Lecture 16

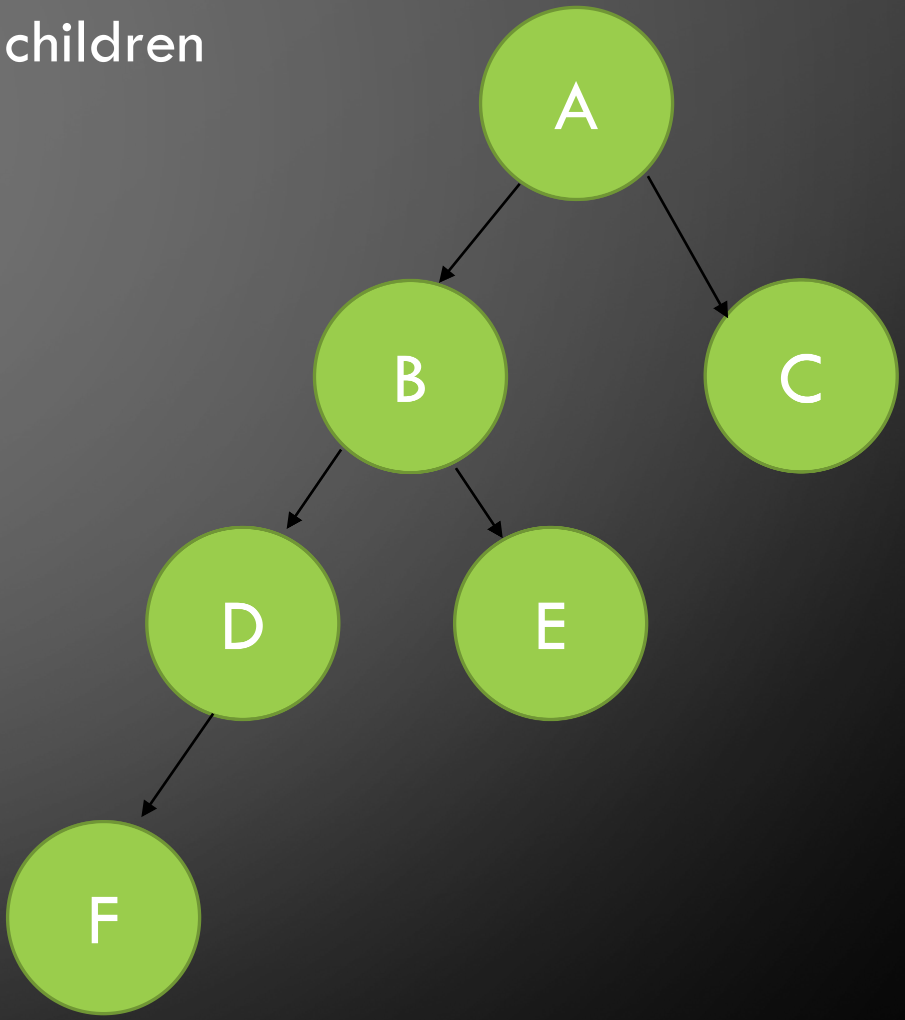
TREE IMPLEMENTATIONS

Outline

- General Implementation
- Array-based Implementation
- Linked-list Implementation
- Tree Sort

Recall Definition of Binary Tree

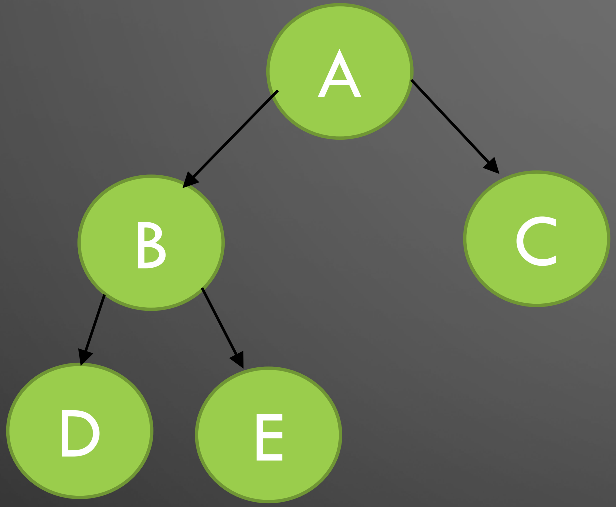
- Binary Tree: One root, each node has 0, 1, or 2 children



