Lecture 10

Binary Search Trees I

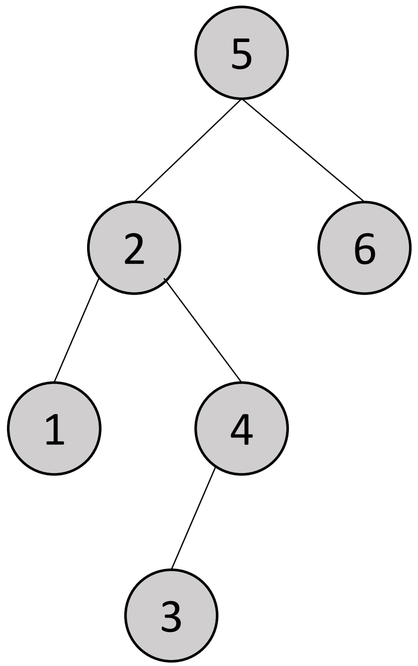
Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Binary Search Tree

- Trees are generally used for index & search.
  - Values in the left subtree are smaller than the root.
  - Values in the right subtree are greater than the root.

- Three new functions to effectively design the tree are needed: insert, remove, search.




```c
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

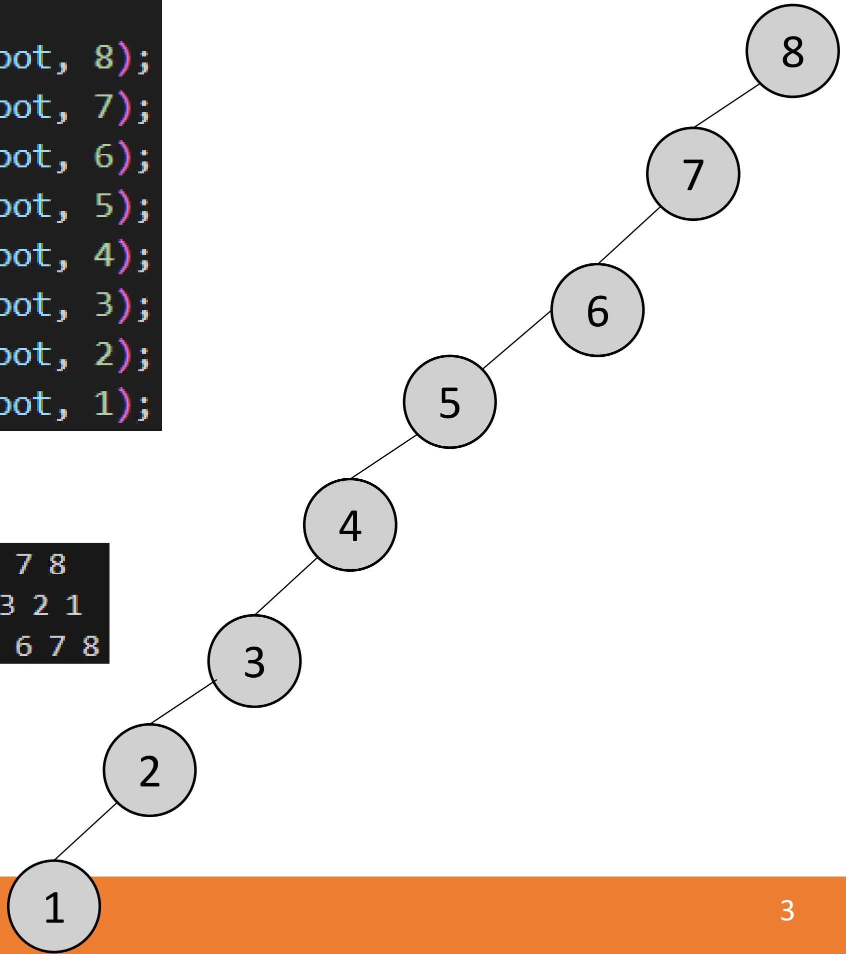
BST Insert

```
Node* insert(Node* node, int data) {
    if (node == NULL) {
        return createNode(data);
    }
    if (data < node->data) {
        node->left = insert(node->left, data);
    } else if (data > node->data) {
        node->right = insert(node->right, data);
    }
    return node;
}
```

- The shape of the tree will change according to the insertion order.
  - No balancing!

