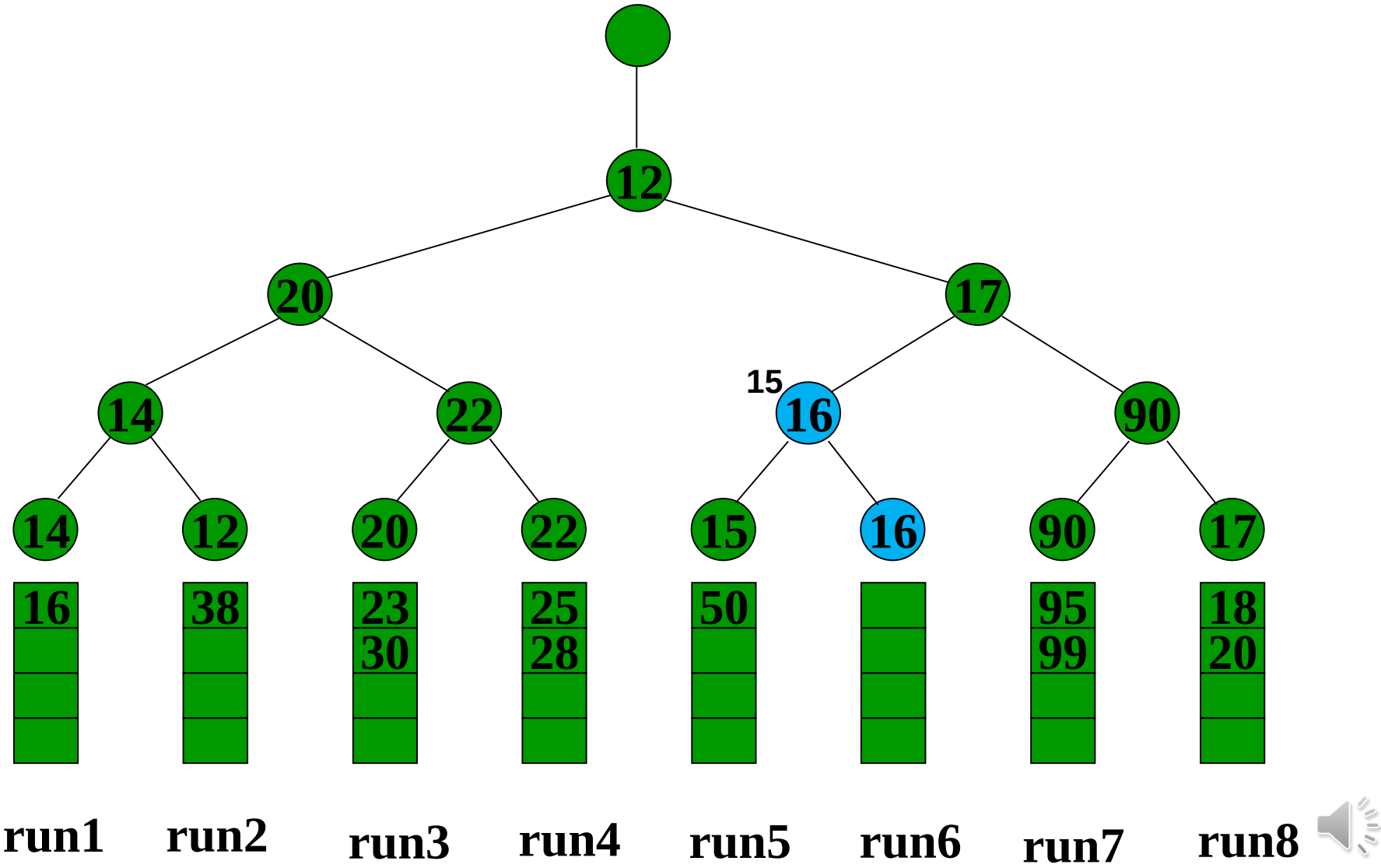
Loser Tree

Run:6 8 9 9 10 11



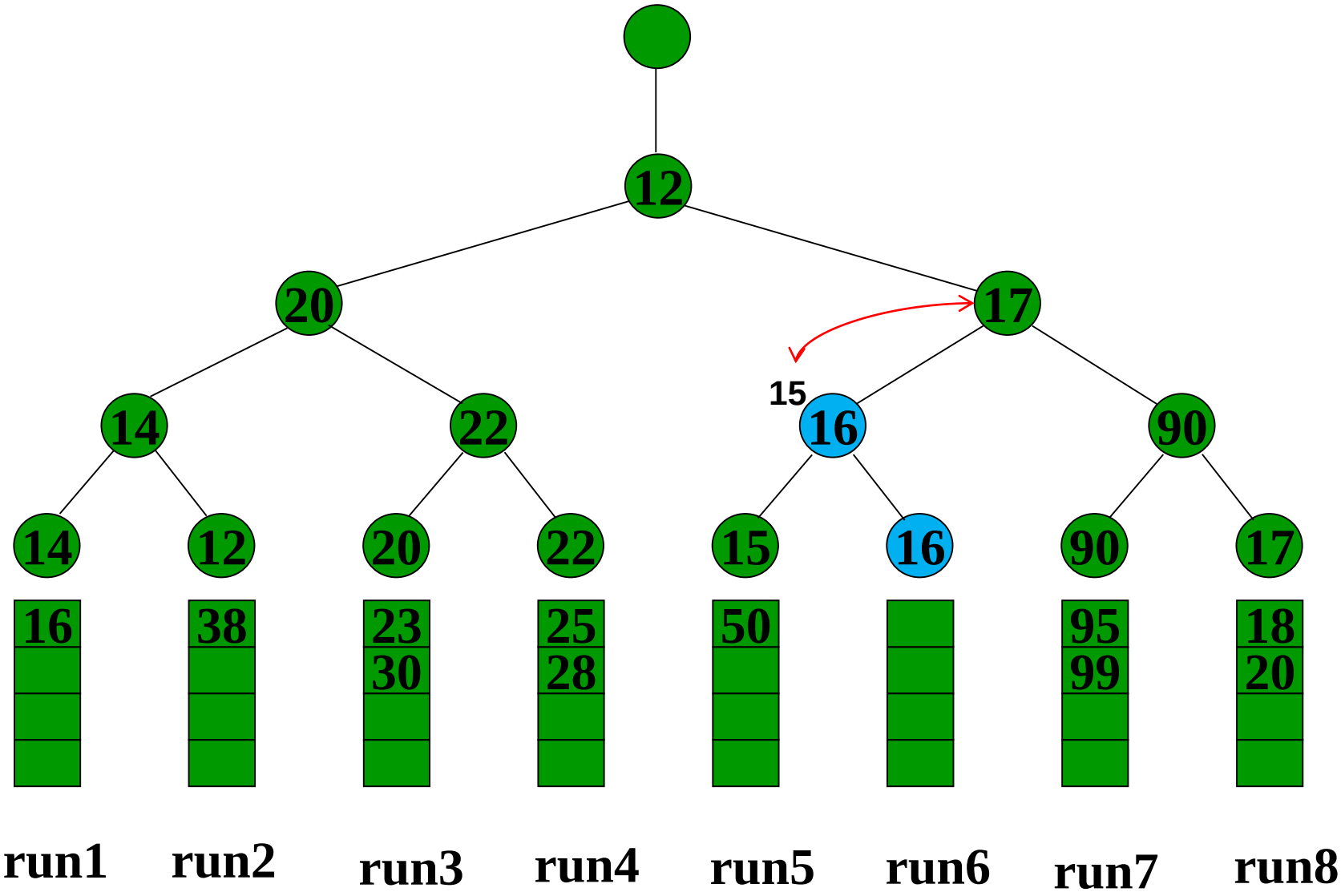
Loser Tree

Run:6 8 9 9 10 11



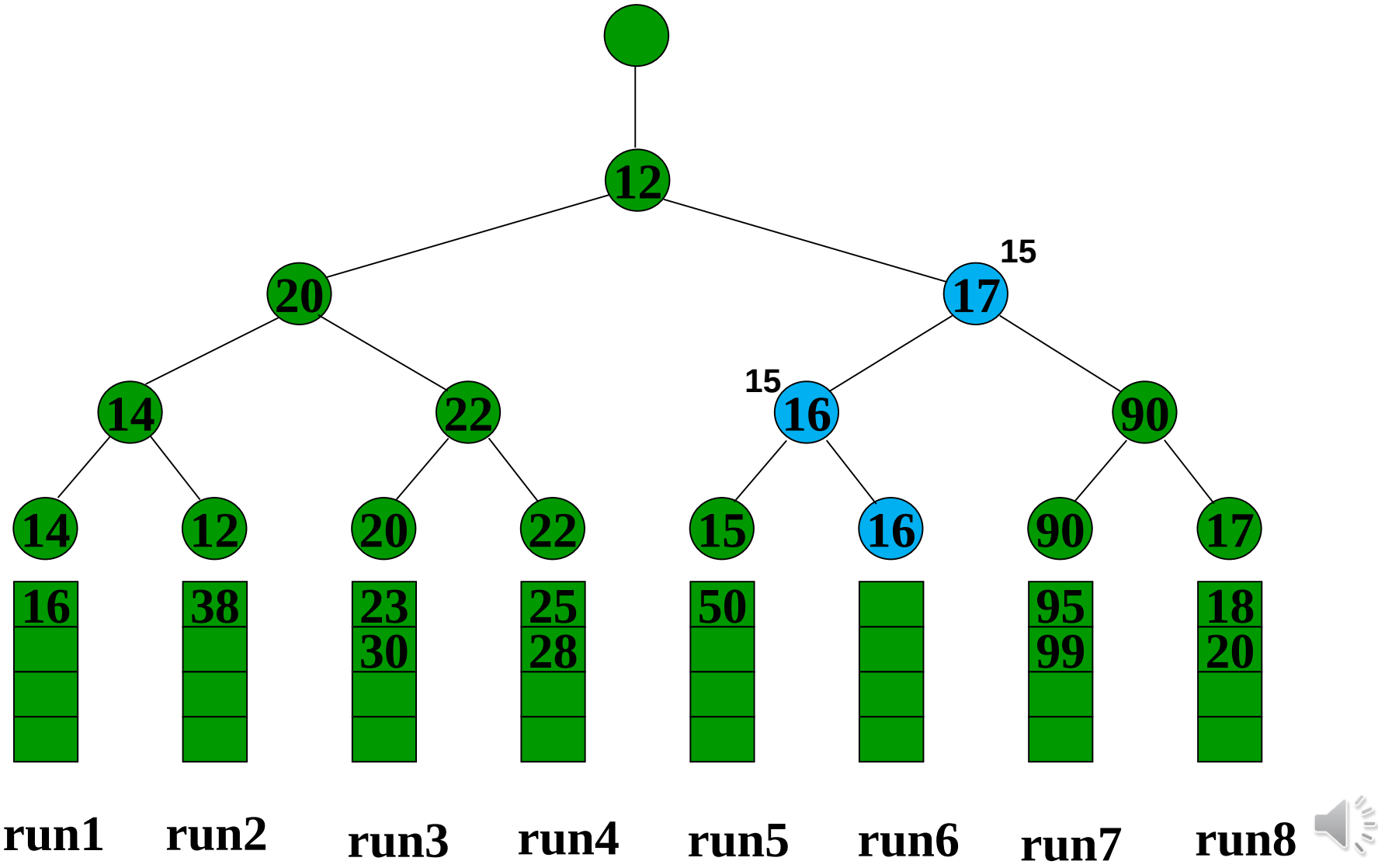
Loser Tree

Run:6 8 9 9 10 11



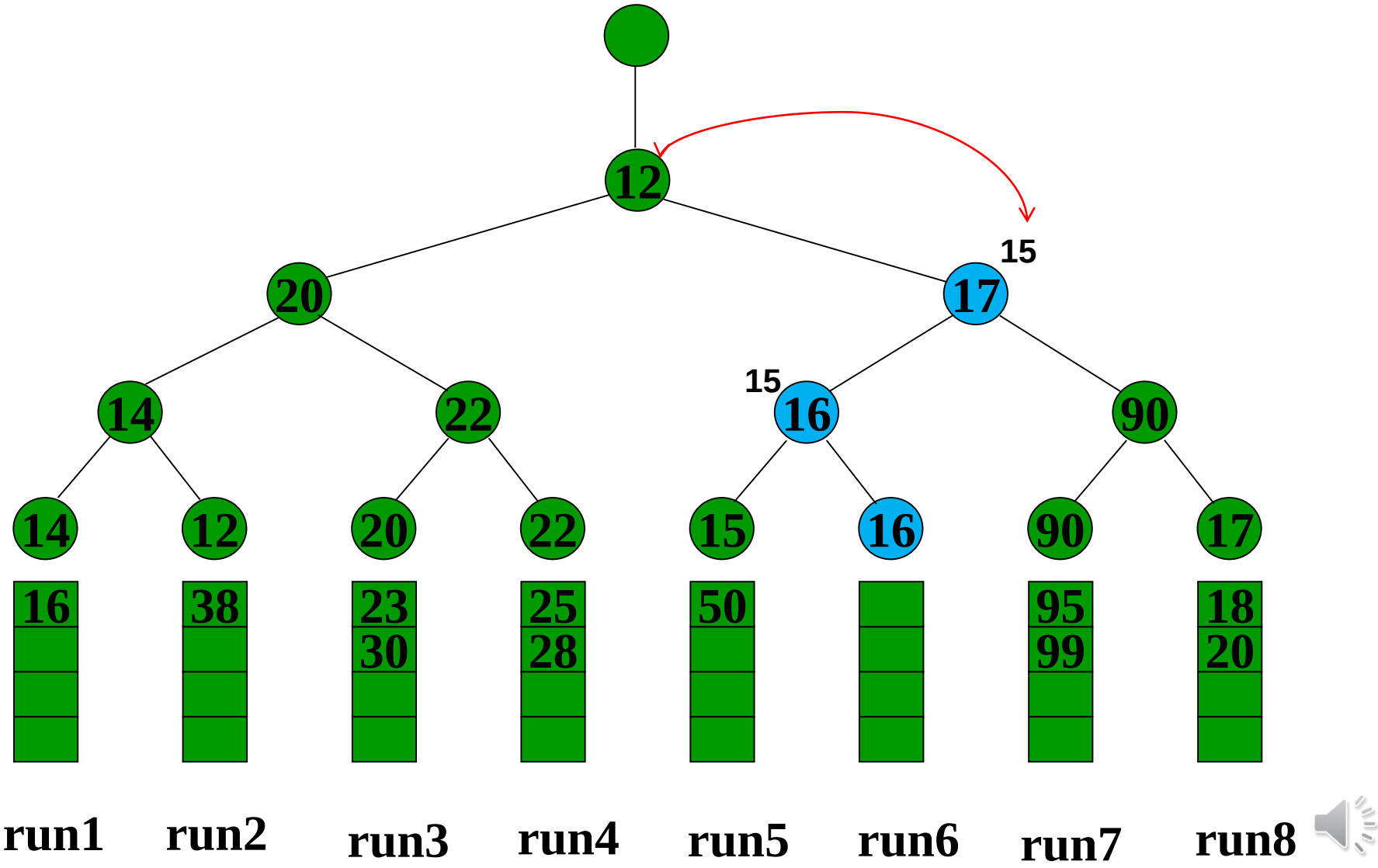
Loser Tree

Run:6 8 9 9 10 11



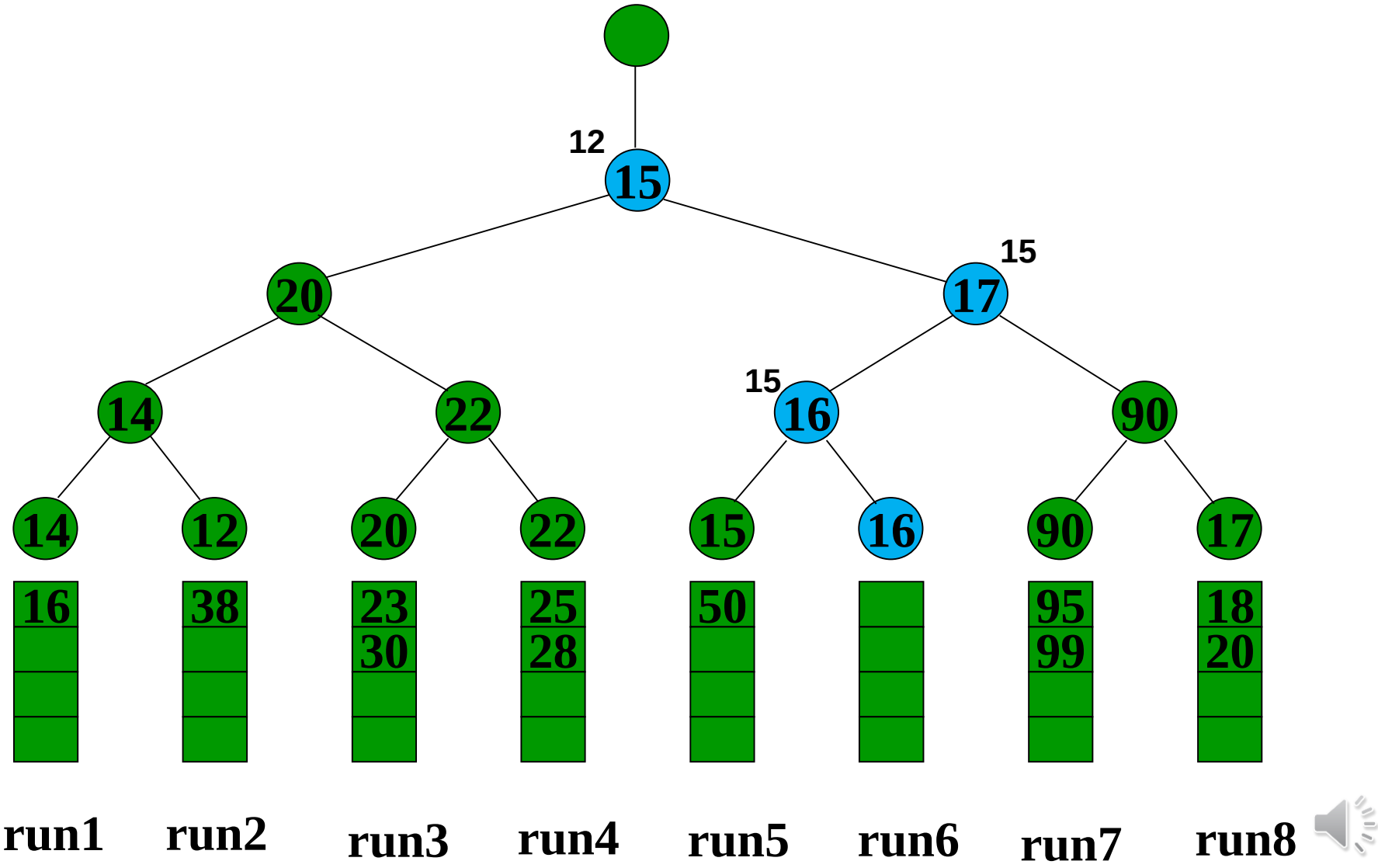
Loser Tree

