Trees

Chapter 15

Chapter Contents

Chapter Objectives

- Show how binary trees can be used to develop efficient codes
- Study problem of binary search trees becoming unbalanced
  - Look at classical AVL approach for solution
- Look at other kinds of trees, including 2-3-4 trees, red-black trees, and B-trees
- Introduce the associate containers in STL and see how red-black trees are used in implementation

Case Study: Huffman Codes

- Recall that ASCII, EBCDIC, and Unicode use same size data structure for all characters
- Contrast Morse code
  - Uses variable-length sequences
- Some situations require this variable-length coding scheme

Variable-Length Codes

- Each character in such a code
  - Has a weight (probability) and a length
- The expected length is the sum of the products of the weights and lengths for all the characters

character	A	B	C	D	E
weight	0.2	0.1	0.1	0.15	0.45

Character	Code
A	01
B	1000
C	1010
D	100
E	0

0.2 x 2 + 0.1 x 4 +
0.1 x 4 + 0.15 x 3 + 0.45 x 1 = 2.1

Immediate Decodability

- When no sequence of bits that represents a character is a prefix of a longer sequence for another character
  - Can be decoded without waiting for remaining bits

Character	Code
A	01
B	1000
C	1010
D	100
E	0

- Note how previous scheme is <u>not</u> immediately decodable
- And this one is

Character	Code
A	01
B	1000
C	0001
D	001
E	1

Huffman Codes

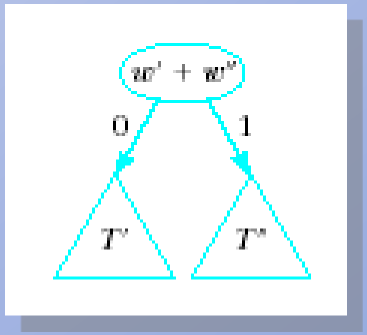
- We seek codes that are
  - Immediately decodable
  - Each character has minimal expected code length
- For a set of n characters $\{ C_1 \ .. \ C_n \}$ with weights $\{ w_1 \ .. \ w_n \}$
  - We need an algorithm which generates n bit strings representing the codes

Huffman's Algorithm

1. Initialize list of **n** one-node binary trees containing a weight for each character
2. Do the following **n – 1** times

   a. Find two trees **T'** and **T"** in list with minimal weights **w'** and **w"**

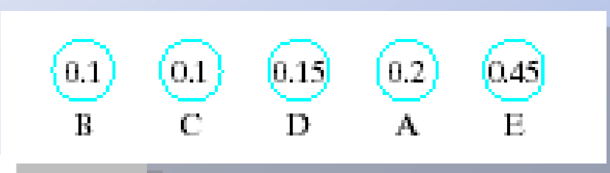
   b. Replace these two trees with a binary tree whose root is **w' + w"** and whose subtrees are **T'** and **T"** and label points to these subtrees 0 and 1



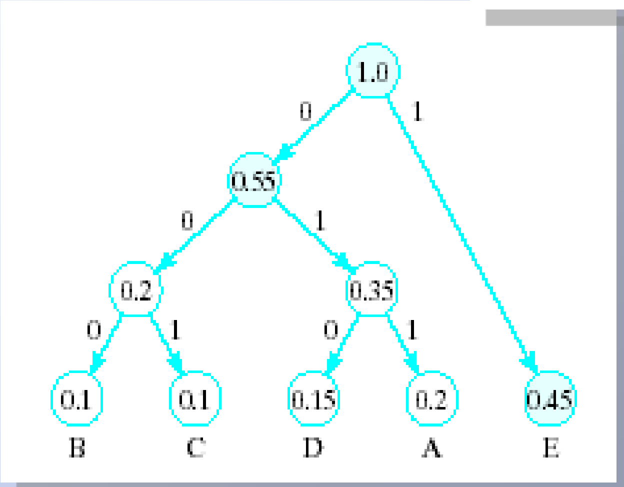
Huffman's Algorithm

3. The code for character C_i is the bit string labeling a path in the final binary tree form from the root to C_i

Given characters



The end result is



with codes

Character	Huffman Code
A	01
B	0000
C	0001
D	001
E	1

Huffman Decoding Algorithm

1. Initialize pointer `p` to root of Huffman tree
2. While end of message string not reached do the following

   a. Let `x` be next bit in string

   b. if `x = 0` set `p` equal to left child pointer
      else set `p` to right child pointer

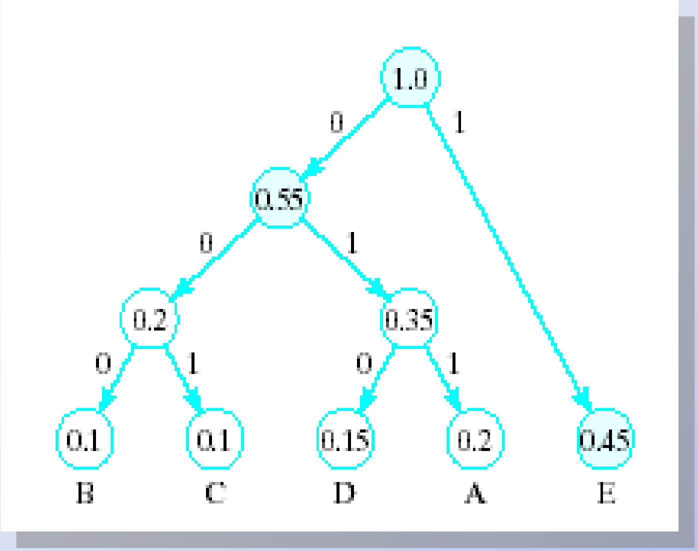
   c. If `p` points to leaf

      i. Display character with that leaf

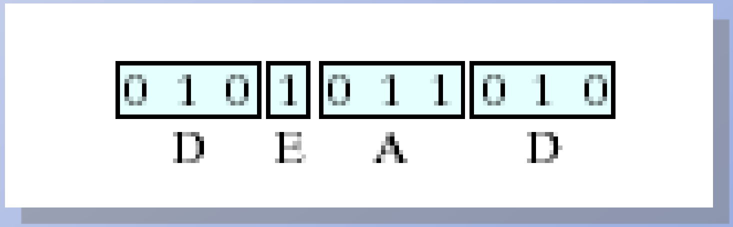
      ii. Reset `p` to root of Huffman tree

