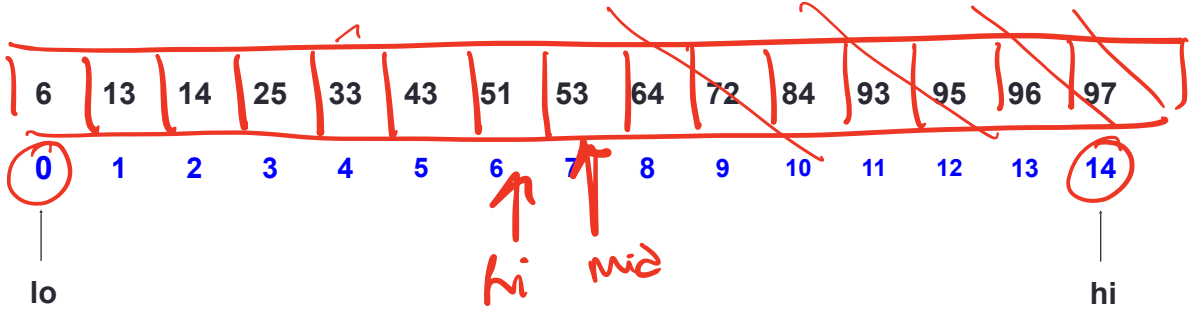
BINARY SEARCH TREES

Problem Solving with Computers-II

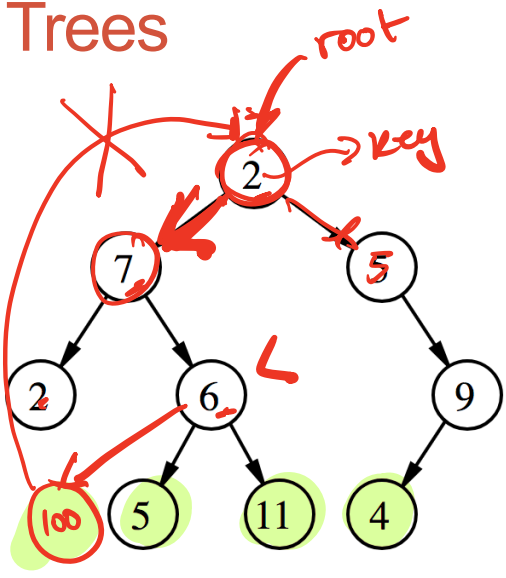
Binary Search

- 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.



Trees



A tree has following general properties:

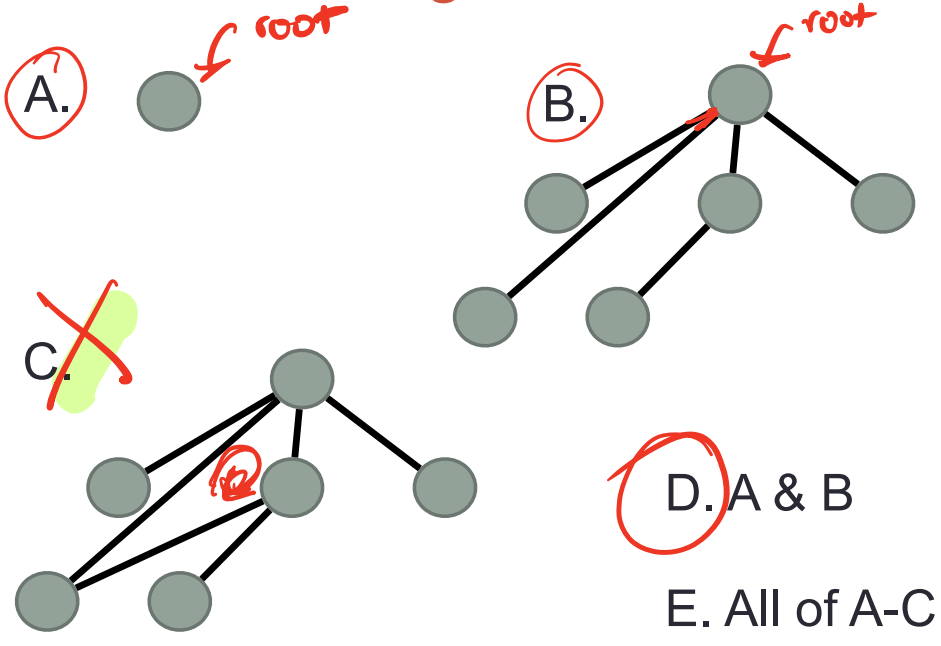
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*

