PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Divide and Conquer Approach
Binary Search

Dr. Shylaja S S

Divide-and-Conquer Approach

Divide-and-Conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

The divide-and-conquer technique is diagrammed in Fig. 1, which depicts the case of dividing a problem into two smaller sub problems, by far the most widely occurring case.

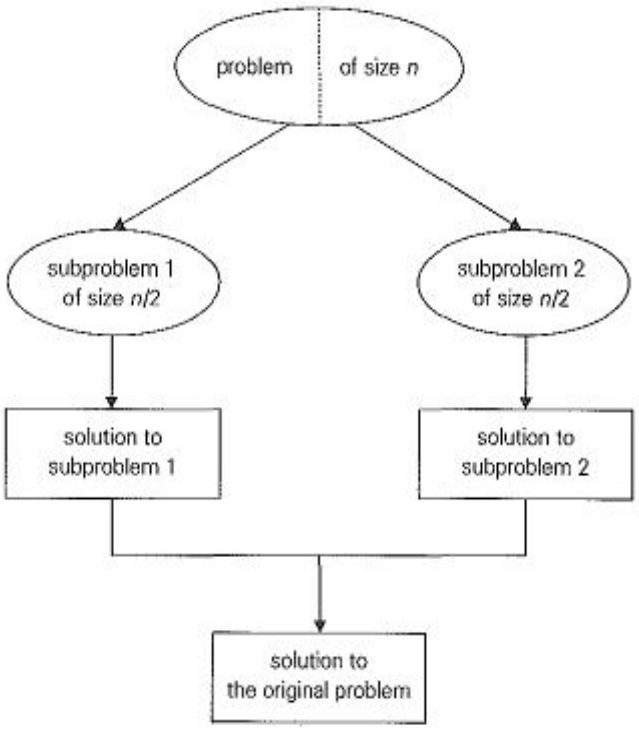


Fig. 1: Divide-and-conquer technique (typical case)

General Divide and Conquer Recurrence

In the most typical cases of Divide and Conquer, a problem's instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. Here a and b are constants; a >= 1 and b >= 1. Assuming that size n is a power of b, we get the following recurrence for the running time:

$T(n) = a * T(n/b) + f(n)$

f(n) is a function that accounts for the time spent on dividing the problem and combining the solutions. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem.

Master Theorem

For the recurrence:

$$T(n) = a * T(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$, where d >= 0 in the recurrence relation, then:

If $a < b^d$, $T(n) \in \Theta(n^d)$

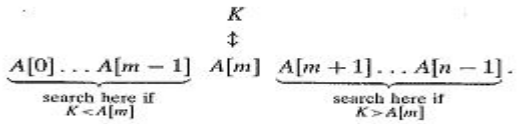
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Analogous results hold for O and Ω as well!

Binary Search

Binary Search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing the search key K with the array's middle element A[m]. If they match, the algorithm stops. Otherwise, the same operation is repeated recursively for the first half of the array if K < A[m] and for the second half if K > A[m].



As an example, let us apply binary search to searching for K = 70 in the array. The iterations of the algorithm are given in the following table.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration1	l						m						r
iteration2								l		m			r
iteration3								l,m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a non recursive algorithm, too. Here is a pseudo code for this non recursive version.

ALGORITHM BinarySearch(A[0 .. n -1], K)

// Implements non recursive binary search

// Input: An array A [0 ... n - 1] sorted in ascending order and a search key K

// Output: An index of the array's element that is equal to K or -1 if there is no //such element

l ← 0; r ← n-1

while l ≤ r do

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if K = A[m] return m

else if K < A[m] r ← m-1

else l←m+1

return -1

Binary Search Analysis

Worst Case: The basic operation is the comparison of the search key with an element of the array. The number of comparisons made is given by the following recurrence:

$$\boldsymbol{C_{worst}(n) = C_{worst}\left(\lfloor n/2 \rfloor\right) + 1 \text{ for } n > 1, C_{worst}(1) = 1}$$

For the initial condition $C_{worst}(1) = 1$, we obtain:

$$\boldsymbol{C_{worst}(2^k) = k + 1 = \log_2 n + 1}$$

For any arbitrary positive integer, n:

$$\boldsymbol{C_{worst}(n) = \lfloor \log_2 n \rfloor + 1}$$

Average Case:

$$\boldsymbol{C_{avg} \approx \log_2 n}$$

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Merge Sort

Dr. Shylaja S S

Merge Sort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array A[0 .. n - 1] by dividing it into two halves A[0 .. $\lfloor n/2 \rfloor$ -1] and A[$\lfloor n/2 \rfloor$ … n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM Mergesort(A[0 .. n - 1])
//Sorts array A[0 .. n - 1] by recursive mergesort
//Input: An array A[0 .. n - 1] of orderable elements
//Output: Array A[0 .. n - 1] sorted in nondecreasing order
if n > 1
copy A[0 .. $\lfloor n/2 \rfloor$ -1] to B[0 .. $\lfloor n/2 \rfloor$ -1]
copy A[$\lfloor n/2 \rfloor$.. n -1] to C[0 .. $\lceil n/2 \rceil$ -1]
Mergesort(B[0 .. $\lfloor n/2 \rfloor$ - 1])
Mergesort(C[0 .. $\lceil n/2 \rceil$ -1])
Merge(B, C, A)

The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM Merge(B[0 .. p- 1], C[0 .. q -1], A[0 .. p + q -1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 .. p -1] and C[0 .. q -1] both sorted
//Output: Sorted array A[0 .. p + q -1] of the elements of B and C
i←0; j←0; k←0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ←C[j]; j← j + 1
    k←k+1

```
if i =  p
        copy C[j .. q-1] to A[k .. p + q - 1]
else
        copy B[i .. p -1] to A[k .. p + q -1]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Fig. 1.

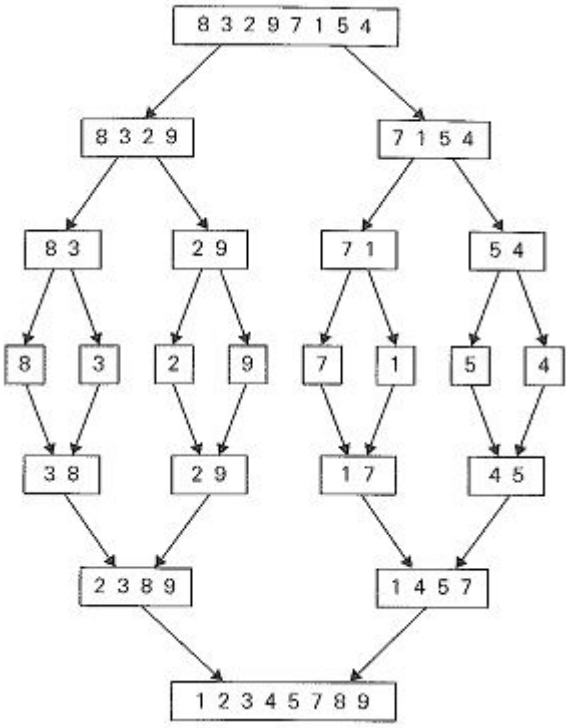


Fig. 1: Example of mergesort operation

Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

$C(n) = 2C(n/2) + C_{merge}(n)$ [for $n > 1$], $C(1) = 0$

The number of key comparisons performed during the merging stage in the worst case is:

$C_{merge}(n) = n - 1$

Using the above equation:

$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ [for $n > 1$], $C_{worst}(1) = 0$

Applying Master Theorem to the above equation:

$C_{worst}(n) \in \Theta (n \log n)$

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Quick Sort

Dr. Shylaja S S

Quick Sort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array A[0 .. n - 1] to achieve its partition, a situation where all the elements before some position s are smaller than or equal to A[s] and all the elements after positions are greater than or equal to A[s]:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition has been achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two sub arrays of the elements preceding and following A[s] independently (e.g., by the same method).

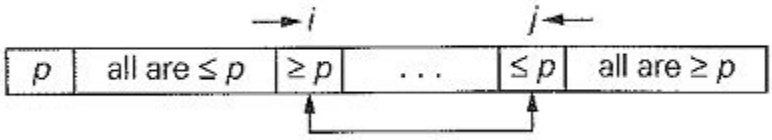
```
ALGORITHM  Quicksort(A[l .. r])
// Sorts a subarray by quicksort
// Input: A subarray A[l … r] of A[0 .. n  -1], defined by its left and right indices l
//and r
// Output: Subarray A[l .. r]  sorted in non decreasing order
if l  <  r
        s ←Partition(A[l .. r])       //s is a split position
        Quicksort(A[l  .. s - 1])
        Quicksort(A[s + 1 .. r])
```

A partition of A [0 ... n- 1] and, more generally, of its subarray A[l .. r] (0 ≤ l < r ≤ n - 1) can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the pivot. There are several different strategies for selecting a pivot. For now, we use the simplest strategy of selecting the subarray's first element: p = A[l].

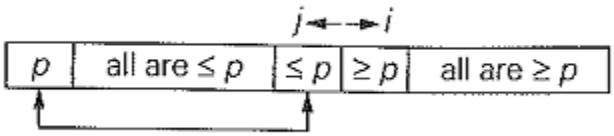
There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of

the subarray: one is left-to-right and the other right-to-left, each comparing the sub-array's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the sub array, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

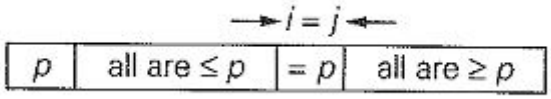
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., i > j, we will have partitioned the array after exchanging the pivot with A[j]:



Finally, if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal top (why?). Thus, we have the array partitioned, with the split positions s = i = j:



We can combine the last case with the case of crossed-over indices (i > j) by exchanging the pivot with A[j] whenever i ≥j.

Here is a pseudocode implementing this partitioning procedure.

