# EST Solutions

Ans 1(a):

KMP Algorithm for Preprocessing:

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

KMP Algorithm for Searching:

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

Ans 1(b):

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT (S,x): inserts the element x into the set S, which is equivalent to the operation S = S U {x}.

MAXIMUM(S): returns the element of S with the largest key.

EXTRACT-MAX(S): removes and returns the element of S with the largest key.

INCREASE-KEY(S,x,k): increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Now we discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in O(1) time.

HEAP-MAXIMUM(A)

1 return A[1]

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop body (lines 3–5) of the HEAPSORT procedure.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

The running time of HEAP-EXTRACT-MAX is O(log n), since it performs only a constant amount of work on top of the O(log n) time for MAX-HEAPIFY.

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index i into the array identifies the priority-queue element whose key we wish to increase. The procedure first updates the key of element A[i] to its new value. Because increasing the key of A[i] might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop of INSERTION-SORT, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As HEAP-INCREASE KEY traverses this path, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds.

```
HEAP-INCREASE-KEY (A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

The running time of HEAP-INCREASE-KEY on an n-element heap is O(log n), since the path traced from the node updated in line 3 to the root has length O(log n).
The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is -infinity. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

```
MAX-HEAP-INSERT (A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY (A, A.heap-size, key)
```

The running time of MAX-HEAP-INSERT on an n-element heap is O(log n).
In summary, a heap can support any priority-queue operation on a set of size n in O(log n) time.

Ans 2 (a):

```
Topological-sort(G)
    for all edges (u, v) in E
        incoming[v]← incoming[v] +1
    for all v in V
        if incoming[v]=0
            Enqueue(S, v)
    While S not empty
        u ← Dequeue(S)
        add u to linked list
        for each edge(u, v) in E
            incoming[v] ← incoming[v]-1
            if active[v]=0
                Enqueue(S, v)
```

**Compute the in-degrees:**

V1: 0
V2: 1
V3: 2
V4: 2
V5: 2

**Find a vertex with in-degree 0: V1**
**Output V1, remove V1 and update the in-degrees:**
**Sorted: V1**
**Remove edges: (V1,V2), (V1, V3) and (V1,V4)**
**Updated in-degrees:**

V2: 0
V3: 1
V4: 1
V5: 2

**The process is depicted in the following table:**

| | Indegree | | | | | |
|---|---|---|---|---|---|---|
| Sorted | | V1 | V1,V2 | V1,V2,V4 | V1,V2,V4,V3 | V1,V2,V4,V3,V5 |
| V1 | 0 | | | | | |
| V2 | 1 | 0 | | | | |
| V3 | 2 | 1 | 1 | 0 | | |
| V4 | 2 | 1 | 0 | | | |
| V5 | 2 | 2 | 1 | 0 | 0 | |

One possible sorting: **V1, V2, V4, V3, V5**
Another sorting: **V1, V2, V4, V5, V3**

Ans 2(b): Although merge sort runs in O(n lg n) worst-case time and insertion sort runs in O(n2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when sub-problems become sufficiently small.
A modification to merge sort can be: in which n/k sub-lists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined. In practice, k should be the largest list length on which insertion sort is faster than merge sort.

```
MERGE-SORT(A, p, r, k)
  if p - r <= k
      INSERTION-SORT(A, p, r)
  else
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q, k)
      MERGE-SORT(A, q + 1, r, k)
      MERGE(A, p, q, r)
```

```
INSERTION-SORT(A, p, r)
1  for j = p + 1 to r
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Ans 3(a):

**Dijkstra's Algorithm:**

- Finds shortest path from a given startNode to all other nodes reachable from it in a digraph.
- Assumes that each link cost $c(x, y) \geq 0$.
- Complexity: $O(N^2)$, $N$ = #(nodes in the digraph)

**Floyd's Algorithm:**

- Finds a shortest-path for all node-pairs $(x, y)$.
- We can have one or more links of negative cost, $c(x, y) < 0$, but no cycle of negative cost. (Assume that $c(x_i, x_i) = 0$ for each node $x_i$, which is the same as not having the links $(x_i, x_i)$.)
- Complexity: $O(N^3)$, where $N$ = #(nodes in digraph).

