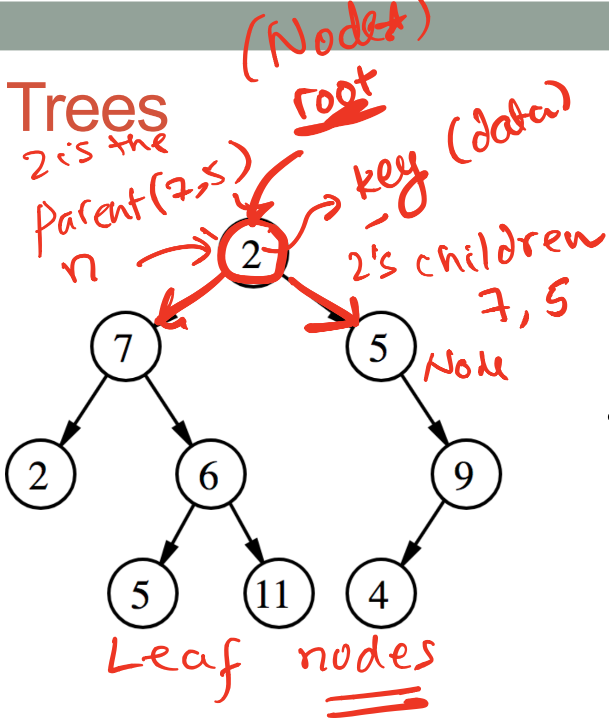
BINARY SEARCH TREES

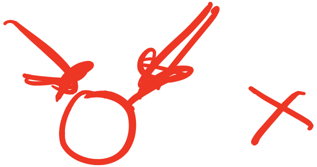
Problem Solving with Computers-II

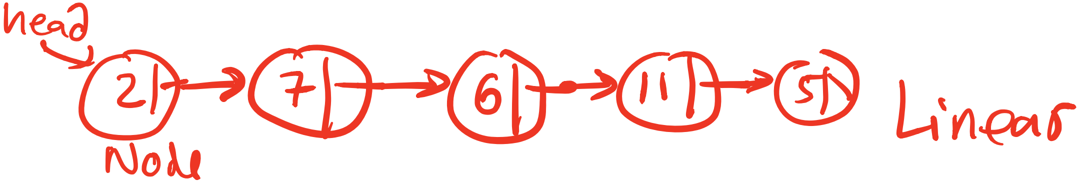
Trees



A tree has following general properties:

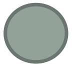
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*





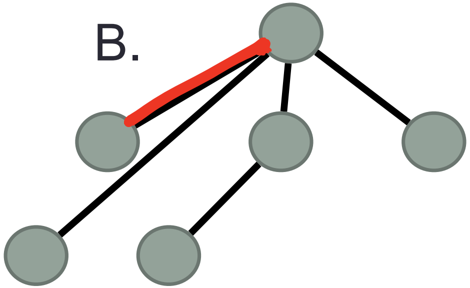
Which of the following is/are a tree?

A.

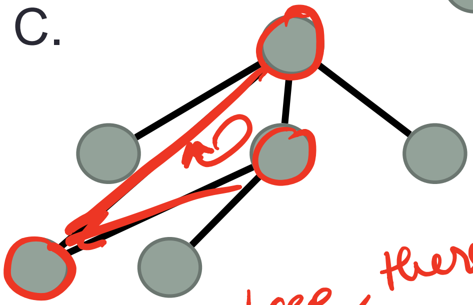


(Single node tree)

B.



C.



not a tree, there is a loop

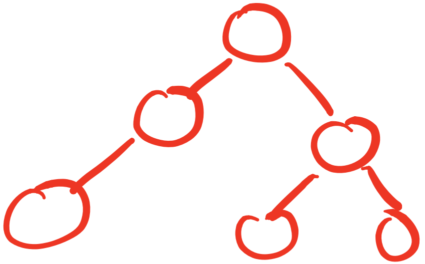
D. A & B

E. All of A-C

Binary Tree

Tree where every node can have at most two children

Binary Search Trees



- What are the operations supported?

