

Topic 8

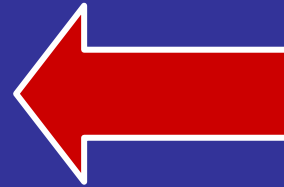
Information Retrieval

CSE1303 Part A

Data Structures and Algorithms

Overview

- Binary **Search** Trees.
- Hash Tables.

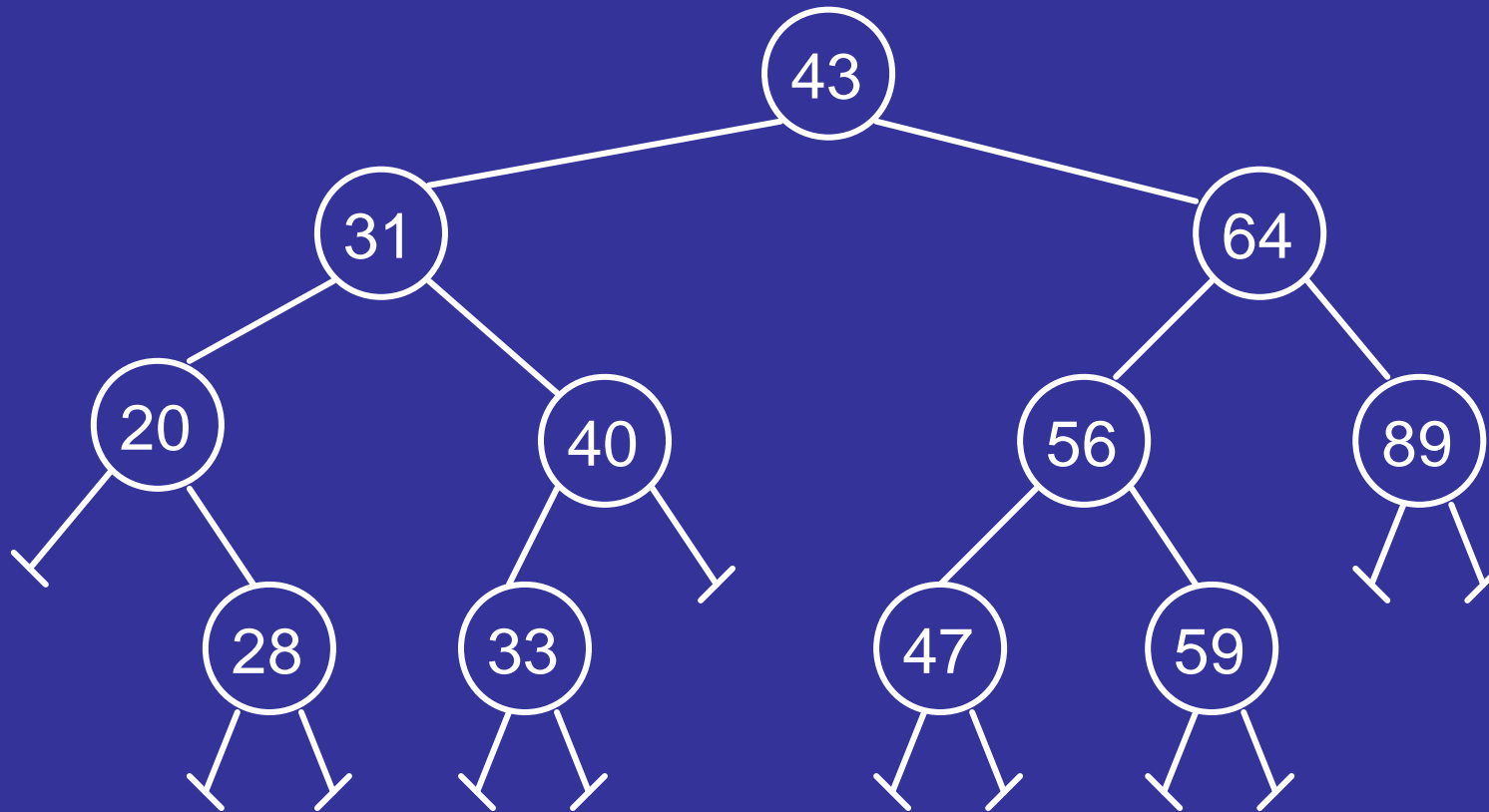


Recall - Binary Search Tree

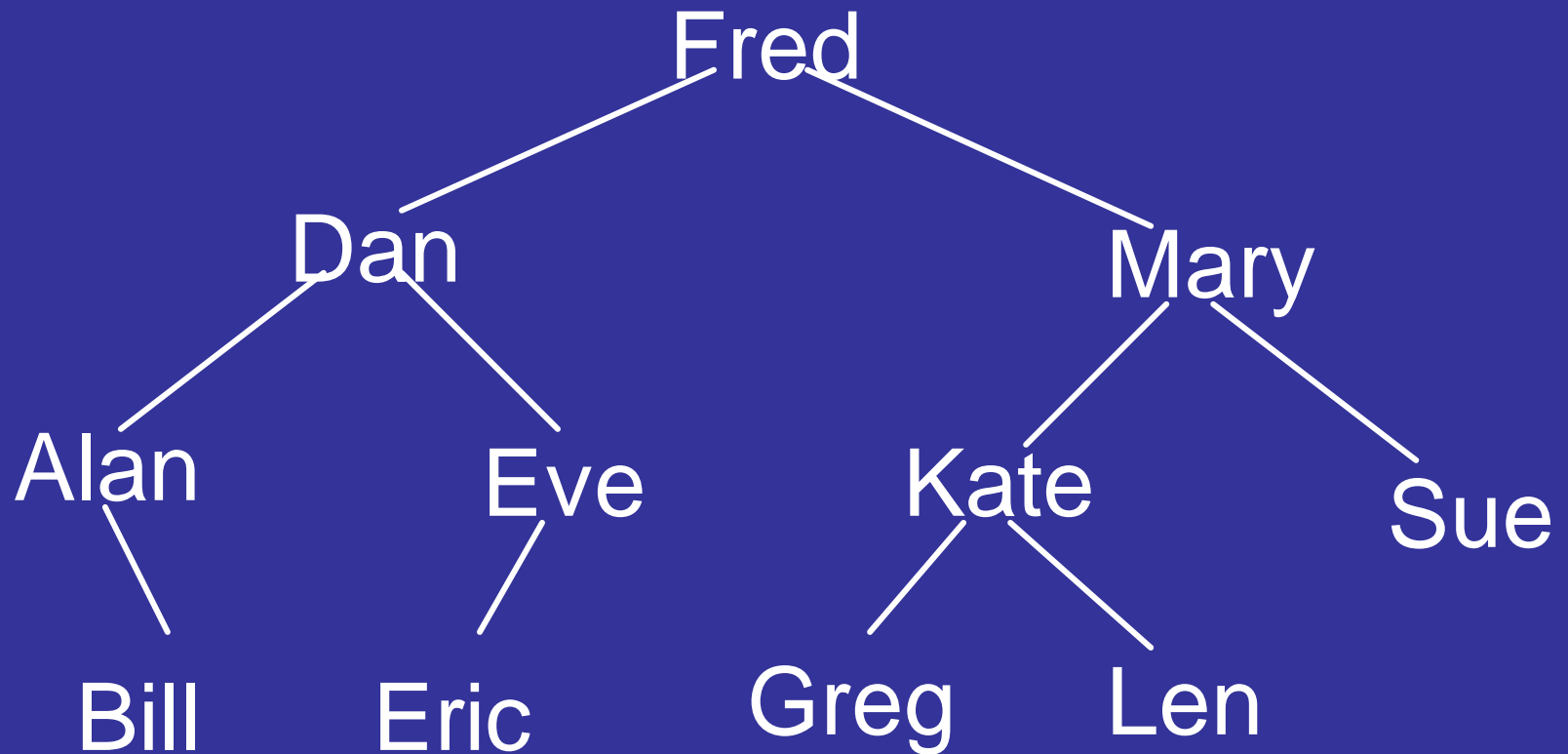
A Binary Tree such that:

- Every node entry has a **unique** key.
- **All** the keys in the **left subtree** of a node are **less** than the key of the node.
- **All** the keys in the **right subtree** of a node are **greater** than the key of the node.

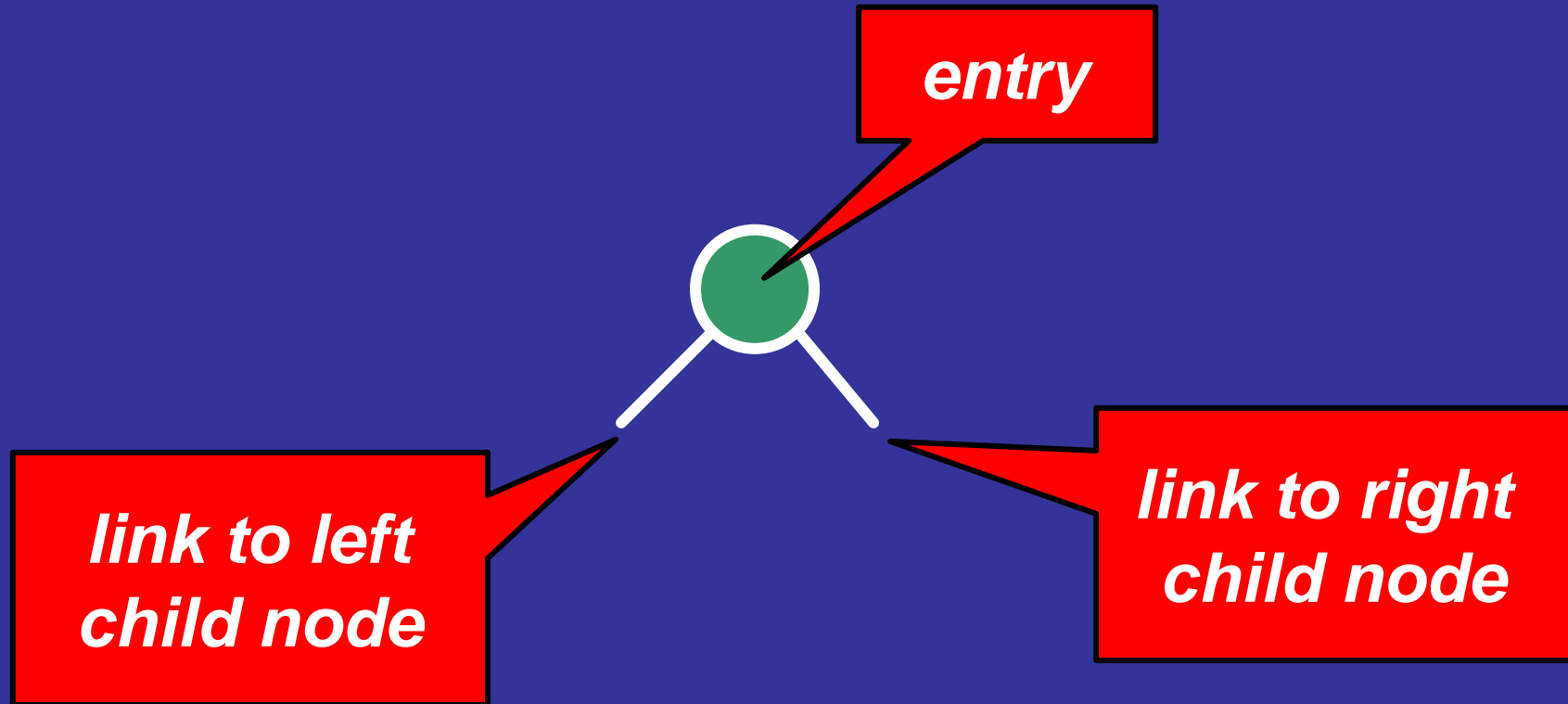
Example 1: *key is an integer*



Example 2: *key is a string*



Binary Tree Node



Binary Search Tree Node

Example 1:

```
struct TreeNodeRec
{
    int    key;

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Binary Search Tree Node

Example 2:

```
#define MAXLEN 15
```

```
struct TreeNodeRec
```

```
{
```

```
    char    key[MAXLEN];
```

```
    struct TreeNodeRec* leftPtr;
```

```
    struct TreeNodeRec* rightPtr;
```

```
};
```

```
typedef struct TreeNodeRec TreeNode;
```



Recall:

```
#define MAXLEN 15
```

```
struct TreeNodeRec  
{
```

```
    char    key[MAXLEN];
```

```
    struct TreeNodeRec* leftPtr;
```

```
    struct TreeNodeRec* rightPtr;
```

```
};
```

```
typedef struct TreeNodeRec TreeNode;
```

*maximum
string length
is fixed*

Example 3:

```
struct TreeNodeRec
{
    char*   key;

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```



Recall:

```
struct TreeNodeRec
```

```
{
```

```
    char*    key;
```

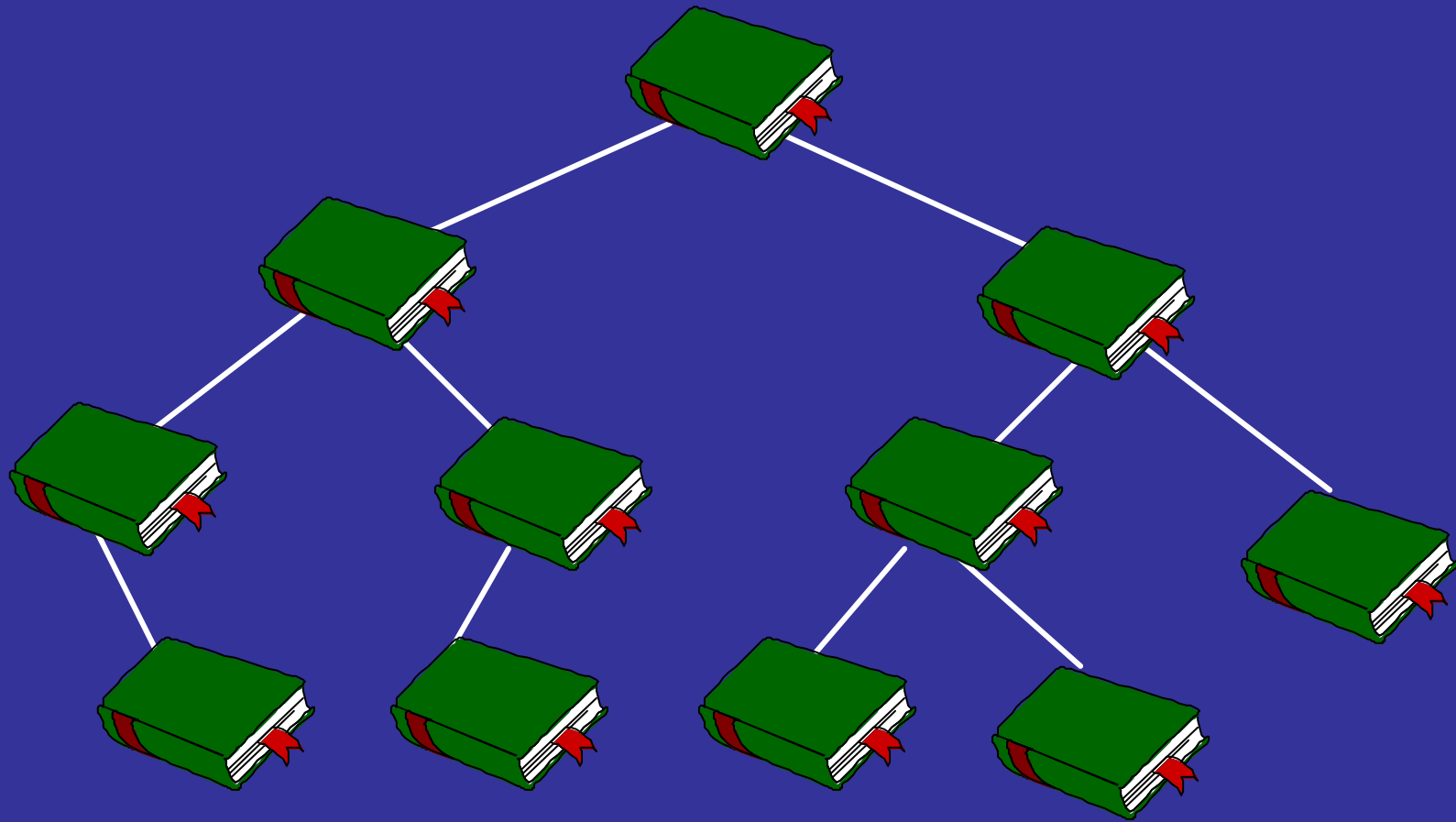
```
    struct TreeNodeRec*    left;
```

```
    struct TreeNodeRec*    right;
```

```
};
```

```
typedef struct TreeNodeRec TreeNode;
```

- Allows strings of arbitrary length.
- Memory needs to be allocated dynamically before use.
- Use **strcmp** to compare strings.

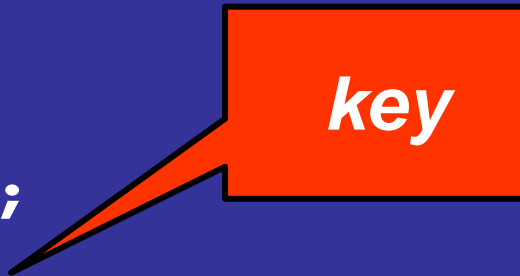


Book Record

```
struct BookRec
{
    char*   author;
    char*   title;
    char*   publisher;

    /* etc.: other book information. */
};

typedef struct BookRec Book;
```



A red speech bubble with a black outline and a tail pointing to the `title` field in the struct definition. The word `key` is written in white inside the bubble.

Example 4: *Binary Search Tree Node*

```
struct TreeNodeRec
{
    Book    info;

    struct TreeNodeRec*    leftPtr;
    struct TreeNodeRec*    rightPtr;
};

typedef struct TreeNodeRec    TreeNode;
```

Tree Node

```
struct TreeNodeRec
{
    float    key;

    struct TreeNodeRec*    leftPtr;
    struct TreeNodeRec*    rightPtr;
};

typedef struct TreeNodeRec    TreeNode;
```

```
#ifndef TREE_H
#define TREE_H

struct TreeNodeRec
{
    float          key;
    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;

TreeNode* makeTreeNode(float value);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void printInorder(const TreeNode* nodePtr);
void printPreorder(const TreeNode* nodePtr);
void printPostorder(const TreeNode* nodePtr);

#endif
```

MakeNode

- **parameter:** item to be inserted
- **steps:**
 - allocate **memory** for the new node
 - check if memory allocation is **successful**
 - if so, put **item** into the new node
 - set **left** and **right** branches to **NULL**
- **returns:** pointer to (i.e. address of) new node

```

TreeNode*
makeTreeNode(float value)
{
    TreeNode* newNodePtr = NULL;

    newNodePtr = (TreeNode*)malloc(sizeof(TreeNode));

    if (newNodePtr == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    else
    {
        newNodePtr->key = value;
        newNodePtr->leftPtr = NULL;
        newNodePtr->rightPtr = NULL;
    }
    return newNodePtr;
}

```

value

3.3

newNodePtr

NULL

newNodePtr

0x2000



0x2000



newNodePtr

0x2000



0x2000



NULL

NULL

Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{

}

}
```



*initially, pointer
to root node*

Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{

    traverse left sub-tree
    visit the node
    traverse right sub-tree

}
```

Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{

    printInorder(nodePtr->leftPtr);
    printf(" %f, nodePtr->key);
    printInorder(nodePtr->rightPtr);

}
```

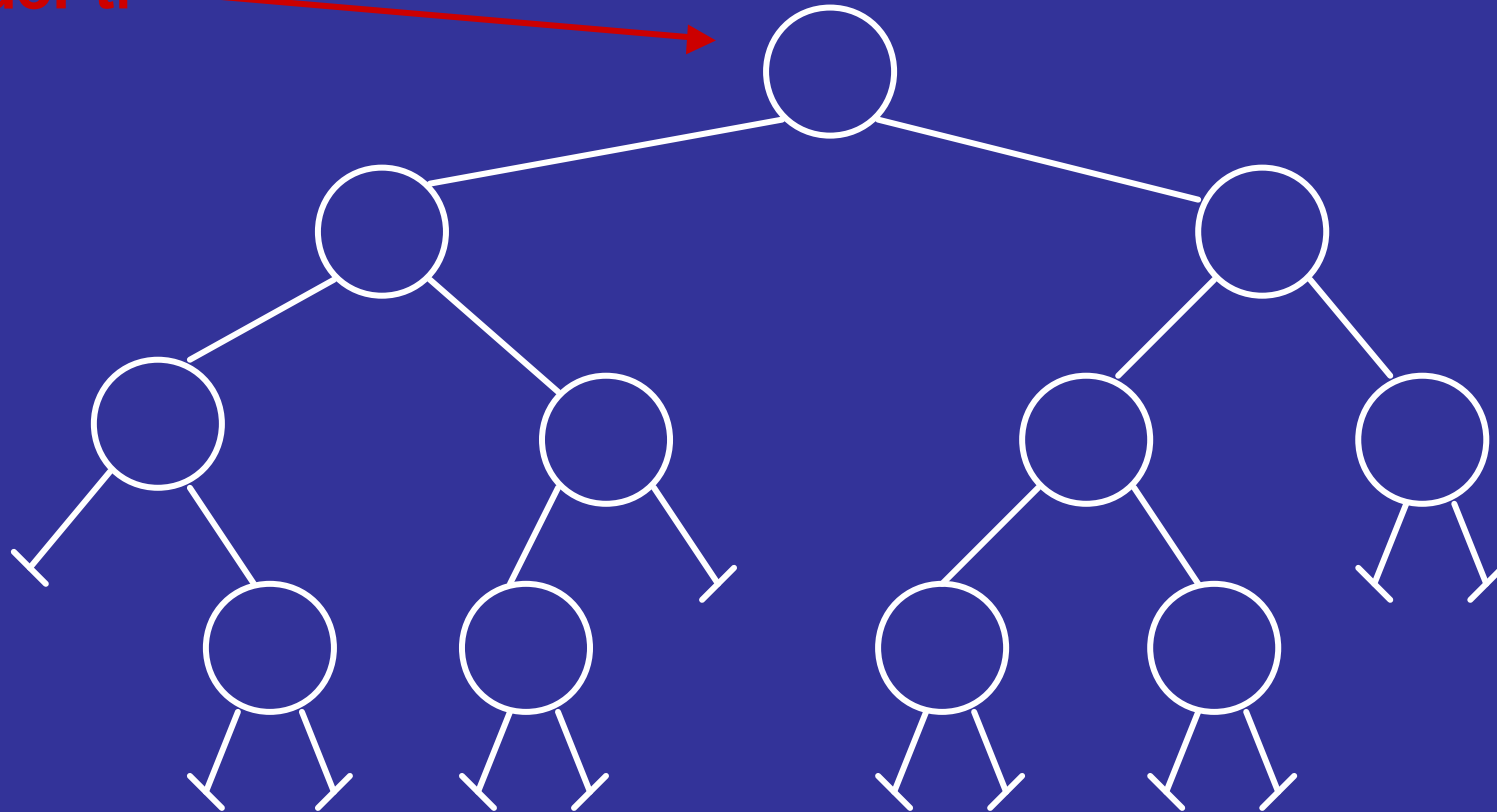
Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{
    if (nodePtr != NULL)
    {
        printInorder(nodePtr->leftPtr);
        printf(" %f", nodePtr->key);
        printInorder(nodePtr->rightPtr);
    }
}
```

