Binary Trees (Ch. 12)

Binary trees are an efficient way of storing data so that searches can be O(lgn).

Used when we need a data structure that supports dynamic set operations, e.g., for tree S and key x (searching is primary purpose of BST)

- **INSERT(S, x)**
- **Search(S, k)**
- **MINIMUM(S), MAXIMUM(S)**
- **SUCCESSOR(S, x), PREDESSOR(S, x)**
- **Preorder, Inorder and Postorder traversal help to evaluate expressions**

Binary Search Trees

Requirements for Binary Search Tree (BST):

1. Must be a binary tree.
2. All keys must be unique (key values are like individual ID numbers).
3. Each node in tree is the root of a BST such that:
   - All nodes to left of root will have keys < root.key, and
   - all nodes to right will have keys > root.key.

Main advantage of BST is **rapid search** and low memory use; memory use is dependent only on size of data set.

Time of search = O(depth of BST) = maximum number of nodes on root to leaf path.

Binary Search Trees

Every binary tree node (internal or leaf) contains a key. These keys are unique.

Binary Search Tree Property: For every node x in tree,

- **y.key < x.key for every y in x.left** (left subtree of x)
- **y.key > x.key for every y in x.right** (right subtree of x)

BST has stronger requirements than heaps do.

These dynamic set operations are supported on BST S and key x

- INSERT(S, x), DELETE(S, x)
- SEARCH(S, x), MINIMUM(S), MAXIMUM(S)
- SUCCESSOR(S, x), PREDESSOR(S, x)
- InorderTraversal(S, x) (to list or print sorted set)

Binary Search Trees

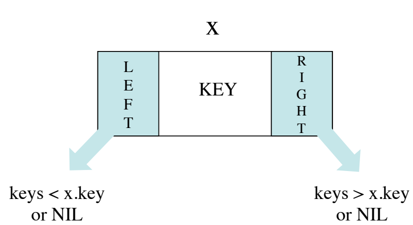
Binary search trees, like heaps, can do most operations quickly because the operations depend on the height (depth) of the tree.

Unlike heaps, binary search trees keep all the data in semi-sorted order such that the cost to print all the data in sorted order is linear in the number of items in the tree. For heapsort, this cost is O(nlgn).

We would like all the dynamic set functions mentioned on the last slide to be O(lgn). However, this good running time depends on the bst implementation.

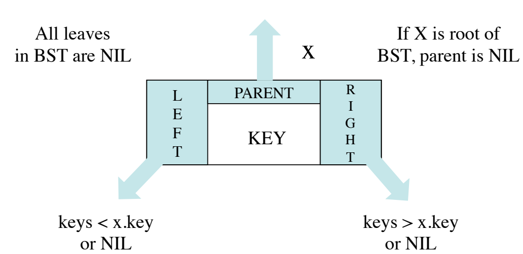
Structure of bst nodes

Each bst node contains fields *left*, *right*, and *key*. Each bst node is the root of a binary search tree. Assume all keys are unique and all leaves are NIL (singly-linked bst)



Structure of bst nodes

A doubly-linked bst node contains fields *left*, *right*, *parent* and *key*. Each bst node is the root of a binary search tree.



Binary Search Trees

The BST's total ordering does the heap's partial ordering one better; not only is there a relationship between a BST node and its children, but there is also a relationship between the children, i.e. the value of a node's left child is always less than the value of its right child.

In fact, every node in the left subtree of a node x has key < x.key and every node in the right subtree of x has key > x.key

BST Insert

```
Insert(T, z)
   y = NIL
   x = T.root
   while x ≠ NIL
       y = x
       if z.key < x.key
           x = x.left
       else
           x = x.right
   z.parent == y
   if y == NIL
       T.root = z
   else if z.key < y.key
       y.left = z
   else
       y.right = z
```

Input is a BST T and a node z such that z.left = z.right = z.parent = NIL

All leaves in T are NIL

Every node is inserted as a leaf

Binary Search Trees

<u>Class exercise</u>: Insert the keys 6, 4, 5, 2, 7, 9 into a BST called A and show the resulting tree. How many links must be traversed to find the node with key=5 after all insertions?

