| **Activity No. 9.1** | |
| --- | --- |
| **Tree ADT** | |
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed:** 11/13/24 |
| **Section:** CPE21S4 | **Date Submitted:** |
| **Name(s):**  Masangkay, Frederick D.  Bona, Andrei Nycole So.  Roallos, Jean Gabriel Vincent G.  Santos, Andrei R.  Zolina, Anna Marie L. | **Instructor:** Professor Maria Rizette Sayo |
| **A. Output(s) and Observation(s)** | |
| **Task 1: Create code in C++ that will create a tree as shown in the figure above. Use linked lists as the internal**  **representation of this tree. Indicate your code screenshot and comments in table 9-1.**  **SOURCE CODE:**  #include <iostream>  using namespace std;  // Definition of a node in the general tree  struct TreeNode {  char data;  TreeNode\* firstChild;  TreeNode\* nextSibling;  TreeNode(char value) : data(value), firstChild(nullptr), nextSibling(nullptr) {}  };  // Class to represent the general tree  class GeneralTree {  public:  GeneralTree() : root(nullptr) {}  // Function to set the root of the tree  void setRoot(char value) {  root = new TreeNode(value);  }  // Function to add a child to a given parent  void addChild(char parentValue, char childValue) {  TreeNode\* parent = findNode(root, parentValue);  if (parent) {  TreeNode\* child = new TreeNode(childValue);  if (!parent->firstChild) {  parent->firstChild = child; // Add as first child  } else {  TreeNode\* sibling = parent->firstChild;  while (sibling->nextSibling) {  sibling = sibling->nextSibling; // Traverse to the last sibling  }  sibling->nextSibling = child; // Add as next sibling  }  } else {  cout << "Parent not found!" << endl;  }  }  // Function to perform pre-order traversal  void preOrder() {  preOrderRec(root);  cout << endl;  }  private:  TreeNode\* root;  // Helper function to find a node with a specific value  TreeNode\* findNode(TreeNode\* node, char value) {  if (!node) return nullptr;  if (node->data == value) return node;  TreeNode\* foundNode = findNode(node->firstChild, value);  return foundNode ? foundNode : findNode(node->nextSibling, value);  }  // Helper function for pre-order traversal  void preOrderRec(TreeNode\* node) {  if (node) {  cout << node->data << " ";  preOrderRec(node->firstChild);  preOrderRec(node->nextSibling);  }  }  };  // Main function to demonstrate the general tree  int main() {  GeneralTree tree;  tree.setRoot('A'); // Setting 'A' as the root  // Adding children  char children[][2] = {  {'A', 'B'}, {'A', 'C'}, {'A', 'D'},  {'A', 'E'}, {'A', 'F'}, {'A', 'G'},  {'D', 'H'}, {'E', 'I'}, {'E', 'J'},  {'F', 'K'}, {'F', 'L'}, {'F', 'M'},  {'G', 'N'}, {'J', 'P'}, {'J', 'Q'}  };  for (const auto& pair : children) {  tree.addChild(pair[0], pair[1]);  }  // Pre-order traversal of the tree  cout << "Pre-order Traversal: ";  tree.preOrder();  return 0;  }  **Table 9-1. General Tree**  **Task 2: Complete the following table:**   | **NODE** | **HEIGHT** | **DEPTH** | | --- | --- | --- | | **A** | 4 | 0 | | **B** | 3 | 1 | | **C** | 2 | 1 | | **D** | 2 | 2 | | **E** | 1 | 2 | | **F** | 1 | 2 | | **G** | 1 | 2 | | **H** | 1 | 3 | | **I** | 0 | 3 | | **J** | 0 | 3 | | **K** | 0 | 3 | | **L** | 0 | 3 | | **M** | 0 | 3 | | **N** | 0 | 3 | | **P** | 0 | 4 | | **Q** | 0 | 4 |   **Table 9-2. Completed Table**  **Task 3: After implementing the code for above, answer the following:**  **3.1 Given the tree diagram, find the result of the pre-order, post-order and in-order traversal strategies by hand.**  **Include this as table 9-3 in section 6.**   | **Pre-order** | A B C D H E I J P Q F K L M G N | | --- | --- | | **Post-order** | B C H D I P Q J E K L M F N G A | | **In-order** | A B C D E F G H I J K L M N P Q |   **Table 9-3. Traversal Strategies**  **Create a function for pre-order, post-order and in-order traversal. Make sure that each function displays an**  **output into the console. Once you have the output, create and fill table 9-4 in section 6 so that it contains the**  **screenshot of the function, screenshot of the output and your observations. Your observations consist of a**  **comparison between output in #1 and the output of your functions in #2.