Lecture 28–30

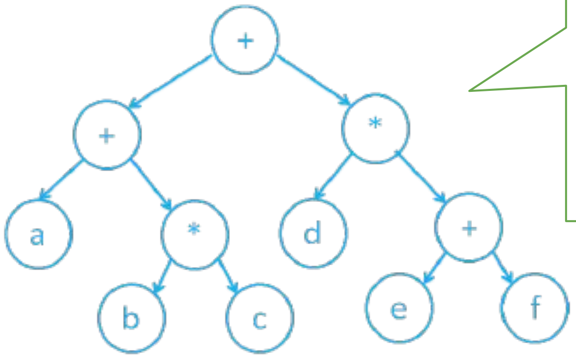
- Expression Trees, Threaded Binary Trees, and Binary Search Trees

IT205: Data Structures (AY 2023/24 Sem II Sec B) — **Dr. Arpit Rana**

Expression Tree

A **Strictly Binary Tree** can be used to represent an expression containing *operands* and *binary operators*.

- The root of this tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees.
- A node representing an operator is a non-leaf, while a node representing an operand is a leaf.

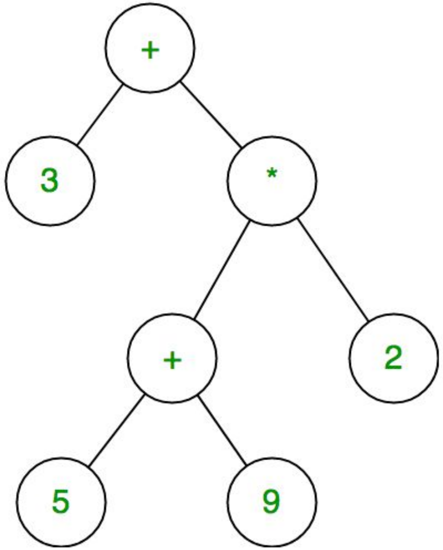


The expression tree is immutable, and once built, we cannot change or modify it further, so to make any changes, we must completely construct the new expression tree.

a + (b * c) + d * (e + f)

Expression Tree Traversals

- The pre-order, postorder, and inorder traversal of the expression tree yield prefix, postfix, and infix expressions.
- Binary expression tree does not contain parentheses since the ordering of the operations are implied by the structure of the tree.
  - the value present at the depth of the tree has the highest priority
- Thus, an expression whose infix form requires parentheses to override explicitly the conventional parentheses rules cannot be retrieved by a simple inorder traversal.



3 + ((5 + 9) * 2) = 31

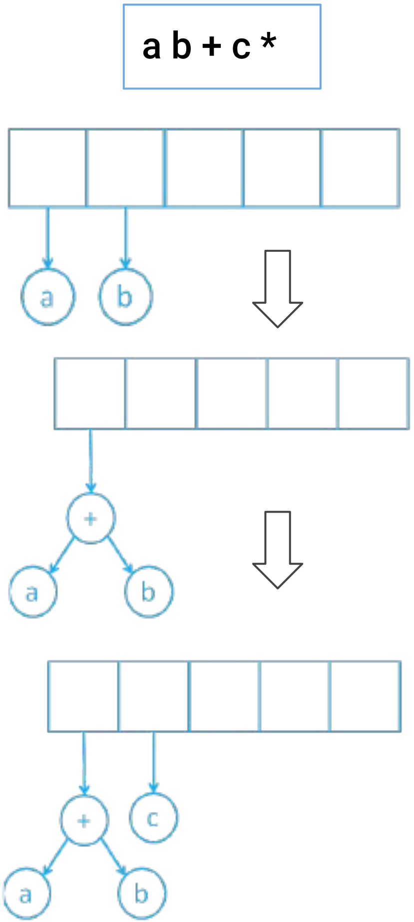
Expression tree maintain precedence in term of depth
in expression tree last part of tree means at depth d is evaluated first .

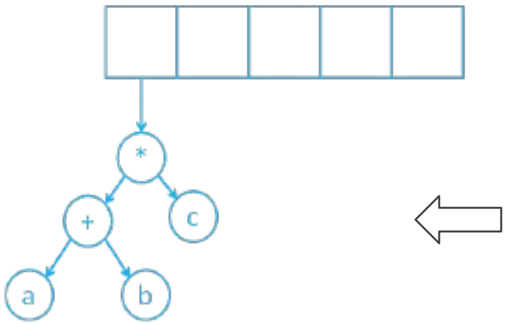
Construction of an Expression Tree

A stack is used to build an expression tree. For each character, we cycle through the input expressions and do the following.