Binary Tree

Representing Binary Trees

- Array-based implementation for complete trees
  - (Recall definition of complete tree: The level above the height of the tree is full; all nodes are as far left as possible)



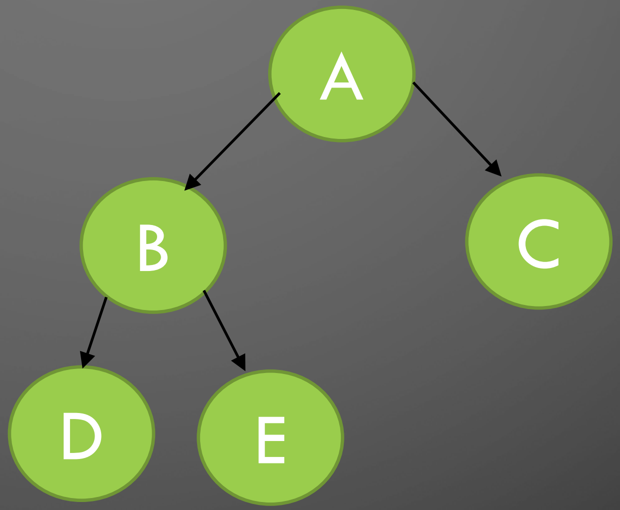
A	B	C	D	E

- Why does this not work for non-complete trees?

Linked list

- Pointer-based implementation
  - Root is a *struct node*, which has two linked-list children *struct node*

```
struct node
{
    item value;
    node * left;
    node * right;
}
```



Binary Tree Implementation

- This is an example of a binary tree with many operators to allow fine-grained control. Typically, a sparser level of control in used, in what is referred to as a binary search tree

```
// binary tree operations:
bool isEmpty() const;
TreeItemType rootData() const;
void setRootData(const TreeItemType& newItem);
void attachLeft(const TreeItemType& newItem);
void attachRight(const TreeItemType& newItem);
void attachLeftSubtree(BinaryTree& leftTree);
void attachRightSubtree(BinaryTree& rightTree);
void detachLeftSubtree(BinaryTree& leftTree);
void detachRightSubtree(BinaryTree& rightTree);
BinaryTree leftSubtree() const;
BinaryTree rightSubtree() const;
void preorderTraverse(FunctionType visit);
void inorderTraverse(FunctionType visit);
void postorderTraverse(FunctionType visit);
```

Public Methods

```
// Copies the tree rooted at treePtr into a tree rooted
// at newTreePtr. Throws TreeException if a copy of the
// tree cannot be allocated.
void copyTree(NodeType* treePtr, NodeType*& newTreePtr) const;

// Copies the tree rooted at treePtr into a tree rooted
// at newTreePtr. Throws TreeException if a copy of the
// tree cannot be allocated.
void destroyTree(NodeType*& treePtr);

NodeType* rootPtr() const { return root; };

void setRootPtr(NodeType* newRoot);

// The next two functions retrieve and set the values
// of the left and right child pointers of a tree node.
void getChildPtrs(NodeType* nodePtr, NodeType*& leftChildPtr,
    NodeType*& rightChildPtr) const;
void setChildPtrs(
    NodeType* nodePtr, NodeType* leftChildPtr, NodeType* rightChildPtr);

void preorder(NodeType* treePtr, FunctionType visit);
void inorder(NodeType* treePtr, FunctionType visit);
void postorder(NodeType* treePtr, FunctionType visit);
```

Protected Methods

Binary Search Tree Operations

```cpp
template <typename KeyType, typename ValueType>
class AbstractBST
{
public:
    // determine if the tree is empty
    virtual bool is_empty() = 0;

    // Search for key.
    // If found is true returns the value associated with that key.
    // If found is false, returns a default constructed ValueType
    virtual ValueType search(const KeyType& key, bool& found) = 0;

    // Insert value into the BST with unique key.
    virtual void insert(const KeyType& key, const ValueType& value) = 0;

    // Remove value from the BST with key.
    virtual void remove(const KeyType& key) = 0;
};
```

