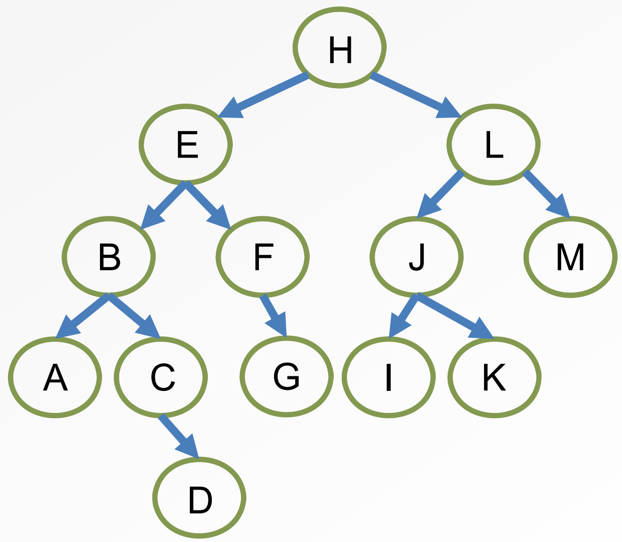
CE1007/CZ1007 DATA STRUCTURES

Review : **Binary Search Trees**

College of Engineering
School of Computer Science and Engineering

OUTLINE

- **Binary Search Trees (BST)**
- BST Operations:
  - Traversal
  - Inserting a node
  - Removing a node



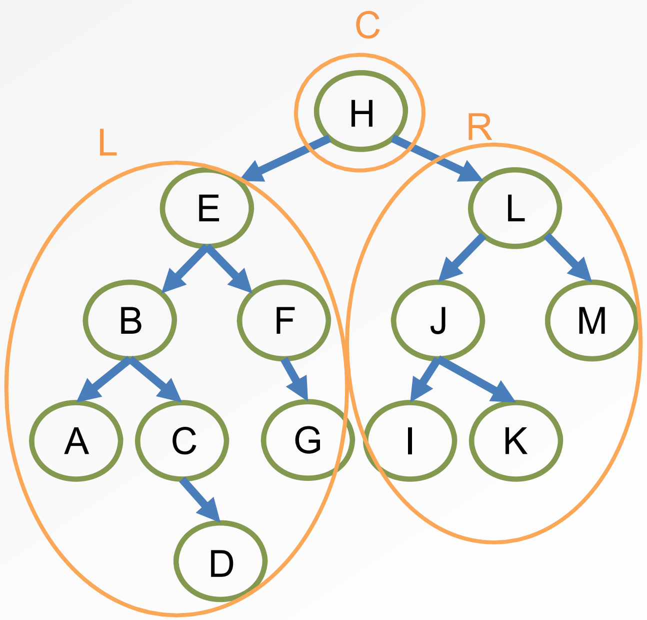
BINARY SEARCH TREE(BST)

- BSTs are a special form of BT
- BST rule:

  At every node C,

  L < C < R, where

  - C is the data in the current node
  - L represents the data in any/ all nodes from C's left subtree
  - R represents the data in any/all nodes from C's right subtree



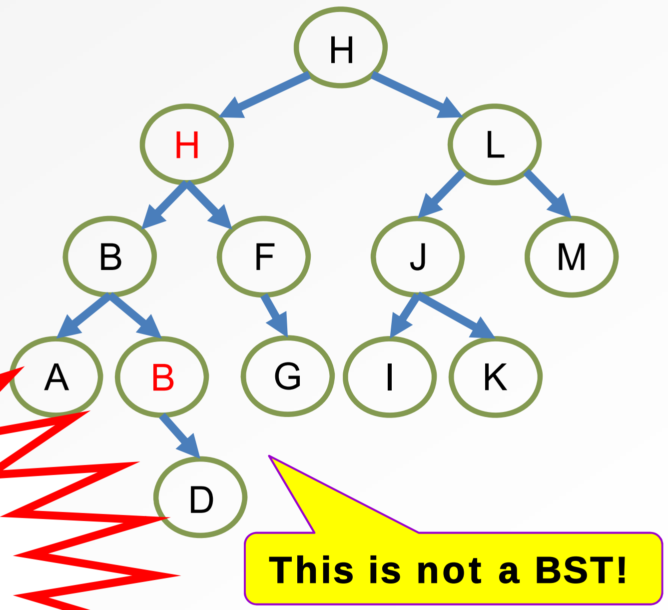
BINARY SEARCH TREE

- BSTs are a special form of BT
- At every node C,

  L <= C <= R, where

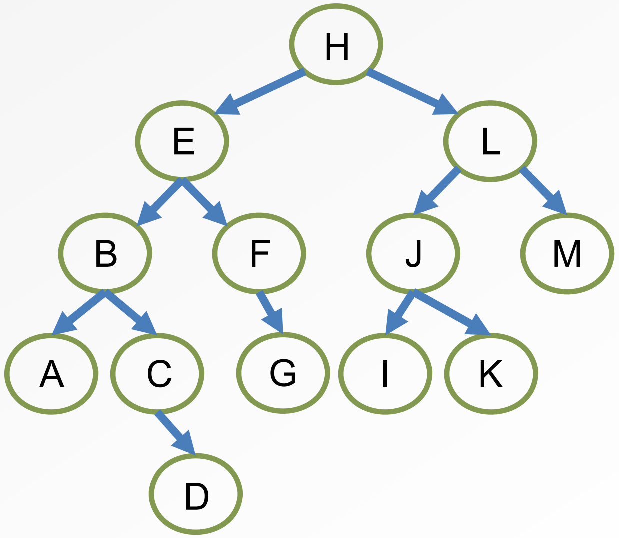
  - C is the data in the current node
  - L repr[illegible]s t[illegible]e d[illegible]ta in an[illegible]

NO = in the BST!
There must be no duplicate nodes in BST!



