COMP2054
Spring Semester
Algorithms Data structures & Efficiency (**ADE**)

Introduction

Andrew Parkes (Room C78)

andrew.parkes@nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

“ADE”: Algorithms Data structures & Efficiency

- Covers
  - **A**lgorithms,
  - **D**ata Structures, and
  - **E**fficiency
- **Algorithm:** step-by-step procedure for solving a problem in a **finite** amount of time.
  - No “MAGIC steps” allowed! (No “Zeno machines”)
- **Data structures** : lists, binary trees, heaps, Hashmaps, graphs, etc.
- **Efficiency**: We will start with methods to analyse, classify and describe the efficiency of algorithms
  - Needed in order to be able to select “best algorithms”

Rough Comments

- On average, the ADE module is less formal than the COMP2065-IFR. However:
  - Some portions (definitions and proofs) are formal – and should be treated as such
    - (Optional: think how to convert them to Lean.)
  - Some portions – interpretations and "every day usage" are less formal
    - E.g. targeted to what would a software development team want to know

ADE Assessments

- 25% "standard coursework"
  - THREE in-class **CLOSED BOOK IN-PERSON** exam-conditions tests of 30 minutes during the Monday lab (either written or on Moodle).
  - Final mark based on "best 2 of 3" !
  - Dates: To Be Confirmed (TBC). Provisionally:

    C/W 1: Mon 26 Feb
    C/W 2: Mon 18 March or Mon 25 March
    C/W 3: Mon 13 May

    Put these dates in your diary! There are no "late submissions" – but you ask for an EC for missed tests, and can miss one anyway as it is best 2 or 3
- 75 % ExamSys/written exam
  - Summer exam session (May/June)
  - **CLOSED BOOK, IN-PERSON** !!
  - Two hours ExamSys; possibly with "on-paper" addition

Tutorials

- ONE 1-hour tutorial per person
  - ADE sessions will just be some weeks, not all – I will notify you on Moodle and email
  - These are essential for understanding the material, and will help with C/Ws, etc.
  - Please use the hour that is assigned to you on your personal timetable

Labs

- ONE 1-hour tutorial per person
- Labs will be sessions for doing some formative exercises and getting help as needed in preparation for the C/Ws.
  - ADE labs will just be some weeks - not all – I will notify you on Moodle and email
- Please use the hour that is assigned to you on your personal timetable

Email Etiquette

- **Only send email from your University account**! (Except in exceptional circumstances).
  - Only this allows staff to know who is really sending. We cannot discuss anything with random external email addresses as they may not be you.
  - Such emails may well just be treated as spam and deleted
- Do not expect an immediate reply – 3 working days is the norm. After that, send a gentle reminder.
- Include the module code in the subject line
  - This is a good practice for all modules. Staff teach multiple modules; hence writing "I need help with your module" will just cause delays.
  - To check which emails need to be answered, I may well search by the module code.
- Include all relevant details. E.g. lecture and slide number, tutorial question, etc.
  - If it is about personal process, or scores, etc, then include your student ID number.
  - Please do not just write "Help, I'm stuck". It does not help us to help you !
  - Often the process of fixing the details leading to answering the question

Study Skills Suggestion

Personal View:

- Often problems arise from going too fast
- Likely to often need to backtrack or start again
  - Backtracking is normal; do not see it as a failure
- Learn to seek chances to identify when need to "fix a bug in your thinking"
  - Allows to make better overall progress
  - Rebuilding 'internal models' is positive!
    - (The main goal of "real learning"?)

"Exercises" & Hints

- Will often put "**Exercise**" in slides
  - Please do them! At least think about them
  - They are not assessed, but are vital
  - They might well be a vital part of an exam question
- Hints:
  - If stuck because of 'algebra' then "generate your own examples" simply insert small numbers. E.g. put a=2, etc.
  - **Strongly encourage to start with the very simplest examples and work 'upwards'**
    - Many problems arise from reluctance to focus on the basics first

What are Algorithms?

- Algorithm: a well-defined step-by-step procedure for solving a problem in a finite amount of time.

For example:

- A recipe for cooking pizza
- A set of instructions for assembling a piece of furniture
- A procedure for finding web pages containing keywords
- A procedure for computing n! given n
- Sorting and searching algorithms
  - E.g. how to sort a pack of cards.
- A deep learning algorithm for deciding whether to grant someone a mortgage

What are Data Structures?

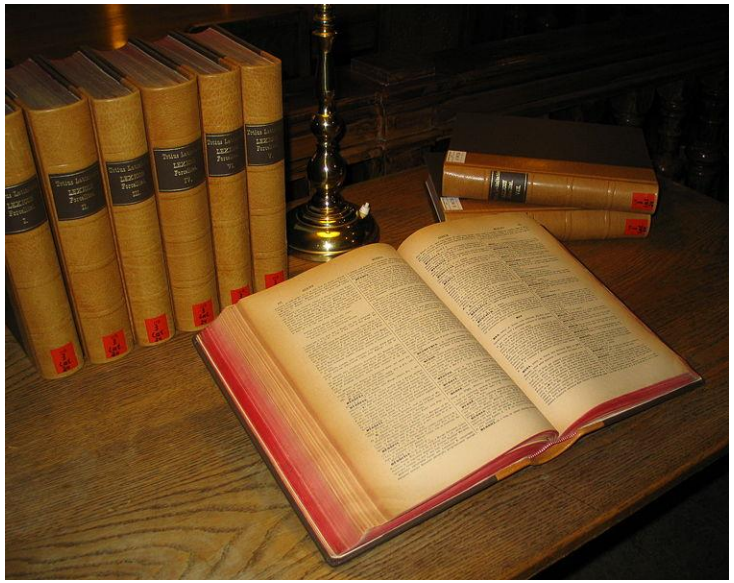
- Ways in which to store data in a well-defined and structured fashion to allow algorithms to:
  - make better use of it
  - maintain it more easily as the data changes

We will need to also define “better” and “more easily”

Data Structures in Everyday Life? …

Data Structure Example: Dictionary

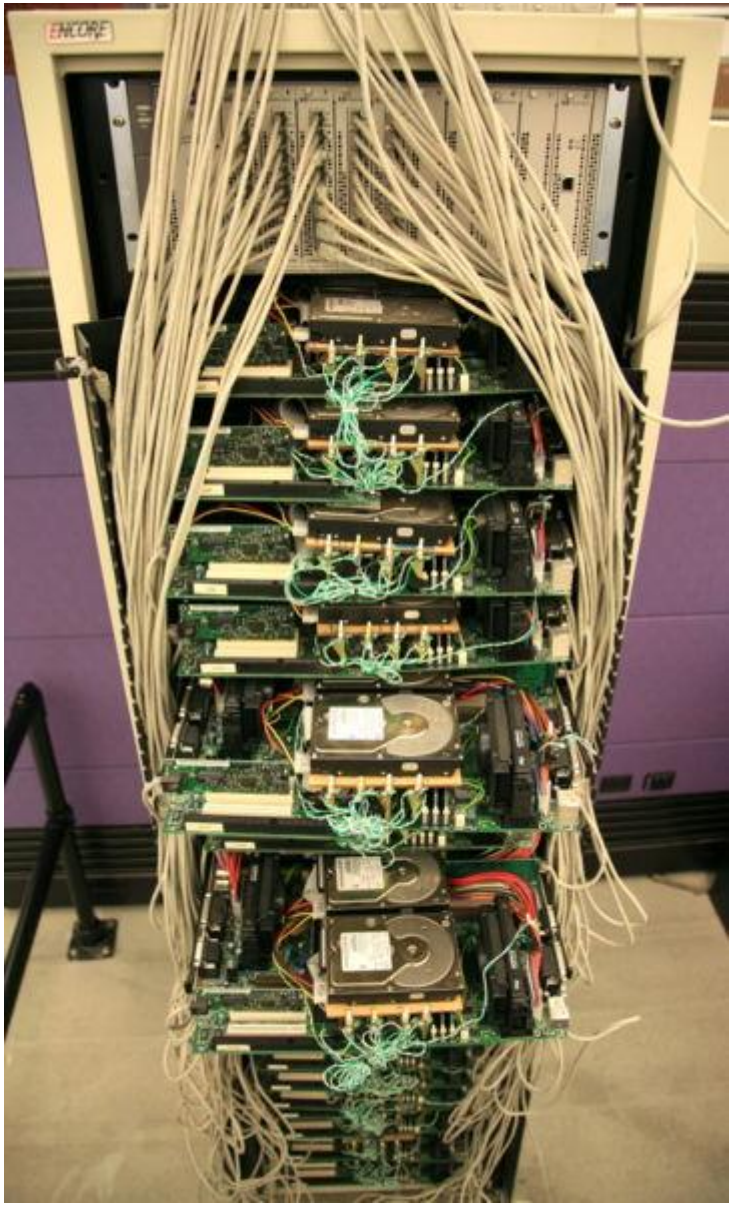
- Organising the data correctly helps to find it
- Predates computers by thousands of years
- Good for lookups, but bad for maintenance
  - Deletion leaves holes
  - Insertion is physical cut-and-paste
    - E.g. why a thesis might be double-spaced



Picture from https://en.wikipedia.org/wiki/Dictionary

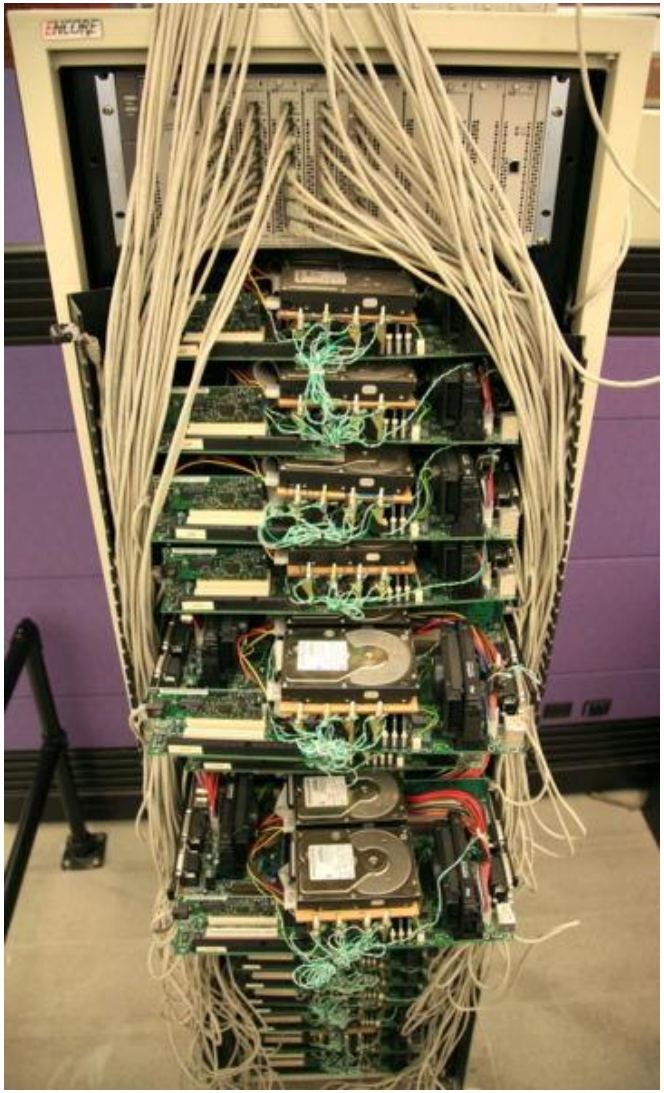
Data Structure Examples:

- Any guesses as to what this is?



Data Structure Examples: (ans)

- Any guesses as to what this is?
- Google's first (production) server



From https://en.wikipedia.org/wiki/History_of_Google

Aims of the ADE Module

Aim: understanding of issues involved in program design; good working knowledge of some common algorithms and data structures

Objectives:

- design data structures and algorithms which express this functionality in way that makes **efficient** use of
  - time
  - memory
  - **(and so indirectly: grid electricity usage, and battery usage)**
- be able to evaluate a given implementation in terms of its **efficiency**

Why care about efficiency?

Beware:

- Recent history is that hardware improved rapidly
  - This compensated for many poor programs
- 3GHz desktop with 2GB RAM on a small "first-year-style" program
  - almost any implementation will work 'instantly'

These might give a false sense that efficiency does not matter.

Why care about efficiency? (batteries!)

But:

- Clock speeds are 'stuck'
  - Not all programs can exploit multi-core, GPU, clouds, but must run on the single core
- Mobile devices are much less powerful
- Also 'inefficient' consumes more energy
  - **poorly written apps will drain the mobile device battery**
  - **2% (?) of world electricity goes in server farms**

Problem Oriented Approach

- The module has ‘maths’ and ‘theory’
- But these are designed to help with solving real problems/tasks
- What kinds of tasks? …

Example task 1

- “What method would you use to look up a word in a dictionary? (Correctly and efficiently)”

Example task 2
(optional offline exercise)

- "You are given a list of numbers. When you reach the end of the list you will come back to the beginning of the list (a circular list).
- Write the most efficient algorithm to find the minimum number in this list.
- Find any given number in the list.
- The numbers in the list are always increasing but you don't know where the circular list begins, e.g.: 38, 40, 55, 89, 6, 13, 20, 23, 36."

Example task 3
(optional offline exercise)

- "Write a function f(a, b) which takes two character string arguments and returns a string containing only the characters found in both strings in the order of a. Write a version which is order N-squared and one which is order N."

Example task 4

- "Describe the algorithm for a depth-first graph traversal."

Example task 5

- "Tree search algorithms. Write BFS and DFS code, explain run time and space requirements. Modify the code to handle trees with weighted edges and loops with BFS and DFS, make the code print out path to goal state."

Example task 6 (optional offline exercise)

- "There is an array A[N] of N numbers. You have to compose an array Output[N] such that Output[i] will be equal to multiplication of all the elements of A[N] except A[i]. For example Output[0] will be multiplication of A[1] to A[N-1] and
  Output[1] will be multiplication of A[0] and from A[2] to A[N-1].
  Solve it without division operator and in O(n)."

Example task 7

- How long it would take to sort 1 billion numbers?
- Come up with a rough estimate.
- "Billion" is 10^9 (US and modern UK usage) In old UK usage it was 10^{12}

- **<u>EXERCISE: Offline – try to estimate it using "back of an envelope" style</u>**

Previous "Example tasks" are irrelevant?

- **If you think the previous were irrelevant then be aware:**
- **They were taken (mostly) from "Google Interview Questions: Software Engineer"**
  - (http://blog.seattleinterviewcoach.com/2009/02/140-google-interview-questions.html#software_engineer but page is no longer active)

Interview Question

- Question: “There is an array A[N] of N numbers. … Solve it … in O(n).”

- What happens if your answer is
  - “what does ‘O(n)’ mean?”

  - ☹

Rough Context within "Programming"

- "Low-level" Programming (Modules in C, Assembly)
  - for, while loops etc

- **ADE: Algorithm Efficiency**
  - **trees, search algorithms, etc**
  - **code: "small but difficult" (each line might be 'hard')**

- Object-oriented design (Java / C++ modules)
  - software reuse, inheritance, etc

- Software systems architecture design (DMS?)
  - designing software components and interactions, etc
  - code: "easy but large" (design is hard, but each line tends to be 'easy')

- Algorithm Correctness – e.g. COMP2065-IFR
  - Proving correctness and/or termination
  - Requires "formal reasoning about programs" – VERY challenging for real-world programs

Rough Contents of the ADE portion

Basics/Theory:

- **Algorithm analysis, "big-Oh" and its family**
- (Revision) Abstract vs. Concrete Data Types
- (Revision?) Stacks, queues, lists, trees.
- Recurrence relations. "Master Theorem".

Sorting Algorithms:

- Bubble-sort, …, mergesort, quicksort, "stability", efficiency

Standard data structures

- Priority queues, Heaps, Hash tables

Search:

- Maps, hashmaps, binary search trees, balanced vs. unbalanced trees

Graphs & Algorithms:

- Minimum Spanning Trees, "dynamic programming", Shortest Paths (Dijkstra's algorithm, and Floyd-Warshall),

Maths needed includes:

- Properties of inequalities:
  - Reasoning about efficiency involves comparisons
- Properties of logarithms and exponents
  - Used to describe efficiency
    - (Generally don't need sin/cos etc.).
- Arithmetic and Geometric Series
- (Function fitting
  - "Regression" a.k.a. "trend lines in Excel")
- Proof by induction
- (Simple) Propositional and predicate logic

(A lot of this should be "revision".)
Tutorials and other material will support these skills.

Maths needed includes: 'comparisons"

- **Properties of inequalities**:

$a < b$ implies $2a < 2b$

$a \leq b$ implies $2a \leq 2b$

$a \leq b$ implies $a + c \leq b + c$

$a < b$ implies $-a > -b$

$a \leq b$ implies $-a \geq -b$

Etc.

Maths needed includes: (logs!)

- **Properties of exponentials**: (examples)

  $a^{(b+c)} = a^b a^c$ $\qquad$ $a^0 = 1$

  $a^{bc} = (a^b)^c$ $\qquad$ $a^b / a^c = a^{(b-c)}$

- **Properties of logarithms:**

  $b = a^{\log_a b}$ $\qquad$ ("definition of 'log base a' ")

  <u>Offline future exercises</u>:
  derive each of the following

  $b^c = a^{c*\log_a b}$

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b(x/y) = \log_b x - \log_b y$

  $\log_b x^a = a \log_b x$

  $\log_x a = \log_x b \log_b a$

  Quick self-test: $\qquad$ What is $\log_8(2)$?

  (Yes, we do need this!)

Maths: "Geometric and Arithmetic Series"

- **(Offline) Without using a calculator What are:**

"Geometric"

1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512

"Arithmetic"

1 + 3 + 5 + 7 + 9 + 11 + 13 + 15 + 17 + 19

Summary

- Main points of ADE part:
  - Develop habit of thinking about efficiency
    - E.g. Develop skills (mostly 'big-Oh' and family) to describe efficiency, and reason about efficiency

- But note:
  - This does not mean you always have to write the most efficient program
  - The skills (big-Oh family) will not be a panacea; they do not answer all questions about efficiency (but this does not mean they are irrelevant)

Notes

- Last year there was a module with code COMP2054
  - it was the Autumn portion of COMP2009-ACE – not the same as this module !

COMP2054-ADE
Algorithms Data Structures & Efficiency

ADE Lec01
Analysis of Algorithms

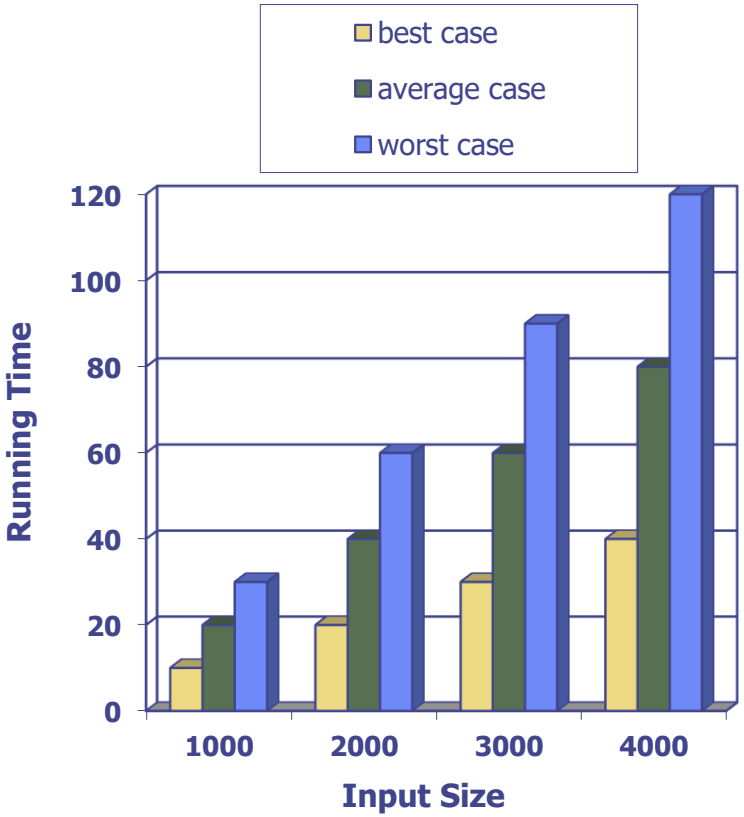
Lecturer: Andrew Parkes

Email: andrew.parkes@nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

Running Time: “finite” but how big?

