Red-Black Trees

Alina Shaikhet

Carleton University

Red-Black Trees

Red-Black Trees are the underlying implementation of **TreeSet** and **TreeMap**

- **Red-Black Trees** are a variation of **binary search trees** with logarithmic height.
- They are one of the most widely used data structures.

1. A red-black tree storing $\boldsymbol{n}$ values has height at most $\mathbf{2}\log \boldsymbol{n}$.
2. The add$(\boldsymbol{x})$ and remove$(\boldsymbol{x})$ operations on a red-black tree run in $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$ **worst-case** time.
3. The amortized number of rotations performed during an add$(\boldsymbol{x})$ or remove$(\boldsymbol{x})$ operation is **constant**.

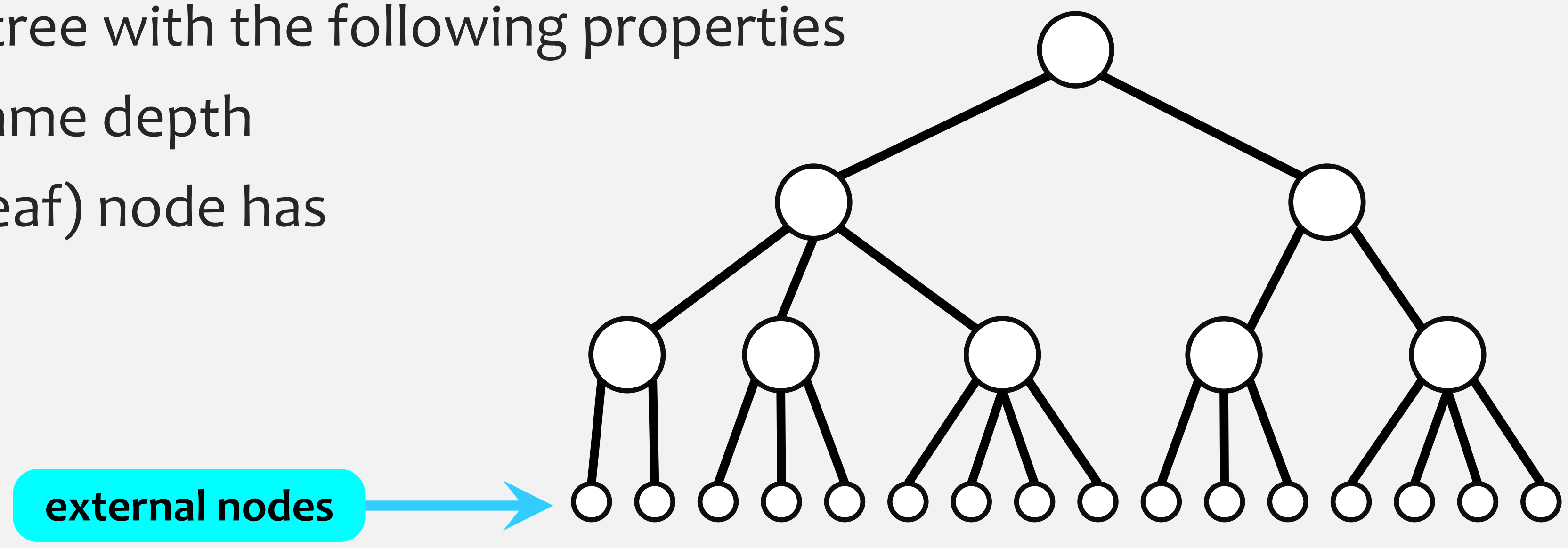
Recall SSet implementations:

- **SkipLists** and **Treaps** rely on randomization – $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$ running times are only **expected**.
- **ScapegoatTrees** have a guaranteed bound on their height but add$(\boldsymbol{x})$ and remove$(\boldsymbol{x})$ operations only run in $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$ **amortized** time.

2-4 Trees

A **2-4 tree** is a rooted tree with the following properties

- All leaves have the same depth
- Every internal (non-leaf) node has $\mathbf{2}$, $\mathbf{3}$, or $\mathbf{4}$ children.



Claim: If a **2-4 tree** has height $\boldsymbol{h}$ then it has at least $\mathbf{2}^{\boldsymbol{h}}$ leaves.

Claim: If a **2-4 tree** has $\boldsymbol{n}+\mathbf{1}$ leaves then it has height at most $\mathbf{log_2}(\boldsymbol{n}+\mathbf{1})$.

 # leaves $\geq \mathbf{2}^{\boldsymbol{h}}$

$$\boldsymbol{n}+\mathbf{1} \geq \mathbf{2}^{\boldsymbol{h}}$$

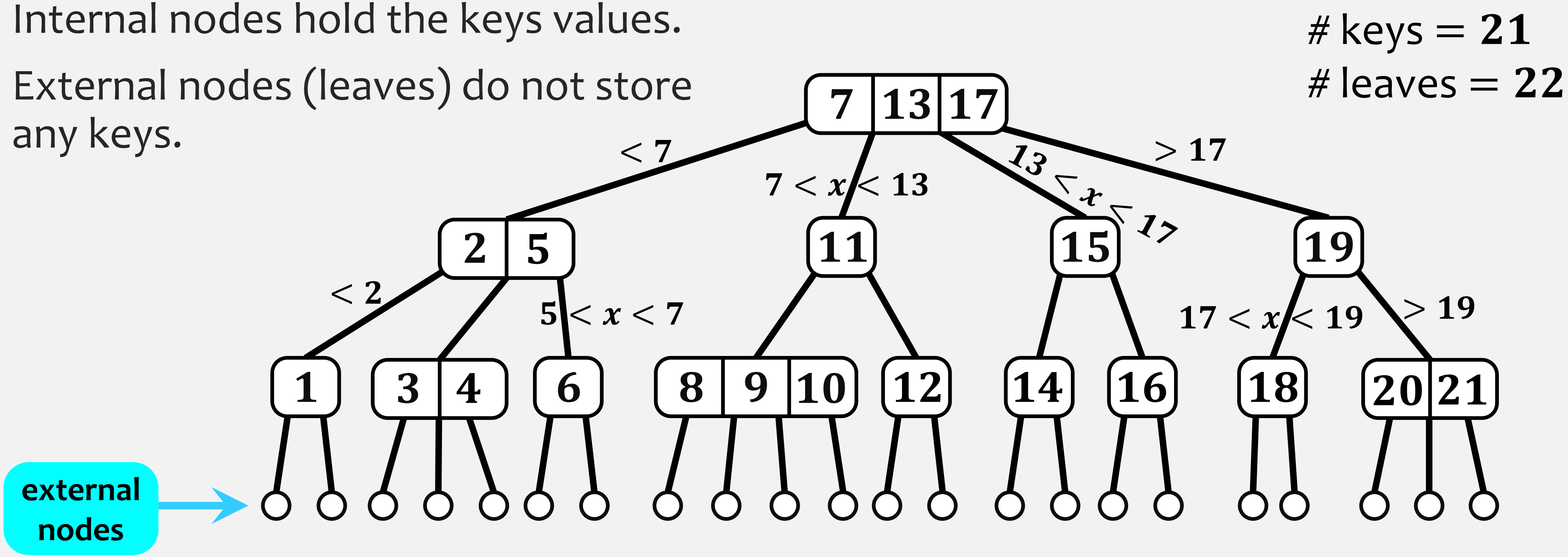
$$\mathbf{log_2}(\boldsymbol{n}+\mathbf{1}) \geq \boldsymbol{h}$$

2-4 Trees – find(x)

$O(\log n)$

Internal nodes hold the keys values.

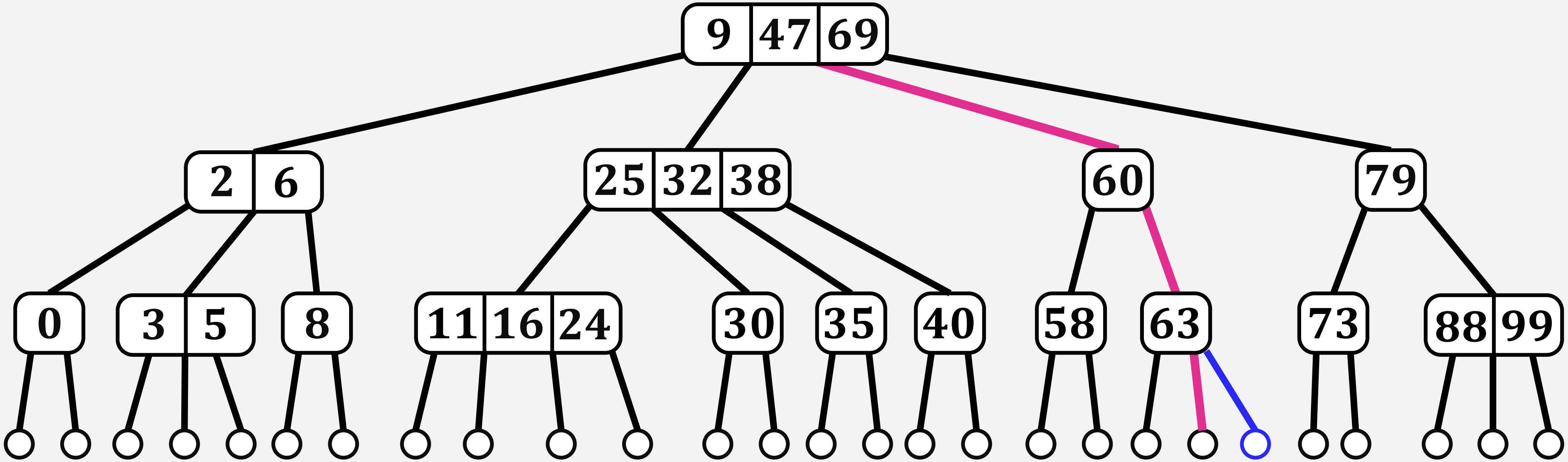
External nodes (leaves) do not store any keys.



Claim: If a **2-4 tree** has n keys then it has exactly $n + 1$ leaves.