<u>Class exercise</u>: Insert the keys 2, 4, 7, 6, 5, 9 into a BST called B and show the resulting tree. How many links must be traversed to find the node with key=5 after all insertions?

<u>Class exercise</u>: Insert the keys 90, 80, 87, 84, 86 into a BST called C and show the resulting tree after all insertions. What is the height of this BST?

BST Search

```
Iterative-Tree-Search(x, k)
   while (x != NIL) and (k != x.key)
       if k < x.key
           x = x.left
       else
           x = x.right
   return x
```

The iterative version is more efficient, in terms of space used, on most computers.

Both have running times of O(h), where h is the height of the tree.

```
Recursive-Tree-Search(x, k)
   if (x == NIL) or (k == x.key)    \\  base cases
       return x
   if (k < x.key)  \\ recursive case 1: search left
       return Recursive-Tree-Search (x.left, k)
   else            \\ recursive case 2: search right
       return Recursive-Tree-Search (x.right, k)
```

BST Min & Max

The minimum element in a BST can always be found by following left child pointers to a leaf (until a NIL left child pointer is encountered). Likewise, the maximum element can be found by following right child pointers to a leaf.

```
Tree-Minimum(x)
   while x.left ≠ NIL
       x = x.left
   return x
```

```
Tree-Maximum(x)
   while x.right ≠ NIL
       x = x.right
   return x
```

Both have running times of O(h), where h is the height of the tree.

BST Inorder Successor

In a BST, the Inorder Successor of x can also be defined as the node with the smallest key greater than x. This algorithm is used when deleting a node from a BST.

1. If x has a right child, then x.successor is the smallest node in the subtree rooted at x.right.
2. If x has no right child, then x.successor is the nearest ancestor of x whose left child is either an ancestor of x or x itself.

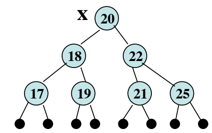
```
Tree-Successor(x)
   if x.right != NIL
       return Tree-Minimum(x.right)
   temp = x.parent
   while temp != NIL and x == temp.right
       x = temp
       temp = temp.parent
   return temp
```

BST Inorder Predecessor

```
Tree-Predecessor(x)
    if x.left != NIL then
        then return Tree-Maximum(x.left)
    temp = x.parent
    while temp != NIL and x = temp.left
        x = temp
        temp = temp.parent
    return temp
```

- If x has a leftchild, then x.predecessor is the largest node in the subtree rooted at x.left.
- If x has no leftchild, then x.predecessor is the nearest ancestor of x whose right child is either an ancestor of x, or x itself

BST Inorder Successor



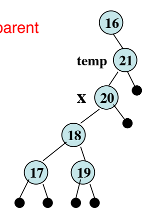
Case where x.right is not equal to NIL

```
Tree-Successor(x)
    if x.right != NIL
        return Tree-Minimum(x.right)
```

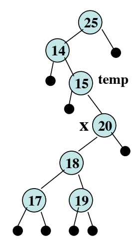
Return value?

```
Tree-Successor(x)
    if x.right != NIL
        return Tree-Minimum(x.right)
    temp = x.parent
    while temp != NIL and x == temp.right
        x = temp
        temp = temp.parent
    return temp
```

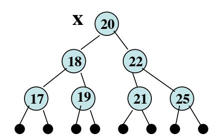
Cases where x.right = NIL



Return value?



Return value?

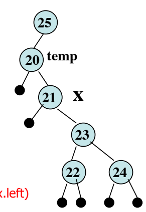


Cases where x.left = NIL

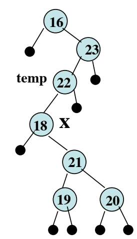
Case where x.left not equal to NIL

Return value?

```
Tree-Predecessor(x)
    if x.left != NIL then
        then return Tree-Maximum(x.left)
    temp = x.parent
    while temp != NIL and x = temp.left
        x = temp
        temp = temp.parent
    return temp
```



Return value?