**Matrix Initialization**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 0 | 5 | 0 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 0 | 5 | 0 | 0 | 0 |
| D | 5 | 5 | 0 | 0 | 20 |
| E | 0 | 10 | 0 | 20 | 0 |

**Step 1**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 0 | 5 | 0 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 0 | 5 | 0 | 0 | 0 |
| D | 5 | 5 | 0 | 0 | 20 |
| E | 0 | 10 | 0 | 20 | 0 |

**Step 2**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 15 | 5 | 20 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 15 | 5 | 0 | 10 | 15 |
| D | 5 | 5 | 10 | 0 | 15 |
| E | 20 | 10 | 15 | 15 | 0 |

**Step 3**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 15 | 5 | 20 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 15 | 5 | 0 | 10 | 15 |
| D | 5 | 5 | 10 | 0 | 15 |
| E | 20 | 10 | 15 | 15 | 0 |

**Step 4**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 15 | 5 | 20 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 15 | 5 | 0 | 10 | 15 |
| D | 5 | 5 | 10 | 0 | 15 |
| E | 20 | 10 | 15 | 15 | 0 |

**Step 5**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 10 | 15 | 5 | 20 |
| B | 10 | 0 | 5 | 5 | 10 |
| C | 15 | 5 | 0 | 10 | 15 |
| D | 5 | 5 | 10 | 0 | 15 |
| E | 20 | 10 | 15 | 15 | 0 |

Ans 3(b):

Dynamic finger property : if last access was to key $x_{i-1}$ and the present access is to key $x_i$ then the cost of the present access is $O(\lg|x_i-(x_{i-1})|)$

This property performs well on access patterns which exhibit spatial locality.

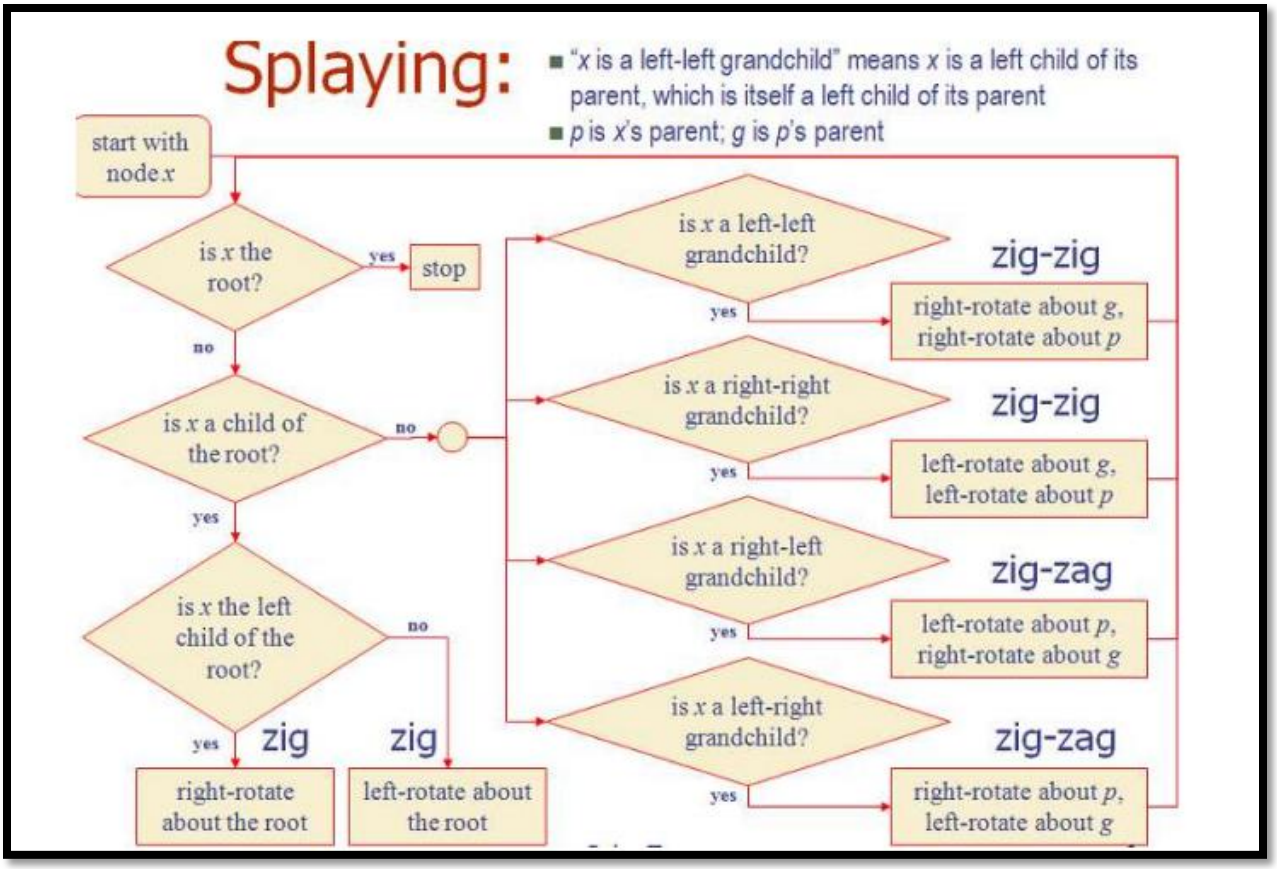
Working Set property: Let $t_j$ the number of distinct keys between the present operation and the last time the same key was accessed. Let $t_j$ be the number of distinct keys in $\{x_i..x_j\}$ then the cost of accessing $x_j$ is $O(\lg t_j+2)$

