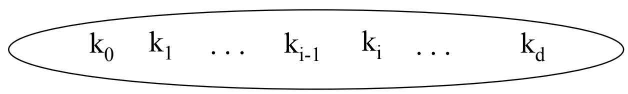
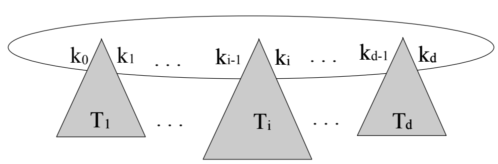
Multiway Search Trees

- Each internal node v of a multi-way search tree T
  - has **at least two children**
  - contains d-1 items, where
    - d is the number of children of v
    - an item is of the form (k_i,x_i) for $1 \le i \le d-1$, where
      - k_i is a key such that $k_i \le k_{i+1}$ for $1 \le i < d-1$
      - x_i is an element
  - "contains" two pseudo-items $k_0 = -\infty$ and $k_d = +\infty$



Multiway-Search Trees (cont'd)

- Children of each internal node are "between" the items in that node.
  - If T_i is the subtree rooted at child v_i, then all keys in T_i fall between the keys k_{i-1} and k_i, that is, $k_{i-1} \le T_i \le k_i$



- As before, external nodes are just place holders.

Height of Multiway Search Trees

- **Proposition:** A multi-way search tree T storing n items has n+1 external nodes.
- **Proof:** By induction over the height of T
  - **Induction base:** height(T)=1
    - Must have a single node with n items, with all subtrees of height 0. By our definition, there are n+1 subtrees and thus external nodes.

Height of Multiway Search Trees (cont'd)

- **Induction step:** height(T)>1
  - Let the root node store m items
  - By the definition of a multi-way node, it has m+1 subtrees
  - **By induction**, each subtree T_i, $1 \le i \le m+1$, has p_i items and $p_i + 1$ external nodes.
  - We thus know that T holds a total of

    $$\mathbf{X} = (m + \Sigma_{1 \le i \le m+1} p_i)$$

    items
  - … and that T holds a total of

    $$\mathbf{Y} = \Sigma_{1 \le i \le m+1} (p_i+1)$$

    external nodes (i.e., add up all external nodes of all subtrees).
  - $\mathbf{Y} = \Sigma_{1 \le i \le m+1} (p_i+1) = (m+1) + \Sigma_{1 \le i \le m+1} p_i = \mathbf{X} + 1$

Q.E.D.

Searching in Multiway Search Trees

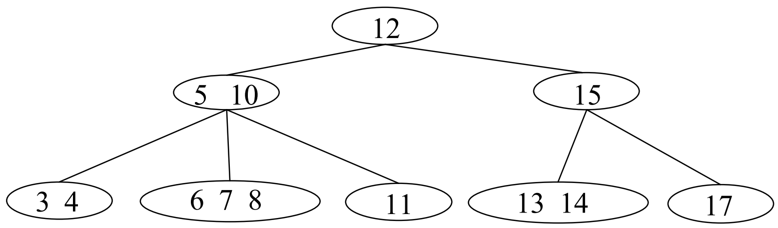
- The obvious **generalization** of searching in a binary search tree.
- Let s be the search key:
  - If $\mathbf{s < k_1}$, then search the **leftmost** child
  - If $\mathbf{s > k_{d-1}}$, then search **rightmost** child
  - Else, find two keys $\mathbf{k_{i-1}}$ and $\mathbf{k_i}$ such that s falls **between** them and then search the child $\mathbf{v_i}$
- A search for a key in a multi-way tree will be **O(height)**

Special Cases of Multiway Search Trees

- Definition of Multiway Search Tree is very general
- We will look at two **special cases**:
  - (2,4) Trees
  - B-Trees

(2,4) Trees

- A (2,4) tree is a **multi-way search tree** such that
  - **Size property:** Every internal node has at most four children
  - **Depth property:** All external nodes have the same depth



- Size property offers us **just enough flexibility** to enforce the rather strict depth property
- Both properties together ensure that (2,4) trees have **logarithmic height**

Height of (2,4) Trees

- **Proposition:** The height h of a (2,4) tree is Θ(log n).
- **Proof:**

Let T be (2,4) tree with n items, m = n+1 external nodes.

By size and depth properties:

$$2^h \le m \le 4^h$$

By external node proposition of multi-way trees, we have m = n+1.

