Data Structures - Binary Trees

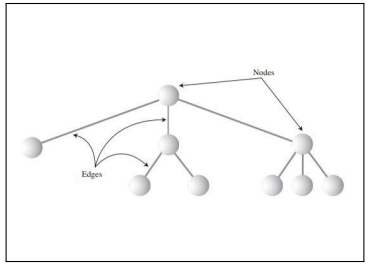
Dr. TGI Fernando [1] [2]

[1]Email: tgi.fernando@gmail.com
[2]URL: http://tgifernando.wordpress.com/

Trees

A tree consists of nodes connected by edges.

- The nodes are represented as circles.
- The edges are represented as lines.



In computer programs,

- Nodes - people, car parts, airline reservations, and so on (objects).
- Edges - relationship between two nodes (represented as a reference in a Java program).

Traversing

- Only way to get from node to node is to follow a path along the lines.

Why Use Binary Trees?

- Fundamental data structure
- Combines the advantages of an ordered array and a linked list.
- You can search an ordered array quickly $[O(\log N)]$.
- You can insert and delete items quickly in a linked list $[O(1)]$.
- It would be nice, if there were a data structure with quick insertion/deletion and quick search.
- Trees provide both of these characteristics.

Trees (Contd.)

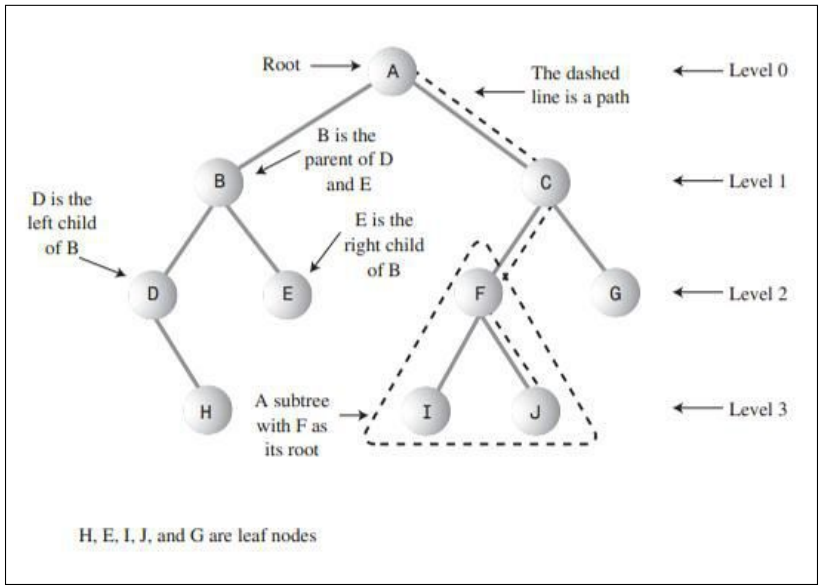
Structure

- There is one node (**root**) in the top row of a tree.
- One or more nodes in the second row connecting to the root node.
- Even more nodes in the third row and they are connected to nodes in the second row, and so on.
- Programs starts an operation at the top (root), and traverse from top to bottom.

Binary Tree
Each node in a binary tree has a maximum of two children.

More generally, a tree can have more than two children, and are called as **multiway trees**.

Tree Terminology

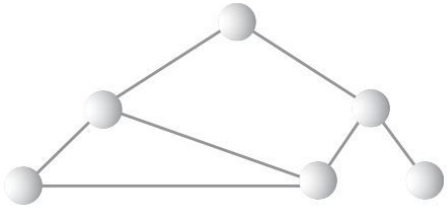


Tree Terminology (Contd.)

Path - Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a path.

Root - The node at the top of the tree.

- There is only one root in a tree.
- For a collection of nodes and edges to be a tree, there must be one and only one path from the root to any other node.



A non-tree

Tree Terminology (Contd.)

Parent - Any node (except the root) has exactly one edge running upward to another node. The node above it is called the parent of the node.