Binary Search Tree

- Map sorted list operations onto binary search tree
- Insert and delete can use the binary structure
  - More efficient
  - Better than binary search on a linked list
- Consider items of TreeItemType to have an associated key of KeyType
  - This is how you sort, based on the key

Binary Search Tree ADT

- +createBST() +destroyBST() +isEmpty():bool
  - Create empty BST Destroy BST Check if BST is empty
- +insert(in newItem:TreeItemType): bool
  - Insert newItem into the BST based on key value and reports success/failure
- +remove(in searchKey:KeyItemType): bool
  - Delete item with searchKey from BST, report success/failure
- +retrieve(in searchKey: KewyItemType, out treeItem:TreeItemType): bool
  - Get item with searchKey from BST, report success/failure
- +preorderTraverse() +inorderTraverse() +postorderTraverse()
  - Argument in visit:FunctionType – what to do when you visit each node

Pseudo-code for searching

If n < root go left; if n > root go right

```
search(intree:BinarySearchTree, in key:KeyItemType)
if (tree.isEmpty())
        no item found
if (key == key of the root)
        item found
else if (key < key of the root)
        search (leftsubtreeof tree, key)
else
        search (rightsubtreeof tree, key)
```

What is the complexity of the search?

How to insert into the BST to maintain ordering?

