Data Structures and Algorithms: Assignment 3 Solutions

Muhammad Rizwan Shafiq

SP24-BCS-069

Introduction

This document presents the solutions for Assignment 3 of the Data Structures and Algorithms course. The assignment involves implementing eight data structures and algorithms—Binary Search Tree, AVL Tree, Heap Tree, Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's Algorithm, Prim's Algorithm, and Kruskal's Algorithm—applied to real-world scenarios. Each section below describes the scenario, explains the working of the algorithm or data structure, and provides key code snippets from the C++ implementations, along with diagrams illustrating the structures.

1 Task 1: Binary Search Tree

1.1 Scenario

Phone Directory System: A phone directory stores names and phone numbers. A Binary Search Tree (BST) is used to store contacts with names as keys and phone numbers as values, enabling quick searches based on names.

1.2 Working

The BST organizes contacts alphabetically by name. Each node contains a name and phone number, with left and right pointers to child nodes. Key operations include: - **Insertion**: New contacts are inserted by comparing the name with the root and recursively traversing left (if name is lexicographically smaller) or right (if larger). - **Search**: To find a contact (e.g., "Ali"), the algorithm compares the target name with the root and moves left or right based on alphabetical order. - **Inorder Traversal**: Produces a sorted list of contacts, useful for displaying the directory in alphabetical order.

1.3 Key Code

```
struct Node {
  string name;
  string phone;
  Node *left, *right;
```

```
  Node(string n, string p) : name(n), phone(p), left(nullptr), right
      (nullptr) {}
};

Node* insert(Node* root, string name, string phone) {
  if (!root) return new Node(name, phone);
  if (name < root->name)
    root->left = insert(root->left, name, phone);
  else
    root->right = insert(root->right, name, phone);
  return root;
}

void inorder(Node* root) {
  if (root) {
    inorder(root->left);
    cout << root->name << " : " << root->phone << endl;
    inorder(root->right);
  }
}
```

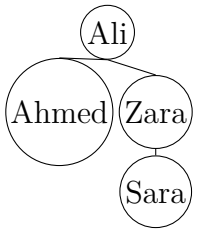

Figure 1: BST for Phone Directory with sample contacts (names only).

2 Task 2: AVL Tree

2.1 Scenario

Student Records Management System: An AVL Tree maintains student records using roll numbers as keys and names as values. It ensures balanced storage for efficient insertion, deletion, and retrieval as students enroll or drop out.

2.2 Working

The AVL Tree is a self-balancing BST where the height difference between left and right subtrees (balance factor) is at most 1. Operations include: - **Insertion**: A new student record is inserted based on roll number. After insertion, the balance factor is checked, and rotations (LL, RR, LR, RL) are applied if needed to restore balance. - **Balancing**: Rotations adjust the tree structure to maintain height balance, ensuring O(log n) time complexity. - **Inorder Traversal**: Displays records sorted by roll number.

2.3 Key Code

```
struct node {
  int rollNo;
  char name[50];
  node *left, *right;
  int ht;
};

int height(node *T) {
  int lh, rh;
  if (T == NULL) return 0;
  lh = T->left ? 1 + T->left->ht : 0;
  rh = T->right ? 1 + T->right->ht : 0;
  return lh > rh ? lh : rh;
}

int balanceFactor(node* T) {
  return T == NULL ? 0 : height(T->left) - height(T->right);
}

node* rotateRight(node* x) {
  node* y = x->left;
  x->left = y->right;
  y->right = x;
  x->ht = height(x);
  y->ht = height(y);
  return y;
}

node* rotateLeft(node* x) {
  node* y = x->right;
  x->right = y->left;
  y->left = x;
  x->ht = height(x);
  y->ht = height(y);
  return y;
}

node* insert(node* T, int roll, const char* name) {
  if (T == NULL) {
    T = new node;
    T->rollNo = roll;
    strcpy(T->name, name);
    T->left = T->right = NULL;
    T->ht = 0;
    return T;
  }
  if (roll < T->rollNo) {
    T->left = insert(T->left, roll, name);
    if (balanceFactor(T) == 2) {
```

```
      if (roll < T->left->rollNo)
        T = rotateRight(T);
      else {
        T->left = rotateLeft(T->left);
        T = rotateRight(T);
      }
    }
  } else if (roll > T->rollNo) {
    T->right = insert(T->right, roll, name);
    if (balanceFactor(T) == -2) {
      if (roll > T->right->rollNo)
        T = rotateLeft(T);
      else {
        T->right = rotateRight(T->right);
        T = rotateLeft(T);
      }
    }
  }
  T->ht = height(T);
  return T;
}
```

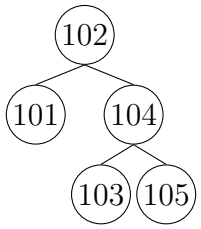

Figure 2: AVL Tree for Student Records with roll numbers.