  Same operations as linkedlist or array :

  Sorted array + fast insert & delete.

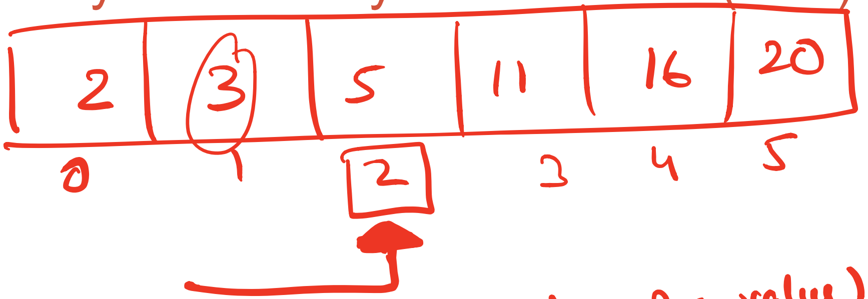
- What are the running times of these operations?

  Build intuition → formalize complexity next week.

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

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

Sorted array

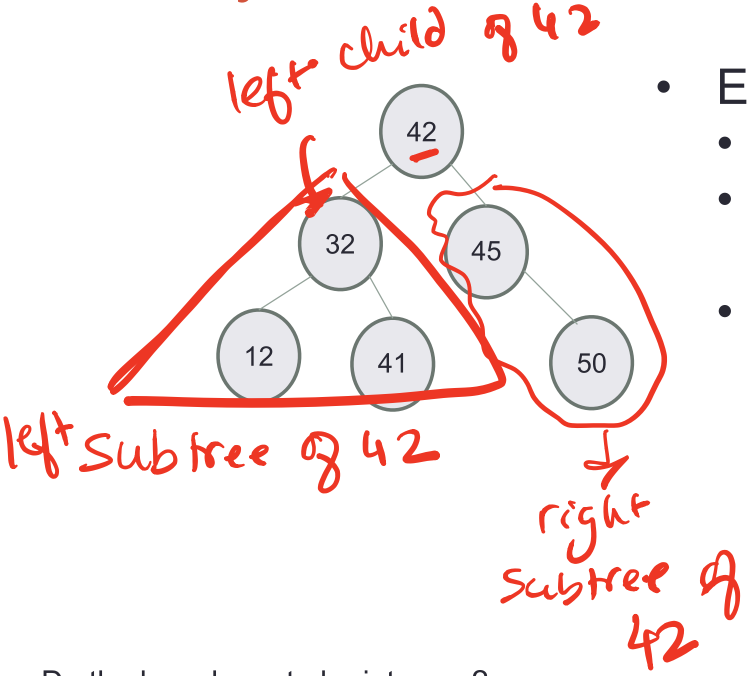


Operations
Min
Max
Successor
Predecessor
Search
Insert
Delete
Print elements in order

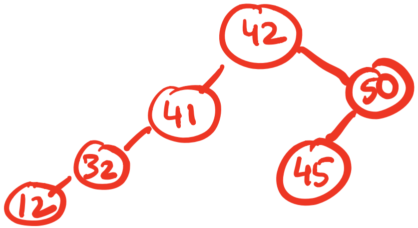
given a value (or index of a value)
find the next largest value

slower in sorted arrays.

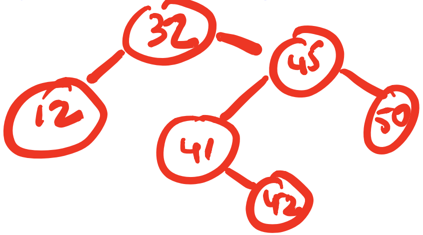
Binary Search Tree – What is it?