```
ALGORITHM  Partition(A[l .. r])
// Partitions a subarray by using its first element as a pivot
// Input: A  subarray A[l..r] of A[0 .. n  - 1],  defined by its left and right  indices l
// and r  (l  <  r)
// Output: A partition of A[l .. r ], with the split position returned as this
//function's value
p ←A[l]
i ← l;  j ← r  + 1
repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j  - 1 until A[j] ≤ p
        swap(A[i],  A[j])
until i ≥ j
swap(A[i],  A[j])       //undo last swap when i ≥ j
swap(A[i],  A[j])
return  j
```

Note that index i can go out of the subarray bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array A[0 .. n- 1] a "sentinel" that would prevent index i from advancing beyond position n. The more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Fig. 1.

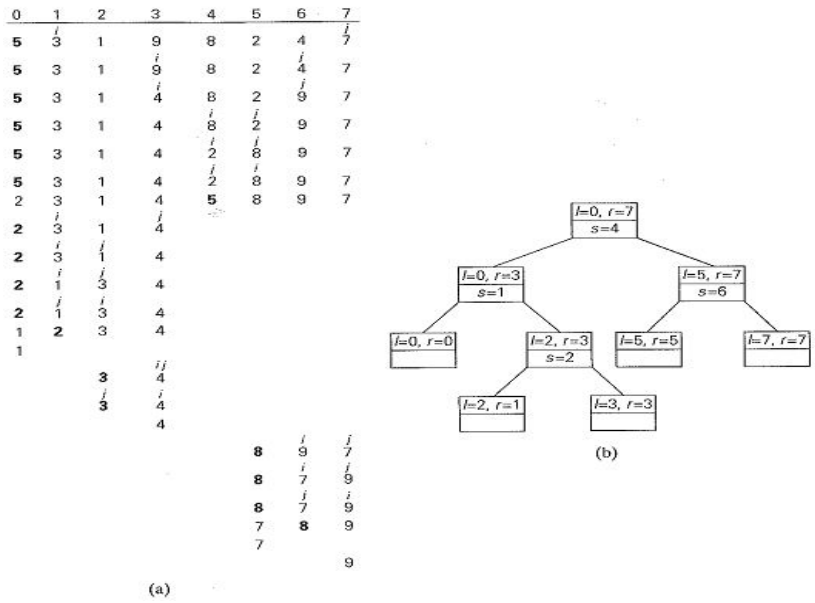


Fig. 1: Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to Quicksort with input values l and r of subarray bounds and split positions of a partition obtained.

Quick sort Analysis:

Best Case: The number of comparisons in the best case satisfies the recurrence:

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$

According to Master Theorem

$$C_{best}(n) \in \Theta(n \log_2 n)$$

Worst Case: The number of comparisons in the worst case satisfies the recurrence:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Average Case: Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n.

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \; for \; n > 1$$

The solution for the above recurrence is:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n$$

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Binary Tree

Dr. Shylaja S S

Binary Tree

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called as the left and right subtree of the root. The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique. The binary tree is a Divide – And – Conquer ready structure.

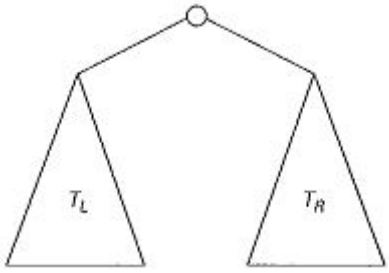


Fig. 1: Standard representation of a Binary Tree

Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if T = Ø return -1

else return max(Height(T_L), Height(T_R))+ 1

Height of a Binary Tree Analysis:

We measure the problem's instance size by the number of nodes n(T) in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions A(n(T)) made by the algorithm are the same.

We have the following recurrence relation for A(n(T)):

$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$ for $n(T) > 0$,

$A(0) = 0$.

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking-and this is very typical for binary tree algorithms-that the

tree is not empty. For example, for the empty tree, the comparison T = Ø is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are three and one, respectively.

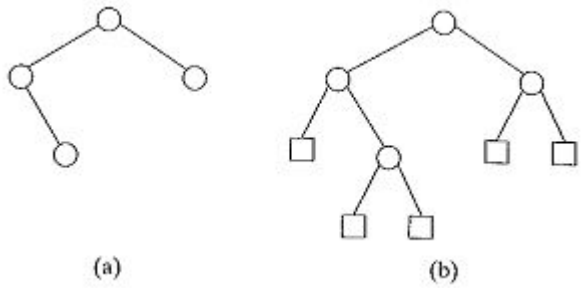


