Heap Sort

Data Structures and Algorithms

Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap data structure.
- It is similar to selection sort where we first find the maximum element and place the maximum element at the end.
- We repeat the same process for remaining element.

Binary Heap

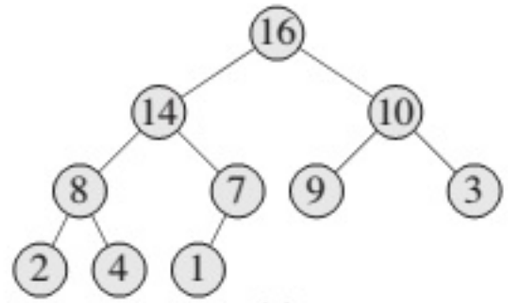
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes.

- The former is called as max heap and the latter is called min heap.

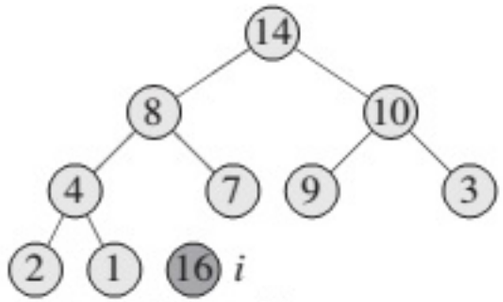
Heap Sort Algorithm

- **1.** Build a max heap from the input data.
- **2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
- **3.** Repeat above steps while size of heap is greater than 1.

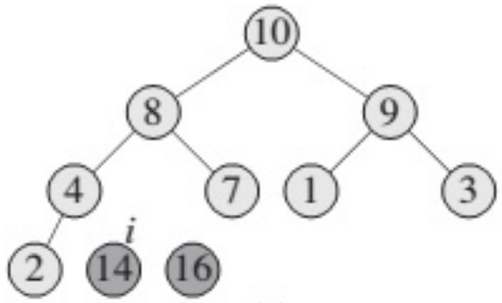
Priority Queue



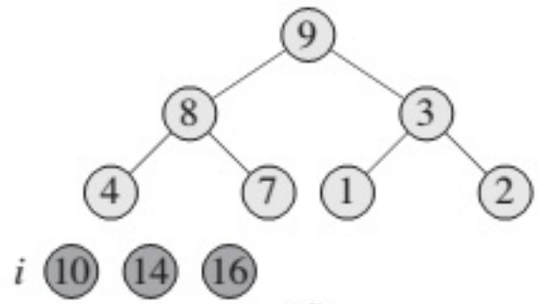
(a)



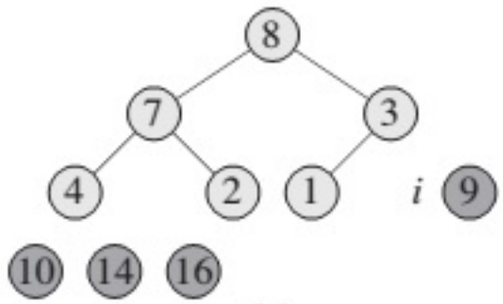
(b)



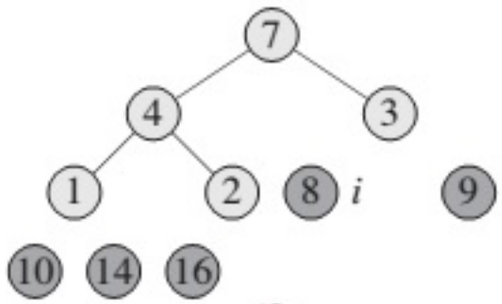
(c)



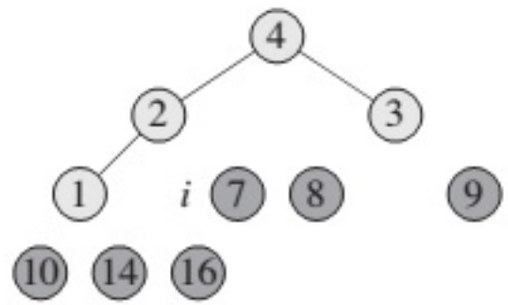
(d)