Child - Any node may have one or more lines running downward to other nodes. These nodes below a given node are called its children.

Leaf - A node that has no children is called a leaf node or simply a leaf. There can be only one root in a tree, but there can be many leaves.

Subtree - Any node may be considered to be the root of a subtree, which consists of its children, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.

Tree Terminology (Contd.)

Visiting

- A node is visited when the program control arrives at the node.
- Carry out some operation on the node (**E.g.** display contents).
- Passing the control from one node to another is <u>not</u> considered as visiting the node.

Traversing

- Visit all the nodes in some specified order (**E.g.** visiting all the nodes in ascending order).

Tree Terminology (Contd.)

Level of a node

- How many generations the node is from the root.
- Root - level 0
- Its children - level 1
- Its grandchildren - level 2

⋮

Keys

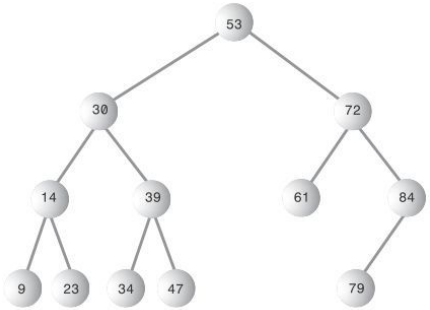
- One data field in an object is designated as a key value.
- This value is used to search for the item or perform other operations on it.

Binary Trees

- A tree that can have at most two children.
- Simplest and most common in Data Structures.
- Two children - left and right (corresponding to their position).
- A node in a binary tree doesn't need to have the maximum of two children.
- It may have only a left child, or only a right child, or it can have no children at all (leaf).
- **E.g.** See slide 5.

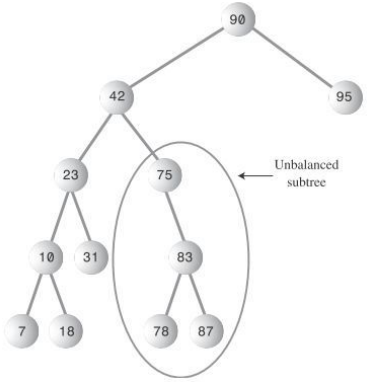
Binary Search Trees

- Its a binary tree.
- For each node,
  - Left child must have a key value less than the node's key value.
  - Right child must have a key value greater than or equal to the node's key value.



Unbalanced Trees

- Most of their nodes on one side of the root or the other.
- Trees become unbalanced because of the order in which the data items are inserted.
- **E.g.** If the sequence of numbers 11, 18, 33, 42 and 65 is inserted into a binary search tree, all the values will be right children and the tree will be unbalanced.



The `Node` Class

The class Node objects contain

1. Data
2. Reference to its left child
3. Reference to its right child
4. A method to display the node's data.

.

```
class Node {
  int data;
  Node left; // Left child
  Node right; // Right child

  public void displayNode () {
    // display node contents
    // ...
  }
}
```

The `Tree` Class

- A class to instantiate the tree itself.
- An object of this class holds all the nodes of the tree.
- It has only one field: a node variable that holds all the nodes.
- It doesn't need fields for other nodes because they are all accessed from the root.
- It has number of methods - finding, inserting, deleting, traversing and displaying the tree.

```
class Tree {
  private Node root; // the only data field in Tree
  public Node find (int key) {...}
  public void insert (int i) {...}
  public void delete (int i) {...}
  // various other methods
} // end of class Tree
```

The `TreeApp` Class

```
class TreeApp {
  public static void main (String[] args) {
    Tree myTree = new Tree; // make a tree
    // insert 3 nodes
    myTree.insert (10);
    myTree.insert (15);
    myTree.insert (5);
    // find node with key 15
    Node found = myTree.find (15);
    if (found != null)
      System.out.println ("Found the node with key 15");
    else
      System.out.println ("Could not find the node with key
      15");
  } // end of main()
} //end of class TreeApp
```

Finding a Node

- Finding a node with a specific key.
- This key could be
  - employee number
  - car part number
  - student index number
  - insurance policy number
  - etc.

