COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Balanced Trees

Slide Set: 10

Learning Objectives

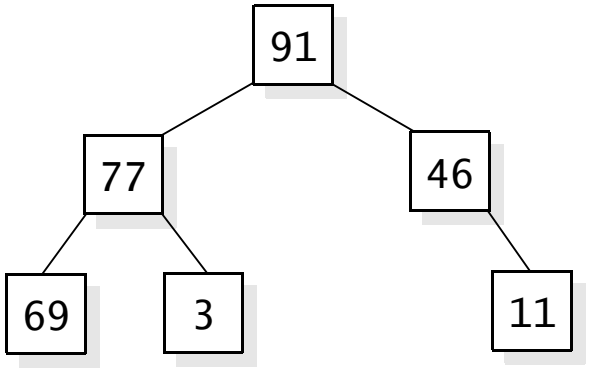
- The rules for B-tree, heap, AVL tree
- Algorithms for searching, inserting, and removing an element from a B-tree
- Use the STL priority queue, heap, map and multimap, and be able to explain their typical implementations using balanced trees
- Recognize which operations have logarithmic worst-case performance on balanced trees

Heaps

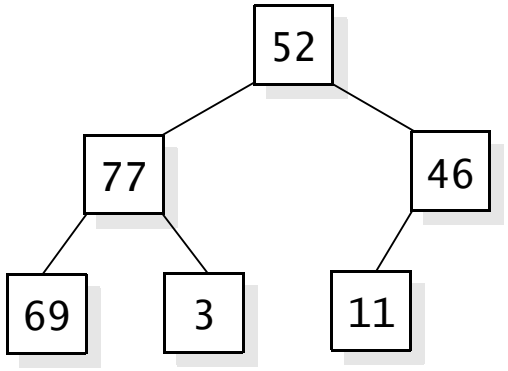
Heap Storage Rules

- **A binary tree where the entries of the nodes can be compared with the less-than operator of a strict weak ordering**
- In addition:
  - Heap is **a complete binary tree**, so that every level except the deepest must contain as many nodes as possible; and at the deepest level all the nodes are as far left as possible
  - A **max-heap** is a **complete binary tree** in which the value in each internal node is greater than or equal to the values in the children of that node
  - A **min-heap** is defined similarly

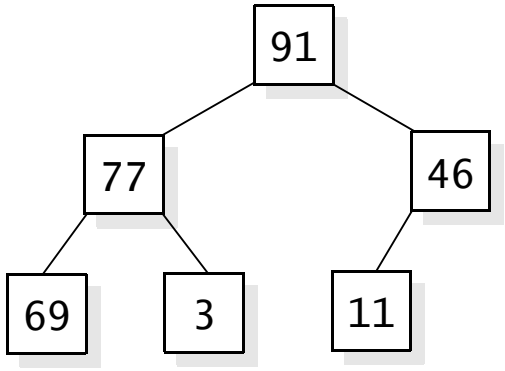
Heap Storage Rules



Not a heap because it is not complete



Not a heap because the value of node 52 is less than its child 77



A heap

Heap Storage Rules

- A heap can be implemented with the binary tree nodes
- As heap is a complete binary tree, **it can be easily implemented as an array**
  - A fixed size array if we know the maximum size
  - A dynamic array to grow and shrink the size

The Priority Queue with Heaps

- In a **priority queue:**
  - Entries are placed in the queue and later taken out
  - Each entry in a priority queue can be compared with the other entries using a less-than operator
  - The highest priority entries always leave first

- In the heap implementation of a priority queue:
  - Each node of the heap contains one entry
  - Each entry can be compared to each of the other entries by the less-than operator

Adding an Entry to a Heap

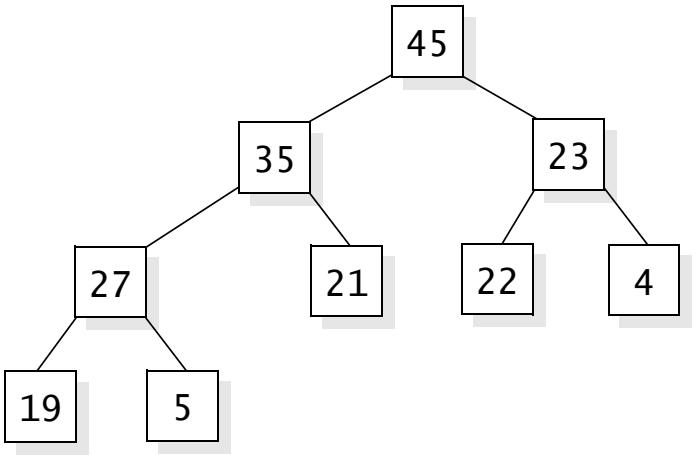
Pseudocode for **adding an entry**

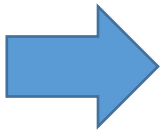
1. **Place the new entry in the heap in the first available location**. This keeps the structure as a complete binary tree, but it might no longer be a heap since the new entry's parent might be less than the new entry
2. ***while*** **(the new entry's parent is less than the new entry)**

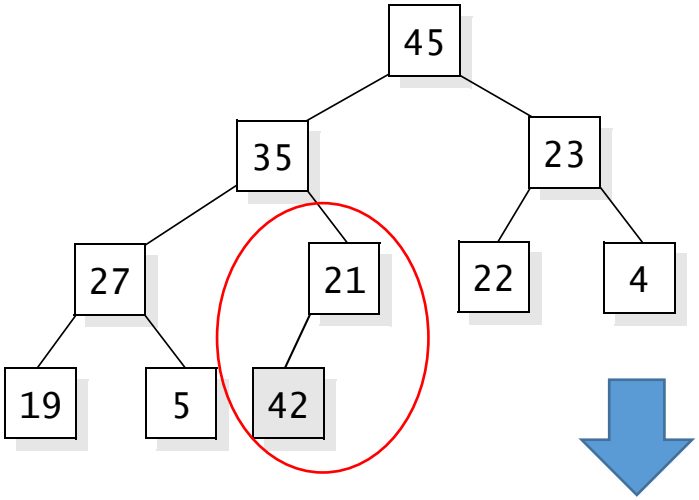
   swap the new entry with its parent

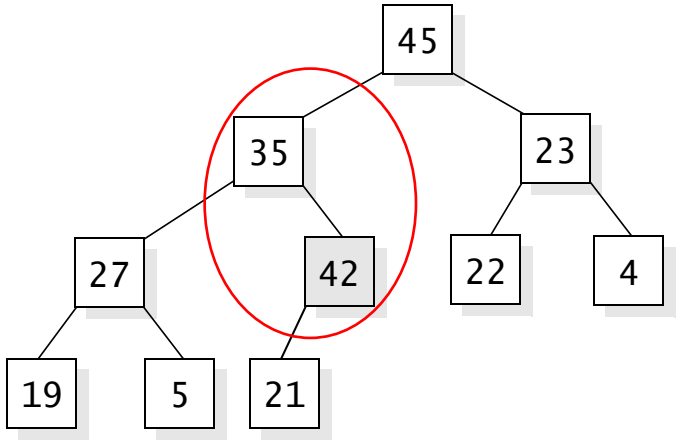
Adding an Entry to a Heap

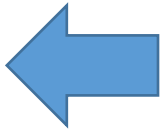
- Assume we want to add 42

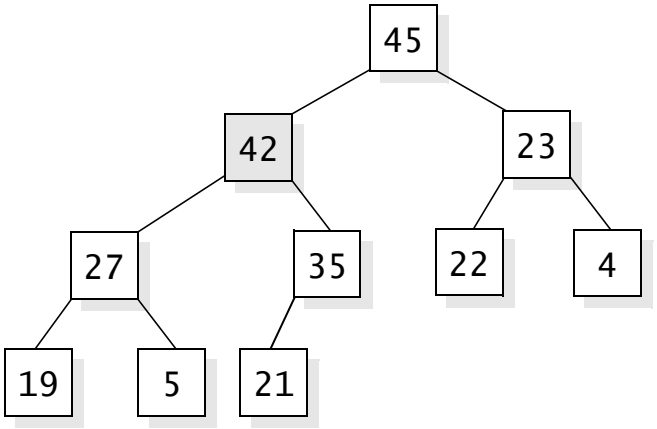












Adding an Entry to a Heap

- **Re-heapification Upward:**
  - The “new entry rising” stops when the new entry has a parent with a higher or equal priority, or when the new entry reaches the root

Removing an Entry from a Heap

- **Always remove the entry with the highest priority:** The entry that stands "on top of the heap"

Pseudocode for Removing an Entry

1. Copy **the entry at the root** of the heap to the variable that is used to return a value

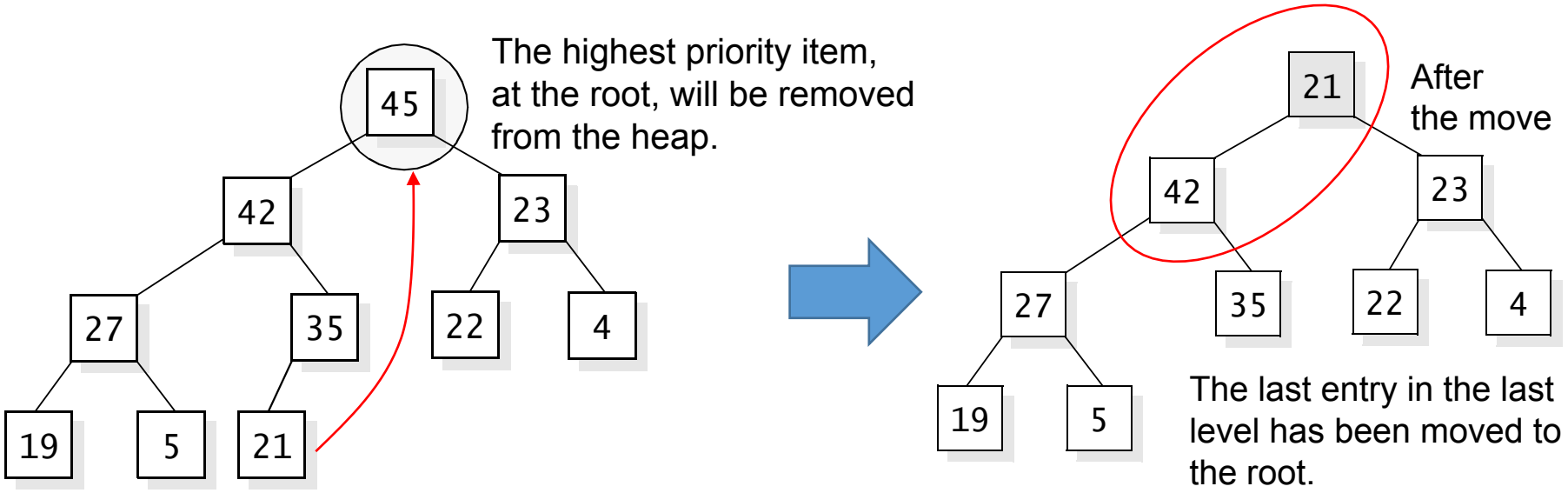
2. Copy the **last entry in the deepest level to the root**, and then take this last node out of the tree; This entry is called the "out-of-place" entry

3. *while* (the out-of-place entry is less than one of its children)
   Swap the out-of-place entry with its highest child

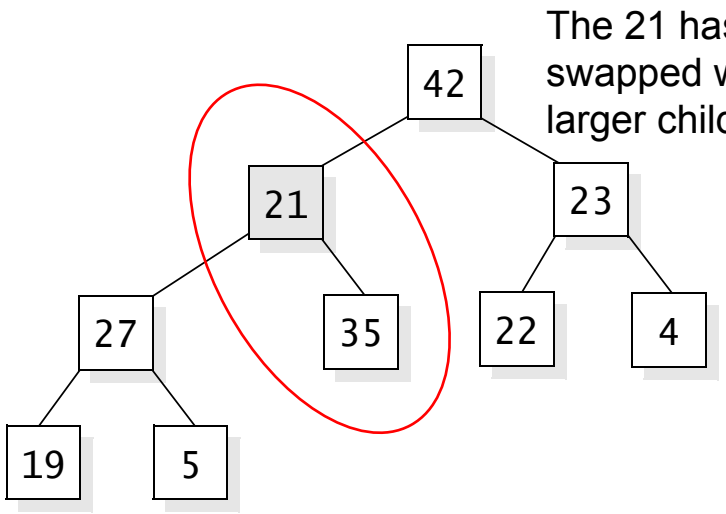
4. Return the answer that was saved in Step 1

Adding an Entry to a Heap

- Assume we want to remove an entry from the heap

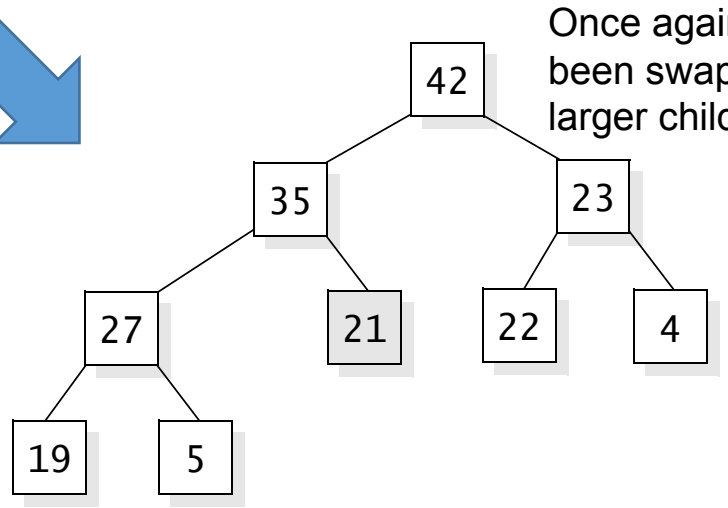


Adding an Entry to a Heap



The 21 has been swapped with its larger child.

Re-heapification downward



Once again, the 21 has been swapped with its larger child.

The STL Priority Queue and the Heap Algorithms

The STL Priority Queue

- The STL includes a priority queue class

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>   >
class priority_queue;
```

- **`T`**

Type of the elements

- **`Container`**

Type of the internal *underlying container* object where the elements are stored

- **`Compare`**

A binary predicate that takes two elements (of type T) as arguments and returns a bool

The STL Priority Queue

- The container:
  - May be any of the standard container class templates or some other specifically designed container class
  - Shall be accessible through random access iterators and support the following operations: `empty()`, `size()`, `front()`, `push_back()`, `pop_back()`
- The standard container classes `vector` and `deque` fulfill these requirements

The STL Priority Queue

Some Member functions:

- **`pop():`** Removes the **highest priority** item of the priority queue
- **`push(const Item& entry):`** Adds an item to the priority queue
- **`top():`** Returns the **highest priority** item of the queue (without removing it)

- **Note:** The `top()` prototype is: `const value_type& top() const;`
- It cannot be used to change the top item
- **Why?**

- The `top()` member function effectively calls member `front` of the underlying container object.

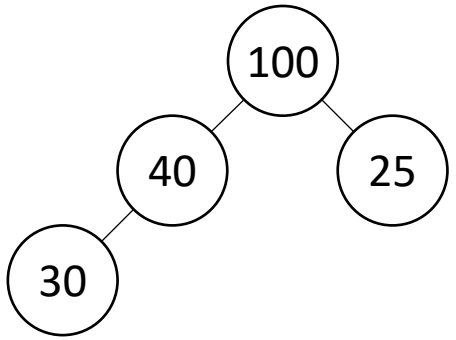
The STL Priority Queue

```
#include <iostream>         // std::cout
#include <queue>            // std::priority_queue
int main ()
{
    std::priority_queue<int> mypq;

    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);

    std::cout << "Popping out elements...";
    while (!mypq.empty())
    {
        std::cout << ' ' << mypq.top();
        mypq.pop();
    }
    std::cout << '\n';

    return 0;
}
```

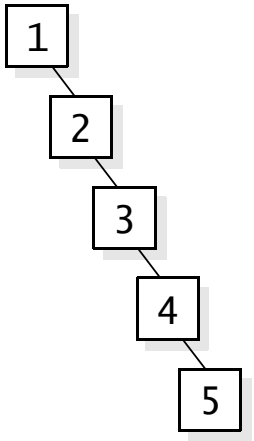


Output:
```
Popping out elements... 100 40 30 25
```

AVL Tree

The Problem of Unbalanced Trees

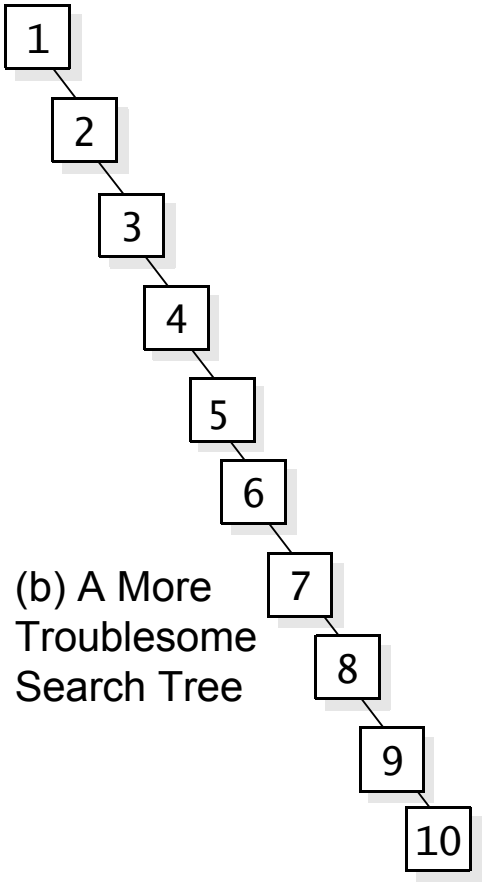
- Assume we add 1,2,3,4, and 5 to a **binary search tree**



(a) A Troublesome Search Tree

We are no better off than the linked-list implementation

- Assume we add 1,2,3,4,5,6,7,8,9 and 10 to a binary search tree



(b) A More Troublesome Search Tree

AVL Tree

- BST's worst-case performance is closest to linear search algorithms, that is $O(n)$
- As we cannot predict data pattern and their frequencies, a need arises to balance out the existing BST
- An AVL tree is a **self-balancing** binary search tree
  - The first invented self-balancing tree

- **The heights of the two child subtrees of any node differ by at most one**
- If at any time they differ by more than one, rebalancing is done to restore this property
- Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases
- Insertions and deletions may require the tree to be rebalanced **by one or more tree rotations**

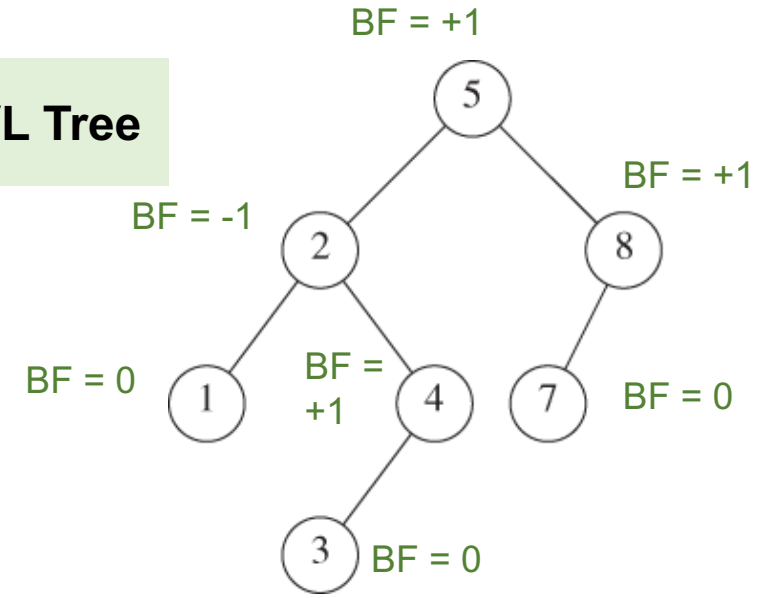
AVL Tree

Balance Factor

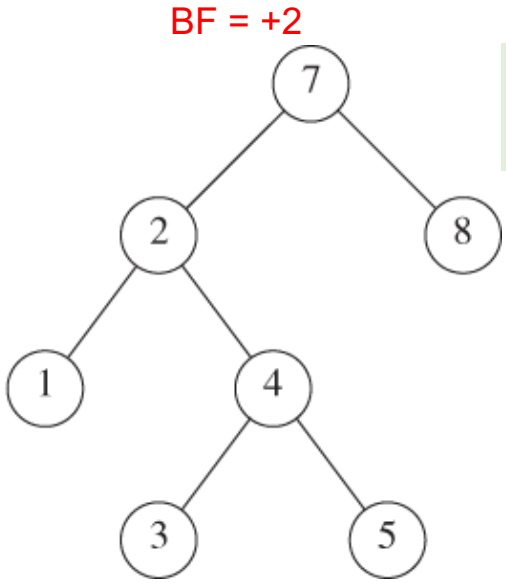
- The **Balance Factor** of node *n* is defined as:
$$BalanceFactor(n) = Height\big(LeftSubtree(n)\big) - Height(RightSubtree(n))$$

- A binary tree is called **AVL tree** if the following invariant holds for every node *n* in the tree
$$BalanceFactor(n) \in \{-1, 0, +1\}$$

AVL Tree



Not an AVL Tree



Rotation

An AVL tree may perform the following four kinds of rotations to balance itself:

- Left rotation
- Right rotation

Single rotation

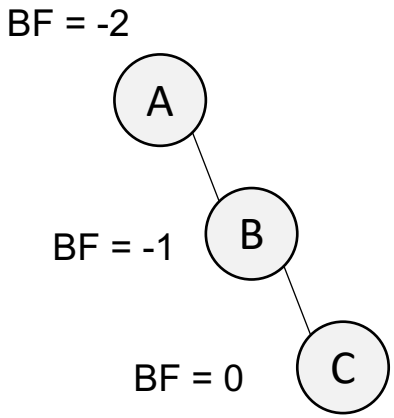
- Left-Right rotation
- Right-Left rotation

Double rotation

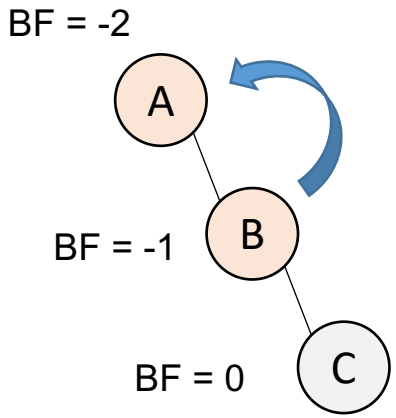
Rotation

Left Rotation

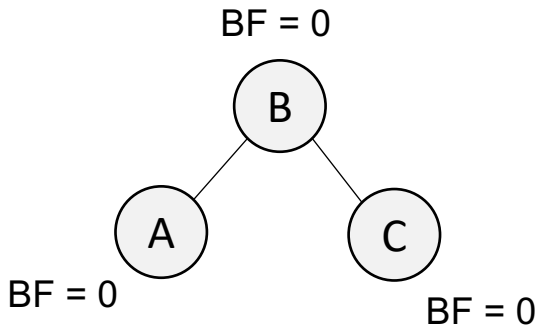
- If a tree becomes unbalanced, when a node is inserted into the **right subtree of the right subtree**
- **Left rotation: Swaps the parent node with its right child**



Unbalanced tree



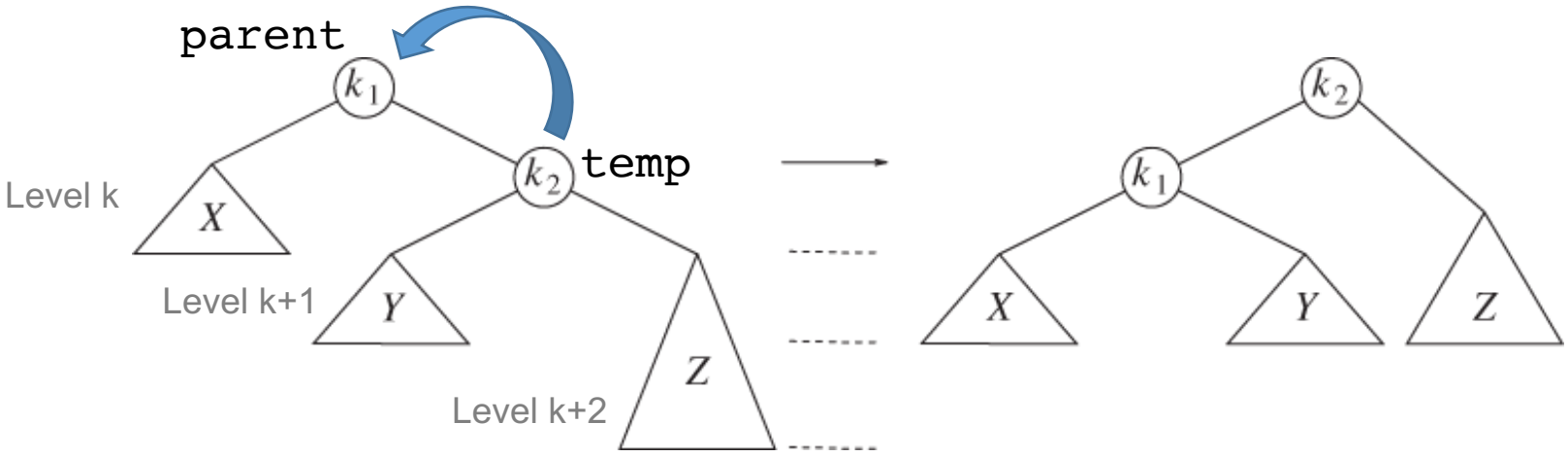
Left rotation



Balanced

Rotation

❑Example: Left Rotation (Cont'd)

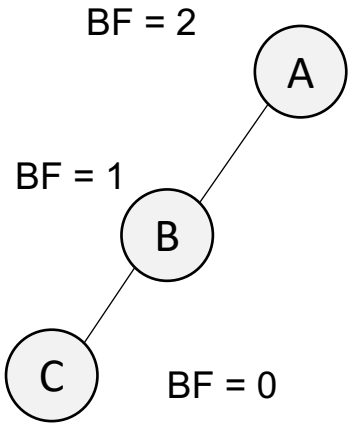



