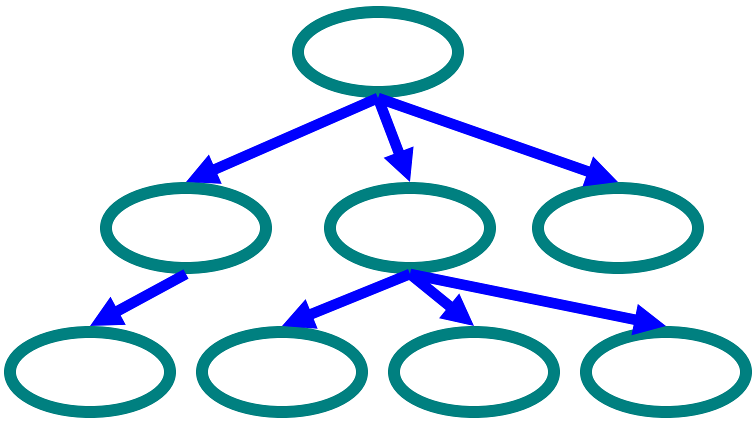
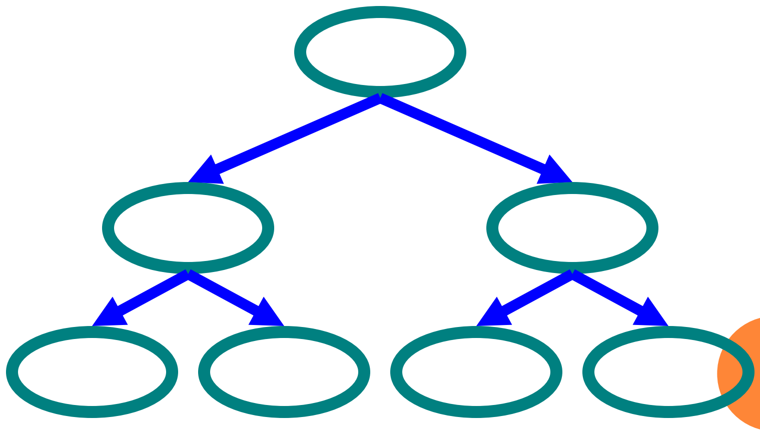
Tree Data Structures

Trees Data Structures

- Tree
  - Nodes
  - Each node can have 0 or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0–2 children per node



