Zaijan Alvarado

Symon Cristoffer Cano

Jon Matthew B. Mella Data Structure and Algorithm

Finals Project in DSA

Overview of the Game

The Creature Game is a multifaceted program that allows users to interact with creatures possessing unique names and power levels. The game employs three data structures: Binary Search Trees (BST), Binary Trees (BT), and Heaps (Max-Heap and Min-Heap), each offering distinct functionalities to manage and manipulate the creatures. Players can insert creatures, traverse data structures, search for specific creatures based on power levels, and even delete them. The game begins with a mandatory addition of at least five creatures and then offers various options to work with the implemented data structures.

How to Run the Game

To play the Creature Game, compile and run the program in a C++ environment. Upon starting, the user will input at least five creatures with their names and power levels. Afterward, the main menu provides options to work with BST, BT, or Heaps. For BST and BT, users can insert, delete, search, or traverse (preorder, inorder, and postorder). The Heap menu allows the insertion of creatures into Max-Heap or Min-Heap, display of current heaps, and heapification of creature powers. The program runs interactively, enabling users to navigate between menus and exit the game when done.

Code and Description

Data Structures and Functionalities

1. Creature Structure: Represents a creature with attributes name and power.
2. Node Structure: Represents a node in the BST or BT, containing a Creature, and pointers to left and right child nodes.

Binary Search Tree (BST)

* Insertion: Inserts a creature into the BST based on its power.
* Search: Locates a node with a specific power level.
* Deletion: Removes a node by handling cases of one child, no child, or two children.
* Traversal: Supports preorder, inorder, and postorder traversal.

Binary Tree (BT)

* Insertion: Adds a creature to the BT using level-order traversal.
* Search: Locates a node based on power.
* Deletion: Deletes a node and replaces it with the deepest node.
* Traversal: Offers preorder, inorder, and postorder traversal.

Heaps

* Max-Heap and Min-Heap: Priority queues manage creatures by their power levels. Max-Heap orders from strongest to weakest, while Min-Heap does the reverse.
* Heapify: Converts an array of powers into either a Max-Heap or Min-Heap and displays the results.

Main Functions

* Display Creatures: Lists all creatures with their names and powers.
* BST and BT Menus: Allow users to interact with respective structures.
* Heap Menu: Provides heap operations, including insertion, heapification, and heap display.

Gameplay Flow

* Initial Setup: Prompts the user to input at least five creatures.
* Main Menu: Offers choices to work with BST, BT, or Heaps.
* Exit Option: Allows users to end the game gracefully.

Code of the Program:

#include <iostream> //handles input and output operations, such as printing to the console and reading user input.

#include <vector> //provides the std::vector container, a dynamic array for storing elements that can grow or shrink in size.

#include <memory> //includes utilities for managing dynamic memory, such as smart pointers (std::unique\_ptr, std::shared\_ptr).

#include <queue> //provides functionality for queue containers like std::queue, which follow a FIFO order.

#include <algorithm> //offers a variety of algorithms, such as sorting and searching, to be used with containers.

#include <string> //provides the std::string class for working with text and string manipulation.

using namespace std;

/\*The Creature struct represents a creature with two attributes name, a string for the creature's name,

and power, an integer for the creature's power.

The constructor initializes these attributes when a new Creature object is created.\*/

struct Creature {

string name;

int power;

Creature(string n, int p) : name(n), power(p) {}

};

/\*The Node struct is designed for use in a binary tree or binary search tree (BST).

It holds a Creature object and has two shared pointers (left and right) to its child nodes,

allowing for left and right subtrees. The constructor initializes a node with a Creature and

sets the child pointers to nullptr. This structure supports tree traversal and operations like insertion and search.\*/

struct Node {

Creature creature;

shared\_ptr<Node> left;

shared\_ptr<Node> right;

Node(Creature c) : creature(c), left(nullptr), right(nullptr) {}

};

//this line starts the function of binary search tree

/\*This function adds a new Creature to the Binary Search Tree by

recursively comparing its power with the current node’s power.

It inserts the new node into the left subtree if the power is smaller,

or the right subtree if the power is greater, and then returns the updated root.\*/

shared\_ptr<Node> insertBST(shared\_ptr<Node> root, Creature c) {

if (!root) return make\_shared<Node>(c);

if (c.power < root->creature.power)

root->left = insertBST(root->left, c);

else

root->right = insertBST(root->right, c);

return root;

}

/\*This function searches for a Creature with a specific power in the Binary Search Tree.