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){
  if (nodePtr != NULL){
    printInorder(nodePtr->leftPtr);
    printf(" %f", nodePtr->key);
    printInorder(nodePtr->rightPtr);
  }
}
```

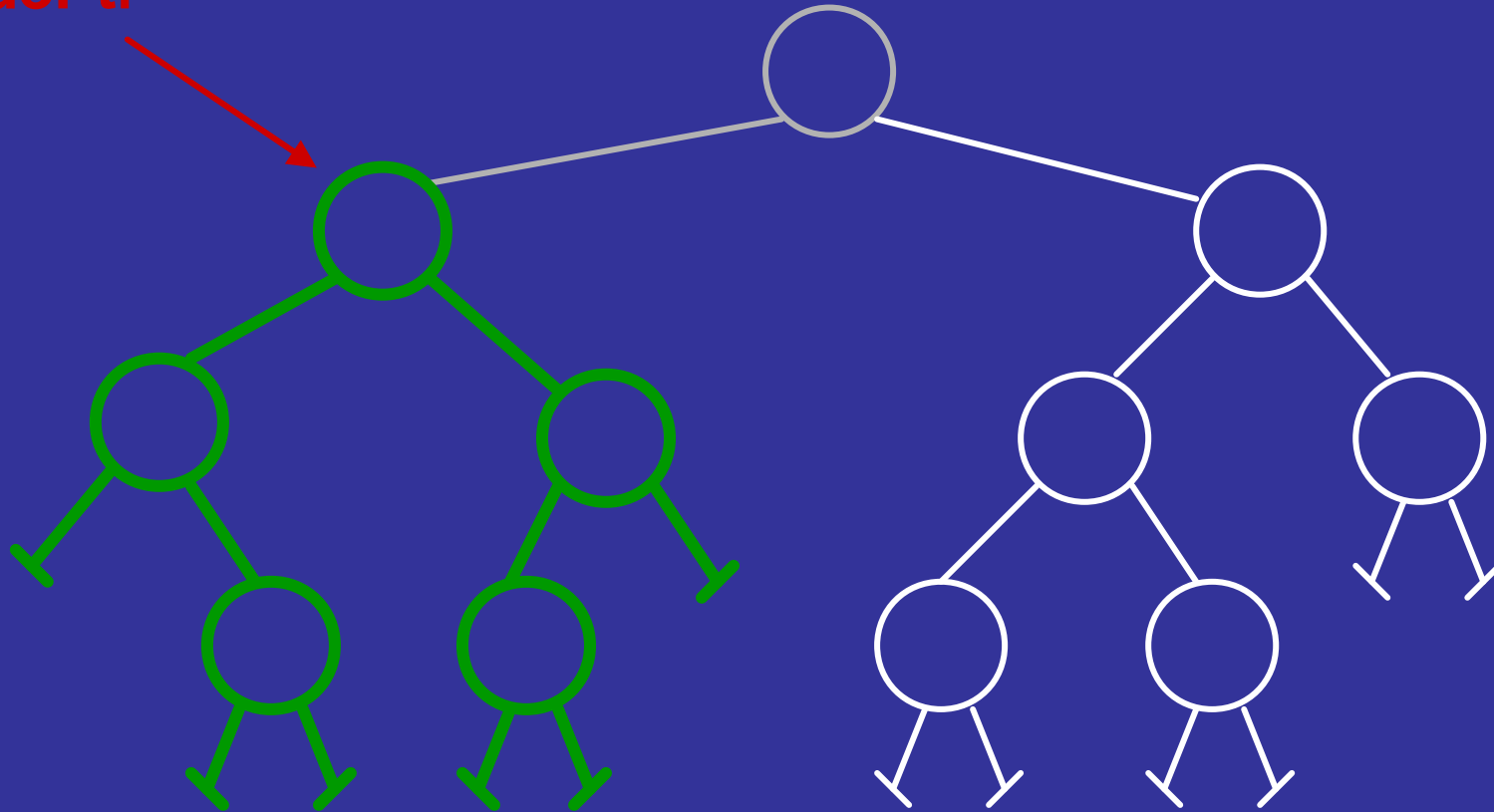
2/7/02

CSE1303 Part A

24

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

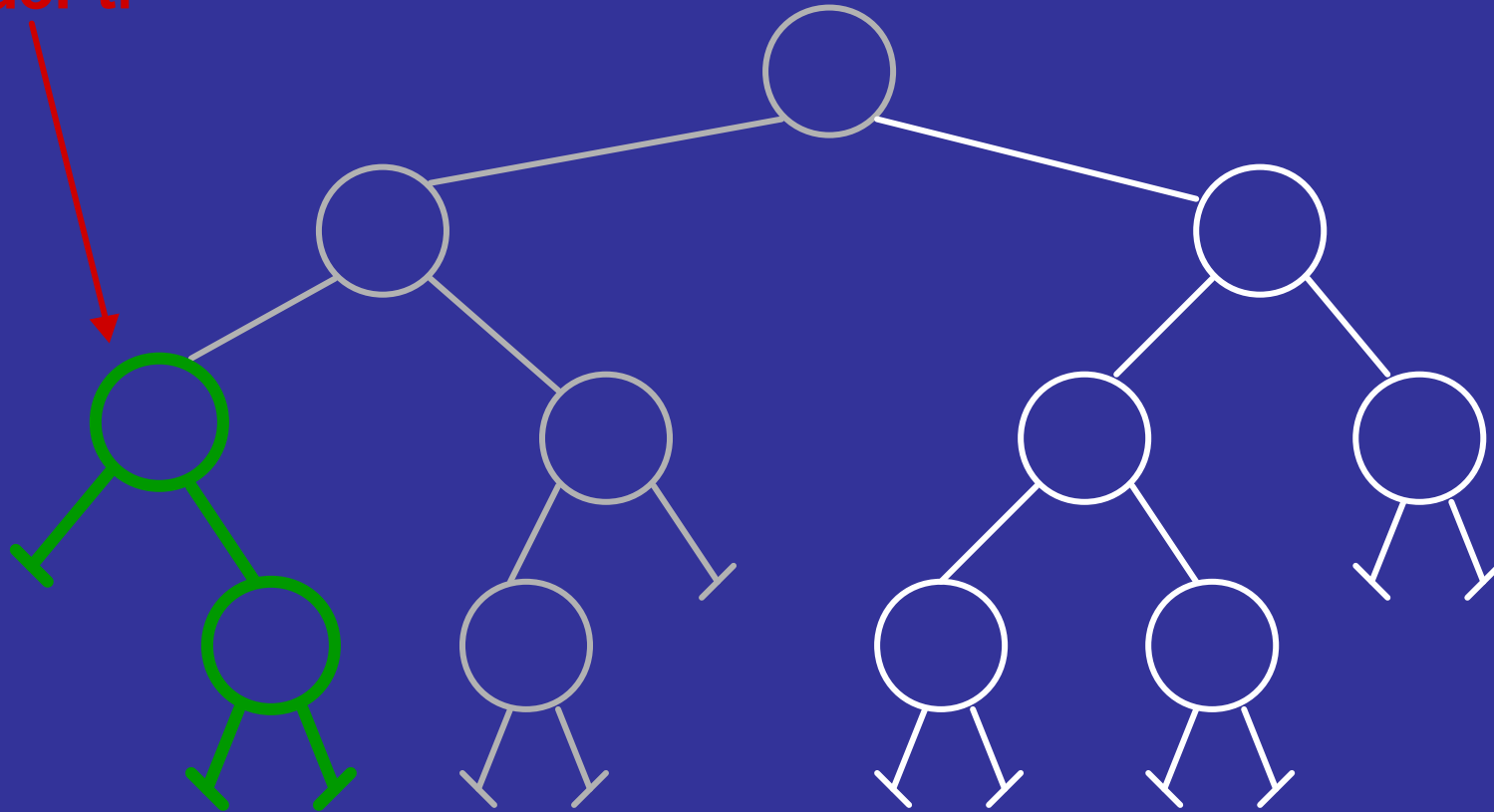
2/7/02

CSE1303 Part A

25

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

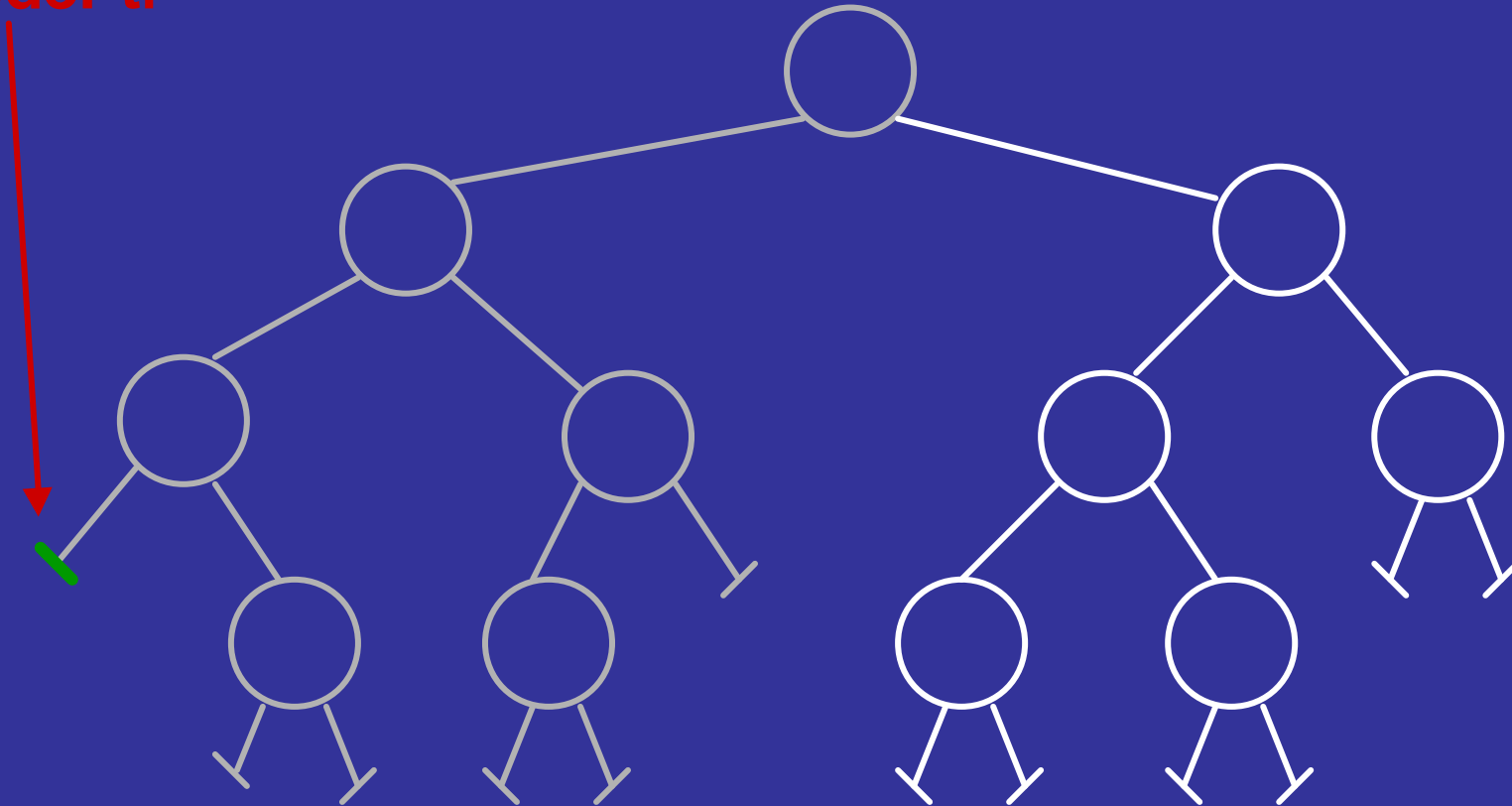
2/7/02

CSE1303 Part A

26

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

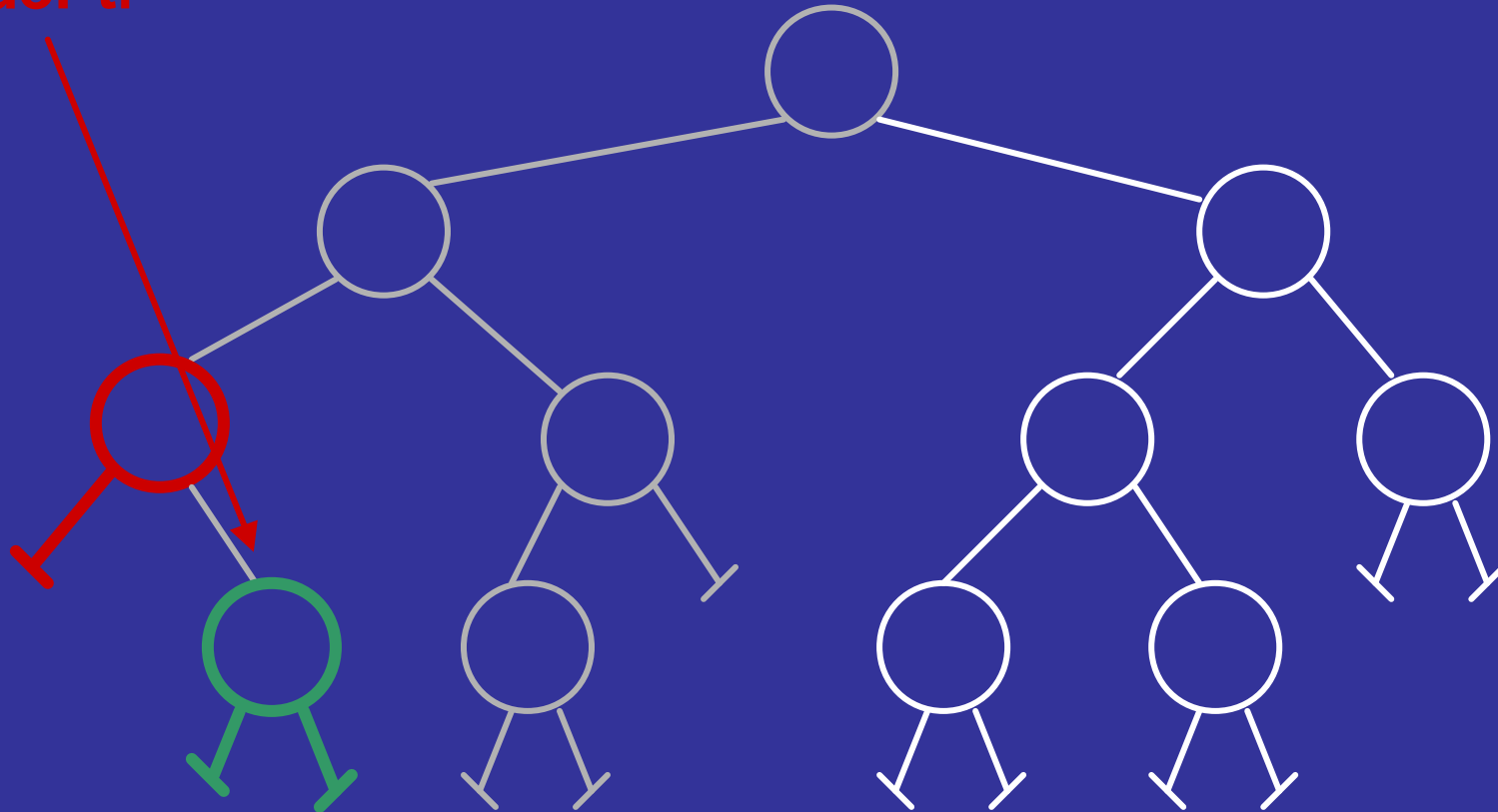
2/7/02

CSE1303 Part A

27

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

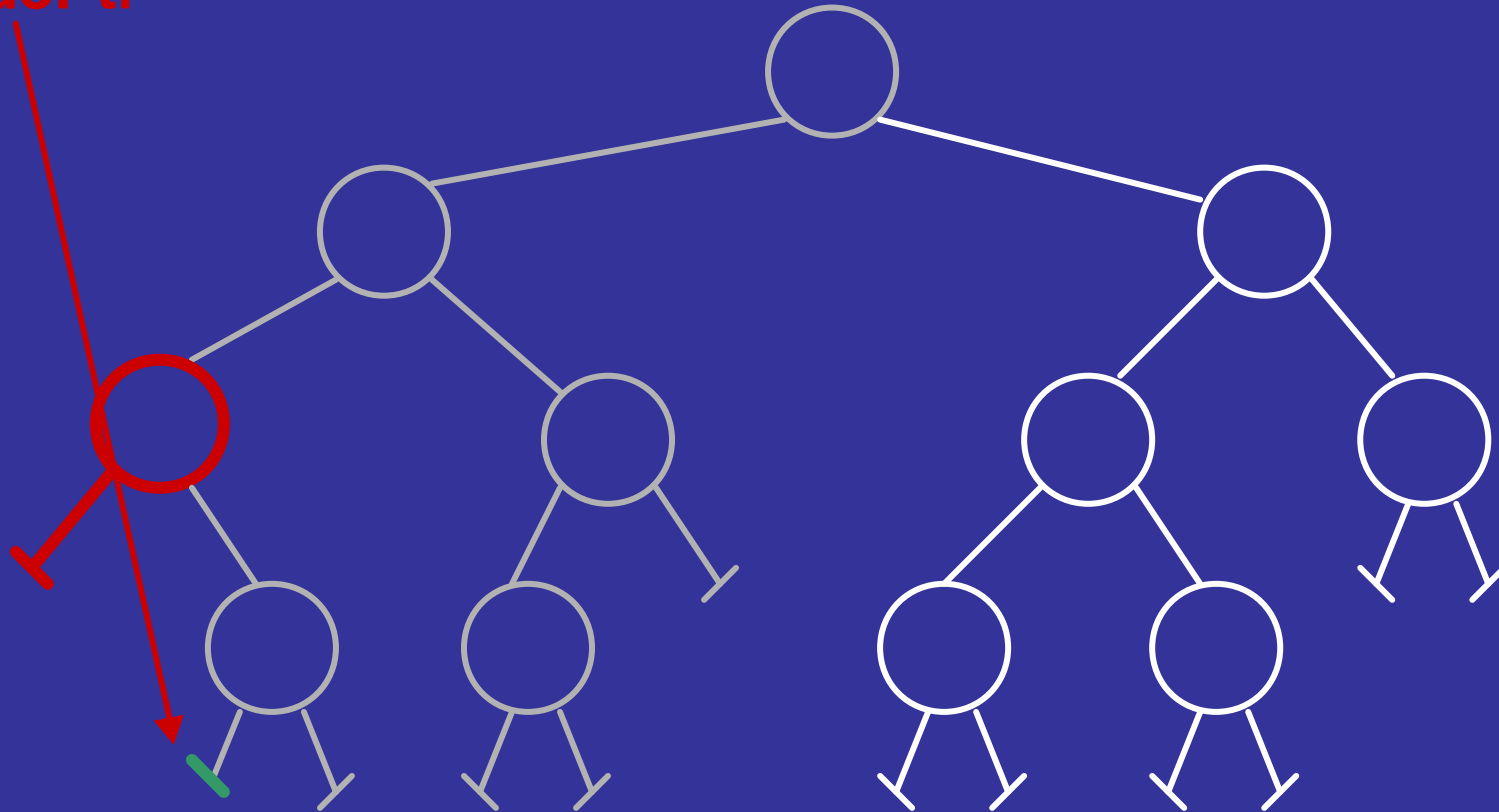
2/7/02

CSE1303 Part A

28

