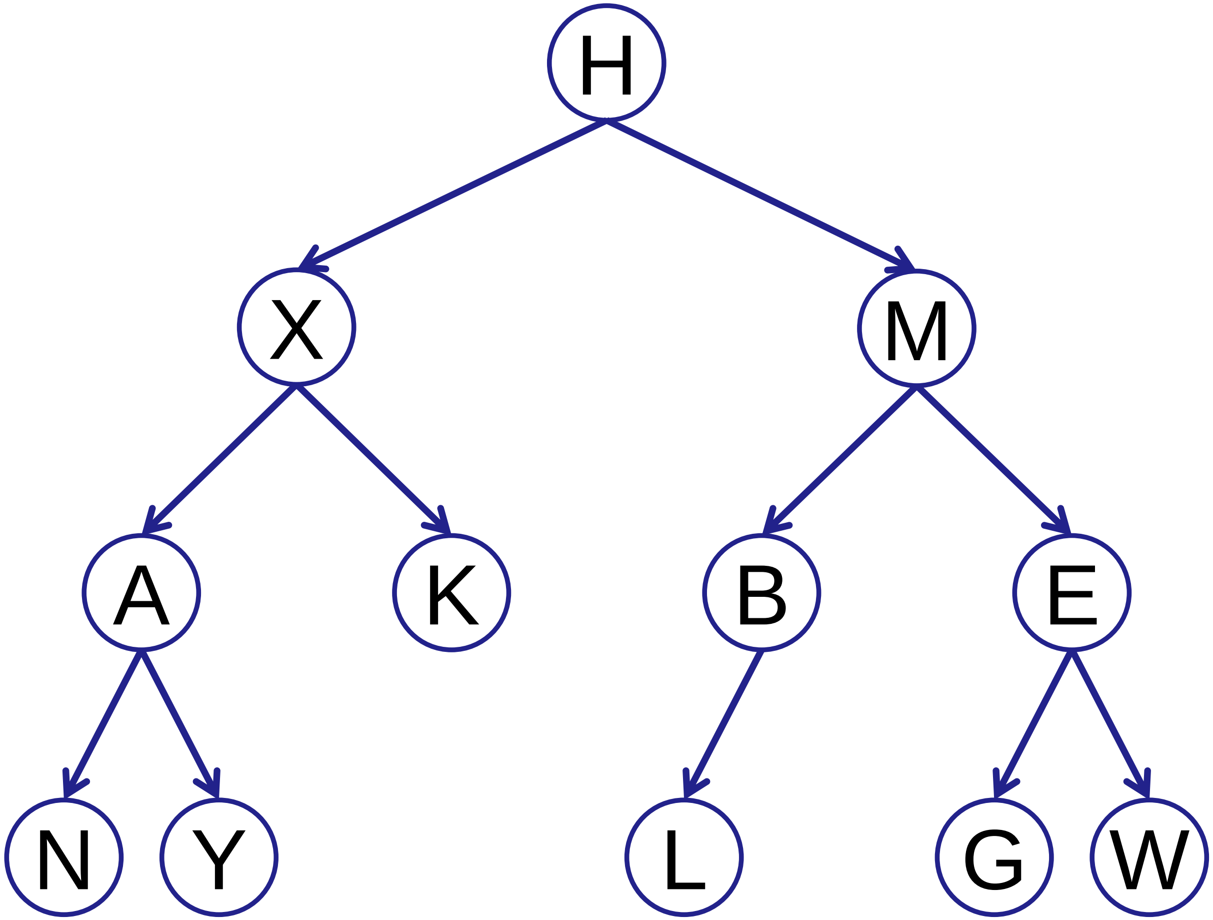
CMSC 341
Lecture 10 Binary Search Trees

John Park

Based on slides from previous iterations of this course

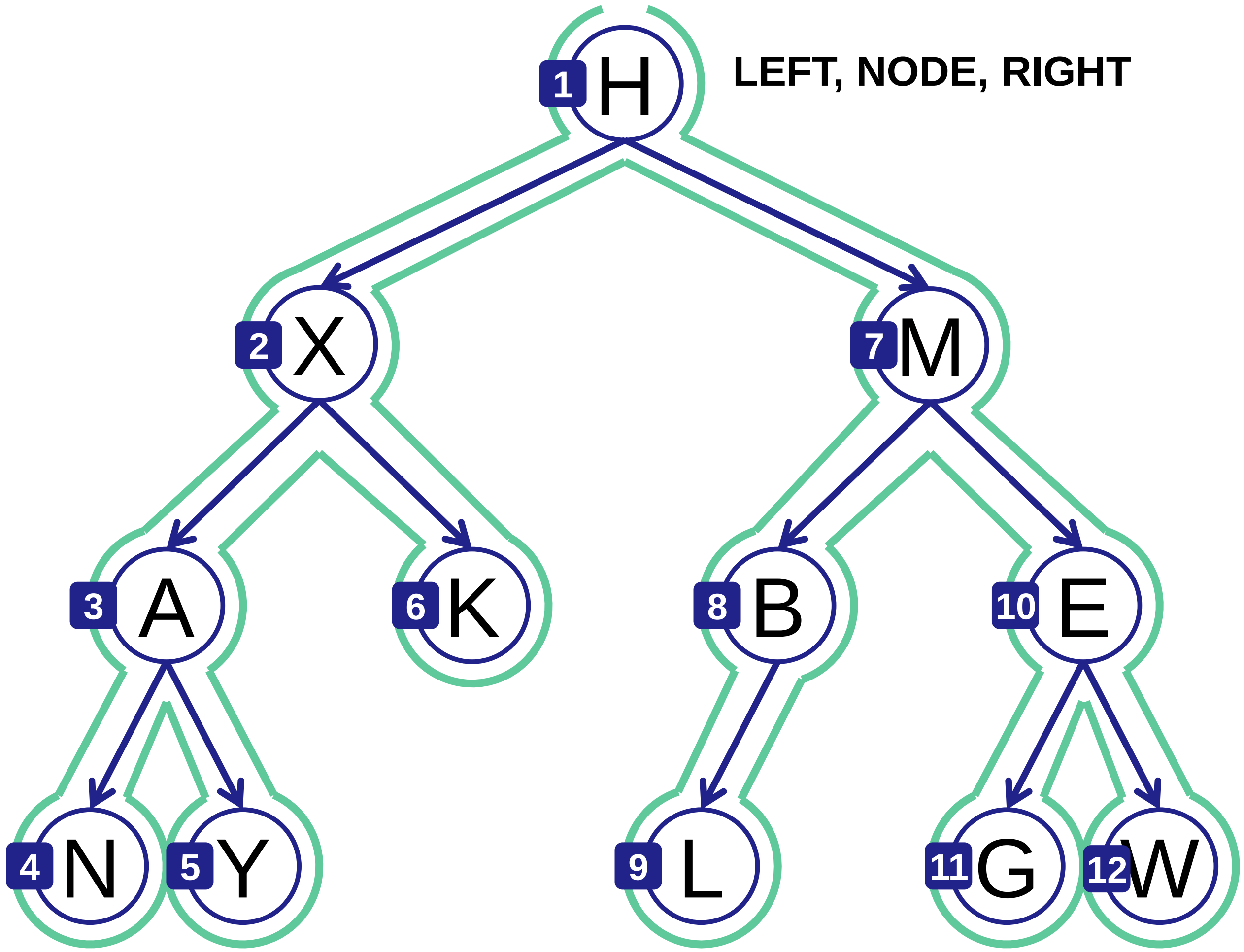
Review: Tree Traversals

Traversal – Preorder, Inorder, Postorder



Preorder Traversal

