SORT AND SELECTION

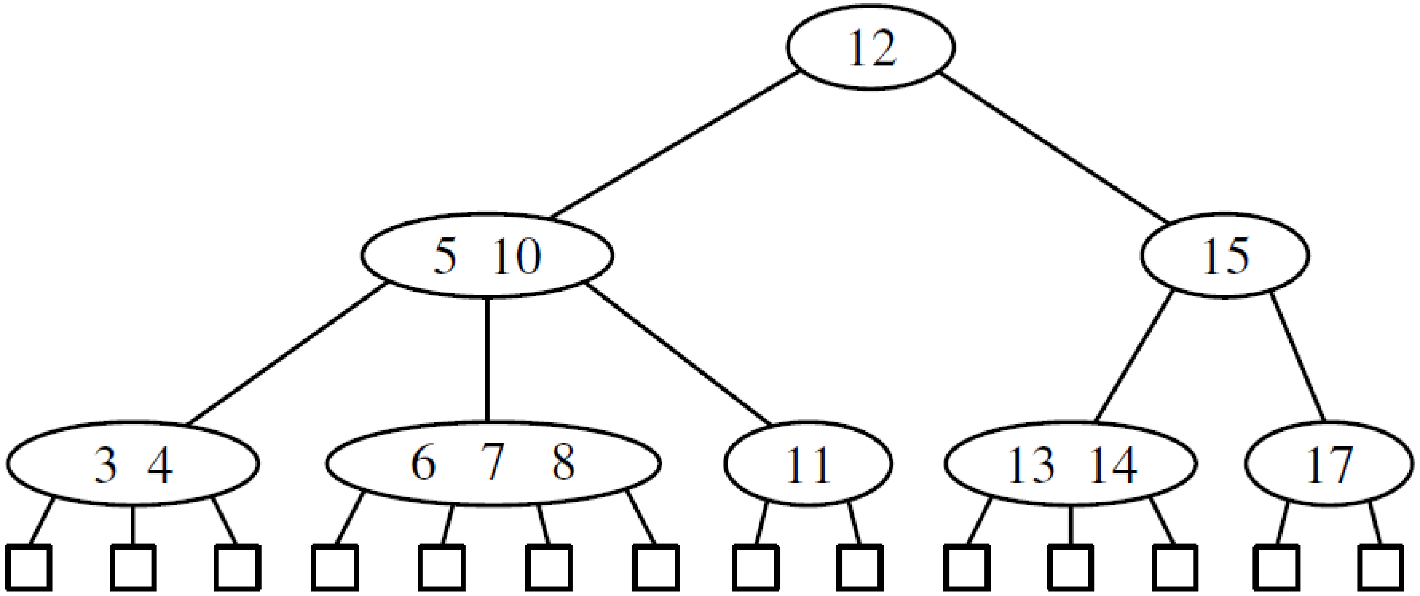
School of Artificial Intelligence

PREVIOUSLY ON BINARY SEARCH TREES

- Binary Search Tree
  - Performance: O(h)
- Balancing search tree
  - Rotation
    - X-Y rotation
    - Trinode rotation
- AVL tree
  - Height of AVL tree: number of nodes in a path
  - Height balance property
- Splay tree
  - Splay operations: search, add, remove
- 2-4 tree
  - Multi-way search tree
- Red black tree
  - Balanced search tree constraint by 3 main properties

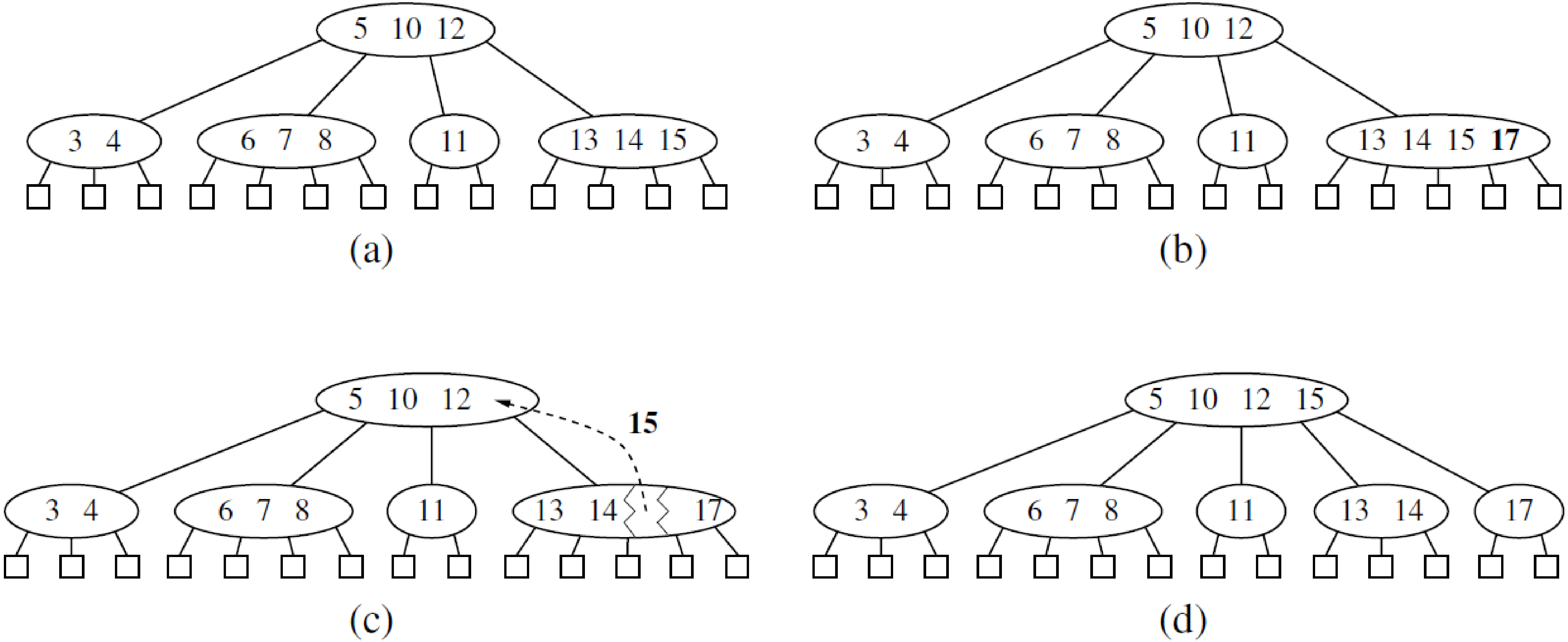
(2,4) TREES

- Sometimes 2-4 tree or 2-3-4 tree
- **Size property**: every internal node has at most four children
- **Depth property**: all external nodes have the same depth
- The height of a 2-4 tree storing n items is O (log n)
  - Sufficient to keep the tree balanced
  - Search takes O(log n) time



