Binary Search Tree

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

Simulation Weblink

https://www.cs.usfca.edu/~galles/visualization/BST.html

1 Motivation

In the last couple of lectures, we have studied two important algorithms and data structures – *binary search* and *linked list*. In the former, we can search an n-length sorted array in $O(\log n)$ time to detect whether a *key* is present in the array or not. However, being an array-based algorithm, we cannot support updates (i.e., insertion or deletion) efficiently. Specifically, in the worst-case, we need $O(n)$ time to insert or delete a single element, even at the front or end of the array. On the other hand, linked list supports fast (in $O(1)$ time) insertion and deletion at the front or at the end of the list. However, since linked list does not allow direct access to an element in an arbitrary position, binary search is no longer possible – searching for *key*, even on a sorted linked list having n nodes, takes $O(n)$ time in the worst case. This brings us to the following important question – *can we design a data structure that supports fast searching as well as fast updates?*

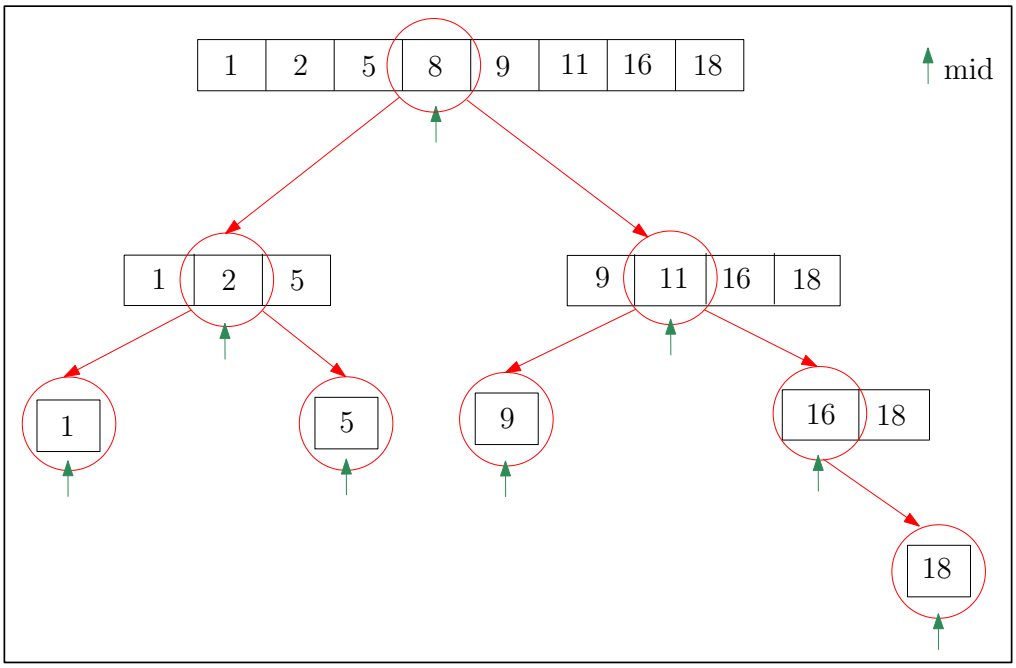
Using a balanced binary search tree, we can support all operations (search, insert, and delete), in worst case $O(\log n)$ time. In this course, we will study the classical (unbalanced) binary search tree. Search and update operations are still $O(n)$ in the worst case, however, for most cases, it performs decently well.

Slightly modified search problem: Design a data structure that stores a set $\mathcal{S}$ of integers and supports the following operations:

- $search(val)$: returns $true$ if $\mathcal{S}$ contains val, else it returns $false$
- $insert(val)$: adds val to the set $\mathcal{S}$ if already not present
- $delete(val)$: deletes val from the set $\mathcal{S}$, provided it exists

2 Definition

Let us closely look at binary search and see what it does. At each iteration, we are picking mid of the relevant array. Then, we are either returning mid (in case $A[mid] = key$), or searching the elements to the right of mid (in case $key > A[mid]$), or searching the elements to the left of mid (in case $key < A[mid]$). Thus, based on mid, we are dividing our relevant array into two parts – elements to the right and to the left of mid.



See an illustration on the next page. If we forget about the underlying array, and look at the *red circles and arrows*, we get a binary search tree.[1]

2.1 Terminologies

The *red circles* are called nodes (akin to nodes in linked list), and the *red arrows* connecting two nodes is called an *edge* (akin to next-link in linked list).