Display the current node's value
Traverse the left subtree (may be NULL)
Traverse the right subtree (may be NULL)

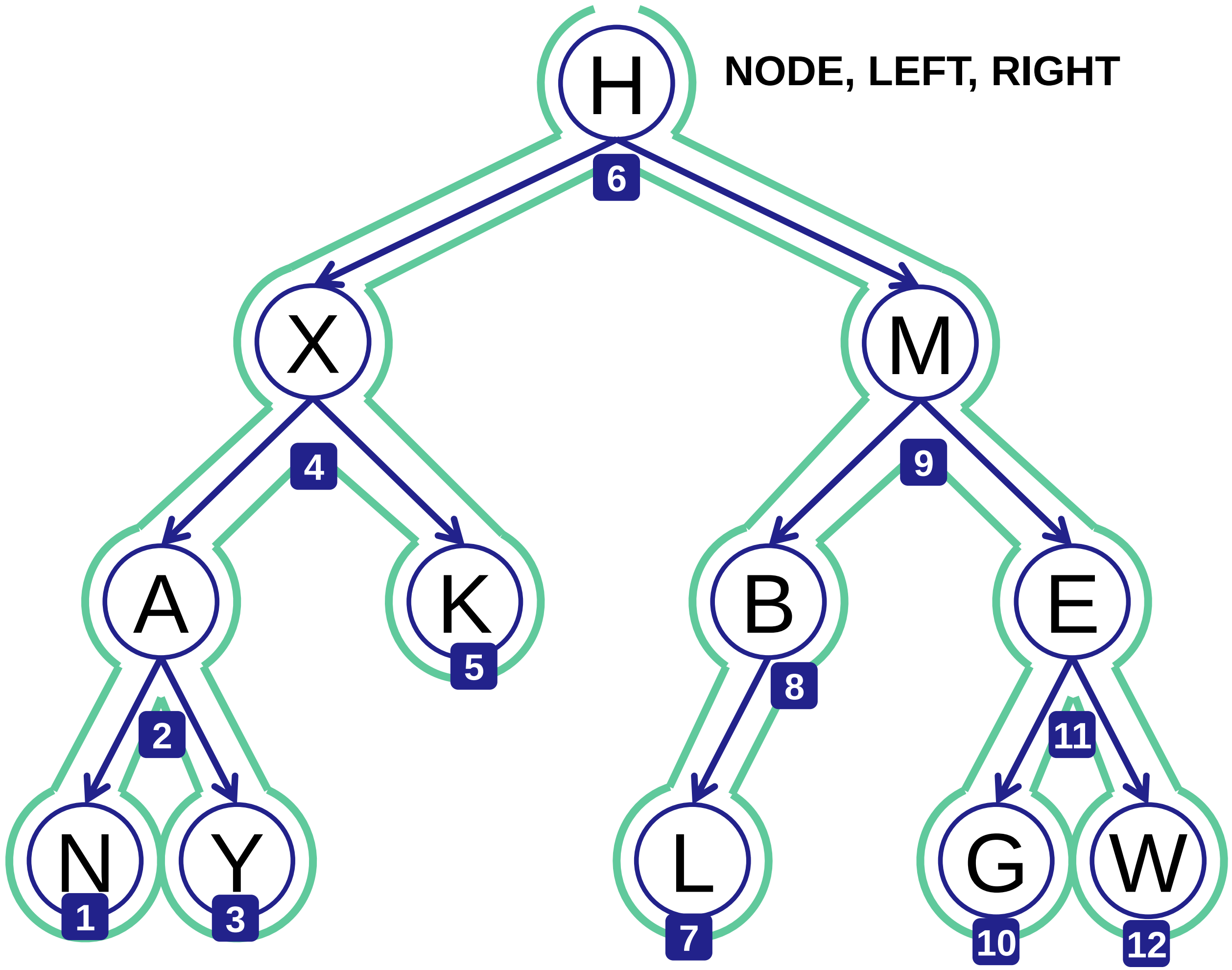


LEFT, NODE, RIGHT

Inorder Traversal

Traverse the left subtree (may be NULL)
Display the current node's value
Traverse the right subtree (may be NULL)

