Building up a heap in linear time

Building up a heap

Building up a heap

- n standard insert-operations for a heap result in $O(n \log(n))$ time.

Building up a heap

- n standard insert-operations for a heap result in $O(n \log(n))$ time.

- Can we build up a heap for n given elements faster? Is $O(n)$ possible?

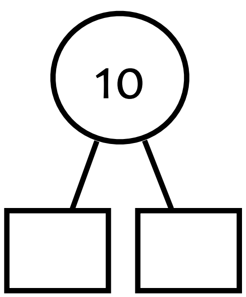
150 70 30 10 20 60 40 50 | 140 100 80 90 130 110 120

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

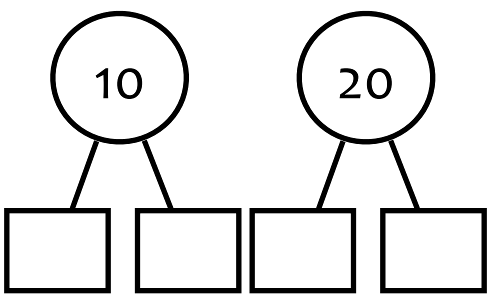
150 70 30 10 20 | 60 40 50 | 140 100 80 90 130 110 120

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

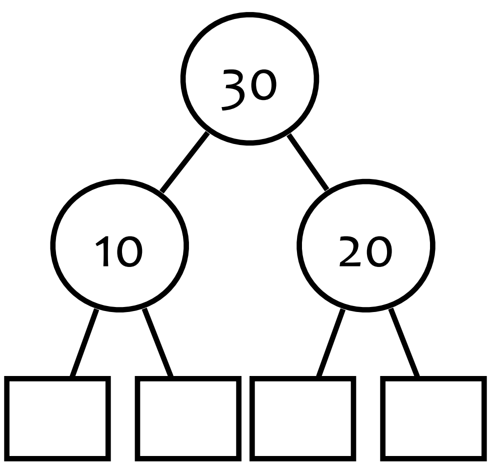
150 70 30 10 | 20 | 60 40 50 | 140 100 80 90 130 110 120



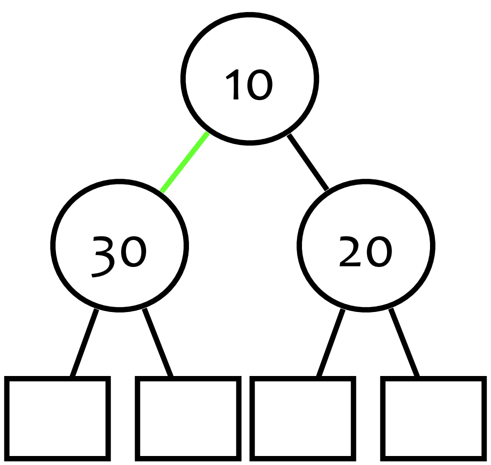
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



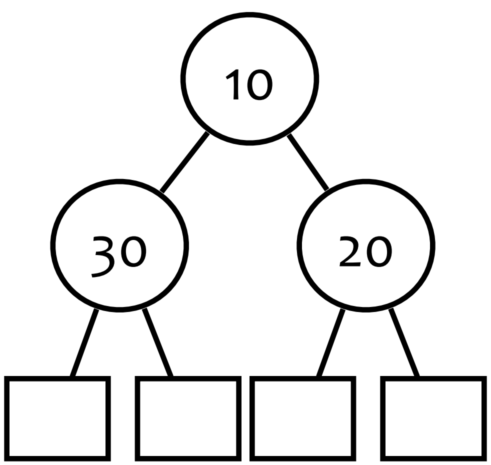
150 70 30 10 | 20 | 60 40 50 | 140 100 80 90 130 110 120



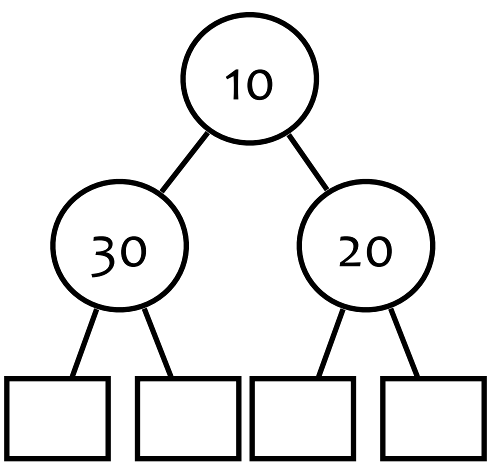
150 70 30 10 | 20 | 60 40 50 | 140 100 80 90 130 110 120



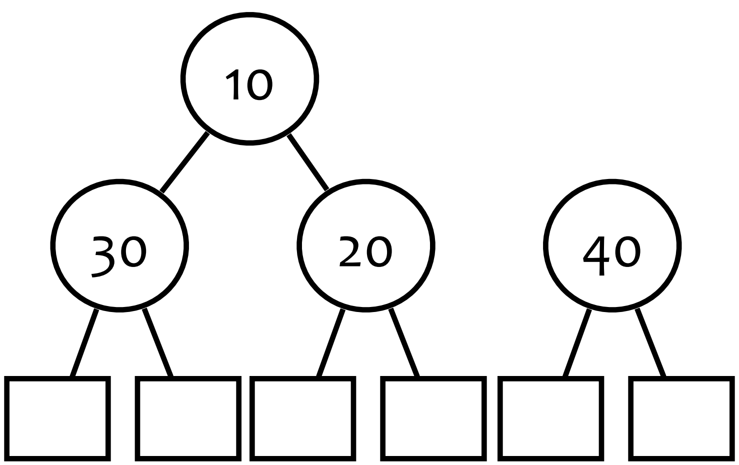
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



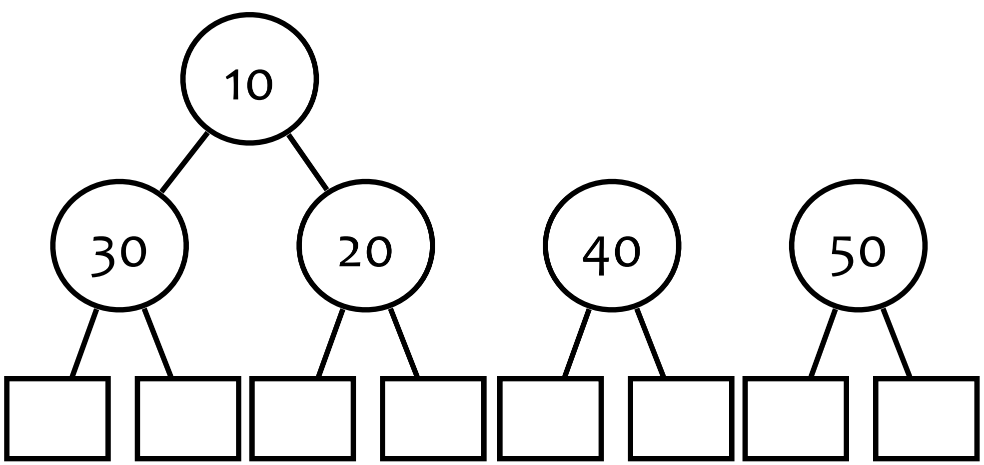
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



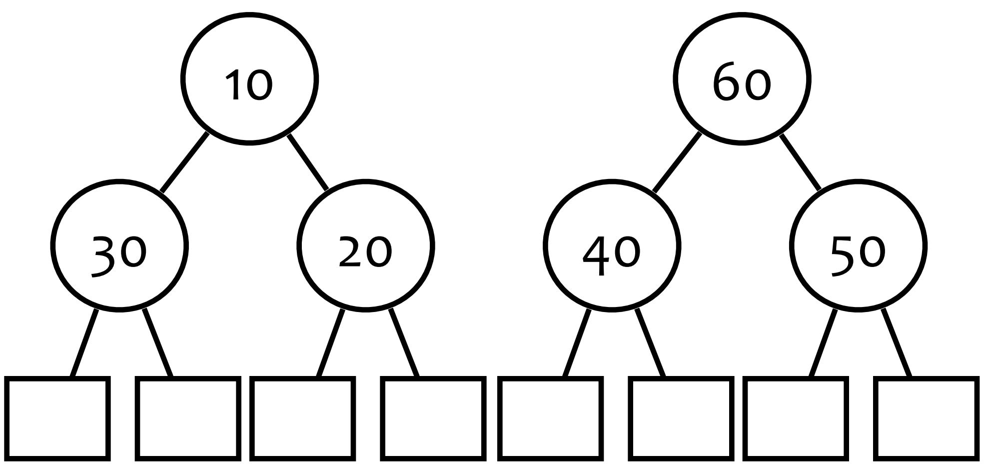
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



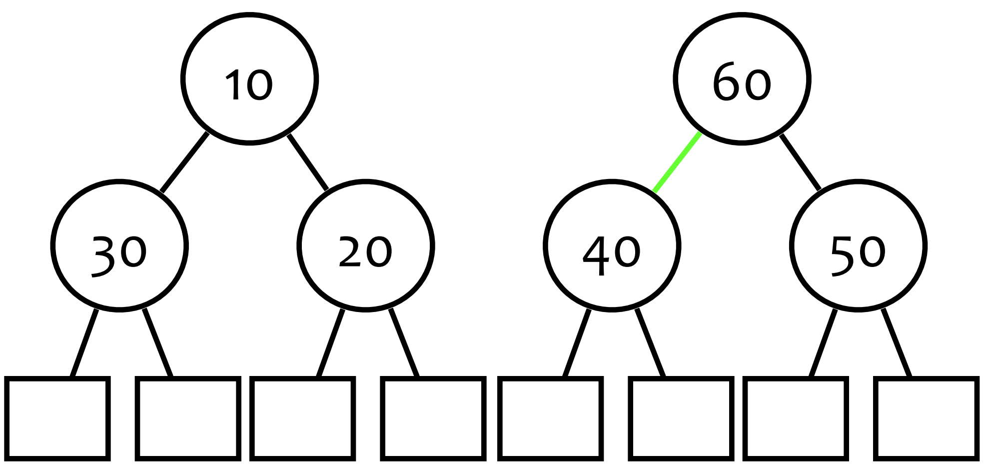
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



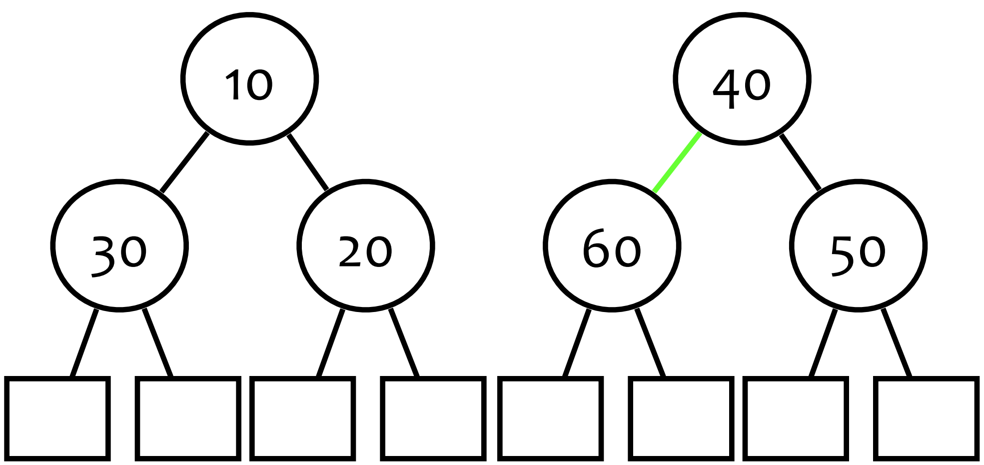
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



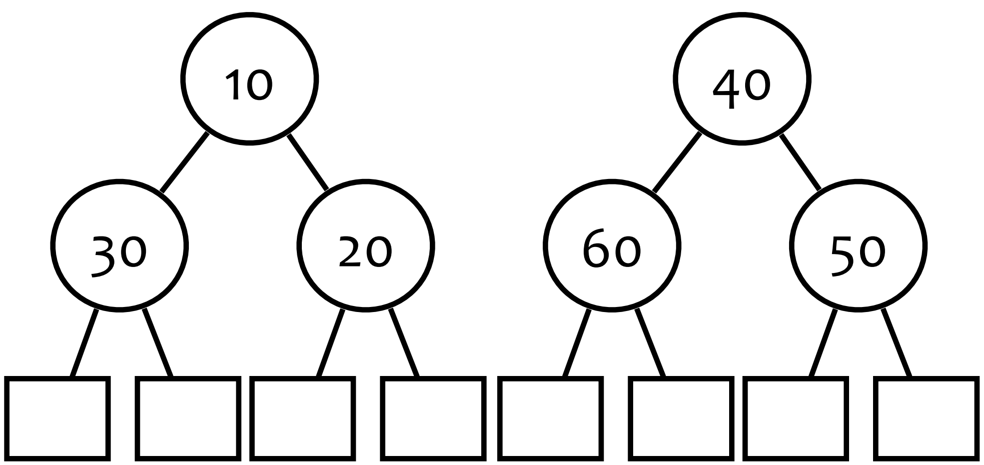
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



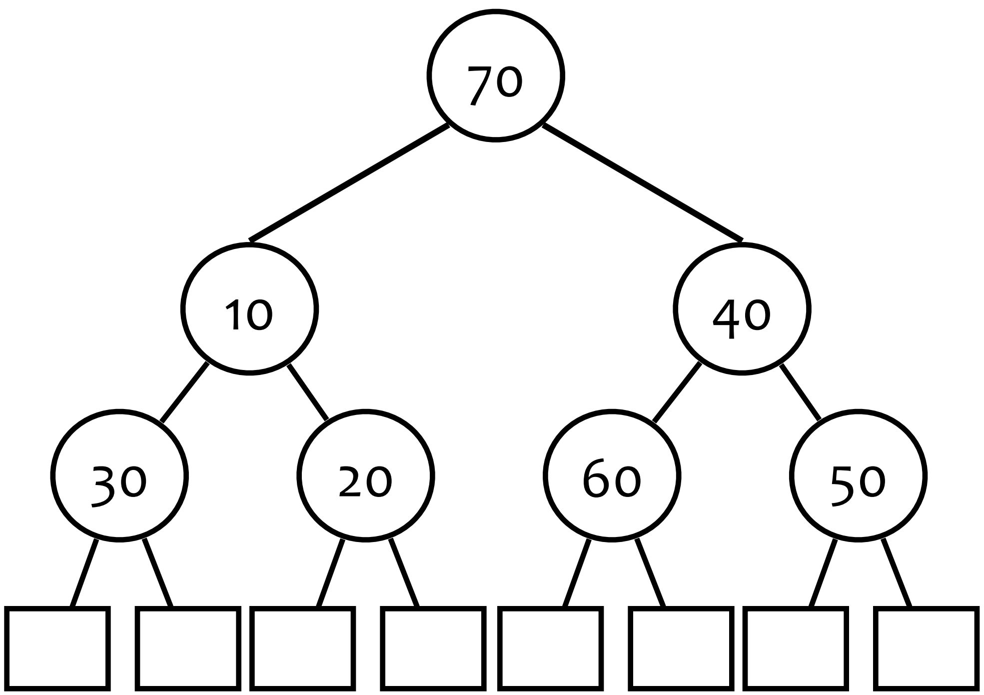
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



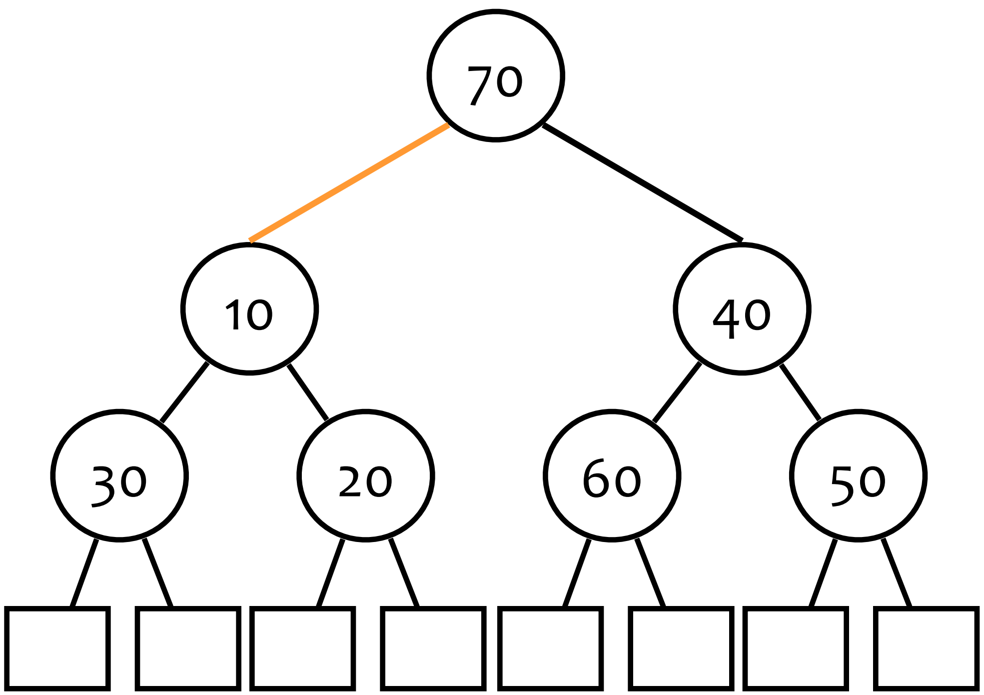
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



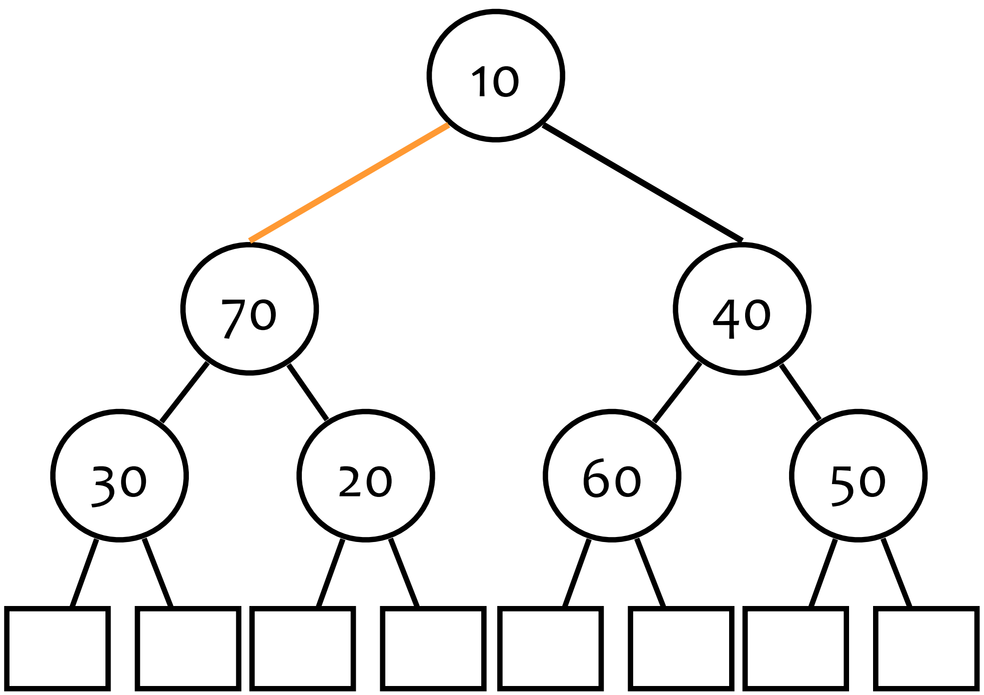
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



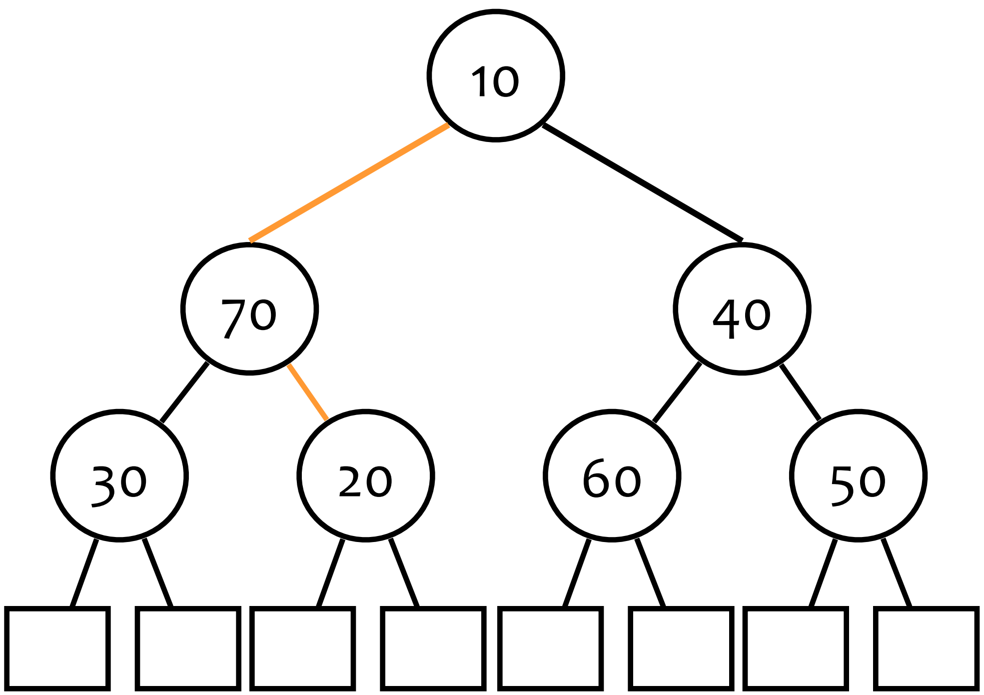
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