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

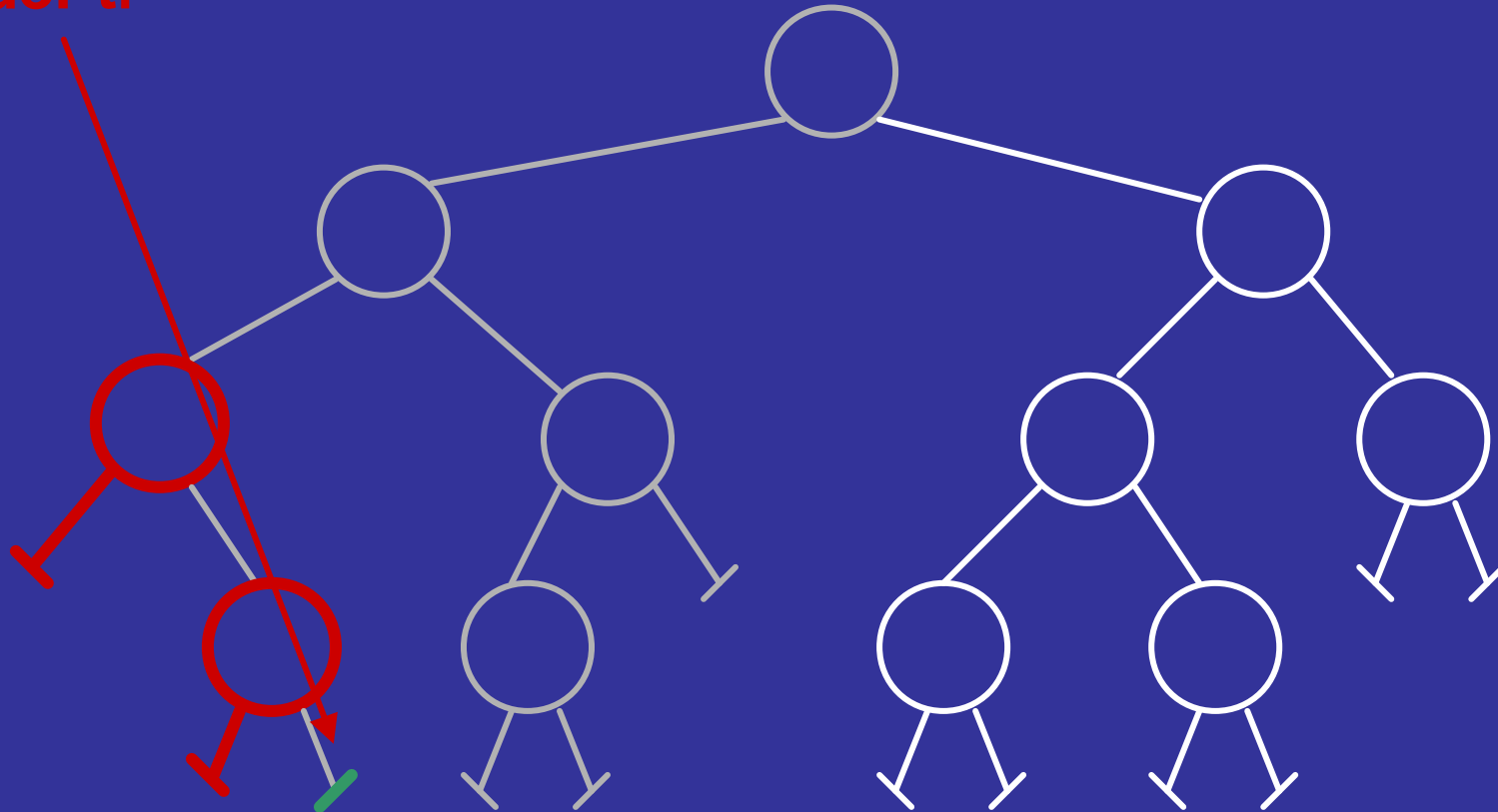
2/7/02

CSE1303 Part A

29

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

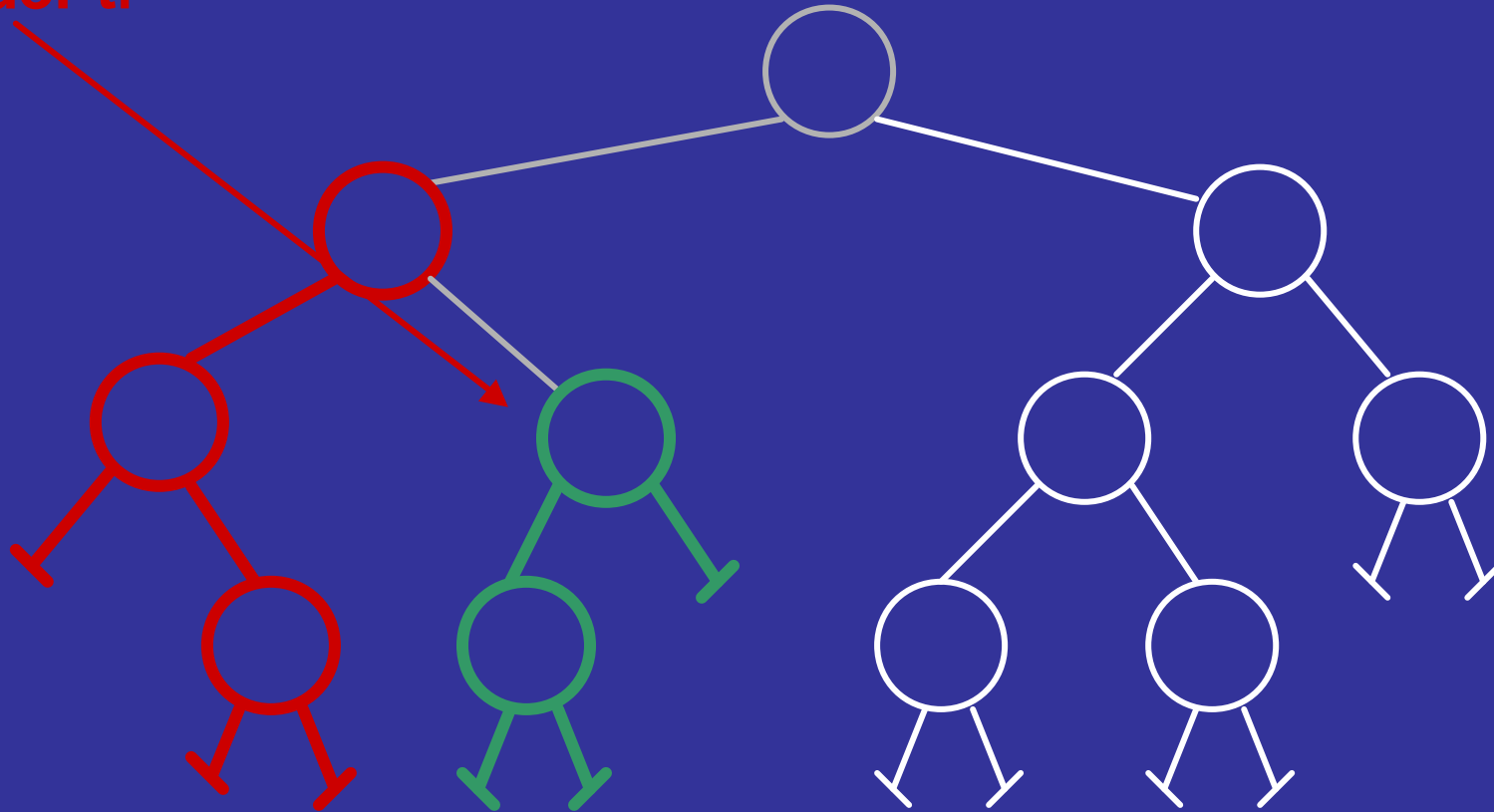
2/7/02

CSE1303 Part A

30

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

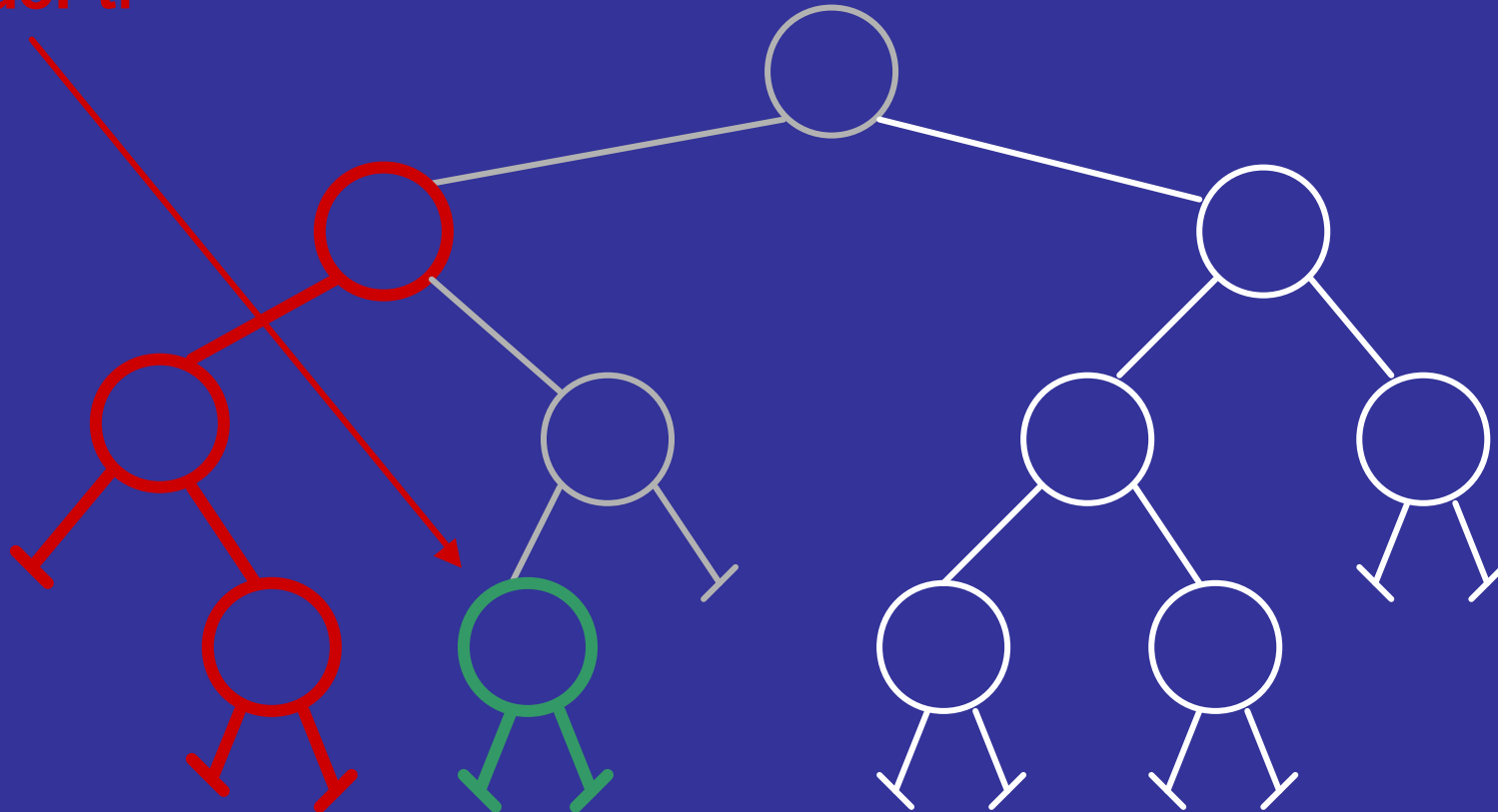
2/7/02

CSE1303 Part A

31

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

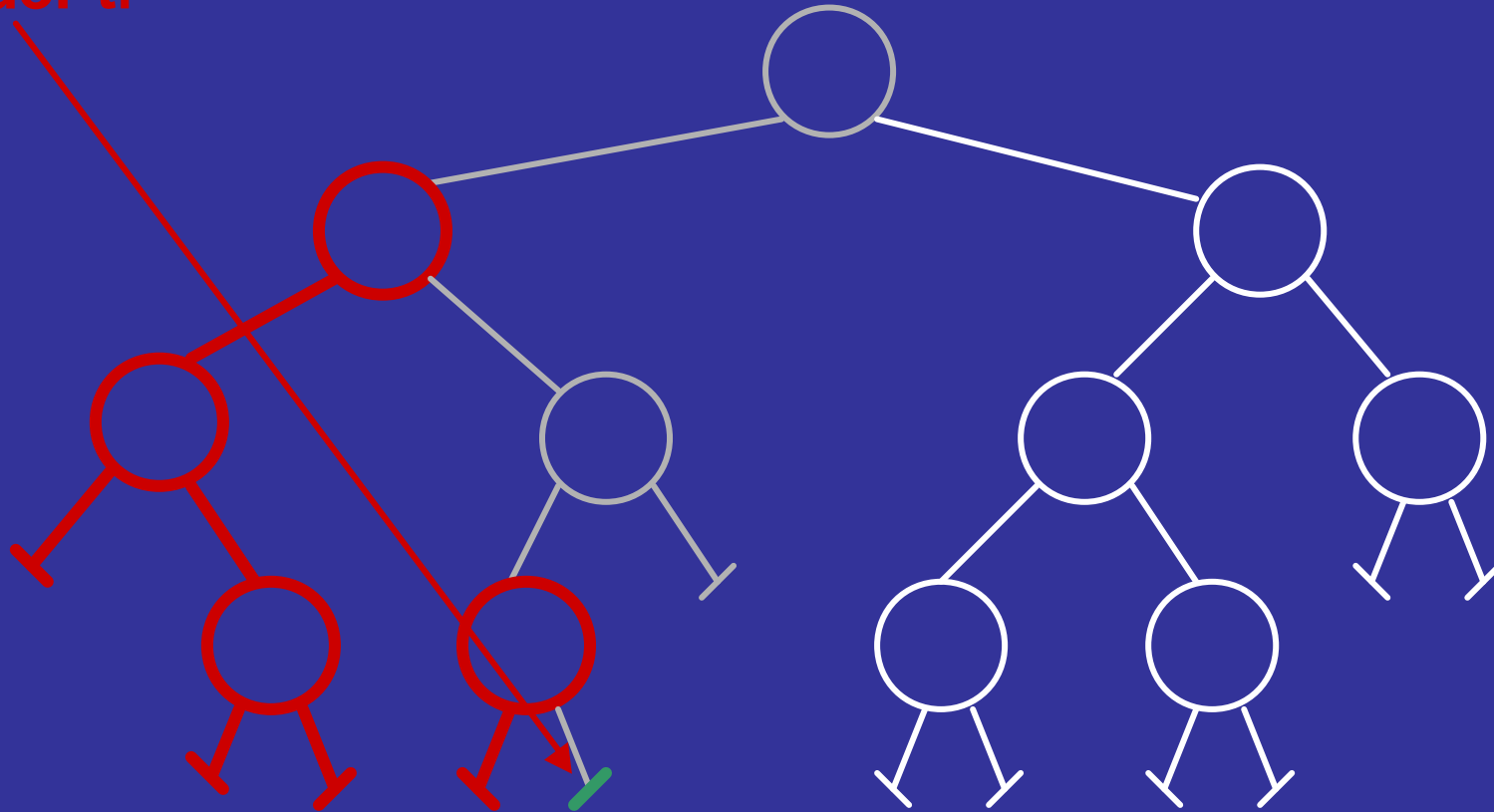
2/7/02

CSE1303 Part A

32

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

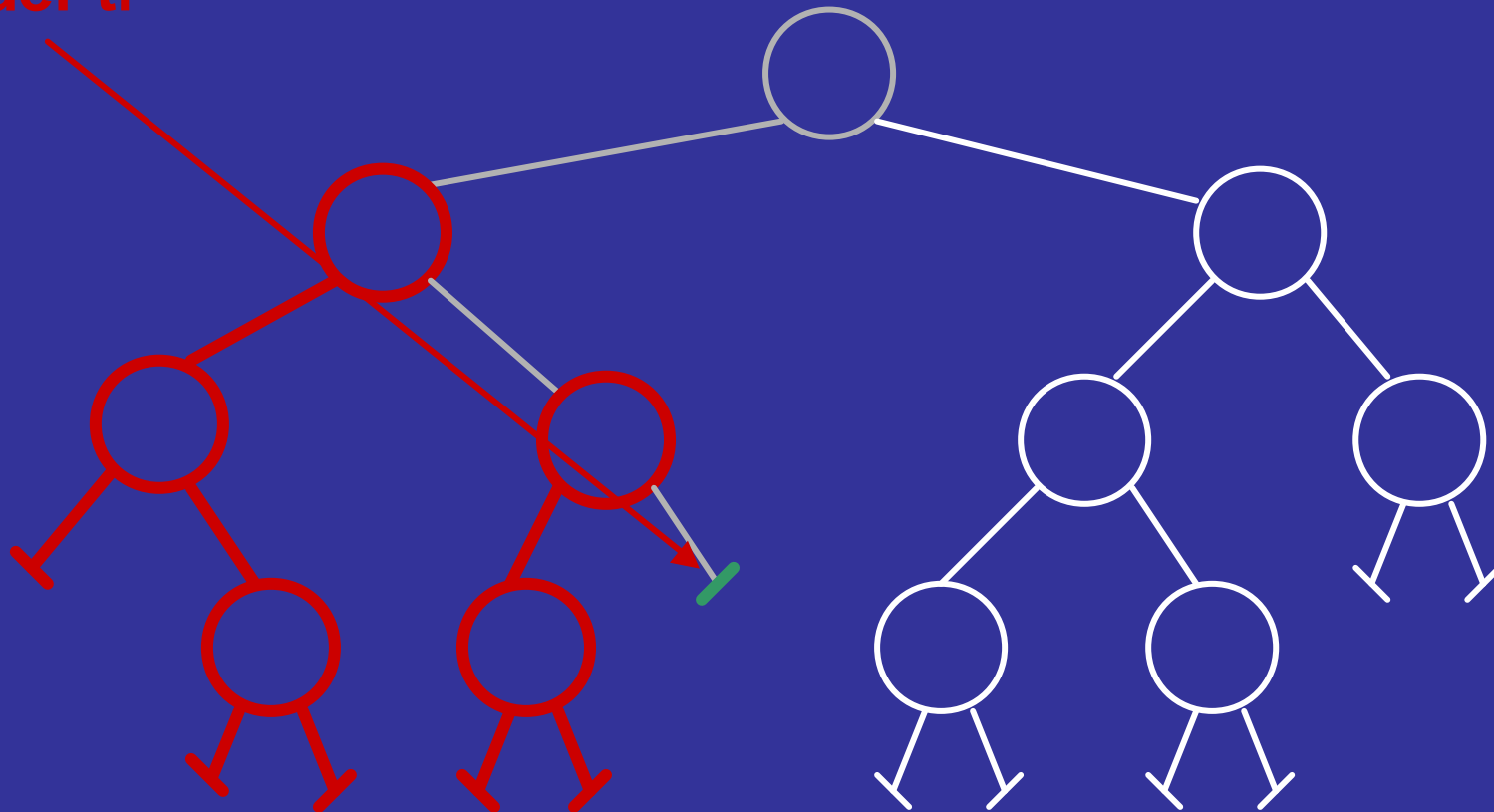
2/7/02

CSE1303 Part A

34

Inorder

nodePtr



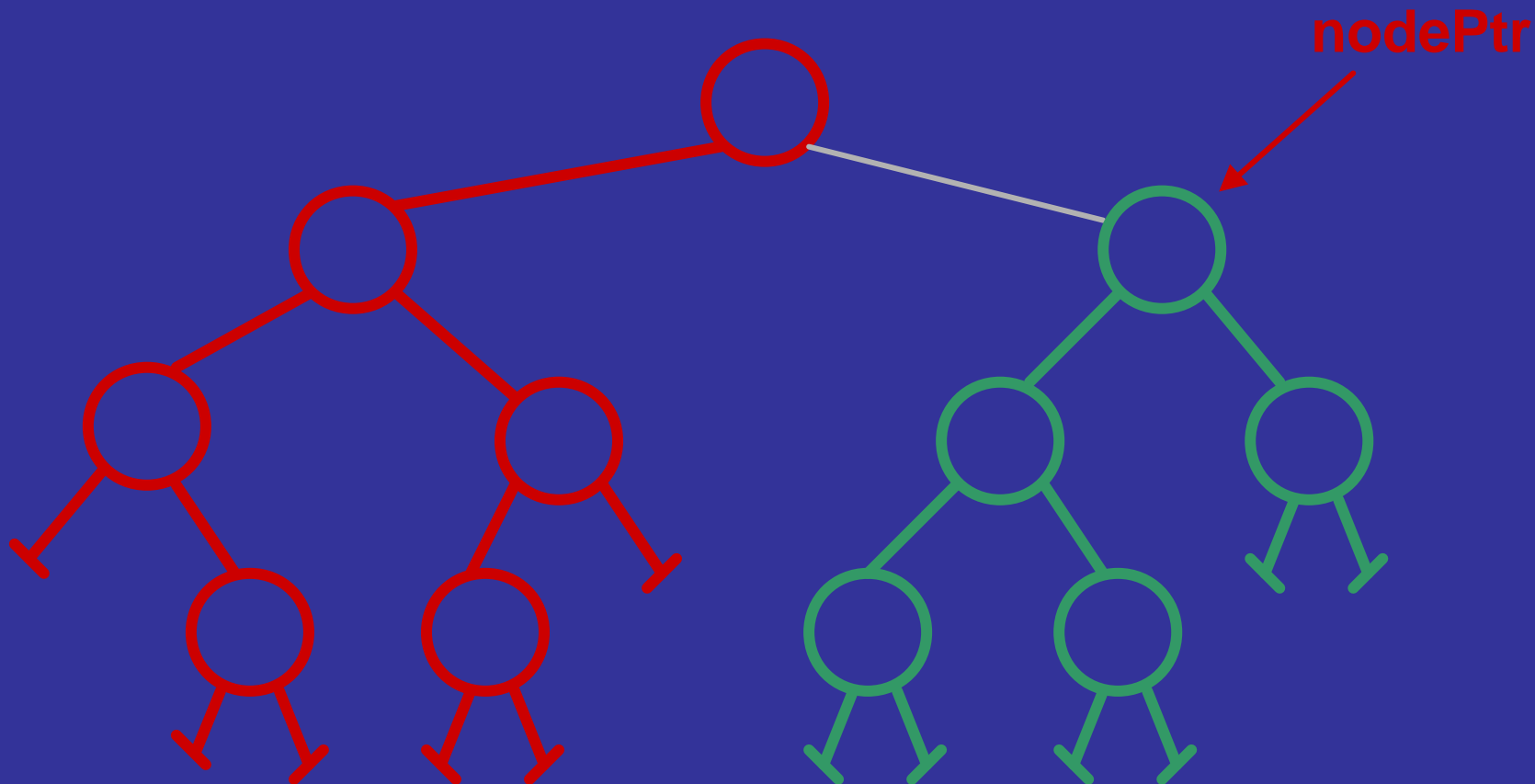
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

2/7/02

CSE1303 Part A

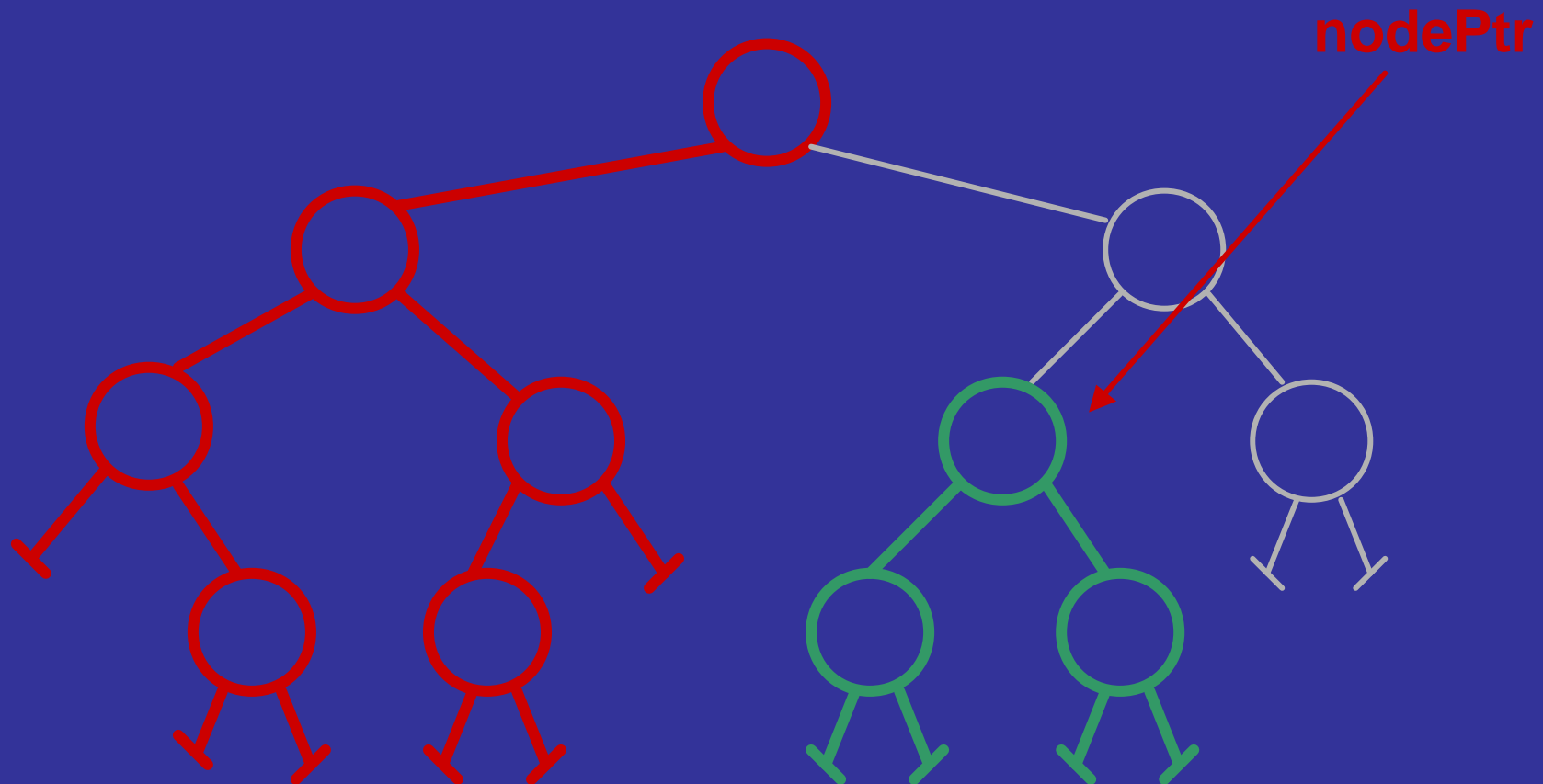
35

