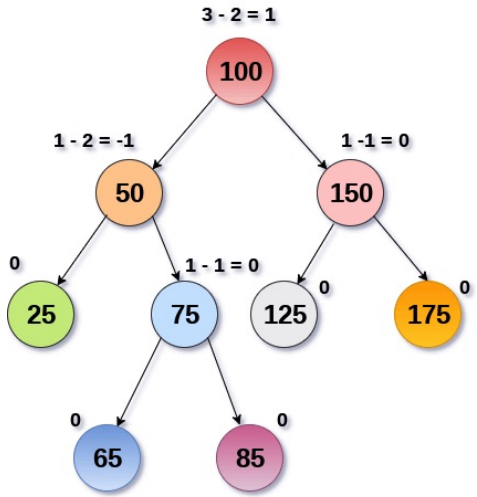
CSCB63 – Design and Analysis of Data Structures

Akshay Arun Bapat

Recap AVL Tree



AVL Tree

- one node stores one key
- what if we increase this?
  - can reduce the height further
  - faster search, as all operations dependent on height

B-Tree

- balanced search tree with possibly more than 2 children.
- number of child per node is called **fanout** (degree)

Use-case

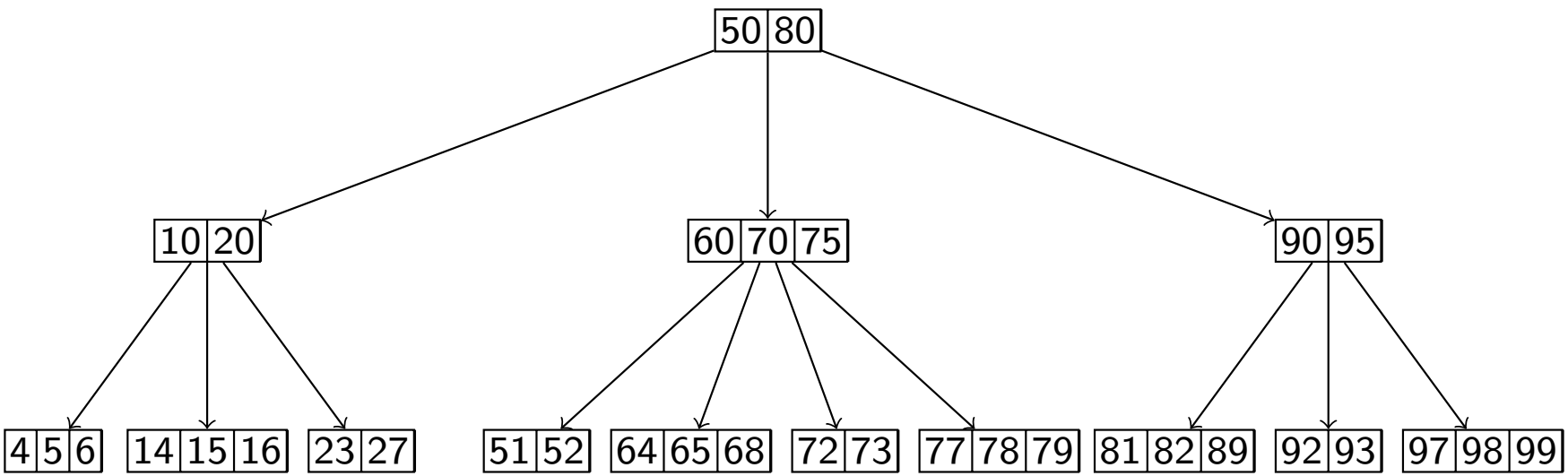
- enhancement of BST when searching in disk storage
- used extensively in databases as an index on tables

Every node stores:

- some number of key-value pairs
  - keys are sorted
  - maximum (fanout - 1)
  - minimum half full [non-root]
- the number of keys currently stored in the node
- internal nodes have pointers to all their children

B-Tree

- b-tree with fanout 5



Operations on B-Tree

- search - search for a key
- insert - insert a key-value in the tree
- delete - delete a key from the tree

B-Tree search

- similar to search in BST
- multi-way branching for more than 2 children
- if key found, return value, else "key not found"

```
0. search(x, k):
1.   for i in 0 to x.total:
2.     if x.keys[i] = k:
3.       return (k-value)
4.     elif x.keys[i] > k:
5.       break

6.     if x is leaf:
7.       return "key not found"
8.     else:
9.       return search(x.child[i], k)
```

B-Tree insert

- all inserts happen in leaf nodes
- have to ensure that node has the capacity to store a key
- if node overflow
  - split the node into two separate nodes
  - propagate the split upwards

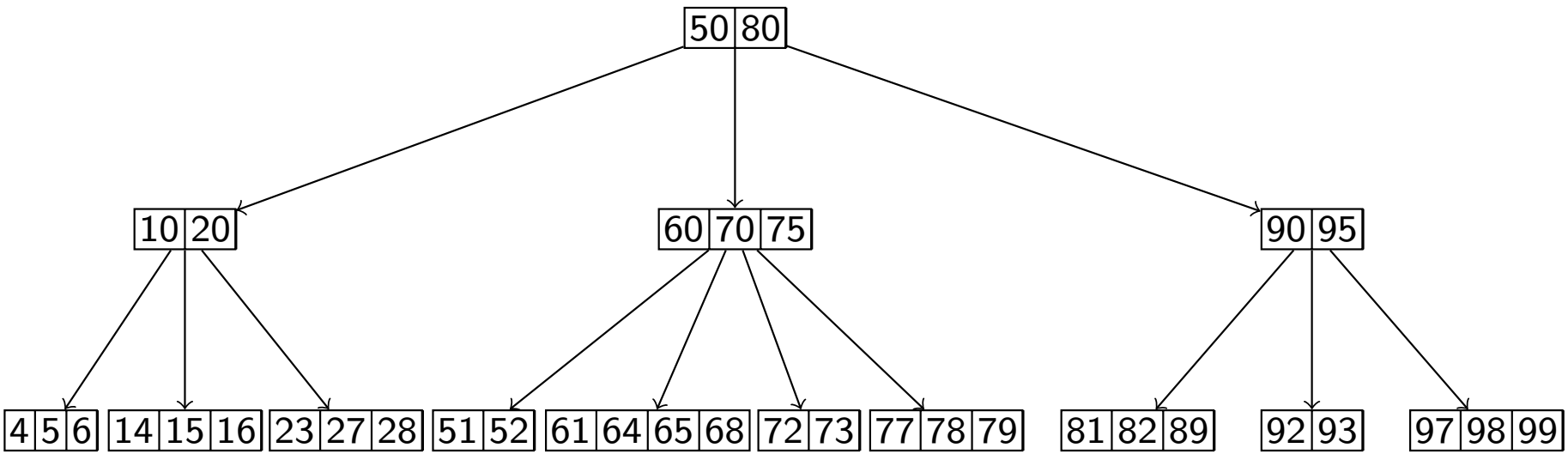
Complexity?