Huffman Decoding Algorithm

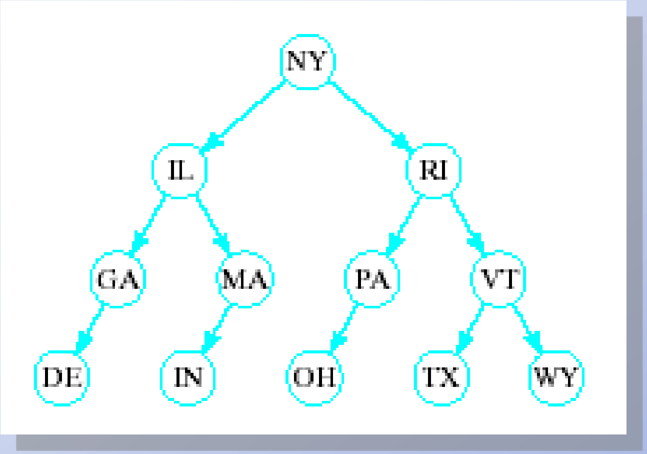
- For message string 0101010010 10
  - Using Hoffman Tree and decoding algorithm



Click for answer



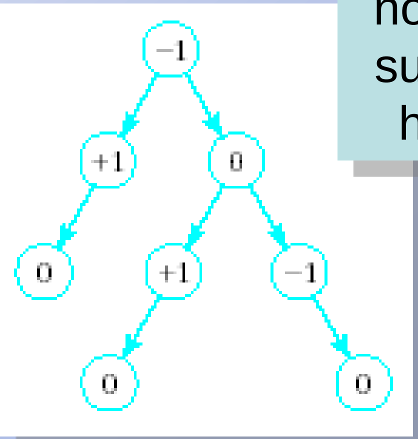
Tree Balancing: AVL Trees



- Consider a BST of state abbreviations
- When tree is nicely balanced, search time is $O(\log_2 n)$
- If abbreviations entered in order DE, GA, IL, IN, MA, MI, NY, OH, PA, RI, TX, VT, WY
  - BST degenerates into linked list with search time O(n)

AVL Trees

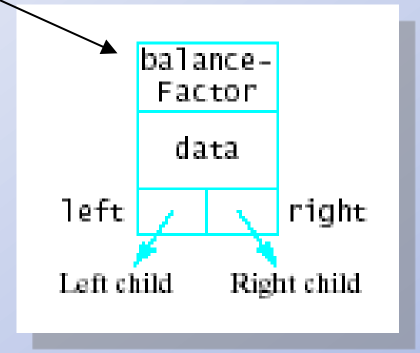
- A height balanced tree
  - Named with initials of Russian mathematicians who devised them
- Defined as
  - Binary search tree
  - Balance factor of each node is 0, 1, or -1



(Balance factor of a node = left height of sub tree minus right height of subtree)

AVL Trees

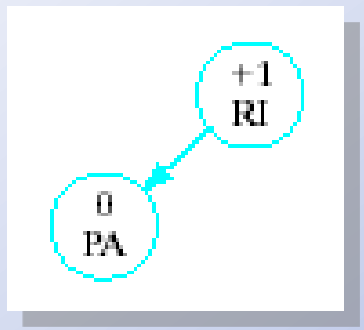
- Consider linked structure to represent AVL tree nodes
  - Includes data member for the balance factor

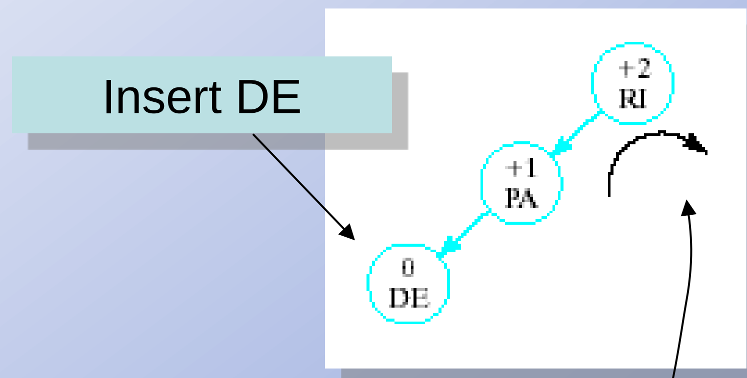


- Note source code for AVLTree class template

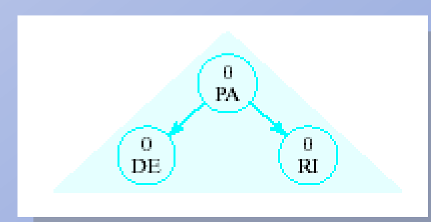
AVL Trees

- Insertions may require adjustment of the tree to maintain balance
  - Given tree





  - Becomes unbalanced
  - Requires right rotation on subtree of RI
  - Producing balanced tree



