NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Lecture 09: **Binary Search Trees**

Dr. Owen Noel Newton Fernando

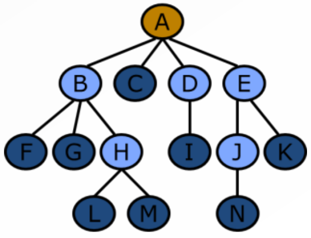
College of Engineering

School of Computer Science and Engineering

RECALL: WHY TREES?

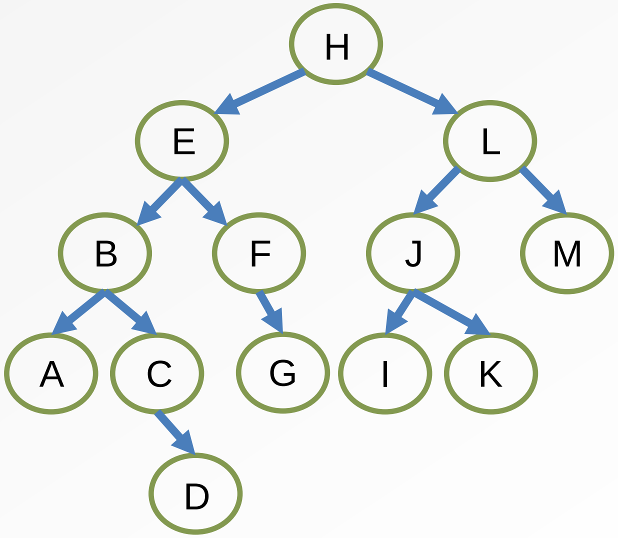
- Model layouts with hierarchical relationships between items
  - Chain of command in the army
  - Personnel structure in a company
  - (Binary tree structure is limited because each node can have at most two children)

- Tree structures also allow us to
  - Some problems require a tree structure: some games, most optimization problems, etc.
  - Allow us to do the following very quickly: (we'll see that in the following lectures)
    - **Search for a node with a given value**
    - **Add a given value to a list**
    - **Delete a given value from a list**



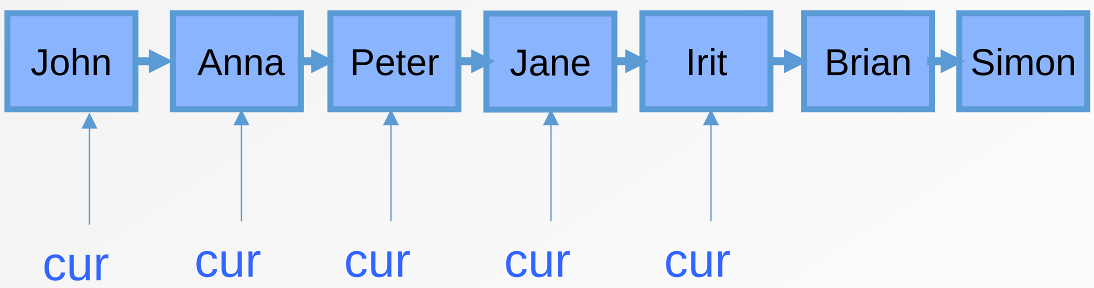
OUTLINE

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



inefficient

Given a linked list of names, how do we check whether a given name(e.g., Irit) is in the list?



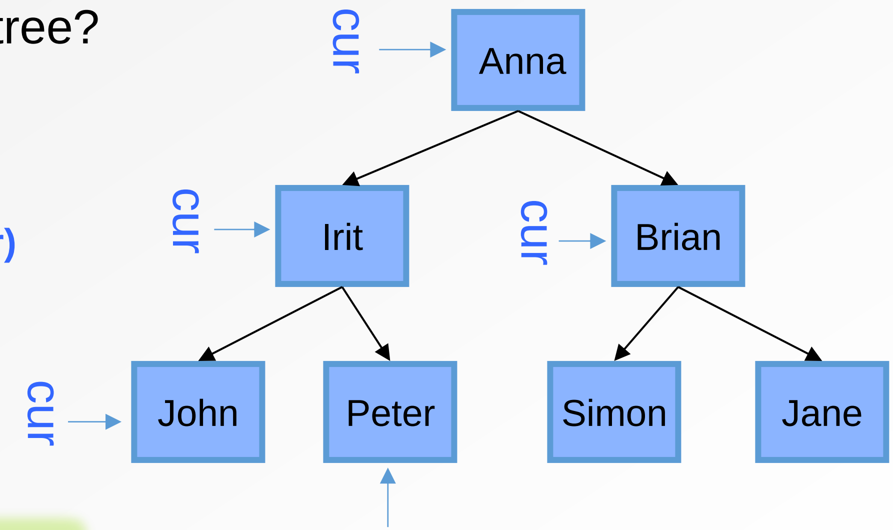

```
while (cur!=NULL) {
        if cur->item == "Irit"
            found and stop searching;
        else
            cur = cur->next; }
```

How many nodes are visited during search?
--best case: 1 node (John)
--worst case: 7 nodes (Simon)
--avg. case: (1+2+3+…+7)/7=4 nodes

ITEM SEARCH-LINKED LIST

inefficient

Given a binary tree of names, how do we check whether a given name(e.g., Brian) is in the tree?



Use the TreeTraversal (Pre-order) template, to check every node

```
TreeTraversal(Node N)
 If N==NULL return;
 if N.item=given_name return;
 TreeTraversal(LeftChild);
 TreeTraversal(RightChild);
 Return;
```

How many nodes are visited during search?
--best case: 1 node (Anna)
--worst case: 7 nodes (Jane)
--Avg. case: (1+2+3+…+7)/7=4 nodes

IS THERE A WAY TO ORGANIZE THE DATA SO THAT ITEM SEARCH CAN BE MORE EFFICIENT?

- What's a good way to arrange data in our tree so that we can efficiently store and retrieve items?
- How do we efficiently (minimize number of operations) find an item in a binary tree?
- Given a list of items, how do we check if some given item is contained inside?

For the list: John → Anna → Peter → Jane → Irit → Brian → Simon

Sort it using alphabetical order:

Divide them into groups

Search the given name X

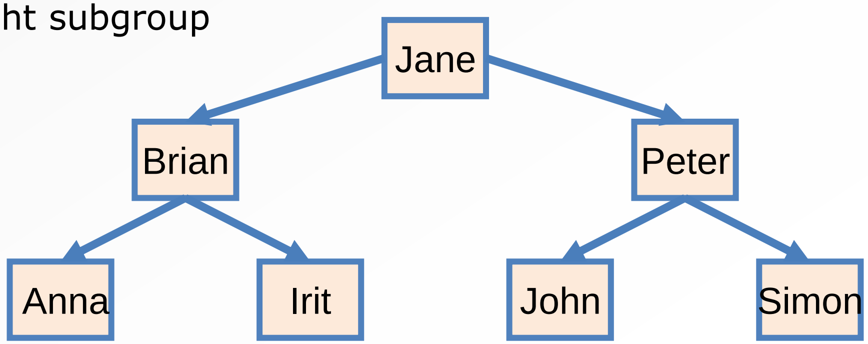
- Pick the one in the middle, “Jane”, if X!=“Jane”:
  - For names before “Jane”, lookup its left subgroup, ignore its right subgroup
    - Pick the one in the middle of the subgroup, “Brain”, if X!=“Brain”:
      - names before “Brian”, lookup its left subgroup
      - names after “Brian”, lookup its right subgroup
  - For names after “Jane”, lookup its right subgroup, ignore its left subgroup
    - Pick the one in the middle of the subgroup, “Peter”, if X!=“Peter”:
      - <“Peter”, lookup its left subgroup
      - >“Peter”, lookup its right subgroup

Anna → Brian → Irit → Jane → John → Peter → Simon

IT FORMS A BINARY SEARCH TREE (BST)

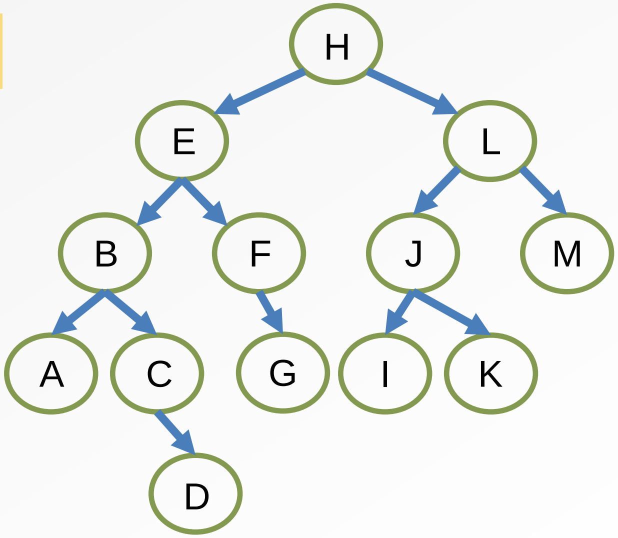
Divide them into groups

- Pick the one in the middle, “Jane”, if X!=“Jane”:
  - For names before “Jane”, lookup its left subgroup, ignore its right subgroup
    - Pick the one in the middle of the subgroup, “Brain”, if X!=“Brain”:
      - names before “Brian”, lookup its left subgroup
      - names after “Brian”, lookup its right subgroup
  - For names after “Jane”, lookup its right subgroup, ignore its left subgroup
    - Pick the one in the middle of the subgroup, “Peter”, if X!=“Peter”:
      - <“Peter”, lookup its left subgroup
      - >“Peter”, lookup its right subgroup



