



**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**

## **CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS**

### **Binary Search Trees**

**College of Engineering**

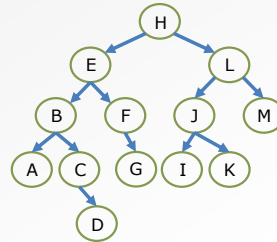
School of Computer Science and Engineering

## OUTLINE

- **Binary Search Trees (BST)**

- BST Operations:

- Traversal
- Inserting a node
- Removing a node



## BINARY SEARCH TREE(BST)

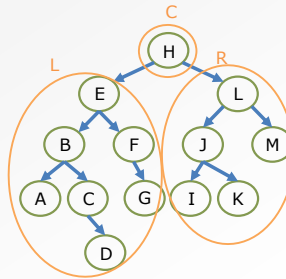
- BSTs are a special form of BT

- **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 -  $L < C < R$

## BINARY SEARCH TREE

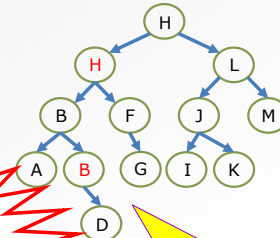
- BSTs are a special form of BT

- At every node C,

$L \leq C \leq R$ , where

- C is the data in the current node
- L represents the left child
- R represents the right child

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



This is not a BST!

Binary Search Tree cannot have duplicate nodes.

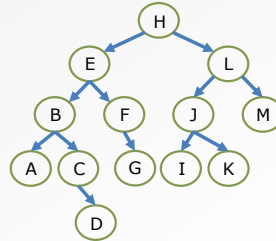
## OUTLINE

- Binary Search Trees (BST)

- BST Operations:

- **Traversal**

- Inserting a node
    - Removing a node



## BINARY SEARCH TREE(BST)

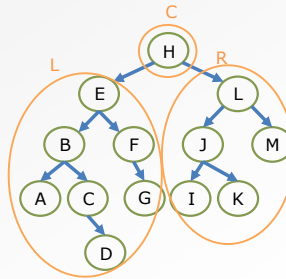
- BSTs are a special form of BT

- **BST rule:**

At every node  $C$ ,

$L < C < R$ , where

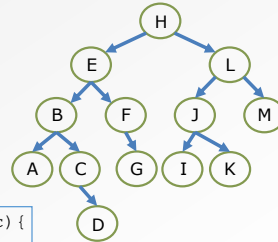
- $C$  is the data in the current node
- $L$  represents the data in any/ all nodes from  $C$ 's left subtree
- $R$  represents the data in any/all nodes from  $C$ 's right subtree



Binary Search Tree -  $L < C < R$

## 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

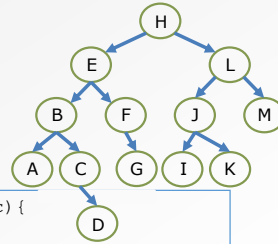
Visit the left child node

Visit the right child node

Binary Search Tree -  $L < C < R$

## 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

Using the tree traversal template we can construct the code for BST Traversal.

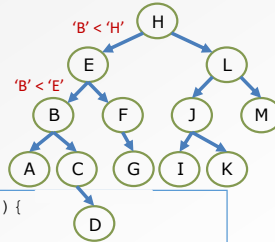
- After defining the function 'BSTT' with a bnode typed pointer and a character typed variable we first check whether the current node is NULL or NOT NULL..

If the current node is not what we required we check whether C is less than the current item. If it is less than current item, it should be in current item's left subtree. Therefore we have to move to the left 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);  
}
```

Content Copyright Nanyang Technological University

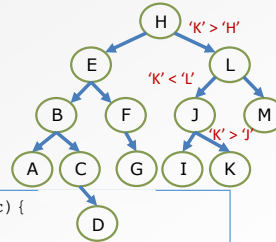
9

For an example, if you want search for 'B' in the given binary search tree;

- You start with 'H' which is the root node.
- Check whether 'H' is what we are searching for. Since we are searching for 'B', 'H' is not the correct node.
- Then we check whether 'B' is less than 'H'. 'B' comes before 'H' in the alphabet. Therefore 'B' < 'H'. Then we move to the left subtree of 'H', which is 'E'.

## 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);  
}
```

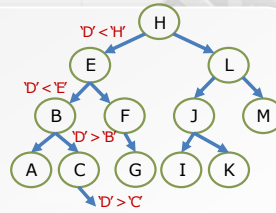
Same process should follow to search node 'K'.

This time 'K' comes after 'H' in the alphabet 'K' is large than 'H'.

Therefore we have to check with right subtree of 'H' this time.

