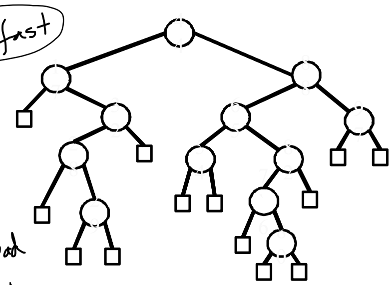
Binary Search Trees

Search of Array
Insert/Delete of Lists

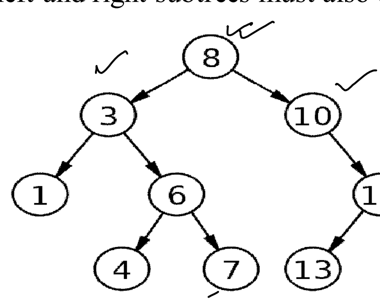
Search
Array: Search is fast
O(log n)
List : O(n) ← Bad

Insert / Delete
Array : O(n) ← bad
List : O(1) ← good



Binary Search Trees

- A **binary search tree** (**BST**) is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
  - Both the left and right subtrees must also be binary search trees.



Binary Search Trees

- Firstly, it is a binary tree
- It is represented by a linked data structure
- It combines
  - the advantage of an array -- the ability to do a binary search with
  - the advantage of a linked list -- its dynamic size
- The efficiency of all of the operations is $O(h) = O(\log n)$, only if the tree is reasonably height-balanced
- Each node contains at least the following fields
  - *key*: an identifying field
  - *left*: pointer to a left child (may be NIL)
  - *right*: pointer to a right child (may be NIL)
  - *p*: pointer to the parent node (NIL for root)

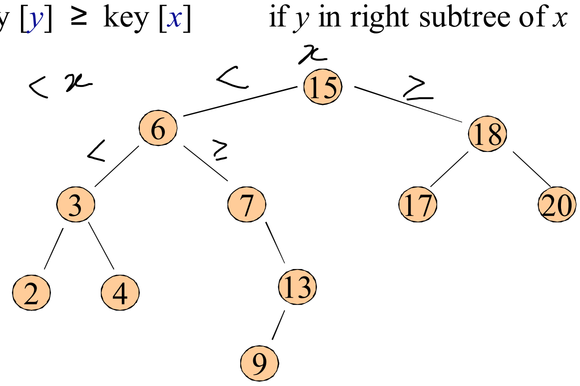
Binary Search Trees

- Secondly, the binary-search-tree property
  - For any node x,

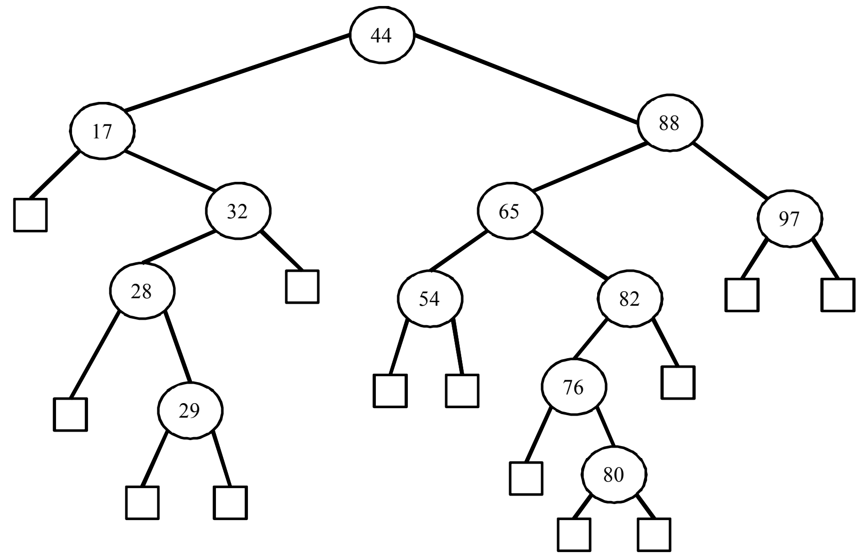
    key [y] $<$ key [x] if y in left subtree of x

