

15CSE201 : Data Structures and Algorithms

Application of Stacks Ritwik M

Based on the reference materials by Prof. Goodrich, OCW METU, Dr. George Bebis and Dr. Vidhya Balasubramanian

Application of Stacks

- A Recap – Stacks, Amortization
- Recursion
- Tower of Hanoi
- Evaluation of Expressions

The Stack Interface: Python

```
class MyStack():  
    def push(self, value): //pushes the value into the stack  
    def pop(self): //returns top element of stack if not empty, else  
        throws exception  
    def top(self): //returns top element without removing it if the stack  
        is not empty, else throws exception  
    def size(self): //returns the number of elements currently in stack  
    def isEmpty(self): //returns True if stack is empty
```

Amortization

- Amortized running time =
$$\frac{(\text{worstcaserunningtimeofaseriesofoperations})}{(\text{totalnumberofoperations } (n))}$$
- Time Complexity
 - Size() - $O(1)$
 - IsEmpty()- $O(1)$
 - top- $O(1)$
 - Push - $O(1)$
 - Pop - $O(1)$

Increasing Array Size

- Size of the new array
 - Increasing strategy
 - Increase size by constant c
 - Doubling strategy
 - Double the size
- Comparison
 - Use Amortization Analysis
 - Analyze total time $t(n)$ needed to perform a series of push operations
 - Assume stack is empty and represented with array of size 1
 - Calculate amortized time of push = $t(n)/n$

Incremental Strategy Analysis

- We replace the array k times where $k = n/c$
- The total time $T(n)$ of a series of n push operations is proportional to
 - $n + c + 2c + 3c + 4c + \dots + kc =$
 - $n + c(1 + 2 + 3 + \dots + k) =$
 - $n + ck(k + 1)/2$
- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e.,
 - $O(n^2)$
- The amortized time of a push operation is $O(n)$

Doubling Strategy Analysis

We replace the array $k = \log_2 n$ times

The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

$$n + 2^{k+1} - 1 = 2n - 1$$

$T(n)$ is $O(n)$

The amortized time of a push operation is $O(1)$

Application of Stacks

- Recursion
- Tower of Hanoi
- Evaluation of Expressions

Recursion

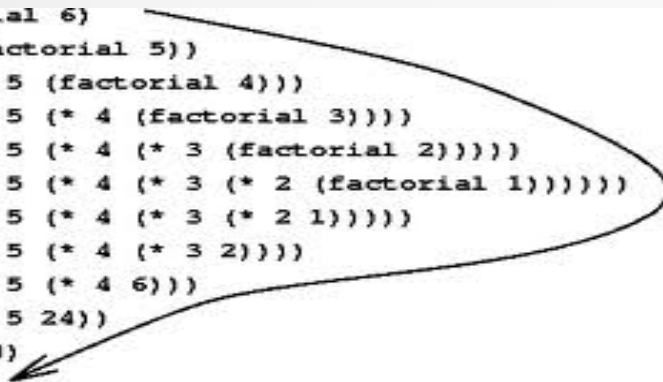
- Concept of defining a function that calls itself as a sub-routine
 - Allows us to take advantage of the repeated structure in many problems
 - e.g finding the factorial of a number



Linear Recursion

- Function is defined so that it makes atmost one recursive call at each time it is invoked
- Useful when an algorithmic problem
 - Can be viewed in terms of a first or last element, plus a remaining set with same structure

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```



Src:mitpress.mit.edu

Algorithm LinearSum(A, n):

