B trees

Motivation for B-trees

- Perfect BSTs are 2 trees.
- Extended to 2-3 trees.
- Extended to 2-3-4 trees.
- Can be extended to 2-3-4-5 trees, 2-3-4-5-6 trees, …
- Can also be extended to 3-4-5 trees:
  - Each non-leaf node has 3 or 4 or 5 children; each non-leaf node has 2 or 3 or 4 keys.
  - All the leaves are at the same level.
- Can be extended to 3-4-5-6 trees, 3-4-5-6-7 trees, …
- In general, we can have m-(m-1)-(m-2)-…-(m+p) trees; call them (m,p)-trees.
- Perfect BSTs are (1,1)-trees, 2-3 trees are (2,1)-trees, 2-3-4 trees are (2,2)-trees, 3-4-5-6-7 trees are (3,4)-trees.

B-trees

- A special case of (m,p) trees where
  - m = p
  - a relaxation on the number of children/keys condition at the root

- Perfect BSTs, 2-3-4 trees, 3-4-5-6 trees are B-trees.

- A B-tree with parameter t is a search tree such that
  - (non-leaf) root node has 2 or 3 or … or 2t children (1 or 2 or … or 2t-1 keys)
  - non-leaf nodes have t or t+1 or … or 2t children (t-1 or t or … or 2t-1 keys)
  - all the leaf nodes are at the same level.

- 2-3-4 trees are parameter-2 B-trees conversely, not all parameter-2 B-trees are 2-3-4 trees.

Implementing B-trees

- Each node x of a parameter-t B-tree has the following fields:
  - $n(x)$: the number of keys of node x
  - $k_1(x) < k_2(x) < \ldots < k_{n(x)}(x)$: the keys at node x (in increasing order)
  - $leaf(x)$: a Boolean variable to indicate whether x is a leaf node or not
  - $C_1(x), C_2(x), \ldots, C_{n(x)}(x)$: pointers to children of x (NIL if x is a leaf)

- If k_i is a key at any node in the subtree $C_i(x)$ then $k_i < k_i(x) < k_{i+1}$; i.e., keys at node x separate the ranges/intervals of keys in its subtrees.

- If n, h are the number of nodes and height of a parameter-t ($t \geq 2$) B tree then

$$h \leq \log_t((n+1)/2), \qquad n \geq 2t^h - 1$$

Searching in B-trees

To search for a key k at subtree rooted at node x in a B-tree:

BTreeSearch(k, x)

1. $i \leftarrow 1$
2. While $(i \leq n(x))$ and $(k > k_i(x))$

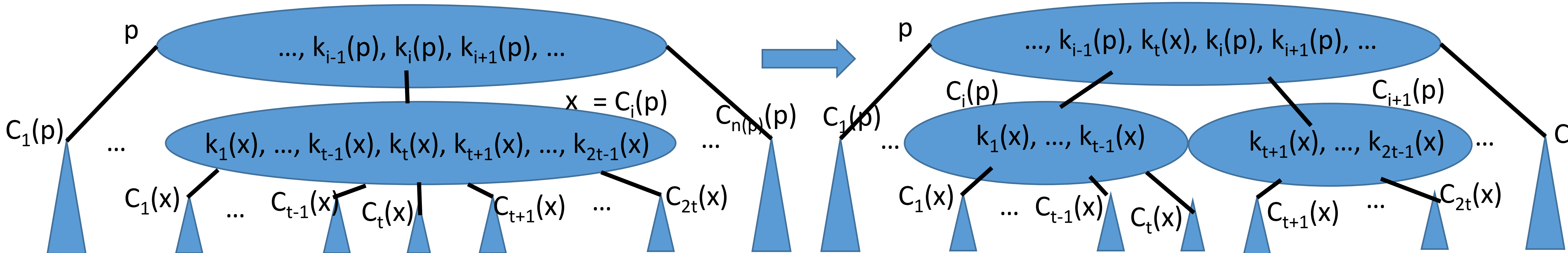
   $i \leftarrow i+1$
3. If $(i \leq n(x))$ and $(k = k_i(x))$ then Return(i, x)
4. If (leaf(x) = True) then Return NIL

   Else Return BTreeSearch(k, $C_i(x)$)

$O(\log n)$

Inserting in B-trees: top-down approach

- There may be a need to split a full node
  - A full node will have 2t-1 (odd number of) keys.
  - The median/middle node is pushed to the parent (which will not be a full node).
  - The first t-1 keys become the keys of the first split child and the last t-1 keys become the keys of the second split child.
- As with 2-3-4 trees, it can be shown that the split in a parameter-t B-tree results in a parameter-t B-tree (may be with an increase in the height by one).
- A pictorial representation of split at a non-root node:



Splitting non-root nodes in B-trees

To split a non-root node x in a parameter-t B-tree with p as Parent[x], x as the i-th child of p; i.e., $x = C_i[p]$, z as the newly added child of p; i.e., z is the $i+1$-th child of p after the split (and is the second half of x before it was split):

$O(t)=O(1)$

BTreeSplit(p, i, x)

1. $z \leftarrow$ NewBTreeNode( )
2. $leaf(z) \leftarrow leaf(x)$
3. $n(z) \leftarrow t-1$

Initializing the fields for new node z

4. for $j=1$ to $t-1$
   $k_j(z) \leftarrow k_{j+t}(x)$

Assigning keys for z

5. if $leaf(x)$=false then
   for $j=1$ to t
   $C_j(z) \leftarrow C_{j+t}(x)$

Assigning children for z

6. $n(x) \leftarrow t-1$

Updating the number of keys at x

7. for $j=n(p)$ down to $i+1$
   $C_{j+1}(p) \leftarrow C_j(p)$

Updating/shifting the right-children of p

8. $C_{i+1}(p) \leftarrow z$

Making z as the i-th child of p

9. for $j=n(p)$ down to $i+1$
   $k_{j+1}(p) \leftarrow k_j(p)$

Updating/shifting the right-keys of p

10. $k_i(p) \leftarrow k_i(x)$

Pushing the key from (old) x to p

11. $n(p) \leftarrow n(p)+1$

Updating the number of keys at p

Insertion into non-full nodes in B-trees

To insert key k into a non-full node x in a parameter-t B-tree

BTreeInsertNonFull(x,k)

1. $i \leftarrow n(x)$
2. If leaf(x) = True then
   - while $(i \geq 1)$ and $(k < k_i(x))$
     - $k_{i+1}(x) \leftarrow k_i(x)$
     - $i \leftarrow i-1$
   - $k_i(x) \leftarrow k$
   - $n(x) \leftarrow n(x)+1$

   Else
   - while $(i \geq 1)$ and $(k < k_i(x))$
     - $i \leftarrow i-1$
   - $i \leftarrow i+1$
   - If $(n(C_i(x))=2t-1)$ then
     - BTreeSplit$(x, i, C_i(x))$
     - If $k > k_i(x)$ then $i \leftarrow i+1$
   - BTreeInsertNonFull$(C_i(x),k)$

Run-time for a single split is $O(t)$, during insertion, there may be up to h splits; so total run-time is $O(t\ h) = (t \log_t n) = O(\log n)$