Data structures with this property perform well on access patterns which exhibit temporal locality.

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

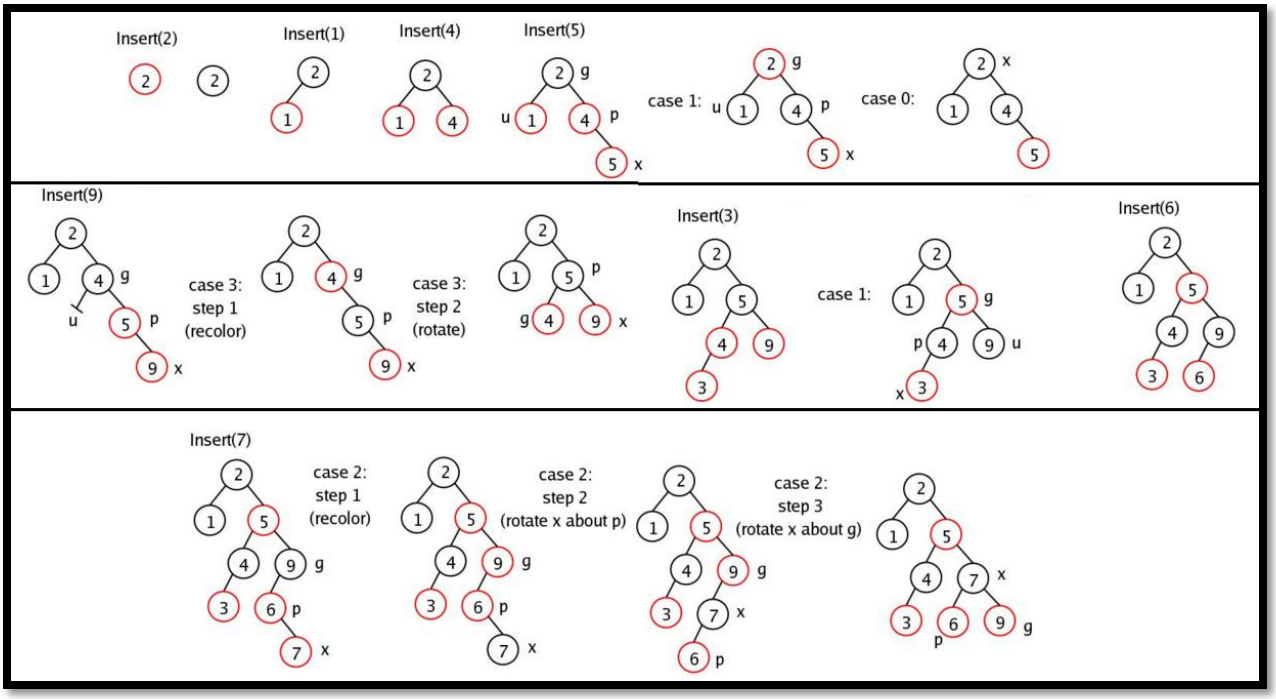
Whether x is the left or right child of its parent node, y,
whether y is the root or not, and if not
whether y is the left or right child of its parent, g (the grandparent of x).