    key [y] $\geq$ key [x] if y in right subtree of x

$y < x$

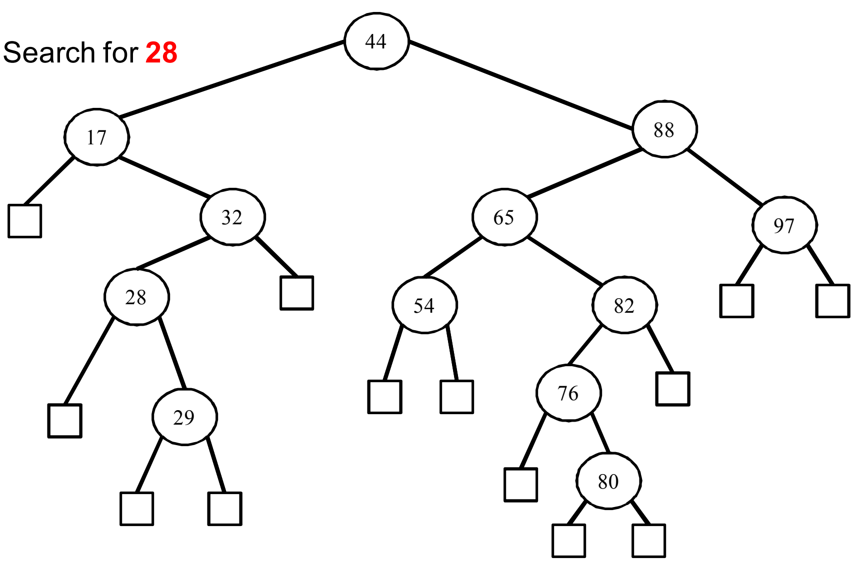


Binary Search Trees: Example

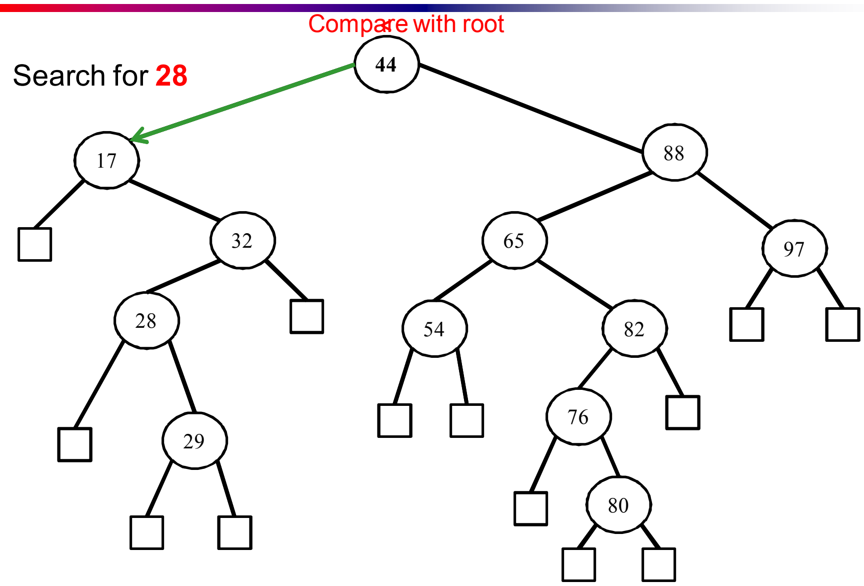


Operations on BSTs: Search

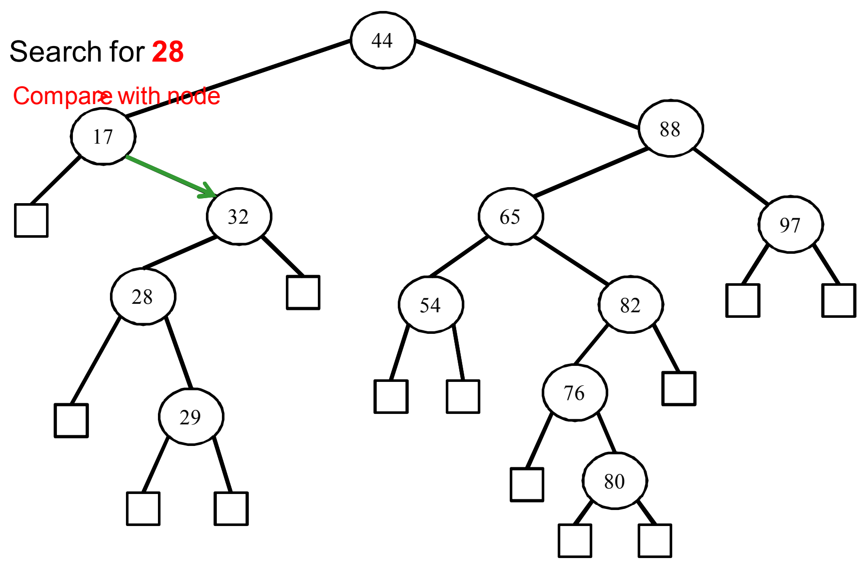
Search for **28**



Operations on BSTs: Search

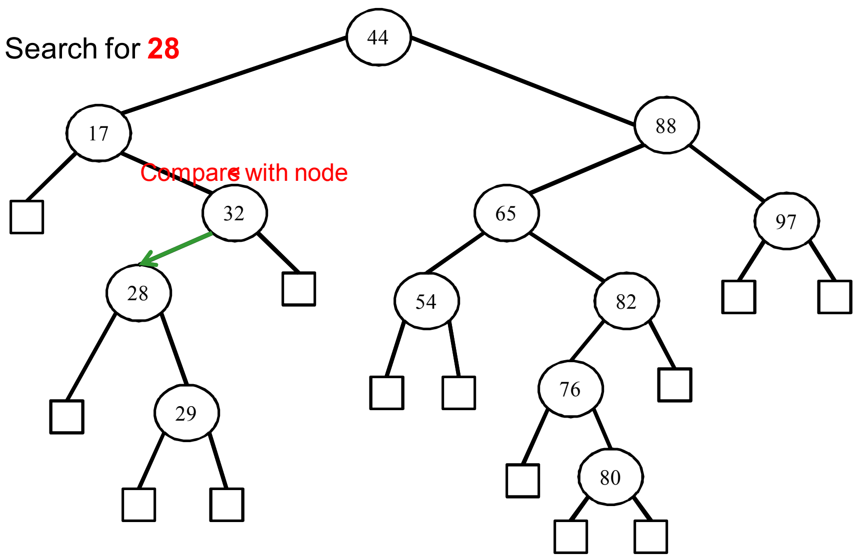


Operations on BSTs: Search



Operations on BSTs: Search

Search for **28**

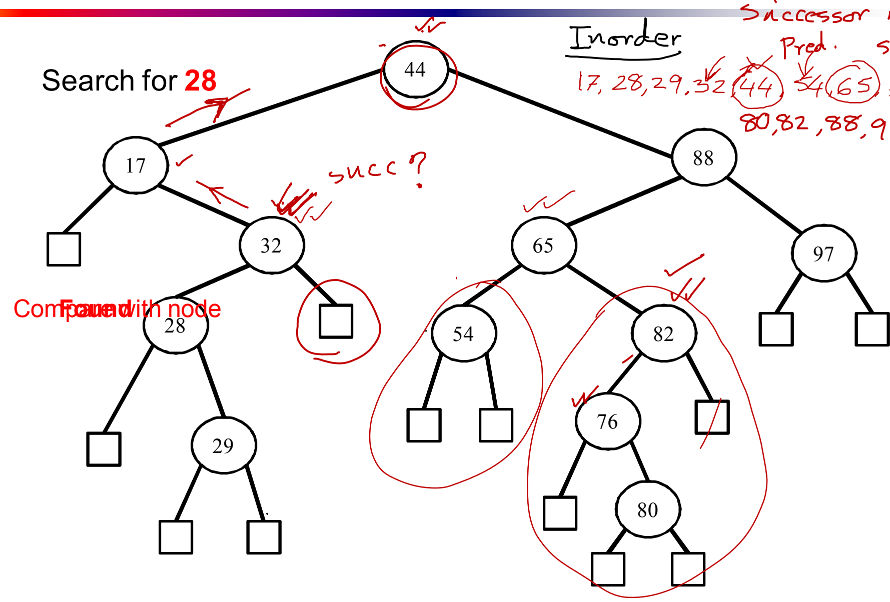


Operations on BSTs: Search

predecessor
Successor node

Inorder
17, 28, 29, 32, 44, 54, 65, 76,
80, 82, 88, 97
Pred. Succ

Search for **28**



Operations on BSTs: Search

- Given a key *k* and a pointer to the root, Tree-Search returns a pointer to a node with key *k* or NIL:

```
Tree-Search(x, k)
    if (x = = NIL  or  k = = key[x])
        return x;
    if (k < key[x])
        return Tree-Search(left[x], k);
    else
        return Tree-Search(right[x], k);
```


Operations on BSTs: Search

- Here is another function that does the same:

```
Tree-Search(x, k)
    while (x != NIL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

- *Which of these two functions is more efficient?*

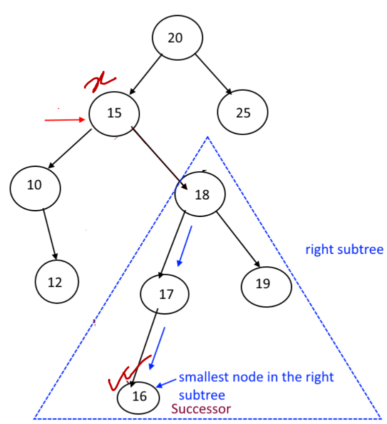
Operations on BSTs: Min & Max

```
TREE-MINIMUM(x)
    while left[x] ≠ NIL
          do x ← left[x]
    return x
```