OUTLINE

- Binary Search Trees (BST)
- BST Operations:
  - **Traversal**
  - Inserting a node
  - Removing a node



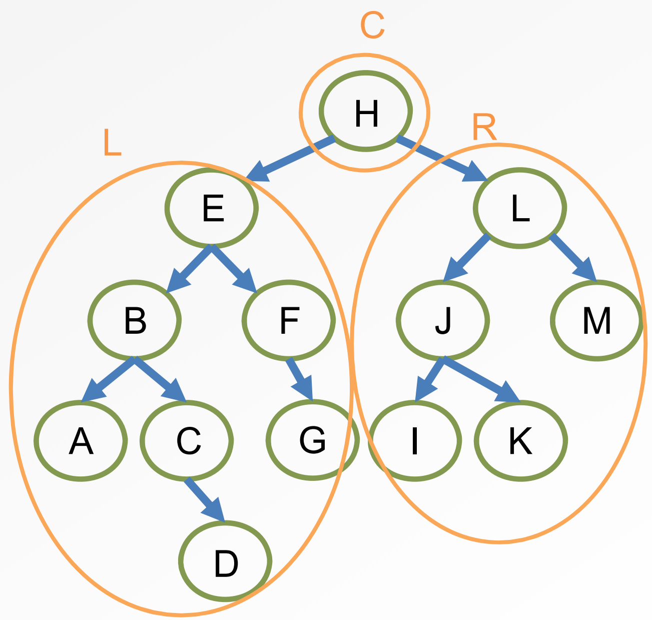
BINARY SEARCH TREE(BST)

- BSTs are a special form of BT
- BST rule:

  At every node C,

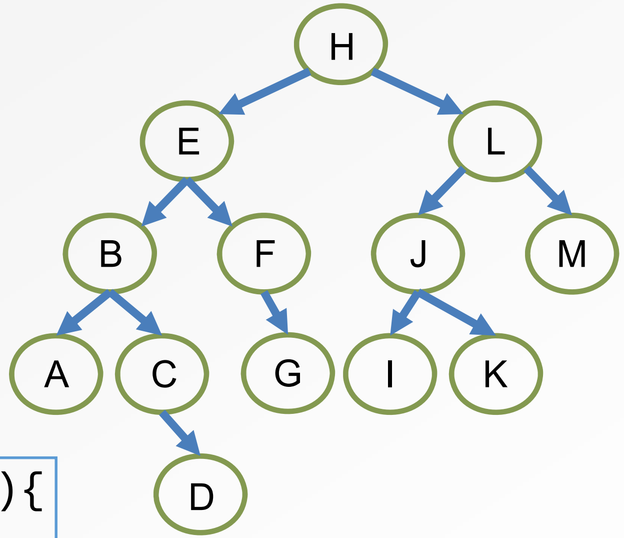
  L < C < R, where

  - C is the data in the current node
  - L represents the data in any/ all nodes from C's left subtree
  - R represents the data in any/all nodes from C's right subtree



BST TRAVERSAL (BSTT)

- BSTT() traverses a BST to search for a node with a matching item
- Begin with TreeTraversal template




```
void BSTT(BTNode *cur, char c){
    if (cur == NULL)
        return;

    // Do something

    BSTT(cur->left);
    BSTT(cur->right);
}
```

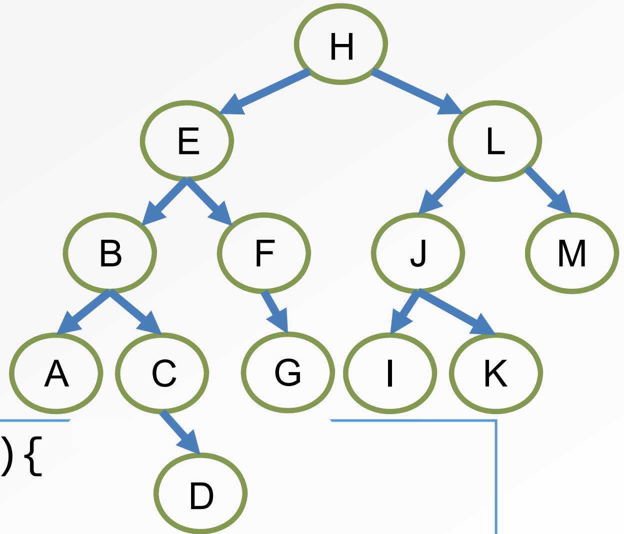
Do something with the current node's data

Visit the left child node

Visit the right child node

BST TRAVERSAL (BSTT)

- Now, at each node, we need to determine which subtree to keep visiting (and which subtree to ignore)




```
void BSTT(BTNode *cur, char c){
    if (cur == NULL) return;

    //do something
    if (c < cur->item)
        BSTT(cur->left,c);
    else
        BSTT(cur->right,c);
}
```

Do something with the current node's data

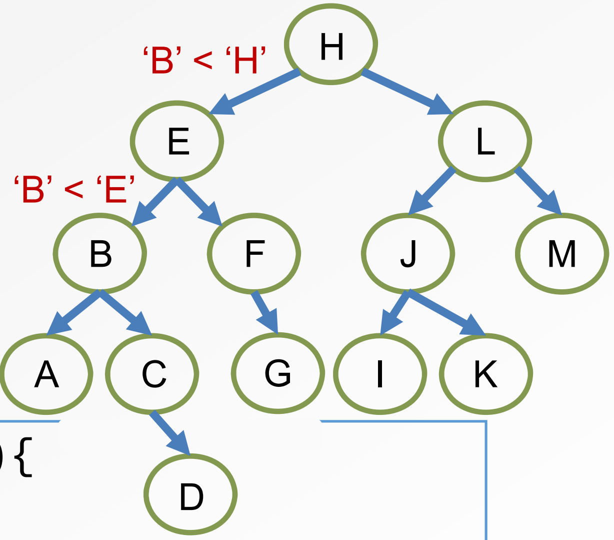
Visit the left child node

Visit the right child node

BST TRAVERSAL (BSTT)

- Check the traversal pattern for

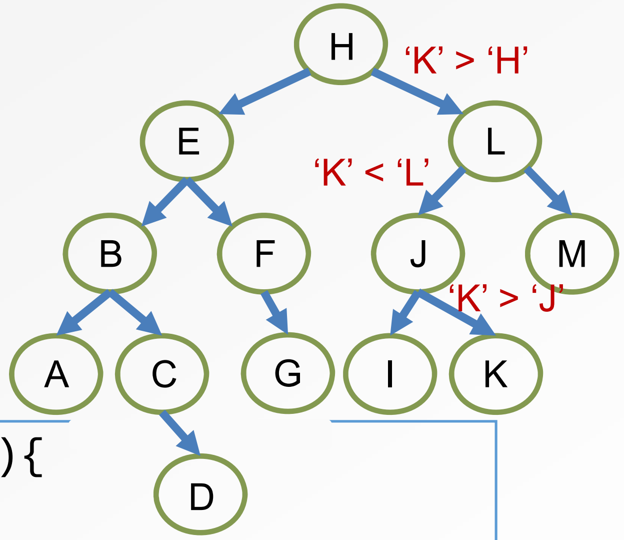
  BSTT(root, 'B')




```
void BSTT(BTNode *cur, char c){
    if (cur == NULL) return;
    if (c==cur->item)
    { printf("found!\n"); return;}
    if (c < cur->item)
       BSTT(cur->left,c);
    else
       BSTT(cur->right,c);
}
```

