CSE 230: Data Structures

Lecture 4-2: Application of Stacks

Dr. Vidhya Balasubramanian

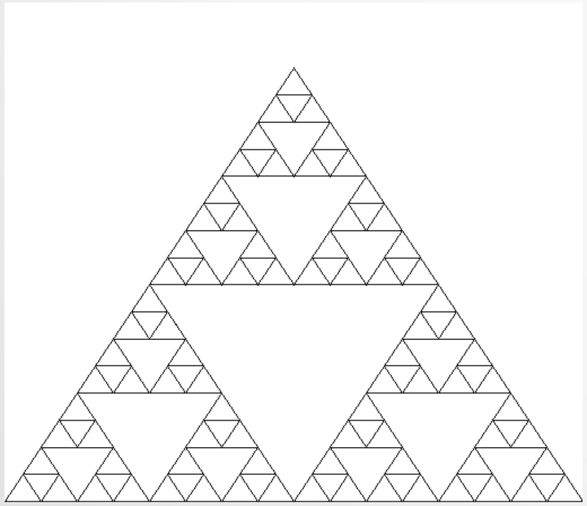
Amrita School of Engineering
Amrita Vishwa Vidyapeetham

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

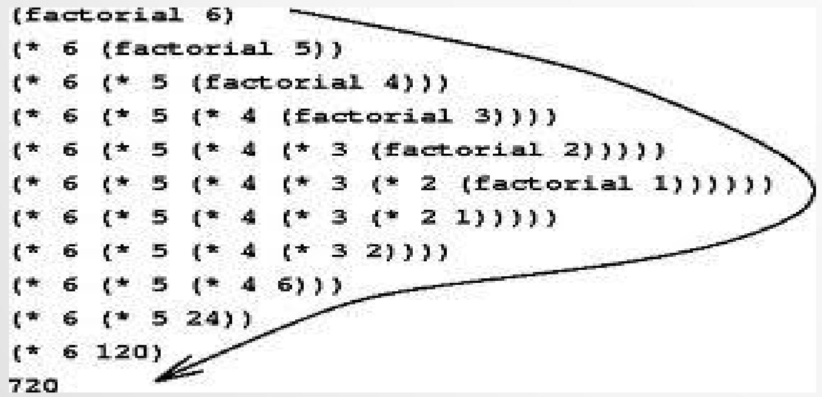


http://introcs.cs.princeton.edu/java/23recursion/images/sierpinski5.png

http://www.spektyr.com/Gallery/Recursion.jpg

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
```

```
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)+$A[n-1]$
```

Src:mitpress.mit.edu

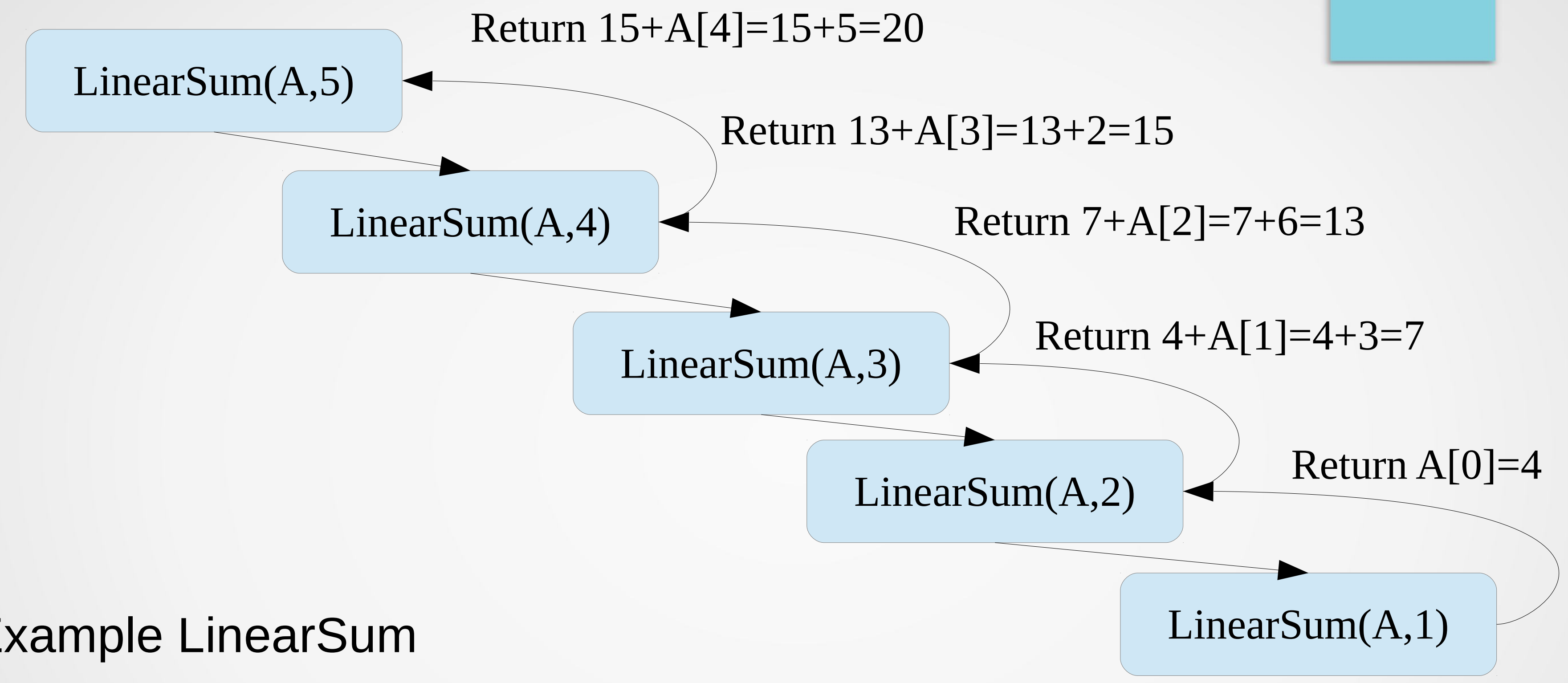
Linear Recursion