```
Node* root = NULL;

root = insert(root, 8);
root = insert(root, 7);
root = insert(root, 6);
root = insert(root, 5);
root = insert(root, 4);
root = insert(root, 3);
root = insert(root, 2);
root = insert(root, 1);
```

```
Inorder traversal: 1 2 3 4 5 6 7 8
Preorder traversal: 8 7 6 5 4 3 2 1
Postorder traversal: 1 2 3 4 5 6 7 8
```



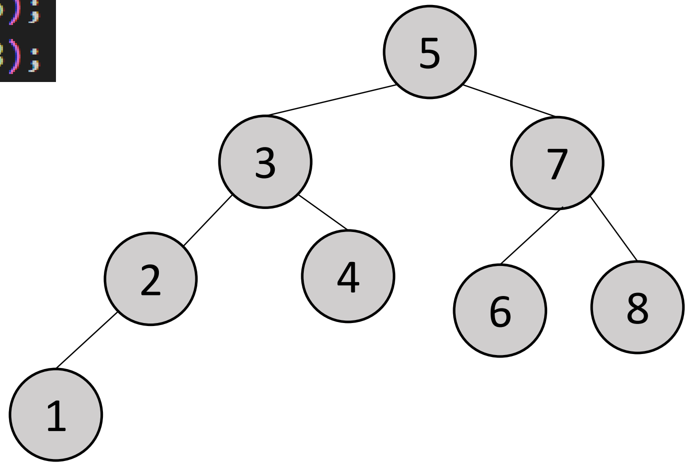
BST Insert

- The shape of the tree will change according to the insertion order.
  - No balancing!

```
Node* insert(Node* node, int data) {
    if (node == NULL) {
        return createNode(data);
    }
    if (data < node->data) {
        node->left = insert(node->left, data);
    } else if (data > node->data) {
        node->right = insert(node->right, data);
    }
    return node;
}
```

```
Node* root = NULL;

root = insert(root, 5);
root = insert(root, 3);
root = insert(root, 7);
root = insert(root, 2);
root = insert(root, 4);
root = insert(root, 1);
root = insert(root, 6);
root = insert(root, 8);
```

```
Inorder traversal: 1 2 3 4 5 6 7 8
Preorder traversal: 5 3 2 1 4 7 6 8
Postorder traversal: 1 2 4 3 6 8 7 5
```

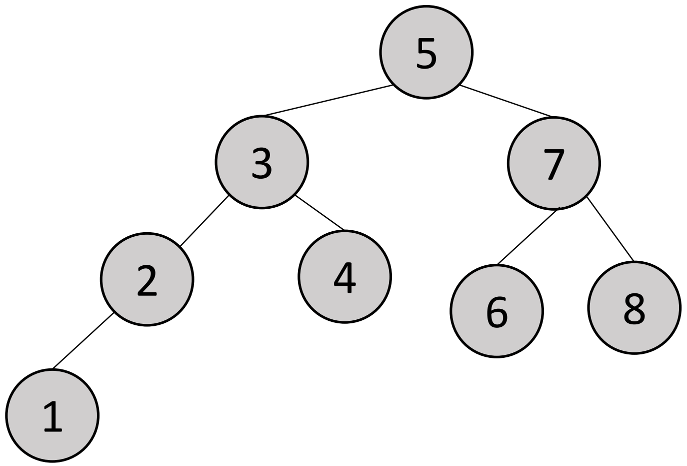


BST Search

```
bool search(Node* node, int key) {
    if (node == NULL) {
        return false;
    }

    printf("Current Node: %d\n", node->data);

    if (node->data == key) {
        return true;
    }
    if (key < node->data) {
        return search(node->left, key);
    }
    return search(node->right, key);
}
```

```
Current Node: 5
Current Node: 3
Current Node: 4
Found value 4
```

- Search function performs a breadth-first search (BFS) using a queue to find a node with the specified key.
- Example: How to find «4»?

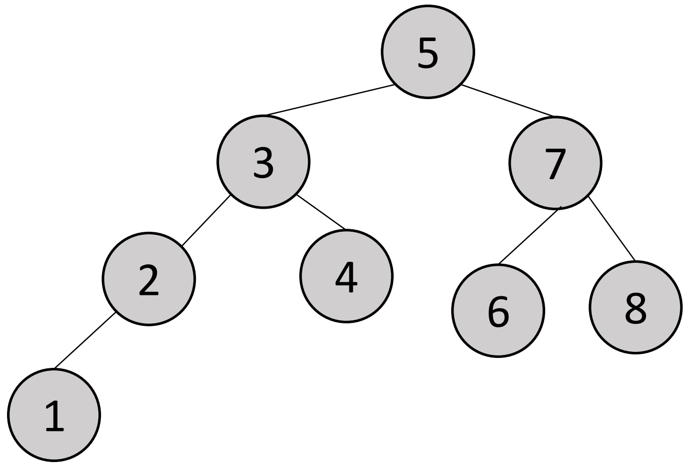


BST Maximum & Minimum

- Checking right/left branches is a valid strategy to find maximum/minimum.