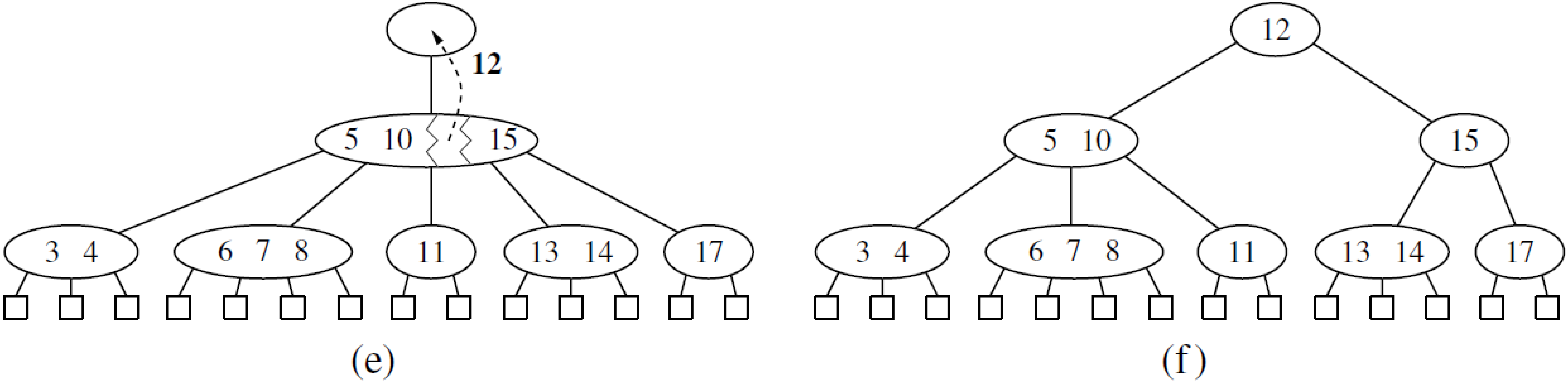
(2,4) TREES INSERTION

- Node split



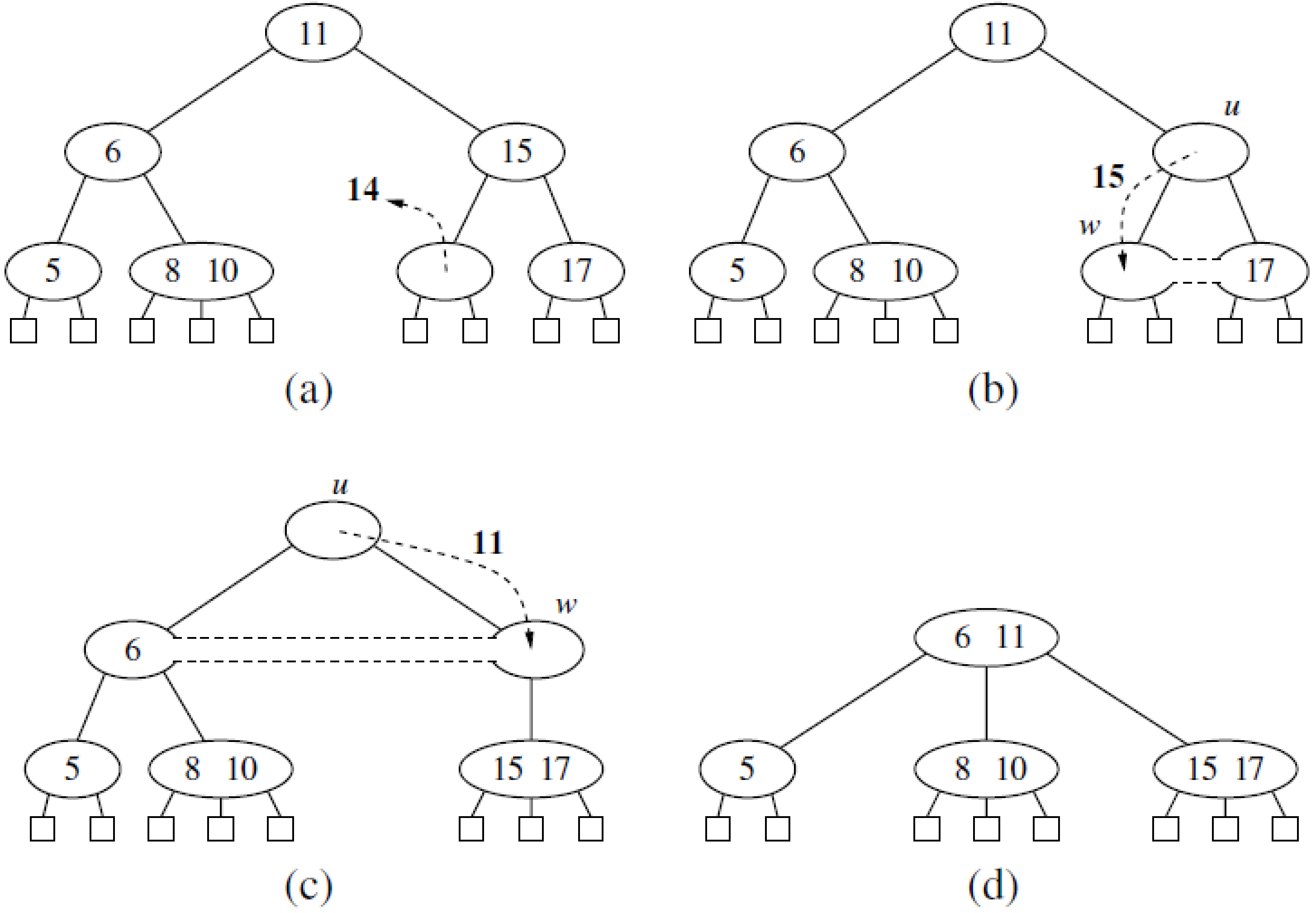
(2,4) TREES INSERTION

- Node split



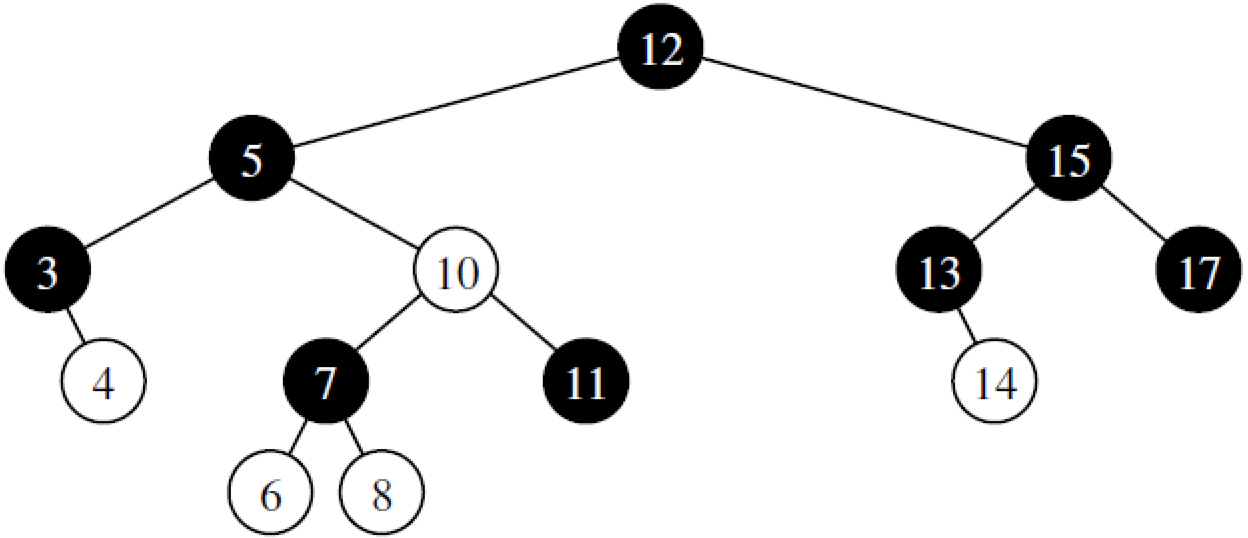
(2,4) TREES DELETION

- Fusion at node w may cause a new underflow to occur at the parent u of w, which triggers a transfer or fusion at u
- Number of fusion operations is bounded by the height of the tree – O(log n)



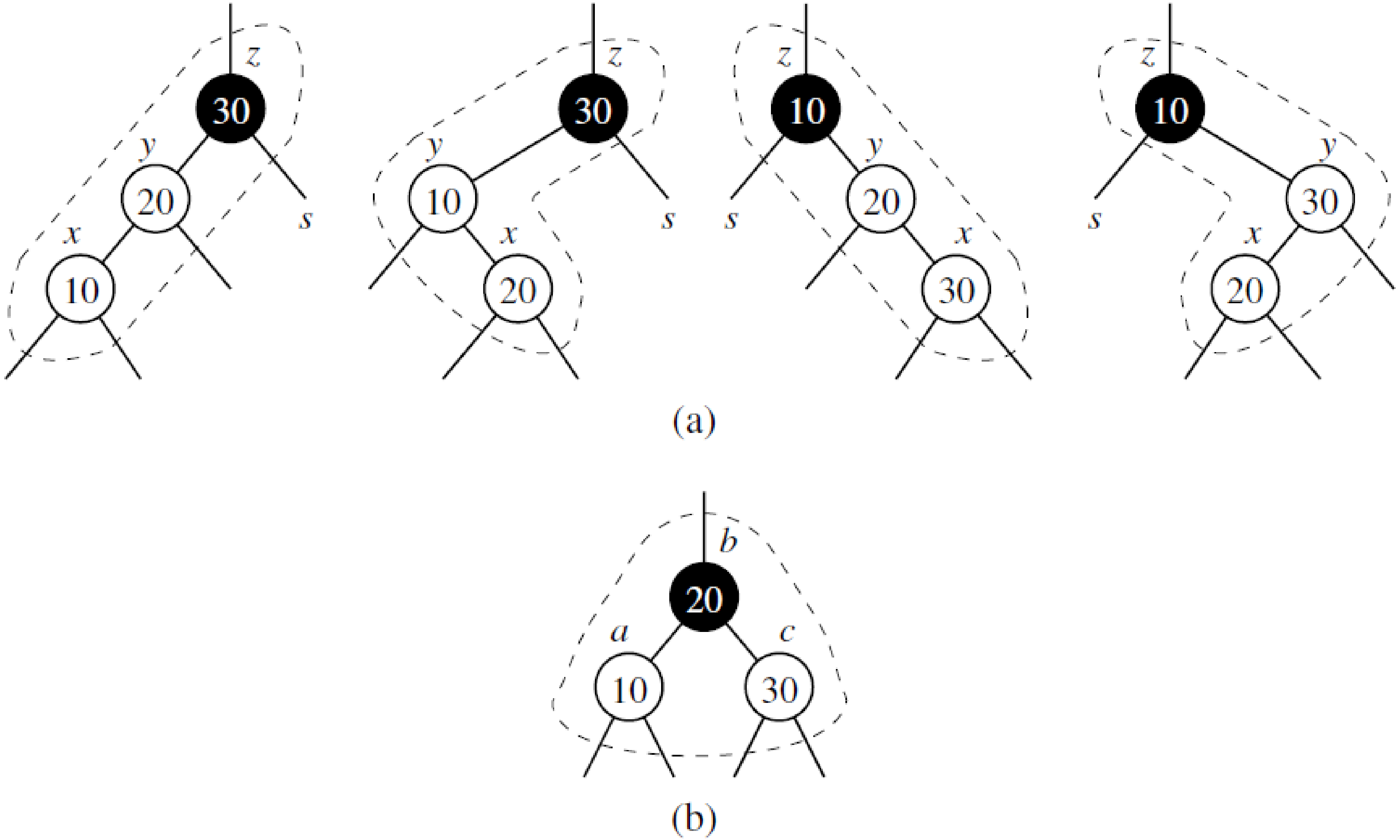
RED-BLACK TREES

- AVL trees – need to perform rotations
- 2-4 trees – need to perform split and fusion operations
- Red-black trees: O(1) structural changes after an update to stay balanced
- Red-black tree
  - Binary search tree, nodes coloured
  - **Root property**: root is black
  - **Red property**: the children of a red node are black
  - **Depth property**: all nodes with zero or one children have the same black depth
  - **Black depth**: number of black ancestors



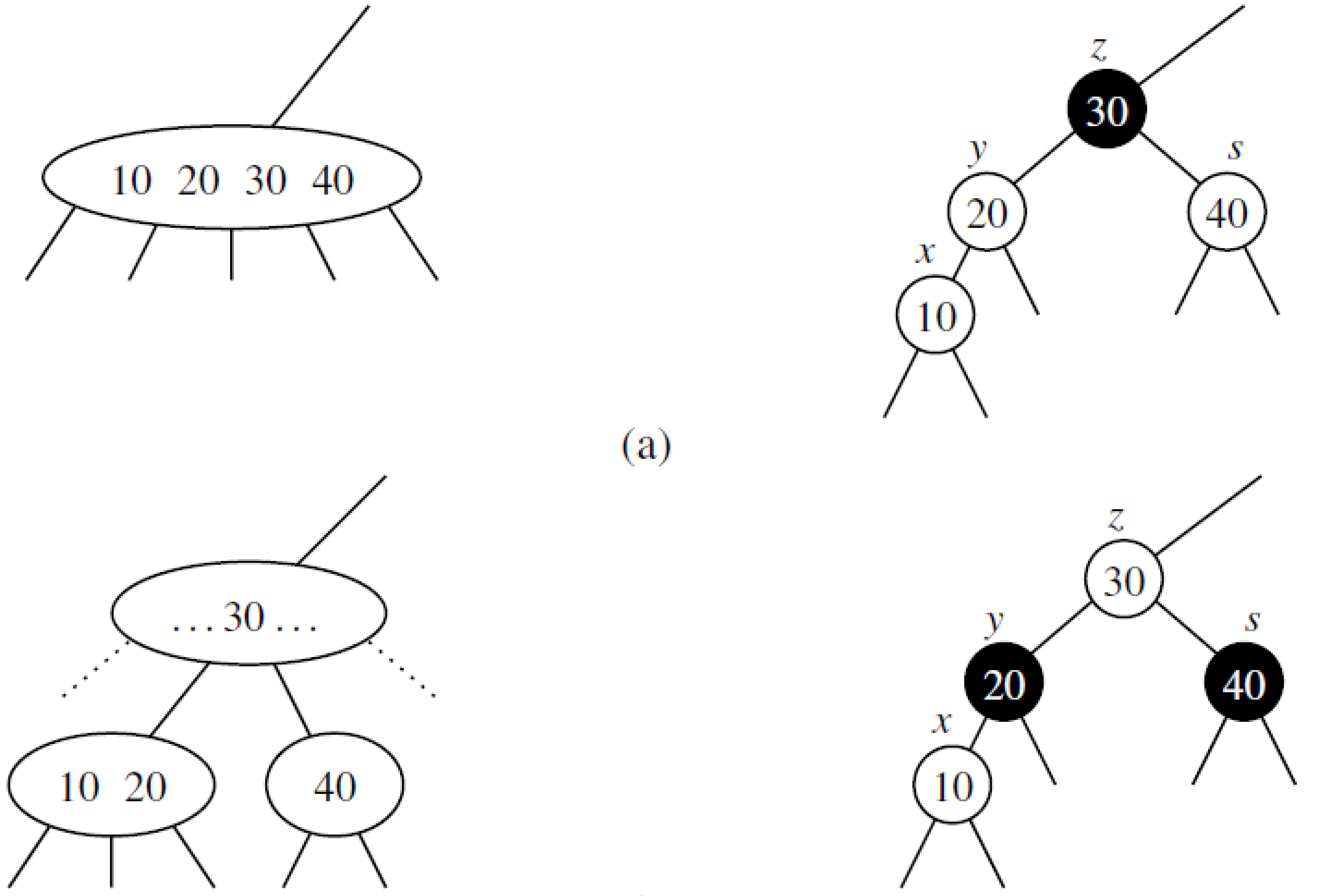
RED-BLACK TREES INSERTION

- Case 1: The sibling s of y is black (or None)
- Trinode restructuring
  - Node x, y, z
  - Label them a, b, and c
  - Replace z with the node labeled b and make nodes a and c the children of b
  - Colour b black and a and c red



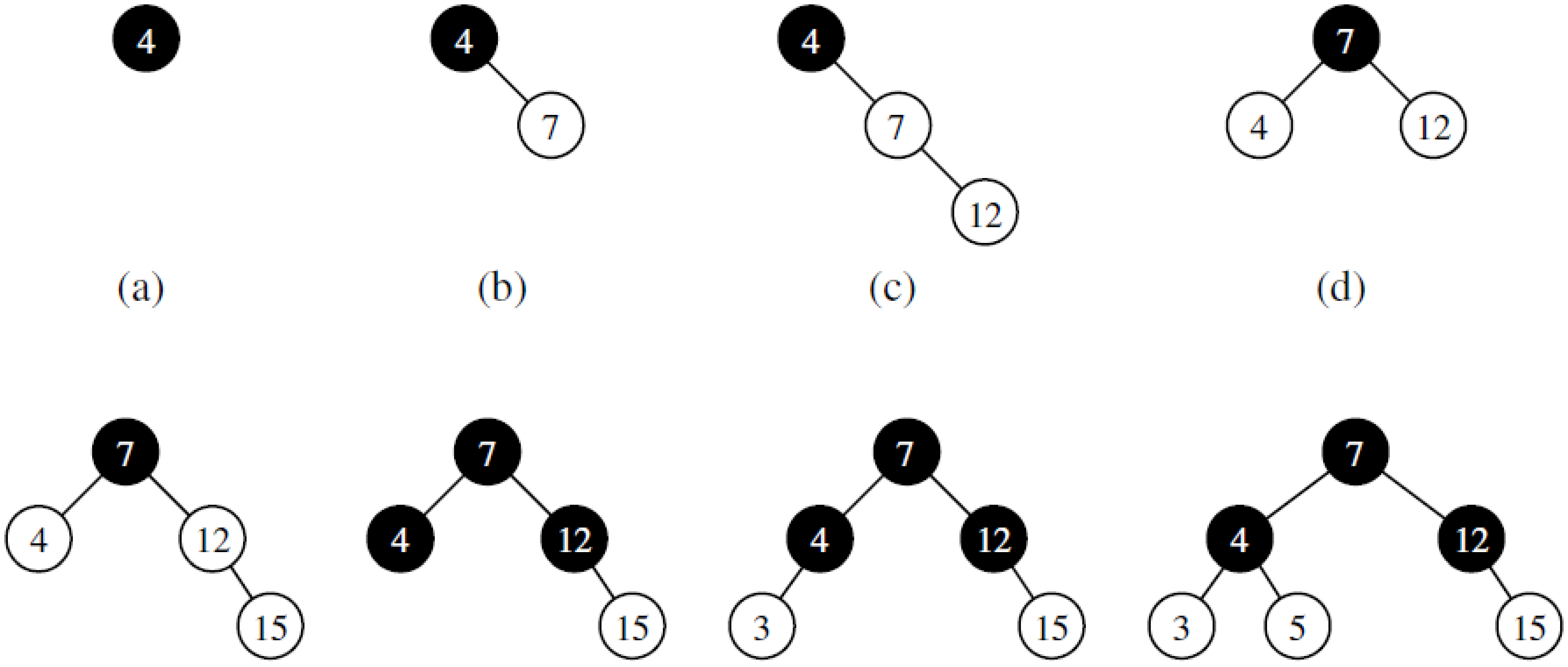
RED-BLACK TREES INSERTION

- Case 2: sibling s of y is red
- Overflow in its equivalent 2-4 tree
- Fix: **split/recolouring**
- Colour y and s black and their parent z red
- If z is root, it remains black
    - Unless z is the root, the portion of any path through the affected part of the tree is incident to one black node



