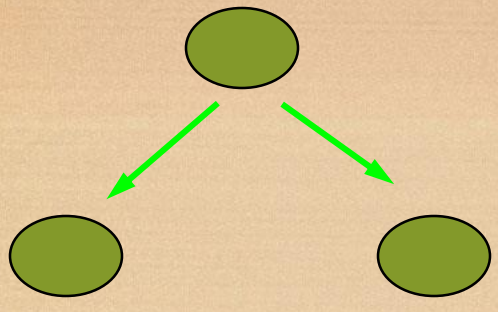
Trees(2)

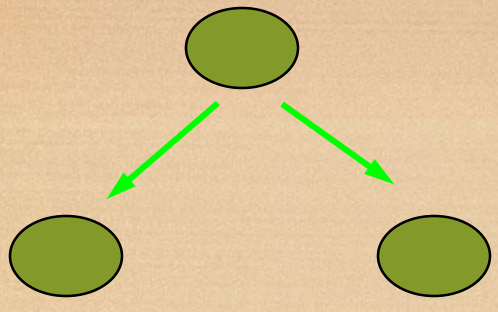
Implementation of Binary Trees

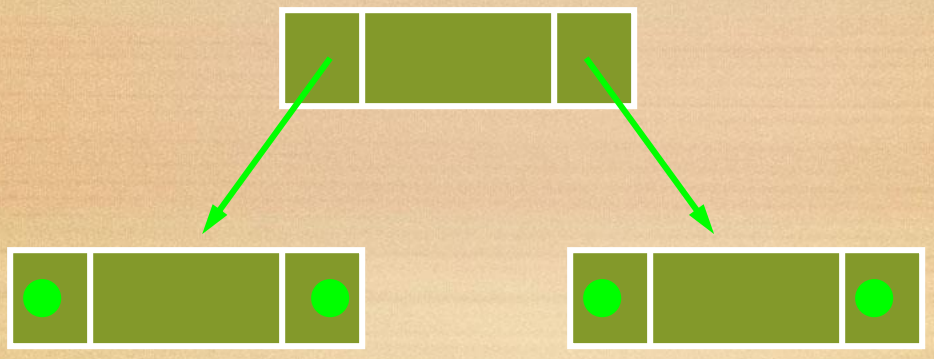
الدكتور
اثير العاني

Implementation using Linked Lists

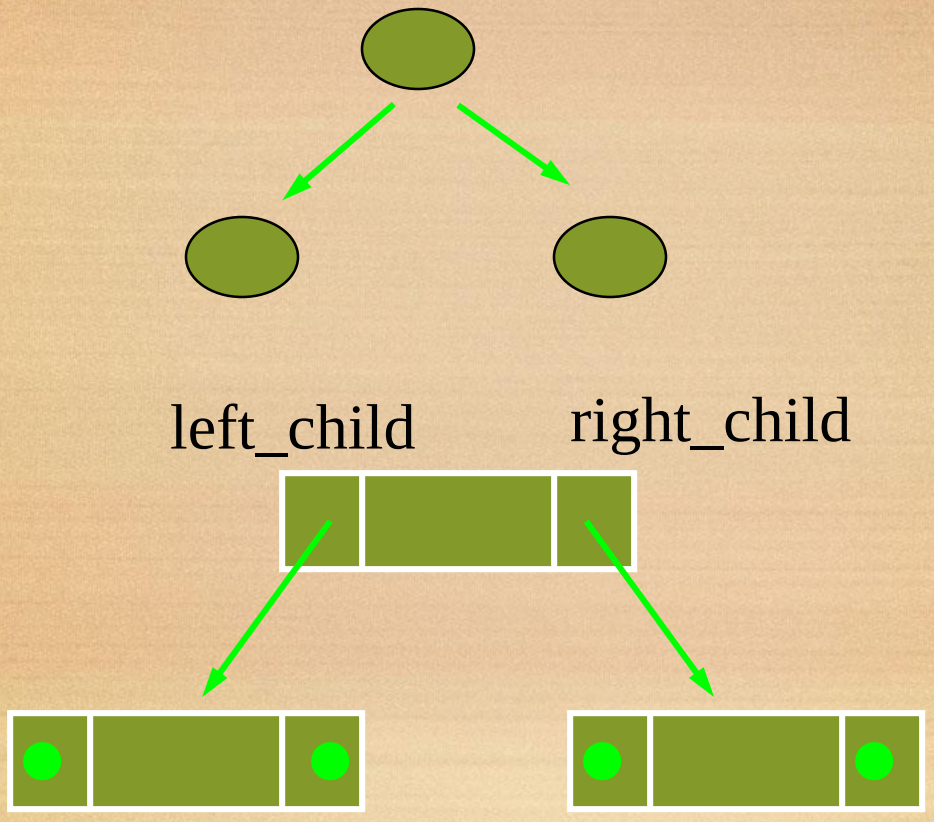


Implementation using Linked Lists

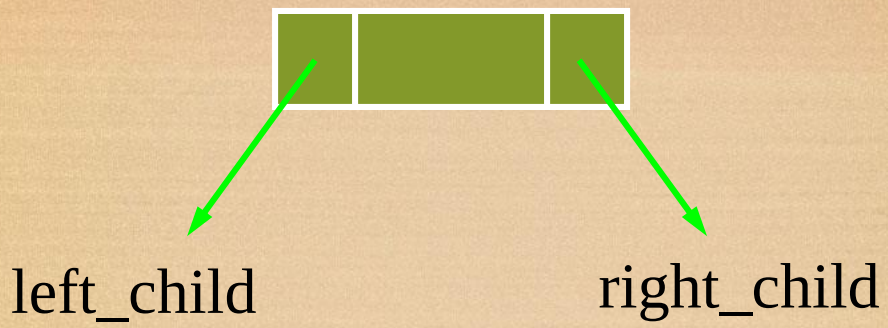




Implementation using Linked Lists



Implementation using Linked Lists



Implementation using Linked Lists

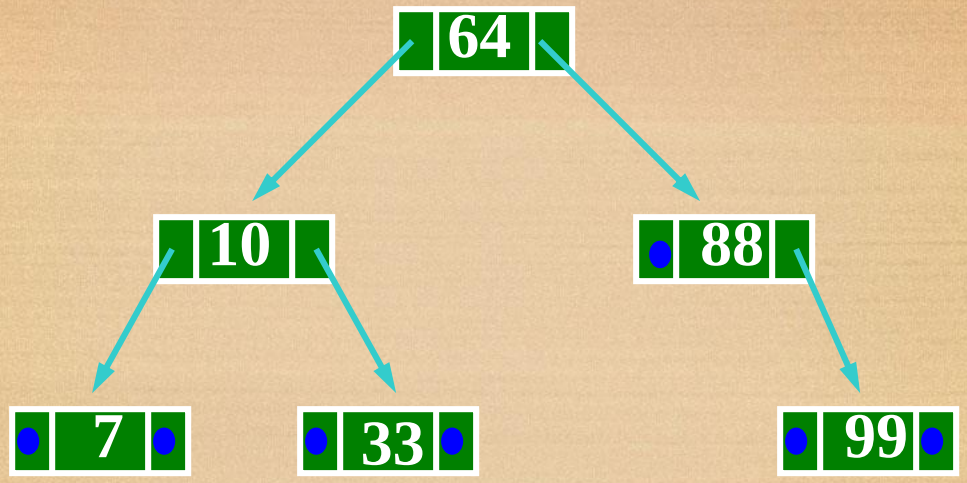
Left child Right child

```
typedef int ELEMENT;
typedef struct BTREE {
        ELEMENT data;
        struct BTREE *Lchild, *Rchild;
} BTREE;

BTREE *tree2;
```

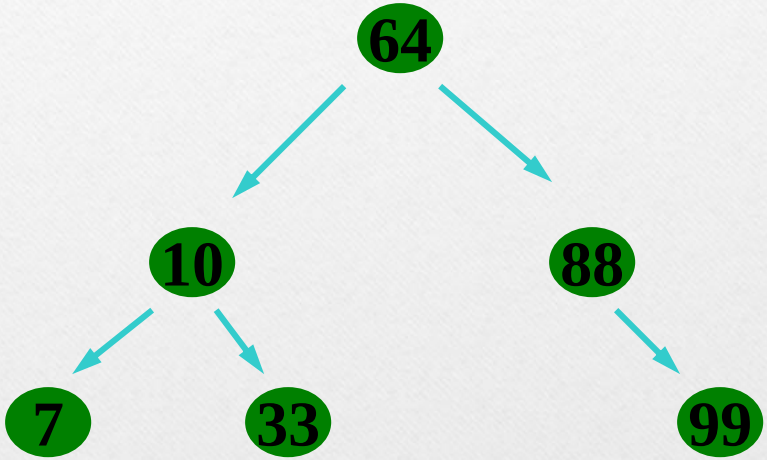
Binary Search Trees

- To store nodes in a particular order
- BST is a special type of Binary Trees
- The value of each node must be :
  - Higher than the value of its left child
  - Lower than the value of its right child
- Example

Binary Search Trees



Binary Search Trees



Basic Operations of BST

- Create BST -
  - Assign NULL to the pointer to BST
- Search BST -
  - Determin whether a particular node exist in the BST BST
- Insert a node into BST -
  - What happen if there are nodes in the BST???
- Traverse BST -
  - Visit every node
- Delete a node from BST -
  - What happen if there is no node in the BST???

Create BST

```
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

Create BST

```
typedef BTREE BST;
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

Create BST

```
typedef struct BTREE {
        ELEMENT data;
        struct BTREE *Lchild, *Rchild;
} BTREE;

typedef BTREE BST;
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

Create BST

```
typedef int ELEMENT;
typedef struct BTREE {
        ELEMENT data;
        struct BTREE *Lchild, *Rchild;
} BTREE;

typedef BTREE BST;
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

Create BST

```
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

VERSUS

```
void CreateBST( BST *p2) {
        p2 = NULL;
}
```

?

Create BST

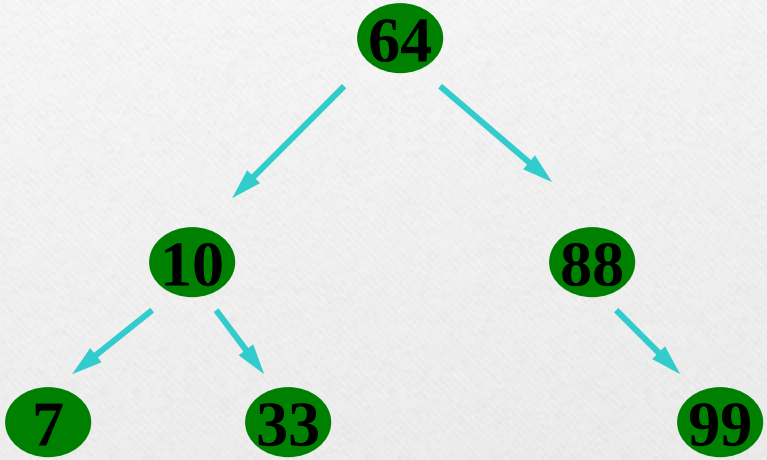
```
void CreateBST( BST **p2) {
        *p2 = NULL;
}
```

VERSUS