- Check the traversal pattern for

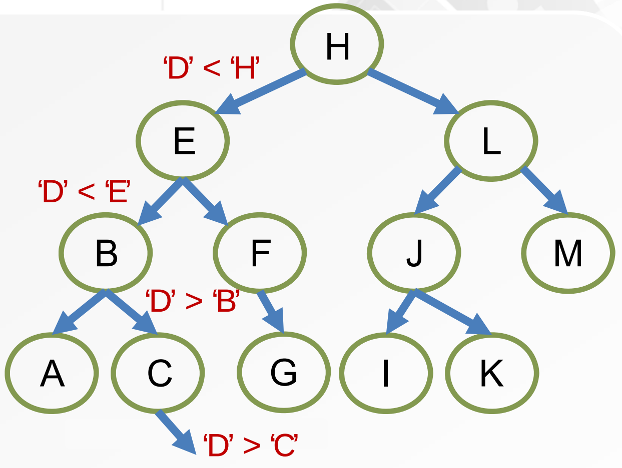
  BSTT(root, ‘K’)




```
void BSTT(BTNode *cur, char c){
    if (cur == NULL) return;
    if (c==cur->item)
    { printf(“found!\n”); return;}
    if (c < cur->item)
       BSTT(cur->left,c);
    else
       BSTT(cur->right,c);
}
```

BST TRAVERSAL (BSTT)

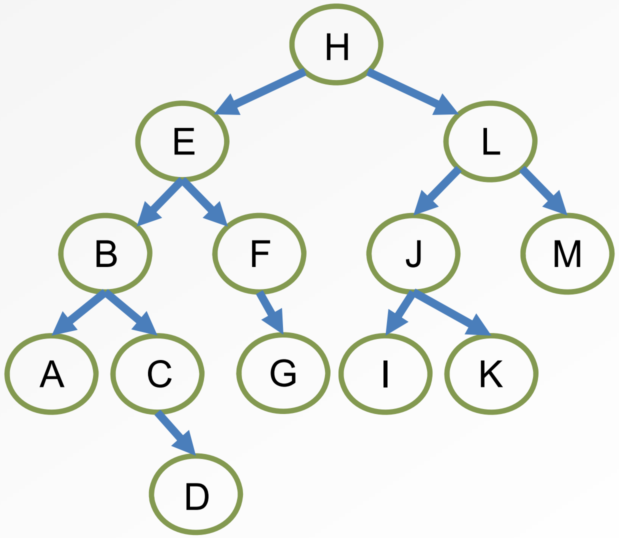
- What if the item doesn't exist?
- If we remove node 'D', and then check the traversal pattern for **BSTT(root, 'D')**




```
void BSTT(BTNode *cur, char c){
    if (cur == NULL){
      printf("can't find!");return;}
    if (c==cur->item){
      printf("found!\n"); return;}
    if (c < cur->item)
       BSTT(cur->left,c);
    else
       BSTT(cur->right,c);
}
```

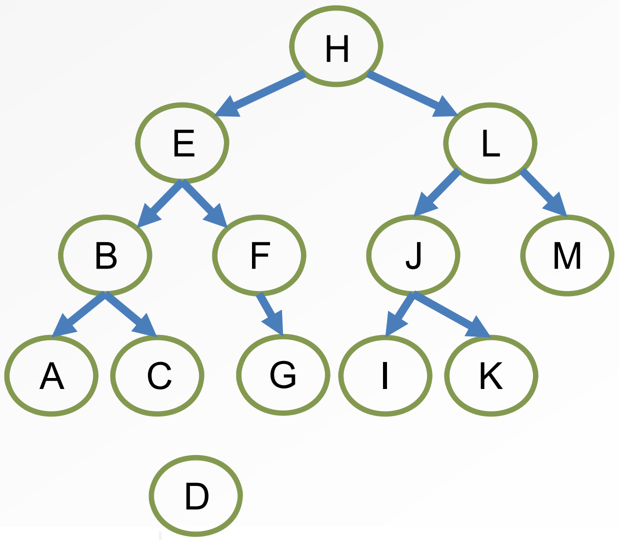
OUTLINE

- Binary Search Trees (BST)
- BST Operations:
  - Traversal
  - **Inserting a node**
  - Removing a node



INSERTING A NODE INTO A BST

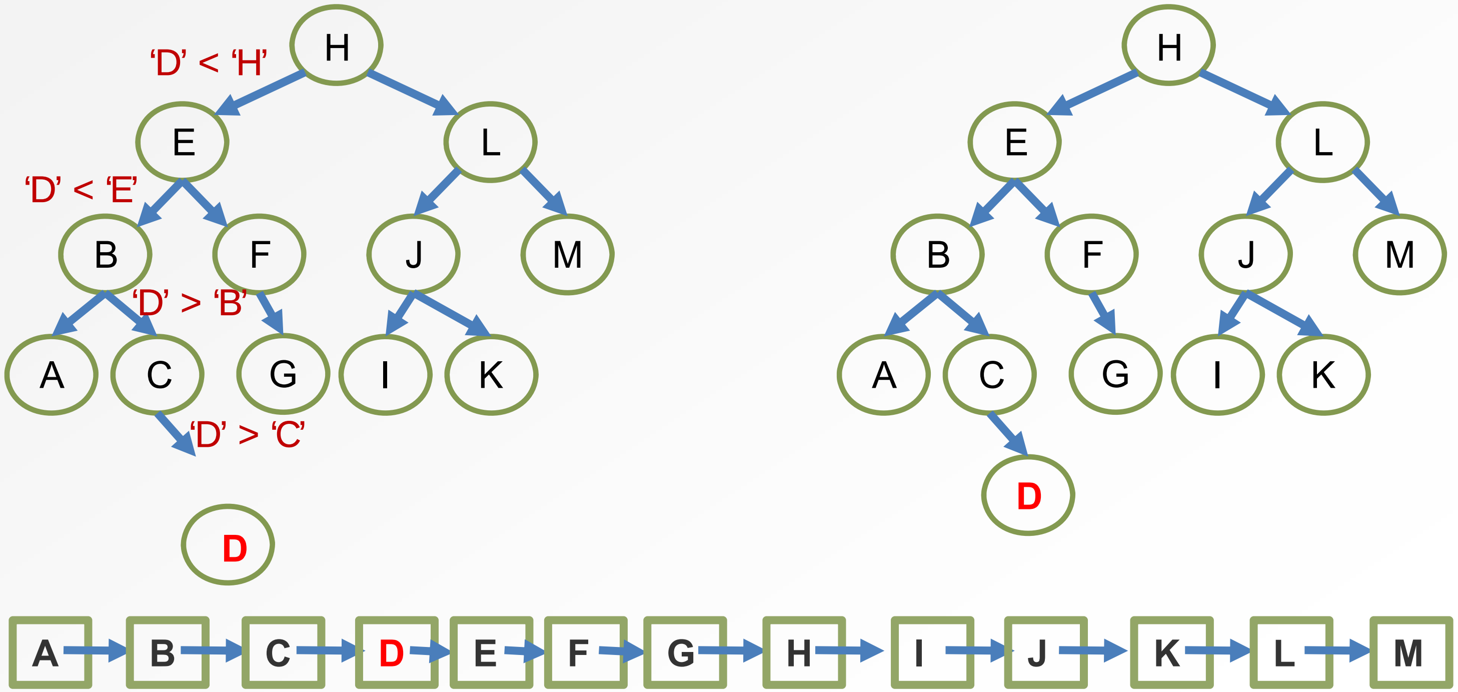
- Given an existing BST, an insertion operation must result in a BST
- How do we know where to place a new node ‘D’?
- Given an existing BST and a new value to store, there is always a unique position for the new value