Inorder



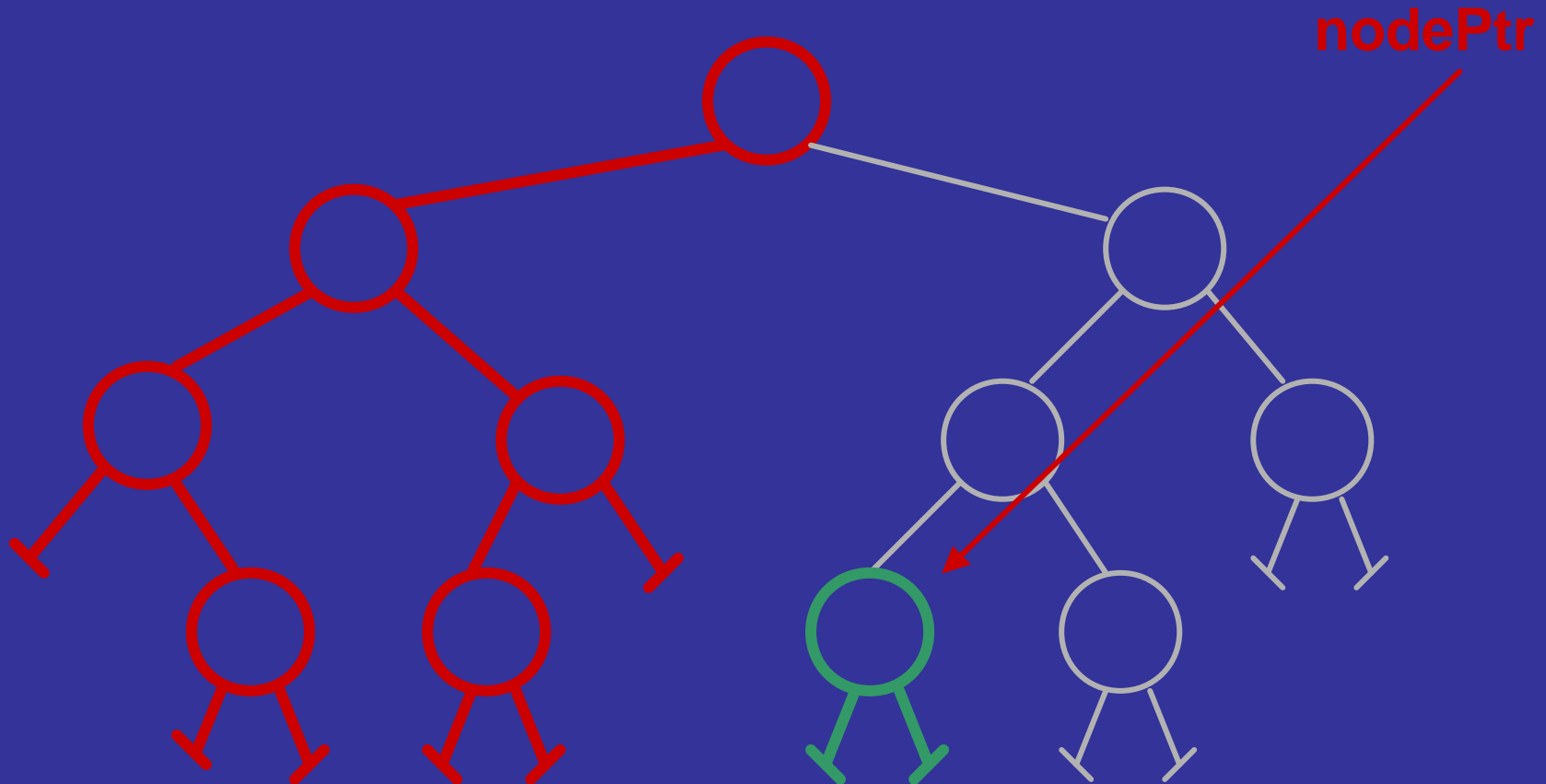
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



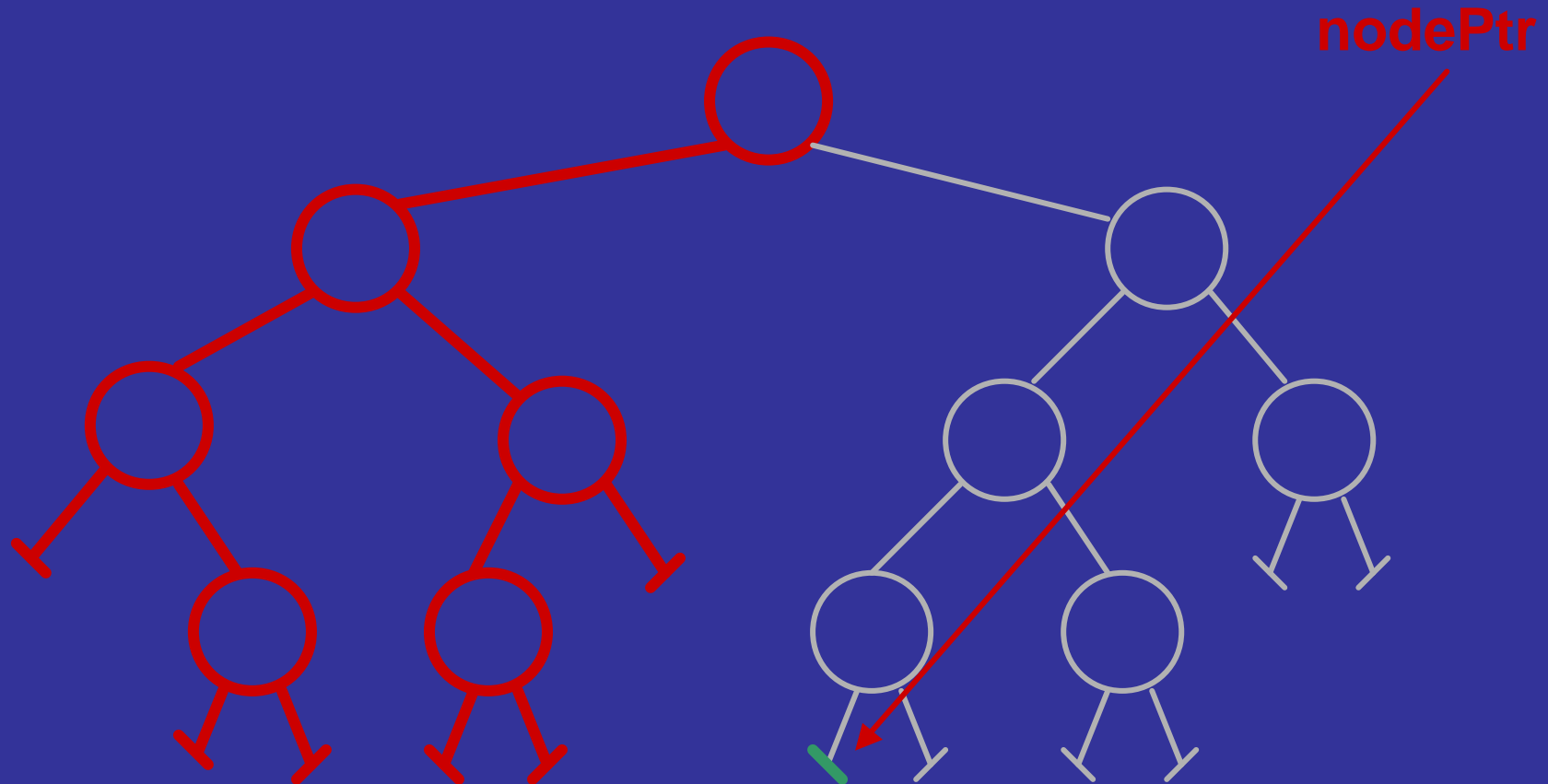
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



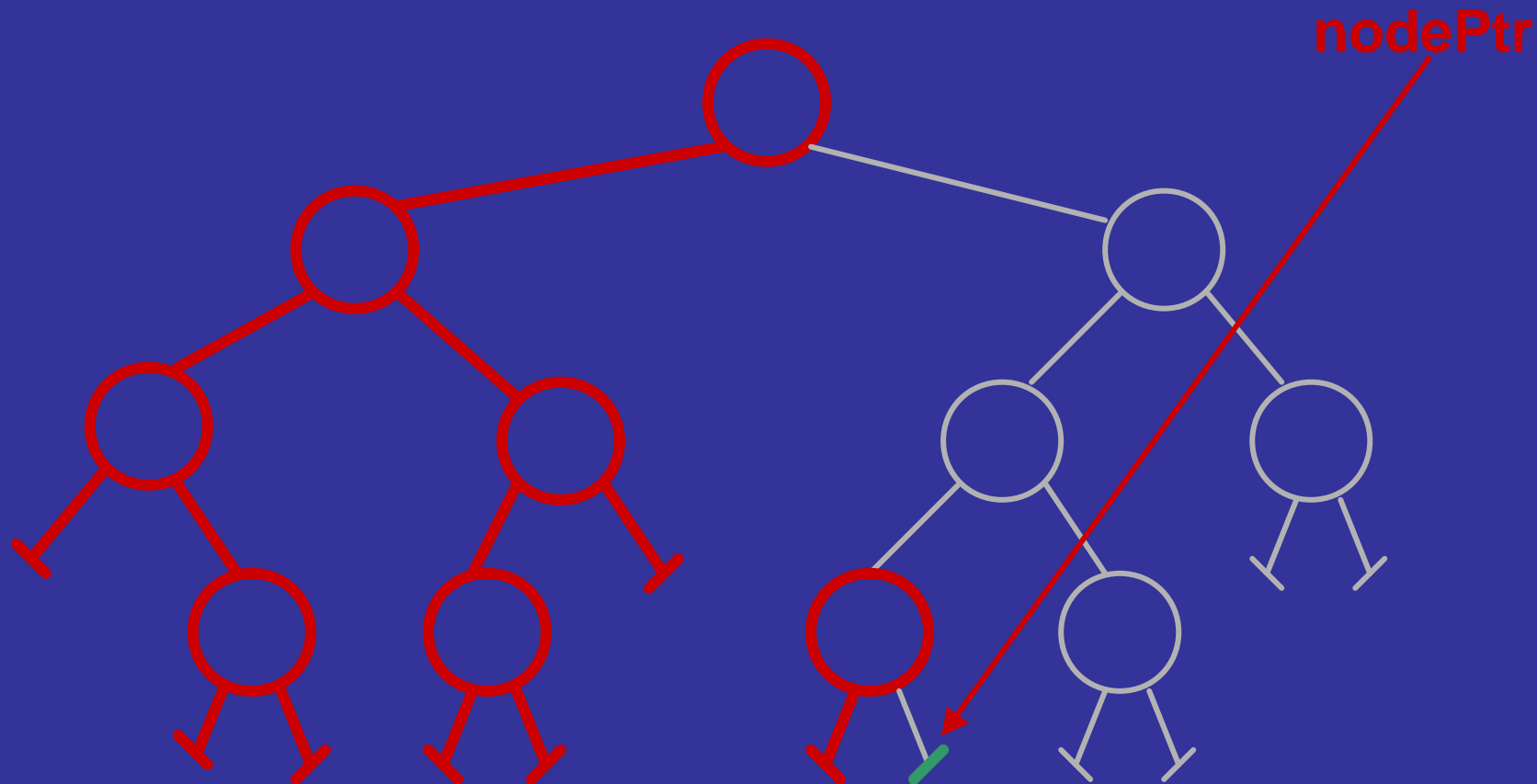
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



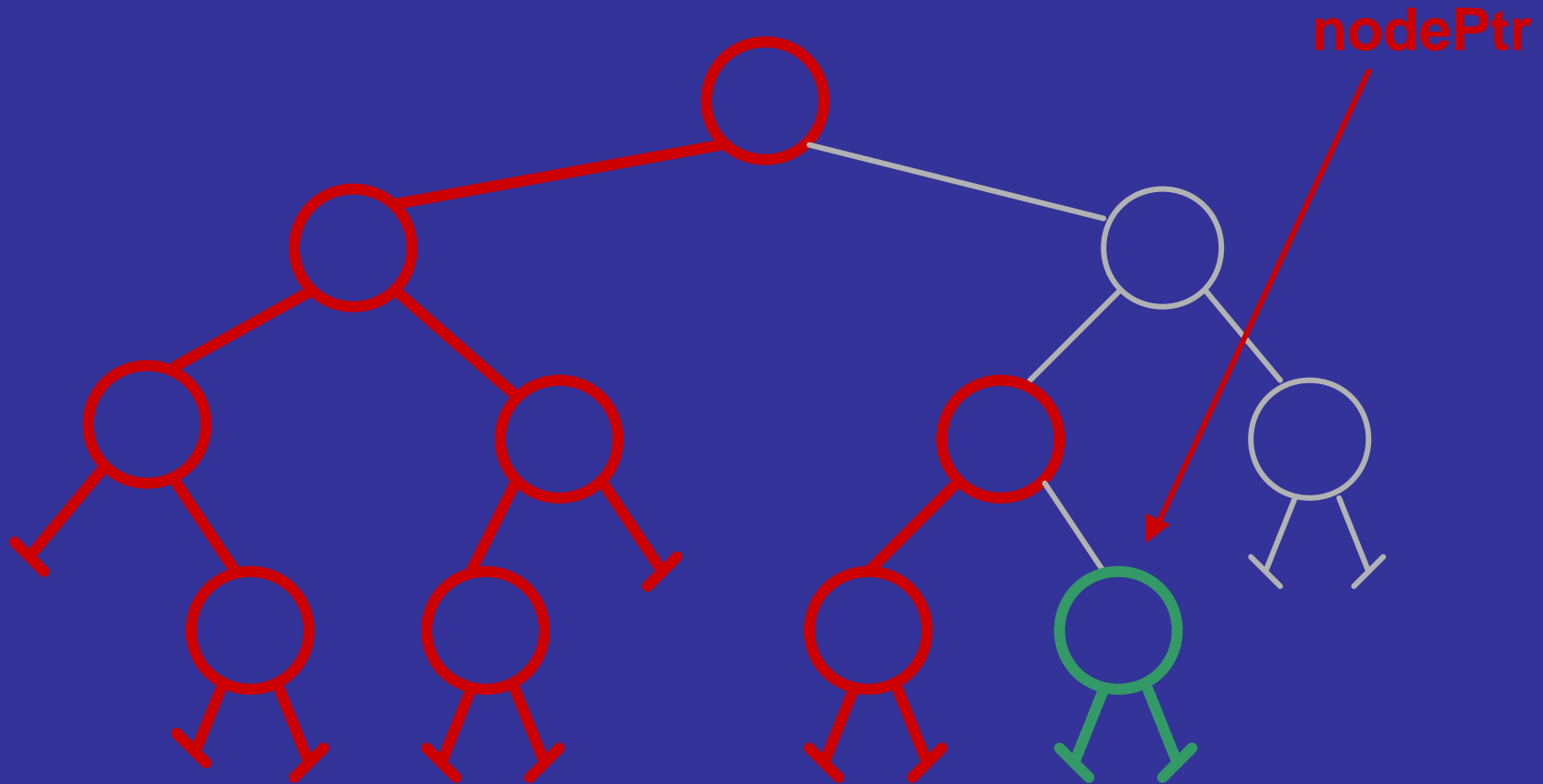
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



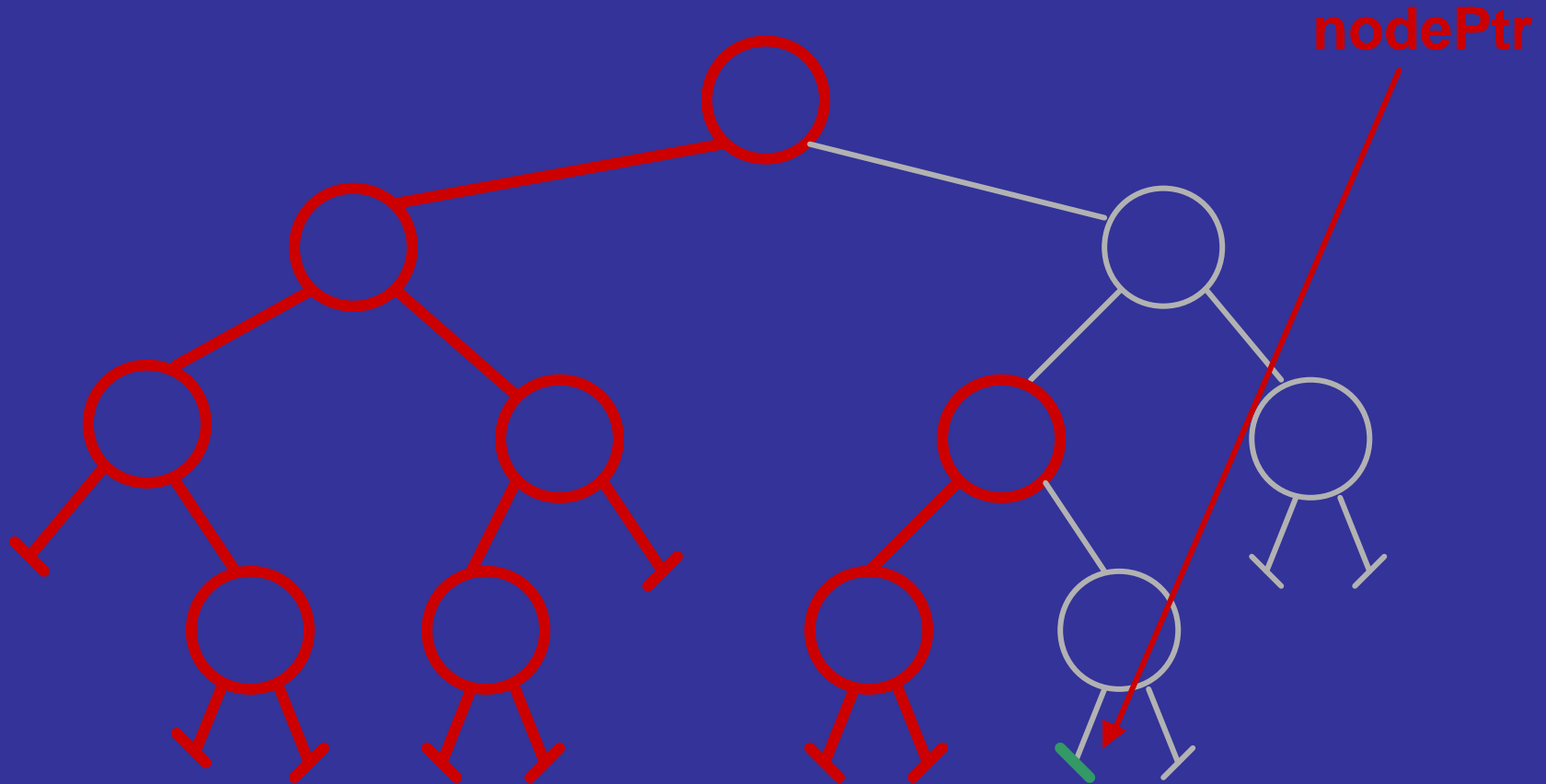
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

2/7/02

CSE1303 Part A

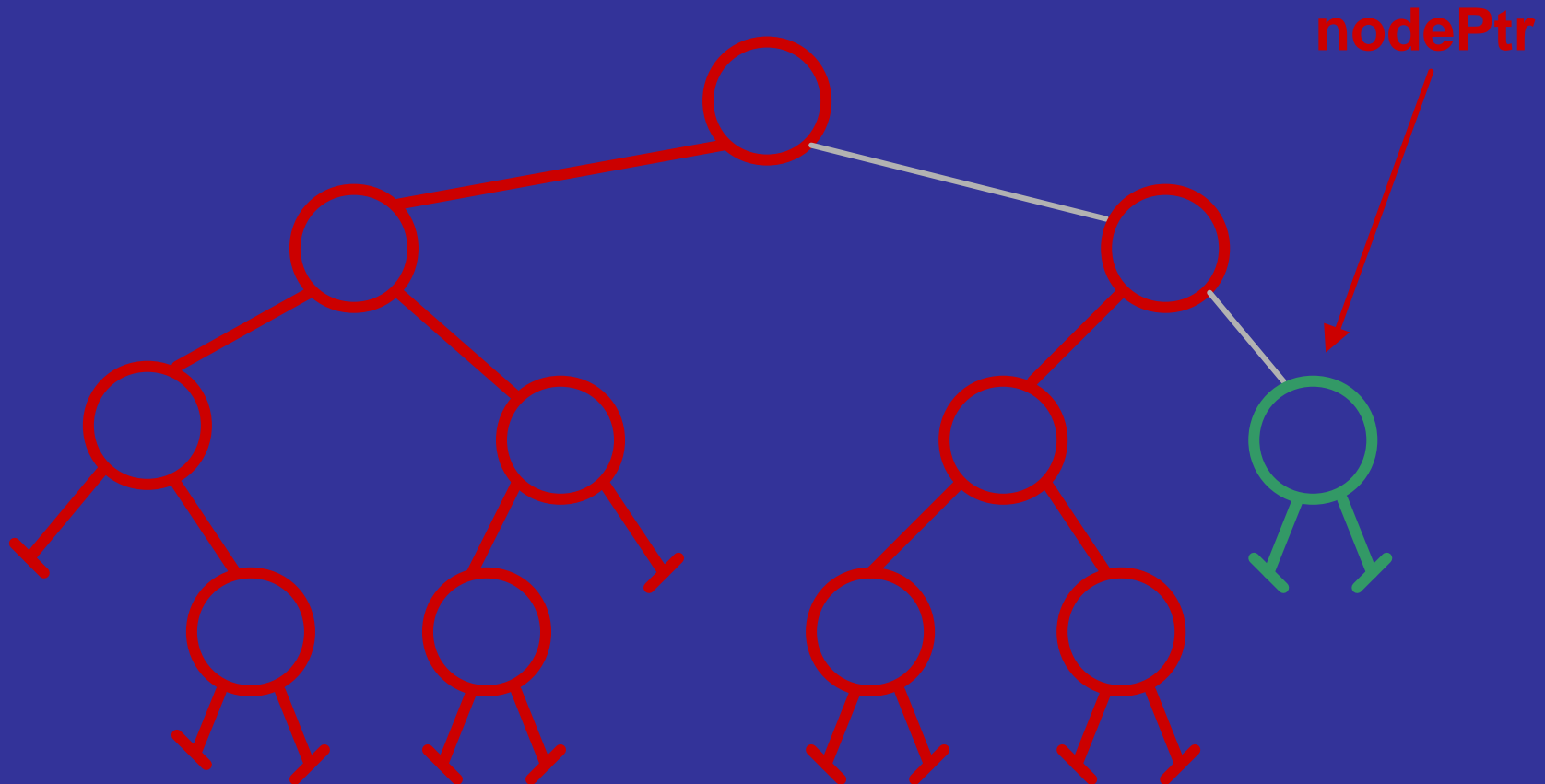
41

Inorder



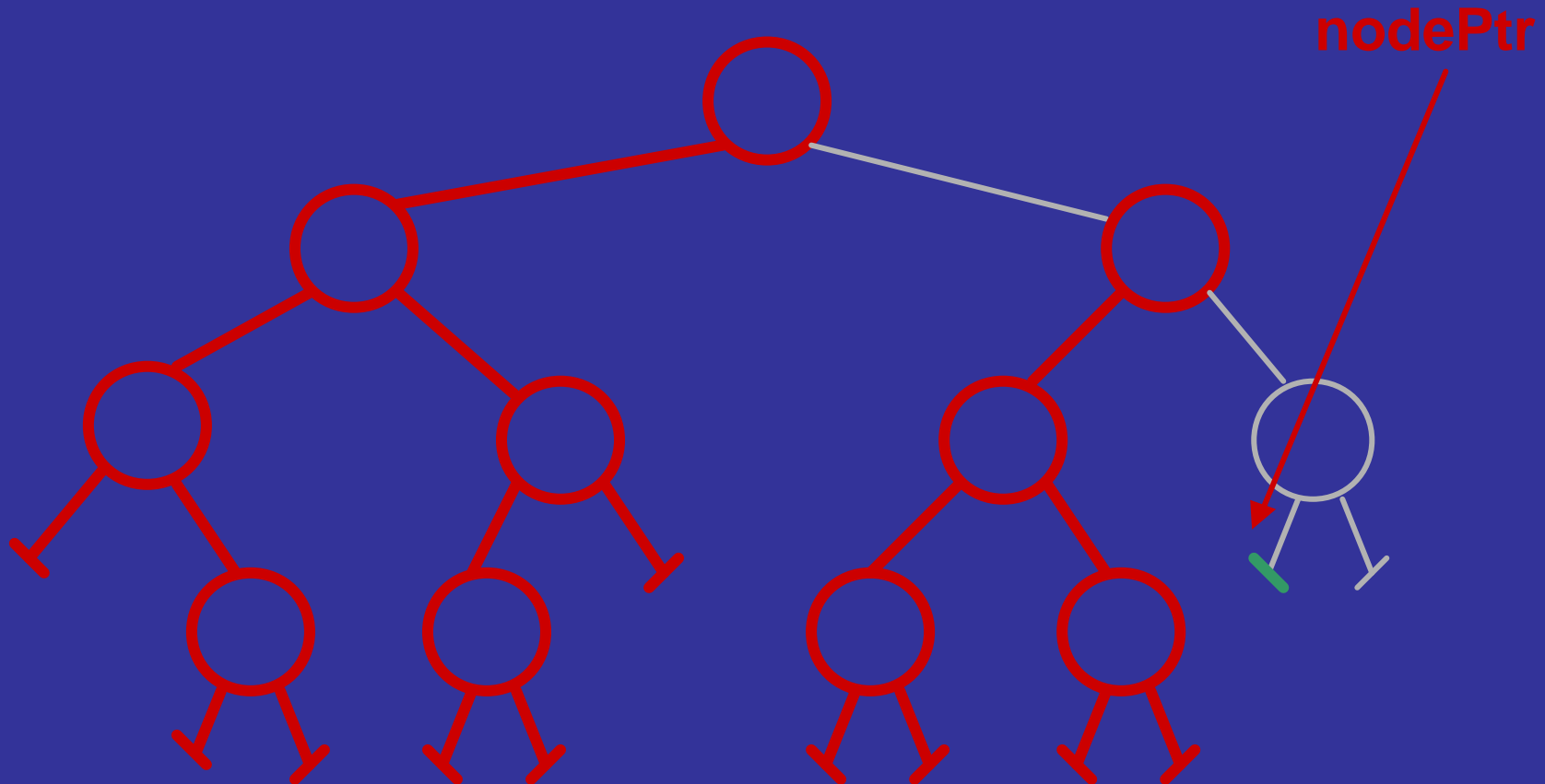
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```


Inorder



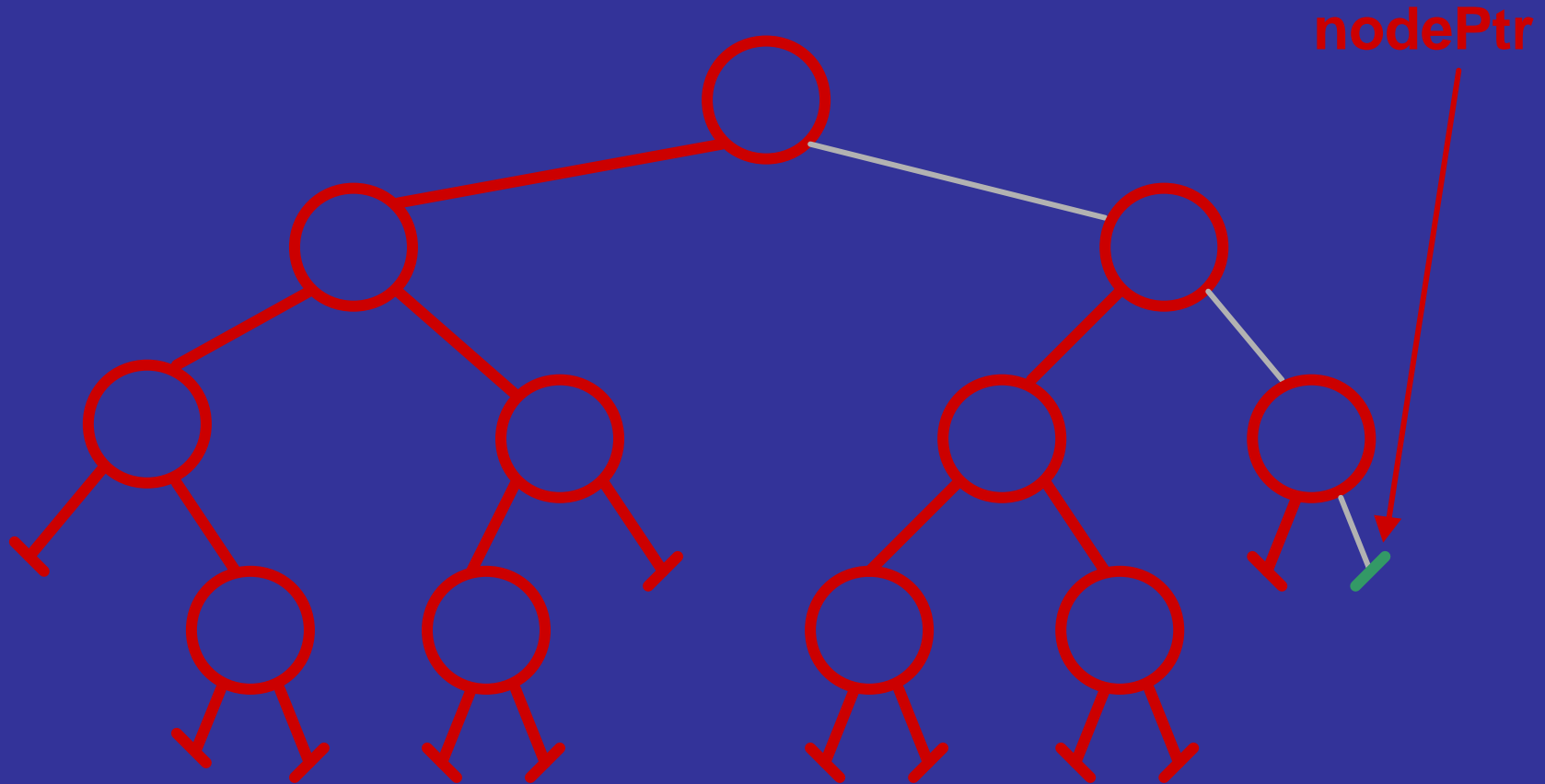
```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