- If a character is an operand, add it to the stack.
- If a character is an operator, pop both values from the stack and make both its children and push the current node again.
- Finally, the stack's lone element will be the root of an expression tree.

ex1.cpp





Source: https://www.prepbytes.com/blog/tree/expression-tree-in-data-structure/

Threaded Binary Tree: Motivation

Recursive and non-recursive algorithms for the in-order traversals: See the example here

Recursive In-order Traversal

```
void intrav(tree){
  If (tree != NULL)
     intrav(left(tree))
     print(data(tree))
     intrav(right(tree))
  EndIf
}
```

Non-Recursive In-order Traversal

```
void intrav2(tree){
     s.top = -1    // s is an empty stack
     p = tree
     do {
          while (p != NULL) {
               push(s, p)
               p = left(p)
          }
          If (!empty(s)) {
               p = pop(s)
               print(data(p))
               p = right(p)
          }
     }while(!empty(s) || p != NULL)
}
```

Threaded Binary Tree: Motivation

The non-recursive inorder traversal algorithm is much slower than the recursive algorithm. Because –

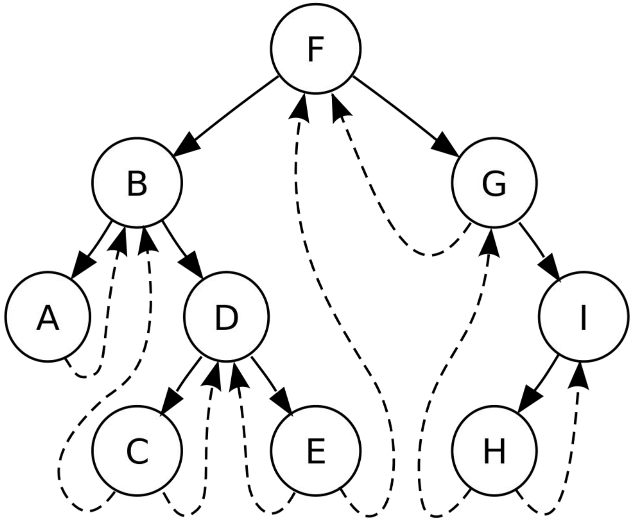
- the calls to push, pop, and empty that involves the superfluous tests for overflow and underflow.
- automatic stacking and unstacking of built-in recursion (implemented through registers) is more efficient than the programmed version.

Threaded Binary Tree helps us traverse the binary tree without using a stack in a non-recursive manner.

Threaded Binary Tree: Definition

A binary tree is threaded by making –

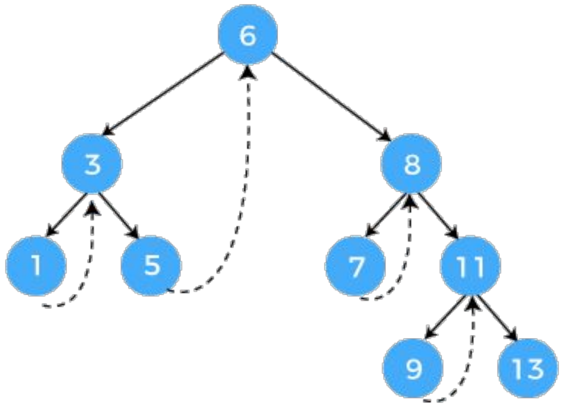
- all **right child pointers** that would normally be null point to the **inorder successor** of the node (if it exists), and
- all **left child pointers** that would normally be null point to the **inorder predecessor** of the node

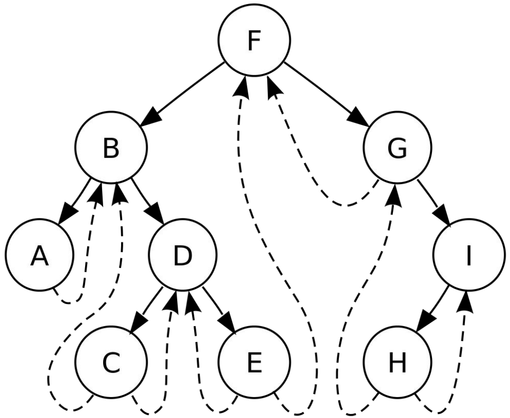


Threaded Binary Tree: Types

A threaded binary tree is of two types –

- **Single threaded**: either right in-threaded or left in-threaded
- **Double threaded**: both left and right in-threaded