It recursively navigates left or right, depending on whether the target power is smaller or larger

than the current node’s power, and returns the node if the value is found\*/

shared\_ptr<Node> searchBST(shared\_ptr<Node> root, int power) {

if (!root || root->creature.power == power)

return root;

if (power < root->creature.power)

return searchBST(root->left, power);

return searchBST(root->right, power);

}

/\*The deleteBST function removes a Creature node with a given power in a Binary Search Tree (BST)

by recursively finding the node and handling three cases: if the node has no children or one child,

it is removed by returning the non-null child; if the node has two children,

it finds the inorder successor, replaces the node's creature with the successor’s creature, and then

deletes the successor node recursively from the right subtree. This ensures the tree remains a valid BST.\*/

shared\_ptr<Node> deleteBST(shared\_ptr<Node> root, int power) {

if (!root) return nullptr;

if (power < root->creature.power) {

root->left = deleteBST(root->left, power);

} else if (power > root->creature.power) {

root->right = deleteBST(root->right, power);

} else {

// Node with one or no child

if (!root->left) return root->right;

if (!root->right) return root->left;

// Node with two children: Get the inorder successor (smallest in the right subtree)

shared\_ptr<Node> temp = root->right;

while (temp->left) temp = temp->left;

root->creature = temp->creature;

root->right = deleteBST(root->right, temp->creature.power);

}

return root;

}

/\*The function of Preorder is it visits the root, then the left subtree,

followed by the right subtree, printing each node's name and power as it goes.\*/

void preorder(shared\_ptr<Node> root) {

if (!root) return;

cout << root->creature.name << " (" << root->creature.power << ") ";

preorder(root->left);

preorder(root->right);

}

/\*Inorder traversal first visits the left subtree, then the root, and finally the right subtree,

which, in a Binary Search Tree, prints the nodes in ascending order of their power.\*/

void inorder(shared\_ptr<Node> root) {

if (!root) return;

inorder(root->left);

cout << root->creature.name << " (" << root->creature.power << ") ";

inorder(root->right);

}

/\*Postorder traversal visits the left subtree, then the right subtree, and finally the root,

printing the node’s information after visiting both children.\*/

void postorder(shared\_ptr<Node> root) {

if (!root) return;

postorder(root->left);

postorder(root->right);

cout << root->creature.name << " (" << root->creature.power << ") ";

}

//This line starts the function of binary tree

// Forward declaration of deleteDeepestNode

void deleteDeepestNode(shared\_ptr<Node> root, shared\_ptr<Node> deepestNode);

/\*This function deletes a node with a specified power in a Binary Tree by performing a level-order traversal

to find both the node to delete and the deepest node. Once the node to delete is found,

it replaces its value with that of the deepest node, and then calls deleteDeepestNode to remove the deepest node from the tree.\*/

shared\_ptr<Node> deleteBT(shared\_ptr<Node> root, int power) {

if (!root) return nullptr;

// Perform level order traversal to find the deepest node

queue<shared\_ptr<Node>> q;

q.push(root);

shared\_ptr<Node> temp = nullptr;

shared\_ptr<Node> nodeToDelete = nullptr;

// Standard level order traversal

while (!q.empty()) {

temp = q.front();

q.pop();

// If we find the node to delete, store it

if (temp->creature.power == power) {

nodeToDelete = temp;

}

// Push children of the current node into the queue

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

}

// If we found the node to delete, replace it with the deepest node's value

if (nodeToDelete) {

// Copy the deepest node's value into the node to be deleted

nodeToDelete->creature = temp->creature;

// Now delete the deepest node (temp) from the tree

deleteDeepestNode(root, temp);

}

return root;

}

/\*This helper function removes the deepest node from the tree by performing a level-order traversal to locate it.

Once found, it updates the parent node's pointer (left or right) to nullptr, effectively removing the deepest node from the tree.\*/

void deleteDeepestNode(shared\_ptr<Node> root, shared\_ptr<Node> deepestNode) {

queue<shared\_ptr<Node>> q;

q.push(root);

// Standard level order traversal

shared\_ptr<Node> temp = nullptr;

while (!q.empty()) {

temp = q.front();

q.pop();

// If this is the deepest node, remove it

if (temp->left == deepestNode) {

temp->left = nullptr;

return;

} else if (temp->right == deepestNode) {

temp->right = nullptr;

return;

}

// Push children of the current node into the queue

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

}

}

/\*The searchBT function performs a level-order traversal (breadth-first search) of the Binary Tree

to find a node with a specified power. It begins by checking if the root is nullptr and

returns nullptr if the tree is empty. It then uses a queue to process each node level by level.

