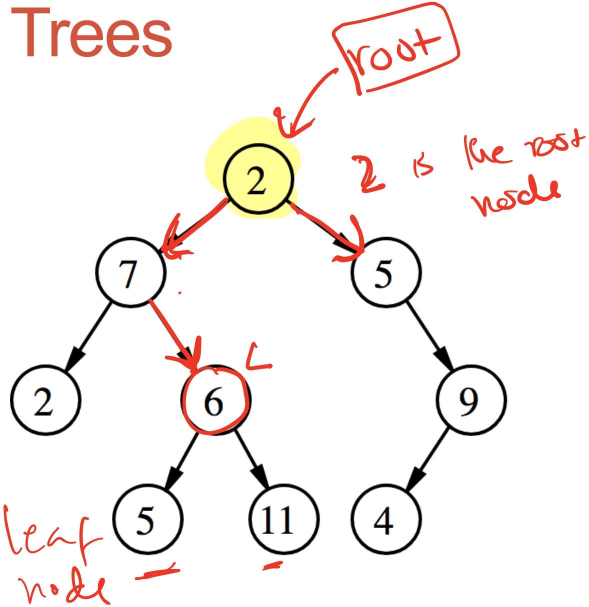
BINARY SEARCH TREES

Problem Solving with Computers-II

Trees

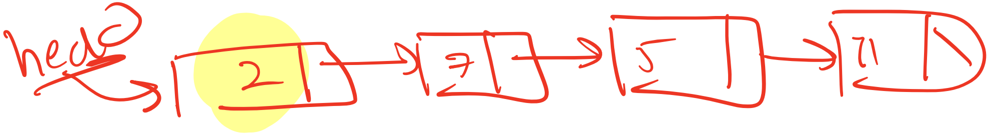


A tree has following general properties:

- 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' children are 7 and 5

Binary tree: every node has at most two children

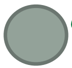


Which of the following is/are a tree?

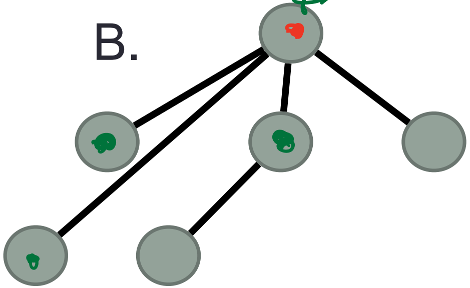
Empty tree

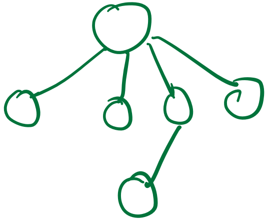
root 

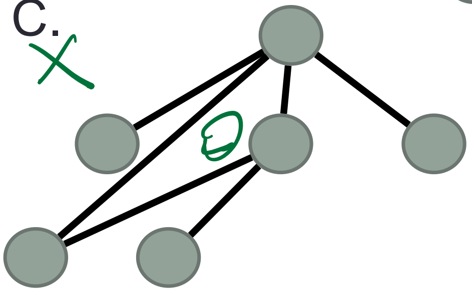
One node tree

A.  root

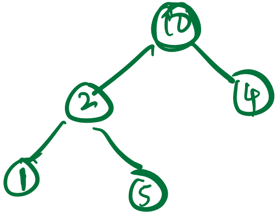
root

B. 



C. 

D. A & B



E. All of A-C

efficiently

Binary Search Trees

(BST)

1. What are the operations supported?

sorted array + fast insert and delete

search, min, max . . . . .

2. What are the running times of these operations?

next lecture

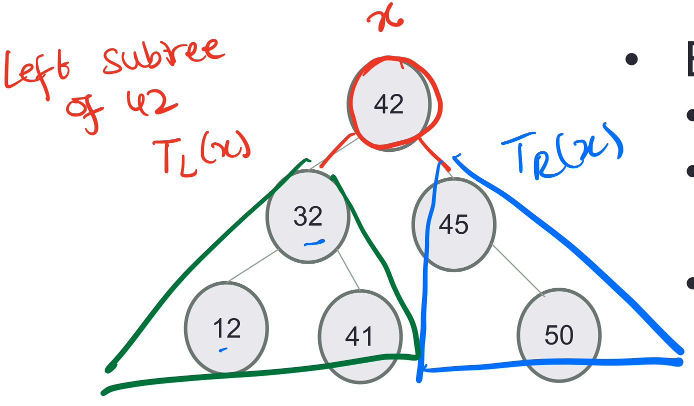
3. How do you implement the BST i.e. operations supported by it?