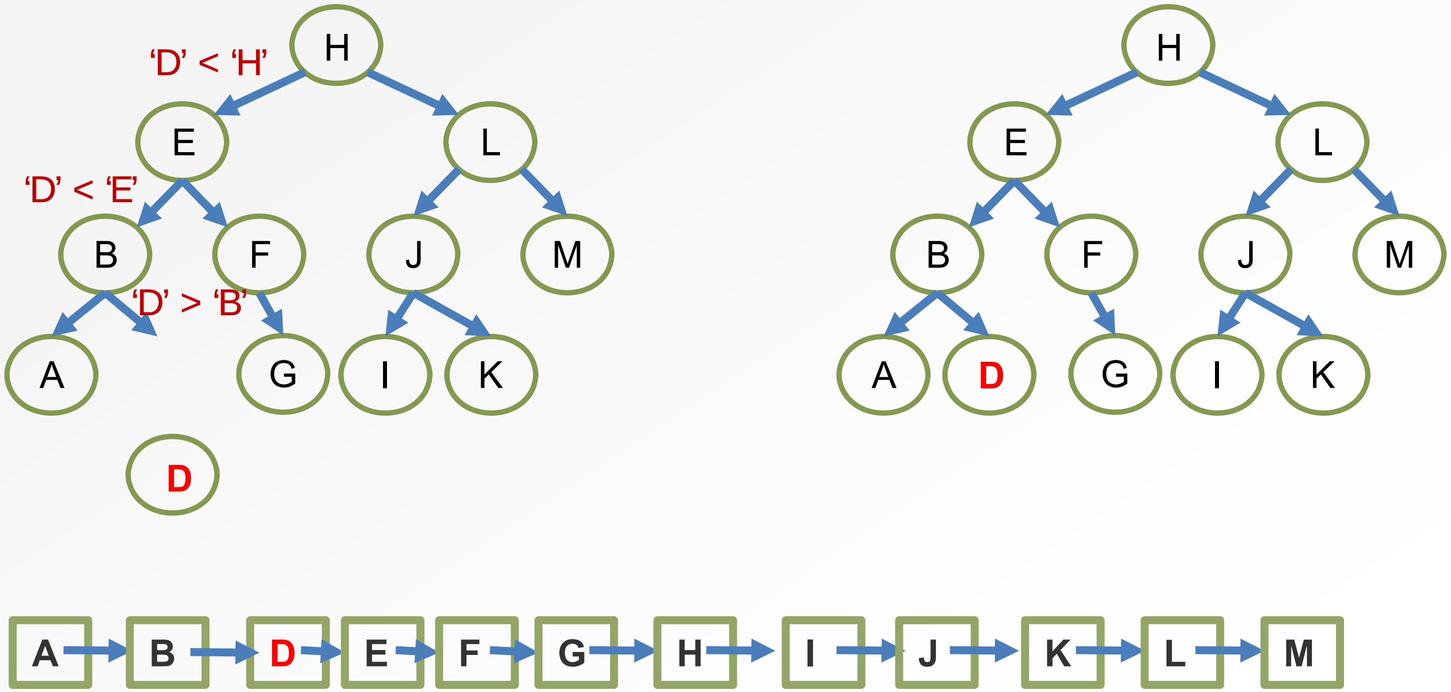
INSERTING A NODE INTO A BST

1. Use BSTT() to get to the correct empty location

2. Add the new node

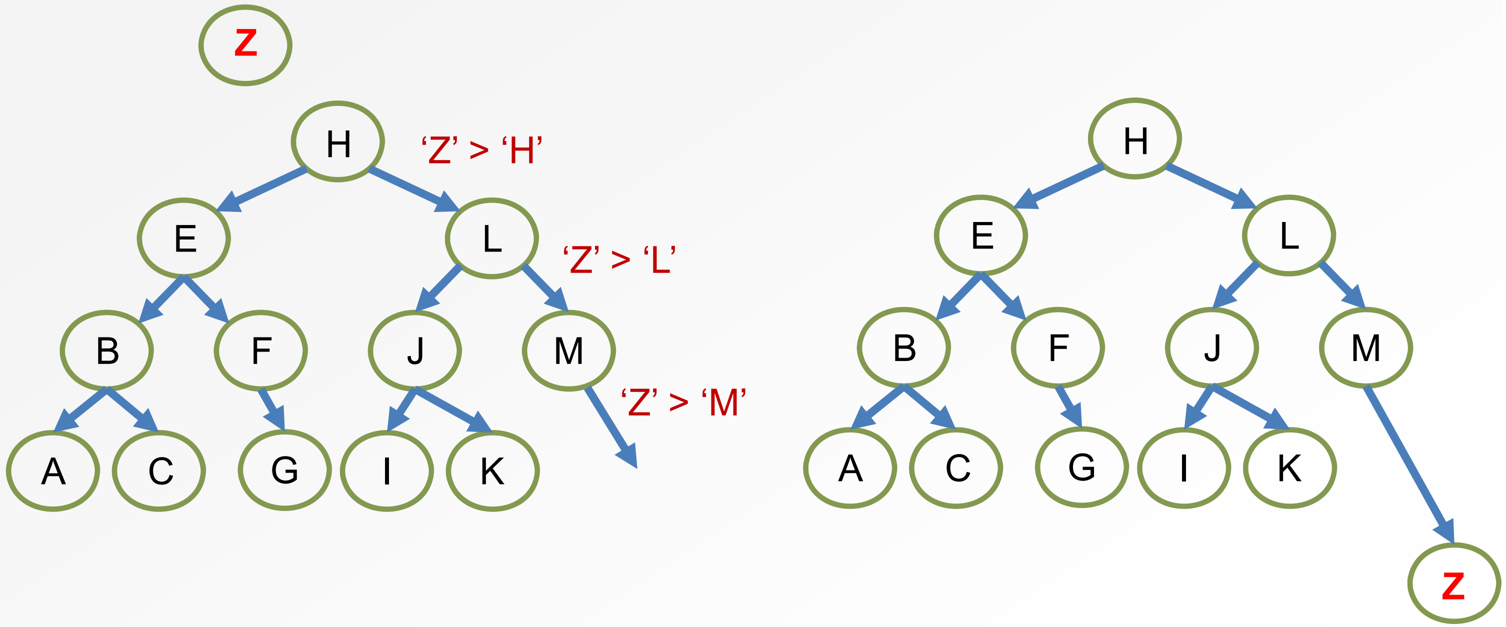