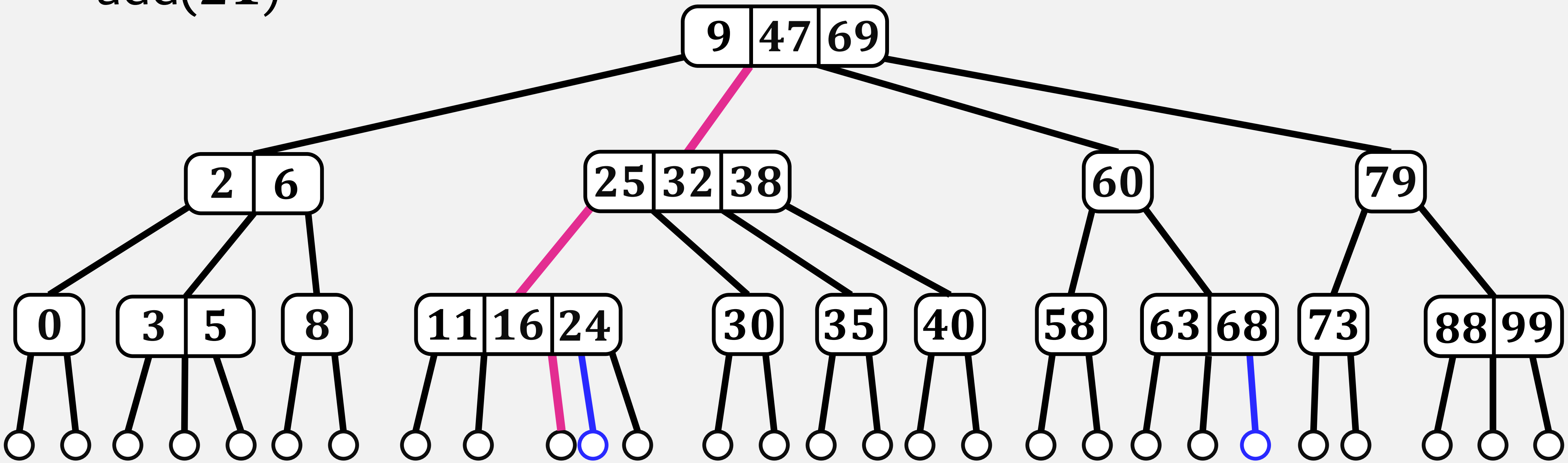
2-4 Trees – add(x)

add(**68**)



2-4 Trees – add(x)

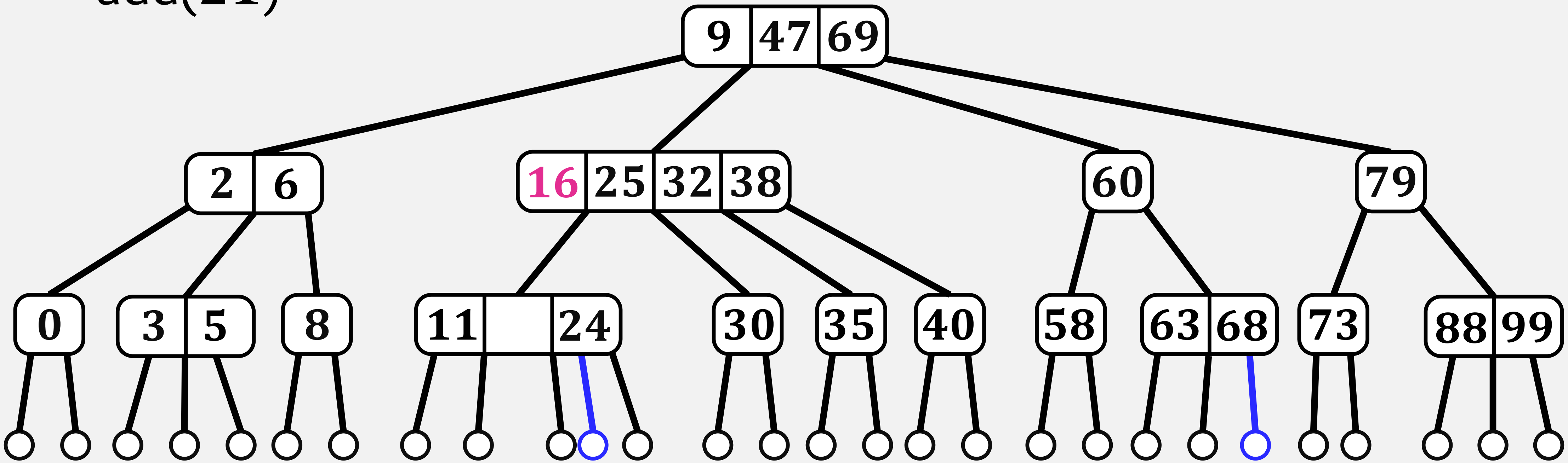
add($\mathbf{68}$)
add($\mathbf{21}$)



2-4 Trees – add($\boldsymbol{x}$)

add($\mathbf{68}$)

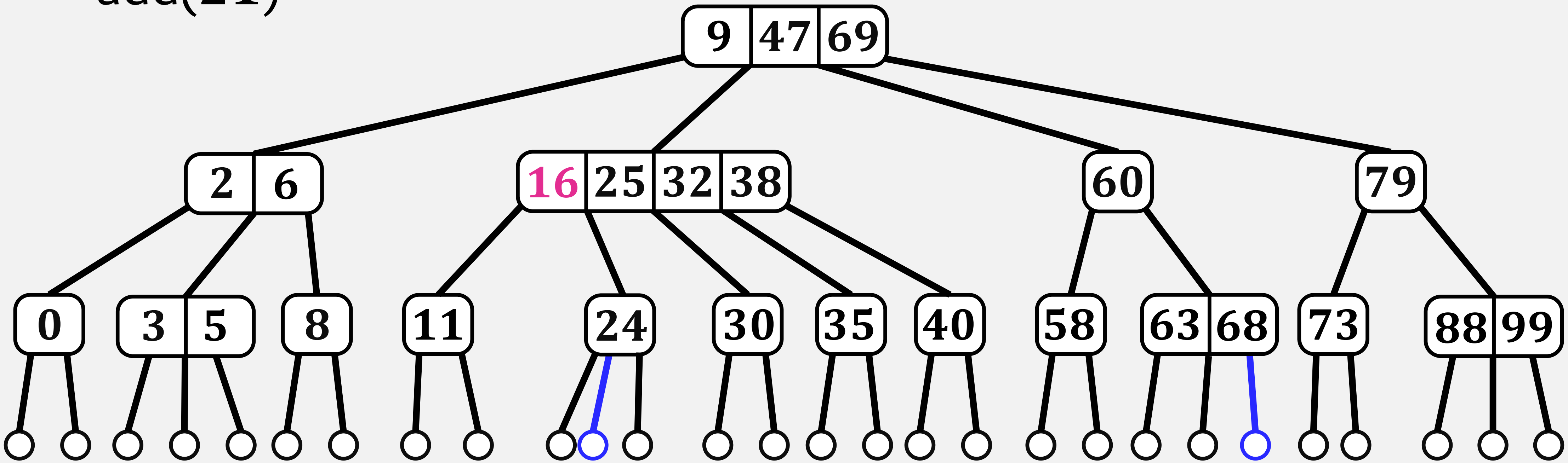
add($\mathbf{21}$)



2-4 Trees – add($\boldsymbol{x}$)

add($\mathbf{68}$)

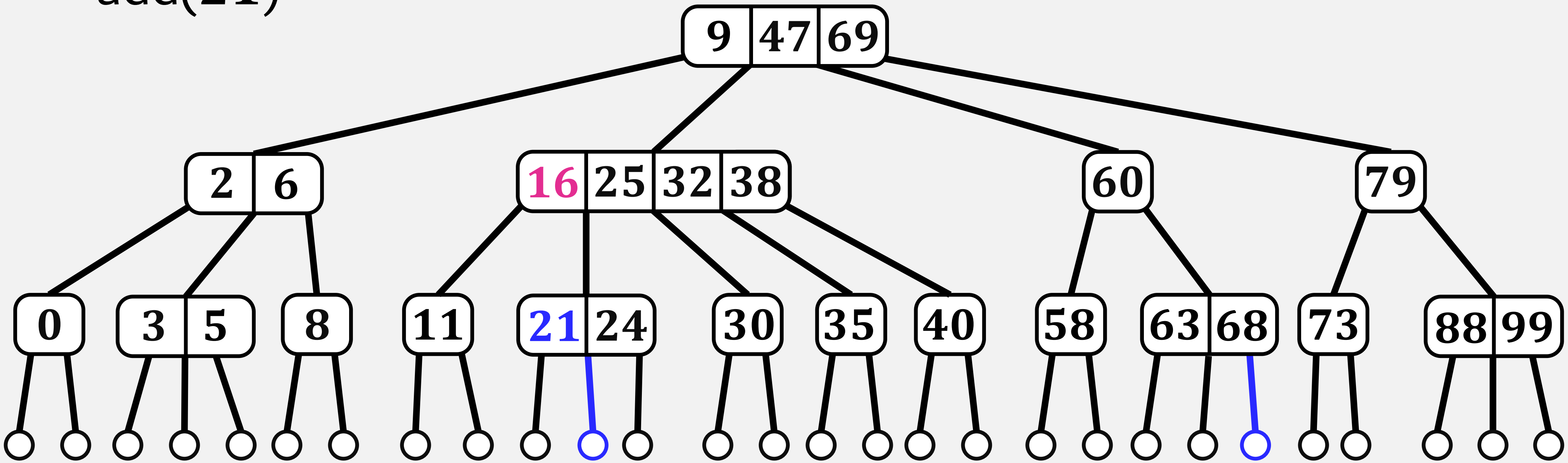
add($\mathbf{21}$)



2-4 Trees – add($\boldsymbol{x}$)

add($\mathbf{68}$)

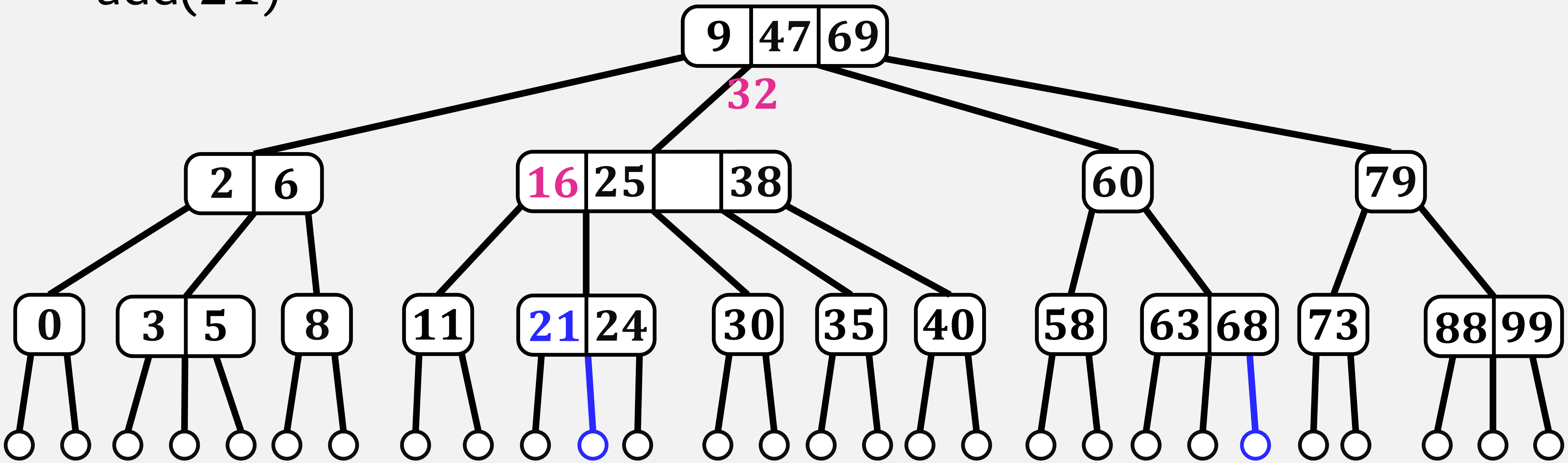
add($\mathbf{21}$)



2-4 Trees – add(x)

