Advanced Tree Data Structures

Prepared by

Dr Annushree Bablani

Balanced Search Trees

- A search-tree data structure for which a height of ***O(log n)*** is guaranteed when implementing a dynamic set of ***n*** items.
- Examples:
  - AVL trees (Discussed in Unit-1)
  - 2-4 trees ***(This Lecture)***
  - B+-trees
  - Red-black trees

2-4 Trees

- Search Trees (but not binary)
- Also known as 2-4, 2-3-4 trees
- Very important as basis for Red-Black trees

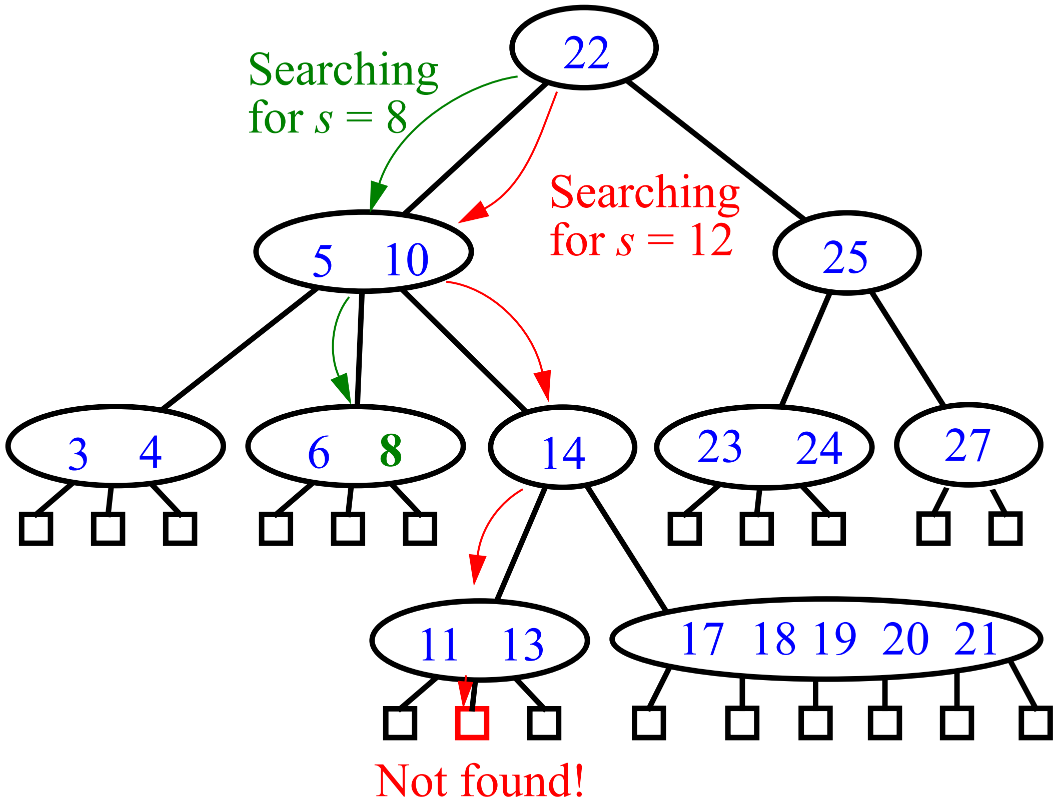
Multi-way Search Trees

- Each internal node of a multi-way search tree T:
  - has at least two children
  - stores a collection of items of the form (k, x), where k is a key and x is an element
  - contains d - 1 items, where d is the number of children
  - “contains” 2 pseudo-items: $k_0 = -\infty$, $k_d = \infty$
- Children of each internal node are “between” items
  - all keys in the subtree rooted at the child fall between keys of those items
- External nodes are just placeholders

