CS2020
Data Structures and Algorithms
(Recitation)

Welcome!

Today

2-3-4 Trees:

- New type of balanced search tree

Cache Aware Algorithms

- Modeling cache performance
- B-trees

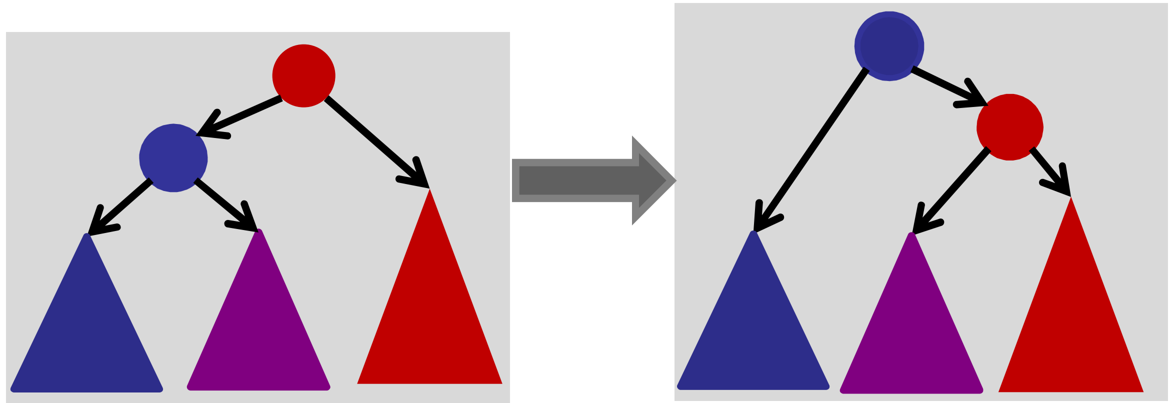
Balanced Search Trees

Many types:

- AVL trees
- Red-Black trees
- Splay trees
- …

All use <u>rotations</u>

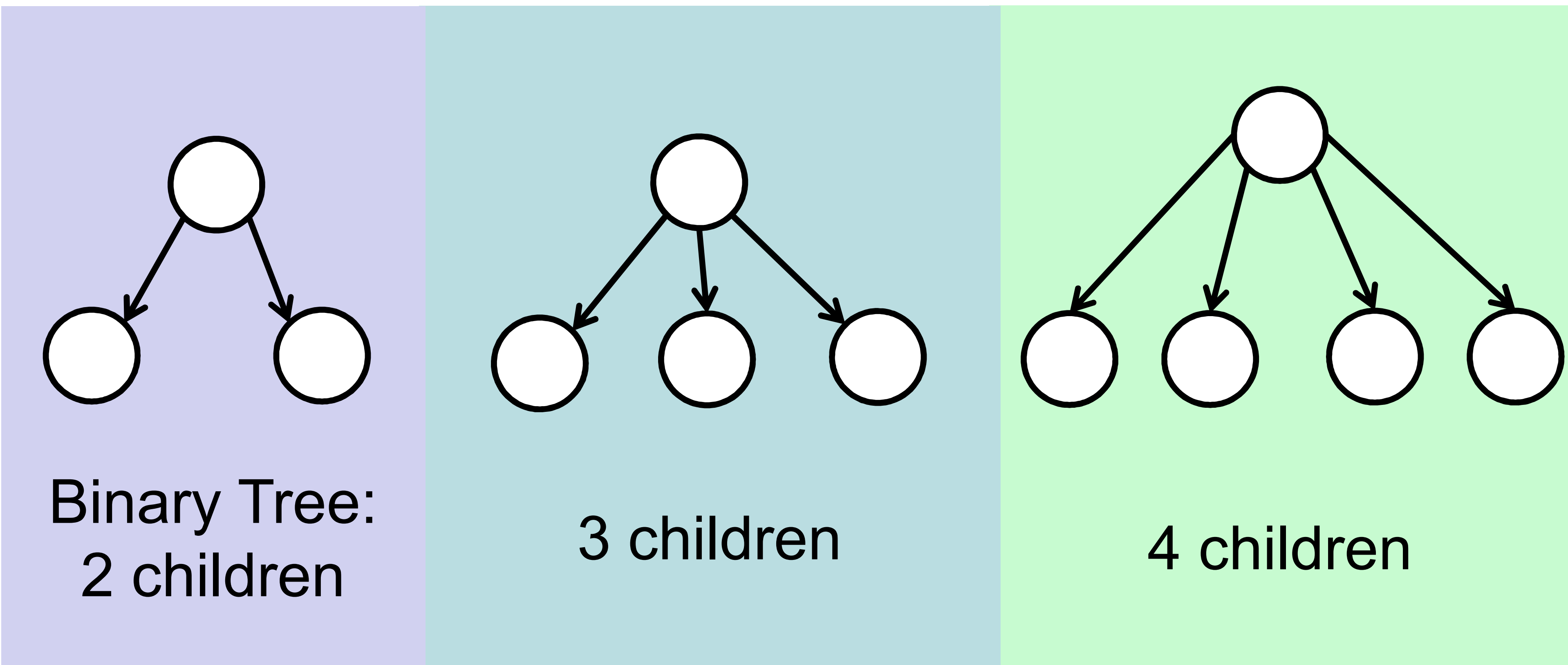
- Complicated and unintuitive
- Concurrency is hard
- Messy: lots of case analysis



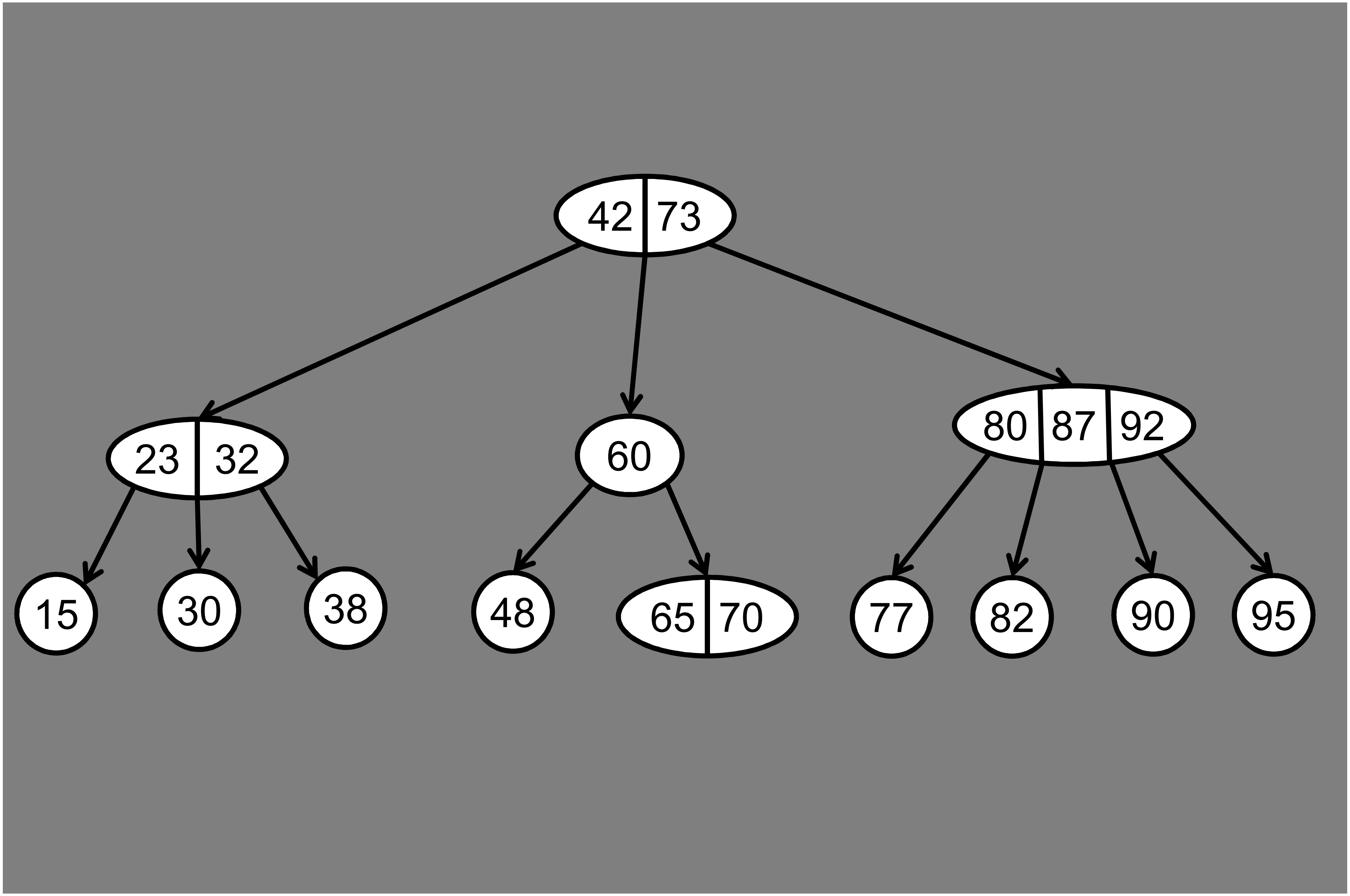
2-3-4 Trees

Rule # 1:

- Every non-leaf node has either:
  2 or 3 or 4 children



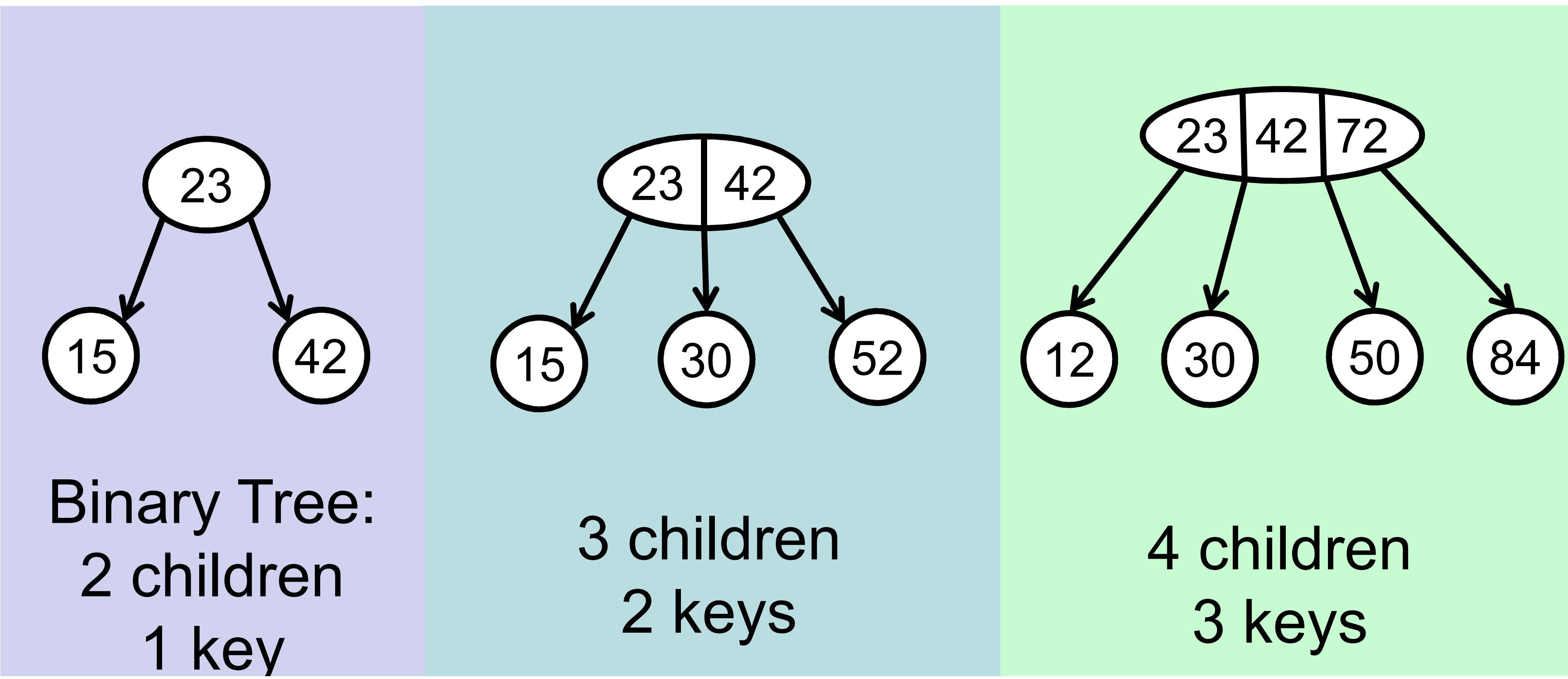
2-3-4 Trees



2-3-4 Trees

Rule # 2:

- Satisfies binary search property



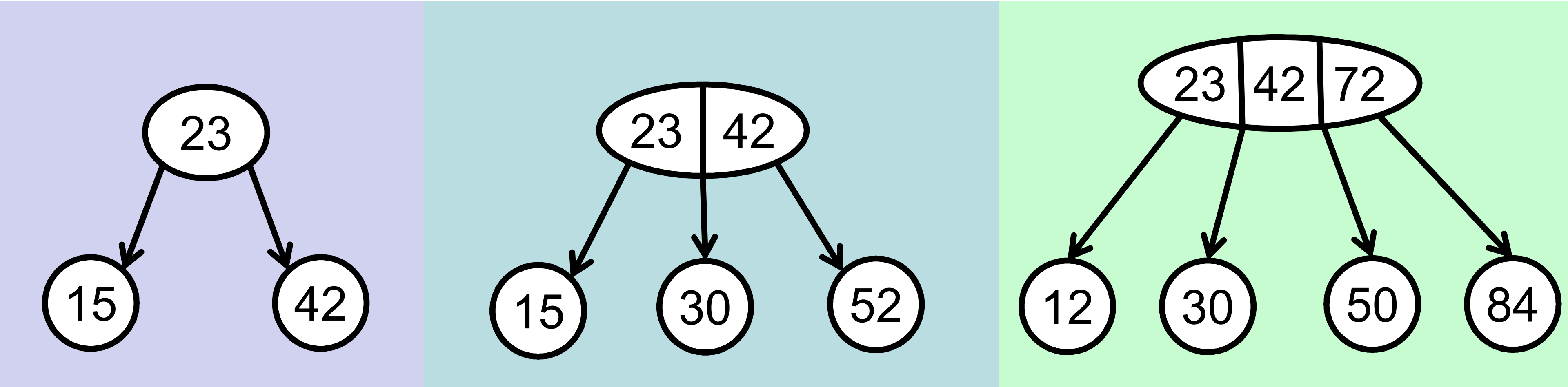
2-3-4 Trees

Rule # 2:

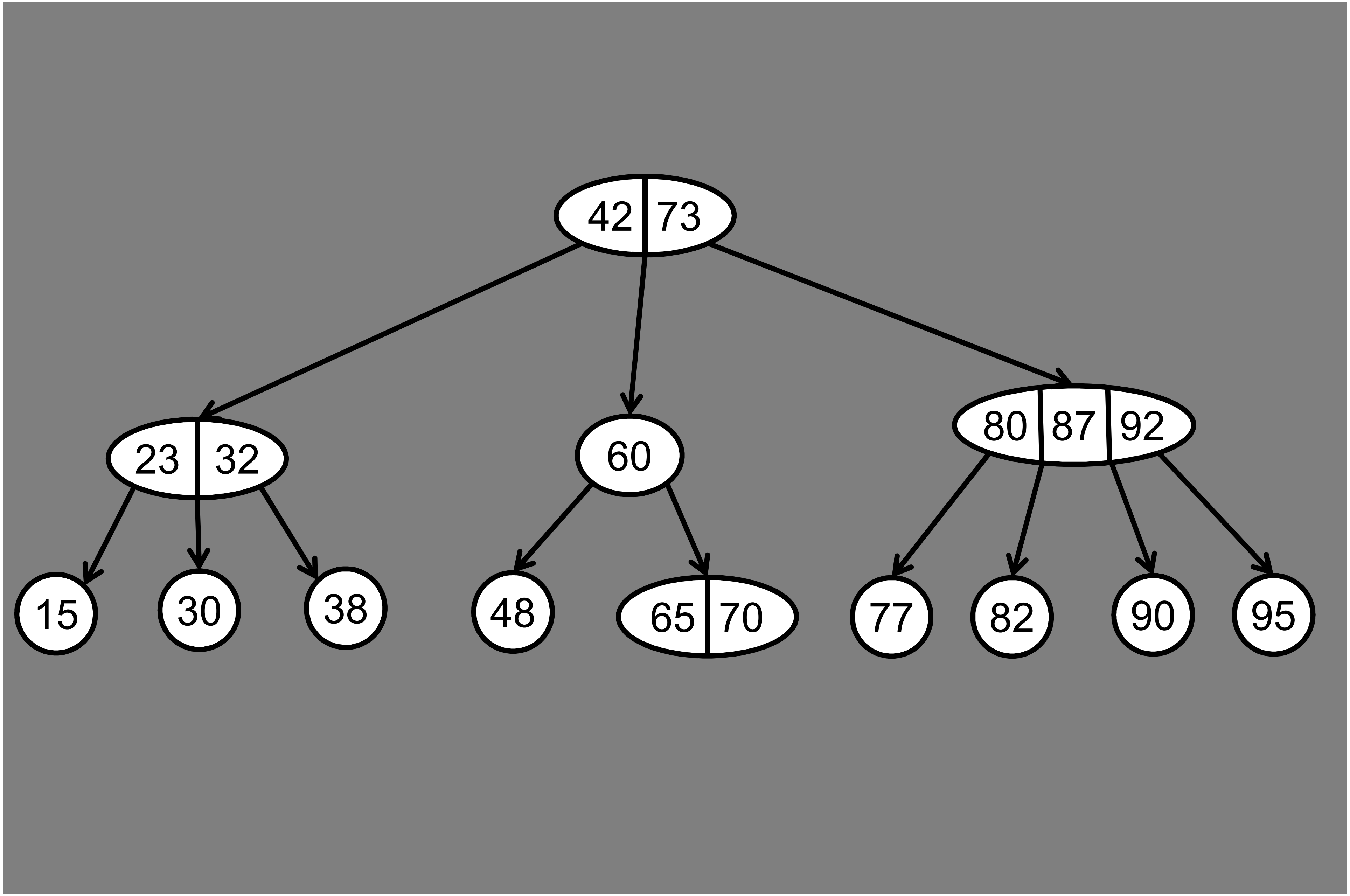
- Satisfies binary search property

  Node contains keys $\{k_1, \ldots, k_r\}$, $r \in \{1,2,3\}$

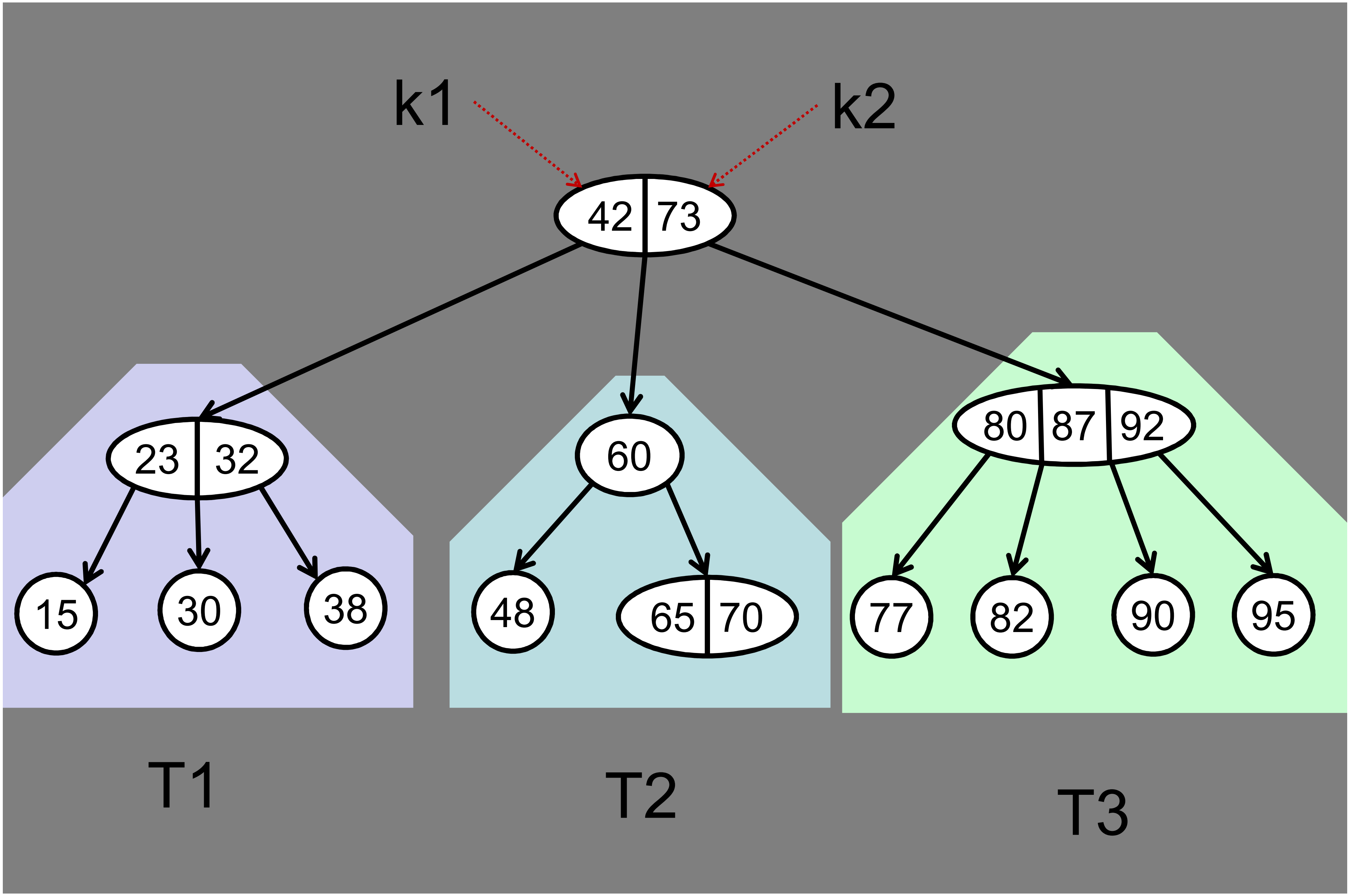
  Node contains subtrees $\{T_1, \ldots, T_{r+1}\}$, $r \in \{1,2,3\}$



2-3-4 Trees



2-3-4 Trees



2-3-4 Trees

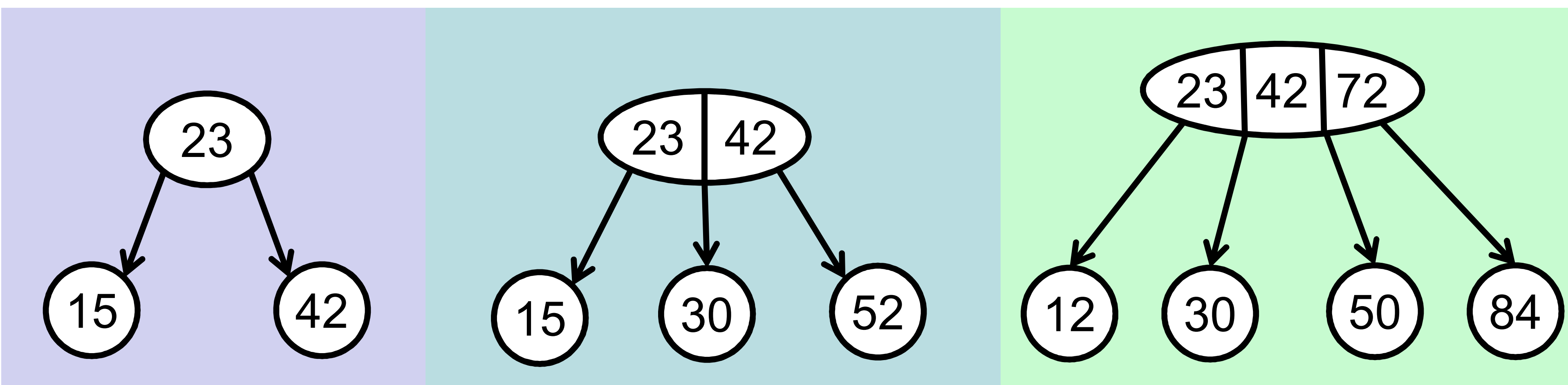
Rule # 2:

– Satisfies search property

Node contains keys $\{k_1, \ldots, k_r\}$, $r \in \{1,2,3\}$

Node contains subtrees $\{T_1, \ldots, T_{r+1}\}$, $r \in \{1,2,3\}$

$$\text{keys}(T_1) < k_1 < \text{keys}(T_2) < k_1 < \text{keys}(T_3) < k_3 < \text{keys}(T_4)$$



