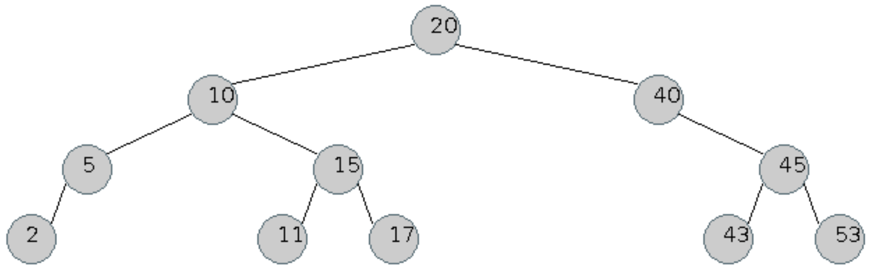
08: binary trees #1

Searching | Unbalanced Insertion | Traversal | Deletion | Threaded Trees | Tree Parameters

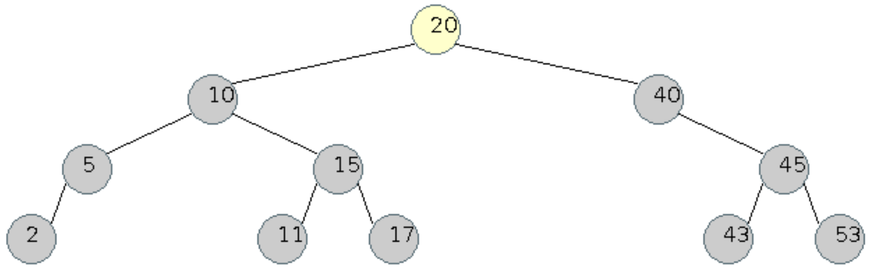
Binary Tree Search

Binary Tree Search has a time complexity that is proportional to the height of the tree. At each element, the left hand side element is smaller and the right hand side element is greater. If the height is h, the max search time is O(lg(h)).

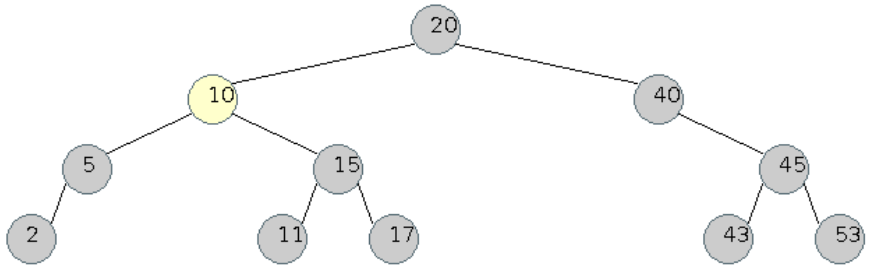
1. Start:



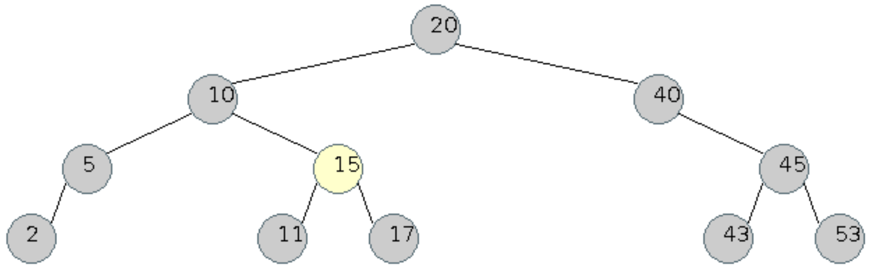
2. Searching for 17, Start at root.



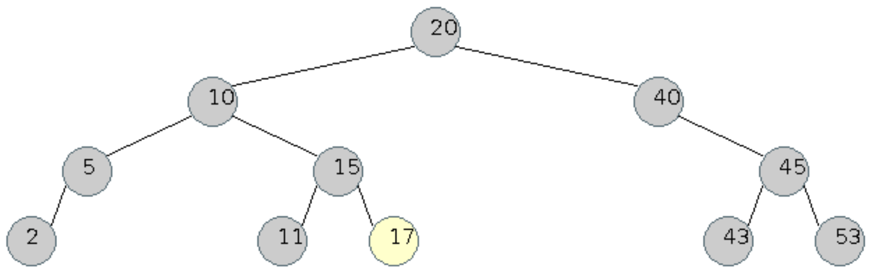
3. 17 is less than 20, go to left node.



