BINARY SEARCH TREES (CONTD)

Problem Solving with Computers-II

A node in a BST

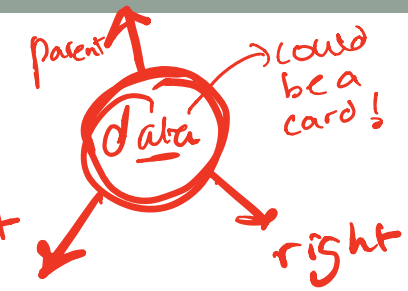
```
class BSTNode {

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

struct Card {
char suit;
int value;
};

type of data

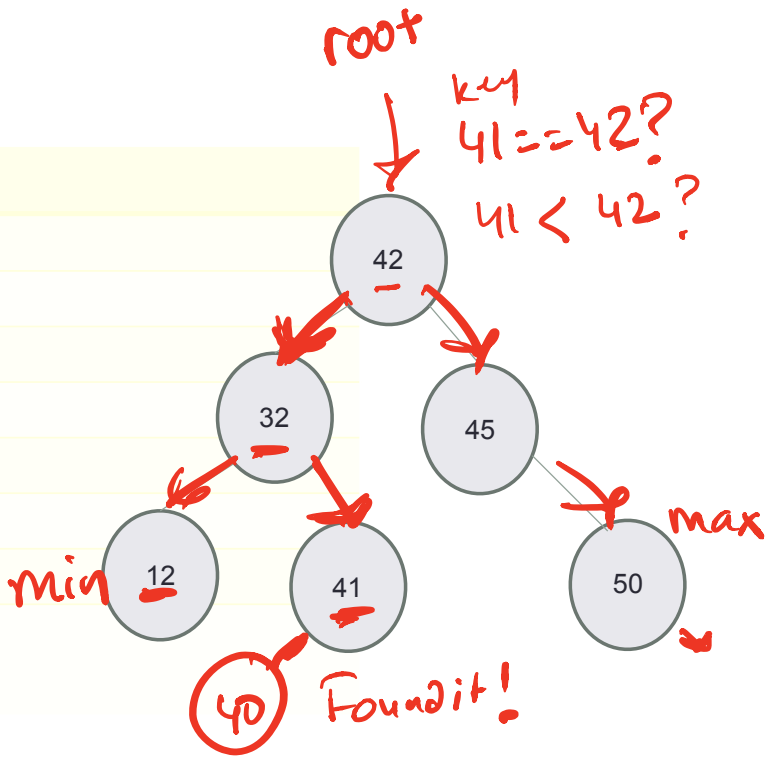
The data stored in any BST node could be of any type e.g. Card. as long as the operations ==, < and > are defined on that type



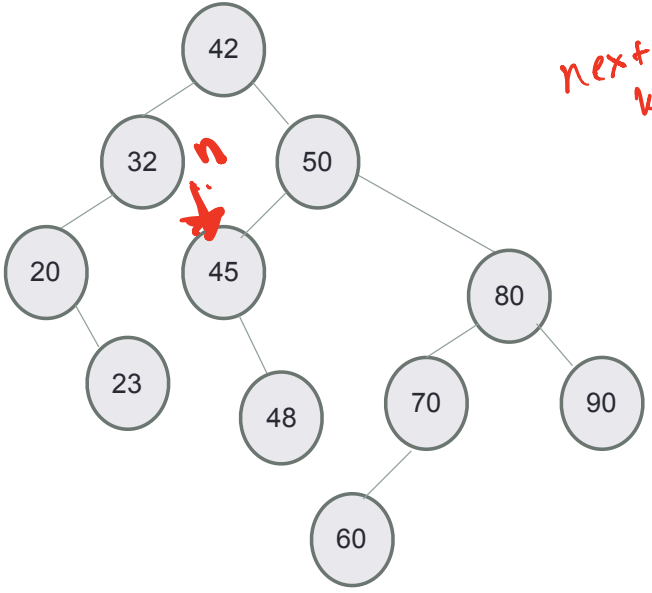
class BST {
private:
BSTNode* root;
};

Define the BST ADT

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



Successor: Next largest element



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

next largest key → int

key

int successor (int value);

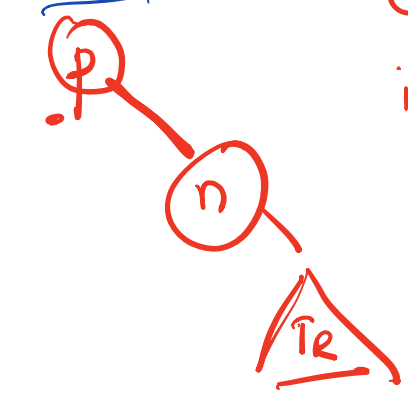
Our algorithm will be for a private version of this function that takes a pointer to a node as input and returns a pointer to the node with the next largest value.

Node* successor-Private (Node* n);

Case1: n has a right subtree T_R

$key(n) < keys(T_R)$ ..... (1)

Case 1a



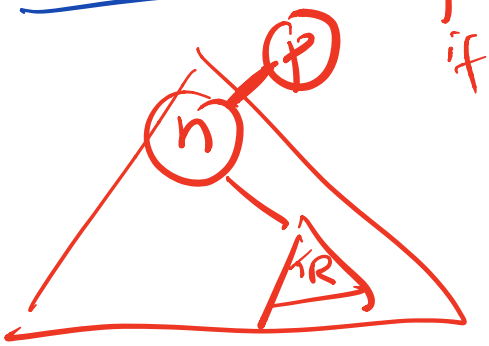
if n is parent's right child:

$key(p) < key(n)$ ..... (2)

From (1) & (2)

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

Case 1b



if n is its parent's left child