```
template <class Item>
binary_tree_node<Item>* left_rotation(binary_tree_node<Item>*&parent)
{
   binary_tree_node<Item>* temp;
   temp = parent->right();
   parent->set_right(temp->left());
   temp->set_left (parent);
   return temp;
   }
```

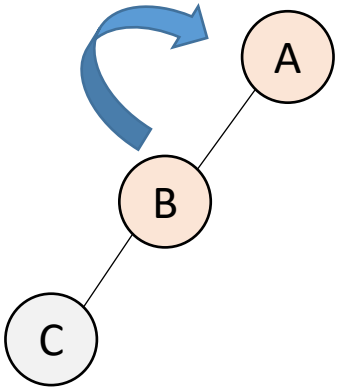
Rotation

Right Rotation

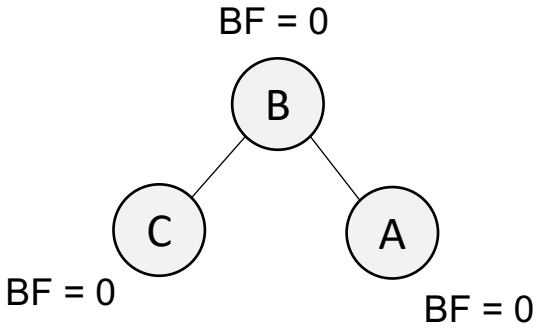
- If a tree becomes unbalanced, when a node is inserted into the **left subtree of the left subtree**
- **Right rotation: Swaps the parent node with its left child**



Unbalanced tree



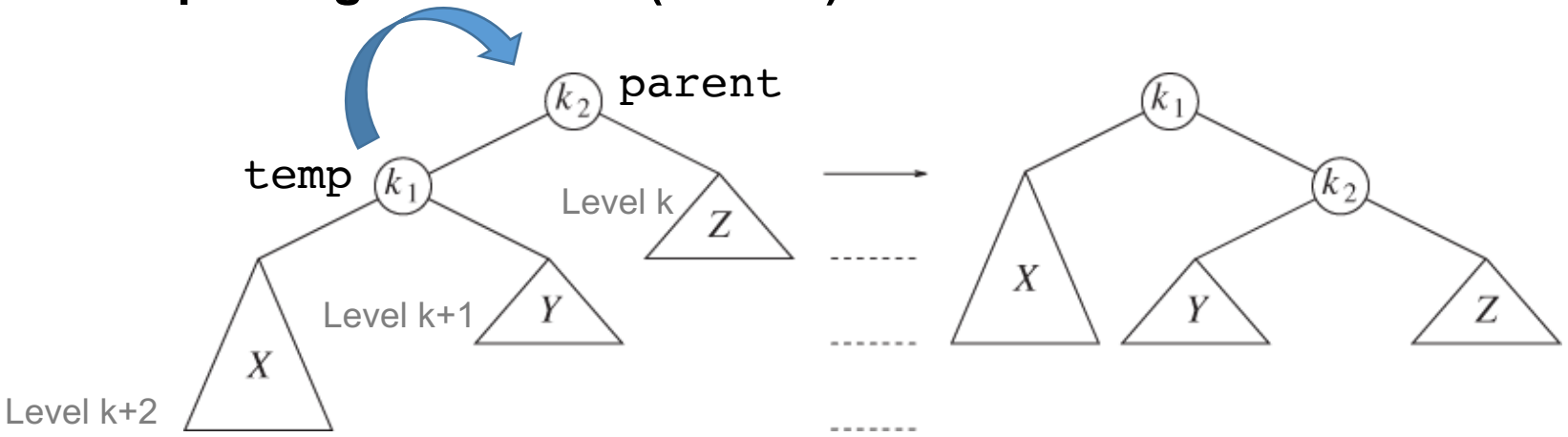
Right rotation



Balanced

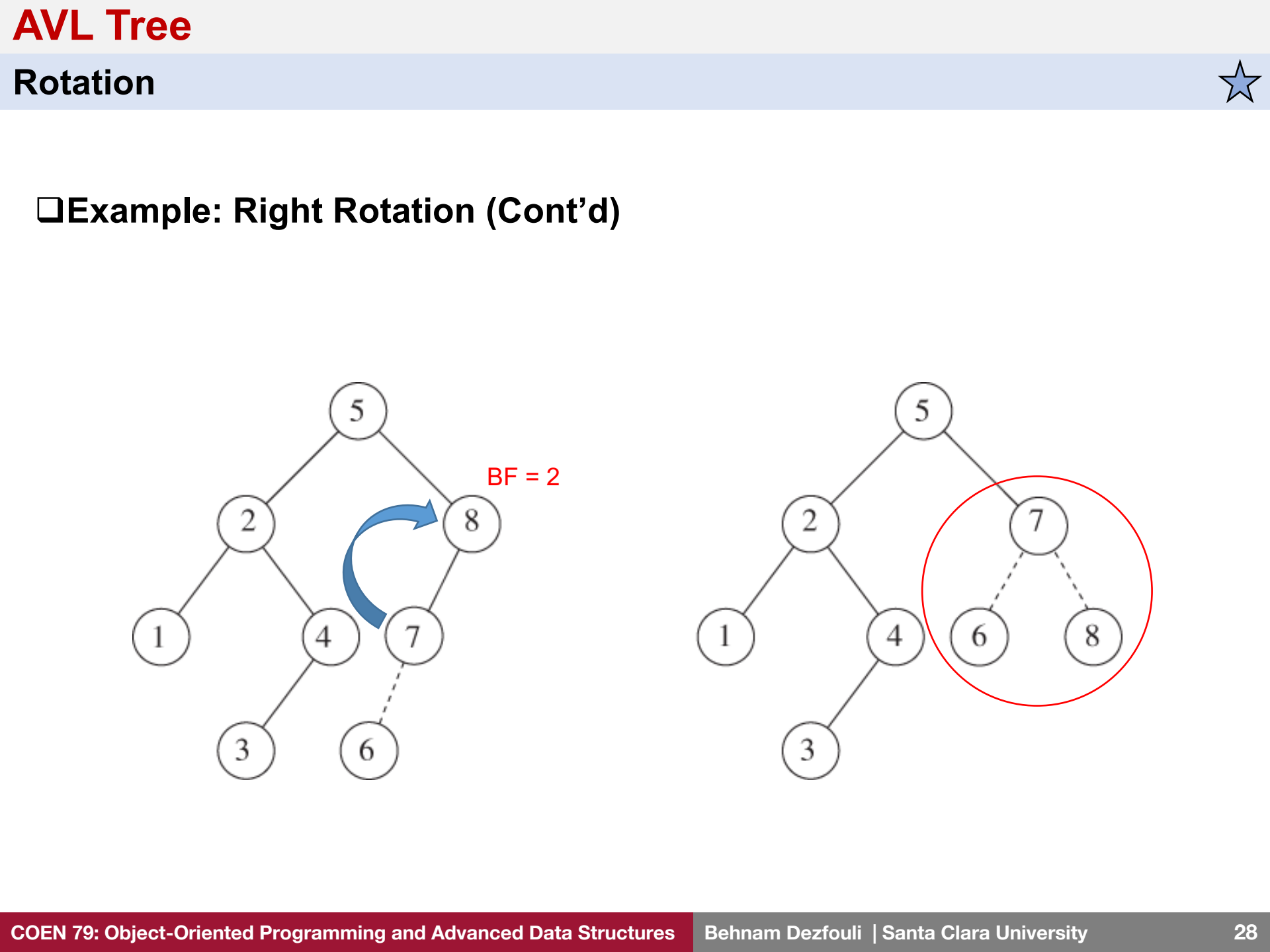
Rotation

❑Example: Right Rotation (Cont'd)