- one pass down the tree to insert the key into a leaf
- one pass up the tree to split if needed

B-Tree insert

- Insert 28 in the tree.
- Insert 61 in the tree.

B-Tree insert



Splitting the node

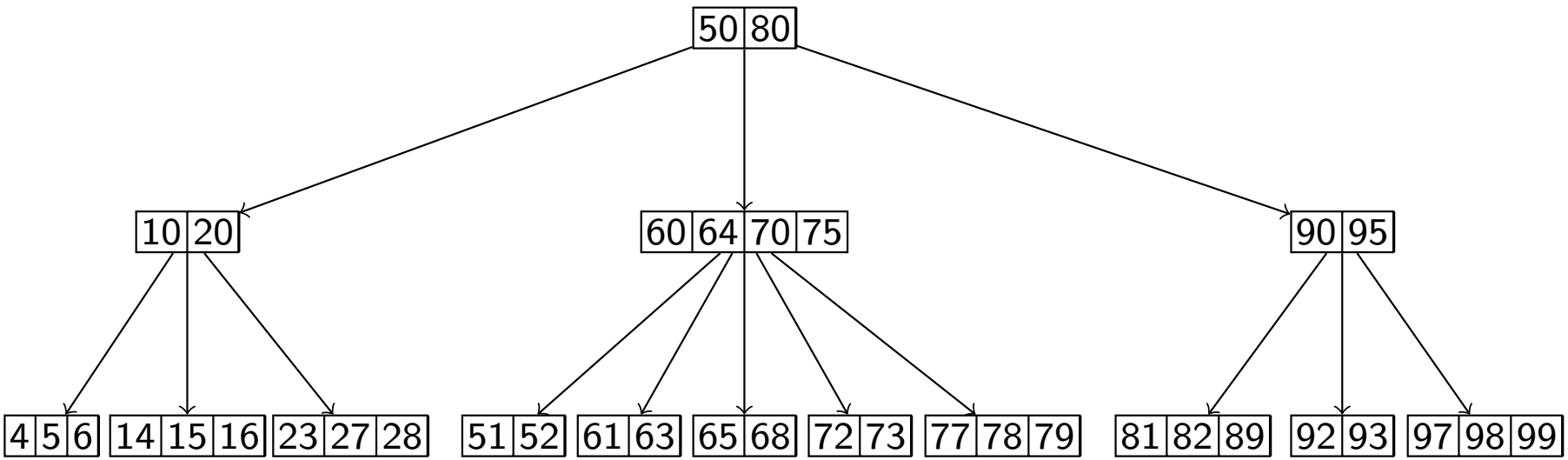
- add key first
- split into half (one node becomes two nodes)
- have to add a pointer in the parent
- could cause overflow in parent

What happens if the root node overflows? New root

B-Tree insert

- Insert 28 in the tree.
- Insert 61 in the tree.
- Insert 63 in the tree.

B-Tree insert



B-Tree insert

```
 0. insert(x, k):
 1.   for i in 0 to x.total:
 2.     if x.keys[i] = k:
 3.       update (k-value)
 4.     elif x.keys[i] > k:
 5.       break

 6.   if x is leaf:
 7.     shift keys from i right
 8.     x.keys[i] = k
 9.     if x overflows:
10.       split(x)
11.   else:
12.     insert(x.child[i], k)
13.     if x overflows:
14.       split(x)
```

B-Tree insert

```
0. split(x):
1.   median = (fanout - 1) / 2
2.   x_left = new node from keys 0 to median-1
3.   x_right = new node from keys median to fanout-1
4.   ascend(x.parent, x.keys[median], x_left, x_right)
5.   free node x
```

B-Tree insert

```
 0. ascend(p, key, x_left, x_right):
 1.   if p is nil:
 2.     root = new node with key 'key'
 3.     root.child[0] = x_left
 4.     root.child[1] = x_right
 5.     return root

 6.   for i in 0 to p.total:
 7.     if p.keys[i] > key:
 8.       break
 9.   shift keys from i right
10.   shift child pointers from i right
11.   p.keys[i] = key
12.   p.child[i] = x_left
13.   p.child[i+1] = x_right
```

B-Tree delete

- if key not present? No change
- some keys reside in leafs (easy to remove)
- some reside in internal nodes
- delete can delete any of them

- First, search the key
- If found in leaf node
  - Case 1: no underflow - delete and done
  - Case 2: underflow - delete and handle underflow
- If found in internal node
  - Case 1: if extra key in right child - replace by successor
  - Case 2: if extra key in left child - replace by predecessor
  - Case 3: no extra key in child - delete, merge children and handle underflow

B-Tree delete

How to handle underflow?