2-3-4 Trees

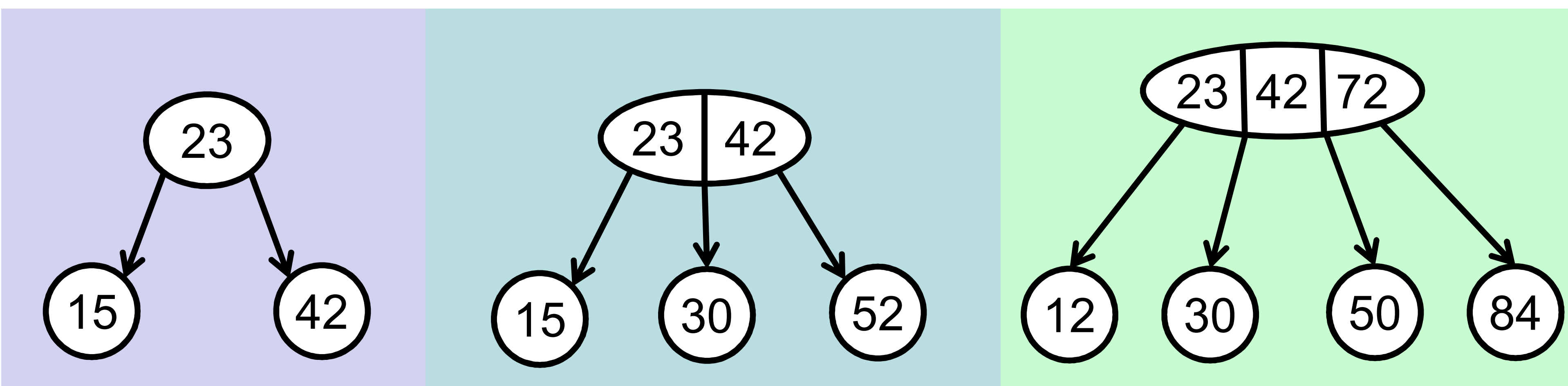
Rule # 2:

– Satisfies search property

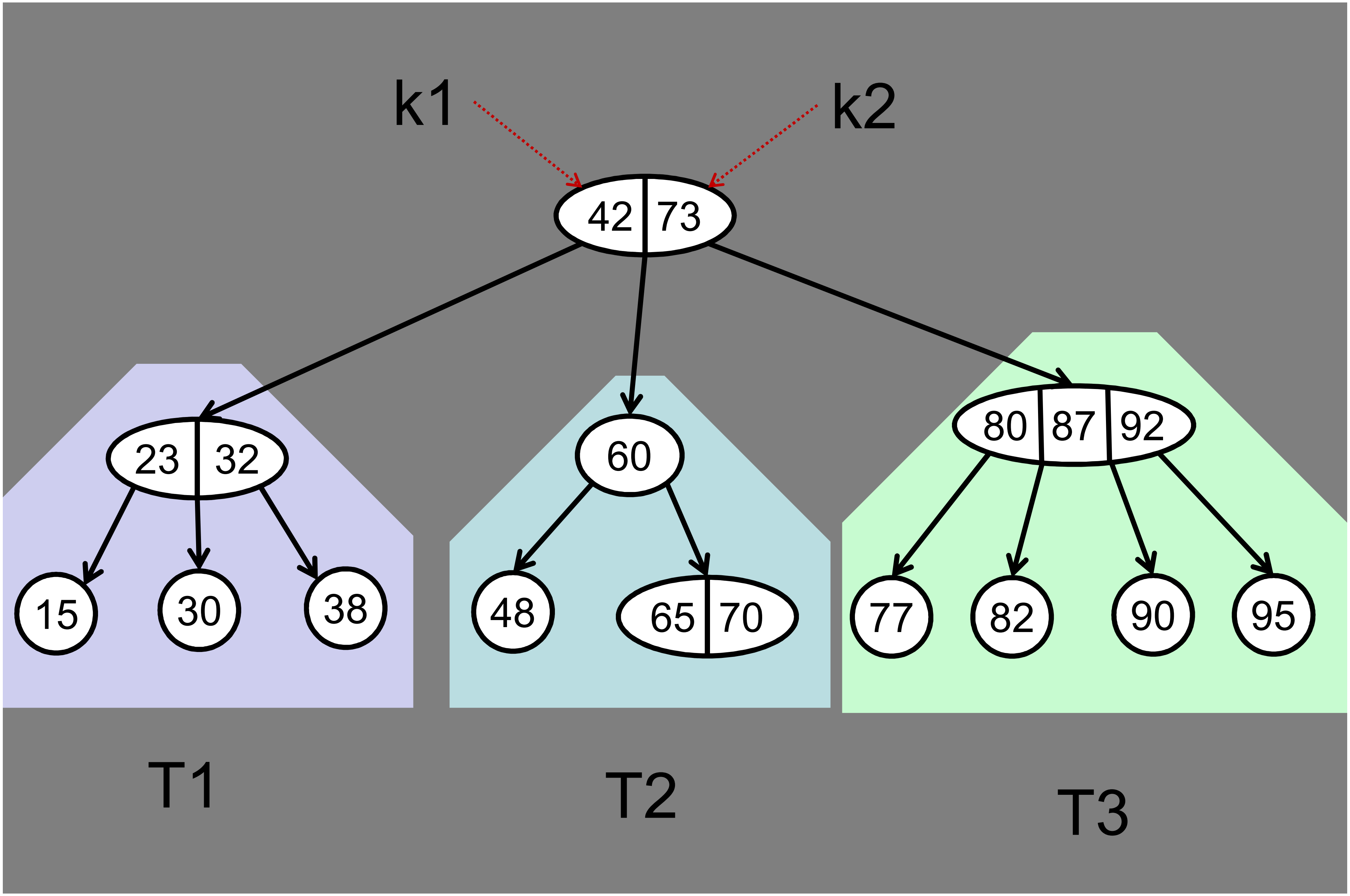
Node contains keys $\{k_1, \ldots, k_r\}$, $r \in \{1,2,3\}$

Node contains subtrees $\{T_1, \ldots, T_{r+1}\}$, $r \in \{1,2,3\}$

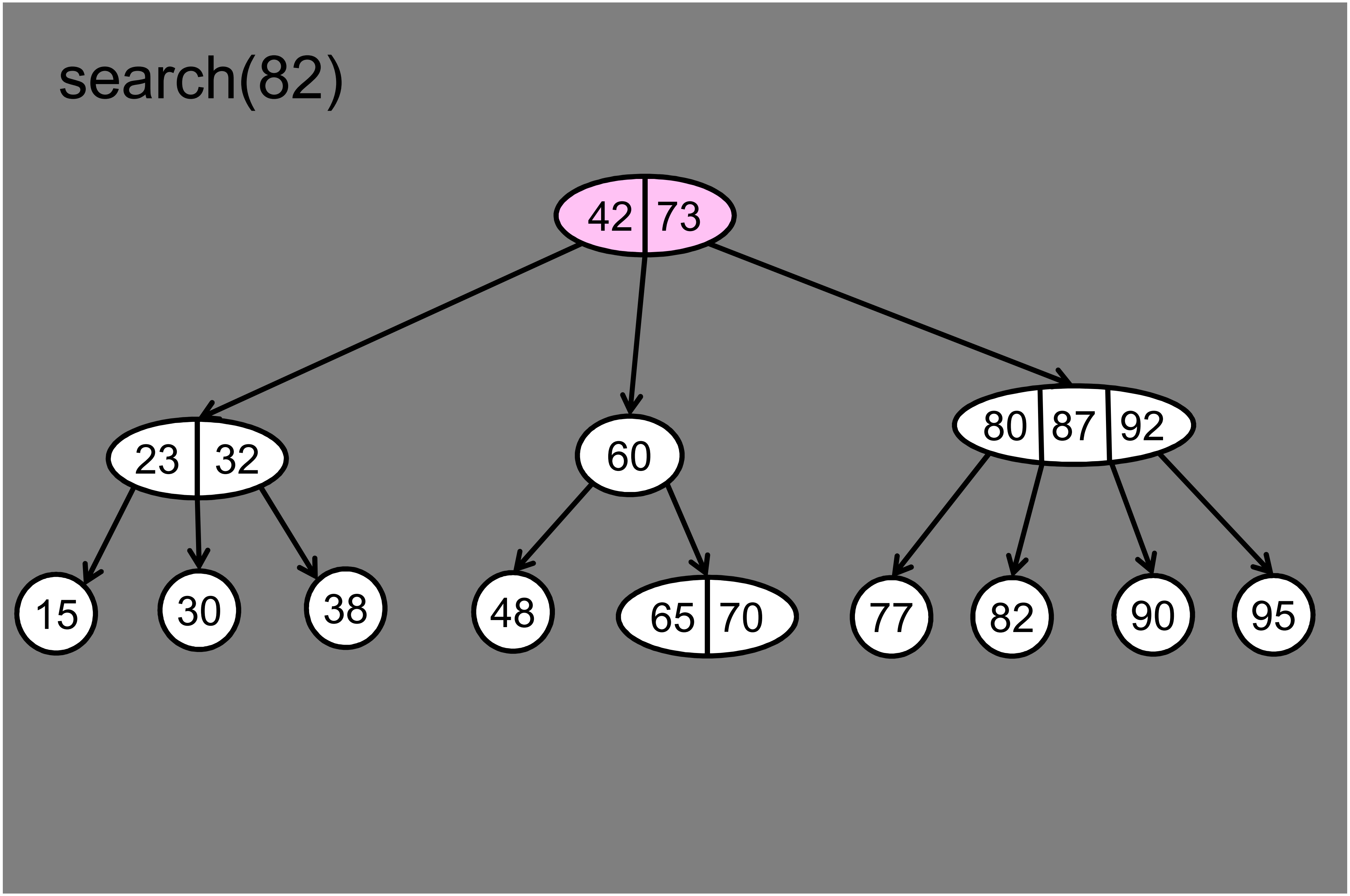
$$\forall r \in \{1,2,3\} : keys(T_r) < k_r < keys(T_{r+1})$$



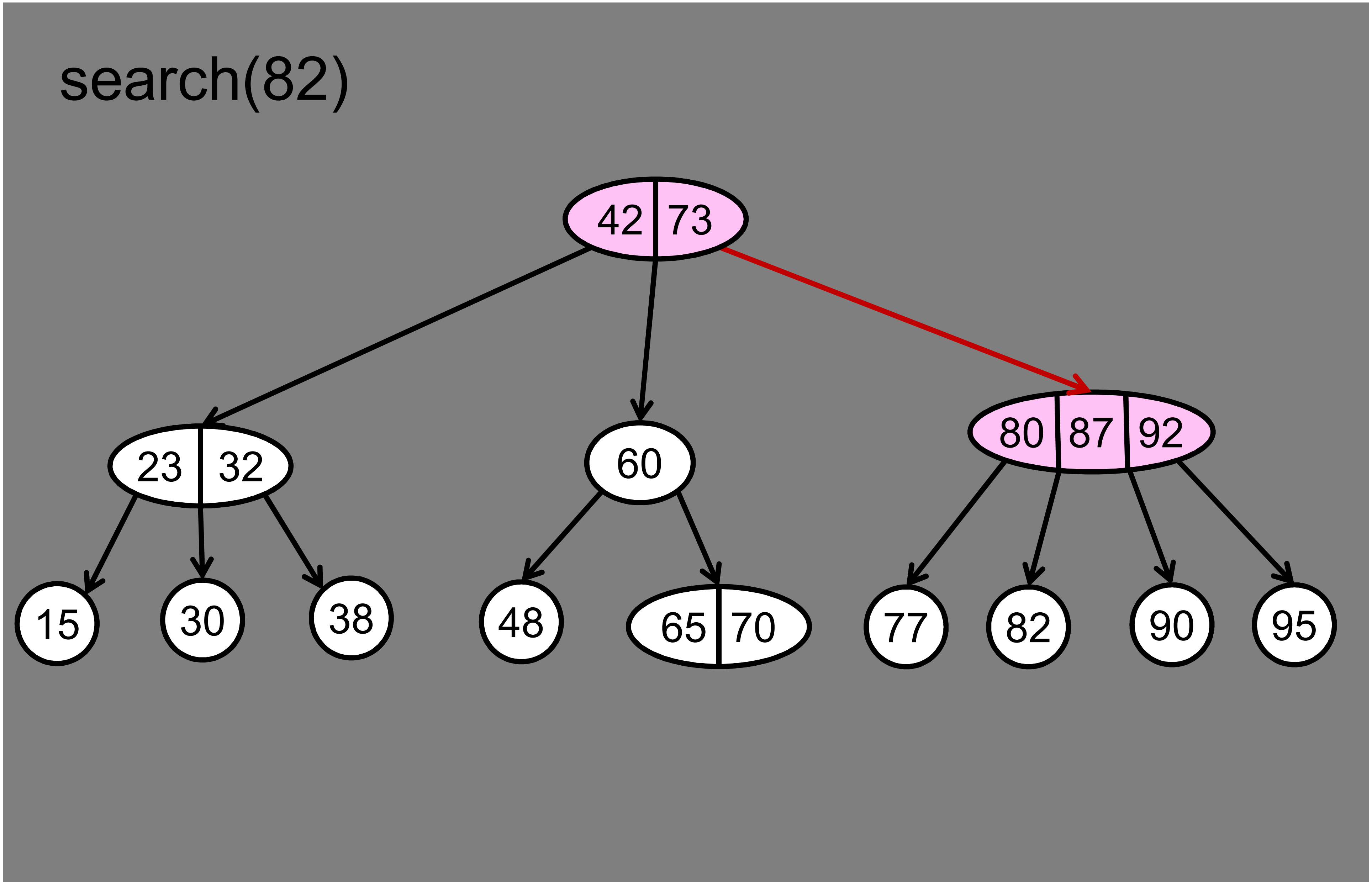
2-3-4 Trees



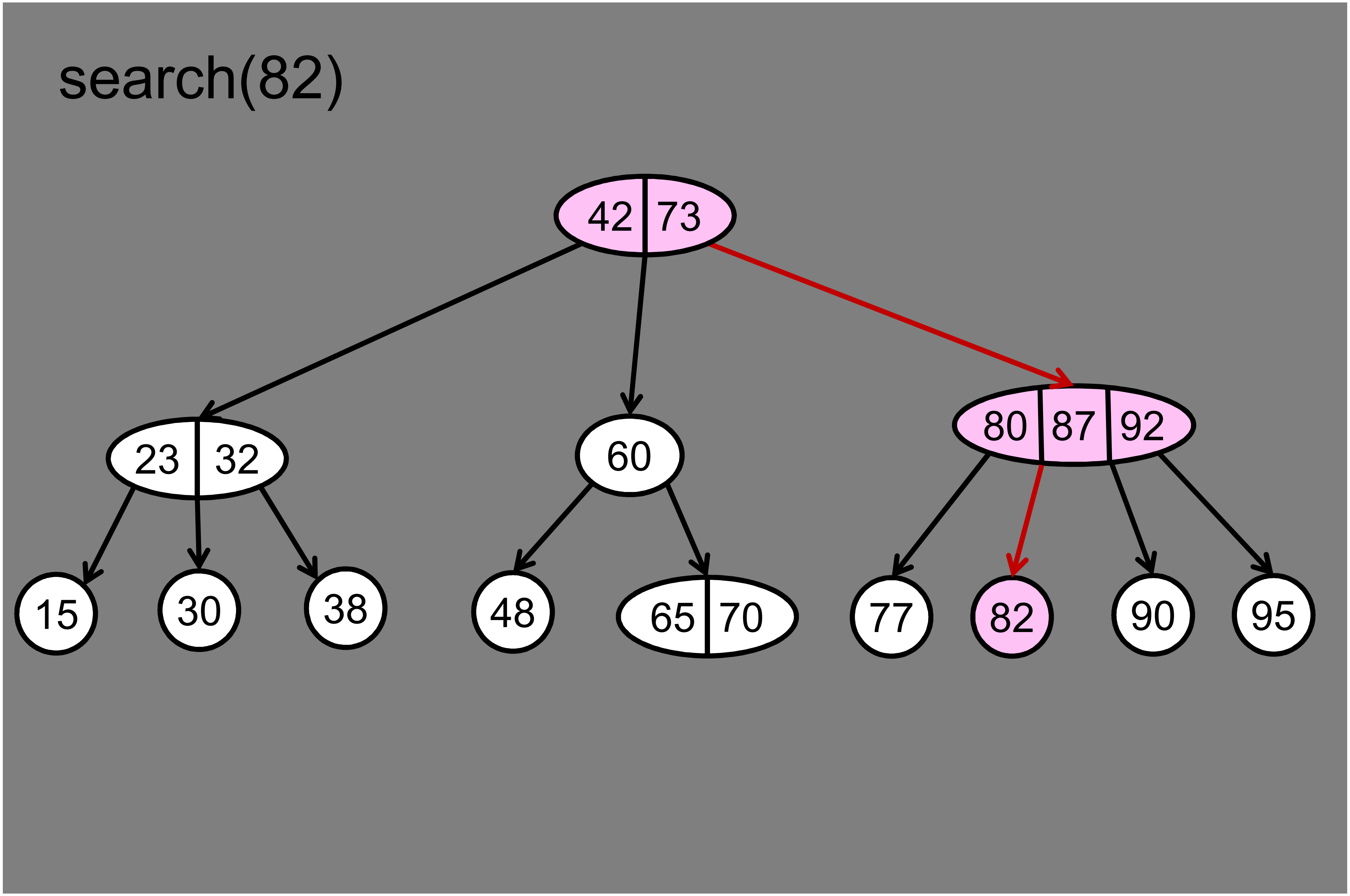
2-3-4 Trees: Searching



2-3-4 Trees: Searching



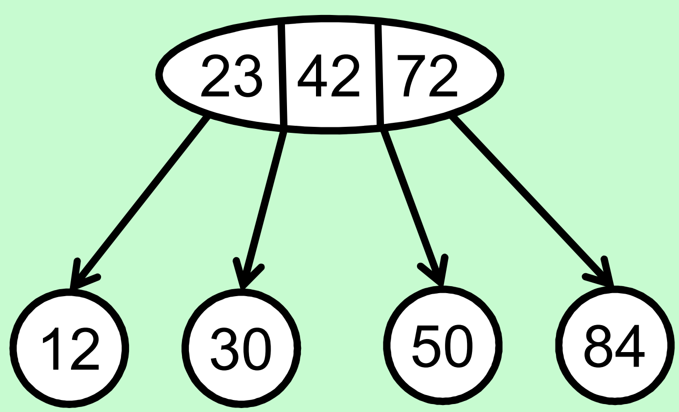
2-3-4 Trees: Searching



2-3-4 Trees

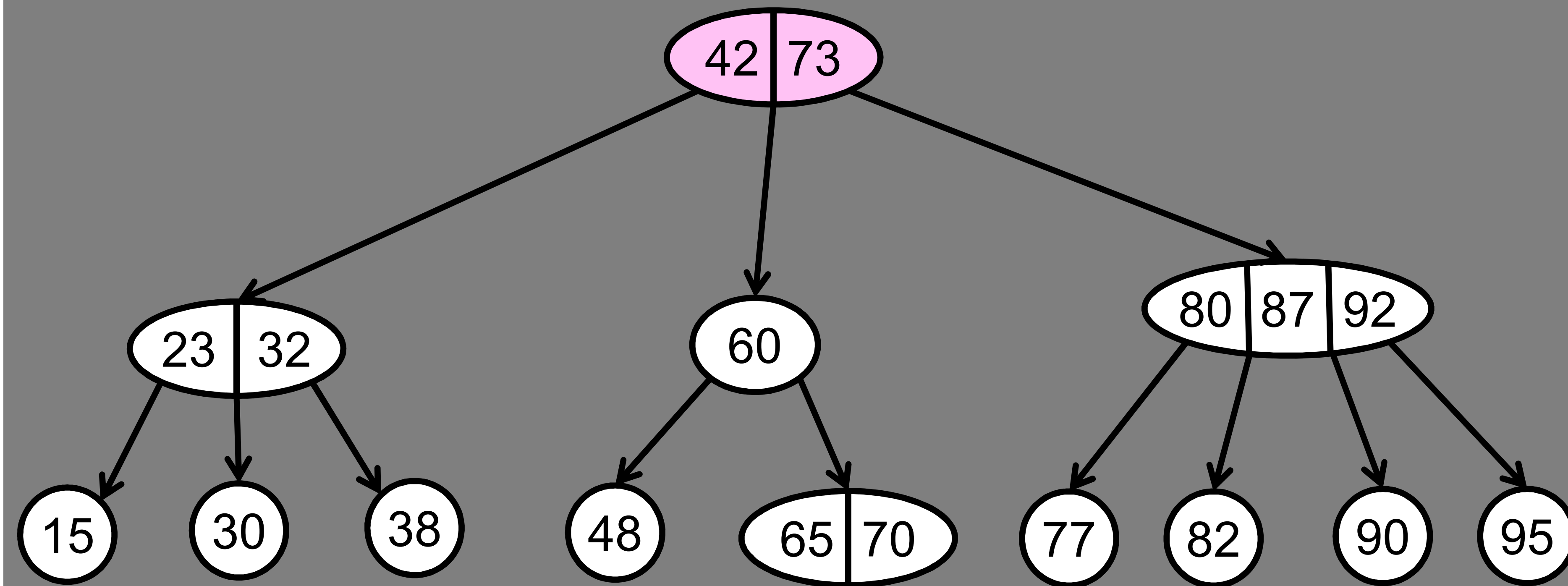
search(k):

```
j = 1;
while (j <= num_keys) and (k > k_j) do j++;
if (j <= num_keys) and (k == k_j) then return true;
else if (T_j != null) then return T_j.search(k);
else return false;
```



2-3-4 Trees: in-order-traversal

in-order-traversal()

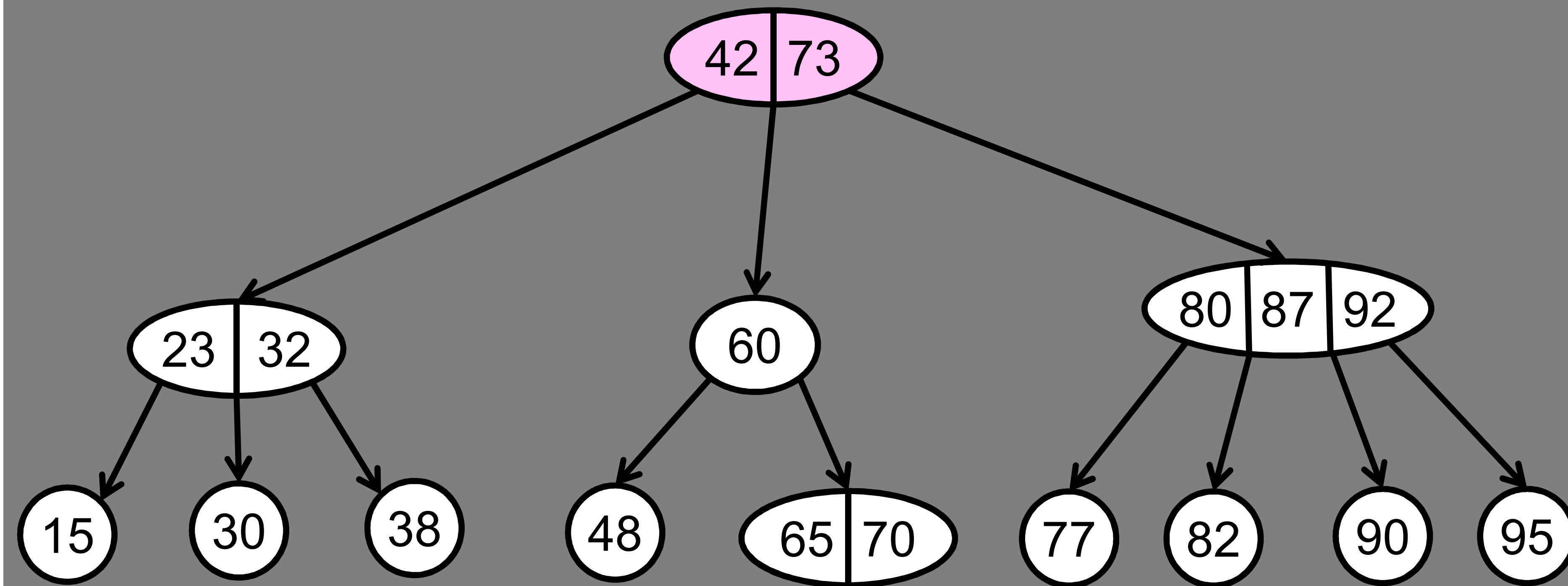


traverse T1:

15, 23, 30, 32, 38

2-3-4 Trees: in-order-traversal

in-order-traversal()