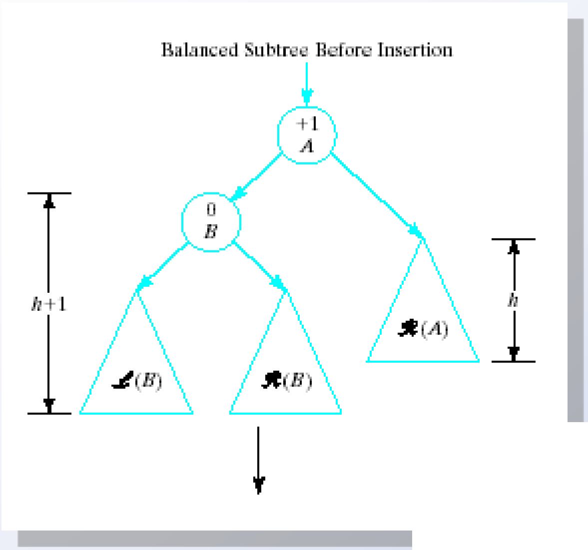
Basic Rebalancing Rotations

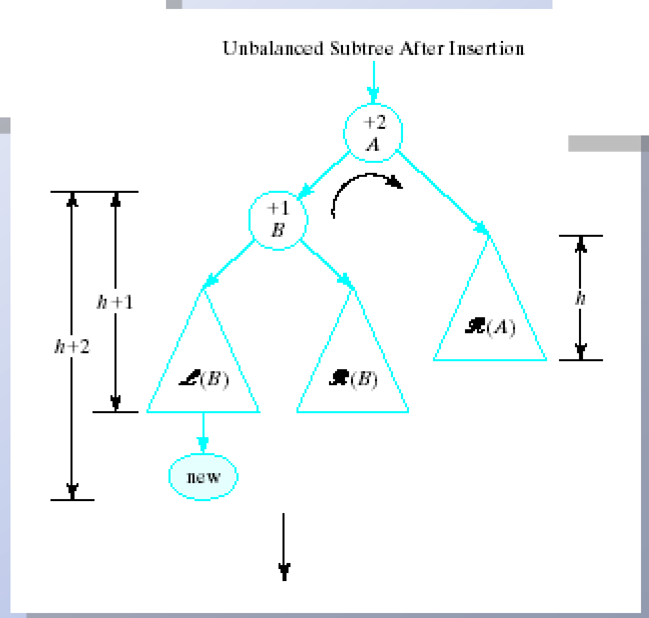
1. Simple right rotation
   When inserted item in left subtree of left child of nearest ancestor with factor +2
2. Simple left rotation
   When inserted item in right subtree of right child of nearest ancestor with factor -2
3. Left-right rotation
   When inserted item in right subtree of left child of nearest ancestor with factor +2
4. Right-left rotation
   When inserted item in left subtree of right child of nearest ancestor with factor -2

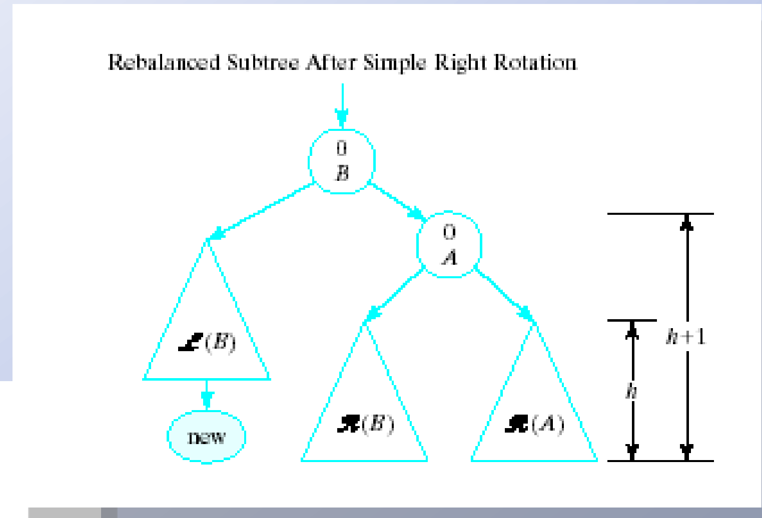
Basic Rebalancing Rotations

- Rotations carried out by resetting links
- Right rotation with A = nearest ancestor of inserted item with balance factor +2 and B = left child
  - Resent link from parent of A to B
  - Set left link of A equal to right link of B
  - Set right link of B to point A
- Next slide shows this sequence