- Case 1: if extra key in left sibling - borrow from left
- Case 2: if extra key in right sibling - borrow from right
- Case 3: no extra key in siblings - merge with left sibling
  - with the parent key as well
  - will cause parent keys to reduce by 1
  - handle underflow in parent

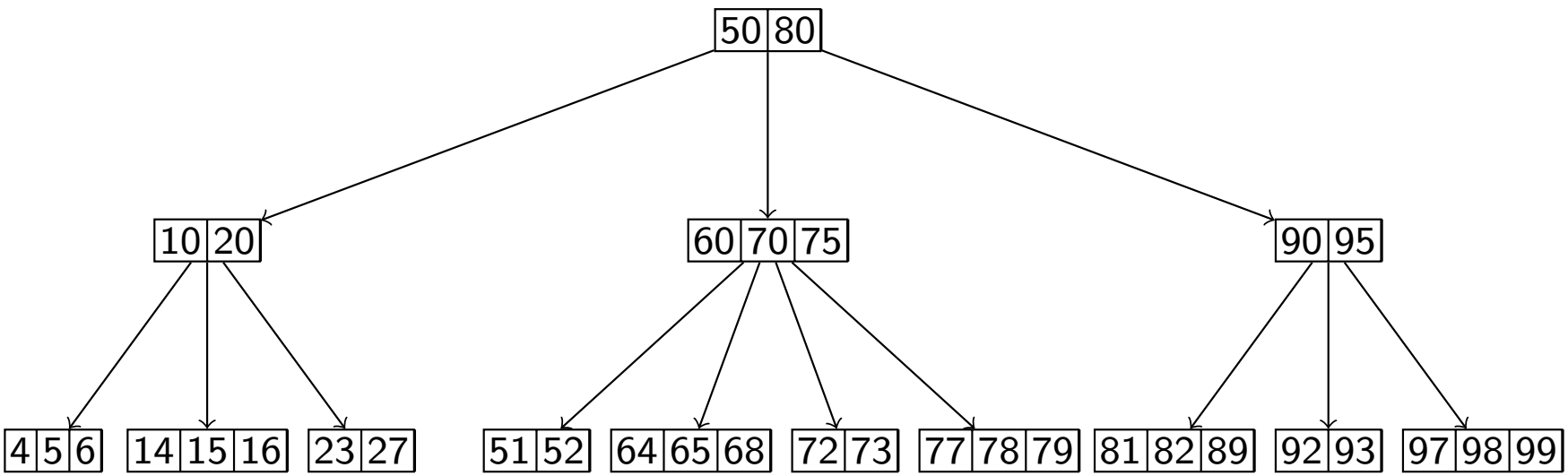
    What if parent was root?
    root does not have underflow,
    worst case it gets empty, tree height reduces by 1

Complexity?

- one pass down the tree to search the key
- one pass up the tree to propagate the underflow if needed

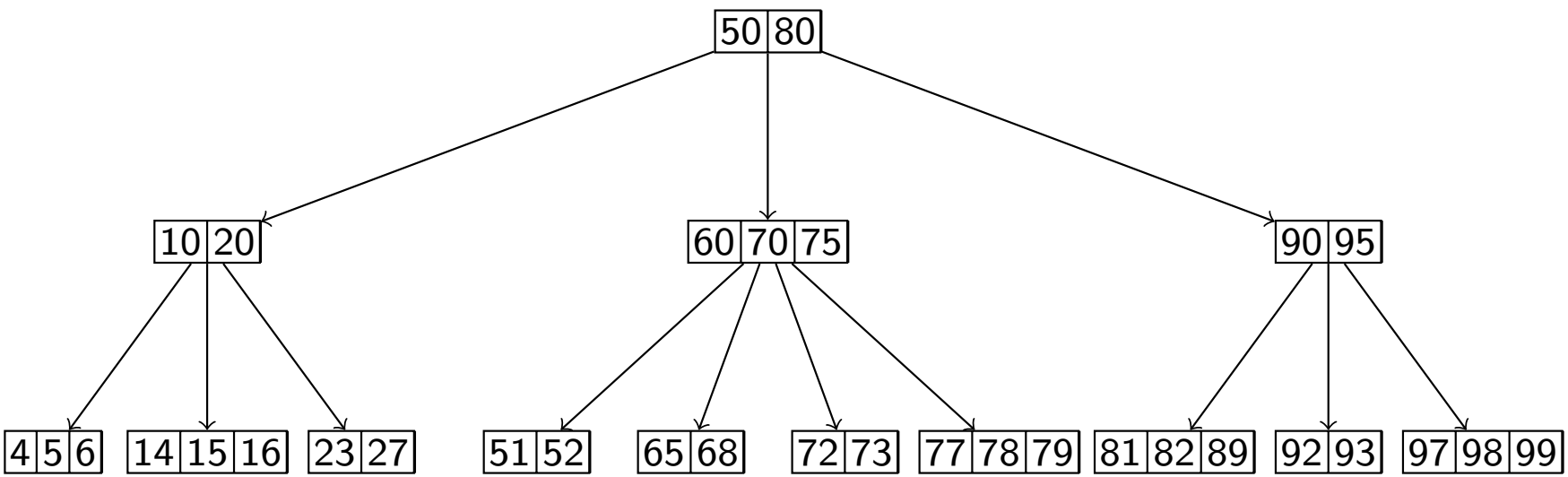
B-Tree delete

- b-tree with fanout 5
- delete the keys 64, 23, 72, 65, 20 in this order
- delete the keys 70, 95, 77, 80, 97 in this order
- delete the keys 6, 27, 60, 16, 50 in this order