Fig. 2: (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Fig. 2) are called external; the original nodes (shown by little circles) are called internal. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node. Thus, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. Checking Fig. 2 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n:

$$x=n+1 \qquad \text{---------- (1)}$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (1).

Note that equation (1) also applies to any nonempty full binary tree, in

which, by definition, every node has either zero or two children: for a **full binary tree**, n and x denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm Height, the number of comparisons to check whether the tree is empty is

$$C(n)=n+x=2n+1,$$

while the number of additions is

$$A(n) = n$$

Binary Tree Traversal

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ just by the timing of the root's visit:

In the **preorder traversal**, the root is visited before the left and right subtrees are visited (in that order).

In the **inorder traversal**, the root is visited after visiting its left subtree but before visiting the right subtree.

In the **postorder traversal**, the root is visited after visiting the left and right subtrees (in that order).

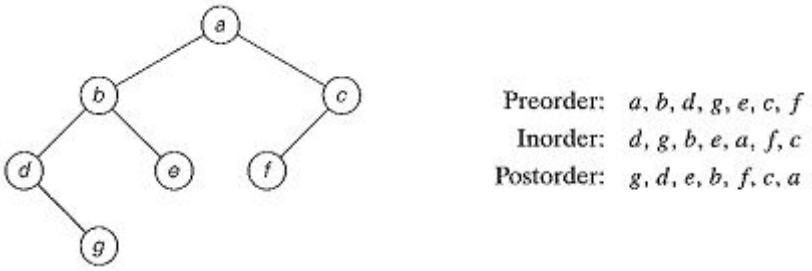


Fig. 3: Binary tree and its traversals

Algorithm Inorder(T)
if T ≠ Ø
    Inorder(T_{left})
    print(root of T)
    Inorder(T_{right})

Algorithm Preorder(T)
if T ≠ Ø
    print(root of T)
    Preorder(T_{left})
    Preorder(T_{right})

Algorithm Postorder(T)
if T ≠ Ø
    Postorder(T_{left})
    Postorder(T_{right})
    print(root of T)

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Multiplication of Large Integers and Strassen's Matrix Multiplication

Dr. Shylaja S S

Multiplication of large Integers

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use the classic pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad a shorter number with leading zeros to equal their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, it turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two - digit integers, say, 23 and 14. These numbers can be represented as follows:

$23 = 2. 10^1 + 3. 10^0$ and $14 = 1. 10^1 + 4. 10^0$

Now let us multiply them:

$23 * 14 = (2 . 10^1 + 3. 10^0) * (1 . 10^1 + 4. 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products 2 * 1 and 3 * 4 that need to be computed anyway:

$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit integers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$c = a * b = c_2 10^2 + c_1 10^1 + c_0$,

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle-after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0 , respectively. In these notations, a= a_1a_0 implies that a= $a_1 10^{n/2} + a_0$, and b = b_1b_0 implies that b = $b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$c = a*b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1)10^n + (a_1*b_0 + a_0*b_1)10^{n/2} + (a_0*b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0$