Input: Integer array A and element n

Output: Sum of first n elements of A

if $n=1$ then

 return $A[0]$

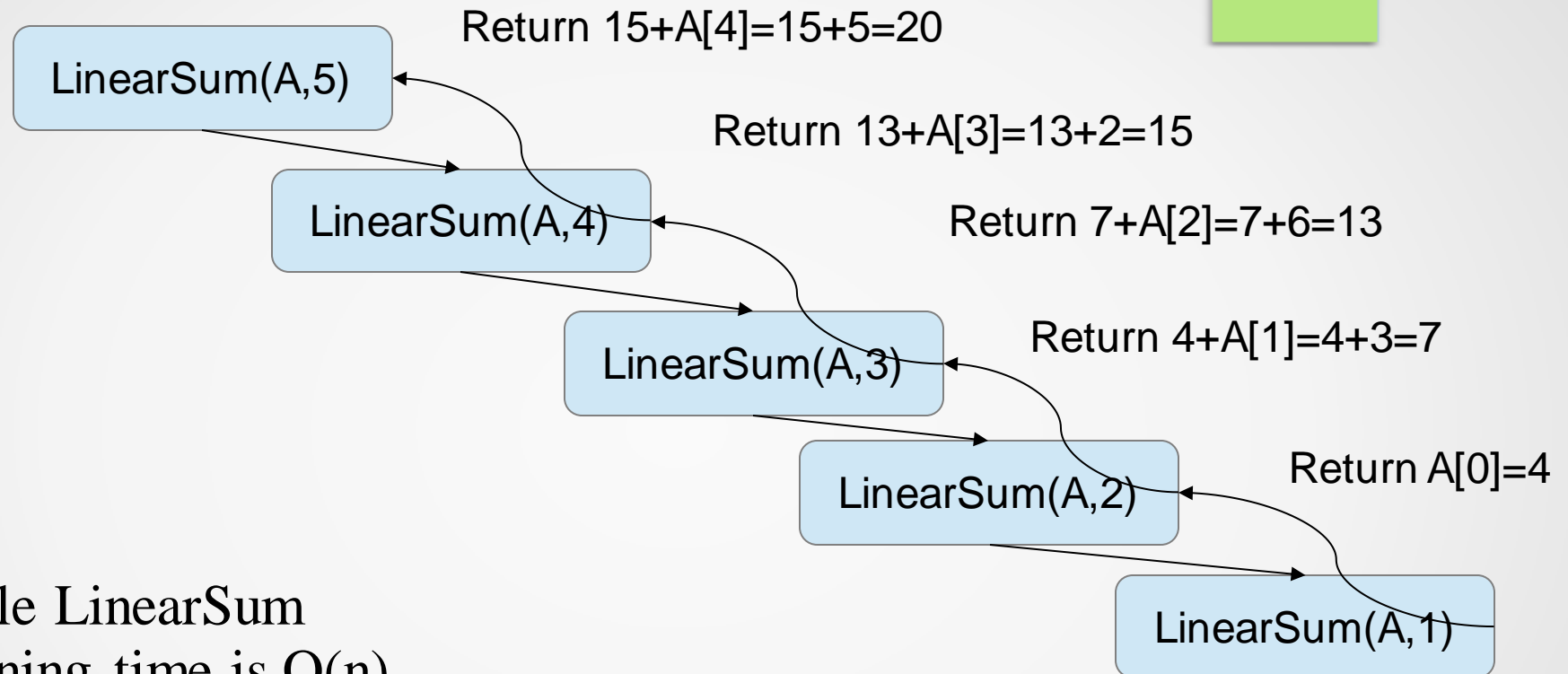
else

 return LinearSum($A, n-1$)+ $A[n-1]$

Linear Recursion

- Algorithm using linear recursion uses the following
 - Test for base cases
 - Base cases defined so that every possible chain of recursive call reaches base case
 - Helps in termination
 - Recurse
 - May decide on one of multiple recursive calls to make
 - Recursive calls must progress to base case

Analyzing Recursive Algorithms



•Example LinearSum

- Running time is $O(n)$
- Space Complexity: $O(n)$

Stack Trace

- Example stack trace

LinearSum(A,1)	Return $A[0]=4$
LinearSum(A,1)+A[1]	Return $4+A[1]=4+3=7$
LinearSum(A,2)+A[2]	Return $7+A[2]=7+6=13$
LinearSum(A,3)+A[3]	
LinearSum(A,4)+A[4]	

Problem 1

- Write a simple recursive function to print n elements.

Problem 2

- Describe a linear recursive algorithm for finding the minimum element in an n-element array

Problem 3

- Computing Powers via Linear Recursion

$$power(x, n) = \begin{cases} 1 & \text{if } n=0 \\ x \cdot power(x, n-1) & \text{otherwise} \end{cases}$$

$$power(x, n) = \begin{cases} 1 & \text{if } n=0 \\ x \cdot power(x, (n-1)/2)^2 & \text{if } n>0 \text{ is odd} \\ power(x, n/2)^2 & \text{if } n>0 \text{ is even} \end{cases}$$

Problem 4

- Reverse and array using linear recursion
- Solution

– **Algorithm** ReverseArray(A, i, n):

Input: Integer array A and integers i, n

Output: Reversal of n integers in A starting from i

if $n=1$ **then**

 Swap $A[i]$ and $A[i+n-1]$

 Call ReverseArray($A, i+1, n-2$)

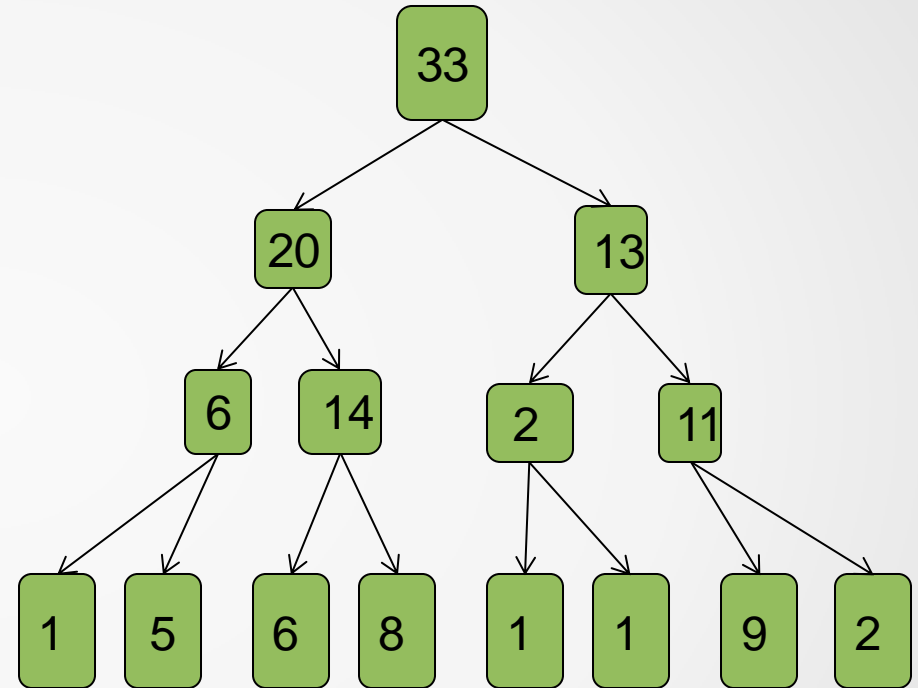
return

Higher-Order Recursion

- Uses more than one recursion call
 - e.g 3-way merge sort
- Binary recursion
 - Two recursion calls
 - e.g BinarySum
 - **Algorithm** BinarySum(A,i,n):
 - **Input:** Integer array A and integers i, n
 - **Output:** Sum of first n elements of A starting at index i
 - **if** $n=1$ then
 - **return** A[i]
 - **else**
 - **return** BinarySum(A,i,[n/2])+BinarySum(A,[n/2],[n/2])

Analysis

- Recursion trace is a tree
- Depth of recursion
 - $O(\log n)$
 - Lesser additional space needed
- Running time
 - $O(n)$
 - Have to visit every element in the array



Problem - 1

Generate the kth Fibonacci number using Binary Recursion

Solution

Algorithm BinaryFib(k):

Input: Integer array A

Output: k^{th} Fibonacci Number

if $k \leq 1$ **then**

return k

else

return BinaryFib($k-1$)+BinaryFib($k-2$)

Problem - 2

Describe a binary recursive method for searching an element x in an n -element unsorted array A .

