Greedy Algorithms | Set 4 (Efficient Huffman Coding for Sorted Input)

We recommend to read following post as a prerequisite for this.

Greedy Algorithms | Set 3 (Huffman Coding)

Time complexity of the algorithm discussed in above post is O(nLogn). If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in O(n) time. Following is a O(n) algorithm for sorted input.

1. Create two empty queues.

2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.

3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times
…..**a)** If second queue is empty, dequeue from first queue.
…..**b)** If first queue is empty, dequeue from second queue.
…..**c)** Else, compare the front of two queues and dequeue the minimum.

4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to second queue.

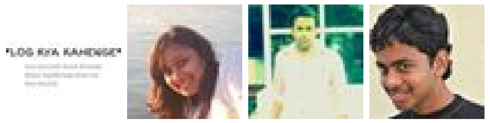
5. Repeat steps#3 and #4 until there is more than one node in the queues. The remaining node is the root node and the tree is complete.

```
// C Program for Efficient Huffman Coding for Sorted input
```

Google™ Custom Search

GeeksforGeeks

53,523 people like GeeksforGeeks.



Interview Experiences
Advanced Data Structures
Dynamic Programming
Greedy Algorithms
Backtracking
Pattern Searching
Divide & Conquer
Mathematical Algorithms
Recursion
Geometric Algorithms

```
#include <stdio.h>
#include <stdlib.h>

// This constant can be avoided by explicitly calculating height of Hu
#define MAX_TREE_HT 100

// A node of huffman tree
struct QueueNode
{
    char data;
    unsigned freq;
    struct QueueNode *left, *right;
};

// Structure for Queue: collection of Huffman Tree nodes (or QueueNode
struct Queue
{
    int front, rear;
    int capacity;
    struct QueueNode **array;
};

// A utility function to create a new Queuenode
struct QueueNode* newNode(char data, unsigned freq)
{
    struct QueueNode* temp =
       (struct QueueNode*) malloc(sizeof(struct QueueNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a Queue of given capacity
struct Queue* createQueue(int capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue))
    queue->front = queue->rear = -1;
    queue->capacity = capacity;
    queue->array =
      (struct QueueNode**) malloc(queue->capacity * sizeof(struct Queu
    return queue;
}

// A utility function to check if size of given queue is 1
int isSizeOne(struct Queue* queue)
{
```

Popular Posts

- All permutations of a given string
- Memory Layout of C Programs
- Understanding "extern" keyword in C
- Median of two sorted arrays
- Tree traversal without recursion and without stack!
- Structure Member Alignment, Padding and Data Packing
- Intersection point of two Linked Lists
- Lowest Common Ancestor in a BST.
- Check if a binary tree is BST or not
- Sorted Linked List to Balanced BST

```
    return queue->front == queue->rear && queue->front != -1;
}

// A utility function to check if given queue is empty
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}

// A utility function to check if given queue is full
int isFull(struct Queue* queue)
{
    return queue->rear == queue->capacity - 1;
}

// A utility function to add an item to queue
void enQueue(struct Queue* queue, struct QueueNode* item)
{
    if (isFull(queue))
        return;
    queue->array[++queue->rear] = item;
    if (queue->front == -1)
        ++queue->front;
}

// A utility function to remove an item from queue
struct QueueNode* deQueue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    struct QueueNode* temp = queue->array[queue->front];
    if (queue->front == queue->rear)  // If there is only one item in
        queue->front = queue->rear = -1;
    else
        ++queue->front;
    return temp;
}

// A utility function to get from of queue
struct QueueNode* getFront(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    return queue->array[queue->front];
}

/* A function to get minimum item from two queues */
```

```
struct QueueNode* findMin(struct Queue* firstQueue, struct Queue* seco
{
    // Step 3.a: If second queue is empty, dequeue from first queue
    if (isEmpty(firstQueue))
        return deQueue(secondQueue);

    // Step 3.b: If first queue is empty, dequeue from second queue
    if (isEmpty(secondQueue))
        return deQueue(firstQueue);

    // Step 3.c:  Else, compare the front of two queues and dequeue mi
    if (getFront(firstQueue)->freq < getFront(secondQueue)->freq)
        return deQueue(firstQueue);

    return deQueue(secondQueue);
}

// Utility function to check if this node is leaf
int isLeaf(struct QueueNode* root)
{
    return !(root->left) && !(root->right) ;
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// The main function that builds Huffman tree
struct QueueNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct QueueNode *left, *right, *top;

    // Step 1: Create two empty queues
    struct Queue* firstQueue  = createQueue(size);
    struct Queue* secondQueue = createQueue(size);

    // Step 2:Create a leaf node for each unique character and Enqueue
    // the first queue in non-decreasing order of frequency. Initially
    // queue is empty
    for (int i = 0; i < size; ++i)
        enQueue(firstQueue, newNode(data[i], freq[i]));
```

