DSA FINAL EXAM SOLUTION

Question 1: **[5+5=10 points]**

Construct a B-tree of order 3:

i. Inserting the following values in the given order.
50, 20, 75, 10, 60, 40, 90, 30, 80, 100, 15, 70, 85, 25, 65, 35, 95, and 5

ii. Locate 75 and 40 in the B-Tree and remove them one by one. Explain the deletion process and show how the B-Tree rebalances

Show all steps clearly.

Question 2: **[3+3+4 =10 points]**

Write code for implement the following functions

i. LinearProbingHashDuplicates such that it keeps duplicate keys in the table.
ii. ReturnValues Return any value associated with the given key.
iii. RemoveKey (Hard delete) all items in the table that have keys equal to the given key for delete()

SOLUTION

```
#include <iostream>

const int TABLE_SIZE = 10;

class HashTable {
private:
   // Entry structure: key and value
   struct Entry {
      int key;
      int value;
      bool isOccupied;

      Entry() : key(0), value(0), isOccupied(false) {}
   };

   Entry table[TABLE_SIZE];

   // Hash function
   int hash(int key) {
      return key % TABLE_SIZE;
   }

public:
   HashTable() {
      for (int i = 0; i < TABLE_SIZE; ++i) {
```

```
        table[i].isOccupied = false;
    }
}

// Function i: Insert key-value pair with linear probing allowing duplicates
void LinearProbingHashDuplicates(int key, int value) {
    int index = hash(key);
    while (table[index].isOccupied) {
        index = (index + 1) % TABLE_SIZE;
    }
    table[index].key = key;
    table[index].value = value;
    table[index].isOccupied = true;
}

// Function ii: Return all values associated with a given key
void ReturnValues(int key) {
    bool found = false;
    std::cout << "Values for key " << key << ": ";
    for (int i = 0; i < TABLE_SIZE; ++i) {
        if (table[i].isOccupied && table[i].key == key) {
            std::cout << table[i].value << " ";
            found = true;
        }
    }
    if (!found) {
        std::cout << "No values found";
    }
    std::cout << "\n";
}

// Function iii: Remove all entries with the given key
void RemoveKey(int key) {
    for (int i = 0; i < TABLE_SIZE; ++i) {
        if (table[i].isOccupied && table[i].key == key) {
            table[i].isOccupied = false;
        }
    }
}

// Utility function to display the hash table
void display() {
    for (int i = 0; i < TABLE_SIZE; ++i) {
        if (table[i].isOccupied) {
            std::cout << "Index " << i << ": (" << table[i].key << ", " << table[i].value << ")\n";
        } else {
            std::cout << "Index " << i << ": Empty\n";
```

```
        }
      }
    }
};

int main() {
    HashTable hashTable;

    // Insert some key-value pairs
    hashTable.LinearProbingHashDuplicates(1, 10);
    hashTable.LinearProbingHashDuplicates(11, 20);
    hashTable.LinearProbingHashDuplicates(1, 30);
    hashTable.LinearProbingHashDuplicates(21, 40);

    std::cout << "Hash Table After Insertions:\n";
    hashTable.display();

    // Retrieve values for key 1
    std::cout << "\n";
    hashTable.ReturnValues(1);

    // Remove key 1
    hashTable.RemoveKey(1);
    std::cout << "\nHash Table After Removing Key 1:\n";
    hashTable.display();

    return 0;
}
```

Question 3: **[5+5 = 10 points]**

i. Given the data of all roads in a city as a graph, write a function to determine whether all roads in the city are two-way (bidirectional) or if there are any one-way (directed) roads. Print Yes if all roads are bidirectional. Otherwise, print No. (Hint: Using an adjacency matrix).

ii. Implement a function using Depth-First Search (DFS) to check if there exists a path between two specific landmarks in the city.

