Trees

Fall 2020

School of Software Engineering

South China University of Technology

Contents

- Definitions of tree
- Binary tree
- AVL tree
- Splay tree
- B-tree

Why Do We Need Trees?

- Lists, Stacks, and Queues are linear relationships
- Information often contains hierarchical relationships
  - File directories or folders
  - Moves in a game
  - Hierarchies in organizations
- Can build a tree to support fast searching

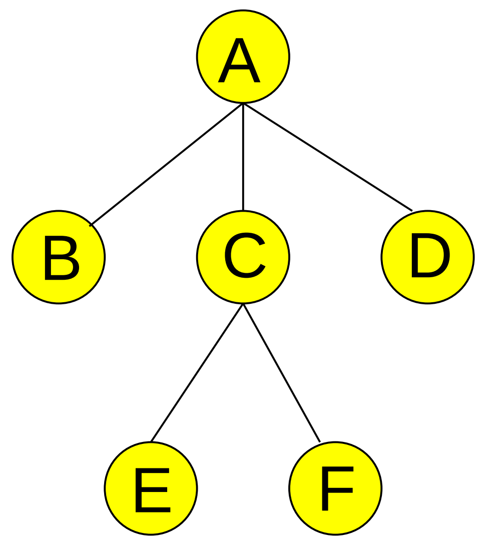
Tree Jargon

- root
- nodes and edges
- leaves

- parent, children, siblings
- ancestors, descendants

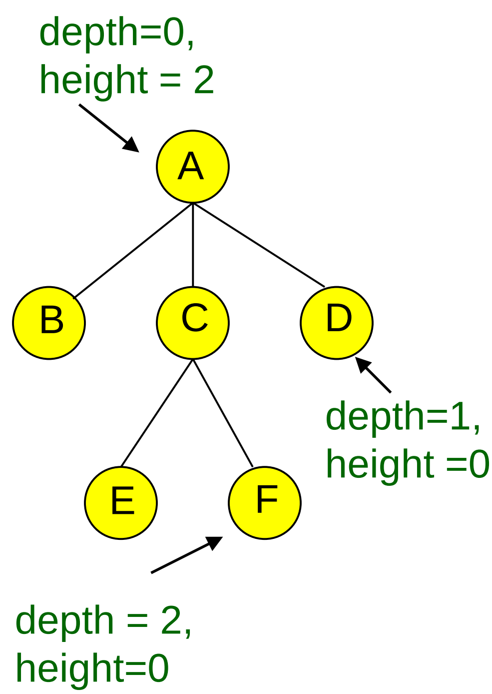
- subtrees

- path, path length
- height, depth



More Tree Jargon

- **Length** of a path = number of edges
- **Depth** of a node N = length of path from root to N
- **Height** of node N = length of longest path from N to a leaf
- **Depth of tree** = depth of deepest node
- **Height of tree** = height of root



Definition and Tree Trivia

- A tree is a set of nodes,i.e., either
  - it's an empty set of nodes, or
  - it has one node called the root from which zero or more trees (subtrees) descend

- Two nodes in a tree have at most one path between them

- Can a non-zero path from node N reach node N again?

  No. Trees can never have cycles (loops)

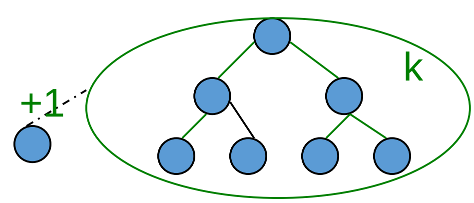
Paths

- A tree with N nodes always has N-1 edges (prove it by induction)

Base Case: N=1   one node, zero edges

Inductive Hypothesis: Suppose that a tree with N=k nodes always has k-1 edges.

Induction: Suppose N=k+1…
The k+1st node must connect to the rest by 1 or more edges.
If more, we get a cycle. So it connects by just 1 more edge