- 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

- Use visual tool – recursion trace
  - Contains box for each recursive call
    - Contains parameters of the call
  - Links showing the return value
- Use recurrence relation
  - Mathematical formulation to capture the recursion process

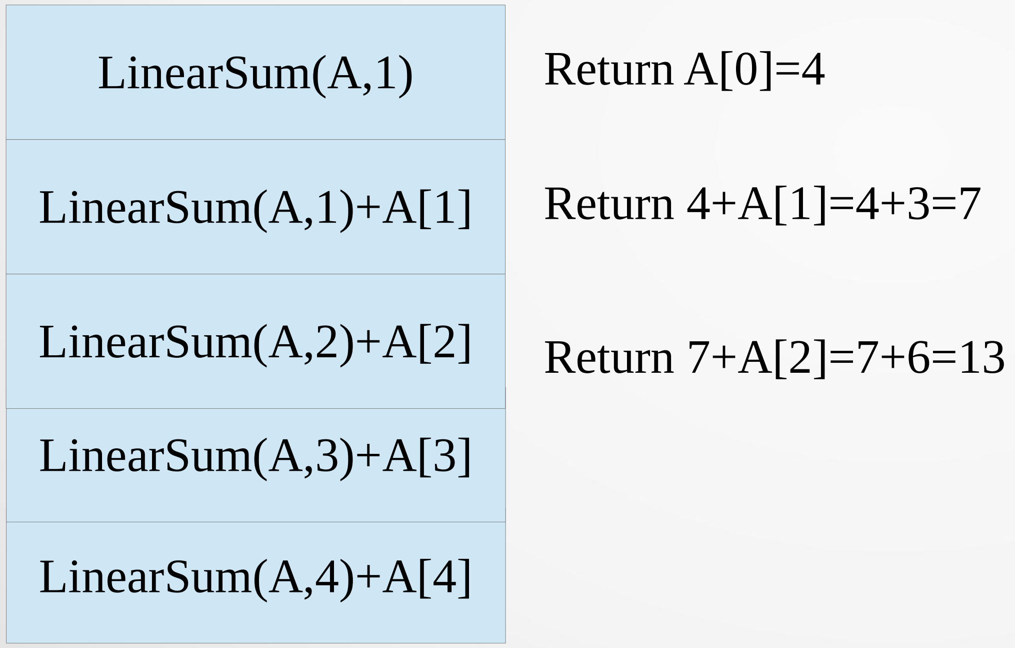
Analyzing Recursive Algorithms



- Example LinearSum
  - Running time is O(n)
  - Space Complexity: O(n)

Stack Trace

- Example stack trace



Problem 1

- Reverse an 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
    return
else
    Swap A[i] and A[i+n-1]
    Call ReverseArray(A,i+1,n-2)
return
```