Each node has AT MOST two children (nodes) – *left child* and *right child*. Obviously, the left child (resp. right child) of a node is the one pointed to by the left arrow (resp. right arrow). Thus, 2 is the left child of 8, whereas 16 is the right child of 11. A node which has no children is called a *leaf node*. Thus, 9 is a leaf node. If x is a child of y, then we say y is the *parent* of x. A node which does not have a parent node is called the *root*. Thus, 16 is the parent of 18 and 8 is the root node. **Note** that the parent of a node (if exists) is unique and the root node is unique. Additionally, any node is a child of a single parent, i.e., we cannot have an edge from 5 to 9 in the figure above.

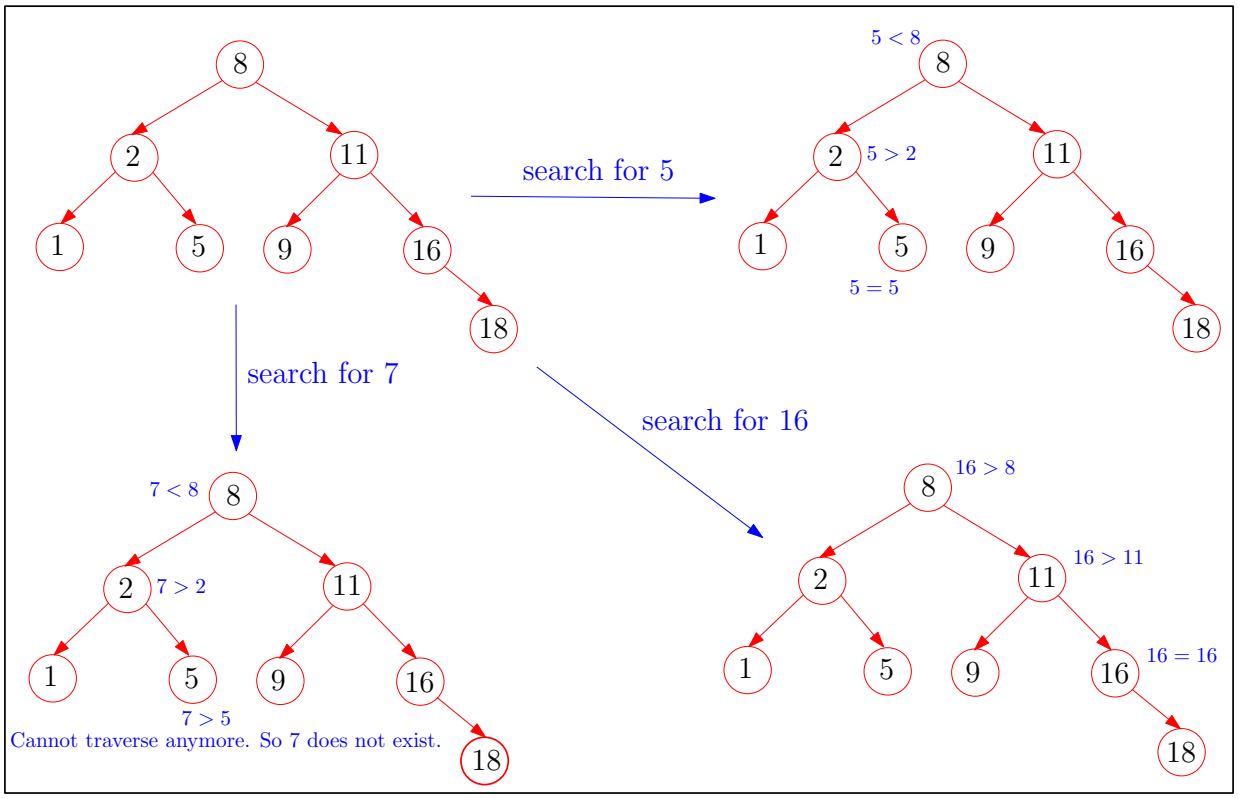
The subtree of a node x is the tree rooted at node x. Thus, the subtree of node 11 is the binary tree containing the nodes 9, 11, 16, and 18.

The height of the tree is the number of nodes on the longest path from a root to a leaf node. Thus, the height of the tree shown above is 4.

What we have defined so far is called a **binary tree**. Once, we study recursion, we will look at a more formal definition. Each node in the binary tree is associated with a value. Let us now define the important property that converts a *binary tree* into a *Binary Search Tree* (in short, BST).

