**Data Structures Lab 10**

**Course:** Data Structures (CL2001) **Semester:** Fall 2023

**Instructor: Shafique Rehman**  **T.A:** N/A

**Note:**

* + - * Lab manual cover following topics

**{**Heaps, Min Heap, Max Heap**}**

* Maintain discipline during the lab.
* Just raise your hand if you have any problem.
* Completing all tasks of each lab is compulsory.
* Get your lab checked at the end of the session.

**HEAP Data Structures**

What is Heap Data Structure?

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Operations of Heap Data Structure:

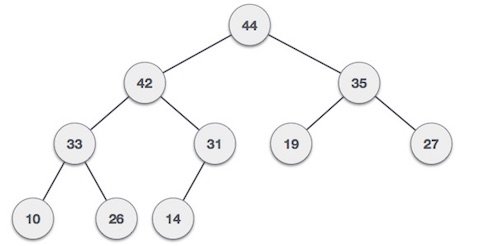
1. Heapify: a process of creating a heap from an array.
2. Insertion: process to insert an element in existing heap time complexity O(log N).
3. Deletion: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
4. Peek: to check or find the most prior element in the heap, (max or min element for max and min heap).



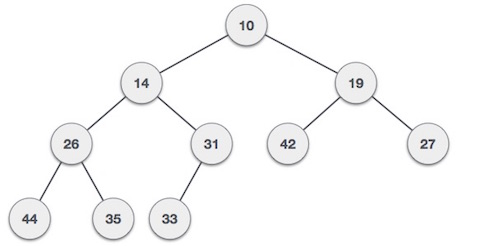
Types of Heap Data Structure

Generally, Heaps can be of two types:

Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.



Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.



**How to construct a HEAP:**

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

**When implementing a max heap using an array,**

* Its left child is at index 2i.
* Its right child is at index 2i + 1.
* Its parent is at index i/2 (integer division/take floor value e.g. 5/2=2.5=2).

**Note: Start array from 1 index for mathematical convenience and ease of parent-child calculations.**

**For max heap check: A[parent]>=A[i] (except root node because root does not have the parent.)**

**Step 1** − Add the new element at the end of the array representing the heap.

**Step 2** − Compare the new element with its parent.

**Step 3** − If the new element is greater than its parent, swap the new element with its parent.

**Step 4** − Repeat steps 2 and 3 until the new element is in its correct position (i.e., until it's no longer greater than its parent or until it becomes the root).

**Note** − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Graphical user interface, text, application

Description automatically generated

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

**Step 1** − Replace the root (maximum element) with the last element in the heap.

**Step 2** − Remove the last element.

**Step 3** − Heapify the heap to maintain the max heap property:

* + Compare the new root with its children.
  + Swap the root with its largest child if necessary.
  + Repeat this process recursively until the heap property is restored.

A picture containing watch, different

Description automatically generated

Application of Heaps (Huffman Tree):

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

These are called fixed-length codes. If all characters were used equally often, then a fixed-length coding scheme is the most space efficient method. But such thing isn’t possible in real word. If some characters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other characters require longer codes, but this might be worthwhile if such characters appear rarely enough. Huffman coding variable-length codes approaches. While it is not commonly used in its simplest form for file compression, One motivation for studying Huffman coding is because it provides  type of tree structure referred to as a search trie.

There are mainly two major parts in Huffman Coding

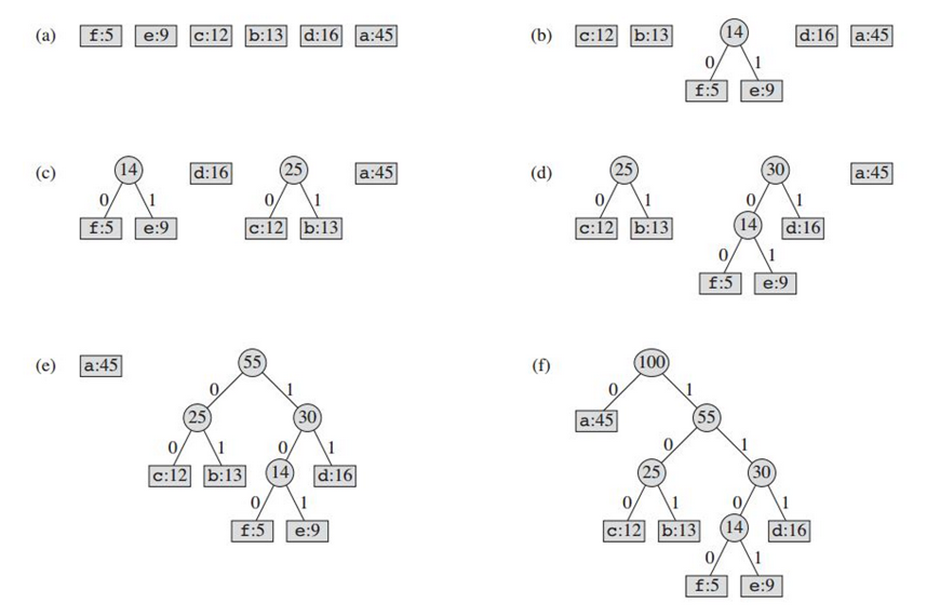
    Build a Huffman Tree from input characters.

    Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree (Compression Technique)

1. The technique works by creating a binary tree of nodes. Tree can stored in a regular array, the size of which depends on the number of symbols, n. A node can either be a leaf node or an internal node. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. Priority queue is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
3. Remove the two nodes of highest priority from the queue.
4. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.
5. Add the new node to the queue.
6. The remaining node is the root node and the Huffman tree is complete.



**Decompression Technique:**

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately.

Procedure HuffmanDecompression(root, S):   // root represents the root of Huffman Tree

n := S.length                              // S refers to bit-stream to be decompressed

for i := 1 to n

    current = root

    while current.left != NULL and current.right != NULL

        if S[i] is equal to '0'

            current := current.left

        else

            current := current.right

        endif

        i := i+1

    endwhile

    print current.symbol

endfor