Insertion into in B-trees

To insert key k into a parameter-t B-tree T (by subsuming the case of splitting at the root)

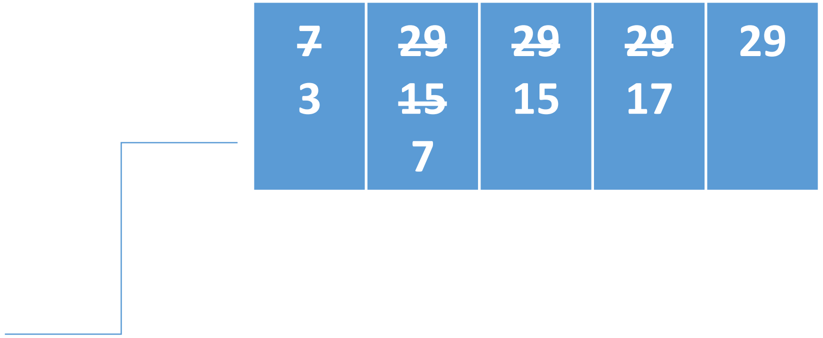
```
BTreeInsert(T, k)
1. r ← root(T)
2. If n(r) = 2t-1 then
       s ← NewBTreeNode( )
       root(T) ← s
       leaf(s) ← FALSE
       n(s) ← 0
       C_1(s) ← r
       BTreeSplit(s, 1, r)
       BTreeInsertNonFull(s,k)
   Else
       BTreeInsertNonFull(r,k)
```

Inserting in a B-tree

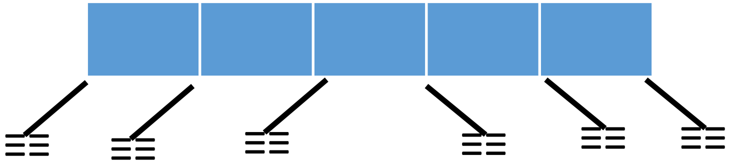
Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}

Note: Since the parameter for the B-tree is 3,

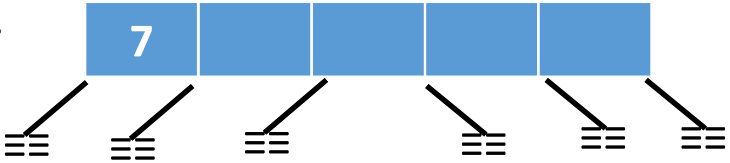
- Each non-root node should have 2 or 3 or 4 or 5 keys
- Root node should have 1 or 2 or 3 or 4 or 5 keys
- A full node will have 5 keys in it.



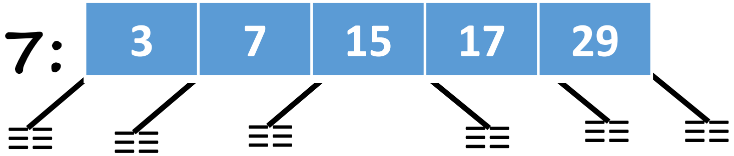
Initial tree (empty tree):



Insert 7:



Insert 29, 15, 3, 17:

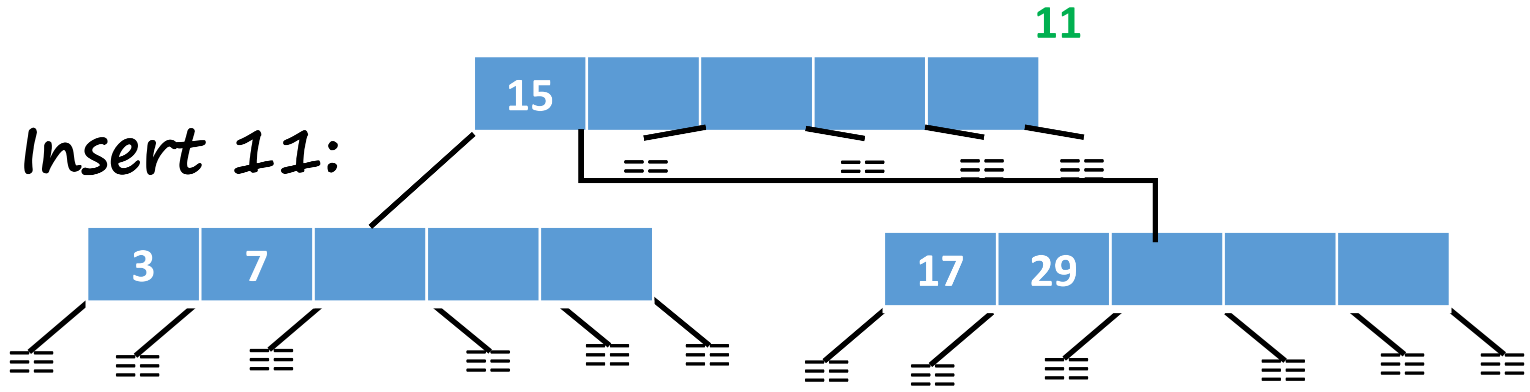


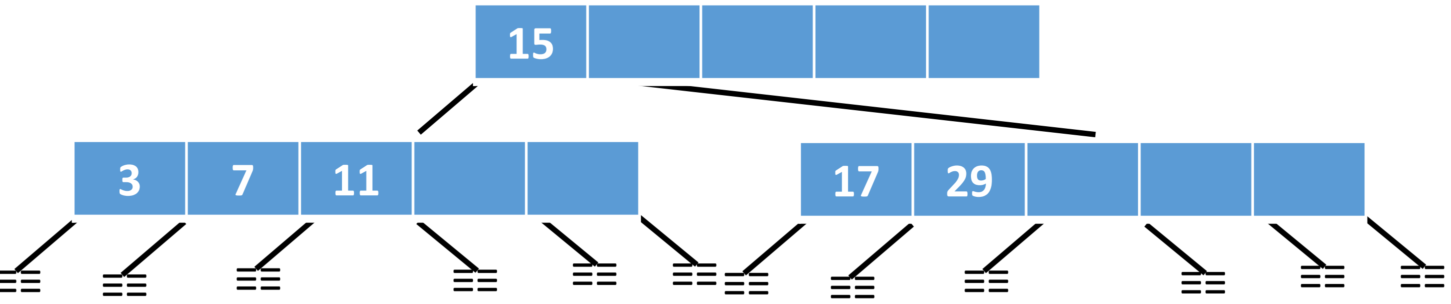
Insert 11: Can't enter the root because it is a full node, so split before continuing with the search. The median at the full node becomes a key of the (new) root node.