Ans 4(a):

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```



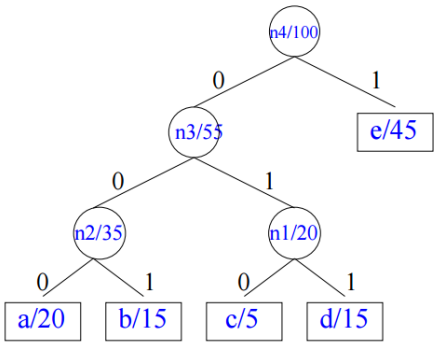

Ans 4(b): Dynamic programming typically applies to optimization problems in which we make a set of choices in order to arrive at an optimal solution. As we make each choice, sub-problems of the same form often arise. Dynamic programming is effective when a given sub-problem may arise from more than one partial set of choices; the key technique is to store the solution to each such sub-problem in case it should reappear. Eg: Matrix Chain Multiplication, Knapsack etc.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which we make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner. A simple example is coin-changing: to minimize the number of U.S. coins needed to make change for a given amount, we can repeatedly select the largest-denomination coin that is not larger than the amount that remains. A greedy approach provides an optimal solution for many such problems much more quickly than would a dynamic-programming approach. We cannot always easily tell whether a greedy approach will be effective, however. Some

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only to show that the principle of optimality applies to the problem.

Ans 5(a):

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```



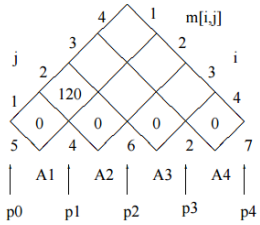

$\{a = 000, b = 001, c = 010, d = 011, e = 1\}$

Ans 5(b):

