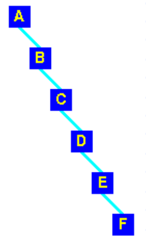
Problem of BST

- Binary Search is not balanced.
- Take this list of characters and form a tree

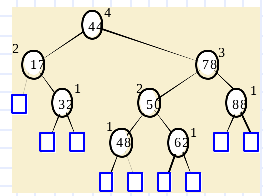
A B C D E F



BST degenerates to a linked list

AVL Tree

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*



An example of an AVL tree where the heights are shown next to the nodes:

Height of an AVL Tree

- **Proposition**: The ***height*** of an AVL tree T storing n keys is O(log n).
- **Justification**: The easiest way to approach this problem is to find **n(h):** the *minimum number of internal nodes* of an AVL tree of height h.
- We see that n(1) = 1 and n(2) = 2
- For n = 3, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and the other AVL subtree of height n-2.
- i.e. n(h) = 1 + n(h-1) + n(h-2)

Height of an AVL Tree (cont)

- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2)

  n(h) > 2n(h-2)
  n(h) > 4n(h-4)
  n(h) > 8n(h-6)
  ...
  n(h) > 2^in(h-2i)
  For any integer i such that h-2i $\geq$ 1
  Let h – 2i = 1, then i = (h – 1)/2
- Solving the base case we get: n(h) = $2^{(h-1)/2}$
- Taking logarithms: h < 2log n(h) + 1
- Thus the height of an AVL tree is O(log n)

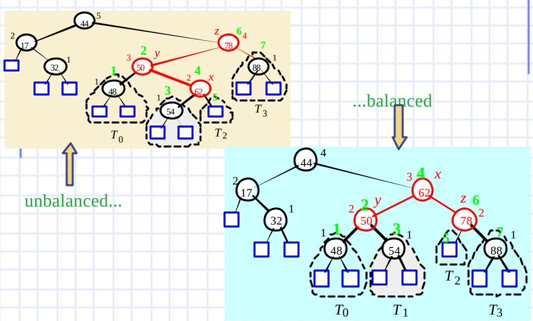
Insertion

- A binary search tree T is called ***balanced*** if for every node v, the height of v's children differ by at most one.
- Inserting a node into an AVL tree changes the heights of some of the nodes in T.
- If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.
- Since z became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2
- Now to rebalance...

Insertion: rebalancing

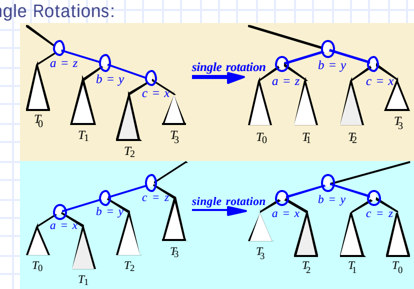
- To rebalance the subtree rooted at z, we must perform a *restructuring*
- we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

Insertion (contd.)



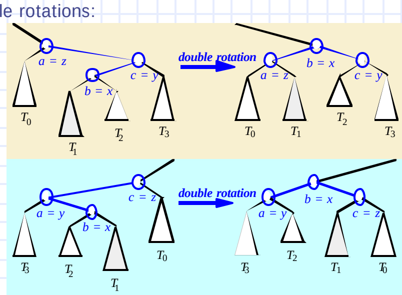
Restructuring

- The four ways to rotate nodes in an AVL tree, graphically represented
  - Single Rotations:



Restructuring (contd.)

- double rotations:



Restructure Algorithm

function restructure(x):

Input: A node *x* of a binary search tree T that has both a parent *y* and a grandparent *z*

Output: Tree T restructured by a rotation (either single or double) involving nodes x, y, and z.

1: Let (*a, b, c*) be an inorder listing of the nodes *x, y, and z*, and let (T0, T1, T2, T3) be an inorder listing of the the four subtrees of *x, y, and z*.

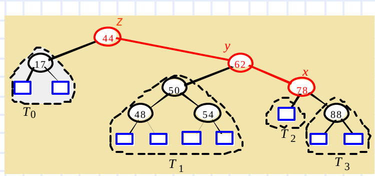
2. Replace the subtree rooted at z with a new subtree rooted at *b*

3. Let *a* be the left child of *b* and let T0, T1 be the left and right subtrees of *a*, respectively.

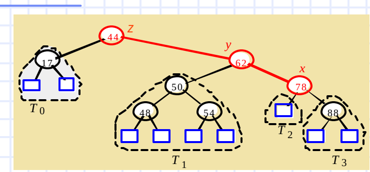
4. Let *c* be the right child of *b* and let T2, T3 be the left and right subtrees of *c*, respectively.