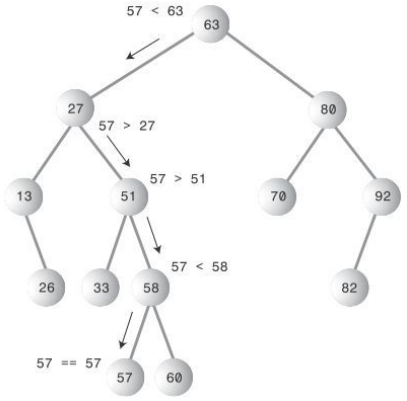
E.g.

```
class Node {
  int empNo; // key
  String name;
  String address;
  int tel;
  float salary;
  Node left; // Left child
  Node right; // Right child

  public void displayNode () {
    // display node contents
    // ...
  }
}
```

Finding Node 57

- The program compares the key value 57 with the value at the root, which is 63.
- The key is less, desired node must be on the left side of the root.
- So the key is compared with the left child of the root, which is 27.
- Since $57 > 27$, the key is in the right subtree of 27.
- Next value 51, which is less than the key, so we go to the right, to 58, and then to the left, to 57.



- This time key and the value at the node are the same, so we found the node we want.

Code: Finding a Node

```
public Node find(int key){
  // find node with given key (assumes non-empty tree)
  Node current = root; // start at root

  while(current.data != key) { // while no match,
    if(key < current.data) // go left?
      current = current.left;
    else
      current = current.right; // or go right?

    if(current == null) // if no child,
      return null; // didn't find it
  }

  return current; // found it
}
```

Stopping Criteria: Finding a Node

Can't find the node

- If `current` becomes equal to `null`, we couldn't find the next node in the sequence.
- Reached to the end of line without finding the node looking for.
- So, the `key` does not exist in the tree.
- Indicate this by returning `null`.

Found the node

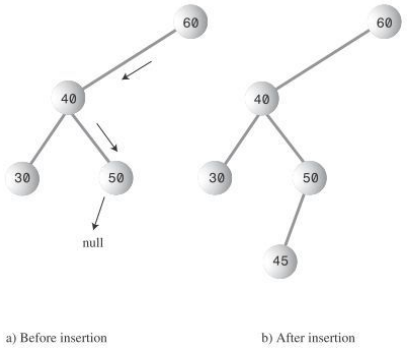
- If the method found the node tha contains the `key` looking for, it returns the node.
- Any routine that called `find()` can access any data of the found node.

Efficiency: Finding a Node

- The time required to find a node depends on how many levels down it is situated.
- If the tree is balanced, the tree has approximately $\log_2 N$ levels.
- Therefore, the program can find any node using a maximum of only five comparisons.
- This is $O(\log N)$ time.

Inserting a Node

- First find the place to insert it.
- Similar to 'Can't find the node' in finding a node.
- Follow the path from the root to the appropriate node, which will be the parent of the new node.
- When the parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less or greater than that of the parent.



a) Before insertion b) After insertion

Inserting a Node (Contd.)

Note:

- We need a new variable, `parent` (the parent of the current node) to remember the last `non-null` node we encountered.
- For this, before visiting the left or right child of the current node, set the current node's reference to `parrent` variable.

Traversing the Tree

- Visiting each node in the specified order.
- Three simple ways to traverse a tree.
  (1) Preorder
  (2) Inorder
  (3) Postorder

Inorder Traversal (LNR)

- Visits nodes in ascending order in a binary tree.
- The simplest way to carry out a traversal is the use of recursion.
- The method is called with a node as an argument.
- Initially, this node is the root.
- The method needs to do only three things:
  (1) Call itself to traverse the node's left subtree.
  (2) Visit the node (**E.g.** Displaying the node contents, writing its contents to a file, etc.).
  (3) Call itself to traverse the node's right subtree.

