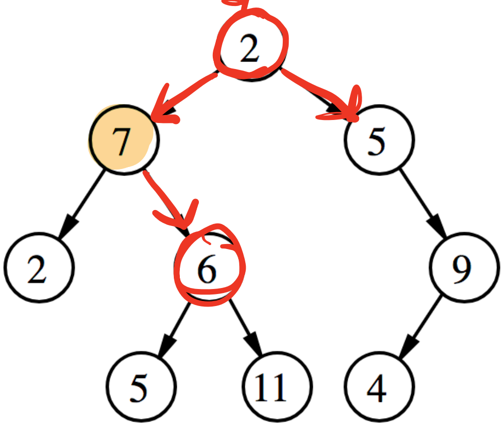
# BINARY SEARCH TREES

Problem Solving with Computers-II

# Trees

Hierarchy

root




Linked List (Linear)

head → 2 → 7 → 6 → 5 → 11

A tree has following general properties:

- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*

2's children are 7 and 5

Parent of 7 is 2

" " 5 " 2

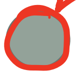
# Which of the following is/are a tree?

Empty

root

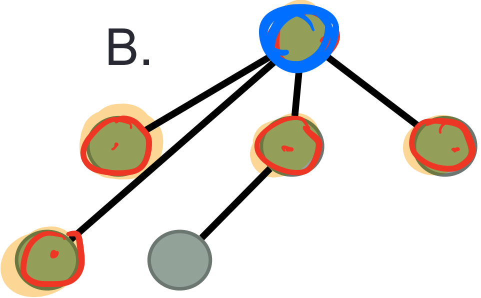
A.

root

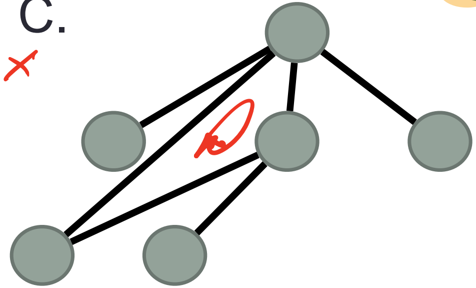


Single node tree.

B.



C.



D. A & B

E. All of A-C

# Binary Search Trees (BST)

*Binary Search

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 5 | 7 | 11 |

1. What are the operations supported?

   All the operations supported by sorted arrays + fast insert and delete

2. What are the running times of these operations?

   efficient

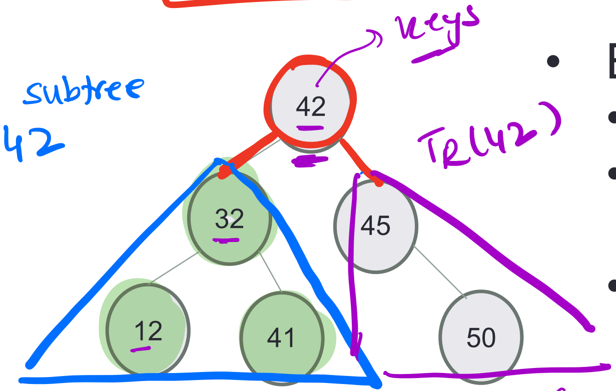
3. How do you implement the BST i.e. operations supported by it?

# Sorted arrays vs Binary Search Trees (BST)

| **Operations** | |
|---|---|
| Min | |
| Max | |
| Successor | |
| Predecessor | |
| Search | |
| Insert | |
| Delete | |
| Print elements in order | |

# Binary Search Tree – What is it?

Binary Trees




- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property

For any node,
Keys in node's left subtree < Node's key
Node's key < Keys in node's right subtree

$\text{keys}(T_L(42)) < 42 < \text{key}(T_R(42))$

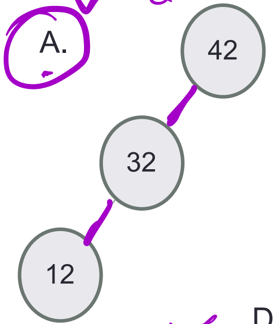
$\text{key } T_L(x) < \text{key}(x) < \text{key } T_R(x)$

Do the keys have to be integers?

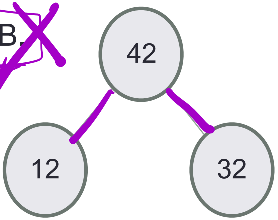
# Which of the following is/are a binary search tree?