Cut/Link Restructure Algorithm

- Let's go into a little more detail on this algorithm…
- Any tree that needs to be balanced can be grouped into 7 parts: x, y, z, and the 4 trees anchored at the children of those nodes (T0-T3)



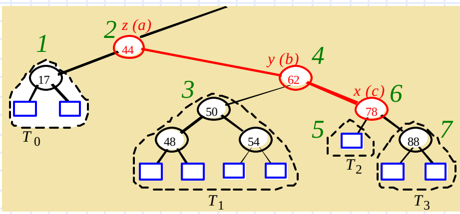
Cut/Link Restructure Algorithm



- Make a new tree which is balanced and put the 7 parts from the old tree into the new tree so that the numbering is still correct when we do an in-order traversal of the new tree.
- This works regardless of how the tree is originally unbalanced.
- Let's see how it works!

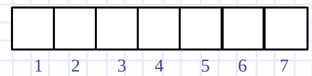
Cut/Link Restructure Algorithm

- Number the 7 parts by doing an in-order-traversal. (note that x,y, and z are now renamed based upon their order within the traversal)

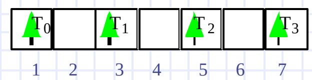


Cut/Link Restructure Algorithm

- Now create an Array, numbered 1 to 7 (the 0th element can be ignored with minimal waste of space)

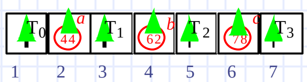


•Cut() the 4 T trees and place them in their inorder rank in the array



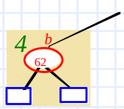
Cut/Link Restructure Algorithm

- Now cut x,y, and z in that order (child,parent,grandparent) and place them in their inorder rank in the array.



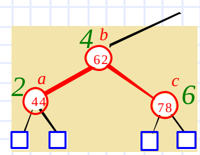
•Now we can re-link these subtrees to the main tree.

•Link in rank 4 (b) where the subtree's root formerly



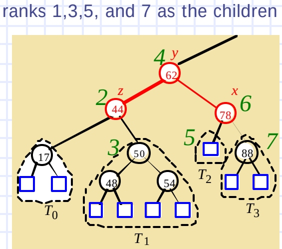
Cut/Link Restructure Algorithm

- Link in ranks 2 (a) and 6 (c) as 4's children.



Cut/Link Restructure Algorithm

- Finally, link in ranks 1,3,5, and 7 as the children of 2 and 6.



- Now you have a balanced tree!

Cut/Link Restructure algorithm

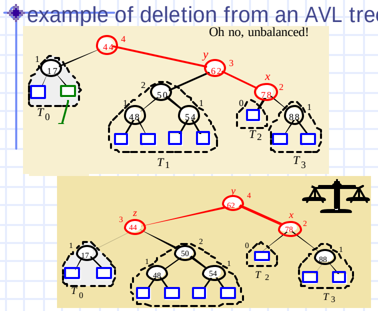
- This algorithm for restructuring has the exact same effect as using the four rotation cases discussed earlier.
- Advantages: no case analysis, more elegant
- Disadvantage: can be more code to write
- Same time complexity

Removal

- We can easily see that performing a remove(w) can cause T to become unbalanced.
- Let z be the first unbalanced node encountered while traveling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- We can perform operation restructure(x) to restore balance at the subtree rooted at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

Removal (contd.)

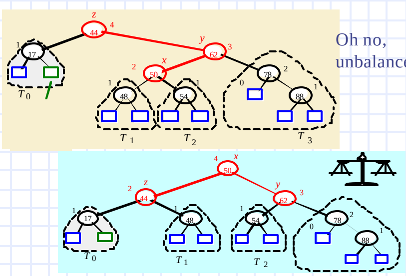
- example of deletion from an AVL tree:



Whew, balanced!

Removal (contd.)

example of deletion from an AVL tree:



Oh no, unbalanced!

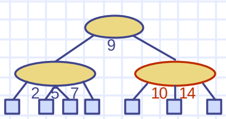
Whew, balanced!

