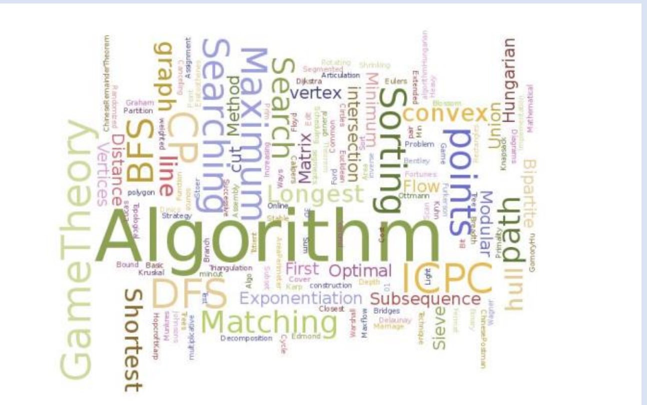
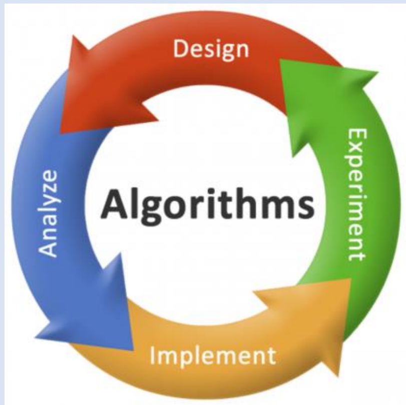
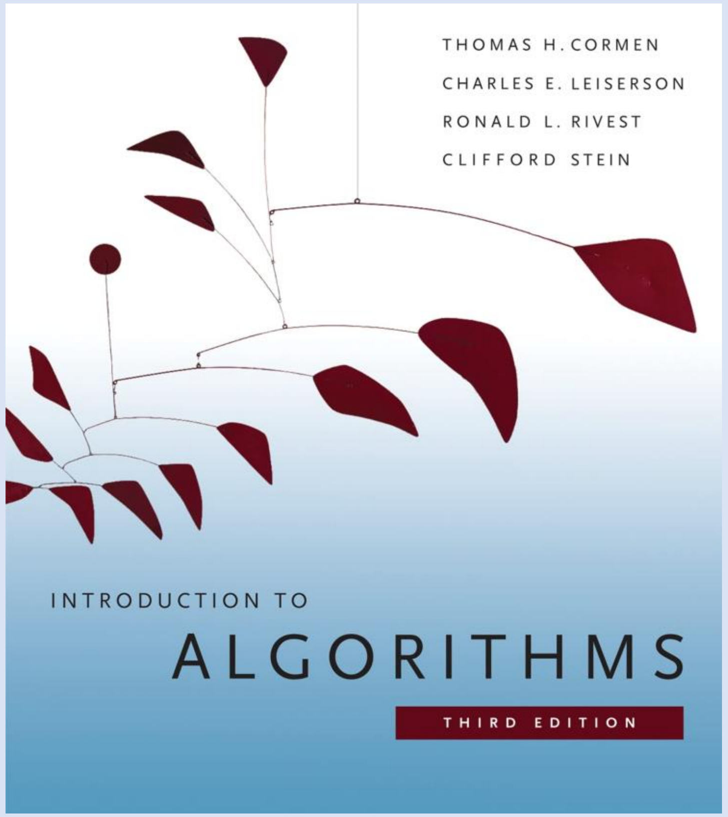
B-Trees

COP 3503
Spring 2025
Department of Computer Science
University of Central Florida
Dr. Steinberg

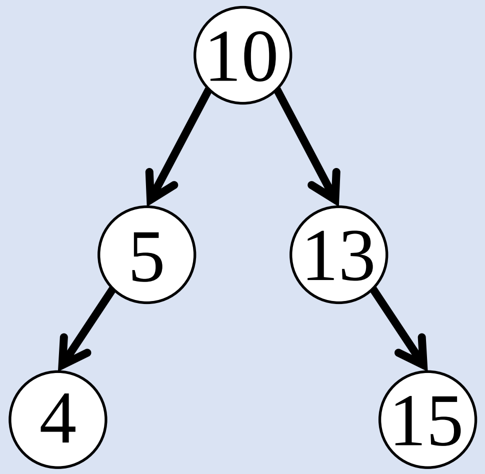
Reference

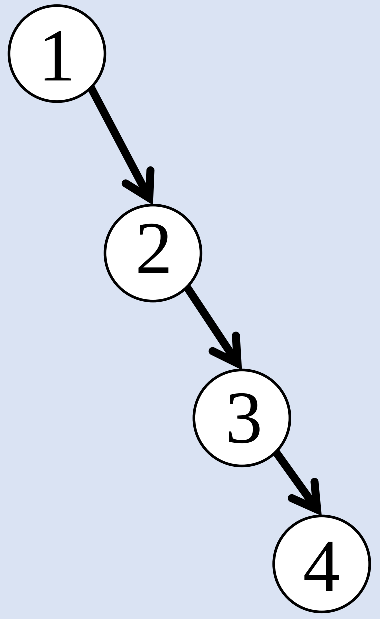
- The following presentation is referenced from the Cormen Introduction to Algorithms 3rd edition Textbook.



Introduction

- Binary Search Trees – Trees where a node has only at most two children. The value of the left node is smaller than the parent node value and the right node value is larger than the parent node.
- Examples:

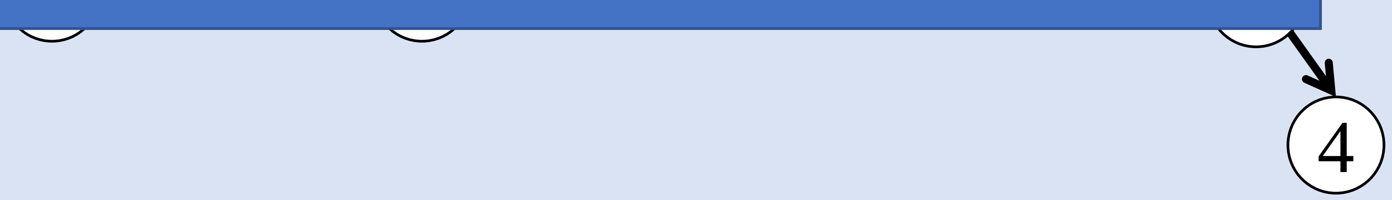




Introduction

- Binary Search Trees – Trees where a node has only at most two children. The value of the left node is smaller than the parent node value and the right node value is larger than the parent node.
- Examples:

What is wrong with the right Binary search tree?

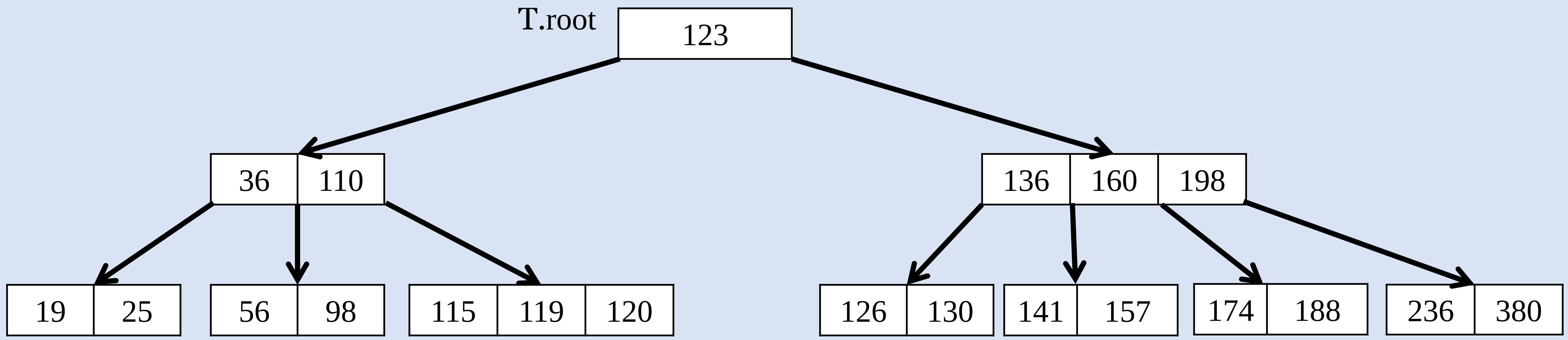


B-Trees

- Balanced Search Trees
- One con with a Binary Search Tree (BST) is that we can potentially have our algorithms run in linear time rather than height level time.
- B-Trees nodes have many children (few to even thousands!)
- This data structure is primarily used in disks and other direct-access secondary storage devices.
- Database systems use B-Trees or even variants.
- The height of a B-tree is $O(\lg n)$

B-Tree Sample

- In this sample, the keys are integers
- Each node has x.n keys and x.n + 1children
- How does a search work for key 126?



Data Structures on Secondary Storage

- Primary memory (main memory) consists of silicon memory chips.
- Secondary storage consists of magnetic storage
  - Tapes
  - Disks
- Disks are cheaper and have higher capacity than the main memory.
- Disks are slower than main memory due to motion mechanical components.

Disk Drive

- The average access time for the disk ranges from 8 to 11 milliseconds.
- The average access time in main memory is about 50 nanoseconds!
- Information on a disk is divided into pages which range from $2^{11} - 2^{14}$ bytes.
- Each disk reads and/or writes on a single or multiple pages.

B-Tree Applications

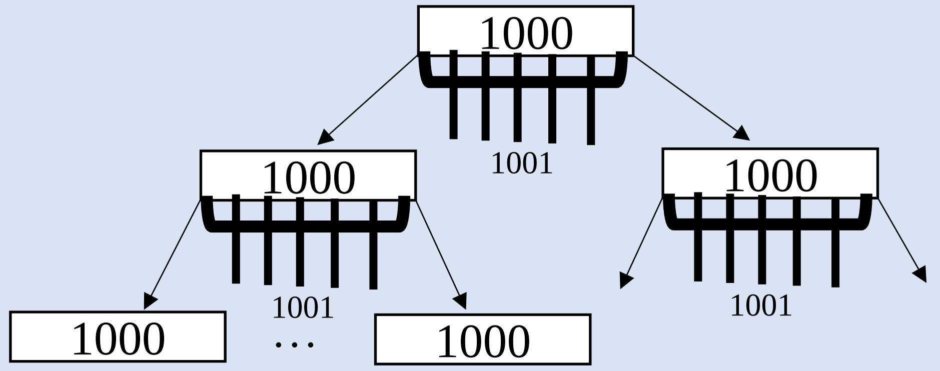
- The Whole B-Trees does not fit in the main memory!!!
- Operating Systems copies the pages from the disks into main memory. After performing tasks, the operating system writes back to the respective pages that were modified.

```
x = a pointer to some object
DISK-WRITE(x)
operations that access and/or modify attributes of x

DISK-READ(x)
other operations that access but do not modify attributes of x
```

B-Tree Example

- Branch Factor = 1001
- Height = 2



- B-Trees often have branching factors ranging from 50 -2000
- Root node is permanently in main memory in order to find any key with at most two disk accesses.

B-Tree Example

- Branch Fac
- Height = 2

Think About it…
Imagine trying to store this in RAM!!
CRAZY!!!!!!!!!!

- B-Trees oft 00
- Root node is permanently in main memory in order to find any key with at most two disk accesses.

The Official B-Tree Definition

- A B-Tree is a rooted tree (where T.root is the root) with the following properties:
  1. Every node x has the following attributes
     a) x.n is the number of keys currently stored in x
     b) The keys $x.key_1, x.key_2, \ldots, x.key_{x.n}$ such that
        $$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$
     c) x.leaf – a Boolean value which is true if x is a leaf and false if x is an internal node
  2. Each internal node x has $x.n + 1$ pointers $x._{c1}, x._{c2}, \ldots, x._{cx.n+1}$ to its children. If x is a leaf then the pointers are undefined.
  3. If k_i is any key stored in the subtree with root $x.c_i$ then:
     $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

The official B-Tree Definition Continued

- All leaves have the same depth, which is the tree high h.
- The B-Tree has a minimum degree t (where t is an integer t >= 2):
  - Every node other than the root must have >= t – 1 keys and >= t children; if B-tree is nonempty, then the root has at least one key
  - Every node has $\leq 2t - 1$ keys and $\leq 2t$ children
    A node is considered full if it has 2t – 1 keys inserted.