```
void CreateBST( BST *p2) {
        p2 = NULL;
}
```

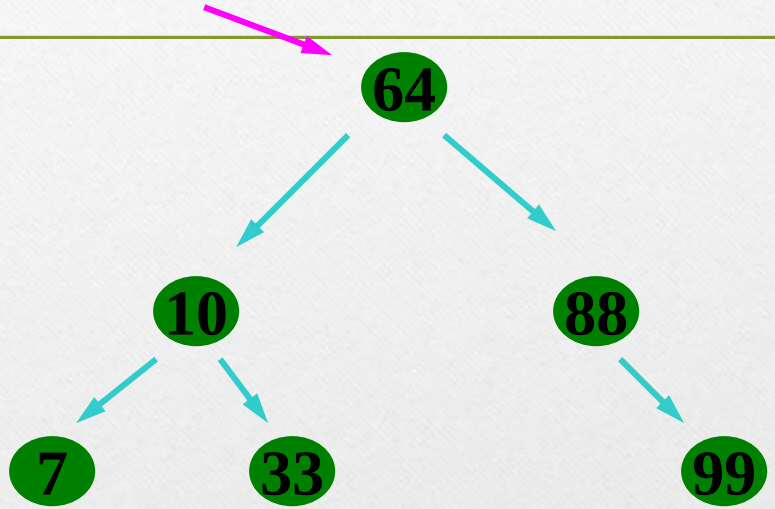
Search BST

Find 33



Search BST

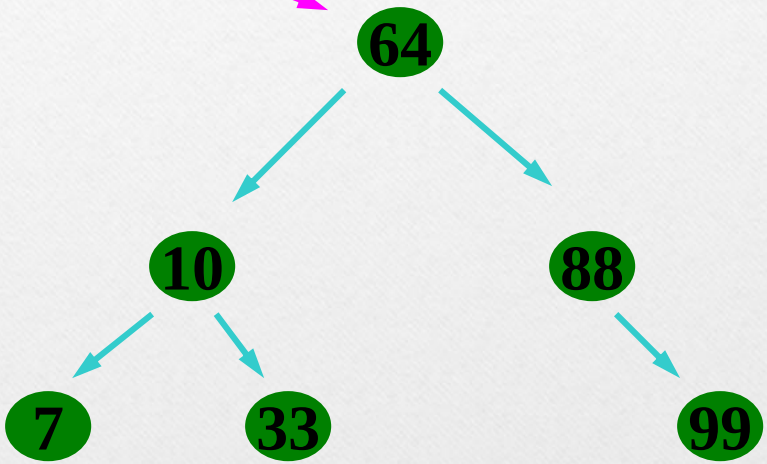
Find 33



Search BST

33 = 64?

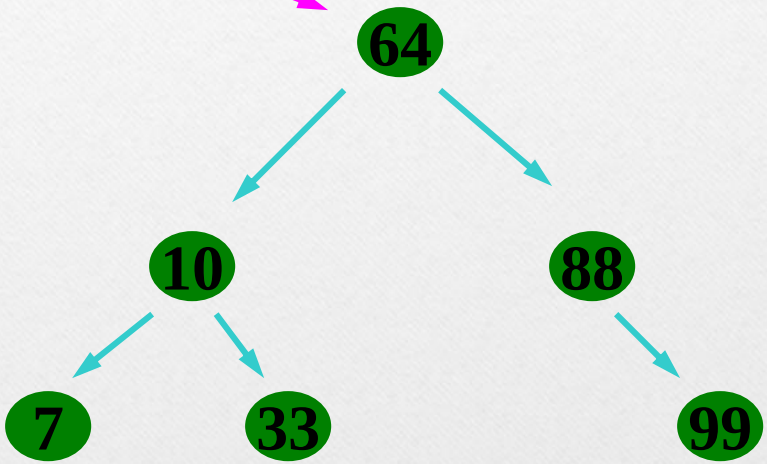
Find 33



Search BST

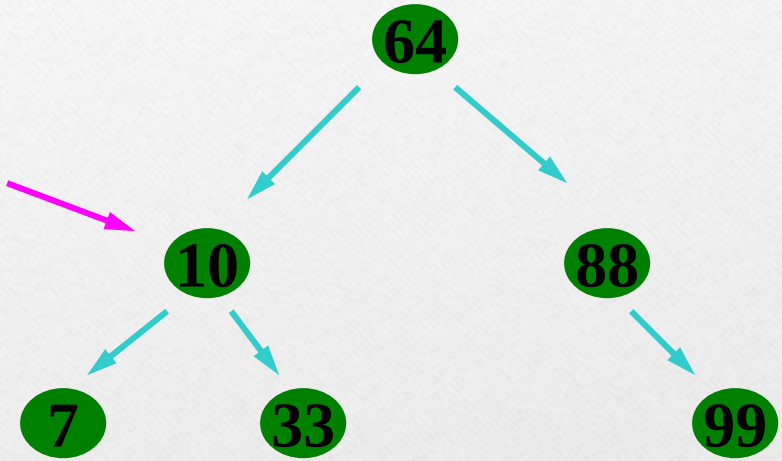
33 < 64?

Find 33



Search BST

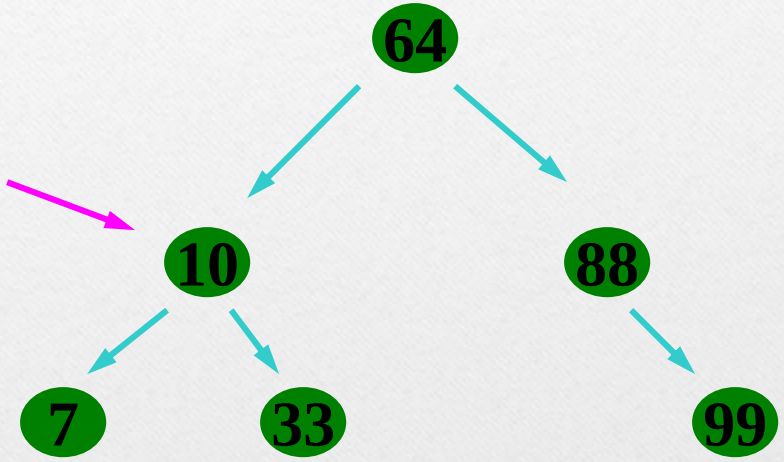
Find 33



Search BST

Find 33

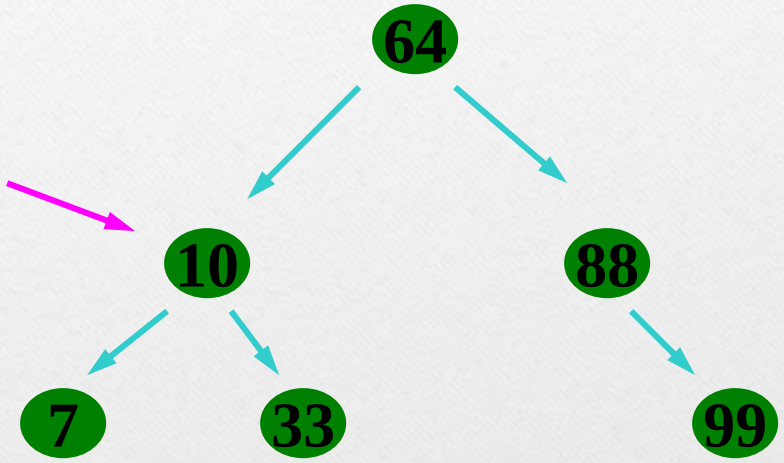
33 = 10?



Search BST

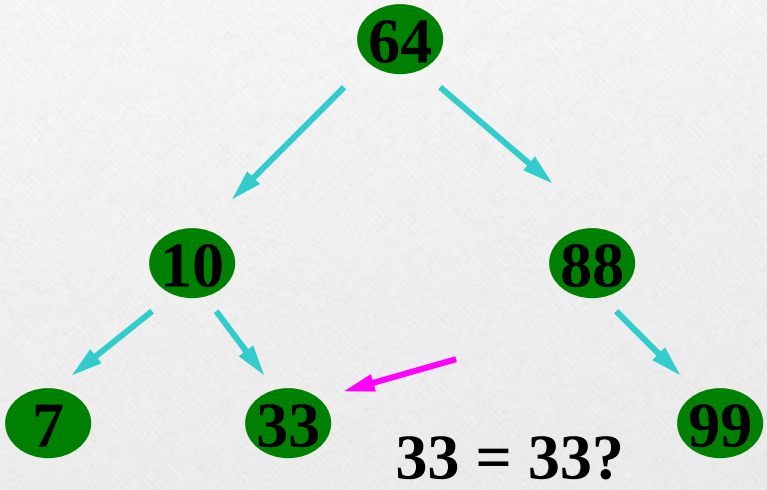
Find 33

33 < 10?



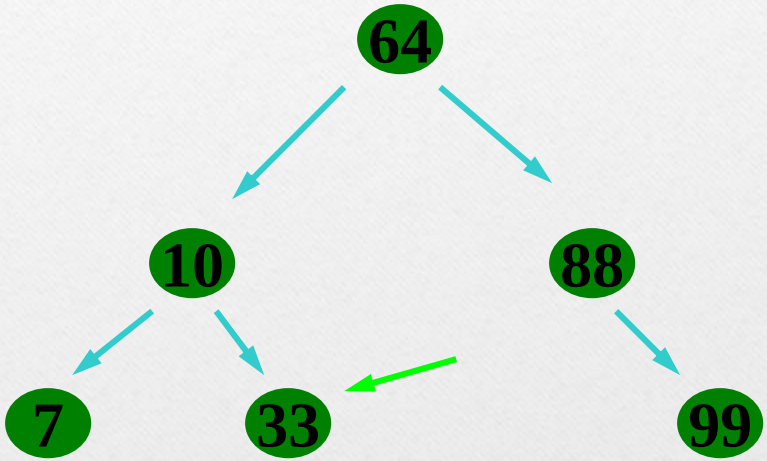
Search BST

Find 33



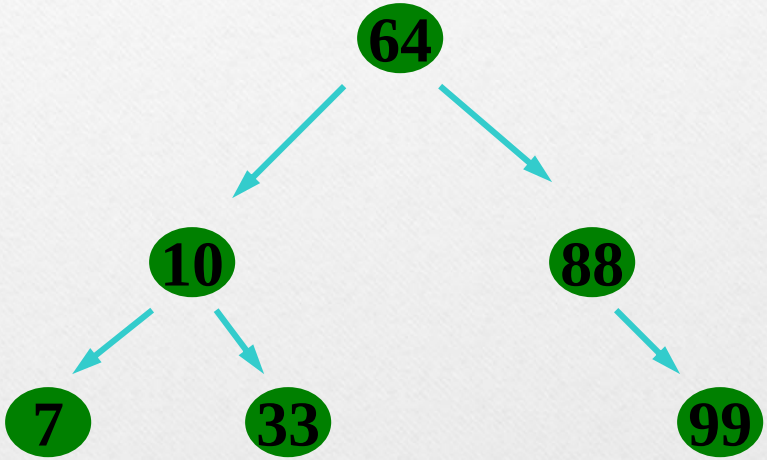
Search BST

Find 33



Search BST

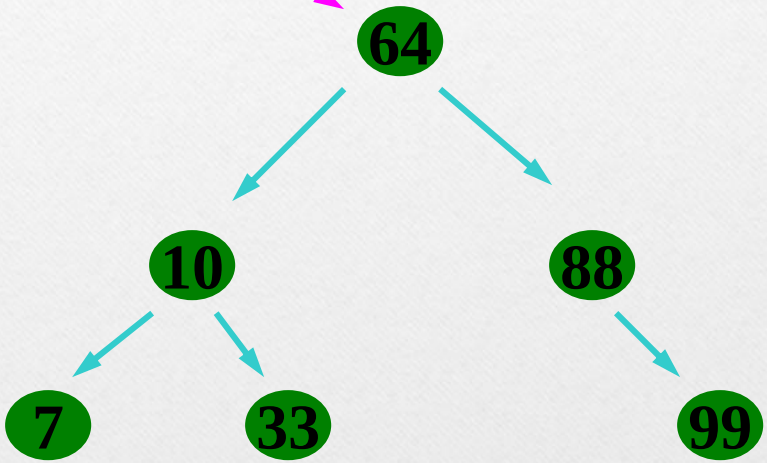
Find 6



Search BST

6 = 64?

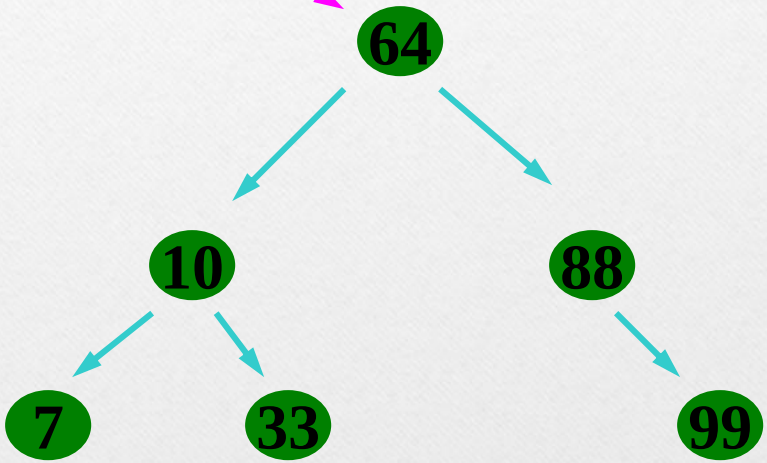
Find 6



Search BST

6 < 64?

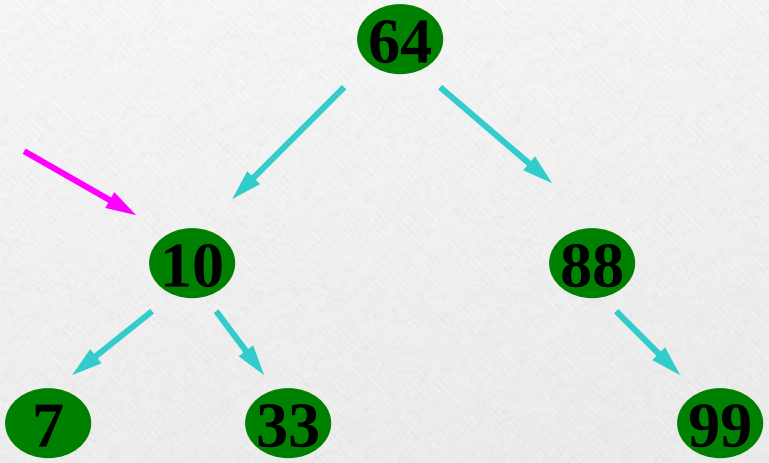
Find 6



Search BST

Find 6

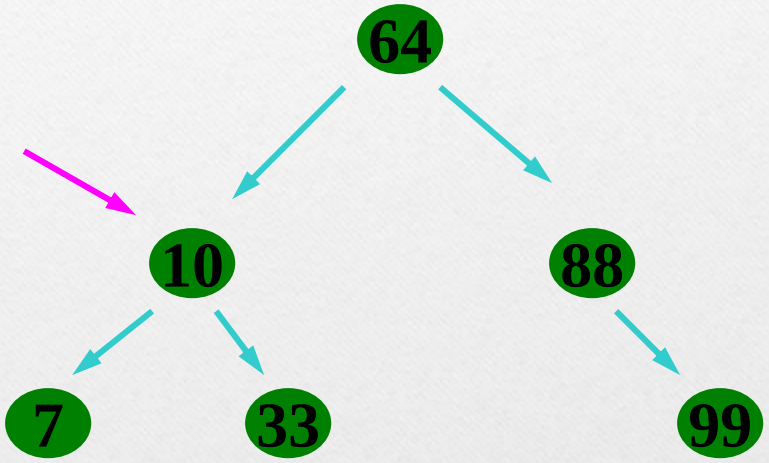
6 = 10?



Search BST

Find 6

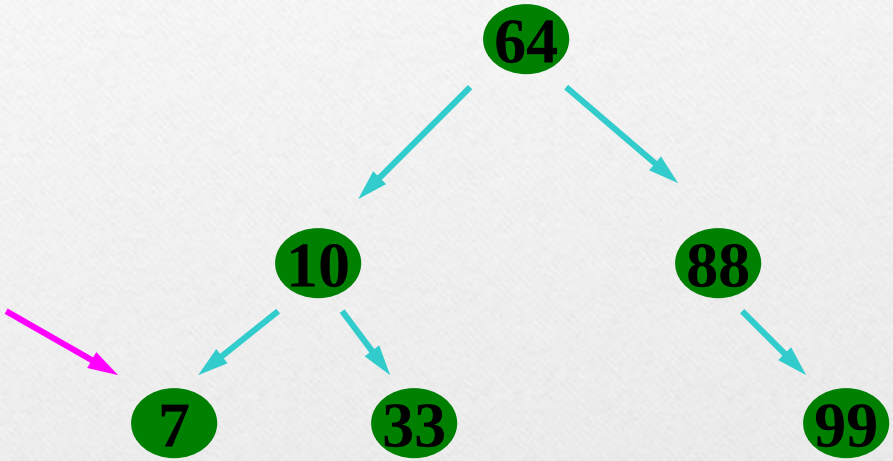
6 < 10?



Search BST

Find 6

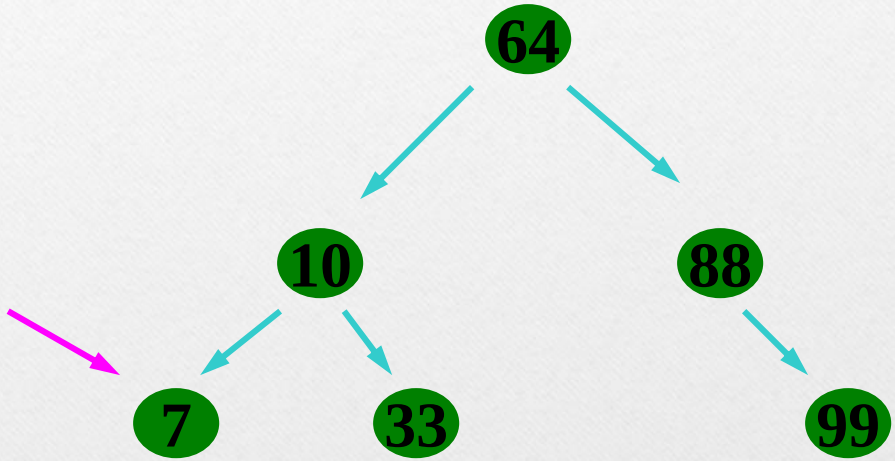
6 = 7?



Search BST

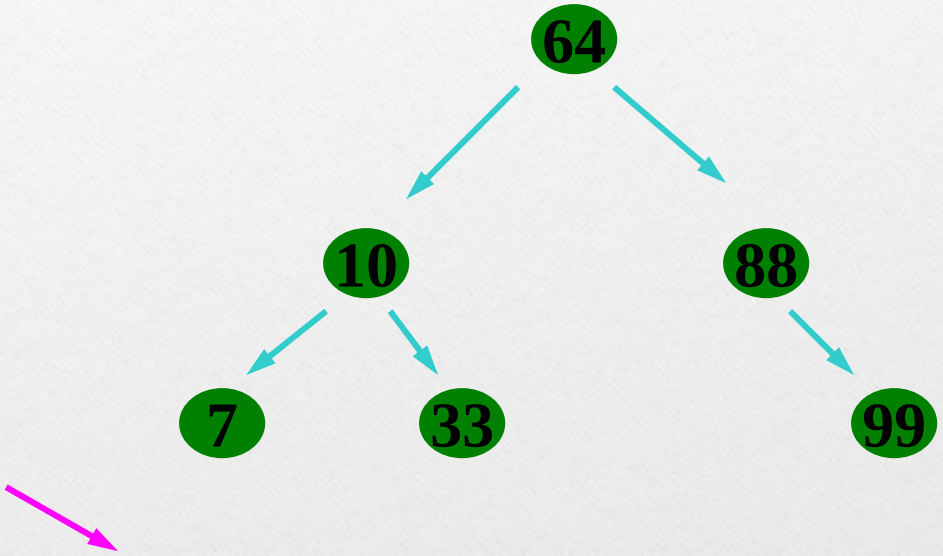
Find 6

6 < 7?



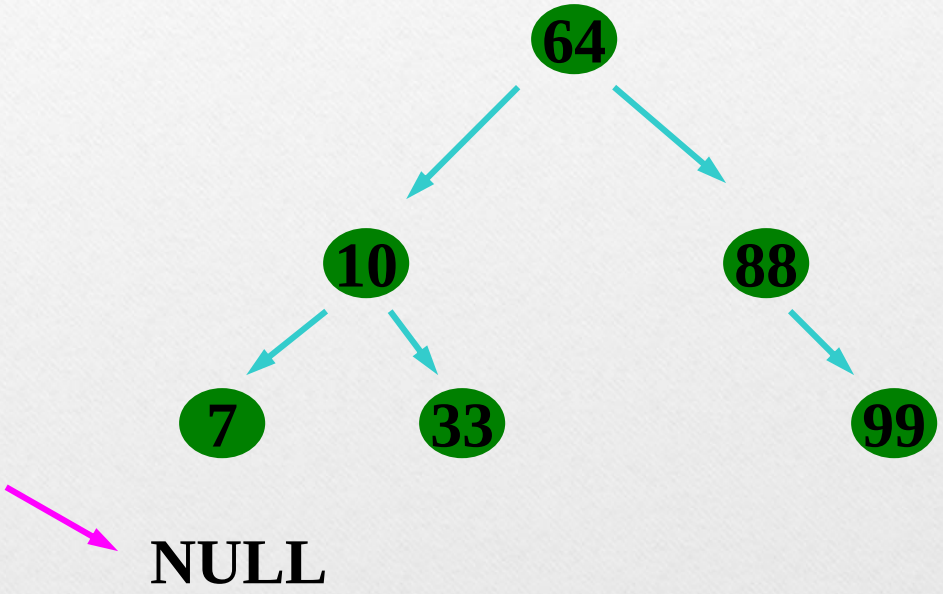
Search BST

Find 6



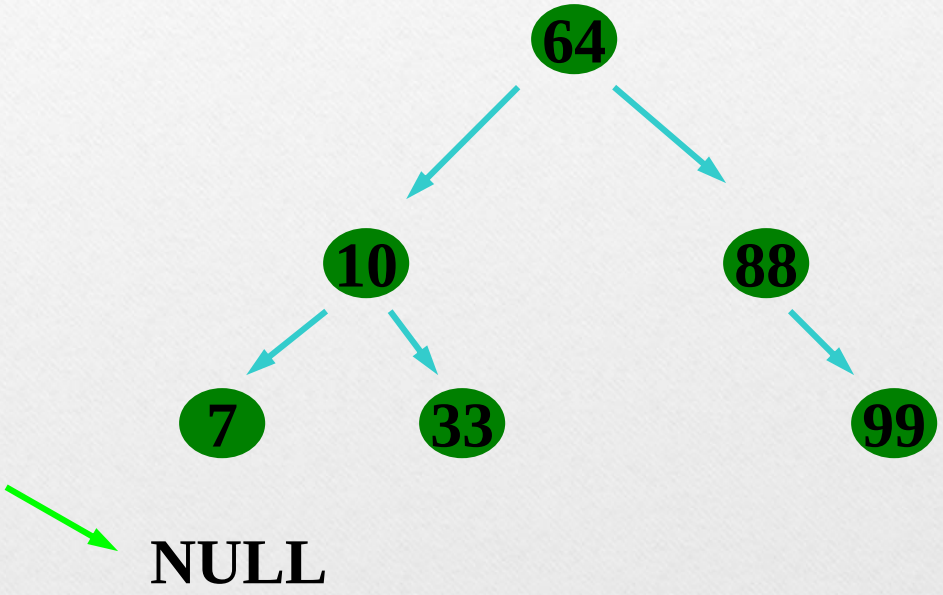
Search BST

Find 6



Search BST

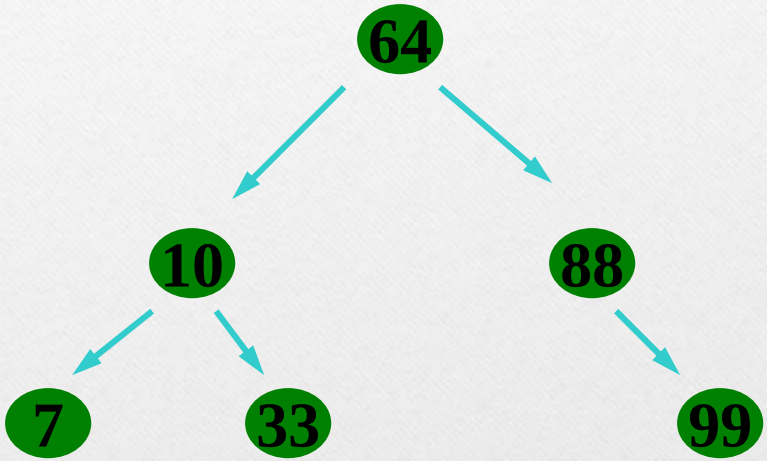
Find 6



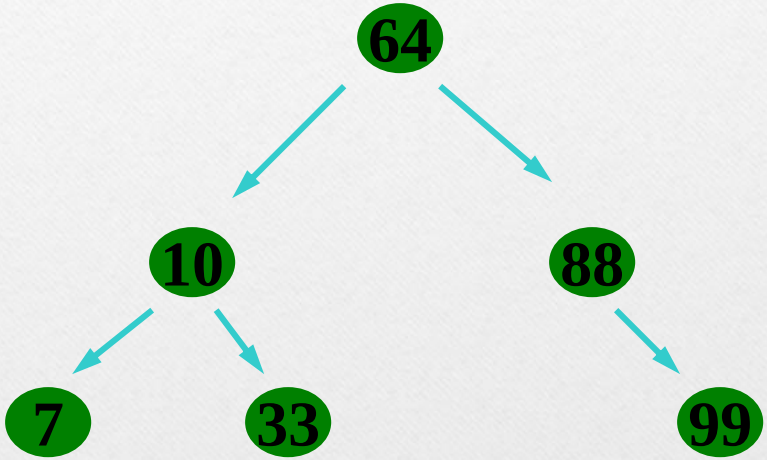
Search BST