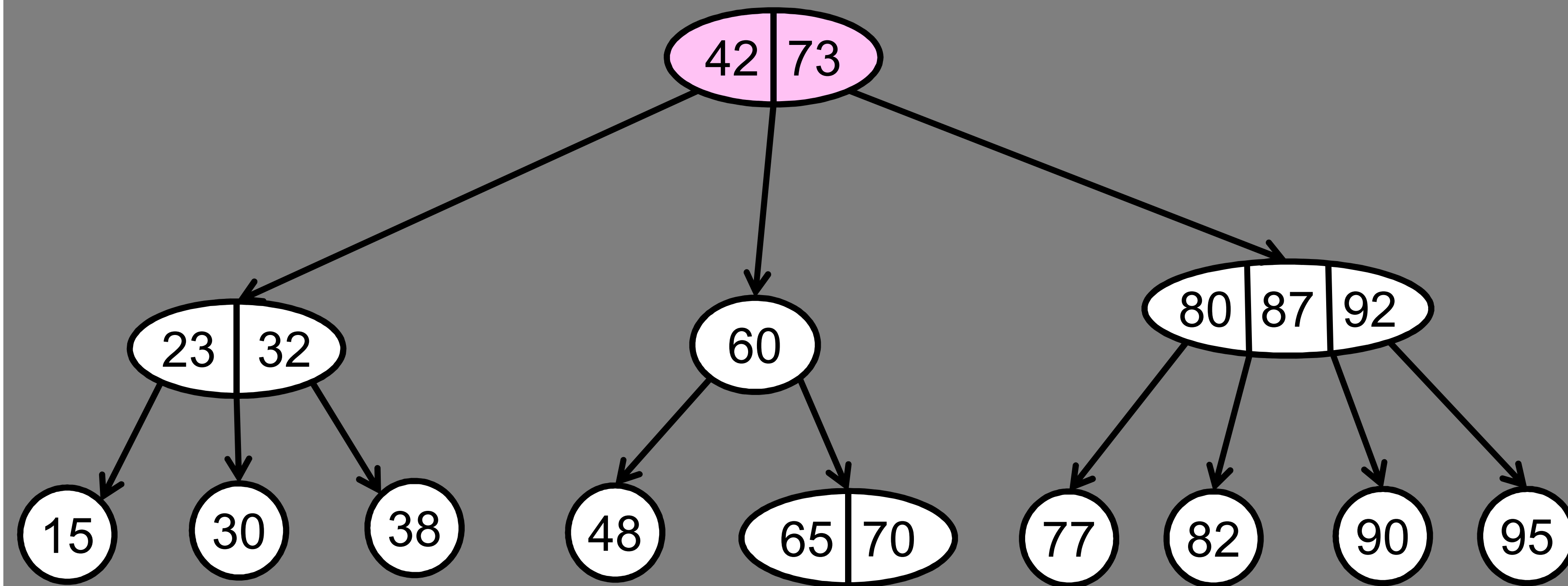


print k1:

15, 23, 30, 32, 38, 42

2-3-4 Trees: in-order-traversal

in-order-traversal()

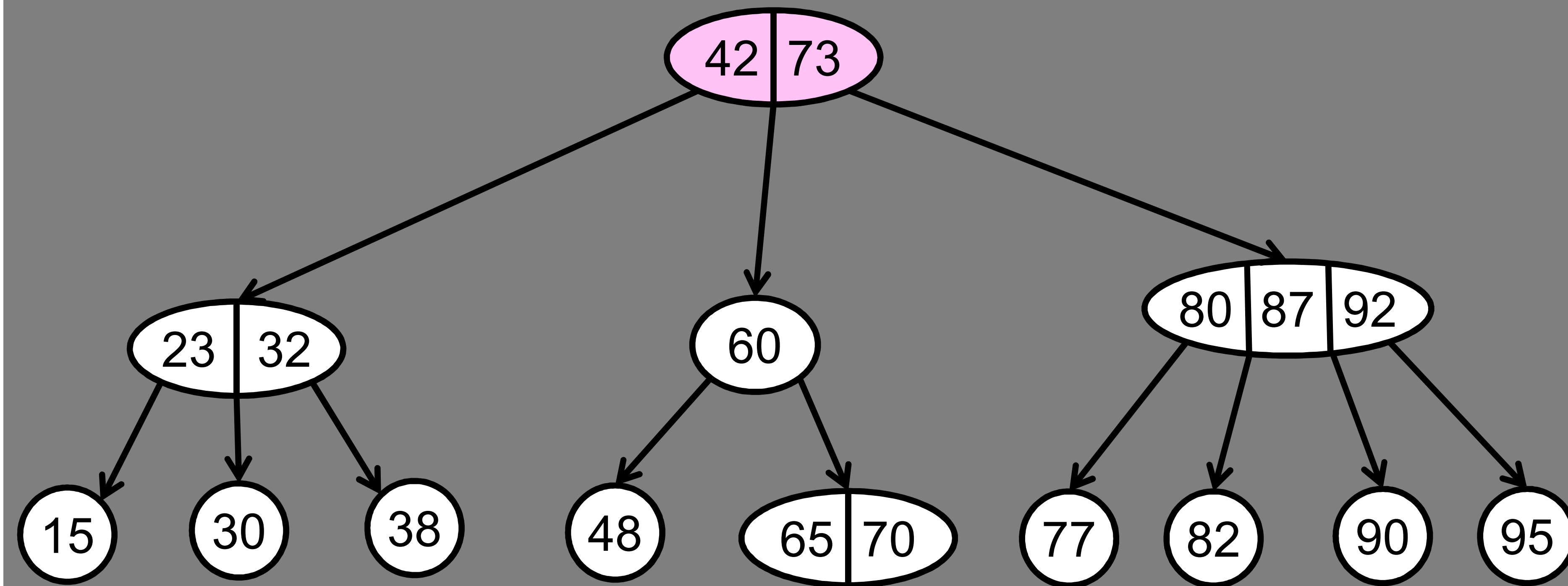


traverse T2:

15, 23, 30, 32, 38, 42, 48, 60, 65, 70

2-3-4 Trees: in-order-traversal

in-order-traversal()

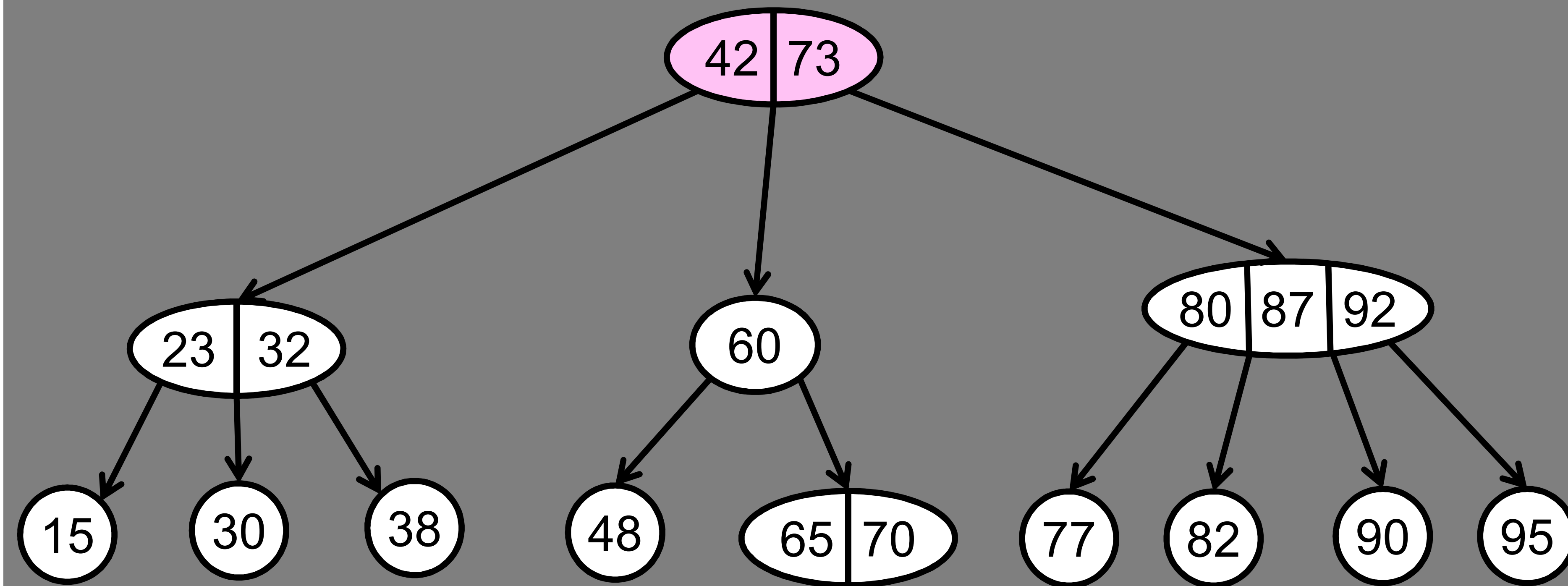


print k2:

15, 23, 30, 32, 38, 42, 48, 60, 65, 70, 73

2-3-4 Trees: in-order-traversal

in-order-traversal()



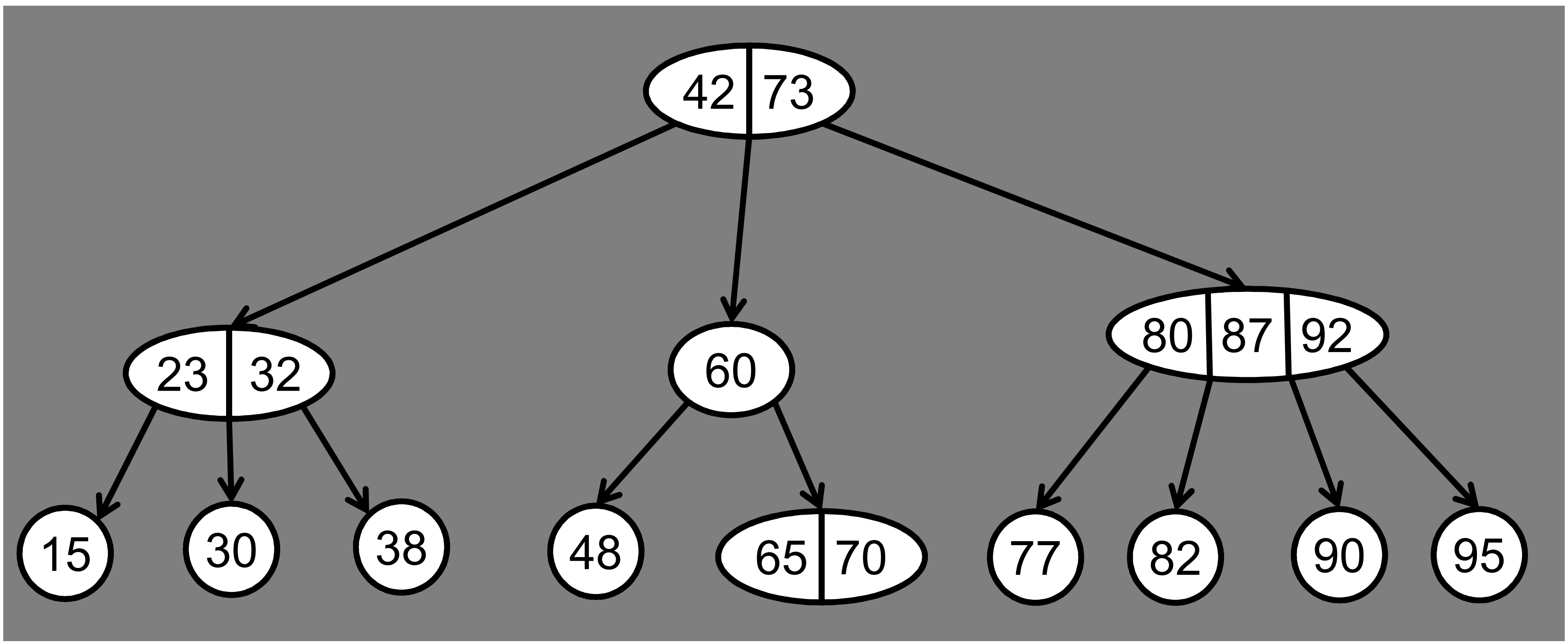
traverse T3:

15, 23, 30, 32, 38, 42, 48, 60, 65, 70, 73, 77, 80,

2-3-4 Trees

Rule #3: Every leaf has the same depth.

– Every path from root->leaf is the same length.



2-3-4 Trees

Claim: a B-tree with n keys has height O(log n)

2-3-4 Trees

Claim: a B-tree with n keys has height O(log n)

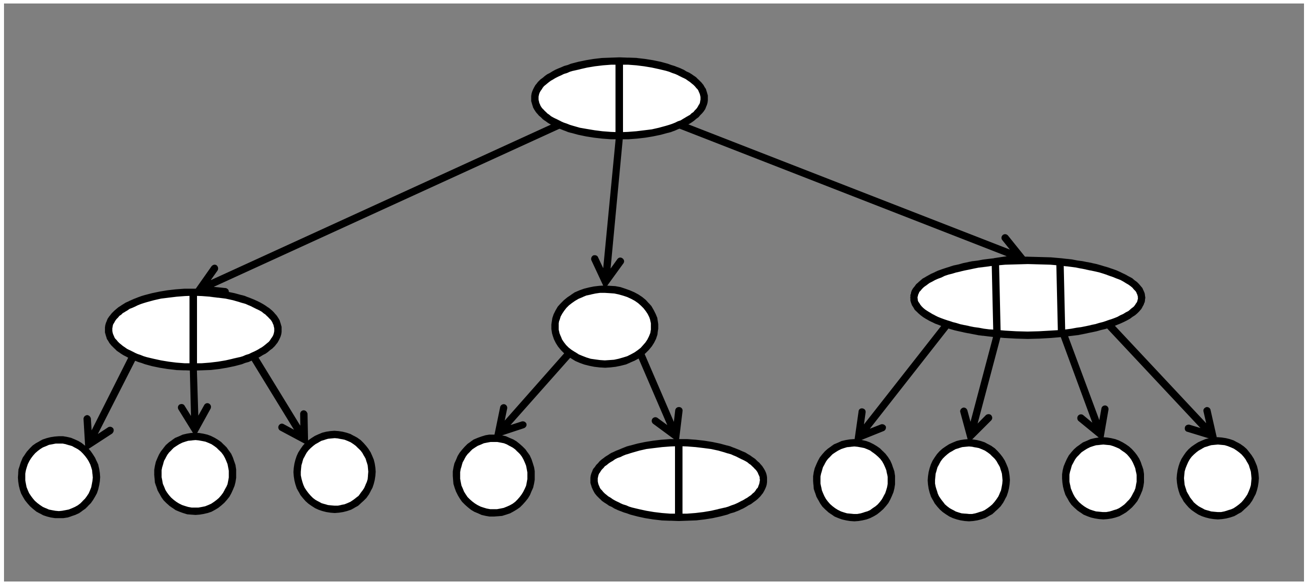
Intuition:

- Every node has at least two children.
- When the height increases by 1, the number of keys doubles.
- After log(n) levels, there are at least n keys.

2-3-4 Trees

Claim: h < c log n

Proof:



depth = 0: 1

depth = 1: 2

depth = 2: 4

depth = 3: 8

…

depth = h: 2^h

$$n \geq 1 + 2 + 4 + 8 + \ldots + 2^h = \sum_{j=0}^{h} 2^j$$

Math Intermission

Prove: $$\sum_{j=1}^{h} 2^j = 2^{h+1} - 1$$

Math Intermission

Prove: $\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$

Induction:

- Base case: h=0

$$\sum_{j=0}^{0} 2^j = 2^0 = 1 = 2^1 - 1 = 2^{h+1} - 1$$

Math Intermission

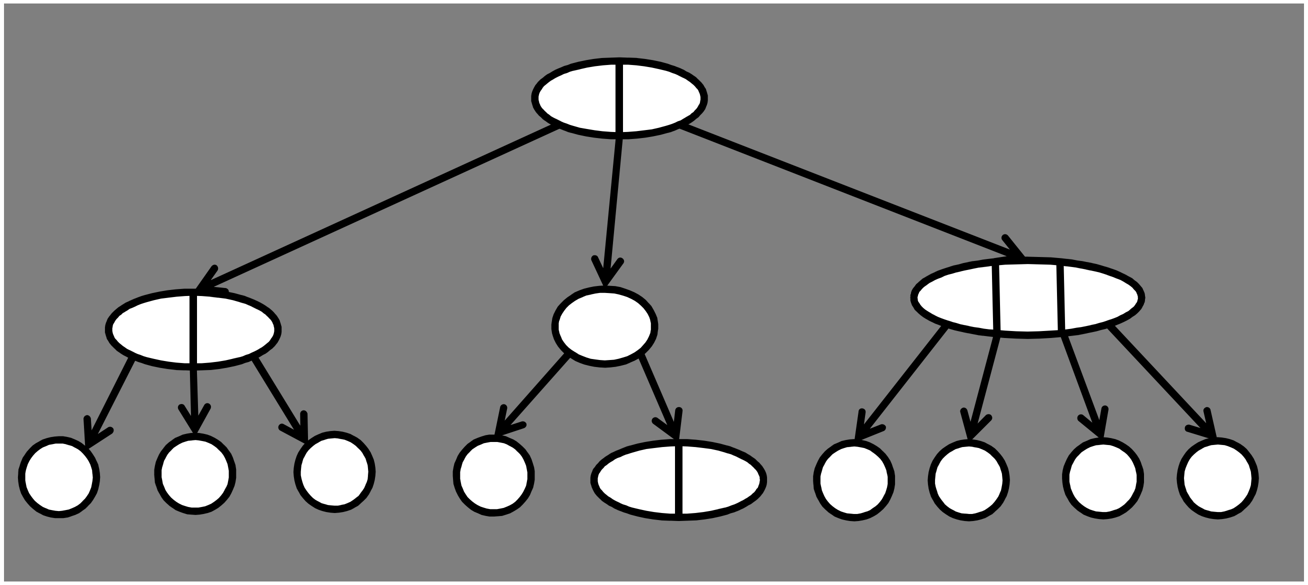
Prove: $\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$

Inductive step:
$$\begin{aligned} \sum_{j=0}^{h} 2^j &= 2^h + \sum_{j=0}^{h-1} 2^j \\ &= 2^h + \left(2^h - 1\right) \\ &= 2^{h+1} - 1 \end{aligned}$$

2-3-4 Trees

Claim: h = O(log n)

Proof:



depth = 0:   1

depth = 1:   2

depth = 2:   4

depth = 3:   8

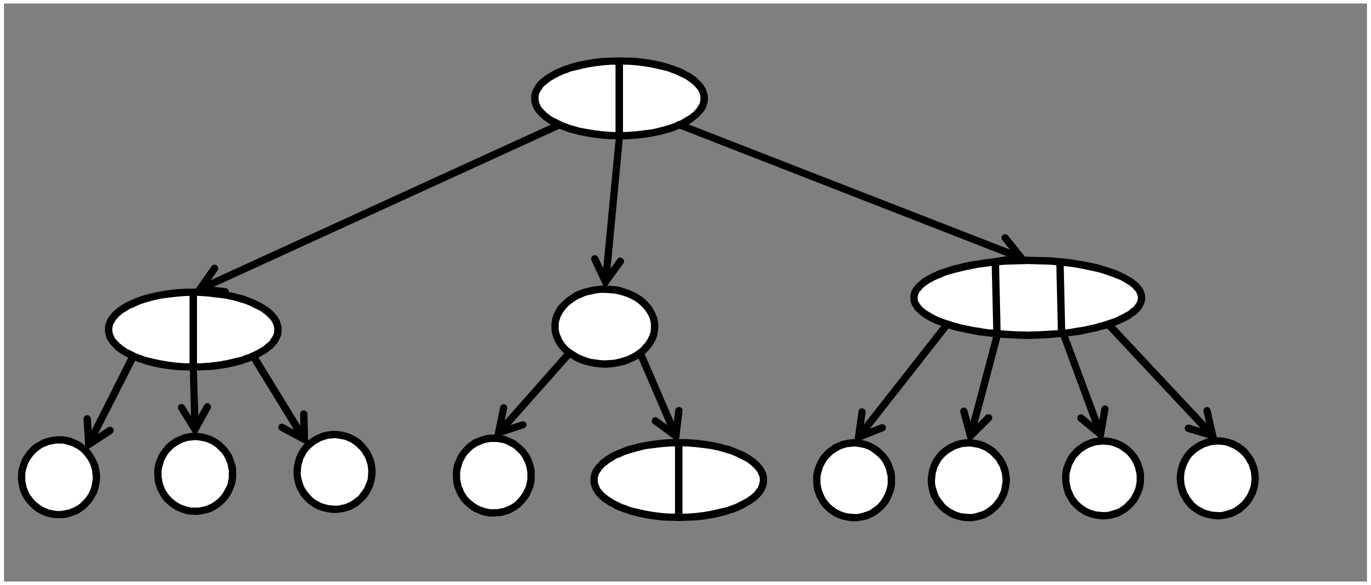
…

depth = h: 2^h

$$n \geq 1 + 2 + 4 + 8 + \ldots + 2^h = \sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$

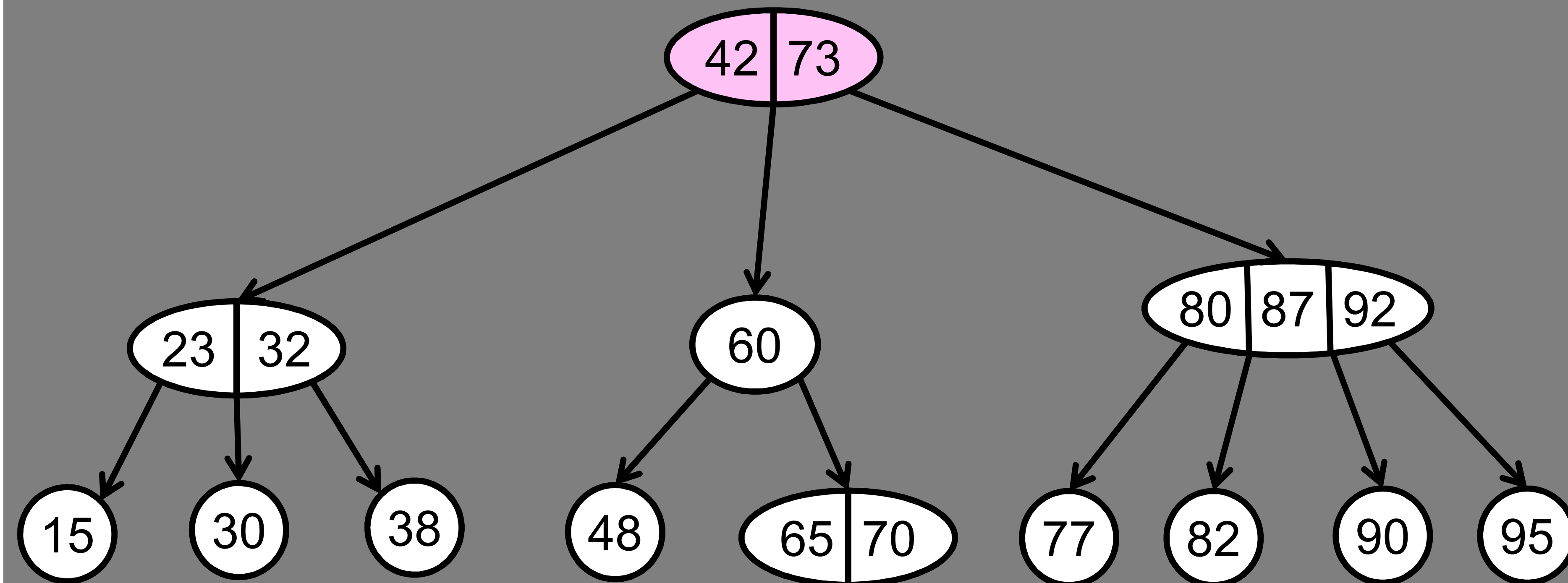
2-3-4 Trees

search: O(log n)

