BINARY SEARCH TREE

// Delete a Node with a given value from a Binary Search Tree.

```
Node* deleteNode(Node* root, int data) {
	if (root == NULL)
		return NULL;

	if (data < root->key) {
		root->left = deleteNode(root->left, data);
	}
	else if (data > root->key) {
		root->right = deleteNode(root->right, data);
	}
	// Node found
	else {
		// Case 1: No child (leaf node)
		if (root->left == NULL && root->right == NULL) {
			delete root;
			return NULL;
		}

		// Case 2: One child (right)
		else if (root->left == NULL) {
			Node* temp = root->right;
			delete root;
			return temp;
		}

		// Case 2: One child (left)
		else if (root->right == NULL) {
			Node* temp = root->left;
			delete root;
			return temp;
		}

		// Case 3: Two children
		// Find the smallest node in the right subtree
		Node* temp = root->right;
		while (temp->left != NULL) {
			temp = temp->left;
		}
		root->key = temp->key;
			root->right = deleteNode(root->right, temp->key);
		}

		return root;
 }

void levelOrder(Node* root) {
	if (root == NULL)
		return;
	// BFS
	queue<Node*> q;
	q.push(root);
	while (!q.empty()) {
		Node* temp = q.front();
		q.pop();
		cout << temp->key << " ";
		if (temp->left != NULL) {
			q.push(temp->left);
		}
		if (temp->right != NULL) {
			q.push(temp->right);
		}
	}
 }

// Calculate the height of a given Binary Search Tree.
int height(Node* root) {
	if (root == NULL)
		return 0;
	int leftHeight = height(root->left);
	int rightHeight = height(root->right);
	return (leftHeight > rightHeight) ? leftHeight + 1 :
rightHeight + 1;
}

// Count the number of Nodes from a given Binary Search Tree.
int countNode(Node* root) {
	if (root == NULL)
		return 0;
	return countNode(root->left) + countNode(root->right) + 1;
}
```

```
// Calculate the total value of all Nodes from a given Binary Search
Tree.
int sumNode(Node* root) {
	if (root == NULL)
		return 0;
	return sumNode(root->left) + sumNode(root->right) + root-
>key;
}

// Count the number of leaves from a given Binary Search Tree.
int countLeaf(Node* root) {
	if (root == NULL)
		return 0;
	if (root->left == NULL && root->right == NULL)
		return 1;
	return countLeaf(root->left) + countLeaf(root->right);
}

// Count the number of Nodes from a given Binary Search Tree which
key value is less than a given value.
int countLess(Node* root, int x) {
	if (root == NULL)
		return 0;
	if (root->key < x) {
		return countLess(root->left, x) + countLess(root-
>right, x) + 1;
	}
	return countLess(root->left, x) + countLess(root->right, x);
}

// Count the number of Nodes from a given Binary Search Tree which
key value is greater than a given value.
int countGreater(Node* root, int x) {
	if (root == NULL)
		return 0;
	if (root->key > x) {
		return countGreater(root->left, x) + countGreater(root-
>right, x) + 1;
	}
	return countGreater(root->left, x) + countGreater(root-
>right, x);
 }
```

