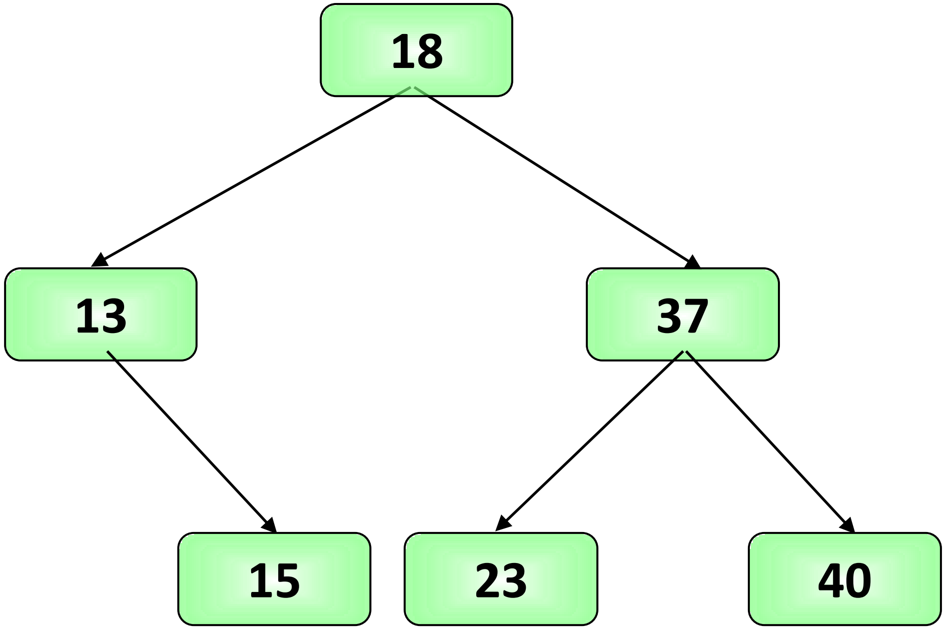
BINARY SEARCH TREE

Definition

- **Binary Search Tree** can be defined as a class of binary trees, in which the nodes are arranged in a specific order:
  - value of all the nodes in the left sub-tree is less than the value of the root.
  - value of all the nodes in the right sub-tree is greater than (or equal to) the value of the root.
  - The left and right subtree each must also be a binary search tree.



Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists.
   - In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Data structure of a node

```
struct TNode
{
    int     Key;
    TNode *pLeft;
    TNode *pRight;
};

typedef TNode *TREE;
```

Operations on Binary Search Tree

- Initialize an empty BST
- Insert a new node into BST
- Delete a node having the key x
- Search a node having the key x