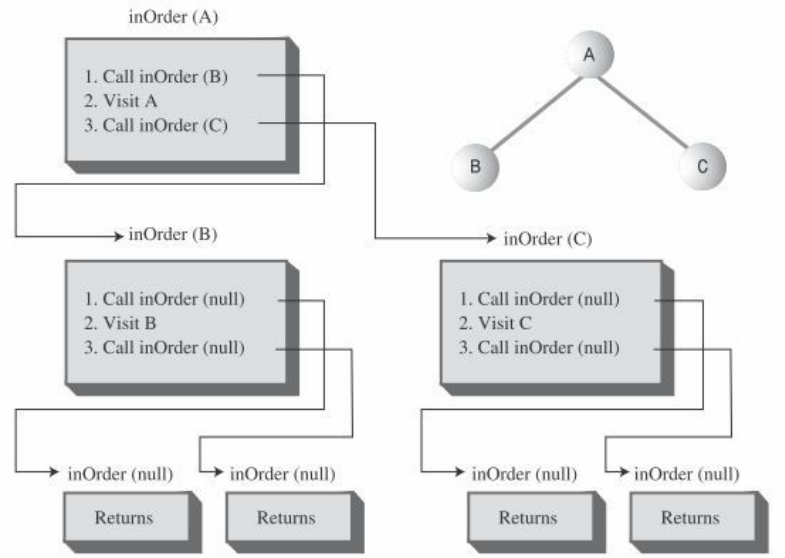
Code: Inorder Traversal

```
private void inOrder (Node localRoot) {
  if (localRoot != null) {
    inOrder(localRoot.left);
    System.out.print(localRoot.data + "␣");
    inOrder(localRoot.right);
  }
  else {
    return; // return to previous recursive version
  }
}
```

Note: This method is initially called with the root as an argument.

```
inOrder (root);
```

Simple Example: Inorder Traversal



Simple Example: Inorder Traversal (Contd.)

- We start by `inOrder()` with the root `A` as an argument (say `inOrder(A)`).
- `inOrder(A)` first calls `inOrder()` with its left child, `B`, as an argument (`inOrder(B)`).
- `inOrder(B)` now calls `inOrder()` with its left child as an argument.
- However, it has no left child, so this argument is `null` (`inOrder(null)`).
- There are now three frames of `inOrder()` in the call stack (`inOrder(A)`, `inOrder(B)` and `inOrder(null)`).
- However, `inOrder(null)` returns immediately to its called method (`inOrder(B)`) because its argument is `null`.

Simple Example: Inorder Traversal (Contd.)

- Now `inOrder(B)` goes on to visit `B` and display it.
- Then `inOrder(B)` calls `inOrder()` again, with its right child (`null`) as an argument (i.e. `inOrder(null)`).
- Again it returns immediately to `inOrder(B)`.
- Now `inOrder(B)` completed its three steps, so the program control goes back to `inOrder(A)`.
- Now `inOrder(A)` visits `A` and display it.
- Next `inOrder(A)` calls `inOrder()` with its right child, `C`, as an argument (`inOrder(C)`).
- Like `inOrder(B)`, `inOrder(C)` has no children, so step 1 returns with no action, step 2 visits `C`, and step 3 returns with no action.
- Next, control goes back to `inOrder(A)`, and `inOrder(A)` has completed its 3 steps, and the entire traversal is complete.

Preorder and Postorder Traversals

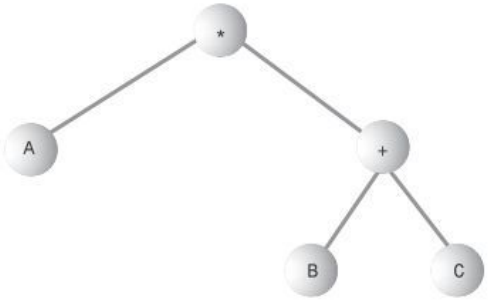
Preorder Traversal (NLR)

1. Visit the node.
2. Call itself to traverse the nodes left subtree.
3. Call itself to traverse the nodes right subtree.