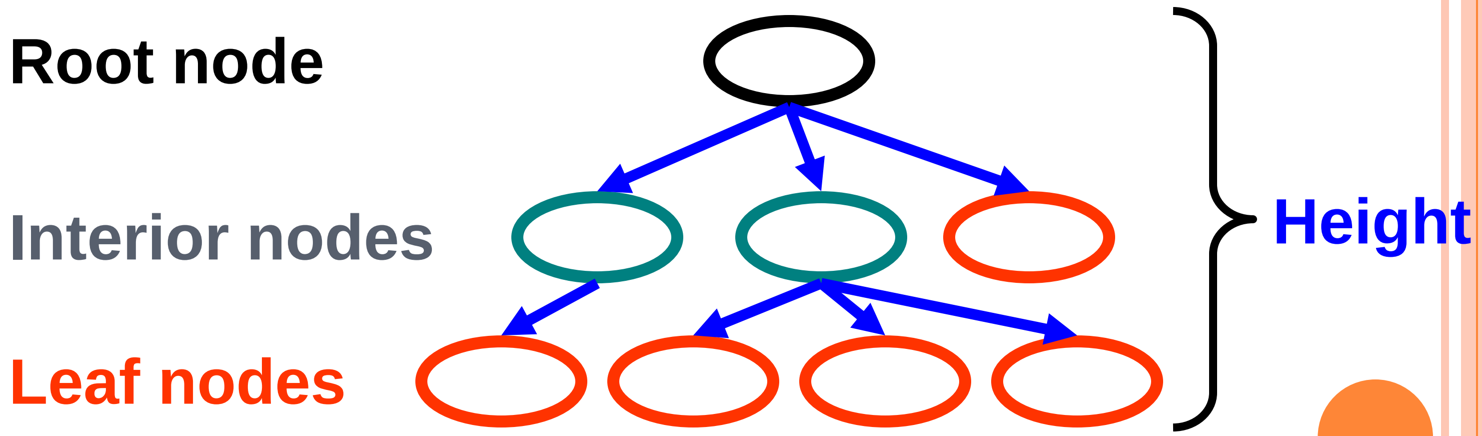
Tree



Binary Tree

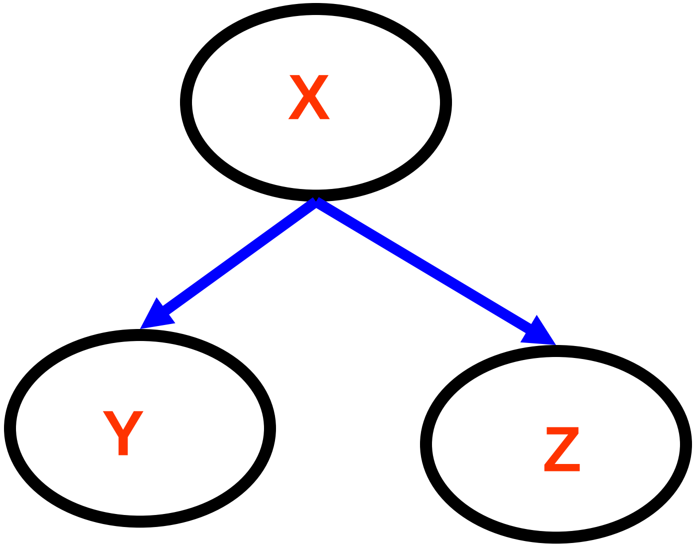
Trees

- Terminology
  - Root ⇒ no parent
  - Leaf ⇒ no child
  - Interior ⇒ non-leaf
  - Height ⇒ distance from root to leaf



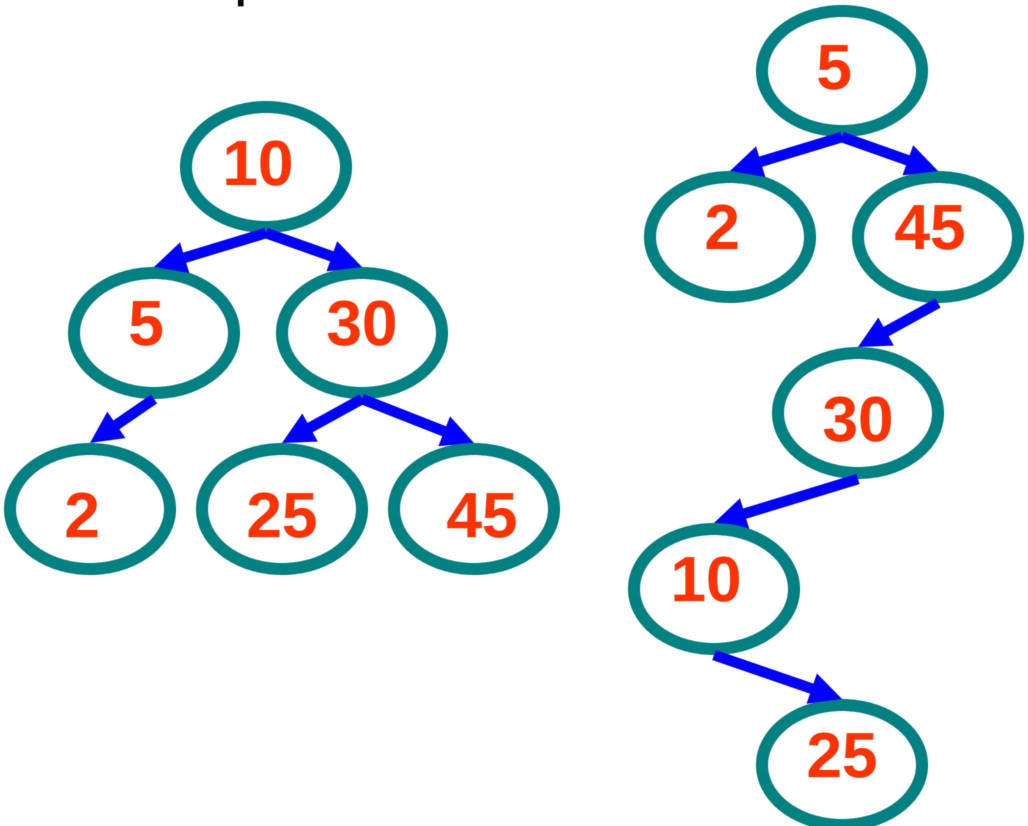
Binary Search Trees

- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z

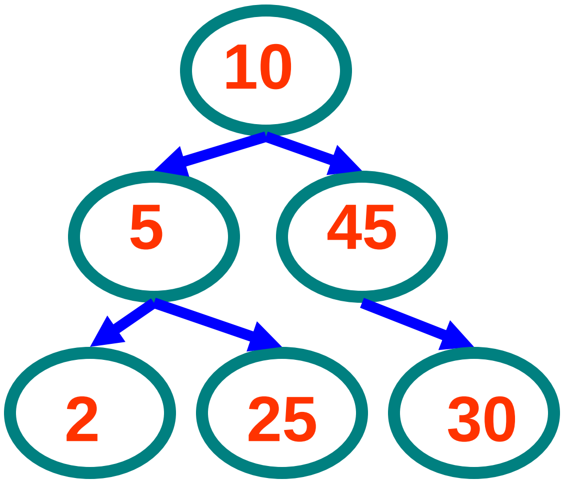


Binary Search Trees

- Examples



Binary search trees



Not a binary search tree

Binary Tree Implementation

```
Class Node {
    int data; // Could be int, a class, etc
    Node *left, *right;        // null if empty

    void insert ( int data ) { … }
    void delete ( int data ) { … }
    Node *find ( int data ) { … }
        …
}
```