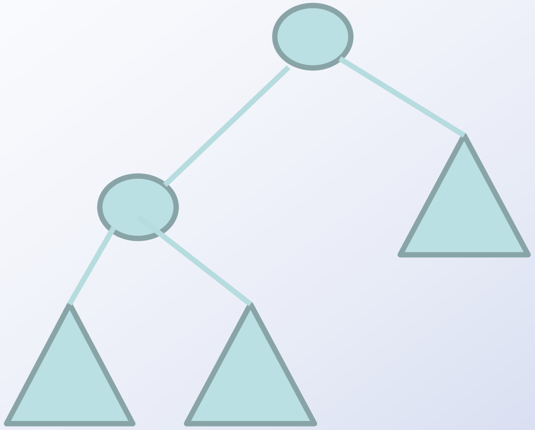
Basic Rebalancing Rotations

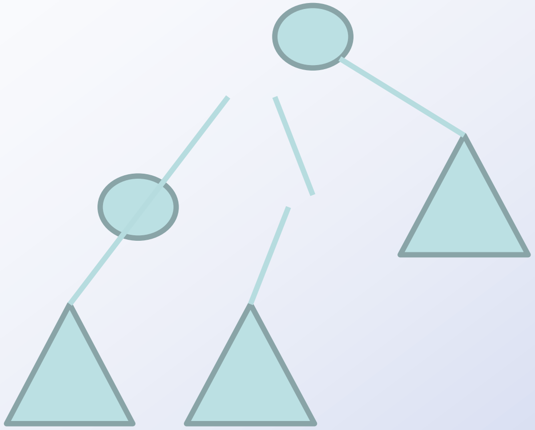






Rotation





Basic Rebalancing Rotations

Right-Left rotation

Item C inserted in right subtree of left child B of nearest ancestor A

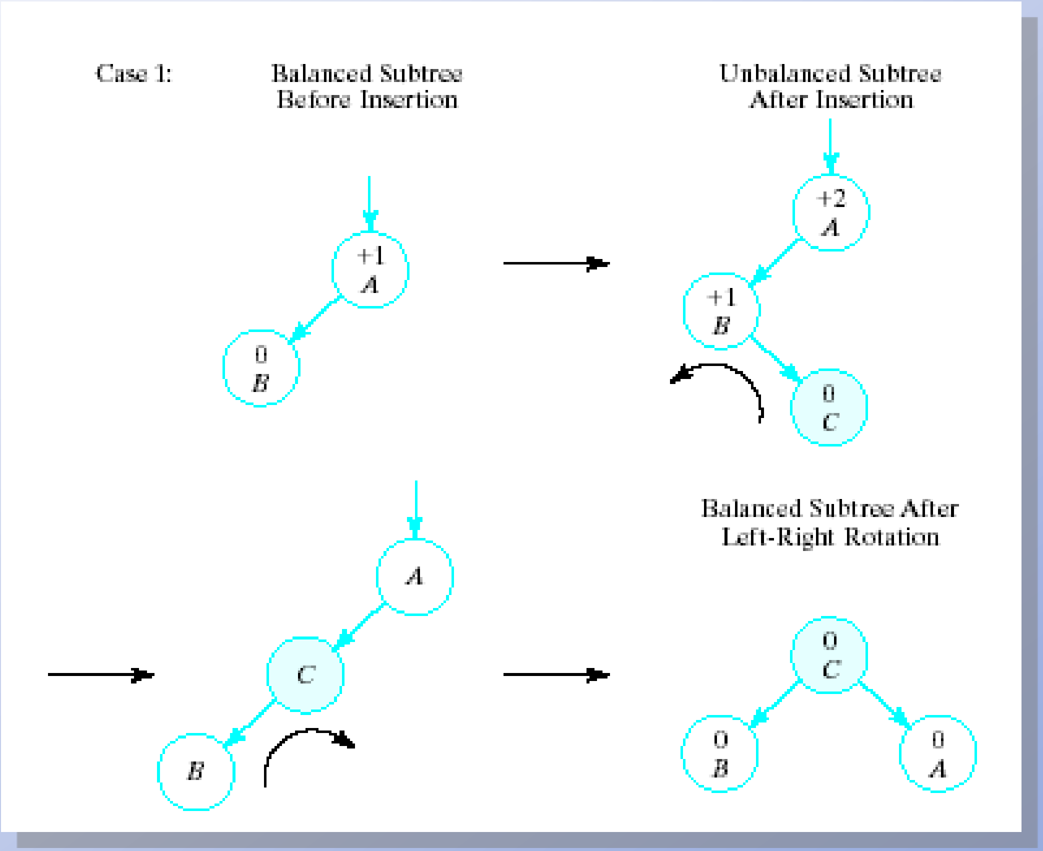
1. Set left link of A to point to root of C
2. Set right link of B equal to left link of C
3. Set left link of C to point to B

   Now the right rotation

4. Resent link from parent of A to point to C
5. Set link of A equal to right link of C
6. Set right link of C to point to A

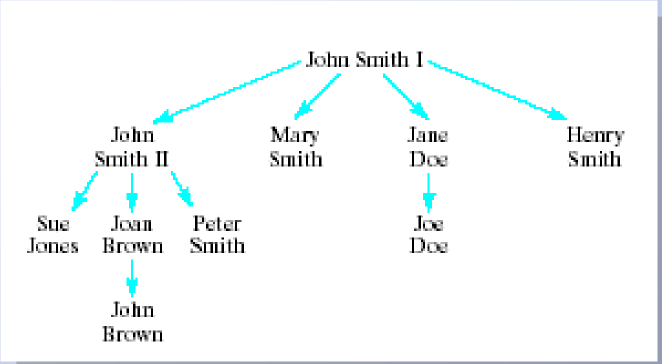
Basic Rebalancing Rotations

- When B has no right child before node C insertion

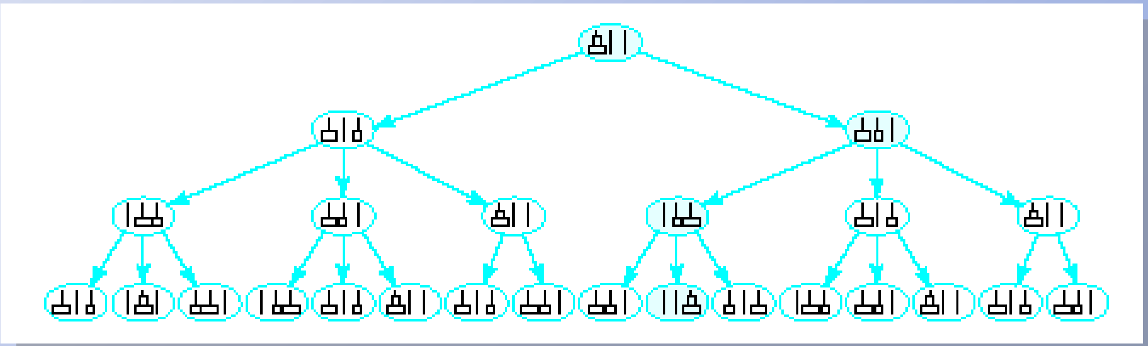


Non Binary Trees

- Some applications require more than two children per node
  - Genealogical tree



  - Game tree

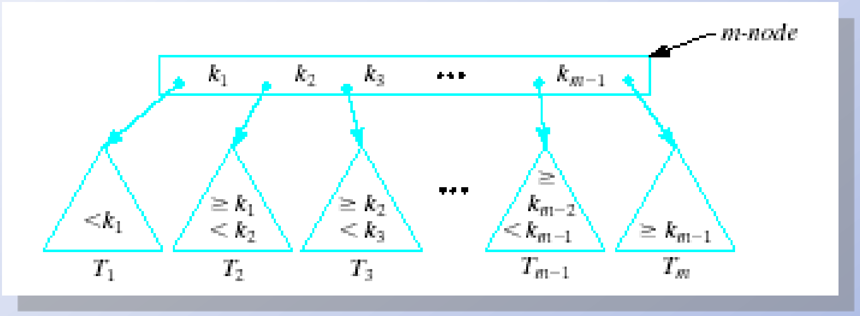


2-3-4 Trees

- We extend searching technique for BST
  - At a given node we now have multiple paths for a search path may follow
- Define m-node in a search tree
  - Stores $m - 1$ data values $k_1 < k_2 \ldots < k_{m-1}$
  - Has links to m subtrees $T_1 .. T_m$
  - Where for each i
    all data values in $T_i < k_i \leq$ all values in T_{k+1}

2-3-4 Trees

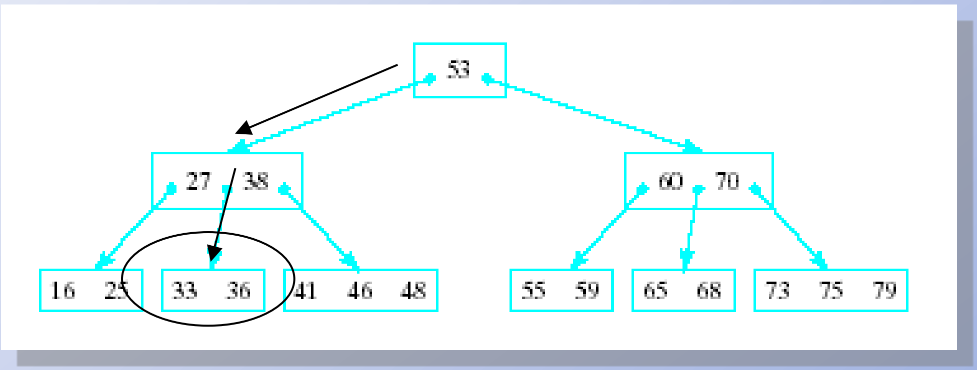
- Example of m-node



- Define 2-3-4 tree
  - Each node stores at most 3 data values
  - Each internode is a 2-node, a 3-node, or a 4-node
  - All the leaves are on the same level