Inserting in a B-tree

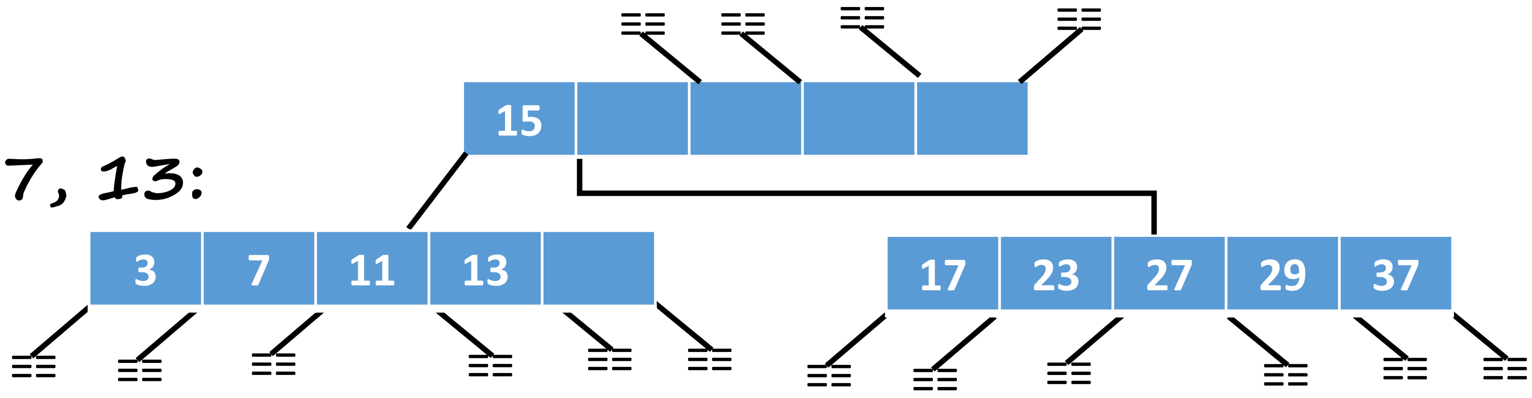
Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}

Insert 11:



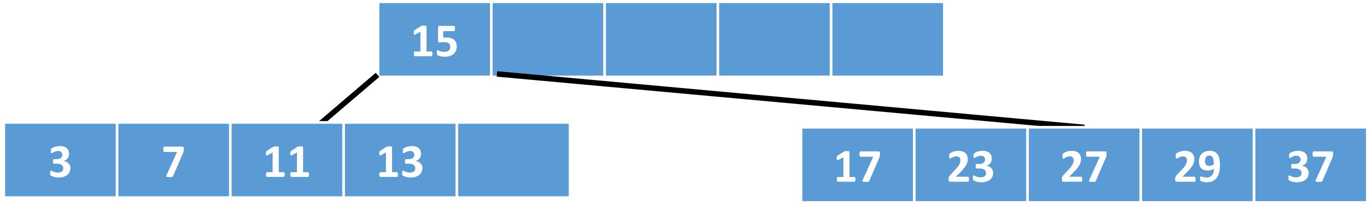


Insert 23, 37, 27, 13:

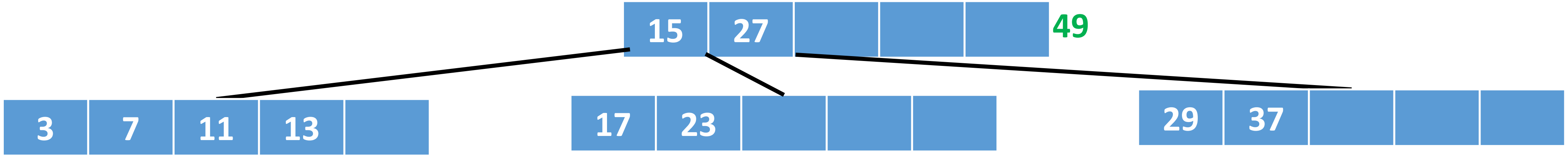


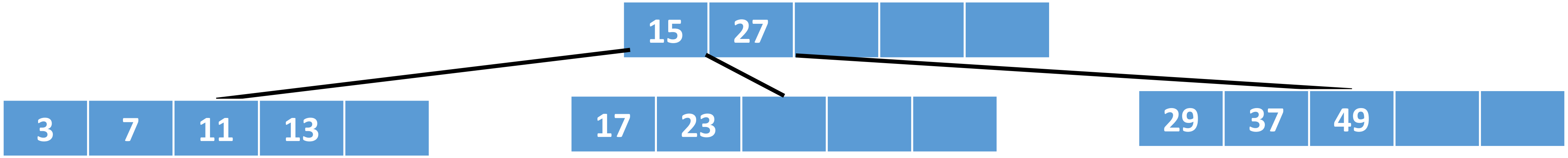
Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}



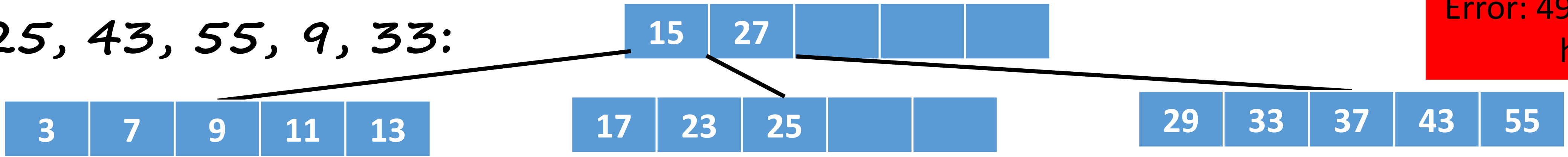
Insert 49: The node [17, 23, 27, 29, 37] is on the search path and is full, so split it before continuing with the search





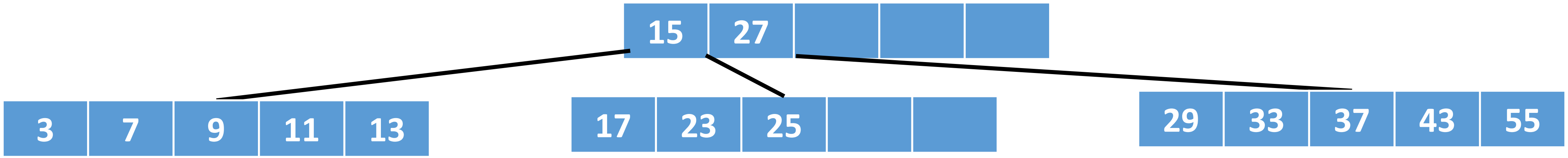
Error: 49 is missing henceforth

Insert 25, 43, 55, 9, 33:

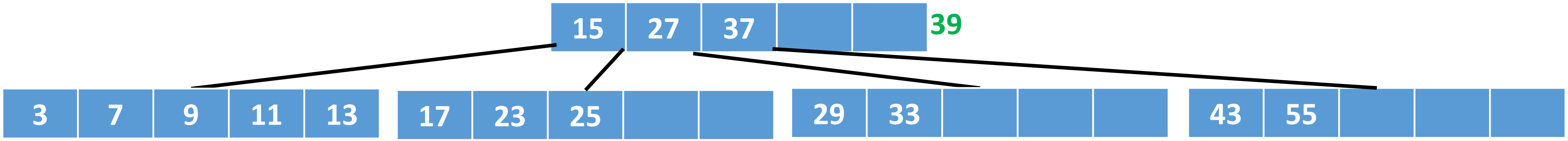


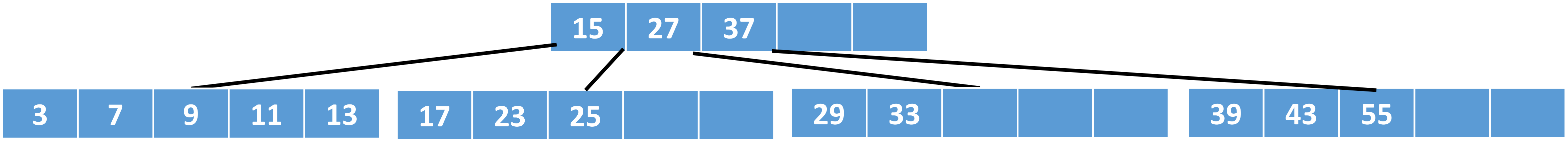
Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}