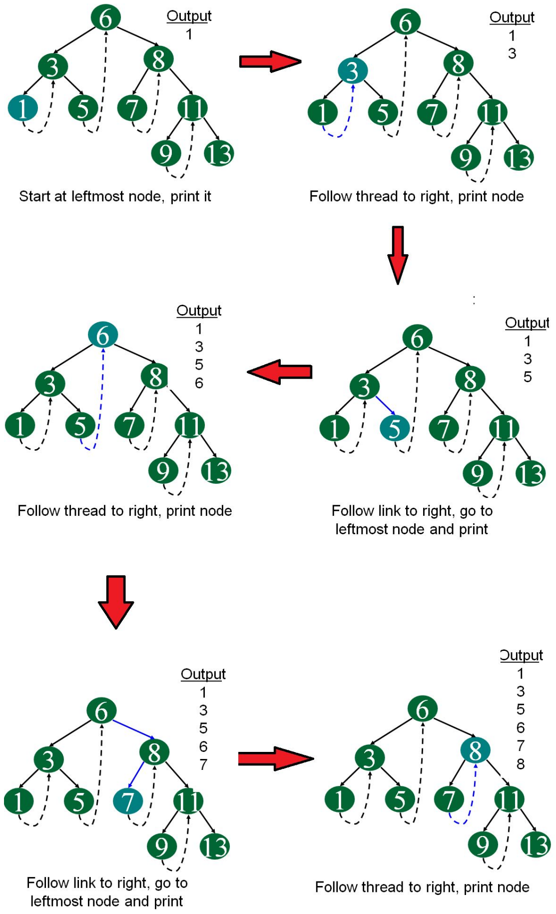




Threaded Binary Tree Traversal

Inorder traversal using threads

Operation	Time Complexity	Space Complexity
Insertion	O(log n)	O(1)
Deletion	O(log n)	O(1)
Search	O(log n)	O(1)
In-order Traversal	O(n)	O(1)
In-order Successor/ Predecessor	O(1)	O(1)

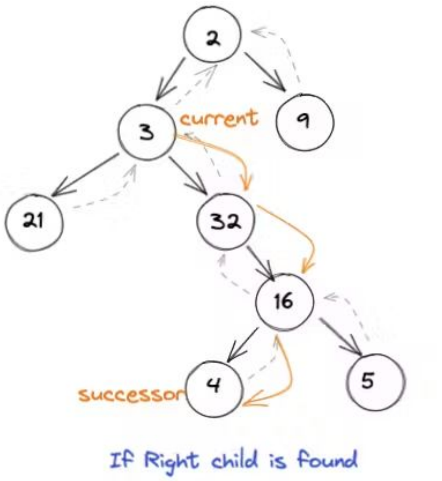


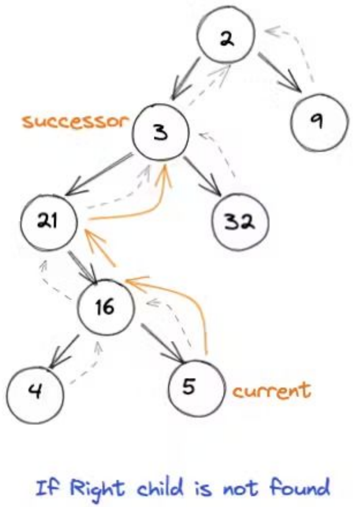
continue same way for remaining node.....

Inorder Traversal with Parent (*father*) Pointer

If each tree node contains a ***father*** field, neither a stack nor threads are necessary for non-recursive traversal.

- Instead, when the traversal process reaches a leaf node, father field can be used to climb back up the tree.