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



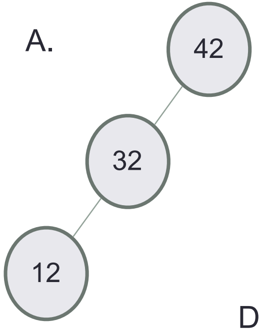
For any node,
Keys in node's left subtree <= Node's key
Node's key < Keys in node's right subtree



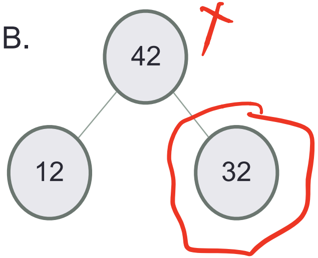
Do the keys have to be integers?

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

A.

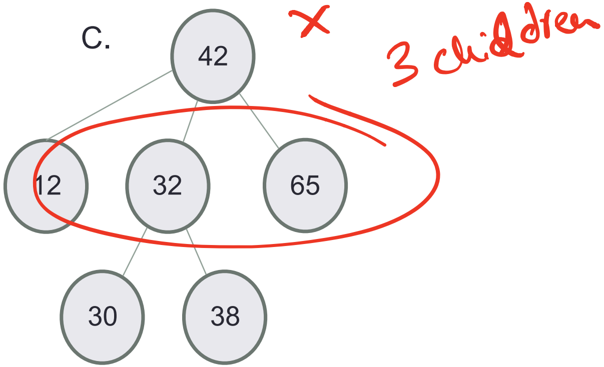


B.

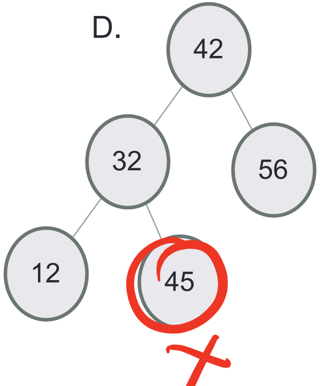


32 < 42
but it's in
42's right
subtree

C.

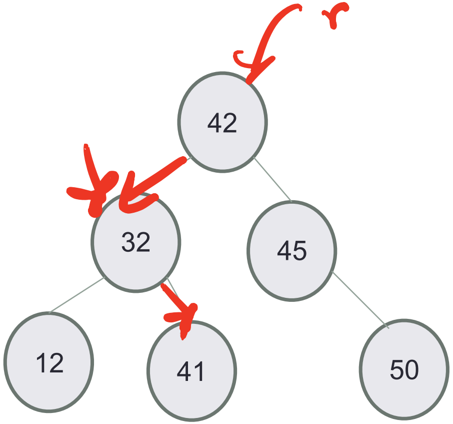


D.

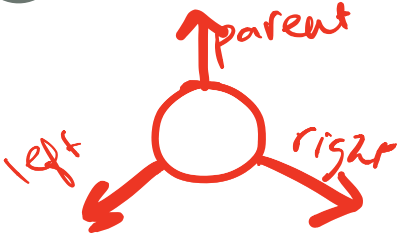


E. More than one of these

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




```
class Node {
public:
    int data;
    Node* left;
    Node* right;
    Node* parent;
}
```