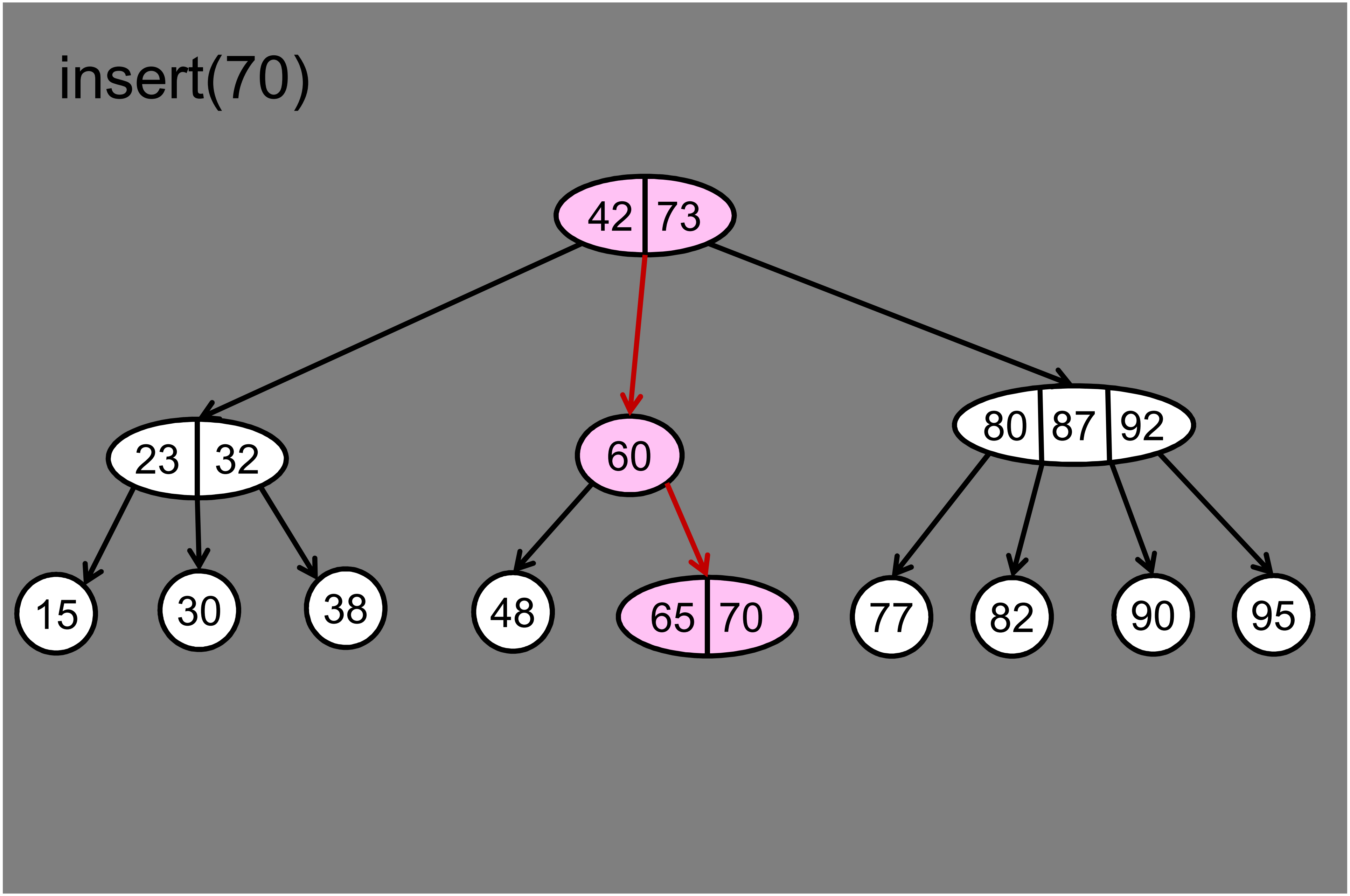


2-3-4 Trees: Inserting

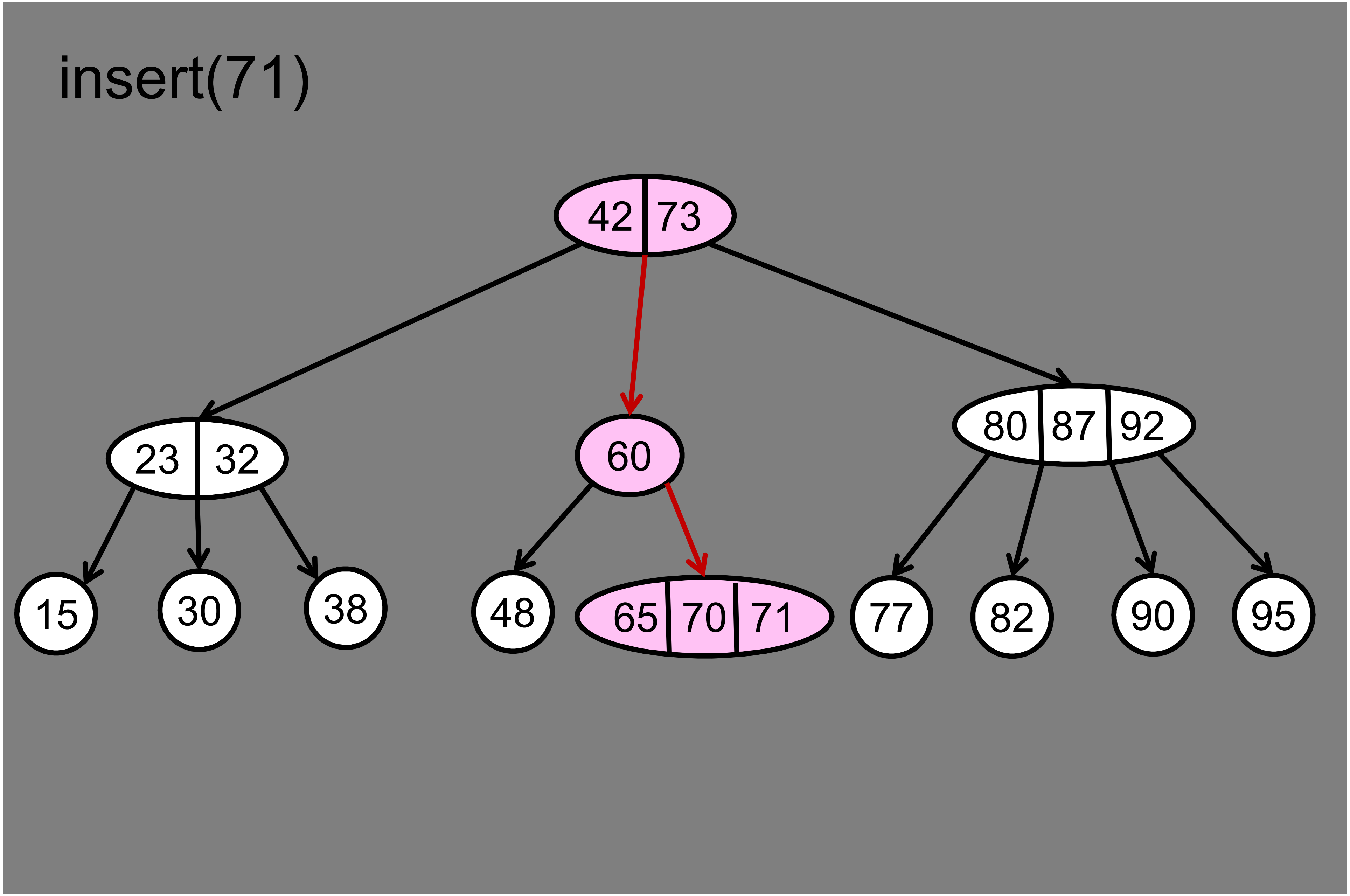
insert(70)



2-3-4 Trees: Inserting

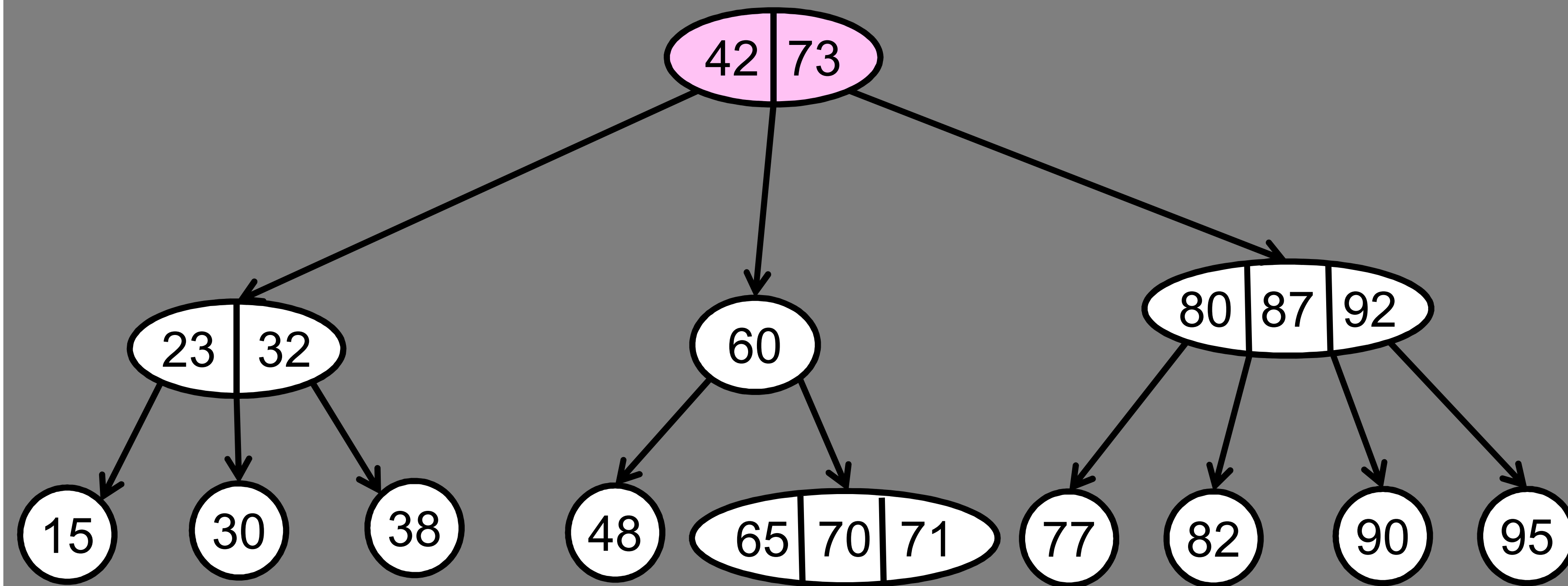


2-3-4 Trees: Inserting



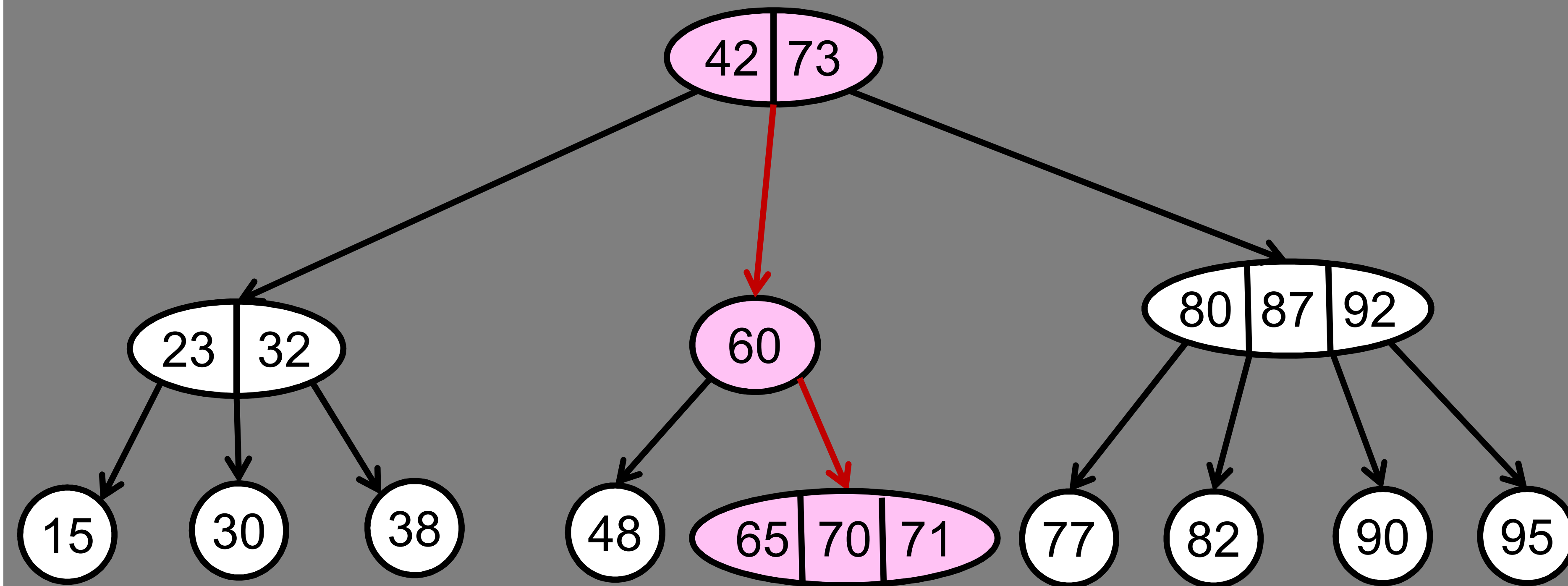
2-3-4 Trees: Inserting

insert(72)



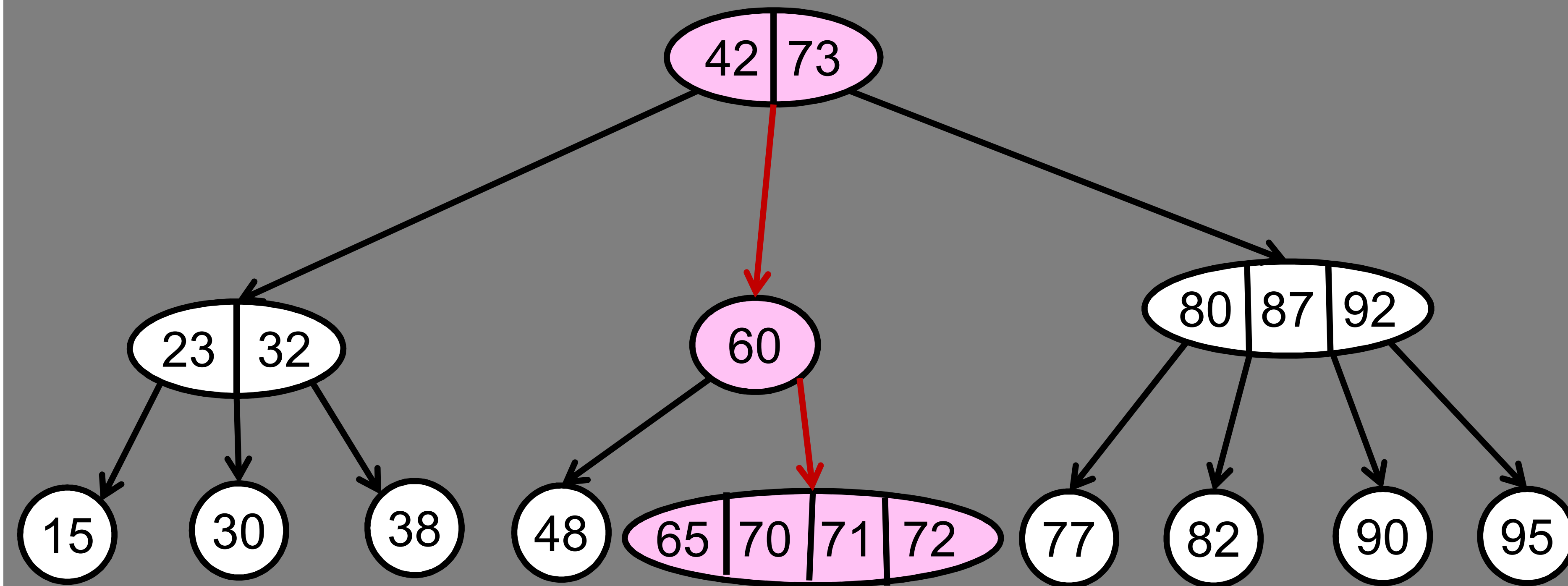
2-3-4 Trees: Inserting

insert(72)



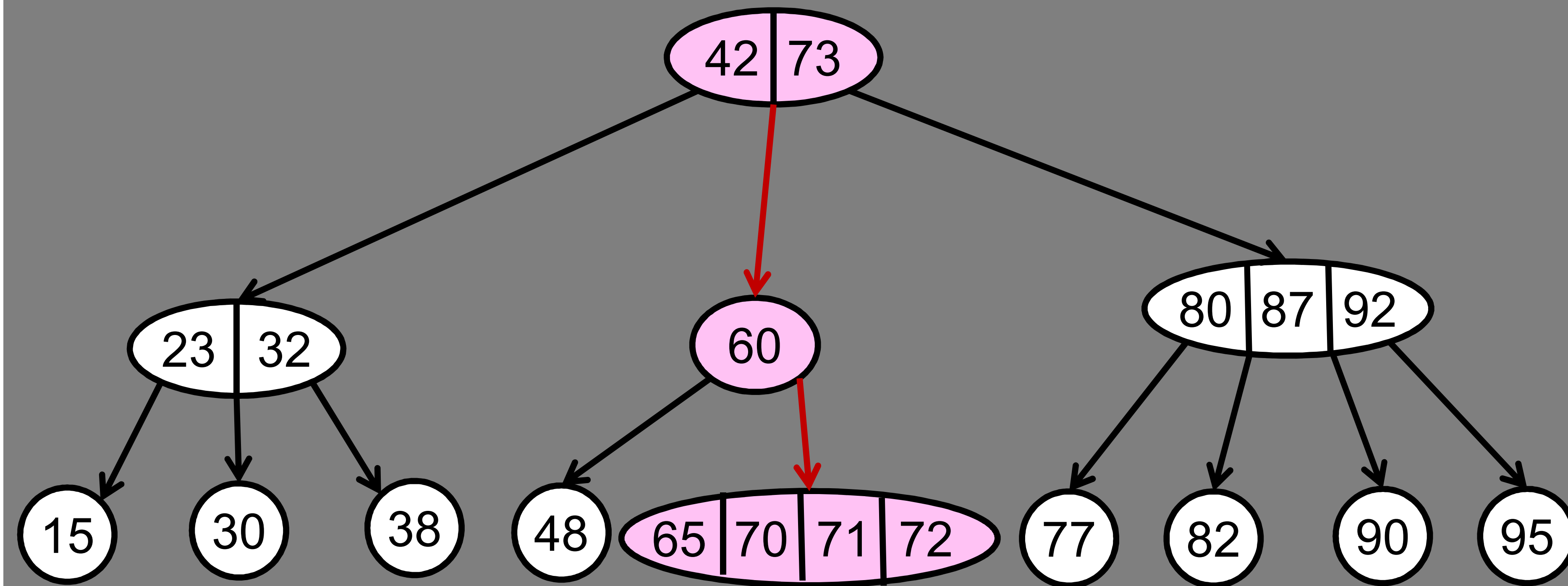
2-3-4 Trees: Inserting

insert(72)



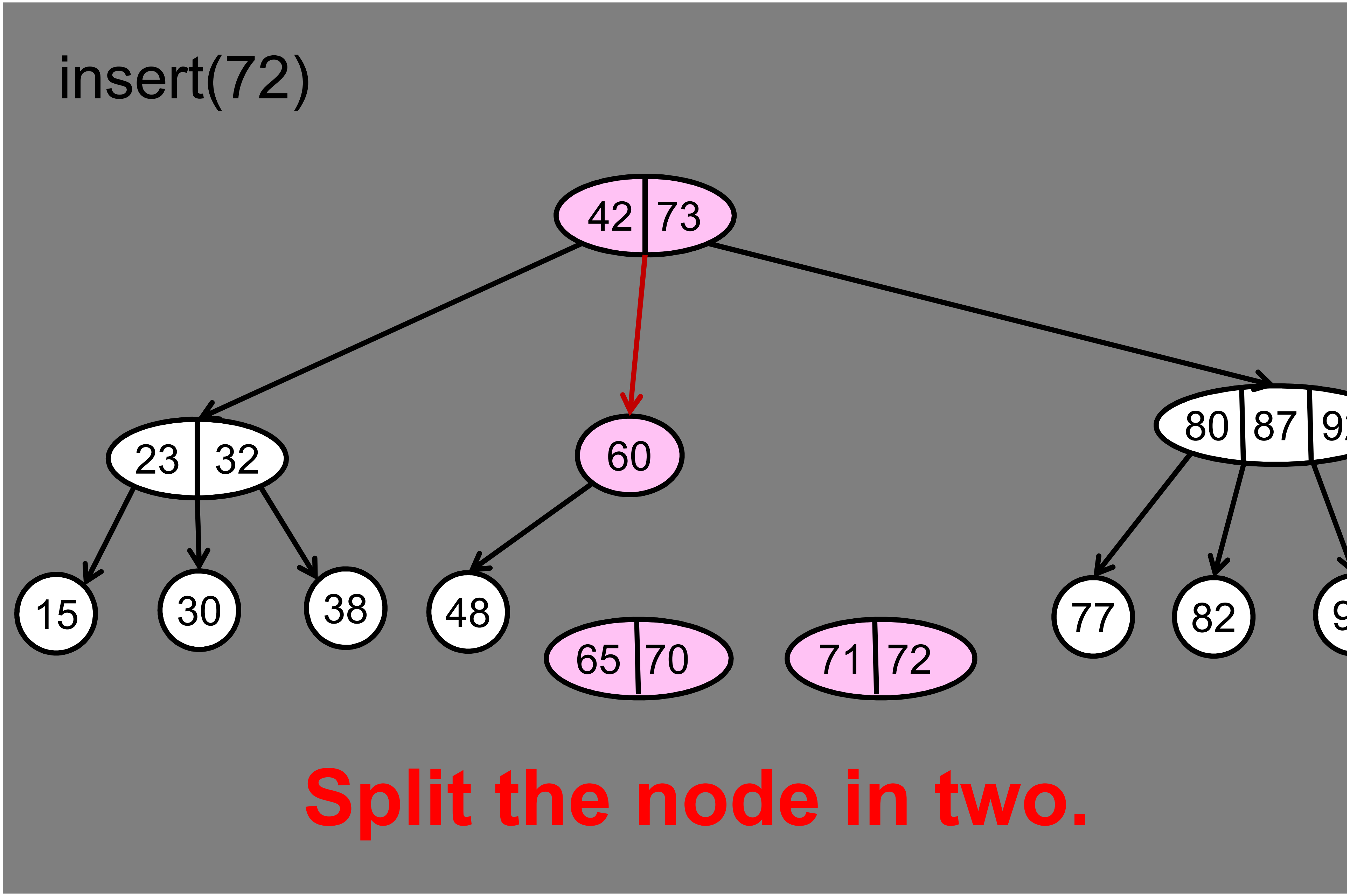
2-3-4 Trees: Inserting

insert(72)

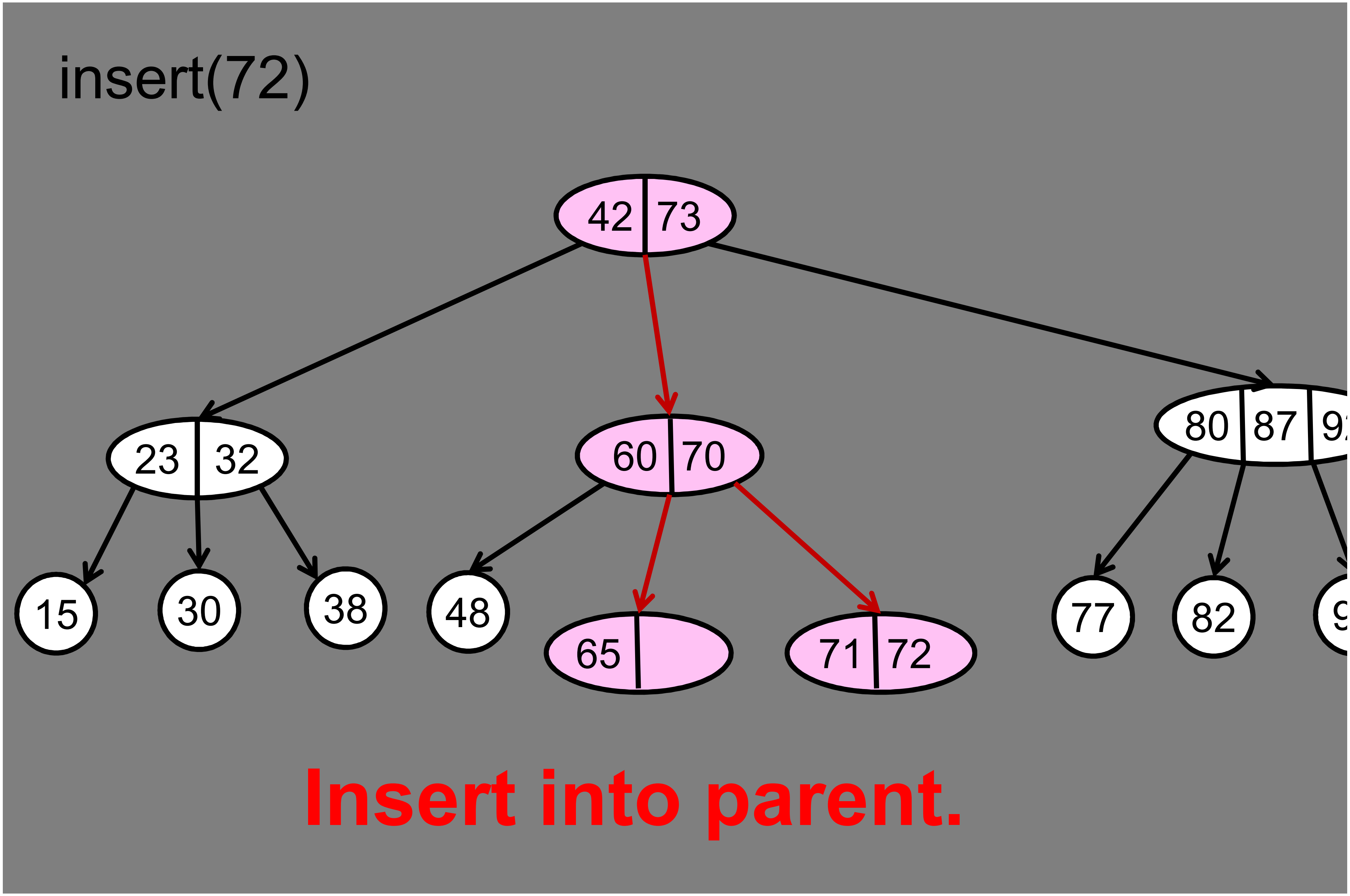


Too many keys!!!