4. 17 is greater than 10, go to right node.



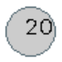
5. 17 is greater than 15, go to right node. found 17.



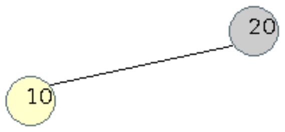
Binary Tree Insertion

To insert we search down to the lowest level and then insert when we get to NULL

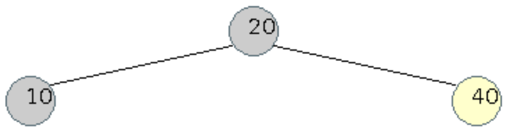
1. Start:



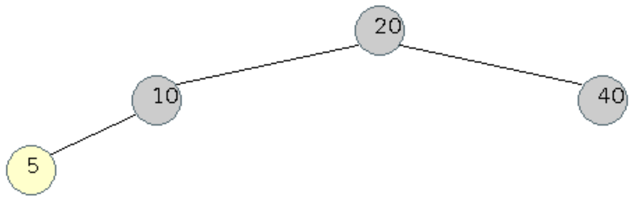
2. Insert 10



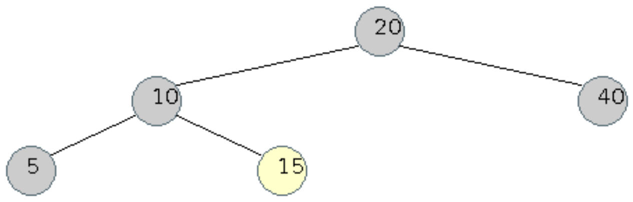
3. Insert 40



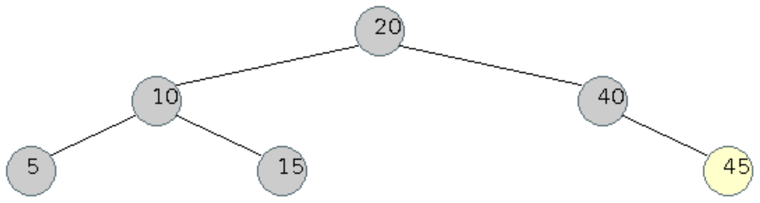
4. Insert 5



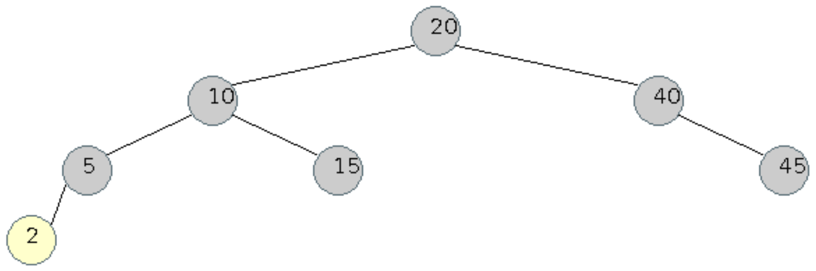
5. Insert 15



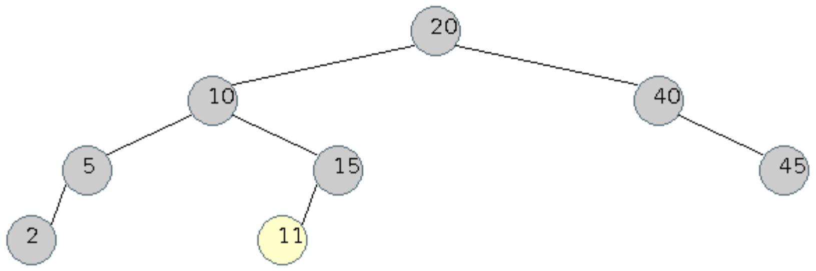
6. Insert 45



