COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

CSC212
Data Structure
- Section FG

Lecture 17

B-Trees and the Set Class

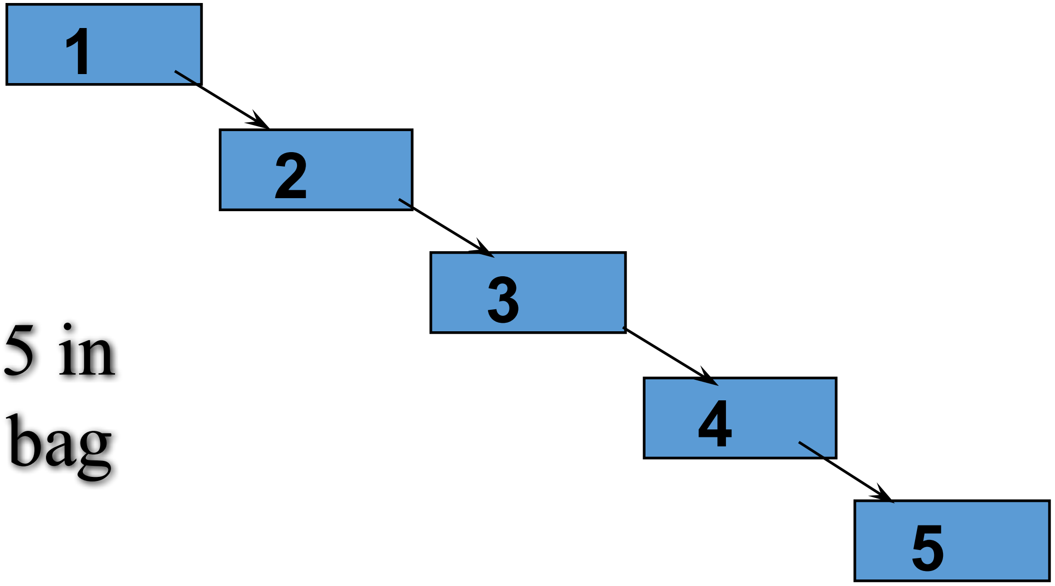
Instructor: Feng HU
Department of Computer Science
City College of New York

Topics

- Why B-Tree
  - The problem of an unbalanced tree
- The B-Tree Rules
- The Set Class ADT with B-Trees
- Search for an Item in a B-Tree
- Insert an Item in a B-Tree (*)
- Remove a Item from a B-Tree (*)

The problem of an unbalanced BST

- Maximum depth of a BST with n entires: n-1



- An Example:

  Insert 1, 2, 3, 4, 5 in that order into a bag using a BST

- Run BagTest!

Worst-Case Times for BSTs

- Adding, deleting or searching for an entry in a BST with n entries is O(d) in the worst case, where d is the depth of the BST
- Since d is no more than n-1, the operations in the worst case is (n-1).
- Conclusion: the worst case time for the add, delete or search operation of a BST is O(n)

Solutions to the problem

- Solution 1
  - Periodically balance the search tree
  - **Project 10.9, page 516**
- Solution 2
  - A particular kind of tree : B-Tree
  - proposed by Bayer & McCreight in 1972

The B-Tree Basics

- Similar to a binary search tree (BST)
  - where the implementation requires the ability to compare two entries via a ***less-than operator (<)***
- But a B-tree is NOT a BST – in fact it is not even a binary tree
  - *B-tree nodes have many (more than two) children*
- Another important property
  - *each node contains more than just a single entry*
- Advantages:
  - *Easy to search, and not too deep*

Applications: bag and set

- The Difference
  - two or more equal entries can occur many times in a bag, but not in a set
  - C++ STL: set and multiset (= bag)
- The B-Tree Rules for a Set
  - We will look at a "set formulation" of the B-Tree rules, but keep in mind that a "bag formulation" is also possible