OUTLINE

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



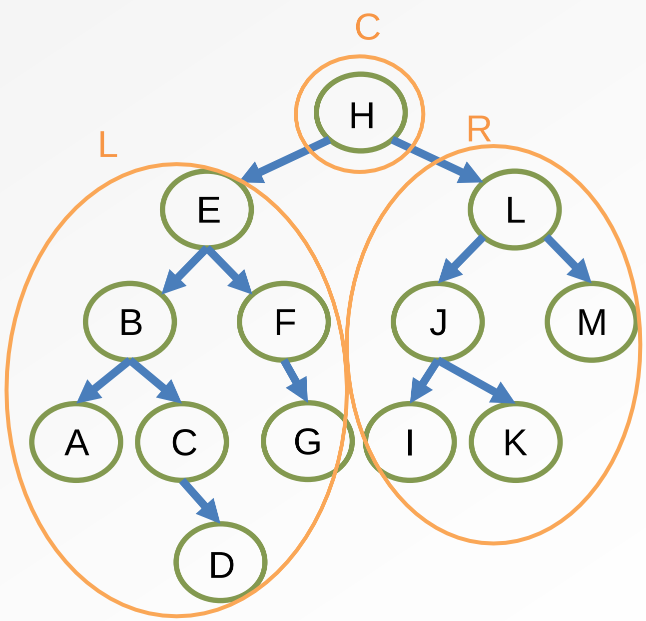
BINARY SEARCH TREE(BST)

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

  At every node C,

  L < C < R, where

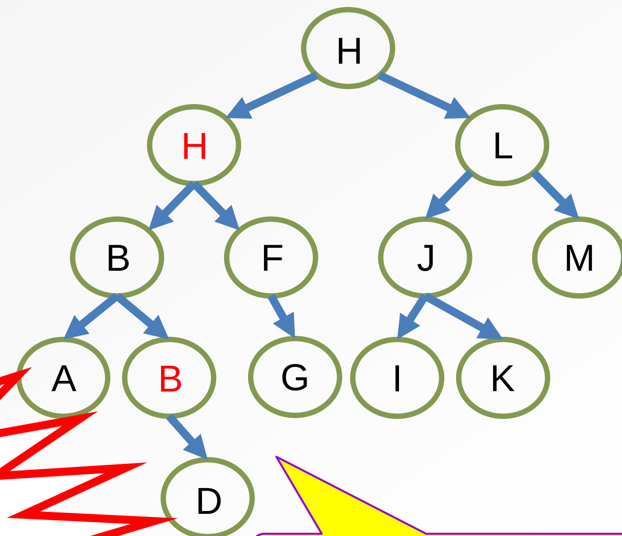
  - C is the data in the current node
  - L represents the data in any/ all nodes from C's left subtree
  - R represents the data in any/all nodes from C's right subtree



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

  L <= C <= R, where

  - C is the data in the current node
  - L represents the data in an…

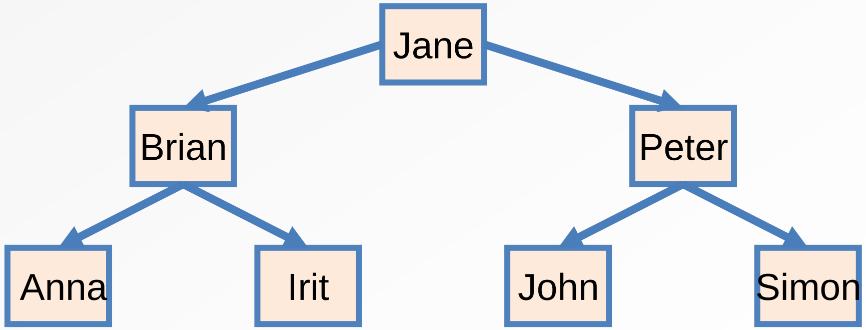


This is not a BST!

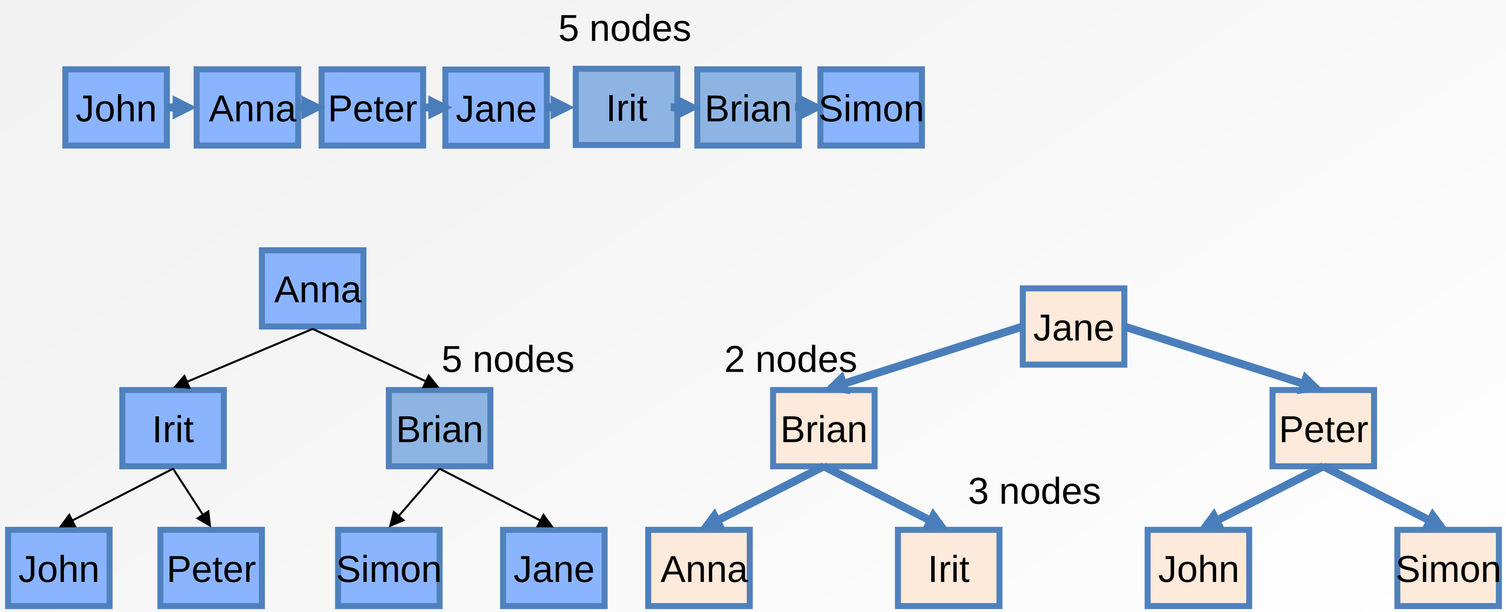
NO = in the BST!
There must be no duplicate nodes in BST!

BST IS FOR EFFICIENT ITEM SEARCH

- This is a BST, satisfies 'L < C < R'



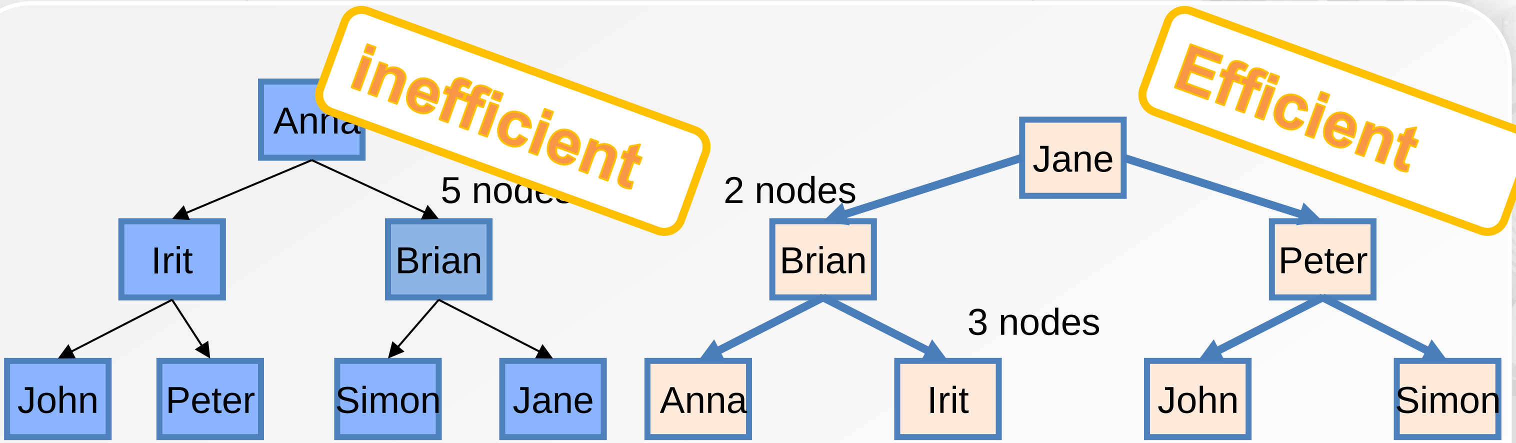
BST IS EFFICIENT FOR ITEM SEARCH



How many nodes are visited during search?
--best case: 1 node (Anna)
--worst case: 7 nodes (Jane)
--Avg. case: (1+2+3+…+7)/7=4 nodes