Iterative Search of Binary Tree

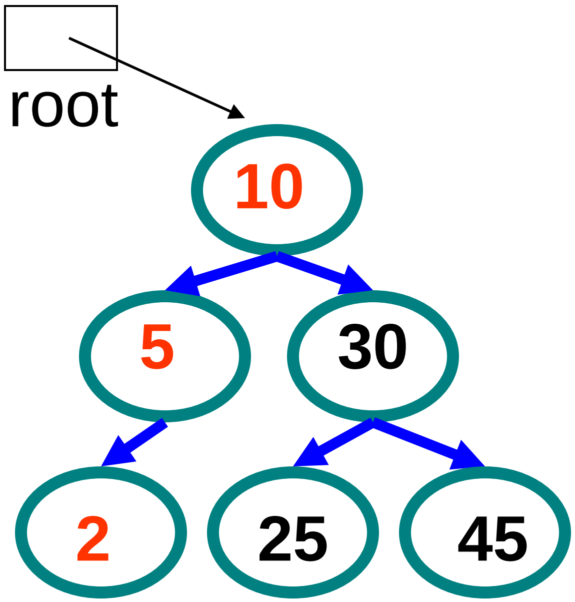
```
Node *Find( Node *n, int key) {
     while (n != NULL) {
     if (n->data == key)        // Found it
           return n;
        if (n->data > key)                   // In left subtree
           n = n->left;
        else                                 // In right subtree
           n = n->right;
     }
     return null;
}
Node * n = Find( root, 5);
```

Recursive Search of Binary Tree

```
Node *Find( Node *n, int key) {
    if (n == NULL)              // Not found
        return( n );
    else if (n->data == key)        // Found it
        return( n );
    else if (n->data > key)  // In left subtree
        return Find( n->left, key );
    else                            // In right subtree
        return Find( n->right, key );
}
Node * n = Find( root, 5);
```

Example Binary Searches

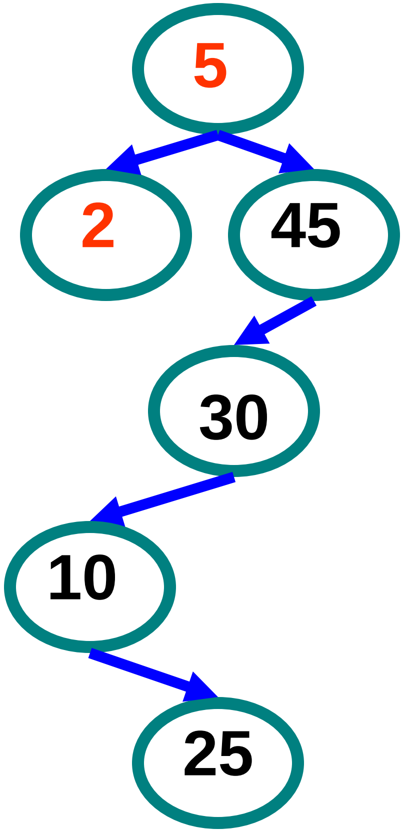
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

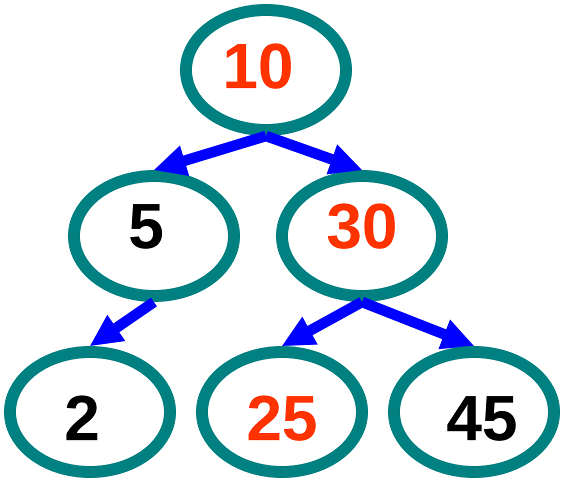


5 > 2, left

2 = 2, found

Example Binary Searches

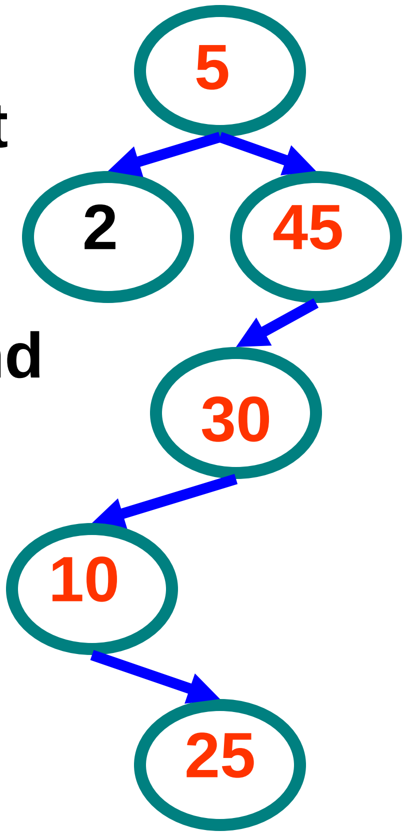
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