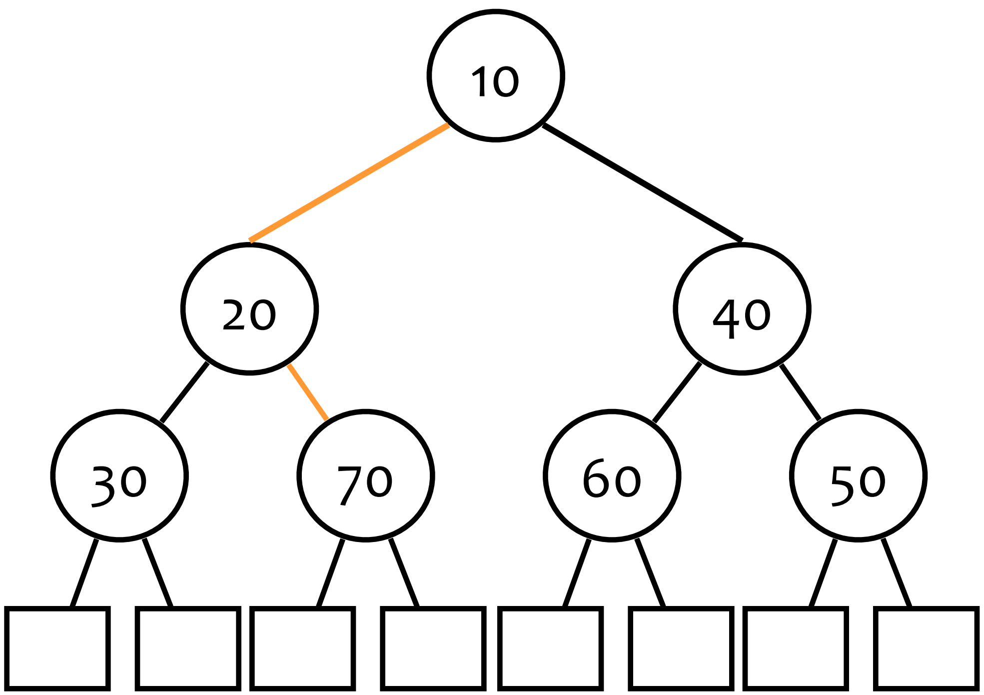
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



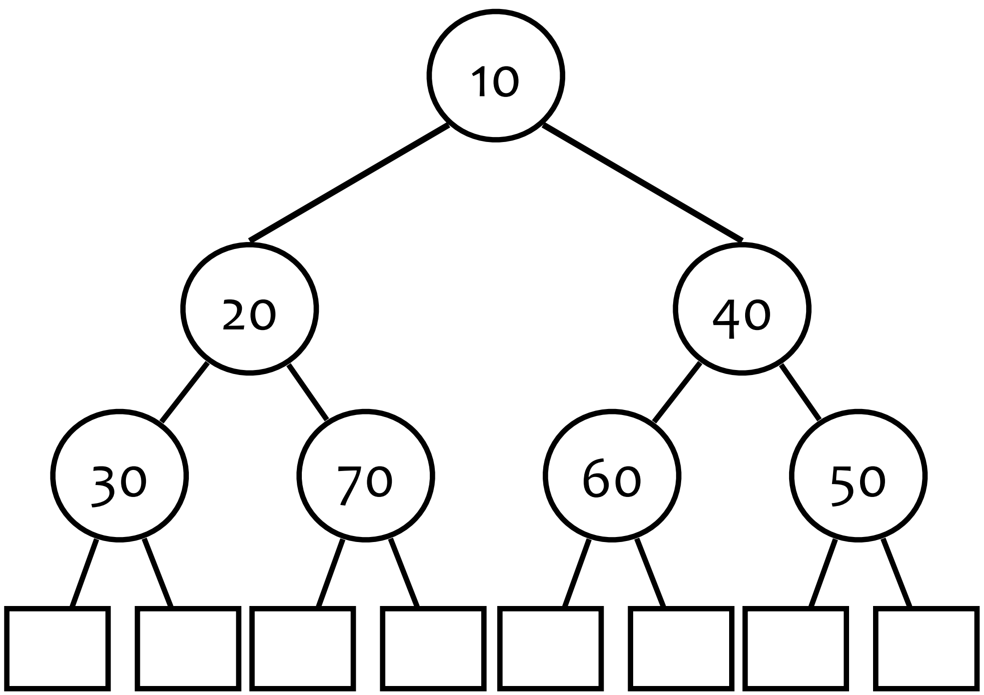
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



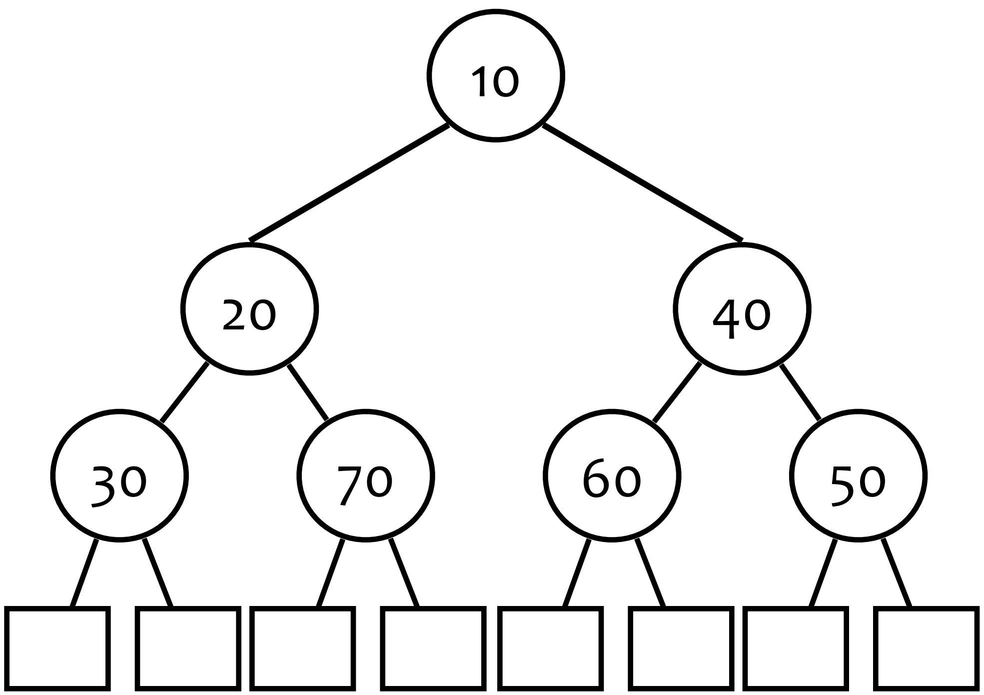
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



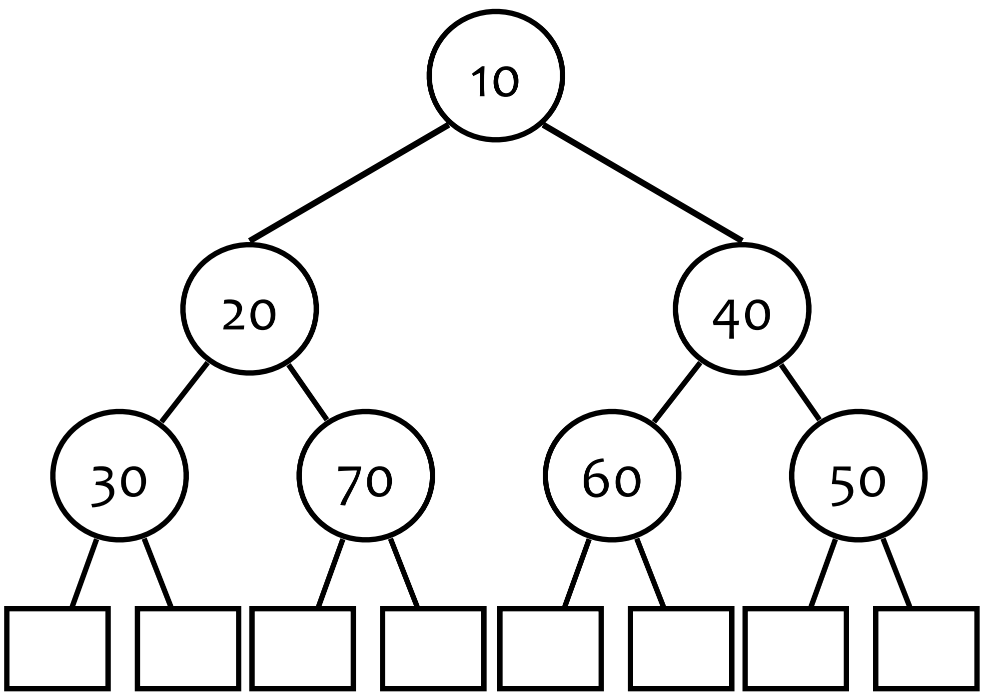
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



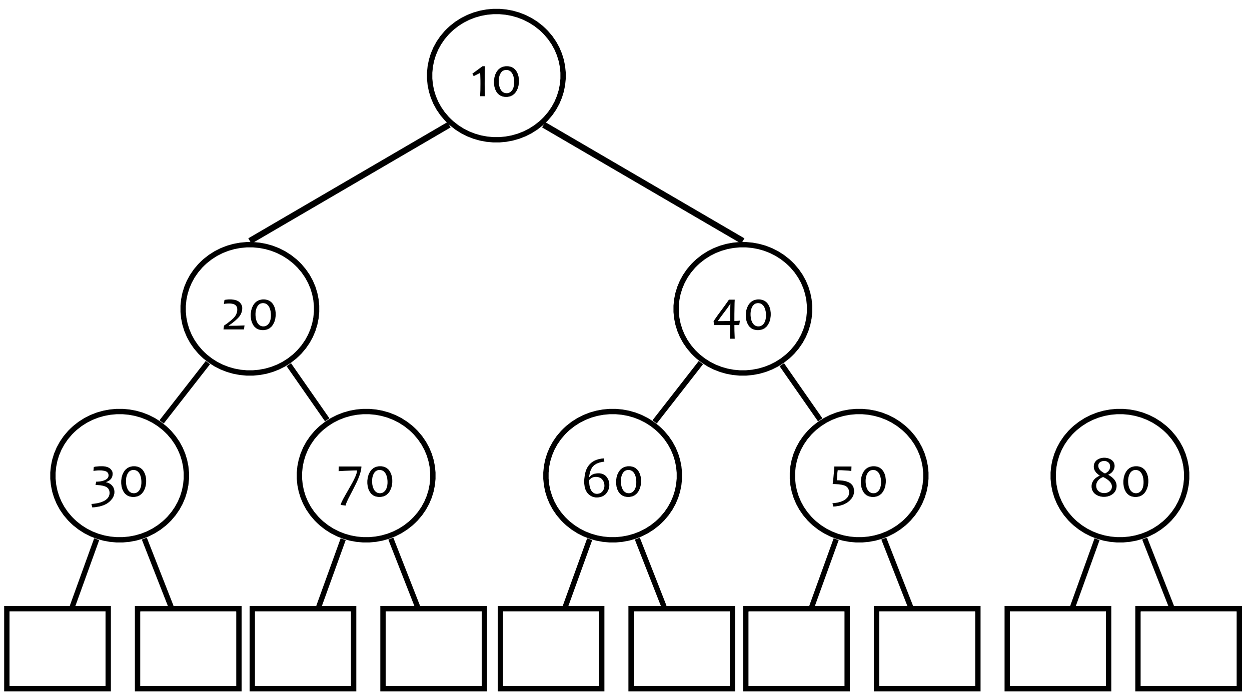
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



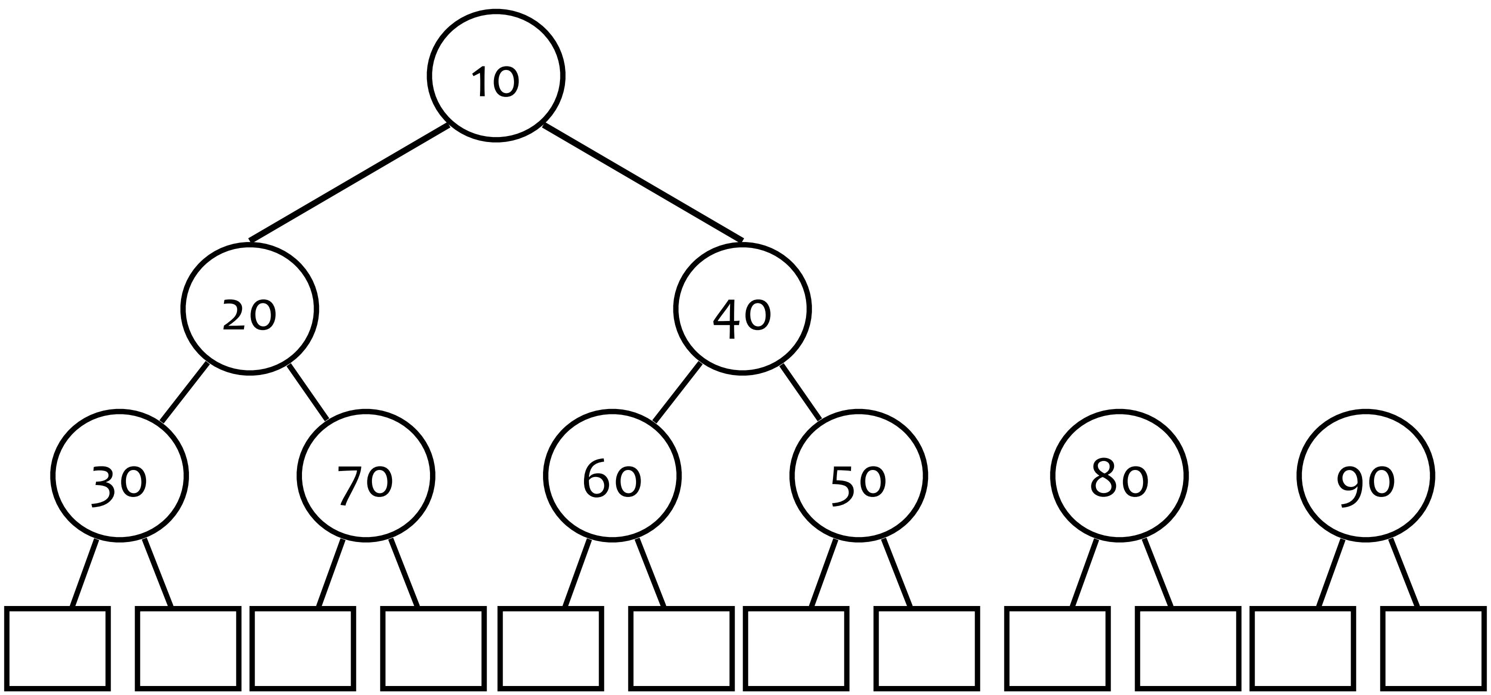
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



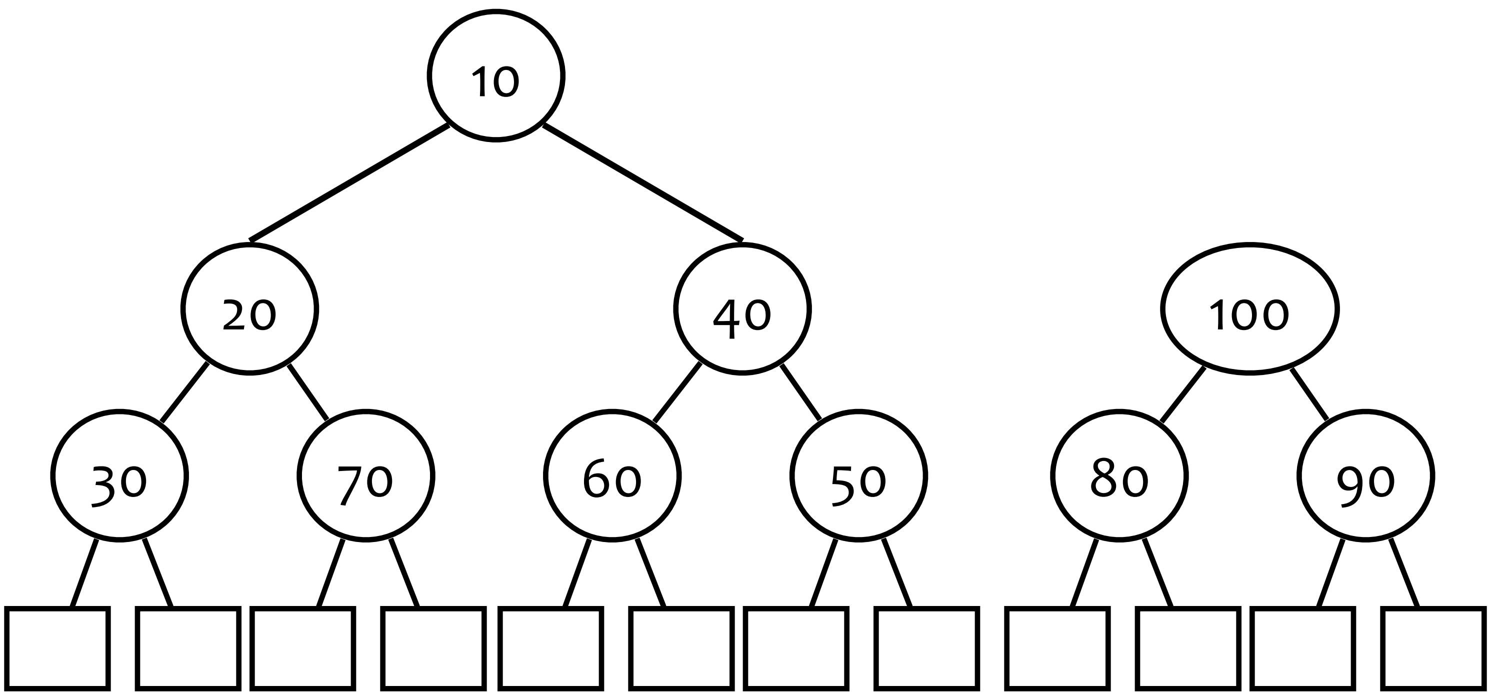
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



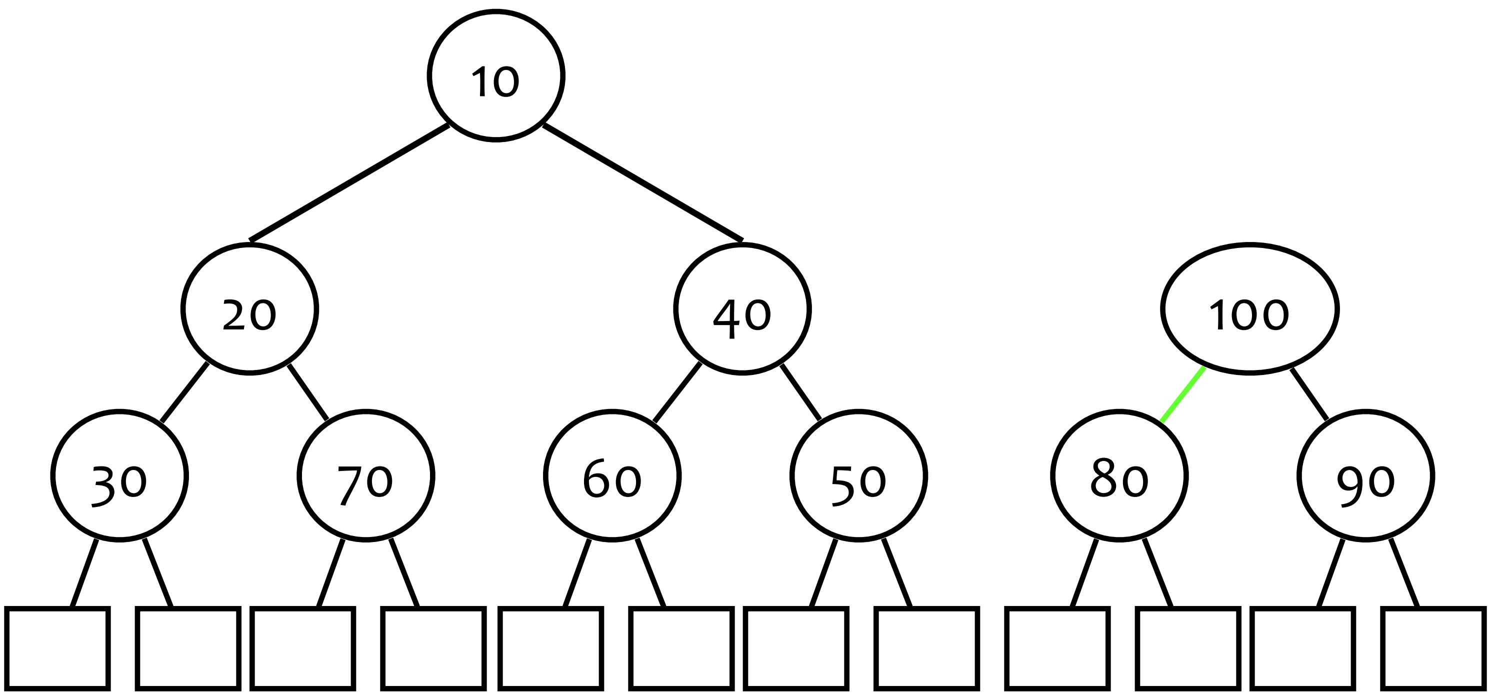
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



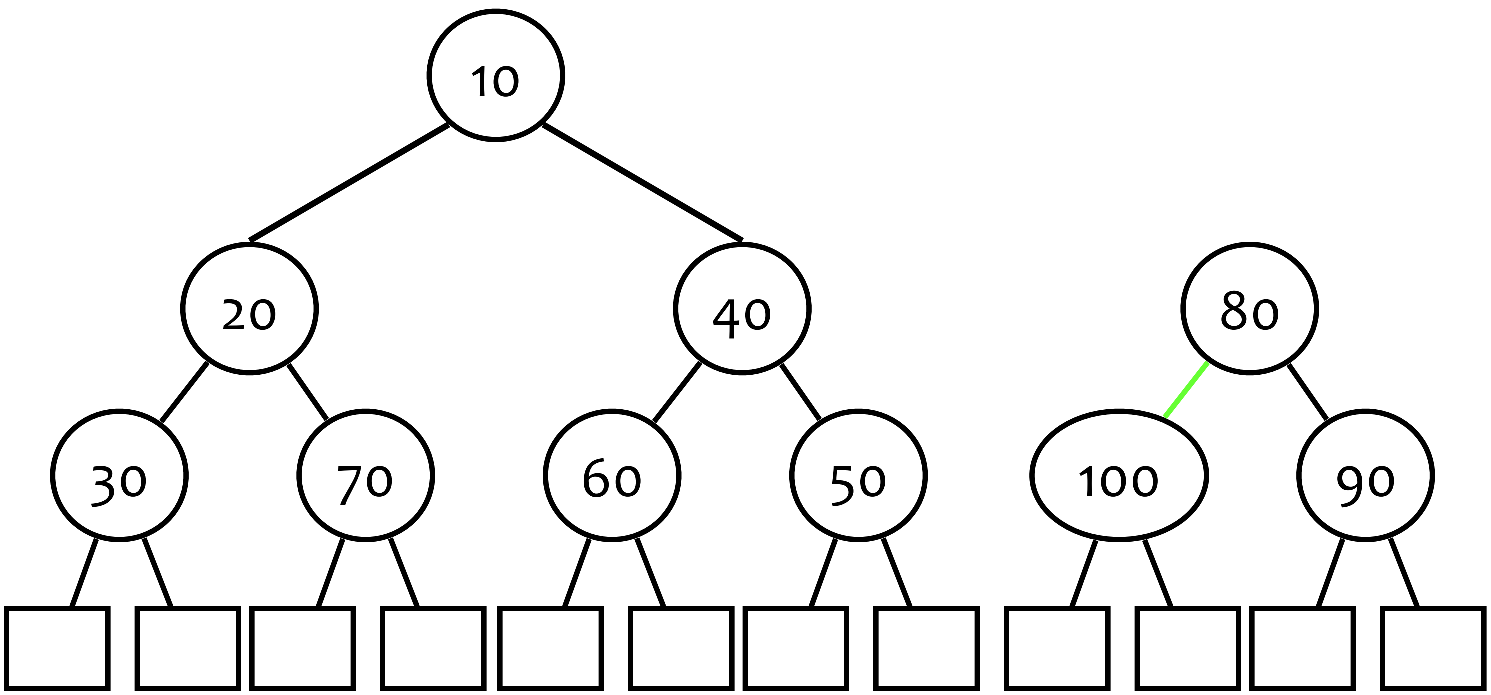
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