Postorder Traversal (LRN)

1. Call itself to traverse the nodes left subtree.
2. Call itself to traverse the nodes right subtree.
3. Visit the node.

Preorder and Postorder Traversals (Contd.)



Preorder - `*A+BC` (infix notation)
Inorder - `A*B+C`
Postorder - `ABC+*`

Deleting a Node

First, find the node you want to delete, using the same approach we saw in `find()` and `insert()`.

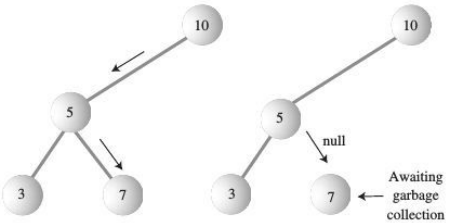
When you have found the node, there are three cases to consider:

1. The node to be deleted is a leaf (has no children).
2. The node to be deleted has one child.
3. The node to be deleted has two children.

Deleting a Node - Leaf Node

To delete a leaf node, you simply change the appropriate child field in the node's parent to point to null, instead of to the node.

The node will still exist, but it will no longer be part of the tree.



Deleting a Node - Leaf Node

Note:

- Java - No need to explicitly delete the node.

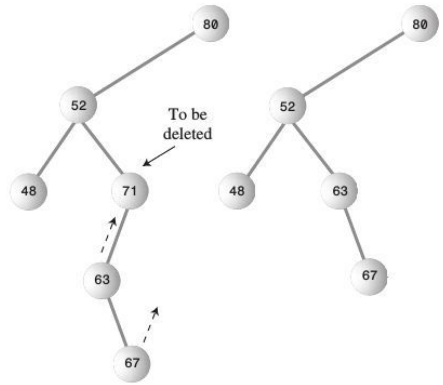
  The garbage collector removes this memory when it realizes that the node is no more used by the program.
- C or C++ - Remove the node from the memory using `free()` or `delete()`.

Deleting a Node - One Child

The node has only two connections: to its parent and to its only child.

To remove the node, connect its parent directly to its child.

This process involves changing the appropriate reference in the parent (`leftChild` or `rightChild`) to point to the deleted nodes child.



a) Before deletion b) After deletion

Deleting a Node - One Child (Contd.)

Two cases:

1. No right child (deleting node could be the root or another node)
2. No left child (deleting node could be the root or another node)

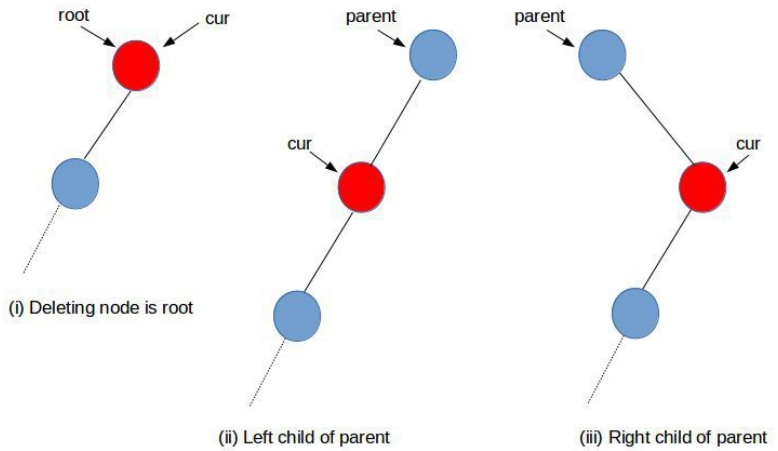
E.g. (Case 1 - No Right Child)(see next slide)

```
if (cur.right == null)
   if (cur == root) // (i)
      root = cur.left;
   else if (isLeftChild == true) // (ii) left child of
   parent
      parent.left = cur.left;
   else // (iii) right child of parent
      parent.right = cur.left;
```

Ex. Write codes for case 2 - no left child.

