UNIT-2

Overview

Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.
The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrixchain
multiplication. We are given a sequence (chain) _$A1$, $A2$, ..., An_ of n matrices to be multiplied, and we wish to compute the product
(15.10)
We can evaluate the expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is _$A1$, $A2$, $A3$, $A4$_, the product $A1$ $A2$ $A3$ $A4$ can be fully parenthesized in five distinct ways:
($A1$ ($A2$ ($A3$ $A4$))) ,
($A1$ (($A2$ $A3$) $A4$)) ,
(($A1$ $A2$) ($A3$ $A4$)) ,
(($A1$ ($A2$ $A3$)) $A4$) ,
((($A1$ $A2$) $A3$) $A4$).
The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

```
MATRIX-MULTIPLY(A, B)
1 if columns[A] ≠ rows[B]
2 then error "incompatible dimensions"
3 else for i ← 1 to rows[A]
4 do for j ← 1 to columns[B]
5 do C[i, j] ← 0
```

```
6 for $k \leftarrow 1$ to $columns[A]$
7 do $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
8 return $C$
```

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 7, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain _A1, A2, A3_ of three matrices. Suppose that the dimensions of the matrices are 10 × 100, 100 × 5, and 5 × 50, respectively. If we multiply according to the parenthesization ((A1 A2) A3), we perform 10 · 100 · 5 = 5000 scalar multiplications to compute the 10 × 5 matrix product A1 A2, plus another 10 · 5 · 50 = 2500 scalar multiplications to multiply this matrix by A3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization (A1 (A2 A3)), we perform 100 · 5 · 50 = 25,000 scalar multiplications to compute the 100 × 50 matrix product A2 A3, plus another 10 · 100 · 50 = 50,000 scalar multiplications to multiply A1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

The ***matrix-chain multiplication problem*** can be stated as follows: given a chain _A1, A2, ...,An_ of n matrices, where for i = 1, 2, ..., n, matrix Ai has dimension pi-1 × pi, fully parenthesize the product A1 A2 An in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield anefficient algorithm. Denote the number of alternative parenthesizations of a sequence of $n$matrices by $P(n)$. When n = 1, there is just one matrix and therefore only one way to fullyparenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and (k + 1)st matrices for any k = 1, 2, ..., n - 1. Thus, we obtain the recurrence

Step 1: The structure of an optimal parenthesization

Our first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. For the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation Ai_j, where $i \leq j$, for the matrix that results from evaluating the product Ai Ai+1 Aj. Observe that if the problem is nontrivial, i.e., $i < j$, then any parenthesization of the product Ai Ai+1 Aj must split the product between Ak and Ak+1 forsome integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices Ai_k and Ak+1_j and then multiply them together to

produce the final product $A_{i..j}$. The cost of this parenthesization is thus the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together. The optimal substructure of this problem is as follows. Suppose that an optimal parenthesization of $A_i A_{i+1} A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the "prefix" subchain $A_i A_{i+1} A_k$ within this optimal parenthesization of $A_i A_{i+1} A_j$ must be an optimal parenthesization of $A_i A_{i+1} A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} A_k$, substituting that parenthesization in the optimal parenthesization of $A_i A_{i+1} A_j$ would produce another parenthesization of $A_i A_{i+1} A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for the parenthesization of the subchain $A_{k+1} A_{k+2} A_j$ in the optimal parenthesization of $A_i A_{i+1} A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} A_j$.
Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances.
Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} A_k$ and $A_{k+1} A_{k+2} A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places so that we are sure of having examined the optimal one.
Step 2: A recursive solution
Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our sub problems the problems of determining the minimum cost of a parenthesization of $A_i A_{i+1} A_j$ for $1 \le i \le j \le n$.
Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the cost of a cheapest way to compute $A_{1..n}$ would thus be $m[1, n]$.
We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, ..., n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that the optimal parenthesization splits the product $A_i A_{i+1} A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost ofmultiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.
Step 3: Computing the optimal costs
At this point, it is a simple matter to write a recursive algorithm based on recurrence to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 A_n$. As we see, however, this algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.
MATRIX-CHAIN-ORDER(p)

```
1  $n \leftarrow length[p] - 1$
2  for $i \leftarrow 1$ to $n$
3      do $m[i, i] \leftarrow 0$
4  for $l \leftarrow 2$ to $n$ ▹ $l$ is the chain length.
5      do for $i \leftarrow 1$ to $n - l + 1$
6             do $j \leftarrow i + l - 1$
7                $m[i, j] \leftarrow \infty$
8                for $k \leftarrow i$ to $j - 1$
9                    do $q \leftarrow m[i, k] + m[k + 1, j] + pi-1\ pkpj$
10                      if $q < m[i, j]$
11                         then $m[i, j] \leftarrow q$
12                              $s[i, j] \leftarrow k$
13 return $m$ and $s$
```

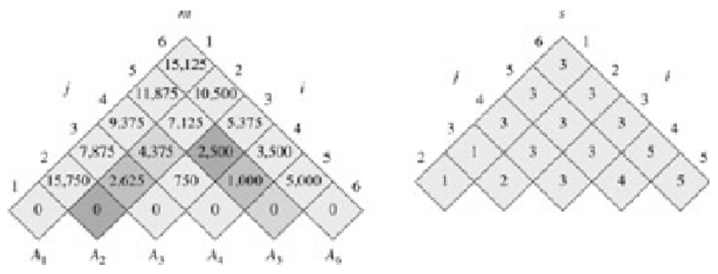


Figure : The *m* and *s* tables computed by MATRIX-CHAIN-ORDER for *n* = 6 and the following matrix dimensions:

matrix	**dimension**
*A*1	30 × 35
*A*2	35 × 15
*A*3	15 × 5
*A*4	5 × 10
*A*5	10 × 20
*A*6	20 × 25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the *m* table, and only the upper triangle is used in the *s* table. The minimum number of scalar multiplications to multiply the 6 matrices is *m*[1, 6] = 15,125. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of *O*(*n*3) for the algorithm. The loops are nested three deep, and each loop index (*l*, *i*, and *k*) takes on at most *n* -1 values. Example asks you to show that the running time of this algorithm is in fact also Ω(*n*3). The algorithm requires Θ(*n*2) space to store the *m* and *s* tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. It is not difficult to construct an optimal solution from the computed information stored in the table *s*[1 _ *n*, 1 _ *n*]. Each entry *s*[*i*, *j*] records the value of *k* such that the optimal parenthesization of *Ai Ai*+1 ··· *Aj* splits the product between *Ak* and *Ak*+1. Thus,we know that the final matrix multiplication in computing *A*1_*n* optimally is *A*1_*s*[1,*n*] *As*[1,*n*]+1_*n*.
The earlier matrix multiplications can be computed recursively, since *s*[1, *s*[1, *n*]] determines
the last matrix multiplication in computing *A*1_*s*[1,*n*], and *s*[*s*[1, *n*] + 1, *n*] determines the last
matrix multiplication in computing *As*[1,*n*]+1_*n*. The following recursive procedure prints an optimal parenthesization of _*Ai*, *Ai*+1, ..., *Aj*_, given the *s* table computed by MATRIXCHAIN-
ORDER and the indices *i* and *j*. The initial call PRINT-OPTIMAL-PARENS(*s*, 1, *n*)
prints an optimal parenthesization of _*A*1, *A*2, ..., *An*_.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1 if i = j
2 then print "A"i
3 else print "("
4 PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5 PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6 print ")"
```

In the example of Figure, the call PRINT-OPTIMAL-PARENS(*s*, 1, 6) prints the
parenthesization ((*A*1 (*A*2 *A*3)) ((*A*4 *A*5)*A*6)).

Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine the two key ingredients that an optimization problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems. We also look at a variant method, called memoization,[1] for taking advantage of the overlapping subproblems property.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.
Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming might apply. (It also might mean that a greedy strategy applies, however.In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.
You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing a preceding assembly-line station or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.

To characterize the space of subproblems, a good rule of thumb is to try to keep the space as simple as possible, and then to expand it as necessary. For example, the space of subproblems that we considered for assembly-line scheduling was the fastest way from entry into the factory through stations *S*1, *j* and *S*2, *j*. This subproblem space worked well, and there was no need to try a more general space of subproblems.

Longest Common Subsequence

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence S = <s_1, s_2, s_3, s_4, …,s_n>.

A sequence Z = <z_1, z_2, z_3, z_4, …,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, *X* and *Y* are two sequences over a finite set of elements. We can say that *Z* is a common subsequence of *X* and *Y*, if *Z* is a subsequence of both *X* and *Y*.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

Naïve Method

Let *X* be a sequence of length *m* and *Y* a sequence of length *n*. Check for every subsequence of *X* whether it is a subsequence of *Y*, and return the longest common subsequence found.

There are 2^m subsequences of *X*. Testing sequences whether or not it is a subsequence of *Y* takes *O(n)* time. Thus, the naïve algorithm would take $O(n2^m)$ time.

Dynamic Programming

Let $X = < x_1, x_2, x_3, \dots, x_m >$ and $Y = < y_1, y_2, y_3, \dots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table *C[m, n]* is computed in row major order and another table *B[m,n]* is computed to construct optimal solution.

```
Algorithm: LCS-Length-Table-Formulation (X, Y)
```

```
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if $x_i$ = $y_j$
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := ‘D’
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := ‘U’
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := ‘L’
return C and B
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and j = 0
  return
if B[i, j] = ‘D’
  Print-LCS(B, X, i-1, j-1)
  Print($x_i$)
else if B[i, j] = ‘U’
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of X and Y.