```
template <classItem>
binary_tree_node<Item>* right_rotation(binary_tree_node<Item>*& parent)
{
        binary_tree_node<Item>* temp;
        temp = parent->left();
        parent->set_left (temp->right());
        temp->set_right(parent);
        return temp;
}
```

Rotation

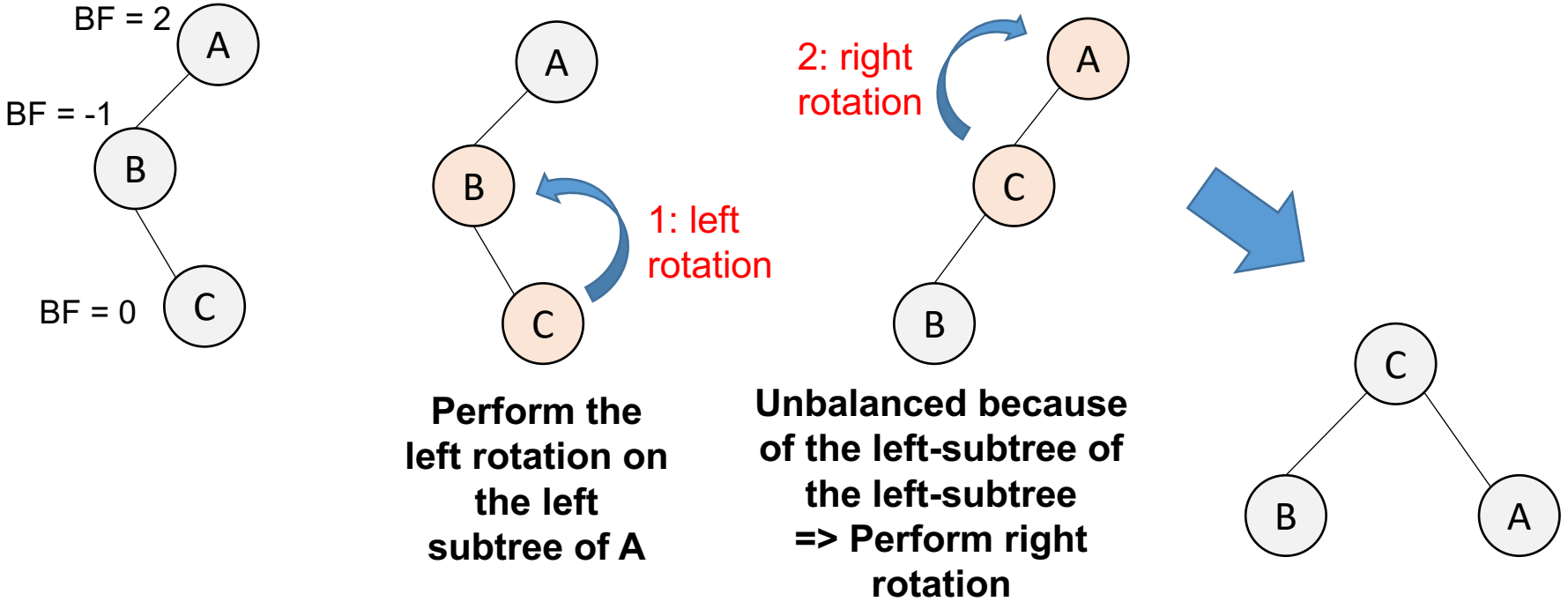


❑Example: Right Rotation (Cont'd)

Rotation

Left-Right Rotation

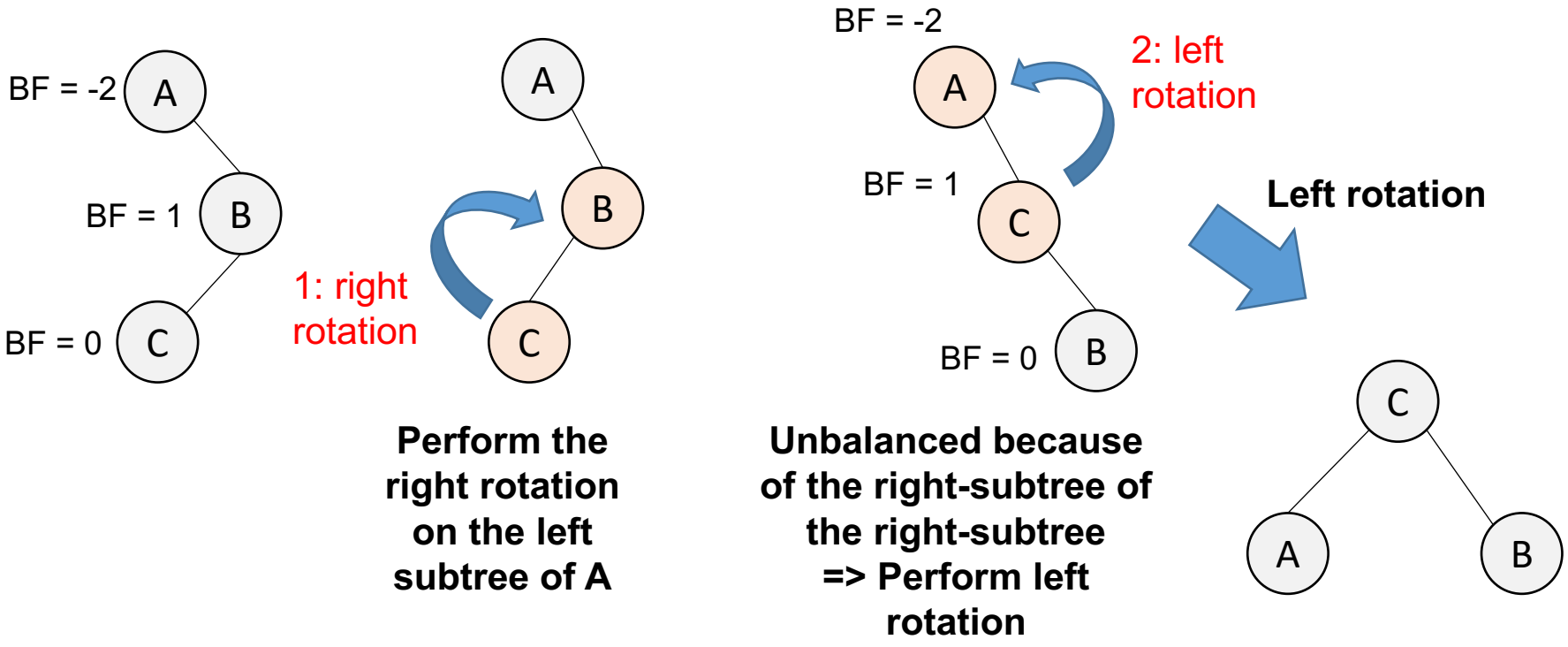
- A node has been inserted into the right subtree of the left subtree
- **Note: B < C < A ⇒ C becomes the new parent**



Rotation

Right-Left Rotation

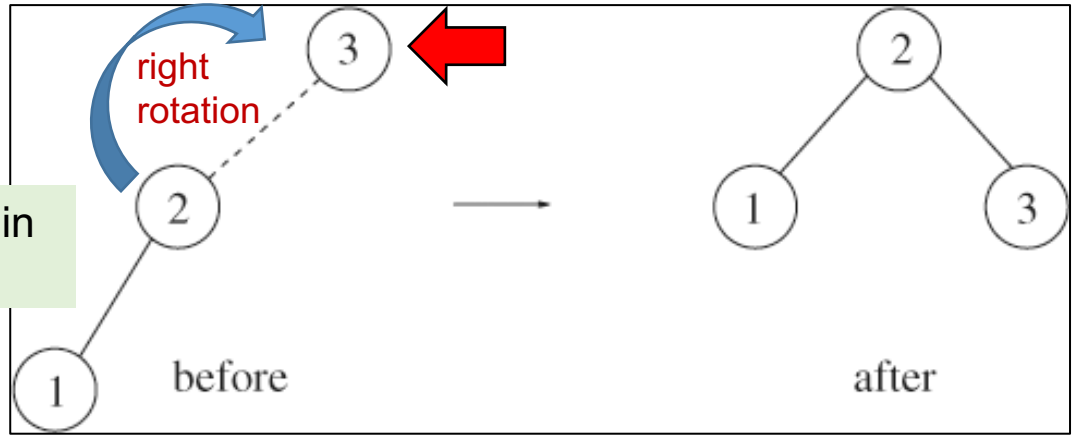
- A node has been inserted into the left subtree of the right subtree
- **Note: A < C < B ⇒ C becomes the new parent**



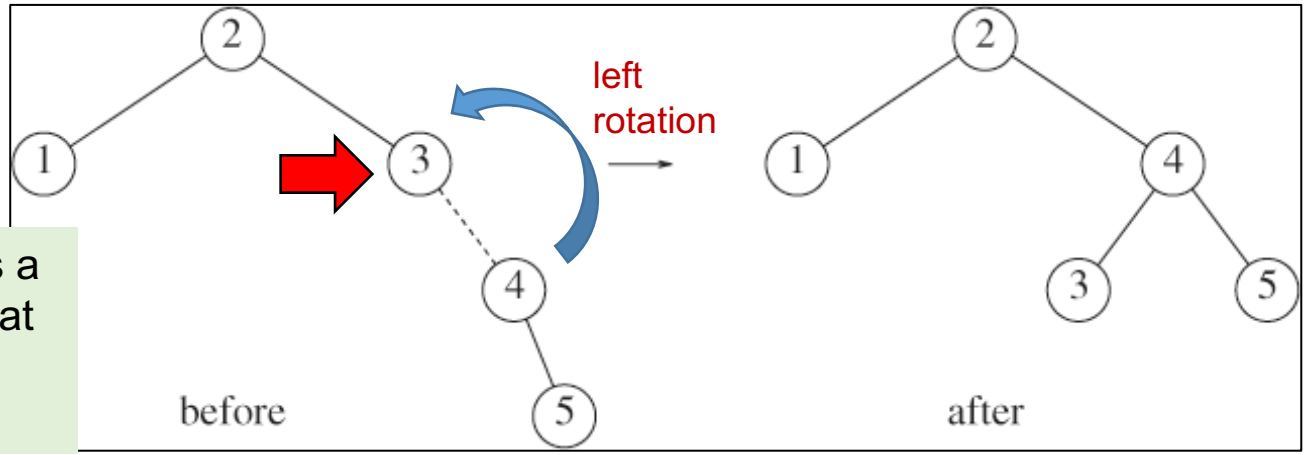
AVL Tree

- Example: We start with an initially empty AVL tree and insert the items 3, 2, 1, and then 4 through 7 in sequential order, then we insert 16 through 10

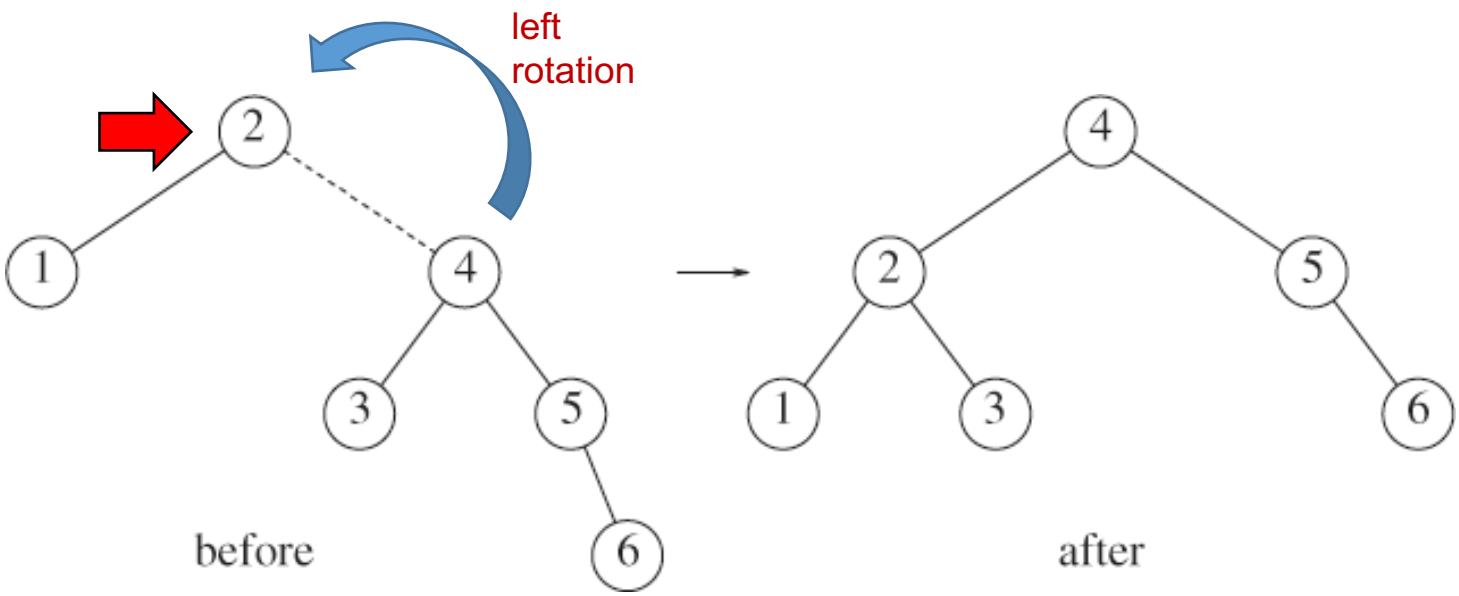
Insertion of 1 results in an unbalance tree



Insertion of 5 creates a violation at node 3 that is fixed by a single rotation



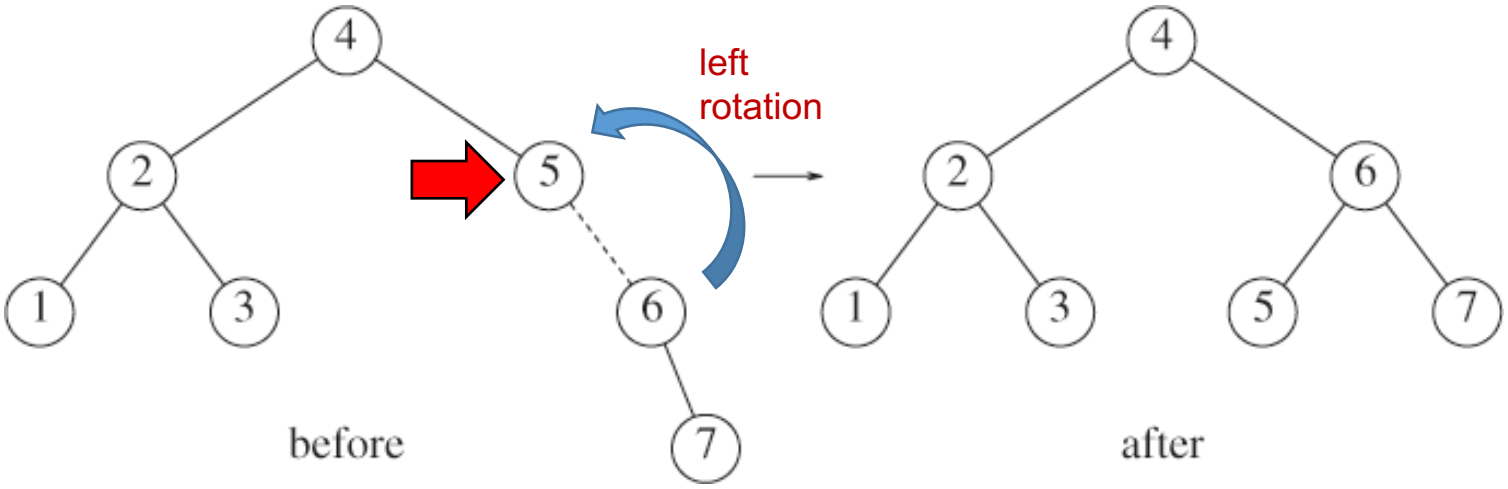
AVL Tree



- Insertion of 6 results in an unbalance tree
- This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2
- Therefore, we perform a single rotation at the root between 2 and 4

- The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2

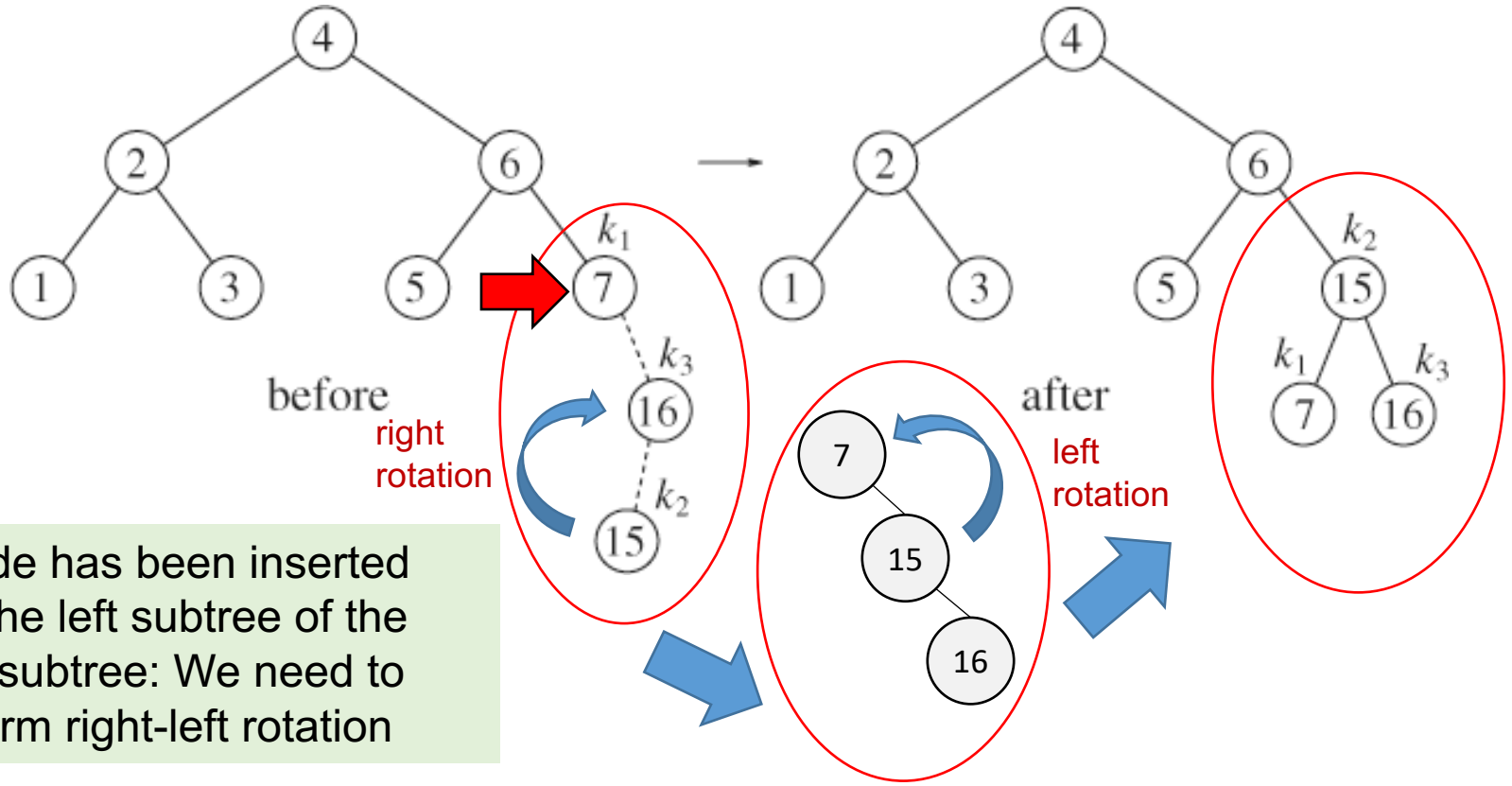
AVL Tree



- Insertion of 7 results in an unbalance tree

AVL Tree

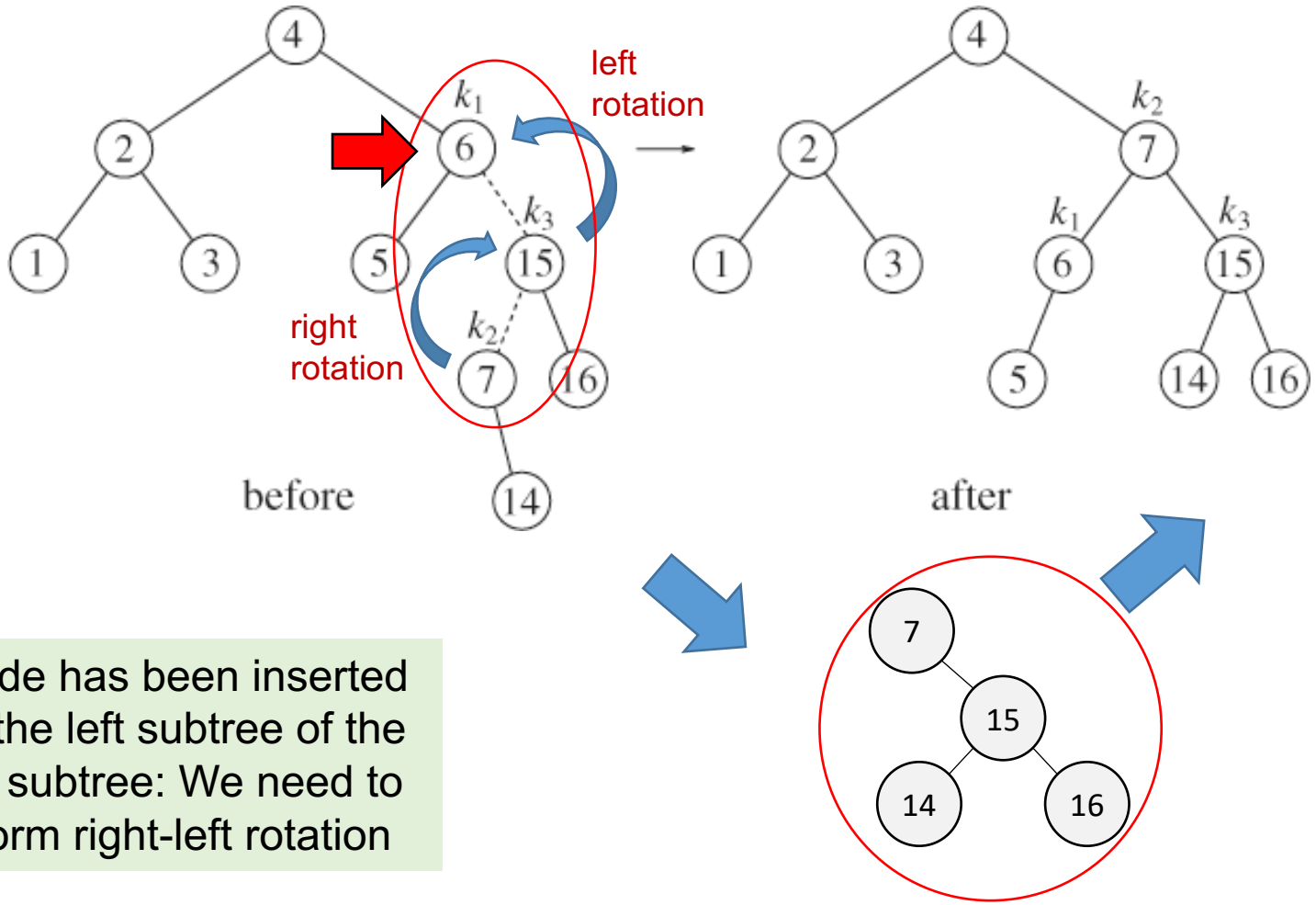
- Insertion of 16, and then 15, which results in an unbalance tree



- A node has been inserted into the left subtree of the right subtree: We need to perform right-left rotation

AVL Tree

- Inserting 14 results in an unbalanced tree



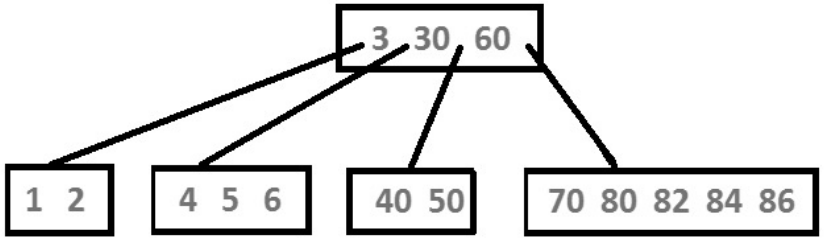
- A node has been inserted into the left subtree of the right subtree: We need to perform right-left rotation

B-Tree

The B-Tree Rules

- Similar with binary search trees, the implementation requires the ability to compare two entries via a less-than operator
- A B-tree is not a binary search tree:
  - **B-tree is not even a binary tree because B-tree nodes have many more than two children**
- **Each node contains more than just a single entry**

- **The rules for a B-tree:**
  - Make it easy to search for a specified entry
  - Ensure that leaves do not become too deep

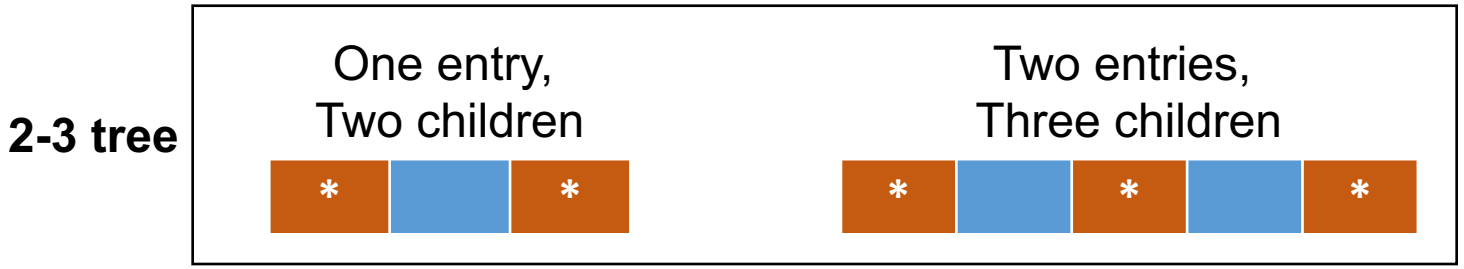


The B-Tree Rules

- Every B-tree depends on a positive constant integer called **MINIMUM**
- **B-tree Rule 1: The root may have as few as one entry** (or even no entries if it also has no children); **every other node** has at least MINIMUM entries
- **B-tree Rule 2:** The maximum number of entries in a node is twice the value of MINIMUM (MAXIMUM = 2 * MINIMUM)
- MINIMUM + 1 is the **minimum degree** or **branching factor/order** of the tree
- The **highest branching order** of a B-Tree is 2*MINIMUM+1

The B-Tree Rules

- Note: If MINIMUM = 1, we have a **2-3 tree**: two entries per node, and at most three children
  - In other words, **a B-tree of order 3 is a 2-3 tree**

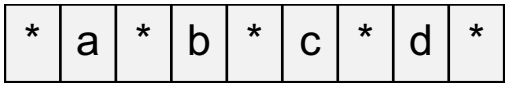


The B-Tree Rules

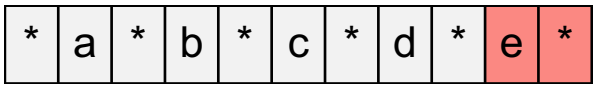
- The factor of 2 will guarantee that nodes can be split or combined
  - For example: If an internal node has 2xMINIMUM keys, then adding a key to that node can be accomplished by splitting the hypothetical 2*MINIMUM + 1 key node into two MINUMUM key nodes and moving the key that would have been in the middle to the parent node

❑ Example

MINIMUM = 2
MAXIMUM = 4

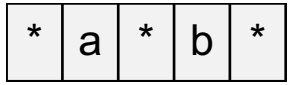


Inserting an element into this node results in MAXIMUM + 1 elements



Split into two nodes each containing MINIMUM elements

Move entry c to the parent



The B-Tree Rules

- **B-tree Rule 3:** The entries of each B-tree node are stored in a partially filled array, sorted from the smallest entry (at index 0) to the largest entry (at the final used position of the array)

- **B-tree Rule 4:** The number of subtrees below a non-leaf node **is always** one more than the number of entries in the node

Note that the above figure is implemented as two arrays: One for entries (keys), and one for pointers to subtrees

The B-Tree Rules

The followings are required to perform insertion and deletion on B-Trees:

- **The number of entries in a B-Tree can be temporarily more than MAXIMUM**
- Therefore, the entries of a B-Tree are stored as

```
Item data[MAXIMUM+1];
```

- **The number of children of a node can be temporarily more than MAXIMUM+1**
- Therefore, the pointers to children is stored as