Operations supported by Sorted arrays and Binary Search Trees (BST)

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

Binary Search Tree – What is it?

no duplicates!



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

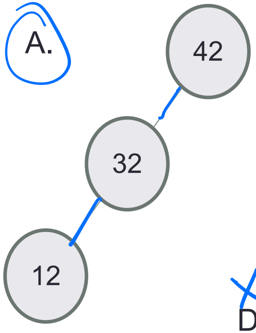
For any node,
Keys in node's left subtree < Node's key
Node's key < Keys in node's right subtree

$T_L(x) < x < T_R(x)$

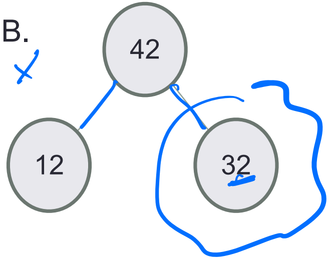
Do the keys have to be integers?

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

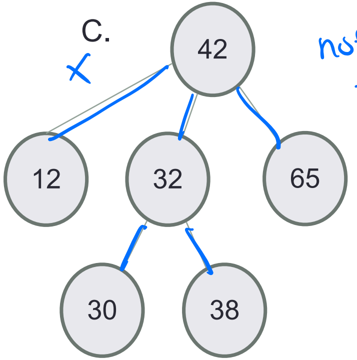
A.



B.

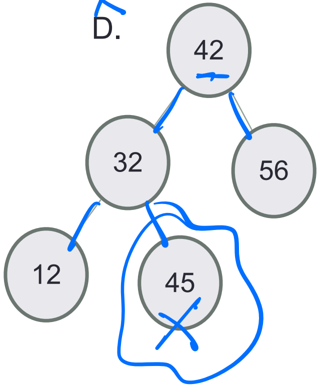


C.



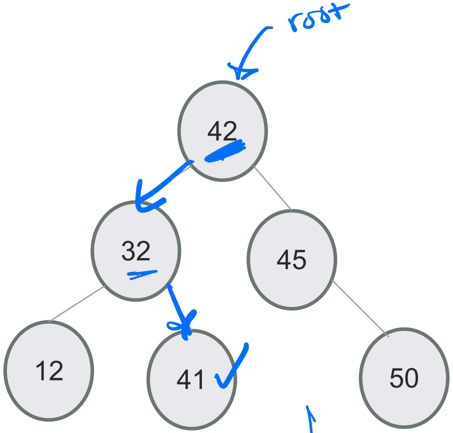
not a binary tree

D.

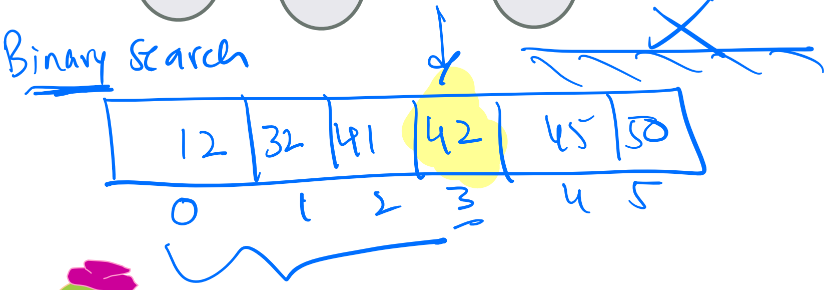


E. More than one of these

BSTs allow efficient search!



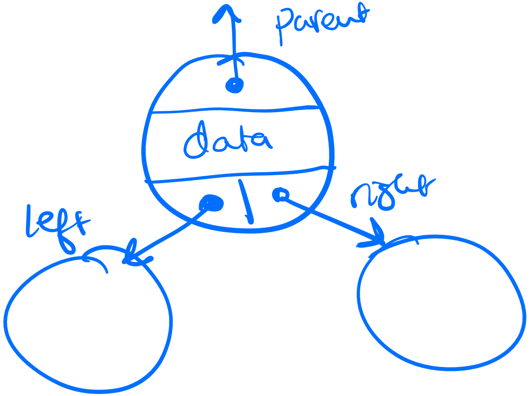
- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x