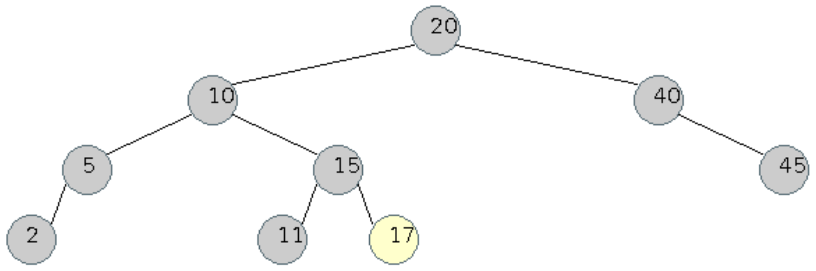
7. Insert 2



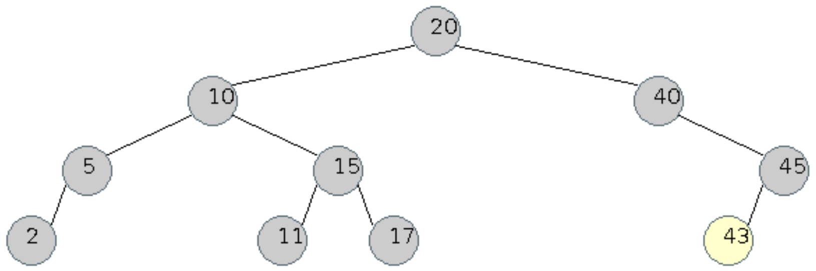
8. Insert 11



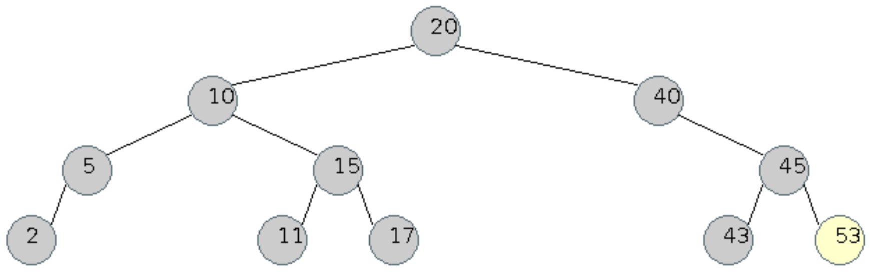
9. Insert 17



10. Insert 43



11. Insert 45



Binary Tree Traversal

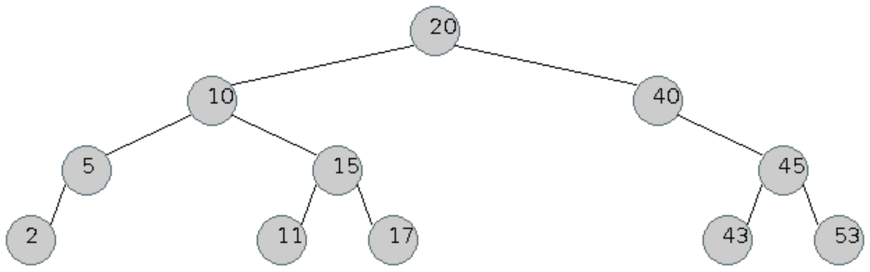
Traversal is the process of visiting each node exactly once in a specified order.

Breadth-First Traversal

A queue is used for breadth-first traversal. After a node is visited it's children are put on the end of the queue. No child of a level is visited until all nodes in the current level are visited because the children are behind the current level in the queue.

