# Brief Guide to B-trees

What they are

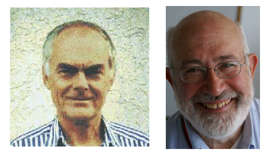
How they are implemented

## B-trees

Self-balancing search trees

Fast search, insertion, deletion

Widely used for databases and file systems

Introduced by Bayer & McCreight (1970) 

What does B stand for? Balanced? Bushy? Boeing?

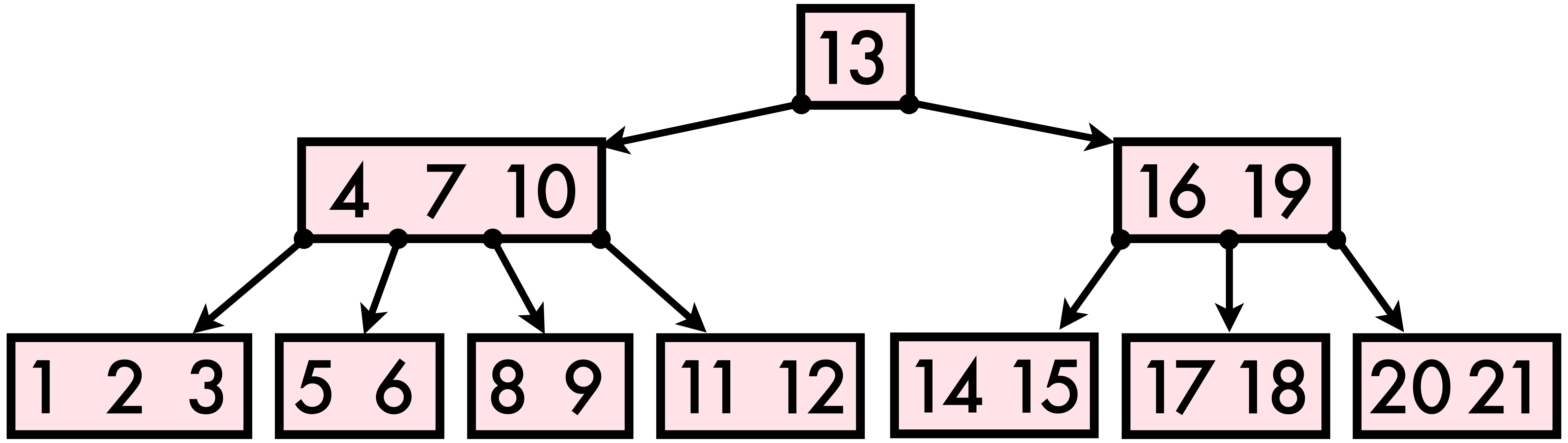
*"The more you think about what the B could mean, the more you learn about B-trees"* (Bayer)

**B-tree complexity (for $n$ data items)**

Worst case: search, insert, delete $\rightarrow O(\log n)$

Storage of B-trees: $\Theta(n)$

**B-trees**: Balanced search trees where nodes can have many children (e.g. thousands)




Higher branching factor $\Longrightarrow$ Reduced tree height

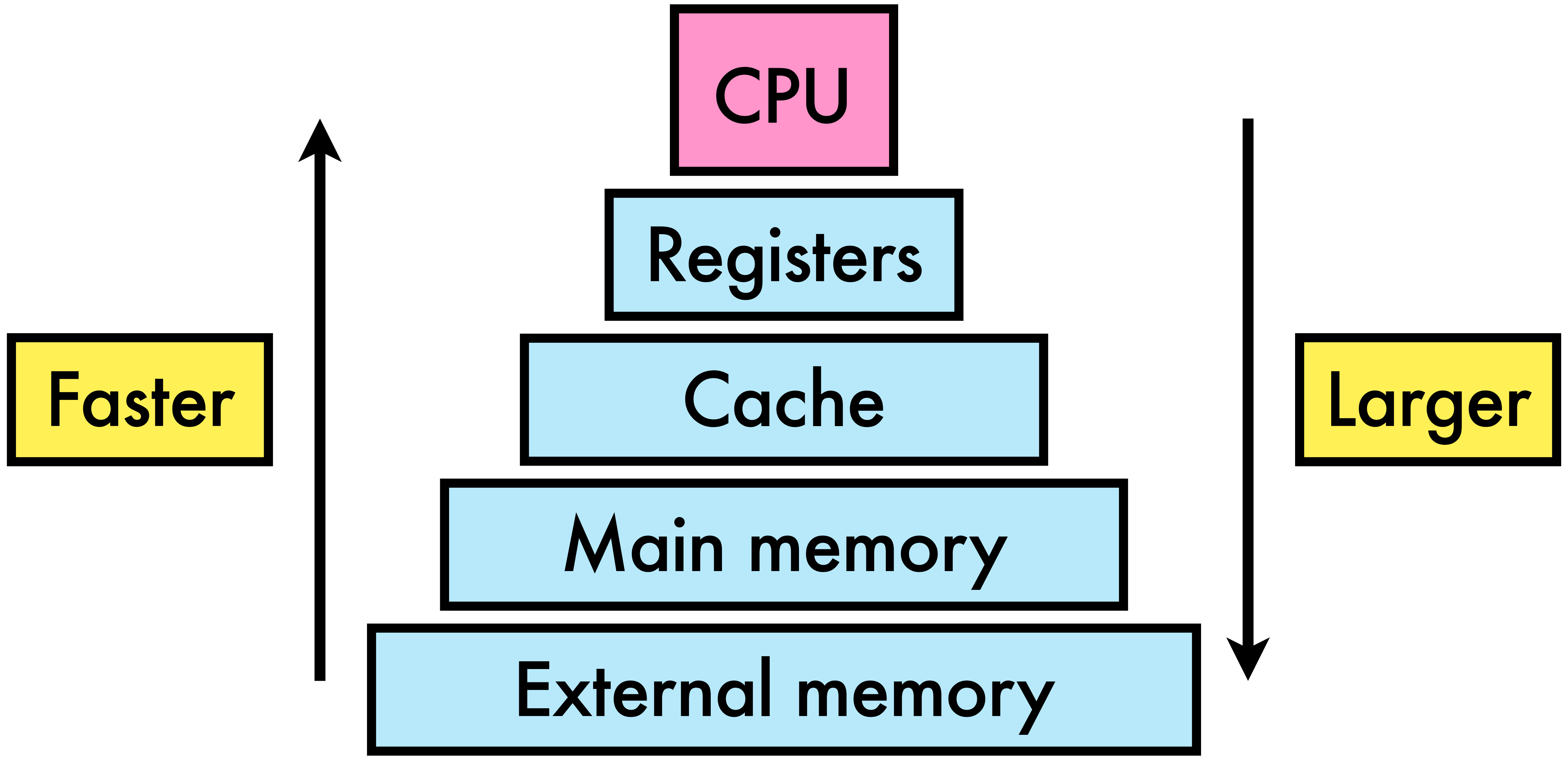
$\Longrightarrow$ Fewer disk accesses

Example applications: MySQL ApFS btrfs

References/Notes/Image credits:
R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)
(R. Bayer) https://www.computerhope.com/people/rudolf_bayer.htm
(E. McCreight photo and discussion of naming) https://www.mccreight.com/people/ed_mcc/index.htm
(B-tree use in MySQL) https://www.vertabelo.com/blog/all-about-indexes-part-2-mysql-index-structure-and-performance/
(B-tree use in ApFS) https://www.ntfs.com/apfs-structure.htm
(B-tree use in btfs) https://en.wikipedia.org/wiki/Btrfs