SOLUTION

```
#include <iostream>
#include <vector>
#include <stack>

const int MAX_NODES = 100; // Maximum number of landmarks in the city

// Function i: Check if all roads are bidirectional using adjacency matrix
```

```
bool areAllRoadsBidirectional(int adjacencyMatrix[MAX_NODES][MAX_NODES], int numNodes) {
   for (int i = 0; i < numNodes; ++i) {
      for (int j = 0; j < numNodes; ++j) {
         if (adjacencyMatrix[i][j] != adjacencyMatrix[j][i]) {
            return false; // Found a one-way road
         }
      }
   }
   return true; // All roads are bidirectional
}

// Function ii: Check if a path exists between two landmarks using DFS
bool isPathDFS(int adjacencyMatrix[MAX_NODES][MAX_NODES], int numNodes, int start, int end)
{
   std::vector<bool> visited(numNodes, false);
   std::stack<int> stack;

   stack.push(start);

   while (!stack.empty()) {
      int current = stack.top();
      stack.pop();

      if (current == end) {
         return true; // Path found
      }

      if (!visited[current]) {
         visited[current] = true;
         for (int i = 0; i < numNodes; ++i) {
            if (adjacencyMatrix[current][i] && !visited[i]) {
               stack.push(i);
            }
         }
      }
   }
   return false; // No path found
}

int main() {
   int numNodes = 5;
   int adjacencyMatrix[MAX_NODES][MAX_NODES] = {
      {0, 1, 0, 0, 1},
      {1, 0, 1, 0, 0},
      {0, 1, 0, 1, 0},
      {0, 0, 1, 0, 1},
      {1, 0, 0, 1, 0}
```

```
    };

    // Check if all roads are bidirectional
    if (areAllRoadsBidirectional(adjacencyMatrix, numNodes)) {
        std::cout << "Yes, all roads are bidirectional.\n";
    } else {
        std::cout << "No, there are one-way roads.\n";
    }

    // Check if there is a path between two landmarks
    int start = 0, end = 3; // Landmarks to check path between
    if (isPathDFS(adjacencyMatrix, numNodes, start, end)) {
        std::cout << "There is a path between landmark " << start << " and landmark " << end <<
".\n";
    } else {
        std::cout << "No path exists between landmark " << start << " and landmark " << end << ".\n";
    }

    return 0;
}
```

Question 4: **[3+3+4 = 10 points]**

You are given a string with no spaces and you need to generate all possible sentences (combinations of words) that can be formed by inserting spaces into the input string such that every word is in the dictionary. The words must appear in the same order as in the input string.

For example
Input String: "catsanddog"
Dictionary: ["cat", "cats", "and", "sand", "dog"]

The valid sentences are:
"cat sand dog"
"cats and dog"

i. What will be the base case?
ii. Show the step-by-step working of the recursive function using a dry run. At each step, list the prefix being considered, indicate whether the prefix is valid (in the dictionary) and show the remaining string passed to the recursive call.
iii. Clearly explain where backtracking is being done.

SOLUTION

i. Base Case

The base case occurs when the input string becomes empty after processing valid prefixes. This signifies that all parts of the string have been segmented into valid words from the dictionary.

ii. Step-by-Step Dry Run

Input String: "catsanddog"
Dictionary: ["cat", "cats", "and", "sand", "dog"]

We use a recursive function wordBreak that considers all prefixes starting from the beginning of the string:

1. **Start with "catsanddog":**
    - **Prefix: "c" → Not in dictionary. Continue.**
    - **Prefix: "ca" → Not in dictionary. Continue.**
    - **Prefix: "cat" → Valid. Recur with remaining string "sanddog".**
2. **Recur with "sanddog":**
    - **Prefix: "s" → Not in dictionary. Continue.**
    - **Prefix: "sa" → Not in dictionary. Continue.**
    - **Prefix: "san" → Not in dictionary. Continue.**
    - **Prefix: "sand" → Valid. Recur with remaining string "dog".**
3. **Recur with "dog":**
    - **Prefix: "d" → Not in dictionary. Continue.**
    - **Prefix: "do" → Not in dictionary. Continue.**
    - **Prefix: "dog" → Valid. Recur with empty string "".**
4. **Recur with "" (base case):**
    - **The string is empty, so a valid sentence "cat sand dog" is formed.**