```
TREE-MAXIMUM(x)
    while right[x] ≠ NIL
          do x ← right[x]
    return x
```

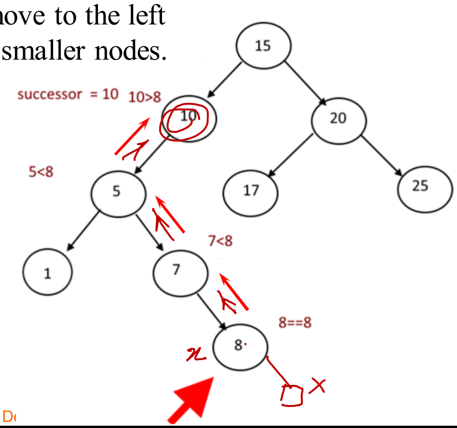

Operations on BSTs: Successor

- Two cases:
  - *Case* 1: *x* has a right subtree:
    - successor is minimum node in right subtree



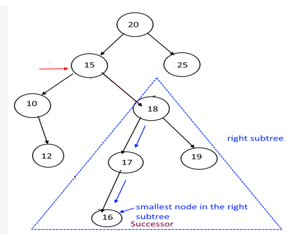
Operations on BSTs: Successor

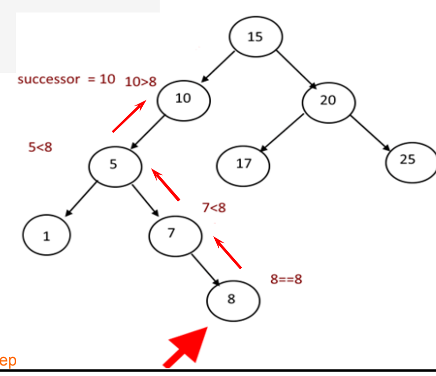
- Two cases:
  - *Case* 2: *x* has no right subtree:
    - successor is first ancestor of *x* whose left child is also ancestor of *x*
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.



Operations on BSTs: Successor

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```





Operations on BSTs: Predecessor

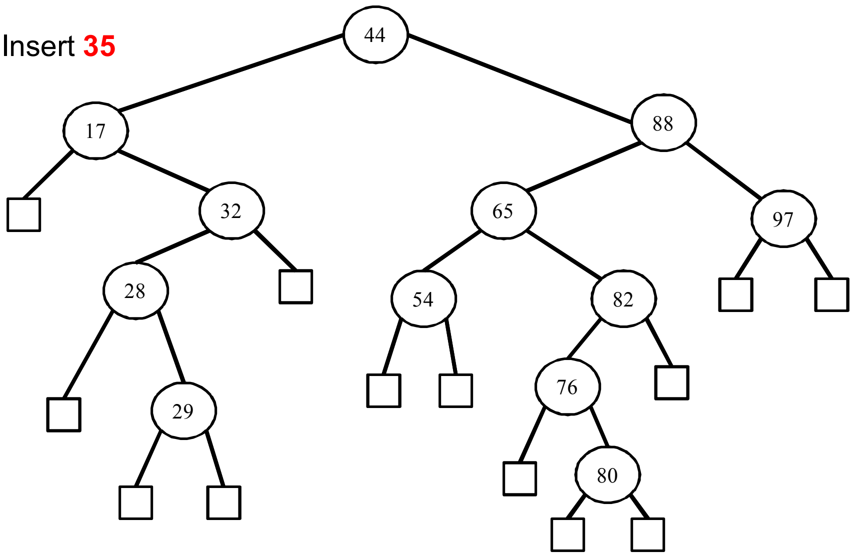
- Two cases:
  - similar to Successor algorithm

Operations of BSTs: Insert

- Inserts an element z to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure as discussed before
  - Insert z in place of NIL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)