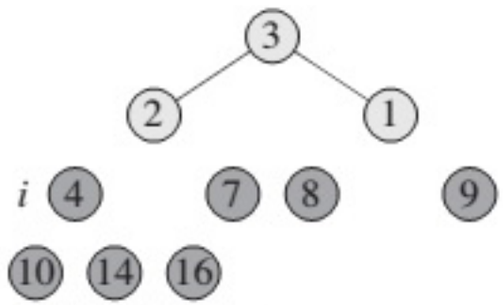
(e)



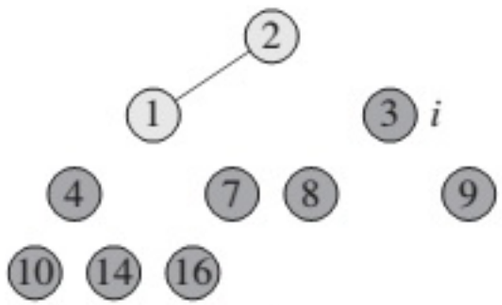
(f)



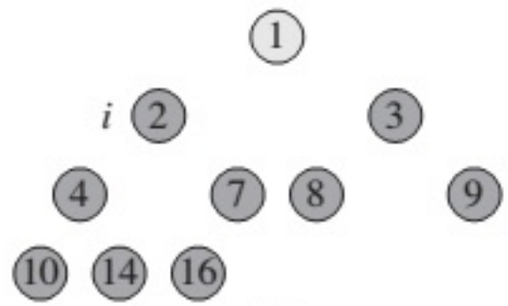
(g)



(h)



(i)



(j)

(k)

Heap Sort Complexity

Table 3.14.3: Fast sorting algorithm's best, average, and worst case runtime complexity.

Sorting algorithm	Best case runtime complexity	Average case runtime complexity	Worst case runtime complexity
Quicksort	O($NlogN$)	O($NlogN$)	O(N^2)
Merge sort	O($NlogN$)	O($NlogN$)	O($NlogN$)
Heap sort	O(N)	O($NlogN$)	O($NlogN$)
Radix sort	O(N)	O(N)	O(N)

```c
#include<stdio.h>
#include <conio.h>
void heapify_function(int arr[])
{
    int i,n;
    n=arr[0];
    for(i=n/2;i>=1;i--)
    adjust(arr,i);
}
void adjust(int arr[],int i)
{
    int j,temp,n,k=1;
    n=arr[0];
    while(2*i<=n && k==1)
    {
       j=2*i;
       if(j+1<=n && arr[j+1] > arr[j])
           j=j+1;

       if( arr[j] < arr[i])
           k=0;
       else
       {
           temp=arr[i];
           arr[i]=arr[j];
           arr[j]=temp;
           i=j;
        }
      }
}
```

```c
void main()
{
int arr[100],n,temp,last,i;
clrscr();
printf("How many Numbers you want to enter in your array: \n");
scanf("%d",&n);
printf("Enter Elements in array:\n");
for(i=1;i<=n;i++)
scanf("%d",&arr[i]);
arr[0]=n;
heapify_function(arr);
while(arr[0] > 1)
{
    last=arr[0];
    temp=arr[1];
    arr[1]=arr[last];
    arr[last]=temp;
    arr[0]--;
    adjust(arr,1);
}

printf("Array After Heap Sort\n");
for(i=1;i<=n;i++)
printf("%d ",arr[i]);
getch();
}
```