2's children are 7 and 5

parent of 7 is 2

parent of 5 is 2

Which of the following is/are a tree?



empty tree
null
root

Binary Search Trees

- What are the operations supported?

all the operations possible with sorted arrays + fast insert/delete

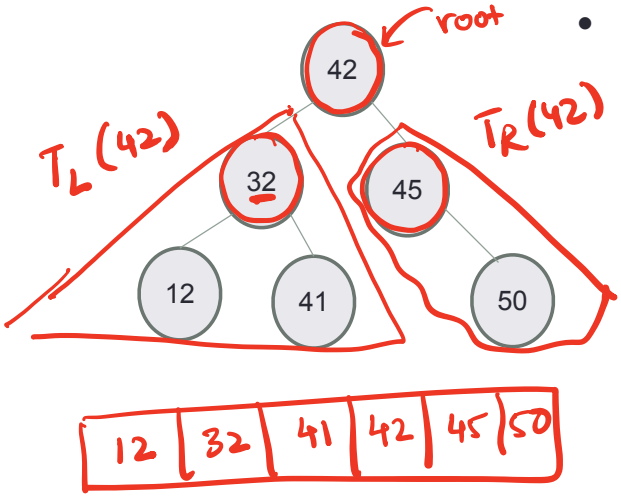
- What are the running times of these operations?

- 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 next larger value
Predecessor next smaller value
Search
Insert
Delete
Print elements in order

Binary Search Tree – What is it?



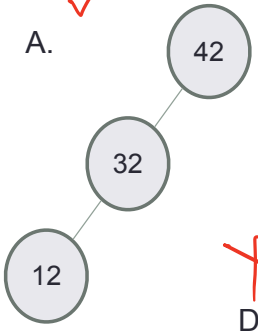
- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property

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

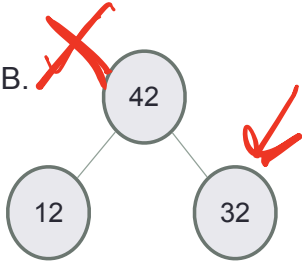
Do the keys have to be integers?

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

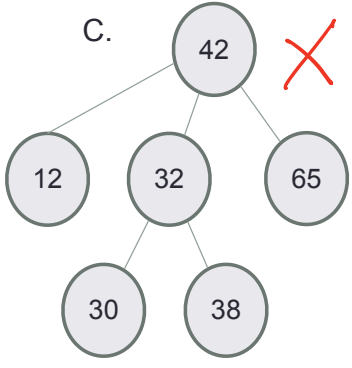
A.



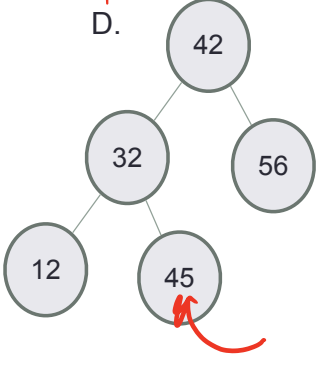
B.



C.

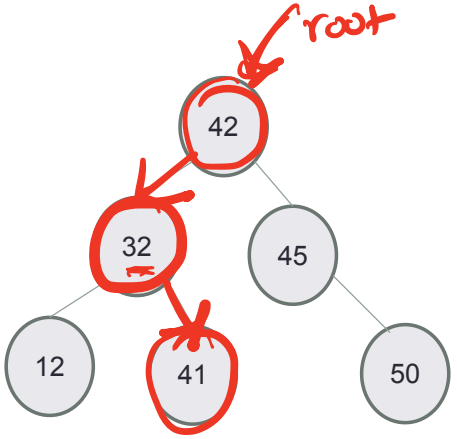


D.