# Precursor: Memory Hierarchy/External Memory

Computer memory is arranged as a hierarchy




For many problems, we care about **two levels**:

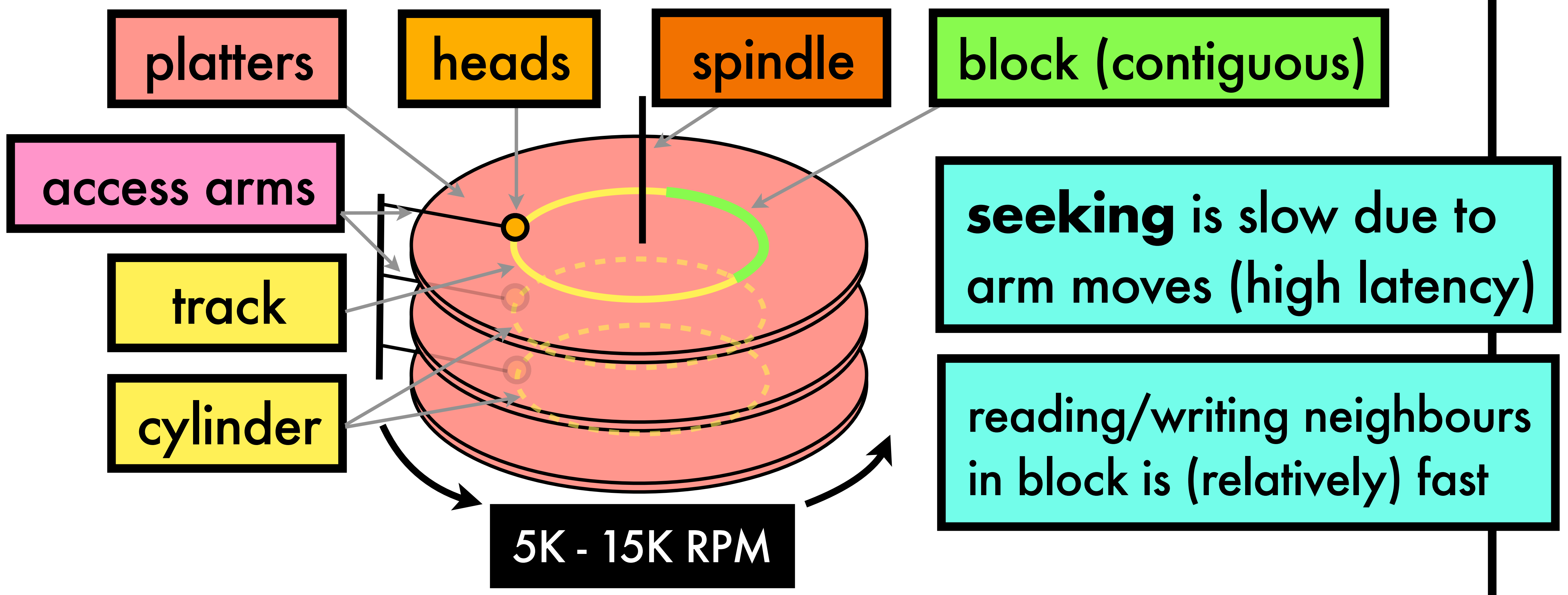
- The level that can store all items in the problem
- The level above this

Transfer is often the bottleneck

B-trees are well-suited to addressing this challenge

Focus on External memory↔Main memory

Hard Disk Drive (introduced in 1956)




**SSDs**: lower latency than HDDs, but still higher than main memory (both SATA SSD & NVMe SSD)

SSDs also use blocks for data access


References:
(Memory hierarchy) M. T. Goodrich et al., "Algorithm design and applications", Chap. 20 (2015)
(Introduction of HDD in 1956) https://www.ibm.com/ibm/history/exhibits/storage/storage_350.html
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 5.4.9 (1998)
T. Cormen et al., "Introduction to algorithms", Chap. 18, MIT press (2022)

# B-trees and Counting Disk Accesses

**A key idea for B-trees:**

Make number of children as large as possible while ensuring each node fits in a single block

Navigating down a shallow-but-wide B-tree then involves very few disk accesses

**Example**: Suppose we have 200 children (199 keys) at each internal node

A (full) B-tree with depth 3 will contain $1 + 200 + 200^2 + 200^3 = 8040201$ nodes

If we keep the root node in memory, we can access $\approx 1.6\text{B}$ keys with just three disk accesses!

**Counting disk accesses**

Reading/writing blocks from disk is expensive, so we track both: CPU time | Disk block read/writes

To access an object u that is not in memory, we must read the block that contains it `read_block(u)`

To store changes to u, we need to write its block to disk `write_block(u)`

References:
M. T. Goodrich et al., "Algorithm design and applications", Chap. 20 (2015)
T. Cormen et al., "Introduction to algorithms", Chap. 18, MIT press (2022)
Note: A similar cost model for counting disk accesses (based on page accesses) is described in detail by R. Sedgewick et al., "Algorithms", 4th Ed. (2011)

# B-tree Definition

## B-tree properties (based on CLRS)

A B-tree is a tree with minimum degree, $t$:

- Node u has attributes:

u.keys | list (ascending order) | u.is_leaf

- Internal node u has len(keys) $+1$ children

u.children | list of length len(keys) + 1

- The keys of node u separate its children's keys

u.keys[i] ≤ u.children[i+1].keys[j] ≤ u.keys[i+1] | ∀ valid j

u.children[0].keys[j] ≤ u.keys[0] | u.keys[-1] ≤ u.children[-1].keys[j]

- All leaves have the same depth
- All nodes (except root) have $\geq t-1$ keys
- All nodes have $\leq 2t-1$ keys

When $t=2$, the B-tree is called a 2-4 tree or 2-3-4 tree

**Warning:** there are many different notation/definition conventions for B-trees!

Bayer & McCreight | Knuth (TAOCP) | CLRS

The height of an $n$-key B-tree grows $\Theta(\log n)$

Num. nodes in a max height ("skinny") tree

$$= \underbrace{1}_{\text{root}} + 2 + 2t + 2t^2 + \ldots = 1 + 2\left(\frac{t^h - 1}{t-1}\right)$$

Num. keys $n = 1 + \cancel{(t-1)} \cdot 2\left(\frac{t^h - 1}{\cancel{t-1}}\right) = 2t^h - 1$

$$\implies h_{\max} = \left\lfloor \log_t\left(\frac{n+1}{2}\right) \right\rfloor$$

Floor $\lfloor \cdot \rfloor$ for other $n$ values

Note: base $t$ in the log makes B-trees short!