AVL Trees - Data Structures

- AVL trees can be implemented with a flag to indicate the balance state

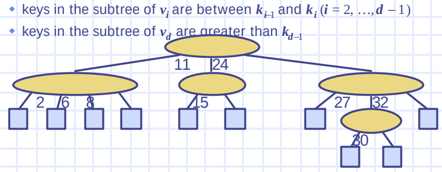
```
typedef enum {LeftHeavy, Balanced,
              RightHeavy} BalanceFactor;

typedef struct node {
      BalanceFactor bf;
      void *item;
      struct node *left, *right;
} AVL_node;
```


(2,4) Trees



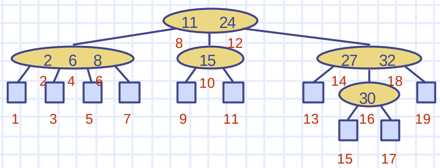
Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
  - For a node with children $v_1 v_2 \ldots v_d$ storing keys $k_1 k_2 \ldots k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d-1)$
    - keys in the subtree of v_d are greater than k_{d-1}



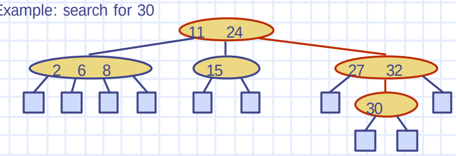
Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order



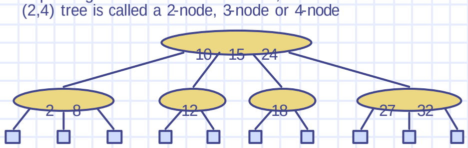
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1 v_2 \ldots v_d$ and keys $k_1 k_2 \ldots k_{d-1}$
  - $k = k_i$ ($i = 1, \ldots, d-1$): the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i$ ($i = 2, \ldots, d-1$): we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Reaching an external node terminates the search unsuccessfully
- Example: search for 30



(2,4) Tree

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

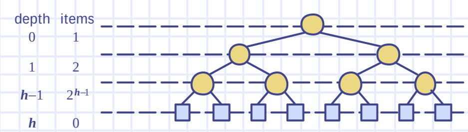


Height of a (2,4) Tree

- Theorem: A (2,4) tree storing n items has height $O(\log n)$

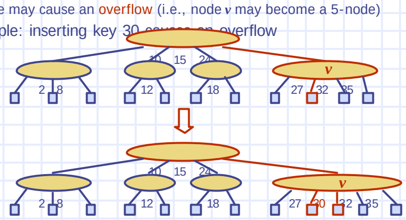
  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have

    $$n \ge 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \le \log(n+1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time



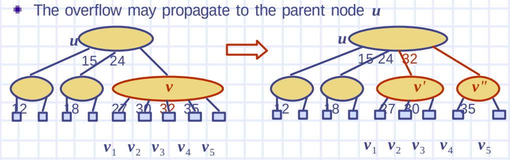
Insertion

- We insert a new item (k, o) at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 30 causes an overflow



Overflow and Split

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v is replaced nodes v' and v''
    - v' is a 3-node with keys k_1 k_2 and children v_1 v_2 v_3
    - v'' is a 2-node with key k_4 and children v_4 v_5
  - key k_3 is inserted into the parent u of v (a new root may be created)
- The overflow may propagate to the parent node u



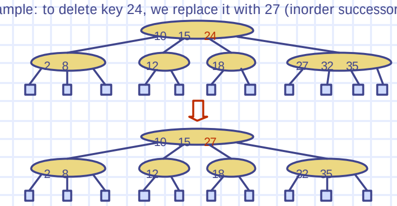
Analysis of Insertion

```
function insertItem(k, o)

1. We search for key k to locate the
   insertion node v

2. We add the new item (k, o) at node v

3. while (overflow(v)) {
     if (isRoot(v))
        create a new empty root above v;
     v ← split(v) // return parent of v;
   }
```

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
  - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2 takes $O(1)$ time
  - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

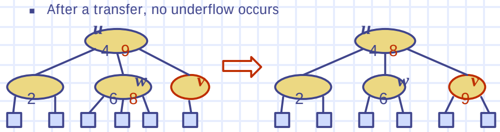
Deletion

- We reduce deletion of an item to the case where the item is at the node with leaf children
- Otherwise, we replace the item with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter item
- Example: to delete key 24, we replace it with 27 (inorder successor)



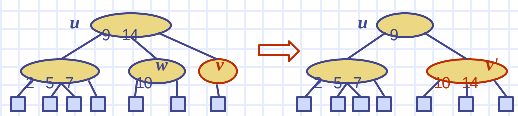
Underflow and Transfer

- Deleting an item from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys
- To handle an underflow at node v with parent u, we consider two cases
- Case 1: an adjacent sibling w of v is a 3-node or a 4-node
  - Transfer operation:
    1. we move a child of w to v
    2. we move an item from u to v
    3. we move an item from w to u
  - After a transfer, no underflow occurs



Underflow and Fusion

- Case 2: the adjacent siblings of v are 2-nodes
  - Fusion operation: we merge v with an adjacent sibling w and move an item from u to the merged node v'
  - After a fusion, the underflow may propagate to the parent u



Analysis of Deletion

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the item
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time