Compute the running time and space complexity of your algorithm

Implementing Recursion using a stack

- Stacks are used to store
 - Function calls
 - Parameters in the functions, and value to be returned
- Each time a function is called it is pushed into a stack
- Each pop() returns the corresponding value
 - The top element in the complete stack is the base value

Towers of Hanoi

- Also known as 'Tower of Brahma'
- According to an old story in one of the temples, the existence of the universe is calculated in terms of the time taken by a number of priest, who are working all the time, to move 64 disks from one pole to another in the same order. The total number of steps is said to be the time of one cycle of Brahma.



- Rules:

- move only one disk at a time
- for temporary storage, a third pole may be used
- a disk of larger diameter may not be placed on a disk of smaller diameter

Source: https://upload.wikimedia.org/wikipedia/commons/6/60/Tower_of_Hanoi_4.gif

Tower of Hanoi

Use recursion to solve this

Algorithm

```
MoveTower(n,src,dest,spare)
```

```
  if n=0
```

```
    move disk from src to dest
```

```
  else
```

```
    MoveTower(n-1, src,spare,dest);
```

```
    //move n-1 discs from source to  
    spare
```

```
    move disc n from src to dest
```

```
    //move n-1 discs from spare to dest
```

```
    MoveTower(n-1, spare,dest,source);
```



Minimum number of moves

$$2^N - 1$$

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

Evaluation of Expressions

- Arithmetic expressions are written in the following styles
 - Infix notation
 - Common notation
 - Operators written between operands they act on
 - e.g $A + B$
 - Prefix notation
 - Operands follow the operator
 - e.g $+AB$
 - Postfix notation
 - Operator follows operands
 - e.g $AB+$

Postfix Notation

- To correctly evaluate expressions the order is essential
 - Parenthesization is needed in many cases
 - Operators in Postfix notation are always in the correct evaluation order
- e.g.
 - $5 * 3 + 2 + 6 * 4$
 - Postfix notation is:
 - $5 3 * 2 + 6 4 * +$
 - Eq to $((5*3)+2)+(6*4)$

Converting Infix to Postfix

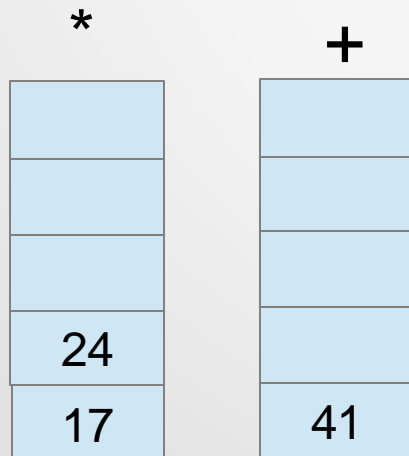
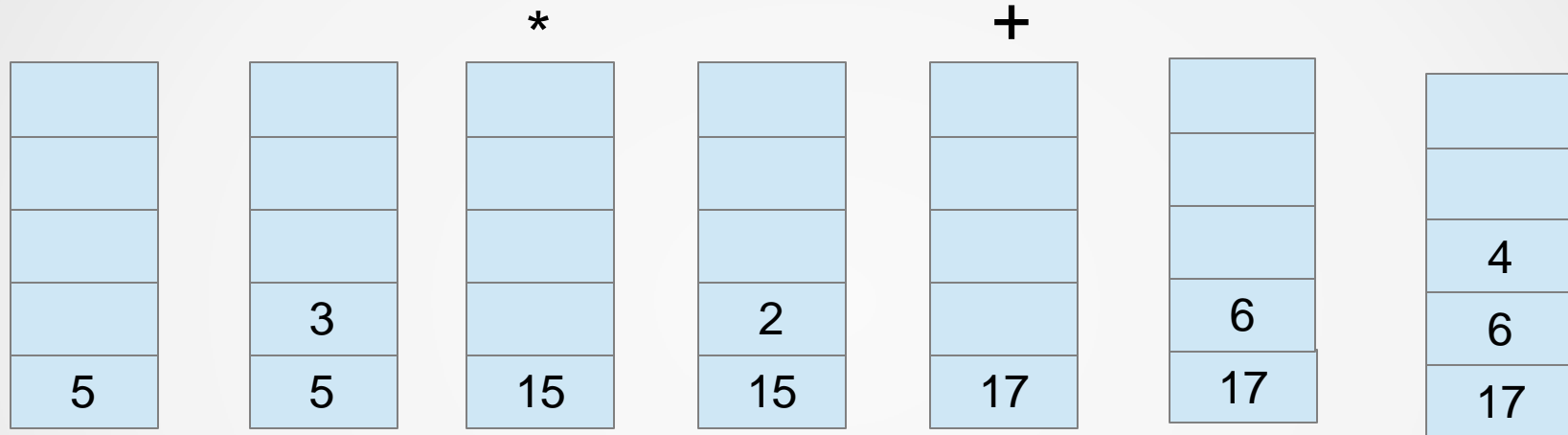
- Precedence of * and / higher than + and -
 - For same precedence, use left associativity
- e.g $a + b * c$
 - Parenthesize it $\Rightarrow a + (b * c)$
 - Convert the multiplication $\Rightarrow a + (b c *)$
 - convert the addition $\Rightarrow a(bc*)+$
 - Remove parenthesis $\Rightarrow a b c * +$
- Find the postfix form of $a + ((b * c) / d)$

Evaluating a Postfix using stack

- Each operator in a postfix string corresponds to the previous two operands
- Each time we read an operand we push it onto a stack
- When an operator is reached, its associated operands (the top two elements on the stack) are popped from the stack
- Perform the operation and push the result on top of the stack
 - available for use as one of the operands for the next operator
- Process stops when there are no more operators