if (n→parent→left == n):

$key(n) < key(p)$ — (3)

$key(n) < key(T_R)$ — (4)

How does key(p) compare with key(T_R)

$key(p) < key(T_R)$

$key(p) > key(T_R)$ — (5)

Can't say

from ③ ④ ⑤

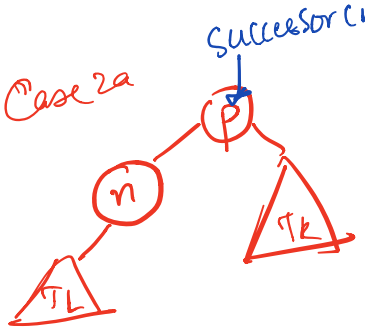
$key(n) < keys(T_R) < key(p)$

↑

Therefore Successor(n) must be in T_R

Case 2: n has no right subtree (this was left as an assignment) but now that you now about the Inorder Traversal, the proof might be easier to follow

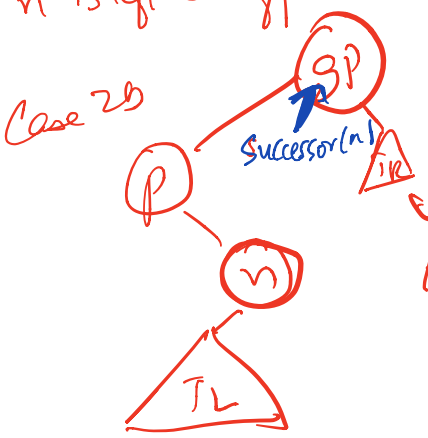
Case 2a



n is left child of parent

If n is the left child of its parent (as shown on the figure to the left), an inorder traversal of the tree would print the parent's key after printing n. Therefore p is the successor of n

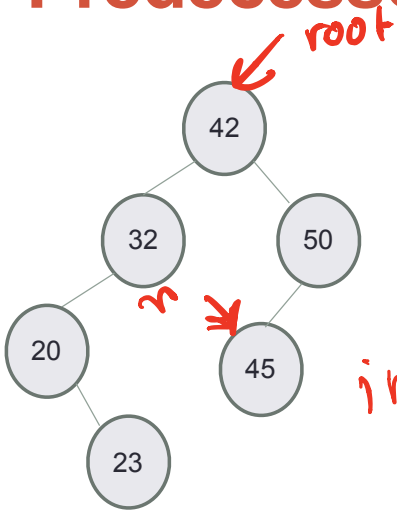
Case 2b



n is right child of parent

If n is the right child of its parent(p) then, the parent's recursive call is done after n's key is printed and we proceed to the grandparent's recursive call OR more generally proceed all the way up the tree until we find a node that is the left child of its parent, and return the parent

Predecessor: Next smallest element



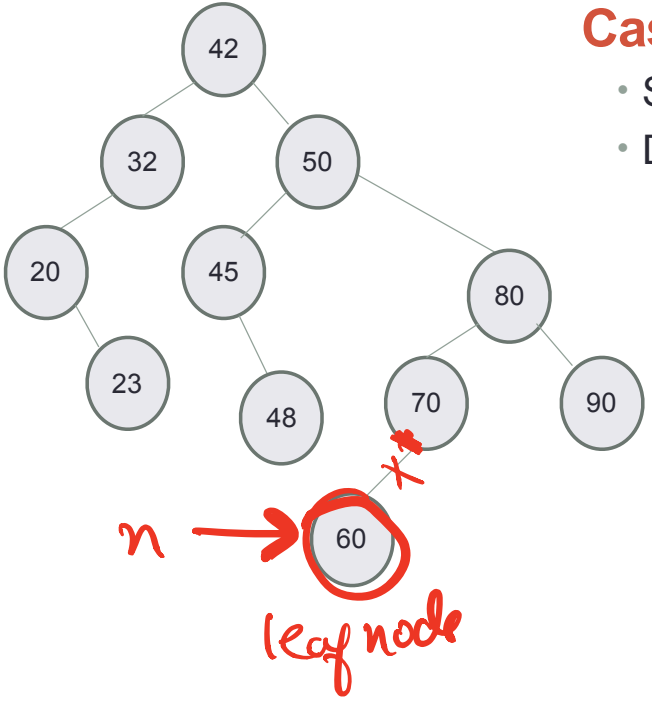
- What is the predecessor of 32?
- What is the predecessor of 45?

int