How many nodes are visited during search?
--best case: 1 node (Jane)
--worst case: 3 nodes (Anna)
--Avg. case: (1+2*2+3*4)/7=2.43 nodes

BST IS EFFICIENT FOR ITEM SEARCH



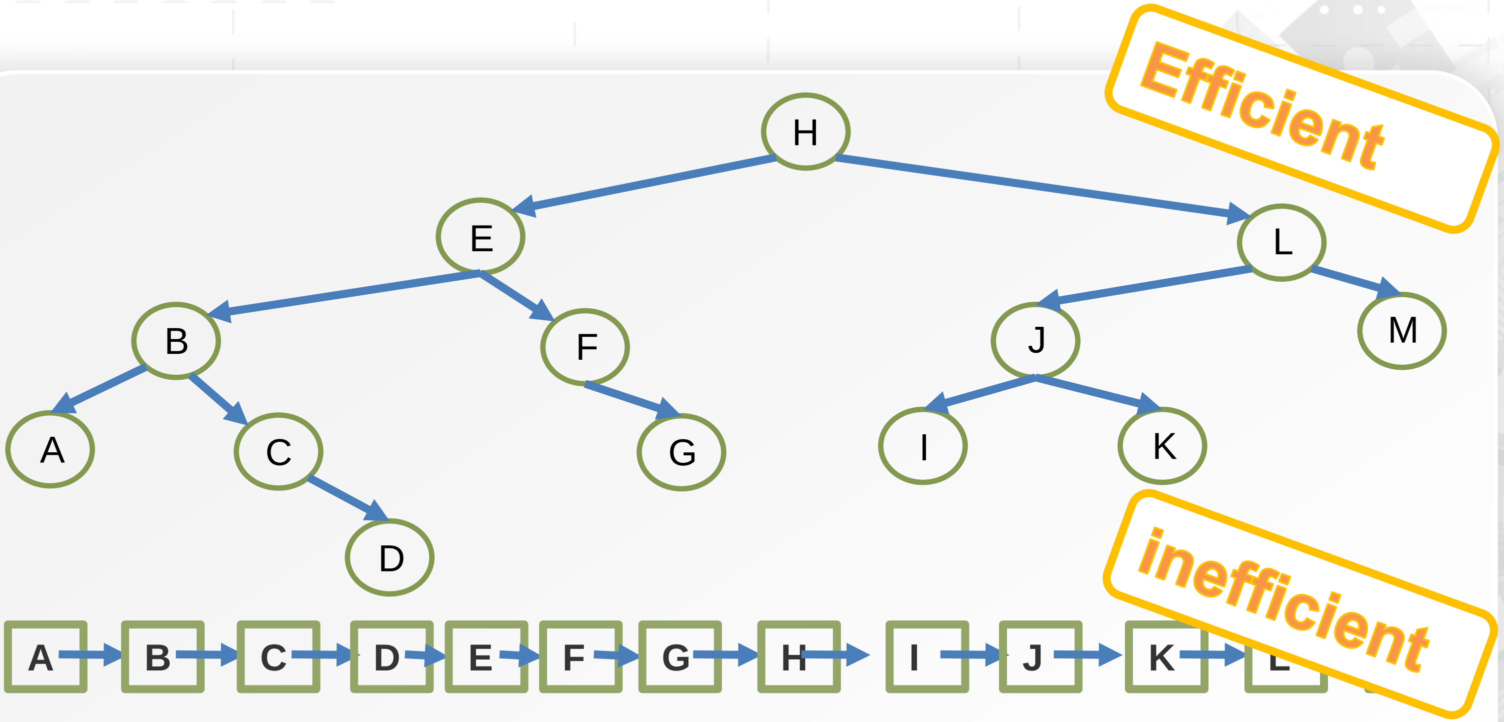
How many nodes are visited during search?
In general, for a BT with n nodes:
--best case: First node in traversal
--worst case: Last node in traversal, n

How many nodes are visited during search?
In general, for a BST with n nodes:
--best case: First node in traversal
--worst case:
leaf node: the height of the root + 1
Minimal height H = $\lfloor \log_2 n \rfloor$

As n becomes a big number, the difference between them becomes even greater

Height of a node = number of links from that node to the deepest leaf node

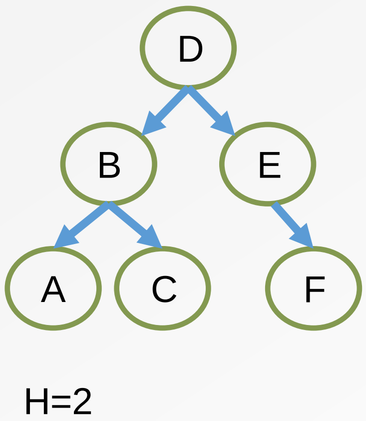
ANOTHER BST

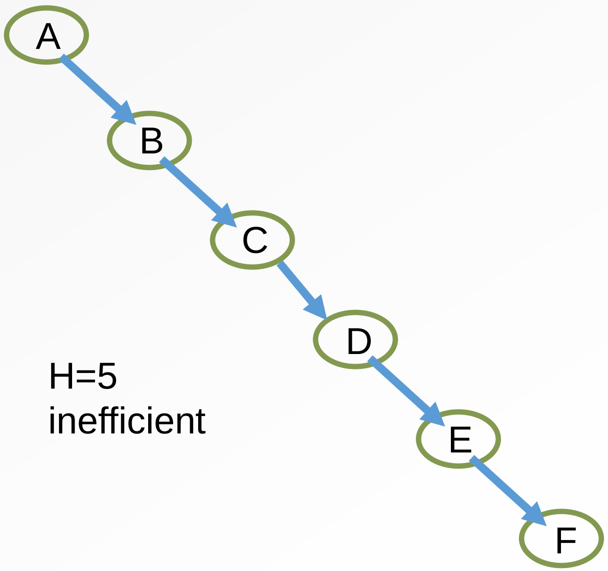


- How do we check if a given item X is stored in the tree?
- To find D, we need to visit H-E-B-C-D, 5 nodes.
- To find any node in the tree, at most visit 5 nodes.
- But for the linked list, the worst case need to visit all nodes - 13 nodes.

