Binary Tree Properties and Optimal Encoding

Tamir Krief, Iaian Milton, Blessing Abumere
COSC 336
10/24/2024

100

Assignment 5

If a binary tree is not full, for example, if there is a node u with only one child v:

Case 1

If u is the root, delete u and make v the new root.

Case 2

If u is not the root:

- Let w be the parent of u.
- Delete u and replace it with v as a child of w.

In both cases, the resulting structure will maintain the properties of a binary tree, and the prefix code will remain valid. However, the depth of the tree may change, and any bits required to encode v may decrease, indicating that the original code may not have been optimal.

If deleting a node and restructuring the tree results in a more efficient encoding, it suggests that the original tree structure was not the most compact representation possible. An optimal prefix code (such as a Huffman code) would have already positioned more frequently used symbols closer to the root, thereby minimizing the overall encoding length.

Fibonacci Relationship

$$F_{i+1} + F_i + F_{i-1} = F_i + F_{i+2} - 1$$

Frequencies

- $a : 1$
- $b : 1$
- $c : 2$
- $d : 3$
- $e : 5$
- $f : 8$
- $g : 13$
- $h : 21$

Binary Tree Properties and Optimal Encoding

If a binary tree is not full, for example, if there is a node u with only one child v:

Case 1

If u is the root, delete u and make v the new root.

Case 2

If u is not the root:

- Let w be the parent of u.
- Delete u and replace it with v as a child of w.

In both cases, the resulting structure will maintain the properties of a binary tree, and the prefix code will remain valid. However, the depth of the tree may change, and any bits required to encode v may decrease, indicating that the original code may not have been optimal.

If deleting a node and restructuring the tree results in a more efficient encoding, it suggests that the original tree structure was not the most compact representation possible. An optimal prefix code (such as a Huffman code) would have already positioned more frequently used symbols closer to the root, thereby minimizing the overall encoding length.

Fibonacci Relationship

To find the Fibonacci relationship, we start with the identity:

$$F_{i+1} + F_i = F_{i+2}$$

If we also consider F_{i-1}, we can rewrite it as:

$$F_{i+1} + F_i + F_{i-1} = F_i + F_{i+1}$$

This leads to:

$$F_{i+1} + F_i + F_{i-1} = F_i + F_{i+2} - 1$$

In simple terms, this relationship shows how the Fibonacci numbers are connected. It highlights that if you add together certain Fibonacci numbers, you can find others. The pattern in these numbers can help us understand how to organize information more efficiently.

Frequencies

- $a : 1$
- $b : 1$
- $c : 2$
- $d : 3$
- $e : 5$
- $f : 8$
- $g : 13$
- $h : 21$

Optimal Huffman Codes

- $h : 0$
- $g : 10$
- $f : 110$
- $e : 1110$
- $d : 11110$

- c : 111110
- b : 1111110
- a : 1111111

Generalization

When we use the first n Fibonacci numbers as the frequencies for symbols, the process for creating Huffman codes works the same way. This means that more frequent Fibonacci numbers will end up with shorter codes, leading to a more efficient way of encoding the information.

100

Programming Task 5

Input	Max Sum	Sequence
1,14,5,6,2,3	15	1 14
input-5.1.txt	16	1,5,5,5
input-5.2.txt	134366	41,18467,26500,29358,30000,30000
input-5.3.txt	130021	4601,20255,23073,32092,50000
input-5.4.txt	143418	25197,26355,29960,30953,30953

Table 1: **Programming Task Results**

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;
import java.util.Stack;

public class Assignment5 {

    public static void main(String[] args) {
        System.out.println("Max Increasing Subsequences:");
            System.out.println("[1,14,5,6,2,3]: " + sumIncreasingSubsequence(new int[]
{1,14,5,6,2,3}));

            System.out.println("input-5.1.txt: " + sumIncreasingSubsequence(inputFile("input-
5.1.txt")));

            System.out.println("input-5.2.txt: " + sumIncreasingSubsequence(inputFile("input-
5.2.txt")));

            System.out.println("input-5.3.txt: " + sumIncreasingSubsequence(inputFile("input-
5.3.txt")));

            System.out.println("input-5.4.txt: " + sumIncreasingSubsequence(inputFile("input-
5.4.txt")));
    }

     /** reads the file and converts it to an int array */
    public static int[] inputFile(String filename) {
        try (Scanner console = new Scanner(new FileReader(filename))) {
            //size is the first number
            final int n = console.nextInt();

            //reads ints in the file
            int[] A = new int[n];
            for (int i = 0; i < n; i++)
                A[i] = console.nextInt();

            return A;
        }catch(FileNotFoundException e) {
            System.err.println("File not found: '" + filename + "'");
        }

        return null;
    }

    /** calculates the sum of the increasing subsequence
     * @param A the array of numbers
     * @return the max sum
     *
     */
    static public int sumIncreasingSubsequence(int[] A){
        if (A == null || A.length == 0) return 0;

        int N = A.length;
        int[] s = new int[N]; // max sum of an increasing subsequence with last element
        int[] p = new int[N]; // array of indexes of the elements preceding a[i];
        int i_maxSum = 0; // index of the max sum

        for (int i = 0; i < N; i++) {
            //You start with s[i] = a[i] and p[i] = ? where {?} is a number that represents the index
of the element before a[i] ; the number chosen doesnt matter .
            s[i] = A[i]; //the starting sum of A[i] which is whatever the first number from i is
            p[i] = -1; // element preceding index ; only useful to show summation {?} can be any
number less than 0 cus its an index
            for (int j = 0; j < i; j++) { //from left to whatever index i is ; j is an index to the
```

```
left of i
                if (A[i] >= A[j] && s[j] + A[i] >= s[i]){  // if A[j] is less than or equal to A[i]
and the sum of the subsequence ending at j plus A[i] is greater than the sum of the subsequence
ending at i
                    s[i] = s[j] + A[i] ; //sum is recalculated and left index of i is added to A[i]
                    p[i] = j;
                }
            }
            if (s[i] > s[i_maxSum]) // if summation at current i is greater than the last max sum
then i is the new max sum
                i_maxSum = i;
        }

        //part that exists only to show summation; doesnt effect maxsum
        {
            Stack<Integer> summation = new Stack<>();
            for (int i = i_maxSum; i != -1; i = p[i])
                summation.push(A[i]);

            System.err.print("\nSummation: \t");

            while (!summation.isEmpty())
                System.err.print(summation.pop() + " ");

            System.err.println();
        }

        return s[i_maxSum];
    }
}
```