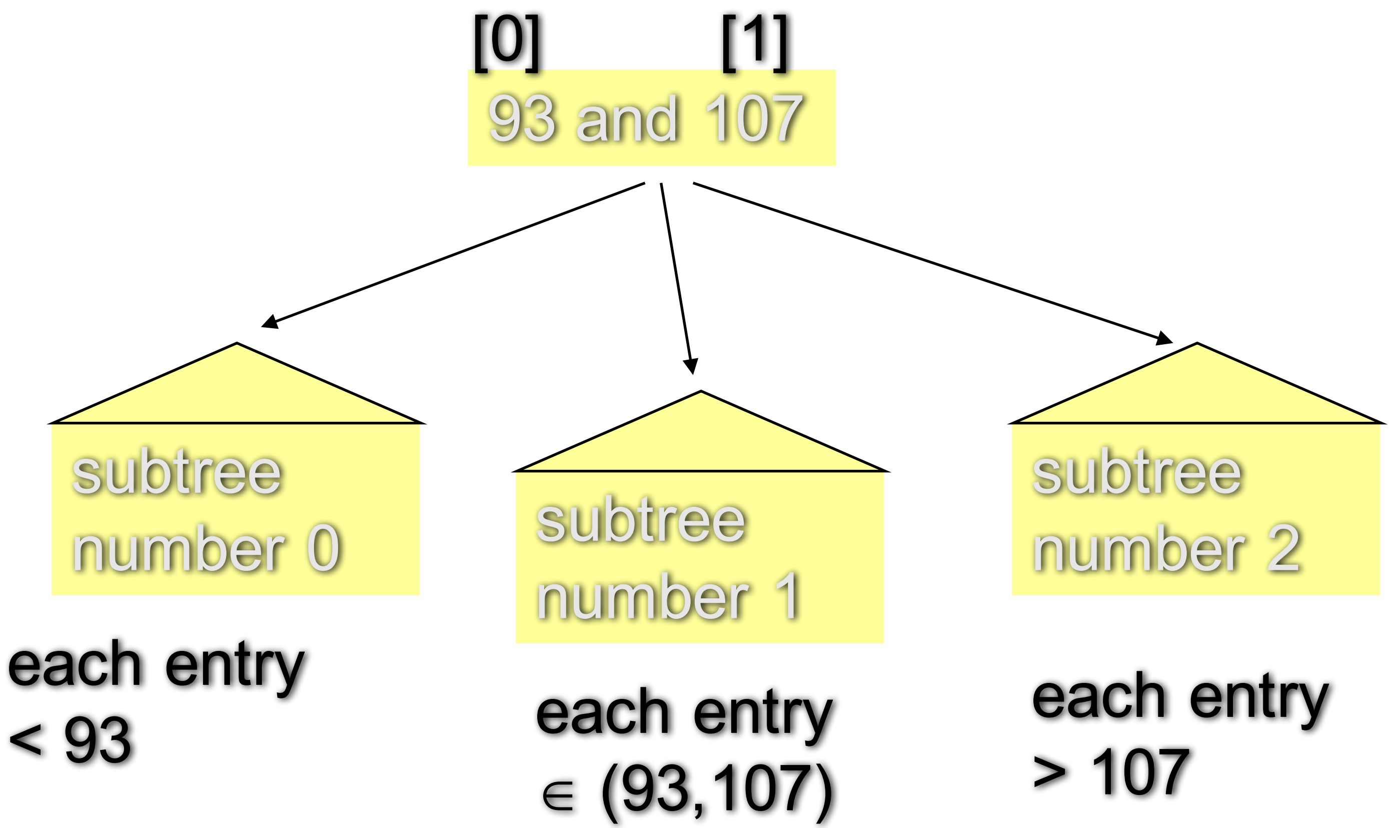
The B-Tree Rules

- The entries in a B-tree node
  - B-tree Rule 1: The root may have as few as one entry (or 0 entry if no children); every other node has at least MINIMUM entries
  - B-tree Rule 2: The maximum number of entries in a node is 2* MINIMUM.
  - B-tree Rule 3: The entries of each B-tree node are stored in a partially filled array, sorted from the smallest to the largest.

The B-Tree Rules (cont.)

- The subtrees below a B-tree node
  - B-tree Rule 4: The number of the subtrees below a non-leaf node with n entries is always n+1
  - 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
    - (b) An entry at index i is less than all the entries in subtree number i+1 of the node

An Example of B-Tree

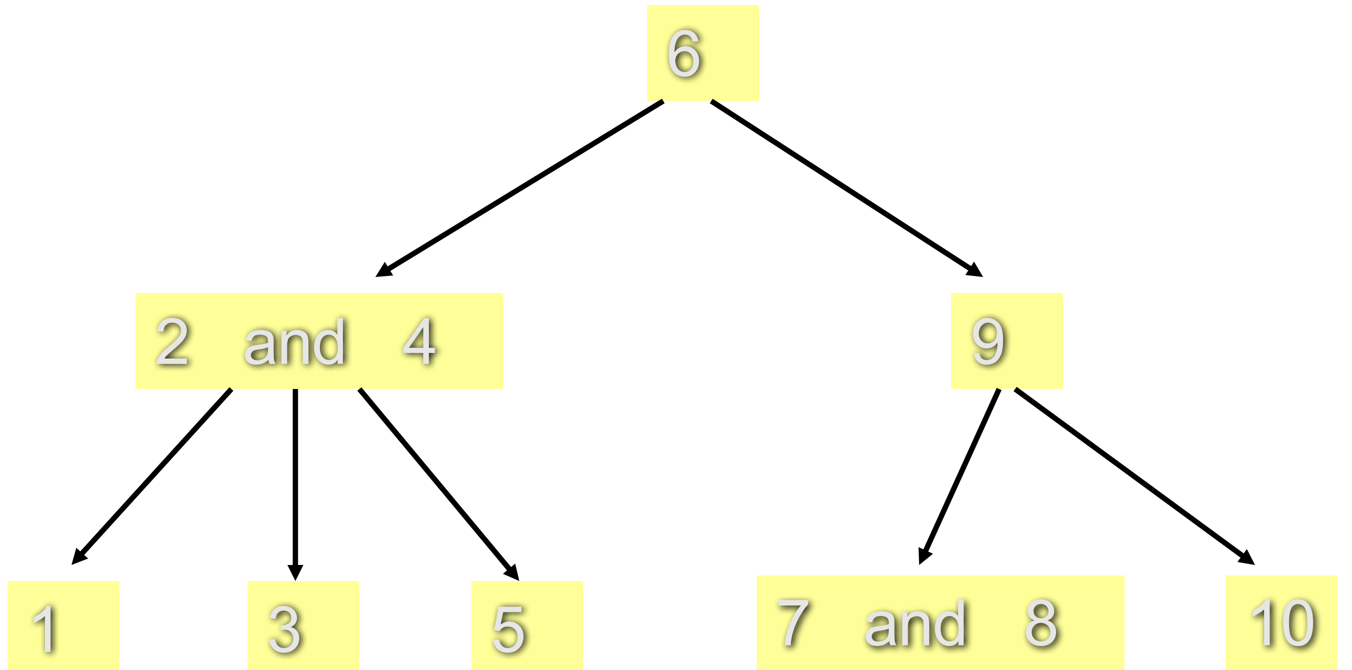


What kind traversal can print a sorted list?

The B-Tree Rules (cont.)

- A B-tree is balanced
  - B-tree Rule 6: Every leaf in a B-tree has the same depth
- This rule ensures that a B-tree is balanced

Another Example, MINIMUM = 1



Can you verify that all 6 rules are satisfied?

The **set** ADT with a B-Tree

set.h (p 528-529)

- Combine fixed size array with linked nodes
  - data[]
  - *subset[]
- number of entries vary
  - data_count
  - up to 200!
- number of children vary
  - child_count
  - = data_count+1?

```
template <class Item>
  class set
  {
  public:
        ... ...
     bool insert(const Item& entry);
     std::size_t erase(const Item& target);
     std::size_t count(const Item& target) const;
  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]; // why +1? -for insert/erase
     std::size_t child_count;
     set *subset[MAXIMUM+2]; // why +2? - one more

};
```

Invariant for the **set** Class

- The entries of a set is stored in a B-tree, satisfying the six B-tree rules.
- The number of entries in a node is stored in data_count, and the entries are stored in data[0] through data[data_count-1]
- The number of subtrees of a node is stored in child_count, and the subtrees are pointed by set pointers subset[0] through subset[child_count-1]

Search for a Item in a B-Tree

- Prototype:
  - std::size_t count(const Item& target) const;

- Post-condition:
  - Returns the number of items equal to the target
  - (either 0 or 1 for a set).

Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)

2) If (data[i] is target)

    return 1;

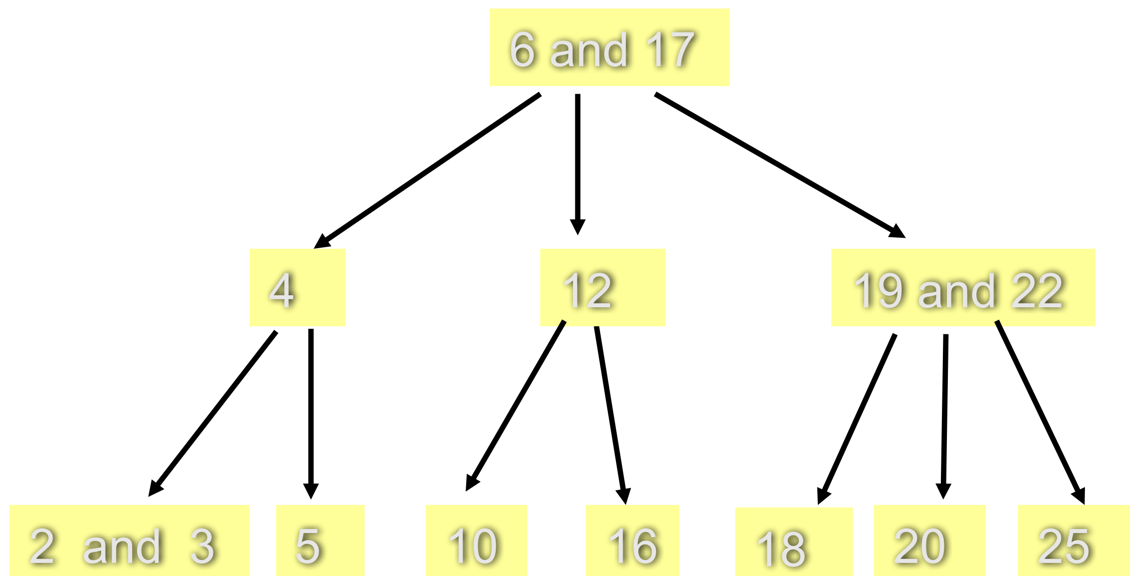
  else if (no children)

    return 0;

  else

    return

     subset[i]->count (target);



Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)

2) If (data[i] is target)

   return 1;

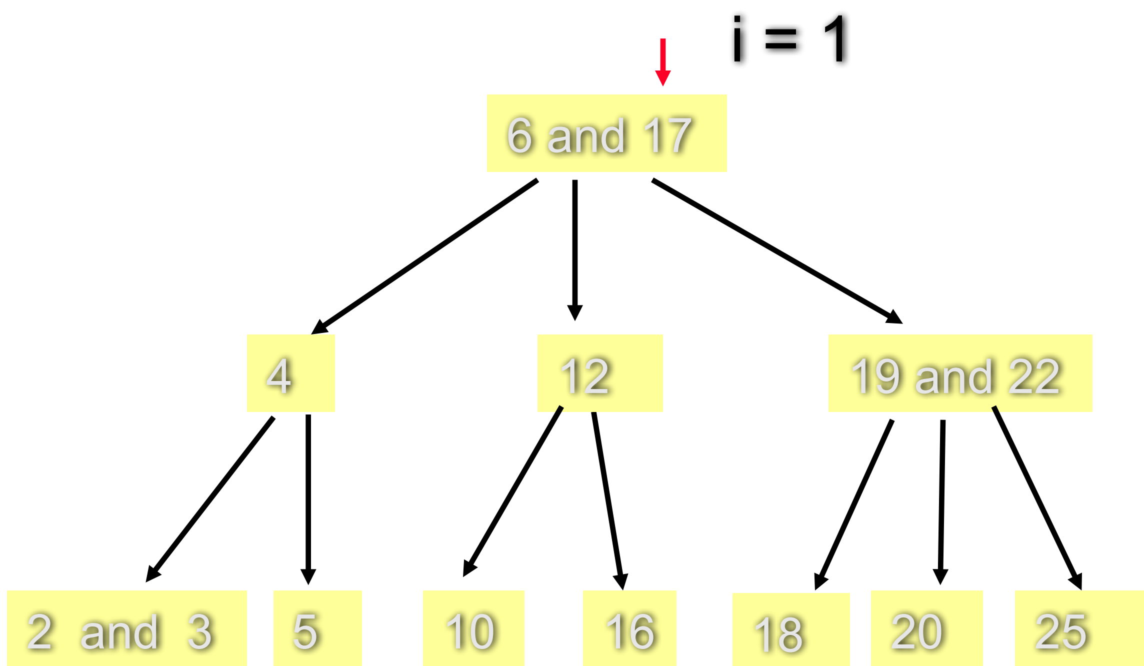
   else if (no children)

   return 0;

   else

   return

   subset[i]->count (target);



Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

    return 1;

