CSC 226

Algorithms and Data Structures: II

Rich Little
rlittle@uvic.ca

Kruskal's Algorithm Correctness

- Initialize each vertex in it's own tree

Cut property

- Merge two trees by finding lightest edge not yet in a tree and merging two trees together; repeat until all vertices are in one tree

Cycle property

- If the lightest edge connects two vertices in one tree already; discard that edge

- *Example of greedy algorithm*

Pseudocode: Kruskal's Algorithm

Algorithm KruskalMST(G):
Input: A weighted connected graph G with n vertices and m edges
Output: an MST T for G
Data structures: Disjoint set C; Priority Queue Q; and tree T

```
for each vertex v in G do
    C(v) ← {v}
Let Q be a min priority queue storing all edges in G by weight
T ← ∅          // initialize  tree T
while T has less than n − 1 edges do
    (u, v) ← Q.removeMin()
    Let v ∈ C(v)
    Let u ∈ C(u)
    if C(v) ≠ C(u) then
        Add edge (u, v) to T
        Union C(v) and C(u)
return T
```

C :disjoint set of vertices called clusters

Q :a priority queue for the edges according to edge weights

Implementing Kruskal's Algorithm

- Two new ideas arise out of Kruskal's algorithm here
  - Efficient heap construction – bottom-up heap
  - Efficient cycle detection – union-find data structure

Idea 1: Bottom-up Heap

- Avoid sorting the edge weights by storing the edges in a heap …..

Building up a heap

- m standard insert-operations for a heap result in $\mathrm{O}(m \log(m))$ time.

- Can we build up a heap for m given elements faster? Is $\mathrm{O}(m)$ possible?

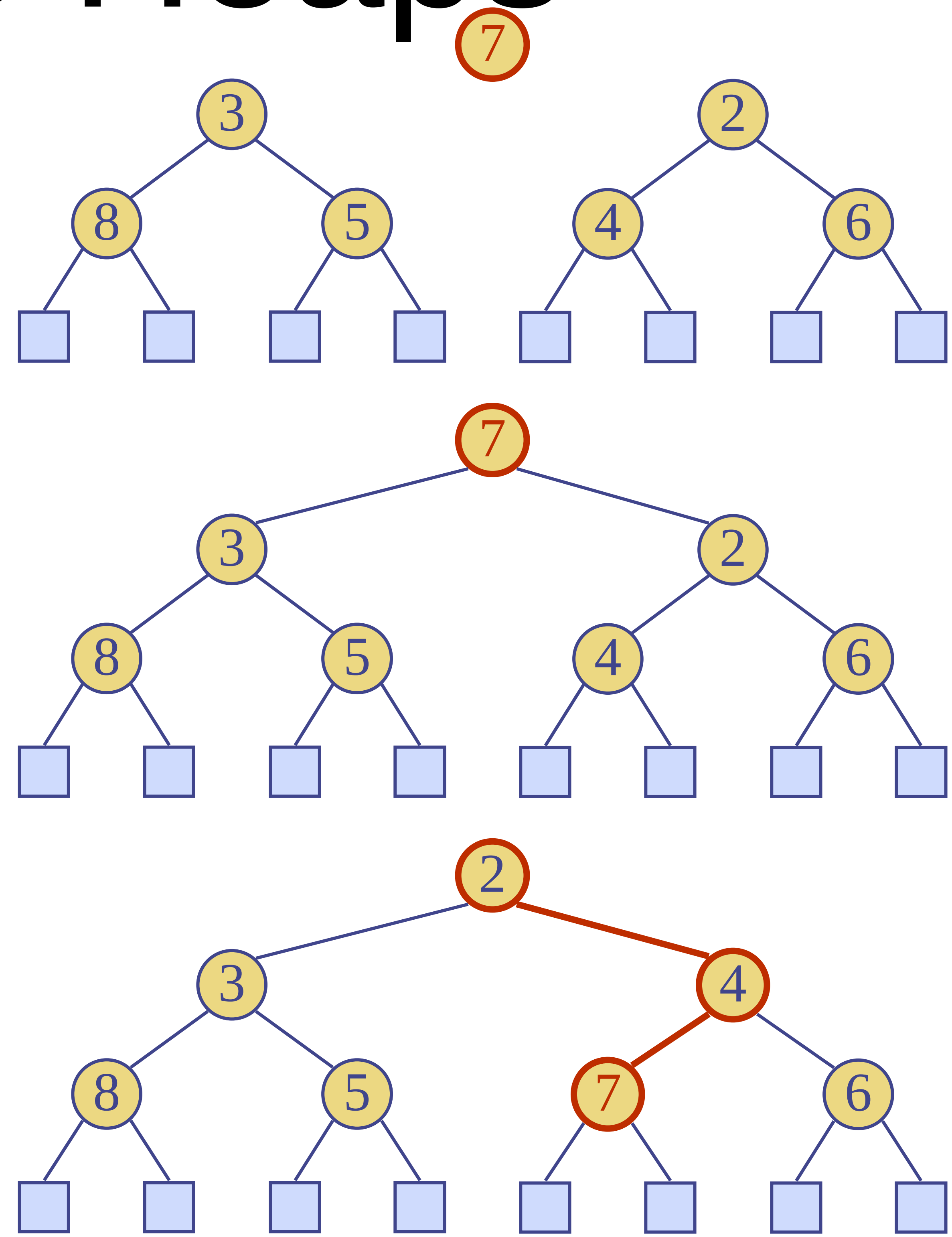
Bottom-Up Heap (pgs. 176-178)

```
Algorithm BottomUpHeap(S):
   Input: A list S storing m keys
   Output: A heap T storing the m keys

   if S is empty then
      return external node
   remove the first key, k, from S
   split S in half, lists S1 and S2
   T1 ← BottomUpHeap(S1)
   T2 ← BottomUpHeap(S2)
   T ← merge(k,T1,T2)
   DownHeap(T,root)
   return T
```

Merging Two Heaps

- We are given two heaps and a key *k*
- We create a new heap with the root node storing *k* and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



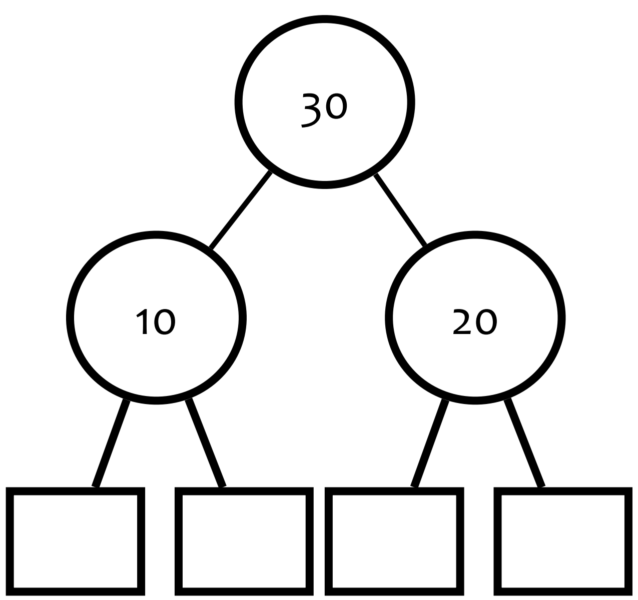
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

150 70 30 10 20 60 40 50 | 140 100 80 90 130 110 120

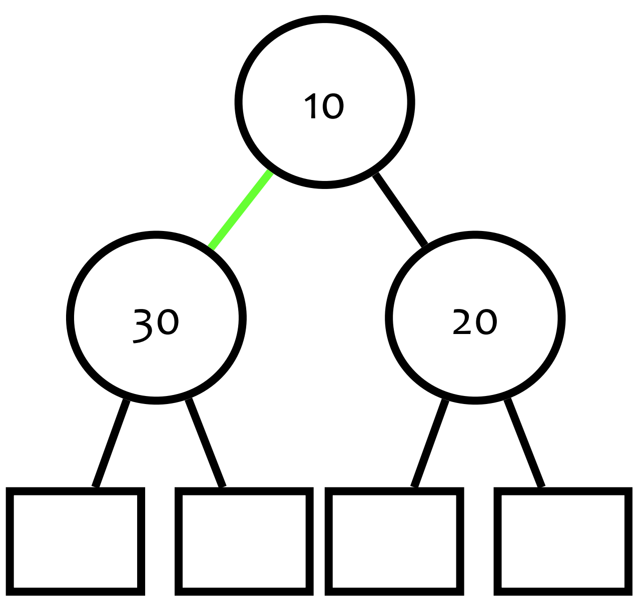
150 70 30 10 20 60 40 50 | 140 100 80 90 130 110 120

150 70 30 10 20 | 60 40 50 | 140 100 80 90 130 110 120

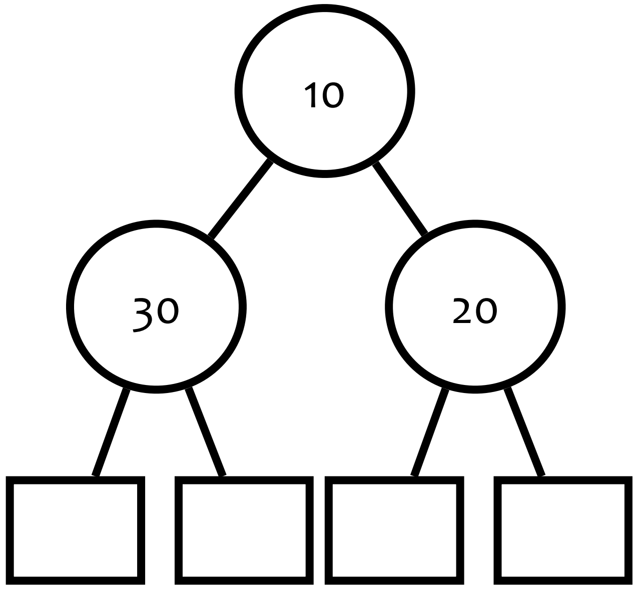
150 70 30 10 | 20 | 60 40 50 | 140 100 80 90 130 110 120



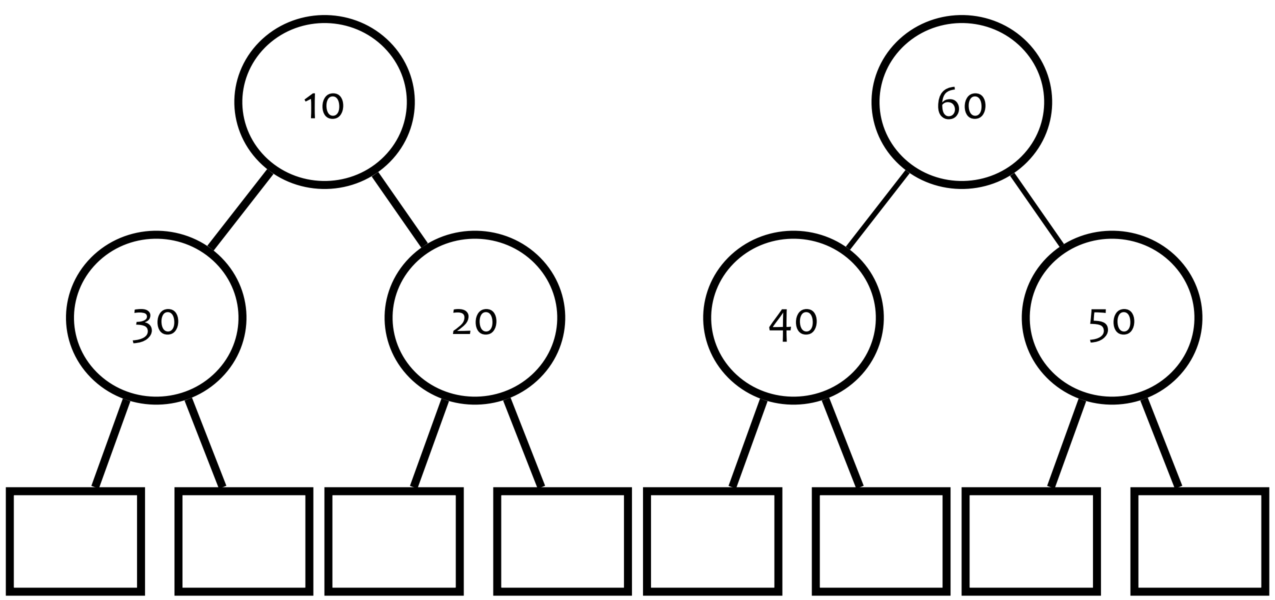
150 70 30 10 | 20 | 60 40 50 | 140 100 80 90 130 110 120



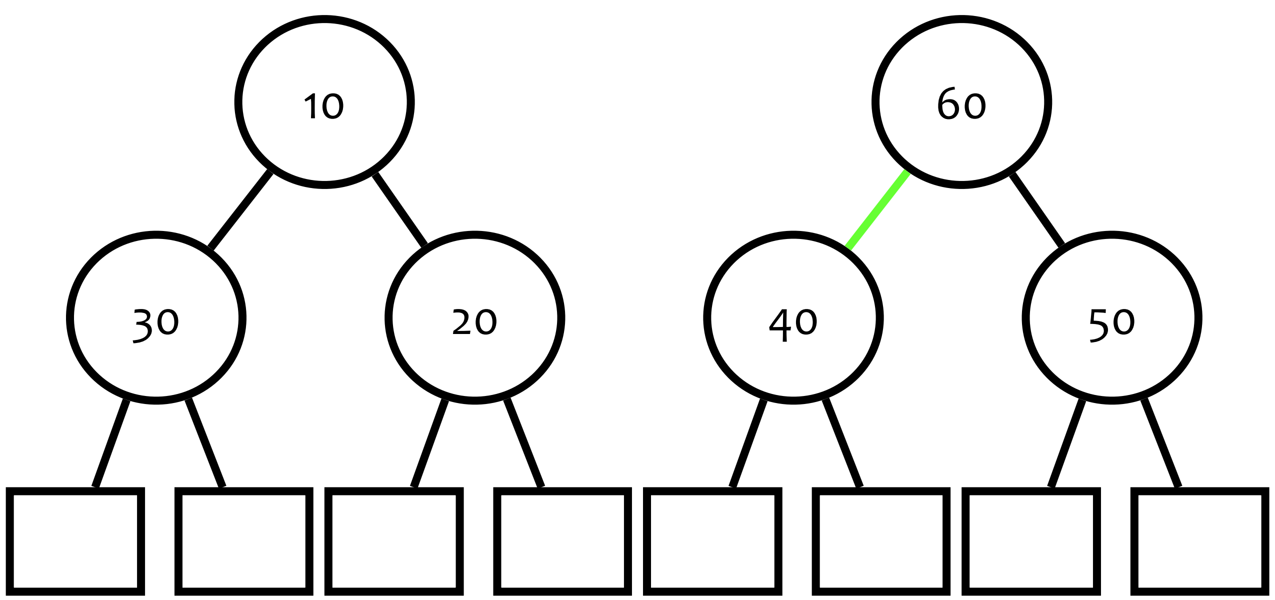
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



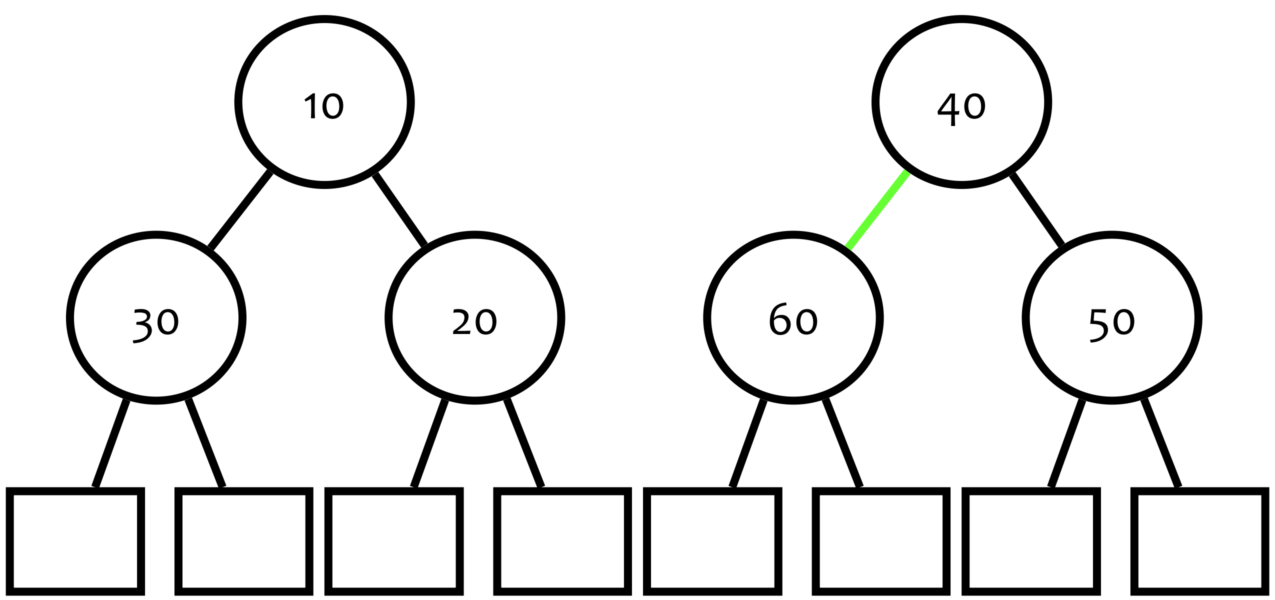
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