Example

5 3 * 2 + 6 4 * +

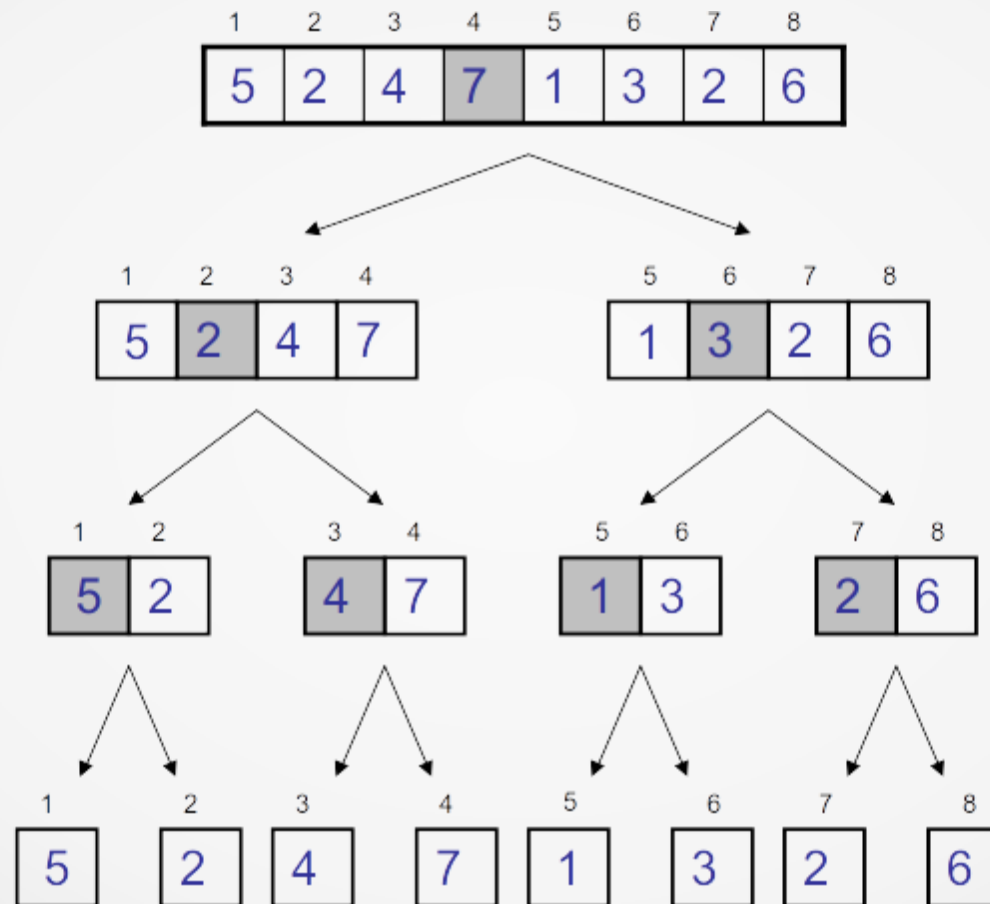


Merge Sort

- Uses multiple recursion(divide and conquer strategy) to sort a set of numbers
 1. Divide:
 - If S has zero or one element, return S
 - Divide S into two sequences S_1 and S_2 each containing half of the elements of S
 2. Recur
 - Recursively apply merge sort to S_1 and S_2
 3. Conquer
 - Merge S_1 and S_2 into a sorted sequence