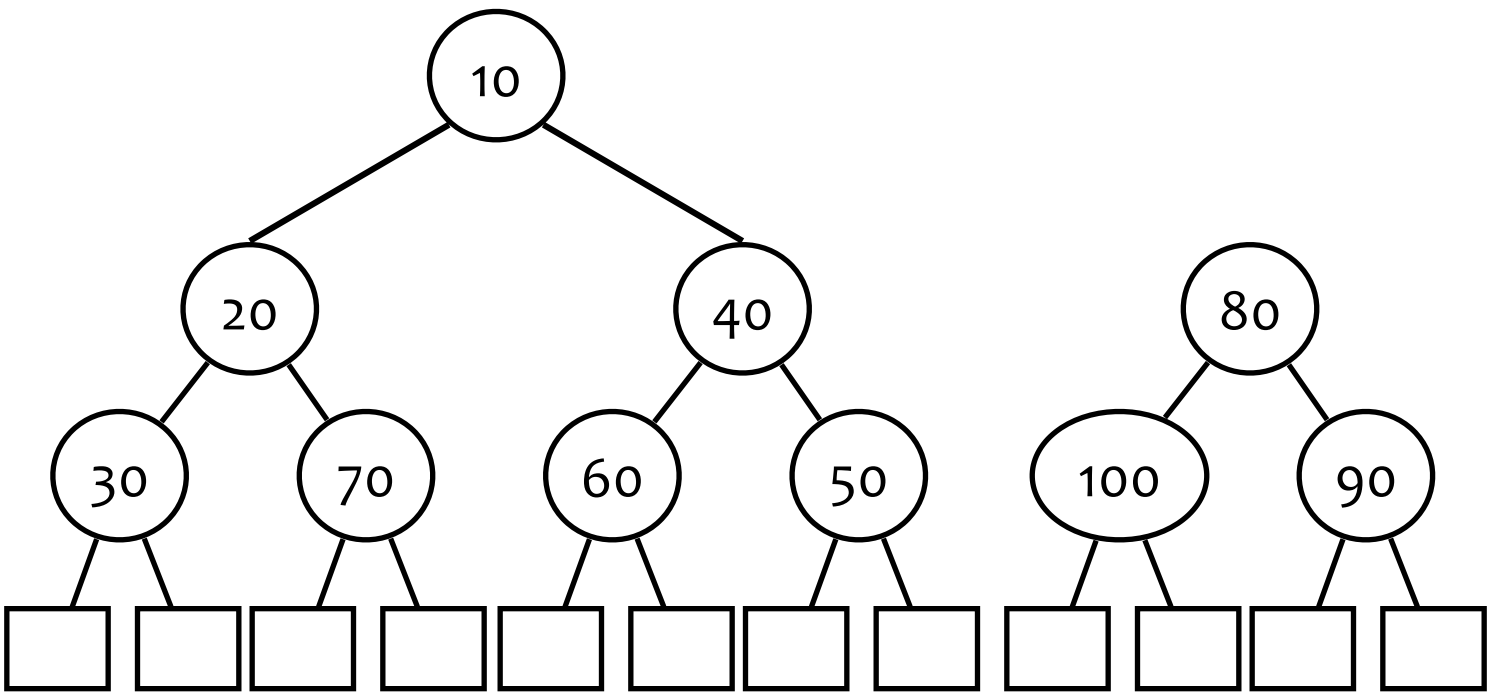
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



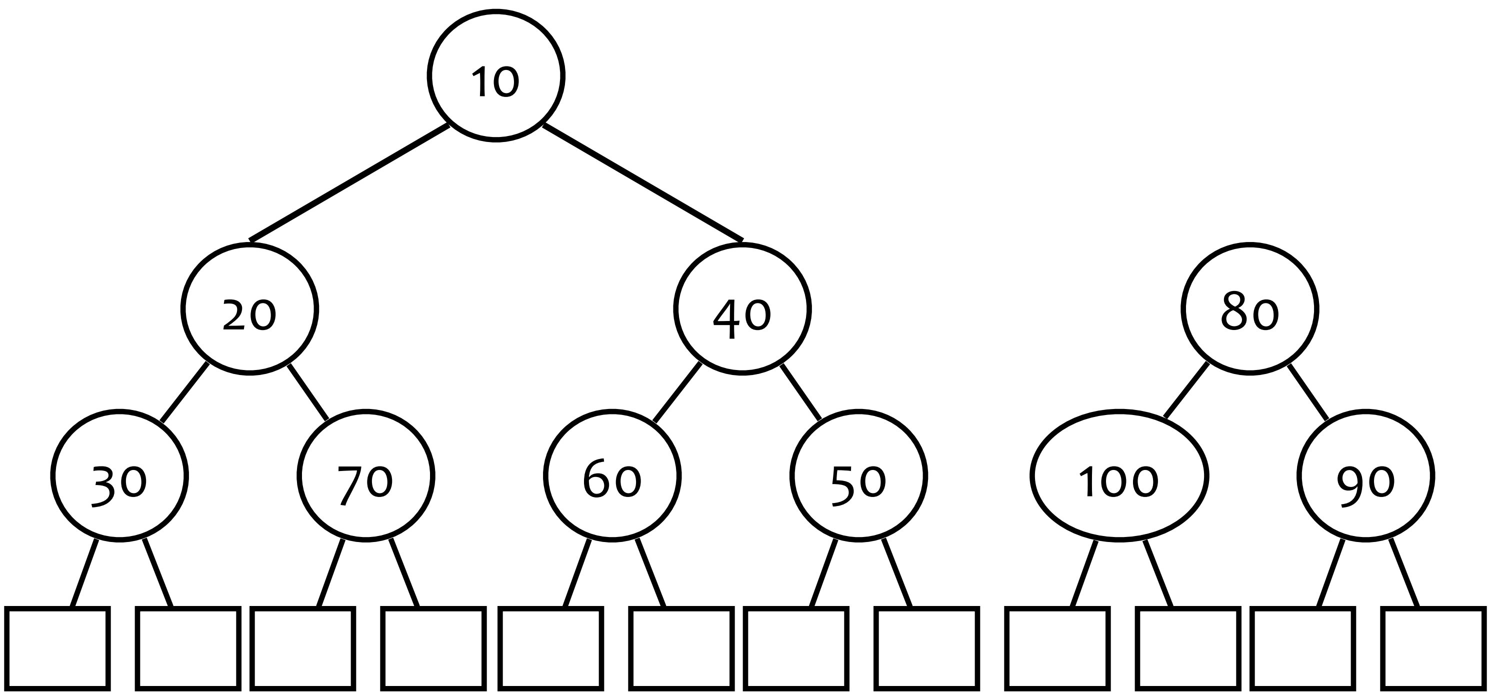
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



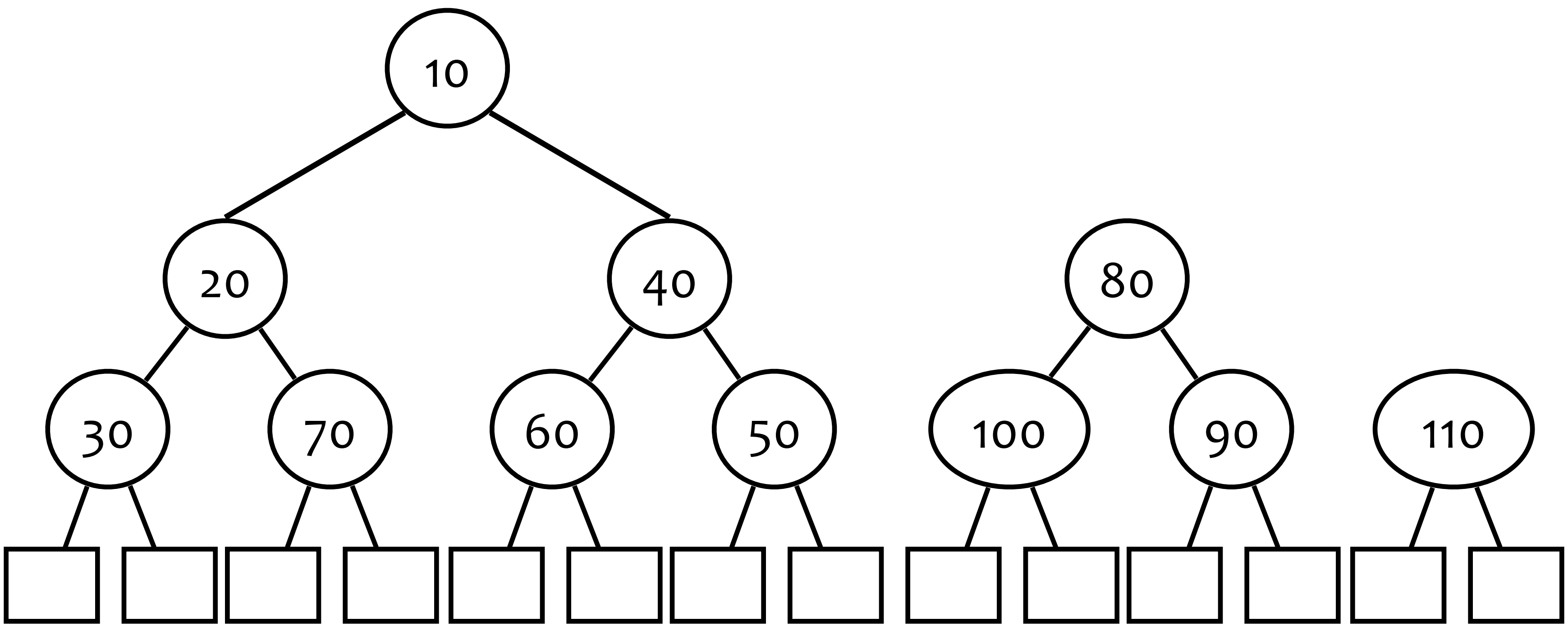
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



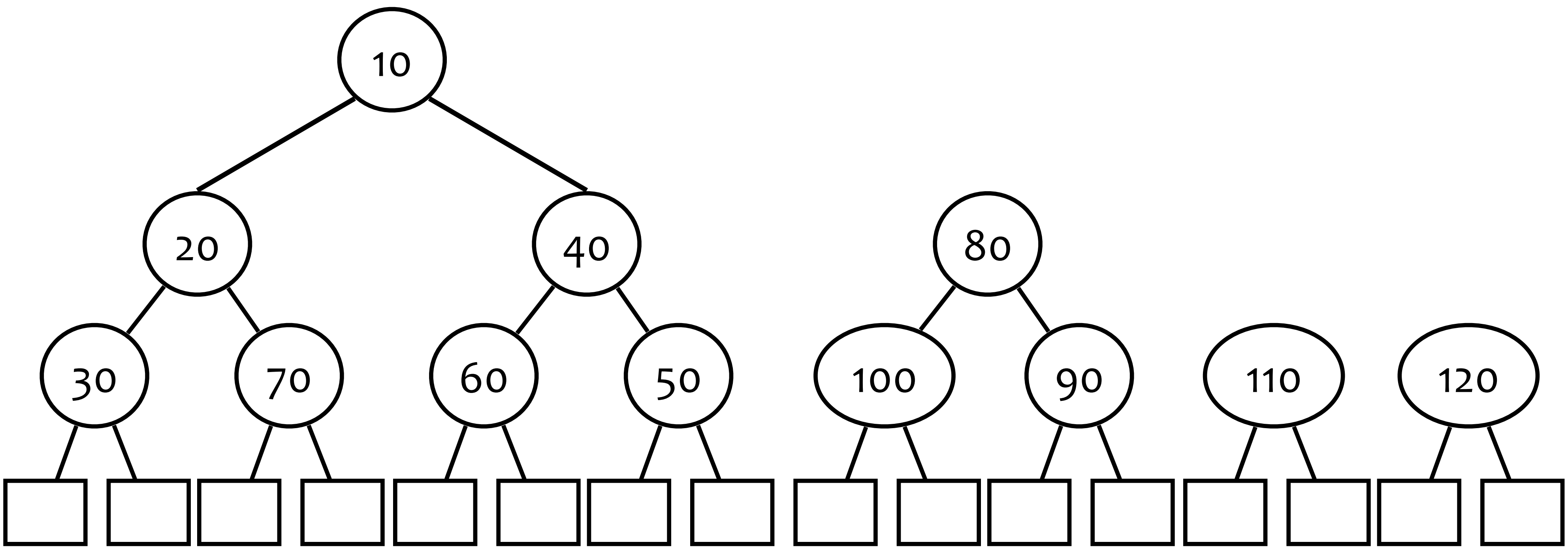
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



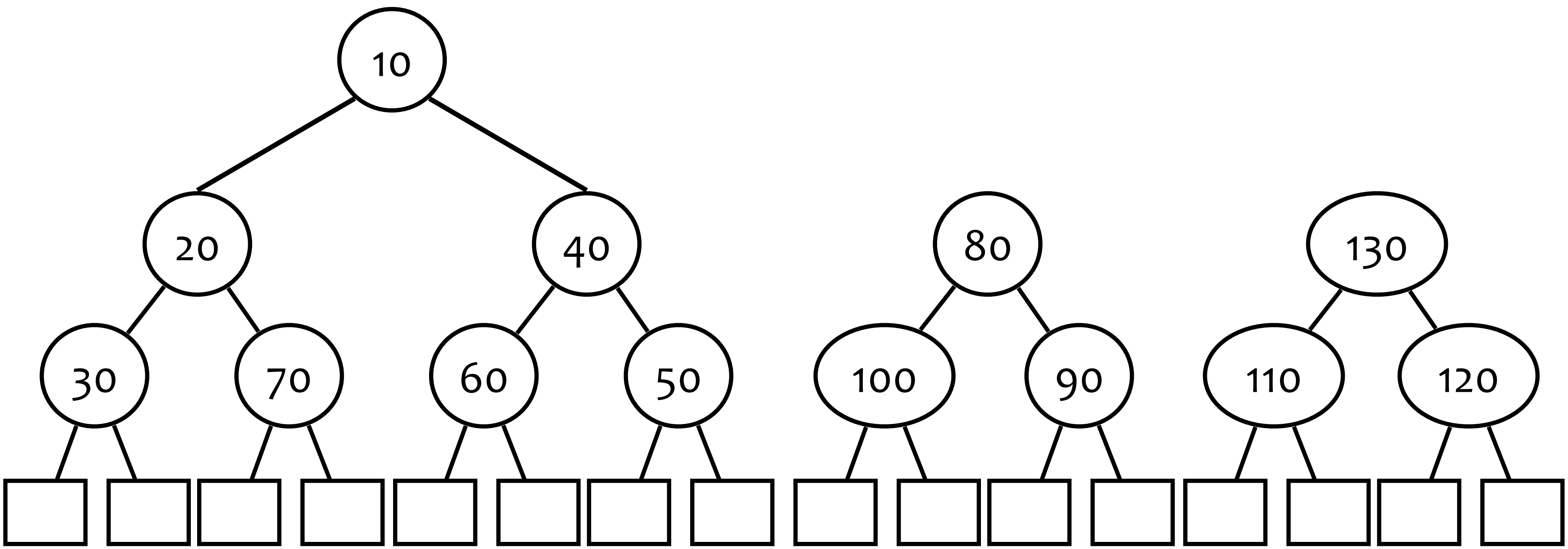
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



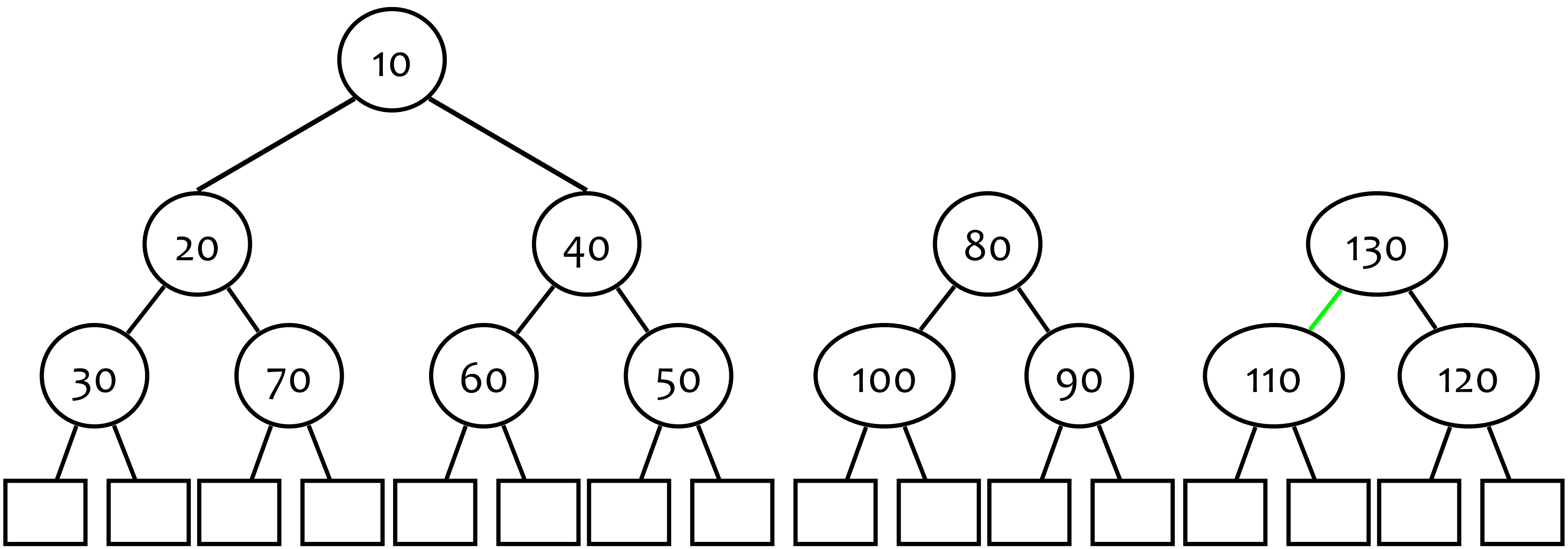
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120



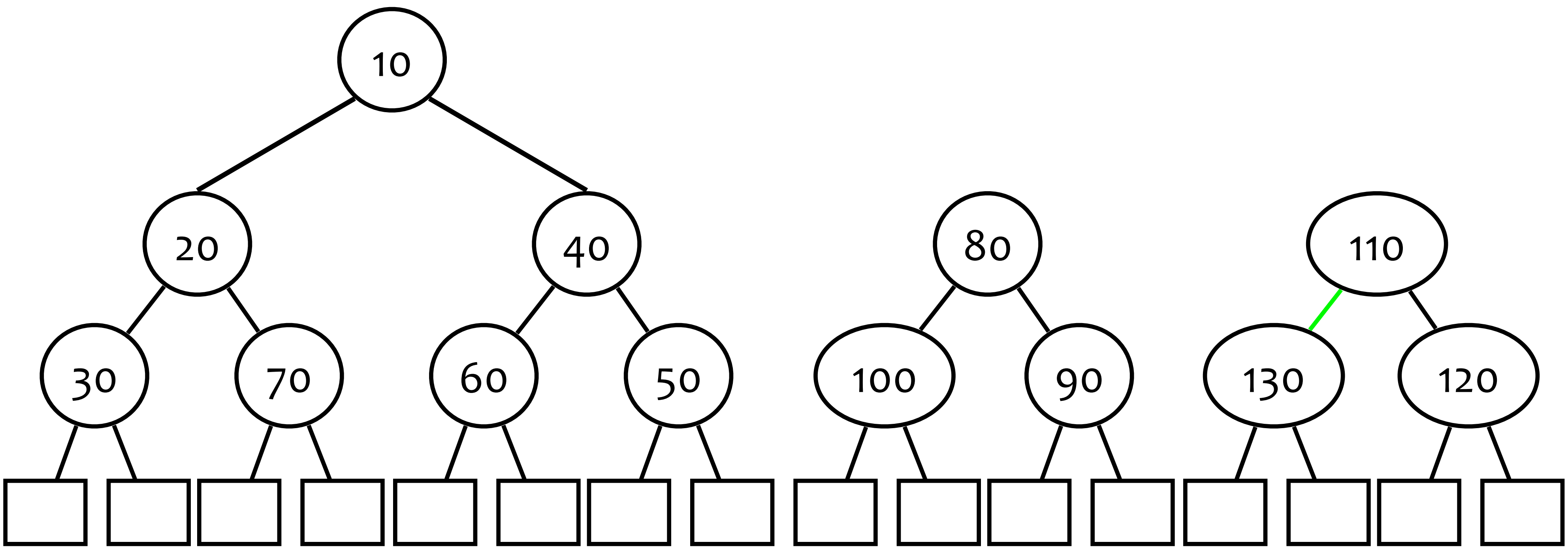
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