```
predessor ( int value ) {
Node * n = findH( value );
return predecessorH (n)->data;
}

predecessorH ( Node * n ) {
}
Node*
```

Delete: Case 1

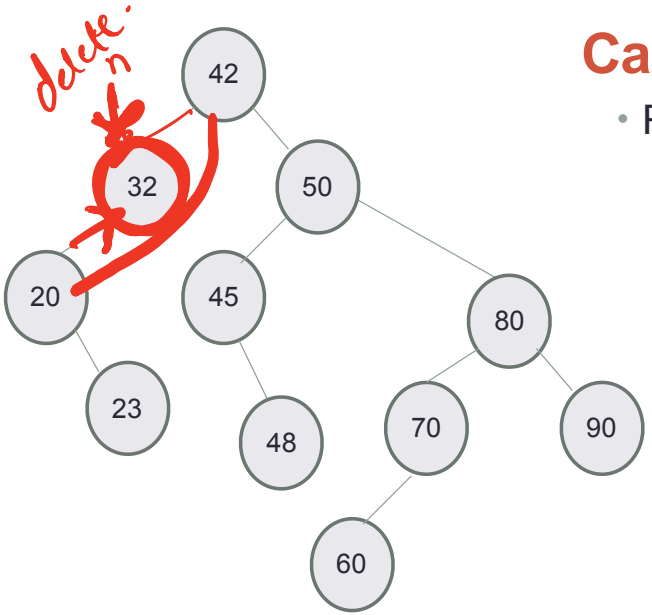


Case 1: Node is a leaf node

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

```
if (n->left && !n->right){
    // leaf node
    if(n == n->parent->left)
        n->parent->left = 0;
    else
        n->parent->right = 0;
    delete n;
```

Delete: Case 2

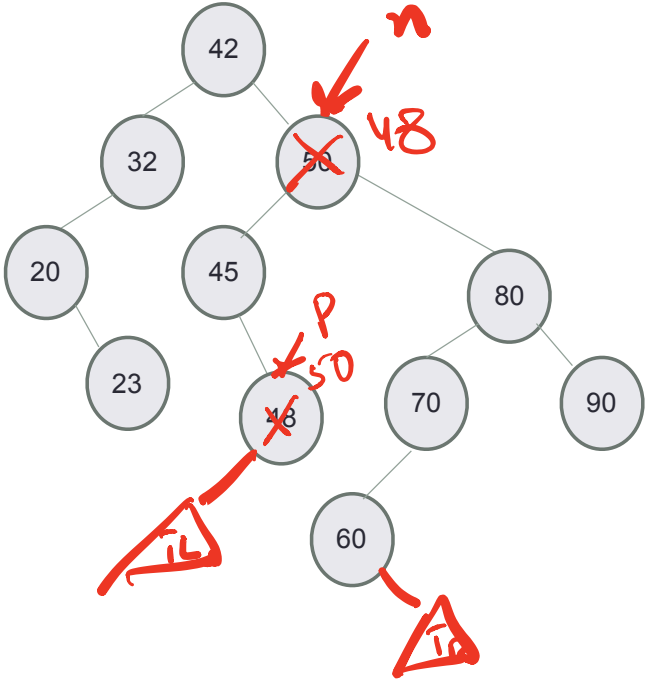


Case 2 Node has only one child

- Replace the node by its only child