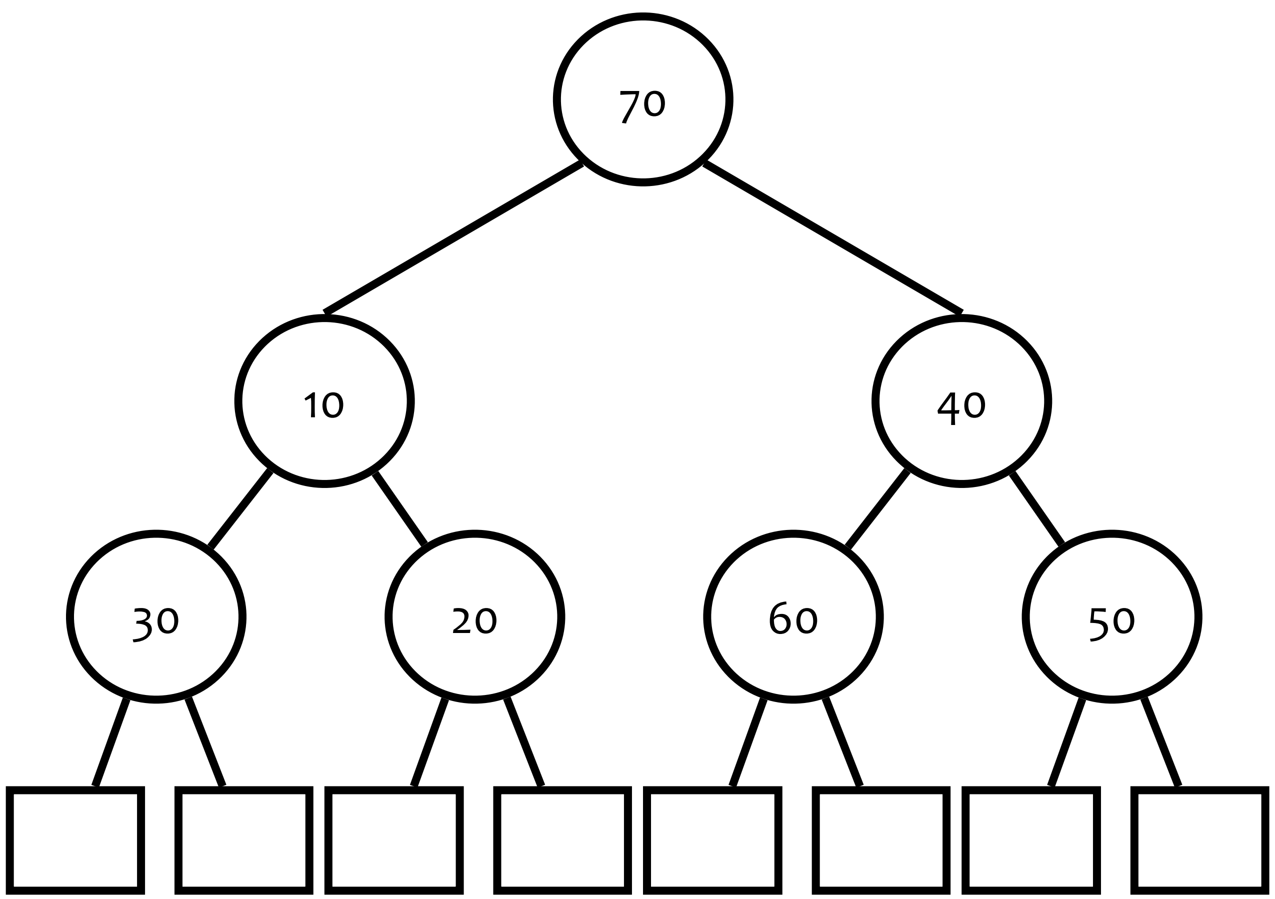
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



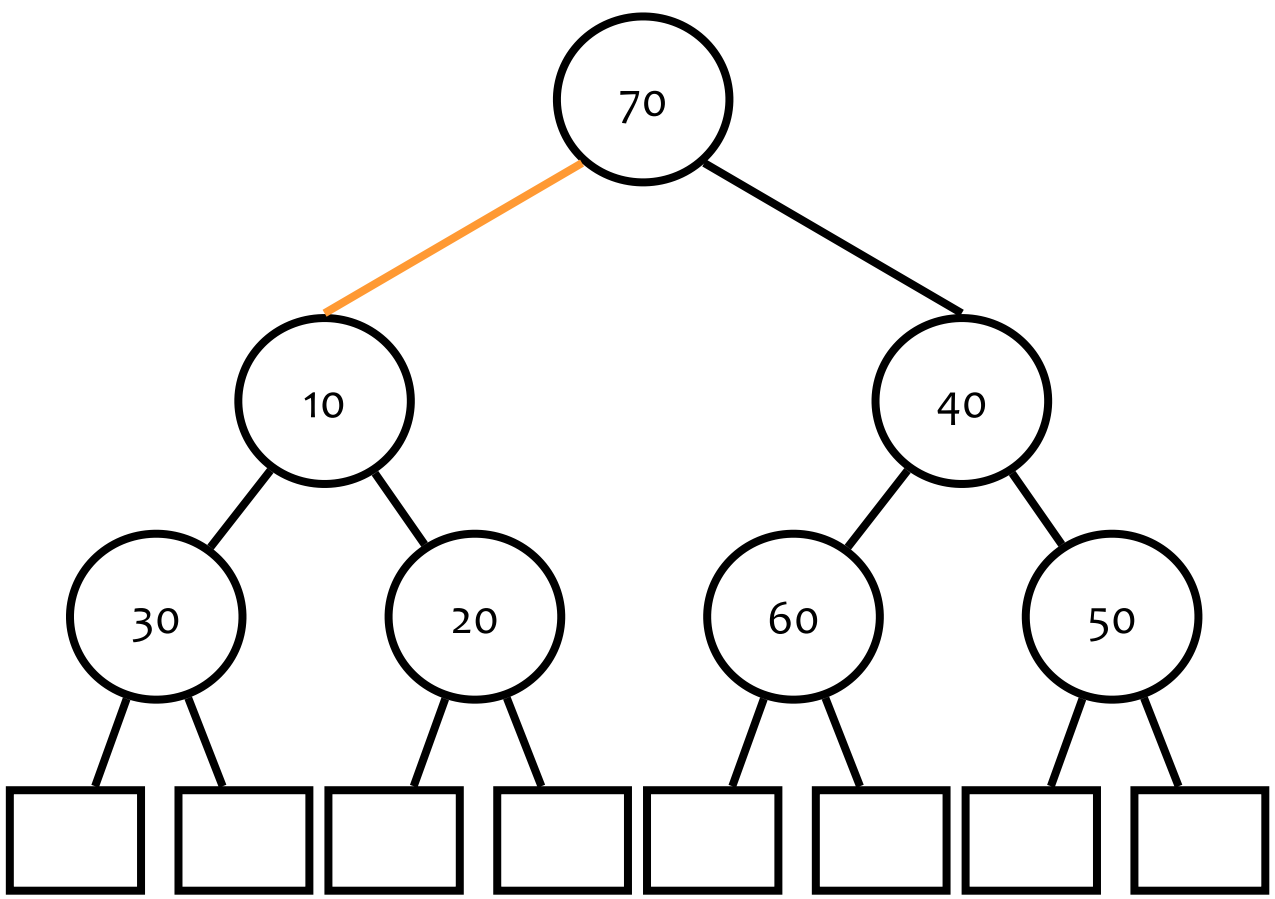
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



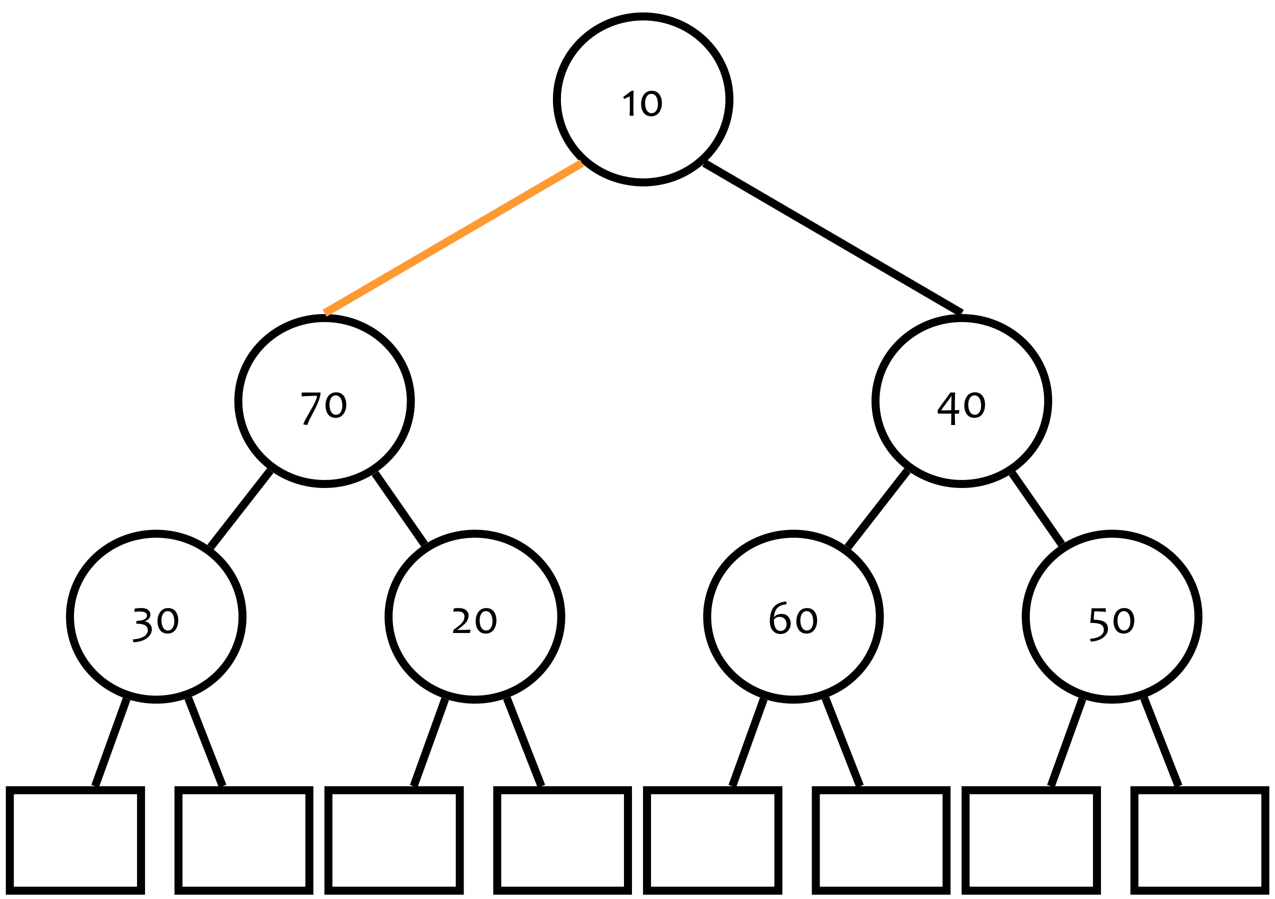
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



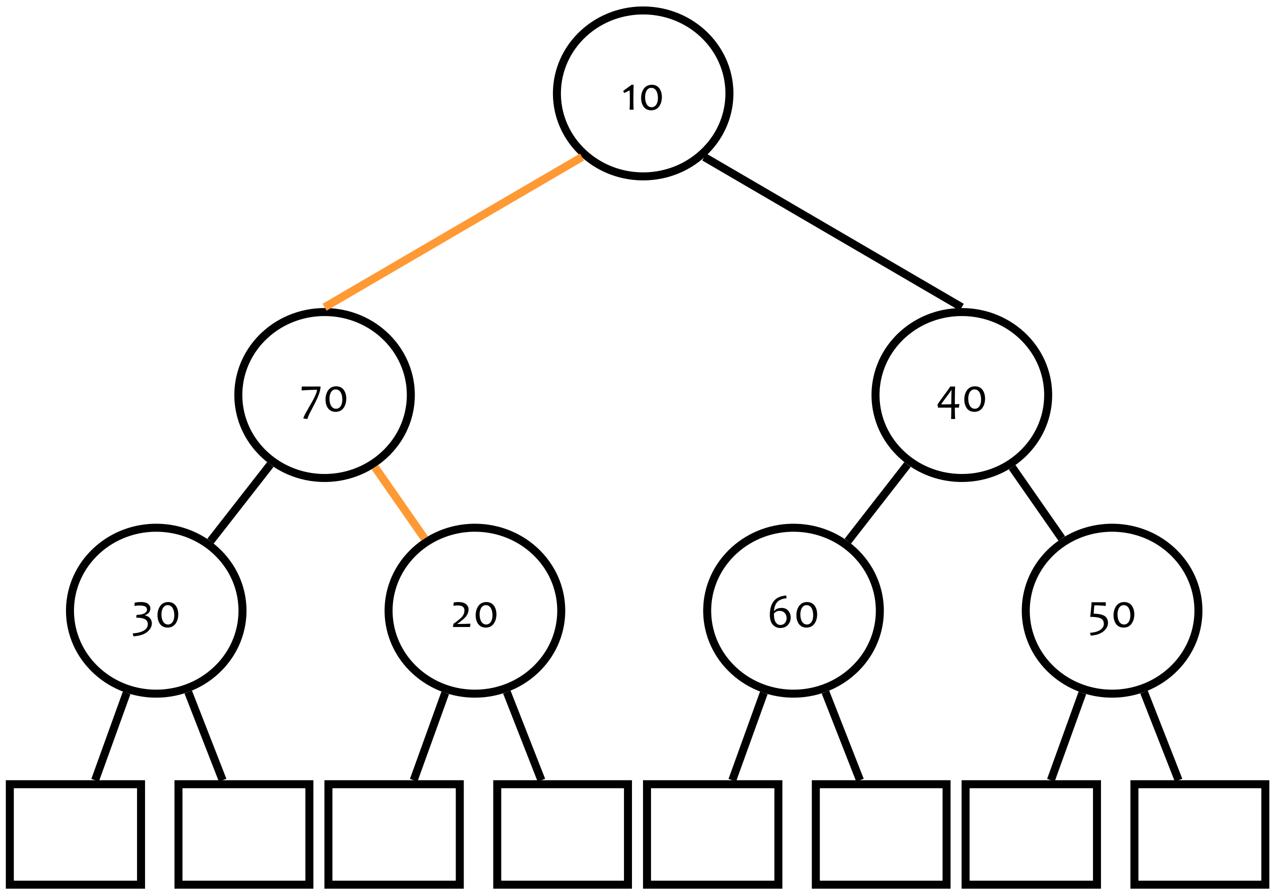
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



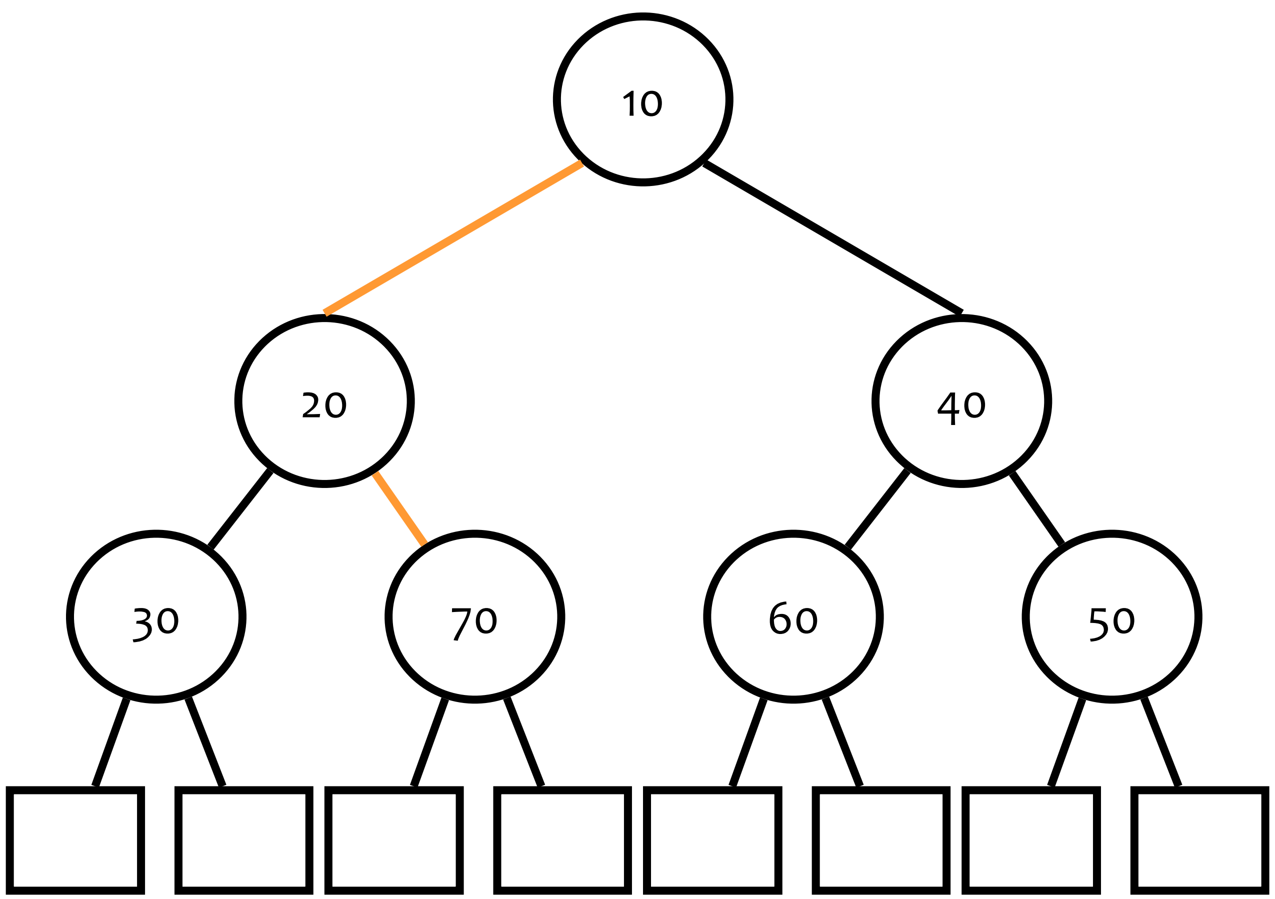
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