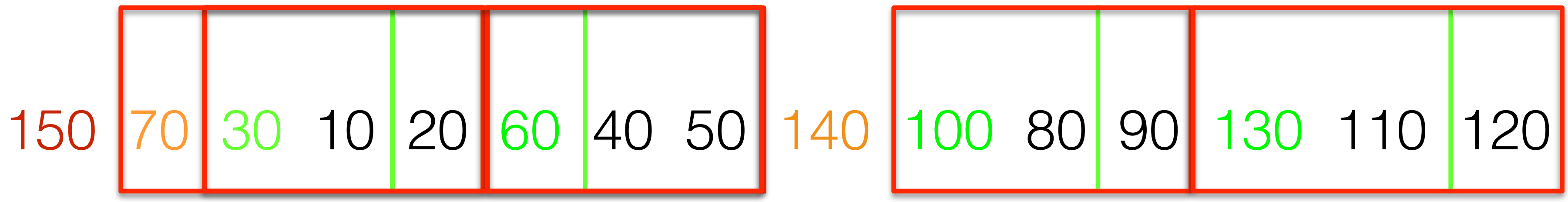


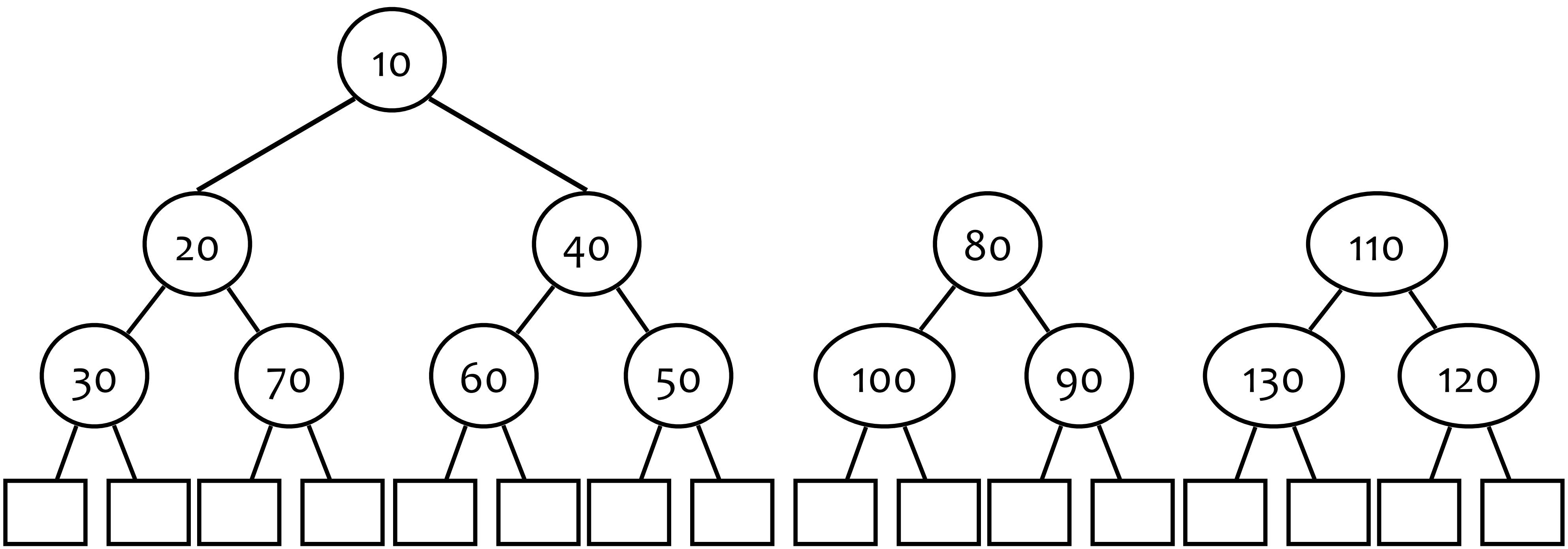
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

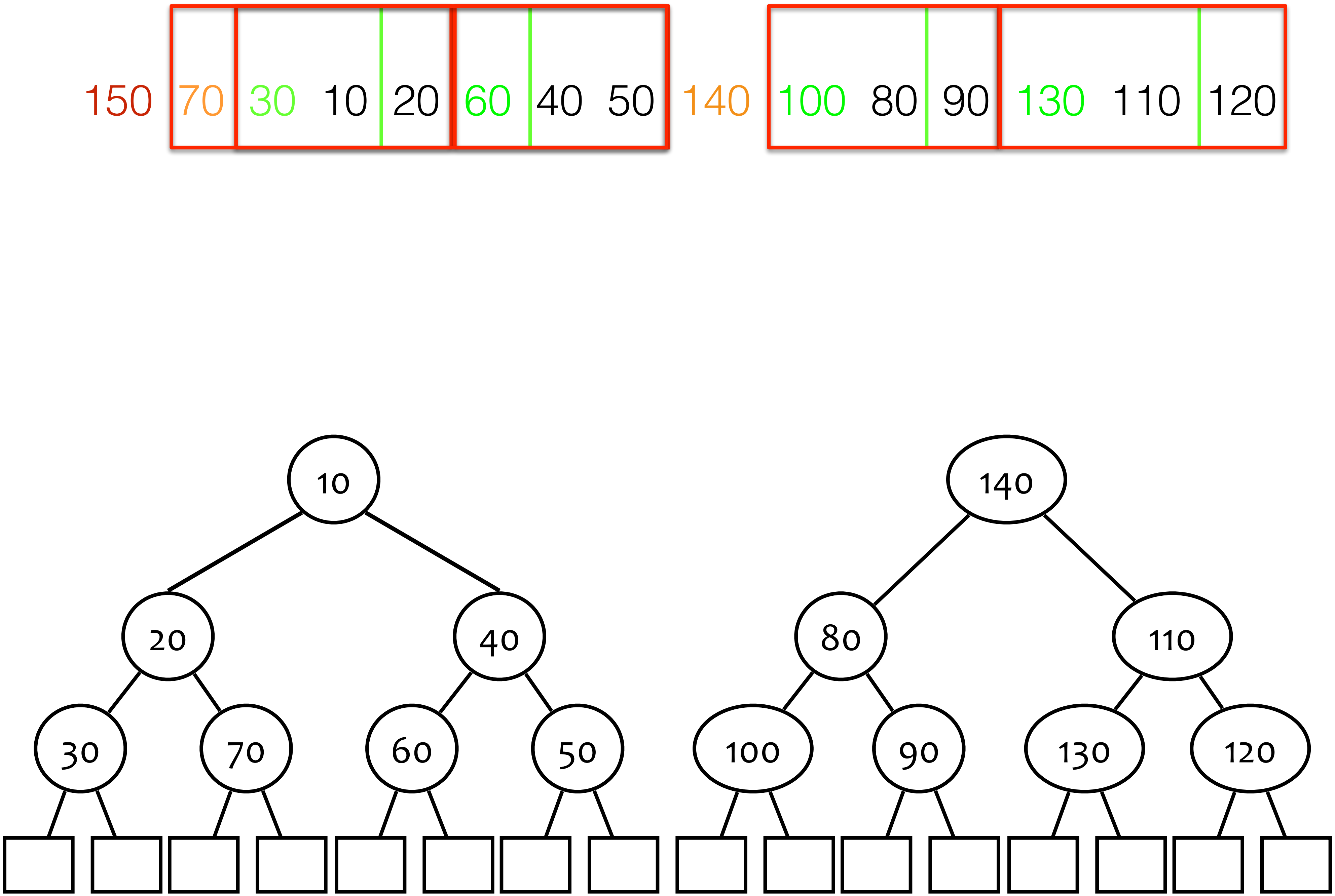


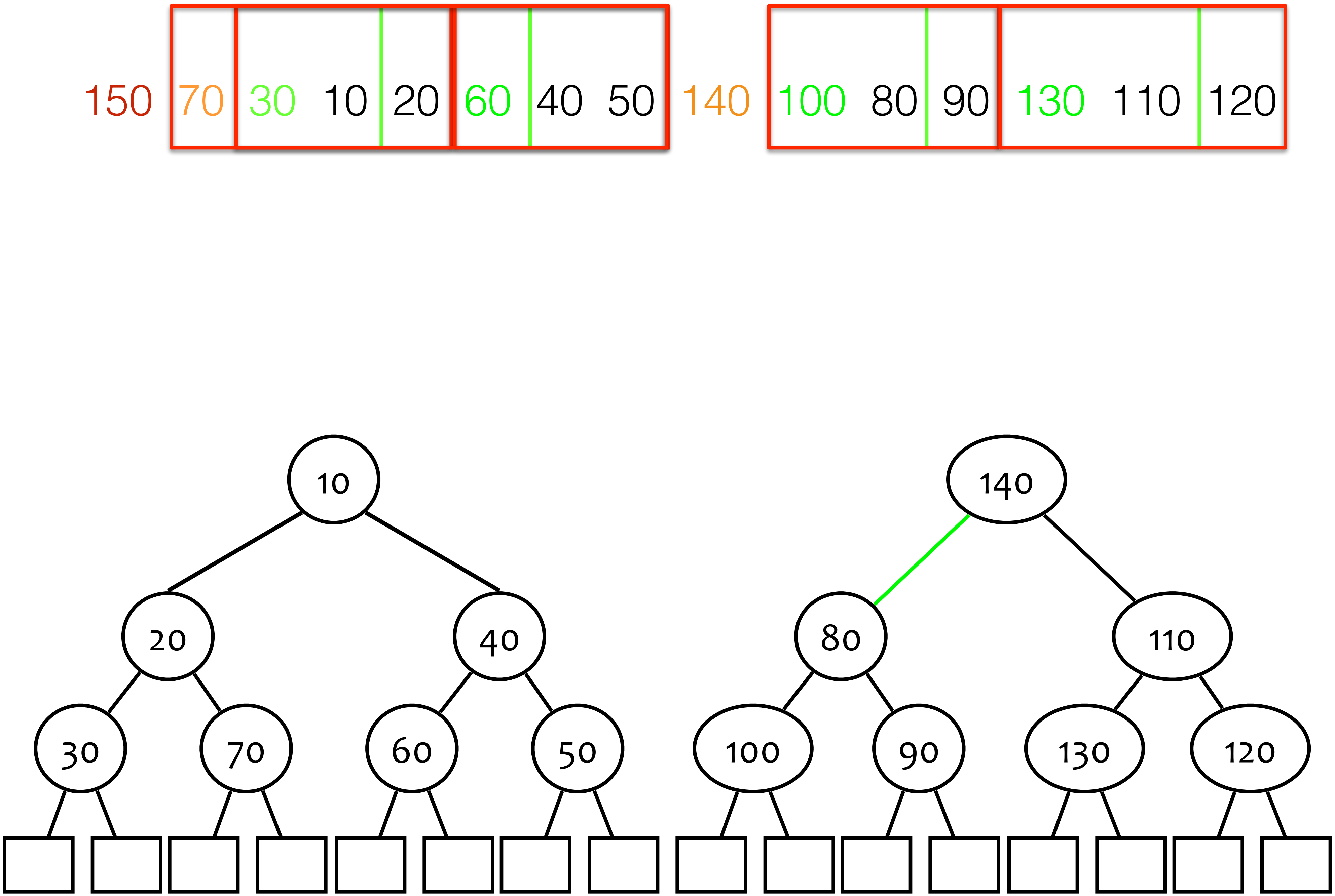
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120