Interesting Theorem About Height in B-Trees

- Theorem: if $n \geq 1$, then for any n-key B-tree T of height h and minimum degree t,

$$h \leq \log_t \frac{n+1}{2}$$

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1}$$

$$= 1 + 2(t-1)\ \frac{t^h - 1}{t-1} = 2t^h - 1$$

$$t^h \leq \frac{n+1}{2}$$

$$h \leq \log_t \frac{n+1}{2}$$

$h = O(\log n)$

Theorem cont.

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

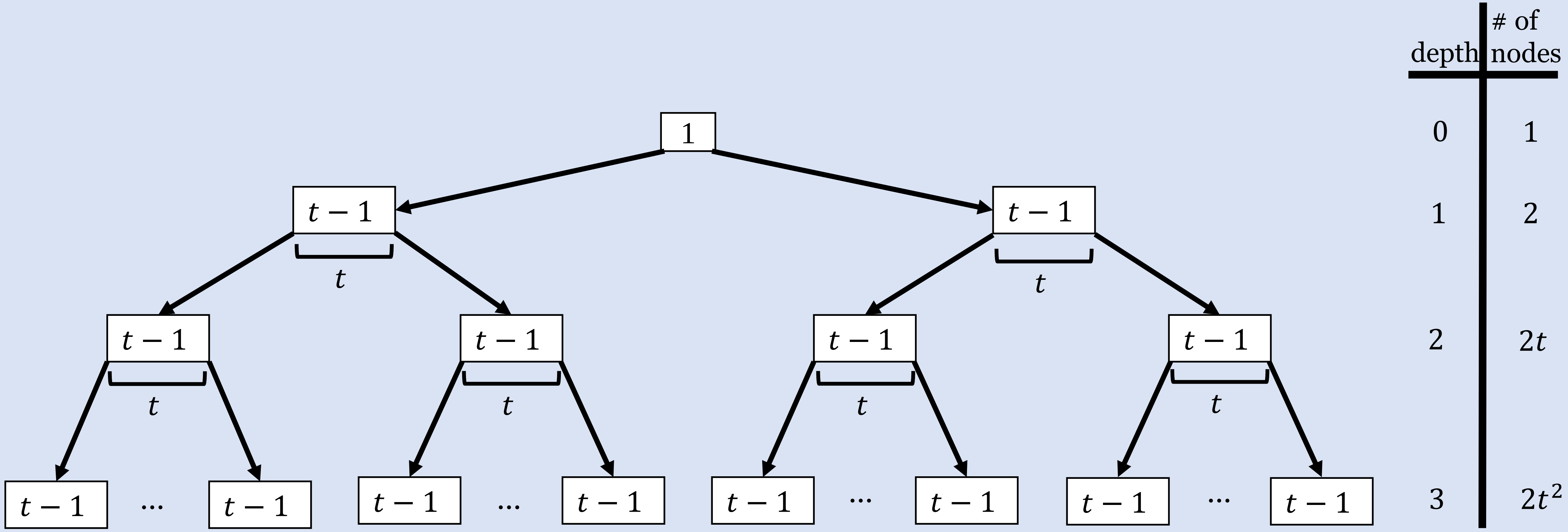
$$= 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1}$$

$$= 1 + 2(t-1)\ \frac{t^h - 1}{t-1} = 2t^h - 1$$

$$t^h \leq \frac{n+1}{2}$$

$$h \leq \log_t \frac{n+1}{2}$$

$h = O(\log n)$


1
$t-1$
t
...
depth
# of nodes
0
1
2
$2t$
3
$2t^2$

Operations We Will Observe for B-Trees

- B-Tree-Search
- B-Tree-Create
- B-Tree-Insert
- B-Tree-Delete

B-Tree Search

```
**B-Tree-Search(x, k)**
i = 1
while $i \leq x.n$ and $k \ > x$
    i = i + 1
if $i \leq x.n$ and k == $x.key_i$
    return (x, i)
else if x.leaf == True
    return NULL
else DISK-READ($x.c_i$)
    return B-Tree-Search($x.c_i, k$)
```

RT: $O(tlog_t n)$

B-Tree-Create

- Creating an empty tree with root node

```
B-Tree-Create(T)
x = Allocate-Node()
x.leaf = True
x.n = 0
Disk-Write(x)
T.root = x
```

B-Tree Insert Operations

- The insert operation has 3 functions/methods we need to understand
- B-Tree-Split-Child(x,i)
- B-Tree-Insert(T,k)
- B-Tree-Insert-Nonfull(x,k)

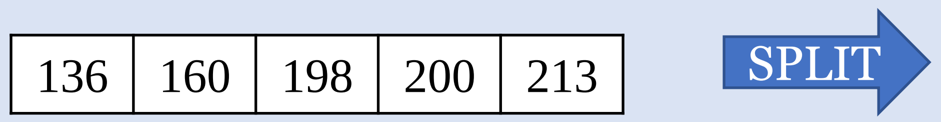
Insert Operation and Overall Goal

- Search for a leaf where to put new key
- Inserting into an existing leaf node
  - Cannot create a new leaf
- If the leaf node is full, then split around the median key
- The overall goal is to insert they key while maintaining B-Tree rules. As the algorithms traverses down the tree, it splits each full node along the way, including the leaf.