1. Use BSTT() to get to the correct empty location
2. Add the new node



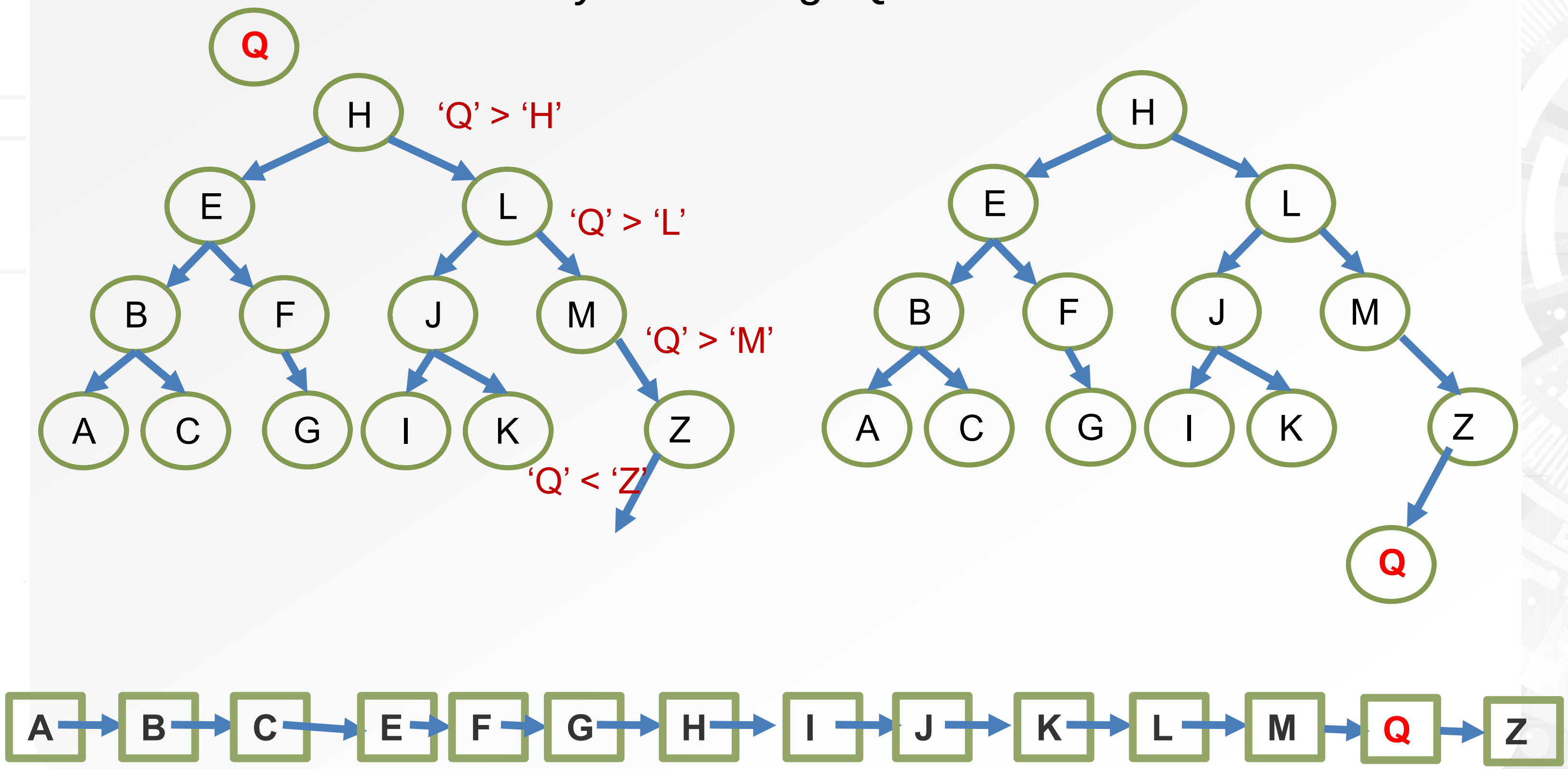
INSERTING A NODE INTO A BST

- Node insertion is relatively simple!
- Further exercise: Try Inserting ‘Z’