Run:6 8 9 9 10 11



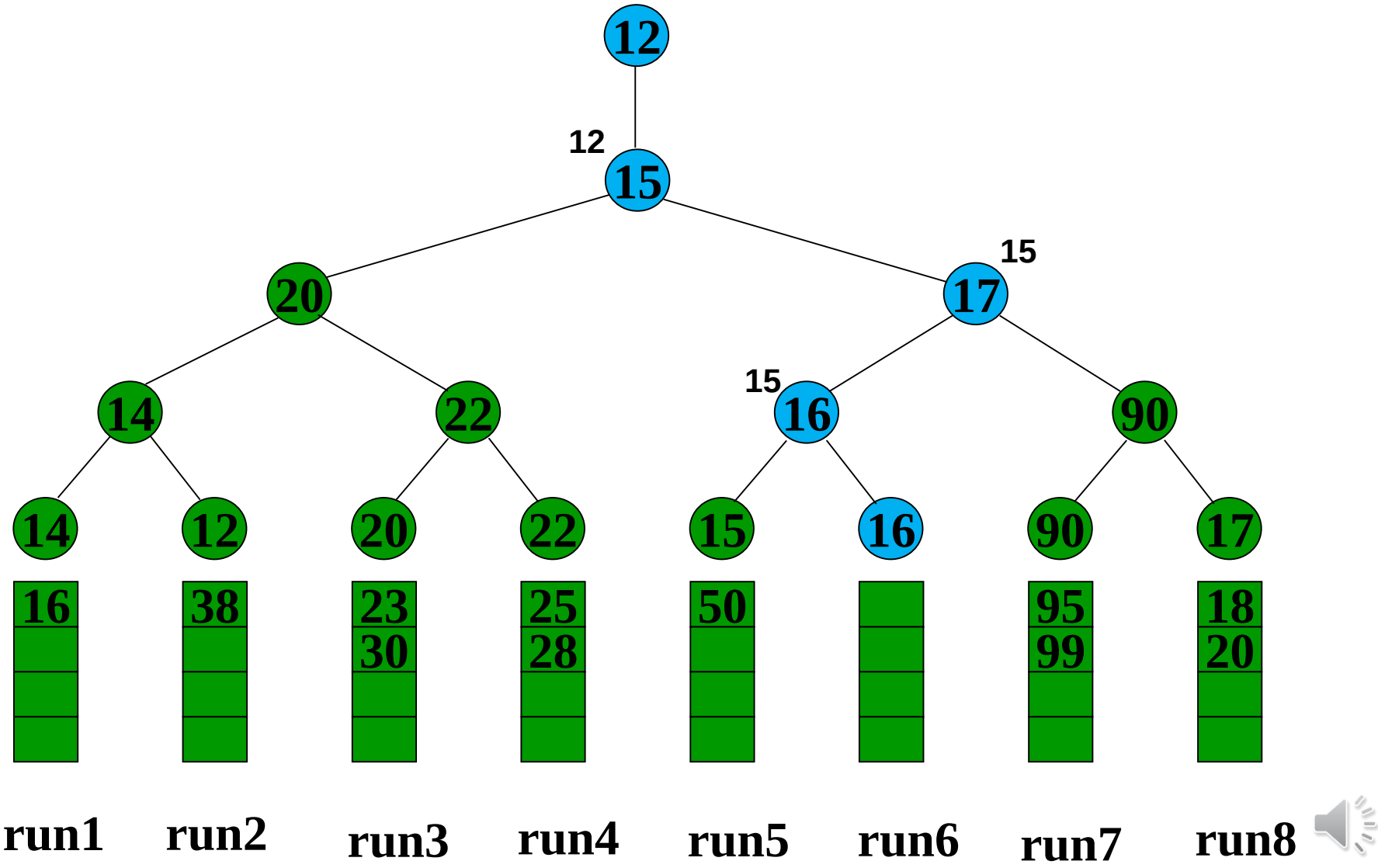
Loser Tree

Run:6 8 9 9 10 11



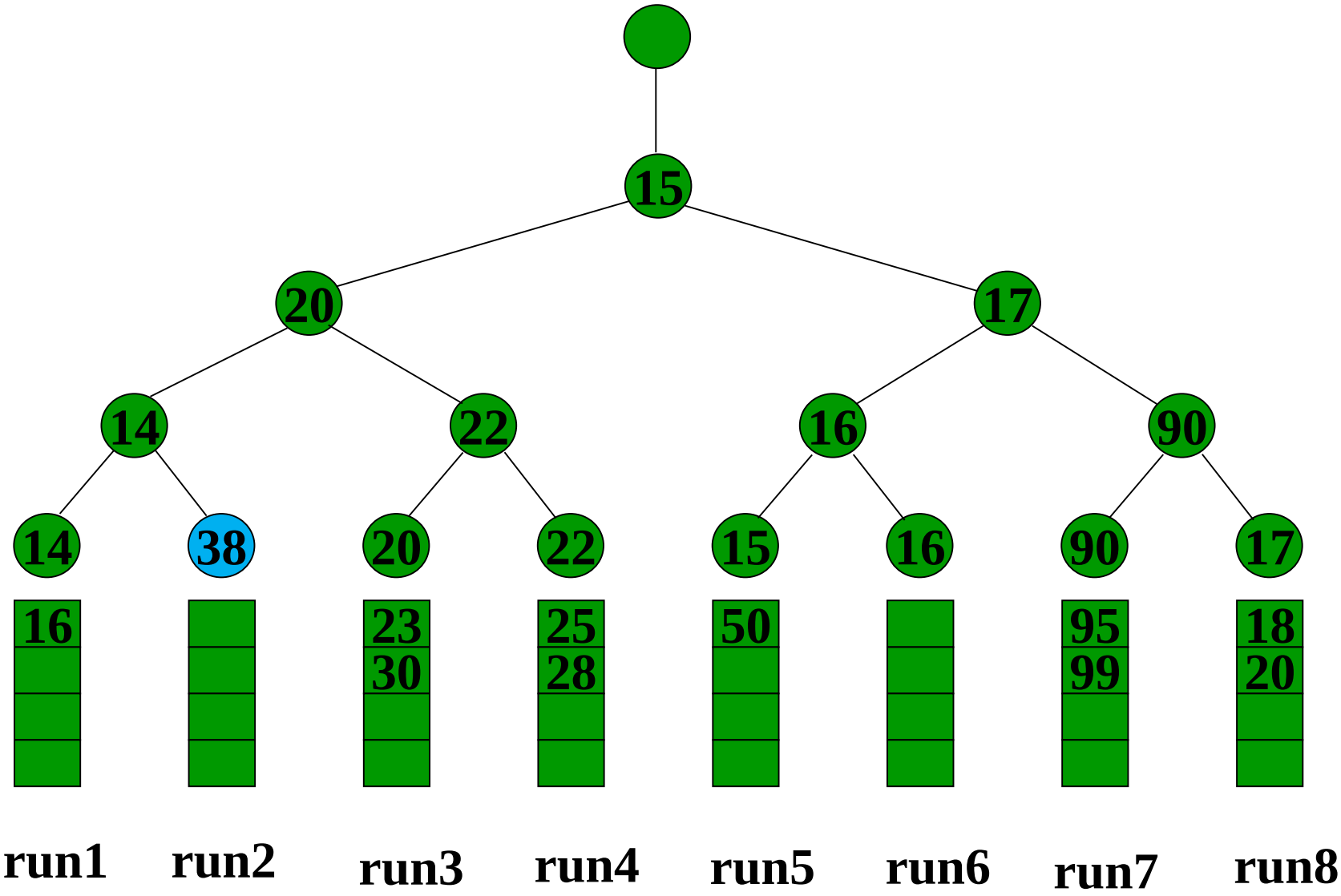
Loser Tree

```
Run:6  8  9  9  10  11  12
```



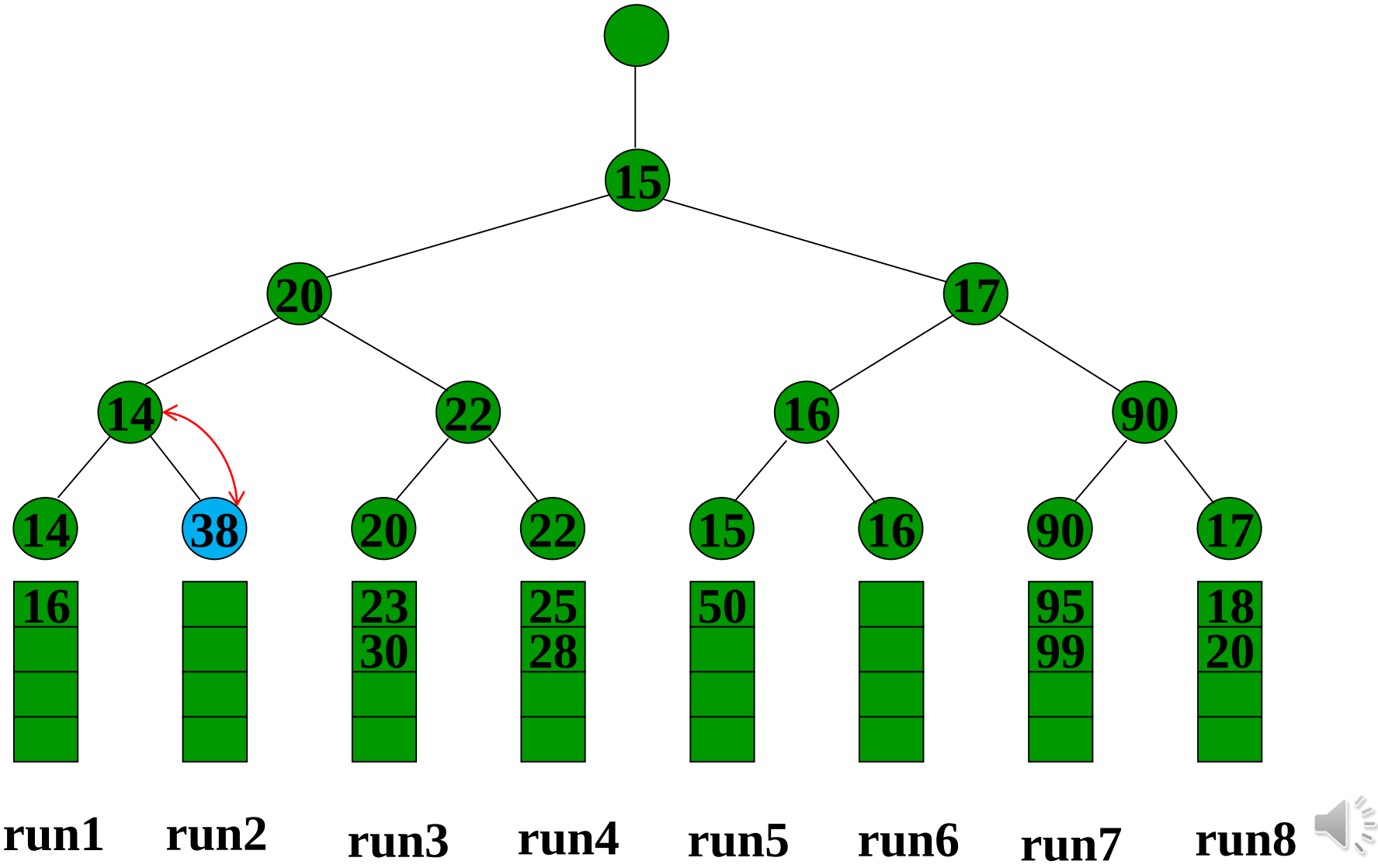
Loser Tree

```
Run:6  8  9  9  10  11  12
```



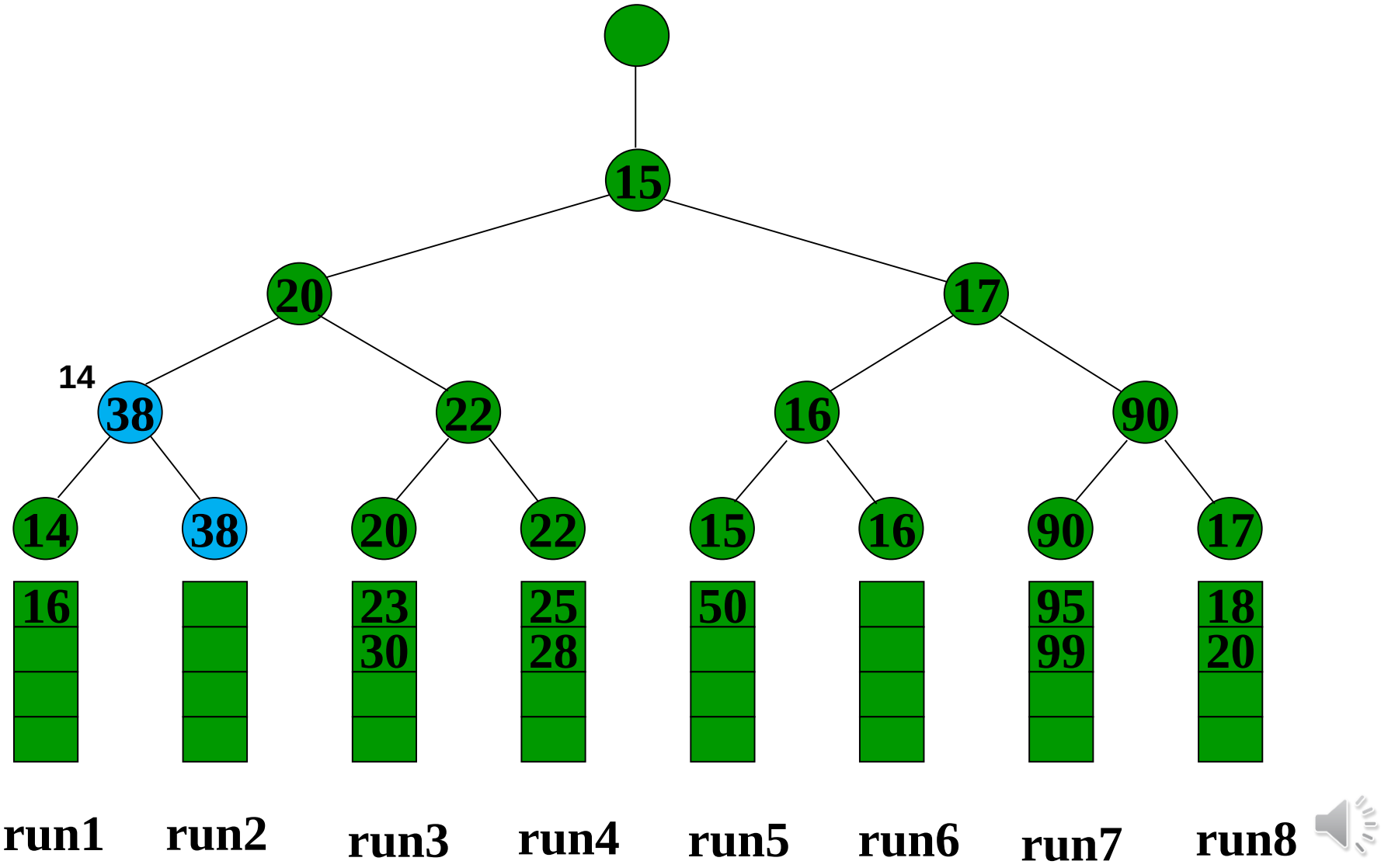
Loser Tree

```
Run:6  8  9  9  10  11  12
```