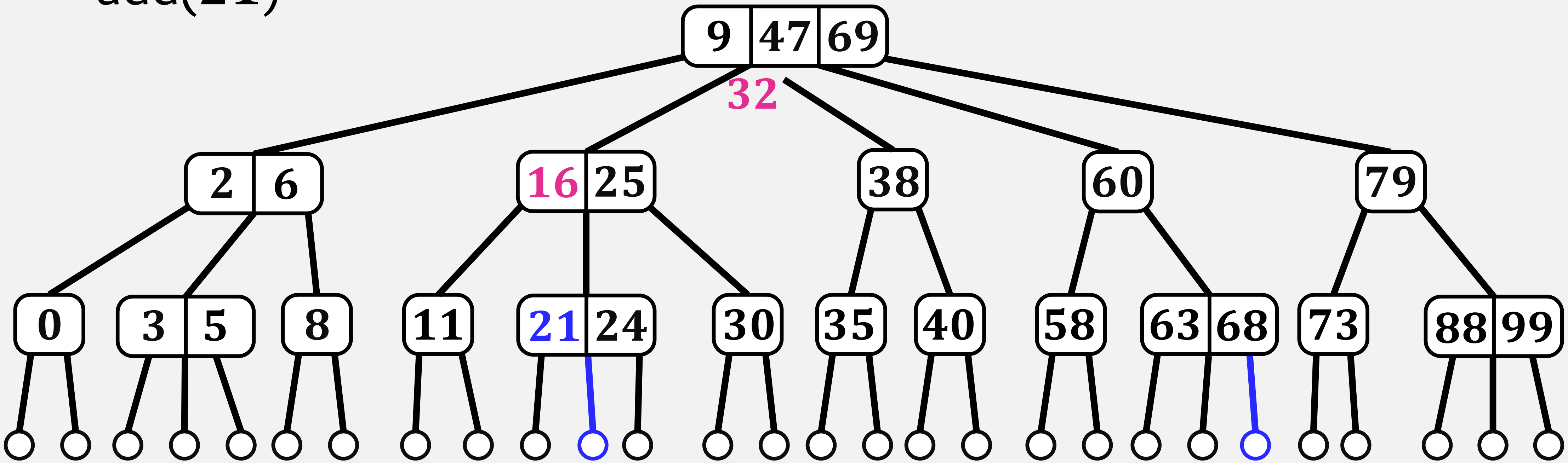
add($\mathbf{68}$)

add($\mathbf{21}$)



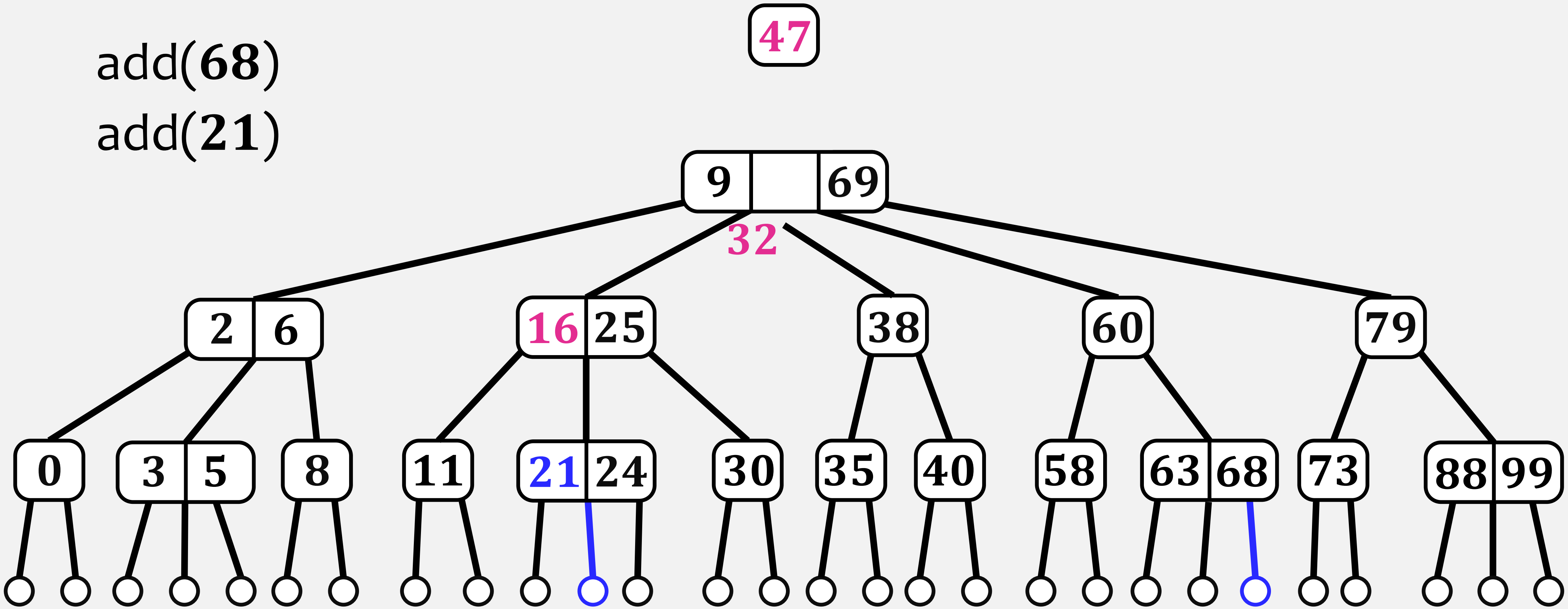
2-4 Trees – add($\boldsymbol{x}$)

add($\mathbf{68}$)
add($\mathbf{21}$)



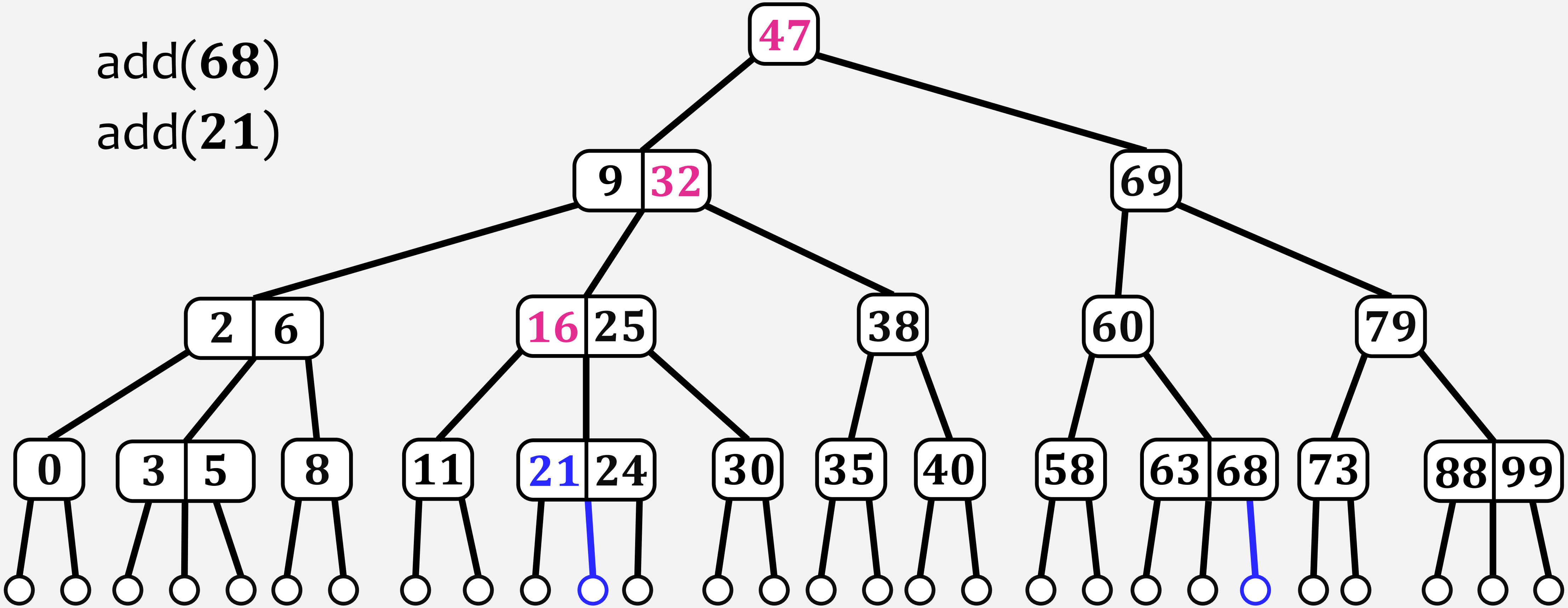
2-4 Trees – add($\boldsymbol{x}$)

add(**68**)
add(**21**)



2-4 Trees – add(x)

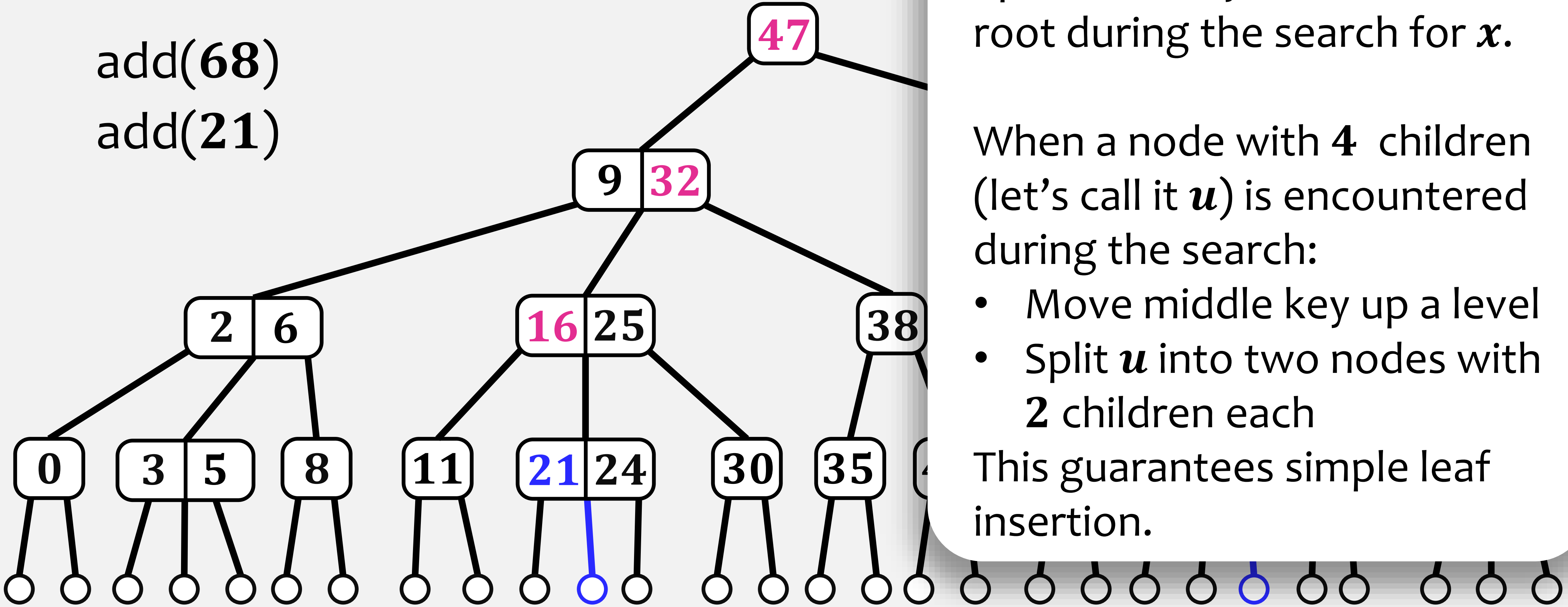
add(**68**)
add(**21**)



Insertion can be done in one pass $\boldsymbol{O}(\log \boldsymbol{n})$

2-4 Trees – add(x)

add(**68**)
add(**21**)



Implementation:
Nodes with **4** children are split up on the way down from the root during the search for x.