Inorder



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

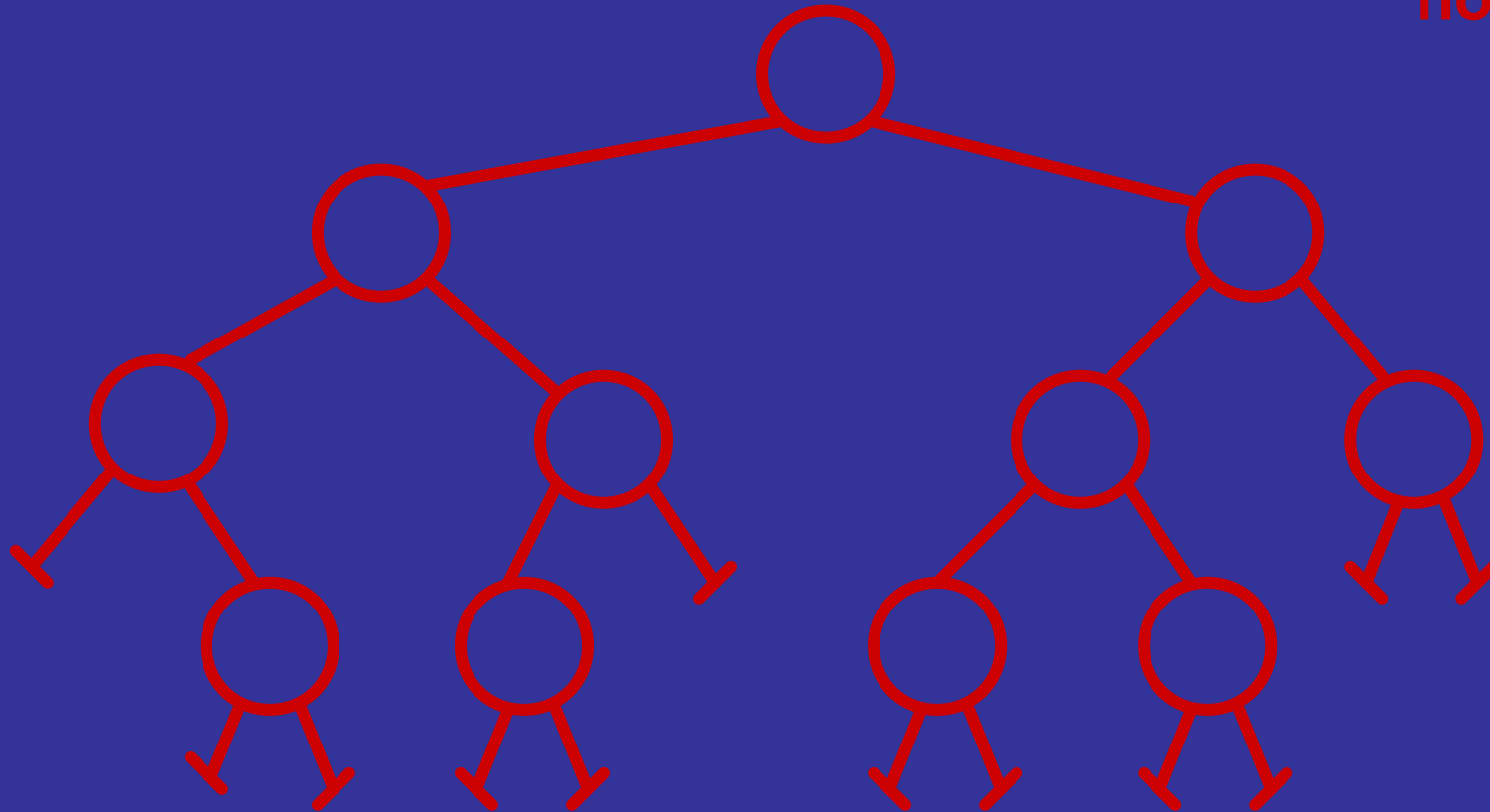
2/7/02

CSE1303 Part A

46

Inorder

nodePtr



```
void printInorder(TreeNode* nodePtr){  
    if (nodePtr != NULL){  
        printInorder(nodePtr->leftPtr);  
        printf(" %f", nodePtr->key);  
        printInorder(nodePtr->rightPtr);  
    }  
}
```

2/7/02

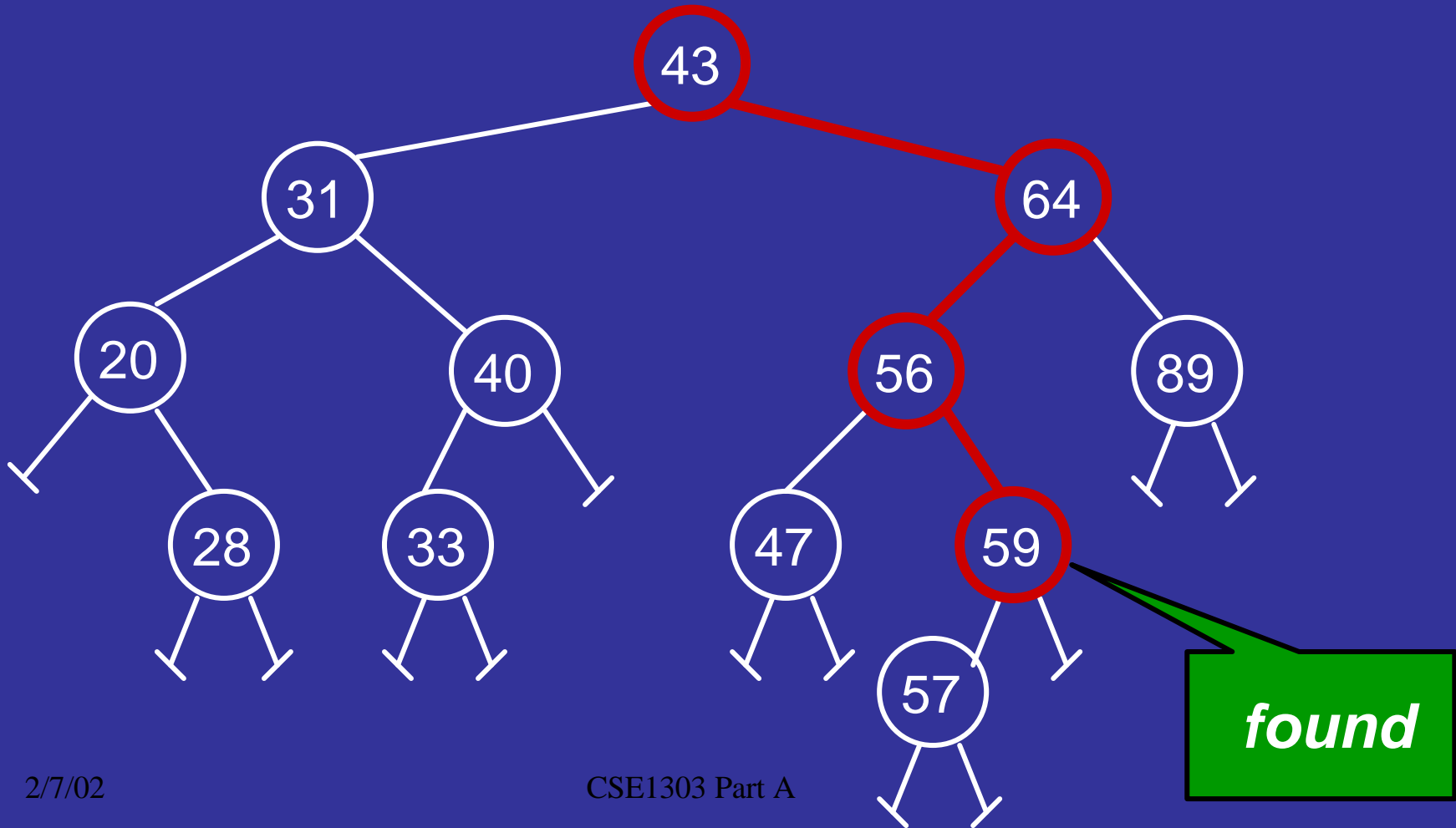
CSE1303 Part A

47

Search

Example:

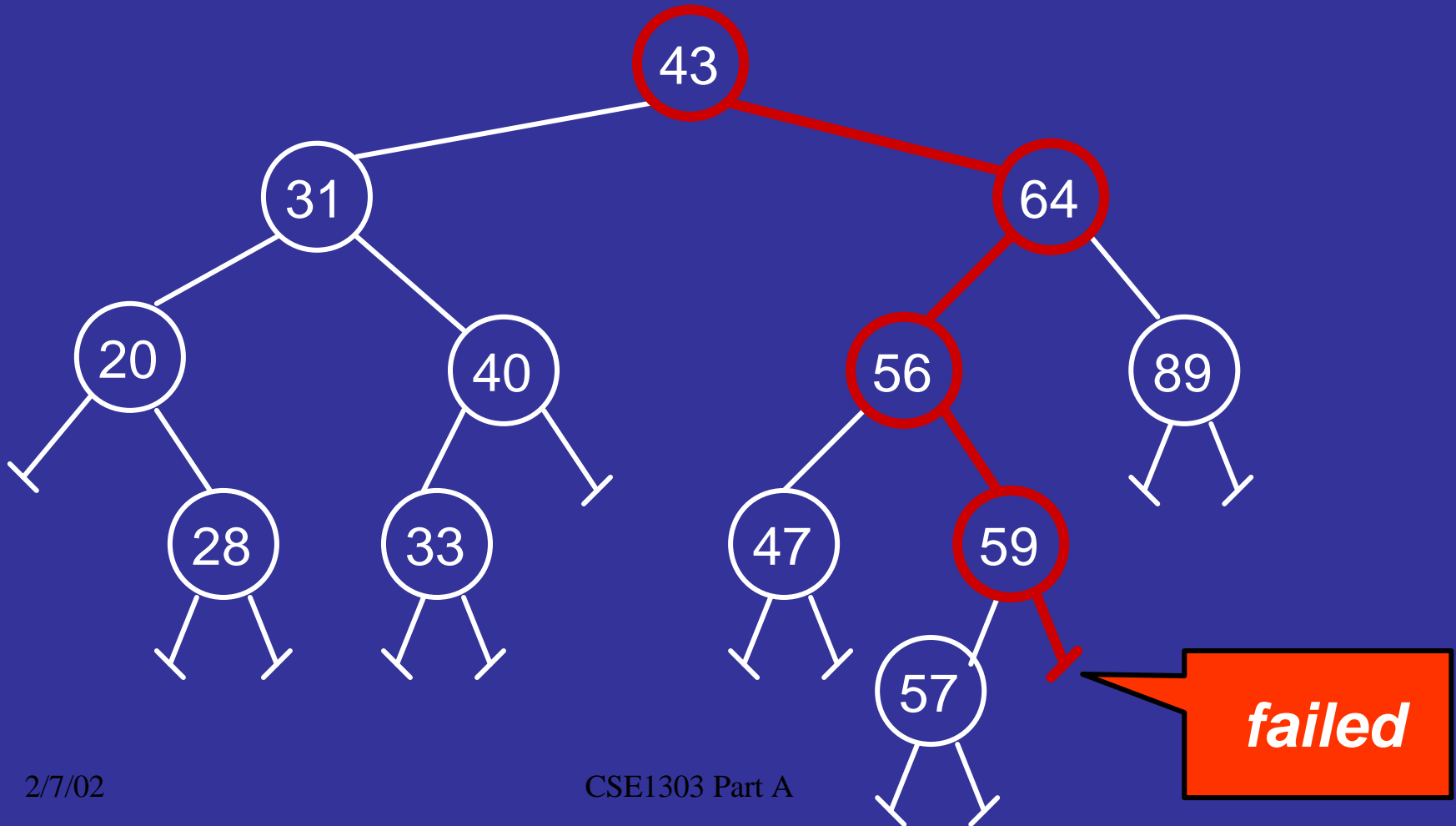
59



Search

Example:

61



Search: Checklist

- if **target key** is **less** than current node's key, search the **left sub-tree**.
- else, if **target key** is **greater** than current node's key, search the **right sub-tree**.
- returns:
 - if found, or if **target key** is **equal** to current node's key, a **pointer** to node containing target key.
 - otherwise, **NULL** pointer.

```
TreeNode*
search(TreeNode* nodePtr, float target)
{
    if (nodePtr != NULL)
    {
        if (target < nodePtr->key)
        {
            nodePtr = search(nodePtr->leftPtr, target);
        }
        else if (target > nodePtr->key)
        {
            nodePtr = search(nodePtr->rightPtr, target);
        }
    }
    return nodePtr;
}
```

Function Call to Search

```
/* ...other bits of code omitted.. */

printf("Enter target ");
scanf("%f", &item);

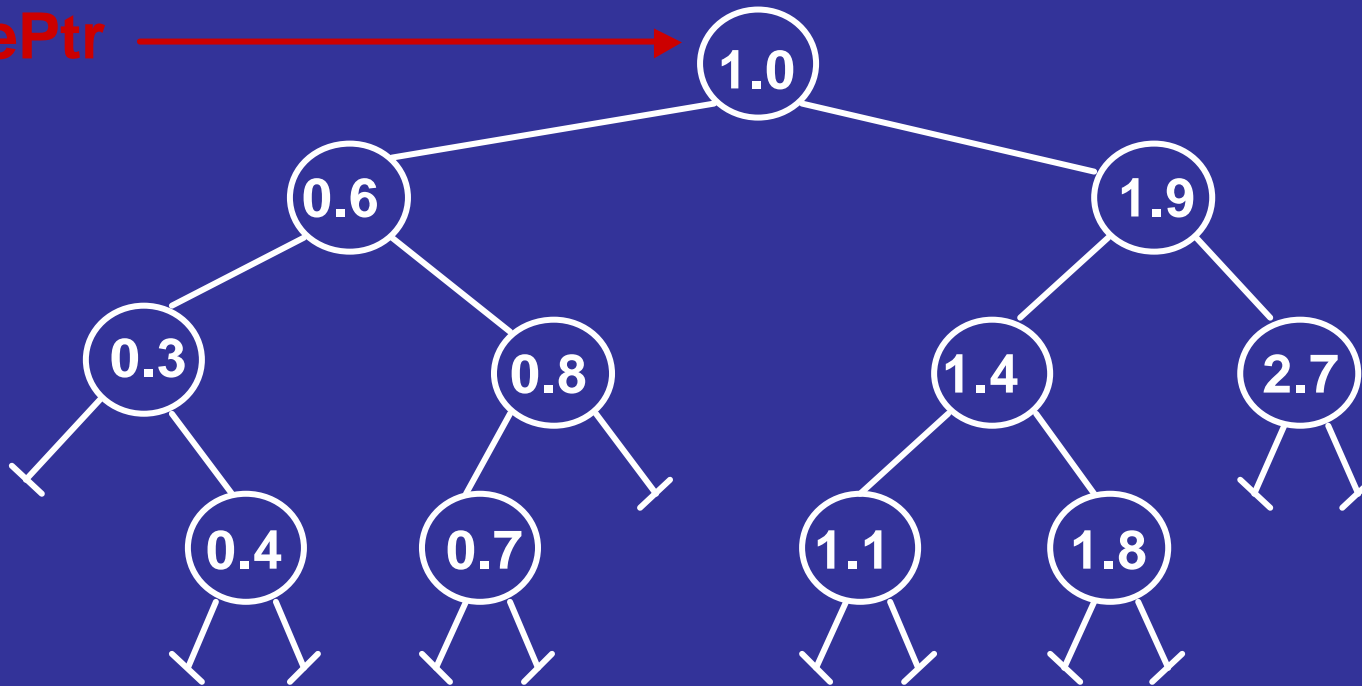
if (search(rootPtr, item) == NULL)
{
    printf("Item was not found\n");
}
else
{
    printf("Item found\n");
}

/* ...and so on.. */
```

Search

Find 0.7

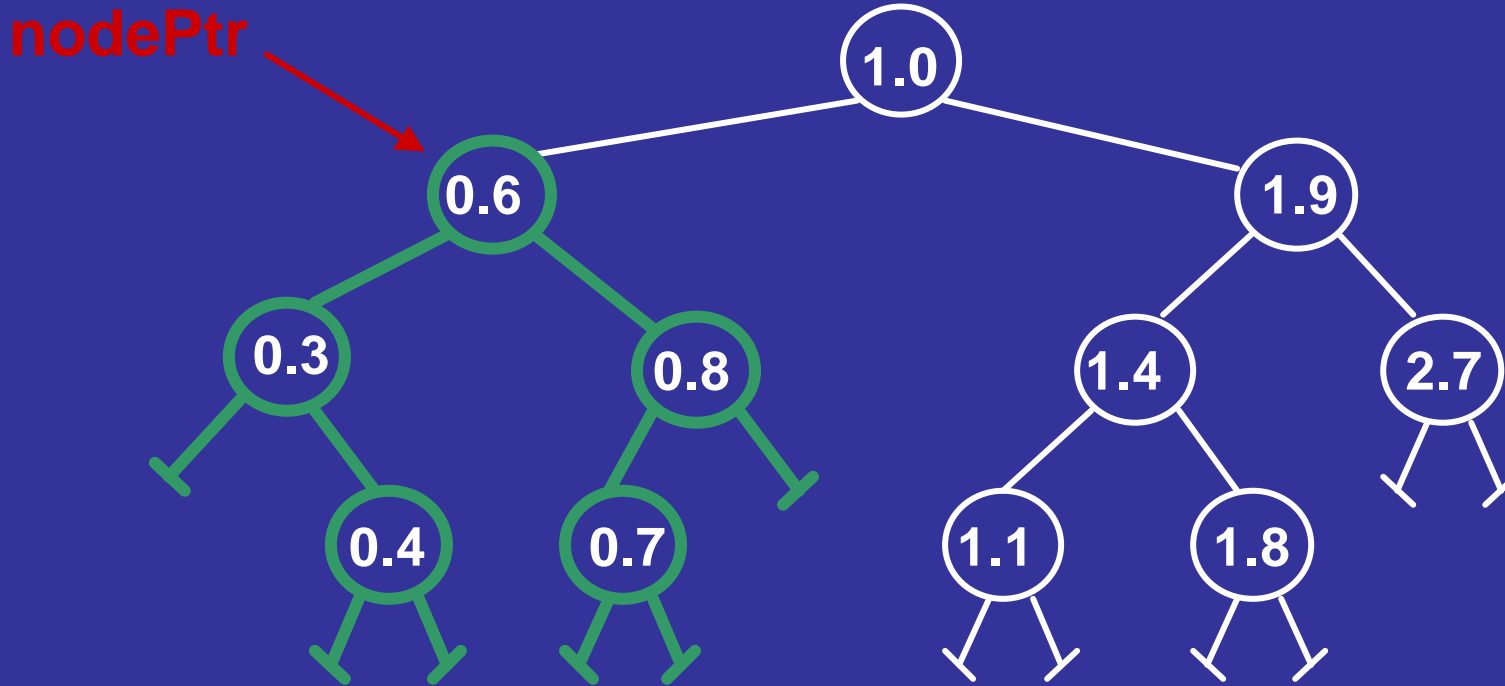
nodePtr



```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.7

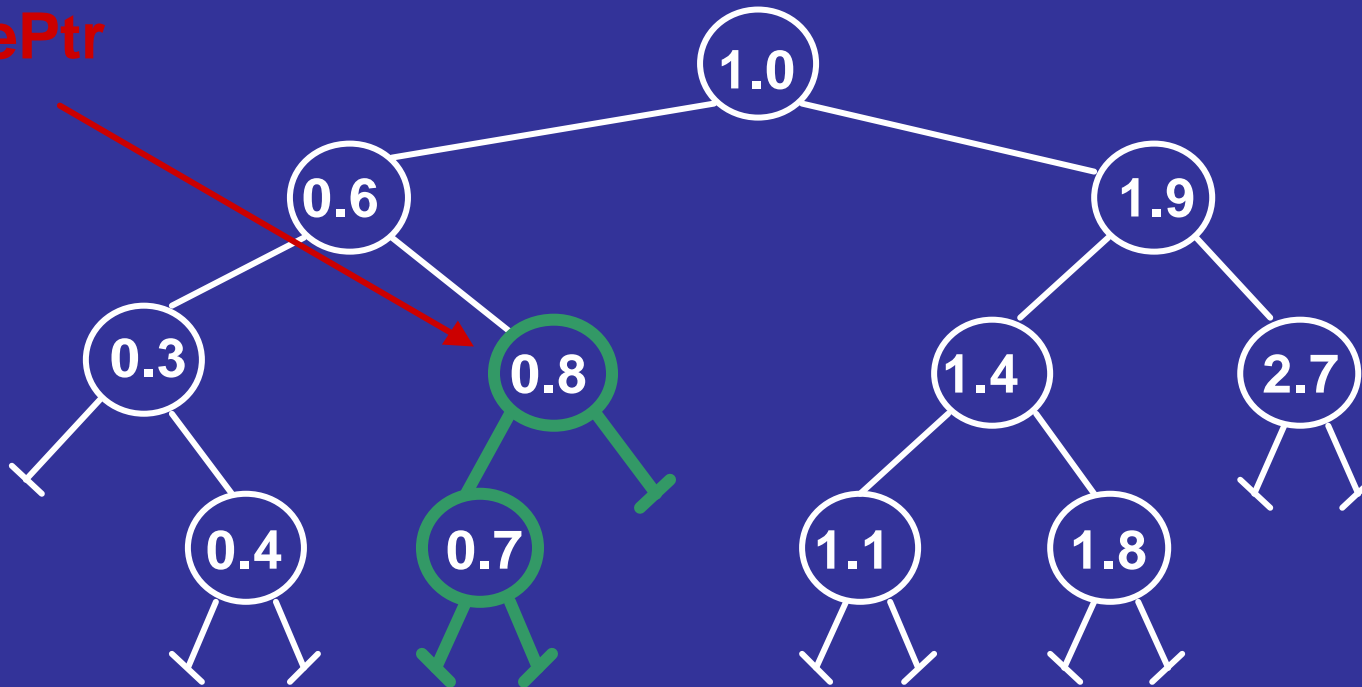


```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.7

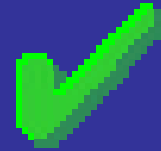
nodePtr



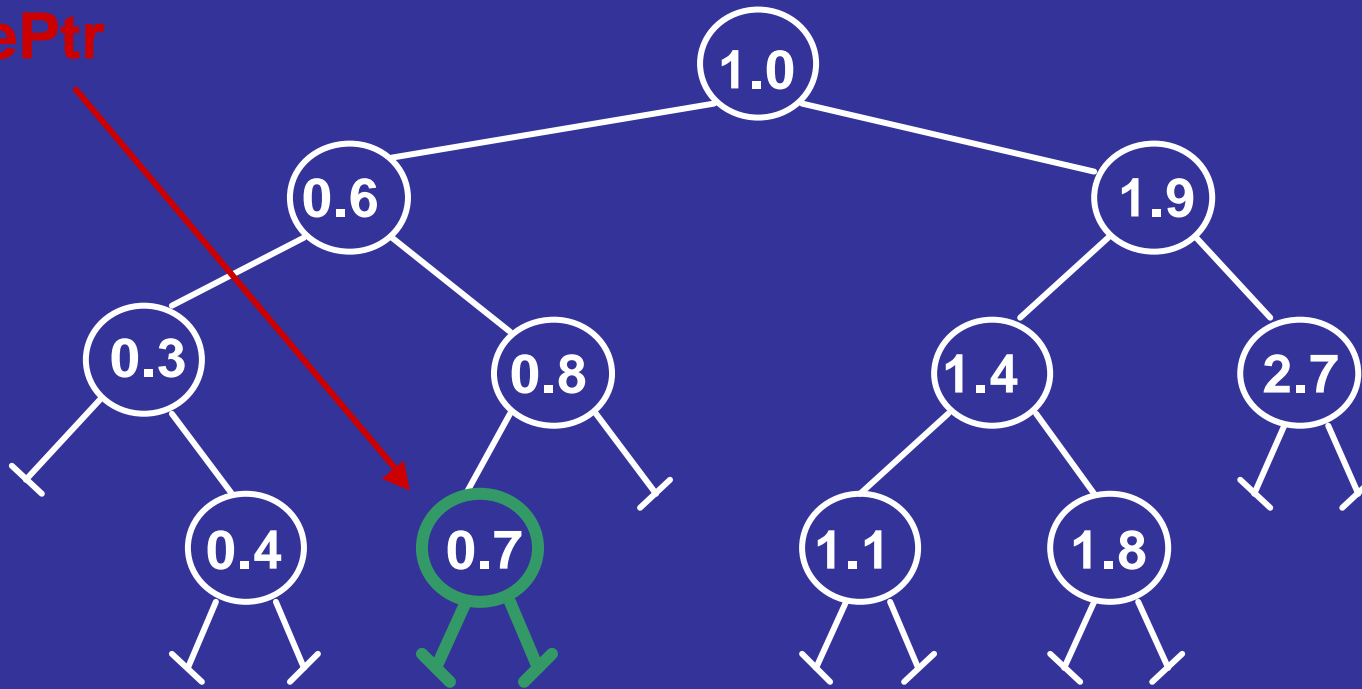
```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.7



nodePtr

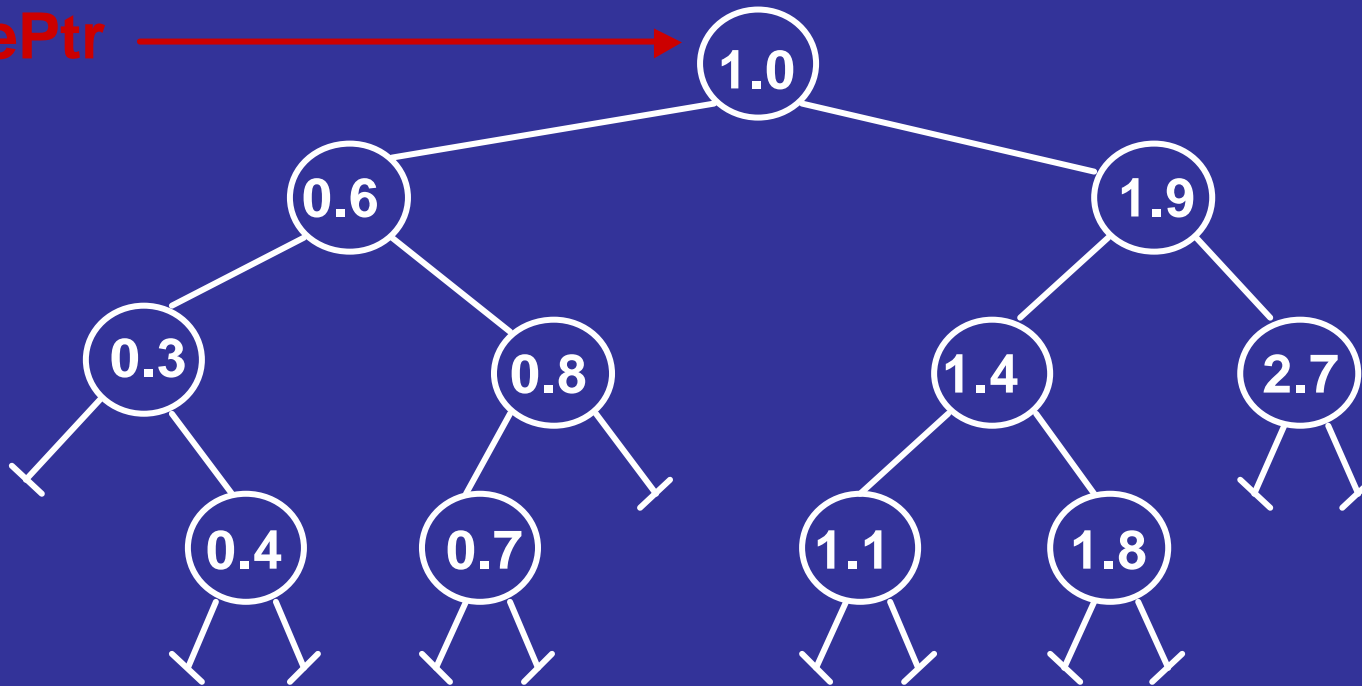