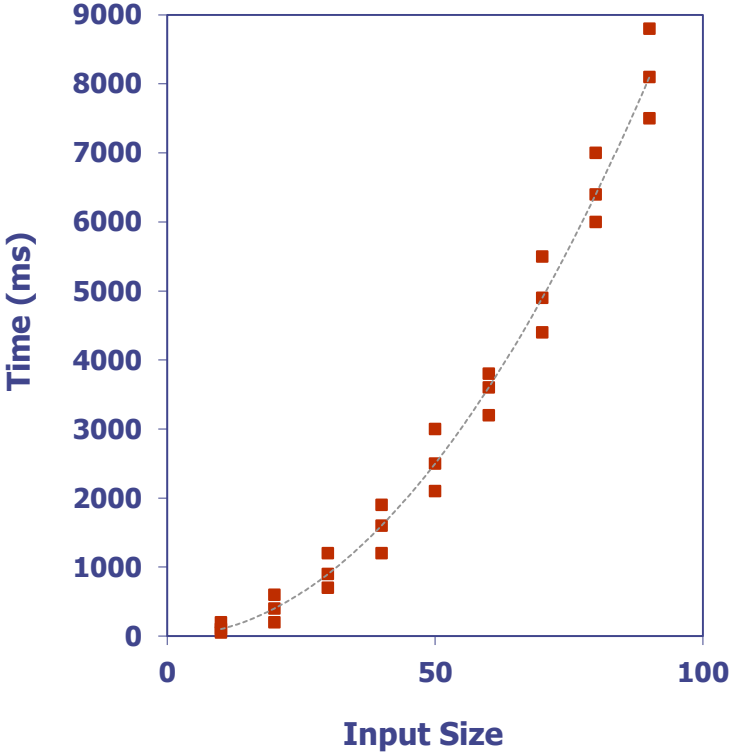
- Consider “batch algorithms”: transform input data into output data – as opposed to “interactive”
- The running time of an algorithm typically grows with the input size.
- Even at given size the runtime is usually not fixed.
  - So have “best”, “average” and “worst” cases.
  - A typical example →
- We (usually) focus on the worst case running time at given size
  - Useful, and easier to analyse
  - Average case time is often difficult to determine.



Experimental Studies

General Pattern:

- Write a program implementing the algorithm
- Run the program with inputs of "varying size and composition"
- Use a system method to get an (in)accurate measure of the actual running time
- Plot the results
  - Example is shown →
- Interpret & analyse. E.g. is it
  - A "power law", n^k, for some k
  - An "exponential", b^n, for some b



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult or time-consuming
- **Results may not be indicative of the running time on other inputs not included in the experiment.**
  - **Maybe we miss the "real worst case"**
- In order to compare two algorithms directly, the same hardware and software environments must be used

Limitations of Theory

- It is necessary to implement the theory, which may be difficult or time-consuming
- Results may not be indicative of the typical running time on inputs encountered in real world.

So can be useful to be able to use both experiment and theory.

Aside: Theory vs. Experiment

Standard science:
"Never believe a theory until it has been `confirmed' by an experiment"

Partially joking:

"Never believe an experiment until it has been confirmed by a theory"

- **Attributed to Sir Arthur Eddington https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3597502/**

Theoretical Analysis

- **AIM: Characterise running time as a function of the input size, *n*.**
- Uses a "high-level" description of the algorithm instead of an implementation
  - Takes into account all possible inputs
  - Allows us to evaluate the speed of an algorithm independently of the hardware/software/language environment

Pseudocode (recap)

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm ***arrayMax***(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details (recap)

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions

  ← Assignment (like = in Java)

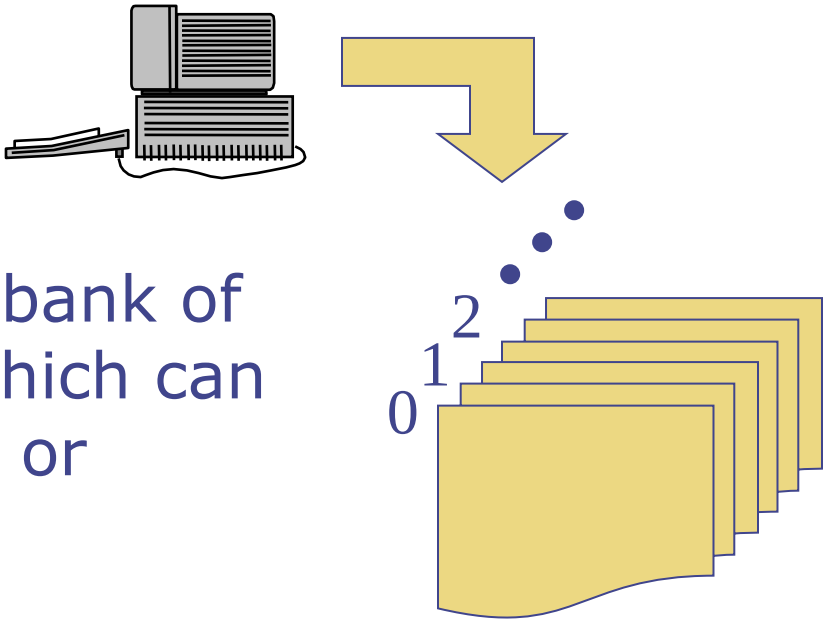
  = Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the "RAM model" (next slide)
  - Tend to be close to "Assembly language"
  - No "hidden expenses"
- Examples:
  - Assigning a value to a variable
  - Indexing into an array
  - Comparing two numbers
  - Adding/subtracting/ multiplying/dividing two numbers
  - Calling a method
  - Returning from a method

The Random Access Machine (RAM) Model



- A **CPU**
- A potentially-unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

- Memory cells are numbered and accessing any cell in memory takes unit time (some fixed time).
  - (Note that RAM can stand for both "Random Access Machine" and "Random Access Memory," Which is an unfortunate, but standard, over-loading of terminology.)

Limitations of RAM model

- "… can hold an arbitrary number …" ?
  - Can we really expect to store "938566359286151800351666177735777777177177374717717471571777776136566161816161616" in one cell on a real computer?
- Here, we ignore such "bignum" issues. Instead:
- "all numbers are of equal size, as they all fit in a single register of the CPU"
- 64bits (signed int) allows up to 9,223,372,036,854,775,807
  - Exercise (offline): compare to: nanoseconds since big bang; national debt.
- Note: on real machines (usually) computing

  1 + 1 takes as long as 381513 + 243542

  Hence, we typically ignore the sizes of numbers in the arithmetic operations.

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

	# operations
Algorithm ***arrayMax***(A, n)	
$currentMax \leftarrow A[0]$	?
for $i \leftarrow 1$ **to** $n - 1$ **do**	?
if $A[i] > currentMax$ **then**	?
$currentMax \leftarrow A[i]$	?
return $currentMax$	?
Total	**EXERCISE "Try it!"**

Counting Primitive Operations (partial) – "Offline -> Pause"

- **Worst case** number of primitive operations executed as a function of the input size, n

Algorithm ***arrayMax***(A, n)	# operations
$currentMax \leftarrow A[0]$	?
for $i \leftarrow 1$ **to** $n - 1$ **do**	?
if $A[i] > currentMax$ **then**	?
$currentMax \leftarrow A[i]$	$2\ (n - 1)$
	?
return $currentMax$	?
Total	??

Counting Primitive Operations (partial)

- Worst case number of primitive operations executed as a function of the input size, n

	# operations
Algorithm ***arrayMax***(A, n)	
$currentMax \leftarrow A[0]$	?
for $i \leftarrow 1$ **to** $n - 1$ **do**	1 // for $i \leftarrow 1$
if $A[i] > currentMax$ **then**	2 $(n - 1)$
$currentMax \leftarrow A[i]$	??
return $currentMax$	?
Total	??

Counting Primitive Operations (all)

- Worst case number of primitive operations executed as a function of the input size, n

Algorithm ***arrayMax***(A, n)	# operations	
$currentMax \leftarrow A[0]$	2	
for $i \leftarrow 1$ **to** $n - 1$ **do**	1	
if $A[i] > currentMax$ **then**	$2\,(n - 1)$	
$currentMax \leftarrow A[i]$	$2\,(n - 1)$	(worst case)
{ increment counter: i++ }	$2\,(n - 1)$	("hidden")
{ test counter: $i \leq$ (n-1) }	$2\,(n - 1)$	("hidden")
return $currentMax$	1	
Total	$8n - 4$	

Counting is "underspecified"

- Consider "$c \leftarrow A[i]$" then 'full' process can be
  - get A = pointer to start of array A, and store into a register
  - get i, and store into a register
  - compute A+i = pointer to location of A[i], and store back into a register
  - get value of "*(A+i)" (from RAM) and store value it into a register
  - copy the value into the location of c in the RAM
- might not want to count all this, e.g. just count
  - 'plus' of "A+i"
  - the assignment

Counting is "underspecified"

- There can be multiple right answers – if you get '2' and I count '4' then it does not mean you are wrong!
- Note: If I think an answer is '4' then '2' is probably also acceptable – but "2 n" probably will not be.
- It is most important to be able to
  - know what is happening in the underlying process
    - be able to link to C and assembly level notions
  - be able to use this to give a reasonably consistent justification of your answers

Note: Correctness vs. Efficiency

- Primitive operation counting is relevant to Efficiency – but not (directly) for Correctness
- For correctness, do not about runtime of the alg.
  - Do care about the time to find a proof if doing an automated search for proofs.
  - The verification seems to be quick in lean: e.g.

    #eval 2035713999 + 350135299

  - Very quick (<1sec) – not using succ internally (?)
- Note that we did not prove the algorithm correct
  - Would need to do arrays/lists in Lean first
    - Just for thought: And then what?
  - Can do efficiency of incorrect algorithms ☺

Estimating Running Time

- Algorithm $\boldsymbol{arrayMax}$ executes $8\boldsymbol{n} - 4$ primitive operations in the worst case. Define:
  - $\boldsymbol{a}$ = Time taken by the fastest primitive operation
  - $\boldsymbol{b}$ = Time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of $\boldsymbol{arrayMax}$. Then

$$\boldsymbol{a}\,(8\boldsymbol{n} - 4) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(8\boldsymbol{n} - 4)$$

- Hence, $\boldsymbol{T}(\boldsymbol{n})$ is bounded 'above and below' by two linear functions
- Usually said as "*arrayMax* runs in linear time"

Remarks

- Do not get too obsessed with the fine details of counting of primitive operations
- The details of the counting and timing would probably depend
  - the compiler, and require inspection of the assembly code
  - the CPU architecture, pipelining, cache misses, etc, etc

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- **The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm $arrayMax$**

Exercise: (exam-style question)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
    n ← n/2
    m++
return m
```

Give an analysis of its runtime

Exercise: what is T(n) of alg-lec1?

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2.

I.e. there exists k such that $2^k = n$

```
m ← 0
while (n ≥ 2)
        n ← n/2
        m++
return m
```

Exercise: what is T(n) of alg-lec1? (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	?
n ← n/2	?
m++	?
return m	1

Exercise

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	? per pass
n ← n/2	? per pass
m++	? per pass
return m	1

(Pause and try/think)

Internal Steps:

$n \leftarrow n/2$?

1. read n from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. read 2 from memory and store in a register r2
3. send registers r1 r2 through arithmetic division and store result in a register r3
4. write r3 back to n

CPU steps needed is 4, does not depend on n

Internal Steps: different compiler

$n \leftarrow n/2$?

1. read n from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. send registers r1 through a right shift of the bits and store result in a register r3
   e.g. compute 13/2=6 by 1101 → 110
3. write r3 back to n

CPU steps needed is 3, different but still does not depend on n

Exercise (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1	
while (n ≥ 2)	3	per pass
n ← n/2	3	per pass
m++	3	per pass
return m	1	

Thought Exercise (offline)

- Based on your knowledge of assembly, and machine architectures try to estimate the number of CPU cycles that might actually be used.
- "Divide" or "shift"? Which is faster?
- **Point: try to eventually build a mental model that is an "internal interpreter" so as to know how a program will run**
- **Such "internal interpreters" are vital for understanding programming (IMHO)**

How many passes through the loop of alg-lec1?

Hint: If ever stuck:

- Try simple concrete examples
- Start from "ridiculously simple" and work up to harder examples

(Real mathematicians often work this way; but then hide it when writing up ☹)

- Do a "trace of the program" by hand:

How many passes through loop?

- Focus on the relevant portions:

  while (n ≥ 2)

  n ← n/2

- Simplest example?
- Exercise: What is smallest positive integer that is a power of two?
- Answer: 1 as $2^0 = 1$
- (If confused, or if you answered "2", then consider revising your maths about exponents and logarithms)

How many passes through loop? (cont)

while (n ≥ 2) { n ← n/2 ; }

- Case: $n = 1 = 2^0$ passes = 0
- Case: $n = 2 = 2^1$
  - n=2, then n=1 ; passes=1
- Case: $n = 4 = 2^2$
  - n=4, then n=2, then n=1 ; passes = 2
- Case: $n = 8 = 2^3$
  - n=8, 4 , 2, then n=1 ; passes = 3

How many passes through loop? (cont)

- Case: $n = 2^m$
  - $n=2^m, 2^{m-1}, 2^{m-2}, \ldots, 2$
  - m passes through loop
- but note, n is the input not m, so want to write answer in terms of n. Use
  - $m = \log_2(n)$
- Result: passes through loop = $\log_2(n)$

Exercise (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	$3\ (\log_2(n)+1)$
n ← n/2	$3\ \log_2(n)$
m++	$3\ \log_2(n)$
return m	1

all together: $9\ \log_2(n) + 5$

(the "+1" on line 2, is because the test is done even if it fails)

Remarks

- Each pass through the loop the size of n is halved
  - the "$\log_2(n)$" is typical of such "halving on each iteration"
- **This concept also appears in sorting and searching; hence you MUST make sure you fully understand this example**
  - **The ADE half of the module will probably be incomprehensible otherwise**

Summary

Goal:

- Build foundations for time-analysis of programs

Skills needed:

- Count primitive operations
- Counting of operations with
  - Loops
  - (Recursion)

Removing details

- According to precisely how we count steps we might get many different answers, e.g. something like
  - $5 \log_2(n) + 2$
  - $9 \log_2(n) + 5$, etc
- Also this counts "steps"
  - the translation to runtime depends on the compiler, hardware, etc
- **Need a way to suppress such details**

Next Lecture

"Suppressing the details"

A motivation and introduction to big-Oh

Oh… ohh!!

Exercise: (Advanced, Offline/Take-home)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
      n ←  sqrt(n)
      m++
return m
```

Give an analysis of its runtime.

For the counting, assume that the square root "sqrt" is a primitive operation.

COMP2054-ACE: Introduction to big-Oh

ADE-Lec02a

Lecturer: Andrew Parkes

Email: andrew.parkes@Nottingham.ac.uk

from http://www.cs.nott.ac.uk/~pszajp/

Recap: Removing details

- (For an example program) According to precisely how we count steps we might get one of
  - $5 \log_2(n) + 2$
  - $9 \log_2(n) + 5$, etc.
- Also this counts "steps"
  - the translation to runtime depends on the compiler, hardware, etc.
- **Need a way to suppress such 'implementation-dependent details'**

Aim: **Classification of Functions**

- CS needs a way to group together **functions** by their scaling behaviour, and the classification should
  - Remove unnecessary details
  - Be (relatively) quick and easy
  - Handle 'weird' functions that can happen for runtimes
  - Still be mathematically well-defined
- Experience of CS is that this is best done by the "big-Oh notation and family":

  O (Big-Oh), Ω (Big-Omega), Θ (Big-Theta),
  o (little-oh) and ω (little-omega)

"Surprising"/"Advanced" uses of Big-Oh family

- Stirling's approximation (which we use later)
  https://en.wikipedia.org/wiki/Stirling%27s_approximation

  $$\ln(n!) = n \ln n - n + O(\ln n)$$

  - where "ln" is the natural logarithm "log base e"
  - (Note there is no reference to an algorithm!)
- Polynomial functions are $n^{O(1)}$
- The "best case of algorithm X is O(n log n)"
- Does there exist an algorithm for TSP with runtime $2^{o(n)}$?

Need a good/proper understanding of Big-Oh family to interpret the above

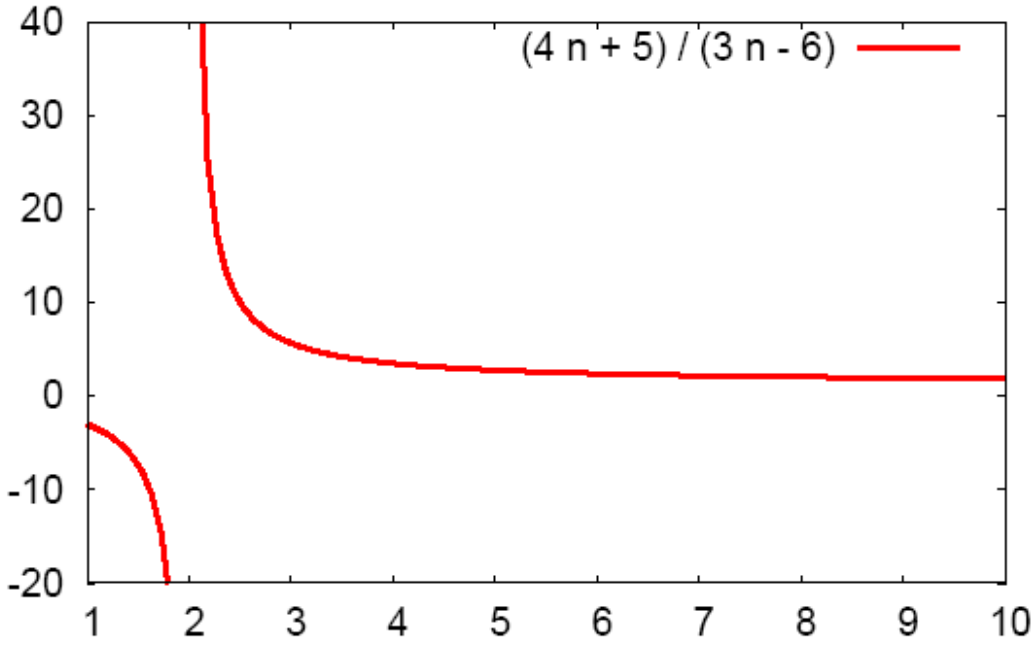
Big-Oh Notation: Motivations

- Suppose we have two functions
  1. $4n + 5$
  2. $3n - 6$
- We want to express that the behaviour is driven by fact that both are linear in n, and to suppress the details of the exact function
- One way (not the only way!) to motivate definitions is to look at their ratio:

Important (Subtle?) Comment

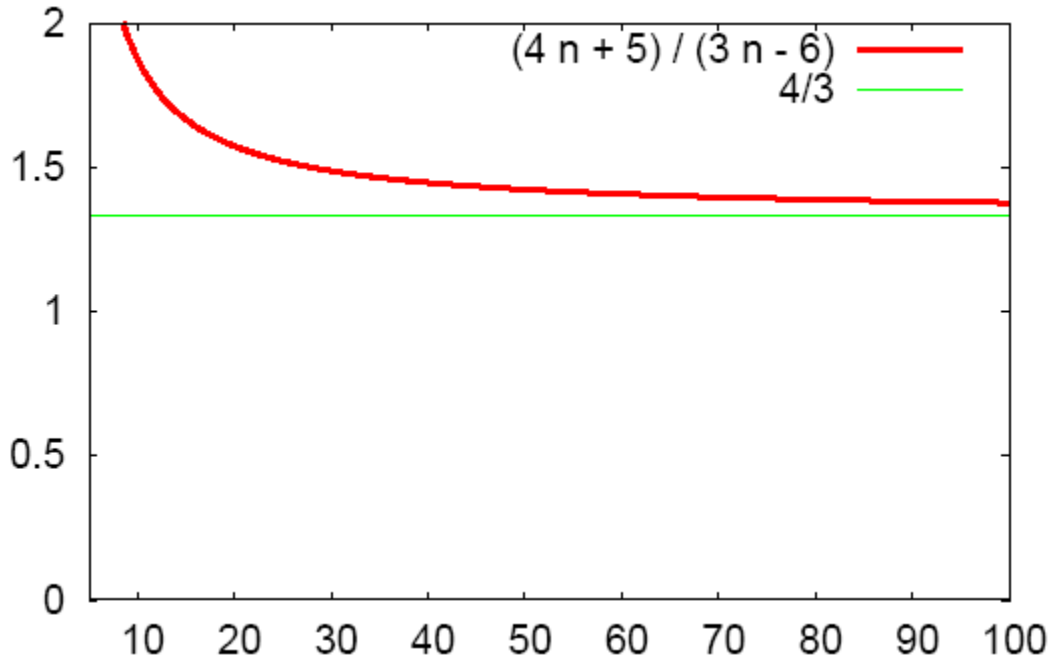
- Note that at this point we are just talking about functions, "f(n)", of a single parameter n
- Big-Oh is often applied to runtimes, but this is not essential
  - The big-Oh definitions are just in terms of functions
  - It is not only for "worst case runtime"
  - But the big-Oh definition should be "designed" to be suitable for discussion of "runtime functions"
  - How do we motivate a good design of the definitions?

