1.2.3 Best Case Analysis

- The best case analysis of an algorithm gives a lower bound on the resources required by the algorithm.
- The best case running time of an algorithm determines the minimum amount of time taken by the algorithm on any input.
- The best case running time ensures that the algorithm runs the fastest.

  In general worst case, average case and best case analyses are done to determine the running time of algorithms but they can be useful for determining the requirement of memory or other resources as well.

1.3 AMORTIZED ANALYSIS

- In an amortized analysis of running time of an algorithm, the time required to perform a sequence of data structure operations is averaged over all the operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average case analysis in the sense that amortized analysis does not involve probability. Amortized analysis guarantees the average performance of each operation in the worst case.

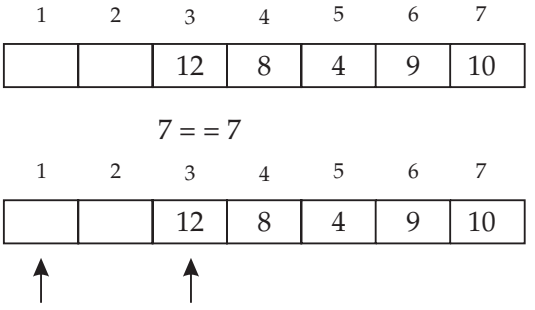
1.3.1 Aggregate Analysis

- In an aggregate analysis, for all n, a sequence of n operations takes worst case time $T(n)$ in total.
- Therefore, in the worst case, the average cost or amortized cost, per operation is $T(n)/n$. Here the amortized cost applies to each operation, even when there are several types of operations in the sequence.

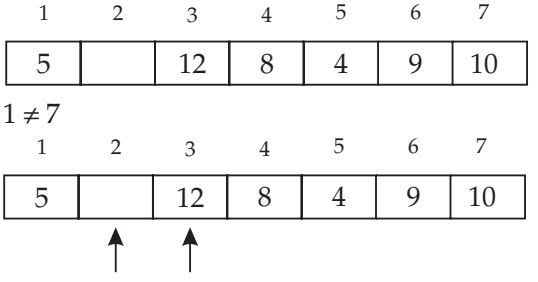
1.3.2 The Potential Method

- The potential method represents the prepaid work as 'potential' which can be released to pay for future operations.
- This potential is associated with the data structure as a whole rather than with specific objects within the data structure.
- Let n operations are performed, starting with an initial data structure D_0.

i. Enqueue (Q, 10)



ii. Enqueue (Q, 5)



Algorithm

Dequeue (Q)

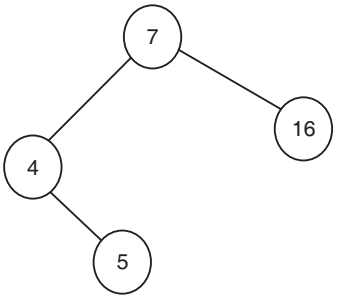
```
Line 1    x ← Q [head [Q]]
Line 2    if  head [Q] = = length [Q]
Line 3          head [Q] ← 1
Line 4    else  head [Q] ← head [Q] + 1
Line 5    return  x
```

Analysis

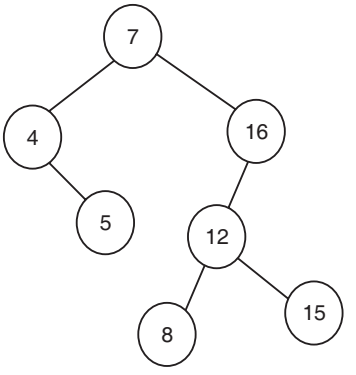
- The Dequeue procedure takes O (1) time.

Explanation

- The input to the Dequeue procedure is a queue Q. This procedure is used to delete elements from the queue.
- Line 1 indicates that x is the element at location Q[head [Q]].
- Line 2 checks the *if* condition. If this condition is true then the execution of Line 3 takes place else the control goes to Line 4.
- The *if* condition is true when the attribute head [Q] is equal to the length of Q. It means head [Q] points to the last location of Q.



After the execution of Line 11 and Line 12



Algorithm

Right-Rotate (T, x)

```
Line 1    y ← left [x]
Line 2    left [x] ← right [y]
Line 3    if   right [y] ≠ nil [T]
Line 4         P [right [y]] ← x
Line 5    P [y] ← P [x]
Line 6    if   P [x] = = nil [T]
Line 7         root [T] ← y
Line 8    elseif   x = = right [P[x]]
Line 9         right [P[x]] ← y
Line 10   else     left [P[x]] ← y
Line 11   right [y] ← x
Line 12   P [x] ← y
```

Assumptions

Following are the assumptions made from the above algorithm:

i. *y* is left child of *x*
ii. *y* is not nil [*T*]
iii. Parent of root is nil [*T*]

- Line 3 makes pointer y to point to the right child of the parent of parent of z. It means that y points to the right child of the grandparent of z, or in other words y points to z's uncle.
- Line 4 checks the *if* condition. This condition is true if the color of node y is red. If this condition is true then the execution of Lines 5, 6, 7 and 8 takes place. If this condition is false then the procedure checks the *elseif* condition of Line 9.
- Line 5 paints the parent of z black.
- Line 6 paints node y black.
- Line 7 paints grandparent of z red.
- Line 8 makes the earlier grandparent of z as the new z. So the pointer z moves up two levels in the tree.
- Line 9 checks the *elseif* condition. This condition is true if z is the right child of the parent of z. If this condition is false then the control goes to Line 12.
- Line 10 makes the earlier parent of z as the new z. So the pointer z moves one level up in the tree.
- Line 11 calls the procedure Left-Rotate (T, z).
- Line 12 paints the parent of z black.
- Line 13 paints the grandparent of z red.
- Line 14 calls the procedure Right-Rotate (T, P [P[z]]).
- Line 15 makes pointer y to point to the left child of the parent of parent of z. It means that y points to the left child of the grandparent of z or in other words y points to z's uncle.
- Line 16 checks the *if* condition. This condition is true if the color of node y is red. If this condition is true then the execution of Lines 17, 18, 19 and 20 takes place. If this condition is false then the procedure checks the *elseif* condition of Line 21.
- Line 17 paints the parent of z black.
- Line 18 paints the node y (z's uncle) black.
- Line 19 paints grandparent of z red.
- Line 20 makes the earlier grandparent of z as the new z. So the pointer z moves up two levels in the tree.
- Line 21 checks the *elseif* condition. This condition is true if z is the left child of the parent of z. If this condition is false then the control goes to Line 24.
- Line 22 makes the earlier parent of z as the new z. So the pointer z moves one level up in the tree.
- Line 23 calls the procedure Right-Rotate (T, z).
- Line 24 paints the parent of z black.

Contents