## 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);  
}
```

Content Copyright Nanyang Technological University

11

### What if we remove node 'D' and search for it?

We start from the root node 'H' and follow the discussed process until 'C'.

Now 'D' is larger than 'C'. Therefore we move from 'C' to its right child node.

Now the pointer points at an empty node because we have already removed 'D'.

## OUTLINE

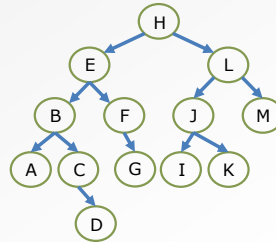
- Binary Search Trees (BST)

- BST Operations:

- Traversal

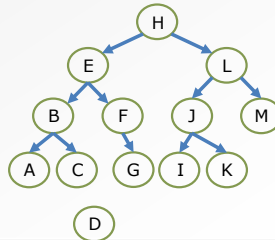
- **Inserting a node**

- 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



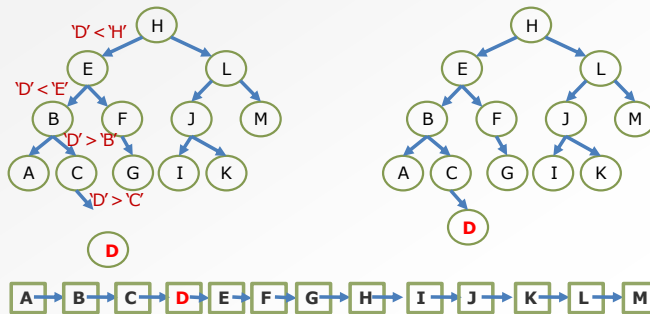
Content Copyright Nanyang Technological University

13

For the given binary search tree in the slide we try to insert node 'D' and still maintain the tree as a binary search tree.

## INSERTING A NODE INTO A BST

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



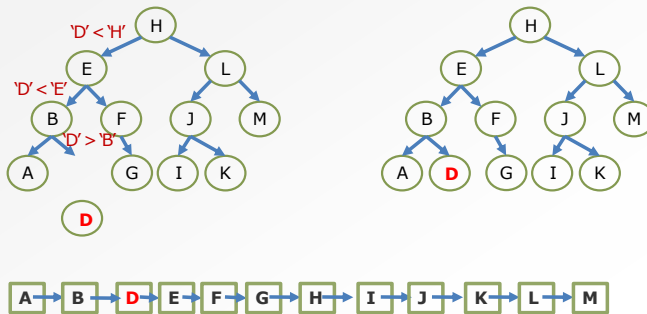
Content Copyright Nanyang Technological University

14

Before inserting the node we use Binary Tree Traversal Template to identify the correct empty location for the node.

## INSERTING A NODE INTO A BST

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



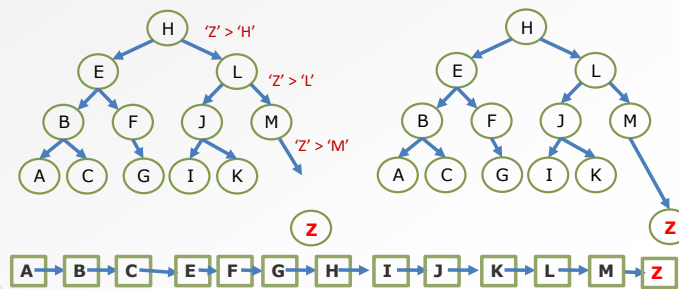
Content Copyright Nanyang Technological University

15

Before inserting the node we use Binary Tree Traversal Template to identify the correct empty location for the node.

## INSERTING A NODE INTO A BST

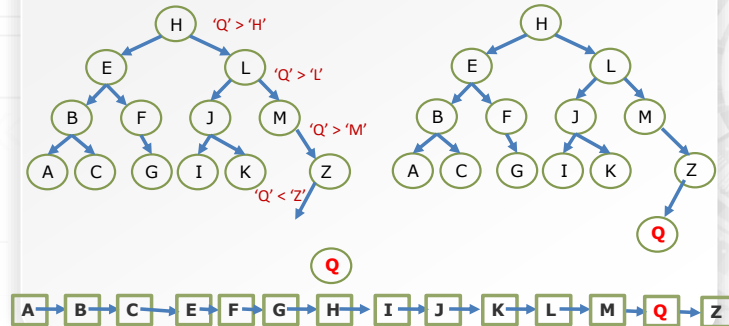
- Node insertion is relatively simple!
- Further exercise: Try Inserting 'Z'





## INSERTING A NODE INTO A BST

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



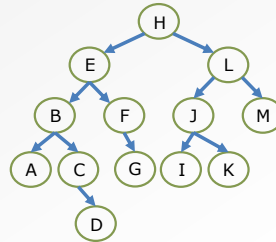
## OUTLINE

- 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:

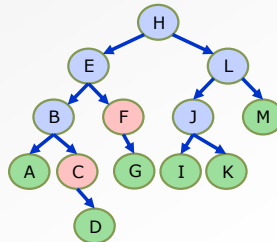
- Remove x

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



There are 3 cases we should consider when we try to remove a node from a BST.

## REMOVING A NODE FROM A BST

- Remove node X - a bit tricky

- 3 cases:

1. **x has no children:**

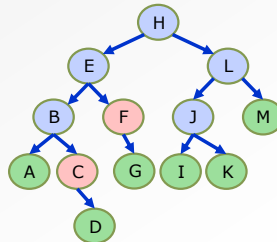
- Remove x

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



### Case 1: X has no children

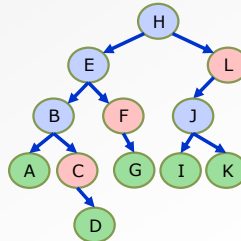
If the node which we are trying to remove has no children, or in other words, if we are trying to remove a leaf node, we can just remove the node and the remaining tree is still a BST.

## 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



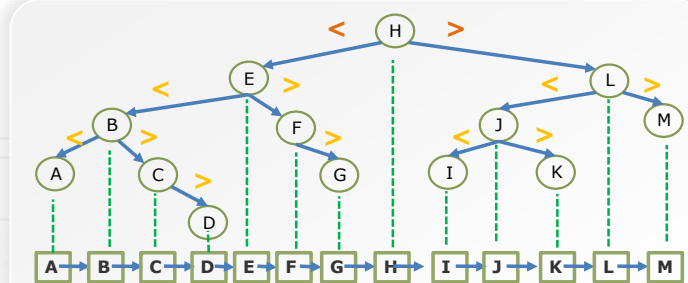
### Case 2: If X has one child Y

The nodes coloured in 'pink' in the slide has only one child node.