2-3-4 Trees

- Example of a 2-3-4 node which stores integers



- To search for 36
  – Start at root … 36 < 53, go left
  – Next node has 27 < 36 < 38. take middle link
  – Find 36 in next lowest node

Building a Balanced 2-3-4 Tree

If tree is empty

- Create a 2-node containing new item, root of tree

Otherwise

1. Find leaf node where item should be inserted by repeatedly comparing item with values in node and following appropriate link to child
2. If room in leaf node
   Add item
   Otherwise … (ctd)

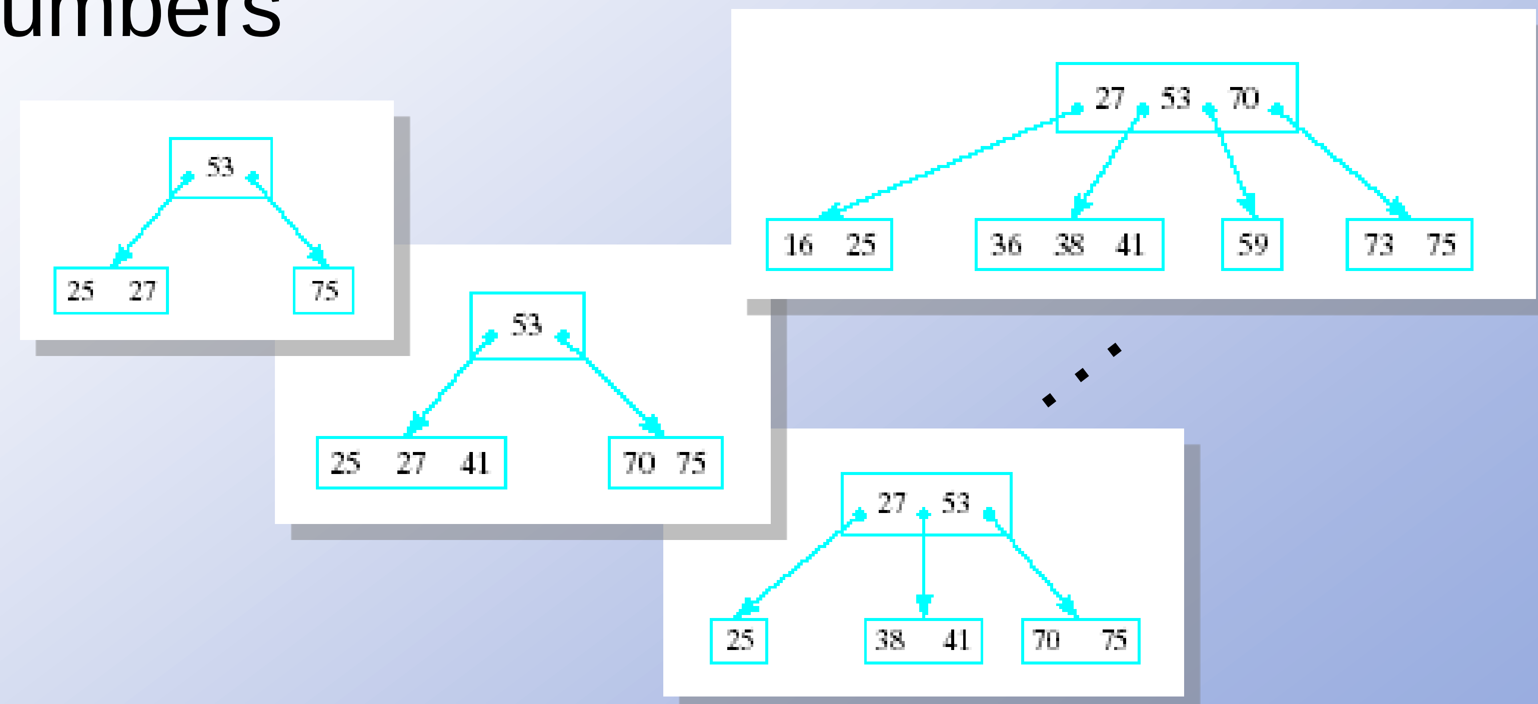
Building a Balanced 2-3-4 Tree

Otherwise:

a. Split 4-node into 2 nodes, one storing items less than median value, other storing items greater than median. Median value moved to a parent having these 2 nodes as children
b. If parent has no room for median, spilt that 4-node in same manner, continuing until either a parent is found with room or reach full root
c. If full root 4-node split into two nodes, create new parent 2-node, the new root

Building a Balanced 2-3-4 Tree

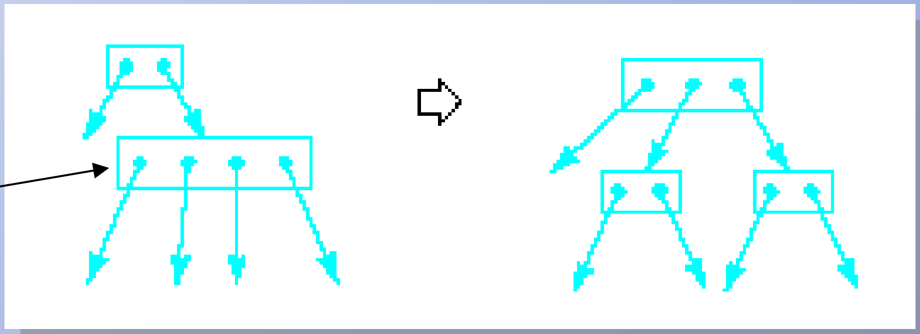
- Note sequence of building a tree by adding numbers