Operations of BSTs: Insert

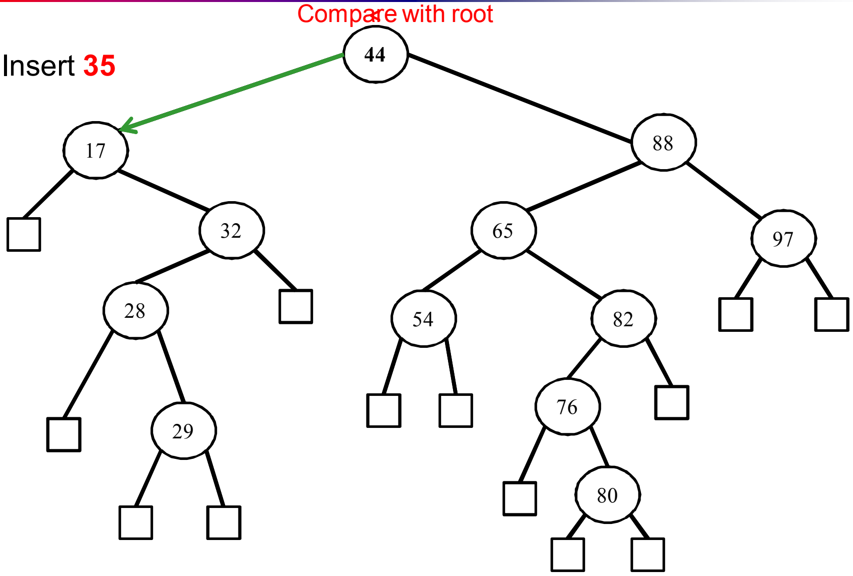
Insert **35**



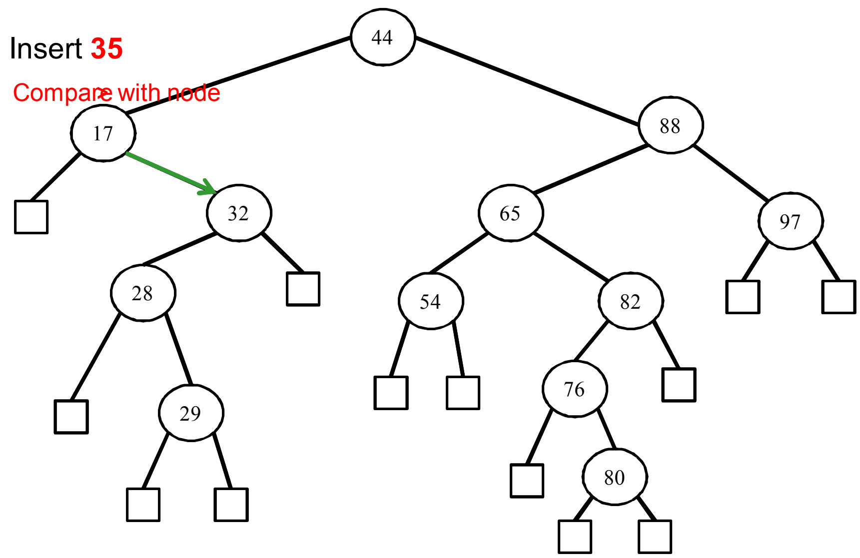
Operations of BSTs: Insert

Compare with root

Insert **35**

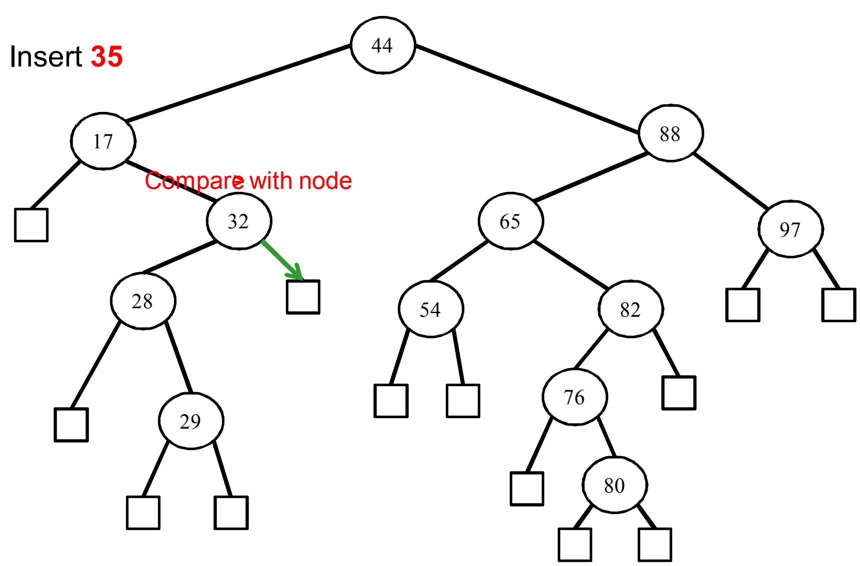


Operations of BSTs: Insert



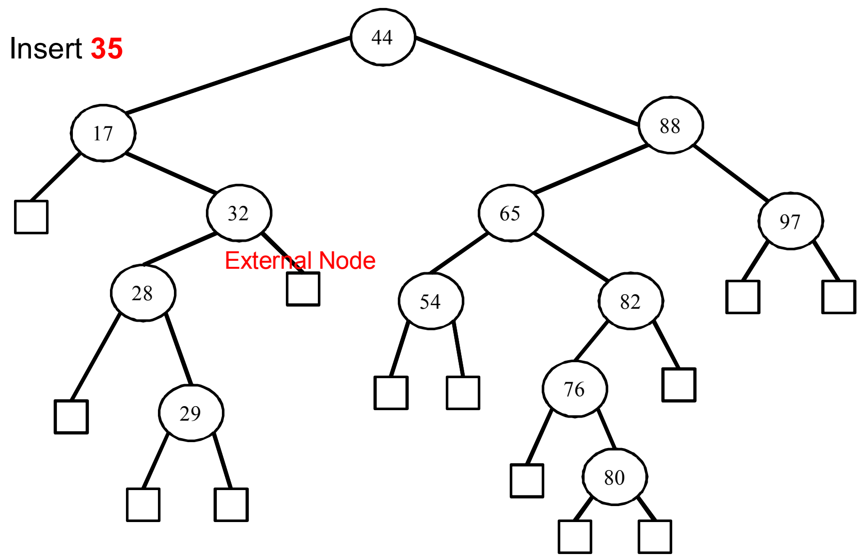
Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 21

Operations of BSTs: Insert



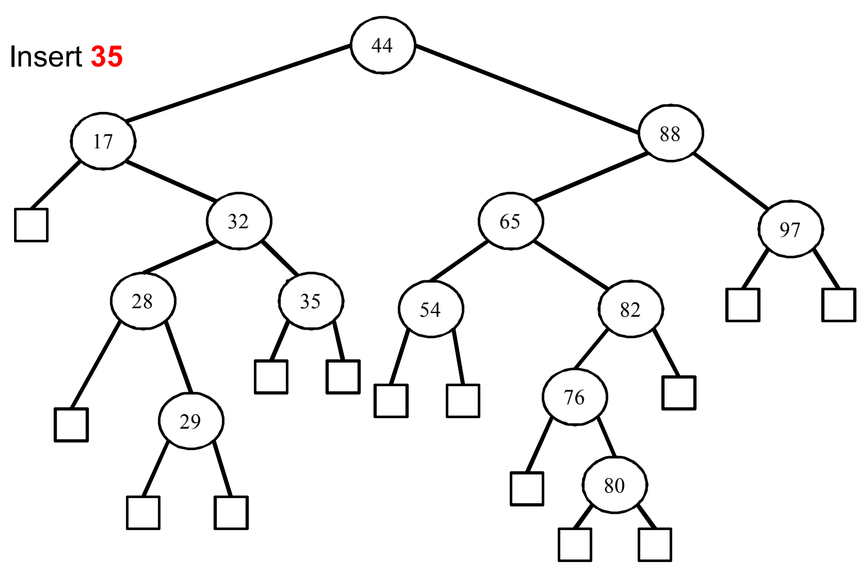
Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 22

Operations of BSTs: Insert



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 23

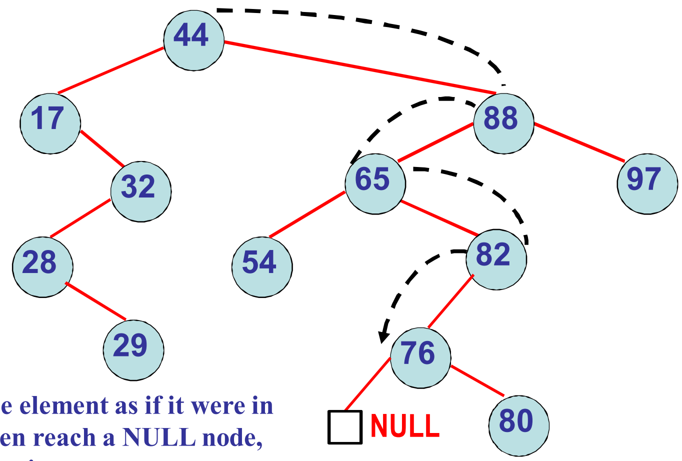
Operations of BSTs: Insert



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 24

Operations of BSTs: Insert

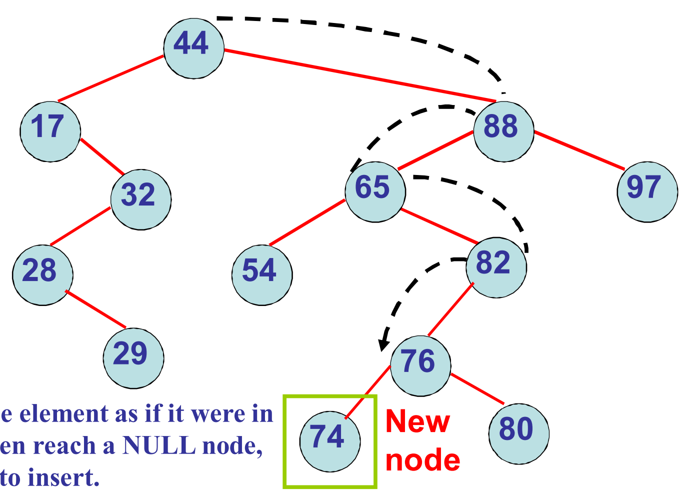
- Example: Insert 74



Search for the element as if it were in the tree. When reach a NULL node, that's where to insert.

Operations of BSTs: Insert

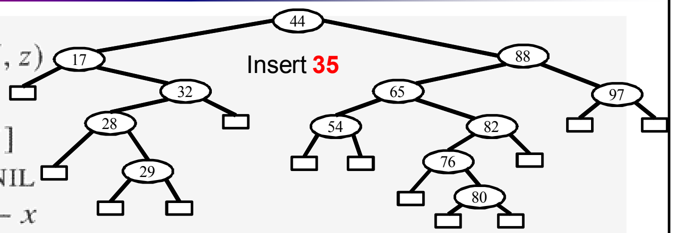
