Introduction

Instructor: Prof. Tianyi ZANG

tianyi.zang@gmail.com

School of Computer Science and Technology

Harbin Institute of Technology

Course Information

- Course code: 13SC03100120
- Involved class:
  - 1503101,1503102,1503103,1503104,1503105,1503106, 1503107,1503201,1503202,1503401,1536101
  - Lecture hours: 48 + 12
  - Credit: 3.5
- When:
  - Week 4-16
  - Friday, 8:00-9:45
  - Sunday, 8:00-9:45
- Where:
  - R120, Zhen Xin Building

References

- 《数据结构与算法(第4版)》廖明宏，郭福顺，张岩，李秀坤，高等教育出版社 （**Textbook in Chinese)**

- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, And Clifford Stein, MIT Press
- **A Practical Introduction to Data Structures and Algorithm Analysis**, Clifford A. Shaffer, Dover Publication
- **Data Structure and Algorithm Analysis**, Mark Allen Weiss, Pearson Education
- 《数据结构(C语言版)》严蔚敏，吴伟民，清华大学出版社

More information

- **The Design and Analysis of Computer Algorithms**, Alfred V. Aho, ***John E. Hopcroft***, Jeffrey D. Ullman, Addison-Wesley Series in Computer Science and Information Processing, 1974.
- ***John E. Hopcroft***,
  - **ACM Turing Award (1986)**
  - For his fundamental achievements in the design and analysis of algorithms and data structures
- His books continue to influence students 40 years later

Coursework and grading

- Programming exercises/assignments. 10%
  - practice implementing and using data structures and algorithms by writing programs
  - collaboration and lateness policy

- Programming practice. 20%
  - practice implementing and using data structures and algorithms by writing programs
  - When: 13:45-15:30, Sunday in week 9, 11,13, and 15
  - Where: Room 208, Ge Wu Building
- Final Exams. 70%
  - When: week 18, Thursday, 15:45-17:45 (5^{th}, January, 2017
  - Where: Room 24, Zheng Xin Building

Some more …

- You may …
  - Stop me when you are not clear with what I am talking, but put your hand up first.
  - Bring some drinks for refreshment, but …
- Please …
  - Don’t be late!
  - Switch off your mobile!
  - Don’t talk anything that has nothing to do with this course!

Questions

- Don’t hesitate to ask questions if you are not sure.
- There is no silly question, …
- But silly answers!!!

Overview

What is this course about?

- Data structure: method to store information
- Algorithm: method for solving a problem

- Programming and problem solving, with applications, based on projects.
  - *To apply the method you learned to solve practical problems.*

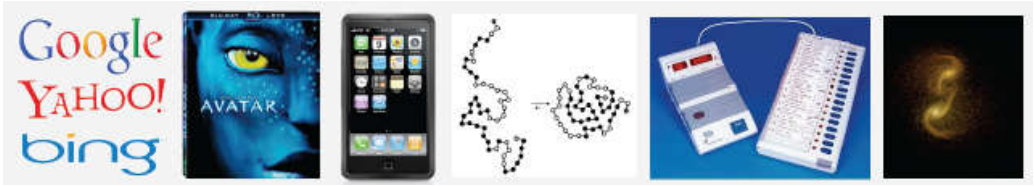
Goals

- Understand the fundamental concepts, and to have big idea about the course of data structure and algorithms.
- Grasp the important concept of Abstract Data Type (ADT)
- Understand the definition of algorithm, complexity of algorithm, algorithm refinement, and basic process and method to solve a problem.

Why study algorithms?

The impact of algorithms is broad and far-reaching

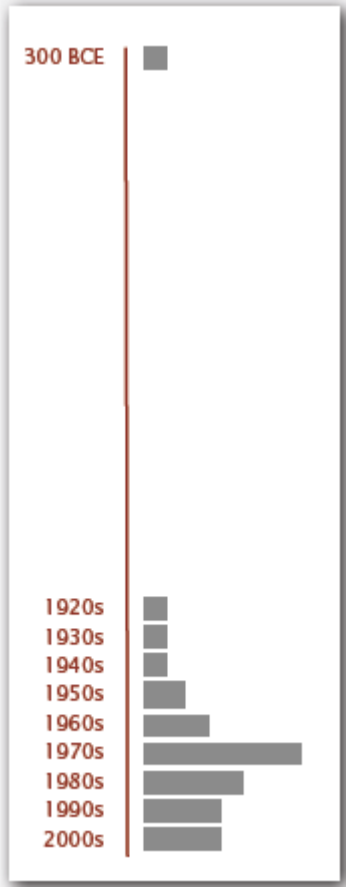
- Internet. Web search, packet routing, distributed file sharing, ...
- Biology. Human genome project, protein folding, …
- Computers. Circuit layout, file system, compilers, …
- Computer graphics. Movies, video games, virtual reality, …
- Security. Cell phones, e-commerce, voting machines, …
- Multimedia. MP3, JPG, HDTV, song recognition, face recognition, …
- Social networks. Recommendations, dating, advertisements, …
- Physics. N-body simulation, particle collision simulation, …



Why study algorithms?

Old roots, new opportunities.

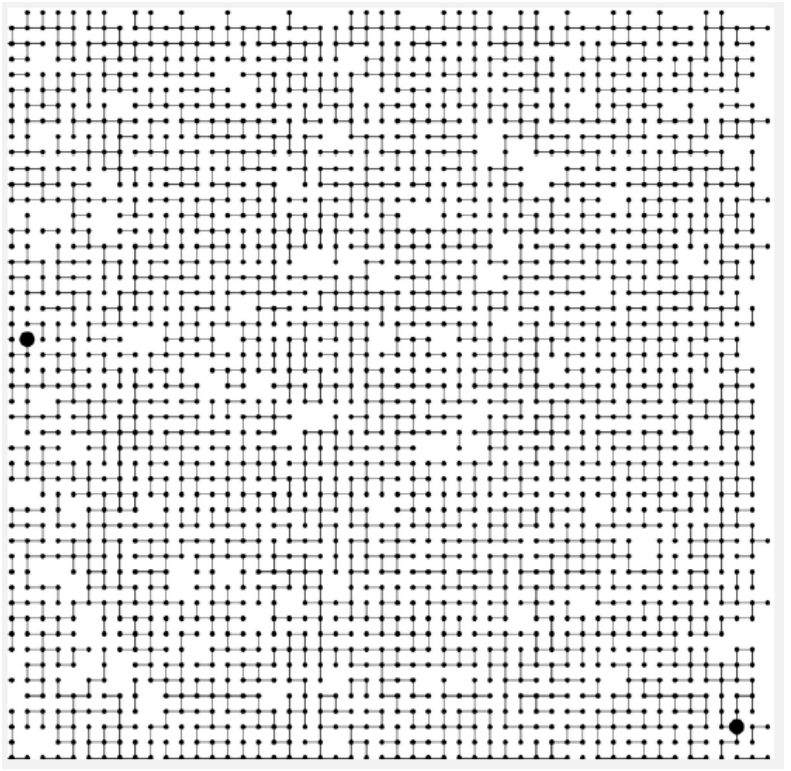
- Study of algorithms dates at least to Euclid.
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!



Why study algorithms?

To solve problems that could not otherwise be addressed

- Ex. Network connectivity.



Why study algorithms?

For intellectual stimulation.

Frank Nelson Cole

"On the Factorization of Large Numbers"

American Mathematical Society, 1903

$2^{67}-1 = 193{,}707{,}721 \times 761{,}838{,}257{,}287$

Why study algorithms?

- Computational models are replacing mathematical models in scientific inquiry.

$$E = mc^2$$

$$F = ma \qquad F = \frac{Gm_1m_2}{r^2}$$

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right]\Psi(r) = E\,\Psi(r)$$

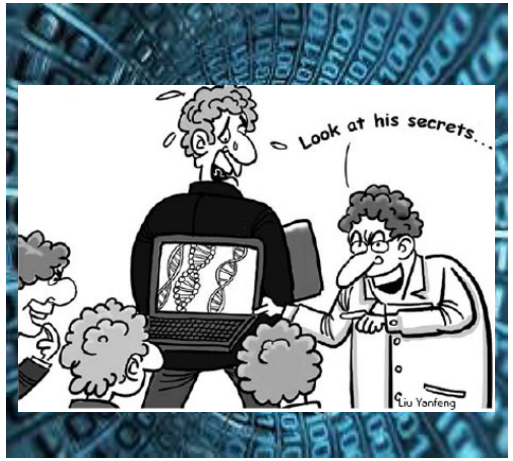
20th century science
(formula based)

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

21st century science
(algorithm based)

Why study algorithms?

They may unlock the secrets of life and of the universe.



" Algorithms: a common language for nature, human, and computer. " — *Avi Wigderson*

Why study algorithms?

To become a proficient programmer.

"The difference between a bad programmer and a good one is whether [the programmer] considers code or data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

— Linus Torvalds (creator of Linux)

" Algorithms + Data Structures = Programs. " *— Niklaus Wirth*

Why study algorithms?

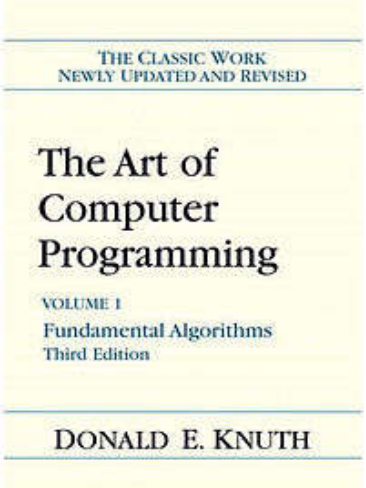
Why study algorithms?

" For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. " — *Francis Sullivan*

Why study algorithms?

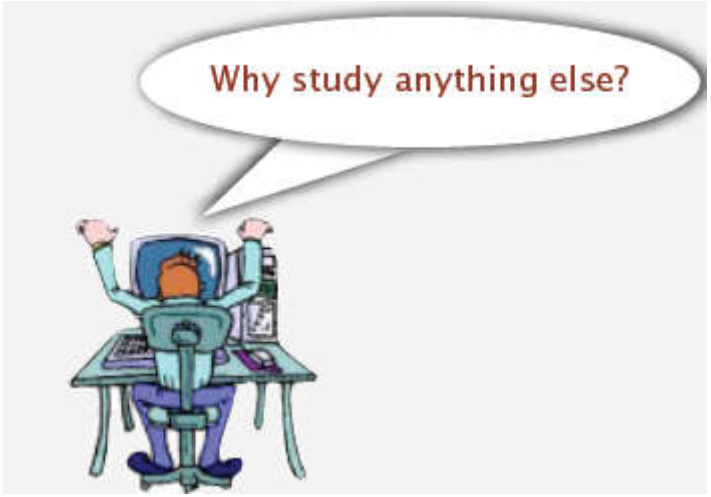
> *" An algorithm must be seen to be believed. "* — *Donald Knuth*

- **Known for**
  - *The "father" of the analysis of algorithms*
  - *The Art of Computer Programming*
  - *Turing Award (1974, only 35 years old)*



Why study algorithms?

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- Computational models are replacing mathematical models in scientific inquiry.
- They may unlock the secrets of life and of the universe.
- To become a proficient programmer.
- For fun and profit.



Courses relationships.

Basic Requirement

Grasp basic programming method

Grasp data organization and data processing methods.

Grasp the methods of developing large software projects

C, C++, Java → Data Structure → Software engineer

Learn spelling words

Learn writing short paper

Learn writing book

What is Data

- Data
  - The set of symbols which can be recognized, stored and processed by the computer.
  - The carrier of information.
- Examples
  - Integer or real. (processed by digital processing programs)
  - Images. (processed by image processing programs)
  - Students in our class room
  - Source program.(processed by compiler)

Abstract data types

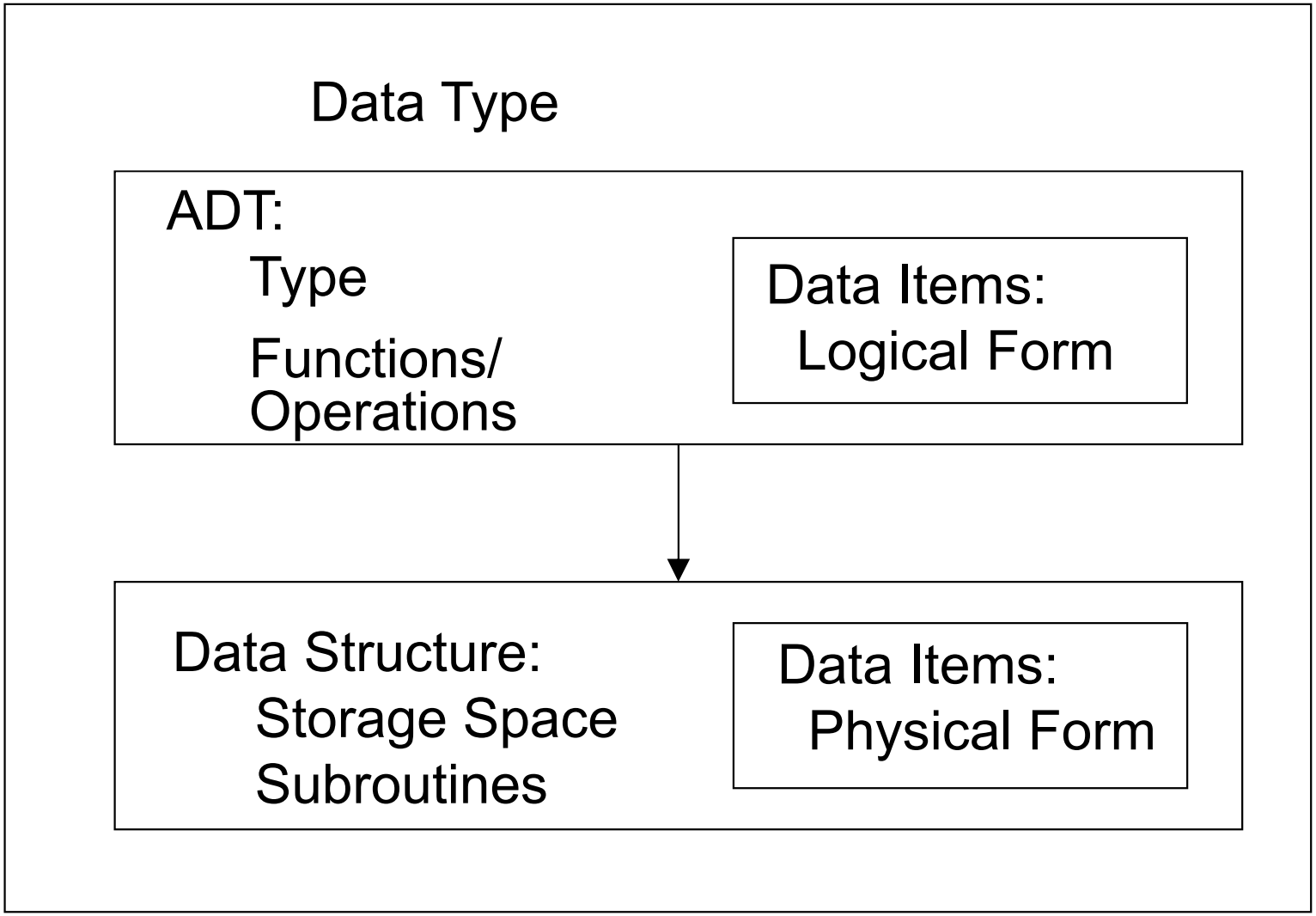
- A ***type*** is a collection of values. Boolean type, integers, etc.
- A ***data item*** is a piece of information or a record whose value is drawn from a type.
- ***Data types***
  - A *data type* is a set of values and a set of operations on those values.
- ***Abstract data types (ADT)***
  - A mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
  - A data type whose internal representation is hidden from the client/user.

Data Structure

- A ***data structure*** is the physical implementation of an ADT.
  - Data **organization**
  - Each function/operation associated with the ADT is implemented by **one or more subroutines** in the implementation.
- Data structure
  - usually refers to an organization for data in main memory.
- File structure
  - an organization for data on peripheral storage, such as a disk drive, CD etc.

Logical vs. Physical Form

- Data items have both a logical form and a physical form.
- **Logical form**: definition of the data item within an ADT.
  - Ex: Integers in mathematical sense: +, -
- **Physical form**: implementation of the data item within a data structure.
  - Ex: 32/64 bit integers, overflow


Data Type
ADT:
Type
Functions/
Operations
Data Items:
Logical Form
Data Structure:
Storage Space
Subroutines
Data Items:
Physical Form

Research Topics of Data Structure

- Logical structure
  - The logical relationship between data.
- Storage structure
  - Storage mapping of logical structure in storage.
- How to implement operations based on above logical and storage structure

Logical Structure

- A data structure is a **representation** (in a computer) for a collection of **related** Data.

- Data_structure = {D,R}
  - D: Data Object
  - R: A set of relationships between data objects.

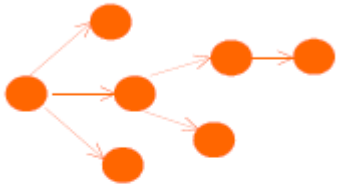
Logical Structure

- The logical structure can be formally represented as :
  - L = (N, R)
  - N: the set of the node.
  - R: the set of ordered or unordered pair on N.
- Examples
  - $L = (N, R)$
  - $N = \{ a_0, a_1, a_2 \}$ $R = \{ < a_0, a_1 >, < a_0, a_2 > \}$
  - "$<a_i, a_{i+1}>$" is ordered pair means that elements a_i, a_{i+1} are adjacent and a_i is located before a_{i+1}, a_{i+1} is located after a_i.

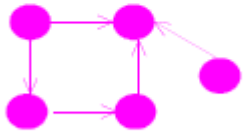
Basic Logical Structures

- Linear structure

- Tree structure



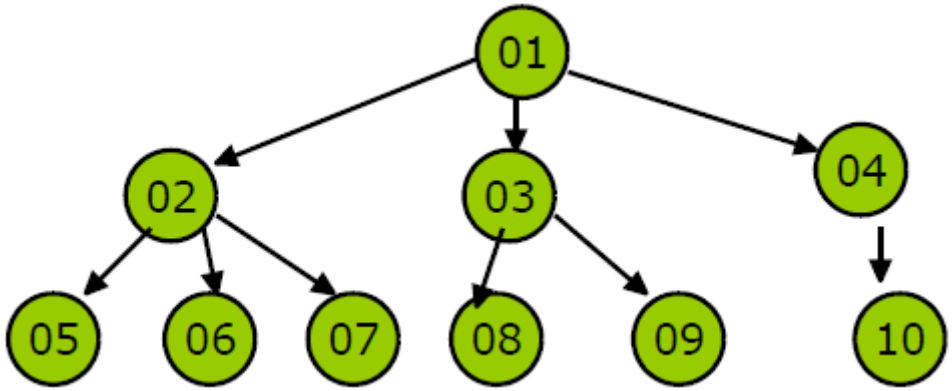
- Graph structure



- Set structure

Example tree

- Data Structure Tree=(K,R)
  - K={01,02,03,04,05,06,07,08,09,10}
  - R={<01,02>,<01,03>,<01,04>,<02,05>,<02,06>,<03,07>,<03,08>, <03,09>,<04,10>}



Basic Structure

- Linear Structure
  - There is only one start node and final node.
  - Every internal node have one and only one predecessor and successor.

- None linear structure
  - Nodes in the structure can have more than one predecessor and successor.

Operations on logical data structures

- A set of operations can be defined on a logical data structure.

- For example, operations on the data structure "student table" can be: insert, delete, update, search.

St. NO.	Name	Gender	Class
35	John	F	1201
20	Mike	F	1202
12	Vivian	M	1203

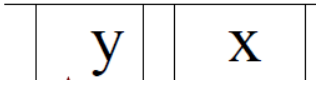
Storage Structure

- Storage structure of data
  - The mapping of data element
  - The mapping of relationship

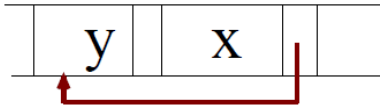
- The mapping of data element
  - representing a data element using binary bits
  - $(321)_{10} = (101000001)_2$
  - A = $(001000001)_2$

Storage Structure

- Mapping of relationship
  - ***Sequential mapping***: representing successor relationship using relative storage location.
  - Storage structure only contains data element information. No additional info.



  - ***Chaining mapping*** : representing successor relationship using **pointer.**
  - Besides data element information, there is additional information in storage structure to represent the logical structure.



Operations on the Data Structure

- The implementation of the same operation is different using different storage mapping.

- How to map the logical structure depends on time and storage costs.

Why So Many Data Structures?

- Ideal data structure:
  - "fast", "elegant", memory efficient
- Generates tensions:
  - time vs. space
  - performance vs. elegance
  - generality vs. simplicity
  - one operation's performance vs. another's

The study of data structures is the study of tradeoffs. That's why we have so many of them!

Efficiency vs. Cost

- A solution is said to be efficient if it solves the problem within its resource constraints.
  - Space
  - Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine the basic operations that must be supported.
- Quantify the resource constraints (space & time) for each operation.
- Select the data structure that best meets these requirements.

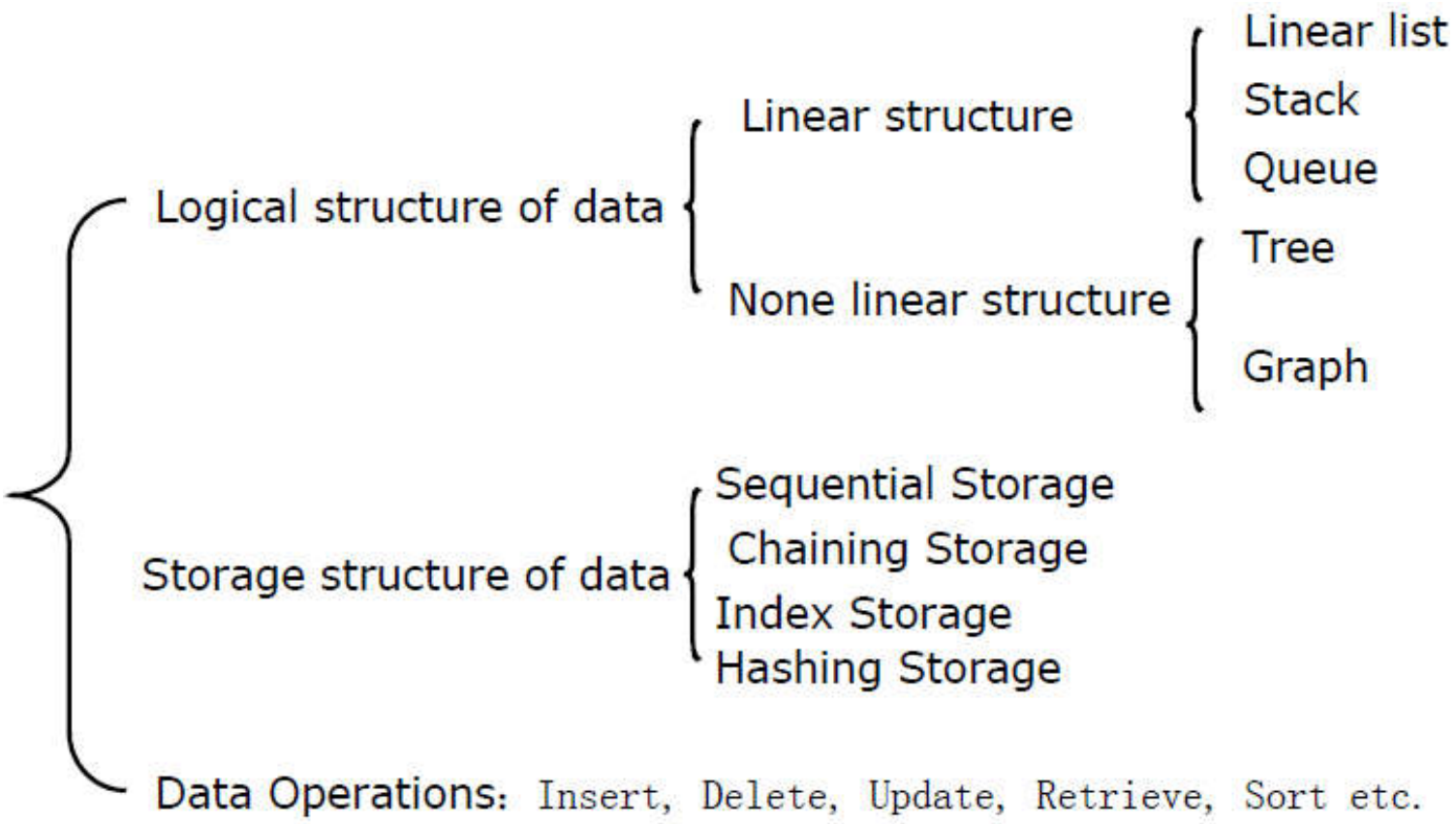
Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.

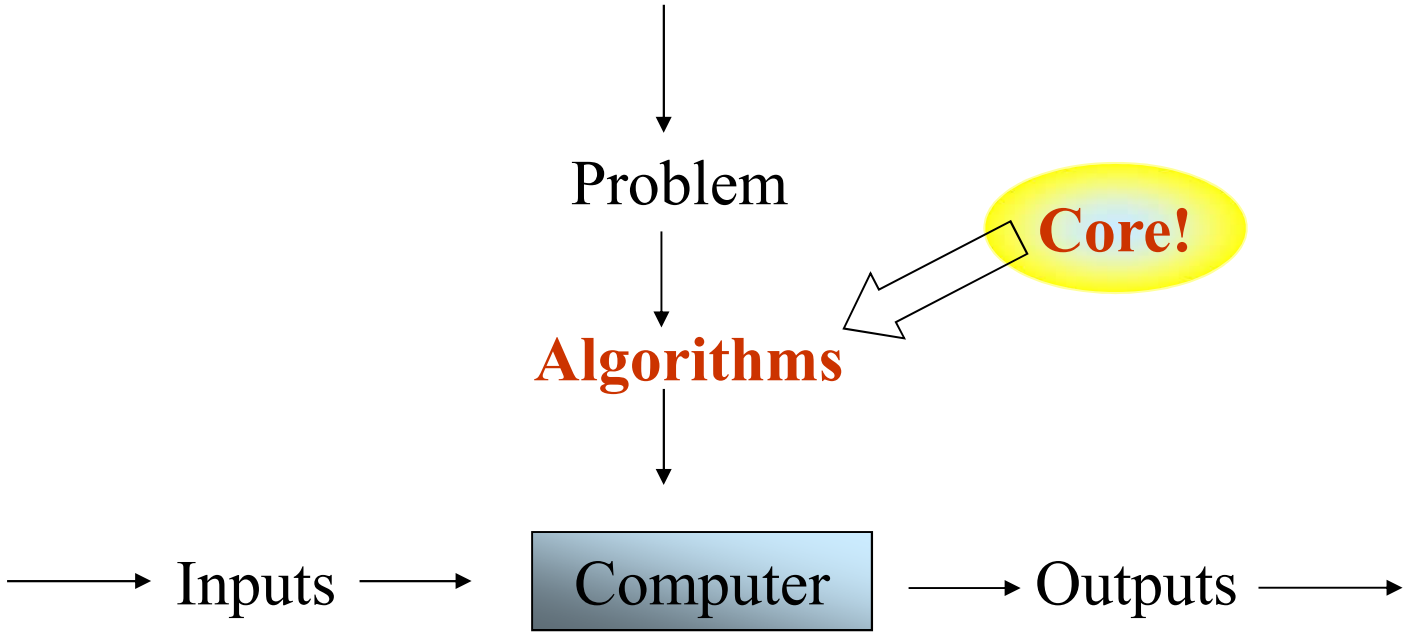
Data Structure Philosophy (cont)

- Each problem has constraints on available space and time.
- Only after a careful analysis of problem characteristics can we know the best data structure for the task.
- Bank example:
  - Start account: a few minutes
  - Transactions: a few seconds
  - Close account: overnight

Summary:
Three aspects of data structure



How a problem is solved?



Algorithm

- Algorithm:
  - a limit sequence of instructions used to solve some problems.
  - a method or a process followed to solve a problem.

- A algorithm can be described using nature language or pseudo programming language.
- An algorithm must meet the following 5 requirements:
  - (1) finite (2) definitude (3) input (4) output (5) feasibility

Algorithm Requirements/Properties

1. **Input**: one algorithm can have zero or many input data. The input data get from definite object set.
2. **Output**: an algorithm can have one or many output. The output has specific relationship with input.
3. **Finite**: terminate after a finite number of steps.
   - It must terminate.
4. **Definitude**: clear and unambiguous
   - Same input will get same output.
   - There can be no ambiguity as to which step will be performed next.
5. **Feasibility**: instruction is basic enough to be carried out

Which requirements they violate?

```
void exam1()
{
    n=2;
    while (n%2==0)
        n=n+2;
    printf("%d\n",n);
}
```

Endless loop. infinite

```
void exam2()
{
y=0;
x=5/y;
printf(
"%d,%d\n",x,y);
}
```

violate Feasibility

Algorithms and Programs

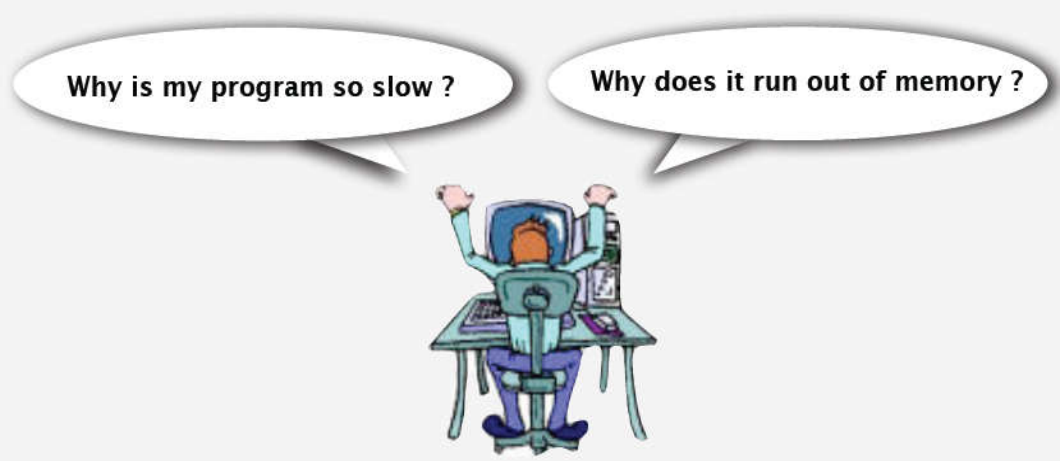
- An algorithm takes the input to a problem (function) and transforms it to the output.
  - A mapping of input to output.
- A problem can have many algorithms.

- ***Programs = Algorithms + Data Structures***
- A ***computer program*** is an instance, or concrete representation, for an algorithm in some programming language.

Taking away

- Any organization for a collection of records can be searched, processed in any order, or modified.

- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.



Principles of designing algorithm

- Correctness: for all legal input, after finite times of execution, the algorithm will generate correct result.
- Readability : easier for designer and others to read, understand, update and reuse.
- Robust: When input illegal data, the algorithm can react properly.
- Efficient and low storage requirement:
  - efficiency means the execution time of the algorithm
  - Amount of storage means the maximum storage requirement.
  - Both of them related to the scale of the problem.

Algorithm Efficiency

- There are often many approaches (algorithms) to solve a problem. How do we choose between them?
- For the computer program design, there are two **goals** (sometimes conflicting) .
  - To design an algorithm that is easy to understand, code, debug, and ease to use.
    - programmers effort
  - To design an algorithm that makes efficient use of the computer's resources.
    - Time, Space (disk, RAM)

Algorithm Efficiency (cont)

- Goal (1) is the concern of Software Engineering.
- Goal (2) is the concern of data structures and algorithm analysis.
- When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

- Analysis after execution: Empirical comparison
  - must execute the program
  - affected by some other factors, machine dependent
  - difficult to do “fairly” and is time consuming.

- Analysis before execution: Asymptotic algorithm analysis
  - space complexity: storage requirement
  - time complexity: computing time

Factors affect execution time

What kind of factors?

- The strategy selected by the algorithm
- The scale of the problem
- Specific input values for given problem size
- The programming language
- The machine language generated by the compiler.
- The speed the computer executing instructions, machine load, and OS.

It is not good for evaluating the efficiency of an algorithm based on absolute time unit.

Simplifying the calculations

> *" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. "* — *Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

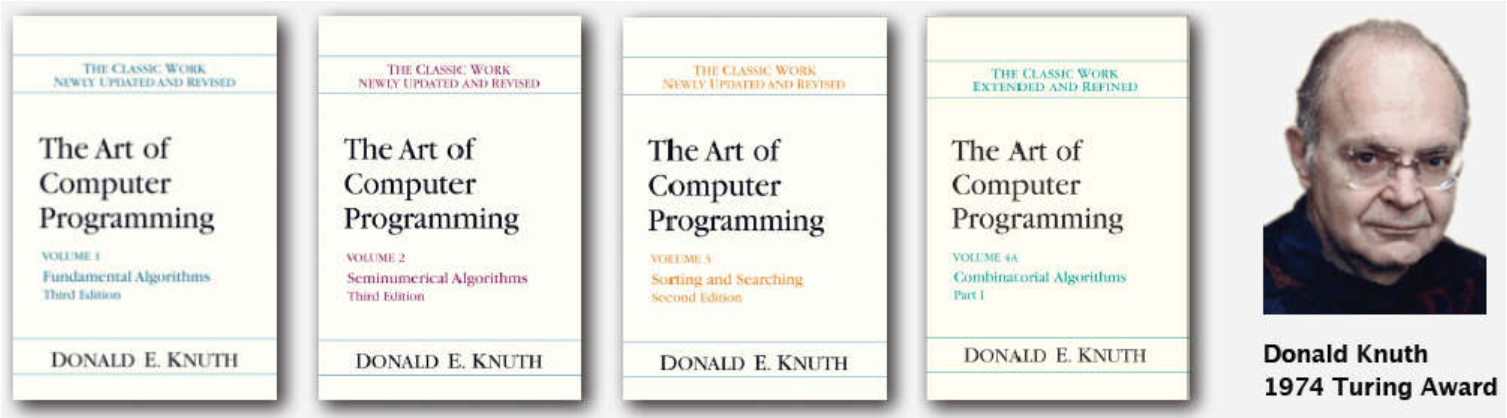
[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

How to Measure Efficiency?

- Total running time: sum of cost × frequency for all operations.
  - Need to analyze program to determine set of operations.
  - Cost depends on machine, compiler.
  - Frequency depends on algorithm, input data (distribution).



In principle, accurate mathematical models are available.

Performance analysis before execution

- Algorithm=control structures + basic operations

- The measurement of time complexity is based on the **maximum execution times (frequency) of basic operations**.
- Normally **basic operations are sentences** in the most inner loops.

Basic Operations

- Normally the ***max execution times*** of some basic operations are used as the time measurement of complicity of the algorithm.

```
#define MAX   20
void matrixadd(int n,int A[MAX][MAX],
        int B[MAX][MAX],int C[MAX][MAX])
   {    int i,j;
        for (i=0;i<n;i++)
        for (j=0;j<n;j++)
                C[i][j]=A[i][j]+B[i][j];
   }
```

How many times? *O(n2)*

Asymptotic Analysis

- The main purpose of the analysis is to get a sense for the trend (**growth rate, order-of-growth**) in the algorithm's running time.
- **Asymptotic analysis** is based on two simplifying assumption, which hold in most (but not all) cases.
  - **Large input sizes**: We are most interested in how the running time grows for large values of n
  - **Ignore constant factors**: The actual running time of the program depends on various constant factors in the implementation

Asymptotic Analysis

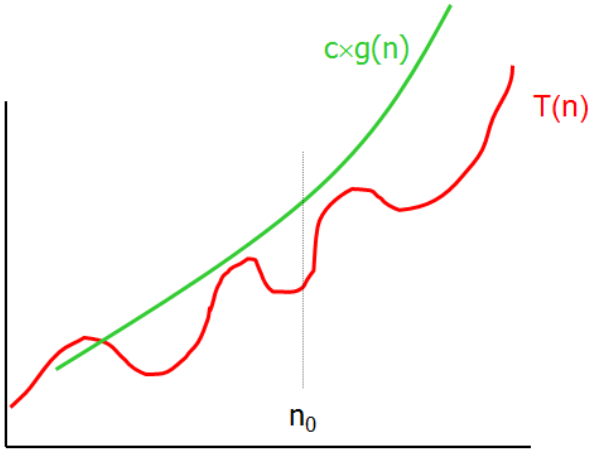
- For most algorithms, running time depends on "size" of the input.
- Running time is expressed as *T(n)* for some function T on input size n.
- *T (n)* is called time complexity of the algorithm

Asymptotic Analysis: Big-oh

Definition: For **T**(*n*) a non-negatively valued function, **T**(*n*) is in the set O(*g*(*n*)), if there exist two positive constants *c* and n_0 such that **T**(*n*) <= *cg*(*n*) for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n>n_0$), the algorithm always executes in less than *cg*(*n*) steps in [best, average, worst] case.

Use: The algorithm is in O(n^2) in [best, average, worst] case.



To be asymptotically equivalent to n^2 as $n \to \infty$

Big-oh Notation (cont)

Big-oh notation indicates an upper bound.

Example: If $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $O(n^2)$.

Look for the tightest upper bound:

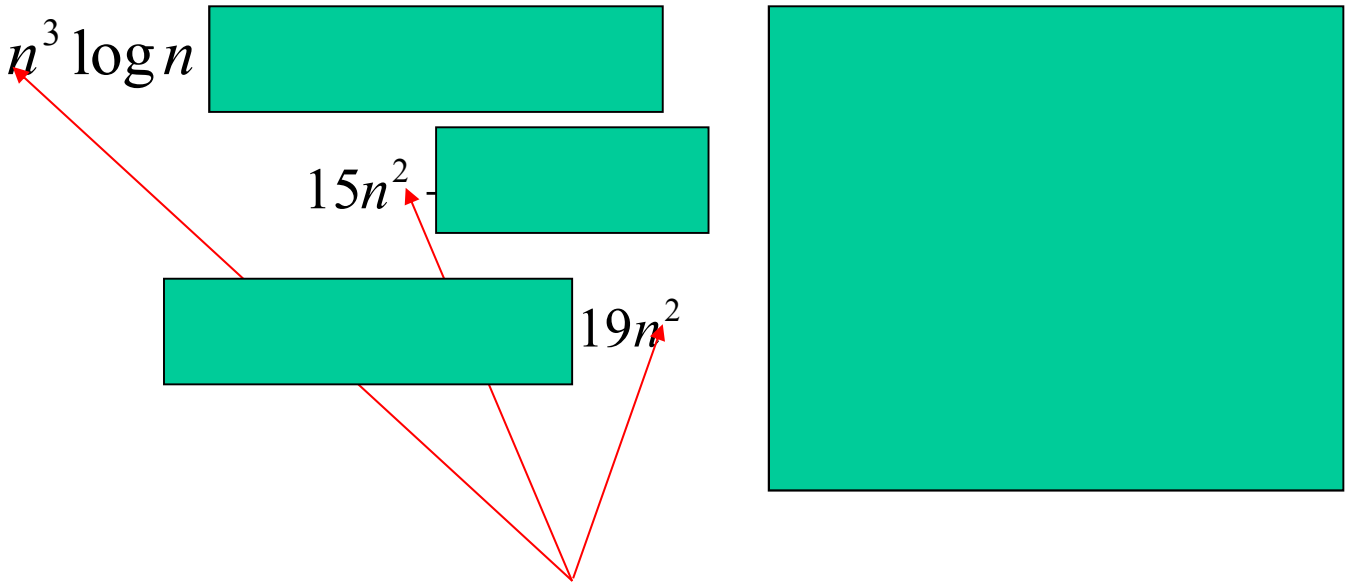
While $\mathbf{T}(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Simplifying with Big-O

By definition, Big-O allows us to:

- Eliminate low order terms
  - $4n + 5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
- Eliminate constant coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $\log n \Rightarrow \log n$
  - $\log_3 n = (\log_3 2) \log n \Rightarrow \log n$

Examples



The first algorithm is significantly slower for large n, while the other two are comparable, up to a constant factor

Big-Oh Examples

Example 1: Finding value *X* in an array (average cost).

Then $\mathbf{T}(n) = c_s n/2$.

For all values of $n > 1$, $c_s n/2 <= c_s n$.

Therefore, the definition is satisfied for $f(n)=n$, $n_0 = 1$, and $c = c_s$.

Hence, $\mathbf{T}(n)$ is in $O(n)$.

Big-Oh Examples (2)

Example 2: Suppose T(n) = $c_1n^2 + c_2n$, *where* c_1 and c_2 are positive.

$c_1n^2 + c_2n <= c_1n^2 + c_2n^2 <= (c_1 + c_2)n^2$ for all $n > 1$.

Then T(n) <= cn^2 whenever $n > n_0$, for $c = c_1 + c_2$ and n_0=1.

Therefore, **T**(n) is in O(n^2) by definition.

Example 3: **T**(n) = c. Then **T**(n) is in O(1).

Big-Oh Example (3):

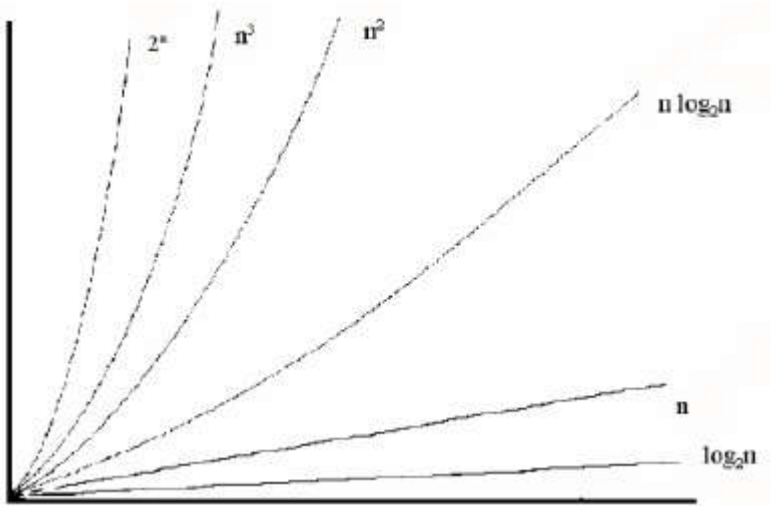
```
void Mult_matrix( int c[][], int a[][], int b[][]，  int n)
   {
      for (i=1; i<=n; ++i)
         for (j=1; j<=n; ++j) {
            c[i,j] = 0;
            for (k=1; k<=n; ++k)
               c[i,j] += a[i,k]*b[k,j];
         }
   }// Mult_matrix
```

What is the big-oh? $O(n^3)$

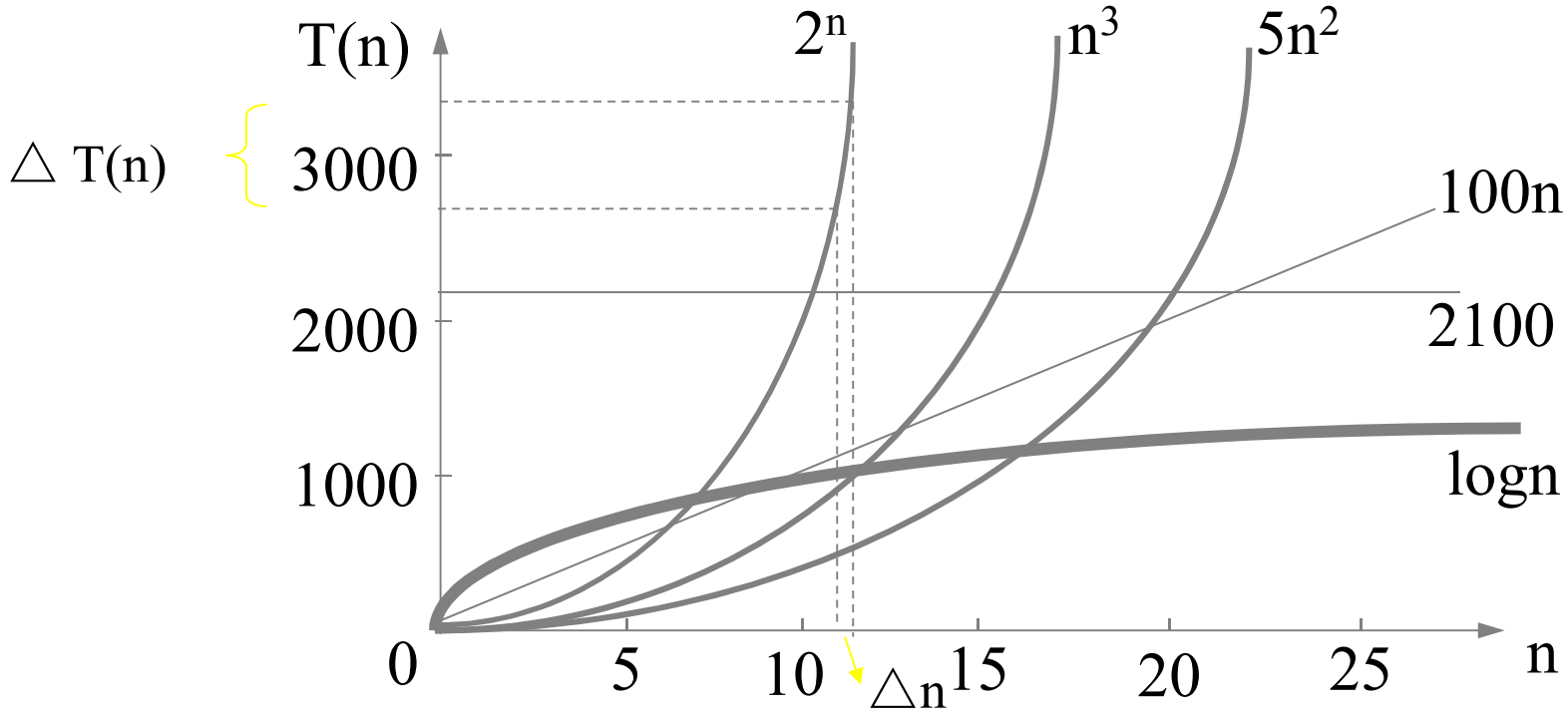
Classify of algorithm with Big-O

- examples：
  - (a) x=x+1 $O(1)$
  - (b) for(i=0; i<n; i++) x=x+1 $O(n)$
  - (c) for(i=0; i<n; i++) $O(n^2)$
    for (j=0; j<n; j++) x=x+1

- Classify of algorithm
  - Constant $O(c)$
  - Logarithmic $O(\log_2 n)$
  - linear $O(n)$
  - Polynomial $O(n^m)$
  - Exponential $O(2^n)$



Comparison of T(n)

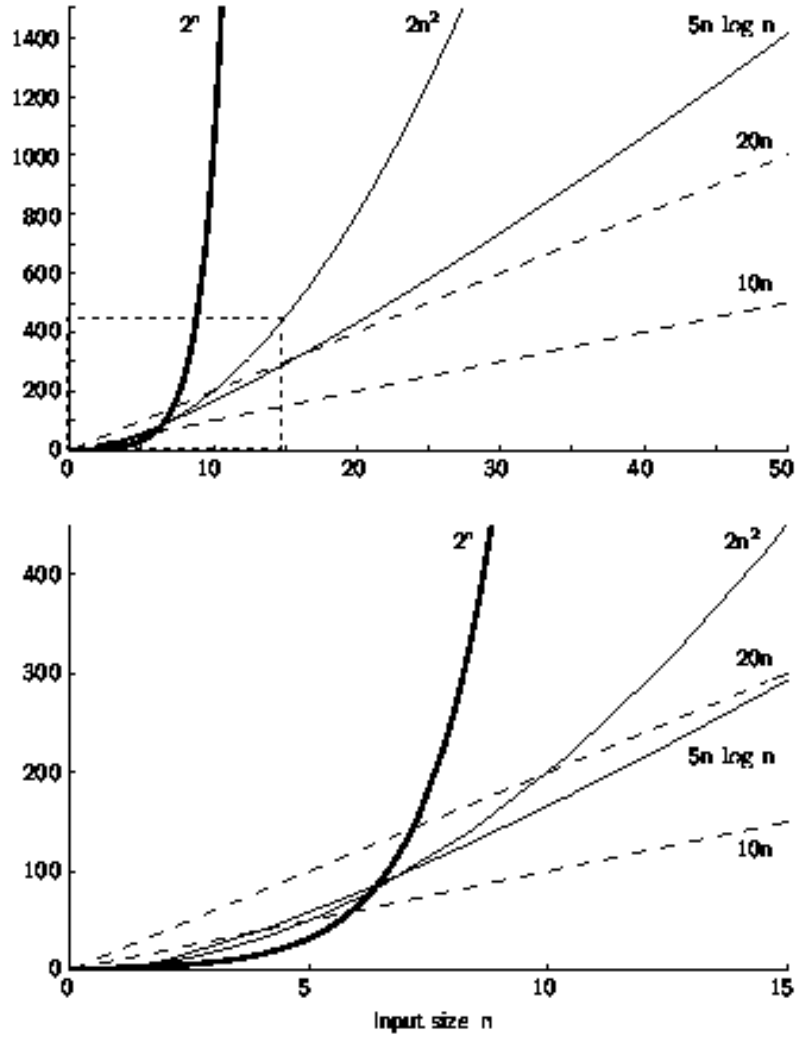


$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$

Growth Rate Graph

- Reorder the following efficiencies from smallest to largest

  2^n, n!, n^5, 10000, $n\log_2(n)$



Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for some constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Time Complexity Examples (1)

Example: **a = b**;

This assignment takes constant time, so it is O(1).

Example:

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

What it the big-oh?

$O(n)$, even though the value of sum is n^2.

Time Complexity Examples (2)

Example :

```
sum = 0;     //O(1)
for (j=1; j<=n; j++)   //double for loop O(n^2)
  for (i=1; i<=j; i++)
    sum++;
for (k=0; k<n; k++)  // O(n)
  A[k] = k;
```

Time Complexity Examples (3)

Example:

```
sum1 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum1++;
```

The only difference is
j<=n vs. j<= i.