Building a Balanced 2-3-4 Tree

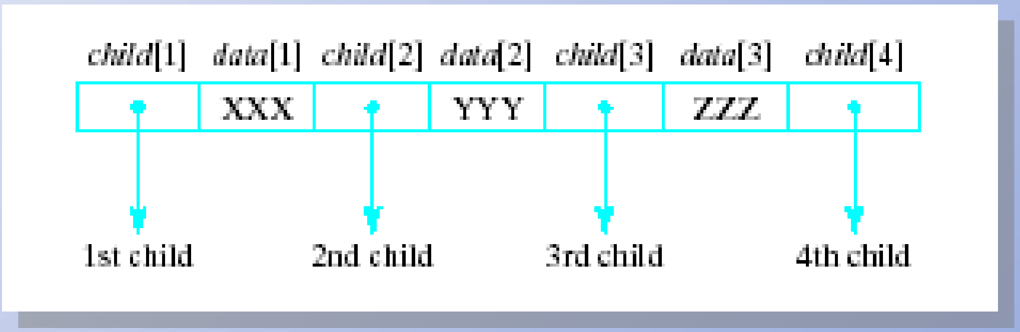
- Note that last step in algorithm may require multiple splits up the tree
- Instead of bo
  - Can do top-d
  - Do not allow any parent nodes to become 4-nodes
  - Split 4-odes along search path as encountered

Note that 2-3-4 trees stay balanced during insertions and deletions



Implementing 2-3-4 Trees

```
class Node234
{
public
  DataType data[3]; // type of data item in
  nodes
  Node234 * child[4];
  // Node234 operations
};
typedef Node234 * Pointer234
```



Red-Black Trees

- Defined as a BST which:
  – Has two kinds of links (red and black)
  – Every path from root to a leaf node has same number of black links
  – No path from root to leaf node has more than two consecutive red links
- Data member added to store color of link from parent

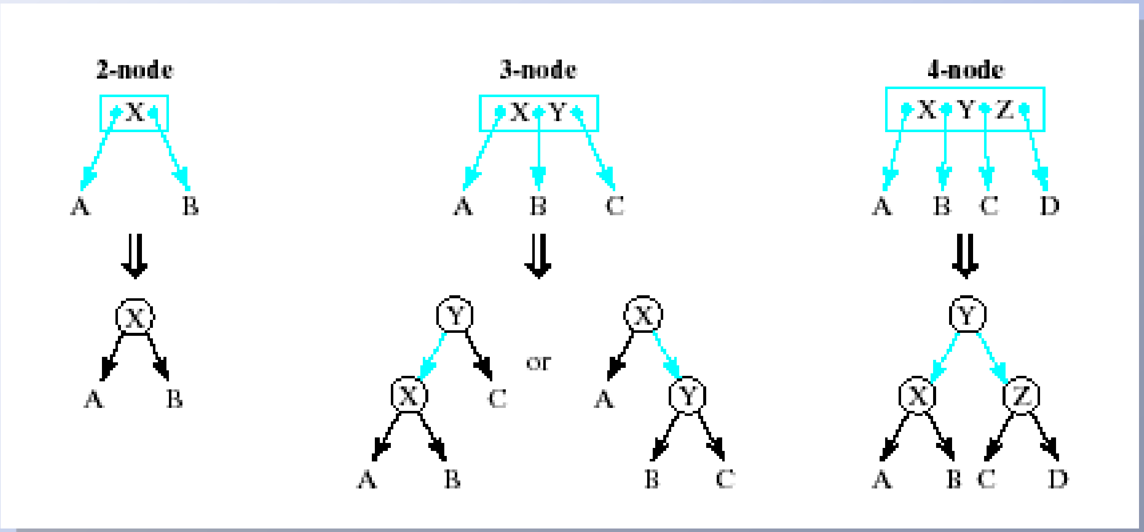
Red-Black Trees

```
enum ColorType {RED, BLACK};
class RedBlackTreeNode
{
public:
  DataType data;
  ColorType parentColor;  // RED or BLACK
RedBlackTreeNode * parent;
RedBlackTreeNode * left;
RedBlackTreeNode * right;
}
```

- Now possible to construct a red-black tree to represent a 2-3-4 tree

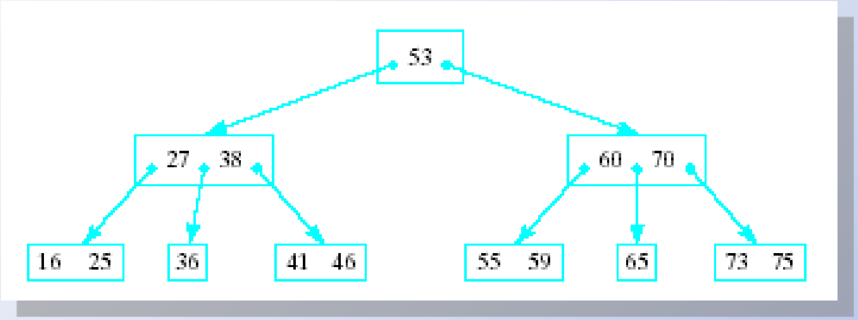
Red-Black Trees

- 2-3-4 tree represented by red-black trees as follows:
  - Make a link black if it is a link in the 2-3-4 tree
  - Make a link red if it connects nodes containing values in same node of the 2-3-4 tree