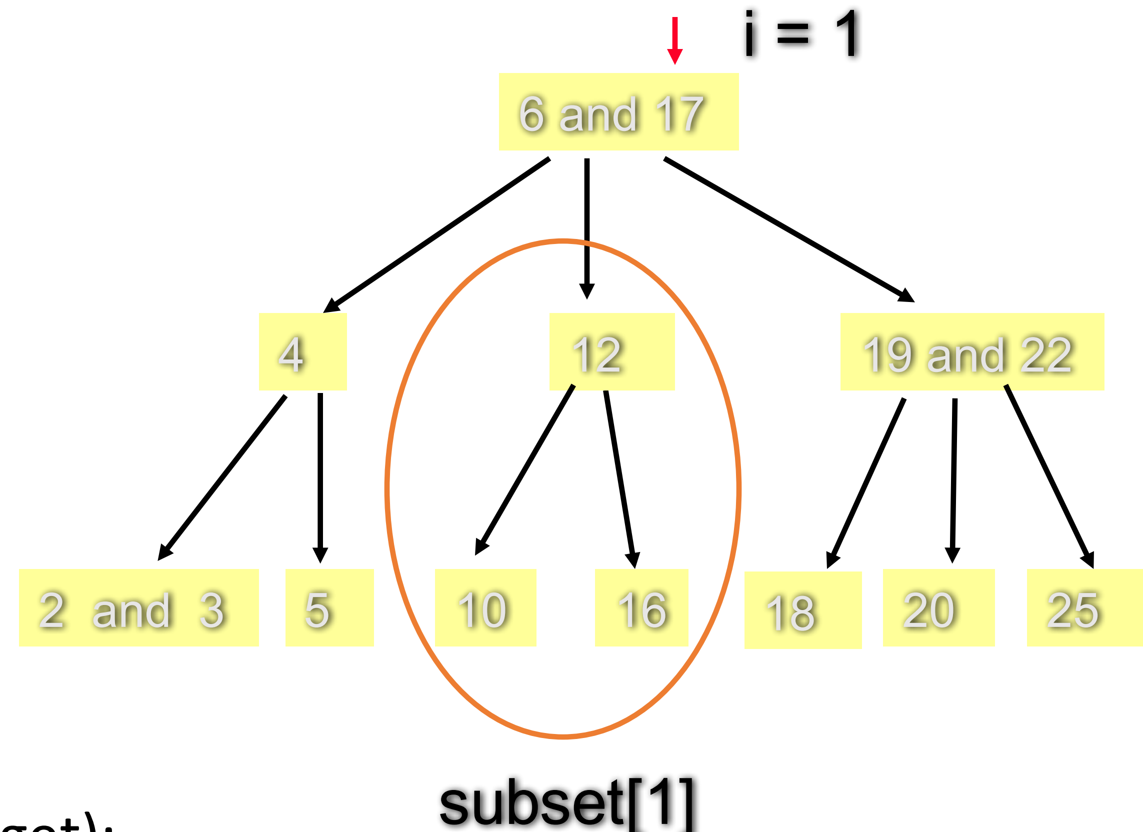
  else if (no children)

    return 0;

  else

    return

     subset[i]->count (target);



Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

    return 1;

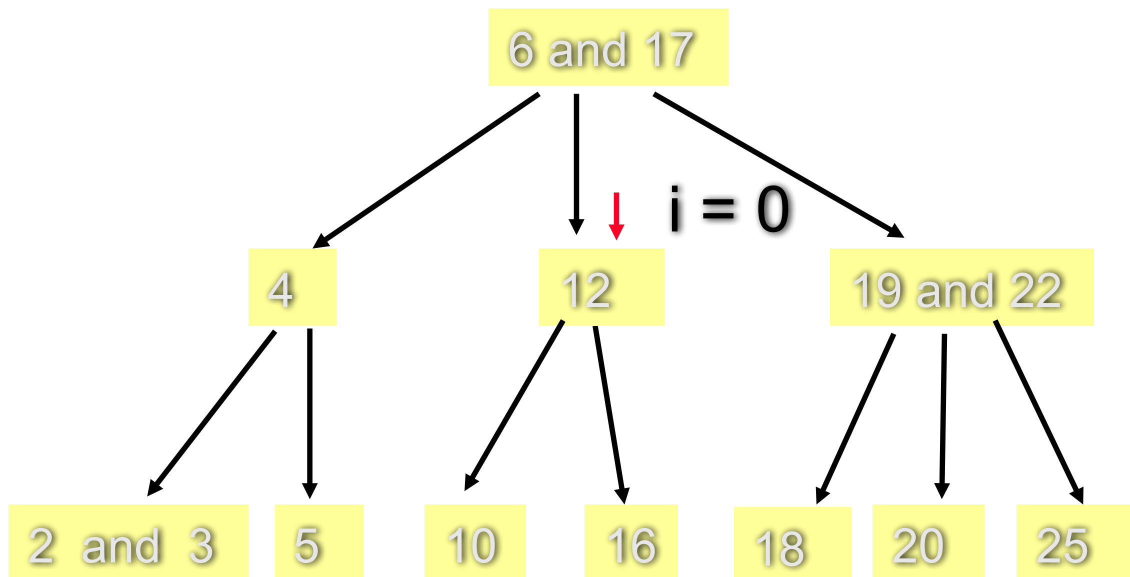
  else if (no children)

    return 0;

  else

    return

     subset[i]->count (target);



Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

    return 1;

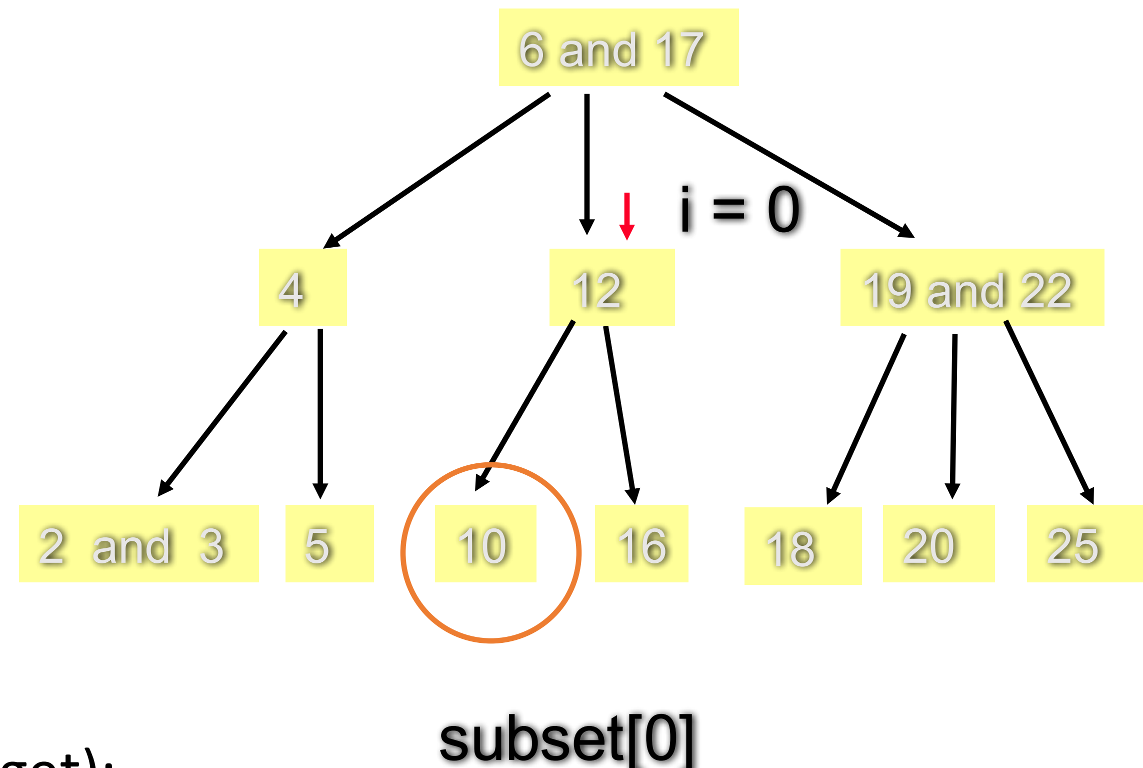
  else if (no children)

    return 0;

  else

    return

     subset[i]->count (target);