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4       m[i, i] = 0
 5  for l = 2 to n                // l is the chain length
 6       for i = 1 to n − l + 1
 7            j = i + l − 1
 8            m[i, j] = ∞
 9            for k = i to j − 1
10                 q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11                 if q < m[i, j]
12                      m[i, j] = q
13                      s[i, j] = k
14  return m and s
```

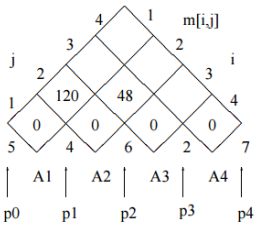
**Stp 1: Computing** $m[1,2]$ By definition

$$m[1,2] = \min_{1\le k<2}(m[1,k]+m[k+1,2]+p_0p_kp_2)$$
$$= m[1,1]+m[2,2]+p_0p_1p_2 = 120.$$



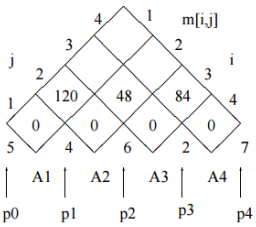

**Stp 2: Computing** $m[2,3]$ By definition

$$m[2,3] = \min_{2\le k<3}(m[2,k]+m[k+1,3]+p_1p_kp_3)$$
$$= m[2,2]+m[3,3]+p_1p_2p_3 = 48.$$



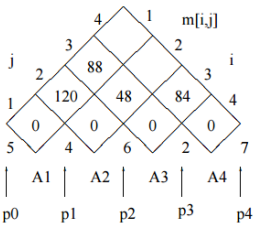

**Stp3: Computing** $m[3,4]$ By definition

$$m[3,4] = \min_{3\le k<4}(m[3,k]+m[k+1,4]+p_2p_kp_4)$$
$$= m[3,3]+m[4,4]+p_2p_3p_4 = 84.$$



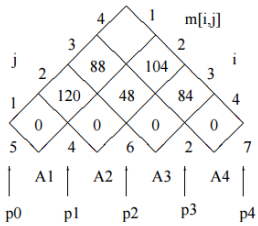

**Stp4: Computing** $m[1,3]$ By definition

$$m[1,3] = \min_{1\le k<3}(m[1,k]+m[k+1,3]+p_0p_kp_3)$$
$$= \min\begin{Bmatrix} m[1,1]+m[2,3]+p_0p_1p_3 \\ m[1,2]+m[3,3]+p_0p_2p_3 \end{Bmatrix}$$
$$= 88.$$



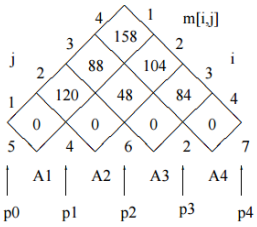

**Stp5: Computing** $m[2,4]$ By definition

$$m[2,4] = \min_{2\le k<4}(m[2,k]+m[k+1,4]+p_1p_kp_4)$$
$$= \min\begin{Bmatrix} m[2,2]+m[3,4]+p_1p_2p_4 \\ m[2,3]+m[4,4]+p_1p_3p_4 \end{Bmatrix}$$
$$= 104.$$




**St6: Computing** $m[1,4]$ By definition

$$m[1,4] = \min_{1\le k<4}(m[1,k]+m[k+1,4]+p_0p_kp_4)$$
$$= \min\begin{Bmatrix} m[1,1]+m[2,4]+p_0p_1p_4 \\ m[1,2]+m[3,4]+p_0p_2p_4 \\ m[1,3]+m[4,4]+p_0p_3p_4 \end{Bmatrix}$$
$$= 158.$$



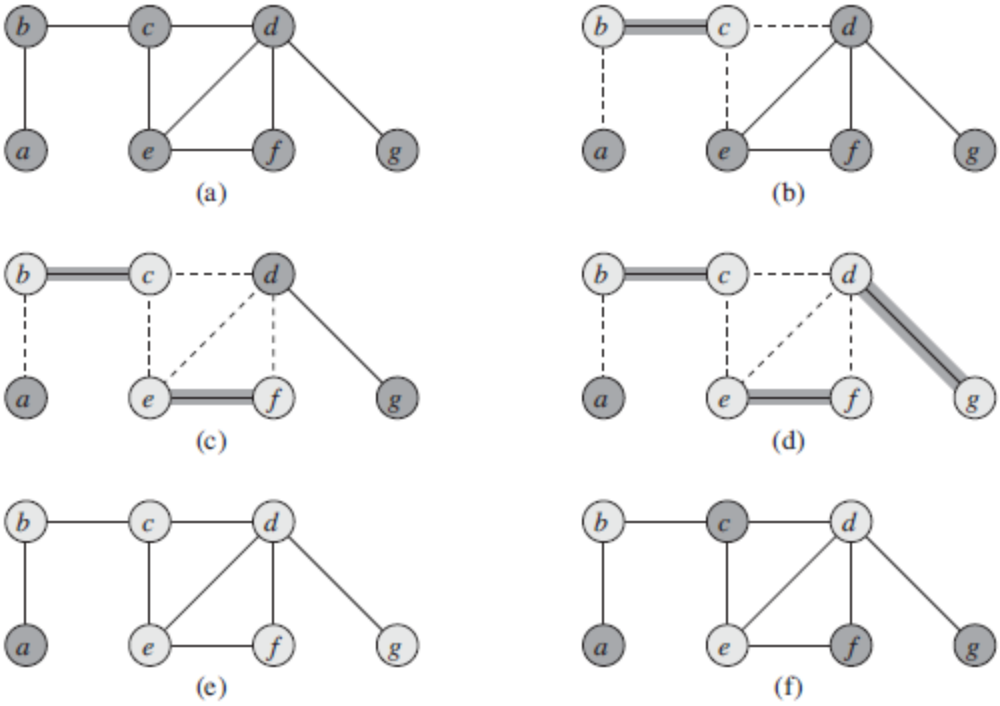

Ans 6:

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

Even though we don't know how to find an optimal vertex cover in a graph $G$ in polynomial time, we can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph $G$ and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

APPROX-VERTEX-COVER$(G)$