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0

If n/2 is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes one. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of n/2-digit numbers, the recurrence for the number of multiplications M (n) will be

$M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

Solving it by backward substitutions for $n = 2^k$ yields

$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2})$

$= \ldots = 3^i M(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k$

Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$

Strassen's Matrix Multiplication

Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying

matrices. Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Thus, to multiply two 2-by-2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2-by-2 matrices by Strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity.

Let A and B be two n-by-n matrices where n is a power of two. (If n is not a power of two, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n /2-by-n /2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as

$A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of n/2-by-n/2 matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation for it:

$M(n)=7M(n/2)$ for $n> 1$, $M(1)=1$.

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \ldots$$

$$= 7^i M(2^{k-i}) \ldots = 7^k M(2^{k-k}) = 7^k$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

which is smaller than n^3 required by the brute-force algorithm.

Since this saving in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions A(n) made by Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order n/2 and make 18 additions of matrices of size n/2; when $n = 1$, no additions are made since two numbers are simply multiplied.
These observations yield the following recurrence relation:
$A(n) = 7\,A(n/2) + 18(n/2)^2$ for $n>1$, $A(1) =0$

According to the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Heap and Heap Sort

Heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:
1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

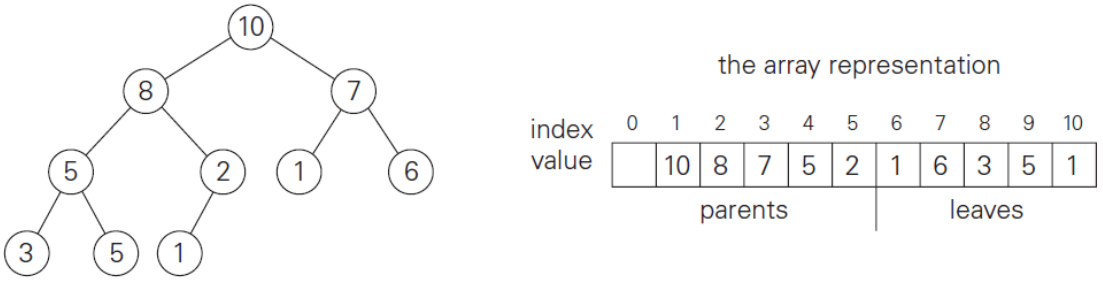


FIGURE 6.10 Heap and its array representation.

properties of heaps

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor.$$

Bottom Up Construction

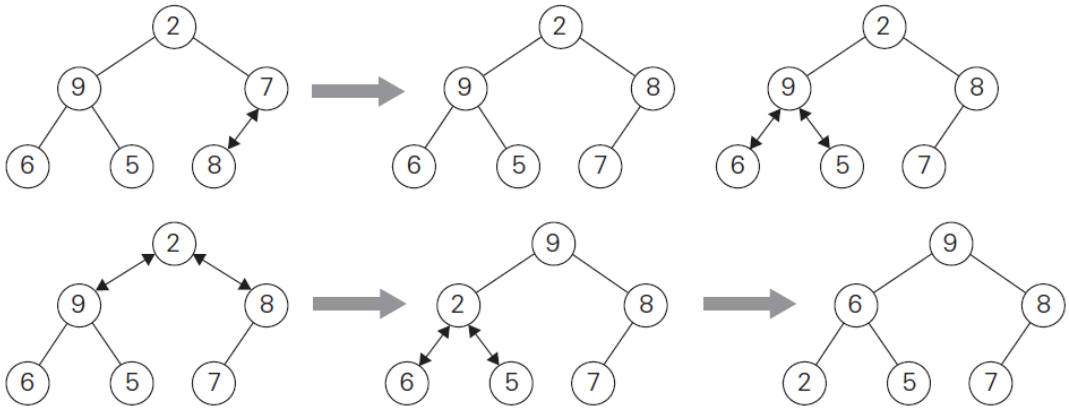


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

Top Down Construction

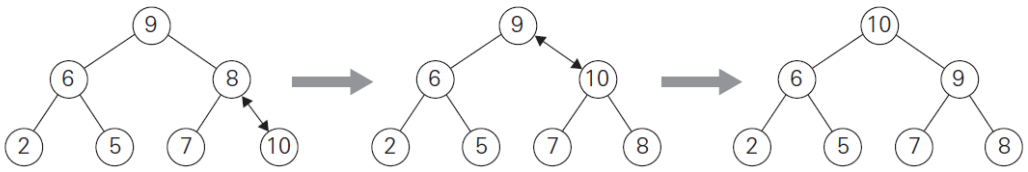


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

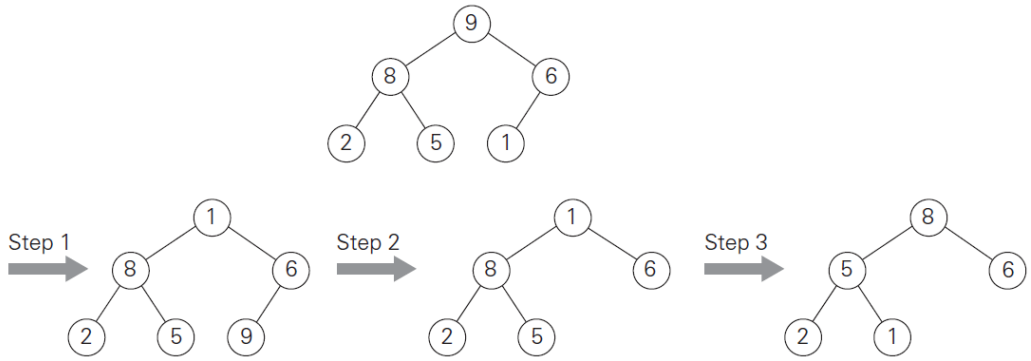


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Heapsort