Searching for an Item: **count**

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

    return 1;

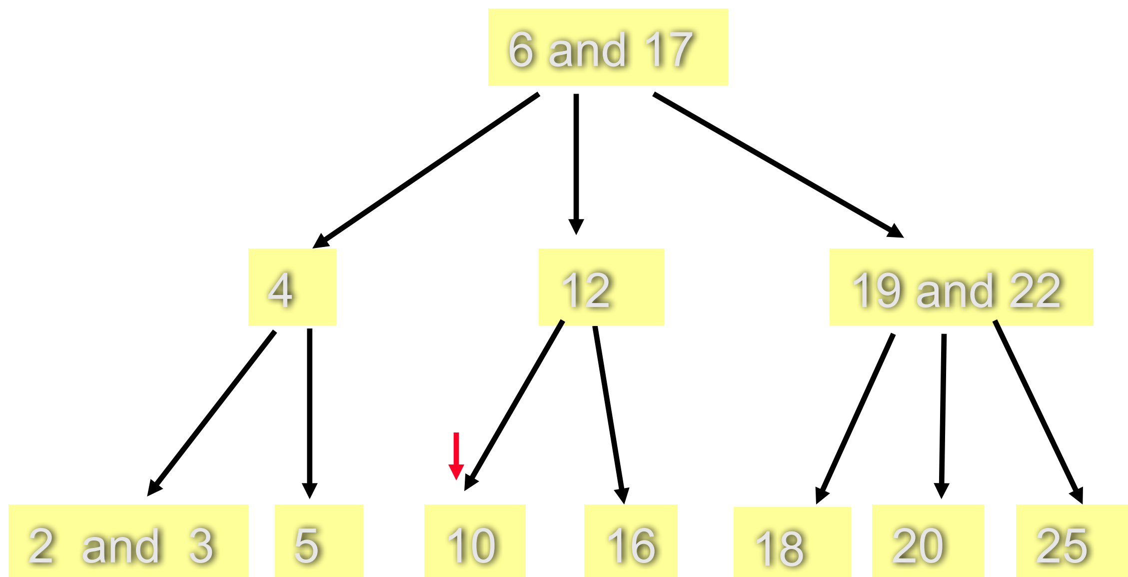
  else if (no children)

    return 0;

  else

    return

     subset[i]->count (target);



i = 0

data[i] is target !

Insert a Item into a B-Tree

- Prototype:
  - bool insert(const Item& entry);

- Post-condition:
  - If an equal entry was already in the set, the set is unchanged and the return value is false.
  - Otherwise, entry was added to the set and the return value is true.

Insert an Item in a B-Tree

insert (11);

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

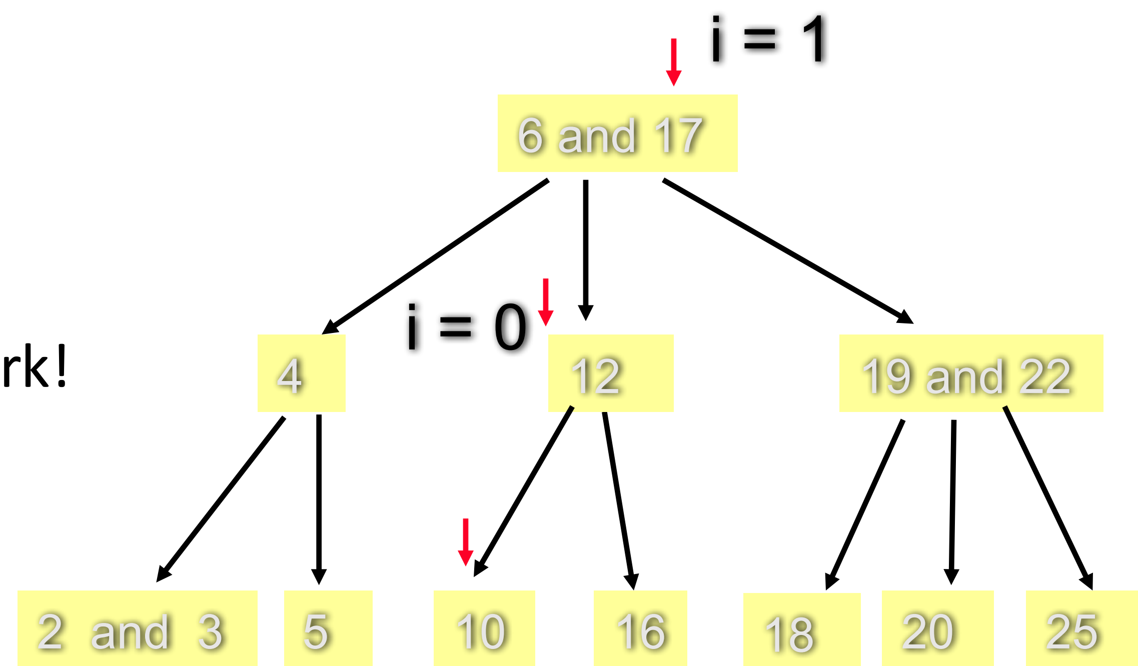
    insert entry at i;

     return true;

  else

    return

     subset[i]->insert (entry);