```
if ( !n->right || n->parent )
    // connect n's left child to n's parent
    n->left->parent = n->parent;
    if (n == n->parent->left) {
        n->parent->left = n->left;
    else
        n->parent->right = n->left;
} // other
delete n;
```

Delete: Case 3

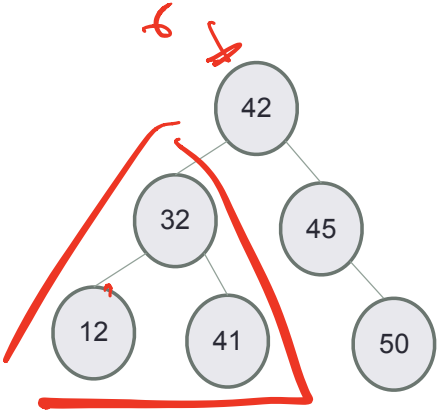


Case 3 Node has two children

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

// Swap n→data with its predecessor OR its successor
// delete the predecessor
// Default to either case 1 or case 2

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)

What is the output of doing an inorde traversal on the above tree

A. 32 12 41 42 45 50

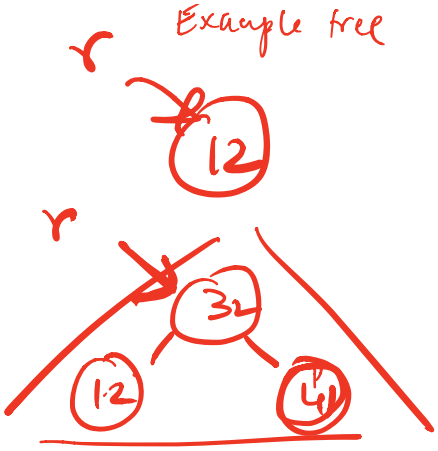
B. 12 32 41 42 45 50

C. 12 32 42 45 41 50

D. None of the above

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

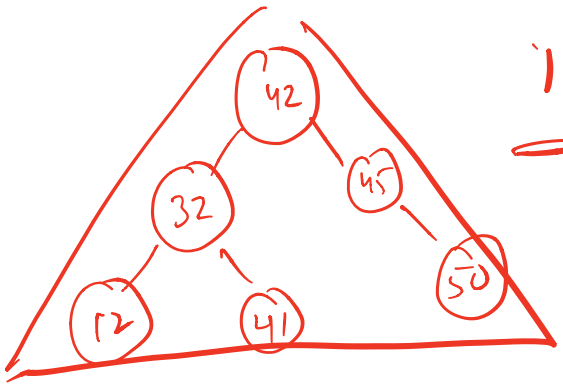
Example tree



Output of In Order Traversal

12

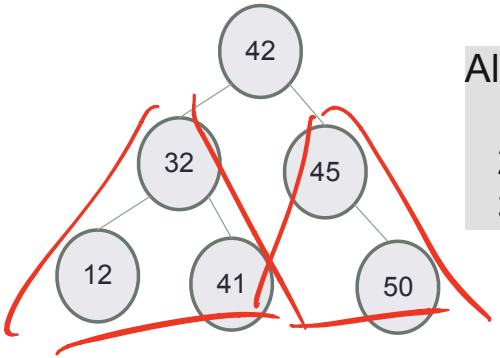
12 32 41



12 32 41 42 45 50

Keys are printed in sorted order

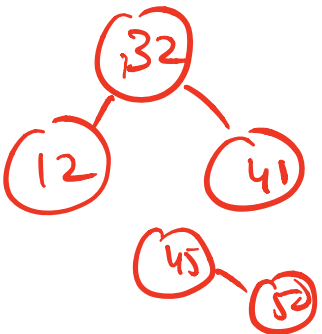
Pre-order traversal: nice way to linearize your tree!



Algorithm Preorder(tree)

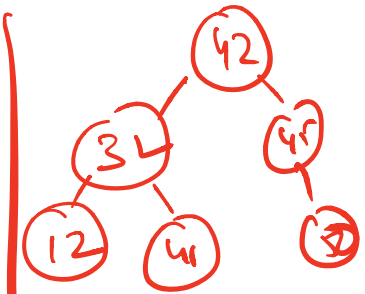
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

42 32 12 41 45 50

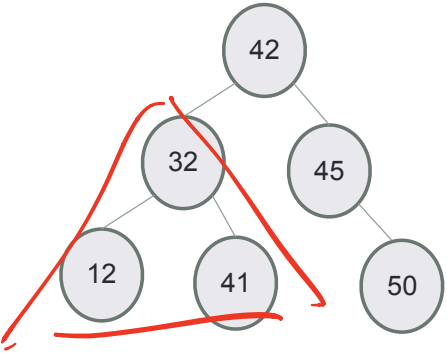


32 12 41

45 50



Post-order traversal: use in recursive destructors!



12 41 32

Algorithm Postorder(tree)

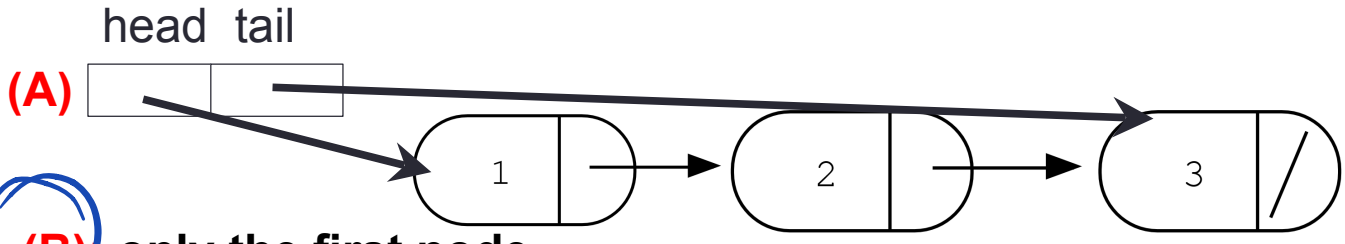
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Concept Question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
class Node {
    public:
       int info;
       Node *next;
};
```