As it processes each node, it checks if the node's power matches the target; if a match is found,

the function returns the node. If no match is found after traversing all nodes,

it returns nullptr, indicating that the node does not exist in the tree.\*/

shared\_ptr<Node> searchBT(shared\_ptr<Node> root, int power) {

if (!root) return nullptr;

queue<shared\_ptr<Node>> q;

q.push(root);

while (!q.empty()) {

shared\_ptr<Node> current = q.front();

q.pop();

if (current->creature.power == power) {

return current; // Return the node if power matches

}

if (current->left) q.push(current->left);

if (current->right) q.push(current->right);

}

return nullptr; // Return nullptr if the creature is not found

}

/\*The preorderBT function is a recursive traversal that visits the root node first,

followed by its left subtree and then the right subtree.

It prints the name and power of each node as it visits them.\*/

void preorderBT(shared\_ptr<Node> root) {

if (!root) return;

// Visit the root node

cout << root->creature.name << " (" << root->creature.power << ") ";

// Traverse the left subtree

preorderBT(root->left);

// Traverse the right subtree

preorderBT(root->right);

}

/\*The inorderBT function performs an in-order traversal of the Binary Tree,

where it visits the left subtree first, then the root node, and then the right subtree.

This is commonly used to print nodes in sorted order for a Binary Search Tree (BST). \*/

void inorderBT(shared\_ptr<Node> root) {

if (!root) return; // Base case: If the root is null, return

inorderBT(root->left); // Recursively visit the left subtree

cout << root->creature.name << " (" << root->creature.power << ") "; // Visit the root

inorderBT(root->right); // Recursively visit the right subtree

}

/\*The postorderBT function is a recursive traversal that visits the left subtree first,

then the right subtree, and then the root node. It is used when you want

to process all descendants before the node itself. \*/

void postorderBT(shared\_ptr<Node> root) {

if (!root) return;

postorderBT(root->left);

postorderBT(root->right);

cout << root->creature.name << " (" << root->creature.power << ") ";

}

/\*The insertBT function inserts a new node into a Binary Tree using a level-order traversal.

It checks if the root is nullptr and creates a new node if the tree is empty.

If the tree is not empty, it uses a queue to traverse each level of the tree,

checking each node’s left and right children for an empty spot. Once it finds an empty spot,

it inserts the new node there. This process ensures that the tree remains

balanced by filling levels from left to right.\*/

void insertBT(shared\_ptr<Node>& root, Creature c) {

if (!root) {

root = make\_shared<Node>(c);

return;

}

queue<shared\_ptr<Node>> q;

q.push(root);

while (!q.empty()) {

shared\_ptr<Node> current = q.front();

q.pop();

if (!current->left) {

current->left = make\_shared<Node>(c);

return;

} else {

q.push(current->left);

}

if (!current->right) {

current->right = make\_shared<Node>(c);

return;

} else {

q.push(current->right);

}

}

}

/\*The addCreatureToMaxHeap function adds the power of a Creature to a max-heap (priority queue).

It pushes the creature’s power value into the heap, ensuring that the highest values are

always at the top of the heap, which is characteristic of a max-heap.\*/

void addCreatureToMaxHeap(priority\_queue<int>& maxHeap, Creature c) {

maxHeap.push(c.power);

}

/\*The addCreatureToMinHeap function adds the power of a Creature to a min-heap.

It uses a priority queue with a custom comparator (greater<int>) to maintain the

smallest values at the top of the heap, typical of a min-heap structure.\*/

void addCreatureToMinHeap(priority\_queue<int, vector<int>, greater<int>>& minHeap, Creature c) {

minHeap.push(c.power);

}

/\*The heapify function maintains the heap property in a binary heap.

It ensures that the subtree rooted at a given index i satisfies the heap property.

If building a max-heap (isMaxHeap = true), the function compares the current node with

its left and right children and ensures the largest value is at the root, recursively fixing the heap if needed.