```
set *subset[MAXIMUM+2];
```

The B-Tree Rules

❑ **Example:** Suppose MINIMUM is 200 for a B-tree

- What is the maximum number of entries that a node may contain?

- 400, although during an insertion we may temporarily have a node with 401 entries

- What is the maximum number of children that a node may have?

- 401, although during an insertion, we may temporarily have a node with 402 children

- What is the minimum number of entries that a non-root node may have?

- 200, although during a removal we may temporarily have a node with 199 entries

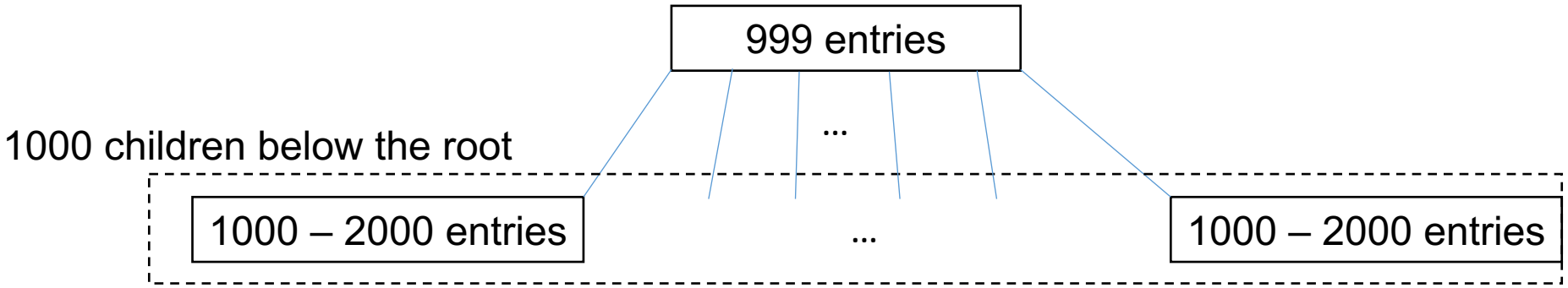
- What is the minimum number of children that a non-leaf, non-root node may have?

- 201, although during a removal we may temporarily have a node with 200 children

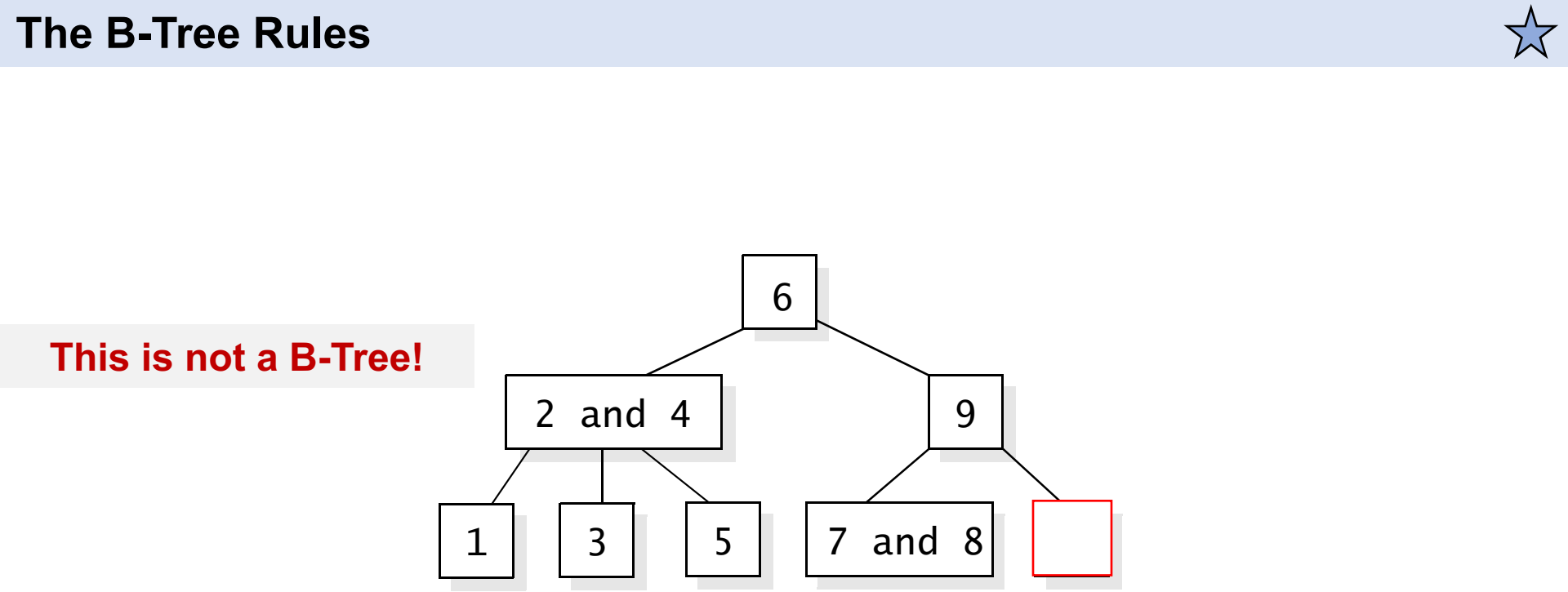
The B-Tree Rules

- **Example:** Suppose MINIMUM is 1000 for a B-tree. The tree has a root and one level of 1000 nodes (children) below that

- The root has 999 entries, and each of the 1000 nodes at the next level has between 1000 and 2000 entries
- The total number of entries in the tree is between 1,000,999 and 2,000,999 entries



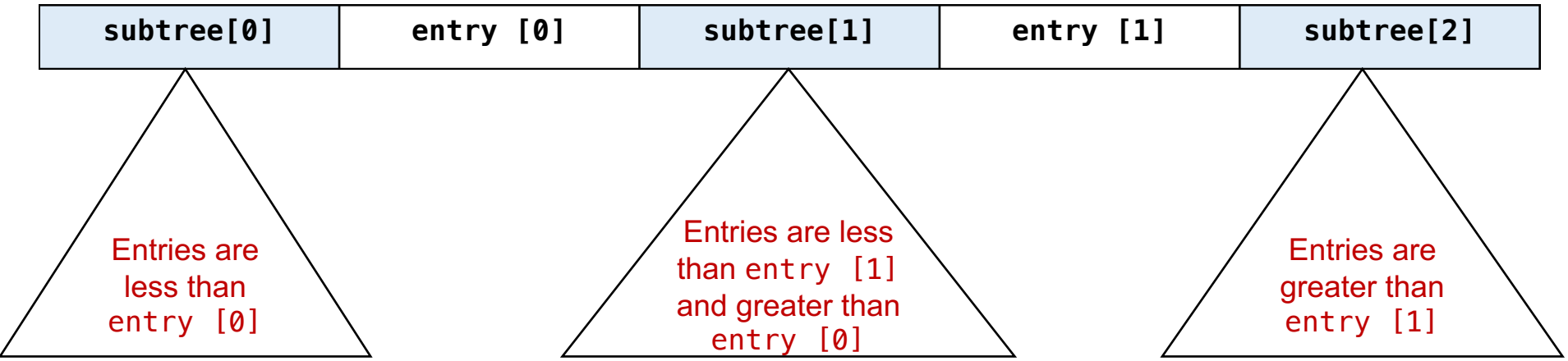
The B-Tree Rules



Violates Rule 4:
The number of subtrees below a non-leaf node is always one more than the number of entries in the node

The B-Tree Rules

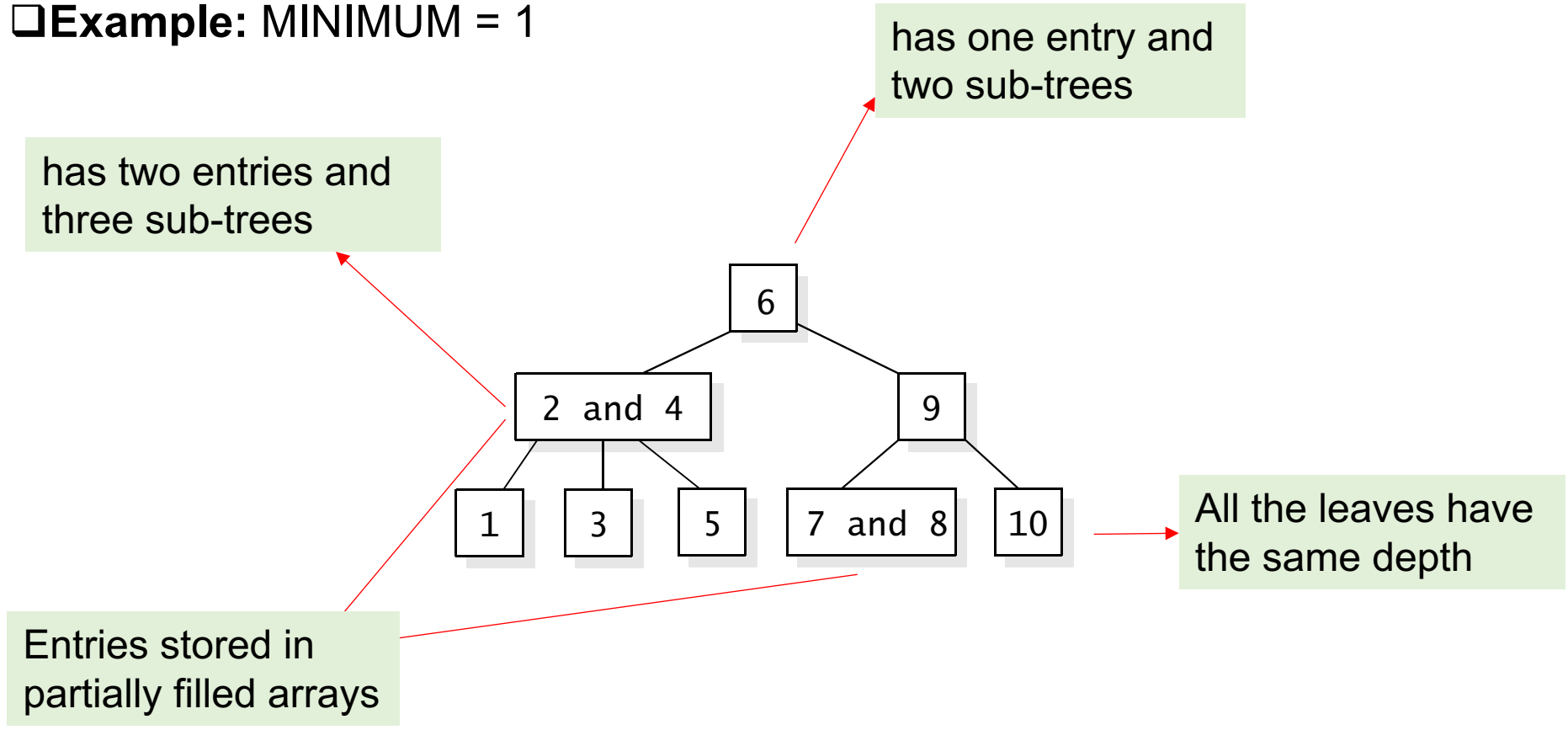
- The entries in each subtree are organized in a way that makes it easy to search the B-tree for any given entry
- **B-tree Rule 5:** For any non-leaf node: (a) An entry at index *i* is greater than all the entries in subtree number *i* of the node, and (b) an entry at index *i* is less than all the entries in subtree number *i* + 1 of the node



The B-Tree Rules

- **B-tree Rule 6:** Every leaf in a B-tree has **the same depth**

❑**Example:** MINIMUM = 1



Comparison

B-Tree vs Binary Search Tree:

- Each node in a B-tree may contain more than one entry (key)
- A binary search tree has one entry per node

- The number of children for a non-leaf node in a B-Tree is exactly one more than the number of entries
- In a binary search tree, each node has at most two children

- When searching for an item in a binary search tree, the searcher must always choose between going left or right at each node
- In a B-tree, the searcher must choose among more than just two possible children

The Set ADT with B-Tree

```
#ifndef SCU_COEN79_SET_H
#define SCU_COEN79_SET_H
#include <cstdlib>   // Provides size_t
namespace scu_coen79_11
{
    template <class Item>
    class set
    {

    public:
        // TYPEDEFS
        typedef Item value_type;

        // CONSTRUCTORS and DESTRUCTOR
        set( );
        set(const set& source);
        ~set( ) { clear( ); }

        || MODIFICATION MEMBER FUNCTIONS
        ...
        || CONSTANT MEMBER FUNCTIONS
        ...
```

The Set ADT with B-Tree

```
    private:
        // MEMBER CONSTANTS
        static const std::size_t MINIMUM = 200;
        static const std::size_t MAXIMUM = 2 * MINIMUM;

        // MEMBER VARIABLES
        std::size_t data_count;
        Item data[MAXIMUM+1];

        std::size_t child_count;
        set *subset[MAXIMUM+2];
};
```

Member variables store information about the root of the B-tree
We normally need MAXIMUM fields, but we need one extra field

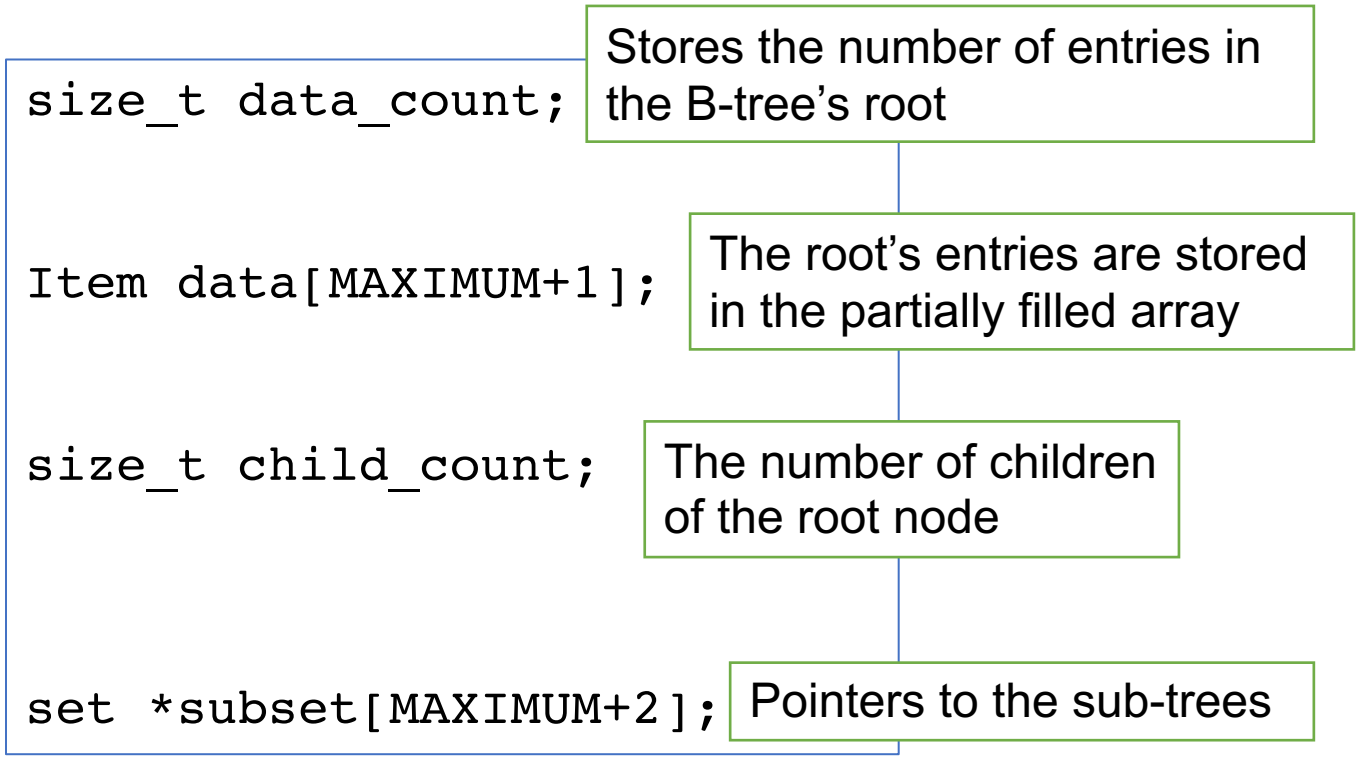
We store pointers to the sub B-Trees
We normally need MAXIMUM + 1 fields, but we need one extra field

subtree[0]	entry [0]	…	entry [MAXIMUM + 1]	Subtree [MAXIMUM + 2]

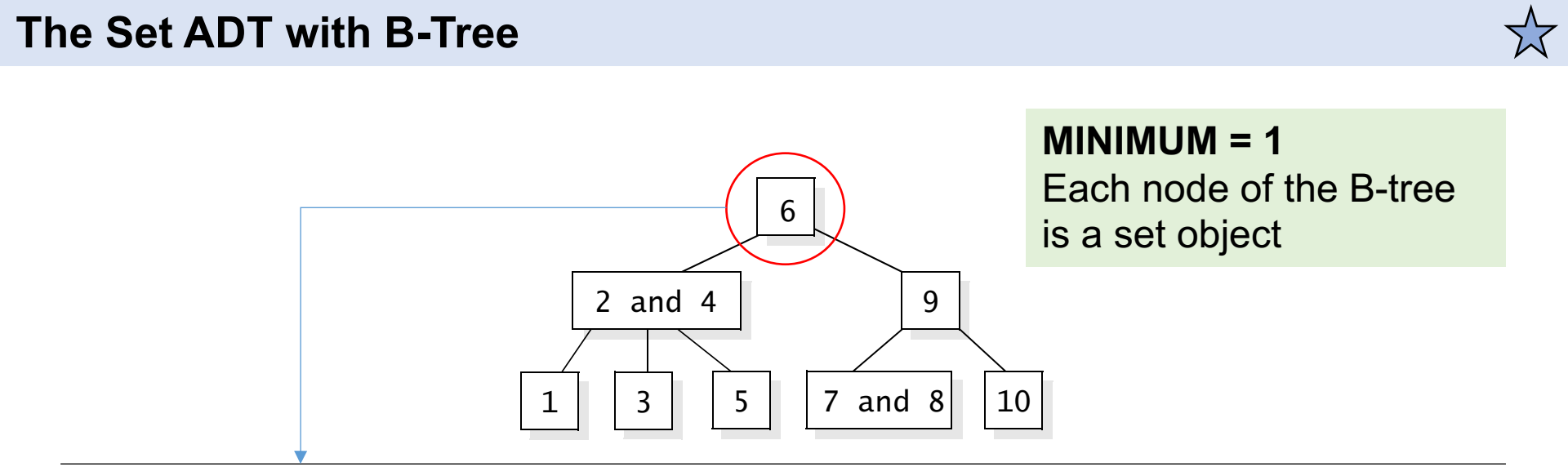
The Set ADT with B-Tree

NOTE: Every child of the root is also the root of a smaller B-tree

member variables store information about the root of the B-tree



The Set ADT with B-Tree

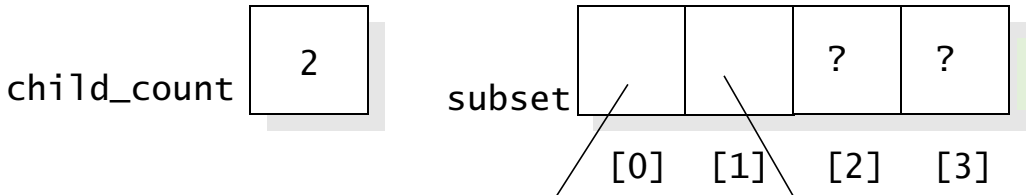


MINIMUM = 1
Each node of the B-tree is a set object

number of entries in the B-tree's root


```
NOTE:
MINIMUM = 1
MAXIMUM = MINIMUM *2
data[MAXIMUM+1];
```

number of children of the root




