CONTENTS

RCS–502 : Design and Analysis of Algorithm

1

UNIT

Introduction

PART-1

Introduction : Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements.

CONCEPT OUTLINE : PART-1

- **Algorithm :** An algorithm is a sequence of computational steps that transform the input into the output.

  Input → Algorithm → Output

- **Complexity of algorithm :** Complexity of an algorithm is defined by two terms :
  i. Time complexity
  ii. Space complexity
- **Analysis of algorithm :** The analysis of an algorithm provides some basic information about that algorithm like time, space, performance etc.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.1. What do you mean by algorithm ? Write the characteristics of algorithm. **AKTU 2013-14, Marks 05**

Answer

1. An algorithm is a set of rules for carrying out calculation either by hand or on machine.
2. It is a finite step-by-step procedure to achieve a required result.
3. It is a sequence of computational steps that transform the input into the output.
4. An algorithm is a sequence of operations performed on data that have to be organized in data structures.

Characteristics of algorithm are :

1. **Input and output :** These characteristics require that an algorithm produces one or more outputs and have zero or more inputs that are externally supplied.

2. **Definiteness :** Each operation must be perfectly clear and unambiguous.
3. **Effectiveness :** This requires that each operation should be effective, *i.e.*, each step can be done by a person using pencil and paper in a finite amount of time.
4. **Termination :** This characteristic requires that an algorithm must terminate after a finite number of operations.

Que 1.2. What do you mean by analysis or complexity of an algorithm ? Give its types and cases.

Answer

Analysis/complexity of an algorithm :

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

Types of complexity :

1. **Space complexity :** The space complexity of an algorithm is the amount of memory it needs to run to completion.
2. **Time complexity :** The time complexity of an algorithm is the amount of time it needs to run to completion.

Cases of complexity :

1. **Worst case complexity :** The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.
2. **Average case complexity :** The running time for any given size input will be the average number of operations over all problem instances for a given size.
3. **Best case complexity :** The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

Que 1.3. What do you understand by asymptotic notations ? Describe important types of asymptotic notations.

AKTU 2013-14, Marks 05

OR

Discuss asymptotic notations in brief. **AKTU 2014-15, Marks 05**

Answer

1. Asymptotic notation is a shorthand way to represent 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed.

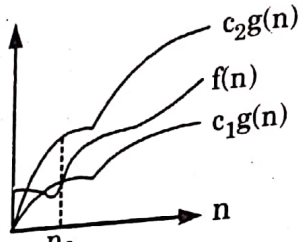
2. It is a line that stays within bounds.
3. These are also referred to as 'best case' and 'worst case' scenarios and are used to find complexities of functions.

Notations used for analyzing complexity are :

1. Θ-Notation (Same order) :

a. This notation bounds a function within constant factors.

b. We say $f(n) = \Theta g(n)$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.



f(n) = Θ (g(n))

Fig. 1.3.1.

2. O-Notation (Upper bound) :

a. Big-oh is formal method of expressing the upper bound of an algorithm's running time.

b. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.

c. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$.

$$f(n) \leq cg(n)$$

d. Then, $f(n)$ is big-oh of $g(n)$. This is denoted as :

$$f(n) \in O(g(n))$$

i.e., the set of functions which, as n gets large, grow faster than a constant time $f(n)$.

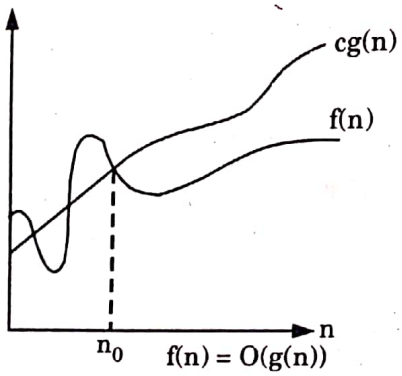


Fig. 1.3.2.

3. Ω-Notation (Lower bound) :

a. This notation gives a lower bound for a function within a constant factor.

b. We write $f(n) = \Omega g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

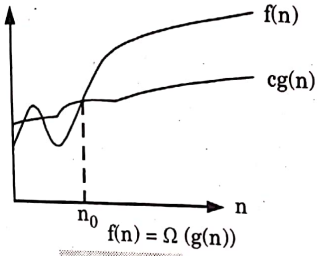


Fig. 1.3.3.

4. Little-oh notation (o) : It is used to denote an upper bound that is asymptotically tight because upper bound provided by O-notation is not tight.

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, if a constant $n_0 > 0$ such that $0 \le f(n) < cg(n) \; \forall \; n \ge n_0\}$

5. Little omega notation (ω) : It is used to denote lower bound that is asymptotically tight.

$\omega(g(n)) = \{f(n)$: For any positive constant $c > 0$, if a constant $n_0 > 0$ such that $0 \le cg(n) < f(n) \; \forall \; n \ge n_0\}$

Que 1.4. If $f(n) = 100 * 2^n + n^5 + n$, then show that $f(n) = O(2^n)$.

Answer

If $f(n) = 100 * 2^n + n^5 + n$

For $n^5 \ge n$

$$100 * 2^n + n^5 + n \le 100 * 2^n + n^5 + n^5$$
$$\le 100 * 2^n + 2n^5$$

For $2^n \ge n^5$

$$100 * 2^n + n^5 + n \le 100 * 2^n + 2.2^n$$
$$\le 102 * 2^n$$

Thus, $f(n) = O(2^n)$ $[\because \; n \le 1, n_0 = 23]$

Que 1.5. Consider the following function :

```
int SequentialSearch(int A[ ], int &x, int n)
{
int i;
for (int i = 0; i < n && a[i] != x ; i++)
if (i == n) return – 1;
return i;
```

}

Determine the average and worst case time complexity of the function sequential search.

Answer

Average case time complexity :

1. Let P be a probability of getting successful search and n is the total number of elements in the list.
2. The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.
3. The probability of getting unsuccessful search is $(1-P)$.
4. Average case time complexity $C_{avg}(n)$ = Probability of successful search + Probability of unsuccessful search

$$= \left[1.\frac{P}{n} + 2.\frac{P}{n} + \ldots + i.\frac{P}{n}\right] + n(1-P)$$

$$= \frac{P}{n}[1 + 2 + \ldots + i] + n(1-P)$$

$$= \frac{P}{n}\frac{n(n+1)}{2} + n(1-P) \qquad (\because\ i = i{+}{+} \text{ till } i < n)$$

$$C_{avg}(n) = \frac{P(n+1)}{2} + n(1-P)$$

5. So, for unsuccessful search, $P = 0$

   Then $C_{avg}(n) = 0 + n(1-0)$

   $= n$

6. Thus the average case time complexity becomes equal to n for unsuccessful search.
7. For successful search,

   $P = 1$

$$C_{avg}(n) = \frac{1(n+1)}{2} + n(1-1) = \frac{n+1}{2} + 0$$

$$C_{avg}(n) = O\left(\frac{n}{2}\right)$$

8. That means the algorithm scans about half of the elements from the list.

Worst case time complexity :

1. In this algorithm, we will consider the worst case when the element to be searched is present at n^{th} location then the algorithm will run for longest time and we will get the worst case time complexity as :

   $C_{worst}(n) = O(n)$
2. For any instance of input of size n, the running time will not exceed $O(n)$.

Que 1.6. Write Master's theorem and explain with suitable examples.

Answer

Master's theorem :

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1 \text{ are constants}$$

a = the number of sub-problems in the recursion

$1/b$ = the portion of the original problem represented by each sub-problem

$f(n)$ = the cost of dividing the problem + the cost of merging the solution

Then $T(n)$ can be bounded asymptotically as follows :

Case 1 :

If it is true that :

$$f(n) = O(n^{\log_b a - E}) \text{ for } E > 0$$

It follows that :

$$T(n) = \Theta(n^{\log_b a})$$

Example :

$$T(n) = 8T\left(\frac{n}{b}\right) + 1000n^2$$

In the given formula, the variables get the following values :

$$a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$$

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

For $E = 1$, we get

$$f(n) = O(n^{\log_b a - E}) = O(n^{3-E})$$

$$f(n) = O(n^{3-1}) = O(n^2)$$

Since this equation holds, the first case of the master's theorem applies to the given recurrence relation, thus resulting solution is

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Case 2 :

If it is true that :

$$f(n) = \Theta(n^{\log_b a})$$

It follows that :

$$T(n) = \Theta(n^{\log_b a} \log(n))$$

Example :

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

In the given formula, the variables get the following values :

$$a = 2, b = 2, f(n) = n, \log_b a = \log_2 2 = 1$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

Since this equation holds, the second case of the Master's theorem applies to the given recurrence relation, thus resulting solution is :

$$T(n) = \Theta(n^{\log_b a} \log(n)) = \Theta(n \log n)$$

Case 3 :

If it is true that :

$$f(n) = \Omega(n^{\log_b a + E}) \text{ for } E > 0$$

and if it is also true that :

if $af\left(\frac{n}{b}\right) \leq cf(n)$ for $a, c < 1$ and all sufficiently large n

It follows that :

$$T(n) = \Theta(f(n))$$

Example :

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

In the given formula, the variables get the following values :

$$a = 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Omega(n^{\log_b a + E})$$

For $E = 1$ we get

$$f(n) = \Omega(n^{1+1}) = \Omega(n^2)$$

Since the equation holds, third case of Master's theorem is applied.

Now, we have to check for the second condition of third case, if it is true that :

$$af\left(\frac{n}{b}\right) \leq c\, f(n)$$

If we insert once more the values, we get :

$$2\left(\frac{n}{2}\right)^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$$

If we choose $c = \frac{1}{2}$, it is true that :

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 \ \forall\ n \geq 1$$

So, it follows that :

$$T(n) = \Theta(f(n))$$

If we insert once more the necessary values, we get :

$$T(n) \in \Theta(n^2)$$

Thus, the given recurrence relation $T(n)$ was in $\Theta(n^2)$.

Que 1.7. **The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A has a running time $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a A' is asymptotically faster than A ?** **AKTU 2017-18, Marks 10**

Answer

Given that :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \qquad \ldots(1.7.1)$$

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2 \qquad \ldots(1.7.2)$$

Here, equation (1.7.1) defines the running time for algorithm A and equation (1.7.2) defines the running time for algorithm A'. Then for finding value of a for which A' is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare equation (1.7.1) by $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get, $a = 7$

$b = 2$

$f(n) = n^2$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Now, apply cases of Master's, theorem as :

Case 1 : $f(n) = O(n^{\log_2 7 - E})$

$\Rightarrow$ $f(n) = O(n^{2.81 - E})$

$\Rightarrow$ $f(n) = O(n^{2.81 - 0.81})$

$\Rightarrow$ $f(n) = O(n^2)$

Hence, case 1 of Master's theorem is satisfied.

Thus, $T(n) = \theta(n^{\log_b a})$

$\Rightarrow$ $T(n) = \theta(n^{2.81})$

Since recurrence given by equation (1.7.1) is asymptotically bounded by θ-notation by which is used to show optimum time we have to show that recurrence given by equation (1.7.2) is bounded by Ω-notation which shows minimum time (best case).

For the use satisfy the case 3 of Master theorem, let $a = 16$

$$T'(n) = 16T'\left(\frac{n}{4}\right) + n^2$$

$\Rightarrow$ $a = 16$

$b = 4$

$f(n) = n^2$

$$\Omega(n^{\log_b a + E}) = \Omega(n^{2+E})$$

Hence, case 3 of Master's theorem is satisfied.

$\Rightarrow$ $T(n) = \theta(f(n))$

$\Rightarrow$ $T(n) = \theta(n^2)$

Therefore, this shows that A' is asymptotically faster than A when $a = 16$.

Que 1.8. Solve the following recurrences :

$T(n) = T(\sqrt{n}) + O(\log n)$ **AKTU 2014-15, Marks 10**

Answer

$$T(n) = T(\sqrt{n}) + O(\log n) \quad \ldots(1.8.1)$$

Let $m = \log_2 n$

$n = 2^m$

$n^{1/2} = 2^{m/2}$ ...(1.8.2)

Put value of $\sqrt{n}$ in equation (1.8.1) we get

$$T(2^m) = T\left(2\frac{m}{2}\right) + O(\log 2^m) \quad \ldots(1.8.3)$$

$x(m) = T(2^m)$ ...(1.8.4)

Putting the value of $x(m)$ in equation (1.8.3)

$$x(m) = x\left(\frac{m}{2}\right) + O(m) \quad \ldots(1.8.5)$$

Solution of equation (1.8.5) is given as

$a = 1, \quad b = 2, \quad f(n) = O(m)$

$f(n) = O(m^{\log_2 1 + E})$ where $E = 1$

$T(n) = x(m) = O(m) = O(\log n)$

Que 1.9. What do you mean by recursion ? Explain your answer with an example.

Answer

1. Recursion is a process of expressing a function that calls itself to perform specific operation.
2. Indirect recursion occurs when one function calls another function that then calls the first function.
3. Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P.
4. Then P is called recursive procedure. So, the program will not continue to run indefinitely.
5. A recursive procedure must have the following two properties :
   a. There must be certain criteria, called base criteria, for which the procedure does not call itself.
   b. Each time the procedure does call itself, it must be closer to the criteria.
6. A recursive procedure with these two properties is said to be well-defined.
7. Similarly, a function is said to be recursively defined if the function definition refers to itself.

For example :

The factorial function may also be defined as follows :

a. If $n = 0$, then $n! = 1$.

Here, the value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value).

b. If $n > 0$, then $n! = n.\ (n-1)!$

Here, the value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$.

Que 1.10. **What is recursion tree ? Describe.**

AKTU 2013-14, Marks 05

Answer

1. Recursion tree is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded.
2. In a recursion tree, each node represents the cost of a single subproblem.
3. Recursion trees are particularly useful when the recurrence describes the running time of a divide and conquer algorithm.
4. A recursion tree is best used to generate a good guess, which is then verified by the substitution method.
5. It is a method to analyze the complexity of an algorithm by diagramming the recursive function calls.
6. This method can be unreliable.

Que 1.11. **Solve the recurrence :**

$$T(n) = T(n-1) + T(n-2) + 1, \text{ when } T(0) = 0 \text{ and } T(1) = 1.$$

Answer

$$T(n) = T(n-1) + T(n-2) + 1$$

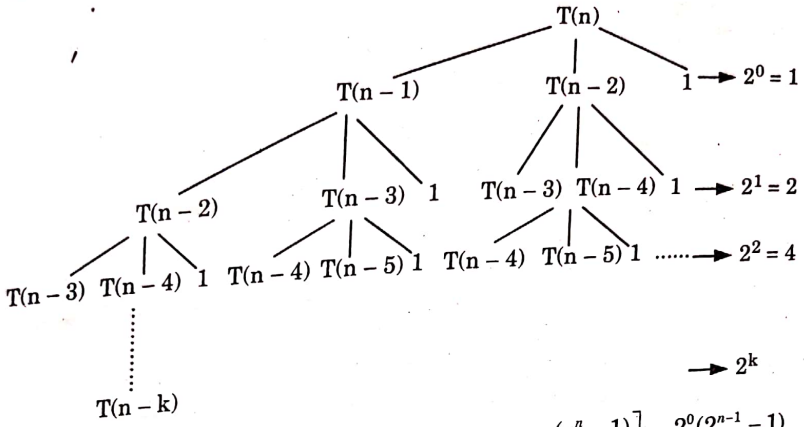
At k^{th} level, $T(1)$ will be equal to 1 when,

$$n - k = 1$$

$$k = n - 1$$

$$= 2^0 + 2^1 + 2^2 + \ldots\ldots 2^k$$

$$= 2^0 + 2^1 + 2^2 + \ldots\ldots 2^{n-1}$$



$$\left[\text{Sum of } n \text{ terms of geometric progression} = \frac{a(r^n - 1)}{r - 1}\right] = \frac{2^0(2^{n-1} - 1)}{2 - 1}$$

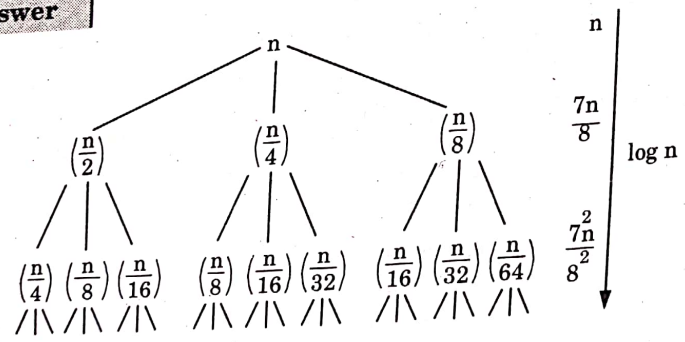
$$= 2^{n-1} - 1 = O(2^n)$$

Que 1.12. **Solve the following recurrences :**

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

AKTU 2014-15, Marks 2.5

Answer



$$T(n) = n + \frac{7n}{8} + \frac{7^2 n}{8^2} + \text{.....+ } \log n \text{ times}$$

$$= \Omega(n \log n)$$

Que 1.13. **Consider the recurrences**

$T(n) = 3T(n/3) + cn$, **and**

$T(n) = 5T(n/4) + n^2$ **where** c **is constant and** n **is the number of inputs. Find the asymptotic bounds.** **AKTU 2013-14, Marks 05**

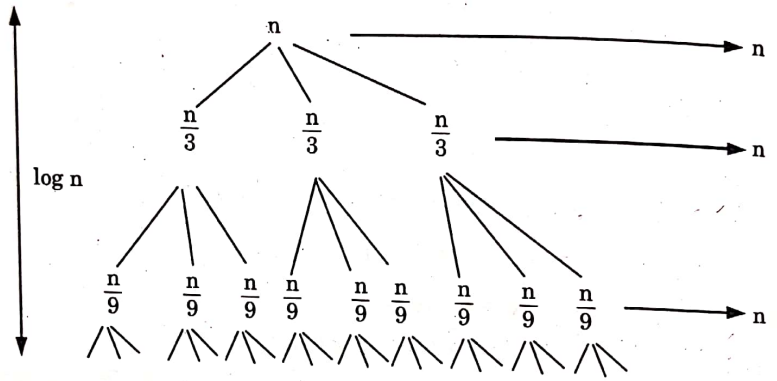
Answer

$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$

we can draw recursion tree for $c \geq 1$

$$T(n) = n + n + n + \ldots\ldots\ldots + \log_3 n \text{ times}$$

$$T(n) = \Omega(n \log n)$$



$$T(n) = n + n + n + \ldots\ldots + \log_3 n \text{ times}$$

$$T(n) = \Omega(n \log n)$$

$$T(n) = 5T\left(\frac{n}{4}\right) + n^2 \qquad \ldots(1.13.1)$$

Comparing equation (1.13.1) with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we get

$$a = 5, b = 4$$

$$f(n) = n^2$$

$$n^{\log_b a} = n^{\log_4 5} = n^{1.160}$$

Now apply cases of Master's theorem as :

$$f(n) = \Omega(n^{\log_b a + E}) = \Omega(n^{1.160 + E})$$

$$= \Omega(n^{1.160 + 0.84}) = \Omega(n^2) \text{ where } E = 0.81$$

Hence, case 3 of Master's theorem is satisfied.

$$T(n) = \theta(f(n))$$

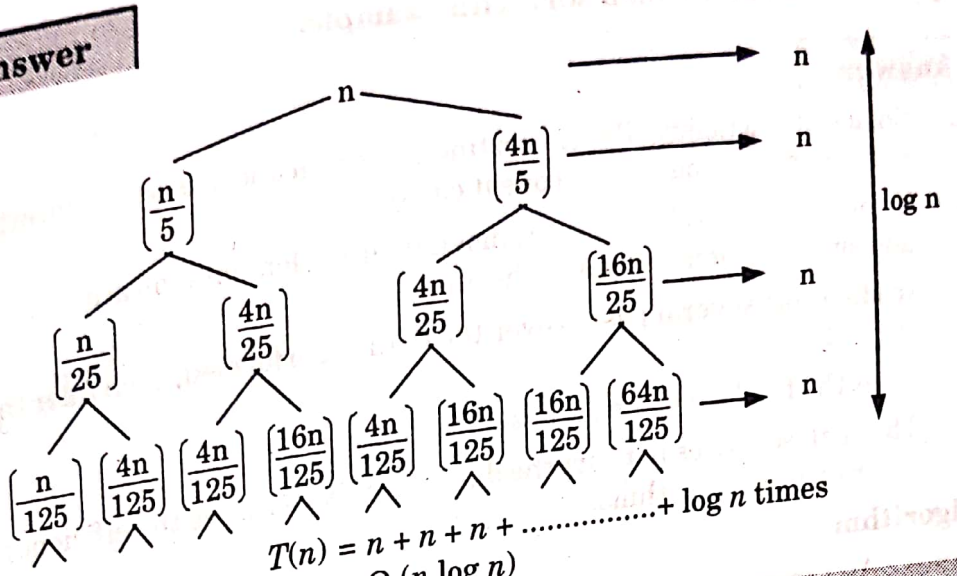
$$T(n) = \theta(n^2)$$

Que 1.14. **Solve the following by recursion tree method**

$$T(n) = n + T(n/5) + T(4n/5)$$

AKTU 2017-18, Marks 10

Answer



$T(n) = n + n + n + \ldots\ldots\ldots\ldots$ + log n times

$= \Omega (n \log n)$

PART-2

Sorting and Order Statistic : Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.

CONCEPT OUTLINE : PART-2

- **Shell Sort :** It is an algorithm that roughly sort the data first and move large elements towards one end and smaller ones towards the other.
  Complexity : $O(n^2)$
- **Heap Sort :** The heap is an array that can be viewed as a complete binary tree. The tree is filled on all levels except the lowest.
  Complexity : $O(n \log n)$
- **Merge Sort :** It works on divide and conquer approach first, it divides a list into two sublist and sort it and then combine as a new sorted one list.
  Complexity : $O(n \log n)$
- **Quick Sort :** It works on the principle of divide and conquer. It works by partitioning a given array.
  Complexity : $O(n^2)$

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.15. Explain shell sort with example.

Answer

1. Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm and we can code it easily.
2. It roughly sorts the data first, moving large elements towards one end and small elements towards the other.
3. In shell sort several passes over the data is performed, each finer than the last.
4. After the final pass, the data is fully sorted.
5. The shell sort does not sort the data itself, it increases the efficiency of other sorting algorithms.

Algorithm :

Input : An array a of length n with array elements numbered 0 to $n-1$.