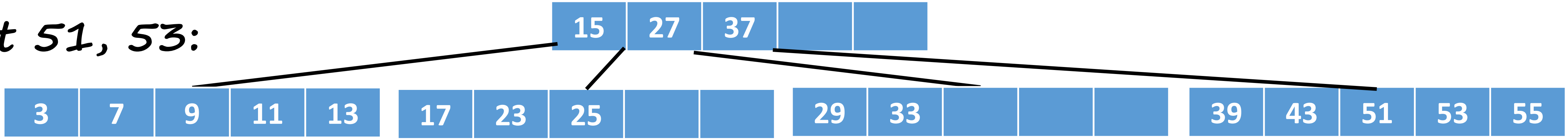


Insert 39: The node [29, 33, 37, 43, 55] is on the search path and is full, so split it before continuing with the search



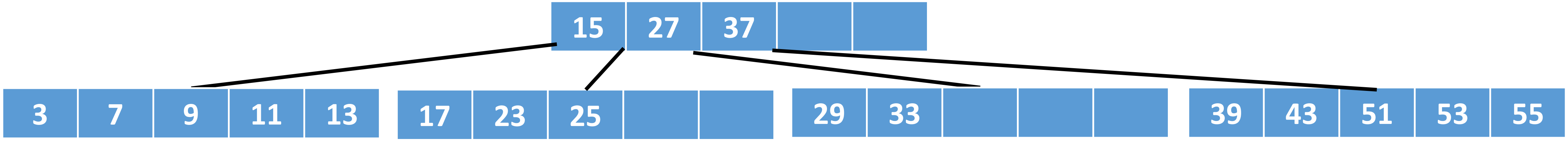


Insert 51, 53:

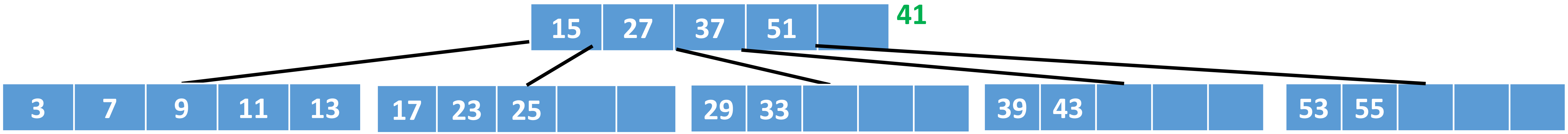


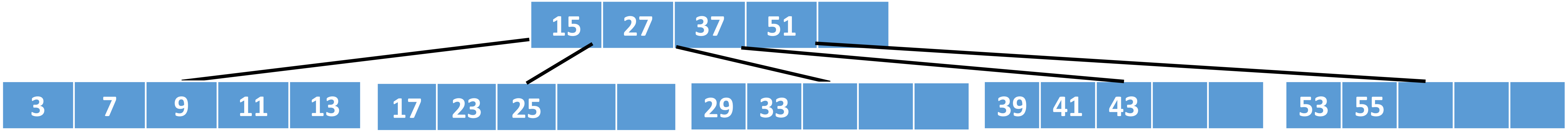
Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}



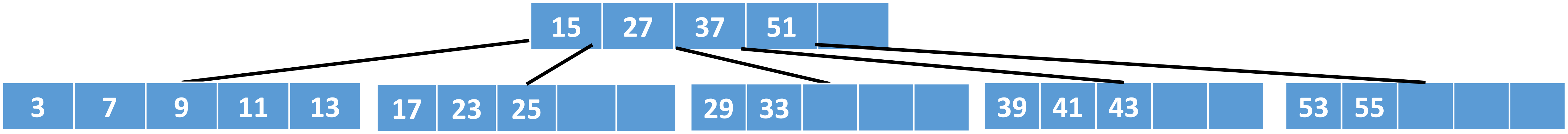
Insert 41: The node [39, 43, 51, 53, 55] is on the search path and is full, so split it before continuing with the search



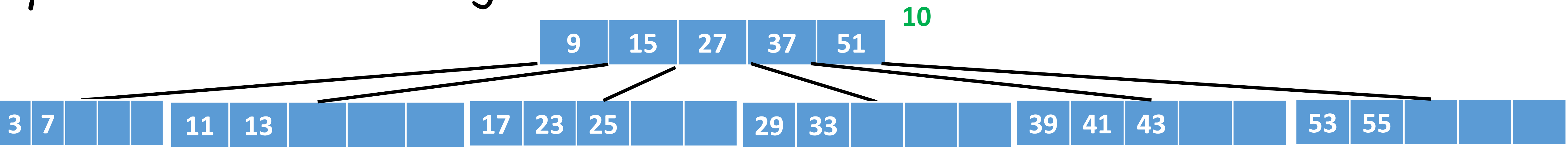


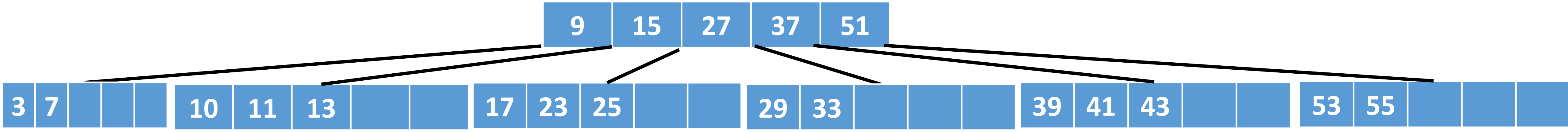
Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}



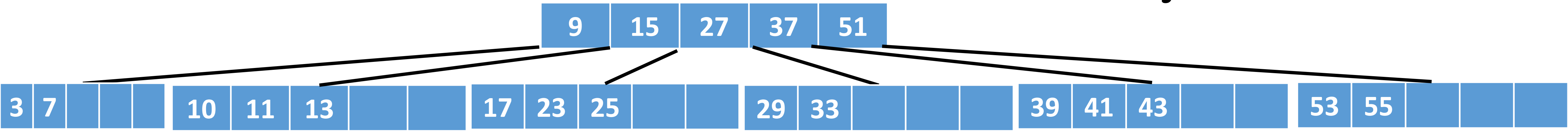
Insert 10: The node [3, 7, 9, 11, 13] is on the search path and is full, so split it before continuing with the search