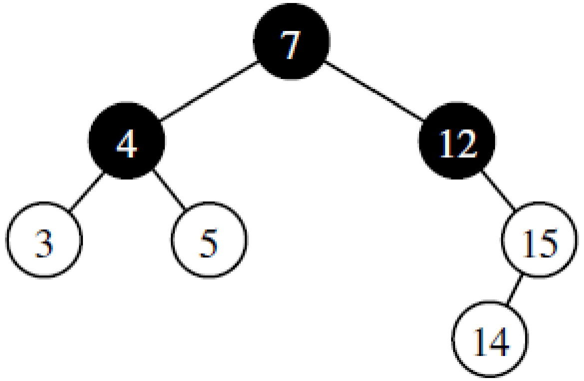
(a)
(b)

RED-BLACK TREES INSERTION

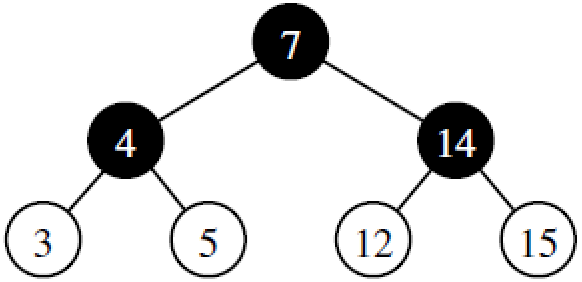


(a) (b) (c) (d) (e) (f) (g) (h)

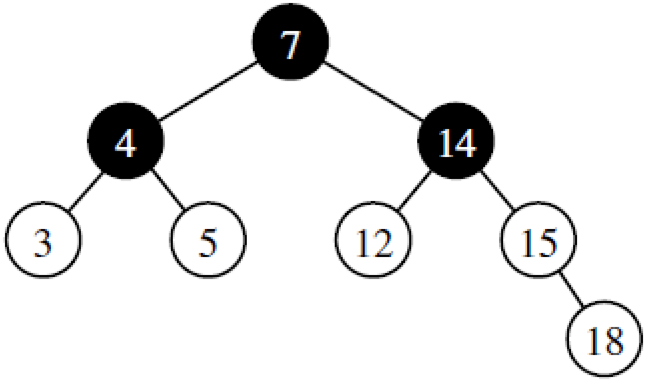
RED-BLACK TREES INSERTION



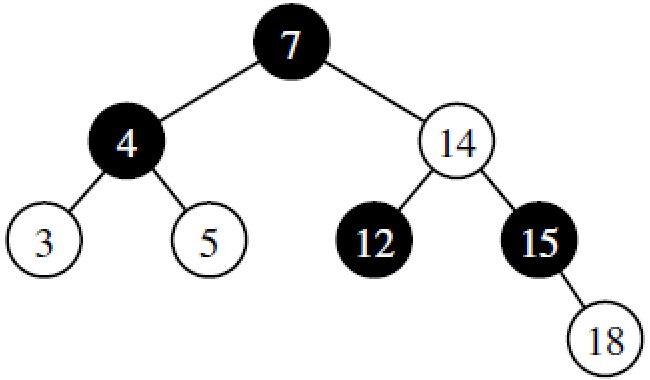
(i)



(j)

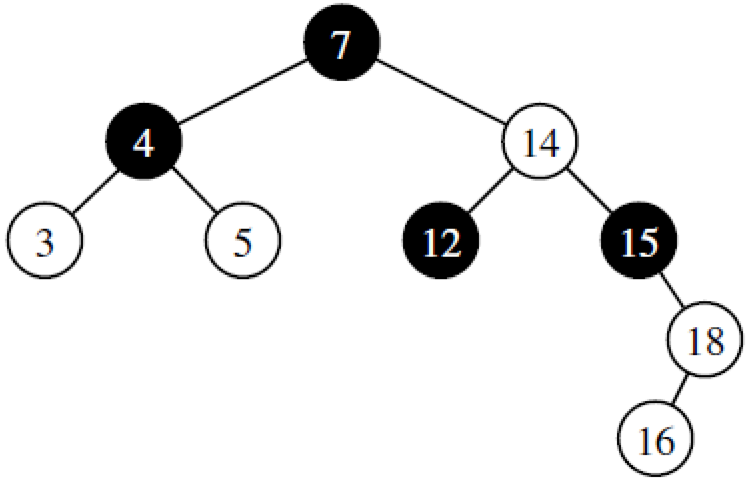


(k)

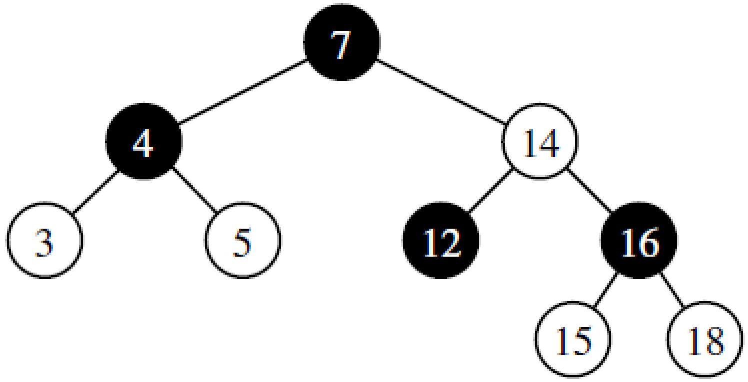


(l)

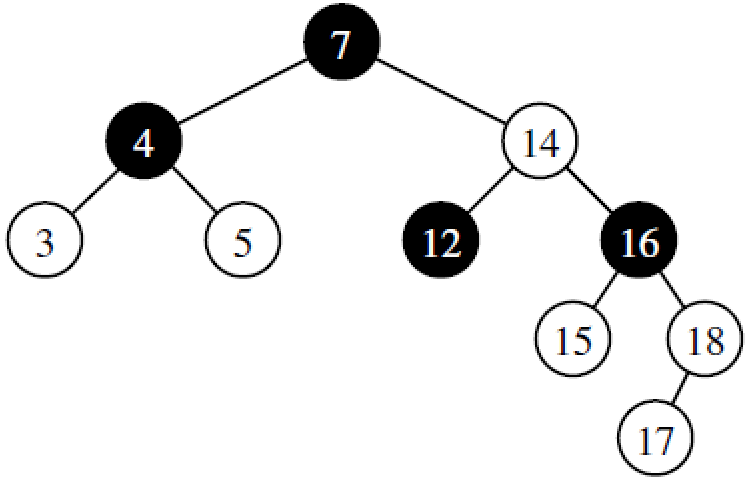
RED-BLACK TREES INSERTION



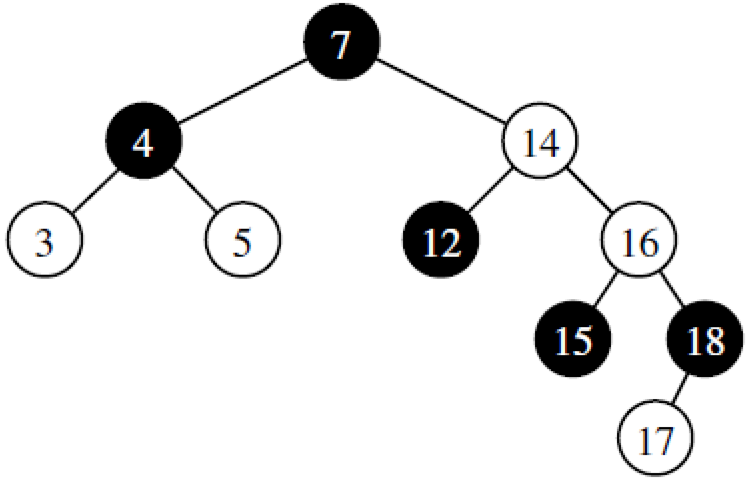
(m)



(n)

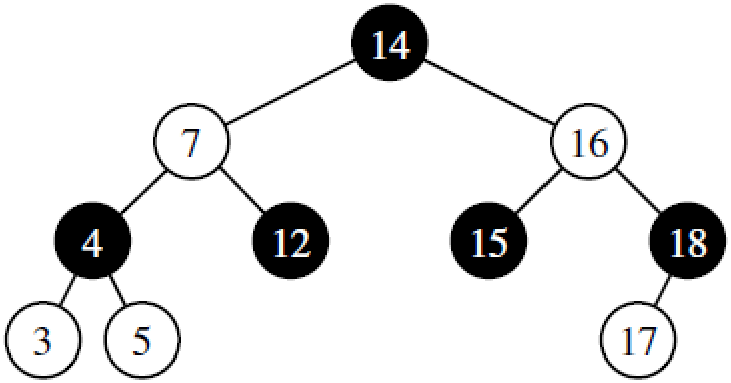


(o)



(p)

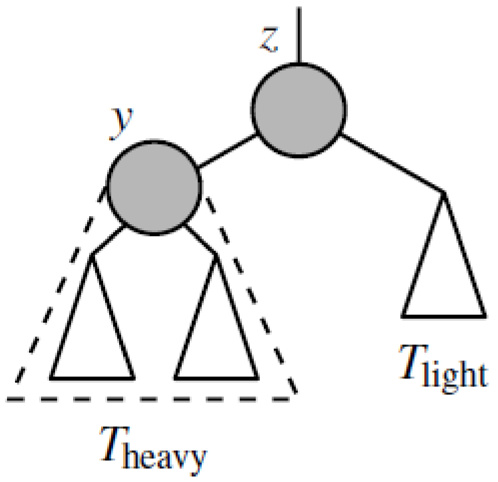
RED-BLACK TREES INSERTION



(q)