Multi-way Searching

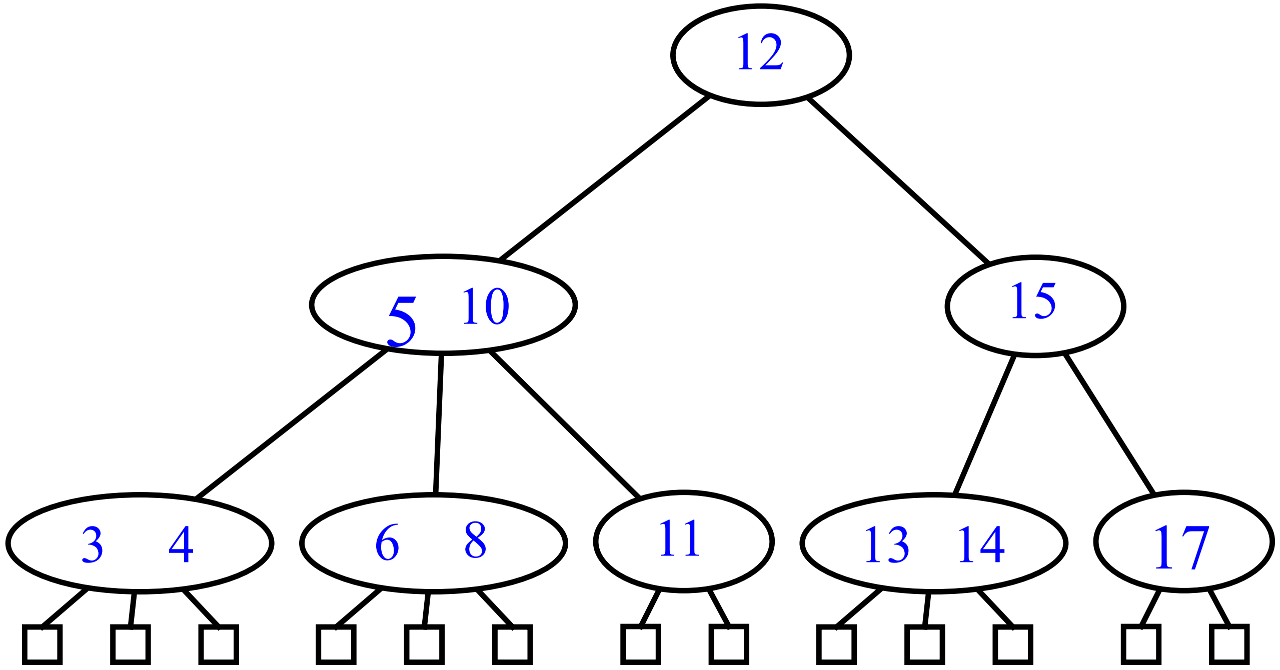
- Similar to binary searching
- If search key $s < k_1$, search the leftmost child
- If $s > k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if $d > 2$?
- Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.

Multi-way Searching



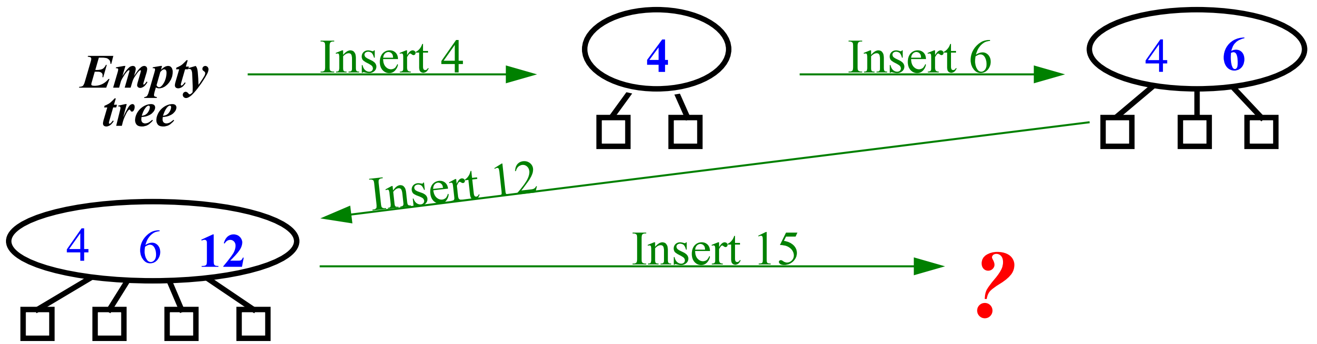
(2,4) Trees

- At most 4 children
- All external nodes have same depth
- Height *h* of (2,4) tree is $O(\log n)$.
- How is this fact useful in searching?