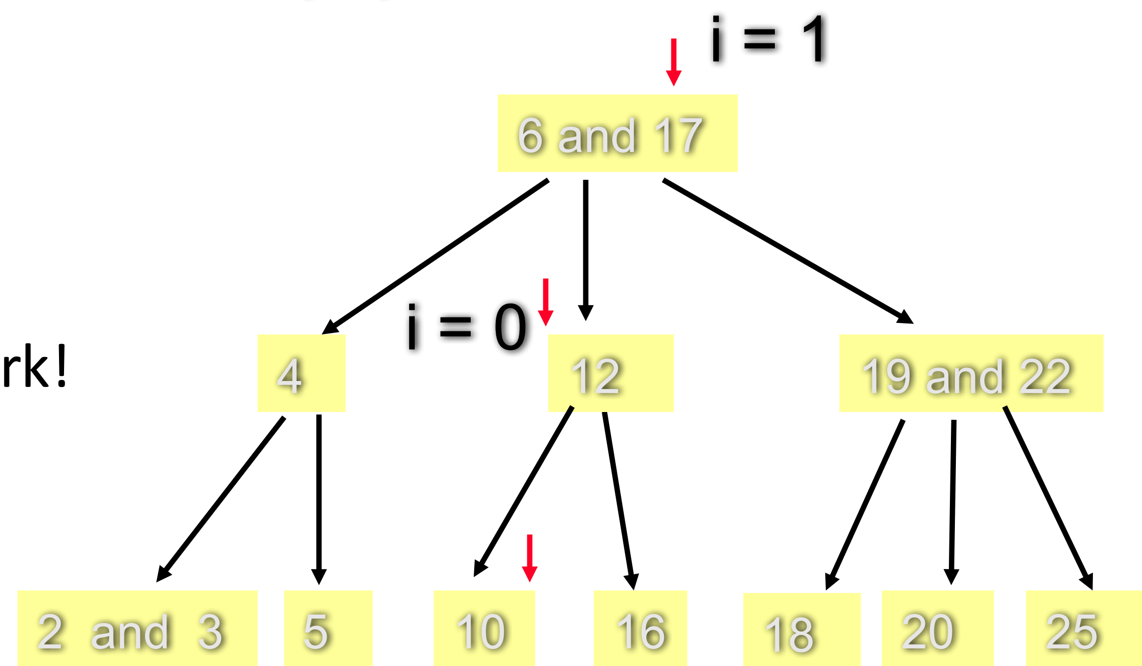


i = 0

data[i] is target !

Insert an Item in a B-Tree

insert (11); // MIN = 1 -> MAX = 2



i = 1

data[0] < entry !

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)
   return false; // no work!
   else if (no children)
   insert entry at i;
   return true;
   else
   return
   subset[i]->insert (entry);

Insert an Item in a B-Tree

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

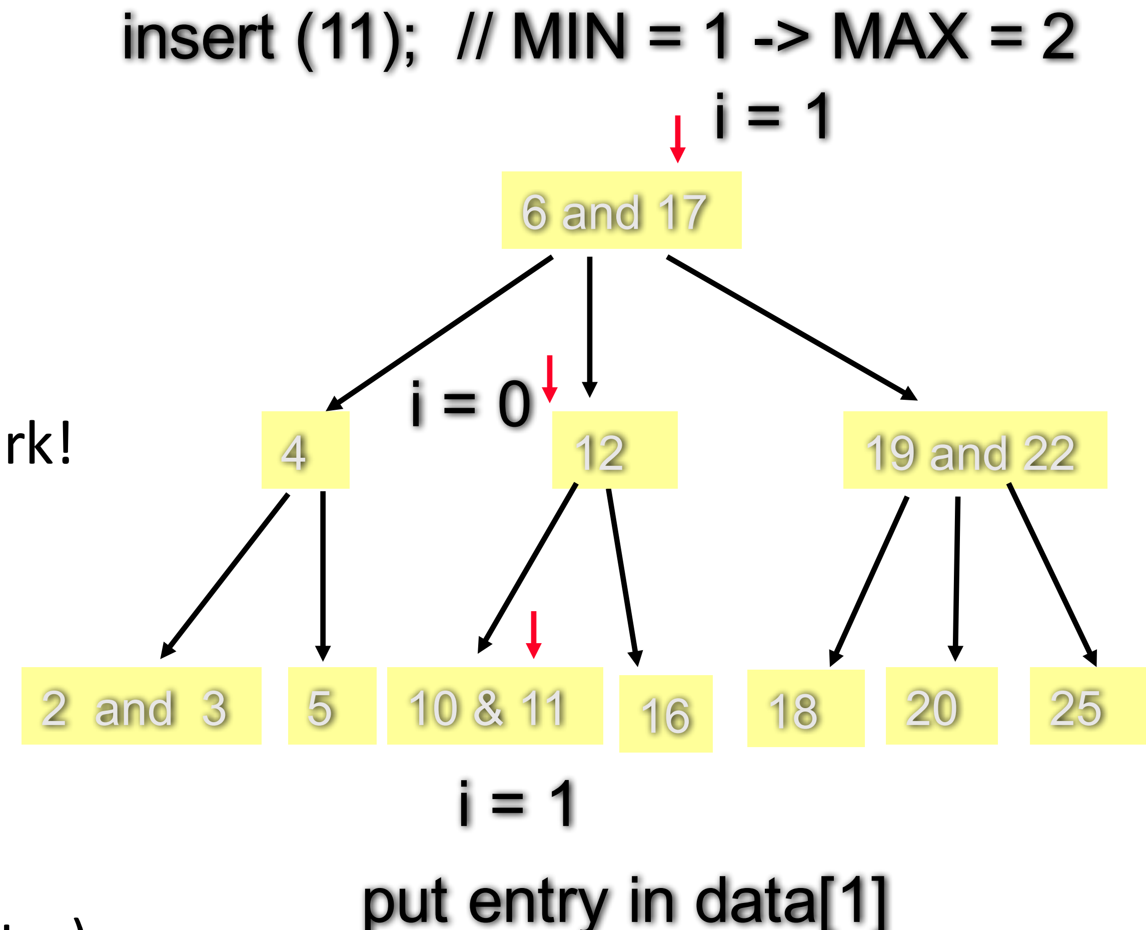
    insert entry at i;

     return true;

  else

    return

     subset[i]->insert (entry);



Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

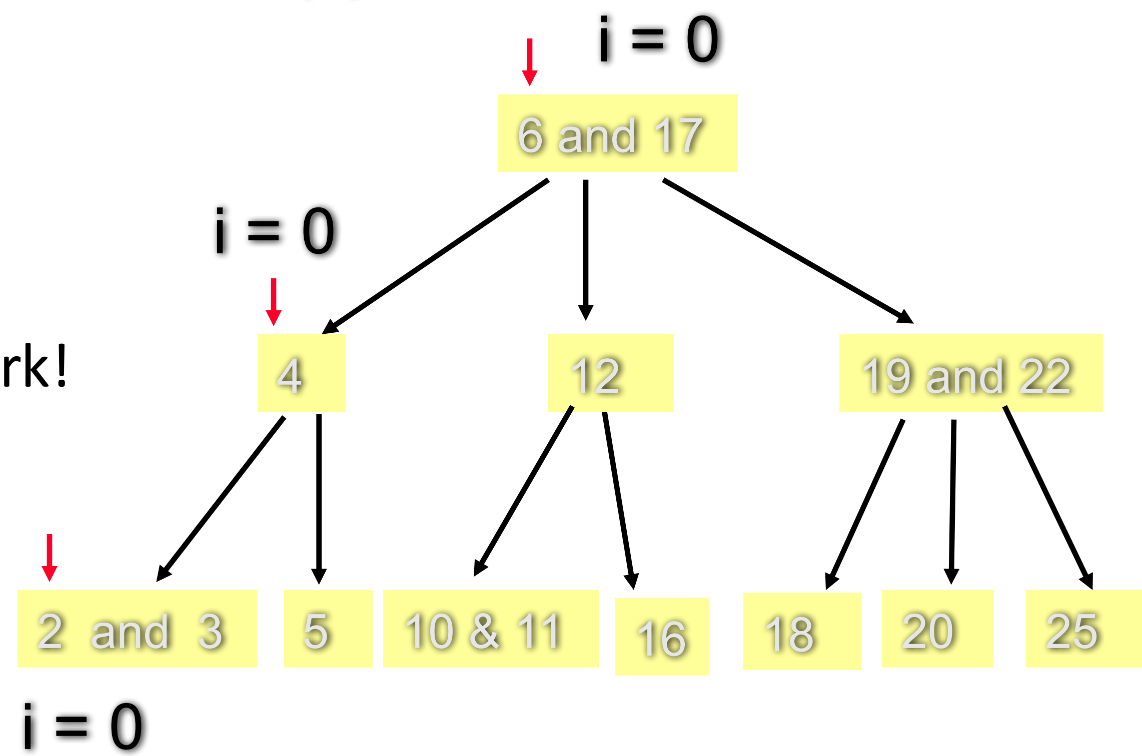
    insert entry at i;

     return true;

  else

    return

     subset[i]->insert (entry);



=> put entry in data[0]

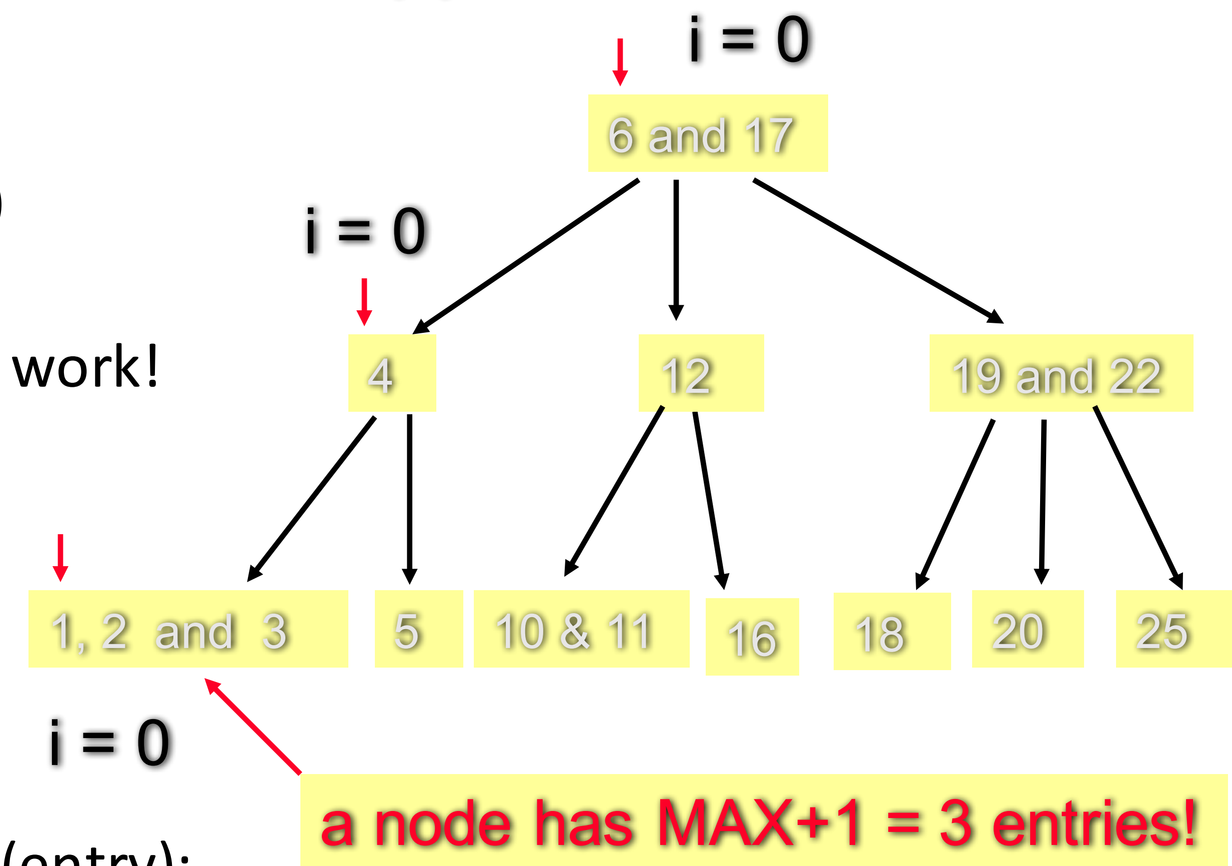
Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

```
      return false; // no work!
   else if (no children)
      insert entry at i;
       return true;
   else
      return
       subset[i]->insert (entry);
```