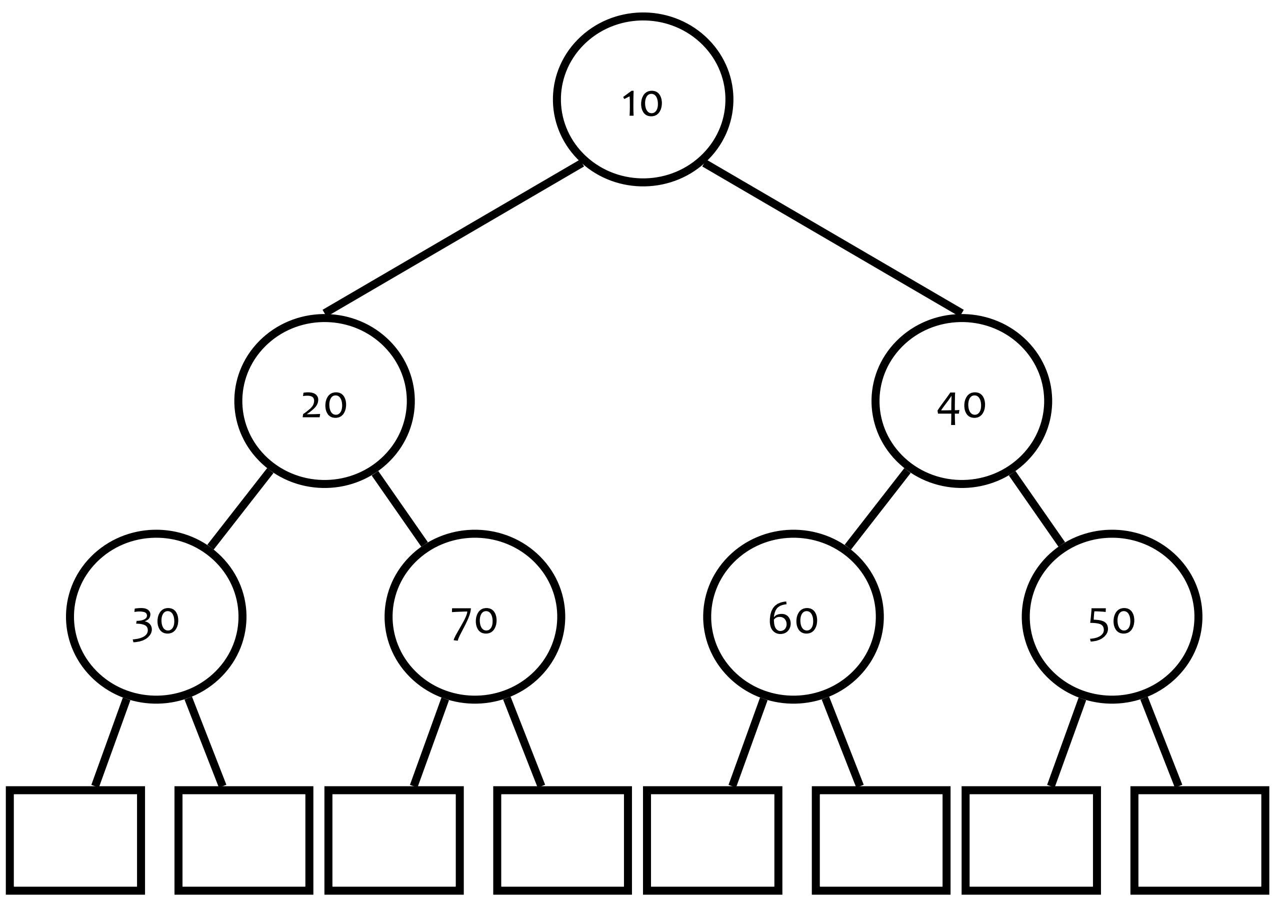
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



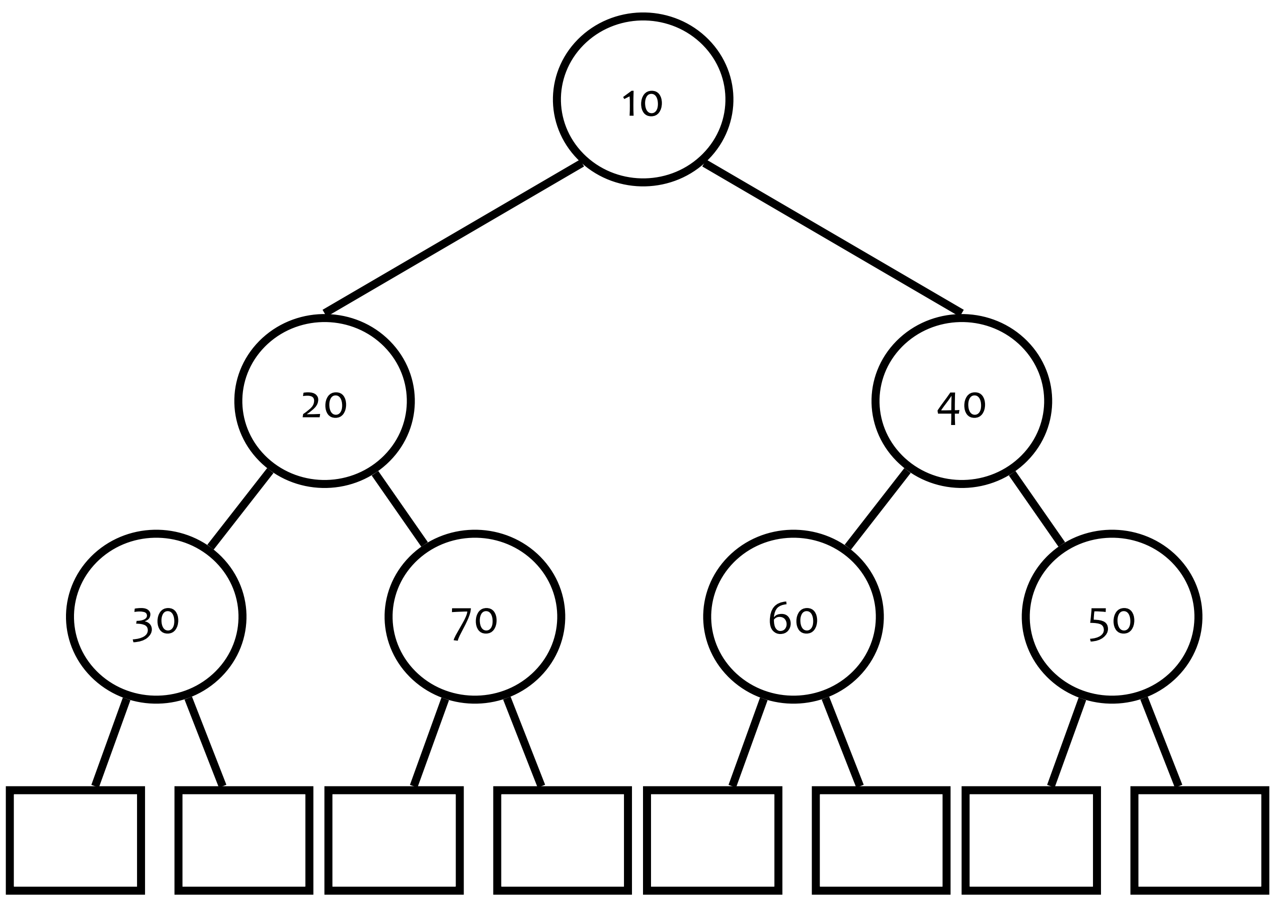
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



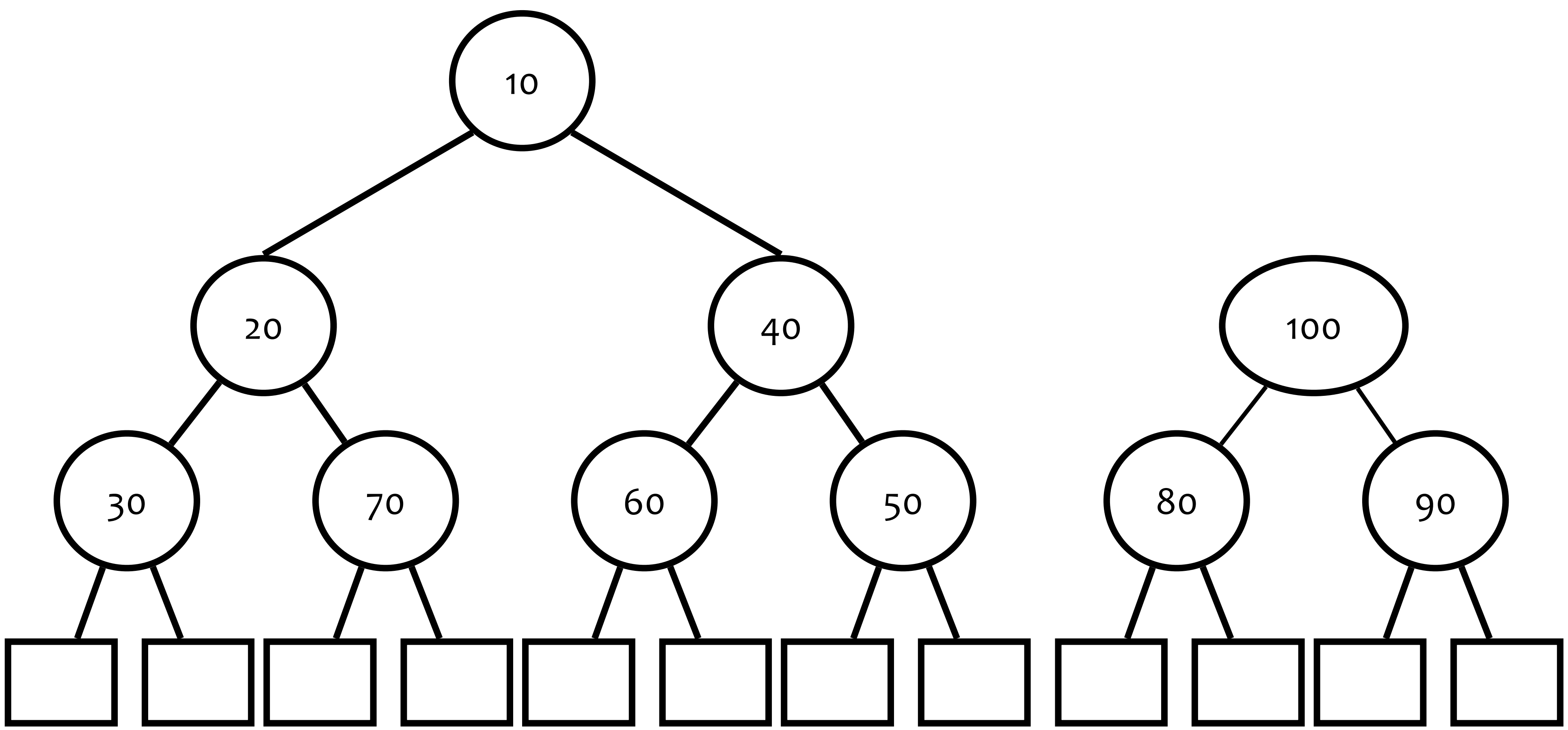
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



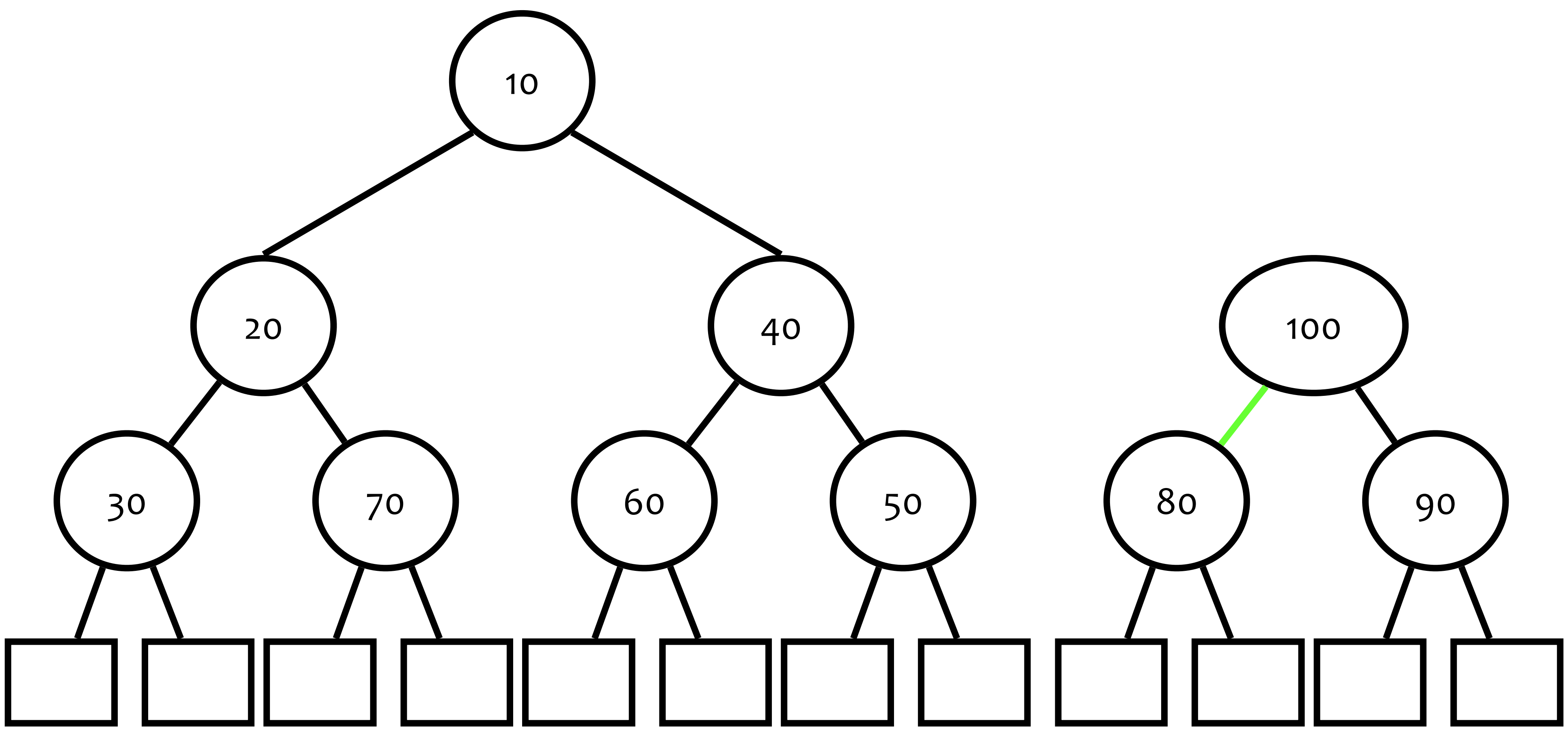
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



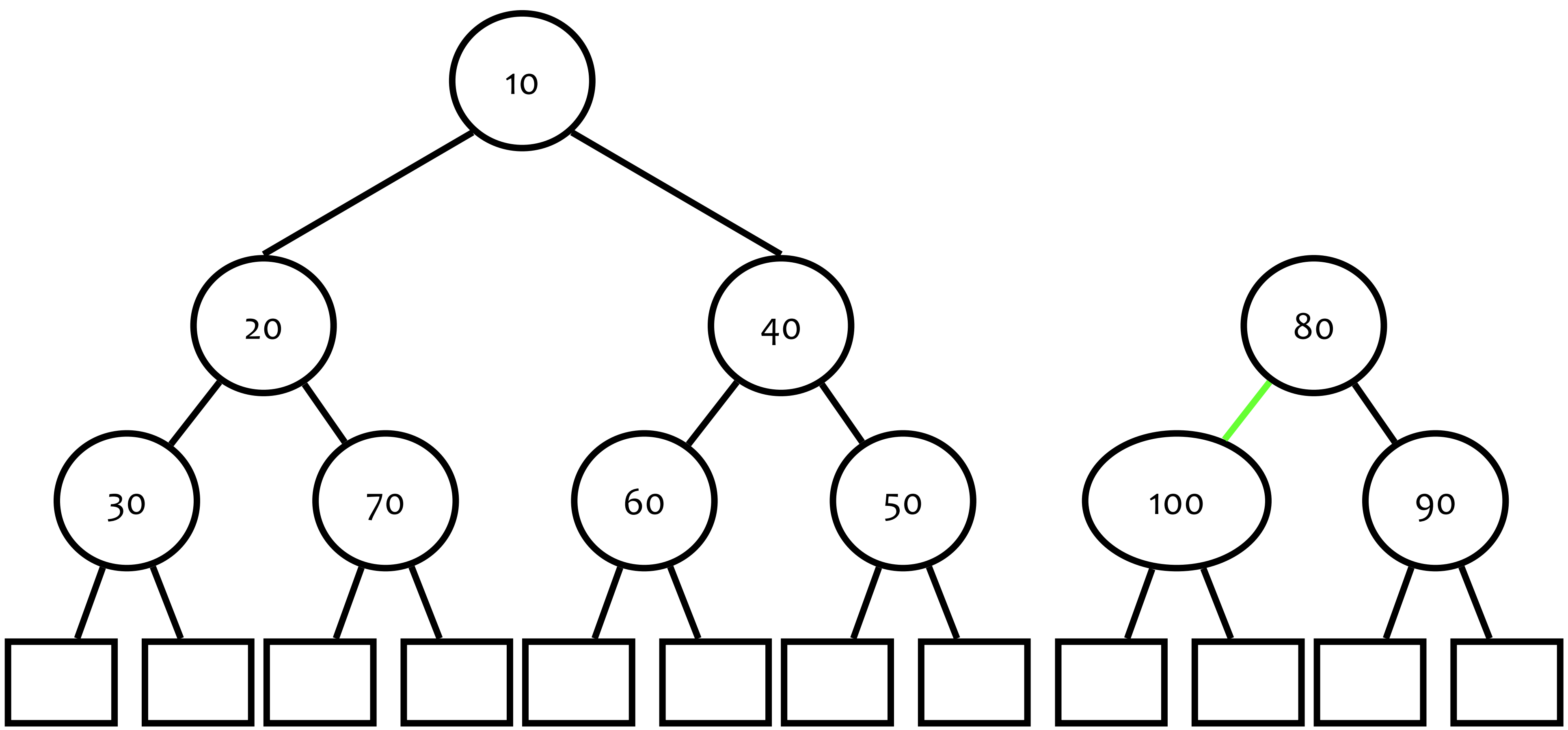
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