AVL TREE

```
int height(Node* node) {
	if (node == NULL) return 0; // height initialized with 1
	return node->height;
}

/*
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left
side) or x (on the right side)

     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - - >      T1   y
  / \       < - - - - - - - -          / \
 T1  T2     Left Rotation             T2  T3
*/

Node* rightRotate(Node* y) {
	Node* x = y->left;
	Node* z = x->right;

	// rotate
	x->right = y;
	y->left = z;

	// update heights
	y->height = max(height(y->left), height(y->right)) + 1;
	x->height = max(height(x->left), height(x->right)) + 1;

	return x;
}

Node* leftRotate(Node* x) {
	Node* y = x->right;
	Node* z = y->left;

	// rotate
	y->left = x;
	x->right = z;

	// update heights
	x->height = max(height(x->left), height(x->right)) + 1;
	y->height = max(height(y->left), height(y->right)) + 1;

	return y;
}

int getBalance(Node* root) {
	if (root == NULL) return 0;
	return height(root->left) - height(root->right);
}

Node* insertNode(Node* root, int data) {
	if (root == NULL) return newNode(data);

	if (data < root->key) {
		root->left = insertNode(root->left, data);
	}
	else if (data > root->key) {
		root->right = insertNode(root->right, data);
	}
	// do not add existing key
	else {
		return root;
	}

	// update height
	root->height = max(height(root->left), height(root->right)) +
1;

	// get root's balance factor
	int balance = getBalance(root);

	// LL
	if (balance > 1 && data < root->left->key) {
		return rightRotate(root);
	}
	// RR
	if (balance < -1 && data > root->right->key) {
		return leftRotate(root);
	}
	// LR
	if (balance > 1 && data > root->left->key) {
		root->left = leftRotate(root->left);
		return rightRotate(root);
```

```
        }
        // RL
        if (balance < -1 && data < root->right->key) {
                root->right = rightRotate(root->right);
                return leftRotate(root);
        }

        return root;
}

Node* searchNode(Node* root, int data) {
        if (root == NULL || data == root->key) return root;

        if (data < root->key) {
                return searchNode(root->left, data);
        }

        return searchNode(root->right, data);
}

Node* deleteNode(Node* root, int data) {
        if (root == NULL) return NULL;

        if (data < root->key) {
                root->left = deleteNode(root->left, data);
        }
        else if (data > root->key) {
                root->right = deleteNode(root->right, data);
        }
        // Node found
        else {

                // Case 1: No child (leaf node)
                if (root->left == NULL && root->right == NULL) {
                        delete root;
                        return NULL;
                }

                // Case 2: One child (right)
                else if (root->left == NULL) {
                        Node* temp = root->right;
                        delete root;
                        return temp;
                }

                // Case 2: One child (left)
                else if (root->right == NULL) {
                        Node* temp = root->left;
                        delete root;
                        return temp;
                }

                // Case 3: Two children
                // Find the smallest node in the right subtree
                Node* temp = root->right;
                while (temp->left != NULL) {
                        temp = temp->left;
                }
                root->key = temp->key;
                root->right = deleteNode(root->right, temp->key);
        }

        if (root == NULL) return root;

        // UPDATE HEIGHT
        root->height = max(height(root->left), height(root->right)) +
1;

        // GET THE BALANCE FACTOR
        int balance = getBalance(root);

        // LL
        if (balance > 1 && getBalance(root->left) >= 0)
                return rightRotate(root);

        // LR
        if (balance > 1 && getBalance(root->left) < 0) {
                root->left = leftRotate(root->left);
                return rightRotate(root);
        }

        // RR
        if (balance < -1 && getBalance(root->right) <= 0)
                return leftRotate(root);

        // RL
        if (balance < -1 && getBalance(root->right) > 0) {
                root->right = rightRotate(root->right);
                return leftRotate(root);
        }

        return root;
 }

/*      -------------------- EXERCISE 2 -------------------- */

bool isFull(Node* root) {
        if (root == NULL) return true;

        if (root->left == NULL && root->right == NULL) {
                return true;
        }
        if (root->left && root->right) {
                return isFull(root->left) && isFull(root->right);
        }

        return false;
}

bool isComplete(Node* root) {
        if (root == NULL) return true;
        queue<Node*> q;
        q.push(root);

        // if a NULL node is met, all its following nodes must be
NULL
        bool flag = false;

        // level order traversal
        while (!q.empty()) {
                Node* temp = q.front();
                q.pop();

                // check left child
                if (temp->left == NULL) {
                        flag = true;
                }
                else {
                        if (flag) return false;
                        q.push(temp->left);
                }

                // check right child
                if (temp->right == NULL) {
                        flag = true;
                }
                else {
                        if (flag) return false;
                        q.push(temp->right);
                }
        }

        return true;
}

int countNodes(Node* root) {
        if (root == NULL) return 0;

        return 1 + countNodes(root->left) + countNodes(root->right);
}

bool isPerfect(Node* root) {
        if (root == NULL) return true;

        // a perfect binary tree has 2^(h+1) - 1 nodes
        int h = root->height;
        return countNodes(root) == pow(2, h + 1) - 1;
}

// Find the least common ancestor for any two given nodes in AVL
Tree.
int findCommonAncestor(Node* root, int x, int y) {
        if (root == NULL) {
                cout << "Empty tree" << endl;
                return -1;
        }

        // check if x and y are nodes in the tree
        if (searchNode(root, x) == NULL || searchNode(root, y) ==
NULL) {
                cout << "Node not found" << endl;
                return -1;
	}

	if (root->key > x && root->key > y) {
		return findCommonAncestor(root->left, x, y);
	}
	if (root->key < x && root->key < y) {
		return findCommonAncestor(root->right, x, y);
	}

	return root->key;
}

/*
Find all nodes with 2 child nodes, and the left child is a divisor
of the right child. Print them to the console in ascending order.
*/
void printSpecialNodes(Node* root) {
	if (root == NULL) return;

	// in-order traversal
	printSpecialNodes(root->left);
	if (root->left != NULL && root->right != NULL) {
		if (root->left->key != 0 && root->right->key % root-
>left->key == 0) {
			cout << root->key << " ";
		}
	}
	printSpecialNodes(root->right);
 }
```