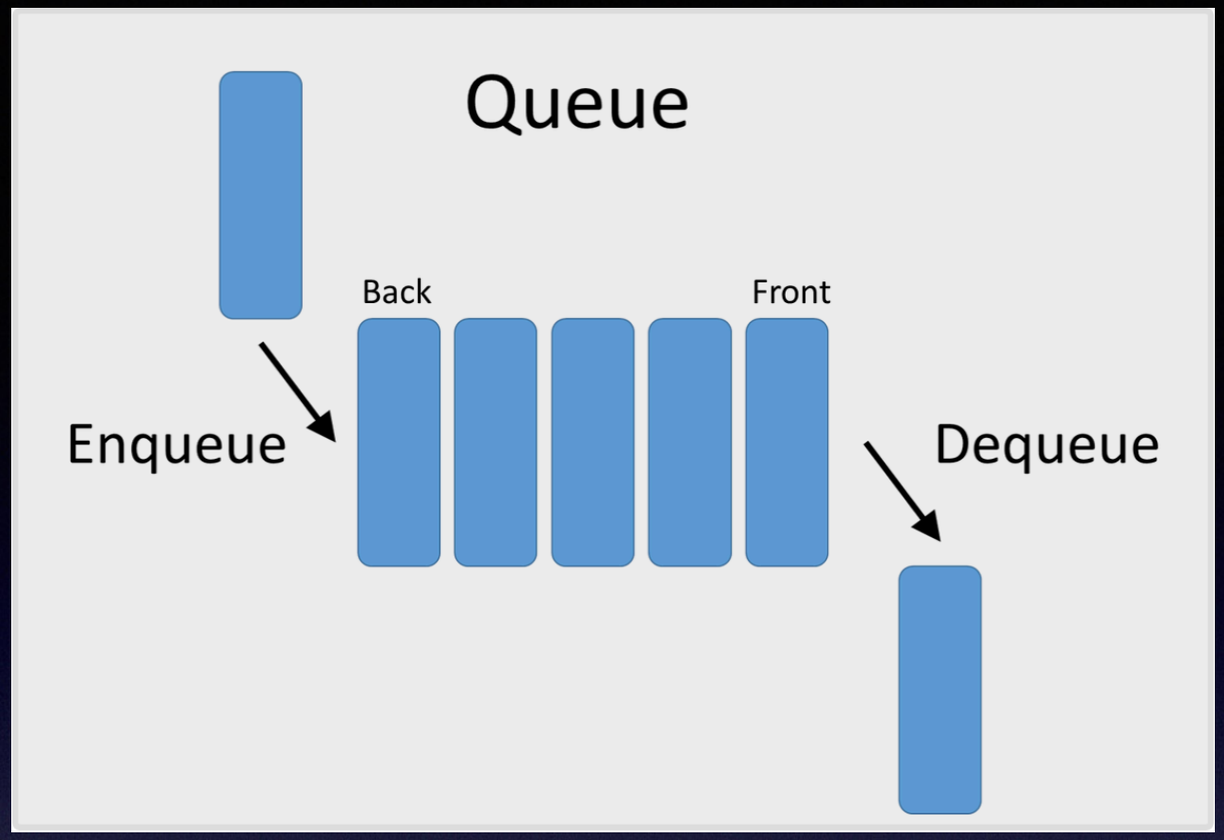
Advantages of Heap Sort

- It has a logarithmic time complexity
- Always suggested for huge arrays.
- It is an in-place sorting algorithm that does not require extra memory space for additional Array.
- The same time complexity for average, best, and worst cases

Disadvantages of Heap Sort

- It is not a stable algorithm, which means the order of the same element may be changed.
    - Stable means if the two elements have the same key, they remain in the same order or positions. But that is not the case for Heap sort.
    - **Heapsort** is not stable because operations on the heap can change the relative order of equal items.
- Not that much efficient as compared to quick and merge sort
- The recursive call can be complicated.

Priority Queue


Queue
Back
Front
Enqueue
Dequeue

Priority Queue
3
Back
Front
1
1
1
2
3
Dequeue
Enqueue
3

Priority Queue

- Priority Queue is an extension of queue with following properties.
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Priority Queue Operations

- **insert(item, priority):** Inserts an item with given priority.
- **getHighestPriority():** Returns the highest priority item.
- **deleteHighestPriority():** Removes the highest priority item.

Applications of Priority Queue

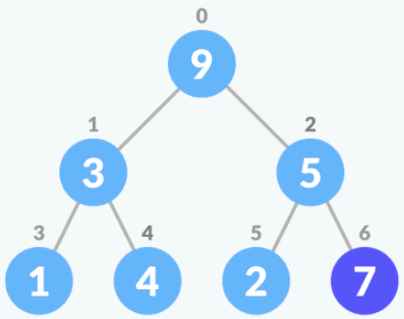
- CPU Scheduling
- Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
- All queue applications where priority is involved.