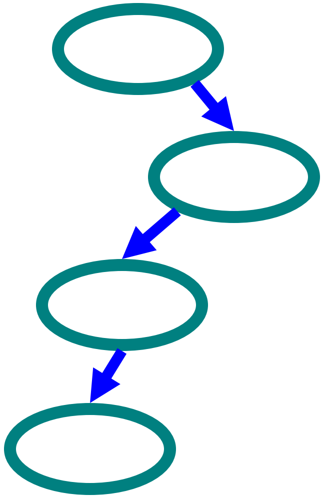
30 > 25, left

10 < 25, right

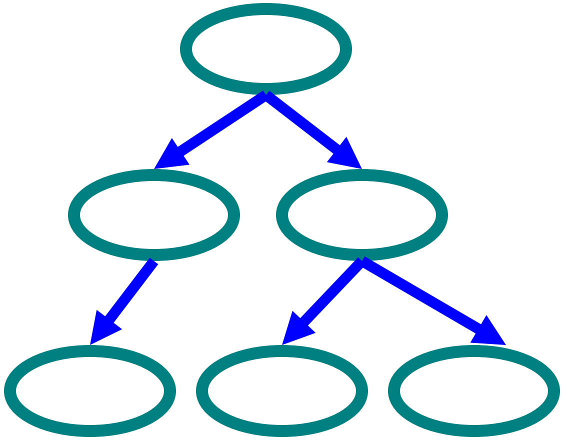
25 = 25, found

Types of Binary Trees

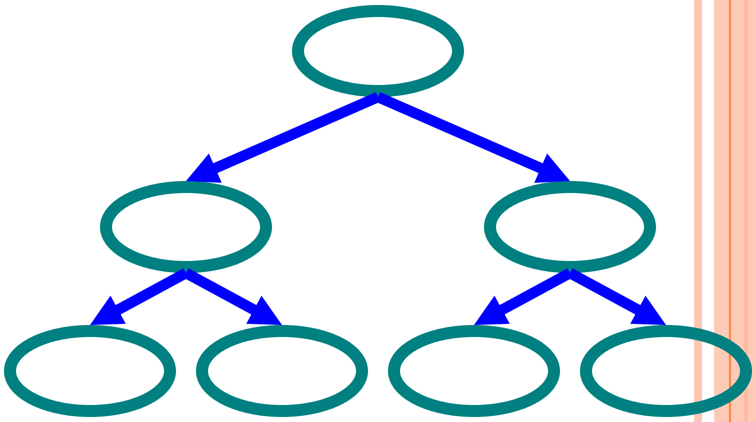
- Degenerate – only one child
- Complete – always two children
- Balanced – “mostly” two children
  - more formal definitions exist, above are intuitive ideas



Degenerate binary tree



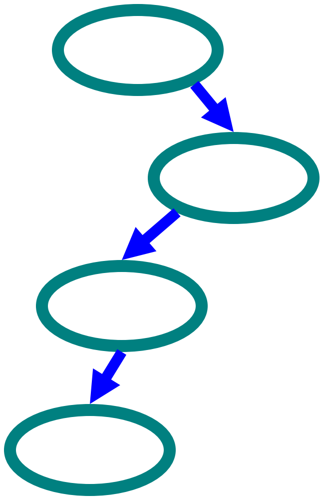
Balanced binary tree



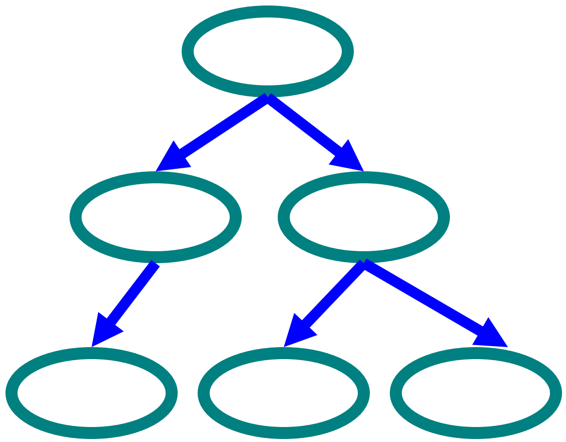
Complete binary tree

Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list
- Balanced
  - Height = O(log(n)) for n nodes
  - Useful for searches



Degenerate binary tree



Balanced binary tree

Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerate tree
    - O(n) time
    - Like searching linked list / unsorted array