```
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}
```

```
Node* maxValueNode(Node* node) {
    Node* current = node;
    while (current && current->right != NULL) {
        current = current->right;
    }
    return current;
}
```

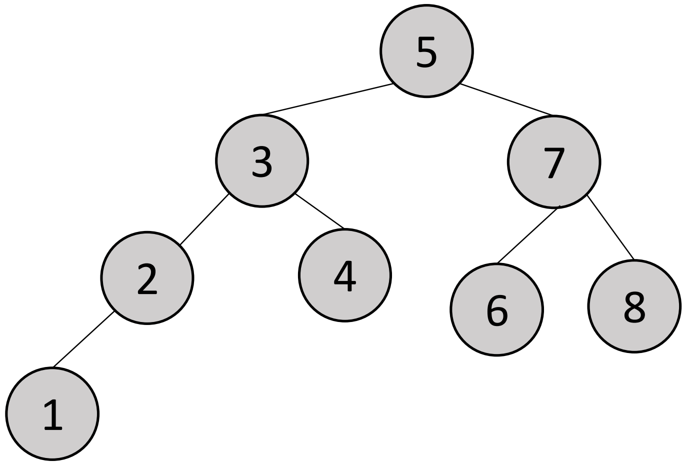



```
printf("Minimum value in the tree: %d\n", minValueNode(root)->data);
printf("Maximum value in the tree: %d\n", maxValueNode(root)->data);
```

```
Minimum value in the tree: 1
Maximum value in the tree: 8
```

How to find the median?

- We need to find the element count N, and then the $\left(\frac{N}{2}+1\right)^{th}$ element.




```
int countNodes(Node* root) {
    if (root == NULL) {
        return 0;
    }
    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

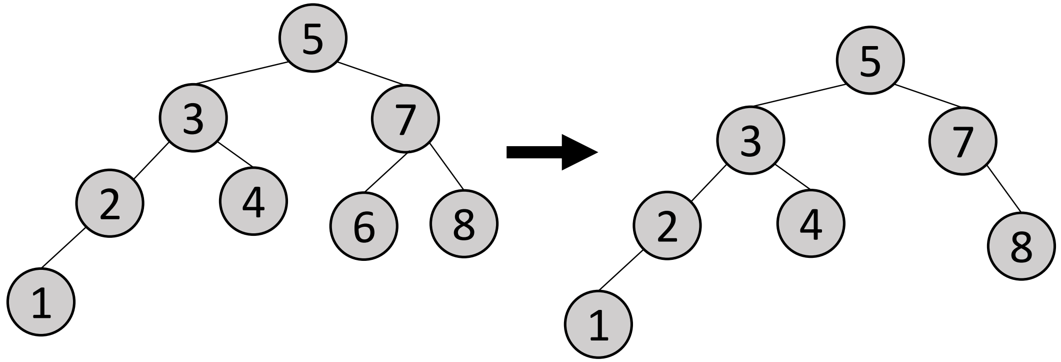
```
void findNthNode(Node* root, int* currentCount, int n, int* result) {
    if (root == NULL) {
        return;
    }

    findNthNode(root->left, currentCount, n, result);

    (*currentCount)++;
    if (*currentCount == n) {
        *result = root->data;
        return;
    }
    findNthNode(root->right, currentCount, n, result);
}
```

```
float findMedian(Node* root) {
    if (root == NULL) {
        return 0.0;
    }

    int totalNodes = countNodes(root);

    if (totalNodes % 2 == 1) {
        int medianValue = 0;
        int currentCount = 0;
        findNthNode(root, &currentCount, (totalNodes / 2) + 1, &medianValue);
        return (float)medianValue;
    } else {
        int leftMiddleValue = 0, rightMiddleValue = 0;
        int currentCount = 0;

        findNthNode(root, &currentCount, totalNodes / 2, &leftMiddleValue);
        currentCount = 0;
        findNthNode(root, &currentCount, (totalNodes / 2) + 1, &rightMiddleValue);

        return (leftMiddleValue + rightMiddleValue) / 2.0;
    }
}
```