Which of the following objects are deleted when the destructor of Linked-list is called?



(B): only the first node

(C): A and B

(D): All the nodes of the linked list

(E): A and D

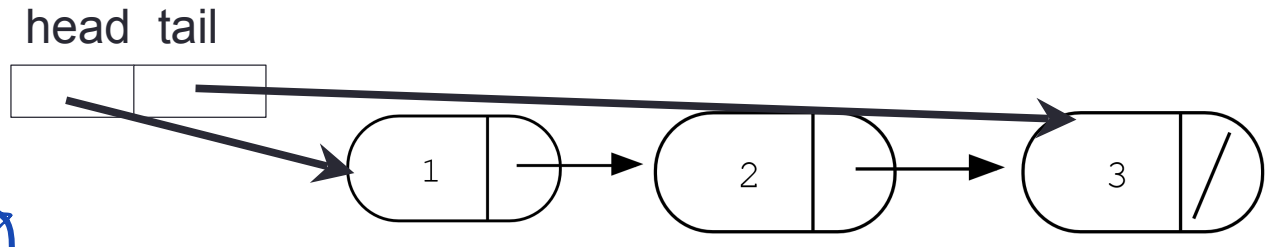
Concept Question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
Node::~Node(){
   delete next;
}
```

Which of the following objects are deleted when the destructor of Linked-list is called?

(A)



(B): All the nodes in the linked-list

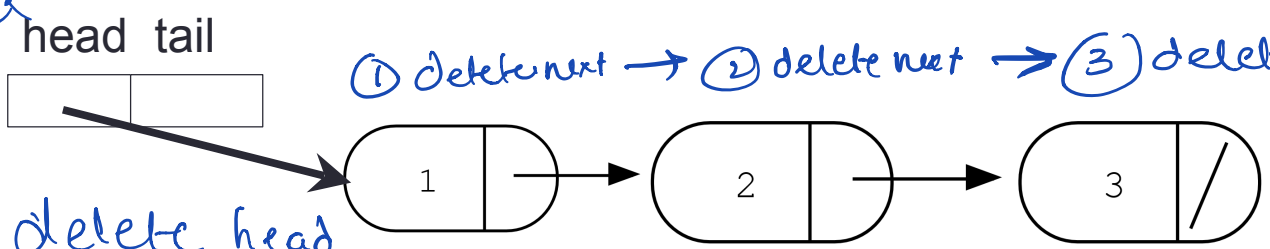
(C): A and B

(D): Program crashes with a segmentation fault

(E): None of the above

```
LinkedList::~LinkedList(){
   delete head; // Calls the first node's destructor
}
```

```
Node::~Node(){
   delete next; // Calls the next node's destructor
}
```



① delete next → ② delete next → ③ delete 0

does not seg fault

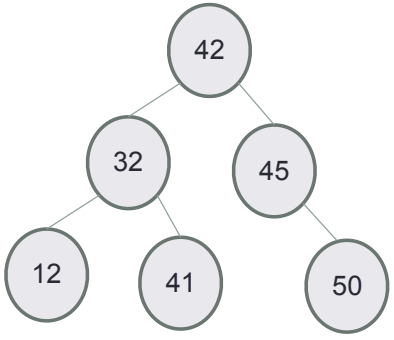
call to destructor complete, Node -3 is delete

call to destructor of 2 completes Node 2 is deleted

call to destructor of node 1 completes, Node 1 is deleted

ll is deleted

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.

~BSTNode() {

delete left;

delete right;

}