Binary Search Tree Construction

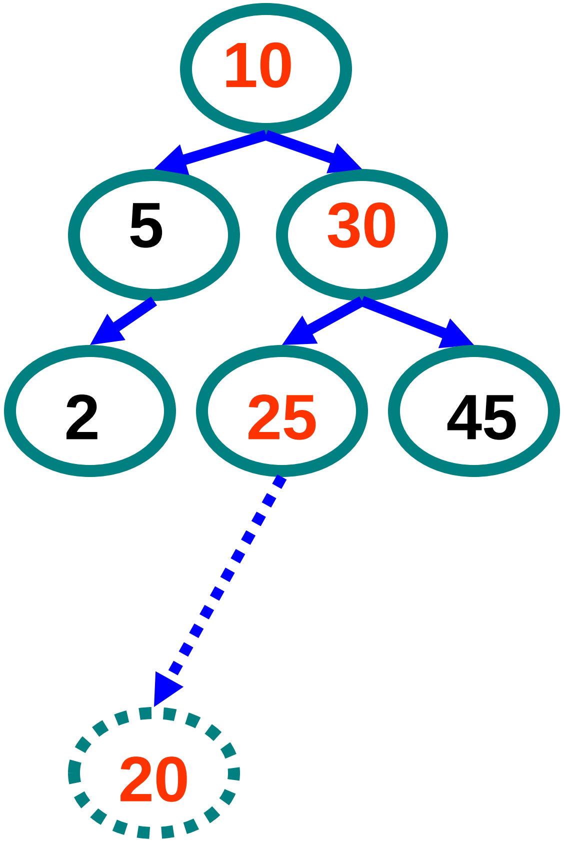
- How to build & maintain binary trees?
  - Insertion
  - Deletion
- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree

Binary Search Tree – Insertion

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y
- Observations
  - O(log(n)) operation for balanced tree
  - Insertions may unbalance tree

Example Insertion

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

Binary Search Tree – Deletion

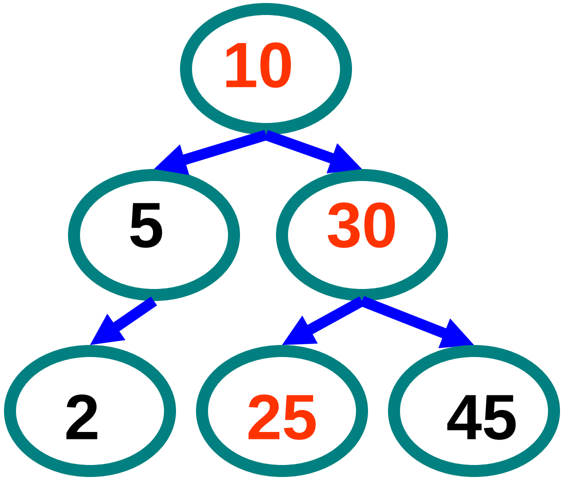
- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

     a) Replace with largest value Y on left subtree
     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

EXAMPLE DELETION (LEAF)

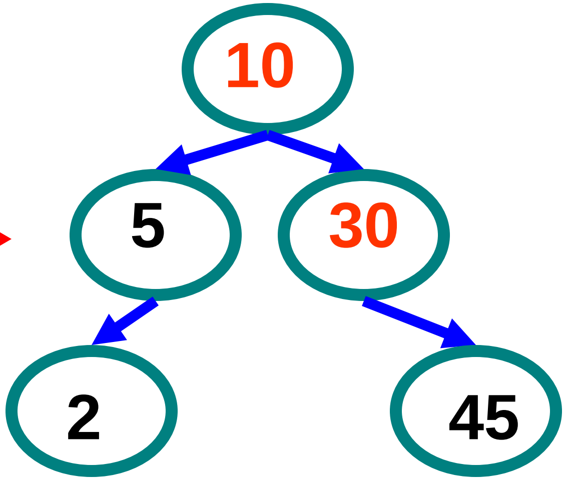
- Delete (25)