Search for 41, then search for 53

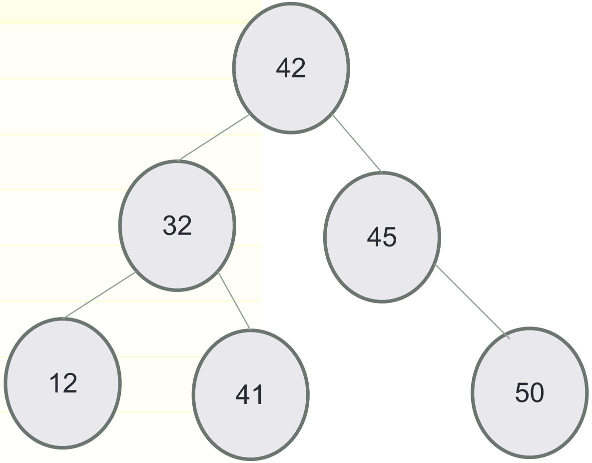
A node in a BST

```
class BSTNode {

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

  BSTNode( const int & d ) : data(d) {
    left = right = parent = 0;
  }
};
```



Define the BST ADT

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

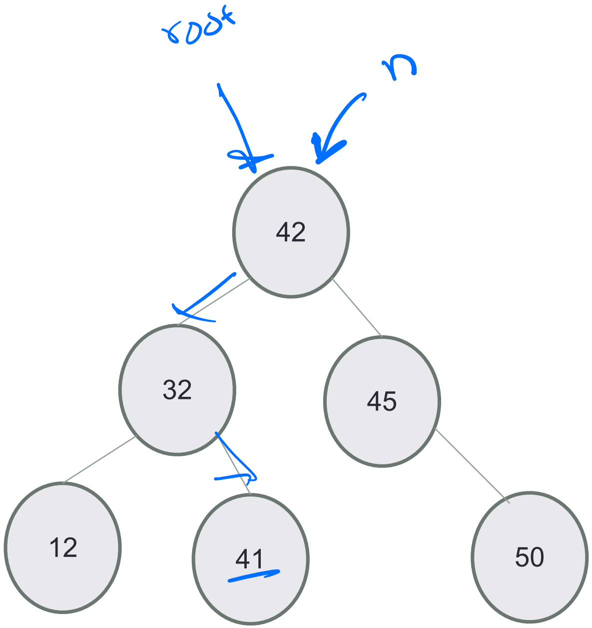


Traversing down the tree

- 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



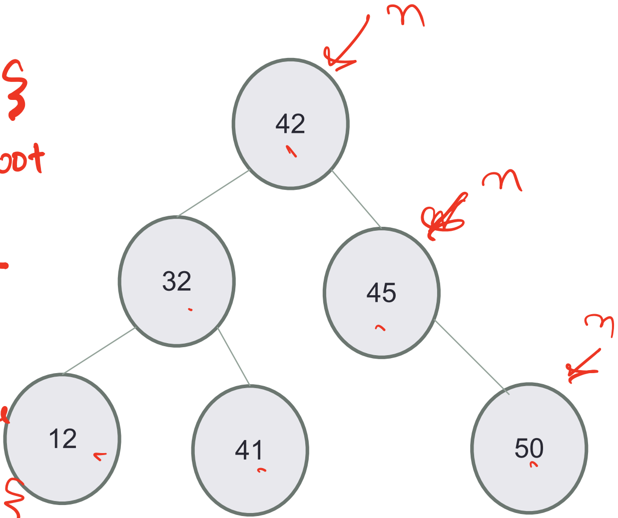
Traversing up the tree

- 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;
```

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