Example-1

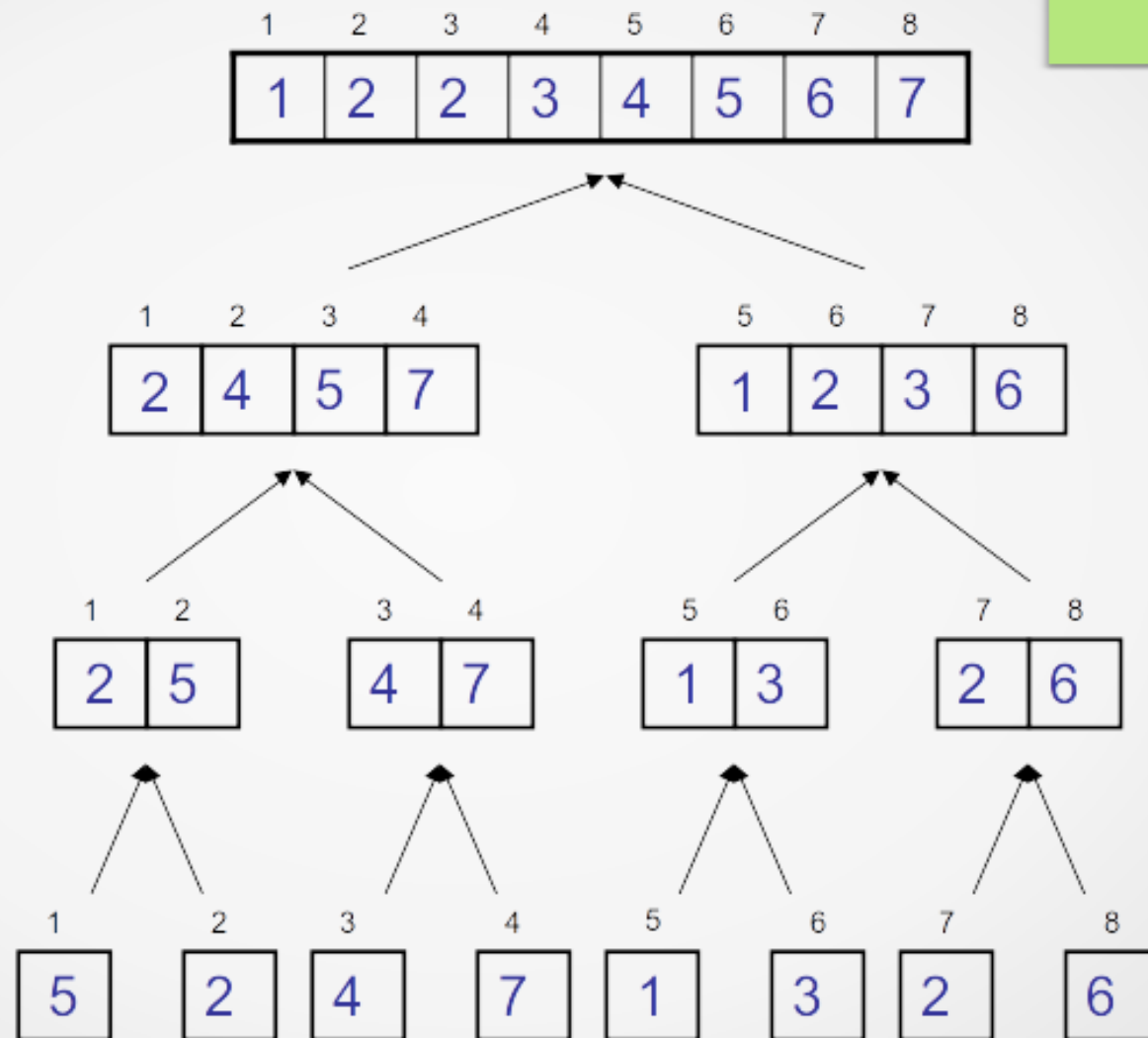
Divide



$q = 4$

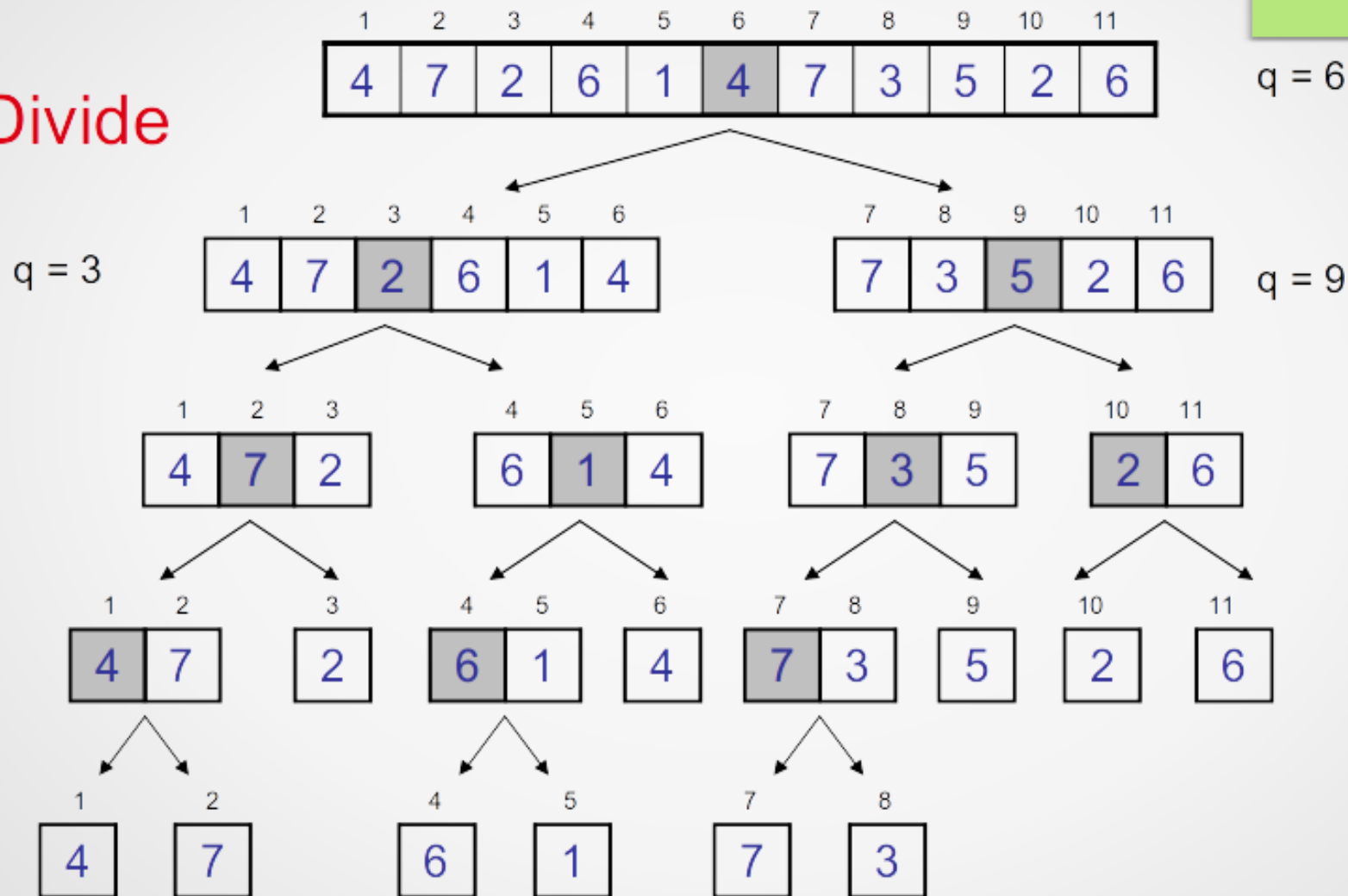
Example 1- Contd..