Value $\boxed{41}$

Search for 41, then search for 53

A node in a BST

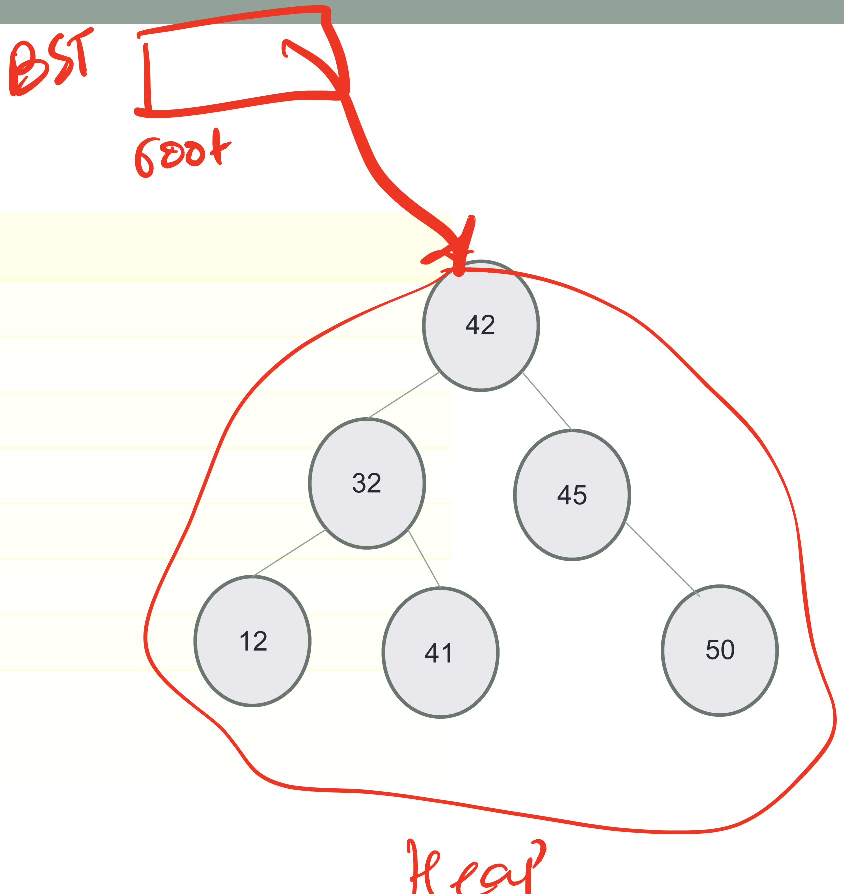
```
class BSTNode {

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

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

Define the BST ADT

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

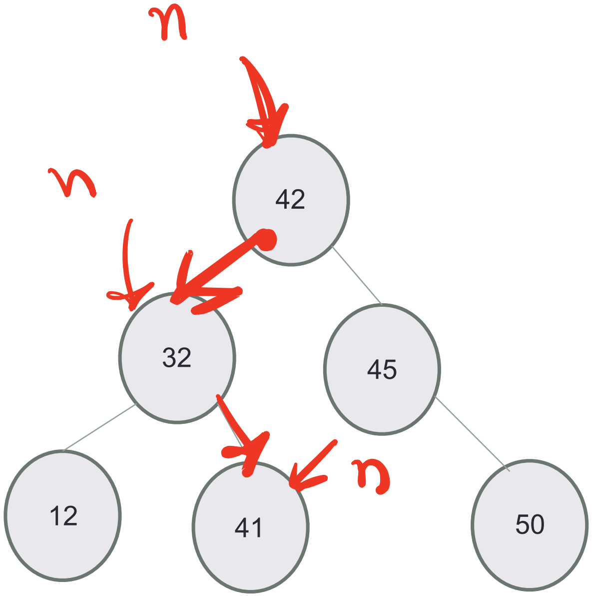


Traversing down the tree

- 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



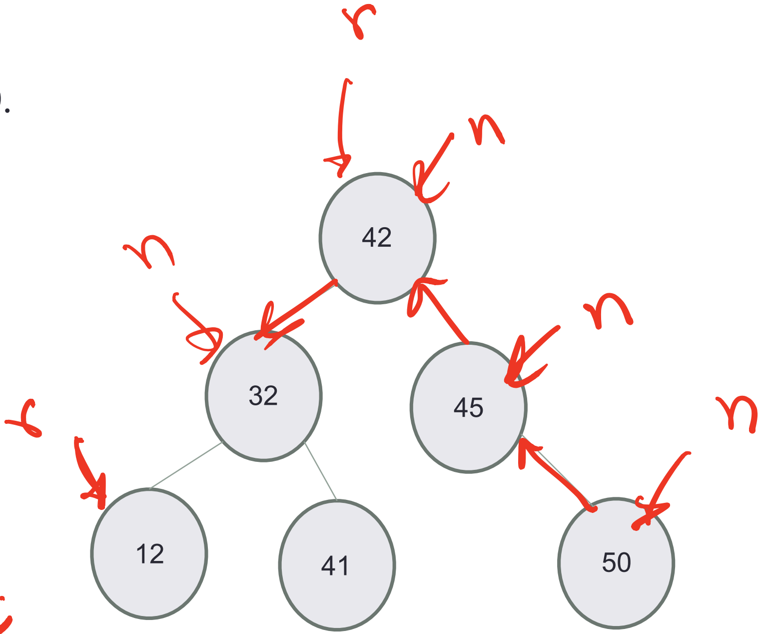
Traversing up the tree

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

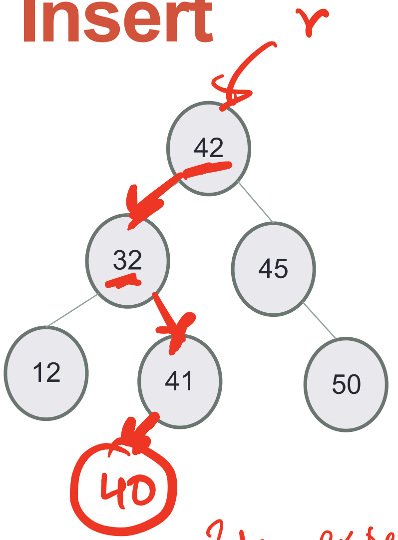
```
n = n->parent;
n = n->parent;
n = n->left;
cout<<n->data<<endl;
```

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

while(r&&r->left)
r= r->left;



Insert



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

We expect to find 40 in the left subtree of 41

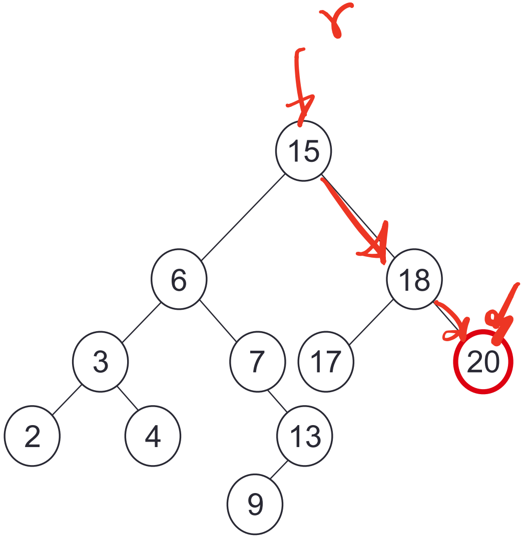
Max

Goal: find the maximum key value in a BST

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

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