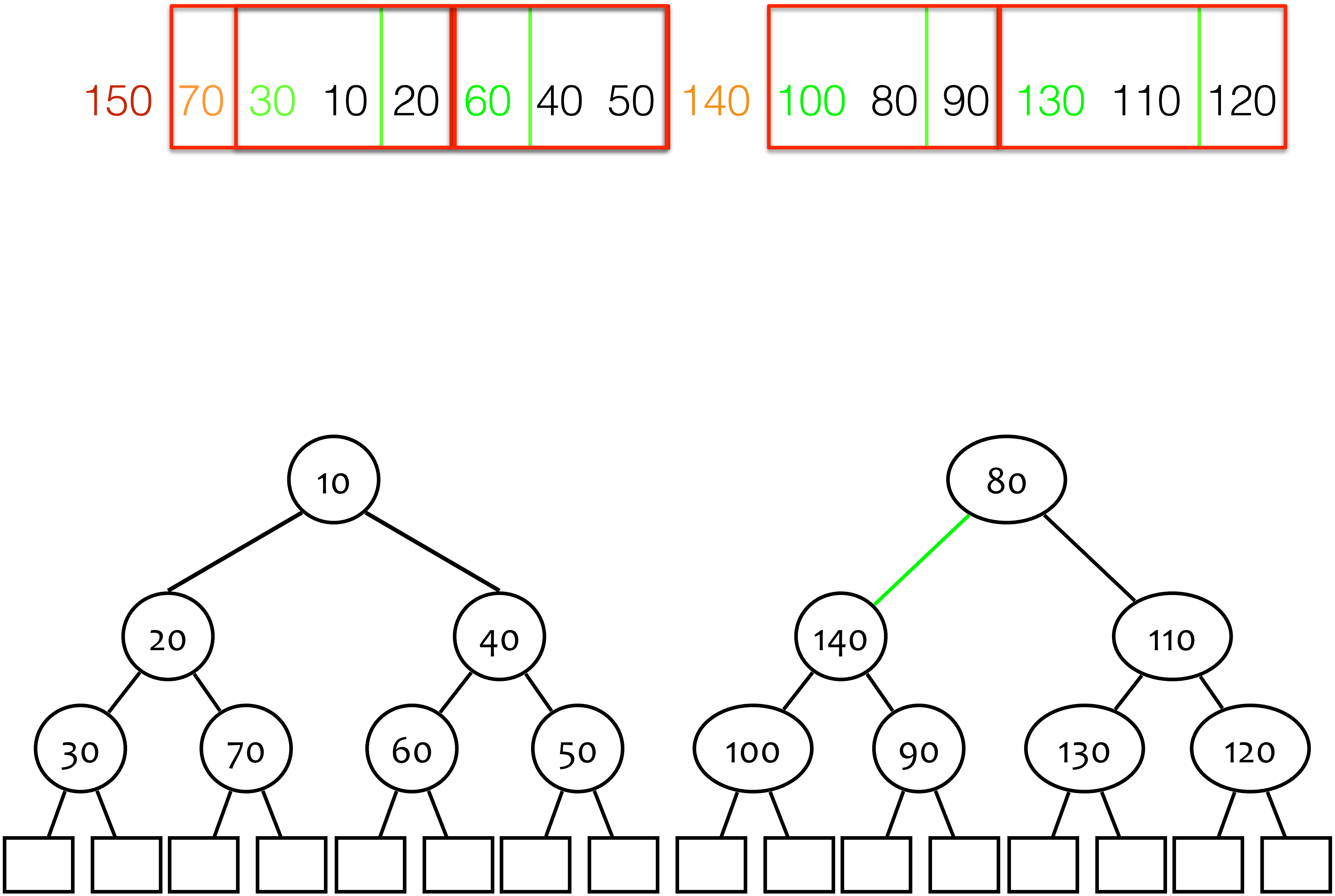


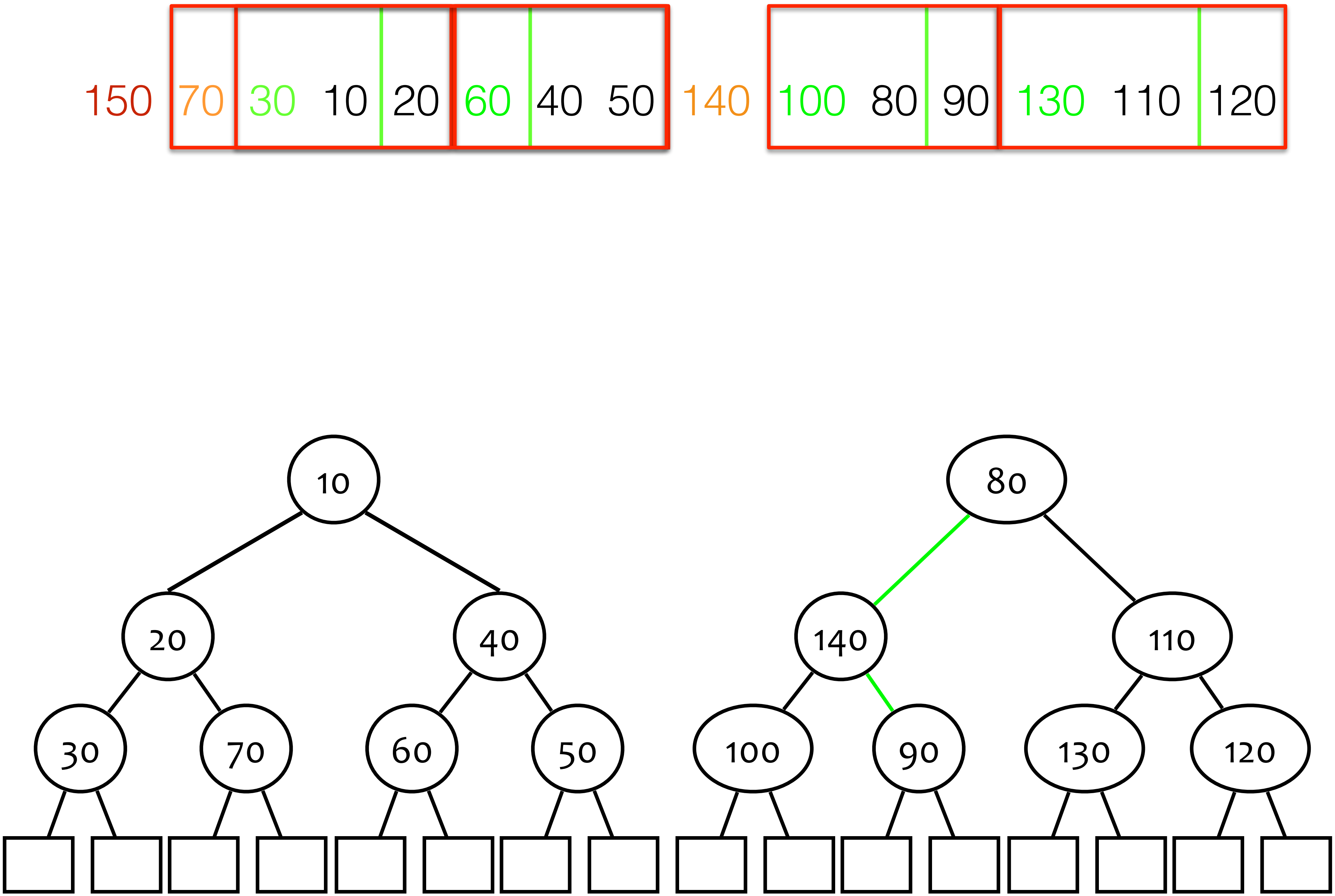

150
70
30
10
20
60
40
50
140
100
80
90
130
110
120

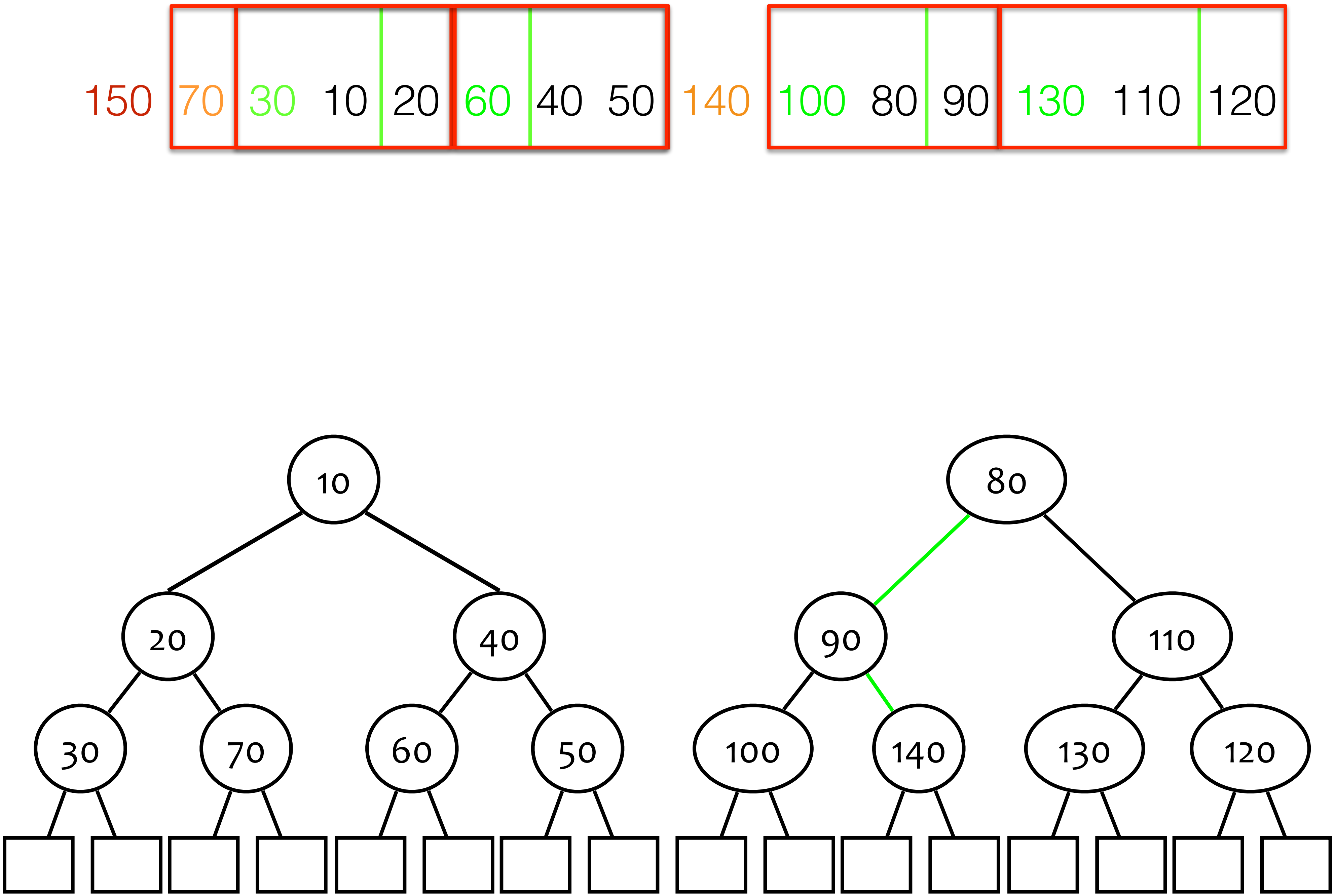

10
20
40
30
70
60
50
80
100
90
110
130
120

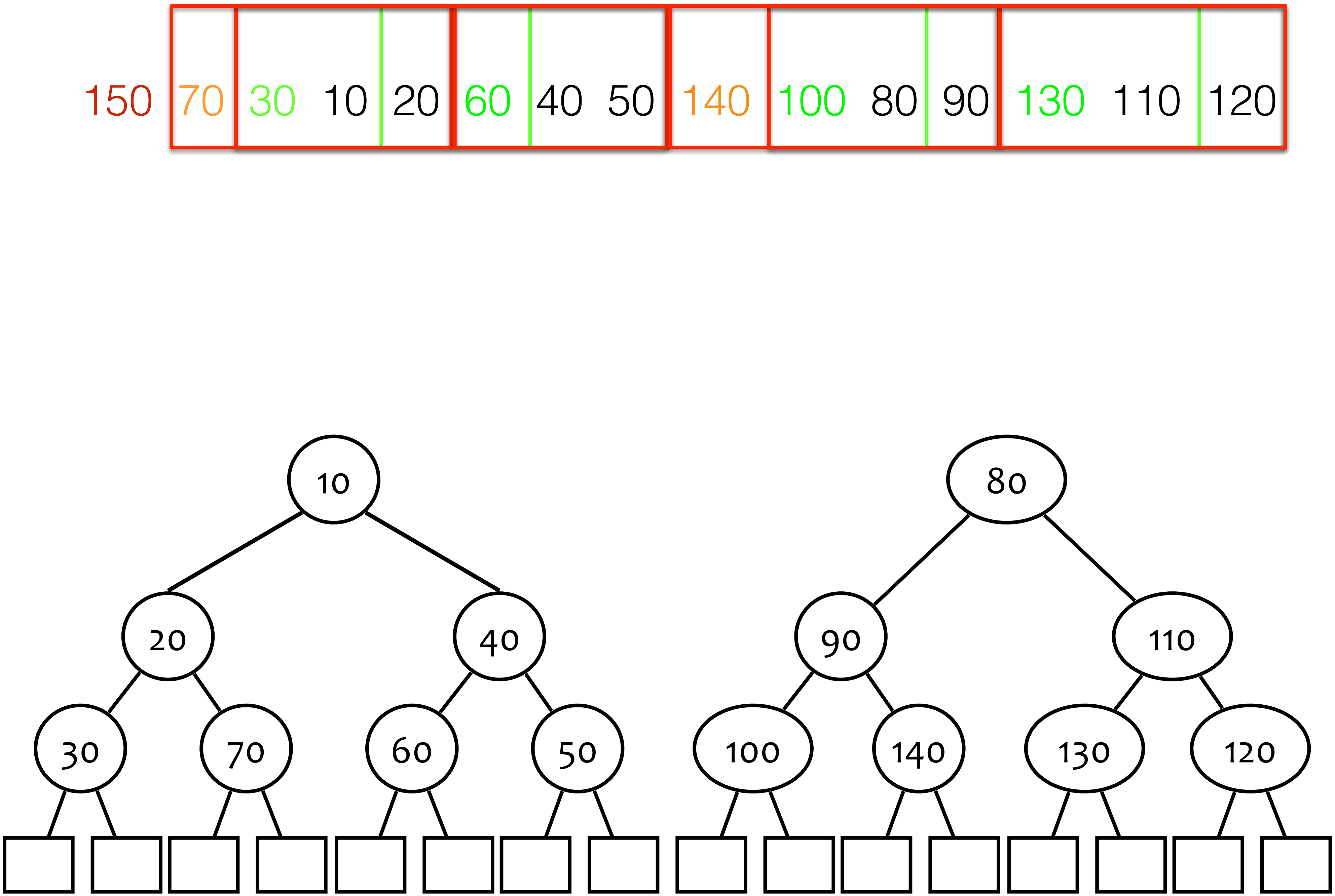

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
140
80
110
100
90
130
120

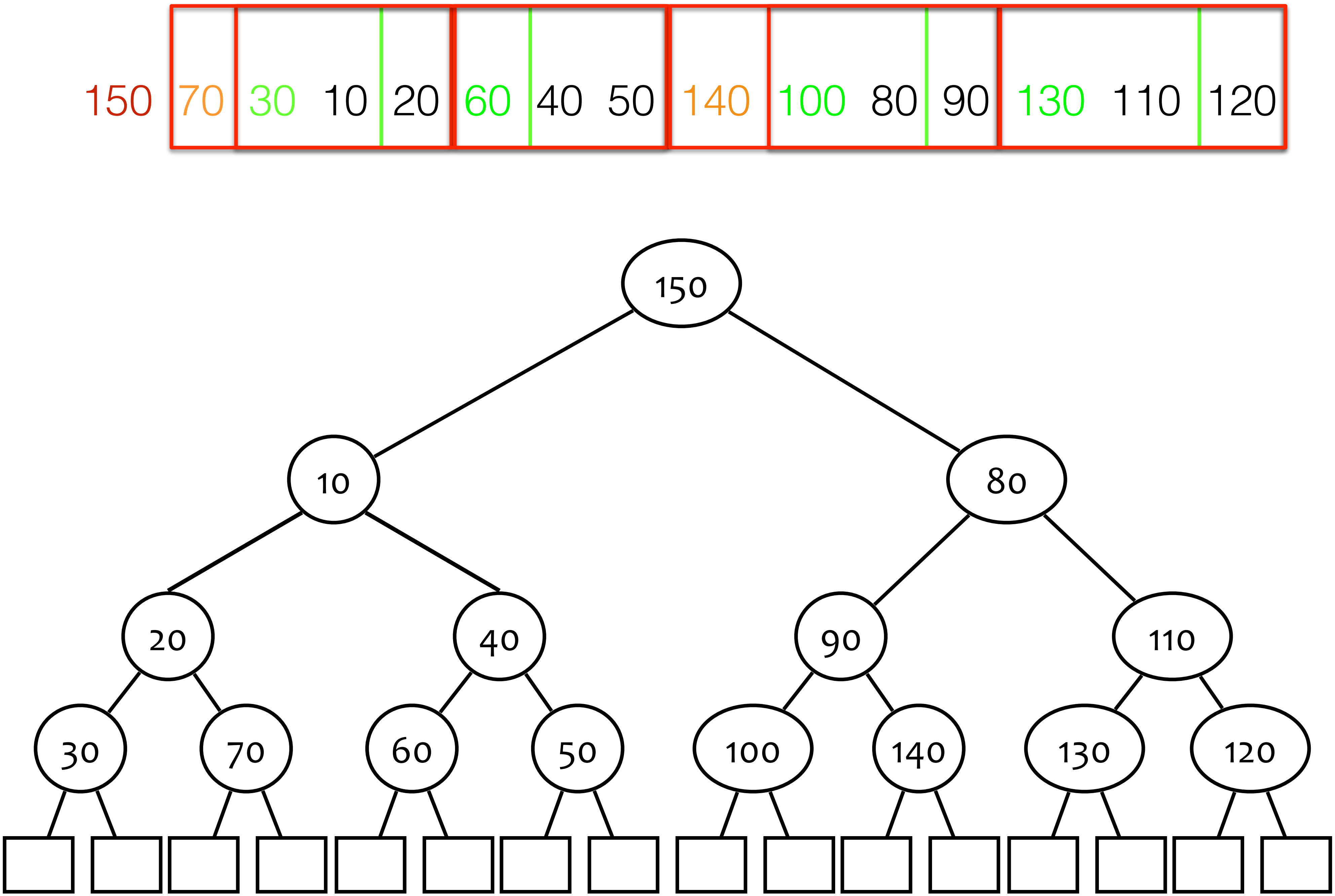

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20 40
30 70 60 50
140
80 110
100 90 130 120


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

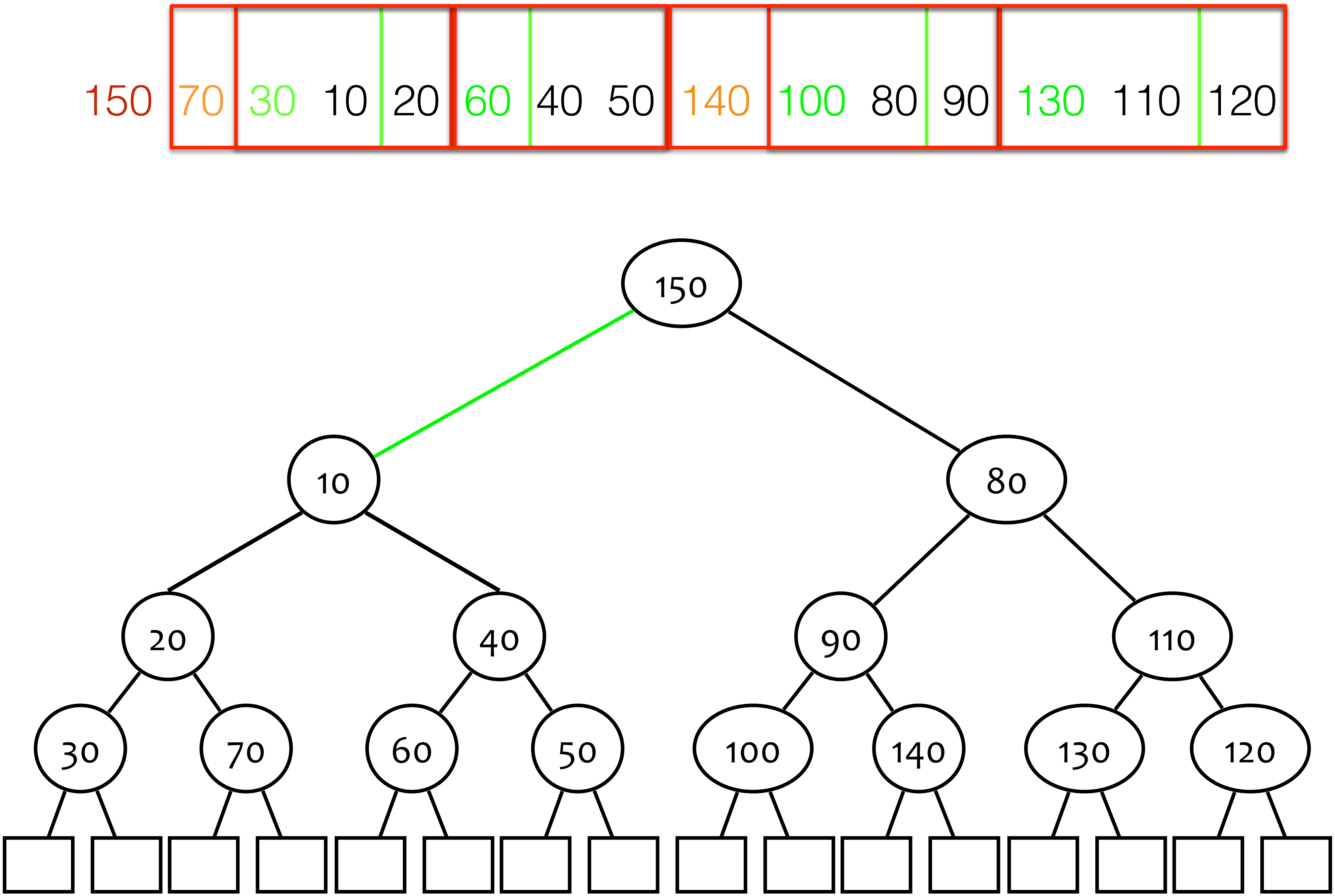

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
80
140
110
100
90
130
120

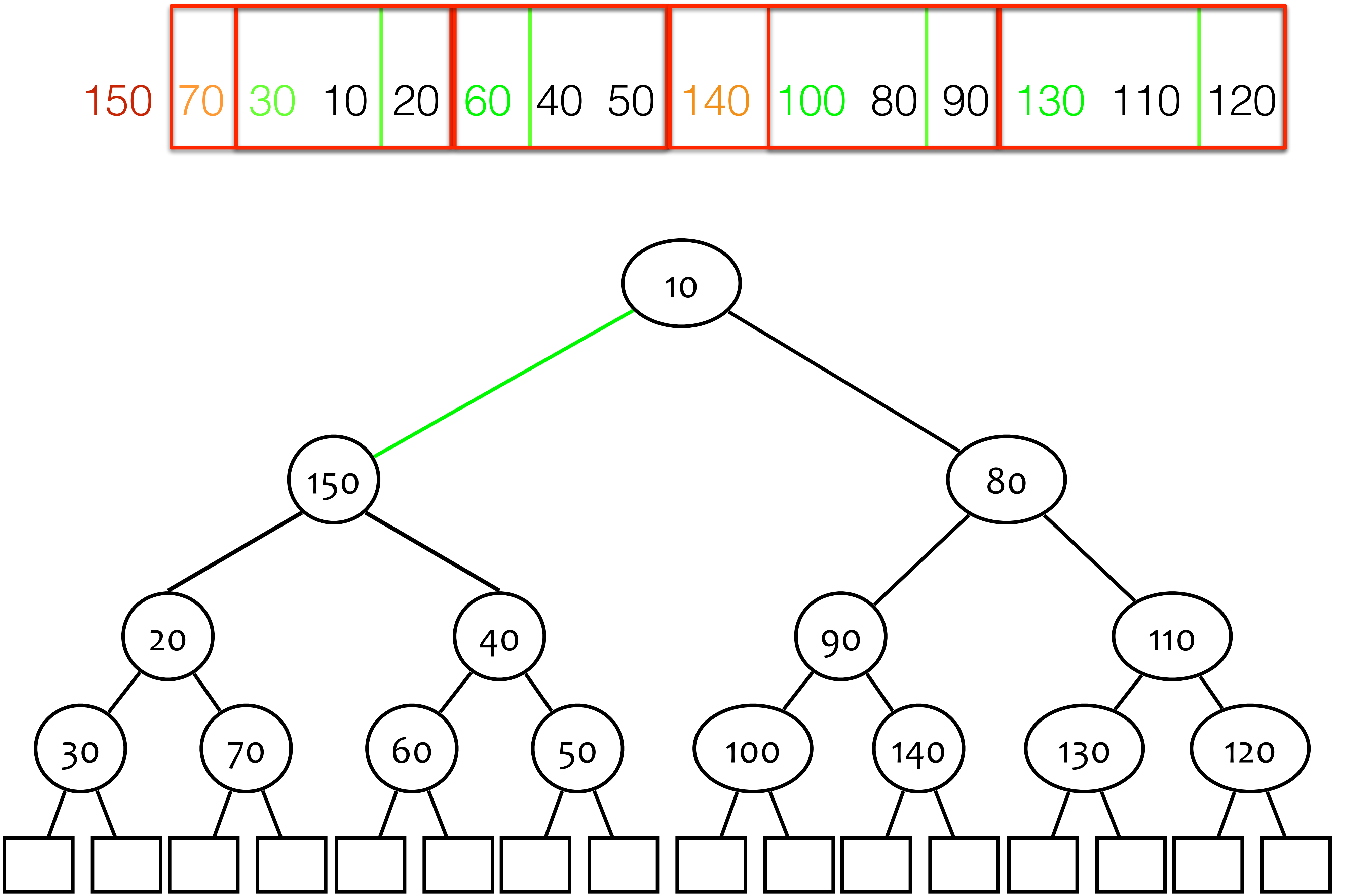

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
40
30
70
60
50
80
90
110
100
140
130
120

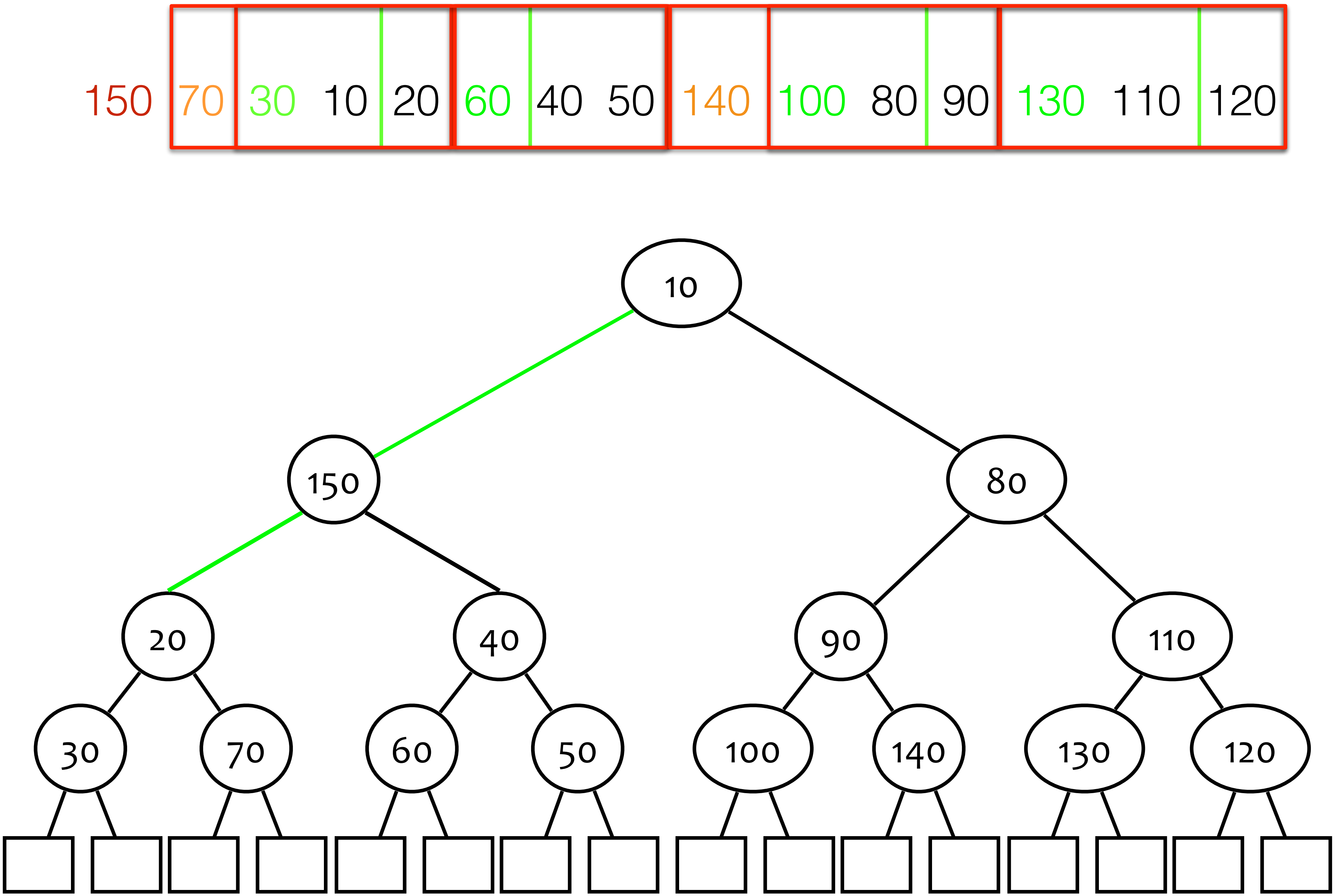

150
70
30 10
20
60
40 50
140
100 80
90
130 110
120
10
20
40
30
70
60
50
80
90
110
100
140
130
120

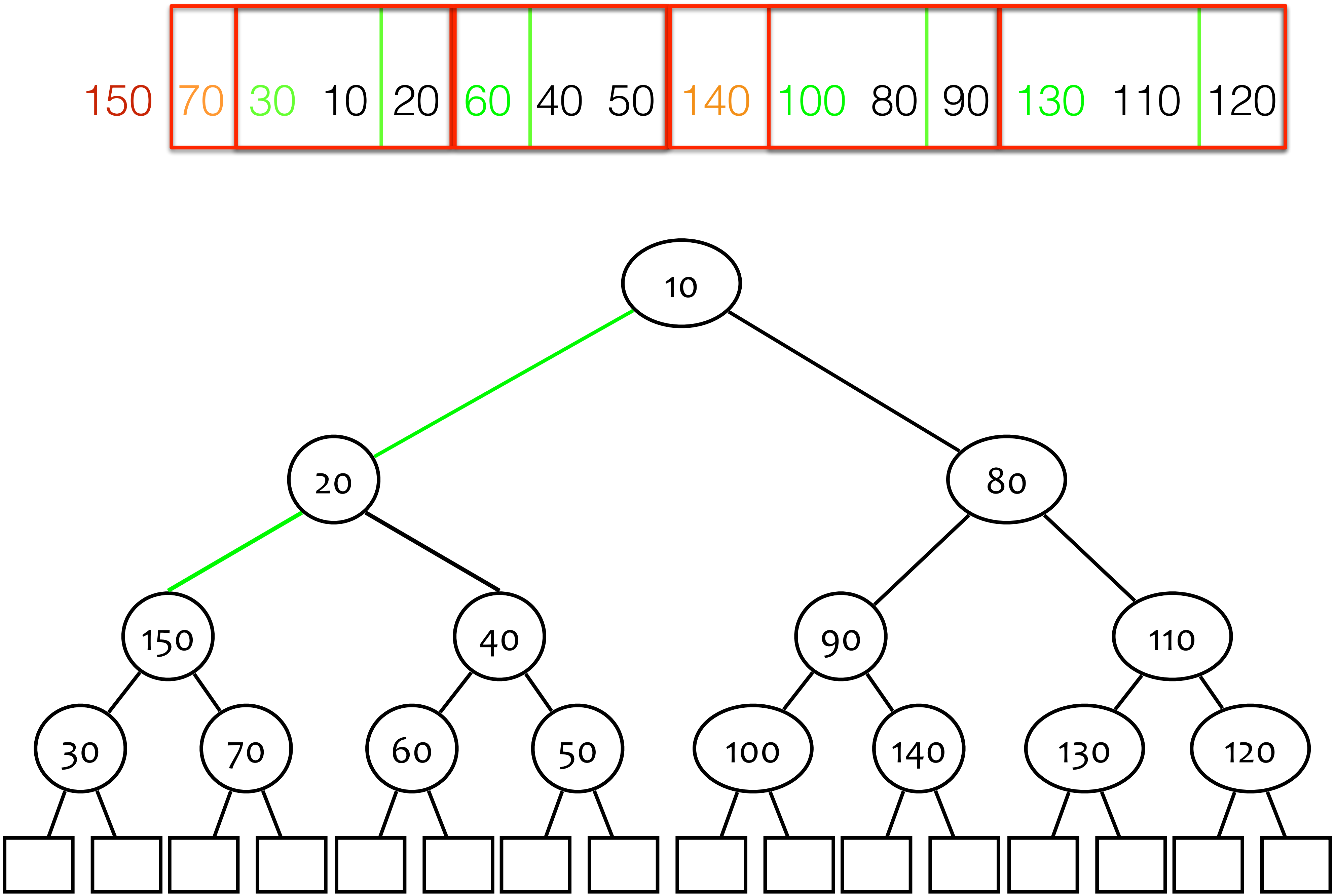


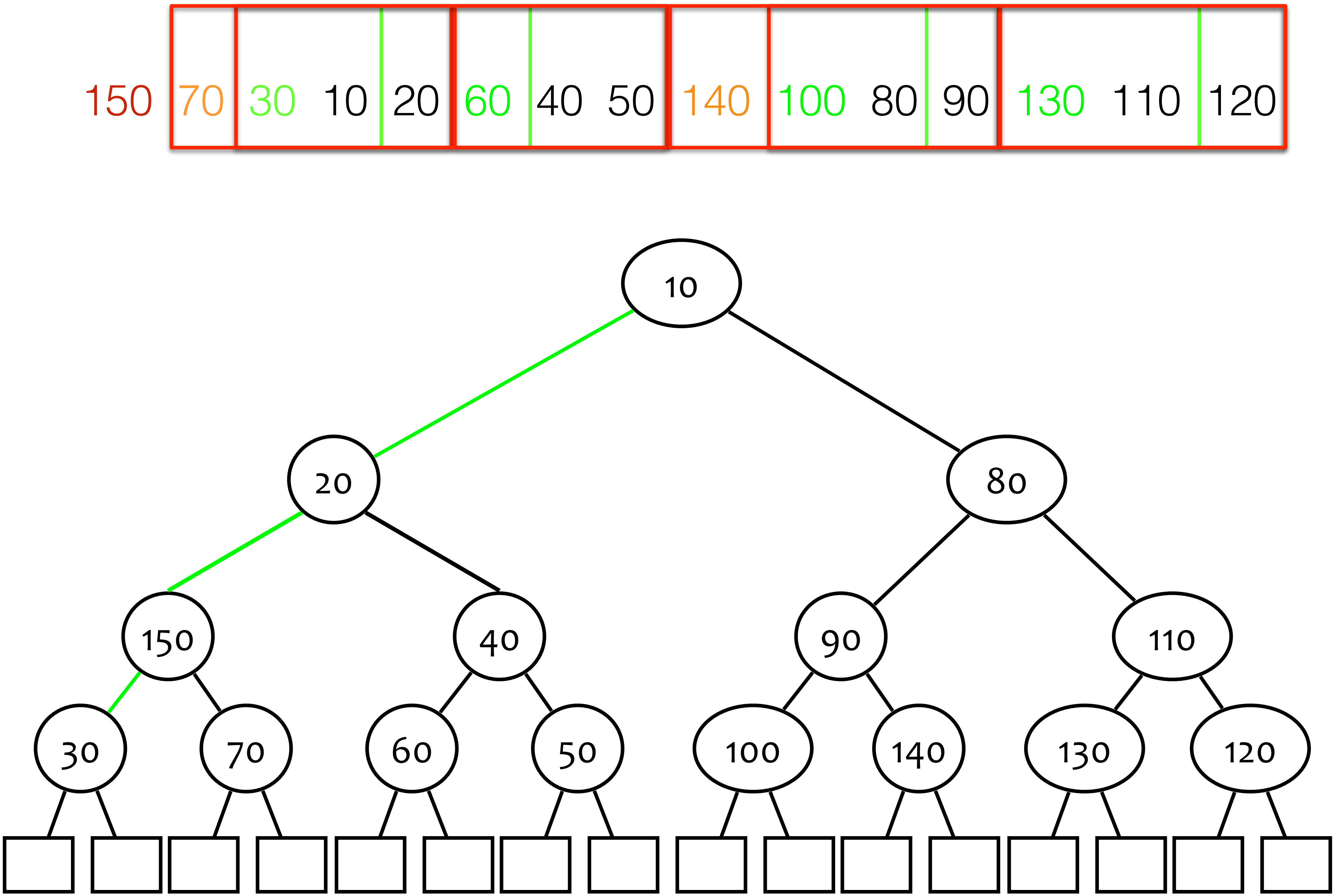
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
150
10
80
20
40
90
110
30
70
60
50
100
140
130
120