Implementation of Priority Queue

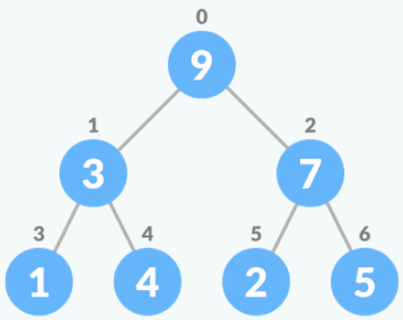
	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`

Priority Queue Operations - Insert

1. Insert the new element at the end of the tree.

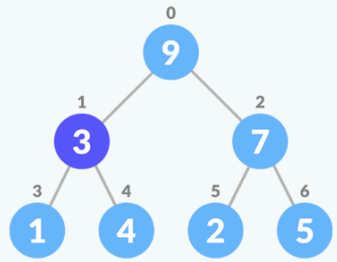


2. Heapify the tree.

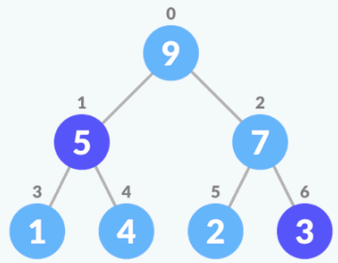


Priority Queue Operations - Delete

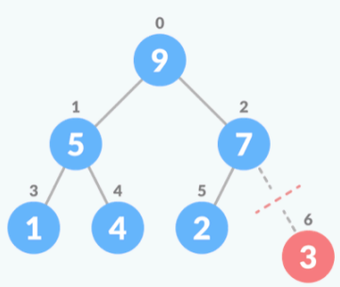
1. Select the element to be deleted.



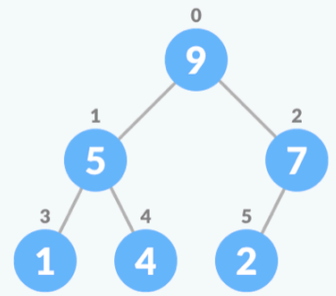
2. Swap it with the last element.



3. Remove the last element.



4. Heapify the tree.



Huffmann Encoding

Data Compression

- In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression.

- The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file).