BST Property: Consider a node x. Let x_L and x_R be the left and the right child of x respectively. Then the value of any node in the subtree of x_L is smaller than the value of x, whereas the value of any node in the subtree of x_R is bigger than the value of x.

[1] **Important:** This is NOT a binary search tree for the numbers of the array in left-to-right order. This is simply to convey that the underlying concept behind a binary search tree is binary search.



3 Searching *key*

Main Observation: Consider a node x. If value at x equals *key*, then key exists. If value at x is larger than *key*, then *key* can only be in the subtree of the left child of x. If value at x is smaller than *key*, then *key* can only be in the subtree of the right child of x.

Algorithm: Traverse from root comparing *key* with the value stored at a node to determine whether we should look at the right subtree or the left subtree. Terminate once we reach a node with value equaling *key*, or once we reach a node where we cannot traverse anymore.

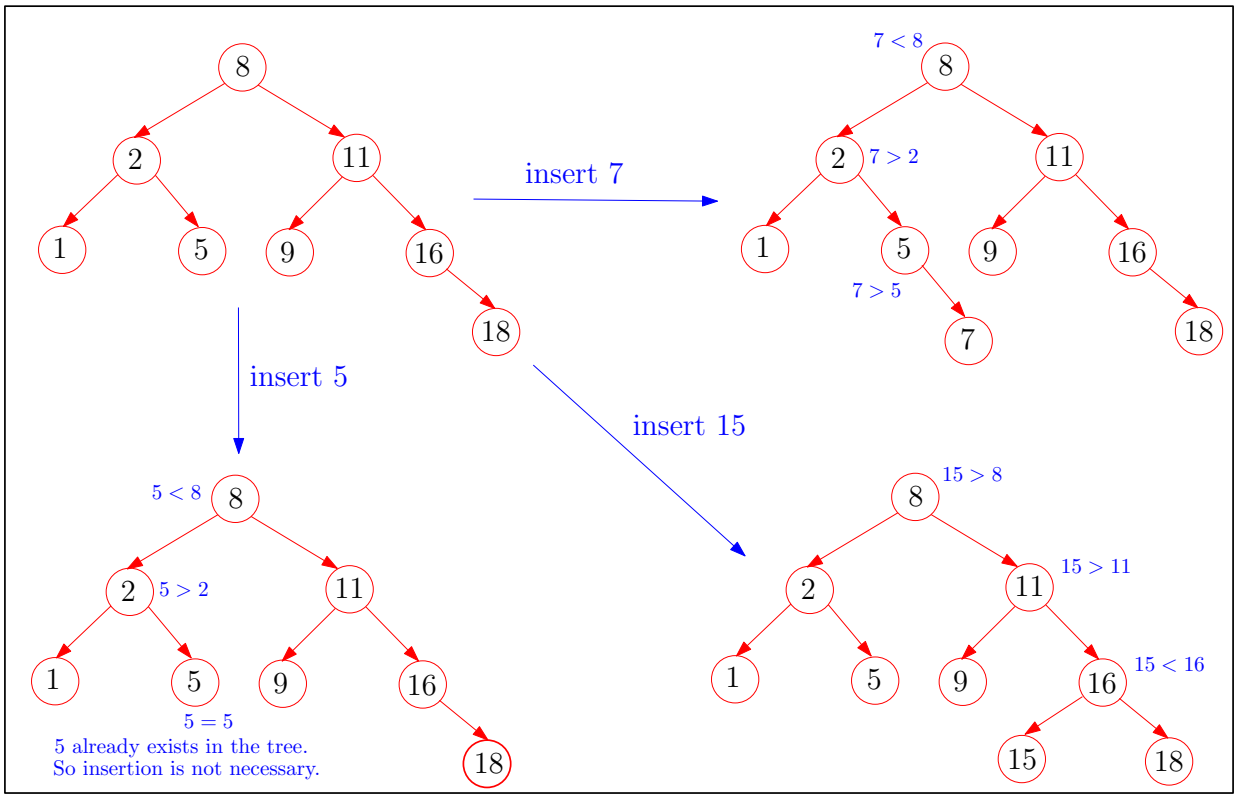
Refer to the figure above for illustration.