Deleting a Node - One Child (Contd.)

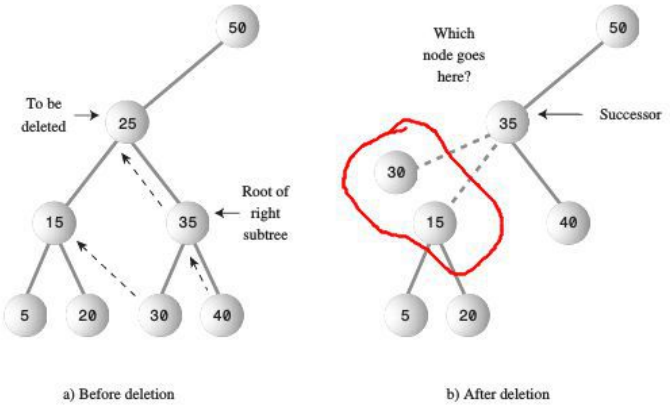
E.g. (Case 1 - No Right Child)



(i) Deleting node is root (ii) Left child of parent (iii) Right child of parent

Deleting a Node - Two Children

If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children. Why not?

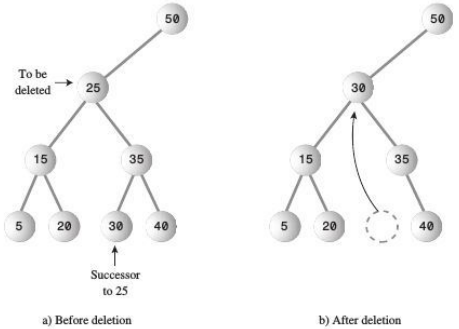


Deleting a Node - Two Children (Contd.)

Inorder Successor - In a binary search tree, nodes are arranged in order of ascending keys. For each node, the node with the next-highest key is called its inorder successor, or simply its successor.

For example, node 30 is the inorder successor (or successor) of node 25.

Solution: To delete node with two children, replace the node with its immediate successor.



Deleting a Node - Two Children (Contd.)

Finding the successor - Two cases

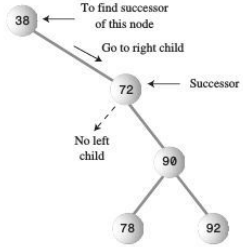


Figure : (1) Deleting node's right child has no left child

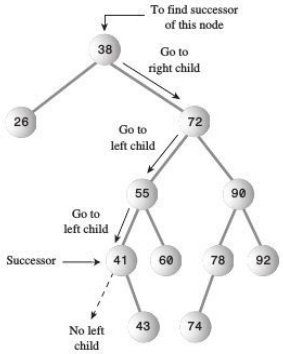


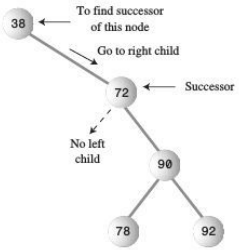
Figure : (2) Deleting node's right child has left child

Deleting a Node - Two Children (Contd.)

Finding the successor

(1) Deleting node's right child has no left child

- If the right child of the original node has no left children, this right child is itself the successor.

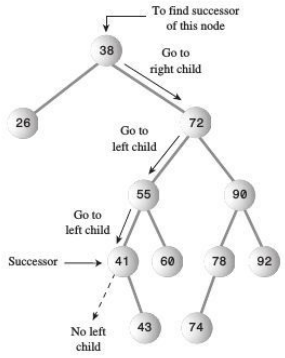


Deleting a Node - Two Children (Contd.)

Finding the successor

(2) Deleting node's right child has left child

- First, the program goes to the original nodes right child.
- Then it goes to this right child's left child, and to this left child's left child, and so on, following down the path of left children.
- The last left child in this path is the successor of the original node.