Analysis

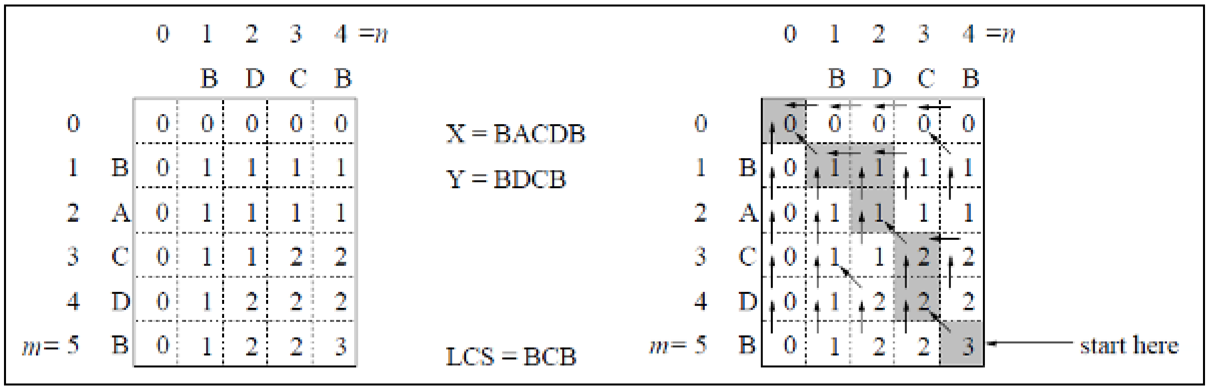
To populate the table, the outer for loop iterates *m* times and the inner for loop iterates *n* times. Hence, the complexity of the algorithm is *O(m, n)*, where *m* and *n* are the length of two strings.

Example

In this example, we have two strings *X = BACDB* and *Y = BDCB* to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).

In table B, instead of ‘D’, ‘L’ and ‘U’, we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m = 5	B	0	1	2	2	3

A

Greedy Algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

For example, a greedy strategy for the travelling salesman problem (which is of a high computational complexity) is the following heuristic: "At each step of the journey, visit the nearest unvisited city." This heuristic does not intend to find a best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps. In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids, and give constant-factor approximations to optimization problems with submodular structure.

In general, greedy algorithms have five components:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution, and
5. A solution function, which will indicate when we have discovered a complete solution

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work will have two properties:

Greedy choice property

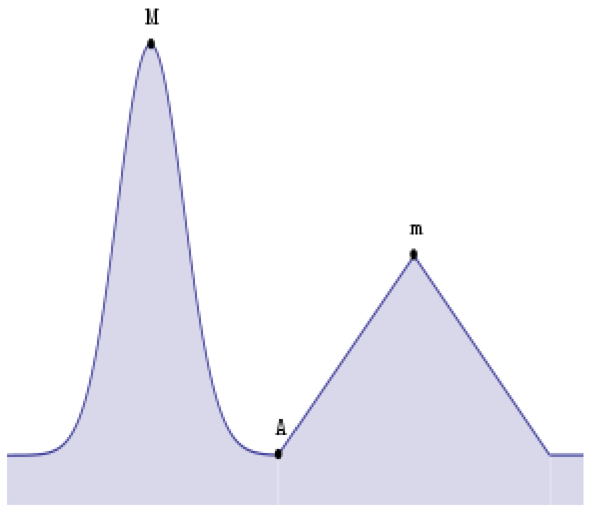
We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Optimal substructure

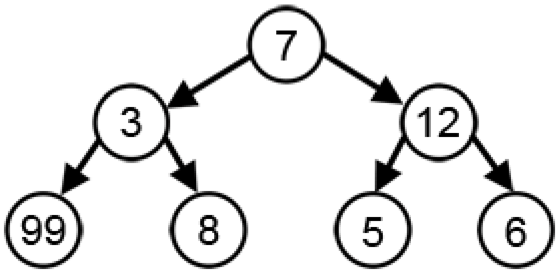
"A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems."[2]

Cases of failure

Examples on how a greedy algorithm may fail to achieve the optimal solution.



Starting from A, a greedy algorithm that tries to find the maximum by following the greatest slope will find the local maximum at "m", oblivious to the global maximum at "M".



With a goal of reaching the largest sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the *unique worst possible* solution. One example is the traveling salesman problem mentioned above: for each number of cities, there is an assignment of distances between the cities for which the nearest-neighbor heuristic produces the unique worst possible tour.

Elements of the Greedy Strategy

Optimal Substructure:

An optimal solution to the problem contains within it optimal solutions to sub-problems. A' = A - {1} (greedy choice) A' can be solved again with the greedy algorithm. $S' = \{ i \quad S, s_i \quad f_i \}$

When do you use DP versus a greedy approach? Which should be faster?

The 0 - 1 knapsack problem:

A thief has a knapsack that holds at most W pounds. Item i : (v_i, w_i) (v = value, w = weight) thief must choose items to maximize the value stolen and still fit into the knapsack. Each item must be taken or left (0 - 1).

Fractional knapsack problem:

takes parts, as well as wholes

Both the 0 - 1 and fractional problems have the optimal substructure property: Fractional: v_i / w_i is the value per pound. Clearly you take as much of the item with the greatest value per pound. This continues until you fill the knapsack. Optimal (Greedy) algorithm takes O (n lg n), as we must sort on $v_i / w_i = d_i$.

Consider the same strategy for the 0 - 1 problem:

W = 50 lbs. (maximum knapsack capacity)

$w_1 = 10$	$v_1 = 60$	$d_1 .= 6$
$w_2 = 20$	$v_2 = 100$	$d_2 .= 5$
$w_3 = 30$	$v_3 = 120$	$d_3 = 4$

were d is the value density

Greedy approach: Take all of 1, and all of 2: $v_1 + v_2 = 160$, optimal solution is to take all of 2 and 3: $v_2 + v_3 = 220$, other solution is to take all of 1 and 3 $v_1 + v_3 = 180$. All below 50 lbs.

When solving the 0 - 1 knapsack problem, empty space lowers the effective d of the load. Thus each time an item is chosen for inclusion we must consider both

- i included
- i excluded

These are clearly overlapping sub-problems for different i's and so best solved by DP!

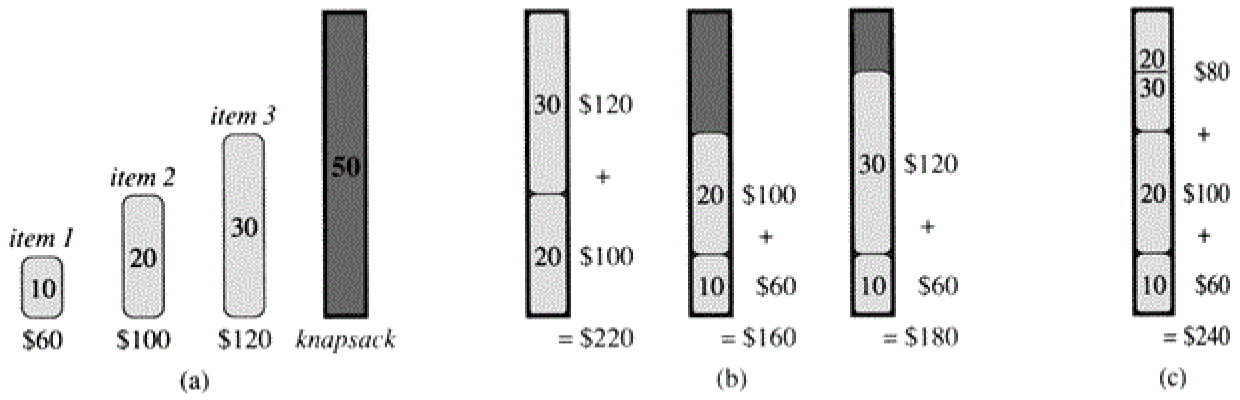


Figure The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Activity Selection problem

Let us consider the Activity Selection problem as our first example of Greedy algorithms. Following is the problem statement.
You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.
1. Sort the activities according to their finishing time.
2. Select the first activity from the sorted array and print it.
3. Do following for remaining activities in the sorted array.
…….a) If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.
In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

Example 1 : Consider the following 3 activities sorted by
by finish time.
    start[] = {10, 12, 20};
    finish[] = {20, 25, 30};
A person can perform at most two activities. The
maximum set of activities that can be executed
is {0, 2} [These are indexes in start[] and

finish[]]

Example 2 : Consider the following 6 activities sorted by by finish time.
    start[] = {1, 3, 0, 5, 8, 5};
    finish[] = {2, 4, 6, 7, 9, 9};
A person can perform at most four activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]

GREEDY-ACTIVITY-SELECTOR(s , f)

```
1. n=length[s]
2. A={1}
3. j=1
4. for i= 2 to n
5. do if $s_i>=f_j$
6. then  A=A U{i}
7. j=i
8. return A
```

Huffman code

Huffman Coding Algorithm With Example

Huffman coding algorithm was invented by David Huffman in 1952. It is an algorithm which works with integer length codes. A Huffman tree represents Huffman codes for the character that might appear in a text file. Unlike to ASCII or Unicode, Huffman code uses different number of bits to encode letters. If the number of occurrence of any character is more, we use fewer numbers of bits. Huffman coding is a method for the construction of minimum redundancy codes.

Huffman tree can be achieved by using compression technique. Data compression have lot of advantages such as it minimizes cost, time, bandwidth, storage space for transmitting data from one place to another.In regular text file each character would take up 1 byte (8 bits) i.e. there are 16 characters (including white spaces and punctuations) which normally take up 16 bytes. In the ASCII code there are 256 characters and this leads to the use of 8 bits to represent each character but in any test file we do not have use all 256 characters. For example, in any English language text, generally the character 'e' appears more than the character 'z'. To achieve compression, we can often use a shorter bit string to represent more frequently occurring characters. We do not have to represent all 256 characters, unless they all appear in the document. The data encoding schemes broadly categorized
in two categories.

Fixed Length Encoding Scheme

Fixed length encoding scheme compresses our data by packing it into the minimum number of bits i.e. needed to represent all possible values of our data. The fixed length code can store maximum 224,000 bits data.

Variable Length Encoding Scheme

In variable length encoding scheme we map source symbol to variable number of bits. It allows source to be compressed and decompressed with zero error.

Construction of Huffman Code

A greedy algorithm constructs an optimal prefix code called Huffman code. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves (C is the number of
characters) and perform |C| – 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the number of set of characters, z denotes the parent node and x & y are the left & right child of z respectively.

Huffman (C)

```
1. n=|C|
2. Q=C
3. for i=1 to n-1
4. do
5. z=allocate_Node()
6. x=left[z]=Extract_Min(Q)
7. y=right[z]=Extract_Min(Q)
8. f[z]=f[x]+f[y]
9. Insert(Q,z)
10. return Extract_Min(Q)
```

Analysis

The Q is initialized as a priority queue with the character C.
Q=C can be performed by using Build_Heap in O(n) time.
For loop takes (|n|-1) times because each heap operation requires O(log n) time.
Hence the total running time of Huffman code on the set of n characters is O(n log n).