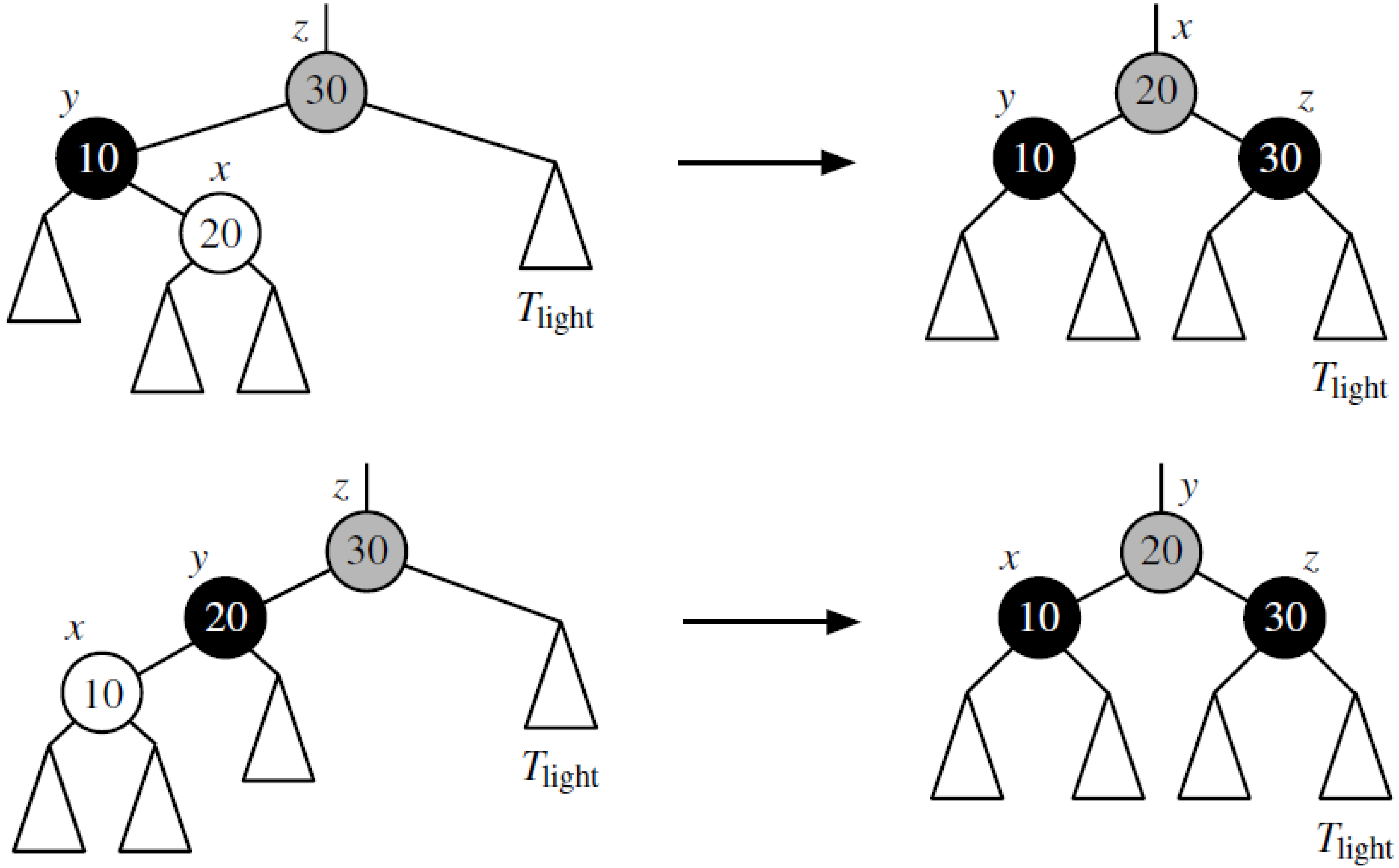
RED-BLACK TREES DELETION

- Search – O(log n)
- If removed node is red – no affect on the black depth property, or red violations
- If removed node is black and has one child that was a red leaf
  - Recolour solves the problem
- If removed node is a black leaf
  - Black deficit of 1
  - Removed node must have a sibling whose subtree has black height 1
  - More general setting with a node z with two subtrees: T heavy and T light. Black depth of T heavy is one more than T light
  - Z: parent of removed leaf
  - Y: root of T heavy



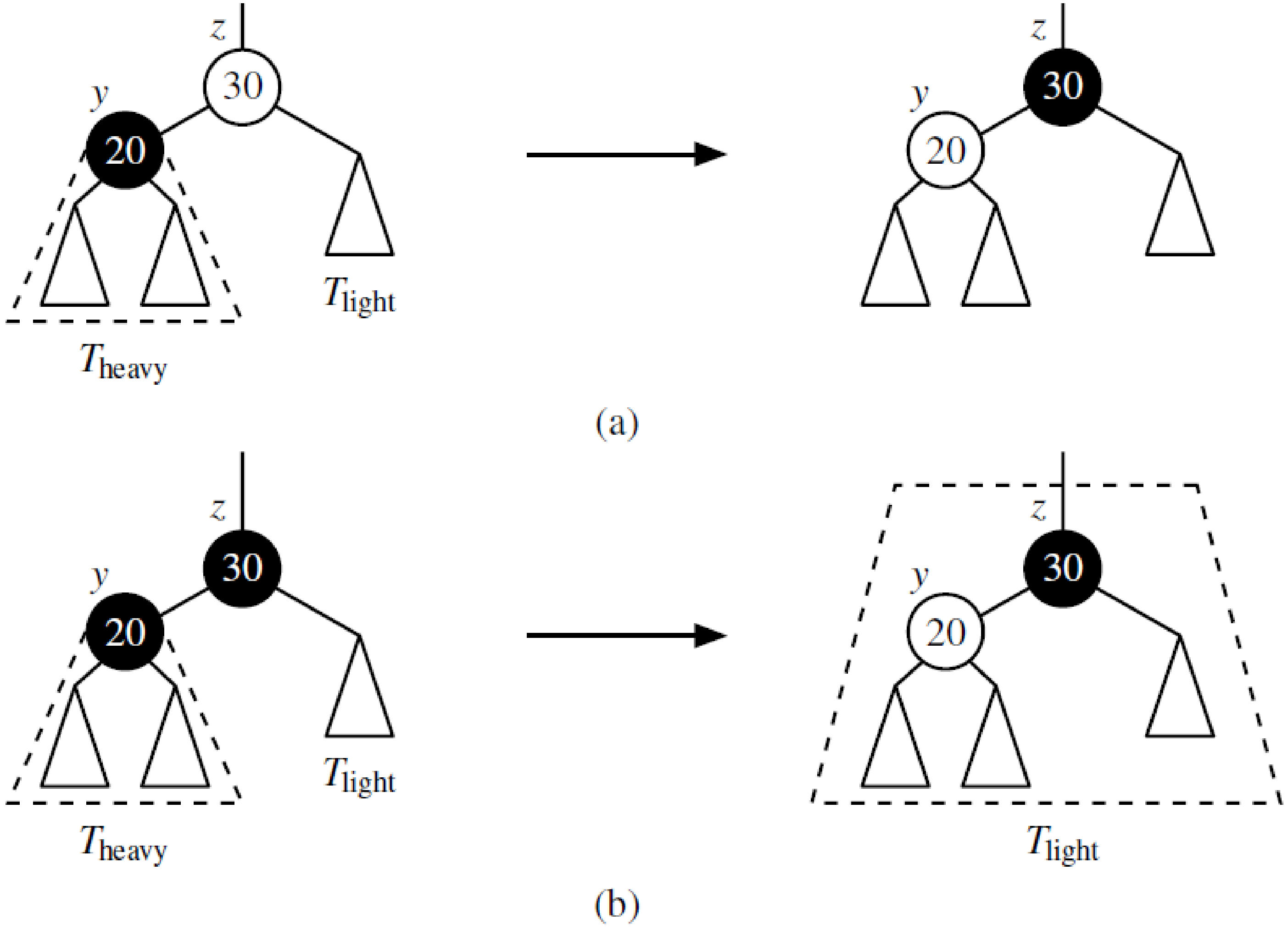
RED-BLACK TREES DELETION

- Case 1: node y is black and has a red child x
- Trinode restructuring:
- x, y and z
- a, b and c
- Make b the parent of the other two
- Colour a and c black
- Give b the previous colour of z



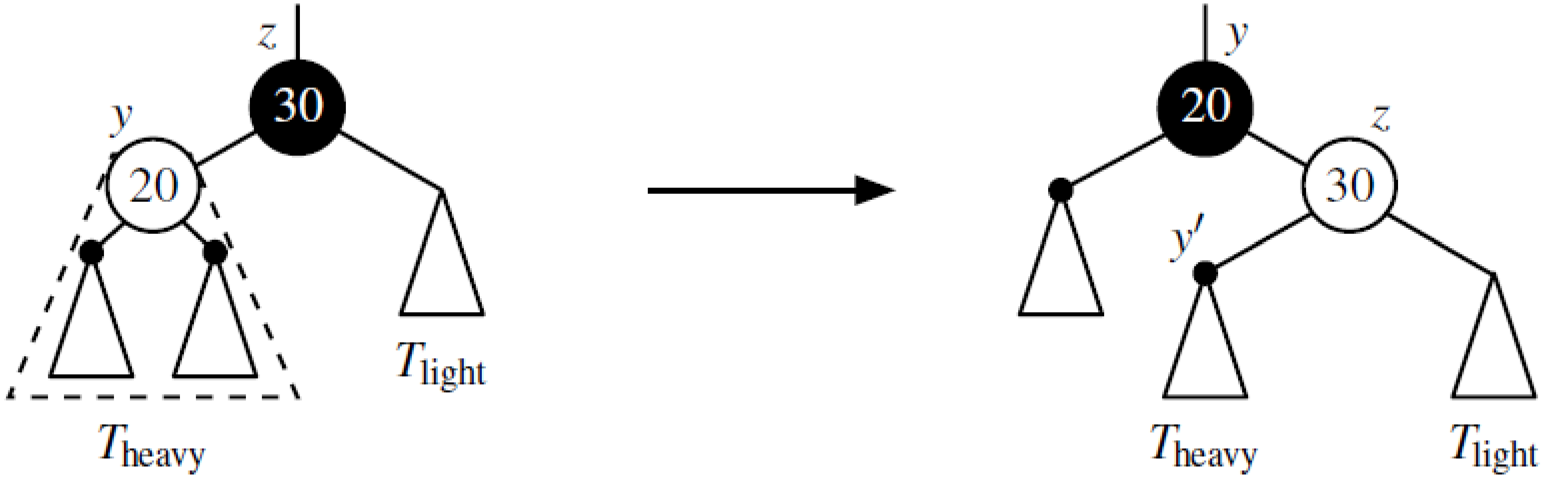
RED-BLACK TREES DELETION

- Case 2: node y is black and both children of y are black(or None)
- Recolouring: colour y red and if z is red, colour it black
- Z becomes deficient, repeat consideration of all three cases at the parent of z as a remedy