Now we can describe ***heapsort***—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

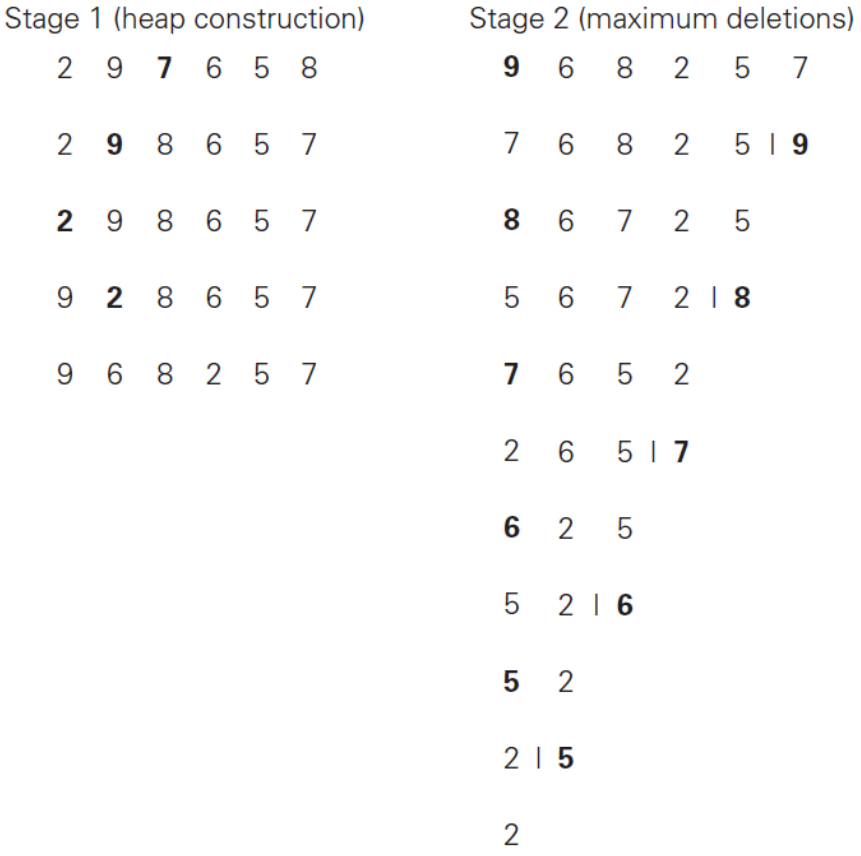


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \le 2n \log_2 n.$$

Red Black Tree

Introduction:

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Interesting points about Red-Black Tree:

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is $h <= 2 \log_2(n + 1)$.

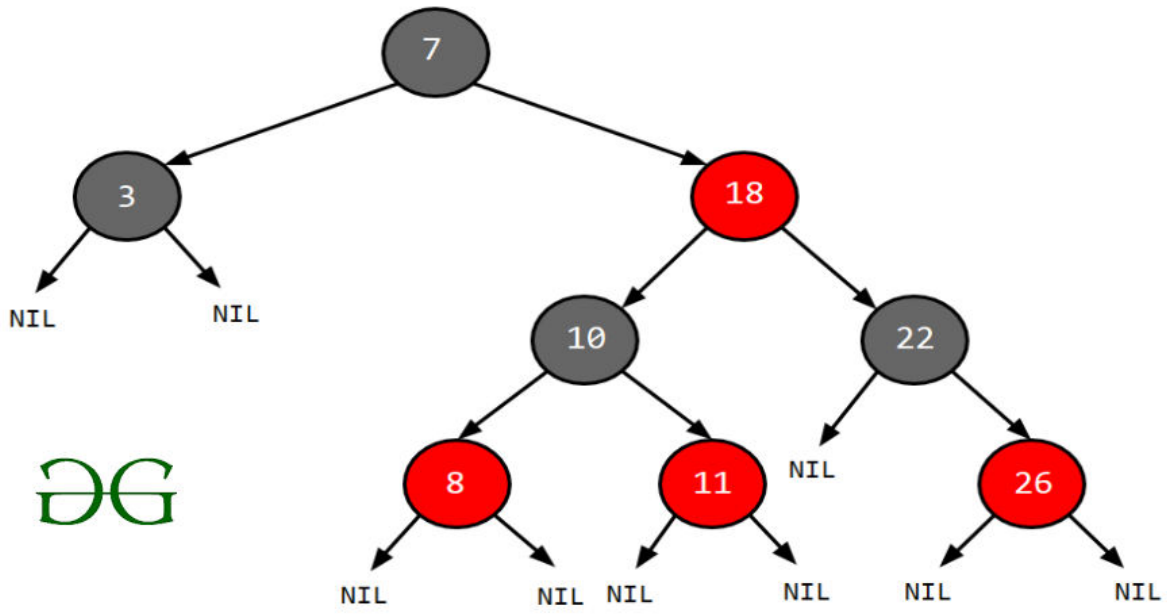
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

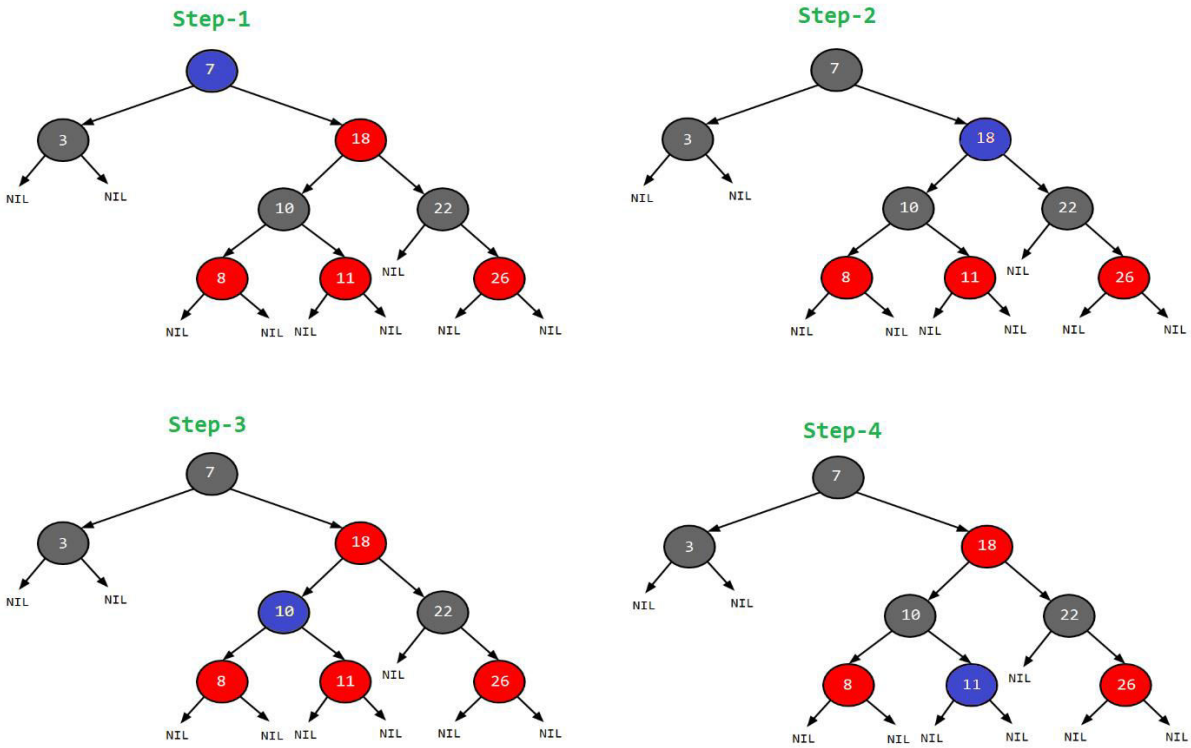
Search Operation in Red-black Tree:
As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

searchElement (tree, val)

Step 1:
If tree -> data = val OR tree = NULL
  Return tree
Else
If val data
    Return searchElement (tree -> left, val)
  Else
    Return searchElement (tree -> right, val)
  [End of if]
[End of if]

Step 2: END





Applications:

1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
2. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
3. Besides they are used in the K-mean clustering algorithm for reducing time complexity.
4. Moreover, MySQL also uses the Red-Black tree for indexes on tables.

Insertion in Red Black Tree:

Algorithm:

Let x be the newly inserted node.

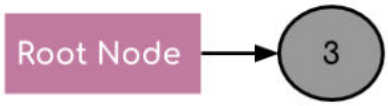
1. Perform standard BST insertion and make the colour of newly inserted nodes as RED.
2. If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).
3. Do the following if the color of x's parent is not BLACK **and** x is not the root.

   a) If x's uncle is RED (Grandparent must have been black from property

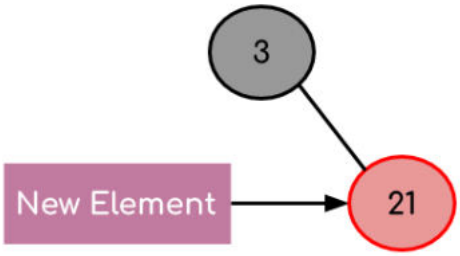
   4). (i) Change the colour of parent and uncle as BLACK.

(ii) Colour of a grandparent as RED.
(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.
b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)
(i) Left Left Case (p is left child of g and x is left child of p)
(ii) Left Right Case (p is left child of g and x is the right child of p)
(iii) Right Right Case (Mirror of case i)
(iv) Right Left Case (Mirror of case ii)

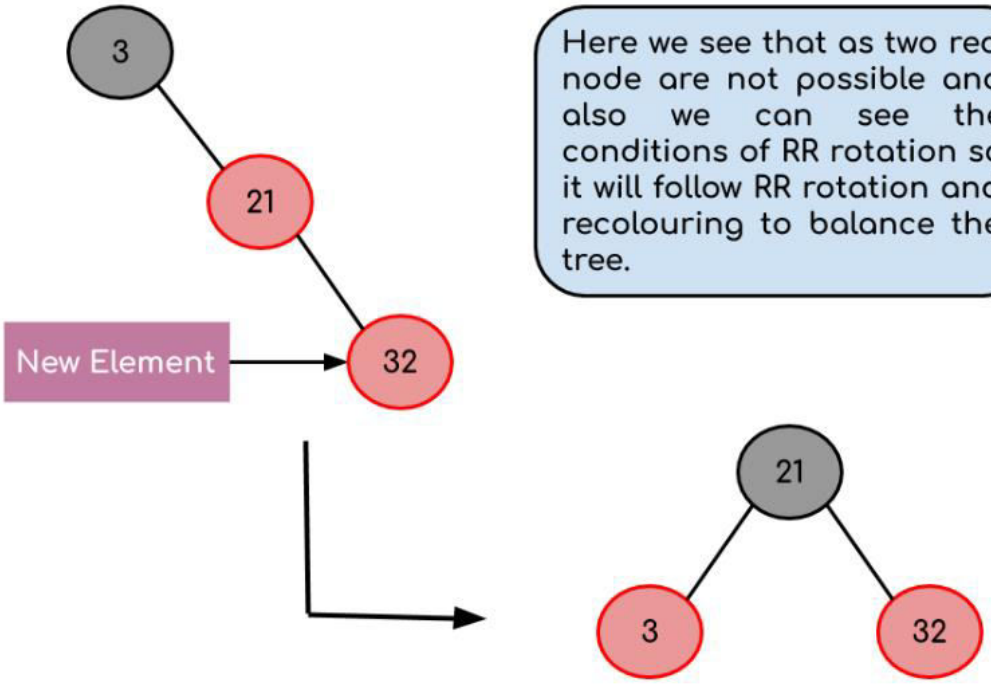
Step 1: Inserting element 3 inside the tree.



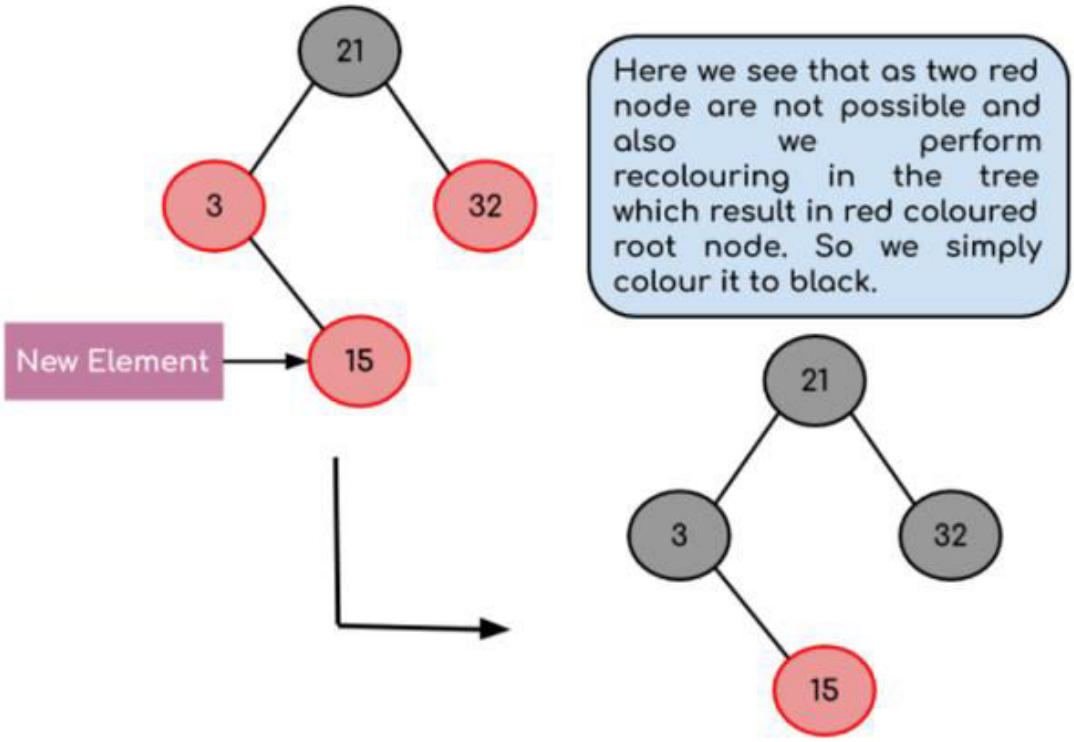
Step 2: Inserting element 21 inside the tree.



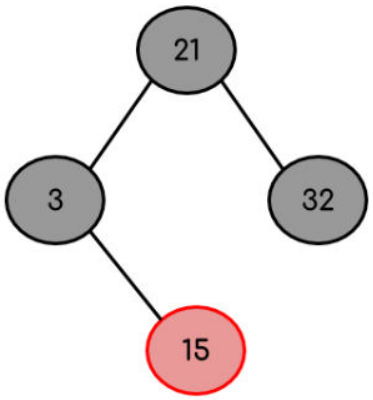
Step 3: Inserting element 32 inside the tree.



Step 4: Inserting element 15 inside the tree.



Final Red Black Tree:



Red Black Tree

Introduction:

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Interesting points about Red-Black Tree:

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is $h <= 2 \log_2(n + 1)$.

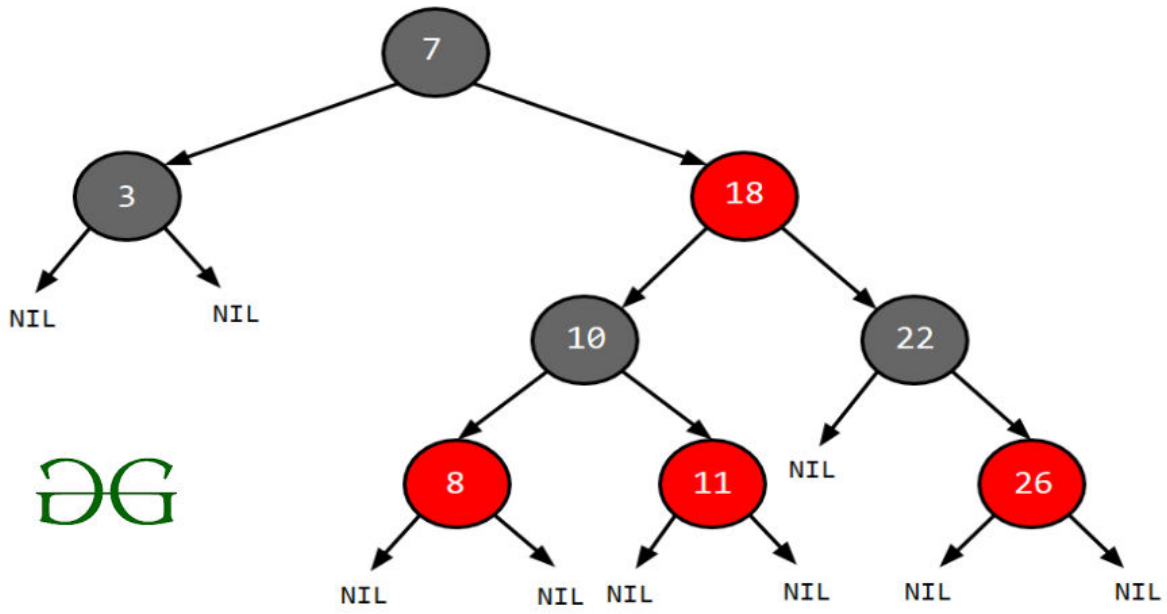
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

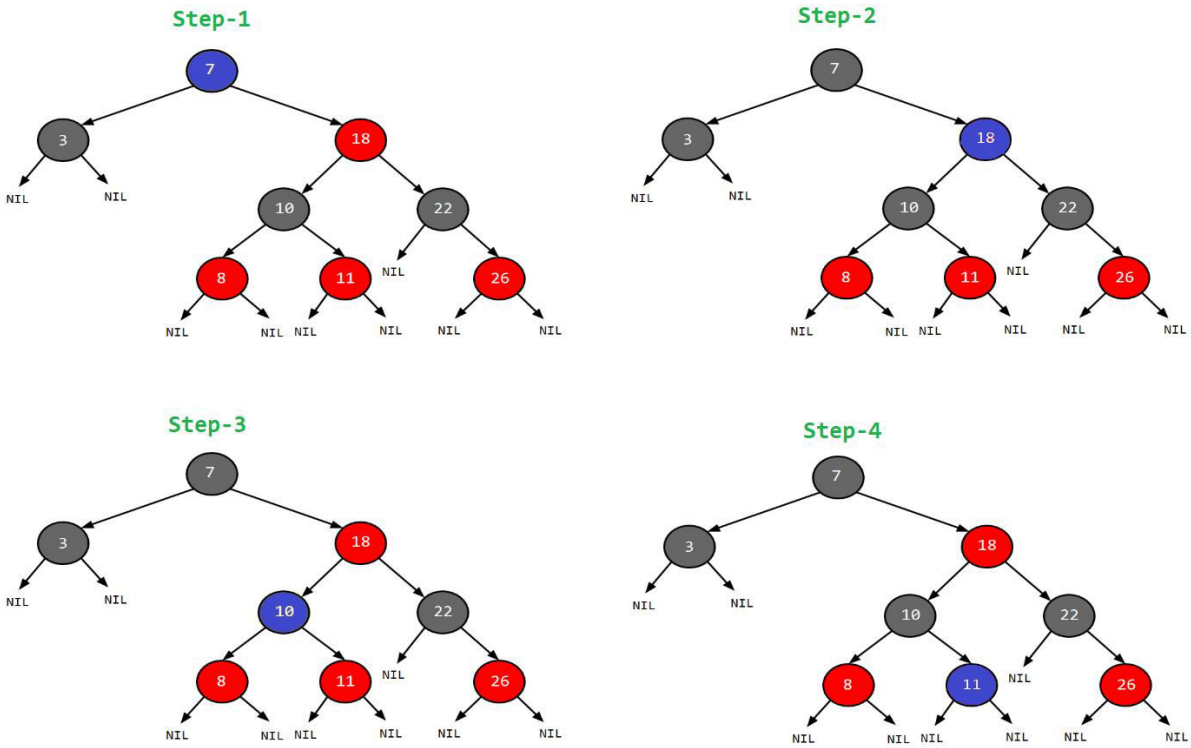
Search Operation in Red-black Tree:
As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

searchElement (tree, val)

Step 1:
If tree -> data = val OR tree = NULL
  Return tree
Else
If val data
    Return searchElement (tree -> left, val)
  Else
    Return searchElement (tree -> right, val)
  [End of if]
[End of if]

Step 2: END





Applications:

1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
2. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
3. Besides they are used in the K-mean clustering algorithm for reducing time complexity.
4. Moreover, MySQL also uses the Red-Black tree for indexes on tables.

Insertion in Red Black Tree:

Algorithm:

Let x be the newly inserted node.

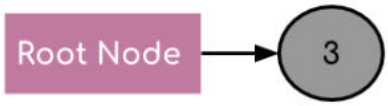
1. Perform standard BST insertion and make the colour of newly inserted nodes as RED.
2. If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).
3. Do the following if the color of x's parent is not BLACK **and** x is not the root.

   a) If x's uncle is RED (Grandparent must have been black from property

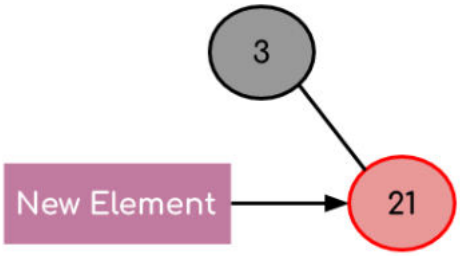
   4). (i) Change the colour of parent and uncle as BLACK.

(ii) Colour of a grandparent as RED.
(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.
b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)
(i) Left Left Case (p is left child of g and x is left child of p)
(ii) Left Right Case (p is left child of g and x is the right child of p)
(iii) Right Right Case (Mirror of case i)
(iv) Right Left Case (Mirror of case ii)

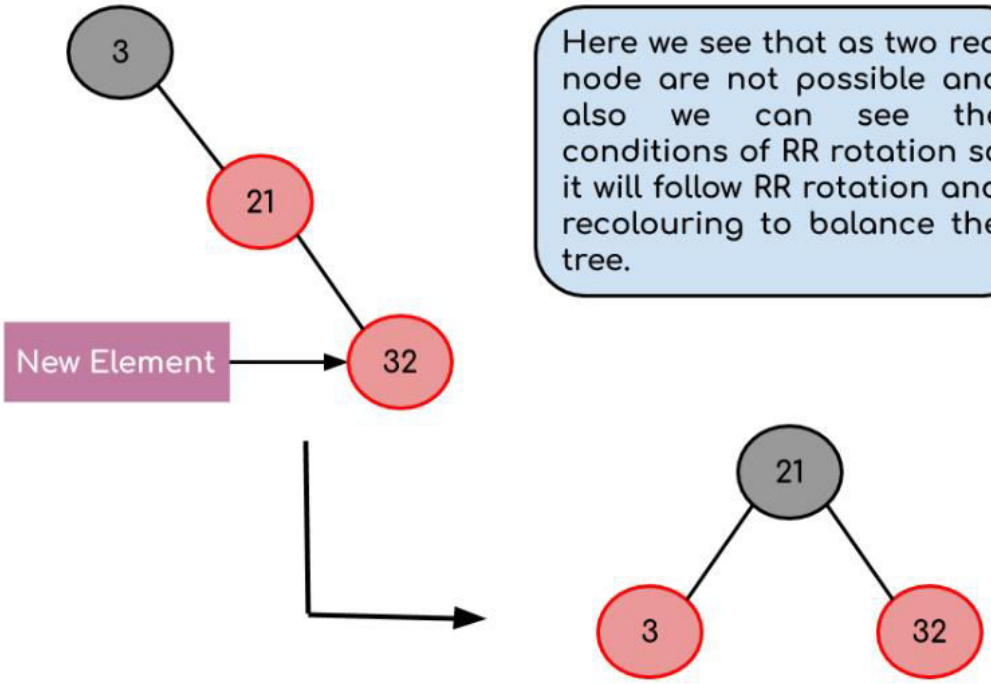
Step 1: Inserting element 3 inside the tree.



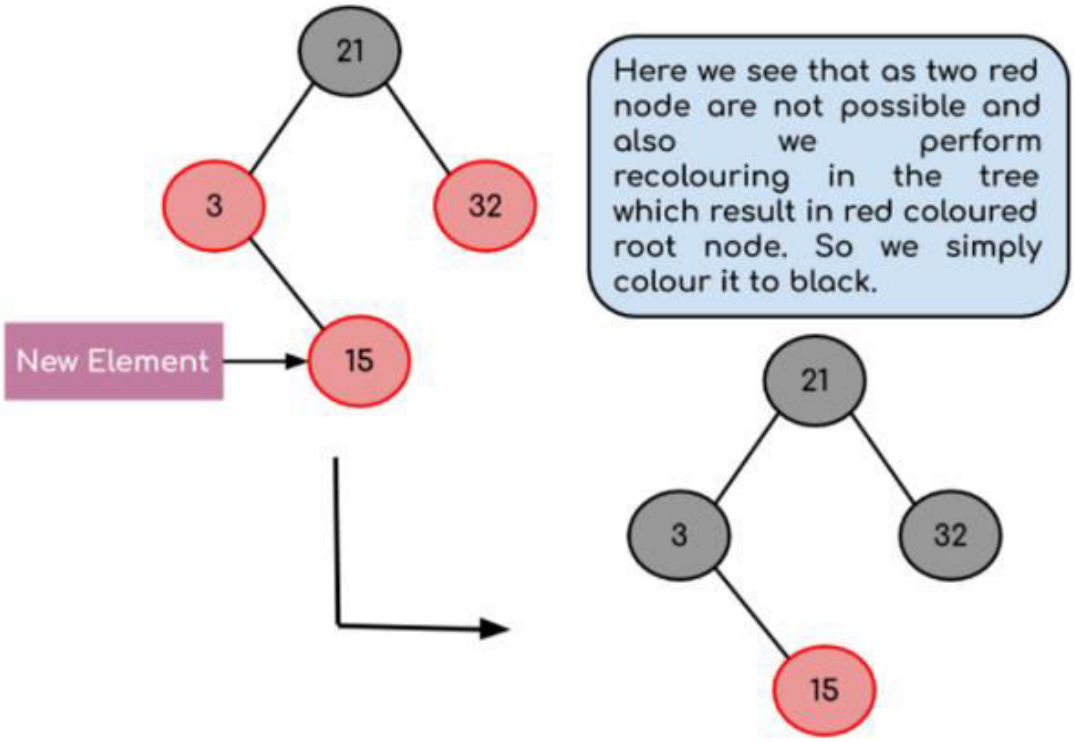
Step 2: Inserting element 21 inside the tree.



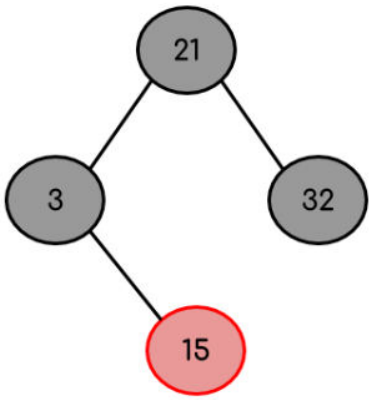
Step 3: Inserting element 32 inside the tree.



Step 4: Inserting element 15 inside the tree.



Final Red Black Tree:



2-3 Tree

a **2–3 tree** is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. A 2-3 tree is a B-tree of order 3. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements. 2−3 trees were invented by John Hopcroft in 1970

2–3 trees are required to be balanced, meaning that each leaf is at the same level. It follows that each right, center, and left subtree of a node contains the same or close to the same amount of data.

We say that an internal node is a **2-node** if it has *one* data element and *two* children.

We say that an internal node is a **3-node** if it has *two* data elements and *three* children.

We say that *T* is a **2–3 tree** if and only if one of the following statements hold:

- *T* is empty. In other words, *T* does not have any nodes.
- *T* is a 2-node with data element *a*. If *T* has left child *p* and right child *q*, then
    - *p* and *q* are 2–3 trees of the same height
    - *a* is greater than each element in *p*; and
    - *a* is less than or equal to each data element in *q*.