Method

The following general procedure is applied for construction a Huffman tree:
Search for the two nodes having the lowest frequency, which are not yet assigned to a parent node.
Couple these nodes together to a new interior node.
Add both the frequencies and assign this value to the new interior node.
The procedure has to be repeated until all nodes are combined together in a root node.

Huffman Coding Algorithm Example

Construct a Huffman tree by using these nodes.

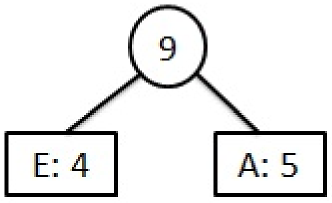
Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Solution:

Step 1: According to the Huffman coding we arrange all the elements (values) in ascending order of the frequencies.

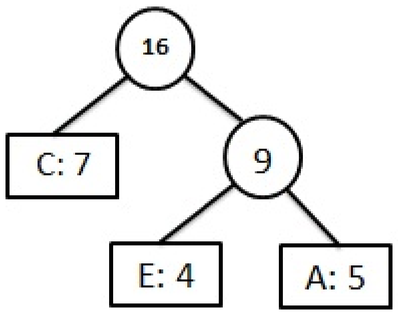
Value	E	A	C	F	D	B
Frequency	4	5	7	12	15	25

Step 2: Insert first two elements which have smaller frequency.



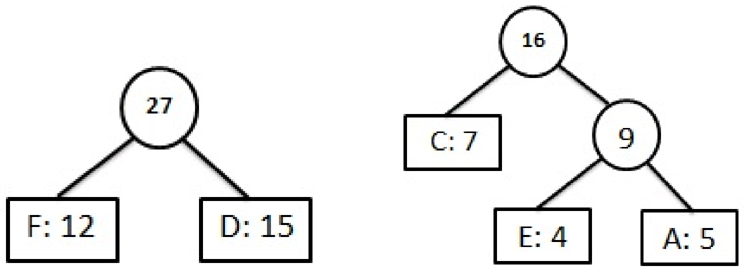
Value	C	EA	F	D	B
Frequency	7	9	12	15	25

Step 3: Taking next smaller number and insert it at correct place.



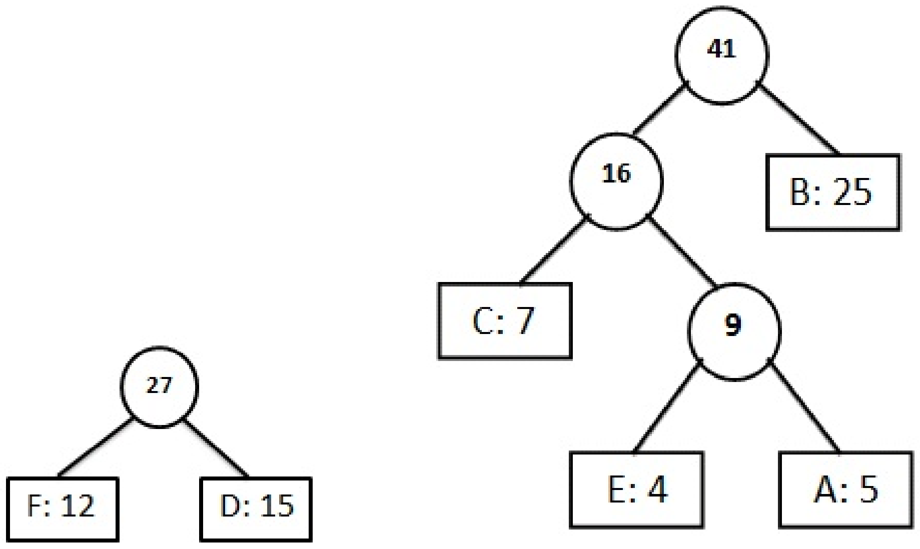
Value	F	D	CEA	B
Frequency	12	15	16	25

Step 4: Next elements are F and D so we construct another subtree for F and D.



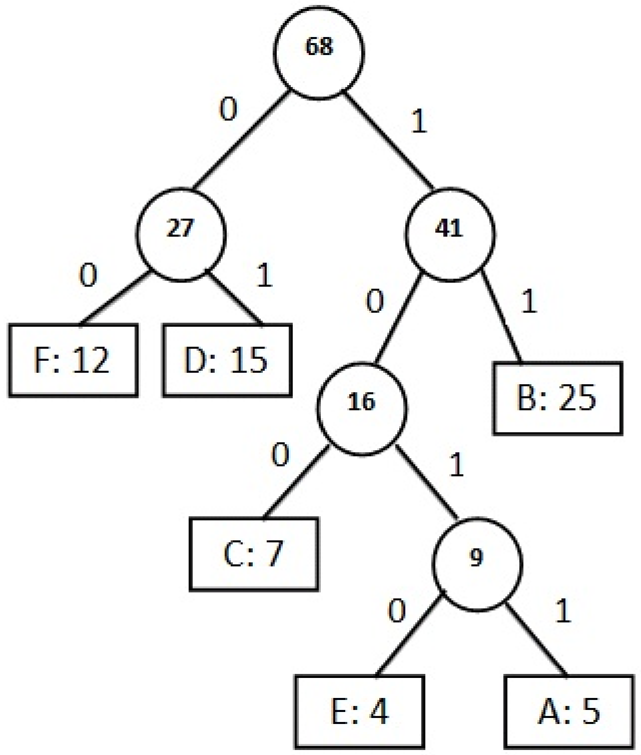
Value	CEA	B	FD
Frequency	16	25	27

Step 5: Taking next value having smaller frequency then add it with CEA and insert it at correct place.



Value	FD	CEAB
Frequency	27	41

Step 6: We have only two values hence we can combined by adding them.



Huffman Tree

Value	FDCEAB
Frequency	68

Now the list contains only one element i.e. FDCEAB having frequency 68 and this element (value) becomes the root of the Huffman tree.

Activity or Task Scheduling Problem

This is the dispute of optimally scheduling unit-time tasks on a single processor, where each job has a deadline and a penalty that necessary be paid if the deadline is missed.

A unit-time task is a job, such as a program to be rush on a computer that needed precisely one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which to perform these tasks. The first task in the schedule starts at time 0 and ends at time 1; the second task begins at time 1 and finishes at time 2, and so on.

The dispute of scheduling unit-time tasks with deadlines and penalties for each processor has the following inputs:

- a set S = {1, 2, 3.....n} of n unit-time tasks.
- a set of n integer deadlines d_1 d_2 d_3...d_n such that d_i satisfies 1≤ d_i ≤ n and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties w_1 w_2....w_n such that we incur a penalty of w_i if task i is not finished by time d_i, and we incurred no penalty if a

task finishes by its deadline.

Here we find a schedule for S that minimizes the total penalty incurred for missed deadlines.

A task is **late** in this schedule if it finished after its deadline. Otherwise, the task is early in the schedule. An arbitrary schedule can consistently be put into **early-first form**, in which the first tasks precede the late tasks, i.e., if some new task x follows some late task y, then we can switch the position of x and y without affecting x being early or y being late.

An arbitrary schedule can always be put into a **canonical form** in which first tasks precede the late tasks, and first tasks are scheduled in order of nondecreasing deadlines.

A set A of tasks is **independent** if there exists a schedule for the particular tasks such that no tasks are late. So the set of first tasks for a schedule forms an independent set of tasks 'I' denote the set of all independent set of tasks.

For any set of tasks A, A is independent if for t = 0, 1, 2.....n we have $N_t(A) \leq t$ where $N_t(A)$ denotes the number of tasks in A whose deadline is t or prior, i.e. if the tasks in A are expected in order of monotonically growing deadlines, then no task is late.

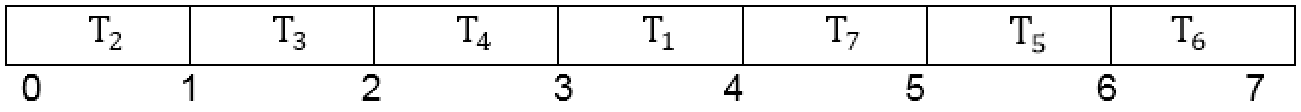
Example: Find the optimal schedule for the following task with given weight (penalties) and deadlines.

	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution: According to the Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that the maximum time for which uniprocessor machine will run in 6 units because it is the maximum deadline.

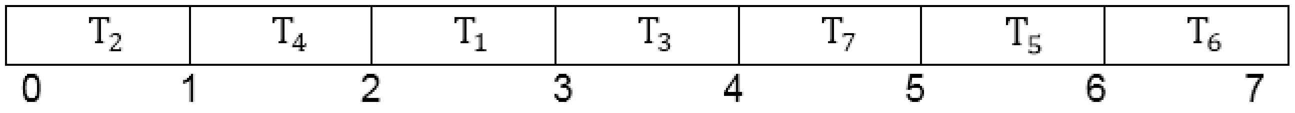
Let T_i represents the tasks where i = 1 to 7



T_5 and T_6 cannot be accepted after T_7 so penalty is

$w_5 + w_6 = 30 + 20 = 50$ (2 3 4 1 7 5 6)

Other schedule is



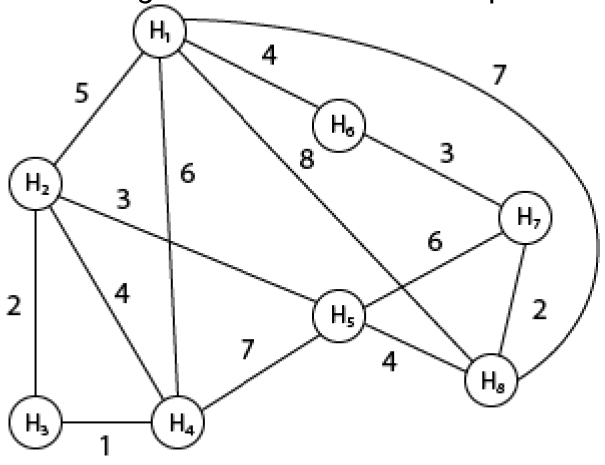
(2 4 1 3 7 5 6)

There can be many other schedules but (2 4 1 3 7 5 6) is optimal.