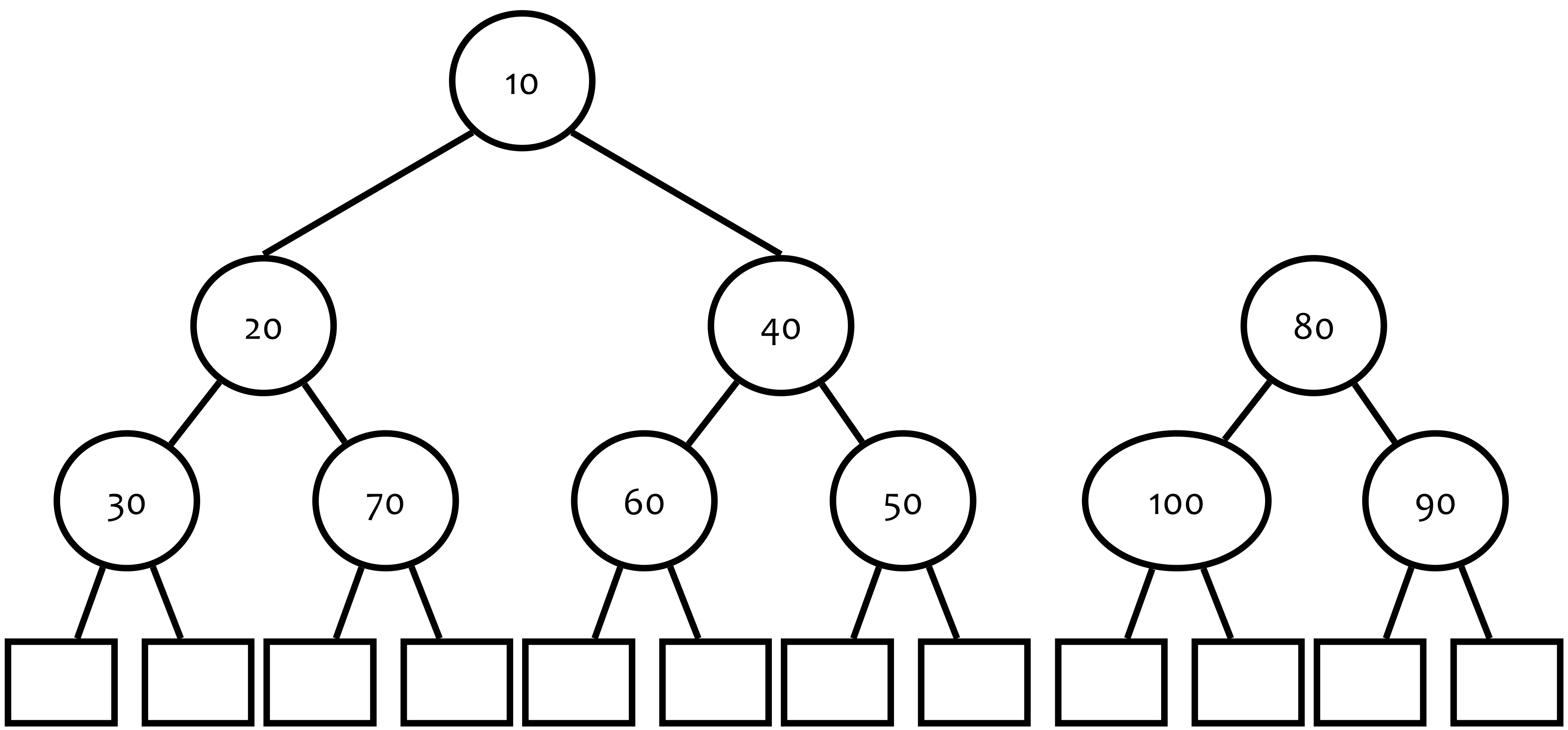
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



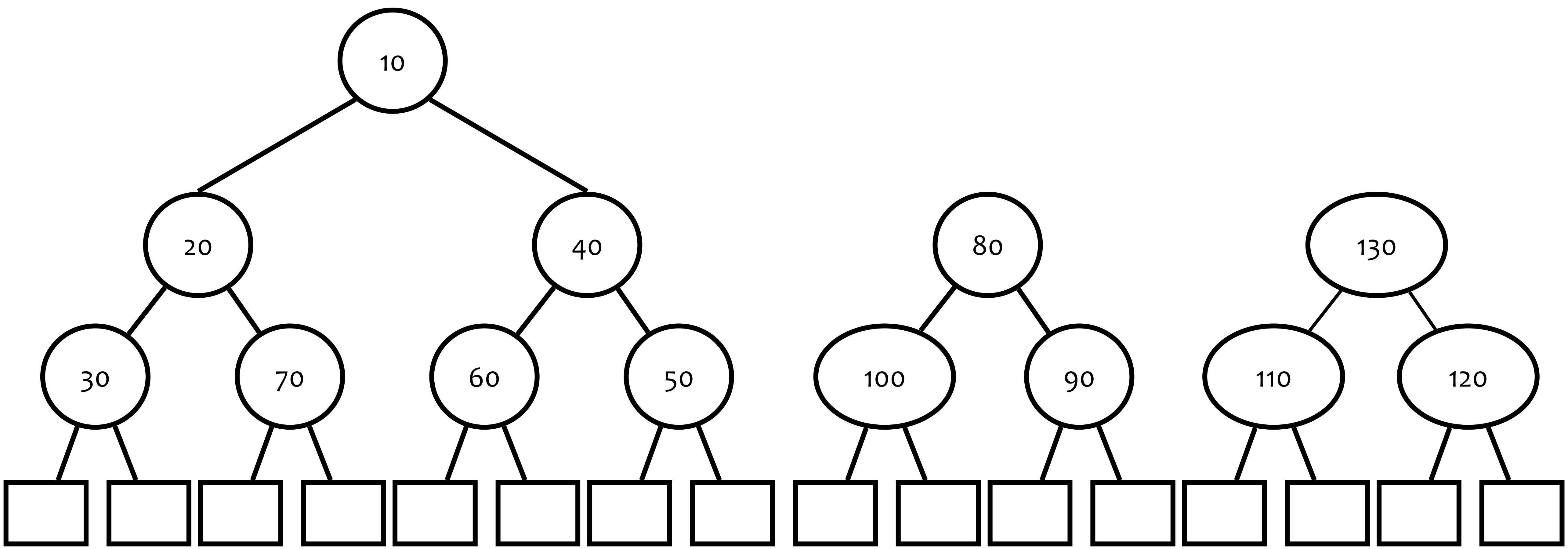
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



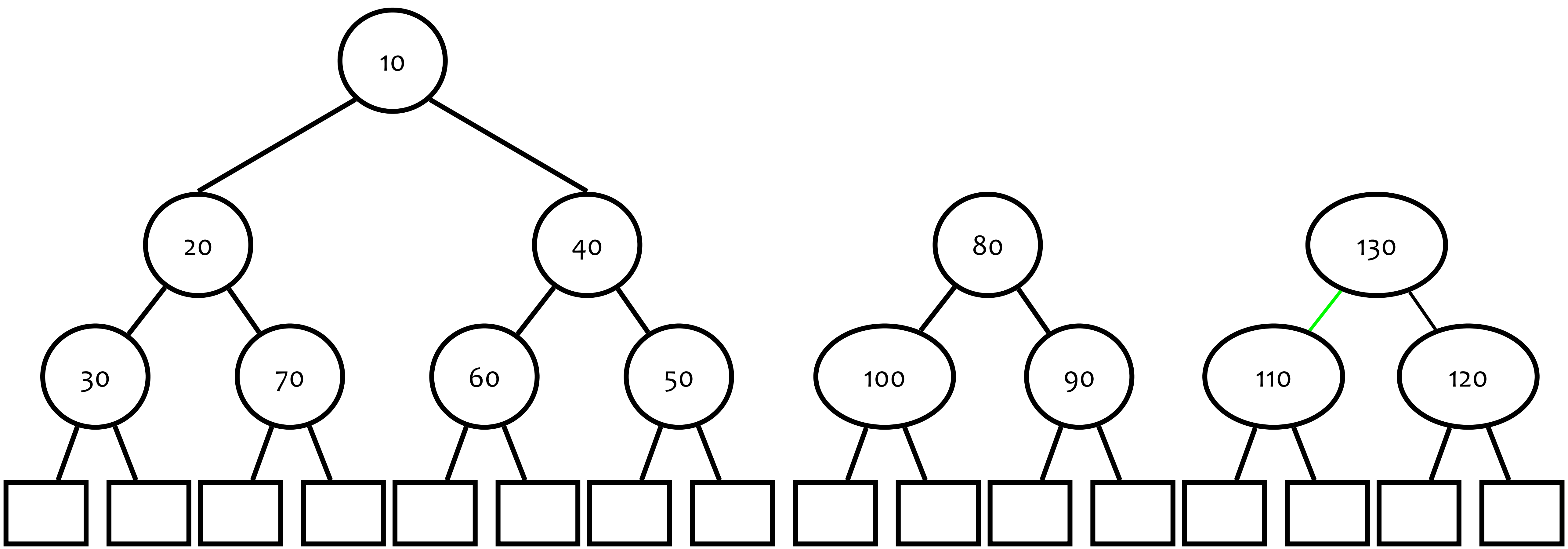
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



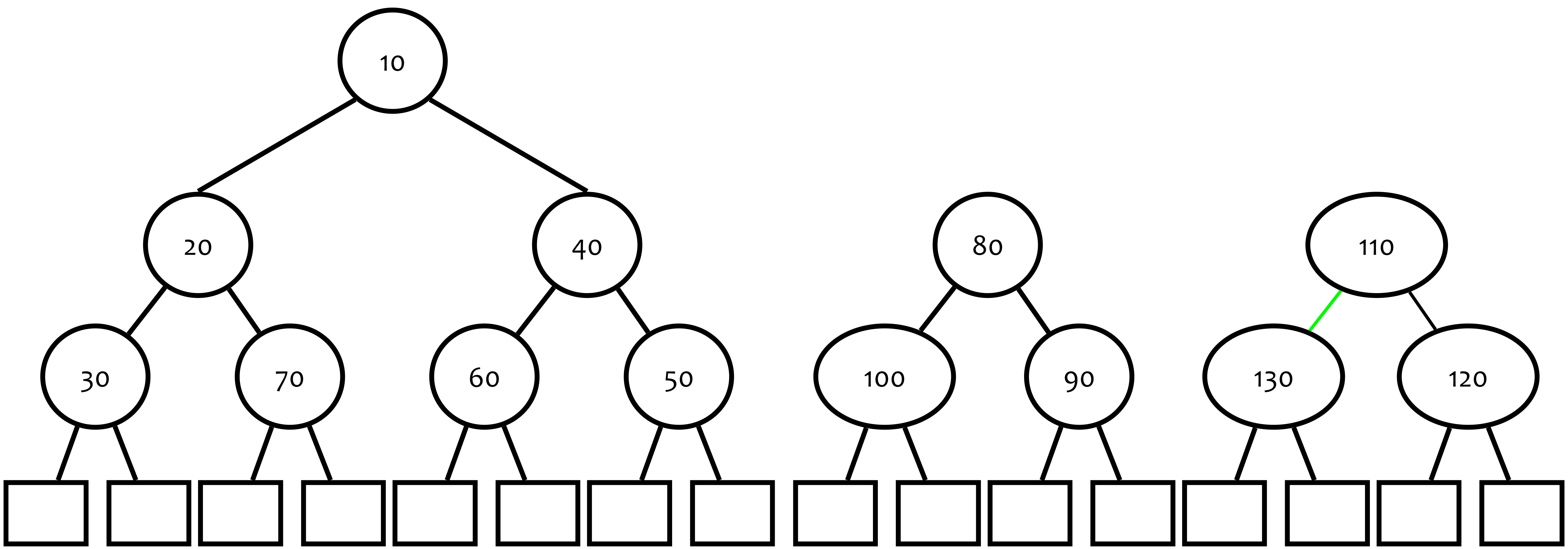
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