If $t$ fixed, use $\Theta(\log n)$ not $\Theta(\log_t n)$ (base change is constant factor)


References:

R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)

D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 6.2.4 (1998)

(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.1, MIT press (2022)

(2-3-4 trees) https://en.wikipedia.org/wiki/2-3-4_tree

M. T. Goodrich et al., "Algorithm design and applications", Chap. 20.2 (2015)

# B-tree Search

Python B-tree search procedure (recursive):

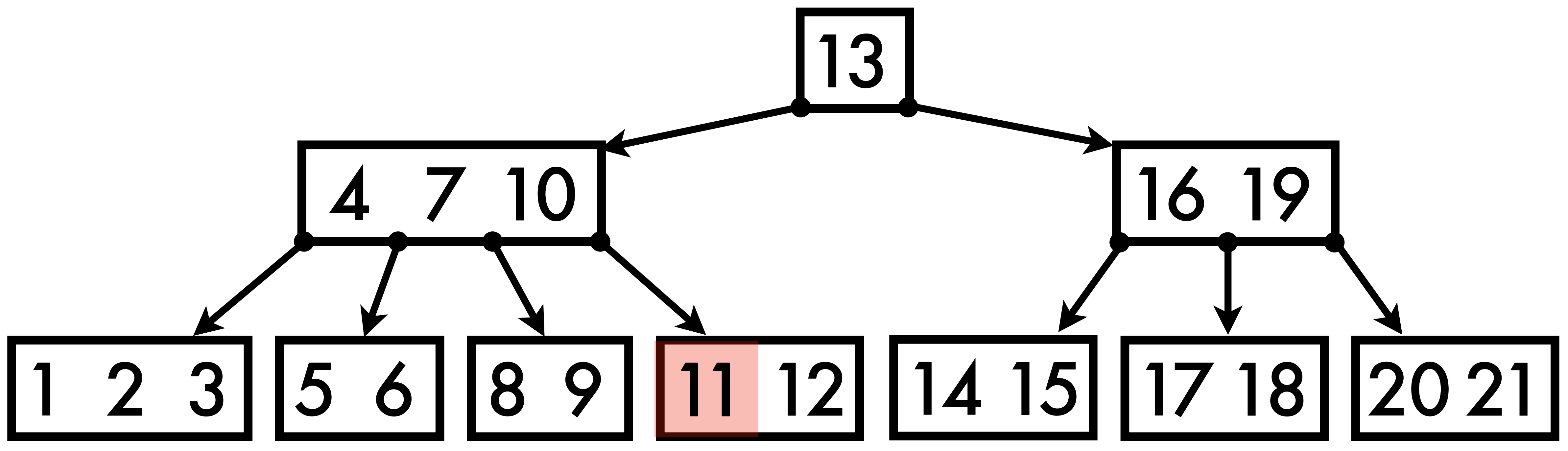
```
def search(self, u, key): # u is a node
    # linear scan to find index of key
    i = 0
    while i < len(u.keys) and key > u.keys[i]:
        i += 1
    if i < len(u.keys) and key == u.keys[i]:
        return (u, i)
    if u.is_leaf:
        return None
    read_block(u.children[i])
    return self.search(u.children[i], key)
```

Arguments: (root, 11)

Returns (node, 0)

Could replace linear scan with binary search

(not always useful due to caching effects)




Example B-tree

**Search complexity**

Consider costs with min. degree, $t$ and num. keys, $n$

We've seen that tree height is $O(\log_t n)$ for $n$ keys

CPU Linear scan $O(t)$ per node, $O(t \log_t n)$ total

(if binary search used, $O(\log t \log_t n)$ total)

Disk block reads $O(\log_t n)$


References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
(Current Linux B-+-tree - uses linear scans) https://github.com/torvalds/linux/blob/7f317d34906c1033f0752fc137dda04e43979bb8/include/linux/btree.h

# B-tree Insertion

## Overview of strategy

Idea: search for leaf node and insert key

What if that node is already full?

Split full node into two nodes at median key:

Keys to left of median key go to the first

Keys to right of median key go to the second

Move median key up into parent

What if the parent is already full...?

Two strategies for B-tree insertion:

1. "**Insert-then-fix**" (Bayer & McCreight)

Insert at leaf, then reverse up tree and fix

2. "**Fix-then-insert**" (CLRS) Split full nodes on the way down, then insert at leaf

Benefit: "1 pass"

```
def insert(self, key): # self is a B-tree
    root = self.root
    if root.is_full(self.t): # has 2t - 1 keys
        root = self.split_root()
    self.insert_not_full(root, key)
```

Note: `split_root()` is the *only way* that B-tree height increases

References:
R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Insertion: split_child()