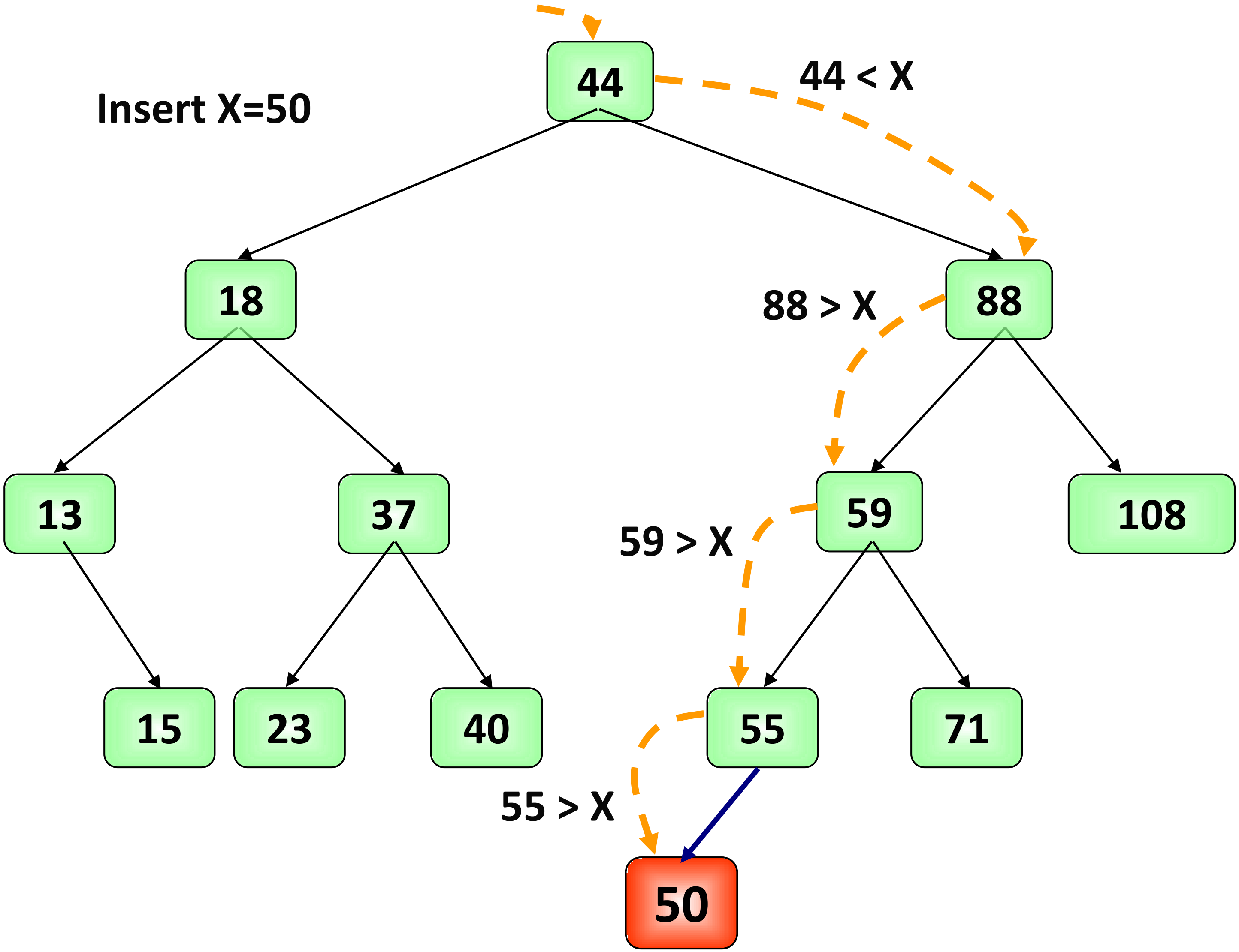
Initialize an empty BST

```
void InitializeTree(TREE &T)
{
    T=NULL;
}
```

Insert a new node to the BST

```
int insertNode(TREE &T, Data X)
{
    if(T)
    {       if(T->Key == X)
                    return 0;
            if(T->Key > X)
                    return insertNode(T->pLeft, X);
            else
                    return insertNode(T->pRight, X);
    }
    T = new TNode;
    T->Key = X;
    T->pLeft =T->pRight = NULL;
    return 1;
}
```