- Example: Insert 74



Search for the element as if it were in the tree. When reach a NULL node, that's where to insert.

Operations of BSTs: Insert

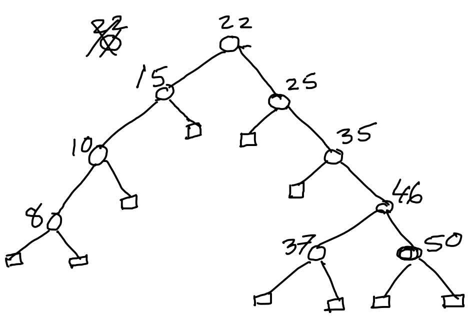
```
TREE-INSERT(T, z)
  y ← NIL
  x ← root[T]
  while x ≠ NIL
      do y ← x
          if key[z] < key[x]
             then x ← left[x]
             else x ← right[x]
  p[z] ← y
  if y = NIL
     then root[T] ← z                    ▷ Tree T was empty
     else if key[z] < key[y]
             then left[y] ← z
             else right[y] ← z
```



Operations of BSTs: Insert

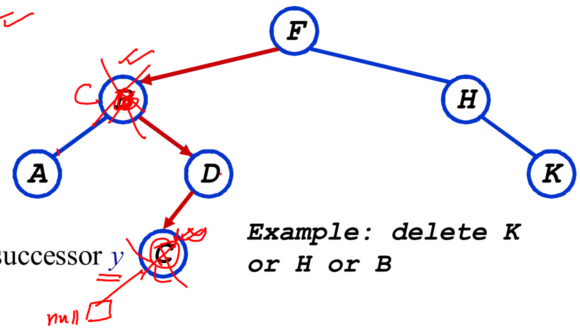
Draw a binary search tree (show each step) after inserting the following keys.

22, 15, 25, 10, 35, 46, 37, 8, 50



Operations of BSTs: Delete

- Delete node x
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Find its inorder successor y
    - Replace x with y
    - Delete y

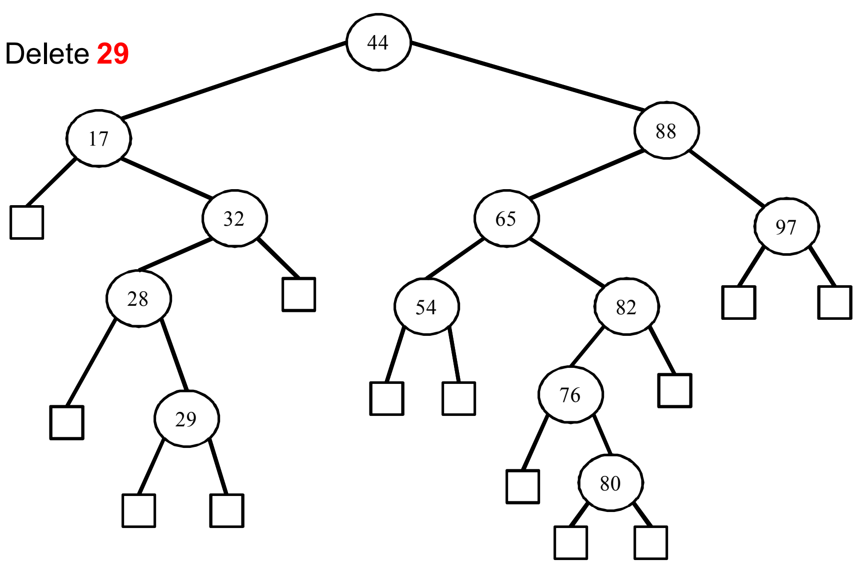


Example: delete K or H or B

[redacted]