First loop, sum is n^2.

```
sum2 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    sum2++;
```

Second loop, sum is $(n+1)(n)/2$.

Both are $O(n^2)$.

Other Control Statements

- **while** loop: Analyze like a **for** loop.
- if statement: Take greater complexity of then/else clauses.

  The probabilities for the then/else clauses being executed must be independent of *n*.

switch statement: Take complexity of most expensive case.

The probabilities of the various clauses being executed must be independent of *n*.

Subroutine call: Complexity of the subroutine.

Best, Worst, Average Cases

- Not all inputs of a given size take the same time to run.
- Sequential search for *K* in an array of n integers:
  - Begin at first element in array and look at each element in turn until *K* is found

Best case: Find at first position. Cost is 1 compare.

Worst case: Find at last position. Cost is *n* compares.

Average case: $(n+1)/2$ compares if we assume the element with value *K* is equally likely to be in any position in the array.

Time complexity Exercise 1

```
void BubbleSort( int A[], int n )
    {    int i, j, temp;
(1)      for (i=0; i<n-1; i++)
(2)          for (j=n-1; j>=i+1; j--)
(3)              if (A[j-1]>A[j]) {
(4)                  temp=A[j-1];
(5)                  A[j-1]=A[j];
(6)                  A[j]=temp;
                 }
    }
```

Big-oh? $O(\sum_{i=0}^{n-2}(n-i-1)) \le O(n(n-1)/2) = O(n^2)$

Time complexity Exercise 2: Recurrence Relations

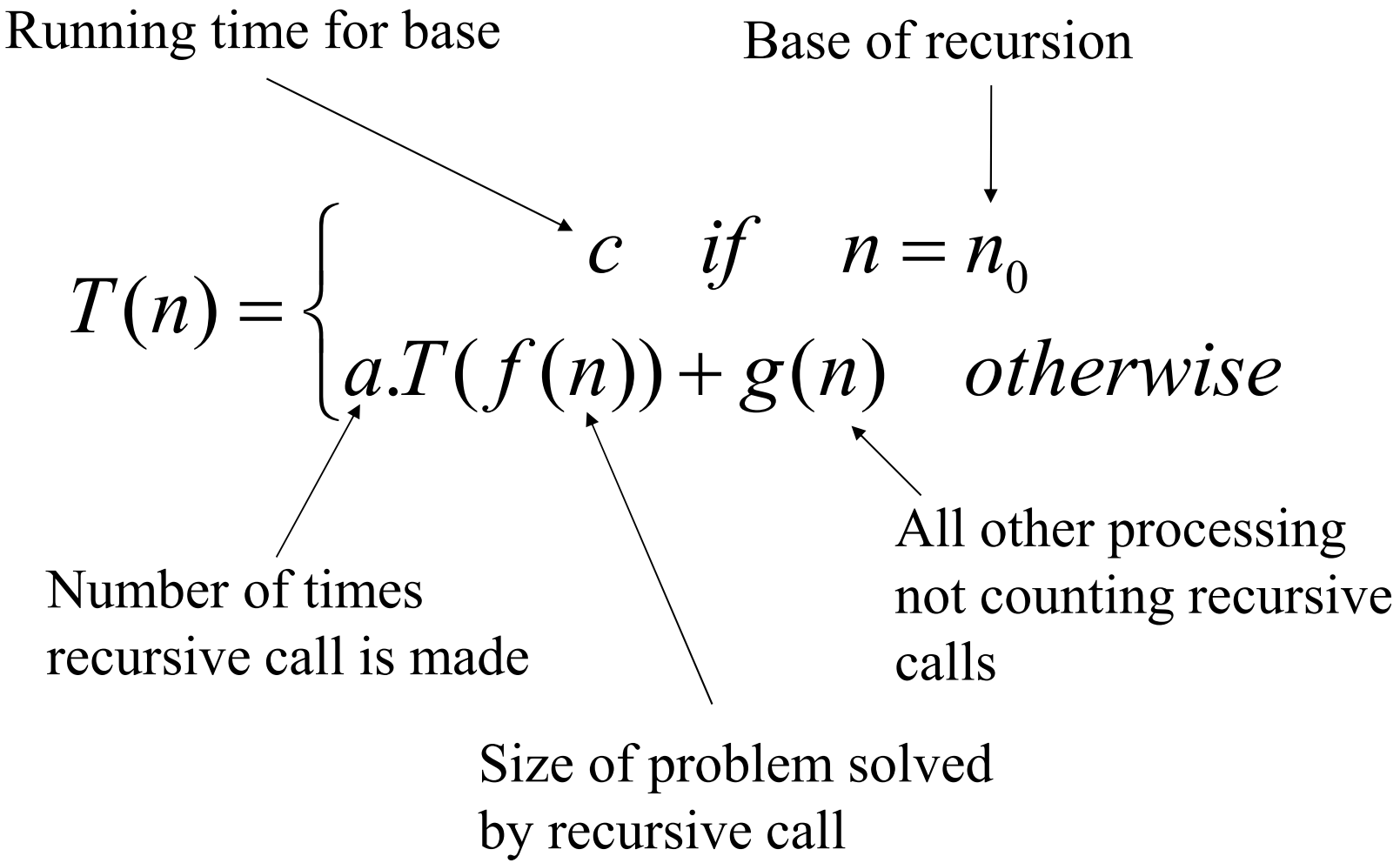
The Recurrence Process of Factorials n!

Pseudocode (recursive):

```
function factorial is:
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × ··· × 1]

    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]

end factorial
```

```
long  fact ( int  n)
{   if ( n==0 ) || ( n ==1 )
        return( 1 );
    else
        return( n * fact(n – 1));
}
```

Solving Recurrence Relations



Substitution method

$$T(n) = \begin{cases} c, & n = 1 \\ T(n-1) + d, & n > 1 \end{cases}$$

T(n) = T(n-1) + d

T(n-1) = T(n-2) + d

T(n-2) = T(n-3) + d

…

T(2) = T(1) + d

T(1) = c

T(n) = T(n-1) + d

= (T(n-2) + d) + d

= T(n-2) + 2d

= (T(n-3) + d) + 2d

= T(n-3) + 3d

= …

T(n) = T(1) + d(n-1)

= dn + c – d

= O(n)

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Table lookup

  Factorials

Disk-based Space/Time Tradeoff Principle:

The larger you make the disk storage requirements, the faster your program will run.

Space Complexity

Space complexity can also be analyzed with asymptotic complexity analysis.

Time: Algorithm

Space: Data Structure

Incremental Development

How to fail at implementing your project:

1.Write the project

2.Debug the project

How to succeed:

1.Write the smallest possible kernel

2.Debug the kernel thoroughly

3.Repeat until completion:

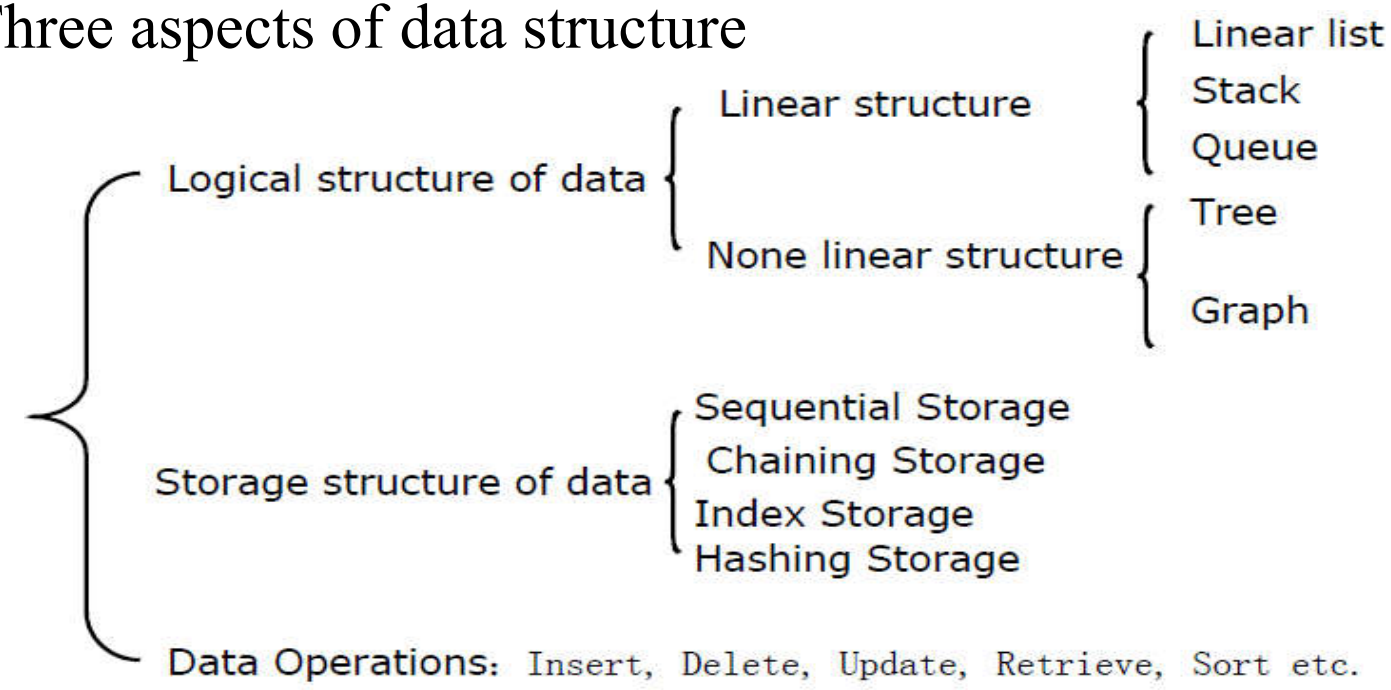
   i. Add a functional unit

   ii. Debug the resulting program

4.Have a way to track details

Summary

- Fundamental and important concepts
  - ADT, Data structure, and Algorithms
- Algorithm analysis and how to calculate the complexity of an algorithm
- Basic process and method to solve a problem.
- Three aspects of data structure



Lists

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com

outline

- Definition of ADT
- Array-based List (Sequential List)
- Singly Linked List
- Circular Linked List
- Doubly Linked List
- Applications

Lists

- A list is a finite, ordered sequence of data items.
  - "Ordered" in this definition means that each element has a position in the list.
  - We will not use "ordered" in this context to mean that the list elements are sorted by value.
- Important concept: List elements have a position.
- Notation: $<a_0, a_1, \ldots, a_{n-1}>$
- What operations should we implement?

List Implementation Concepts

- Operations will act relative to the current position.
- Our list implementation will support the concept of a current position.

- Example: <20, 23 | 12, 15>
  - we use a bar to its left to indicate current position in the notation
  - indicate the list of four elements
  - 12 is the “current” node

List ADT

```
public interface List<E> {  //type name E,
// Clear contents from the list, to make it empty
  public void clear();
// Insert an element at the current location.
public void insert(E item);
// Append an element at the end of the list
  public void append(E item);
// Remove and return the current element
  public E remove();
//Set the current position to the start of the list
public void moveToStart();
 // Set the current position to the end of the list
  public void moveToEnd();
```

List ADT

```
 // Move the current position one step left. No change if
    already at beginning.
  public void prev();
// Move the current position one step right. No change if
  public void next();
// Return: The number of elements in the list.
  public int length();
// Return: The position of the current element.
  public int currPos();
// Set current position. pos: The position to make current.
  public void moveToPos(int pos);
// Return: The current element.
  public E getValue();
}
```

This says nothing about HOW the list is implemented!

List ADT Examples

List: <12 | 32, 15>

L.insert(99);

Result: <12 | 99, 32, 15>

Iterate through the whole list:

```
for (L.moveToStart(); L.currPos()<L.length(); L.next()) {
  it = L.getValue();
  doSomething(it);
}
```

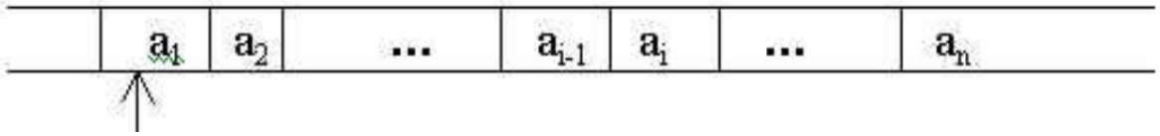
List Find Function

```
//return True if k is in list L,
// false otherwise */

public static boolean find(List<Integer> L, int k) {
  for (L.moveToStart();
          L.currPos()<L.length(); L.next())
    if (k == L.getValue()) return true;
  return false;          // k not found
}
```

Array-Based List Implementation

Array-Based List Implementation

- **Array_Based List** : The elements are stored in a consecutive storage area one by one



Basic address of the list, which is the starting address

Array-Based List: Notes

- With ordered pair $<a_{i-1}, a_i>$ to express "Storage is adjacent to"

$$loc(a_i) = loc(a_{i-1}) + C$$

- Unnecessary to store logic relationship
- First data component location can decide all data elements locations

$$LOC(a_i) = LOC(a_1) + (i-1) \times C$$

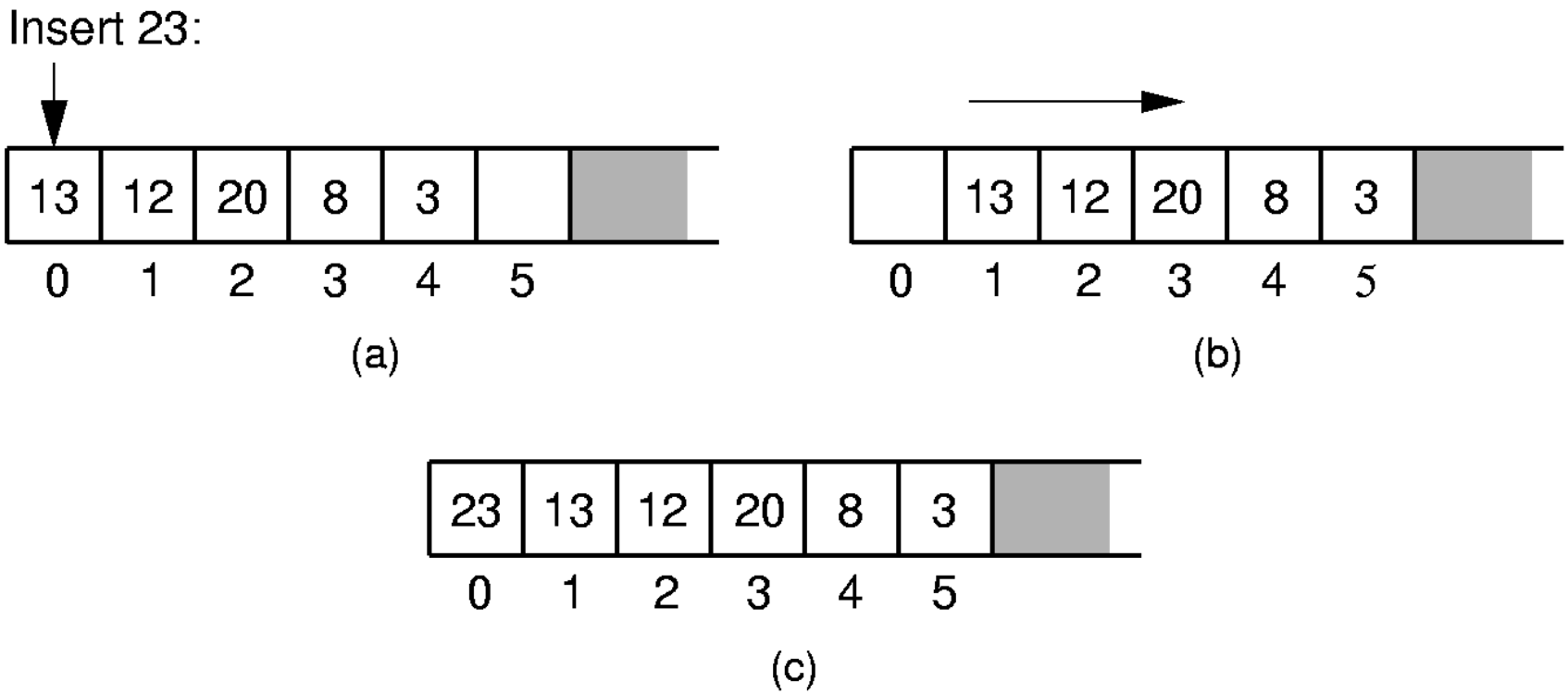
Array-Based List Class

- public void clear() { listSize = curr = 0; }
- public void moveToStart() { curr = 0; }
- public void moveToEnd() { curr = listSize; }
- public void prev() { if (curr != 0) curr--; }
- public void next() { if (curr < listSize) curr++; }
- public int length() { return listSize; }
- public int currPos() { return curr; }

Array-Based List Class (3)

```
public void moveToPos(int pos) {
  assert (pos>=0) && (pos<=listSize) :
        "Position out of range";
  curr = pos;
}

public E getValue() {
  assert (curr >= 0) && (curr < listSize) :
        "No current element";
  return listArray[curr];
}
```

Array-Based List Insert



Insert

```
//Insert "it" at current position,starting
  from i=0

public void insert(E it) {
  assert listSize < maxSize :
         "List capacity exceeded";
  for (int i=listSize; i>curr; i--)
    listArray[i] = listArray[i-1];
  listArray[curr] = it;
  listSize++;
}
```

The time complexity of Insert

- Analysis of the time complexity
  - Best case ($i=n+1$):
    - Basic operation 0 time, $O(1)$
  - Worst case ($i=1$):
    - Basic operation: n times, $O(n)$
  - Average case ($1 \le i \le n+1$):

$$\sum_{i=1}^{n+1} p_i (n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2} = O(n)$$

  - Time complexity: $O(n)$

Append

```
public void append(E it) { // Append "it"
  assert listSize < maxSize :
         "List capacity exceeded";
  listArray[listSize++] = it;
}
```

Remove

```
/** Remove and return the current element */
public E remove() {
  if ((curr < 0) || (curr >= listSize))
    return null;
  E it = listArray[curr];
  for(int i=curr; i<listSize-1; i++)
    listArray[i] = listArray[i+1];
  listSize--;
  return it;
}
```

Time complexity: O(n)

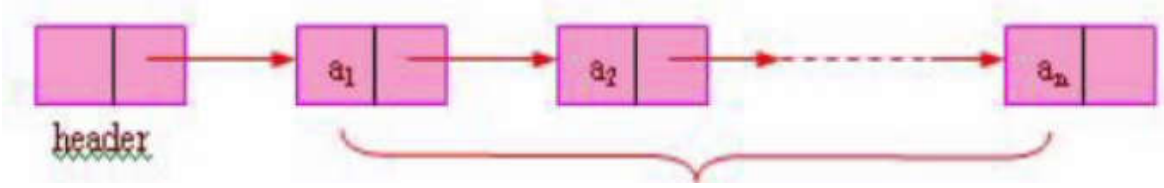
Summary

- Advantages
  - Stores a collection of items contiguously.
    - Stores no relations
    - Access randomly
- Disadvantages
  - Need to shift many elements in the array whenever there is an insertion or deletion.
  - Need to allocate a fix amount of memory in advance.

Singly Linked List

Introduction

- Array
  - successive items locate a fixed distance
- disadvantage
  - data movements during insertion and deletion
  - waste space in storing *n* ordered lists of varying size
- possible solution
  - linked list

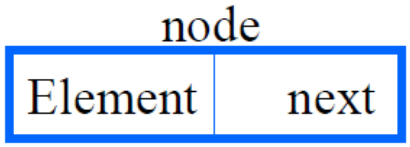


list

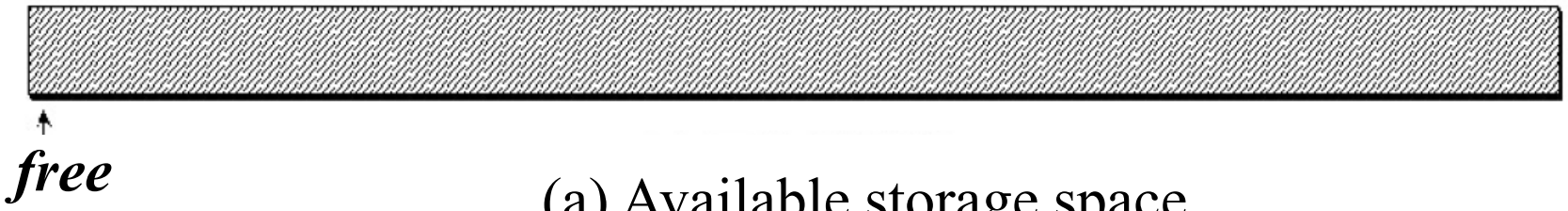
Linked List

- A linked list is made up of a series of objects, called the nodes of the list.

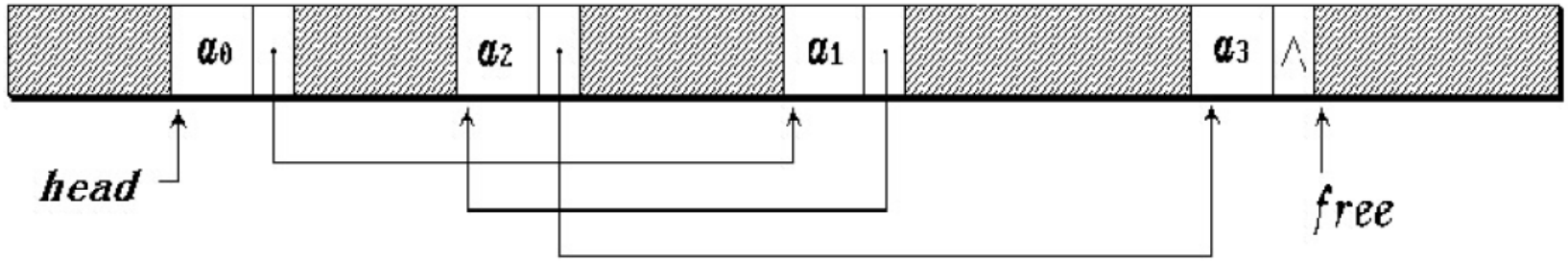
- The linked list uses dynamic memory allocation



Singly linked list storage mapping

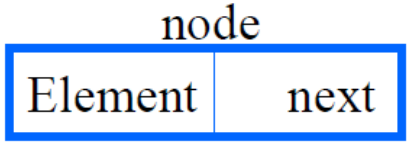


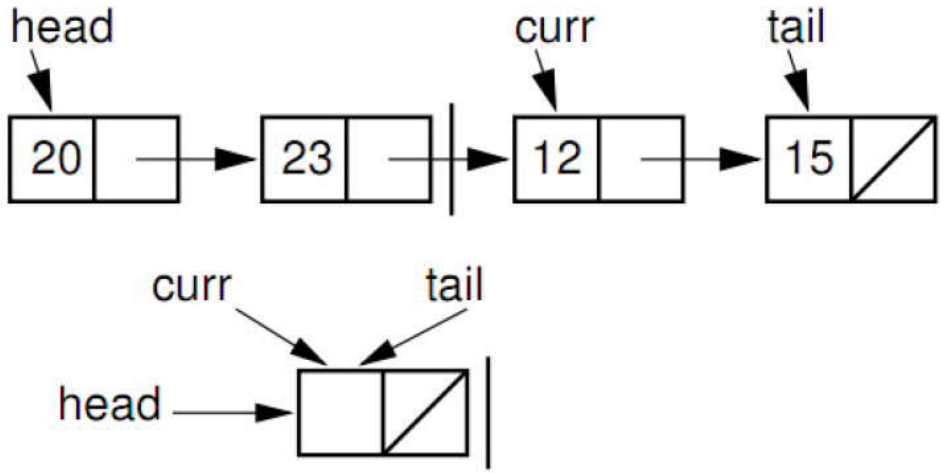
(a) Available storage space



(b) The linked list after a period of running time

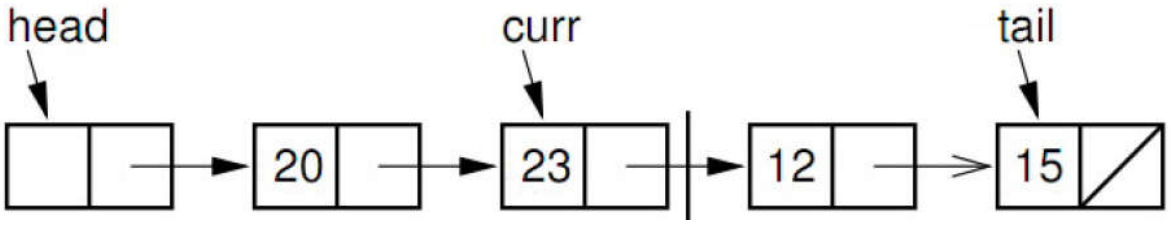
Singly linked List





- *head*: a pointer point to the list's first node.
- *tail*: a pointer is kept to the last node of the list.
- *curr*: a pointer indicate the current element.
- *cnt*: the length of the list

Singly linked List



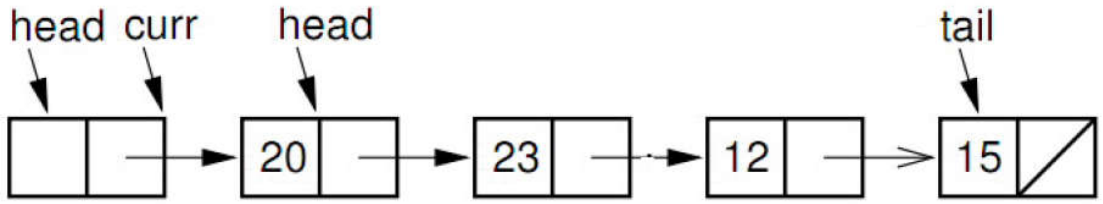
- *header node*: an additional node before the first element node of the list.
- The header node saves coding effort because
  - we no longer need to consider special cases for empty lists or
  - when the current position is at one end of the list.

Linked List Class (1)

```
Link<E>* head; // Pointer to list header
Link<E>* tail; // Pointer to last element
Link<E>* curr; // Access to current element
int cnt; // Size of list

void init() { // Intialization method
    curr = tail = head = new Link<E>; // head node
    cnt = 0;
}
```

Linked List Class (2)

```
void removeall() { // Return link nodes to free store
    while(head != NULL) { // from the head to the last
        curr = head;
        head = head->next;
        delete curr;
    }
}
```



Array-based linked list

- Storage structure of data
  - type definition

```
#define  max  100
struct LIST{
    ElemType elements[max];
    int last;
};
```

  - position type:

    typedef int p;

- Example: LIST L
  - L.elements[p] //the *pth* element of L

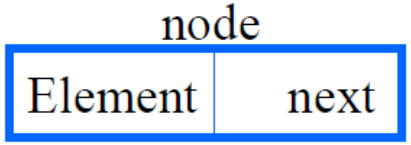
Singly linked list

- Storage structure of data
  - node type definition

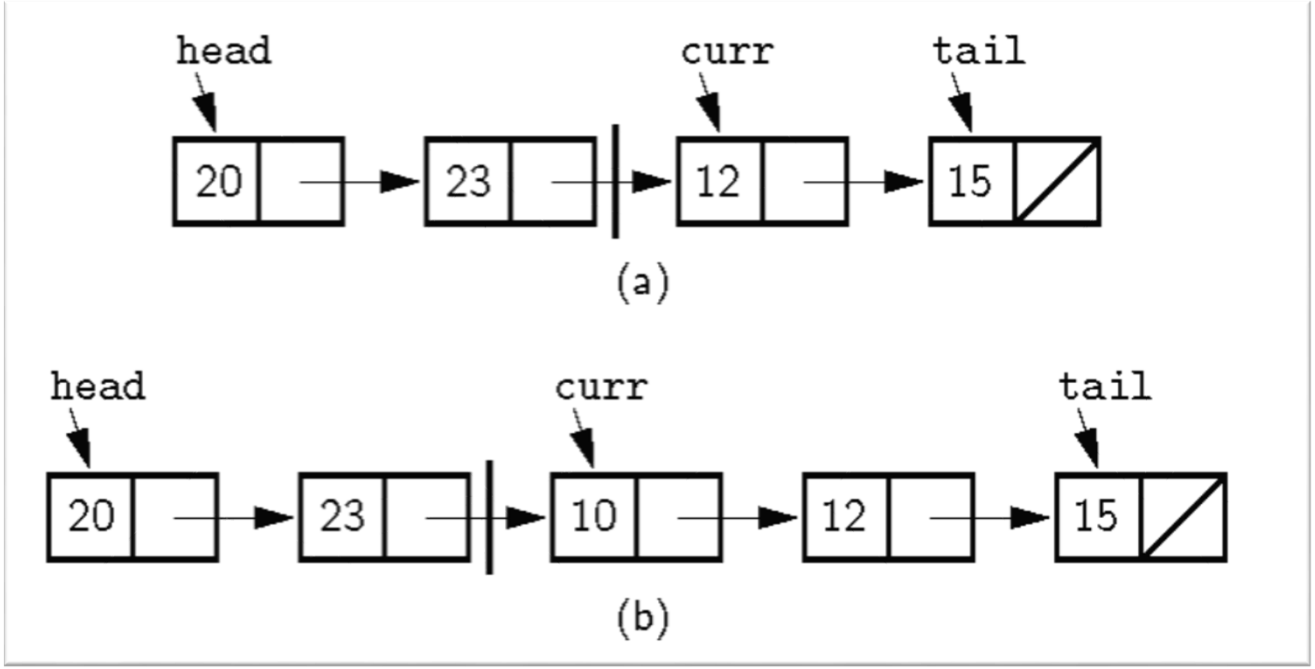
```
struct   celltype  {
        ElemType  element ;
        celltype  *next ;
} ; /*node type*/
```

- Examples:
  - typedef celltype *LIST; //a list type
  - typedef celltype *position; //a position type

- Node structure
  - data field
  - pointer field
  - pointer to the node

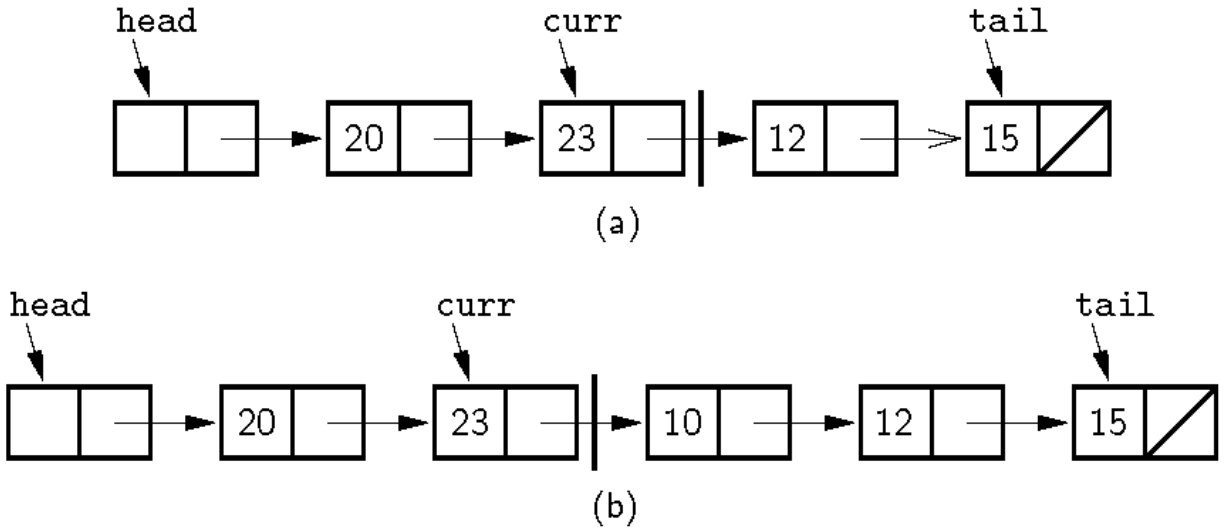


Insertion



- **A faulty linked-list implementation where *curr* points directly to the current node.**

Insertion

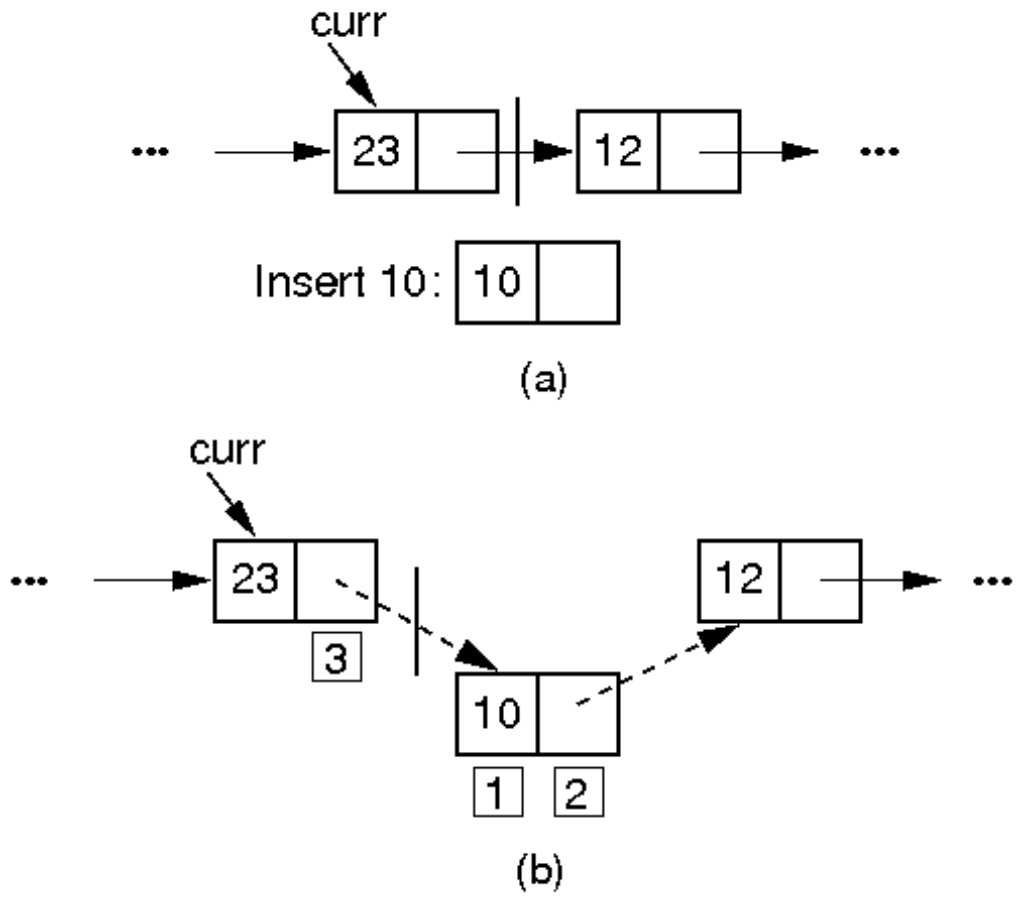


- Insertion using ahead node, with curr pointing one node ahead of the current element.

Insertion: The linked list insertion process

Inserting a new element is a three-step process:

- **First**, the new list node is created and the new element is stored into it.
- **Second**, the next field of the new list node is assigned to point to the current node (the one after the node that *curr* points to).
- **Third**, the next field of node pointed to by *curr* is assigned to point to the newly inserted node.



Exercise: Insertion (New tail) and Append

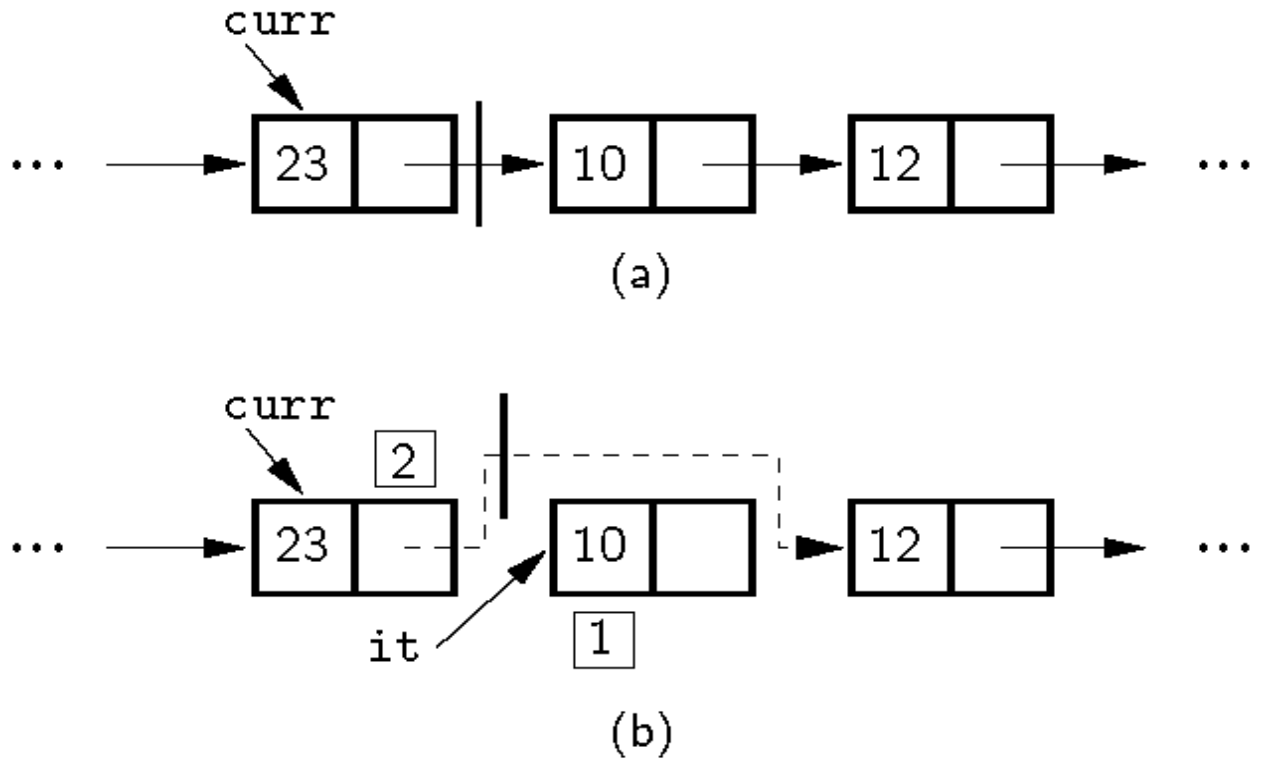
Insertion

```
// Insert "it" at current position

void insert(const E& it) {
    curr->next = new Link<E>(it, curr->next);
    if (tail == curr) tail = curr->next; // New tail
    cnt++;
}
```

Append

```
// Append "it" to list

void append(const E& it) {
    tail = tail->next = new Link<E>(it, NULL);
    cnt++;
}
```

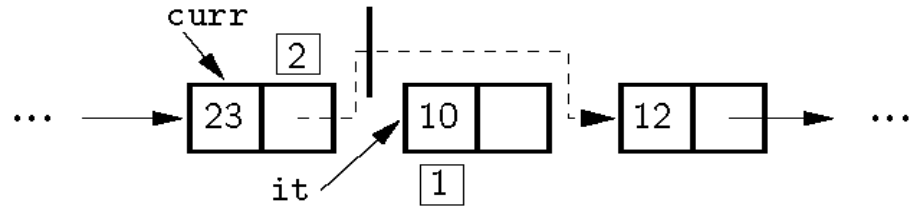
Removal



- The linked list removal process.

Remove

```
// Remove and return current element
E remove() {
    Assert(curr->next != NULL, "No element");
    E it = curr->next->element; // Remember value
    Link<E>* ltemp = curr->next; // Remember link node
    if (tail == curr->next) tail = curr; // Reset tail
    curr->next = curr->next->next; // Remove from list
    delete ltemp; // Reclaim space
    cnt--; // Decrement the count
return it;
}
```



MoveToStart & MoveToEnd

```
void moveToStart() // Place curr at list start
    { curr = head; }

void moveToEnd()  // Place curr at list end
    { curr = tail; }
```

Prev

```
// Move curr one step left; no change if already at front
void prev() {
    if (curr == head) return;  // No previous element
    Link<E>* temp = head;
    // March down list until we find the previous element
    while (temp->next!=curr) temp=temp->next;
    curr = temp;
}
```

Next / Length

```
// Move curr one step right; no change if already at
   end
void next()
   { if (curr != tail) curr = curr->next; }

// Return length
int length() const
   { return cnt; }
```

Get/Set Position

```
// Return the position of the current element
int currPos() const {
    Link<E>* temp = head;
    int i;
    for (i=0; curr != temp; i++)     // counting
          temp = temp->next;
    return i;
}
```

Move to a Position

```
// Move down list to "pos" position
void moveToPos(int pos) {
    Assert ((pos>=0)&&(pos<=cnt), "Position out of
range");
    curr = head;
    for(int i=0; i<pos; i++) curr = curr->next;
}
```

GetValue

```
// Return current element

const E& getValue() const{
    Assert(curr->next != NULL, "No value");
    return curr->next->element;
}
```

Comparison of Implementations

Array-Based Lists:

- Insertion and deletion are O(n).
- Prev and direct access are O(1).

- Array must be allocated in advance.
- Fixed
- No overhead if all array positions are full.

- Random access

Linked Lists:

- Insertion and deletion are O(1).
- Prev and direct access are O(n).

- Space grows with number of elements.
- Extendable
- Every element requires overhead.

- Sequential access

Space Example

- Array-based list: Overhead is one pointer (4 bytes) per position in array –whether used or not.

- Linked list: Overhead is two pointers per link node
  - one to the element, one to the next link
- Data is the same for both.
- When is the space the same?
  - When the array is half full

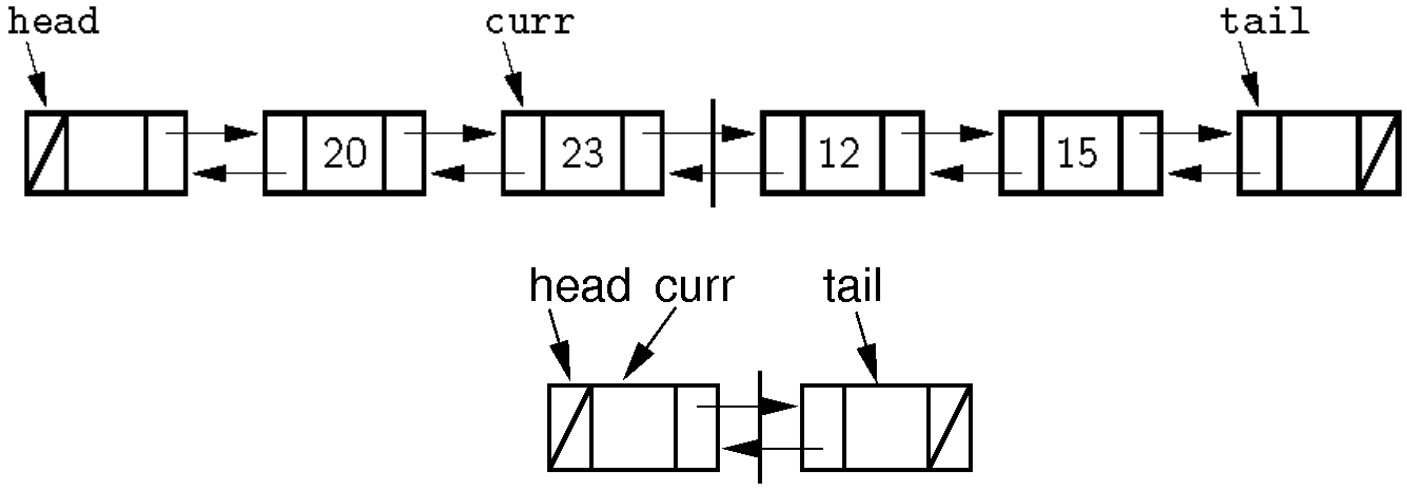
Doubly Linked List & Circularly Linked Lists

Doubly Linked Lists

- Singly Linked Lists
  - The singly linked list allows for direct access from a list node **only to the next node** in the list.
- Doubly Linked Lists
  - A doubly linked list allows convenient access from a list node **to the next node and also to the preceding node** on the list.
- How to accomplish?
  - The doubly linked list node accomplishes this in the obvious way **by storing two pointers**:
  - one to the node following it (as in the singly linked list), and
  - a second pointer to the node preceding it.

Doubly Linked Lists

- Storage structure of data
  - Node structure
  - Advantage: searching in two way/direction
  - Disadvantage: additional space used.



Doubly Linked Lists

- Node type:

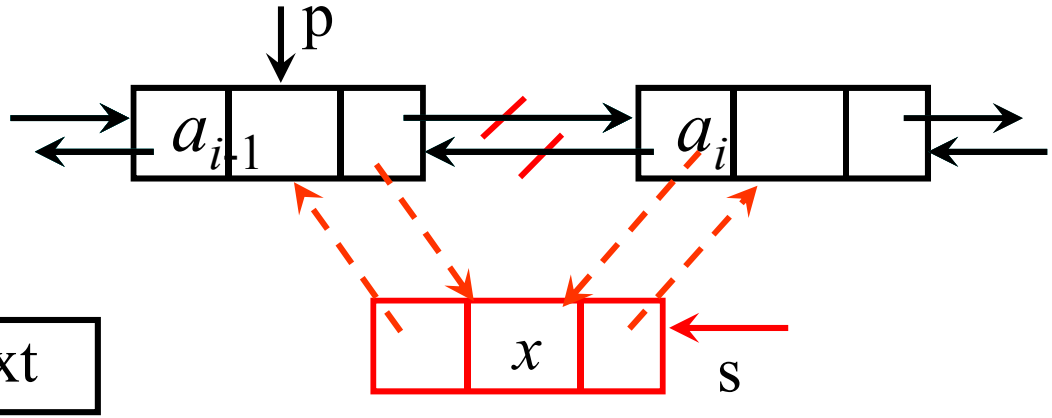
```
struct  dcelltype  {
    ElemType  data ;
    dcelltype  *next, *prior ;
} ;
```

- List type:
  - typedef dcelltype *DLIST ;
- Position type:
  - typedef dcelltype *position ;

Insert

```
// insert element x in position p
void  Insert( ElemType x, position p,  DLIST  &L )
{      s = new dcelltype;
       s->data = x;
       s->prior=p;
       s->next=p->next;
       p->next->prior=s;
       p->next=s;
}
```

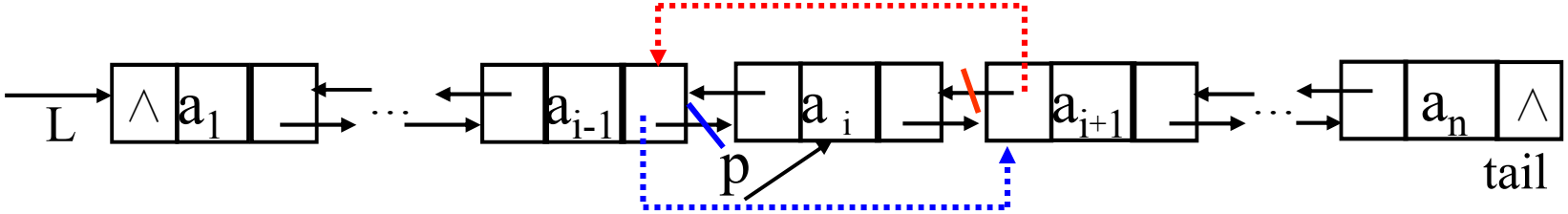
O(C)



Delete

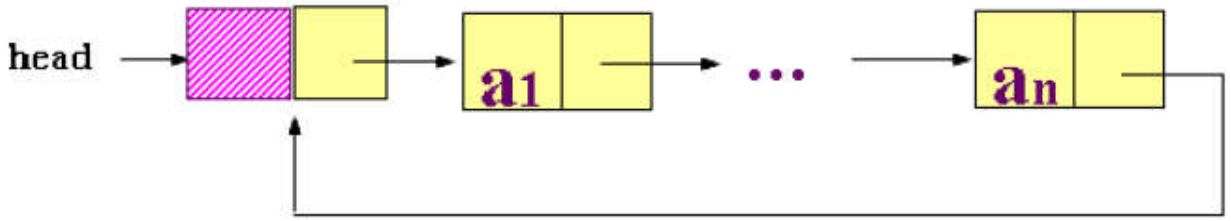
```
// delete the element at position p
void Delete( position  p, DLIST &L)
{      if (p->prior!=NULL)
            p→prior→next = p→next;
       if (p→next!=NULL)
            p→next →prior = p→prior;
      delete p;
}
```

O(C)



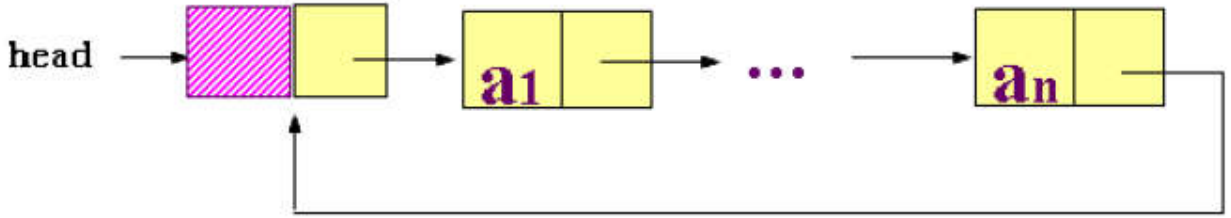
Circularly Linked Lists

- Singly Linked Lists
  - the last node contain a NULL pointer
- Circularly Linked Lists
  - the last node contains a pointer to the first node
- Advantage
  - start from any node, can access the others.