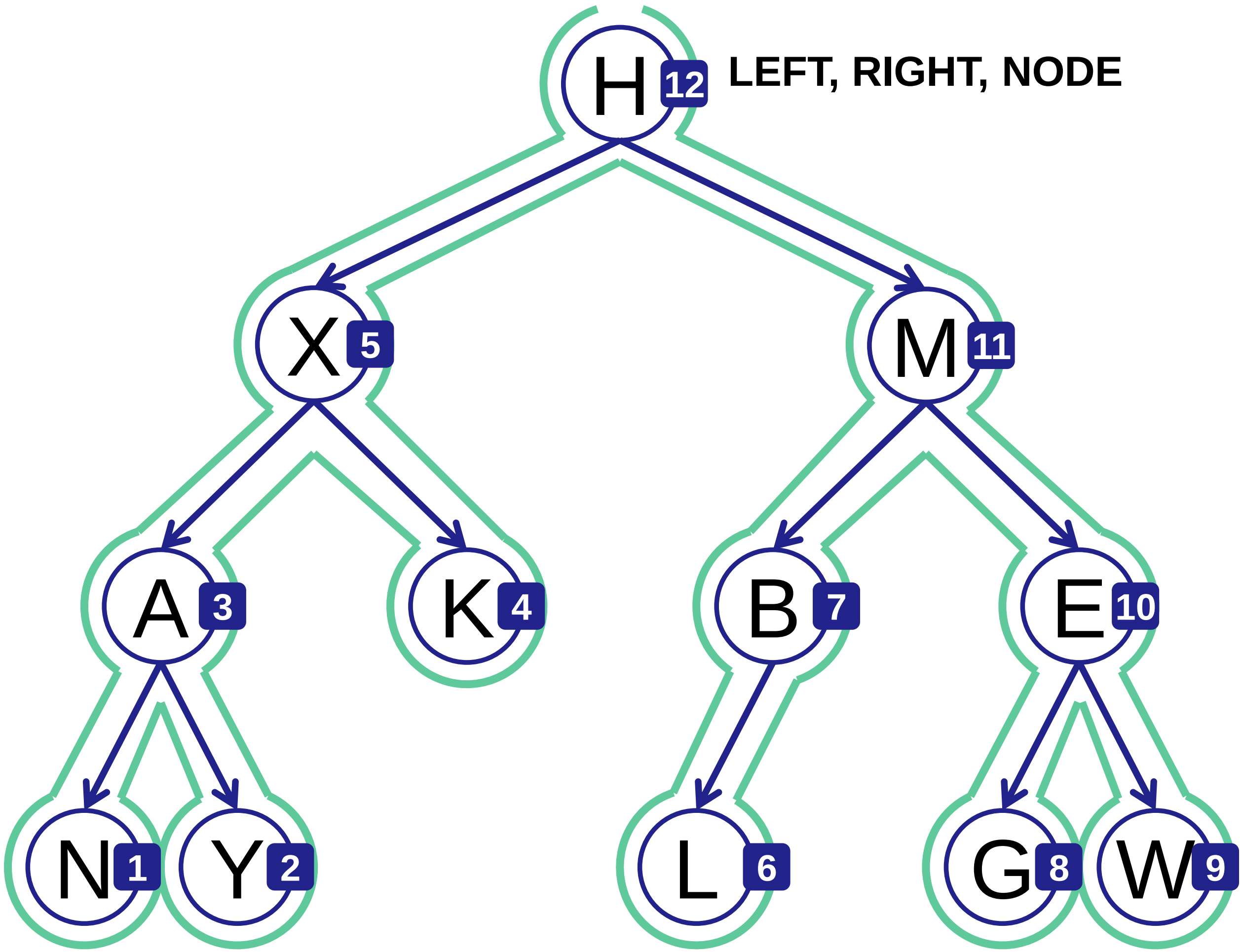
NODE, LEFT, RIGHT



Postorder Traversal

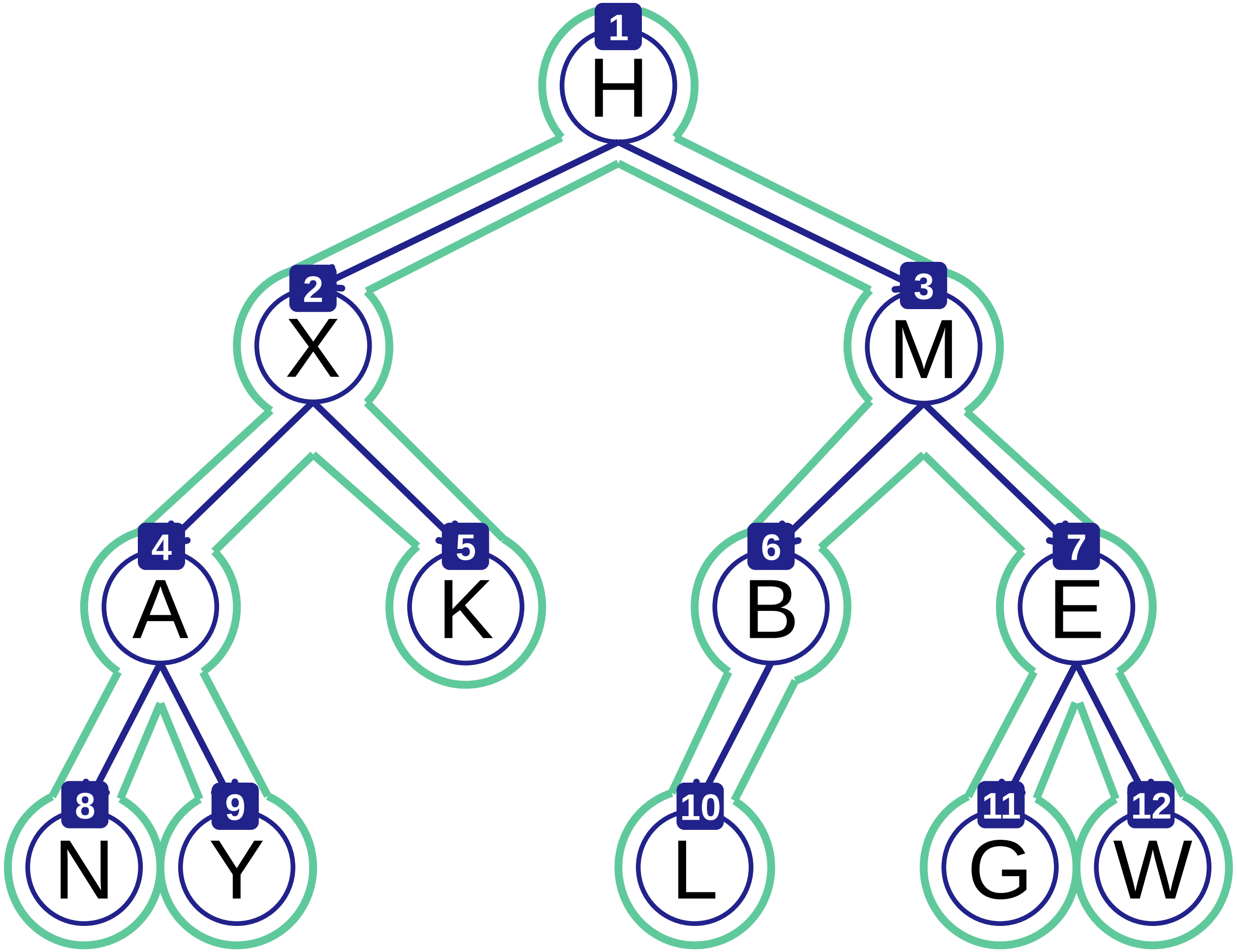
Traverse the left subtree (may be NULL)
Traverse the right subtree (may be NULL)
Display the current node's value

