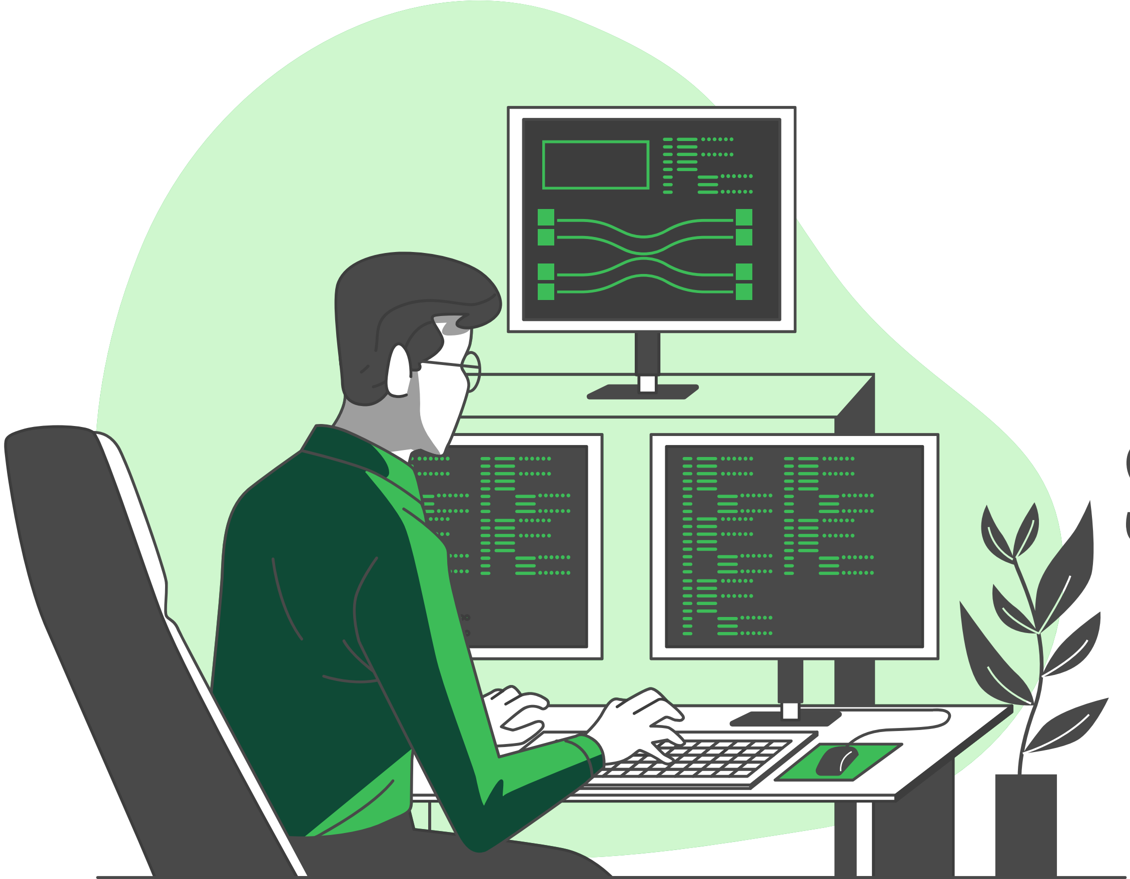


Binary Search Tree

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

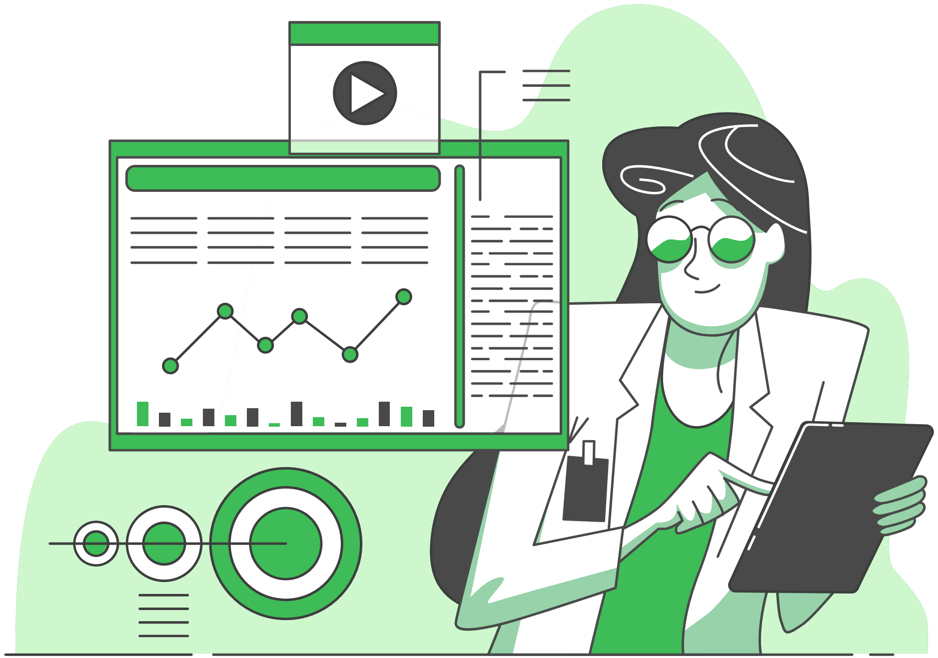
Table of Contents

01

Binary Search Tree

Searching in an ordered tree

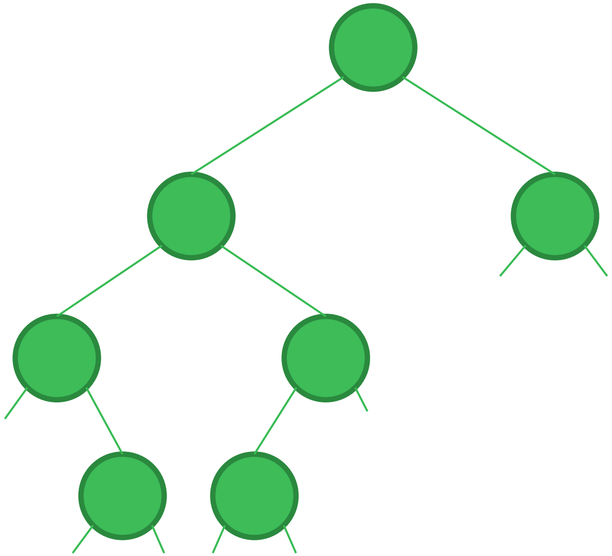
...