```
TreeNode* search(TreeNode* nodePtr, float target){  
    if (nodePtr != NULL){  
        if (target < nodePtr->key)  
            nodePtr = search(nodePtr->leftPtr, target);  
        else if (target > nodePtr->key)  
            nodePtr = search(nodePtr->rightPtr, target);  
    }  
    return nodePtr;  
}
```

Search

Find 0.5

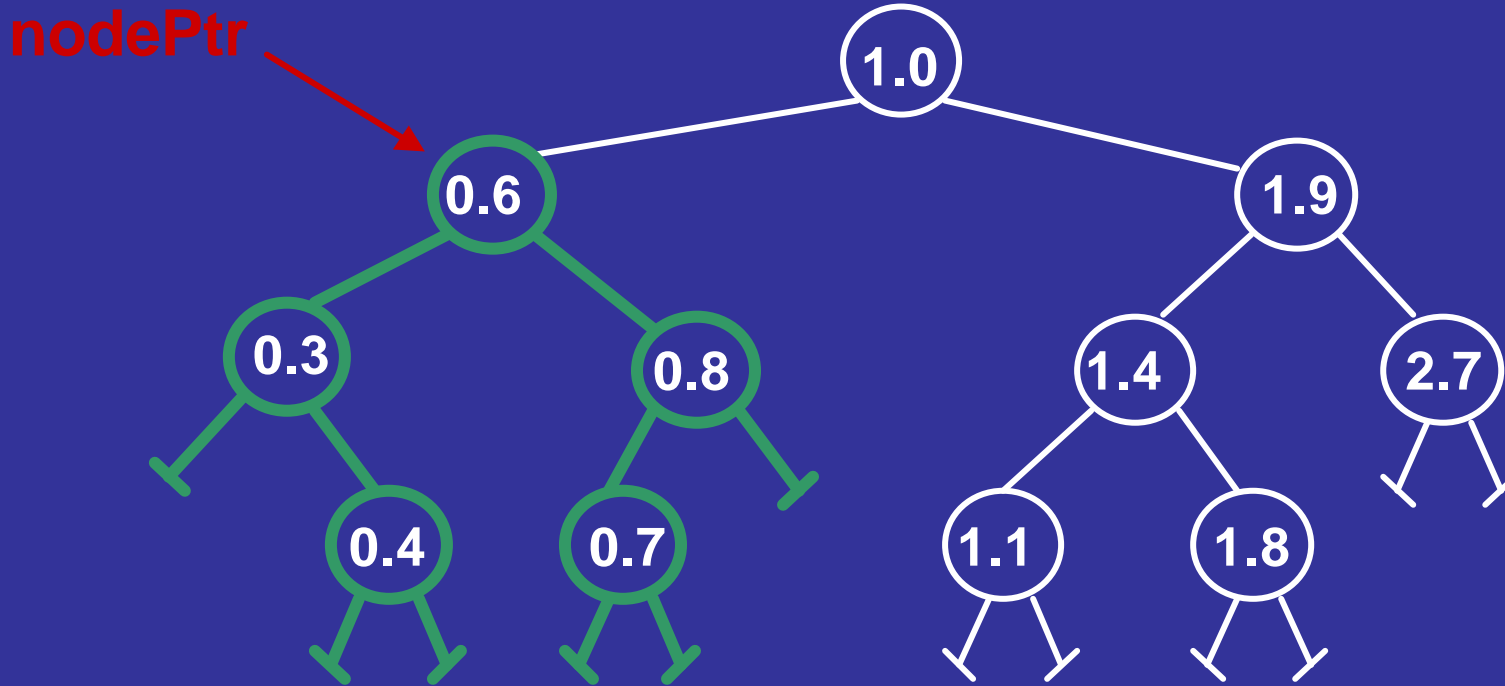
nodePtr



```
TreeNode* search(TreeNode* nodePtr, float target){  
    if (nodePtr != NULL){  
        if (target < nodePtr->key)  
            nodePtr = search(nodePtr->leftPtr, target);  
        else if (target > nodePtr->key)  
            nodePtr = search(nodePtr->rightPtr, target);  
    }  
    return nodePtr;  
}
```

Search

Find 0.5

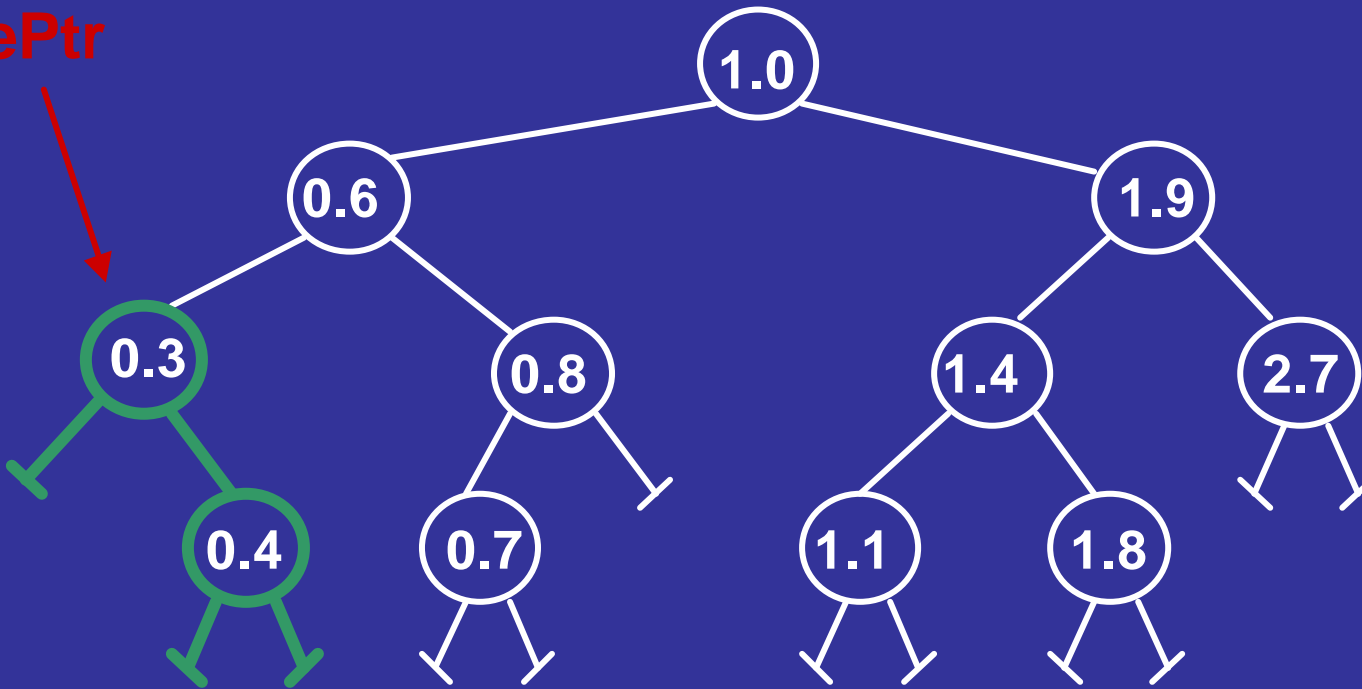


```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.5

nodePtr

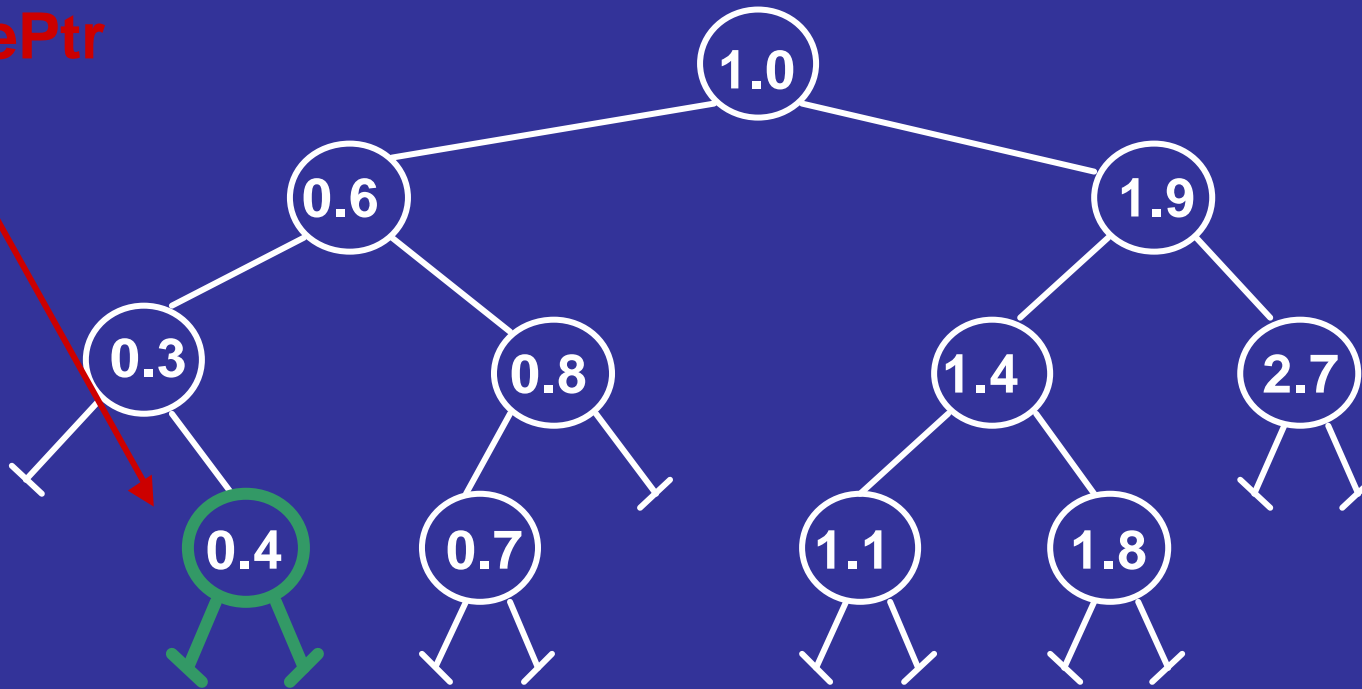


```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.5

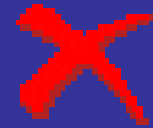
nodePtr



```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}
```

Search

Find 0.5



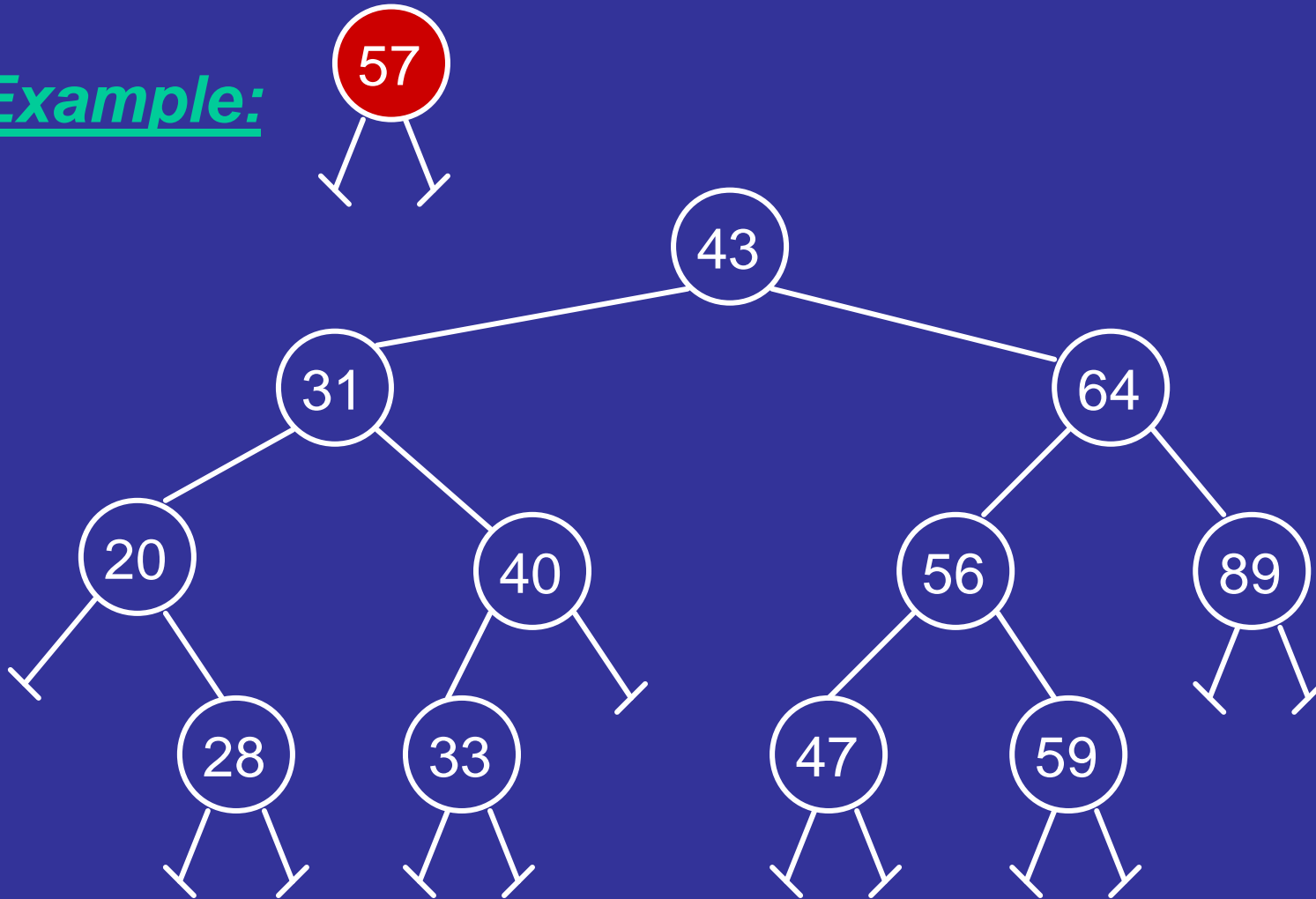
nodePtr



```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
```

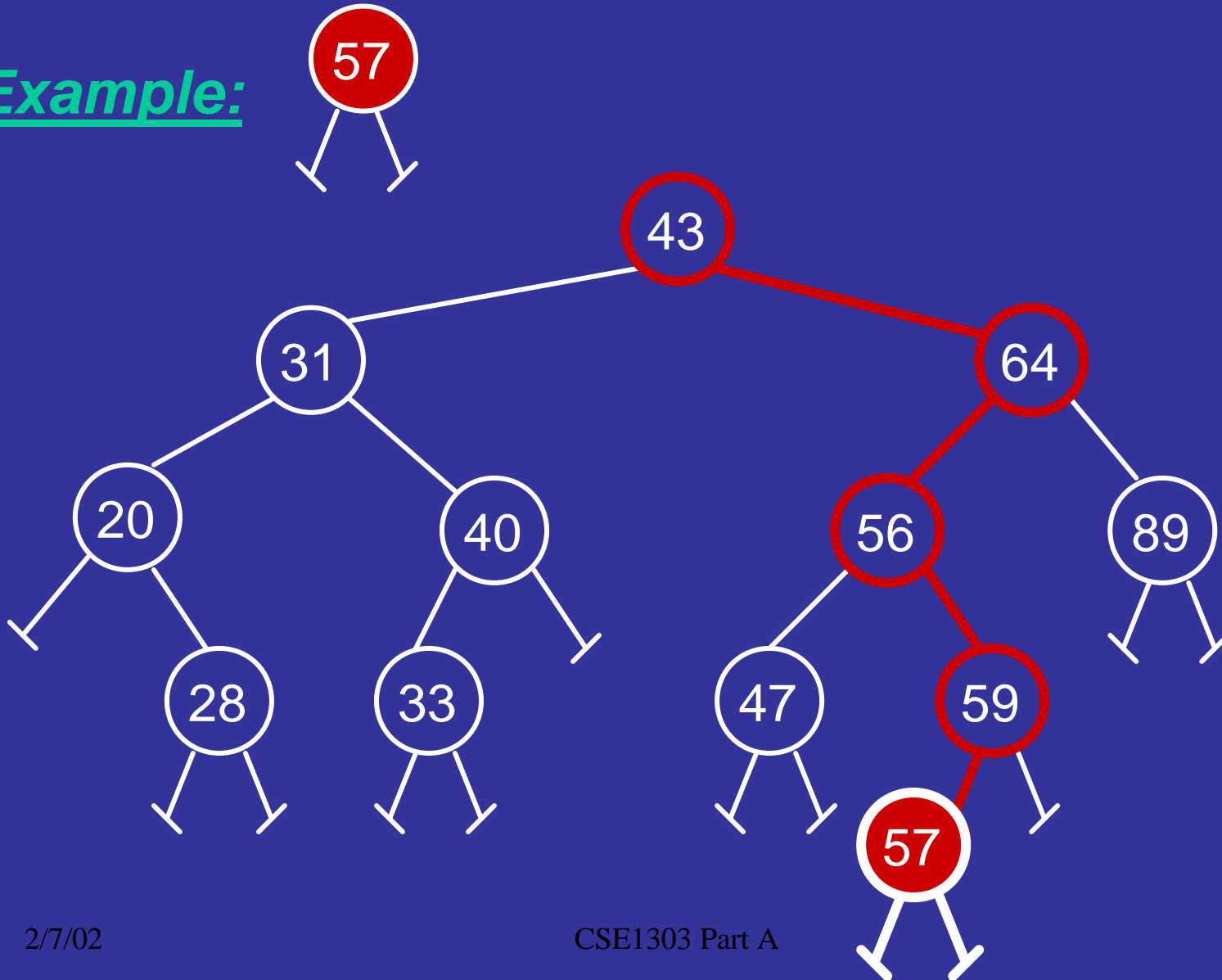
Insert

Example:



Insert

Example:



Insert

- Create **new node** for the item.
- Find a **parent node**.
- Attach new node as a **leaf**.

Insert: Recursive

- parameters:
 - pointer to **current node** (initially: root node).
 - **item** to be inserted.
- If current node is **NULL**
 - Create a new node and return it.
- Else if item's key is **less** (**greater**) than current node's key:
 - otherwise, let the **left** (**right**) child node be the current node, setting the parent **left** (**right**) link equal that node, and repeat **recursively**.

```
TreeNode*
insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
    {
        nodePtr = makeTreeNode(item);
    }
    else if (item < nodePtr->key)
    {
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    }
    else if (item > nodePtr->key)
    {
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    }

    return nodePtr;
}
```

Function Call to Insert

```
/* ...other bits of code omitted.. */

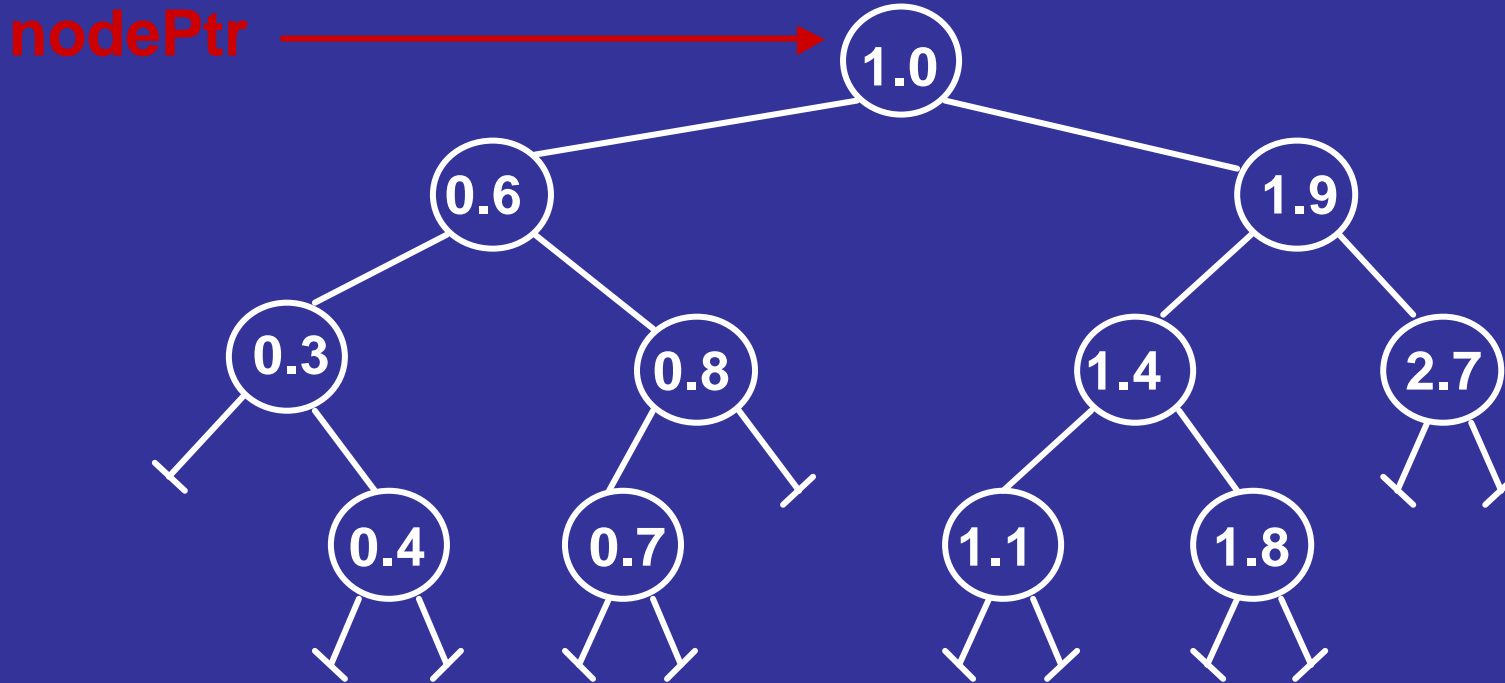
printf("Enter number of items ");
scanf("%d", &n);

for (i = 0; i < n; i++) {
    scanf("%f", &item);
    rootPtr = insert(rootPtr, item);
}