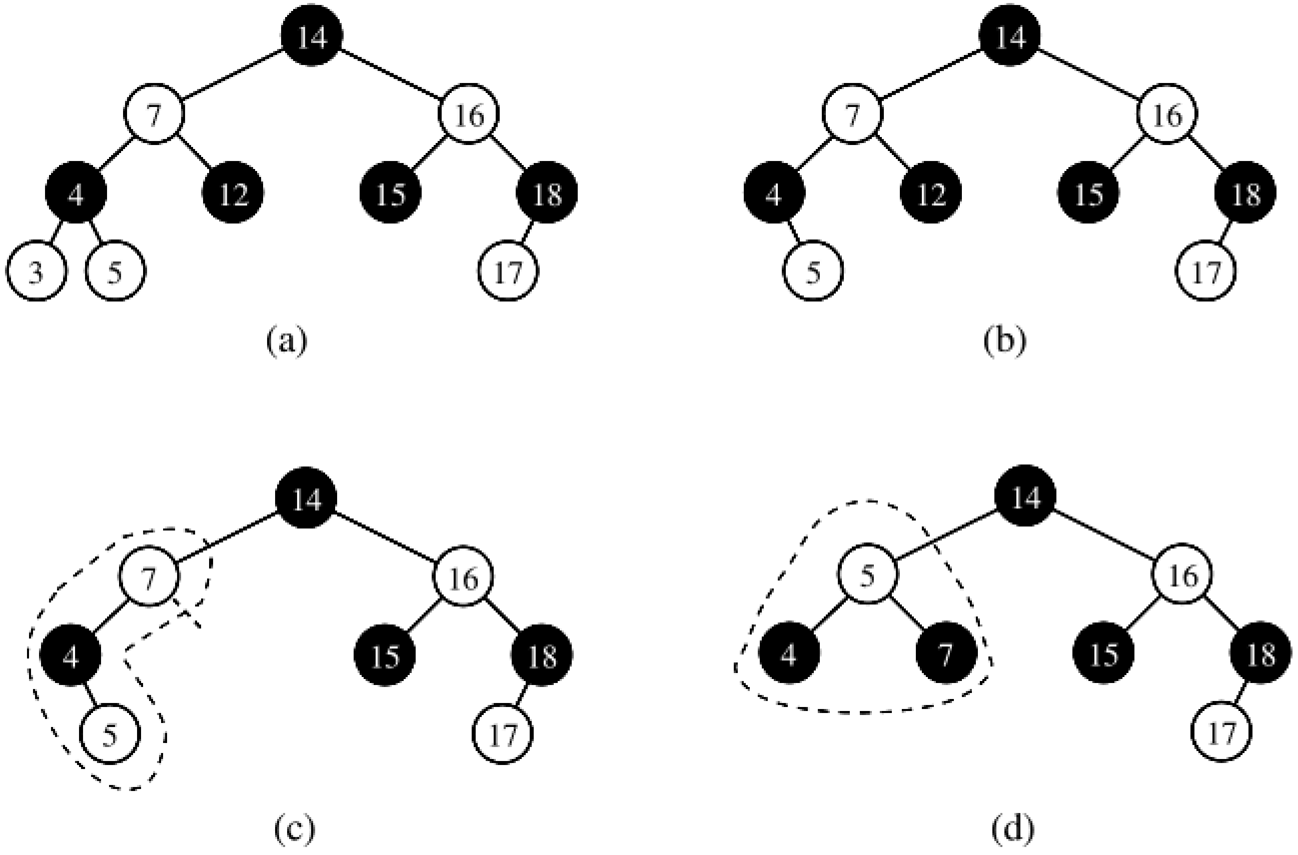


RED-BLACK TREES DELETION

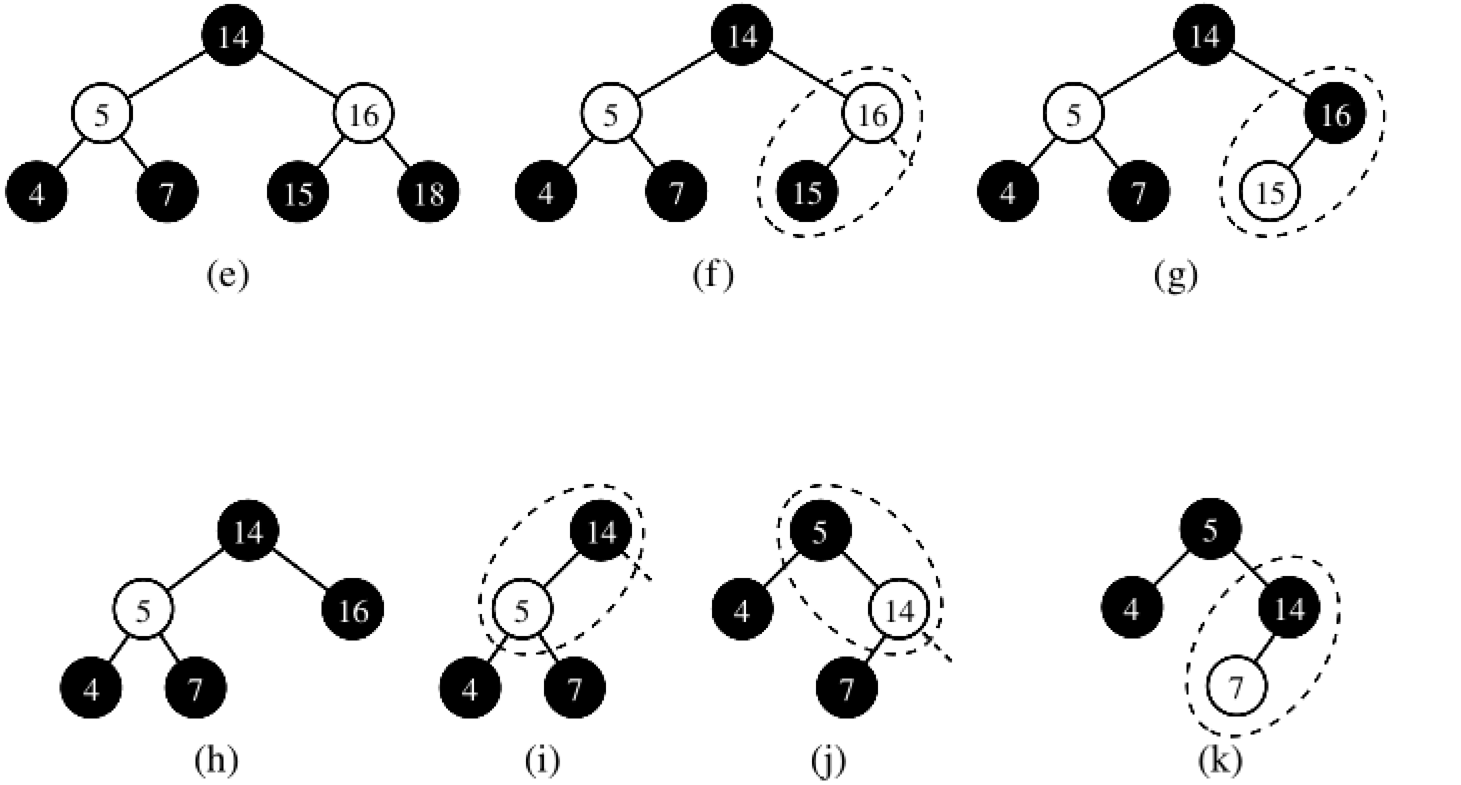
- Case 3: node y is red
- Rotation about y and z
- Recolor y black and z red
- Repeat step 1, 2 and 3 if necessary



RED-BLACK TREES
DELETION



RED-BLACK TREES DELETION



THIS LECTURE

- Sorting algorithms
  - Rearrange a collection of elements so that they are ordered from smallest to largest
  - Such an order exists in Python: the < operator

Irreflexive property: $k \not< k$.

Transitive property: if $k_1 < k_2$ and $k_2 < k_3$, then $k_1 < k_3$.

  - Most important and well studied computing problem
  - Data sets are often stored in sorted order to allow for efficient searches
    - E.g. binary search algorithm
  - Python: built-in support for sorting data
    - sort()

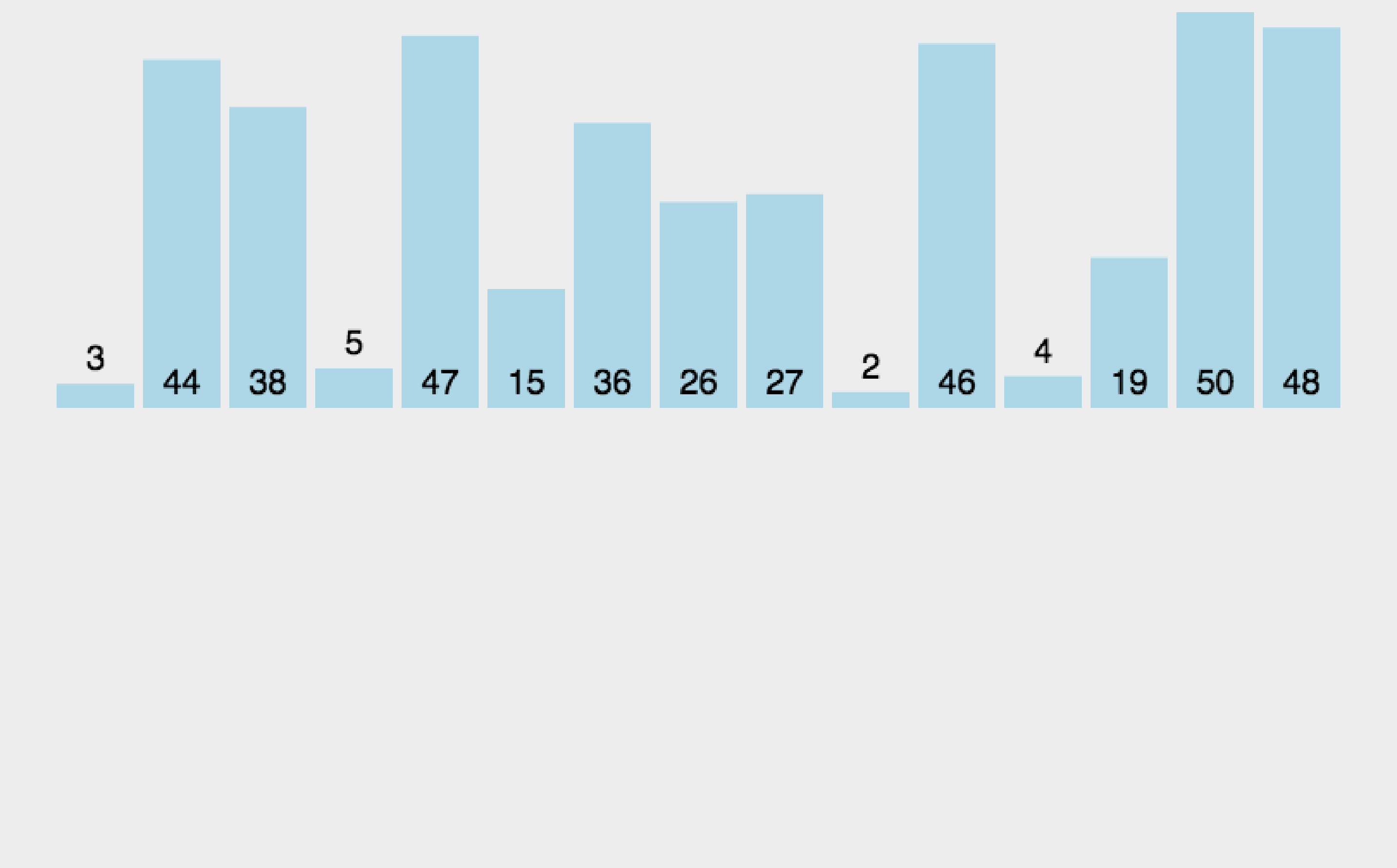
SORTING ALGORITHM

- Sorting algorithms we have seen so far
  - Insertion-sort
  - Selection-sort
  - Bubble-sort
  - Heap-sort
  - Merge-sort

INSERTION SORT

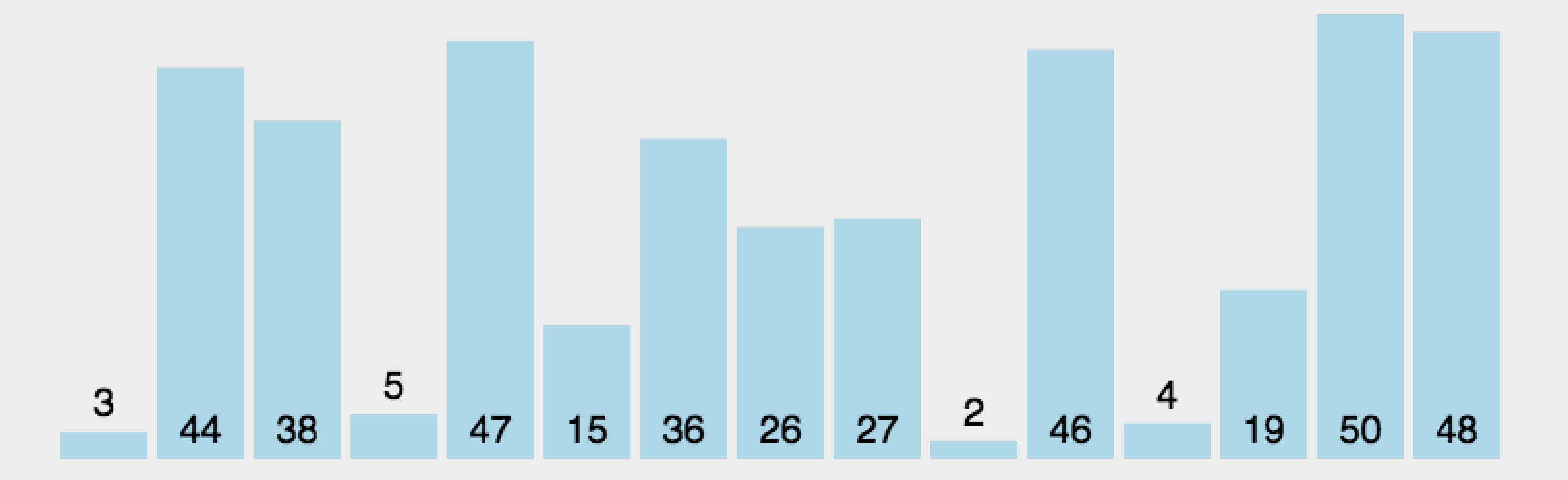
- Insertion sort

```
def insertion_sort(A):
  """Sort list of comparable elements
  for k in range(1, len(A)):
    cur = A[k]
    j = k
    while j > 0 and A[j−1] > cur:
      A[j] = A[j−1]
      j −= 1
    A[j] = cur
```



SELECTION SORT

- Selection sort
- We have seen this
  - In Priority Queue




```python
def selectionSort(arr):
    for i in range(len(arr) - 1):
        # 记录最小数的索引
        minIndex = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[minIndex]:
                minIndex = j
        # i 不是最小数时，将 i 和最小数进行交换
        if i != minIndex:
            arr[i], arr[minIndex] = arr[minIndex], arr[i]
    return arr
```

SORTING WITH A PRIORITY QUEUE

- Priority queue ADT: any type of object can be used as a key, as long as they can be compared with the comparison operator <
- Comparison operators need to be irreflexive and transitive

```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)          # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                    # store smallest remaining element in C
```

k3.