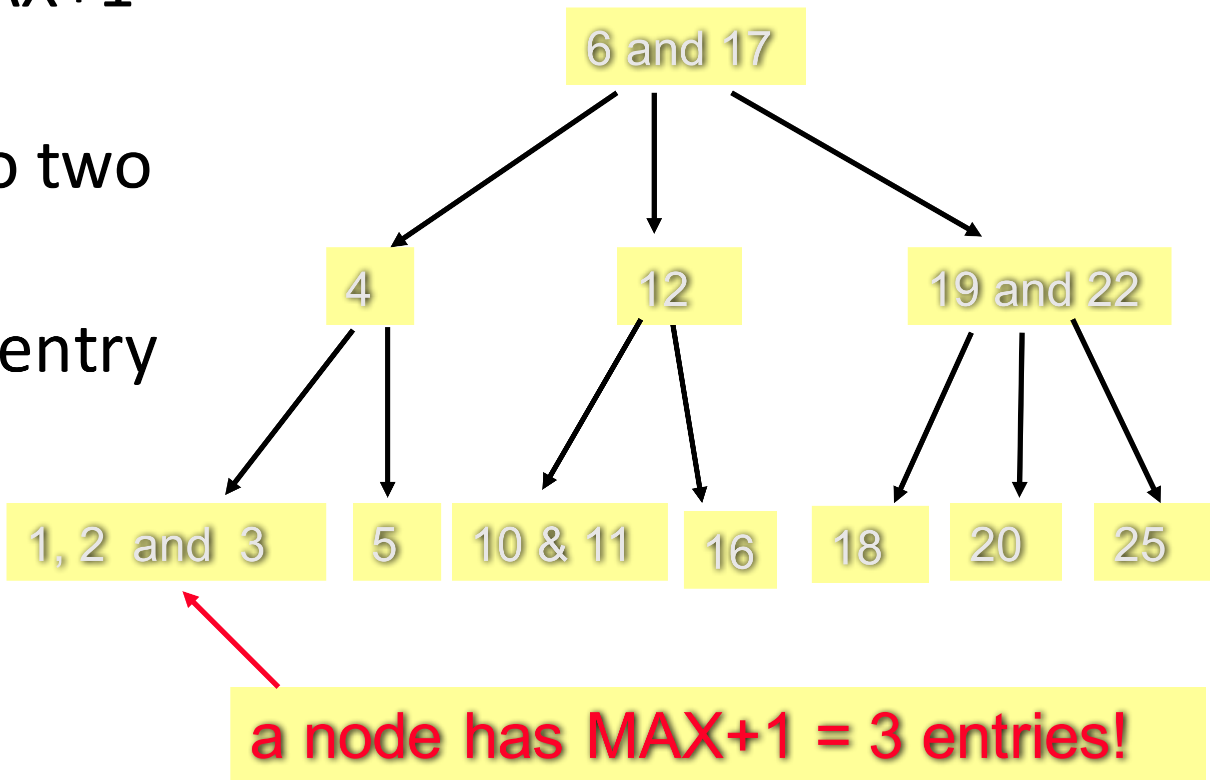
Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Fix the node with MAX+1 entries

❶ split the node into two from the middle

❶ move the middle entry up



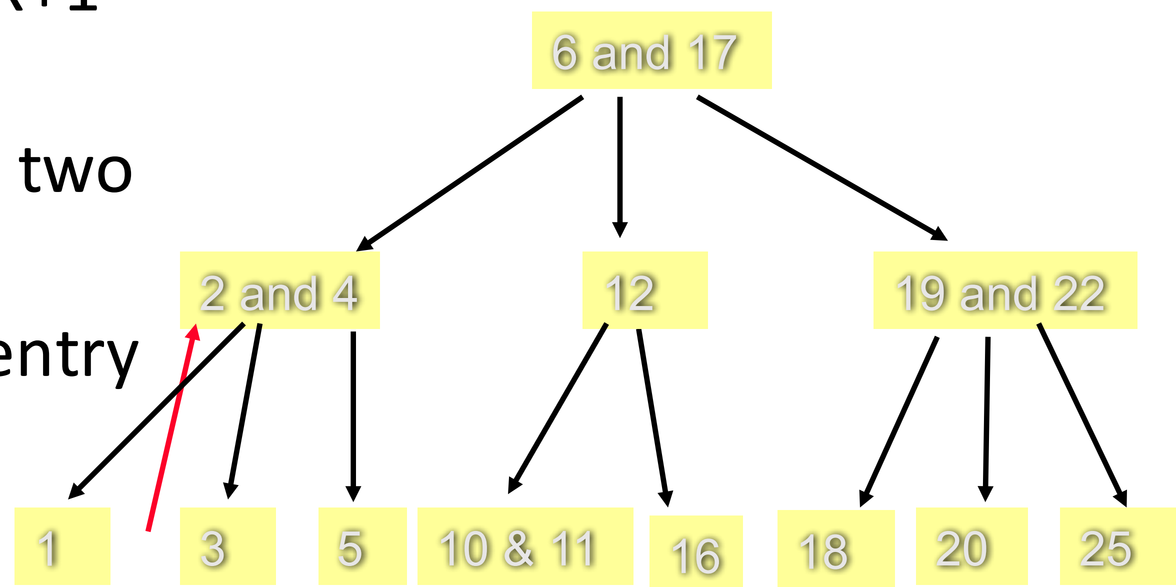
Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Fix the node with MAX+1 entries

❶ split the node into two from the middle

❶ move the middle entry up



Note: This shall be done recursively... the recursive function returns the middle entry to the root of the subset.

Inserting an Item into a B-Tree

- What if the node already have MAXIMUM number of items?
- Solution – loose insertion (p 551 – 557)
  - A loose insert may results in MAX +1 entries in the root of a subset
  - Two steps to fix the problem:
    - fix it – but the problem may move to the root of the set
    - fix the root of the set

Erasing an Item from a B-Tree

- Prototype:
  - std::size_t erase(const Item& target);
- Post-Condition:
  - If target was in the set, then it has been removed from the set and the return value is 1.
  - Otherwise the set is unchanged and the return value is zero.

Erasing an Item from a B-Tree

- Similarly, after "loose erase", the root of a subset may just have MINIMUM –1 entries
- Solution: (p557 – 562)
  - Fix the **shortage** of the subset root – but this may move the problem to the root of the entire set
  - Fix the **root** of the entire set (tree)

Summary

- A B-tree is a tree for sorting entries following the six rules
- B-Tree is balanced - every leaf in a B-tree has the same depth
- Adding, erasing and searching an item in a B-tree have worst-case time O(log n), where n is the number of entries
- However the implementation of adding and erasing an item in a B-tree is not a trivial task.