Travelling Sales Person Problem

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman

needs to minimize the total length of the trip.
Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.
Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.
The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:
$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

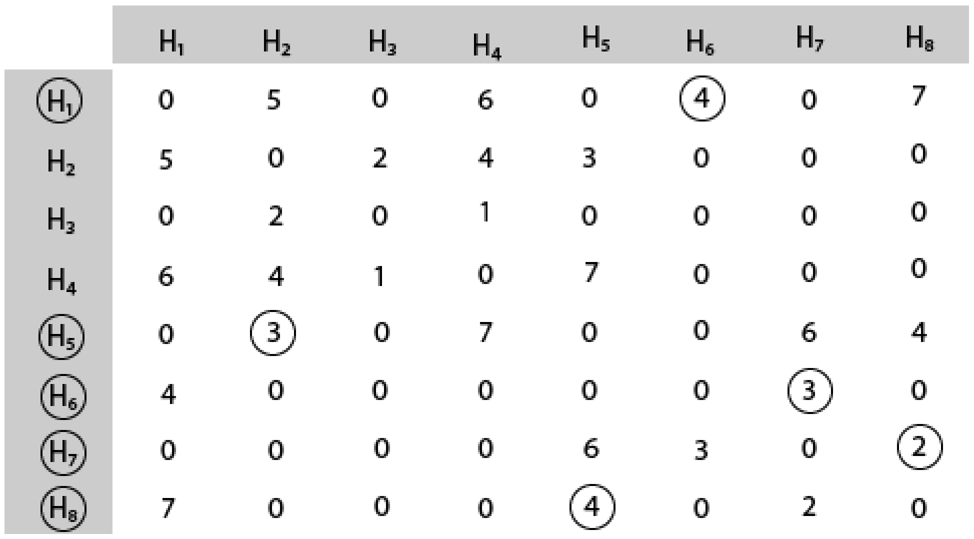
Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

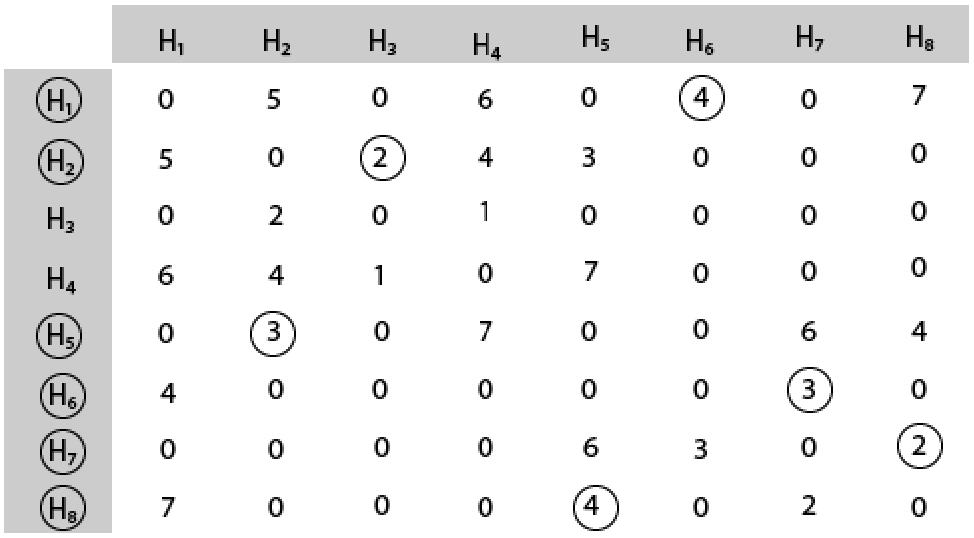
	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.



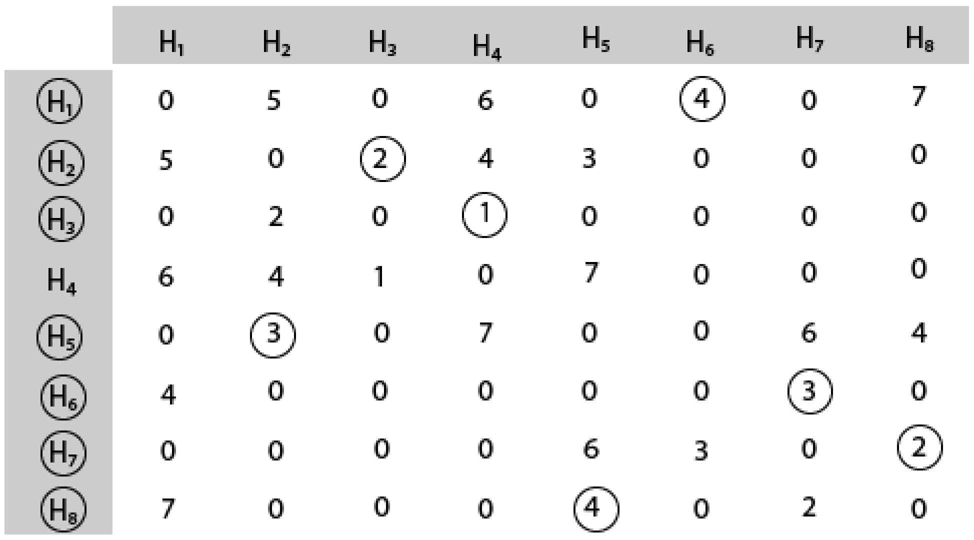
	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
[✓ H_1]	0	5	0	6	0	[✓ 4]	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
[✓ H_5]	0	[✓ 3]	0	7	0	0	6	4
[✓ H_6]	4	0	0	0	0	0	[✓ 3]	0
[✓ H_7]	0	0	0	0	6	3	0	[✓ 2]
[✓ H_8]	7	0	0	0	[✓ 4]	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
[✓ H_1]	0	5	0	6	0	[✓ 4]	0	7
[✓ H_2]	5	0	[✓ 2]	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
[✓ H_5]	0	[✓ 3]	0	7	0	0	6	4
[✓ H_6]	4	0	0	0	0	0	[✓ 3]	0
[✓ H_7]	0	0	0	0	6	3	0	[✓ 2]
[✓ H_8]	7	0	0	0	[✓ 4]	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	④	0	7
H_2	5	0	②	4	3	0	0	0
H_3	0	2	0	①	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	③	0	7	0	0	6	4
H_6	4	0	0	0	0	0	③	0
H_7	0	0	0	0	6	3	0	②
H_8	7	0	0	0	④	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

4 3 2 4 3 2 1 6

$H_1 \rightarrow H_6 \rightarrow H_7 \rightarrow H_8 \rightarrow H_5 \rightarrow H_2 \rightarrow H_3 \rightarrow H_4 \rightarrow H_1$.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them "**symmetric binary B-trees**."

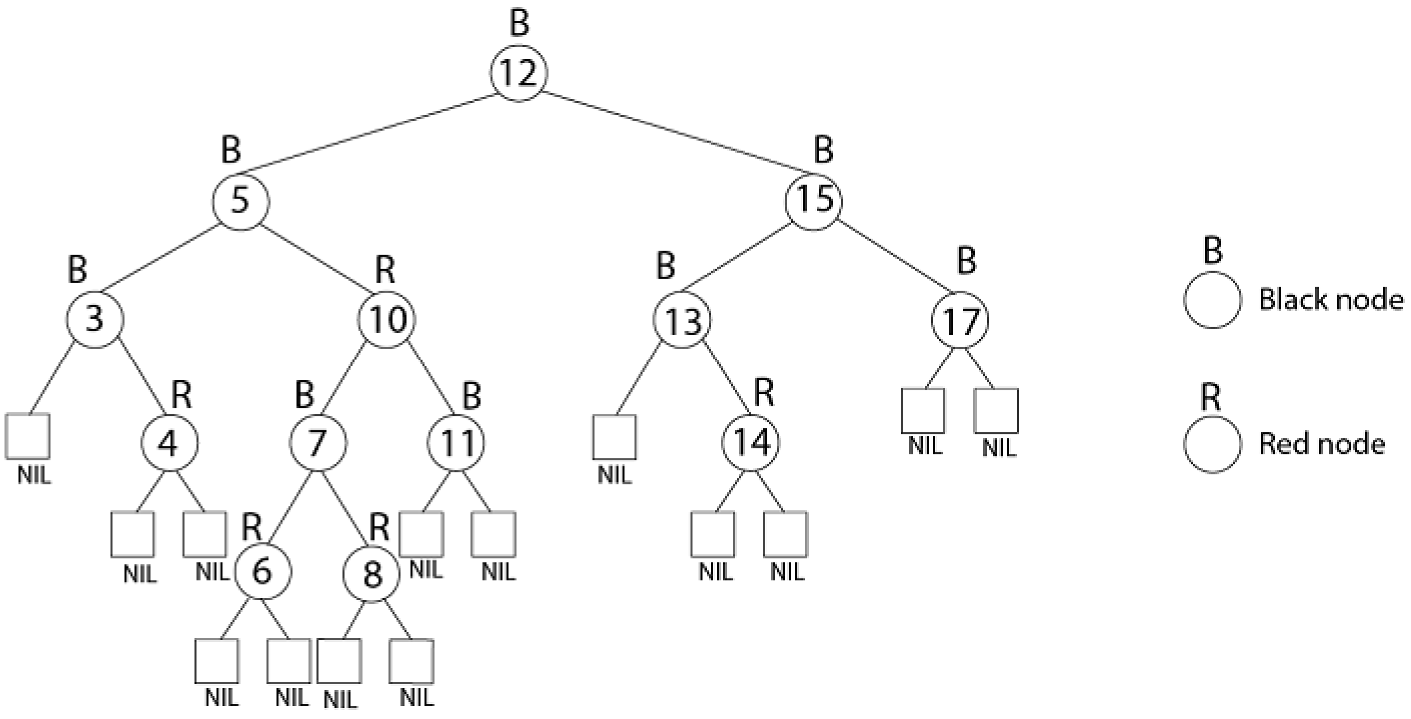
A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.
4. **Black Height Rule:** For particular node v, there exists an integer bh (v) such that specific downward path from v to a nil has correctly bh (v) black real (i.e. non-nil) nodes. Call this portion the black height of v. We determine the black height of an RB tree to be the black height of its root.

A tree T is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.