INSERTING A NODE INTO A BST

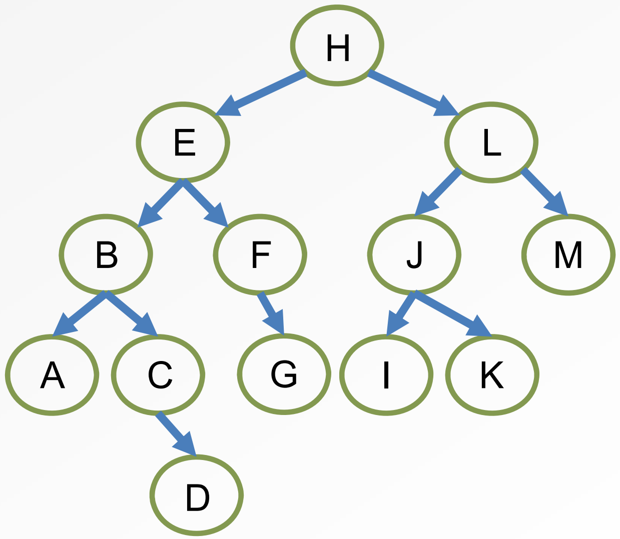
- Node insertion is relatively simple!
- Further exercise: Try Inserting 'Q'



OUTLINE

- Binary Search Trees (BST)
- BST Operations:
  - Traversal
  - Inserting a node
  - **Removing a node**

After removal, the tree is still a BST



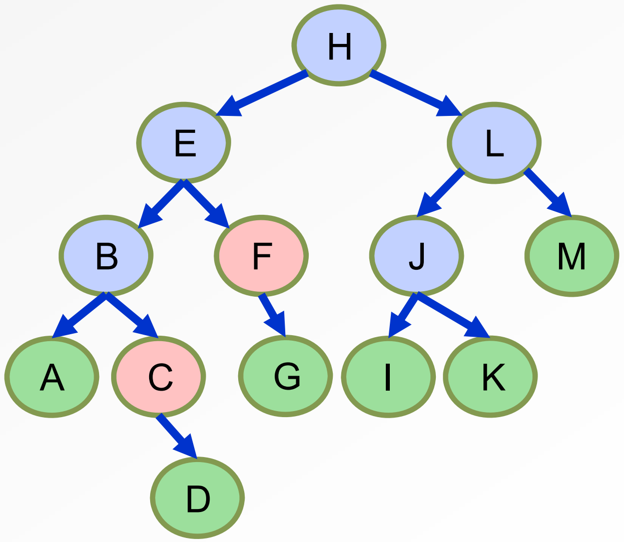
REMOVING A NODE FROM A BST

- Node removal is more complicated
- Beginning with a BST, the resulting tree after removing a node must still be a BST

  Obey the BST rule: L < C < R

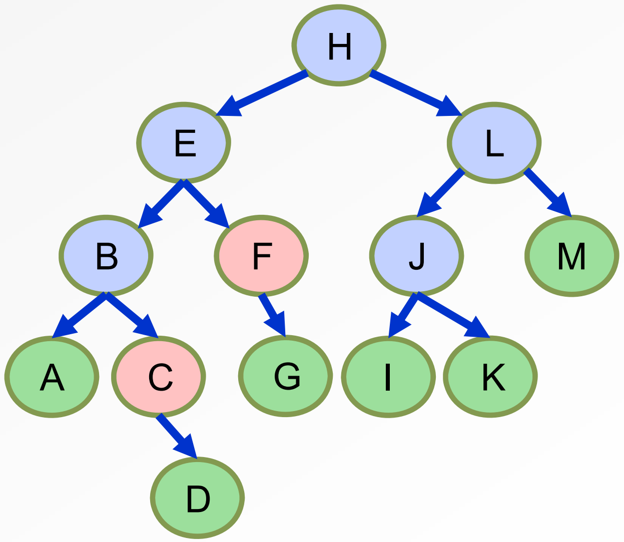
REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. x has no children:
     - Remove x
  2. x has one child y:
     - Replace x with y
  3. x has two children:
     - Swap x with successor
     - Perform case 1 or 2 to remove it



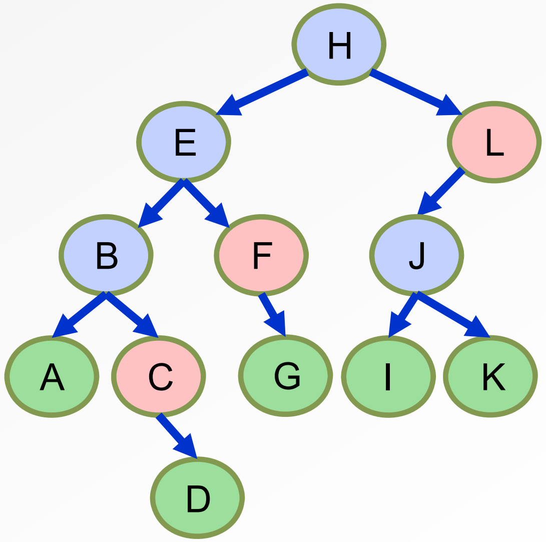
REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. **x has no children:**
     - **Remove x**
  2. x has one child y:
     - Replace x with y
  3. x has two children:
     - Swap x with successor
     - Perform case 1 or 2 to remove it