```
*subset[MAXIMUM+2];
```

Points to a smaller set object that contains 1, 2, 3, 4, and 5

Points to a smaller set object that contains 7, 8, 9, and 10

The Set ADT with B-Tree

Invariant for the Set Class Implemented with a B-Tree

1. The items of the set are stored in a B-tree, satisfying the six B-tree rules
2. The number of entries in the tree's root is stored in the member variable `data_count`, and the number of sub-trees of the root is stored in the member variable `child_count`
3. The root's entries are stored in `data[0]` through `data[data_count-1]`
4. If the root has subtrees, then these subtrees are stored in sets pointed to by the pointers `subset[0]` through `subset[child_count-1]`

Searching for an item in the B-Tree

Three cases:

- **The target does appear in the root**: the search is done
- **The target is not in the root, and the root has no children:** the search is done
- **The target is not in the root, but there are subtrees below:** there is only one possible subtree where the target can appear, so the function makes a recursive call

Searching for an Item in the B-Tree (Cont'd)

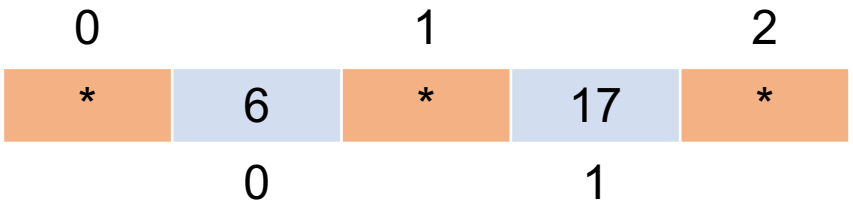
Pseudocode for searching

1. Make a local variable, `i`, equal to the first index such that `data[i]` is not less than the target. If there is no such index, then set `i` equal to `data_count`, indicating that all of the entries are less than the target.

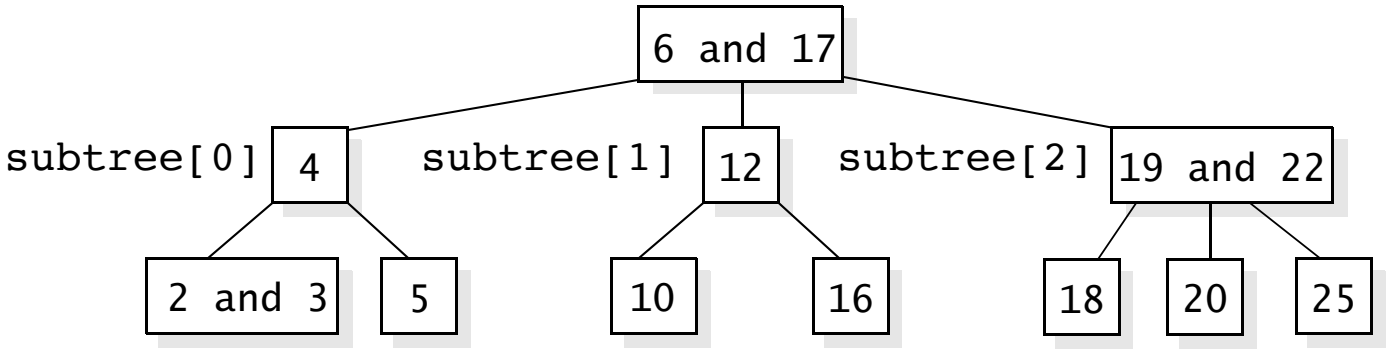
2. *if* (we found the target at `data[i]`)
   return 1;
   else if (the root has no children)
   return 0;
   else
   return `subset[i]->count(target);`

Searching for an Item in the B-Tree (Cont'd)

❑ Assume we are searching for 10



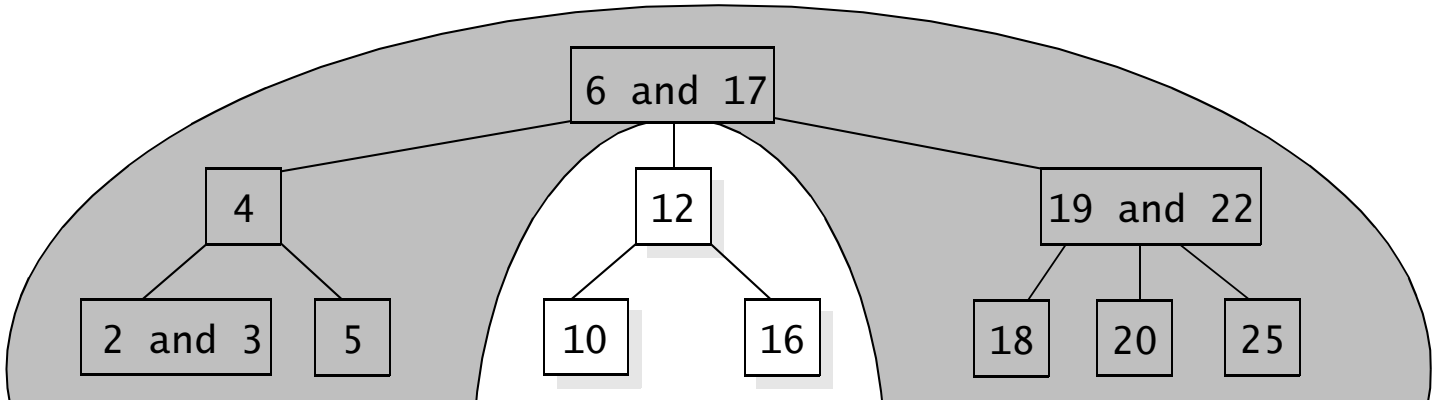
- 17 is the first item in the root that is not less than the target (10)
- Occurs at `data[1]`
- Step 1 sets `i = 1`



- We have not found the target at `data[1]`
- But the root does have children
- We make a recursive call: `subtree[1]->count(10)`

Searching for an Item in the B-Tree (Cont'd)

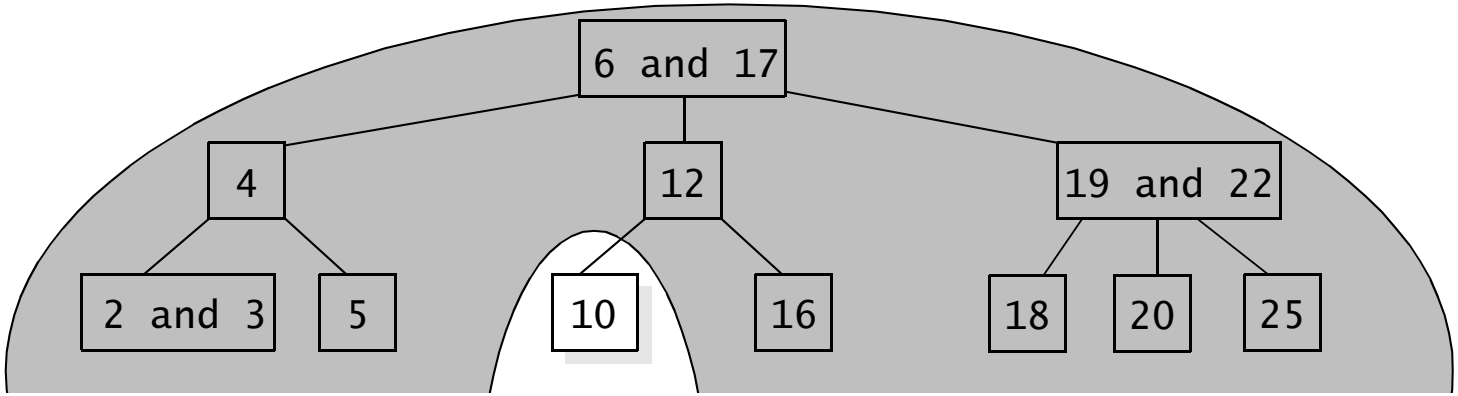
❑ **Assume we are searching for 10 (Cont'd)**



- The recursive call of the `count` member function has its own local variable `i`
- This variable `i` is set to 0 in Step 1 of the recursive call (since `data[0]` of the subtree is greater than or equal to the target 10)
- We make a recursive call: `subtree[0]->count(10)`

Searching for an Item in the B-Tree (Cont'd)

❑ **Assume we are searching for 10 (Cont'd)**

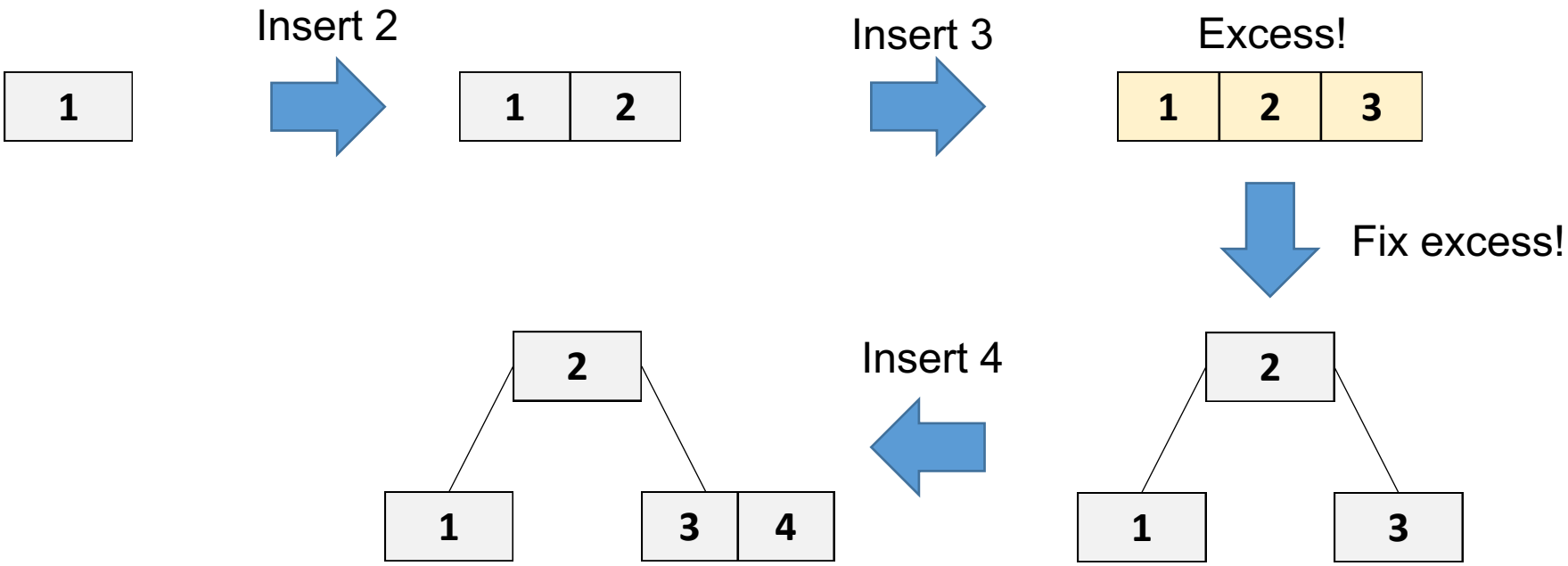


- `data[0]` is not less than the target
- We found the target at `data[0]`
- `Return 1`

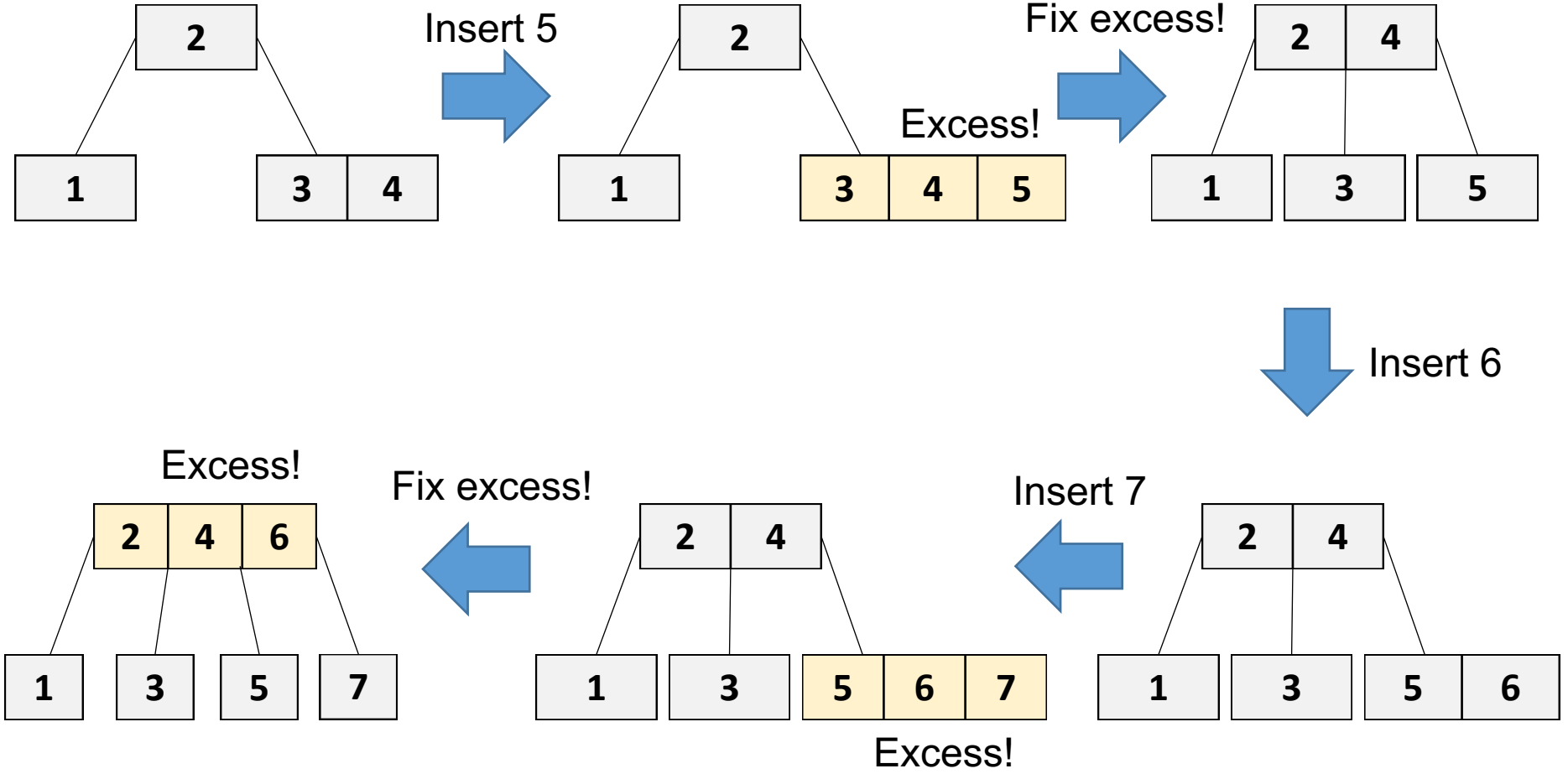
Inserting an Item into the B-Tree

❑ **Example: Inserting items into a B-Tree**

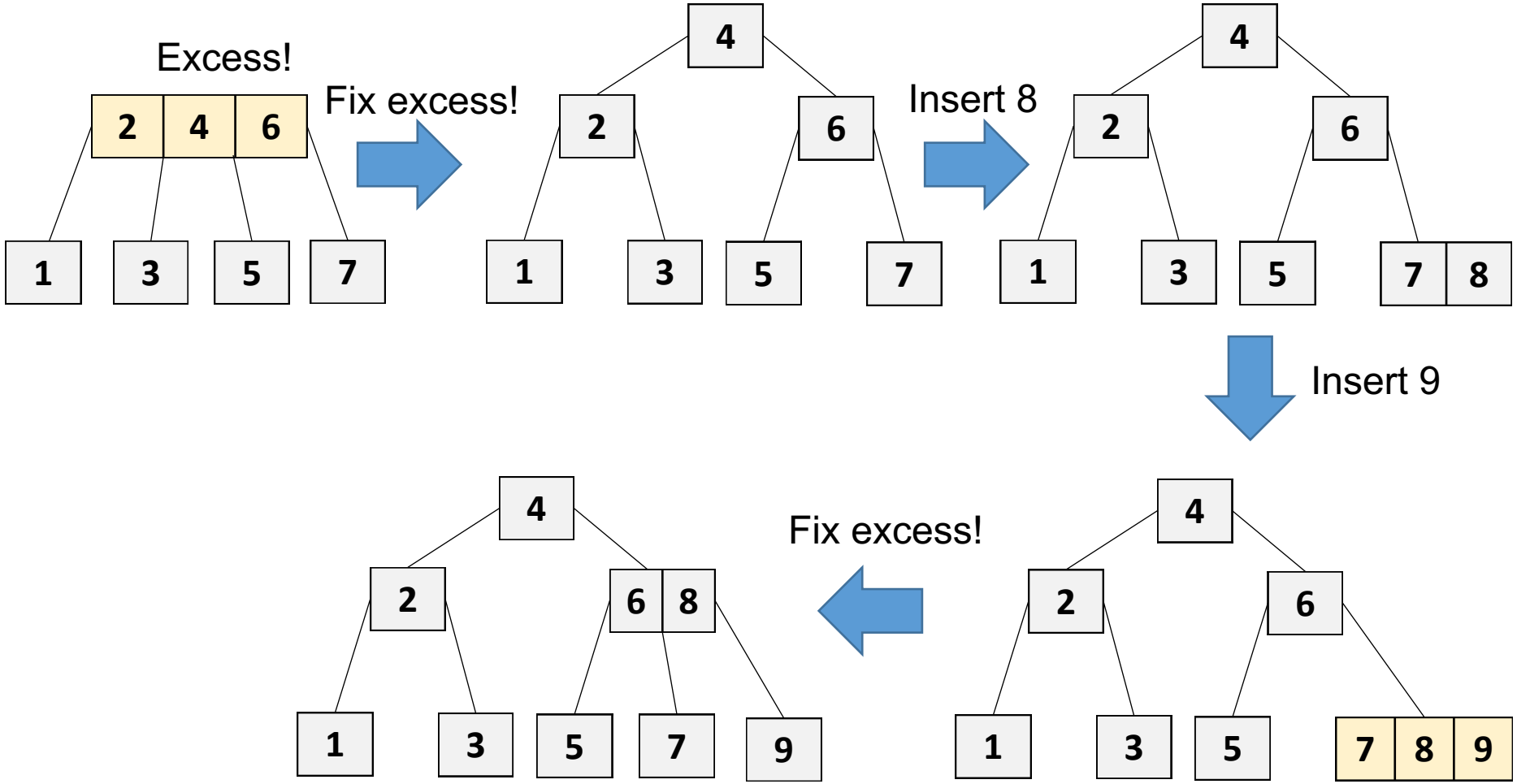
- Start with an empty B-tree, with **MINIMUM = 1**
  - Each node can contain MAXIMUM = 2 entries
  - Nodes can temporarily contain 3 entries (results in excess)
- Enter the integers 1 through 9



Inserting an Item into the B-Tree



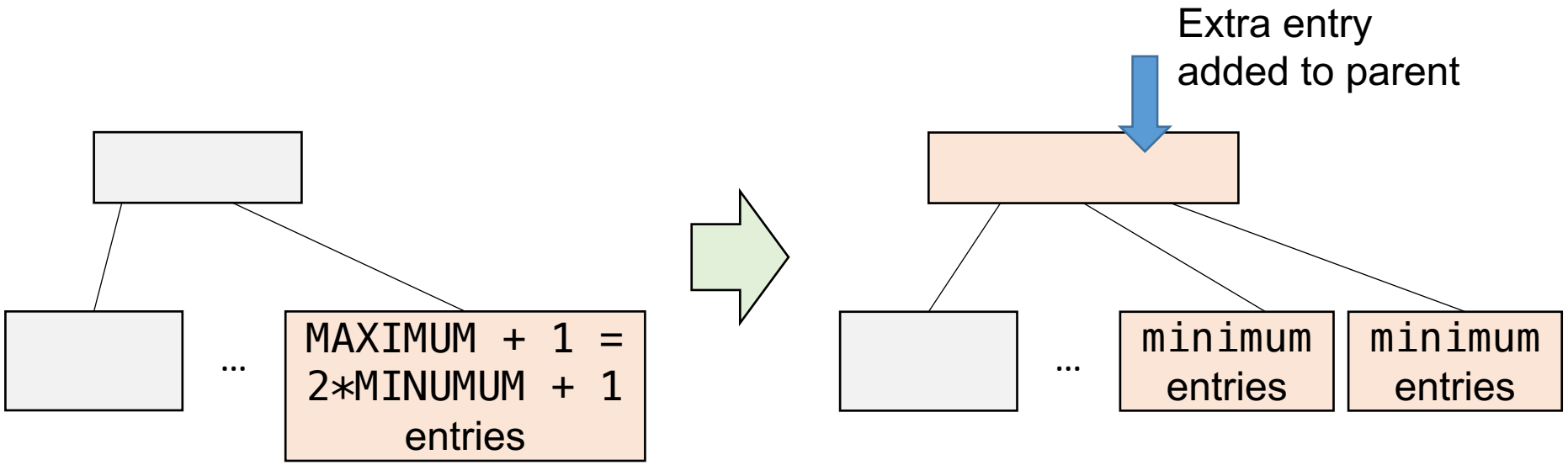
Inserting an Item into the B-Tree



Inserting an Item into the B-Tree (Cont'd)

Fixing a Child with an Excess Entry

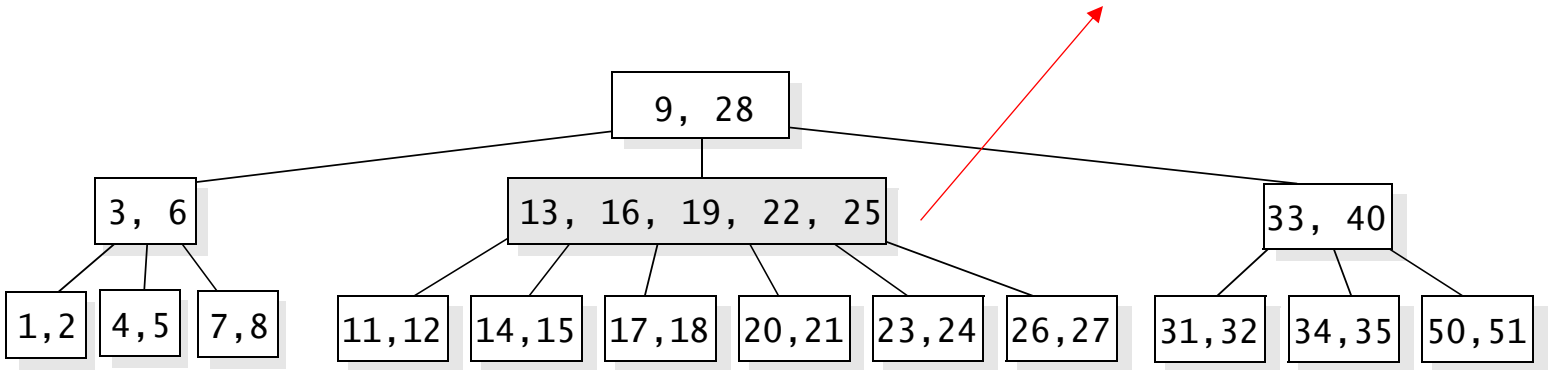
- To fix a child with MAXIMUM + 1 entries, the child node is split into two nodes that each contain MINIMUM entries
- This leaves one extra entry, which is passed upward to the parent



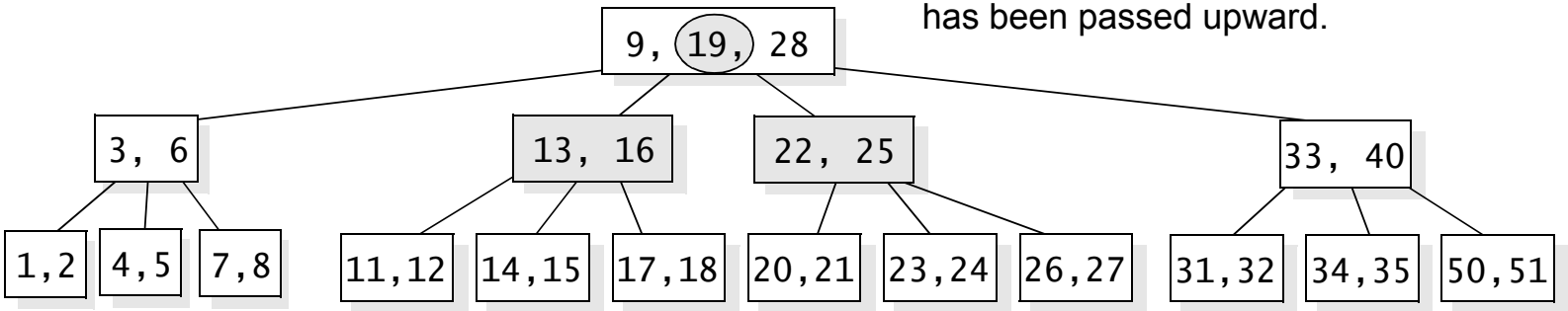
Inserting an Item into the B-Tree (Cont'd)

- **Another example**
  - MINUMUM = 2
  - MAXIMUM = 4

- The problem node (with 2*MINIMUM + 1 entries)
- Split into two smaller nodes with MINIMUM entries each



The full node has been split into two nodes, and the middle entry (19) has been passed upward.



Removing an Item from a B-Tree (Cont'd)

How to remove an item?

1. Make a local variable, `i`, equal to the first index such that `data[i]` is not less than target. If there is no such index, then set `i` equal to `data_count` (all of the entries are less than the target)

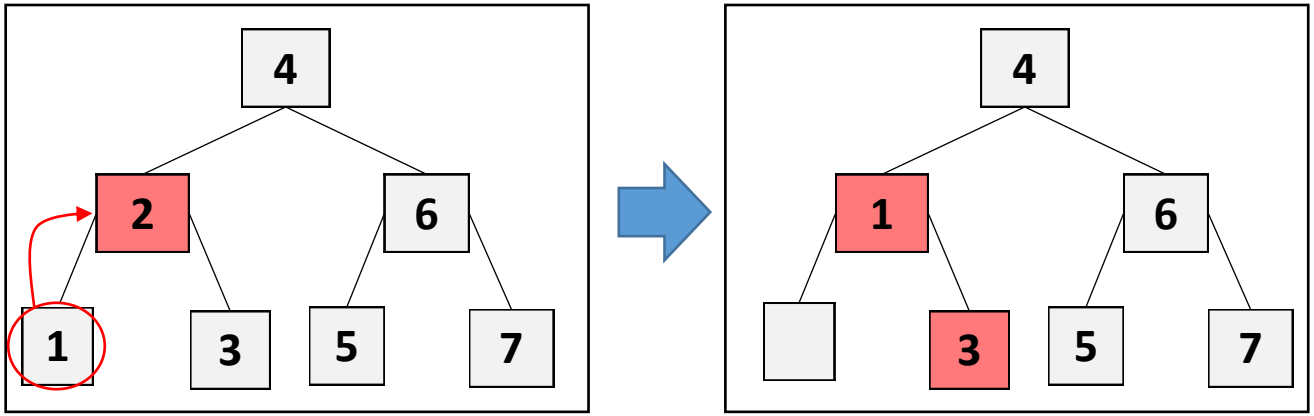
2. Deal with one of these four possibilities:
   - **2a.The root has no children, and we did not find the target:** There is no work to do (return `false`)
   - **2b.The root has no children, and we found the target**: In this case, remove the target from the data array, and return `true`
   - **2c.The root has children, and we did not find the target:** We did not find the target in the root, but the target still might appear in `subset[i]`
   - **2d.** (see next slide)

Removing an Item from a B-Tree (Cont'd)

2d.The root has children, and we found the target

- We cannot simply remove it from the data array because there are children below, and this may violate the following B-Tree rule:
- **Reminder: B-tree Rule 4:** The number of subtrees below a non-leaf node is always one more than the number of entries in the node
- **Solution**: We will go into `subset[i]` and remove the *largest* item in this subset. We will take a copy of this largest item and place it into `data[i]`

❑ Assume we want to **remove 2** from the following B-Tree

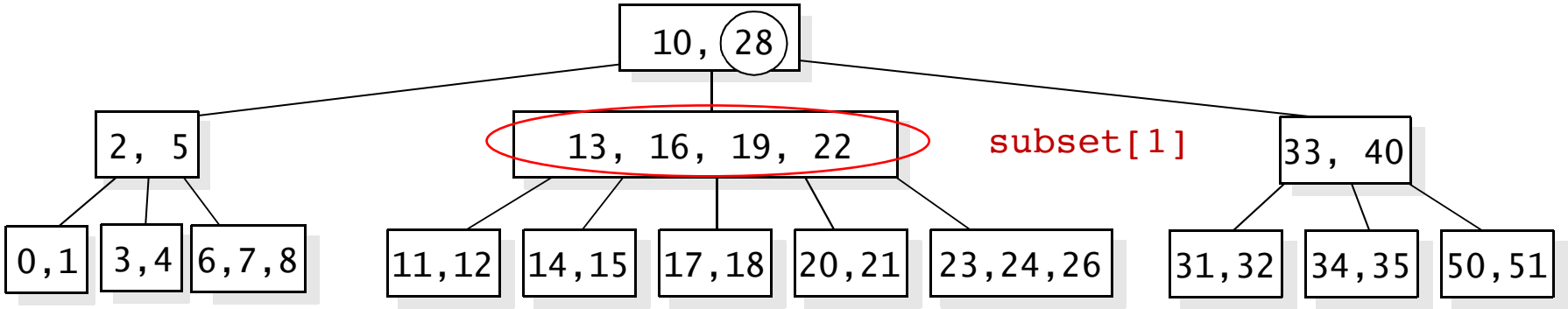


Removing an Item from a B-Tree (Cont'd)

❑ **Example:**

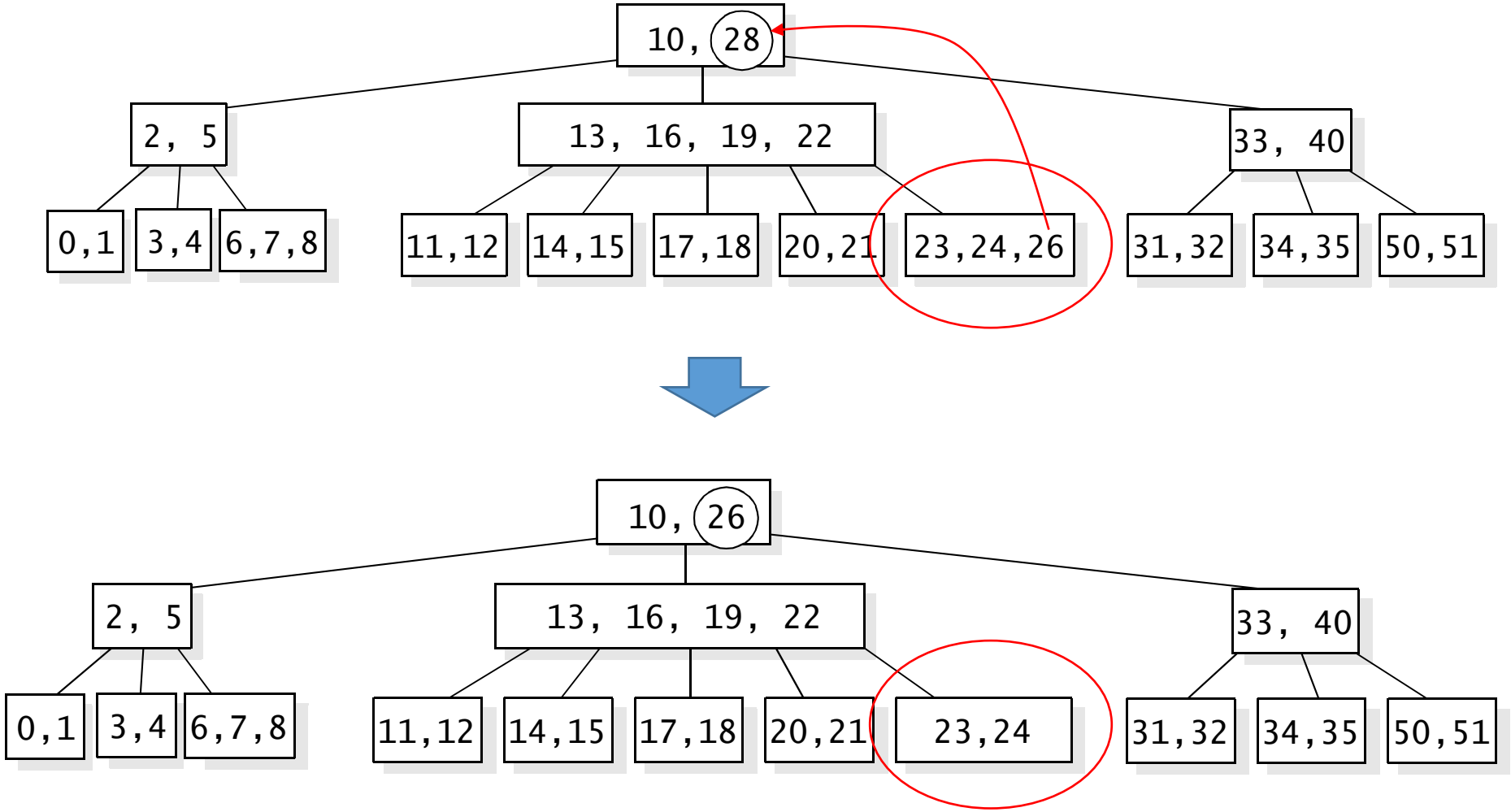
We want to delete the target 28, which is at `data[1]`

- The root has children and we found the target
- `i = 1` is the index of the first item that is not less than 28



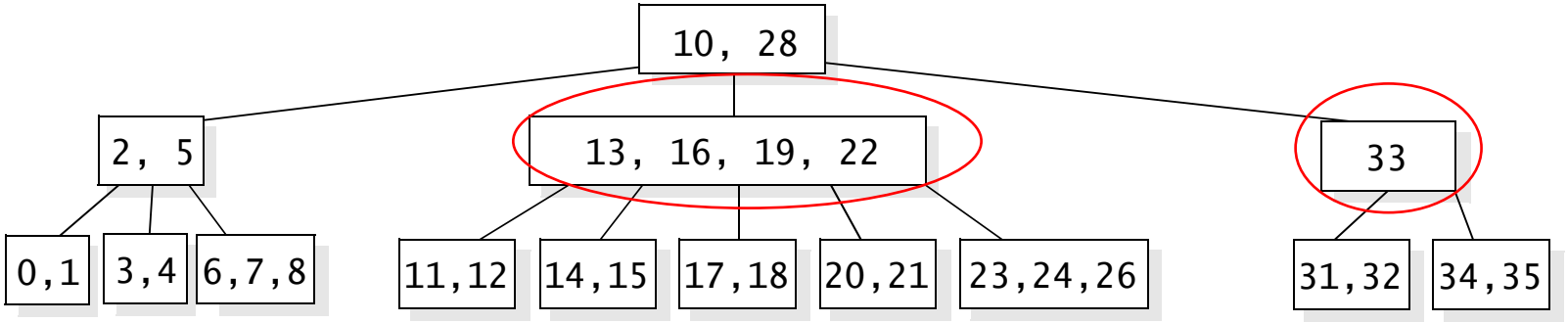
Our plan is to go into `subset[1]`, remove the largest item (the 26), and place a copy of this 26 on top of the target

Removing an Item from a B-Tree (Cont'd)



Removing an Item from a B-Tree (Cont'd)

- **MINIMUM = 2**
- **Assume we end up with the following tree:**
  - `subset[2]` needs to be fixed as it has less than 2 entries



Removing an Item from a B-Tree (Cont'd)

- When a set has only `MINIMUM - 1` entries
- How can we correct this problem?

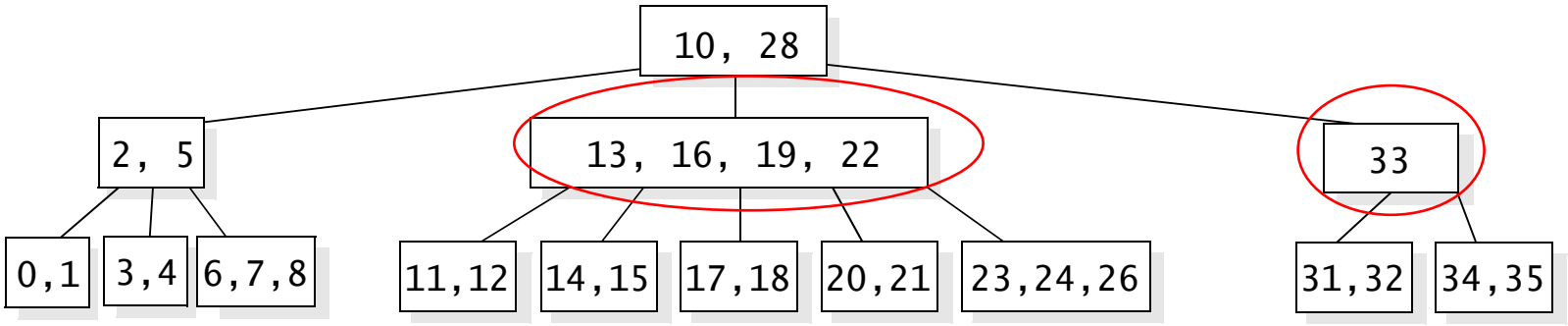
We consider four cases:

- **Case 1**
  - Transfer an extra entry from `subset[i-1]`
- **Case 2**
  - Transfer an extra entry from `subset[i+1]`
- **Case 3**
  - Combine `subset[i]` with `subset[i-1]`
- **Case 4**
  - Combine `subset[i]` with `subset[i+1]`

Removing an Item from a B-Tree (Cont'd)

Case 1: Transfer an extra entry from `subset[i-1]`

Suppose that `subset[i-1]` has more than the minimum number of entries



- **Assume MINIMUM = 2**
- `subset[2]` needs to be fixed
- `subset[1]` has more than 2 entries

Removing an Item from a B-Tree (Cont'd)

a) **Transfer `data[i-1]` down to the front of `subset[i]->data`**
Remember to shift over the existing entries to make room, and add one to `subset[i]->data_count`

b) **Transfer the final item of `subset[i-1]->data` up to replace `data[i-1],`** and subtract one from `subset[i-1]->data_count`

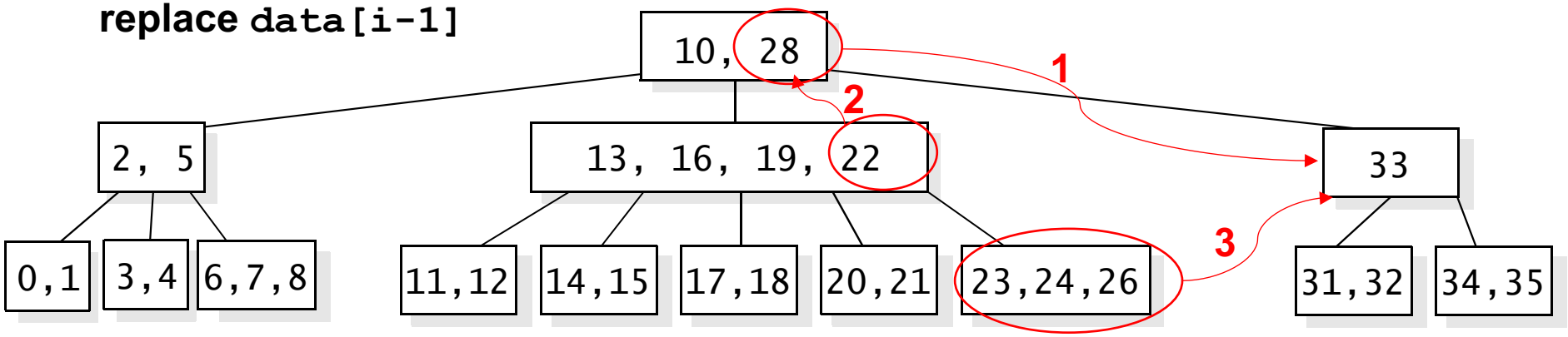
c) **If `subset[i-1]` has children, transfer the final child of subset[i-1] over to the front of `subset[i]`**
- This involves shifting over the existing array `subset[i]->subset` to make room for the new child pointer at `subset[i]->subset[0]`
- Also add one to `subset[i]->child_count`, and subtract one from `subset[i-1]->child_count`

Removing an Item from a B-Tree (Cont'd)

- `subtree[1]` donates an entry to `subtree[2]`
- Note that we cannot directly transfer an item from `subtree[1]` to `subtree[2]`

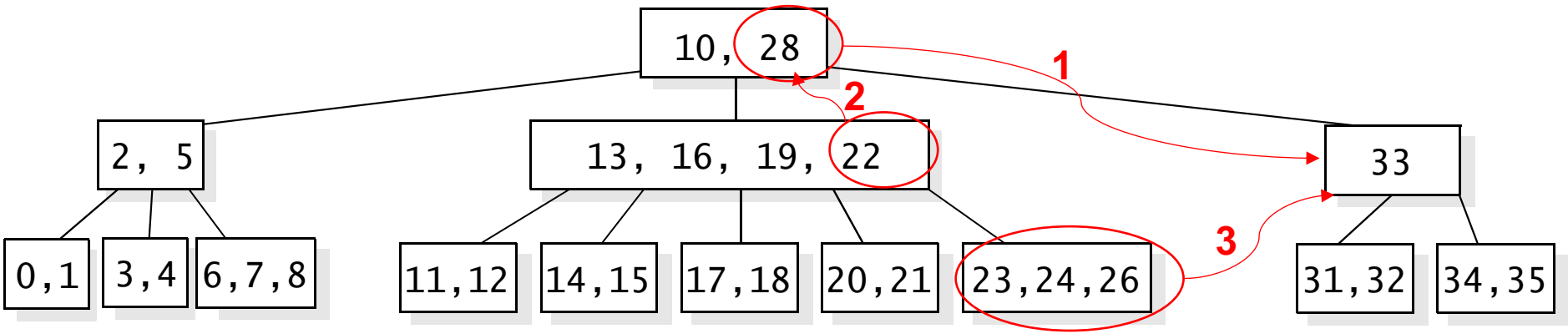
2: Transfer the final item of `subset[i-1]->data` up to replace `data[i-1]`

1: Transfer `data[i-1]` down to the front of `subset[i]->data`



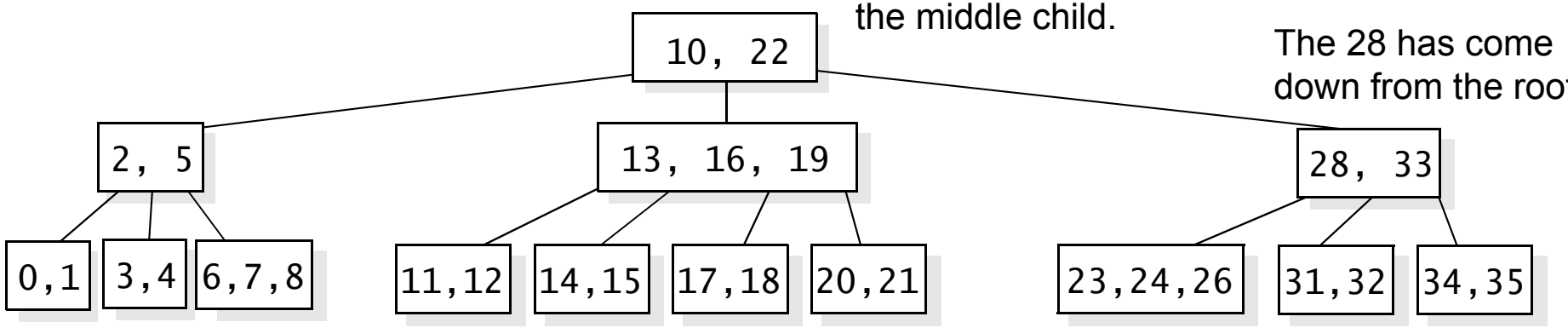
3: If `subset[i-1]` has children, transfer the final child of `subset[i-1]` over to the front of `subset[i]`

Removing an Item from a B-Tree (Cont'd)



This 22 has come up from the middle child.

The 28 has come down from the root.



This child has been moved over.