If building a min-heap (isMaxHeap = false), it ensures the smallest value is at

the root by comparing the current node with its children. The function then swaps

elements as needed and recursively ensures the heap property is preserved.\*/

void heapify(vector<int>& arr, int n, int i, bool isMaxHeap) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (isMaxHeap) {

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

} else {

if (left < n && arr[left] < arr[largest]) {

largest = left;

}

if (right < n && arr[right] < arr[largest]) {

largest = right;

}

}

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest, isMaxHeap);

}

}

/\*The buildHeap function constructs a heap (either max-heap or min-heap) from an unsorted array.

It starts from the last non-leaf node (index n / 2 - 1) and calls the heapify function for each node,

working its way up the tree to the root. This ensures that the entire array is rearranged into a valid heap.\*/

void buildHeap(vector<int>& arr, bool isMaxHeap) {

int n = arr.size();

for (int i = n / 2 - 1; i >= 0; --i) {

heapify(arr, n, i, isMaxHeap);

}

}

/\*The heapify\_creatures function is designed to build either a max-heap or a min-heap from a vector of creature powers.

It first calls make\_heap to create the heap structure, either a max-heap (for strongest creatures first) or a min-heap (for weakest creatures first),

based on the isMaxHeap flag. After creating the heap, the function uses a temporary priority queue to print the heap’s contents in order. For a max-heap,

it prints the strongest creatures (highest powers) first, and for a min-heap, it prints the weakest creatures (lowest powers) first,

in ascending order. Finally, the heap's structure is displayed in a readable format.\*/

void heapify\_creatures(vector<int>& powers, bool isMaxHeap) {

if (isMaxHeap) {

make\_heap(powers.begin(), powers.end()); // Max-Heap

} else {

make\_heap(powers.begin(), powers.end(), greater<int>()); // Min-Heap

}

// Display heap structure

cout << (isMaxHeap ? "Max-Heap (strongest creatures first): " : "Min-Heap (weakest creatures first): ");

if (isMaxHeap) {

// For Max-Heap

priority\_queue<int> tempHeap(powers.begin(), powers.end());

while (!tempHeap.empty()) {

cout << tempHeap.top() << " ";

tempHeap.pop();

}

} else {

// For Min-Heap

priority\_queue<int, vector<int>, greater<int>> tempHeap(powers.begin(), powers.end(), greater<int>());

while (!tempHeap.empty()) {

cout << tempHeap.top() << " ";

tempHeap.pop();

}

}

cout << endl;

}

/\*The displayCreatures function prints a list of all creatures in the provided vector, displaying each creature’s name and power.

It iterates through the vector and for each creature, outputs the name along with its power in a readable format.

This function helps to visually present the details of all creatures stored in the vector.\*/

void displayCreatures(const vector<Creature>& creatures) {

cout << "\nAll Creatures:" << endl;

for (const auto& creature : creatures) {

cout << "- " << creature.name << " (Power: " << creature.power << ")" << endl;

}

}

/\*The handleBST function provides an interactive menu for performing operations on a Binary Search Tree (BST).

It allows the user to insert, delete, search for creatures by their power level, and perform tree traversals (preorder, inorder, and postorder).

The function continuously presents options until the user selects the "Exit BST Menu" choice. Based on user input,

it performs the corresponding operation, such as inserting a creature, deleting a creature, searching for one, or displaying the tree in a specified order.

The function makes use of helper functions like insertBST, deleteBST, and traversal functions to manipulate and display the BST structure.\*/

void handleBST(shared\_ptr<Node>& bstRoot) {

int choice;

bool bstRunning = true;

while (bstRunning) {

cout << "\n=== BST Menu ===" << endl;

cout << "1. Insert Creature" << endl;

cout << "2. Delete Creature" << endl;

cout << "3. Search Creature" << endl;

cout << "4. Preorder Traversal" << endl;

cout << "5. Inorder Traversal" << endl;

cout << "6. Postorder Traversal" << endl;

cout << "7. Exit BST Menu" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string name;

int power;

cout << "Enter the creature's name: ";

cin.ignore();

getline(cin, name);

cout << "Enter the creature's power: ";

cin >> power;

bstRoot = insertBST(bstRoot, Creature(name, power));//this line responsible for inserting that is being put by the user

break;

}

case 2: {

int power;

cout << "Enter the power level to delete: ";

cin >> power;

bstRoot = deleteBST(bstRoot, power);//this line responsible for deleting a power input by the user

break;