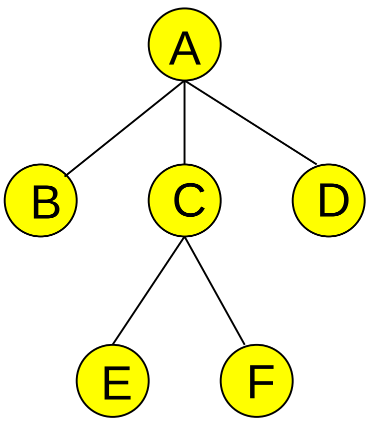


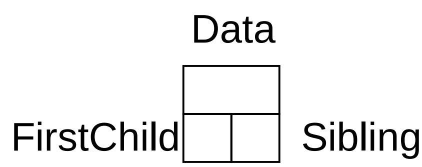
Implementation of Trees

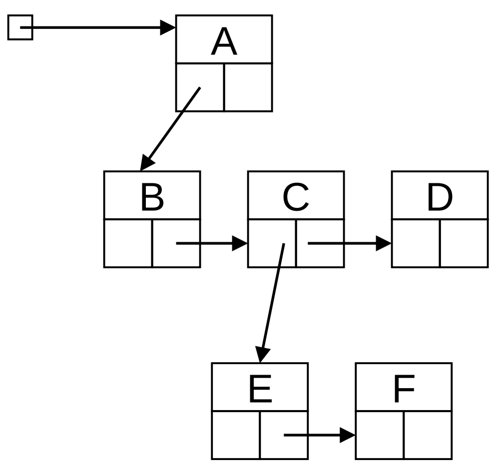
- One possible pointer-based Implementation
  - tree nodes with value and a pointer to each child
  - but how many pointers should we allocate space for?

- A more flexible pointer-based implementation
  - 1^{st} Child / Next Sibling List Representation
  - Each node has 2 pointers: one to its first child and one to next sibling
  - Can handle arbitrary number of children

Arbitrary Branching







Nodes of same depth

Implementation of Trees

```
//Node declarations for trees
Struct TreeNode{
    Object element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
};
```

Binary Trees

Binary Trees

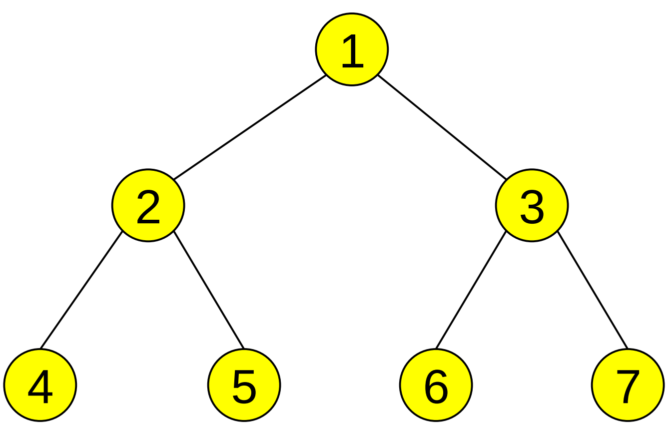
- Every node has at most two children
  - Most popular tree in computer science
- Given N nodes, what is the minimum depth of a binary tree?
  - This means all levels but the last are full!
  - At depth d, you can have $N = 2^d$ to $N = 2^{d+1}-1$ nodes

$$\boxed{2^d \leq N \leq 2^{d+1} - 1 \ \text{ implies } \ d_{min} = \lfloor \log_2 N \rfloor}$$

Minimum depth vs node count

- At depth d, you can have $N = 2^d$ to 2^{d+1}-1 nodes
- minimum depth d is $\Theta(\log N)$

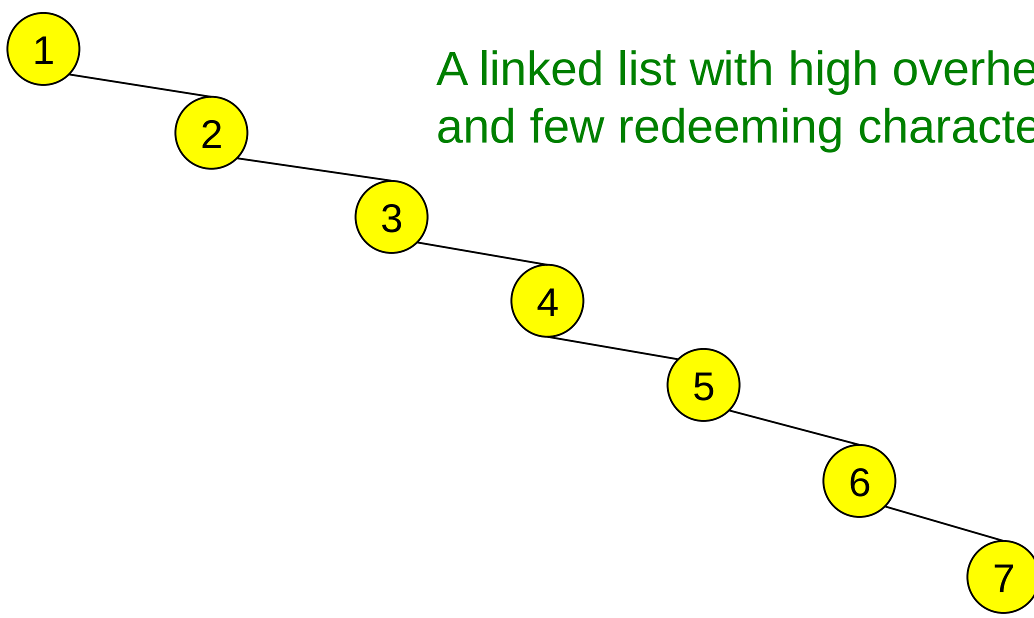
T(n) = Θ(f(n)) means
T(n) = O(f(n)) and f(n) = O(T(n)),
i.e. T(n) and f(n) have the same growth rate