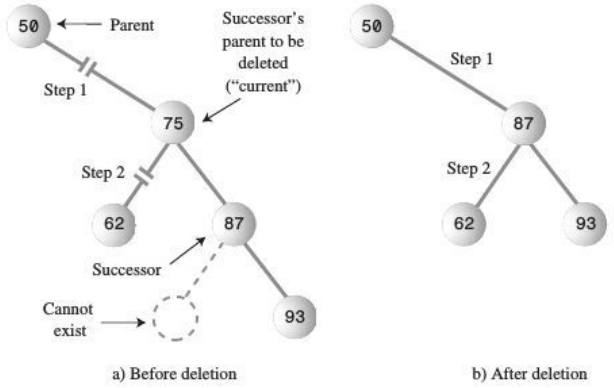
Deleting a Node - Two Children (Contd.)

Finding the successor

Ex. Write Java codes to find the successor of a given node. Note that you may need to access successor's parent in `delete()` method.

Deleting a Node - Two Children (Contd.)

(1) Successor is right child of the deleting node



Deleting a Node - Two Children (Contd.)

(1) Successor is right child of the deleting node

(i) Unplug `current` from the right child field of its `parent` (or left child field if appropriate), and set this field to point to `successor`.

`parent.right = successor;` (or `parent.left = successor;`)

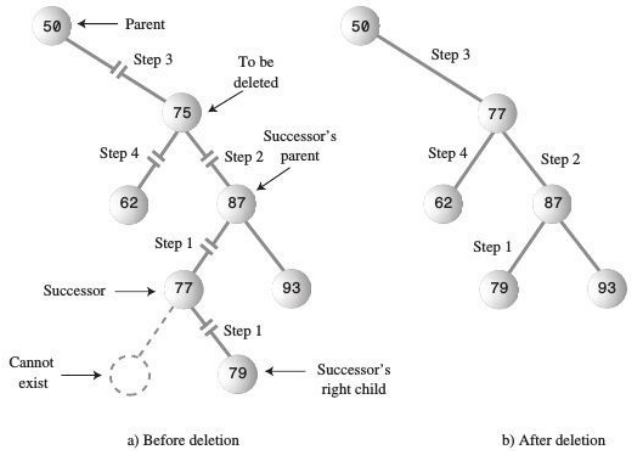
(ii) Unplug `current's` left child from `current`, and plug it into the left child field of `successor`.

```
successor.left = current.left;
```

Note: Mind the case `current = root`.

Deleting a Node - Two Children (Contd.)

(2) Successor is left descendant of right child of deleting node



Deleting a Node - Two Children (Contd.)

(2) Successor is left descendant of right child of deleting node

(i) Plug the right child of `successor` into the left child field of the `successor's` parent.

(ii) Plug the right child of the node to be deleted into the right child field of `successor`.

(iii) Unplug `current` from the right child field of its parent, and set this field to point to `successor`.

(iv) Unplug `current's` left child from `current`, and plug it into the left child field of `successor`.

Deleting a Node - Simple Method

Some programmers for the sake of simplicity, maintain a separate boolean field (`isDeleted`) for each node to indicate the node has been deleted.

When doing other operations, like `find()`, check this field to be sure the node isn't marked as deleted before working with it.

Memory is filled up with the deleted nodes.

Appropriate when there won't be many deletions in a tree.

The Efficiency of Binary Trees

Let's take N = No of nodes and L = No of levels

$$\begin{aligned} N &= 2^L - 1 \\ N+1 &= 2^L \\ L &= log_2(N+1) \end{aligned}$$

Thus, the time needed to carry out the common tree operations is proportional to the base 2 log of N.

In Big O notation we say such operations take $O(logN)$ time.

Number of Nodes	Number of Levels
1	1
3	2
7	3
15	4
31	5
...	...
1,023	10
...	...
32,767	15
...	...
1,048,575	20
...	...
33,554,432	25
...	...
1,073,741,824	30

References

These slides are prepared using the following book:

1. R. Lafore, Data structures & algorithms in Java. Indianapolis, Ind.: Sams, 2003.