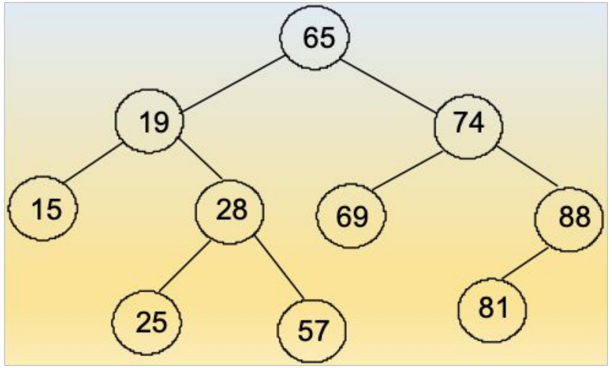


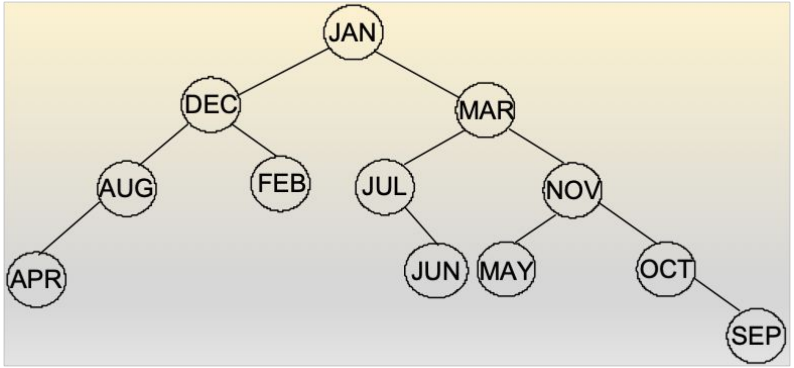


Binary Search Tree

A binary tree T is termed as ***binary search tree*** (or ***binary sorted tree***) if each node ***nd*** of T satisfies the following property:

- All elements in the left subtree of a node ***nd*** are less than the contents of ***nd***, and
- all elements in the right subtree of ***nd*** are greater than or equal to the contents of ***nd***.





This is useful for searching because in this we need less comparison and less time.

->if we travers in inorder then we get sorted order in output.

->mostly linked list is used for representation of bst.

->for searching it take O(logn) in most of case

Binary Search Tree: Operations

The following operations are defined on a Binary Search Tree –

- **Traversal**: Traversing the tree
- **Search**: Searching for an element
- **Insertion**: Inserting an element into it
- **Deletion**: Deleting an element from it

->for searching we use tail recursion in that we don't required stack for iterative definition of searching
Tail recursion is defined as a recursive function in which the recursive call is the last statement is executed by the fun . so basically nothing is left to execute after the recursion call.

Traversal in a Binary Search Tree

- All the traversal operations for binary tree are applicable to binary search trees without any alteration.

- It can be verified that *inorder traversal on a binary search tree will give the sorted order of data in ascending order.*

    - If we require to sort a set of data, a binary search tree can be built with those data and then inorder traversal can be applied.

    - This method of sorting is known as ***binary sort*** and this is why binary search tree is also termed as binary sorted tree.

    - This sorting method is considered as one of the efficient sorting methods.

Search in a Binary Search Tree

- Searching an element in a binary search tree is much faster than in arrays or linked lists.

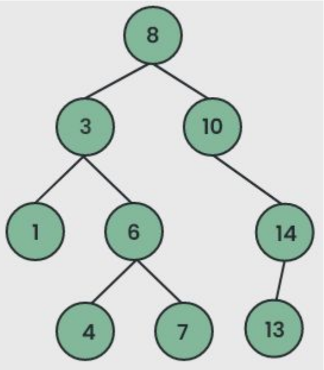
- In the applications where frequent searching operations are to be performed, BST is used to store data.

```
search(root)
    If root == NULL
        return NULL;
    If key == data(root)
        return root;
    If key < data(root)
        return search(left(root))
    If key > data(root)
        return search(right(root))
```

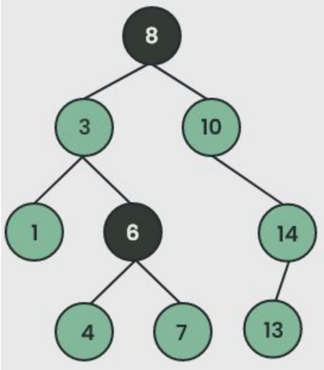
bst1.cpp
for recursive and iterative both

Search in a Binary Search Tree

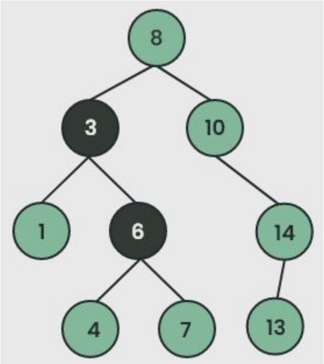
- Searching an element in a binary search tree is much faster than in arrays or linked lists.
- In the applications where frequent searching operations are to be performed, BST is used to store data.