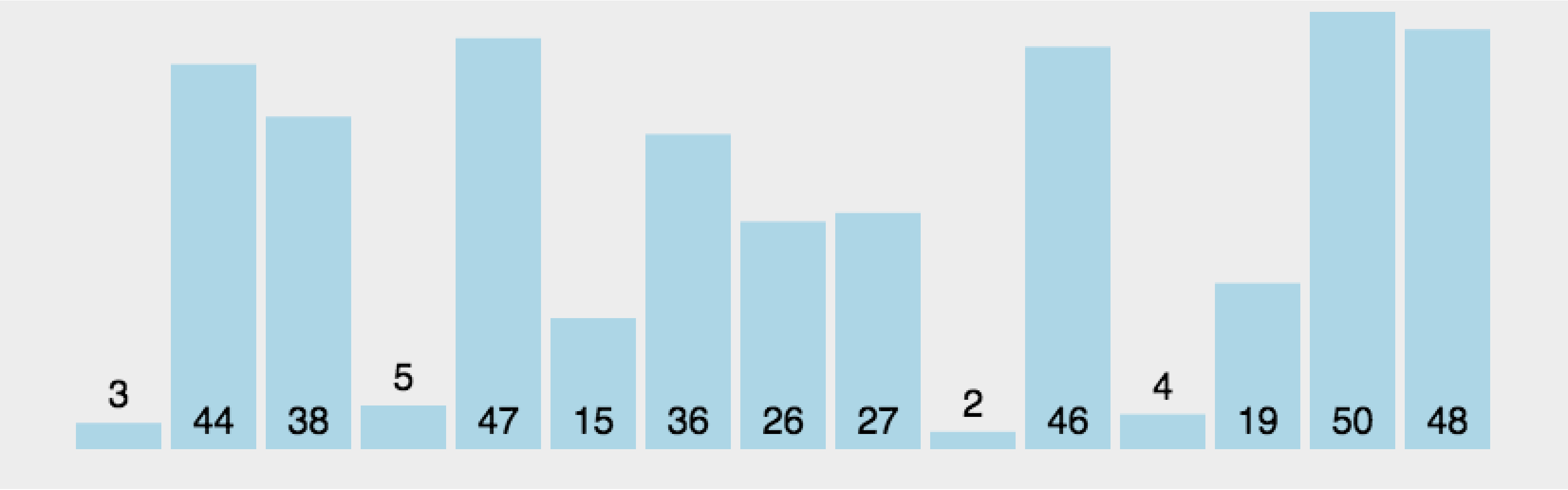
- Use priori … nents.
- Insert all e … nove_min to get an increc

SORTING WITH A PRIORITY QUEUE

- pq_sort(): works OK, but its complexity?
- Depends on add() and remove_min()
- Selection-Sort: implement P with an unsorted list
  - add() takes O(n) time in total since it is O(1) for add()
  - remove_min(): selecting element to dequeue()
  - Total running time: $O(n + (n-1) + (n-2) + \ldots + 1) = O(n^2)$
- Insertion-Sort: implement P with a sorted list
  - remove_min() takes O(n) time in total since it is O(1) for each remove_min()
  - add(): finding the proper place to add takes $O(n^2)$ time in total

BUBBLE SORT

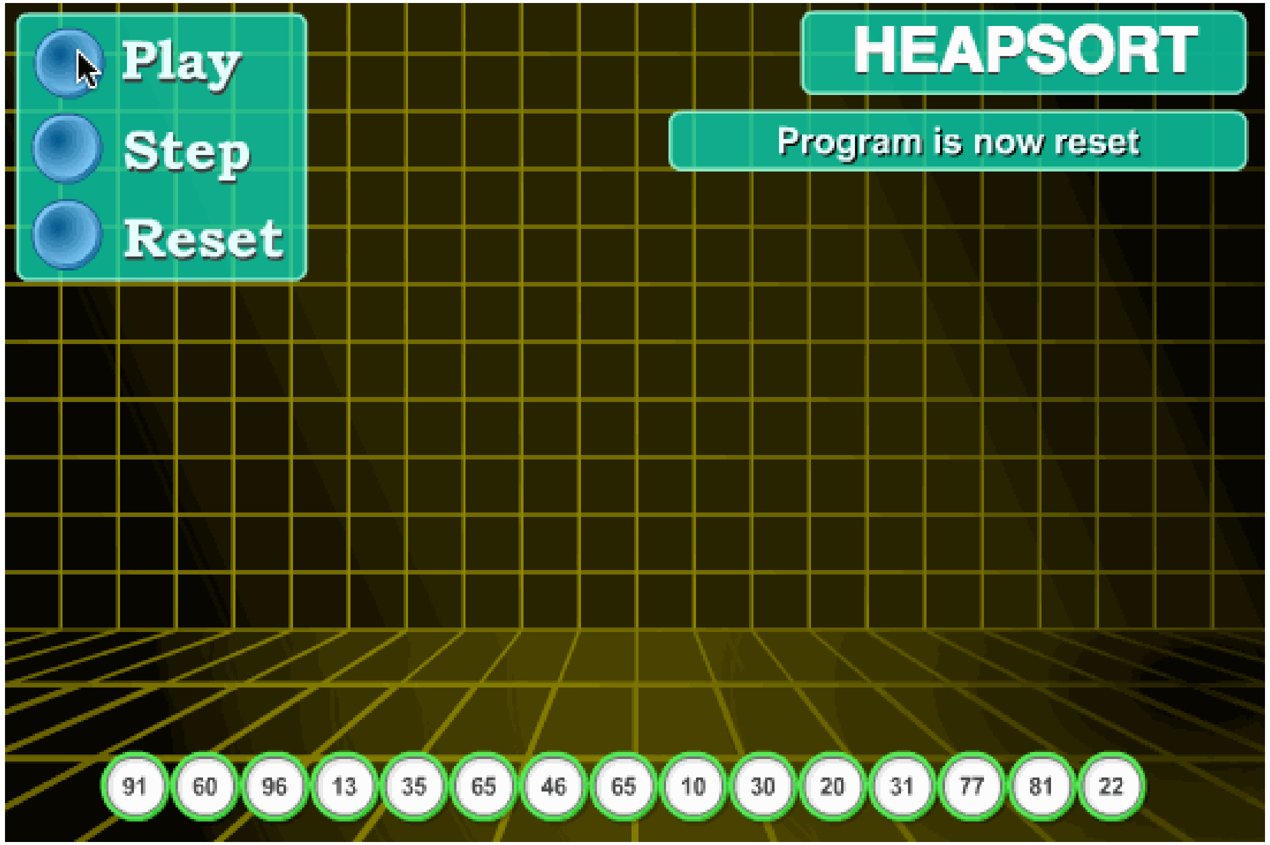
- Bubble sort




```python
def bubbleSort(arr):
    for i in range(1, len(arr)):
        for j in range(0, len(arr)-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

HEAP SORT

- Heap sort



HEAP（堆）

- Heap: a binary tree that stores a collection of items at its positions
  - A relational property defined in terms of the way keys are stored in T
  - A structural property defined in terms of the shape of T itself
- Relational property (**heap order property**): In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent
- Structural property (**complete binary tree property**): A heap T with height h is a complete binary tree if levels 0, 1, 2, …, h-1 of T have the maximum number of nodes possible (level i has 2^i nodes, for 0<= i <= h-1) and the remaining nodes at level h reside in the lfeftmost possible positions at that level

HEAP SORT

- Implementing in-place heap-sort （原地堆排序）
- Need to modify the algorithm
- Maximum-oriented heap: each position’s key being at least as large as its children. At any time during the execution, use the left portion of C, up to a certain index i-1, to store the entries of the heap, and the right portion of C, from i to n-1, to store the elements of the sequence
- In the first phase of the algorithm, start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time.
- In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time.

MERGE SORT

- Divide and conquer
  - **Divide**: if the input size is smaller than a certain threshold, solve the problem directly. Otherwise, divide the input data into two or more disjoint subsets
  - **Conquer**: recursively solve the sub-problems associated with the subsets
  - **Combine**: take the solutions to the sub-problems and merge them into a solution to the original problem

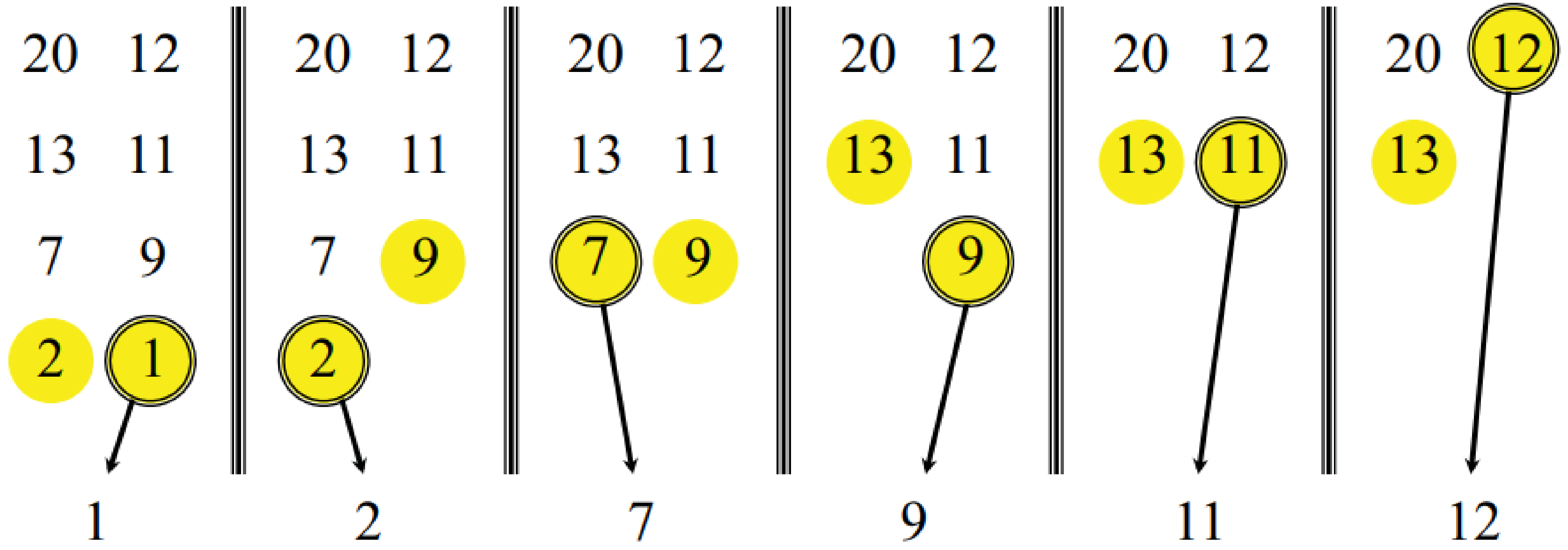
MERGE SORT

- Divide and conquer for sorting
  - **Divide**: if S has zero or one element, return S. Otherwise, remove all the elements from S and put them into two sequences, S1 and S2, each containing half of the elements of S
  - **Conquer**: recursively sort sequence S1 and S2
  - **Combine**: put back the elements into S by merging the sorted sequences S1 and S2 into a sorted sequence

MERGE SORT

- Merge sort

```
def merge(S1, S2, S):
  """Merge two sorted Python lists S1 and S2 into prop
  i = j = 0
  while i + j < len(S):
    if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
      S[i+j] = S1[i]                    # copy ith element
      i += 1
    else:
      S[i+j] = S2[j]                    # copy jth element
      j += 1
```



MERGE SORT

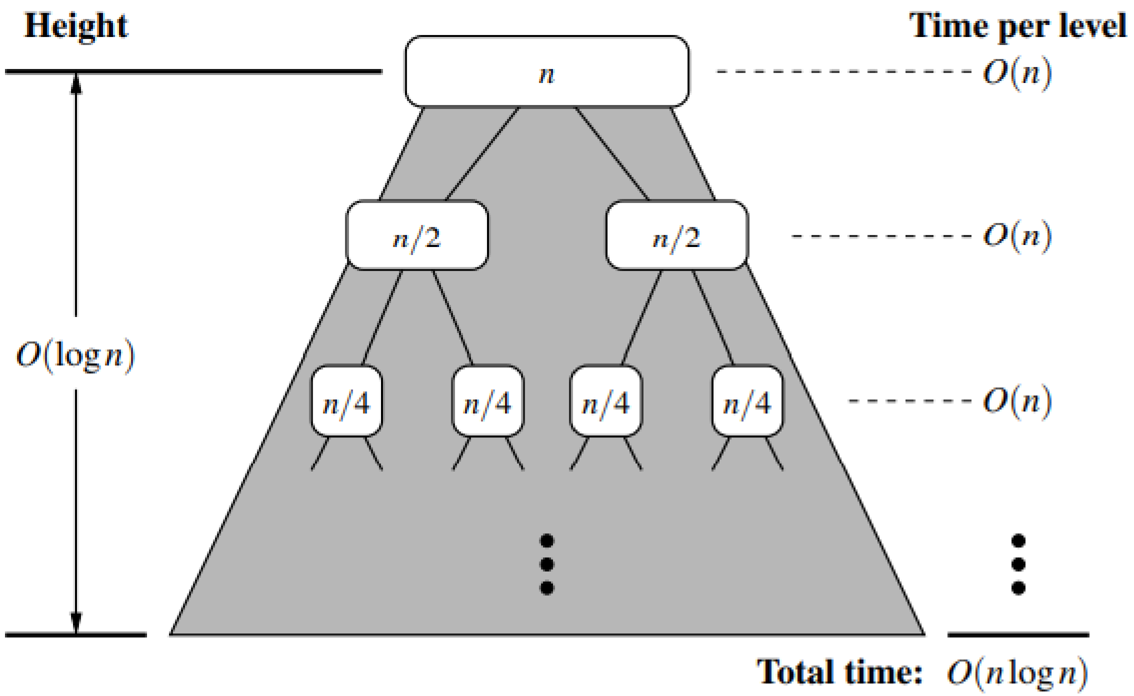
- Merge sort