(2,4) Insertion

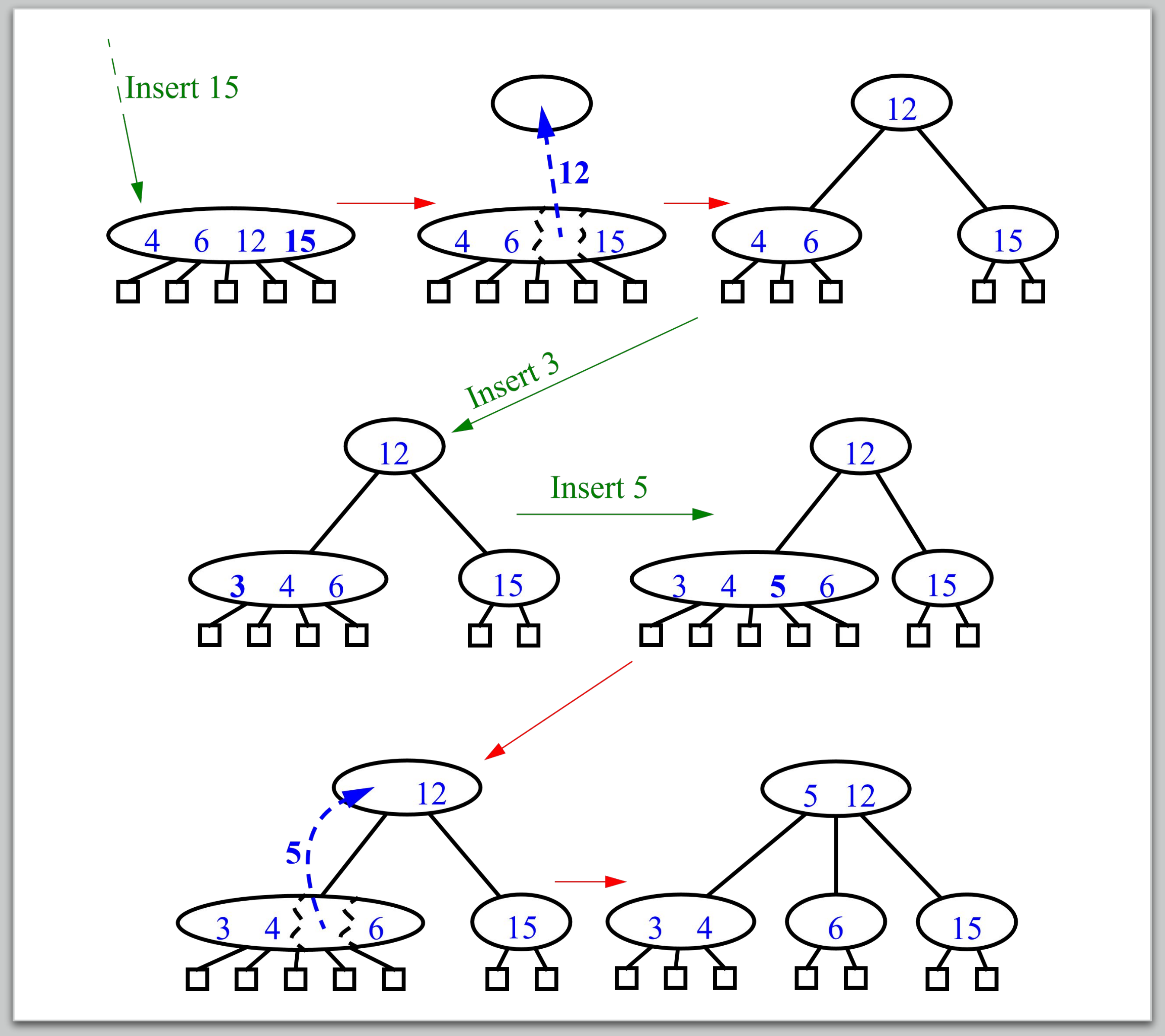
- Always maintain depth condition
- Add elements only to existing nodes