1. inc ← round (n/2)
2. while inc > 0
3. for i = inc to $n-1$
   temp ← $a[i]$
   $j \leftarrow i$
   while $j \geq$ inc and a [j – inc] > temp
   $a[j] \leftarrow a[j - \text{inc}]$
   $j \leftarrow j - \text{inc}$
   $a[j] \leftarrow$ temp
4. inc ← round (inc/2.2)

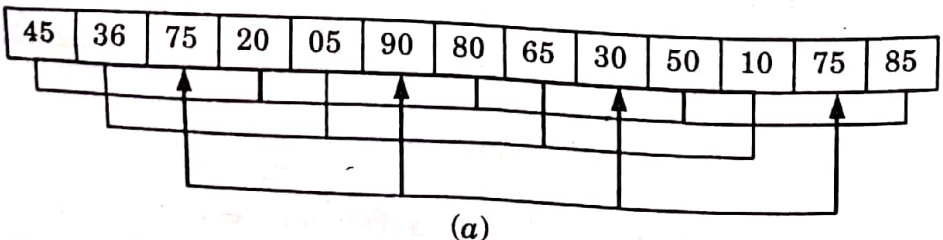
For example :

45	36	75	20	05	90	80	65	30	50	10	75	85

The distance between the elements to be compared is 3. The subfiles generated with the distance of 3 are as follows :

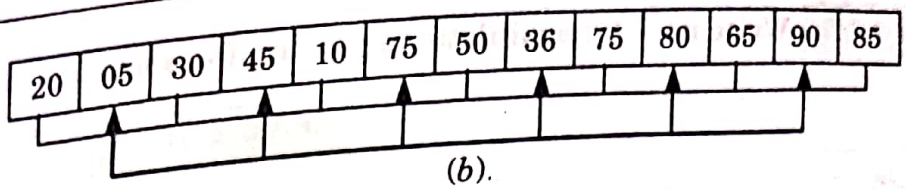
Subfile 1	a[0]	a[3]	a[6]	a[9]	a[12]
Subfile 2	a[1]	a[4]	a[7]	a[10]	
Subfile 3	a[2]	a[5]	a[8]	a[11]	

Input to pass 1 with distance = 3



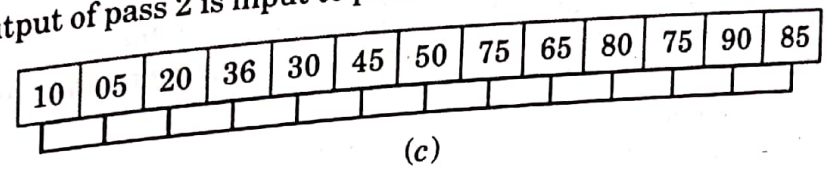
(a)

Output of pass 1 is input to pass 2 and distance = 2



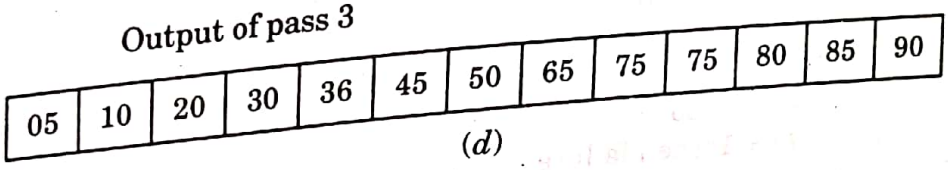
(b).

Output of pass 2 is input to pass 3 and distance = 1



(c)

Output of pass 3



(d)

Fig. 1.15.1.

Que 1.16. Describe any one of the following sorting techniques :

i. **Selection sort**

ii. **Insertion sort**

AKTU 2013-14, Marks 05

Answer

i. **Selection sort (A) :**

1. $n \leftarrow$ length $[A]$
2. for $j \leftarrow 1$ to $n-1$
3. smallest $\leftarrow j$
4. for $i \leftarrow j + 1$ to n
5. if $A[i] < A$ [smallest]
6. then smallest $\leftarrow i$
7. exchange ($A[j]$, A [smallest])

ii. **Insertion_Sort(A) :**

1. for $j \leftarrow 2$ to length$[A]$
2. do key $\leftarrow A[j]$
3. Insert $A[j]$ into the sorted sequence $A[1 \ldots\ldots j-1]$
4. $i \leftarrow j-1$
5. while $i > 0$ and $A[i] >$ key
6. do $A[i+1] \leftarrow A[i]$
7. $i \leftarrow i-1$
8. $A[i+1] \leftarrow$ key

Que 1.17. **Write non-deterministic algorithm for sorting.**

AKTU 2016-17, Marks 10

Answer

Non-deterministic algorithms are algorithm that, even for the same input, can exhibit different behaviours on different runs, iterations and executions.

NSORT(A, B):

1. for $i = 1$ to n do
2. j = choice(1 . . . n)
3. if $B[j]$!= 0 then failure
4. $B[j] = A[i]$
5. endfor
6. for $i = 1$ to $n - 1$ do
7. if $B[i] < B[i + 1]$ then failure
8. endfor
9. print(B)
10. success

Que 1.18. **Explain the concepts of quick sort method and analyze its complexity with suitable example.** **AKTU 2016-17, Marks 10**

Answer

Quick sort :

Quick sort works by partitioning a given array $A[p \ldots r]$ into two non-empty subarray $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that every key in $A[p \ldots q-1]$ is less than or equal to every key in $A[q+1 \ldots r]$. Then the two subarrays are sorted by recursive calls to quick sort.

Quick_Sort (A, p, r)

1. If $p < r$ then
2. $q \leftarrow$ Partition (A, p, r)
3. Recursive call to Quick_Sort $(A, p, q-1)$
4. Recursive call to Quick_Sort $(A, q+1, r)$

As a first step, Quick sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partition (A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j \leftarrow p$ to $r - 1$

4. do if $A[j] \leq x$
5. then $i \leftarrow i + 1$
6. then exchange $A[i] \leftrightarrow A[j]$
7. exchange $A[i + 1] \leftrightarrow A[r]$
8. return $i + 1$

Example : This example shows that how "Pivot" and "Quick sort" work suppose A = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9,]

Sort the array A using quick sort algorithm.

Solution : Given array to be sorted

3	1	4	1	5	9	2	6	5	3	5	8	9

Step 1 : The array is Pivoted about it first element *i.e.*, Pivot $(P) = 3$

3	1	4	1	5	9	2	6	5	3	5	8	9

P

Step 2 : Find first element larger then Pivot (make underline) and find element not larger than pivot from end make over line.

3	1	$\underline{4}$	1	5	9	2	6	5	$\overline{3}$	5	8	9

P Underline Overline

Step 3 : Swap these element and scan again.

3	1	3	1	5	9	2	6	5	4	5	8	9

P

Array after swapping

3	1	3	1	$\underline{5}$	9	$\overline{2}$	6	5	4	5	8	9

P Underline Overline

Apply swapping,

3	1	3	1	2	9	5	6	5	4	5	8	9

Again apply scanning,

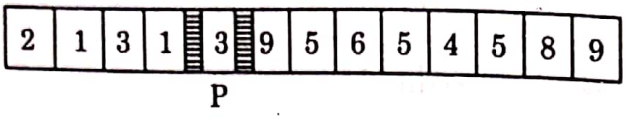
3	1	3	1	$\overline{2}$	$\underline{9}$	5	6	5	4	5	8	9

Overline Underline

The pointers have crossed

i.e., overline on left of underlined

Then, in this situation swap Pivot with overline.



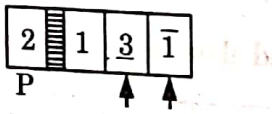
Now, Pivoting process is complete.

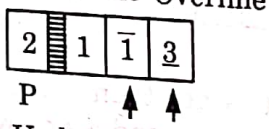
Step 4 : Recursively sort subarrays on each side of Pivot.

Subarray 1 : 2 1 3 1

Subarray 2 : 9 5 6 5 1 5 8 9

First apply Quick sort for subarray 1.

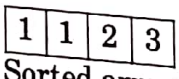




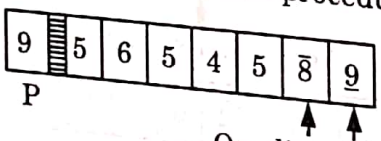
The pointers have crossed.

i.e., overline on left of underlined.

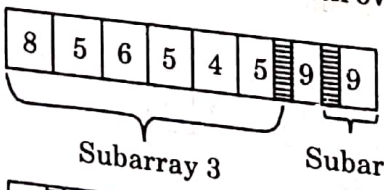
Swap pivot with overline

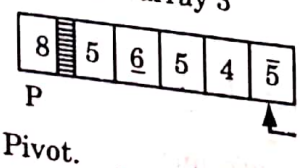


Now, for subarray 2 we apply Quick sort procedure.

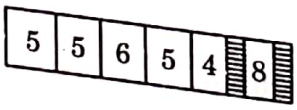


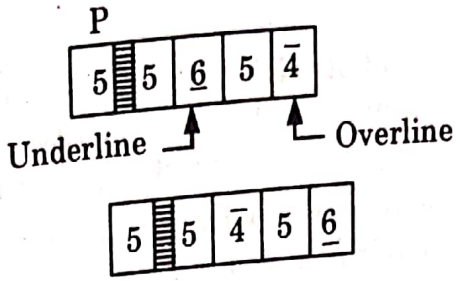
The pointer has crossed. Then swap Pivot with overline.





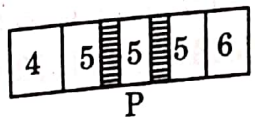
Swap overline with Pivot.



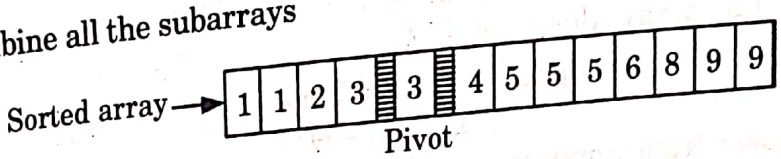


Overline on left of underlined.

Swap Pivot with overline.



Now combine all the subarrays



Analysis of complexity :

i. Worst case :

1. Let $T(n)$ be the worst case time for quick sort on input size n. We have a recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \quad ...(1.18.1)$$

where q ranges from 0 to $n-1$, since the partition produces two regions, each having size $n-1$.

2. Now we assume that $T(n) \le cn^2$ for some constant c.

Substituting our assumption in equation (1.18.1) we get

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$= \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

3. Since the second derivative of expression $q^2 + (n-q-1)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $0 \le q \le n-1$ at one of the endpoints.

4. This gives the bound

$$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$$

5. Continuing with the bounding of $T(n)$ we get

$$T(n) \le cn^2 - c(2n-1) + \Theta(n) \le cn^2$$

6. Since we can pick the constant c large enough so that the $c(2n-1)$ term dominates the $\Theta(n)$ term. We have

$$T(n) = O(n^2)$$

7. Thus, the worst case running time of quick sort is $\Theta(n^2)$.

ii. Average case :

1. If the split induced of RANDOMIZED_PARTITION puts constant fraction of elements on one side of the partition, then the recurrence tree has depth $\Theta(\log n)$ and $\Theta(n)$ work is performed at each level.
2. This is an intuitive argument why the average case running time of RANDOMIZED_QUICKSORT is $\Theta(n \log n)$.
3. Let $T(n)$ denotes the average time required to sort an array of n elements. A call to RANDOMIZED_ QUICKSORT with a 1 element array takes a constant time, so we have $T(1) = \Theta(1)$.
4. After the split RANDOMIZED_QUICKSORT calls itself to sort two subarrays.
5. The average time to sort an array A[1 .. q] is $T[q]$ and the average time to sort an array A[q+1 .. n] is $T[n-q]$. We have

$$T(n) = 1/n\,(T(1) + T(n-1) + {}^{n-1}S_{q=1}\,T(q) \quad T(n-q))) + \Theta(n) \qquad \ldots(1.18.1)$$

We know from worst-case analysis

$$T(1) = \Theta(1) \text{ and } T(n-1) = O(n^2)$$

$$T(n) = 1/n\,(\Theta(1) + O(n^2)) + 1/n\,{}^{n-1}\Theta_{q=1}\,(r(q) + T(n-q)) + Q(n)$$

$$= 1/n\,{}^{n-1}S_{q=1}\,(T(q) + T(n-q)) + Q(n) \qquad \ldots(1.18.2)$$

$$= 1/n\,[2\,{}^{n-1}S_{k=1}(T(k))] + \Theta(n)$$

$$= 2/n\,{}^{n-1}S_{k=1}\,(T(k)) + \Theta(n) \qquad \ldots(1.18.3)$$

6. Solve the above recurrence using substitution method. Assume that $T(n) \le an\,n \log n + b$ for some constants $a > 0$ and $b > 0$.

If we can pick 'a' and 'b' large enough so that $n \log n + b > T(1)$.Then for $n > 1$, we have

$$T(n) \ge {}^{n-1}\Theta_{k=1}\,2/n\,(ak \log k + b) + \Theta(n)$$

$$= 2a/n\,{}^{n-1}S_{k=1}\,k \log k - 1/8(n^2) + 2b/n\,(n-1) + \Theta n \qquad \ldots(1.18.4)$$

At this point we are claiming that

$${}^{n-1}\Theta_{k=1}\,k \log k \le 1/2\,n^2 \log n - 1/8(n^2)$$

Substituting this claim in the equation (1.18.4), we get

$$T(n) \le 2a/n\,[1/2n^2 \log n - 1/8(n^2)] + 2/n\,b(n-1) + \Theta(n)$$

$$\le an \log n - an/4 + 2b + \Theta(n) \qquad \ldots(1.18.5)$$

In the equation (1.18.5), $\Theta(n) + b$ and $an/4$ are polynomials and we can choose 'a' large enough so that $an/4$ dominates $\Theta(n) + b$.
We conclude that QUICKSORT's average running time is $\Theta(n \log n)$.

Que 1.19. Discuss the best case and worst case complexities of quick sort algorithm in detail. **AKTU 2014-15, Marks 05**

Answer

Best case :

1. The best thing that could happen in quick sort would be that each partitioning stage divides the array exactly in half.
2. In other words, the best to be a median of the keys in $A[p .. r]$ every time procedure 'Partition' is called.
3. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.
4. If the procedure 'Partition' produces two regions of size $n/2$, then the recurrence relation is :

$$T(n) \leq T(n/2) + T(n/2) + \Theta(n) \leq 2T(n/2) + \Theta(n)$$

And from case (2) of master theorem

$$T(n) = \Theta(n \log n)$$

Worst case : Refer Q. 1.18, Page 1–17B, Unit-1.

Que 1.20. Explain the concept of merge sort with example.

AKTU 2016-17, Marks 10

Answer

1. Merge sort is a sorting algorithm that uses the idea of divide and conquer.
2. This algorithm divides the array into two halves, sorts them separately and then merges them.
3. This procedure is recursive, with the base criteria that the number of elements in the array is not more than 1.

Algorithm :

MERGE_SORT (a, p, r)

1. if $p < r$
2. then $q \leftarrow \lfloor (p + r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT $(A, q + 1, r)$
5. MERGE (A, p, q, r)

MERGE (A, p, q, r)

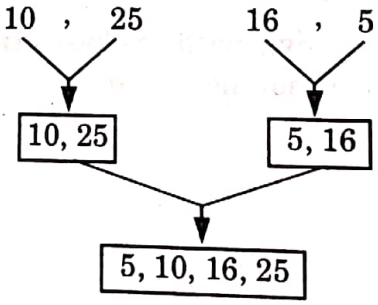
1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays L [1 .....$n_1 + 1$] and R [1.....$n_2 + 1$]
4. for $i = 1$ to n_1
   do
   $L[i] = A\ [p + i - 1]$

endfor

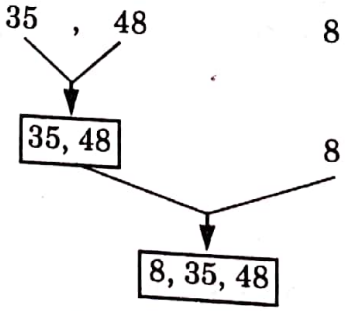
5. for $j = 1$ to n_2
   do
   $R[j] = A[q + j]$
   endfor
6. $L[n_1 + 1] = \infty$, $R[n_2 + 1] = \infty$
7. $i = 1, j = 1$
8. for $k = p$ to r
   do
   if $L[i] \leq R[j]$
   then $A[k] \leftarrow L[i]$
   $i = i + 1$
   else $A[k] = R[j]$
   $j = j + 1$
   endif
   endfor
9. exit

Example :
10, 25, 16, 5, 35, 48, 8

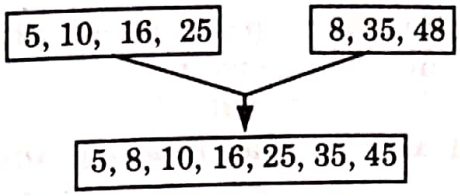
1. **Divide into two halves :** [10, 25, 16, 5] [35, 48, 8]
2. **Consider the first part :** 10, 25, 16, 5 again divide into two sub-arrays



3. **Consider the second half :** 35, 48, 5 again divide into two sub-arrays



4. Merge these two sorted sub-arrays,



This is the sorted array.

Que 1.21. Determine the best case time complexity of merge sort algorithm.

Answer

1. The best case of merge sort occurs when the largest element of one array is smaller than any element in the other array.
2. For this case only $n/2$ comparisons of array elements are made.
3. Merge sort comparisons are obtained by the recurrence equation of the recursive calls used in merge sort.
4. As it divides the array into half so the recurrence function is defined as :

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = 2T\left(\frac{n}{2}\right) + n \quad \text{...(1.21.1)}$$

5. By using variable k to indicate depth of the recursion, we get

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \quad \text{...(1.21.2)}$$

6. For the best case there are only $n/2$ comparisons hence equation (1.21.2) can be written as

$$T(n) = 2^k \left(\frac{n}{2^k}\right) + k\frac{n}{2}$$

7. At the last level of recursion tree

$$2^k = n$$

$$k = \log_2 n$$

8. So the recurrence function is defined as :

$$T(n) = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \frac{n}{2}\log_2 n$$

$$= nT(1) + \frac{n}{2}\log_2 n = \frac{n}{2}\log_2 n + n$$

$$T(n) = O(n \log_2 n)$$

Hence, the best case complexity of merge sort is $O(n \log_2 n)$.

Que 1.22. Explain heap sort algorithm with its analysis.

OR

What is the running time of heap sort on an array A of length n that is already sorted in increasing order ?

OR

Discuss the complexity of Max-heapify and Build Max Heap procedures. **AKTU 2014-15, Marks 10**

Answer

1. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.
2. The general approach of heap sort is as follows :
   a. From the given array, build the initial max heap.
   b. Interchange the root (maximum) element with the last element.
   c. Use repetitive downward operation from root node to rebuild the heap of size one less than the starting.
   d. Repeat step a and b until there are no more elements.

Analysis of heap sort :

Complexity of heap sort for all cases is $O(n \log_2 n)$.

MAX-HEAPIFY (A, i) :

1. $i \leftarrow$ left $[i]$
2. $r \leftarrow$ right $[i]$
3. if $l \leq$ heap-size $[A]$ and $A[l] > A[i]$
4. then largest $\leftarrow 1$
5. else largest $\leftarrow i$
6. if $r \leq$ heap-size $[A]$ and $A[r] > A$ [largest]
7. then largest $\leftarrow r$
8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A$[largest]
10. MAX-HEAPIFY [A, largest]

HEAP-SORT(A) :

1. BUILD-MAX-HEAP (A)
2. for $i \leftarrow$ length $[A]$ down to 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. heap-size $[A] \leftarrow$ heap-size $[A] - 1$
5. MAX-HEAPIFY (A, 1)

BUILD-MAX-HEAP (A)

1. heap-size (A) $\leftarrow$ length $[A]$
2. for $i \leftarrow$ (length $[A]/2$) down to 1 do

3. MAX-Heapify (A, i)

We can build a heap from an unordered array in linear time.

The HEAPSORT procedure takes time $O(n \log n)$ since the call to BUILD_HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-Heapify takes time $O(\log n)$.