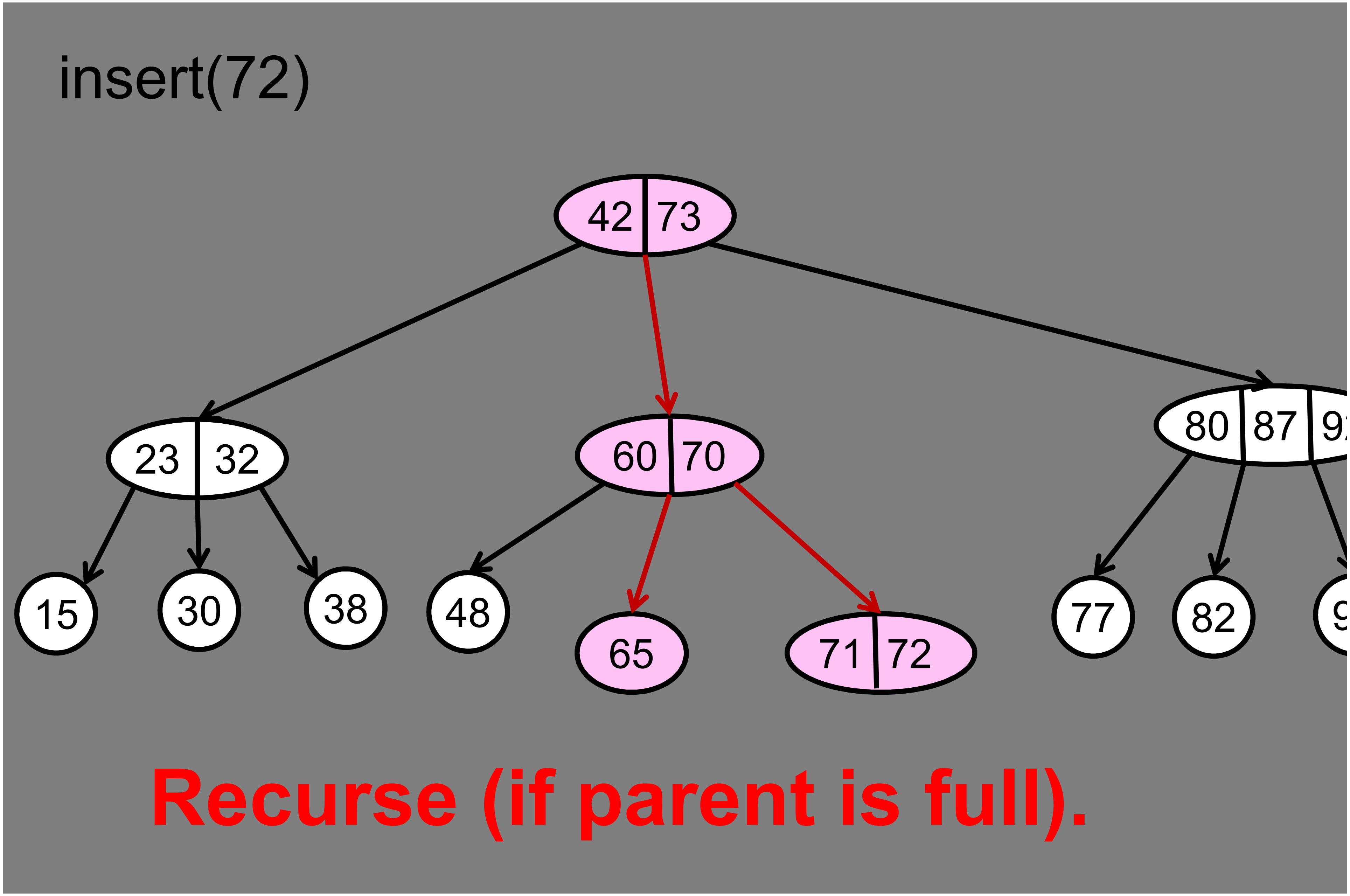
2-3-4 Trees: Inserting



2-3-4 Trees: Inserting

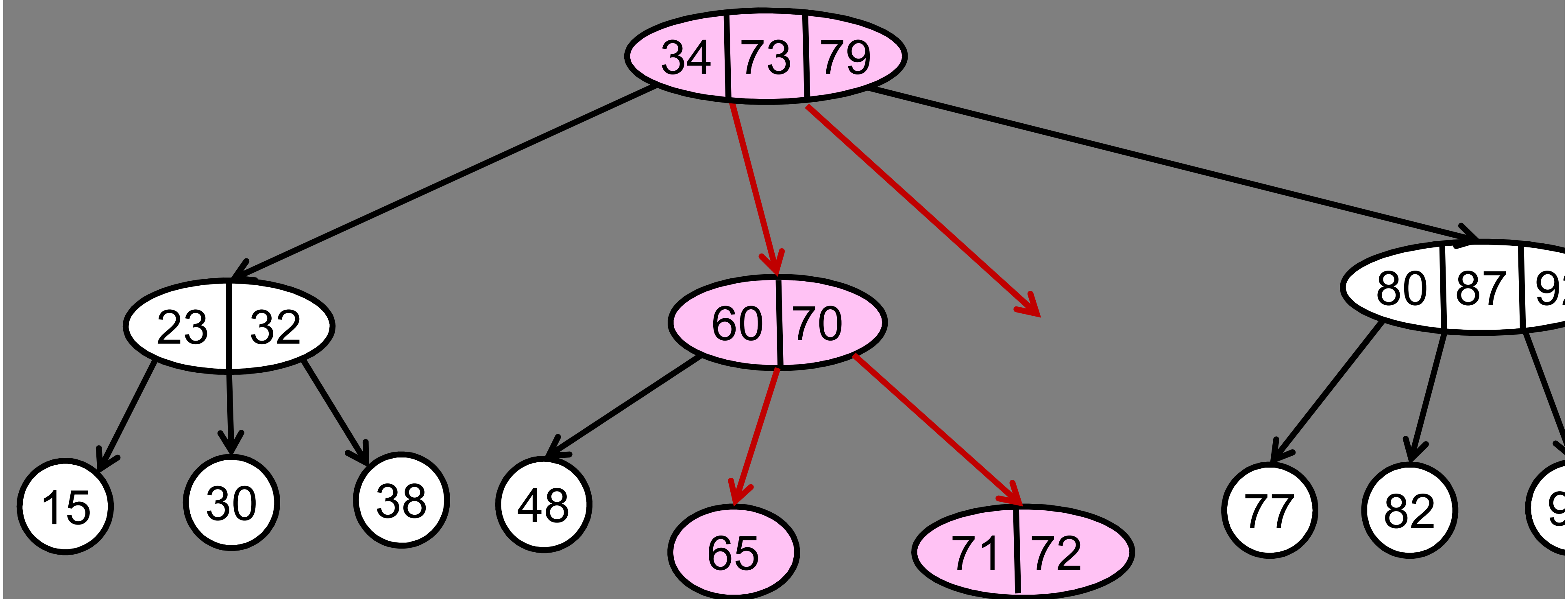


2-3-4 Trees: Inserting



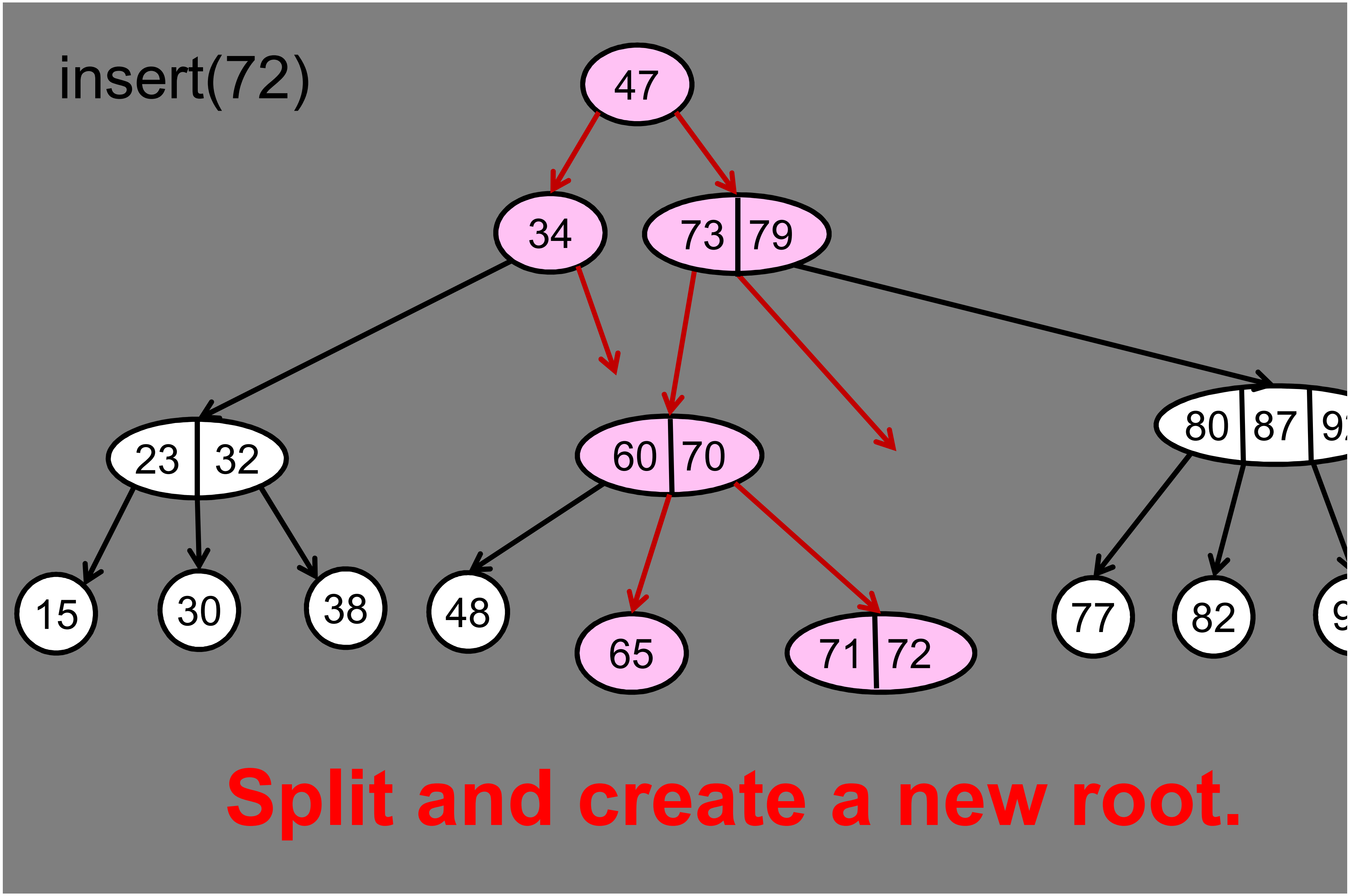
2-3-4 Trees: Inserting

insert(72)



What if the root is full?

2-3-4 Trees: Inserting

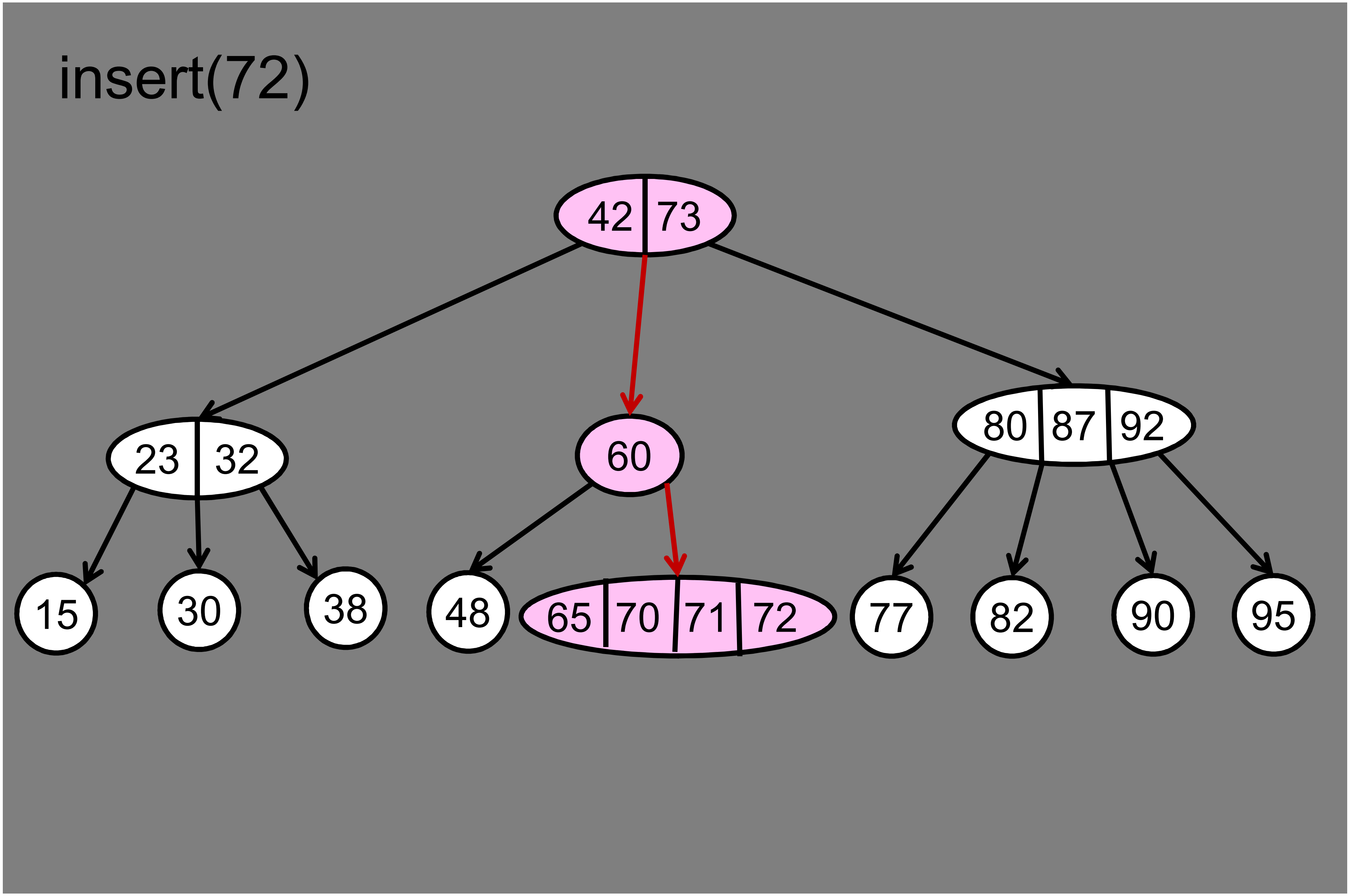


2-3-4 Trees

insert(key)

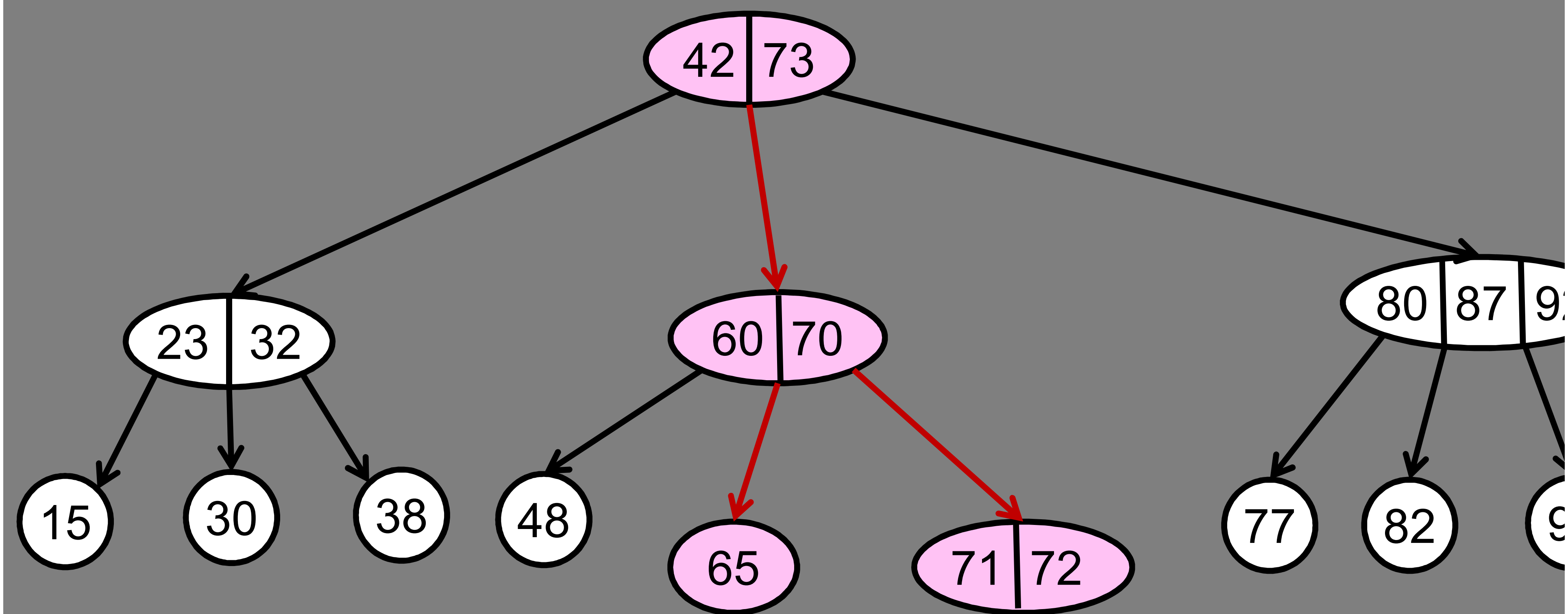
1. Search for leaf **x** to start insert.
2. Repeat until done:
3. Insert **key** into **x**.
4. If **x** overfull:
5. **key** = median(**x**);
6. remove **key** from **x** and split(**x**);
7. If (x == root) then create new root.
8. Else **x** = parent(**x**)
9. Else done;

2-3-4 Trees: Inserting



2-3-4 Trees: Inserting

insert(72)



2-3-4 Trees

Preserves 3 Properties:

1. Every node has 2 or 3 or 4 children.
2. Search tree property.
3. All leaves have the same depth.

2-3-4 Trees

Lazy option: do splitting when needed.

Proactive option: split in advance.

One pass insertion:

- If root contains 3 keys, split root and create new root (containing one key).
- While searching for the leaf, split any node that is full (i.e., contains 3 keys).
- On arrival at leaf, there is enough space in the leaf to add the key!

2-3-4 Trees

x.insert(key):   (assume x *not* full)

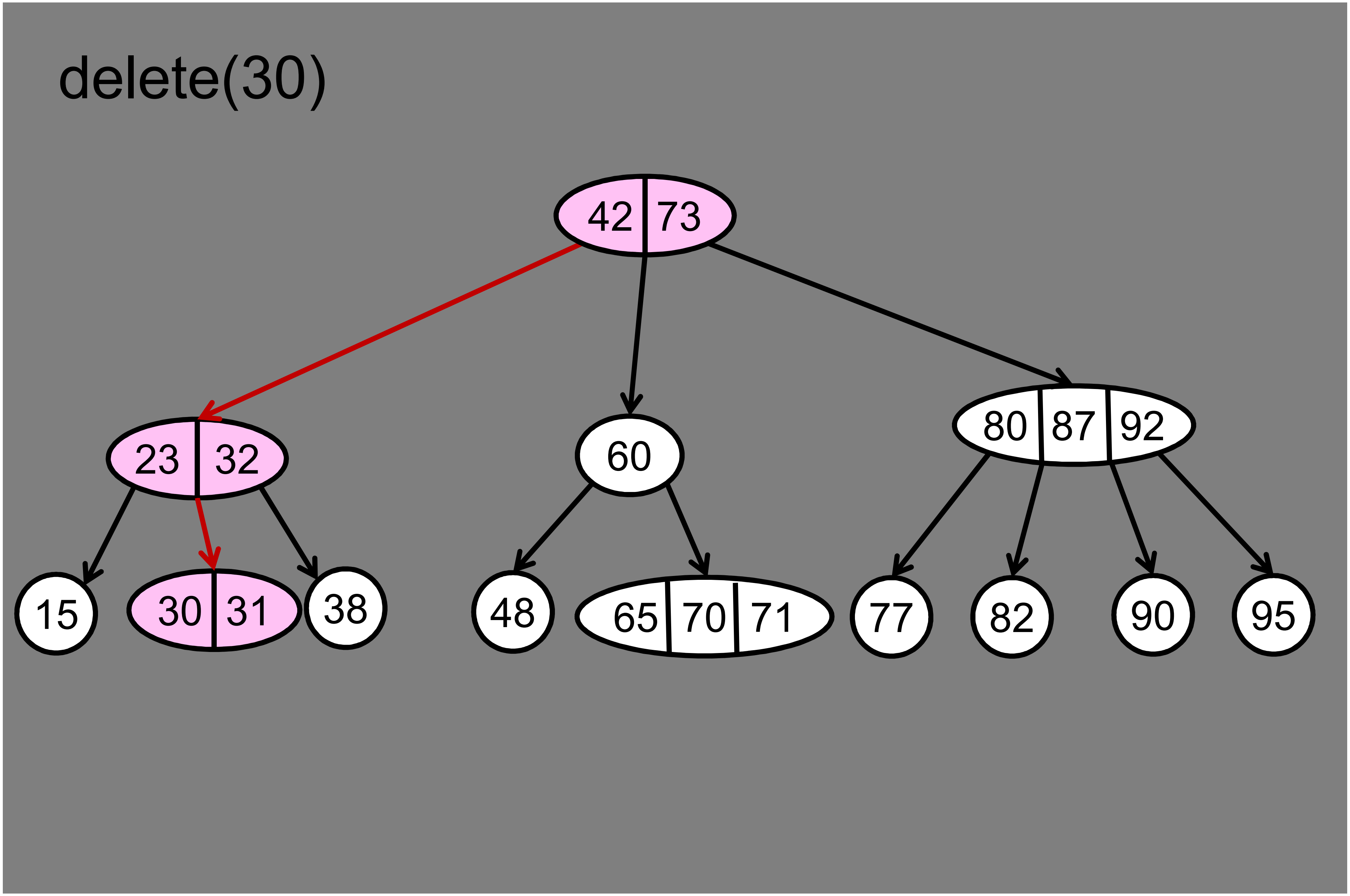
1. if leaf() then add key to array of keys.
2. else
3.   let j be the insertion point in x;
4.   if Tj has 3 keys, then
5.   split(x, j, Tj)
6.   if (key > kj) then j++;
7.   Tj.insert(key)

2-3-4 Trees

x.delete(key): (assume x has at least 2 keys)

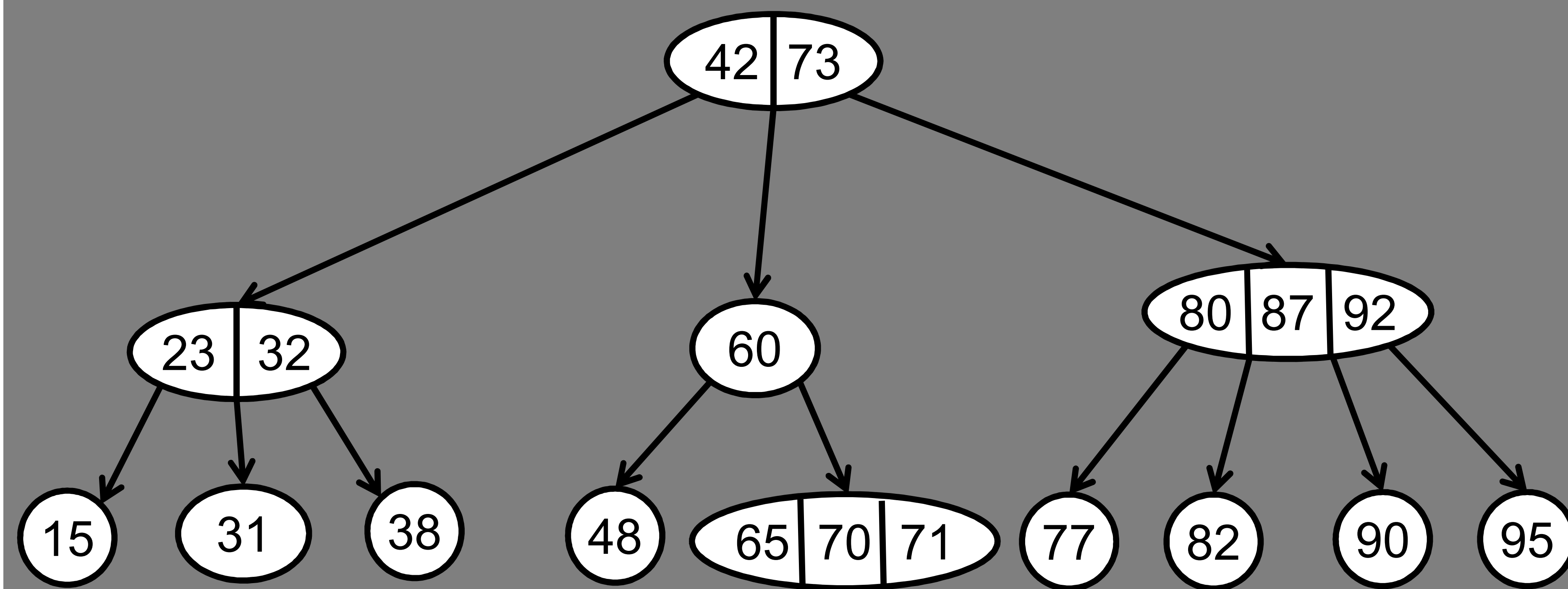
1. if leaf() then remove key
2. else if (key found at x in key kj)
3. if T(j) and T(j+1) have 1 key each:
4. merge T(j) and T(j+1) and delete kj
5. else replace kj with successor in T(j+1) or predecessor in T(j)
6. else if (key in Tj and Tj has at least 2 keys)
7. Tj.delete(key)
8. else …