- *T* is a 3-node with data elements *a* and *b*, where $a < b$. If *T* has left child *p*, middle child *q*, and right child *r*, then
    - *p*, *q*, and *r* are 2–3 trees of equal height;
    - *a* is greater than each data element in *p* and less than or equal to each data element in *q*; and
    - *b* is greater than each data element in *q* and less than or equal to each data element in *r*.

Properties

- Every internal node is a 2-node or a 3-node.
- All leaves are at the same level.
- All data is kept in sorted order.

Searching

Searching for an item in a 2–3 tree is similar to searching for an item in a binary search tree. Since the data elements in each node are ordered, a search function will be directed to the correct subtree and eventually to the correct node which contains the item.

1. Let *T* be a 2–3 tree and *d* be the data element we want to find. If *T* is empty, then *d* is not in *T* and we're done.
2. Let *t* be the root of *T*.
3. Suppose *t* is a leaf.
    - If *d* is not in *t*, then *d* is not in *T*. Otherwise, *d* is in *T*. We need no further steps and we're done.
4. Suppose *t* is a 2-node with left child *p* and right child *q*. Let *a* be the data element in *t*. There are three cases:
    - If *d* is equal to *a*, then we've found *d* in *T* and we're done.

- If d<a, then set *T* to *p*, which by definition is a 2–3 tree, and go back to step 2.
- If d<a, then set *T* to *q* and go back to step 2.

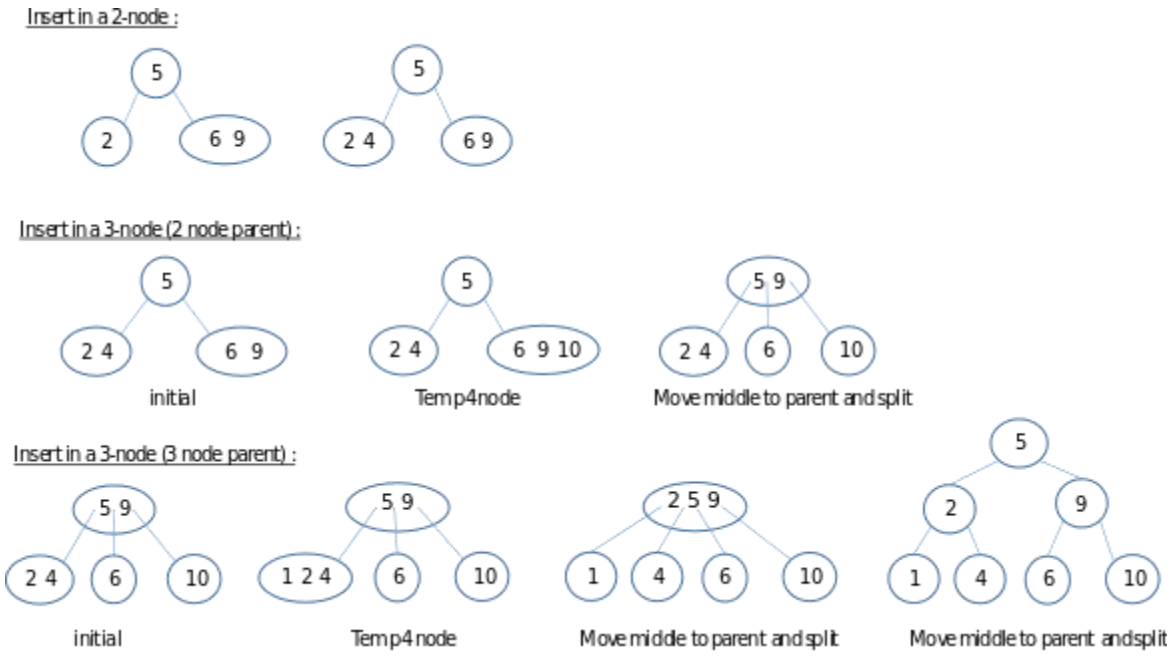
5. Suppose *t* is a 3-node with left child *p*, middle child *q*, and right child *r*. Let *a* and *b* be the two data elements of *t*, where a<b. There are four cases:
   - If *d* is equal to *a* or *b*, then *d* is in *T* and we're done.
   - If d<a, then set *T* to *p* and go back to step 2.
   - If d<a, then set *T* to *q* and go back to step 2.
   - If d<a, then set *T* to *r* and go back to step 2.

Insertion

Insertion maintains the balanced property of the tree.[5]

To insert into a 2-node, the new key is added to the 2-node in the appropriate order.

To insert into a 3-node, more work may be required depending on the location of the 3-node. If the tree consists only of a 3-node, the node is split into three 2-nodes with the appropriate keys and children.



The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
    - remove the node containing k
    - "steal" one of the sibling's children
    - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
    - remove the node containing k
    - make n's remaining child a child of n's sibling
    - fix leftMax and middleMax fields of n's sibling as needed
    - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

B Tree