LEFT, RIGHT, NODE



Level Order Traversal

Requires the use of a Queue



Pointers vs References

Passing by Value

- The "default" way to pass variables to functions

```
// function prototype
void PrintVal (int x);

int  x     = 5;
int *xPtr = &x;
PrintVal(x);      // function call
PrintVal(*xPtr); // also valid call
```

Passing a Pointer (Reference by Value)

- Uses pointers (address to the variable)
  - Uses * to dereference, and & to get address

```
void ChangeVal(int *x); //prototype

int  x    = 5;
int *xPtr = &x;
ChangeVal(&x);   // function call
ChangeVal(xPtr); // also valid call
```

Passing a Reference

- Uses references (different from pointers)
  - Allows called function to modify caller's variable

```
void ChangeVal(int &x); //prototype

int  x     = 5;
int *xPtr = &x;
ChangeVal(x);      //function call
ChangeVal(*xPtr); //also valid call
```

Passing a Reference

- Uses references (different from pointers)
  - Allows called function to modify caller’s variable

```
void ChangeVal(int &x); //prototype

int  x     = 5;
int &xRef = x;     //create reference
ChangeVal(x);      //function call
ChangeVal(xRef);   //also valid call
```

Pointers vs. References

- How are references different from pointers?

- References **must** be initialized at declaration
- References **cannot** be changed
- References can be treated as another "name" for a variable
  - No dereferencing to get the value
  - Functions that take values and references have identical definitions

Advantages of Passing by Pointer/Ref

- Advantages:
  - Allows a function to change the value
  - Doesn’t make a copy of the argument (fast!)
  - We can return multiple values
- Disadvantages:
  - Dereferencing a pointer is slower than direct access to the value. (References are internally implemented via pointers)

From: http://www.learncpp.com/cpp-tutorial/74-passing-arguments-by-address

Advantages of References vs. Pointers

- Reference advantages:
  - Can pass as `const` to avoid unintentional changes
  - Values don't have to be checked to see if they're NULL
- Disadvantages:
  - Hard to tell if the function is passing by value or reference without looking at the function itself

From: http://www.learncpp.com/cpp-tutorial/74-passing-arguments-by-address

Properties of Binary Search Trees

Advantages of a BST

- Binary Search Trees are sorted as they're made

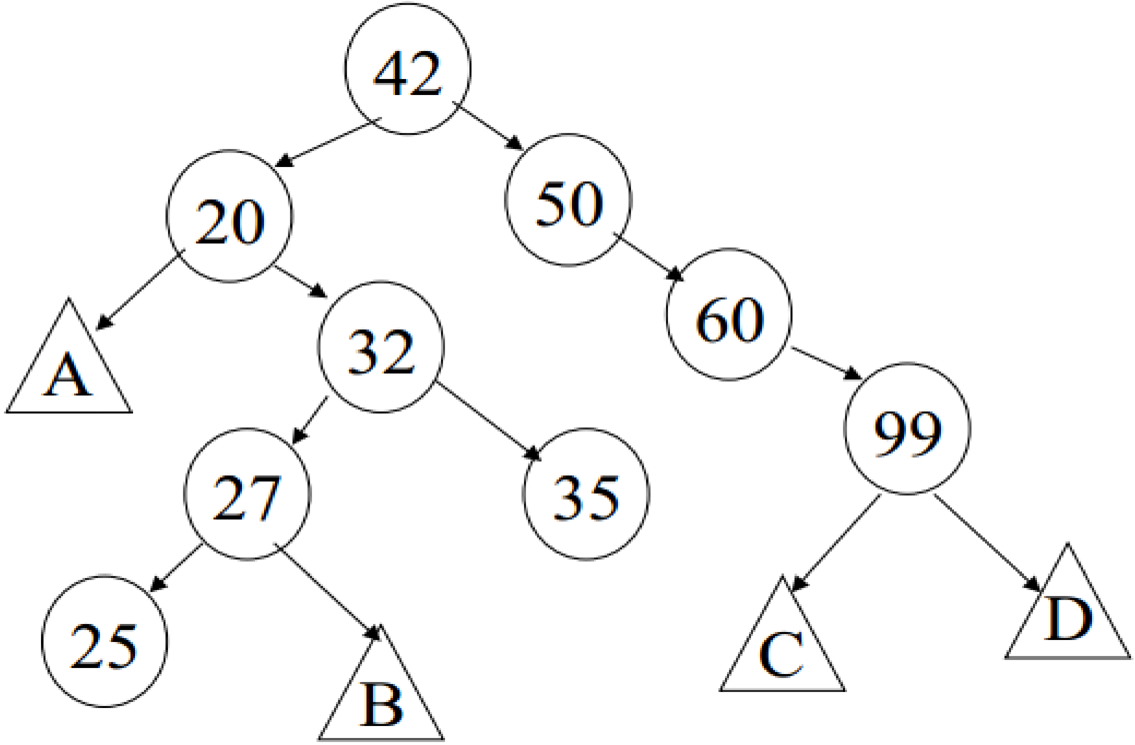
- How quickly does linear binary search find a value?
  - O(log n)
- Binary Search Trees work on the same principle
  - What if the tree isn't "perfect"?
    - Performance will be better/worse: worst-case O(n)
    - But on average, will be O(log n)