d=2
N=2^2 to 2^3-1
(i.e, 4 to 7 nodes)

Maximum depth vs node count

- What is the maximum depth of a binary tree?
  - Degenerate case: Tree is a linked list!
  - Maximum depth = N-1

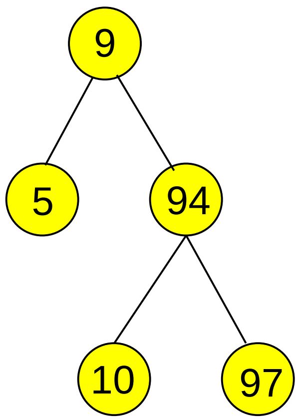


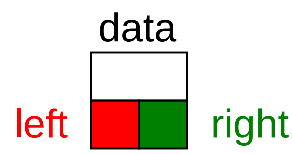
A linked list with high overhead and few redeeming characteristics

Maximum depth vs node count

- Goal: Would like to keep depth at around logN to get better performance than linked list for operations like Find

Binary Tree






```
template <typename E> class BinNode {
 public:
  virtual E& element() = 0; //return the node's value
  virtual void setElement(const E&) = 0; //set the node's value

  virtual BinNode* left() const = 0;  //return the node's left child
  virtual void setLeft(BinNode*) = 0; //set the left child

  virtual BinNode* right() const = 0; //return the right child
  virtual void setRight(BinNode*) = 0; //set the right child

  virtual bool isLeaf() = 0; //check if a node is a leaf or not
};
```

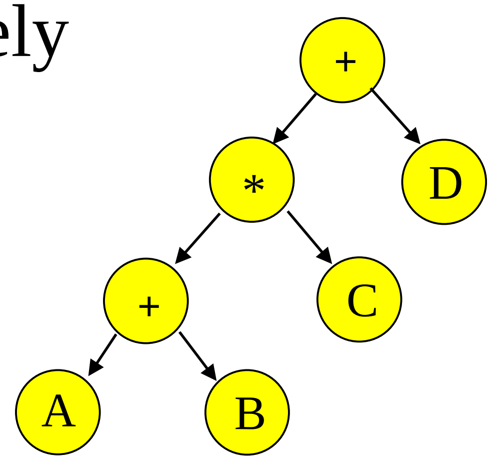
Traversing Binary Trees

- The definitions of the traversals are recursive definitions. For example:
  - Visit the root
  - Visit the left subtree (i.e., visit the tree whose root is the left child) and do this recursively
  - Visit the right subtree (i.e., visit the tree whose root is the right child) and do this recursively

- Traversal definitions can be extended to general (non-binary) trees

Traversing Binary Trees

- Preorder: Node, then Children (starting with the left) recursively
  - + * + A B C D



- Inorder: Left child recursively, Node, Right child recursively
  - A + B * C + D

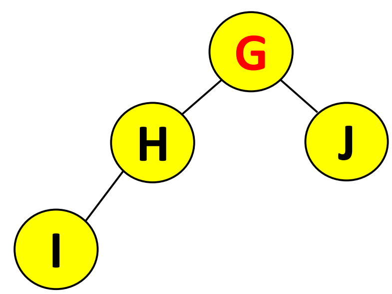
- Postorder: Children recursively, then Node
  - A B + C * D +