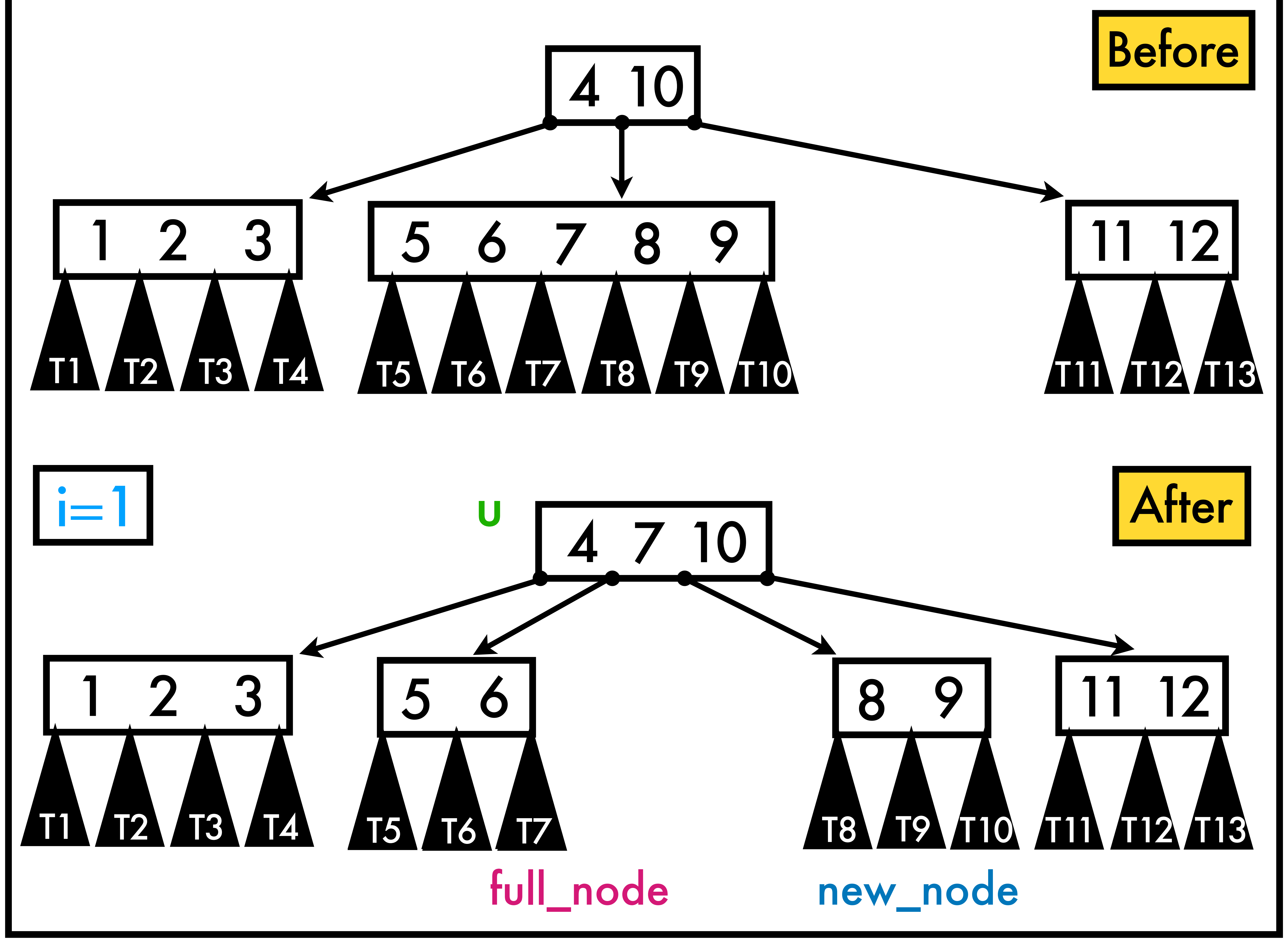
## `split_child()` helper function

Splits the (full) $i^{th}$ child of u (not full):

```
def split_child(self, u, i): # self is a B-tree
    t = self.t  # t is a property of the B-tree
    full_node = u.children[i]
    new_node = Node()
    new_node.is_leaf = full_node.is_leaf
    new_node.keys = full_node.keys[t:]
    if not full_node.is_leaf:
        new_node.children = full_node.children[t:]
    u.children.insert(i+1, new_node)
    u.keys.insert(i, full_node.keys[t-1]) # median
    full_node.keys = full_node.keys[:t-1]
    full_node.children = full_node.children[:t]
    write_block(full_node)
    write_block(new_node)
    write_block(u)
```

Not optimised for efficiency

## Example B-tree (t = 3)





References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

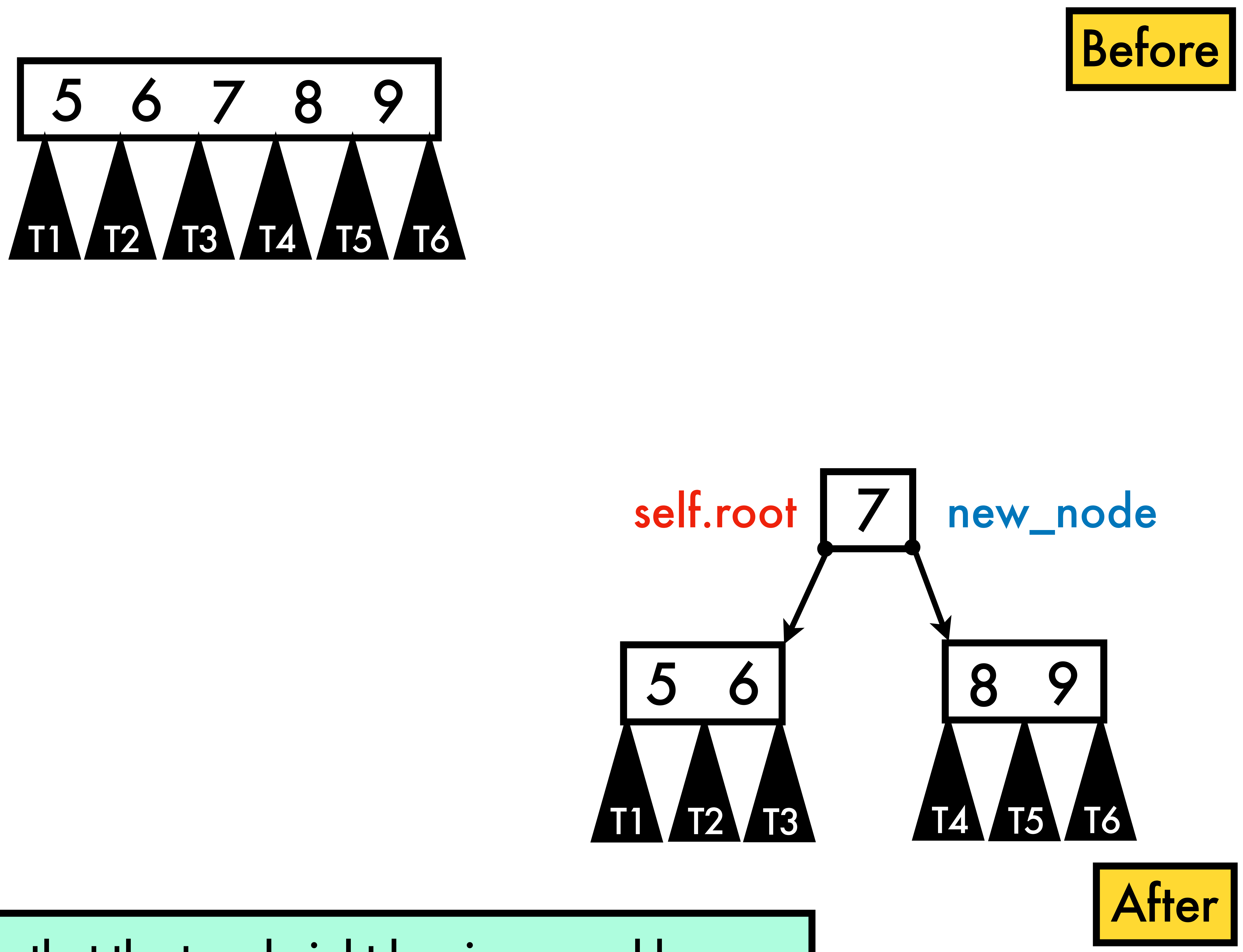
# B-tree Insertion - split_root()