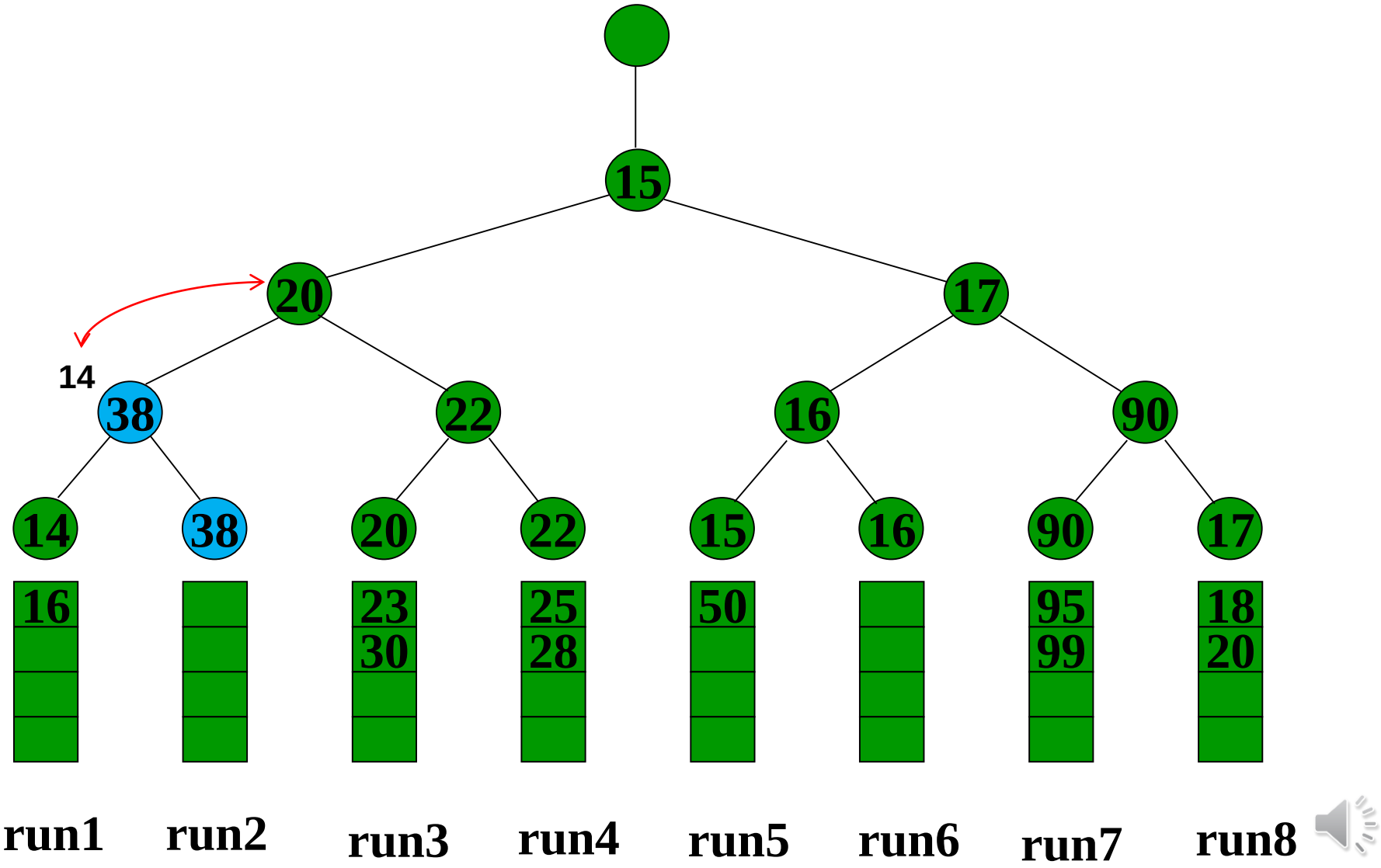
Loser Tree

```
Run:6  8  9  9  10  11  12
```



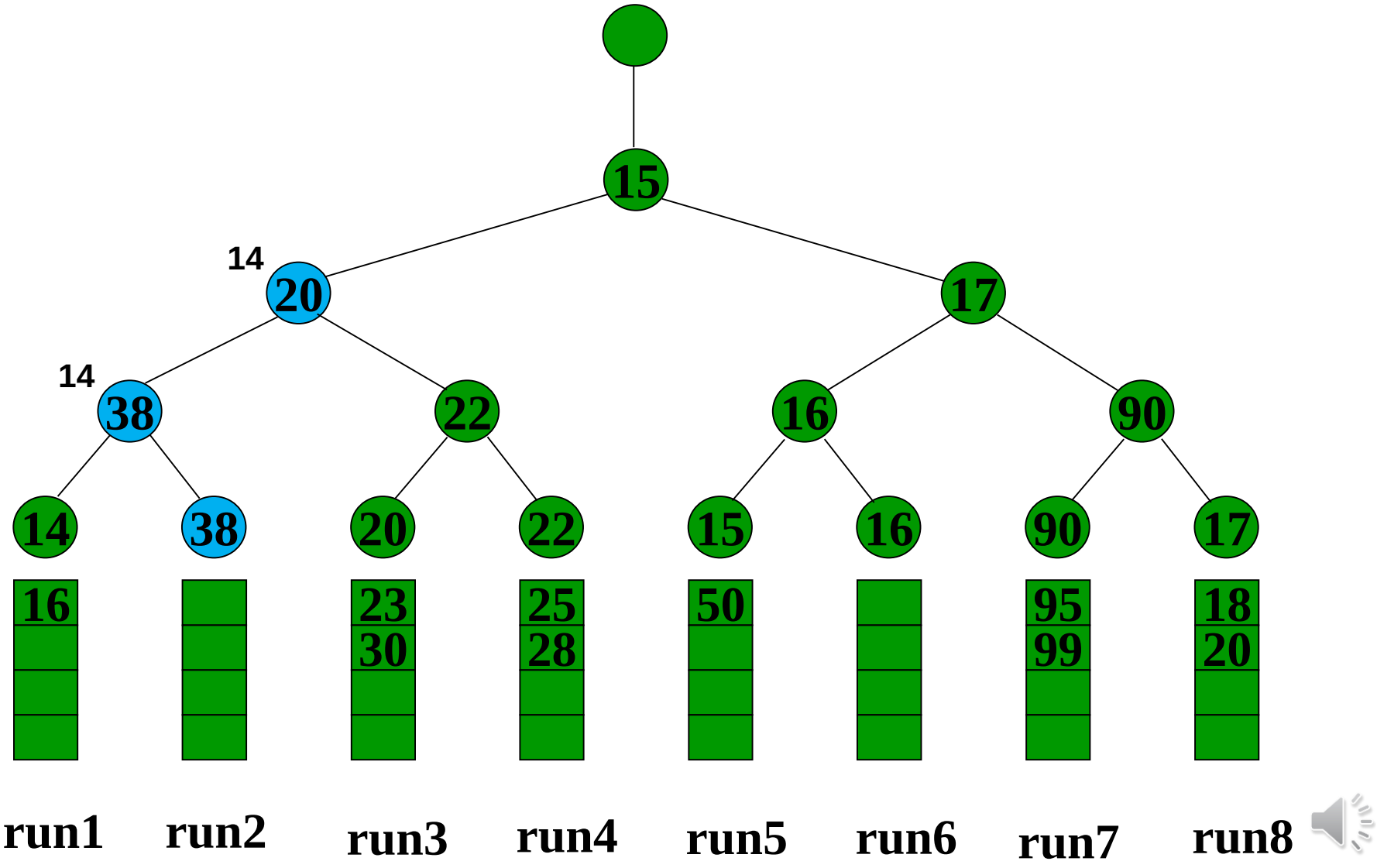
Loser Tree

```
Run:6  8  9  9  10  11  12
```



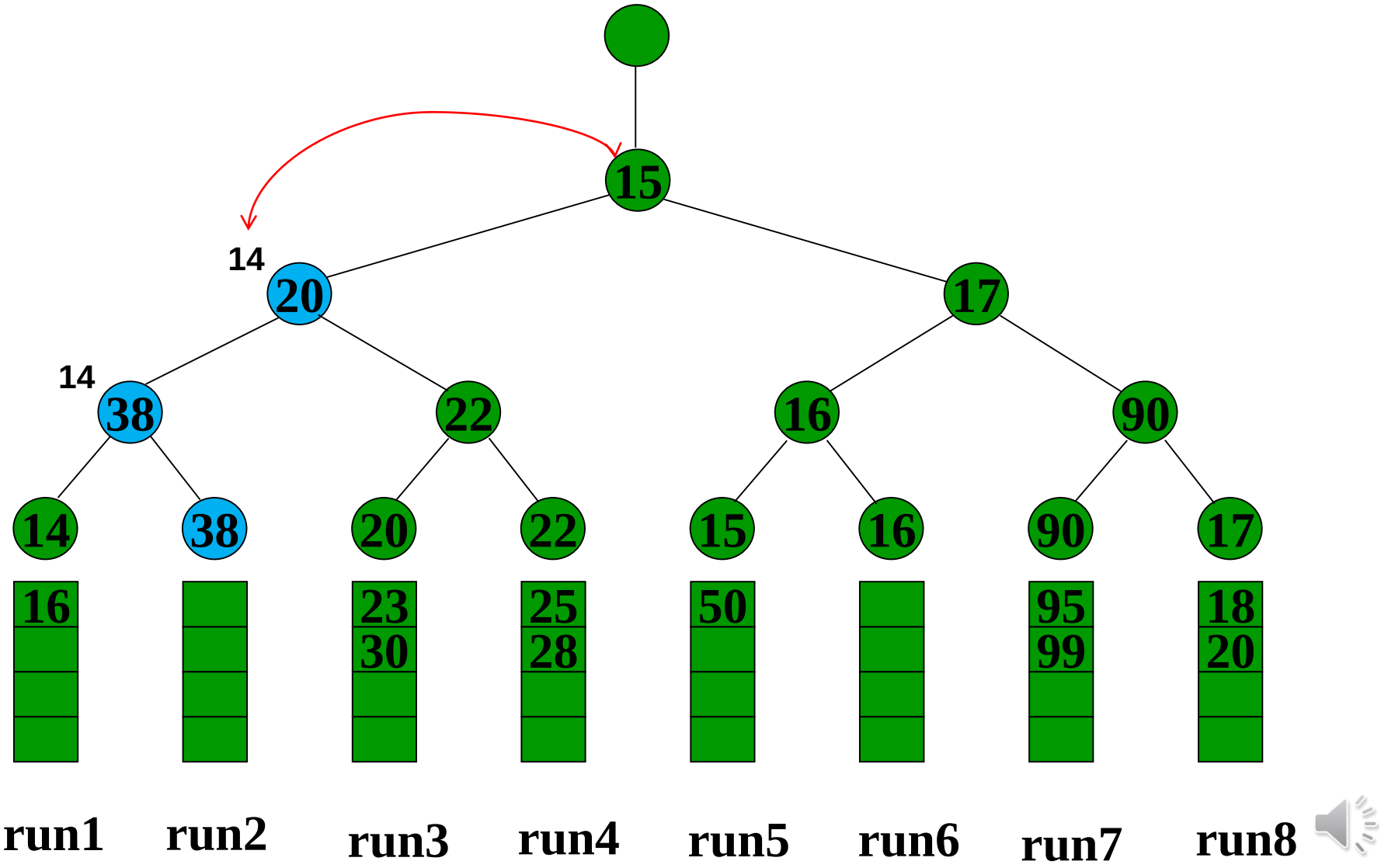
Loser Tree

Run:6 8 9 9 10 11 12



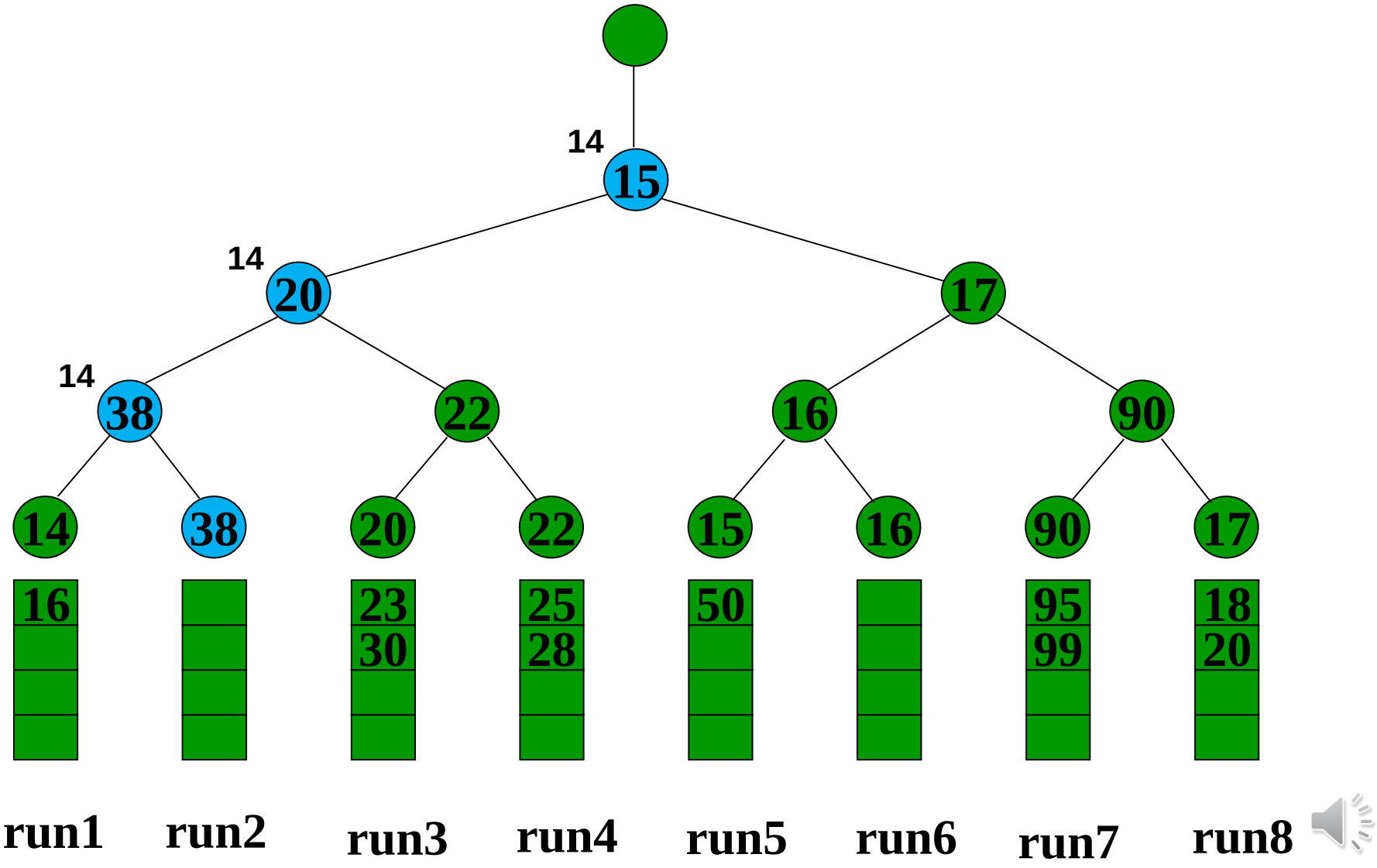
Loser Tree

```
Run:6  8  9  9  10  11  12
```