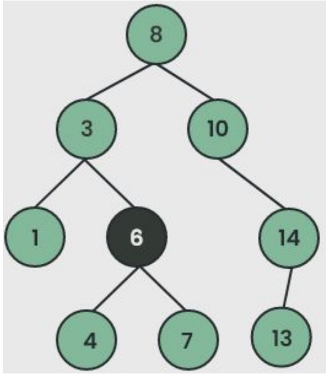
Search for a **key** = 6



As 6 < 8, search in left subtree



As 6 > 3, search in right subtree

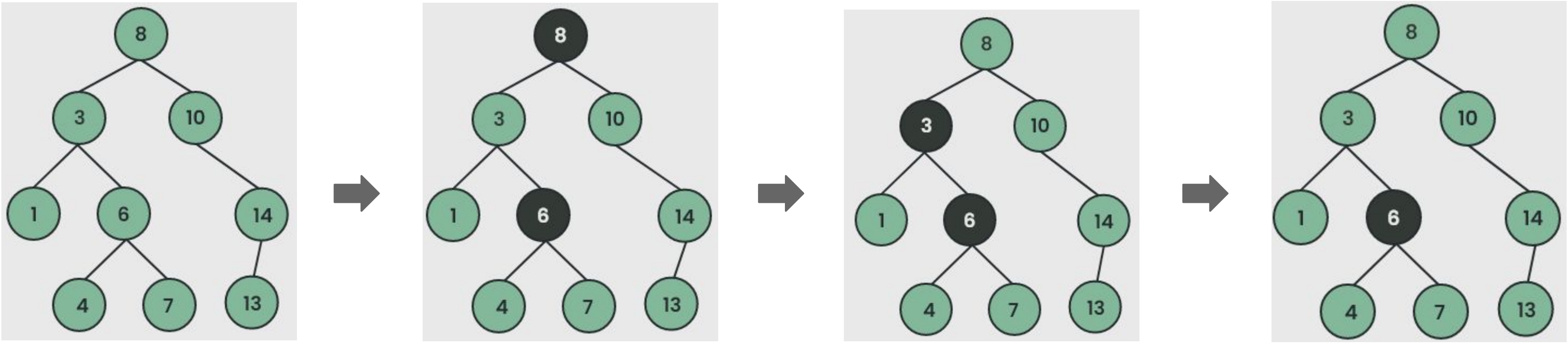


As 6 == key, searching ends.

Search in a Binary Search Tree

- **Time complexity**: O(d), where d is the depth of the BST. or O(h) where h is height

- Auxiliary Space: O(d), where d is the depth of the BST. This is because the maximum amount of space needed to store the recursion stack would be d.



Search for a **key** = 6

As **6 < 8,** search in left subtree

As **6 > 3,** search in right subtree

As **6 == key,** searching ends.

Insertion into a Binary Search Tree

Insertion operation on a binary search tree is conceptually very simple.

- It is in fact, one step more than the searching operation.
- To insert a node with data, say ITEM, into a tree, the tree is to be searched starting from the root node.
- If ITEM is found, do nothing; otherwise, ITEM is to be inserted at the dead end where search halts.

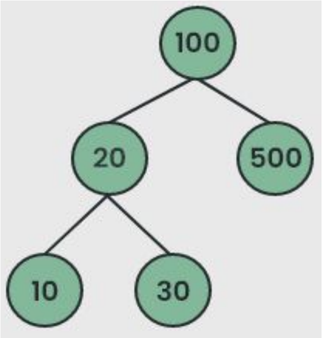
bst2.cpp for recursive definition

```
q = NULL
p = tree
While(p != NULL){
     If (key == k(p))
          return p
     q = p
     If (key < k(p))
          p = left(p)
     Else
          p = right(p)
}
v = maketree(rec, key)
If (q == NULL)
     tree = v
Else
     If (key < k(q))
          left(q) = v
     Else
          right(q) = v
return v
```