01
Binary Search Tree

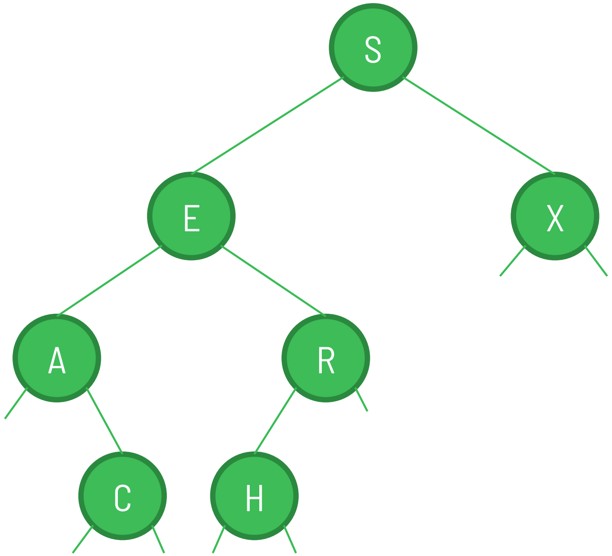
Searching in an ordered tree

Binary Tree



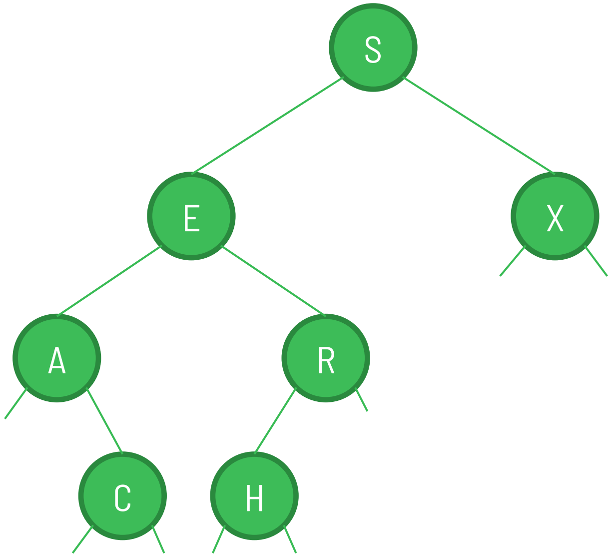
- **Recursive definition:** A Binary Tree is either a null link or a root with a left link and a right link that each point to Binary Trees.
- A root has one parent (except the overall tree root) and two children (left and right links).

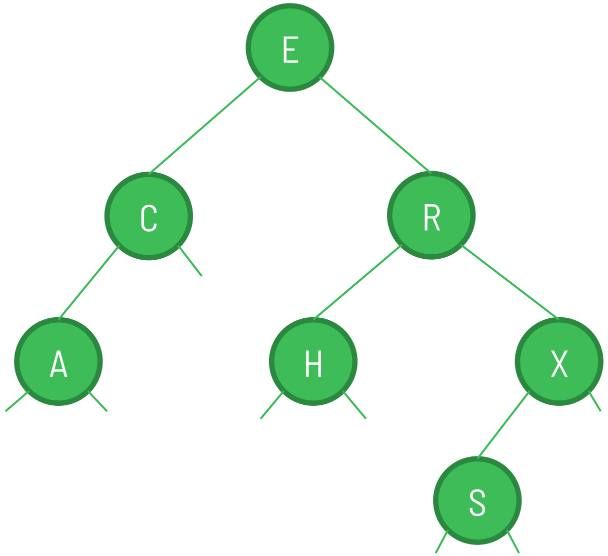
Binary Search Tree



- A Binary Search Tree is an ordered Binary Tree such that any root is greater than any element in its left subtree and less than any element in its right subtree.

