CIS 5511. Programming Techniques

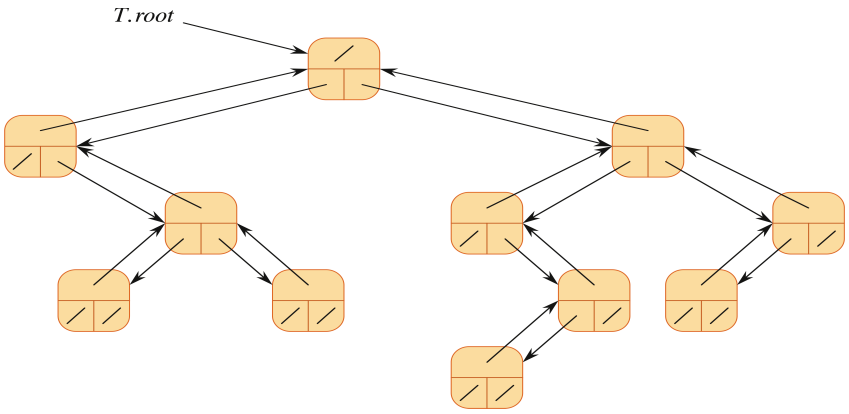
Binary Search Trees

1. Trees and binary trees

A tree is a data structure in which every node, except one called "root", has exactly one predecessor but any number of successors. In each node, its successors are usually distinguished by order.

A binary tree is a tree where each node can only have a left successor and a right successor, or, recursively, as either empty or a root node with a left subtree and a right subtree (both are binary trees).

To implement a binary tree, usually each node has two pointers to its successors, though it may also contains a pointer to its predecessor.



To implement a (general) tree, it is possible to use an array or linked list to point to its successors, or convert the tree into a corresponding binary tree, following the "first child : left child, next sibling : right child" mapping.

A complete binary tree can be efficiently stored in an array (as in heaps), though for binary trees in general, too much memory will be wasted using that approach.

A systematic visit of the nodes of a tree is called "walk" or "traversal". For a binary tree, there are three common walk orders, all defined recursively:

- **inorder:** walk the left subtree, visit the root, walk the right subtree
- **preorder:** visit the root, walk the left subtree, walk the right subtree
- **postorder:** walk the left subtree, walk the right subtree, visit the root

The algorithm for inorder walk:

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

The other two can be obtained by changing the position of the recursive calls. For general trees, only the last two orders are defined.

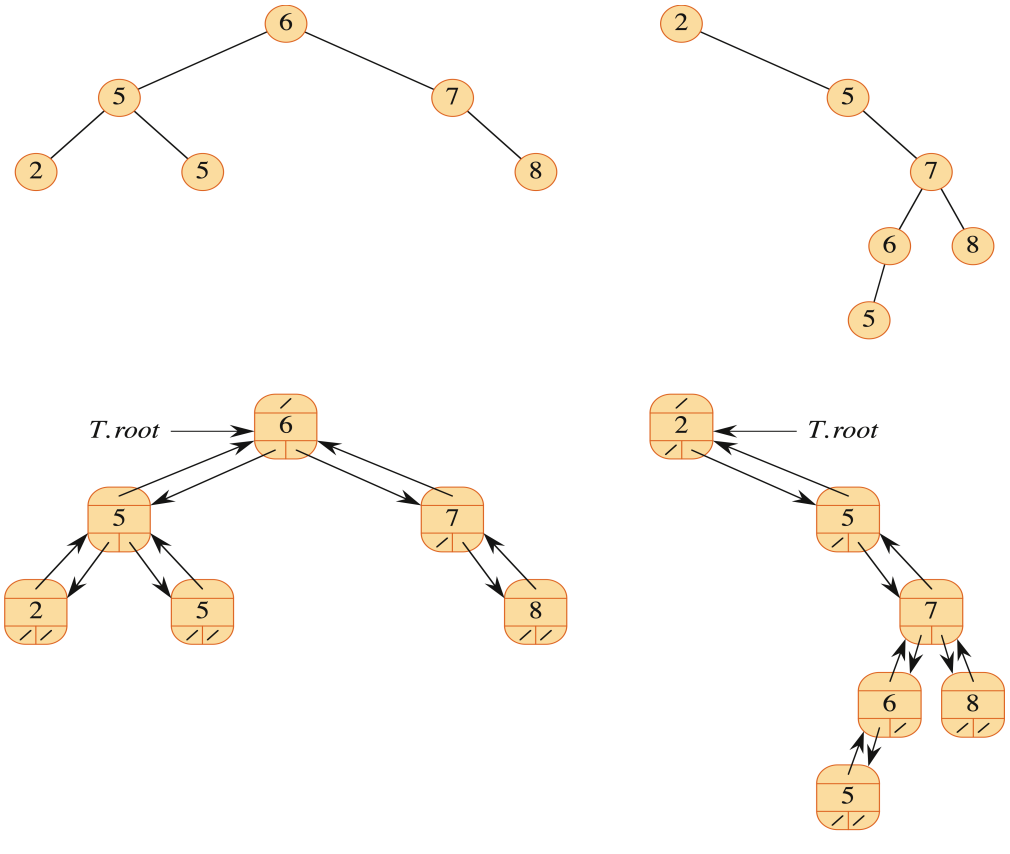
These orders can be followed manually by tracing the outline drawn around the tree:

- Preorder: visit node as you pass it going down,
- Inorder: visit node as you pass under it,
- Postorder: visit node as you pass it going up.

If the nodes in the tree are not sorted in any way, search can be done by tree walk, and takes $\Theta(n)$ time. Even so, "search" is different from "walk" in its input arguments and ending condition.

2. BST and search

Binary Search Tree (BST) is a special type of binary tree where *for every node in the tree, all nodes in its left subtree have smaller keys, and all nodes in its right subtree have larger keys*, though this definition can be extended to allow duplicate "keys". Different BSTs may contain the same set of keys. Examples:



(a) (b)

Given this definition, an inorder walk of a binary search tree lists all of its node in order. In this sense, BST is "sorted horizontally". In comparison, a heap is "sorted vertically", so only maintains order in a path, which is a partial order among all the nodes in the tree.

For such a binary tree, search is similar to binary search in a sorted array. The algorithm can be either recursive or iterative.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2       if k < x.key
3            x = x.left
4       else x = x.right
5  return x
```

The path the algorithm following is from the root to a node where the key to be searched is or should be in the tree, therefore the running time is proportional to the length of the path, and the worst case running time is proportional to the height of the tree.

We can take the search for the minimum and maximum keys as special cases of the search operation. In these cases, the comparisons in the path become unnecessary, and the algorithm simply goes to the end of one direction: left for the minimum and right for the maximum.

Given a node *x* in a binary search tree, its (inorder) successor is the node with the smallest key greater than *x.key*, so in an inorder tree walk this node will immediately follow *x*. The following algorithm requires the pointer to parent in each node. If x has a right subtree, then its successor is the minimum node in it, otherwise its successor is its closest ancestor that x is in its left subtree.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)   // leftmost node in right subtree
3  else // find the lowest ancestor of x whose left child is an ancestor of x
4       y = x.p
5       while y ≠ NIL and x == y.right
6            x = y
7            y = y.p
8       return y
```

The Tree-Predecessor algorithm is symmetric to the above one.

Repeatedly calling Tree-Successor will give us a non-recursive inorder tree walk algorithm.

All the search algorithms on BST run in O(h) time, where h is the height of the tree.

3. Insertion and deletion in BST

Insertions and deletions in a binary search tree by key consist of (1) search for the given key, (2) actual modification. The result should keep the BST properties (shape and order).

The following algorithm inserts node z into BST T (assume z is not already in T):