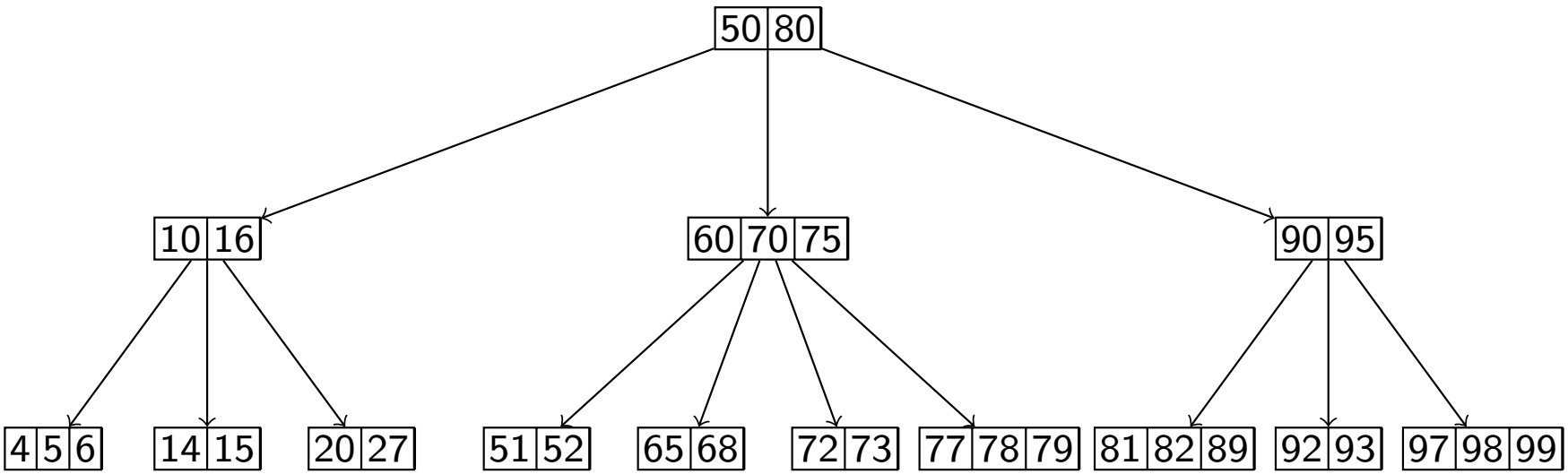
B-Tree delete

- b-tree with fanout 5
- deleting the key 64
- key in leaf, no underflow



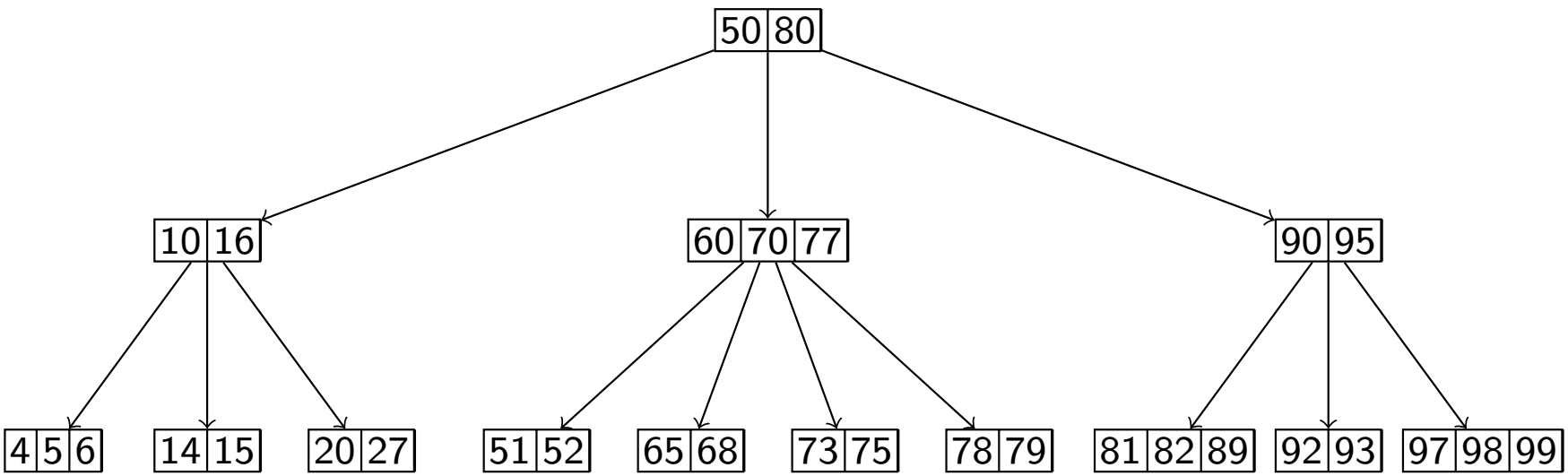
B-Tree delete

- b-tree with fanout 5
- deleting the key 23
- key in leaf, underflow
- borrow key from left sibling