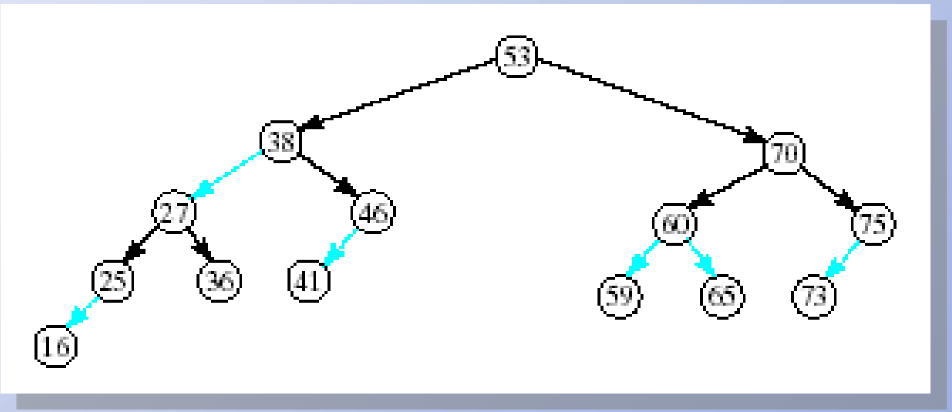


Red-Black Trees

- Example: given 2-3-4 tree



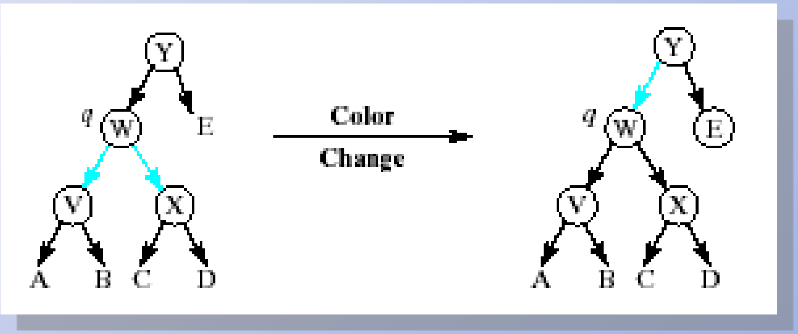
Represented by this red-black tree



Constructing Red-Black Tree

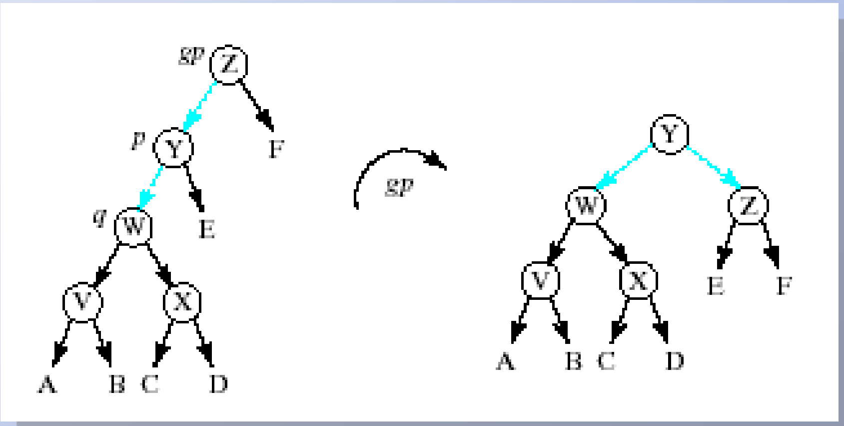
- Do top-down insertion as with 2-3-4 tree

1. Search for place to insert new node – Keep track of parent, grandparent, great grandparent
2. When 4-node **q** encountered, split as follows:
   a. Change both links of **q** to black
   b. Change link from parent to red:



Constructing Red-Black Tree

3. If now two consecutive red links, (from grandparent `gp` to parent `p` to `q`)
   Perform appropriate AVL type rotation determined by direction (left-left, right-right, left-right, right-left) from `gp -> p-> q`

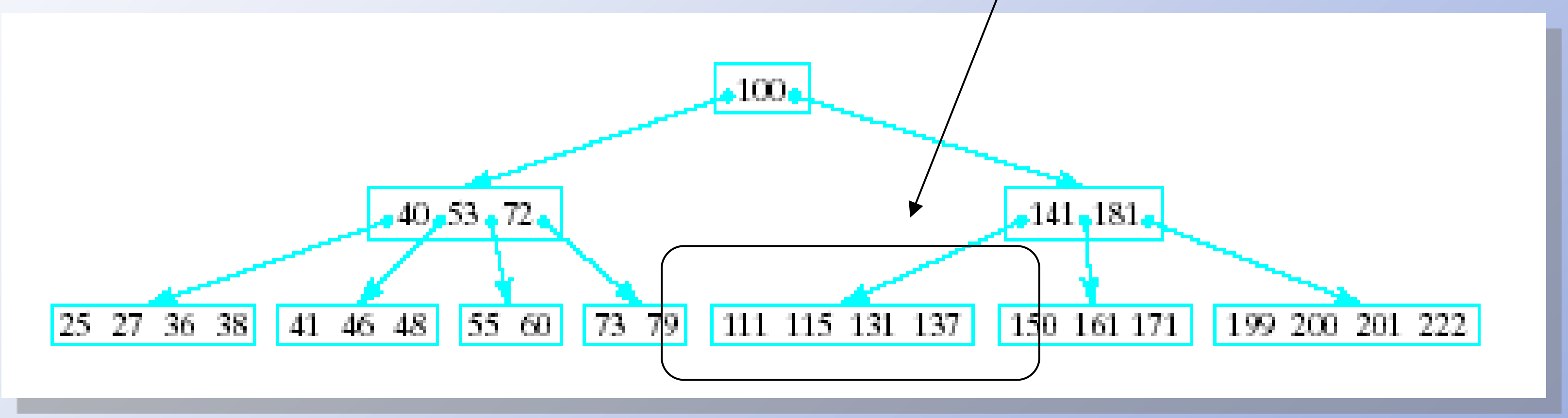


B-Trees

- Previous trees used in *internal* searching schemes
  - Tree sufficiently small to be all in memory
- B-tree useful in *external* searching
  - Data stored in 2ndry memory
- B-tree has properties
  - Each node stores at most m – 1 data values
  - Each internal nodes is 2-node, 3-node, …or m-node
  - All leaves on same level

B-Trees

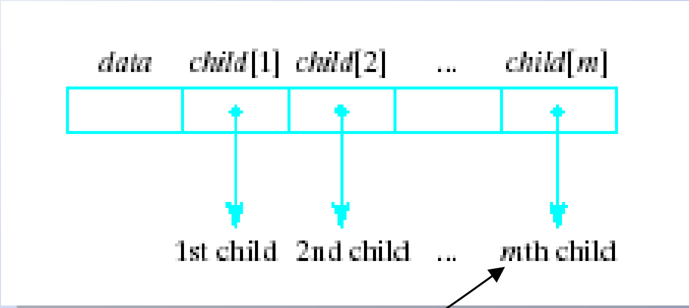
- Thus a 2-3-4 tree is a B-tree of order 4
- Note example below of order 5 B-tree