```
BST *SearchBST( BST *p2, ELEMENT item) {

        while (p2 && (item != p2->data)) {
                if (item < p2->data)
                        p2 = p2->Lchild;
                else
                        p2 = p2->Rchild;
        }
        return p2;
}
```

Traverse BST

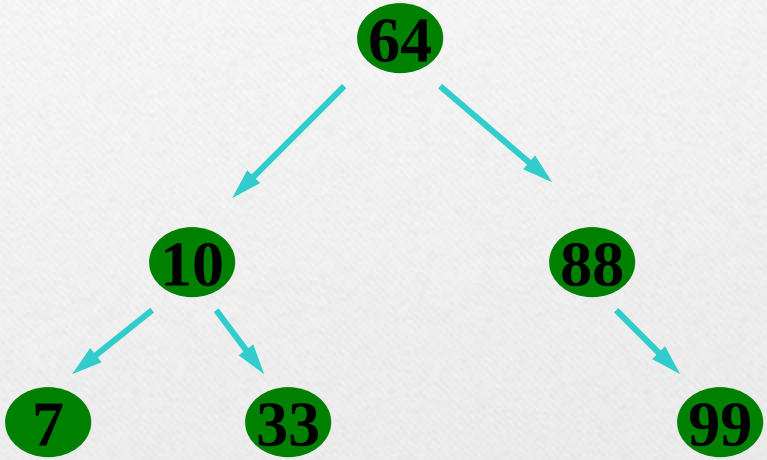


Traverse BST



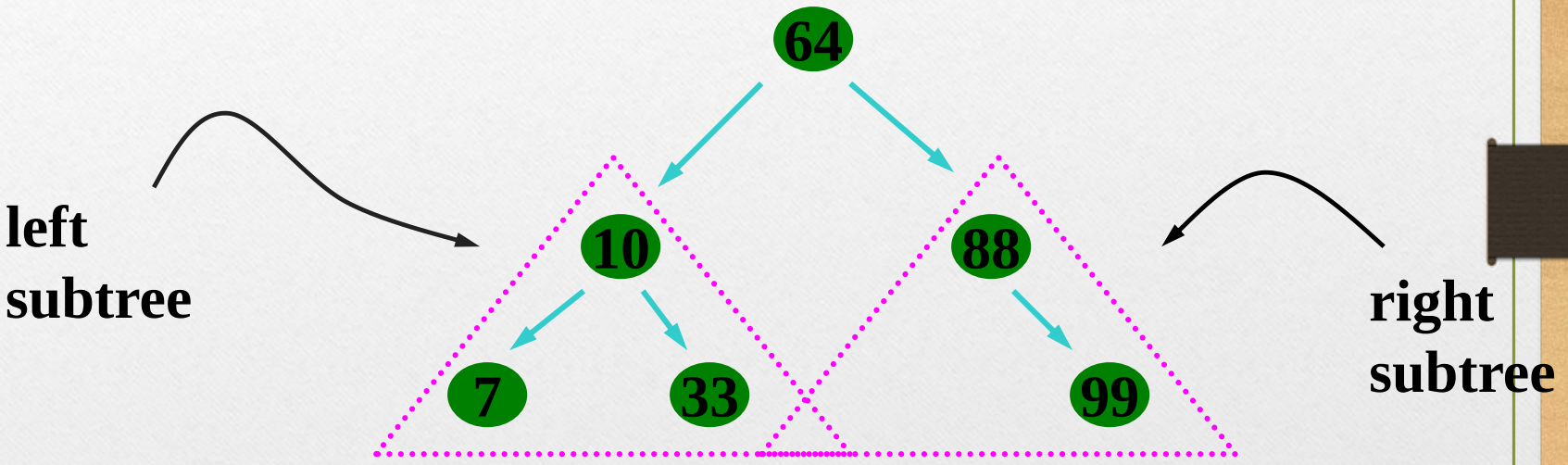
How to traverse????

Traverse BST

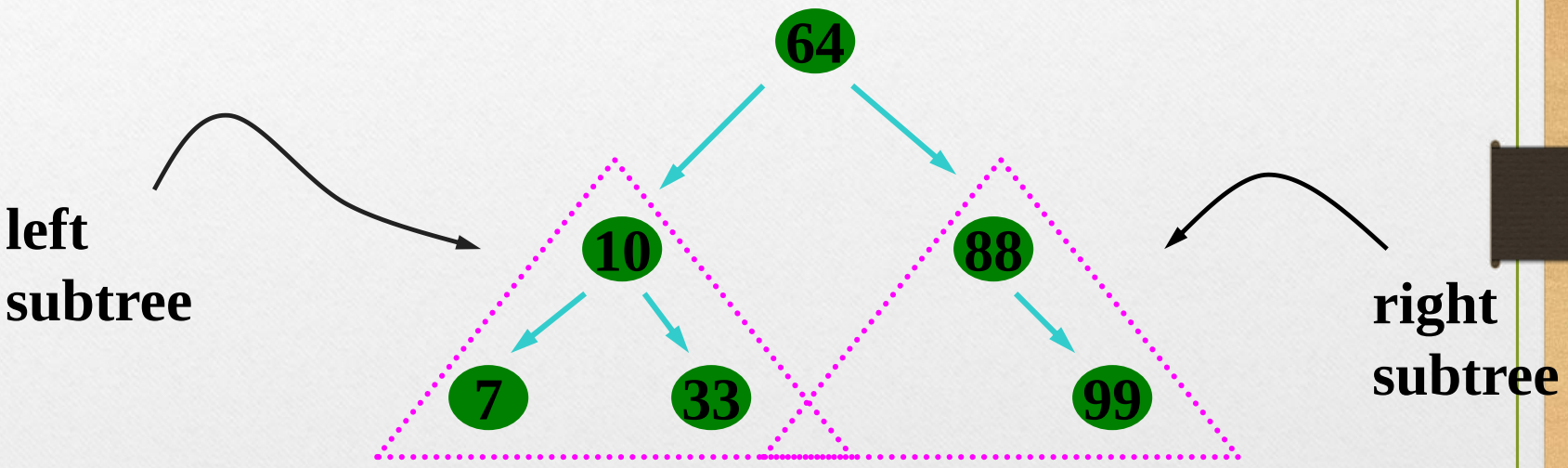


How to traverse????
Assuming the nodes are houses and the arrows are 2-way roads

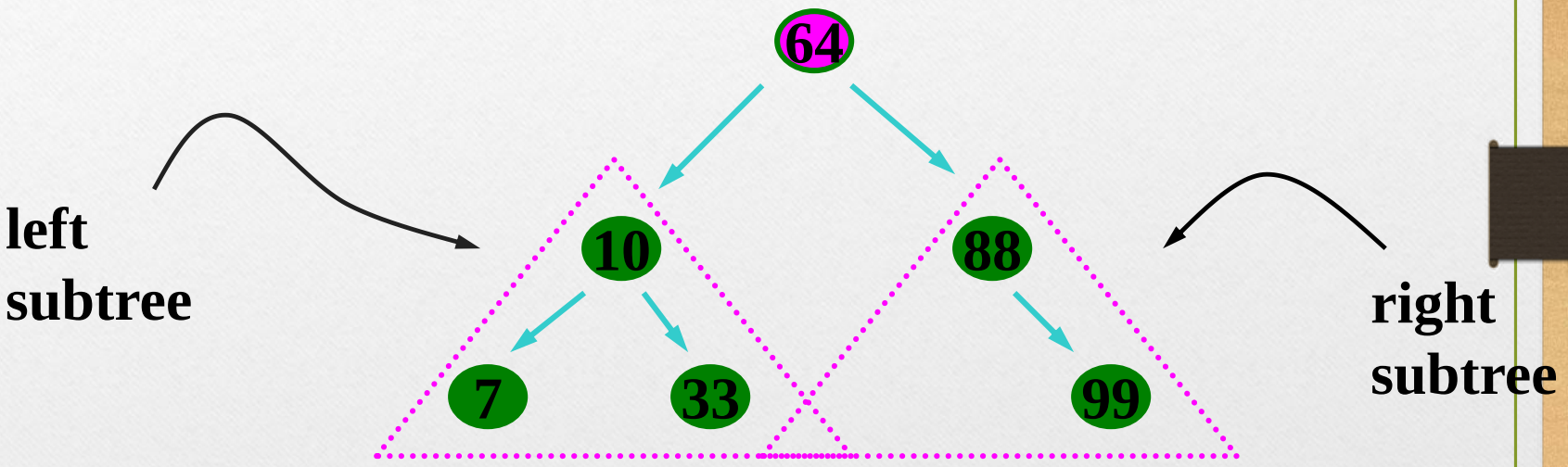
Traverse BST



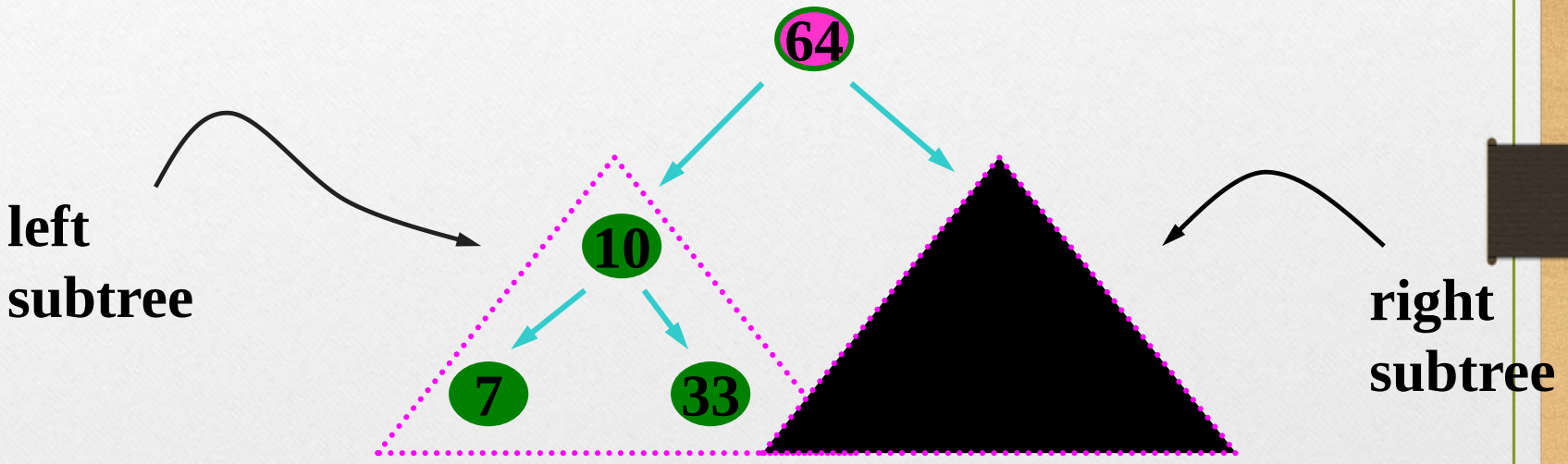
Traverse BST



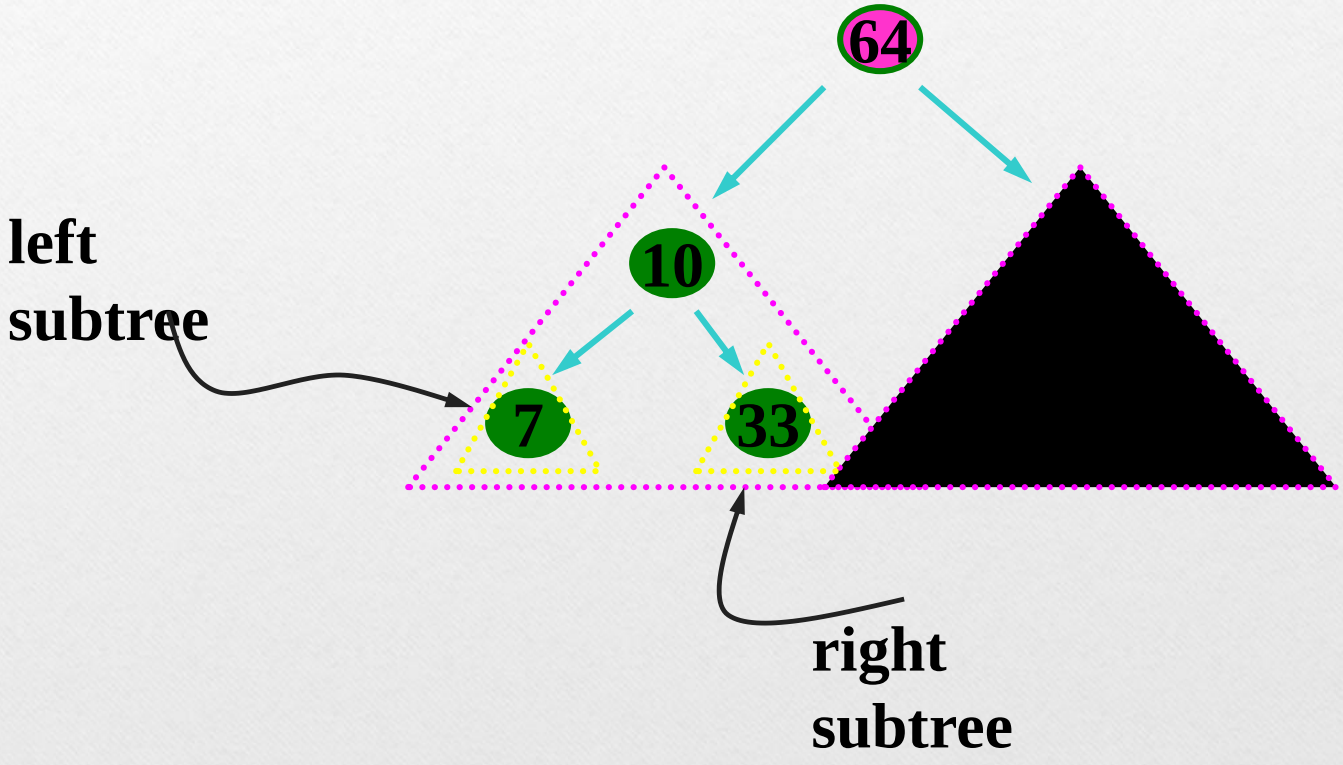
Traverse BST



Traverse BST