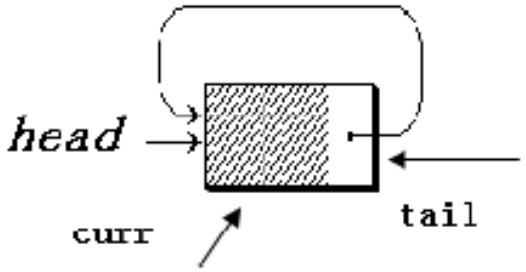


Example of circular linked list

- Nonempty list



- Empty list

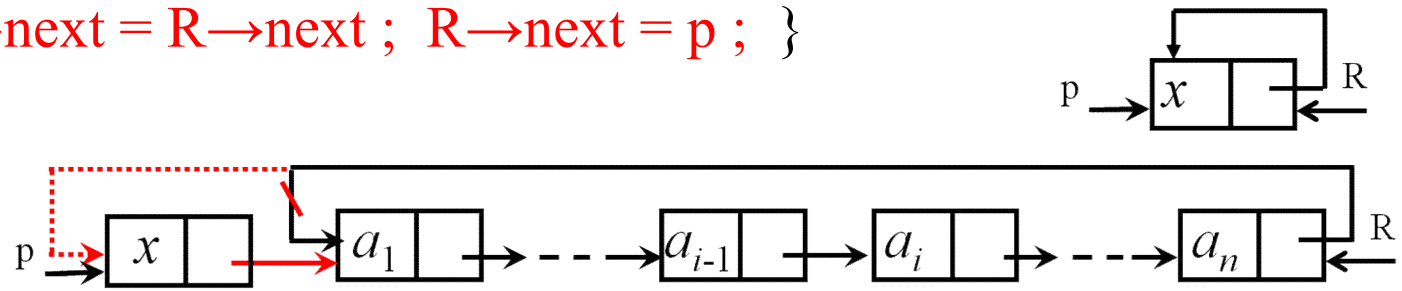


Operations: insert (1)

// Insert a node on the left of the List:

// LInsert(x,R) → Insert(x,First(R),R)

```
void  LInsert( Elementtype  x , LIST  R )
{     celltype  *p ;
      p = new  celltype ;
      p→data = x ;
      if ( R == NULL )
      {   p→next = p ;   R = p ;  }
      else
      {   p→next = R→next ;  R→next = p ;  }
}
```

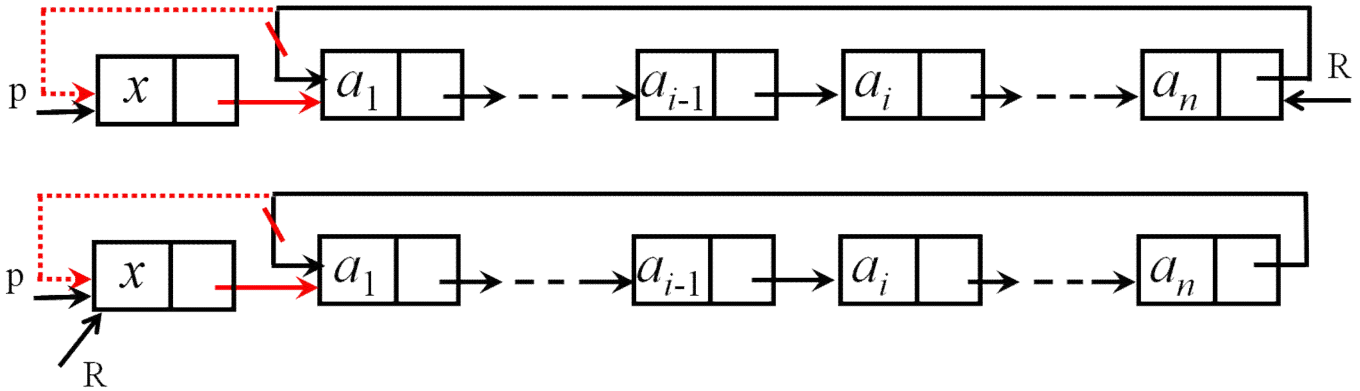


Operations: insert (2)

// Insert a node on the right of the List:

// RInsert(x,R))→ Insert(x,End(R), R)

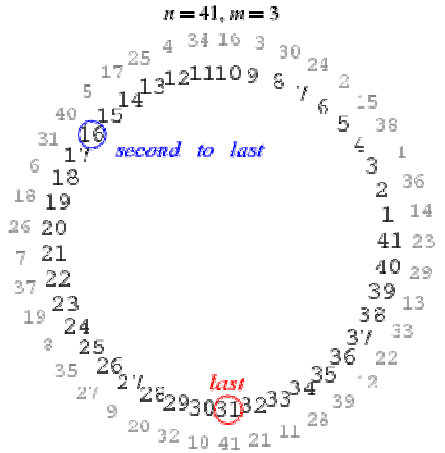
```
void   RInsert( ElemType  x , LIST  R )
{   LInsert ( x , R ) ;
    R = R→next ;
}
```



Example: Josephus problem

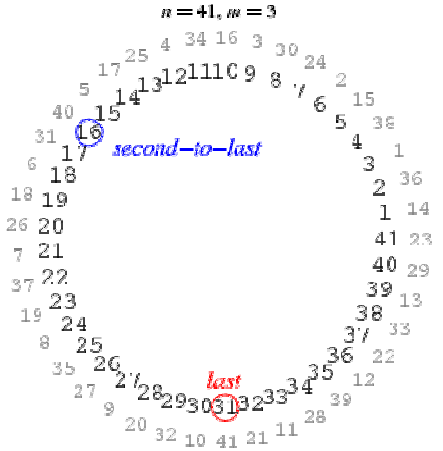
- Given a group of n men arranged in a circle under the edict that
- every m^{th} man will be executed going around the circle until only one remains,
- find the position in which you should stand in order to be the last survivor (Ball and Coxeter 1987).

- Data structure?
- Algorithm?



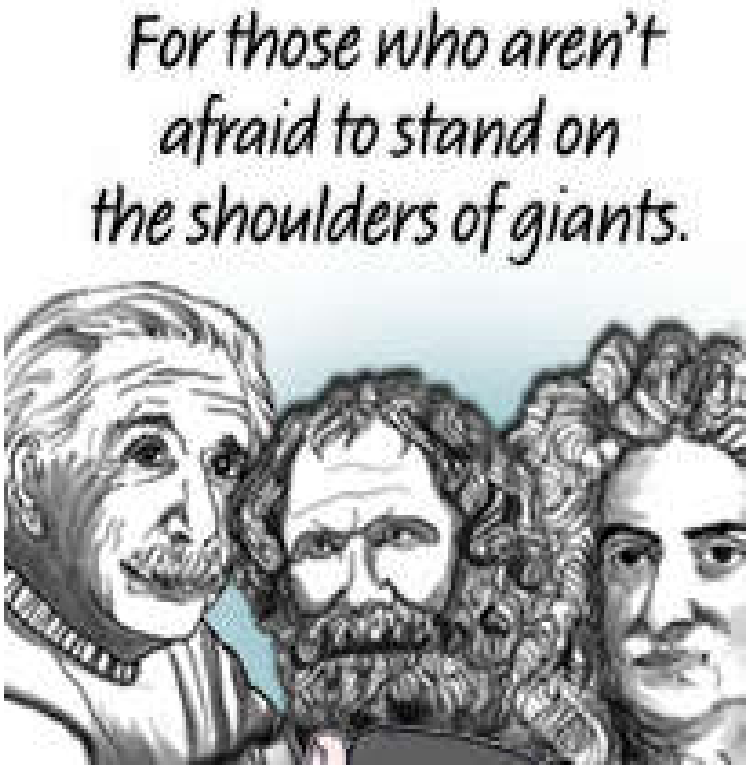
Solution to Josehus problem

```
void Josephus ( List  &Js, int n, int m)
{    celltype *p=Js, *pre=NULL;
     for (int i=0; i<n-1; i++) { // n-1
          for (int j=0; j<m-1; j++) { //count m
               pre=p; p=p->next;
           }
          cout<<"Out of line:"<<p->data<<endl;
          pre->next =p->next; delete p; // p is out of the list
          p=pre->next;  // next turn
     }
}
```



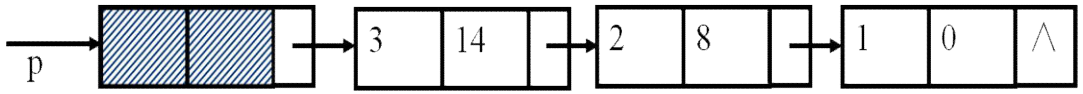
Exercise/Assigment

- Josephues problem use doubly linked list Algorithm



Application: polynomial algebraic operation

- Example: $p(x) = 3x^{14} + 2x^{8} + 1$
- Expressing the polynomial
  - Singly linked list



- Storage structure of node

```
struct polynode {
    int   coef ;  // coefficient
    int   exp ;    // exponent
    polynode  *link ;  //pointer to next
};  //node type
```

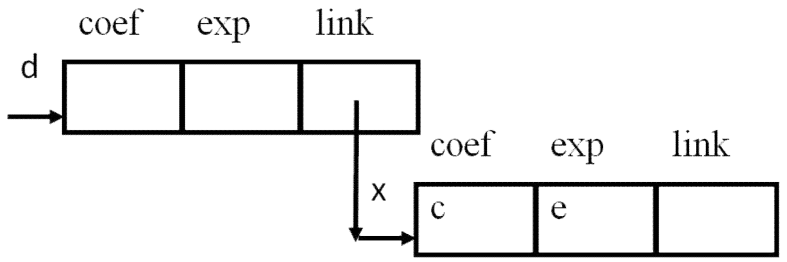
- Polynomial type:

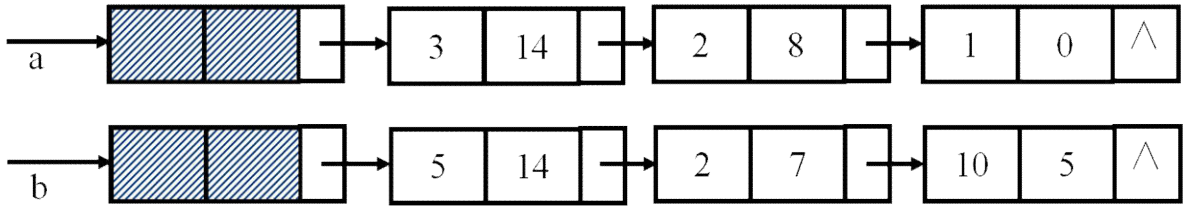
```
typedef  polynode *polypointer ;
```

Algorithm: attach

//construct a new node, where, coef=c， exp=e
//return the pointer to the node after attached to *p* node.

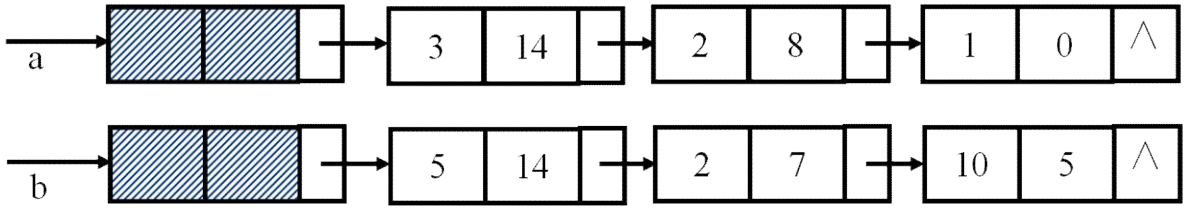
```
polypointer  Attch ( int  c , int e , polypointer  d )
{    polypointer   x  ;
     x = new   polynode ;
     x→coef = c ;
     x→exp = e ;
     d→link = x ;
     return  x ;
}
```





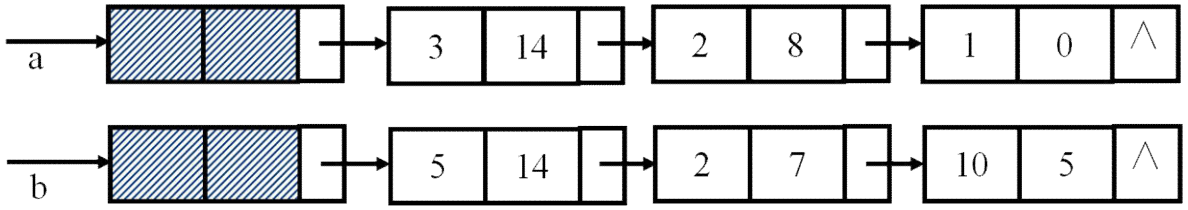
Polynomial adding to (1)

```
polypointer  PolyAdd ( polypointer  a , polypointer b )
{    polypointer  p, q, d, c ;
     int  y ;
     p = a→link ;   q = b→link ;
     c = new polynode ;    d = c ;
     while ( (p != NULL) && (q != NULL) )
          switch ( Compare ( p→exp, q→exp ) )
          {  case '=' :
                    y = p→coef + q→coef ;
                     if ( y )   d = Attch( y, p→exp, d ) ;
                     p = p→link ; q = q→link ;
                     break ;
               case '>':
                    d = Attch( p→coef, p→exp, d ); p = p→link ; break ;
```



Polynomial adding to (2)

```
   case '<':
          d = Attch( q→coef, q→exp, d ) ;
          q = q→link ;
          break ;
}
while ( p != NULL )
{    d = Attch( p→coef, p→exp, d ) ;
     p = p→link ;
}
while ( q !=NULL )
{    d = Attch( q→coef, q→exp, d ) ;
     q = q→link ;
}
```



Polynomial adding to (3)

```
        d→link = NULL ;
      p = c ;  c = c→link ;
      delete  p ;
      return  c ;
}
```

- Time complexity:

  O(m+n) , where,

  m, n are the highest power of the two polynomial respectively

- Other operations:

  AddPolyn (…)

  SubtractPolyn (…)

  MultiplyPolyn (…)

  PolynLength (P)

Stack

Instructor: Prof. Tianyi ZANG

tianyi.zang@gmail.com

TA: Wei QUAN, Mingrui SUN

School of Computer Science and Technology

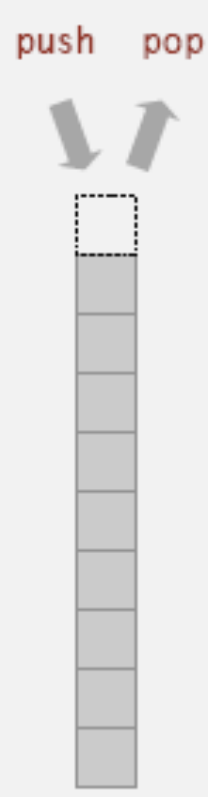
Harbin Institute of Technology

Outline

- Stack ADT
- Array-Based Stack
- Linked Stack
- Comparison of Array-Based and Linked Stacks
- Applications

Stacks

- Restricted form of list: Insert and remove only at front of list.
- The stack is a list-like structure in which
  - elements may be inserted or removed from only one end,
  - called the ***top*** of the stack.



- All access is restricted to the most recently inserted elements.
- LIFO: Last In, First Out.

Stack operations (ADT)

- Stack operations

  MakeNull (S) *//create an empty stack*

  Top (S) *// Return: A copy of the top element.*

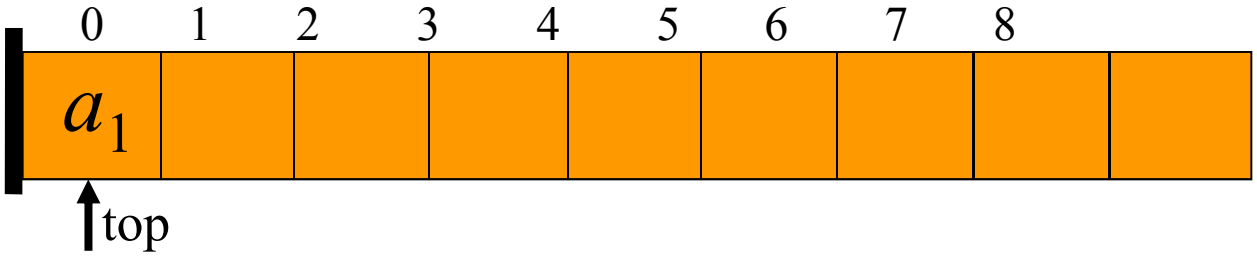
  Pop (S) *//remove and return the element most recently added, which is at the top of the stack*

  Push (x , S) *//push an element x onto the top of the stack*

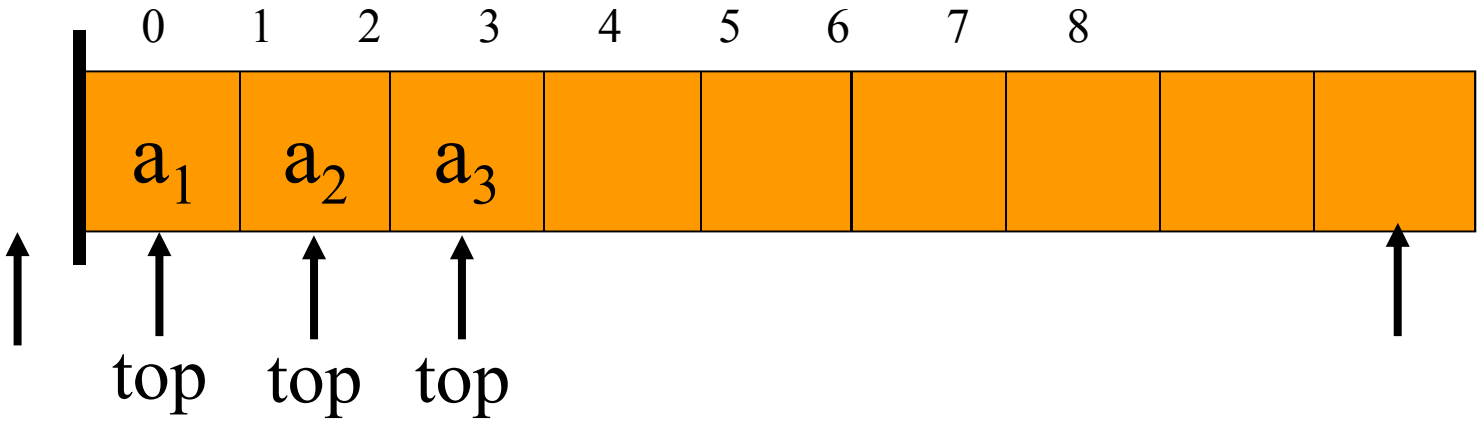
  Empty (S) *//is the stack empty?*

Array-Based Stack

- The array-based stack implementation is essentially a simplified version of the array-based list.
- Which end of the array should represent the top of the stack?
  - Make top be at **position 0 in the array.** inefficient
  - Have the top element be at **position n-1**
- **Top is the** array index of the top element in the stack.



Array-Based Stack



Push: top is increased by 1

Pop: top is decreased by1

Empty stack: top= −1

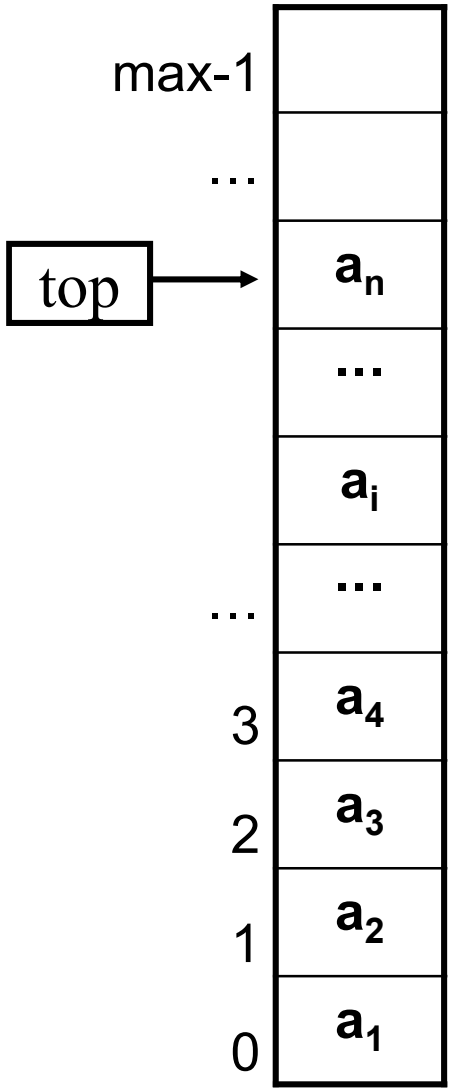
Full stack: top= MAX_SIZE

Array-Based Stack

- Storage structure

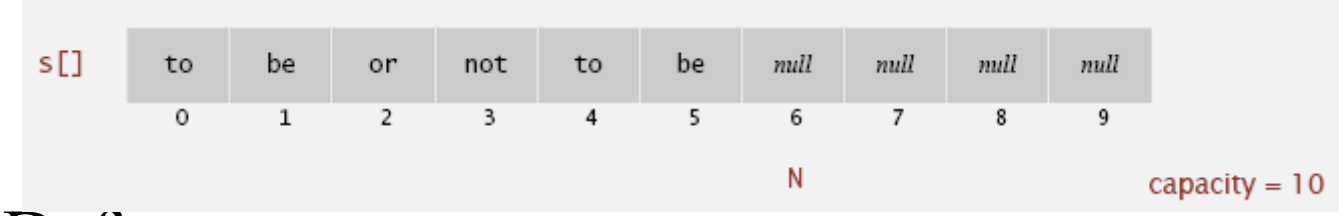
```
typedef   struct {
            ElemType  elements[max];
            int  top ;
  } STACK ;
 STACK  S
```

  - Capacity of the stack: max– 1 ;
  - Pointer to the top: S.top
  - Element at the top: S.elements[S.top] ;
  - Empty stack: S.top = -1 ;
  - Full stack: S.top = max–1 ;



Stack: array implementation

- Array implementation of a stack.
  - Use array s[] to store N items on stack.
  - push(): add new item at s[N].
  - pop(): remove item from s[N-1].

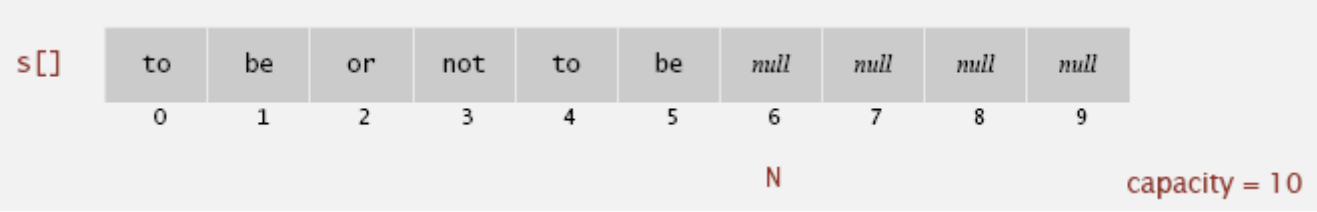


- Defect.
  - Stack overflows when N exceeds capacity.

Stack considerations

- Overflow and underflow.
  - Underflow: throw exception if pop from an empty stack.
  - Overflow: use resizing array for array implementation. [stay tuned]

- Null items. We allow null items to be inserted.



Operations of array-based stack

① void MakeNull(STACK &S)

```
{ S.top = -1 ; }
```

② Boolean Empty(STACK S)

```
{ if ( S.top < 0 )
        return TRUE
  else
        return FALSE ;
}
```

Operations of array-based stack

③ ElemTtype Top(STACK S)

```
{   if ( Empty( S )
        return NULL;
    else
        return ( S.elements[ S.top ] );
}
```

④ void Pop(STACK &S)

```
{
    if ( Empty (S ) )
        cout<< "The stack is empty." ;
    else
        S.top = S.top – 1 ;
}
```

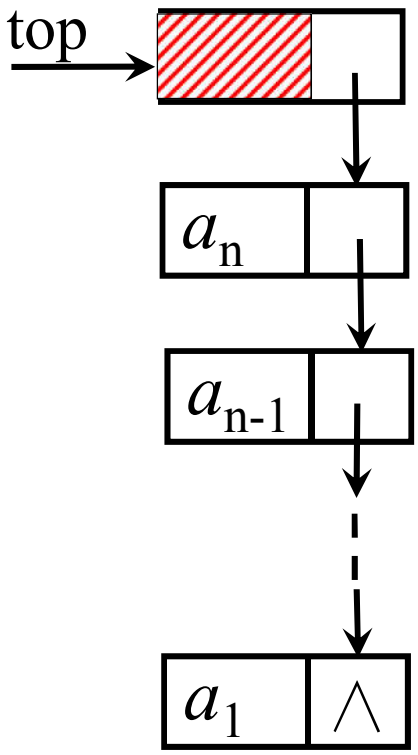
Operations of array-based stack

⑤ void Push (ElemTtype x, STACK &S)

```
{
    if ( S.top == max - 1 )
            cout<< “The stack is full.” ;
    else
    {       S.top = S.top + 1 ;
            S.elements[ S.top ] = x ;
    }
}
```

Linked Stack

- Elements are inserted and removed only from the head of the list.
- A header node is not used because no special-case code is required for lists of zero or one elements.
- The only data member is **top,**
  - **a pointer to the first (top) link node of** the stack.

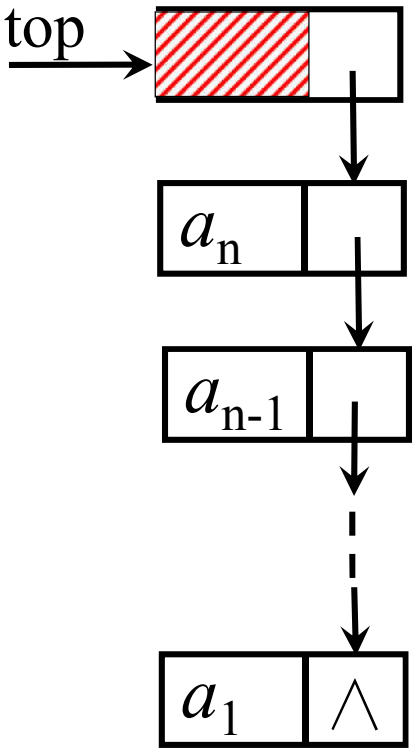


- Maintain pointer to first node in a linked list; insert/remove from front.

Linked Stack

- Storage structure

```
struct node{
        ElemType data;
        node *next;
};//node type

//Type of the stack
typedef node *STACK;
```



Operations of linked stack

①STACK MakeNull()

```
{    STACK s;
 s=new node;
/*s=(node *)malloc(sizeof(node));*/
 s->next=NULL;
 return s;
}
```

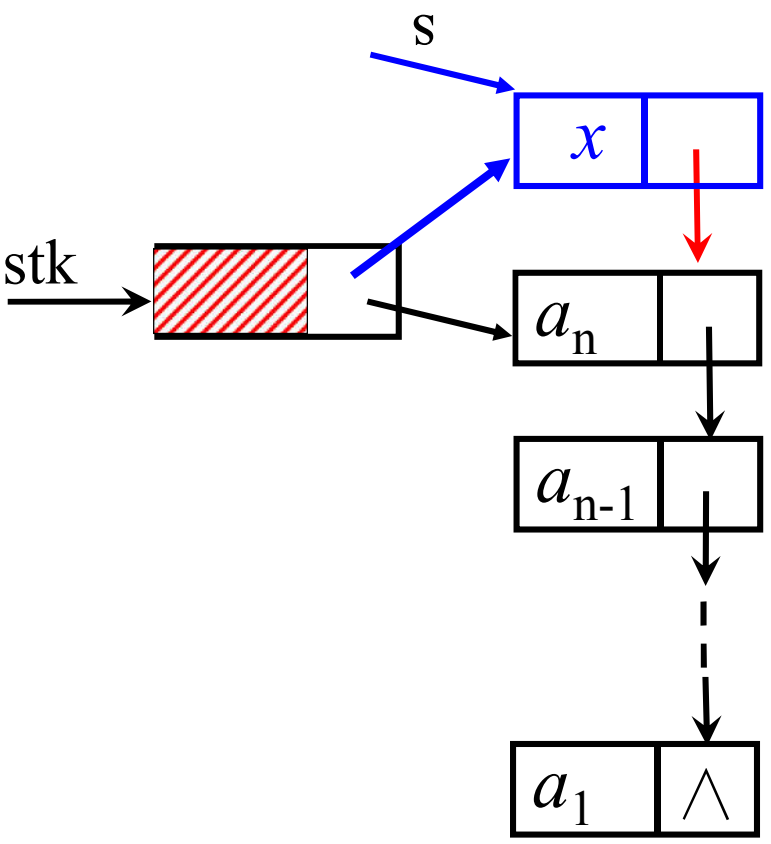
②boolean Empty(STACK stk)

```
{       if (stk->next)
            return FALSE;
    else
            return TRUE;
}
```

Operations of linked stack

③ void Push(Elementtype elm, STACK stk)

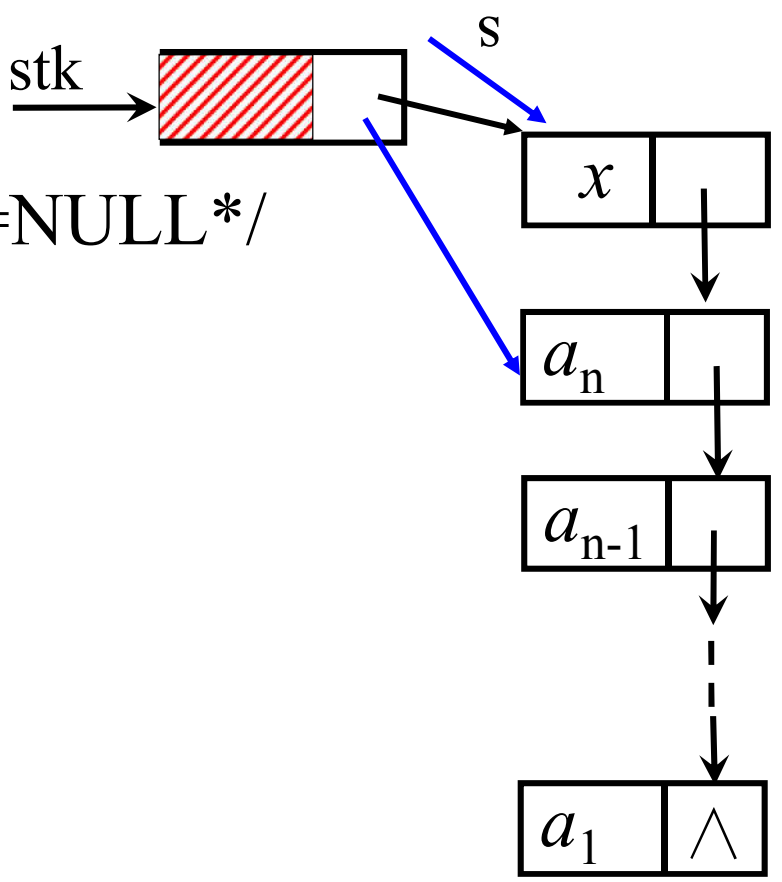
```
{
    STACK s;
    s=new node;
    s->data=elm;
    s->next=stk->next;
    stk->next=s;
}
```



Operations of linked stack

④ void Pop(STACK stk)

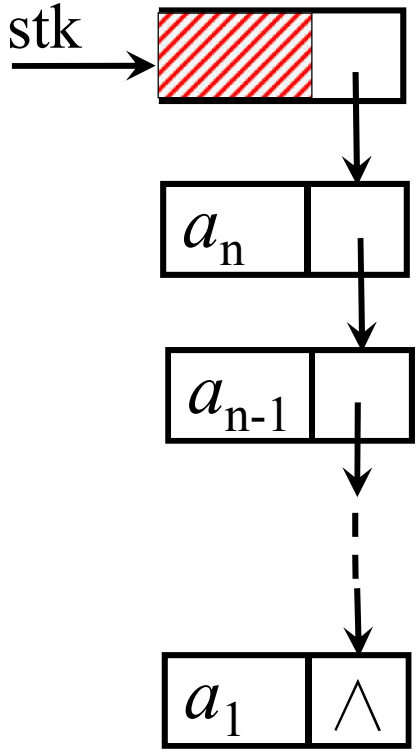
```
{     STACK s;
 if (stk->next){/*stk->next!=NULL*/
       s=stk->next;
       stk->next=s->next;
       delete s;   /*  free(s) */
  }
}
```



Operations of linked stack

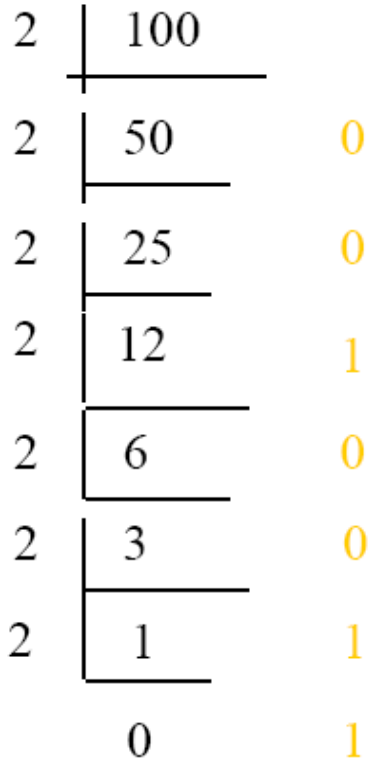
⑤ElemType Top(STACK stk)

```
{    if (stk->next)
        return (stk->next->data);
  else
        return NULL;
}
```



Application: Data conversion

- A simple algorithm uses the following law:
  - N=(N/ d) × d+ N% d eg: Convert decimal to binary
- (100)10 = (1100100)2
  - divide decimal number by 2 continuously
  - record the remainders until the quotient becomes 0
  - print the remainders in backward order .
- Stack is an ideal structure to implement this process.
- How to implement it?



A notation for arithmetic expressions:

A notation for arithmetic expressions:

- Infix notation: operators written between the operands
- Postfix notation (Reverse Polish Notation): operators written after the operands
- Prefix notation : operators written before the operands
- Examples:

INFIX	POSTFIX	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A–(B –(C – D))	A B C D - - -	- A - B - C D
A – B – C – D	A B - C - D -	- - - A B C D

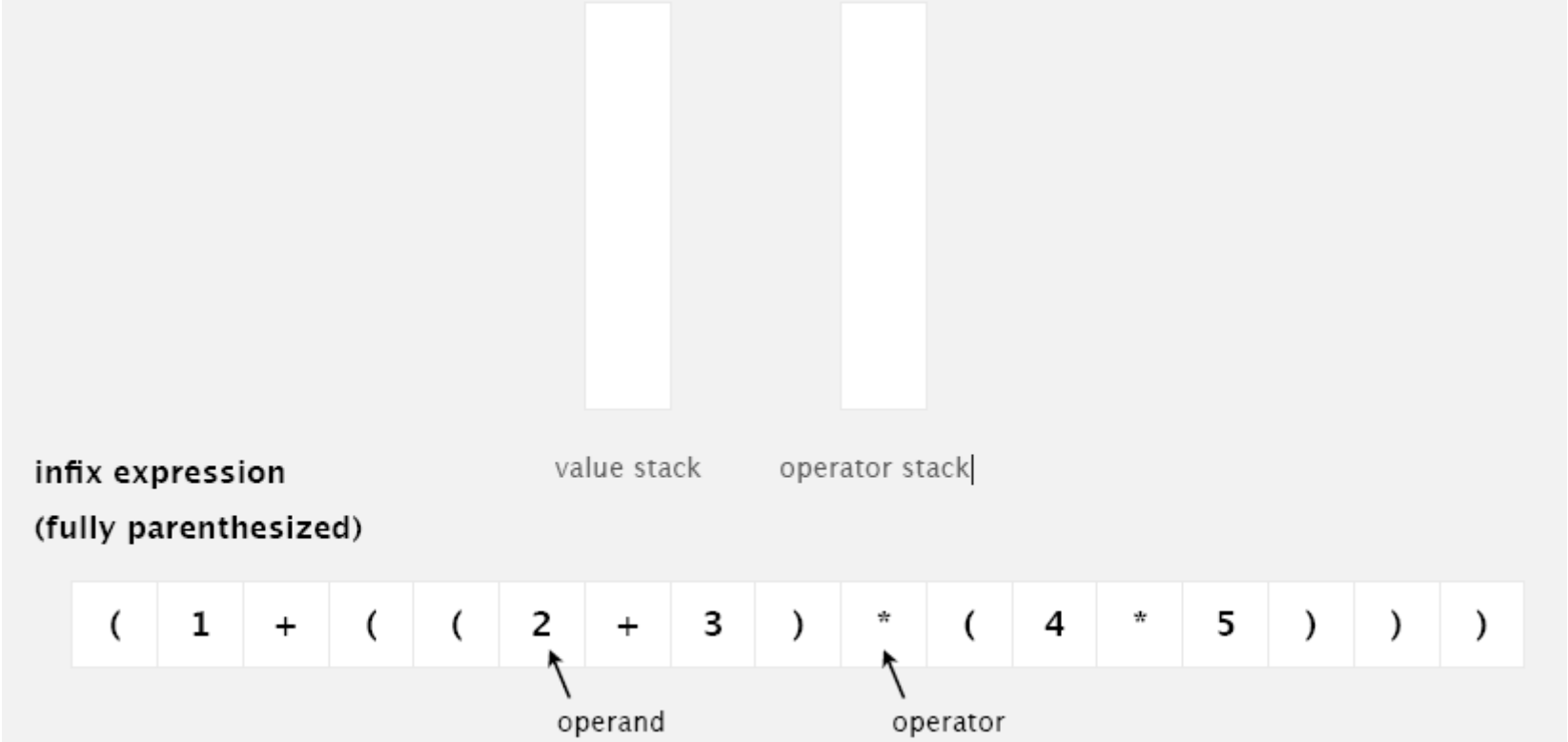
Applications:
Dijkstra's two-stack algorithm 1

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Applications:
Dijkstra's two-stack algorithm 2

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack operator stack

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Applications:
Dijkstra's two-stack algorithm 3

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack operator stack

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)
	↑															

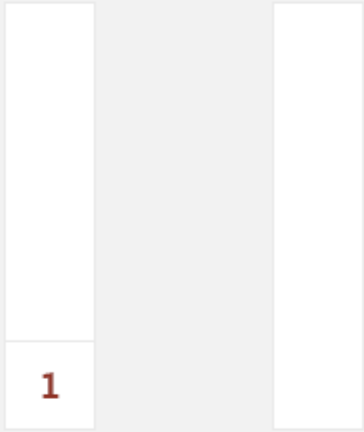
Applications:
Dijkstra's two-stack algorithm 4

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)
	↑															

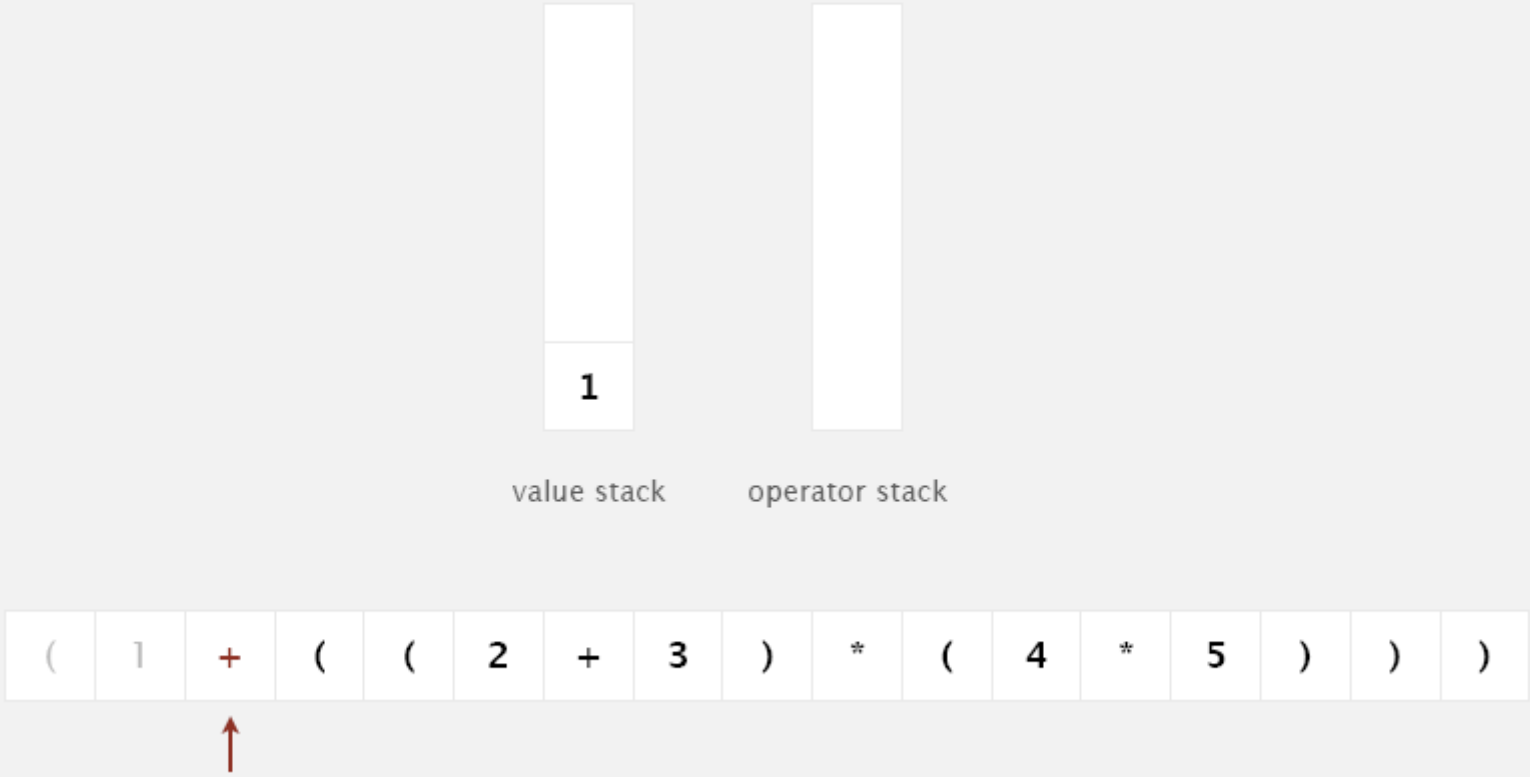
Applications:
Dijkstra's two-stack algorithm 5

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



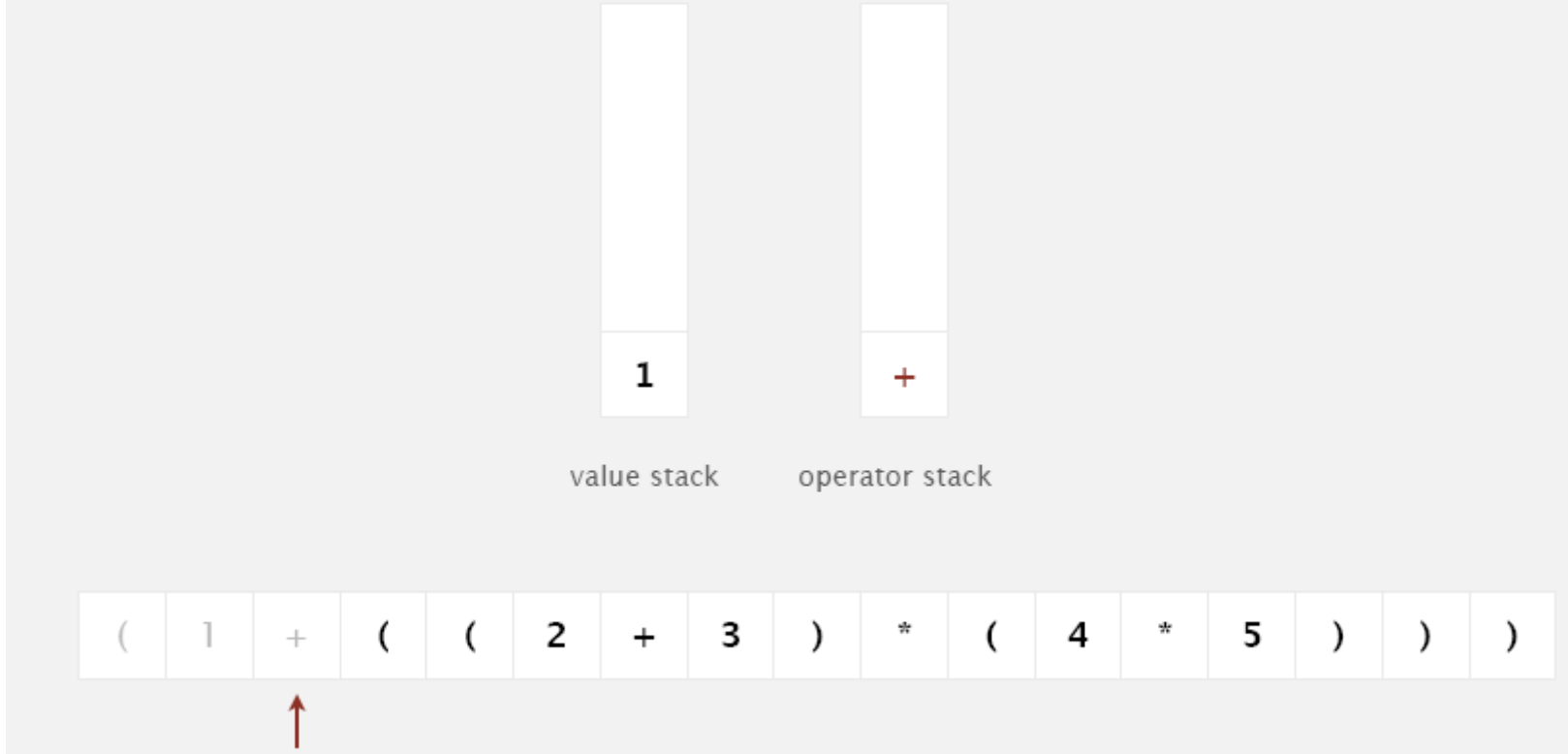
Applications:
Dijkstra's two-stack algorithm 6

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



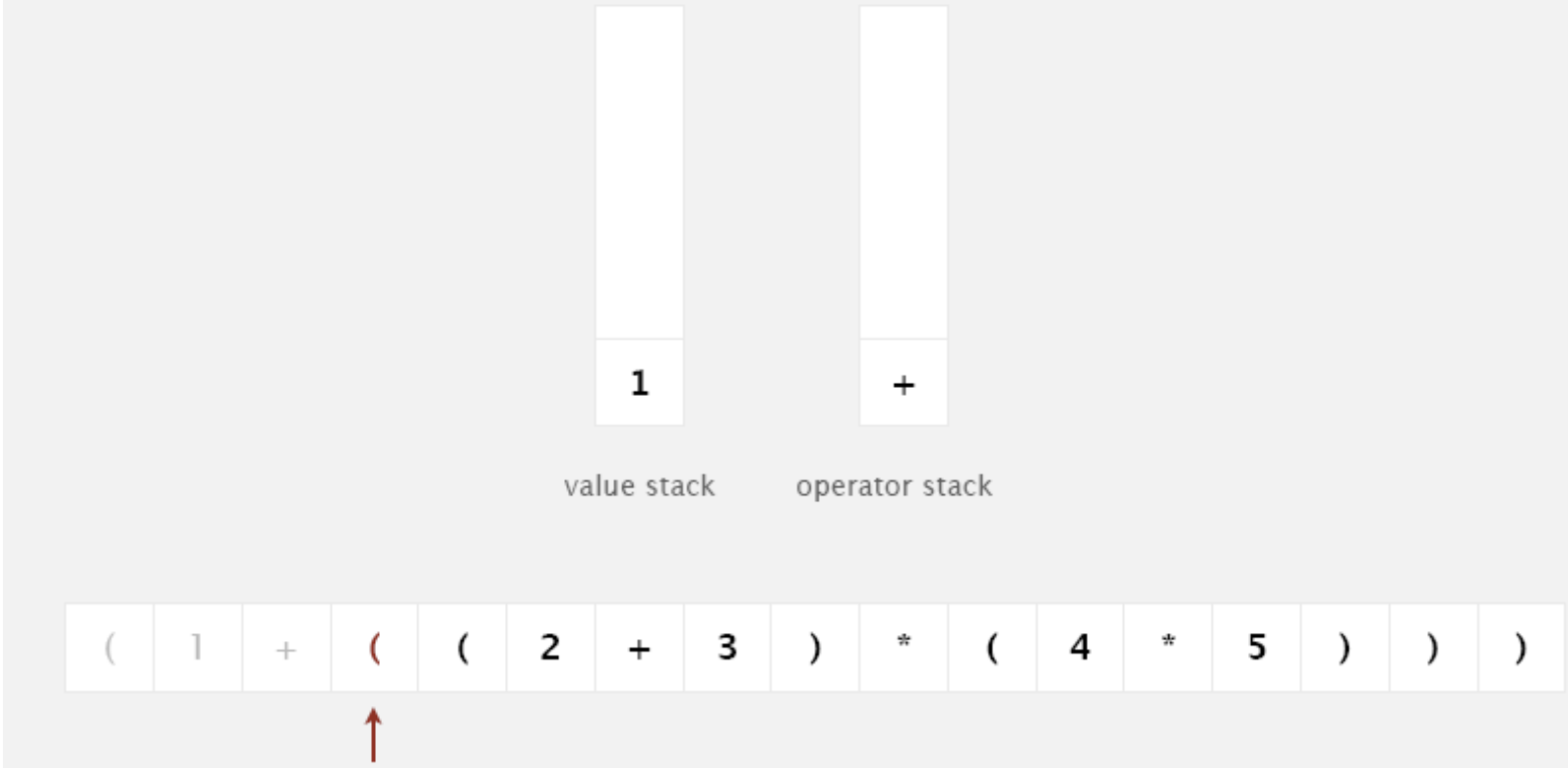
Applications:
Dijkstra's two-stack algorithm 7