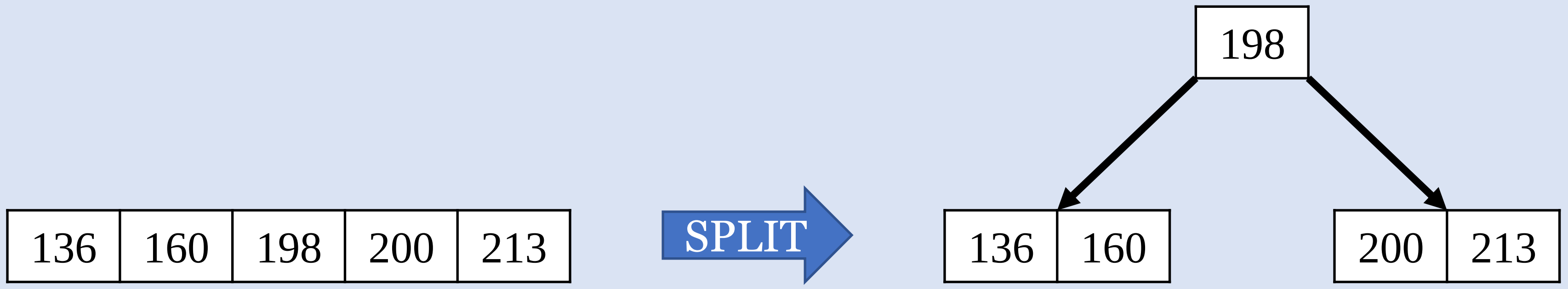
Splitting a Node

136	160	198	200	213

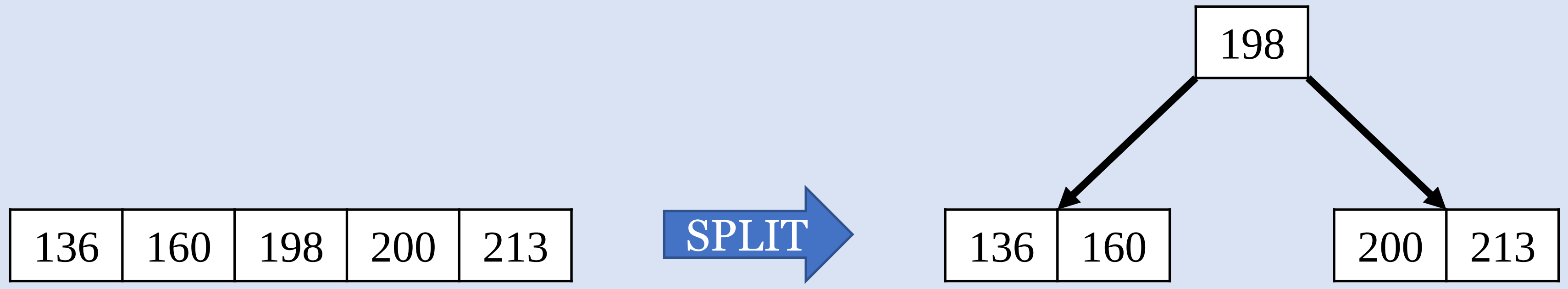
Splitting a Node

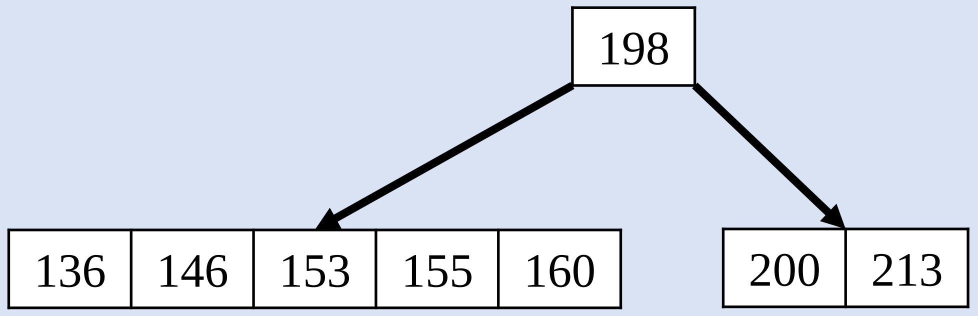


Splitting a Node

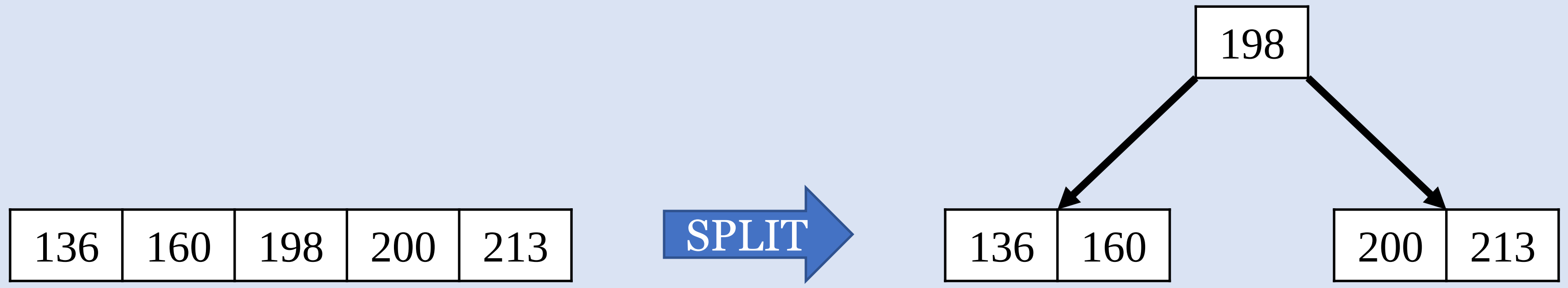


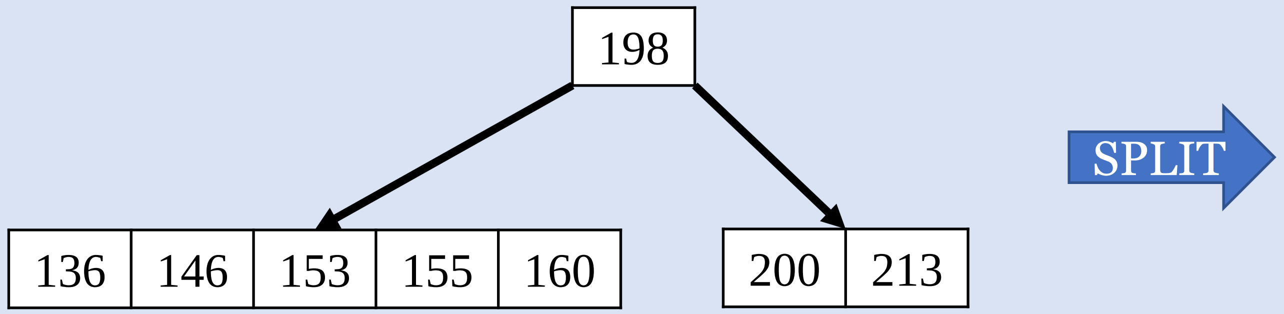
Splitting a Node



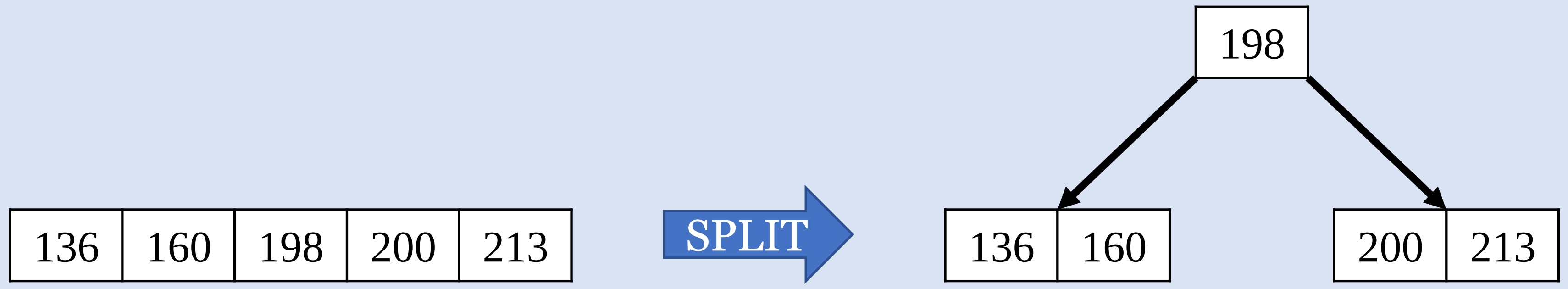


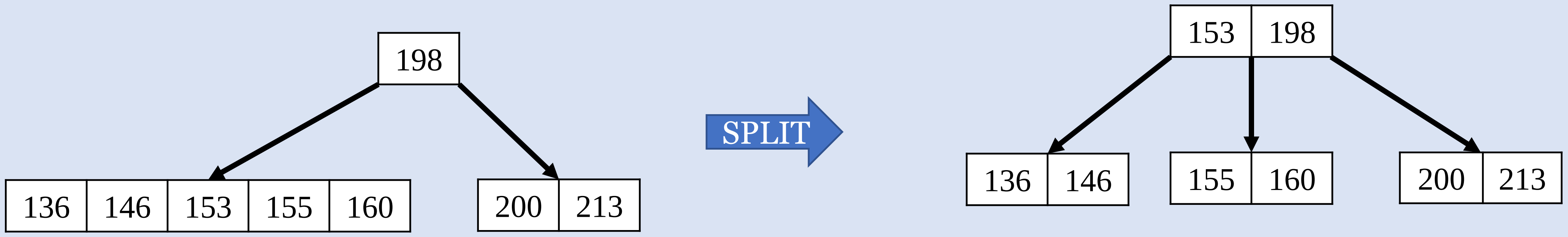
Splitting a Node





Splitting a Node

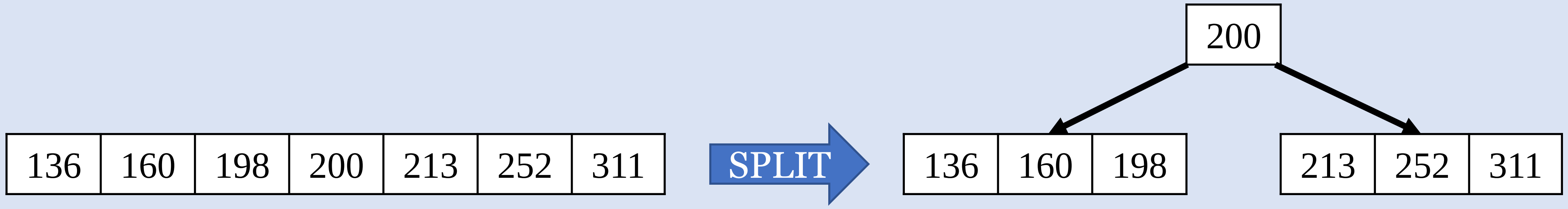




```
B-TREE-SPLIT-CHILD$(x, i)$
$z$ = ALLOCATE-NODE()
$y = x.c_i$
$z.leaf = y.leaf$
$z.n = t - 1$
for $j = 1$ to $t - 1$
    $z.key_j = y.key_{j+t}$
if not $y.leaf$
    for $j = 1$ to $t$
        $z.c_j = y.c_{j+t}$
$y.n = t - 1$
for $j = x.n + 1$ downto $i + 1$
    $x.c_{j+1} = x.c_j$
$x.c_{i+1} = z$
for $j = x.n$ downto $i$
    $x.key_{j+1} = x.key_j$
$x.key_i = y.key_t$
$x.n = x.n + 1$
DISK-WRITE$(y)$
DISK-WRITE$(z)$
DISK-WRITE$(x)$
```

B-Tree-Insert()

- t = 4
  range of keys 3-7



- If root node is full, then split the root and new node will become the root

```
B-TREE-INSERT($T, k$)
    $r = T.root$
    if $r.n == 2t - 1$
        $s =$ ALLOCATE-NODE()
        $T.root = s$
        $s.leaf =$ FALSE
        $s.n = 0$
        $s.c_1 = r$
        B-TREE-SPLIT-CHILD($s, 1$)
        B-TREE-INSERT-NONFULL($s, k$)
    else B-TREE-INSERT-NONFULL($r, k$)
```

```
B-TREE-INSERT-NONFULL($x, k$)
    $i = x.n$
    if $x.leaf$
        while $i \geq 1$ and $k < x.key_i$
            $x.key_{i+1} = x.key_i$
            $i = i - 1$
        $x.key_{i+1} = k$
        $x.n = x.n + 1$
        DISK-WRITE($x$)
    else while $i \geq 1$ and $k < x.key_i$
            $i = i - 1$
        $i = i + 1$
        DISK-READ($x.c_i$)
        if $x.c_i.n == 2t - 1$
            B-TREE-SPLIT-CHILD($x, i$)
            if $k > x.key_i$
                $i = i + 1$
        B-TREE-INSERT-NONFULL($x.c_i, k$)
```

RT for Insert

- $O(th) = O(t log_t n)$

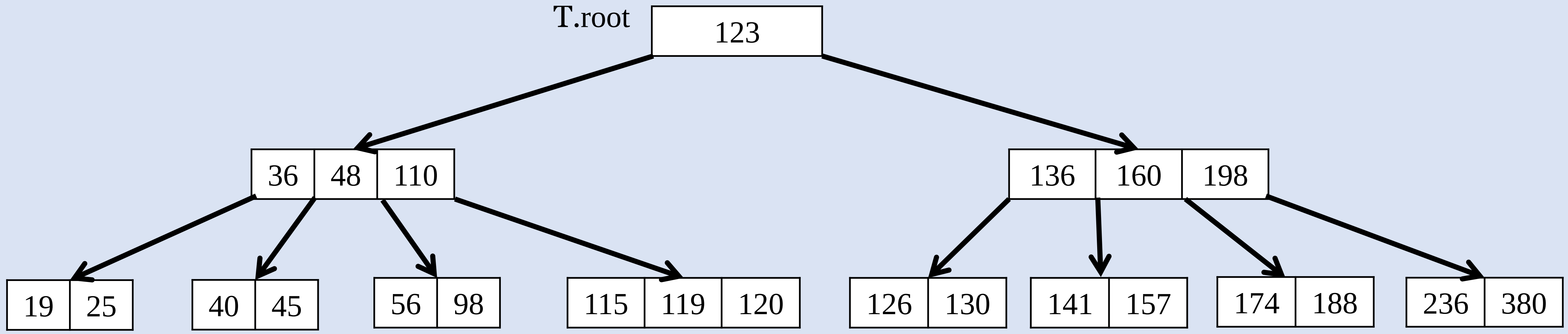
B-Tree-Delete()

- Important things to remember!
  - When a key is removed, we must rearrange the node's children!
  - Any node (EXCEPT FOR THE ROOT) cannot have fewer than t -1 keys
  - The algorithm deletes a key k from the subtree rooted at x
  - Something to consider: When delete is called on a node, we should guarantee that the number of keys in x is greater than or equal to t.
- The overall objective is to remove a key while maintaining the B-tree properties.
- There are 3 rules to consider when deleting from the B-Tree