3 Task 3: Heap Tree

3.1 Scenario

Print Job Manager: A max heap manages a printer queue where each job has a priority (higher number = higher priority). Jobs are processed in order of priority, and the queue can be converted to a min heap for demonstration.

3.2 Working

A max heap ensures the highest-priority job is at the root. Key operations include: - **Insertion**: A new job is added at the end and shifted up to maintain the max heap property. - **Process Job**: The root (highest priority) is removed, and the last element is moved to the root, followed by shifting down to restore the heap. - **Min Heap Conversion**: For demonstration, the heap can be reorganized to prioritize the lowest-priority job.

3.3 Key Code

```
int H[50];
int heapSize = -1;

int parent(int i) { return (i - 1) / 2; }
int leftChild(int i) { return (2 * i) + 1; }
int rightChild(int i) { return (2 * i) + 2; }

void shiftUp(int i) {
  while (i > 0 && H[parent(i)] < H[i]) {
    swap(H[parent(i)], H[i]);
    i = parent(i);
  }
}

void insertJob(int priority) {
  heapSize++;
  H[heapSize] = priority;
  shiftUp(heapSize);
}

void shiftDown(int i) {
  int maxIndex = i;
  int l = leftChild(i);
  if (l <= heapSize && H[l] > H[maxIndex]) maxIndex = l;
  int r = rightChild(i);
  if (r <= heapSize && H[r] > H[maxIndex]) maxIndex = r;
  if (i != maxIndex) {
    swap(H[i], H[maxIndex]);
    shiftDown(maxIndex);
  }
}

int processJob() {
  int result = H[0];
  H[0] = H[heapSize];
  heapSize--;
  shiftDown(0);
  return result;
}
```

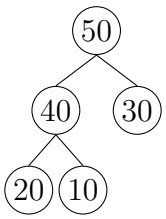

Figure 3: Max Heap for Print Job Priorities.

4 Task 4: Breadth-First Search (BFS) Algorithm

4.1 Scenario

Social Network Friend Suggestion: In a social network, BFS finds friends-of-friends (users two hops away) by exploring the network level by level, suggesting potential connections for a given user.

4.2 Working

The social network is modeled as an undirected graph with users as vertices and friendships as edges. BFS starts at a user, explores direct friends (Level 1), then their friends (Level 2), excluding direct friends and the user themselves. A queue tracks nodes to visit, and a visited array prevents revisiting. The algorithm tracks levels to identify users exactly two hops away.