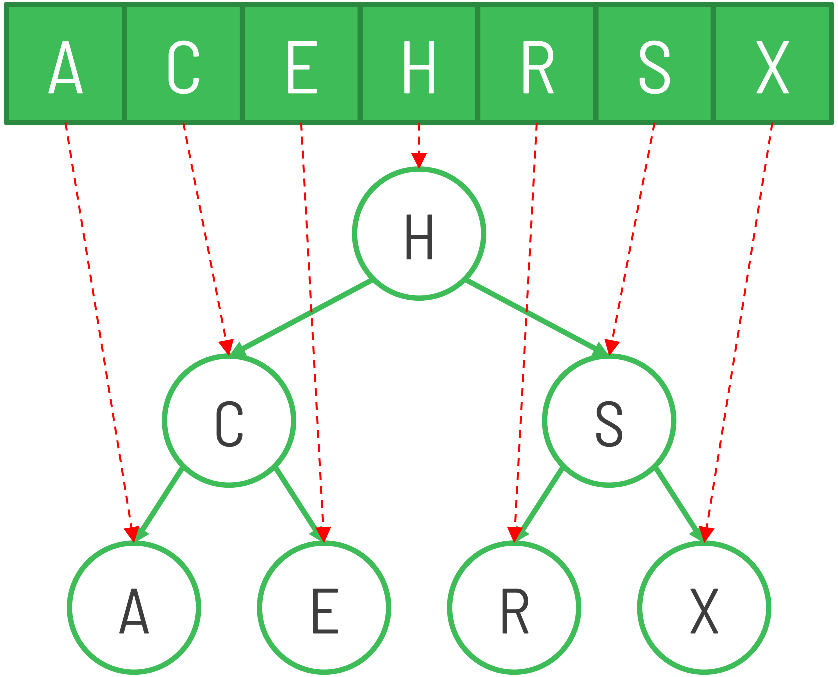
Examples





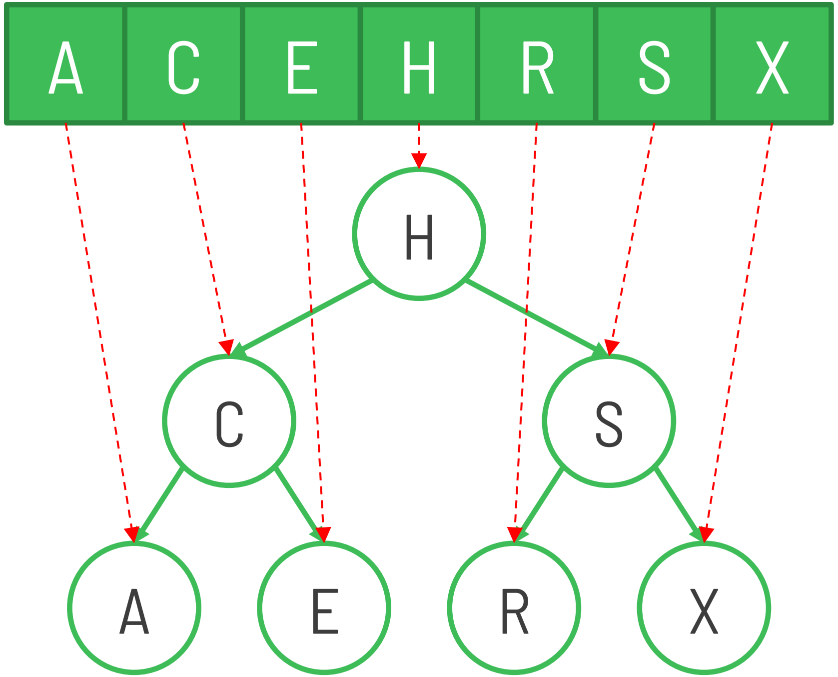
Sorted Array to BST

Let A = [A, C, E, H, R, S, X].
Transform the array into a Binary Search Tree.



Sorted Array to BST

```
algorithm generateBST(A:array, l:ℤ≥0, r:ℤ≥0)
   if r < l then
      return null
   end if

   m ← floor((l + r) / 2)
   let root be a new node
   root.item ← A[m]
   root.left ← generateBST(A, l, m - 1)
   root.right ← generateBST(A, m + 1, r)

   return root
end algorithm
```

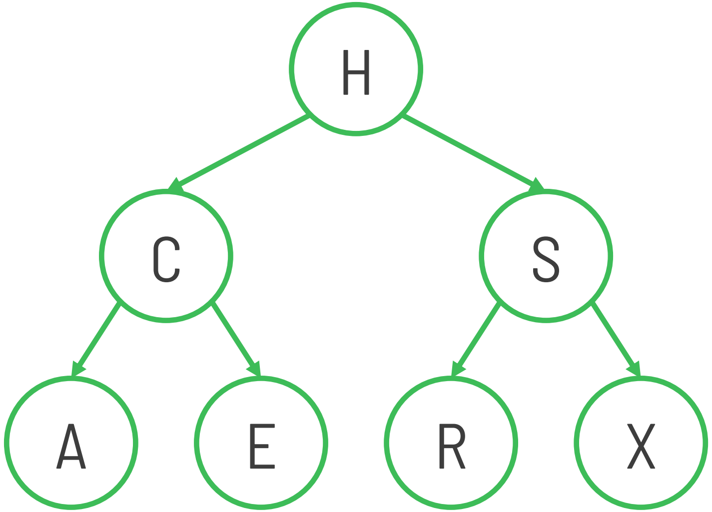


Insertion in a BST

```
algorithm insert(root:node, x:item) → node

   if root is null then
      let n be a new node
      n.item ← x
      n.left ← null
      n.right ← null
      return n
   end if

   if x <= root.item then
      root.left ← insert(root.left, x)
   else if x > root.item
      root.right ← insert(root.right, x)
   end if

   return root
end algorithm
```

