CIS 330
C/C++ and Unix

Binary Search Trees

Binary Search Tree (BST)

Search Tree

- Data structure that support many dynamic-set operations (i.e., change over time), including
- **Search, minimum, maximum, predecessor, successor, insert, and delete**

Binary Search Tree - Search tree where each node has two children (except for the leaves)

- Items can be looked up, as they are stored in sorted form (by key)

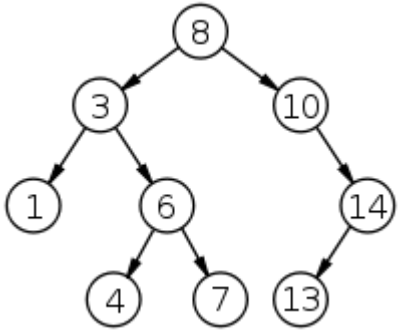
Complexity, worst case

- Log N - if complete
- N - if a linear chain with n nodes

Property of a BST

Binary Search Tree with

- Size 9
- Height 3
- Root at 8



Each node contains

- Key
- Left, right, parent node pointers (nil/null if missing)

Property

- Let x be a node in a BST -
  - if y is a node in the **left** subtree, then **key[y] <= key[x]**
  - if y is a node in the **right** subtree, then **key[x] <= key[y]**

In-order Walk

Traverse the BST (recursively) such that keys are printed **in order**

In-order Walk

Traverse the BST (recursively) such that keys are printed **in order**

```
INORDER-TREE-WALK(x)
if  x != NIL then
   INORDER-TREE-WALK(left[x])
   print key[x]
   INORDER-TREE-WALK(right[x])

INORDER-TREE-WALK(root)
```

Tree Search

Search the tree for a given key

This can be done similarly to tree-walk

```
TREE-SEARCH(x, k)
    if x = NIL or k = key[x] then
        return x
    If k < key[x] then
        return TREE-SEARCH(left[x], k)
    else
        return TREE-SEARCH(right[x], k)
```

Tree Search

Search the tree for a given key

This can be done similarly to tree-walk

Non-recursively

```
TREE-SEARCH(x, k)
    while x != NIL and k != key[x]
        if k < key[x] then
            x <- left[x]
        else
            x <- right[x]
    return x
```

Minimum (key)

BST property - the lowest key will be stored where?

Minimum (key)

BST property - the lowest key will be stored where?

Left-most node

```
TREE-MINIMUM(x)
    while left[x] != NIL
        x <- left[x]
    return x;
```

Maximum (key)

BST property - the largest key will be stored in

Right-most node

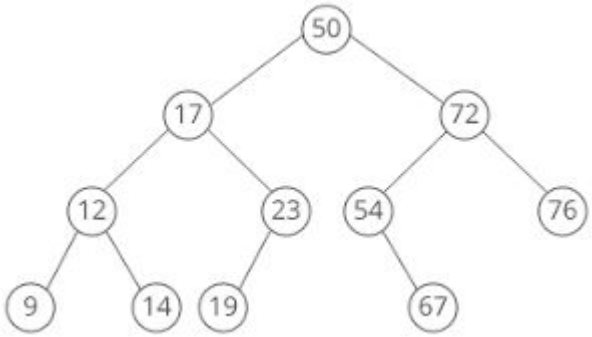
```
TREE-MAXIMUM(x)
    while right[x] != NIL
        x <- right[x]
    return x;
```

Successor and Predecessor

Successor - node with the next larger key

Predecessor - node with the next smaller key

Successor




```
TREE-SUCCESSOR(x)
    if right[x] != NIL then
        return TREE-MINIMUM(right[x])
    y <- p[x]
    while y != NIL and x == right[y]
        x <- y
        y <- p[y]
    return y
```

Predecessor?