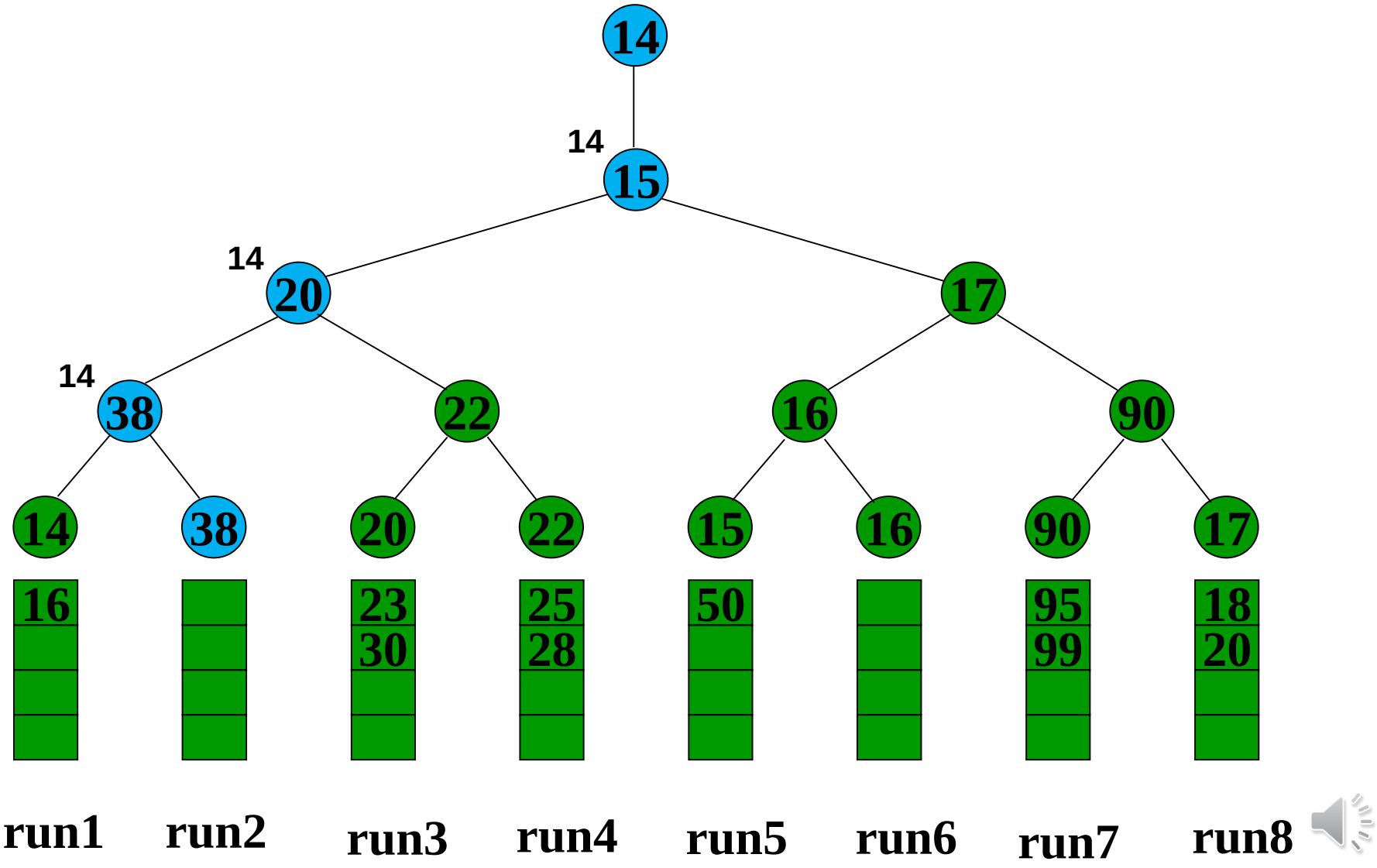
Loser Tree

Run:6 8 9 9 10 11 12



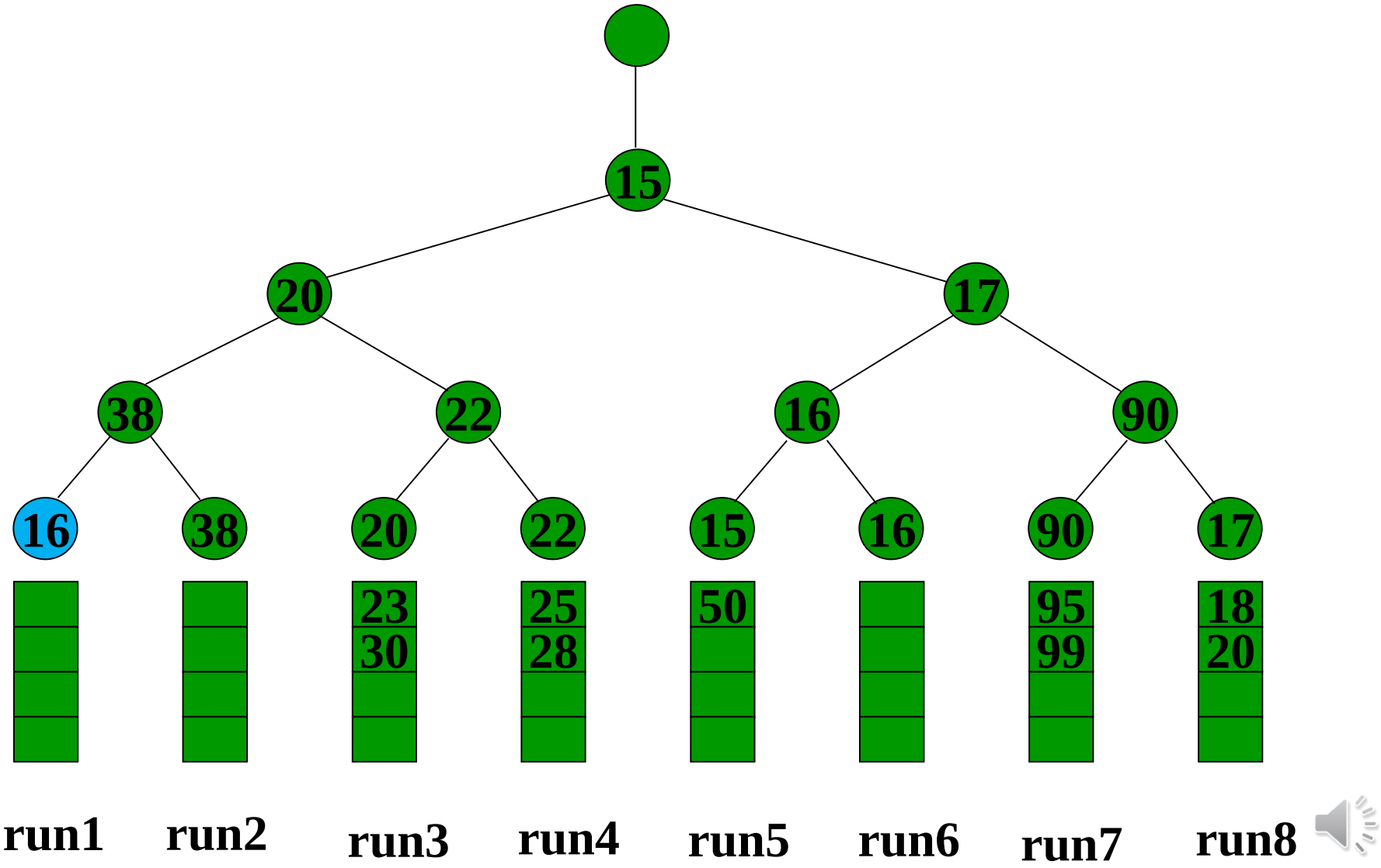
Loser Tree

Run:6 8 9 9 10 11 12 14



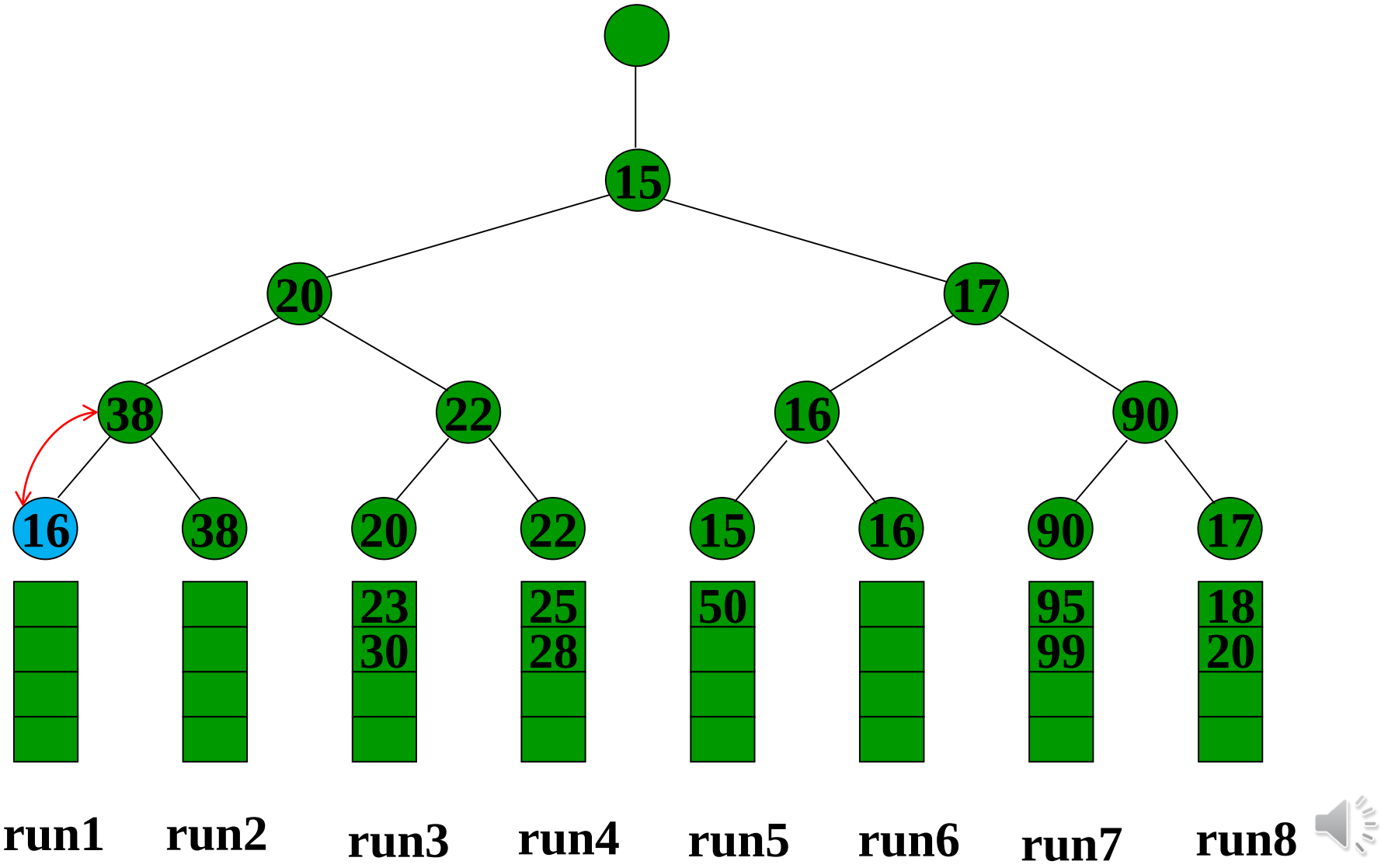
Loser Tree

Run:6 8 9 9 10 11 12 14



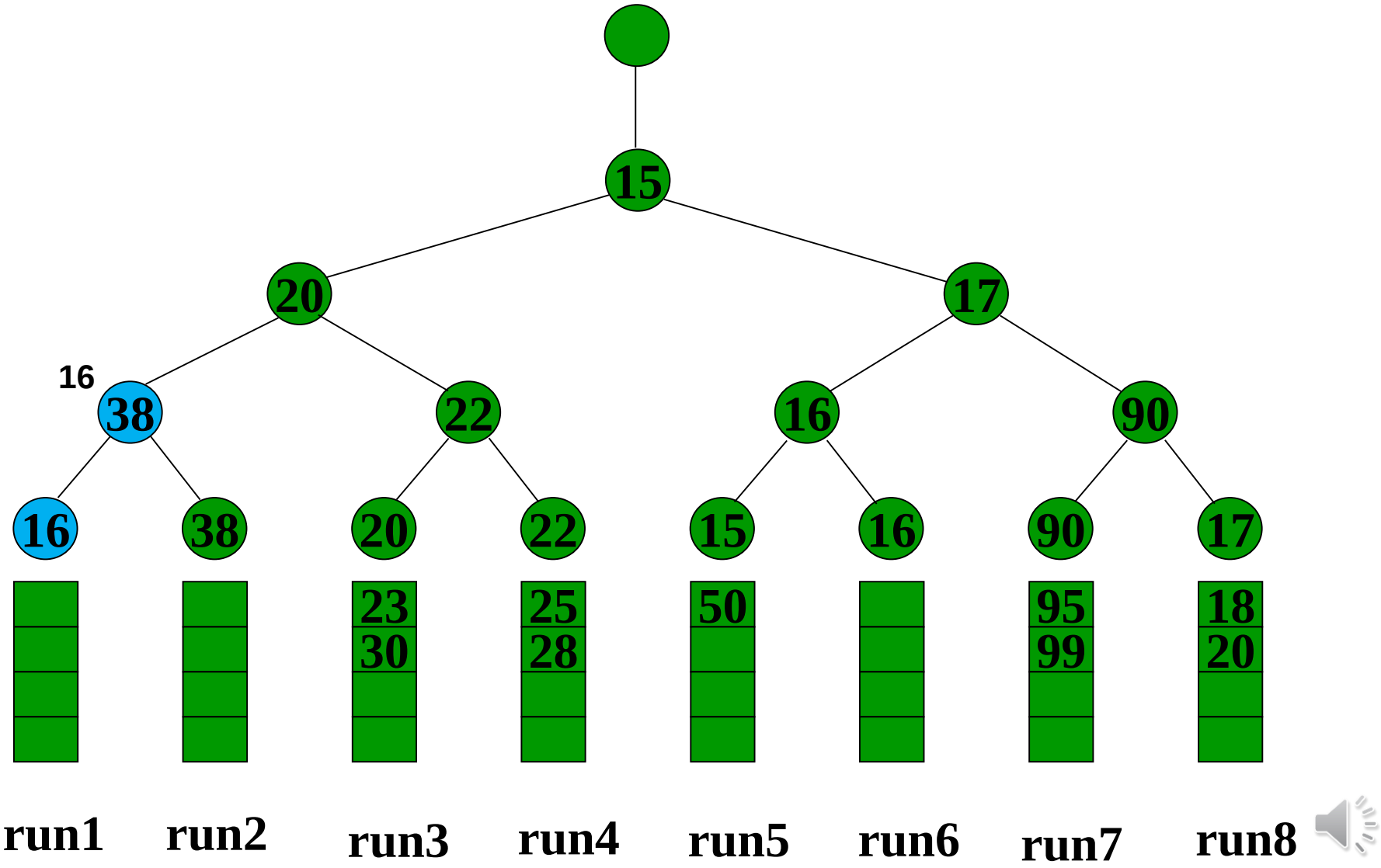
Loser Tree

Run: 6 8 9 9 10 11 12 14



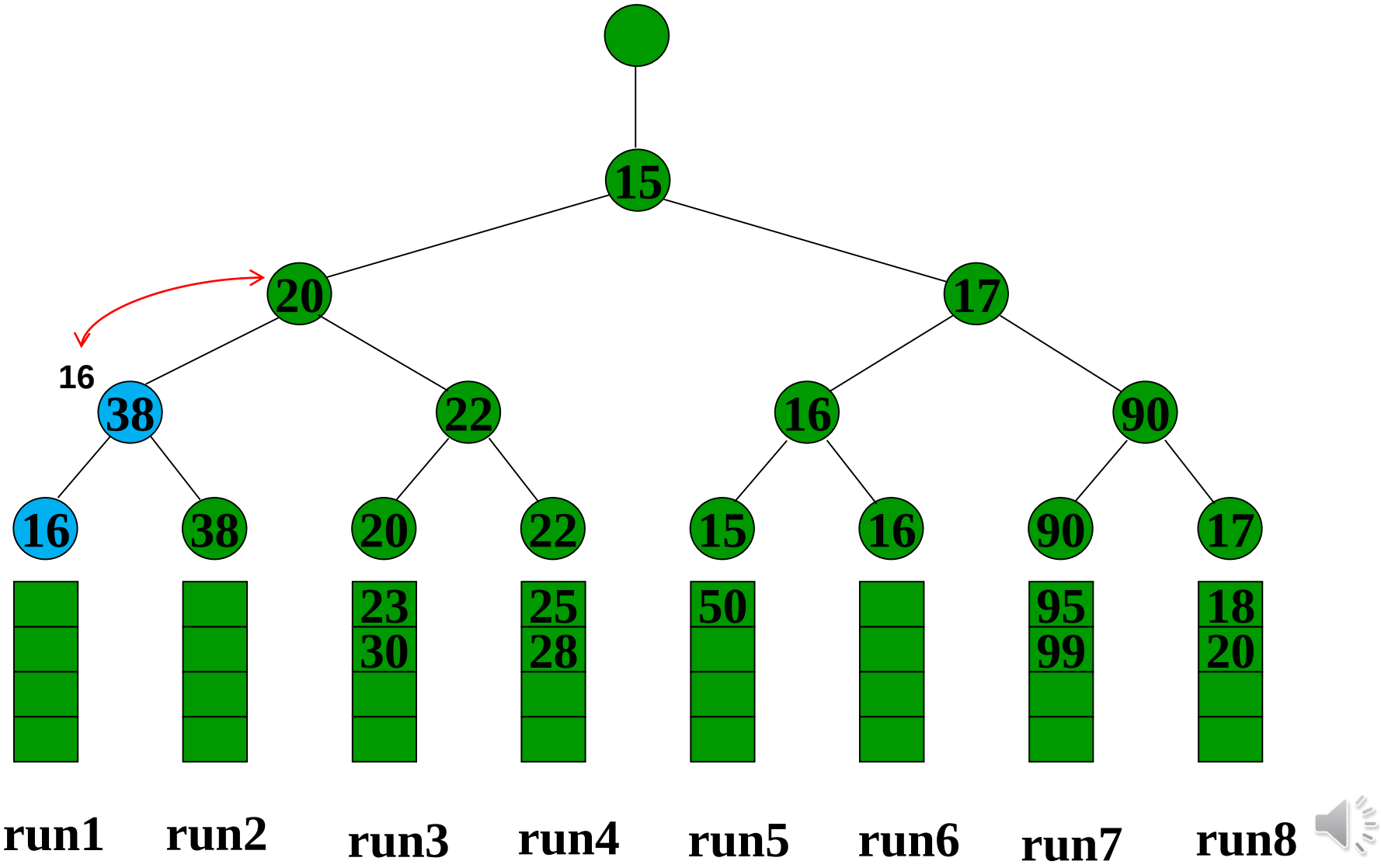
Loser Tree

Run:6 8 9 9 10 11 12 14



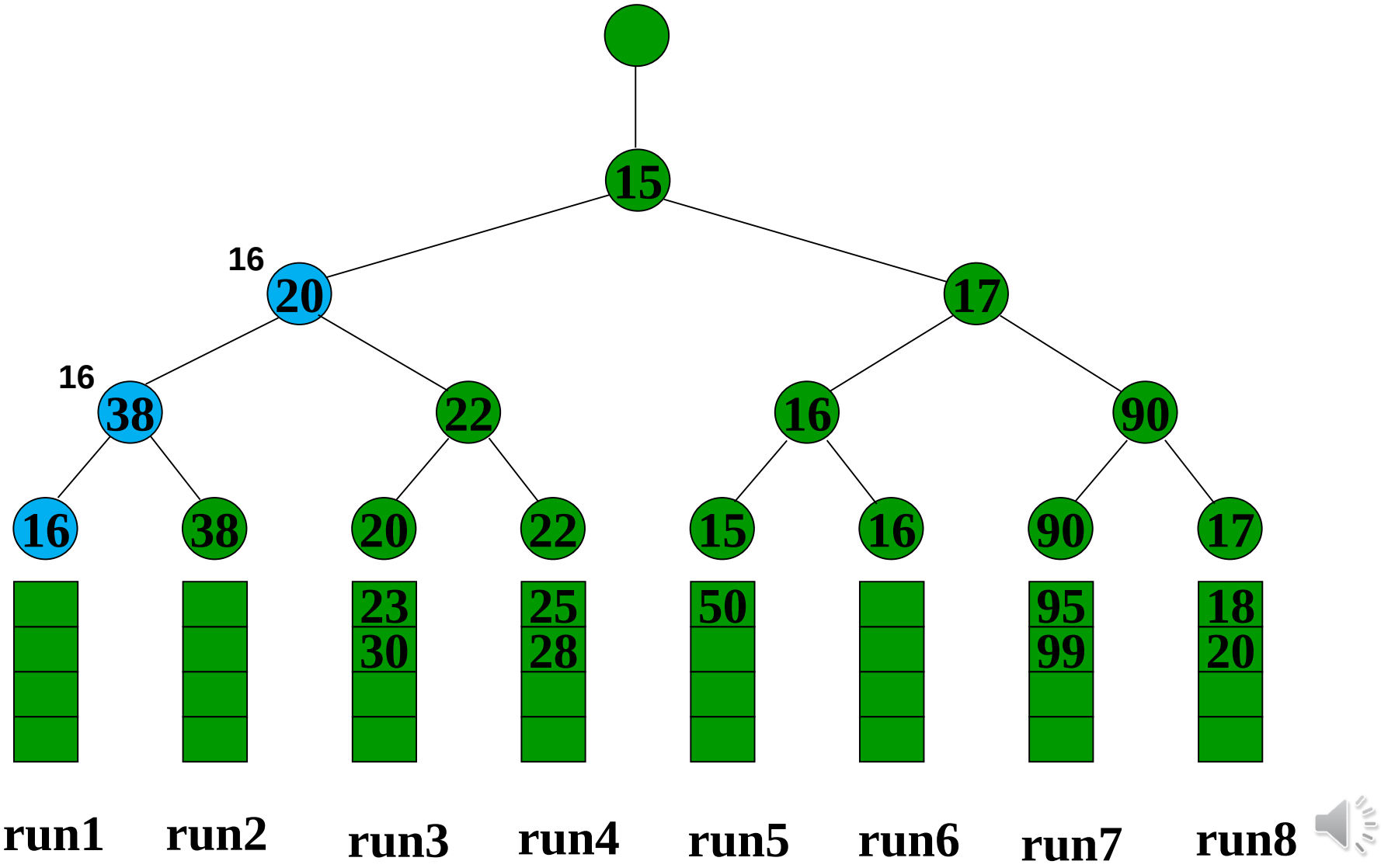
Loser Tree

```
Run:6  8  9  9  10  11  12  14
```