Traversing Binary Trees

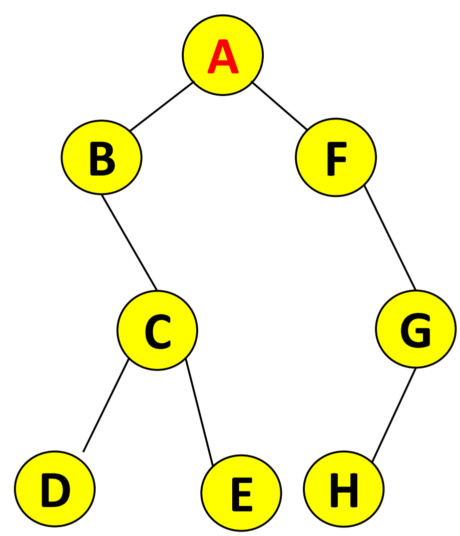
```
//Implementing preorder traversal as a recursive
// function
template<typename E>
void preorder(BinNode<E>* root){
  if (root == NULL) return; //Empty subtree
  visit(root);  //Perform desired action
  preorder(root->left());
  preorder(root->right());
}
```

Traversing Binary Trees

- Given two traversal enumerations, can you draw a unique binary tree?
  - Ex1:
    - preorder: **G** H I J;
    - inorder: I H **G** J

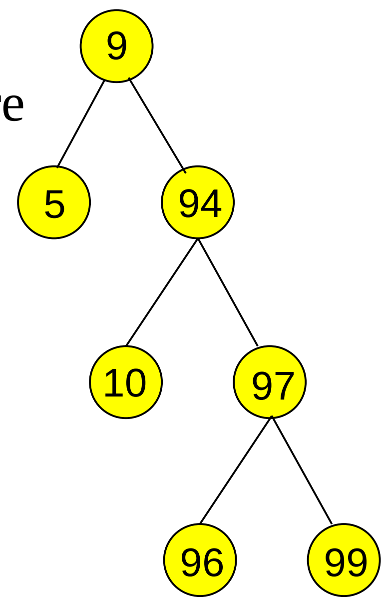


  - Ex2:
    - Postorder: D E C B H G F **A**
    - Inorder: B D C E **A** F H G



Binary Search Trees

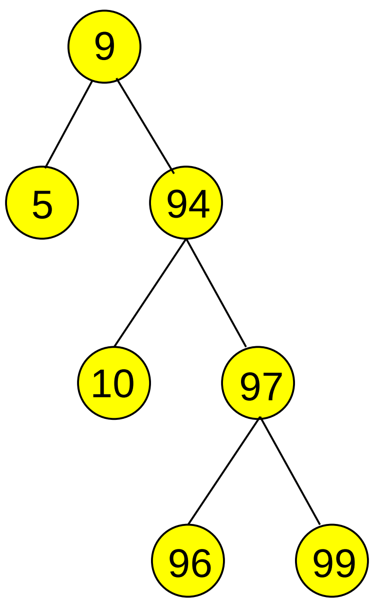
- Binary search trees are binary trees in which
  - all values in the node's left subtree are less than node value
  - all values in the node's right subtree are greater than value (not less than) node

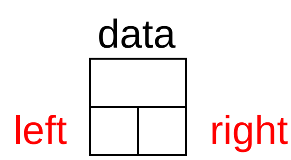


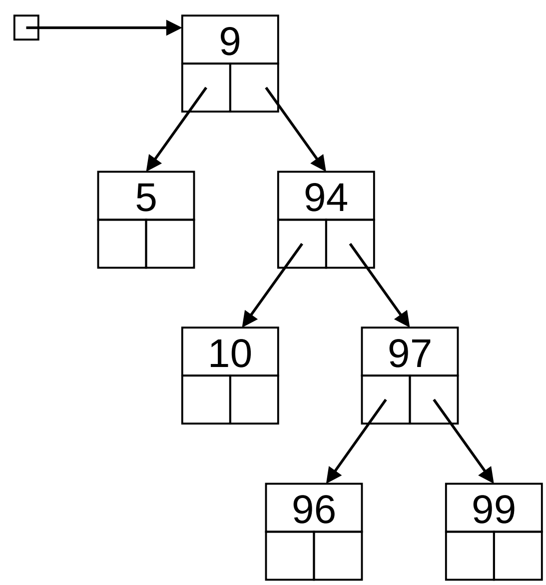
- Operations:
  - Find, FindMin, FindMax, Insert, Delete

What happens when we traverse a BST in inorder?