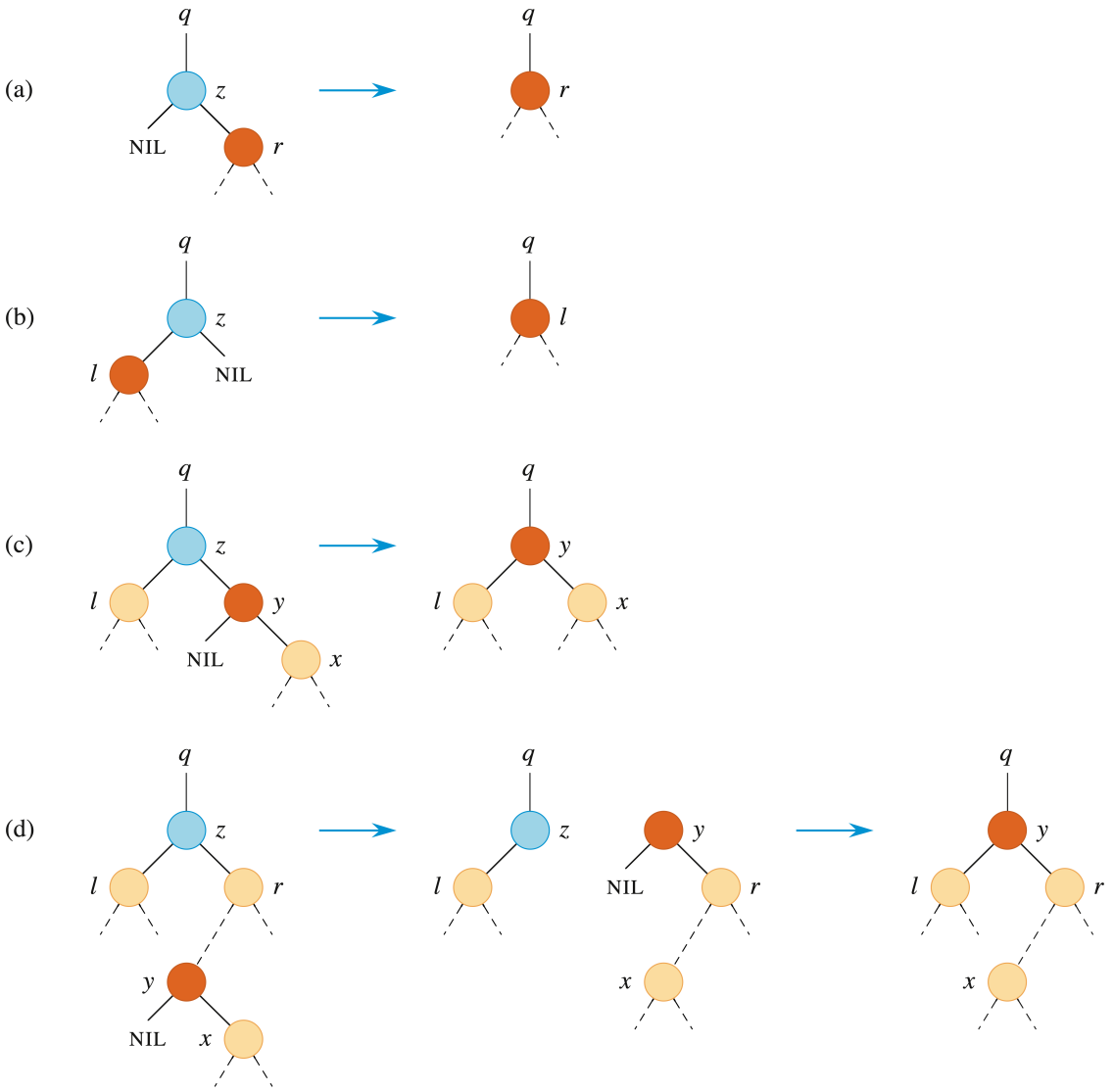
TREE-INSERT(T, z)

```
1   x = T.root              // node being compared with z
2   y = NIL                 // y will be parent of z
3   while x ≠ NIL           // descend until reaching a leaf
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y                 // found the location—insert z with parent y
9   if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

In the algorithm, x traces a path to the insertion point, and y indicates the parent of x.

The deletion algorithm is more complicated, because after a non-leaf node is deleted, the "hole" in the structure needs to be filled by another node. There are three possibilities:

1. To delete a leaf node (no children): disconnect it.
2. To delete a node with one child: bypass the node and directly connect to the child.
3. To delete a node with two children: replace the node by its inorder successor (or predecessor). As that node can only have none or one child, the situation becomes one of the above two.



This solution is realized with the help of an algorithm *TRANSPLANT* that replaces one subtree with root u with another subtree with root v.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

In the following algorithm, z is an input argument referring to the node to be deleted from the BST T, and the local variable y refers to its successor.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)          // replace z by its right child
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)           // replace z by its left child
 5  else y = TREE-MINIMUM(z.right)         // y is z's successor
 6      if y ≠ z.right                     // is y farther down the tree?
 7          TRANSPLANT(T, y, y.right)      // replace y by its right child
 8          y.right = z.right              // z's right child becomes
 9          y.right.p = y                  //       y's right child
10      TRANSPLANT(T, z, y)                // replace z by its successor y
11      y.left = z.left                    // and give z's left child to y,
12      y.left.p = y                       //       which had no left child
```

Both above algorithms have run time O(h).

Since in BST all major operations have run time O(h), the height of a binary search tree determines the worst-case run time. For a binary tree with n nodes, the shortest tree (complete binary tree) has a height $h = \Theta(\lg(n))$, and the highest tree (linear list) has a height $h = \Theta(n)$. A randomly formed BST has an expected height $h = \Theta(\lg(n))$.