NOTICE: NOT ALL BST ARE EFFICIENT FOR SEARCH

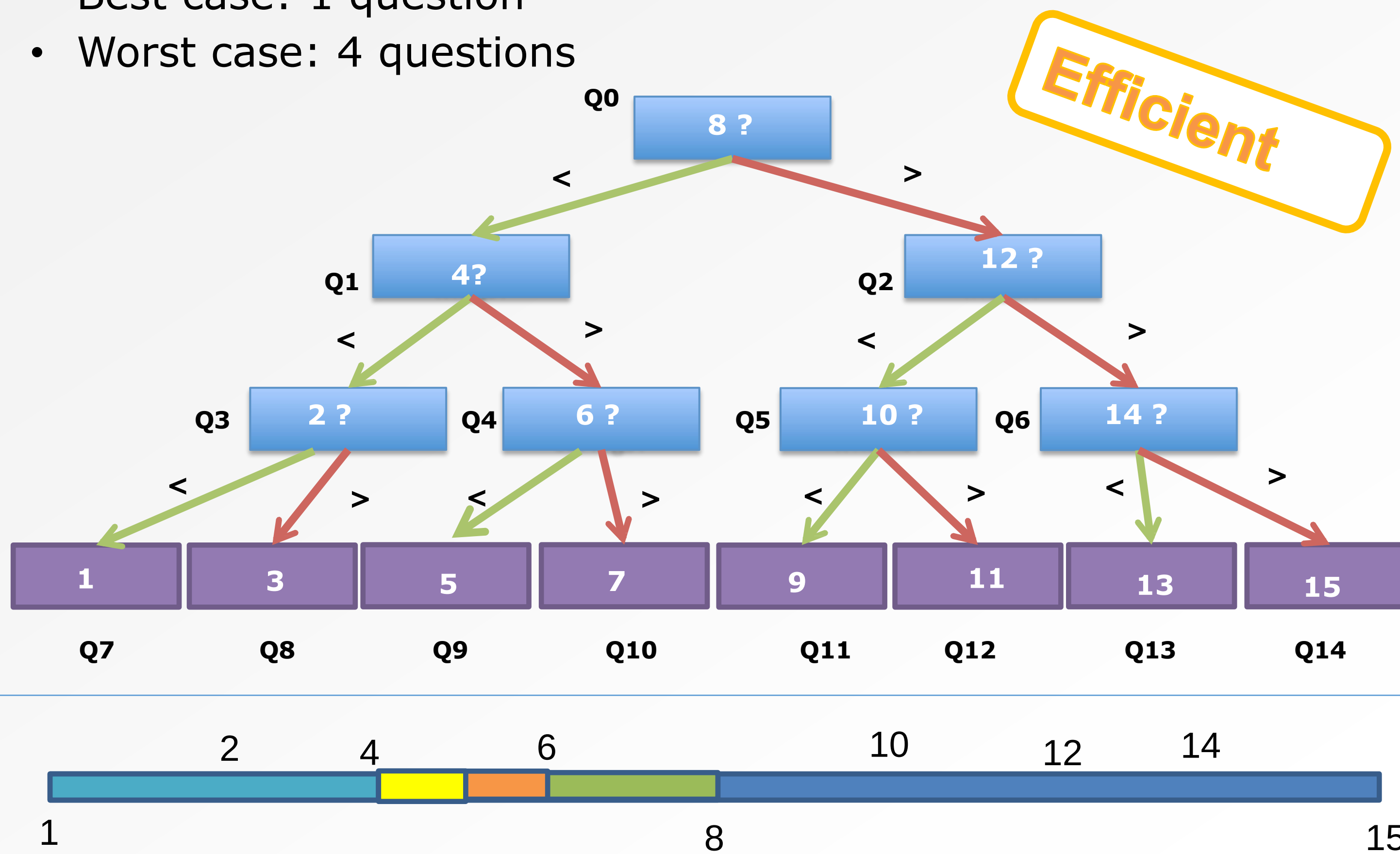
- What does a good/bad BST look like?
- Two possible BST representations of the list





RECALL: NUMBER GAME [1, 15]

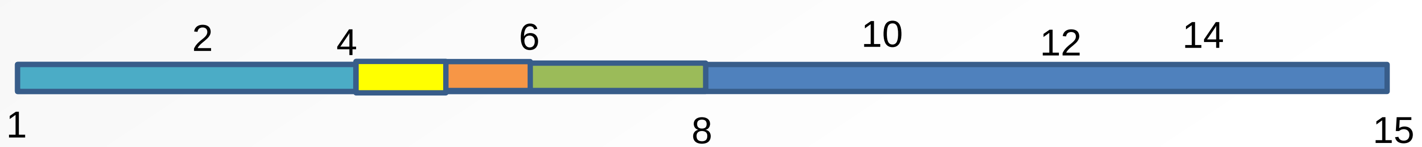
- How many questions do you ask to guess the number?
- Best case: 1 question
- Worst case: 4 questions



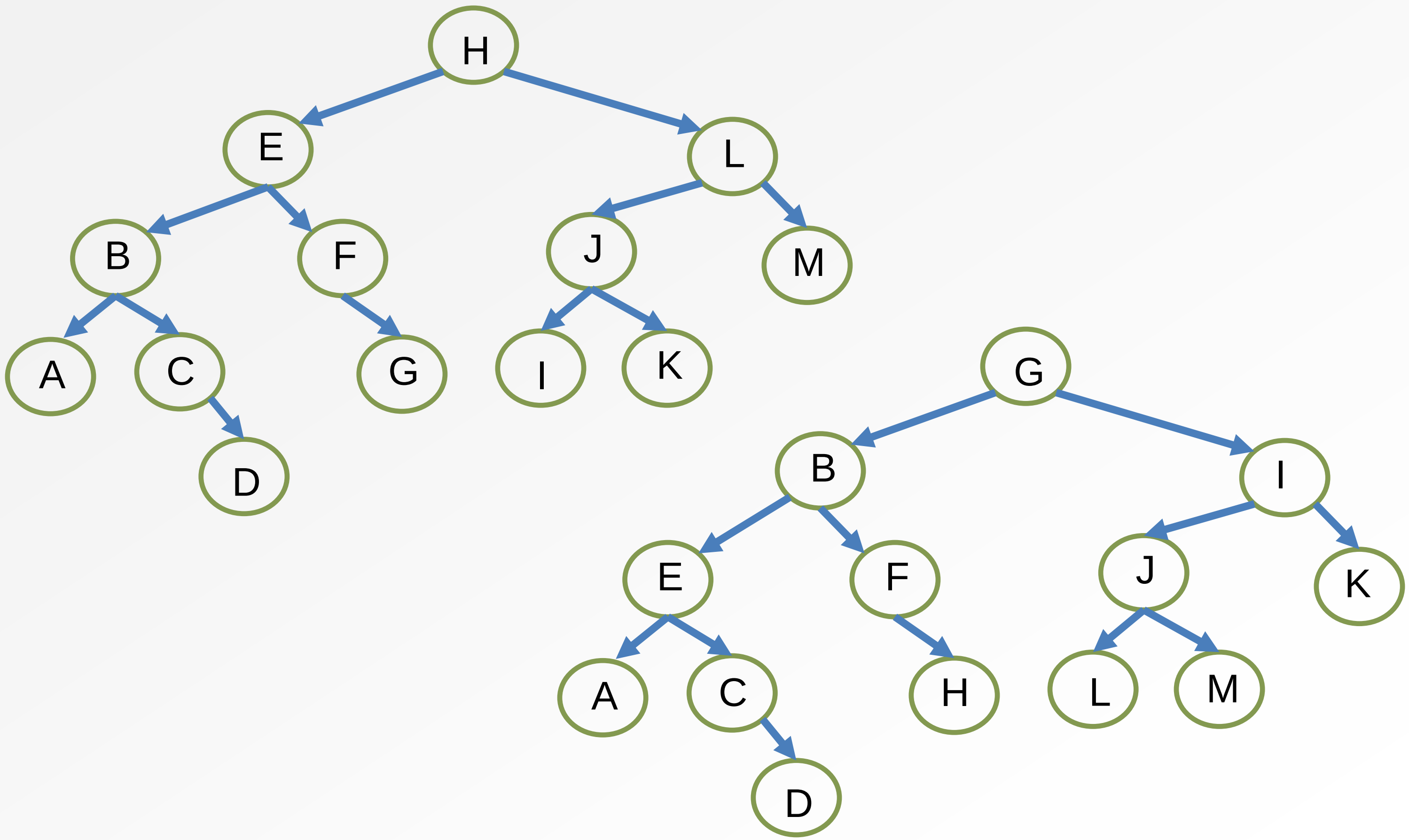
RECALL: NUMBER GAME [1, 15]

- How many questions do you ask to guess the number?
- If the strategy is to guess **1 first, and then 2, 3, …, 15**
- Best case: **1** question
- Worst case: **15** questions

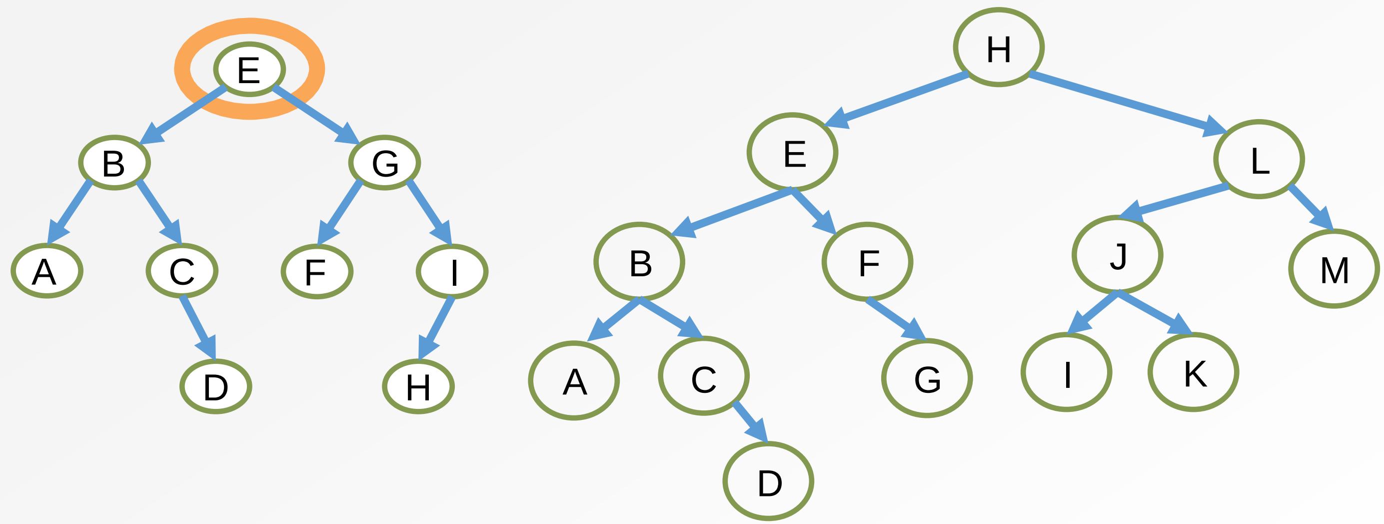
inefficient



EXERCISE:
WHICH BINARY TREES IS EFFICIENT FOR SEARCH ?