Insertion - example



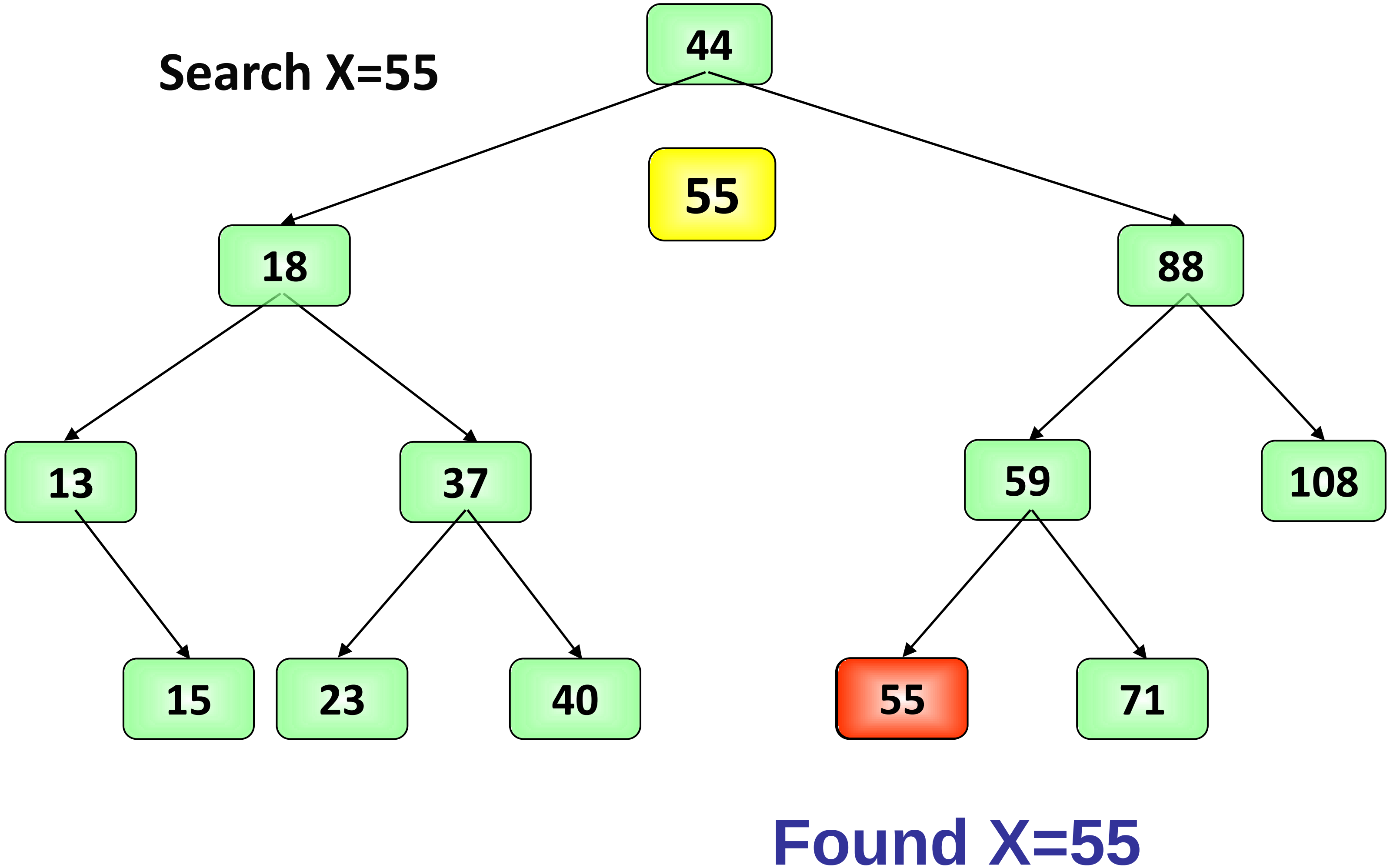
Search (using loop)

```
TNode * searchNode(TREE Root,  Data x)
{       Node *p = Root;
        while (p != NULL)
        {       if(x == p->Key) return p;
                else
                if(x < p->Key)   p = p->pLeft;
                else  p = p->pRight;
        }
        return NULL;
}
```

Search (using recursion)