}

case 3: {

int power;

cout << "Enter the power level to search for: ";

cin >> power;

shared\_ptr<Node> result = searchBST(bstRoot, power);// this line responsible for searching a power input by the user

if (result)

cout << "Found: " << result->creature.name << " (Power: " << result->creature.power << ")" << endl;

else

cout << "Creature not found!" << endl;

break;

}

case 4:

cout << "Preorder Traversal (BST): "; // will print the preorder traversal

preorder(bstRoot);

cout << endl;

break;

case 5:

cout << "Inorder Traversal (BST): ";//will print the inorder traversal

inorder(bstRoot);

cout << endl;

break;

case 6:

cout << "Postorder Traversal (BST): ";//will print the postorder traversal

postorder(bstRoot);

cout << endl;

break;

case 7:

bstRunning = false;

break;

default:

cout << "Invalid choice!" << endl;

}

}

}

/\*The handleBT function offers a menu-driven interface for managing a binary tree of creatures, allowing the user to insert,

delete, and search for creatures based on their power. Additionally, it provides options to traverse the tree in preorder,

inorder, or postorder. Each choice corresponds to a specific operation that manipulates or displays the structure of the

binary tree, making it easier to interact with and visualize the data.\*/

void handleBT(shared\_ptr<Node>& btRoot) {

int choice;

bool btRunning = true;

while (btRunning) {

cout << "\n=== BT Menu ===" << endl;

cout << "1. Insert Creature" << endl;

cout << "2. Delete Creature" << endl;

cout << "3. Search Creature" << endl;

cout << "4. Preorder Traversal" << endl;

cout << "5. Inorder Traversal" << endl;

cout << "6. Postorder Traversal" << endl;

cout << "7. Exit BT Menu" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string name;

int power;

cout << "Enter the creature's name: ";

cin.ignore();

getline(cin, name);

cout << "Enter the creature's power: ";

cin >> power;

insertBT(btRoot, Creature(name, power));

break;

}

case 2: {

int power;

cout << "Enter the power level to delete: ";

cin >> power;

btRoot = deleteBT(btRoot, power); // Delete creature from BT

break;

}

case 3: {

int power;

cout << "Enter the power level to search for: ";

cin >> power;

shared\_ptr<Node> result = searchBT(btRoot, power); // Search for creature in BT

if (result) {

cout << "Found: " << result->creature.name << " (Power: " << result->creature.power << ")" << endl;

} else {

cout << "Creature not found!" << endl;

}

break;

}

case 4:

cout << "Preorder Traversal (BT): ";

preorderBT(btRoot); // Use preorder traversal for BT

cout << endl;

break;

case 5:

cout << "Inorder Traversal (BT): ";

inorderBT(btRoot); // Use inorder traversal for BT

cout << endl;

break;

case 6:

cout << "Postorder Traversal (BT): ";

postorderBT(btRoot); // Use postorder traversal for BT

cout << endl;

break;

case 7:

btRunning = false;

break;

default:

cout << "Invalid choice!" << endl;

}

}

}

/\*The handleHeap function allows users to interact with both Max-Heap and Min-Heap data structures through a menu-driven interface.

The options include inserting creatures with specified power values into either heap, displaying the contents of both heaps,

and performing a heapify operation on a list of creature powers to reorganize them into proper heap structures.

The heapify process builds a Max-Heap and a Min-Heap from the current list of powers, ensuring that they maintain

the heap property, with the highest power at the top of the Max-Heap and the lowest power at the top of the Min-Heap.

This function facilitates heap management, including addition, display, and structural reorganization based on user input.\*/

void handleHeap(priority\_queue<int>& maxHeap,

priority\_queue<int, vector<int>, greater<int>>& minHeap,

vector<int>& powers,

vector<Creature>& creatures) {

int choice;

bool heapRunning = true;

while (heapRunning) {

cout << "\n=== Heap Menu ===" << endl;

cout << "1. Insert Creature to Max-Heap" << endl;

cout << "2. Insert Creature to Min-Heap" << endl;

cout << "3. Display Max-Heap" << endl;

cout << "4. Display Min-Heap" << endl;

cout << "5. Display Heapify" << endl;

cout << "6. Exit Heap Menu" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string name;

int power;

cout << "Enter the creature's name: ";

cin.ignore();

getline(cin, name);

cout << "Enter the creature's power: ";

cin >> power;

addCreatureToMaxHeap(maxHeap, Creature(name, power));

powers.push\_back(power); // Add power to the powers vector

creatures.push\_back(Creature(name, power)); // Also add creature to the creatures list

break;