Conquer
and
Merge



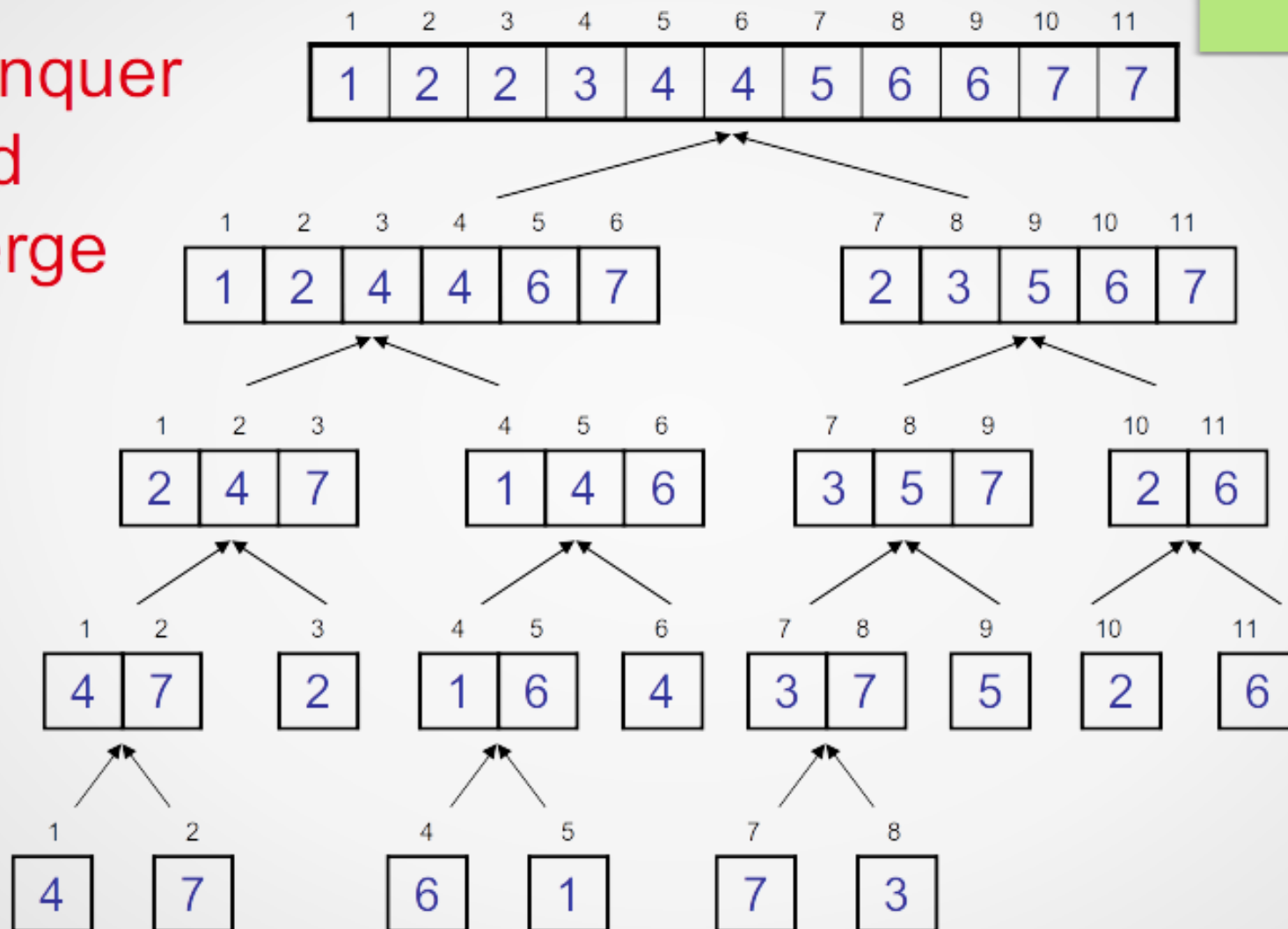
Example 2

Divide



Example 2 Contd...