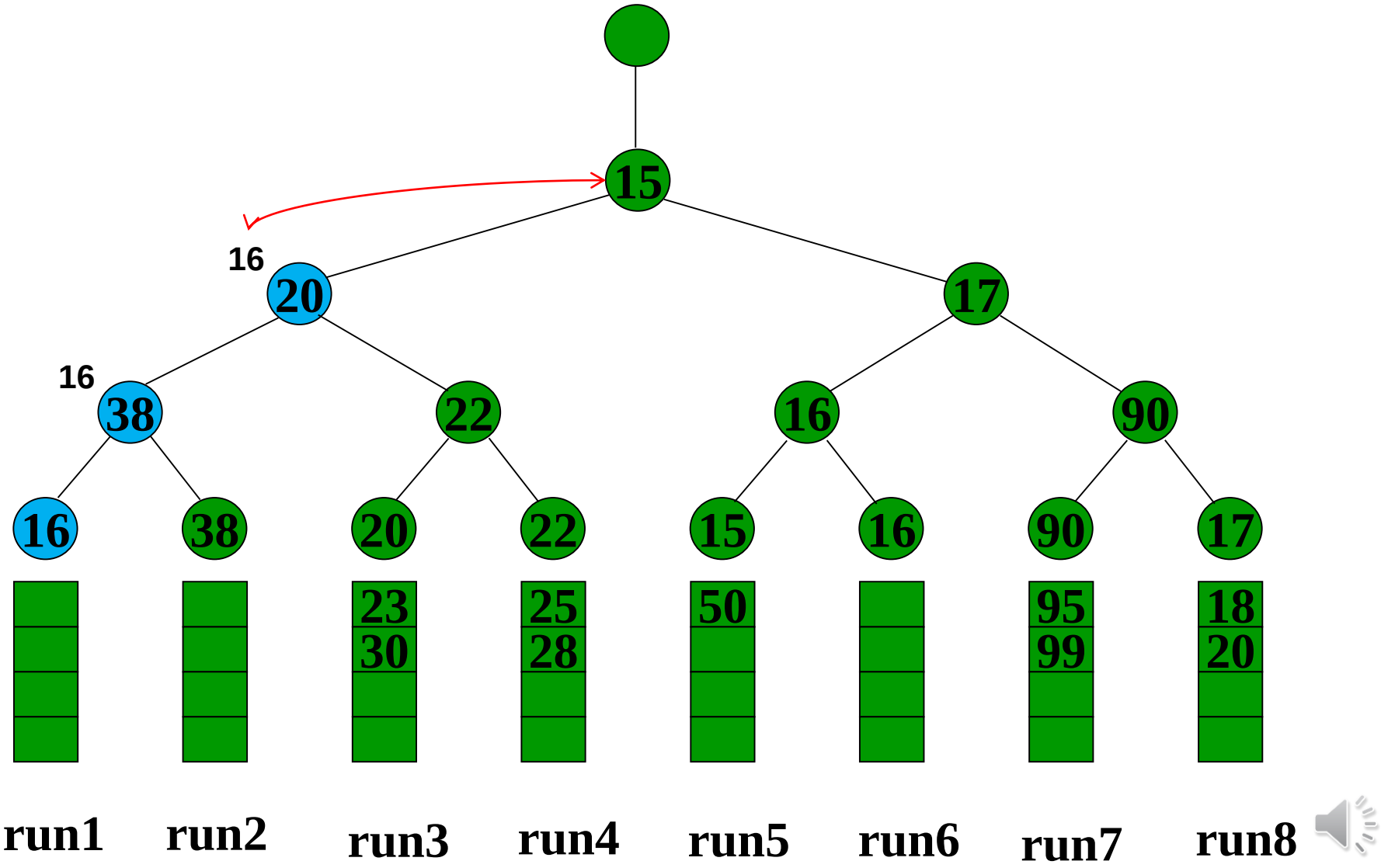
Loser Tree

Run: 6 8 9 9 10 11 12 14



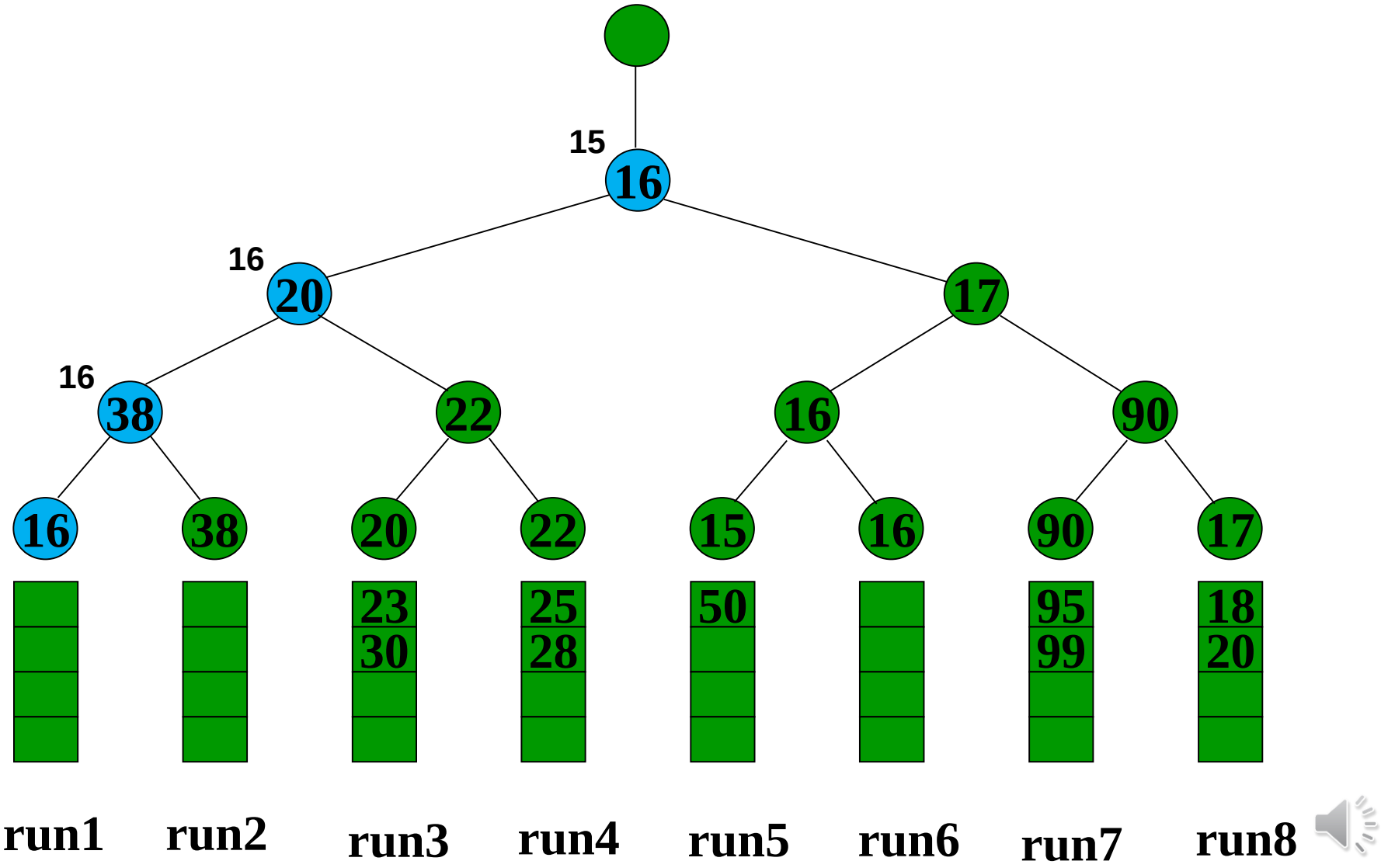
Loser Tree

```
Run:6  8  9  9  10  11  12  14
```



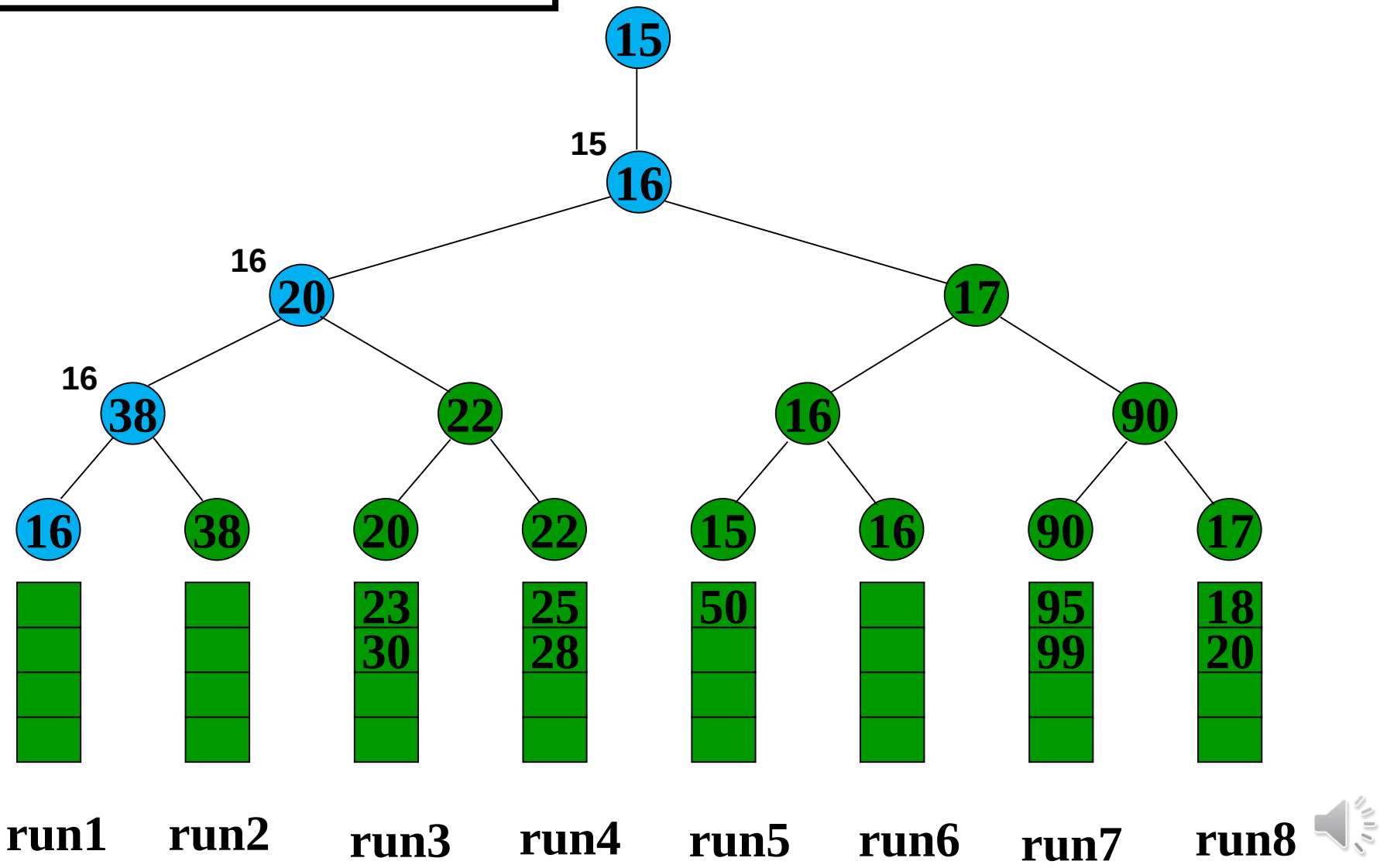
Loser Tree

Run: 6 8 9 9 10 11 12 14



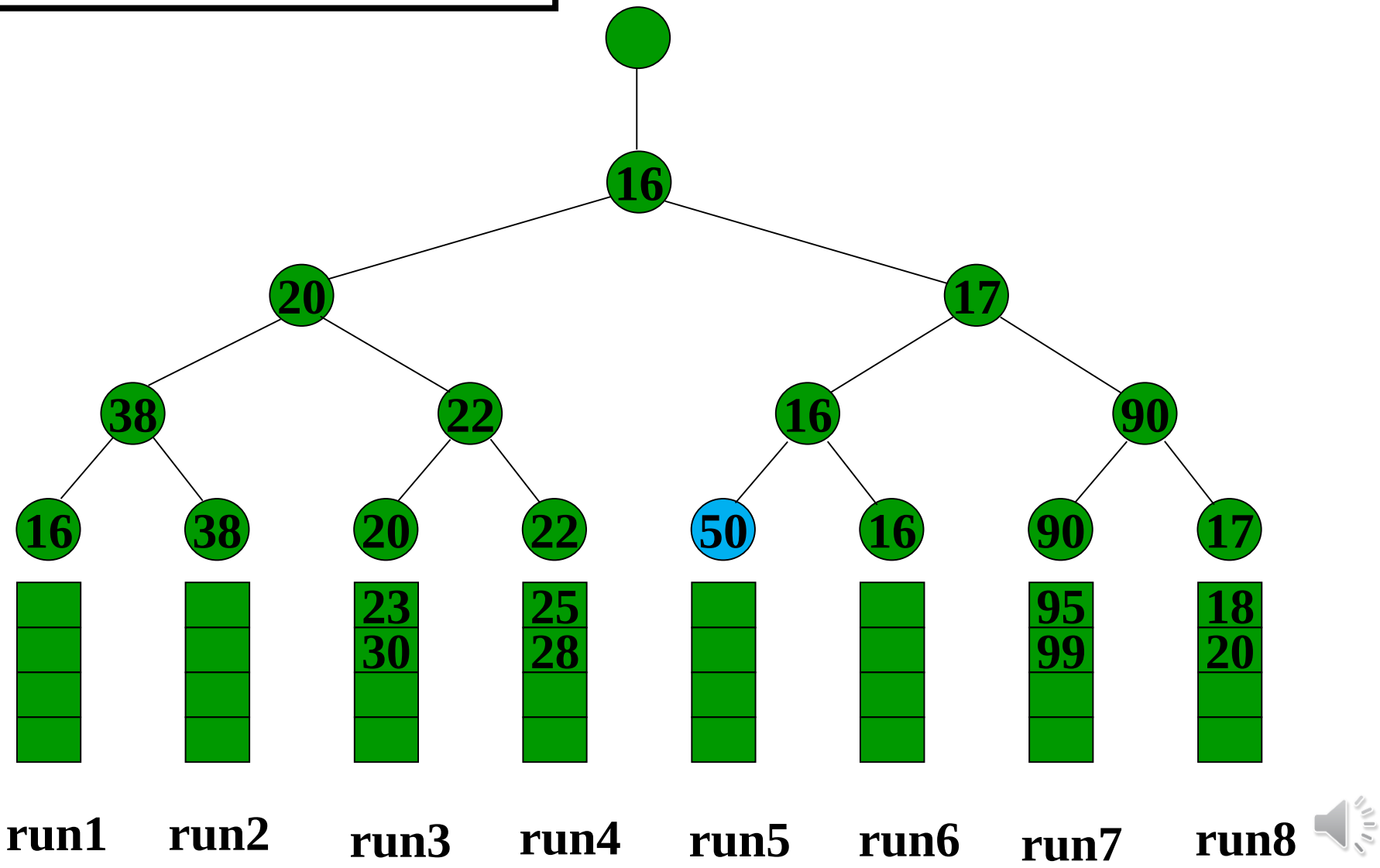
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



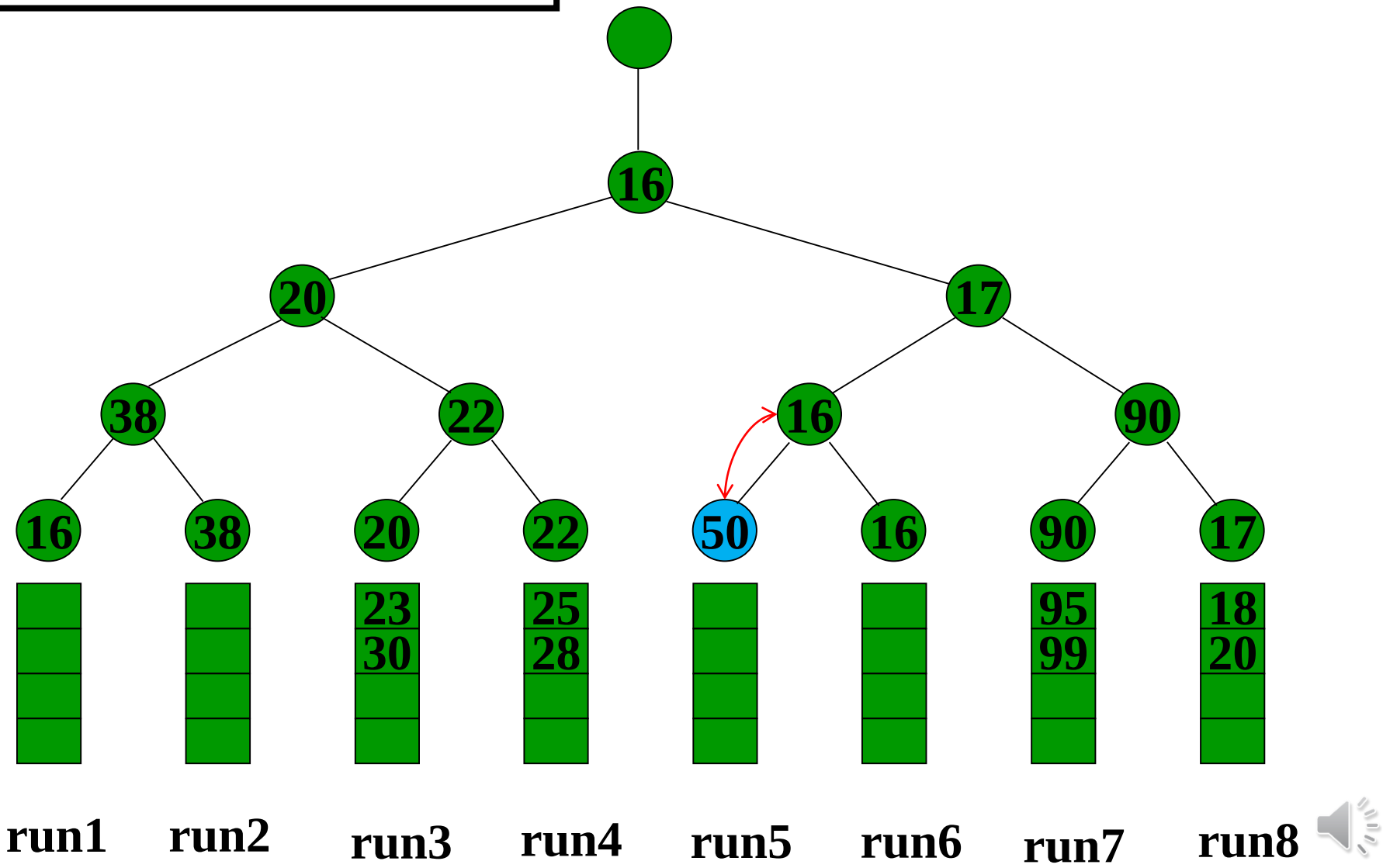
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