Ratio of two linear functions



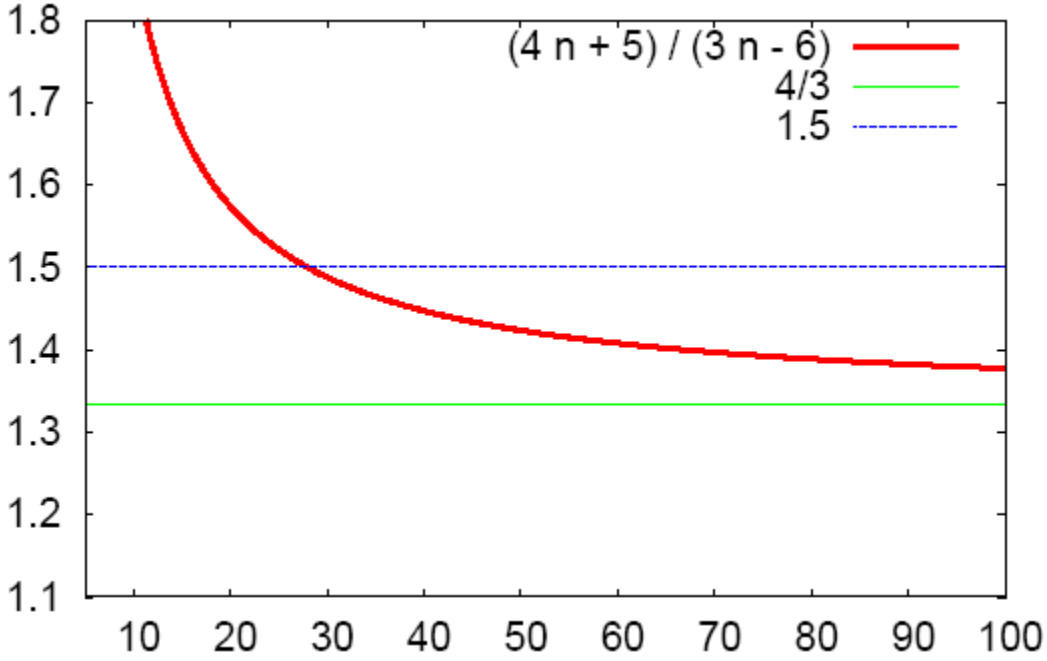
- The behaviour at small ***n*** is messy

Ratio of two linear functions



- At larger values of ***n*** the ratio starts to behave more predictably
  - Suggests definitions should use "forall n >= ..."

Ratio of two linear functions



- It looks like:
  $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 30$?

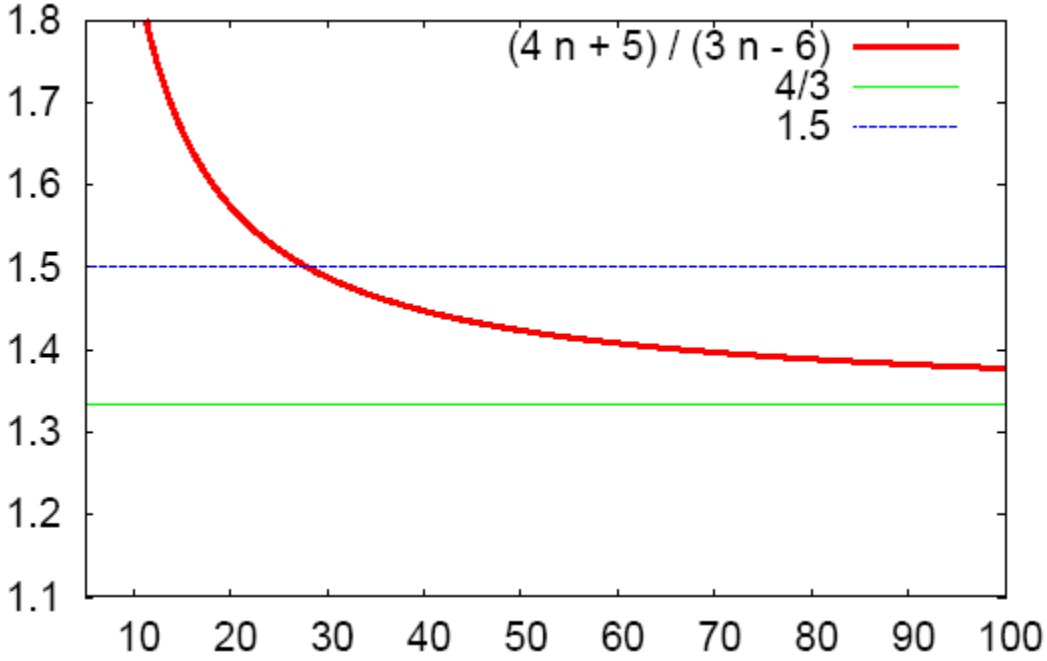
Exercise: computation

- For precisely what values of n is the following true:

  $$(4n+5) \leq 1.5 * (3n-6)$$

- Answer:
  - $8n + 10 \leq 9n - 18$ (by x2)
  - $28 \leq n$ (add 18-8n to both sides)

Ratio of two linear functions



- Hence we have:

  $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 28$
- Suggests definitions that place upper bounds on the function relative to another function:

Big-Oh Notation: Definition

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$$f(n) \text{ is } O(\, g(n)\,)$$

if and only if there exist positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \qquad \text{for all } n \geq n_0$$

THIS DEFINITION IS VITAL – PLEASE QUESTION, LEARN AND UNDERSTAND ALL PARTS OF IT.

Which functions are used?

- Big Oh is intended to functions of **positive integers** (size), and that are **positive real values** because they (often) represent runtimes:
  - f : $\mathbb{N}^+ \rightarrow \mathbb{R}^+$ (and similarly for g)
  - Where $\mathbb{N}^+ = \{ 1, 2, 3, \ \ldots \}$ and
  - $\mathbb{R}^+ = \{ x \in \mathbb{R} \,|x \geq 0 \}$
- That is, assume, $f(n) \geq 0 \ \ \forall n \geq 1$
- Sometimes, for convenience, might relax to
  - $f(n) \geq 0 \ \ \forall n \geq N$ for some constant N

Big-Oh Notation: Definition***

Carefully note the structure and order of the quantifiers:

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\ g(n)\ )$

if and only if **there exist** positive constants c and n_0 such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

i.e. "exists-exists-forall" structure:

$\exists\ c>0.\ \exists\ n_0.$ such that $\forall\ n \geq n_0.\ f(n) \leq c\, g(n)$

A common mistake is to get the nesting and placement of the quantifiers in the wrong order

Big-Oh Notation: Definition

Carefully note that the quantifier structure is "pick c, before handling forall n" and so c is not allowed to be a function of n:

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\ g(n)\ )$

if and only if **there exist** positive constants c and n_0

such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

- We cannot pick that c depends on n.
- Otherwise we can (usually) just take c = f/g and the definition becomes useless as it would not fail, so
  - it would not allow us to say anything useful about the relative growth rates of f and g.

Big-Oh Notation: Definition

Carefully note "forall n …"

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\ g(n)\ )$

if and only if **there exist** positive constants c and n_0 such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

Showing something works at $n = n_0$ is pointless as says nothing about the growth rates at large n.

How do formally prove statements of "forall n"?

In worst case would need induction;
but in general we will just use "obvious properties" ("standard lemmas")

Big-Oh Notation: Definition***

Note the words that do **not** appear in

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

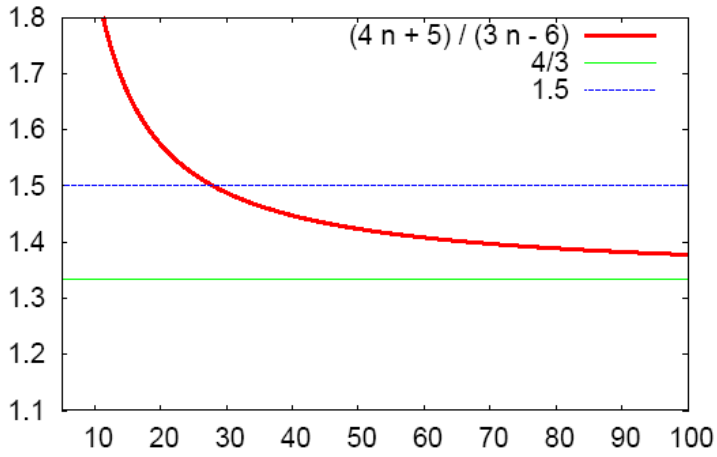
$f(n)$ is $O(\ g(n)\ )$

if and only if **there exist** positive constants c and n_0 such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

The definition does not mention "worst case of an algorithm runtime". It does not even mention "algorithm".

It only mentions "functions".

Ratio of two linear functions



- Since: $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 28$
- We now have: $(4n+5)$ is $O((3n-6))$
  - Using $c=1.5$ and $n_0=28$

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = 1

  is O(1)

  ANS: (pause and try!)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = 1

  is O(1)

  ANS: pick c,n0 s.t 1 <= c 1 forall n>=n0

  c=1 n0=1 Done.

  c=3499993, n0=392875

  Proof still works. **Definition does not say pick smallest c,n0.**

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

$$f(n) = 2$$

is O(1)?

ANS: (pause and try!)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

$$f(n) = 2$$

is O(1)?

ANS: need to
pick c,n0 s.t. 2 <= c 1 A n >= n0
c=2 n0=1
DONE

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \le c\ g(n)$ **for all** $n \ge n_0$

- Show that the function:

  f(n) = k

  is O(1) for any fixed constant k

ANS: pick c,n0. k <= c 1

c = 2*max(k,1) is allowed (not using n)

n0=2

c=k/2 fails – but does not matter as need "exists c" not "forall c"

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

  f(n) = 1

  is O(n)

  ANS: (pause and try!)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

  f(n) = 1

  is O(n)

  ANS: pick c,n0. s.t. 1 <= c n forall n >= n0

  n0=1

  c=1 need 1 <= n forall n >= 1

DONE: i.e. 1 is O(n),

but have 1 is O(1)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

$$f(n) = k$$

is O(n) for any fixed constant $k \geq 0$

ANS:

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

$$f(n) = k$$

is O(n) for any fixed constant $k \geq 0$

ANS: want

$$k \leq c\ n \qquad \text{for } n \geq n_0$$

Note that c=k, n_0=1 will suffice

Hence, k is O(n), as well as O(1).

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  $f(n) = k\ n$

  is O(n) for any fixed constant k

ANS: (pause and try!)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n + k

  is O(n) for any fixed constant k

ANS: pick c,n0 s.t
  n+k <= c n forall n >= n0

Pick c=1 n+k <= n fails (for k>0)

c=2 k <= n pick n0=k DONE

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n

  is **not** O(1).

  ANS: (pause and try)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n

  is **not** O(1).

  ANS: can we pick? c,n0 s.t.

  n <= c forall n >= n0

Pick c = n ? NO!!!!!!

CANNOT pick c. c=infty is not allowed

Hence not O(1).

It is also O(n^2), but O(n) is "best".

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n

  is not O(1).

ANS: We would need to provide **constants** c and n_0 (c cannot depend on n) and prove

$$n \leq c\ .\ 1 \qquad \forall\ n \geq n_0$$

But this is clearly impossible.

E.g. take n=max(c+1,n_0).

BREAK until next lecture lec02b

COMP2054-ACE: Introduction to big-Oh

ADE-Lec02b

Lecturer: Andrew Parkes

Email: andrew.parkes@Nottingham.ac.uk

from http://www.cs.nott.ac.uk/~pszajp/

Recap
Aim: **Classification of Functions**

- In computer science, we often need a way to group together **functions** by their scaling behaviour, and the classification should
  - Remove unnecessary details
  - Be (relatively) quick and easy
  - Be able to deal with 'weird' functions that can happen for runtimes
  - Still be mathematically well-defined
- Experience of CS is that this is best done by the "big-Oh notation and family"

Recap: Big-Oh Notation: Definition***

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$$f(n) \text{ is } O(\, g(n)\,)$$

if and only if there exist positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \qquad \text{for all } n \geq n_0$$

THIS DEFINITION IS VITAL – PLEASE QUESTION, LEARN AND UNDERSTAND ALL PARTS OF IT.

RECAP EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- It is easy to show directly from the definition that the function:

  f(n) = 1 is O(n)

  I.e. we have both

  1 is O(n) - "strange but true"
  1 is O(1) - "natural"

Nothing in the definition forces a choice of a "best" or "most useful" g(n).

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Oh behaviour?

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Oh behaviour?

ANS: It is O(n).

Proof: if $n \geq 1$ then $f(n) \leq n$

hence can take $c=1$ $n_0=1$

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Oh behaviour?

ANS: It is O(n).

Proof: if $n \geq 1$ then $f(n) \leq n$

hence can take $c=1$ $n_0=1$

Note: despite the two cases, "even"/"odd", the definition requires to provide <u>one</u> c and <u>one</u> n0.

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

Is it O(1) ?

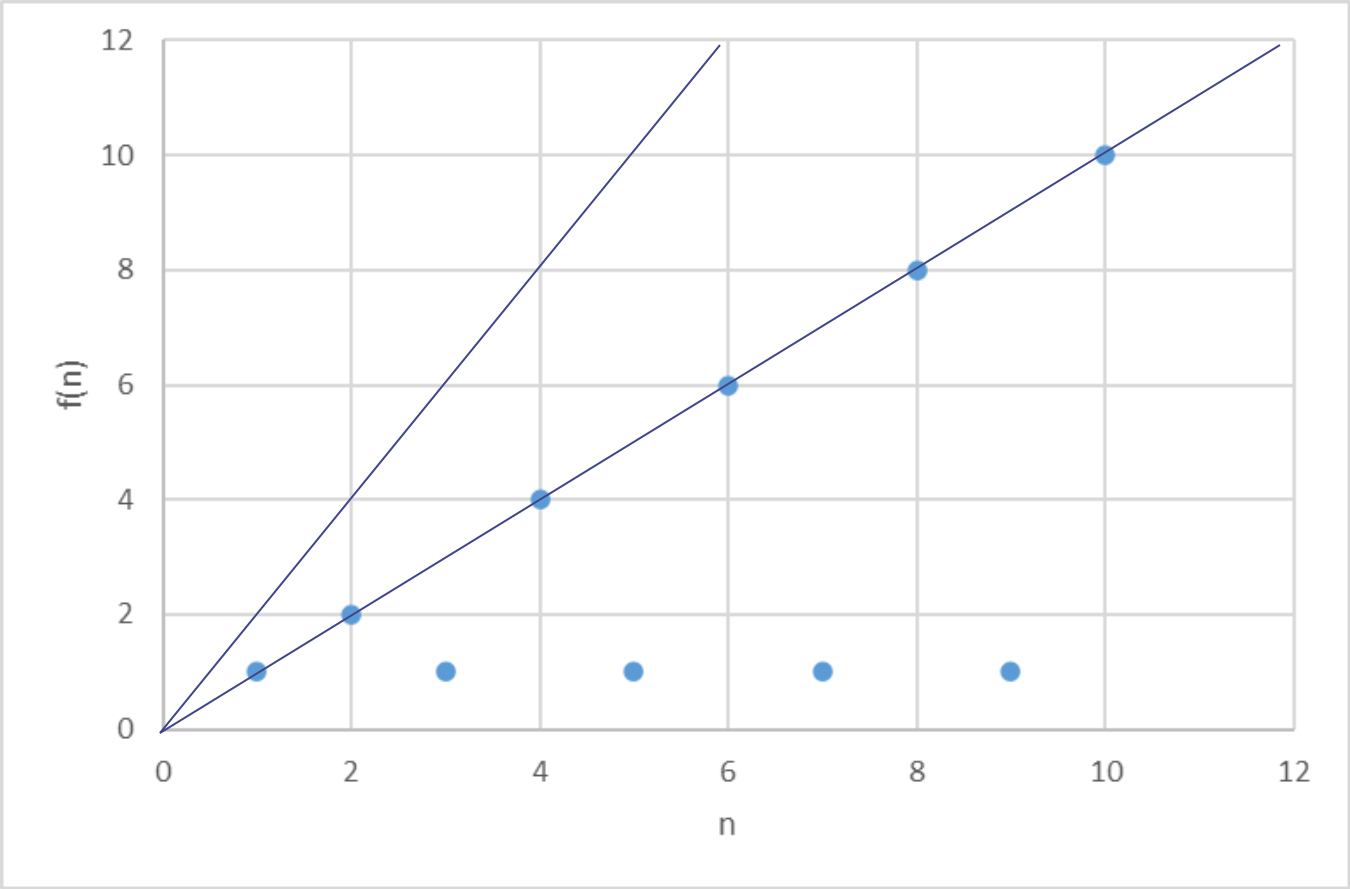
ANS: No, as we would need

Exists c,n0. n ≥ c 1 forall n >= n0

And this fails for the cases n is even.

Big-Oh Graphically

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$



EXERCISE

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

What is the limit of the ratio $f(n)/n$ as n tends to ∞ (infinity, i.e. becomes very large)?

EXERCISE (ans)

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

What is the limit of the ratio $f(n)/n$?

ANS: It does not have a limiting ratio !

$$f(n)/n = \begin{cases} 1 & \text{if } n \text{ is even, limit is } 1 \\ 1/n & \text{if } n \text{ is odd, limit is } 0 \end{cases}$$

Shows how "big-Oh" handles weird functions and is different from limits.

Ratios vs. big-Oh

- f(n) can be O(g(n)) even if the ratio f(n)/g(n) does not exist
- Hence, big-Oh can be used in situations that ratios cannot
- The possibility of 'weird functions' means that big-Oh is more suitable than ratios for doing analysis of efficiency of programs

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f2(n) = \begin{cases} n & \text{if n is even} \\ 4 & \text{if n is odd} \end{cases}$$

What is its big-Oh behaviour?

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

$$f2(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 4 & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Oh behaviour?

ANS: It is O(n).

Proof: if $n \geq 4$ then $f2(n) \leq n$

hence can take $c=1$ $n_0=4$

"other providers of c and n0 are available"

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Consider the function:

  f2(n) = n if n is even
  4 if n is odd

What is its big-Oh behaviour?

ANS: It is O(n).

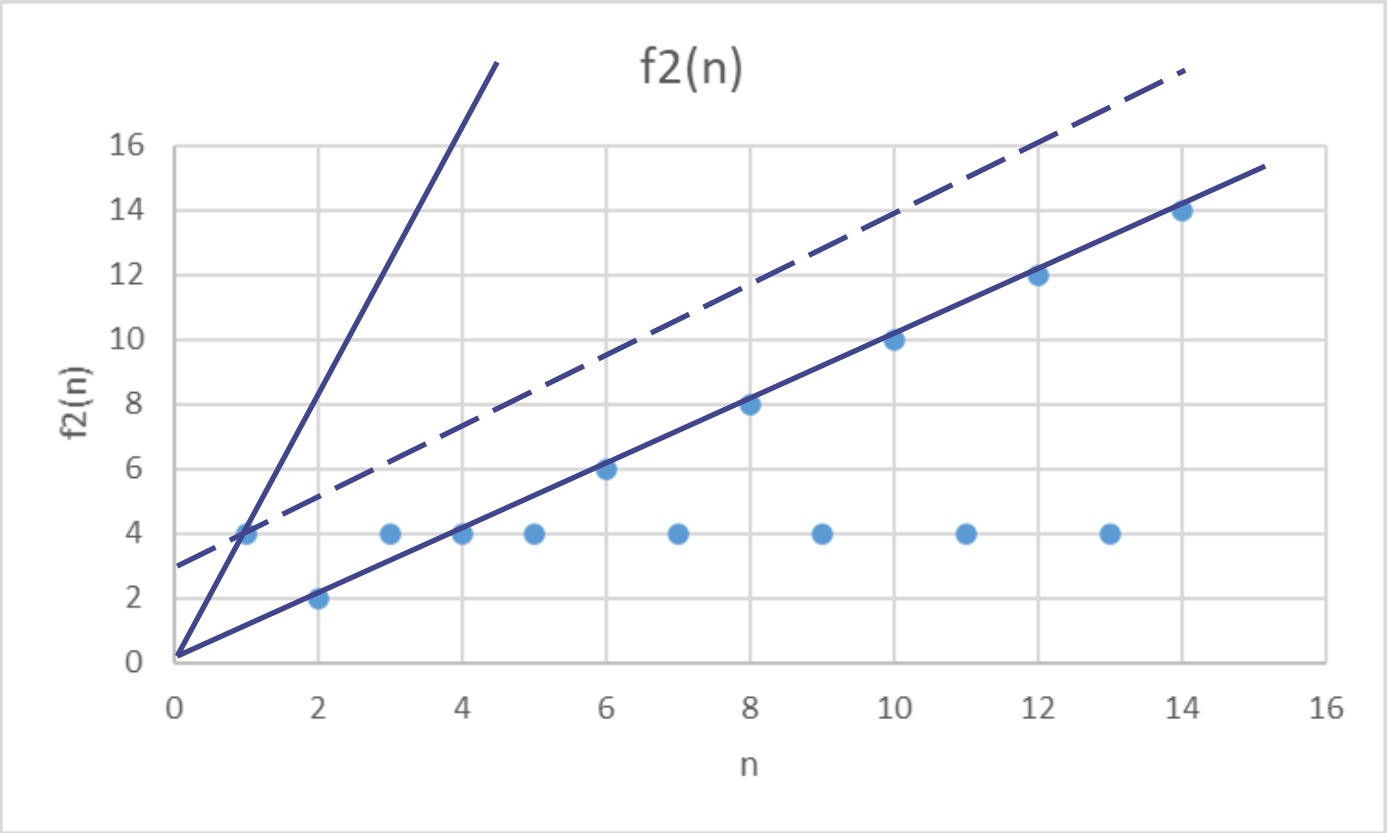
Not a (full) proof "from the definition":
for even case: pick c=1 n0=1
for odd case: pick c=4 n0=1.