Backtrack to "catsanddog":

- **Prefix: "cats" → Valid. Recur with remaining string "anddog".**

5. **Recur with "anddog":**
    - **Prefix: "a" → Not in dictionary. Continue.**
    - **Prefix: "an" → Not in dictionary. Continue.**
    - **Prefix: "and" → Valid. Recur with remaining string "dog".**
6. **Recur with "dog" (already computed):**
    - **Forms the valid sentence "cats and dog".**

Valid Sentences:

1. **"cat sand dog"**
2. **"cats and dog"**

iii. Explanation of Backtracking

Backtracking happens when:

1. **A prefix is valid, but the remaining string cannot form valid sentences.**
2. **After processing one valid path, the algorithm returns to explore other potential prefixes for the string.**

For example:

- **After processing "cat sand dog", the function backtracks to "catsanddog" to consider "cats" as a prefix and generate "cats and dog".**

Question 5: **[5+5 = 10 points]**

i. You are given the following an array of integers, A of size N and two indices, st (start index) and ed (end index), such that $0 \leq st \leq ed < N$. Your task is to sort the subarray A[st] to A[ed] in ascending order using a priority queue (heap). The implementation should be in-place.
Write code to implement the above. Provide step-by-step explanation of your code with an example dry-run.

ii. Consider the given input text T = 000 000 0101 000 00010 and a pattern P = 010. Show your working to calculate the number of comparisons performed by the following string-matching algorithms. Your answer must clearly show how you obtained your answer.
   a) Brute Force algorithm
   b) KMP algorithm

SOLUTION

```
#include <iostream>
#include <queue>
#include <vector>

// Function to sort a subarray using a priority queue (heap)
void sortSubarrayInPlace(std::vector<int> &A, int st, int ed) {
  // Step 1: Create a min-heap (priority queue)
  std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;

  // Step 2: Push elements of the subarray into the heap
  for (int i = st; i <= ed; ++i) {
    minHeap.push(A[i]);
  }

  // Step 3: Replace elements in the subarray with sorted order from the heap
  for (int i = st; i <= ed; ++i) {
```

```
        A[i] = minHeap.top();
        minHeap.pop();
    }
}

int main() {
    // Example input array and indices
    std::vector<int> A = {10, 5, 8, 12, 15, 6, 3, 9};
    int st = 2, ed = 6;

    std::cout << "Original Array: ";
    for (int num : A) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Sort the subarray in-place
    sortSubarrayInPlace(A, st, ed);

    std::cout << "Array After Sorting Subarray [" << st << ", " << ed << "]: ";
    for (int num : A) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Step-by-Step Explanation:

1. **Input Array and Indices**:
   - You are given an array `A` and indices `st` (start) and `ed` (end).
   - Example: `A = {10, 5, 8, 12, 15, 6, 3, 9}`, `st = 2`, `ed = 6`.
2. **Using a Min-Heap**:
   - A min-heap is implemented using a priority queue with a custom comparator to maintain ascending order.
   - All elements in the subarray `A[st]` to `A[ed]` are pushed into the heap.
3. **Extracting Sorted Order**:
   - The priority queue ensures that the smallest element is at the top.
   - Extract elements one by one and replace them back into the subarray in ascending order.

Dry Run:

Input:

- Array: `{10, 5, 8, 12, 15, 6, 3, 9}`
- Indices: `st = 2`, `ed = 6`

Step-by-Step Process:

1. **Initial Subarray**:
   - Subarray: `{8, 12, 15, 6, 3}`.
2. **Push into Min-Heap**:
   - Elements pushed: `8, 12, 15, 6, 3`.
   - Heap after all pushes: `[3, 6, 15, 12, 8]`.
3. **Replace Subarray with Sorted Order**:
   - Extract and replace:
     - Replace `A[2]` with `3`.
     - Replace `A[3]` with `6`.
     - Replace `A[4]` with `8`.
     - Replace `A[5]` with `12`.
     - Replace `A[6]` with `15`.
4. **Final Array**:
   - `{10, 5, 3, 6, 8, 12, 15, 9}`.

ii)