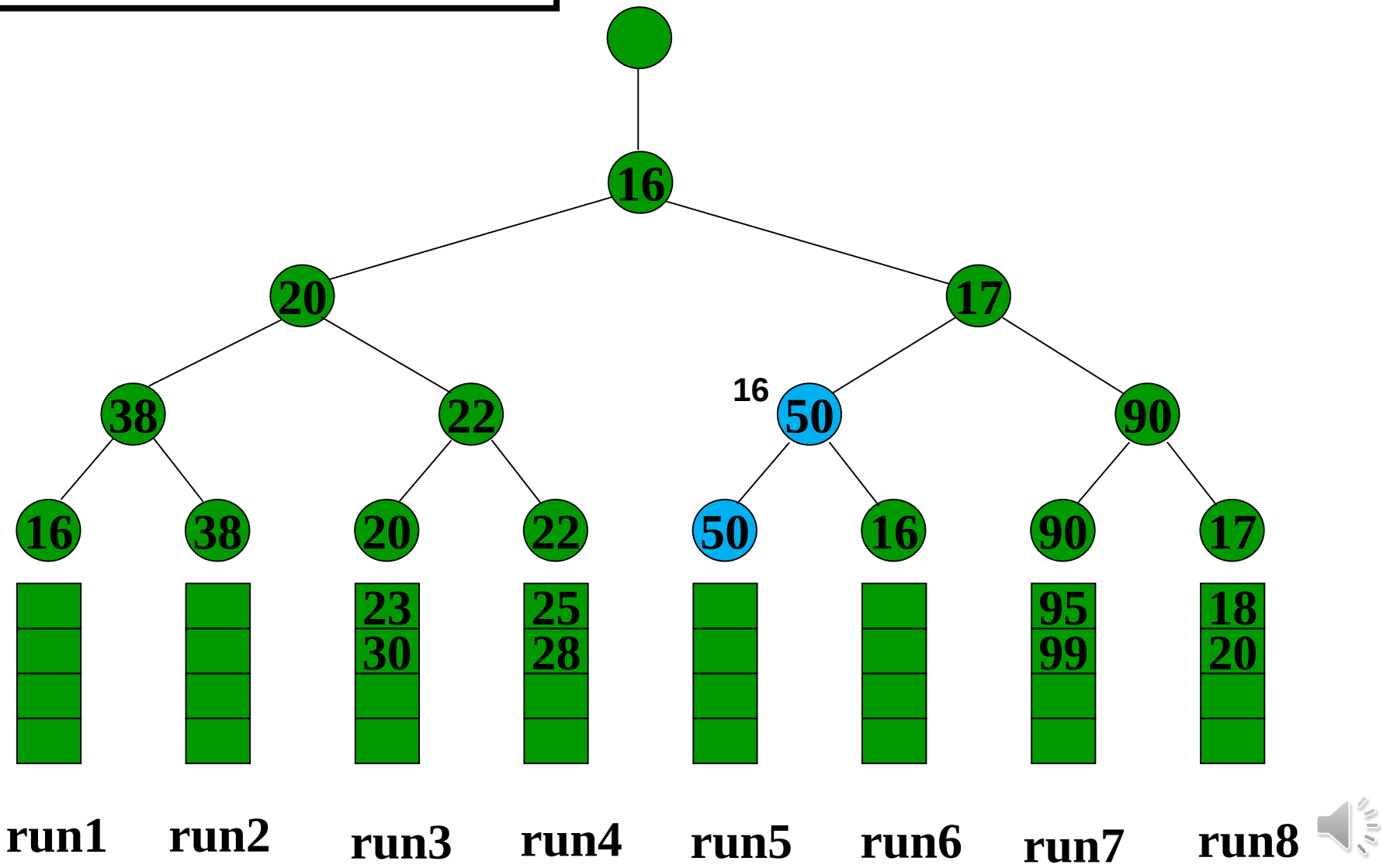
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



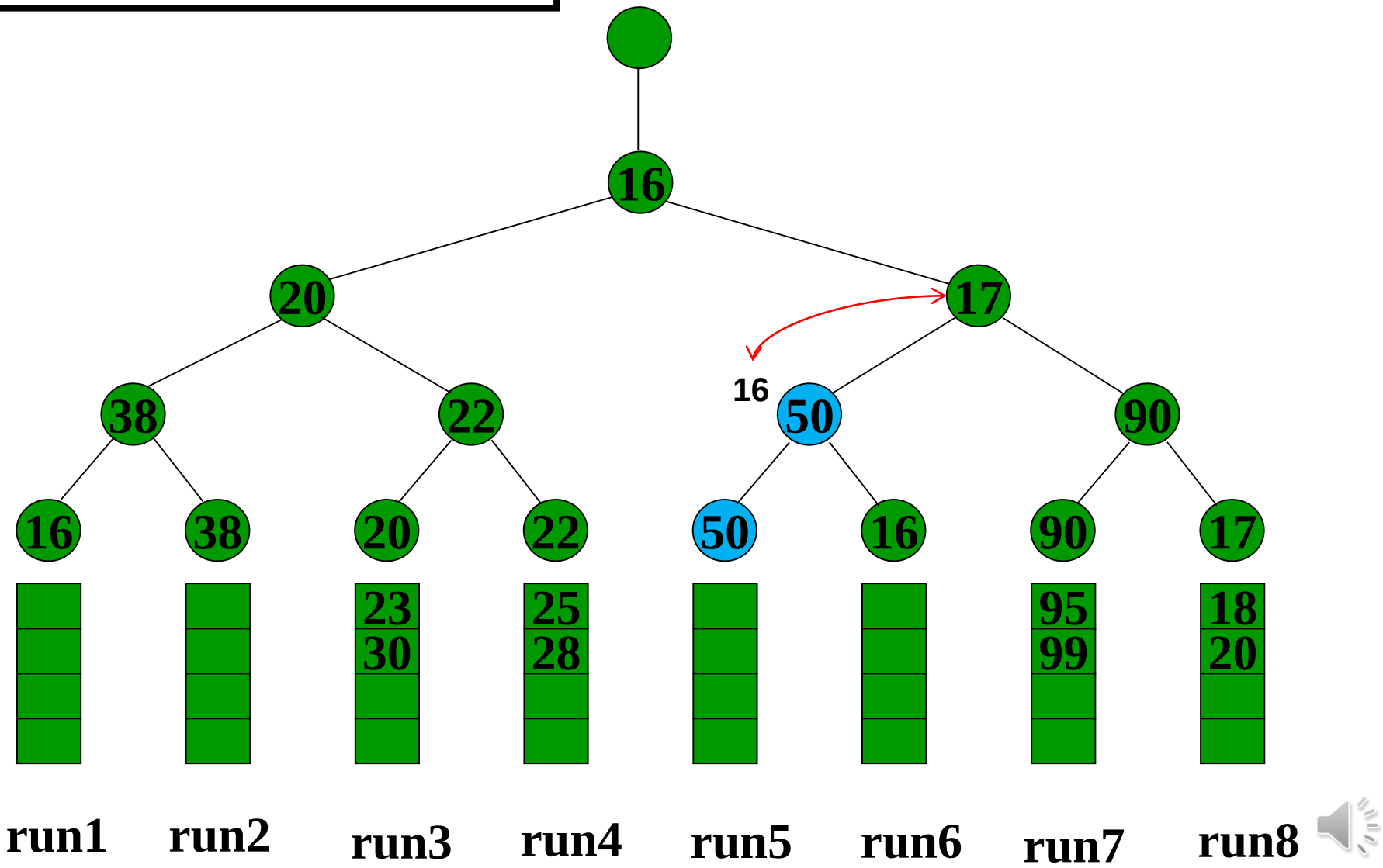
Loser Tree

```
Run:6  8  9  9  10  11  12  14
15
```



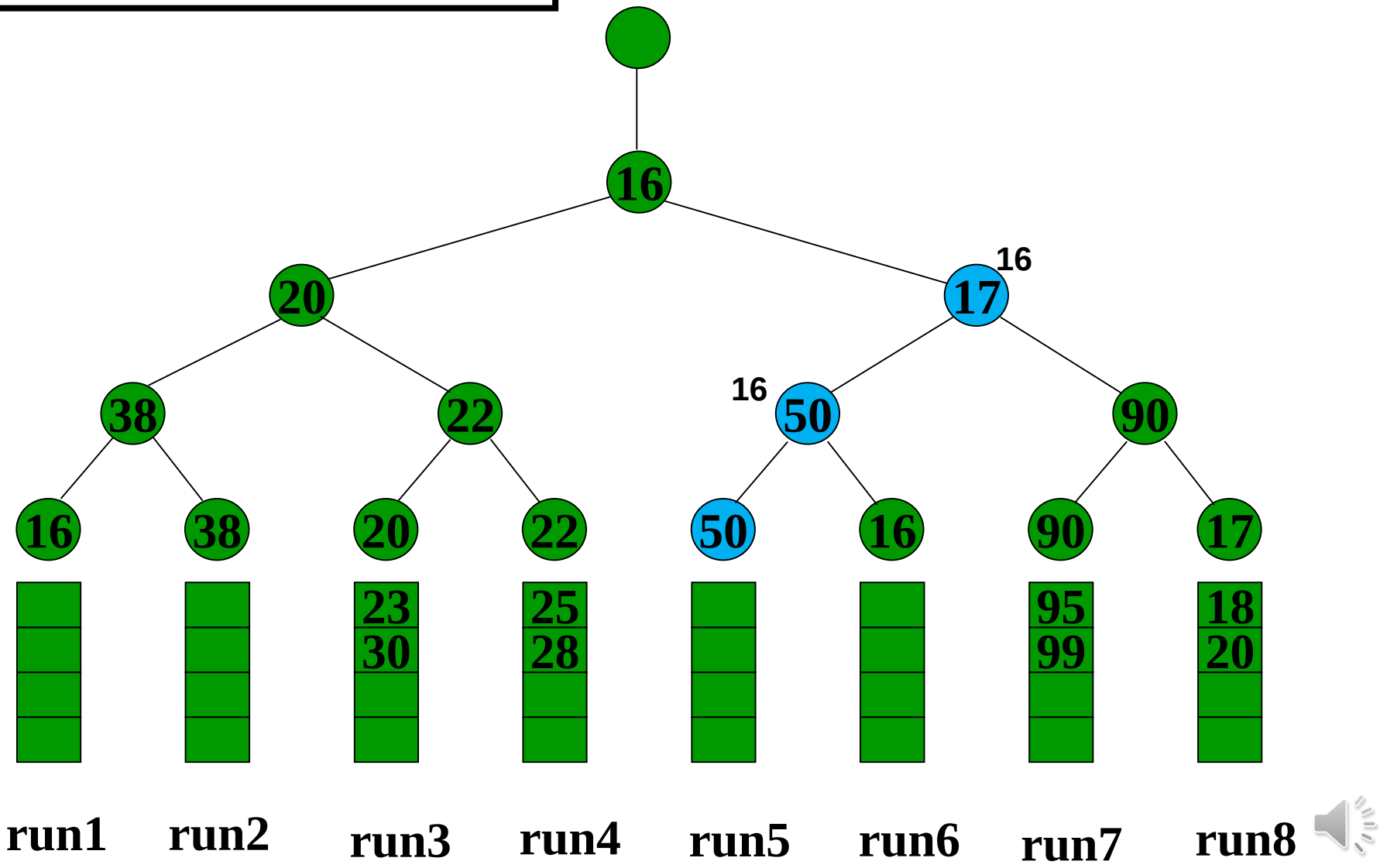
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



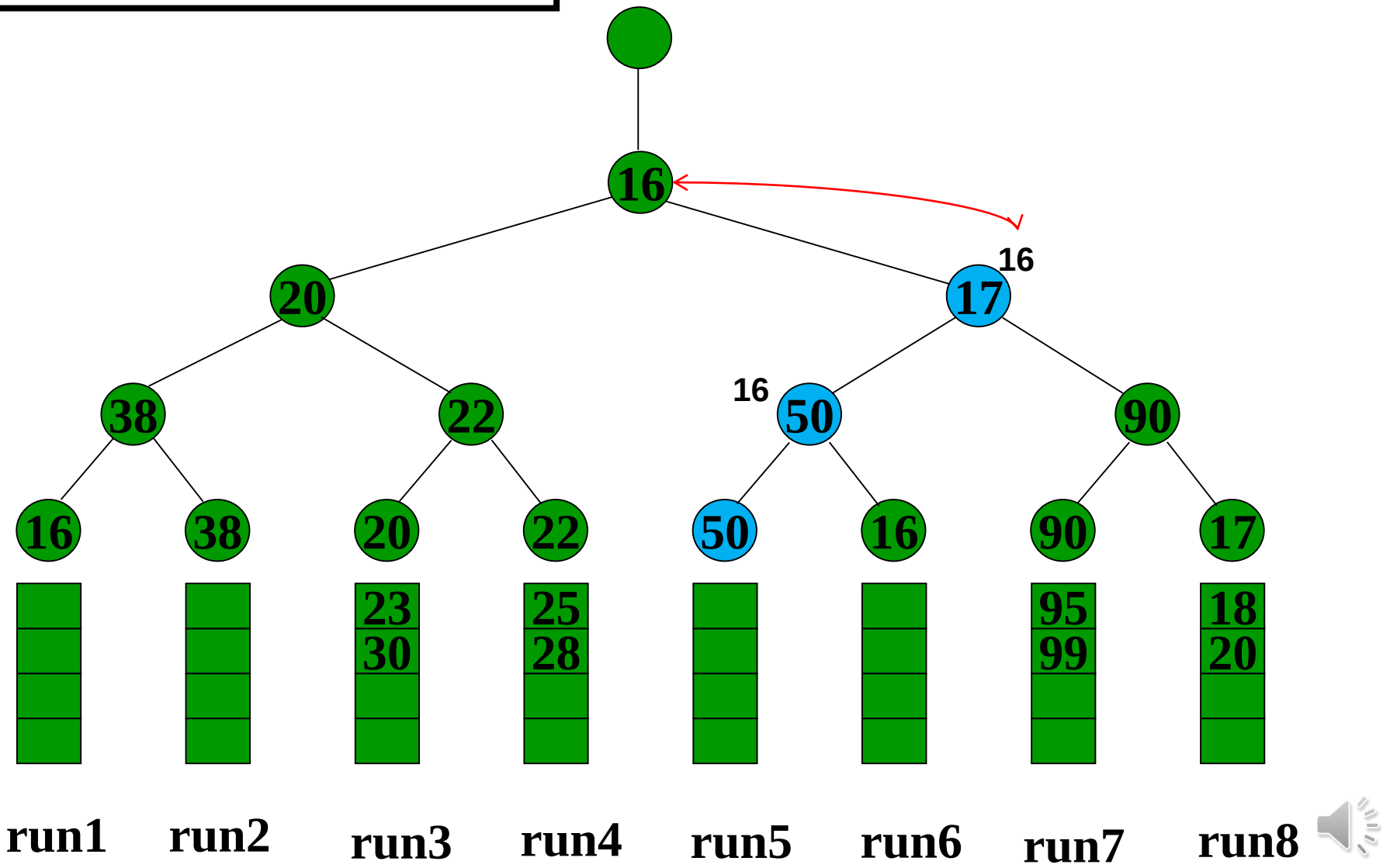
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