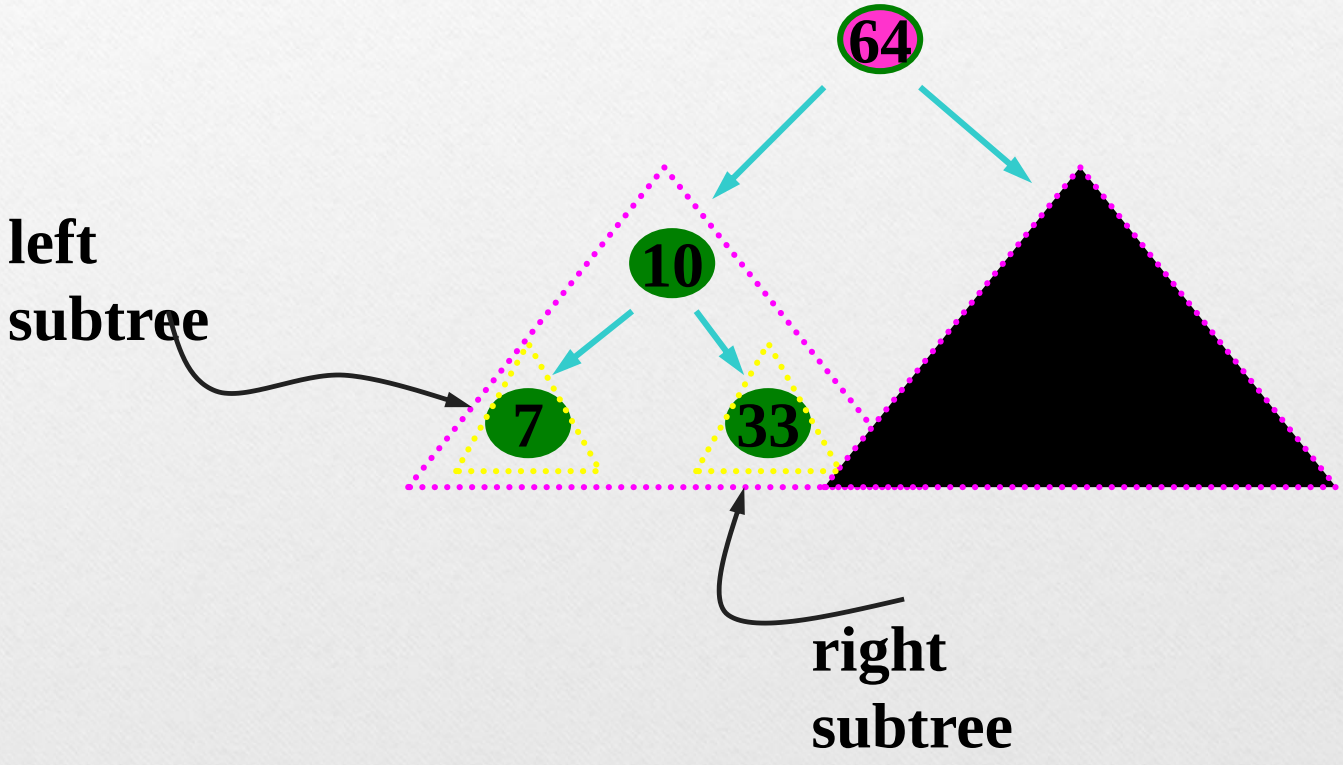


Traverse BST



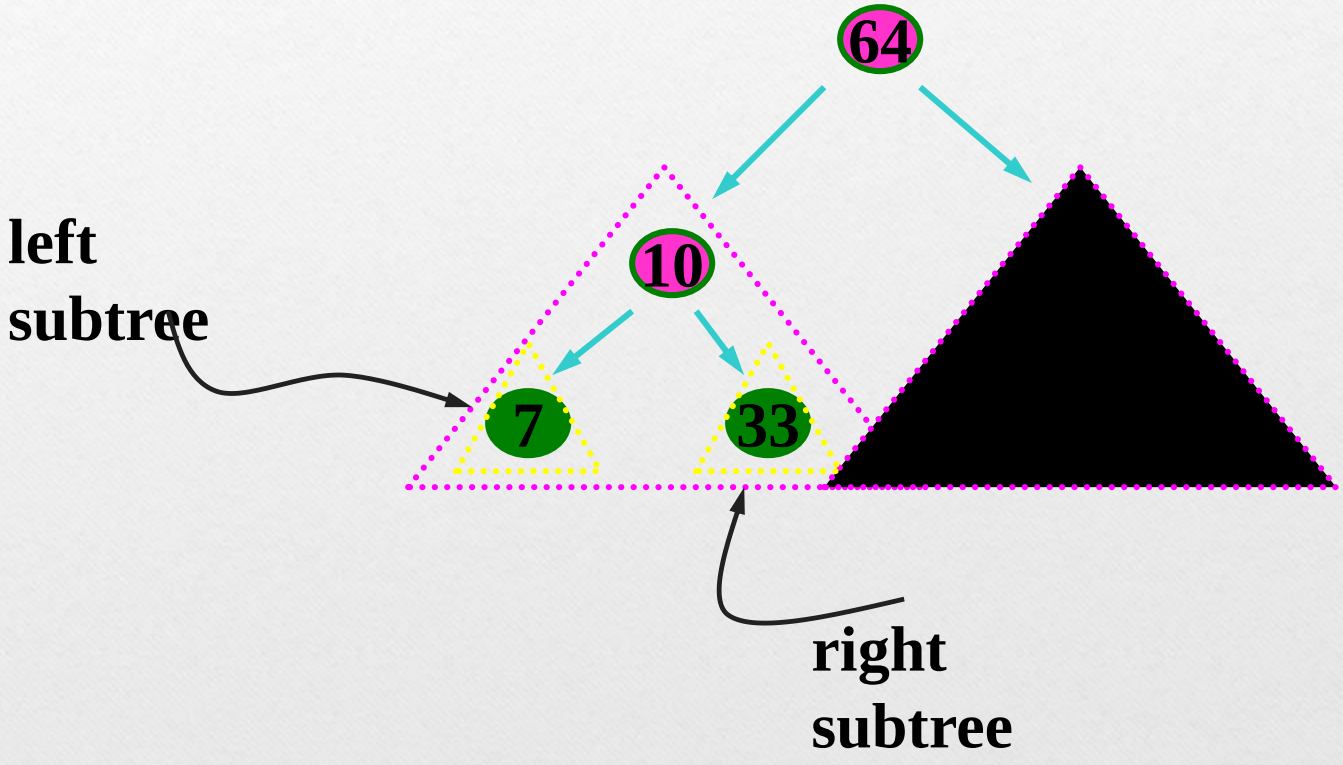
64 10

Traverse BST



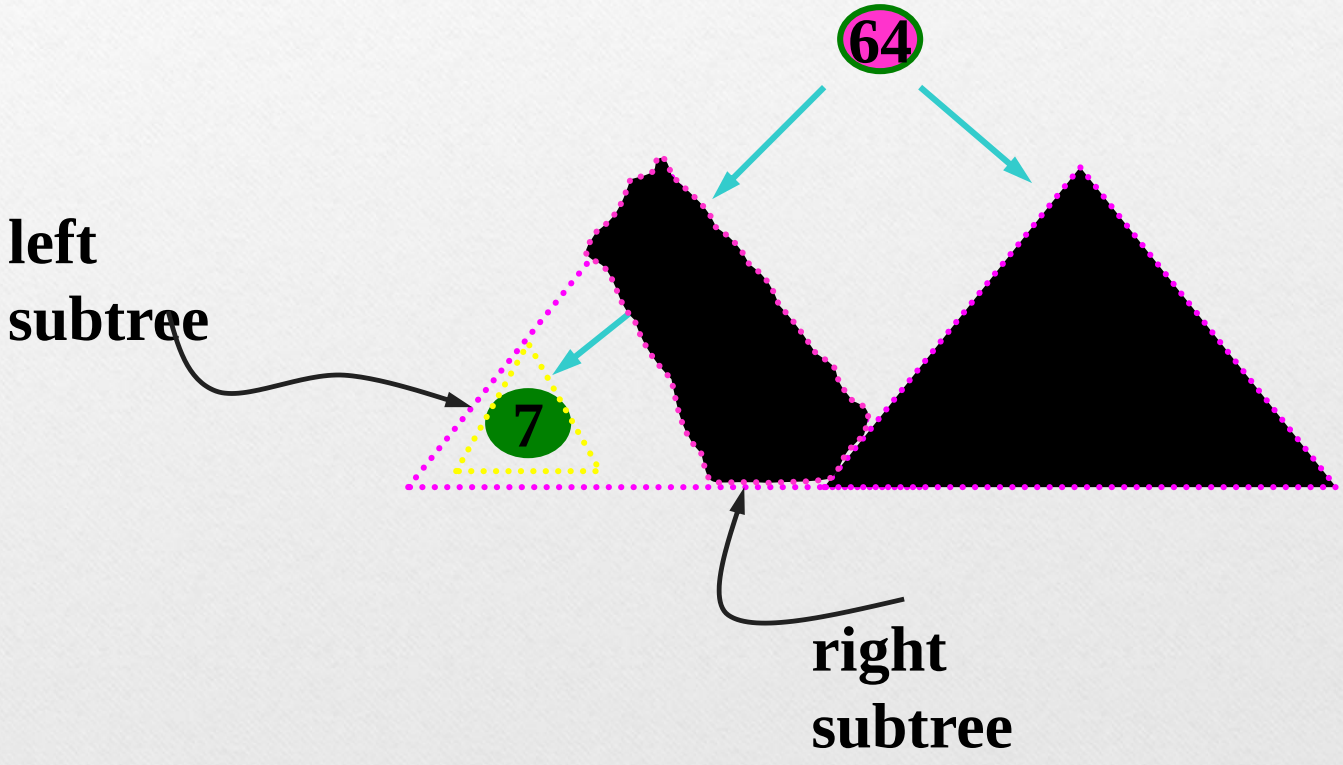
64 10

Traverse BST



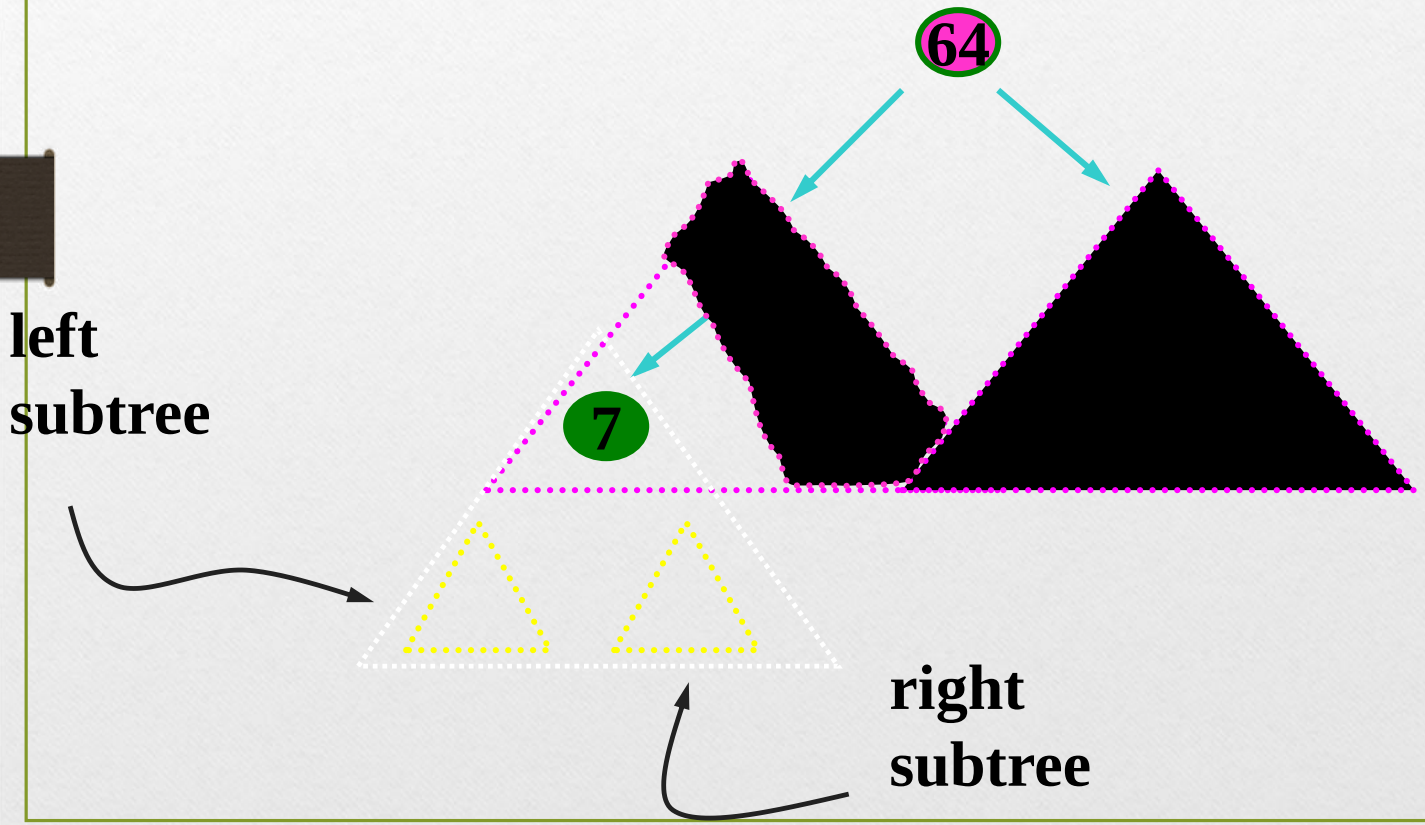
64 10

Traverse BST



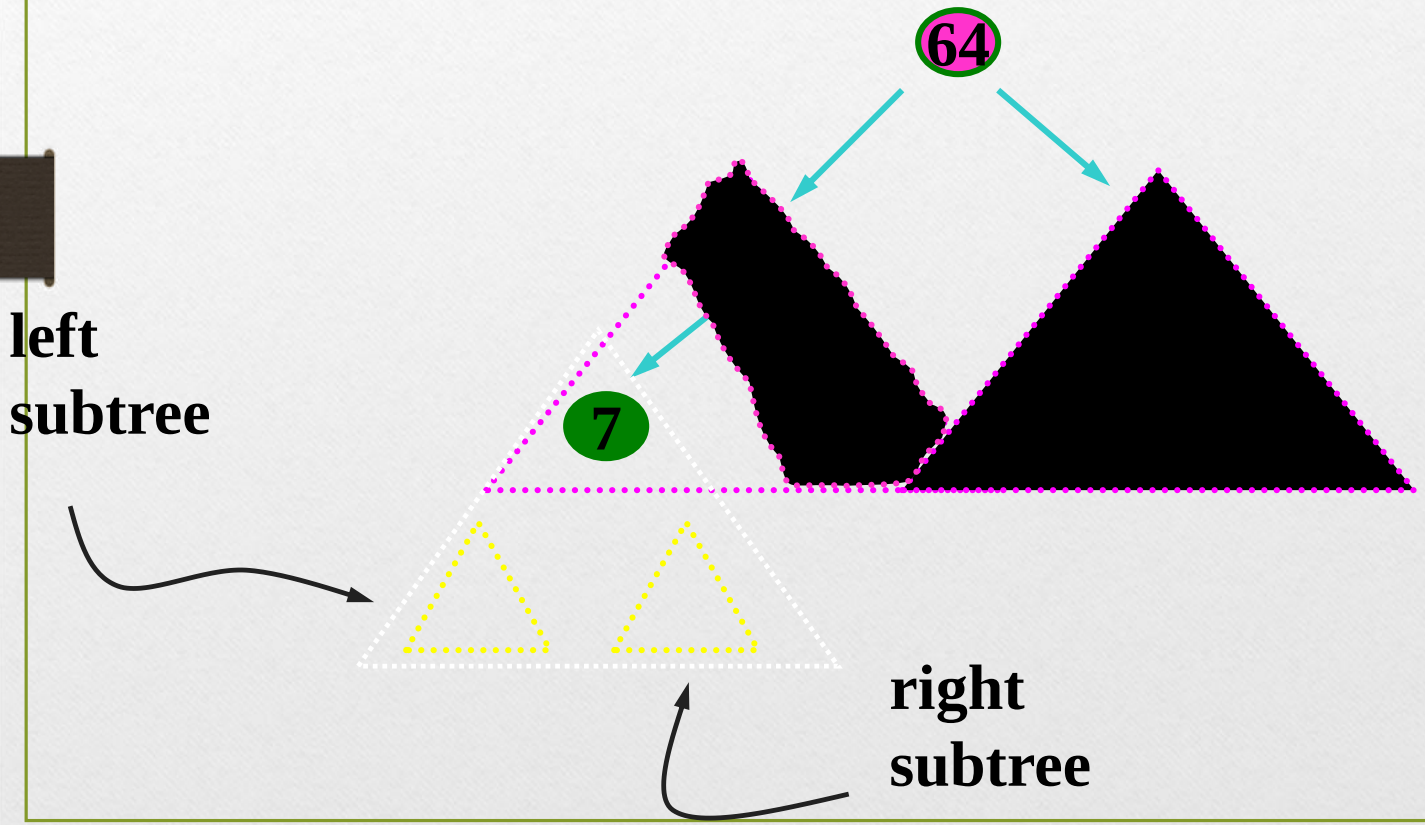
64 10

Traverse BST



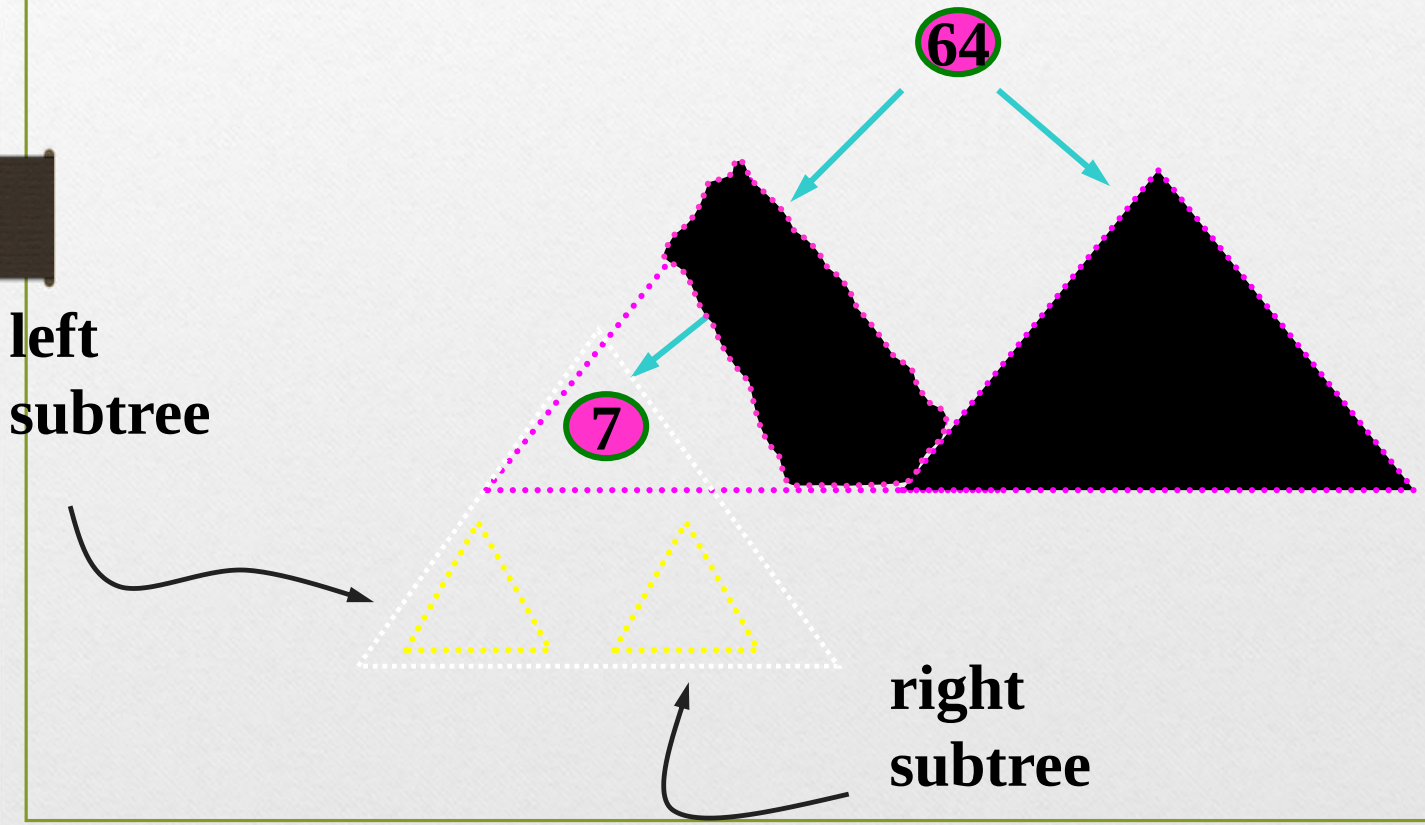
64 10 7

Traverse BST



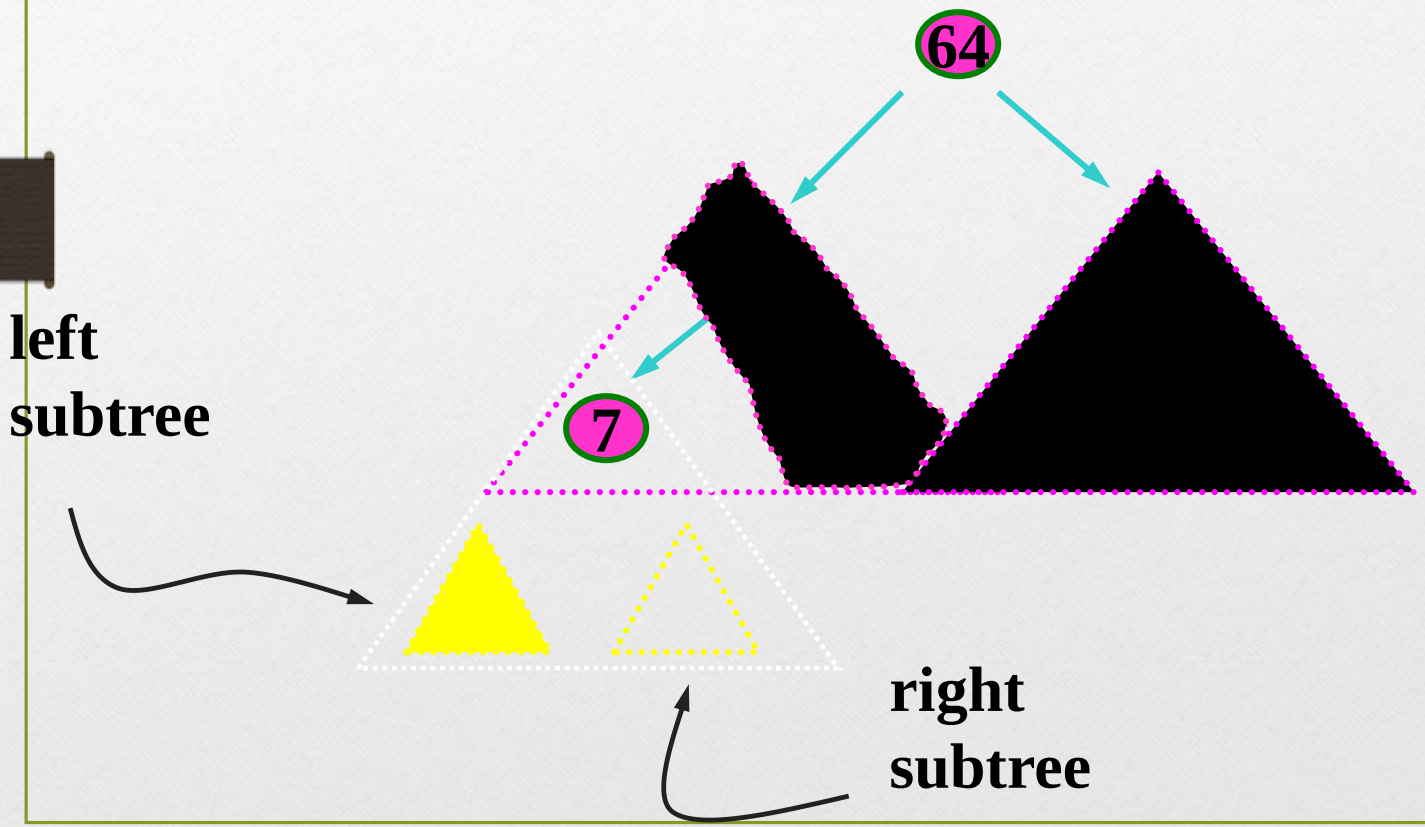
64 10 7

Traverse BST



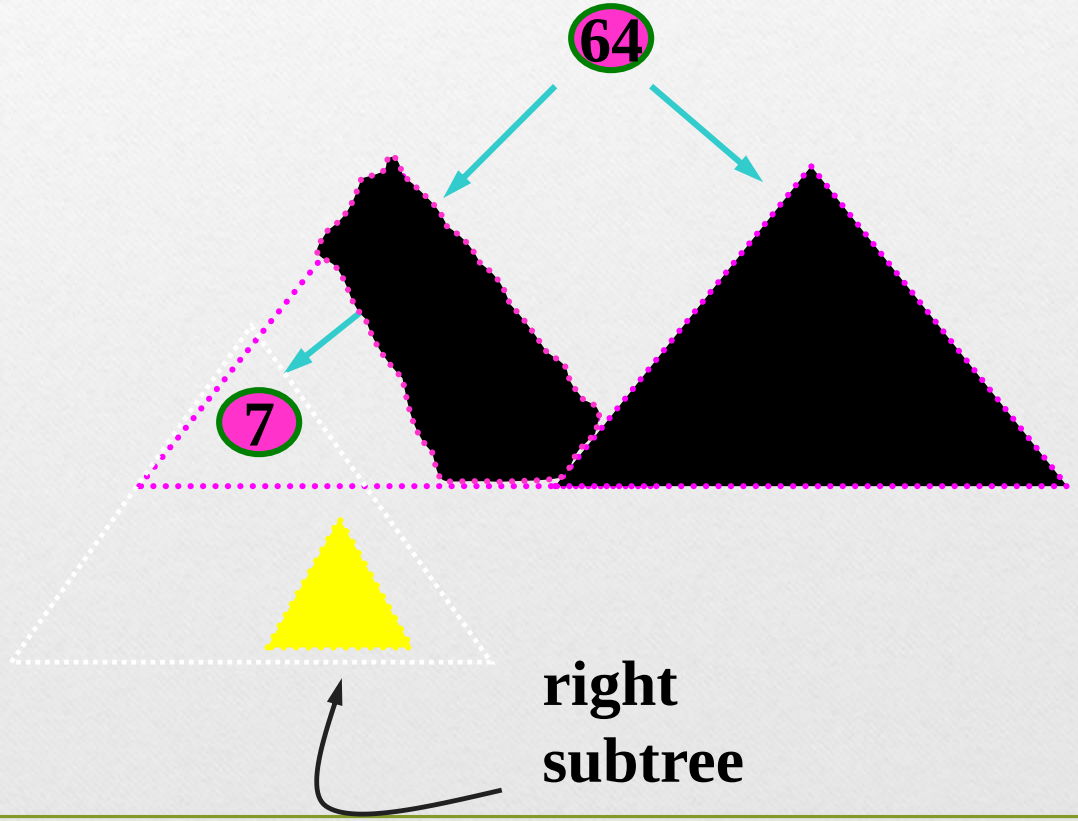
64 10 7

Traverse BST



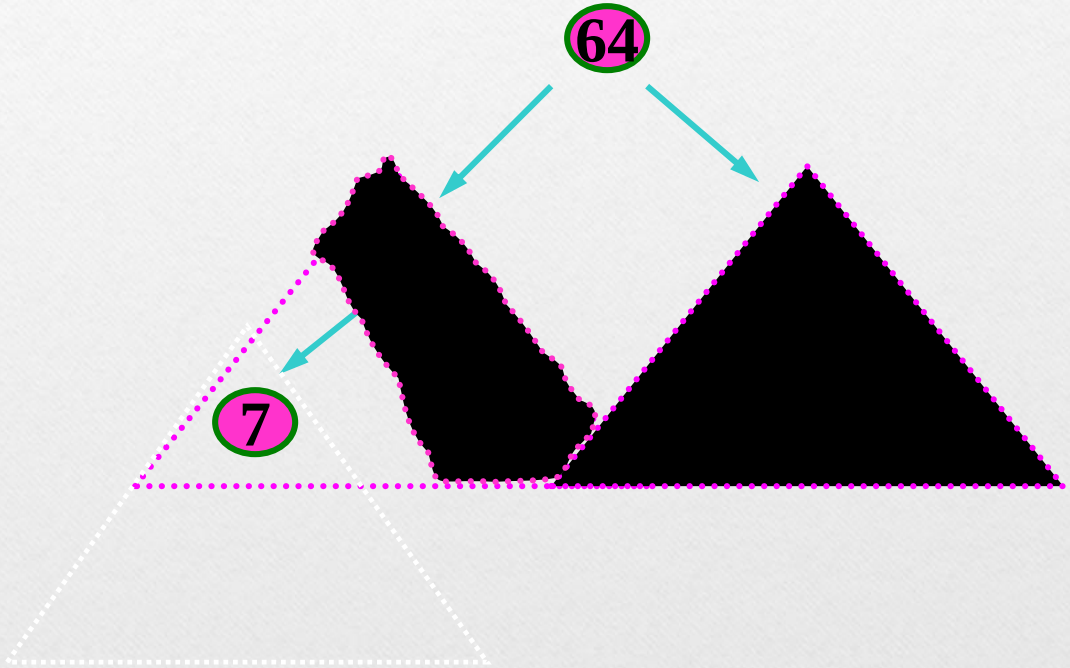
64 10 7

Traverse BST



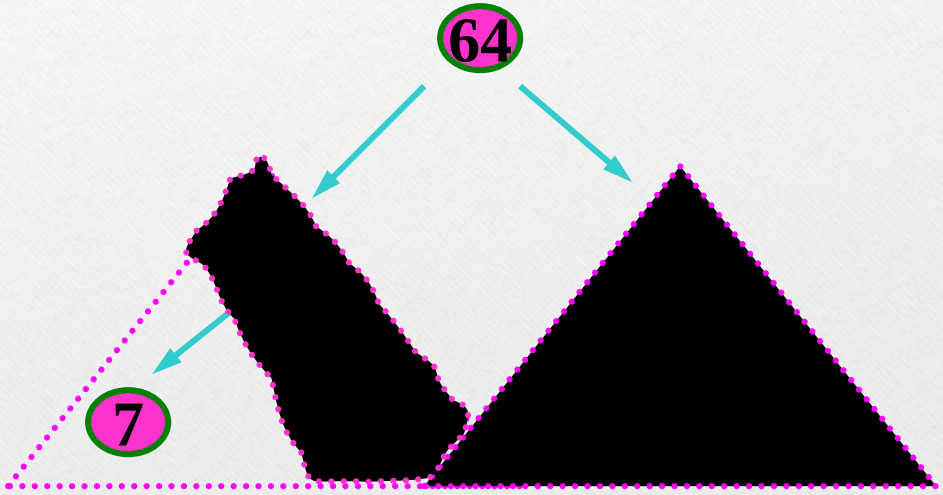
64 10 7

Traverse BST



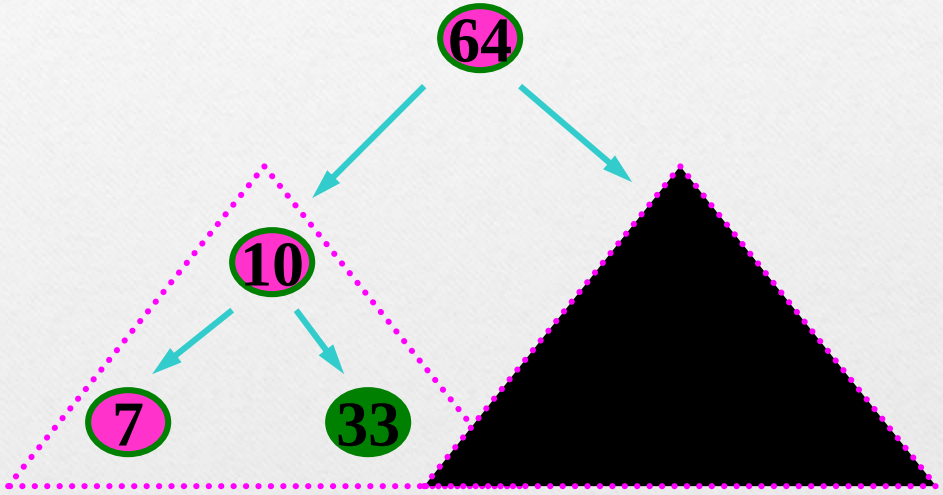
64 10 7

Traverse BST



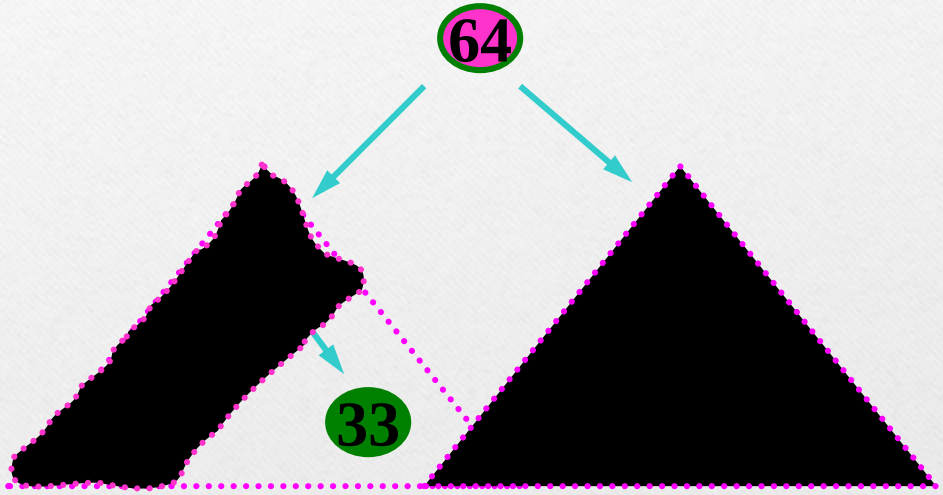
64 10 7