Problem 2:

- Computing Powers via Linear Recursion

$$power(x,n) = \left\{ \begin{matrix} 1 \text{ if } n=0 \\ \text{x.power}(x,n-1) \text{ otherwise} \end{matrix} \right\}$$

$$power(x,n) = \left\{ \begin{matrix} 1 \text{ if } n=0 \\ \text{x.power}(x,(n-1)/2)^2 \text{ if } n>0 \text{ is odd} \\ \text{power}(x,n/2)^2 \text{ if } n>0 \text{ is even} \end{matrix} \right\}$$

Problem 3

- Describe a linear recursive algorithm for finding the minimum element in an n-element array
- Write a function using recursion to print numbers from n to 0.
- Write a recursive function that computes and returns the sum of all elements in an array, where the array and its size are given as parameters.

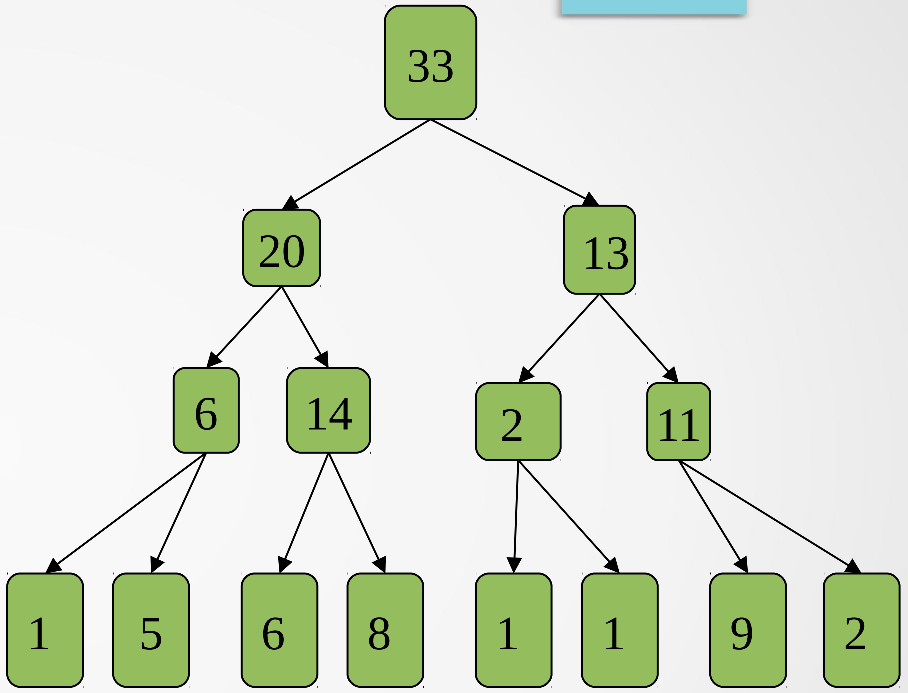
Higher-Order Recursion

- 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,i+⌈n/2⌉,⌊n/2⌋)
```

Analysis

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



Problem

- Generate the kth Fibonacci number using Binary Recursion
- Solution (This method not recommended !! Exponential complexity)
  - **Algorithm** BinaryFib(*k*):