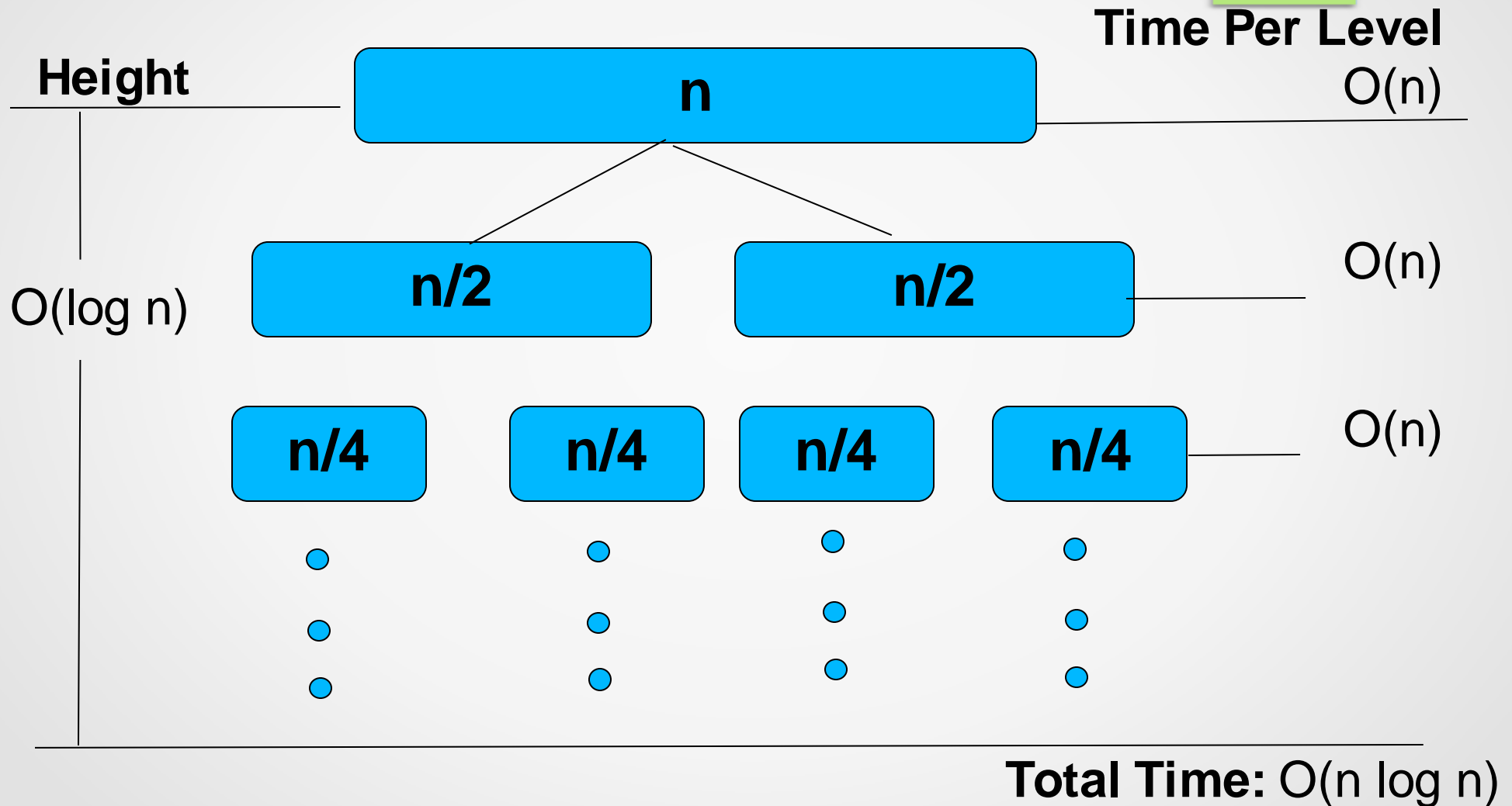
Conquer
and
Merge



Algorithm MergeSort()

```
mergesort(S, low, high) {  
    if (low < high) {  
        middle = (low+high)/2;  
        mergesort(s,low,middle);  
        mergesort(s,middle+1,high);  
        merge(s, low, middle, high);  
    }  
}
```

Analysis of Merge Sort



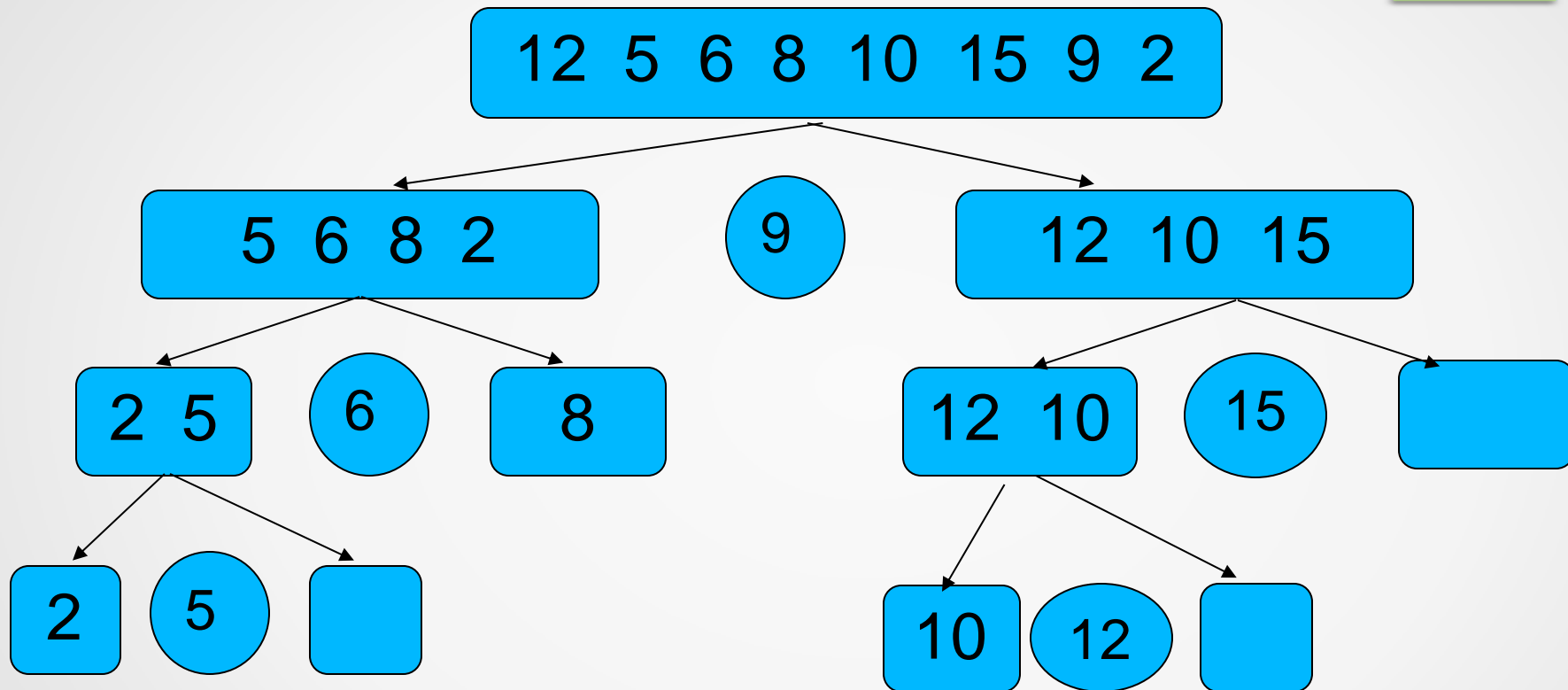
Analysis of Merge Sort Contd...

- Work done on the k th level involves merging 2^k pairs sorted list, each of size $n/(2^{k+1})$
 - A total of atmost n i.e. 2^k comparisons
- Linear time for merging at each level
 - Each of the n elements appear exactly in one of the subproblems at each level
- Requires extra memory