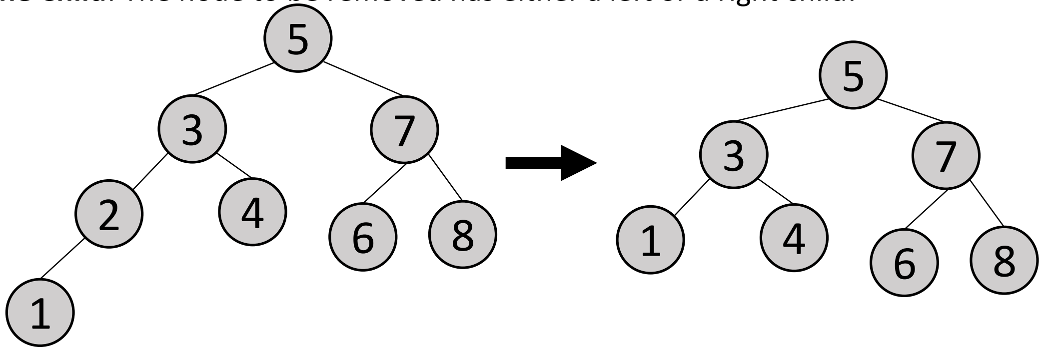
BST Remove

```
Node* removeNode(Node* root, int key) {
    if (root == NULL) {
        return root;
    }

    if (key < root->data) {
        root->left = removeNode(root->left, key);
    } else if (key > root->data) {
        root->right = removeNode(root->right, key);
    } else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }

        Node* temp = minValueNode(root->right);
        root->data = temp->data;

        root->right = removeNode(root->right, temp->data);
    }

    return root;
}
```

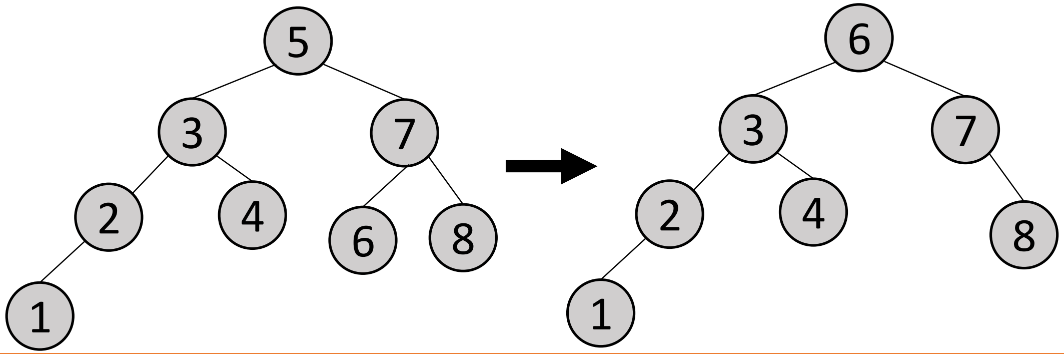
- There are three cases to remove a node from a binary search tree.
  - **Leaf Node**: The node to be removed has no children.



  - **One Child**: The node to be removed has either a left or a right child.



  - **Two Children**: The node to be removed has both a left and a right child.



Big-O Time Complexities of the Functions

- createNode
- insert
- search
- maximum/minimum
- countnodes
- findNthNode
- findMedian
- removeNode

Big-O Time Complexities of the Functions

- createNode: $O(1)$
- insert: $O(logn)$ for best case, $O(n)$ for worst case
- search: $O(logn)$ for best case, $O(n)$ for worst case
- maximum/minimum: $O(logn)$ for best case, $O(n)$ for worst case
- countnodes: $O(n)$
- findNthNode: $O(n)$
- findMedian: $O(n)$
- removeNode: $O(logn)$ for best case, $O(n)$ for worst case