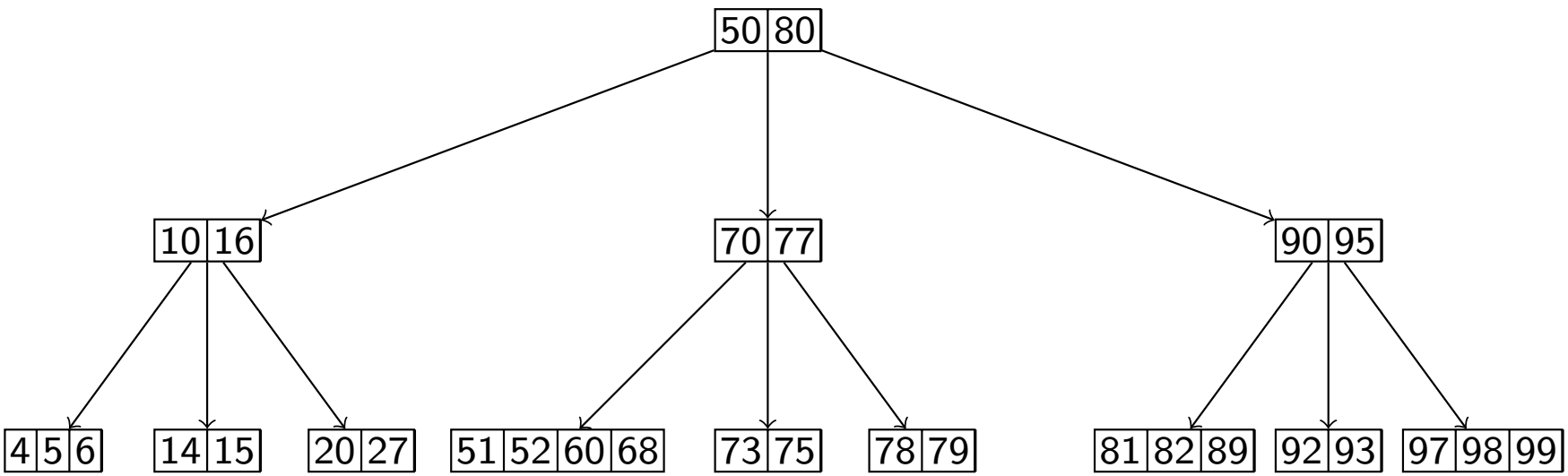
B-Tree delete

- b-tree with fanout 5
- deleting the key 72
- key in leaf, underflow
- borrow key from right sibling



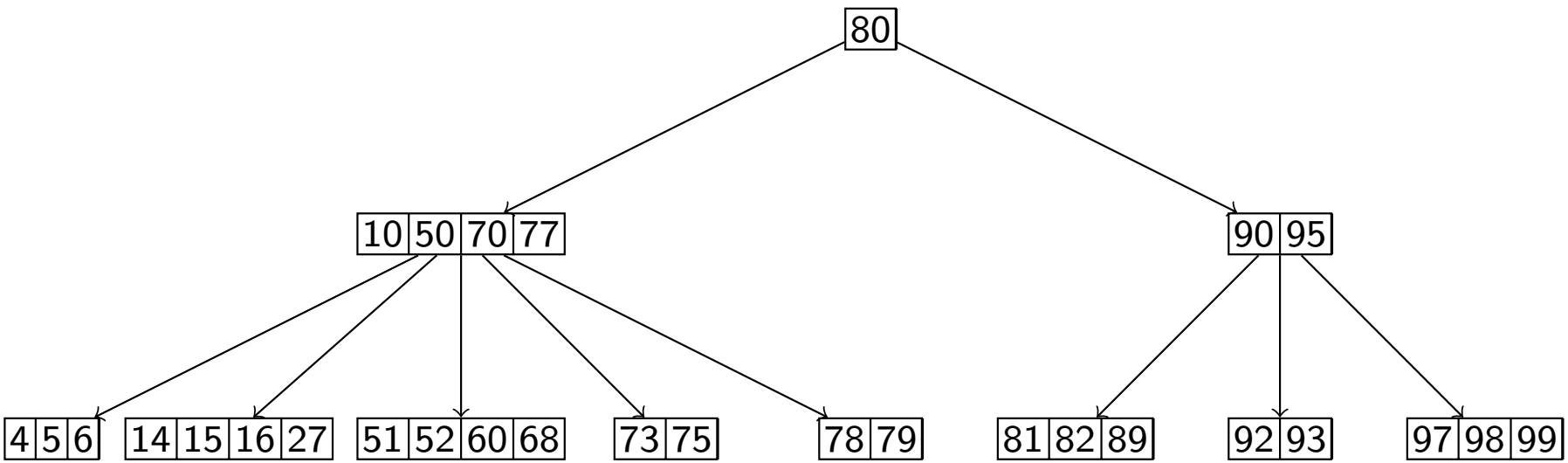
B-Tree delete

- b-tree with fanout 5
- deleting the key 65
- key in leaf, underflow
- borrow key from parent, merge with left sibling



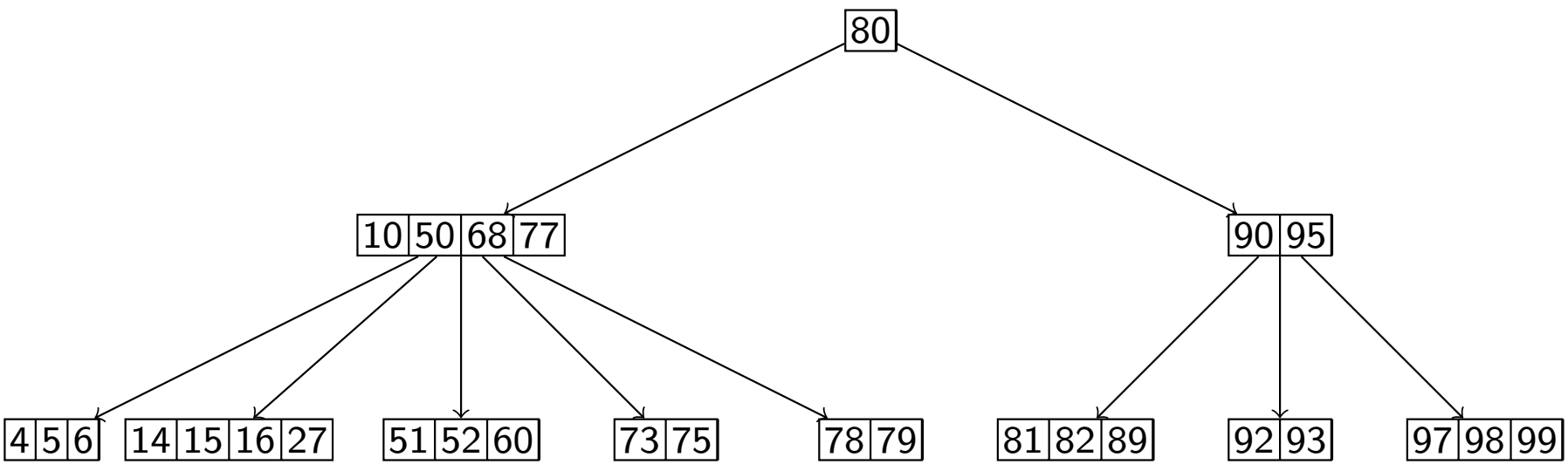
B-Tree delete

- b-tree with fanout 5
- deleting the key 20
- key in leaf, underflow
- borrow key from parent, merge with left sibling
- parent also underflow, borrow key from parent, merge with right sibling
- root has no underflow limit unless empty