}

case 2: {

string name;

int power;

cout << "Enter the creature's name: ";

cin.ignore();

getline(cin, name);

cout << "Enter the creature's power: ";

cin >> power;

addCreatureToMinHeap(minHeap, Creature(name, power));

powers.push\_back(power); // Add power to the powers vector

creatures.push\_back(Creature(name, power)); // Also add creature to the creatures list

break;

}

case 3: {

cout << "Max-Heap: "; // this is for printing the max-heap means from higher to lower

priority\_queue<int> tempMaxHeap = maxHeap;

while (!tempMaxHeap.empty()) {

cout << tempMaxHeap.top() << " ";

tempMaxHeap.pop();

}

cout << endl;

break;

}

case 4: {

cout << "Min-Heap: "; // this is for printing the min-heap means from lower to higher

priority\_queue<int, vector<int>, greater<int>> tempMinHeap = minHeap;

while (!tempMinHeap.empty()) {

cout << tempMinHeap.top() << " ";

tempMinHeap.pop();

}

cout << endl;

break;

}

case 5: {

cout << "Heapify: ";

if (powers.empty()) {

cout << "No creatures to heapify." << endl;

} else {

// Heapify for Max-Heap

cout << "Max-Heap of Creature Powers: ";

buildHeap(powers, true); // Max-Heap

for (int p : powers) {

cout << p << " ";

}

cout << endl;

// Heapify for Min-Heap

cout << "Min-Heap of Creature Powers: ";

buildHeap(powers, false); // Min-Heap

for (int p : powers) {

cout << p << " ";

}

cout << endl;

}

break;

}

case 6:

heapRunning = false;

break;

default:

cout << "Invalid choice!" << endl;

}

}

}

// Main function to start the Creature Game

/\*The playGame function is designed to create an interactive experience where users can manage a collection of creatures,

each with a name and power level, by interacting with various data structures. The game begins by prompting

the user to input at least five creatures, storing them in a Binary Search Tree (BST), Binary Tree (BT), Max-Heap,

and Min-Heap. Once the creatures are entered, the player can select options from a main menu to interact with these

data structures, such as viewing or modifying the BST, BT, or heaps. The loop continues until the player opts to exit the game.\*/

void playGame() {

shared\_ptr<Node> bstRoot = nullptr;

shared\_ptr<Node> btRoot = nullptr;

priority\_queue<int> maxHeap;

priority\_queue<int, vector<int>, greater<int>> minHeap;

vector<int> powers;

vector<Creature> creatures;

cout << "Welcome to the Creature Game!" << endl;

cout << "You need to add at least 5 creatures to start.\n";

// Input at least 5 creatures

while (creatures.size() < 5) {

string name;

int power;

cout << "Enter the creature's name: ";

cin >> name;

cout << "Enter the creature's power level: ";

cin >> power;

Creature newCreature(name, power);

creatures.push\_back(newCreature);

bstRoot = insertBST(bstRoot, newCreature); // BST insertion

insertBT(btRoot, newCreature); // BT insertion

addCreatureToMaxHeap(maxHeap, newCreature); // Max-Heap insertion

addCreatureToMinHeap(minHeap, newCreature); // Min-Heap insertion

powers.push\_back(power); // Add power to the powers vector

}

displayCreatures(creatures);

bool gameRunning = true;

while (gameRunning) {

cout << "\n=== Main Menu ===" << endl;

cout << "1. Work with BST" << endl;

cout << "2. Work with BT" << endl;

cout << "3. Work with Heaps" << endl;

cout << "4. Exit" << endl;

cout << "Enter your choice: ";

int choice;

cin >> choice;

switch (choice) {

case 1:

handleBST(bstRoot);//this will output the bst program which is it call the handleBST

break;

case 2:

handleBT(btRoot); //this will output the bt program which is it call the handleBT

break;

case 3:

handleHeap(maxHeap, minHeap, powers, creatures); // Pass powers and creatures

break;

case 4:

gameRunning = false;//if you choice the number it means you ended the program

break;

default:

cout << "Invalid choice!" << endl;//if you input none in the choices, this means it is invalid

}

}

}

/\*the main function is the entry point of the program. It is where the execution starts.

You typically initialize variables, handle input/output, and call other functions within the main function.\*/

int main() {

playGame();//we call here the playGame function because this is where the game will start

cout<<"Thank you mwa!";

return 0;// responsible to run the program smoothly

}