Thus,

$$2^h \le n+1 \le 4^h$$

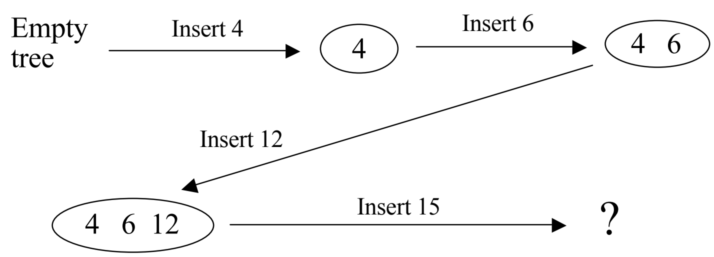
$$h \le \log(n+1) \le 2h$$

$$\_ \log(n+1) \le h \le \log(n+1)$$

Thus, h is Θ(log n). QED

(2,4) Tree Insertion

- Let's start with an example:



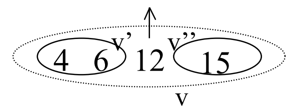
- A node may **overflow** if the node already stores 3 items and you try to insert another item into it.

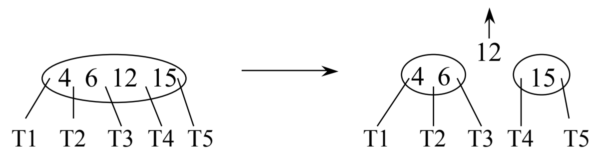
(2,4) Tree Insertion (cont'd)

- Use **search** to find location in tree where new key is to be placed
- **Case**: Search stops successfully
  - key already in tree
  - done
- **Case**: Search stops unsuccessfully at node v
  - Then, v must be "at bottom" of tree, that is, have empty children only
  - **Subcase**: **No overflow**, ie, v has less than 3 keys
    - insert new key
    - Done. New tree is (2,4) tree
  - **Subcase**: **Overflow**, *i.e.*, v has 3 keys
    - temporarily insert key
    - since v now violates size property, perform **split** operation

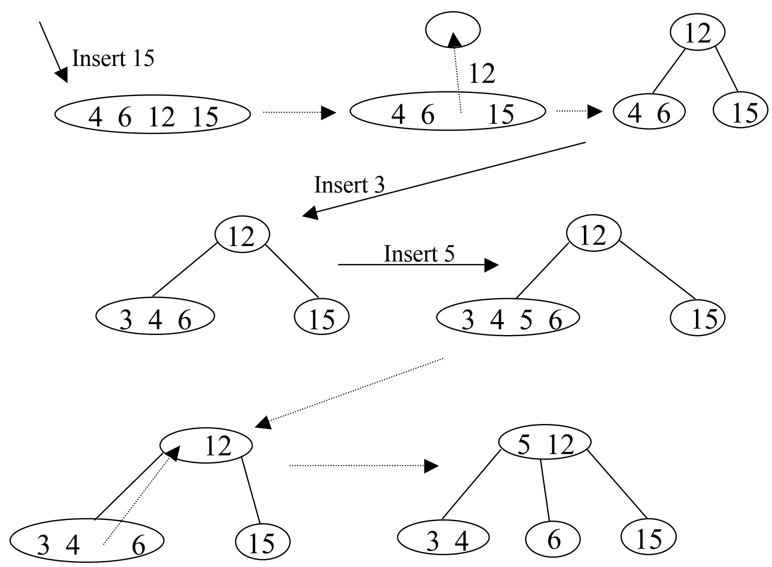
(2,4) Tree Insertion: Split

- Must perform a **split** operation if a node v overflows:
  - **Replace node v with two nodes v' and v''** where
    - v' gets the first two keys
    - v'' gets the last key
  - **Send the other (third) key up the tree**
    - If v is root, create new root node v''' that stores the third key and make v' the left child of v''' and v'' the right child.
    - Otherwise, add third key to the parent of v.