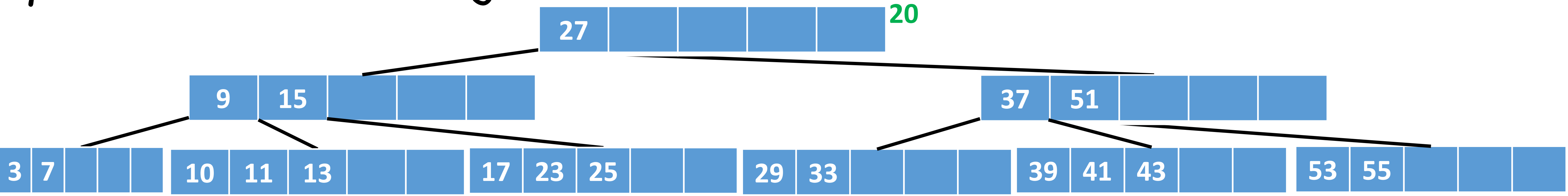


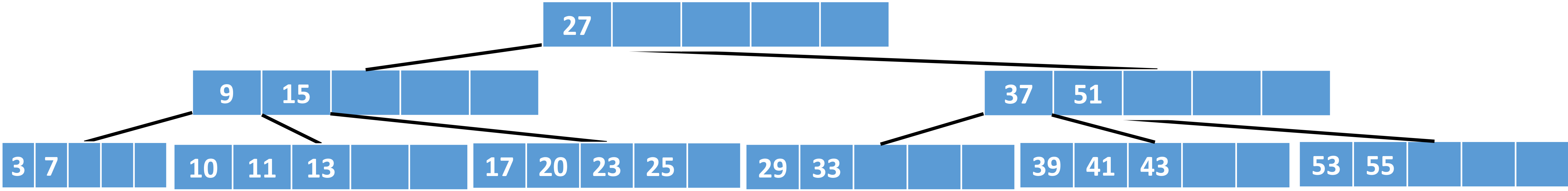
Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 49, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20}



Insert 20: The node [9, 15, 27, 37, 51] is on the search path and is full, so split it before continuing with the search





Inserting in a B-tree

Construct a parameter-3 B-tree for the set {7, 29, 15, 3, 17, 11, 23, 37, 27, 13, 25, 43, 55, 9, 33, 39, 51, 53, 41, 10, 20} with the insertion order being

1. <3, 7, 9, 10, 11, 13, 15, 17, 20, 23, 25, 27, 29, 33, 37, 39, 41, 43, 51, 53, 55>
2. <55, 53, 51, 43, 41, 39, 37, 33, 29, 27, 25, 23, 20, 17, 15, 13, 11, 10, 9, 7, 3>
3. <3, 7, 10, 11, 13, 17, 20, 23, 25, 29, 33, 39, 41, 43, 53, 55, 9, 15, 37, 51, 27>
4. <27, 9, 15, 37, 51, 3, 7, 10, 11, 13, 17, 20, 23, 25, 29, 33, 39, 41, 43, 53, 55>

How structurally similar are these trees?

Deleting from a B-tree

Let T be a parameter-t B-tree. To delete key k from T.

Top-down approach

- Search (recursively) for the location of the key by starting the search at the root of T.
- While searching, never enter a minimal node; i.e., a node with exactly t-1 keys in it (similar to not entering a full node during insertion).
  - "redistribute" before continuing the search at the minimal node (similar to splitting during insertion).
- Delete k from its location by distinguishing the following cases:
  - k is in a (non-minimal) leaf node.
  - k is in a non-leaf node which further distinguishes into cases:
    - root of the predecessor subtree is not minimal
    - root of the successor subtree is not minimal
    - root of both these subtrees are minimal

Redistribution during deletion in a B-tree

Let T be a parameter-t B-tree. To delete key k from T.

Assume that

- the search path has reached node x
- the child $C_i(x)$ is the next node on the search path but is minimal; i.e., $n(C_i(x)) = t-1$.

Case 1: (Immediate) Left or right sibling of $C_i(x)$ is non-minimal; i.e., $n(C_{i-1}(x)) > t-1$ or $n(C_{i+1}(x)) > t-1$.

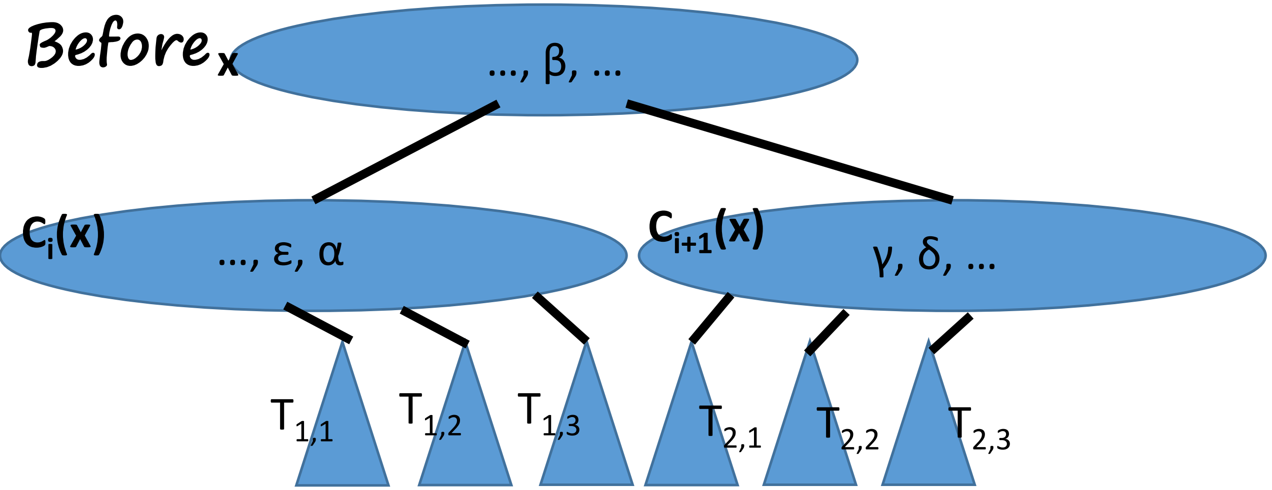
Case 2: Both the (immediate) left and right siblings of $C_i(x)$ are minimal; i.e., $n(C_{i-1}(x)) = t-1$ and $n(C_{i+1}(x)) = t-1$.