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

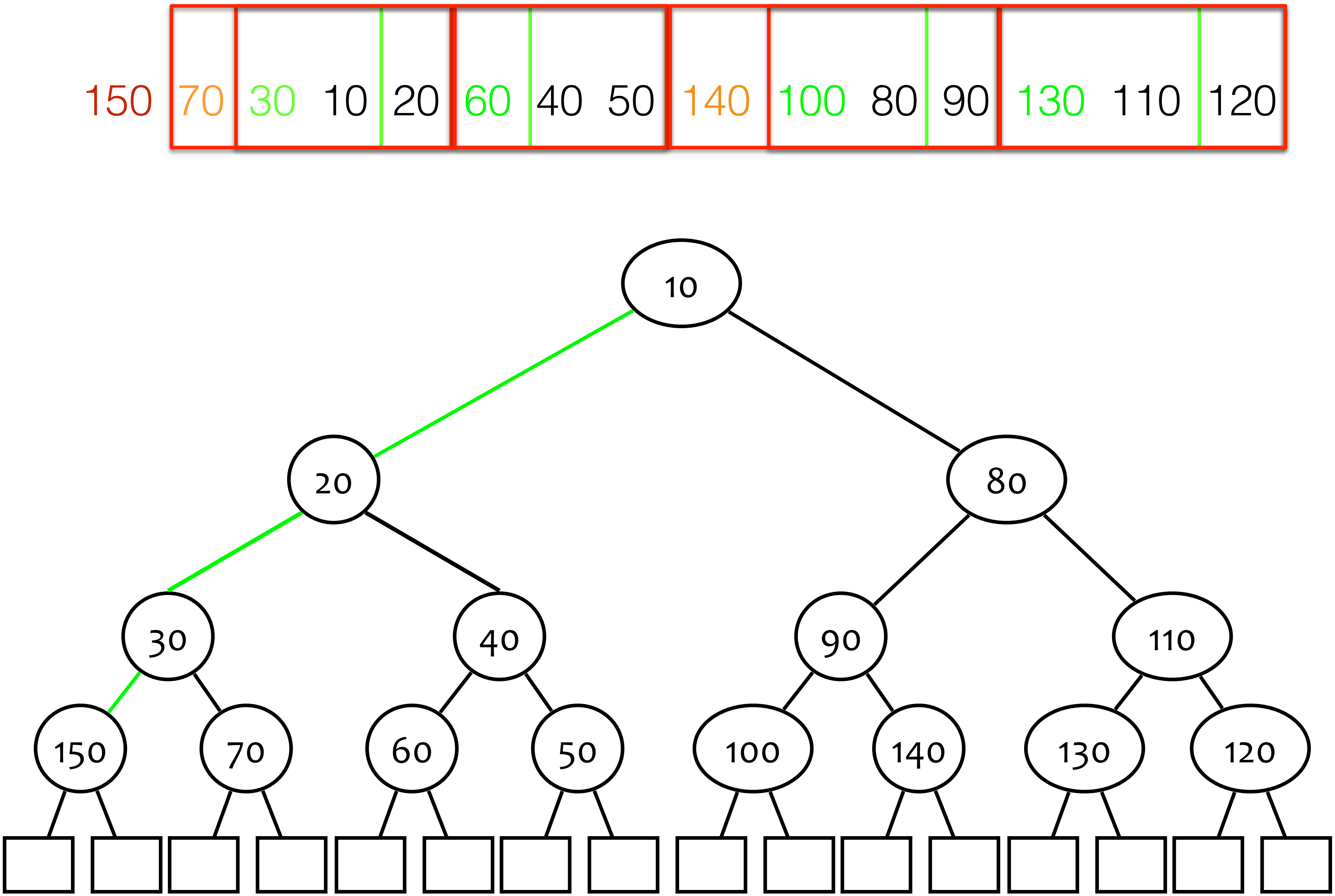

150
70
30
10
20
60
40
50
140
100
80
90
130
110
120
10
150
80
20
40
90
110
30
70
60
50
100
140
130
120

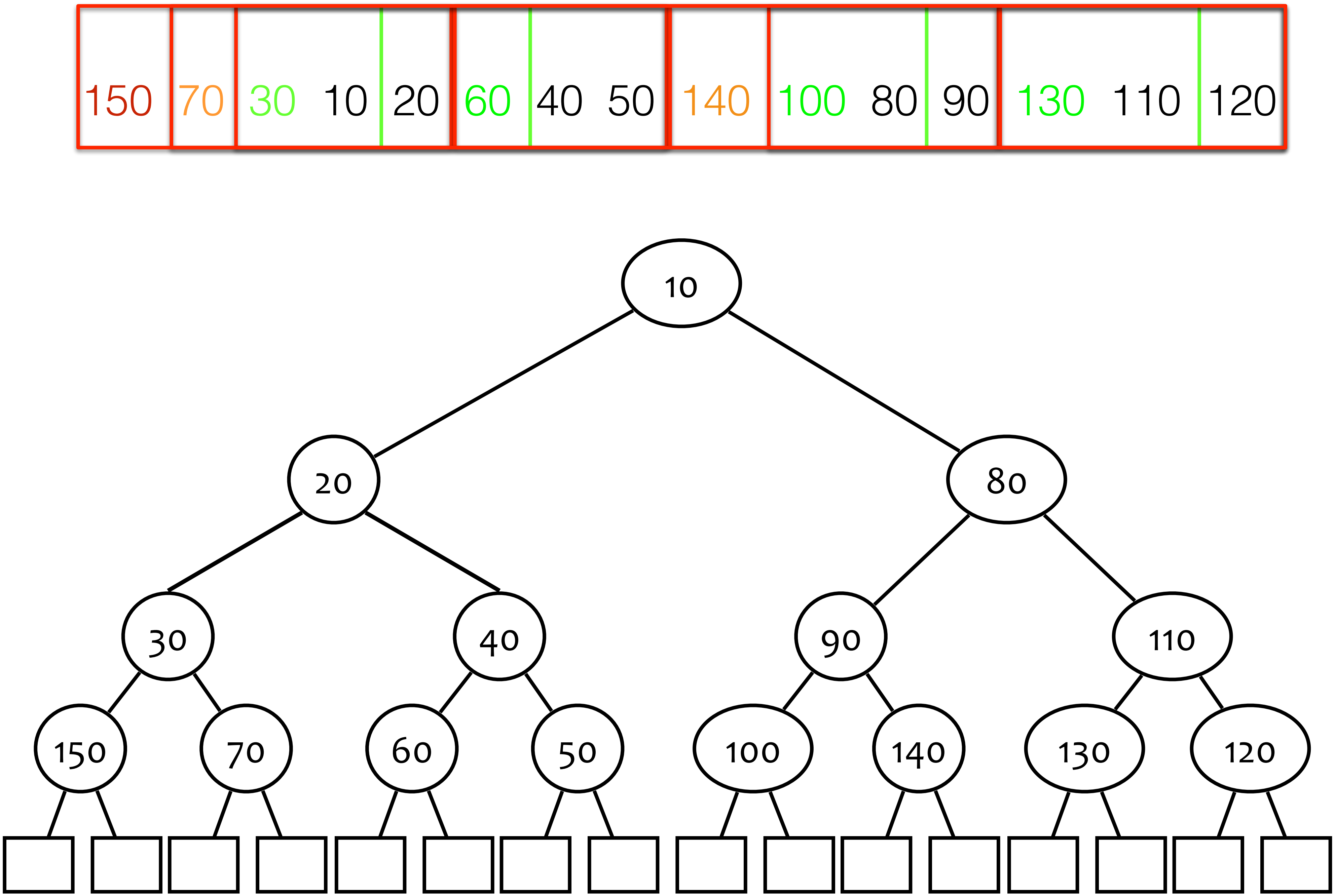

150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
150
80
20
40
90
110
30
70
60
50
100
140
130
120


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



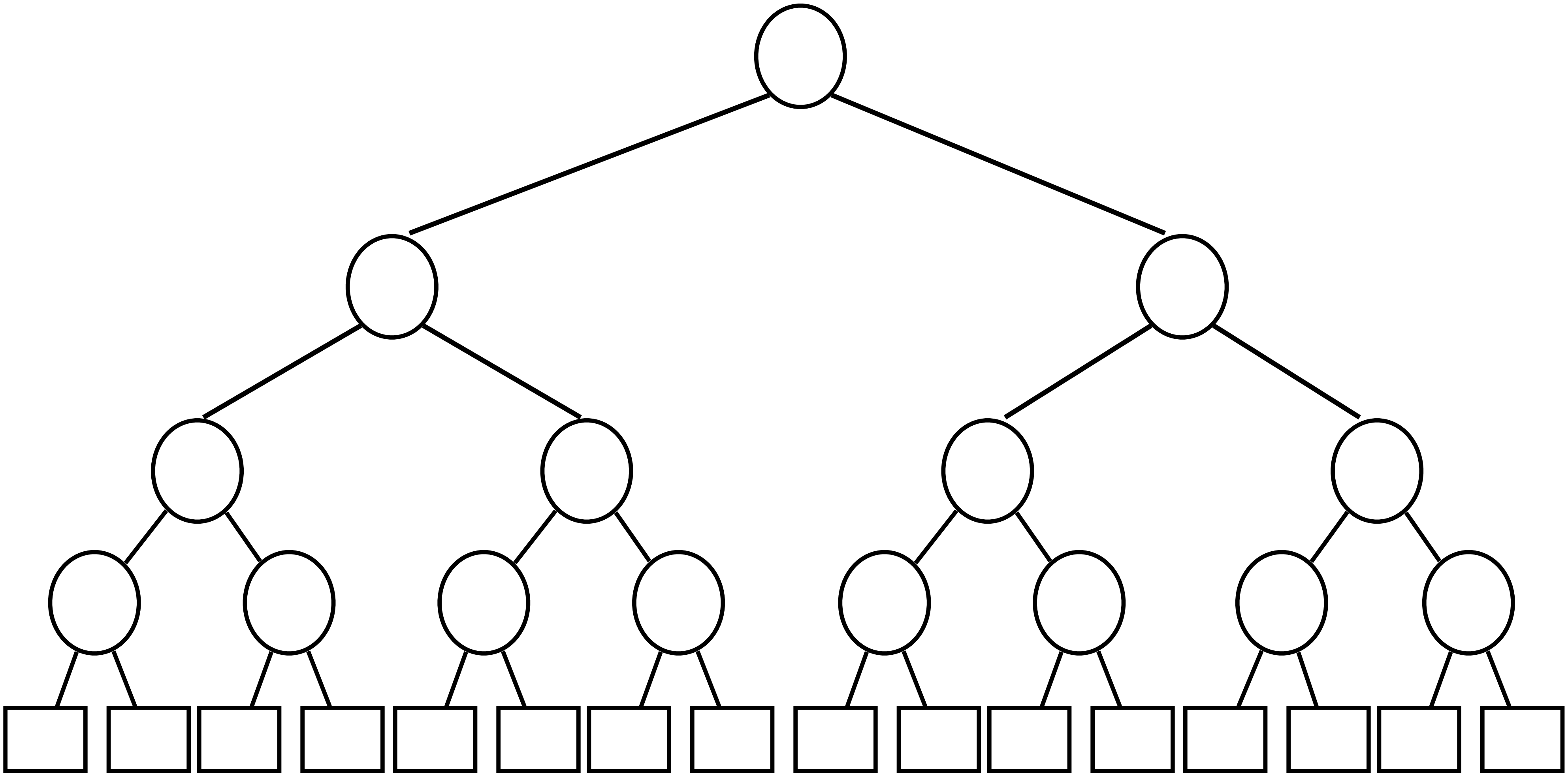
150 70 30 10 20 60 40 50 140 100 80 90 130 110 120
10
20
80
150
40
90
110
30
70
60
50
100
140
130
120


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

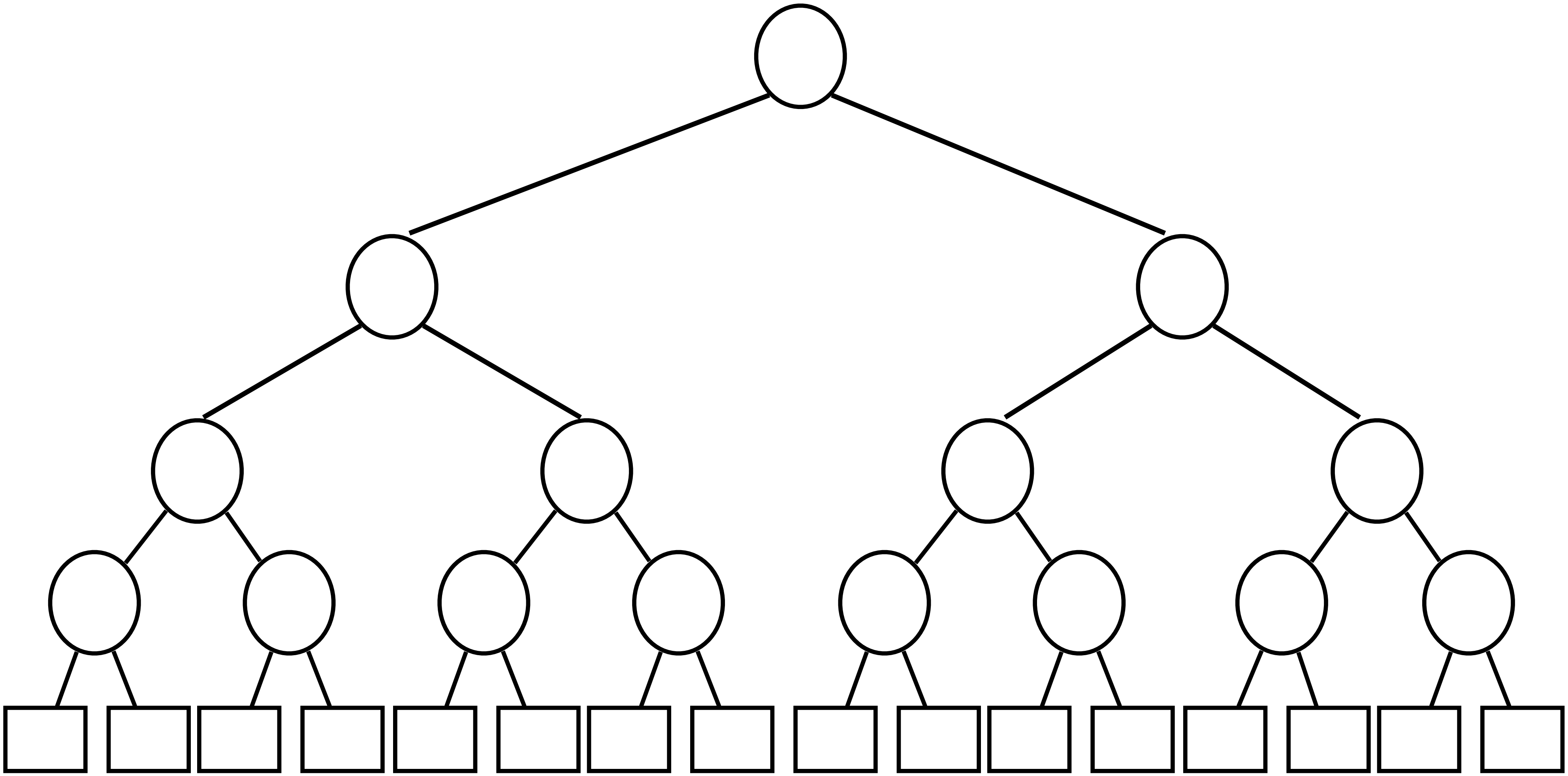


150
70
30
10
20
60
40
50
140
100
80
90
130
110
120
10
20
80
30
40
90
110
150
70
60
50
100
140
130
120

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



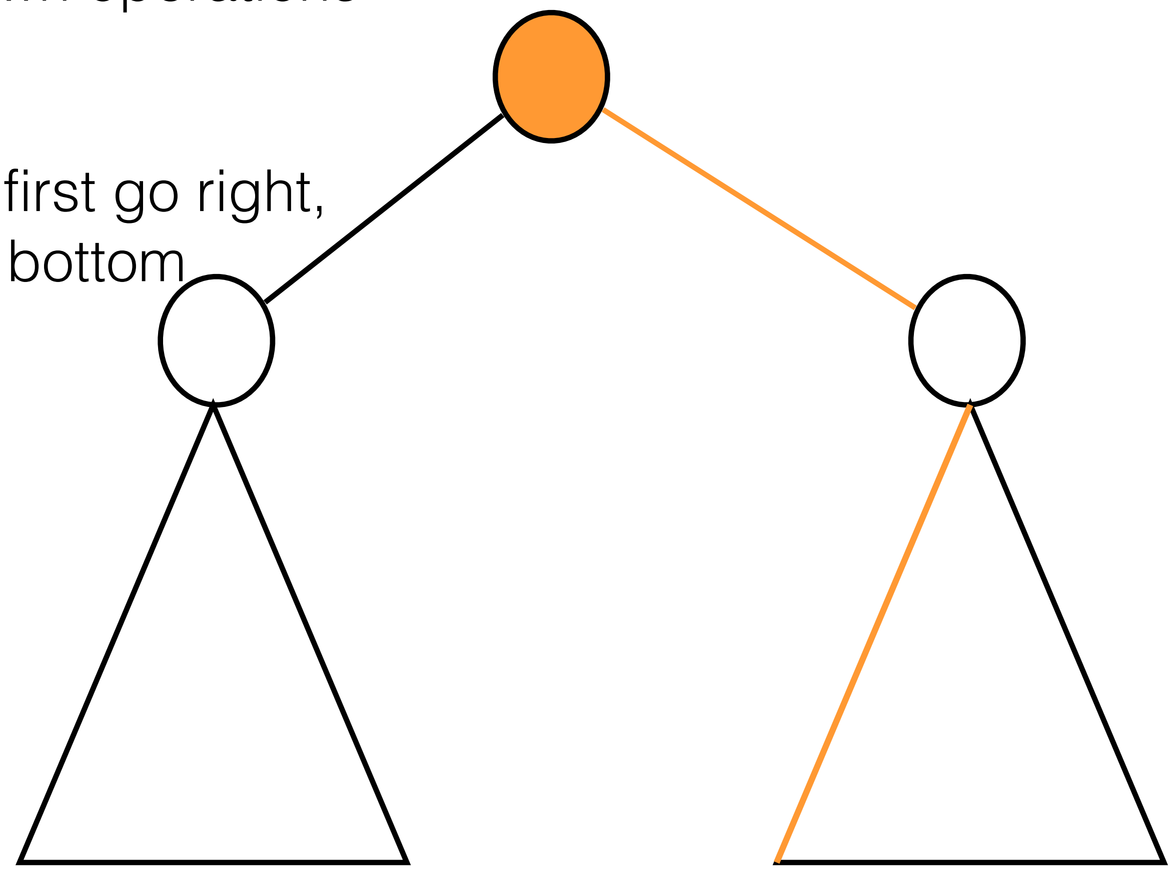
We show: max# bubble down ops $\leq$ # heap edges



Proof idea

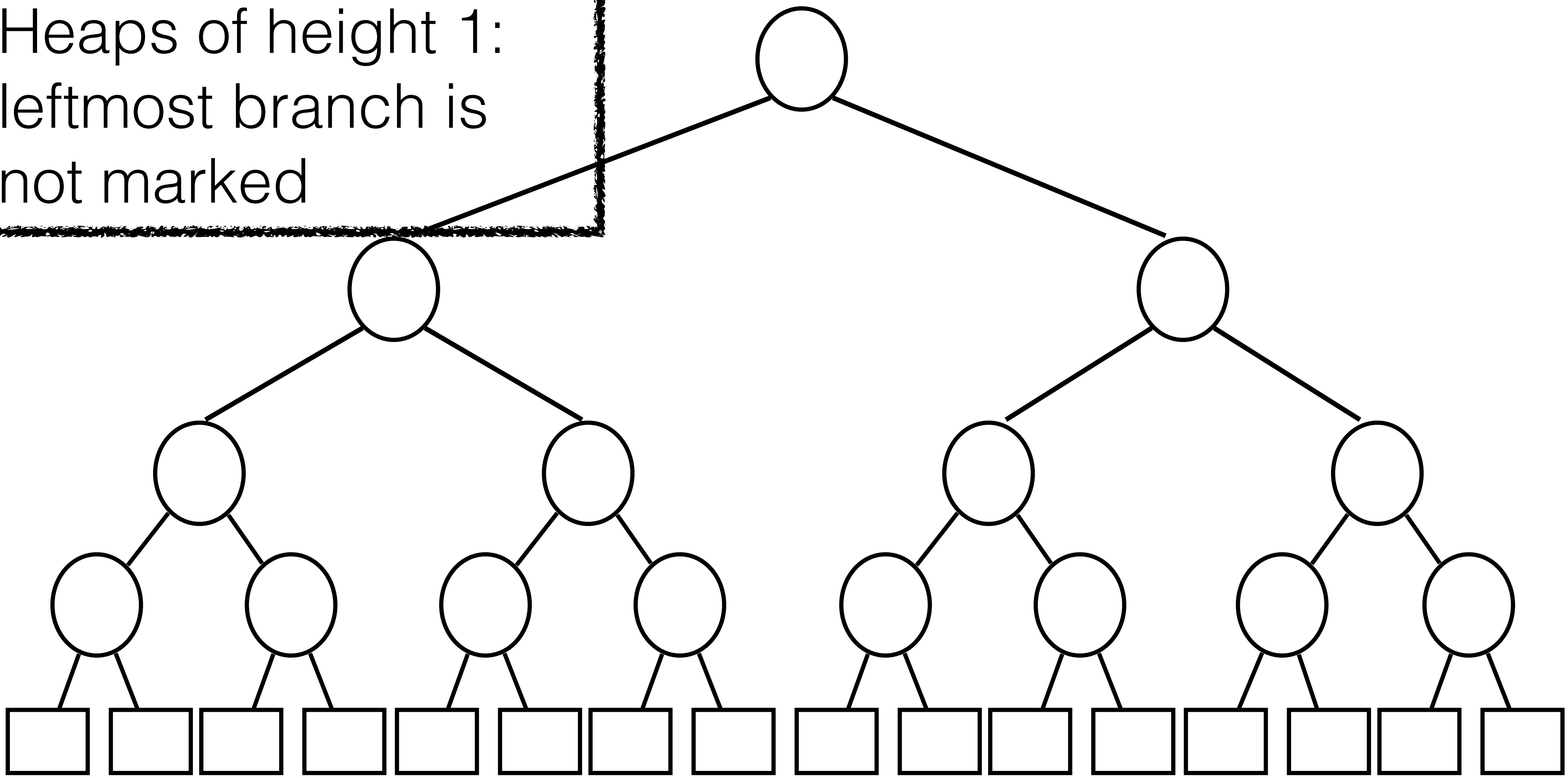
For each new node joining two heaps: mark path of maximum number of bubble-down operations