```
def merge_sort(S):
  """Sort the elements of Python list S using the merge-sort algorithm."""
  n = len(S)
  if n < 2:
    return                        # list is already sorted
  # divide
  mid = n // 2
  S1 = S[0:mid]                   # copy of first half
  S2 = S[mid:n]                   # copy of second half
  # conquer (with recursion)
  merge_sort(S1)                  # sort copy of first half
  merge_sort(S2)                  # sort copy of second half
  # merge results
  merge(S1, S2, S)                # merge sorted halves back into S
```

MERGE SORT

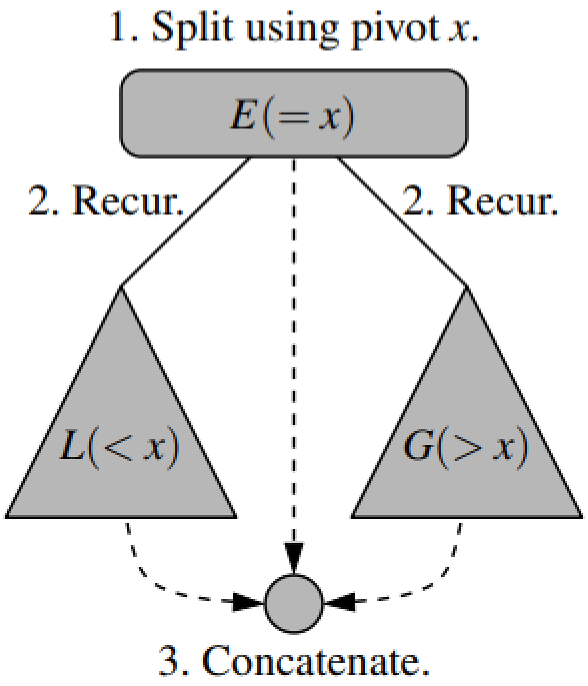
- merge_sort(A[1..n])

1. If n = 1, done
2. recursively sort A[1..n//2], A[n//2+1..n]
3. Merge the two sorted list

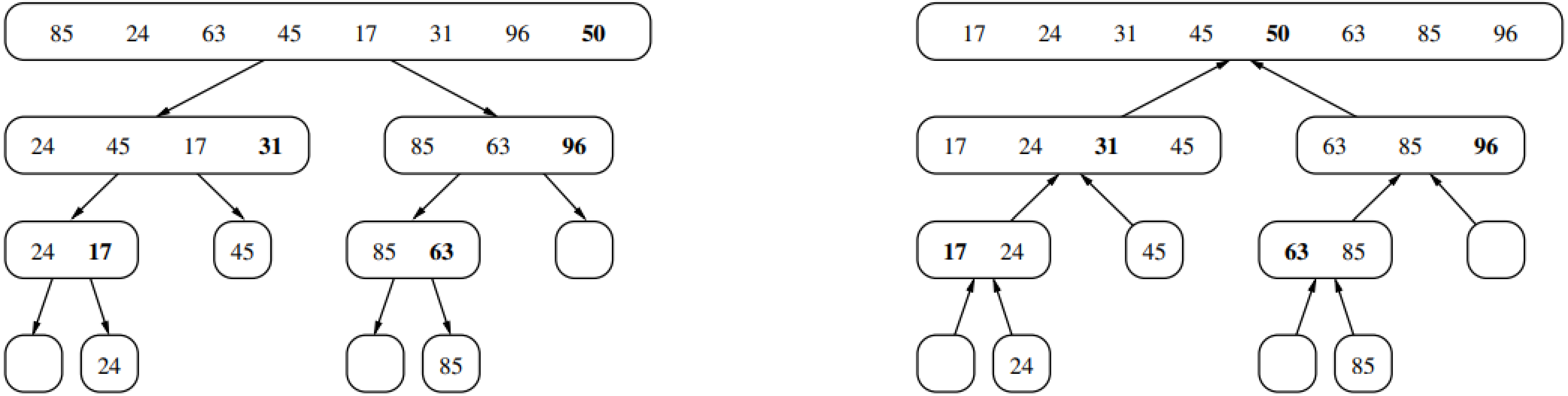


QUICK SORT

- **Divide**: if S has at least two elements, select a specific element x from S, which is called the pivot(枢纽). The common practice is to choose the last element of S. Remove all the elements from S and put them into three sequences:
  - L, storing elements in S less than x
  - E, storing elements in S equal to x
  - G, storing elements in S greater than x
- **Conquer**: recursively sort sequences L and G
- **Combine**: Put back the elements into S in order by first inserting the elements of L, then E, then G

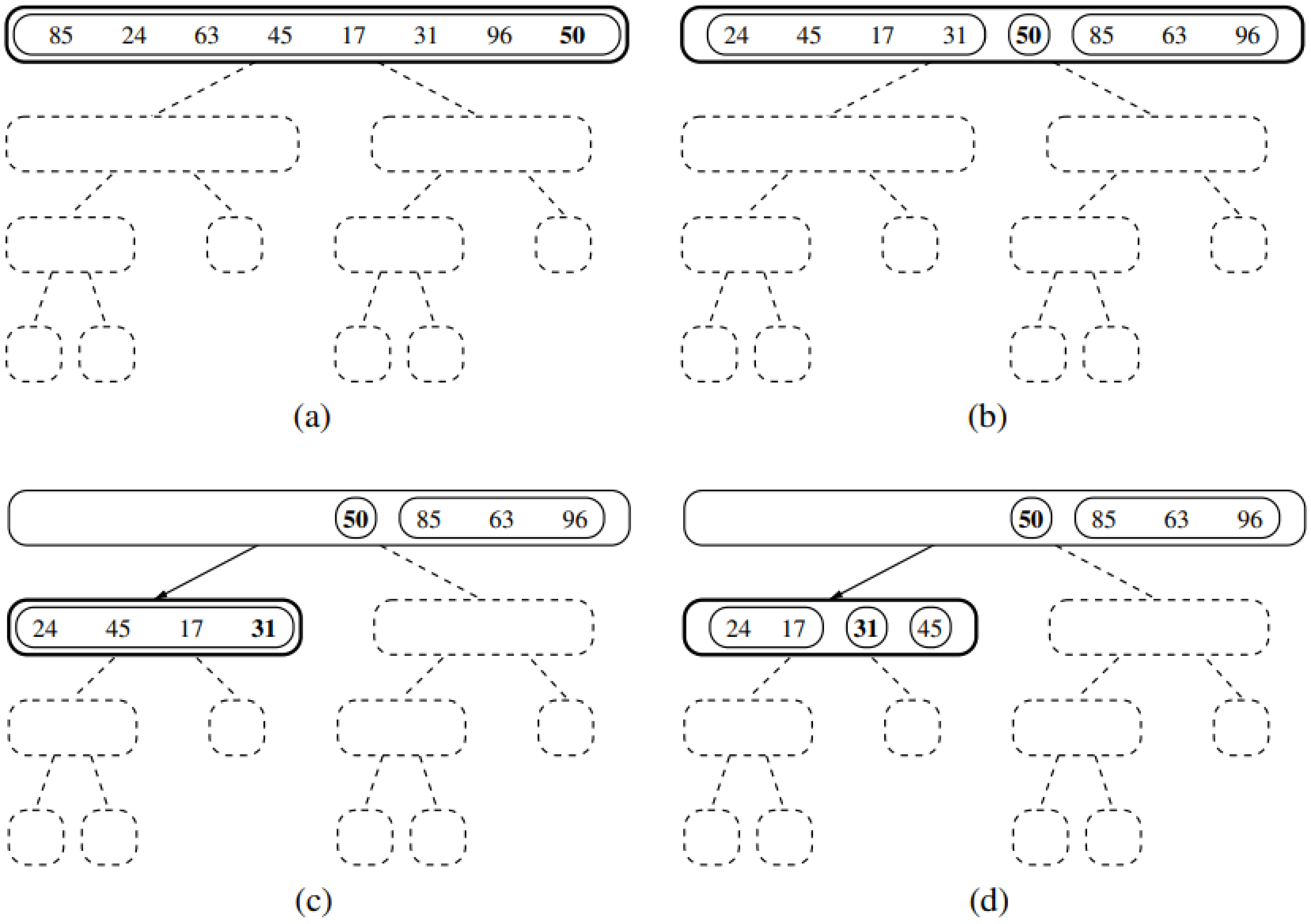


QUICK SORT

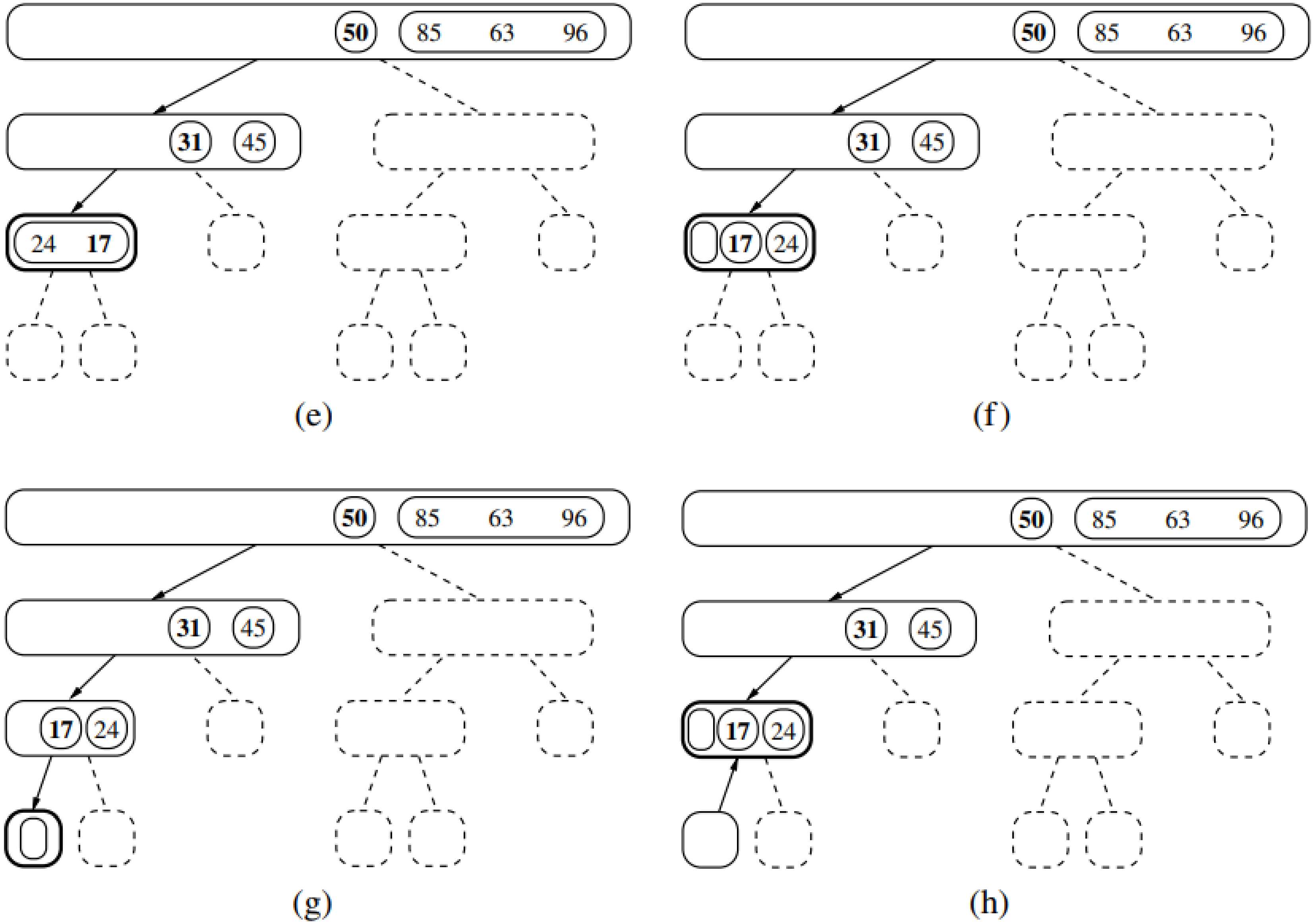


(a) (b)

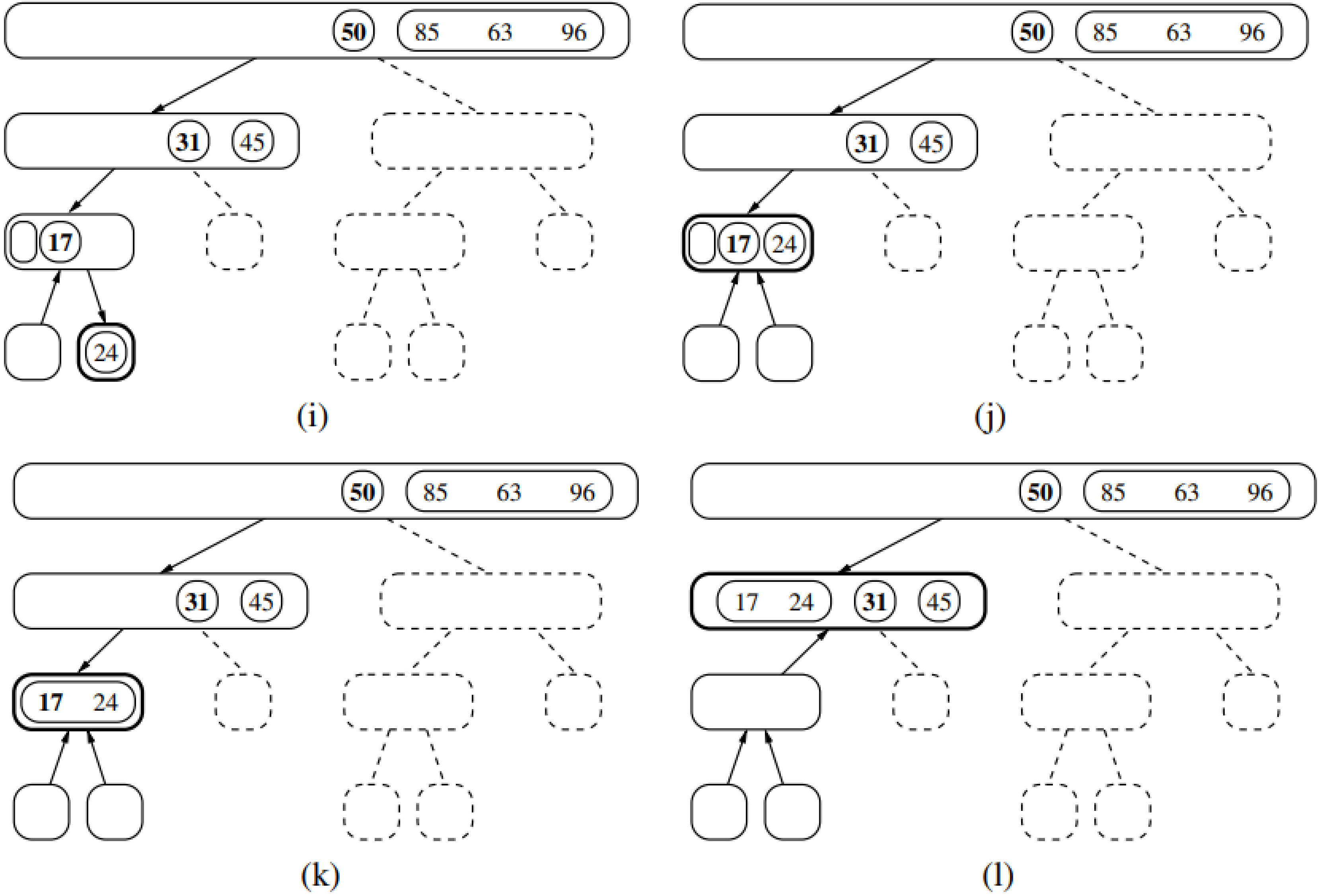
QUICK SORT


85 24 63 45 17 31 96 50
(a)
24 45 17 31 50 85 63 96
(b)
50 85 63 96
24 45 17 31
(c)
50 85 63 96
24 17 31 45
(d)

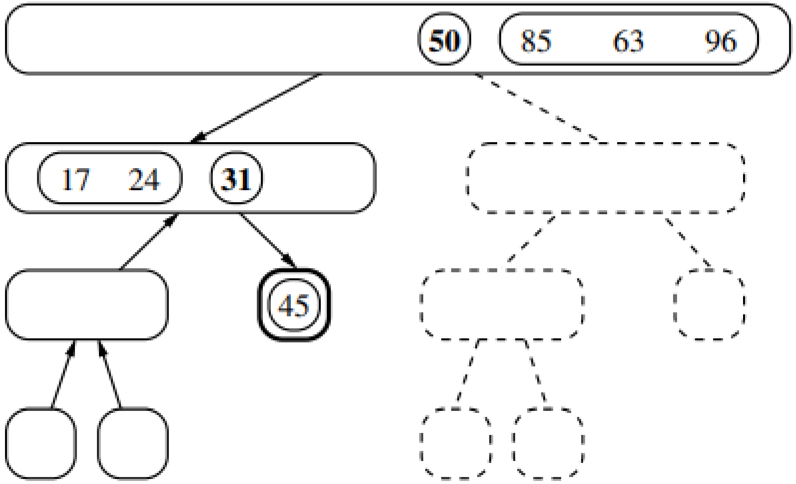
QUICK SORT


50
85 63 96
31
45
24 17
(e)
50
85 63 96
31
45
17
24
(f)
50
85 63 96
31
45
17
24
(g)
50
85 63 96
31
45