```
Node * n = root;
while (n && n->right)
    n = n->right;
return n;
}
```



Maximum = 20

Min

Goal: find the minimum key value in a BST

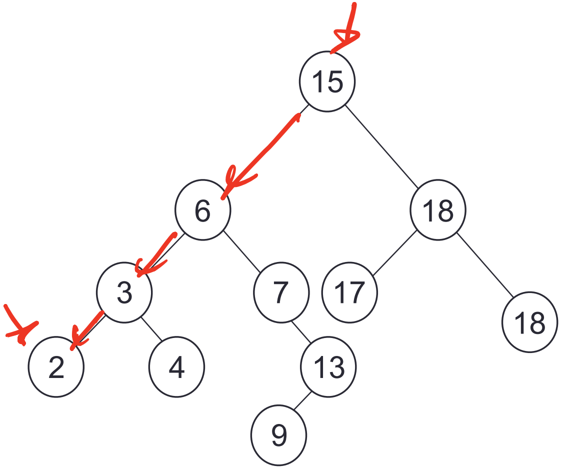
Start at the root.

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

Leaf node has the min key value

Alg: `int BST::min()`

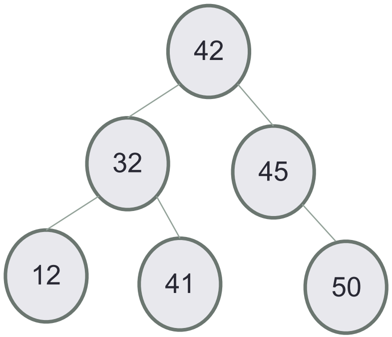
Similar to max



Min = ?

In order traversal: print elements in sorted order

r: pointer to the root of a BST

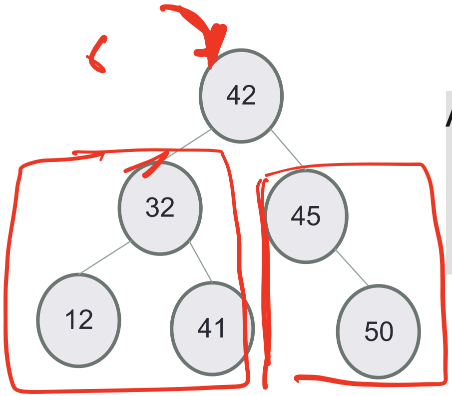


Algorithm Inorder(tree)
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root. ← print the root
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

12 32 41 42 45 50