For an example if we want to remove node 'F' from the BST, we can replace node 'F' with node 'G' because 'F' has only one child which is 'G'.

Same logic can be applied to node 'L' and node 'C'. 'L' can be replaced with 'J' and 'C' can be replaced with 'D'.

## MAPPING: TREE(IN-ORDER) → LIST

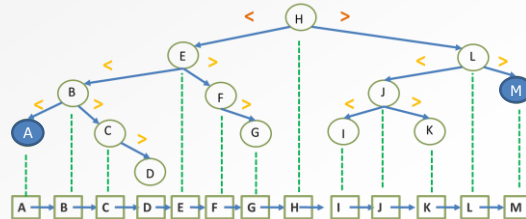


- If we draw the BST carefully:
  - **Left subtree on the left side of the current node;**
  - **Right subtree on the right side of the current node;**
- Mapping to X-axis will produce **a sorted list.**

We get a sorted linkedlist if we flatten a binary search tree because we can map binary search tree into a linkedlist as shown in the slide.

## 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:

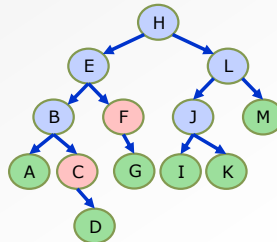
- Remove x

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**



### Case 3: When X has two children

This case is little bit complicated.

## WHAT IS THE SUCCESSOR OF X?

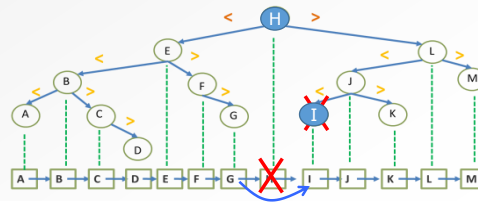
Replacing a node with its in-order successor ensures that the BST rule ( $L < C < R$ ) is maintained

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.



Content Copyright Nanyang Technological University

26

What is successor of X?

For an example, in the given tree, the successor of 'H' is 'I', because 'I' comes immediately after 'H' in the sorted list. 'I' is the node visited after 'H' using in-order traversal also.

Now we have to swap 'H' with 'I' if we want to remove 'H', because if we replace the node with its in-order successor, it ensures the BST rule ( $L < C < R$ ) is maintained.

Therefore first we swap 'H' with 'I' and then 'H' become a leaf node. Now we can apply case 1 which we discussed and remove 'H'.

Now the BST rule is maintained as before.

## REMOVING A NODE FROM A BST

- Remove node X - a bit tricky

- 3 cases:

1. x has no children:

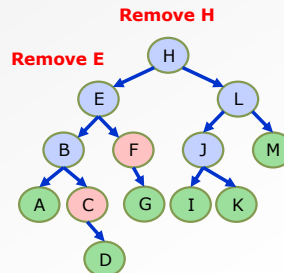
- Remove x

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



As discussed in the previous slide, if we want to remove node X which has two children;

- First we swap X with its successor
- Then perform case 1 or case 2 to remove X.

## 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:

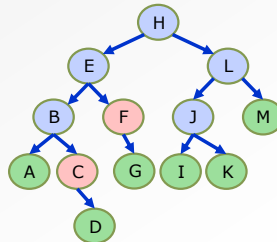
- Remove x

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**



- 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.

## WHAT IS THE SUCCESSOR OF X?

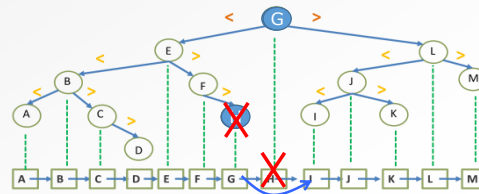
Replacing a node with its in-order **predecessor** ensures that the BST rule ( $L < C < R$ ) is maintained

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.



Content Copyright Nanyang Technological University

30

We can swap X with its predecessor instead of the successor as well.

## 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?