When a node with **4** children (let's call it u) is encountered during the search:

- Move middle key up a level
- Split u into two nodes with **2** children each

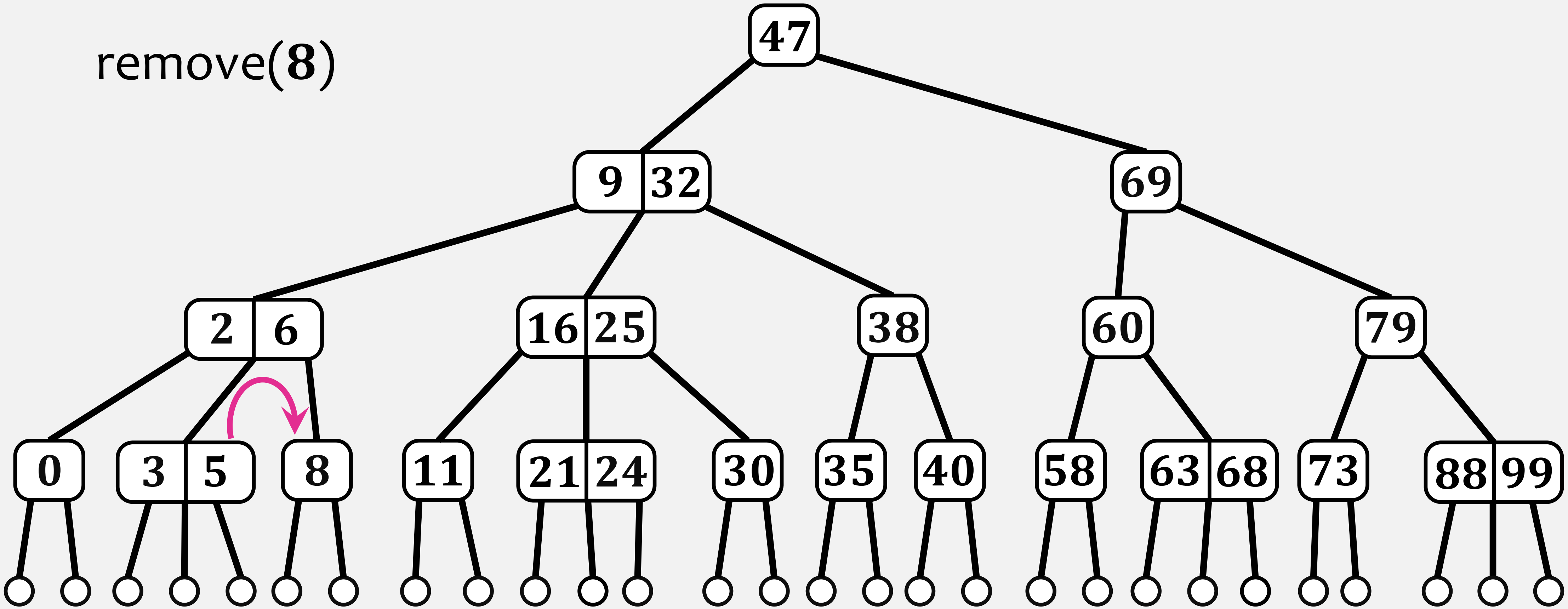
This guarantees simple leaf insertion.

Insertion can be done in one pass $O(\log n)$

Alina Shaikhet – COMP 2402 – Carleton University

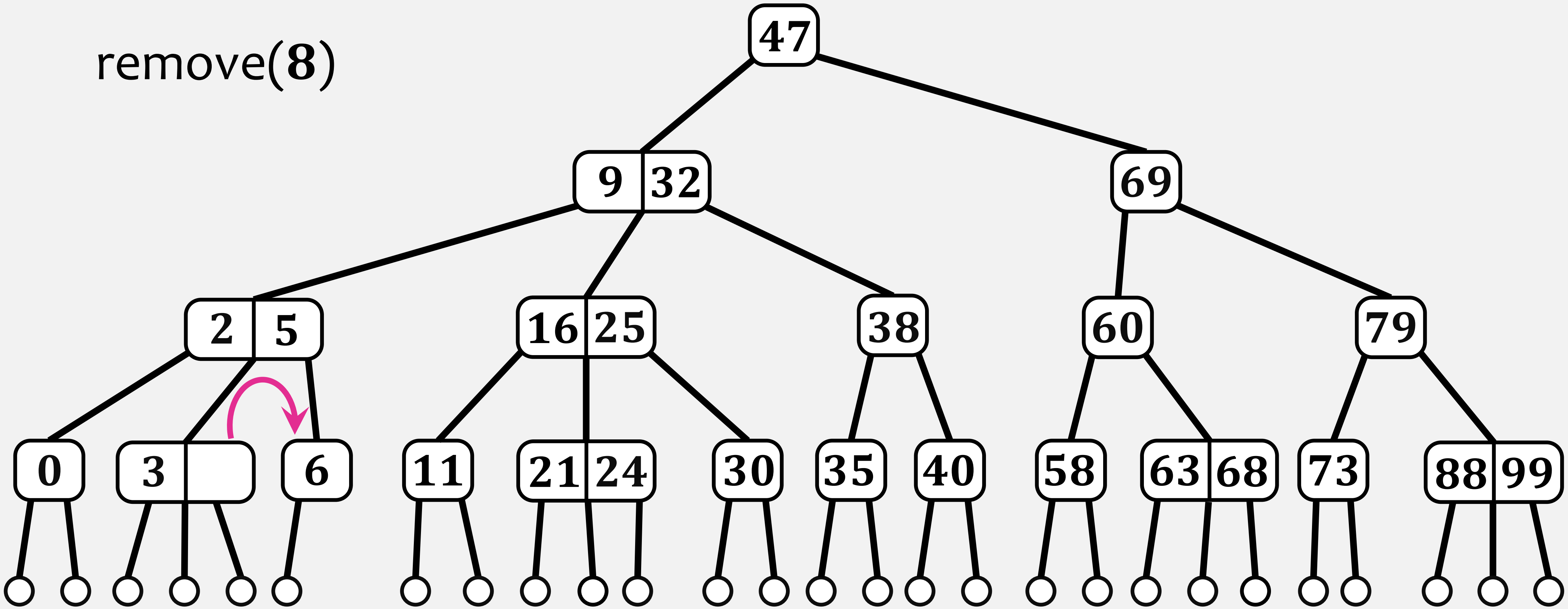
2-4 Trees – remove($\boldsymbol{x}$)

remove($\mathbf{8}$)



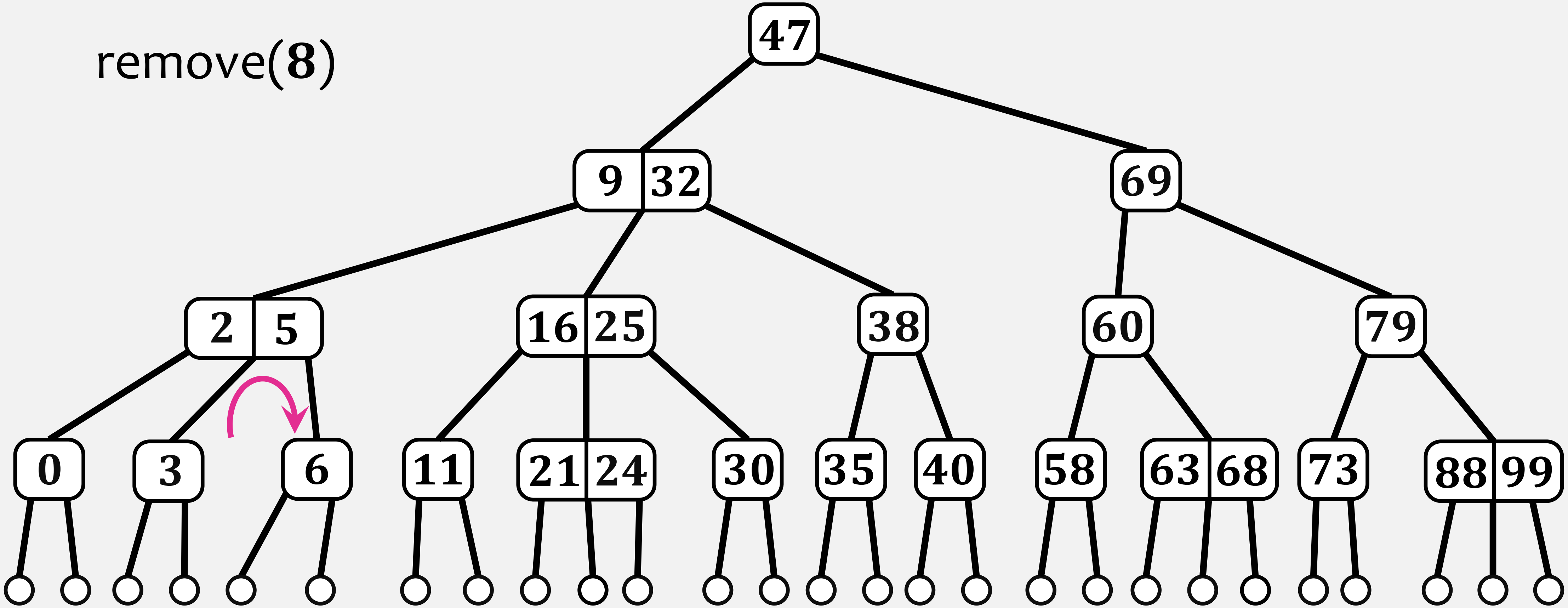
2-4 Trees – remove($\boldsymbol{x}$)

remove($\mathbf{8}$)



2-4 Trees – remove(x)

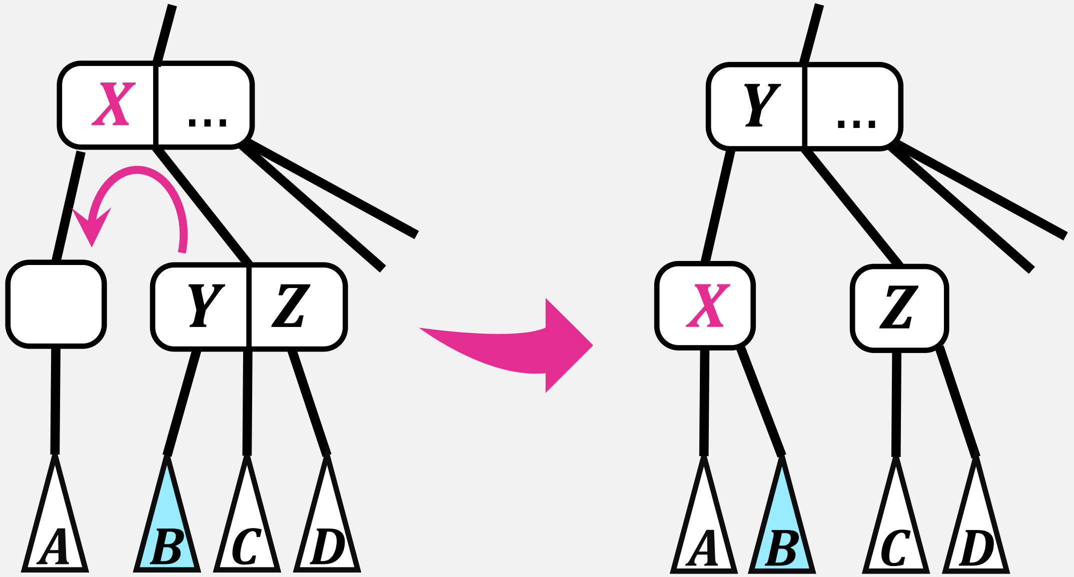
remove(**8**)