(In lean: your proof would fail)

EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- "Messy at small n" ; fine past n=4



Exercises (offline)

Defn: Given functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\ g(n)\ )$

if and only if there exist positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

- From the definition, show that
  - (3n-6) is O((4n+5))
  - (3n-6) is O(n)
  - (4n+5) is O(n)
- Notes
  - The first shows that g(n) does not need to be "nice"
  - The last two 'suppress details' as desired

Comment on some details:

- f(n) = (3n-6) is not always positive, but only is negative on a finite number of values of n
- Hence, to be more strict:
  - Could instead consider f(n) = max(0,3n-6) is really meant
  - Or could just require that n_0 is taken large enough that f(n) is positive: that is require

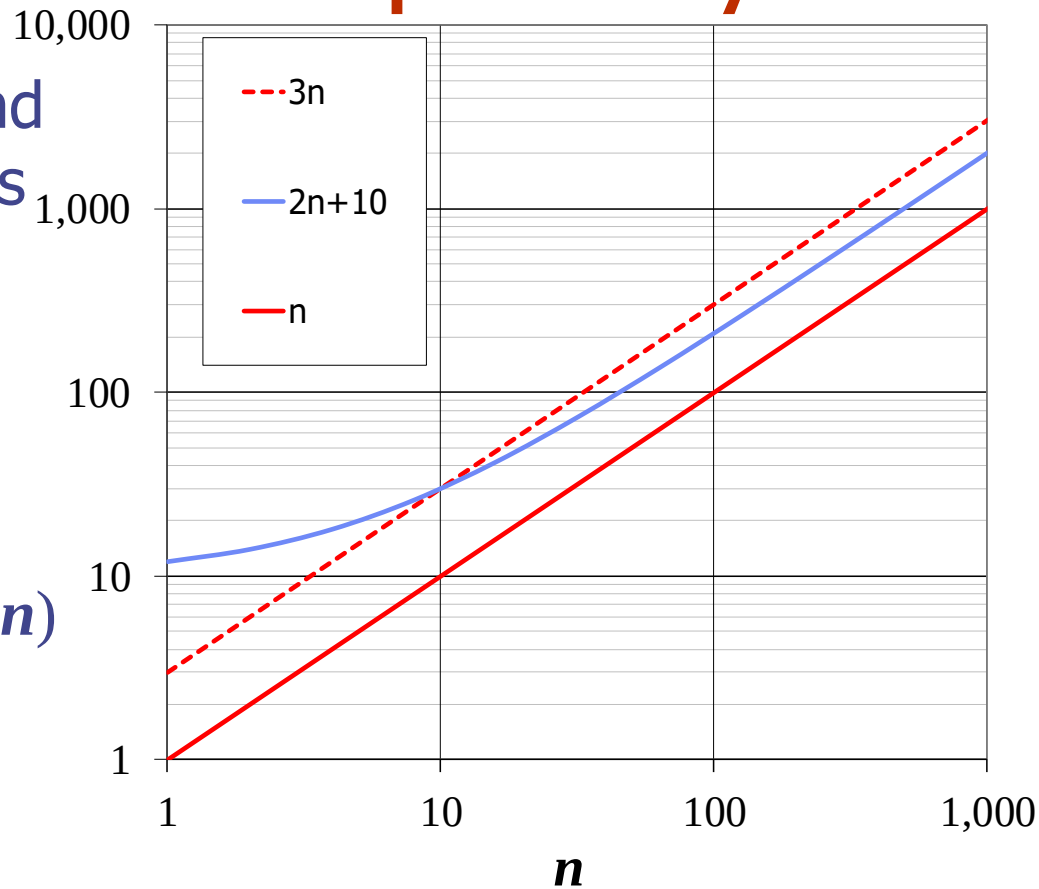
$$\forall\, n \geq n_0. \qquad 0 \leq f(n) \leq c\, g(n)$$

Either of these capture the intent that "f grows no faster than g at large enough n"

Big-Oh Notation: Graphically

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

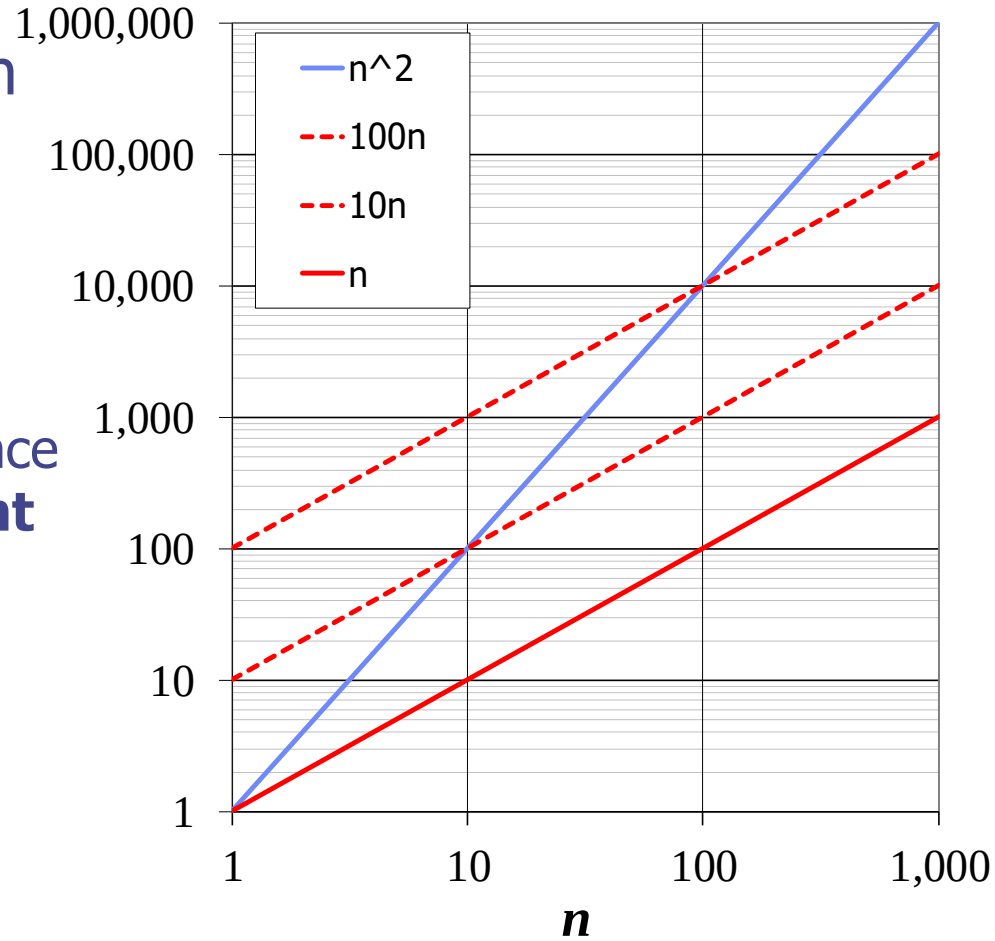
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c **must be a constant**

- Can also see this graphically:



Big Oh as a binary relation

- "f is O(g)" can be regarded as a binary relation between two functions f and g
  - Could be written: "$O(f,g)$" or "$f\ O\ g$"
- Generally, binary relations, R, are characterised by potential properties:
  - Reflexive: x R x
  - Symmetric: x R y iff y R x
  - Transitive: x R y & y R z → x R z

Which of these are satisfied by big-Oh?

Big-Oh is Reflexive

- This is trivial:

- For any function

f(n) is O(f(n))

as just take c=1, and use $\forall n. f(n) \leq f(n)$

Big Oh is not symmetric

- 1 is O(n)

But (from an earlier example)

- n is not O(1)

It only takes one counterexample to show that we cannot say it is symmetric

Big Oh is transitive

- Given "f is O(g)" and "g is O(h)", then can we show "f is O(h)" ?
  - E.g.
    "1 is O(n)" and "n is $O(n^2)$"
    forces
    "1 is $O(n^2)$" ?
- As usual, start by writing down what we know from the definitions:

Big Oh is transitive

- Given "f is O(g)" and "g is O(h)" then can we show "f is O(h) ?
- Exercise: Start by writing down what we know from the definitions:
  exists c1 n1 c2 n2 s.t.
  f(n) <= c1 g(n) forall n >= n1
  g(n) <= c2 h(n) forall n >= n2
  How do we get to "f is O(h)"?

Big Oh is transitive

- Given "f is O(g)" and "g is O(h)" then can we show "f is O(h)" ?
- exists c1 n1 c2 n2 s.t.

  f(n) <= c1 g(n) forall n >= n1

  g(n) <= c2 h(n) forall n >= n2
- Mult 2nd inequality by c1 (uses c1 >0)

  c1 g(n) <= c1 c2 h(n) forall n >= n2
- Set n3 = maximum(n1,n2), and combine gives

f(n) <= c1 g(n) <= c1 c2 h(n) forall n >= n3

Then can take c = c1*c2 to complete the proof.

Big-Oh as a set

- Big Oh as a binary relation is reflexive and transitive but not symmetric
  - It behaves like "$\subseteq$", "$\in$" or "$\leq$", not like "$=$"
  - One might say $n \in O(n)$, and $2n+3 \in O(n)$, etc.
- So may help to think of "O(n)" as a set of functions, with each function f in the set, $f \in O(n)$, satisfying "f is O(n)".
  - Or can say: $\{f\} \subseteq O(n)$
  - So then $O(1) \subseteq O(n)$
    - Any function bounded above by a constant, is also bounded above by a linear function
  - This gives a closer match to hierarchical classification, similar to

    Tom $\in$ Cats, Cats $\subseteq$ Mammals

Big-Oh as a set - notation

May help to think of "O(n)" as a set of functions, with each function f in the set, $f \in O(n)$, satisfying "f is O(n)".

- Many texts use "f = O(n)", but
- I dislike this because usual intuition about "=" would make it natural to say:
  - 1 = O(1) and 1 = O(n) hence O(1) = O(n) which is WRONG
  - E.g. O(n) contains f(n)=1, but O(1) does not
- Also, we cannot swap the order:
  - 1 = O(1) does not mean we can write "1 = O(1)" which is meaningless

Big-Oh as a set

- So may help to think of "O(n)" as a set of functions, with each function f in the set, $f \in O(n)$, satisfying "f is O(n)".
- This allows to give sense to expressions such as

$$n^{O(1)}$$

  to mean "n to the power of f(n) with f(n) being O(1)".
- Hence, this included n, n^2, n^3, etc
- Can sometimes be used as way to say "is some power law"
  - (Under "big-Oh as worst case of an algorithm" then $n^{O(1)}$ would make no sense!).

Summary & Expectations

- Know the definition of big-Oh well!
  - Understand why it is appropriate in CS
- Be able to apply it, and prove results on big-Oh of simple functions
- Big Oh as a binary relation is reflexive and transitive but not symmetric
  - It behaves like "$\subseteq$", "$\in$" or "$\leq$", not like "$=$"

Exercises (offline)

- Show $7n-2$ is $O(n)$

- Show $3n^3 + 20n^2 + 5$ is $O(n^3)$

- Show $3 \log n + 5$ is $O(\log n)$

COMP2054-ADE

ADE Lec03
Rules for quick big-Oh proofs

Lecturer: Andrew Parkes

Email: andrew.parkes@Nottingham.ac.uk

from http://www.cs.nott.ac.uk/~pszajp/

Big-Oh Notation: Definition***

Definition: Given (positive) functions $f(n)$ and $g(n)$, then we say that

$$f(n) \text{ is } O(\, g(n)\,)$$

if and only if there exist positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \qquad \text{for all } n \geq n_0$$

THIS DEFINITION IS VITAL – PLEASE QUESTION, LEARN AND UNDERSTAND ALL PARTS OF IT.

Rules for Finding big-Oh

- Reverting to the definition each time is time-consuming and error prone
- Better to develop a set of rules that allow us to very quickly find big-Oh

"Multiplication Rule" for big-Oh

- Suppose
  - $f_1(n)$ is $O(\ g_1(n)\ )$
  - $f_2(n)$ is $O(\ g_2(n)\ )$
- Then, from the definition, there exist positive constants c_1 c_2 n_1 n_2 such that
  - $f_1(n) \leq c_1\ g_1(n)$ for all $n \geq n_1$
  - $f_2(n) \leq c_2\ g_2(n)$ for all $n \geq n_2$
- Let $n_0 = \max(n_1\ ,\ n_2)$, then multiplying gives
- $f_1(n)\ f_2(n) \leq c_1\ c_2\ g_1(n)\ g_2(n)$ for all $n \geq n_0$
- So $f_1(n)\ f_2(n)$ is $O(\ g_1(n)\ g_2(n)\ )$

"Multiplication Rule" for big-Oh

- "Sanity checks":
  - On doing something new it is usually good to consider some "trivial" cases and verify it works correctly - the "unit tests" of maths ☺
- E.g. What is the big-Oh of f(n) = 5 n ? Does it "get rid of the constants"?

Use 5 is O(1)
n is O(n)

Hence 5 * n is O (1 * n) which is O (n), as expected.

Big-Oh Rules: Drop smaller terms

- If $f(n) = (1 + h(n))$
  with $h(n) \to 0$ as $n \to \infty$
- Then f(n) is O(1)
- (The utility will be to combine with the multiplication rule).

Proof (sketch):

- $h(n) \to 0$ as $n \to \infty$ means that for large enough n then h(n) will become arbitrarily close to zero
- Hence, in particular, there exists n_0 such that
  $h(n) \leq 1$ for all $n \geq n_0$
- So, $f(n) \leq 2$ for all $n \geq n_0$
- Hence is f(n) is O(1) (by using c=2 in the definition)

Exercise

- What is the big-Oh of $f(n) = n^2 + n$?

Exercise

- What is the big-Oh of $f(n) = n^2 + n$?
- ANS:

$$f(n) = n^2 * (1 + 1/n)$$

$1 + 1/n$ is $O(1)$ by 'drop small terms'

n^2 is trivially $O(n^2)$

then use multiplication rule,

$f(n)$ is $O(n^2 * 1) = O(n^2)$

Big-Oh Rules

- After some thought it should become clear that:
- If $f(n)$ a polynomial of degree d, (with positive largest term) then $f(n)$ is $O(n^d)$, i.e.,
    1. Drop lower-order terms
    2. Drop constant terms

Note: degree of a polynomial is the highest power
e.g. $5\ n^4 + 3\ n^2$ is degree 4 and so will be $O(n^4)$

Proof (sketch):

rewrite the polynomial as $k * n^d * (1 + \ldots)$

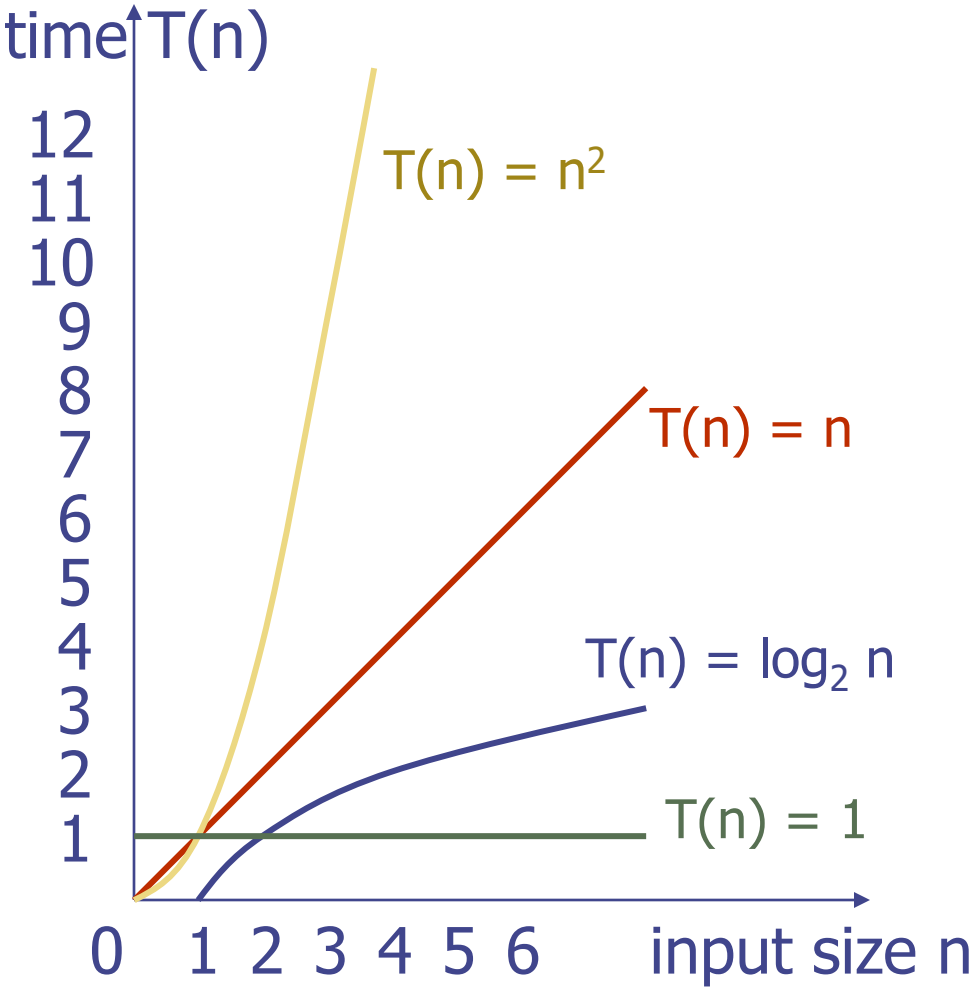
Seven Important Functions

Seven functions that **often** appear in algorithm analysis:

- Constant ≈ ***1***
- Logarithmic ≈ log ***n***
- Linear ≈ ***n***
- N-Log-N ≈ ***n*** log ***n***
- Quadratic ≈ n^2
- Cubic ≈ n^3
- Exponential ≈ 2^n

YOU NEED TO KNOW THESE WELL!!

Plot some graphs yourself



Recall: "Drop smaller terms"

If $f(n) = (1 + h(n))$
with $h(n) \to 0$ as $n \to \infty$

Then f(n) is O(1)

- To use this we will need to have some rules for limits of various functions
  - E.g. what do we do with (1 + (log(n) / n))
  - Does h(n) = log(n) / n go to 0 as n -> infinity?
- If have a messy function can try plotting a graph.
- But there are some useful rules:

Useful limits: exponents vs. powers

- Exponentials grow faster (as $n \to \infty$) than any power:
- for any fixed $k>0$, and $b > 1$

$$b^n / n^k \to \infty$$

and

$$n^k / b^n \to 0$$

E.g. $n^2 / 2^n \to 0$

E.g. $n^{2000} / 2^{n/100} \to 0$

Exercise

- What is the big Oh of $f(n) = 2^n + n^2$?

Reminder on Terminology:

We say "n^2" is a "power law" but we do not call it an "exponential".
An "exponential function of n" is one with n in the exponent, such as 2^n not just having a constant exponent (such as the 2 in n^2).

The difference between n^2 and 2^n is enormous (try computing them at n=1000).

Exercise

- What is the big Oh of $f(n) = 2^n + n^2$?
- ANS:

$$f(n) = 2^n * (1 + n^2 / 2^n)$$

but $n^2/2^n \rightarrow 0$ so can drop it.

Hence $f(n)$ is $O(2^n)$

Useful limits: powers vs. logs

- Powers grow faster (as $n \to \infty$) than any power of a log (assume positive powers)

$$n / (\log n) \to \infty$$
$$(\log n) / n \to 0$$

More generally:

$$(\log n)^k / n^{k'} \to 0 \quad \text{for any fixed } k, k' > 0$$

E.g. $(\log n)^{100} / n^{0.1} \to 0$

Though it might not be obvious until large n.

Roughly:

"**Exponentials dominate powers which dominate logs**"

Exercise

- What is the big Oh of $f(n) = (n \log n) + n^2$?