/* ...and so on.. */
```

Insert

Insert 0.9

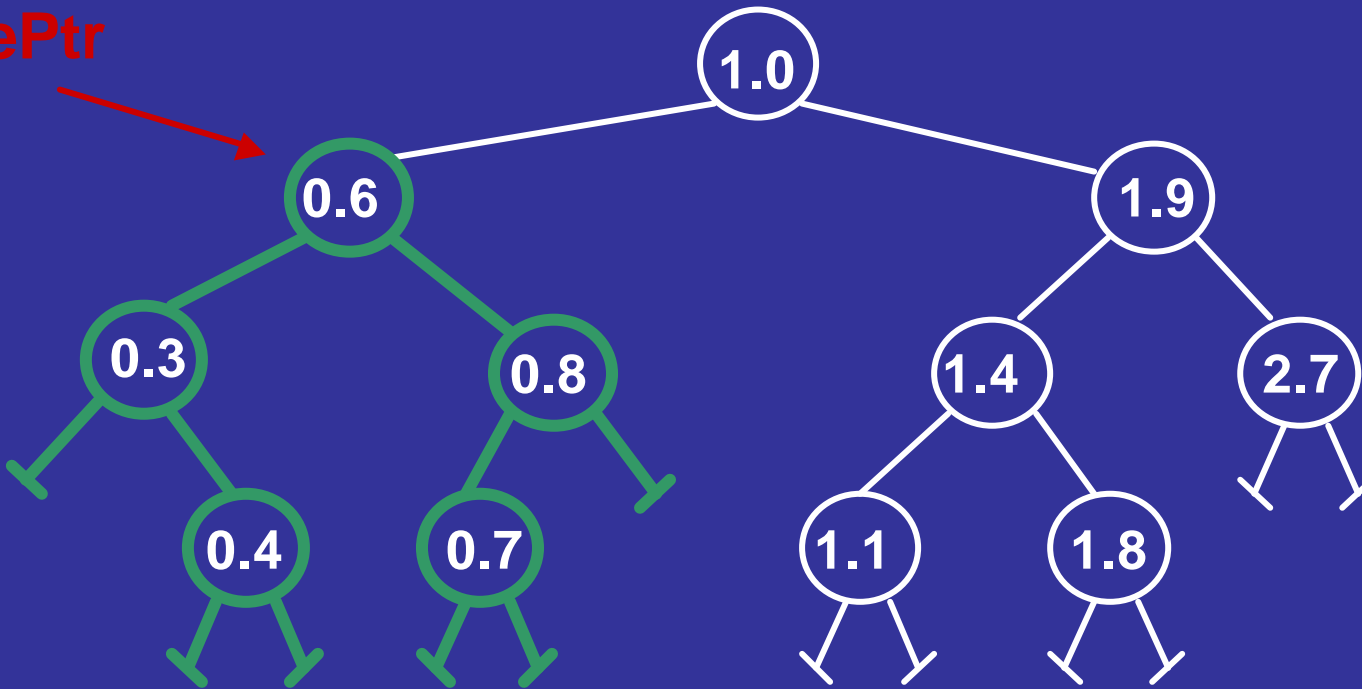


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
```

Insert

Insert 0.9

nodePtr

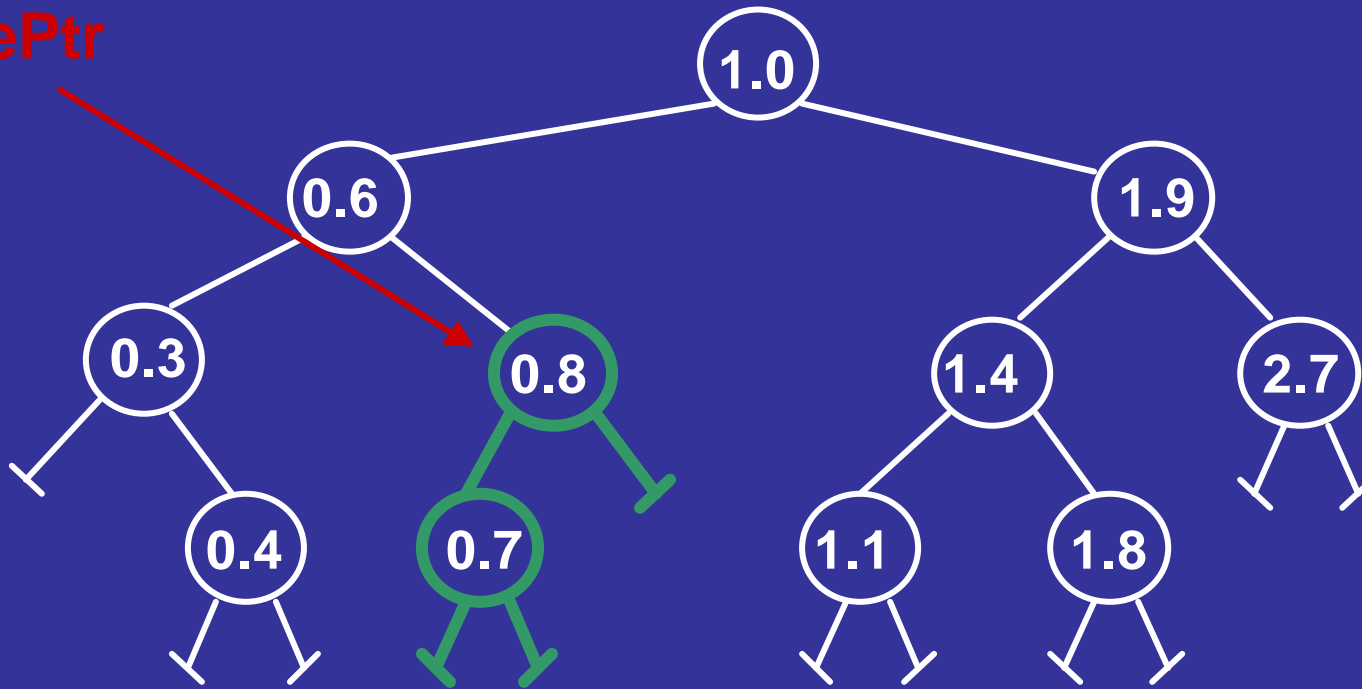


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
```

Insert

Insert 0.9

nodePtr

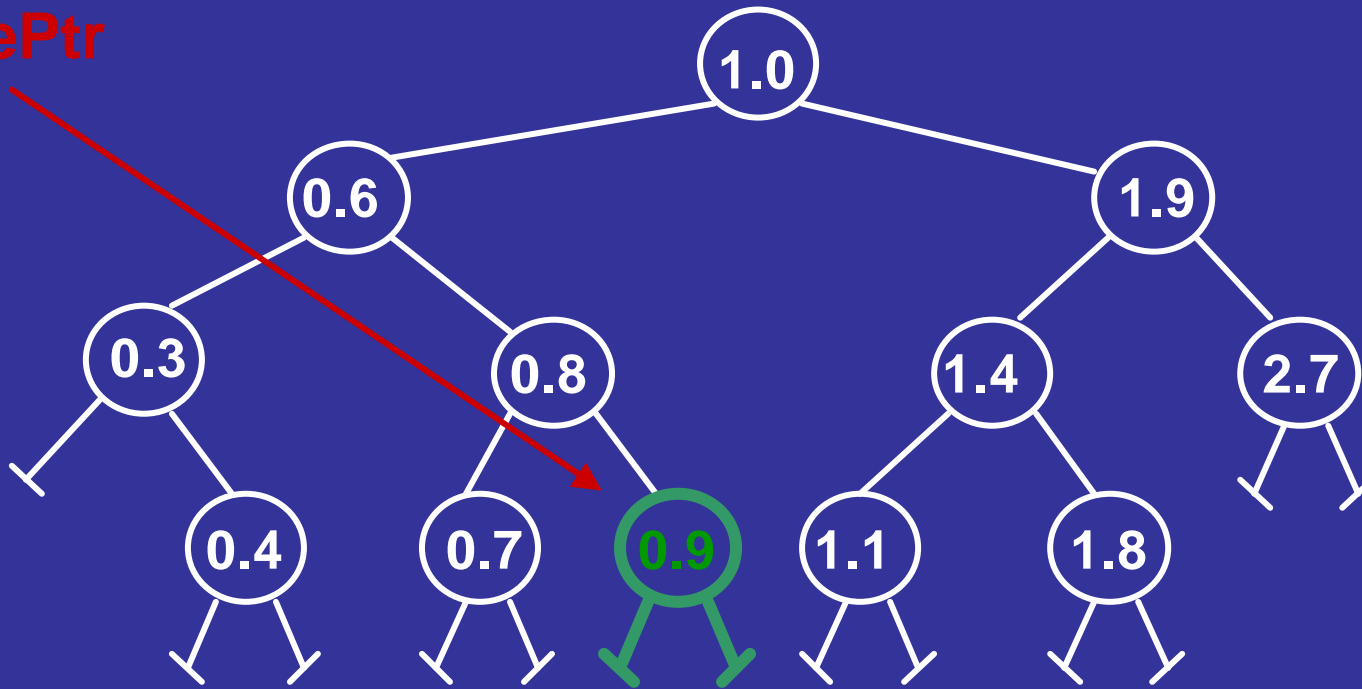


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
```


Insert

Insert 0.9

nodePtr



```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
```

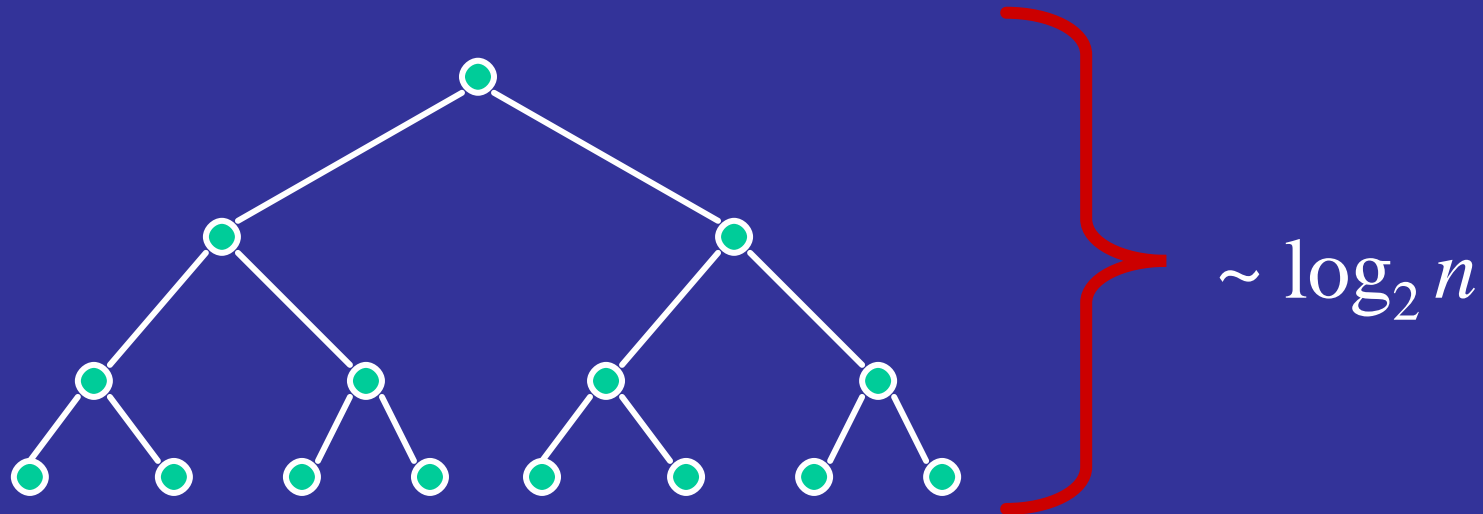
Sorting

To sort a sequence of items:

- **Insert items** into a Binary Search Tree.
- Then **Inorder Traverse** the tree.

Sorting: Analysis

- Average Case: $O(n \log(n))$



- Insert $(i+1)^{\text{th}}$ item: $\sim \log_2(i)$ comparisons

Sort

Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

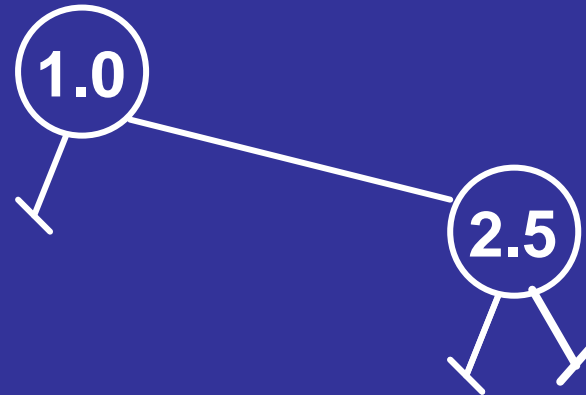
Sort



Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
------------	-----	-----	-----	-----	-----

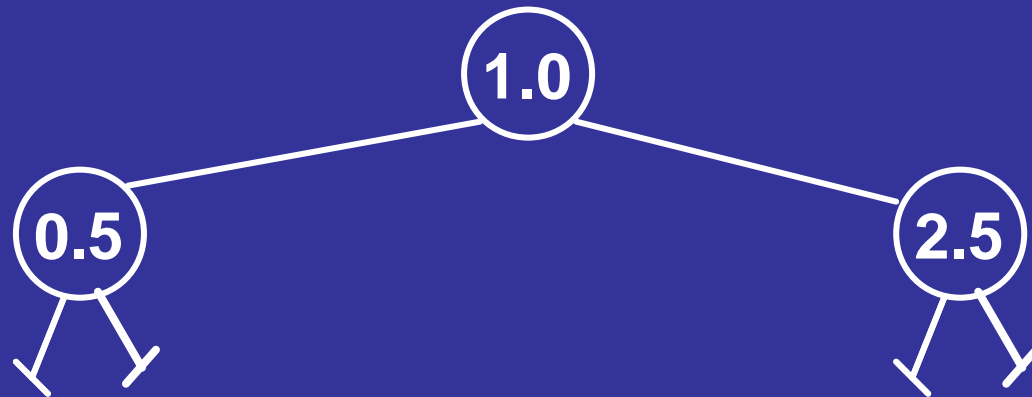
Sort



Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

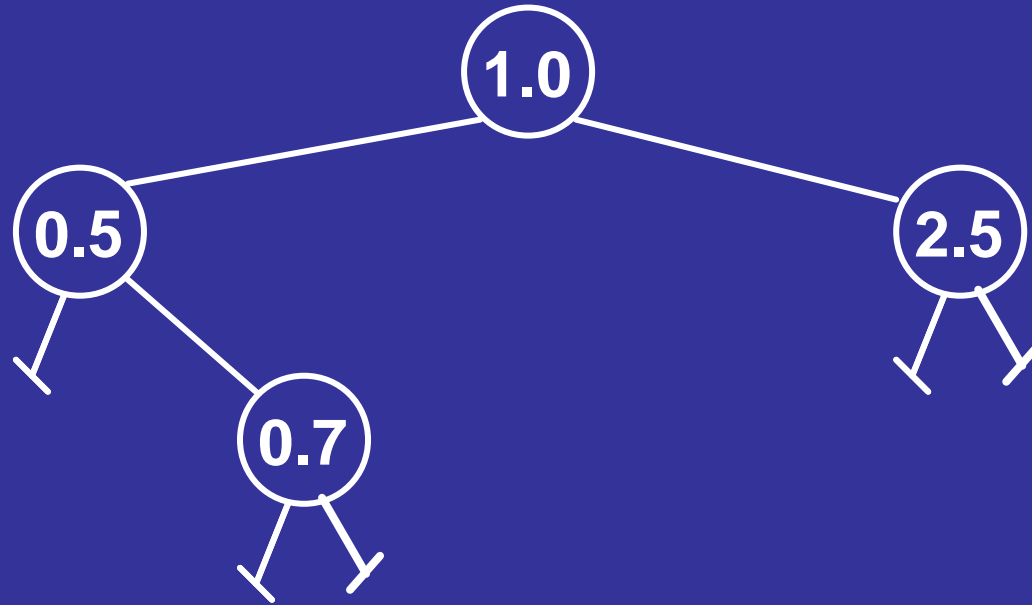
Sort



Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

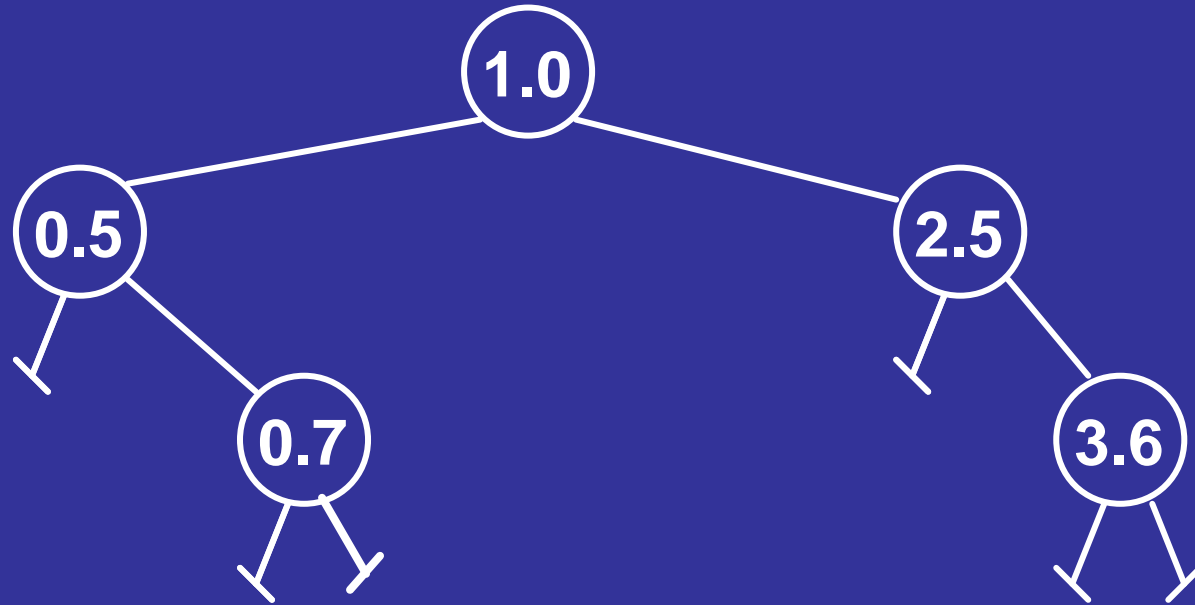
Sort



Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

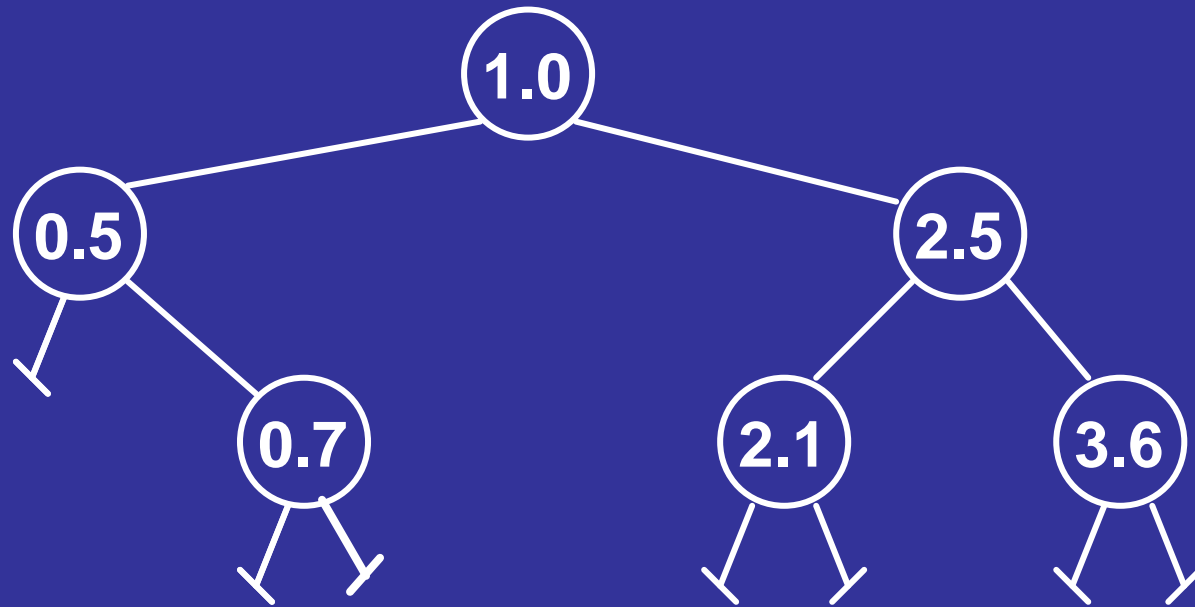
Sort



Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sort



Sort the following list into a binary search tree

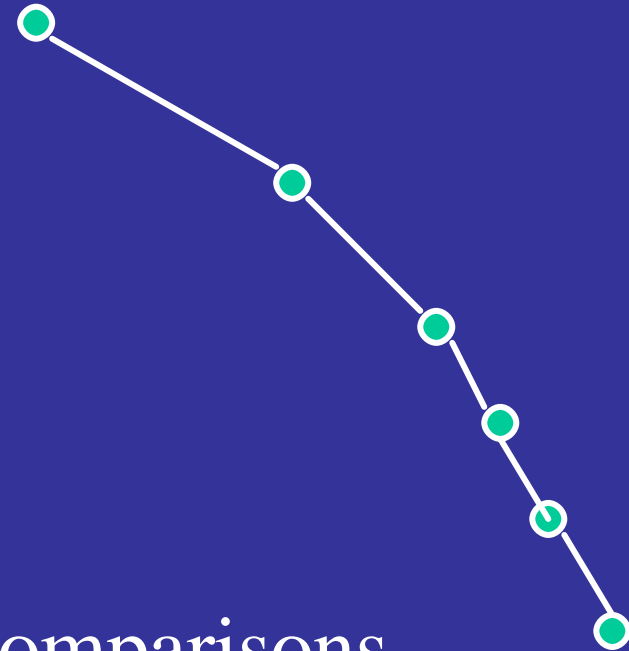
1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sorting: Analysis

- Worst Case: $O(n^2)$

Example:

Insert: 1, 3, 7, 9, 11, 15



- Insert $(i+1)^{\text{th}}$ item: $\sim i$ comparisons

Revision

- Binary Search Tree
- Make Tree Node, Insert item, Search for an item, and Print Inorder.
- Tree sort.

Preparation

- Read Chapter 8.6 in Kruse et al.