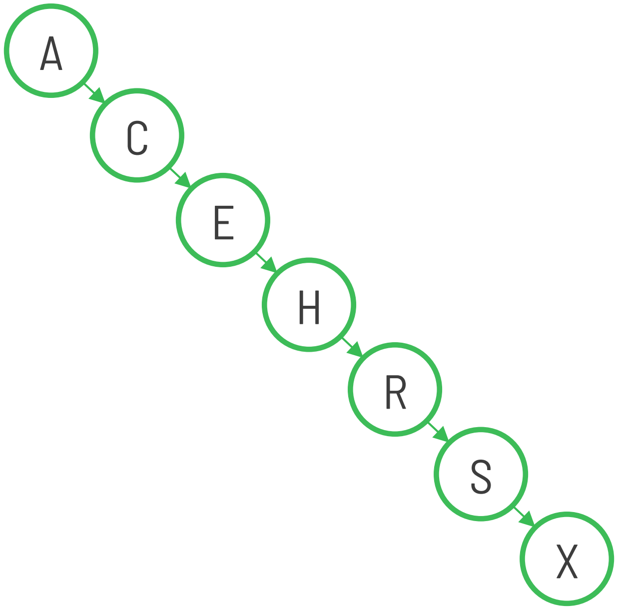
Insertion Example

```
let A be [H, C, E, S, R, A, X]
let n be the size of A
B ← null

for i from 0 to n-1 do
   B ← insert(B, A[i])
end for
```



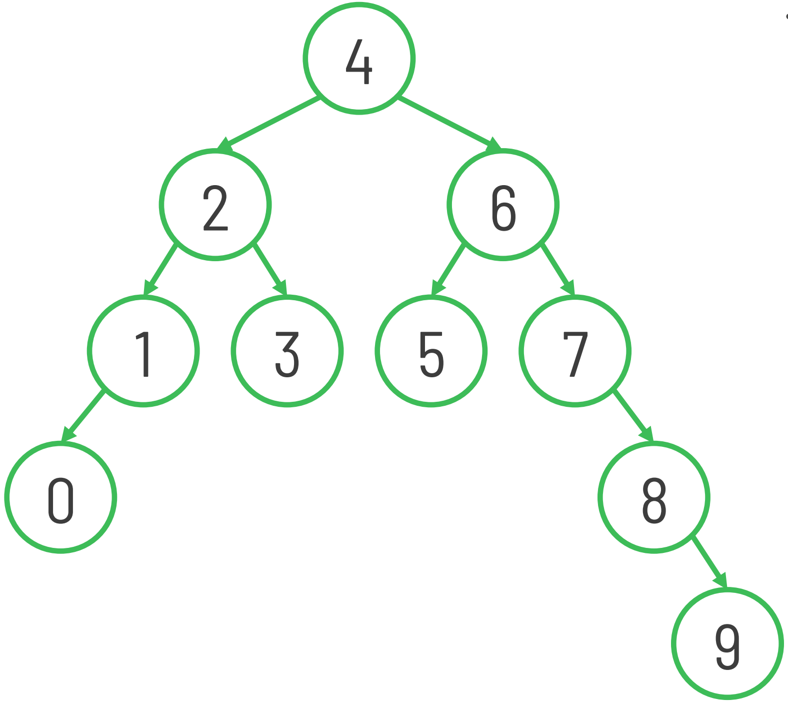
Insertion Example

```
let A be [A, C, E, H, R, S, X]
let n be the size of A
B ← null

for i from 0 to n-1 do
   B ← insert(B, A[i])
end for
```