For marking: first go right, then left until bottom



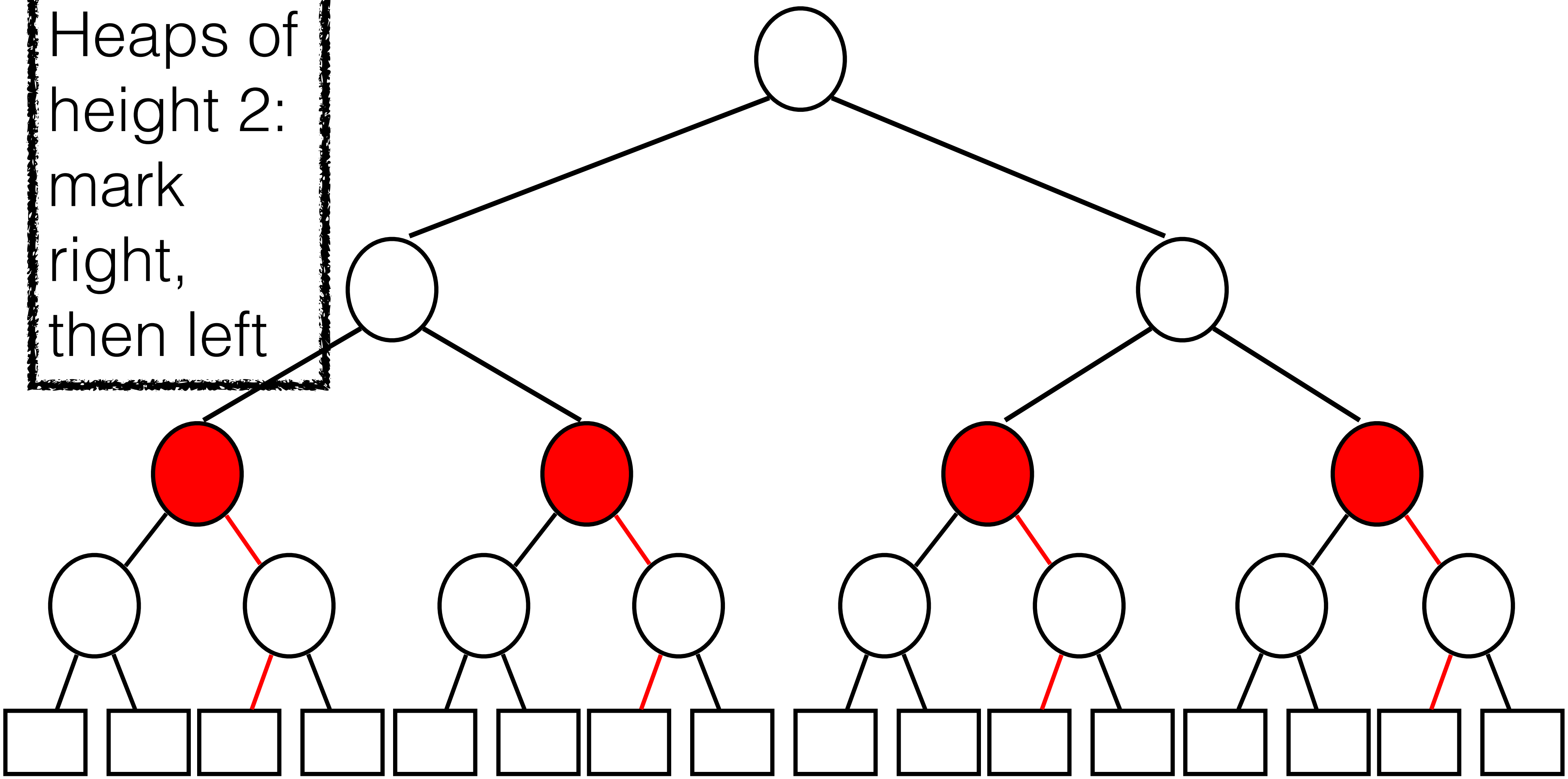
For each new node joining two heaps: mark path with maximum number of bubble-down operations

Heaps of height 1: leftmost branch is not marked

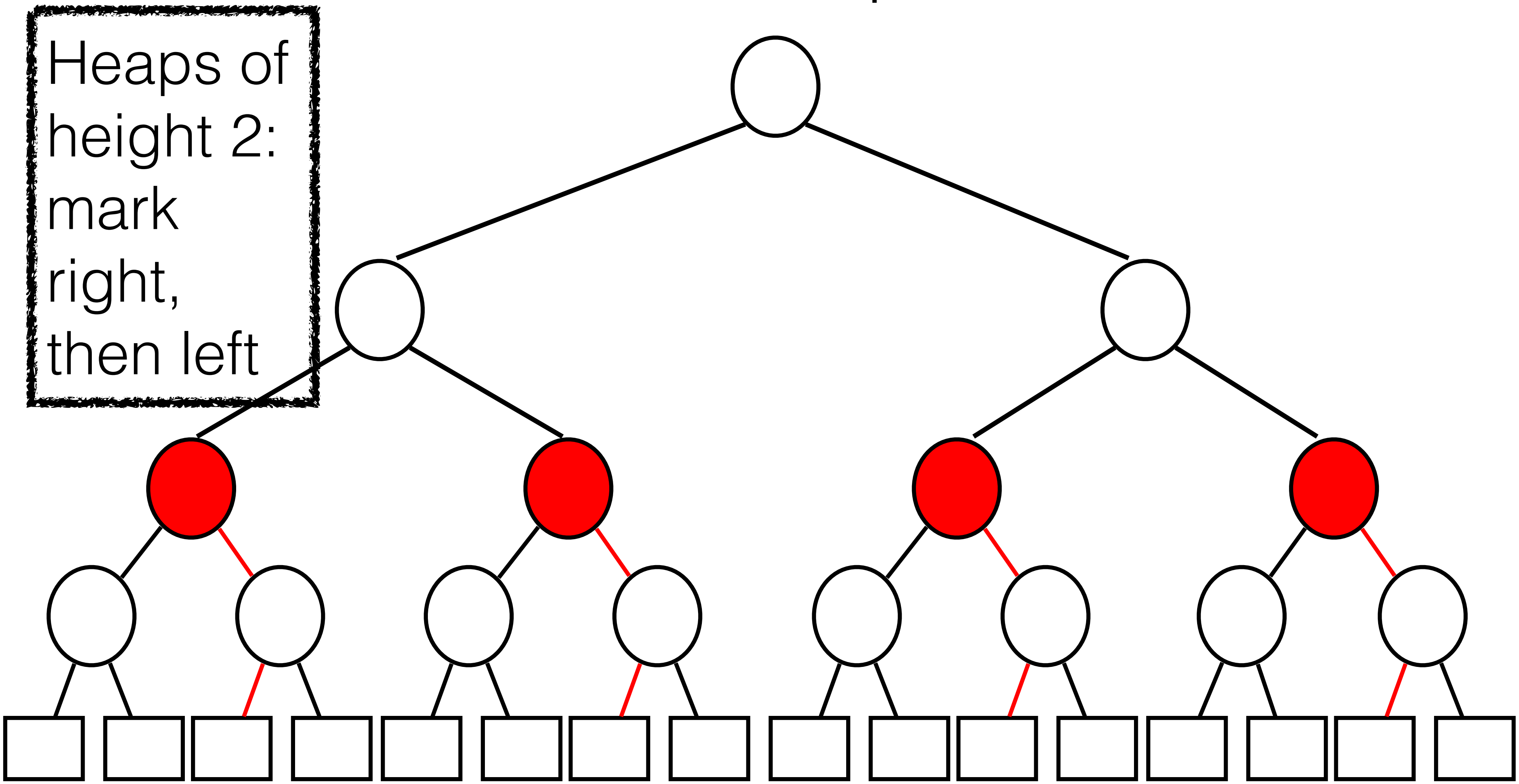


For each new node joining two heaps: mark path with maximum number of bubble-down operations

Heaps of height 2: mark right, then left



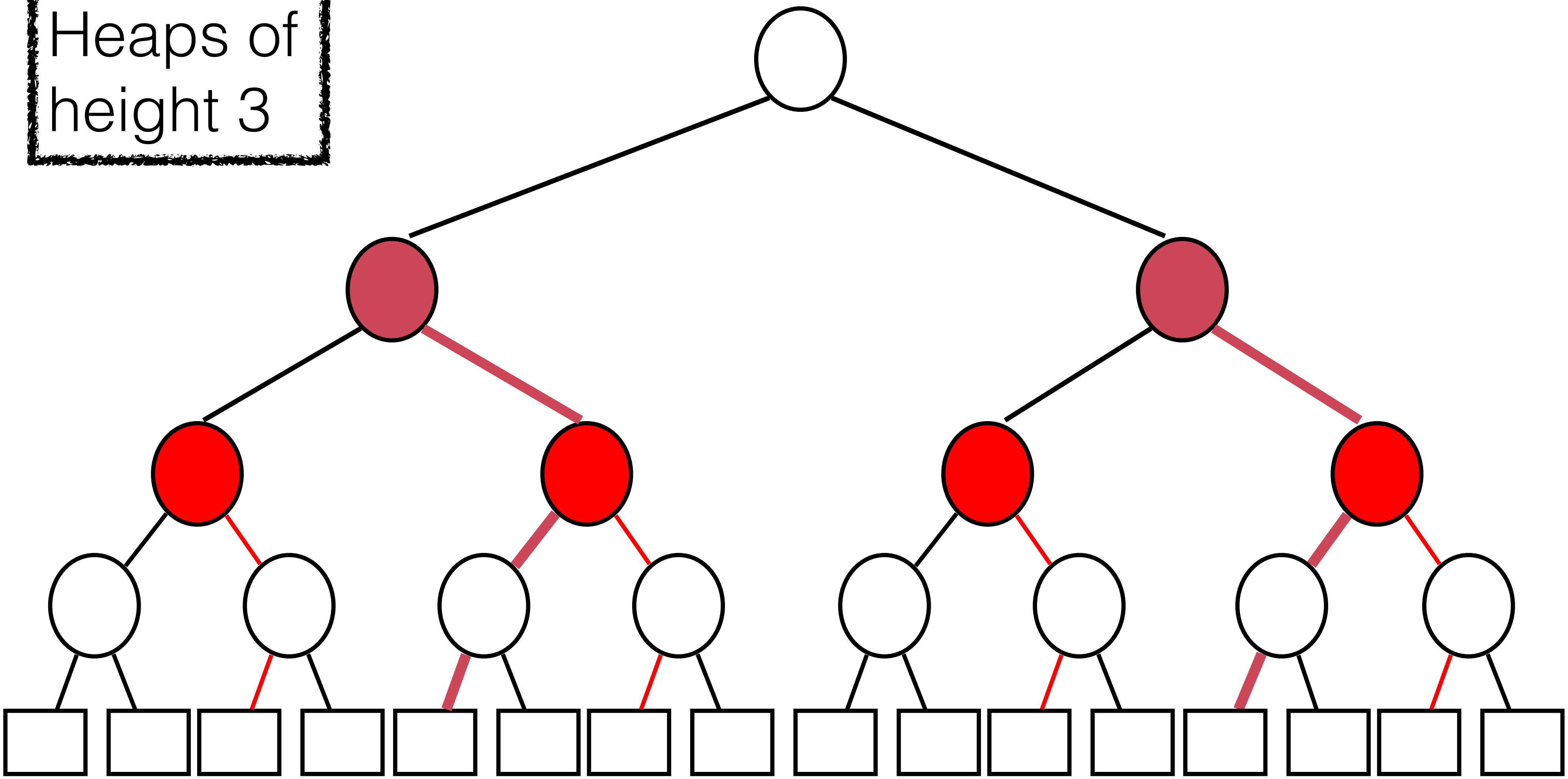
For each new node joining two heaps: mark path with maximum number of bubble-down operations



For each height-2 heap, leftmost branch not marked

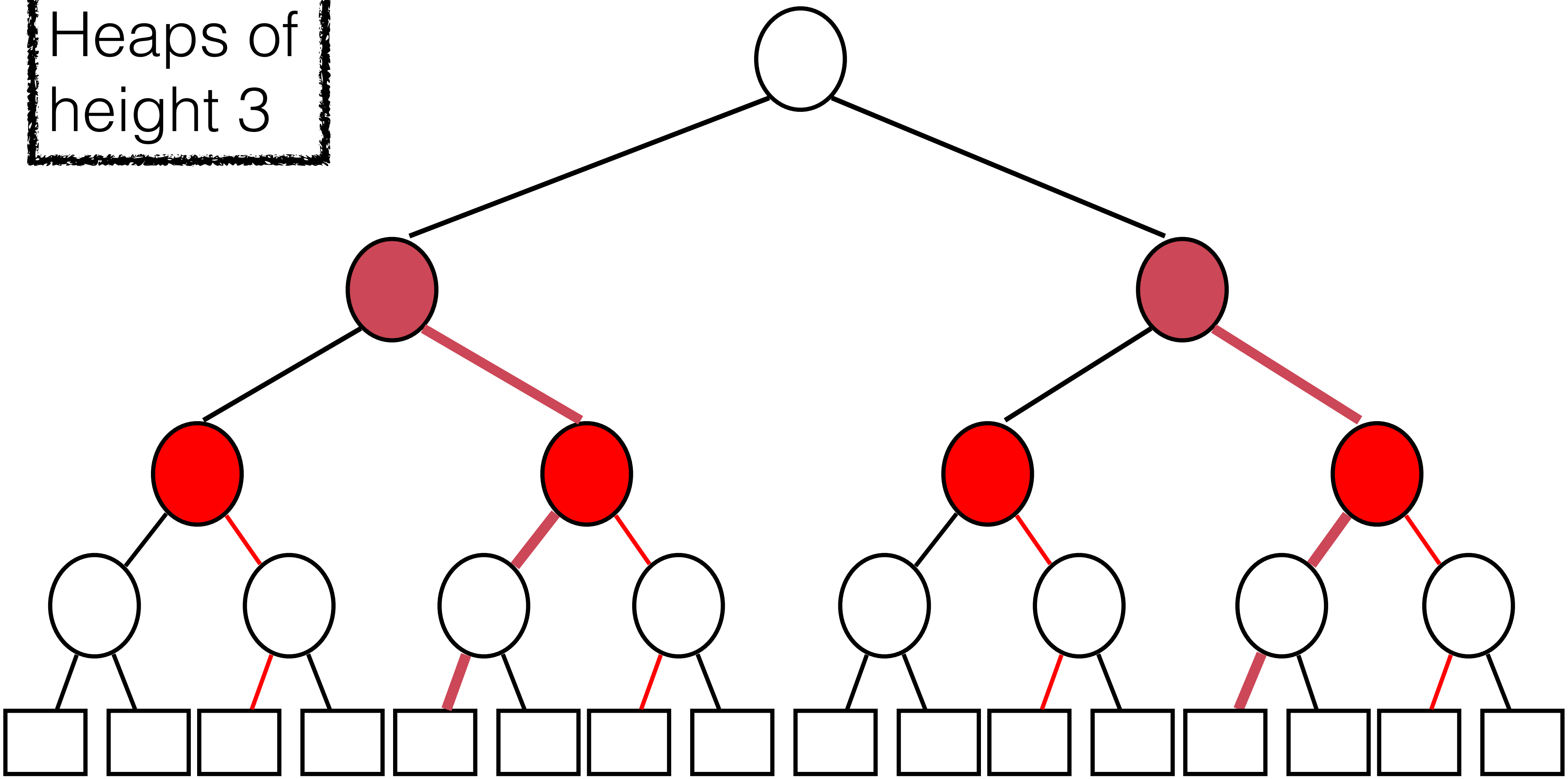
For each new node joining two heaps: mark path with maximum number of bubble-down operations

Heaps of height 3



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

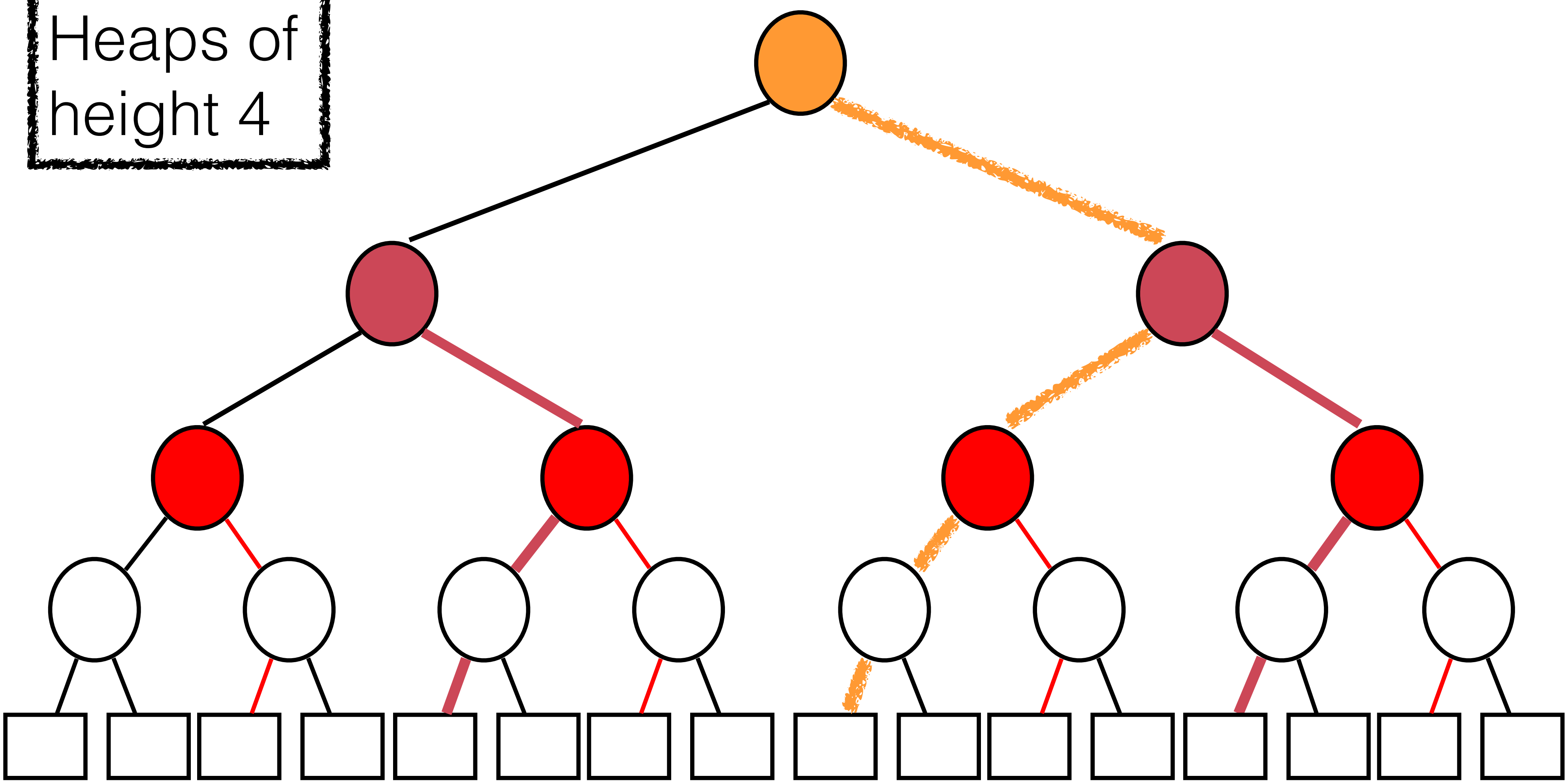
Heaps of height 3



For each height-3 heap, leftmost branch not marked

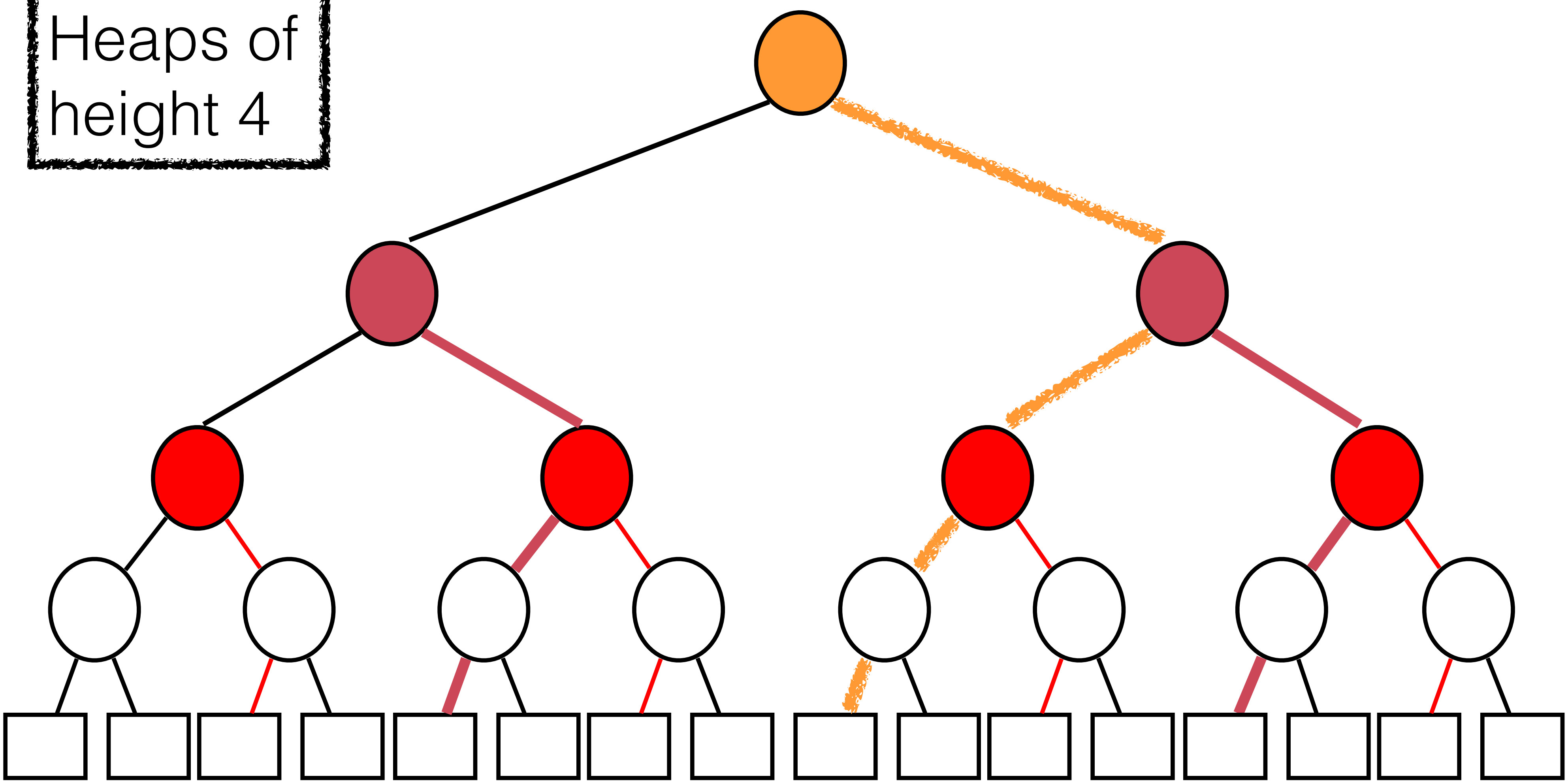
For each new node joining two heaps: mark path with maximum number of bubble-down operations