Traverse BST



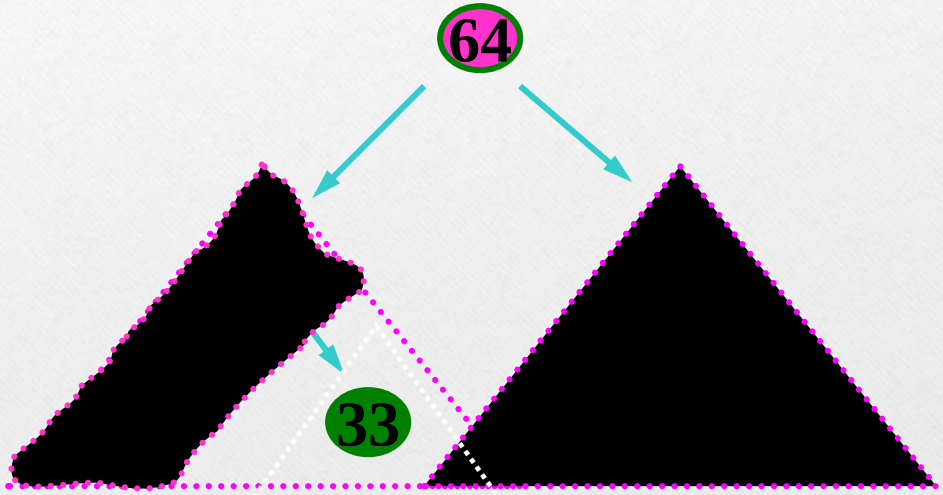
64 10 7

Traverse BST



64 10 7

Traverse BST

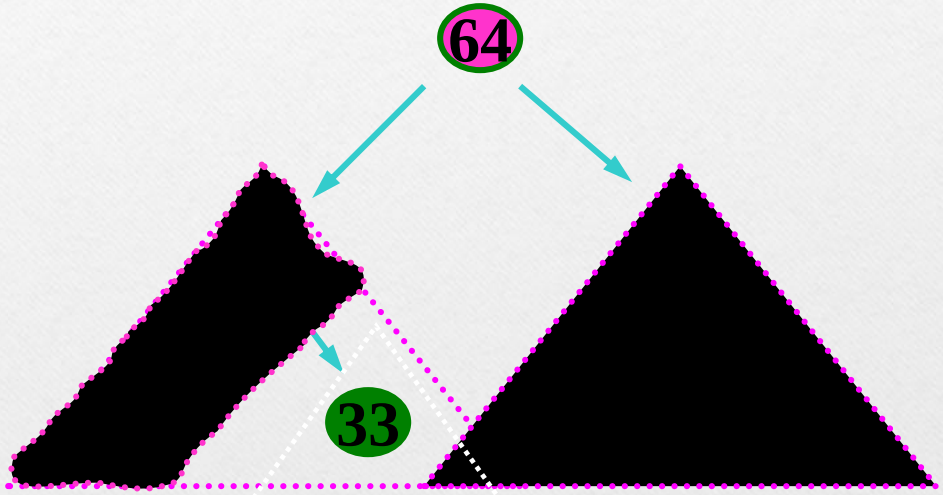


left
subtree

right
subtree

64 10 7 33

Traverse BST

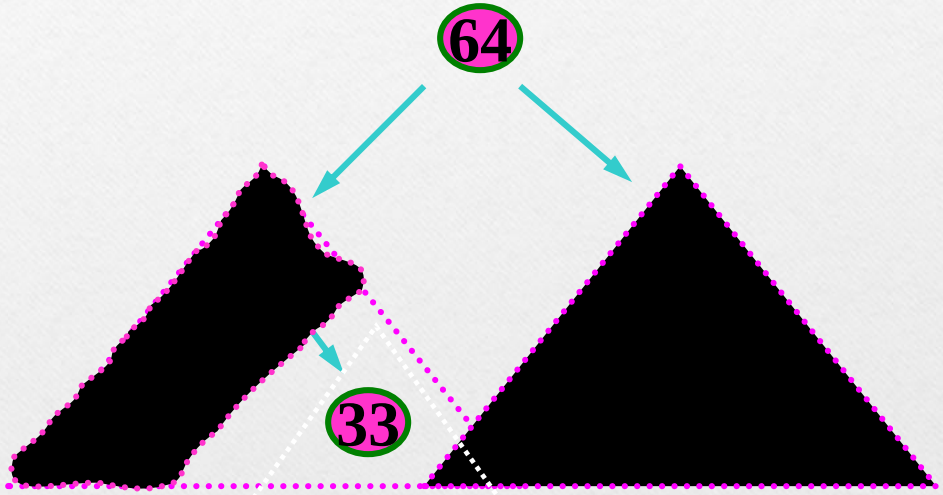


left
subtree

right
subtree

64 10 7 33

Traverse BST

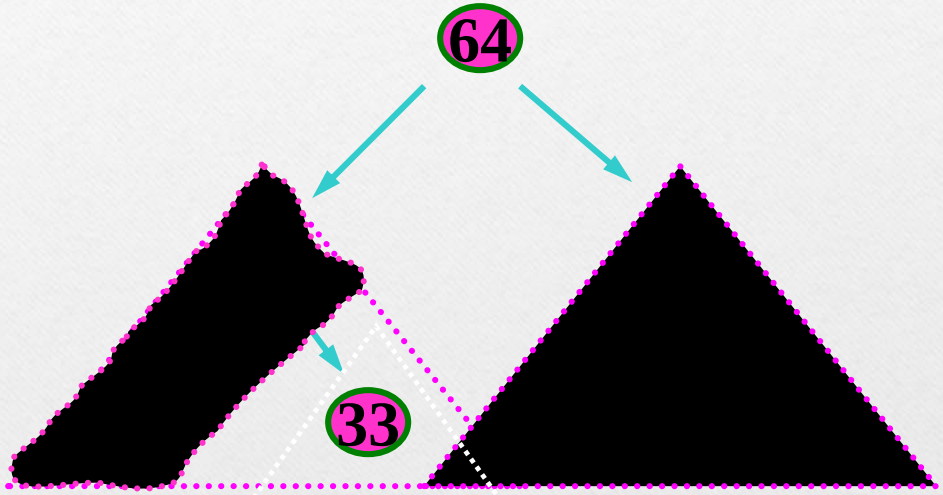


left
subtree

right
subtree

64 10 7 33

Traverse BST

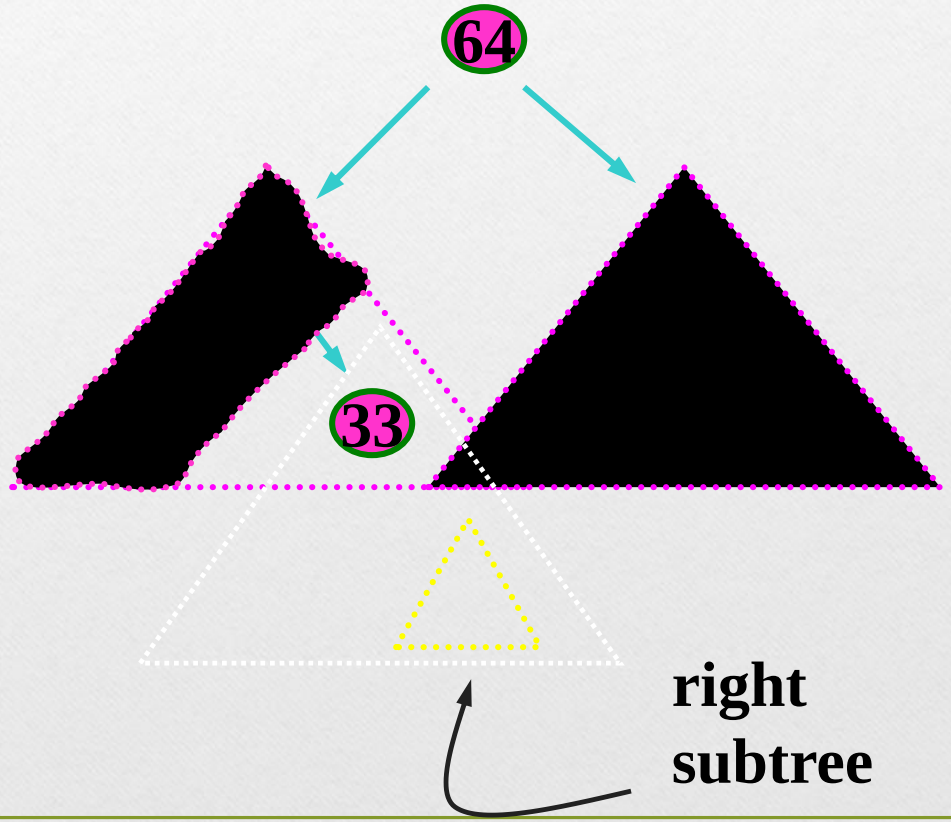


left
subtree

right
subtree

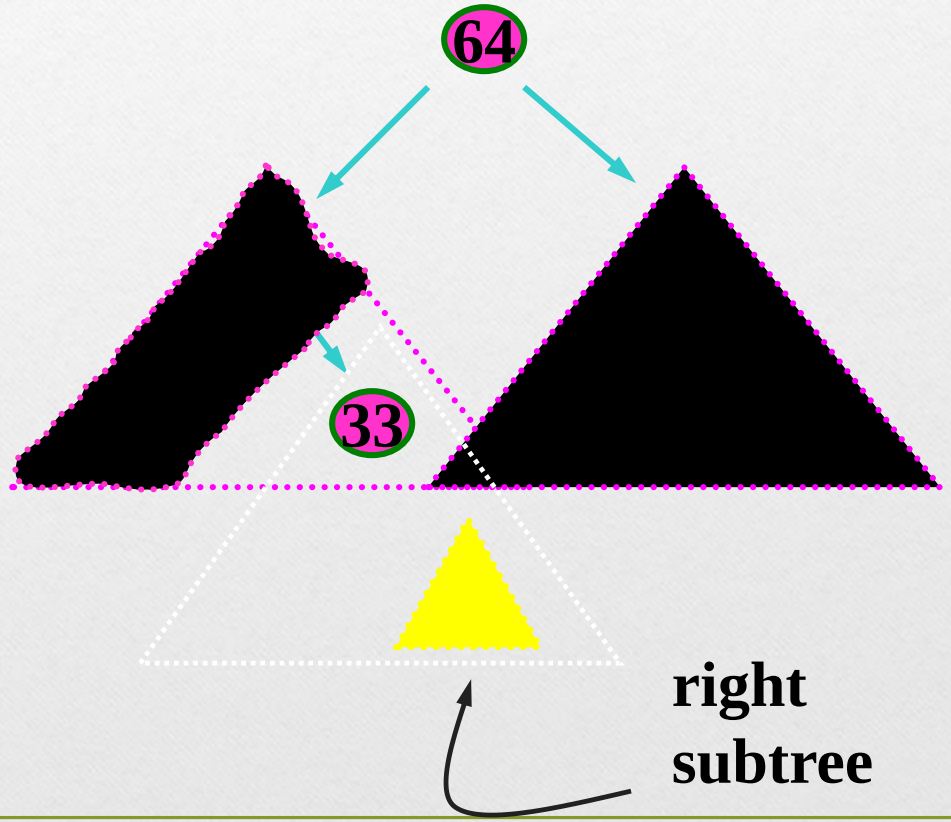
64 10 7 33

Traverse BST



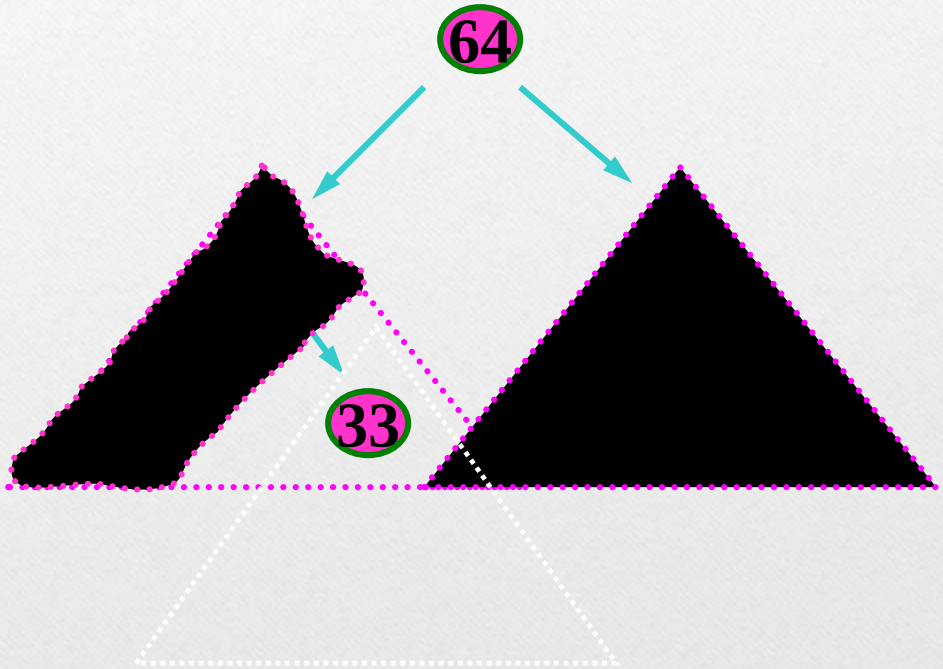
64 10 7 33

Traverse BST



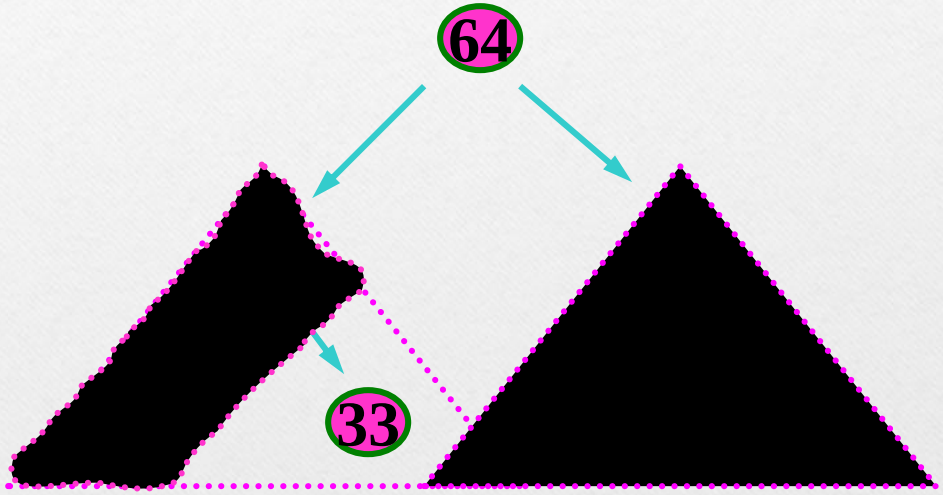
64 10 7 33

Traverse BST



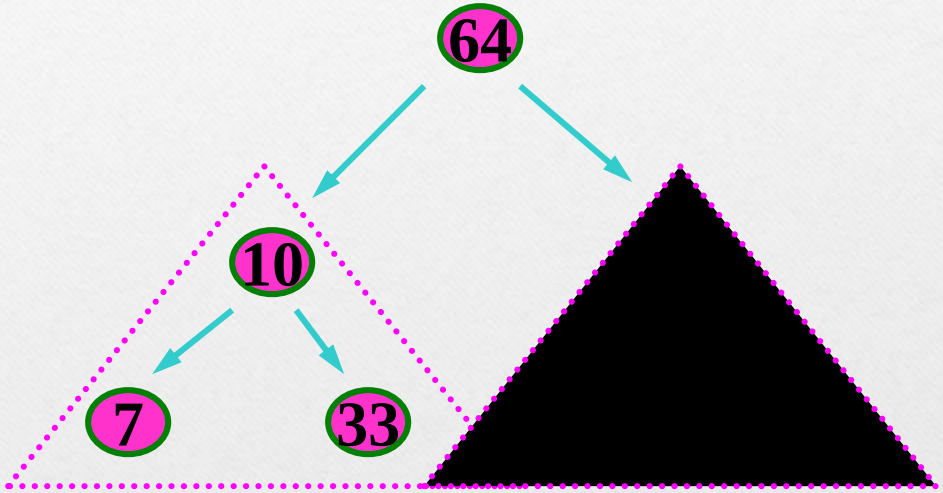
64 10 7 33

Traverse BST