Searching Through a BST

- Easy to locate an element of the tree
  - Find arbitrary element:
    - Compare to the current node's value
    - If current node is bigger, go <u>left</u>; otherwise, go <u>right</u>
  - Minimum:
    - Go <u>left</u> until it's no longer possible
    - (It may not be a leaf – it may have a right subtree)
  - Maximum:
    - Go <u>right</u> until it's no longer possible
    - (It may not be a leaf – it may have a left subtree)

Practice: BST of Integers

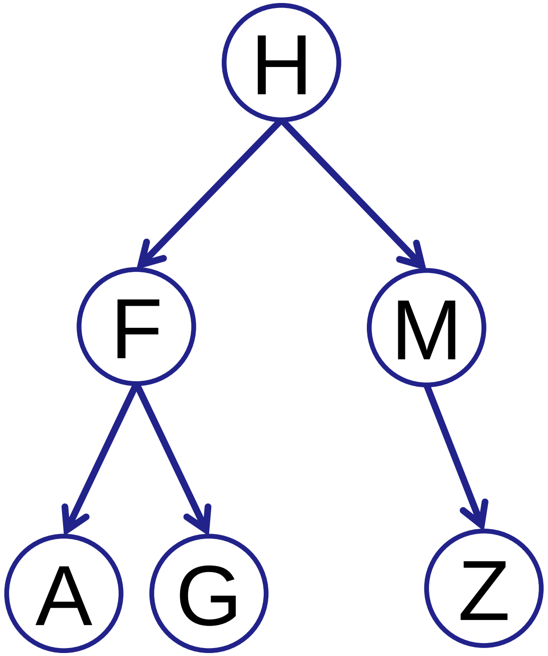
- Describe the values that might appear in the subtrees A, B, C, and D



Example: Creating a BST

- Draw the BST that would result from these values, given in this exact order

- H,F,A,M,G,Z



Practice: Creating a BST

- Draw the BST that would result from these values, given <u>in this exact order</u>

- 8,2,1,9,6,5,3,7,4
- 5,9,1,8,2,6,7,3,4
- 8,1,2,6,9,3,4,7,5
- 1,2,3,4,5,6,7,8,9
- 5,3,7,9,6,1,4

Great website where you can practice and learn about BSTs: http://visualgo.net/bst.html

Subtrees and Recursion

- Every node is the root for its own subtree
  - (Subtree of the actual root is the whole tree)
- Almost everything we do with trees can be (and should be) coded using <u>recursion</u>

- For example: traversal of the tree (pre-, in-, and postorder) can be done recursively
  - Which will print out a BST from low to high?

Implementing a Binary Search Tree

Representing a Binary Search Tree

- What data structure would you use for a BST?
  - Array? Stack? Queue? ???

- (Modified) implementation of Linked List
  - Linked List nodes contain two things:
    - Data, and a pointer to the next node
  - BST nodes should contain…
    - Data, and two pointers: left and right children

Generic Structure for BST node

```
struct BinaryNode
{
    // Member variables
    <AnyType>  element; // Data in the node
    BinaryNode *left;   // Left child
    BinaryNode *right;  // Right child

    // Constructor
    BinaryNode(const <AnyType> & theElement,
               BinaryNode *lt, BinaryNode *rt )
    {
        element  = theElement;
        left = lt;
        right = rt;
    }
}
```

BST Node Functions

- What other functions might we want for a node?

- Constructor that just takes in data (no children)
  - Initializes children to NULL automatically
- **`print()`** function
  - May be mostly handled if the data is really simple or another class with a **`print()`** function
- Destructor (again, may already be handled)
- Getters and setters (mutators/accessors)

Generic Class for BST

```
class BinarySearchTree
{
    public:
        BinarySearchTree( ) :root( NULL )
        { }
        BinarySearchTree( const BinarySearchTree
                          &rhs ) : root( NULL )
        {
            *this = rhs;
        }

    private:
        // this private BinaryNode is within BST
        BinaryNode *root;
}
```

Binary Search Tree Operations

Basic BST Operations

- (BST Setup) → set up a BST
- (Node Setup) → set up a BST Node
- `void insert(x)` → insert x into the BST
- `void remove(x)` → remove x from the BST
- `<type> findMin()` → find min value in the BST
- `<type> findMax()` → find max value in the BST
- `boolean contains(x)` → is x in the BST?
- `boolean isEmpty()` → is the BST empty?
- `void makeEmtpy()` → make the BST empty
- `void PrintTree()` → print the BST

Public and Private Functions

- Many of the operations we want to use will have two (overloaded) versions
- Public function takes in zero or one arguments
  - Calls the private function
- Private function takes in one or two arguments
  - Additional argument is the “root” of the subtree
  - Private function recursively calls itself
    - Changes the “root” each time to go further down the tree