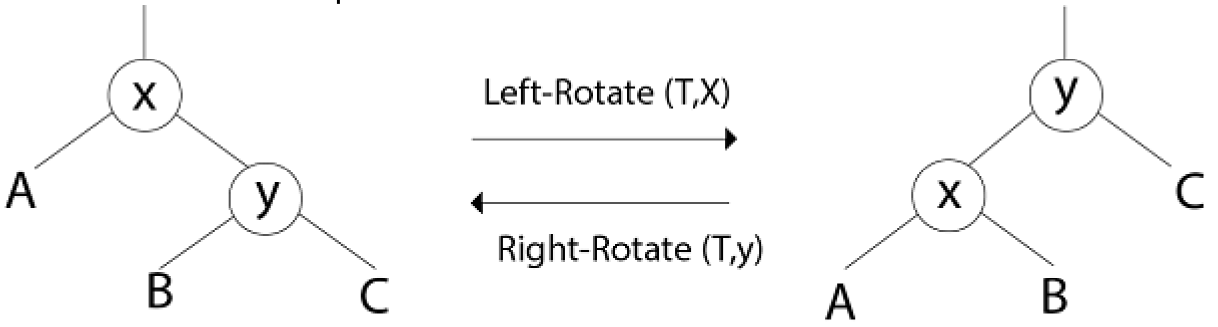


Operations on RB Trees:

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take O (log n) time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



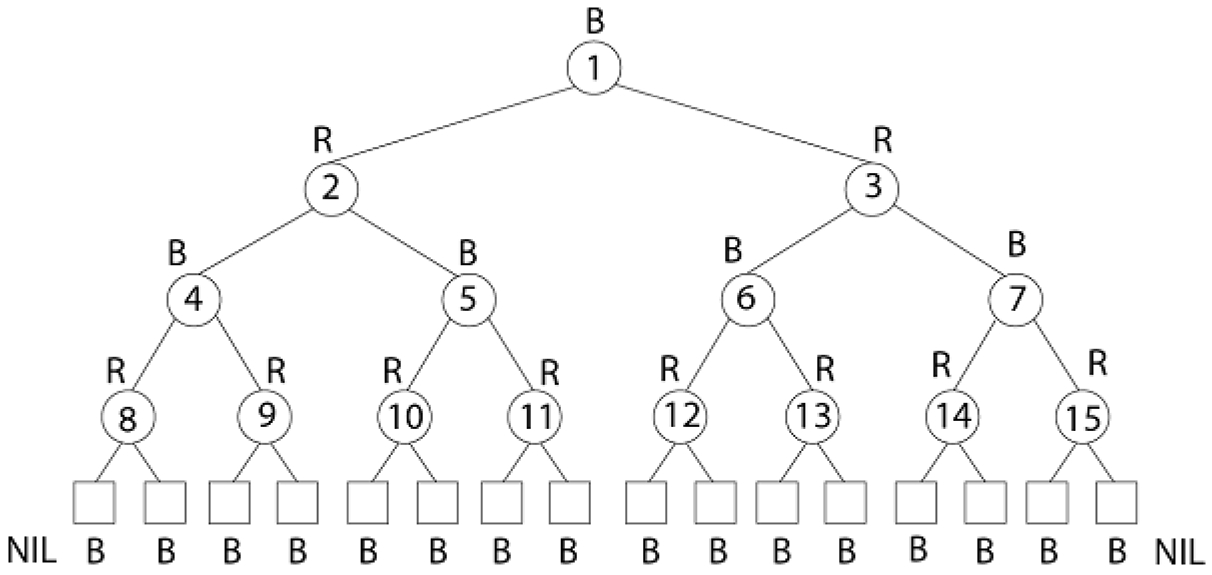
Clearly, the order (Ax By C) is preserved by the rotation operation. Therefore, if we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

```
LEFT ROTATE (T, x)
  1. y ← right [x]
  1. y ← right [x]
  2. right [x] ← left [y]
  3. p [left[y]] ← x
  4. p[y] ← p[x]
  5. If p[x] = nil [T]
```

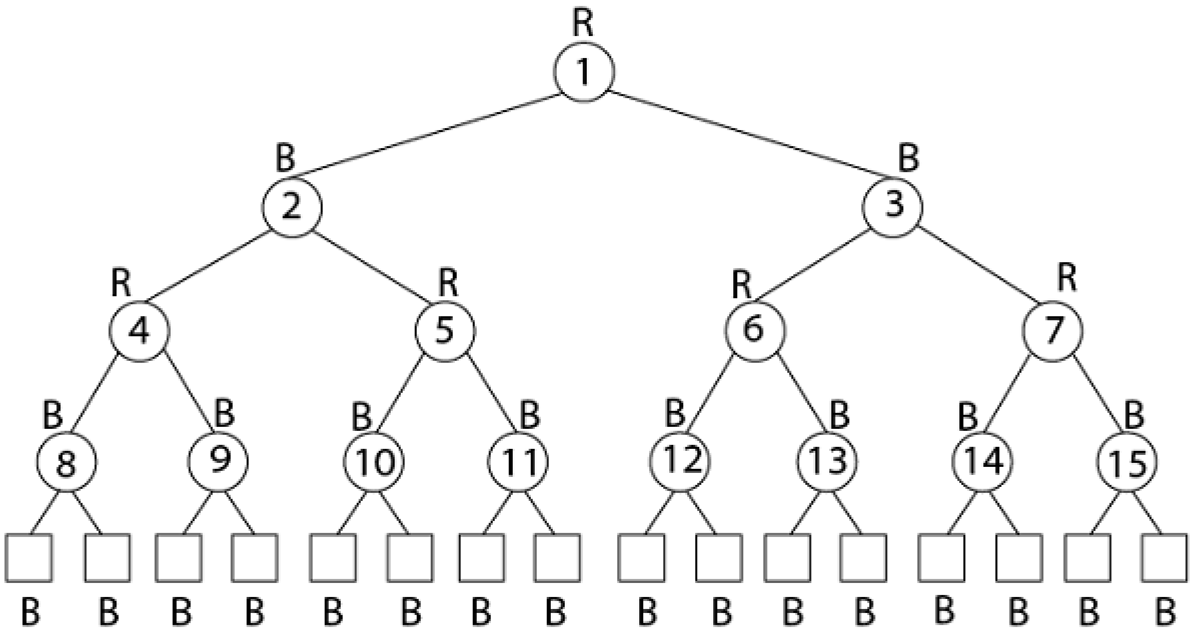
```
      then root [T] ← y
       else if x = left [p[x]]
         then left [p[x]] ← y
       else right [p[x]] ← y
   6. left [y] ← x.
   7. p [x] ← y.
```

Example: Draw the complete binary tree of height 3 on the keys {1, 2, 3... 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting trees are: 2, 3 and 4.

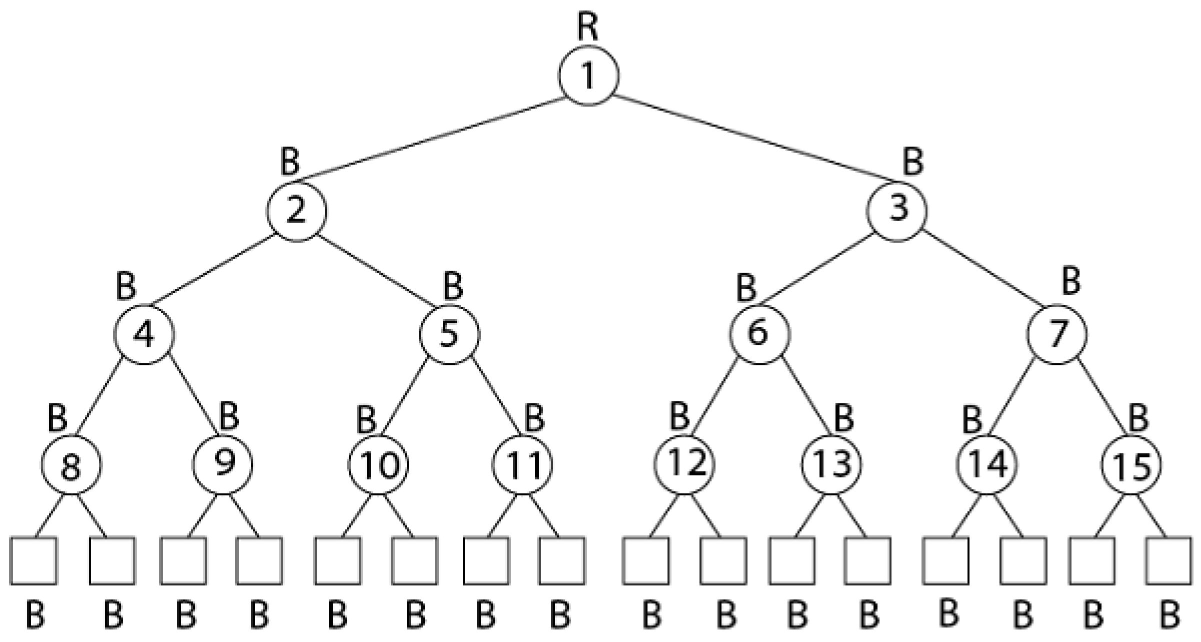
Solution:



Tree with black-height-2



Tree with black-height-3



Tree with black-height-4

2. Insertion:

- Insert the new node the way it is done in Binary Search Trees.
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

```
RB-INSERT (T, z)
 1. y ← nil [T]
 2. x ← root [T]
 3. while x ≠ NIL [T]
 4. do y ← x
 5. if key [z] < key [x]
 6. then x  ← left [x]
 7. else x ←  right [x]
 8. p [z] ← y
 9. if y = nil [T]
 10. then root [T] ← z
 11. else if key [z] < key [y]
 12. then left [y] ← z
 13. else right [y] ← z
 14. left [z] ← nil [T]
 15. right [z] ← nil [T]
```

```
  16. color [z] ← RED
  17. RB-INSERT-FIXUP (T, z)
```