10 < 25, right

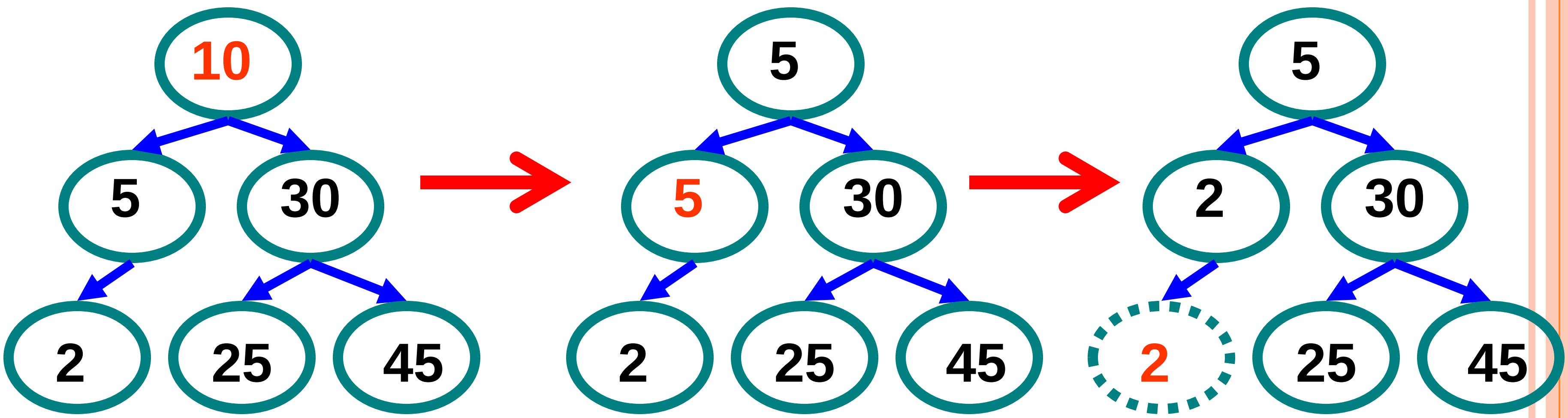
30 > 25, left

25 = 25, delete



EXAMPLE DELETION (INTERNAL NODE)

- Delete (10)



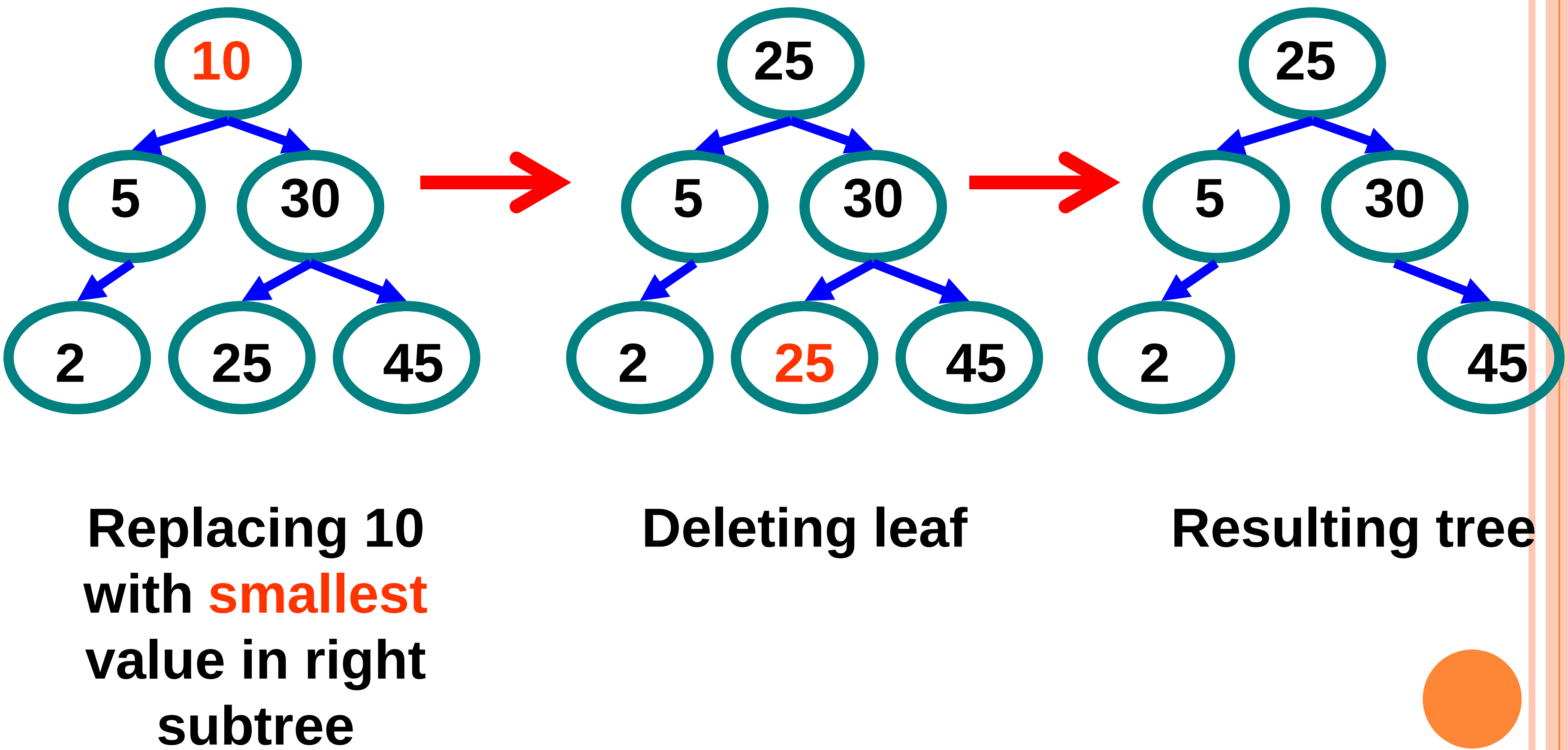
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