2-3-4 Trees: Deleting (leaf)

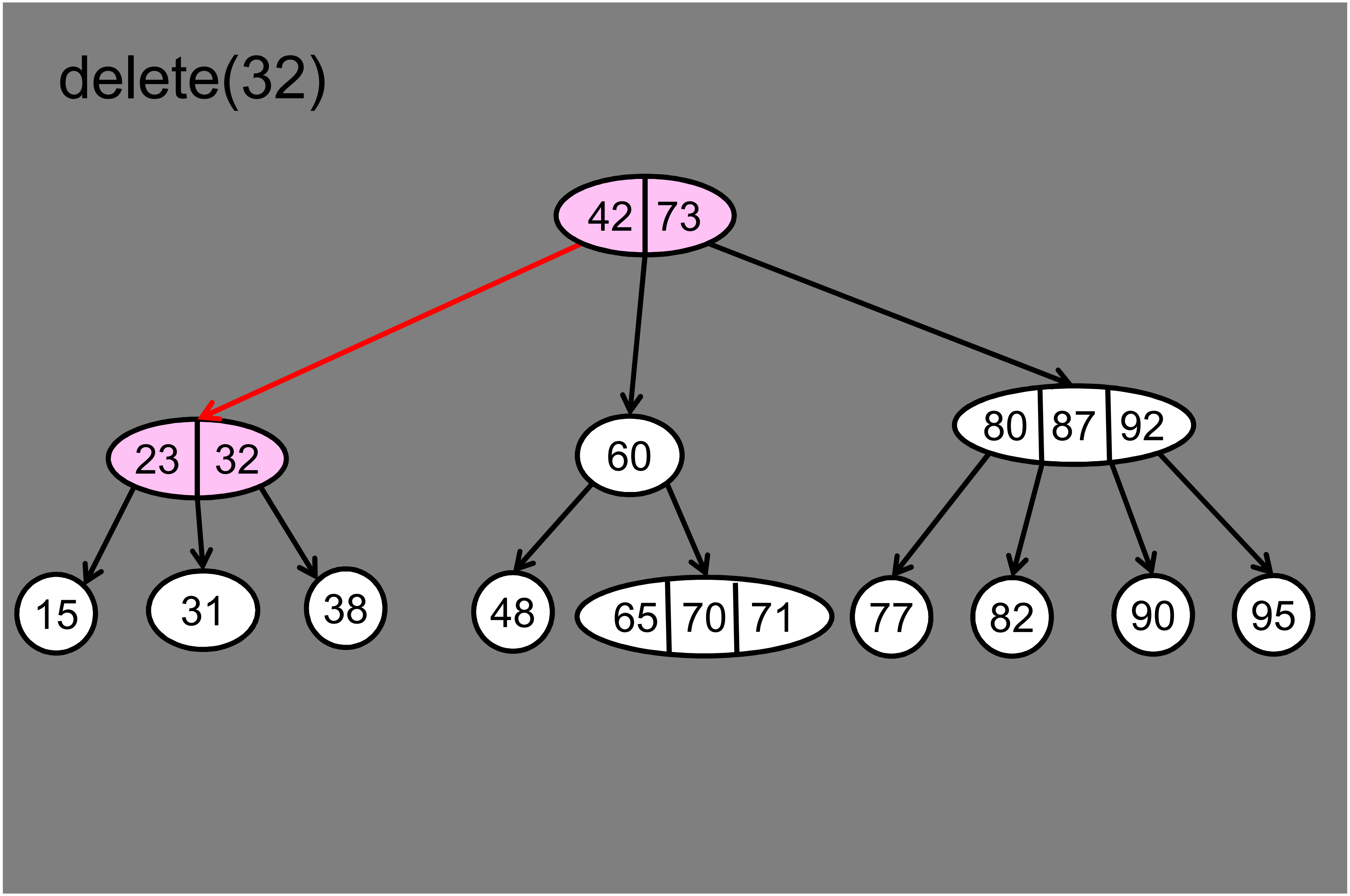


2-3-4 Trees: Deleting (leaf)

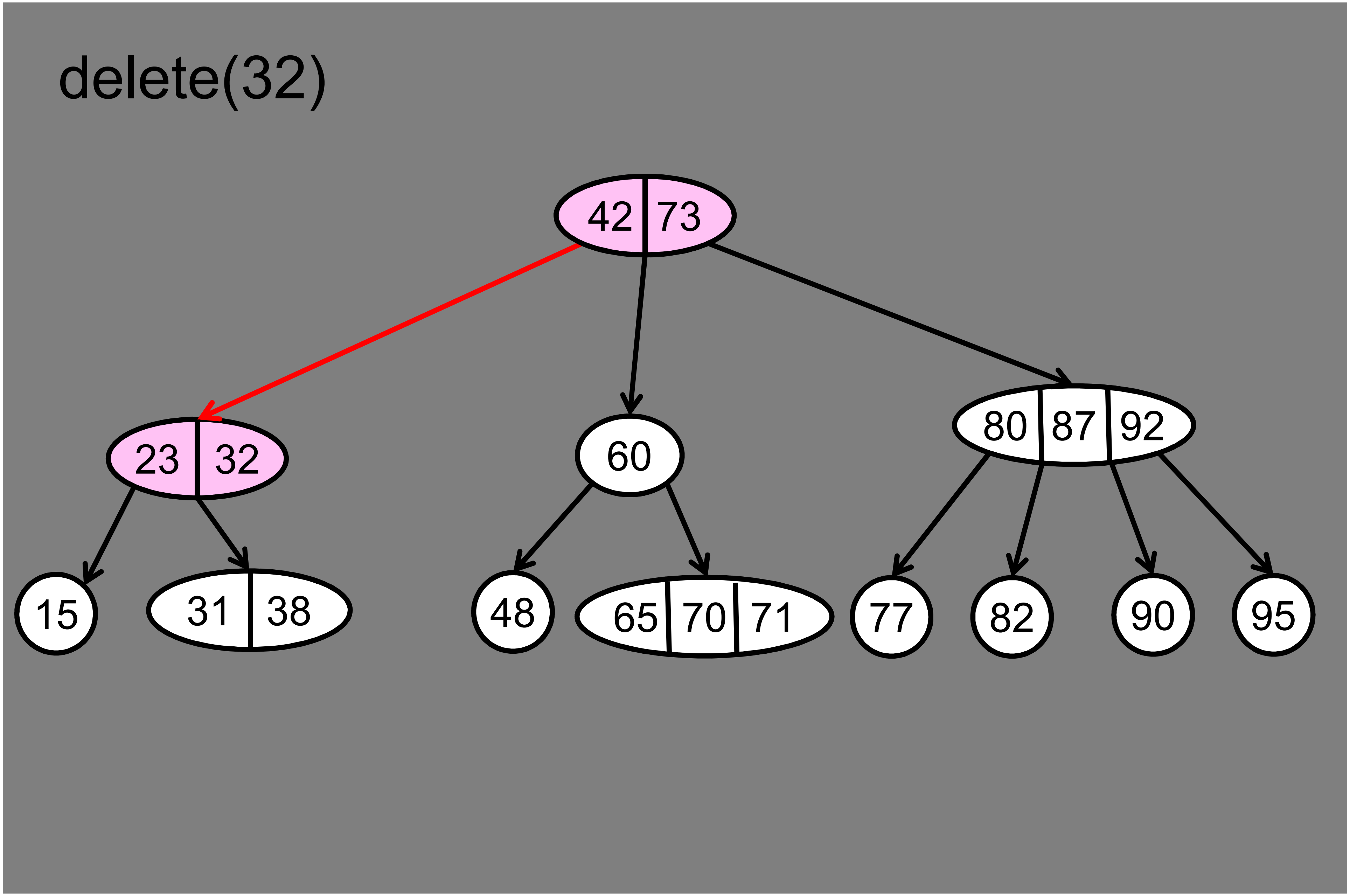
delete(30)



2-3-4 Trees: Deleting (mid-tree)

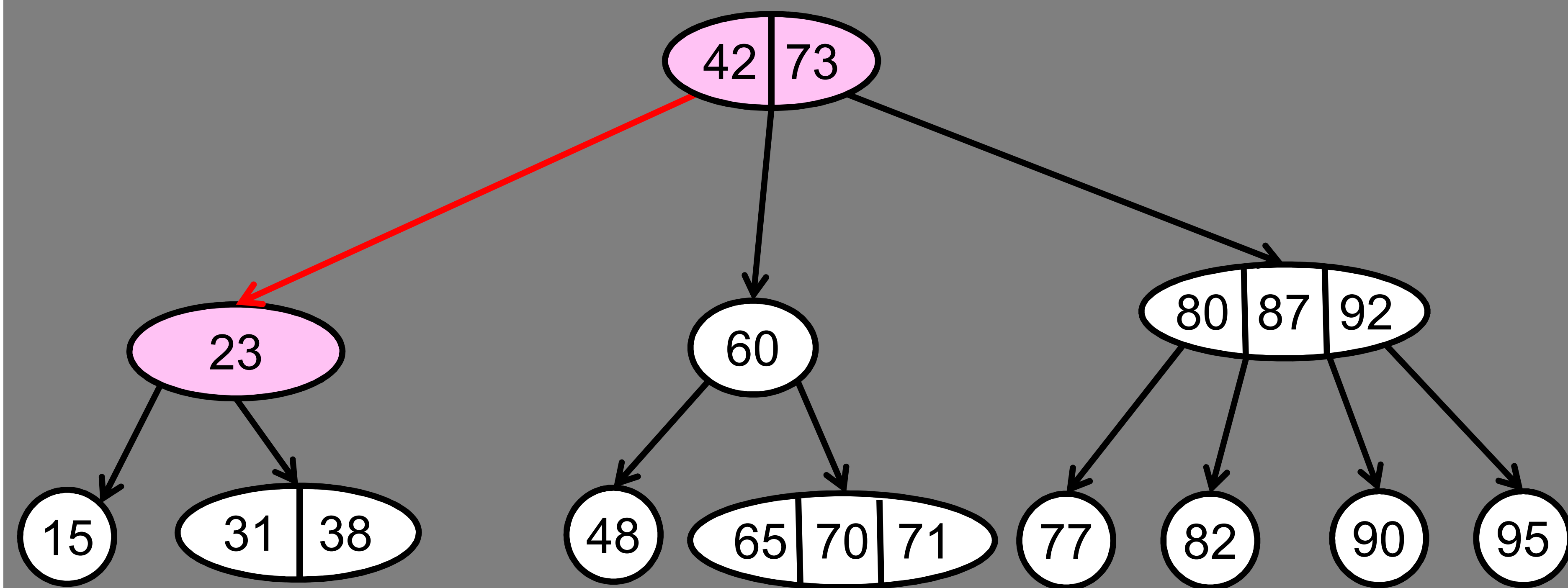


2-3-4 Trees: Deleting (mid-tree)

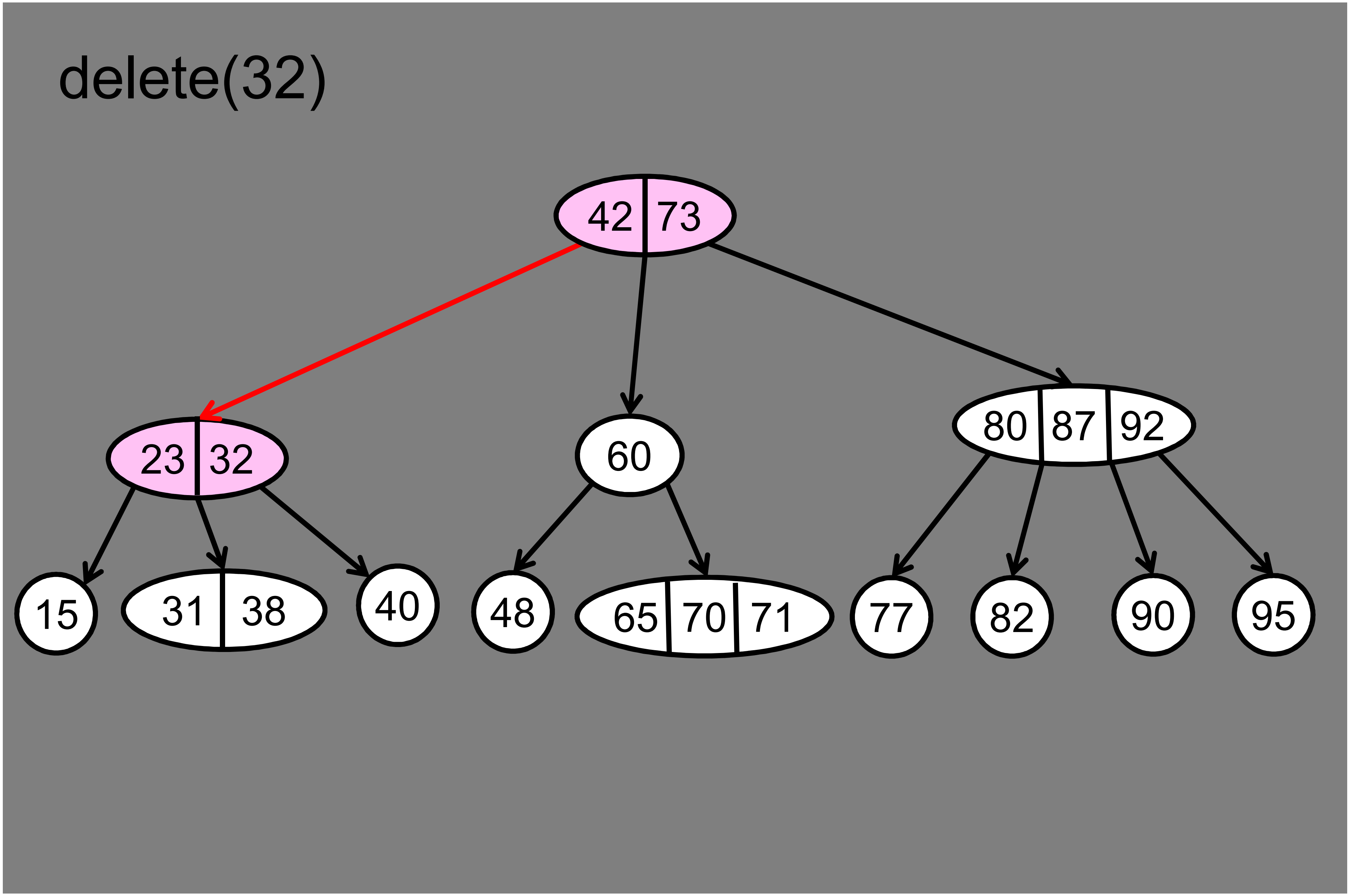


2-3-4 Trees: Deleting (mid-tree)

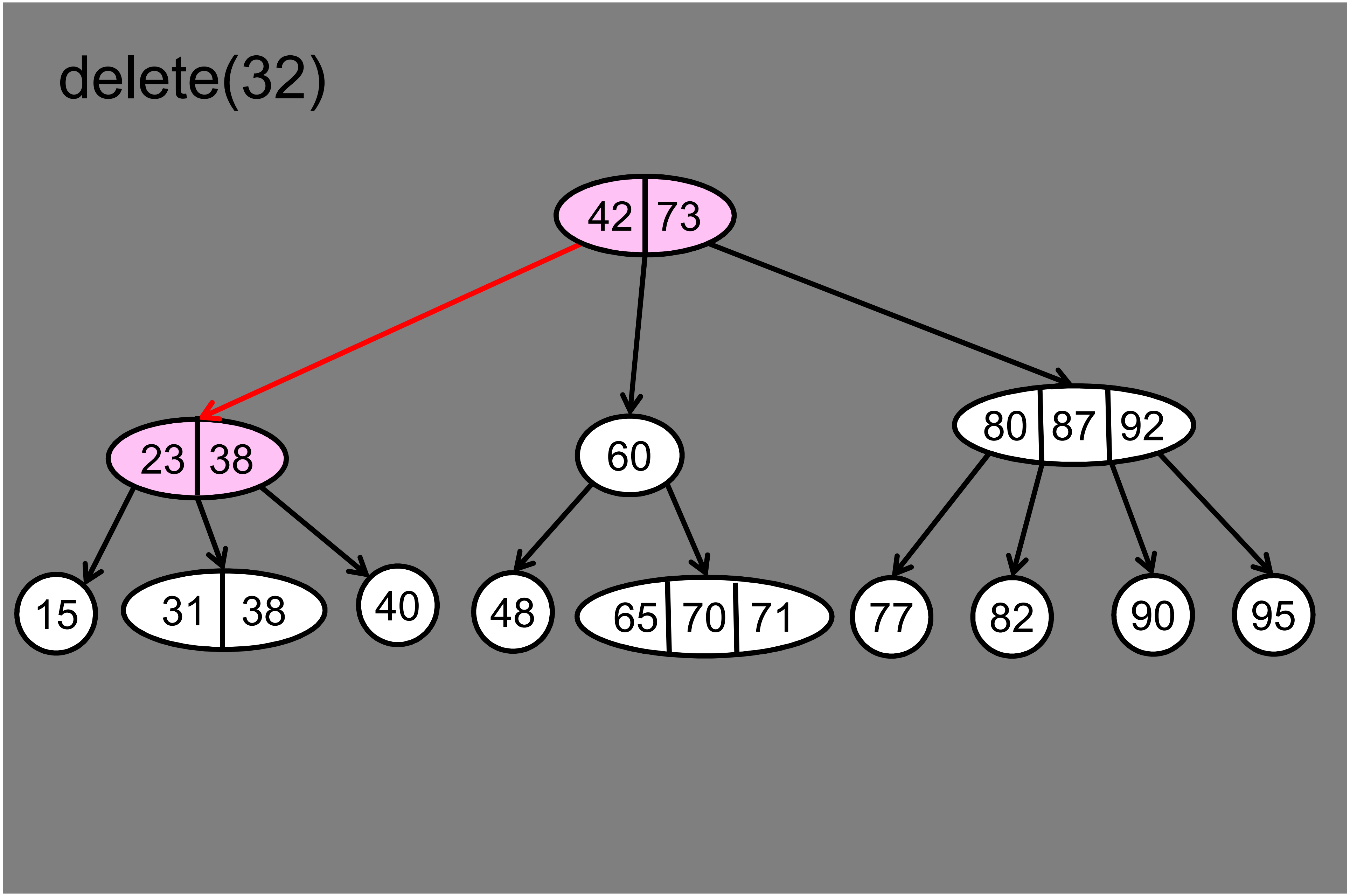
delete(32)



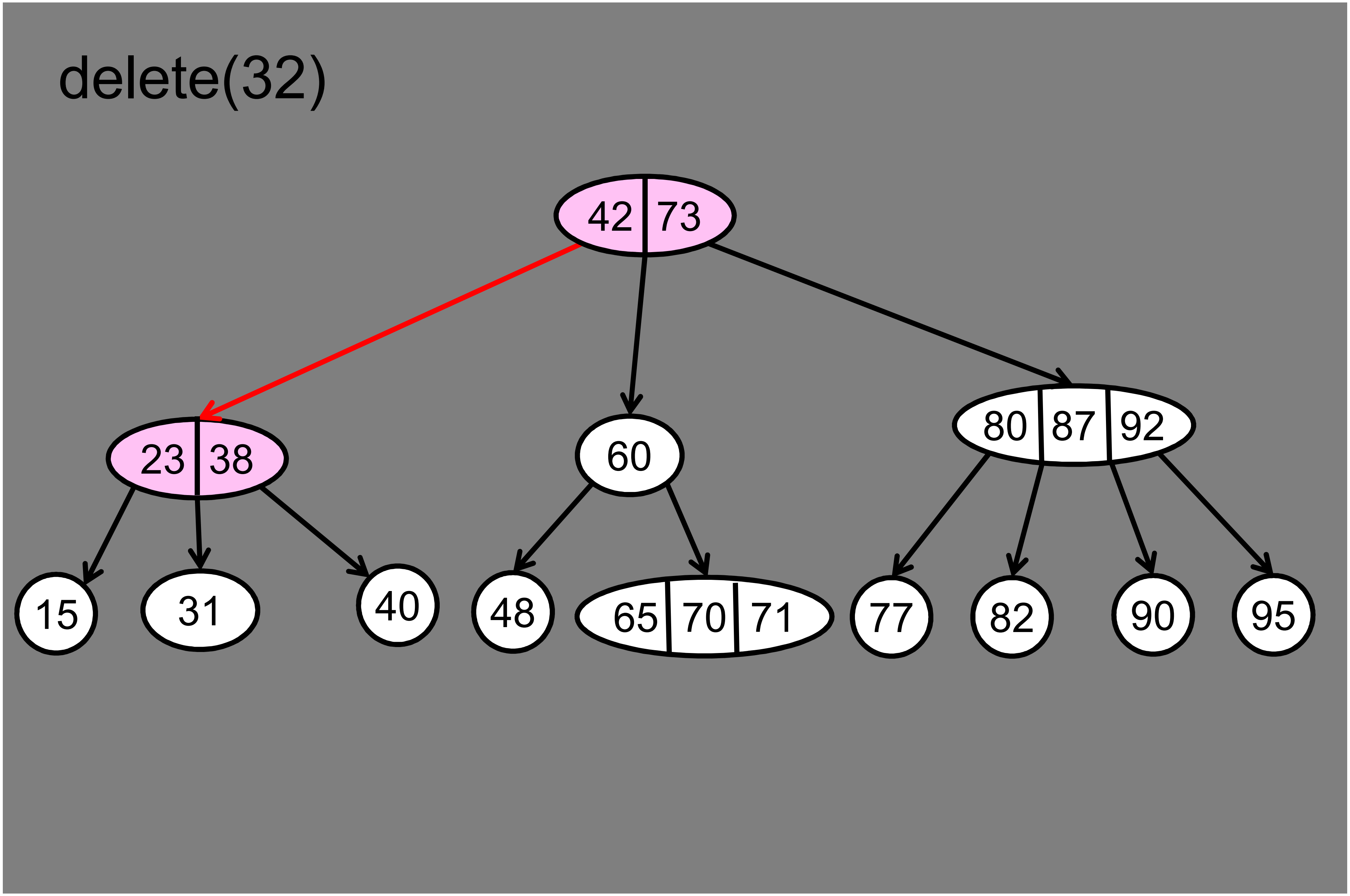
2-3-4 Trees: Deleting (mid-tree)



2-3-4 Trees: Deleting (mid-tree)

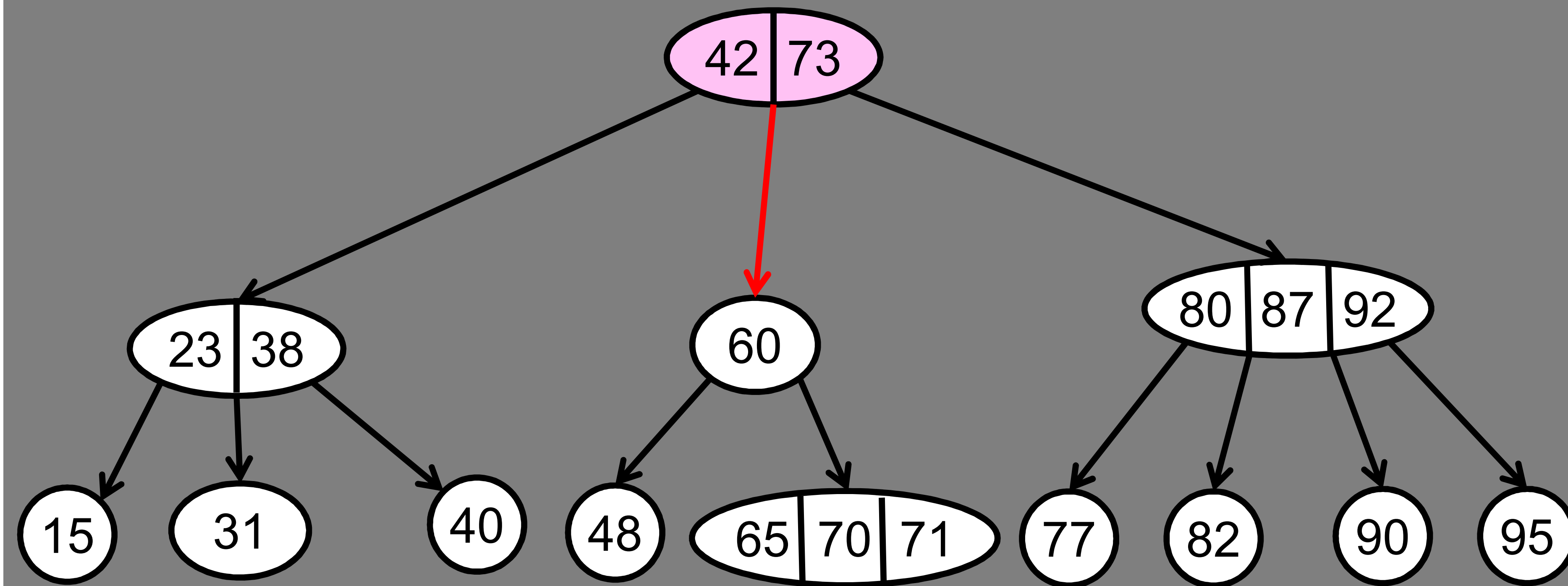


2-3-4 Trees: Deleting (mid-tree)