```
void Inorder ( Node * r ) {
    if (!r) return
    Inorder (r->left);
    cout<< r->data;
    Inorder (r->right);
}
```

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



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)

```
void BST::PreOrder( Node* r ) {
    if(!r) return
    cout<< r->data <<endl;
    PreOrder(r->left);
    PreOrder(r->right);
}
```

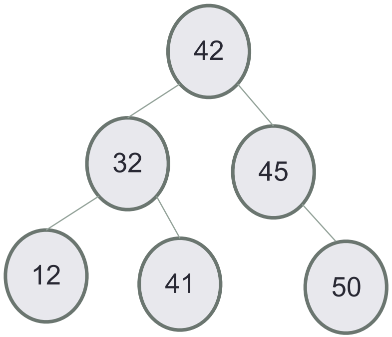
42 $\underbrace{32 \quad 12 \quad 41}$ $\underbrace{45 \quad 50}$

PreOrder on left subtree of 42

PreOrder on right subtree of 42

Post-order traversal: use in recursive destructors!

(See the last slide)



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.

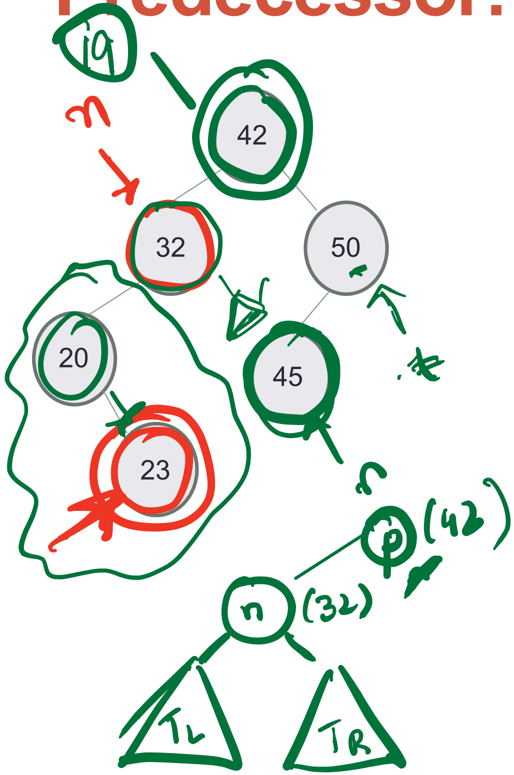
```
void BST::Postorder(Node* r){
if(!r) return;
 Postorder(r->left);
 Postorder(r->right);
 cout << r->data;
}
```

A. 12 32 41 45 50 42

B. 12 41 32 45 50 42

C. 12 41 32 50 45 42

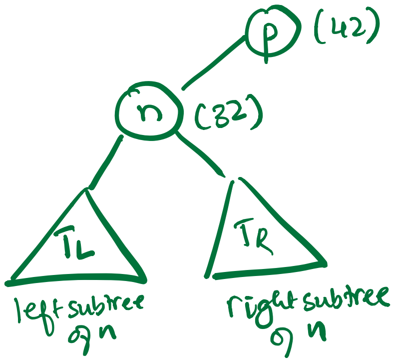
Predecessor: Next smallest element



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