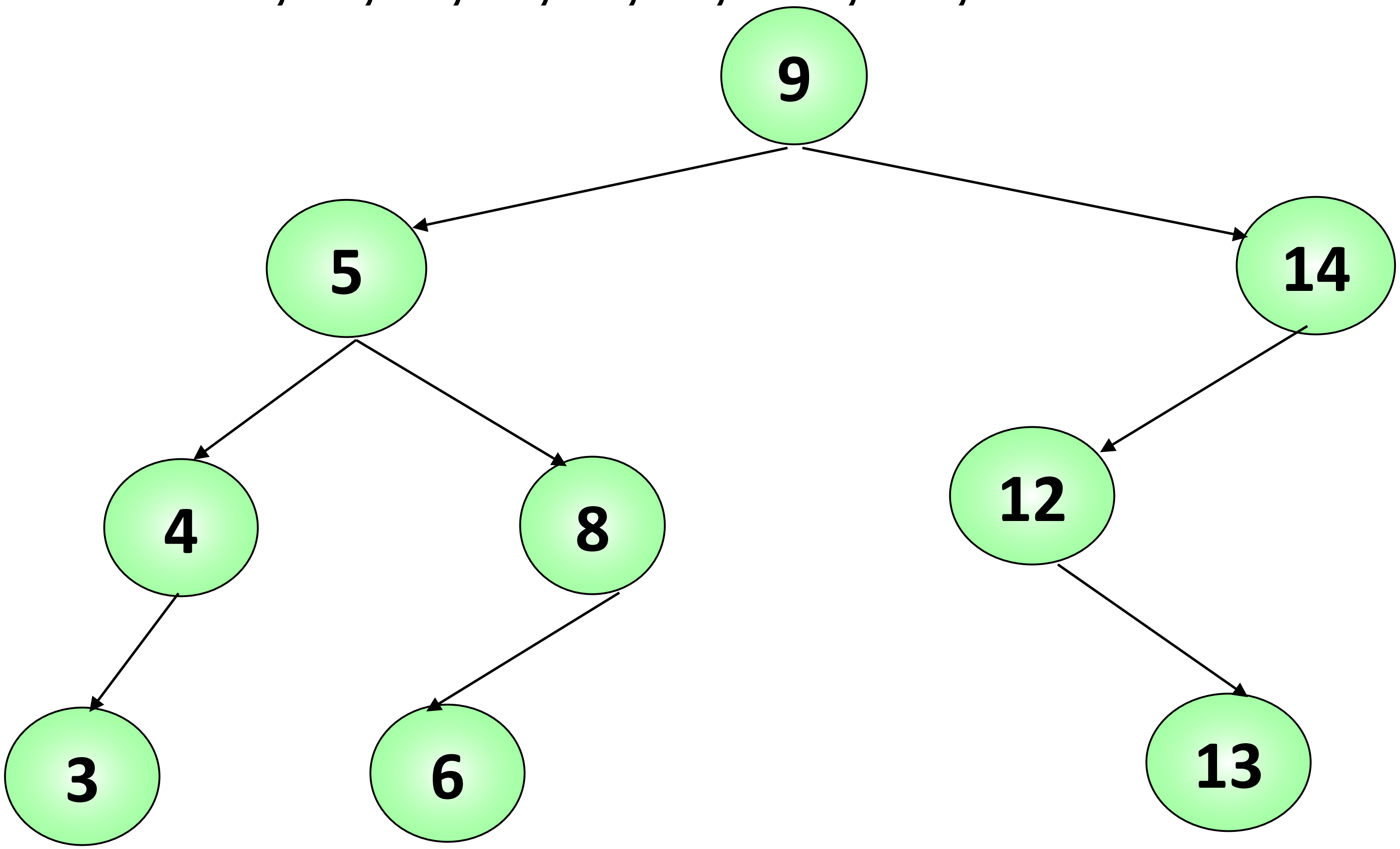
```
TNode *SearchTNode(TREE T, int x)
{
    if(T!=NULL)
    {
        if(T->key==x)
            return T;
        else
            if(x>T->key)
                return SearchTNode(T->pRight,x);
            else
                return SearchTNode(T->pLeft,x);
    }
    return NULL;
}
```