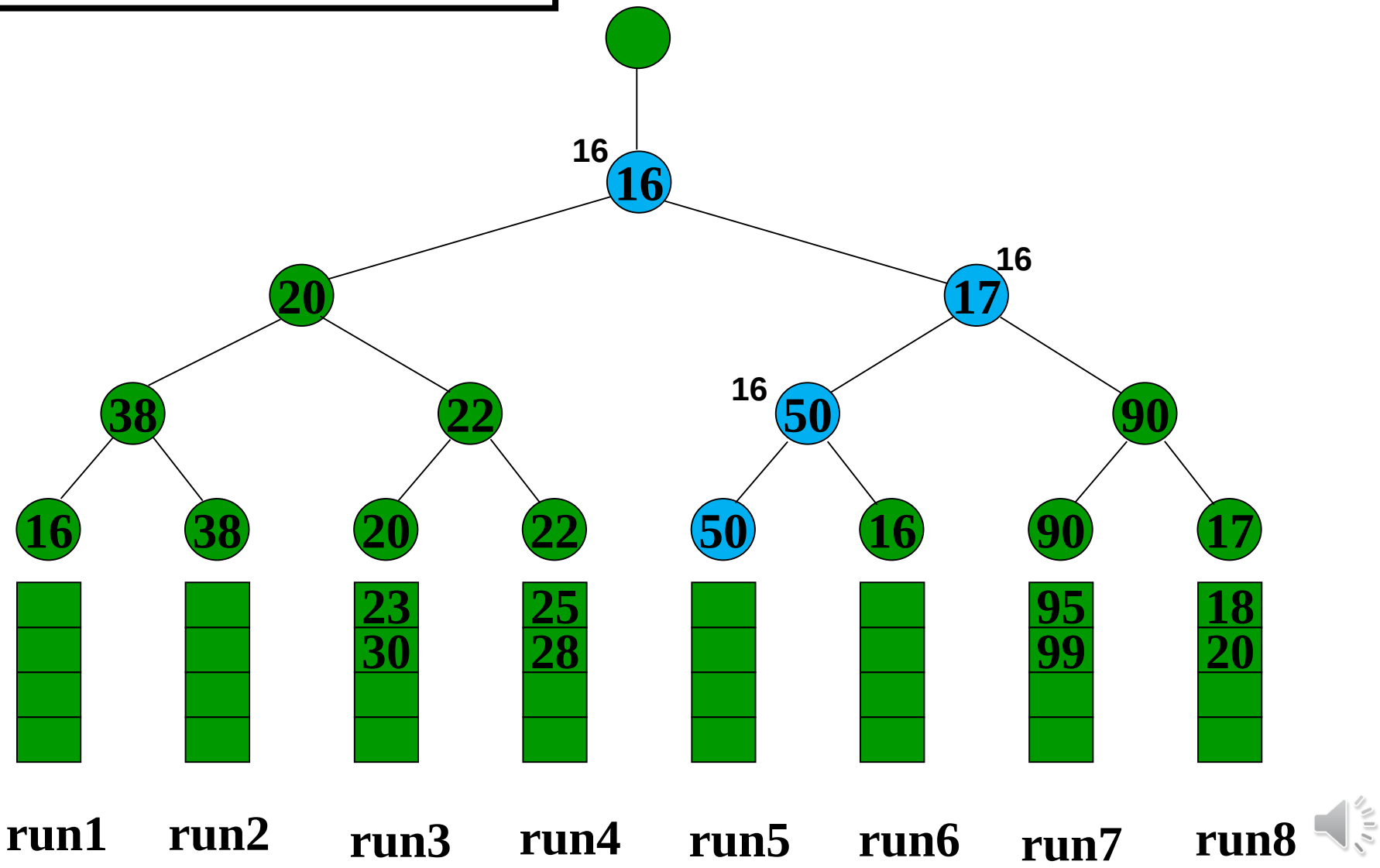
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