Visitation order:

- [20 10 40 5 15 45 2 11 17 43 53]



Depth-First Traversal

Tasks of interest in this type of traversal

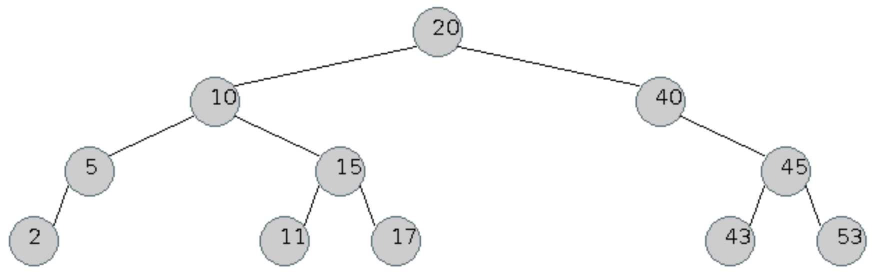
- V - visiting a node
- L - traversing the left subtree
- R - traversing the right subtree

Three standard traversals:

- VLR - preorder tree traversal
- LVR - inorder tree traversal
- LRV - postorder tree traversal

Visitation orders:

- Preorder: [20 10 5 2 15 11 17 40 45 43 53]
- Inorder: [2 5 10 11 15 17 20 40 43 45 53]
- Postorder: [2 5 11 17 15 10 43 53 45 40 20]

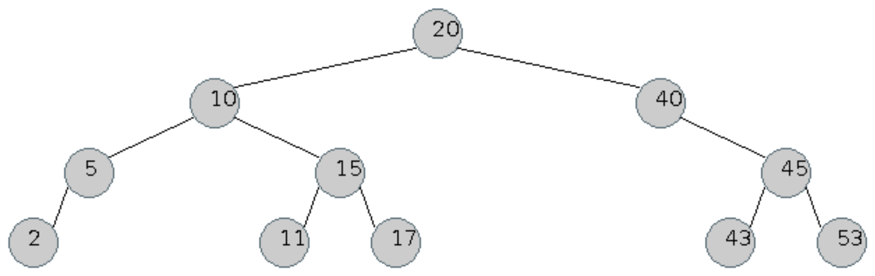


Binary Tree Deletion

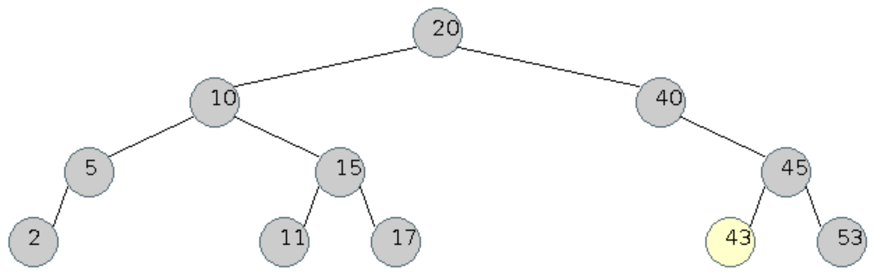
Binary Tree Deletion has several cases:

Deleting a Leaf

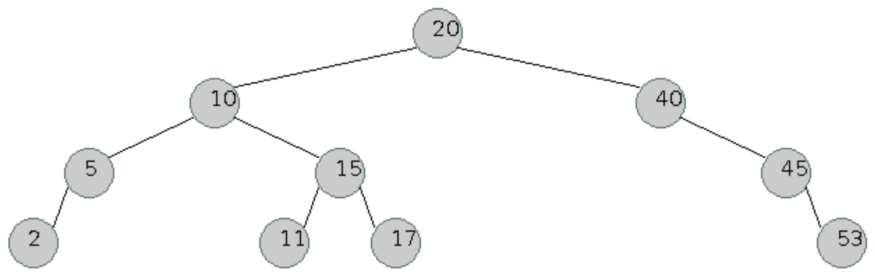
1. Start:



2. Delete 43. Seek to node and keep track of parent.

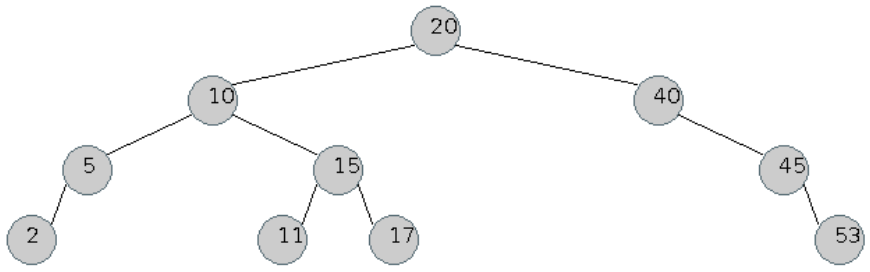


3. Delete curr and set parent->left = NULL

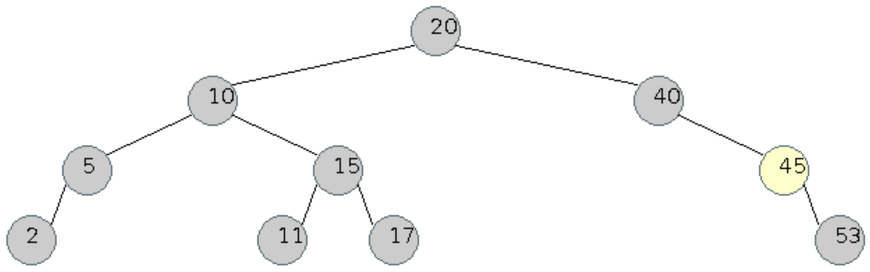


Deleting a Node with One Child

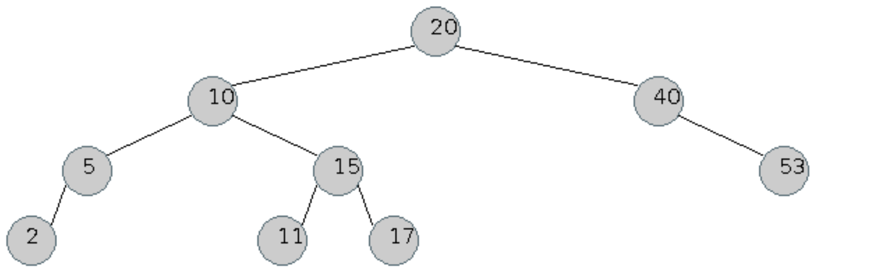
1. Start:



2. Delete 45. Seek to node and keep track of parent.

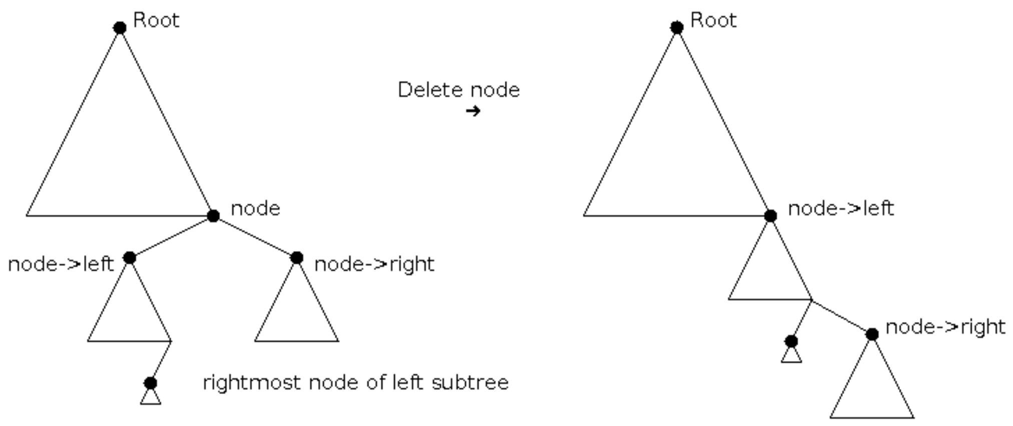


3. Delete curr and set parent->right = curr->right

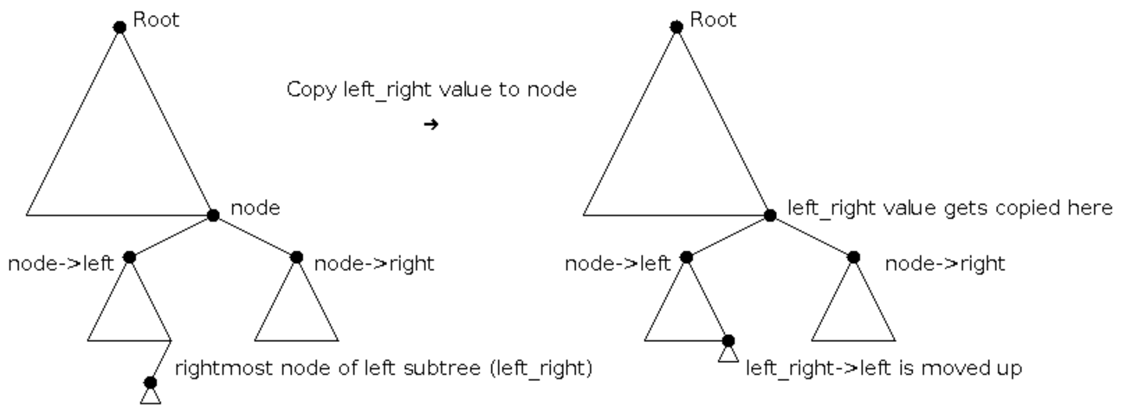


Deletion by Merging

When there are multiple nodes as children of a node to delete, more complicated solutions are needed.



Deletion by Copying


Root
node
node->left
node->right
rightmost node of left subtree (left_right)
Copy left_right value to node
Root
left_right value gets copied here
node->left
node->right
left_right->left is moved up

Threaded Trees

A threaded binary tree has normally null pointers point to the predecessor or successor of nodes.

- An inorder threaded binary tree has normally null right pointers point to the successor

Computation of Tree Parameters

```
// treeParameters.cpp - download here

#include <iostream>

class TreeNode {
public:
  TreeNode(int value);
  int getValue();
  void setLeft(TreeNode * left);
  void setRight(TreeNode * right);
  TreeNode * getLeft();
  TreeNode * getRight();
private:
  int m_Value;
  TreeNode * m_Left;
  TreeNode * m_Right;
};

TreeNode::TreeNode(int value)
  : m_Value(value), m_Left(NULL), m_Right(NULL)
{
}

int TreeNode::getValue(){
  return m_Value;
}

void TreeNode::setLeft(TreeNode * left){
  m_Left = left;
}

void TreeNode::setRight(TreeNode * right){
  m_Right = right;
}

TreeNode * TreeNode::getLeft(){
  return m_Left;
}

TreeNode * TreeNode::getRight(){
  return m_Right;
}

TreeNode * makeTree(){
  TreeNode * root = new TreeNode(20);
  TreeNode * left0 = new TreeNode(10);
  TreeNode * right0 = new TreeNode(40);
  root->setLeft(left0);
  root->setRight(right0);
  TreeNode * leftleft1 = new TreeNode(5);
  TreeNode * leftright1 = new TreeNode(15);
  left0->setLeft(leftleft1);
  left0->setRight(leftright1);
  leftleft1->setLeft(new TreeNode(2));
  leftright1->setLeft(new TreeNode(11));
  leftright1->setRight(new TreeNode(17));
  TreeNode * rightright1 = new TreeNode(45);
  right0->setRight(rightright1);
  rightright1->setLeft(new TreeNode(43));
  rightright1->setRight(new TreeNode(53));
  return root;
}

//Robert Sedgewick, Algorithms in C++, Parts 1-4, page 250.
int countNodes(TreeNode * curr){
  if(curr == NULL){
    return 0;
  }
  return countNodes(curr->getLeft()) + countNodes(curr->getRight()) + 1;
}

//Robert Sedgewick, Algorithms in C++, Parts 1-4, page 250.
int calcHeight(TreeNode * curr){
  if(curr == NULL){
    return 0;
  }
  int left_height = calcHeight(curr->getLeft());
  int right_height = calcHeight(curr->getRight());
  if(left_height > right_height){
    return left_height + 1;
  } else {
    return right_height + 1;
  }
}

int main(int argc, char * argv[]){

  TreeNode * root = makeTree();
  std::cout << "num nodes: " << countNodes(root) << std::endl;
  std::cout << "height: " << calcHeight(root) << std::endl;
  //prints:
  //  num nodes: 11
  //  height: 4

  return 0;
}

```