Figure out the algorithm first

Hint:

- Think of what the equivalent inverse of the successor algorithm would be (e.g., look at node 54 in the previous image)

Insertion

Insert nodes into the BST so that its property is maintained

```
TREE-INSERT (root,z)
y <- NIL
x <- root
while x != NIL
        y <- x
        if key[z] < key[x] then
                x <- left[x]
        Else
                x <- right[x]
p[z] <- y
if y == NIL then
        root <- z
else if key[z] < key[y] then
        left[y] <- z
Else
        right[y] <- z
```

Insertion

Insert nodes into the BST so that its property is maintained

```
TREE-INSERT (root,z)
y <- NIL
x <- root
while x != NIL
        y <- x
        if key[z] < key[x] then
                x <- left[x]
        else
                x <- right[x]
p[z] <- y
if y == NIL then
        root <- z
else if key[z] < key[y] then
        left[y] <- z
else
        right[y] <- z
```

Traverse the BST until you find the right place to insert the node

Insertion

Insert nodes into the BST so that its property is maintained

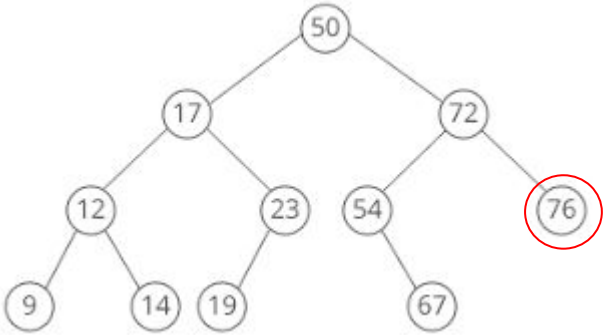
```
TREE-INSERT (root,z)
y <- NIL
x <- root
while x != NIL
        y <- x
        if key[z] < key[x] then
                x <- left[x]
        else
                x <- right[x]
p[z] <- y
if y == NIL then
        root <- z
else if key[z] < key[y] then
        left[y] <- z
else
        right[y] <- z
```

Insert the node

Deletion

Delete node z from the BST so that its property is maintained

Case 1: Node z is a leaf

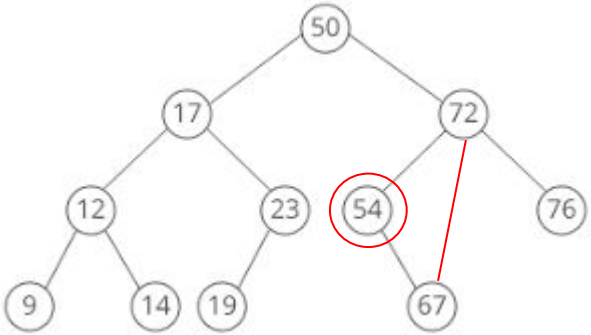


Modify parent to point to NIL and delete z

Deletion

Delete node z from the BST so that its property is maintained

Case 2: Node z has only a single child



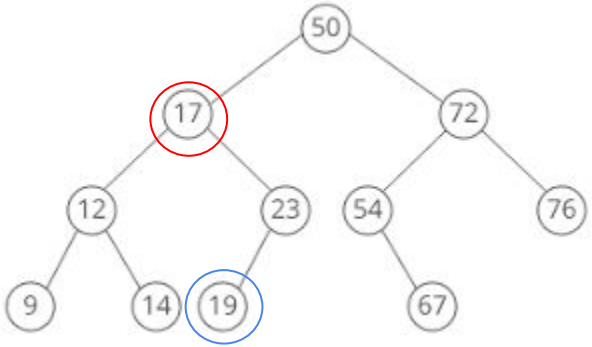
"Splice" out z and make a new "link" between z's parent and child

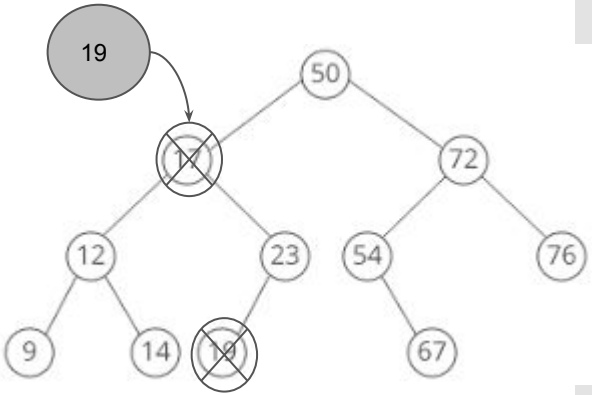
Deletion

Delete node z from the BST so that its property is maintained

Case 3: Node z has two children

- Identify z's successor
- Splice out the successor
- Replace z with the successor