```
Input: k
Output: k^th Fibonacci Number
if k<=1 then
    return k
else
    return BinaryFib(k-1)+BinaryFib(k-2)
```

Problem

- 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

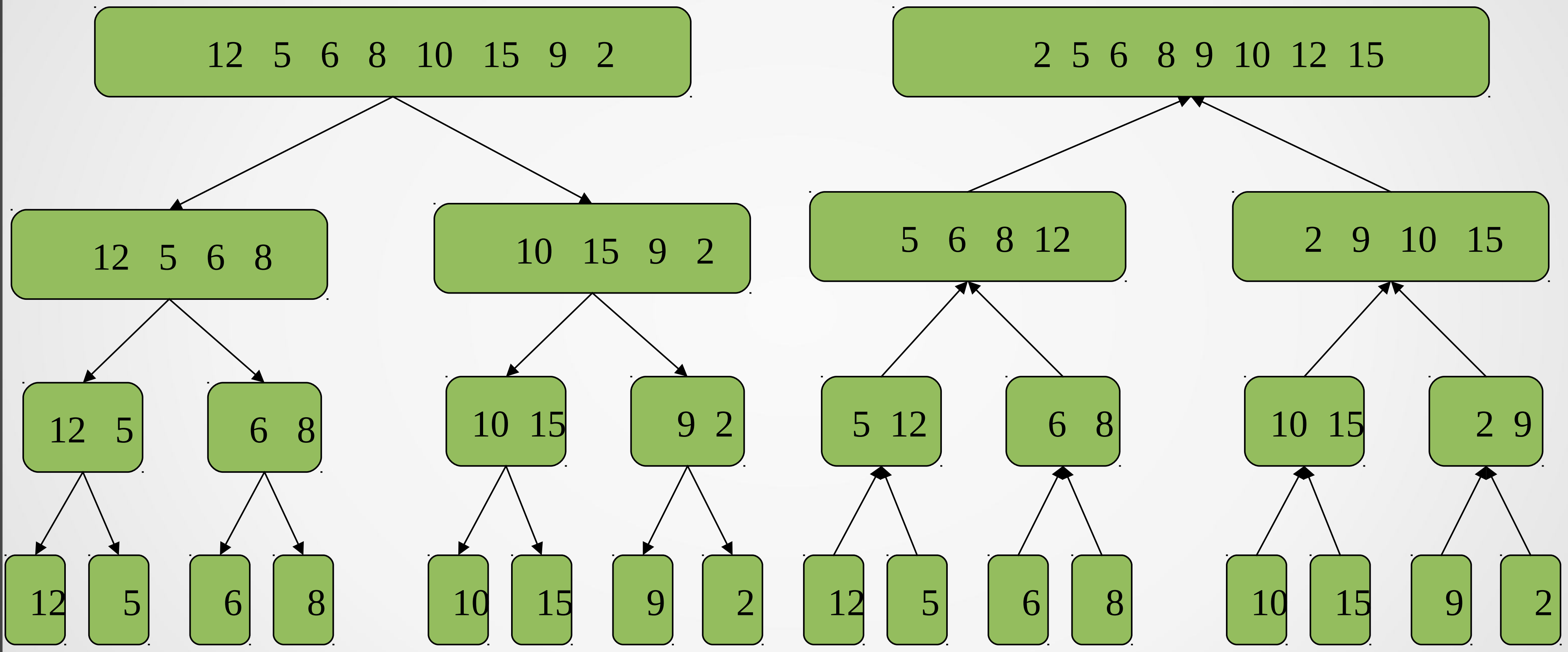
Merge Sort

- Uses divide and conquer strategy (multiple recursion) to sort a set of numbers
- 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
- Recur
  - Recursively apply merge sort to S_1 and S_2