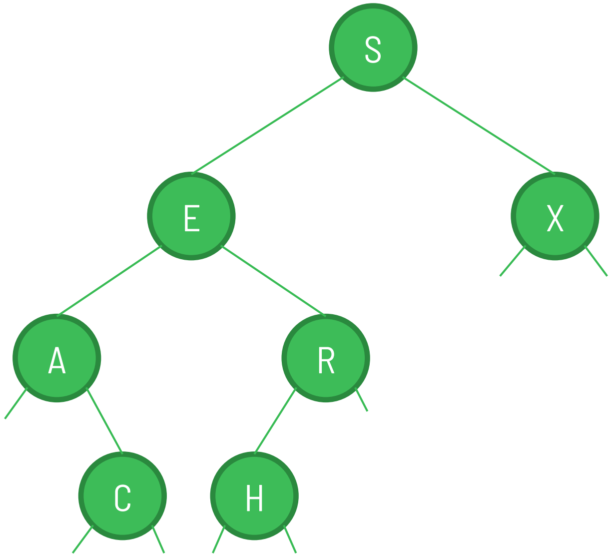


Insertion Example

```
let A be [4, 2, 6, 1, 3, 5, 7, 0, 8, 9]
let n be the size of A
B ← null

for i from 0 to n-1 do
   B ← insert(B, A[i])
end for
```

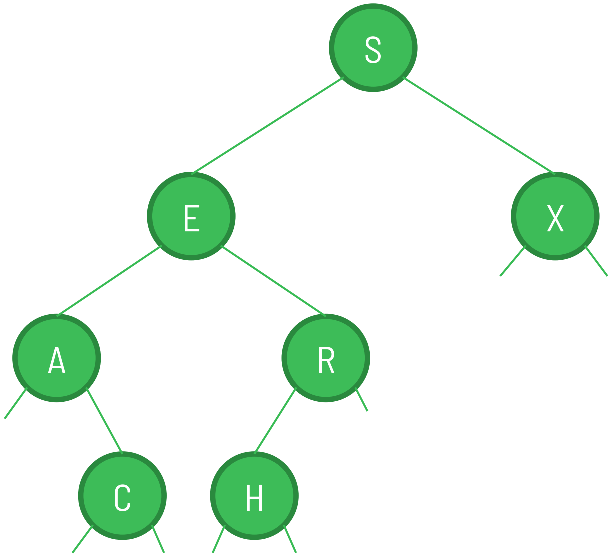


Search




```
algorithm search(root:node, x:item) → node
   if root is null then
      return null
   end if

   if x = root.item then
      return root
   end if

   if x < root.item then
      return search(root.left, x)
   end if

   return search(root.right, x)
end algorithm
```