Que 1.23. Sort the following array using heap sort techniques : {5, 13, 2, 25, 7, 17, 20, 8, 4}. Discuss its worst case and average case time complexities. **AKTU 2013-14, Marks 05**

Answer

Given array is : [5, 13, 2, 25, 7, 17, 20, 8, 4]

First we call Build-Max heap

heap size $[A] = 9$

$i = 4$ to 1, call MAX HEAPIFY (A, i)

so,

i.e., first we call MAX HEAPIFY $(A, 4)$

$A[l] = 8, A[i] = 25, A[r] = 4$

$A[i] > A[l]$

$A[i] > A[r]$

Now call MAX HEAPIFY $(A, 3)$

$A[i] = 2, A[l] = 17, A[r] = 20$

$A[l] > A[i]$

$\therefore$ largest = 6

$A[r] > A$ [largest]

$20 > 17$

$\therefore$ largest = 7

$\therefore$ largest $\neq i$

$\therefore$ $A[i] \leftrightarrow A$ [largest]

$A[i] > A[l]$

$A[i] > A[r]$

Now call MAX HEAPIFY $(A, 2)$

$A[i] < A[l]$

so, largest = 4

A [largest] $> A[r]$

$i \neq$ largest, so $A[i] \leftrightarrow A$ [largest]

$\therefore$

Now, $A[i] > A[l]$

$A[i] > A[r]$

We call MAX HEAPIFY $(A, 1)$

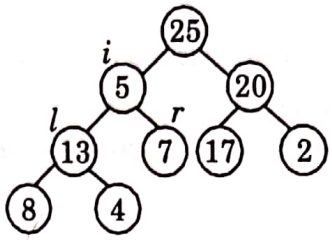
$A[i] < A[l]$

largest = 2

A [largest] $> A[r]$, and largest $\neq i$

$A[i] \leftrightarrow A$ [largest]

largest = 2, so $i = 2$



$A[i] < A[l]$, then largest = 4

A [largest] > A [r], largest $\neq i$

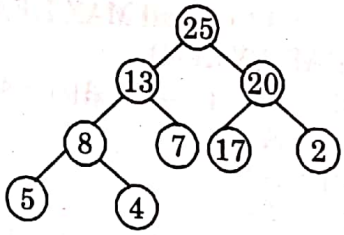
A [i] $\leftrightarrow$ A [largest]

Now, $A[i] < A$ [l]

largest = 8, A [largest] > A [r]

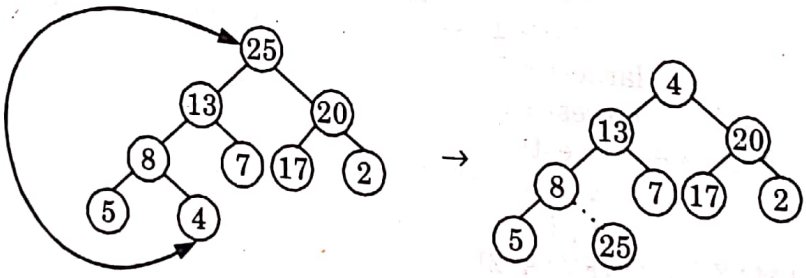
largest $\neq i$, A [largest] $\leftrightarrow A$ [i]

so final tree after Build MAX HEAPIFY



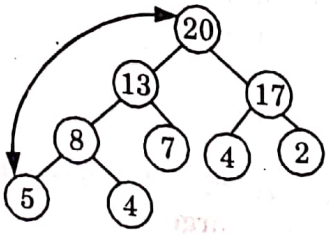
Now i = 9 down to 2, exchange A [1] and A [9] and size = size –1 and call MAX HEAPIFY (A, 1) each time.

exchanging A [1] $\leftrightarrow$ A [9]

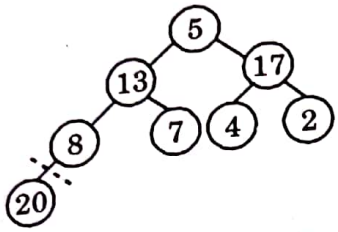


4	13	20	8	7	17	2	5	25

Now call MAX HEAPIFY (A, 1) we get

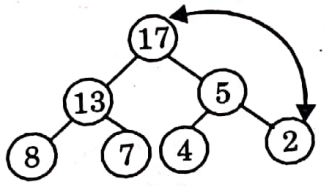


Now exchange A[1] and A[8] and size = size –1 = 8 – 1 = 7

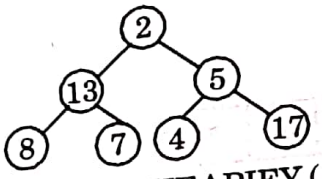


5	13	17	8	7	4	2	20

Again call MAX HEAPIFY (A, 1), we get

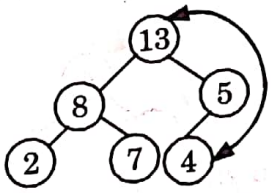


exchange A [1] and A[7] and size = size – 1 = 7 – 1 = 6

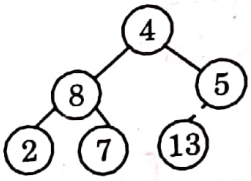


2	13	5	8	7	4	17

Again call MAX HEAPIFY (A, 1), we get

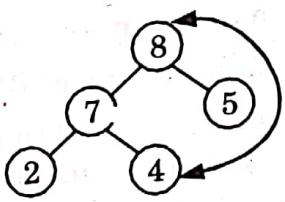


exchange A [1] and A[6] and now size = 6 –1 = 5

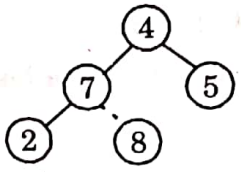


4	8	5	2	7	13

Again call MAX HEAPIFY (A, 1)

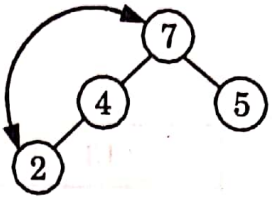


exchange A [1] and A [5] and now size = 5 –1 = 4

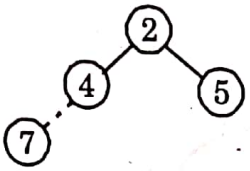


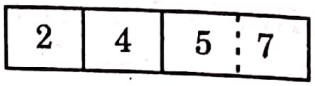
4	7	5	2	8

Again, call MAX HEAPIFY (A, 1)

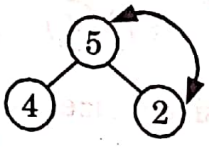


exchange A [1] and A [4] and size = 4–1 = 3

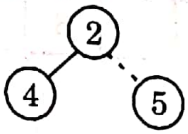


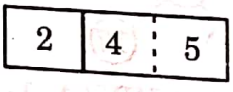


call MAX HEAPIFY (A, 1)

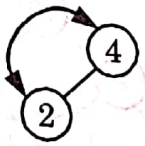


exchange A [1] and A [3], size = 3 – 1 = 2

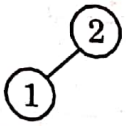


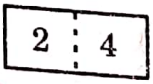


call MAX HEAPIFY (A, 1)

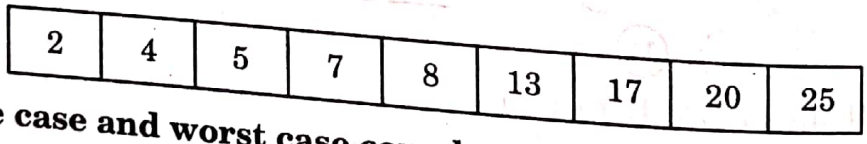


exchange A [1] and A [2] and size = 2 –1 = 1





Thus, sorted array :



Average case and worst case complexity :

1. We have seen that the running time of BUILD-HEAP is O(n).
2. The heap sort algorithm makes a call to BUILD-HEAP for creating a (max) heap, which will take O(n) time and each of the (n – 1) calls to MAX-HEAPIFY to fix up the new heap (which is created after exchanging the root and by decreasing the heap size).
3. We know 'MAX-HEAPIFY' takes time O(log n).
4. Thus the total running time for the heap sort is O(n log n).

Que 1.24. **What is heap sort ? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.** **AKTU 2015-16, Marks 10**

Answer

Heap sort : Refer Q. 1.22, Page 1–24B, Unit-1.

Numerical : Since the given problem is already in sorted form. So, there is no need to apply any procedure on given problem.

Que 1.25. **Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array $A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$** **AKTU 2017-18, Marks 10**

Answer

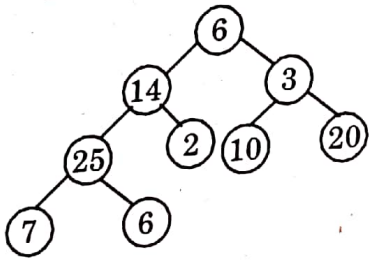
Heap sort : Refer Q. 1.22, Page 1–24B, Unit-1.

Numerical :

Originally the given array is : [6, 14, 3, 25, 2, 10, 20, 7, 6]

First we call Build-Max heap

heap size $[A] = 9$



so, $i = 4$ to 1, call MAX HEAPIFY (A, i)

i.e., first we call MAX HEAPIFY $(A, 4)$

$A\ [l] = 7, A\ [i] = A\ [4] = 25, A\ [r] = 6$

$l \leftarrow$ left $[4] = 8$

$r \leftarrow$ right $[4] = 9$

$8 \leq 9$ and $7 > 25$ (False)

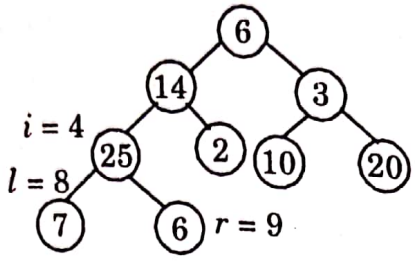
Then, largest $\leftarrow 4$

$9 \leq 9$ and $6 > 25$ (False)

Then, largest = 4

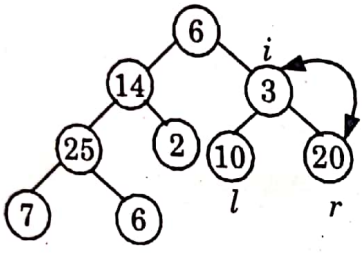
$A\ [i] \leftrightarrow A\ [4]$

Now call MAX HEAPIFY $(A, 2)$

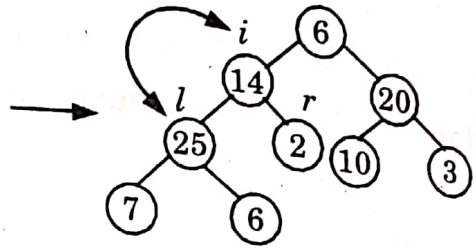


(i)

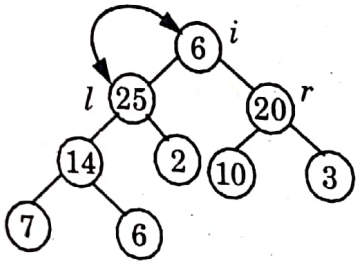
Similarly for $i = 3, 2, 1$ we get the following heap tree.



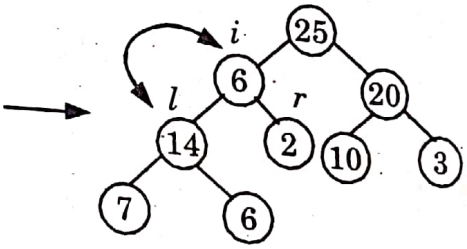
(ii)



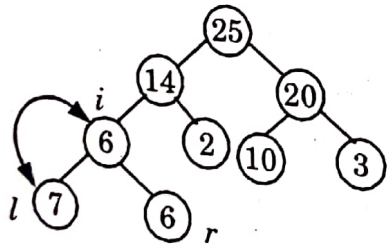
(iii)



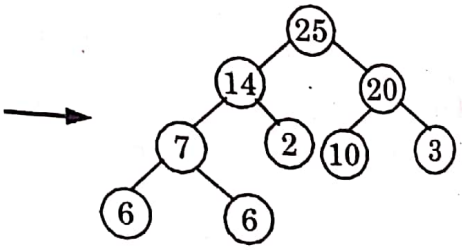
(iv)



(v)

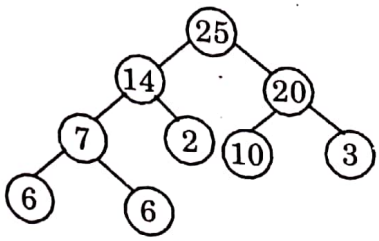


(vi)



(vii)

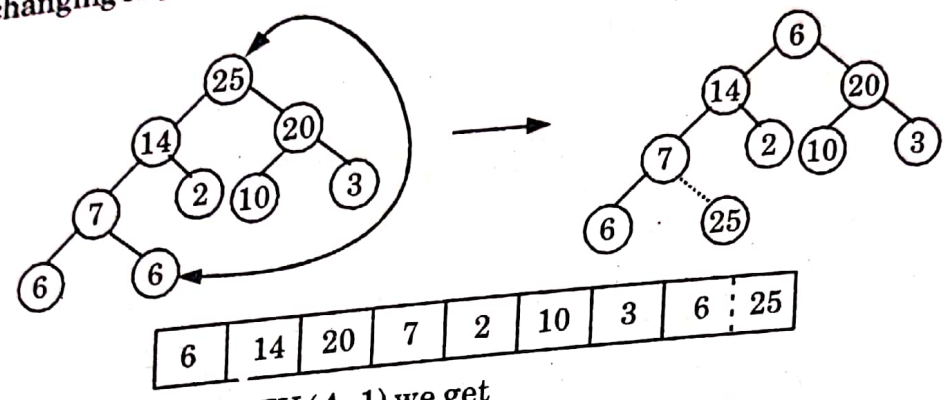
So final tree after BUILD-MAX HEAP is



(viii)

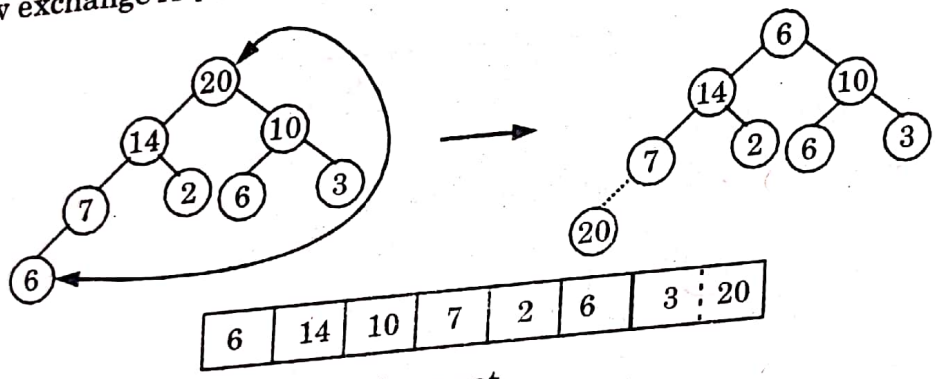
Now $i = 9$ down to 2, and size = size – 1 and call MAX HEAPIFY $(A, 1)$ each time.

exchanging $A[1] \leftrightarrow A[9]$



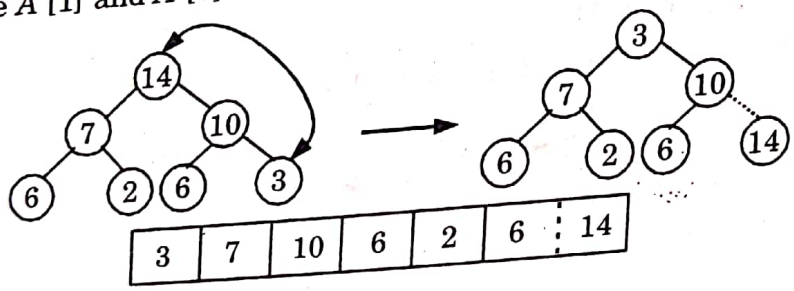
Now call MAX HEAPIFY $(A, 1)$ we get

Now exchange $A[1]$ and $A[8]$ and size = 8 – 1 = 7



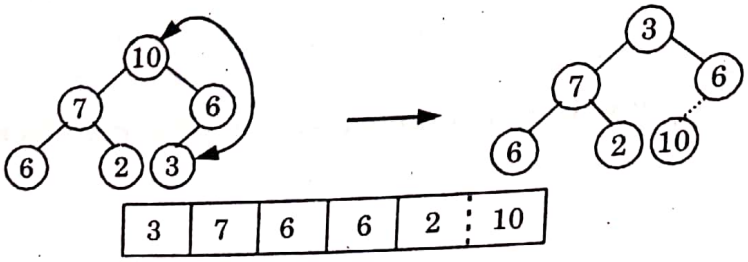
Again call MAX HEAPIFY $(A, 1)$, we get

exchange $A[1]$ and $A[7]$ and size = 7 – 1 = 6

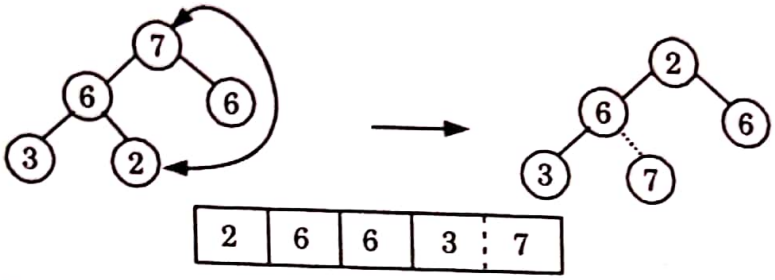


Again call MAX HEAPIFY $(A, 1)$, we get

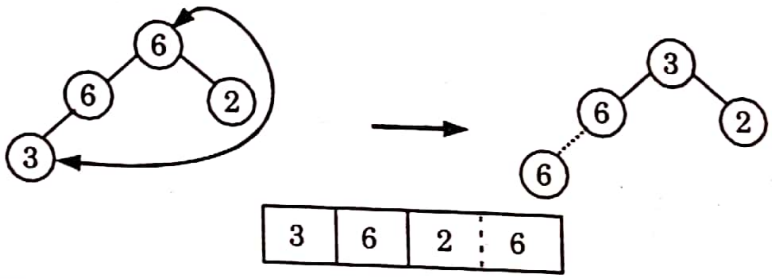
exchange $A[1]$ and $A[6]$ and now size = 6 – 1 = 5



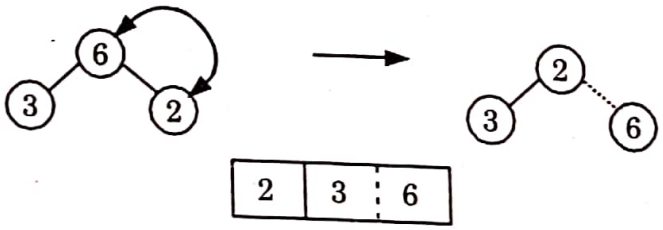
Again call MAX HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [5] and now size = 5 – 1 = 4



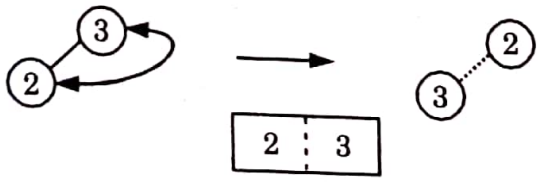
Again, call MAX HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [4] and size = 4 – 1 = 3



call MAX HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [3], size = 3 – 1 = 2



call MAX HEAPIFY (*A*, 1)
exchange *A* [1] and *A* [2] and size = 2 – 1 = 1



Thus, sorted array :

2	3	6	6	7	10	14	20	25

Que 1.26. **How will you compare various sorting algorithms ?**

Answer

Name	Average case	Worst case	Stable	Method	Other notes
Selection sort	$O(n^2)$	$O(n^2)$	No	Selection	Can be implemented as a stable sort
Insertion sort	$O(n^2)$	$O(n^2)$	Yes	Insertion	Average case is also $O(n + d)$, where d is the number of inversion
Shell sort	–	$O(n \log^2 n)$	No	Insertion	No extra memory required
Merge sort	$O(n \log n)$	$O(n \log n)$	Yes	Merging	Recursive, extra memory required
Heap sort	$O(n \log n)$	$O(n \log n)$	No	Selection	Recursive, extra memory required
Quick sort	$O(n \log n)$	$O(n^2)$	No	Partitioning	Recursive, based on divide conquer technique

Que 1.27. Explain the counting sort algorithm.

Answer

Counting sort is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.

Algorithm :

Counting_Sort(A, B, k)

1. let $C[0..k]$ be a new array
2. for $i \leftarrow 0$ to k
3. $C[i] \leftarrow 0$
4. for $j \leftarrow 1$ to length[A]
5. do $C[A[j]] \leftarrow C[A[j]] + 1$
   // $C[i]$ now contains the number of elements equal to i.
6. for $i \leftarrow 1$ to k
7. do $C[i] \leftarrow C[i] + C[i - 1]$
   // $C[i]$ now contains the number of elements less than or equal to i.
8. for $j \leftarrow$ length[A] down to 1
9. do $B[C[A[j]]] \leftarrow A[j]$
10. $C[A[j]] \leftarrow C[A[j]] - 1$

Que 1.28. **What is the time complexity of counting sort ? Illustrate the operation of counting sort on array $A = \{1, 6, 3, 3, 4, 5, 6, 3, 4, 5\}$.**

AKTU 2014-15, Marks 10

Answer

Time complexity of counting sort is $O(n)$.

	1	2	3	4	5	6	7	8	9	10
A	1	6	3	3	4	5	6	3	4	5

Step 1 :

$i = 0$ to 6

$C[i] \leftarrow 0$

$k = 6$ (largest element in array A)

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

Step 2 :

$j = 1$ to 10

$C[A[j]] \leftarrow C[A[j]] + 1$ ∵ length $[A] = 10$

For $j = 1$

$C[A[1]] \leftarrow C[1] + 1 = 0 + 1 = 1$

$C[1] \leftarrow 1$

	0	1	2	3	4	5	6
C	0	1	0	0	0	0	0

For $j = 2$

$C[A[2]] \leftarrow C[6] + 1 = 0 + 1 = 1$

$C[6] \leftarrow 1$

	0	1	2	3	4	5	6
C	0	1	0	0	0	0	1

Similarly for $j = 5, 6, 7, 8, 9, 10$

	0	1	2	3	4	5	6
C	0	1	0	3	2	2	2

Step 3 :

For $i = 1$ to 6

$C[i] \leftarrow C[i] + C[i - 1]$

For $i = 1$

$C[1] \leftarrow C[1] + C[0]$

$C[1] \leftarrow 1 + 0 = 1$

	0	1	2	3	4	5	6
C	0	1	0	3	2	2	2

For $i = 2$

$C[2] \leftarrow C[2] + C[1]$

$C[1] \leftarrow 1 + 0 = 1$

	0	1	2	3	4	5	6
C	0	1	1	3	2	2	2

Similarly for $i = 4, 5, 6$

	0	1	2	3	4	5	6
C	0	1	1	4	6	8	10

Step 4 :

For $j = 10$ to 1

$B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

j	$A[j]$	$C[A[j]]$	$B[C[A[j]]] \leftarrow A[j]$	$C[A[j]] \leftarrow C[A[j]] - 1$
10	5	8	$B[8] \leftarrow 5$	$C[5] \leftarrow 7$
9	4	6	$B[6] \leftarrow 4$	$C[4] \leftarrow 5$
8	3	4	$B[4] \leftarrow 3$	$C[3] \leftarrow 3$
7	6	10	$B[10] \leftarrow 6$	$C[6] \leftarrow 9$
6	5	7	$B[7] \leftarrow 5$	$C[5] \leftarrow 6$
5	4	5	$B[5] \leftarrow 4$	$C[4] \leftarrow 4$
4	3	3	$B[3] \leftarrow 3$	$C[3] \leftarrow 2$
3	3	2	$B[2] \leftarrow 3$	$C[3] \leftarrow 1$
2	6	9	$B[9] \leftarrow 6$	$C[6] \leftarrow 8$
1	1	1	$B[1] \leftarrow 1$	$C[1] \leftarrow 0$

	1	2	3	4	5	6	7	8	9	10
B	1	3	3	3	4	4	5	5	6	6

Que 1.29. Write the bucket sort algorithm.

Answer

1. The bucket sort is used to divide the interval [0, 1] into n equal-sized sub-intervals, or bucket, and then distribute the n-input numbers into the bucket.
2. Since the inputs are uniformly distributed over [0, 1], we do not except many numbers to fall into each bucket.
3. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each.
4. The code assumes that input is in n-element array A and each element in A satisfies $0 \leq A[i] \leq 1$. We also need an auxiliary array $B[0 \ldots n-1]$ for linked-list (buckets).

BUCKET_SORT (A)

1. $n \leftarrow$ length $[A]$
2. For $i \leftarrow 1$ to n
3. do Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. For $i \leftarrow 0$ to $n-1$
5. do Sort list $B[i]$ with insertion sort
6. Concatenate the lists $B[0], B[1], \ldots. B[n-1]$ together in order.

Que 1.30. What do you mean by stable sort algorithms ? Explain it with suitable example.