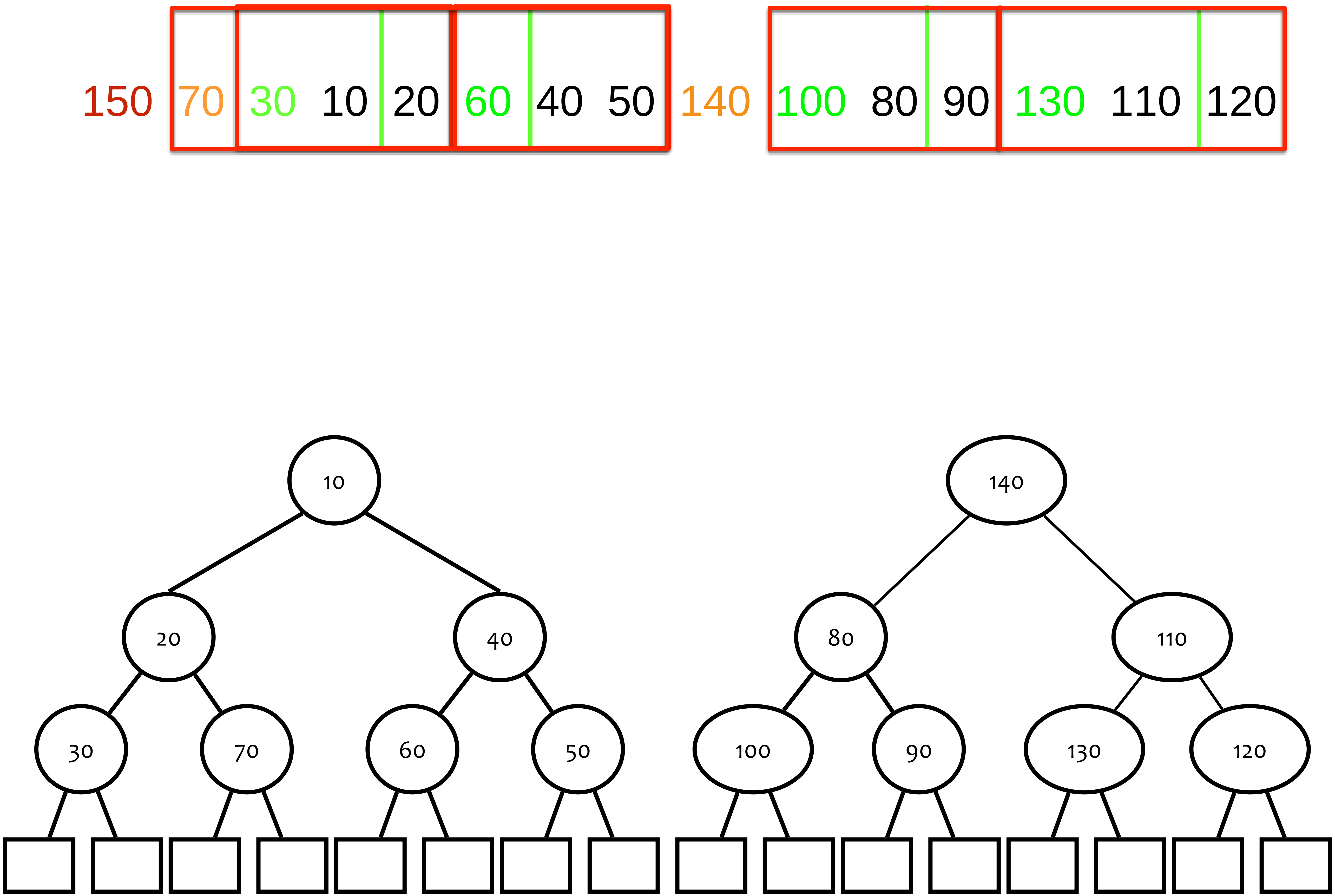


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

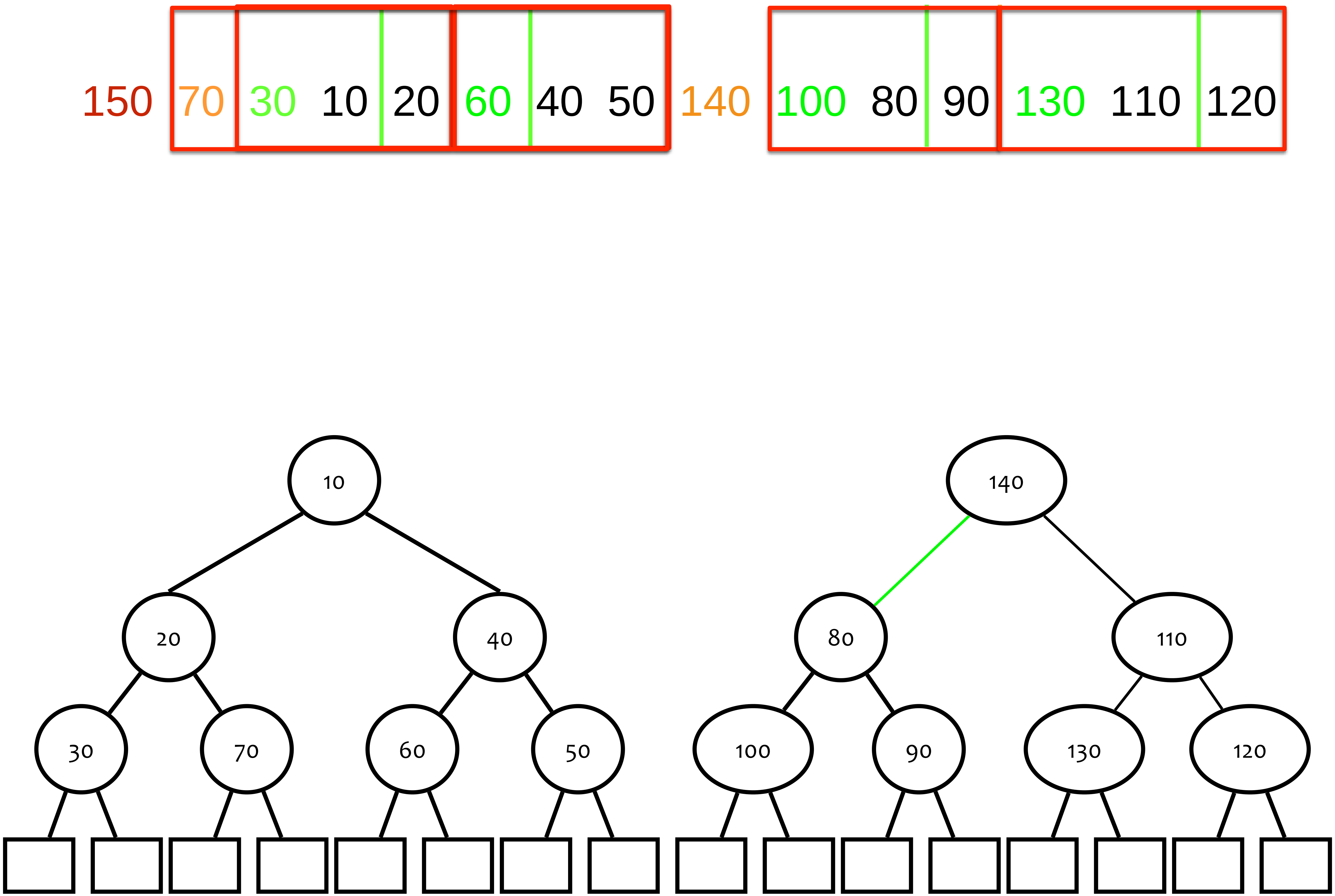


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

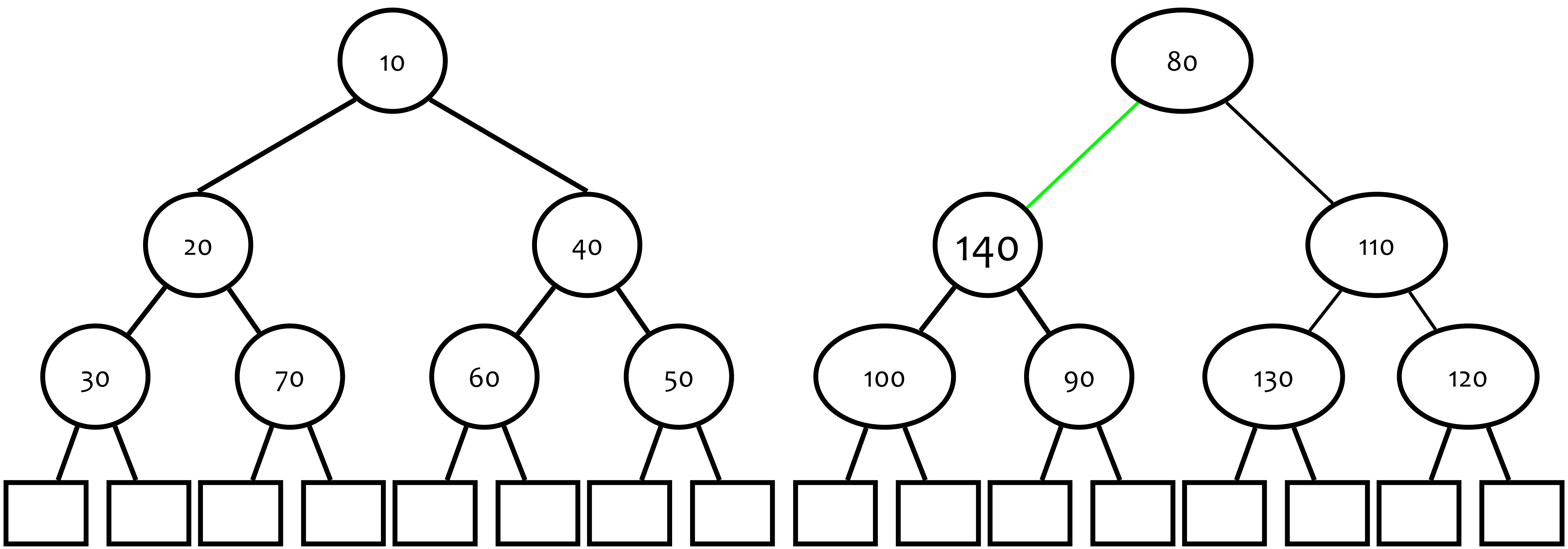


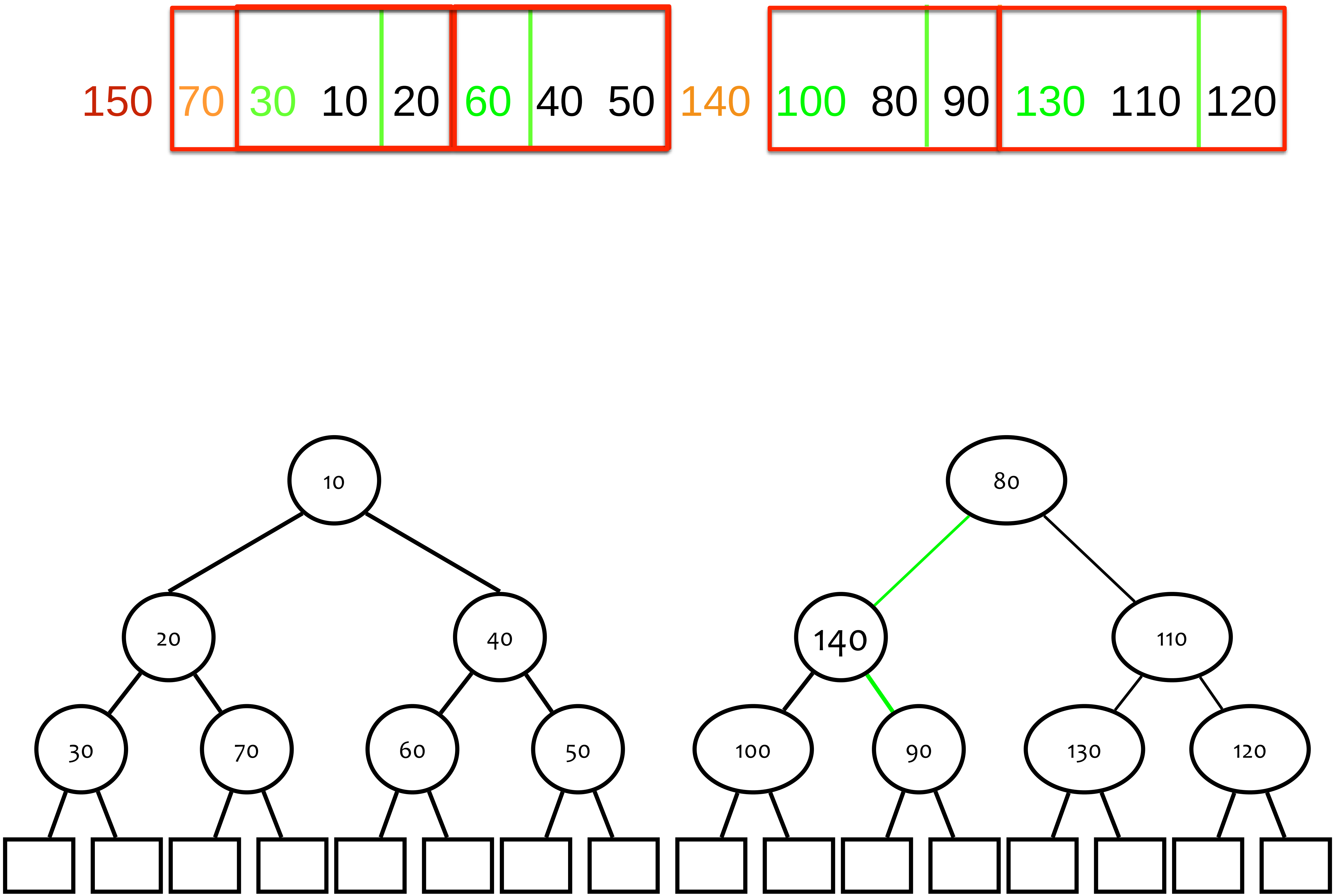


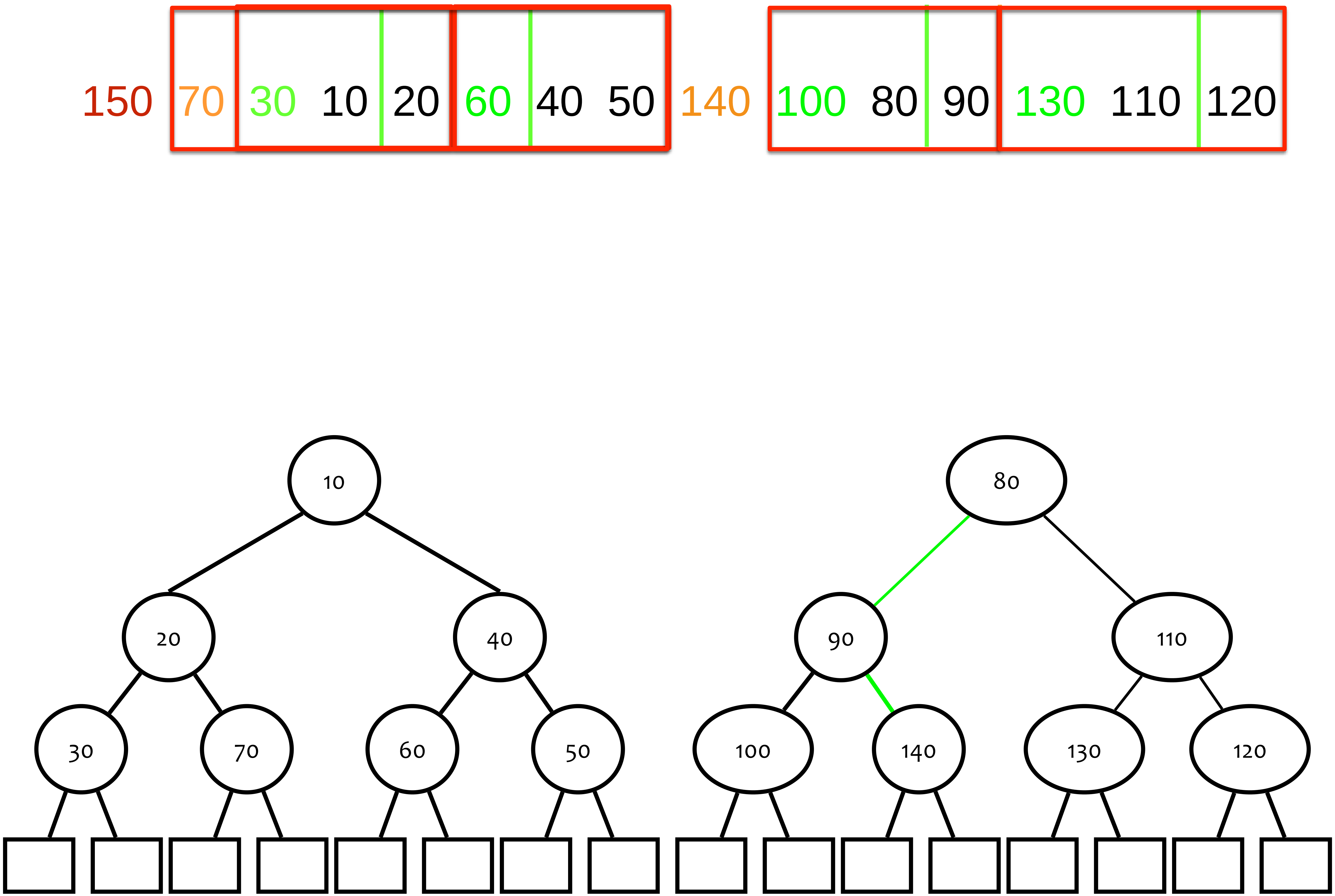
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
140
80
110
100
90
130
120


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
140
80
110
100
90
130
120

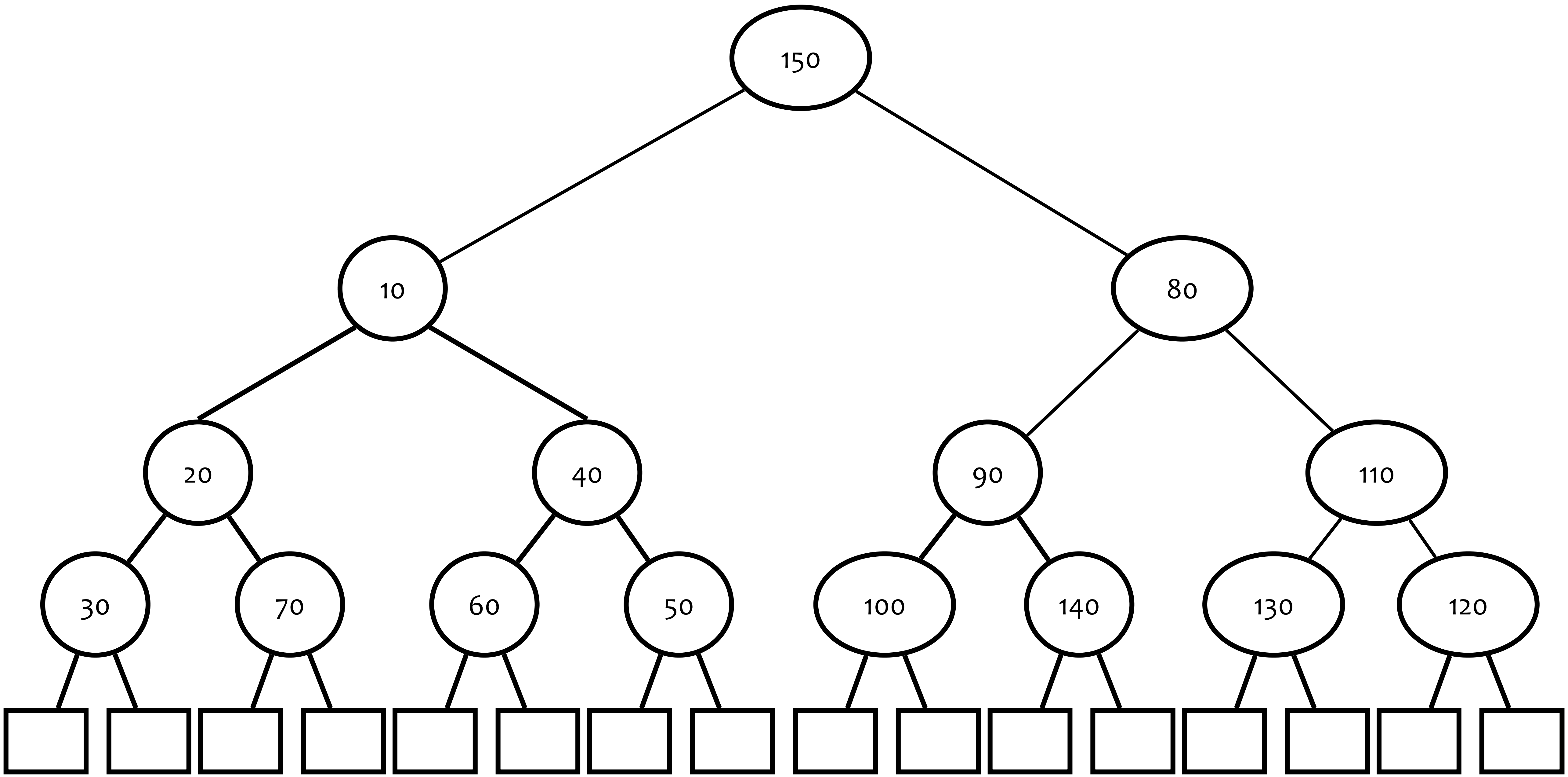
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