Return value?

BST Delete

```
Delete(T, z)
    if z.left == NIL or z.right == NIL
        y = z
    else
        y = Tree-Successor(z)
    if y.left != NIL
        x = y.left
    else
        x = y.right
    if x != NIL
        x.parent = y.parent
    if y.parent == NIL
        T.root = x
    else if y == y.parent.left
        y.parent.left = x
    else
        y.parent.right = x
    if y ≠ z
        swap key[z] and key[y]
    return y
```

Input: BST T and node z to be deleted. Three cases:

1) **z has no children. Just remove it.**

2) **z has only one child. Splice out z, by replacing z with z's child.**

BST Delete

```
Delete(T, z)
    if z.left == NIL or z.right == NIL
        y = z
    else
        y = Tree-Successor(z)
    if y.left != NIL
        x = y.left
    else
        x = y.right
    if x != NIL
        x.parent = y.parent
    if y.parent == NIL
        T.root = x
    else if y == y.parent.left
        y.parent.left = x
    else
        y.parent.right = x
    if y ≠ z
        swap key[z] and key[y]
    return y
```

3) **z has two children. Find z's successor y, which has at most one child.**

Since y is z's successor, then y can have no left child, but it may have a right child.

If y is z's right child, then replace z by y, leaving y's right child as is.

If y is in z's right subtree, but is not z's right child, first replace y by its own right child and replace z by y.

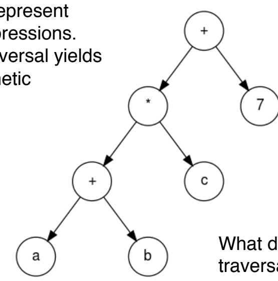
Binary Search Trees

Class exercise: Delete the key 6 in BST A and show the resulting tree.

Class exercise: Delete the key 2 in BST B and show the resulting tree.

Expression Trees

A binary expression tree is a specific kind of a binary tree used to represent arithmetic expressions. An inorder traversal yields an infix arithmetic expression.



What does a preorder traversal yield?

Postorder Traversal

Postorder traversal is a recursive algorithm used to produce a postfix expression from an expression tree (a binary tree). All leaf nodes have NIL left and right children.

```
Postorder-Tree-Walk(x)  /** start at root x **/
1.  if x != NIL
2.      Postorder-Tree-Walk(x.left)
3.      Postorder-Tree-Walk(x.right)
4.      visit the root
```

Running time = $\Theta(n)$ (each node must be visited at least once)

Preorder Traversal

Preorder traversal is a recursive algorithm that is used to get a prefix expression from an expression tree. All leaf nodes have NIL left and right children.

```
Preorder-Tree-Walk(x)  /** start at root x **/
1.  if x != NIL
2.      visit the root
3.      Preorder-Tree-Walk(x.left)
4.      Preorder-Tree-Walk(x.right)
```

Running time = $\Theta(n)$ (each node must be visited at least once)

Inorder Traversal

```
Inorder-Tree-Walk(x)  /** start at root x **/
1.  if x != NIL
2.      Inorder-Tree-Walk(x.left)
3.      visit the root
4.      Inorder-Tree-Walk(x.right)
```

In a BST, inorder-Tree-Walk(root) visits the keys in ascending order, prints out keys in ascending order.

Running time = $\Theta(n)$ (each node must be visited at least once)

Minimizing Running Time

Problem: worst case for binary search tree height is $\Theta(n)$ - no better than a linked list.

Solution: Guarantee tree has small height by making sure it is balanced so that $h = O(\lg n)$.

Method: restructure the tree if necessary. No extra work for searching, but requires extra work when inserting or deleting.

Red-black, AVL and 2-3 trees: special cases of binary trees that avoid the worst-case behavior by ensuring that the tree is nearly balanced at all times.