THINK: WHAT IS THE VISITING SEQUENCE USING IN-ORDER TRAVERSAL FOR A BST?

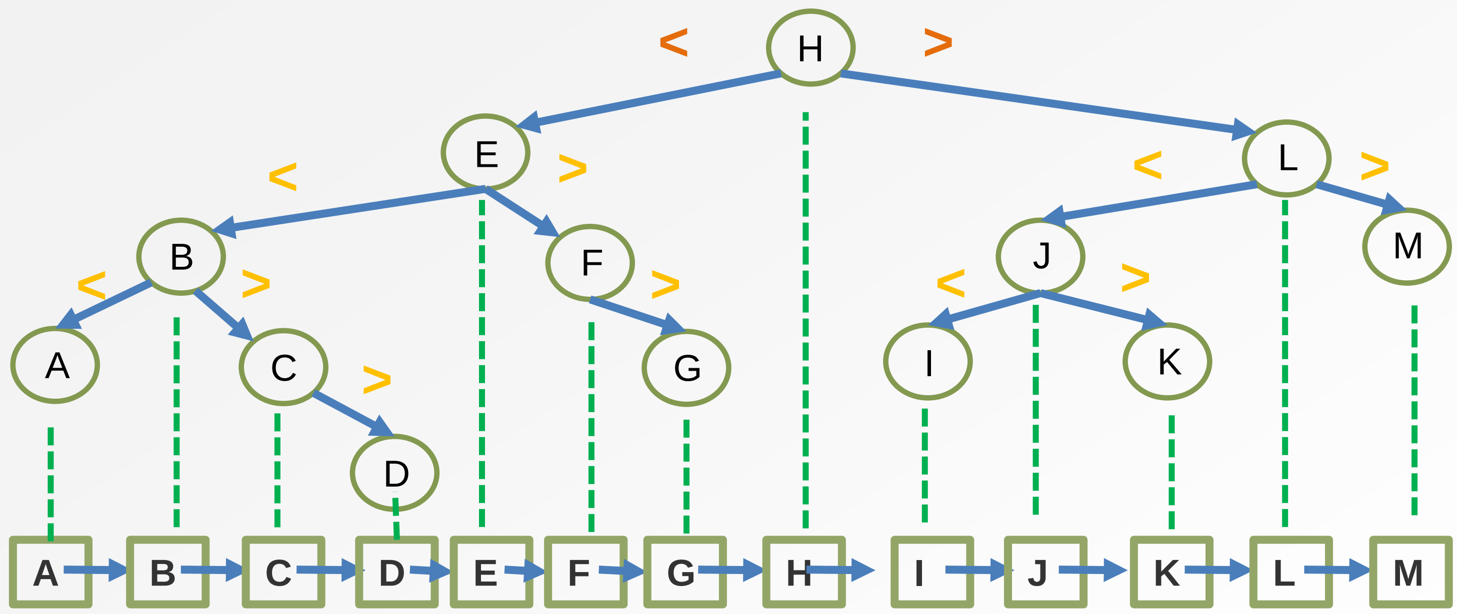


Output:

A B C D E F G H I

?

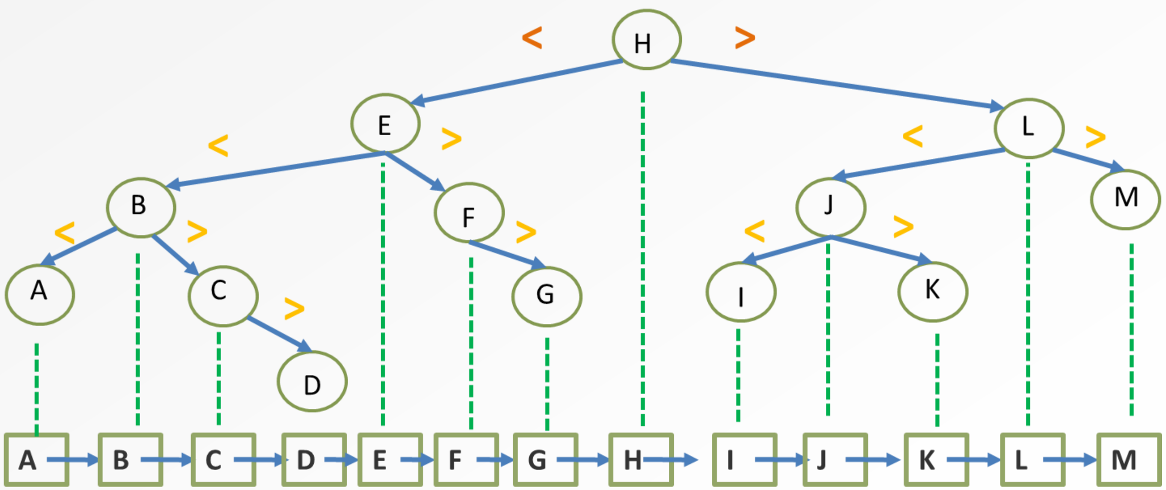
MAPPING: TREE(IN-ORDER) → LIST



- 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.**

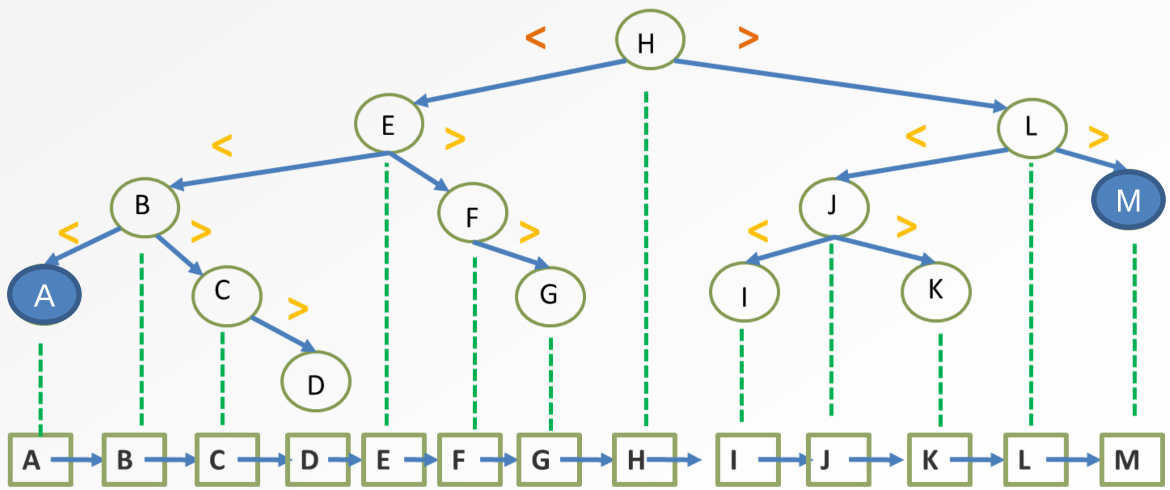
FEATURES

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



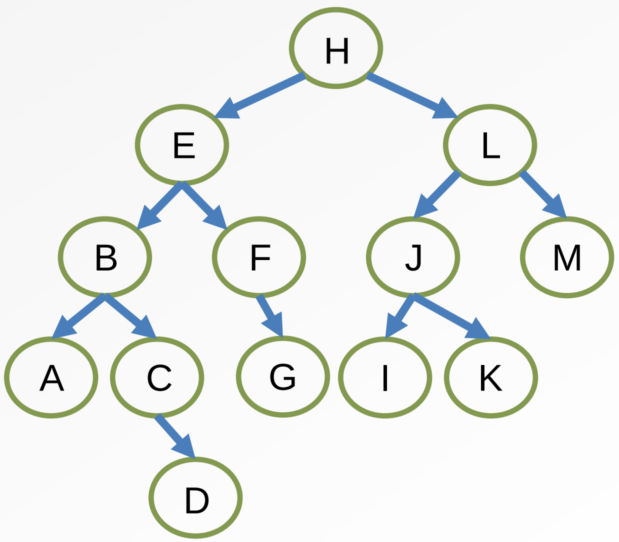
FEATURES

- 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



OUTLINE

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



BINARY SEARCH TREE(BST)