Pseudo code: search

if search fails

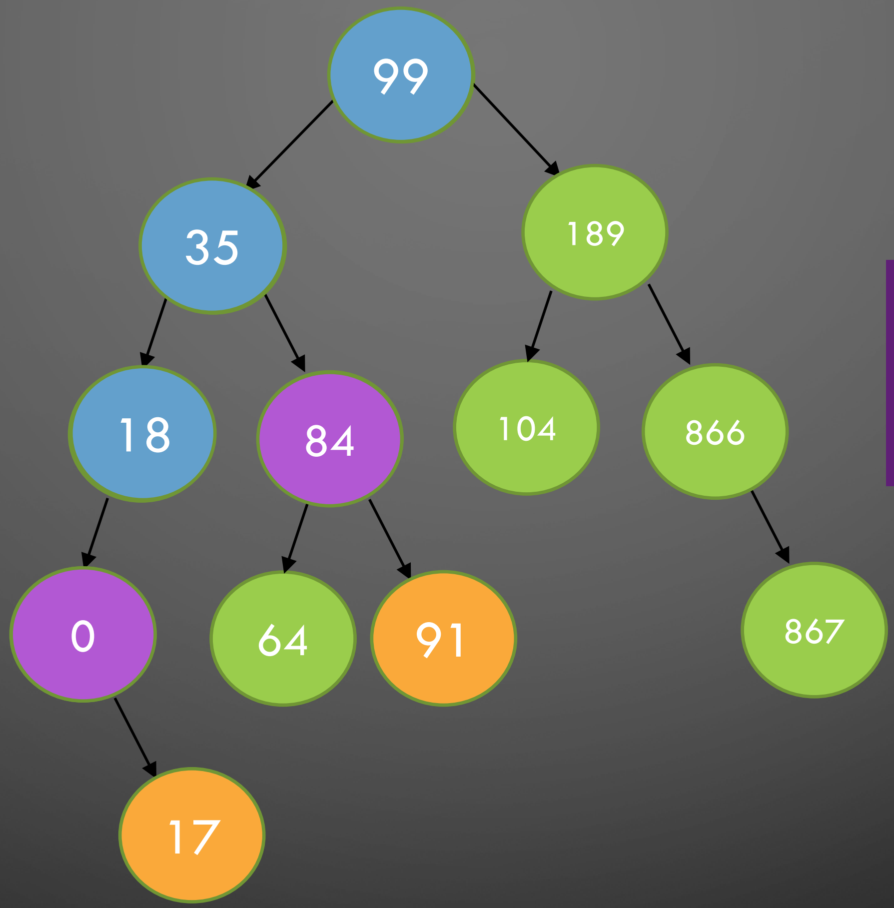
  if key < last visited

    insert at leftsubtree

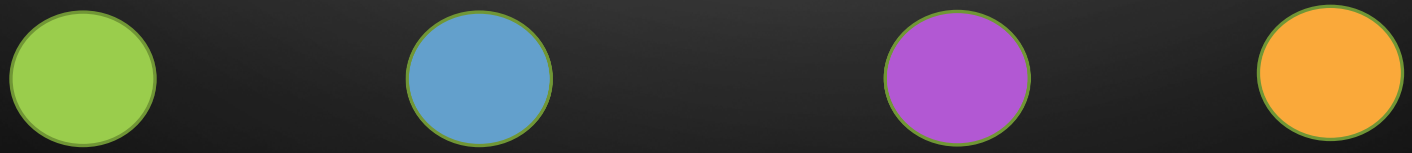
  if key > last visited

    insert at right subtree

- Insert 91
- Insert 17



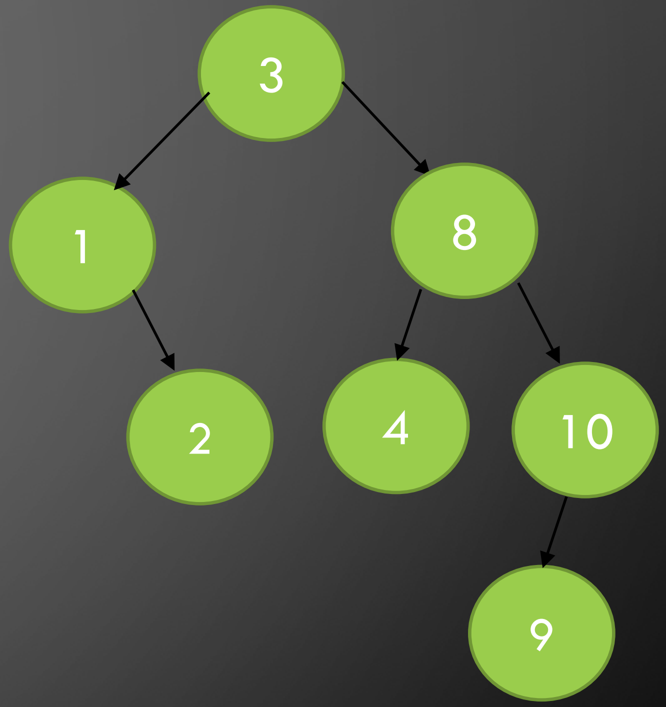
What is the complexity of the insert?



Not traversed   Traversed left subtree   Traversed right subtree   Inserted

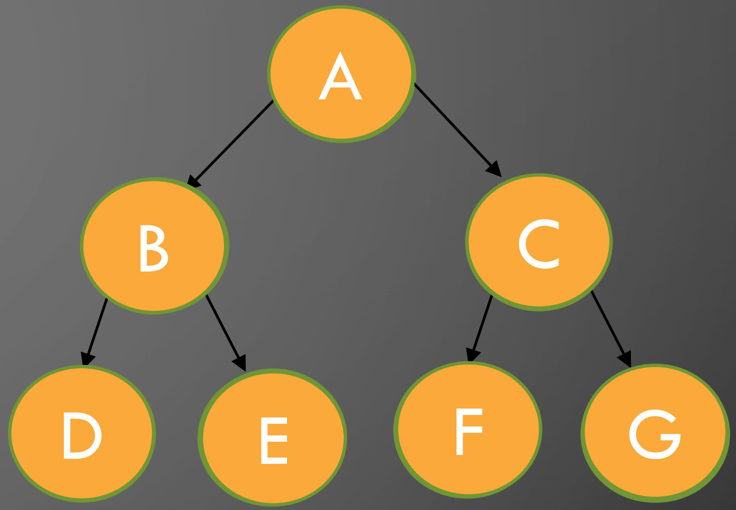
How to delete from the BST?