- Best performance found to be with values for $50 \leq m \leq 400$

Representing Trees & Forests as Binary Trees

- Consider a node with multiple links



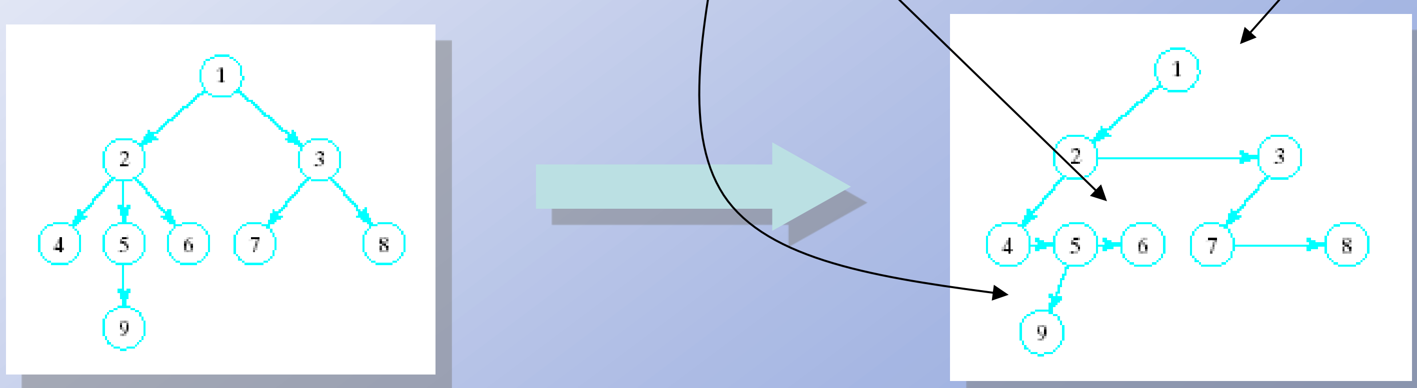

```
const int MAX_CHILDREN = … ;
class TreeNode
{
public:
DataType data
TreeNode * child[MAX_CHILDREN];
// TreeNode operations
}
typedef TreeNode * TreePointer;
```

Note problem with wasted space when not all links used

Representing Trees & Forests as Binary Trees

- Use binary (two link) tree to represent multiple links
  - Use one of the links to connect siblings in order from left to right
  - The other link points to firs … t of its children

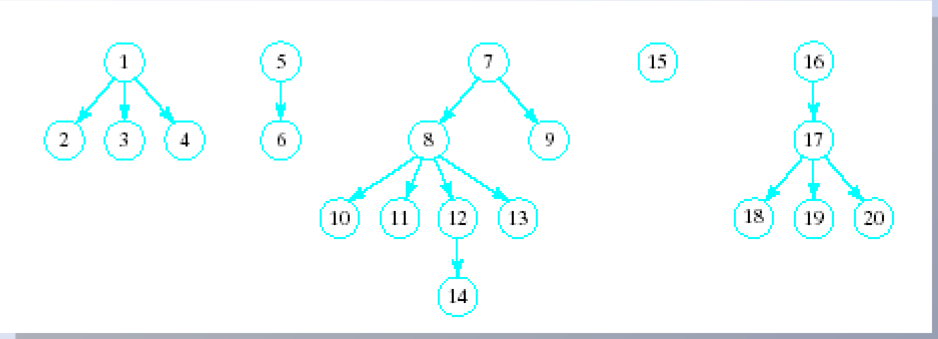
Note right pointer in root always null

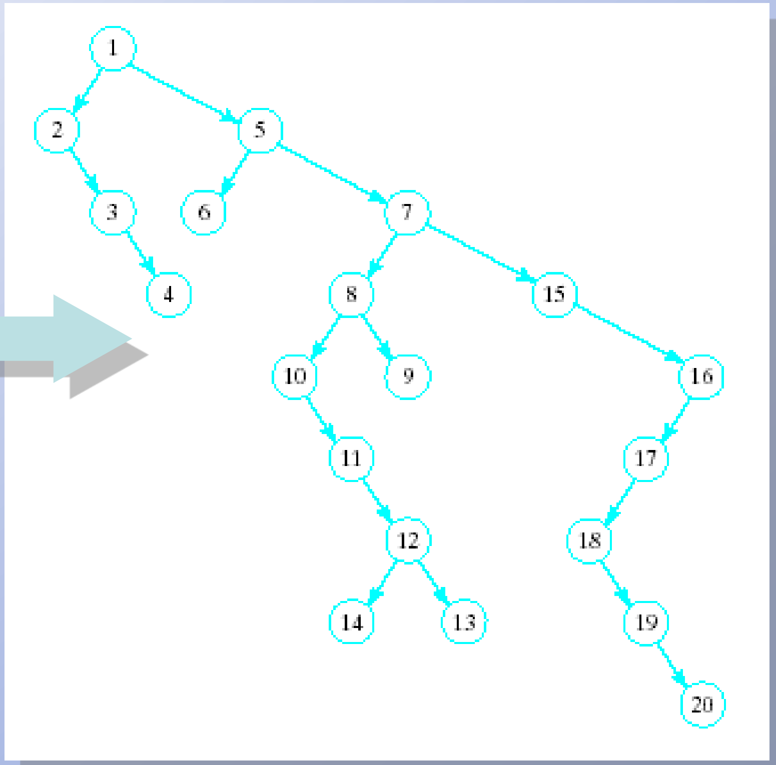


Representing Trees & Forests as Binary Trees

- Now possible to represent collection of trees (a forest!)
- This collection

becomes





Associative Containers in STL `map`s (optional)

- Contrast to sequential containers
  - `vector, deque, list, stack, queue`
- STL also provides associative containers
  - `set, multiset, map, multimap`
- Note operations provided
  - Table page 896, text
- Associative containers implemented with red-black trees
- Note that `map` containers provide subscript operator `[ ]`

map Containers

- Subscript operator allows associative arrays
  - The index type may be <u>any type</u>
- Declaration

```
<KeyType, DataType, les<KeyType> > obj;
```

- See example program, <u>Fig 15.1</u>