Huffman Encoding

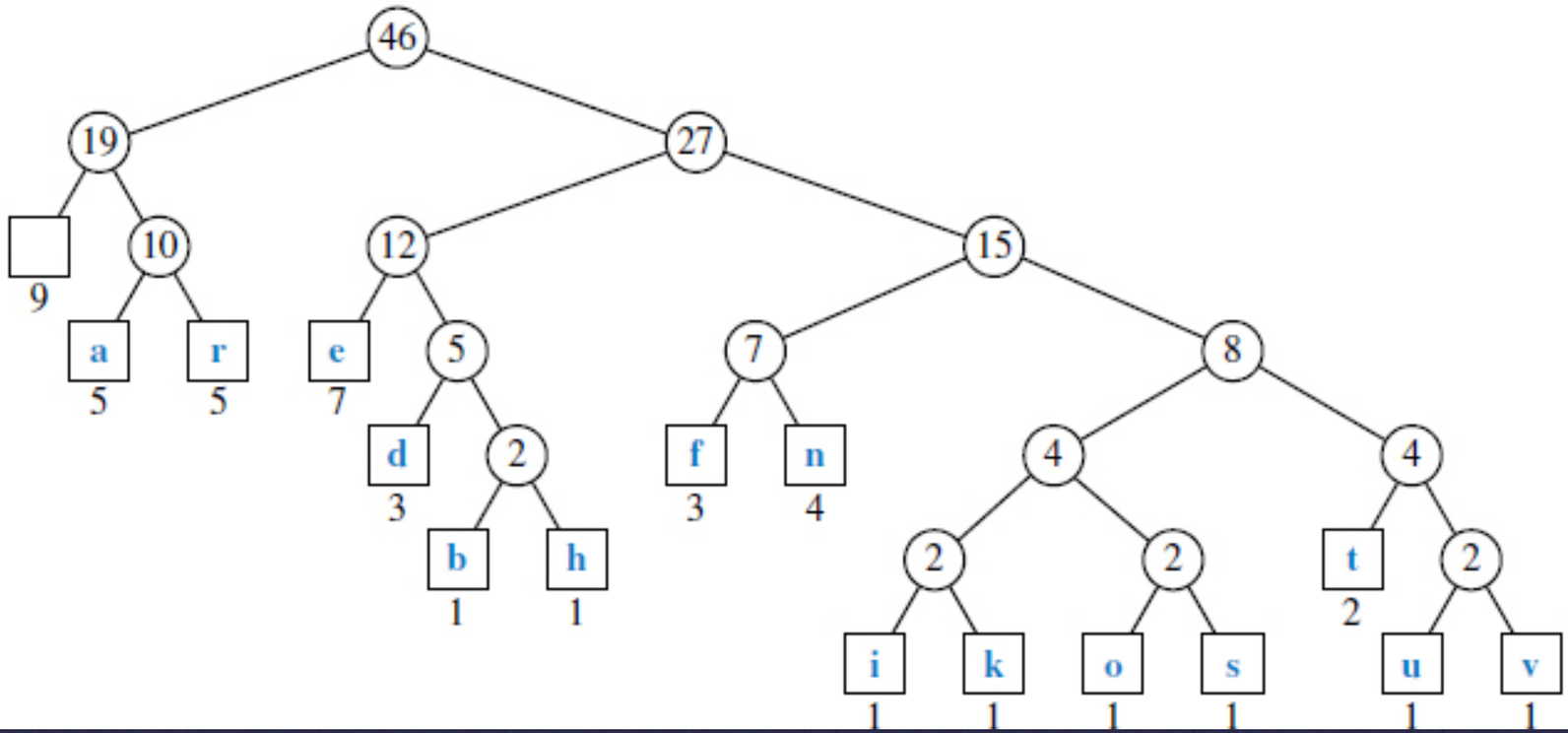
Input and Output

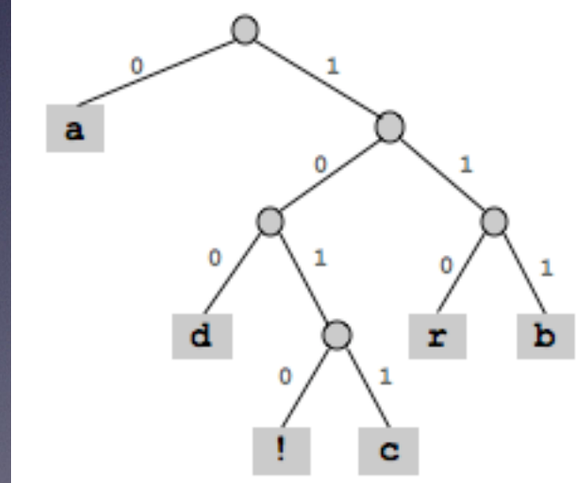
```
Input:
A string with different characters, say “ACCEBFFFFAAXXBLKE”
Output:
Code for different characters:
Data: K, Frequency: 1, Code: 0000
Data: L, Frequency: 1, Code: 0001
Data: E, Frequency: 2, Code: 001
Data: F, Frequency: 4, Code: 01
Data: B, Frequency: 2, Code: 100
Data: C, Frequency: 2, Code: 101
Data: X, Frequency: 2, Code: 110
Data: A, Frequency: 3, Code: 111
```

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

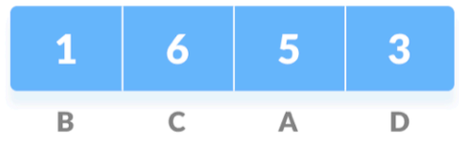




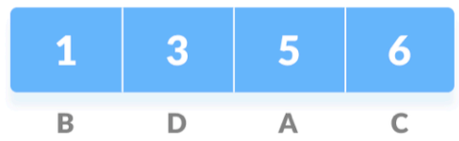
char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Huffman Coding Steps - I

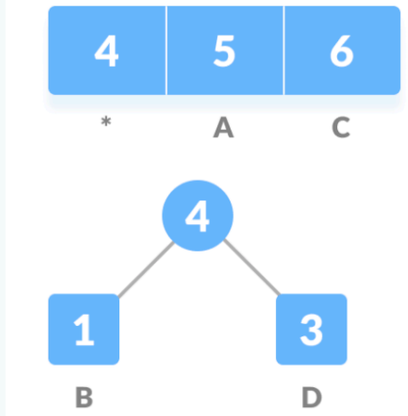
1. Calculate the frequency of each character in the string.