- What if we try to delete node 4?
- What if we try to delete node 1?
  - Not too bad, right?
  - Sort of swap in node 2 for 1…
- What if we try to delete node 10?
  - Same deal, swap in node 9 for 10
- What if we try to delete node 8?
  - Not…simple…need a better solution



Recall In-order Traversal

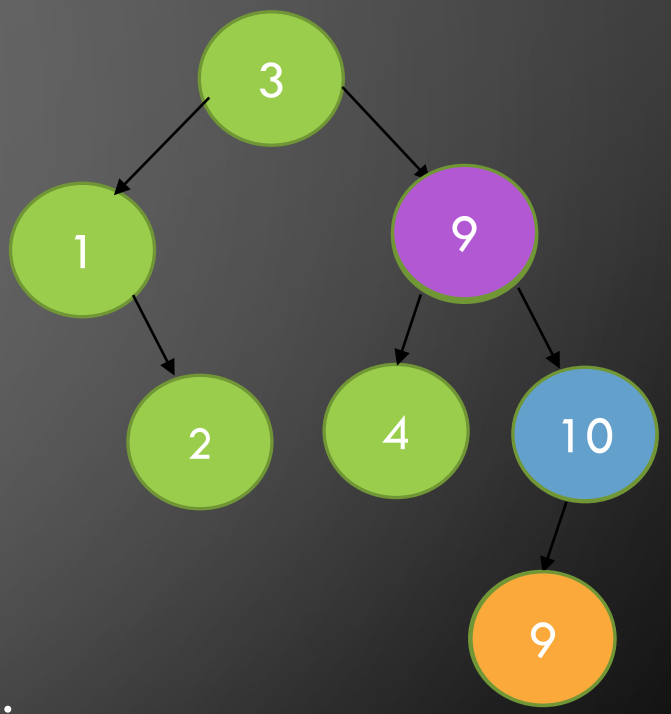
- *In-order traversal*
  - If Tree is not empty
    - In-order traverse left subtree of T
    - Visit the root of T
    - In-order traverse right subtree of T



ORDER: D, B, E, A, F, C, G,

How to delete from the BST?

- Delete node 8
  - Maybe we can find a better node to delete?
- By definition: the inorder successor would be an easier node to remove
- Algorithm:
  - find the inorder successor
    - The inorder successor is the leftmost node of the right subtree
  - Copy the data from the inorder successor to the node flagged for deletion
  - Delete the inorder successor
  - Node 8 is deleted



What is the complexity of the delete op?

Pseudocode for delete

```
delete(in key:KeyItemType)
if( search for key fails)
  delete fails
else
  if (found node is leaf)
    delete it
  if (found node has left/right child only)
    delete node, replace with left/right child,
  else
    find inorder successor, copy to found node
    delete inorder successor
```

Tree Sort

1. Start with an unsorted list
2. Create a Binary Search Tree by inserting data items from array into the tree
3. Perform in-order traversal to access the elements sorted

What is the complexity of tree sort?