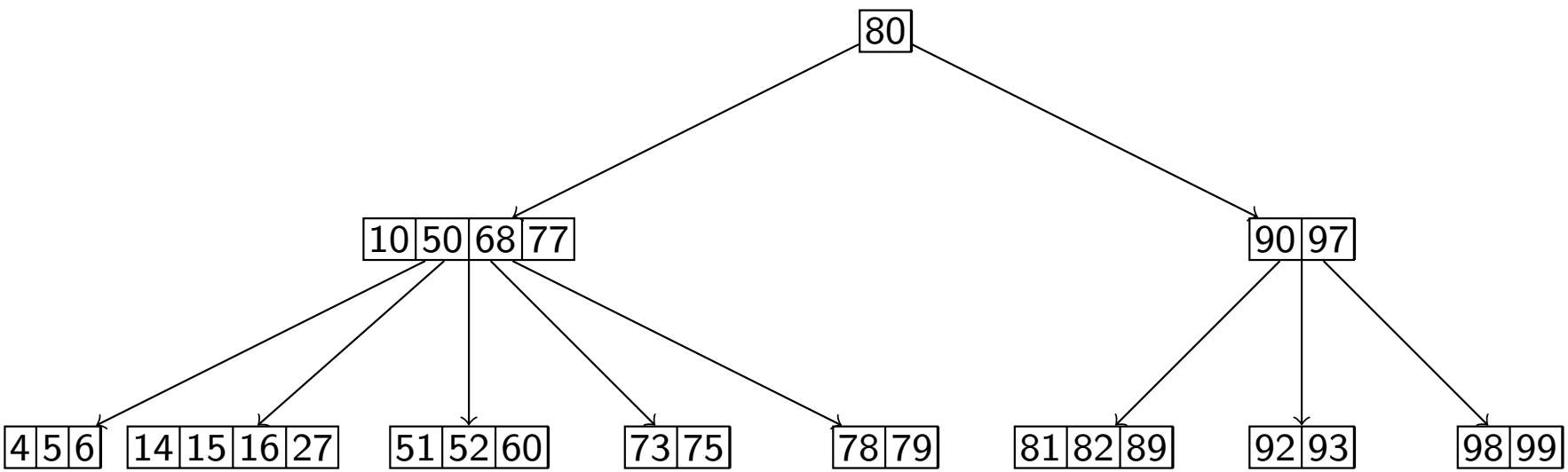
B-Tree delete

- b-tree with fanout 5
- deleting the key 70
- key in internal node, replace by inorder predecessor



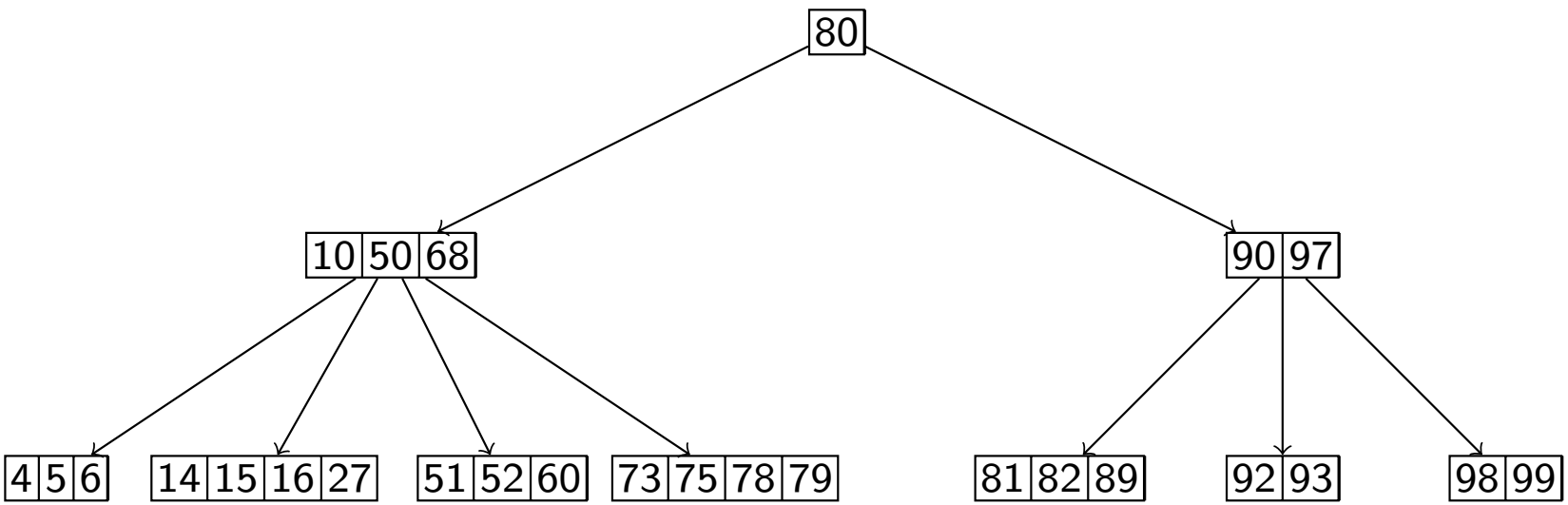
B-Tree delete

- b-tree with fanout 5
- deleting the key 95
- key in internal node, replace by inorder successor