Binary Search Tree







Find

```
/**
* test if an item is in a subtree.
* x is item to search for.
* t is the node that roots the subtree.
*/
bool find( const Comparable & x, BinaryNode *t ) const{
 if( t == nullptr ) return false;
 else if( x < t->element ) return contains( x, t->left );
 else if( t->element < x ) return contains( x, t->right );
 else return true; // Match
}
```

FindMin

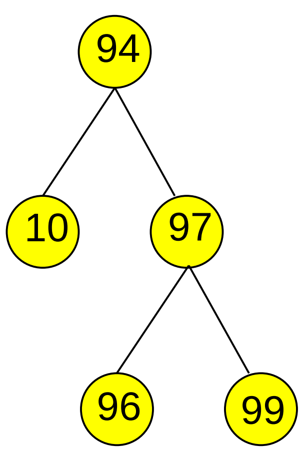
- Design recursive FindMin operation that returns the smallest element in a binary search tree.

```
/**
* find the smallest item in a subtree t.
* Return node containing the smallest item.
**/
BinaryNode * findMin( BinaryNode *t ) const{
  if( t == nullptr ) return nullptr; // root is empty
  if( t->left == nullptr ) return t; //left child is empty
  return findMin( t->left );
}
```

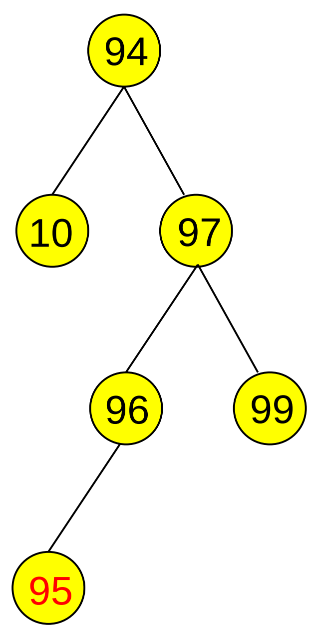
Insert

- Challenge: how to preserve the BST property without making major changes to the structure of the tree
  - Do a “Find” operation for X
  - If X is found → update (no need to insert)
  - Else, “Find” stops at a NULL pointer
  - Insert Node with X there

- Example: Insert 95

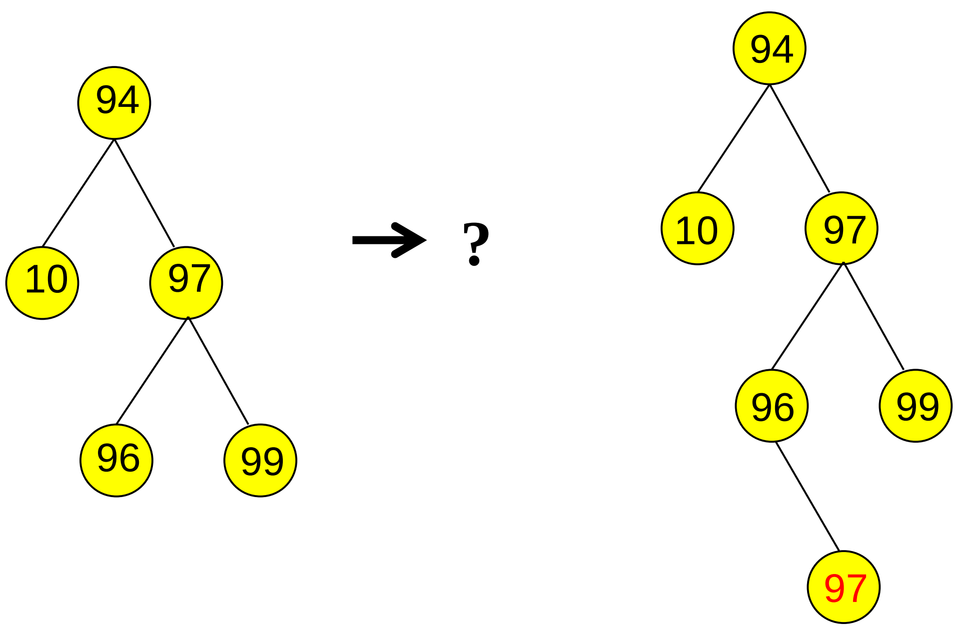


→ ?