4.3 Key Code

```
1  struct Vertex {
2    char label;
3    bool visited;
4  };
5
6  Vertex* lstVertices[MAX];
7  int adjMatrix[MAX][MAX];
8  int vertexCount = 0;
9
10 void addVertex(char label) {
11   Vertex* vertex = new Vertex;
12   vertex->label = label;
13   vertex->visited = false;
14   lstVertices[vertexCount++] = vertex;
15 }
16 void addEdge(int start, int end) {
17   adjMatrix[start][end] = 1;
18   adjMatrix[end][start] = 1;
19 }
20 void displayVertex(int vertexIndex) {
21   cout << lstVertices[vertexIndex]->label << " ";
22 }
23 int getAdjUnvisitedVertex(int vertexIndex) {
24   for (int i = 0; i < vertexCount; i++) {
25     if (adjMatrix[vertexIndex][i] == 1 && !lstVertices[i]->visited)
         {
26       return i;
27     }
28   }
29   return -1;
30 }
31
```

```
void suggestFriendsBFS(int startVertex) {
  queue<int> q;
  vector<bool> visited(vertexCount, false);
  unordered_set<int> directFriends;
  unordered_set<int> suggestions;
  int level = 0;
  int nodesInCurrentLevel = 1;
  int nodesInNextLevel = 0;
  visited[startVertex] = true;
  q.push(startVertex);
  while (!q.empty() && level < 2) {
    int currentVertex = q.front();
    q.pop();
    nodesInCurrentLevel--;
    int unvisitedVertex;
    while ((unvisitedVertex = getAdjUnvisitedVertex(currentVertex))
        != -1) {
      lstVertices[unvisitedVertex]->visited = true;
      q.push(unvisitedVertex);
      nodesInNextLevel++;
      if (level == 0) {
        directFriends.insert(unvisitedVertex);
      } else if (level == 1 && unvisitedVertex != startVertex &&
                 directFriends.find(unvisitedVertex) ==
                    directFriends.end()) {
        suggestions.insert(unvisitedVertex);
      }
    }
    if (nodesInCurrentLevel == 0) {
      level++;
      nodesInCurrentLevel = nodesInNextLevel;
      nodesInNextLevel = 0;
    }
  }
  cout << "Friend suggestions for user " << lstVertices[startVertex
     ]->label << ": ";
  if (suggestions.empty()) {
    cout << "No suggestions found." << endl;
  } else {
    for (int s : suggestions) {
      displayVertex(s);
    }
    cout << endl;
  }
  for (int i = 0; i < vertexCount; i++) {
    lstVertices[i]->visited = false;
  }
}
```

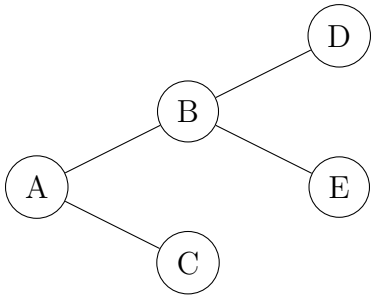

Figure 4: Social Network Graph (A's friends: B, C; suggestions: D, E).

5 Task 5: Depth-First Search (DFS) Algorithm

5.1 Scenario

Social Media Connection Explorer: A social media network is represented as a graph where vertices are users and edges are mutual follow relationships. DFS explores the network starting from a given user (e.g., 'S') to list all connected users in the order they are visited.

5.2 Working

The network is modeled as an undirected graph with an adjacency matrix. DFS uses a stack to explore users, starting from user 'S'. It visits a user, marks it as visited, and pushes it to the stack. It then explores unvisited followers, pushing them to the stack, and backtracks by popping when no unvisited followers remain. The result is a sequence of users in the order they were visited.

5.3 Key Code

```
#include <stack>

struct Vertex {
  char label;
  bool visited;
};

std::stack<int> stk;
struct Vertex* lstVertices[MAX];
int adjMatrix[MAX][MAX];
int vertexCount = 0;

void addVertex(char label) {
  struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct
     Vertex));
  vertex->label = label;
  vertex->visited = false;
  lstVertices[vertexCount++] = vertex;
}
void addEdge(int start, int end) {
```

```
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1;
}
void displayVertex(int vertexIndex) {
  cout << lstVertices[vertexIndex]->label;
}
int getAdjUnvisitedVertex(int vertexIndex) {
  for (int i = 0; i < vertexCount; i++) {
    if (adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited ==
        false) {
      return i;
    }
  }
  return -1;
}

void depthFirstSearch() {
  lstVertices[0]->visited = true;
  displayVertex(0);
  stk.push(0);
  while (!stk.empty()) {
    int unvisitedVertex = getAdjUnvisitedVertex(stk.top());
    if (unvisitedVertex == -1) {
      stk.pop();
    } else {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      stk.push(unvisitedVertex);
    }
  }
  for (int i = 0; i < vertexCount; i++) {
    lstVertices[i]->visited = false;
  }
}
```

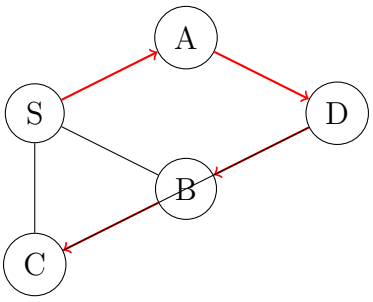

Figure 5: DFS traversal (S → A → D → B → C) in the social media network.

6 Task 6: Dijkstra's Algorithm

6.1 Scenario

GPS Navigation System: Dijkstra's algorithm finds the shortest path between locations on a map, represented as a weighted graph where edges are distances.