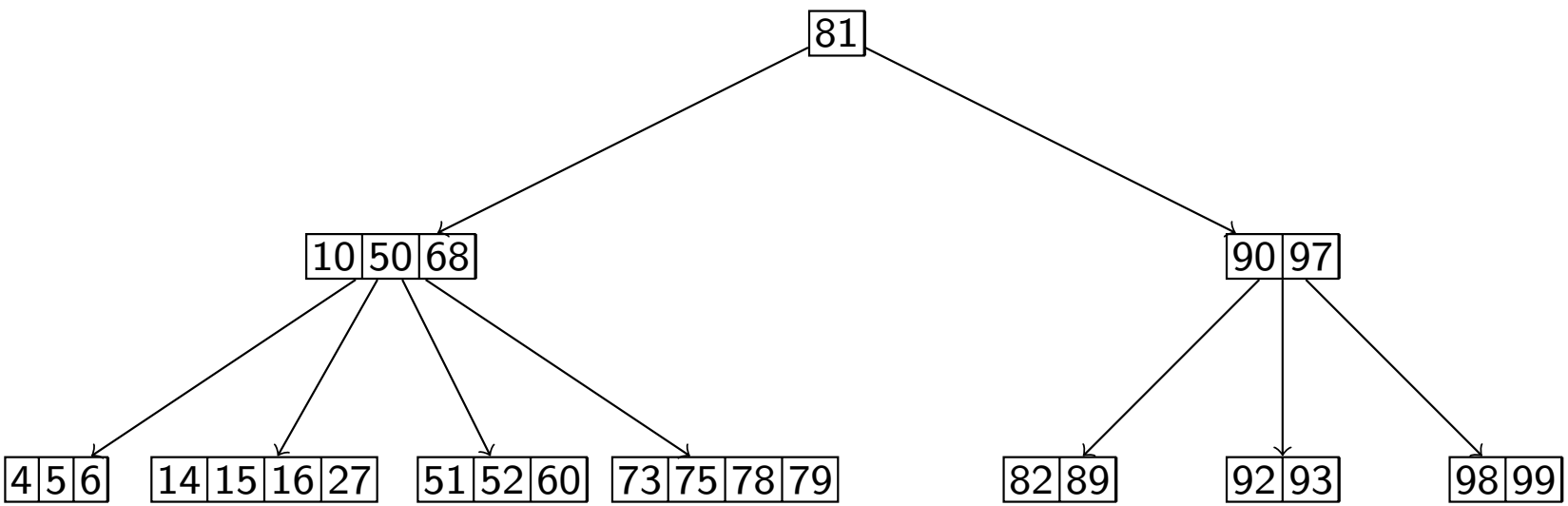
B-Tree delete

- b-tree with fanout 5
- deleting the key 77
- key in internal node, remove, merge children
- node not underflow



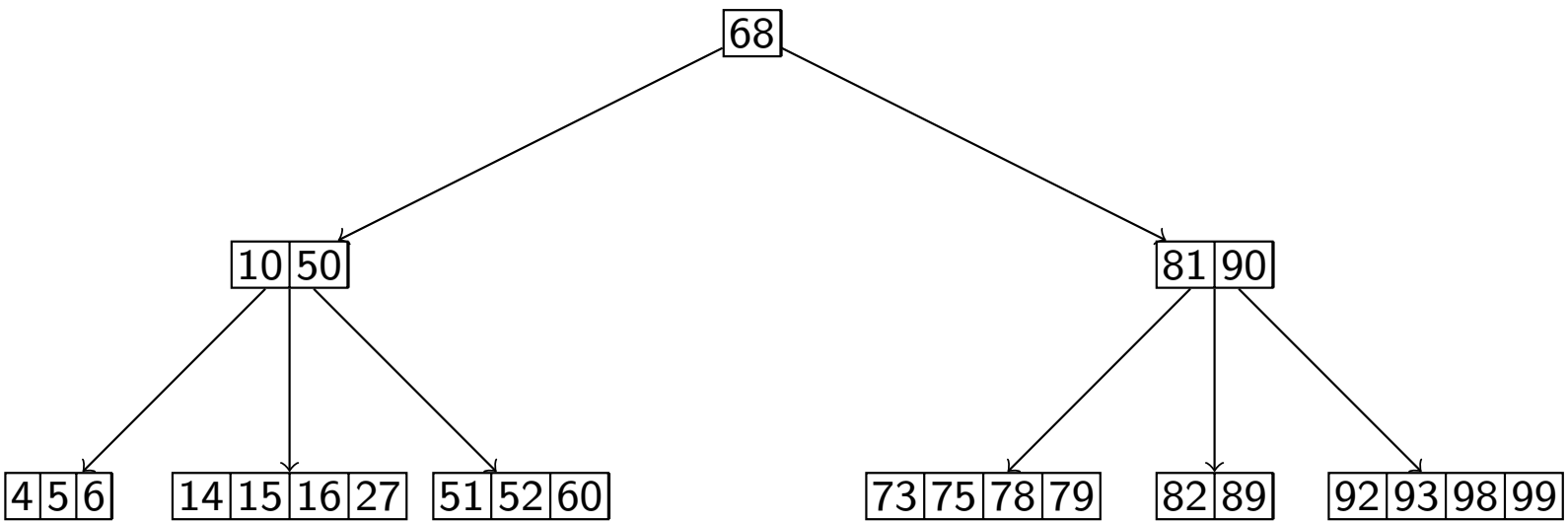
B-Tree delete

- b-tree with fanout 5
- deleting the key 80
- key in internal node, replace by inorder successor