MIN-HEAP

```
struct Heap {
	int* arr;
	int capacity;
	int size;
	Heap(int capacity);
};

Heap::Heap(int capacity) {
	this->capacity = capacity;
	this->size = 0;
	this->arr = new int[capacity];
}

// Insert a new value into the Min-Heap while maintaining the Min-
Heap property.
void insert(Heap* heap, int value) {
	if (heap->size == heap->capacity) {
		cout << "Heap is full" << endl;
		return;
	}
	int i = heap->size;
	heap->arr[i] = value;
	heap->size++;

	int parent = (i - 1) / 2;
	while (i != 0 && heap->arr[parent] > heap->arr[i]) {
		swap(heap->arr[parent], heap->arr[i]);
		i = parent;
		parent = (i - 1) / 2;
	}
}

// Min-Heapify the subtree rooted at the given index.
void heapify(Heap* heap, int index) {
	int left = 2 * index + 1;
	int right = 2 * index + 2;
	int minValue = index;

	if (left < heap->size && heap->arr[left] < heap-
>arr[minValue]) {
		minValue = left;
	}
	if (right < heap->size && heap->arr[right] < heap-
>arr[minValue]) {
		minValue = right;
	}

	if (minValue != index) {
		swap(heap->arr[index], heap->arr[minValue]);
		heapify(heap, minValue);
	}
}

int extractMin(Heap* heap) {
	if (heap->size == 0) {
		cout << "Heap is empty" << endl;
		return -1;
	}
	if (heap->size == 1) {
		heap->size--;
		return heap->arr[0];
	}

	int minValue = heap->arr[0];
	heap->arr[0] = heap->arr[heap->size - 1];
	heap->size--;
	heapify(heap, 0);

	return minValue;
}

int getMin(Heap* heap) {
	if (heap->arr == NULL) {
		cout << "Heap is NULL" << endl;
		return -1;
	}
	if (heap->size == 0) {
		cout << "Heap is empty" << endl;
		return -1;
	}
	return heap->arr[0];
}

void deleteKey(Heap* heap, int index) {
        if (index >= heap->size) {
                cout << "Index out of range" << endl;
                return;
        }
        heap->arr[index] = INT_MIN;
        int i = index;
        int parent = (i - 1) / 2;
        while (i != 0 && heap->arr[parent] > heap->arr[i]) {
                swap(heap->arr[parent], heap->arr[i]);
                i = parent;
                parent = (i - 1) / 2;
        }
        extractMin(heap);
}

// Additional Operations and Properties

int countNodes(Heap* heap) {
        if (heap->arr == NULL) {
                cout << "Heap is NULL" << endl;
                return -1;
        }
        return heap->size;
}

int height(Heap* heap) {
        if (heap->arr == NULL) {
                cout << "Heap is NULL" << endl;
                return -1;
        }
        if (heap->size == 0) {
                cout << "Heap is empty" << endl;
                return -1;
        }
        return (int)log2(heap->size);
}

void printLevelOrder(Heap* heap) {
        if (heap->arr == NULL) {
                cout << "Heap is NULL" << endl;
                return;
        }
        if (heap->size == 0) {
                cout << "Heap is empty" << endl;
                return;
        }
        for (int i = 0; i < heap->size; i++) {
                cout << heap->arr[i] << " ";
        }
        cout << endl;
}

void heapSort(Heap* heap) {
        if (heap->arr == NULL) {
                cout << "Heap is NULL" << endl;
                return;
        }
        if (heap->size == 0) {
                cout << "Heap is empty" << endl;
                return;
        }

        int n = heap->size;
        int* sortedArr = new int[n];

        // Use a temporary array to store the sorted array
        for (int i = 0; i < n; i++) {
                sortedArr[i] = extractMin(heap);
        }

        // Copy sorted array back to heap->arr
        for (int i = 0; i < n; i++) {
                heap->arr[i] = sortedArr[i];
        }

        // Restore the heap size
        heap->size = n;

        delete[] sortedArr;
 }

// Normally used to sort descending
void heapSort(Heap* heap) {
        if (heap->arr == NULL) {
                cout << "Heap is NULL" << endl;
                return;
        }
        if (heap->size == 0) {
                cout << "Heap is empty" << endl;
                return;
        }

        // Preserve the original heap size
        int originalSize = heap->size;

        // Step 1: Build the heap
        for (int i = heap->size / 2 - 1; i >= 0; i--) {
                heapify(heap, i);
        }

        // Step 2: Extract elements from the heap one by one
        for (int i = heap->size - 1; i > 0; i--) {
                // Move current root to end
                swap(heap->arr[0], heap->arr[i]);

                // Reduce the heap size
                heap->size--;

                // Heapify the root element to maintain the heap
property
                heapify(heap, 0);
        }

        // Restore the original heap size
        heap->size = originalSize;
 }
```