Answer

1. A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input sorted array.
2. A stable sort is one where the initial order of equal items is preserved.
3. Some sorting algorithms are naturally stable, some are unstable, and some can be made stable with care.
4. Stability is important when we want to sort by multiple fields, for example, sorting a list of task assignments first by priority and then by assignee (in other words, assignments of equal priority are sorted by assignee).
5. One easy way to do this is to sort by assignee first, then take the resulting sorted list and sort that by priority.
6. This only works if the sorting algorithm is stable otherwise; the sortedness-by-assignee of equal-priority items is not preserved.
7. For example, if sort the words "apple", "tree" and "pink" by length, then "tree", "pink", "apple" is a stable sort but "pink", "tree", apple" is not.
8. Merge sort is a very common choice of stable sorts, achieved by favouring the leftmost item in each merge step (only if item_right < item_left put item_right first).
9. Radix sort is another of the stable sorting algorithms.

Que 1.31. Write a short note on radix sort.

Answer

1. Radix sort is a sorting algorithm which consists of list of integers or words and each has d-digit.
2. We can start sorting on the least significant digit or on the most significant digit.
3. On the first pass entire numbers sort on the least significant digit (or most significant digit) and combine in a array.
4. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combine in an array and so on.

RADIX_SORT (A, d)

1. for $i \leftarrow 1$ to d do

2. use a stable sort to sort array A on digit i

// counting sort will do the job

The code for radix sort assumes that each element in the n-element array A has d-digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

Analysis :

1. The running time depends on the table used as an intermediate sorting algorithm.
2. When each digit is in the range 1 to k, and k is not too large, COUNTING_SORT is the obvious choice.
3. In case of counting sort, each pass over n d-digit numbers takes $\Theta(n + k)$ time.
4. There are d passes, so the total time for radix sort is $\theta(n + k)$ time. There are d passes, so the total time for radix sort is $\Theta(dn + kd)$. When d is constant and $k = \Theta(n)$, the radix sort runs in linear time.

For example : This example shows how radix sort operates on seven 3-digit number.

Table 1.31.1.

Input	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the table 1.31.1, the first column is the input and the remaining shows the list after successive sorts on increasingly significant digits position.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What do you mean by algorithm ? Write its characteristics.

Ans. Refer Q. 1.1.

Q. 2. Write short note on asymptotic notations.

Ans. Refer Q. 1.3.

Q. 3. Explain shell sort with example.
Ans. Refer Q. 1.15.

Q. 4. Discuss quick sort method and analyze its complexity.
Ans. Refer Q. 1.18.

Q. 5. Explain the concept of merge sort with example.
Ans. Refer Q. 1.20.

Q. 6. Write short note on heap sort algorithm with its analysis.
Ans. Refer Q. 1.22.

Q. 7. What is radix sort ?
Ans. Refer Q. 1.31.

2 UNIT

Advanced Data Structure

PART-1

Red-Black Trees, B-trees.

CONCEPT OUTLINE : PART-1

- **Red-black tree :**
  a. A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black.
  b. It is a type of self-balancing binary search tree.
- **B-tree :**
  a. B-tree is a tree data structure that keeps data sorted and allows insertion and deletion in logarithmic amortized time.
  b. In B-trees, internal nodes can have a variable number of child nodes within some predefined range.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.1. Define a red-black tree with its properties. Explain the insertion operation in a red-black tree.

Answer

Red-black tree :

A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black. It is a self-balancing Binary Search Tree (BST) where every node follows following properties :

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendent leave contain the same number of black nodes.

Insertion :

i. We begin by adding the node as we do in a simple binary search tree and colouring it red.

RB-INSERT(T, z)

1. $y \leftarrow$ nil $[T]$
2. $x \leftarrow$ root $[T]$
3. while $x \neq$ nil $[T]$
4. do $y \leftarrow x$
5. if key$[z]$ < key $[x]$
6. then $x \leftarrow$ left $[x]$
7. else $x \leftarrow$ right $[x]$
8. $p[z] \leftarrow y$
9. if y = nil $[T]$
10. then root $[T] \leftarrow z$
11. else if key $[z]$ < key$[y]$
12. then left $[y] \leftarrow z$
13. else right $[y] \leftarrow z$
14. left $[z] \leftarrow$ nil$[T]$
15. right $[z] \leftarrow$ nil$[T]$
16. colour $[z] \leftarrow$ RED
17. RB-INSERT-FIXUP(T, z)

ii. Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

RB-INSERT-FIXUP(T, z)

1. while colour $[p[z]]$ = RED
2. do if $p[z]$ = left$[p\ [p\ [z]]]$
3. then $y \leftarrow$ right$[p\ [p\ [z]]]$
4. if colour$[y]$ = RED ⇒ case 1
5. then colour$[p\ [z]] \leftarrow$ BLACK ⇒ case 1
6. colour$[y] \leftarrow$ BLACK ⇒ case 1
7. colour$[p\ [p\ [z]]] \leftarrow$ RED ⇒ case 1
8. $z \leftarrow p\ [p\ [z]]$
9. else if z = right$[p[z]]$ ⇒ case 2
10. then $z \leftarrow p[z]$ ⇒ case 2
11. LEFT-ROTATE(T, z) ⇒ case 3
12. colour$[p[z]] \leftarrow$ BLACK ⇒ case 3
13. colour$[p[p[z]]] \leftarrow$ RED ⇒ case 3
14. RIGHT-ROTATE(T, $p[p[z]]$)
15. else (same as then clause with "right" and "left" exchanged)
16. colour[root$[T]] \leftarrow$ BLACK

Cases of RB-tree for insertion :

Case 1 : z's uncle is red :

$$P[z] = \text{left}[p[p[z]]]$$

then uncle ← right[p[p[z]]]

a. change z's grandparent to red.

b. change z's uncle and parent to black.

c. change z to z's grandparent.

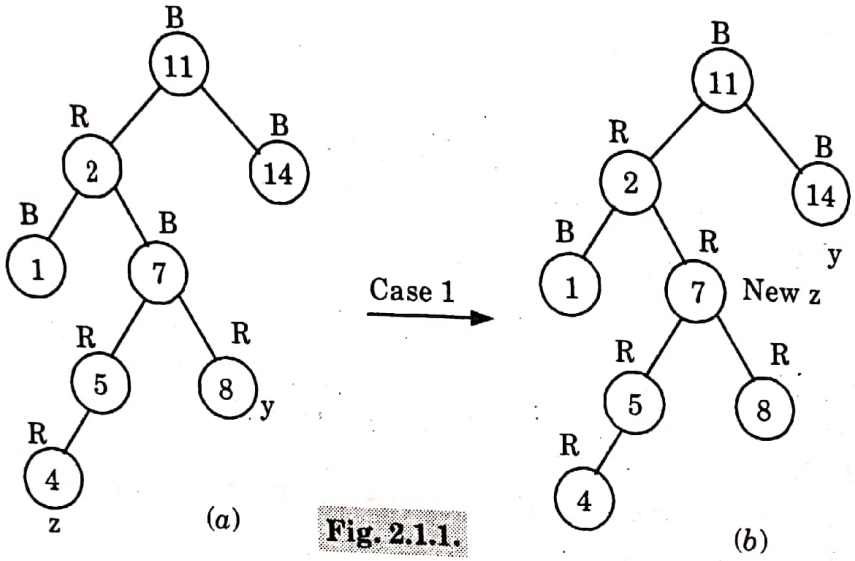


Fig. 2.1.1.

Now, in this case violation of property 4 occurs, because z's uncle y is red, then case 1 is applied.

Case 2 : z's uncle is black, z is the right of its parent :

a. Change z to z's parent.

b. Rotate z's parent left to make case 3.

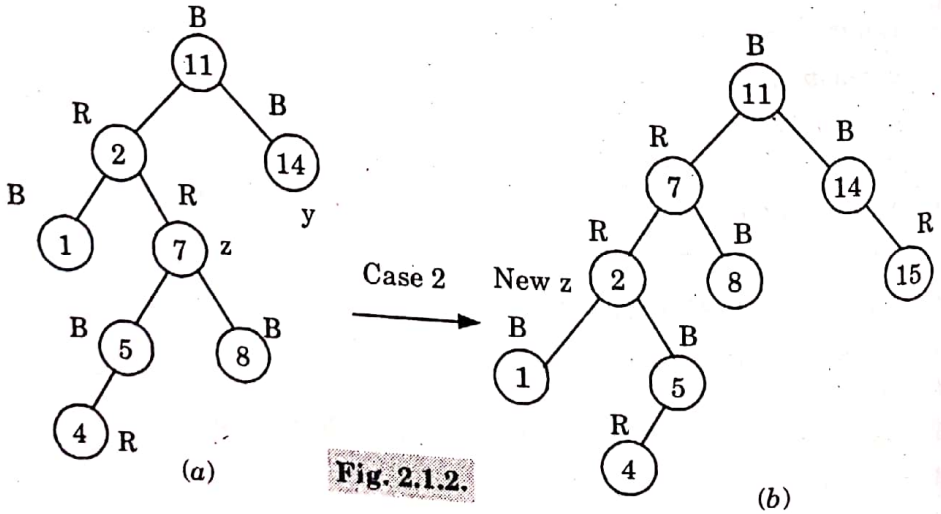


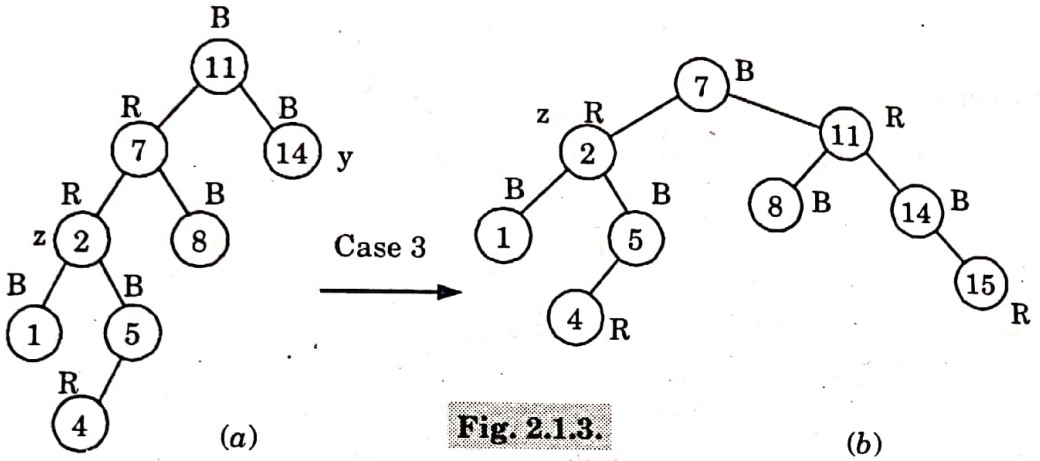
Fig. 2.1.2.

Case 3 : z's uncle is black, z is the left child of its parent :

a. Set z's parent black.

b. Set z's grandparent to red.

c. Rotate z's grandparent right.



(a) **Fig. 2.1.3.** (b)

Que 2.2. What are the advantages of red-black tree over binary search tree ? Write algorithms to insert a key in a red-black tree insert the following sequence of information in an empty red-black tree 1, 2, 3, 4, 5, 5. **AKTU 2014-15, Marks 10**

Answer

Advantages of RB-tree over binary search tree :

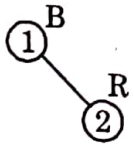
1. The main advantage of red-black trees over AVL trees is that a single top-down pass may be used in both insertion and deletion operations.
2. Red-black trees are self-balancing while on the other hand, simple binary search trees are unbalanced.
3. It is particularly useful when inserts and/or deletes are relatively frequent.
4. Time complexity of red-black tree is O(log n) while on the other hand, a simple BST has time complexity of O(n).

Algorithm to insert a key in a red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :

Insert 1 :

Insert 2 :



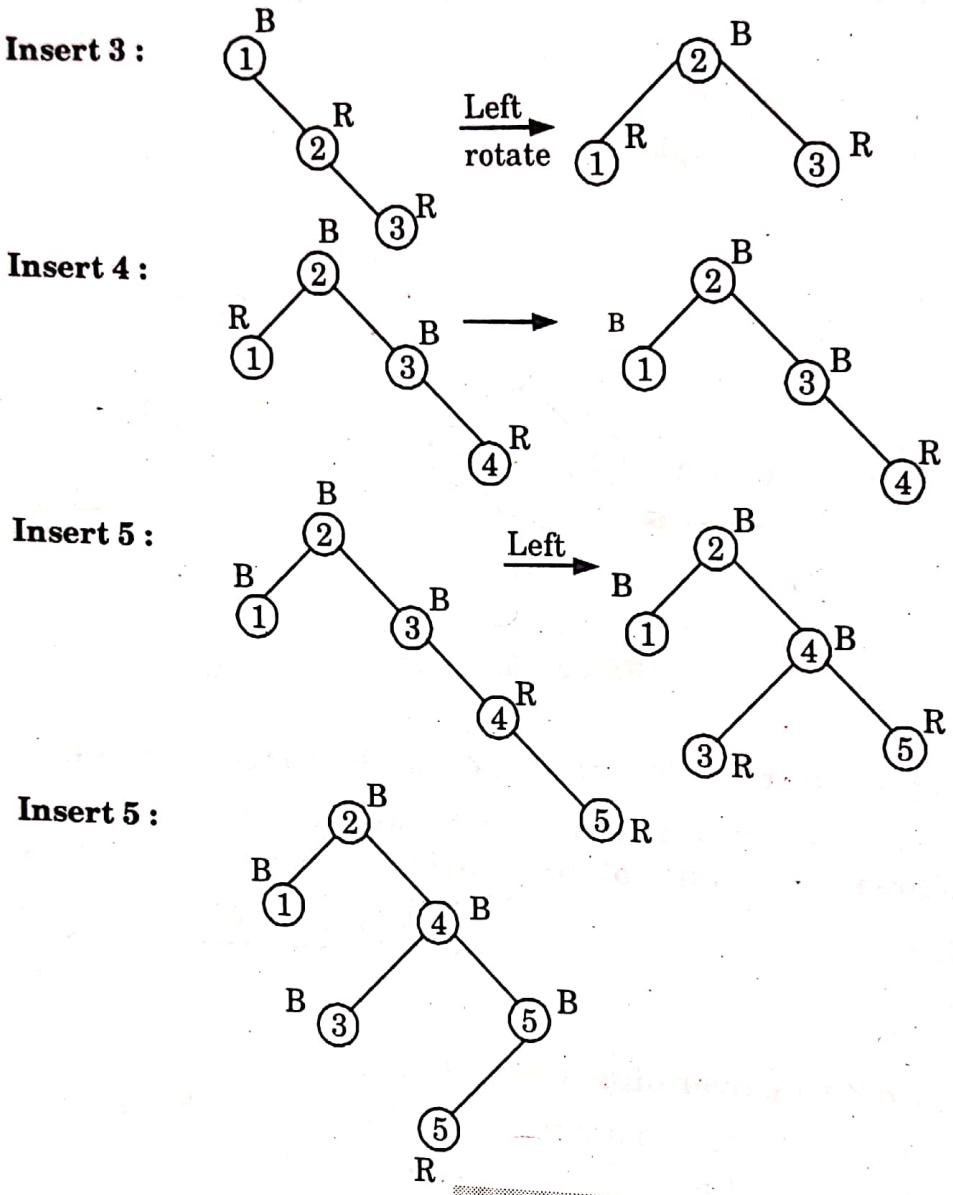


Fig. 2.2.1.

Que 2.3. **Explain red-black tree. Show steps of inserting the keys 41, 38, 31, 12, 19, 8 into initially empty red-black tree.**

AKTU 2013-14, Marks 10

OR

What is red-black tree ? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.

AKTU 2017-18, Marks 10

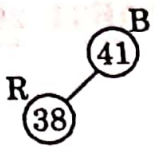
Answer

Red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

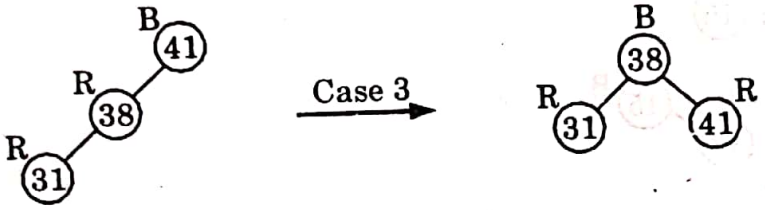
Numerical :

Insert 41 :

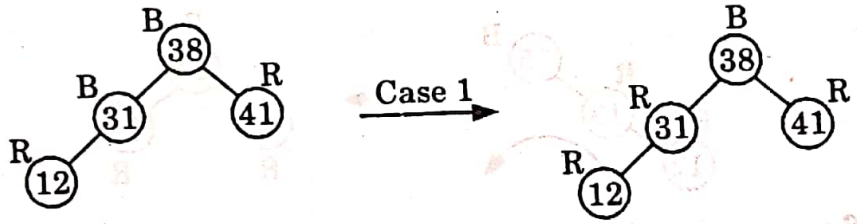
Insert 38 :



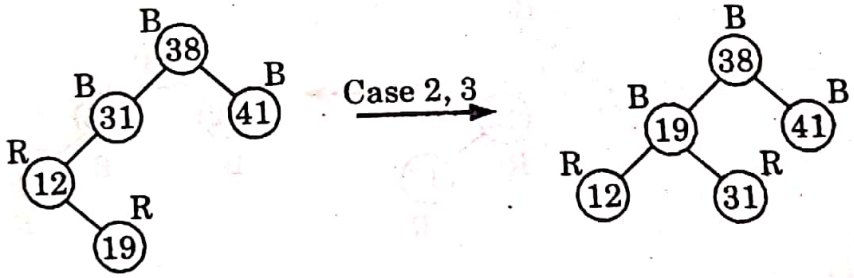
Insert 31 :



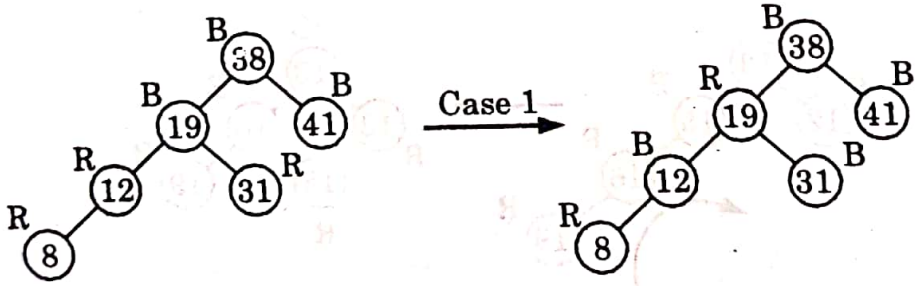
Insert 12 :



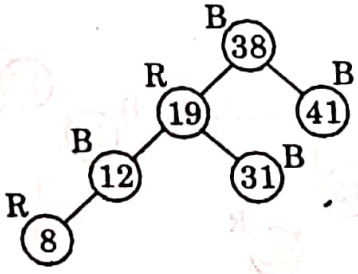
Insert 19 :



Insert 8 :



Thus final tree is



Que 2.4. **Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.**

AKTU 2016-17, Marks 10

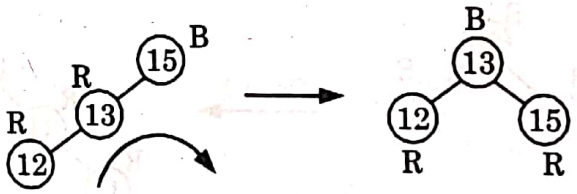
Answer

Insertion :

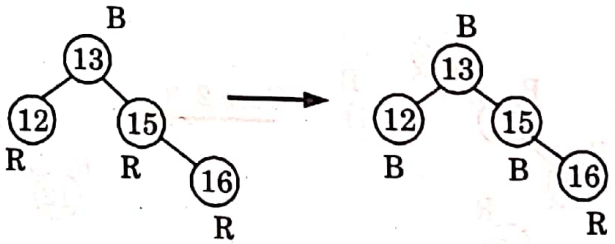
Insert 15 :

Insert 13 :

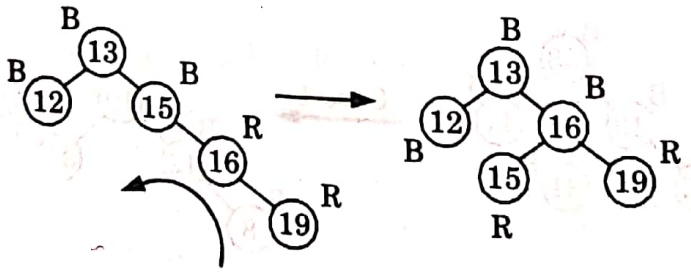
Insert 12 :



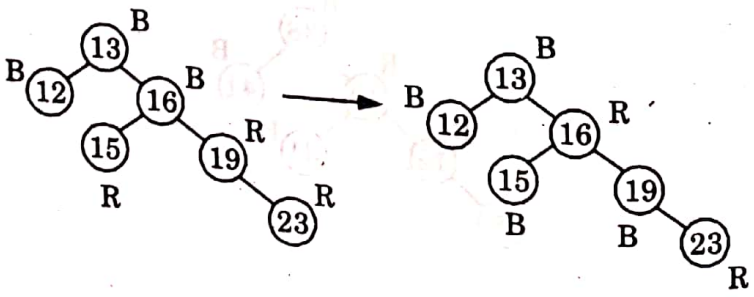
Insert 16 :



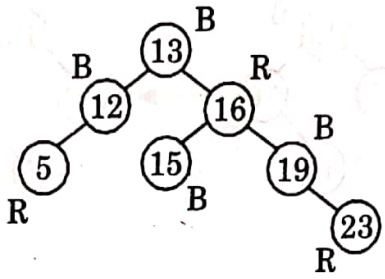
Insert 19 :



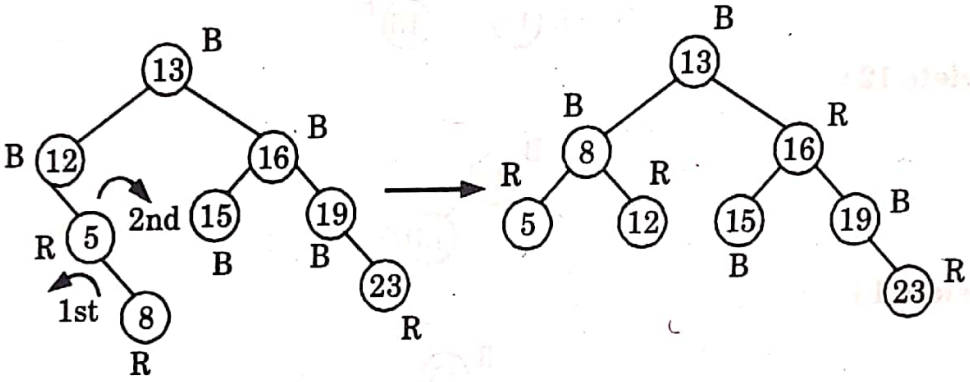
Insert 23 :