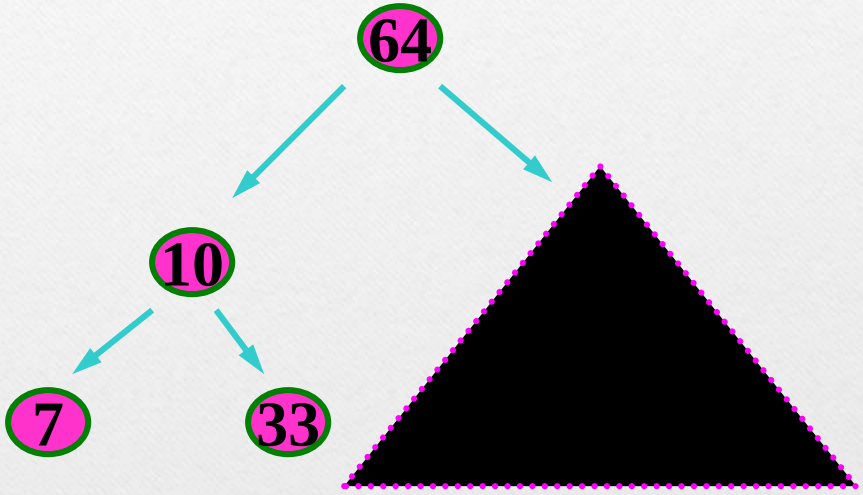
64 10 7 33

Traverse BST



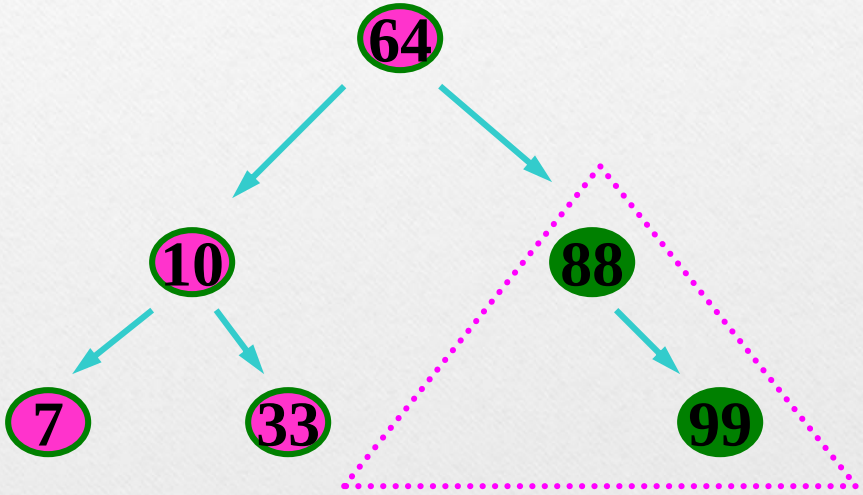
64 10 7 33

Traverse BST



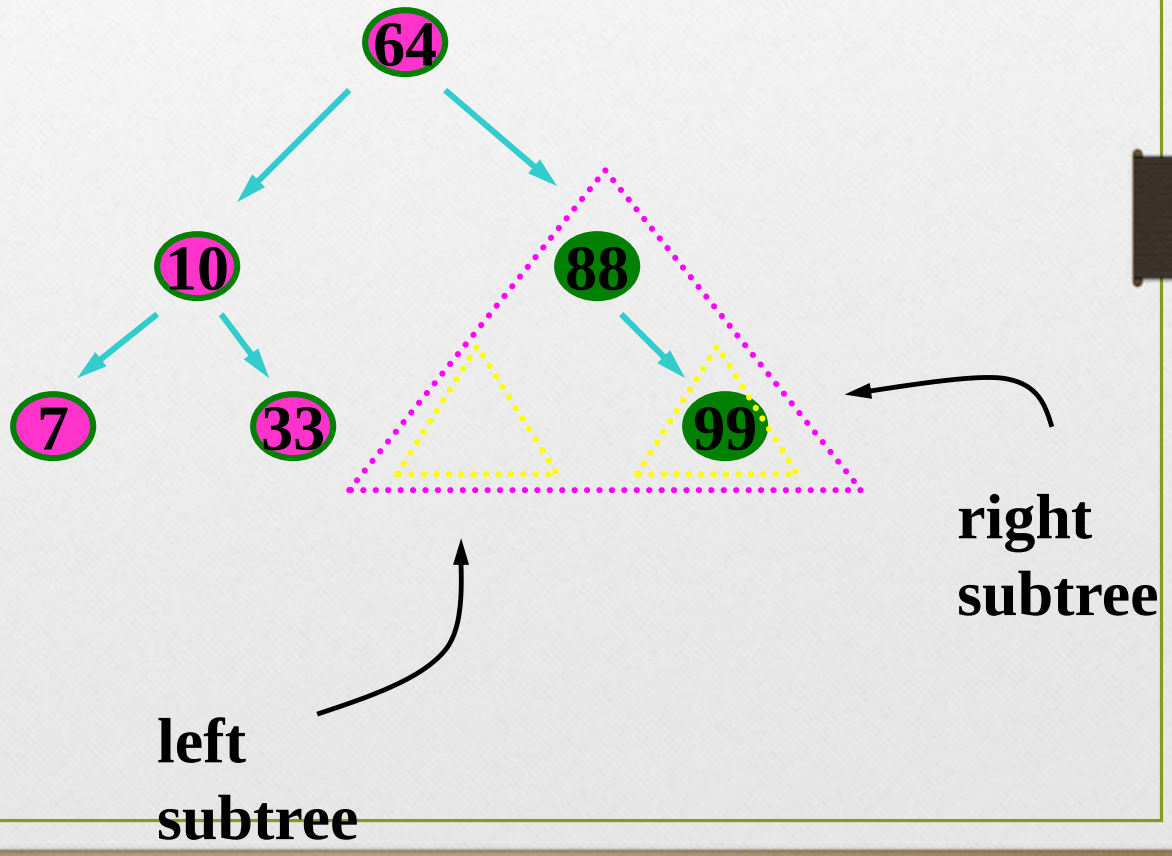
64 10 7 33

Traverse BST



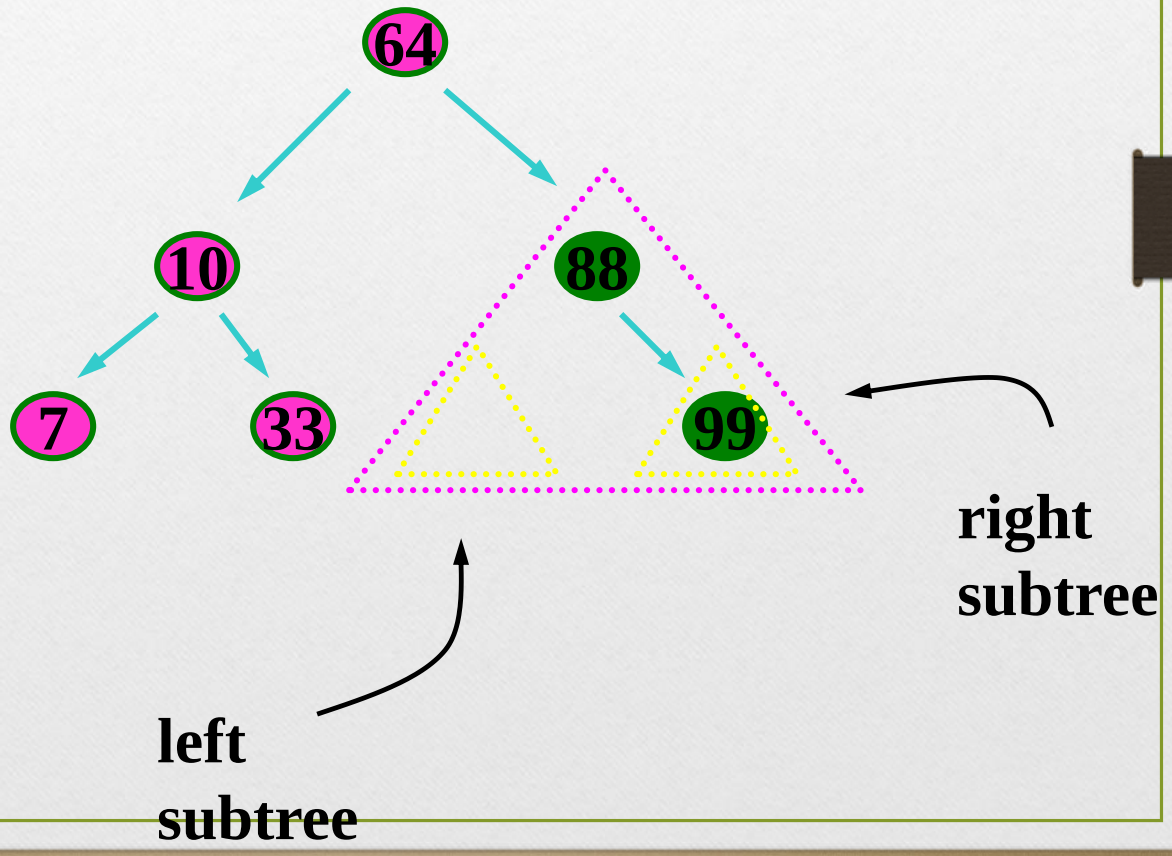
64 10 7 33

Traverse BST



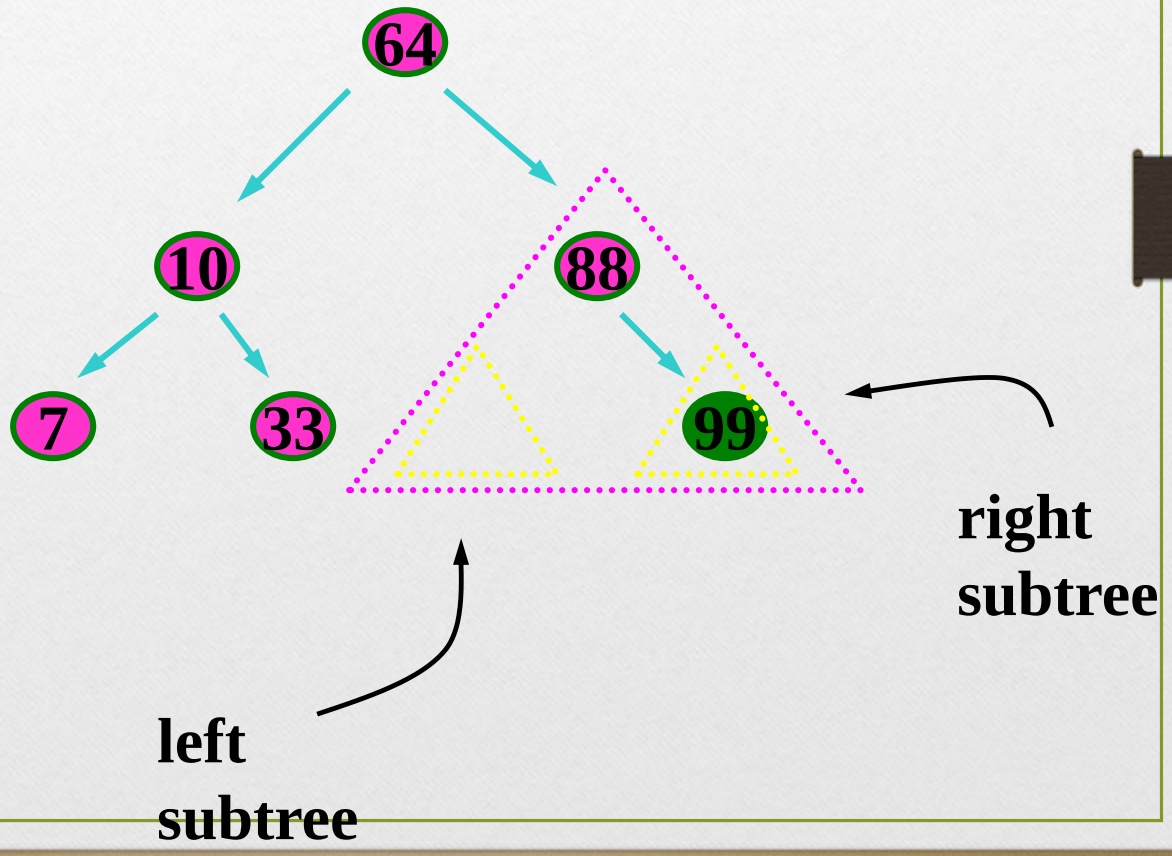
64 10 7 33 88

Traverse BST



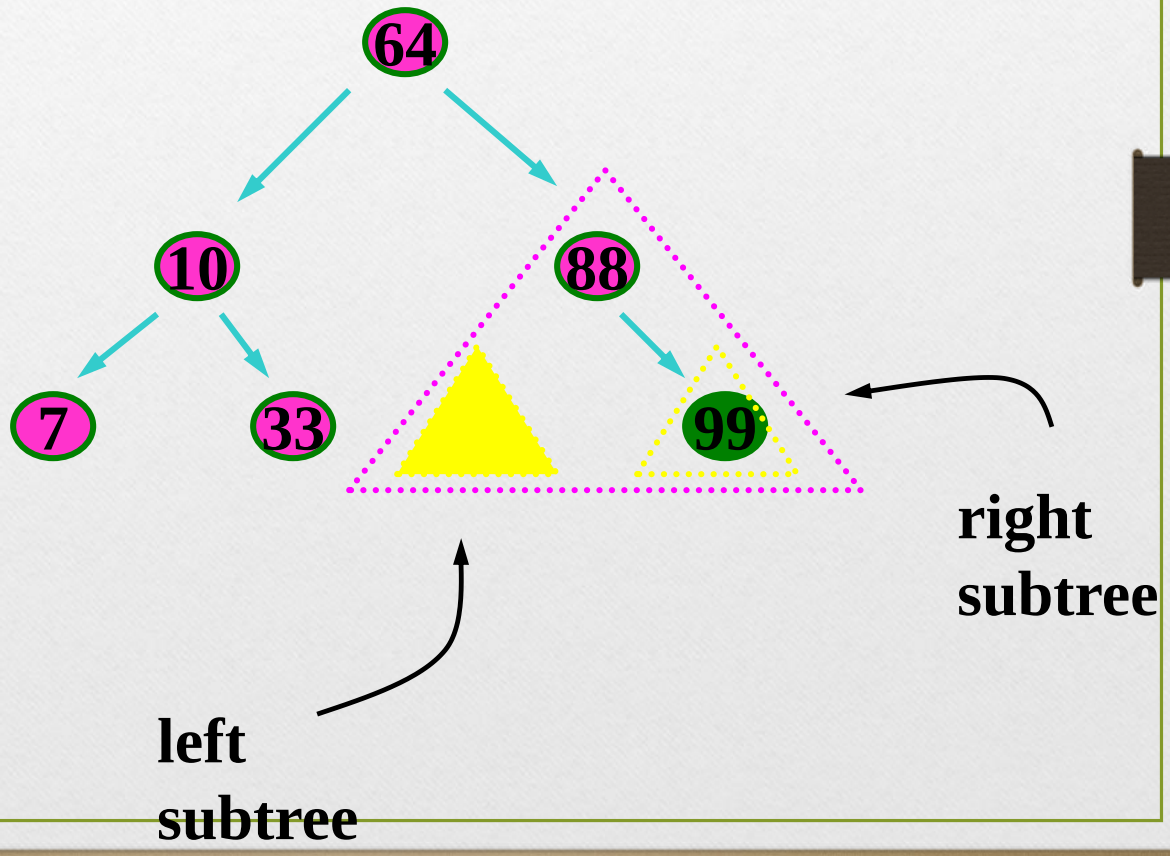
64 10 7 33 88

Traverse BST



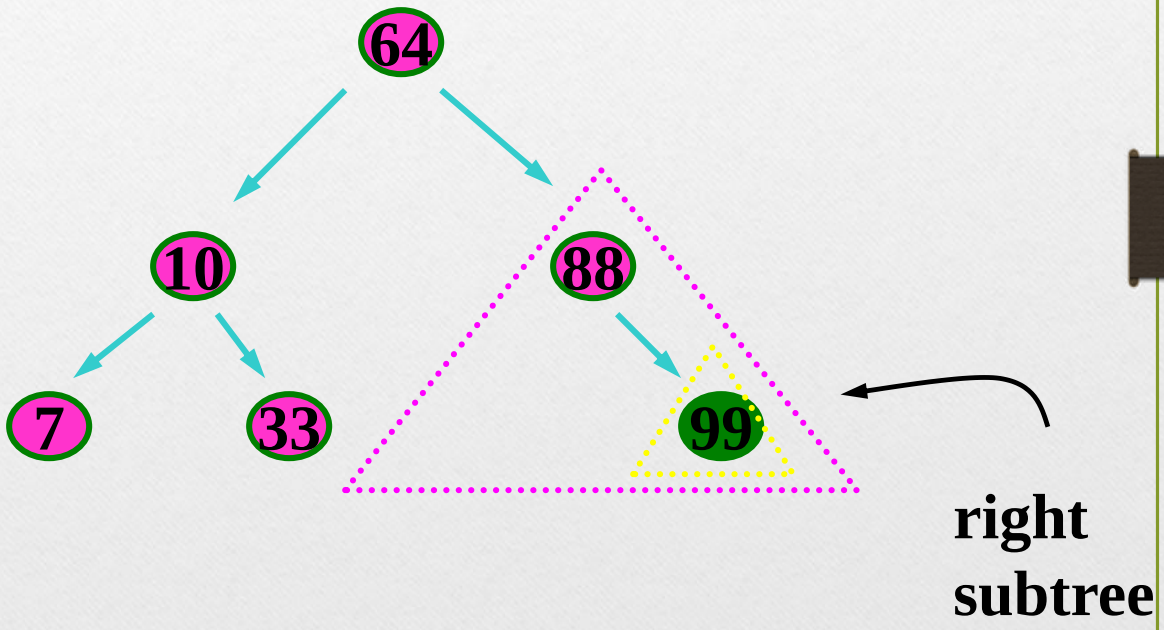
64 10 7 33 88

Traverse BST



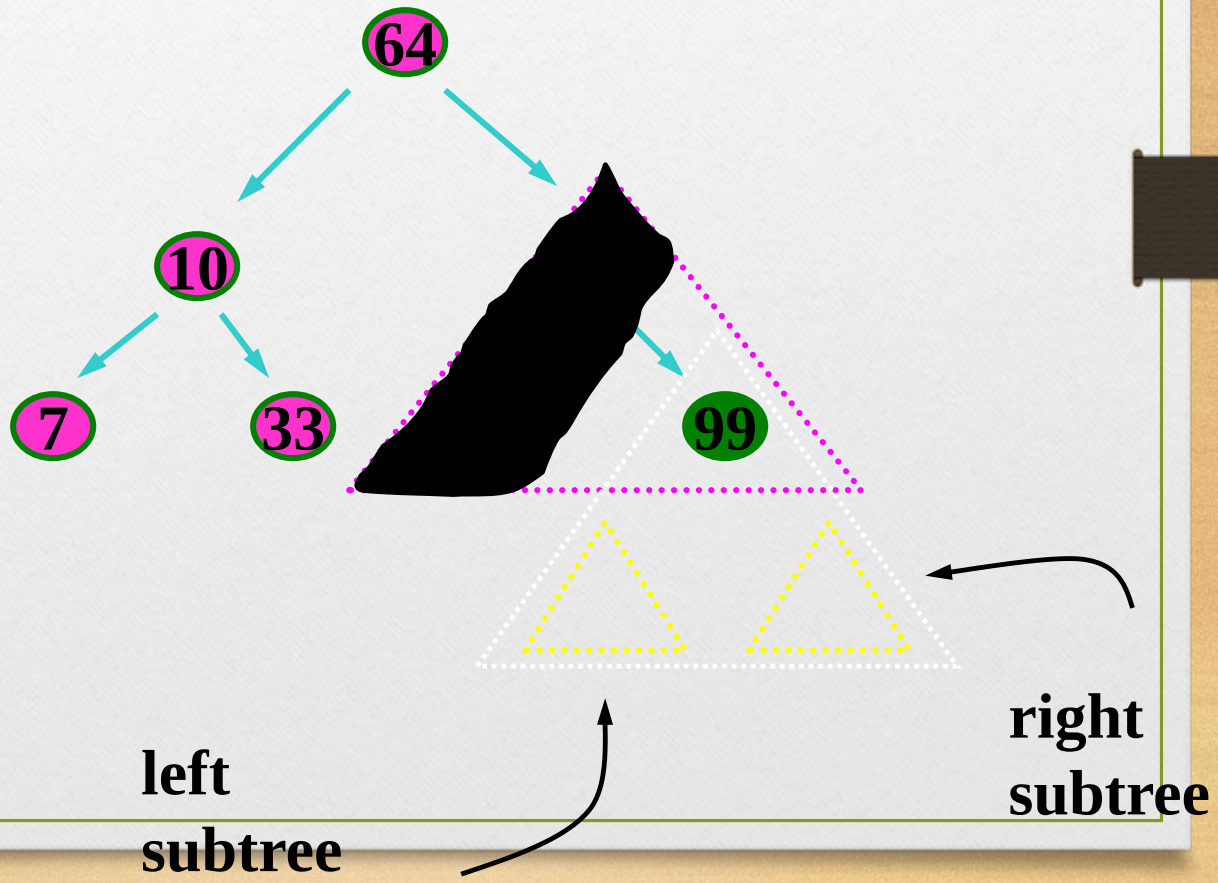
64 10 7 33 88

Traverse BST



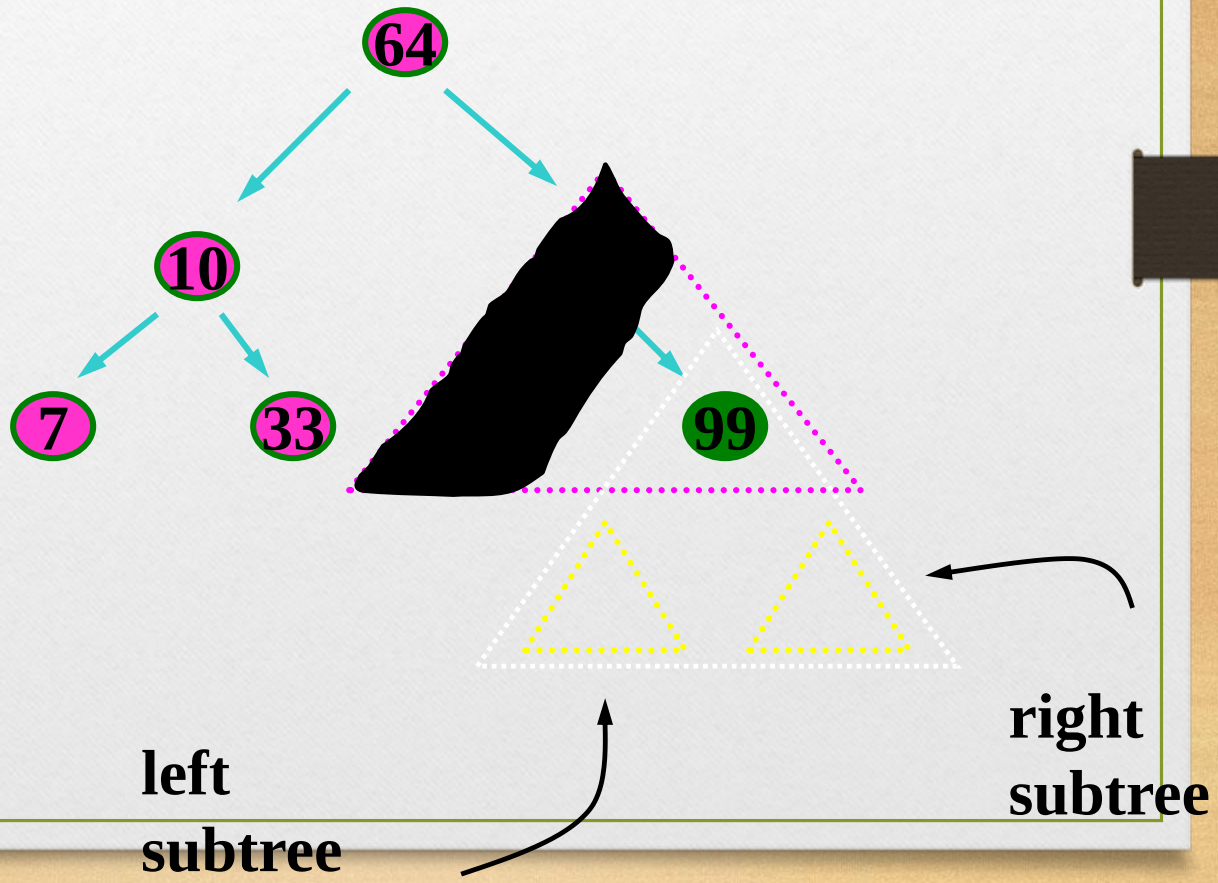
64 10 7 33 88

Traverse BST



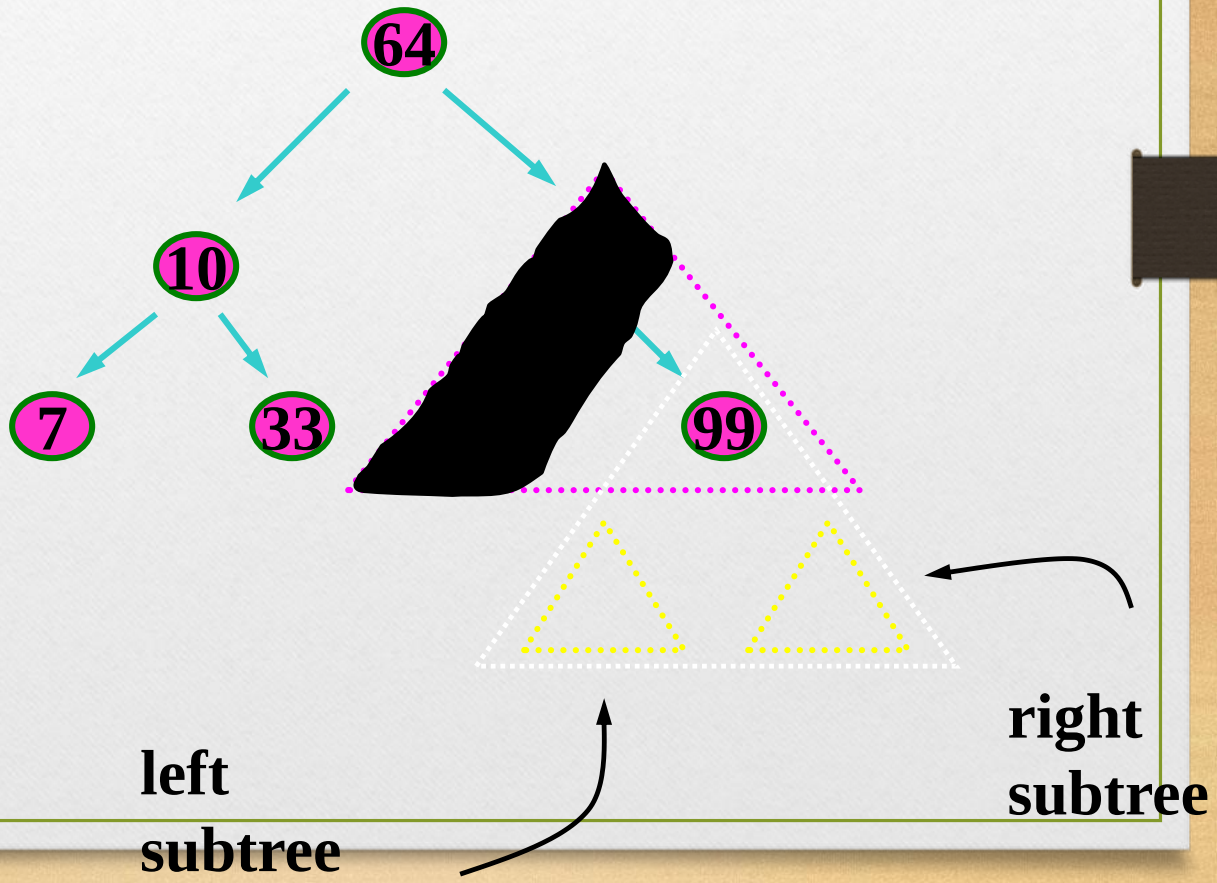
64 10 7 33 88 99

Traverse BST



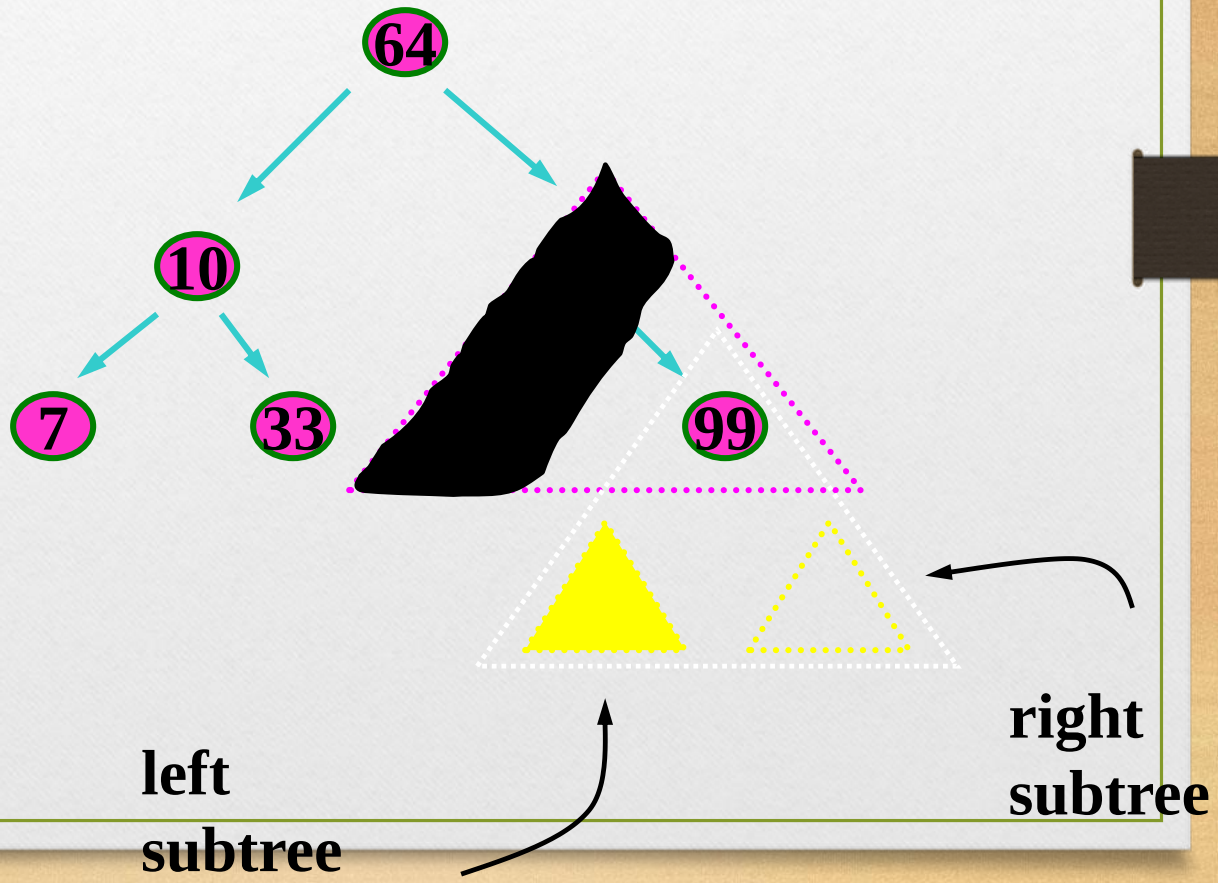