2. Sort the characters in increasing order of the frequency. These are stored in a priority queue `Q`.



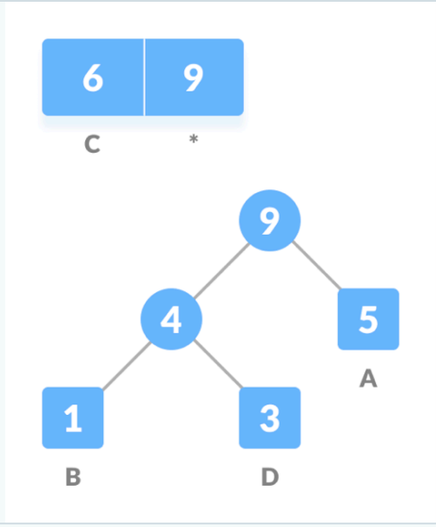
3. Make each unique character as a leaf node.
4. Create an empty node `z`. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of `z`. Set the value of the `z` as the sum of the above two minimum frequencies.

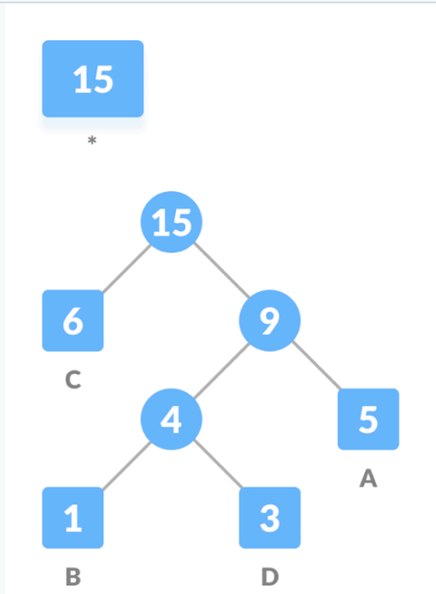


Huffman Coding

Huffman Coding Steps - II

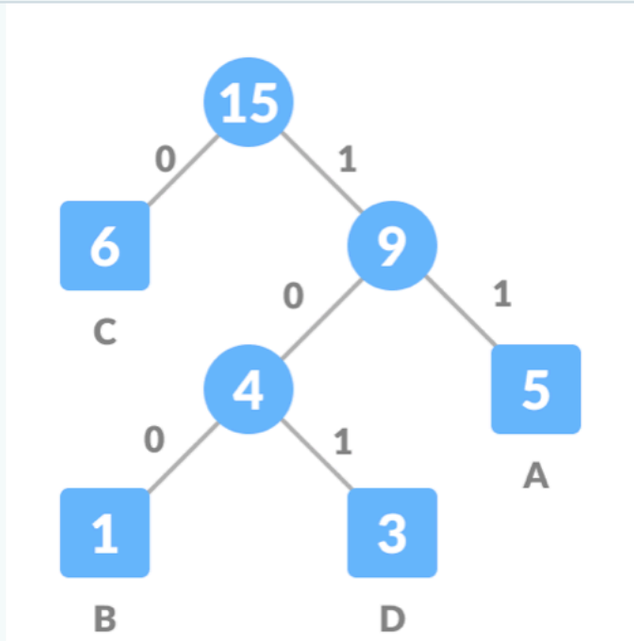
5. Remove these two minimum frequencies from `Q` and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node `z` into the tree.
7. Repeat steps 3 to 5 for all the characters.





Huffman Coding Steps - III

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



TO BE CONTINUED…