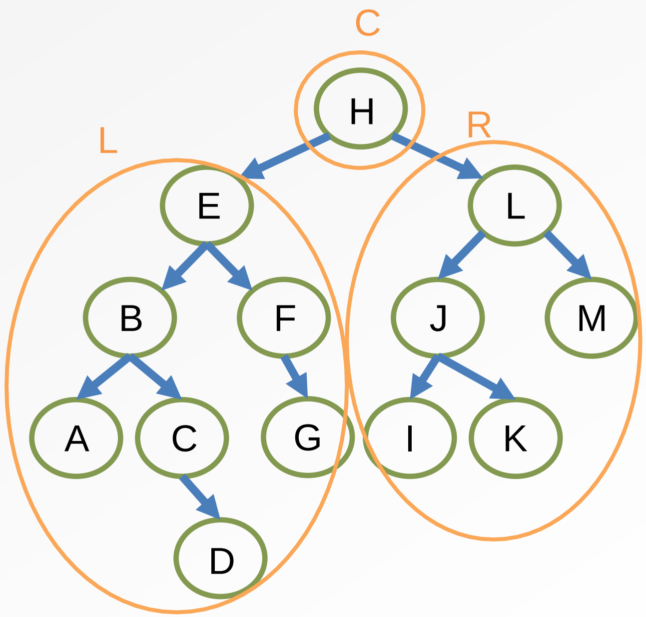
- BSTs are a special form of BT

- BST rule:

  At every node C,

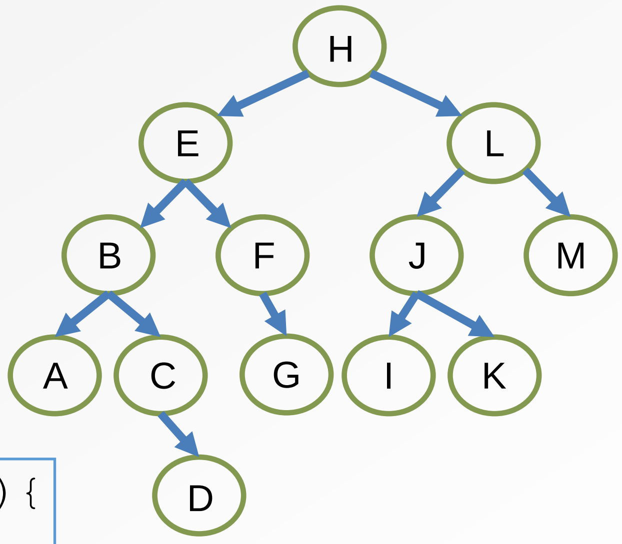
  L < C < R, where

  - 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);
}
```

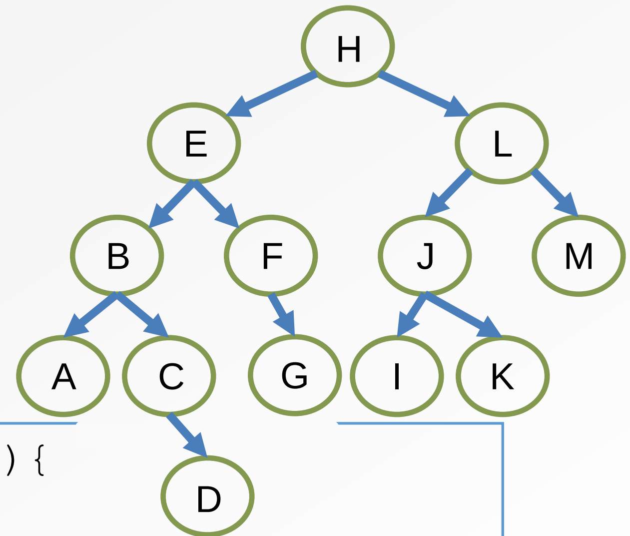
Do something with the current node's data

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

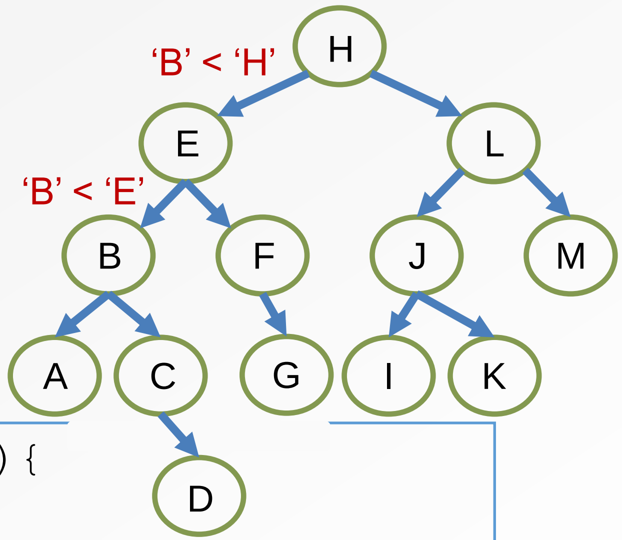
Do something with the current node's data

Visit the left child node

Visit the right child node

- Check the traversal pattern for

  BSTT(root, 'B')

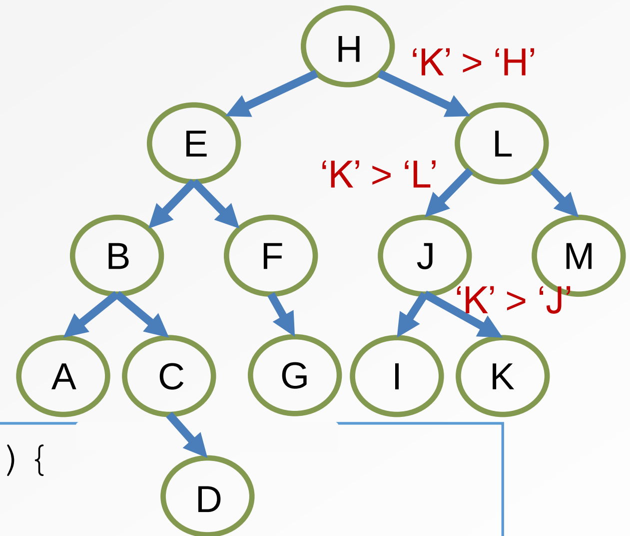



```
void BSTT(BTNode *cur, char c){
    if (cur == NULL) return;
    if (c==cur->item)
    { printf("found!\n"); return;}
    if (c < cur->item)
       BSTT(cur->left,c);
    else
       BSTT(cur->right,c);
}
```

BST TRAVERSAL (BSTT)

- Check the traversal pattern for

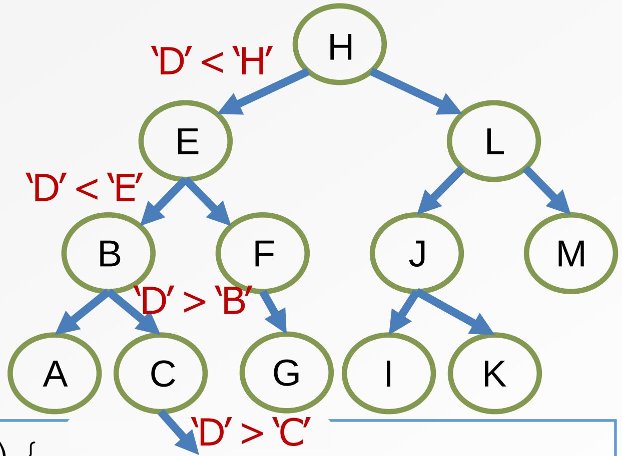
  BSTT(root, 'K')




```
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)

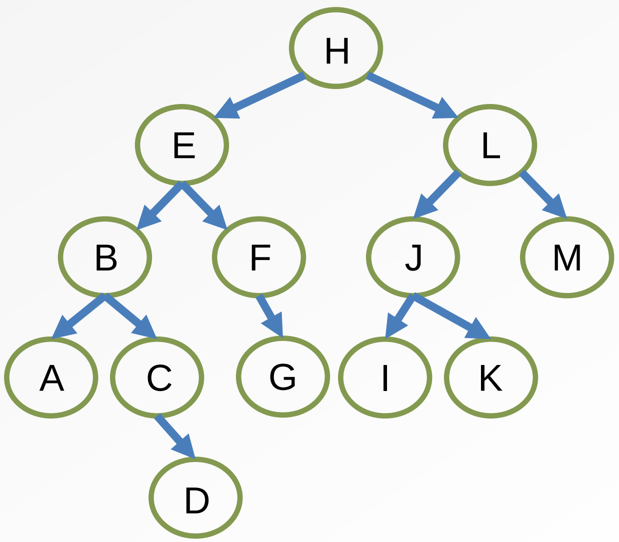
- What if the item doesn't exist?
- If we remove node 'D', and then check the traversal pattern for **BSTT(root, 'D')**




```
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);
}
```