Delete

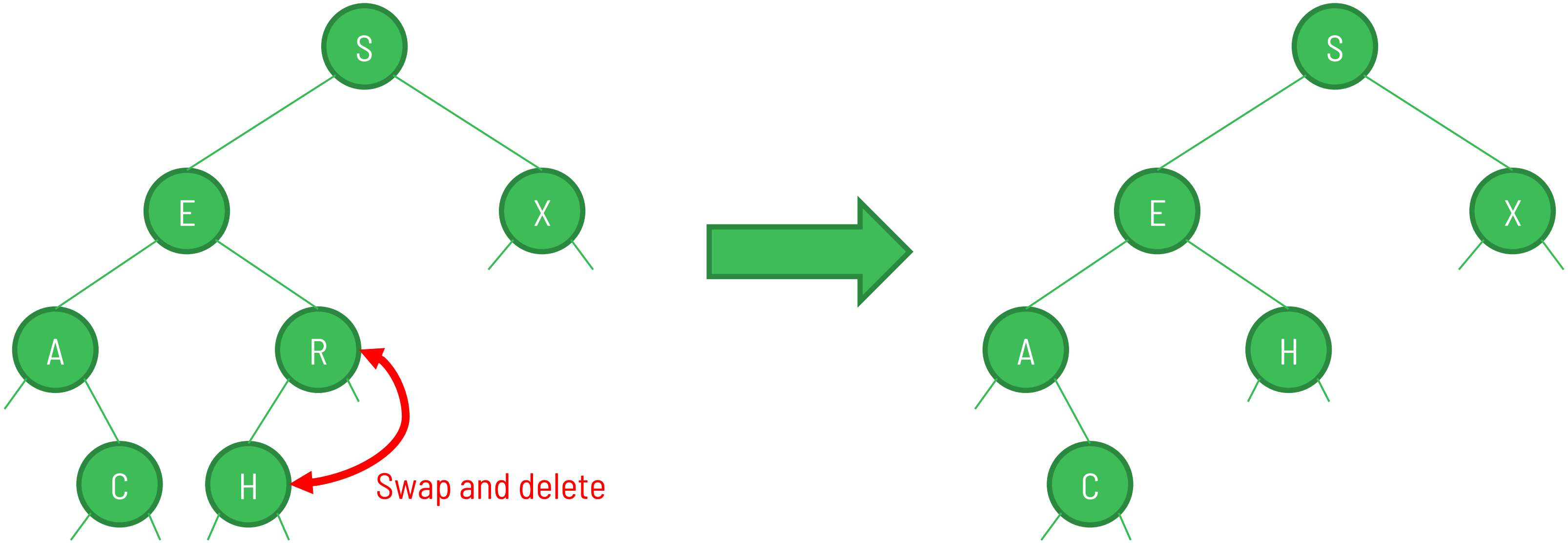


- **Option 0:** item doesn’t exist in the tree.
- **Option 1:** item exists and it has no children.
- **Option 2:** item exists and it has one child.
- **Option 3:** item exists and it has two children.

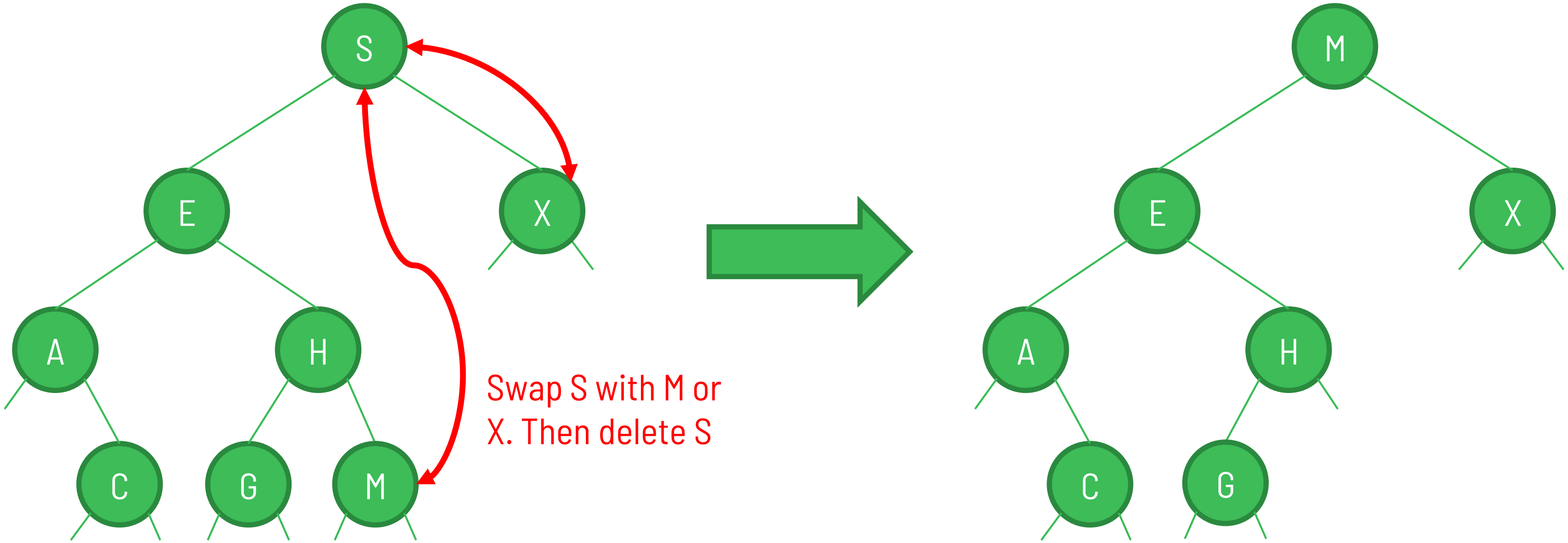
Delete in a BST

```
algorithm delete(root:node, x:item) → node

   if root is null then
      return null
   end if

   if x < root.item then
      root.left ← delete(root.left, x)
   else if x > root.item
      root.right ← delete(root.right, x)
   else

      if root.left is null then
         return root.right
      else if root.right is null then
         return root.left

      successor ← findmin(root.right)
      root.item ← successor.item
      root.right ← delete(root.right, root.item)
   end if

   return root
end algorithm
```