Rule 1

- If the key *k* is part of a leaf node *x*, then just delete the key.
  t = 2

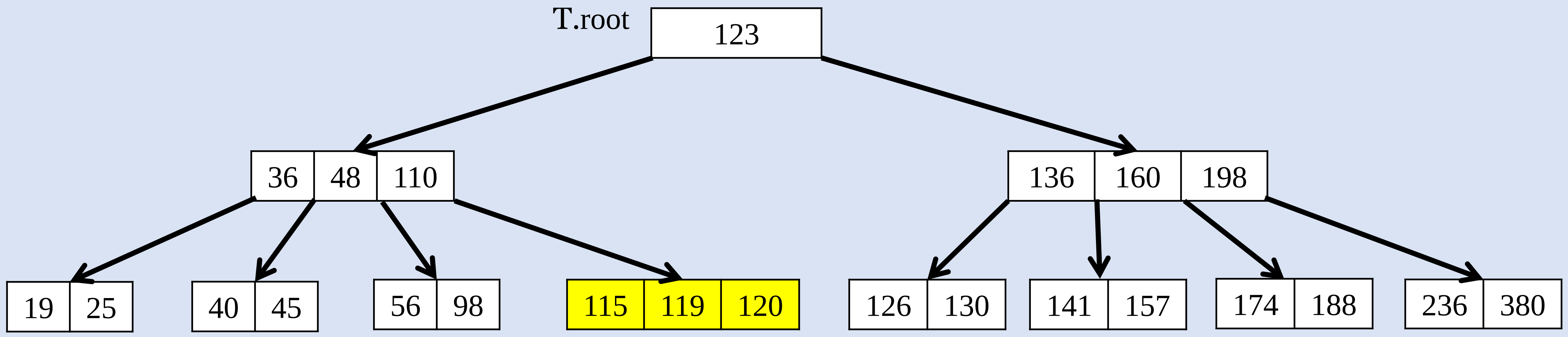
Delete 120



Rule 1

- If the key *k* is part of a leaf node *x*, then just delete the key.
  t = 2

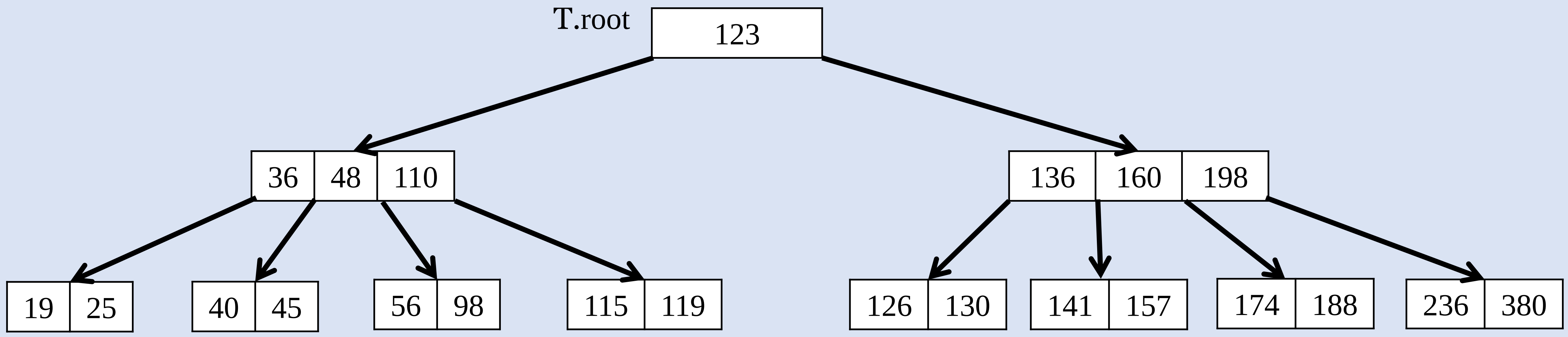
Delete 120



Rule 1

- If the key *k* is part of a leaf node *x*, then just delete the key.
  t = 2

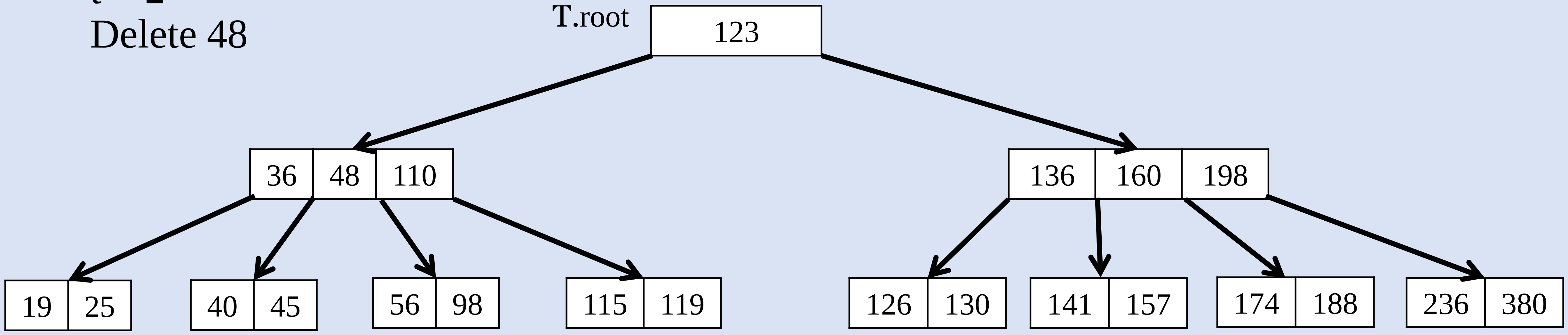
Delete 120



Rule 2a

- If the key *k* belongs to an internal node *x*.
- If the child *y* that precedes *k* in a node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the subtree rooted at *y*. Recursively delete *k'* and replace *k* by *k'* in *x*.

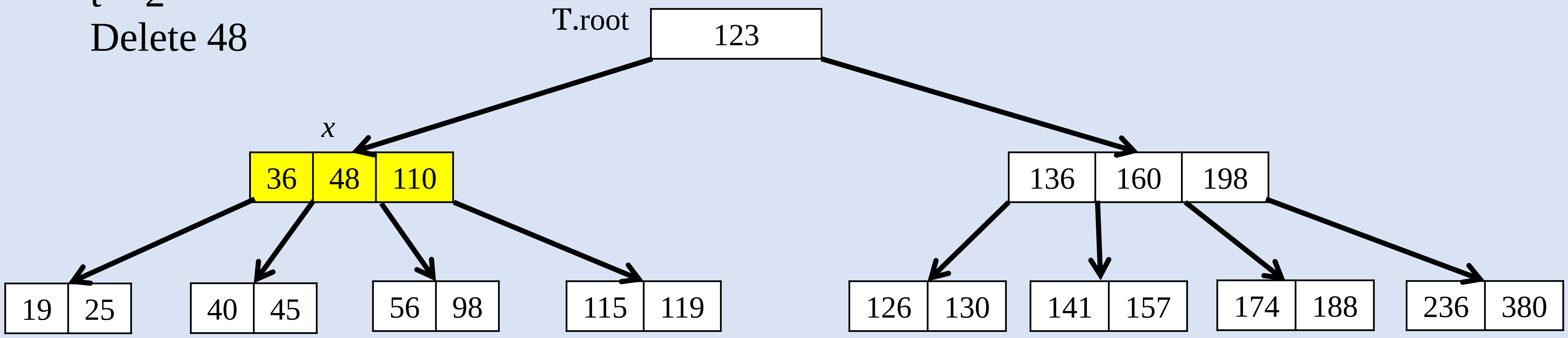
t = 2
Delete 48



Rule 2a

- If the key *k* belongs to an internal node *x*.
- If the child *y* that precedes *k* in a node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the subtree rooted at *y*. Recursively delete *k'* and replace *k* by *k'* in *x*.

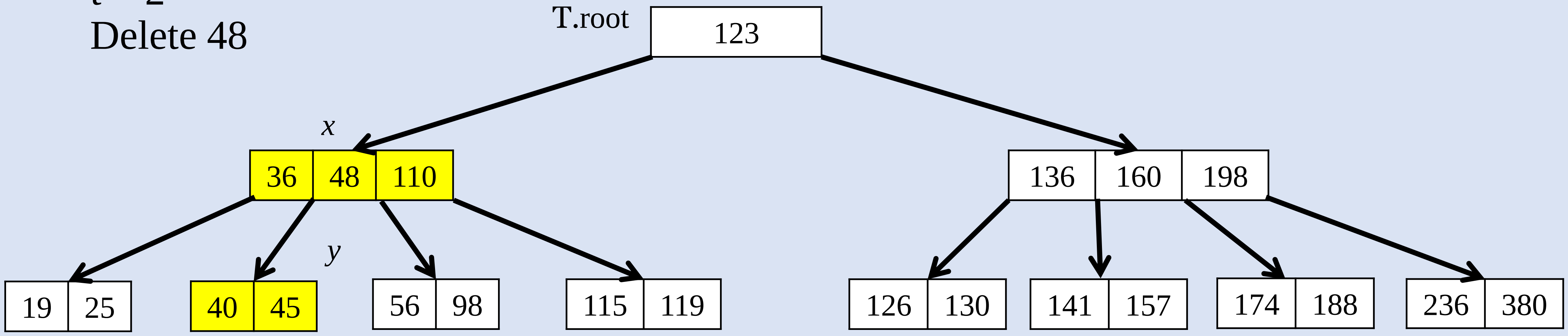
t = 2
Delete 48



Rule 2a

- If the key *k* belongs to an internal node *x*.
- If the child *y* that precedes *k* in a node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the subtree rooted at *y*. Recursively delete *k'* and replace *k* by *k'* in *x*.

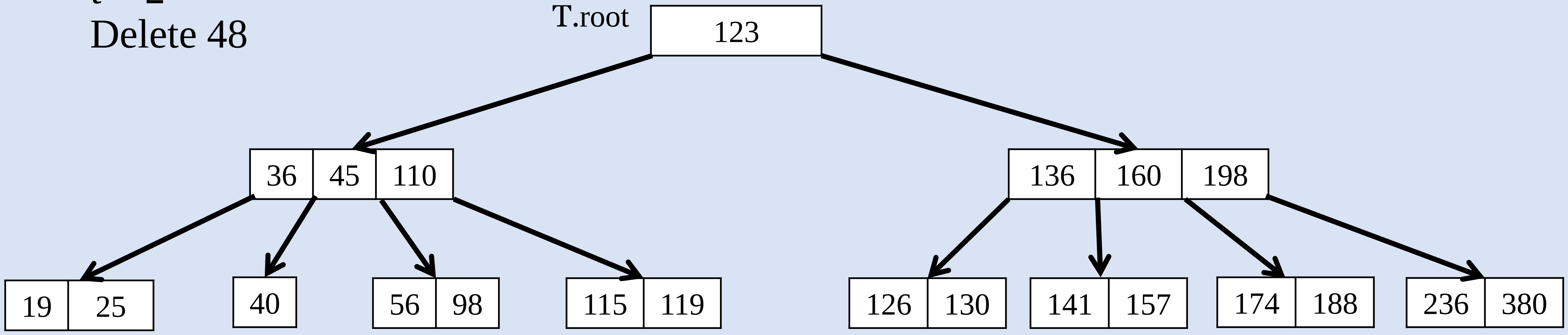
t = 2
Delete 48



Rule 2a

- If the key *k* belongs to an internal node *x*.
- If the child *y* that precedes *k* in a node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the subtree rooted at *y*. Recursively delete *k'* and replace *k* by *k'* in *x*.

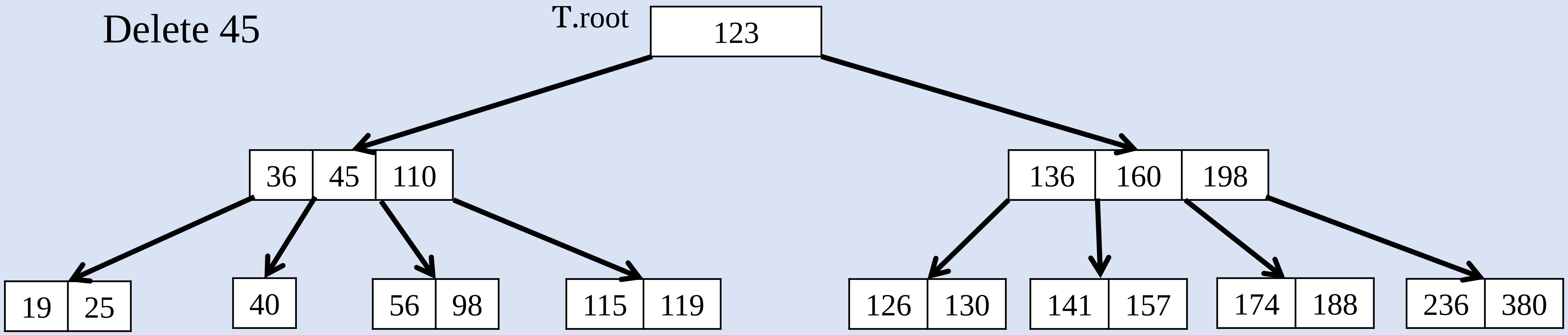
t = 2
Delete 48



Rule 2b

- If the key *k* belongs to an internal node.
- If *y* has fewer than *t* keys, then, symmetrically, examine the child *z* that follows *k* in node *x*. If *z* has at least *t* keys, then find the successor *k'* of *k* in the subtree rooted at *z*. Recursively delete *k'* and replace *k* by *k'* in *x*.

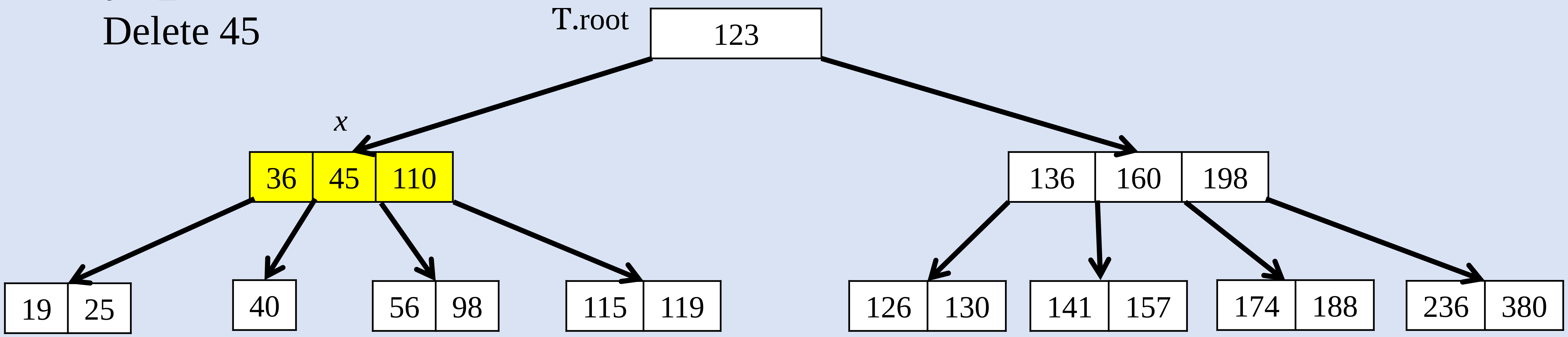
t = 2
Delete 45



Rule 2b

- If the key *k* belongs to an internal node.
- If *y* has fewer than *t* keys, then, symmetrically, examine the child *z* that follows *k* in node *x*. If *z* has at least *t* keys, then find the successor *k'* of *k* in the subtree rooted at *z*. Recursively delete *k'* and replace *k* by *k'* in *x*.