FILE I/O

```
int fprintf ( FILE * f, const char * format, ... );
int sprintf( char* buffer, const char* format, ... );

int fscanf ( FILE * f, const char * format, ... );
VD: fscanf(fptr,”%d”, &num);

char* fgets( char* str, int count, std::FILE* stream );
```

Nếu trước fgets() bạn dùng fscanf() và để dư kí tự enter sẽ bị trôi lệnh, cách xử lý là loại bỏ enter bằng hàm fgetc() hoặc fscanf(f, "\n"), hàm này sẽ đọc 1 kí tự trong file.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
std::size_t fwrite( const void* buffer, std::size_t size, std::size_t count, std::FILE* stream );
```

buffer	-	pointer to the first object in the array to be written
size	-	size of each object
count	-	the number of the objects to be written
stream	-	output file stream to write to

```
VD: fwrite(&num, sizeof(struct threeNum), 1, fptr);

fread(addressData, sizeData, numbersData, pointerToFile);
VD: fread(&num, sizeof(struct threeNum), 1, fptr);
```

```
int fseek( std::FILE* stream, long offset, int origin );
```

stream	-	file stream to modify
offset	-	number of characters to shift the position relative to origin
origin	-	position to which offset is added. It can have one of the following values: SEEK_SET, SEEK_CUR, SEEK_END