Redistribution when a sibling in non-minimal

Case 1: (Immediate) Left or right sibling of $C_i(x)$ is non-minimal, w.l.o.g, let the right sibling; i.e., $C_{i+1}(x)$ be non-minimal

- Make $C_i(x)$ non-minimal by
  - moving a key from x and
  - moving a key from non-minimal sibling to x and
  - moving a child of the non-minimal sibling as a child of $C_i(x)$

Before



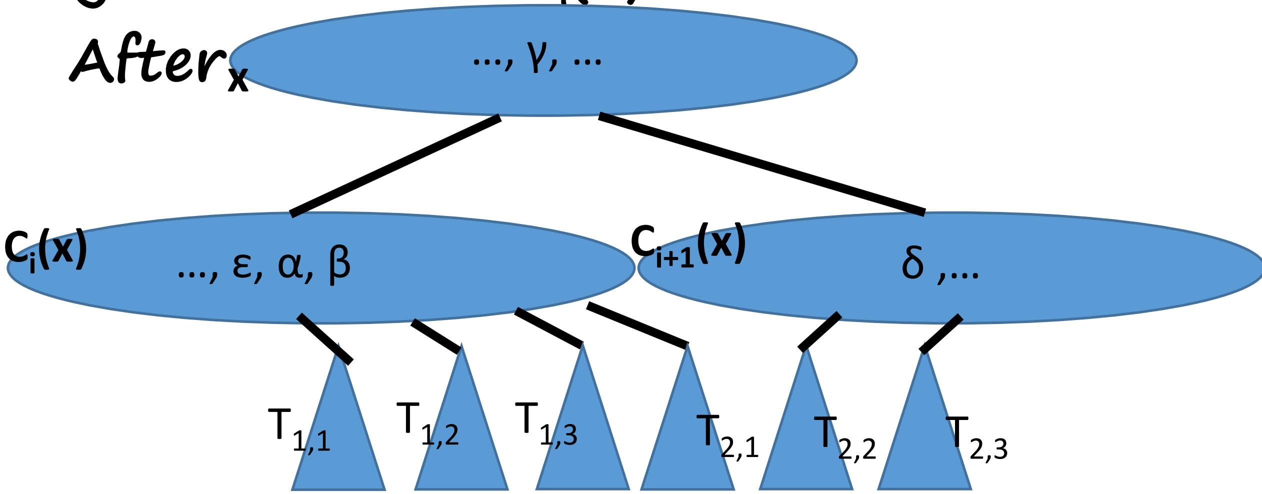
$n(x) \geq t-1$

$n(C_i(x)) = t-1$

$n(C_{i+1}(x)) \geq t$

$T_{1,1}<\varepsilon<T_{1,2}<\alpha<T_{1,3}<\beta$ & $\beta<T_{2,1}<\gamma<T_{2,2}<\delta<T_{2,3}$

After



$n(x) \geq t-1+1-1;$ lost β, got γ

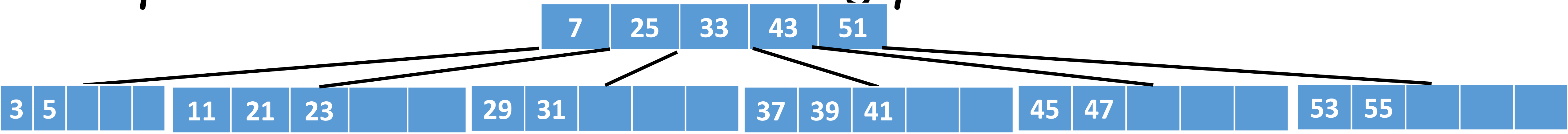
$n(C_i(x)) = t-1+1;$ got β

$n(C_{i+1}(x)) \geq t-1;$ lost γ

$T_{1,1}<\varepsilon<T_{1,2}<\alpha<T_{1,3}<\beta<T_{2,1}<\gamma$ & $\gamma<T_{2,2}<\delta<T_{2,3}$

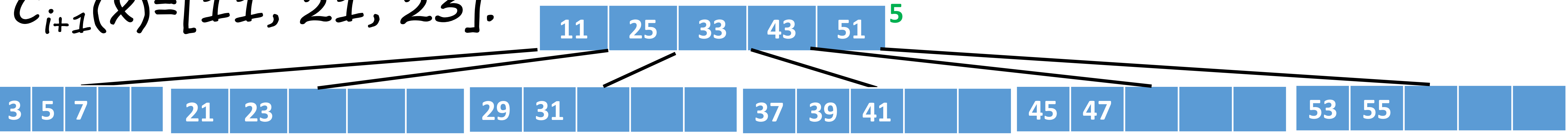
Redistribution when a sibling in non-minimal

Example: Delete 5 from the following parameter-3 B-tree.



Here, k=5

The node to be entered on the search path is [3, 5] which is minimal. It has a non-minimal (right) sibling. So, we do the redistribution with x=[7, 25, 33, 43, 51], $C_i(x)$=[3, 5], $C_{i+1}(x)$=[11, 21, 23].

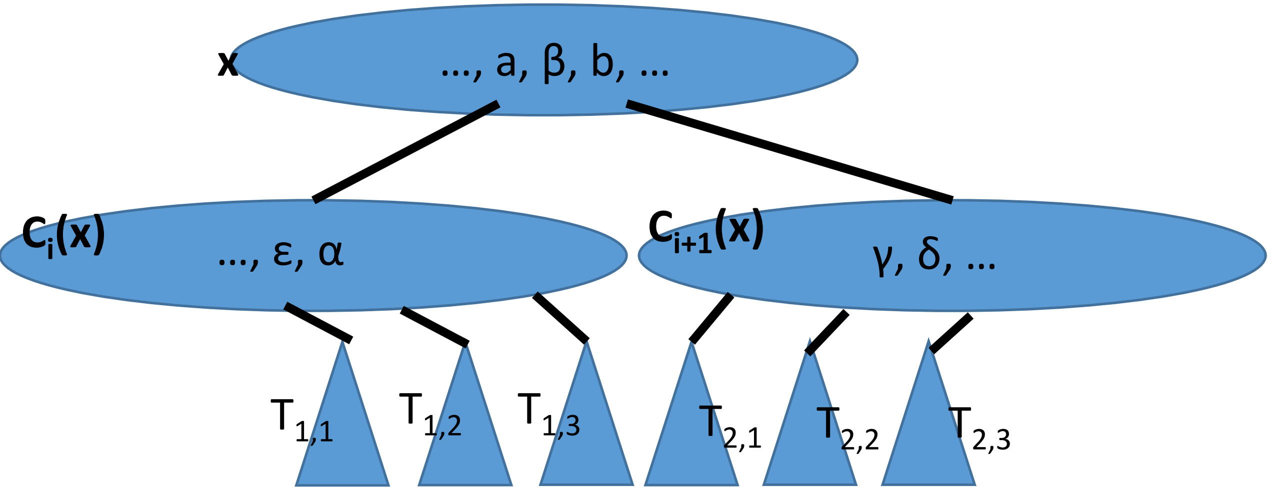


Redistribution when both the siblings are minimal

Case 2: Both the (Immediate) left and right sibling of $C_i(x)$ are minimal,

- Make $C_i(x)$ non-minimal by
  - merging $C_i(x)$ with one of its siblings, say its right sibling $C_{i+1}(x)$
  - move a key from x to the (new) merged node such that this key becomes the median of the merged node.