- Conquer
  - Merge S_1 and S_2into a sorted sequence

Merging of Sorted Sequences

- Iteratively remove smallest element from one of the two sequences S_1 and S_2 and add it to end of output sequence S

Merge Sort

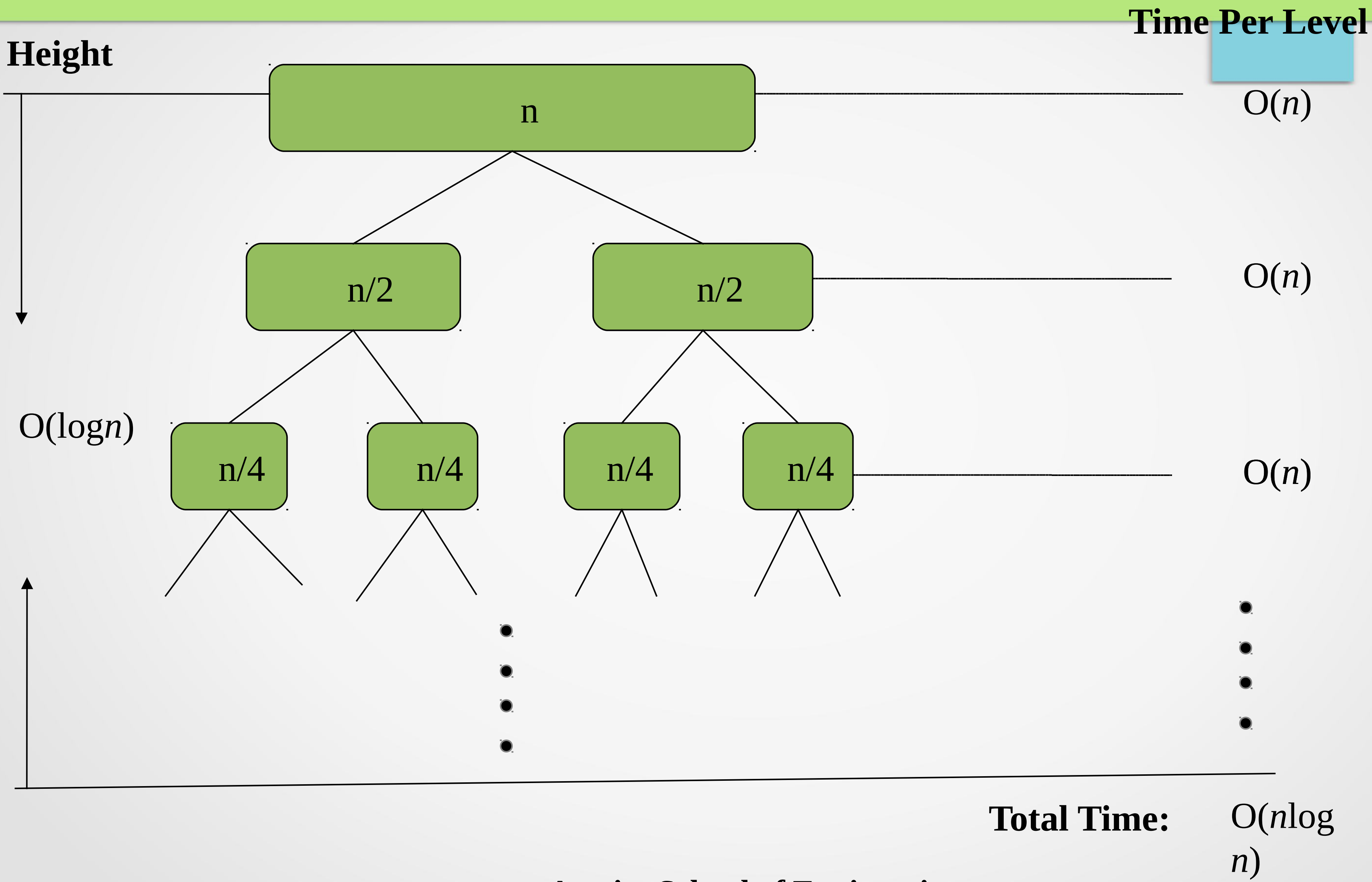

12 5 6 8 10 15 9 2
2 5 6 8 9 10 12 15
12 5 6 8
10 15 9 2
5 6 8 12
2 9 10 15
12 5
6 8
10 15
9 2
5 12
6 8
10 15
2 9
12
5
6
8
10
15
9
2
12
5
6
8
10
15
9
2

Merge Sort