Insert 5 :

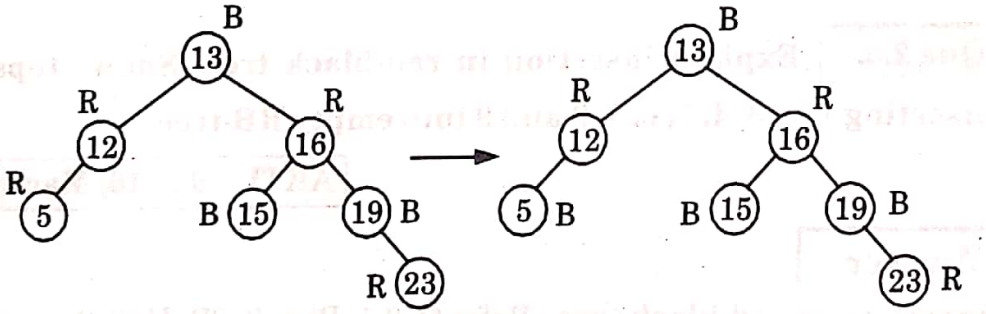


Insert 8 :

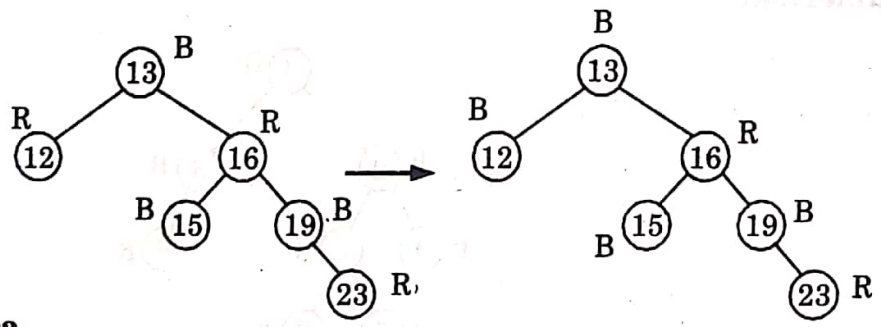


Deletion :

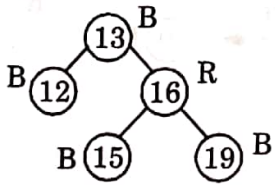
Delete 8 :



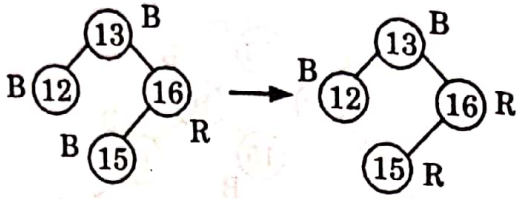
Delete 5 :



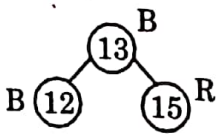
Delete 23 :



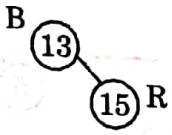
Delete 19 :



Delete 16 :



Delete 12 :



Delete 13 :

Delete 15 :

No tree

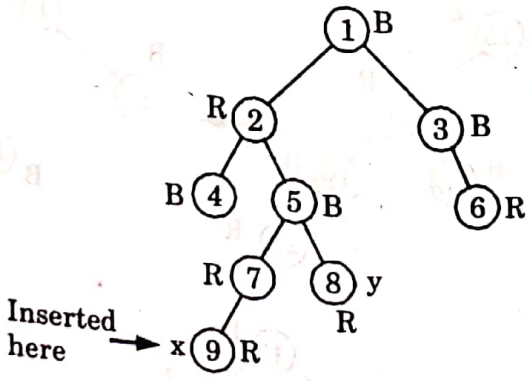
Que 2.5. **Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.**

AKTU 2015-16, Marks 10

Answer

Insertion in red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :



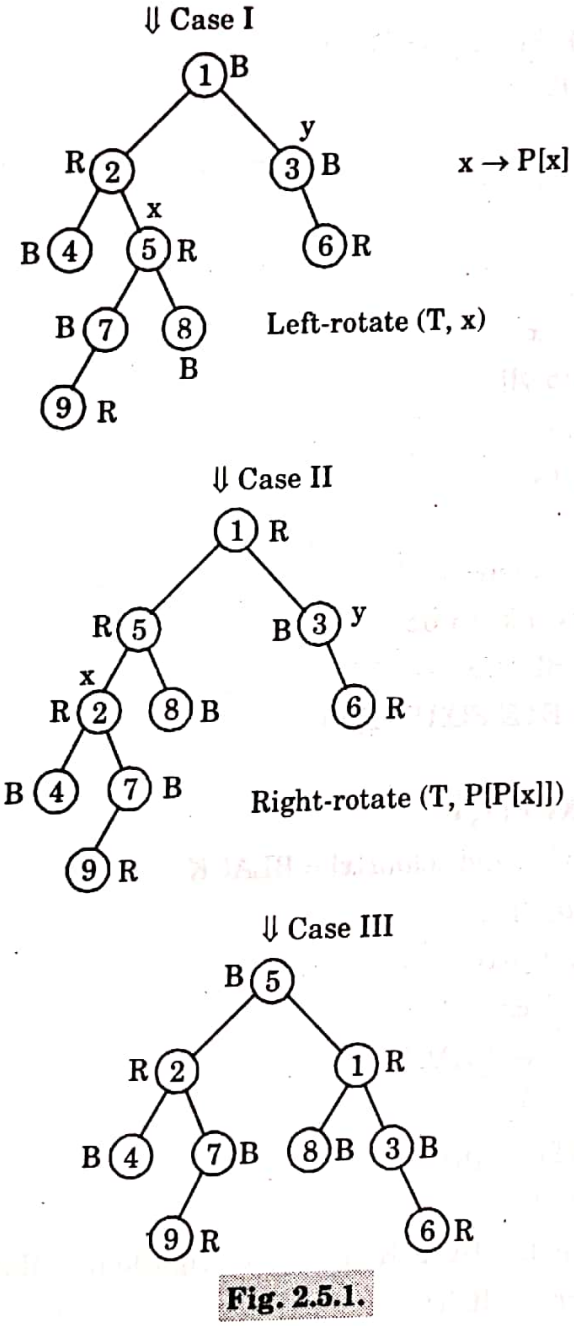


Fig. 2.5.1.

Que 2.6. How to remove a node from RB-tree ? Discuss all cases and write down the algorithm.

Answer

In RB-DELETE procedure, after splitting out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colours and performs rotations to restore the red-black properties.

RB-DELETE(T, z)

1. if left[z] = nil[T] or right[z] = nil[T]

2. then $y \leftarrow z$
3. else $y \leftarrow$ TREE-SUCCESSOR(z)
4. if left[y] $\neq$ nil[T]
5. then $x \leftarrow$ left[y]
6. else $x \leftarrow$ right[y]
7. $p[x] \leftarrow p[y]$
8. if $p[y]$ = nil[T]
9. then root[T] $\leftarrow x$
10. else if y = left[$p[y]$]
11. then left[$p[y]$] $\leftarrow x$
12. else right[$p[y]$] $\leftarrow x$
13. if $y \neq z$
14. then key[z] $\leftarrow$ key[y]
15. copy y's sibling data into z
16. if colour[y] = BLACK
17. then RB-DELETE-FIXUP(T, x)
18. return y

RB-DELETE-FIXUP(T, x)

1. while $x \neq$ root[T] and colour[x] = BLACK
2. do if x = left[$p[x]$]
3. then $w \leftarrow$ right[$p[x]$]
4. if colour[w] = RED
5. then colour[w] $\leftarrow$ BLACK ⇒ case 1
6. colour[$p[x]$] $\leftarrow$ RED ⇒ case 1
7. LEFT-ROTATE($T, p[x]$) ⇒ case 1
8. $w \leftarrow$ right[$p[x]$] ⇒ case 1
9. if colour[left[w]] = BLACK and colour[right[w]] = BLACK
10. then colour[w] $\leftarrow$ RED ⇒ case 2
11. $x \leftarrow p[x]$ ⇒ case 2
12. else if colour[right[w]] = BLACK
13. then colour[left[w]] $\leftarrow$ BLACK ⇒ case 3
14. colour[w] $\leftarrow$ RED ⇒ case 3
15. RIGHT-ROTATE(T, w) ⇒ case 3
16. $w \leftarrow$ right[$p[x]$] ⇒ case 3
17. colour[w] $\leftarrow$ colour[$p[x]$] ⇒ case 4
18. colour[$p[x]$] $\leftarrow$ BLACK ⇒ case 4

19. colour[right[w]] ← BLACK ⇒ case 4
20. LEFT-ROTATE(T, $p[x]$) ⇒ case 4
21. x ← root[T] ⇒ case 4
22. else (same as then clause with "right" and "left" exchanged).
23. colour[x] ← BLACK

Cases of RB-tree for deletion :

Case 1 : x's sibling w is red :

1. It occurs when node w the sibling of node x, is red.
2. Since w must have black children, we can switch the colours of w and $p[x]$ and then perform a left-rotation on $p[x]$ without violating any of the red-black properties.
3. The new sibling of x, which is one of w's children prior to the rotation, is now black, thus we have converted case 1 into case 2, 3 or 4.
4. Case 2, 3 and 4 occur when node w is black. They are distinguished by colours of w's children.

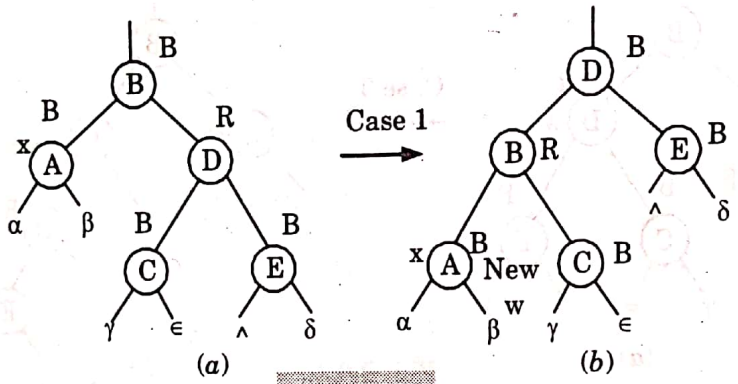


Fig. 2.6.1.

Case 2 : x's sibling w is black, and both of w's children are black :

1. Both of w's children are black. Since w is also black, we take one black of both x and w, leaving x with only one black and leaving w red.
2. For removing one black from x and w, we add an extra black to $p[x]$, which was originally either red or black.
3. We do so by repeating the while loop with $p[x]$ as the new node x.
4. If we enter in case 2 through case 1, the new node x is red and black, the original $p[x]$ was red.
5. The value c of the colour attribute of the new node x is red, and the loop terminates when it tests the loop condition. The new node x is then coloured black.

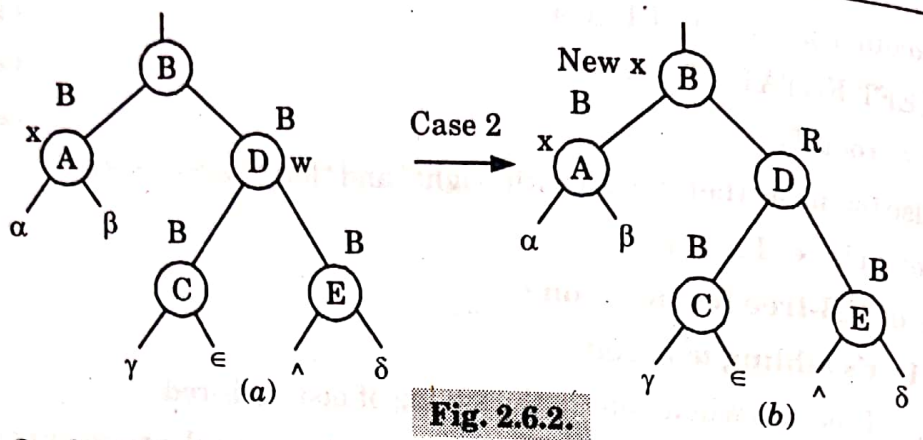


Fig. 2.6.2.

Case 3 : x's sibling w is black, w's left child is red, and w's right child is black :

1. Case 3 occurs when w is black, its left child is red and its right child is black.
2. We can switch the colours of w and its left child left[w] and then perform a right rotation on w without violating any of the red-black properties, the new sibling w of x is a black node with a red right child and thus we have transformed case 3 into case 4.

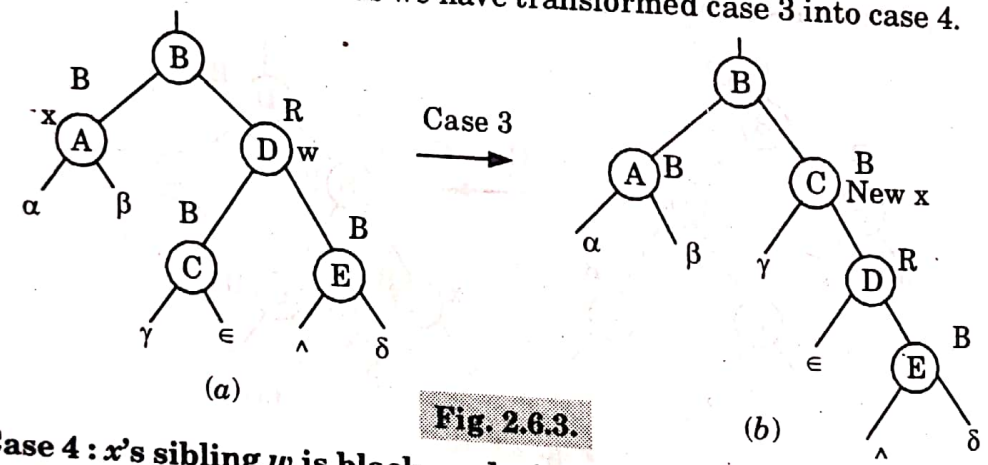


Fig. 2.6.3.

Case 4 : x's sibling w is black, and w's right child is red :

1. When node x's sibling w is black and w's right child is red.
2. By making some colour changes and performing a left rotation on $p[x]$, we can remove the extra black on x, making it singly black, without violating any of the red-black properties.

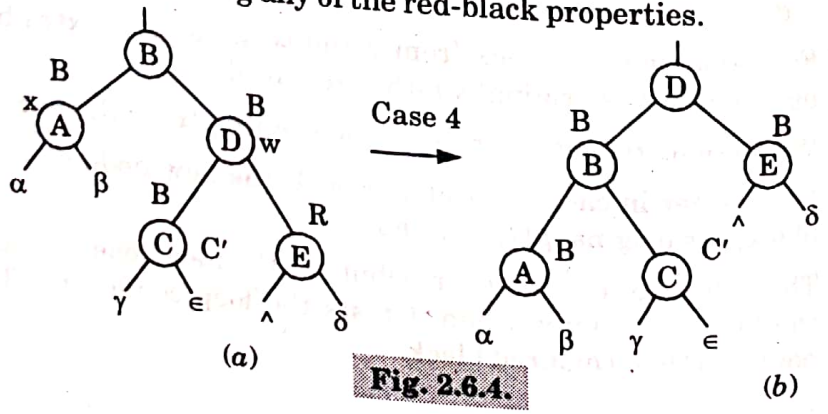


Fig. 2.6.4.

Que 2.7. **Describe the properties of red-black tree. Show the red-black tree with n internal nodes has height at most $2 \log (n + 1)$.**

AKTU 2014-15, Marks 05

OR

Prove the height h of a red-black tree with n internal nodes is not greater than $2 \log (n + 1)$.

Answer

Properties of red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

1. By property 5 of RB-tree, every root-to-leaf path in the tree has the same number of black nodes, let this number be B.
2. So there are no leaves in this tree at depth less than B, which means the tree has at least as many internal nodes as a complete binary tree of height B.
3. Therefore, $n \leq 2^B - 1$. This implies $B \leq \log (n + 1)$.
4. By property 4 of RB-tree, at most every other node on a root-to-leaf path is red. Therefore, $h \leq 2B$.

   Putting these together, we have

   $$h \leq 2 \log (n + 1).$$

Que 2.8. **Define a B-tree of order m. Explain the searching operation in a B-tree.**

Answer

A B-tree of order m is an m-ary search tree with the following properties :

1. The root is either leaf or has atleast two children.
2. Each node, except for the root and the leaves, has between $m/2$ and m children.
3. Each path from the root to a leaf has the same length.
4. The root, each internal node and each leaf is typically a disk block.
5. Each internal node has upto $(m - 1)$ key values and upto m children.

Searching operation in a B-tree :

We adopt following convention to all operation :

a. Root is always in main memory.

b. Nodes passed to operations must have been read.

c. All operations go from root down in one pass, $O(h)$.

SEARCH(x, k)

1. $i \leftarrow 1$

2. while $i \leq n[x]$ and $k > key_i[x]$
3. do $i \leftarrow i + 1$
4. if $i \leq n[x]$ and $k = key_i[x]$
5. then return(x, i)
6. if leaf$[x]$
7. then return NIL
8. else DISK-READ$(c_i[x])$
9. return B-TREE-SEARCH $(c_i[x], k)$

The number of disk pages accessed by B-TREE-SEARCH is $\theta(n) = \theta(\log_t n)$, where h is the height of the tree and n is the number of keys in the tree. Since $n[x] < 2t$, time taken by the while loop of lines 2-3 within each nodes is $O(t)$ and the total CPU time is $O(th) = O(t \log_t n)$

B-TREE-INSERT(T, k)

1. $r \leftarrow$ root$[T]$
2. if $n[r] = 2t - 1$
3. then $s \leftarrow$ ALLOCATE-NODE ( )
4. root$[T] \leftarrow S$
5. leaf$[s] \leftarrow$ FALSE
6. $n[s] \leftarrow 0$
7. $c_1[s] \leftarrow r$
8. B-TREE SPLIT CHILD(S, l, r)
9. B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)

B-TREE SPLIT CHILD(x, i, y)

1. $z \leftarrow$ ALLOCATE-NODE ( )
2. leaf$[z] \leftarrow$ leaf$[y]$
3. $n[z] \leftarrow t - 1$
4. for $j \leftarrow 1$ to $t - 1$
5. do $key_j[z] \leftarrow key_{j+t}[y]$
6. if not leaf$[y]$
7. then for $j \leftarrow 1$ to t
8. do $c_j[z] \leftarrow c_{j+t}[y]$
9. $n[y] \leftarrow t - 1$

10. for $j \leftarrow n[x] + 1$ down to $i + 1$
11. do $c_{j+1}[x] \leftarrow c_j[x]$
12. $c_{i+1}[x] \leftarrow z$
13. for $j \leftarrow n[x]$ down to i
14. do $key_{j+1}[x] \leftarrow key_j[x]$
15. $key_i[x] \leftarrow key_t[y]$
16. $n[x] \leftarrow n[x] + 1$
17. DISK-WRITE[y]
18. DISK-WRITE[z]
19. DISK-WRITE[x]

The CPU time used by B-TREE SPLIT CHILD is $\theta(t)$. The procedure performs $\theta(1)$ disk operations.

B-TREE-INSERT-NONFULL(x, k)

1. $i \leftarrow n[x]$
2. if leaf[x]
3. then while $i \geq 1$ and $k < key_i[x]$
4. do $key_{i+1}[x] \leftarrow key_i[x]$
5. $i \leftarrow i - 1$
6. $key_{i+1}[x] \leftarrow k$
7. $n[x] \leftarrow n[x] + 1$
8. DISK-WRITE(x)
9. else while $i \geq 1$ and $k < key_i[x]$
10. do $i \leftarrow i - 1$
11. $i \leftarrow i + 1$
12. DISK-READ($c_i[x]$)
13. if $n[c_i[x]] = 2t - 1$
14. then B-TREE-SPLIT-CHILD($x, i, c_i[x]$)
15. if $k > key_i[x]$
16. then $i \leftarrow i + 1$
17. B-TREE INSERT NONFULL($c_i[x], k$)

The total CPU time use is $O(th) = O(t \log_t n)$

Que 2.9. What are the characteristics of B-tree ? Write down the steps for insertion operation in B-tree.

Answer

Characteristic of B-tree :

1. Each node of the tree, except the root node and leaves has at least $m/2$ subtrees and no more than m subtrees.
2. Root of tree has at least two subtree unless it is a leaf node.
3. All leaves of the tree are at same level.

Insertion operation in B-tree :

In a B-tree, the new element must be added only at leaf node. The insertion operation is performed as follows :

Step 1 : Check whether tree is empty.

Step 2 : If tree is empty, then create a new node with new key value and insert into the tree as a root node.

Step 3 : If tree is not empty, then find a leaf node to which the new key value can be added using binary search tree logic.

Step 4 : If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

Step 5 : If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6 : If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Que 2.10. Describe a method to delete an item from B-tree.

Answer

There are three possible cases for deletion in B-tree as follows :

Let k be the key to be deleted, x be the node containing the key.

Case 1 : If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted. Key k is in node x and x is a leaf, simply delete k from x.

Case 2 : If key k is in node x and x is an internal node, there are three cases to consider :

a. If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x.

b. Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before.

c. Otherwise, if both y and z have only $t - 1$ (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x, y now contains $2t - 1$ keys, and subsequently k is deleted.

Case 3 : If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only $t - 1$ keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurse to the appropriate child of x.

a. If the root has only $t - 1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

b. If the root and all of its siblings have $t - 1$ keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Que 2.11. How B-tree differs with other tree structures ? Insert the following information *F, S, Q, K, C, L, V, W, M, R, N, P, A, I, Z, E,* into an empty B-tree with degree $t = 2$. **AKTU 2014-15, Marks 10**

Answer

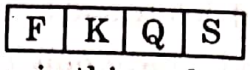
1. In B-tree, the maximum number of child nodes a non-terminal node can have is m where m is the order of the B-tree. On the other hand, other tree can have at most two subtrees or child nodes.
2. B-tree is used when data is stored in disk whereas other tree is used when data is stored in fast memory like RAM.
3. B-tree is employed in code indexing data structure in DBMS, while, other tree is employed in code optimization, Huffman coding, etc.
4. The maximum height of a B-tree is log mn (m is the order of tree and n is the number of nodes) and maximum height of other tree is $\log_2 n$ (base is 2 because it is for binary).
5. A binary tree is allowed to have zero nodes whereas any other tree must have atleast one node. Thus binary tree is really a different kind of object than any other tree.