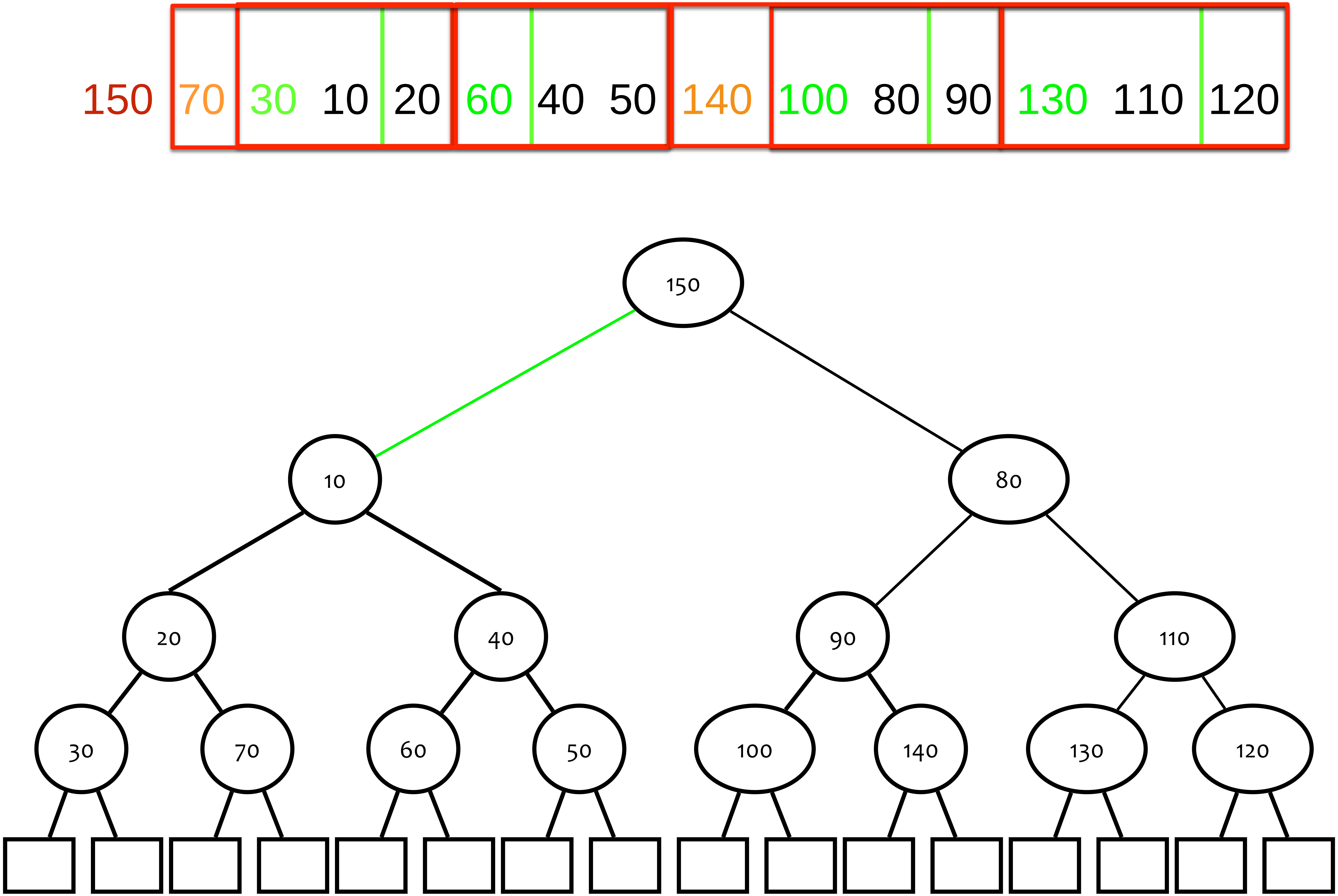


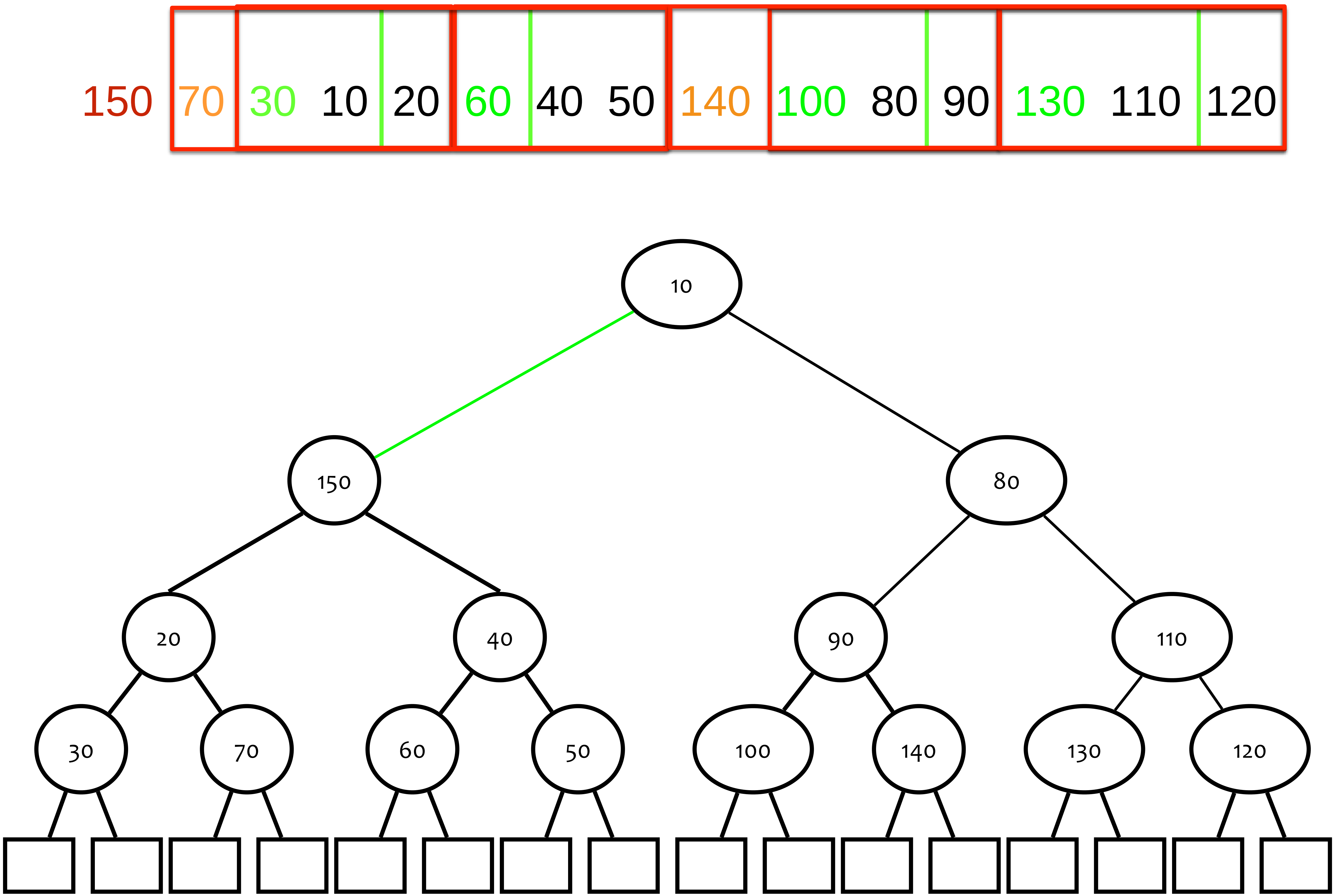

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
80
140
110
100
90
130
120

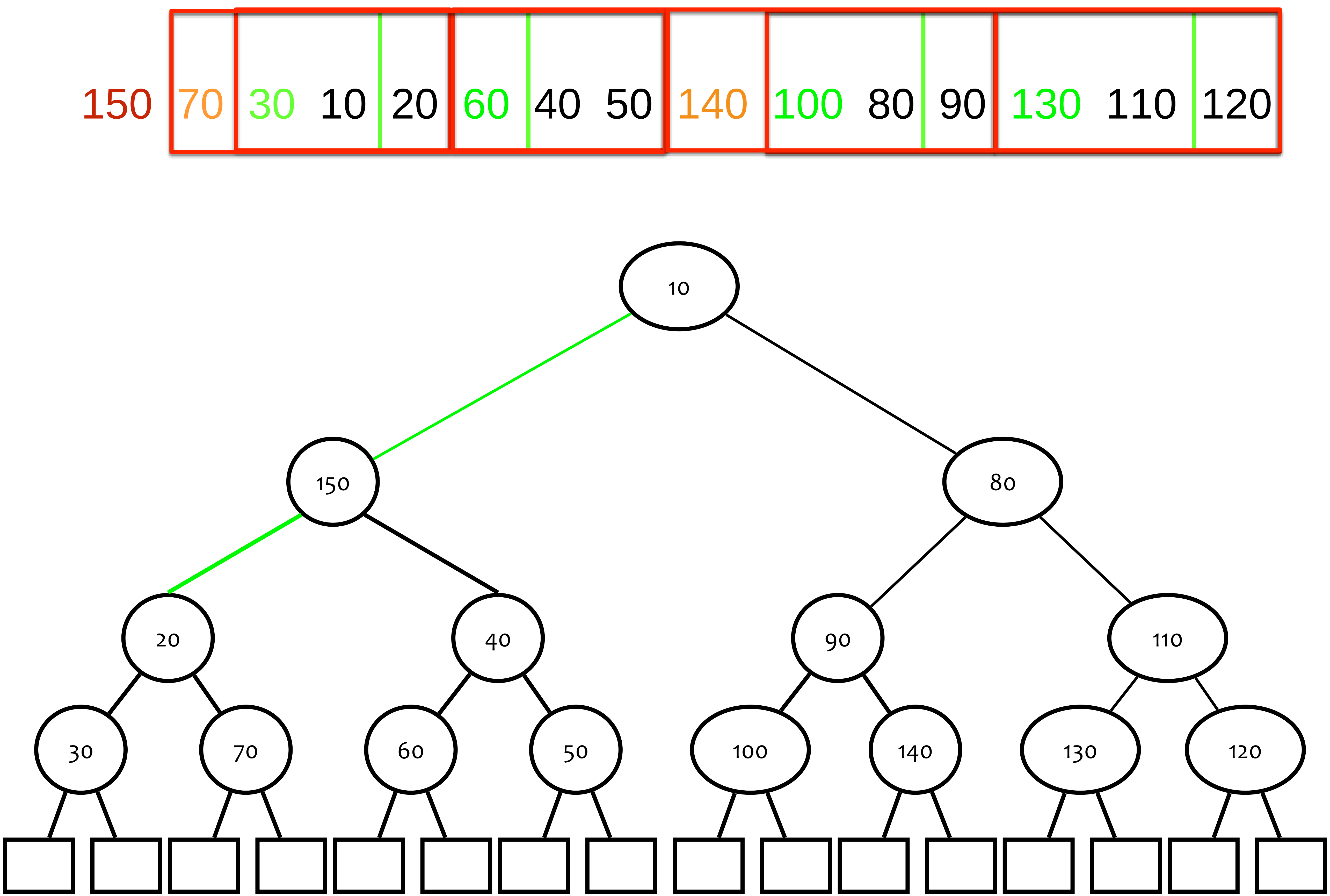

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
80
90
110
100
140
130
120

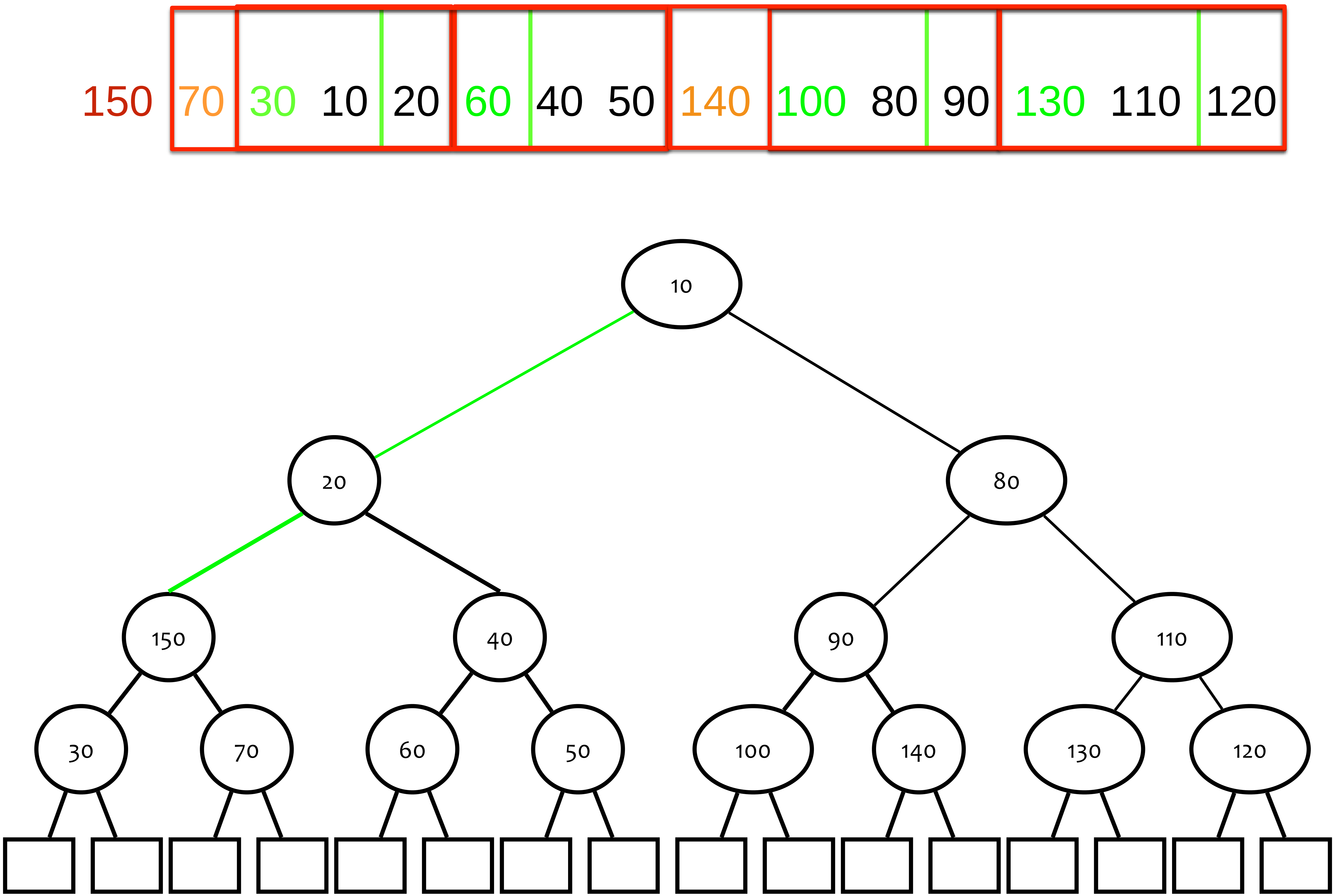
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



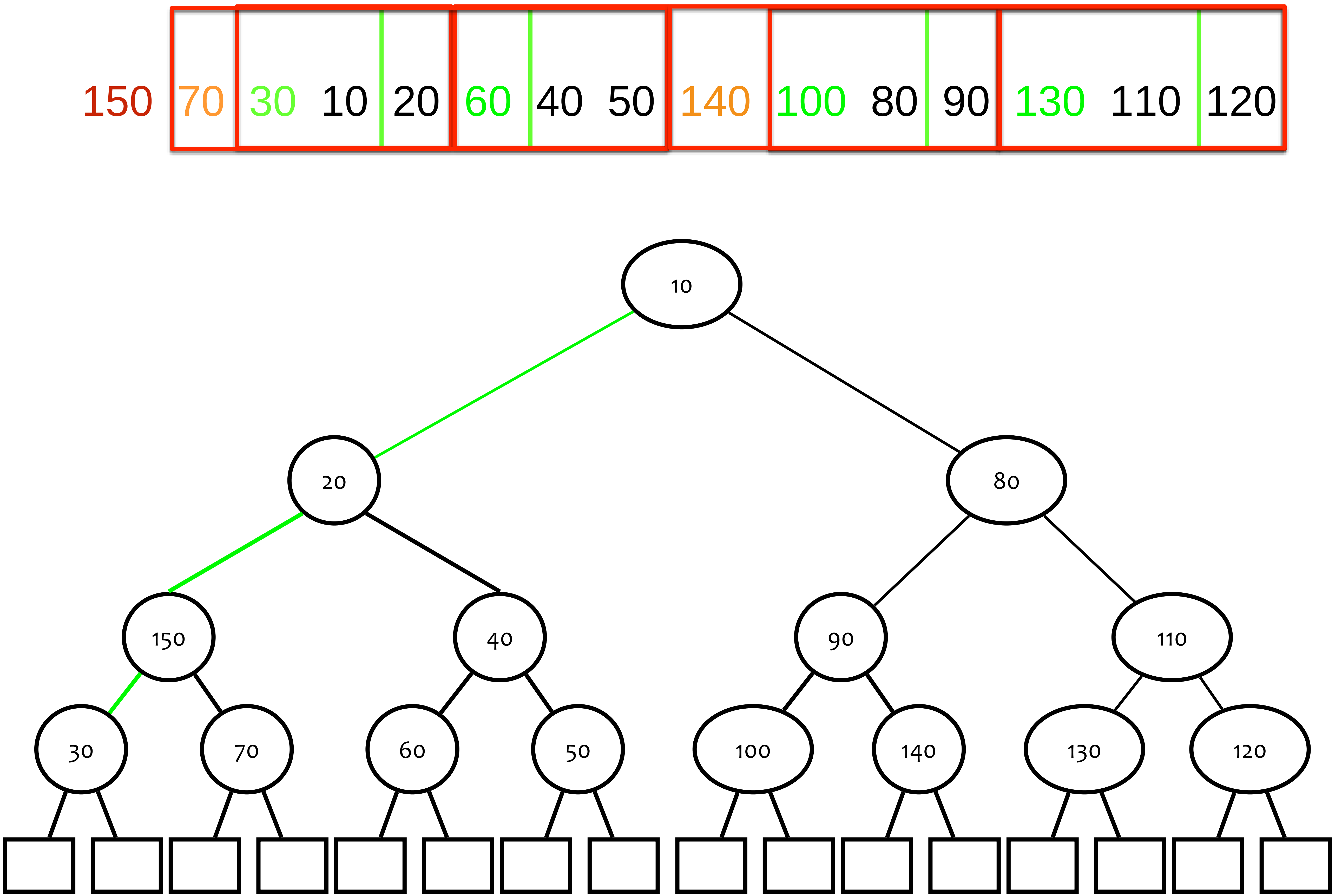

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
150
10
80
20
40
90
110
30
70
60
50
100
140
130
120