Search - example



Create a BST from a list of numbers

9, 5, 4, 8, 6, 3, 14,12,13

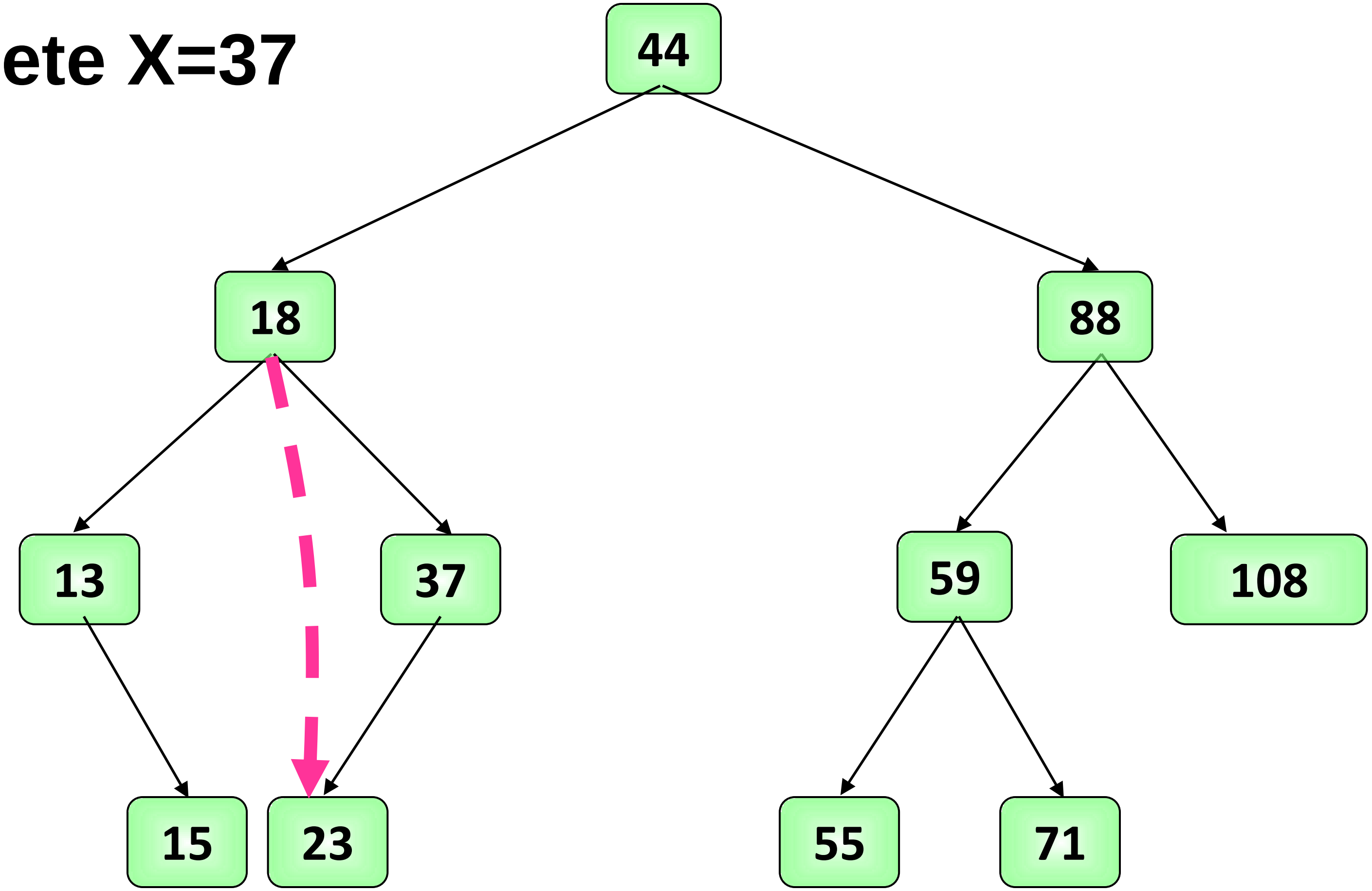


Delete a node having key x

- Condition: Keep the BST attributes after the deletion
- Three types of nodes need to be deleted
  1. X is a leaf node (degree 0)
  2. X has one child (degree 1)
  3. X has two children (degree 2)
- Type 1: Delete X without considering other nodes
- Type 2: Link parent node of X to the unique child node of X before delete X.
- Type 3: Find the successor node