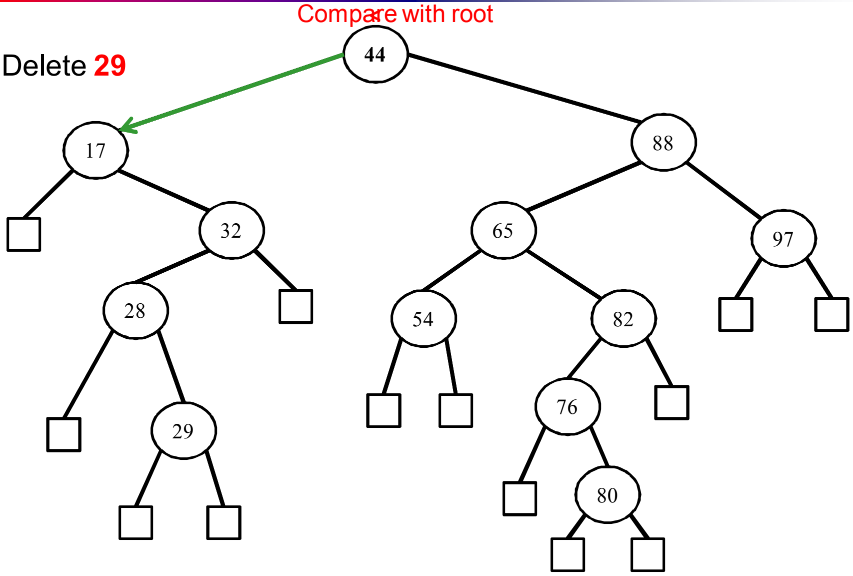
Operations of BSTs: Delete

Delete **29**



Operations of BSTs: Delete

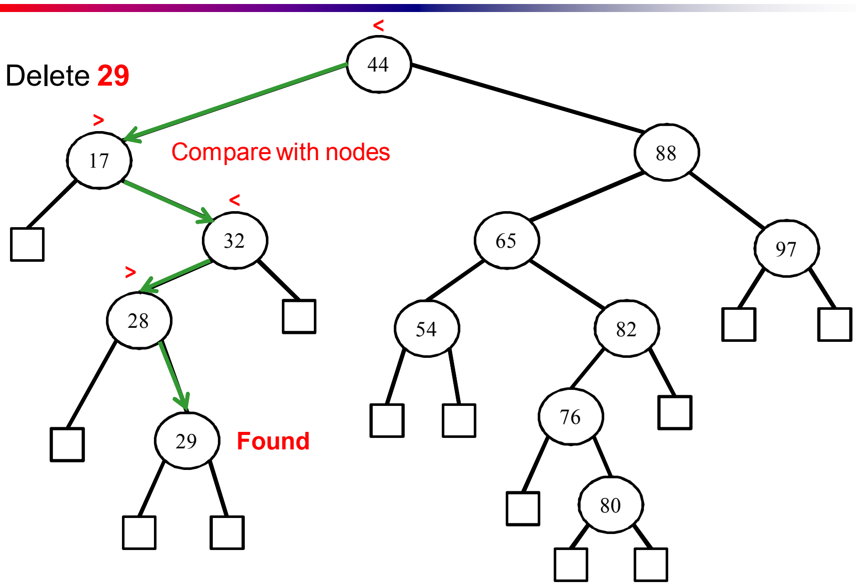
Delete **29**



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 31

Operations of BSTs: Delete

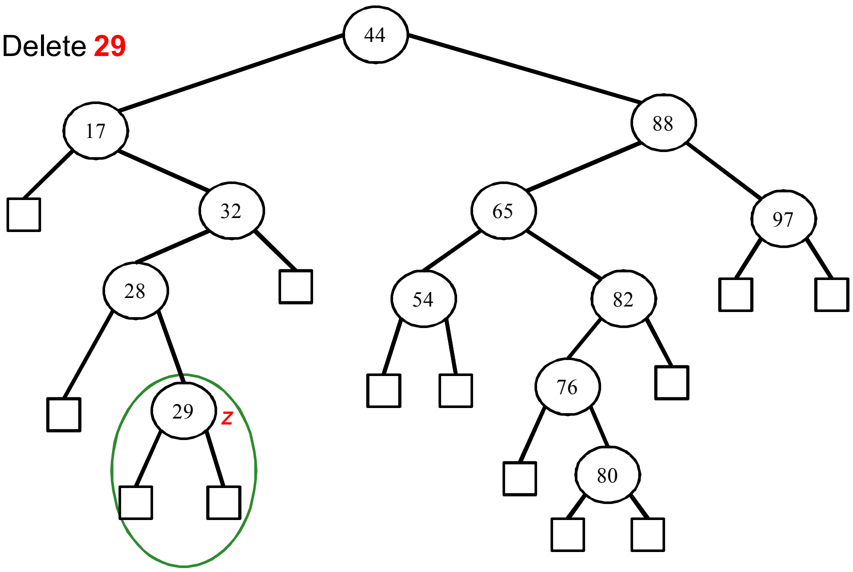
Delete **29**



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 32

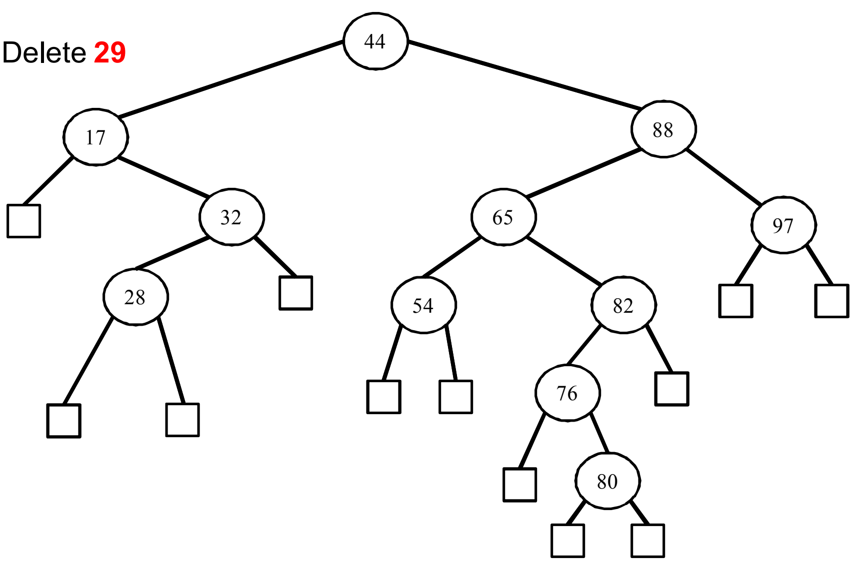
Operations of BSTs: Delete

Delete **29**



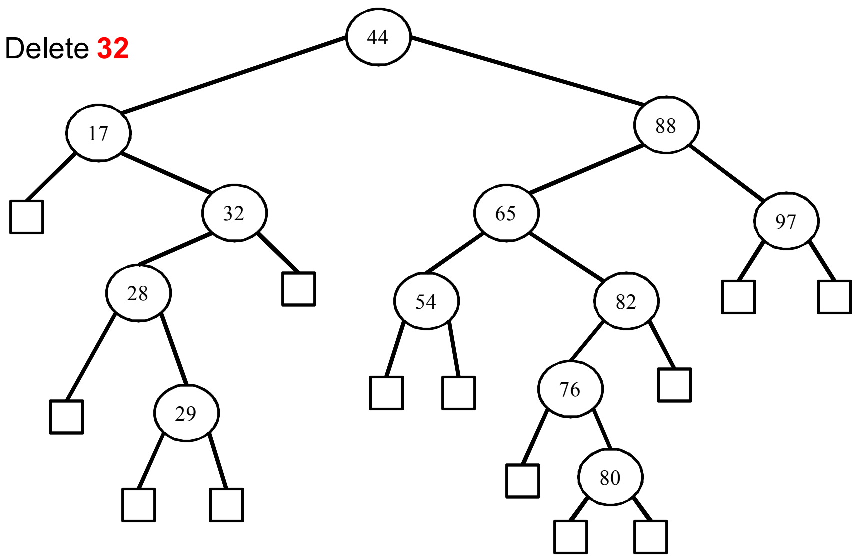
Operations of BSTs: Delete

Delete **29**



Operations of BSTs: Delete

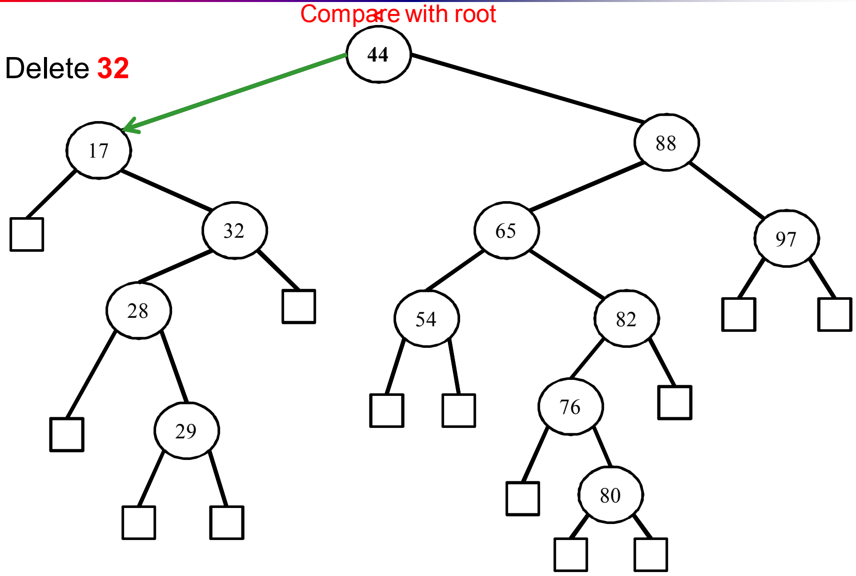
Delete **32**



Operations of BSTs: Delete

Compare with root

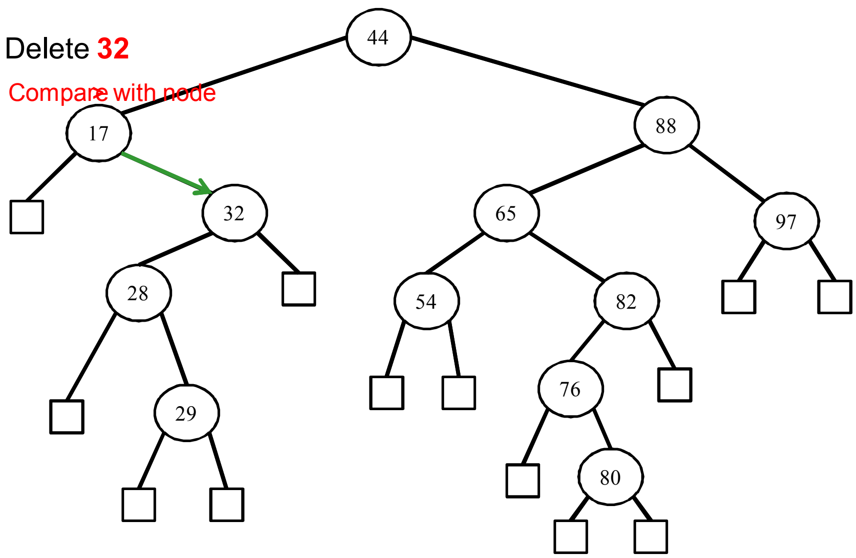
Delete **32**



Operations of BSTs: Delete

Delete **32**

Compare with node

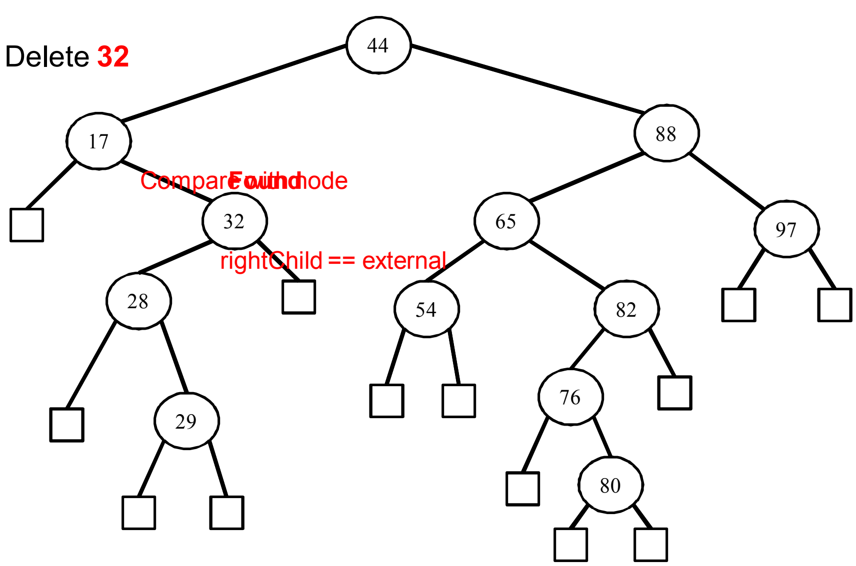


Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

37

Operations of BSTs: Delete

Delete **32**

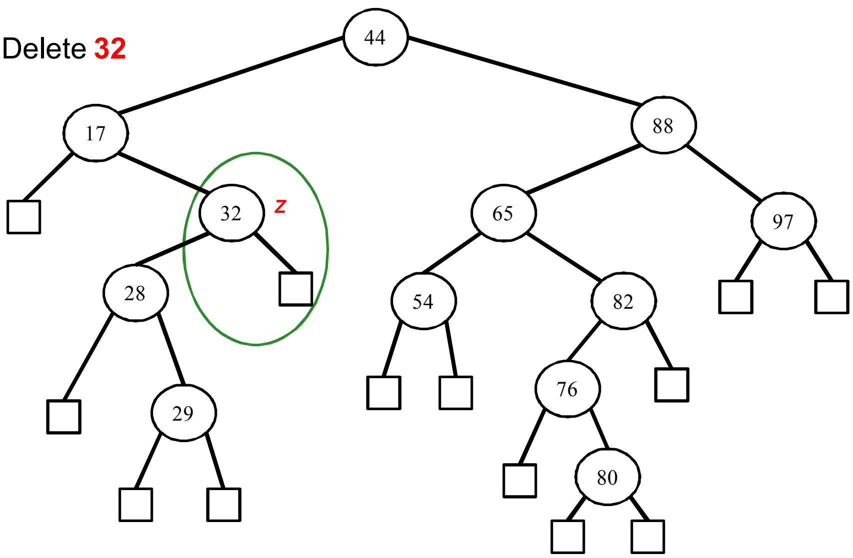


Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

38

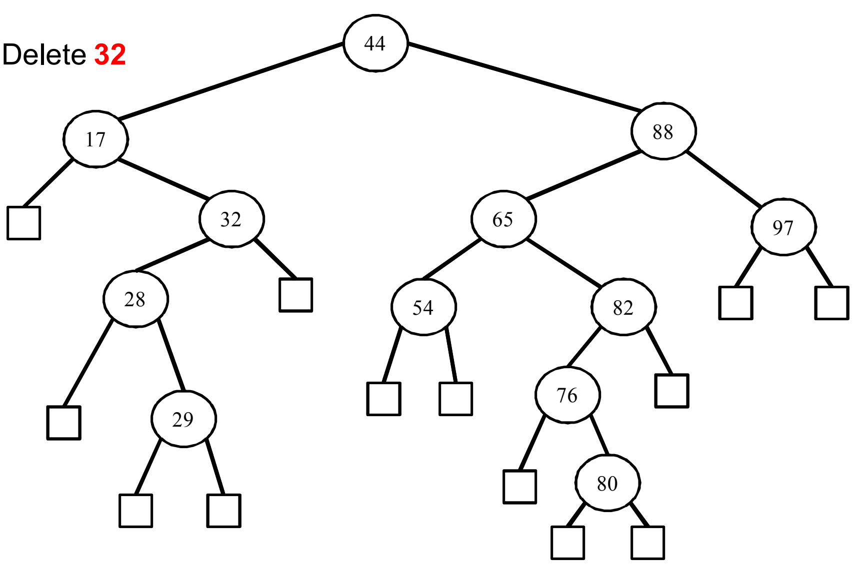
Operations of BSTs: Delete

Delete **32**



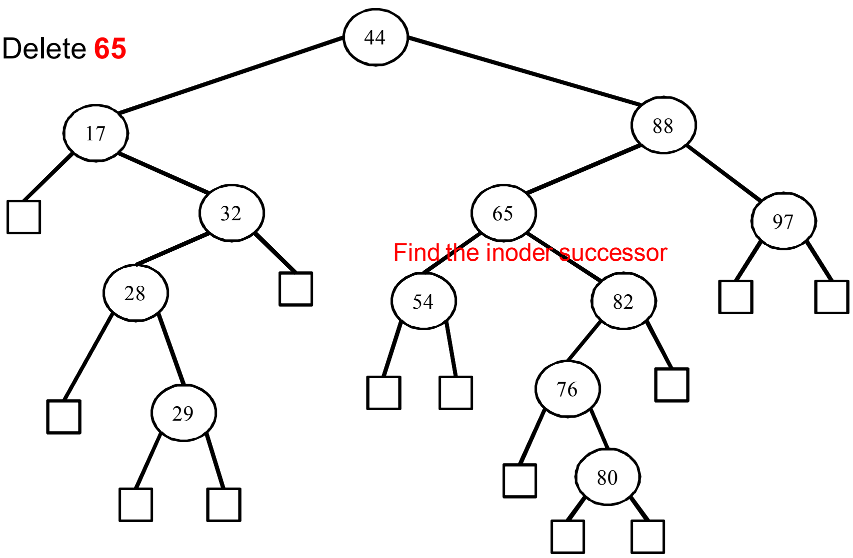
Operations of BSTs: Delete

Delete **32**



Operations of BSTs: Delete

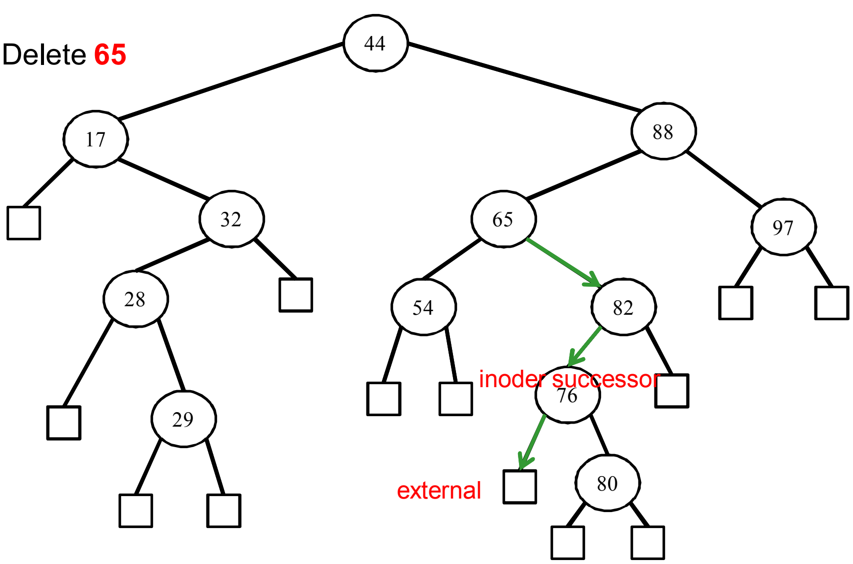
Delete **65**



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 41

Operations of BSTs: Delete

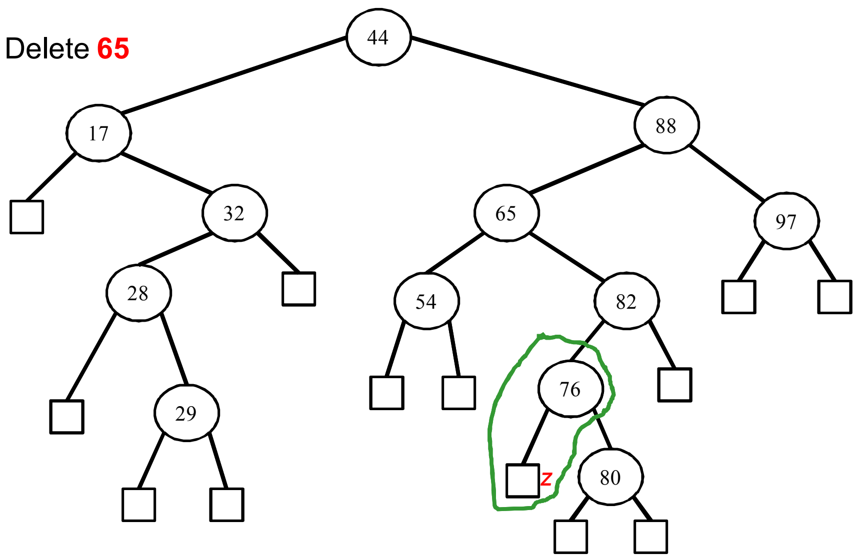
Delete **65**



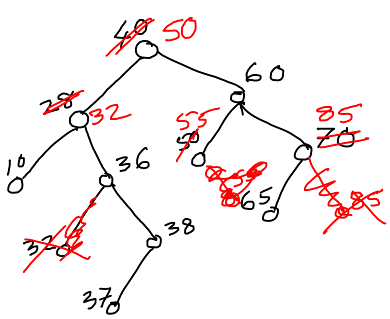
Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 42

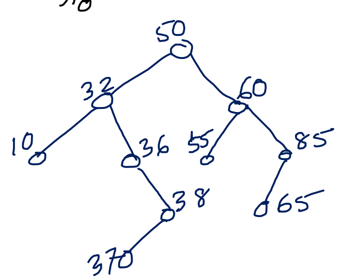
Operations of BSTs: Delete

Delete **65**



Operations of BSTs: Delete





~~Writ~~ Draw the BST after ~~doing the~~ each step of following operations

a) insert 85

b) delete 28

c) delete 70

d) insert 55

e) delete 40

Operations of BSTs: Delete

- Given two nodes *u* and *v*, Transplant replaces the subtree rooted at *u* with the subtree rooted at *v*: node *u*'s parent becomes node *v*'s parent.