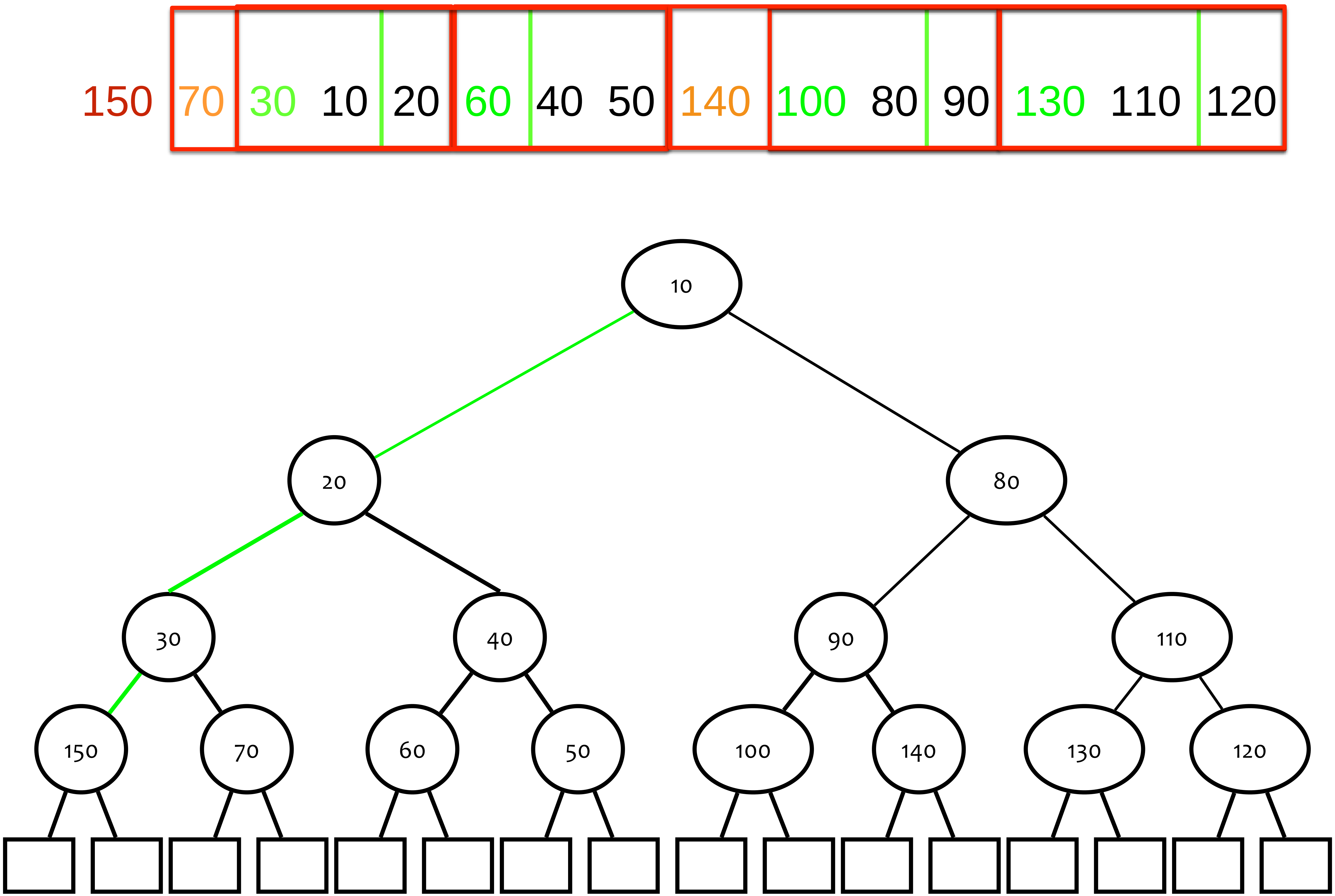

150
70
30 10 20
60 40 50
140
100 80 90
130 110 120
10
150
80
20
40
90
110
30
70
60
50
100
140
130
120


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
150
80
20
40
90
110
30
70
60
50
100
140
130
120

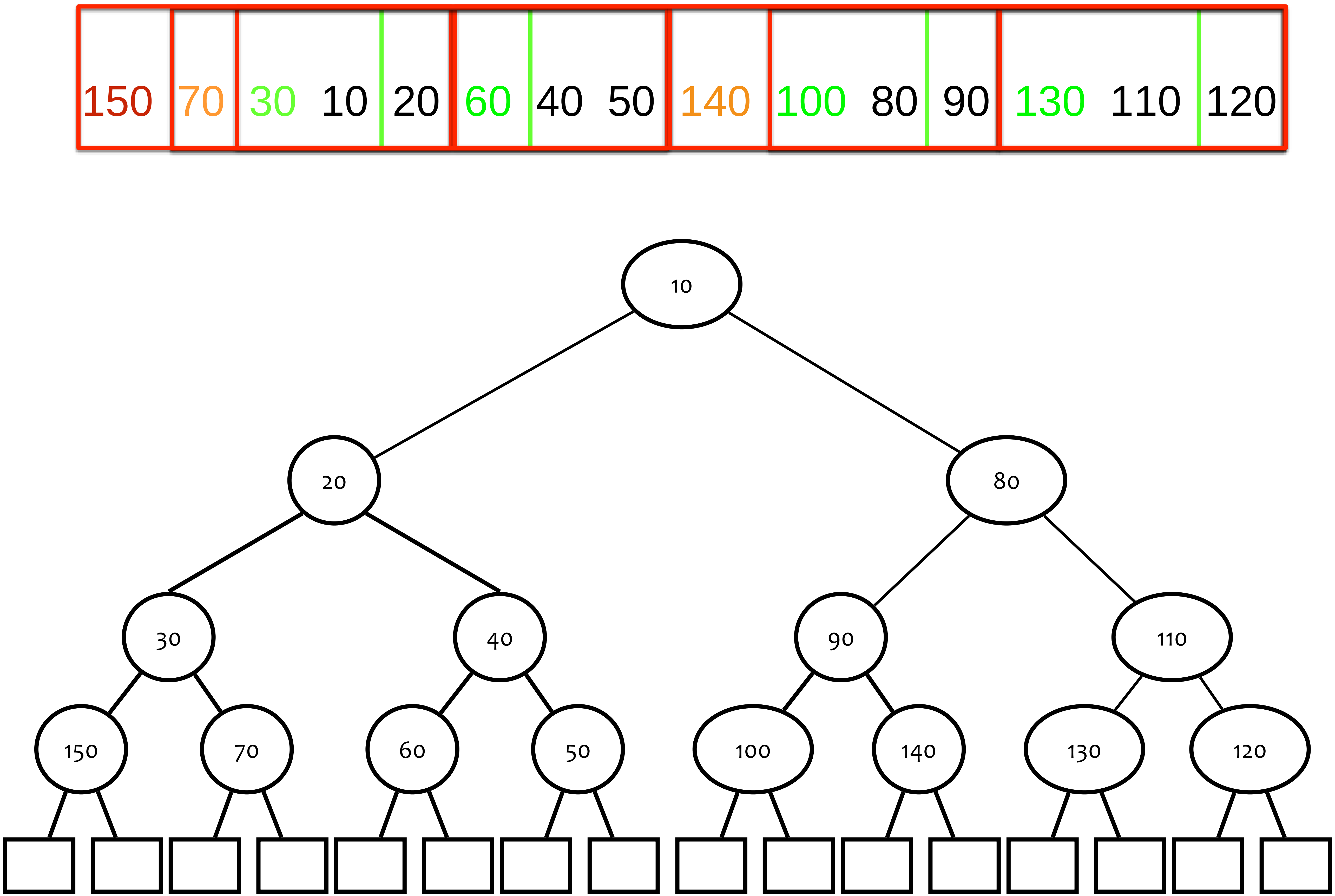


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
80
150
40
90
110
30
70
60
50
100
140
130
120

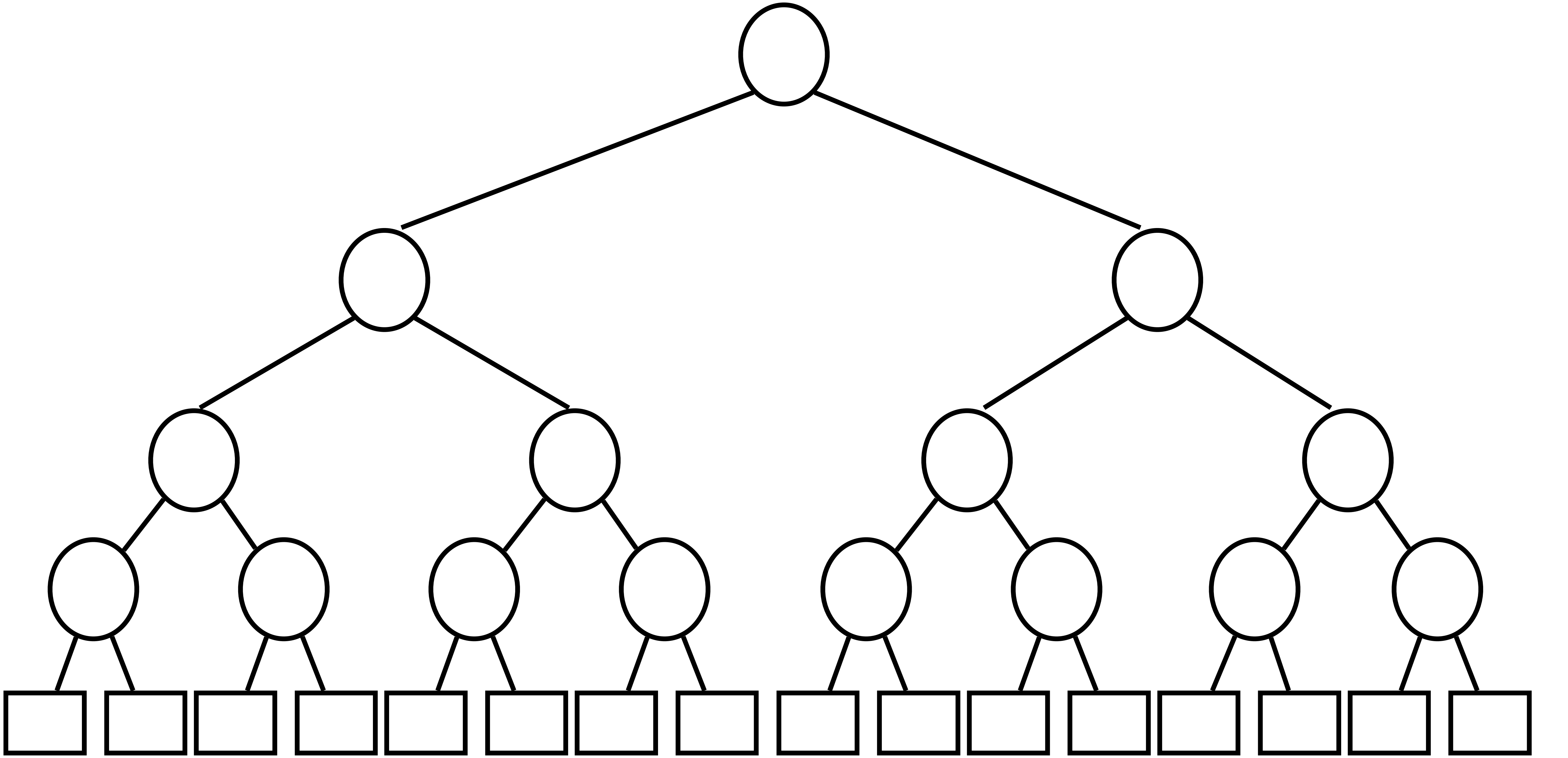

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
80
150
40
90
110
30
70
60
50
100
140
130
120



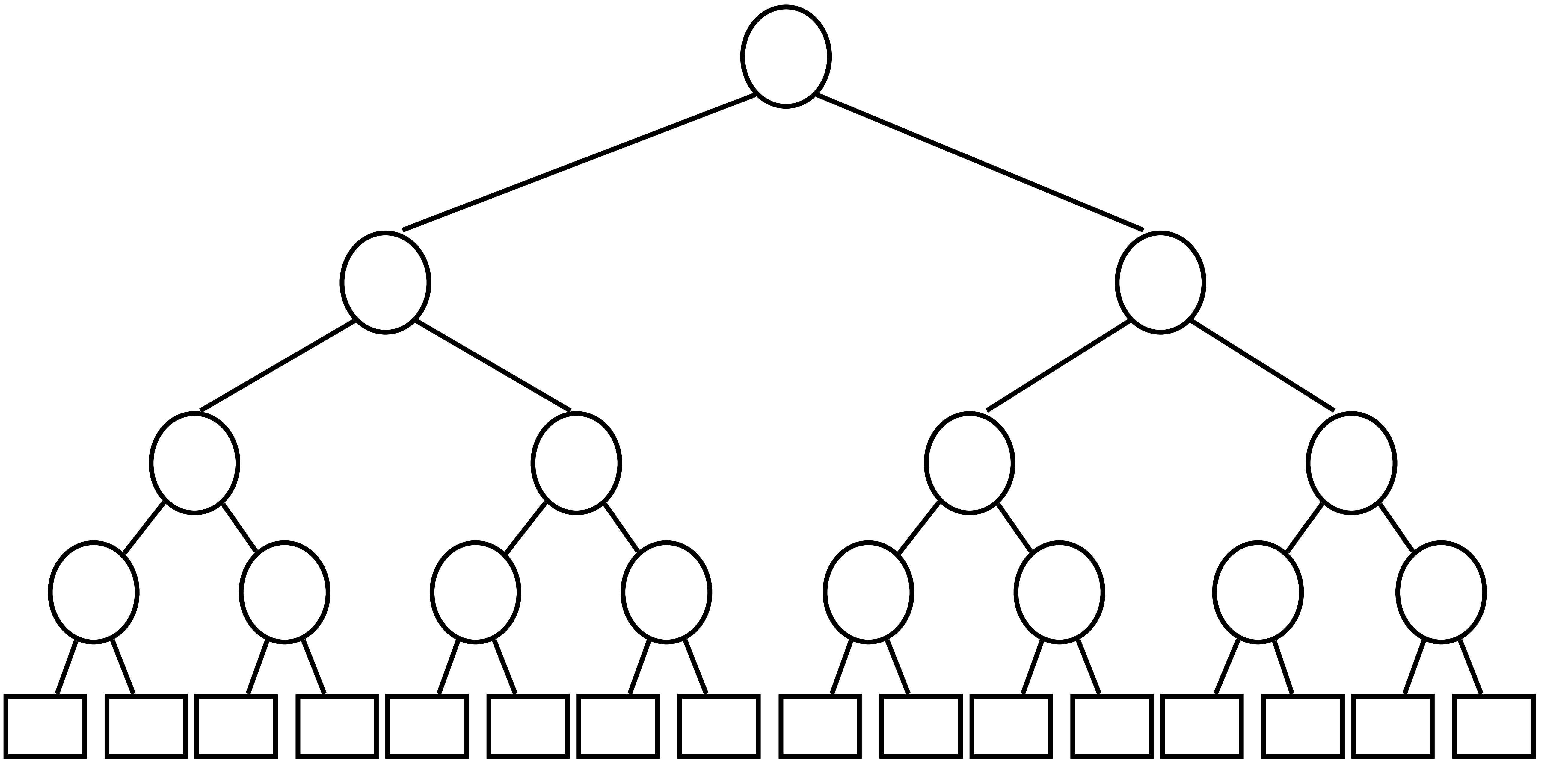
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
80
30
40
90
110
150
70
60
50
100
140
130
120


150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
80
30
40
90
110
150
70
60
50
100
140
130
120

Did we really insert all m elements in $O(m)$ time??