## `split_root()` helper function

Splits the root node when full:

```
def split_root(self): # self is a B-tree
    new_root = Node()
    new_root.is_leaf = False
    new_root.children = [self.root]
    self.root = new_root
    self.split_child(new_root, 0)
    return new_root
```

## Example B-tree (t = 3)




Note that the tree height has increased by one


References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Insertion - insert_not_full()

## `insert_not_full()` helper function

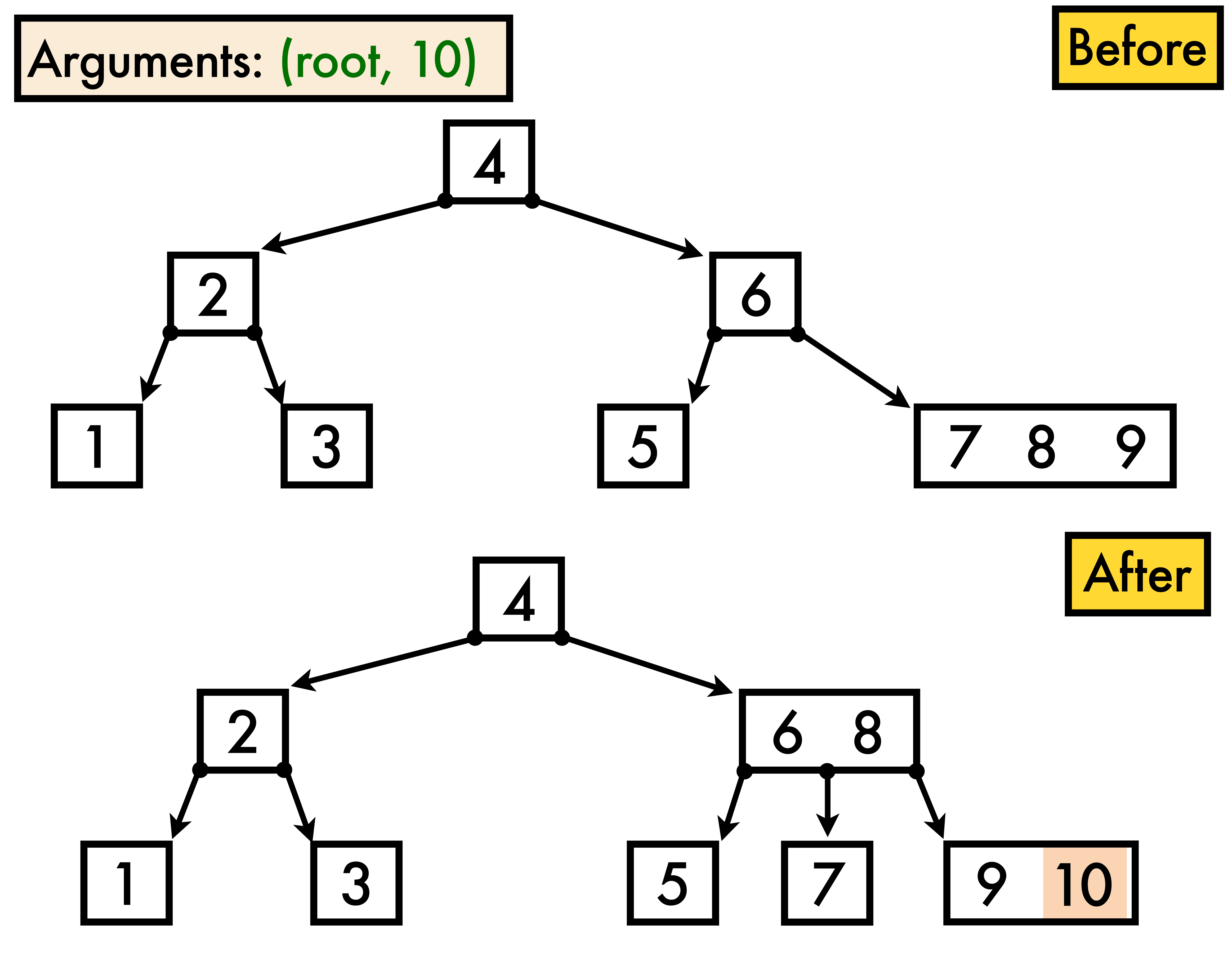
Insert key into node that is not full (recursive):

```
def insert_not_full(self, u, key): # self is a B-tree
    i = 0
    while i < len(u.keys) and key > u.keys[i]:
        i += 1
    if u.is_leaf:
        u.keys.insert(i, key)
        write_block(u)
    else:
        read_block(u.children[i])
        if u.children[i].is_full(self.t):
            self.split_child(u, i)
            i = i if key <= u.keys[i] else i+1
        self.insert_not_full(u.children[i], key)
```

**Tail recursive** — only need $O(1)$ blocks in memory

**Insert complexity** | **CPU** $O(t \log_t n)$ | **Disk** $O(\log_t n)$

## Example B-tree ($t = 2$)





References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Deletion

**Overview of strategy**

Idea: search for node and delete key

What if that node becomes too small?

"Fix-then-delete" - only (recursively) call delete on nodes with $\geq t$ keys (safe to delete 1) **1 pass**

This means we may need to transfer a key down into a child before calling delete

OK since we ensure current node has $t$ keys!