64 10 7 33 88 99

Traverse BST



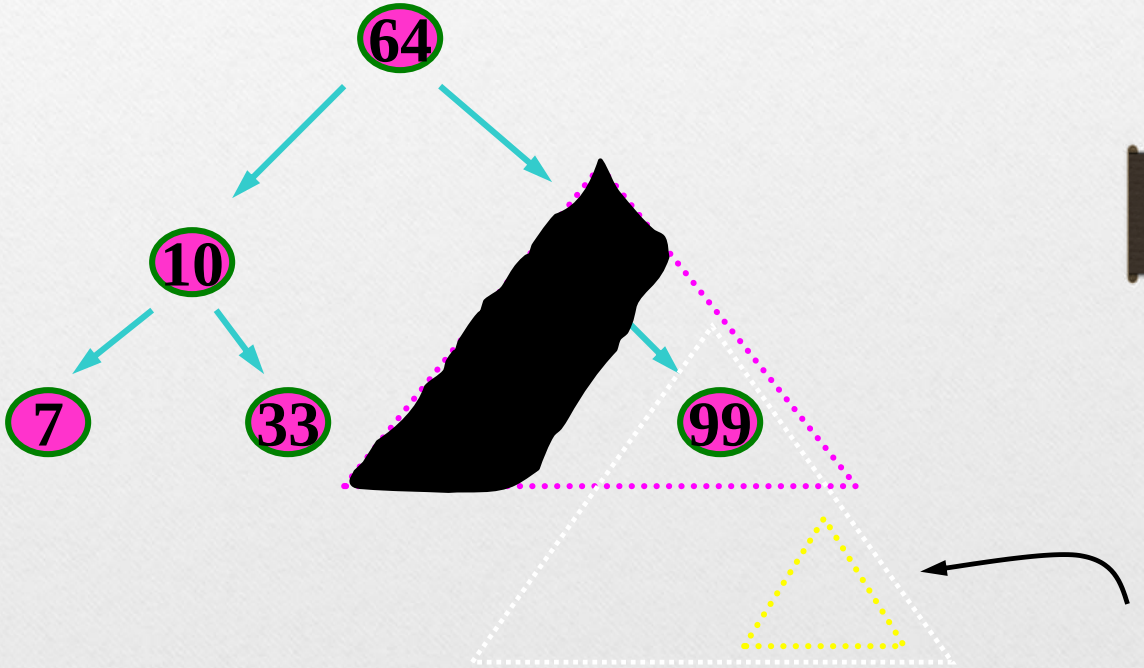
64 10 7 33 88 99

Traverse BST



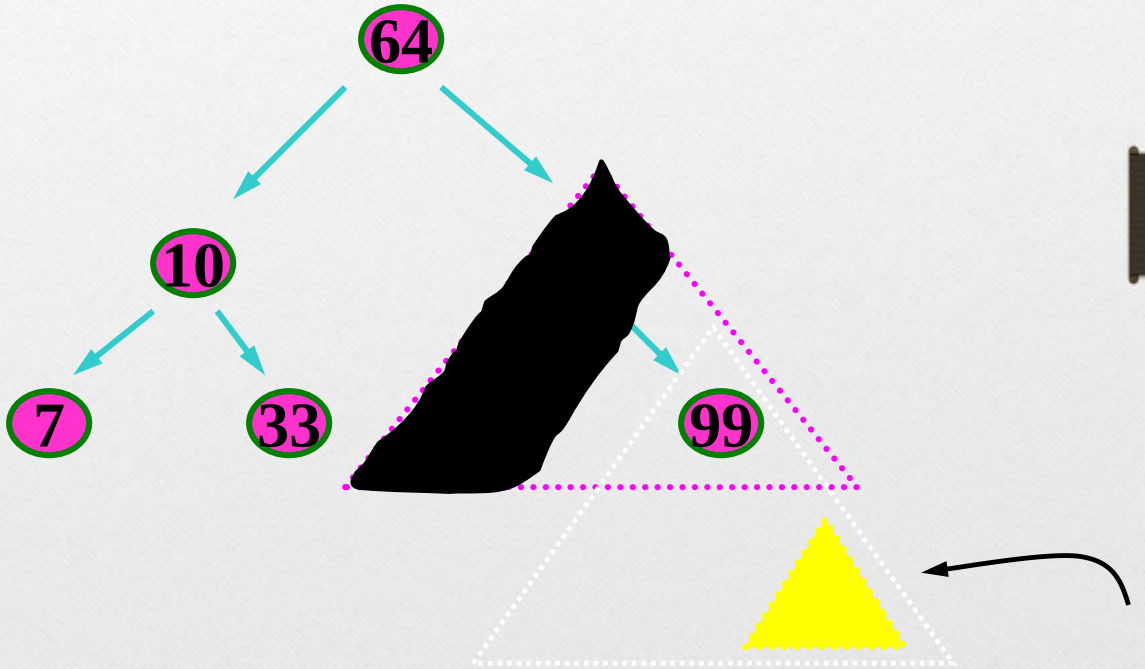
64 10 7 33 88 99

Traverse BST



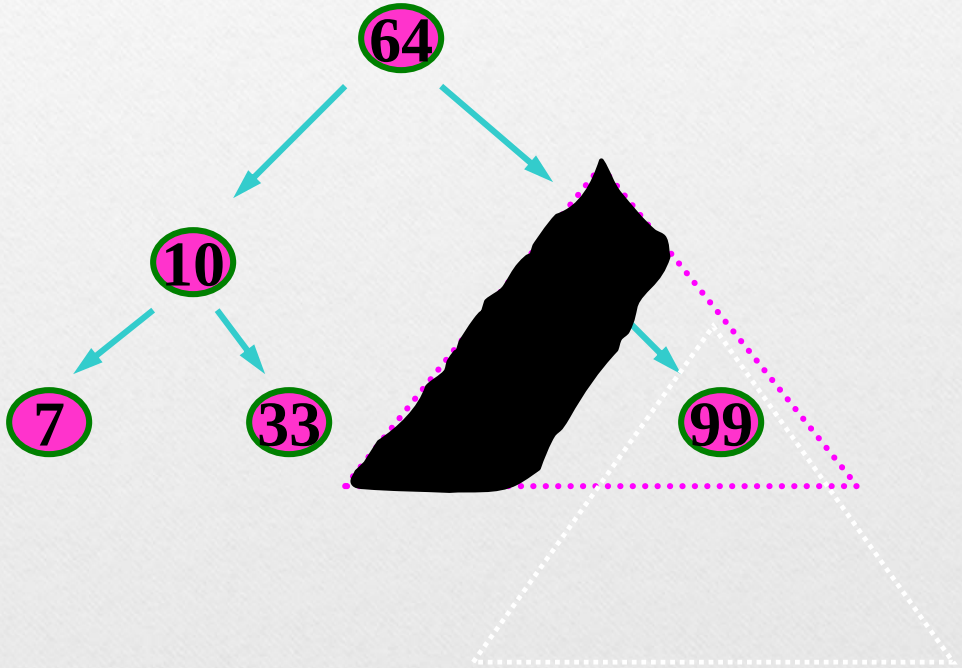
64 10 7 33 88 99

Traverse BST



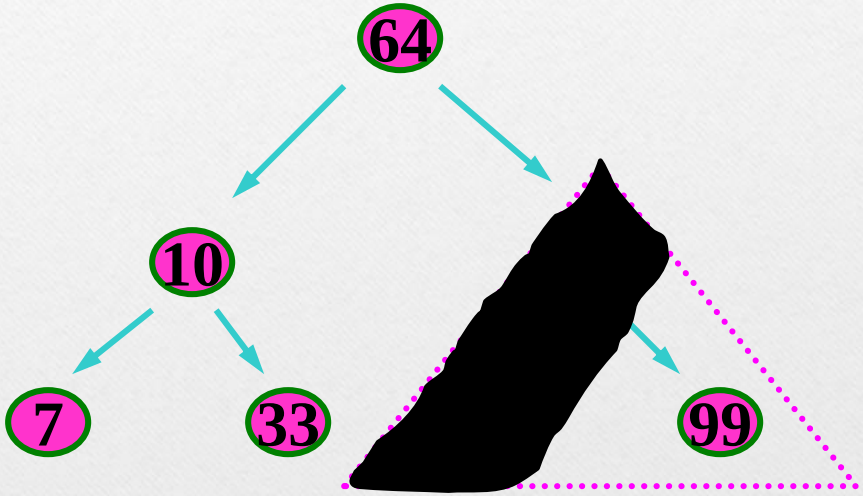
64 10 7 33 88 99

Traverse BST



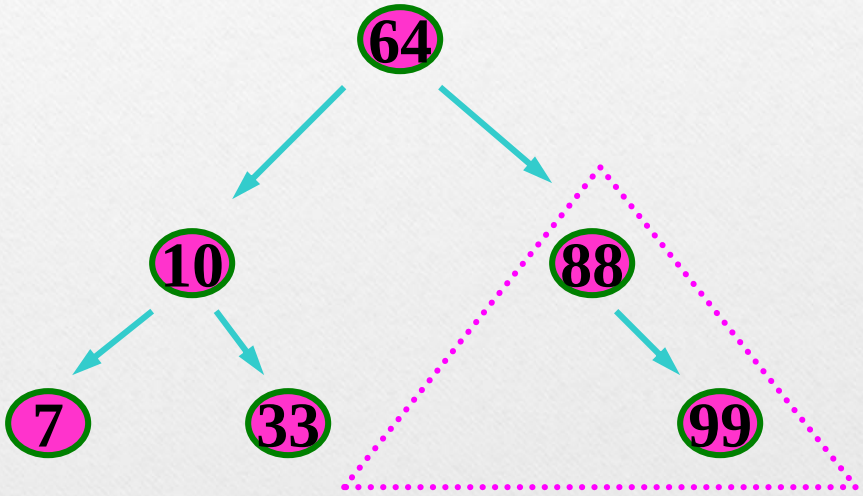
64 10 7 33 88 99

Traverse BST



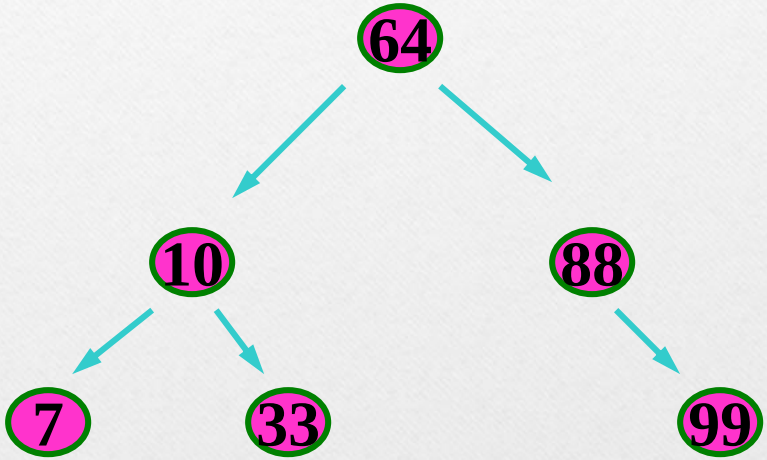
64 10 7 33 88 99

Traverse BST



64 10 7 33 88 99

Traverse BST



Traverse BST

The order of traversal being discussed is as follows:

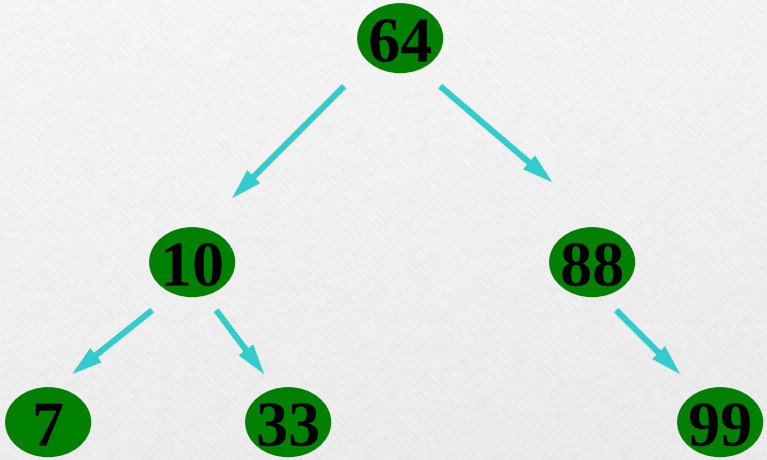
- **N : visit node**
- **L : Traverse left subtree**
- **R : Traverse right subtree**

Traverse BST