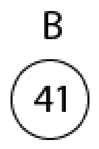
After the insert new node, Coloring this new node into black may violate the black-height conditions and coloring this new node into red may violate coloring conditions i.e. root is black and red node has no red children. We know the black-height violations are hard. So we color the node red. After this, if there is any color violation, then we have to correct them by an RB-INSERT-FIXUP procedure.

```
 RB-INSERT-FIXUP (T, z)
  1. while color [p[z]] = RED
  2. do if p [z] = left [p[p[z]]]
  3. then y ← right [p[p[z]]]
  4. If color [y] = RED
  5. then color [p[z]] ← BLACK    //Case 1
  6. color [y] ← BLACK           //Case 1
  7. color [p[z]] ← RED          //Case 1
  8. z  ← p[p[z]]               //Case 1
  9. else if z= right [p[z]]
  10. then z ← p [z]            //Case 2
  11. LEFT-ROTATE (T, z)         //Case 2
  12. color [p[z]] ← BLACK       //Case 3
  13. color [p [p[z]]] ← RED      //Case 3
  14. RIGHT-ROTATE  (T,p [p[z]])  //Case 3
  15. else (same as then clause)
      With "right" and "left" exchanged
  16. color [root[T]] ← BLACK
```

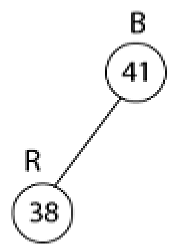
Example: Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

Solution:

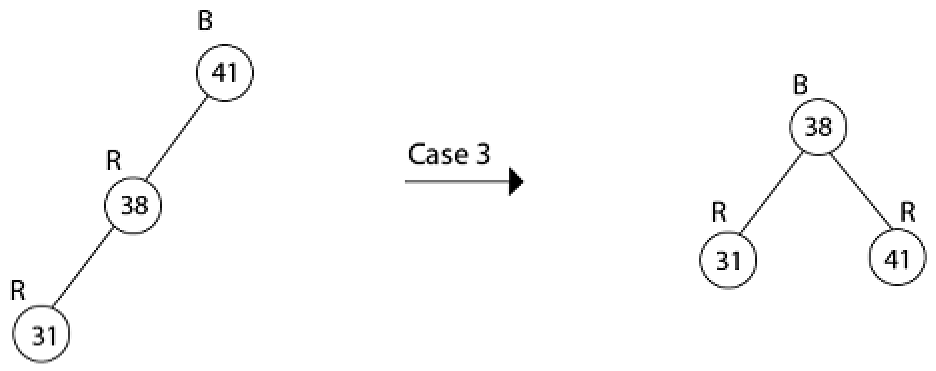
Insert 41



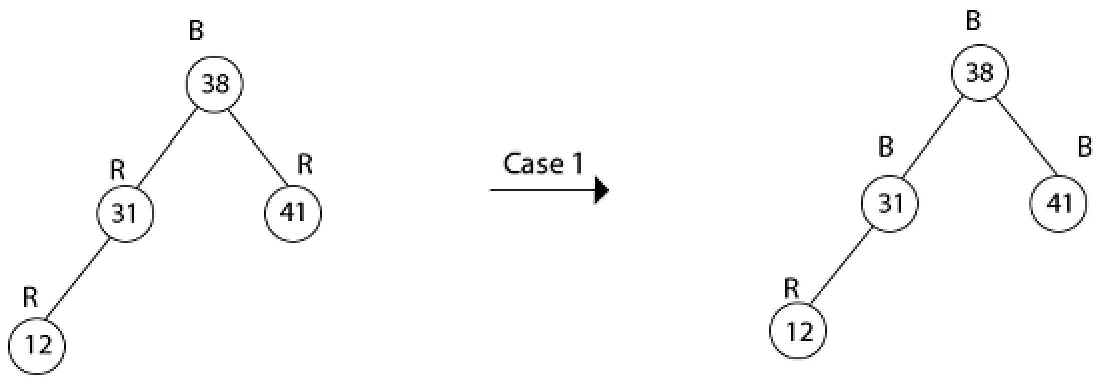
Insert 38



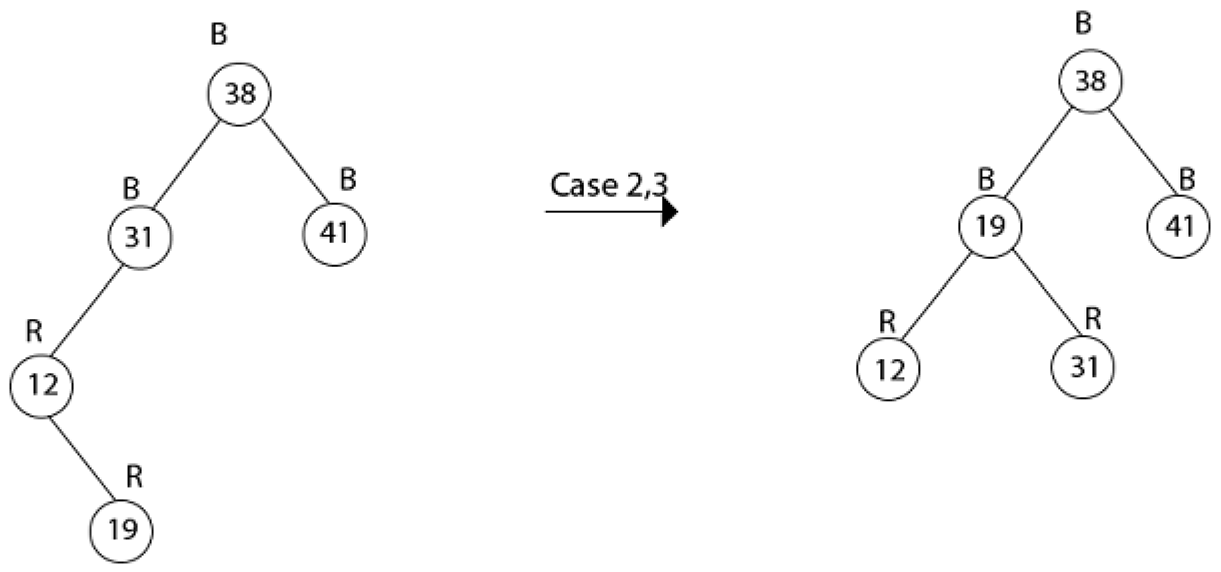
Insert 31

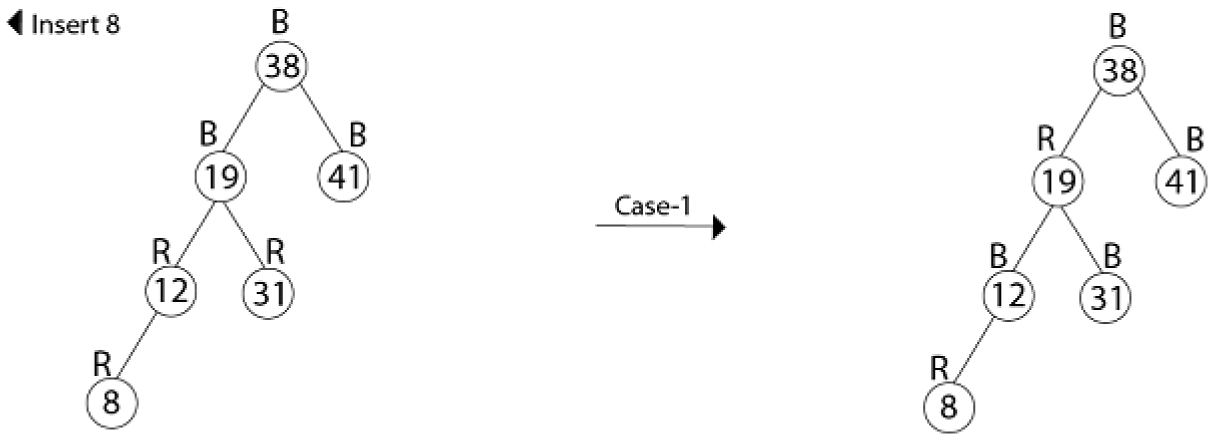


Insert 12

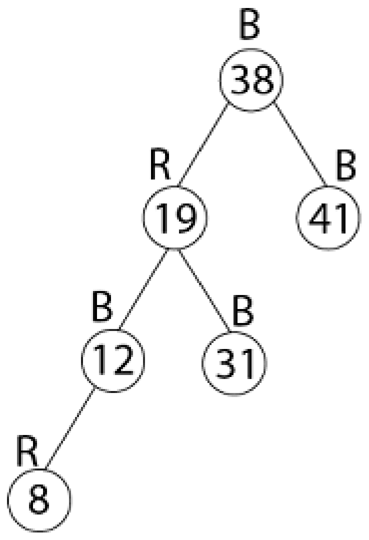


Insert 19





Thus the final tree is



---

3. Deletion:

First, search for an element to be deleted

- If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
- If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
- Two cases arise:

- The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
- The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

```
RB-DELETE (T, z)
 1. if left [z] = nil [T] or right [z] = nil [T]
 2. then y ← z
 3. else y ← TREE-SUCCESSOR (z)
 4. if left [y] ≠ nil [T]
 5. then x ← left [y]
 6. else x ← right [y]
 7. p [x] ←  p [y]
 8. if p[y] = nil [T]
 9. then root [T]  ← x
 10. else if y = left [p[y]]
 11. then left [p[y]] ← x
 12. else right [p[y]] ← x
 13. if y≠ z
 14. then key [z] ← key [y]
 15. copy y's satellite data into z
 16. if color [y] = BLACK
 17. then RB-delete-FIXUP (T, x)
 18. return y
```