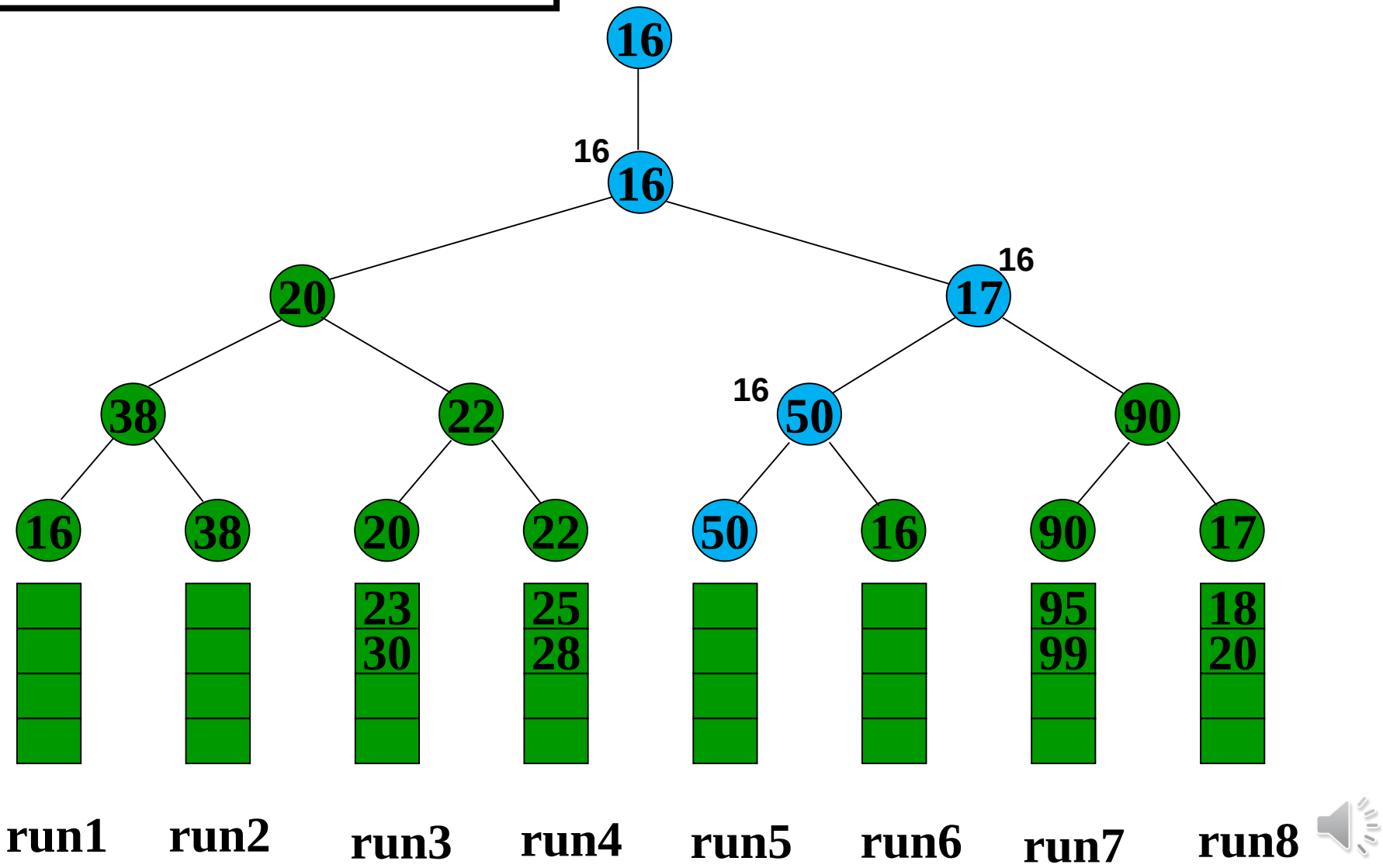
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15
```



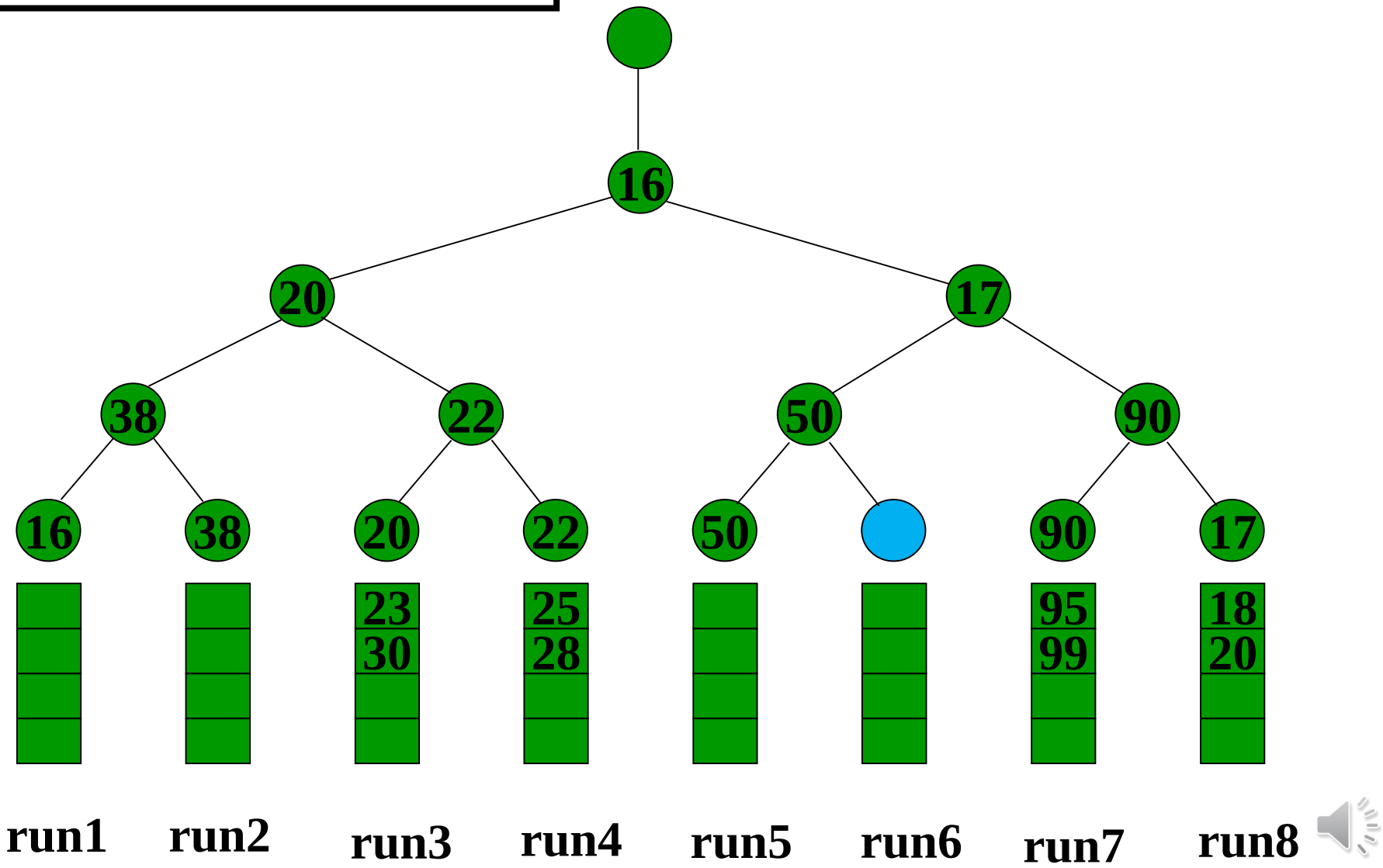
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



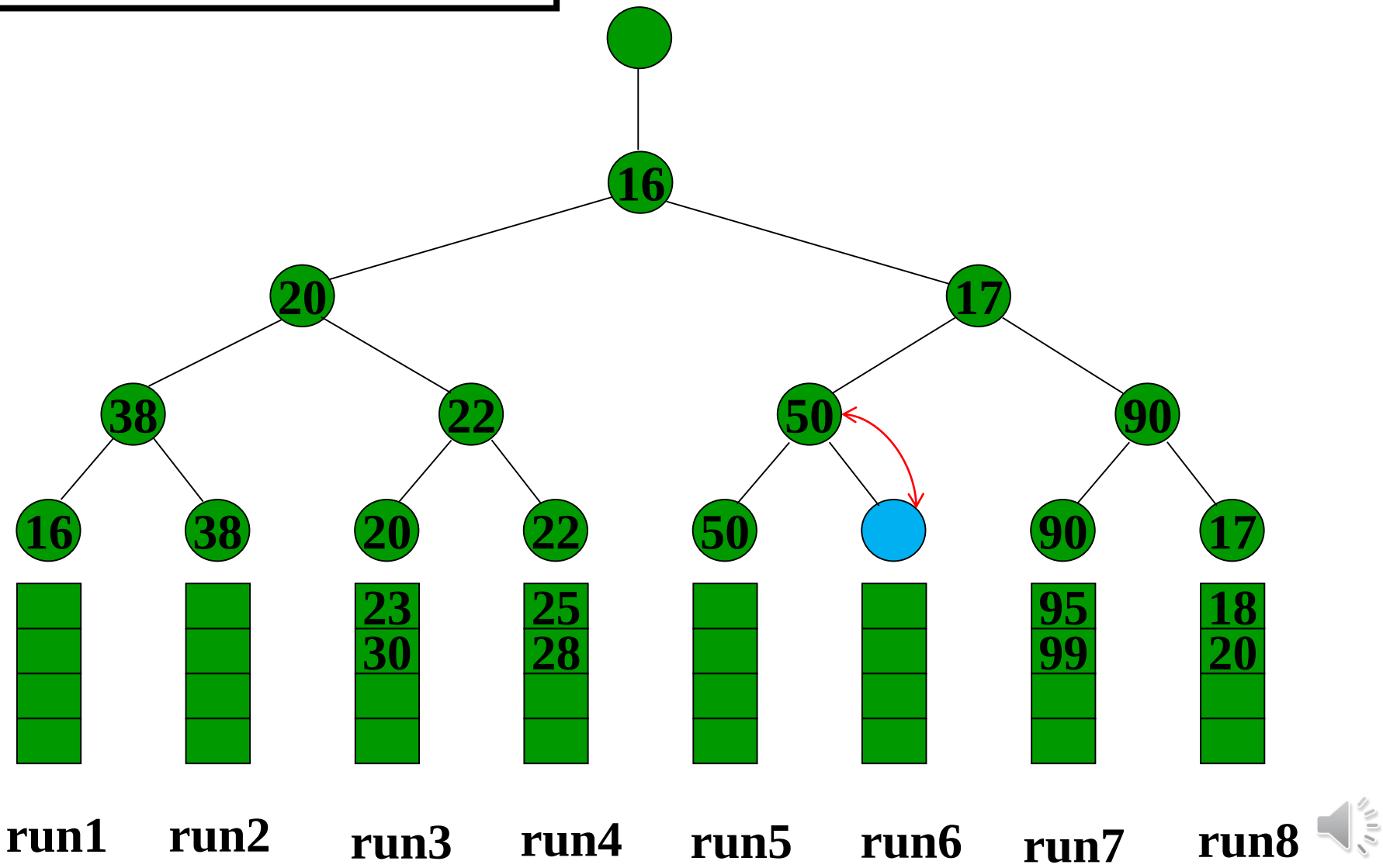
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



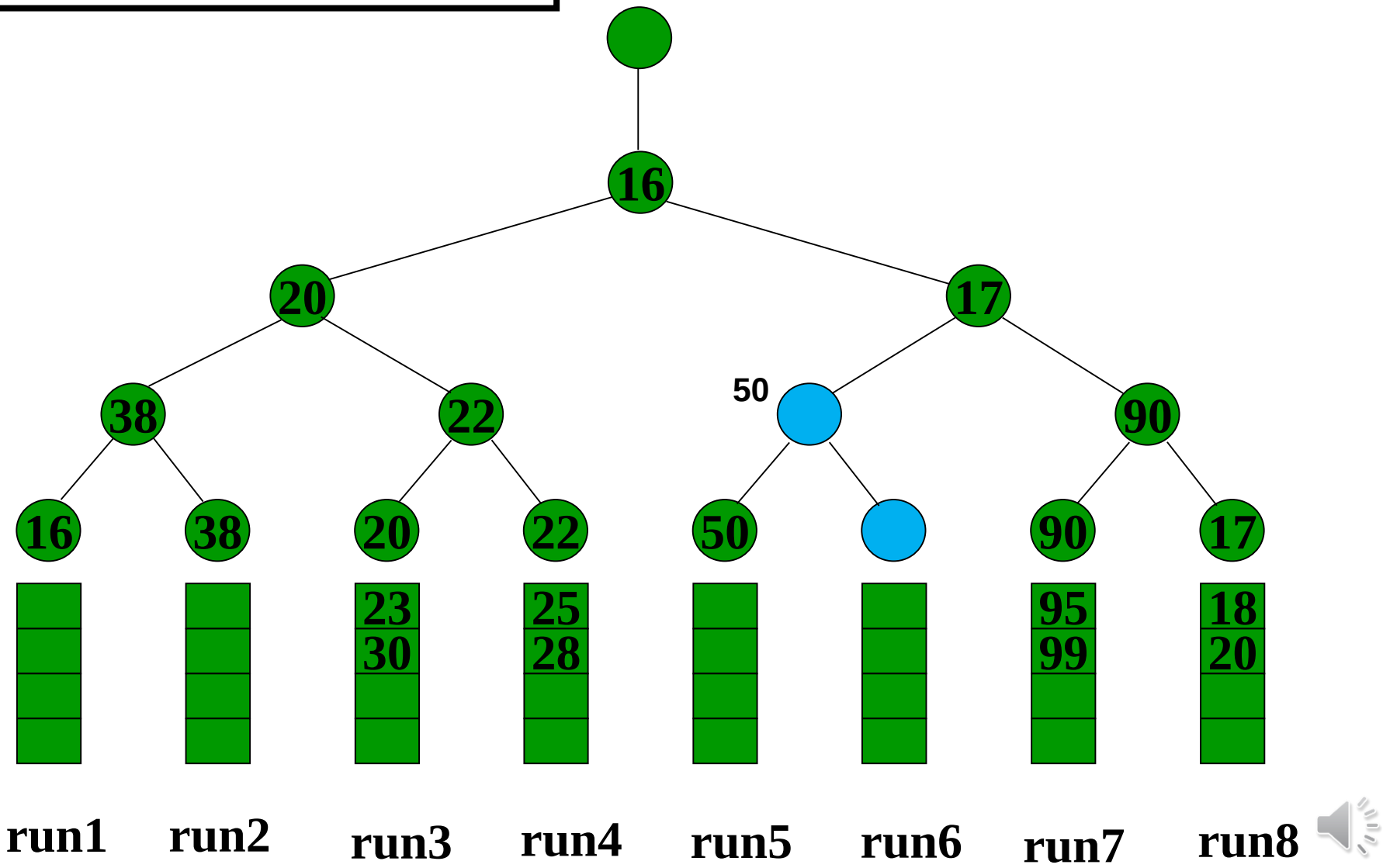
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



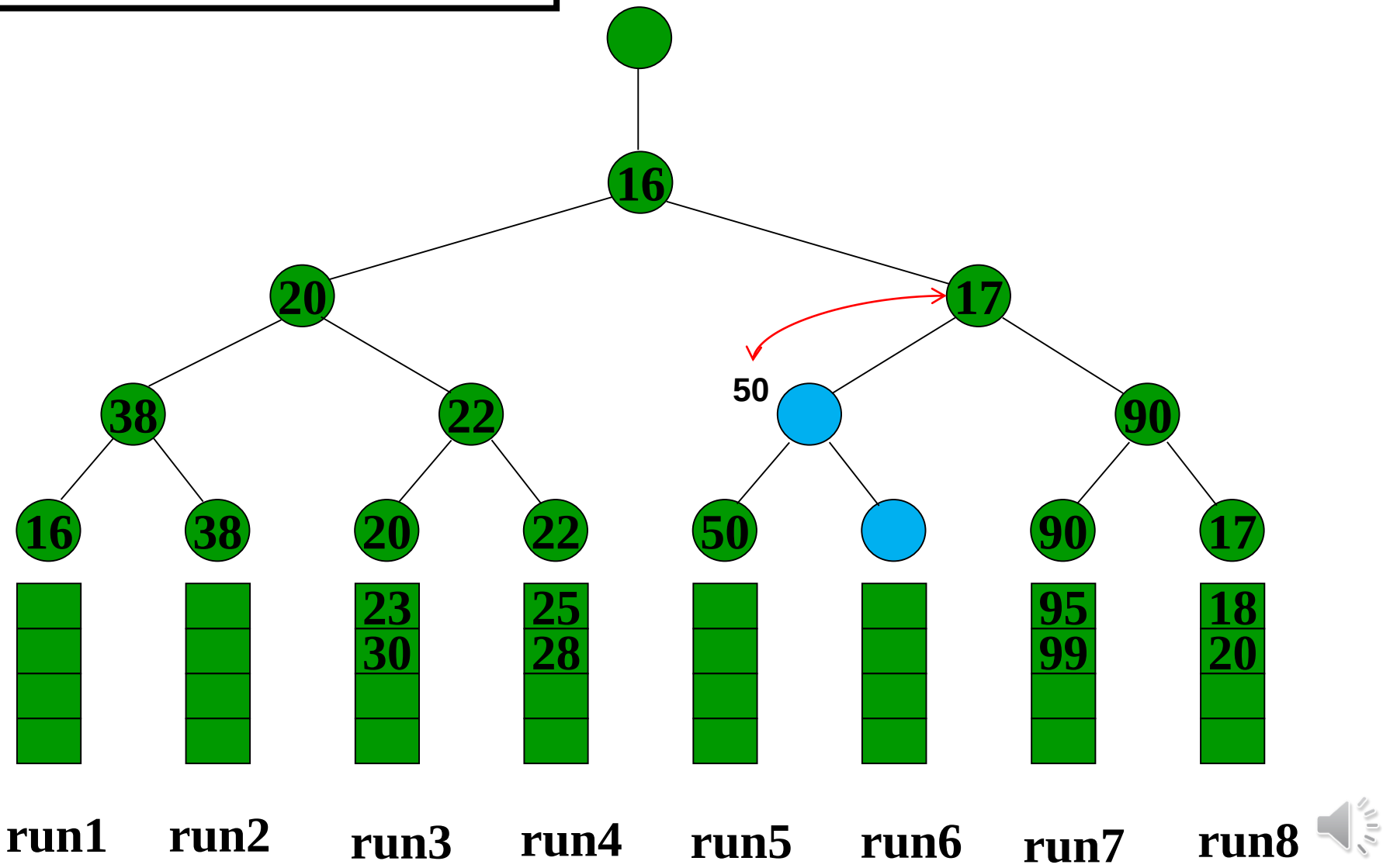
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



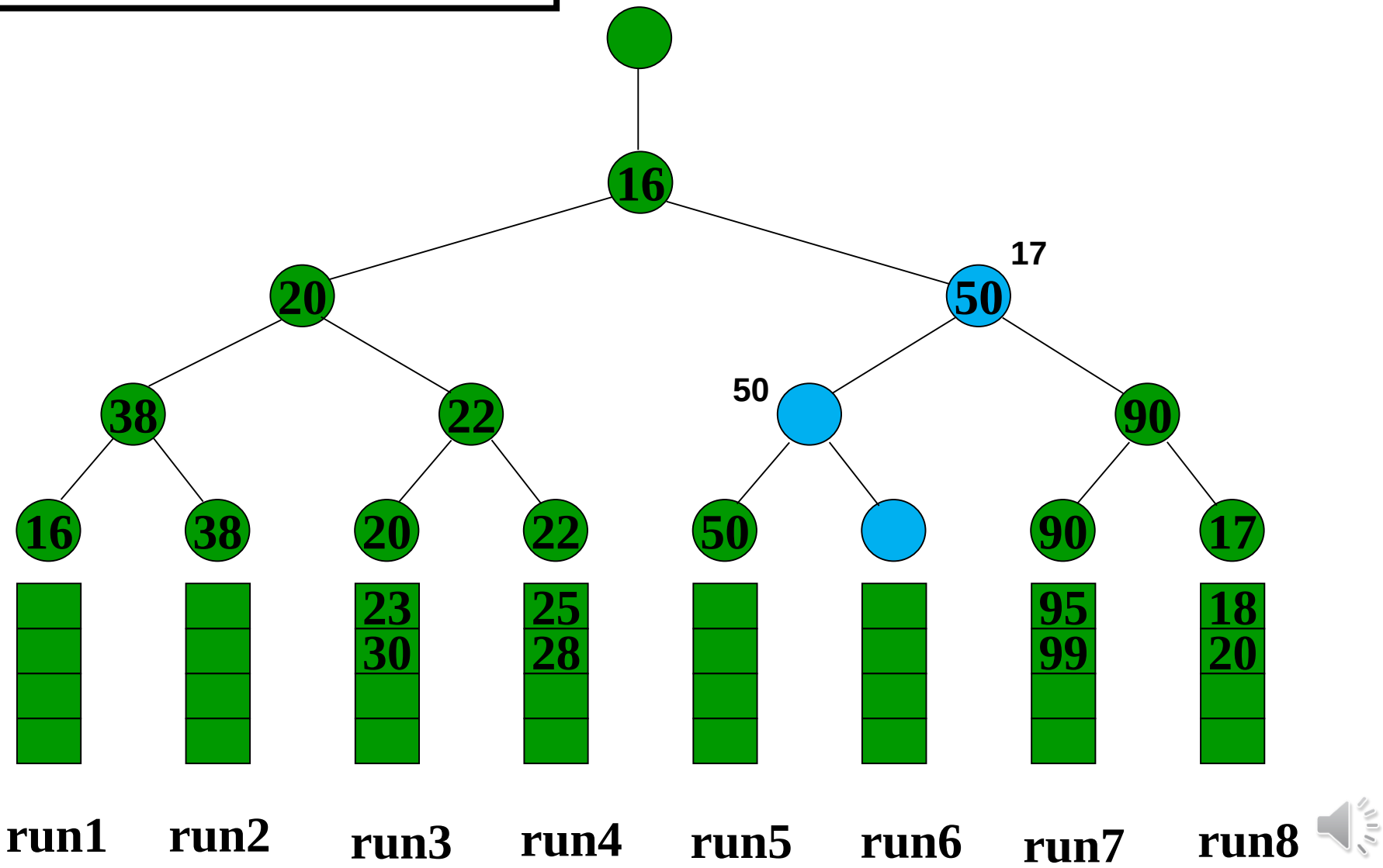
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



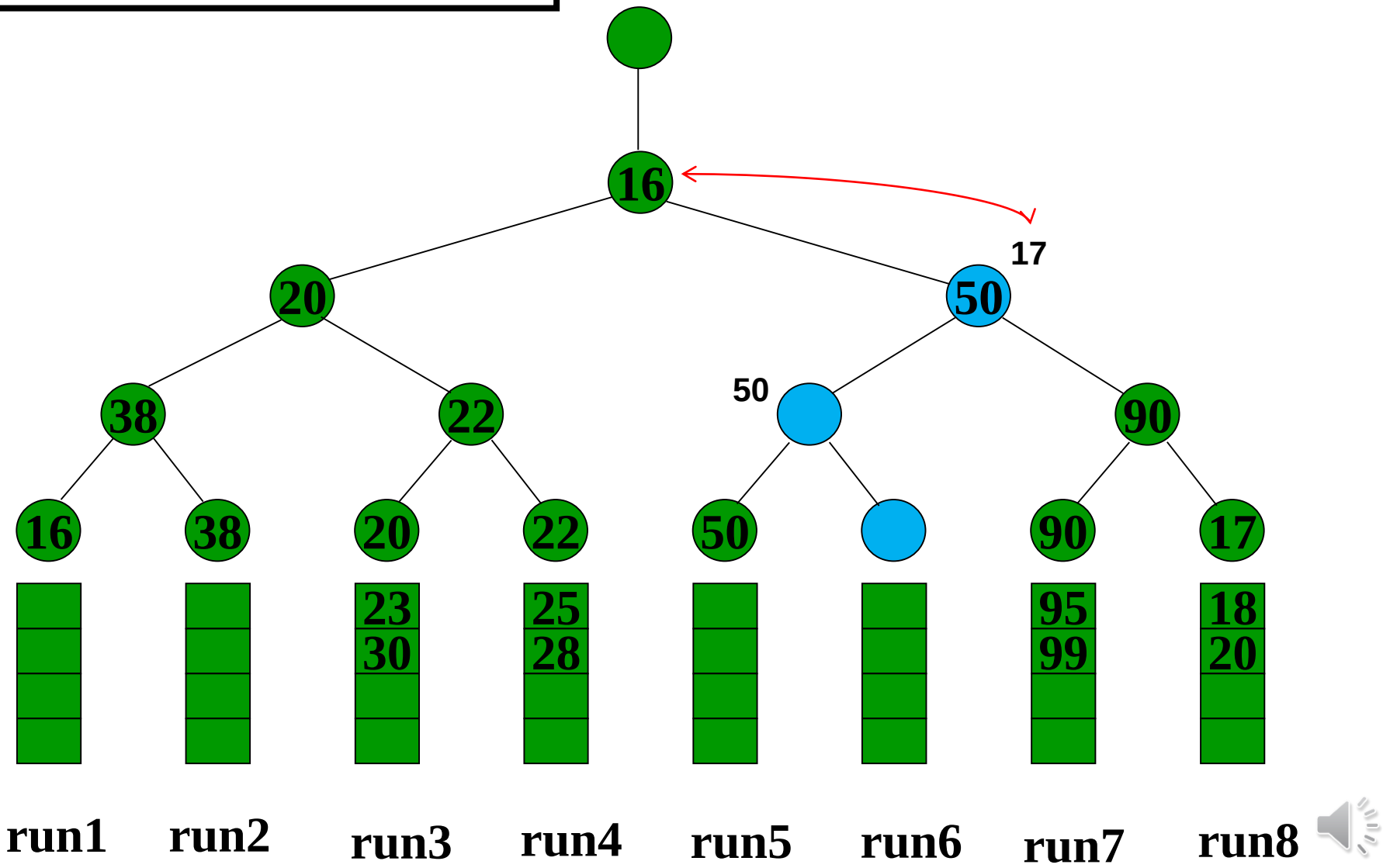
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



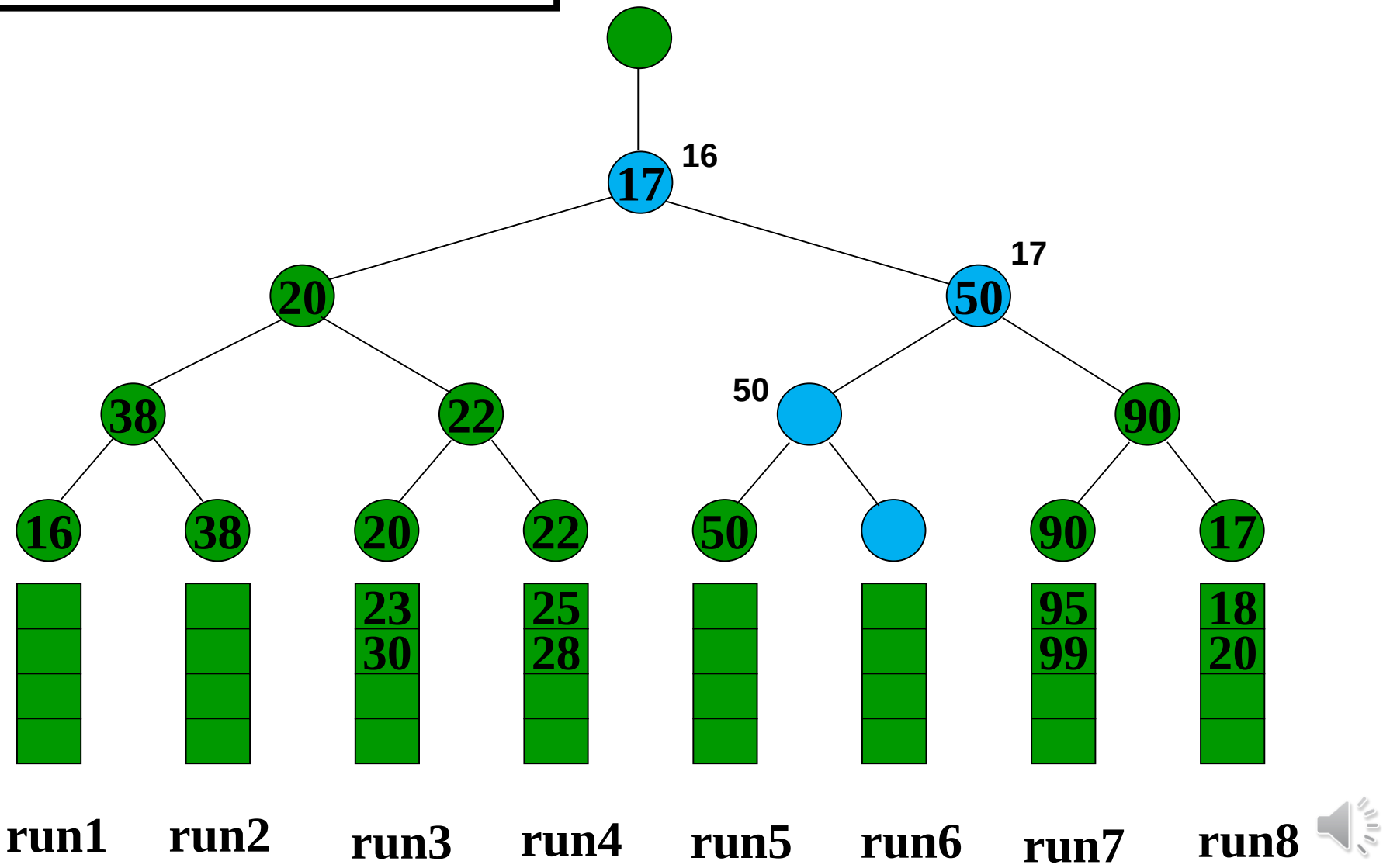
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



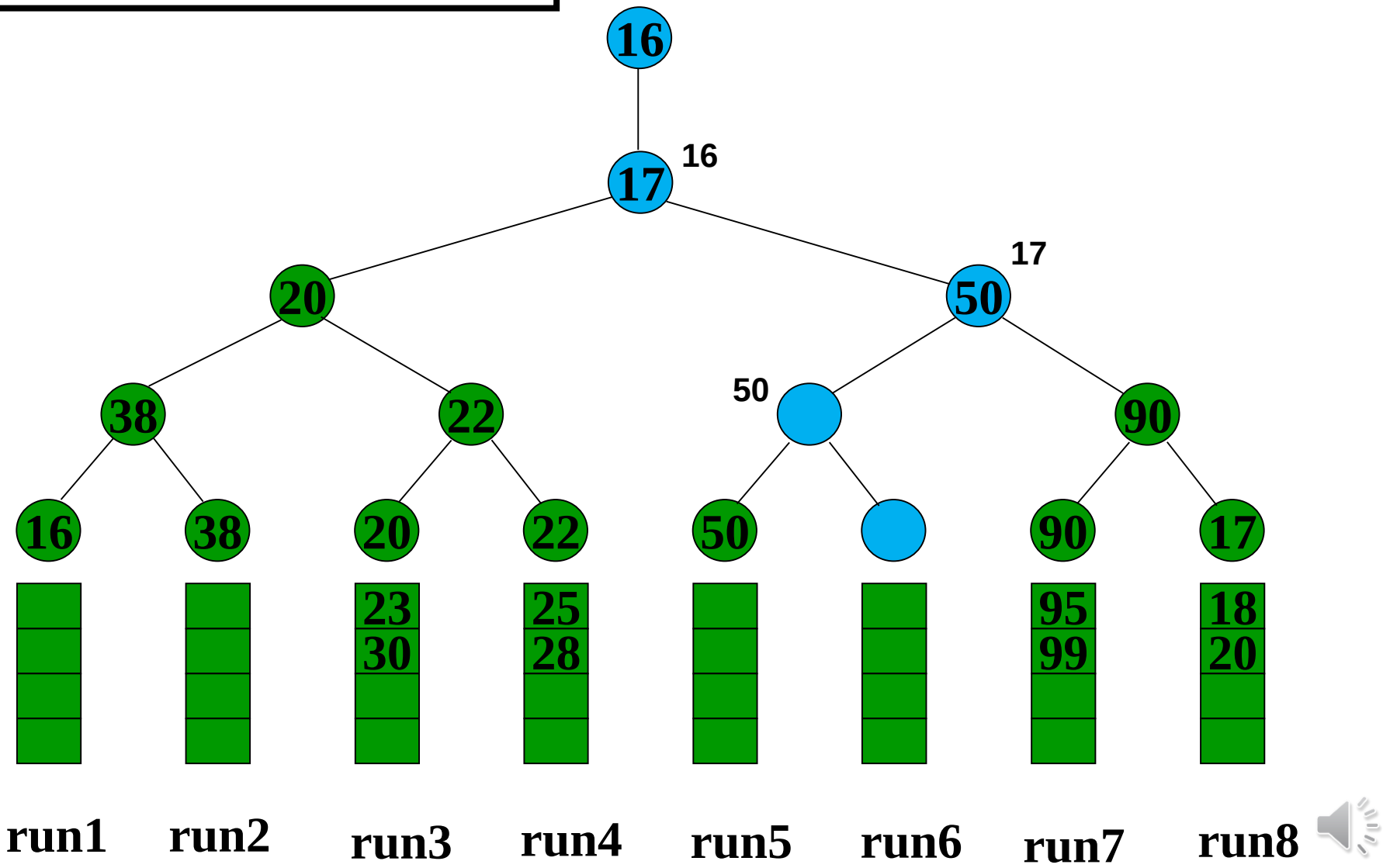
Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16
```



Loser Tree

```
Run:6 8 9 9 10 11 12 14
15 16 16
```



Questions?

감사합니다.