BST that looks like a linked list

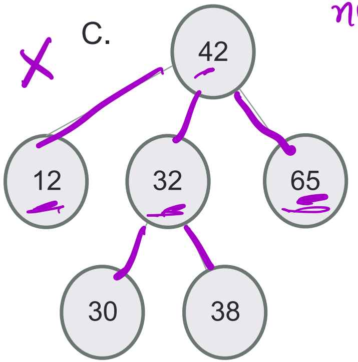
A.




B.

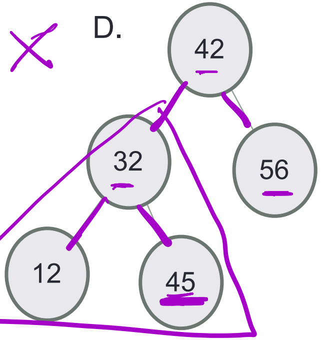



C.




not a binary tree

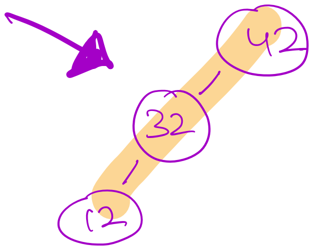
D.



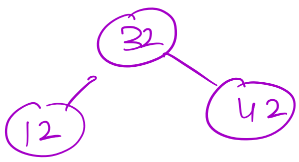

E. More than one of these

45 is more than 42 but its in 42's left subtree

Insert: 42, 32, 12



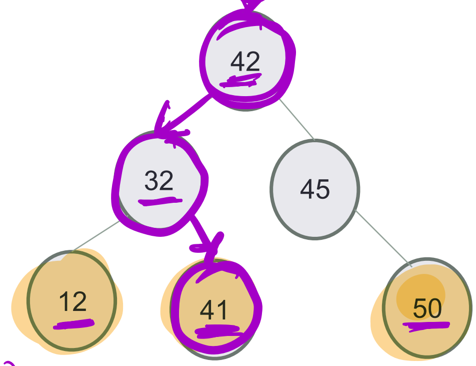

32, 42, 12



# BSTs allow efficient search!

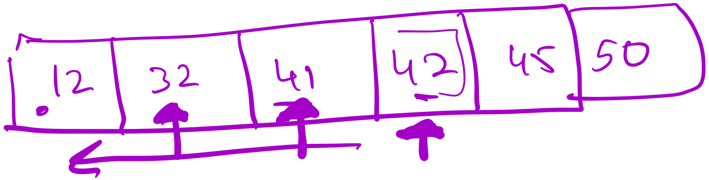
search key : 41

root



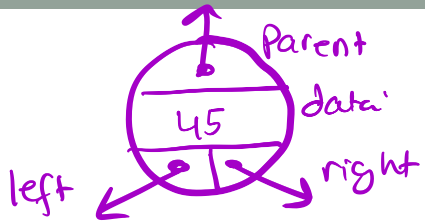

- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x

Sorted array



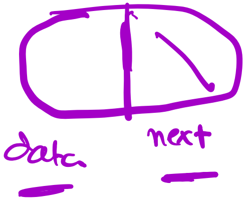

**Search for 41, then search for 53**

# A node in a BST





node in a linked list





```
class BSTNode {

public:
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
    int const data;

    BSTNode( const int & d ) : data(d) {
      left = right = parent = nullptr;
    }
};
```

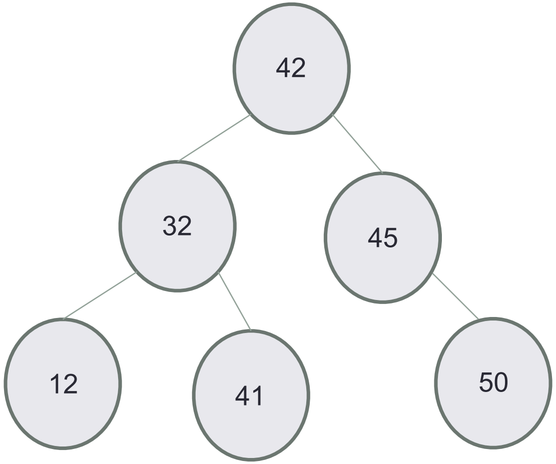
45

initializer list

BSTNode *n = new BSTNode{45};