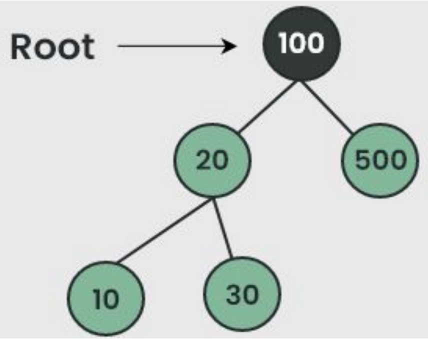
bas1.cpp for iterative method

for inserting n node in binary we should search for each while inserting so searching take O(logn) time and for n nodes it take O(nlogn)

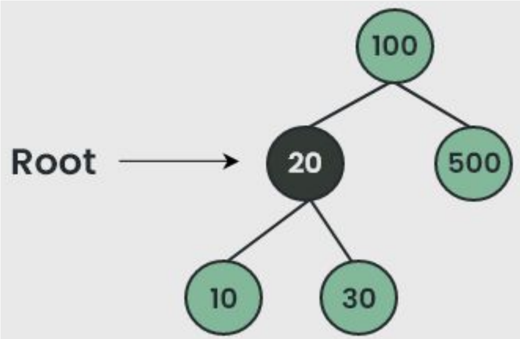
Insertion into a Binary Search Tree



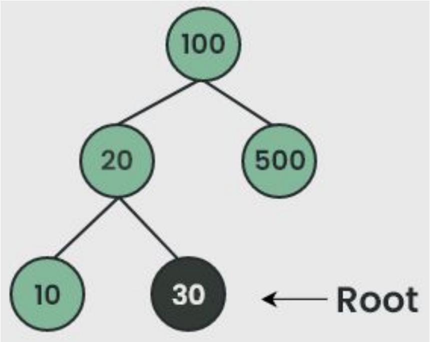
Suppose a **key** = 40 is to be inserted.



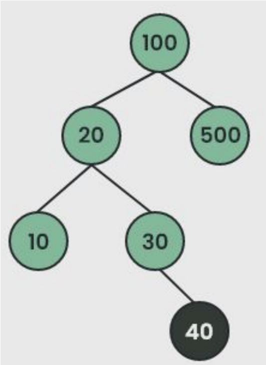
Since **100 > 40,** move to the left subtree.



Since **40 > 20,** move to the right subtree.



Since **40 > 30,** move to the right subtree.



Finally, inserted as the right subtree of 30.

Insertion into a Binary Search Tree

- **Time Complexity:** The worst-case time complexity of insert operation is O(d) where d is the depth of the Binary Search Tree.

  for inserting only one node

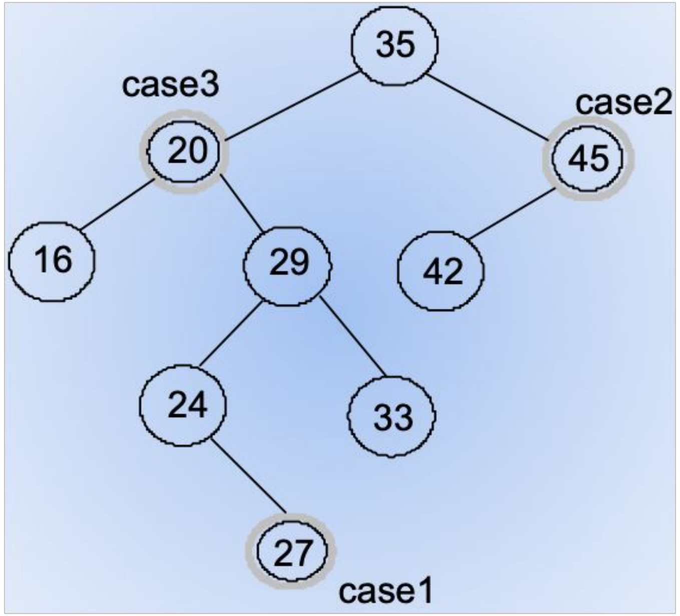
  - In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of insertion operation may become O(n).

- **Space Complexity:** The space complexity of insertion into a binary search tree is O(1) if we apply non-recursive algorithm, O(n) otherwise.