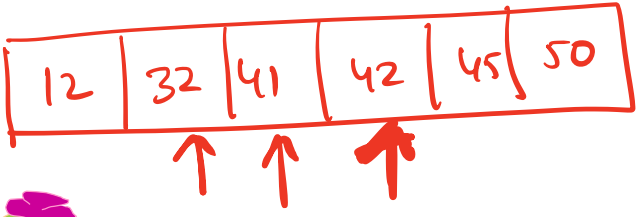


E. More than one of these

BSTs allow efficient search!

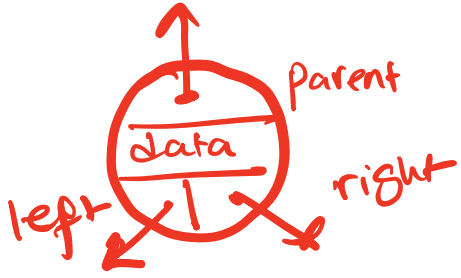


- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
  - If the keys are equal: we have found the key
  - If **k** < key[x] search in the left subtree of x
  - If **k** > key[x] search in the right subtree of x



Search for 41, then search for 53

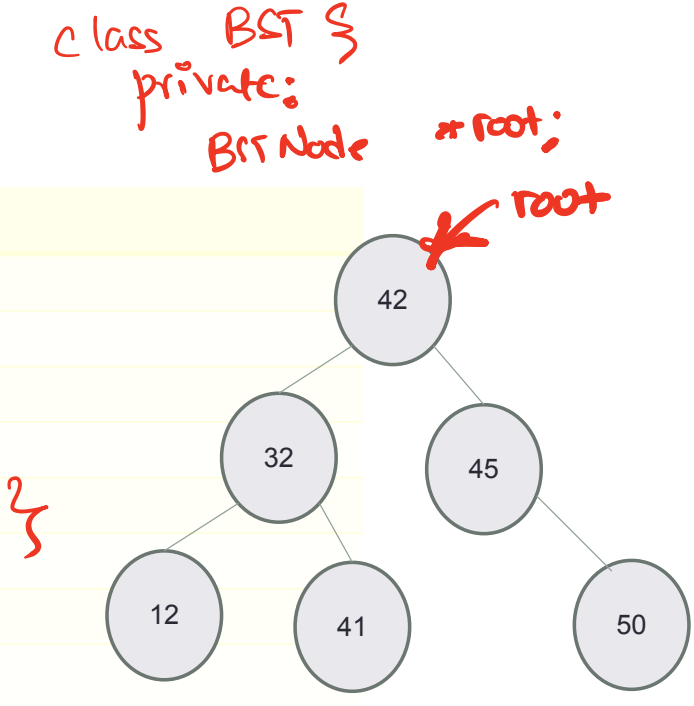
A node in a BST




```
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

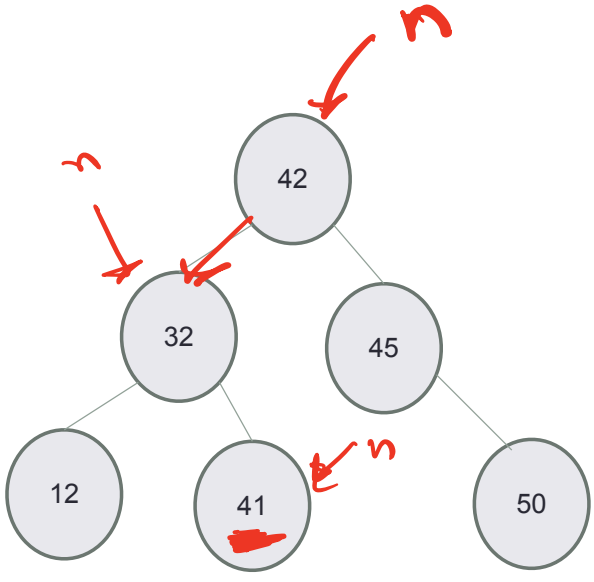


Traversing down the tree

- 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

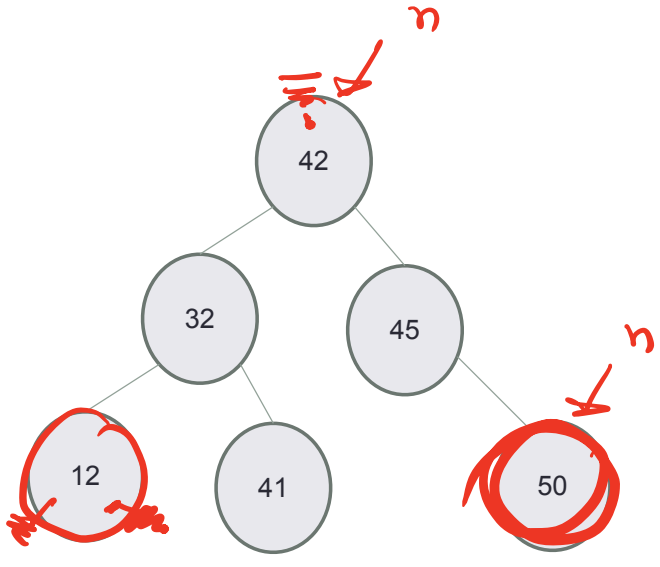


Traversing up the tree

- 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;
```