```
Transplant(T, u, v)
    if (p[u] = = NIL)
        root[T] = v
    elseif (u = = left[ p[u] ]
        left[ p[u] ] = v;
    else right[ p[u] ] = v;
    if v != NIL
        p[v] = p[u];
```


Operations of BSTs:

- Deletes node *z* from binary search tree *T*

```
Tree-Delete(T, z)
    if (left[z] = = NIL)
        Transplant(T, z, right[z])
    elseif (right[z] = = NIL)
        Transplant(T, z, left[z])
    else y = Tree-Minimum(right[z])
        if p[y] != z
            Transplant(T, y, right[y])
            right[y] = right[z]
            p[ right[y] ] = y
        Transplant(T, z, y)     // p[y] = = z
        left[y] = left[z]
        p[ left[y] ] = y
```

Sorting with BSTs

- Informal code for sorting array A of length n:

```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

- *Running time is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case?*

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 47

BST: Remarks

- All complexity depends on height h
- $h = \Omega(\lg n)$
- To guarantee performance:
  - Balanced tree !
- Randomly build tree
  - **Theorem**: The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU 48

Balanced Binary Search Trees

- Keep $O(\lg n)$ height under dynamic operations
- General framework:
  - First a binary search tree
  - Maintain properties that ensure height guarantee
- AVL tree, red-black tree, …

Codes for Binary Search Trees

```
#include<stdio.h>
#include<stdlib.h>

typedef struct BSTnode{
    int key;
    struct BSTnode *left;
    struct BSTnode *right;
} BSTnode;

BSTnode *create( ){
    BSTnode *temp;
    printf("\nEnter key: ");
    temp = (BSTnode*)malloc(sizeof(BSTnode));
    scanf("%d", &temp->key);
    temp->left = temp->right = NULL;
    return temp;
}
```

Codes for Binary Search Trees

```
void insert(BSTnode *root, BSTnode *temp){
    if(temp->key < root->key){
        if(root->left != NULL)
            insert(root->left, temp);
        else
            root->left = temp;
    }
    if(temp->key > root->key){
        if(root->right != NULL)
            insert(root->right, temp);
        else
            root->right = temp;
} }

BSTnode *minValueNode(BSTnode *node){
    BSTnode *current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```


Codes for Binary Search Trees

```
BSTnode* deleteNode(BSTnode *root, int key){
    if (root == NULL) return root;
    // If the key to be deleted is smaller than the root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    // if key is same as root's key, then This is the node to be deleted
    else {          // node with only one child or no child
        if (root->left == NULL) {
            BSTnode *temp = root->right;
            free(root);
            return temp;   }
        else if (root->right == NULL) {
            BSTnode *temp = root->left;
            free(root);
            return temp;   }
```

Codes for Binary Search Trees

```
        // node with two children: Get the inorder successor (smallest in the right subtree)
        BSTnode *temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

void inorder(BSTnode *root){
    if(root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
```


Codes for Binary Search Trees

```
int main() {
    char ch;    int item;    BSTnode *root = NULL, *temp;
    do {
        temp = create();
        if(root == NULL)
            root = temp;
        else
            insert(root, temp);
        printf("\n Do you want to enter more(y/n)? ");    getchar();    scanf("%c", &ch);
    } while(ch=='y'|ch=='Y');
    printf("\n Inorder traversal of the tree:\t");    inorder(root);

    do {
        if(root == NULL){
            printf("\n No key to delete!");  return 0;   }
        else {
            printf("\nDelete what? ");    scanf("%d", &item);
            root = deleteNode(root, item);
            printf("\nInorder traversal of the modified tree \t");  inorder(root);
        }
        printf("\nDo you want to delete more(y/n)? ");    getchar();    scanf("%c", &ch);
    } while(ch=='y'|ch=='Y');
return 0;   }
```