Before

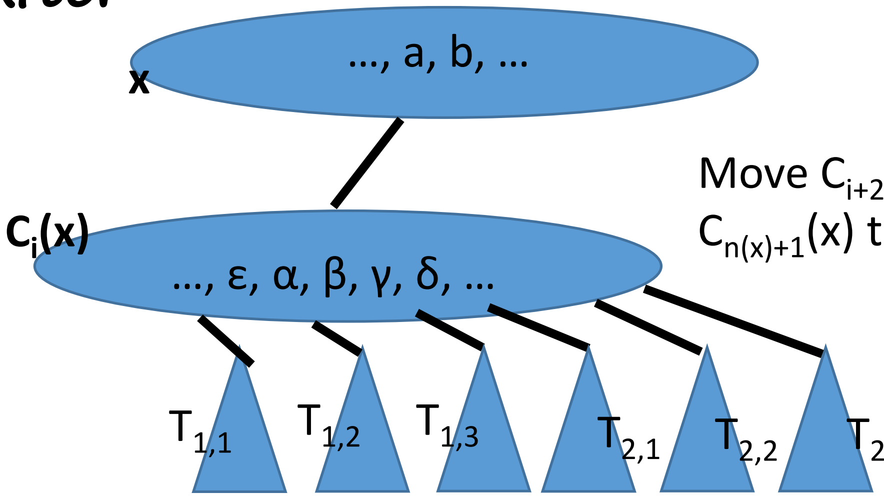


$n(x) \geq t$

$n(C_i(x)) = t-1$

$n(C_{i+1}(x)) = t-1$

After



$n(x) \geq t-1$; lost β

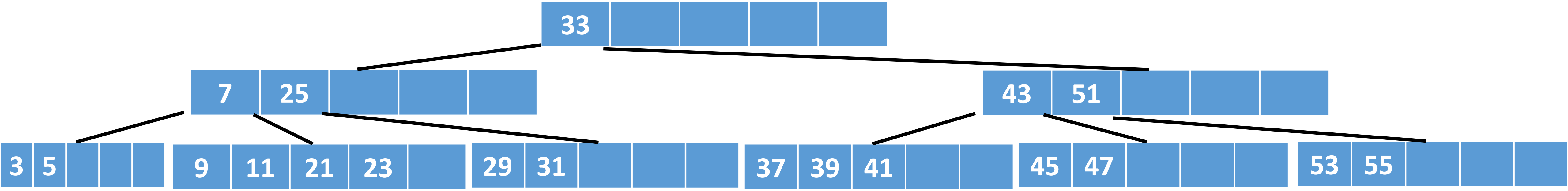
$n(C_i(x)) = t-1+1+t-1 = 2t-1$;

got β and all the keys from $C_{i+1}(x)$

$C_{i+1}(x)$ DNE; lost all its keys

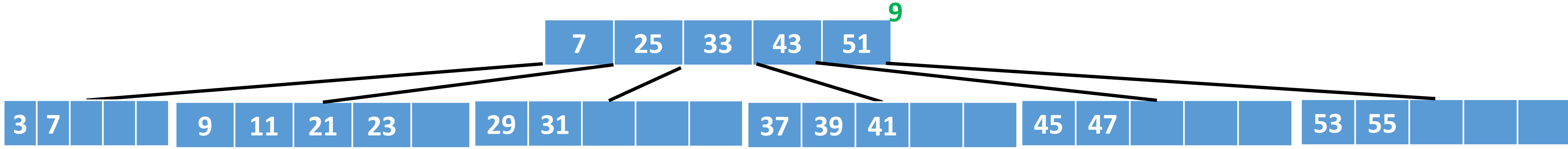
Redistribution when both the siblings are minimal

Example: Delete 9 from the following parameter-3 B-tree.



Here, k=9

The node to be entered on the search path is [33] which is minimal. It is a root node. We do the redistribution with x=[33], $C_i(x)$=[7, 25], $C_{i+1}(x)$=[43, 51].



Deleting from a B-tree

Let T be a parameter-t B-tree. To delete key k from T.

- Let N be the node where k is located,
- having ensured that the search path never entered a minimal node,
- though after redistribution some nodes on the search path (after passing through them) may have become minimal.
- Cases to be considered:
  - Case 1: N is a leaf node
  - Case 2: N is a non-leaf node
    - Case 2.1: Root of the predecessor (w.r.t. k) subtree of N is non-minimal
    - Case 2.2: Root of the predecessor subtree of N is minimal while the successor subtree is non-minimal
    - Case 2.3: Both the predecessor and successor subtrees of N are minimal

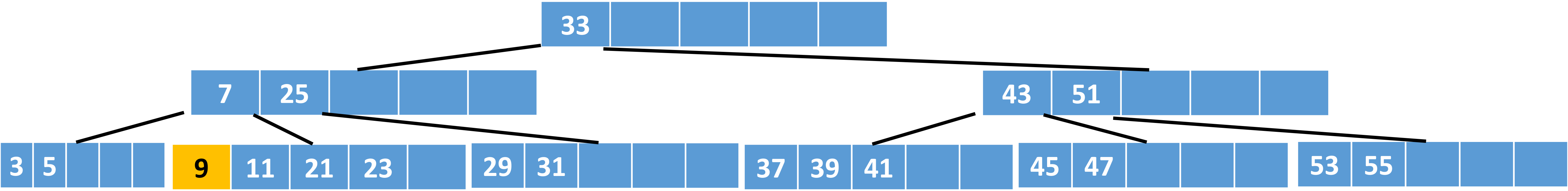
Deleting from a B-tree

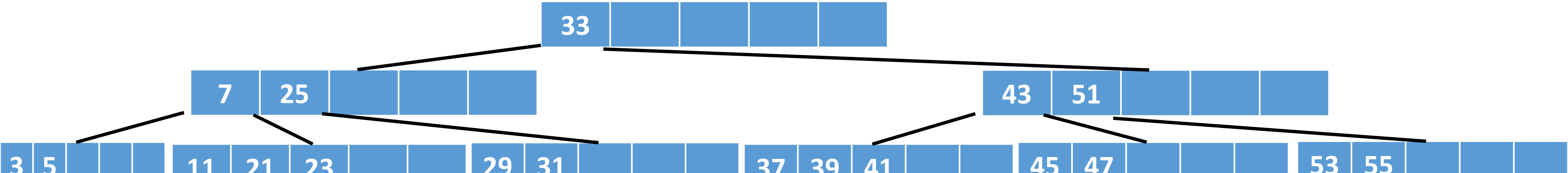
Case 1: N is a leaf node

Note: N is non-minimal; otherwise the search path would not have entered N. So,

- remove the key k from N and
- reduce n(N) by one

Example: To delete 13 from the following B(3)-tree (assuming that redistribution has been done)