There are 3 cases to handle - when search:

1. Reaches leaf node
2. Reaches internal node containing target key
3. Reaches internal node without target key

```
def delete(self, u, key): # self is a B-tree
    # u has t keys or is the root
    i = 0
    while i < len(u.keys) and key > u.keys[i]:
        i += 1
    # handle cases
    # ...
```

Note: if root ends up with no keys it is deleted

Its only child then becomes the root

This event decreases the B-tree height by one

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Deletion - Case 1

**Deletion search reaches a leaf node**

**Case 1** we have reached a leaf node

```
...
if u.is_leaf:
    if i < len(u.keys) and key == u.key[i]:
        u.keys.pop(i)
        write_block(u)
    else:
        raise KeyError(f"{key} not found")
    return
```

Recall that u will still have $\geq t-1$ keys remaining after deletion (we assume it had $t$ keys before)

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Deletion - Case 2

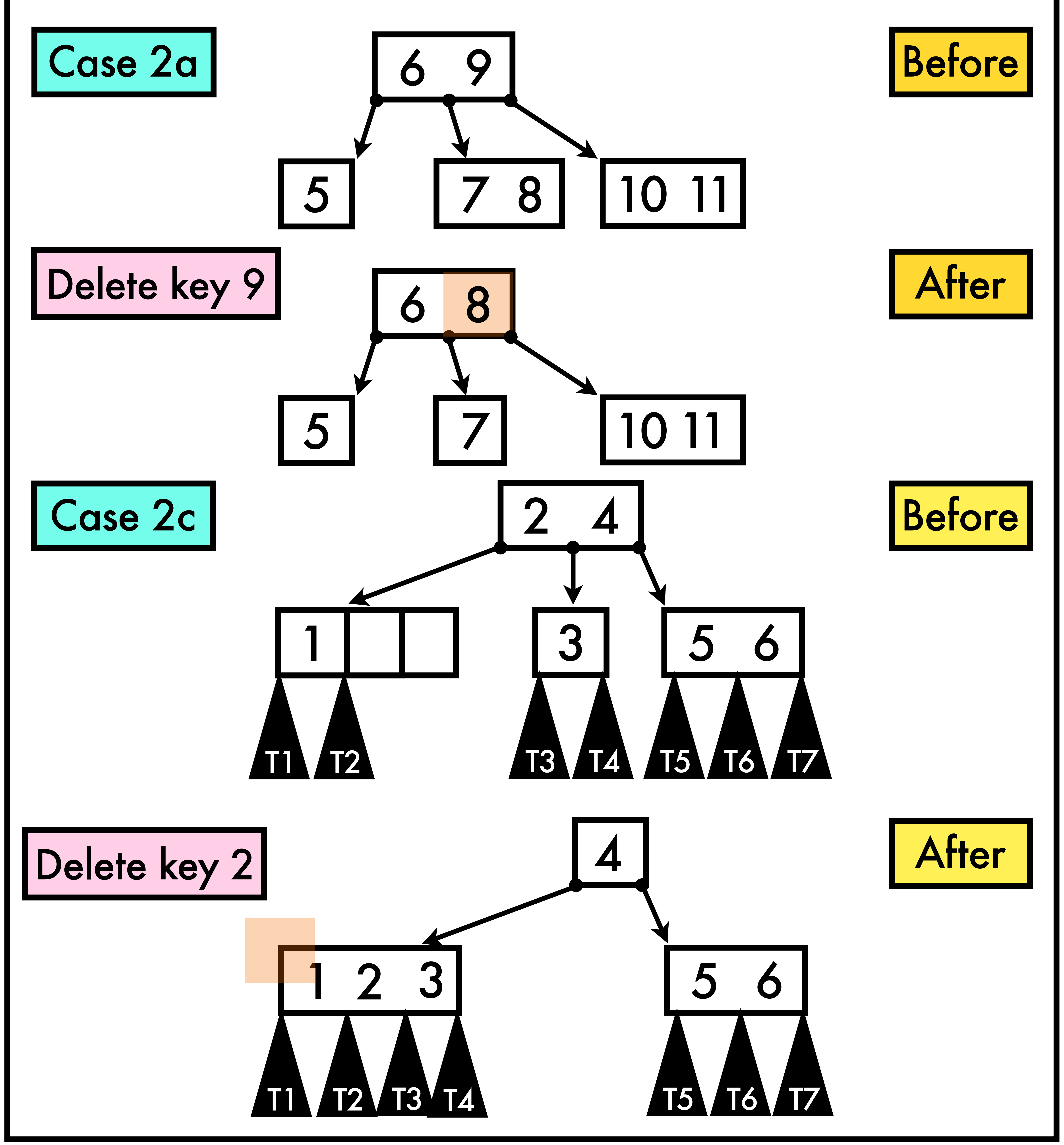
## Deletion search reaches internal node with key

**Case 2** 3 sub-cases (depending on num. keys in u.children[i] and u.children[i+1])

```
... # u not a leaf
if i < len(u.keys) and key == u.key[i]: # case 2
   if len(u.children[i].keys) >= self.t: # case 2a
       pred_key = self.predecessor(key, u.children[i])
       self.delete(u.children[i], pred_key)
       u.keys[i] = pred_key
   elif len(u.children[i+1].keys) >= self.t: # case 2b
       succ_key = self.successor(key, u.children[i+1])
       self.delete(u.children[i+1], succ_key)
       u.keys[i] = succ_key
   else: # case 2c - children i and i+1 both have t - 1 keys
       self.merge_children(u, i)
       if self.root == u and not u.keys:
           self.root = u.children[0] # decrease tree height
       self.delete(u.children[i], key)
```

Note: we omit read_block/write_block calls to reduce code length

## Example B-tree (t = 2)





References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Deletion - Case 3 (3a)