```
void Preorder(BST *p2) {

        if (p2) {
                Visit(p2);
                Preorder(p2->Lchild);
                Preorder(p2->Rchild);
        }
}
```

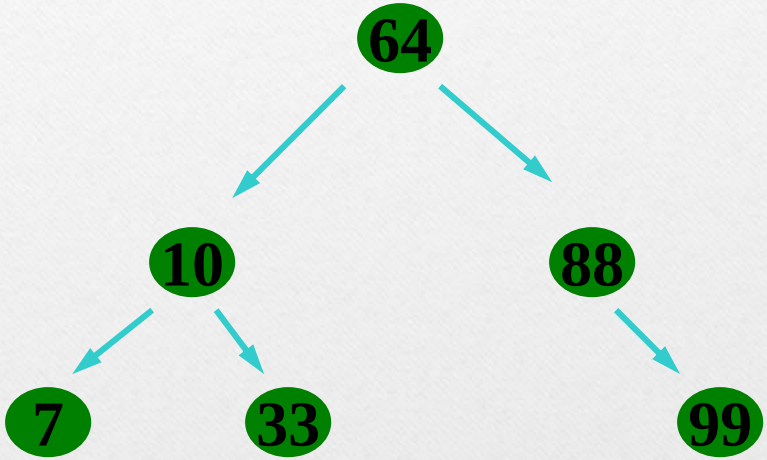
```
void Visit(BST *p2) {
        printf("Lawat %d\n",p->data);

}
```

Traverse BST



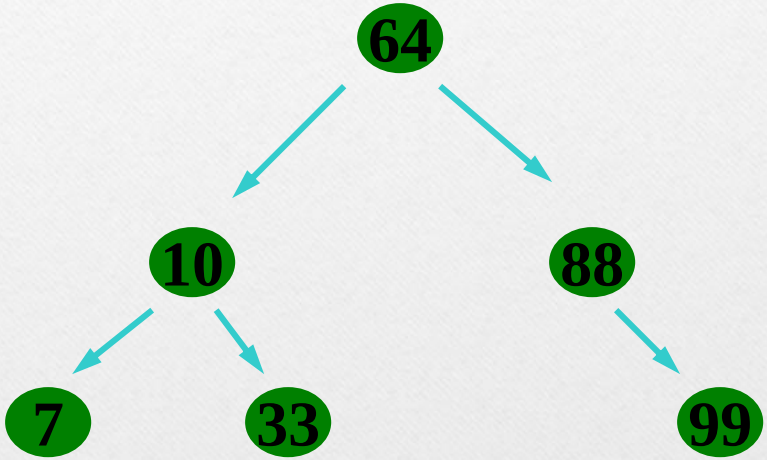
If NLR(Preorder): 64 10 7 33 88 99

Traverse BST



If NLR (Preorder): ???
If LNR (Inorder): ???
If LRN (Postorder): ???

Traverse BST



If NLR (Preorder): ??? = 64 10 7 33 88 99
If LNR (Inorder): ??? = 7 10 33 64 88 99
If LRN (Postorder): ??? = 7 33 10 99 88 64