4 Inserting *val*

Main Observation: Consider a node x. If value at x equals *val*, then insertion is not necessary. If value at x is larger than *val*, then *val* should be inserted in the subtree of the left child of x. If value at x is smaller than *val*, then *val* should be inserted in the subtree of the right child of x.

Algorithm: Traverse from root comparing *key* with the value stored at a node to determine whether we should look at the right subtree or the left subtree. Terminate once we reach a node with value equaling *key* – insertion is not necessary in this case. Otherwise, we reach a node, say *parent*, where we cannot traverse anymore. If *val* is larger than the value stored in *parent*, then insert *val* as the right child of *parent*, else insert *val* as the left child of *parent*.

Refer to the figure on next page for illustration.



5 Deleting *val*

Deletion is arguably the most difficult task to carry out. It has the following steps:

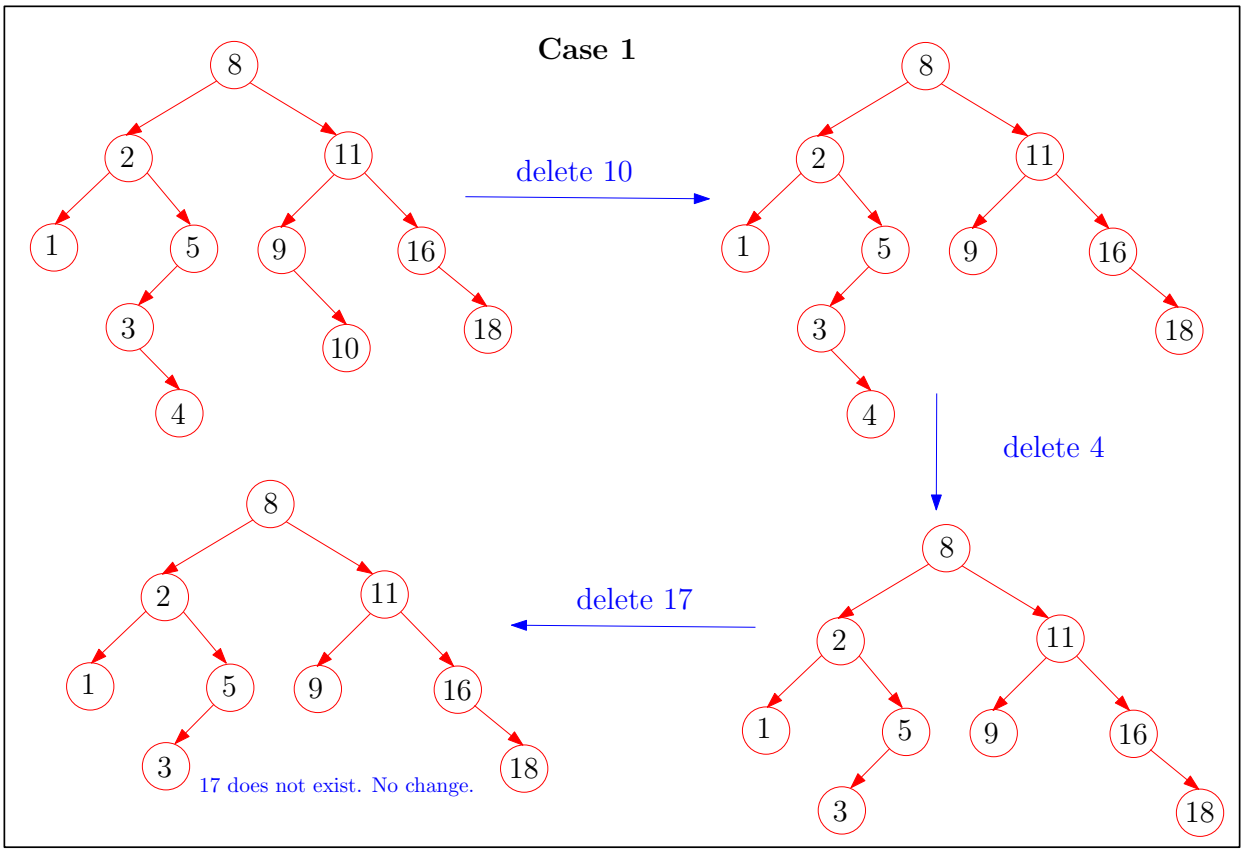
- If *val* is not present in the tree, stop the process. Otherwise, delete the node containing *val* using the following cases.
- **[Case 1 (Leaf):]** Let V be the node that is to be deleted. If V is a leaf node, then delete both V and the child-link from $parent(V)$ to V.
- **[Case 2 (One Child):]** Let V be the node that is to be deleted. If V has only a single child V_{child}, then make V_{child} the left-child of $parent(V)$ if V is the left-child of $parent(V)$, else make V_{child} the right-child of $parent(V)$. Delete V.
- **[Case 3 (Two Child):]** Let V be the node that is to be deleted. If V has both a left child V_{left} and a right child V_{right}, then first find the node $V_{left,max}$ in the subtree of V_{left} which has the maximum value in this subtree.