```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```




The operation of APPROX-VERTEX-COVER. (**a**) The input graph $G$, which has 7 vertices and 8 edges. (**b**) The edge $(b, c)$, shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices $b$ and $c$, shown lightly shaded, are added to the set $C$ containing the vertex cover being created. Edges $(a, b)$, $(c, e)$, and $(c, d)$, shown dashed, are removed since they are now covered by some vertex in $C$. (**c**) Edge $(e, f)$ is chosen; vertices $e$ and $f$ are added to $C$. (**d**) Edge $(d, g)$ is chosen; vertices $d$ and $g$ are added to $C$. (**e**) The set $C$, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices $b, c, d, e, f, g$. (**f**) The optimal vertex cover for this problem contains only three vertices: $b$, $d$, and $e$.

Figure illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable $C$ contains the vertex cover being constructed. Line 1 initializes $C$ to the empty set. Line 2 sets $E'$ to be a copy of the edge set $G.E$ of the graph. The loop of lines 3–6 repeatedly picks an edge $(u, v)$ from $E'$, adds its endpoints $u$ and $v$ to $C$, and deletes all edges in $E'$ that are covered by either $u$ or $v$. Finally, line 7 returns the vertex cover $C$. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent $E'$.

Ans 7(i):

## Deletion in Skip List

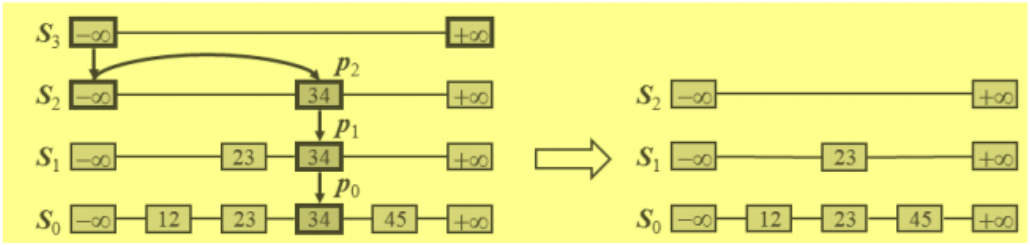
Remove(*x*)

Search for $x$ in skip list and find positions $p_0, p_1, \ldots p_i$ of the items with key $x$, where position $p_j$ is in list $S_j$

Remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$

Remove all but one list containing only two special keys

## Example remove 34




Ans 7(ii):