Exercise

- What is the big Oh of $f(n) = (n \log n) + n^2$?
- ANS:

$f(n) = n^2 * ((\log n)/n + 1)$

But $(\log n)/n \rightarrow 0$ so can drop it

Hence $f(n)$ is $O(n^2)$

Exercises (offline):

Give the big-Oh of the following

1. $3\ n^3 + 10000\ n$
2. $n \log(n) + 2\ n$
3. $2^n + n$

If need help, then ask, e.g. in labs/tutorials.

Big-Oh Conventions

Conventions:

- Use the smallest (slowest growing) 'reasonable' possible class of functions
  - Say "$2\boldsymbol{n}$ is $\boldsymbol{O}(\boldsymbol{n})$" instead of "$2\boldsymbol{n}$ is $\boldsymbol{O}(\boldsymbol{n}^2)$"
- Use the simplest expression of the class
  - Say "$3\boldsymbol{n}+5$ is $\boldsymbol{O}(\boldsymbol{n})$" instead of "$3\boldsymbol{n}+5$ is $\boldsymbol{O}(3\boldsymbol{n})$"

Usage of 'O' in practice

- From the definition is true that for any f(n) that f(n) is O(f(n))
- But such an answer is inappropriate because conventions say that we want a 'useful' answer, not a trivial one.
- If asked for the 'big-Oh' then want the 'tightest nice function' - this is defined by community standards, and done so as to convey the maximum (practical) information
- Warning: This is a convention and not directly part of the definition – and so causes a lot of confusion.
- **It will be expected on assessments that you use the conventions when appropriate**

“Algorithm” or “Problem”

- We will see that the big-Oh for a solution to a particular problem depend on the algorithm used.
- Point: picking an appropriate algorithm is needed in order to get good behaviour, and the big-Oh helps this

- It would often be nice to know, for a particular problem, the “best possible big-Oh”, but
  - such lower-bounds are very rarely known
  - it is very hard to do such analyses
  - there are many problems where the entire CS community has entirely failed to make progress, see Millennium prize on P vs. NP

Exercise

- What is the big Oh of $f(n) = 2^{n/100} + n^{200}$?
- ANS:

$$f(n) = 2^{n/100} * (1 + n^{200} / 2^{n/100})$$

but $n^{200}/2^{n/100} \rightarrow 0$ so can drop it.

Hence f(n) is $O(2^{n/100})$.

But in practice, the large power law will dominate until n is very large.

"Big-Oh" is not a panacea – need to know when it might be misleading

Exercise

Consider an (extreme) example

- Algorithm A has runtime $f1(n) = n^{20}$
- Algorithm B has runtime $f2(n) = 2^{n/10}$

Which is the best?

- If only look at Big-Oh, then A wins
- But at n=100 we have
  - $f1(100) = 100^{20} = 10^{40}$
    far more than the 10^{80} atoms in the universe
  - $f2(100) = 2^{10} = 1024 \sim 10^3$
- Now B clearly wins.
  - Less extreme versions of this happen e.g. "simplex algorithm" (used in optimisation) is exponential in worst case, but often better in practice.
- "Big-Oh" is not a panacea!

Reminder: Big-Oh as a "Set"

One can think of O(n) as

"the set of all functions whose growth is no worse than linear for sufficiently large n"

Hence, it can be thought of as the (infinite) set $\{1,2, \ldots \log n, 2 \log n, \ldots, n,2n,3n,\ldots,n+1,n+2, \ldots\}$

Then "2n+3 is O(n)" is just the statement that the function 2n+3 is in this set, i.e. $2n+3 \in O(n)$

Big-Oh as a "Set"

Although many sources do it, personally,
I recommend against writing $n = O(n)$, because

- $n = O(n^2)$
- $n = O(n)$

should lead to $O(n) = O(n^2)$ which is wrong!

Though, note that $O(n) \subset O(n^2)$.

That is, if $f(n) \in O(n)$ then $f(n) \in O(n^2)$

(though not the converse)

The (subset) inclusion is strict
e.g. consider n^2 which is in $O(n^2)$ but not in $O(n)$

Big-Oh: Usage for Algorithms

Big-Oh definitions themselves, in pure sense, are just ways of classifying functions and not algorithms

Their usage for runtimes of algorithms has further choices. One can use big-Oh to describe any of:

- Worst case runtime, w(n), at each value of n
- Best case runtime, b(n), at each value of n
- Average case runtime, b(n), at each value of n
- etc

If simply say "algorithm X is O(.)" then the usual convention is that it will refer to the worst case.

Big-Oh: Usage for Algorithms

- Worst case runtime, w(n), at each value of n
- Best case runtime, b(n), at each value of n
- Average case runtime, b(n), at each value of n

If simply say "algorithm X is O(.)" then the usual convention is that it will refer to the worst case.

But have the freedom to say:
For algorithm X,

- the worst case (over all possible inputs) is $O(n^3)$
- the average case, given inputs generated uniformly at random, is $O(n^2)$
- the best case (over all possible inputs) is O(n).

E.g. will see:
worst case of quicksort is $O(n^2)$ but average case is O(n log n)

Big-Oh: Usage for Algorithms

Consider:

"Merge-sort is O(n log n)"

expands into:

"If running merge-sort on n integers, then the worst case run-time over all possible inputs, is a member of the set of functions that, is no worse than some fixed constant times n log(n) for all values of n that are at least some fixed value."

Which follows the definition, but the convention might also imply it is the best, and so we might add that the O is a "tight bound" or "cannot be improved" because:

"there will be some inputs for which the runtime is as bad as this."

Asymptotic Algorithm Analysis in practice

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the (worst-case, etc.) number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm $\boldsymbol{arrayMax}$ executes at most $8\boldsymbol{n} + 3$ primitive operations
  - We say that algorithm $\boldsymbol{arrayMax}$ "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"
- In practice:
  Since constant factors and lower-order terms are eventually dropped anyhow, if we only want the big-Oh behaviour, then we could decide to disregard them when counting primitive operations – but be careful not to drop the important terms!

Summary & Expectations

Included, (but not limited to)

- Know the definition of big-Oh well!
  - Be able to apply it, and prove results on big-Oh of simple functions
  - Know how to manipulate
    - Sums
    - Products

  of functions, and be able to prove if needed
- Know the "conventions of usage" and also the advanced usages e.g. "best case is O(.)" "$n^{O(1)}$"

Next Lecture

Big-Omega: Definition

Definition: Given functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $\Omega(\ g(n)\ )$

if there are (strictly) positive constants c and n_0 such that

$f(n) \geq c\, g(n)$ for all $n \geq n_0$

Spot the difference?

COMP2054-ADE:

ADE Lec04a: The Big-Oh family

Lecturer: Andrew Parkes

andrew.parkes 'at' Nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

Relatives of Big-Oh

A close family:

- **big-Oh** **'O'**
- **big-Omega** **'Ω'**
- **big-Theta** **'Θ'**

- **little-oh** **'o'**
  - note this is not in the main text-book but is required for the module
- **little-omega** **'ω'**
  - Not required, as not used a lot, though mention it for completeness

Big-Omega: Definition

Definition: Given functions $\boldsymbol{f}(\boldsymbol{n})$ and $\boldsymbol{g}(\boldsymbol{n})$, we say that

$$\boldsymbol{f}(\boldsymbol{n}) \text{ is } \boldsymbol{\Omega}(\ \boldsymbol{g}(\boldsymbol{n})\ )$$

if there are (strictly) positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$\boldsymbol{f}(\boldsymbol{n}) \geq \boldsymbol{c}\ \boldsymbol{g}(\boldsymbol{n}) \qquad \text{for all } \boldsymbol{n} \geq \boldsymbol{n_0}$$

Big-Omega: Definition

Definition: Given functions $\boldsymbol{f}(\boldsymbol{n})$ and $\boldsymbol{g}(\boldsymbol{n})$, we say that

$$\boldsymbol{f}(\boldsymbol{n}) \text{ is } \boldsymbol{\Omega}(\ \boldsymbol{g}(\boldsymbol{n})\ )$$

if there are (strictly) positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$\boldsymbol{f}(\boldsymbol{n}) \geq \boldsymbol{c}\ \boldsymbol{g}(\boldsymbol{n}) \qquad \text{for all } \boldsymbol{n} \geq \boldsymbol{n_0}$$

- Spot the difference from big-Oh?
- "greater than" rather than "less than"
- Note that need c > 0, and we not allowed c=0
- Note that c must be **constant (cannot depend on n)**

Exercise (online):

- Show n is $\Omega(n)$

Need c > 0, n0 such that

n >= c n forall n >= n0

Try c = 1

n >= n forall n >= n0

Pick n0=1.

DONE.

Exercise (online):

- Show n^2 is $\Omega(n)$

Need c > 0, n0 such that

n^2 >= c n forall n >= n0

Try c = 1

n >= 1 forall n >= n0

Pick n0=1

Also works if try c=2

n >= 2 forall n >= n0

n0 = 2

DONE.

Exercise (online):

- Show $n^3 - n$ is $\Omega(n^3)$

Need c > 0, n0 such that

$n^3 - n >= c\, n^3$ forall n >= n0

Try c = 1

$n^3 - n >= n^3$ forall n >= n0

$- n >= 0$ forall n >= n0

FAILS

Try c = 1 / 2

$2\, n^3 – 2\, n >= n^3$ forall n >= n0

$n^2 >= 2$ forall n >= n0

n0 = 2 DONE.

Exercise (online):

- Is it true that: 1 is $\Omega(n)$?

Need c > 0, n0 such that

1 >= c n forall n >= n0

Note: No need for c to be integer

Try c = … FAILS

Instead use c > 0 to get

(1/c) >= n forall n >= n0

FAILS as eventually n > (1/c)

FAILS.

1 is NOT $\Omega(n)$

c = 1/n is NOT allowed – need c constant

Big-Omega Examples

- We have
  - n is $\Omega(1)$
  - n is $\Omega(n)$
  - n is not $\Omega(n^2)$
- This is "the opposite of Big-Oh"
- **f(n) is Ω(g(n)) says that "f(n) grows at least as fast as g(n) at large n"**
- Compared to
  - f(n) is O(g(n)) says that "f(n) grows no faster than g(n) at large n"

Big-Omega properties

- Similarly to big-Oh, Big-Omega is
  - Reflexive
  - NOT symmetric
  - Transitive

- Exercise (offline): Prove these.

Linking big-Oh and Big-Omega

- Suppose that we are give “f is O(g)”
- What can we say about big-Omega?

Linking big-Oh and Big-Omega

- Suppose that we are give "f is O(g)"
  - We know there exist c n_0 such that

    $f(n) \leq c\, g(n) \quad \forall\, n \geq n_0$
  - We can assume $c > 0$ (Why?). Hence
    - $g(n) \geq \left(\frac{1}{c}\right) f(n) \quad \forall\, n \geq n_0$
  - Hence, $g\ is\ \Omega(f)$.
- That is: $f \in O(g) \rightarrow g \in \Omega(\mathrm{f})$
- Similarly: $f \in \Omega(g) \rightarrow g \in \mathrm{O}(\mathrm{f})$.
- Note: similar to: x <= y → y >= x

Usage of big-Omega

- Once familiar with big-Oh then big-Omega is very similar but "upside down". Same rules apply:
  - Multiplication rule still applies
  - Can still drop smaller terms
- E.g. $n^3 - n$ is $\Omega(n^3)$
- 'Ω ' expresses "grows as least as fast as"
- 'O' expresses "grows at most as fast as"

‘Paradox’ of big-Omega

- “Can still drop smaller terms”

- E.g. $n^3 - n$ is $\Omega(n^3)$

- Note that this might seem counter-intuitive
  - thinking “big Omega does `at least’ and so need the smallest term” is incorrect!
  - instead think of the large n behaviour being dominated by the n^3, and that this grows at least as fast as n^3.

Omega Usage

- A standard usage might be to capture a limitation on the best one can hope for,
  e.g. "*The best case for algorithm X is $\Omega(n^3)$ and so it will not scale well*"
- However, if the worst case behaviour of an algorithm is a 'bizarre' or 'partially unknown' function of n, then one might say,
  e.g. "*The worst case for algorithm X is not precisely known but we know it is $\Omega(n^3)$ and $O(n^4)$*"
- Gives a lot more flexibility than doing ratios.

Big-Theta: Definition

Definition: Given functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $\Theta(\, g(n)\,)$

if there are positive constants c' , c'' and n_0 such that

$$f(n) \leq c' \, g(n)$$

$$f(n) \geq c'' \, g(n)$$

for all $n \geq n_0$

- *What does this say about the growth rate of f(n) ?*
- *How does it connect to O and Omega?*

Big-Theta

Definition: Given functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $\Theta(\, g(n)\,)$

if there are positive constants c' , c'' and n_0 such that

$$f(n) \le c'\, g(n)$$

$$f(n) \ge c''\, g(n)$$

for all $n \ge n_0$

- *f(n)* is *Θ(g(n))* if and only if
  f(n) is *O(g(n))* and also *f(n)* is Ω*(g(n))*
- Example: *2n+1* is *Θ(n)*
- *Θ expresses "grows 'exactly' as fast as"*

EXERCISE

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 2n & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Omega behaviour?

Clearly is $\Omega(n)$.

Just take c=1 n0=1 then

f(n) >= 1 n forall n >= 1

EXERCISE

- Consider the function:

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 2n & \text{if } n \text{ is odd} \end{cases}$$

What is its big-Theta behaviour?

Clearly is $\Theta(n)$. As is both $O(n)$ and $\Omega(n)$.

But note that there is not a single value for ratio $f(n)/n$.

$\Theta(n)$ expresses the growth rate is linear but does not tightly sandwich the specific values.

- This is very useful in CS !
- Sometimes people can really mean Big-Theta when they say Big-Oh

Big-Theta is reflexive and transitive

- Trivial as both O and Ω are.

Is Big-Theta symmetric?

- If $\mathrm{f} \in \Theta(g)$ then it is (automatically and always) true that $\mathrm{g} \in \Theta(f)$

- Exercise (offline): (Follows quickly from previous results).

Θ is an equivalence relation

- Any relation that is
  - Reflexive & Symmetric & Transitive
- is an "equivalence relation"
  - Roughly speaking: it behaves like a "equality":
  - $\Theta(g(n))$ is "the equivalence class of all functions whose large n behaviour is bounded above and below by constants times g(n)"
- It is reasonable to write "$f = \Theta(g)$"
  - (pedantically, arguably, could be "$\{f\} = \Theta(g)$" but dropping "{}" on singleton sets is a common abuse of notation. But also it is somewhat of an abuse as it is not a set inequality, rather it is an "equivalence relation", i.e. closer to "$\{f\} == \Theta(g)$")
  - "Reasonable" is not same as "recommended" ☺

BREAK

- Next lecture “little-oh”

COMP2054-ADE:

ADE Lec04b: The Big-Oh family

Lecturer: Andrew Parkes

andrew.parkes 'at' Nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

Recall: Relatives of Big-Oh

A close family:

- **big-Oh** **'O'**
- **big-Omega** **'Ω'**
- **big-Theta** **'Θ'**

- **little-oh** **'o'**
  - note this is not in the main text-book but is required for the module
- **little-omega** **'ω'**
  - Not required. We will not cover this; but include here so you know it exists.

Recall: Big-Omega: Definition

Definition: Given functions $\boldsymbol{f}(\boldsymbol{n})$ and $\boldsymbol{g}(\boldsymbol{n})$, we say that

$$\boldsymbol{f}(\boldsymbol{n}) \text{ is } \boldsymbol{\Omega}(\ \boldsymbol{g}(\boldsymbol{n})\ )$$

if there are (strictly) positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$\boldsymbol{f}(\boldsymbol{n}) \geq \boldsymbol{c}\ \boldsymbol{g}(\boldsymbol{n}) \qquad \text{for all } \boldsymbol{n} \geq \boldsymbol{n_0}$$

Recall: Big-Theta: Definition

Definition: Given functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $\Theta(\, g(n)\,)$

if there are positive constants c' , c'' and n_0 such that

$$f(n) \leq c' g(n)$$

$$f(n) \geq c'' g(n)$$

for all $n \geq n_0$

- *Means both big-Oh and Big-Omega together.*

Little-Oh: Definition

Definition: Given (positive) functions $f(n)$ and $g(n)$, we say that

$$f(n) \text{ is } o(\, g(n)\,)$$

if for all positive (real) constants $c > 0$

there exists n_0 such that

$$f(n) < c\, g(n) \quad \text{for all } n \geq n_0$$

- Spot the difference from big-Oh?
  - "**for all c > 0**" rather than "**there exists c > 0**"
  - (The change of "≤" to "<" is much less important – see later.)
- Says (roughly) that the ratio *f(n)/g(n) → 0* as *n → infinity*

Little-Oh: Definition

Definition: Given positive functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $o(\ g(n)\ )$

if **for all** positive (real) constants $c > 0$

there exists n_0 such that

$f(n) < c\, g(n)$ for all $n \geq n_0$

- Note that n_0 is allowed to depend on c
  - (As it is inside the scope of the "forall c")
  - It is not: "exists n0, forall c>0"
  - As ever, the order of quantifiers is vital
- This is relevant, and vital, because we must consider all positive values of c.
  - E.g. the inequality must be satisfied for each of c=1, 0.1, 0.01, 0.001, ...
  - c does not need to be "nat"

Exercise

Claim: 1 is o(n)

Exercise

Claim: 1 is o(n)

Claim: **for all** $c > 0$ there exists some n_0 such that

$$1 < c\,n \qquad \text{for all } n \geq n_0$$

E.g. for $c = 0.1$ we would get

$1 < 0.1\,n$ i.e. $n > 10$,

then, for example, we can take $n_0=20$
(or any $n0 > 10$)

Hence, suggests we in general we can pick $n_0 = 2/c$

Proof of claim just uses $1 < c\,n$ for all $n \geq 2/c$

Note that $c>0$ is essential.

Exercise

Claim: n is $o(n^2)$

Is it true that **for all** $c > 0$ there exists some n_0 such that

$$n < c\, n^2 \qquad \text{for all } n \geq n_0$$

Exercise: complete the proof of the claim

Exercise

Prove or disprove: n is $o(n)$

Exercise

Prove or disprove: n is $o(n)$

Is it true that **for all** $c > 0$ there exists some n_0 such that

$$n < c\,n \qquad \text{for all } n \geq n_0(c)$$

I.e. $1 < c$ for all $n \geq n_0(c)$

This is not true for all c, e.g. it fails for c=0.5

Hence, n is NOT o(n)

Generally: little-oh is NOT reflexive
it is more like "<" than "<="

Little-Oh: Definition

Definition: Given functions $f(n)$ and $g(n)$, we say that

$f(n)$ is $o(\ g(n)\ )$

if for all positive constants $c > 0$

there exists n_0 such that

$f(n) < c\, g(n)$ for all $n \geq n_0$

- If *f(n) is o(g(n))* then 'obviously' *f(n) is O(g(n))*

- Vital Exercise: study the definitions until this really is 'obvious' !!

Little-Oh: Usage of rules

- Can also do multiplication:
  f1 is o(g1), f2 is o(g2) implies
  f1*f2 is o(g1*g2)
- Also, can "drop smaller terms"
- Standard results are
  - "powers of logs" are o("powers")
    (assuming "positive powers)
    e.g. $(\log n)^k$ is o(n), and even o(sqrt(n)), etc.
- "powers" are o("exponentials")
  e.g. n^k is o(2^n)
  e.g. n^{100} is o(1.0001^n)

Intuition or 'mnemonics' for Asymptotic Notation

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically "**less than or equal"** to g(n)

Big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically "**greater than or equal"** to g(n)

Big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically "**equal"** to g(n)

little-oh

- f(n) is o(g(n)) if f(n) is asymptotically "**strictly less than"** g(n)

Intuition or 'mnemonics' for Asymptotic Notation

There is also

Little-omega

- f(n) is ω(g(n)) if f(n) is asymptotically "**strictly greater"** g(n)

And is

- Defined in a similar fashion to little-oh
- The "strict" version of Big-Omega
- but it is rarely used in CS
- (not required in this module)

Exercises (offline)

- Make up simple questions and answer them
  - Using simple functions
  - E.g. look at the 'big oh' examples of slides, and work out some theta, omega, little-oh results

- Repeat until 'happy' ☺

Exercise (offline)

Multiple choice (pick one answer)

- If $f(n)$ is $o(g(n))$ then
  1. it can never also be $\Omega(g(n))$
  2. it can sometimes be $\Omega(g(n))$
  3. it is always $\Omega(g(n))$

Important: Even if you cannot do the proofs then do enough examples to be able to have an intelligent guess on such multiple choice questions!

Example (offline):

The function $90\ n^2 \log(n) + n^3$ is

A. $O(\ n^3\ )$ and $o(\ n^3\ )$

B. $O(\ n^2\ )$ and $\Theta(\ n \log n\ )$

C. $O(\ n \log n\ )$

D. $\Omega(\ n^3\ )$

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Caveats & Cautions

- The point of big-Oh-family is that is can hide constants and lower-order terms; but sometimes these are important
- E.g. 100000000 n might be impractical despite being O(n)
- E.g. O(1.02^n) might be practical despite being "exponential"
- Also the worst case might happen too rarely to matter
  - But would you want to ignore it in your flight control software?