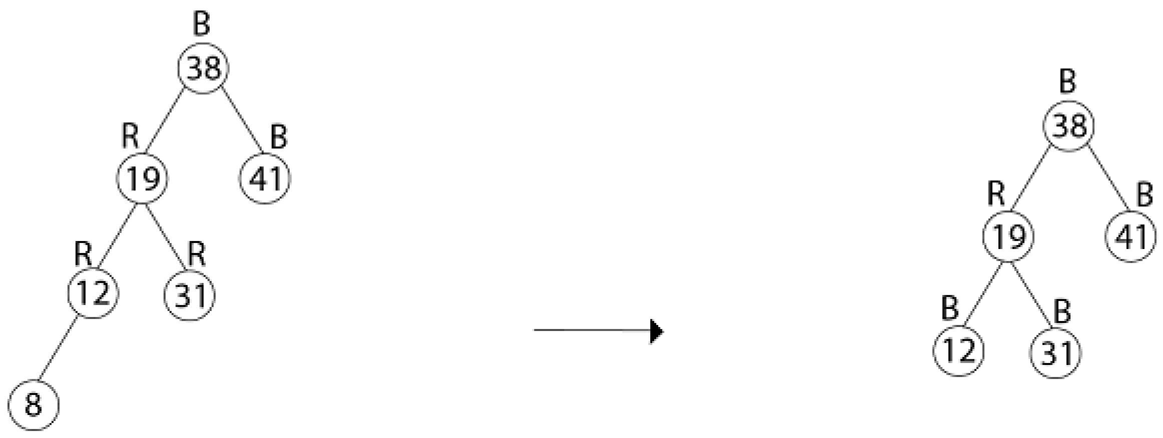
```
RB-DELETE-FIXUP (T, x)
 1. while x ≠ root [T] and color [x] = BLACK
 2. do if x = left [p[x]]
 3. then w ← right [p[x]]
 4. if color [w] = RED
 5. then color [w] ← BLACK       //Case 1
 6. color [p[x]] ← RED           //Case 1
 7. LEFT-ROTATE (T, p [x])       //Case 1
 8. w ← right [p[x]]             //Case 1
 9. If color [left [w]] = BLACK and color [right[w]] = BLACK
 10. then color [w] ← RED        //Case 2
 11. x ← p[x]                    //Case 2
 12. else if color [right [w]] = BLACK
 13. then color [left[w]] ← BLACK //Case 3
 14. color [w] ← RED             //Case 3
 15. RIGHT-ROTATE (T, w)         //Case 3
 16. w ← right [p[x]]            //Case 3
 17. color [w] ← color [p[x]]    //Case 4
 18. color p[x] ← BLACK          //Case 4
```

```
19. color [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. color [x] ← BLACK
```

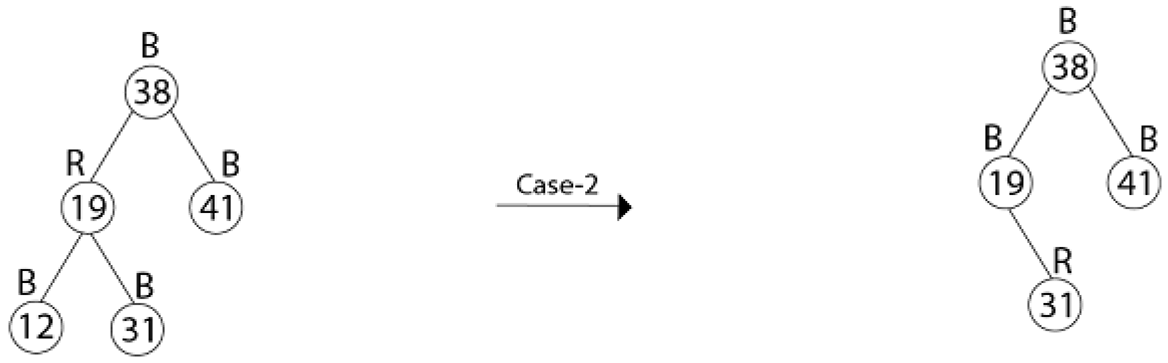
Example: In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

Solution:

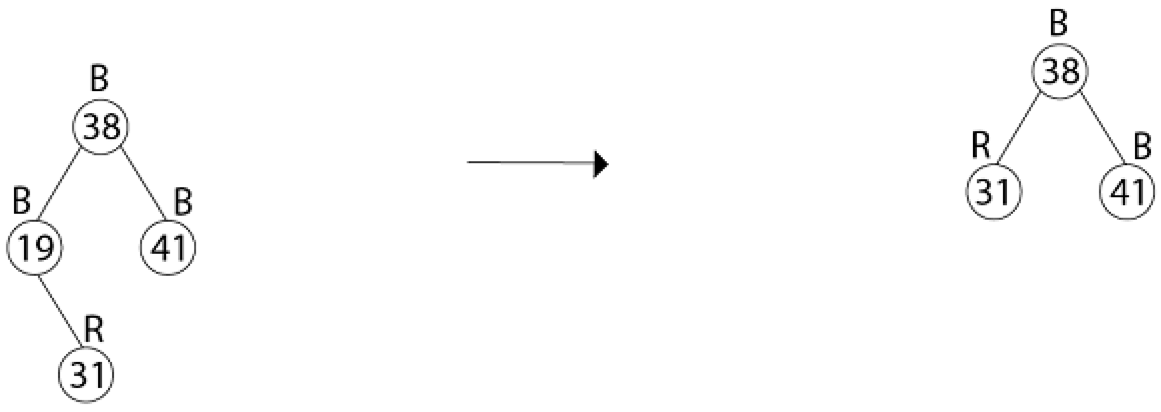
◀ Delete 8

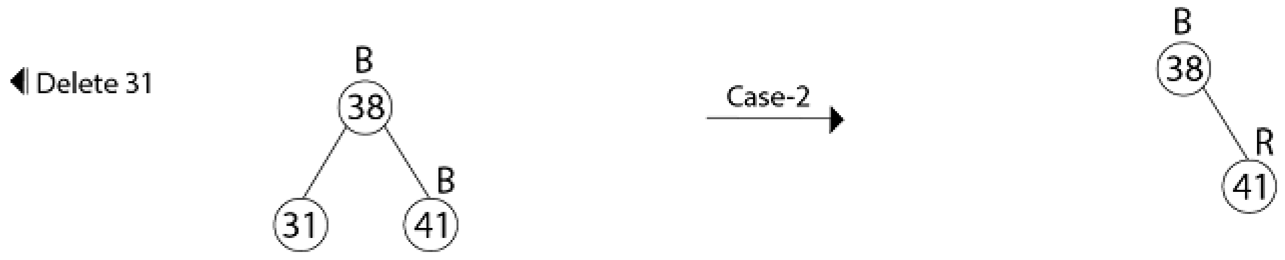


◀ Delete 12

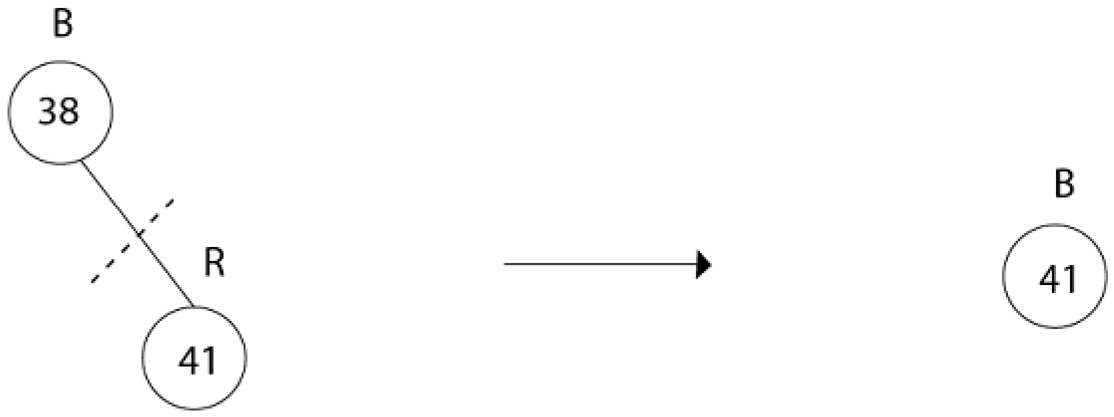


◀ Delete 19





Delete 38



Delete 41
No Tree.

Splay Tree

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Search-Operation

The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

1) **Node is root** We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: *Node is child of root* (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
            y                          x
           / \    Zig (Right Rotation)        /  \
          x   T3  – - – - – - - - ->        T1   y
         / \      < - - - - - - - - -            / \
        T1  T2    Zag (Left Rotation)          T2   T3
```

3) *Node has both parent and grandparent*. There can be following subcases.
……..3.a) Zig-Zig and Zag-Zag Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

```
Zig-Zig (Left Left Case):
     G                  P                    X
    / \               /   \                 / \
   P  T4   rightRotate(G)  X    G    rightRotate(P)  T1   P
  / \     ============> / \  / \   ============>      / \
 X  T3                T1 T2 T3 T4                  T2  G
/ \                                                   / \
T1 T2                                                T3  T4

Zag-Zag (Right Right Case):
 G                  P                    X
/  \              /   \                 / \
T1   P    leftRotate(G)  G    X    leftRotate(P)    P   T4
   / \   ============> / \  / \   ============>  / \
  T2   X            T1 T2 T3 T4             G   T3
     / \                               / \
    T3 T4                             T1  T2
```

……..3.b) Zig-Zag and Zag-Zig Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

```
Zag-Zig (Left Right Case):
     G                  G                    X
    / \               /   \                /   \
   P   T4  leftRotate(P)   X    T4   rightRotate(G)   P    G
  / \     ============> / \       ============>  / \  / \
 T1   X               P  T3                T1  T2 T3  T4
    / \             / \
   T2  T3          T1   T2

Zig-Zag (Right Left Case):
 G                  G                    X
/  \              /  \                 /   \
T1   P   rightRotate(P)  T1   X    leftRotate(P)    G    P
   / \  =============>    / \   ============>  / \  / \
  X  T4               T2   P            T1  T2 T3  T4
 / \                      / \
 T2  T3                 T3  T4
```
Example:
```