```
Node * BST:: Predecessor (Node * n) {
    if (n→left) {
        // predecessor of n is in its left subtree
        // return the node with max key in n's left
        // subtree
    } else {
        // Go up the tree until you find a node whose
        key is smaller than the key of n
        Node * t = n→parent;
        while (t && t→data >= n→data)
            t = t→parent;
    }
    return t;
}
```

Case 1: n has a left subtree(T_L)(possibly also a right subtree) and its the left child of its parent

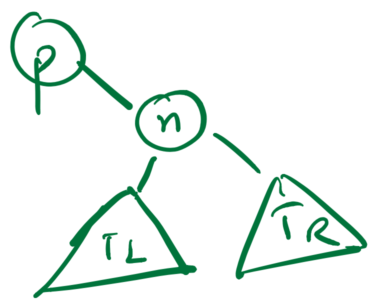


Using the property of BST

$$key(T_L) < key(n) < key(T_R) < key(p)$$

From the above inequality the predecessor of n has to be in its left subtree (T_L) and not in T_R or p

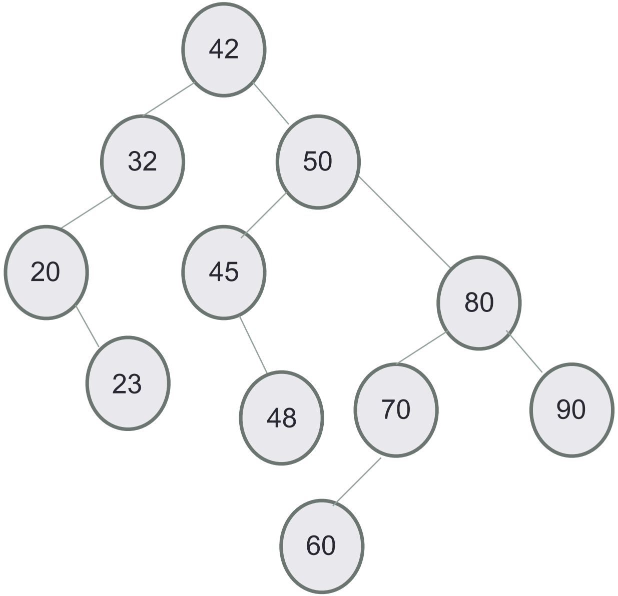
Case 2: n has a left subtree (T_L), possibly a right subtree (T_R) and n is the right child of its parent



$$key(p) < key(T_L) < key(n) < key(T_R)$$

In this case again the pred(n) which is the next smallest node should be in T_L

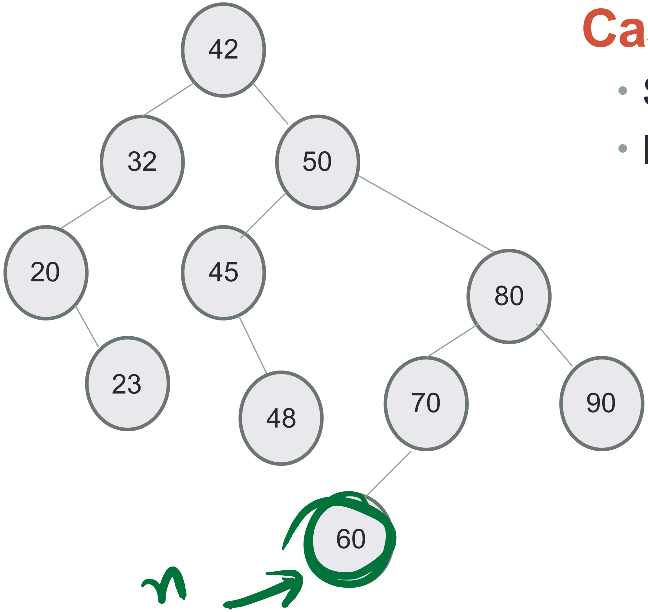
Successor: Next largest element



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

Similar to predecessor

Delete: Case 1

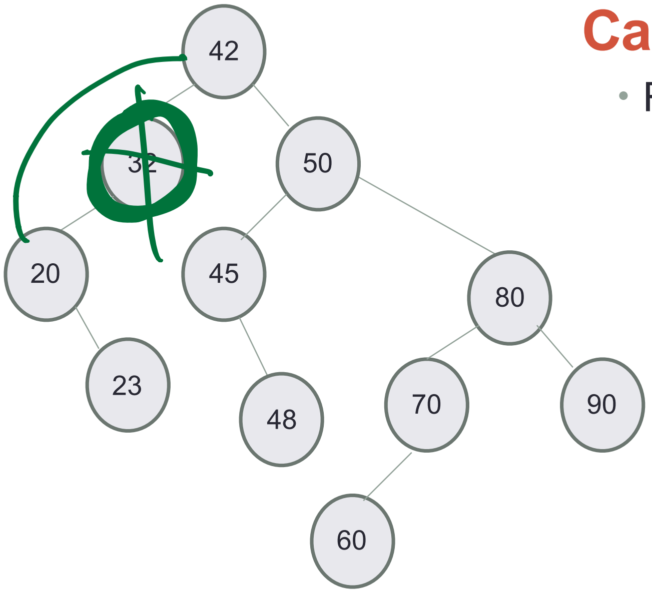


Case 1: Node is a leaf node

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