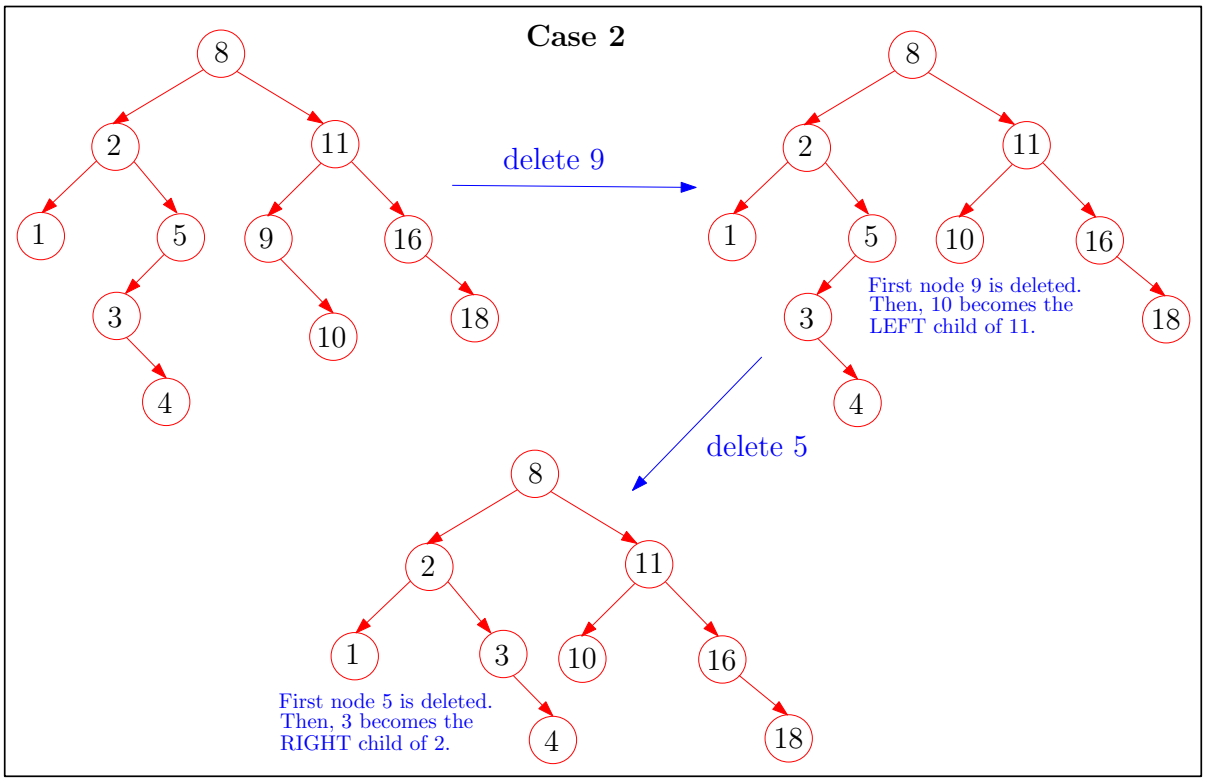
  Observe that $V_{left,max}$ is either a leaf or it only has a left child. This is because $V_{left,max}$ has the highest value in the subtree of V_{left}.