# Define the BST ADT

| **Operations** |
| --- |
| Search |
| Insert |
| Min |
| Max |
| Successor |
| Predecessor |
| Delete |
| Print elements in order |



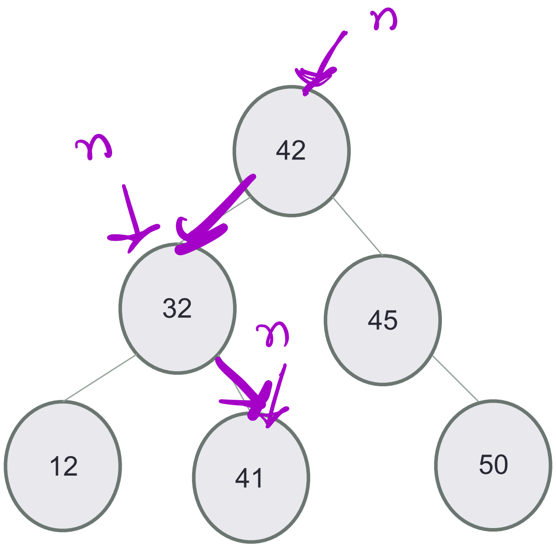
# Traversing down the tree

BSTNode* n=root;

- Suppose n is a pointer to the root. What is the output of the following code:

```
n = n->left;
n = n->right;
cout<<n->data<<endl;
```

A. 42
B. 32
C. 12
D. 41
E. Segfault



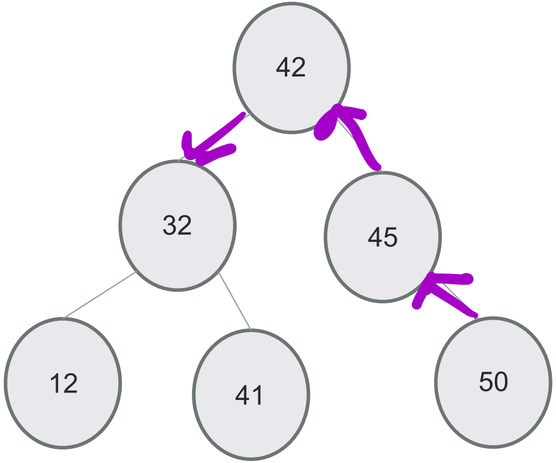
# Traversing up the tree

n $\boxed{\text{nullptr}}$

- Suppose n is a pointer to the node with value 50.
- What is the output of the following code:

```
n = n->parent; //
n = n->parent;
n = n->left;
cout<<n->data<<endl;
```

n = nullptr

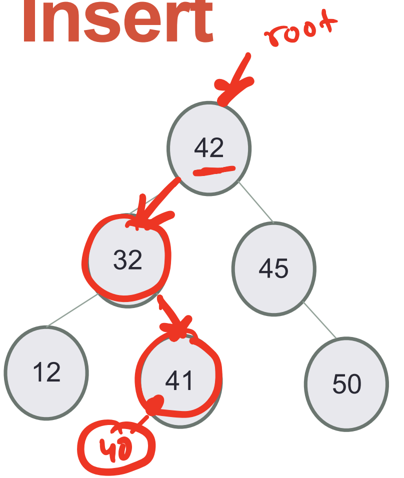



A. 42
B. 32
C. 12
D. 45
E. Segfault

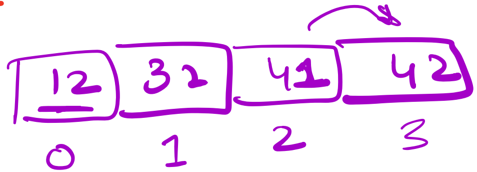
n→left
n→right
n→parent

n is not a null ptr;

# Insert




- Insert 40
- Search for the key
- Insert at the spot you expected to find it




In the case of ~~for~~ sorted array we would need to move the elements once on each insert (we might need to move all elements in the worst case)

The above BST is obtained by inserting keys in the following order:

42, 32, 41, 12, 45, 50

1

# Max , search, insert

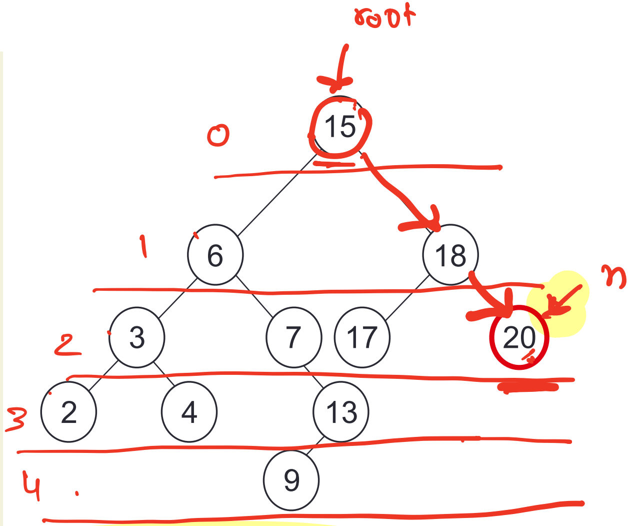
**Goal**: find the maximum key value in a BST

Following right child pointers from the root, until a leaf node is encountered. The least node has the max value

```
# include <limits>
```

**Alg:** `int BST::max()` {

```
    BSTNode *n = root;
    while (n && n->right)
        [n = n->right;]
    }
    if (!n) return std::numeric_limits<int>::max();
    return n->data;
}
```




**Maximum = 20**

# Min

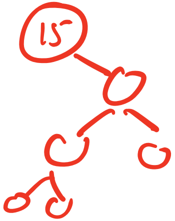
**Goal**: find the minimum key value in a BST

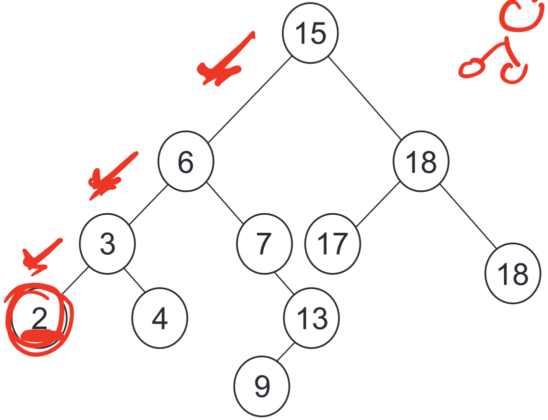
Start at the root.

Follow ________ child pointers from the root, until a leaf node is encountered

Leaf node has the min key value

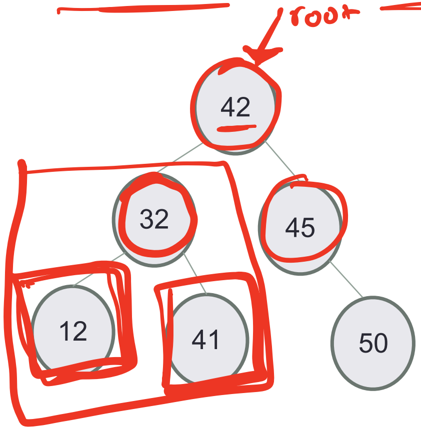
**Alg:** `int BST::min()`




**Min = ?**

# In order traversal: print elements in sorted order




Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root. current node // print the key for this node
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

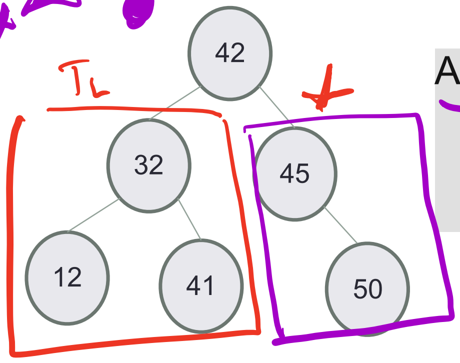
Inorder ( $T_L$(42) )

12 32 41 42

Inorder ( $T_R$(42) )

45 50

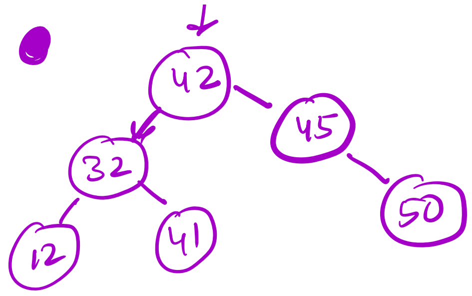
# Pre-order traversal: nice way to linearize your tree!