- Average Case: $O(n*\log(n))$
- Worst Case: $O(n^2)$
- Extra Space: $O(n)$

Tree Sort

- For each element in unsorted array, insert it into a binary search tree

- Result will be a sorted binary tree

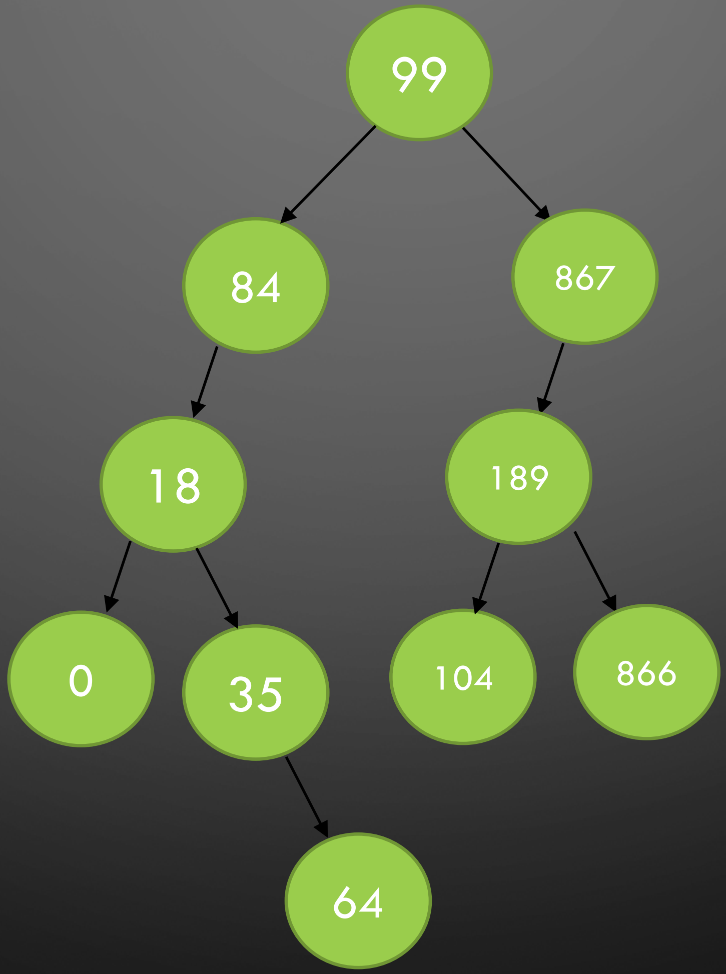
```c
/* A utility function to insert a new
   Node with given key in BST */
Node* insert(Node* node, int key)
{
    /* If the tree is empty, return a new Node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) Node pointer */
    return node;
}
```

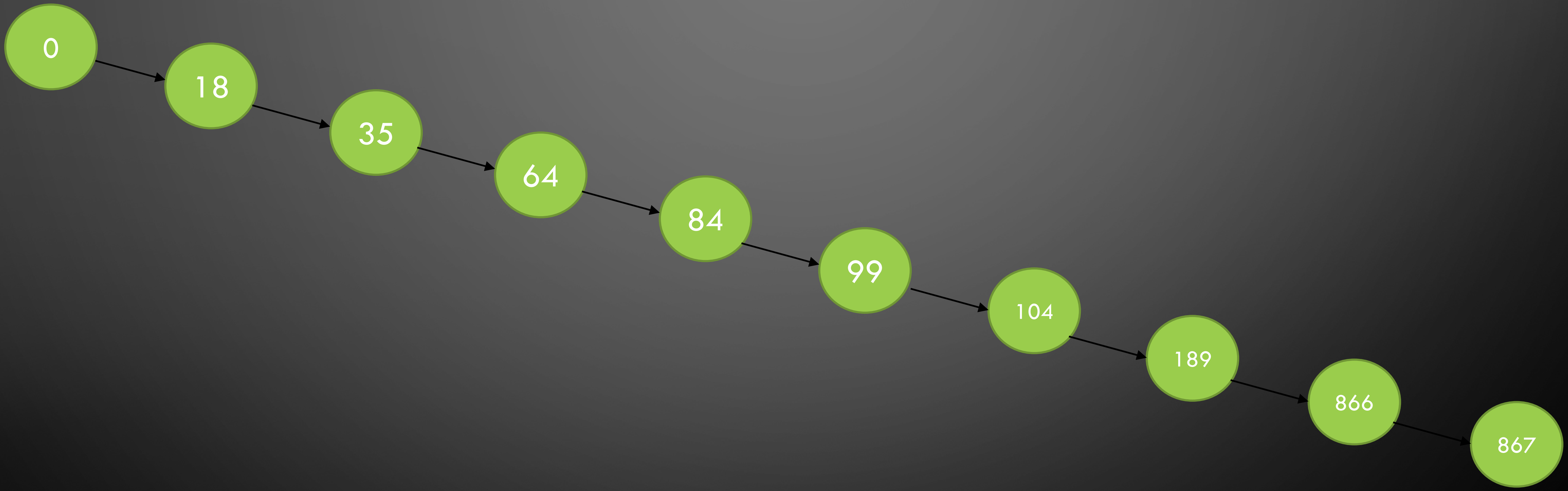
Binary Search Tree – unsorted

99	84	867	18	189	104	35	866	0	64



Binary Search Tree – sorted data

0	18	35	64	84	99	104	189	866	867



Assignment/Homework

- Reading pp. 515 -533
- ICE 7 due on Today.
- P4 due on Thursday.
- ICE 8 due on Thursday.
- Homework 6 released.