       100                 100                  [20]
       /  \                /   \                   \
     50   200            50    200                 50
    /       search(20)    /       search(20)       /  \
   40       ======>     [20]      ========>       30   100
  /          1. Zig-Zig    \        2. Zig-Zig      \     \
 30             at 40      30          at 100       40    200
 /                        \
[20]                       40
```
The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Summary

1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.

2) All splay tree operations take O(Logn) time on average. Splay trees can be rigorously shown to run in O(log n) average time per operation, over any sequence of operations (assuming we start from an empty tree)

3) Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.

4) Unlike AVL tree, a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications. Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software

Binomial Heap

The main application of Binary Heap is as implement priority queue. Binomial Heap is an extension of Binary Heap that provides faster union or merge operation together with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other.

A Binomial Tree of order k has following properties.

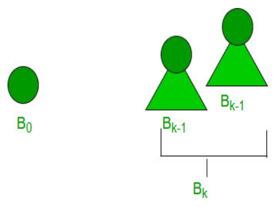
a) It has exactly 2^k nodes.

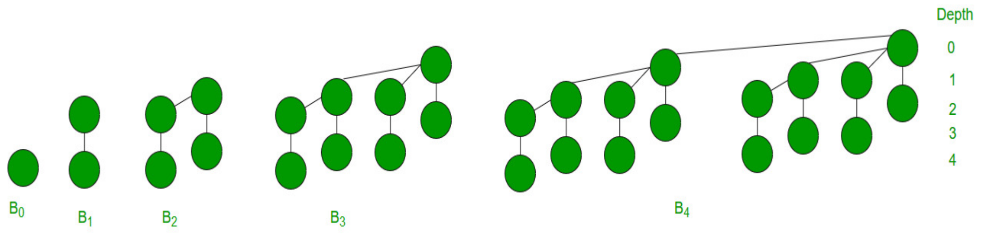
b) It has depth as k.

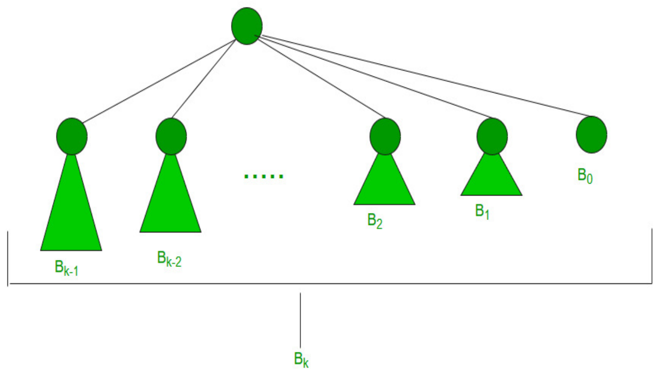
c) There are exactly ${}^kC_i$ nodes at depth i for i = 0, 1, . . . , k.

d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

The following diagram is referred from 2nd Edition of CLRS book.







Binomial Heap:
A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at most one Binomial Tree of any degree.
Examples Binomial Heap:

```
12------------10--------------------20
             /  \                 /  | \
           15    50             70  50  40
           |                  / |    |
           30                80  85  65
                            |
                           100
```

A Binomial Heap with 13 nodes. It is a collection of 3
Binomial Trees of orders 0, 2 and 3 from left to right.

```
    10--------------------20
   /  \                 /  | \
 15    50             70  50  40
 |                  / |    |
 30                80  85  65
                  |
                 100
```

A Binomial Heap with 12 nodes. It is a collection of 2

Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has the number of Binomial Trees equal to the number of set bits in the Binary representation of n. For example let n be 13, there 3 set bits in the binary representation of n (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are O(Logn) Binomial Trees in a Binomial Heap with 'n' nodes.

Operations of Binomial Heap:

The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

1) insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.

3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

Union operation in Binomial Heap:

Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.

1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.

——Case 1: Orders of x and next-x are not same, we simply move ahead.

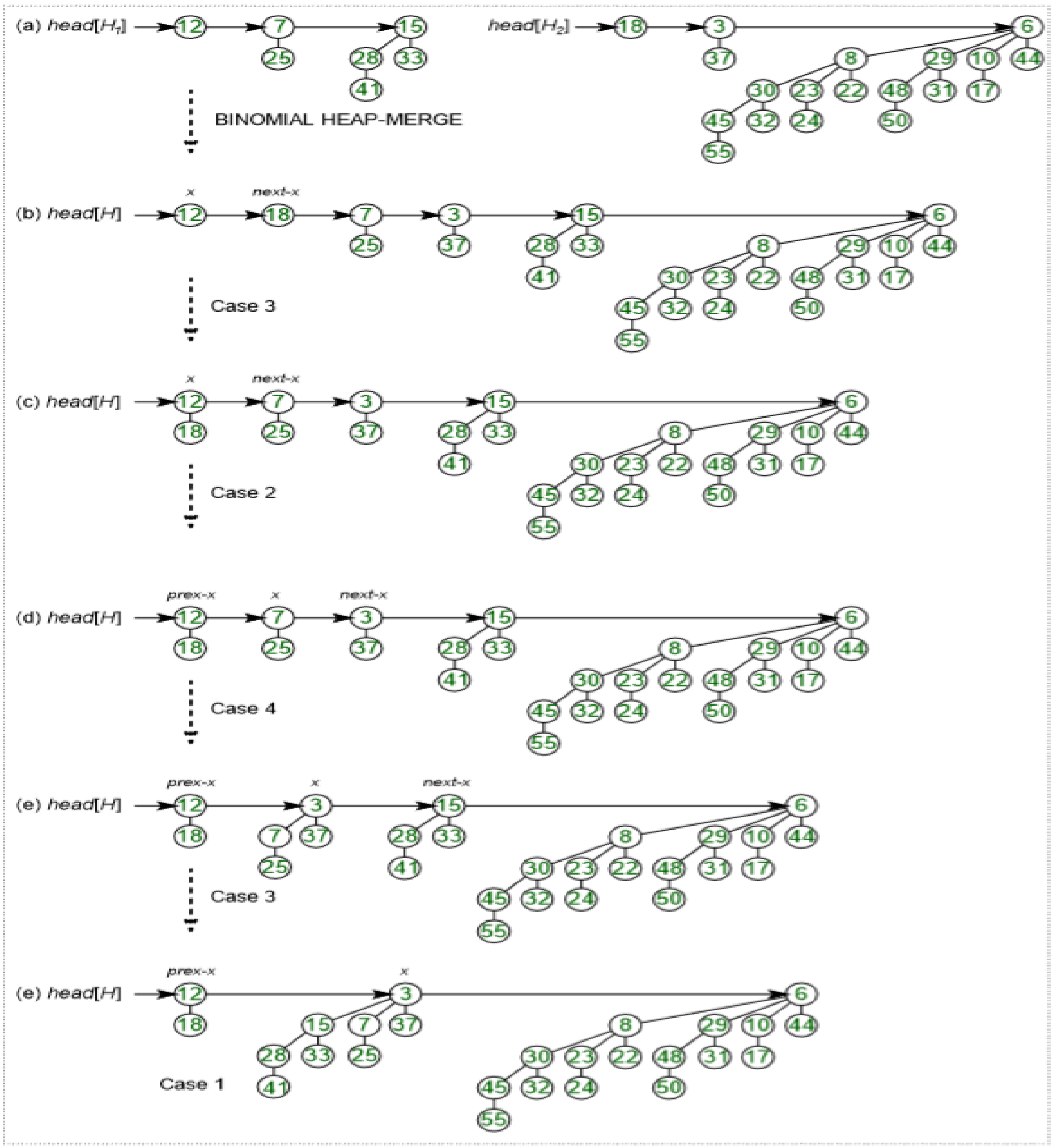
In following 3 cases orders of x and next-x are same.

——Case 2: If the order of next-next-x is also same, move ahead.

——Case 3: If the key of x is smaller than or equal to the key of next-x, then make next-x as a child of x by linking it with x.

——Case 4: If the key of x is greater, then make x as the child of next.

The following diagram is taken from 2nd Edition of CLRS book.



Fibonacci Heap

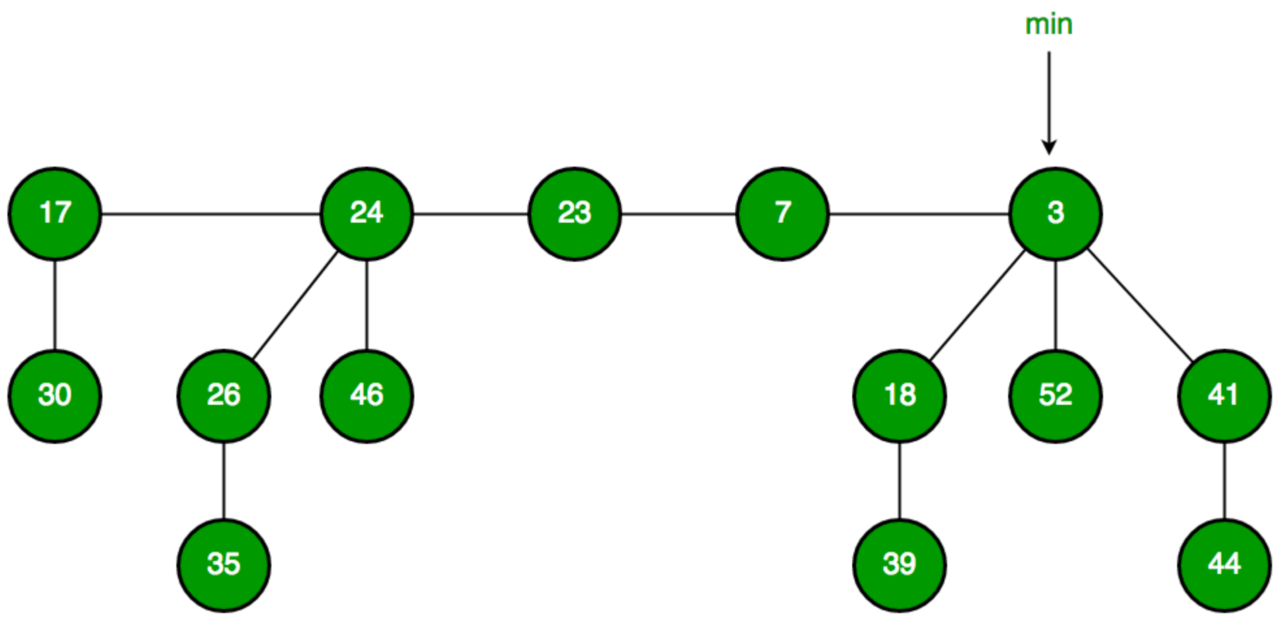
Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are amortized time complexities of **Fibonacci Heap**.

1) Find Min: **Θ(1)** [Same as both Binary and Binomial]
2) Delete Min: **O(Log n)** [Θ(Log n) in both Binary and Binomial]
3) Insert: **Θ(1)** [Θ(Log n) in Binary and Θ(1) in Binomial]
4) Decrease-Key: **Θ(1)** [Θ(Log n) in both Binary and Binomial]
5) Merge: **Θ(1)** [Θ(m Log n) or Θ(m+n) in Binary and Θ(Log n) in Binomial]

Like Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).
Below is an example Fibonacci Heap taken from here.



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.
The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.
Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is O(VLogV + ELogV). If Fibonacci Heap is used, then time complexity is improved to O(VLogV + E)
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source Wiki).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most

O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number.