Numerical :

$$t = 2$$
$$2t - 1 = 2 \times 2 - 1 = 3$$
$$t - 1 = 2 - 1 = 1$$

So, maximum of 3 keys and minimum of 1 key can be inserted in a node.
Now, apply insertion process as :

Insert *F, S, Q, K* :



As, there are more than 3 keys in this node.

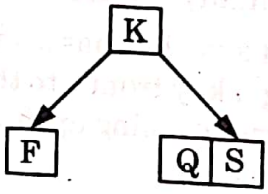
∴ Find median,

$$n[x] = 4 \text{ (even)}$$

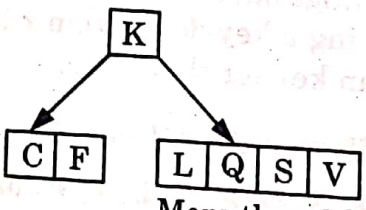
$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$

Now, median = 2,

So, we split the node by 2nd key.



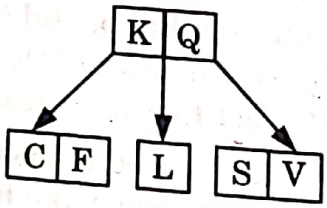
Insert *C, L, V* :



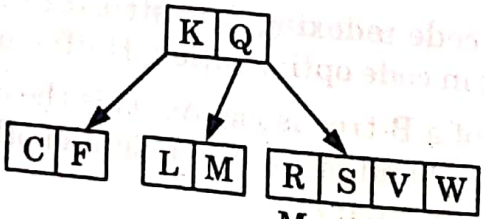
More than 3 keys

split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)



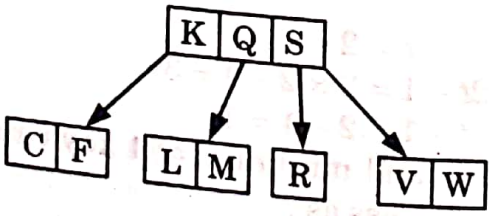
Insert *W, M, R* :



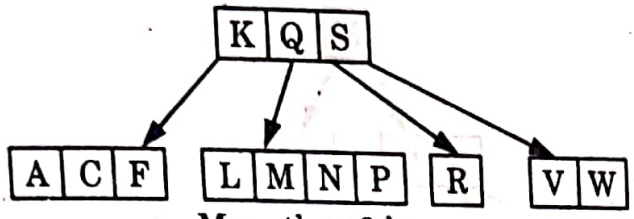
More than 3 keys

split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

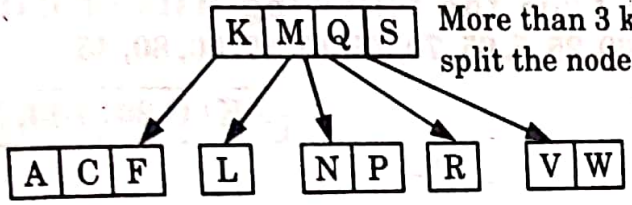


Insert *N, P, A* :



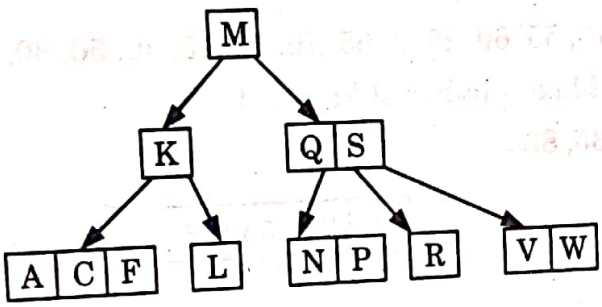
More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

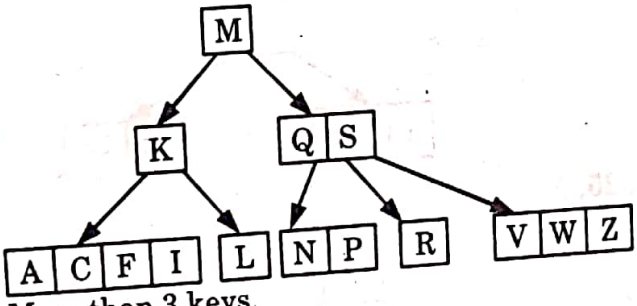


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)

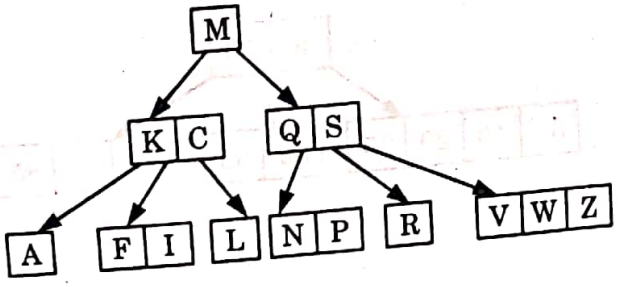


Insert *I, Z* :

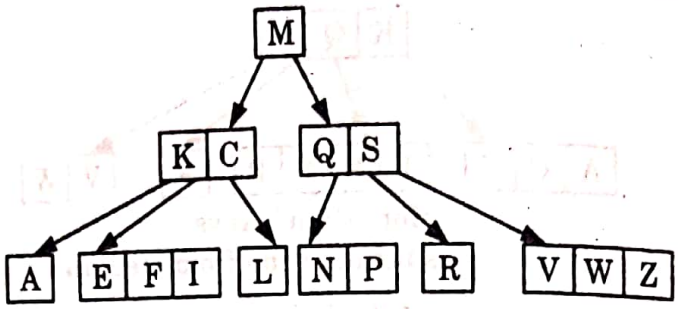


More than 3 keys
split the node from median

$$\text{Median} = \frac{n[x]}{2} = \frac{4}{2} = 2$$ (*i.e.*, 2nd key move up)



Insert *E* :



Que 2.12. Write the characteristics of a B-tree of order *m*. Create B-tree of order 5 from the following lists of data items : 20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45.

AKTU 2013-14, Marks 10

Answer

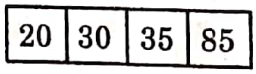
Characteristics of B-tree : Refer Q. 2.9, Page 2–18B, Unit-2.

Numerical :

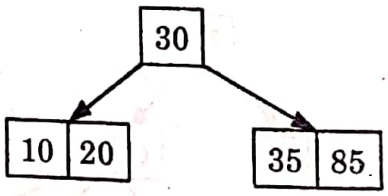
20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45

∵ order = 5, Max. number of keys = 4

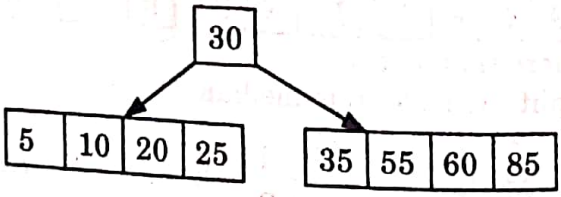
Insert 20, 30, 35, 85 :



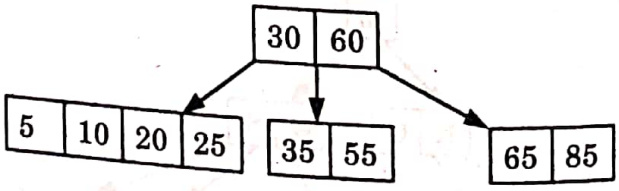
Insert 10 :



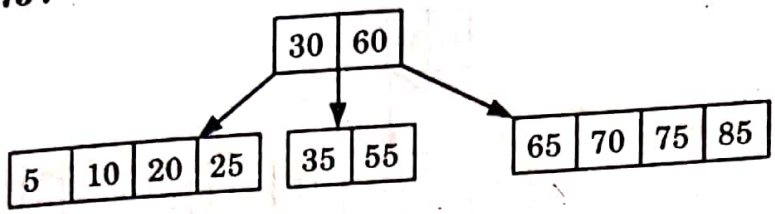
Insert 55, 60, 25, 5 :



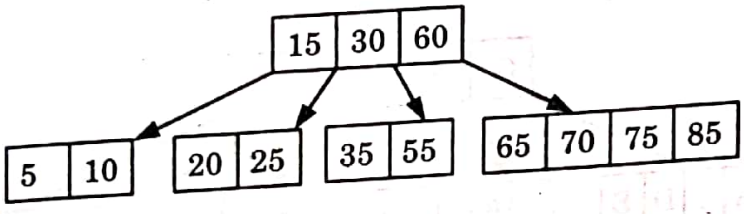
Insert 65 :



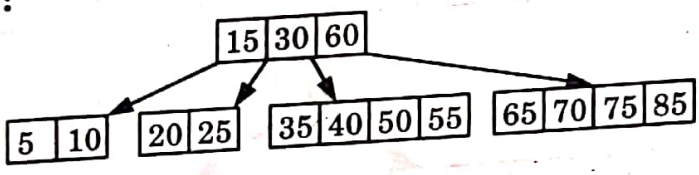
Insert 70, 75 :



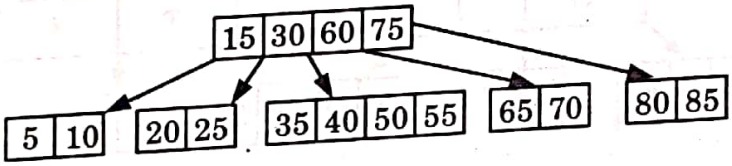
Insert 15 :



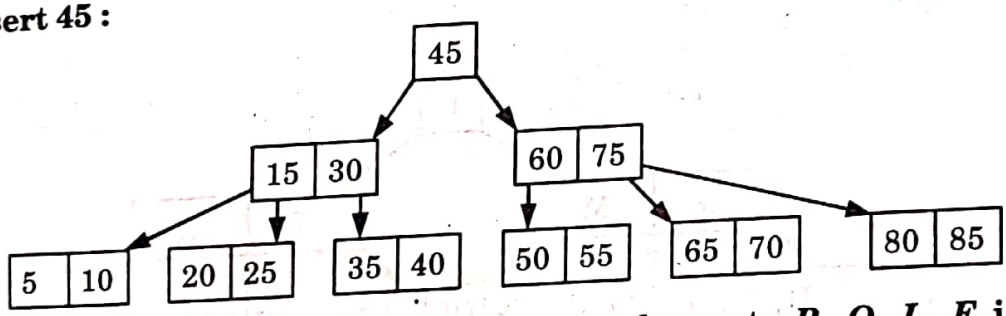
Insert 40, 50 :



Insert 80 :



Insert 45 :



Que 2.13. **Explain B-tree and insert elements *B, Q, L, F* into B-tree Fig. 2.13.1 then apply deletion of elements *F, M, G, D, B* on resulting B-tree.**

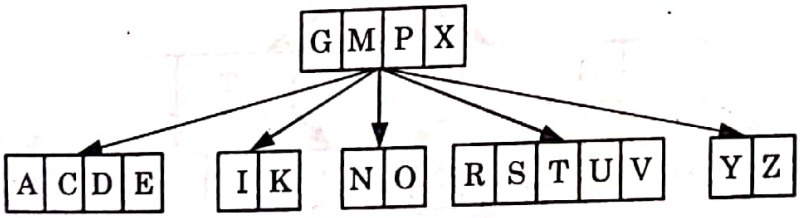


Fig. 2.13.1.

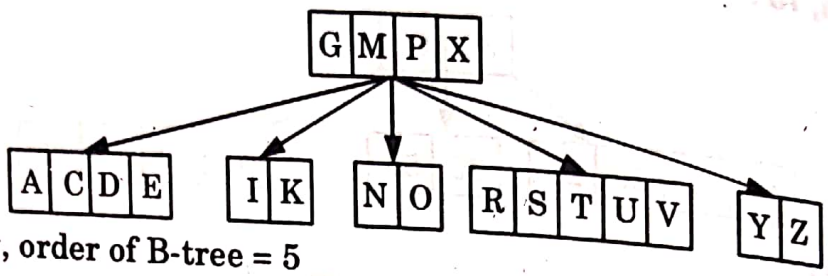
AKTU 2015-16, Marks 10

Answer

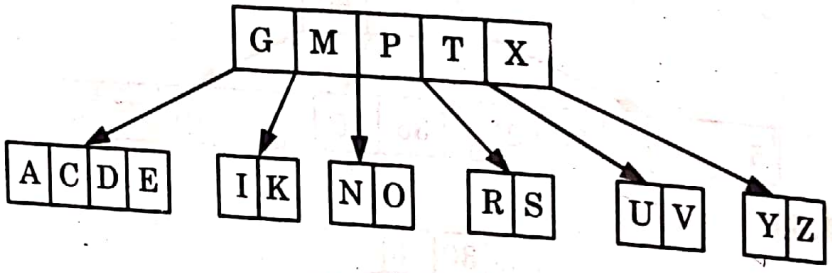
B-tree : Refer Q. 2.8, Page 2–15B, Unit-2.

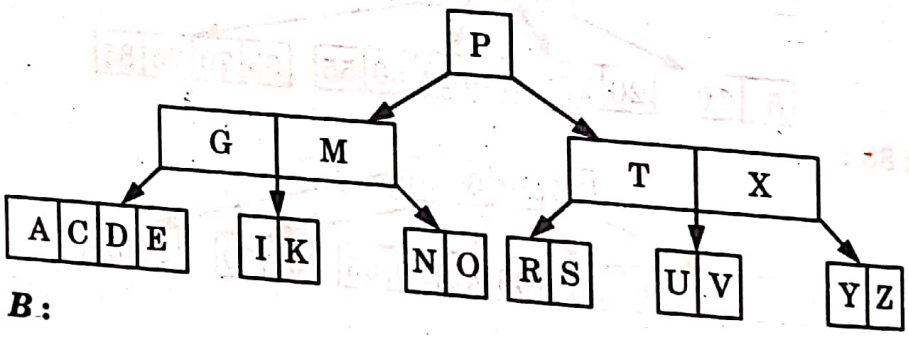
Numerical :

Insertion :

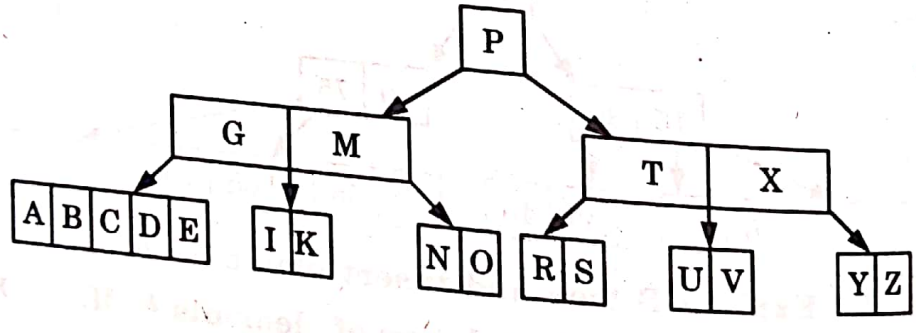


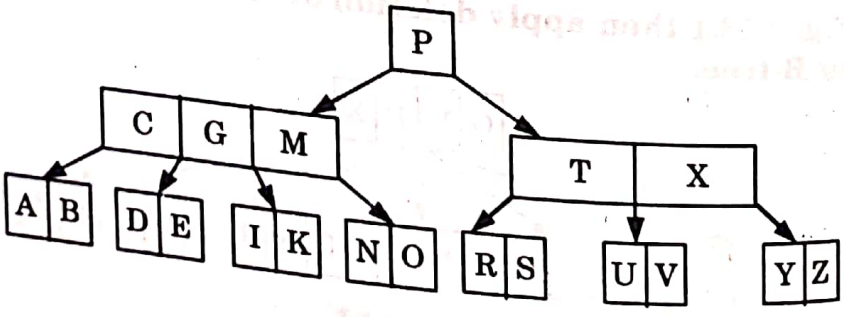
Assuming, order of B-tree = 5



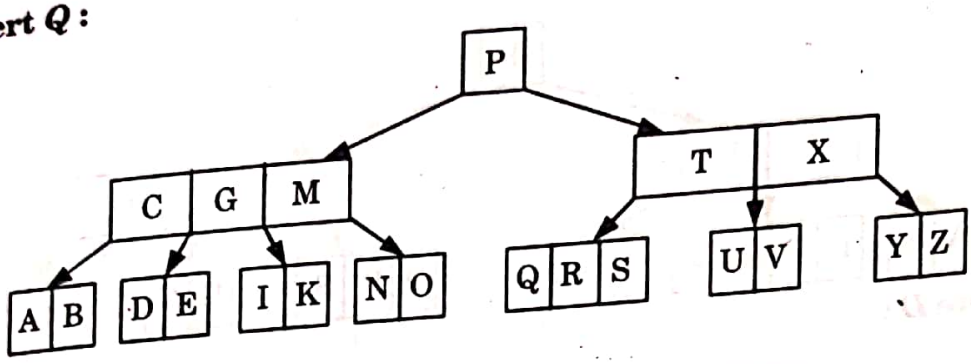


Insert *B* :

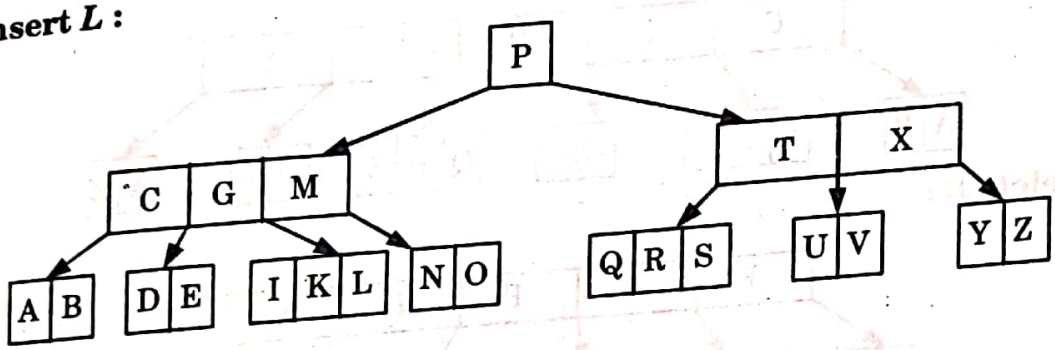




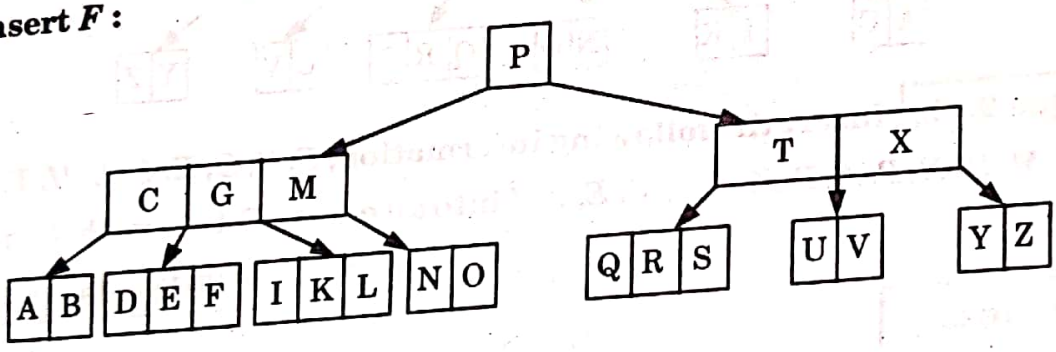
Insert *Q* :



Insert *L* :

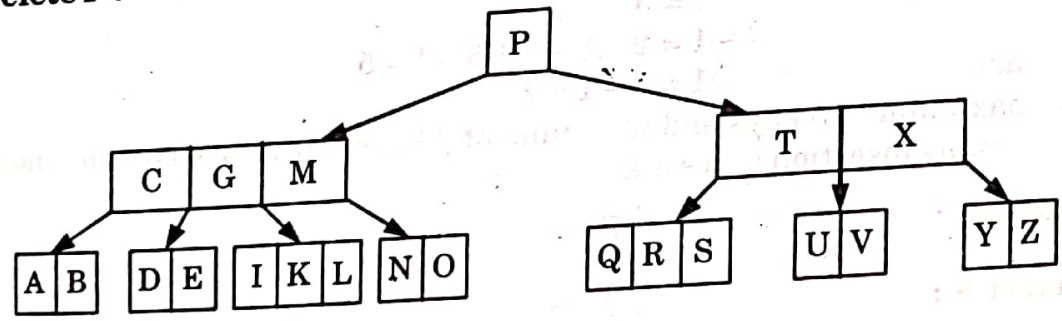


Insert *F* :

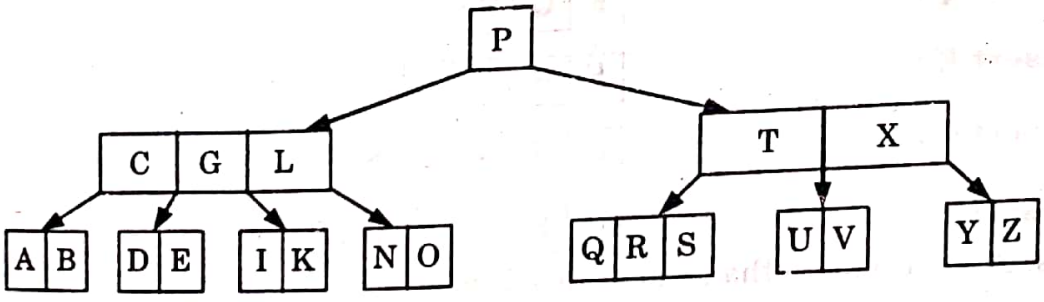


Deletion :

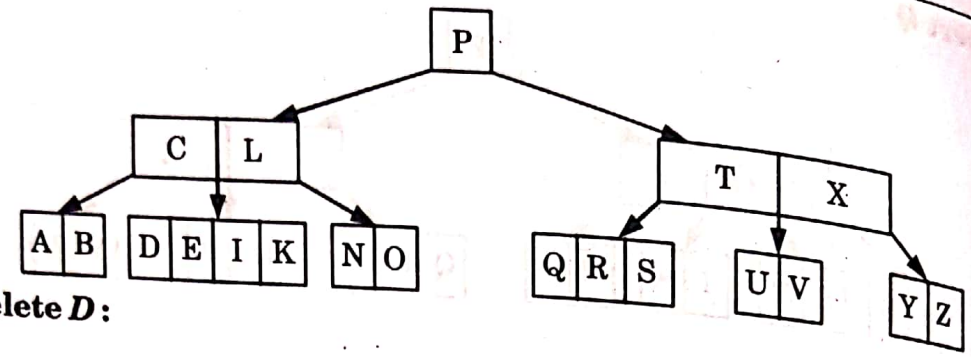
Delete *F* :



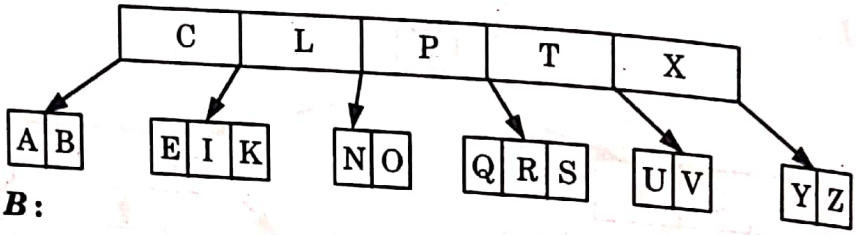
Delete *M* :



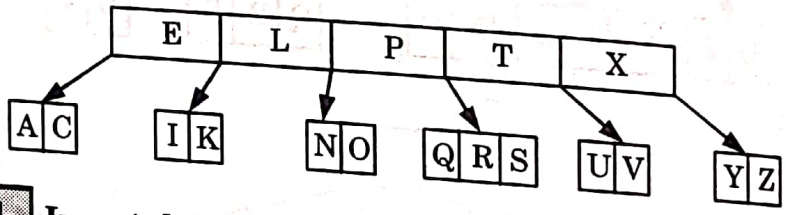
Delete *G* :



Delete *D* :



Delete *B* :



Que 2.14. **Insert the following information, *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I* into an empty B-tree with degree *t* = 3.**

AKTU 2017-18, Marks 10

Answer

Assume that

$$t = 3$$

$$2t - 1 = 2 \times 3 - 1 = 6 - 1 = 5$$

and

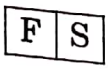
$$t - 1 = 3 - 1 = 2$$

So, maximum of 5 keys and minimum of 2 keys can be inserted in a node.