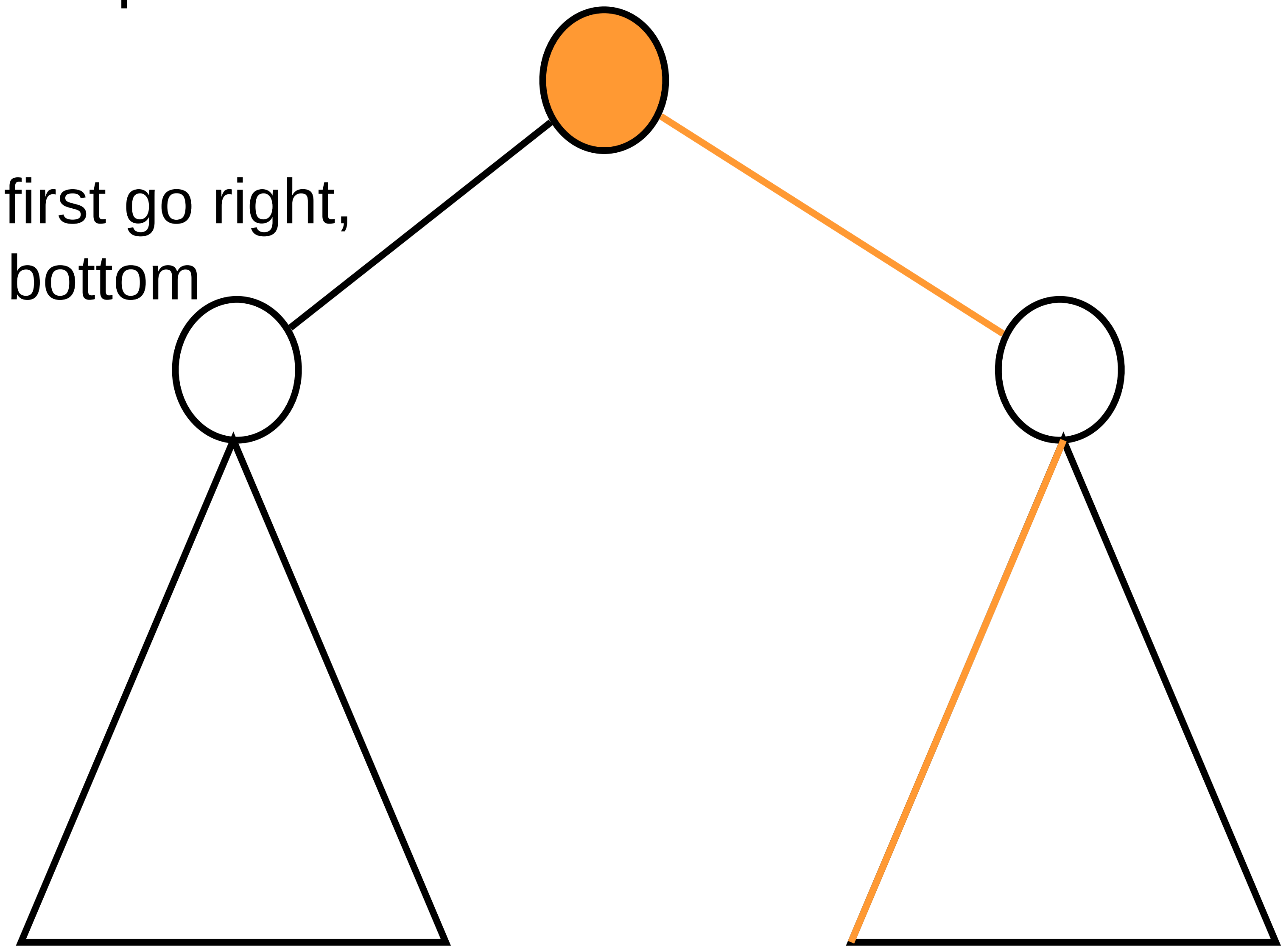
We show: max# bubble down ops < # heap edges



Proof idea

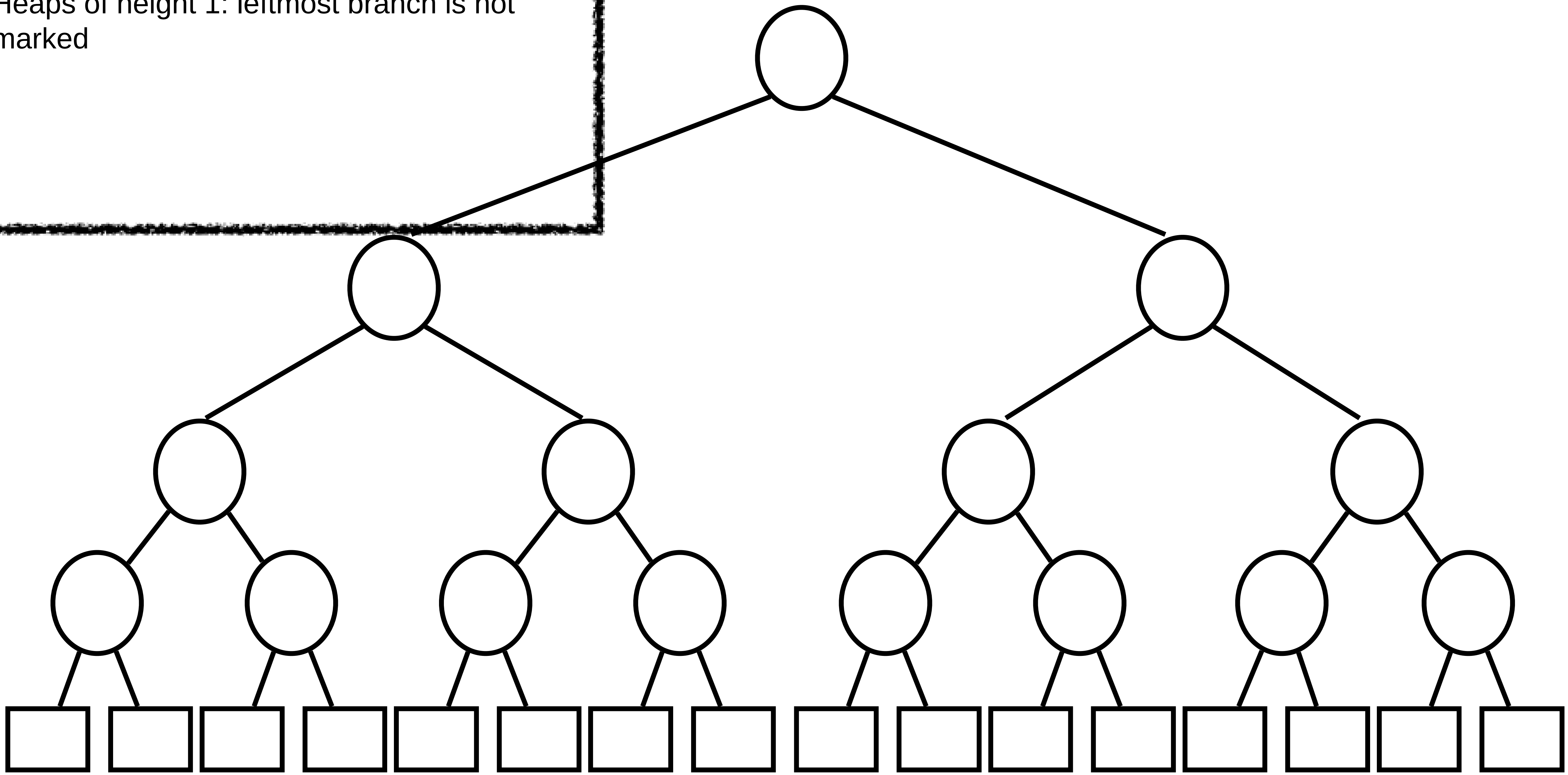
For each new node joining two heaps: mark path of maximum number of bubble-down operations

For marking: first go right, then left until bottom

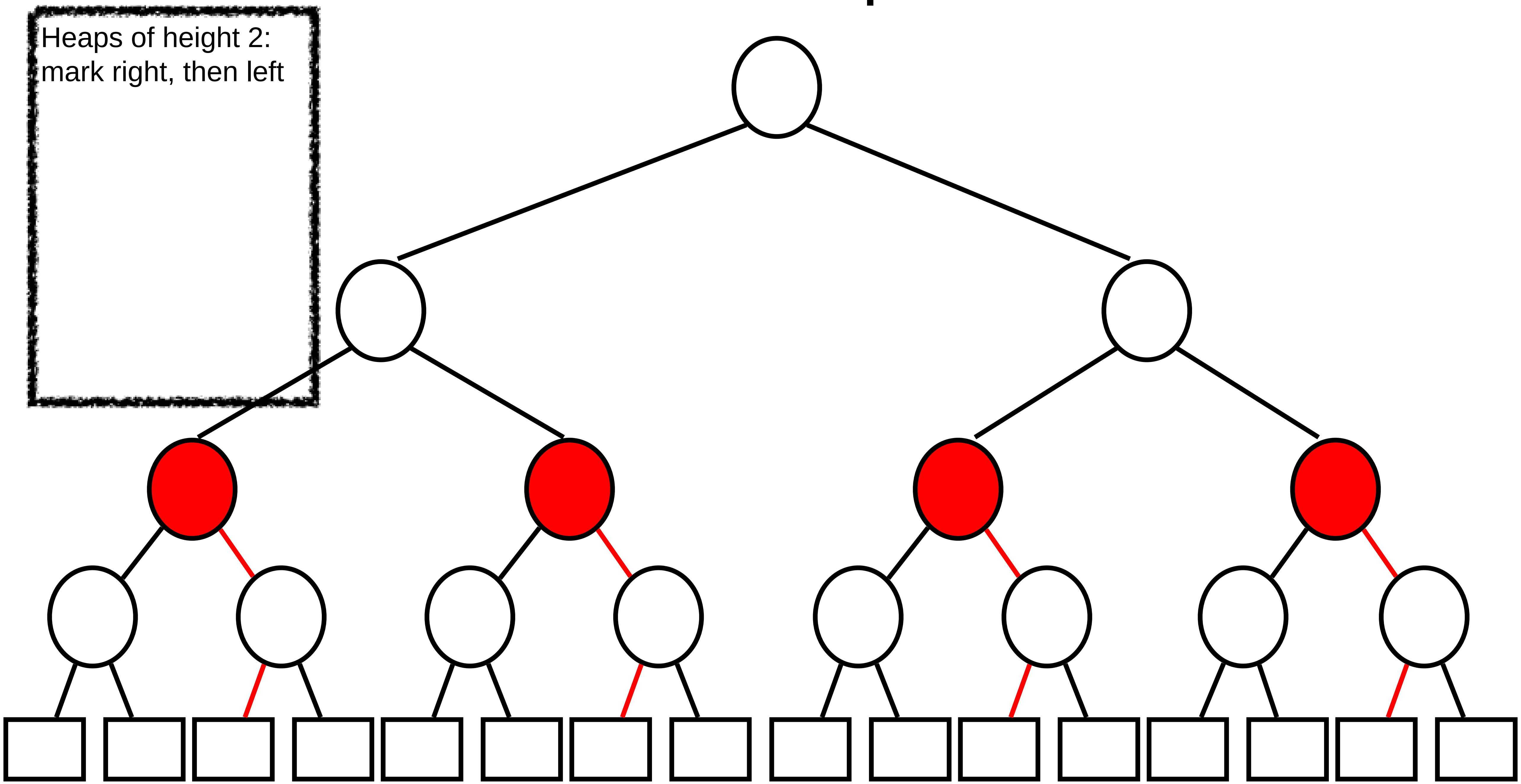


For each new node joining two heaps: mark path with maximum number of bubble-down operations

Heaps of height 1: leftmost branch is not marked

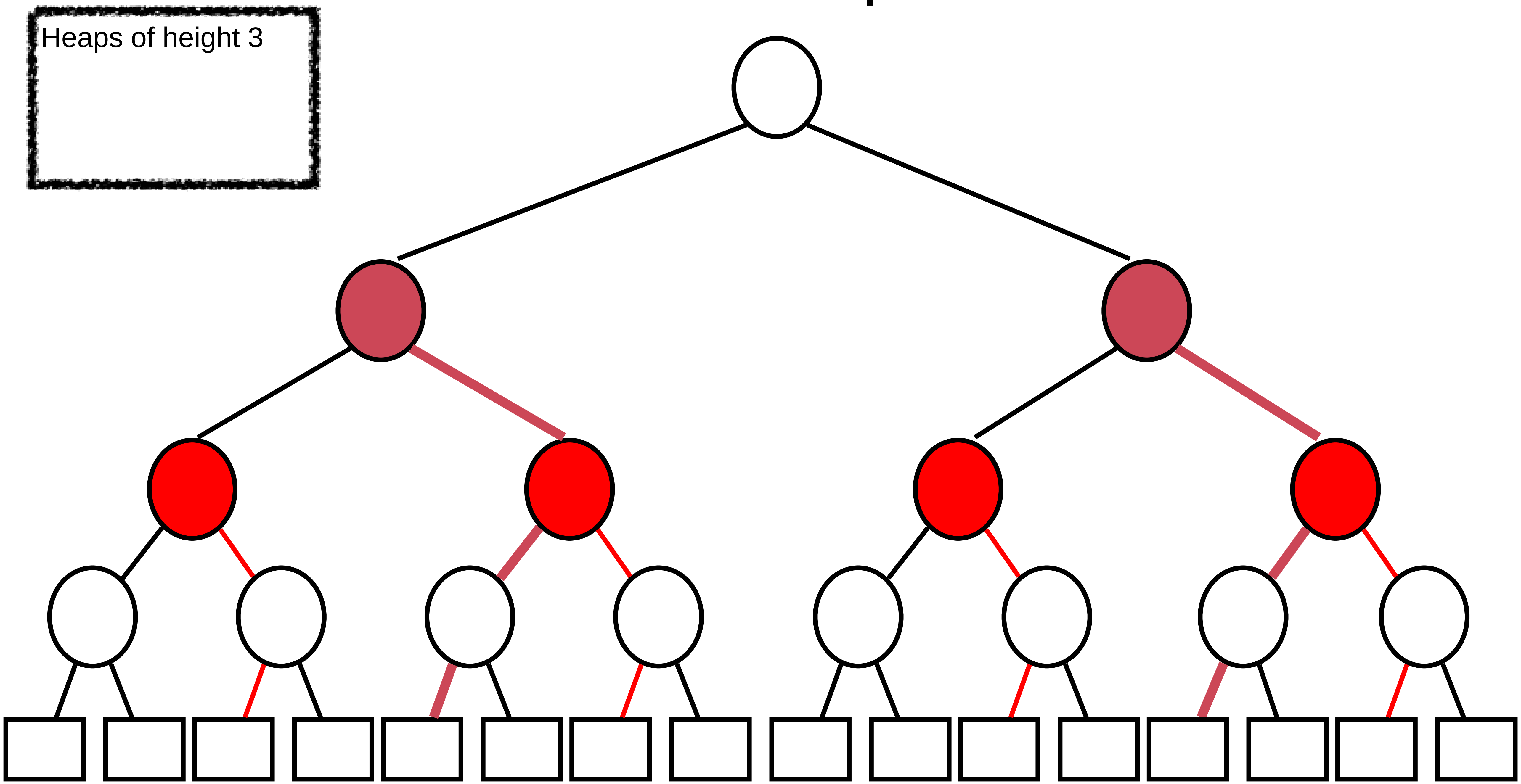


For each new node joining two heaps: mark path with maximum number of bubble-down operations



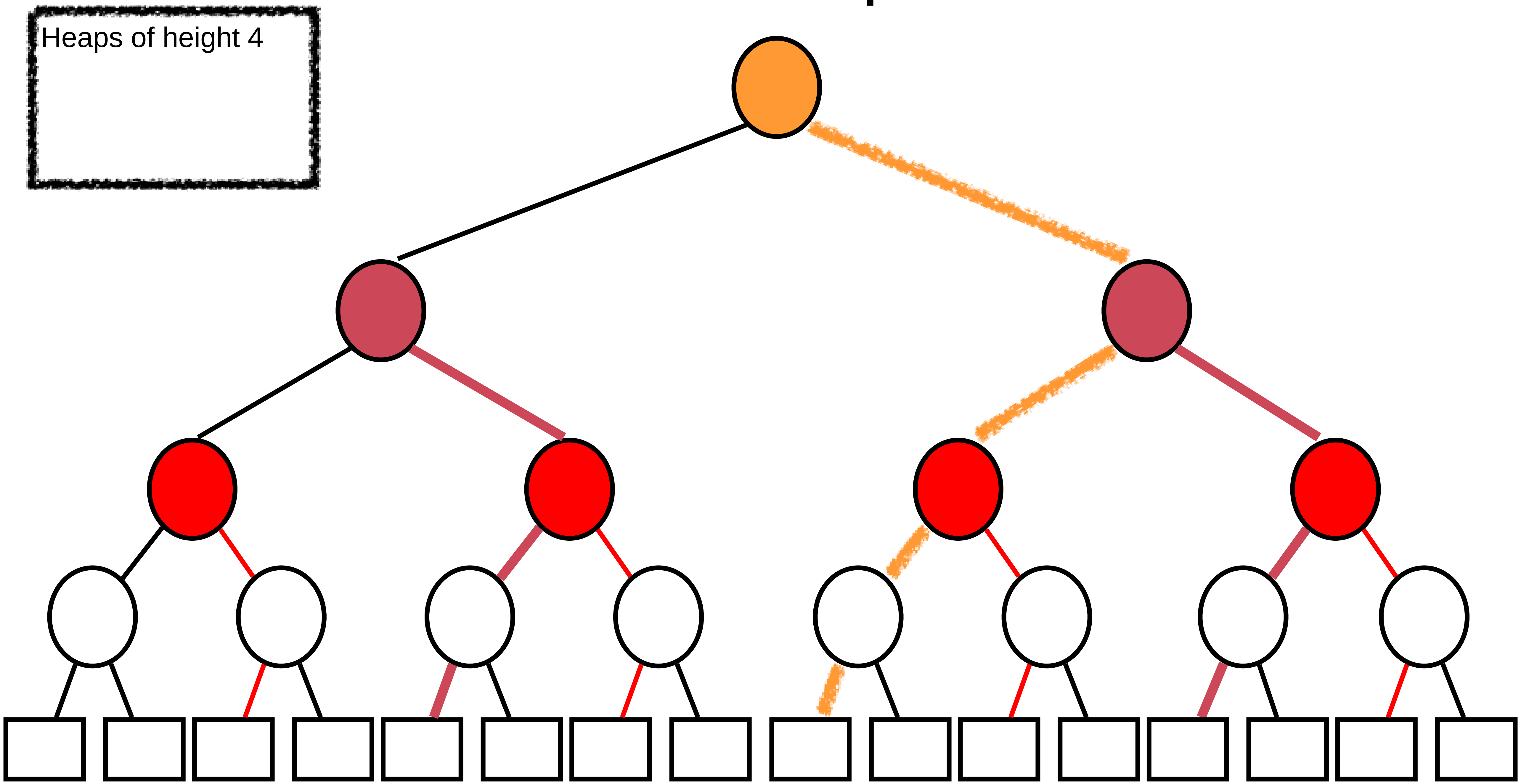
For each height-2 heap, leftmost branch not marked

For each new node joining two heaps: mark path with maximum number of bubble-down operations



For each height-3 heap, leftmost branch not marked

For each new node joining two heaps: mark path with maximum number of bubble-down operations



For height-4 heap, leftmost branch not marked

Inductive argument: marking procedure will never mark all edges in heap, since the leftmost branch is never marked

- Note: leftmost branch in height-h heap: not marked
- When joining 2 heaps of height h to heap of height $h + 1$: new edges to be marked are
  - edge joining new node and right heap of height h, and
  - edges on left path in the right heap of height h
- We conclude: leftmost branch in height ($h+1$) heap is not marked

Build Heap In-place

Algorithm buildHeap(A,n):
  for $i \leftarrow \lfloor n/2 \rfloor$ **to** 1 **do**
    downHeap(A,i)

Algorithm downHeap(A,i):
  $l \leftarrow 2i$
  $r \leftarrow 2i + 1$
  if $l \leq n \wedge A[l] < A[i]$ **then**
    $min \leftarrow l$
  else
    $min \leftarrow i$
  if $r \leq n \wedge A[r] < A[min]$ **then**
    $min \leftarrow r$
  if $i \neq min$ **then**
    swap(i,min)
    downHeap(A,min)