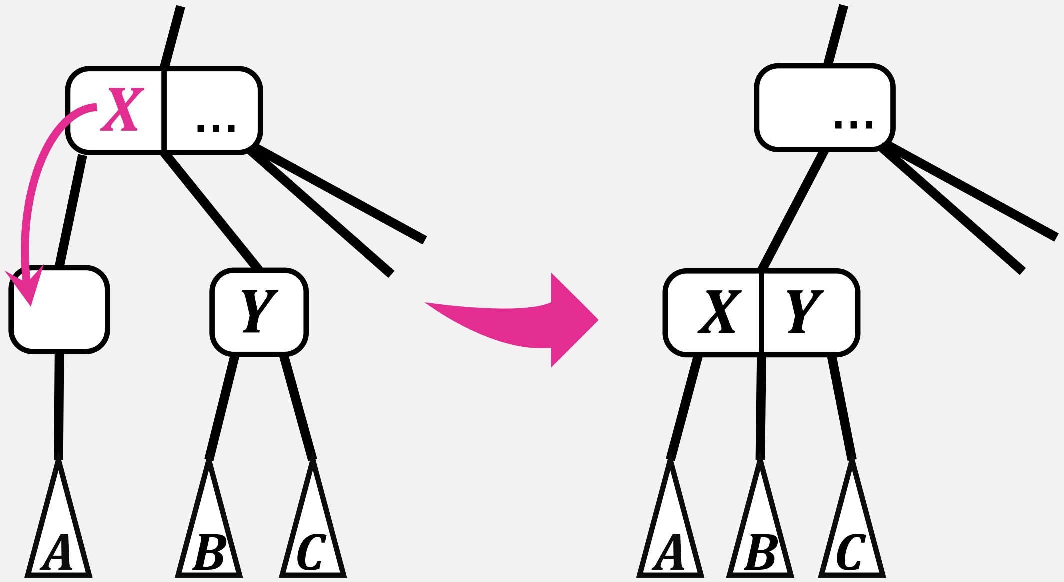
2-4 Trees – remove(x)

To implement remove(x) we need to know:

1. how to borrow a child from a sibling (that has two or three keys)



2. how to merge two nodes (their parent can become empty)



2-4 Trees – remove($\boldsymbol{x}$)

Deletion can be tricky.

Removing a node with **2** children involves **rotations** and/or **fusions** (recombine two nodes with **2** children each back into a node with **4** children)

Deletion can be done in a single pass $\boldsymbol{O}(\log \boldsymbol{n})$

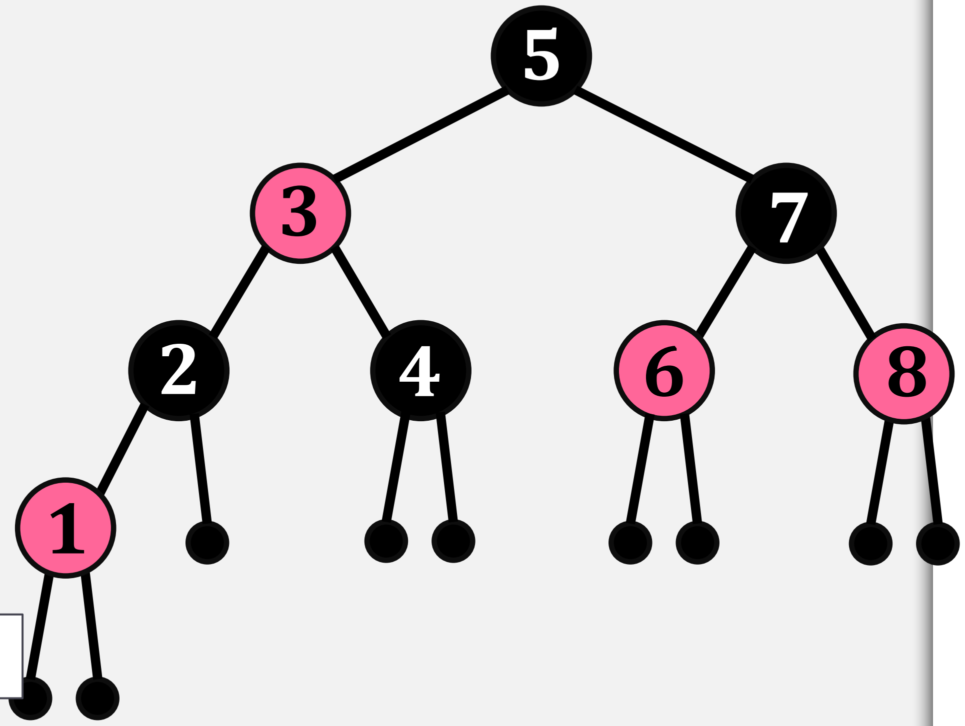
In many cases instead of deleting consider leaving in place, just mark as deleted. You may reuse these nodes/keys in future insertions.

This approach is not good when doing many add/remove ops of **different** values.

RedBlackTree

red $= 0$
black $= 1$

A **red-black tree** is a **binary search tree** such that each node u has a colour which is either **red** or **black**.



Properties:

1. **Black-height:** There are the same number of **black** nodes on every root-to-leaf path.

   The sum of the colours on any root-to-leaf path is the same.

2. **No-red-edge:** No two **red** nodes are adjacent. Each **red** node has a parent, that is a **black** node.

   The root is a **black** node.

   For any node u, except the root,
   u.colour + u.parent.colour ≥ 1.

Internal nodes hold the keys values. External nodes (leaves) do not store any keys.

Alina Shaikhet – COMP 2402 – Carleton University

RedBlackTree

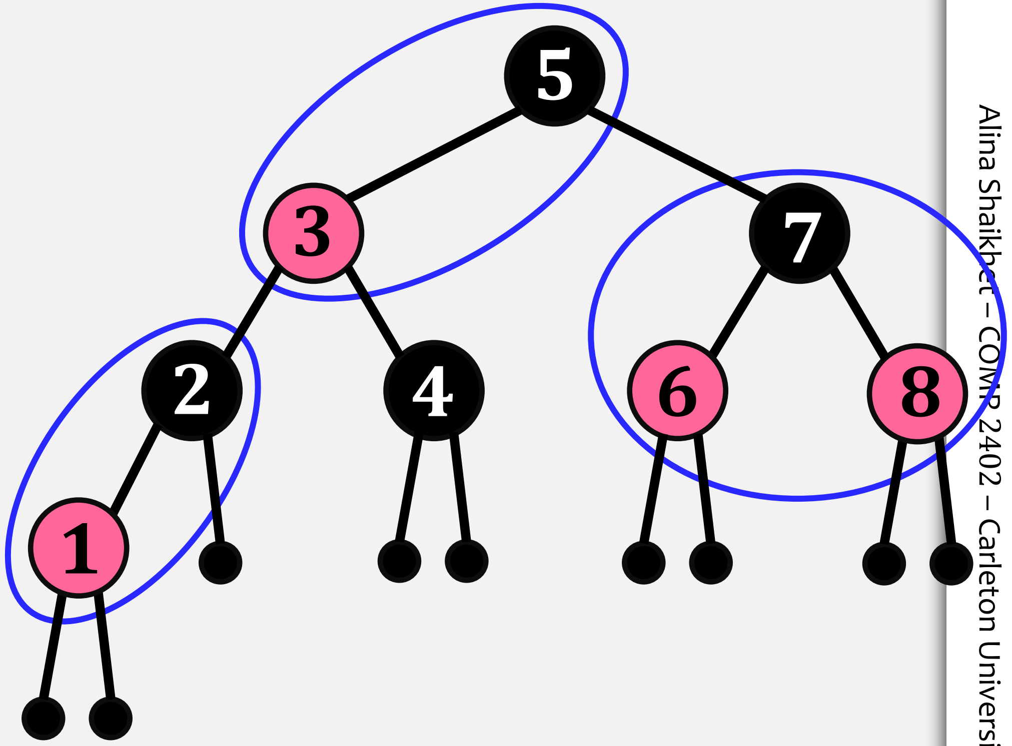
Transformation:

Remove each **red** node u and connect its two children directly to its parent.

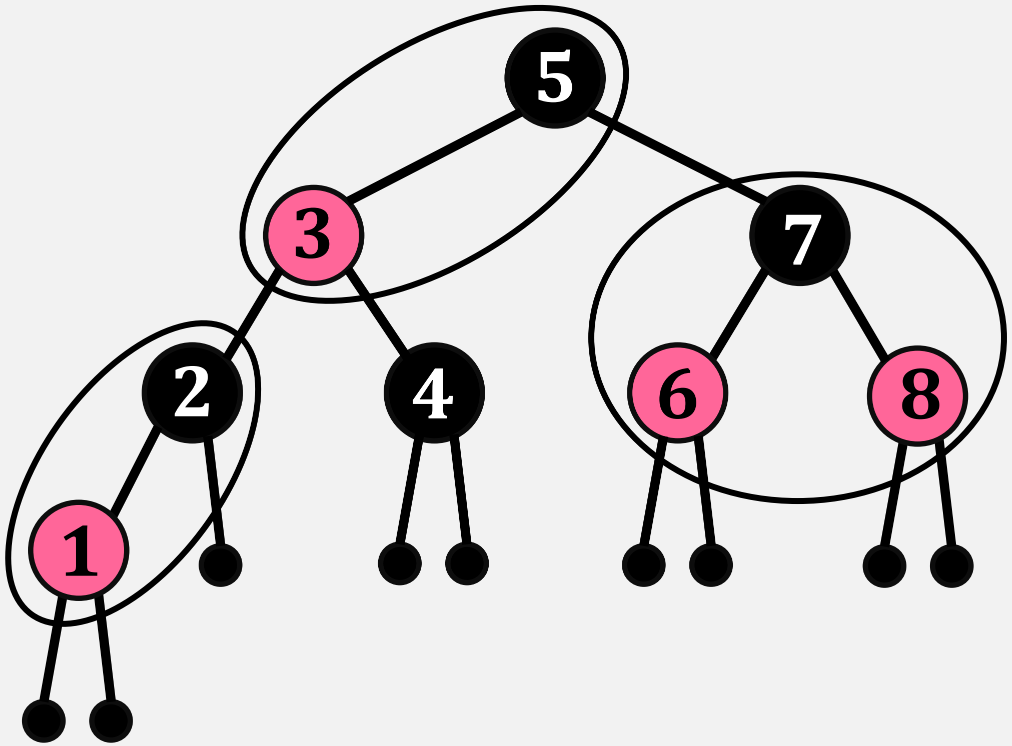
For each **red** node:

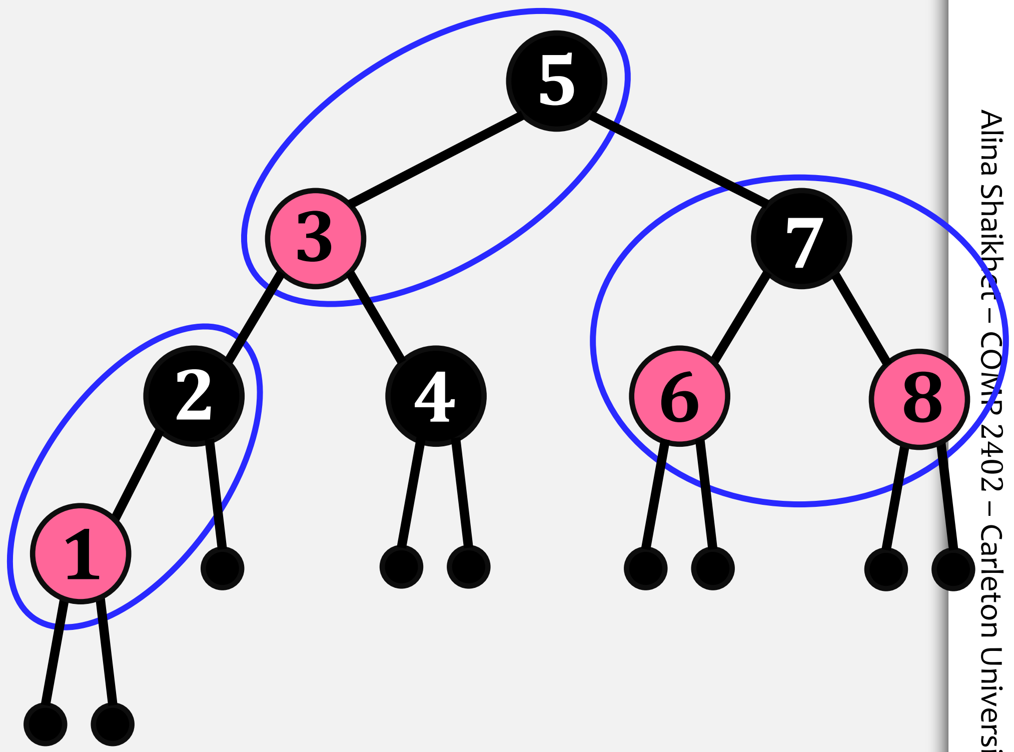
- Parent node is **black**
- Both children nodes are **black**

After the transformation there will be no **red** nodes in the tree.