Insert

- How to handle duplicates?
- Example: Insert 97



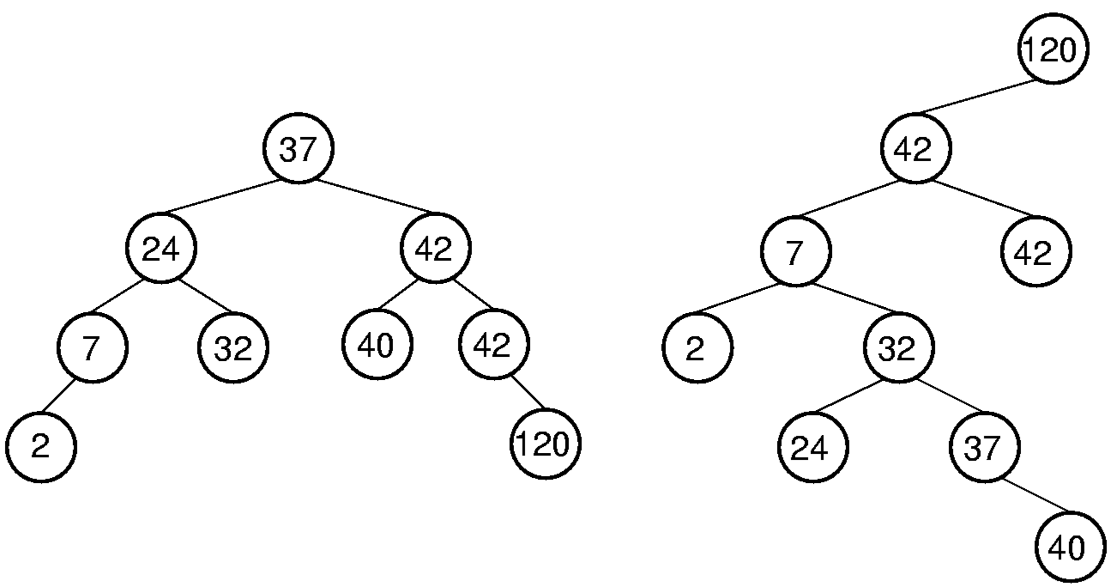
- Construct a new node and insert it into the tree.
  or,
- keeping an extra field in the node record indicating the frequency of occurrence, etc.

Insert

```
/**
  * Internal method to insert into a subtree.
  * x is the item to insert.
  * t is the node that roots the subtree.
  * Set the new root of the subtree.
  */
void insert( const Comparable & x, BinaryNode * & t ){
  if( t == nullptr )
     t = new BinaryNode{ x, nullptr, nullptr };
  else if( x < t->element )
    insert( x, t->left );
  else if( t->element < x )
    insert( x, t->right );
  else
     ; // Duplicate; do nothing
 }
```

Insert

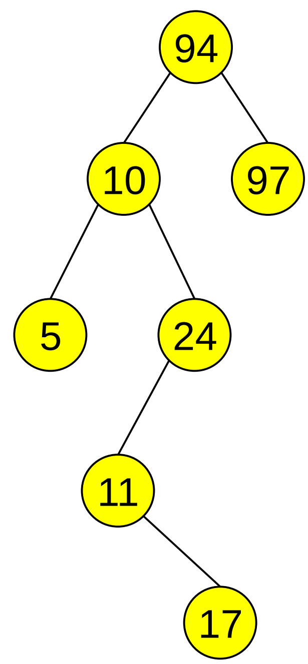
- The shape of a BST depends on the order in which elements are inserted.
  - Left BST: 37, 24, 42, 7, 2, 40, 42, 32, 120
  - Right BST: 120, 42, 42, 7, 2, 32, 37, 24, 40



What if nodes are inserted in a sorted order?

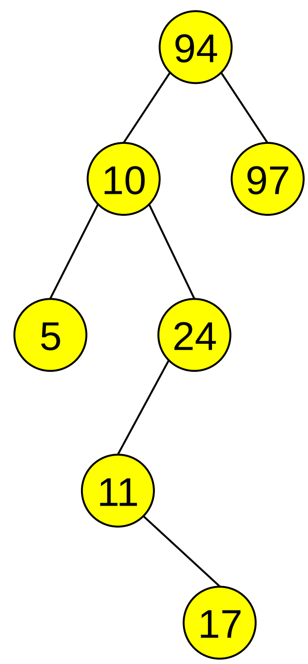
Delete

- Delete is a bit trickier…Why?
- Suppose you want to delete 10
- Strategy:
  - Find 10
  - Delete the node containing 10
- Problem: When you delete a node, what do you replace it by?