Deletion from a Binary Search Tree

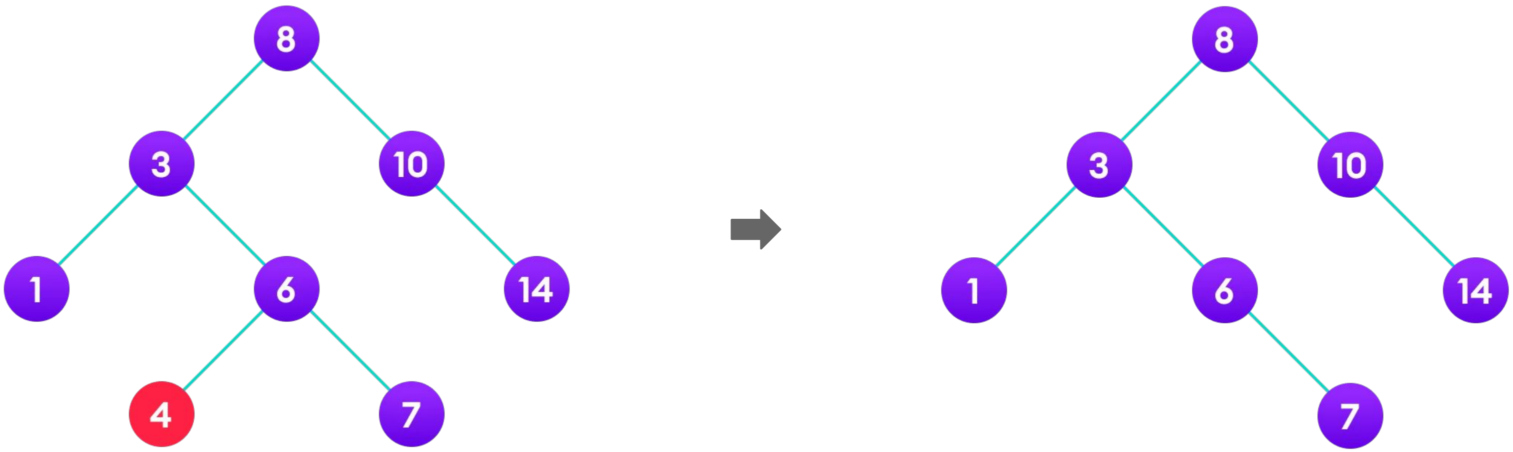
Another frequently used operations on a binary search tree is to delete any node from it. This operation, however, is slightly complicated than the previous two operations discussed.

- Case 1: Node ***nd*** is the leaf node
- Case 2: Node ***nd*** has exactly one child
- Case 3: Node ***nd*** has two children



Deletion from a Binary Search Tree

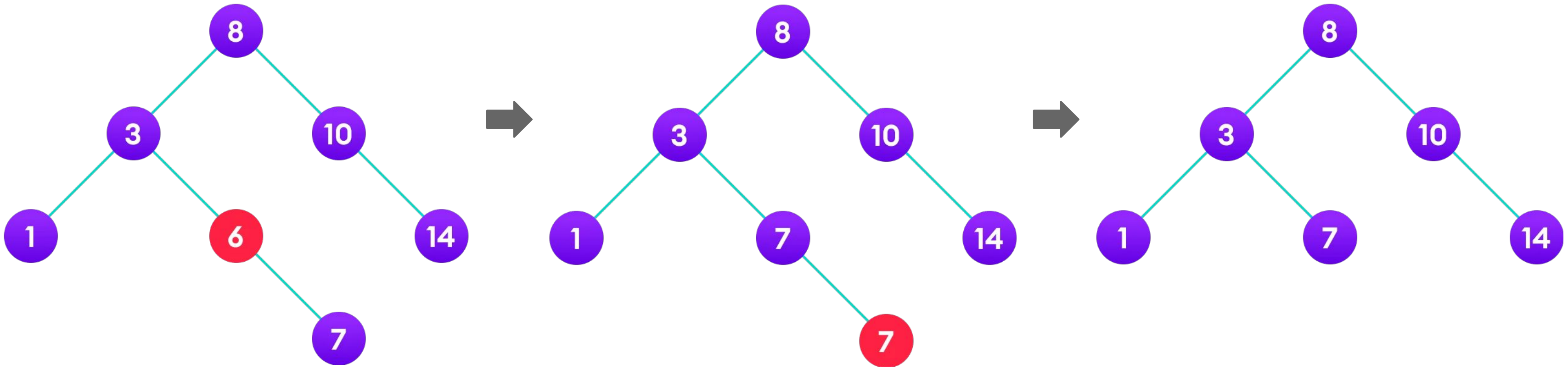
In the first case, the node to be deleted is the leaf node; simply delete the node from the tree.