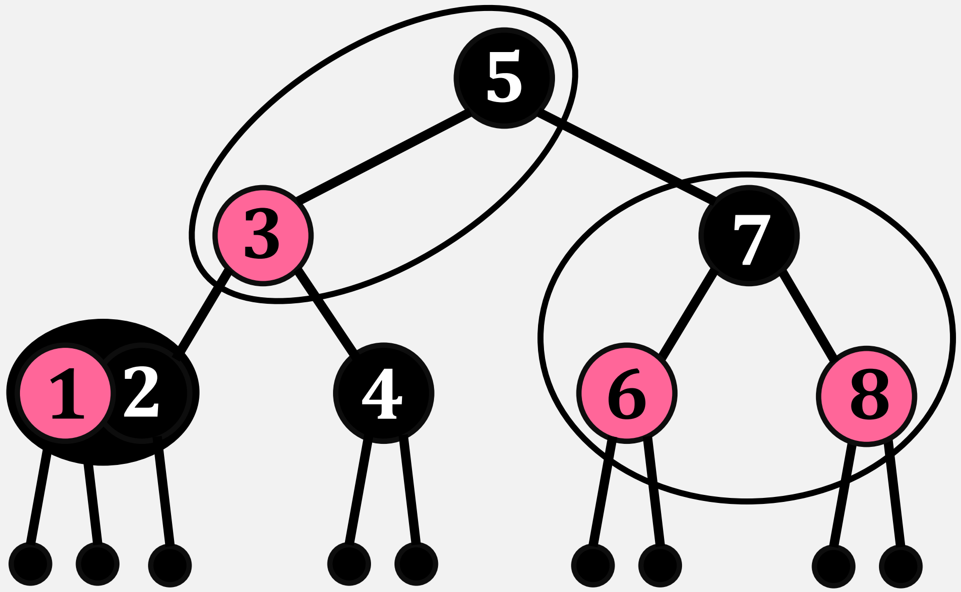


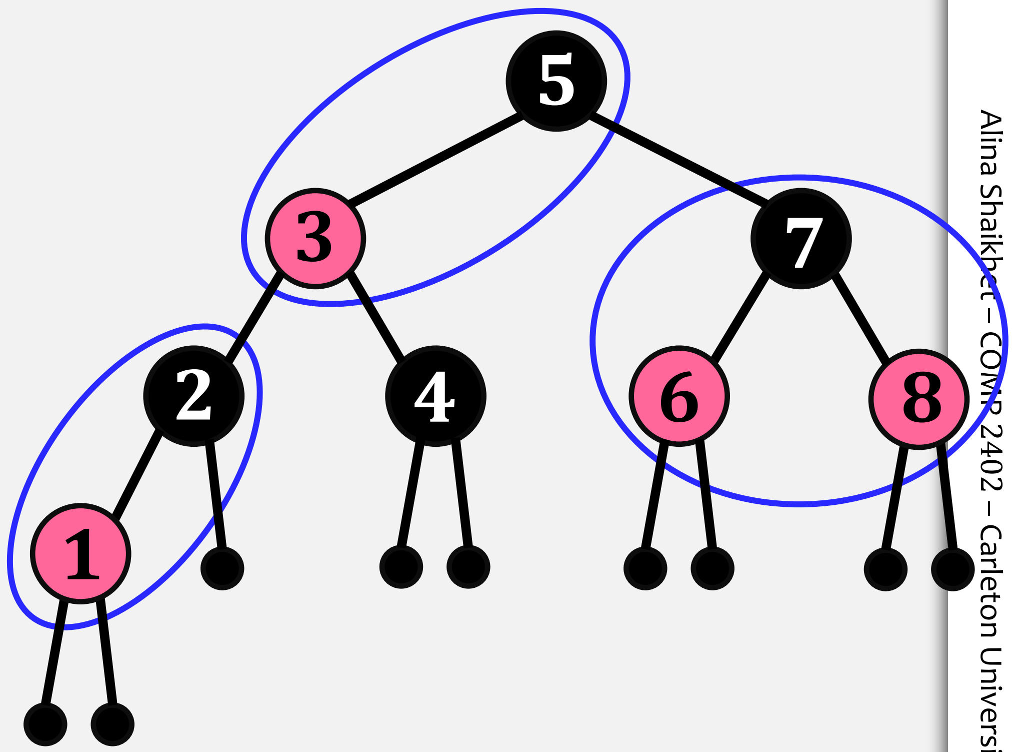
RedBlackTree



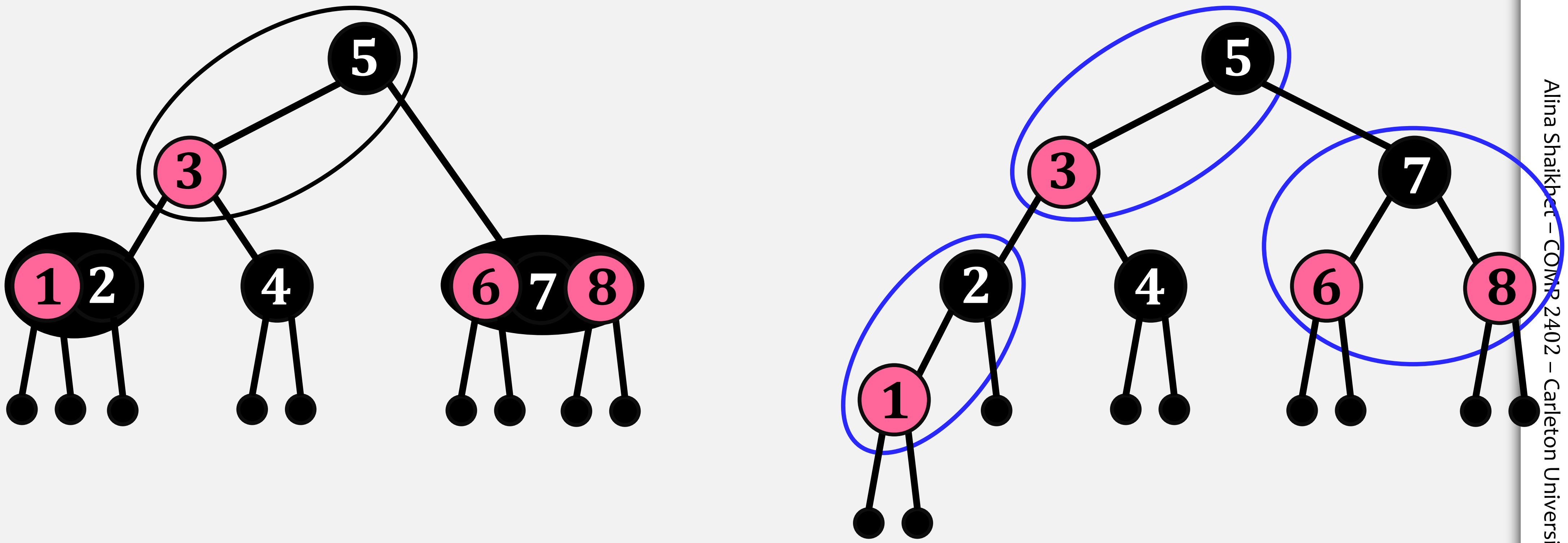


RedBlackTree



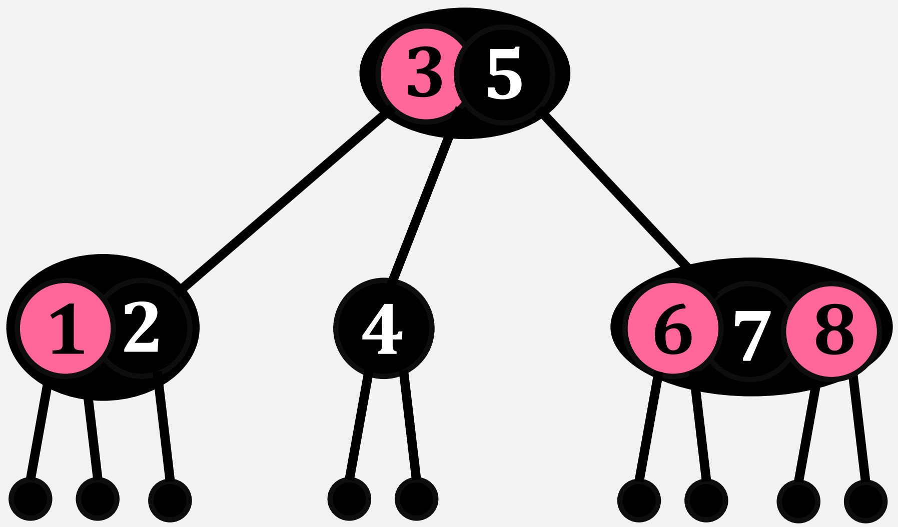


RedBlackTree

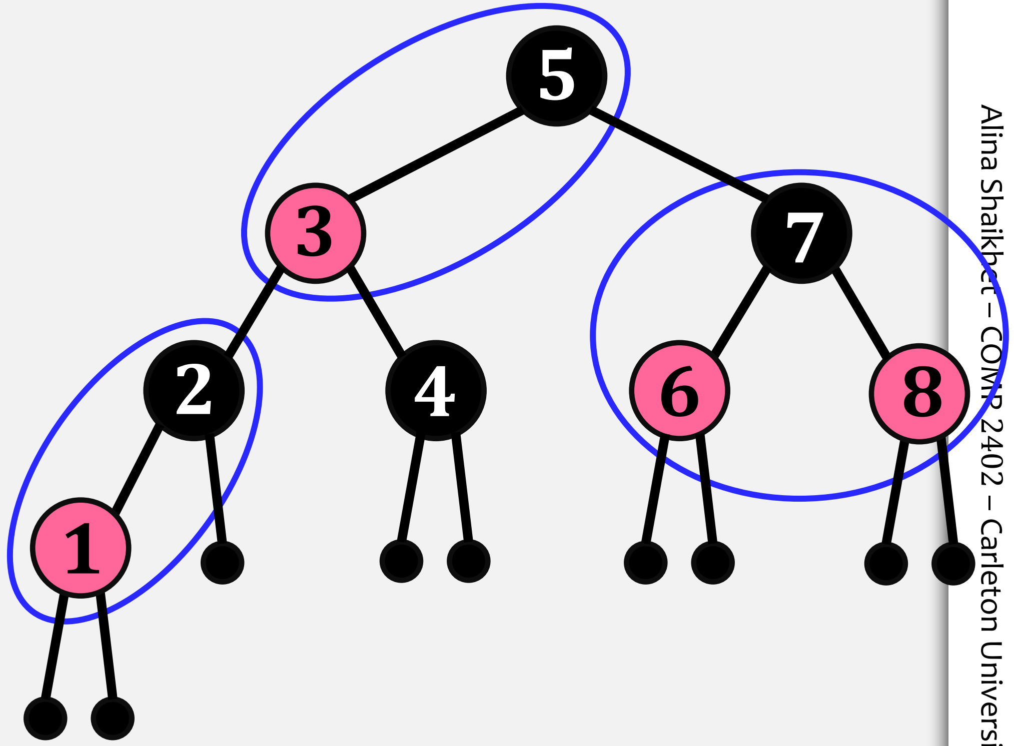


RedBlackTree

RedBlackTree simulates 2-4 tree using binary tree



2-4 tree

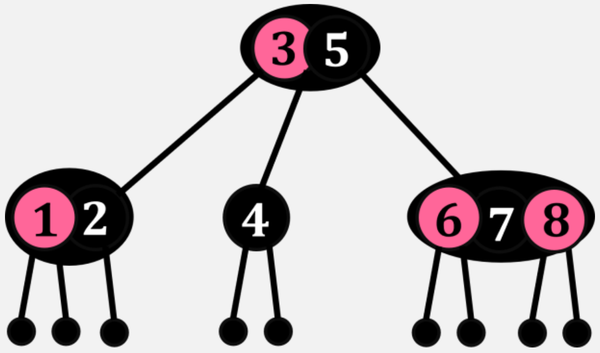


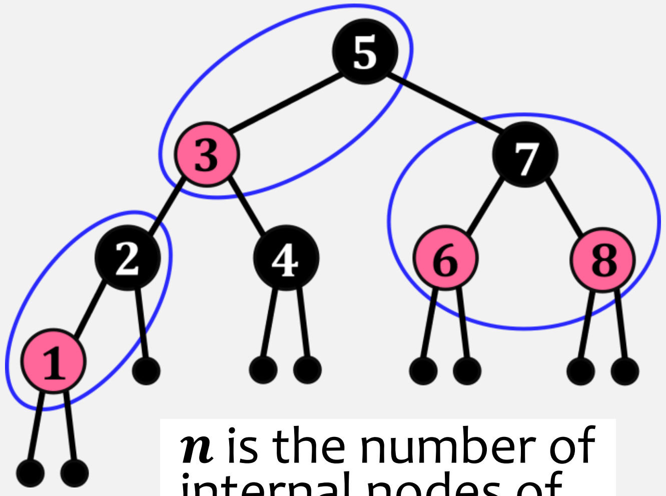
- All leaves have the same depth because of the **black-height** property,
- Every internal (non-leaf) node has **2**, **3**, or **4** children.

Height

A **2-4 tree** has $n + 1$ leaves that correspond to the $n + 1$ external nodes of the red-black tree.

We know that **2-4 tree** has height at most $\log_2(n + 1)$.





n is the number of internal nodes of the **red-black tree**

Every root-to-leaf path in the **2-4 tree** corresponds to a path from the root of the **red-black tree** to an external node.

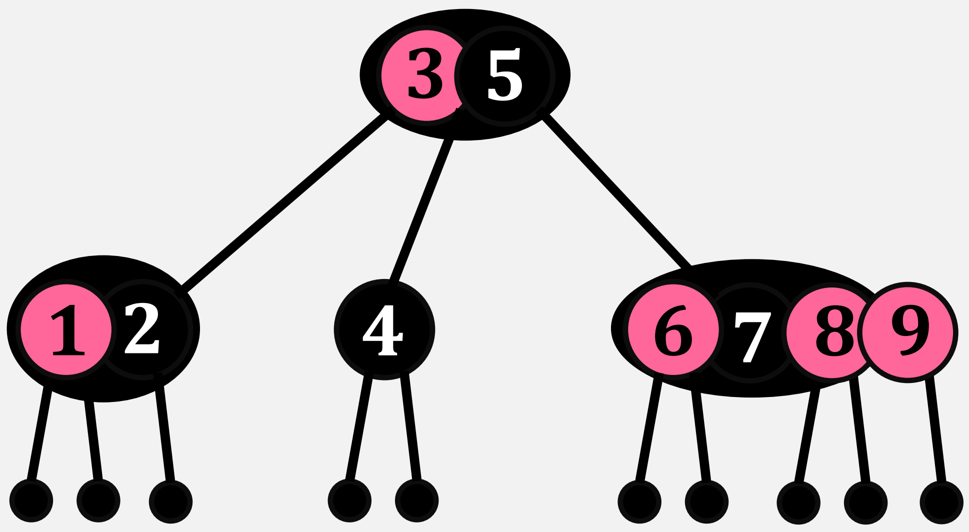
- first and last node in this path are **black** nodes
- at most one (out of every two) internal nodes is a **red** node

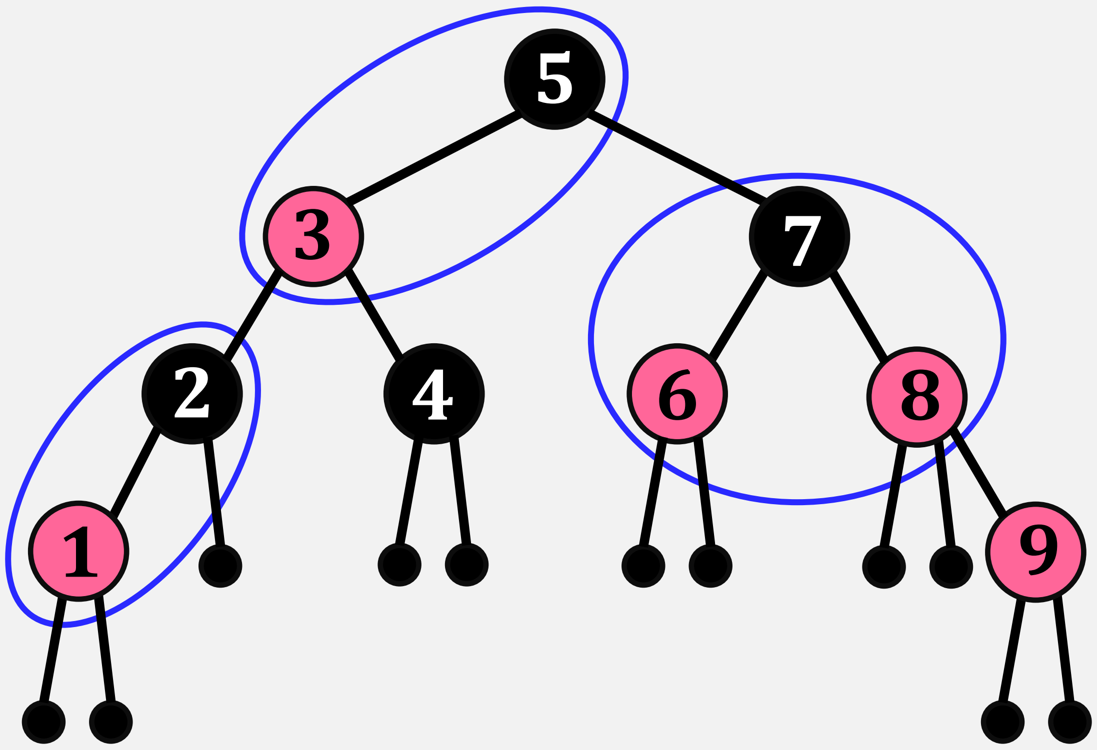
So, this path has at most $\log_2(n + 1)$ **black** nodes,
and at most $\log_2(n + 1) - 1$ **red** nodes.

Thus, the height of the red-black tree is at most $2\log_2(n + 1) - 2 \le 2\log_2 n$

Red-Black Trees – add(x)

add(**9**)





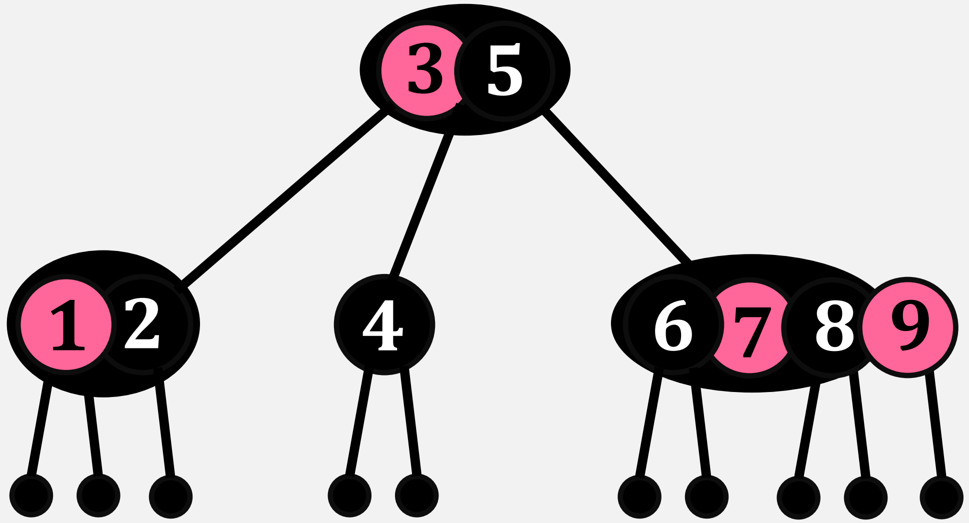
In a **red-black tree** we need a method of simulating splitting a node with five children in a **2-4 tree**:

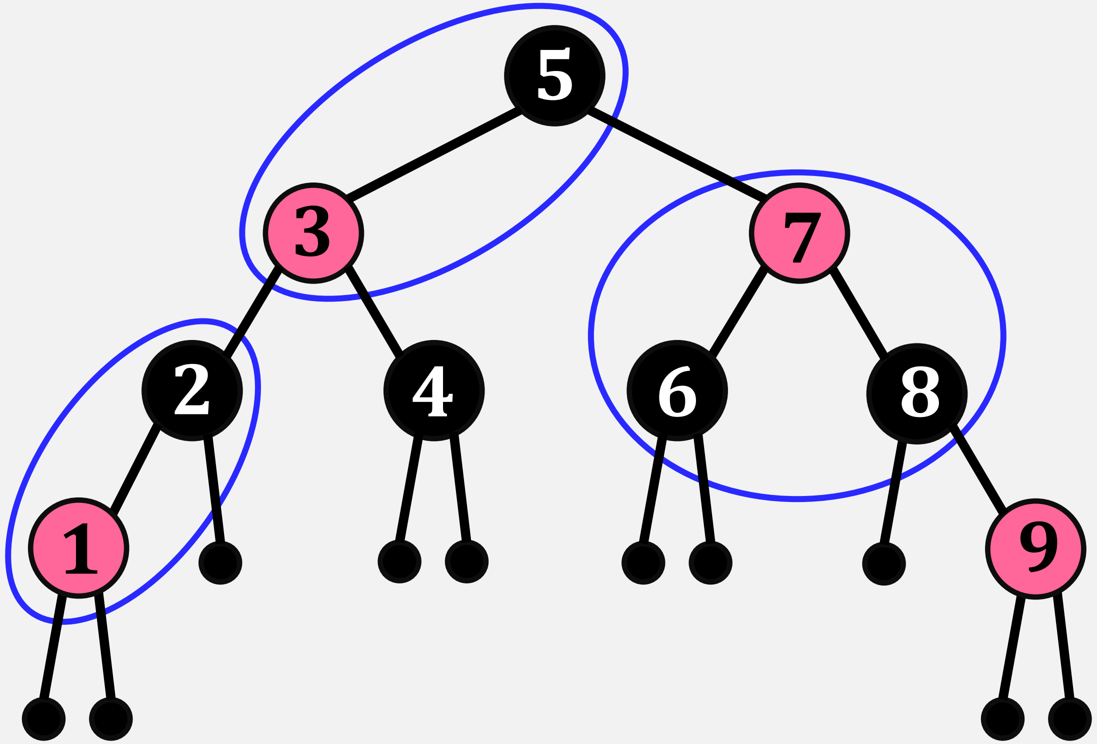
We can “split” the node by recolouring: node **7** becomes red; nodes **6**, **8** become black

Alina Shaikhet – COMP 2402 – Carleton University

Red-Black Trees – add(x)

add(**9**)



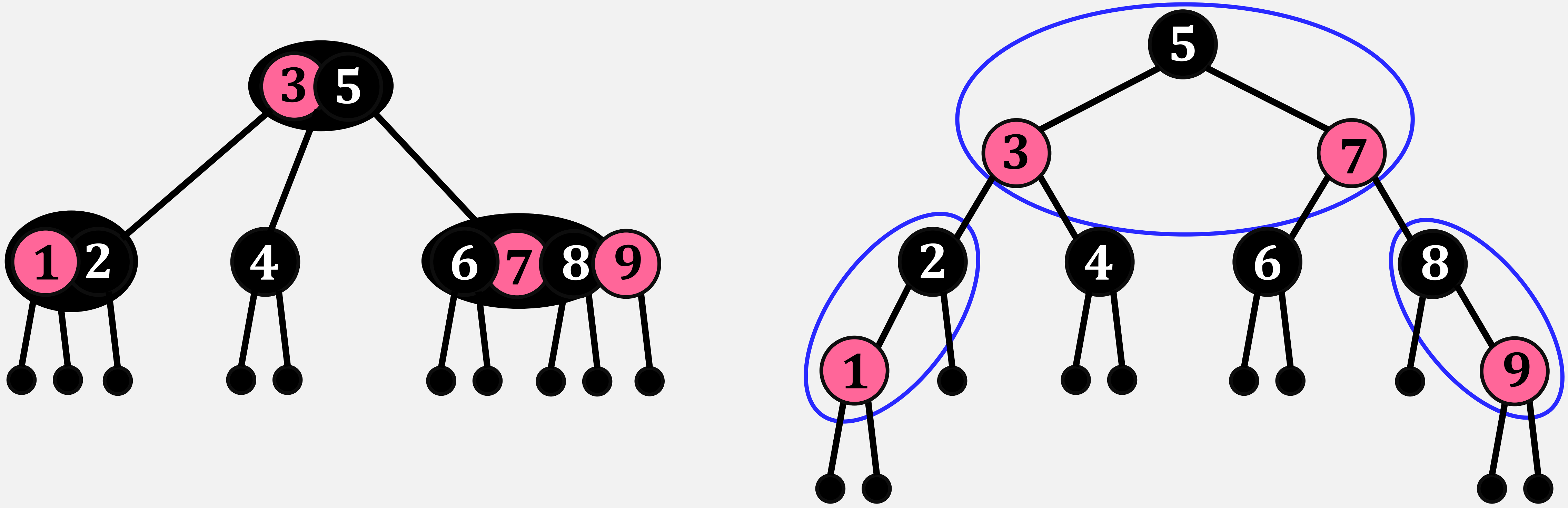


In a **red-black tree** we need a method of simulating splitting a node with five children in a **2-4 tree**:

We can "split" the node by recolouring: node **7** becomes red; nodes **6**, **8** become black

Red-Black Trees – add(x)

add(**9**)

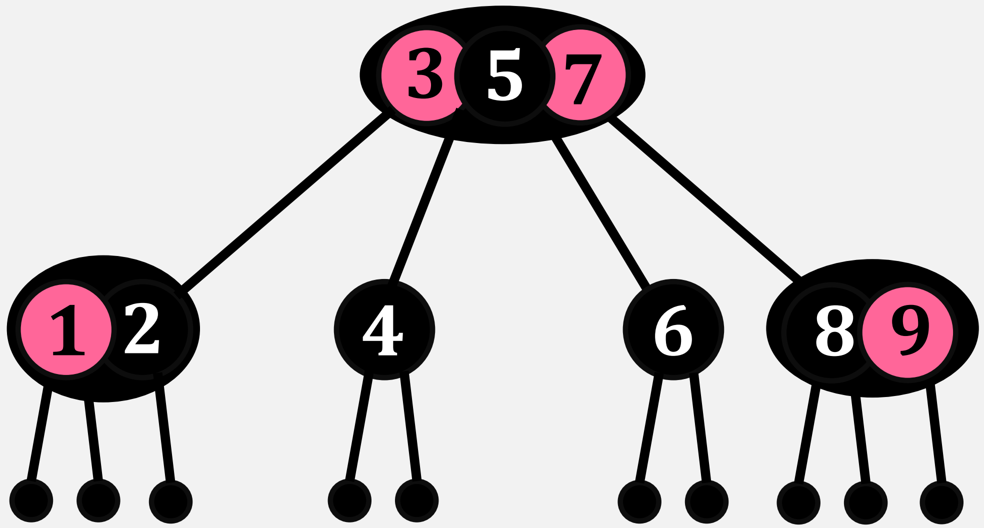


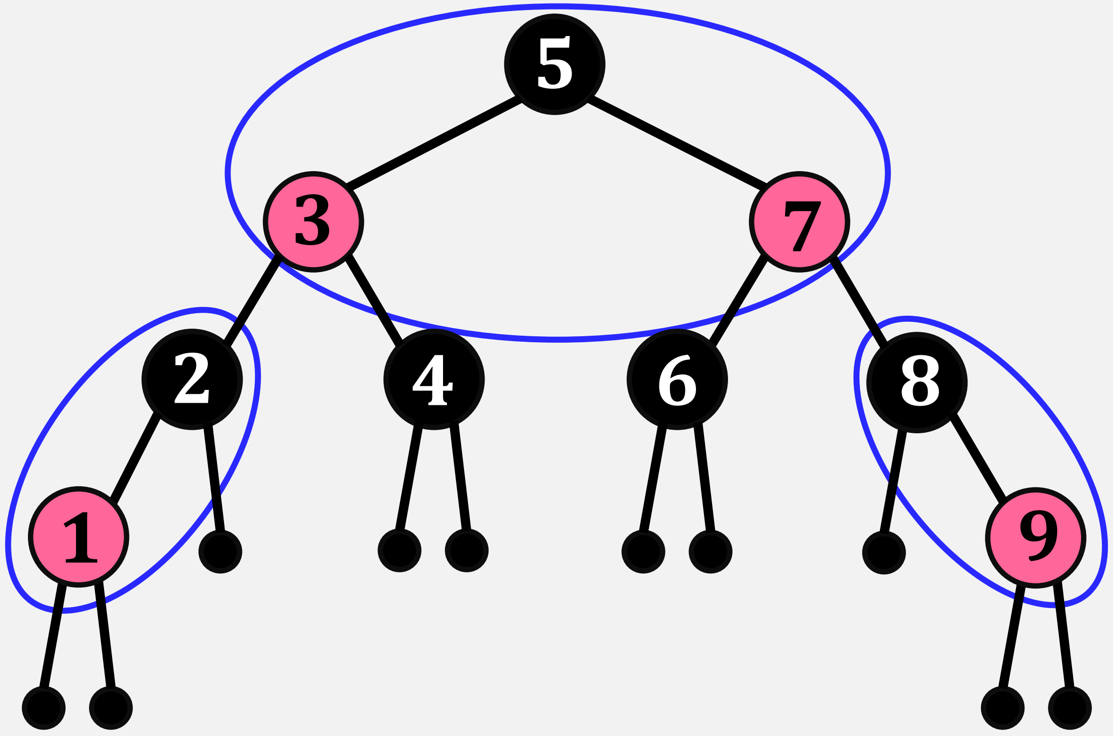
In a **red-black tree** we need a method of simulating splitting a node with five children in a **2-4 tree**:

We can "split" the node by recolouring: node **7** becomes red; nodes **6**, **8** become black

Red-Black Trees – add(x)

add(**9**)





In a **red-black tree** we need a method of simulating splitting a node with five children in a **2-4 tree**:

We can "split" the node by recolouring: node **7** becomes red; nodes **6**, **8** become black

Alina Shaikhet – COMP 2402 – Carleton University

Red-Black Trees – remove(x)

there are many cases that must be considered

To implement remove(x) we need to know:

1. how to merge two nodes and

   Merging two nodes is the inverse of a split. We can do it by colouring two (**black**) siblings **red** and colouring their (**red**) parent **black.**

2. how to borrow a child from a sibling

   Borrowing from a sibling is the most complicated of the procedures and involves both **rotations** and **recolouring** nodes.

We also need to maintain **no-red-edge** property and the **black-height** property.

Theorem 9.1

A **RedBlackTree** implements the **SSet** interface and supports the operations add($\boldsymbol{x}$), remove($\boldsymbol{x}$), and find($\boldsymbol{x}$) in $\boldsymbol{O}(\log \boldsymbol{n})$ worst-case time per operation.