```
if (!n->parent) {
    // at the root
}
if (!n) {
    return
}
// loop to traverse to the root node
while (n && n->parent) {
    n = n->parent;
}
```



(1) Insert a sequence of key iteratively to build a BST

42, 32, 12, 41, 45, 50

(2) The final structure of the BST depends on the order we insert the keys.

Start with empty BST

Insert (32)

Insert (42)

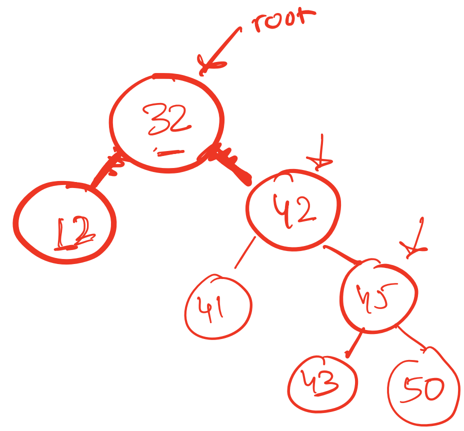
Insert (12)

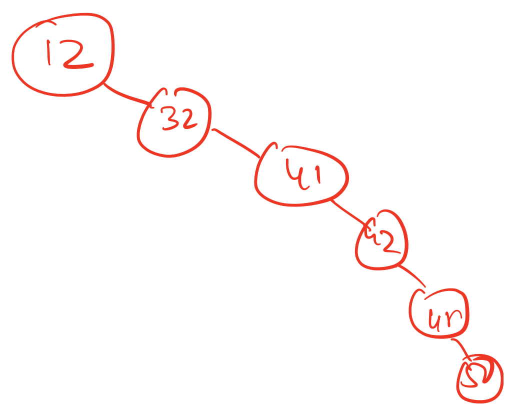
Insert (41)

Insert (45)

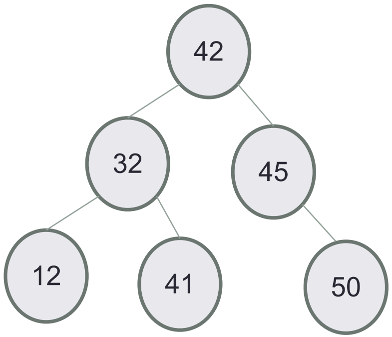
Insert (50)

Insert (43)





Insert



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

Max

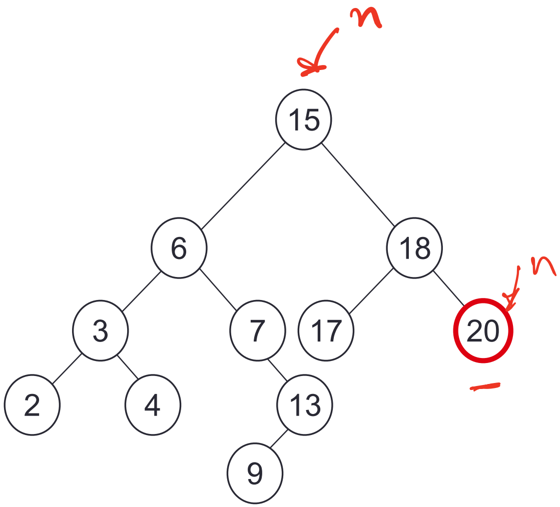
Goal: find the maximum key value in a BST

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

```
# include <limits>
```

Alg: `int BST::max()`

```
{
    BSTNode * n = root;
    if ( !n) {
    return std::numeric_limits<int>::min();
    }
    while ( n->right) {
        n = n->right;
    }
    return   n->data;
}
```



Maximum = 20

Min

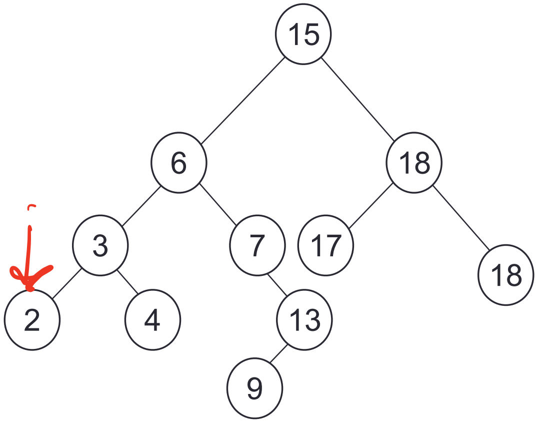
Goal: find the minimum key value in a BST

Start at the root.

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

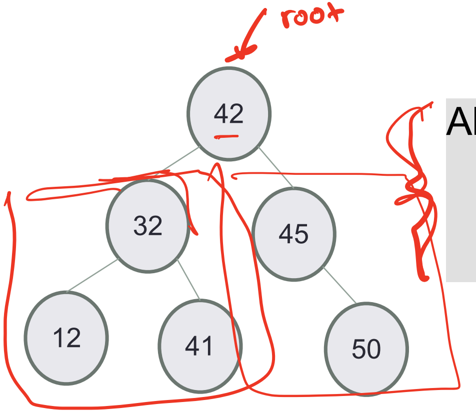
Leaf node has the min key value

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.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Inorder(T_L(42))   InOrder(T_R(42))

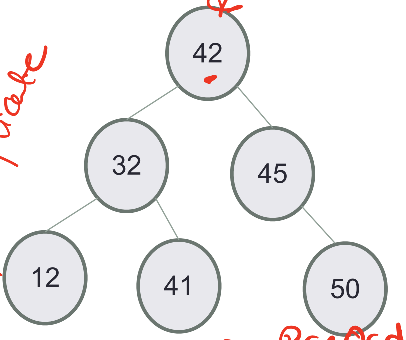
12 32 41	42	45 50

Output is sorted