Delete a node with one child - example

Delete X=37

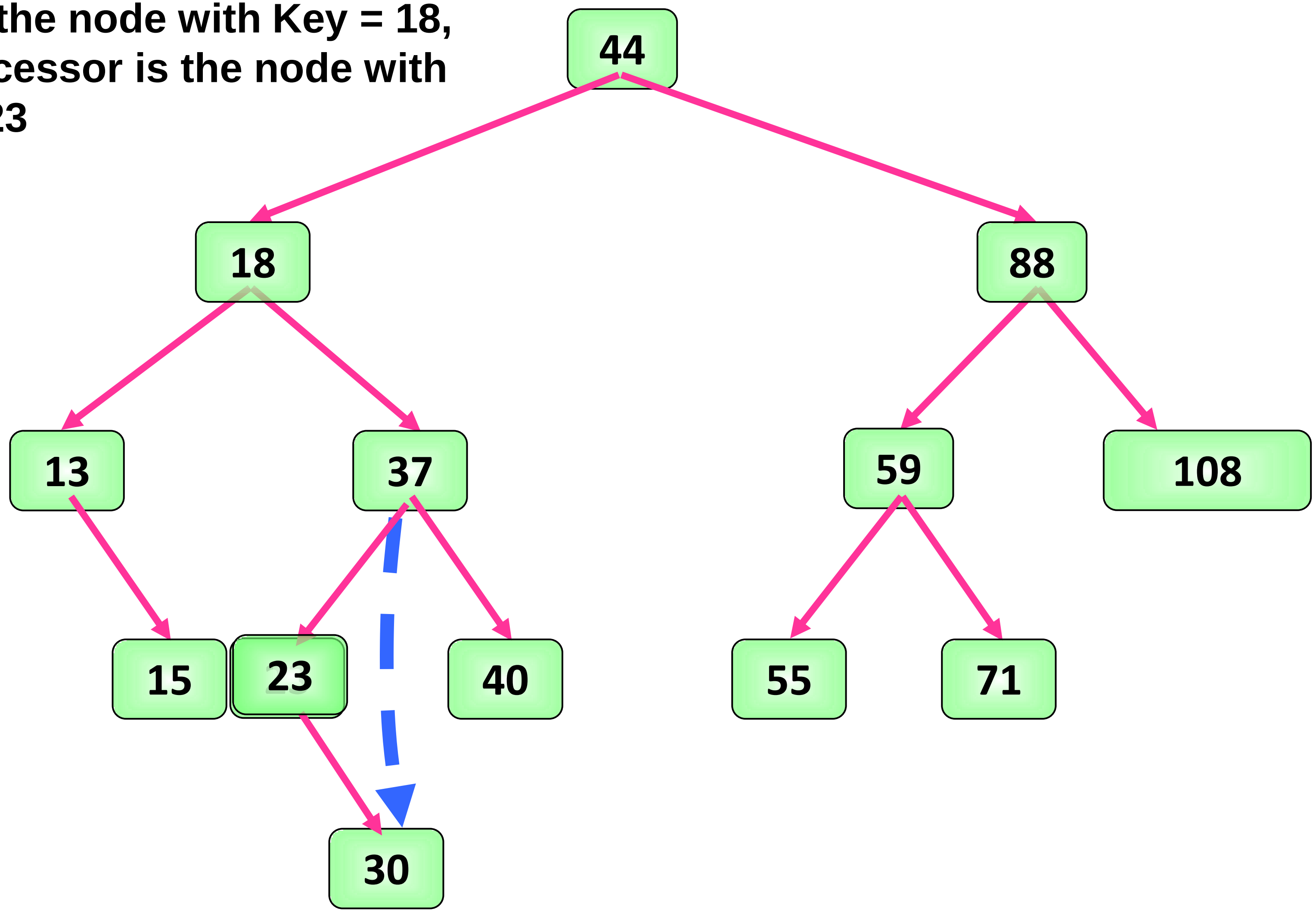


Delete a node with two children

- First, we find the deletion node p (= the node that we want to delete)
- Find the successor node of p.
- Replace the content of node p with the content of the successor node.
- Delete the successor node.

Delete a node with two children

Delete the node with Key = 18, its successor is the node with Key = 23



Delete node with Key = x

```
void DeleteNodeX1(TREE &T,int x){
    if(T!=NULL){
        if(T->Key<x)        DeleteNodeX1(T->Right,x);
        else{
            if(T->Key>x)        DeleteNodeX1(T->Left,x);
            else  //Found the node with key = x
            {           TNode *p;
                        p=T;
                        if (T->Left==NULL)           T = T->Right;
                        else{
                            if(T->Right==NULL)           T=T->Left;
                            else FindSuccessor(p, T->Right);// search on the
    right branch
                        }
                        delete p;
            }
        }
    }
    else   cout << ”Cannot find the node with the key x";
}
```

Find successor

```
void FindSuccessor(TREE &p, TREE &T)
{
    if(T->Left!=NULL)
            FindSuccessor(p,T->Left);
     else
     {
            p->Key = T->Key;
            p=T;
            T=T->Right;
    }
}
```