Deletion from a Binary Search Tree

In the second case, the node to be deleted lies has a single child node.

- Replace that node with its child node.
- Remove the child node from its original position.



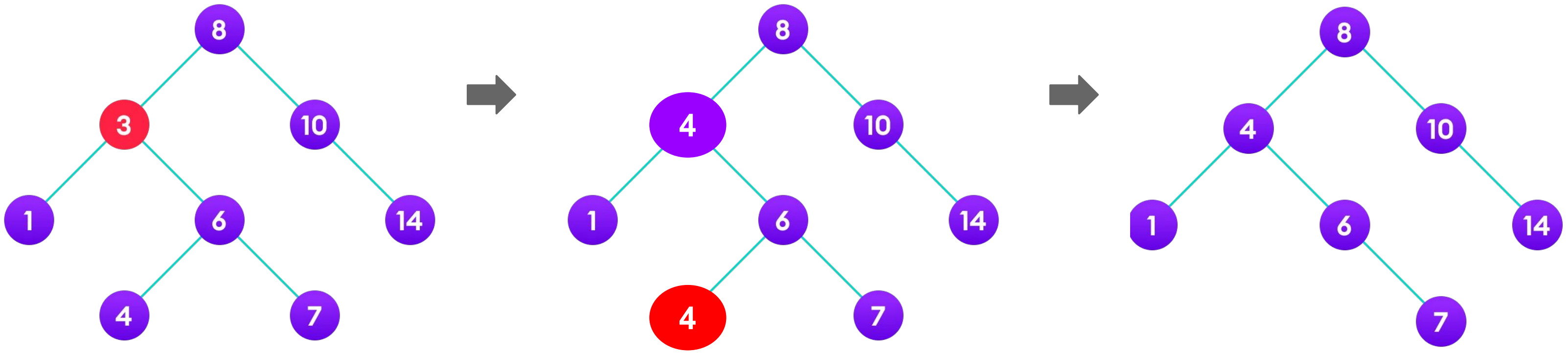
Deletion from a Binary Search Tree

inorder predecessor of 8 is 7 and successor is 10

In the third case, the node to be deleted has two children.

- Get the inorder successor of that node.
- Replace the node with the inorder successor.
- Remove the inorder successor/predecessor from its original position.

code for finding predecessor and successor -> bst3.cpp
code for deletion ->bst4.cpp



Deletion into a Binary Search Tree

- Non-recursive algorithm to perform deletion into a binary search tree.

```
BST-Delete(BST, D)
  if D.left = NIL then
    Shift-Nodes(BST, D, D.right)
  else if D.right = NIL then
    Shift-Nodes(BST, D, D.left)
  else
    E := BST-Successor(D)
    if E.parent ≠ D then
      Shift-Nodes(BST, E, E.right)
      E.right := D.right
      E.right.parent := E
    end if
    Shift-Nodes(BST, D, E)
    E.left := D.left
    E.left.parent := E
  end if
```

```
Shift-Nodes(BST, u, v)
  if u.parent = NIL then
    BST.root := v
  else if u = u.parent.left then
    u.parent.left := v
  else
    u.parent.right := v
  end if
  if v ≠ NIL then
    v.parent := u.parent
  end if
```

Deletion into a Binary Search Tree

- **Time Complexity:** The worst-case time complexity of deletion operation is O(d) where d is the depth of the Binary Search Tree.
  - In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of deletion operation may become O(n). It also requires additional efforts to find the inorder successor/predecessor.

- **Space Complexity:** The space complexity of deletion into a binary search tree is O(1) if we apply non-recursive algorithm, O(n) otherwise.

Applications of Binary Search Tree

- For efficient searching.
- For sorting data in increasing order.
- For indexing records in files.

Sorting

- Complexity ≈ Building a binary search tree ≈ $O(n\log_2 n)$

in creating we insert n node so it take nlogn time

Searching

- Best case: $O(1)$
- Worst case: $O(n)$
- Average case: $O(\log_2 n)$

generating bst from preorder -- >bst5.cpp

Next Lecture

- Balanced Binary Search Tree