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



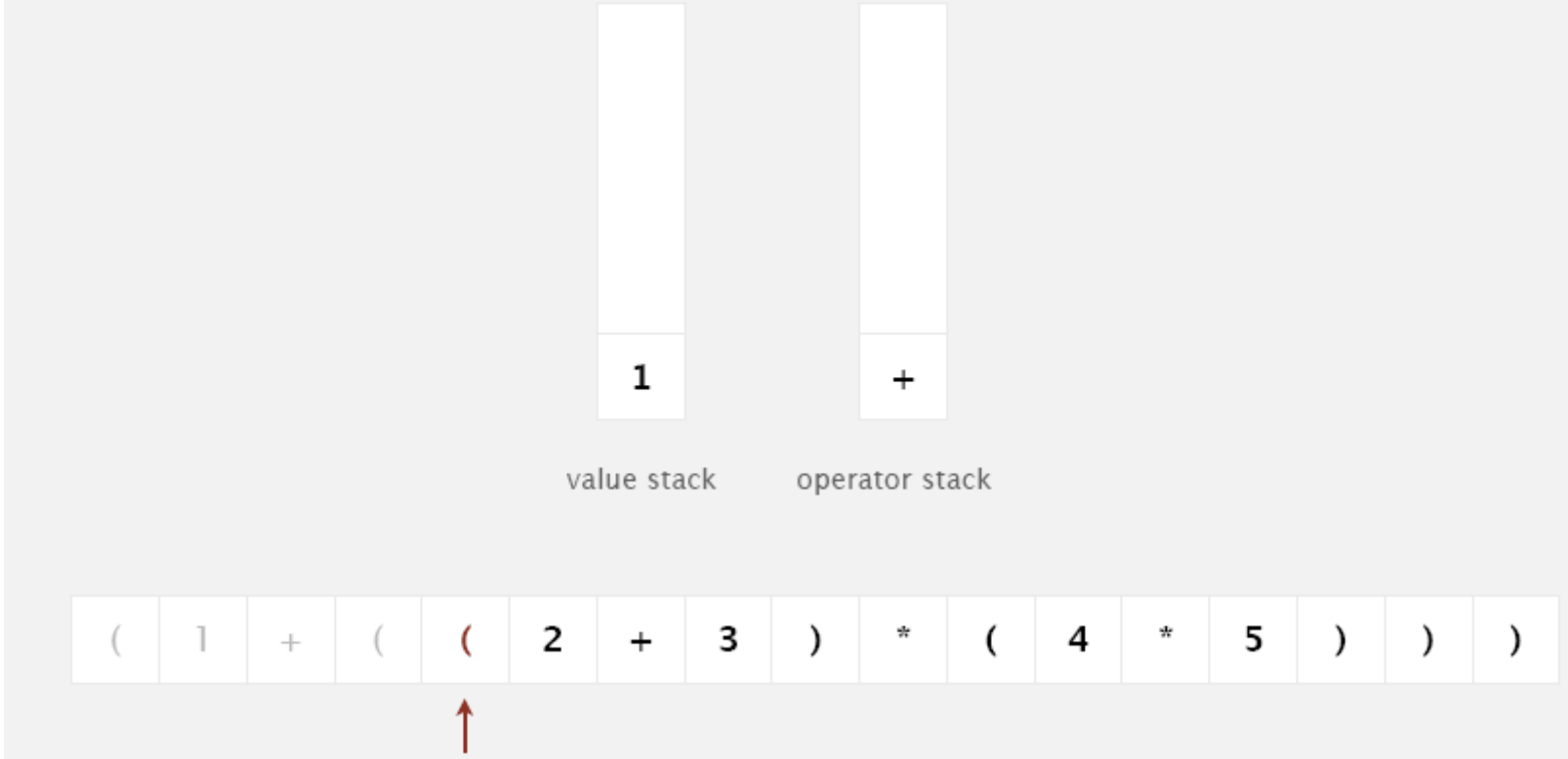
Applications:
Dijkstra's two-stack algorithm 8

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Applications: Dijkstra's two-stack algorithm 9

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

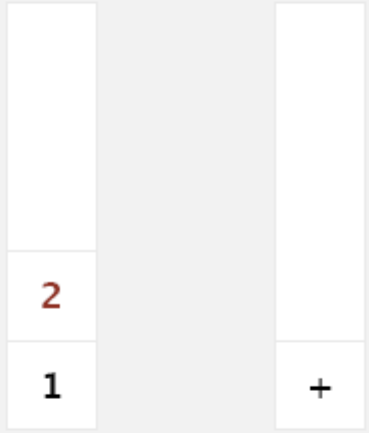
Applications:
Dijkstra's two-stack algorithm 10

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

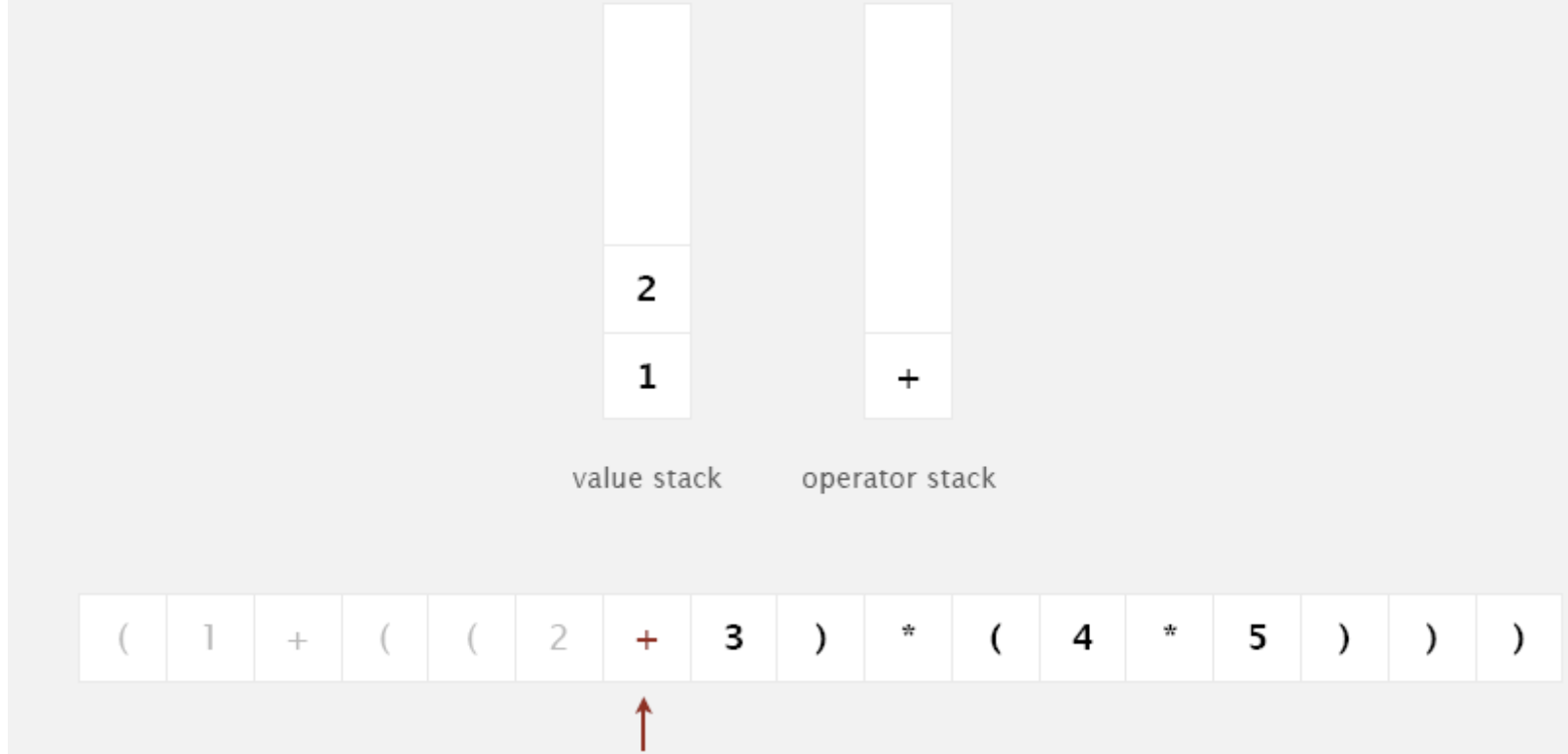
Applications: Dijkstra's two-stack algorithm 11

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



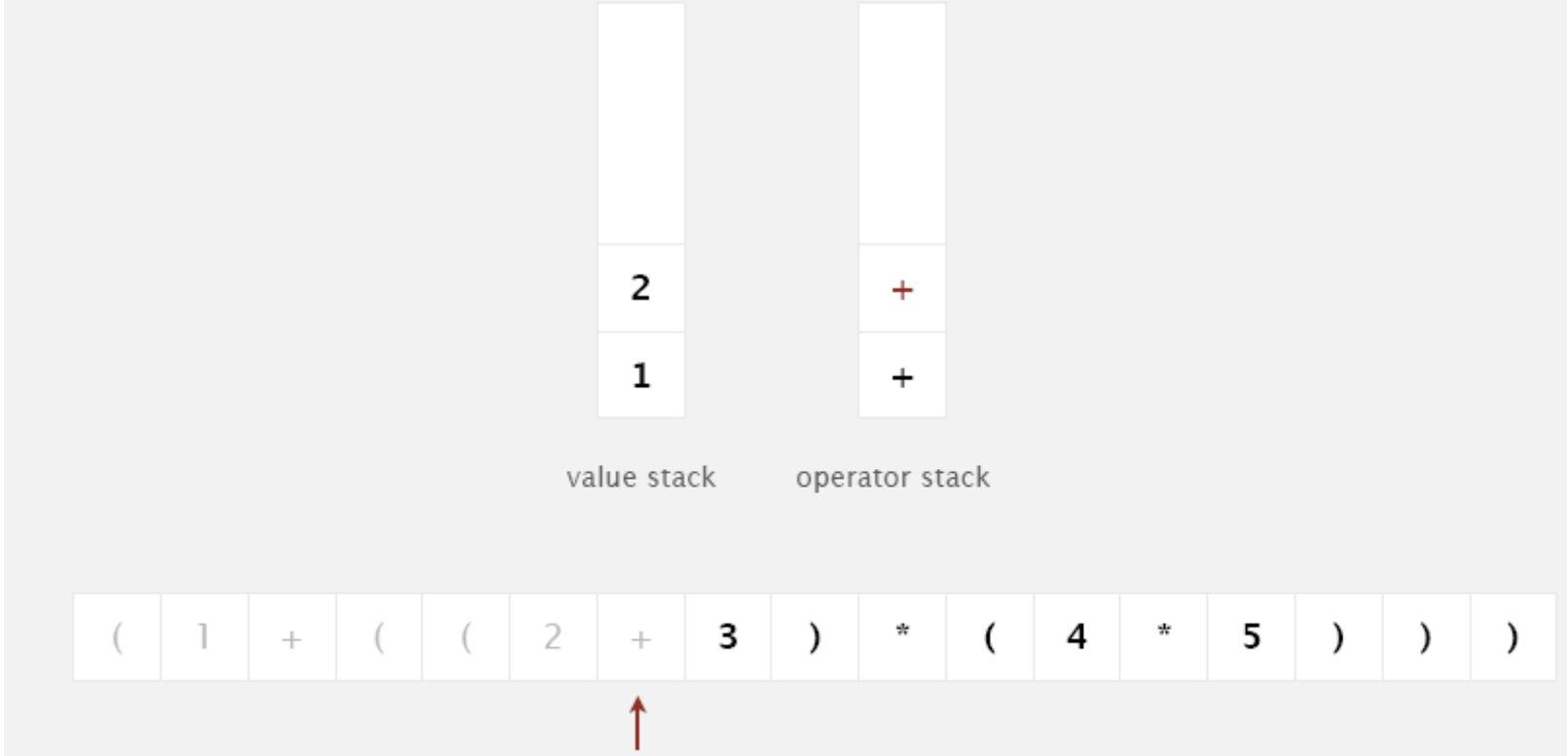
Applications: Dijkstra's two-stack algorithm 12

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



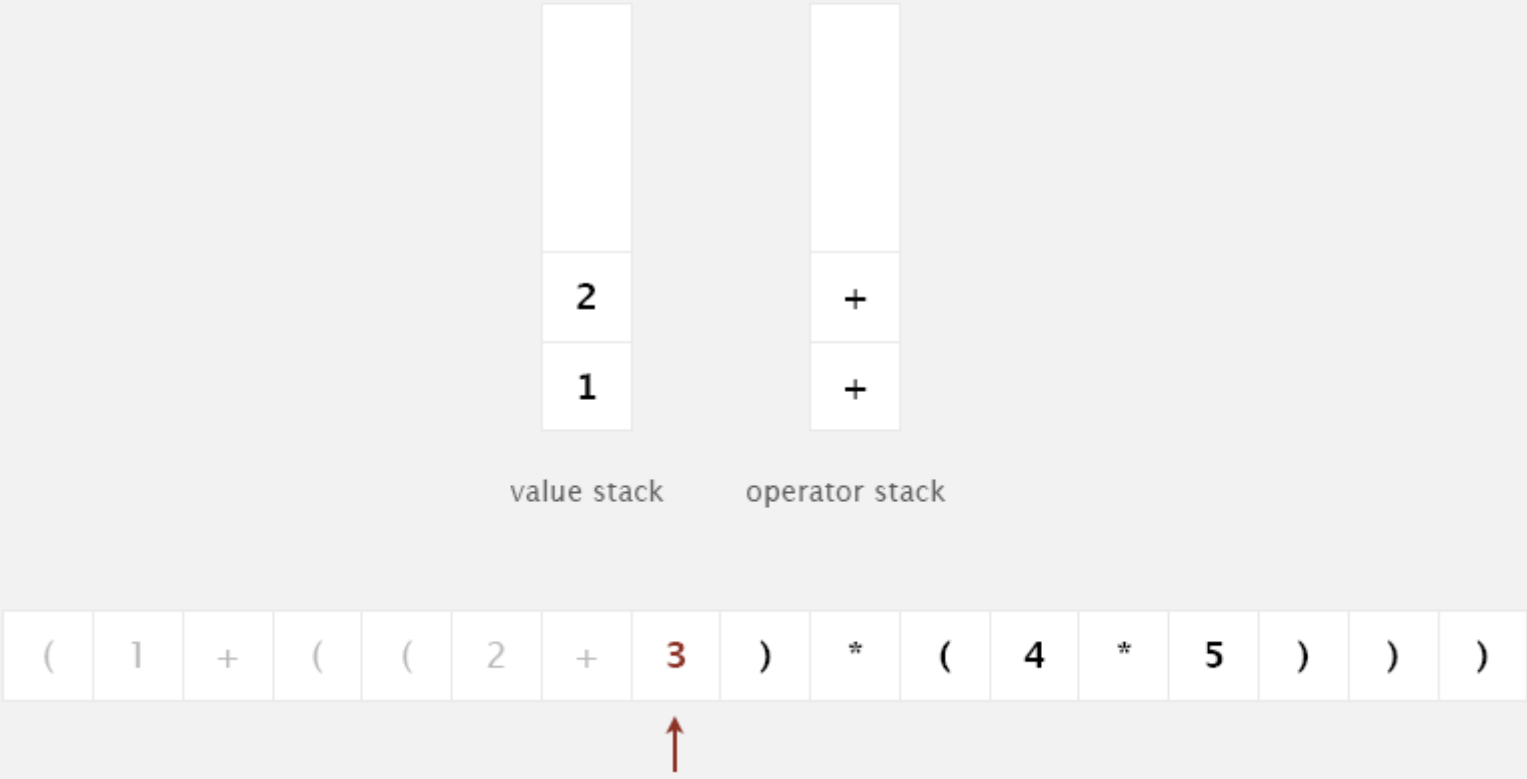
Applications:
Dijkstra's two-stack algorithm 13

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



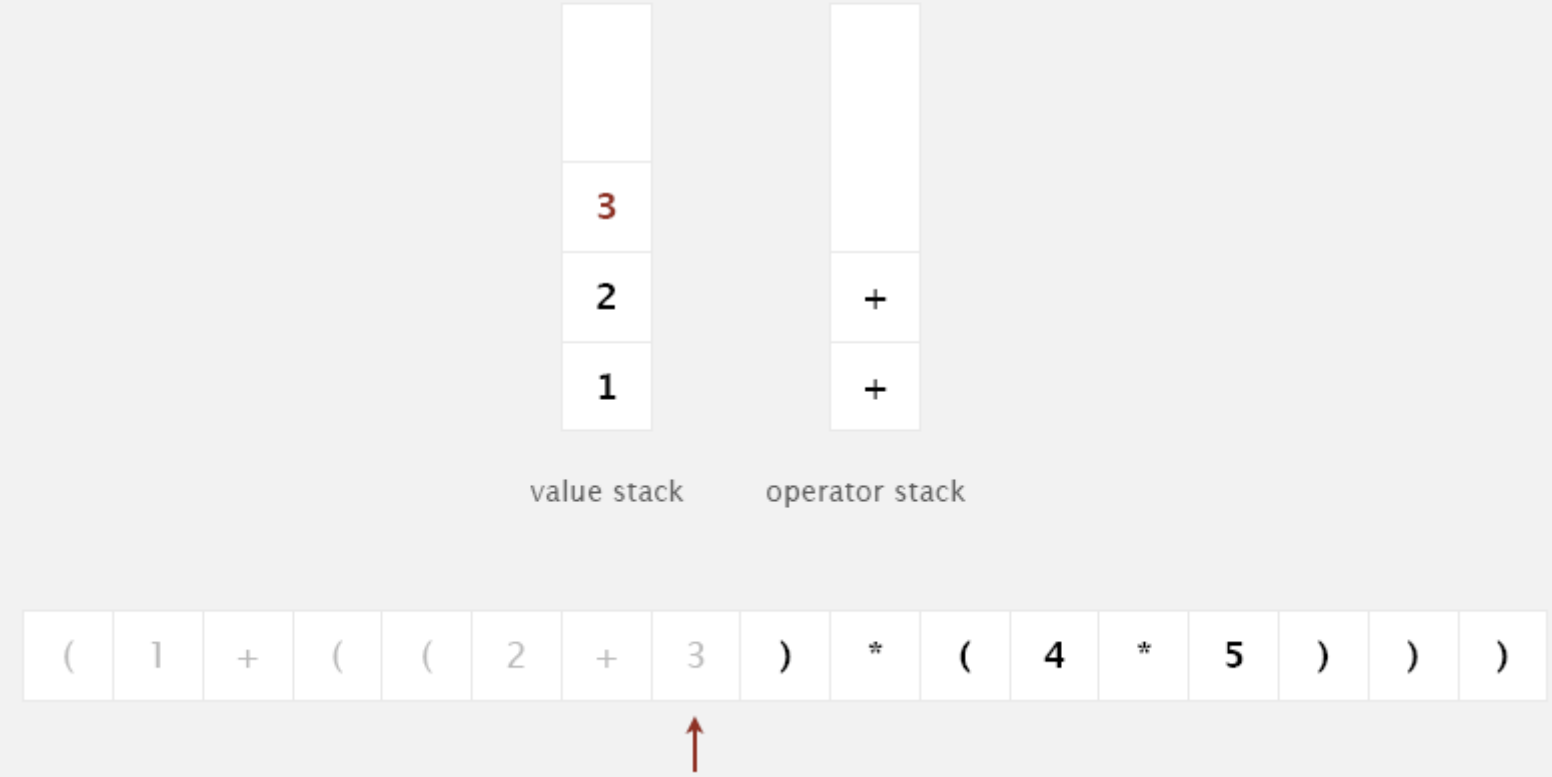
Applications: Dijkstra's two-stack algorithm 14

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



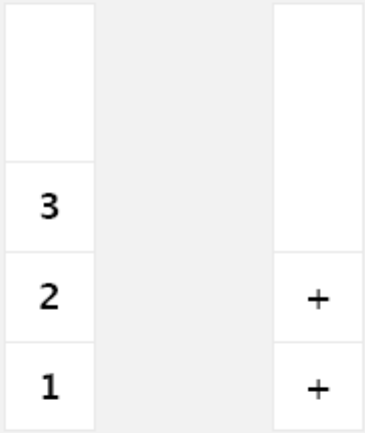
Applications:
Dijkstra's two-stack algorithm 15

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

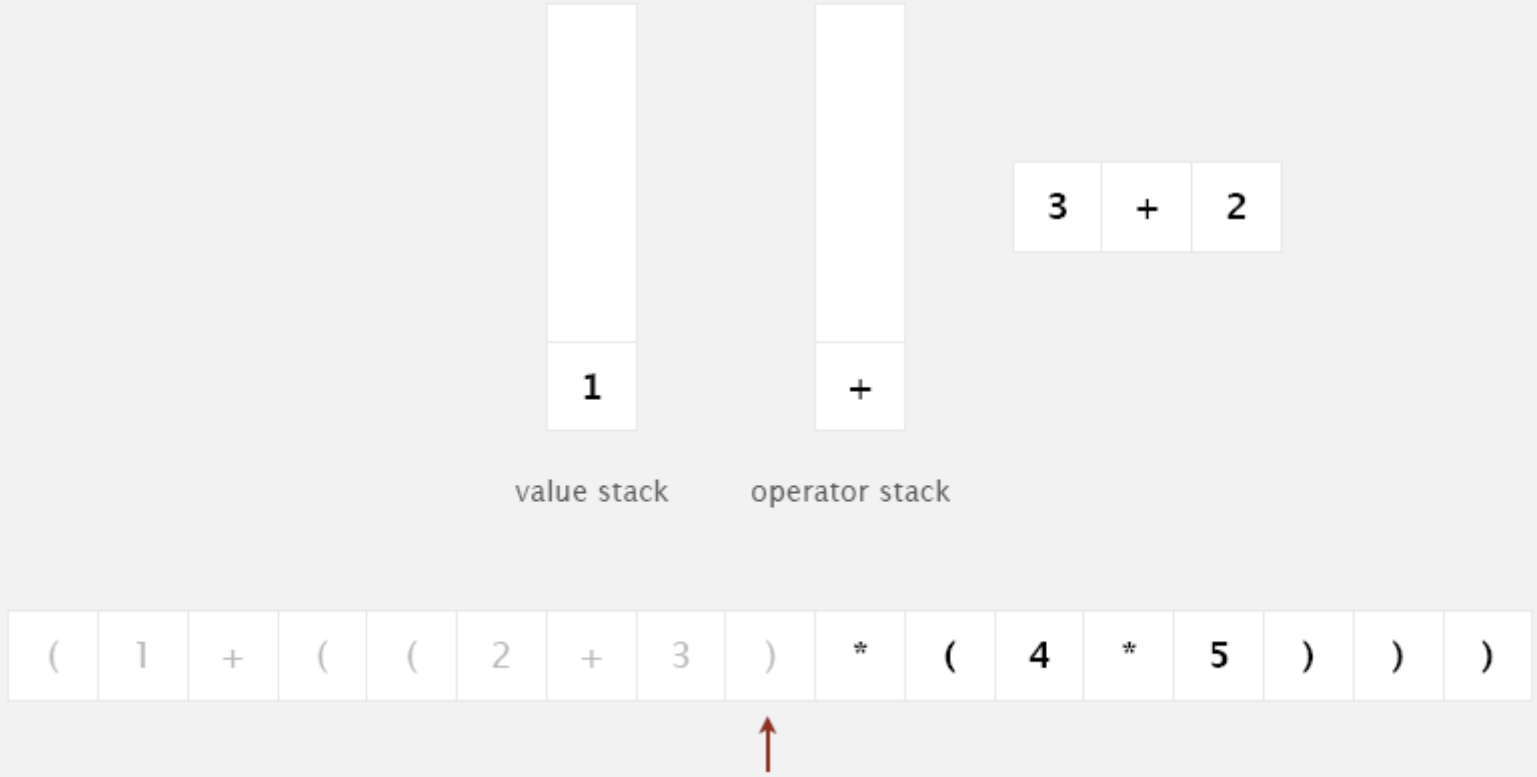
Applications: Dijkstra's two-stack algorithm 16

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



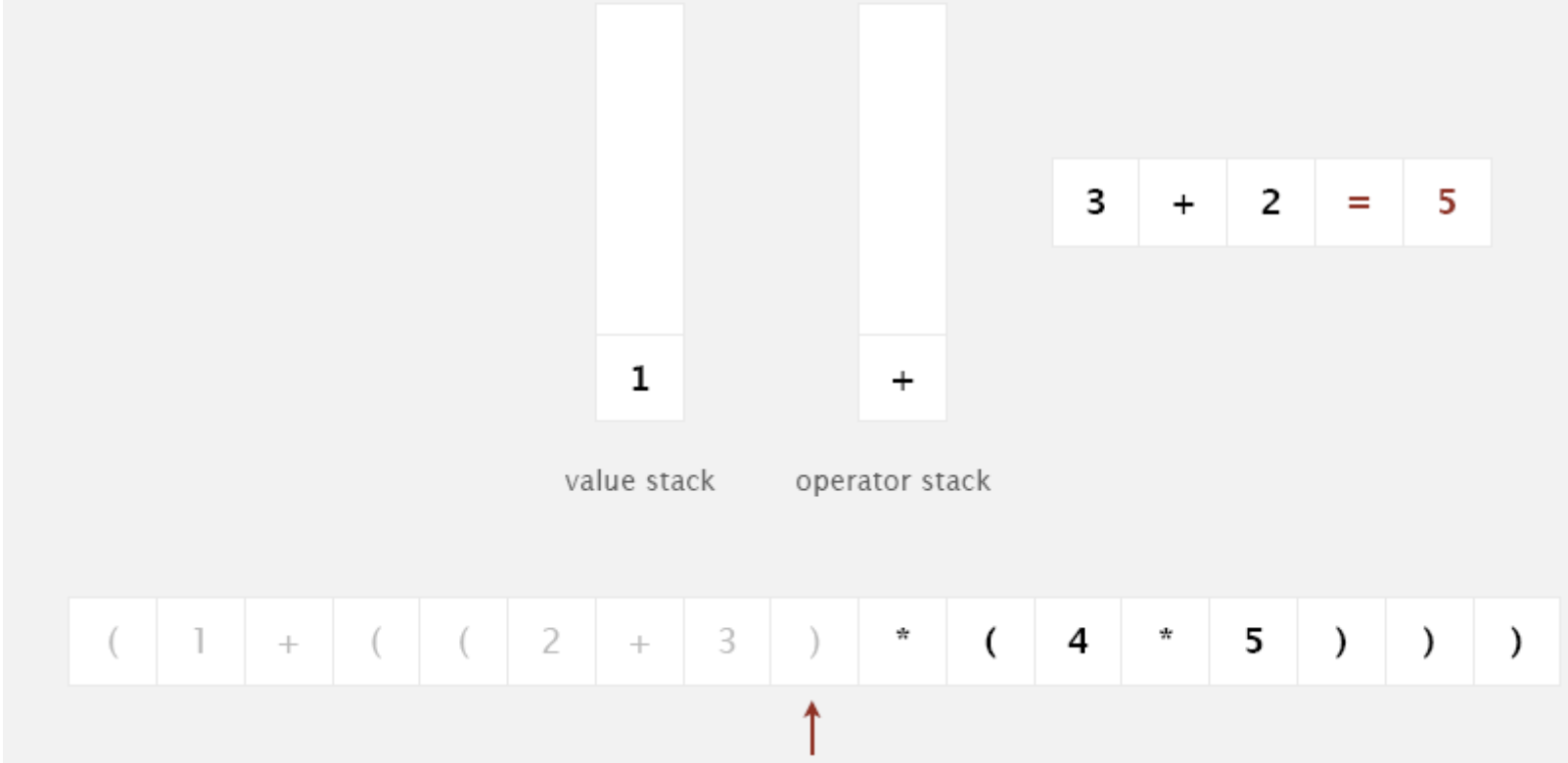
Applications: Dijkstra's two-stack algorithm 17

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



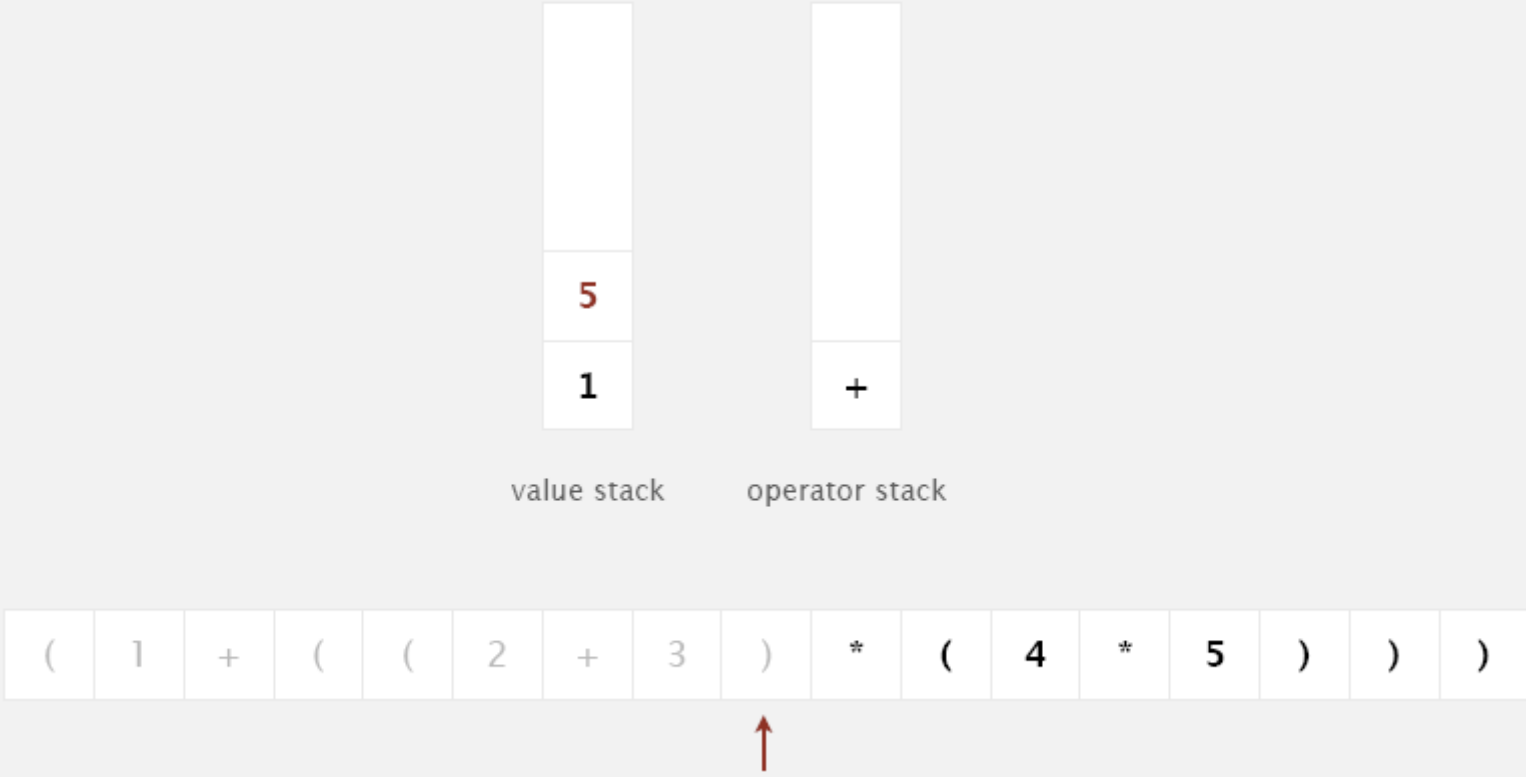
Applications:
Dijkstra's two-stack algorithm 18

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



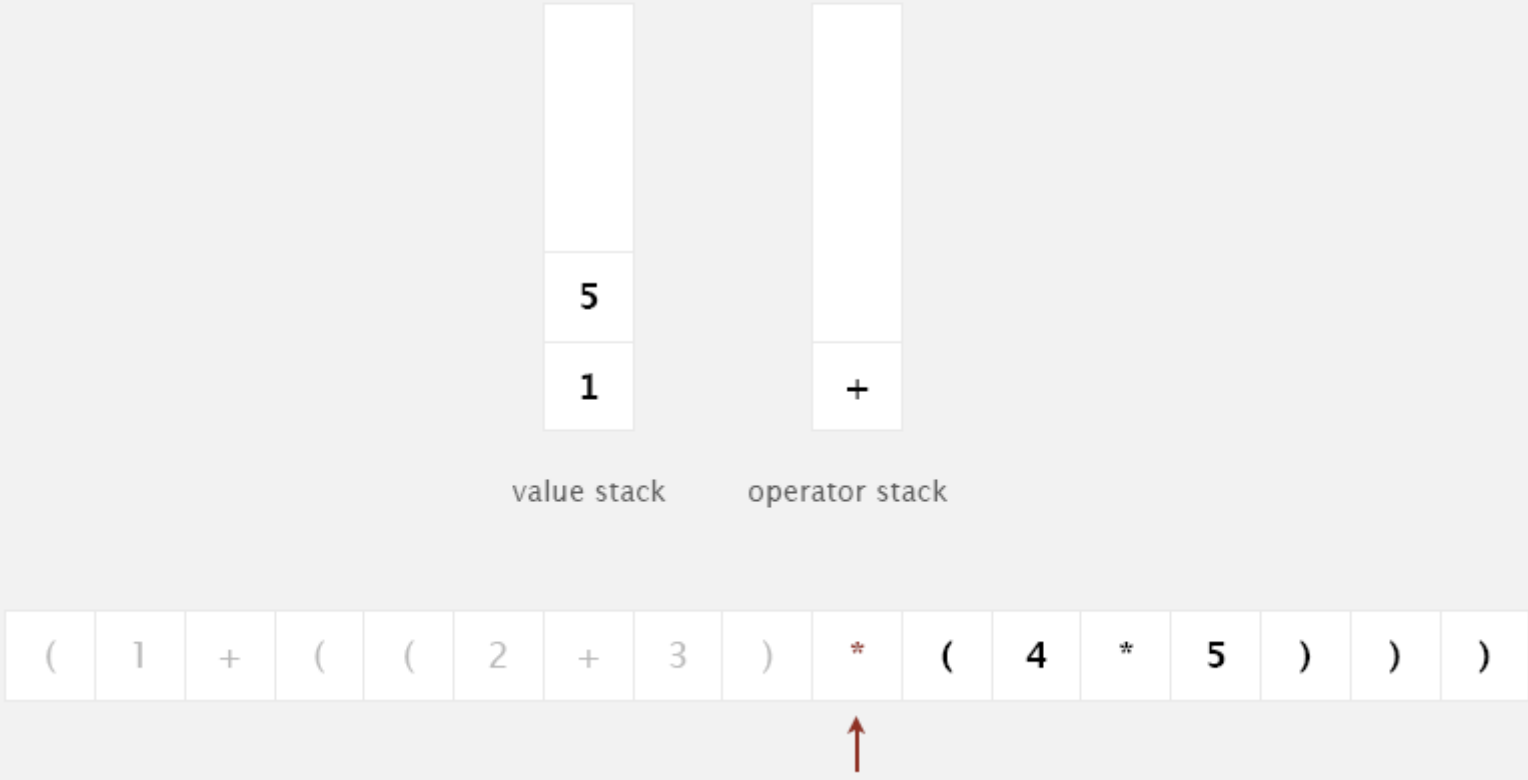
Applications: Dijkstra's two-stack algorithm 19

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



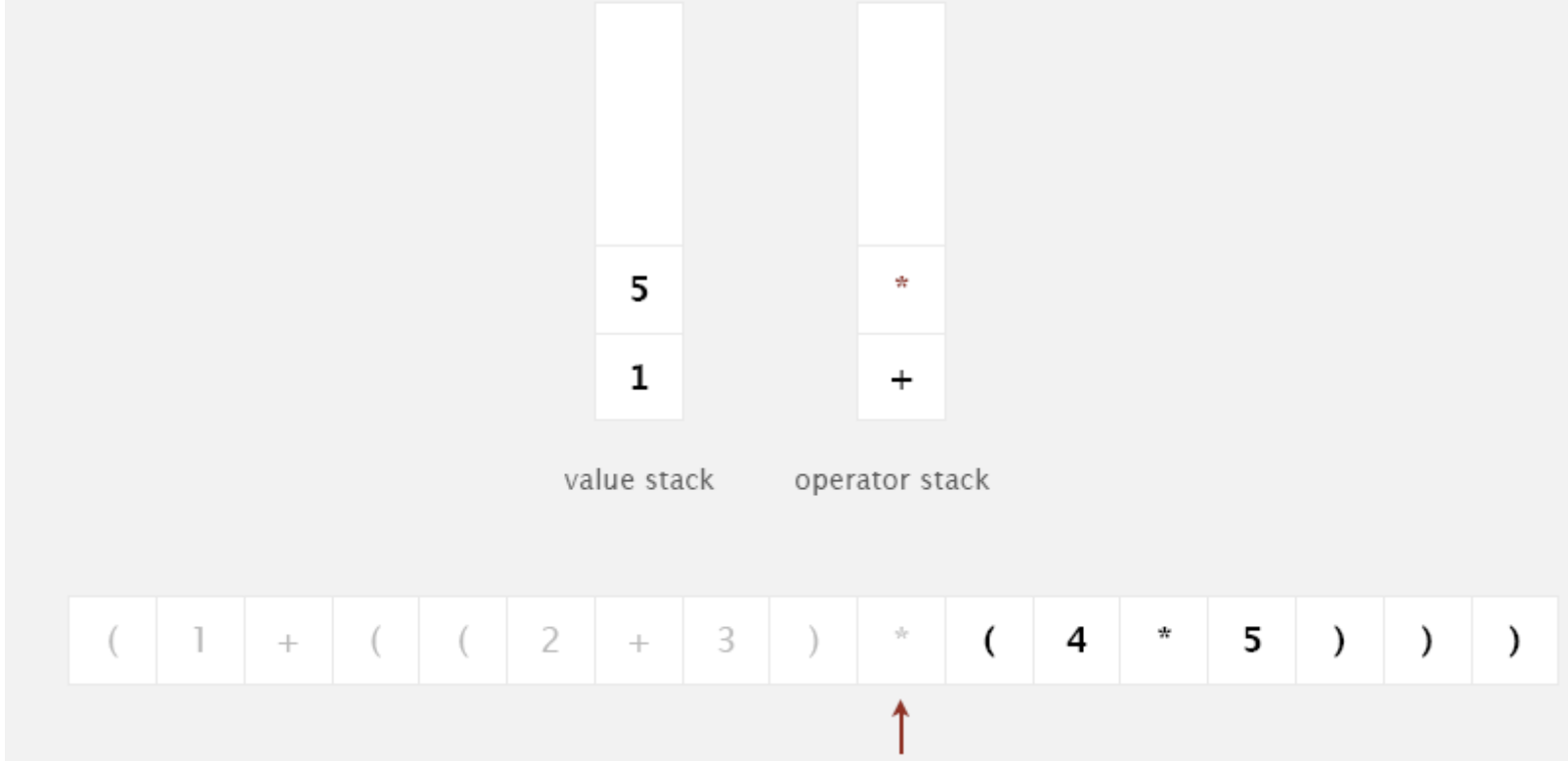
Applications:
Dijkstra's two-stack algorithm 20

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



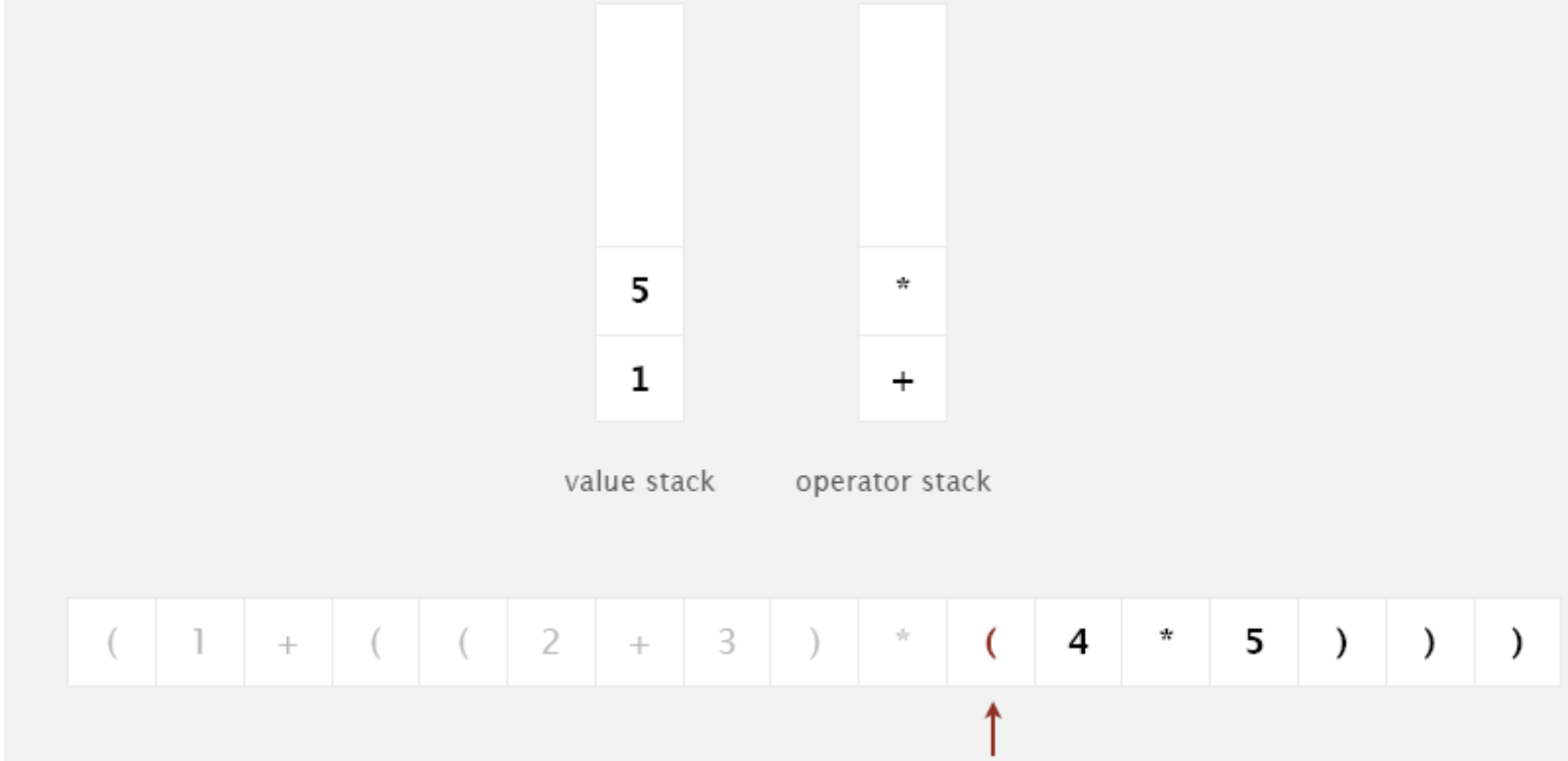
Applications:
Dijkstra's two-stack algorithm 21

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



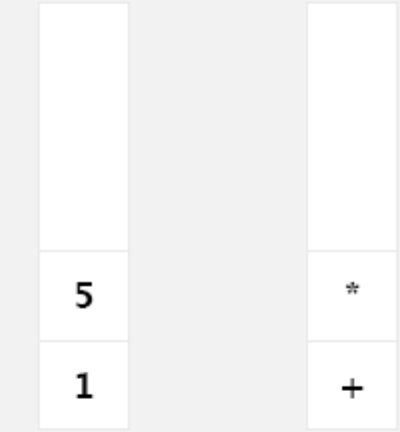
Applications:
Dijkstra's two-stack algorithm 22

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

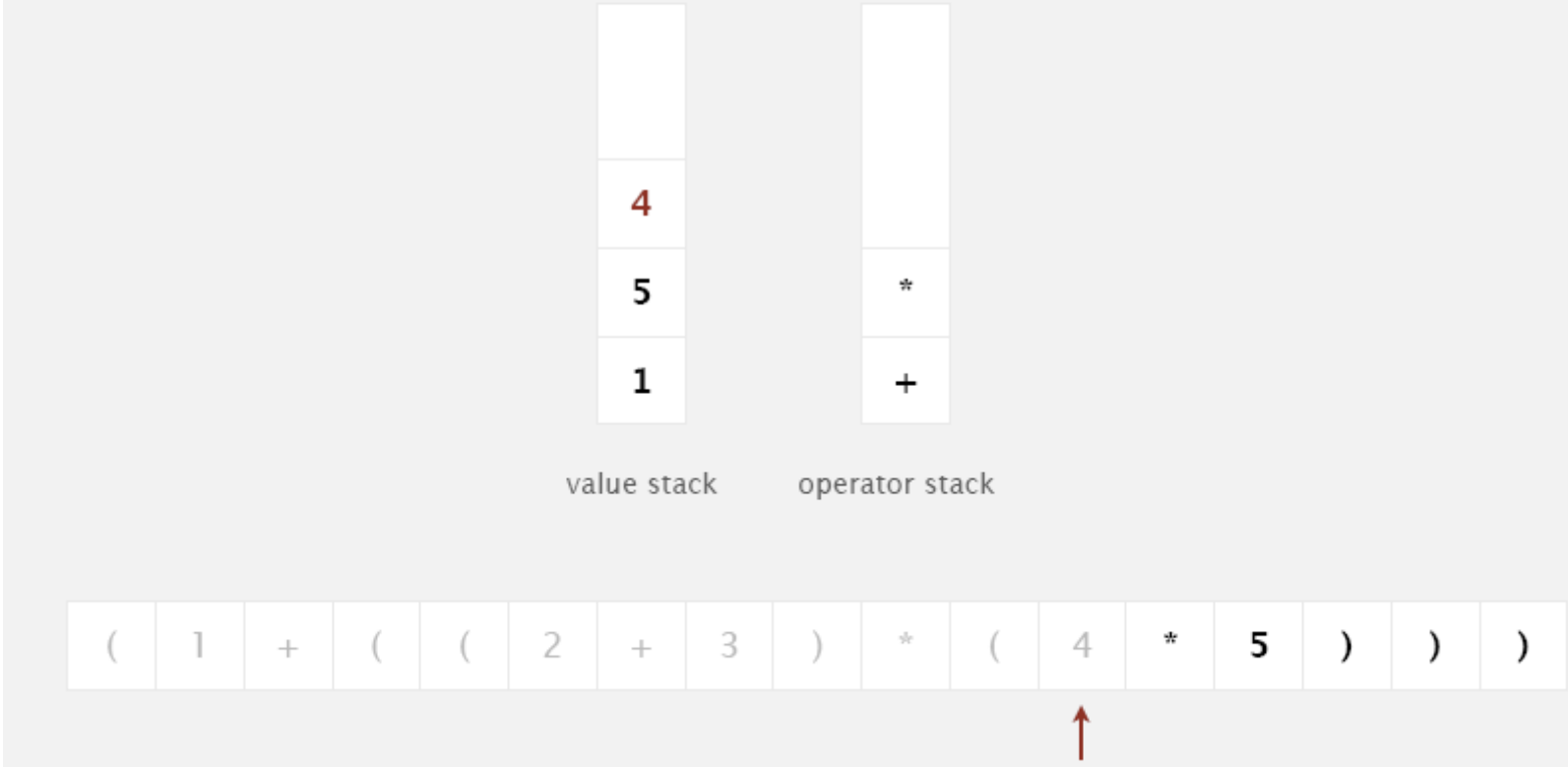
Applications:
Dijkstra's two-stack algorithm 23

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Applications:
Dijkstra's two-stack algorithm 24

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

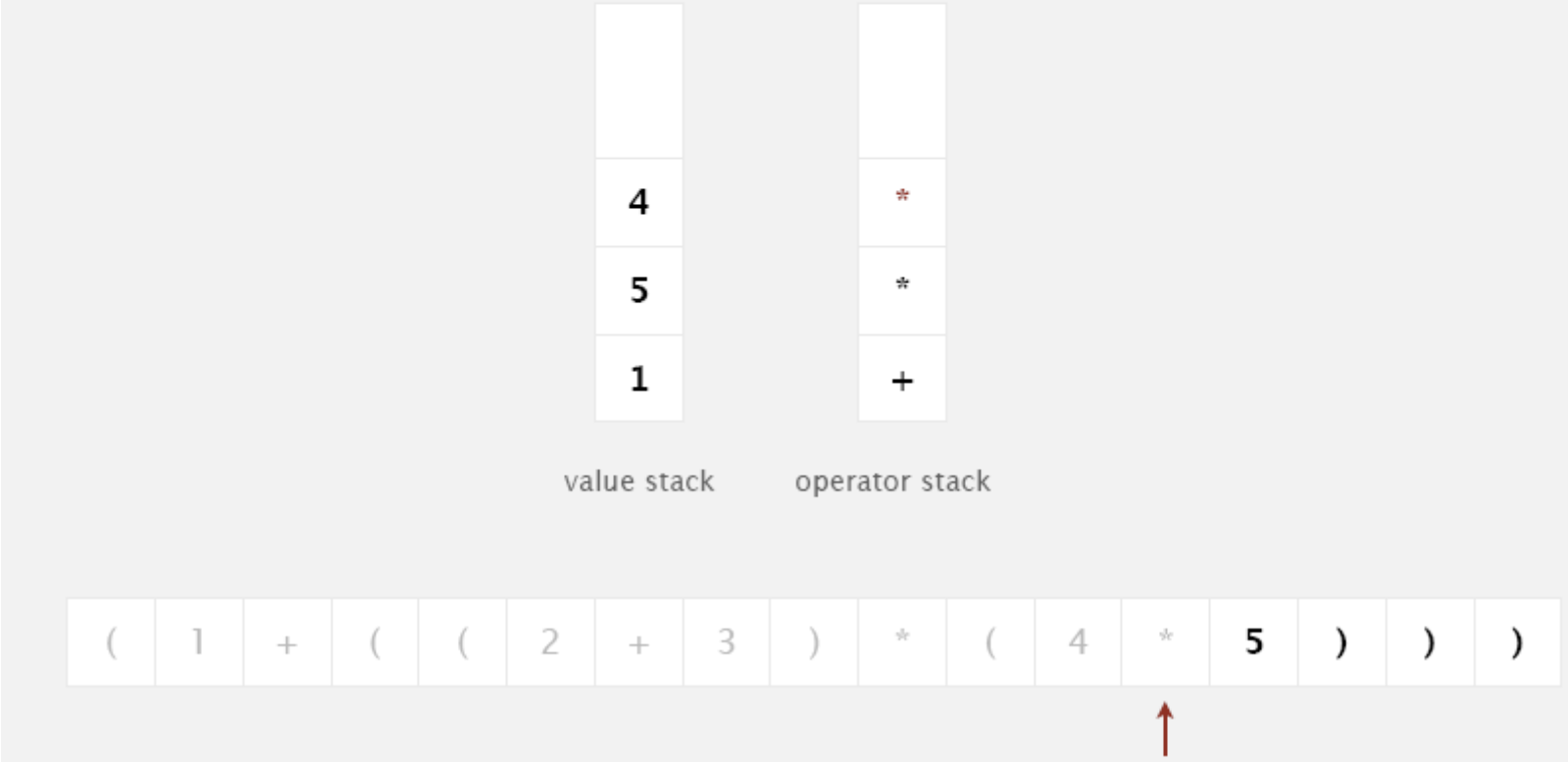
Applications:
Dijkstra's two-stack algorithm 25