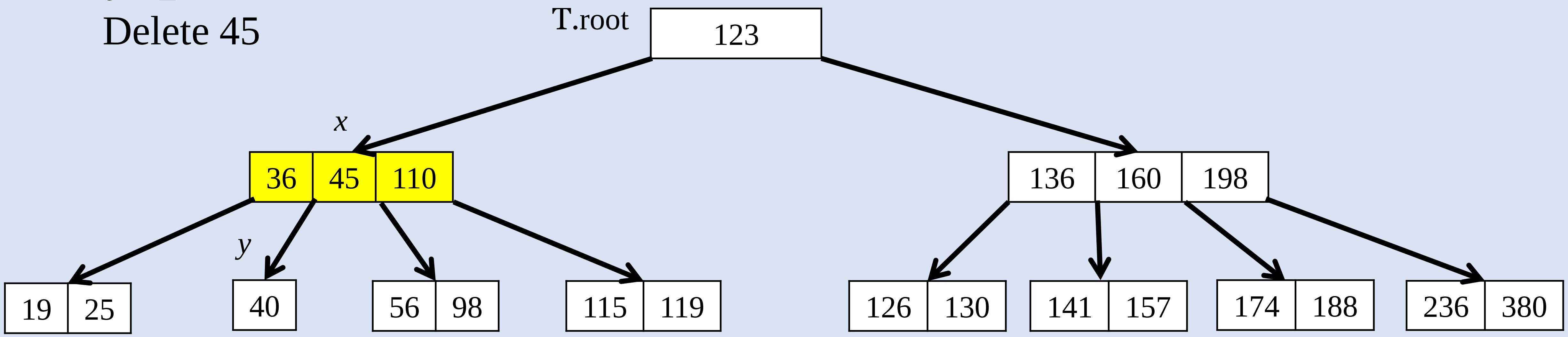
t = 2
Delete 45



Rule 2b

- If the key *k* belongs to an internal node.
- If *y* has fewer than *t* keys, then, symmetrically, examine the child *z* that follows *k* in node *x*. If *z* has at least *t* keys, then find the successor *k'* of *k* in the subtree rooted at *z*. Recursively delete *k'* and replace *k* by *k'* in *x*.

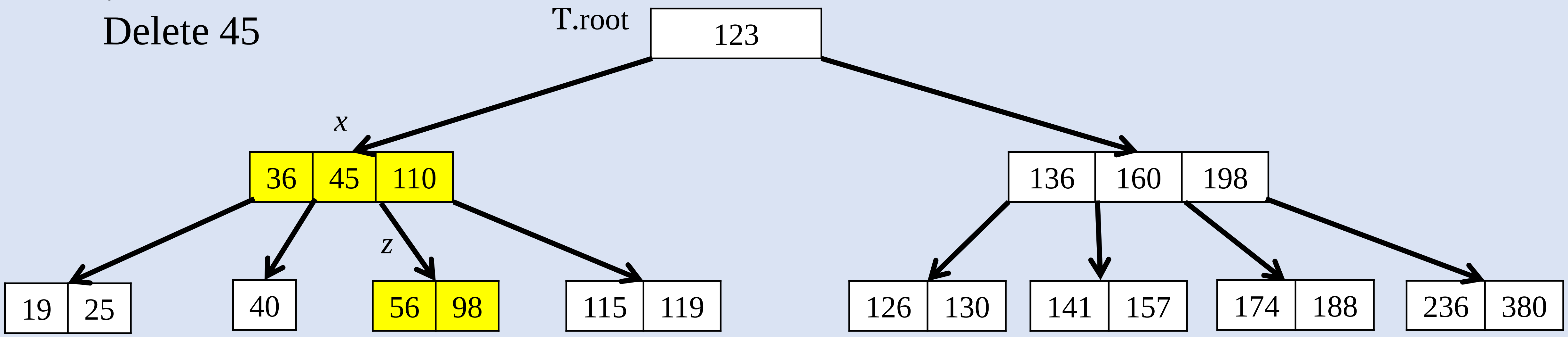
t = 2
Delete 45



Rule 2b

- If the key *k* belongs to an internal node.
- If *y* has fewer than *t* keys, then, symmetrically, examine the child *z* that follows *k* in node *x*. If *z* has at least *t* keys, then find the successor *k'* of *k* in the subtree rooted at *z*. Recursively delete *k'* and replace *k* by *k'* in *x*.

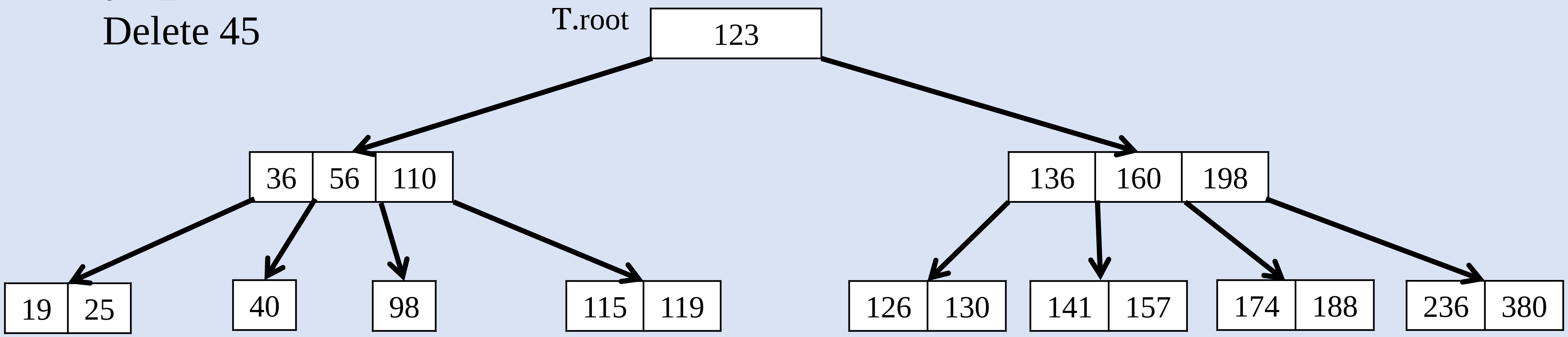
t = 2
Delete 45



Rule 2b

- If the key *k* belongs to an internal node.
- If *y* has fewer than *t* keys, then, symmetrically, examine the child *z* that follows *k* in node *x*. If *z* has at least *t* keys, then find the successor *k'* of *k* in the subtree rooted at *z*. Recursively delete *k'* and replace *k* by *k'* in *x*.

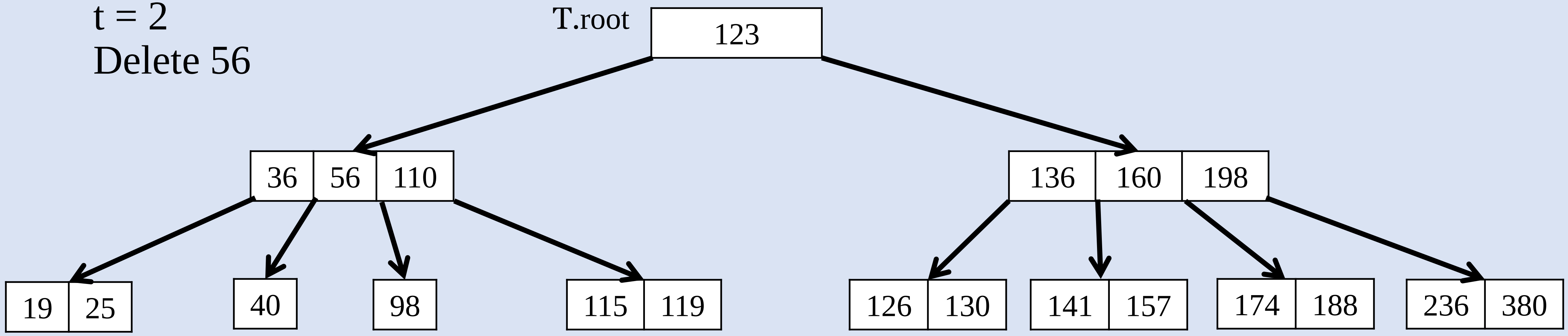
t = 2
Delete 45



Rule 2c

- If the key *k* belongs to an internal node *x*.
- Otherwise, if both *y* and *z* have only *t – 1* keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains *2t -1* keys. Then free *z* and recursively delete *k* from *y*.

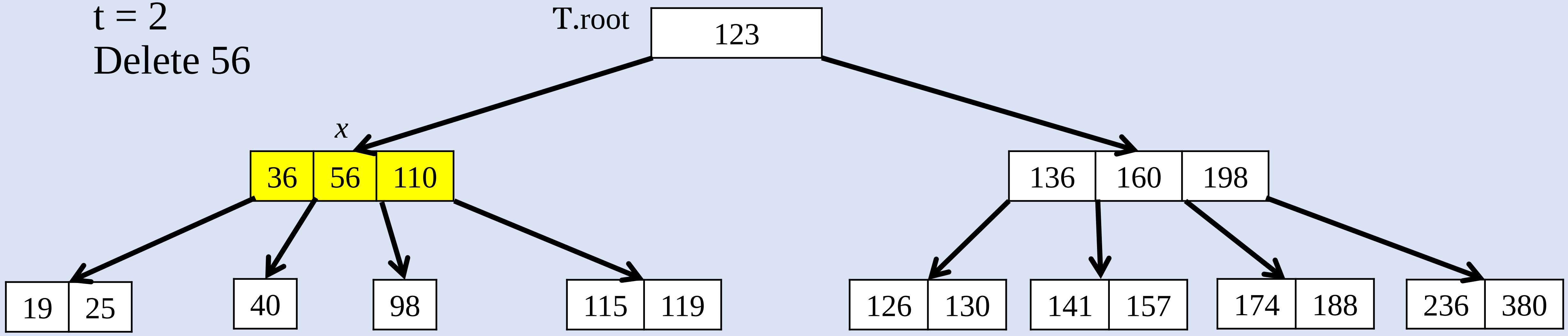
t = 2
Delete 56



Rule 2c

- If the key *k* belongs to an internal node *x*.
- Otherwise, if both *y* and *z* have only *t – 1* keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains *2t -1* keys. Then free *z* and recursively delete *k* from *y*.