6.2 Working

The algorithm maintains a distance array and a visited array. It selects the unvisited node with the minimum distance, updates distances to its neighbors, and repeats until all nodes are processed. The result is the shortest path from the source to all other locations.

6.3 Key Code

```
#define V 5

int minDistance(int dist[], bool visited[]) {
  int min = INT_MAX, min_index;
  for (int v = 0; v < V; v++) {
    if (!visited[v] && dist[v] <= min) {
      min = dist[v];
      min_index = v;
    }
  }
  return min_index;
}

void dijkstra(int graph[V][V], int src) {
  int dist[V];
  bool visited[V];
  for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    visited[i] = false;
  }
  dist[src] = 0;
  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited);
    visited[u] = true;
    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
          dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }
}
```

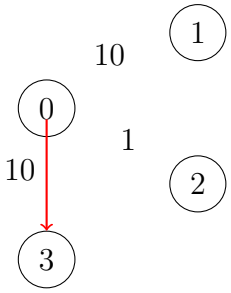

Figure 6: Graph with shortest path from 0 to 3 (weight=10).

7 Task 7: Prim's Algorithm

7.1 Scenario

Laying Fiber Optic Cables: Prim's algorithm connects multiple cities with the minimum total cable length, forming a Minimum Spanning Tree (MST).

7.2 Working

Prim's algorithm starts at a city and greedily selects the smallest-weight edge to an unvisited city, adding it to the MST. It uses a key array to track minimum edge weights and a parent array to build the tree structure.

7.3 Key Code

```
#define V 5

int minKey(int key[], bool mstSet[]) {
  int min = INT_MAX, min_index;
  for (int v = 0; v < V; v++) {
    if (!mstSet[v] && key[v] < min) {
      min = key[v];
      min_index = v;
    }
  }
  return min_index;
}

void primMST(int graph[V][V]) {
  int parent[V];
  int key[V];
  bool mstSet[V];
  for (int i = 0; i < V; i++) {
    key[i] = INT_MAX;
    mstSet[i] = false;
  }
  key[0] = 0;
  parent[0] = -1;
  for (int count = 0; count < V - 1; count++) {
```

```
    int u = minKey(key, mstSet);
    mstSet[u] = true;
    for (int v = 0; v < V; v++) {
      if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
        parent[v] = u;
        key[v] = graph[u][v];
      }
    }
  }
}
```

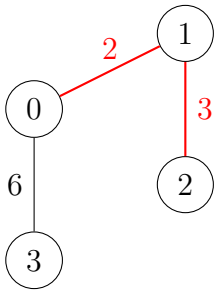

Figure 7: MST from Prim's Algorithm (edges 0–1, 1–2).

8 Task 8: Kruskal's Algorithm

8.1 Scenario

Connecting Rural Villages: Kruskal's algorithm connects villages with roads at minimum total cost. Each village is a vertex, and potential roads are weighted edges with construction costs.

8.2 Working

Kruskal's algorithm constructs a Minimum Spanning Tree (MST) by: - Sorting all edges by weight (cost). - Iterating through sorted edges, adding each to the MST if it doesn't create a cycle. - Using a union-find data structure to detect cycles by checking if two villages are already connected. The result is a tree connecting all villages with minimum total road length.

8.3 Key Code

```
struct Edge {
    int src, dest, weight;
};
struct Subset {
    int parent, rank;
};
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
```

```
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}
void unionSets(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
void kruskal(vector<Edge>& edges, int V) {
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.weight < b.weight;
    });
    vector<Edge> result;
    Subset* subsets = new Subset[V];
    for (int i = 0; i < V; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }
    for (Edge e : edges) {
        int x = find(subsets, e.src);
        int y = find(subsets, e.dest);
        if (x != y) {
            result.push_back(e);
            unionSets(subsets, x, y);
        }
    }
    for (Edge e : result)
        cout << e.src << " - " << e.dest << "\t" << e.weight << "\n"
           ;
    delete[] subsets;
}
```

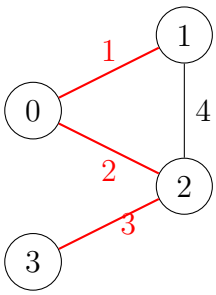

Figure 8: MST from Kruskal's Algorithm (edges 0–1, 0–2, 2–3).