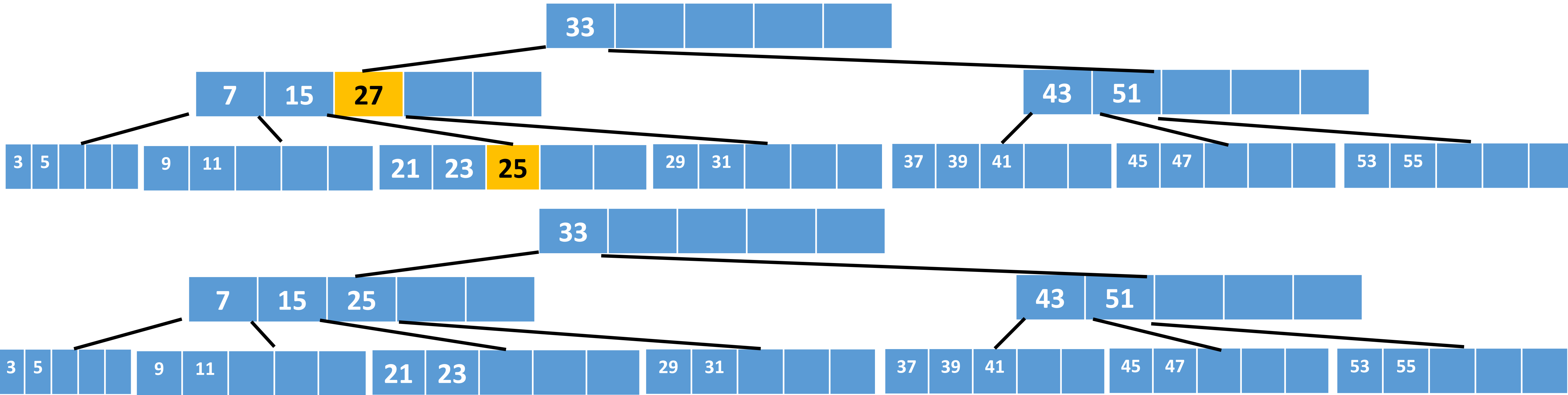
Deleting from a B-tree

Case 2.1: N is a non-leaf node, the root M of the predecessor subtree is non-minimal.

- traverse along the subtree rooted at M to locate the replacement key k' = predecessor of k; let P be the node containing k' [How to identify the predecessor key?]
- delete (recursively) the replacement key, i.e.,
  - at N, replace k with k' and at P, delete k'

Example: To delete 27 from the following B(3)-tree (assuming redistribution)

Here, k=27, N=[7, 15, 27], M=[21, 23, 25], k'=25, P=M=[21, 23, 25]



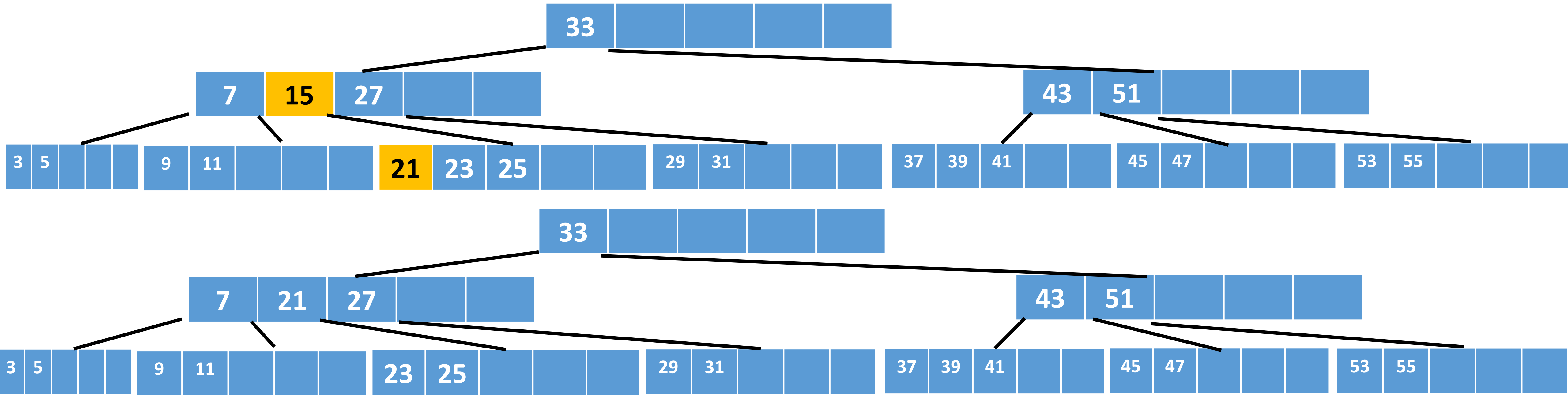
Deleting from a B-tree

Case 2.2: N is a non-leaf node, the root O of the predecessor subtree is non-minimal.

- traverse along the subtree rooted at O to locate the replacement key k' = successor of k; let P be the node containing k' [How to identify the successor key?]
- delete (recursively) the replacement key, i.e.,
  - at N, replace k with k' and at P, delete k'

Example: To delete 15 from the following B(3)-tree (assuming redistribution)

Here, k=15, N=[7, 15, 27], O=[21, 23, 25], k'=21, O=M=[21, 23, 25]



Deleting from a B-tree

Case 2.3: N is a non-leaf node, the root M of the predecessor subtree and the root O of the successor subtree are minimal.

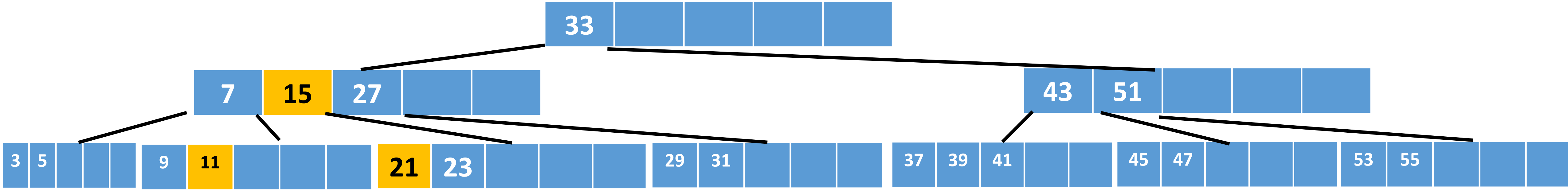
- merge nodes M and O by moving k into this merged node (let M be the merged node, release node O, shift children if require).

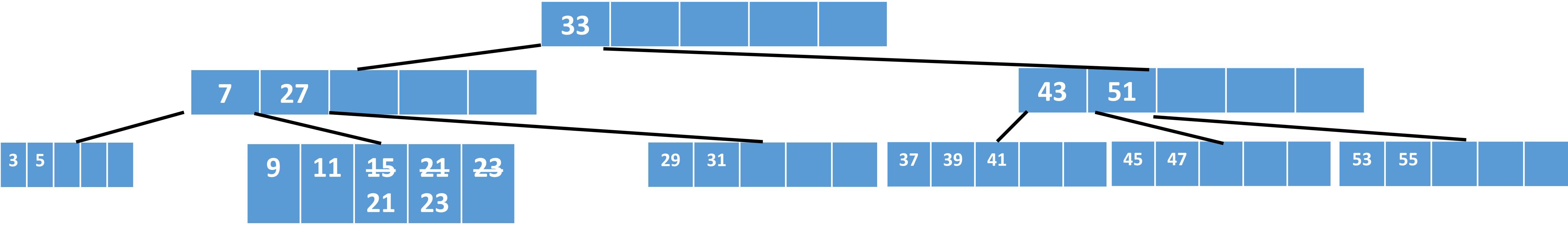
[Now, M is full (non-minimal), k is the median of M]

- delete (recursively) k from M.

Example: To delete 15 from the following B(3)-tree (assuming redistribution)

Here, k=15, N=[7, 15, 27], M=[9, 11], O=[21, 23]





Exercise

Write the pseudocode and compute the run-time for the following operations on a B(t)-tree:

- **Redistribution**
- **Identifying predecessor key**
- **Identifying successor key**
- **Deleting a key**