```
• 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);
      }
  }
```

Src: Skiena, Algorithm Design Manual, Chapter 4

Analysis of Merge Sort



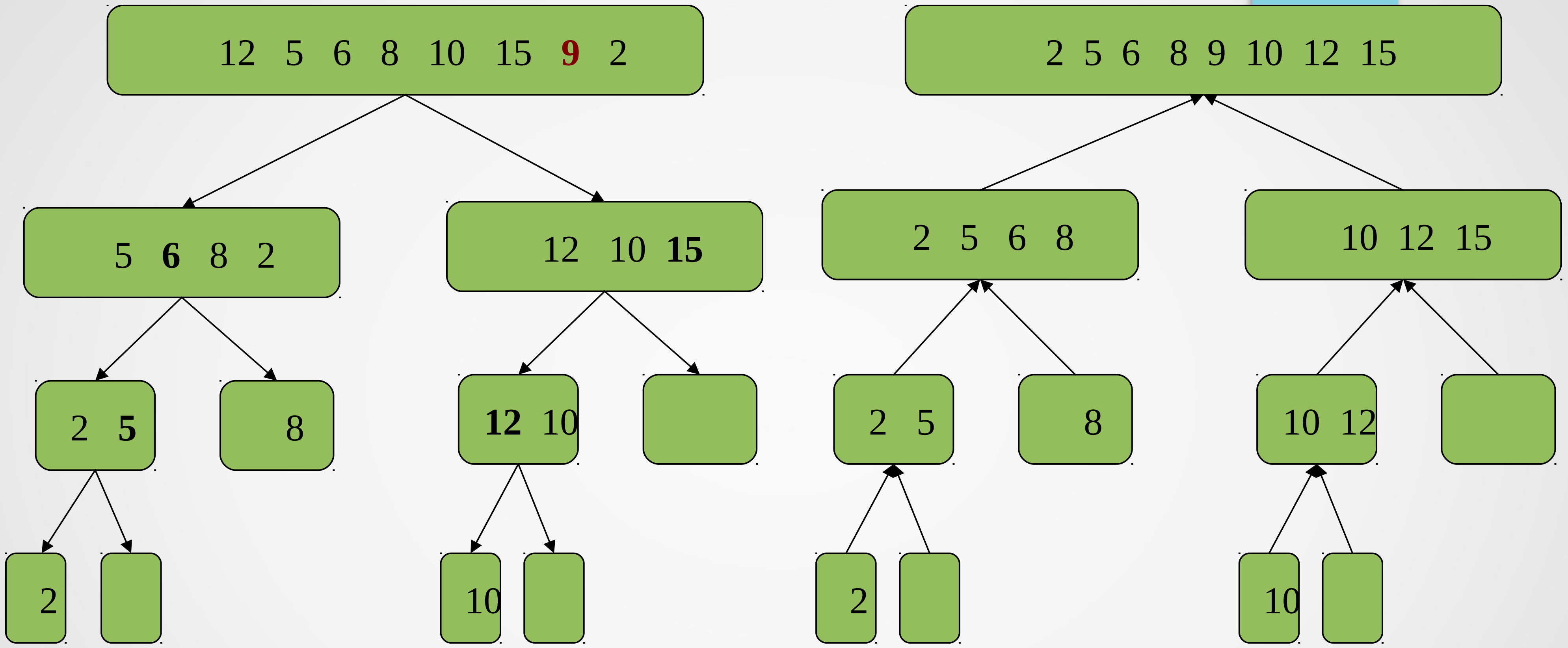
Merge Sort: Analysis

- work done on the kth level involves merging 2^k pairs sorted list, each of size $n/2^{k+1}$
  - A total of atmost n ie 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

Quick Sort

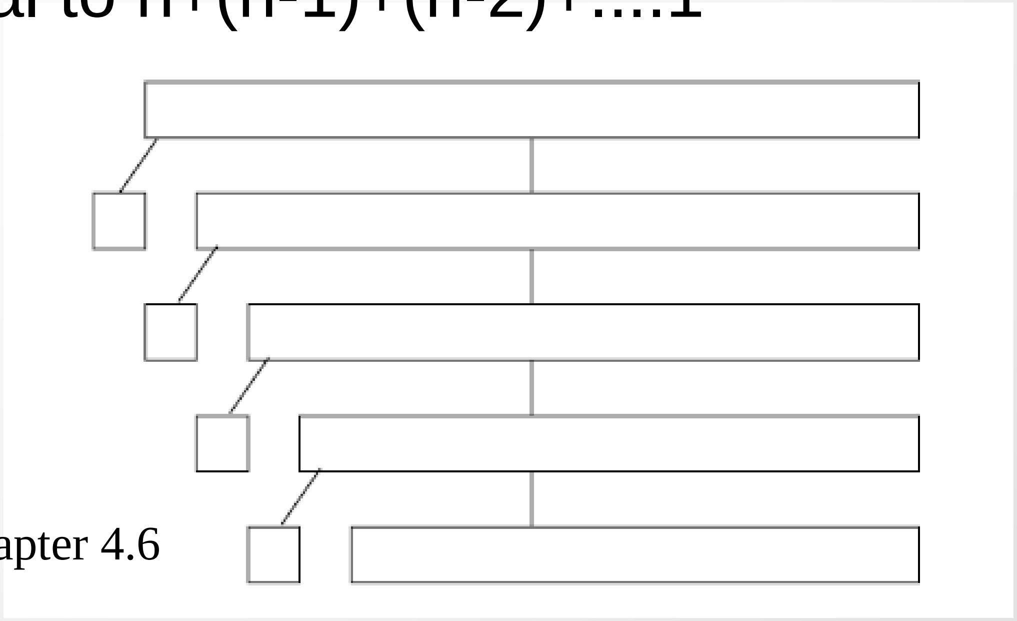

12 5 6 8 10 15 9 2
5 6 8 2
12 10 15
2 5
8
12 10
2
10