Insert

```
void insert( x )
```

Inserting a Node

- Insertion will always create a new leaf node
- In determining what to do, there are 4 choices
  - Insert the node at the current spot
    - The current "node" is NULL (we've reached a leaf)
  - Go down the left subtree (visit the left child)
    - Value we want to insert is smaller than current
  - Go down the right subtree (visit the right child)
    - Value we want to insert is greater than current
  - Do nothing (if we've found a duplicate)

Insert Functions

- Two versions of insert
  - Public version (one argument)
  - Private version (two arguments, recursive)

- Public version immediately calls private one

```
void insert( const Comparable & x )
{
    // calls the overloaded private insert()
    insert( x, root );
}
```

Starting at the Root of a (Sub)tree

- First check if the "root" of the tree is NULL
  - If it is, create and insert the new node
  - Send left and right children to NULL

```
// overloaded function that allows recursive calls
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) // no node here (make a leaf)
        t = new BinaryNode( x, NULL, NULL );
    // rest of function…
}
```

Insert New Node (Left or Right)

- If the “root” we have is not NULL
  - Traverse down another level via its children
  - Call **`insert()`** with new sub-root (recursive)

```
// value in CURRENT root 't' < new value
else if( x < t->element ) {
    insert( x, t->left ); }

// value in CURRENT root 't' > new value
else if( t->element < x ) {
    insert( x, t->right ); }

else;  // Duplicate; do nothing
```

Full Insert() Function

- Remember, this function is recursive!

```
// overloaded function that allows recursive calls
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) // no node here (make a new leaf)
        t = new BinaryNode( x, NULL, NULL );

   // value in CURRENT root 't' < new value
    else if( x < t->element ) { insert( x, t->left ); }

    // value in CURRENT root 't' > new value
    else if( t->element < x ) { insert( x, t->right ); }

    else;  // Duplicate; do nothing
}
```

What's Up With `BinaryNode * & t`?

- The code "`* & t`" is a <u>reference</u> to a <u>pointer</u>
- Remember that passing a reference allows us to change the <u>value</u> of a variable in a function
  - And have that change "stick" outside the function
- When we pass a variable, we pass its value
  - It just so happens that a pointer's "value" is the address of something else in memory

Find Minimum

```
Comparable findMin( )
```

Finding the Minimum

- What do we do?
  - Go all the way down to the left

```
Comparable findMin(BinaryNode *t )
{
    // empty tree
    if (t == NULL) { return NULL; }

    // no further nodes to the left
     if (t->left == NULL) {
         return t->value;   }
    else {
         return findMin(t->left);   }
}
```

Find Maximum

```
Comparable findMax( )
```

Finding the Minimum

- What do we do?
  - Go all the way down to the right

```
Comparable findMax(BinaryNode *t )
{
    // empty tree
    if (t == NULL) { return NULL; }

    // no further nodes to the right
     if (t->right == NULL) {
         return t->value;    }
    else {
         return findMax(t->right);    }
}
```

Recursive Finding of Min/Max

- Just like **`insert()`** and other functions, **`findMin()`** and **`findMax()`** have 2 versions
- Public (no arguments):
  - **`Comparable findMin( );`**
  - **`Comparable findMax( );`**
- Private (one argument):
  - **`Comparable findMax (BinaryNode *t);`**
  - **`Comparable findMax (BinaryNode *t);`**

Delete the Entire Tree

```
void makeEmpty ( )
```

Memory Management

- Remember, we don’t want to lose any memory by freeing things out of order!
  - Nodes to be carefully deleted

- BST nodes are only deleted when
  - A single node is removed
  - We are finished with the entire tree
    - Call the destructor

Destructor

- The destructor for the tree simply calls the **`makeEmpty()`** function

```
// destructor for the tree
~BinarySearchTree( )
{
    // we call a separate function
    // so that we can use recursion
    makeEmpty( root );
}
```

Make Empty

- A recursive call will make sure we hang onto each node until its children are deleted

```
void makeEmpty( BinaryNode * & t )
{
    if( t != NULL )
    {
        // delete both children, then t
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
        // set t to NULL after deletion
        t = NULL;
    }
}
```

Find a Specific Value

```
boolean contains( x )
```

Finding a Node

- Only want to know if it's in the tree, not where
  - Use recursion to traverse the tree

```
bool contains( const Comparable & x ) const  {
    return contains( x, root ); }

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL ) { return false; }
    // our value is lower than the current node's
    else if( x < t->element ) { return contains( x, t->left ); }
    // our value is higher than the current node's
    else if( t->element < x ) { return contains( x, t->right );}
    else { return true; }   // Match
 }
```

Finding a Node

- Only want to know <u>if</u> it's in the tree, not <u>where</u>
  - Use recursion to traverse the tree

```
bool contains( const Comparable & x ) const  {
    return contains( x, root ); }

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL ) { return
    // our value is lower th
    else if( x < t->element                        ->left ); }
    // our value is higher than the current node's
    else if( t->element < x ) {
    else { return true; }    //
 }
```

We have to have a defined overloaded comparison operator for this to work!

(Both of the **`else if`** statements use **<** so we only need to write one)