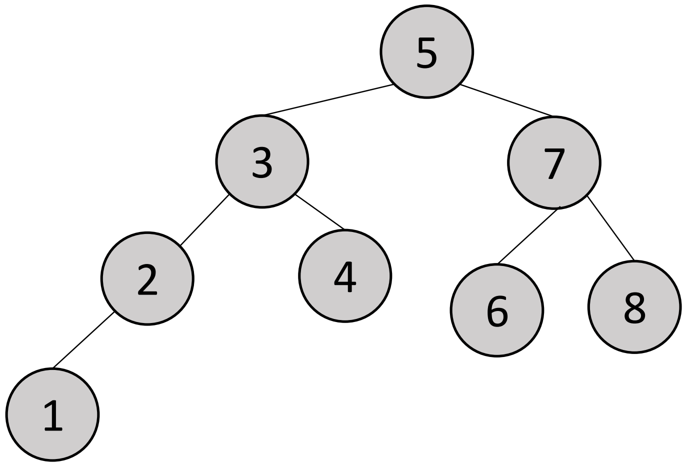
A Vector-Based Structure for Binary Trees

- Another method to represent a binary tree T, involves assigning numbers to its nodes.
  - For each node v in T, we define an integer f(v).
  - If v is the root of T, then $f(v) = 0$
  - If v is the left child of node u, then $f(v) = 2 f(u)+1$.
  - If v is the right child of node u, then $f(v) = 2 f(u)+2$

```c
typedef struct {
    int* tree;
    int capacity;
} ArrayBST;
```

```c
void create(ArrayBST* bst, int capacity) {
    bst->capacity = capacity;
    bst->tree = (int*)malloc(capacity * sizeof(int));
    if (bst->tree == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    for (int i = 0; i < capacity; i++) {
        bst->tree[i] = INT_MIN;
    }
}
```

```c
void insert(ArrayBST* bst, int value) {
    if (bst->tree[0] == INT_MIN) {
        bst->tree[0] = value;
        return;
    }

    int index = 0;
    while (index < bst->capacity) {
        if (bst->tree[index] == INT_MIN) {
            bst->tree[index] = value;
            return;
        } else if (value < bst->tree[index]) {
            index = 2 * index + 1;
        } else {
            index = 2 * index + 2;
        }
    }

    printf("Tree capacity exceeded!");
}
```



0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
5	3	7	2	4	6	8	1

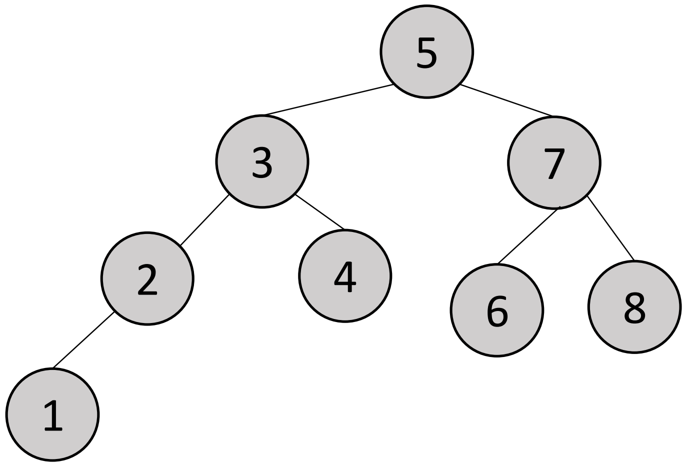
A Vector-Based Structure for Binary Trees

```c
void print_as_array(ArrayBST* bst) {
    printf("Tree elements: ");
    for (int i = 0; i < bst->capacity; i++) {
        if (bst->tree[i] == INT_MIN) {
            printf("_ ");
        } else {
            printf("%d ", bst->tree[i]);
        }
    }
    printf("\n");
}
```

```c
void preorder(ArrayBST* bst, int index) {
    if (index >= bst->capacity || bst->tree[index] == INT_MIN) {
        return;
    }
    printf("%d ", bst->tree[index]);
    preorder(bst, 2 * index + 1);
    preorder(bst, 2 * index + 2);
}
```

```c
ArrayBST bst;

create(&bst, 10);

insert(&bst, 5);
insert(&bst, 3);
insert(&bst, 7);
insert(&bst, 2);
insert(&bst, 4);
insert(&bst, 6);
insert(&bst, 8);

print_as_array(&bst);
printf("Preorder traversal: ");
preorder(&bst, 0);
printf("\n");
free(bst.tree);
```

```
Tree elements: 5 3 7 2 4 6 8 _ _ _
Preorder traversal: 5 3 2 4 7 6 8
```



0	1	2	3	4	5	6	7
5	3	7	2	4	6	8	1