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



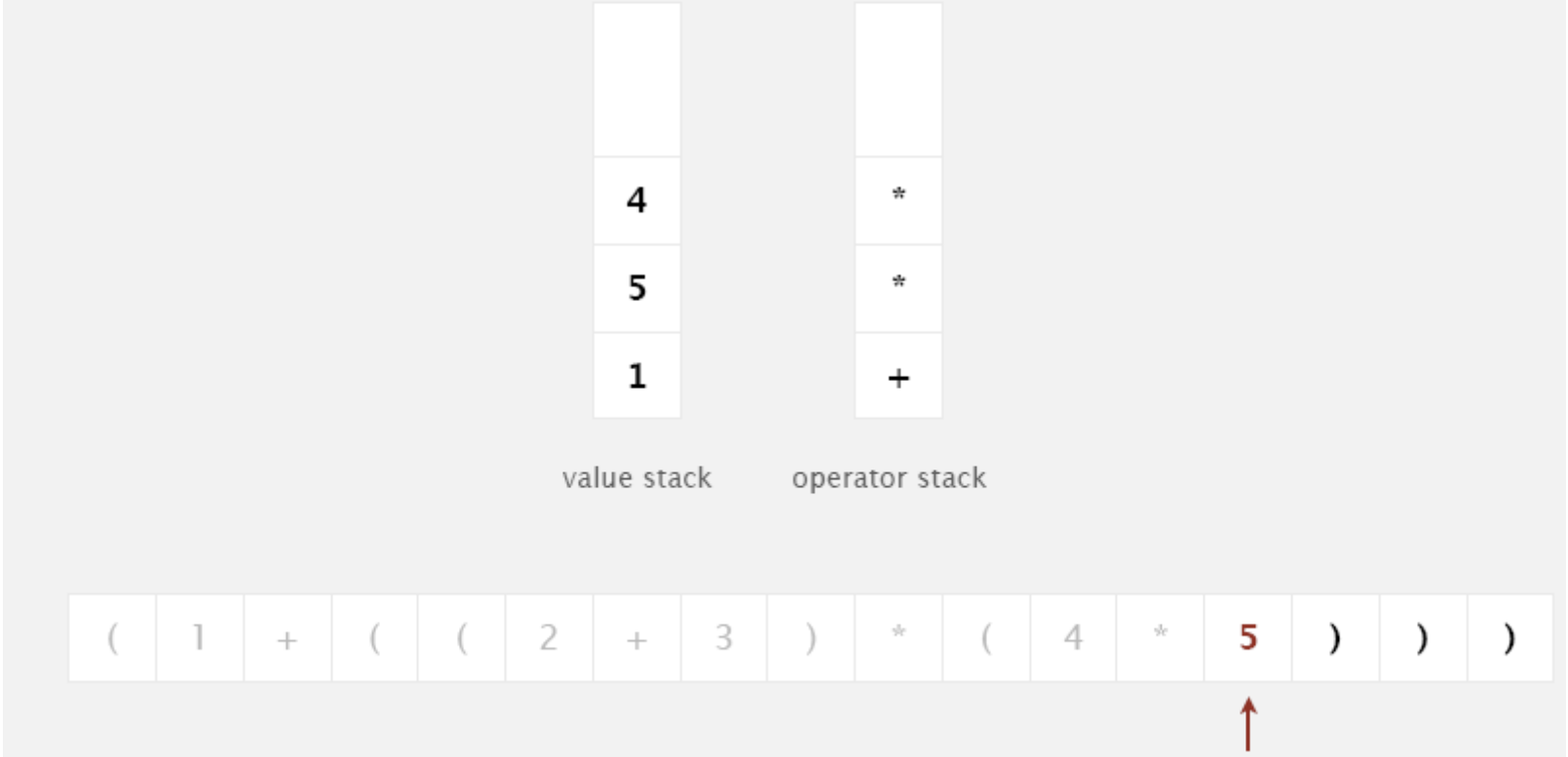
Applications:
Dijkstra's two-stack algorithm 26

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



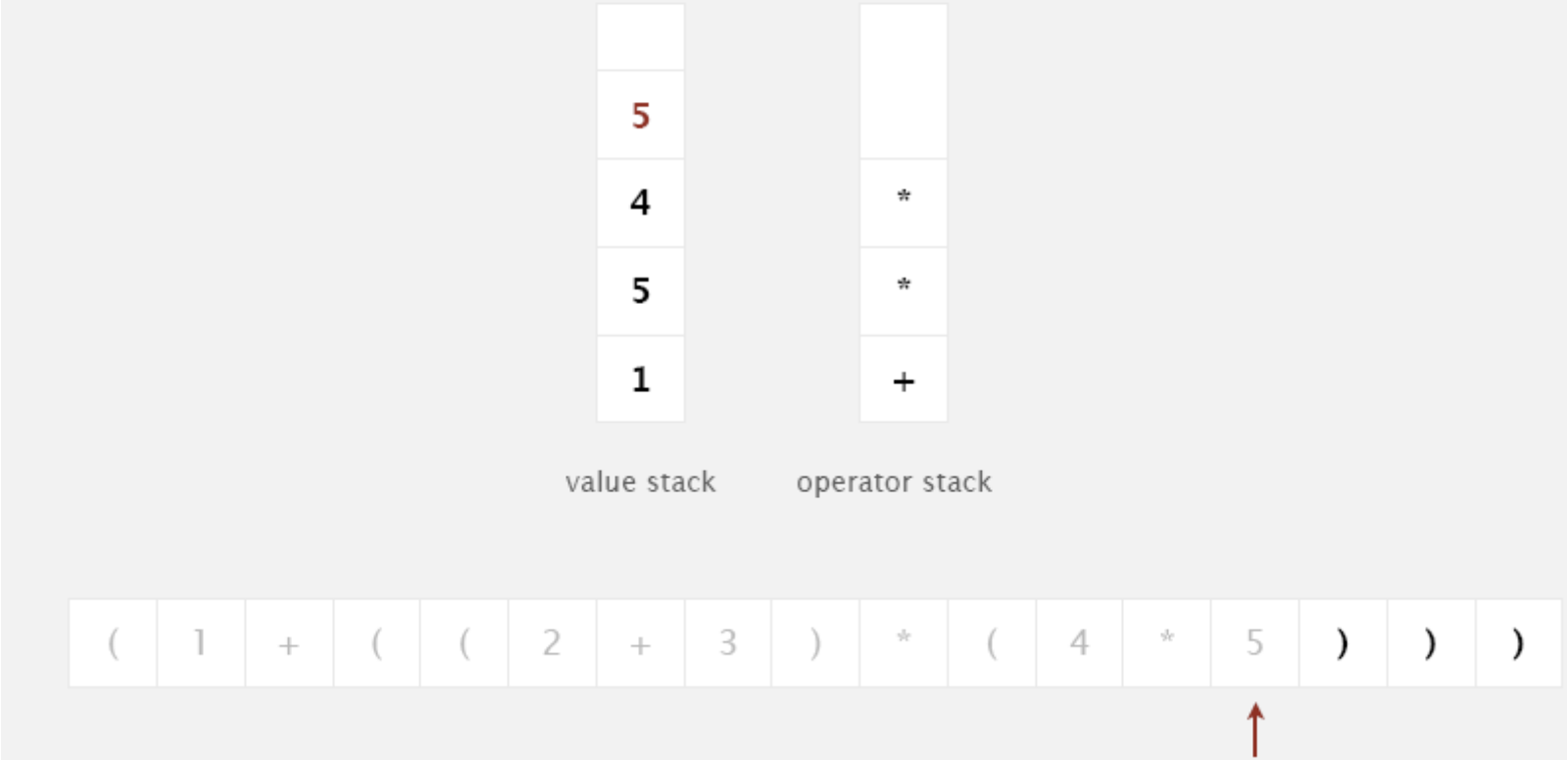
Applications:
Dijkstra's two-stack algorithm 27

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



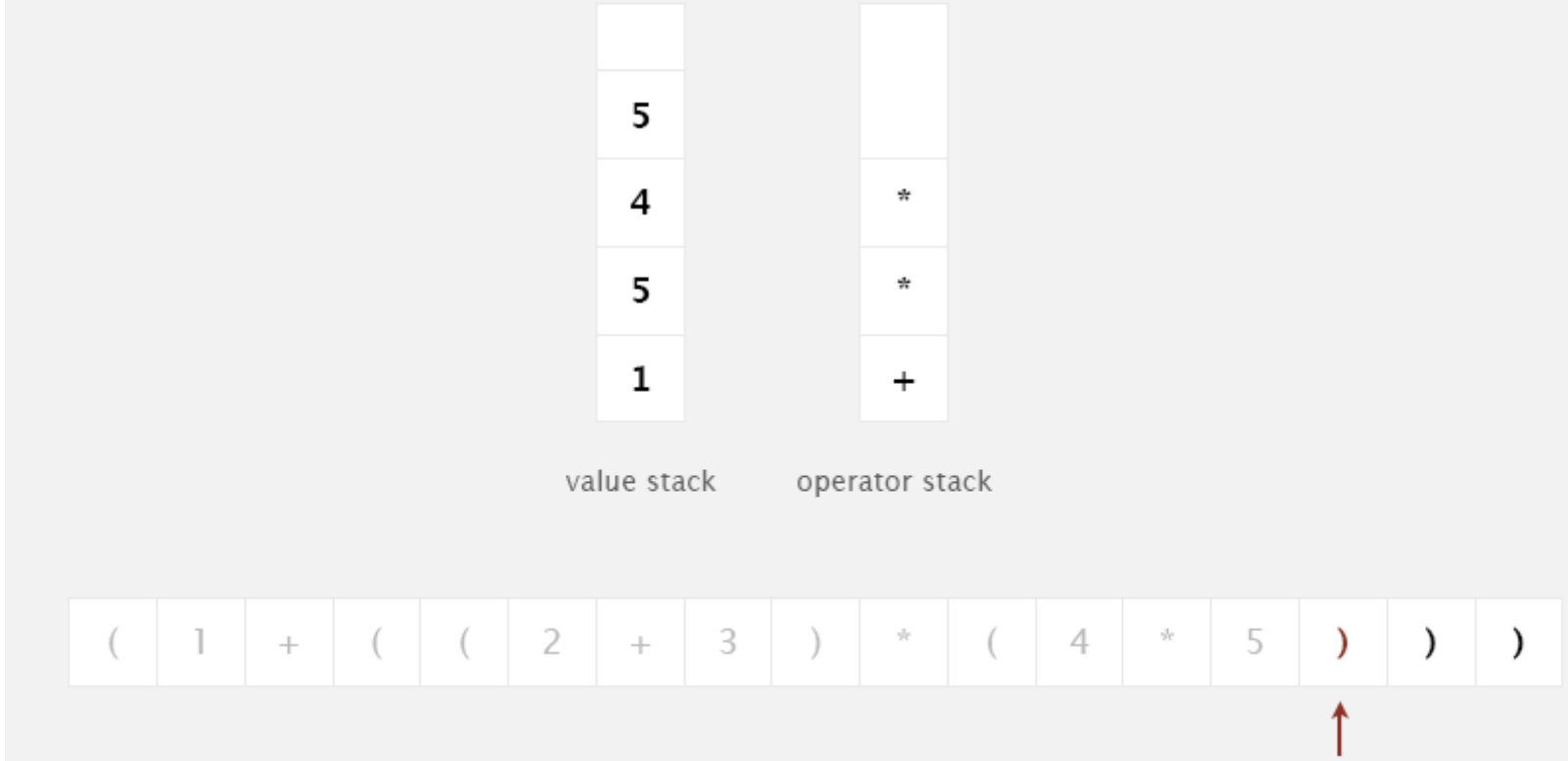
Applications:
Dijkstra's two-stack algorithm 28

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



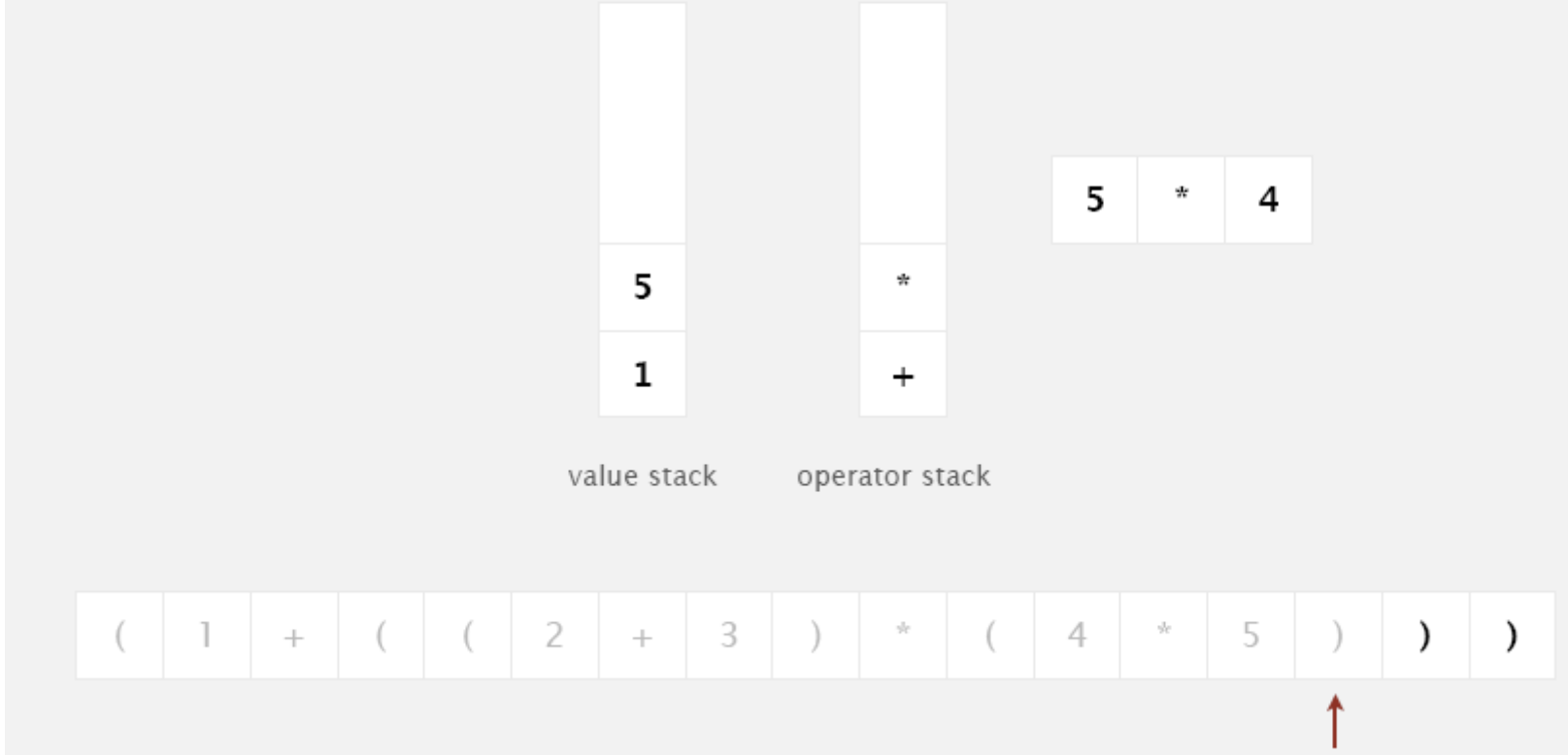
Applications:
Dijkstra's two-stack algorithm 29

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



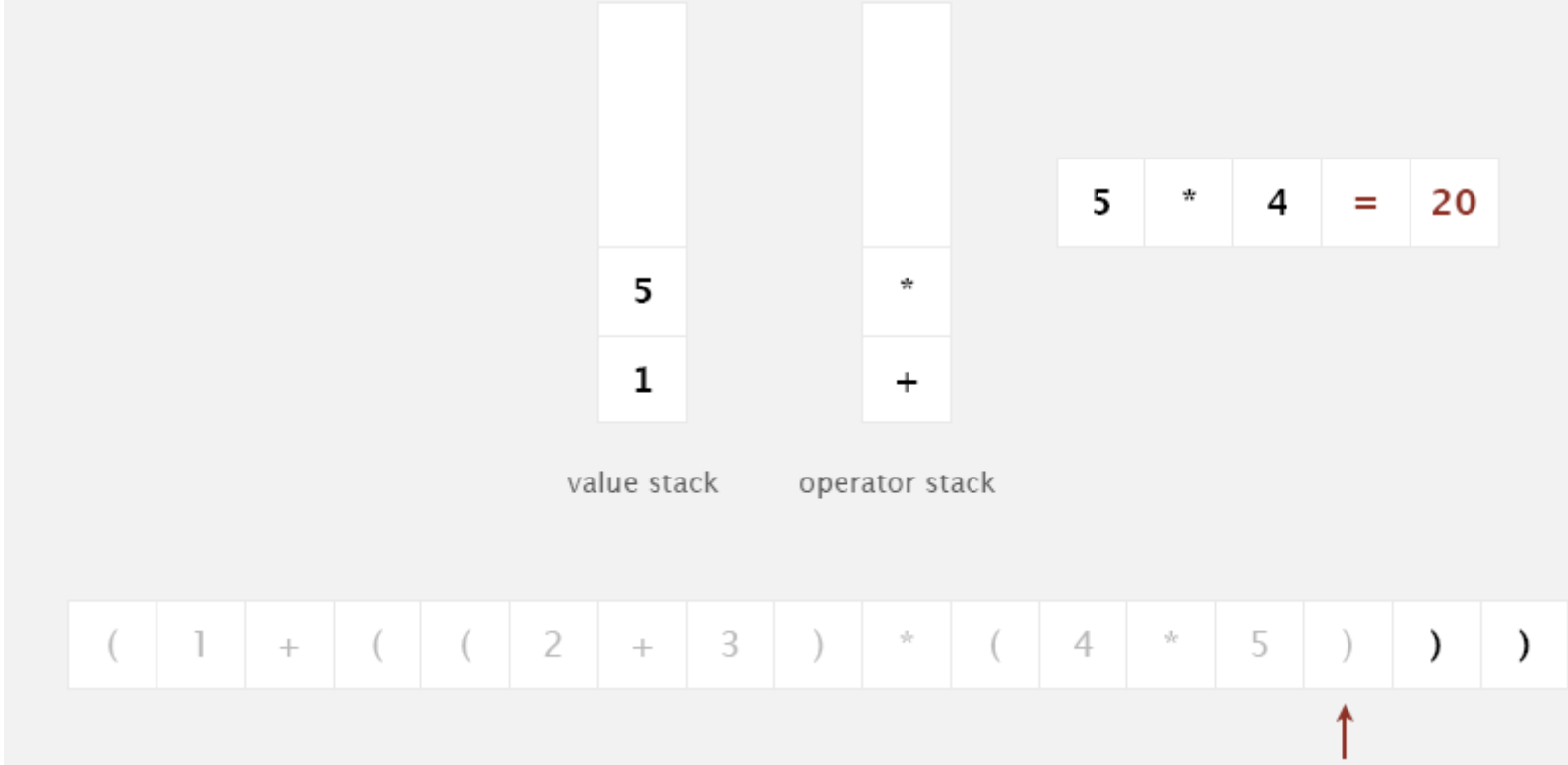
Applications:
Dijkstra's two-stack algorithm 30

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



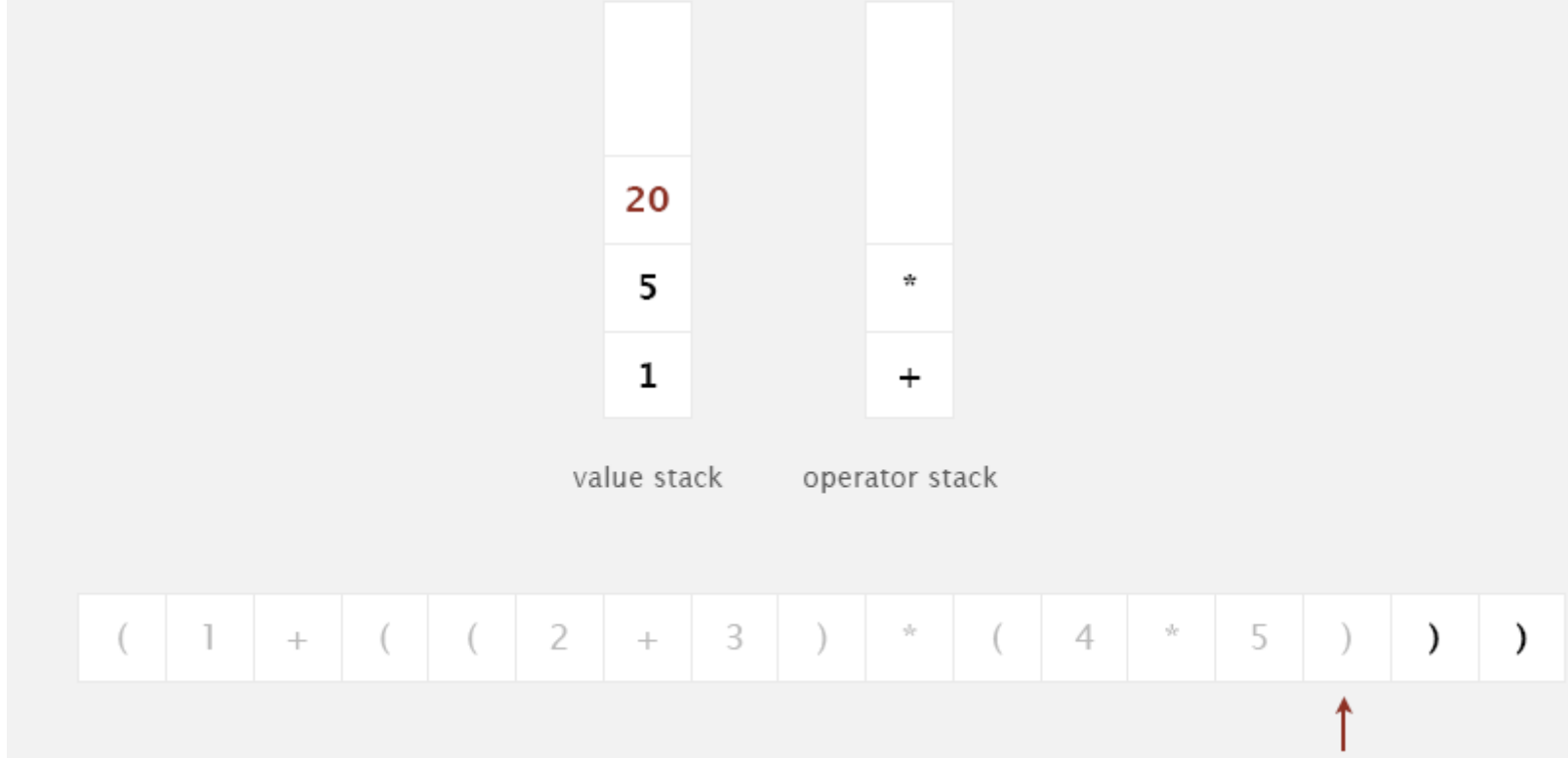
Applications:
Dijkstra's two-stack algorithm 31

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



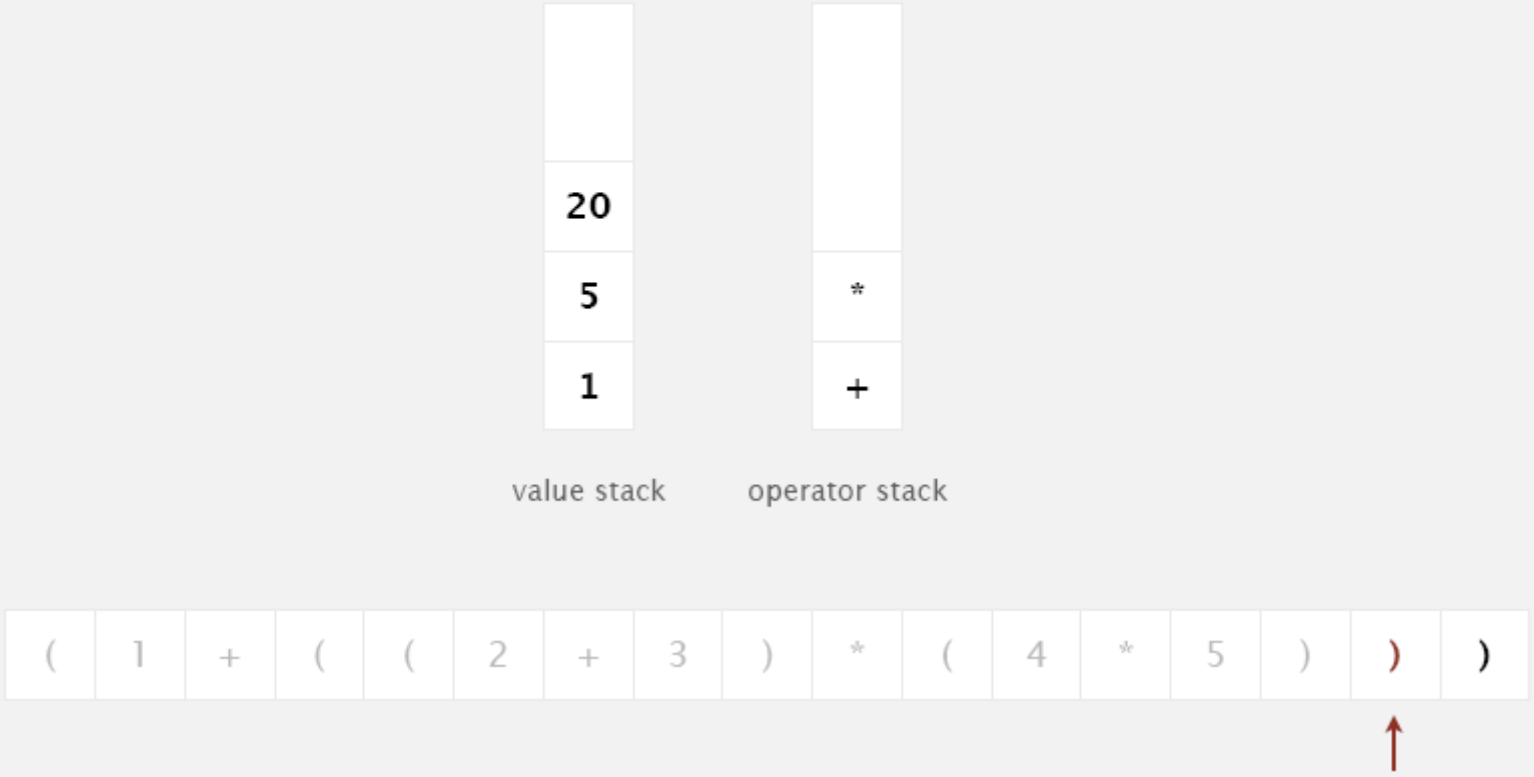
Applications:
Dijkstra's two-stack algorithm 32

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



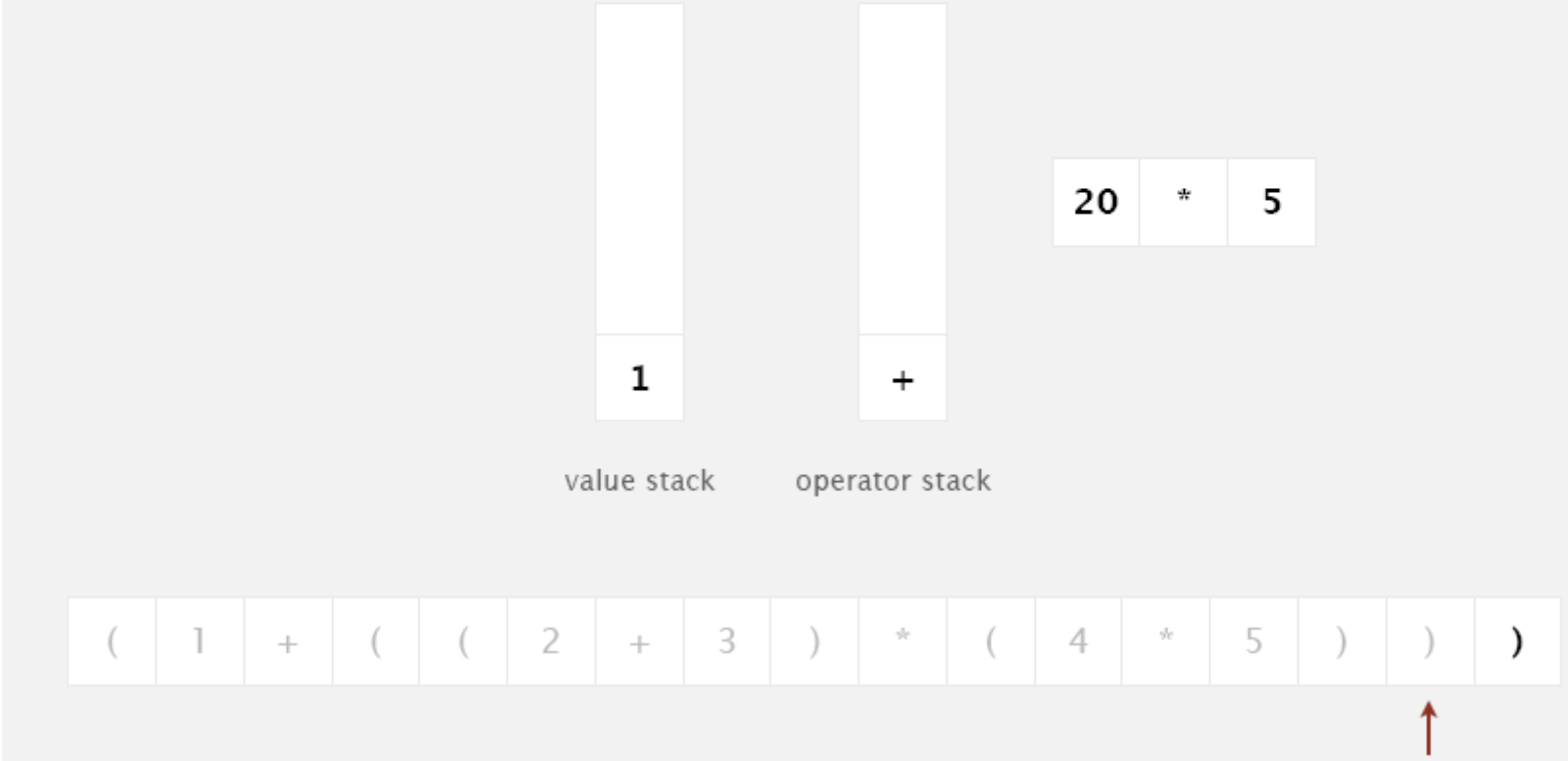
Applications:
Dijkstra's two-stack algorithm 33

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



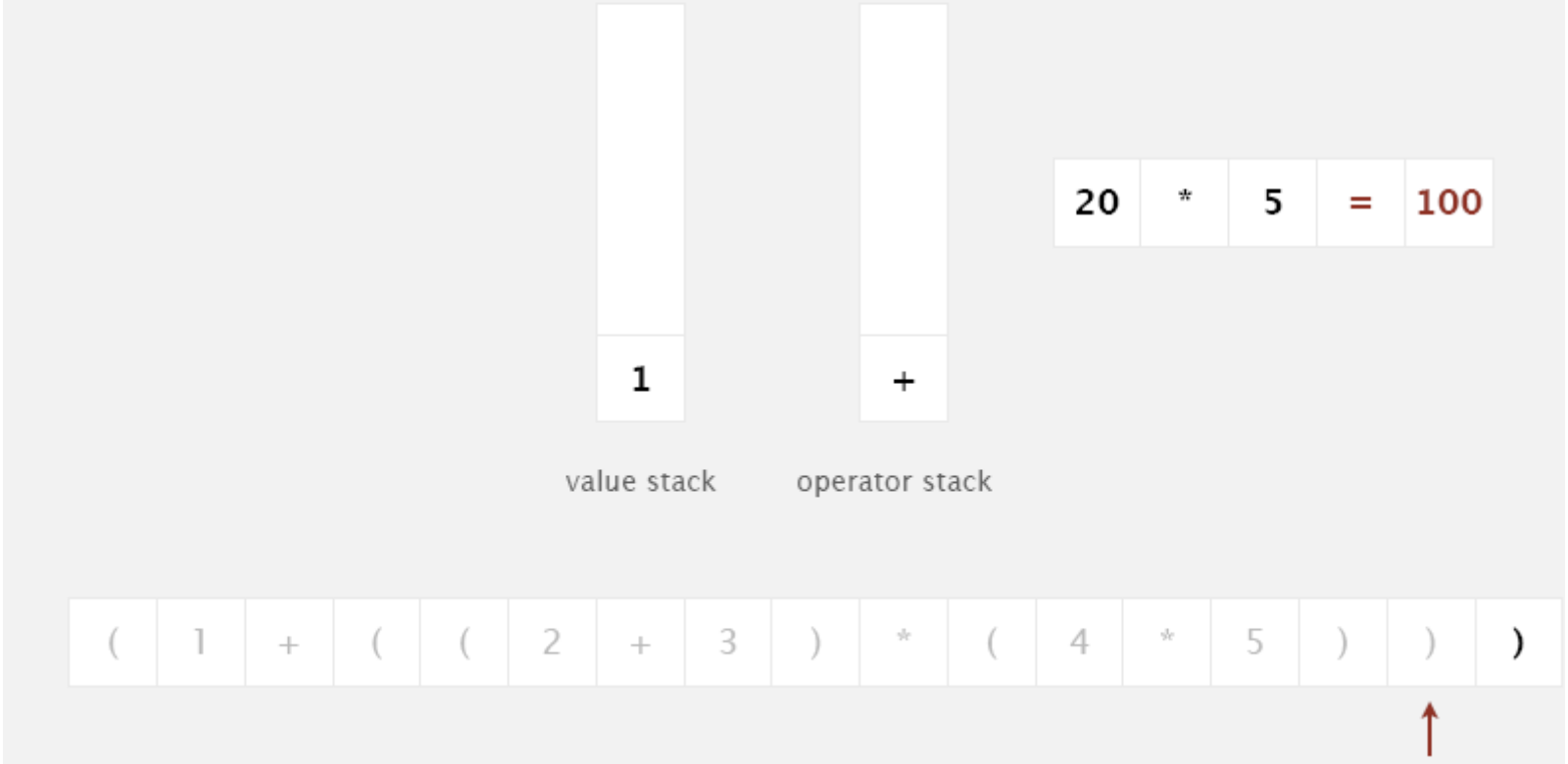
Applications:
Dijkstra's two-stack algorithm 34

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



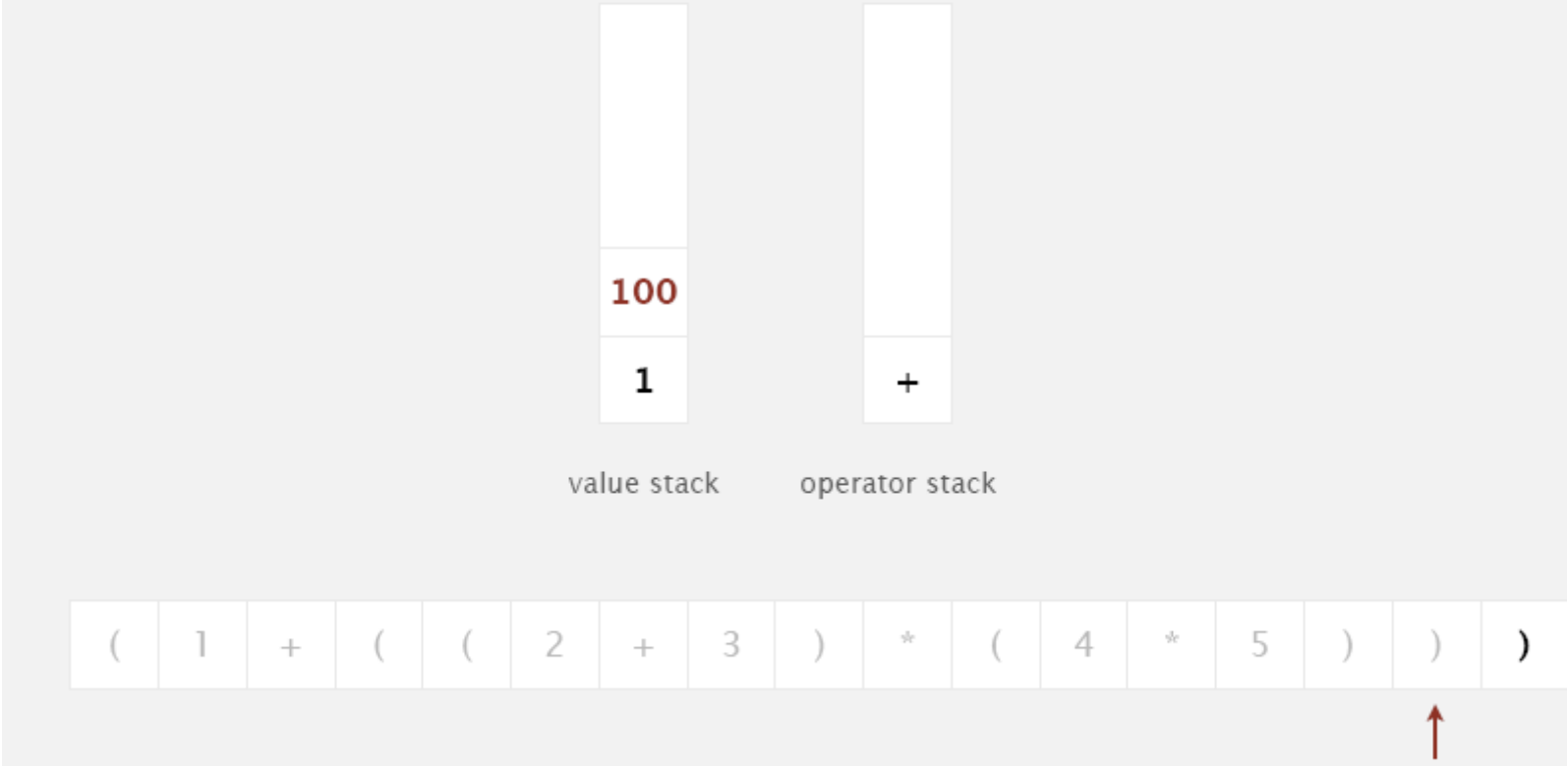
Applications:
Dijkstra's two-stack algorithm 35

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



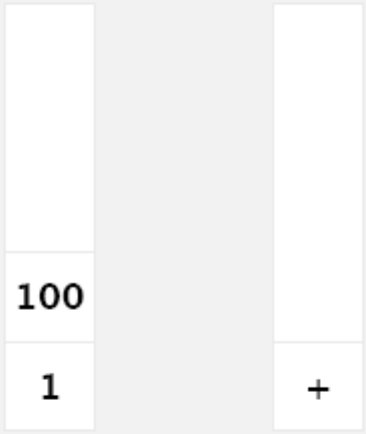
Applications:
Dijkstra's two-stack algorithm 36

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

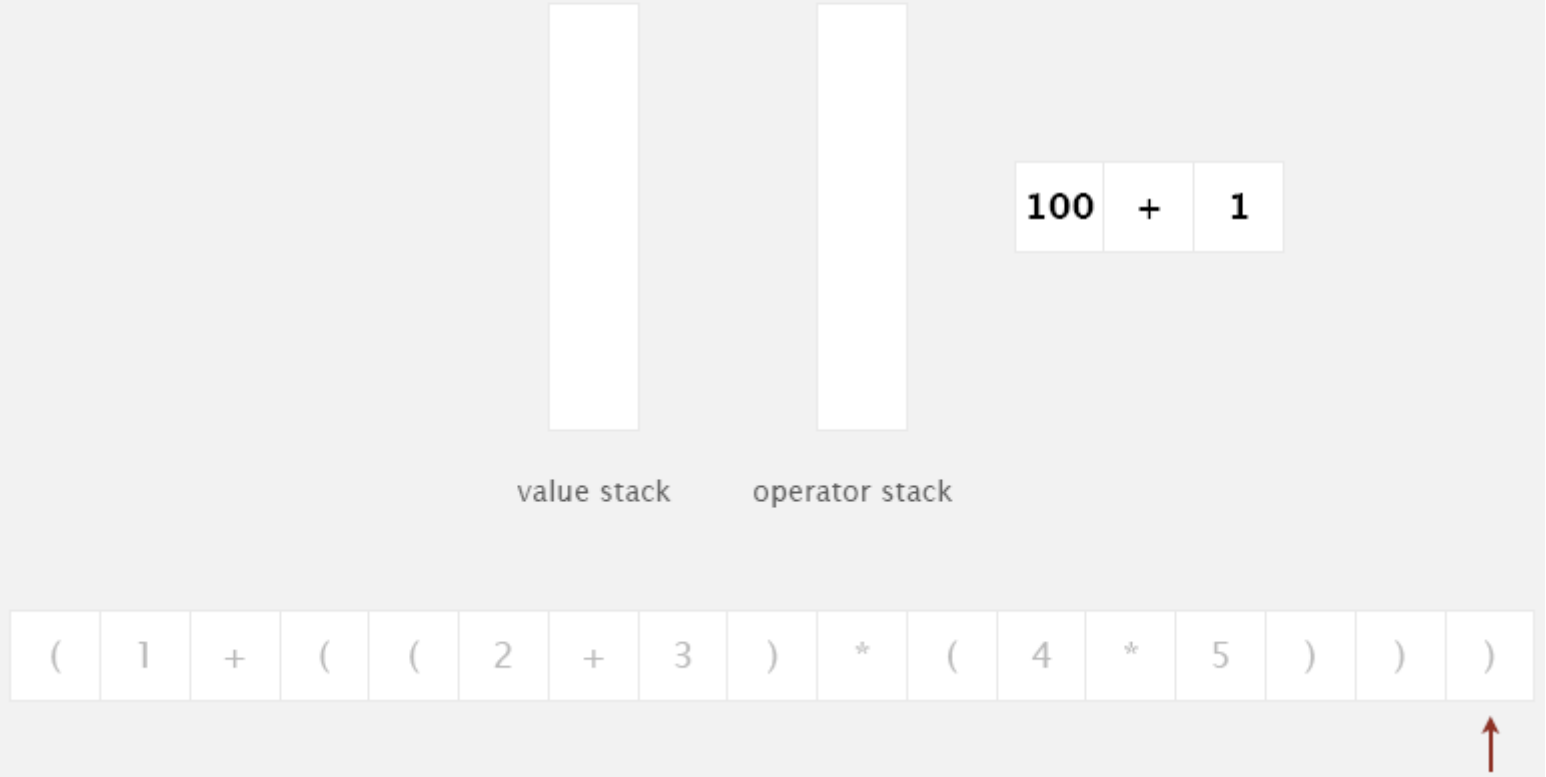
Applications: Dijkstra's two-stack algorithm 37

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Applications:
Dijkstra's two-stack algorithm 38

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack operator stack

100	+	1	=	101

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

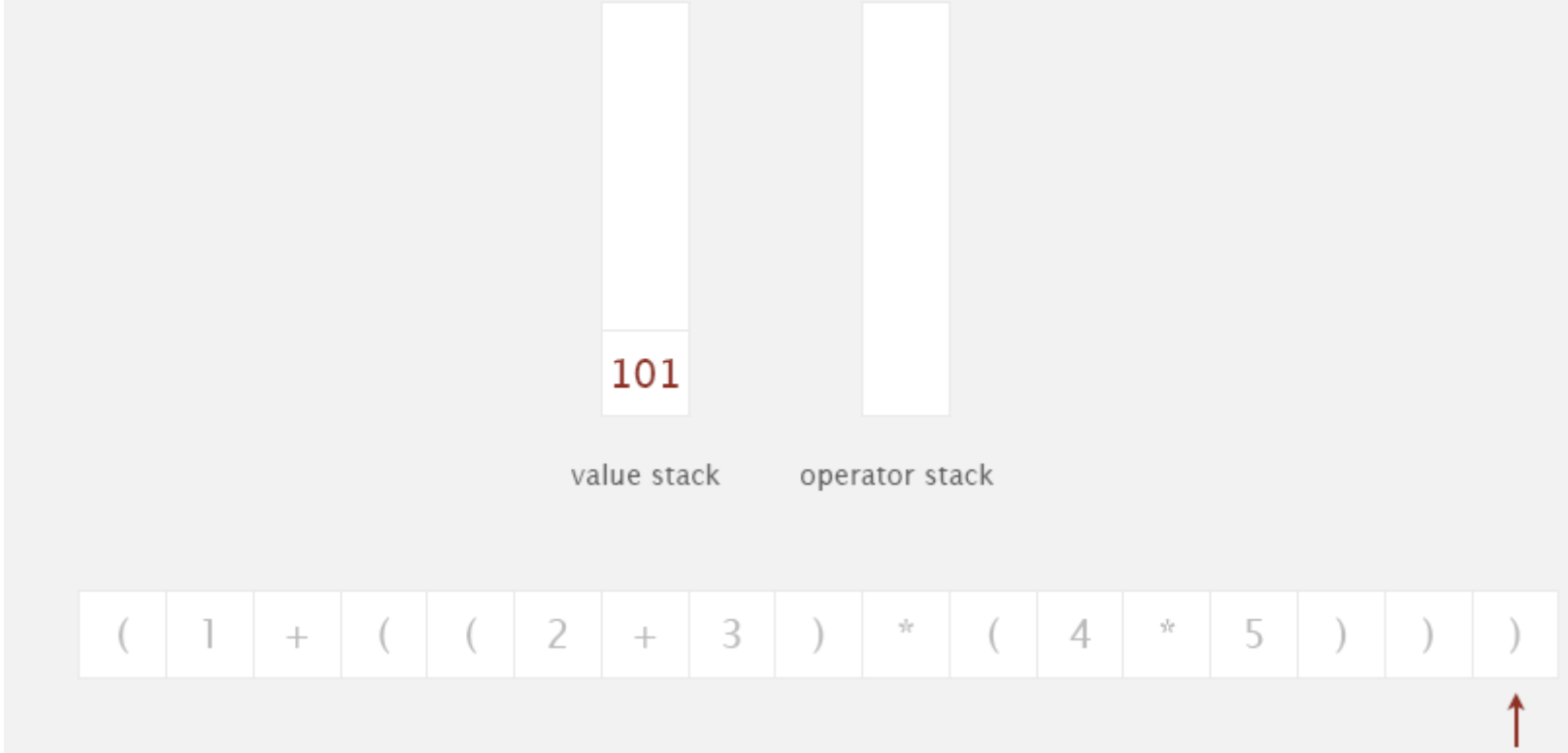
Applications:
Dijkstra's two-stack algorithm 39

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Applications:
Dijkstra's two-stack algorithm 40

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

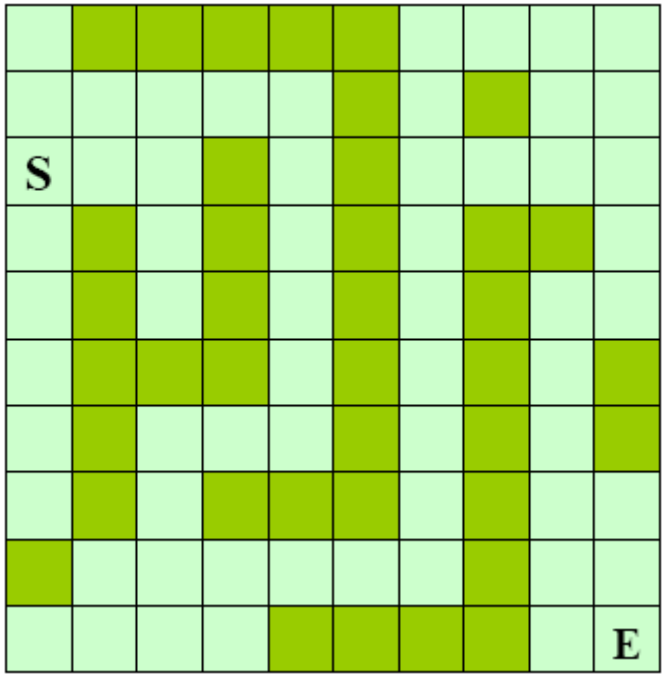
101

result

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Application: Maze

- A maze is a rectangular area with an entrance and an exit.
- The interior of a maze contains walls or obstacles that one cannot walk through.