17
24
(h)

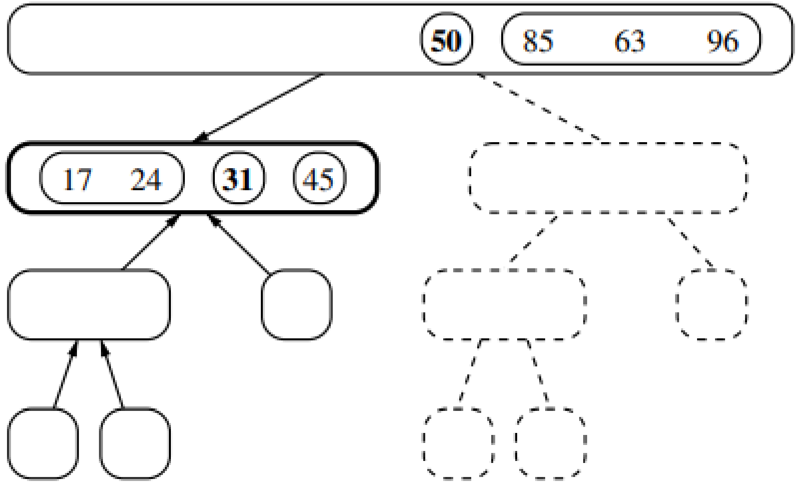
QUICK SORT


50
85 63 96
31
45
17
24
(i)
50
85 63 96
31
45
17
24
(j)
50
85 63 96
31
45
17 24
(k)
50
85 63 96
17 24
31
45
(l)

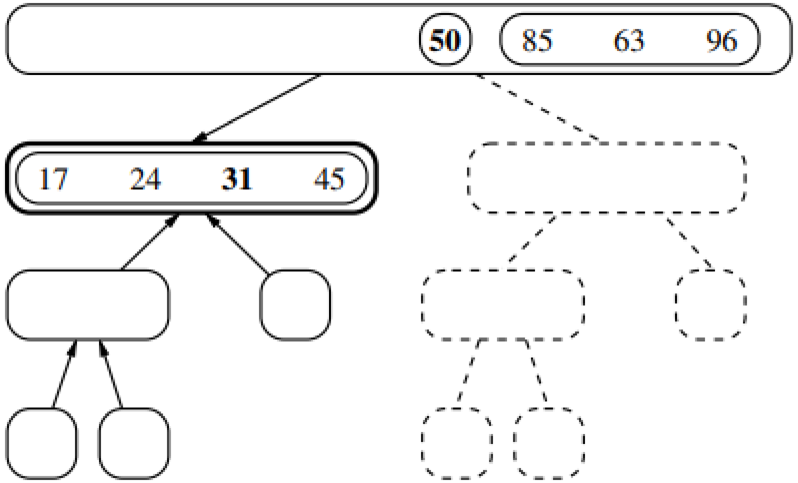
QUICK SORT



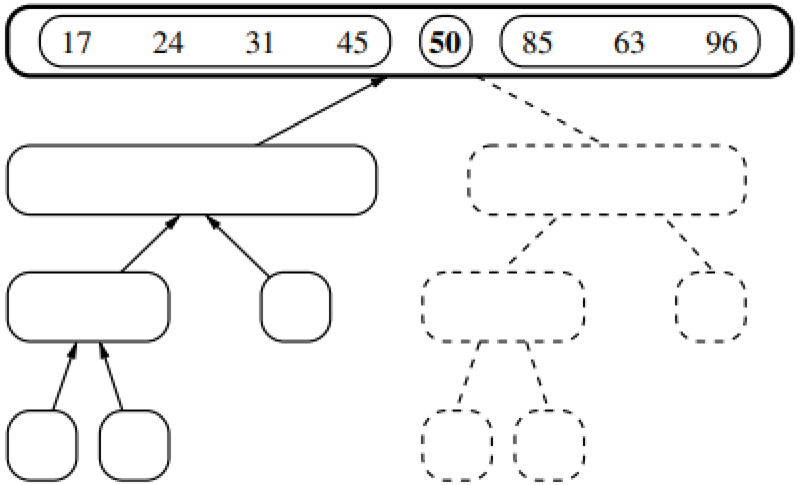
(m)



(n)

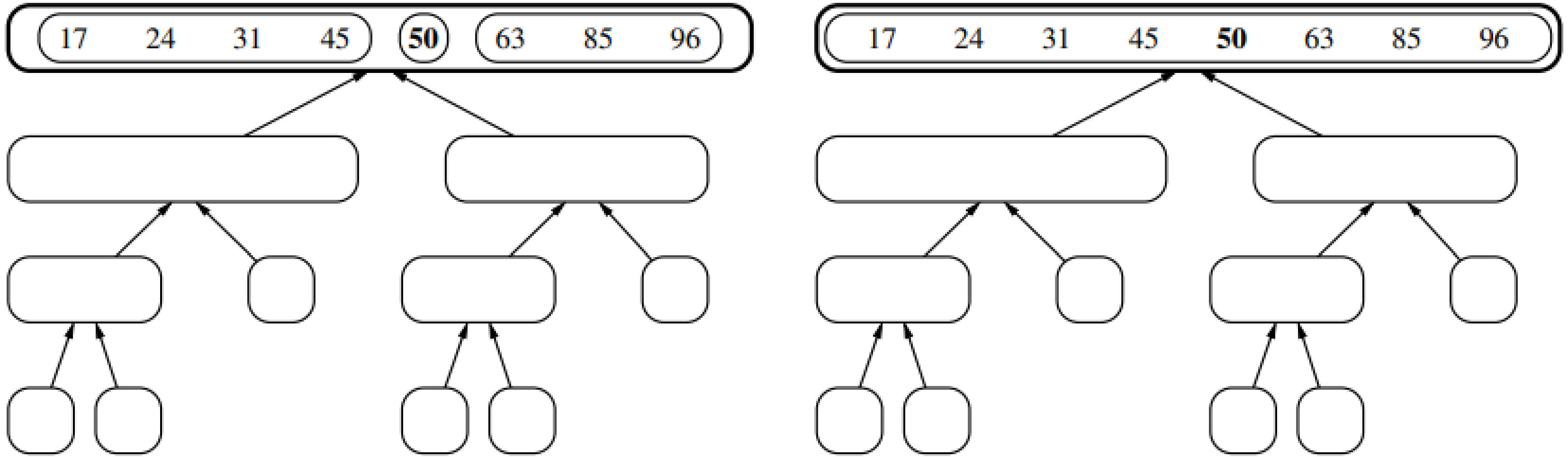


(o)



(p)

QUICK SORT



(q) (r)

- Height of the quick-sort tree: linear in the worst case
  - Why?
- For an already-sorted sequence of n elements
  - Height = n-1
- Running time:
  - Worst case: $O(n^2)$
  - Best case: $O(n \log n)$
- Selection of Pivot: last element of S
- Improved selection method: pick Pivot at random in S
  - Expected running time: $O(n \log n)$

```
def quick_sort(S):
  """Sort the elements of queue S using the quick-sort algorithm."""
  n = len(S)
  if n < 2:
    return                                   # list is already sorted
  # divide
  p = S.first( )                             # using first as arbitrary pivot
  L = LinkedQueue( )
  E = LinkedQueue( )
  G = LinkedQueue( )
  while not S.is_empty( ):                   # divide S into L, E, and G
    if S.first( ) < p:
      L.enqueue(S.dequeue( ))
    elif p < S.first( ):
      G.enqueue(S.dequeue( ))
    else:                                    # S.first() must equal pivot
      E.enqueue(S.dequeue( ))
  # conquer (with recursion)
  quick_sort(L)                              # sort elements less than p
  quick_sort(G)                              # sort elements greater than p
  # concatenate results
  while not L.is_empty( ):
    S.enqueue(L.dequeue( ))
  while not E.is_empty( ):
    S.enqueue(E.dequeue( ))
  while not G.is_empty( ):
    S.enqueue(G.dequeue( ))
```

THANKS

See you in the next session!