(2,4) Insertion

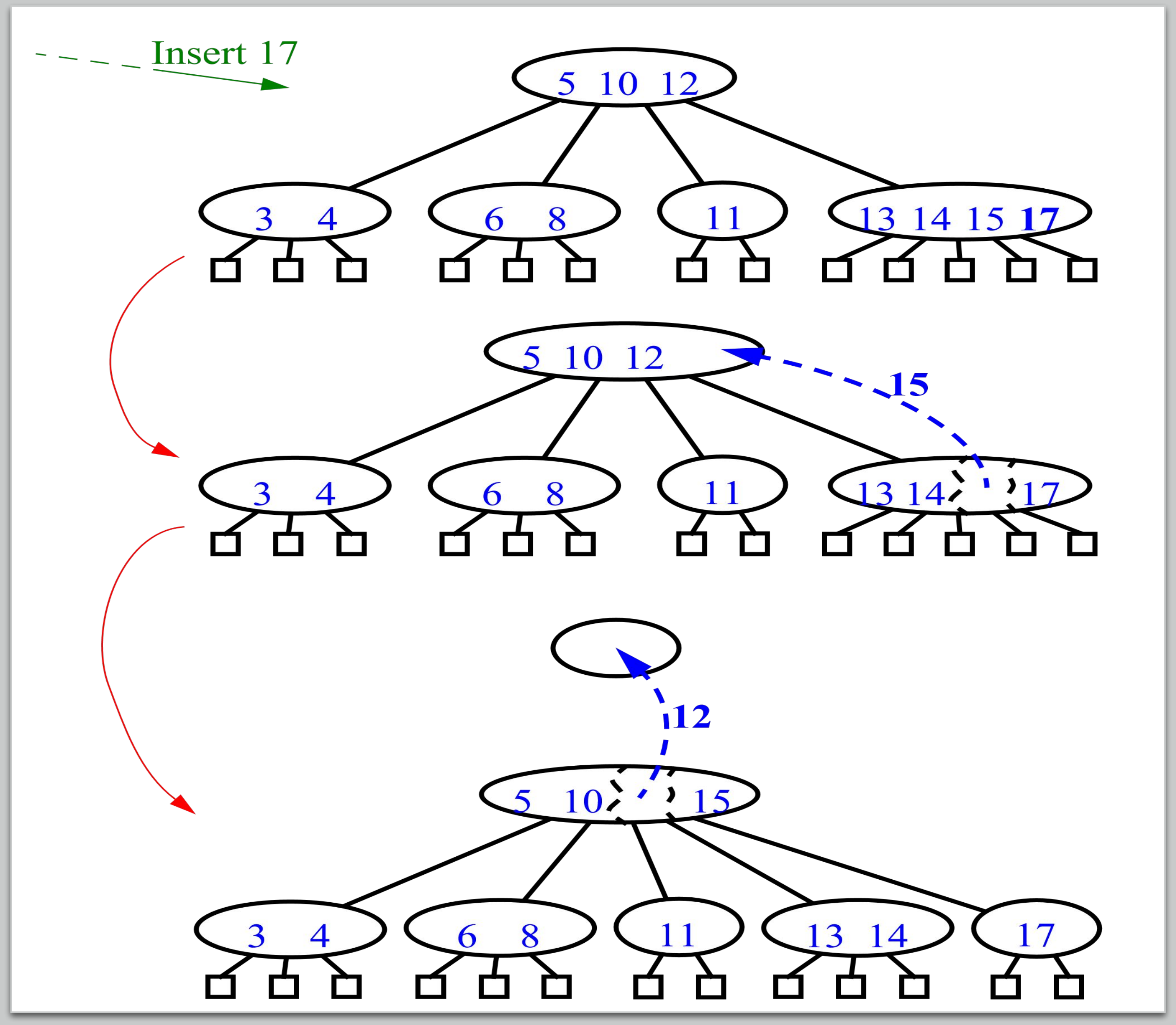
- What if that makes a node too big?
  - ***- overflow***
- Must perform a *split* operation
  - replace node *v* with two nodes *v*' and *v*''
  - *v*' gets the first two keys
  - *v*'' gets the last key - send the other key up the tree
    - if *v* is root, create new root with third key
    - otherwise just add third key to parent

(2,4) Insertion (cont.)



(2,4) Insertion (cont.)