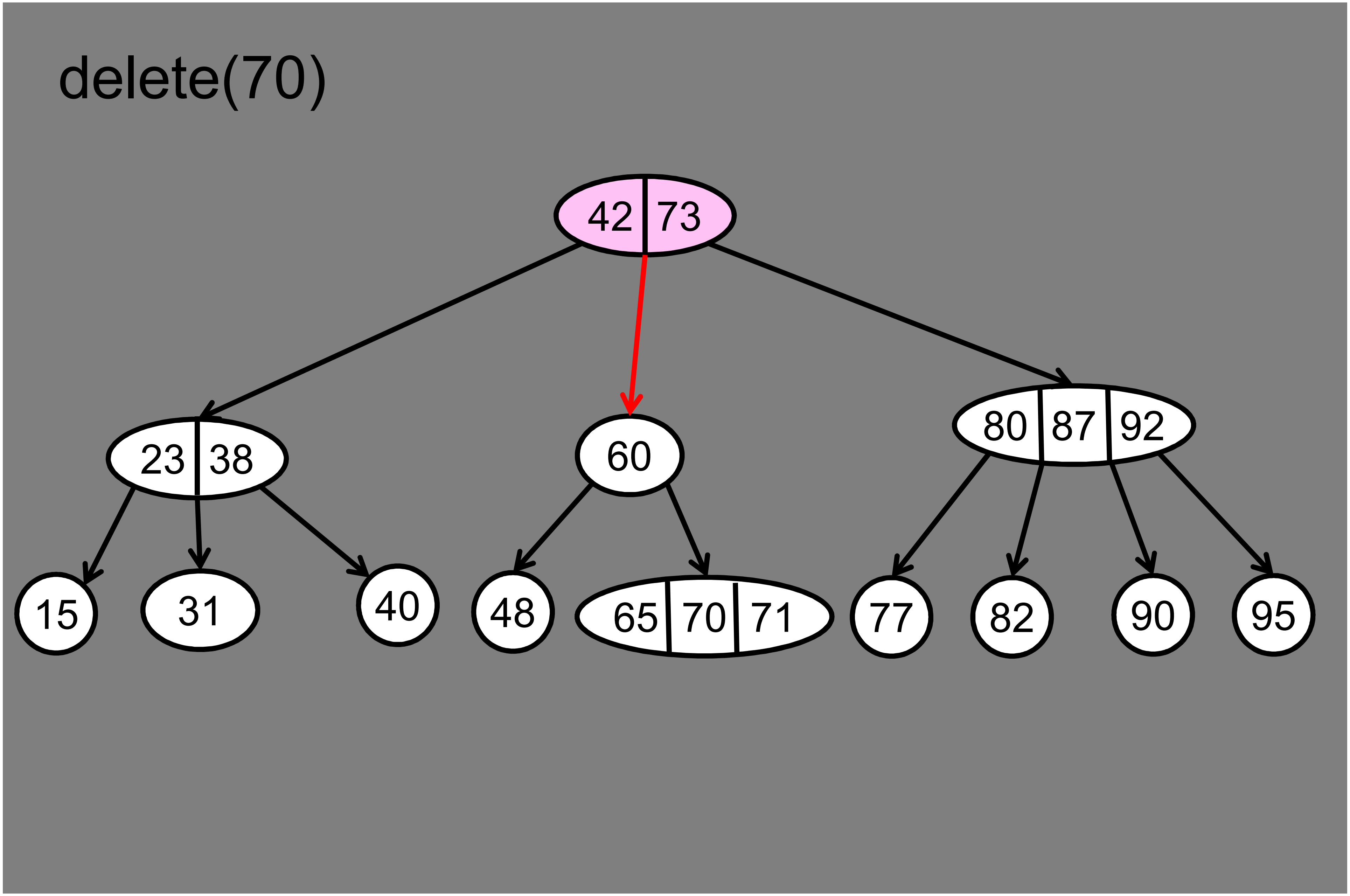


2-3-4 Trees: Deleting (hard case)

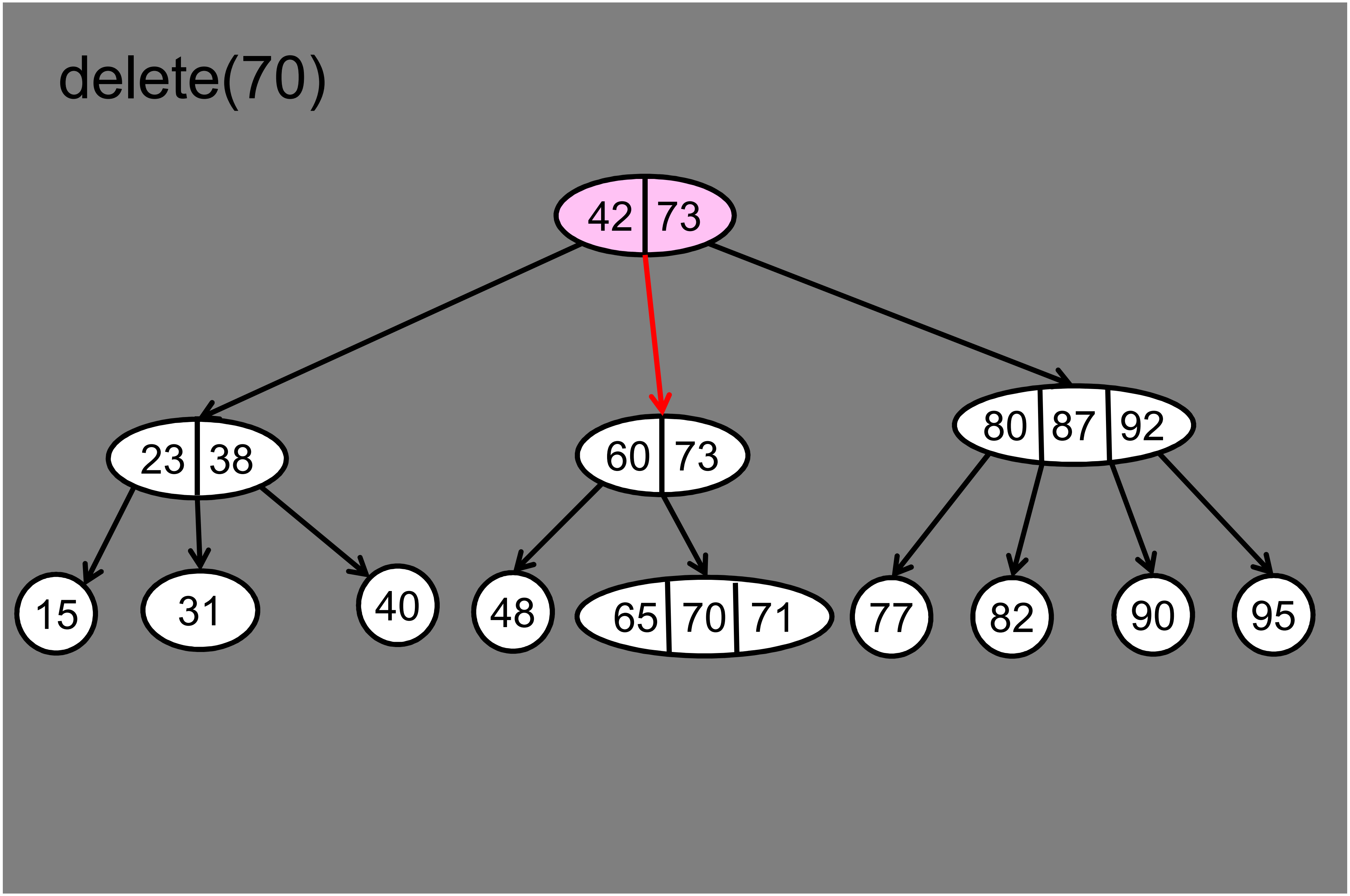
delete(70)



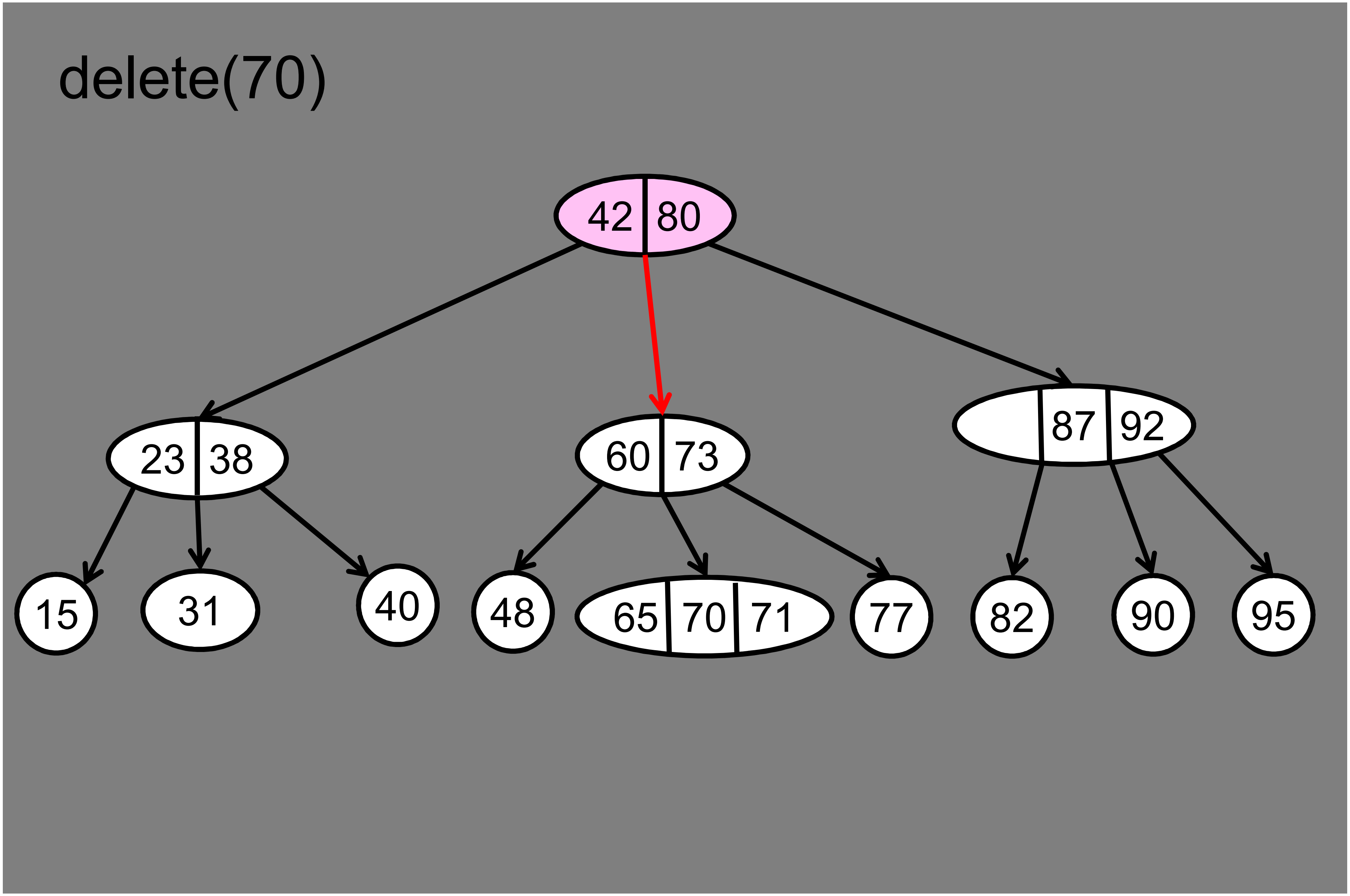
2-3-4 Trees: Deleting (hard case)



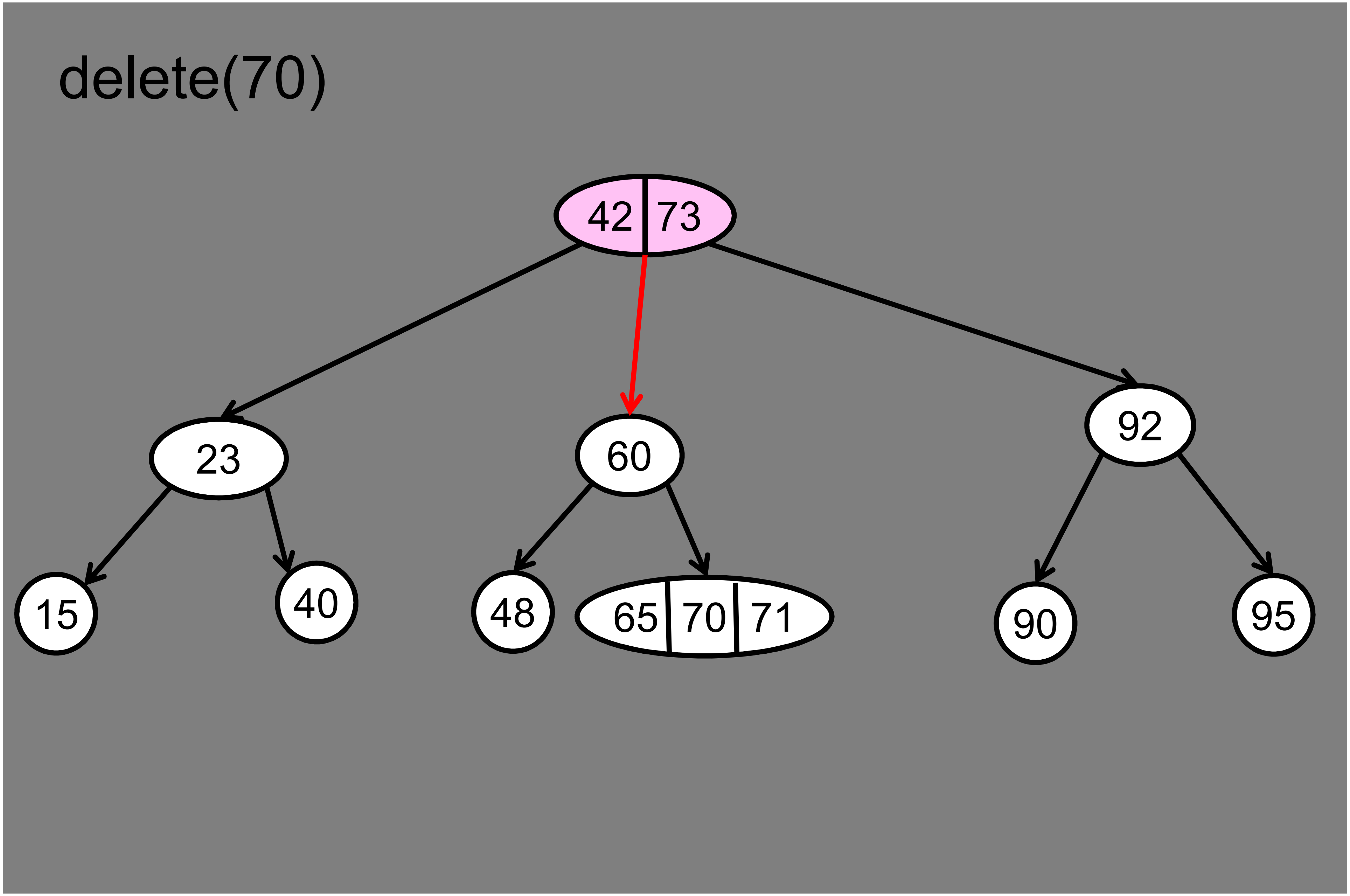
2-3-4 Trees: Deleting (hard case)



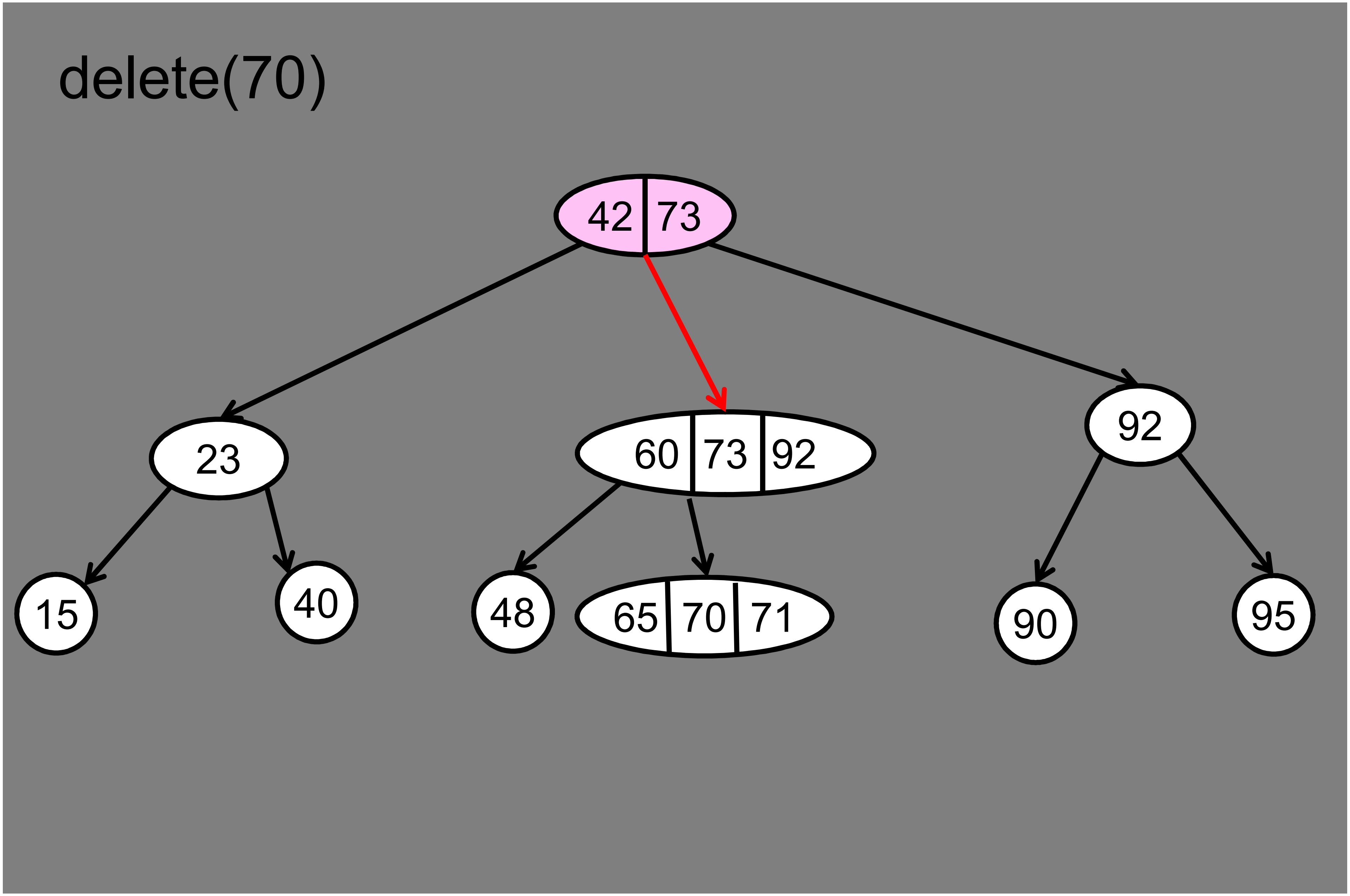
2-3-4 Trees: Deleting (hard case)



2-3-4 Trees: Deleting (hard case 2)

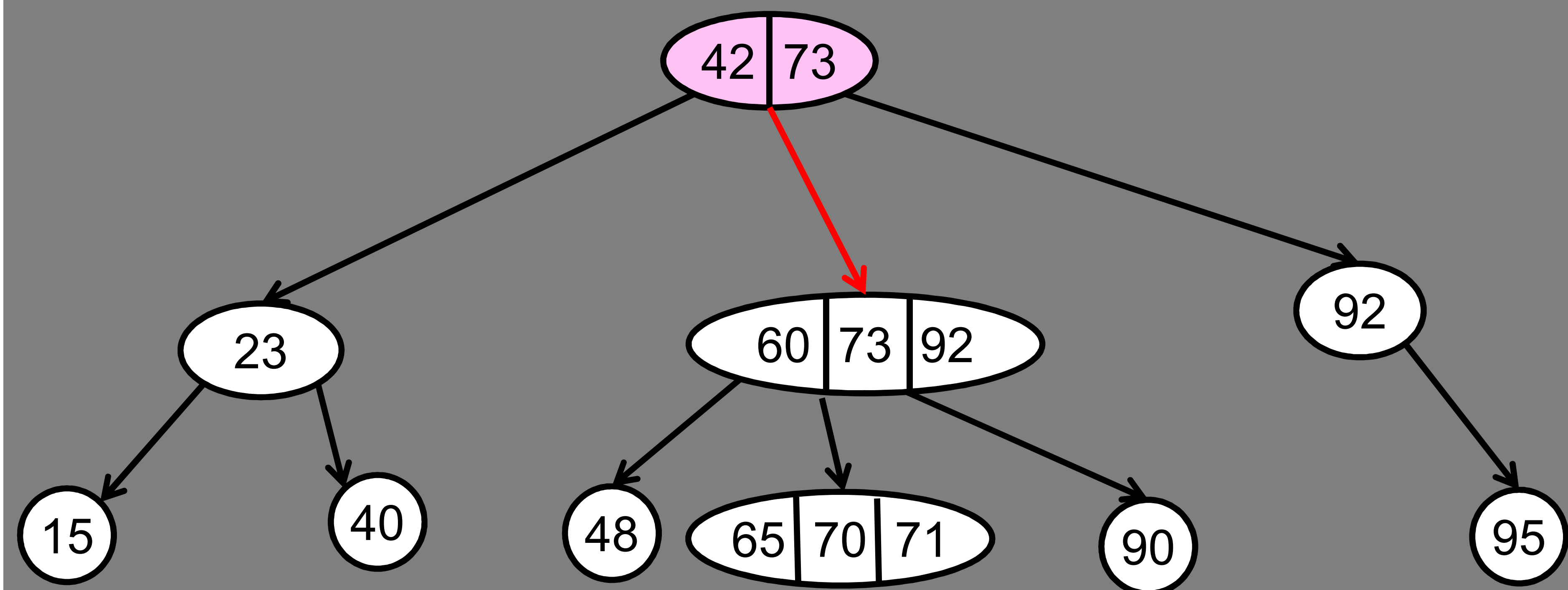


2-3-4 Trees: Deleting (hard case 2)

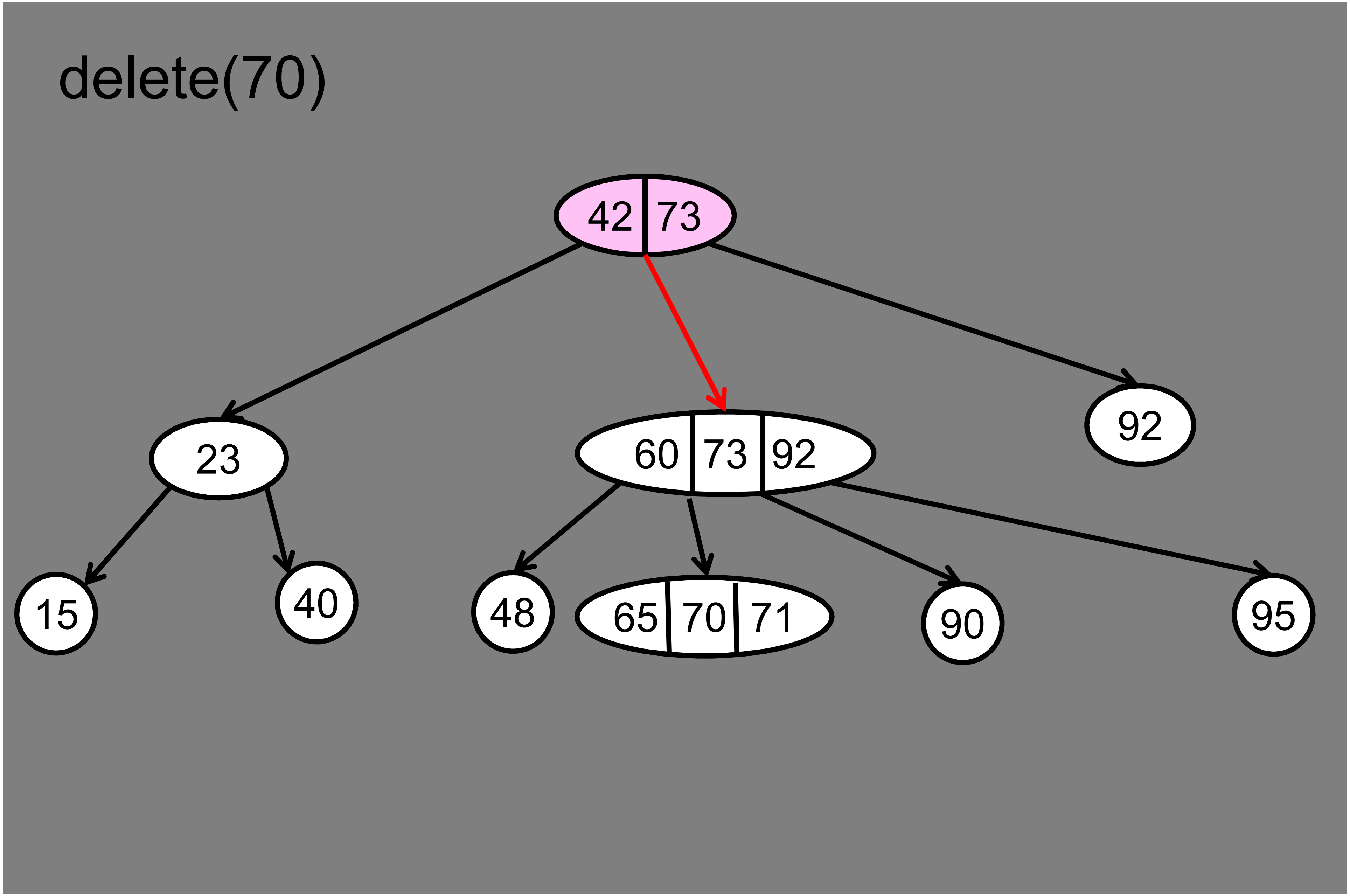


2-3-4 Trees: Deleting (hard case 2)