EXAMPLE DELETION (INTERNAL NODE)

- Delete (10)



Balanced Search Trees

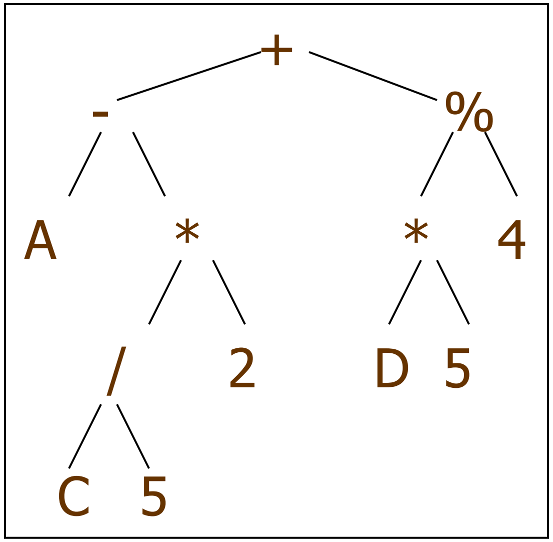
- Kinds of balanced binary search trees
  - height balanced vs. weight balanced
  - “Tree rotations” used to maintain balance on insert/delete
- Non-binary search trees
  - 2/3 trees
    - each internal node has 2 or 3 children
    - all leaves at same depth (height balanced)
  - B-trees
    - Generalization of 2/3 trees
    - Each internal node has between k/2 and k children
      - Each node has an array of pointers to children
    - Widely used in databases

Other (Non-Search) Trees

- Parse trees
  - Convert from textual representation to tree representation
  - Textual program to tree
    - Used extensively in compilers
  - Tree representation of data
    - E.g. HTML data can be represented as a tree
      - called DOM (Document Object Model) tree
    - XML
      - Like HTML, but used to represent data
      - Tree structured

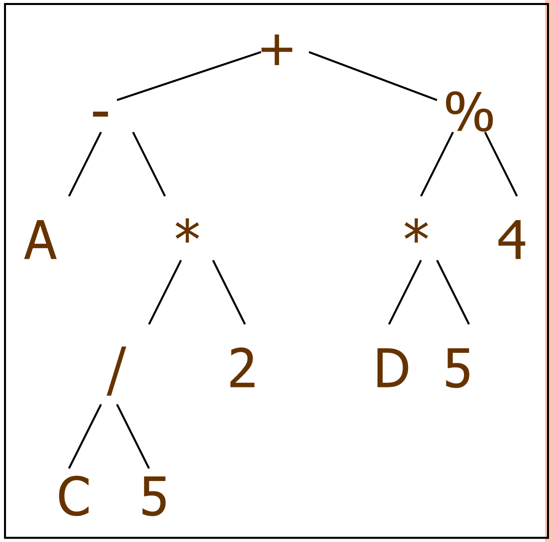
Parse Trees

- Expressions, programs, etc can be represented by tree structures
  - **`E.g. Arithmetic Expression Tree`**
  - **`A-(C/5 * 2) + (D*5 % 4)`**



Tree Traversal

- Goal: visit every node of a tree
- **in-order** traversal



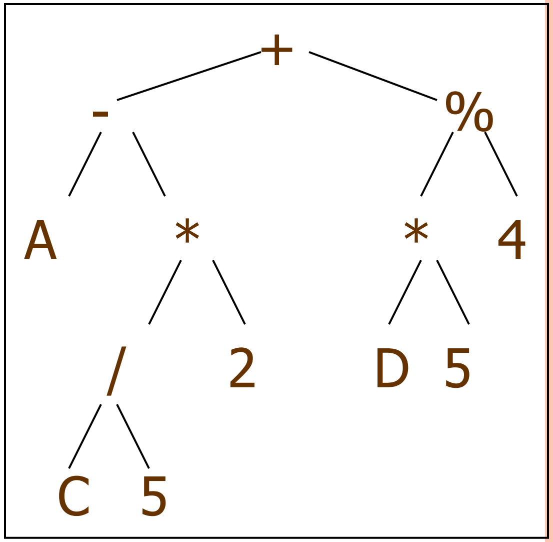

```
void Node::inOrder () {
    if (left != NULL)  {
       cout << "(";  left->inOrder(); cout << ")";
    }
    cout << data << endl;
    if (right != NULL) right->inOrder()
}
```

Output: A – C / 5 * 2 + D * 5 % 4

To disambiguate: print brackets

TREE TRAVERSAL (CONTD.)