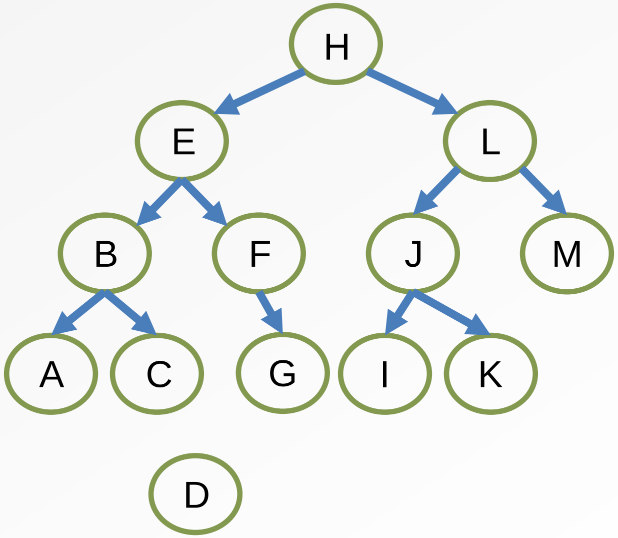
OUTLINE

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



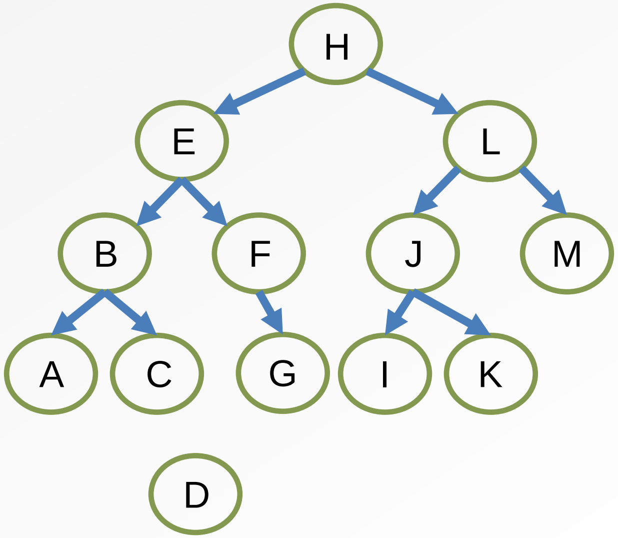
INSERTING A NODE INTO A BST

- Given an existing BST, an insertion operation must result in a BST
- How do we know where to place a new node ‘D’?



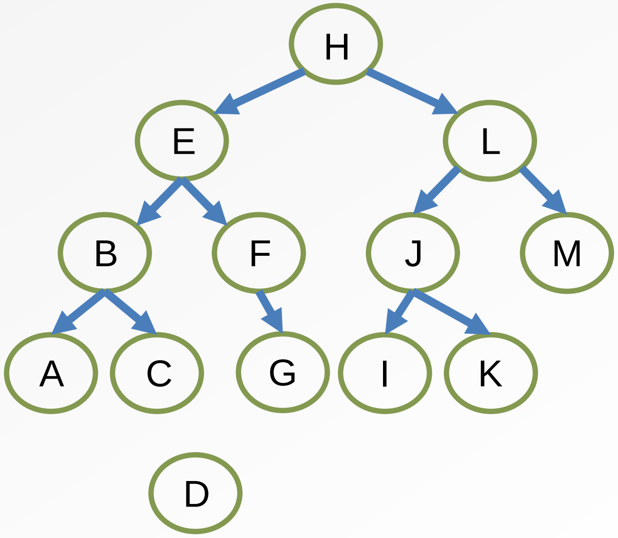
INSERTING A NODE INTO A BST

- Key point:
  - Given an existing BST and a new value to store, there is always a unique position for the new value



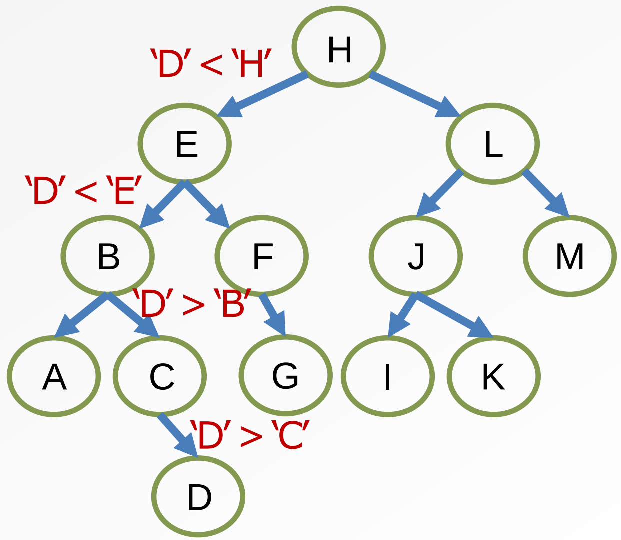
INSERTING A NODE INTO A BST

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



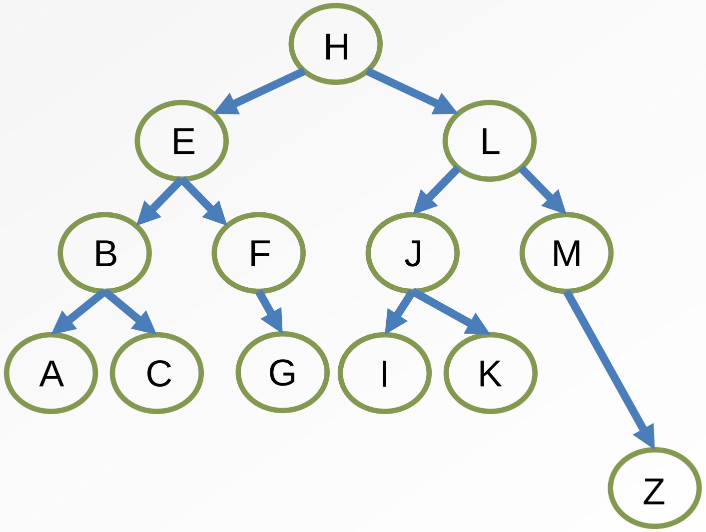
INSERTING A NODE INTO A BST

1. Use BSTT(root, ‘D’) to get to the correct empty location to insert ‘D’
2. Add the new node ‘D’



INSERTING A NODE INTO A BST

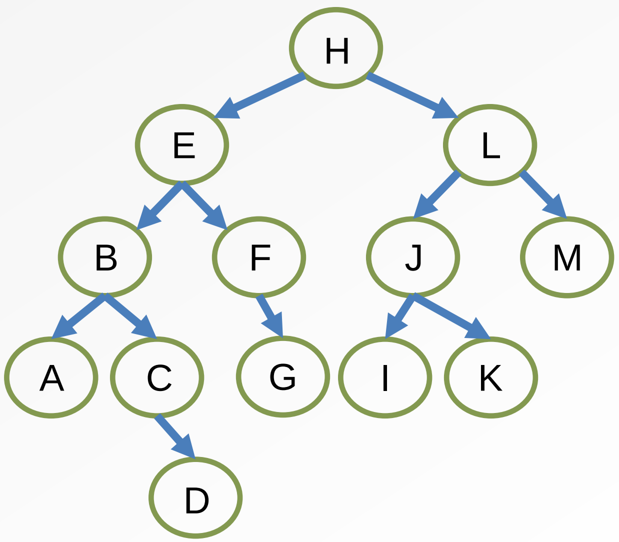
- Node insertion is relatively simple!
- Further exercise: Try Inserting ‘Z’