(2,4) Tree Insertion: Example 1



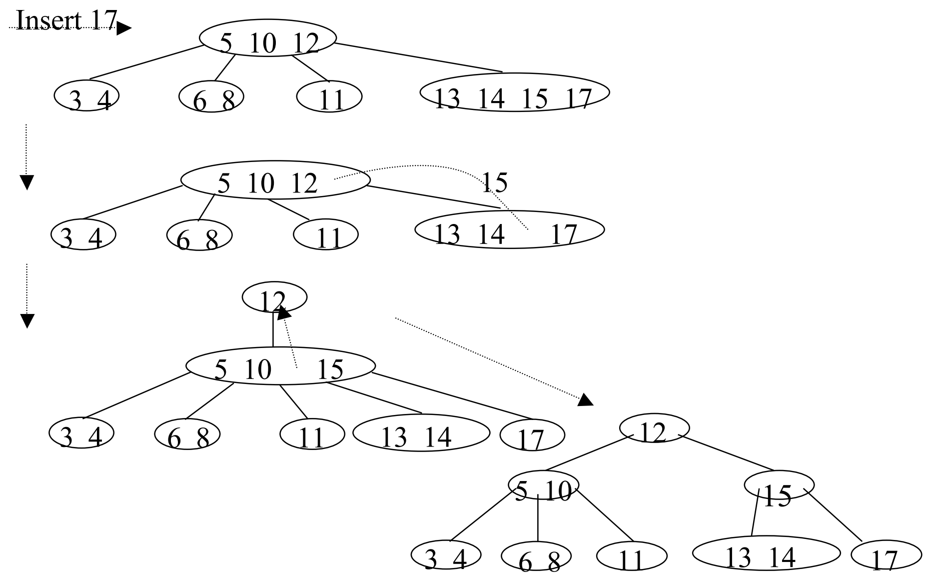
(2,4) Tree Insertion: Split (cont'd)

- A split operation is O(1)
- A split operation on a (2,4) tree retains the (2,4) tree properties:
  - **size** property: new nodes have 2 – 4 children
  - **depth** property: depth of **all** leaves grows by 1
  - **order property:** key values in children increase from left to right.

(2,4) Tree Insertion: Split (cont'd)

- If parent is already full (i.e., has 3 keys) then overflow may propagate, requiring another split operation at a higher level all the way to the root.
- Since height is O(log n), one `insert` can cause at most O(log n) overflows.

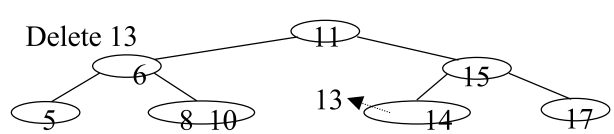
(2,4) Tree Insertion: Example 2



(2,4) Tree Deletion

- Use **search** to find key k in tree
- **Case**: Search stops unsuccessfully
  - key not in tree. Done
- **Case**: Search stops successfully at node v
  - **Subcase**: v is **not** a leaf, *i.e.*, v has nonempty children
    - find key k' the in-order successor of k
    - swap k' and k (k is now in a leaf node we call v)
    - remove one item and one child from v.
  - **Subcase**: v is a leaf, *i.e.*, v has only empty children
    - remove one item and one child from v

Delete 13



(2,4) Tree Deletion: Underflow

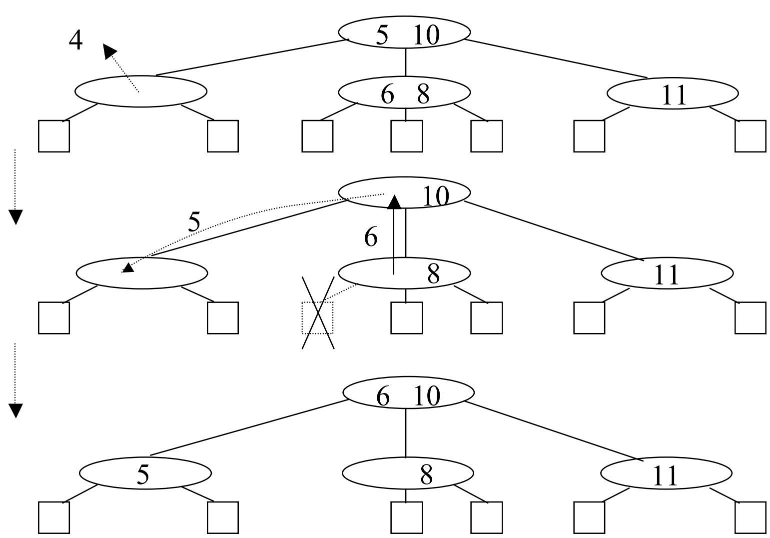
- After deletion if v is left with no items and one child we say that an **underflow** occurs.
- We resolve an underflow in one of two ways.
- **Case 1**: A sibling immediately to the left or to the right of v has more than one item

  → Perform a **transfer** operation.
- **Case 2**: The immediate siblings of v all have exactly one item

  → Perform a **fusion** operation.

