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>

#include <vector>

#include <memory>

#include <queue>

#include <algorithm>

#include <string>

using namespace std;

// Creature structure

struct Creature {

string name;

int power;

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

};

// Node structure for Binary Tree and Binary Search Tree

struct Node {

Creature creature;

shared\_ptr<Node> left;

shared\_ptr<Node> right;

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

};

// BST insertion

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;

}

// BST search

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);

}

// BST deletion

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;

}

// Preorder Traversal

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

if (!root) return;

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

preorder(root->left);

preorder(root->right);

}

// Inorder Traversal

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

if (!root) return;

inorder(root->left);

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

inorder(root->right);

}

// Postorder Traversal

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

if (!root) return;

postorder(root->left);

postorder(root->right);

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

}

//function of bt

// Delete node from Binary Tree (BT)

// Forward declaration of deleteDeepestNode

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

// Function to delete a node in Binary 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;

}

// Helper function to delete the deepest node

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);

}

}

// Search node in Binary Tree (BT)

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

}

// Preorder Traversal for Binary Tree (BT)

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);

}

// Inorder Traversal for Binary Tree (BT)

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

if (!root) return;

inorder(root->left);

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

inorder(root->right);

}

// Postorder Traversal for Binary Tree (BT)

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

if (!root) return;

postorderBT(root->left);

postorderBT(root->right);

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

}

// Binary Tree (BT) insertion

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);

}

}

}

// Max-Heap and Min-Heap insertion

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

maxHeap.push(c.power);

}

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

minHeap.push(c.power);

}

// Heapify function to maintain heap property for integer arrays

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);

}

}

// Function to build a Max-Heap or Min-Heap from an unsorted array

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);

}

}

//need here a function for heapify

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;

}

// Display all creatures

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

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

for (const auto& creature : creatures) {

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

}

}

// Function to handle BST operations

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));

break;

}

case 2: {

int power;

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

cin >> power;

bstRoot = deleteBST(bstRoot, power);

break;

}

case 3: {

int power;

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

cin >> power;

shared\_ptr<Node> result = searchBST(bstRoot, power);

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): ";

preorder(bstRoot);

cout << endl;

break;

case 5:

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

inorder(bstRoot);

cout << endl;

break;

case 6:

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

postorder(bstRoot);

cout << endl;

break;

case 7:

bstRunning = false;

break;

default:

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

}

}

}

// Function to handle BT operations

// Handle BT operations

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;

}

}

}

// Function to handle Heap operations

// Function to handle Heap operations

// Corrected function signature for handleHeap

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: ";

priority\_queue<int> tempMaxHeap = maxHeap;

while (!tempMaxHeap.empty()) {

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

tempMaxHeap.pop();

}

cout << endl;

break;

}

case 4: {

cout << "Min-Heap: ";

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()) { // This checks if the 'powers' vector is empty

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

} else {

cout << "Heapifying Creatures..." << endl;

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

heapify\_creatures(powers, true); // Max-Heap

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

heapify\_creatures(powers, false); // Min-Heap

}

break;

}

case 6:

heapRunning = false;

break;

default:

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

}

}

}

// Main function to start the Creature Game

// Corrected call in playGame

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);

break;

case 2:

handleBT(btRoot);

break;

case 3:

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

break;

case 4:

gameRunning = false;

break;

default:

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

}

}

}

int main() {

playGame();

cout<<"Thank you mwa!";

return 0;

}