Removing an Item from a B-Tree (Cont'd)

Case 2
Transfer an extra entry from `subset[i+1]`

- Suppose that `subset[i+1]` has more than the minimum number of entries
- Similar to what you have just seen for transferring an entry from `subset[i-1]`

Removing an Item from a B-Tree (Cont'd)

Case 3

Combine `subset[i]` with `subset[i-1]`

- Suppose that `subset[i-1]` is present (i > 0)
- However, `subset[i-1]` has only `MINIMUM` entries
- **We cannot transfer an entry from `subset[i-1]`,** but **we can combine `subset[i]` with `subset[i-1]`**

Removing an Item from a B-Tree (Cont'd)

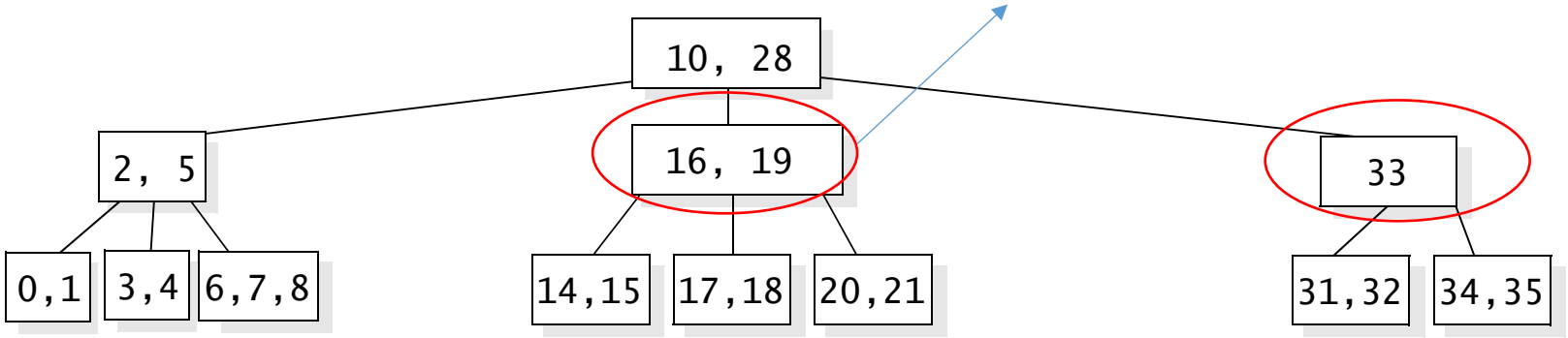
Combining `subset[i]` with `subset[i-1]` happens in 3 steps:

1. **Transfer `data[i-1]` down to the end of `subset[i-1]->data`**
   - This actually removes the item from the root, so shift `data[i]`, `data[i+1]`, and so on, leftward to fill in the gap
   - Remember to subtract one from `data_count`, and add one to `subset[i-1]->data_count`
2. **Transfer all the items and children from subset[i] to the end of `subset[i-1]`**
   - Update the values of `subset[i-1]->data_count` and `subset[i-1]->child_count`
   - Set `subset[i]->data_count` and `subset[i]->child_count` to zero
3. **Delete the node `subset[i]`, and shift `subset[i+1]`, `subset[i+2]`, and so on, leftward to fill in the gap**
   - Reduce `child_count` by one.

Removing an Item from a B-Tree (Cont'd)

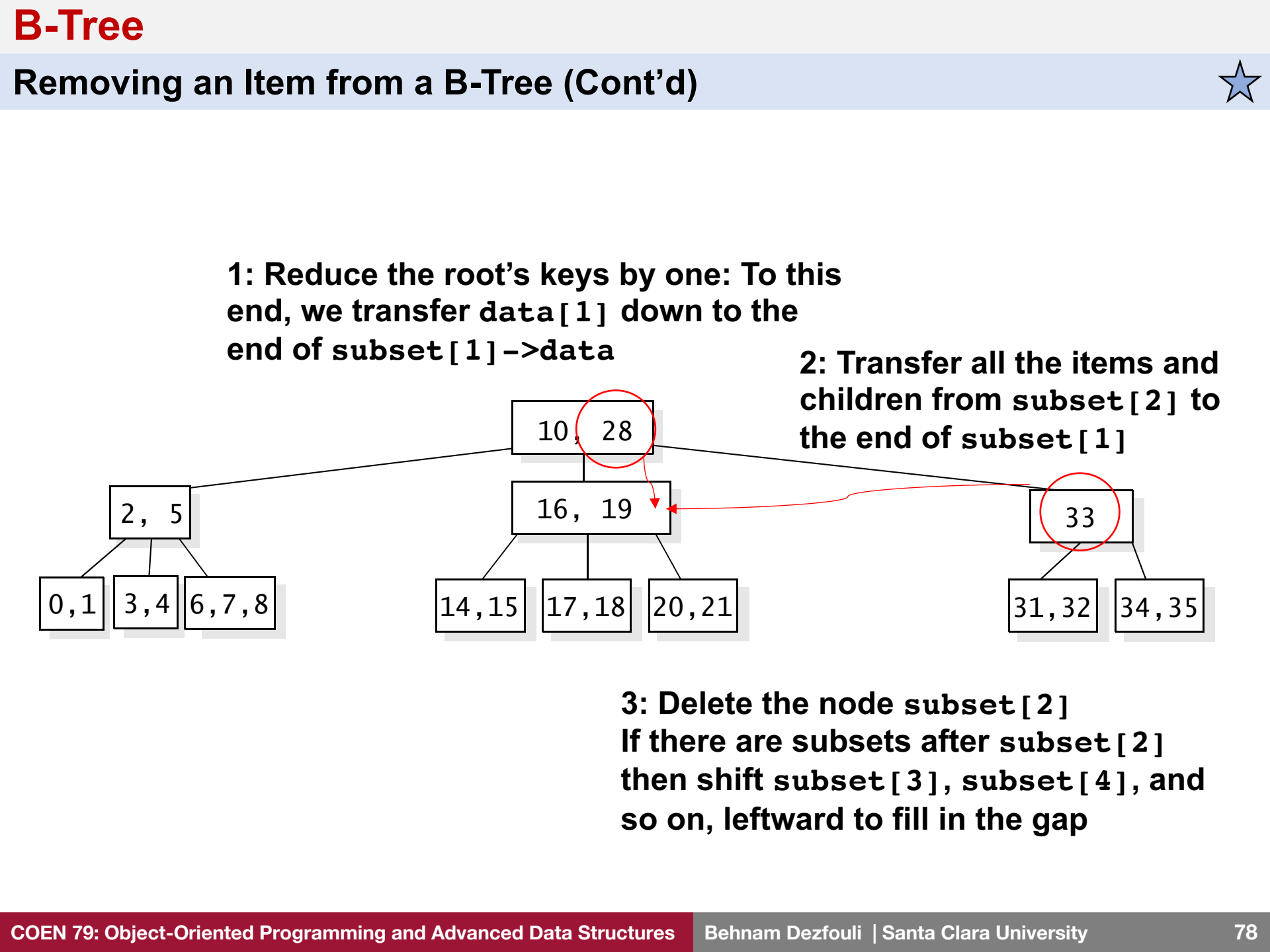
- **MINIMUM = 2**
- **Assume we end up with the following tree:**
  - `subset[2]` needs to be fixed as it has less than 2 entries

- **`subset[1]` has only MINIMUM entries**
- **It cannot donate an entry**

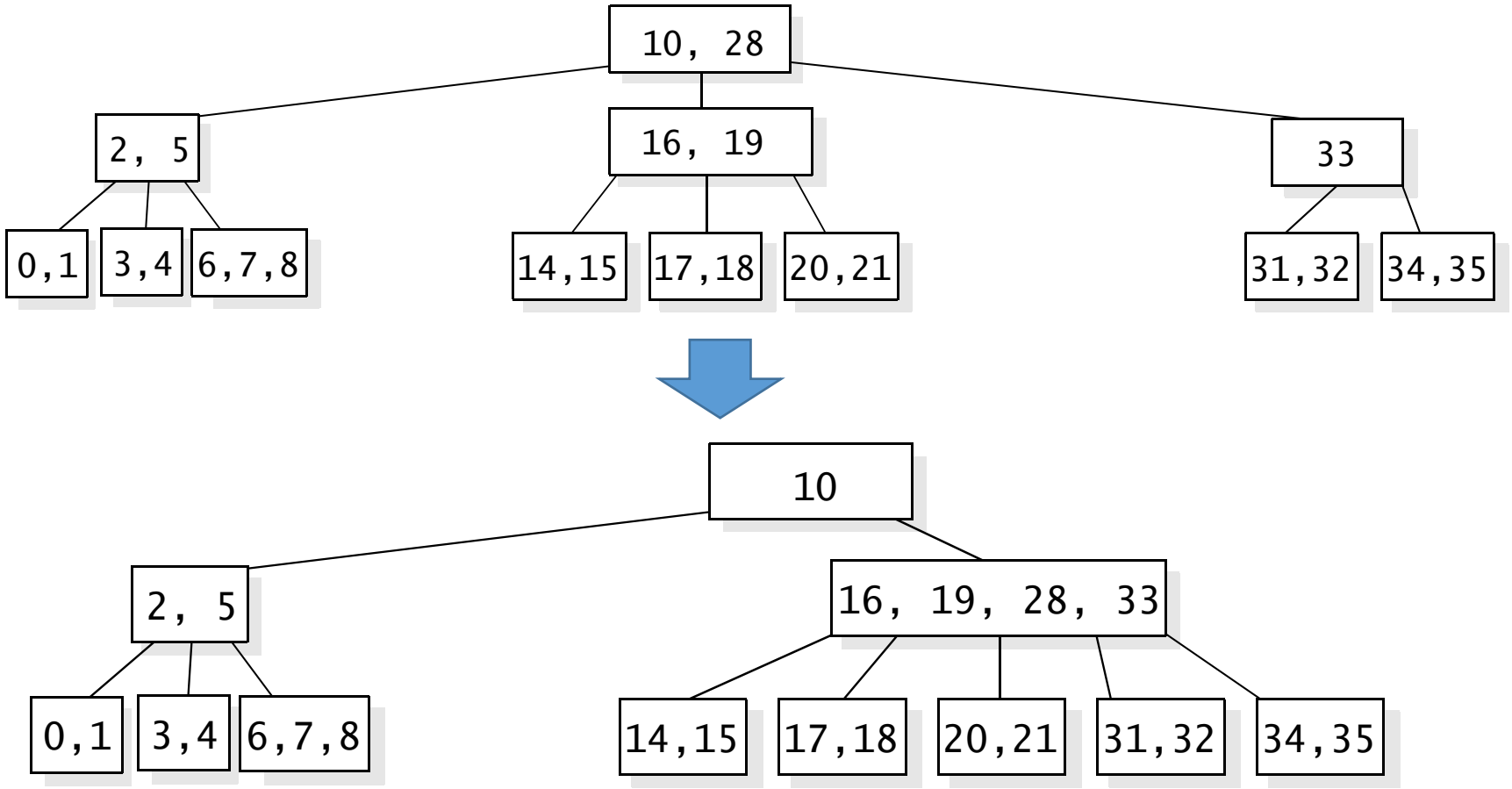


We combine `subset[1]` with `subset[2]`

Removing an Item from a B-Tree (Cont'd)

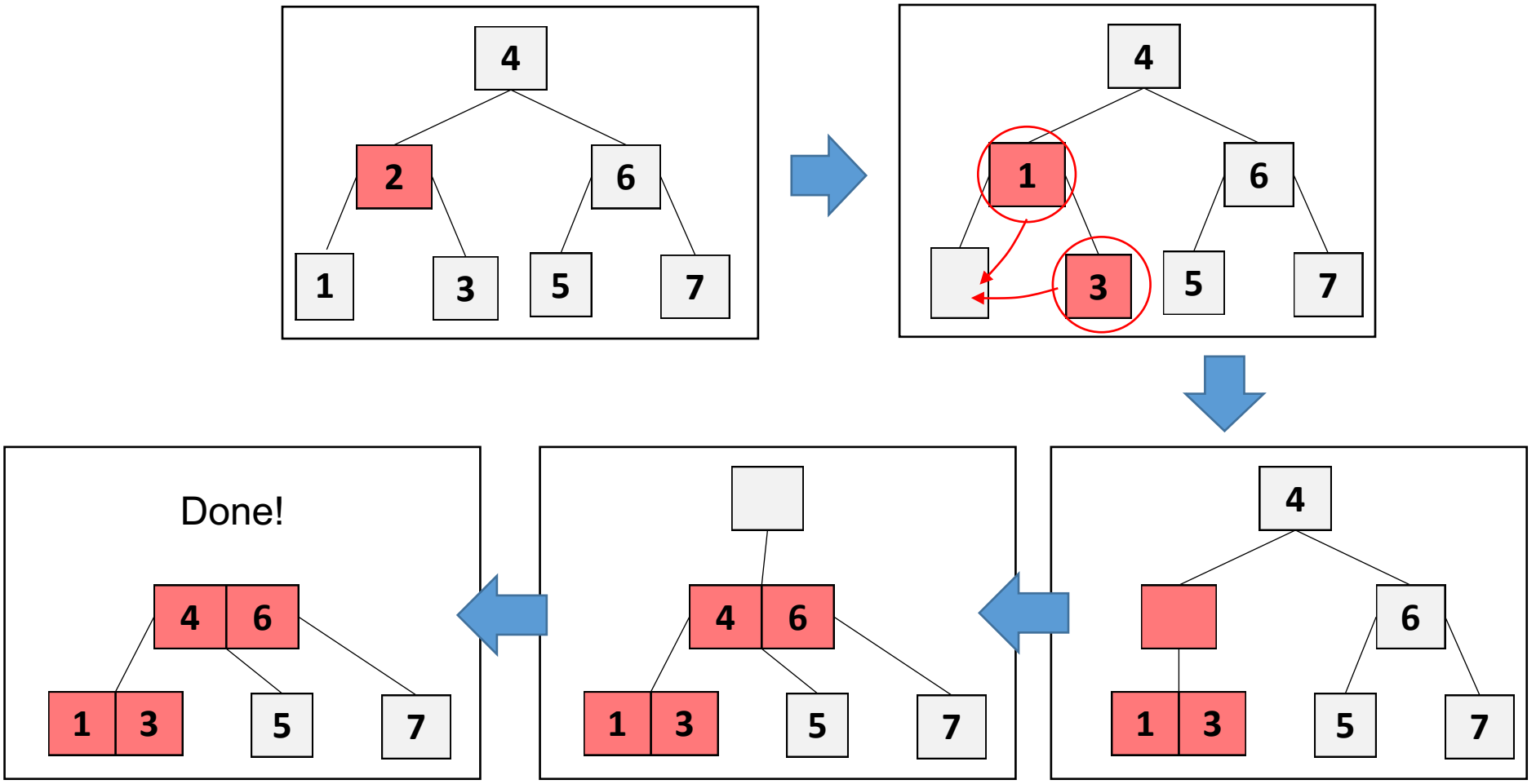


Removing an Item from a B-Tree (Cont'd)



Removing an Item from a B-Tree

- Assume we want to **remove 2** from the following tree
- MINIMUM = 1



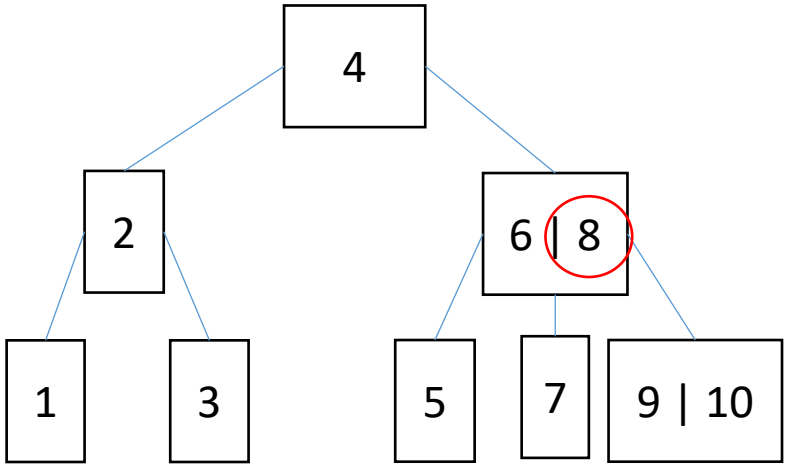
Removing an Item from a B-Tree (Cont'd)

Combining `subset[i]` with `subset[i+1]`

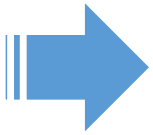
- Similar to combining `subset[i]` with `subset[i-1]`

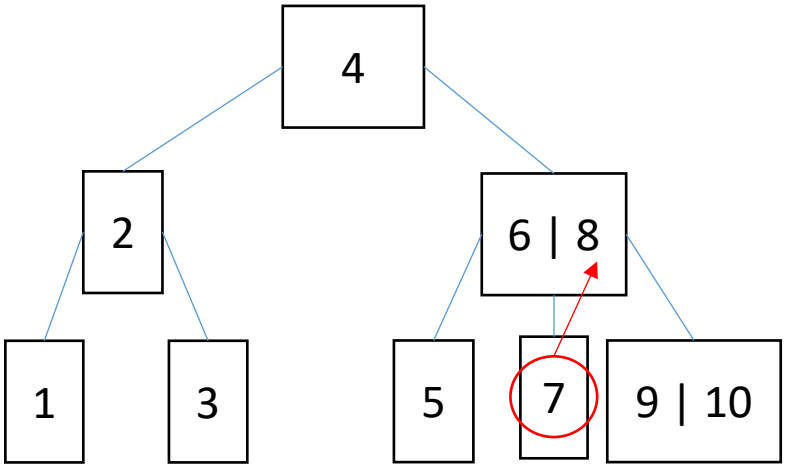
Removing an Item from a B-Tree (Cont'd)

❑ Example: Remove number 8



- MINIMUM = 1
- The root has children, and we found the target
- `i = 1`
- We cannot simply delete the item because there are children below

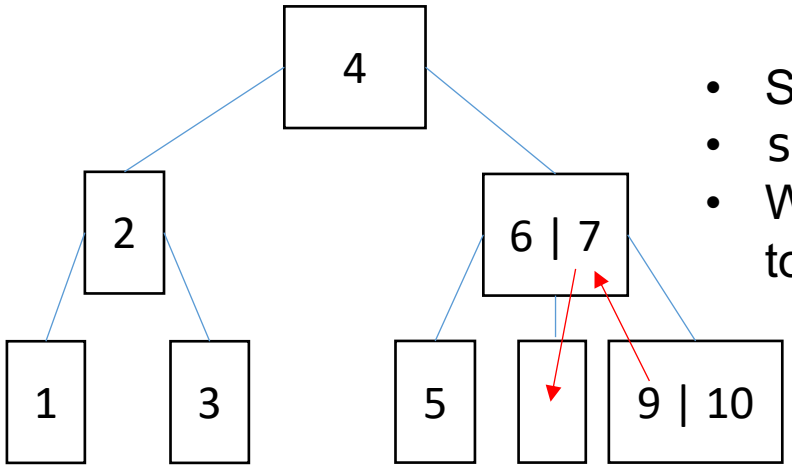




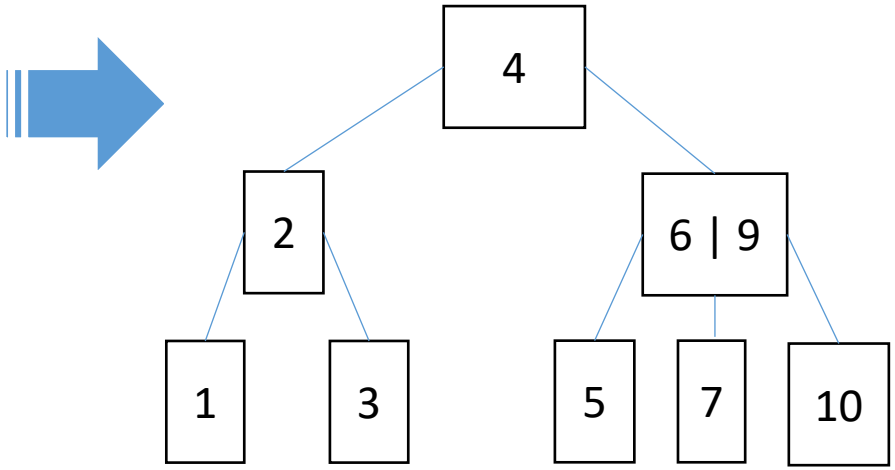
Largest item in `subset[1]` is moved to `data[1]`

Removing an Item from a B-Tree (Cont'd)

❑ Example: Remove number 8 (Cont'd)

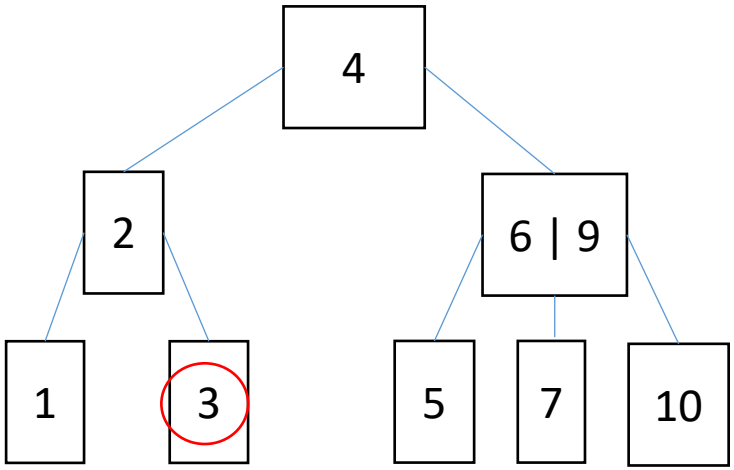


- Shortage in the root of `subset[1]`
- `subset[2]` has more than 1 item
- We transfer an item from `subset[2]` to `subset[1]`

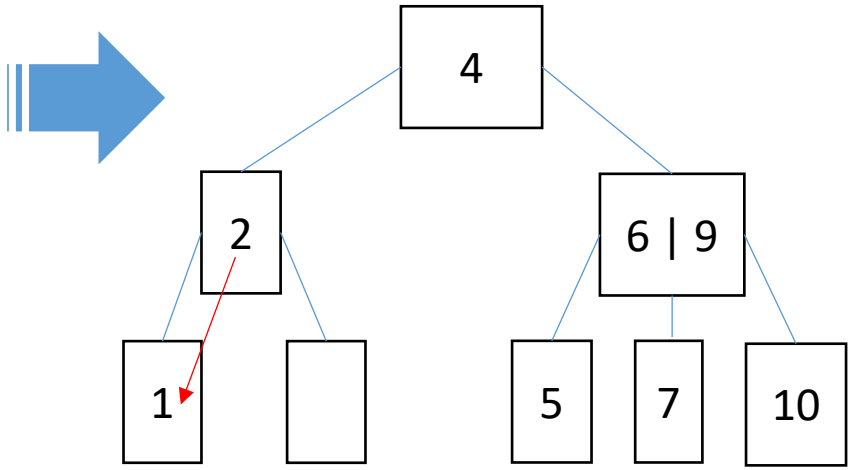


Removing an Item from a B-Tree (Cont'd)

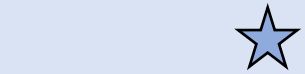
- Example: Remove number 3



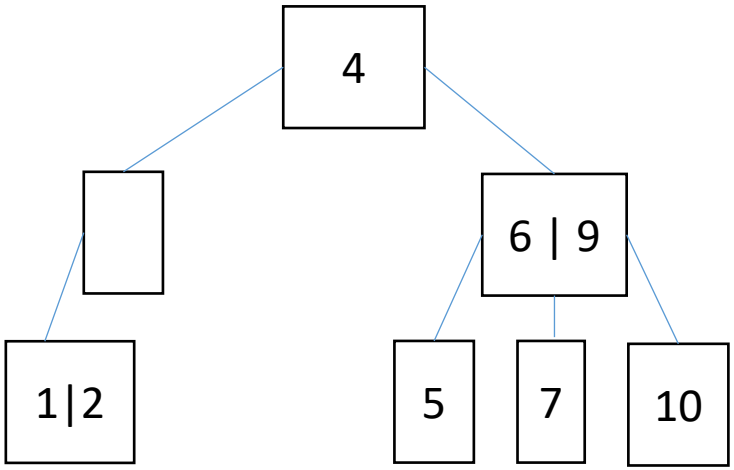
The root has no children, and we found the target



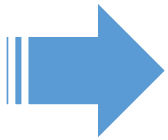
Removing an Item from a B-Tree (Cont'd)

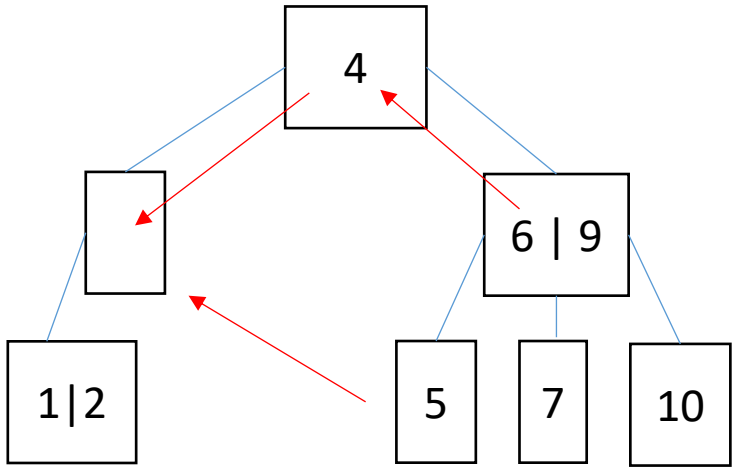


❑ Example: Remove number 3 (Cont'd)



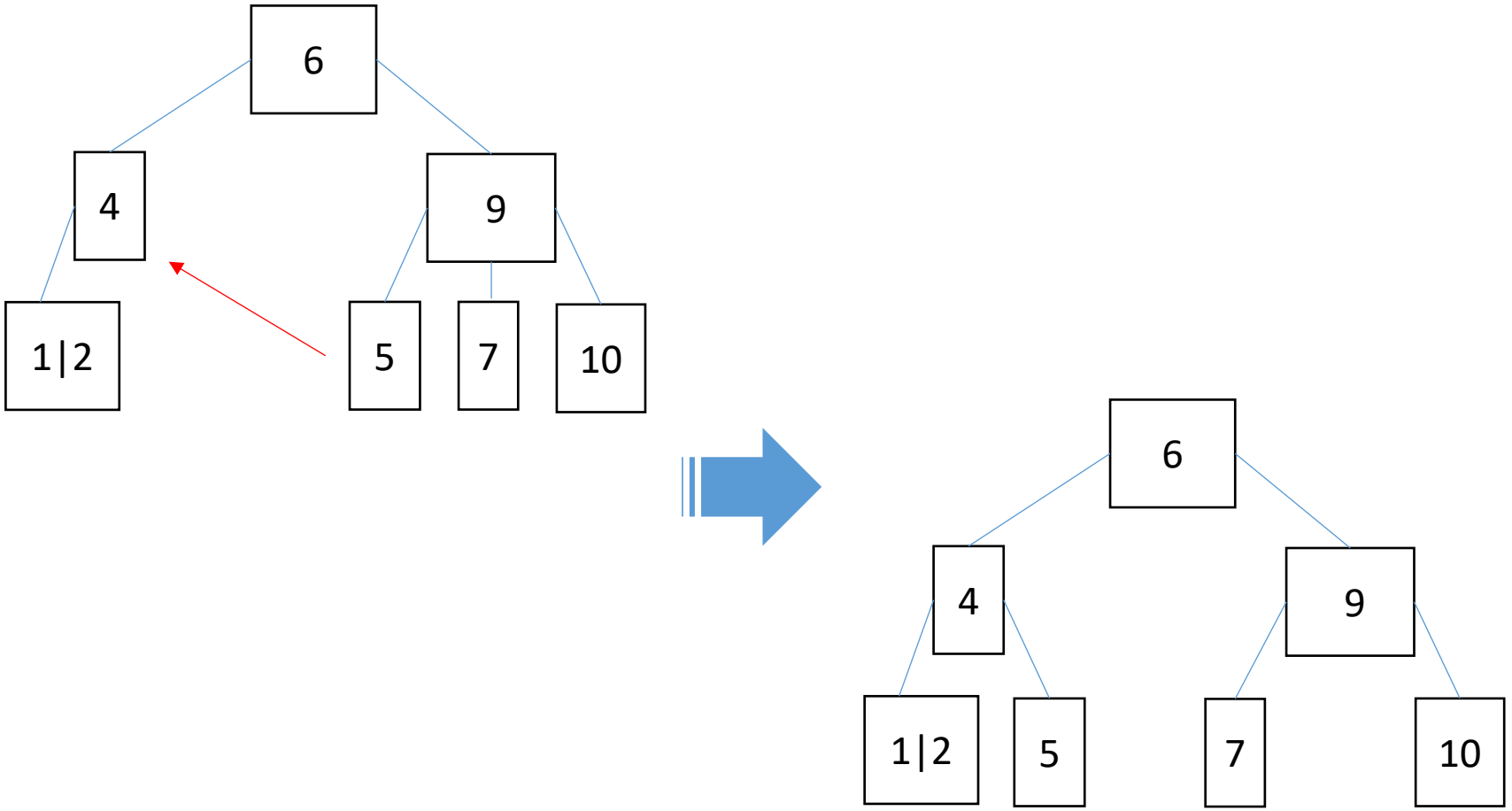
- Shortage in the root of `subset[0]`
- `subset[1]` has more than 1 item
- We transfer an item from `subset[1]` to `subset[0]`





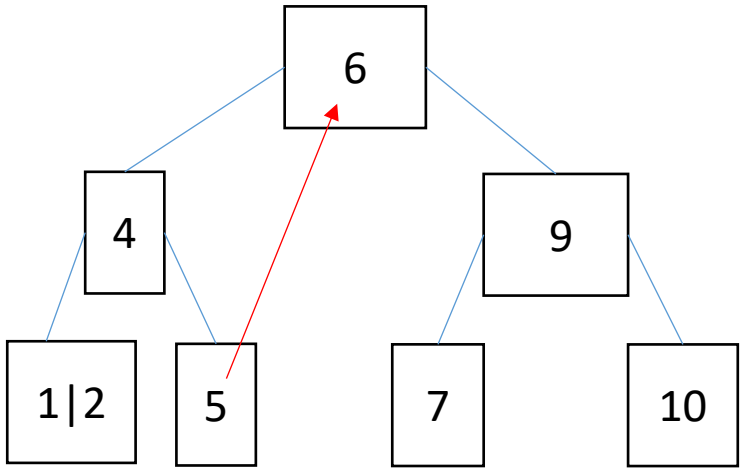
Removing an Item from a B-Tree (Cont'd)

❑ Example: Remove number 3 (Cont'd)

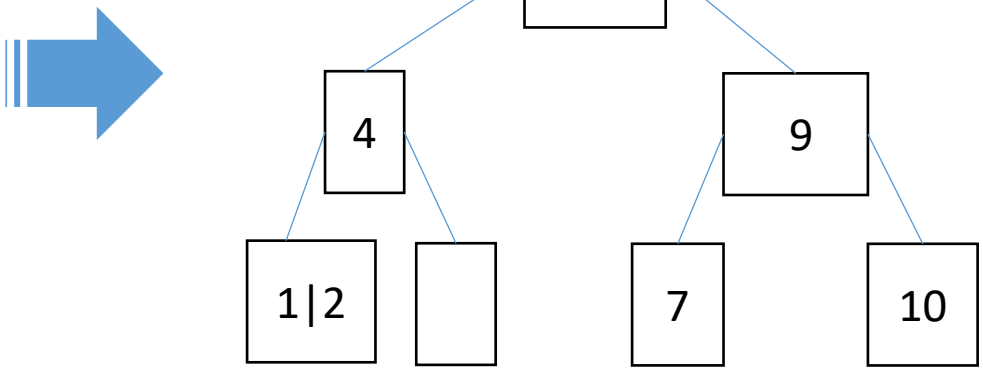


Removing an Item from a B-Tree (Cont'd)

❑ Example: Remove number 6

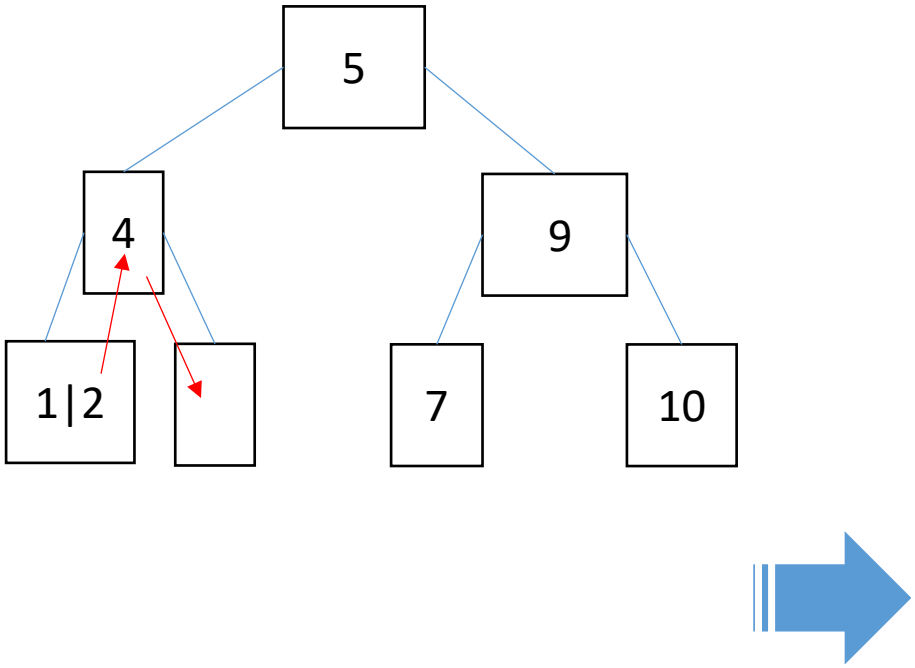


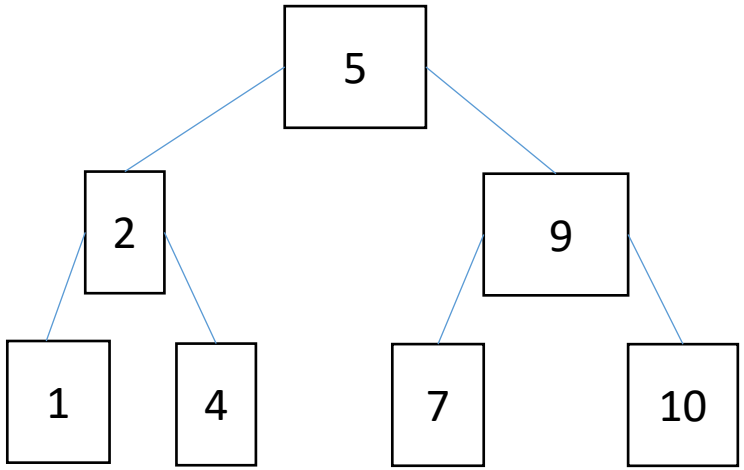
- The root has children, and we found the target
- `i` = 0
- We cannot simply delete the item because there are children below
- Largest item in `subset[0]` is moved to `data[0]`



Removing an Item from a B-Tree (Cont'd)

- Example: Remove the number 6





Trees, Logs, and Time Analysis

Trees, Logs, and Time Analysis

- Both heap and B-Tree control the depth
- **The worst-case time performance for the following operations are all *O*(*d*), where *d* is the depth of the tree:**
  - Adding an entry in a binary search tree, a heap, or a B-tree
  - Deleting an entry from a binary search tree, a heap, or a B-tree
  - Searching for a specified entry in a binary search tree or a B-tree

Time Analysis for Binary Search Tree

- A **BST** with *n* nodes has at most depth *n*-1
- Adding an entry, deleting an entry, or searching for an entry in a binary search tree with *n* entries is: *O*(*d*)
  - where *d* is the depth of the tree

Time Analysis for Heaps

- Number of entries at level *d* is 2^d
- For a **heap** with *d* levels we need at least the following number of nodes:

$$(1 + 2 + 4 + \ldots + 2^{d-1}) + 1$$

Number of nodes at level 0

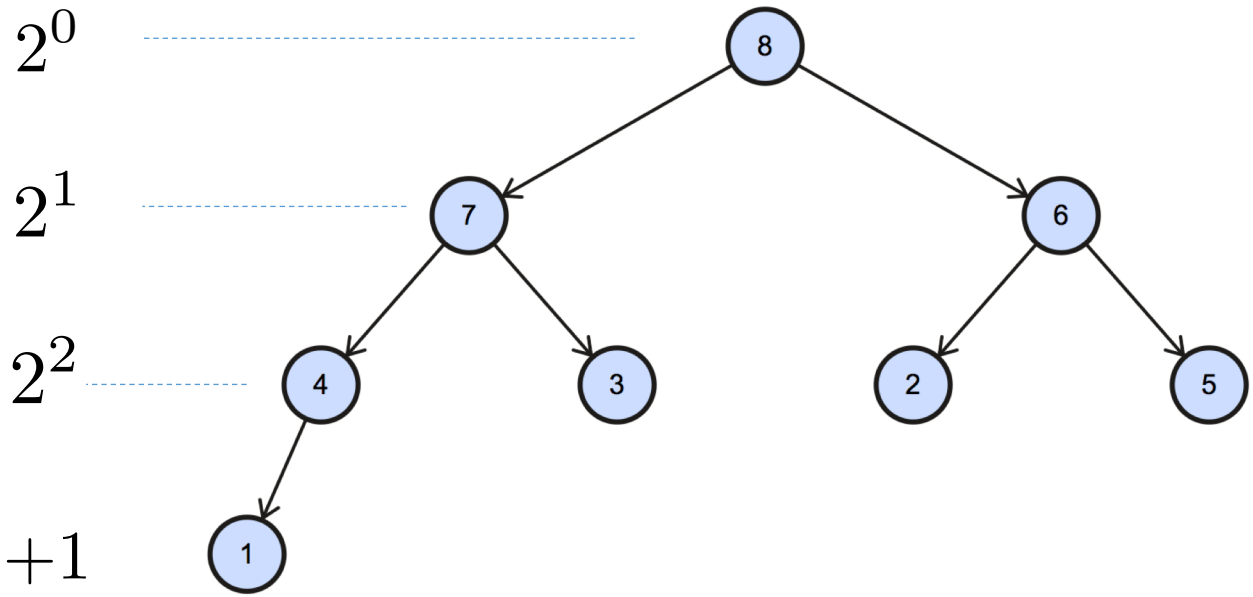
Number of nodes at level 1

$$= 1 + 1 + 2 + 4 + \ldots + 2^{d-1}$$

$$= 4 + 4 + \ldots + 2^{d-1}$$

$$= 2^{d-1} + 2^{d-1} = 2^d$$

Time Analysis for Heaps



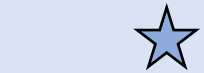
Time Analysis for Binary Search Tree

- What is the maximum depth of a heap with *n* nodes?
  - A heap with depth *d* needs at least 2^d nodes
  - Therefore,

$$2^d \leq n \Rightarrow d \leq log_2 n$$

- Adding or deleting an entry in a heap with *n* entries is *O*(*d*), where *d* is the depth of the tree
- The operations are *O*(log$_2$ *n*)

Time Analysis for B-Tree



- For searching, insertion and deletion, the number of total steps is a constant times the depth of the B-tree
- The depth of a non-empty tree: $d \leq \log_{\texttt{MINIMUM}} n$
- All the functions are: *O*(log *n*)

Logarithm Rules

$$\text{if } x = a^n \text{ then } \log_a x = n$$

$$\log_a xy = \log_a x + \log_a y$$

$$\log_a x^m = m \log_a x$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$\log_a 1 = 0$$

Summary

- A **heap** is a complete binary tree that follows the rule that the entry at any node is never less than any of its children
  - Provide an efficient implementation of **priority queues**
- The STL includes a priority queue class as well as algorithms for building and manipulating heaps
- A **B-tree** is a tree for storing entries in a manner that follows six rules:
  - The first two rules specify the minimum and maximum number of entries for each node
  - The third rule requires each node's entries to be sorted from smallest to largest
  - Rules 4 and 5 indicate how many subtrees a non-leaf node must have and impose an order on the elements of the subtrees
  - The last rule requires each leaf to be at the same depth

Summary

- The tree algorithms that we have seen for binary search trees, heaps, and B-trees **all have worst-case time performance of *O*(*d*)**, where ***d* is the depth of the tree**
- The depth of a heap or B-tree is never more than *O*(log *n*), where *n* is the number of nodes

Appendix 1: The STL Multiset Class and Its Iterator

Appendix 1: The STL Multiset Class and Its Iterator

The Multiset Template Class

- A **multiset** is an STL class similar to our bag, it permits a collection of items to be stored, **where each item may occur multiple times** in the multiset
- The value of an element also identifies it (the value is itself the **key**) – the keys are used to order the items
- The elements in a multiset are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object
- Sets and Multisets are typically implemented as **binary search trees**
- Another STL class, the **set** class, has the same interface as the multiset class, except that **it stores elements without repetition**, additional insertions of an element that is already in a set will have no effect
- A program that uses multisets or sets must include the header file <set>

Appendix 1: The STL Multiset Class and Its Iterator

The Multiset Template Class

❑Example:

```
multiset<int> first;
first.insert(8);
first.insert(4);
first.insert(8);
// After these statements, first contains two 8s and a 4.
```

- The name of the data type is `multiset`, but this name is augmented by `<int>` to indicate the type of elements that will reside in the bag
- This augmentation is called a **template instantiation**, and it differs from the way that we specified the underlying type for our own bag

Appendix 1: The STL Multiset Class and Its Iterator

The Multiset Template Class (Cont'd)

- The type of the item in a multiset has one restriction that is not required for our own bag**: It must be possible to compare two items using a "less than" operator**
- **Usually, this comparison operator is simply the "<" operator** that is provided for a built-in data type (such as integers) or provided as a function for a class (such as strings)

A **strict weak ordering** for a class is a comparison operator (<) that meets these requirements:

- **Irreflexivity:** If x and y are equal, then neither `(x < y)` nor `(y < x)` is true
  - This means that `(x < x)` is never true.
- **Antisymmetry:** If x and y are not equal, then either `(x < y)` or `(y < x)` is true, but not both
- **Transitivity:** Whenever there are three values (`x, y`, and `z`) with (`x < y`) and (`y < z`), then (`x < z`) is also true

Appendix 1: The STL Multiset Class and Its Iterator

Some Multiset Members

- **Constructors**
  - A default constructor creates an empty multiset
  - A copy constructor makes a copy of another multiset

- **Members that are similar to the bag:** These members are similar to our bag