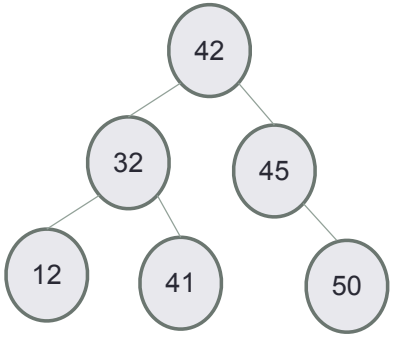
A. 42
B. 32
C. 12
D. 45
E. Segfault



Write a while loop to reach the root node, given a pointer to a node, n

```
while (n && n->parent) {
    n = n->parent;
}
```

Insert



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

42, 41, 32, 12, 45, 50

Max

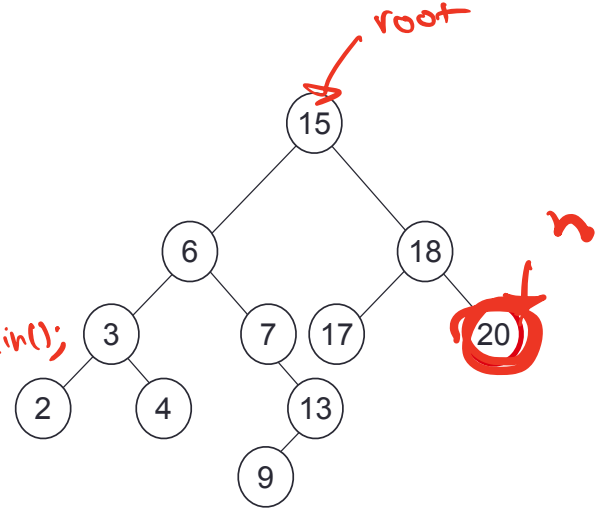
Goal: find the maximum key value in a BST

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

```
#include <limits.h>
```

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

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



Maximum = 20

Min

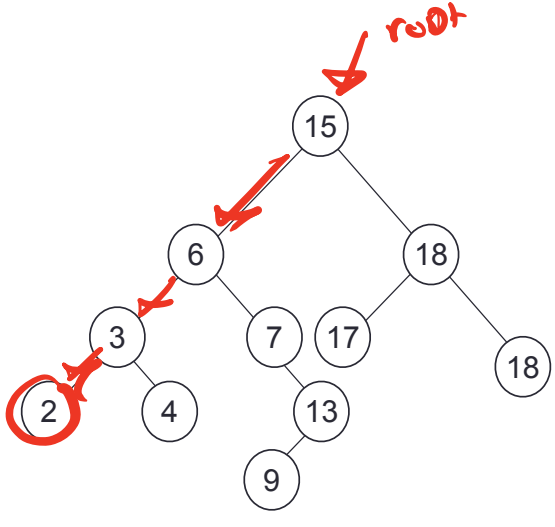
Goal: find the minimum key value in a BST

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 = ?

How is lab 03 going?