```
if (!n->left && !n->right){
    // leaf node
    if (n->parent->left == n)
        n->parent->left = nullptr
    else
        n->parent->right = nullptr
    delete n;
}
```

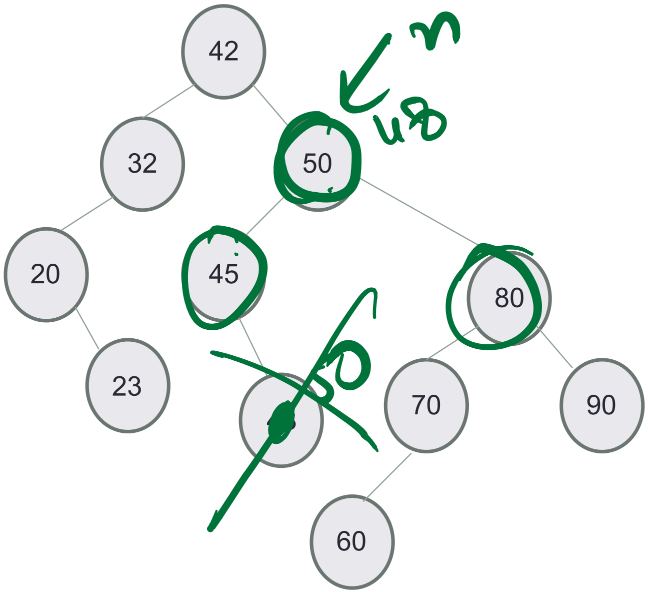
Delete: Case 2



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?

Swap the key of the node with that of its predecessor (or successor)

Delete the node that has the key value

We know that this node has only one child so the deletion defaults to one of the previous two (easier) cases.

Recursive destructor using post order traversal

```
BST :: ~BST () {
        delete root;
}

Node :: ~Node () {
        delete left;
        delet right;
}
```

Recursive deletion using post order traversal these lines call the destructor of Node

In class we discussed different variations and incorrect versions that lead to memory leaks & segfaults