Heaps of height 4



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

Heaps of height 4



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

More efficient version (Kruskal's algorithm)

- union-find data structure

Kruskal's Algorithm

Algorithm Kruskal
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: Disjoint sets (lists or union-find) *DS*;
sorted weights priority queue A; and tree T

for each $v \in V$ **do** C(v) $\leftarrow$ DS.insert(v) **end //** one cluster per vertex
for each (u,*v)* $\in E$ **do** A.insert((u,v)) **end //** sort edges by weight
$T \leftarrow \varnothing$
while T has fewer than n-1 edges **do**
  (u, v) $\leftarrow$ A.deleteMin() // edge with smallest weight
  $C(u) \leftarrow$ DS.findCluster(u);
  $C(v) \leftarrow$ DS.findCluster(v);
  if $C(u) \neq C(v)$ **then**
    add edge (u, v) to T;
    DS.insert(DS.union($C(v)$, $C(u)$)); // merge two clusters
  end
end
return T

Kruskal's Algorithm

Time complexity analysis

Algorithm Kruskal
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: Disjoint sets (lists or union-find) DS;
    sorted weights A[]; and tree T

for each vertex v in V of G **do** C(v) ← DS.insert(v) **end** — $O(n)$
A[] ← sort all edges by edge weight; // sorted array priority queue
$T \leftarrow \emptyset$; $k \leftarrow 0$; — $O(m \log m)$ or O(m) with heap
while T has fewer than n-1 edges **do**
  $(u, v) \leftarrow$ A[k]; $k \leftarrow k + 1$; // next edge with smallest weight; deleteMin()
  $C(v) \leftarrow$ DS.findCluster(v);
  $C(u) \leftarrow$ DS.findCluster(u);
  if $C(v) \neq C(u)$ **then**
    add edge (v, u) to T;
    DS.insert(DS.union($C(v)$, $C(u)$)); // merge two clusters — Total merging $O(m \log n)$
  end
end
return T

Total $O(m \log n)$

Union-find (or disjoint set) data structure

Disjoint Set Data Structure

- Given a set of elements, it is often useful to break them up or partition them into a number of separate, nonoverlapping sets
- A ***disjoint-set data structure*** is a data structure that keeps track of such a partitioning
- A ***union-find algorithm*** is an algorithm that performs two useful operations on such a data structure:
  - ***Find***: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
  - ***Union***: Combine or merge two sets into a single set.
  - ***MakeSet***: Creates a set containing only a given element

http://en.wikipedia.org/wiki/Disjoint-set_data_structure

Disjoint Set Data Structure

- The universe consists of n elements, named $1, 2, \ldots, n$
- The ADT is a collection of sets of elements
- Each element is in exactly one set
  - Sets are disjoint
  - To start, each set contains one element
- Each set has a name
  - Which is the name of one of its elements
  - Any name of one of its elements will do

Disjoint Set Operations

- **find(elementName)**
  - Returns the name of the unique set that contains the given element
- **union(setName1, setName2)**
  - Merges two sets and replaces them with one

- **Time complexity analysis**
  - Involves analyzing the *amortized* worst-case running time over a sequence of f find and u union operations

Disjoint Set Implementation I

- Create a linked list for each set and choose the element at the head of the list as the representative
- ***MakeSet*** creates a list of one element
- ***Union*** simply appends two lists, a constant-time operation
- ***Find*** requires linear time (i.e., may search entire list)
- A sequence of m union-find operations takes time $O(mn)$.The amortized time per operation is $O(n)$.

Disjoint Set Implementation II

- Two complementary techniques, *union by rank* and *path compression,* reduce the amortized time complexity per operation to $O(\alpha(n))$, where $\alpha(n)$ is the *inverse* of function $f(n) = A(n,n)$, and A is the extremely quickly-growing Ackermann function
- Since $\alpha(n)$ is its inverse it grows extremely slowly: it is less than 5 for all remotely practical values of n
- Thus, the amortized running time per operation is effectively a small constant
- Thus, n union and find operations require time $O(n\alpha(n))$

Union-find operations

- possible implementation
- Representing of sets in partition

Union-Find Operations: union(a, b)

- Given a partition, let a and b be *canonical elements* of sets A and B, respectively
- If $a \neq b$:
  - Form a new set that is the union of the two sets
  - Destroy the two old sets
  - Select & return canonical element for new set

How should we represent the sets?

- Each set is rooted tree
- Each element of set corresponds to node in tree
- *Canonical element* (= name of set) is root of tree
- Each node e has reference e.parent to its parent in tree, the root points to itself (r.parent = r)

Implementation I
Naive union-find algorithm

- makeset(e): e.parent $\leftarrow e$
  canonical element is root

- find(e):
  - Follow parent pointers from e to root of tree that contains e
  - Return root

- union(a,b)
  - a.parent $\leftarrow b$
  - Return b as canonical element of new set

Union-Find Operations: makeSet(e)

Create singleton set containing single element e
[we do this if e is encountered but is previously in no set]

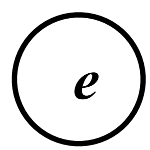
e.parent $\leftarrow e$

canonical element is root, i.e., e.

Union-Find Operations: makeSet(e)

Create singleton set containing single element e
[we do this if e is encountered but is previously in no set]

e.parent $\leftarrow e$

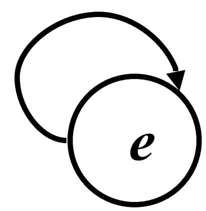


canonical element is root, i.e., e.

Union-Find Operations: makeSet(e)

Create singleton set containing single element e
[we do this if e is encountered but is previously in no set]

e.parent $\leftarrow$ e



canonical element is root, i.e., e.

Union-Find Operations: find(e)

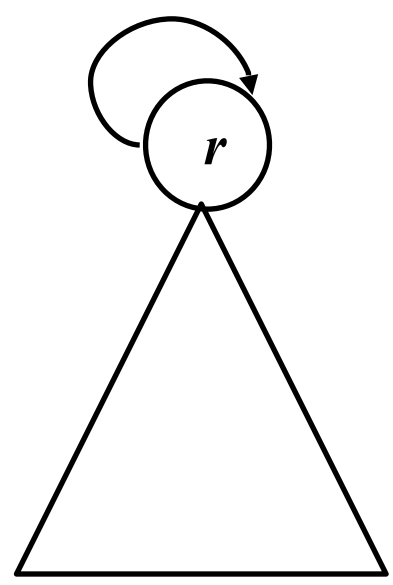
Return canonical element of the set containing e

- Follow parent pointers from e to root r of tree that contains e
- Return root r

Union-Find Operations: find(e)

Return canonical element of the set containing e

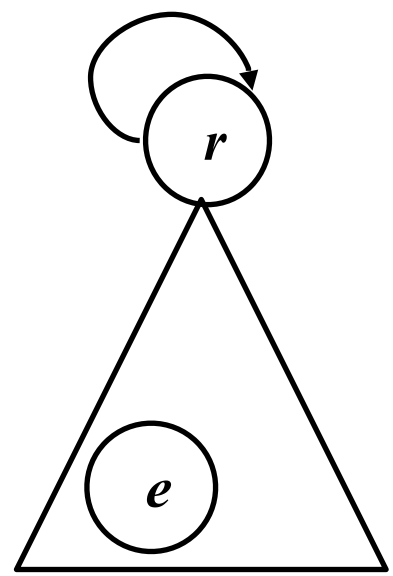
- Follow parent pointers from e to root r of tree that contains e
- Return root r



Union-Find Operations: find(e)

Return canonical element of the set containing e

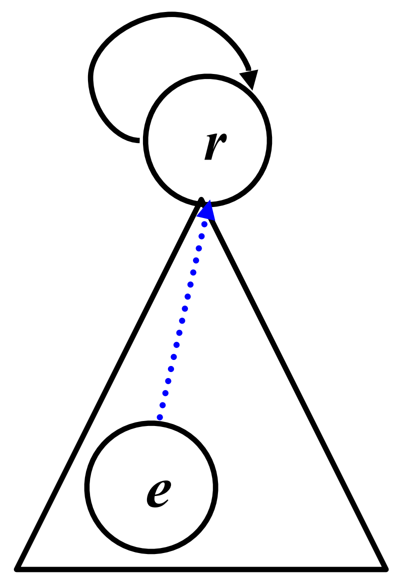
- Follow parent pointers from e to root r of tree that contains e
- Return root r



Union-Find Operations: find(e)

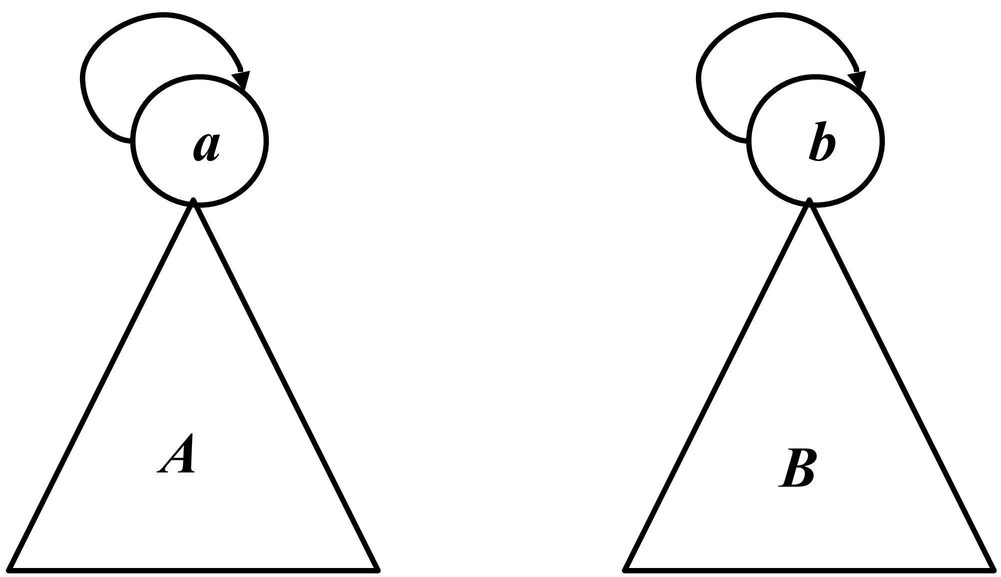
Return canonical element of the set containing e

- Follow parent pointers from e to root r of tree that contains e
- Return root r



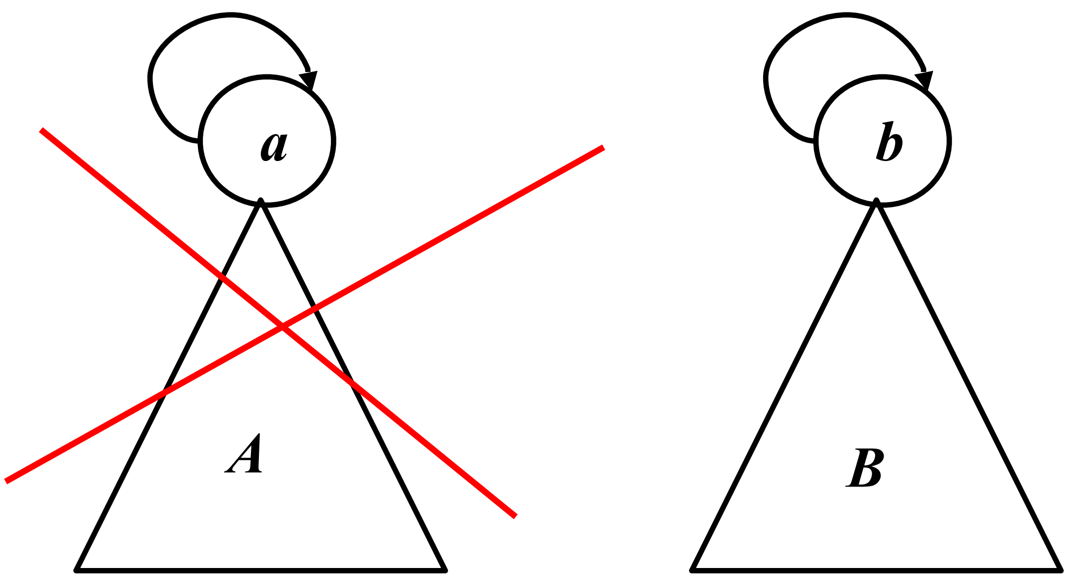
Union-Find Operations: union(a, b)

- Let a and b be canonical elements of sets A and B, respectively
- If $a \neq b$ then form a new set: union of two sets whose canonical elements are a and b
- Destroy two old sets
- Select and return canonical element for new set: a.parent $\leftarrow b$
- Return b as canonical element of new set



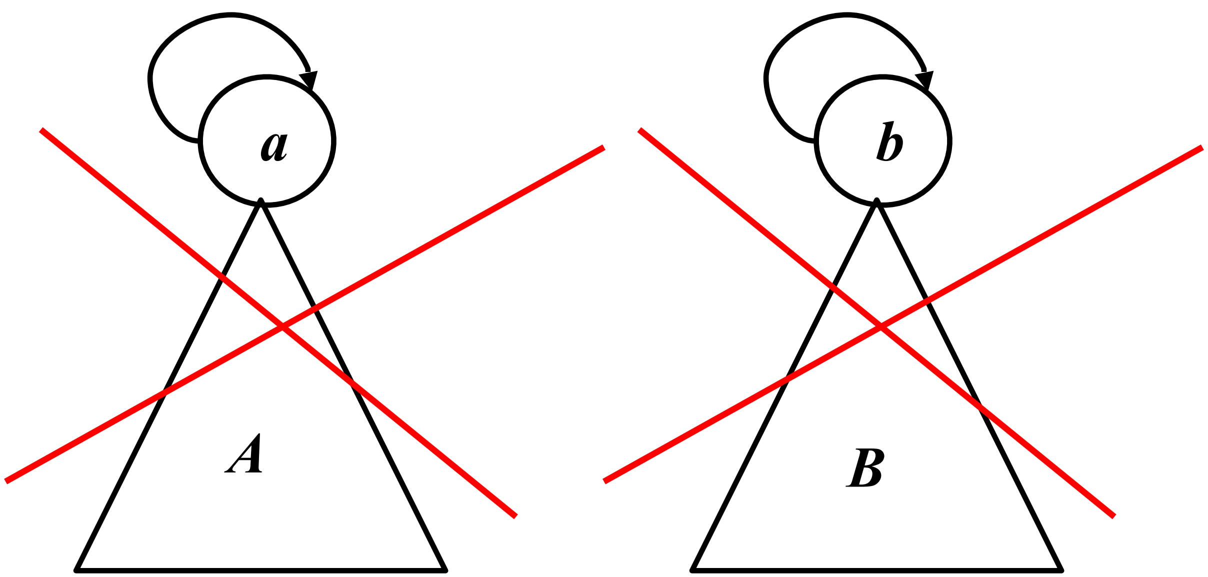
Union-Find Operations: union(a, b)

- Let a and b be canonical elements of sets A and B, respectively
- If $a \neq b$ then form a new set: union of two sets whose canonical elements are a and b
- Destroy two old sets
- Select and return canonical element for new set: a.parent ← b
- Return b as canonical element of new set



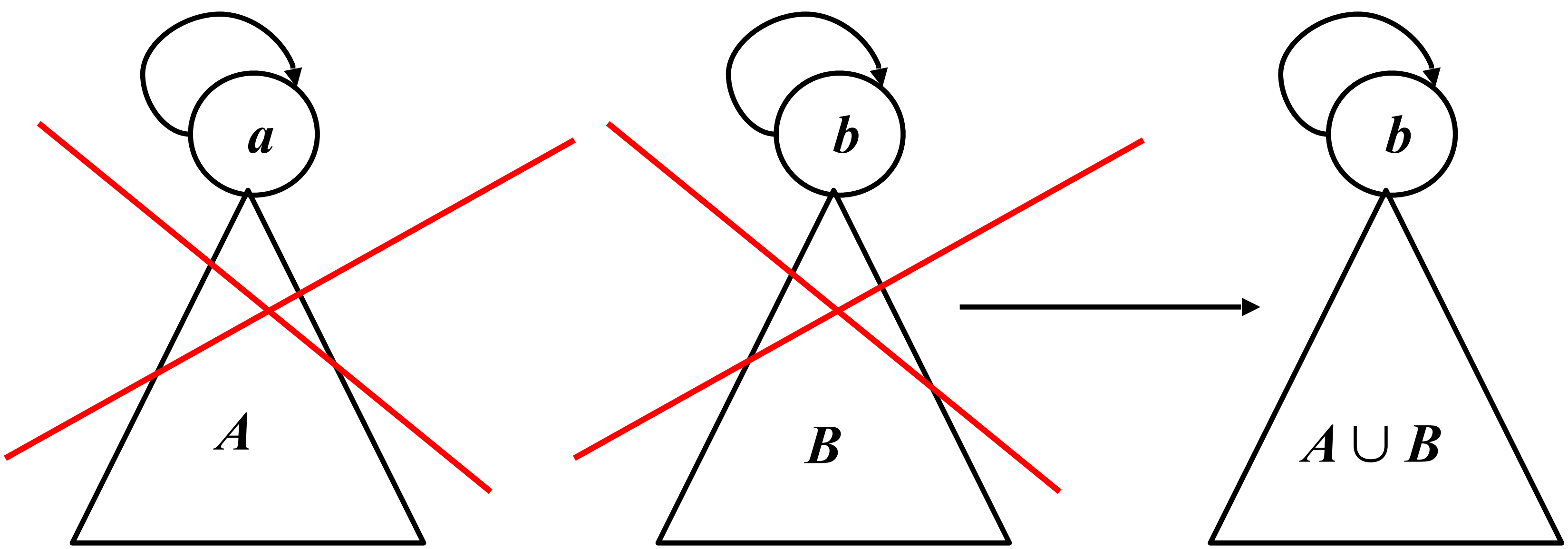
Union-Find Operations: union(a, b)

- Let a and b be canonical elements of sets A and B, respectively
- If $a \neq b$ then form a new set: union of two sets whose canonical elements are a and b
- Destroy two old sets
- Select and return canonical element for new set: a.parent ← b
- Return b as canonical element of new set



Union-Find Operations: union(a, b)

- Let a and b be canonical elements of sets A and B, respectively
- If $a \neq b$ then form a new set: union of two sets whose canonical elements are a and b
- Destroy two old sets
- Select and return canonical element for new set: a.parent ← b
- Return b as canonical element of new set



The naive union-find algorithm

- makeset(e): e.parent $\leftarrow e$ (canonical element is root)

- find(e):
  - Follow parent pointers from e to root of tree containing e
  - Return root

- union(a,b): a.parent $\leftarrow b$ (Return b as canonical element of new set)

The naive union-find algorithm

- makeset(e): e.parent $\leftarrow e$ (canonical element is root)

  $O(1)$

- find(e):
  - Follow parent pointers from e to root of tree containing e
  - Return root

- union(a,b): a.parent $\leftarrow b$ (Return b as canonical element of new set)

The naive union-find algorithm

- makeset(e): e.parent $\leftarrow e$ (canonical element is root)

  O(1)

- find(e):
  - Follow parent pointers from e to root of tree containing e
  - Return root

- union(a,b): a.parent $\leftarrow b$ (Return b as canonical element of new set)

  O(1)

The naive union-find algorithm

- makeset(e): e.parent $\leftarrow e$ (canonical element is root) O(1)
- find(e):
  - Follow parent pointers from e to root of tree containing e
  - Return root

  O(n)
- union(a,b): a.parent $\leftarrow b$ (Return b as canonical element of new set) O(1)

An *improved* union-find algorithm

- makeset(e): e.parent $\leftarrow e$, the canonical element is the root.

- find(e): Follow the parent pointers from e to the root of the tree containing e. Return the root.

- **union(a,b): a.parent $\leftarrow$ canonical element of *larger* of two sets**

Why is this better?

Improved Union-Find: Union by Rank

Idea: Store with each node v the size of the subtree (*rank*) rooted at y. In a union operation, make the tree of the smaller set/rank a subtree of the other tree, and update the size field of the root of the resulting tree.

Union by Rank

- With each node x we store a nonnegative integer x.rank that is an upper bound on the height of x.

- When carrying out makeSet(x), we set x.rank $\leftarrow$ 0.
- To carry out union(x,y), we compare x.rank and y.rank.
  - If x.rank < y.rank then x.parent $\leftarrow y$.
  - If y.rank < x .rank then y.parent $\leftarrow x$.
  - If x.rank = y.rank then x .parent $\leftarrow y$ and y.rank $\leftarrow$ y.rank + 1.

Amortized Time Complexity

- A series of n makeSet, (modified) union, and find operations starting from an initially empty partition take
  O(n log n) time.

- The *amortized* running time per operation is O(log n) time.

A series of n makeSet, (modified) union, and find operations starting from an initially empty partition take O(n log n) time

Lemma: Starting with sets of size 1, using the modified union algorithm, any tree of m nodes has height of at most log m .