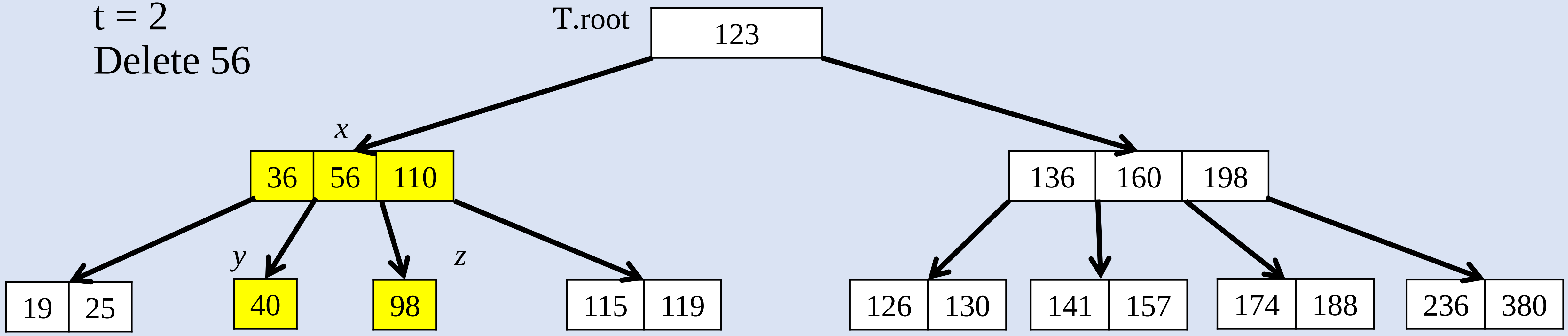
t = 2
Delete 56



Rule 2c

- If the key *k* belongs to an internal node *x*.
- Otherwise, if both *y* and *z* have only *t – 1* keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains *2t -1* keys. Then free *z* and recursively delete *k* from *y*.

t = 2
Delete 56

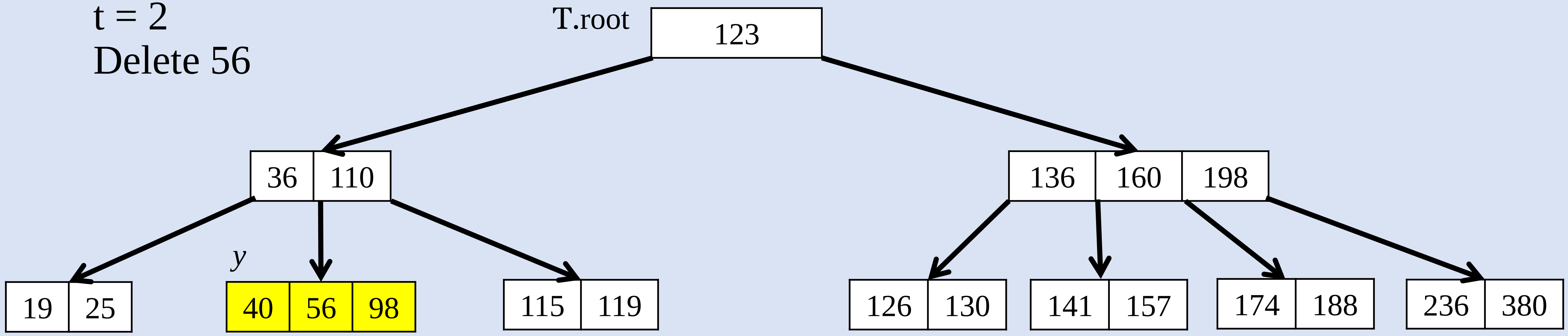


Both Nodes have t – 1 keys

Rule 2c

- If the key *k* belongs to an internal node *x*.
- Otherwise, if both *y* and *z* have only *t – 1* keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains *2t -1* keys. Then free *z* and recursively delete *k* from *y*.

t = 2
Delete 56

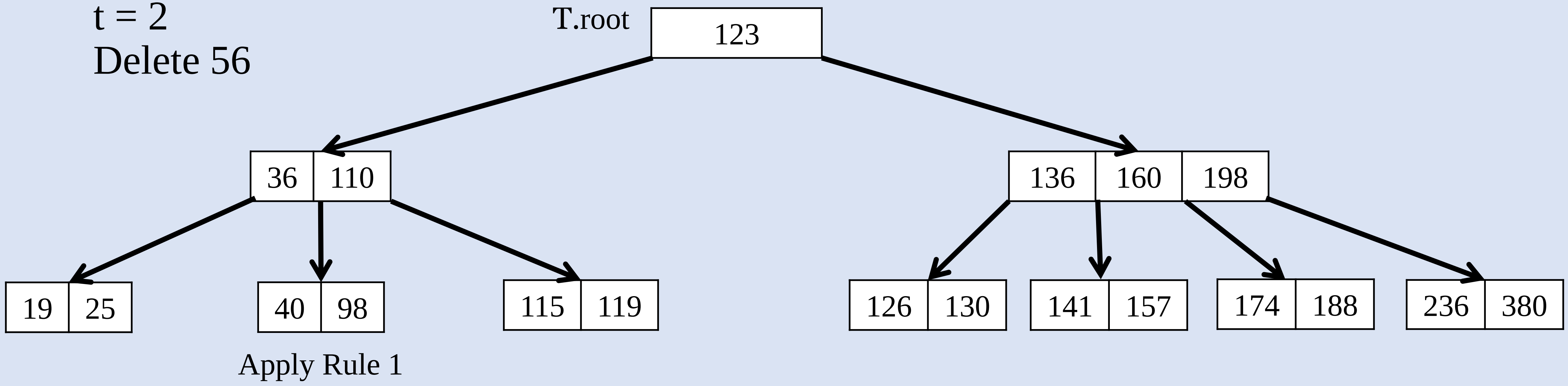


Pull Down and Merge

Rule 2c

- If the key *k* belongs to an internal node *x*.
- Otherwise, if both *y* and *z* have only *t – 1* keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains *2t -1* keys. Then free *z* and recursively delete *k* from *y*.

t = 2
Delete 56



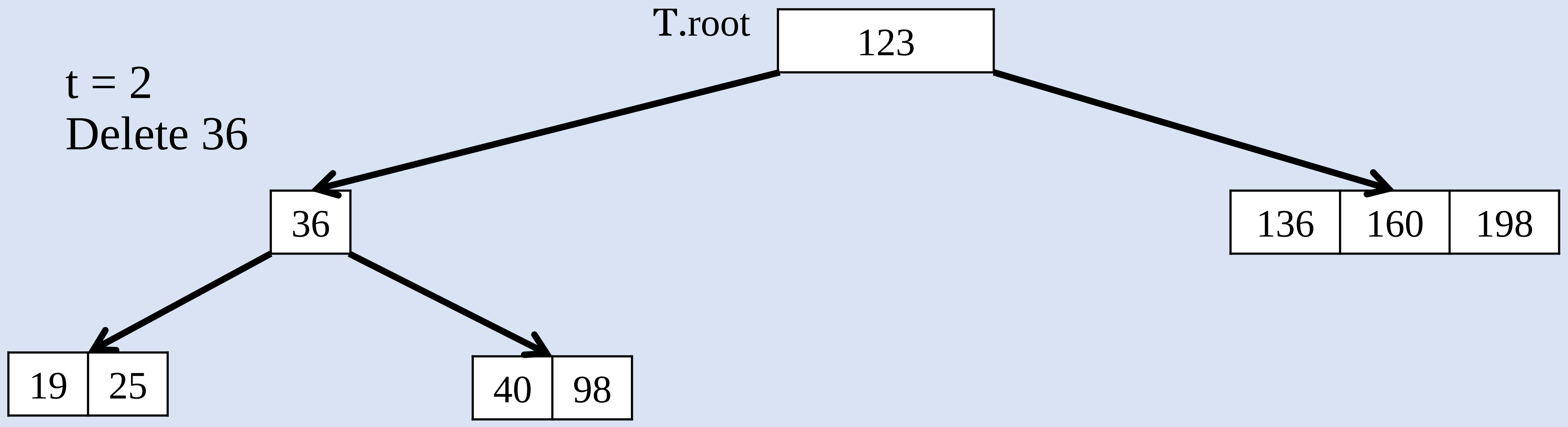
Rule 3a

- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from x to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up x
- Moving the appropriate child pointer from the sibling into $x.c_i$

t = 2
Delete 36



Rule 3a

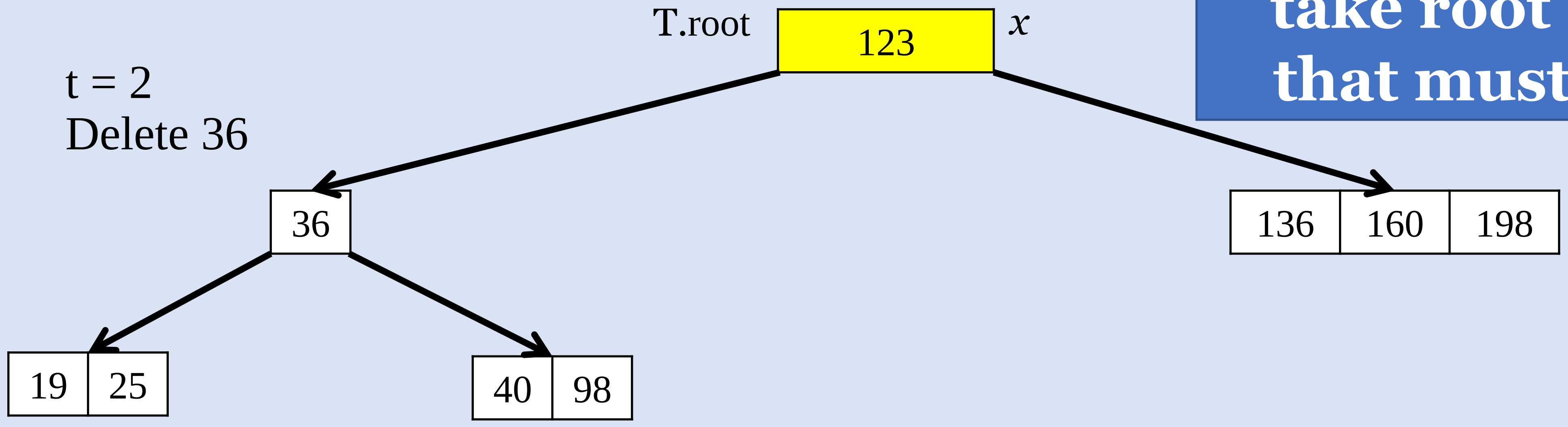
- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from x to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up x
- Moving the appropriate child pointer from the sibling into $x.c_i$

Key is not in highlighted node… take root of subtree that must contain *k*

t = 2
Delete 36



Rule 3a

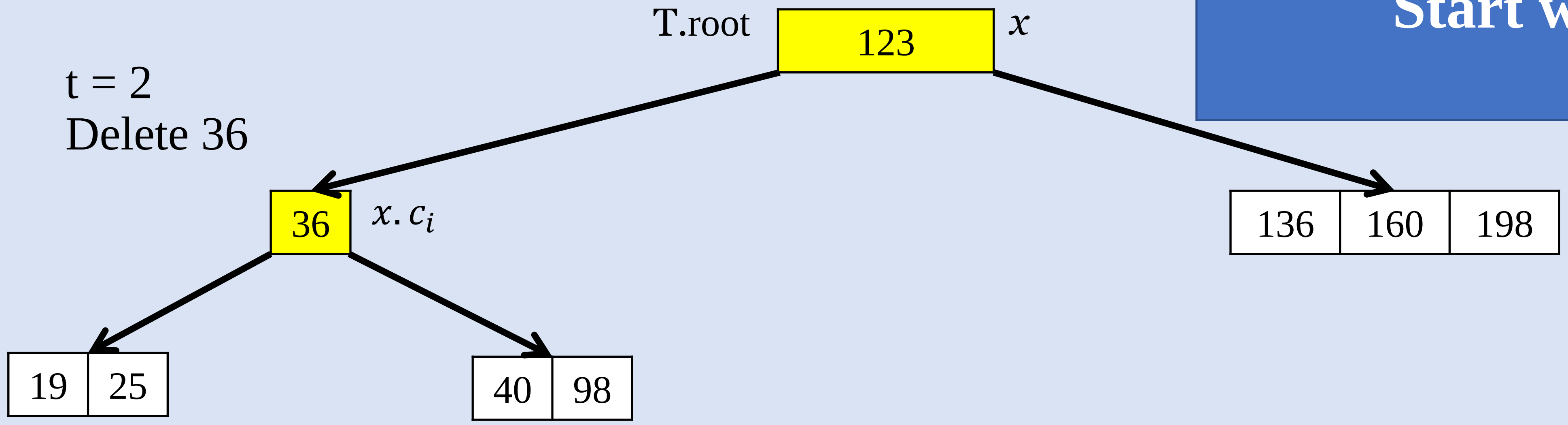
- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from x to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up x
- Moving the appropriate child pointer from the sibling into $x.c_i$