Delete

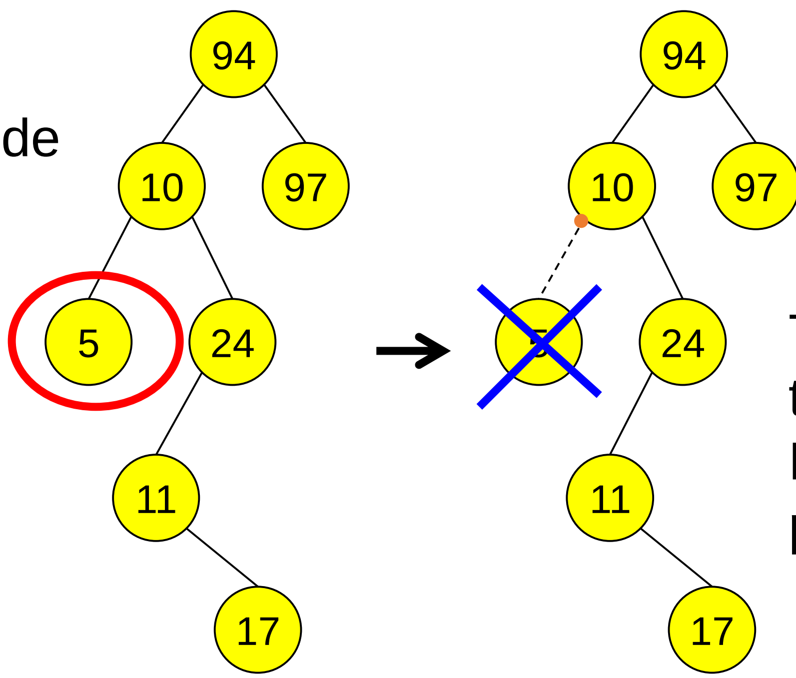
- Problem: When you delete a node, what do you replace it by?
- Solution:
  - If it has no children, by NULL
  - If it has 1 child, by that child
  - If it has 2 children, by the node with the smallest value in its right subtree (the successor of the node)