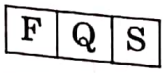
Now, apply insertion process as :

Insert *F* :

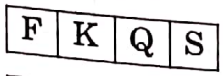
Insert *S* :



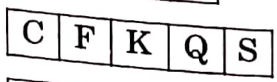
Insert *Q* :



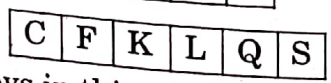
Insert *K* :



Insert *C* :



Insert *L* :



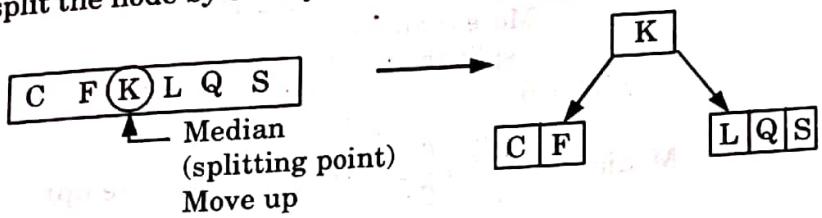
As, there are more than 5 keys in this node.

∴ Find median,

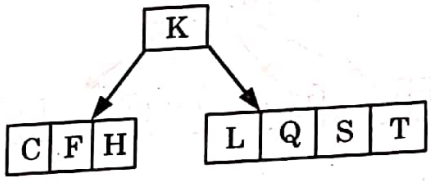
$n[x] = 6$ (even)

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

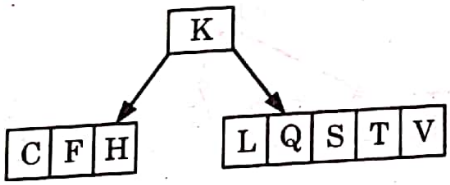
Now, median = 3,
So, we split the node by 3rd key.



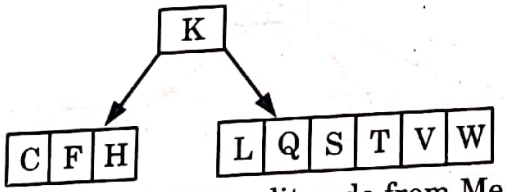
Insert *H, T* :



Insert *V* :



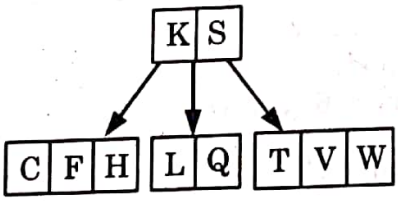
Insert *W* :



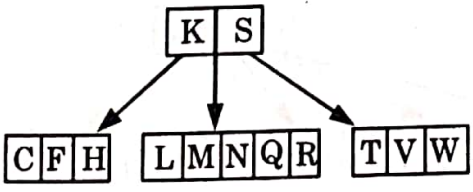
More than 5 keys split node from Median.

$n[x] = 6$ [even]

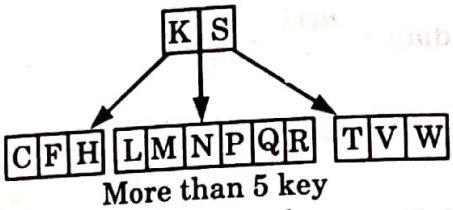
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$ (*i.e.*, 3rd key move up)



Insert *M, R, N* :



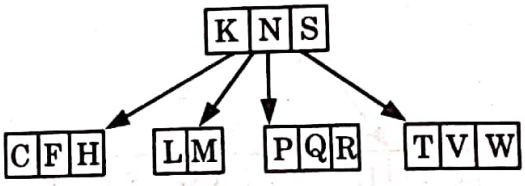
Insert *P* :



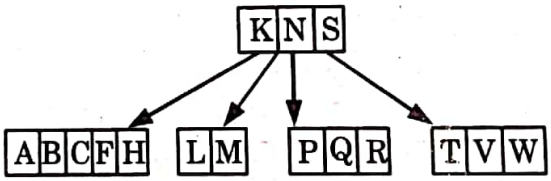
More than 5 key
split the node

$n[x] = 6$

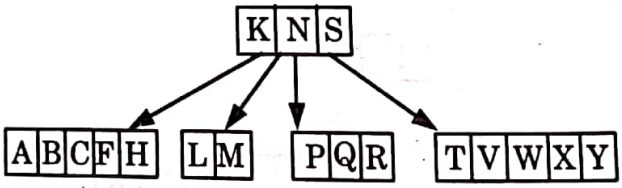
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \text{ (i.e., 3rd key move up)}$$



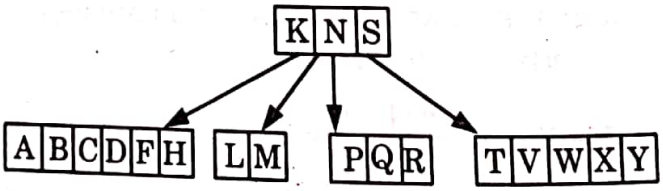
Insert *A*, *B* :



Insert *X*, *Y* :



Insert *D* :

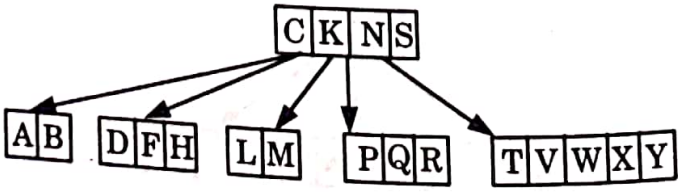


More than 5 key
split the node

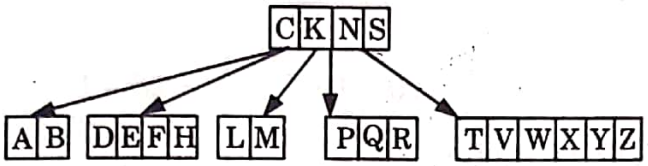
$n[x] = 6$ (even)

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

(*i.e.*, 3rd key move up)



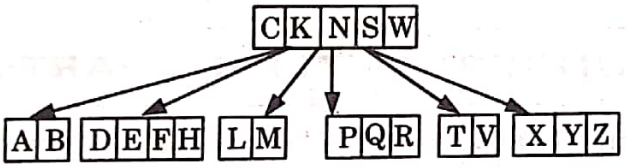
Insert *Z, E* :



More than 5 key split the node

$n[x] = 6$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$ (*i.e.*, 3rd key move up)



Insert *G, I* :

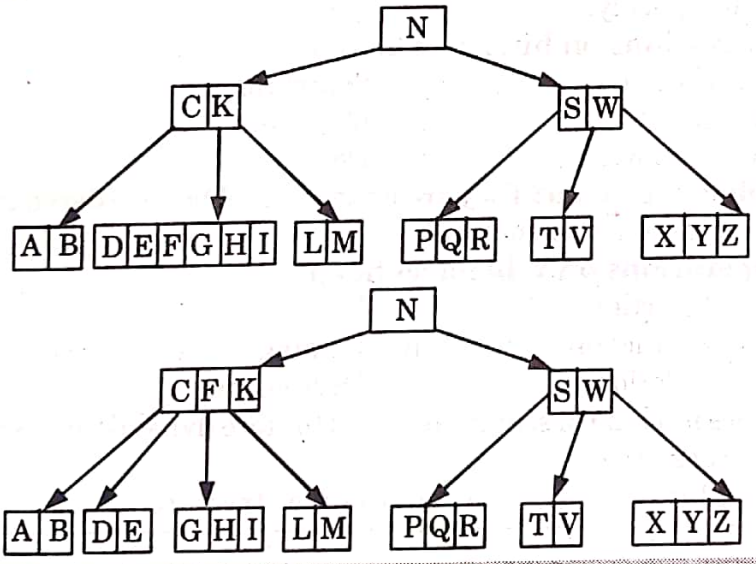


Fig. 2.14.1. Inserted all given information with degree t = 3.

Que 2.15. **If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, Prove that : $h \leq \log_t \frac{(n+1)}{2}$.**

Answer

Proof :

1. The root contains at least one key.
2. All other nodes contain at least $t - 1$ keys.
3. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h.

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

4. So $t^h \leq (n+1)/2$ as required.
   Taking log both sides we get,
   $$h \leq \log_t (n+1)/2$$

PART-2

Binomial Heaps, Fibonacci Heaps, Tries, Skip List.

CONCEPT OUTLINE : PART-2

- **Binomial heap :** A binomial heap is a data structure similar to binary heap but also supporting the operation of merging two heaps quickly.
- **Operations on binomial heap :**
  i. Creation ii. Searching
  iii. Union iv. Insertion
  v. Removal vi. Decreasing
- **Fibonacci heap :** They are linked lists of heap-ordered trees. It is also a collection of trees.
- **Operations on Fibonacci heap :**
  i. Insertion ii. Union
  iii. Extraction iv. Linking
  v. Deletion iv. Decreasing
- Tries is a kind of search tree used to store dynamic or associative sets of strings.
- Skip list is a layered linked list data structure.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.16. Explain binomial heap and properties of binomial tree.

Answer

Binomial heap :

1. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.
2. A binomial heap is implemented as a collection of binomial tree.

Properties of binomial tree :

1. The total number of node at order k are 2^k.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ *i.e.*, ${}^kC_i$ nodes at depth i for $i = 0, 1, \ldots, k$ (this is why the tree is called a "binomial" tree).
4. Root has degree k (children) and its children are $B_{k-1}, B_{k-2}, \ldots, B_0$ from left to right.

Que 2.17. What is a binomial heap ? Describe the union of binomial heap. **AKTU 2013-14, Marks 10**

OR

Define the binomial heap in detail. Write an algorithm for performing the union operation of two binomial heaps and also explain with suitable example. **AKTU 2014-15, Marks 10**

Answer

Binomial heap : Refer Q. 2.16, Page 2–30B, Unit-2.

Union of binomial heap :

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.
2. The following procedure links the B_{k-1} tree rooted at node to the B_{k-1} tree rooted at node z, that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

   BINOMIAL-LINK (y, z)

   i. $p[y] \leftarrow z$

   ii. $sibling\ [y] \leftarrow child[z]$

   iii. $child[z] \leftarrow y$

   iv. $degree[z] \leftarrow degree[z] + 1$
3. The BINOMIAL-HEAP-UNION procedure has two phases :
   a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order.
   b. The second phase links root of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the like operations quickly.

BINOMIAL-HEAP-UNION(H_1, H_2)

1. $H \leftarrow$ MAKE-BINOMIAL-HEAP ( )
2. $head[H] \leftarrow$ BINOMIAL-HEAP-MERGE(H_1, H_2)

3. Free the objects H_1 and H_2 but not the lists they point to
4. if head[H] = NIL
5. then return H
6. prev-x ← NIL
7. x ← head[H]
8. next-x ← sibling[x]
9. while next-x ≠ NIL
10. do if (degree[x] ≠ degree[next-x]) *or*
    (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
11. then prev-x ← x
12. x ← next-x ⇒ case 1 and 2
13. else if key[x] ≤ key[next-x] ⇒ case 1 and 2
14. then sibling[x] ← sibling[next-x]
15. BINOMIAL-LINK(next-x, x) ⇒ case 3
16. else if prev-x = NIL ⇒ case 3
17. then head[H] ← next-x ⇒ case 4
18. else sibling[prev-x] ← next-x ⇒ case 4
19. BINOMIAL-LINK(x, next-x) ⇒ case 4
20. x ← next-x ⇒ case 4
21. next-x ← sibling[x] ⇒ case 4
22. return H

BINOMIAL-HEAP-MERGE(H_1, H_2)

1. a ← head[H_1]
2. b ← head[H_2]
3. head[H_1] ← min-degree (a, b)
4. if head[H_1] = NIL
5. return
6. if head[H_1] = b
7. then b ← a
8. a ← head[H_1]
9. while b ≠ NIL
10. do if sibling[a] = NIL
11. then sibling[a] ← b
12. return
13. else if degree [sibling[a]] < degree[b]
14. then a ← sibling[a]

15. else $c \leftarrow$ sibling[b]
16. sibling[b] $\leftarrow$ sibling[a]
17. sibling[a] $\leftarrow b$
18. $a \leftarrow$ sibling[a]
19. $b \leftarrow c$

There are four cases that occur while performing union on binomial heaps.

Case 1 : When degree[x] $\neq$ degree[next $-x$] = degree [sibling[next $-x$]], then pointers moves one position further down the root list.

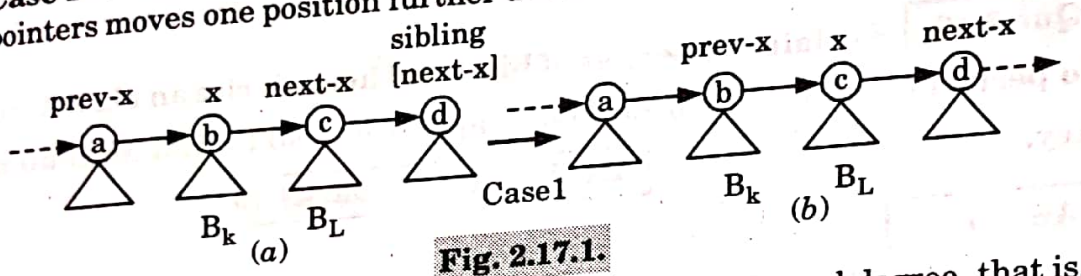


Fig. 2.17.1.

Case 2 : It occurs when x is the first of three roots of equal degree, that is, degree[x] = degree[next $-x$] = degree[sibling[next $-x$]], then again pointer move one position further down the list, and next iteration executes either case 3 or case 4.

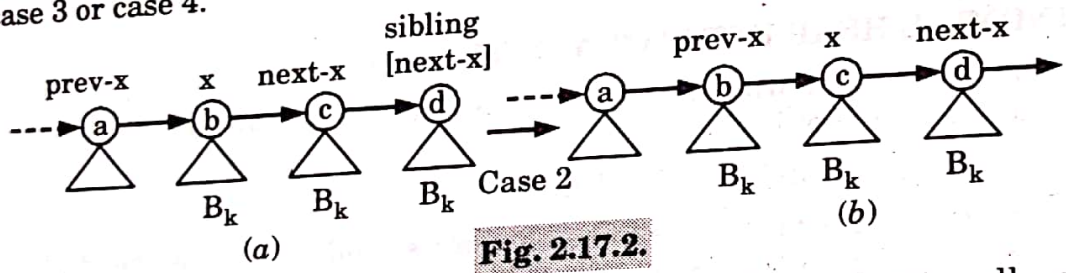


Fig. 2.17.2.

Case 3 : If degree[x] = degree[next $-x$] $\neq$ degree [sibling[next $-x$]] and key[x] $\leq$ key[next $-x$], we remove next $-x$ from the root list and link it to x, creating B_{k+1} tree.

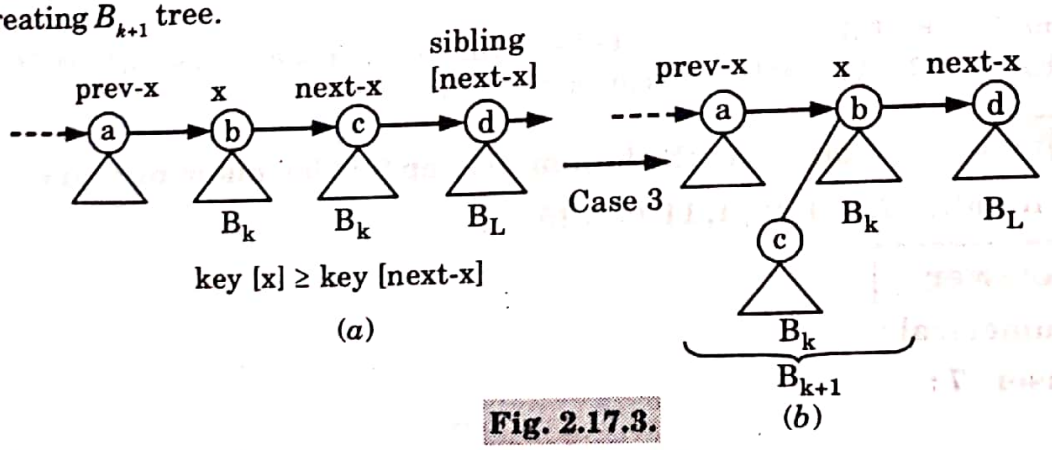


Fig. 2.17.3.

Case 4 : degree[x] = degree[next $-x$] $\neq$ degree[sibling[next $-x$] and key[next $-x$] $\leq$ key x, we remove x from the root list and link it to next $-x$, again creating a B_{k+1} tree.

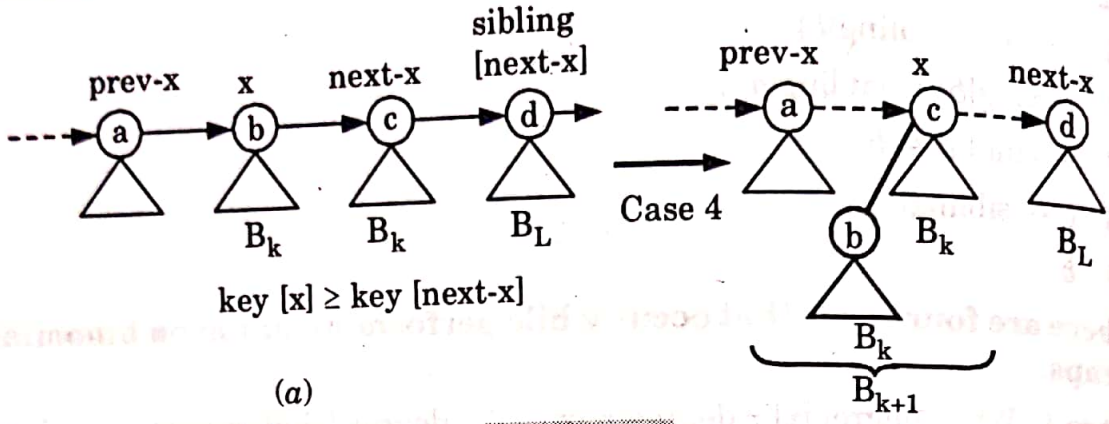


Fig. 2.17.4.

Que 2.18. **Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find Minimum key.** **AKTU 2017-18, Marks 10**

Answer

Properties of binomial heap : Refer Q. 2.16, Page 2–30B, Unit-2.

Algorithm for union of binomial heap : Refer Q. 2.17, Page 2–31B, Unit-2.

Minimum key :

BINOMIAL-HEAP-EXTRACT-MIN (H) :

1. Find the root x with the minimum key in the root list of H, and remove x from the root list of H.
2. $H' \leftarrow$ MAKE-BINOMIAL-HEAP( ).
3. Reverse the order of the linked list of x's children, and set head[H'] to point to the head of the resulting list.
4. $H \leftarrow$ BINOMIAL-HEAP-UNION(H, H').
5. Return x

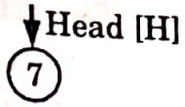
Since each of lines 1-4 takes O(lg n) time of H has n nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in O(lg n) time.