$x.c_i$ has $t-1$. Apply Rule 3. Start with 3A.

t = 2
Delete 36



Rule 3a

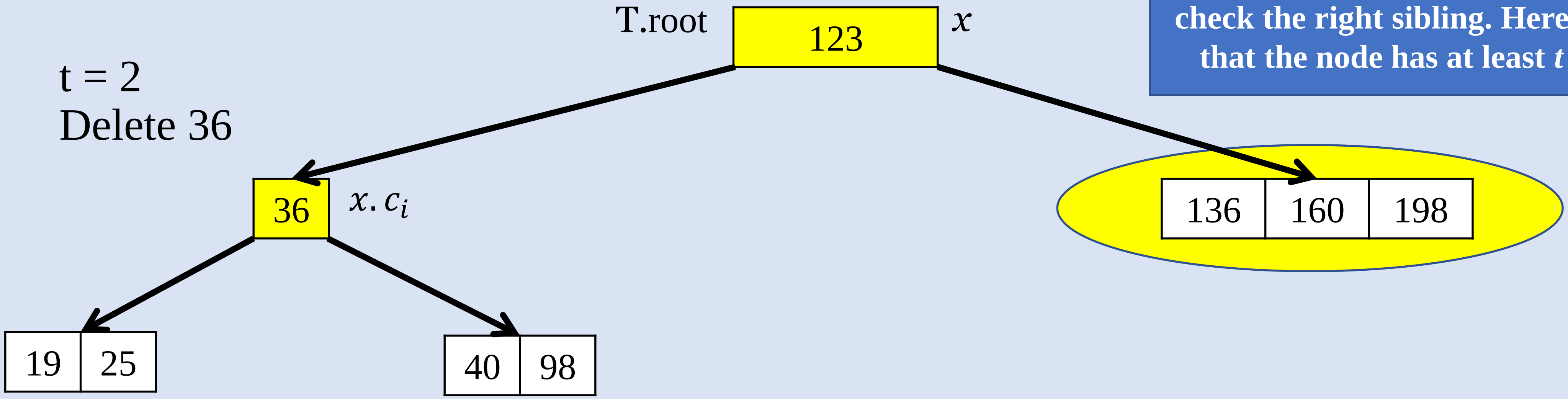
- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from *x* to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up *x*
- Moving the appropriate child pointer from the sibling into $x.c_i$

Start with left sibling. However there is no left sibling. Now we check the right sibling. Here we see that the node has at least *t* keys.

t = 2
Delete 36



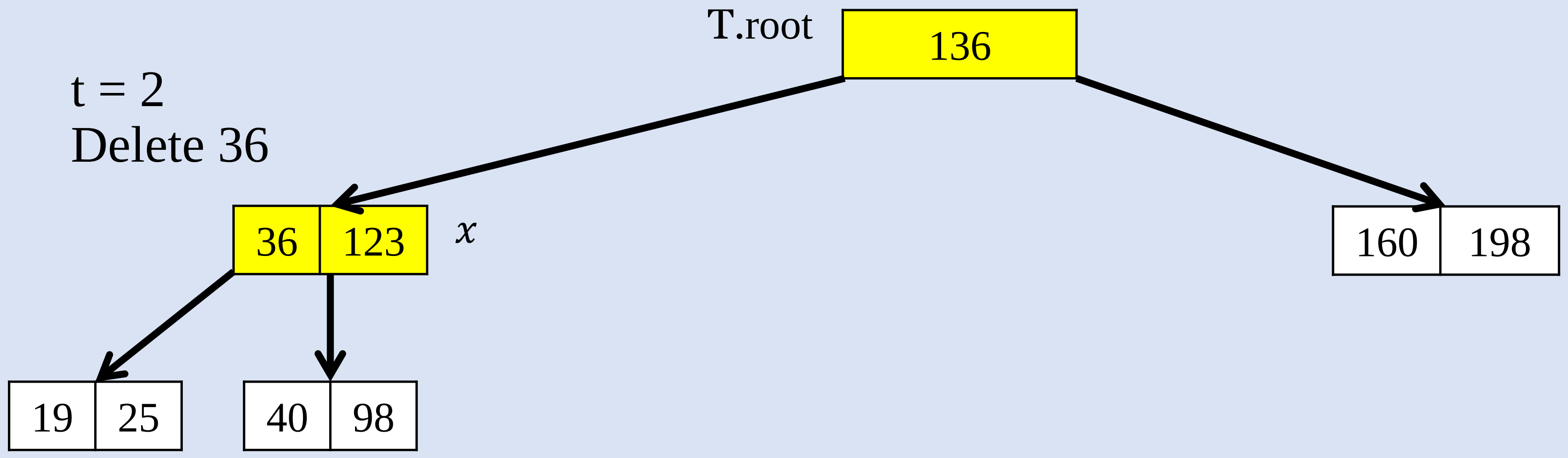
Rule 3a

- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from *x* to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up *x*
- Moving the appropriate child pointer from the sibling into $x.c_i$

t = 2
Delete 36



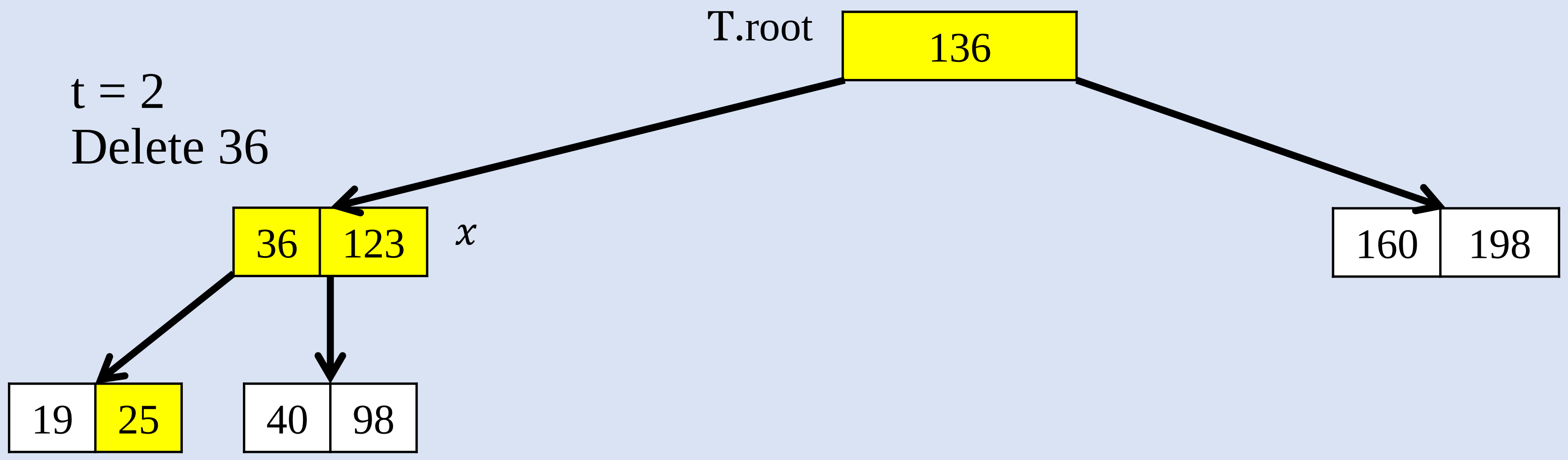
Rule 3a

- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from *x* to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up *x*
- Moving the appropriate child pointer from the sibling into $x.c_i$

t = 2
Delete 36



Apply Rule 2A

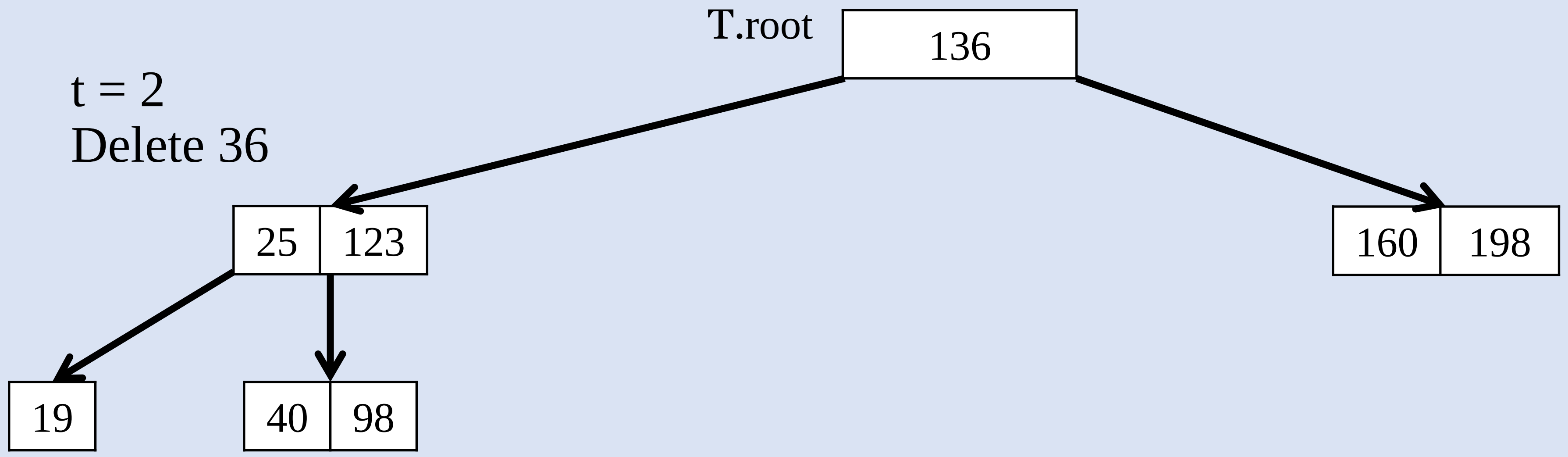
Rule 3a

- If the key *k* is not part of the internal node *x*, take $x.c_i$ the root of the subtree that must contain *k* (if *k* is in the tree). If $x.c_i$ has only *t-1* keys, then use 3a or 3b to guarantee we descend to a node with greater than or equal to *t* keys.

