HW7

Ronan Wallace

May 2019

1) Open Hashing

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Open Hashing.pdf**

2) Closed Hashing

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Closed Hashing.pdf**

3) Hash Table Algorithm

ALGORITHM checkDistinct(A[0...n])
    //Input: Array of values
    //Output: True if all values are distinct / False if not.
    hashTable ← initialize open hash table
    for i ← 0 **to** n-1 **do**
        if Array[i] % num. index position in hashTable **is not** empty **do**
            if currently stored value **is equal to** Array[i] **do**
                return False
            else do
                continue storing values from Array
    return True

4) Palindrome Algorithm

ALGORITHM findLongestSubsequenceLength(sequence)
    //Input: String sequence
    //Output: Length of longest palindrome subsequence.
    $n \leftarrow$ length of sequence
    allSubsequences $\leftarrow$ {"0": 0, "1": 01, "2": 012... "$n-1$": n}
    longestLength $\leftarrow 0$
    palindromeDict $\leftarrow$ {}
    subseqString $\leftarrow$ ""
    for i $\leftarrow 0$ **to** $n-1$ **do**
        subseqString $\leftarrow$ LCS(sequence, allSubsequence[i])
        if CheckPalindrome(subseqString) **is** True **do**
            palindromeDict $\leftarrow$ subseqString
    for i $\leftarrow 0$ **to** length of palindromeDict **do**
        if length of palindromeDict[str(i)] $>$ longestLength **do**
            longestLength $\leftarrow$ length of palindromeDict[str(i)]
    return longestLength

ALGORITHM LCS(A[1...m], B[1...n])
    //Input: Two strings to be compared
    //Output: Longest common subsequence.
    if A **or** B **is** empty **do**
        return ""
    if A[m] **is equal to** B[n] **do**
        return 1 + LCS(A[1...m-1], B[1...n-1])
    else do
        return max(LCS(A[1...m-1], B), LCS(A, B[1...n-1]))

ALGORITHM checkPalindrome(subsequence)
    //Input: Subsequence from original sequence
    //Output: True if palindrome / False if not.
    for i $\leftarrow 0$ **to** $\frac{n-1}{2}$ **do**
        for j $\leftarrow n-1$ **to** $\frac{n-1}{2}$ **do**
            if subsequence[i] $\neq$ subsequence[j] **do**
                return False
    return True

5) Prim's Algorithm

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Prim Algorithm.pdf**

6) Kruskal's Algorithm

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Kruskal Algorithm.pdf**

7) Dijkstra's Algorithm

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Dijkstra Algorithm.pdf**

8) Huffman Coding Tree

Picture(s) located in separate file (see .zip folder).
File Name: **HW7 - Huffman Coding Tree.pdf**

9) Decoding Algorithm

```
ALGORITHM decodeHuffman(huffmanTree, message[0...n])
    //Input: Huffman Tree and encoded message (array of 0's and 1's)
    //Output: Array of decoded characters (decoded message)
    decodedArray ← [ ]
    currentNode ← huffmanTree.root()
    for i ← 0 to n − 1 do
        if message[i] is 0 do
            if currentNode.leftChild() is not Null do
                currentNode ← currentNode.leftChild()
            else do
                Add currentNode to decodedArray
                currentNode ← huffmanTree.root()
        if message[i] is 1 do
            if currentNode.rightChild() is not Null do
                currentNode ← currentNode.rightChild()
            else do
                Add currentNode to decodedArray
                currentNode ← huffmanTree.root()
    return decodedArray
```