Algorithm Preorder(tree)

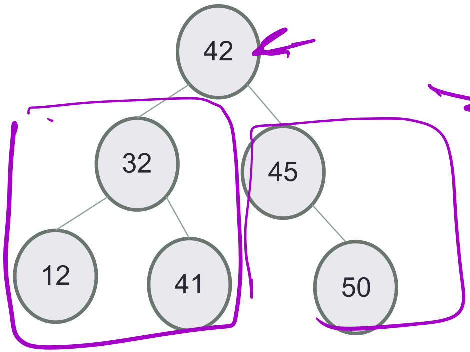
1. Visit the root. print key of the node
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

42 32 12 41 45 50




If we insert the key values from the preorder traversal into an empty tree we will get a duplicate tree

# Post-order traversal: use in recursive destructors!

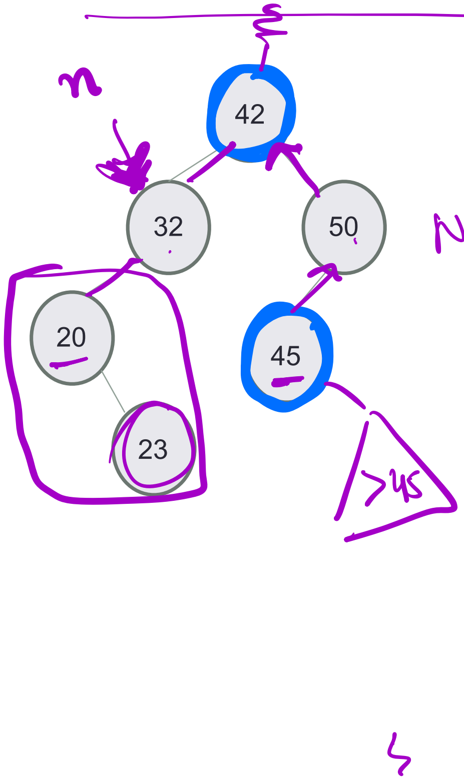



Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

12 41 32 50 45 42

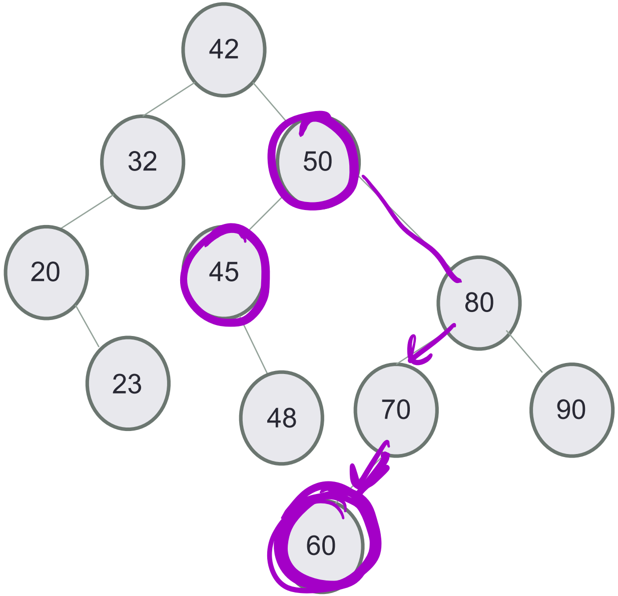
# Predecessor: Next smallest element




- What is the predecessor of 32? 23
- What is the predecessor of 45?

```
Node* predecessor ( Node * n ) {
  if (n -> left) {
    // return the max-node in T_L(n)
  } else {
    // follow parent pointers until you reach a node
       with a value smaller than n -> data
  }
}
```