2 5 6 8 9 10 12 15
2 5 6 8
10 12 15
2 5
8
10 12
2
10

Quick Sort

- QUICKSORT(A, p, r)
  - ***if*** p < r
  - q = 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)+....1
  - O(n^2)



Src: Skiena, Algorithm Design Manual, Chapter 4.6

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

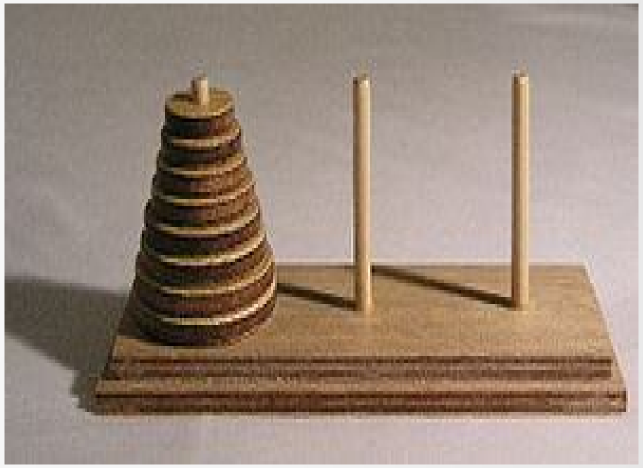
Towers of Hanoi

- 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 for moving from one pole to another
  - 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