```
• A type definition for the value_type
• A type definition for the size_type
• size_type count(const value_type& target) const;
• size_type erase(const value_type& target);
• size_type size( ) const;
```

Appendix 1: The STL Multiset Class and Its Iterator

Some Multiset Members (Cont'd)

- **The insert member function:** multiset's `insert` function can be used exactly like the bag' s `insert` function to add an item to a multiset
- The actual prototype for the **multiset' s insert function** specifies a return value called an **iterator**

```
iterator insert(const value_type& entry);
```

Appendix 1: The STL Multiset Class and Its Iterator

Some Multiset Members (Cont'd)

- An **iterator** is an **object** that permits a programmer to easily step through all the items in a container, **examining** the items and (**perhaps**) **changing** them
- Any **STL container** has a standard member function called **`begin`** that returns an iterator providing access to the first item in the container
- For a multiset, this "first" element is the smallest item according to the "less than" ordering that must be provided for the item type of any multiset

Appendix 1: The STL Multiset Class and Its Iterator

Some Multiset Members (Cont'd)

- A programmer can use the `begin` function and related operations to step through all the items in a container
- There are four operations required for the pattern:
  1. **`begin`**: A container has a `begin` member function and its return value is an iterator that provides access to the first item in the container

❑**Example:** suppose that `actors` is a multiset of strings, we can write this code to obtain the beginning iterator

```
multiset<string> actors;
multiset<string>::iterator role;
role = actors.begin( );
```

- The `iterator` in this example is a variable called `role`, and its data type is `multiset<string>::iterator`
- The `multiset<string>::iterator` data type is part of the multiset class, similar to the way that `value_type` and `size_type` are part of the class

Appendix 1: The STL Multiset Class and Its Iterator

Some Multiset Members (Cont'd)

2. **The `*` `operator`:** Once an iterator has created, the * (asterisk) operator can be used **to access the current element of the iterator**

❑**Example**: To print the current string of the role iterator:

```
cout << *role << endl;
```

- The asterisk can be applied to any iterator, **causing the iterator to return its current item**
- In general, the *role notation can be used for both accessing and changing the iterator's current item

❑**Example:** an item might be changed with an assignment:

```
*role = "shemp";
```

- **Some iterators forbid the item from being changed, like multiset's iterator, where the * operator can be used to access an item in the multiset, but not to change an item**

Some Multiset Members (Cont'd)

3. **The `++ operator`:** can be used to move an iterator forward to the next item in its collection

❑ **Example**

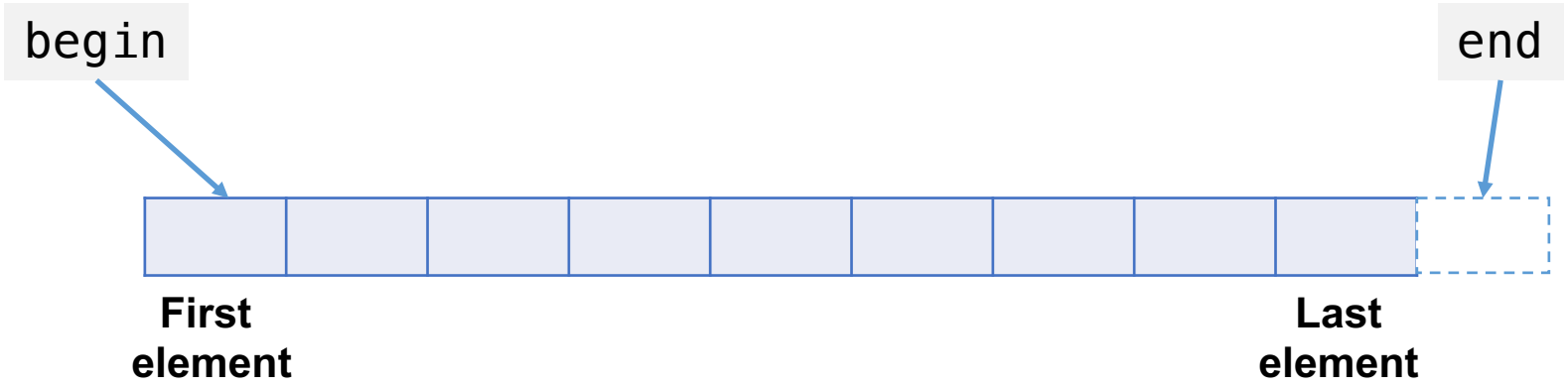
```
++role;
```

- The `++ operator` can be used before the iterator `(++role)` or after the iterator `(role++)`
- Both versions of the `++` operator are actually functions that return an iterator
- The return value of `++role` is the iterator **after** it has already moved forward
- The return value of `role++` is a copy of the iterator **before** it has moved forward
- For most iterators, the `++role` version is more efficient because it does not need to keep a copy of the old iterator before it moved forward

Some Multiset Members (Cont'd)

4. **end:** A container has an end member function that returns an iterator to mark the end of its items

- If an iterator moves forward through the container, it will eventually reach the end
- Once an iterator reaches the end, it has already gone **beyond the last item of the container** and the ∗ operator must not be used any more because there are no more items



Appendix 1: The STL Multiset Class and Its Iterator

❑ Example

- This example creates a multiset of integers, then uses an iterator to step through those integers one at a time:

```
multiset<int> ages;
multiset<int>::iterator it;

ages.insert(4);
ages.insert(12);
ages.insert(18);
ages.insert(12);

for (it = ages.begin( ); it!= ages.end( ); ++it)
{
    cout << *it << endl;
}
```

```
Output:
4
12
12
18
```

Appendix 1: The STL Multiset Class and Its Iterator

❑ Example

```
#include <iostream>
#include <set>

int main ()
{
    int myints[] = {1,2,2,2,3};
    std::multiset<int> mymultiset (myints,myints+5);

    std::multiset<int>::iterator it;

    std::cout << "mymultiset contains:";
    for (std::multiset<int>::iterator it=mymultiset.begin();
                                      it!=mymultiset.end(); ++it)

   std::cout << ' ' << *it;
   std::cout << '\n';
   return 0;
}
```

```
Output:
mymultiset contains: 1 2 2 2 3
```

Appendix 1: The STL Multiset Class and Its Iterator

Testing Iterators for Equality

- You can use:
  - The `!= operation` to test whether **two iterators of the same container** are not equal
  - The `== operation` to test whether **two iterators of the same container** are equal
- For a container object**, two of its iterators are equal if they are at the same location**, or if they have both gone to the end( ) of the container
- Note: It is an error to compare two iterators from different containers

Appendix 1: The STL Multiset Class and Its Iterator

Other Multiset Operations

- Multisets have other operations, some of which use iterators

❑ Example 1

```
iterator find(const value_type& target);
```

Searches for the first item in the multiset that is equal to the `target`

- If `find` function finds such an item, it returns an iterator whose current element is that item
- If there is no such item, then `find` returns an iterator that is equal to `end()`

❑ Example 2

```
void erase(iterator i);
```

- The `erase` function is an alternative to the usual `erase` function, its parameter is an iterator, and the function removes the iterator's current item

Appendix 1: The STL Multiset Class and Its Iterator

❑ Example

- Using `find` and `erase` together, this code will erase the first occurrence of 42 in `multiset m`

```
multiset<int> m;
multiset<int>::iterator position;

position = m.find(42);

if (position != m.end( ))
    m.erase(position);
```

Appendix 1: The STL Multiset Class and Its Iterator

Invalid Iterators

- After an iterator has been set, it can easily move through its container
- **Changes to the container (such as insertions or removal) can cause all of the container's iterators to become invalid**

❑ Examples

  - The position iterator is invalid after its item is erased
  - Containers where insertions can invalidate all iterators

- When an iterator becomes invalid because of a change to its container, that iterator can no longer be used until it is assigned a new value

Appendix 1: The STL Multiset Class and Its Iterator

Clarifying The Const Keyword: Const Iterators

- A **const iterator** is an iterator that is forbidden from changing its underlying container in any way
- It is written as a single word "**`const_iterator`**" and does not use the keyword `const`
- `const` iterators can be obtained from the `begin` and `end` functions of any constant container
- If multiset `m` is declared with the keyword `const`, the return value of `m.begin( )` and `m.end( )` are `const_iterator` rather than iterator
- A `const` iterator can move through its container of items, but it is forbidden from adding, removing, or changing any items

❑ Example: If `it` is a `const` iterator for the multiset `m`, then we may not activate `m.erase(it)`

Appendix 1: The STL Multiset Class and Its Iterator

Clarifying The Const Keyword Part 5: Const Iterators (Cont'd)

- A small function to count how many integers in a multiset are less than a specified target

```
multiset<int>::size_type count_less_than_target
                                (const multiset<int>& m, int target)
{
    multiset<int>::size_type answer = 0;
    multiset<int>::const_iterator it;
   for (it = m.begin( ); it!= m.end( ); ++it)
   {
       if (*it < target)
           ++answer;
   }
    return answer;
}
```

- The `const_iterator` data type is part of the multiset class, just like `value_type, size_type`, and the ordinary iterator

Appendix 2: The STL Map and Multimap Classes

Appendix 2: The STL Map and Multimap Classes

- Set only stores keys; map stores <key, value> pairs
- Balanced trees (**red-black tree**) are used as the underlying data structure for the **map** class

```
template < class Key,                                    // map::key_type
class T,                                                 // map::mapped_type
class Compare = less<Key>,                               // map::key_compare
class Alloc = allocator<pair<const Key,T> >              // map::allocator_type
>
class map;
```

Example: Months of the year as keys, with their number of days as values:

```
map<string, int> months;
```

Type of the key

Type of the data stored

- The map class provides subscripting through its keys
- Values can be entered into a map either through the insert function or through array-type assignment: `months["July"] = 31;`

Appendix 2: The STL Map and Multimap Classes

- The types of *key* and *mapped value* may differ
- They are grouped together in member type `value_type`, which is a `pair` type combining both:

```
typedef pair<const Key, T> value_type;
```

Properties

- ***Associative***
  - Elements are referenced by their *key* and not by their absolute position in the container
- ***Ordered***
  - The elements follow a strict order at all times
- ***Map***
  - Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*
- ***Unique keys***
  - No two elements in the container can have equivalent *keys*

Appendix 2: The STL Map and Multimap Classes

❑ Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;

    mymap.insert ( std::pair<char,int>('a',100) );
    mymap.insert ( std::pair<char,int>('z',200) );

    mymap['z'] = 0;
```

Type of the key

Type of the data stored

Appendix 2: The STL Map and Multimap Classes

```
    std::pair<std::map<char,int>::iterator, bool> ret;


    ret = mymap.insert ( std::pair<char,int>('z',500) );


    if (ret.second == false) {
        std::cout << "element 'z' already existed";
        std::cout << " with a value of " << ret.first->second << '\n';
    }

}
```

```
Output:
element 'z' already existed with a value of 0
```

Appendix 2: The STL Map and Multimap Classes

- Balanced trees are used as the underlying data structure for the **multimap** class
- It allows multiple values to be associated with a single key (and therefore: you can't use the subscripting notation that comes with a map

```
template < class Key,                     // multimap::key_type
class T,                                  // multimap::mapped_type
class Compare = less<Key>,                // multimap::key_compare
class Alloc = allocator<pair<const Key,T> >// multimap::allocator_type
>
class multimap;
```

Appendix 2: The STL Map and Multimap Classes

❑ Example

```
#include <iostream>
#include <map>

using namespace std;

int main(int argc, const char * argv[])
{
    multimap<string, int> cities;
    multimap<string, int>::const_iterator i;

    cities.insert(make_pair("scu", 950));
    cities.insert(make_pair("scu", 53));

   for (i = cities.begin(); i != cities.end(); ++i)
   {
       cout << i->first << '\t' << i->second << endl;
   }

    return 0;
}
```

```
Output:
scu     950
scu     53
```