## AVL deletion

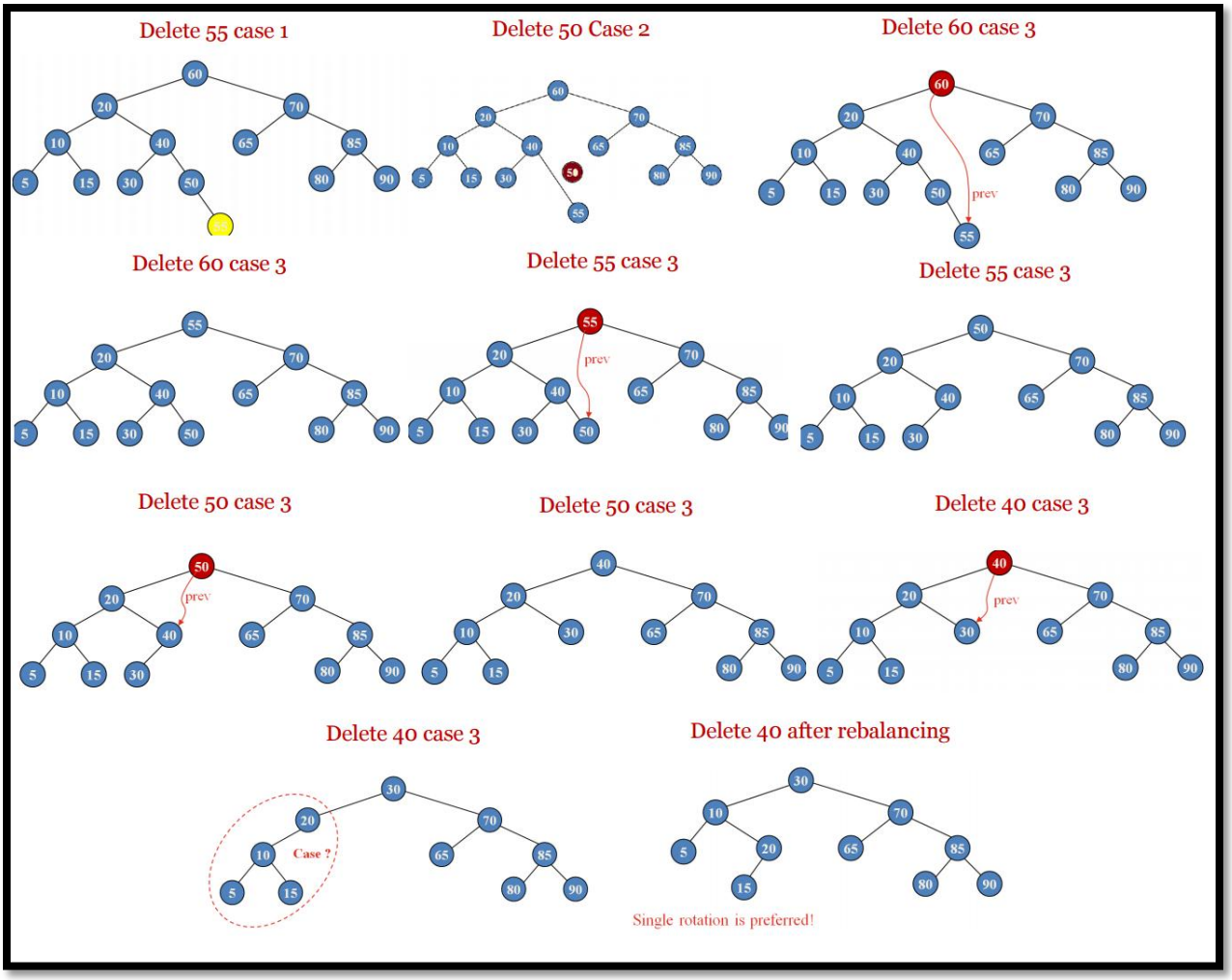
Deletion:

Case 1: if X is a leaf, delete X

Case 2: if X has 1 child, use it to replace X

Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)

Rebalancing


Delete 55 case 1
Delete 50 Case 2
Delete 60 case 3
prev
Delete 60 case 3
Delete 55 case 3
prev
Delete 55 case 3
Delete 50 case 3
prev
Delete 50 case 3
Delete 40 case 3
prev
Delete 40 case 3
Case ?
Delete 40 after rebalancing
Single rotation is preferred!