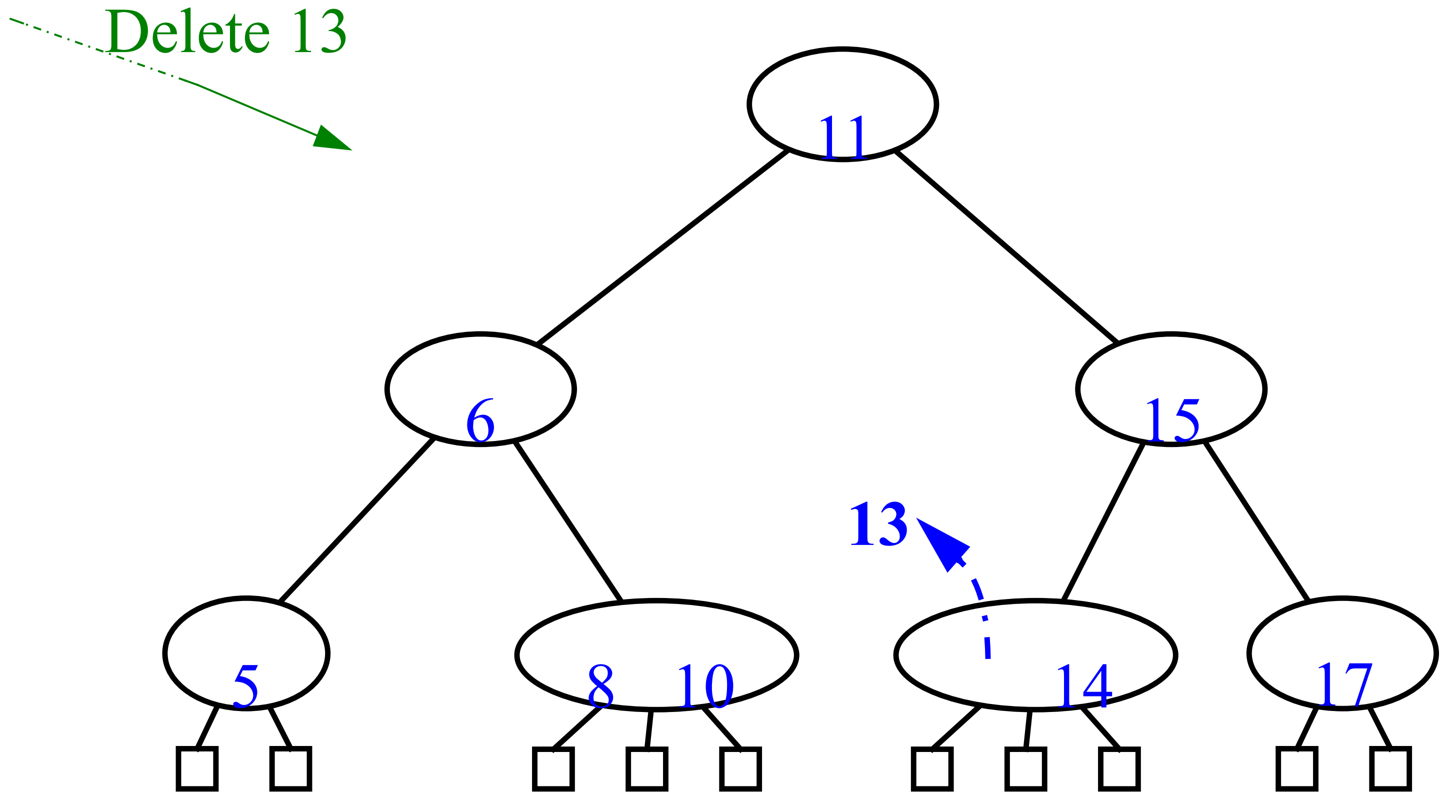
- Tree always grows from the top, maintaining balance
- What if parent is full?
  - Do the same thing
- Overflow cascade all the way up to the root
  - still at most $O(\log n)$



(2,4) Deletion

- First of all, find the key
  - simple multi-way search
- Then, reduce to the case where item to be deleted is at the bottom of the tree
  - Find item which precedes it in in-order traversal
  - Swap them
- Remove the item