# Successor: Next largest element



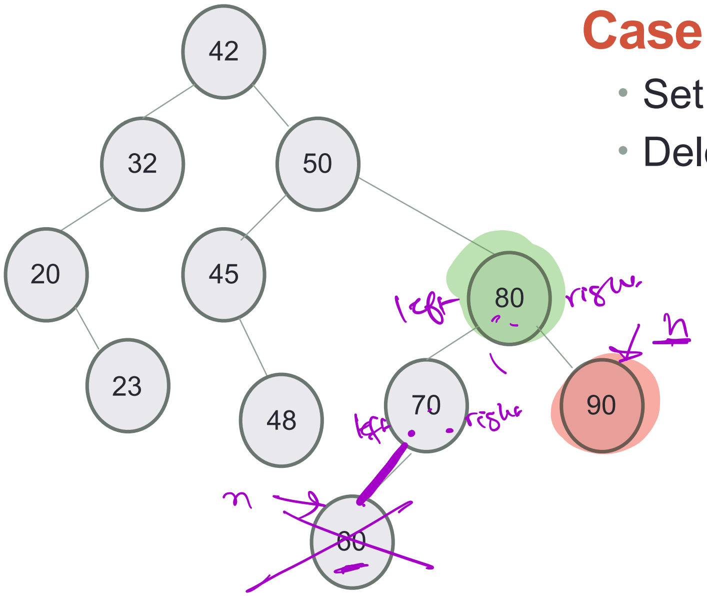

- What is the successor of 45? 48
- What is the successor of 50? 60
- What is the successor of 60? 70

# Delete: Case 1

60 is a leaf node : no children.

## Case 1: Node is a leaf node

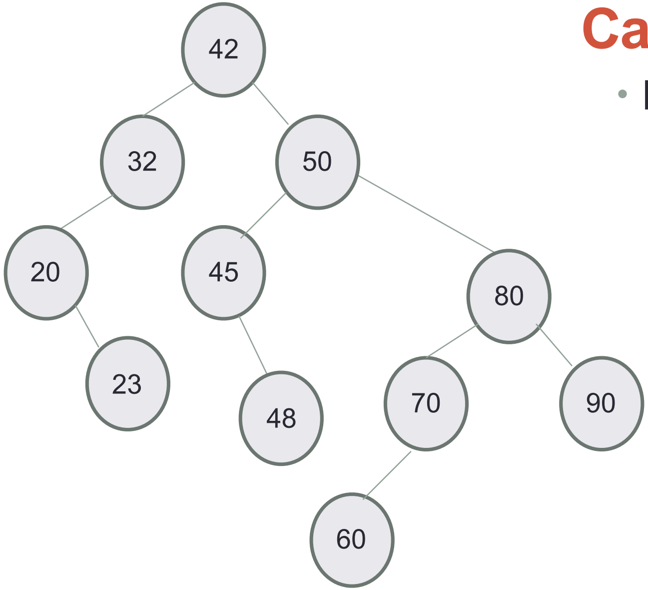
- Set parent's (left/right) child pointer to null
- Delete the node




```
if ( n && !n->left && !n->right ) {
    // leaf node.
    // update n's parent's child pointers.
    if ( n == n->parent->left )
        n->parent->left = nullptr;
    else
        n->parent->right = nullptr;
    delete n;
```

n -> parent -> left is not null ?

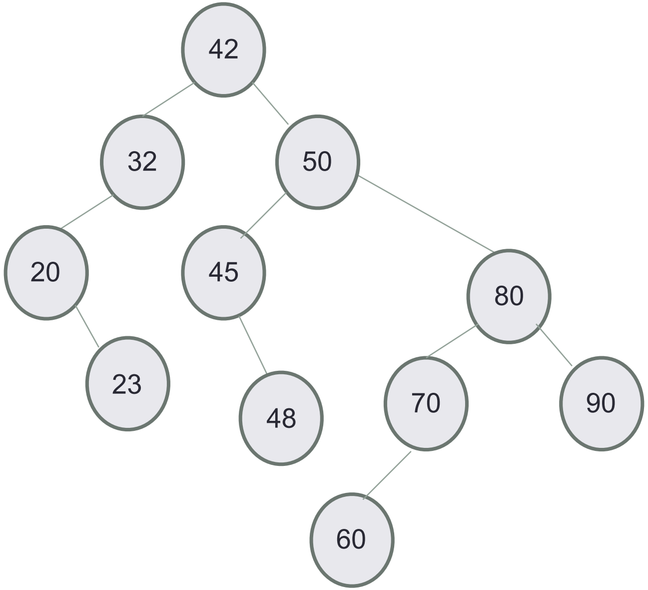
# Delete: Case 2




## Case 2 Node has only one child

- Replace the node by its only child

# Delete: Case 3




## Case 3 Node has two children

- Can we still replace the node by one of its children? Why or Why not?

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

- Ex. Binary search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| lo | | | | | | | | | | | | | | hi |