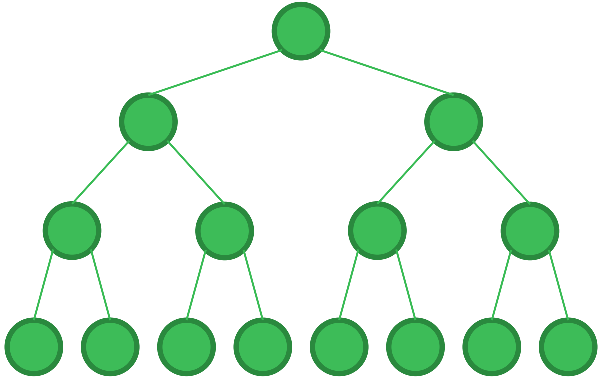
Delete R



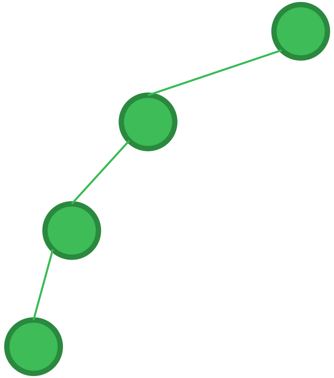
Delete S



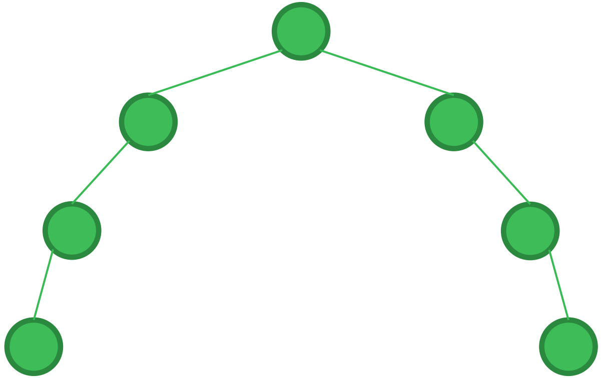
Remember: Balancing Matters



Balanced
Height $\in \Theta(n)$



Unbalanced
Height $\in \Theta(n)$



Unbalanced
Height $\in \Theta(n)$

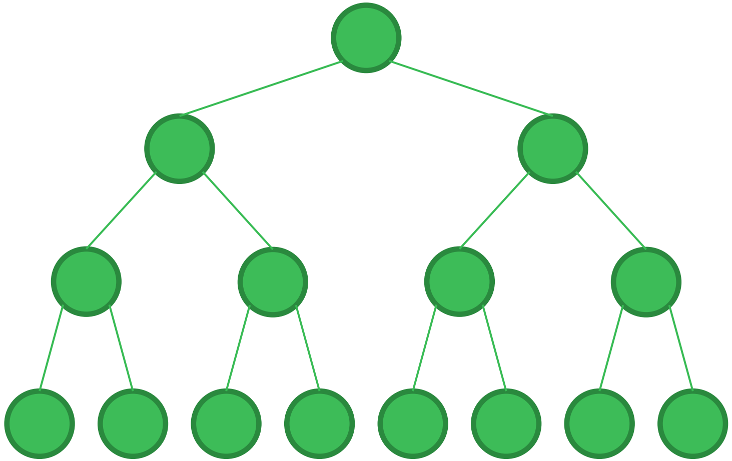
Runtime Analysis

For an unbalanced Binary Search Tree of size n:

1. Insert an item $\in O(n)$
2. Insert all items $\in O(n^2)$
3. Search $\in O(n)$
4. Delete $\in O(n)$

For a balanced Binary Search Tree of size n:

1. Insert an item $\in O(\log(n))$
2. Insert all items $\in O(n \log(n))$
3. Search $\in O(\log(n))$
4. Delete $\in O(\log(n))$



Can we **insert/delete** items in a Binary Search Tree such that we always have a **balanced binary tree**?

END

Do you have any questions?