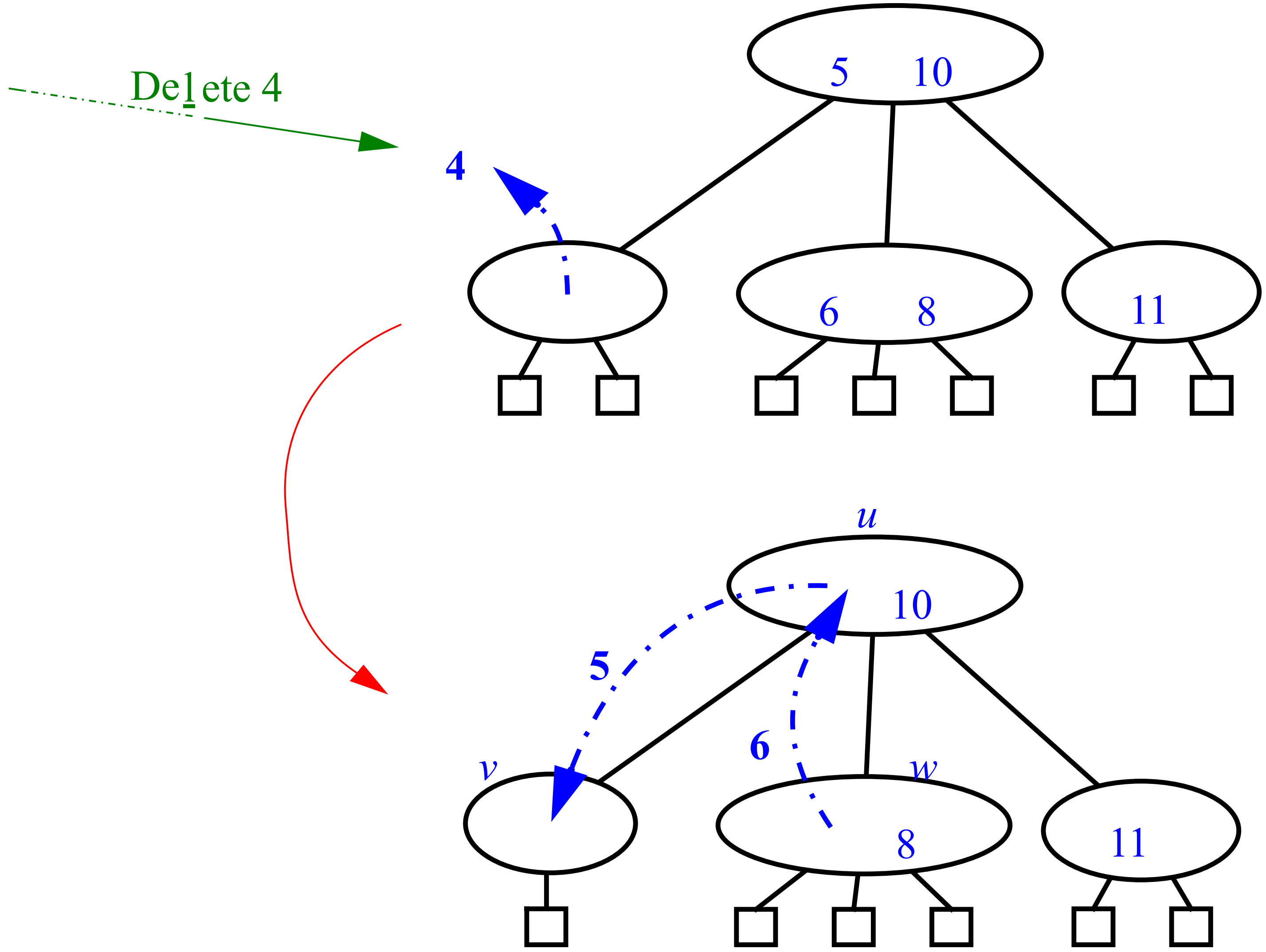
(2,4) Deletion



(2,4) Deletion

- Removing from 2-nodes
- Not enough items in the node
    - ***underflow***
- Pull an item from the parent, replace it with an item from a sibling
    - **called *transfer***