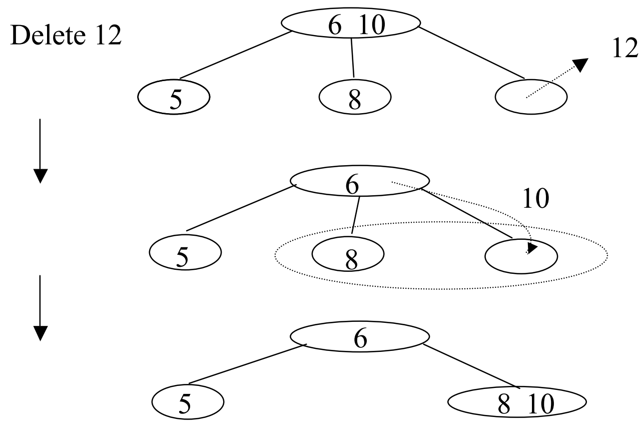
(2,4) Tree Deletion: Transfer Example 1

`Delete 4`

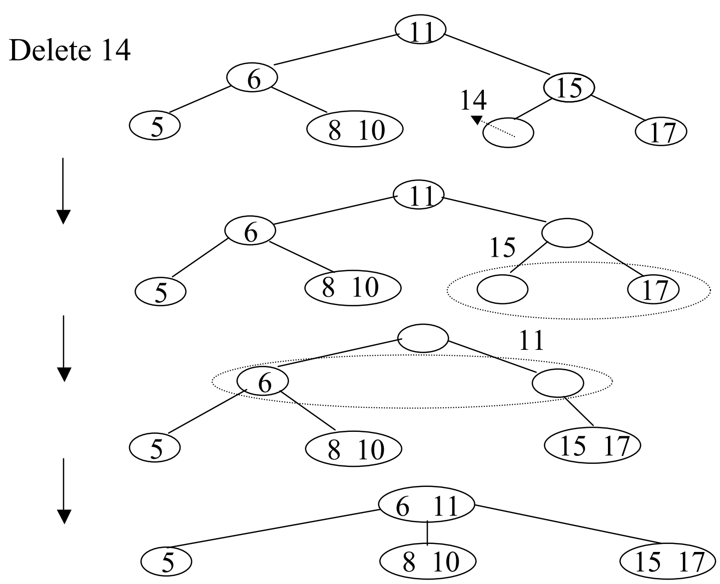


(2,4) Tree Deletion: Fusion Example 1

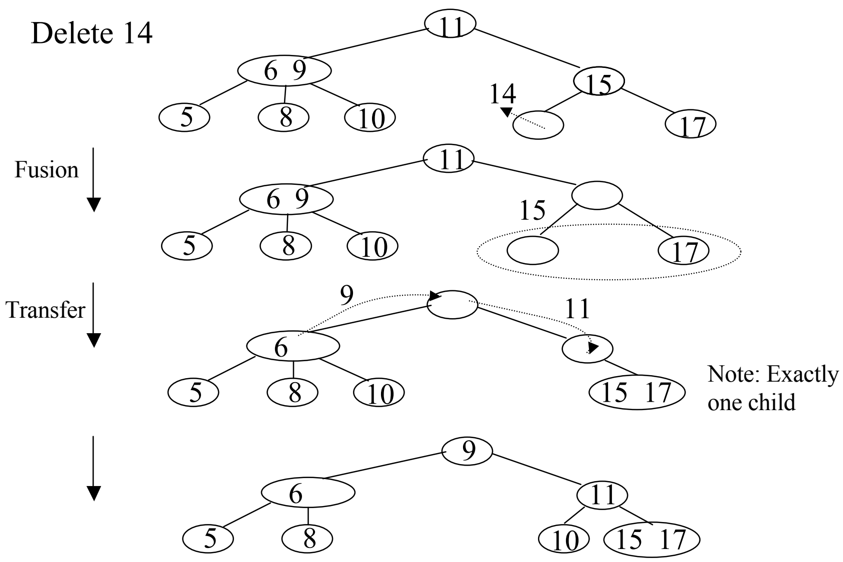
- We know that the node's sibling is a 2-node, so we **fuse** them into one node.