Exercises (offline)

- From the seven functions of a previous slide determine which are O, Ω, Θ and o of n^2

- Work through all the exercises in these slides (and the tutorials!)

Example of Usage

- For a complicated algorithm one could have an analysis such as:

- *"The worst case for algorithm X is known to be $o(n^4)$ and also $\Omega(n^3)$ but the exact behaviour is not known.*
- *The best case is known to be $\Theta(n^2)$.*
- *The average case (over uniformly random inputs) is $O(n^3)$."*

- Note: e.g. the true worst case could be $\Theta(n^{3.5})$ but it could be too hard to compute.

Summary So Far

- Role of Algorithms and Data Structures within computer science
- Random Access Machine (RAM) model
- Developing methods to reason about programs
  - Counting of steps of algorithms in RAM
  - Use of "big-Oh" to omit irrelevant and machine-specific details
  - Maths of big-Oh, Omega, Theta, little-Oh
  - Ability to analyse big-Oh of (simple) programs

“Appendix”

- The following is included to
  - Give a deeper insight into the various definitions
  - Give examples of formal/mathematical reasoning and “ways of thinking”

Definitions and $\leq$ vs. $<$

- It is tempting to think that
  - $f(n) \leq c\, g(n)$
  - and
  - $f(n) < c\, g(n)$
  - would give very different definitions,
  - and so, for example, would explain the big difference between O an o.
- But how much difference does it really make?
  - Strategy: try different definitions and see when they differ:

Other definitions of big-Oh?

Suppose that we had an alternative definition "$O_<$" that used

$$f \text{ is } O_<(g) \text{ iff } \exists\, c > 0, n_0. \forall n \geq n_0.\ f(n) < c\ g(n)$$

instead of usual O, which here we write as "$O_\leq$"

$$f \text{ is } O_\leq(g) \text{ iff } \exists\, c > 0, n_0. \forall n \geq n_0.\ f(n) \leq c\ g(n)$$

Are there pairs f,g for which this makes a difference?

Other definitions of big-Oh?

It is trivial that

$f\ is\ O_{<}(g)$ implies $f\ is\ O_{\leq}(g)$

For the converse: assume that we are given $f\ is\ O_{\leq}(g)$ then we try to show f is also the O<(g). That is:

We know

$\exists c > 0. \ldots f(n) \leq c\ g(n)$

but can we deduce

$\exists c' > 0. \ldots f(n) < c'\ g(n)$

Hint: When does $0 \leq x \leq y$ not imply $x < 2y$?

Other definitions of big-Oh?

So the two potential definitions only differ
When does $0 \leq x \leq y$ not imply $x < 2y$?

If y > 0, then 2y – y = y > 0, so 2y > y.

So if g(n) > 0, and c > 0, then c g(n) > 0,

and so 2 c g(n) > c g(n).

So if $c > 0$ and $g(n) > 0$, and we are given
$f(n) \leq c\, g(n)$
then $f(n) \leq c\, g(n) < (2c)\, g(n)$

Which means, when g>0,
we can prove f(n) is $O_{<}(g(n))$ using c′ = 2c.

Other definitions of big-Oh?

When g>0, we can prove
f(n) is $O_{<}(g(n))$ iff f(n) is $O_{\leq}(g(n))$

So unless g(n)=0 (i.e. the vast majority of time) it makes no difference which of "<" of "<=" is used in the definition!

What about when g(n)=0?

Do we prefer, or not, to be able to say that

0 is O(0) "zero is big Oh of zero" ?

Other definitions of big-Oh?

Do we want, or not, to be able to say that

0 is O(0) "zero is big Oh of zero" ?

With big-Oh, we are used to it being reflexive, or to be able to say "f is big-Oh of f".

We don't want to constantly want to add the caveat "unless f is zero".

Hence, we want a definition in which "zero is big Oh of zero"

Hence, we use "$f(n) \leq c\ g(n)$" in the definition.

Structure of the argument

- We took two alternative definitions, found that a lot of the time they agreed
  - But then one of the definitions was better on the "corner cases" – the properties were more consistent
  - Hence, we select the definition that is more general.
- This sort of "argument structure" is common, but "behind the scenes"
  - Imagine being on a "standard committee for CS, and had to invent definitions for common usage" – there is no authority to ask the answer!

Other definitions of little-oh?

Suppose that we had an alternative definition "$o_{\leq}$" that used

$$f \text{ is } o_{\leq}(g) \text{ iff } \forall\, c > 0, n_0. \forall n \geq n_0.\ f(n) \leq c\ g(n)$$

instead of usual o, which here we write as "$o_{<}$"

$$f \text{ is } o_{<}(g) \text{ iff } \forall\, c > 0, n_0. \forall n \geq n_0.\ f(n) < c\ g(n)$$

Are there pairs f,g for which this makes a difference?

Other definitions of little-oh?

Similarly to the big-Oh we can show that if g(n) > 0, then

$f\ is\ o_{\leq}(g)$ iff $f\ is\ o_{<}(g)$

and so we then have a choice of what to do on the zero function $zero(n)$

Do we prefer, or not, to be able to say

$zero(n)$ is $o(zero(n))$

In this case, generally "f is not o of f".
So we do not want this.
So we can use "<" in the little-oh definition to prevent it.
And maybe "<" is more natural ?

Rationale of definitions

- We use
  - "<=" in Big-Oh
  - "<" in little-oh
- definitions to give wider consistency.
- But this difference is not the most important part of the difference.

The vital difference between O and o is the difference between

$$\exists\, c > 0 \quad \text{and} \quad \forall\, c > 0$$

COMP2009-ACE:

ADE Lec05

Rules for little-oh proofs and other advanced usages

Lecturer: Andrew Parkes
andrew.parkes@Nottingham.ac.uk

from http://www.cs.nott.ac.uk/~pszajp/

NOTES

- These slides are meant for advanced understanding.
  - Some parts have been stated elsewhere
  - The rules are methods for little-oh that might be useful. However, if a question asks for “from the definition” then such rules should not be used.

“Multiplication Rule” for big-Oh * little-Oh ?

Suppose a, c, d are all strictly positive numbers

If $c < d$

then we can multiply by a to get

$a*c < a*d$

Given the mnemonic “<=“ ~ “Big-Oh” and “<“ ~ “little-oh”, it is reasonable to expect to be able to multiply

“a function, and an o to get an o”.

But is this actually correct?

"Multiplication Rule" for function*little-Oh

- Suppose h(n) is (strictly) positive, and

  f(n) is o(g(n)) [[little-oh]]

- Then, from the definition, we know

  Forall c >0 exists n0 $f(n) < c\, g(n)$ for all $n \geq n_0$

- Then multiplying by h(n) gives

  Forall c > 0 exists n0. $h(n)\, f(n) < c\, h(n)\, g(n)$ for all $n \geq n_0$

- Which is precisely: h(n) f(n) is o(h(n) g(n))

- E.g. can multiply "1 is o(n)" by h(n)=n to get "n is $o(n^2)$"

- (Note: for brevity, we may suppress extra conditions, such as needing h(n) > 0, as they will be obvious.)

“Multiplication Rule” for big-Oh * little-Oh ?

Suppose a, b, c, d are all strictly positive numbers

If $a <= b$
and $c < d$

then we can multiply to get

$a*c < b*d$

Given the mnemonic “<=“ ~ “Big-Oh” and “<“ ~ “little-oh”, it is reasonable to expect to be able to multiply “O and o to get an o”.

But is this actually correct?

"Multiplication Rule" for big-Oh * little-Oh

- Suppose
  - $f_1(n)$ is $O(\ g_1(n)\ )$ [[Big-Oh]]
  - $f_2(n)$ is $o(\ g_2(n)\ )$ [[little-oh]]
- Then, from the definition, there exist positive constants $c_1>0$, n_1 such that
  1. Exists c1 >0 exists n1 $f_1(n) \le c_1\ g_1(n)$ for all $n \ge n_1$
  2. Forall c2 >0 exists n2 $f_2(n) < c_2\ g_2(n)$ for all $n \ge n_2$
- Then multiplying gives
  - Exists c1 forall c2. exists n1 exists n2, $n_0 = \max(n_1\ ,\ n_2)$,
    $f_1(n)\ f_2(n) < c_1\ c_2\ g_1(n)\ g_2(n)$ for all $n \ge n_0$
- But for any (strictly) positive value of c1, then "forall c2>0" captures the same as "forall c1*c2".
- Hence: $f_1(n)\ f_2(n)$ is $o(\ g_1(n)\ g_2(n)\ )$
- E.g. n is O(n), and 1 is o(log n), hence n is o(n log n)
- E.g. n is O(n), and log(n) is o(n), hence n log n is $o(n^2)$

Advanced Usages of Big-Oh

- **Some of the following slides on just a reminder of work covered before**
- They are included for
  - The ways of thinking about big-oh
  - Usages that might be seen in advanced academic literature, and in mathematical literature

Usages as sets of functions

Sometimes in (advanced theory) papers might see expressions such as

$$n^{O(1)}$$

If "big-Oh" only means "worst case of an algorithm" then this is nonsensical.

But if the big-Oh family are used to refer to sets of functions then we can take it to mean

$$\{ n^{f(n)} \mid f(n) \text{ is } O(1) \}$$

e.g. including f(n)=1, f(n)=2, f(n)=3, …. So $n^{O(1)}$ includes n^1, n^2, n^3, …

So $n^{O(1)}$ is interpreted as

"any function that is no worse than (Big-Oh of) some power law"

Usages as sets of functions

Another example can be seen in:
https://en.wikipedia.org/wiki/Stirling%27s_approximation

$$\ln(n!) = n \ln n - n + O(\ln(n))$$

Where "ln" is the "natural log" using the base e:
I.e. $\ln(y) = x$ iff $y = e^x$ (e=2.7128…)

It means that the extra term, which one might think of as the "error", is "some function that is O(ln n)"
It means that the error is small compared to the other two terms.

This approximation of n! is relevant to later lectures on "Bounds on Comparison based sorting".

Usages as sets of functions

$1 - o(1)$ would then mean a set of functions:

$\{ 1 - f(n) \mid f(n) \text{ is } o(1) \text{ (and } f(n) \text{ is positive) } \}$

Hence

forall c > 0. exists n0. s.t. forall n >= n0. 0 <= f(n) < c

That is:

However small we pick c, then eventually f(n), becomes, and stays, smaller than c.

This is the same meaning as

f(n) tends to zero as n goes to infinity.

So 1-o(1) is interpreted as "any function which approaches 1 (from below) in the limit of n becoming large" – e.g. $(1 - 1/n)$

COMP2054-ADE

ADE Lec06
Linked Lists

Lecturer: Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Note

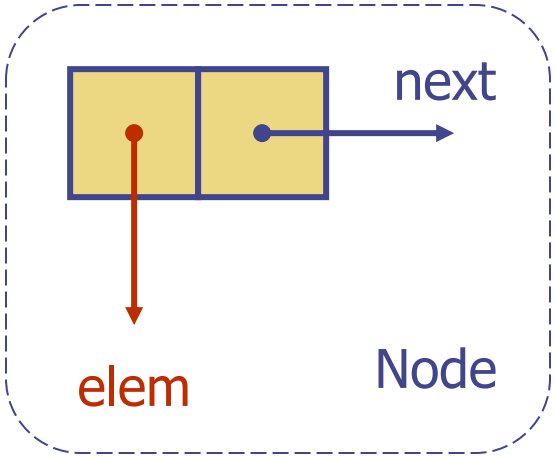
- This short set of slides is just a quick introduction to linked lists, and not everything you need to know.
- Hopefully, most of it is revision from 1st year (or before).

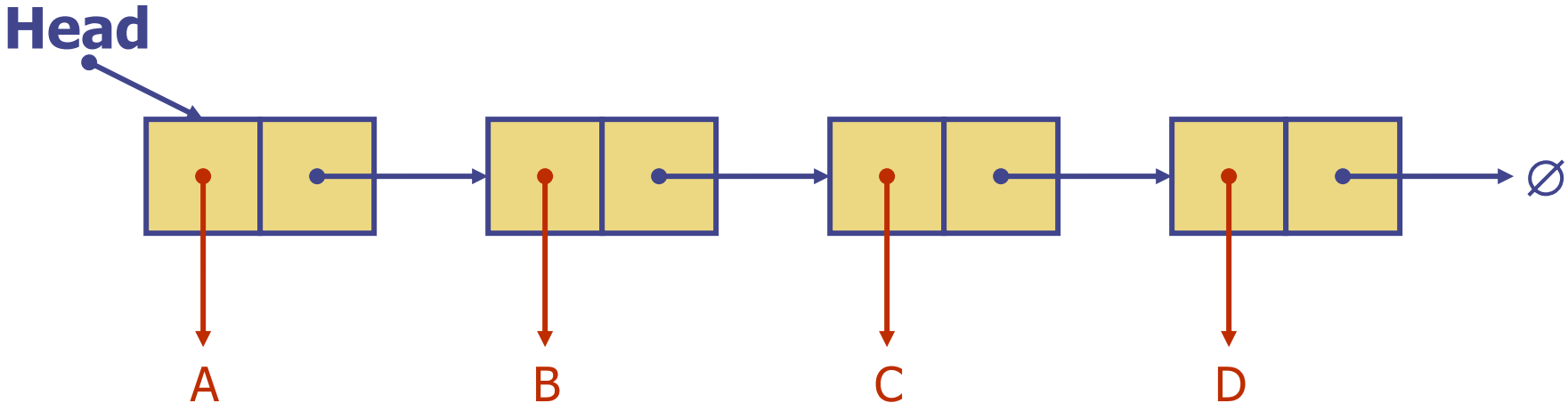
Relevance

- It is generally assumed that you will have done singly and doubly linked lists
  - Please revise them, if not then these slides just cover the basic ideas
- Our goal in this module is to be more careful about their efficiency, that is, to observe the complexity, O(1) versus O(n), of the various standard operations
  - Where 'n' is the length of the list
- Also, they are relevant to "simple sorting" algorithms, and also to "stacks" and "queues"

Recap: Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element e.g.
    - Reference to an Object
    - A primitive date type (int,…)
  - "link": a reference (pointer) to the next node





A Node Class for List Nodes

The relevant code usually looks like:

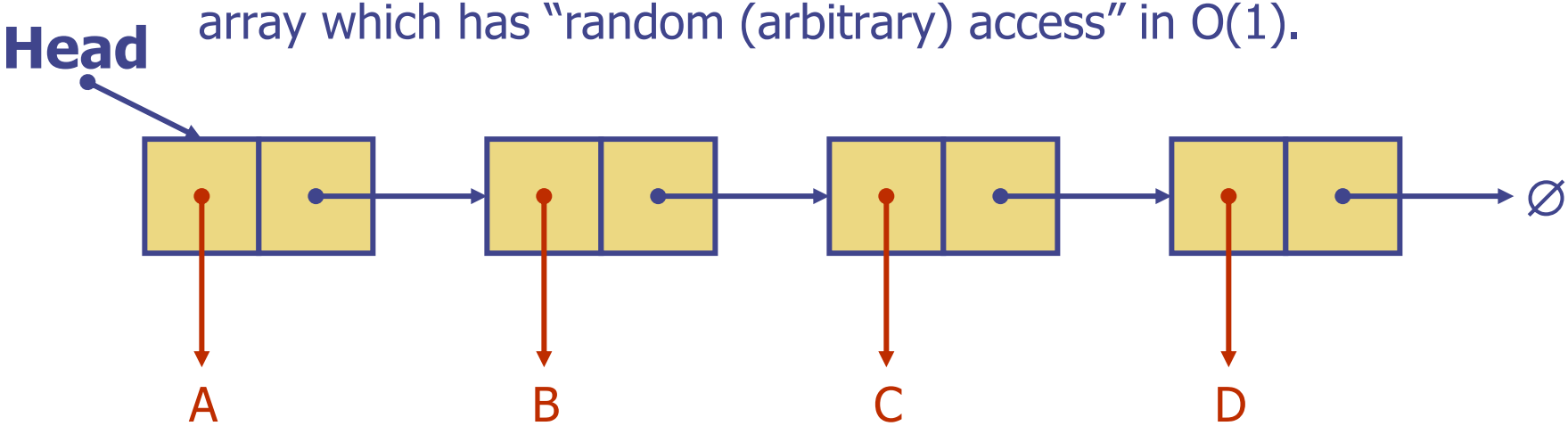
```
public class Node  {
    // Instance variables:
    private  Object element;
    private  Node next; // reference ("pointer") to a 'Node'

    /** Creates a node with the given element and next
    node. */
    public  Node(Object e,  Node n)  {
        element  =  e;
        next  =  n;
  }
```

The Node Class for List Nodes

```
// Accessor methods:
 public  Object getElement()  {
    return  element;
}
 public  Node getNext()  {
    return  next;
}
 // Modifier methods:
 public void  setElement(Object newElem)  {
      element  =  newElem;
}
 public void  setNext(Node newNext)  {
      next  =  newNext;
}
} // end of class Node
```

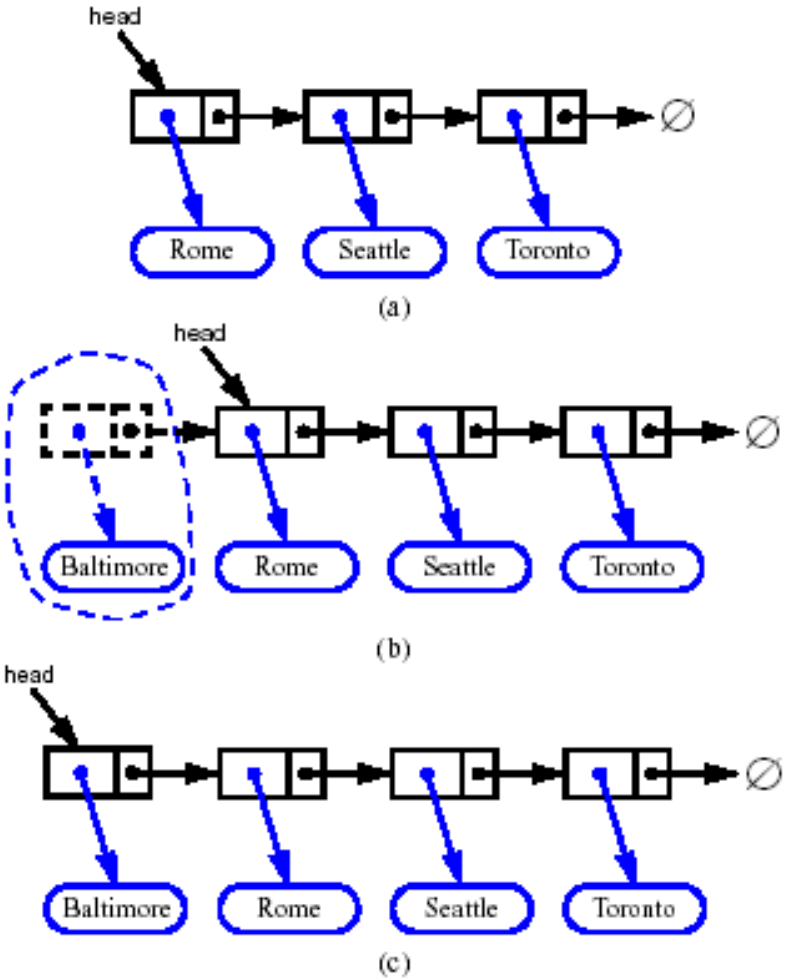
Usage?

- In a simple linked list, all the data is accessible from the head by just "walking along the list"
  - Hence, it is clear (hopefully) that all "standard" operations (insert/delete etc) are implementable
- The key question is what operations are implementable **efficiently**!?
  - Need O(.) of operations in terms of the length n of the list
    - Note: Typically the overall length of a list (or other data structure) is stored in an auxiliary element to allow fast access as it is generally used a lot.
  - Note: There is no direct access to the middle of a list - unlike an array which has "random (arbitrary) access" in O(1).



Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



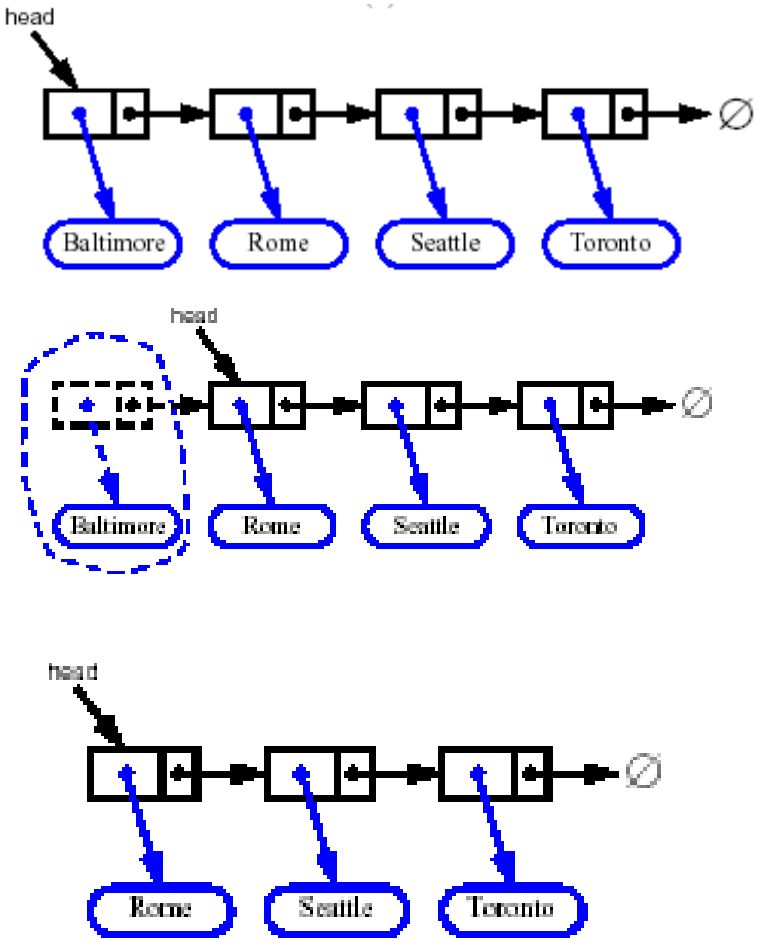
What is the complexity (with n elements in list)?

- Answer: O(1)
- Very efficient!

Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node
   1. Or do explicit free in C/C++

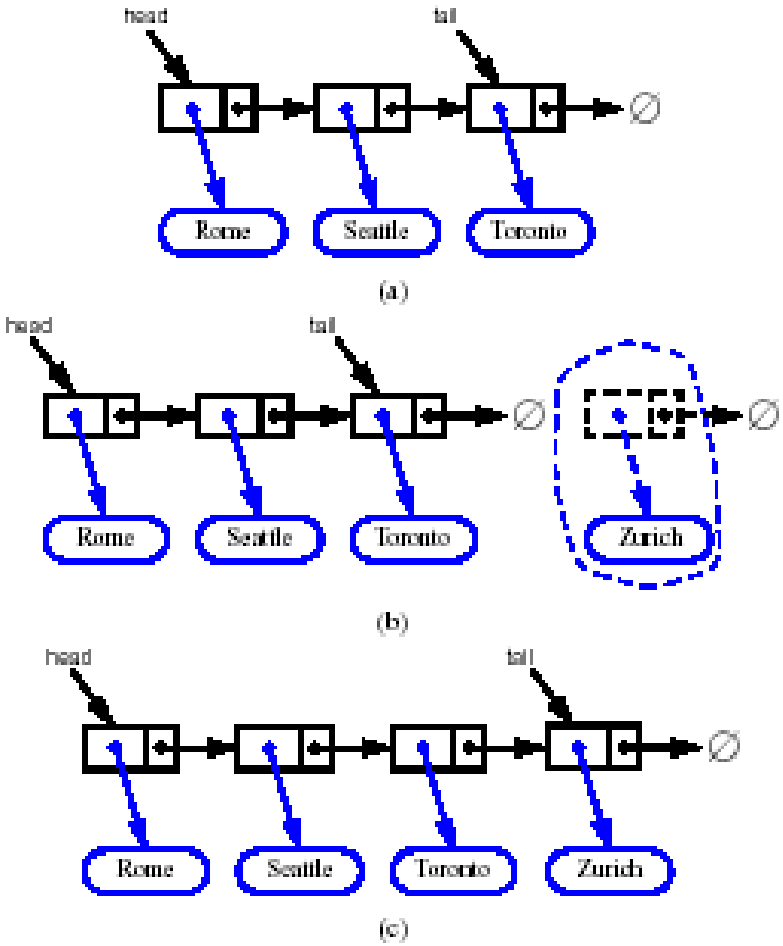
Again, the operation is O(1), and so efficient.



Inserting at the Tail

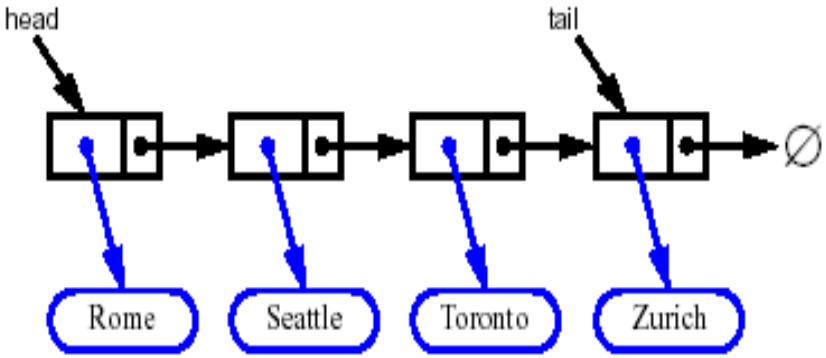
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

Complexity: O(1)



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- To find new tail we must walk the list from the head
  - There is no constant-time way to update the tail to point to the previous node
  - Exercise: Why not keep a "pre-tail" pointing to one before the "tail"? ("Toronto" here)
  - Advanced optional: look up "skip lists"
- Complexity: O(n)

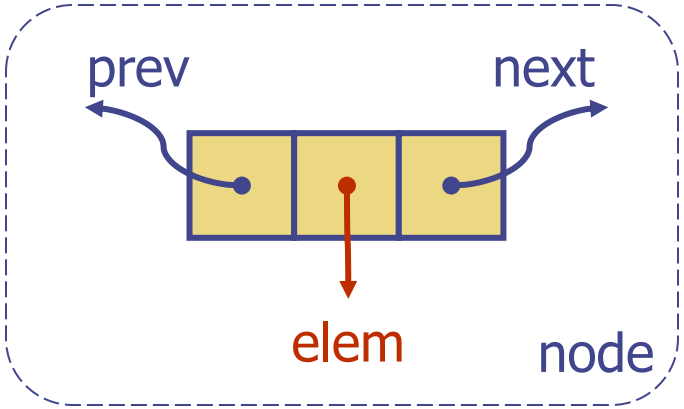


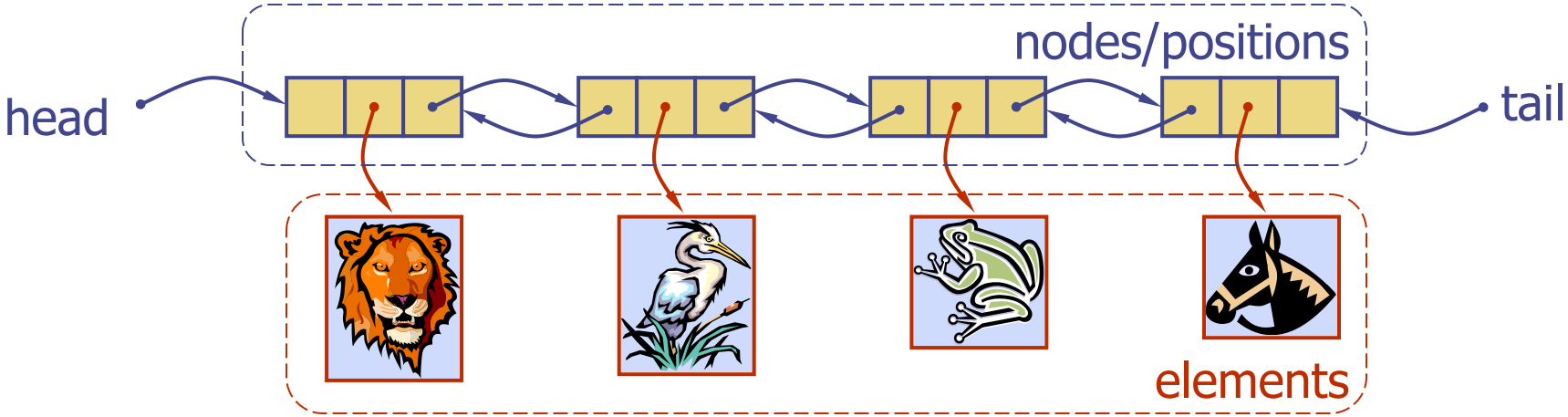
A SinglyLinkedList Class

```
class SinglyLinkedList {
    private Node head;

    public SinglyLinkedList() {
    } // head automatically set to null by Java

    public void insertAtHead(Object newElem)  {
        Node newHead = new Node(newElem,head);
        head = newHead;
    }
// etc
}
```

Doubly Linked List

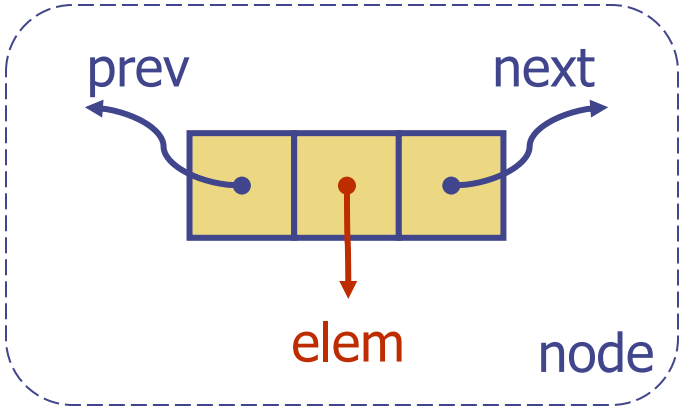
- A doubly linked list provides a natural extension of a singly linked list
- Nodes store:
  - element
  - link to the next node
  - **link to the previous node**
- **Deletion at the tail is now O(1)**
- **But uses more memory**

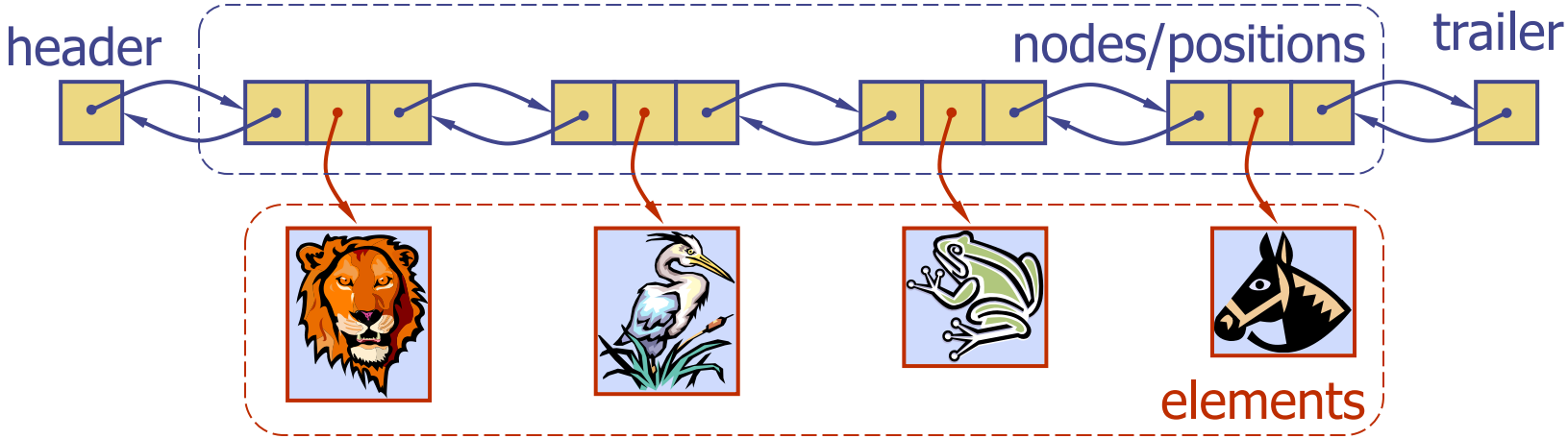




Doubly Linked List (version 2)

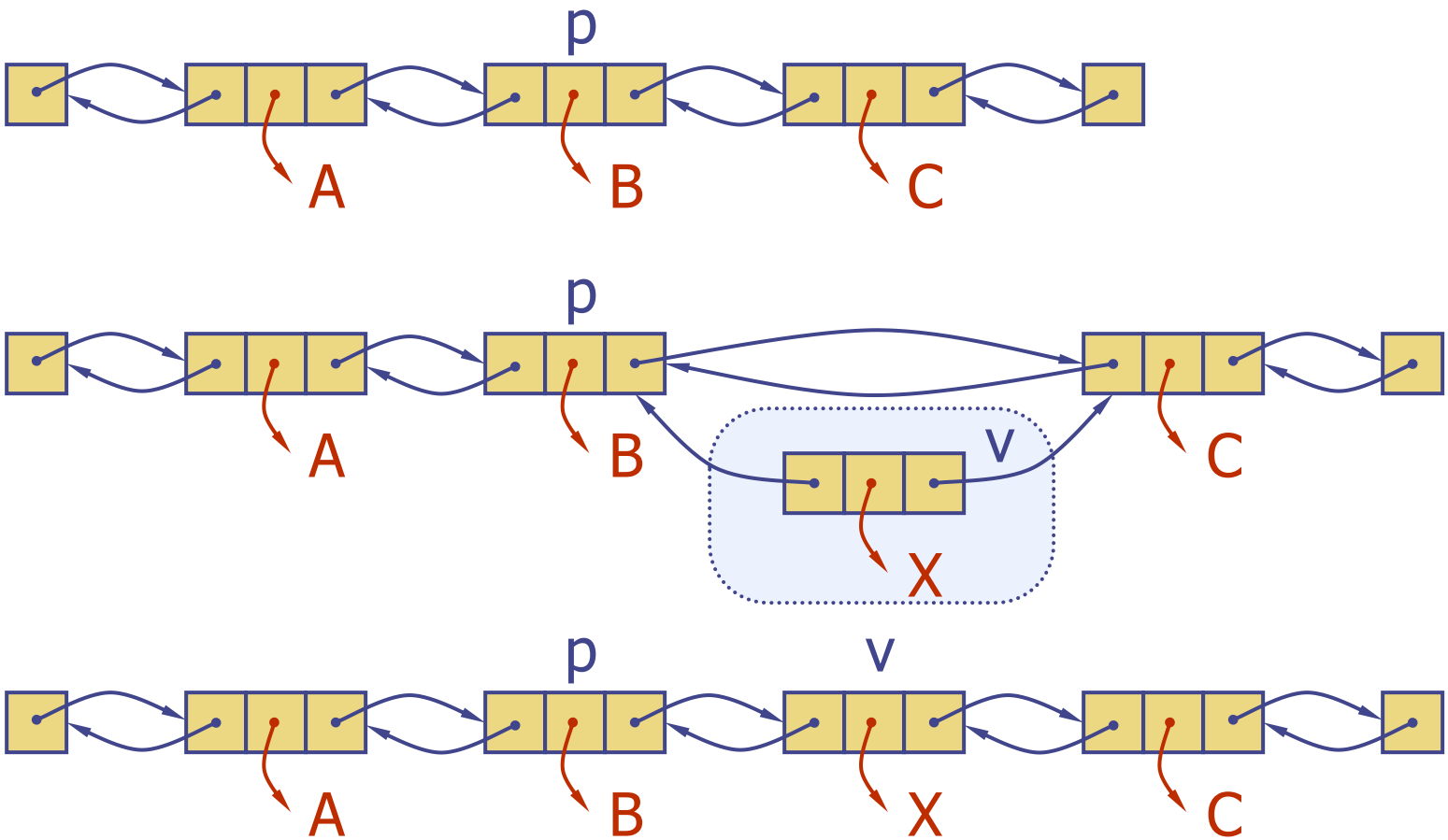
- A doubly linked list provides a natural implementation of a List
- Nodes implement "Position" and store:
  - element
  - **link to the previous node**
  - link to the next node
- Special trailer and header nodes for convenience (an alternative)





Insertion

- We visualize O(1) operation addAfter(p, X), which returns position v



Insertion Algorithm

Algorithm addAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

v.setPrev(*p*) {link *v* to its predecessor}

v.setNext(*p*.getNext()) {link *v* to its successor}

(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}

p.setNext(*v*) {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}

Minimum Expectations

- It is generally assumed that you will be familiar with singly and doubly linked lists
- You should be aware of the complexity, O(1) versus O(n), of the various standard operations, and how to implement them.

COMP2054-ADE

ADE Lec07

Simple Sorting Algorithms

Lecturer: Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Simple sorting algorithms and their complexity

Consider an array of integers, and the goal is to sort into non-decreasing order (put the largest on the right)

1. Bubble sort
2. Selection sort
3. Insertion sort

Bubble Sort: Basic Idea

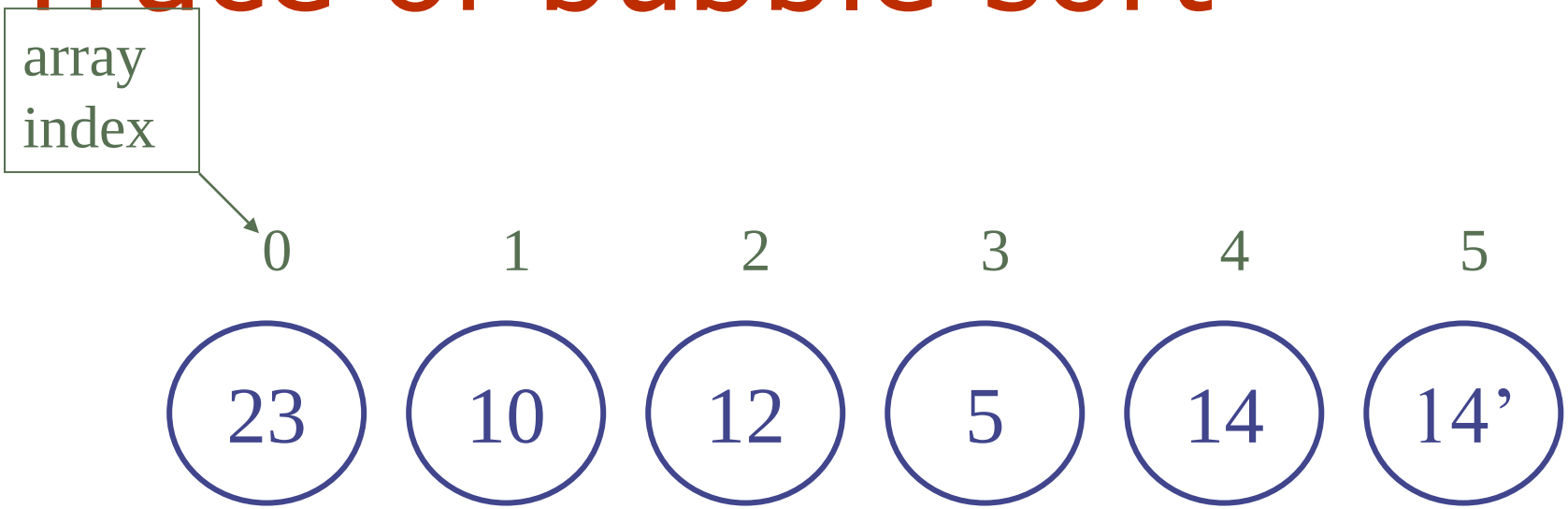
- Outer loop:
  Repeated scans through array
- Inner loop: on each scan do comparison with immediate neighbour
  - think of air bubbles rising in water
  - do swaps to make sure that the largest number "bubbles up" to the end of the array