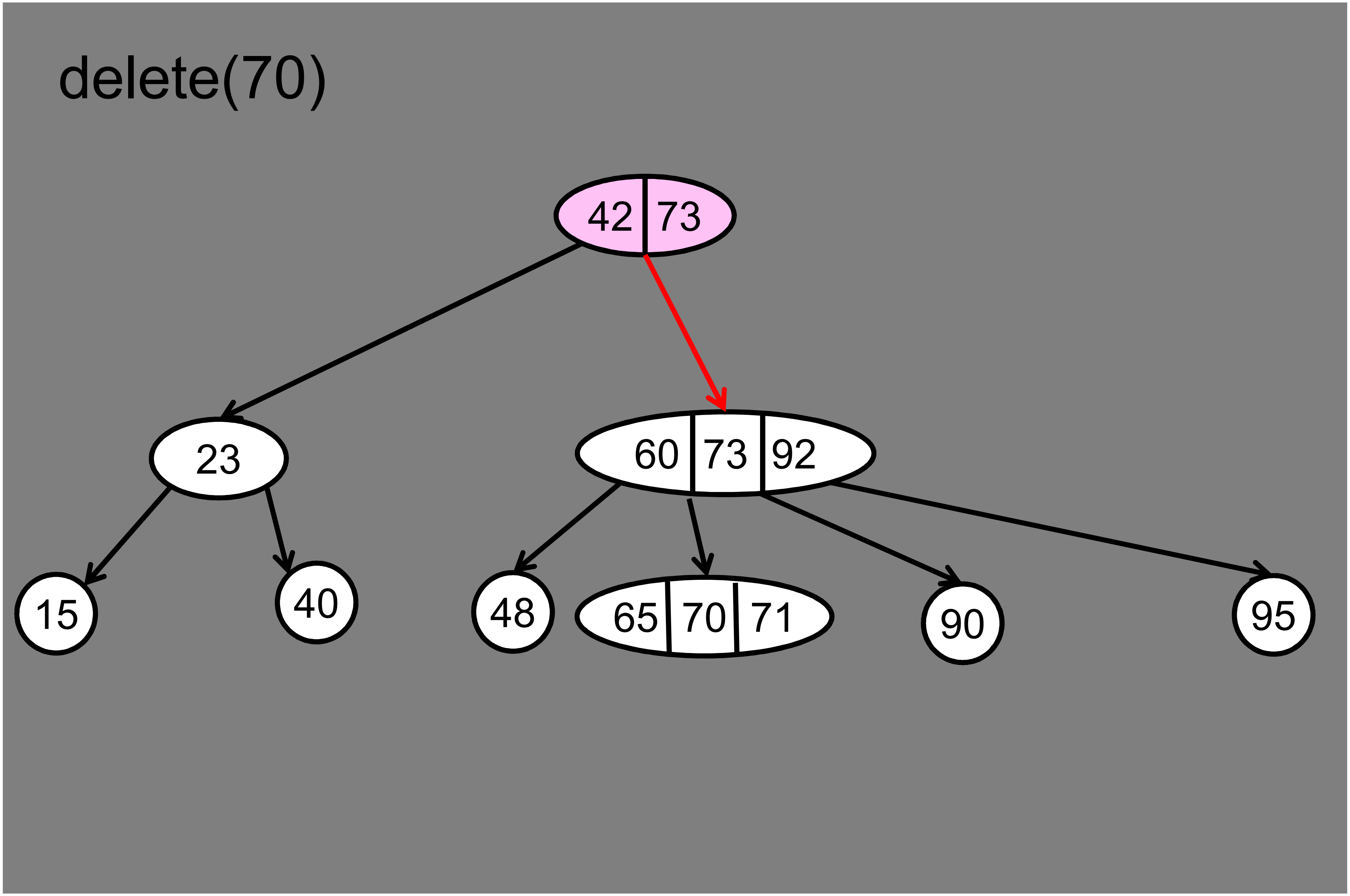
delete(70)



2-3-4 Trees: Deleting (hard case)

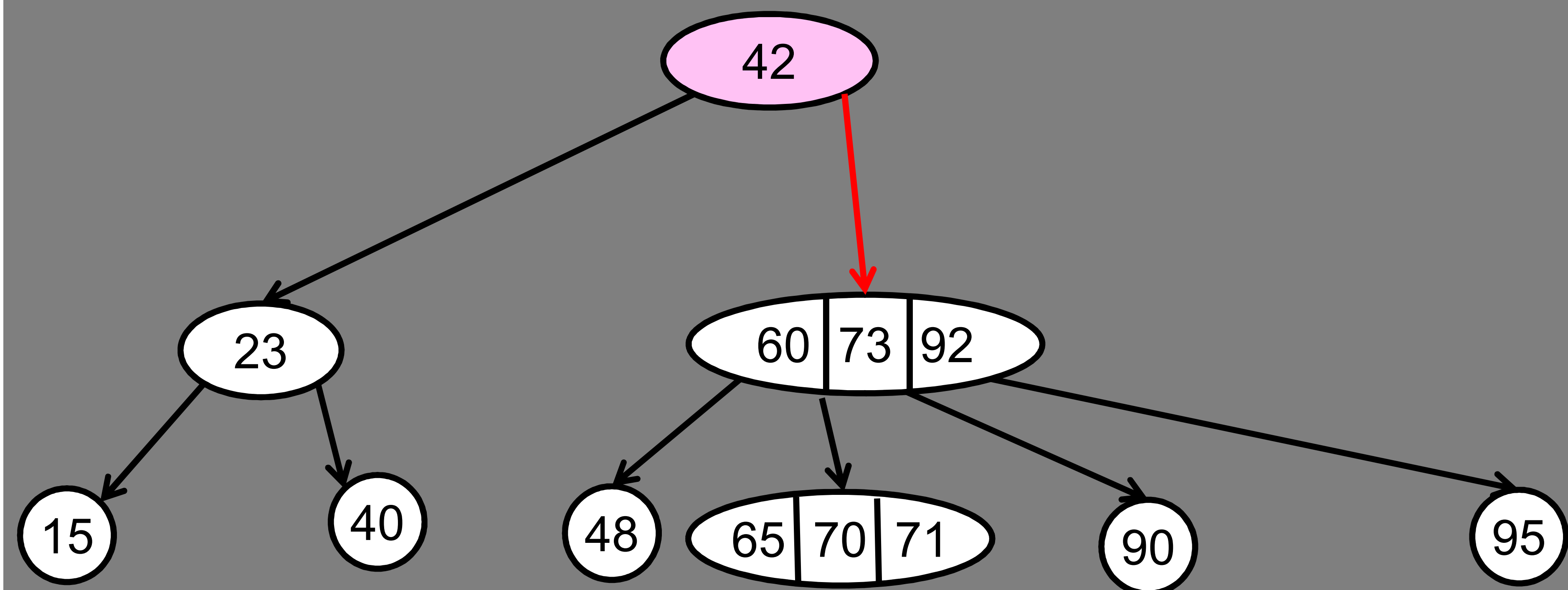


2-3-4 Trees: Deleting (hard case 2)



2-3-4 Trees: Deleting (hard case 2)

delete(70)



2-3-4 Trees

Summary:

- Three rules:
  1. Two, three, or four children
  2. Search Property
  3. All leafs have same depth
- Search: typical tree search
- Insert: split nodes to make room
- Delete: merge and juggle (see CLRS for details)

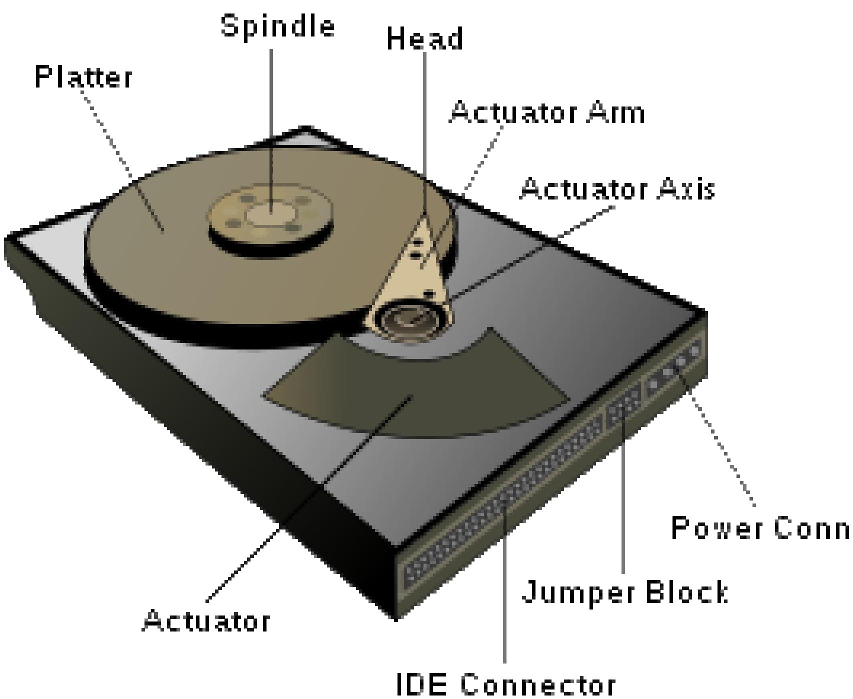
B-trees

The search tree used by *all* large databases…

B-trees

Where is most data stored? Hard disk!

- Magnetic
- Mechanical
- Slow (6000rpm = 10ms)



Two step access:

1. seek *(find right track)*
2. read track

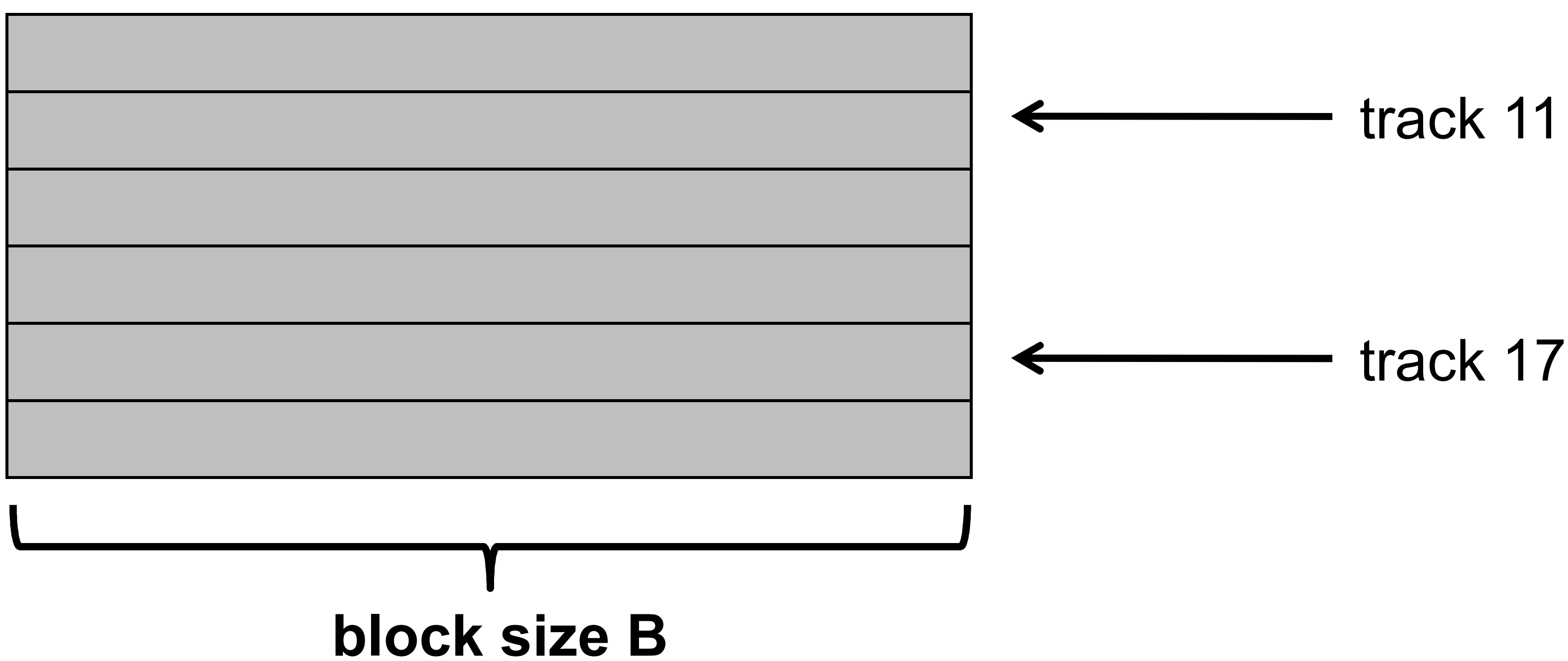
Image source: Wikipedia

B-trees

Two step access:

1. seek *(find right track)*
2. read track

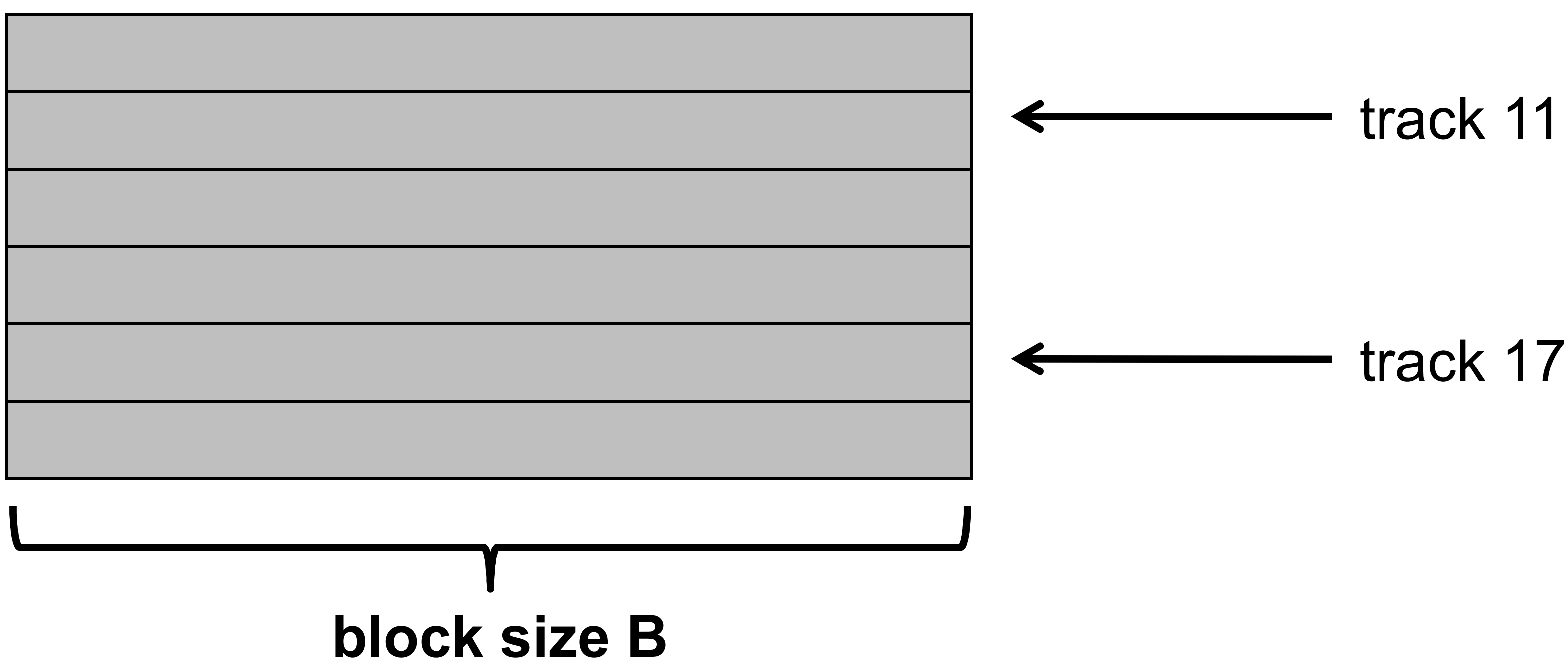
Solution: Cache entire track



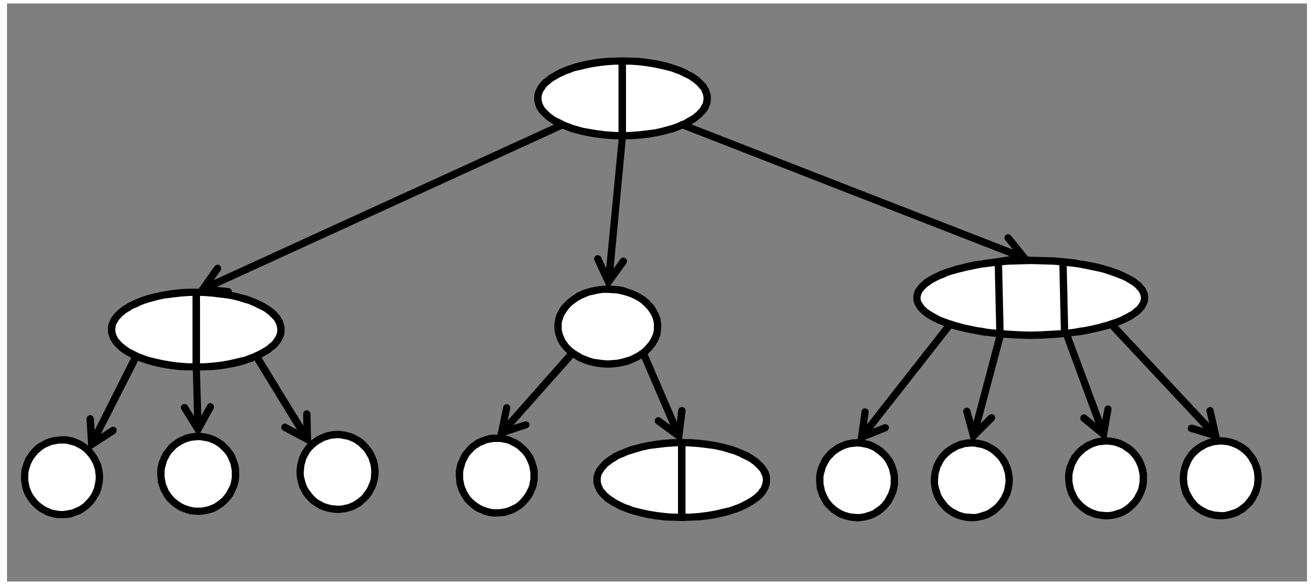
B-trees

Solution: Cache entire track

- Read in new track: costs 1 unit of work
- Read cached data: free



2-3-4 Trees



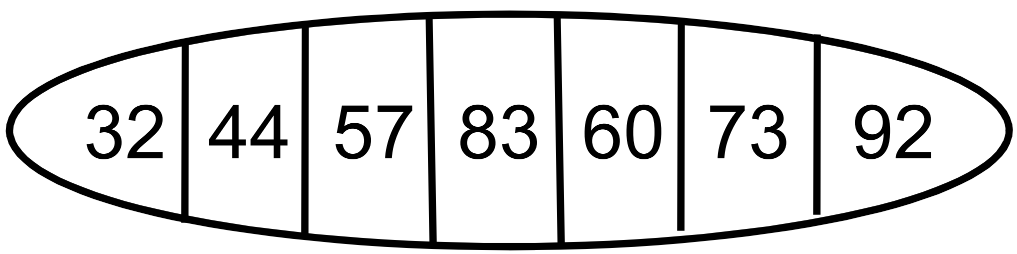
Search:

- Each step of the search reads a new track.
- Reading a track takes 10ms
- Each search takes 10ms*O(log n) time!

B Trees

Bigger nodes:

- Each node stores at least B keys
- No node stores more than 2B-1 keys
- All leaves have same depth.

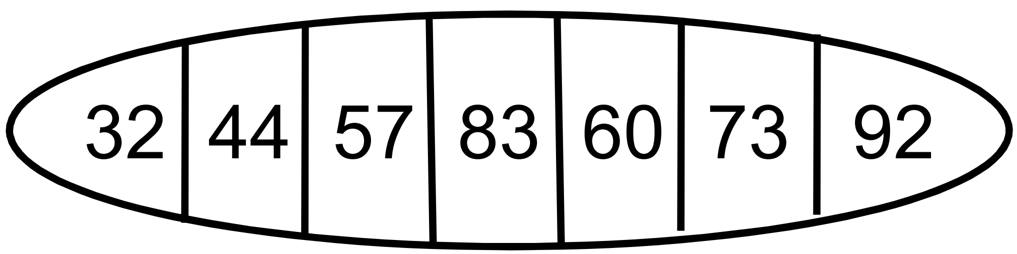


Time to search node: ???

B Trees

Bigger nodes:

- Each node stores at least B keys
- No node stores more than 2B-1 keys
- All leaves have same depth.



Time to search node: O(B) reads

O(1) disk seeks!

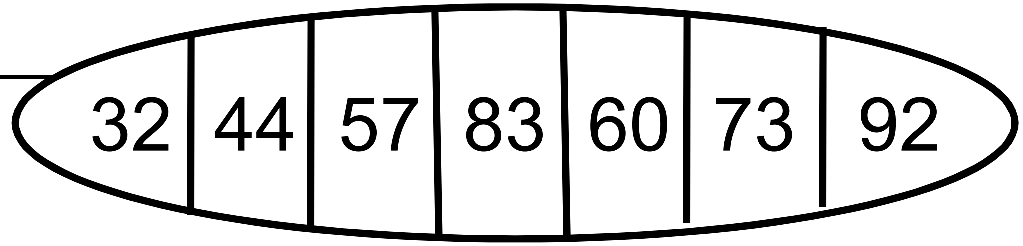
B Trees

Tree depth:

- Level 1: B
- Level 2: B^2
- Level 3: B^3
- Level 4: B^4
- …
- Level h: B^h

$n > B^h$ => $h < \log_B n = \log(n)/\log(B)$

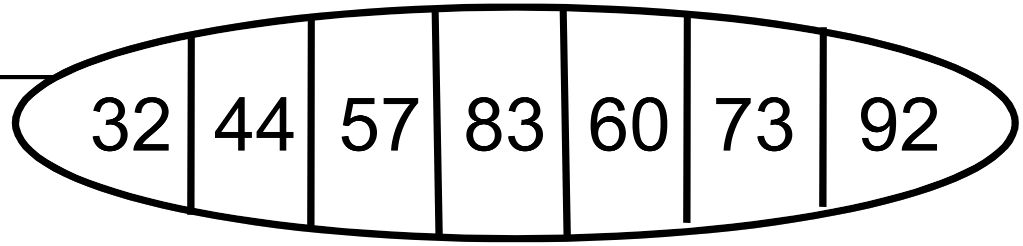
B Trees



Bigger nodes:

- Search, insert, delete as before.
- How to find index in big node?
  - Linear search: O(B)
  - Binary search: O(log B)

B Trees



Bigger nodes:

- Search, insert, delete as before.
- How to find index in big node?
  - Linear search: O(B)
  - Binary search: O(log B)

- But there are level 2 caches and level 3 caches…
  - Use small B-tree for smaller value of B to search for key within node!

B-trees

Summary:

- Caching performance matters.
- B-trees are fast
- B-trees are concurrent
- B-trees are (generally) simpler

Research: (cache-oblivious algorithms)

- What if you don't know the size of your cache?
- Can you devise an algorithm that performs well for all values of B?