```
InOrder ( r ) {
    if (!r) return
    Inorder ( r-> left )
    cout << r-> data
    Inorder ( r-> right )
}
```

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

root



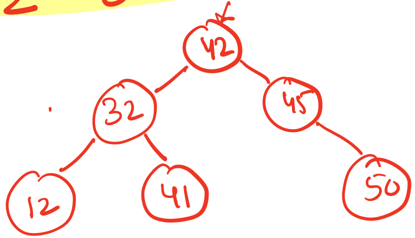
duplicate

Algorithm Preorder(tree)

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

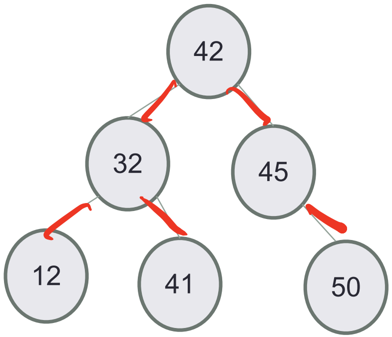
Output of PreOrder on the above tree

42 32 12 41 45 50




```
PreOrder ( r ) {
    if (!r) return
    cout << r-> data
    PreOrder ( r-> left )
    PreOrder ( r-> right )
}
```

Post-order traversal: use in recursive destructors!



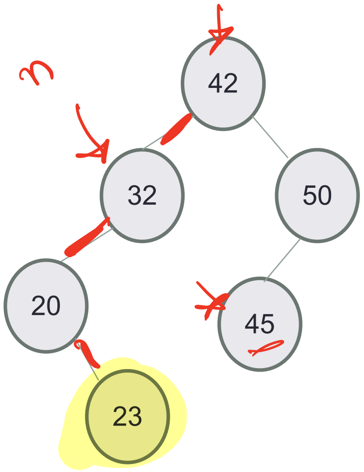
Algorithm Postorder(tree)

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.

```
PostOrder ( r ) {
    if (!r) return
    Post order ( r-> left)
    PostOrder ( r-> right )
    cout << r-> data
}
```

Hint:
Use in clear method
to delete all
the nodes

Predecessor: Next smallest element



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

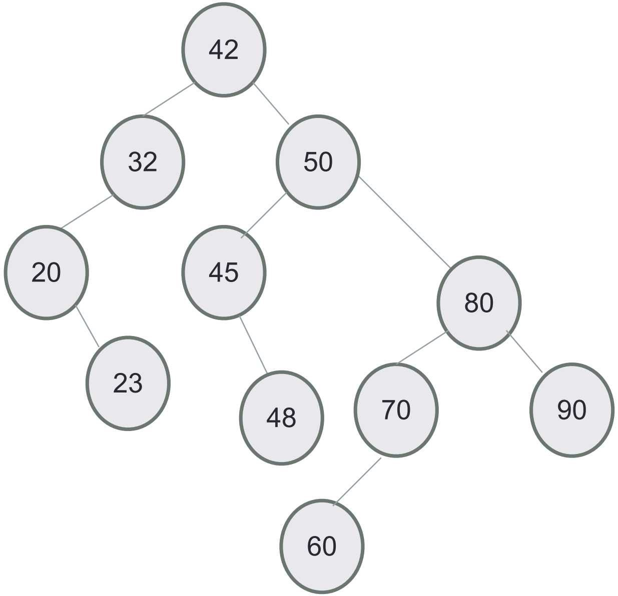
20	23	32	42	45	50

p