- **pre-order** and **post-order**:




```
void Node::preOrder () {
    cout << data << endl;
    if (left != NULL)   left->preOrder ();
    if (right != NULL) right->preOrder ();
}
```

Output: + - A * / C 5 2 % * D 5 4

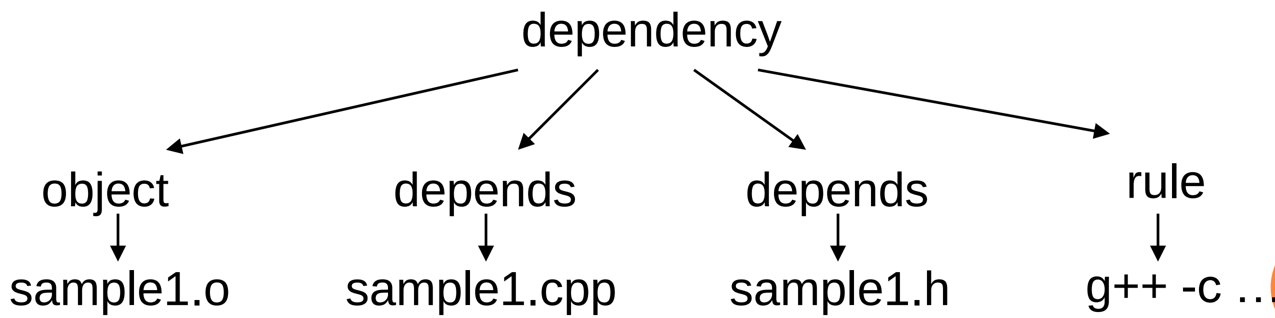
```
void Node::postOrder () {
    if (left != NULL)   left->preOrder ();
    if (right != NULL) right->preOrder ();
    cout << data << endl;
}
```

Output: A C 5 / 2 * - D 5 * 4 % +

XML

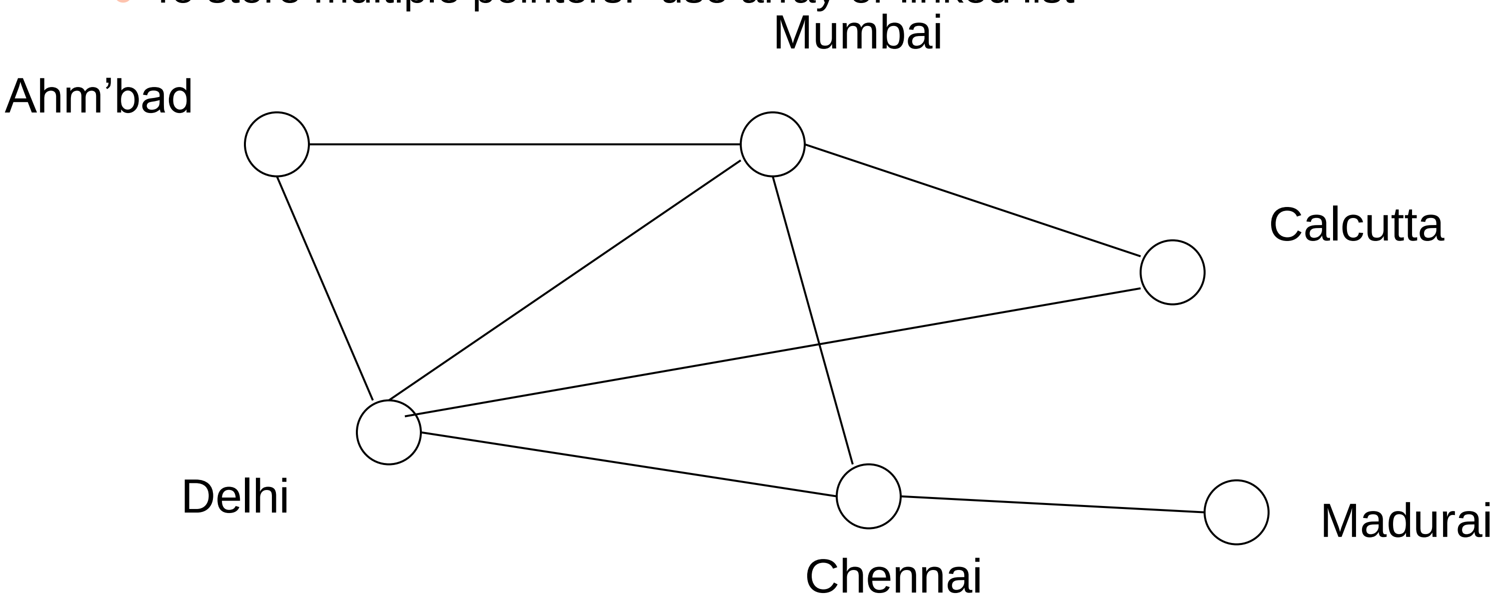
- Data Representation
  - E.g.

    <dependency>
    <object>sample1.o</object>
    <depends>sample1.cpp</depends>
    <depends>sample1.h</depends>
    <rule>g++ -c sample1.cpp</rule>
    </dependency>
  - Tree representation



Graph Data Structures

- E.g: Airline networks, road networks, electrical circuits
- Nodes and Edges
- E.g. representation: class Node
  - Stores name
  - stores pointers to all adjacent nodes
    - i,e. edge == pointer
    - To store multiple pointers: use array or linked list



End of Chapter