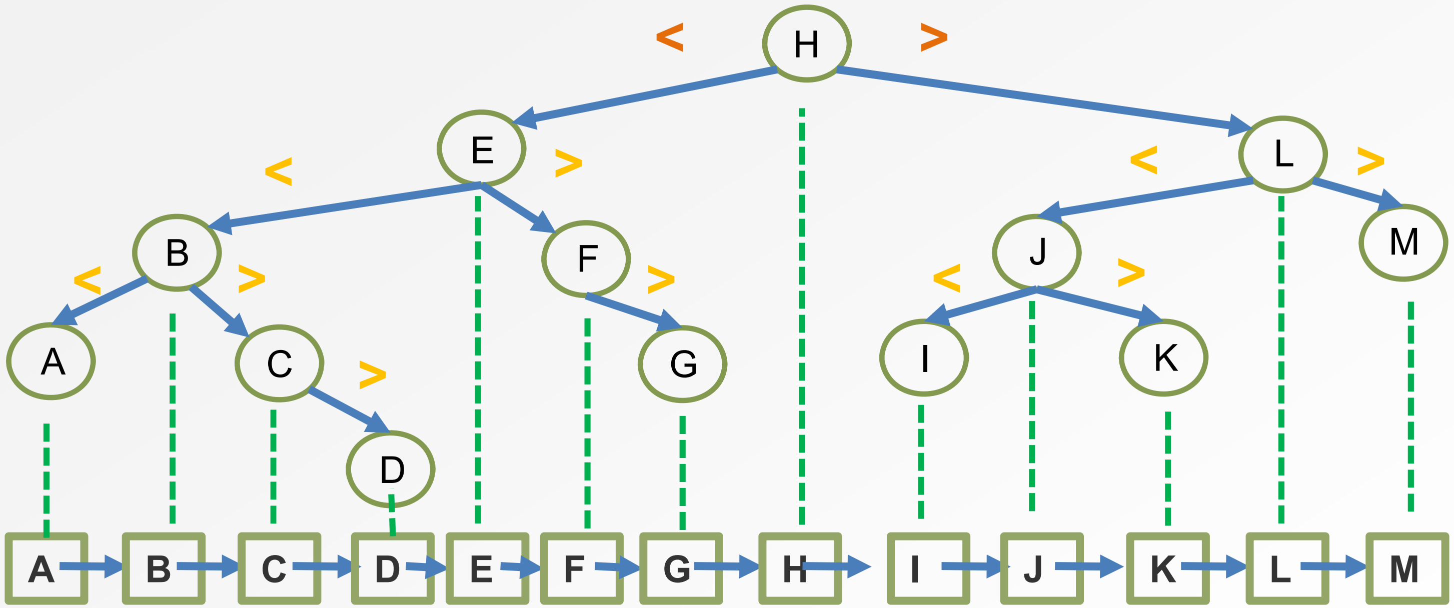


REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. x has no children:
     - Remove x
  2. **x has one child y:**
     - **Replace x with y**
  3. x has two children:
     - Swap x with successor
     - Perform case 1 or 2 to remove it



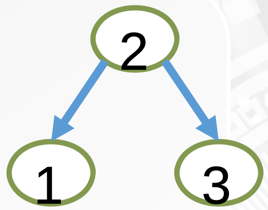
MAPPING: TREE(IN-ORDER) → LIST



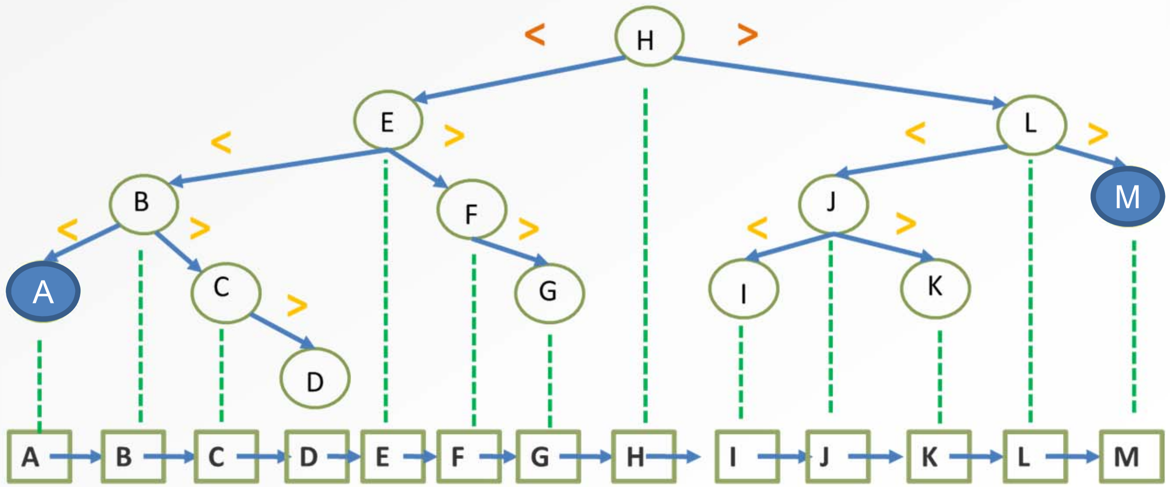
- If we draw the BST carefully:
  - **Left subtree on the left side of the current node;**
  - **Right subtree on the right side of the current node;**
- Mapping to X-axis will produce **a sorted list.**

FEATURES

- BST's in-order traversal produces a sorted list!
  - **L < C < R** rule ensures sorted order

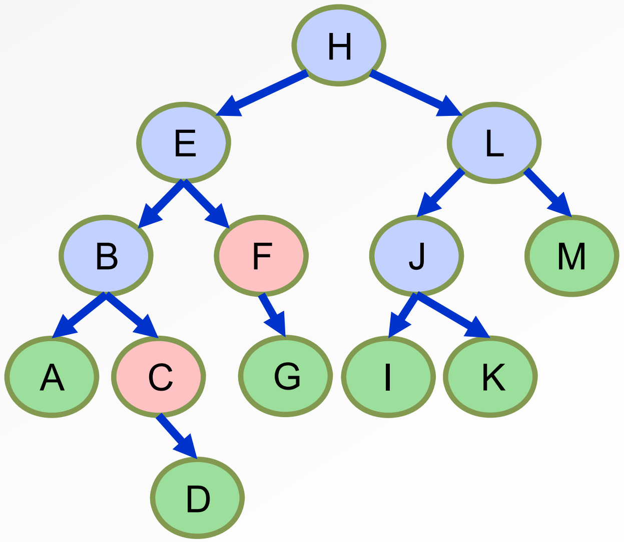


- The binary-search-tree property guarantees that:
  - The minimum is located at the left-most node
  - The maximum is located at the right-most node



REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. x has no children:
     - Remove x
  2. x has one child y:
     - Replace x with y
  3. **x has two children:**
     - **Swap x with successor**
     - **Perform case 1 or 2 to remove it**



WHAT IS THE SUCCESSOR OF X?

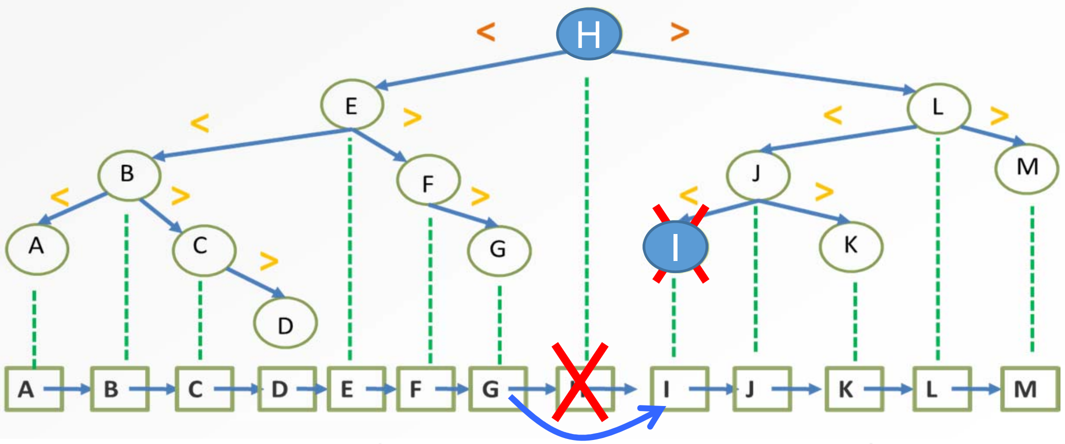
Replacing a node with its in-order successor ensures that the BST rule (L<C<R) is maintained