Que 2.19. **Construct the binomial heap for the following sequence of number 7, 2, 4, 17, 1, 11, 6, 8, 15.**

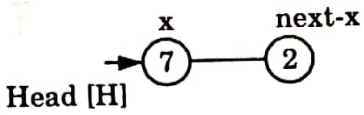
Answer

Numerical :

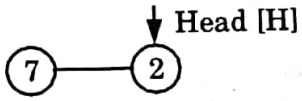
Insert 7 :



Insert 2 :

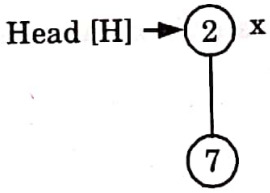


prev-x = NIL
degree [x] = 0. So, degree [x] ≠ degree [next-x] is false.
degree [next-x] = 0 and Sibling [next-x] = NIL
So, case 1 and 2 are false here.
Now key [x] = 7 and key [next-x] = 2
Now prev-x = NIL
then Head [H] ← next-x and
i.e.,



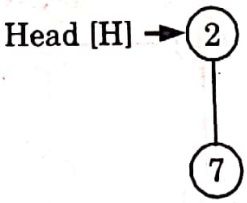
and BINOMIAL-LINK (x, next-x)
i.e.,

Now

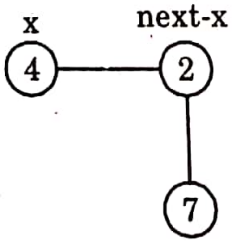


and next-x = NIL

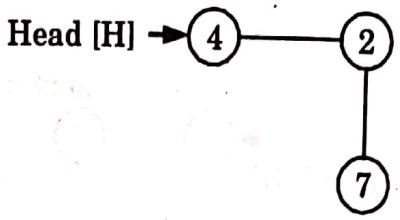
So, after inserting 2, binomial heap is



Insert 4 :

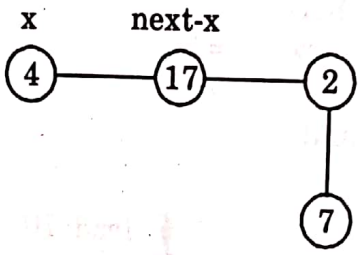


degree [x] ≠ degree [next-x]
So, Now next-x makes x and x makes prev-x.
Now next-x = NIL
So, after inserting 4, final binomial heap is :



Insert 17 :

After Binomial-Heap-Merge, we get



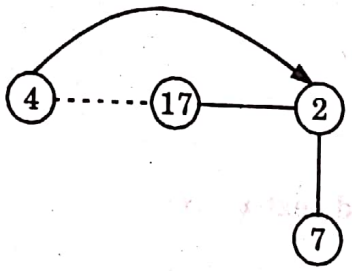
degree [x] = degree [next-x]

degree [Sibling-[next-x]] ≠ degree [x]

key [x] ≤ key [next-x]

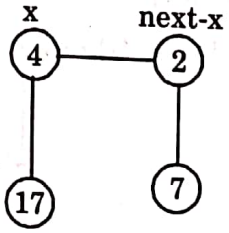
4 ≤ 17 [True]

So,



and call Binomial-Link [next-x, x]

We get



degree [x] = degree [next-x]

Sibling [next-x] = NIL

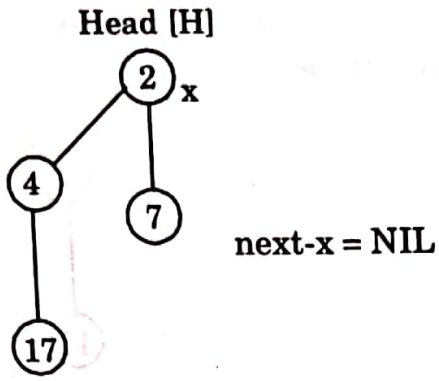
Key [x] ≤ key [next-x] [False]

prev-x = NIL then

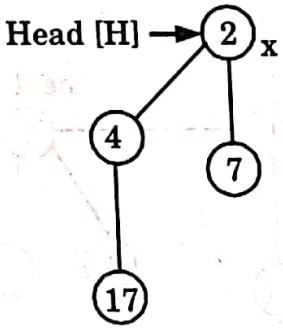
Head [H] ← [next-x]

Binomial-Link [x, next-x]

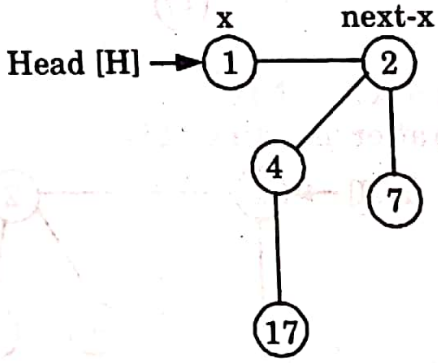
x ← next-x



So, after inserting 17, final binomial heap is :



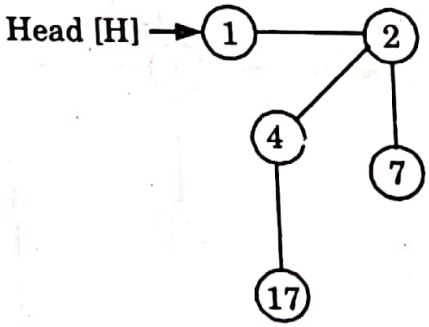
Insert 1 :



degree [x] ≠ degree [next-x]

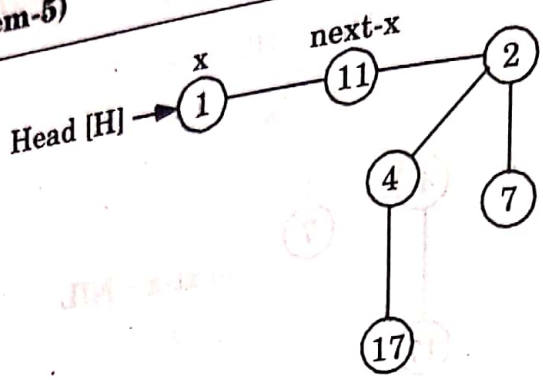
So, next-x makes x and next-x = NIL

and after inserting 1, binomial heap is :

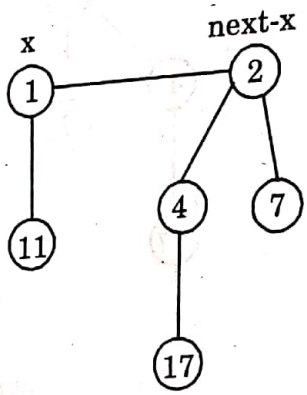


Insert 11 :

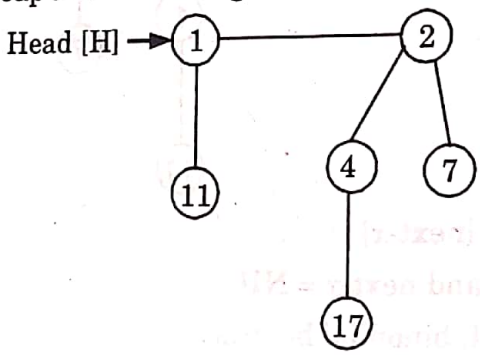
After Binomial-Heap-Merge, we get



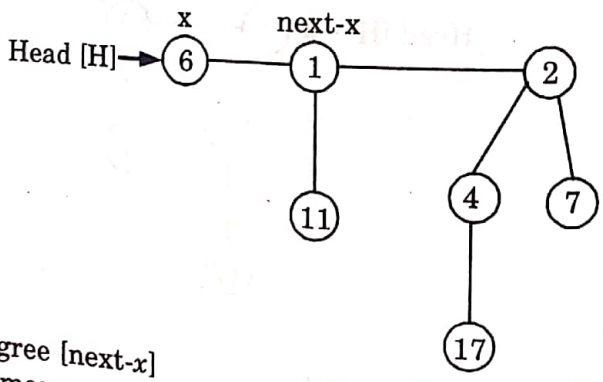
degree [x] = degree [next-x]
degree [Sibling [next-x]] ≠ degree [x]
key [x] ≤ key [next-x] [True]
So,



degree [x] ≠ degree [next-x]
So, next-x makes x and next-x = NIL
and final binomial heap after inserting 11 is

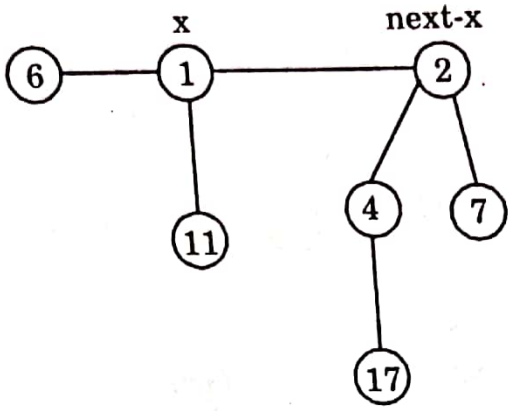


Insert 6 :

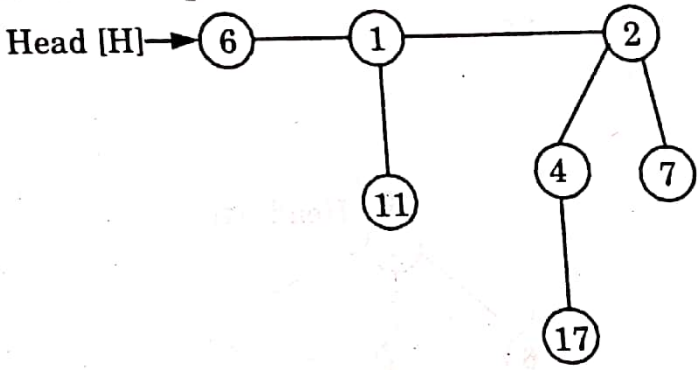


degree [x] ≠ degree [next-x]
So, next-x becomes x
Sibling [next-x] becomes next-x.

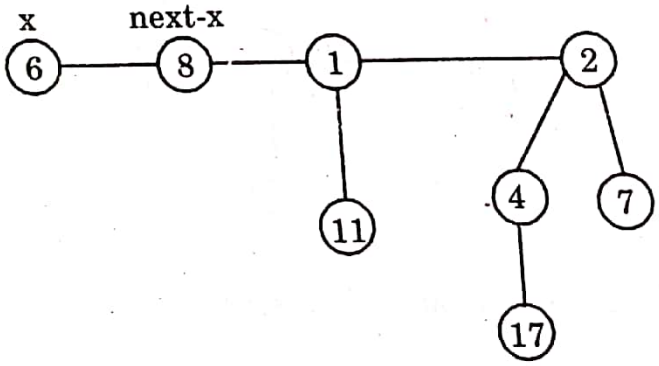
i.e.,



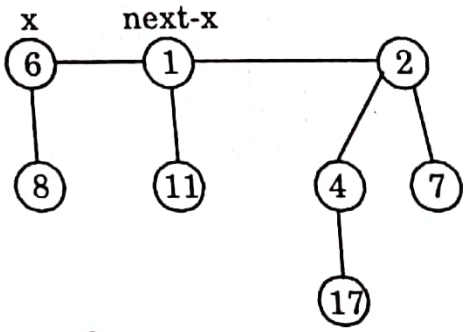
degree [x] ≠ degree [next-x]
So, no change and final heap is :



Insert 8 :

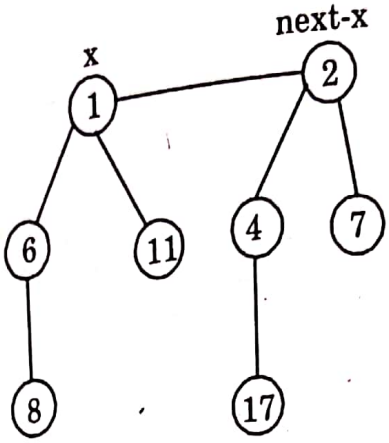


degree [x] = degree [next-x]
degree [Sibling [next-x]] ≠ degree [x]
key [x] ≤ key [next-x] [True]
So,

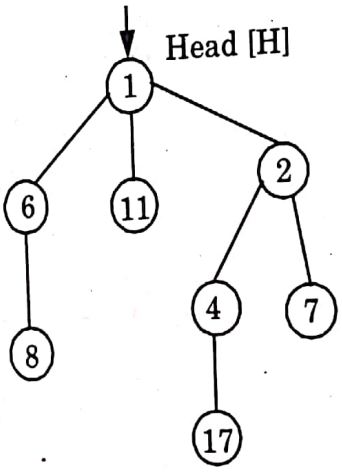


degree [x] = degree [next-x]
degree [Sibling p[next-x]] ≤ degree [x]
key [x] ≤ key [next-x] [False]
prev-x = NIL

So,

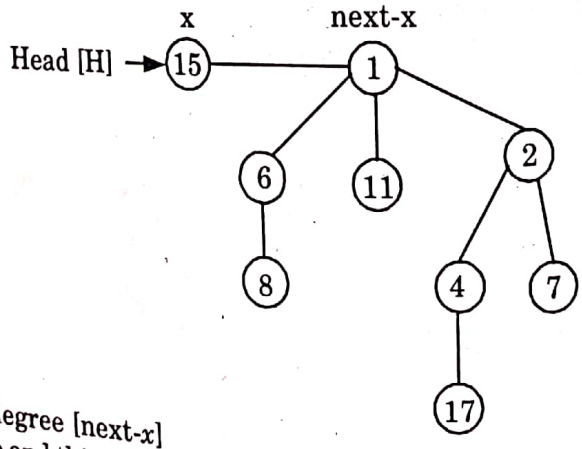


degree [x] = degree [next-x]
Sibling [next-x] = NIL
key [x] ≤ key [next-x] [True]
So, Sibling [x] = NIL.
and



next [x] = NIL
So, this is the final binomial heap after inserting 8.

Insert 15 :



degree [x] ≠ degree [next-x]
So, no change and this is the final binomial heap after inserting 15.

Que 2.20. What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.

Answer

1. A Fibonacci heap is a set of min-heap-ordered trees.
2. Trees are not ordered binomial trees, because
   a. Children of a node are unordered.
   b. Deleting nodes may destroy binomial construction.

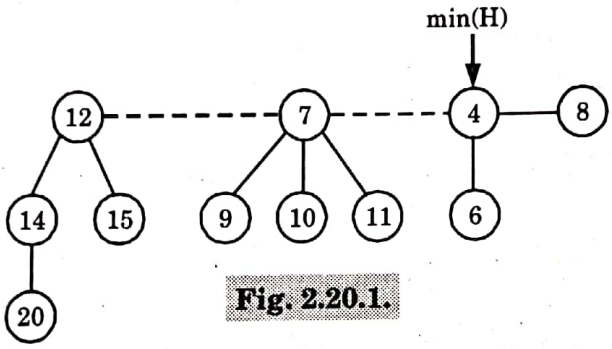


Fig. 2.20.1.

3. Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key. This node is called the minimum node.
4. If Fibonacci heap H is empty, then min[H] = NIL.

Applications of Fibonacci heap :

1. Fibonacci heap is used for Dijkstra's algorithm because it improves the asymptotic running time of this algorithm.
2. It is used in finding the shortest path. These algorithms run in $O(n^2)$ time if the storage for nodes is maintained as a linear array.

Que 2.21. What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.

AKTU 2015-16, Marks 15

Answer

Fibonacci heap : Refer Q. 2.20, Page 2–40B, Unit-2.

CONSOLIDATE operation :

CONSOLIDATE(*H*)

1. for $i \leftarrow 0$ to $D(n[H])$
2. do $A[i] \leftarrow$ NIL
3. for each node w in the root list of H
4. do $x \leftarrow w$
5. $d \leftarrow$ degree[x]
6. while $A[d] \neq$ NIL
7. do $y \leftarrow A[d]$ ▷ Another node with the same degree as x.

8. if key[x] > key[y]
9. then exchange $x \leftrightarrow y$
10. FIB-HEAP-LINK(H, y, x)
11. $A[d] \leftarrow$ NIL
12. $d \leftarrow d + 1$
13. $A[d] \leftarrow x$
14. min[H] $\leftarrow$ NIL
15. for $i \leftarrow 0$ to $D(n[H])$
16. do if $A[i] \neq$ NIL
17. then add $A[i]$ to the root list of H
18. if min[H] = NIL or key [$A[i]$] < key[min[H]]
19. then min[H] $\leftarrow A[i]$

FIB-HEAP-LINK(H, y, x)

1. remove y from the root list of H
2. make y a child of x, incrementing degree[x]
3. mark[y] $\leftarrow$ FALSE

Que 2.22. Define Fibonacci heap. Discuss the structure of a Fibonacci heap with the help of a diagram. Write a function for uniting two Fibonacci heaps.

Answer

Fibonacci heap : Refer Q. 2.20, Page 2–40B, Unit-2.

Structure of Fibonacci heap :

1. **Node structure :**
   a. The field "mark" is True if the node has lost a child since the node became a child of another node.
   b. The field "degree" contains the number of children of this node. The structure contains a doubly-linked list of sibling nodes.

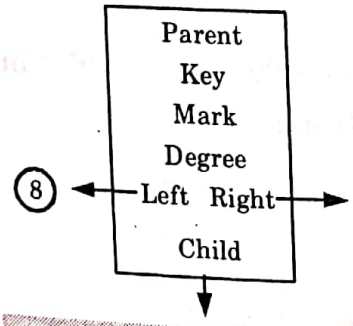


Fig. 2.22.1. Node structure.

2. **Heap structure :**
   min(H) : Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key; this node is called the minimum node. If Fibonacci heap H is empty, then min[H] = NIL.

$n(H)$: Number of nodes in heap H

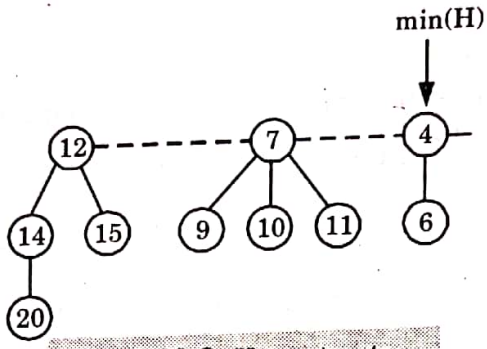


Fig. 2.22.2. Heap structure.

Function for uniting two Fibonacci heap :

Make-Heap :

MAKE-FIB-HEAP( )

allocate(H)

min(H) = NIL

$n(H) = 0$

FIB-HEAP-UNION(H_1, H_2)

1. $H \leftarrow$ MAKE-FIB-HEAP( )
2. min[H] $\leftarrow$ min[H_1]
3. Concatenate the root list of H_2 with the root list of H
4. if (min[H_1] = NIL) or (min[H_2] $\neq$ NIL and min[H_2] < min[H_1])
5. then min[H] $\leftarrow$ min[H_2]
6. $n[H] \leftarrow n[H_1] + n[H_2]$
7. Free the objects H_1 and H_2
8. return H

Que 2.23. Discuss following operations of Fibonacci heap :

i. **Make-Heap**
ii. **Insert**
iii. **Minimum**
iv. **Extract-Min**

Answer

i. **Make-Heap :** Refer Q. 2.22, Page 2–42B, Unit-2.

ii. **Insert : (H, x)**

1. degree[x] $\leftarrow$ 0
2. $p[x] \leftarrow$ NIL
3. child[x] $\leftarrow$ NIL

4. left[x] $\leftarrow x$
5. right[x] $\leftarrow x$
6. mark[x] $\leftarrow$ FALSE
7. concatenate the root list containing x with root list H
8. if min[H] = NIL or key[x] < key[min[H]]
9. then min[H] $\leftarrow x$
10. $n[H] \leftarrow n[H] + 1$

To determine the amortized cost of FIB-HEAP-INSERT, Let H be the input Fibonacci heap and H' be the resulting Fibonacci heap, then $t(H) = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is,

$(t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1$

Since the actual cost is O(1), the amortized cost is O(1) + 1 = O(1)

iii. Minimum :

The minimum node of a Fibonacci heap H is always the root node given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

iv. FIB-HEAP-EXTRACT-MIN(H)

1. $z \leftarrow$ min[H]
2. if $z \neq$ NIL
3. then for each child x of z
4. do add x to the root list of H
5. $p[x] \leftarrow$ NIL
6. remove z from the root list of H
7. if z = right[z]
8. then min[H] $\leftarrow$ NIL
9. else min[H] $\leftarrow$ right[z]
10. CONSOLIDATE (H)
11. $n[H] \leftarrow n[H] - 1$
12. return z

Que 2.24. What is tries ? What are the properties of tries ?

Answer

1. A trie (digital tree / radix tree / prefix free) is a kind of search tree *i.e.*, an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
2. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.

3. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
4. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

Properties of a trie :

1. Tries is a multi-way tree.
2. Each node has from 1 to d children.
3. Each edge of the tree is labeled with a character.
4. Each leaf node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Que 2.25. Write an algorithm to search and insert a key in tries data structure.

Answer

Search a key in tries :

Trie-Search(t, $P[k..m]$) // inserts string P into t

1. if t is leaf then return true
2. else if t.child($P[k]$) = nil then return false
3. else return Trie-Search(t.child($P[k]$), $P[k + 1..m]$)

Insert a key in tries :

Trie-Insert(t, $P[k..m]$)

1. if t is not leaf then //otherwise P is already present
2. if t.child($P[k]$) = nil then

   //Create a new child of t and a "branch" starting with that child and storing $P[k..m]$
3. else Trie-Insert(t.child($P[k]$), $P[k + 1..m]$)

Que 2.26. What is skip list ? What are its properties ?

Answer

1. A skip list is built in layers.
2. The bottom layer is an ordinary ordered linked list.
3. Each higher layer acts as an "express lane", where an element in layer i appears in layer $(i + 1)$ with some fixed probability p (two commonly used values for p are ½ and ¼.).
4. On average, each element appears in $1/(1-p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists.

5. The skip list contains $\log_{1/p} n$ (*i.e.*, logarithm base $1/p$ of n).

Properties of skip list :

1. Some elements, in addition to pointing to the next element, also point to elements even further down the list.
2. A level k element is a list element that has k forward pointers.
3. The first pointer points to the next element in the list, the second pointer points to the next level 2 element, and in general, the i^{th} pointer points to the next level i element.

Que 2.27. Explain insertion, searching and deletion operation in skip list.

Answer

Insertion in skip list :

1. We will start from highest level in the list and compare key of next node of the current node with the key to be inserted.
2. If key of next node is less than key to be inserted then we keep on moving forward on the same level.
3. If key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

```
Insert(list, searchKey)
1.  local update[0...MaxLevel+1]
2.  x : = list → header
3.  for i : = list → level down to 0 do
4.  while x → forward[i] → key  forward[i]
5.  update[i] : = x
6.  x : = x → forward[0]
7.  lvl : = randomLevel()
8.  if lvl > list → level then
9.  for i : = list → level + 1 to lvl do
10. update[i] : = list → header
11. list → level : = lvl
12. x : = makeNode(lvl, searchKey, value)
13. for i : = 0 to level do
14. x → forward[i] : = update[i] → forward[i]
15. update[i] → forward[i] : = x
```

Searching in skip list :

Search(list, searchKey)

1. x : = list → header
2. loop invariant : x → key level down to 0 do
3. while x → forward[i] → key forward[i]
4. x : = x → forward[0]
5. if x → key = searchKey then return x → value
6. else return failure

Deletion in skip list :

Delete(list, searchKey)

1. local update[0..MaxLevel+1]
2. x : = list → header
3. for i : = list → level down to 0 do
4. while x → forward[i] → key forward[i]
5. update[i] : = x
6. x : = x → forward[0]
7. if x → key = searchKey then
8. for i : = 0 to list → level do
9. if update[i] → forward[i] ≠ x then break
10. update[i] → forward[i] : = x → forward[i]
11. free(x)
12. while list → level > 0 and list → header → forward[list → level] = NIL do
13. list → level : = list → level – 1

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define red-black tree and give its properties.

Ans. Refer Q. 2.1.

Q. 2. Explain the insertion and deletion operation in a red-black tree.

Ans. **Insertion :** Refer Q. 2.1.
Deletion : Refer Q. 2.6.