(2,4) Tree Deletion: Fusion Example 2



(2,4) Tree Deletion: Fusion + Transfer Example



(2,4) Tree Deletion: Transfer

- **Transfer** operation: Immediate sibling s of underflow node v has more than one key
  - Take key from parent of v which is between v and s, and add that key to v
  - Replace missing key in the parent with key that is closest in value from the sibling s of v
  - s now stores one fewer keys, but still has an extra subtree
  - If transfer operation occurs when v is at bottom level, the extra subtree is empty and can just be deleted
  - If transfer operation occurs at higher level, **node v will always have exactly one child before the transfer**. We can thus move the extra subtree of s to v
- Transfer does not propagate underflow
- Transfer is O(1)

(2,4) Tree Deletion: Fusion

- **Fusion** operation: The immediate siblings of v have exactly one key
  - Choose an immediate sibling s of node v. Let's say that s contains the key k_s
  - Take key k_p from parent of v which is between v and s, and add that key to v
  - Merge v and s into single new node that contains the single key now in v and the single key from s (i.e., k_s and k_p)
  - The parent now has one fewer keys and one fewer subtrees
  - If parent previously had only one key stored, then it now also suffers from underflow, and contains exactly one subtree
  - If fusion propagates to root, then remove root and make new merged node the new root
- Fusion is O(1), but may **propagate underflow**

(2,4) Tree Deletion Algorithm

Algorithm to delete k:

0. **Search** for k
1. If necessary, **swap** k with inorder successor to bottom of tree into node v such that v has only external children
2. **Delete** k from v
3. If v does not underflow, then **done**
4. If v **underflows**, then
   a) if v root, replace v with its child. **Done**
   b) else **pull key down** from parent into v
5. If an immediate sibling is 3- or 4-node, then
   a) **transfer** key up to parent from sibling. Move subtree if necessary. **Done**
   b) else
   - **fuse** node v and its sibling
   - if underflow in parent, then repeat from step 4 using parent as node v, else **done**

(2,4) Tree Deletion Algorithm (cont'd)

```
delete(Key k) :

   v = search(k);
   if (v not leaf) {
     swap k with successor key;
     v = node of successor key;
     delete k from v
   }
   // v is leaf
   if v does not underflow, then done;
   else handleUnderflow(v);
```


(2,4) Tree Deletion Algorithm (cont'd)

```
handleUnderflow(node v):
  // called when v is underflowing, ie, v is 1-node
  if v is the root
    replace it with its one child
  else if v has left sibling & is a 3 or 4-node
    transfer(v, v's left sibling)
  else if v has right sibling & is 3 or 4-node
    transfer(v, v's right sibling)
  else
    // must do a fusion
    if v has a left sibling
      fuse(v's left sibling, v)
    else
      fuse(v, v's right sibling)
    if fusion made parent into a one-node
       handleUnderflow(parent)
```

(2,4) Tree Deletion (cont'd)

- Underflow may **propagate** all the way to root.
- Underflow only propagates when a **fusion** results in an empty parent.
- **Fusion** resulting in a 2- or 3-node parent or a **transfer** will complete the deletion process.
- That is, deletion involves a (possibly empty) sequence of fusion operations followed (possibly) by a transfer operation
- **Interesting fact:**
  - Can always do **fusion** (possibly followed by a **split**) instead of **transfer**
  - If only use **fusions** and no **transfer**, still get O(log n) but with larger constant factor

(2,4) Tree Deletion Complexity

- **Find** key to remove: O(h)
- **Swap** with successor if not leaf: O(h)
- **Transfer** or **fuse** if underflow: O(1)
- Maximal number of underflows: O(h)
- Thus, since h is O(log n), complexity of deletion is O(log n)

(2,4) Tree Summary

- Height of a (2,4) tree is **Θ(log n)**
- Helper functions: split, transfer, and fusion (all take O(1))
- **`Search`**, **`insert`**, and **`delete`** each take O(log n)
  - **Advantages** of (2,4) trees over AVL trees:
    - Easier to understand and implement (no rotations).
    - Tree may have fewer nodes.
  - **Disadvantages** of (2,4) trees over AVL trees:
    - Both trees perform dictionary operations in O(log n) worst case time. However, AVL trees are more efficient by a constant factor.
    - (2,4) trees use 3 different types of node (*i.e.* 2, 3, or 4 children).