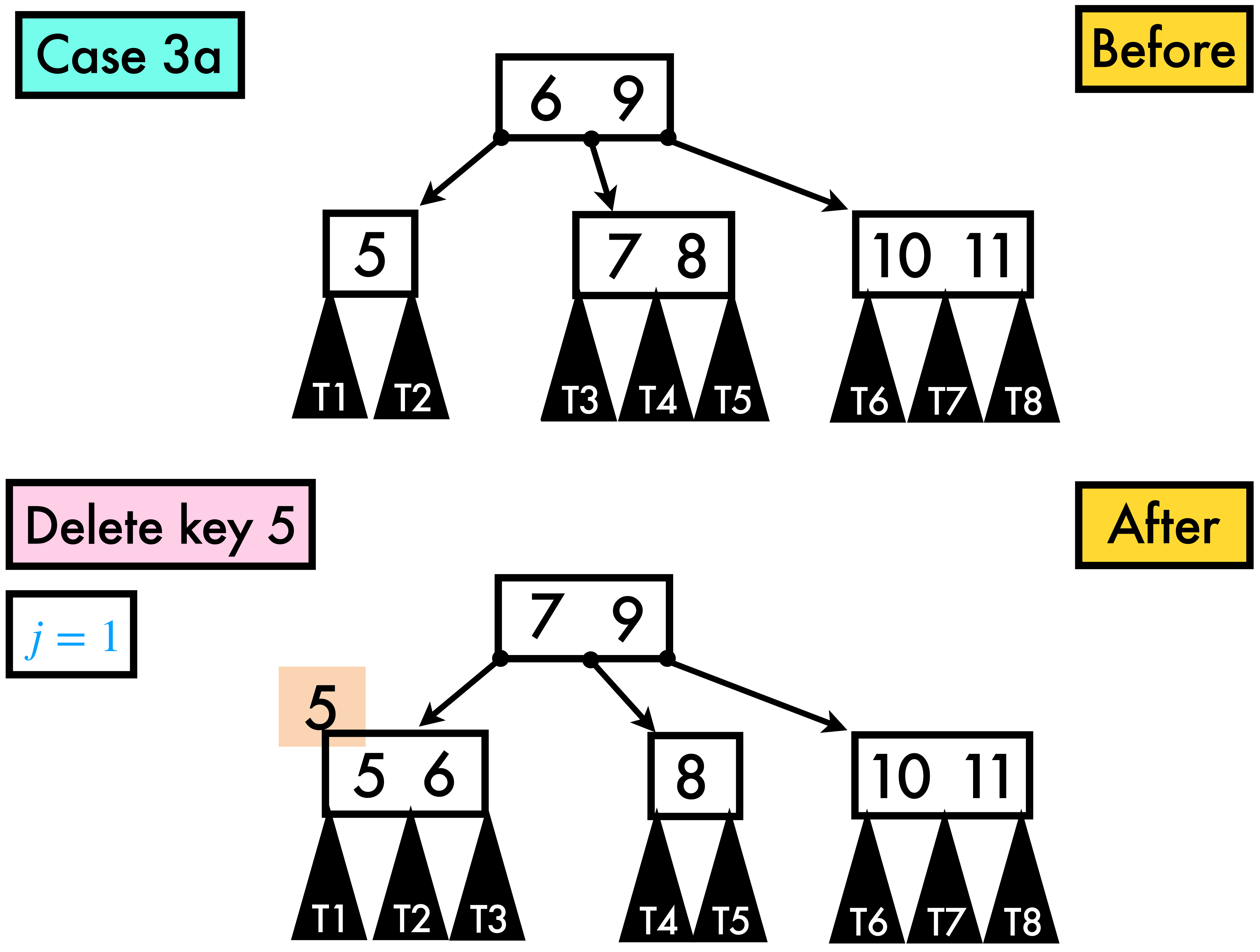
## Search reaches internal node without key

**Case 3** Keep searching and ensure next node visited has $\geq t$ keys

If not initially, case 3a ( $\geq 1$ sibling has $\geq t$ keys) or case 3b (neither sibling does) applies:

```
...
else: # u not a leaf and key not in u.keys
    if len(u.children[i].keys) >= self.t:
        self.delete(u.children[i], key) # continue recursion
    elif self.has_sibling_with_t_keys(u, i): # case 3a
        j = self.index_of_sibling_with_t_keys(u, i)
        if j == i + 1: # sibling with >= t keys to the right
            u.children[i].keys.append(u.keys[i])
            u.keys[i] = u.children[j].keys.pop(0)
            if not u.children[j].is_leaf:
                first_child = u.children[j].children.pop(0)
                u.children[i].children.append(first_child)
        else: # left sibling has at least t keys
            ... # symmetric to case above
    self.delete(u.children[i], key) # continue recursion
```

## Example B-tree ($t = 2$)





References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

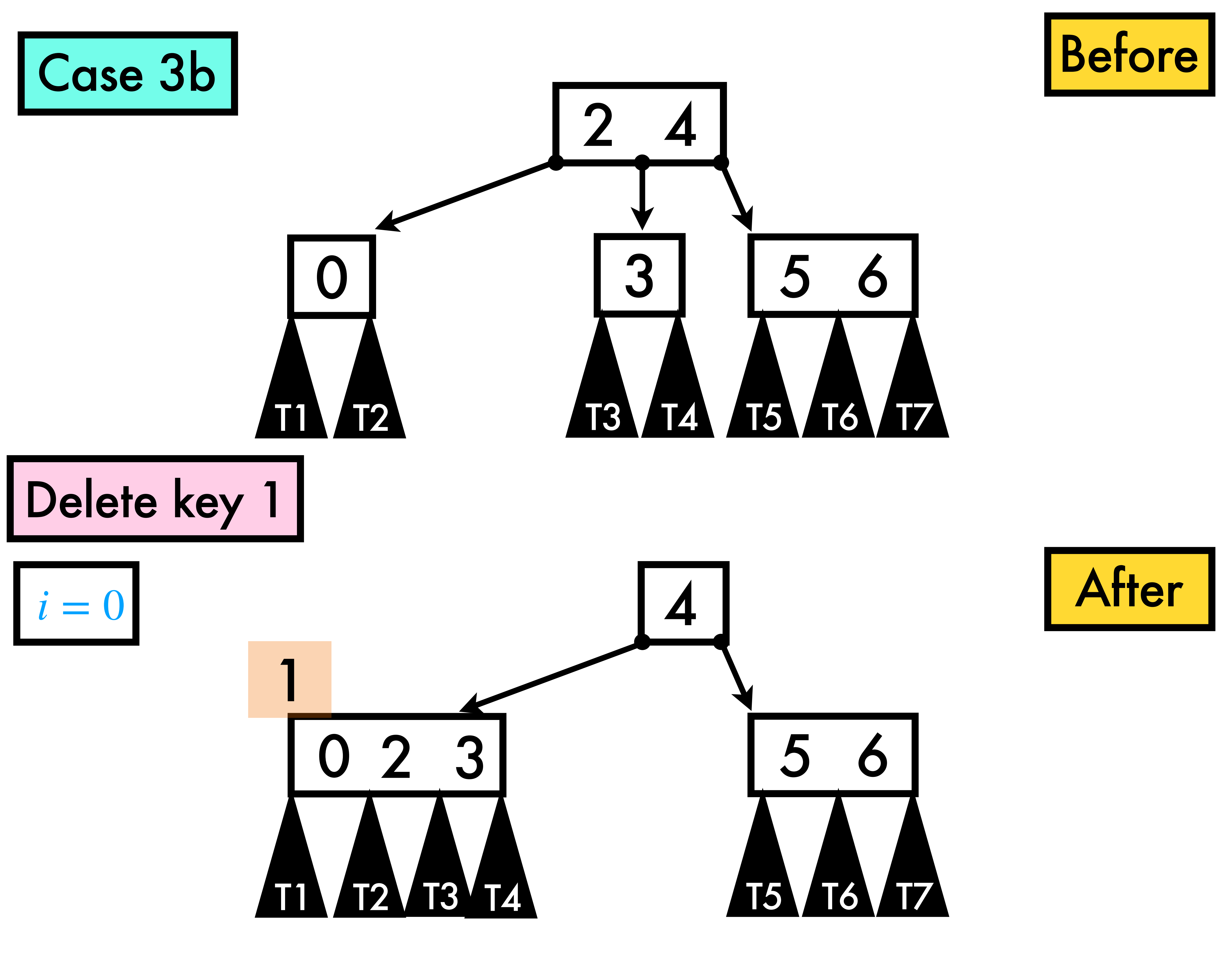
# B-tree Deletion - Case 3 (3b)