Removing a Node

```
void remove( x )
```

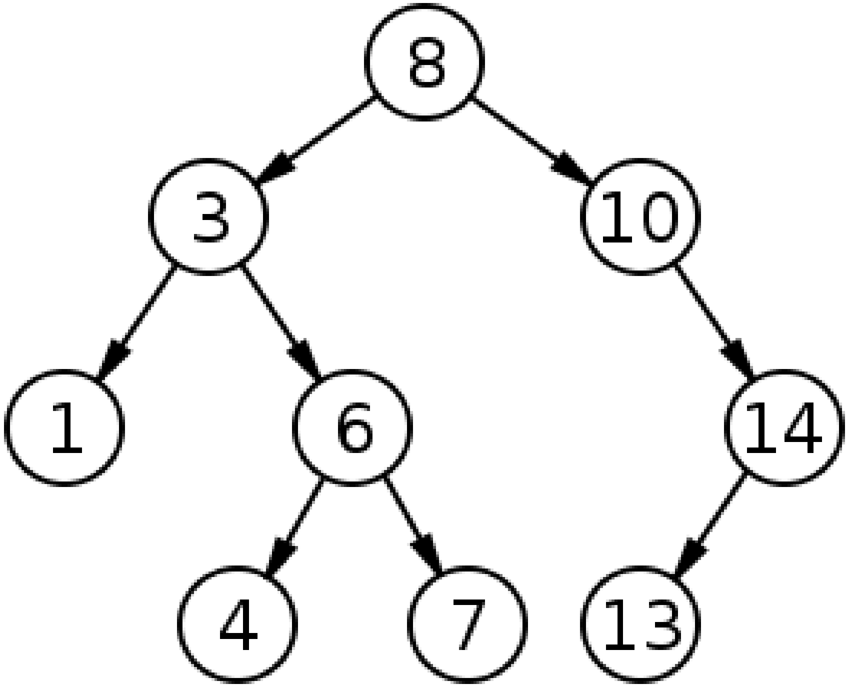
Complicated Removal

- Similar to a linked list, removal is often much more complicated than insertion or complete deletion

- We must first traverse the tree to find the target we want to remove
  - If we “disconnect” a link, we need to reestablish
- Possible scenarios
  - No children (leaf)
  - One child
  - Two children

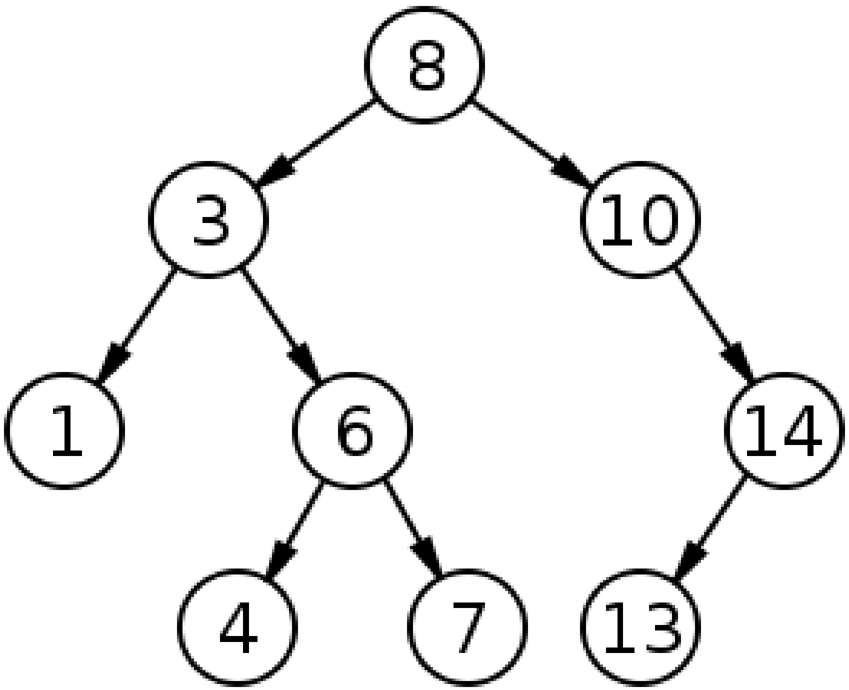
Removing A Node – Example 1

- Remove 4
- Any issues?



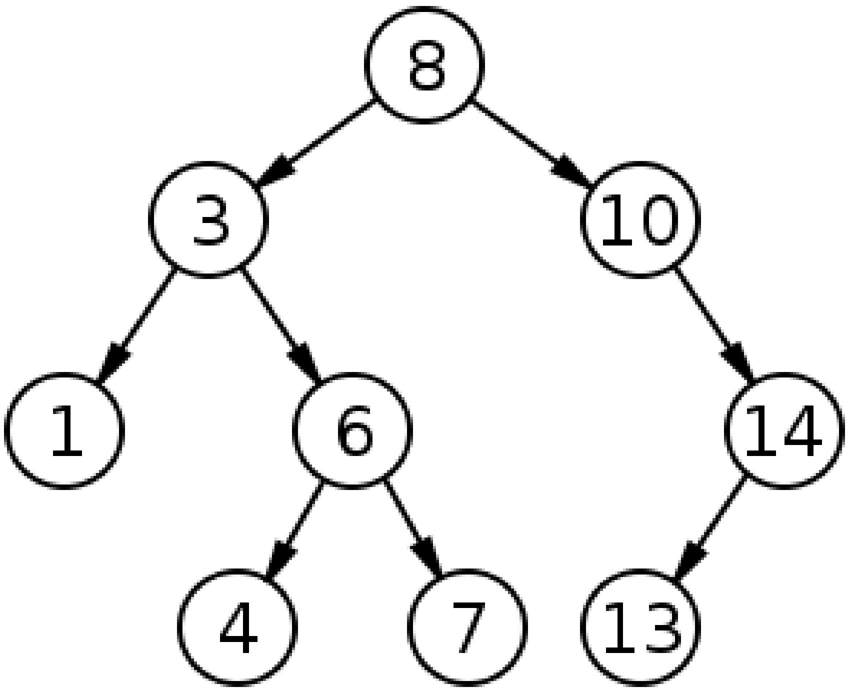
Removing A Node – Example 2

- Remove 6
- Any issues?



Removing A Node – Example 3

- Remove 8
- Any issues?