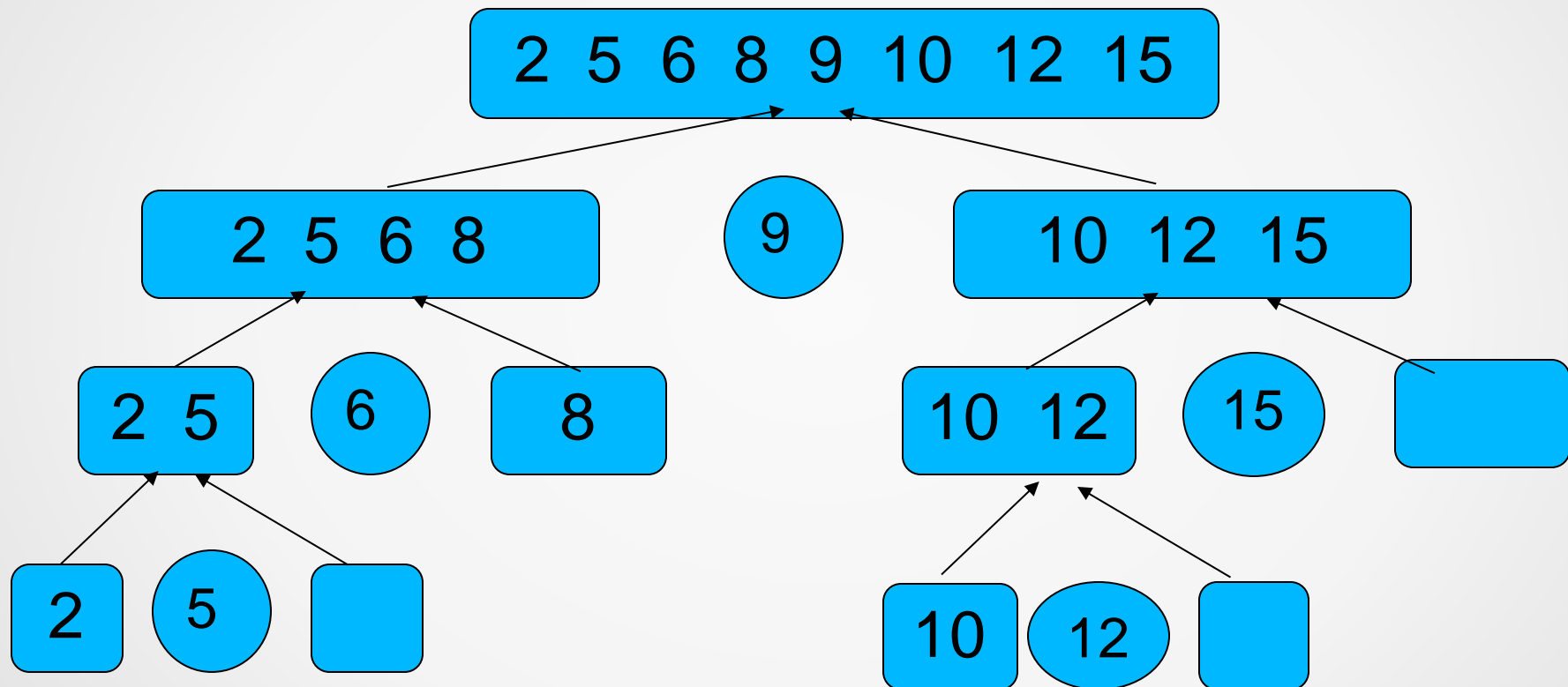
Quick Sort

- A divide and conquer strategy which also uses randomization
- Divide
 - Select a random pivot p . Divide S into two subarrays, where one contains elements $< p$ and the other which are $> p$
- Recur
 - Sort subarrays by recursively applying quicksort on each subarray
- Conquer
 - Since the subarrays are already sorted, no work is needed in merge part

Example



Example Contd



Algorithm QuickSort()

- QUICKSORT(A, p, r)
 - if $p < r$
 - $q = \text{PARTITION}(A, p, r)$
 - QUICKSORT(A, p, $q-1$)
 - QUICKSORT(A, $q+1$, r)
- Selection of pivot
 - Can be first or last element (here)
 - Median element
 - Random element

Quick Sort Analysis

- Worst Case Running Time
 - Occurs when pivot is always the largest element
 - Selecting the first element or last element as pivot causes this problem when list is already sorted
 - Running time proportional to $n+(n-1)+(n-2)+\dots+1$
 - $O(n^2)$

Implementing Recursion using a stack

- Stacks are used to store
 - Function calls
 - Parameters in the functions, and value to be returned
- Each time a function is called it is pushed into a stack
- Each pop() returns the corresponding value
 - The top element in the complete stack is the base value

Stacks - Exercises

1. What values are returned during the following series of stack operations, if executed upon an initially empty stack?
push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().
2. Suppose an initially empty stack S has executed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which raised Empty errors that were caught and ignored. What is the current size of S?
3. Give a recursive method for removing all the elements from a stack.
4. Implement a function that reverses a list of elements by pushing them onto a stack in one order, and writing them back to the list in reversed order

More Exercises

5. Suppose Alice has picked three distinct integers and placed them into a stack S in random order. Write a short, straight-line piece of pseudo-code (with no loops or recursion) that uses only one comparison and only one variable x , yet that results in variable x storing the largest of Alice's three integers with probability $2/3$. Argue why your method is correct.
6. Modify the `ArrayStack` implementation so that the stack's capacity is limited to `maxlen` elements, where `maxlen` is an optional parameter to the constructor (that defaults to `None`). If `push` is called when the stack is at full capacity, throw a `Full` exception (defined similarly to `Empty`).
7. In the previous exercise, we assume that the underlying list is initially empty. Redo that exercise, this time preallocating an underlying list with length equal to the stack's maximum capacity.
8. Describe a nonrecursive algorithm for enumerating all permutations of the numbers $\{1, 2, \dots, n\}$ using an explicit stack.

Still More Exercises

9. Suppose you have three nonempty stacks R, S, and T. Describe a sequence of operations that results in S storing all elements originally in T below all of S's original elements, with both sets of those elements in their original order. The final configuration for R should be the same as its original configuration. For example, if $R = [1, 2, 3]$, $S = [4, 5]$, and $T = [6, 7, 8, 9]$, the final configuration should have $R = [1, 2, 3]$ and $S = [6, 7, 8, 9, 4, 5]$.
10. Postfix notation is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if “(exp1)op(exp2)” is a normal, fully parenthesized expression whose operation is op, the postfix version of this is “pexp1 pexp2 op”, where pexp1 is the postfix version of exp1 and pexp2 is the postfix version of exp2. The postfix version of a single number or variable is just that number or variable. For example, the postfix version of “((5+2) * (8-3))/4” is “5 2 + 8 3 - * 4 /”. Describe a nonrecursive way of evaluating an expression in postfix notation.