A. Done

B. On track to finish

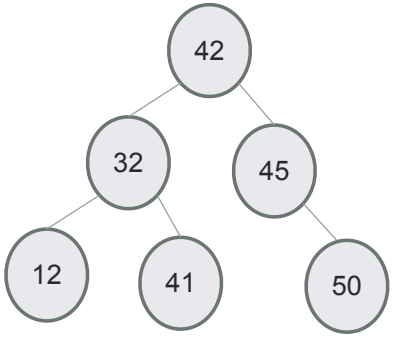
C. Struggling

D. Haven't started

lab 04 released, due next Thurs

midterm, next Wed. check out
gaucho space for past midterm

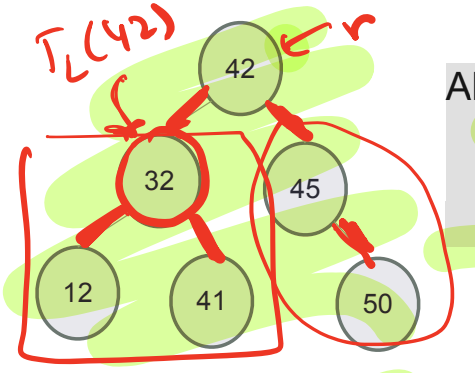
In order traversal: print elements in sorted order



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)

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



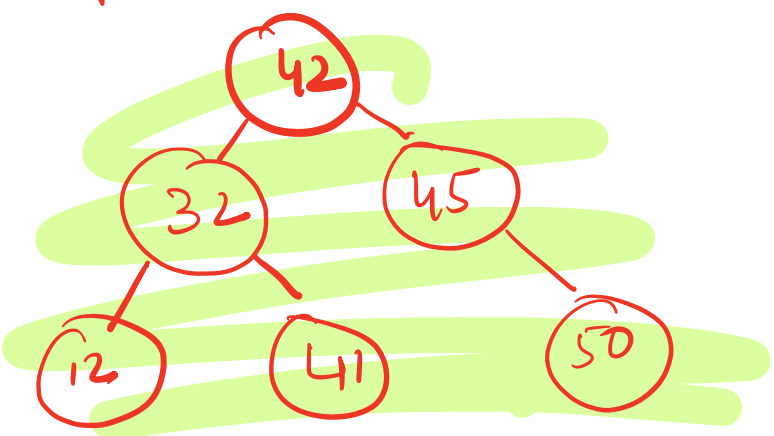
Algorithm Preorder(tree)
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

42	32	12	41	45	50

$T_L(42)$ $T_R(42)$

Result of printing the keys using a preorder traversal

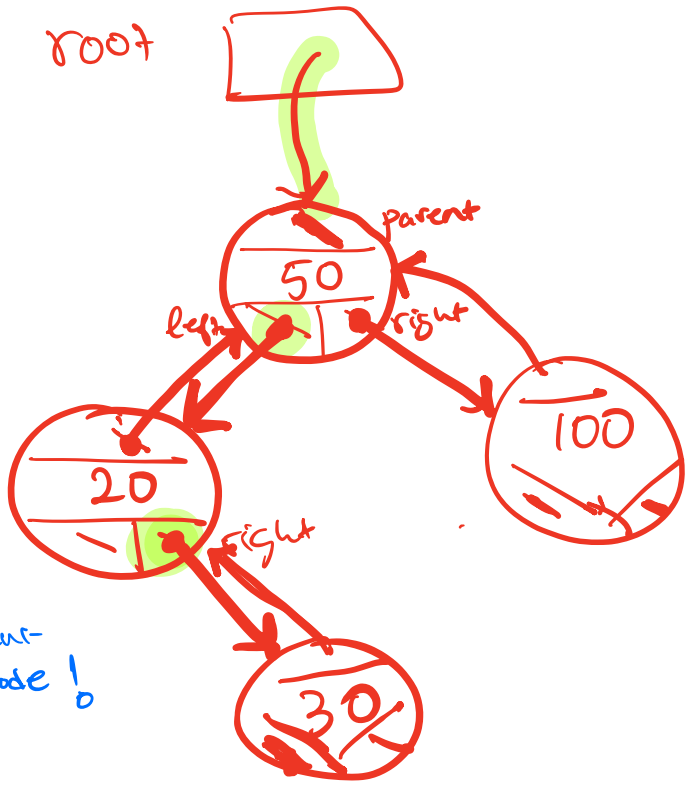
If we were to insert the keys 42, 32, 12, 41, 45, 50 into an initially empty BST, we will create an exact duplicate of the original BST



PreOrder Traversal is useful for implementing the copy constructor of bst class.