Recent Comments

Aman Hi, Why arent we checking for conditions...
Write a C program to Delete a Tree. · 33 minutes ago

kzs please provide solution for the problem...
Backtracking | Set 2 (Rat in a Maze) · 36 minutes ago

Sanjay Agarwal bool tree::Root_to_leaf_path_given_sum(tree...
Root to leaf path sum equal to a given number · 1 hour ago

GOPI GOPINATH @admin Highlight this sentence "We can easily...
Count trailing zeroes in factorial of a number · 1 hour ago

newCoder3006 If the array contains negative numbers also. We...
Find subarray with given sum · 1 hour ago

newCoder3006 Code without using while loop. We can do it...
Find subarray with given sum · 1 hour ago

```
    // Run while Queues contain more than one node. Finally, first que
    // be empty and second queue will contain only one node
    while (!(isEmpty(firstQueue) && isSizeOne(secondQueue)))
    {
        // Step 3: Dequeue two nodes with the minimum frequency by exa
        // the front of both queues
        left = findMin(firstQueue, secondQueue);
        right = findMin(firstQueue, secondQueue);

        // Step 4: Create a new internal node with frequency equal to
        // of the two nodes frequencies. Enqueue this node to second q
        top = newNode('$' , left->freq + right->freq);
        top->left = left;
        top->right = right;
        enQueue(secondQueue, top);
    }

    return deQueue(secondQueue);
}

// Prints huffman codes from the root of Huffman Tree.  It uses arr[]
// store codes
void printCodes(struct QueueNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then it contains one of the input
    // characters, print the character and its code from arr[]
    if (isLeaf(root))
    {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}
```

```
// The main function that builds a Huffman Tree and print codes by tra
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
   //  Construct Huffman Tree
   struct QueueNode* root = buildHuffmanTree(data, freq, size);

   // Print Huffman codes using the Huffman tree built above
   int arr[MAX_TREE_HT], top = 0;
   printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}
```

Output:

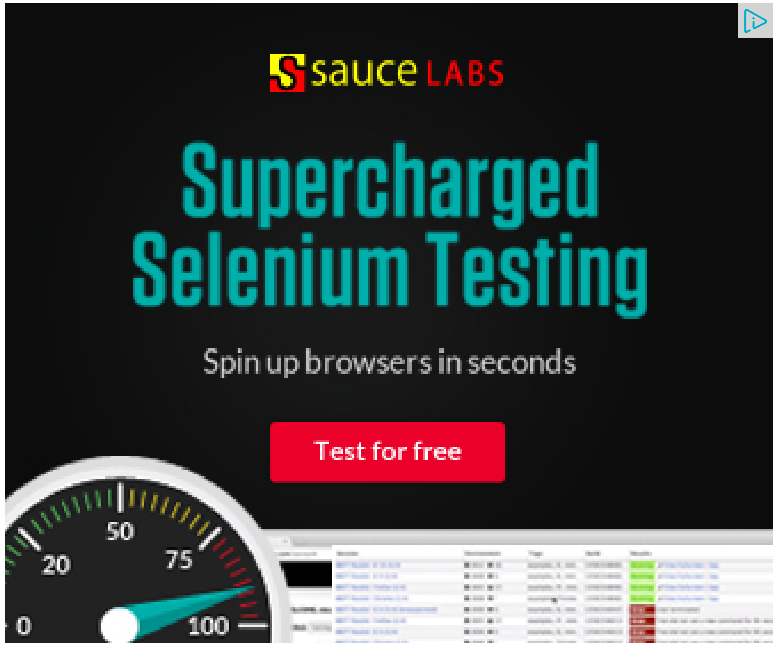
```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time complexity: O(n)

If the input is not sorted, it need to be sorted first before it can be processed by the above algorithm. Sorting can be done using heap-sort or merge-sort both of which run in Theta(nlogn). So, the overall time complexity becomes O(nlogn) for unsorted input.

Reference:
http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.



Related Tpoics:

- Backtracking | Set 8 (Solving Cryptarithmetic Puzzles)
- Tail Recursion
- Find if two rectangles overlap
- Analysis of Algorithm | Set 4 (Solving Recurrences)
- Print all possible paths from top left to bottom right of a mXn matrix
- Generate all unique partitions of an integer
- Russian Peasant Multiplication
- Closest Pair of Points | O(nlogn) Implementation

1 Tweet 0 0

Writing code in comment? Please use **ideone.com** and share the link here.

1 Comment GeeksforGeeks

Sort by Newest

Join the discussion…

veggie · 5 days ago

What about printing it in alphabetical order?

Subscribe Add Disqus to your site