OUTLINE

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

After removal, the tree is still a BST



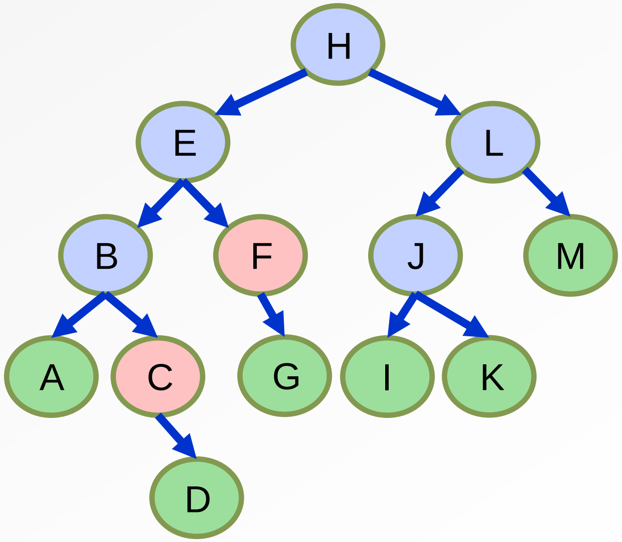
REMOVING A NODE FROM A BST

- 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

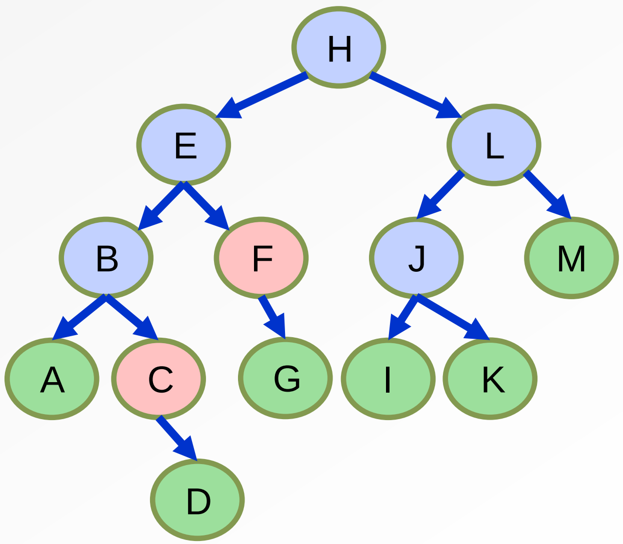
REMOVING A NODE FROM A BST

- 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



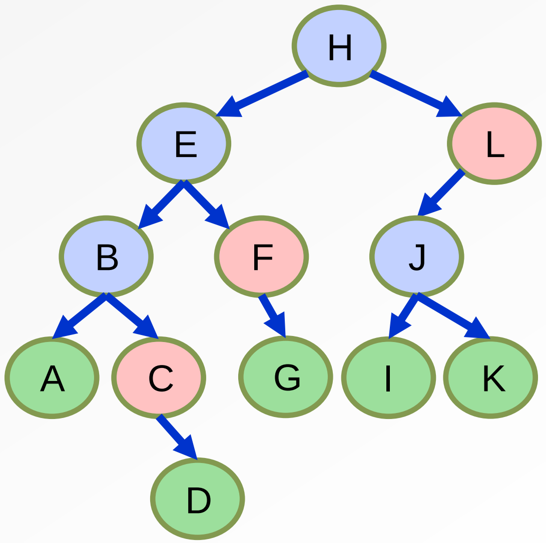
- Remove node X - a bit tricky
- 3 cases:

  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



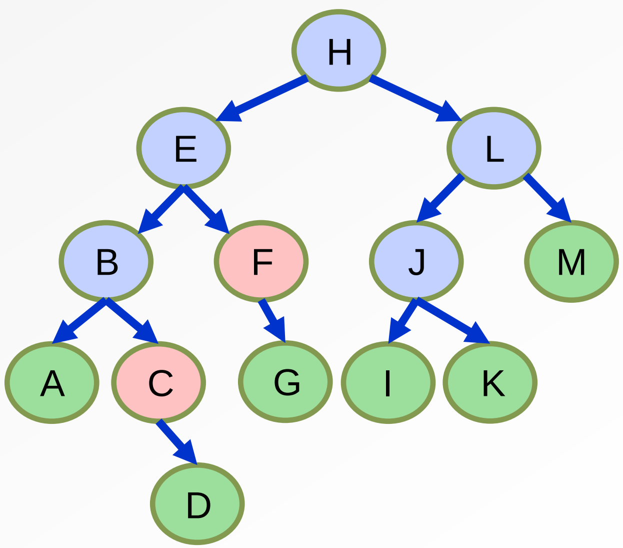
REMOVING A NODE FROM A BST

- 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**



WHAT IS THE SUCCESSOR OF X?

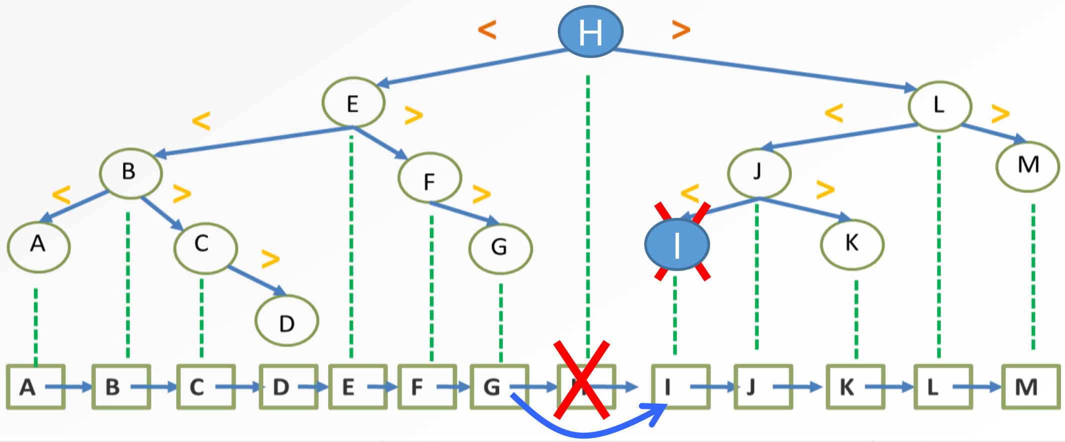
Replacing a node with its in-order successor ensures that the BST rule (L<C<R) is maintained

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

Successor is:

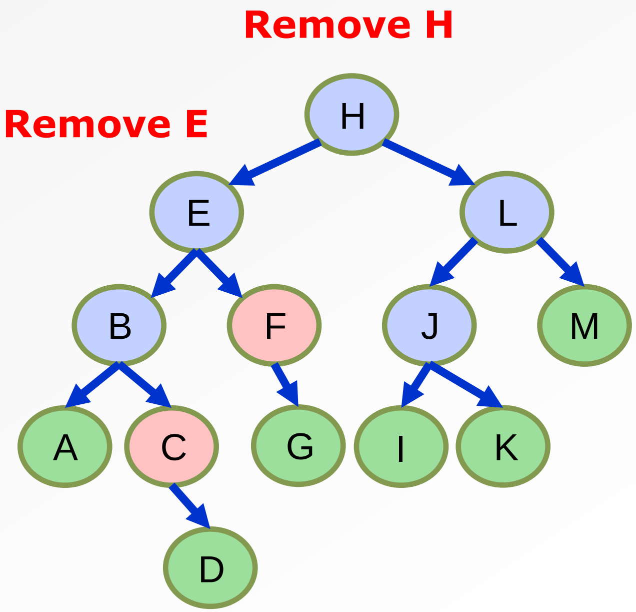
- **The node immediately after it in the sorted list,** or
- **The next node visited using an in-order traversal**

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 the minimum in its right subtree, so the successor should not have left child.

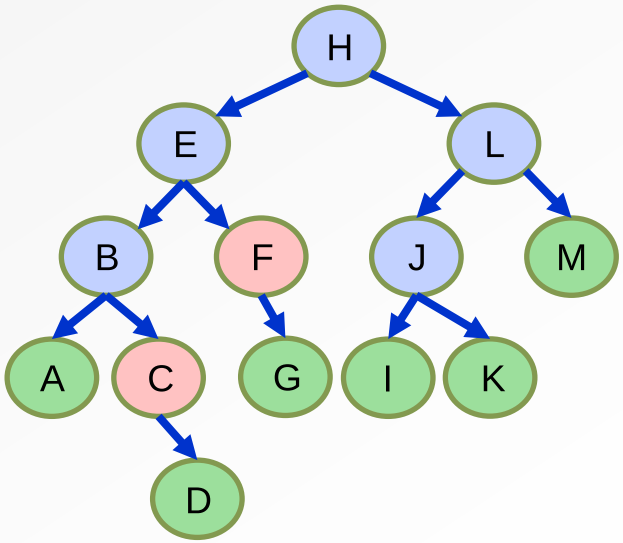
  It might have no child(case 1) or one right child(case 2).

- **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?

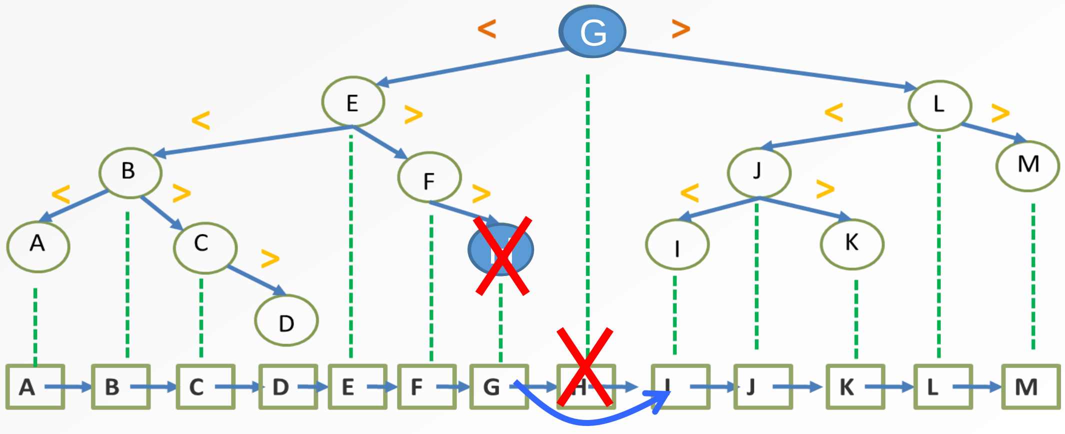
Replacing a node with its in-order **predecessor** ensures that the BST rule (L<C<R) is maintained

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 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?