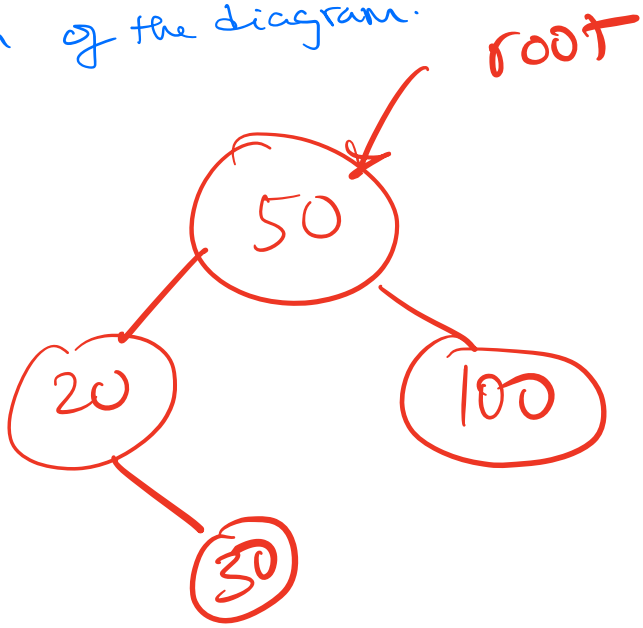
In class demo of code to build a small bst

```
bst b;
b.insert(50);
b.insert(20);
b.insert(30);
b.insert(100);
```



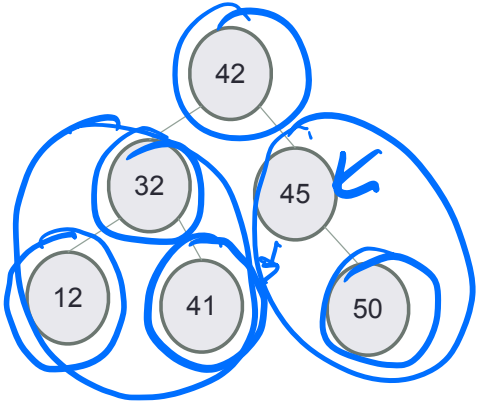
Remember to update parent-pointers for each new node!

Simplified version of the diagram.



Post-order traversal: use in recursive destructors!

deletion of all the nodes

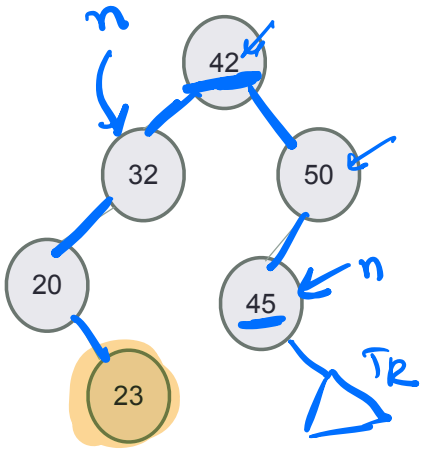


Algorithm Postorder(tree)