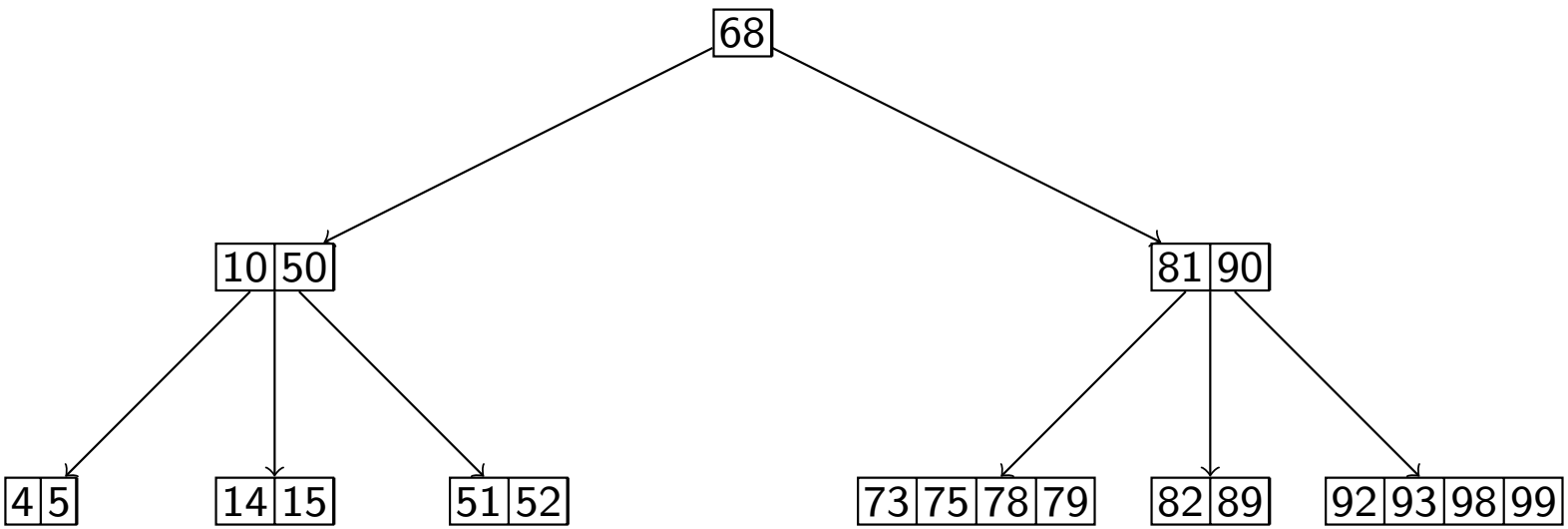
B-Tree delete

- b-tree with fanout 5
- deleting the key 97
- key in internal node, remove, merge children
- node underflow, borrow from left sibling



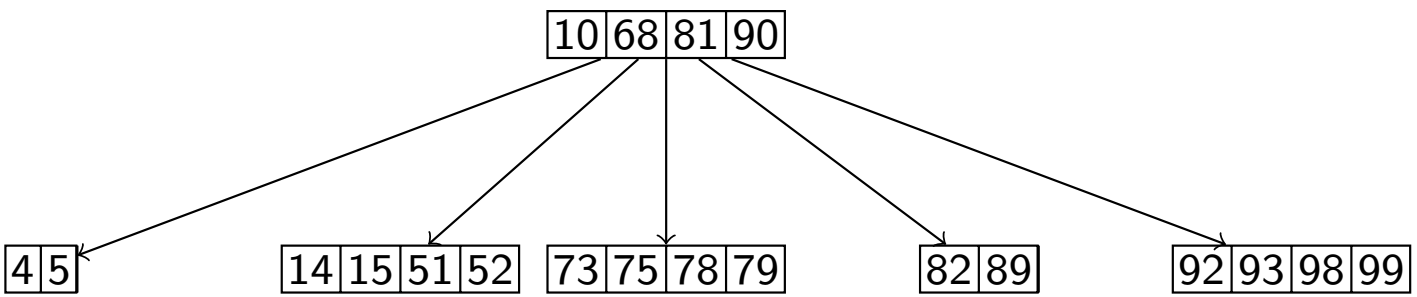
B-Tree delete

- b-tree with fanout 5
- deleting the keys 6, 27, 60 and 16
- all keys in leaf, no underflow



B-Tree delete

- b-tree with fanout 5
- deleting the key 50
- key in internal node, remove, merge children
- node underflow, borrow key from parent, merge with right sibling
- parent (root) has no key, new root of B-Tree



B-Tree delete

```
 0. delete(x, k):
 1.   for i in 0 to x.total:
 2.     if x.keys[i] >= k:
 3.       break

 4.   if x.keys[i] != k:
 5.    if x is leaf:
 6.        return "key not found"
 7.      else:
 8.        delete(x.child[i], k)

 9.   else:
10.     if x is leaf:
11.       handle_delete_leaf(x, i)
12.     else:
13.       handle_delete_internal(x, i)
```

B-Tree delete

```
 0. handle_delete_leaf(x, i):
 1.   shift keys from i+1 left
 2.   if x underflows handle_underflow(x)

 0. handle_delete_internal(x, i):
 1.   if x.child[i+1].total > minimum keys:
 2.     x.keys[i] = get_successor(x, i)
 3.     delete(x.child[i+1], x.keys[i])
 4.   elif x.child[i].total > minimum keys:
 5.     x.keys[i] = get_predecessor(x, i)
 6.     delete(x.child[i], x.keys[i])
 7.   else:
 8.     shift keys from i+1 left
 9.     shift child pointers from i+1 left
10.     x.child[i] = merge(x, x.child[i], x.child[i+1])
11.     if x underflows handle_underflow(x)
```

B-Tree delete

```
 0. handle_underflow(x):
 1.   if x is root and x.total = 0:
 2.     new_root = root.child[0]
 3.     free root
 4.     return new_root

 5.   if x.left_sibling.total > minimum keys:
 6.     borrow key from left, through parent
 7.   elif x.right_sibling.total > minimum keys:
 8.     borrow key from right, through parent
 9.   else:
10.     merge(x.parent, x, x.left_sibling)
11.     handle_underflow(x.parent)
```