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

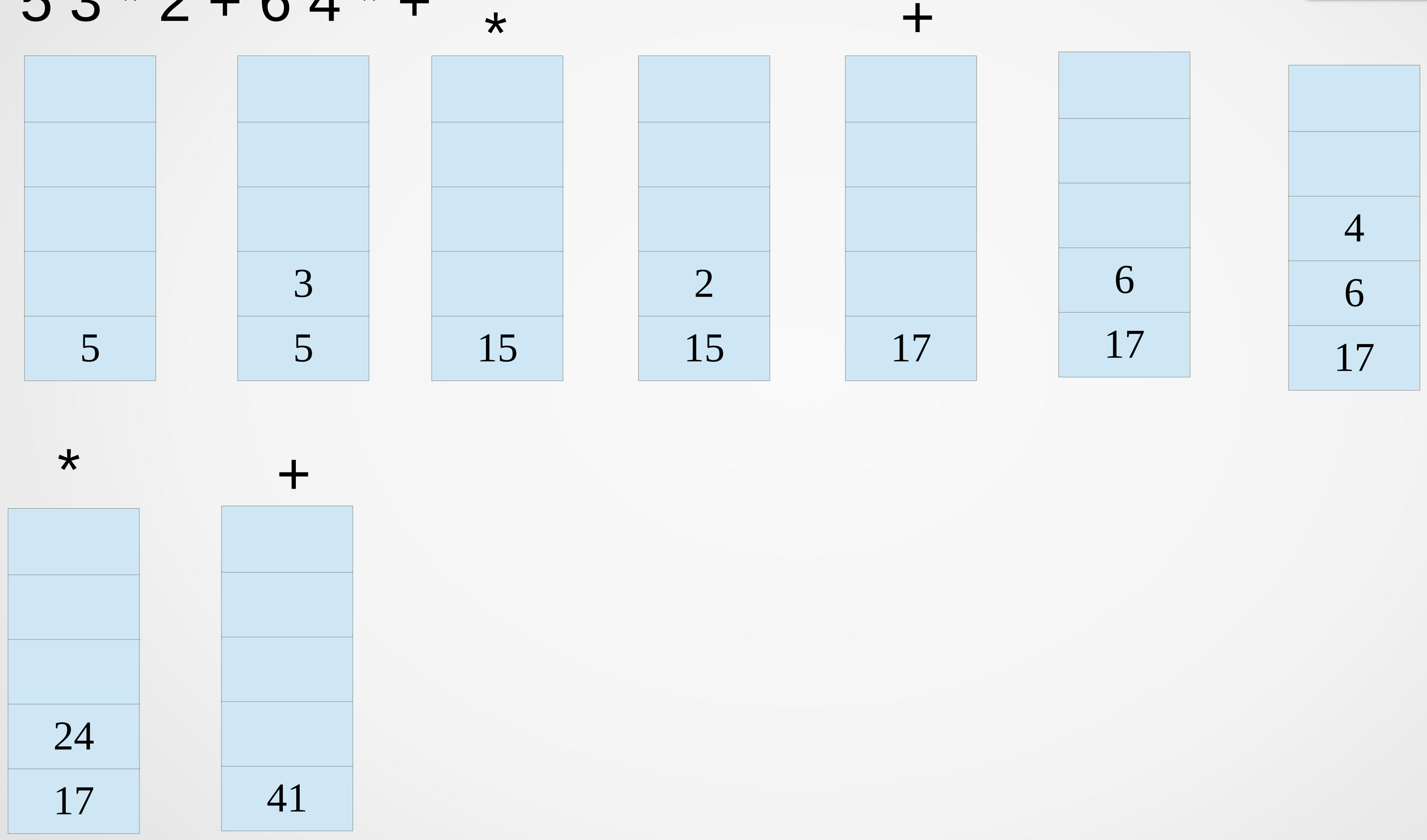
- Precedence of * and / higher than + and -
  - For same precedence, use left associativity
- e.g a + b * c
  - Parenthesize it => a + (b * c)
  - Convert the multiplication => a + (b c *)
  - convert the addition =>a(bc*)+
  - Remove parenthesis => a b c * +
- Find the postfix form of a + ((b * c) /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


5
3
5
*
15
2
15
+
17
6
17
4
6
17
*
24
17
+
41