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. → cout << n->data

12 41 32 50 45 42

Predecessor: Next smallest element



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

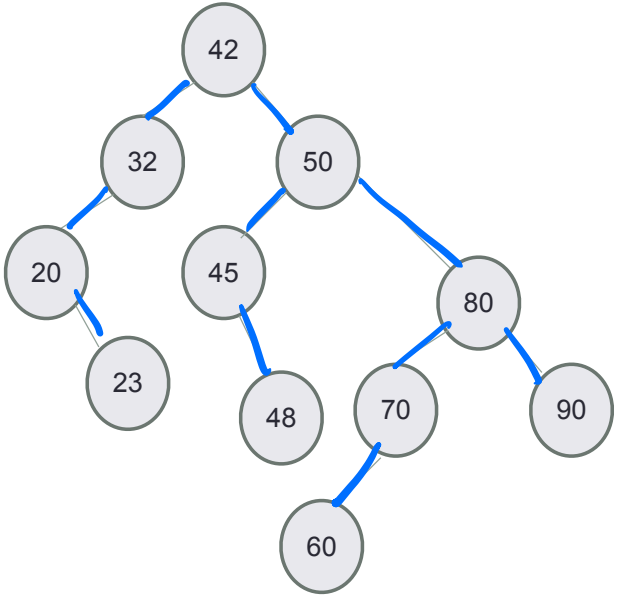
```
int bst::predecessor(bstNode *n){
    if(n && n->left){ return max(n->left);}
}
```

Case 1:

Case 2: traverse parent pointers until you find a node x whose key is less than n->data. return x->data.

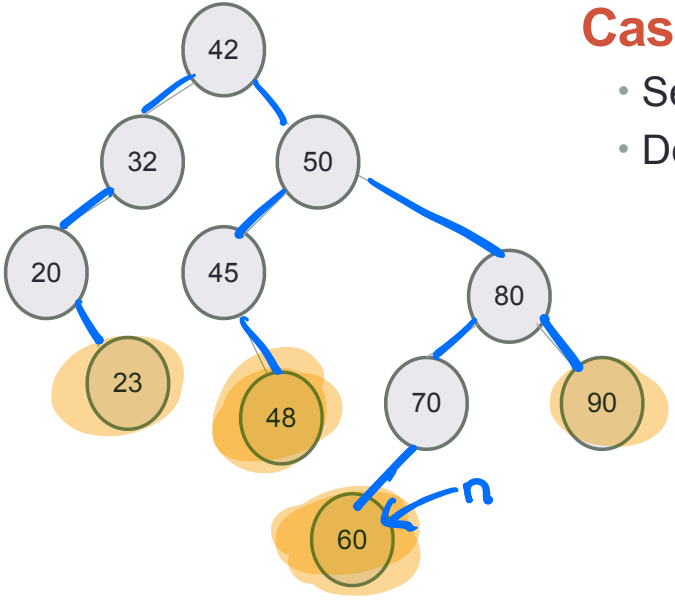
```
Fill in the blank, using min/max helper functions
int bst::min(Node *r); //Returns the min of bst rooted at r
int bst::max(Node *r); //Returns the max of bst rooted at r

A. return n->left;
B. return min(n);
C. return max(n);
D. return min(n->left);
E. return max(n->left);
```

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



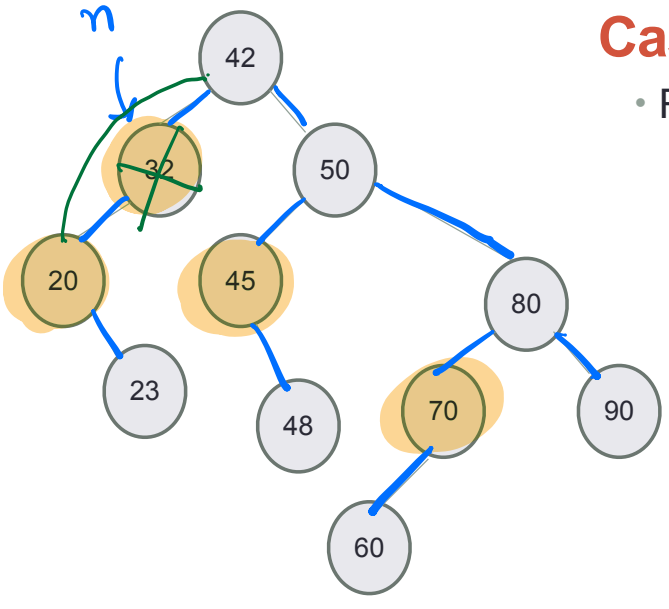
Case 1: Node is a leaf node

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

(1) Set parent's child pointer null.