If $x.c_i$ has an immediate sibling with greater than or equal to t keys, then give $x.c_i$ an extra key by:

- Moving a key from *x* to $x.c_i$
- Moving a key from $x.c_i$'s immediate left or right sibling up *x*
- Moving the appropriate child pointer from the sibling into $x.c_i$

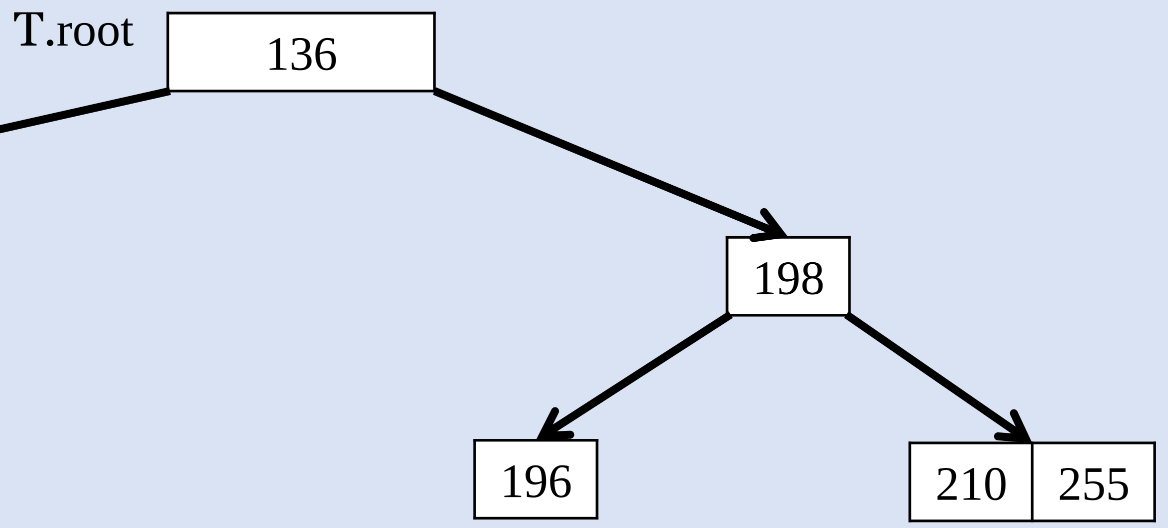
t = 2
Delete 36



Rule 3b

- If both $x.c_i$'s immediate sibling have *t – 1* keys, merge $x.c_i$ with one sibling, which involves moving a key from *x* down into the new merged node to become the median for that node

Delete 25
t = 2

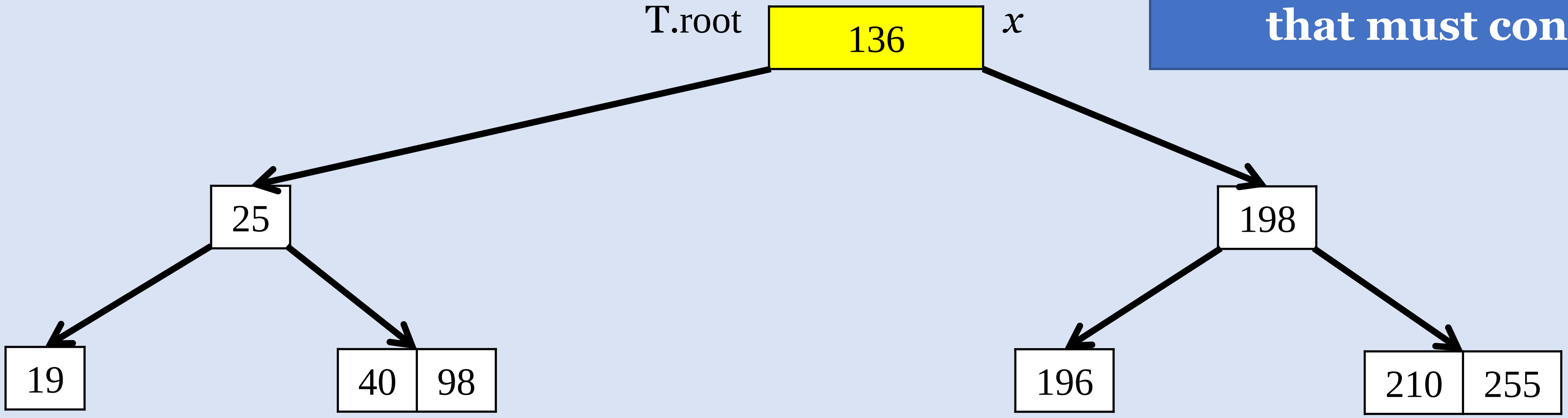


Rule 3b

- If both $x.c_i$'s immediate sibling have *t – 1* keys, merge $x.c_i$ with one sibling, which involves moving a key from *x* down into the new merged node to become the median for that node

Delete 25
t = 2

Key is not in highlighted node… take root of subtree that must contain *k*

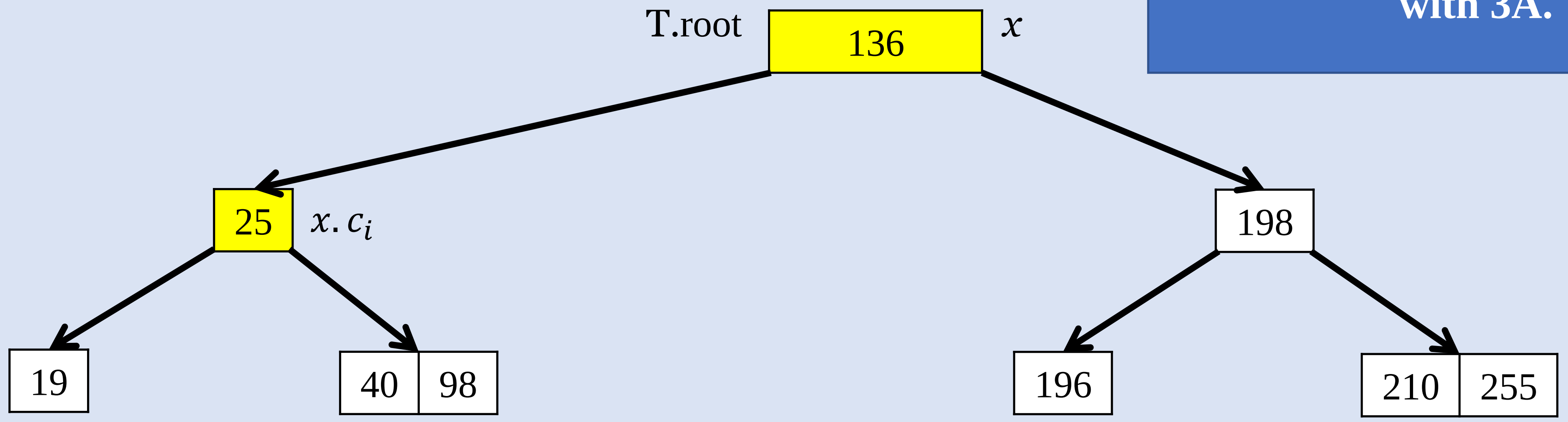


Rule 3b

- If both $x.c_i$'s immediate sibling have *t – 1* keys, merge $x.c_i$ with one sibling, which involves moving a key from *x* down into the new merged node to become the median for that node

Delete 25
t = 2

$x.c_i$ has $t - 1$. Apply Rule 3. Start with 3A.

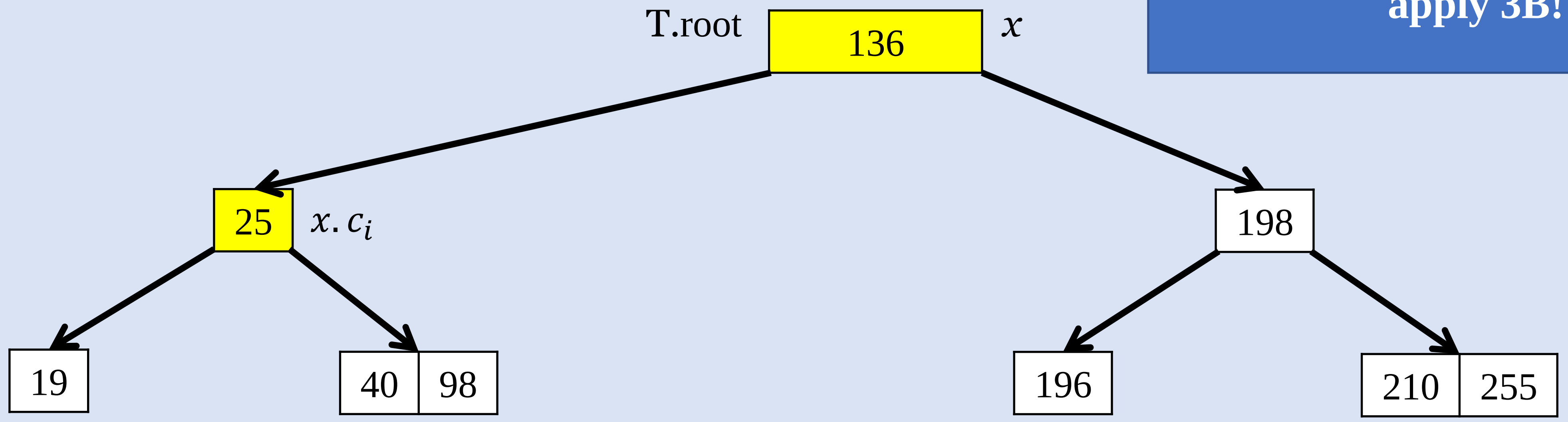


Rule 3b

- If both $x.c_i$'s immediate sibling have *t – 1* keys, merge $x.c_i$ with one sibling, which involves moving a key from *x* down into the new merged node to become the median for that node

3A doesn't work sadly… BUT we apply 3B!

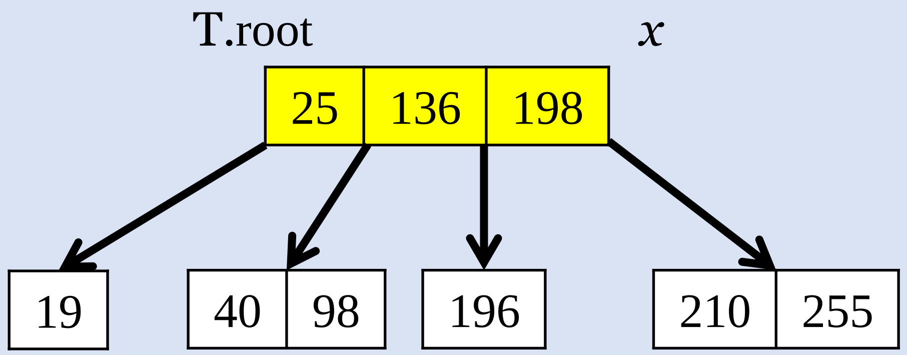
Delete 25
t = 2



Rule 3b

- If both $x.c_i$'s immediate sibling have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node

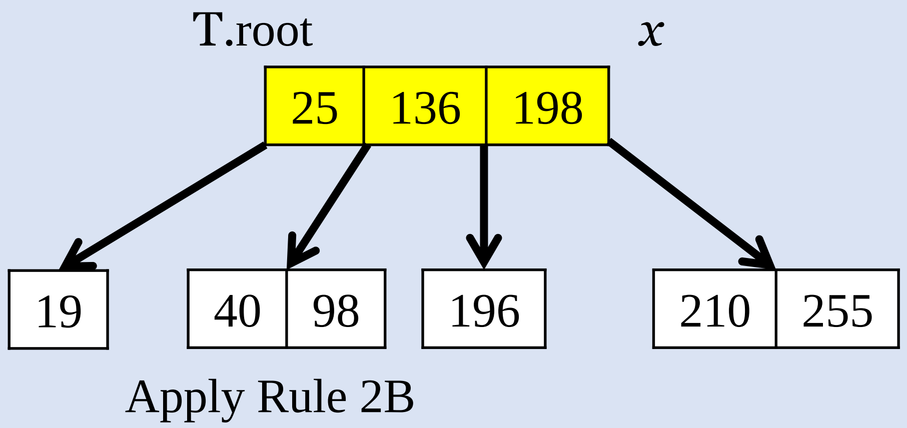
Delete 25
t = 2



Rule 3b

- If both $x.c_i$'s immediate sibling have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node

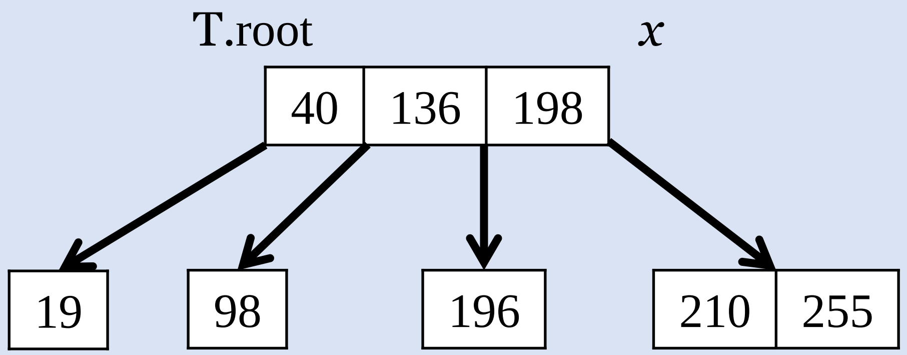
Delete 25
t = 2



Rule 3b

- If both $x.c_i$'s immediate sibling have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node

Delete 25
t = 2



RT for Delete

- RT is $O(th) = O(tlog_t n)$