## Search reaches internal node without key

**Case 3b** Neither sibling has $\geq t$ keys, so we perform a merge

```
... # u not a leaf and key not in u.keys
else: # case 3a both siblings have t - 1 keys
    if i > 0:  # we merge with left sibling
        self.merge_children(u, i - 1)
        i = i - 1  # we now have one less child to the left
    else:  # we merge with right sibling
        self.merge_children(u, i)
    if self.root == u and not u.keys:
        self.root = u.children[0] # decrease tree height
    self.delete(u.children[i], key)
```

## Example B-tree (t = 2)





References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

# B-tree Deletion - merge_children()

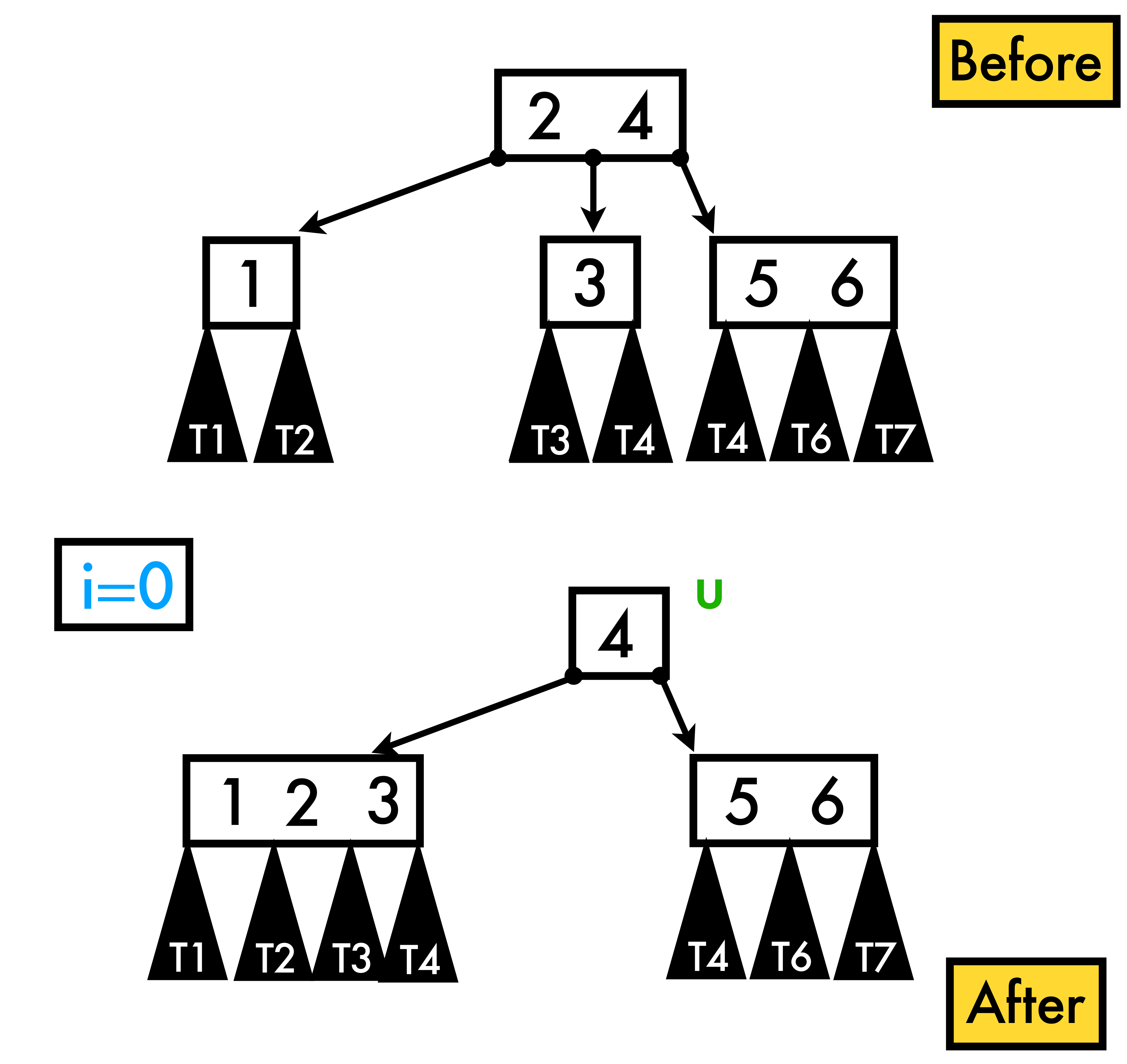
## `merge_children()` helper function

Merge $i^{\text{th}}$ and $(i+1)^{\text{th}}$ children of node u when both children have $t-1$ keys

The merge is performed around a median key that is pushed down from u:

```
def merge_children(self, u, i): # self is a B-tree
    median_key = u.keys.pop(i)
    u.children[i].keys.append(median_key)
    sibling_keys = u.children[i+1].keys
    u.children[i].keys.extend(sibling_keys)
    if not u.children[i].is_leaf:
        sibling_children = u.children[i+1].children
        u.children[i].children.extend(sibling_children)
    u.children.pop(i+1)
```

## Example B-tree (t = 2)



# B-tree Deletion - Complexity

**Deletion complexity (one pass)**

Successor/predecessor calls followed by function termination (still "one pass")

Tree height is $O(\log_t n)$ for $n$ keys:

CPU | Linear scan $O(t)$ per node, $O(t \log_t n)$ total

Disk block reads/writes | $O(\log_t n)$

Note: in practice, most deleted keys are in the leaves (for large values of $t$)

Other B-tree variants we did not discuss:

- B+-tree - all values stored in leaves (not internal nodes) which are linked
- B*-tree - aims to keep non-root nodes "more full" (at least 2/3)

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)
D. Comer, "The Ubiquitous B-tree", ACM Computing Surveys (1979)
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 6.2.4 (1998)