(2) delete the node

Delete: Case 2

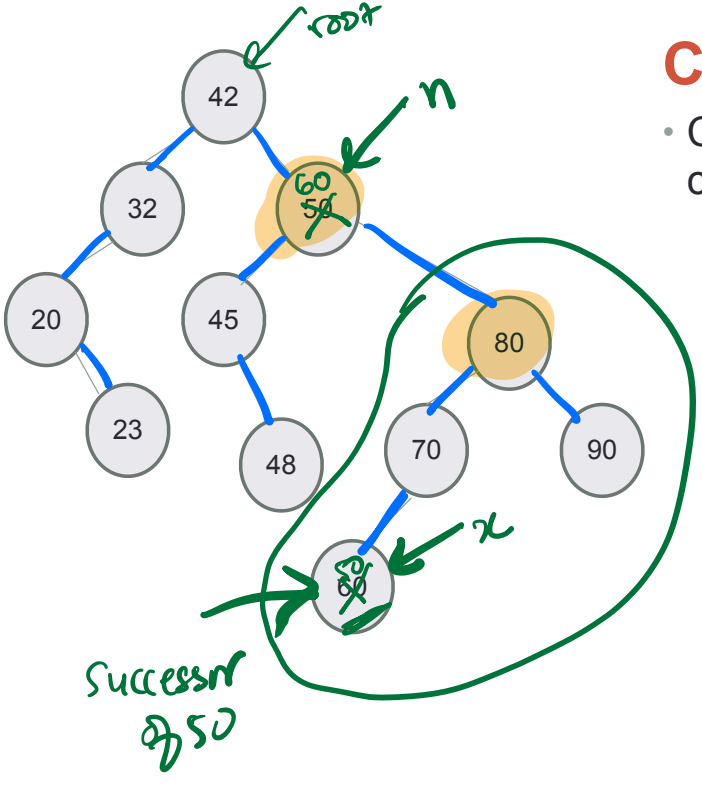


Case 2 Node has only one child

- Replace the node by its only child

b.erase(n)

Delete: Case 3



Case 3 Node has two children

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

(1) swap the value of the node with its successor or predecessor

(2) delete the node at the new location which defaults to case 1 or case 2

Past exam question

Precondition: r points to a bst node that has
- a right subtree but no left subtree
- a non-null parent

Postcondition: the node pointed to by r is correctly deleted

```
void bst::removeNodeWithRightChild(bstNode *r){

     r->parent->right = r->right;
     r->right->parent = r->parent;
     delete r;
}
```

Write a test case that exposes a bug in the above implementation. Assume you have a correct insert function.

void test() {

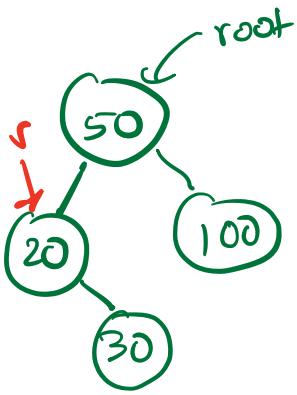
See code from lecture

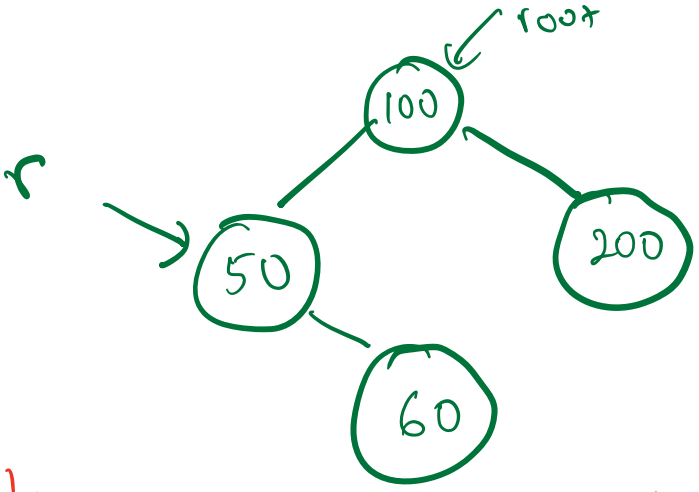
}

A. Done & confident of my answer

B. Done but not sure

C. I have no idea how to proceed





Another example

example test bst