Delete

Delete node with no children

Find 5 node

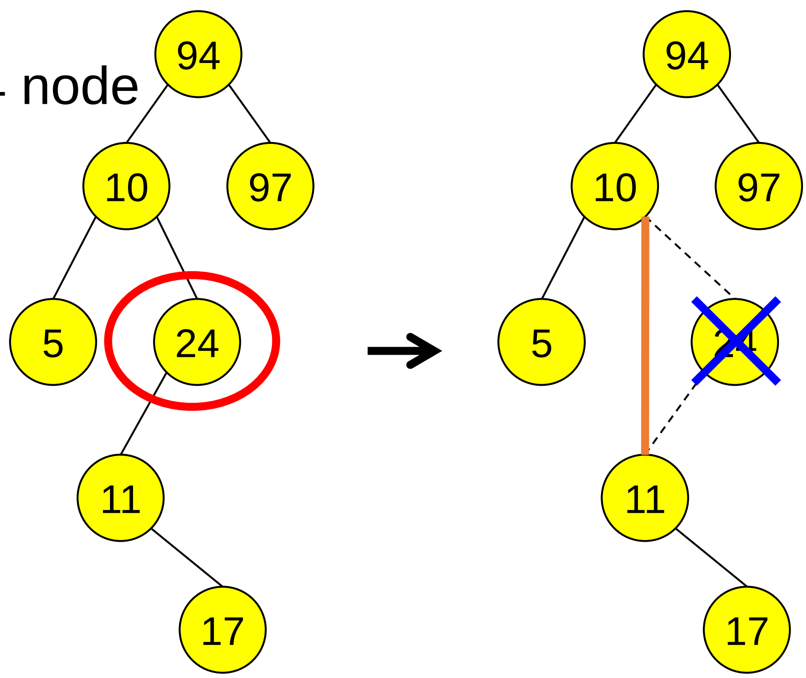


Then Free the 5 node and NULL the pointer to it

Delete

Delete node with one child

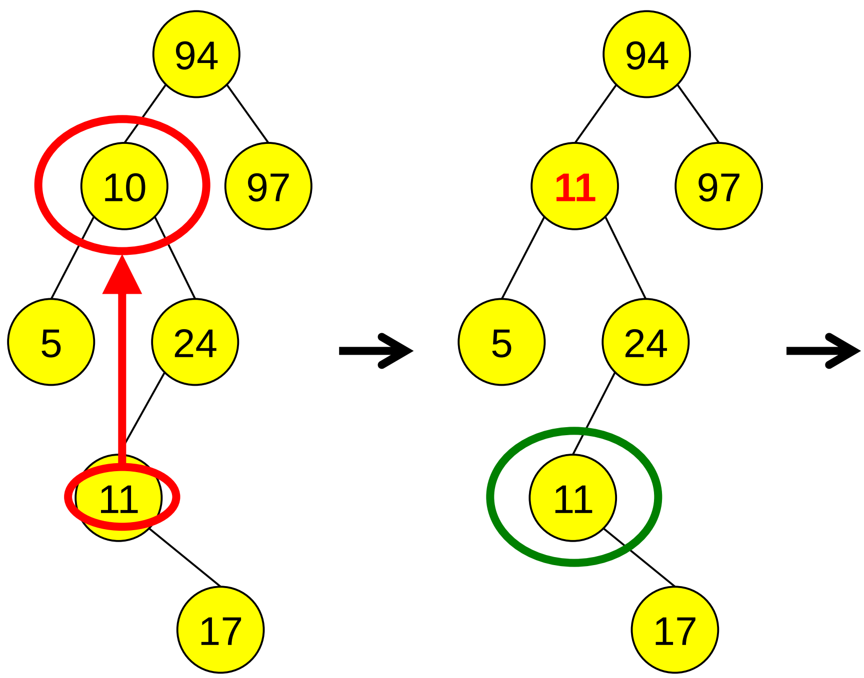
Find 24 node



Then Free the 24 node and replace the pointer to it with a pointer to its child

Delete

Delete node with two children

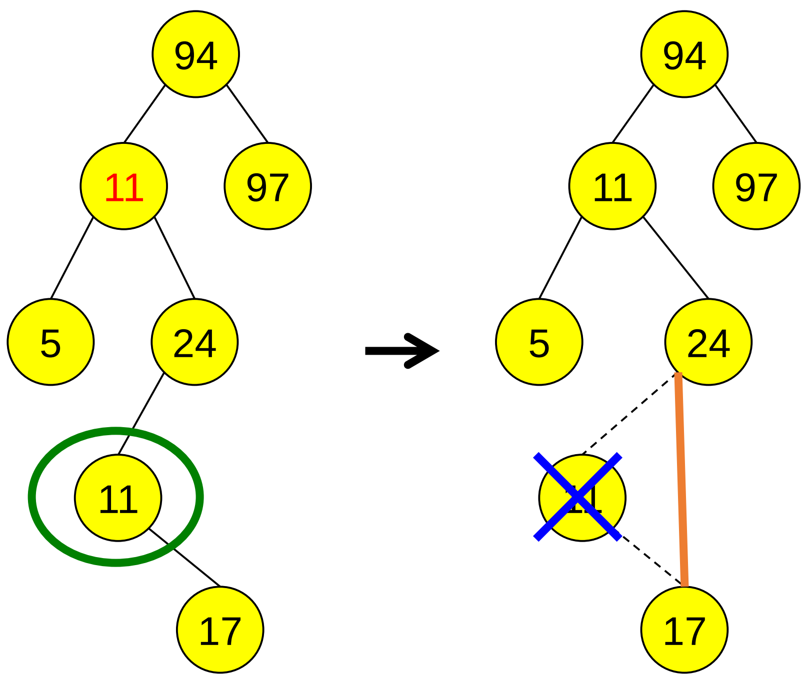


Find 10,
Copy the smallest value in right subtree into the node

Then (recursively) Delete node with smallest value in right subtree
Note: it cannot have two children (why?)

Delete

Delete node with two children (continued)



Remember
11 node

Then Free the 11 node and replace the pointer to it with a pointer to its child

Remove (delete)

```
/**
* remove from a subtree. Textbook page 140
* x is the item to remove.  t is the node that roots the subtree.
* Set the new root of the subtree.
*/
void remove( const Comparable & x, BinaryNode * & t ){
  if( t == nullptr ) return; // Item not found; do nothing
  if( x < t->element ) remove( x, t->left );
  else if( t->element < x ) remove( x, t->right );
  else if( t->left != nullptr && t->right != nullptr ){// Two children
    t->element = findMin( t->right )->element;
    remove( t->element, t->right );
  }
  else{
    BinaryNode *oldNode = t;
    t = ( t->left != nullptr ) ? t->left : t->right;
    delete oldNode;
  }
}
```

Analysis of BST

- Cost of finding, insertion, and removal (one node) is the depth of the deepest node in the tree
  - Desirable to keep BSTs **balanced,** that is, with least possible height
  - Balanced BST in average case: $\Theta(\log n)$
  - Unbalanced BST in worst case: $\Theta(n)$
- Traverse cost: $\Theta(n)$

It is preferable for a BST to be as shallow as possible

Homework

- Homework 3-1
  - Textbook Exercises 4.2, 4.5, 4.9