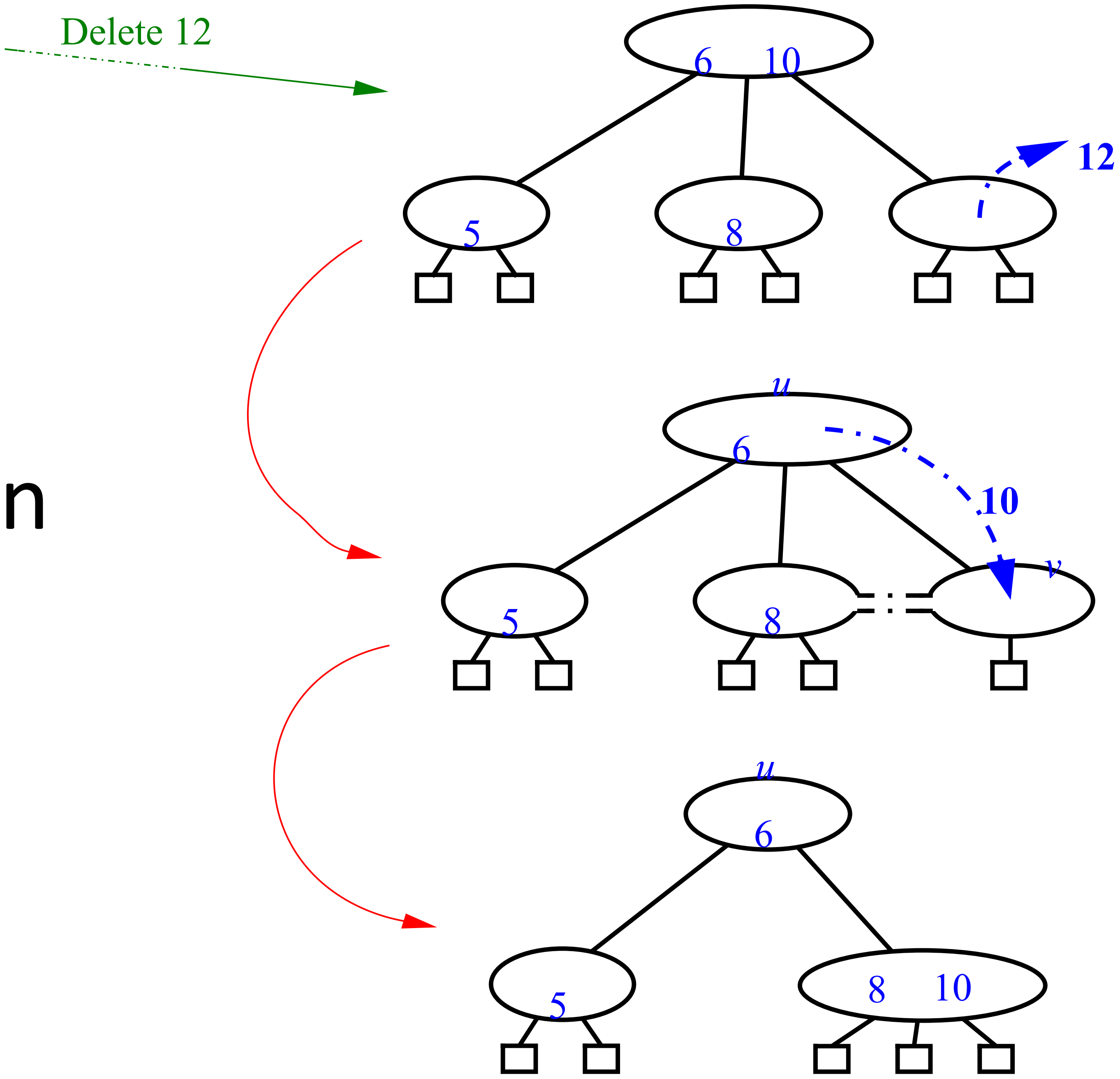
(2,4) Deletion



(2,4) Deletion

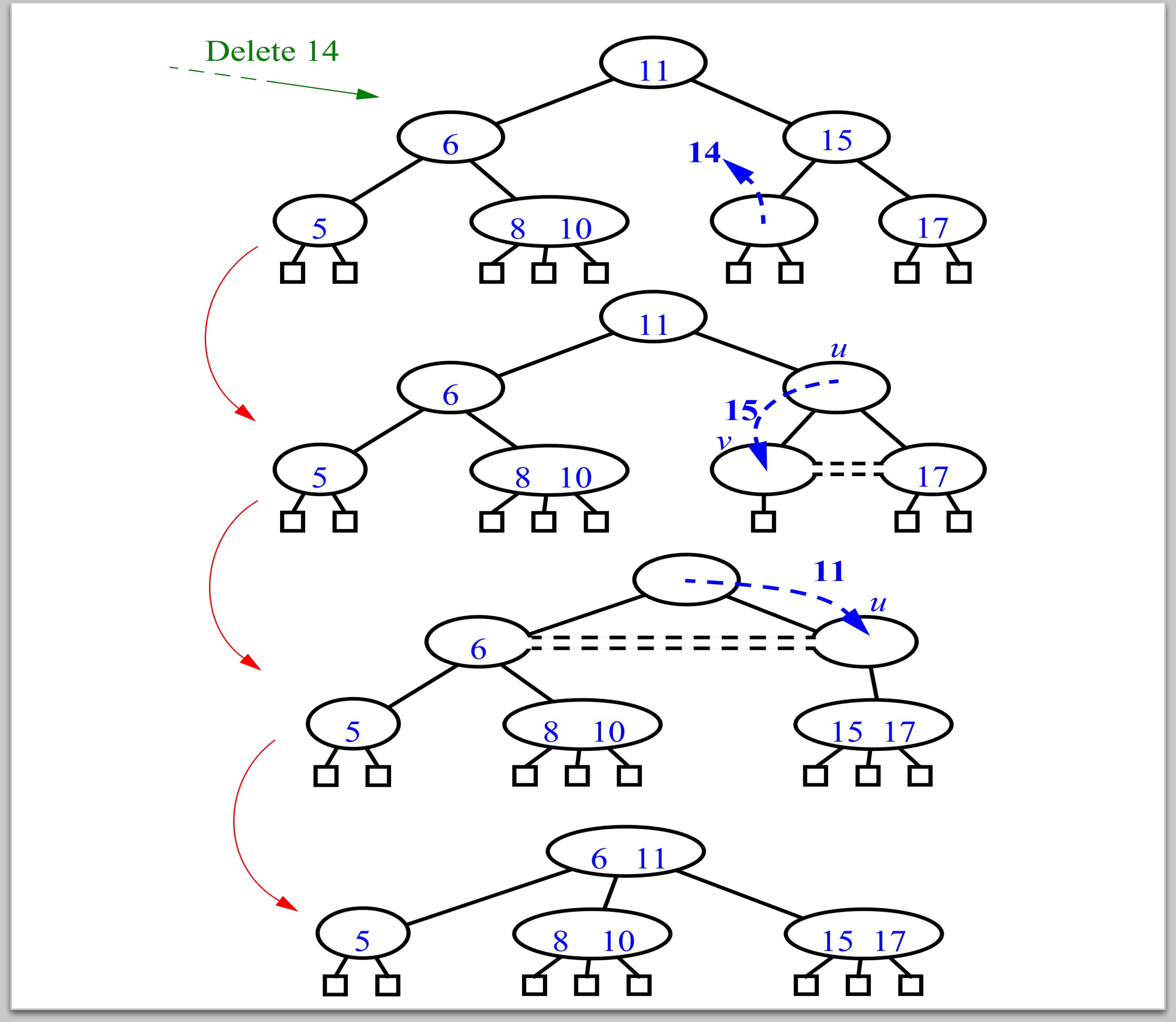
- What happens if siblings are 2-nodes?
- Could we just pull one item from the parent?
  - too many children
- But maybe...
- We know that the node's sibling is just a 2-node
- So we *fuse* them into one after removing an item from the parent,

(2,4) Deletion



(2,4) Deletion

- what if the parent was a2-node?
- Underflow can cascade up the tree, too.



(2-4) Trees Conclusion

- The height of a (2,4) tree is $O(\log n)$.
- Split, transfer, and fusion each take $O(1)$.
- Search, insertion and deletion each take $O(\log n)$.