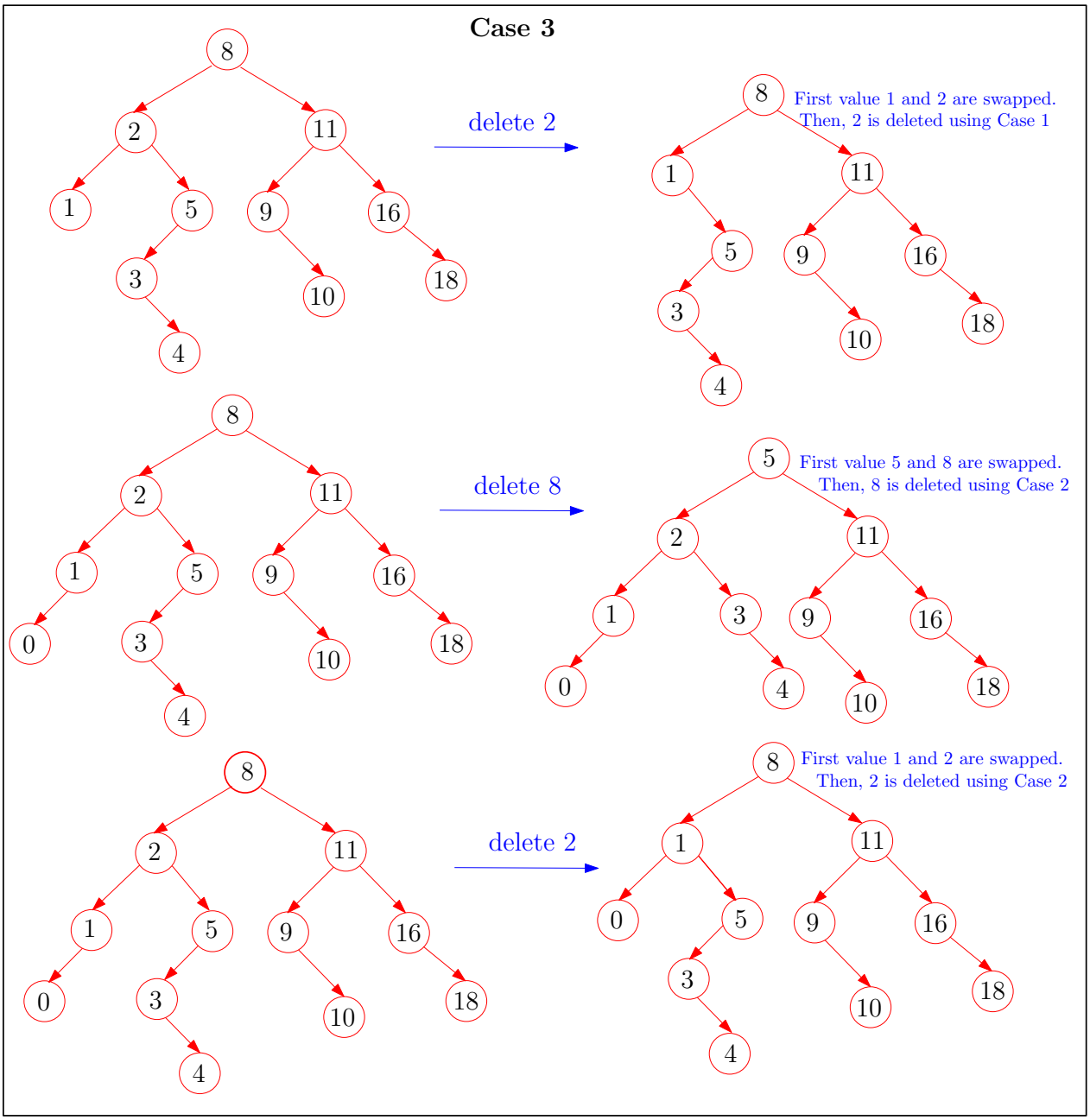
  Swap the values of $V_{left,max}$ and V.

  Delete $V_{left,max}$ using Case 1 or Case 2.

See figures on the following pages for illustration of the above cases.


Case 1
delete 10
delete 4
delete 17
17 does not exist. No change.


Case 2
delete 9
First node 9 is deleted.
Then, 10 becomes the
LEFT child of 11.
delete 5
First node 5 is deleted.
Then, 3 becomes the
RIGHT child of 2.



6 Analysis

It is easy to see that all operations (search, insert, and delete) require $O(H)$ time, where H is the height of the BST. In the worst case, the height of a BST is n, where n is the number of nodes in the tree. This can easily be realized by successively inserting a sorted sequence n distinct integers into the tree. So, quite ironically, BST behavior is worst when a sorted sequence of numbers is inserted into it, because it reduces to a linked list in this case. However, this problem can be alleviated by using *Balanced Binary Search Trees*, such as AVL trees or Red-Black trees, which guarantee a worst-case $O(\log n)$ time for all the operations.