Removing a Node – No Children

- Simplest scenario for removal
  - No children to worry about managing
- Reminder: nodes with no children are leaves

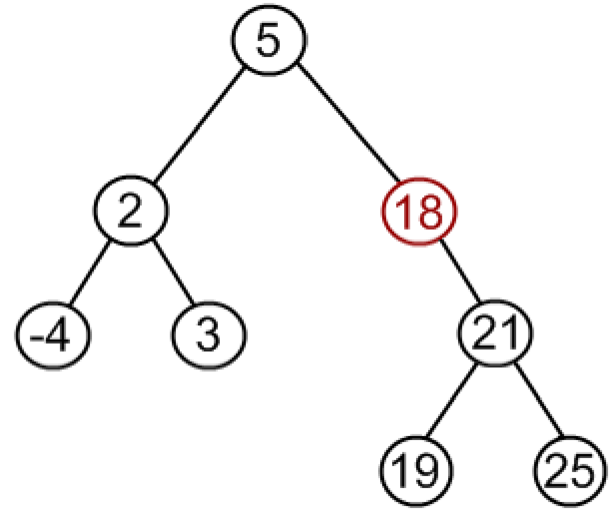
- We still have to find the target node first
- To remove a node with no children, we need to do the following:
  - Cut the link from the parent node
  - Free the memory

Removing a Node – One Child

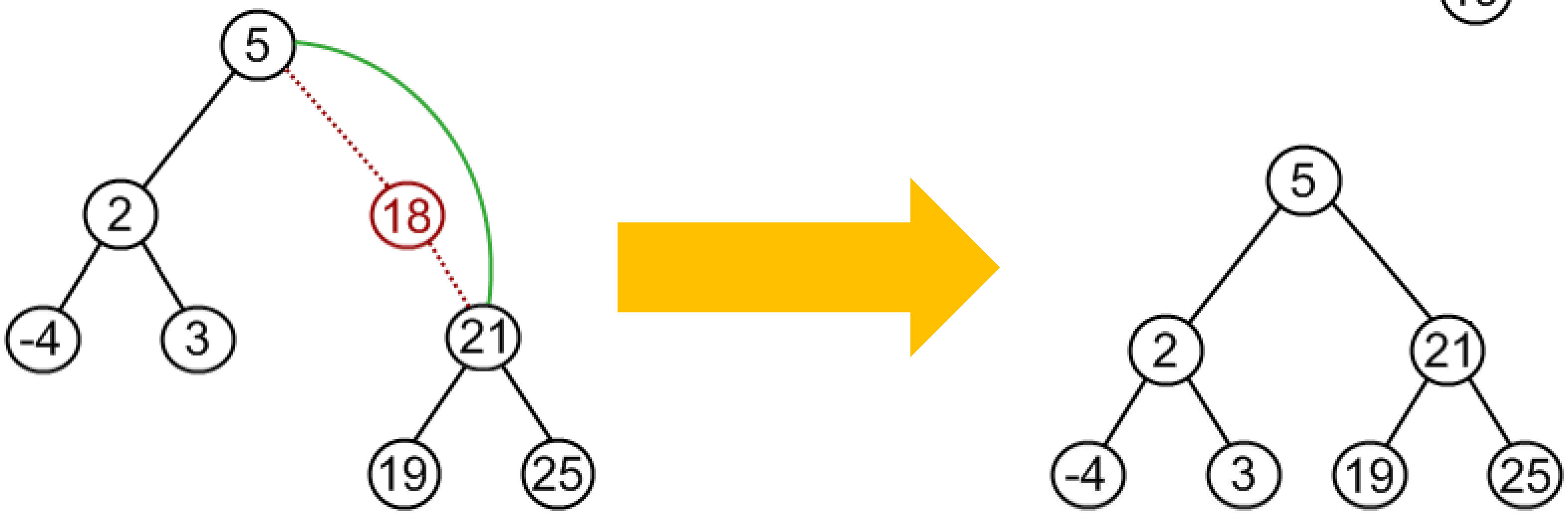
- Second easiest scenario for removal
  - Only one child is linked to the node
- The node can only be deleted after its parent adjusts the link to bypass the node to the child
  - The "grandparent" node takes custody
- To remove a node with one child, we need to do the following:
  - Connect node's parent to its child (custody)
  - Free the memory

Example Removal – One Child

- Remove "18" from this BST:



- Grandparent takes custody



Source: http://www.algolist.net/Data_structures/Binary_search_tree/Removal

Code for Removal

```
void remove( const Comparable & x, BinaryNode * & t )
{
    // code to handle two children prior to this

    else
    {
        // "hold" the position of node we'll delete
        BinaryNode *oldNode = t;

        // ternary operator
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Removing a Node – Two Children

- Most difficult scenario for removal
  - Everyone in the subtree will be affected
- Instead of completely deleting the node, we will replace its value with another node's
  - The smallest value in the right subtree
  - Use **`findMin()`** to locate this value
  - Then delete the node whose value we moved
- Using the minimum of a subtree ensures it does not also have two children to handle

Remove Function

```
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) { return; }  // item not found; do nothing

    // continue to traverse until we find the element
    if( x < t->element ) { remove( x, t->left ); }
    else if( t->element < x ) { remove( x, t->right ); }

    else if( t->left != NULL && t->right != NULL ) // two children
    {
        // find right's lowest value
        t->element = findMin( t->right )->element;
        // now delete that found value
        remove( t->element, t->right );
     }
     else // zero or one child
     {
          BinaryNode *oldNode = t;
          // ternary operator
          t = ( t->left != NULL ) ? t->left : t->right;
          delete oldNode;
     }
}
```

Printing a Tree

```
void printTree( )
```

Printing a Tree

- Printing is simple – only question is which order we want to traverse the tree in?

```
// ostream &out is the stream we want to print to
// (it maybe cout, it may be a file - our choice)
void printTree( BinaryNode *t, ostream & out ) const
{
    // if the node isn't null
    if( t != NULL )
    {
        // print an in-order traversal
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}
```

Performance
Run Time of BST Operations

Big O of BST Operations

Operation	Big O
`contains( x )`	*O(log n)*
`insert( x )`	*O(log n)*
`remove( x )`	*O(log n)*
`findMin/findMax( x )`	*O(log n)*
`isEmpty( )`	*O(1)*
`printTree( )`	*O(n)*