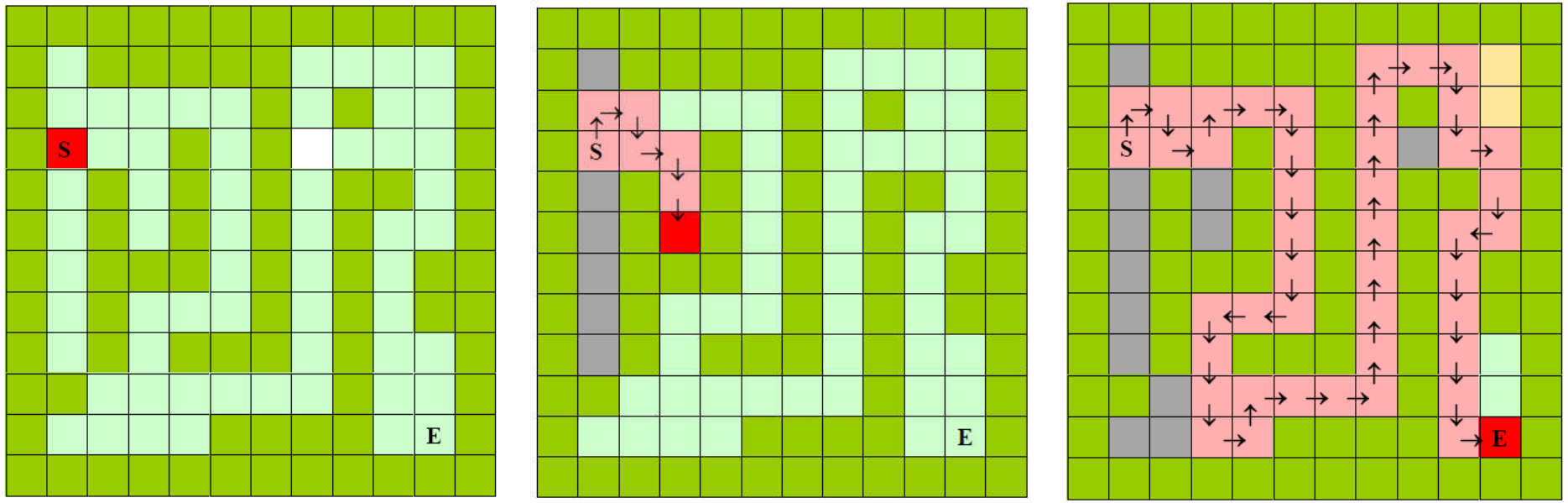


- We view the maze as broken up into equal size “open” squares,
  - some of which are part of the walls and
  - the others are part of the hallways.
- One of the open squares is designated as the Start position and another as the Exit position.

Application: Maze

- To reflect our search method for finding an exit path, we need three state:
  - OPEN, WALL, and VISITED
  - Using enumerated type .
- In traversing the maze, we can only move forward into an OPEN position .
- Thus a given position has 4 possible neighboring positions in the directions east, south, west and north.

Searching the Example Maze



Current position

Visited and pending
(on current path)

Visited and Done
(not on current path)

Maze： Search Method

- First we mark the Start position as VISITED and make it the current position.
- When exploring from a current position P, we **look for an open neighbor** in a fixed pattern: east, south, west and north.
- When an OPEN neighbor Q is found, we mark it as visited.
- In this case we say that Q was entered from P and **that P is "pending"**: we have not finished exploring from P.
- We first check to see if Q is the Exit position.
  - If so, the search concludes successfully. In this case, we print out the path from the Start to the Exit.
  - If not, we continue exploring from position Q.

Maze： Search Method

- If all four neighbors of Q are either WALL or VISITED, we return to the position P from which Q was first visited and continue exploring from P.
  - That is, we "**back out**" from Q to P. This is the only way we can move to a previously visited position.
  - Once we back out from Q, we will never return. Also, Q will not be on the path.
- If we return to the Start position and find no OPEN neighbors, the search concludes unsuccessfully: there is no path from the Start to the Exit.

Application: Maze

- How to work out the algorithm described previously.
- In particular, how do we keep track of the position from which the current position was entered?

- Consider this: Suppose we go from S to P, then P to Q, then Q to R and R to T, then at position T find no OPEN neighbors.
- Then we back out to R. Suppose R has no OPEN neighbors.
- Then we should back out to Q. If Q has no OPEN neighbors, we back out to P and then to S.

S

S→P

S →P →Q

S →P→Q →R

S →P→Q →R →T

S →P→Q →R

S →P→Q

S →P

S

Maze: Algorithm

- So we will maintain a stack of Positions.
- When we first visit a Position, we push it on the stack.
- To back out from a Position, we pop the top of the stack and continue exploring from there.

- Q: what is the path from the Entrance to the Exit?

Recursion

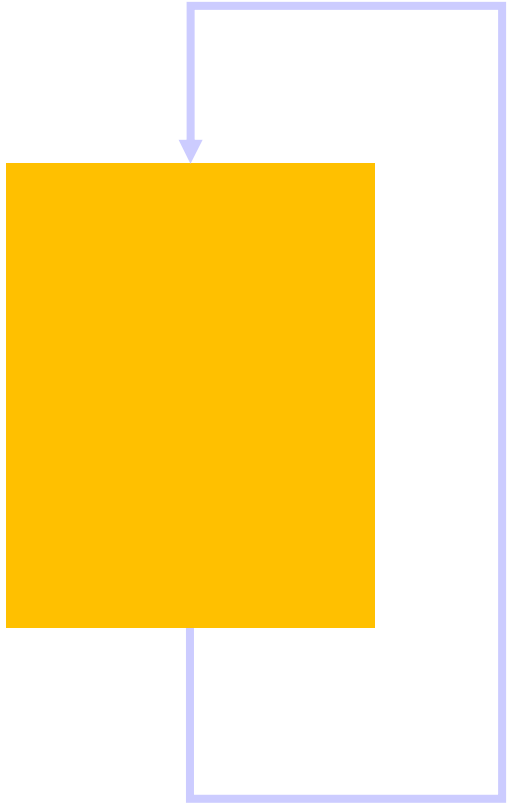
A strong relationship exists between recursion and stacks

Recursion

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first.

- Recursion is a technique that solves a problem by solving **a smaller problem of the same type.**

- An algorithm is **recursive** if it calls itself (directly or indirectly) to do part of its work.



Factorials: A Recursive Definition

- In Mathematics:

$$n! = n \times (n-1) \times \cdots \times 1.$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

Base case

General case that reduces to simpler case

Recursive process

Every recursive process consists of two parts:

1. A smallest, base case that gives a solution;
2. A general method that
   - reduces a particular case to one or more of the smaller cases,
   - reducing the problem all the way to the base case eventually.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

Factorial Functions

- Recursive Functions
  - In a program, functions that call themselves.
- Example

```
public int f(int a){
        if (a==1)
          return(1);
        else
          return(a * f( a-1));          It computes f! (factorial)
}
```

+: concise and elegant

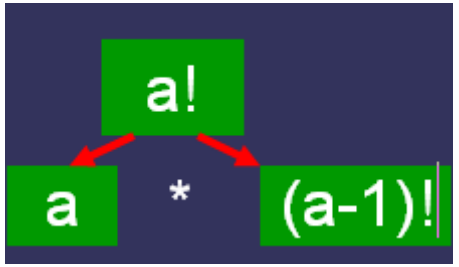
-: to require keeping track of many partial computations

Factorial

- Note:

$$a! = a * (a-1)!$$

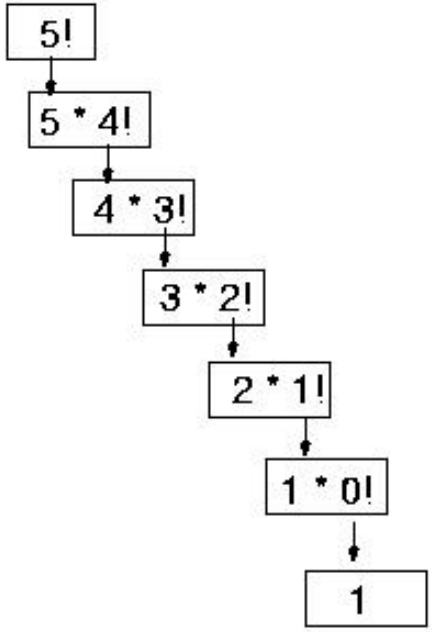
- remember:
  - splitting up the problem into a smaller problem of the same type

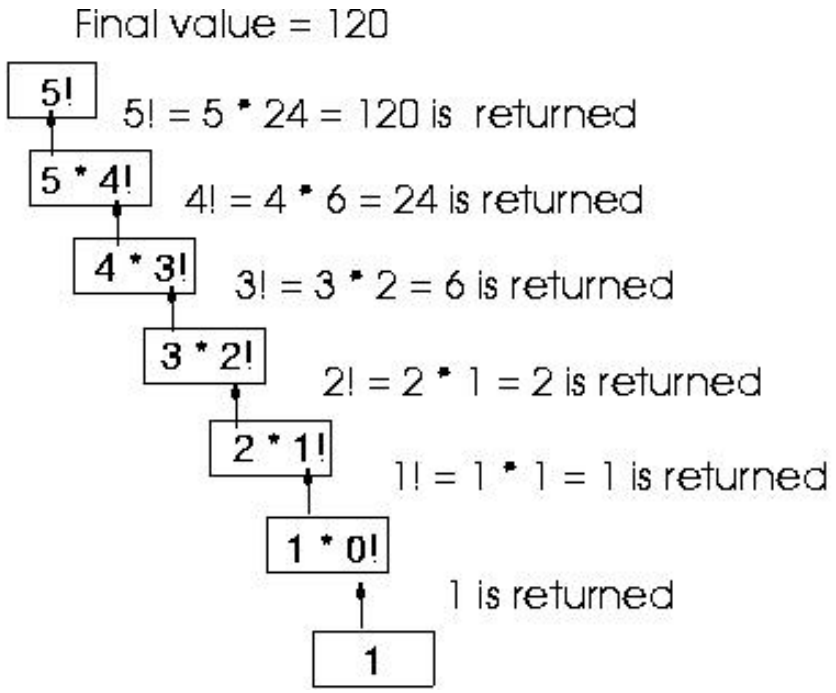


Tracing the example

```
int factorial(int a){
        if (a==0)
          return(1);
        else
          return(a * factorial( a-1));
}
```

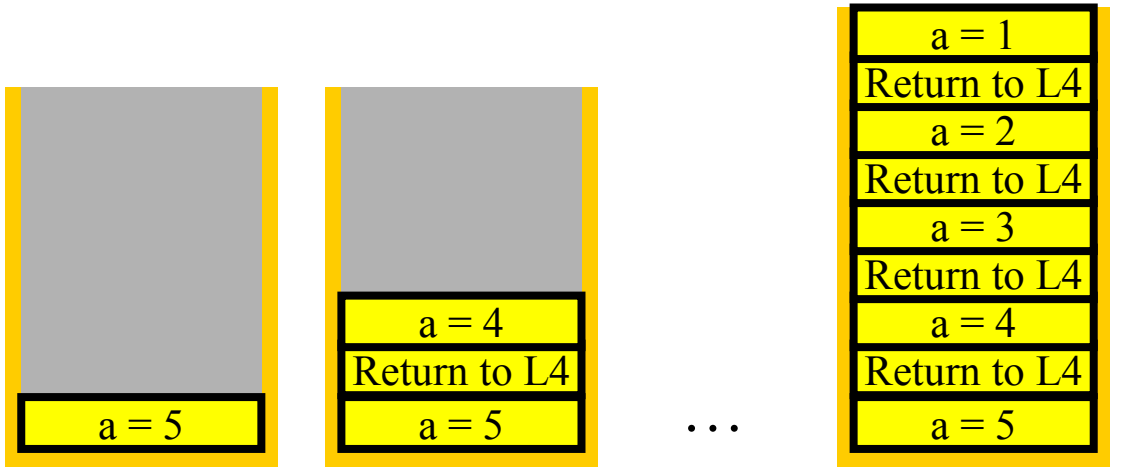
RECURSION !





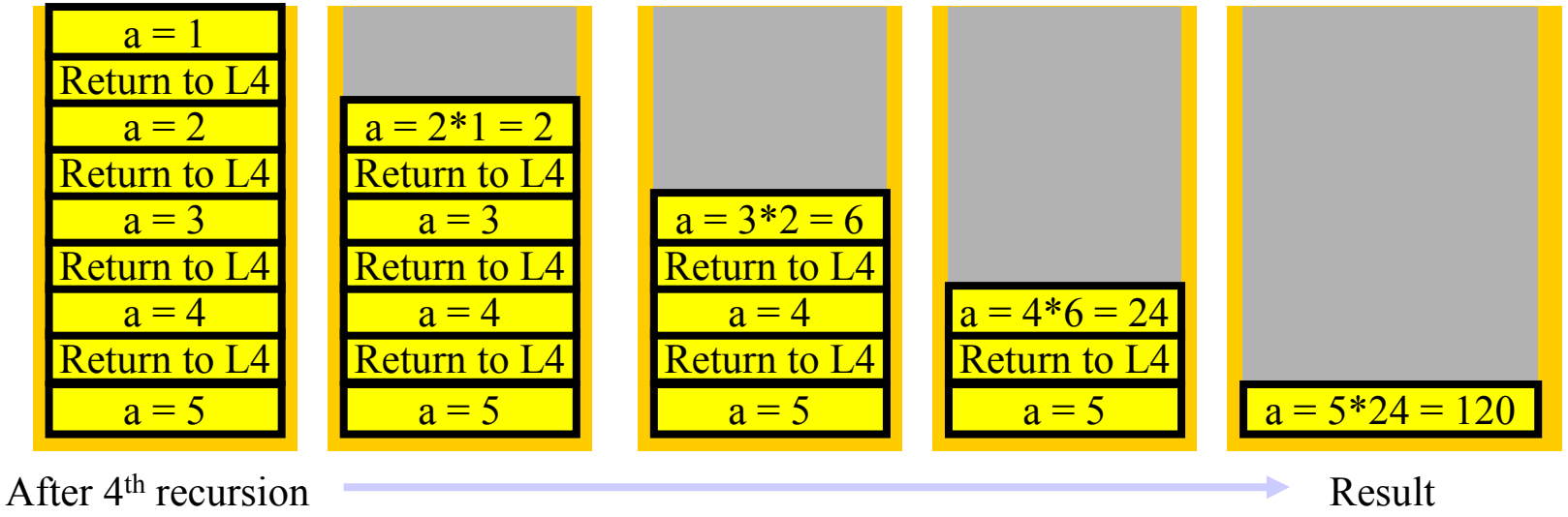
Watching the Stack

```
int factorial(int a){
L1        if (a==0)
L2          return(1);
L3        else
L4          return(a * factorial( a-1));
}
```



Every call to the method creates
a new set of local variables !

```
int factorial(int a){
L1          if (a==0)
L2             return(1);
L3          else
L4             return(a * factorial( a-1));
}
```



Properties of Recursive Functions

Problems that can be solved by recursion have these characteristics:

- One or more stopping cases have a simple, nonrecursive solution
- The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases
- Eventually the problem can be reduced to only stopping cases, which are relatively easy to solve

Follow these steps to solve a recursive problem:

- Try to express the problem as a simpler version of itself
- Determine the stopping cases
- Determine the recursive steps

Solve a recursive problem

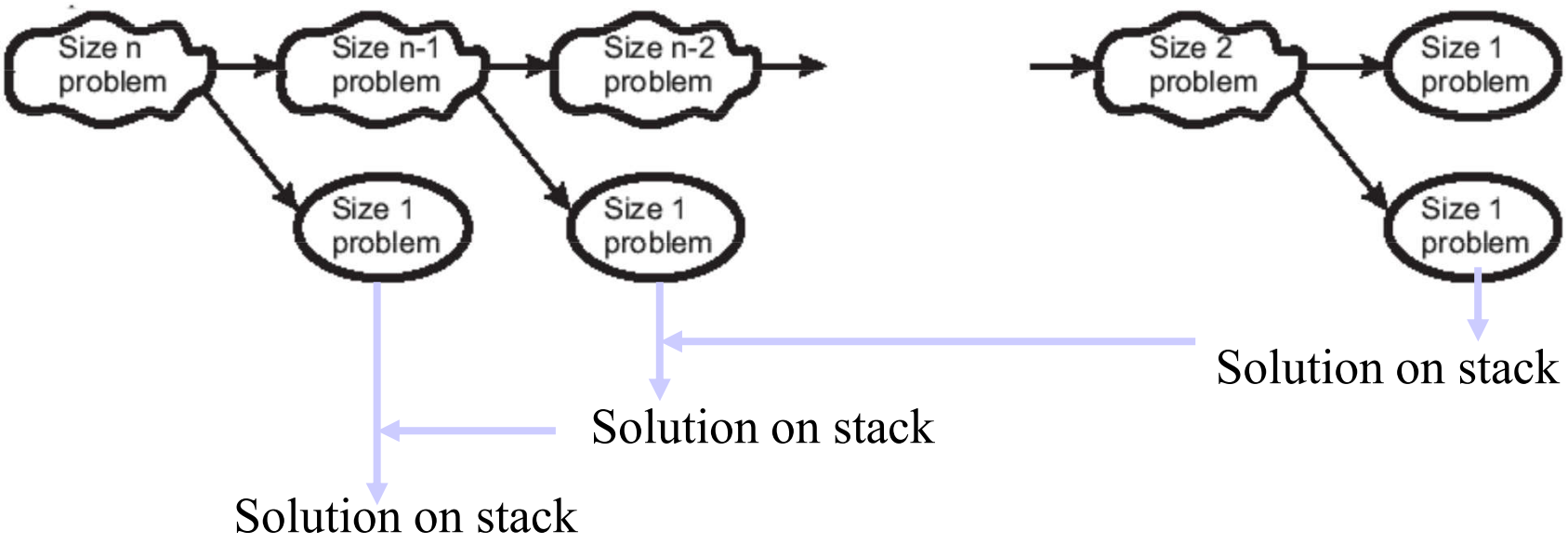
The recursive algorithms we write generally consist of an if statement:

IF

the stopping case is reached solve it

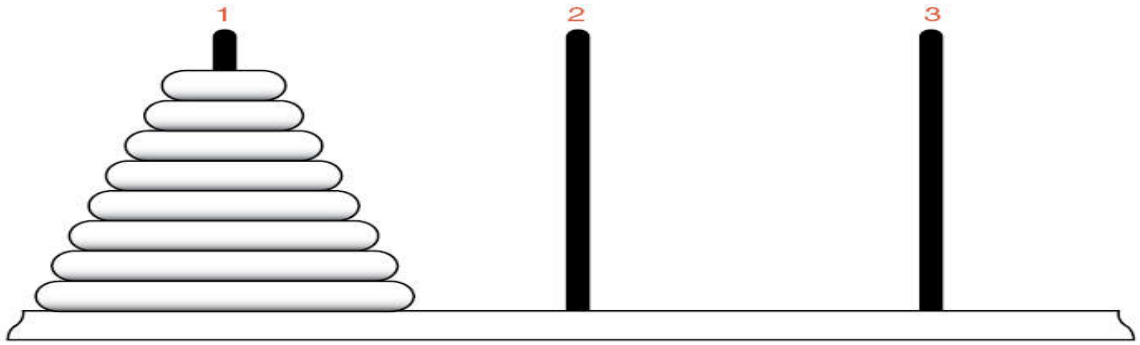
ELSE

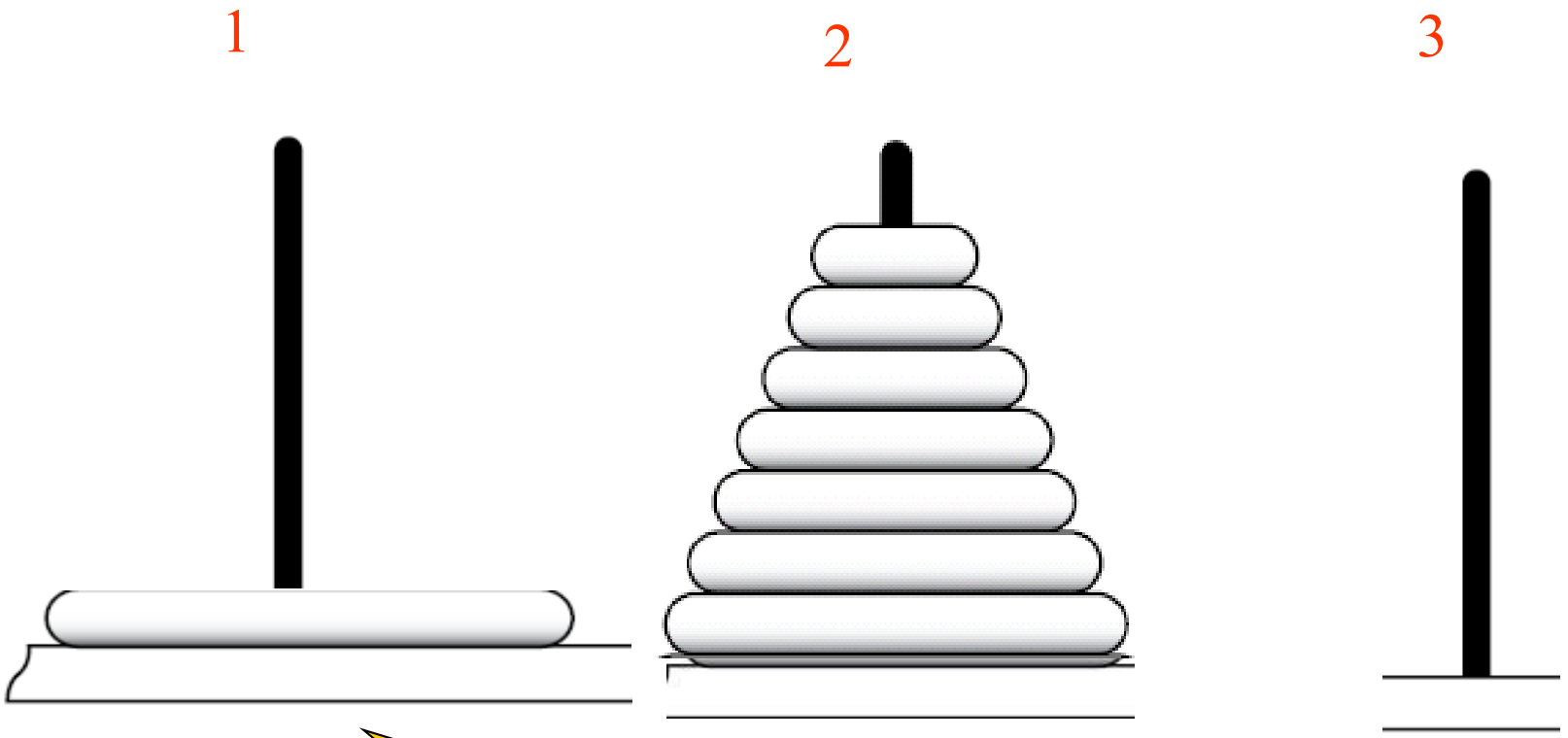
split the problem into simpler cases using recursion



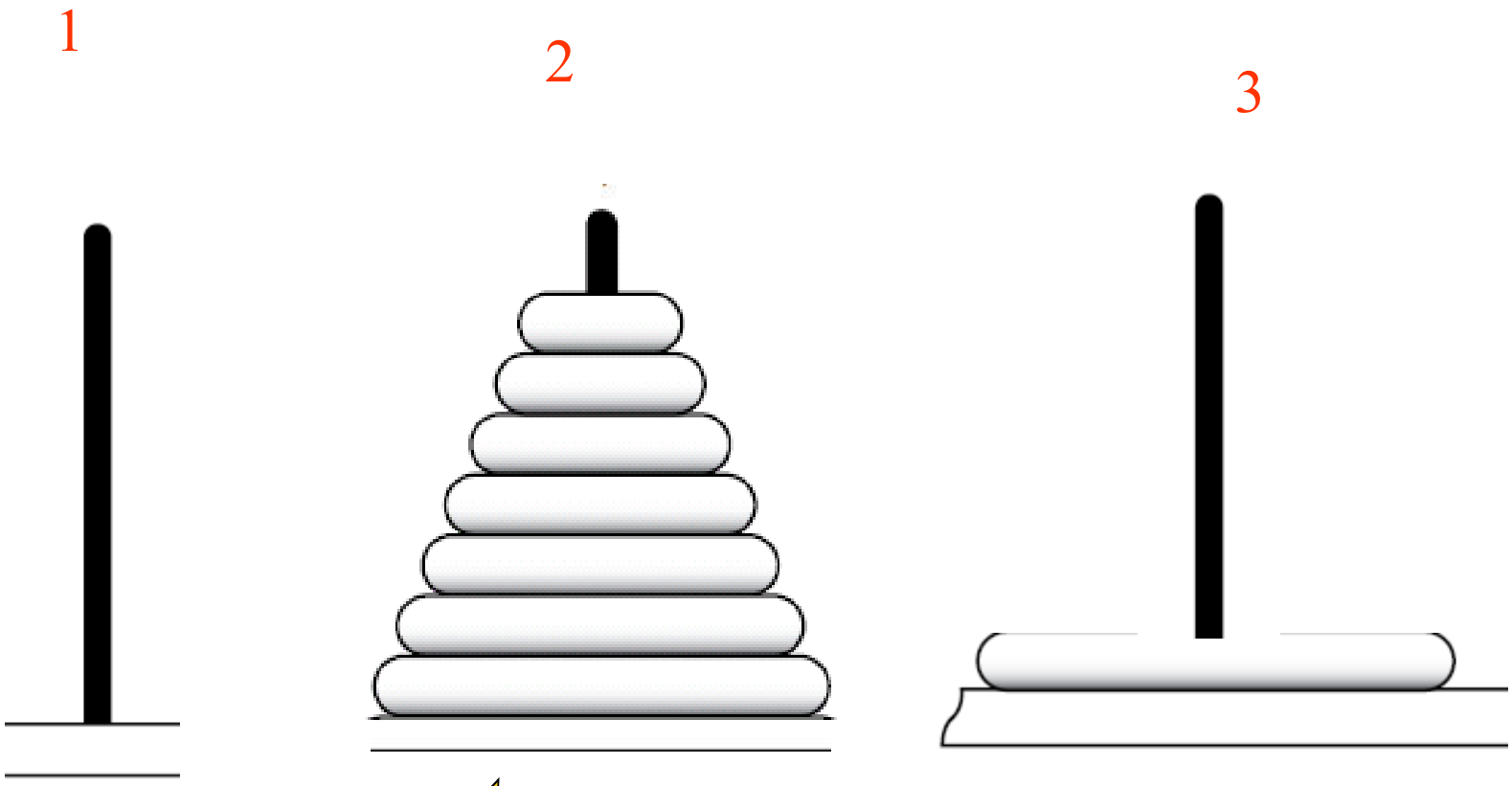
The tower of Hanoi

- **Task**: move the tower from 1 to 3 using 2 as temporary storage.
- **Rules**:
  1. move only one disk at one time;
  2. No larger disk can be placed on a smaller disk.

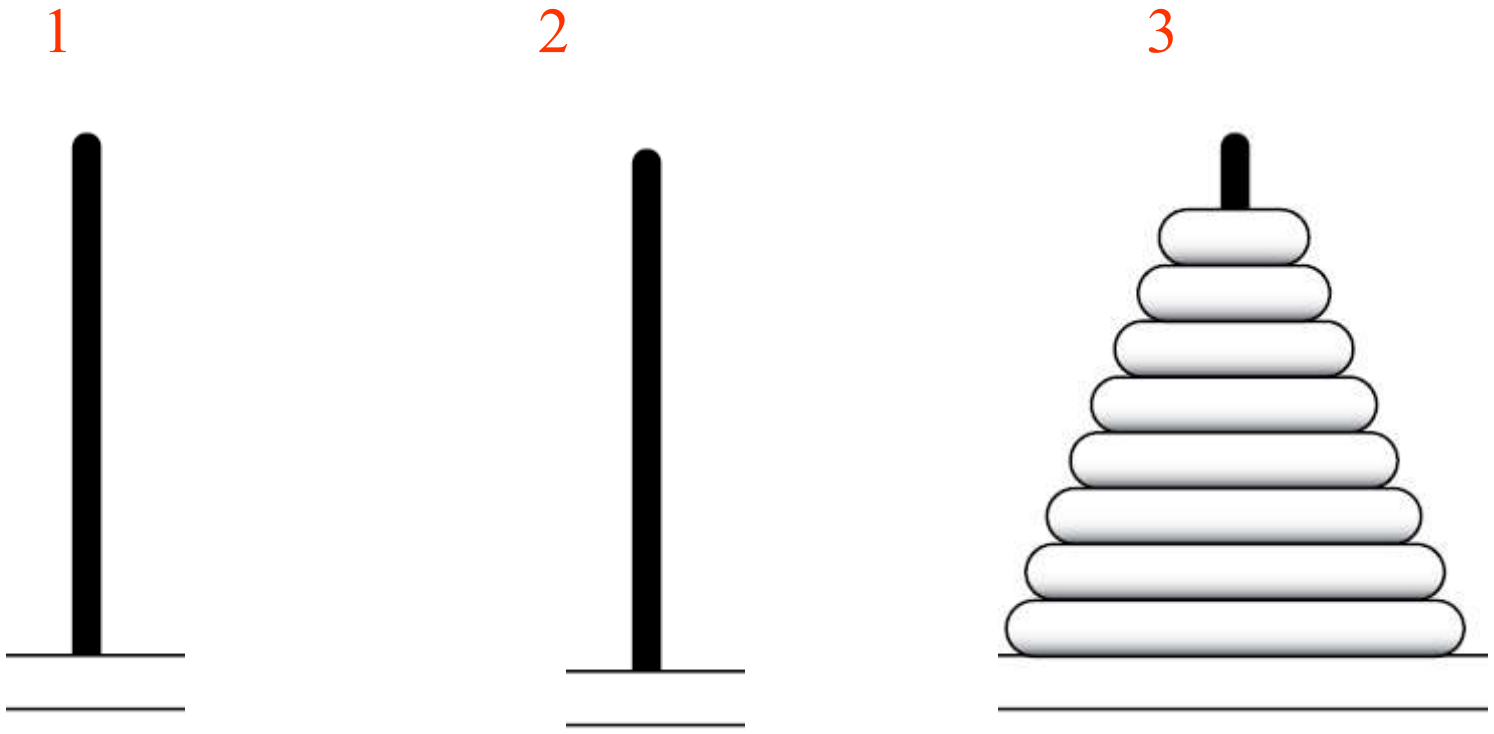




We have to get to this state and then move the bottom disk



A similar problem



Solved by recursion

Solutions

Void move(int *count*, int *start*, int *temp*, int *finish*)

- **Pre:** there are at least ***count*** disks on the tower ***start***. The top disk (if any) on each of towers *temp* and *finish* is larger than any of the top *count* disks on tower *start*.

- **Post:** The top ***count*** disks on ***start*** have been moved to ***finish***; ***temp*** (used for temporary storage) has been returned to its starting position

Hanoi tower: Instance

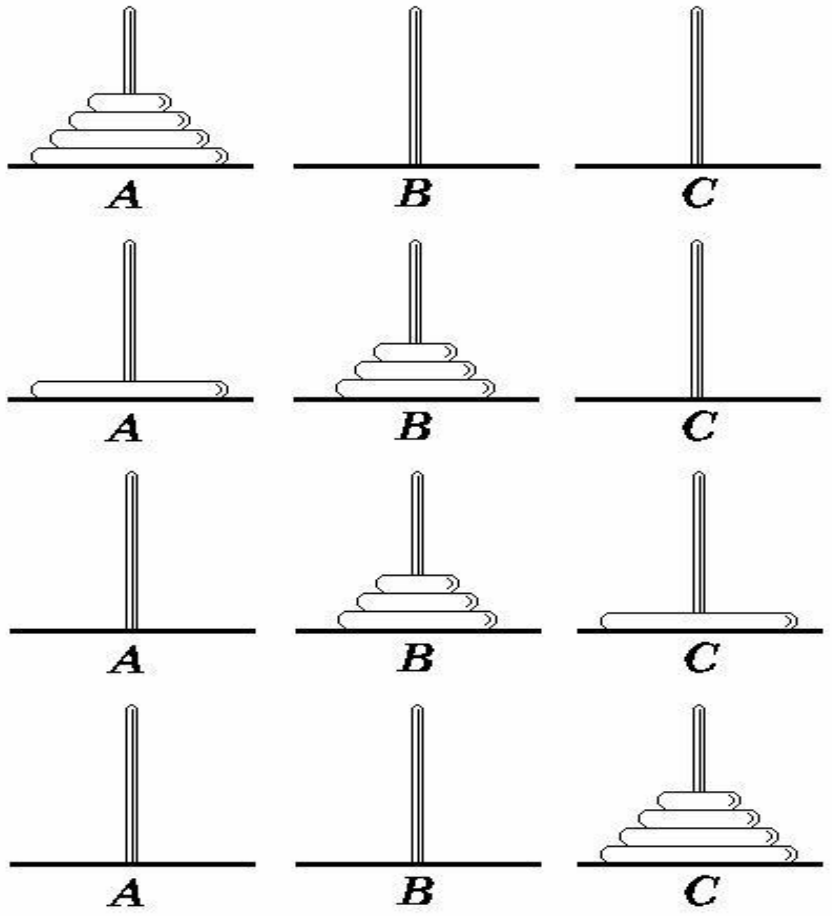
Our goal: move(8, 1, 3, 2)

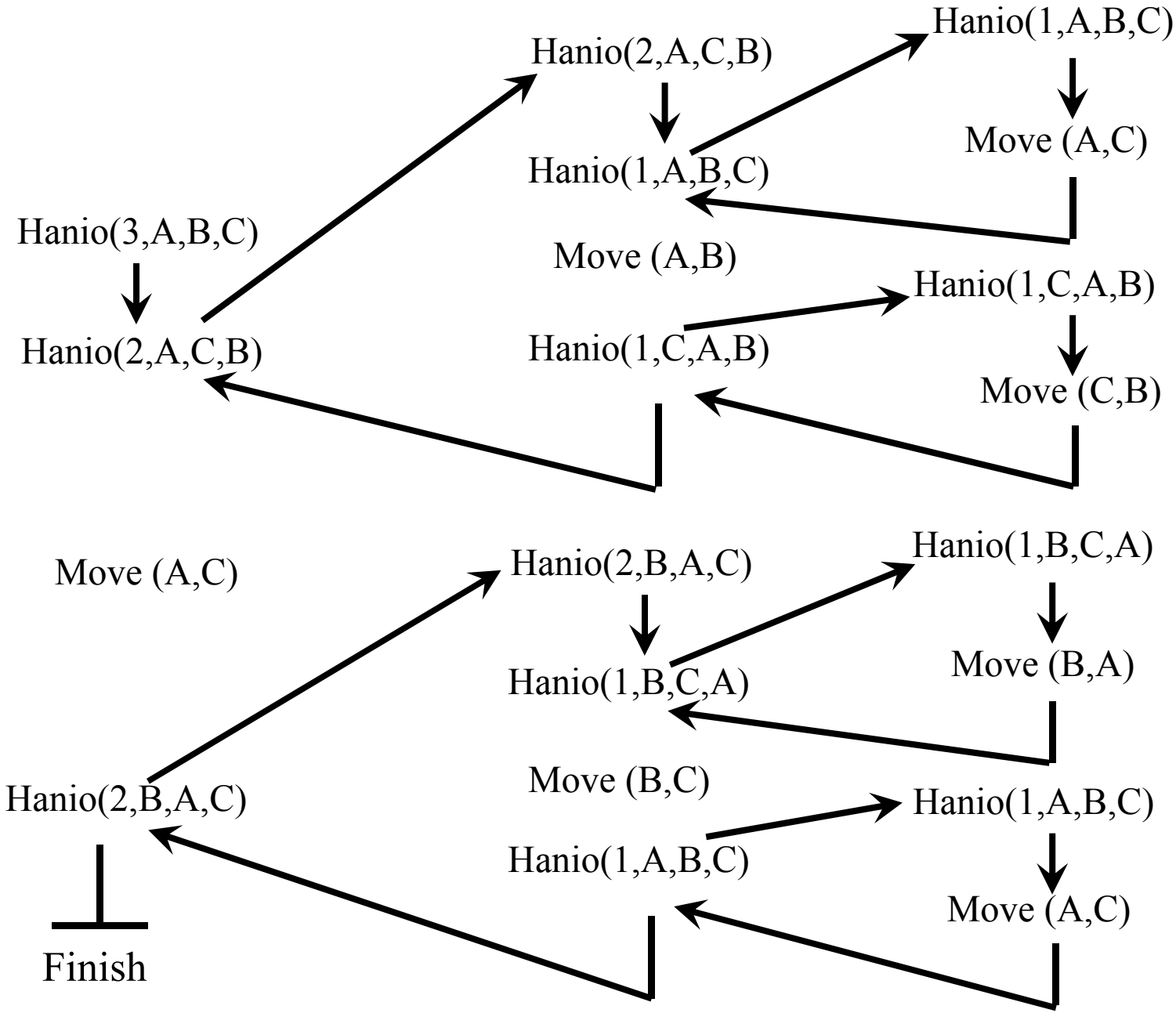
Steps to solve the problem:

1. move(7, 1, 2, 3); /*move 7 disks from tower 1 to 2 */
2. cout << “move disk 8 from tower 1 to 3.”<<endl;
3. move(7, 2, 3, 1); /* move 7 disks from tower 2 to 3 */

Solution-2: Recursive Function

```
void Hanoi(int n, char A, char B, char C)
{
    if (n==1)  Move(A, C);
     else {
            Hanoi(n-1, A, C, B);
            Move(A, C);
            Hanoi(n-1, B, A, C);
     }
}
```




Hanio(3,A,B,C)
Hanio(2,A,C,B)
Hanio(2,A,C,B)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
Move (A,B)
Hanio(1,C,A,B)
Hanio(1,C,A,B)
Move (C,B)
Move (A,C)
Hanio(2,B,A,C)
Hanio(2,B,A,C)
Hanio(1,B,C,A)
Hanio(1,B,C,A)
Move (B,A)
Move (B,C)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
Finish

Stack applications

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

Queue

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

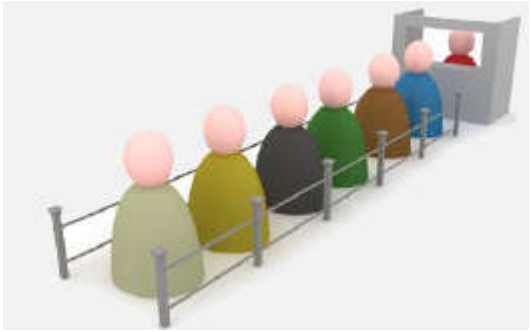
tianyi.zang@gmail.com

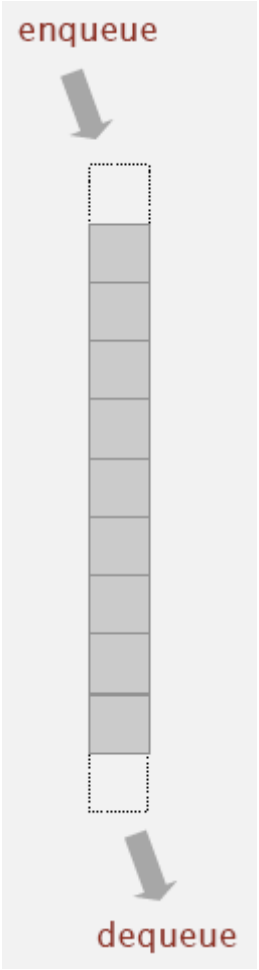
Outline

- Stack ADT
- Array-Based Stack
- Linked Stack
- Comparison of Array-Based and Linked Stacks
- Applications

Queues

- Like the stack, the queue is a list-like structure that provides restricted access to its elements.
- Queue elements may only be inserted at the back
  - called an *enqueue* operation and
- Removed from the front
  - called a *dequeue* operation



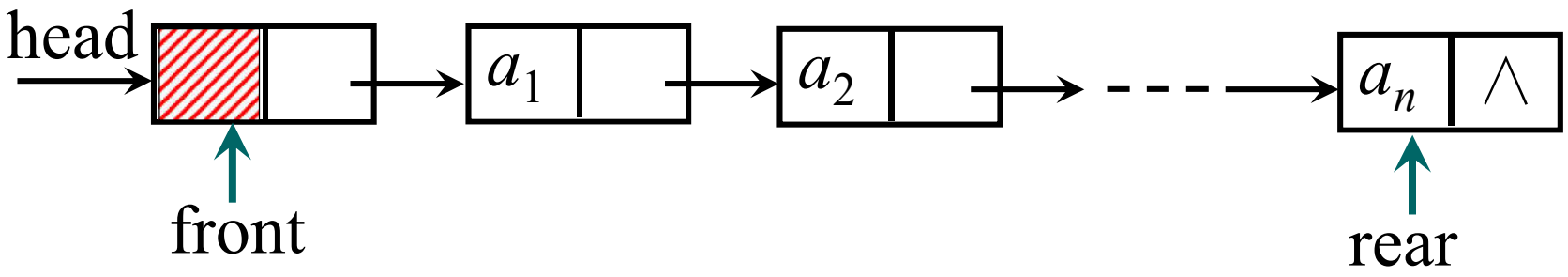


Queues

- Property:
  - FIFO: First in, First Out
  - Restricted form of list: Insert at one end, remove from the other.
- Notation:
  - Insert: EnQueue; Delete: DeQueue
  - First element: Front; Last element: Rear
- Operations:
  - MakeNull(Q) //Reinitialize the queue.
  - Front(Q) // Return: A copy of the front element.
  - EnQueue(x, Q) // Place an element x at the rear of the queue.
  - DeQueue(Q) // Remove and return element at the front of the queue.
  - Empty(Q) // return *true* is the queue Q is empty, *false* otherwise

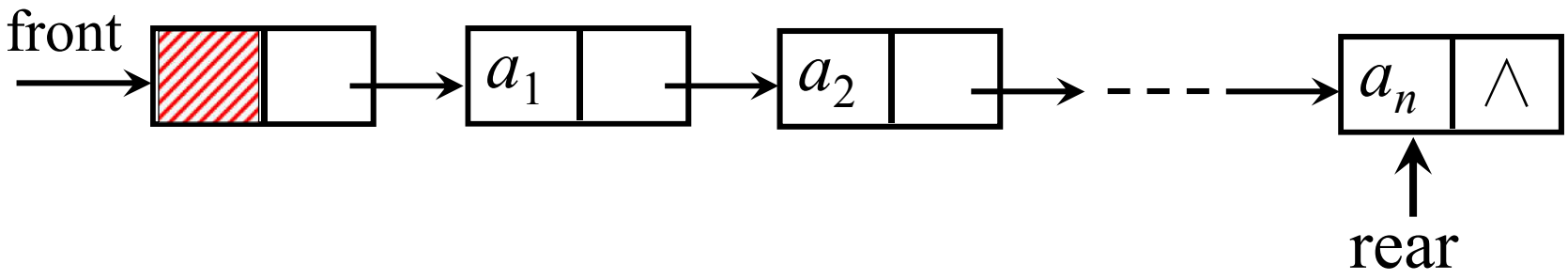
Linked Queues Implementation

- A straightforward adoption of the linked list.
- The *front* will always points to the header node while the *rear* points to the true last link node in the queue.

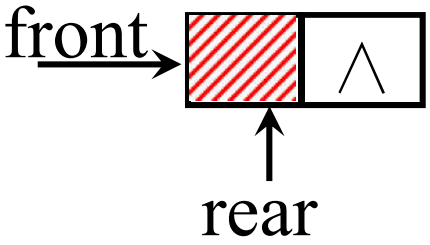


Storage structure

- None empty



- Empty



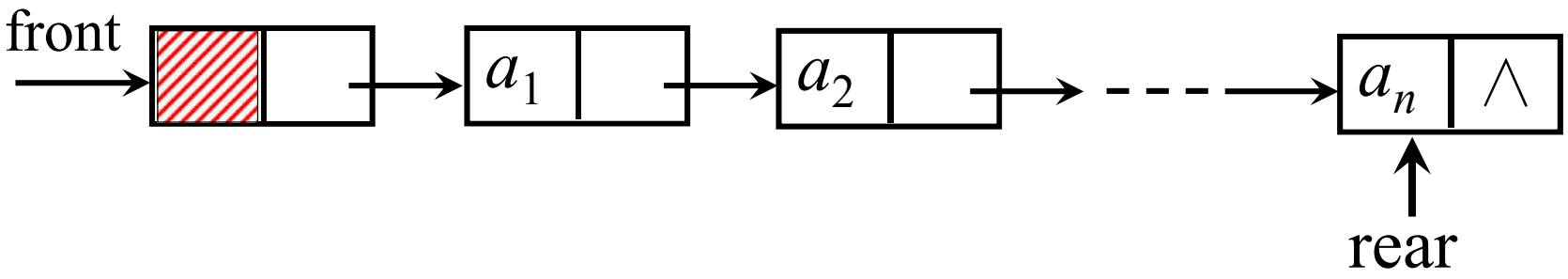
Storage structure definition

- // the node type

```
struct  celltype  {
      ElemType  data;
       celltype  *next;
} ;
```

- // the queue type

```
struct  QUEUE {
      celltype  *front;
      celltype  *rear;
};
```



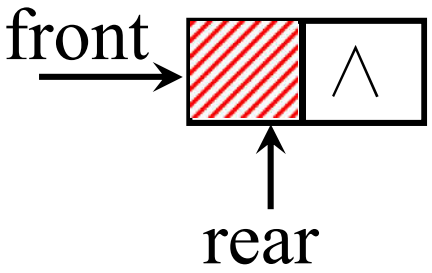
The operations implementation

① void MakeNull(QUEUE &Q)

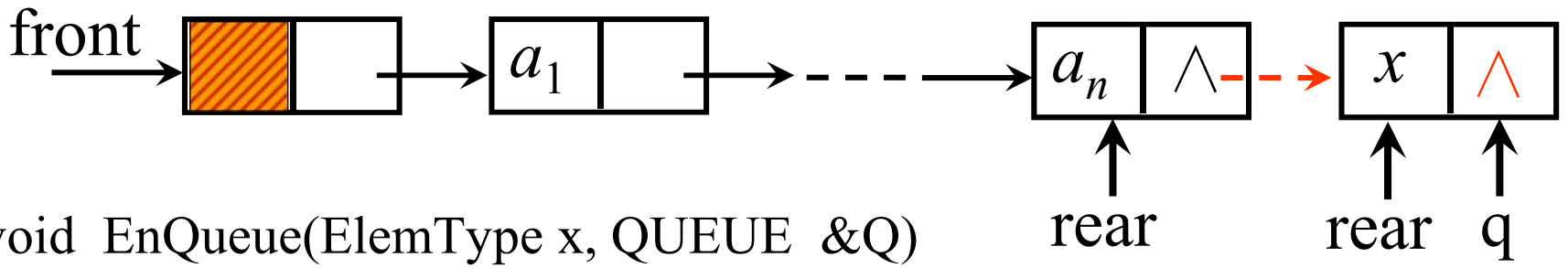
```
{
    Q.front = new celltype ;
    Q.front→next = NULL ;
    Q.rear = Q.front ;
}
```

② Boolean Empty(QUEUE &Q)

```
{  if ( Q.front == Q.rear )
        return TRUE ;
   else
        return FALSE ;
}
```



The operations: EnQueue



③ void EnQueue(ElemType x, QUEUE &Q)

{

q=new cwlltype;

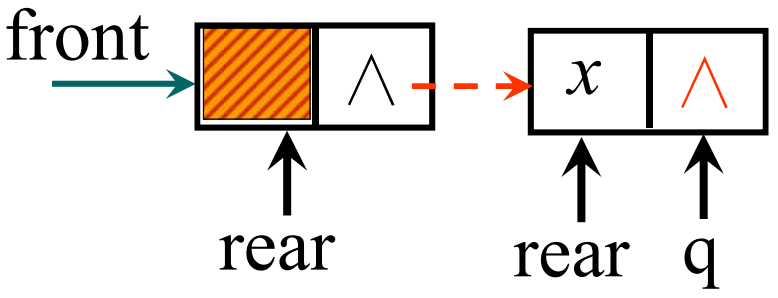
q->data=x ;

q->next=NULL;

Q.rear->next=q;

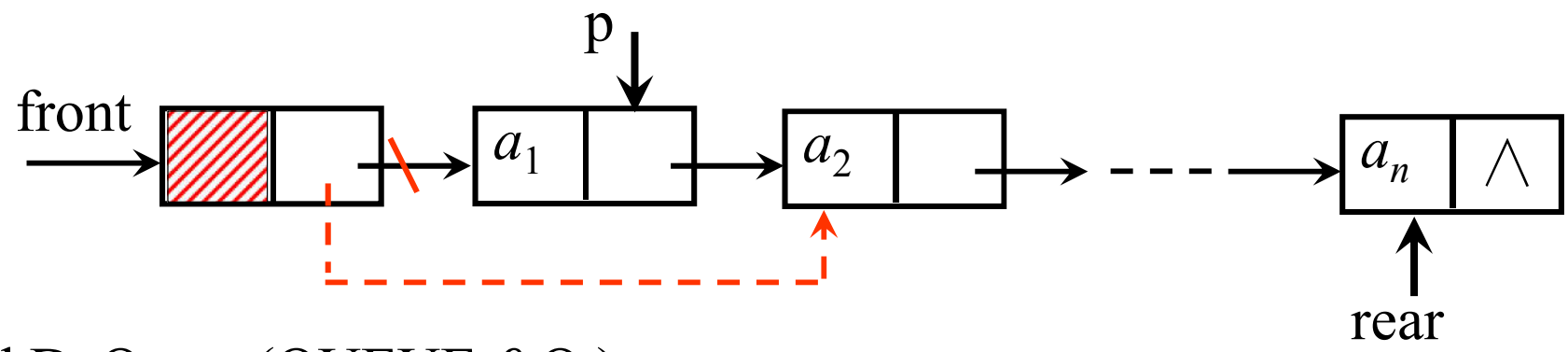
Q.rear=q;

}

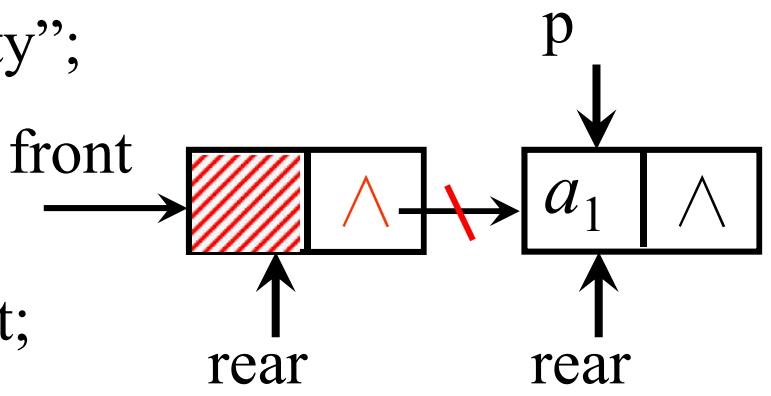


How about without head node?