B Tree is a Self-balancing search tree that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems. B-trees were invented by Rudolf Bayer and Edward M.McCreight while working at Boeing Research Labs, for the purpose of efficiently managing index pages for large random access files. The basic assumption was that indexes would be so voluminous that only small chunks of the tree could fit in main memory.

A B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k - 1$ keys.
5. All leaves appear in the same level and carry no information.

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

Internal nodes

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of U children and a **minimum** of L children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between $L-1$ and $U-1$). U must be either $2L$ or $2L-1$; therefore each internal node is at least half full. The relationship between U and L implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there's room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

The root node

The root node's number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree with no children at all.

Leaf nodes

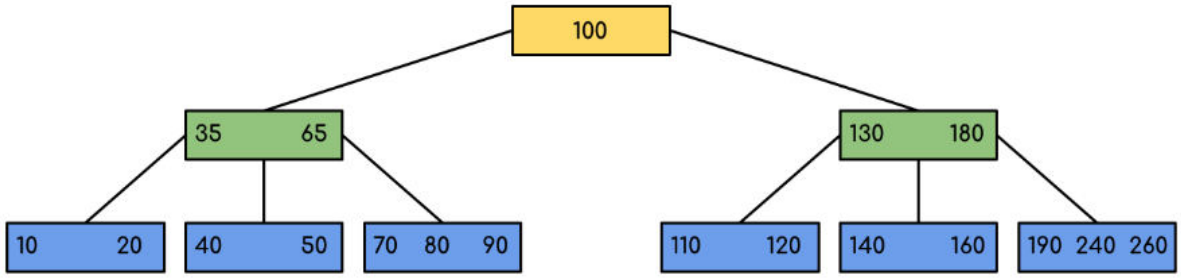
In Knuth's terminology, leaf nodes do not carry any information. The internal nodes that are one level above the leaves are what would be called "leaves" by other authors: these nodes only store keys (at most m-1, and at least m/2-1 if they are not the root) and pointers to nodes carrying no information.

A B-tree of depth n+1 can hold about U times as many items as a B-tree of depth n, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

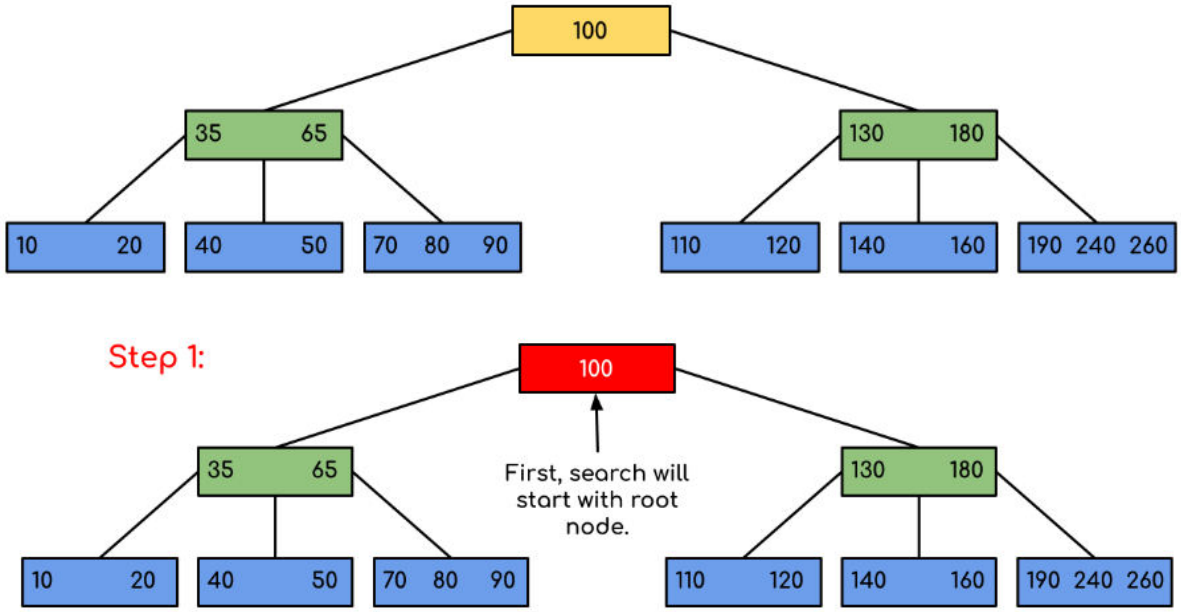
Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree except leaf nodes.

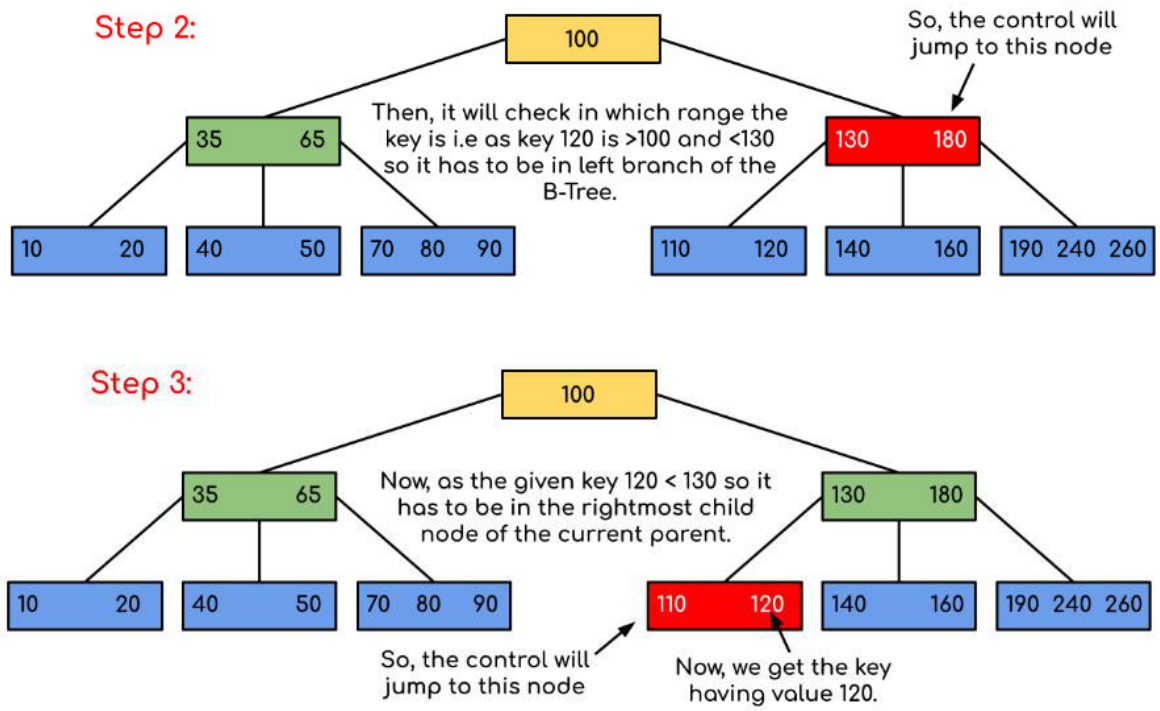
Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).



Searching 120 in the given B-Tree





Insertion

1. Initialize x as root.

2) While x is not leaf, do following

..**a)** Find the child of x that is going to be traversed next. Let the child be y.

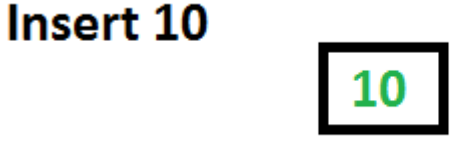
..**b)** If y is not full, change x to point to y.

..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

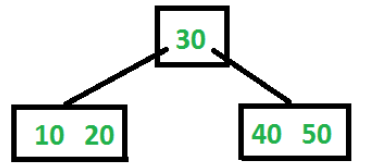


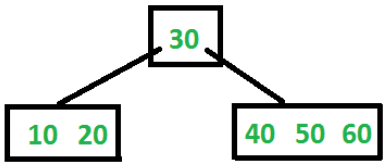
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

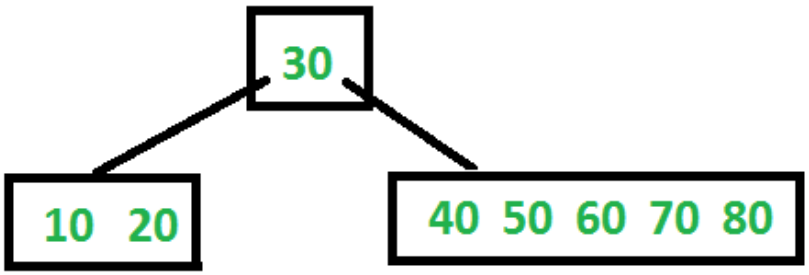
Insert 60





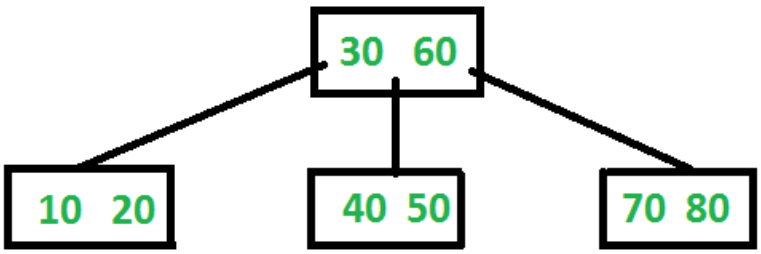
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



Delete Operation in B-Tree

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

1.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

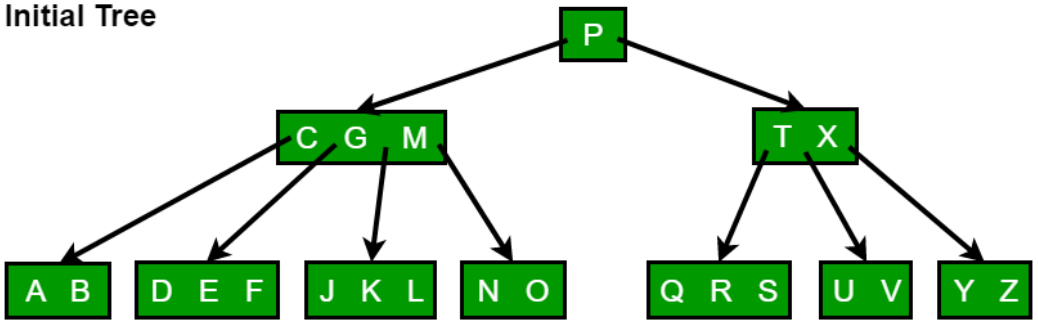
a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

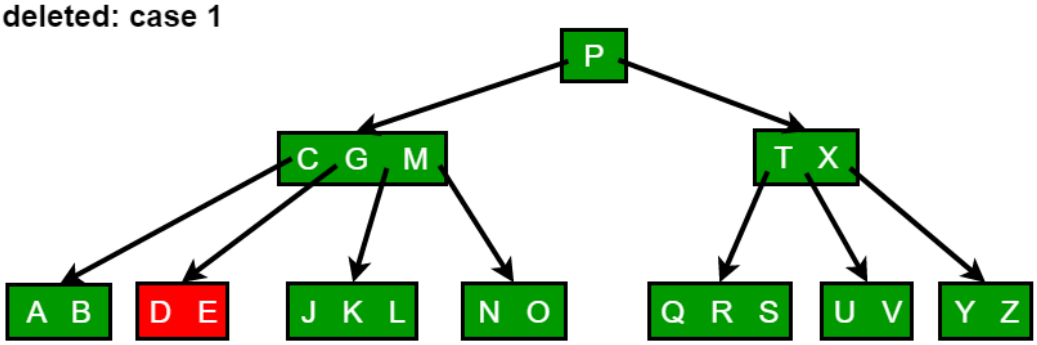
c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor

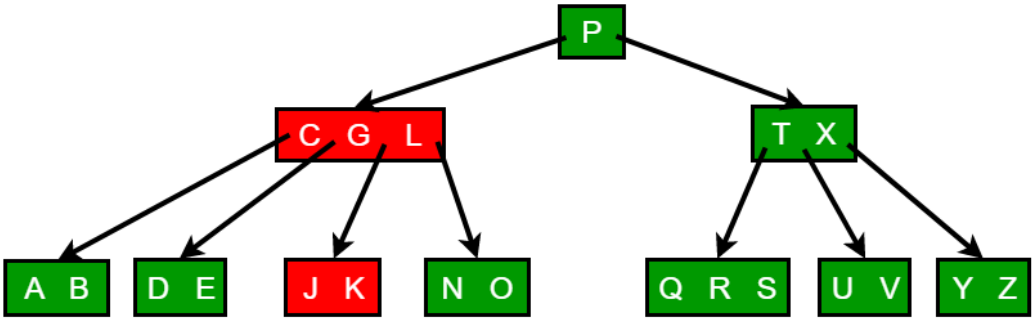
(a) Initial Tree



(b) F deleted: case 1



(c) M deleted: case 2a



(d) G deleted: case 2c