```
predecessor (r) {
  if (r→left) {
    return the max value in the left subtree of r
  } else {
    traverse parent pointers until you find a node (n) with key value less than the key of r
    return n→data;
  }
}
```

$x < T_L(32) < 32$

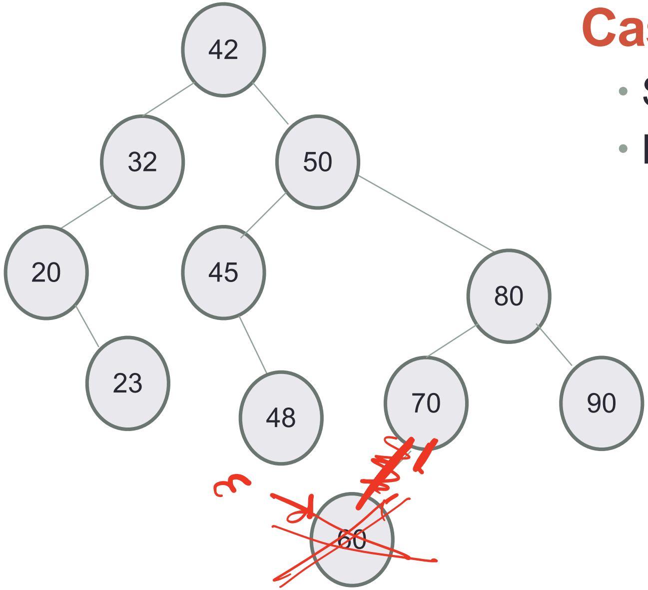
Successor: Next largest element



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

Delete: Case 1

bst. erase (60)



Case 1: Node is a leaf node

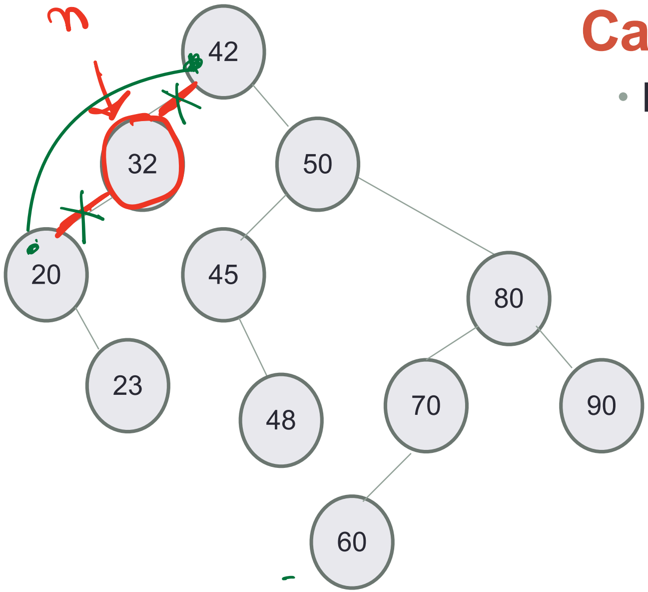
- Set parent's (left/right) child pointer to null
- Delete the node

check if the node is a leaf node

```
if ( !n→left && !n→right ){
    // case 1
    if ( n→parent && n == n→parent→left ){
        n→parent→left = nullptr;
    }
    // add more code .
    delete n;
}
```

Delete: Case 2

bst.erase(32)



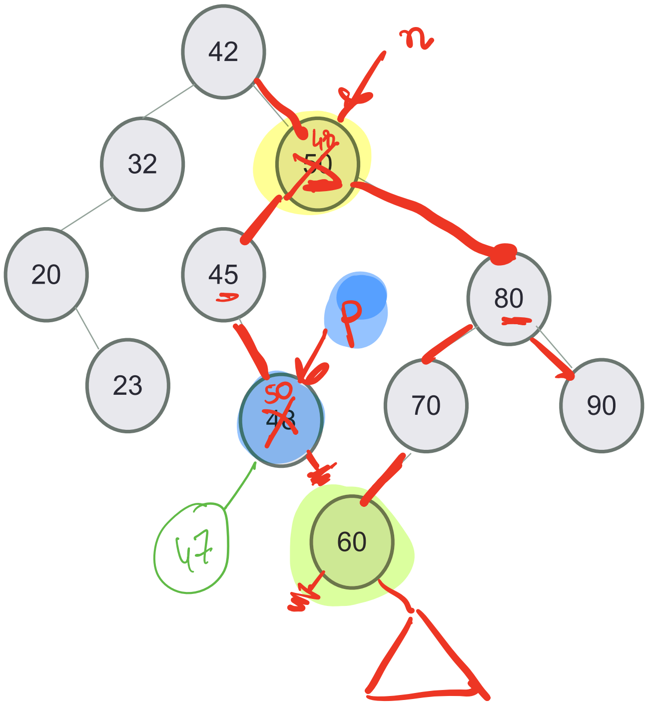
Case 2 Node has only one child

- Replace the node by its only child

delete n;

Delete: Case 3

bst . erase (50)



Case 3 Node has two children

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

1. swap the key of the node that we want to delete with its successor or predecessor
2. delete the node that used to be the predecessor (or successor) using either case 1 or case 2 logic

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