The operations: DeQueue

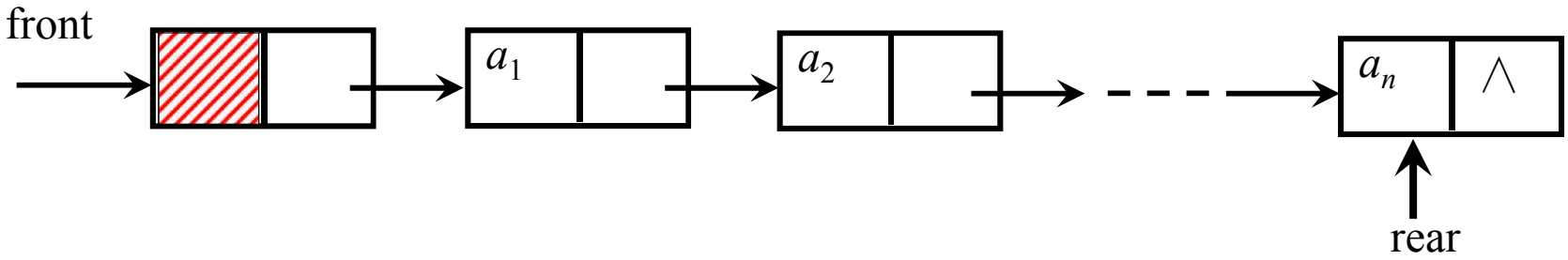



```
void DeQueue(QUEUE &Q )
{    if (Q.rear==Q.front) cout<<“Emputy”;
     p=Q.front->next;
     Q.front->next=p->next;
    if (p->next==NULL) Q.rear=Q.front;
     delete p;
}
```



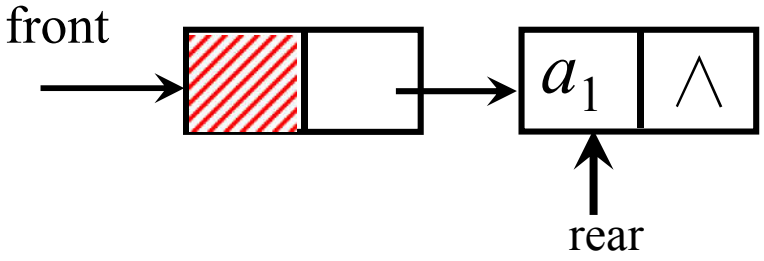
Boundary condition: only one element in the queue?

The operations: Front



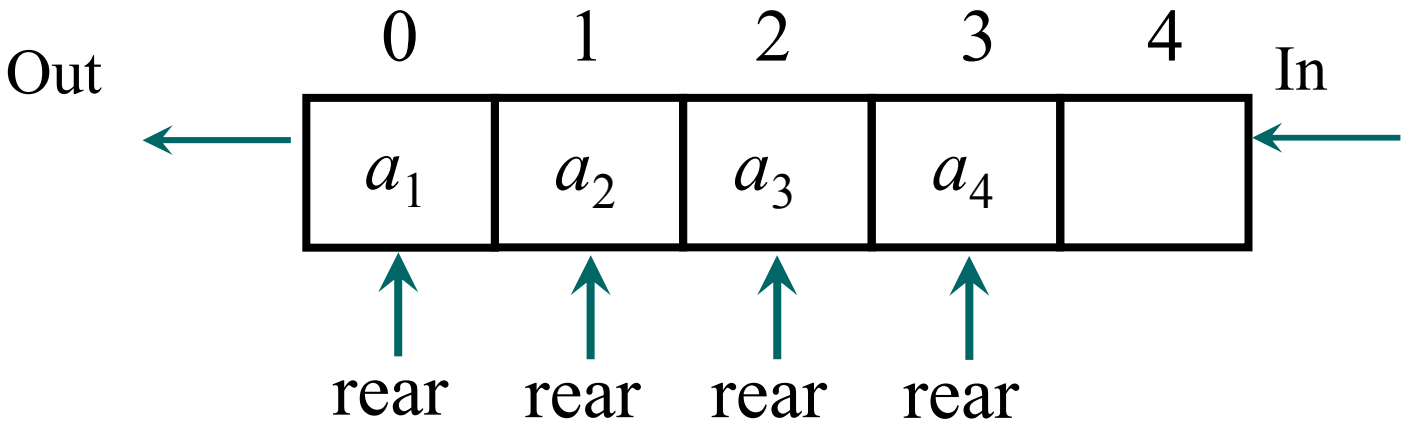
③ ElemType Front (QUEUE Q)

```
{    if ( Q.front→next )
        return  Q.front→next→ data ;
}
```



Array-based Implementation

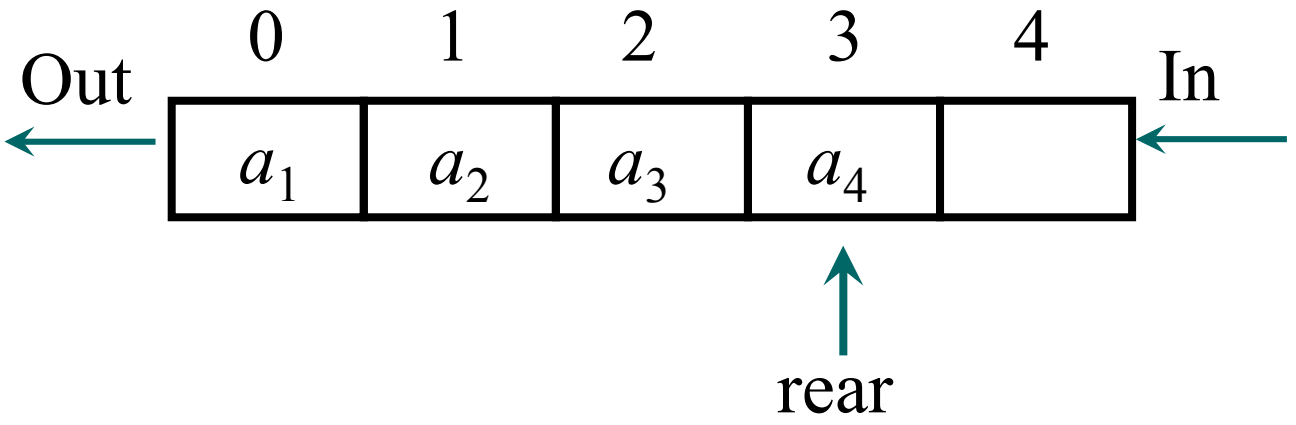
•EnQueue $a_1a_2a_3a_4$ in turn.



The time complexity of EnQueue is: O(*1*)

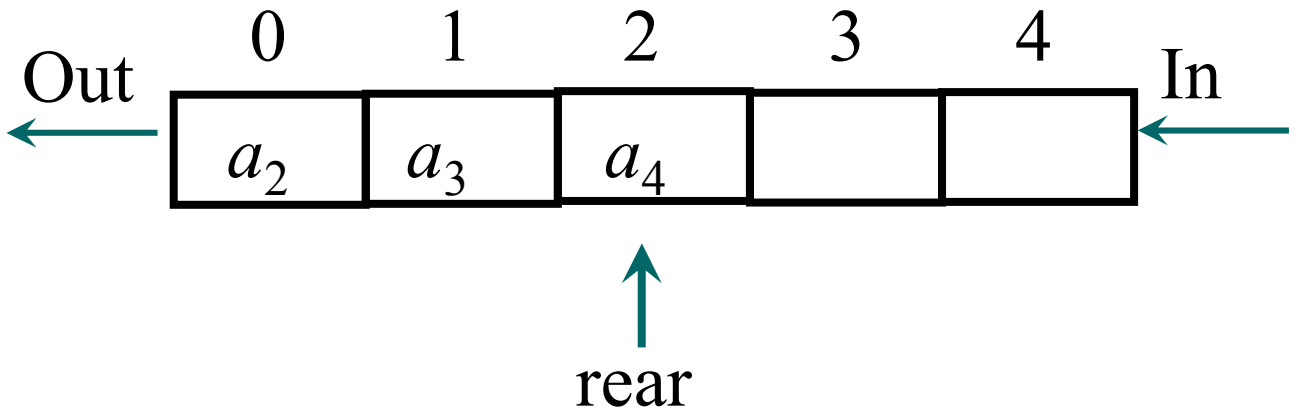
Array-based Implementation

- DeQueue $a_1 a_2$ in turn



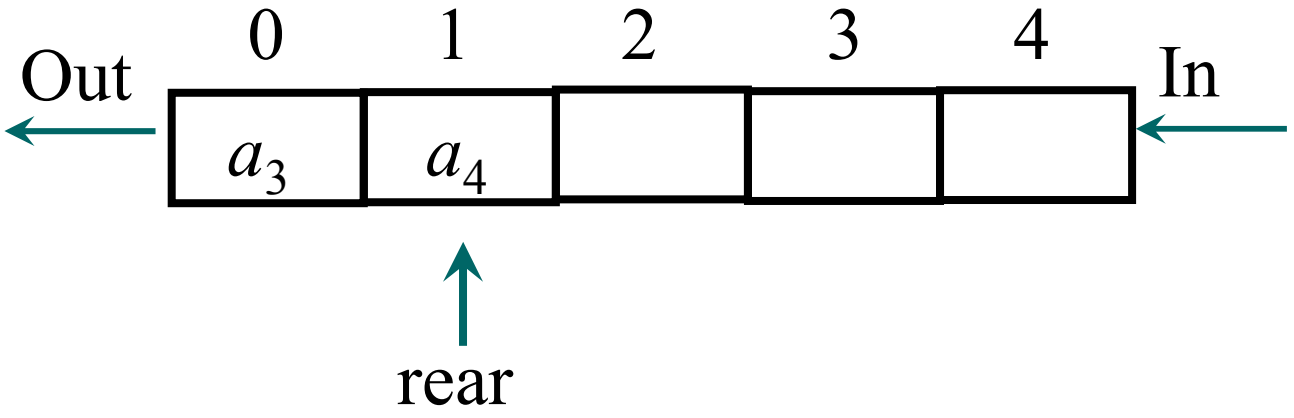
Array-based Implementation

- DeQueue $a_1 a_2$ in turn



Array-based Implementation

- DeQueue $a_1 a_2$ in turn

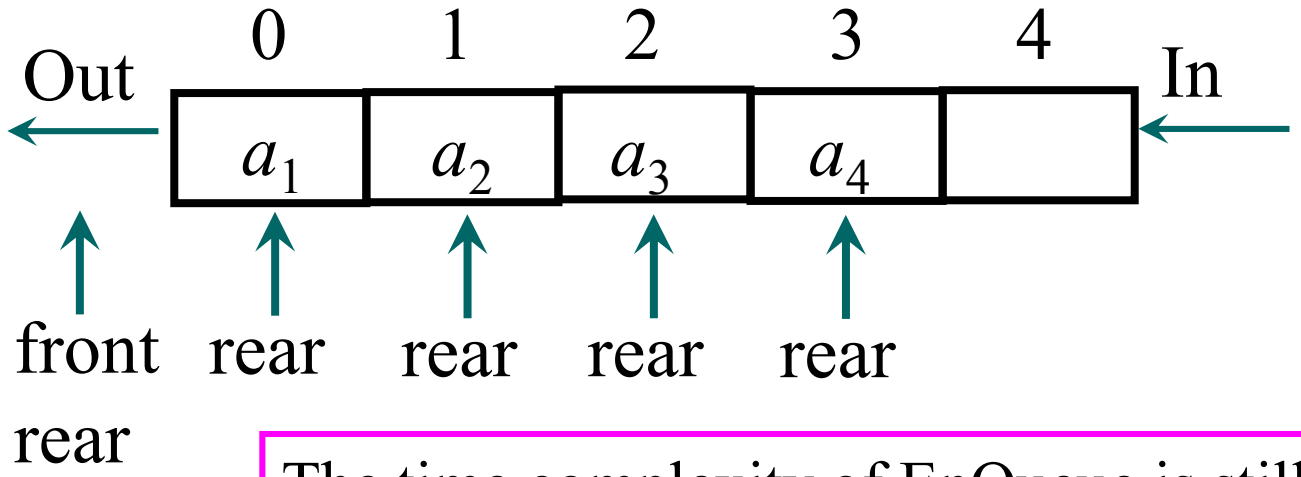


The time complexity of DeQueue: O(n)

Array-based Implementation

How to improve the performance? (1)

- Two pointers of *front* and *rear* are introduced to point to the head and the tail of the queue.
- EnQueue $a_1a_2a_3a_4$ in turn:

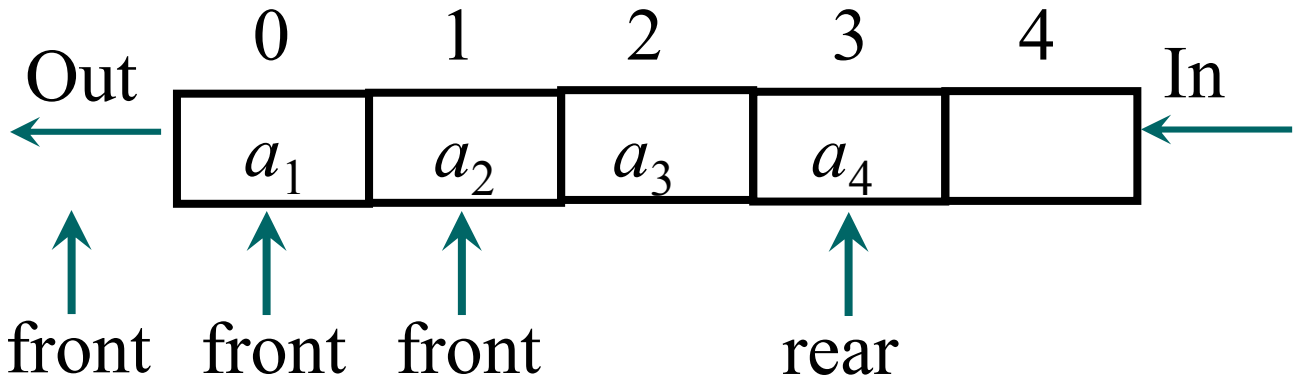


The time complexity of EnQueue is still $O(1)$

Array-based Implementation

How to improve the performance? (1)

- Two pointers of *front* and *rear* are introduced to point to the head and the tail of the queue.
- DeQueue $a_1 a_2$ in turn:

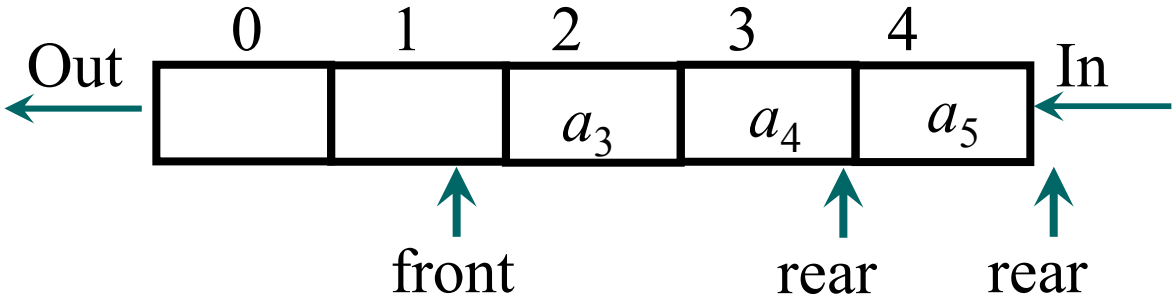


The time complexity of DeQueue is improved as $O(1)$

Array-based Implementation

How to improve the performance? (1)

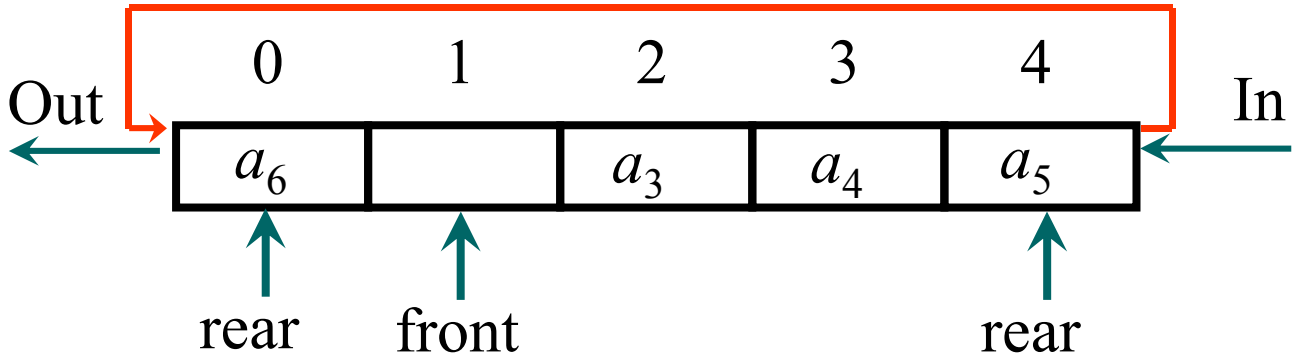
- This implementation raises a new problem.
  - When elements are added and removed, the entire queue will drift toward the end.
  - Eventually, there will be no space to the right of the queue, even though there is space in the array.
  - *"drifting queue"*



Array-based Implementation

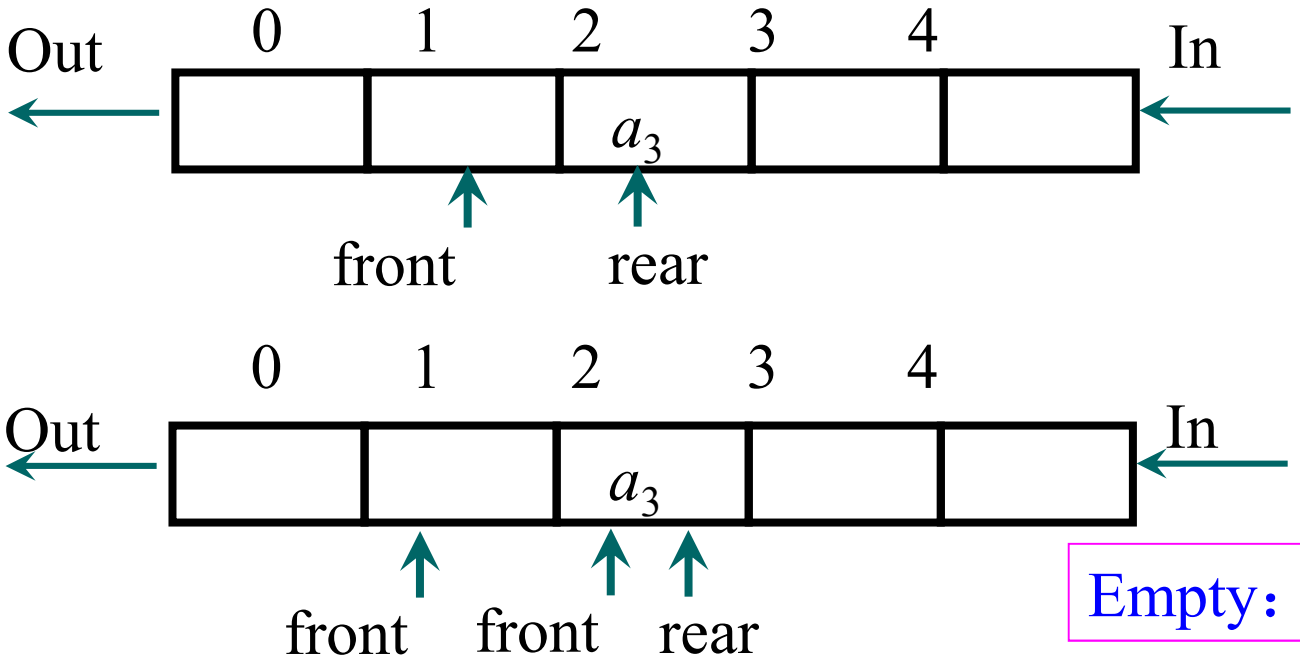
How to improve the performance? (2)

- The "drifting queue" problem can be solved by pretending that **the array is circular**, and so
- allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position



Circular Queue

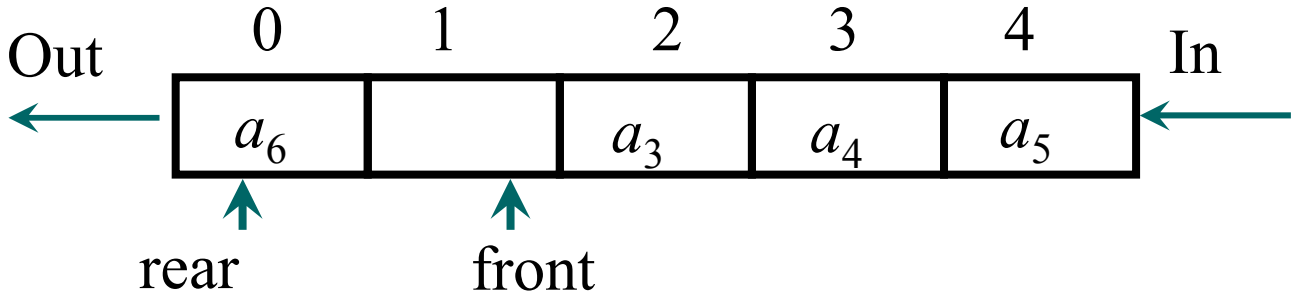
- How can we recognize when the circular queue is empty or full?
  - empty

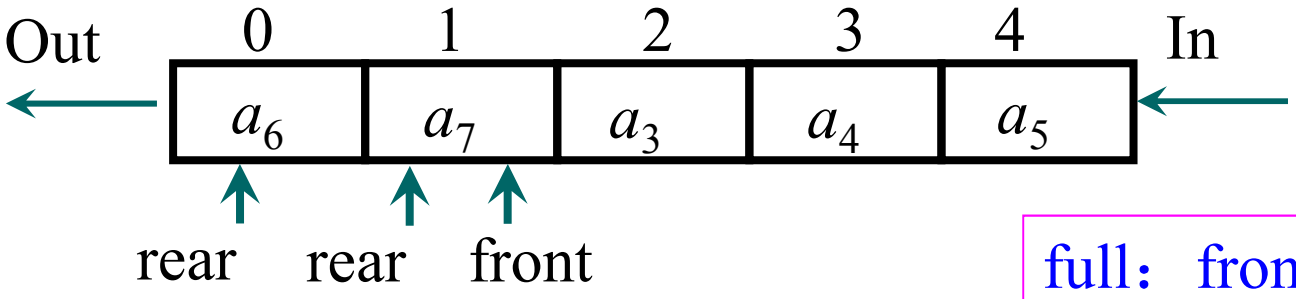


Empty：front==rear

Circular Queue

- How can we recognize when the circular queue is empty or full?
  - full





full：front==rear

Circular Queue

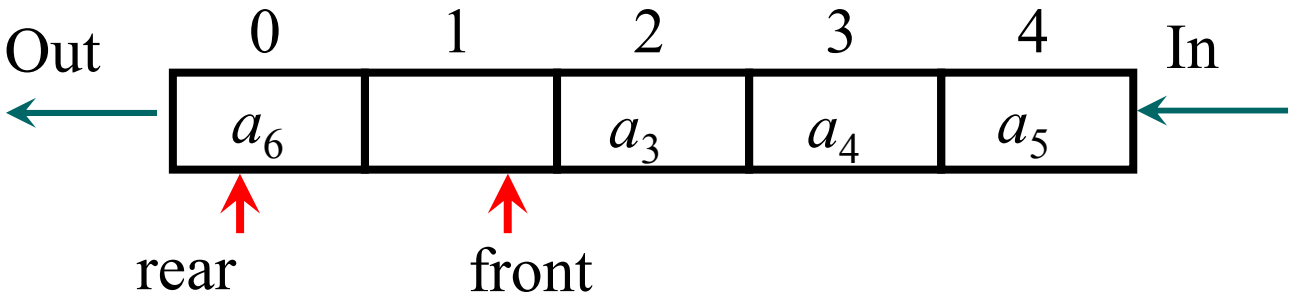
Recognize when the circular queue is empty or full:

- One obvious solution is to keep **an explicit count of the number** of elements in the queue.
  - If front==rear and count==0, then the queue is empty
  - If front==rear and count==MaxSize, then full
- A Boolean variable ***flag*** that indicates whether the queue is empty or not.
  - If front==rear and flag==0, then empty
  - If front==rear and flag==1, then full
- Another solution is to make the array be **of size *n*+*1***, and only allow *n* elements to be stored.
  - If front==rear, then empty
  - If ((rear+1)%MaxSize==front) then full
  - There is an idle item when the queue is full.

Operation implementations

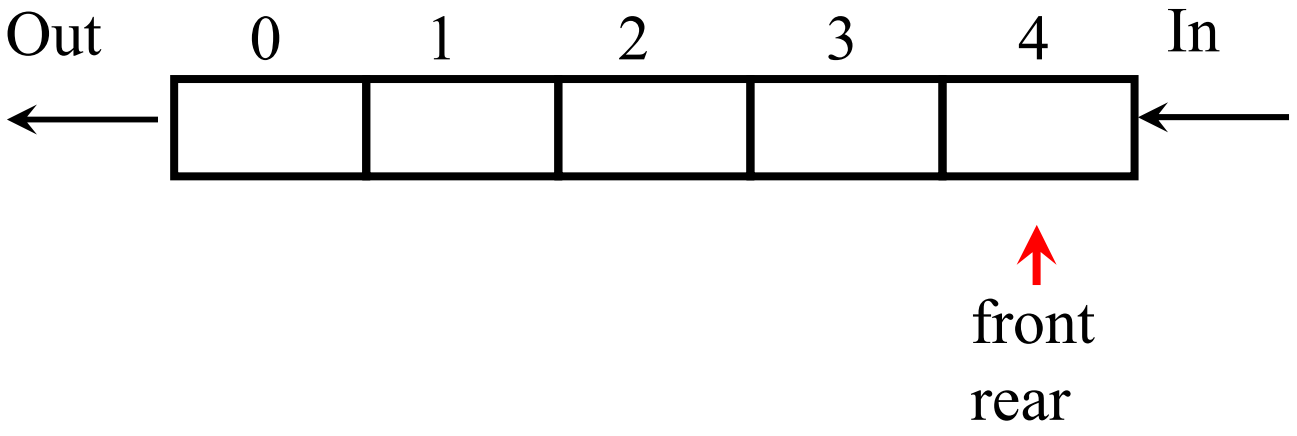
- Storage structure definition

```
struct  QUEUE {
        ElemType  data [ MaxSize ];
        int   front;
        int   rear;
};   //the type of the queue
```



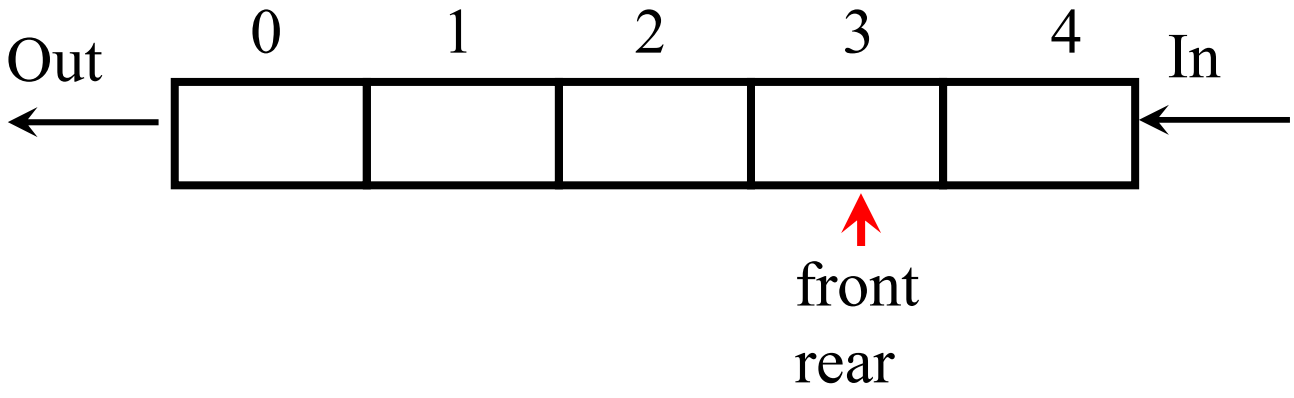
Operation implementations

```
void  MakeNull ( QUEUE  &Q)  // queue initialization
{
      Q.front = MaxSize-1;
      Q.rear   = MaxSize-1;
}
```



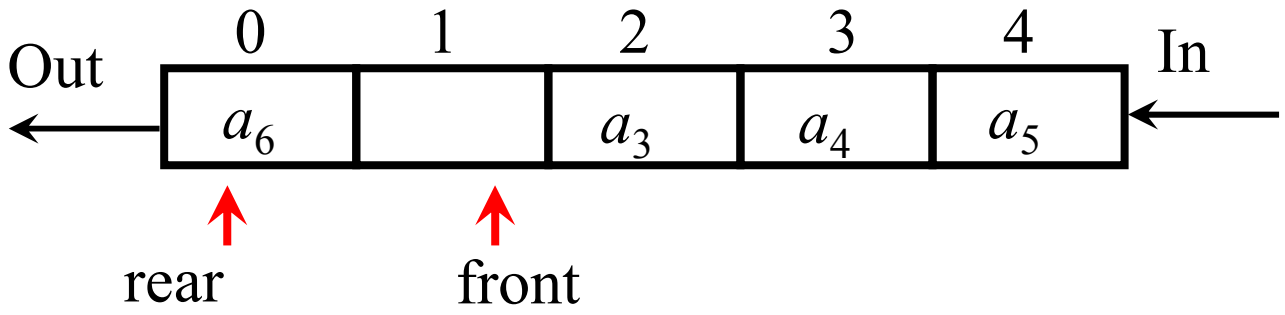
Operation implementations

```
bool Empty( QUEUE  Q )
    {    if ( Q.rear ==  Q.front  )
                return  TRUE ;
           else
                return  FALSE ;
    }
```



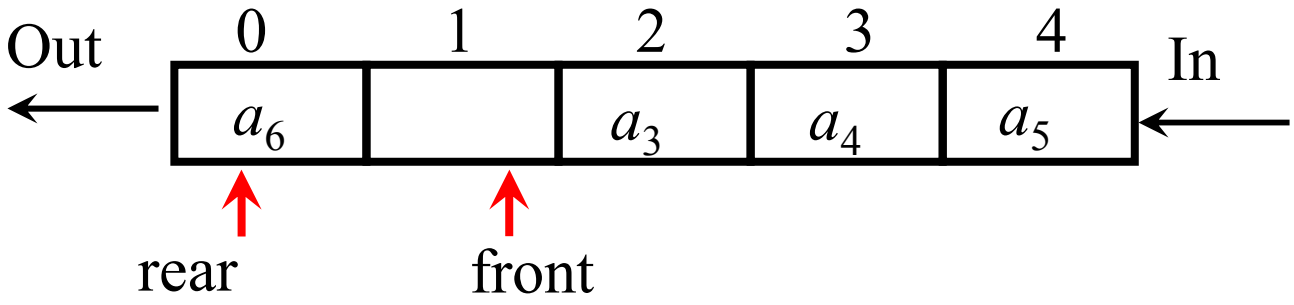
Operation implementations

```
ElemType  Front( QUEUE  Q )
    {   if  ( Empty( Q ) )   return  NULL;
        else {return  (Q.data[Q.front+1)%MaxSize ] ;
         }
    }
```



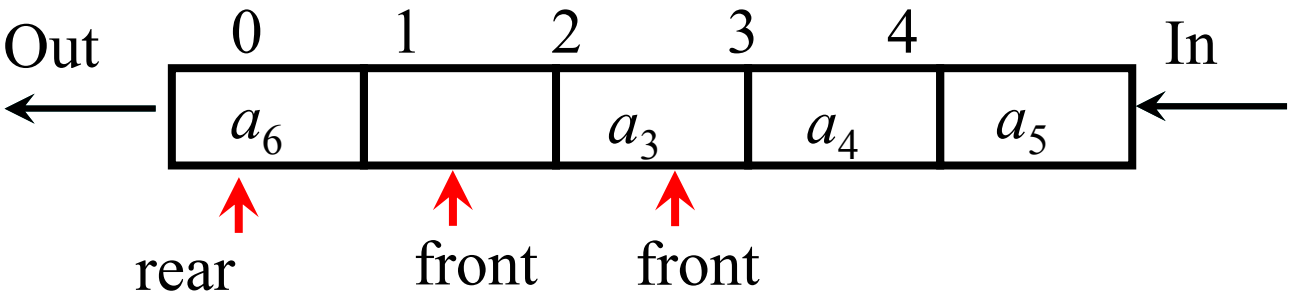
Operation implementations

```
void  EnQueue ( ElemType x, QUEUE &Q )
{      if ( (Q.rear+1)%MaxSize ==Q.front )
                  cout<< "Queue is full !" ;
        else{      Q.rear=(Q.rear+1)%MaxSize ;
                   Q.data[ Q.rear ] = x ;
 }
```



Operation implementations

```
void  DeQueue ( QUEUE  Q ) ;
 {    if ( Empty ( Q ) )
            cout<< "Queue is Empty!"  <<endl;
       else
            Q.front = (Q.front+1)%MaxSize ;
 }
```



Comparison of Array-Based and Linked Queues

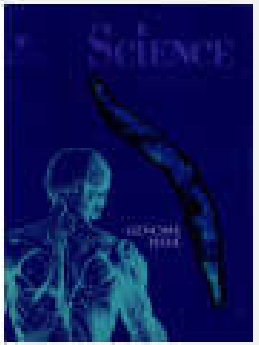
- All functions for both the array-based and linked queue implementations require constant time.
- The space comparison issues are the same as for the equivalent stack implementations
- FIFO is the key to applying queue.
  - Josephus problem.
  - Waiting in queue

String

String

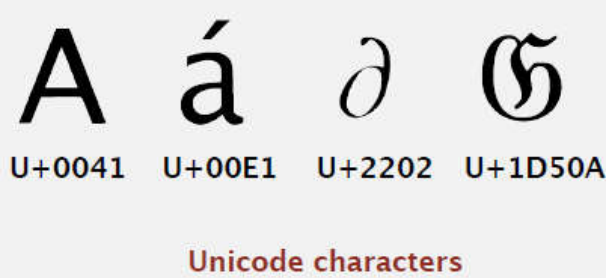
- What is a string?
  - Sequence of characters.
- Important fundamental abstraction.
  - Genomic sequences.
  - Information processing.
  - Communication systems (e.g., email).
  - Programming systems (e.g., C programs).
  - …

" The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology. " — *M. V. Olson*



String

- Notation of a string?
  - Empty string: " "
  - None empty string: *S*=" *s*1 *s*2 …… *sn* "
- The set of characters
  - ASII, Extended ASII
    - 8-bit integer can represent at most 256 characters.
  - Unicode
    - 16-bit Unicode, 8-bit Unicode



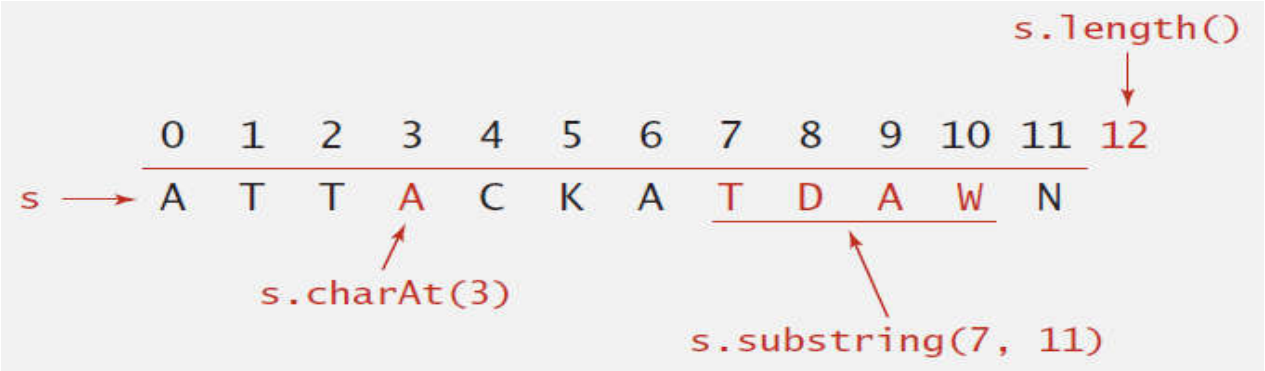
Unicode characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

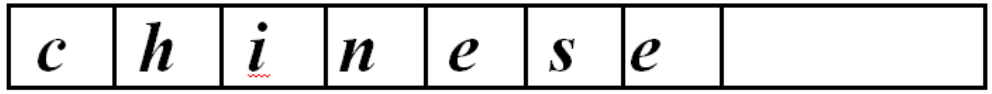
String

- ADT
  - string MakeNull( ) ;
  - bool IsNull (S) ;
  - void In(S, a) ;
  - int Len(S) ; // Number of characters
  - void Concat(S1, S2) ; // Append one string to end of another
  - string Substr(S, m, n) ; // Get a contiguous subsequence of characters
  - bool Comp(S, S1) ;

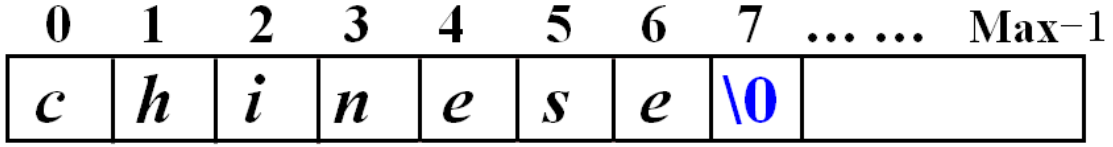


Storage structure of a string

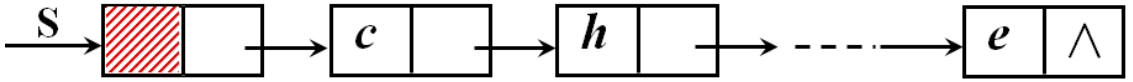
- Array based implementation



- How to know the end of a string?
  - By a special character:

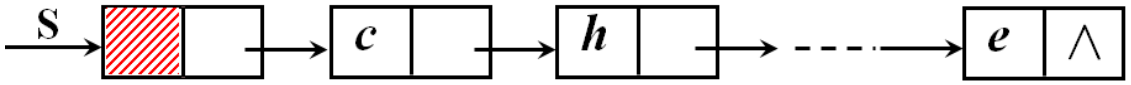


- Linked-list based implementation

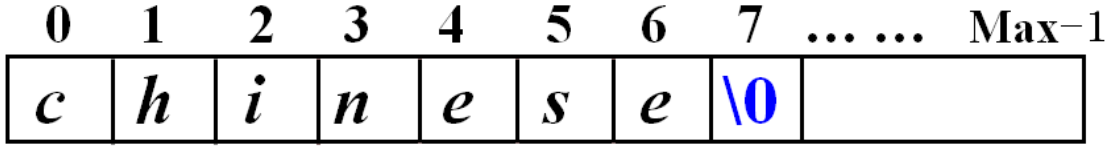


Storage structure of a string

- Linked-list based implementation



- How to know the end of a string?
  - By a special character:



String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Pattern: happy

Text: It is never too late to have a happy childhood.

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, **realign pattern one position to the right** and repeat Step 2

Example of the string match

Text: S="ababcabcacbab", pattern: P="abcac"

```
Pass 1      text    ababcabcacbab     i=3
          pattern   abc               j=3  failed
Pass 2      text    ababcabcacbab     i=2
          pattern    abc              j=1  failed
Pass 3      text    ababcabcacbab     i=7
          pattern     abcac           j=5  failed
Pass 4      text    ababcabcacbab     i=4
          pattern       abc           j=1  failed
Pass 5      text    ababcabcacbab     i=5
          pattern         abc         j=1  failed
Pass 6      text    ababcabcacbab     i=6
          pattern         abcac       j=6  succeed
```

Pay attention to the position of both S and P.

Algorithm of the string match

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Time complexity

- Best case
  - O(n+m)

$$\sum_{i=1}^{n-m+1} p_i(i-1+m) = \frac{1}{n-m+1}\sum_{i=1}^{n-m+1}(i-1+m) = \frac{1}{2}(m+n)$$

- Worst case
  - O(n*m)

$$\sum_{i=1}^{n-m+1} p_i(i\times m) = \frac{m}{n-m+1}\sum_{i=1}^{n-m+1} i = \frac{1}{2}m(n-m+2)$$

  - The elements of ‘S’ with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations

The Knuth-Morris-Pratt Algorithm

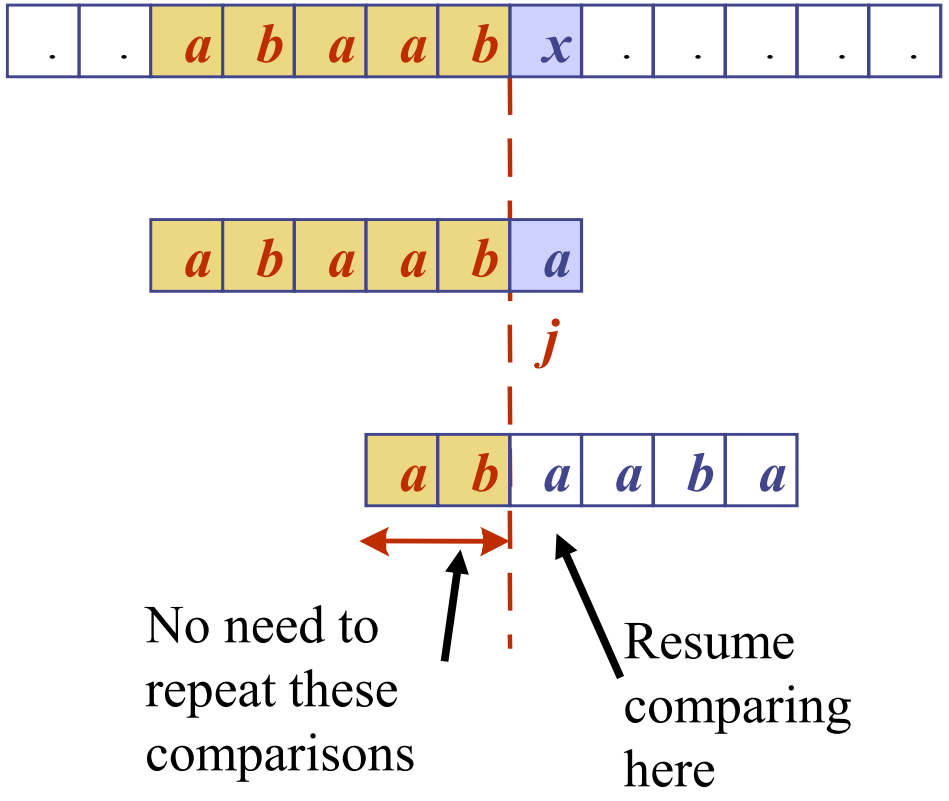
- Knuth, Morris and Pratt proposed a linear time O(n) algorithm for the string matching problem.

- The new algorithm is based on the observation that by starting the next pattern comparison at its beginning each time, we may be discarding valuable information gathered during previous comparisons.

- Naturally, we wish this shift to be as large as possible, and backtracking on the string ‘S’ never occurs

Components of KMP algorithm

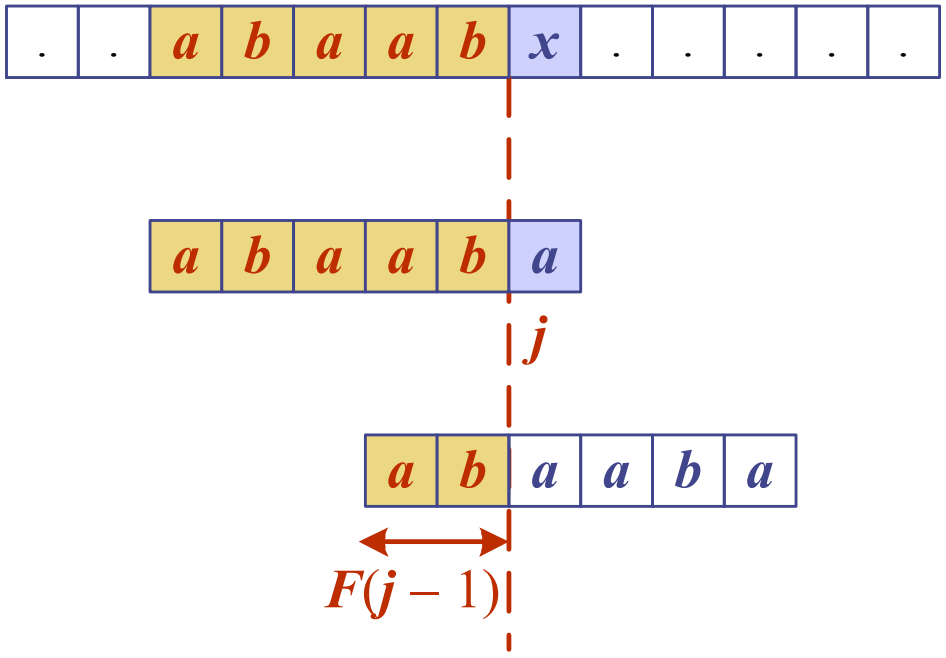
- KMP's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer**: the largest prefix of *P*[0..*j*] that is a suffix of *P*[1..*j*]**



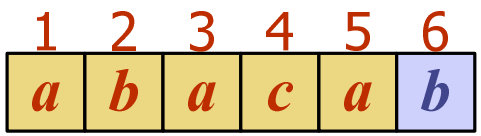
KMP Failure Function

- KMP's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- KMP algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

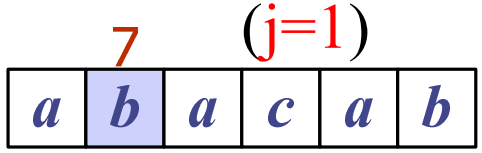
j	0	1	2	3	4	5
$P[j]$	*a*	*b*	*a*	*a*	*b*	*a*
$F(j)$	0	0	1	1	2	3



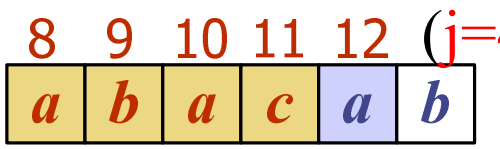
Example



(j=5)



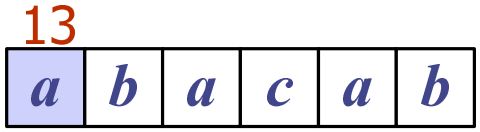
(j=1)

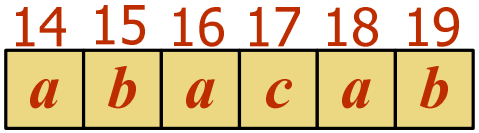


(j=4)

j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Next start character (index=*F(j-1)*)





The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time :

  $O(m+n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
                // j is the position for
                // next comparison
            else // T[0] != P[0], then T[1]
                i ← i + 1
    return −1 { no match }
```

Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP's algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

Algorithm *failureFunction*(P)

```
F[0] ← 0
i ← 1
j ← 0
while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j − 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

Example: compute *F* for the pattern 'p' below:

p	a	b	a	b	a	c	a

Initially: m = length[p] = 7

F[1] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0					

Step 1: q = 2

F [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1				

Step 2: q = 3

F[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2			

Step 3: q = 4

F [4] = 2

The size of the largest prefix of ***P***[0..***j***] that is also a suffix of ***P***[1..***j***]

Step 4: q = 5

F [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5: q = 6

F [6] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Step 6: q = 7

F [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

After iterating 6 times, the prefix function computation is complete:

→

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

The size of the largest prefix of ***P***[0..***j***] that is also a suffix of ***P***[1..***j***]

The KMP Matcher

Input: pattern ‘p’, string ‘S’ and prefix function ‘Π
Output: finds a match of p in S.

KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                          //number of characters matched
5 for i ← 1 to n                                 //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7              do  q ← Π[q]                      //next character does not match
8         if p[q+1] = S[i]
9           then q ← q + 1                       //next character matches
10        if q = m                               //is all of p matched?
11          then print “Pattern occurs with shift” i – m
12               q ← Π[ q]                       // look for the next match
```

Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

For ‘p’ the failure function, F was computed previously and is as follows:

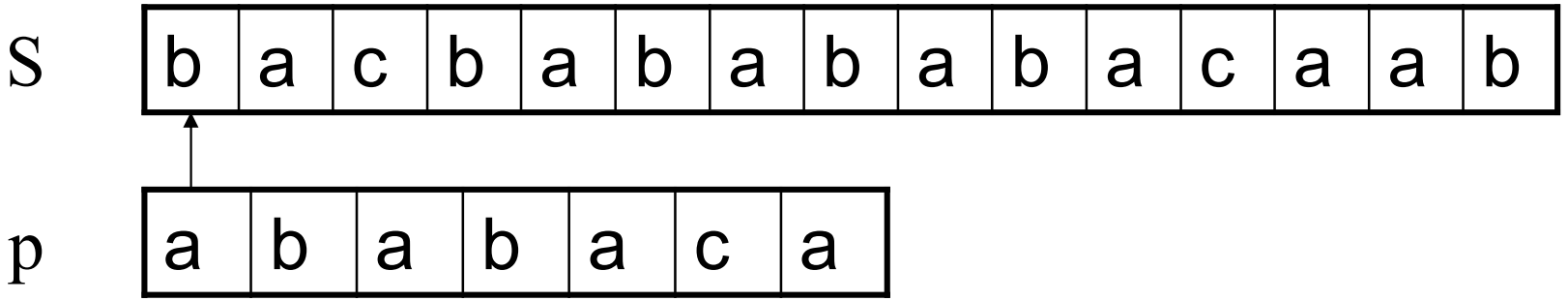
j	0	1	2	3	4	5	6
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

Initially: n = size of S = 15;

m = size of p= 7

Step 1: i = 1, q = 0

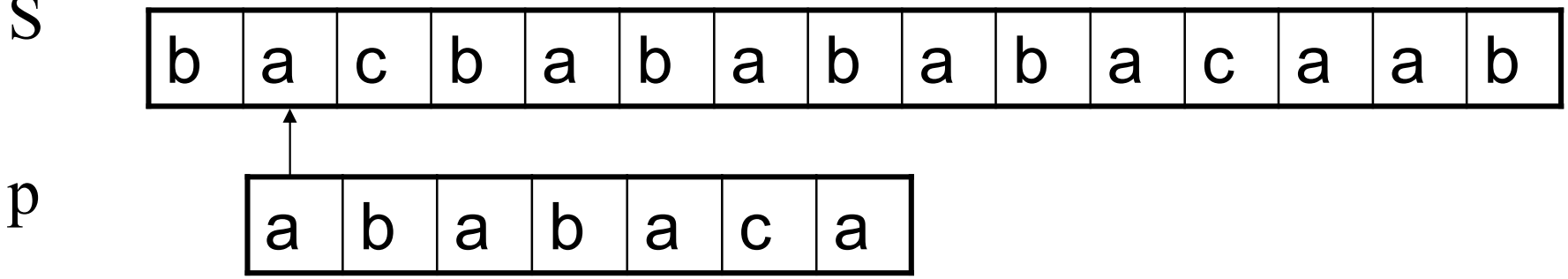
comparing p[1] with S[1]



p[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

Step 2: i = 2, q = 0

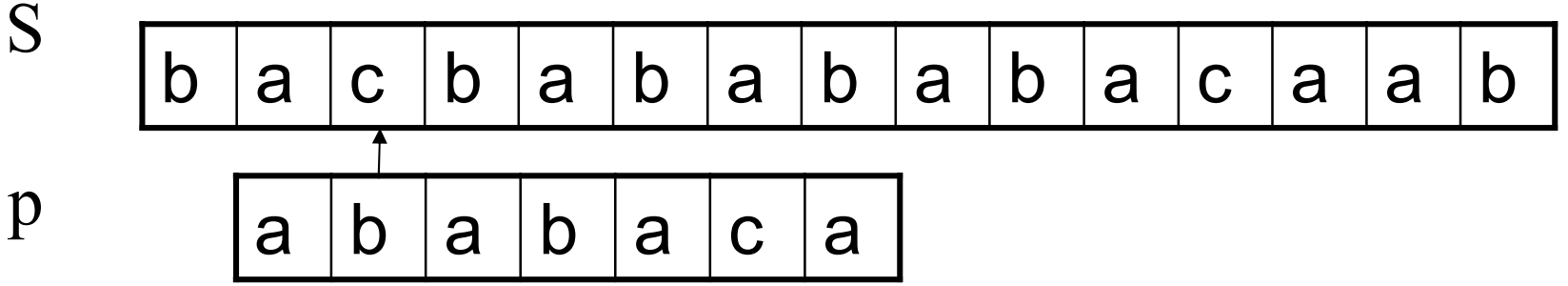
comparing P[1] with S[2]



p[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

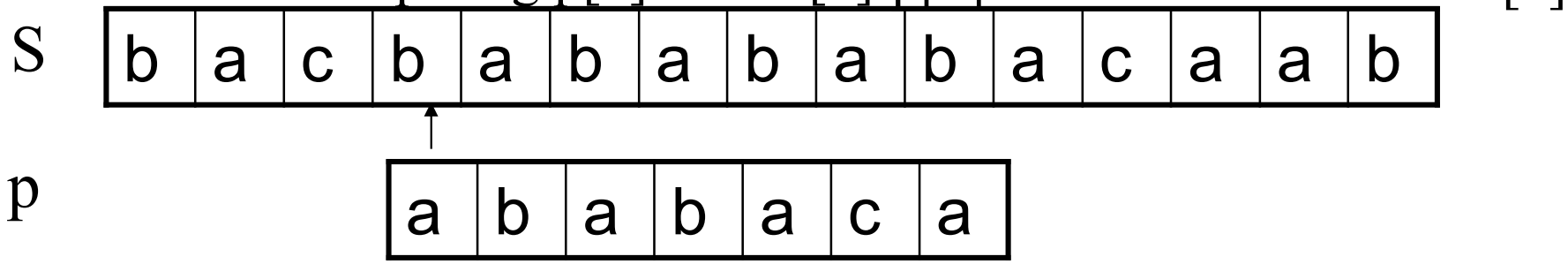
Comparing p[2] with S[3] p[2] does not match with S[3]



Backtracking on p, comparing p[1] and S[3]

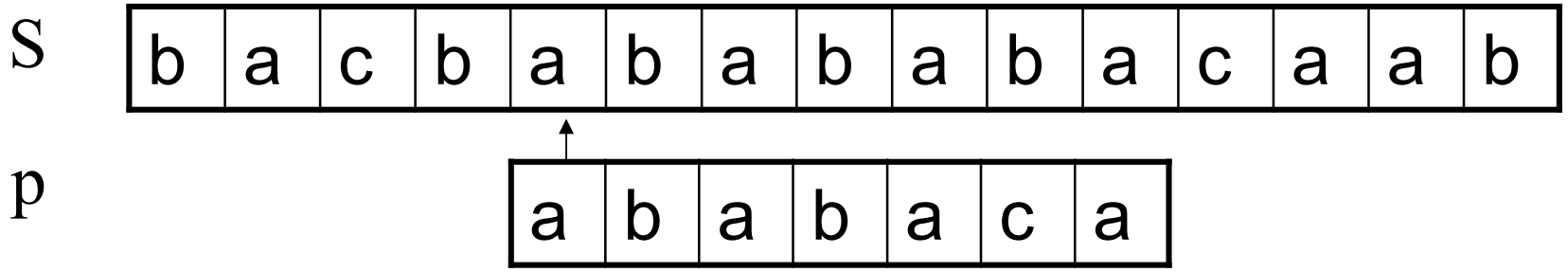
Step 4: i = 4, q = 0

comparing p[1] with S[4] p[1] does not match with S[4]



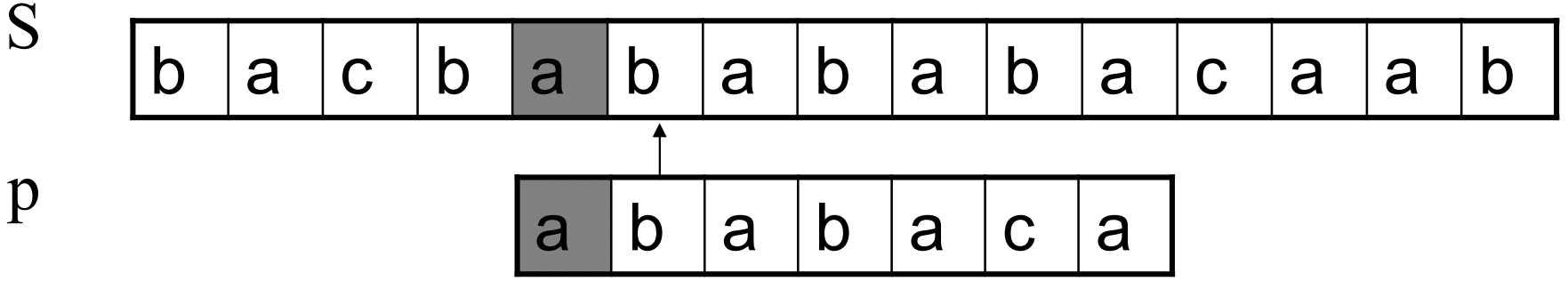
Step 5: i = 5, q = 0

comparing p[1] with S[5] p[1] matches with S[5]



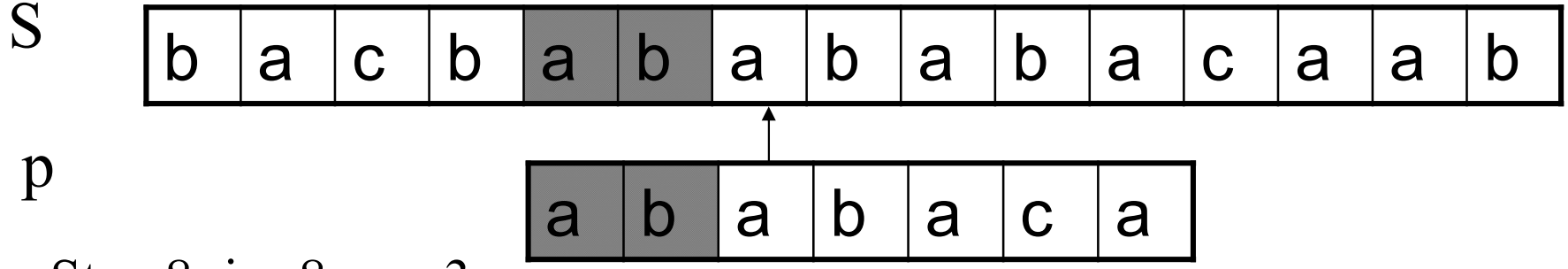
Step 6: i = 6, q = 1

Comparing p[2] with S[6] p[2] matches with S[6]



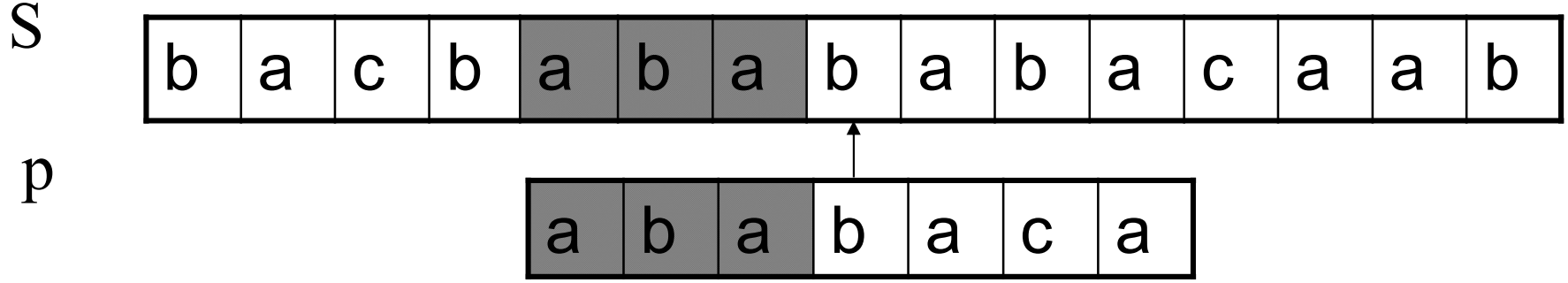
Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]



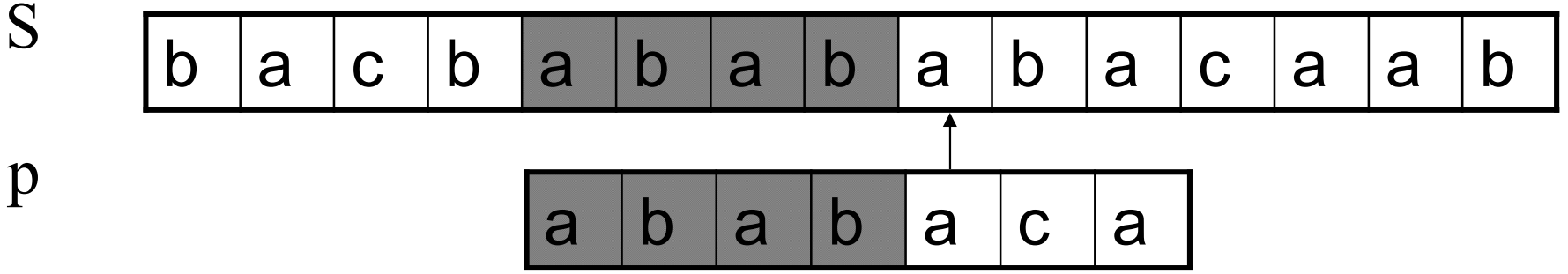
Step 8: i = 8, q = 3

Comparing p[4] with S[8] p[4] matches with S[8]



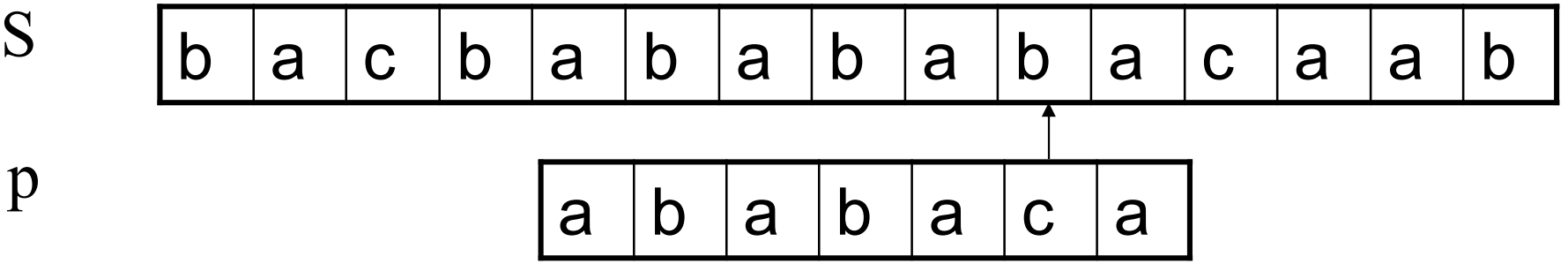
Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]



Step 10: i = 10, q = 5

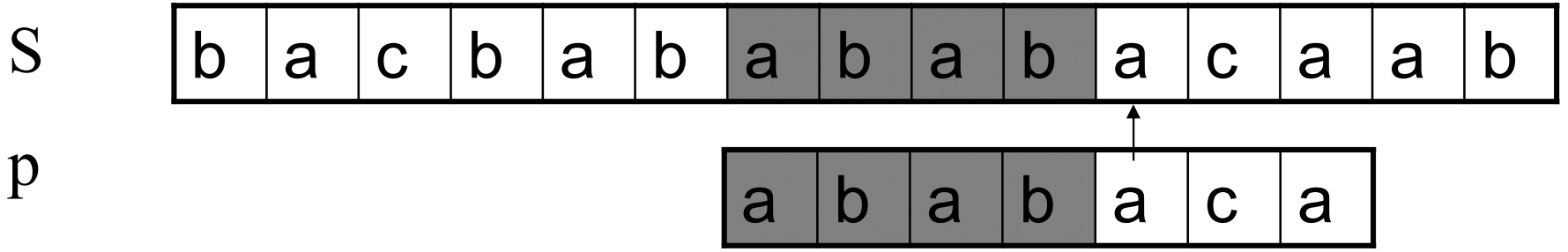
Comparing p[6] with S[10] p[6] doesn't match with S[10



Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

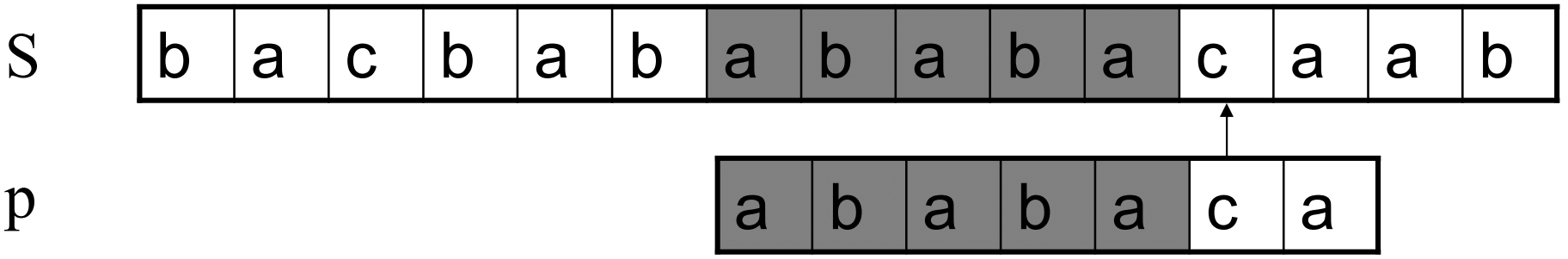
Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]



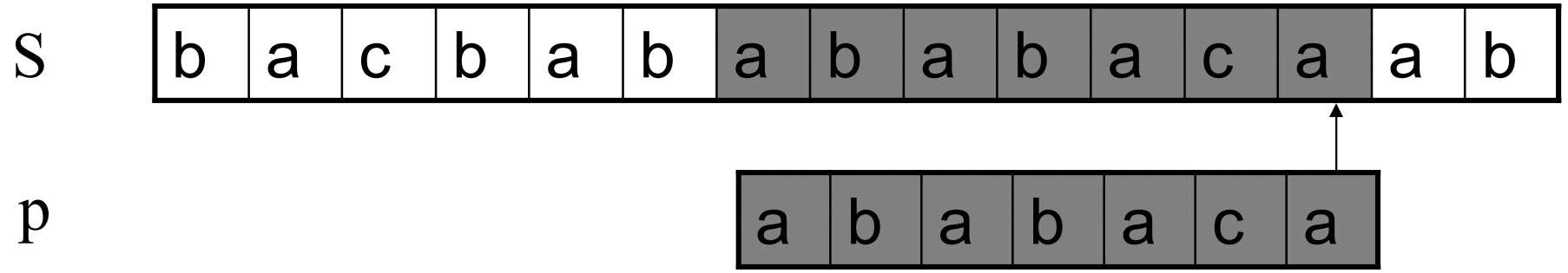
Step 12: i = 12, q = 5

Comparing p[6] with S[12] p[6] matches with S[12]



Step 13: i = 13, q = 6

Comparing p[7] with S[13] p[7] matches with S[13]



Pattern ‘p’ has been found to completely occur in string ‘S’.

The total number of shifts that took place for the match to be found are:

i – m = 13 – 7 = 6 shifts.

Multi-dimensional Array

Multi-dimensional Array

Definition: array type

- an **array type** is a data type that is meant to describe a collection of *elements* (values or variables) of identical types.
- each selected by one or more indices (identifying keys) that can be computed at run time by the program.

Definition: dimension

- The number of indices needed to specify an element is called the *dimension, dimensionality,* or *rank* of the array type.

Multidimensional array

- A multidimensional array is treated as an array of arrays.
  - Let a be a k-dimensional array, the elements of a can be accessed using the following syntax:

    a [i_1] [i_2]…[i_k]

Multi-dimensional Array

Property:

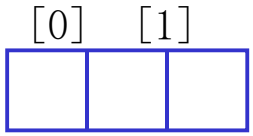
- arrays allow random access

Usage:

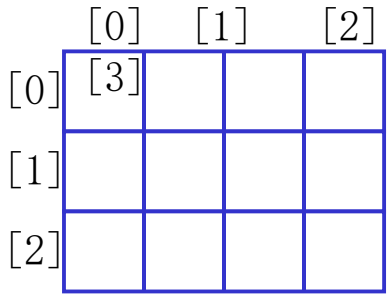
- use one-dimensional arrays to model linear collections of elements.
- use a two-dimensional array to represent a matrix or a table

Multi-dimensional Array

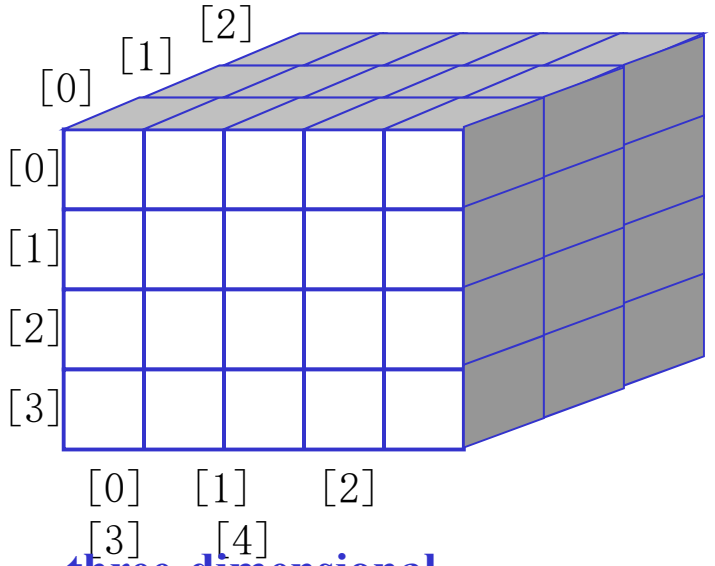
One-dimensional arrays are linear containers.



Multi-dimensional Arrays



two-dimensional



three-dimensional

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

Multi-dimensional Array

Basic Operations

- Create( )// initialization
  - Create an empty array；
  - int A[ ][ ]
- Retrieve (array, index) // get the value of an element
  - By given indices, return the value of specified element
  - A[i][j]
- Store(array, index, value) //store or update/modify the value
  - By given indices, store or update the value of indicated element
  - A[i][j]=8
- No need for *insert* and *remove* operations

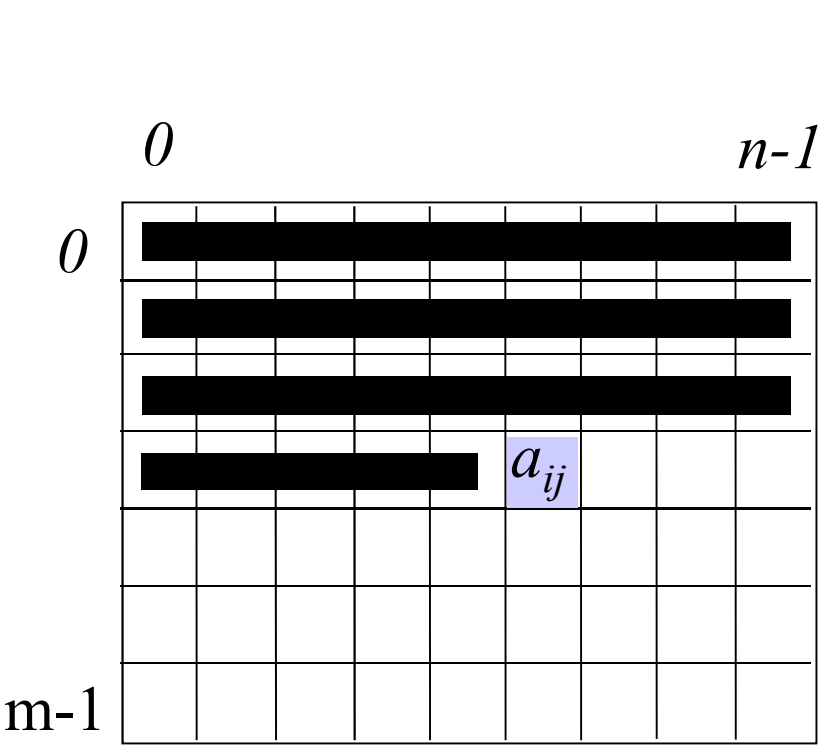
Multi-dimensional Array

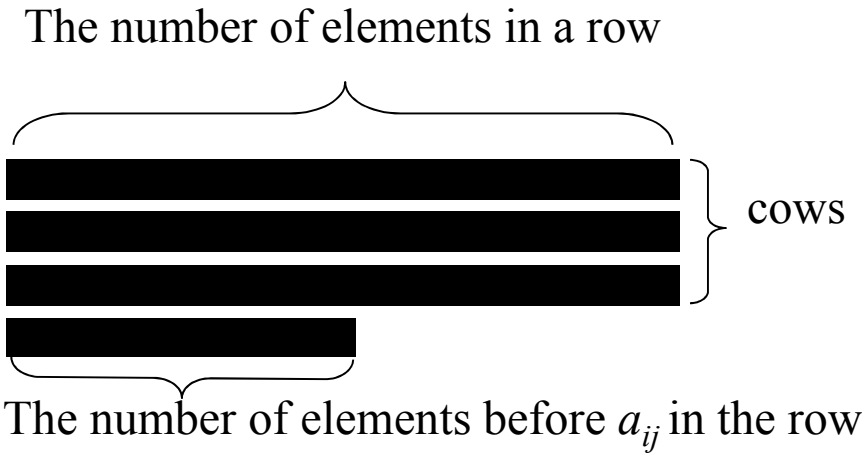
Storage structure

- Computer memory is linear –
- In accessing a multidimensional array you are really accessing a one-dimensional array in memory.
- You can simulate multidimensional arrays by packing them into a one-dimensional array.
- The two manners of packing:
  - row-major order, where the array is filled one row at a time, and
  - column-major order, where columns are placed into the array.

Multi-dimensional Array

– How to locate an element: 2-d array





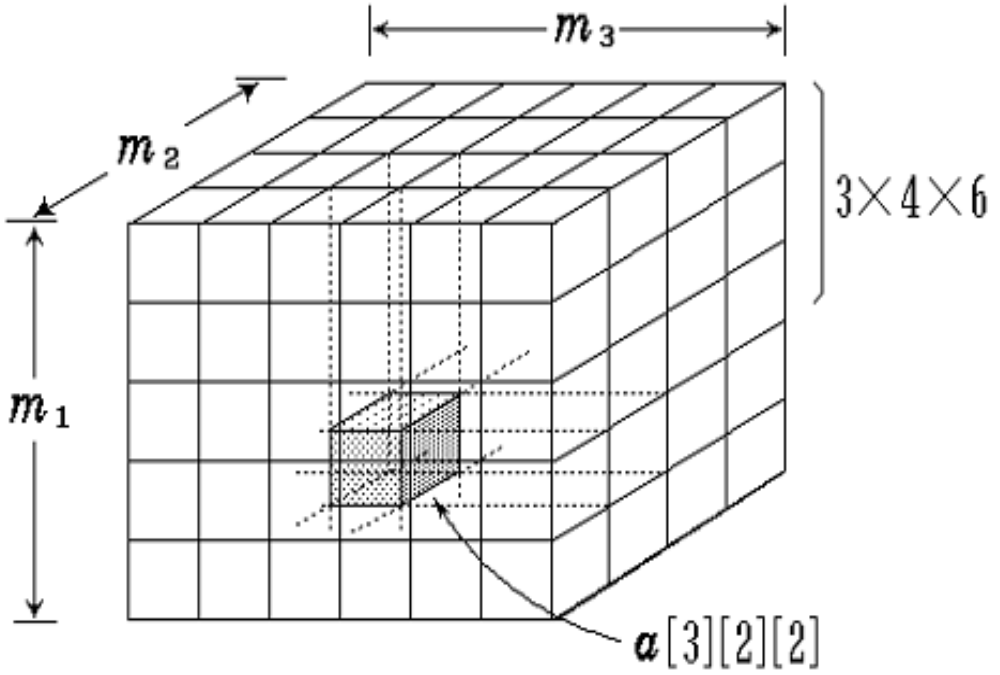
The number of elements before a_{ij} $=i \times n + j$

$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{00}) + (i \times n + j) \times c$$

Multi-dimensional Array

– Row major order----multi-dimenssion

n（$n>2$）dimension use the two ways to store.

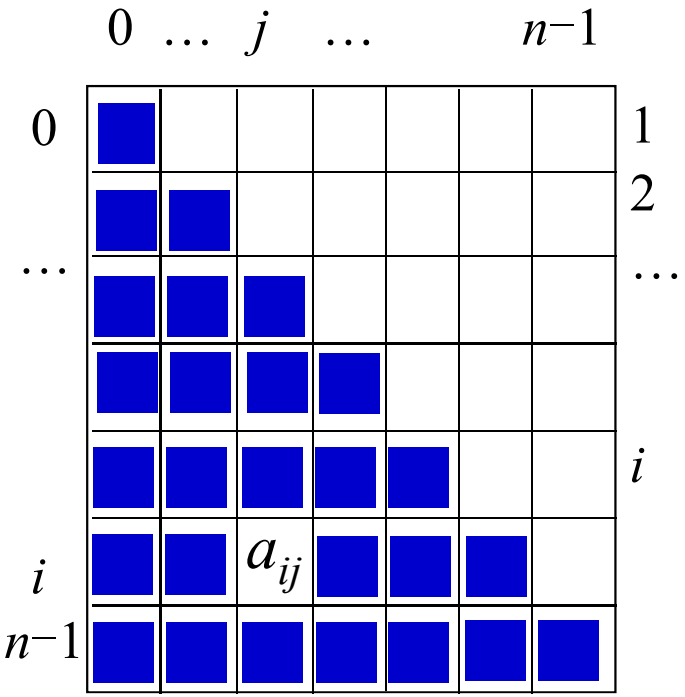


$$\mathrm{Loc}(a_{ijk}) = \mathrm{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

Special cases

- Symmetric matrix
  - $a_{ij}=a_{ji}$
  - Store one triangle part.

$$A=\begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$



Index:

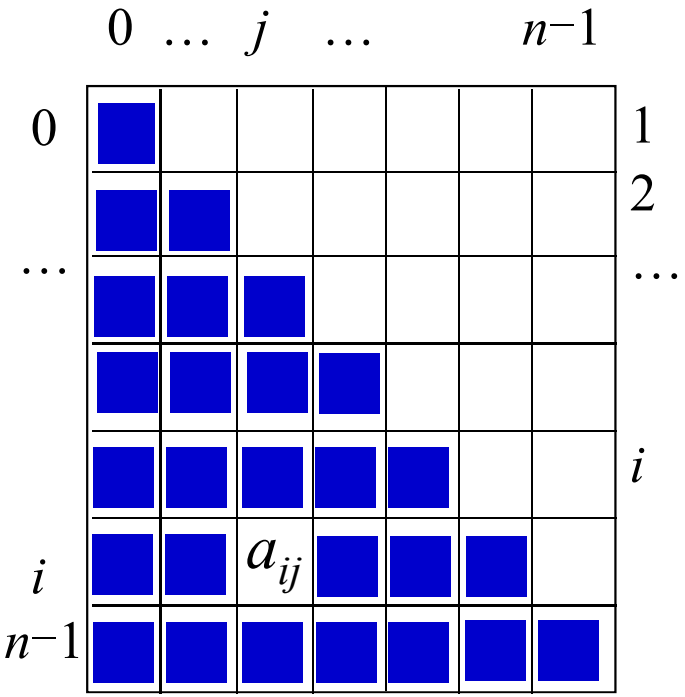
$k=i\times(i+1)/2+j$ $(i\geq j)$

$k=j\times(j+1)/2+i$ $(i<j)$

Special cases

- Triangular matrix
  - keep only the triangle part
  - keep only one constant

$$A=\begin{pmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$



Index：

$k= i\times(i+1)/2+ j \qquad (i\geq j)$

$k= n\times(n+1)/2+ i \qquad (i< j)$

Sparse matrix

- Sparse matrix----List of 3 tuple
  - Row-major order

$$A=\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	item
0	0	0	15
1	0	3	22
2	0	5	−15
3	1	1	11
4	1	2	3
5	2	3	6
6	4	0	91
M-1	Null		

7 No. the non-zero

5 Number of rows

6 Number of columns

Sparse matrix

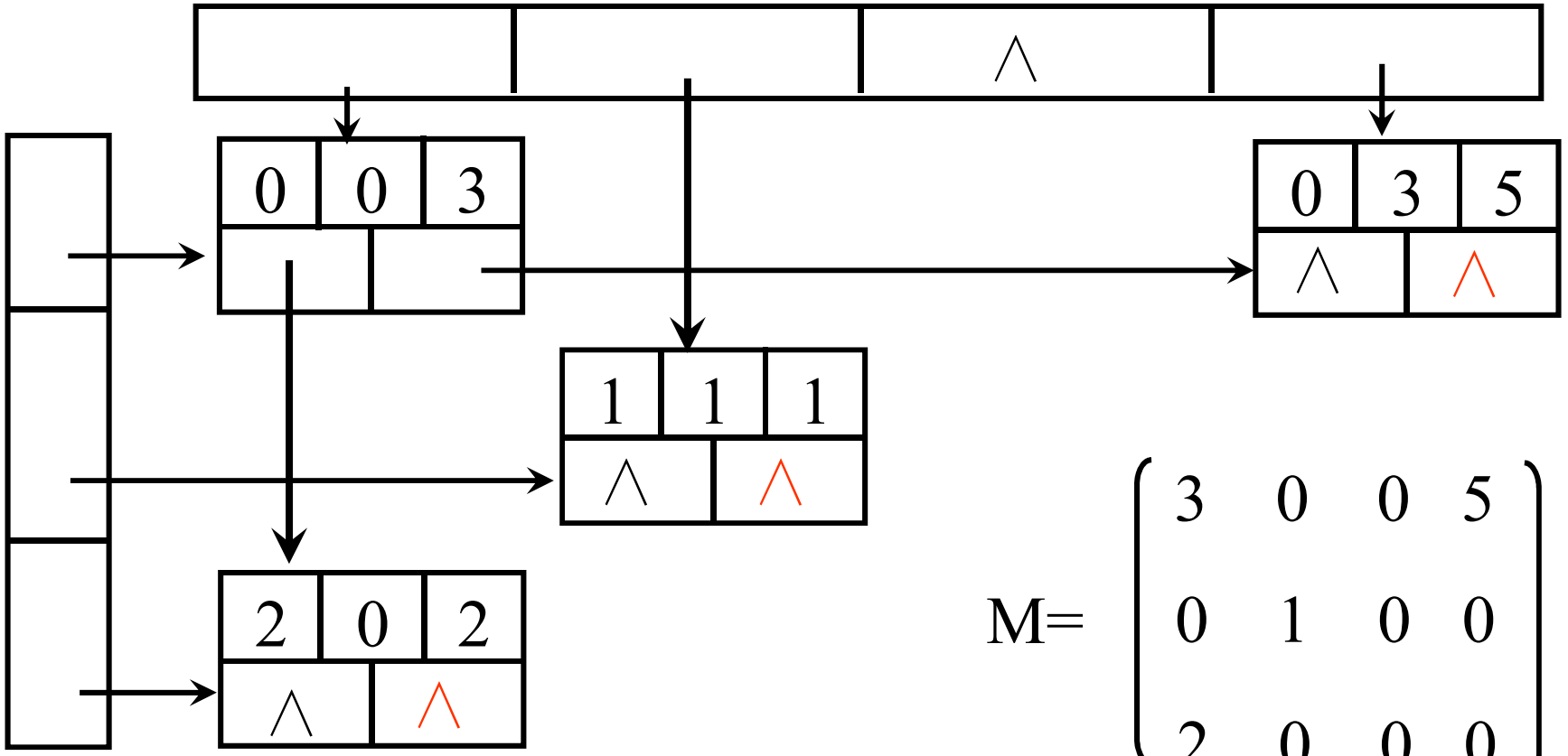
- Sparse matrix- ----cross linked list
  - The node of the cross linked list：

row	col	item
down		right

row： No. of the row
col： No. of the column
item： value of the element. (non-zero)
right： pointer field, pointing to the next node in the same row
down： pointer field, pointing to the next node in the same column

Sparse matrix

- Sparse matrix- ----cross linked list



$$M=\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Summary

- Array types are often implemented by array data structures,
- but sometimes by other means, such as
  - linked lists
  - hash tables, or
  - search trees.

Generalized list

- A generalized list, LS, is a finite sequence of n≥0 elements, denoted as:

$$LS=(a_1,\ a_2,\ \ \ldots\ldots,\ \ a_n)$$

  where a_i is either an atom or a list.
- The element a_i , 0≤i ≤ n-1, that are not atoms are said to be the sublists of LS.
  - *A is the name of the list.*
  - *n is the length of LS.*
  - α_0 *is the head of LS.*
  - $(\alpha_1, \ldots, \alpha_{n-1})$ *is the tail of LS.*
- Lists may be shared by other lists.
- Lists may be recursive.

Generalized list

- Examples:
  - A = (a, (b, a, b), (), c, (((2))));
  - B = ();
  - C = (e);
  - D = (A, B, C);
  - E = (a, E);

Generalized list

- Basic operations

①Cal（ L ）//return the first element

②Cdr（ L ）// return all elements except for the fist

③Append（ L，M ）//return L + M

④Equal（ L，M ）// boolean

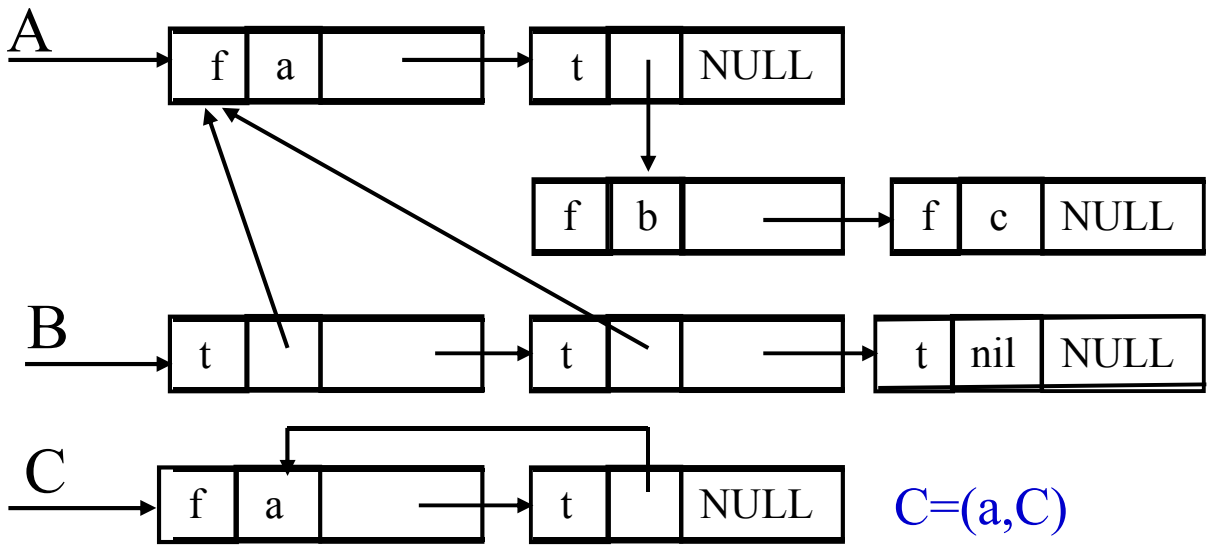
⑤Length（ L ）//return the length of the L

Generalized list

- Storage structure

A=(a,(b,c))

B=(A,A,( ))




```
struct  listnode {
      listnode  *link ;
      boolean  tag ;
      union  {
            char  data ;
            listnode  *dlink ;
      } ;
 } ;
typedef   listnode  *listpointer;
```

f: atom, t: list

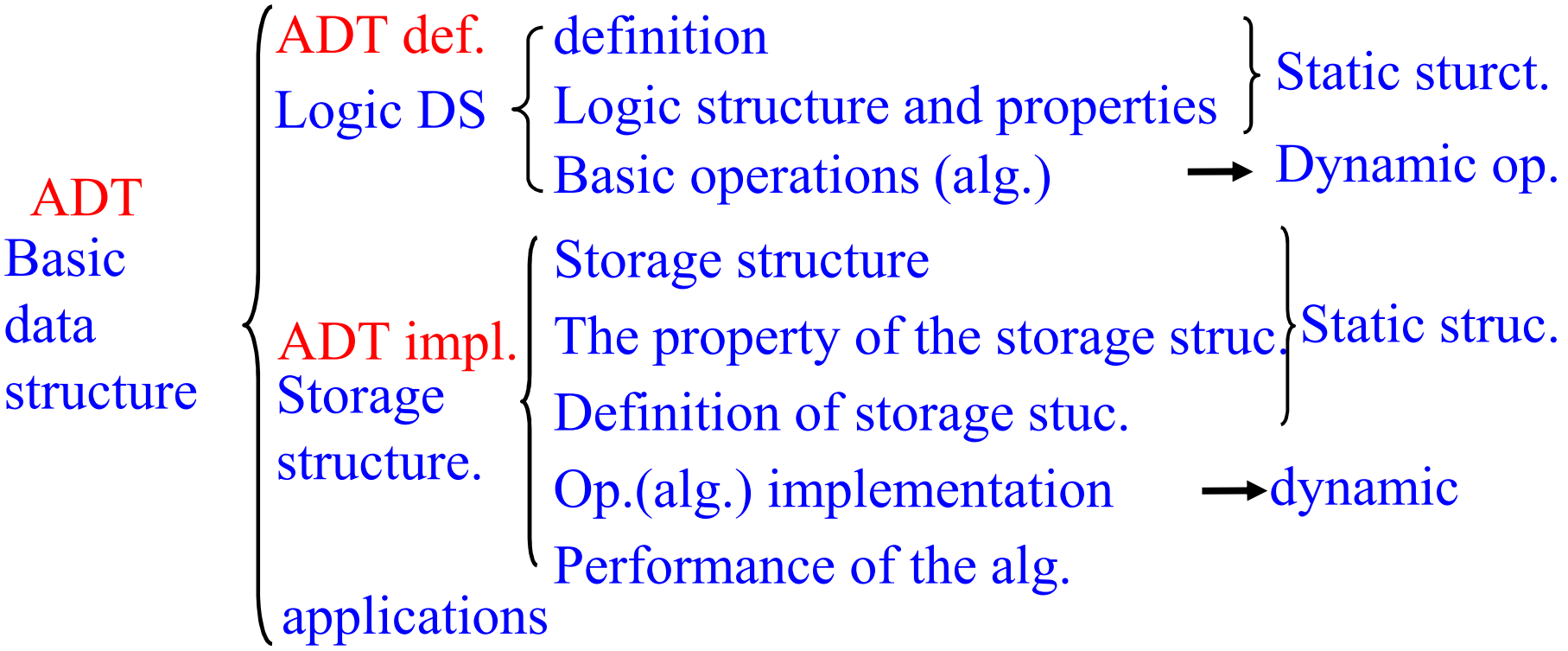
Generalized list

- The implementation of the GL

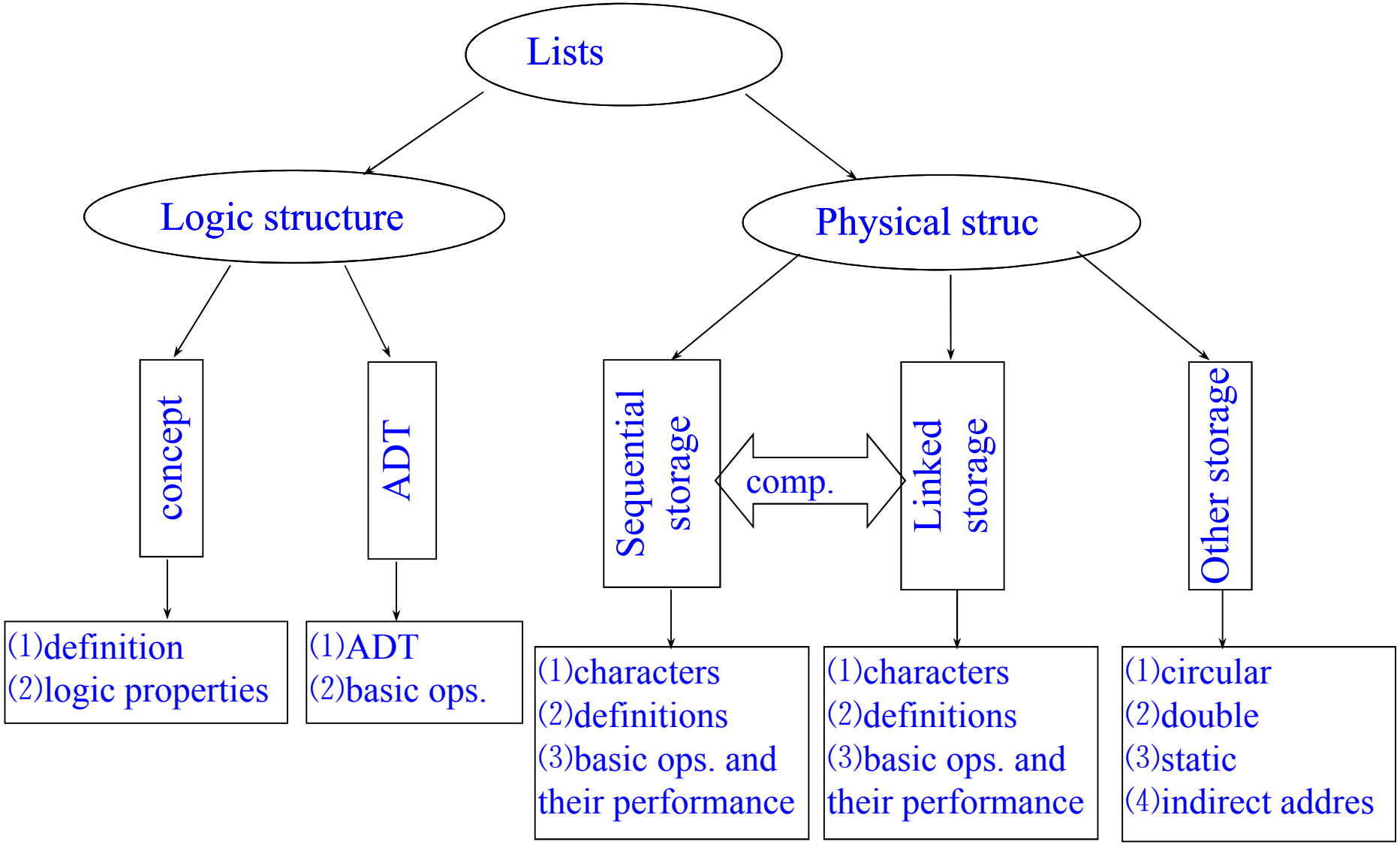
```
bool  Equal( listpointer  S, listpointer T )
{    boolean  x, y ;
     y = FALSE ;
     if  ( ( S == NULL ) && ( T == NULL ) )
          y = TRUE ;
     else if  ( ( S != NULL ) && ( T != NULL ) )
          if  ( S→tag == T→tag )
              {  if ( S→tag == FALSE)
                    { if ( S→element.data == T→element.data )
                         x = TRUE ;
                      else
                         x = FALSE ;
                 else
                      x = Equal( S→element.data,T→element.data );
                 if ( x== TRUE )
                      y = Equal( S→link, T→link ) ;
               }
        return y ;
}   //S and T are not recursive lists
```

Summary of the Chapter

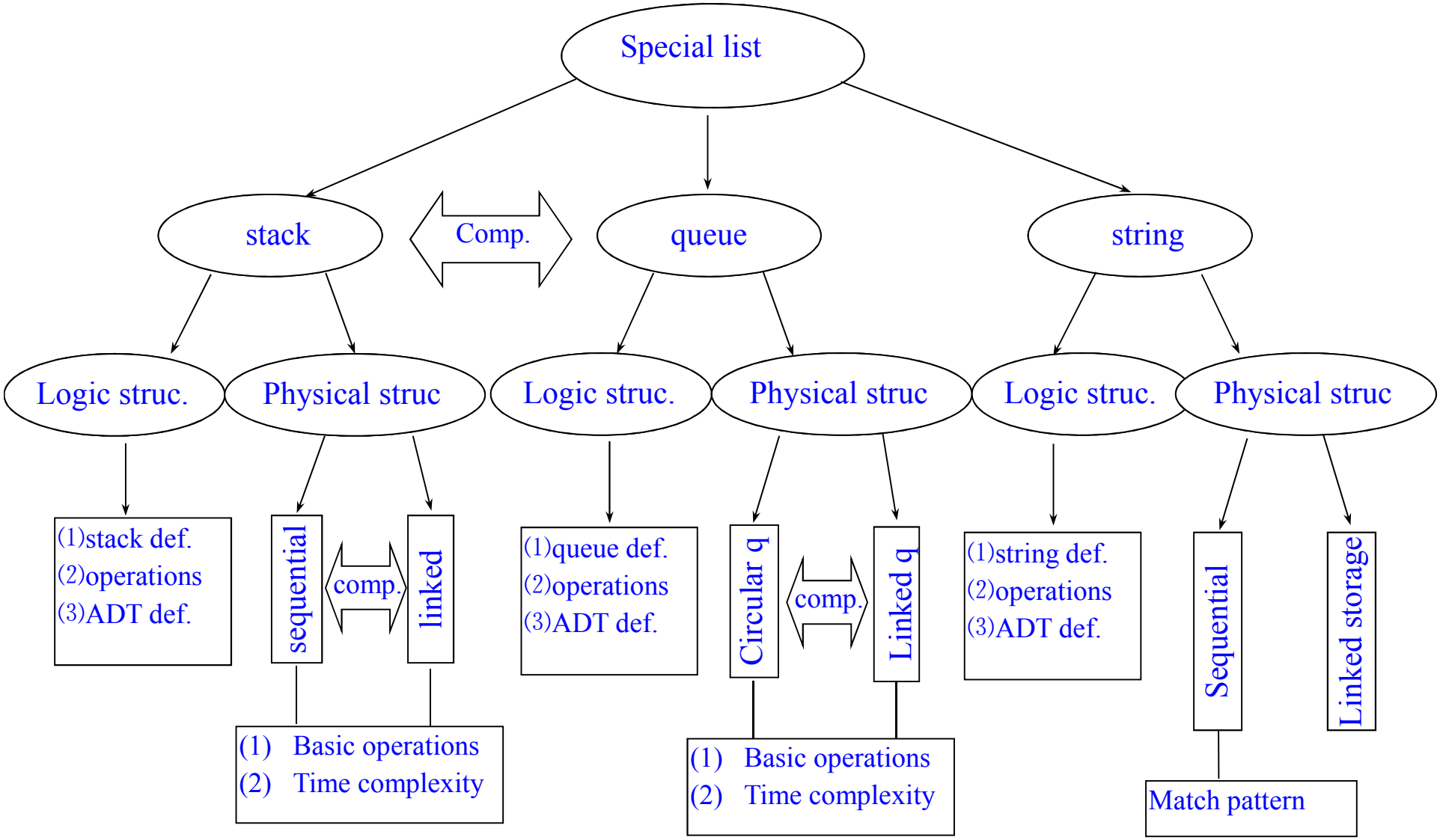
- Data structure：
  - List, stack, queue, string, (multi-dimension) array, generalized list
- Systematic view:



Summary of the Chapter



Summary of the Chapter



Summary of the Chapter