**  #include <iostream>  using namespace std;  // Definition of a node in the general tree  struct TreeNode {  char data;  TreeNode\* firstChild;  TreeNode\* nextSibling;  TreeNode(char value) : data(value), firstChild(nullptr), nextSibling(nullptr) {}  };  // Class to represent the general tree  class GeneralTree {  public:  GeneralTree() : root(nullptr) {}  // Function to set the root of the tree  void setRoot(char value) {  root = new TreeNode(value);  }  // Function to add a child to a given parent  void addChild(char parentValue, char childValue) {  TreeNode\* parent = findNode(root, parentValue);  if (parent) {  TreeNode\* child = new TreeNode(childValue);  if (!parent->firstChild) {  parent->firstChild = child; // Add as first child  } else {  TreeNode\* sibling = parent->firstChild;  while (sibling->nextSibling) {  sibling = sibling->nextSibling; // Traverse to the last sibling  }  sibling->nextSibling = child; // Add as next sibling  }  } else {  cout << "Parent not found!" << endl;  }  }  // Function to perform pre-order traversal  void preOrder() {  cout << "Pre-order Traversal: ";  preOrderRec(root);  cout << endl;  }  // Function to perform post-order traversal  void postOrder() {  cout << "Post-order Traversal: ";  postOrderRec(root);  cout << endl;  }  // Function to perform in-order traversal  void inOrder() {  cout << "In-order Traversal: ";  inOrderRec(root);  cout << endl;  }  private:  TreeNode\* root;  // Helper function to find a node with a specific value  TreeNode\* findNode(TreeNode\* node, char value) {  if (!node) return nullptr;  if (node->data == value) return node;  TreeNode\* foundNode = findNode(node->firstChild, value);  return foundNode ? foundNode : findNode(node->nextSibling, value);  }  // Helper function for pre-order traversal  void preOrderRec(TreeNode\* node) {  if (node) {  cout << node->data << " "; // Visit the current node  preOrderRec(node->firstChild); // Visit first child  preOrderRec(node->nextSibling); // Visit next sibling  }  }  // Helper function for post-order traversal  void postOrderRec(TreeNode\* node) {  if (node) {  postOrderRec(node->firstChild); // Visit first child  postOrderRec(node->nextSibling); // Visit next sibling  cout << node->data << " "; // Visit the current node  }  }  // Helper function for in-order traversal  void inOrderRec(TreeNode\* node) {  if (node) {  inOrderRec(node->firstChild); // Visit first child  cout << node->data << " "; // Visit current node  inOrderRec(node->nextSibling); // Visit next sibling  }  }  };  // Main function to demonstrate the general tree  int main() {  GeneralTree tree;  tree.setRoot('A'); // Setting 'A' as the root  // Adding children  char children[][2] = {  {'A', 'B'}, {'A', 'C'}, {'A', 'D'},  {'A', 'E'}, {'A', 'F'}, {'A', 'G'},  {'D', 'H'}, {'E', 'I'}, {'E', 'J'},  {'F', 'K'}, {'F', 'L'}, {'F', 'M'},  {'G', 'N'}, {'J', 'P'}, {'J', 'Q'}  };  for (const auto& pair : children) {  tree.addChild(pair[0], pair[1]);  }  // Perform traversals  tree.preOrder(); // Output pre-order traversal  tree.postOrder(); // Output post-order traversal  tree.inOrder(); // Output in-order traversal  return 0;  }           | Observation | In this code, the pre-order traversal visits each node before its children, giving A B C D H E I J P Q F K L M G N. The post-order traversal visits all children before the node itself, producing H D I P Q J E K L M F N G C B A. The in-order function is unconventional for a general tree, as it visits each node’s first child, the node, then siblings, resulting in the same output as post-order. This in-order approach fits this structure but isn’t standard for non-binary trees. | | --- | --- | | |
| **B. Answers to Supplementary Activity** | |
| Step 1:  Step 2:   In-order Traversal  Pre-order Traversal    Post-order Traversal  Step 3:  **In-order Traversal**    In in-order traversal, the nodes are visited in the order of left subtree, root, and right subtree. This traversal is useful for printing nodes in sorted order in a binary search tree (BST).  **Pre-order Traversal**      In pre-order traversal, the root is visited first, followed by the left subtree and then the right subtree. It is often used for copying or serializing a tree structure.  **Post-order Traversal**      In post-order traversal, the left and right subtrees are visited before the root node. This traversal is useful for operations like tree deletion or postfix notation in expressions. | |
| **C. Conclusion & Lessons Learned** | |
| In conclusion, this laboratory introduced us to tree data structures and their importance in programming. Trees comprise connected nodes, forming a hierarchy useful for organizing data. We learned about binary trees, where each node has up to two children, and explored different types like full, complete, and balanced binary trees.  Coding these structures and understanding tree traversals was challenging, but it helped us get a better grasp of how they work. Though some parts were tough, We see it as part of the learning process and be prepared to work with data structures in future projects. | |
| **D. Assessment Rubric** | |
|  | |
| **E. External References** | |
|  | |