Deletion

```
TREE-DELETE(root, z)
if left[z] == NIL or right[z] == NIL
        then y <- z
        else y <- TREE-SUCCESSOR(z)
if left[y] != NIL
        then x <- left[y]
        else x <- right[y]
if x != NIL then
        P[x] <- p[y]
If p[y] == NIL then
        root <- x
else if y == left[p[y]] then
        left[p[y]] <- x
Else
        right[p[y]] <- x
if y != z then
        key[z] <- key[y];
        Copy y's satellite data into z
return y
```

Deletion

```
TREE-DELETE(root, z)
if left[z] == NIL or right[z] == NIL
        then y <- z
        else y <- TREE-SUCCESSOR(z)
if left[y] != NIL
        then x <- left[y]
        else x <- right[y]
if x != NIL then
        P[x] <- p[y]
If p[y] == NIL then
        root <- x
else if y == left[p[y]] then
        left[p[y]] <- x
Else
        right[p[y]] <- x
if y != z then
        key[z] <- key[y];
        Copy y's satellite data into z
return y
```

Determine node y to splice out
input node z if 0 or 1 child
successor if 2 children

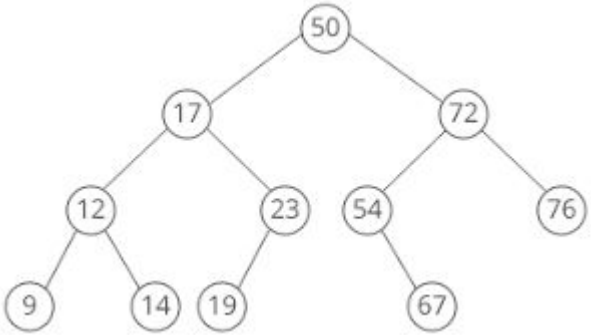
Deletion

```
TREE-DELETE(root, z)
if left[z] == NIL or right[z] == NIL
        then y <- z
        else y <- TREE-SUCCESSOR(z)
if left[y] != NIL
        then x <- left[y]
        else x <- right[y]
if x != NIL then
        P[x] <- p[y]
If p[y] == NIL then
        root <- x
else if y == left[p[y]] then
        left[p[y]] <- x
Else
        right[p[y]] <- x
if y != z then
        key[z] <- key[y];
        Copy y's satellite data into z
return y
```

x is set to child of y or NIL if no child

if y is z -> only 1 child or NIL

if y is succ -> has only 1 child (and no left child)



Deletion

```
TREE-DELETE(root, z)
if left[z] == NIL or right[z] == NIL
        then y <- z
        else y <- TREE-SUCCESSOR(z)
if left[y] != NIL
        then x <- left[y]
        else x <- right[y]
if x != NIL then
        p[x] <- p[y]
If p[y] == NIL then
        root <- x
else if y == left[p[y]] then
        left[p[y]] <- x
else
        right[p[y]] <- x
if y != z then
        key[z] <- key[y];
        Copy y's satellite data into z
return y
```

Splice out z

Deletion

```
TREE-DELETE(root, z)
if left[z] == NIL or right[z] == NIL
        then y <- z
        else y <- TREE-SUCCESSOR(z)
if left[y] != NIL
        then x <- left[y]
        else x <- right[y]
if x != NIL then
        p[x] <- p[y]
If p[y] == NIL then
        root <- x
else if y == left[p[y]] then
        left[p[y]] <- x
else
        right[p[y]] <- x
if y != z then
        key[z] <- key[y];
        Copy y's satellite data into z
return y
```

Replace content of z with content of y

One thing to note (for the homework)

```
class Node {
private:
    int key;
    Node* parent;
    Node* left;
    Node* right;
public:
    // Default constructor
    Node();
    // Constructor
    Node(int in);
    // Destructor
    ~Node();
```

No public interface to change `key`

```
    // Add to parent of current node
    void add_parent(Node* in);
    // Add to left of current node
    void add_left(Node* in);
    // Add to right of current node
    void add_right(Node* in);

    // Get key
    int get_key();
    // Get parent node
    Node* get_parent();
    // Get left node
    Node* get_left();
    // Get right node
    Node* get_right();

    virtual void print_info();
};
```

One thing to note (for the homework)

You can’t simply copy the content of one node to another

You must splice out the existing node and replace it with the new node