In-order traversal of a BST produce a sorted list (in ascending order)

Successor is:

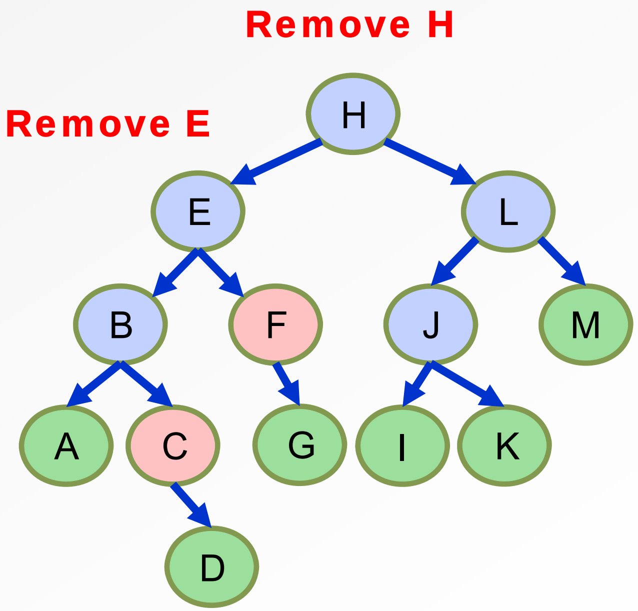
- **The node immediately after it in the sorted list,** or
- **The next node visited using an in-order traversal**

X has two children, so X's successor is minimum node in its right subtree.
E.g.: H's successor is I, E's successor is F, J's successor is K.

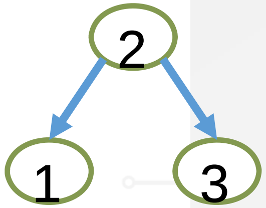


REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. x has no children:
     - Remove x
  2. x has one child y:
     - Replace x with y
  3. **x has two children:**
     - **Swap x with successor**
     - **Perform case 1 or 2 to remove it**



QUESTIONS



- **Why will case 3 always go to case 1 or case 2?**

  A: because when X has 2 children, its successor is **> X** but the **minimum** in its **right** subtree, so the successor should not have left child.

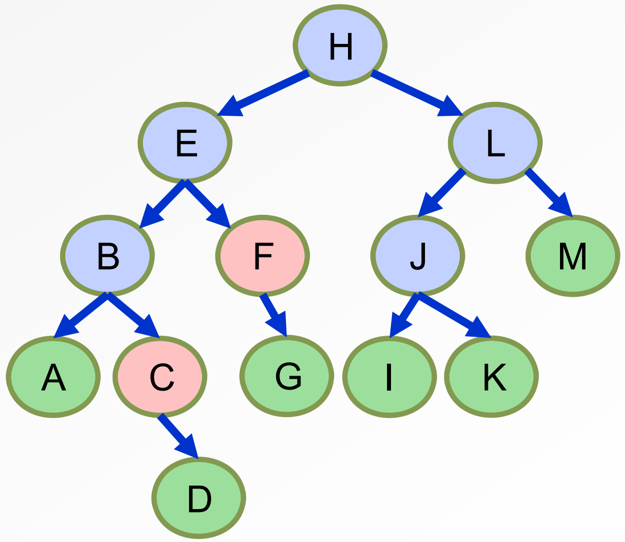
  It might have no child(case 1) or one right child(case 2).

- **Could we swap x with predecessor instead of successor?**

  A: yes.

REMOVING A NODE FROM A BST

- Remove node X - a bit tricky
- 3 cases:
  1. x has no children:
     - Remove x
  2. x has one child y:
     - Replace x with y
  3. **x has two children:**
     - **Swap x with successor**
     - **Perform case 1 or 2 to remove it**



WHAT IS THE SUCCESSOR OF X?

Replacing a node with its in-order **predecessor** ensures that the BST rule (L<C<R) is maintained

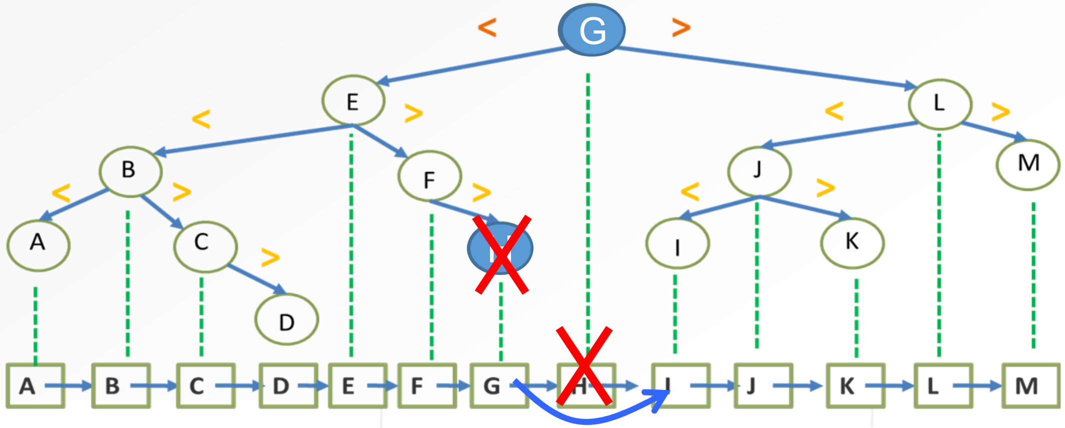
In-order traversal of a BST produce a sorted list (in ascending order)

Successor / predecessor:

- **The node immediately after / before it in the sorted list**
- **The next / previous node visited using an in-order traversal**

X has two children, so X's predecessor is **< X** but **maximum** node in its **left** subtree.

E.g.: H's predecessor is G, E's predecessor is D, J's predecessor is I.



TODAY YOU SHOULD BE ABLE TO

- Define a Binary Search Tree
- From a list, how do we construct a Binary Search Tree? Is it efficient?
- How do we traverse a BST to search a item?
- How do we insert/remove a node from a BST?