Bubble sort

```
void  bubbleSort(int arr[]){
    int i;
    int j;
    int temp;
    for(i = arr.length-1; i > 0; i--){
        for(j = 0; j < i; j++){
            if(arr[j] > arr[j+1]){
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }//
        }// end inner loop
    }//end outer loop}// end bubble sort
```

swap adjacent elements, if in the wrong order

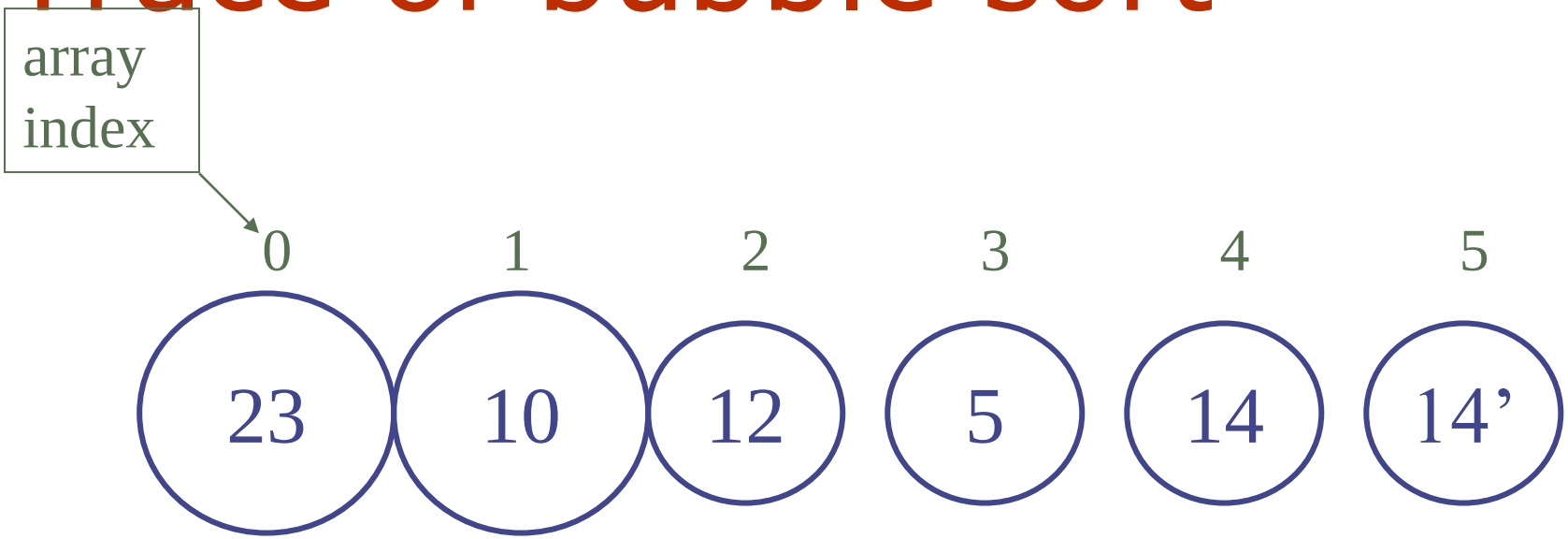
Trace of bubble sort



i = 5, first iteration of the outer loop

“Stability”: 14 and 14’ are two copies of the same number but we keep track of which copy ends up where. (Explained why later)

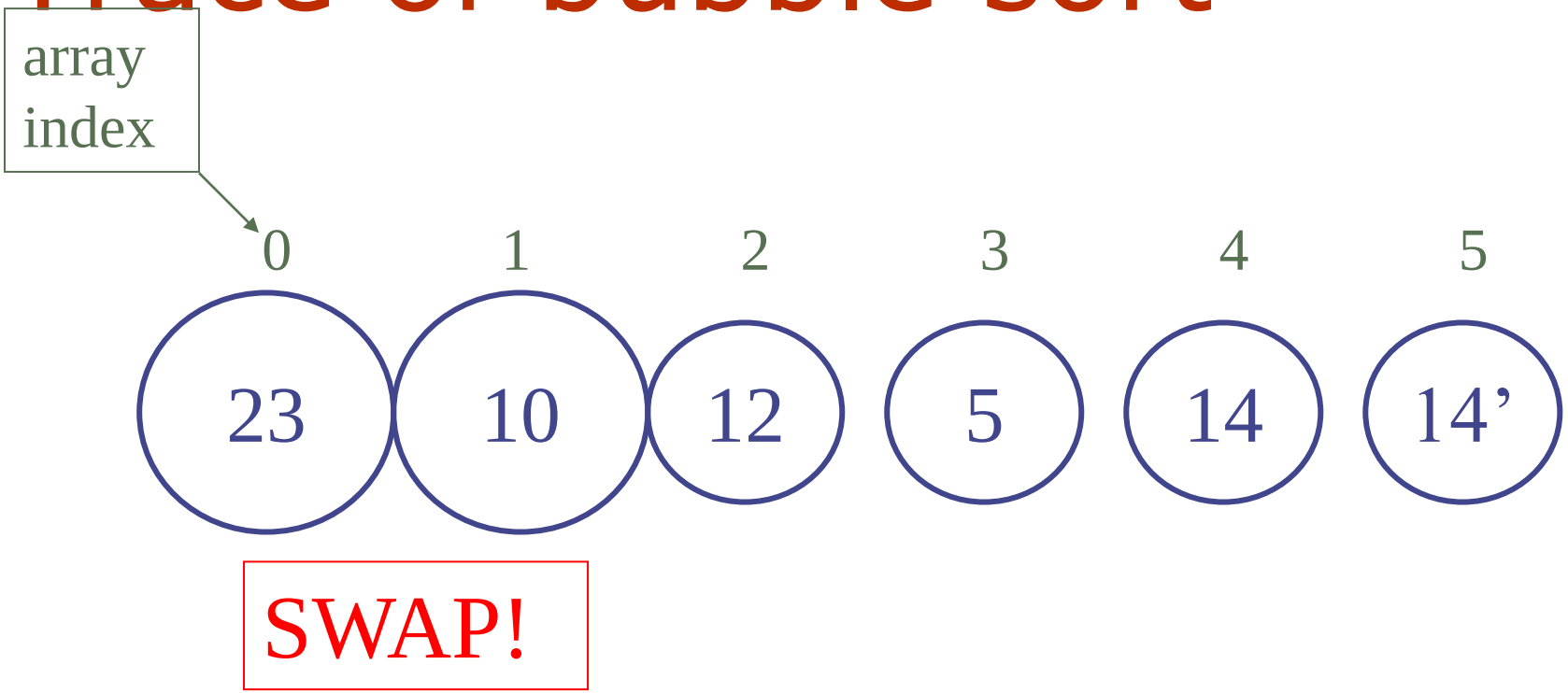
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

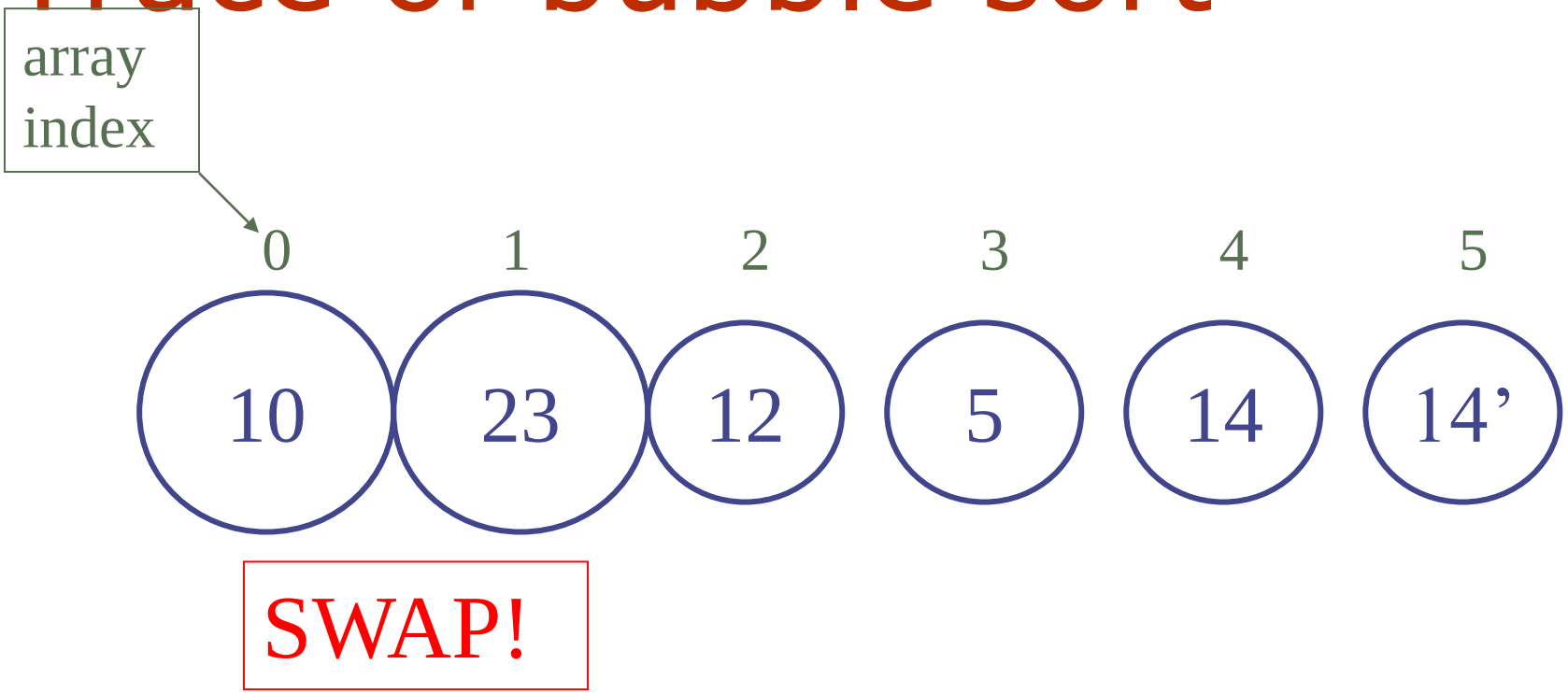
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

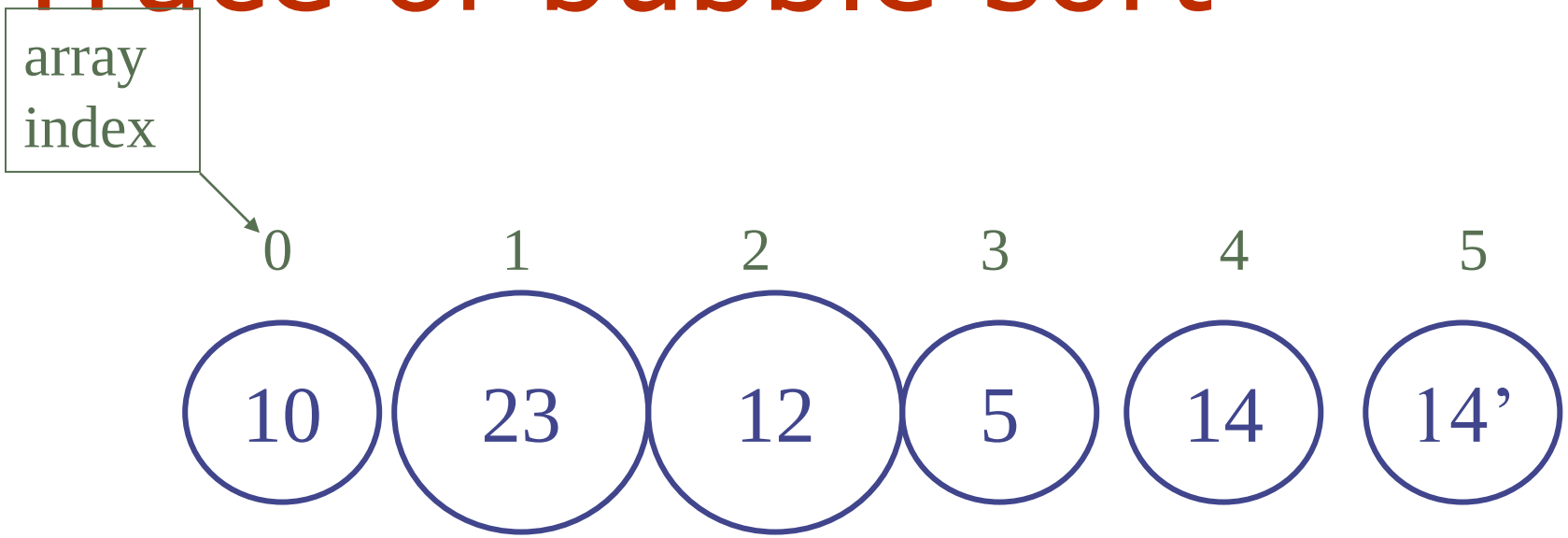
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

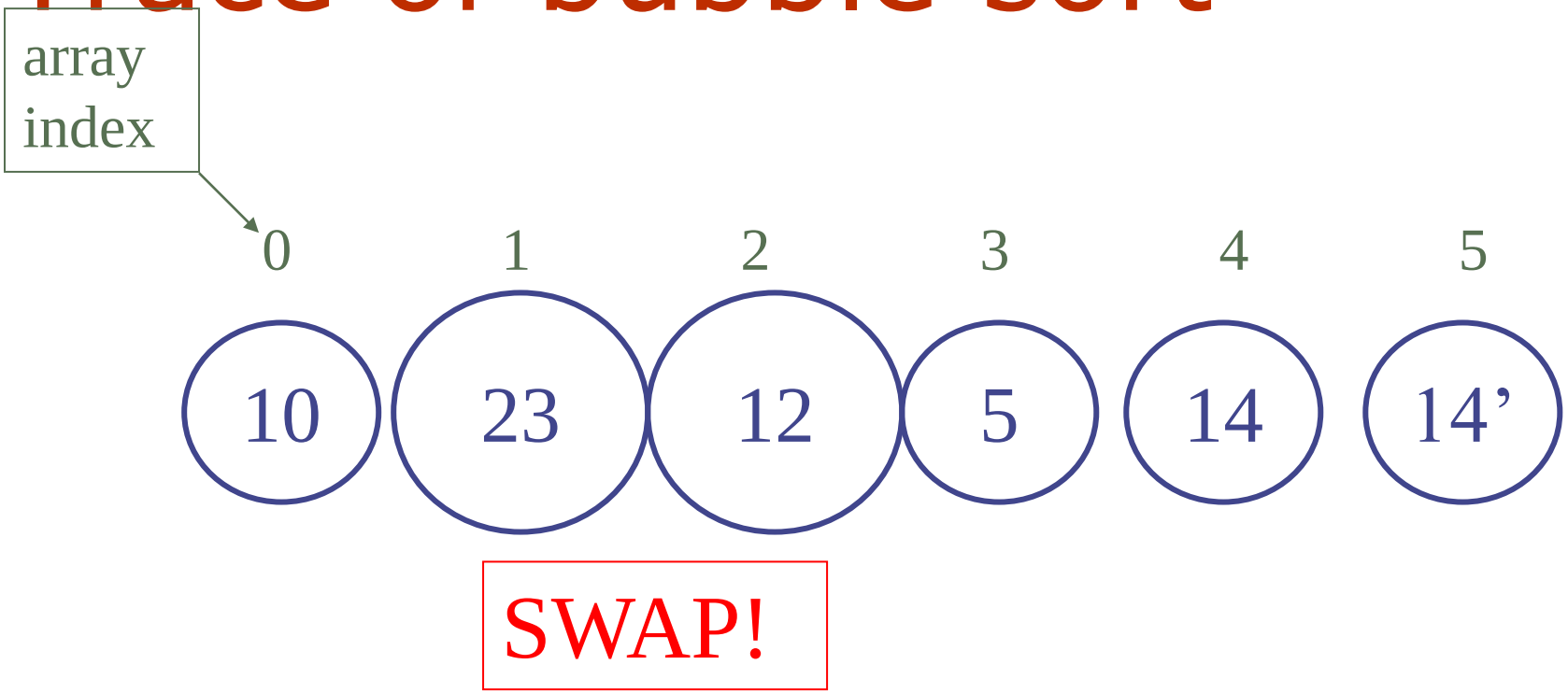
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

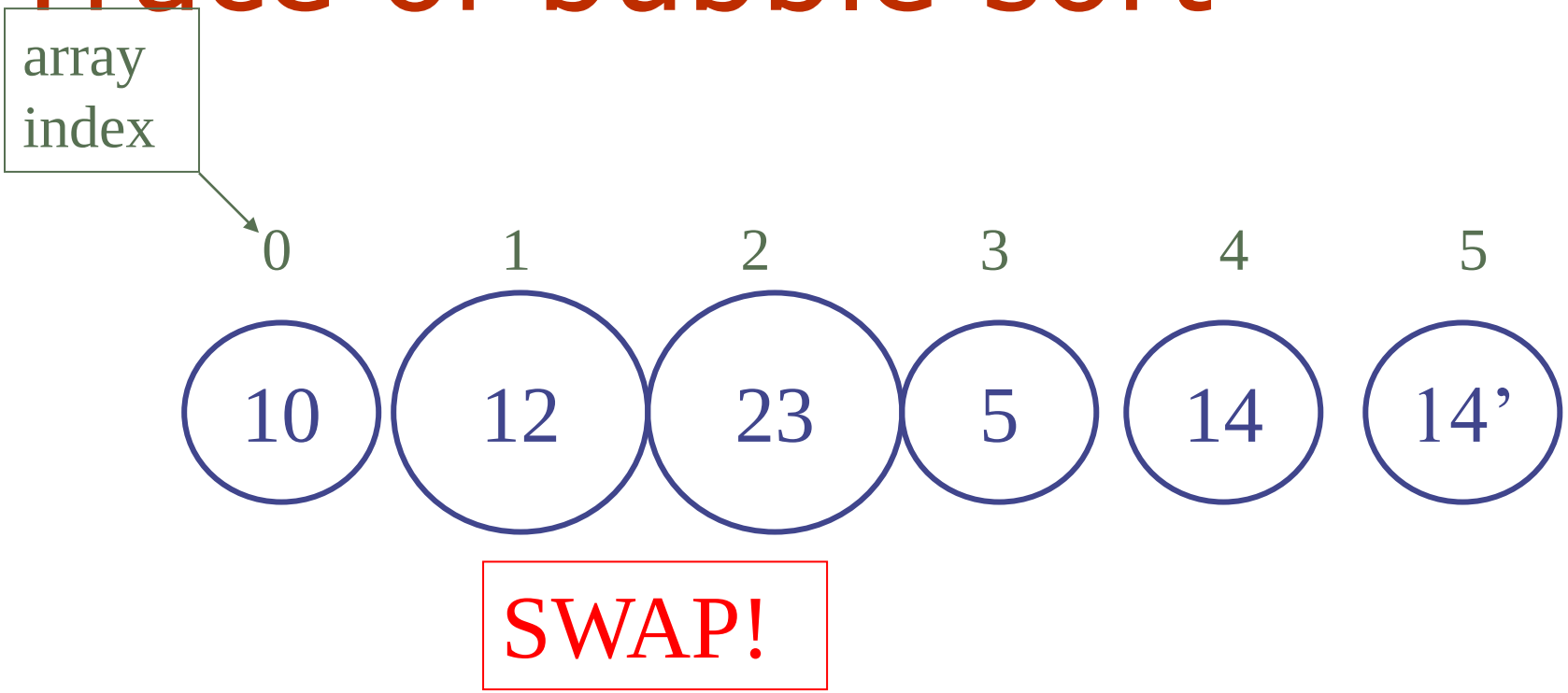
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

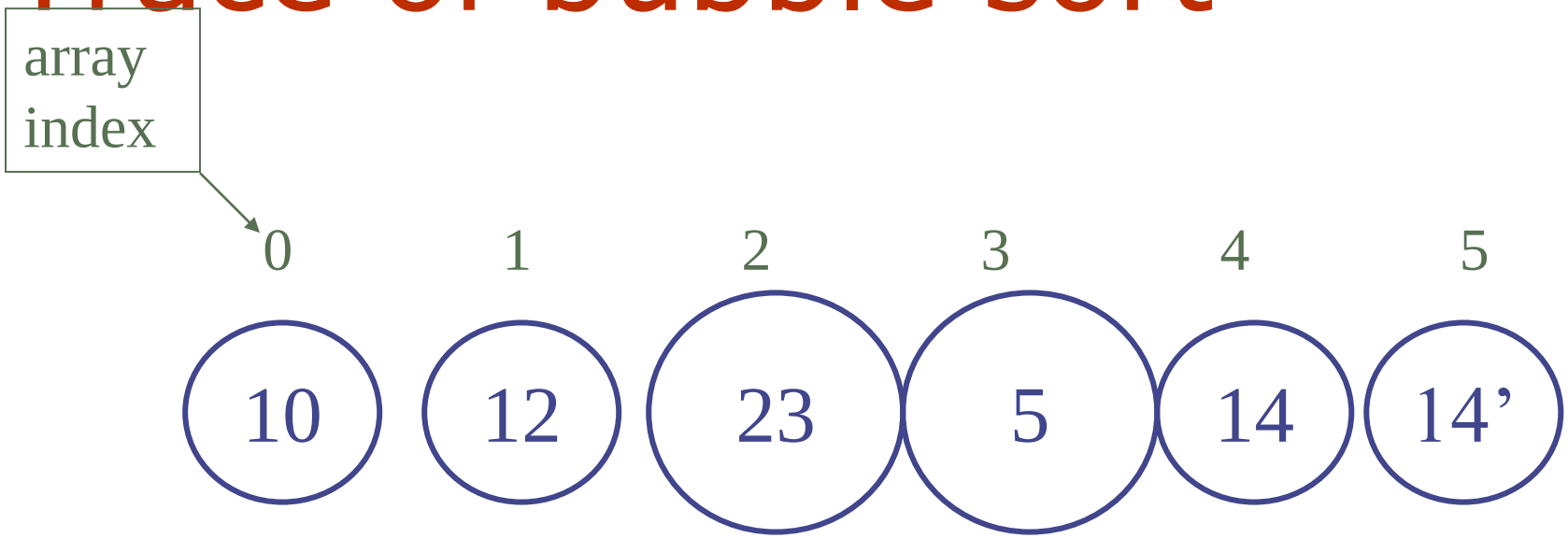
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

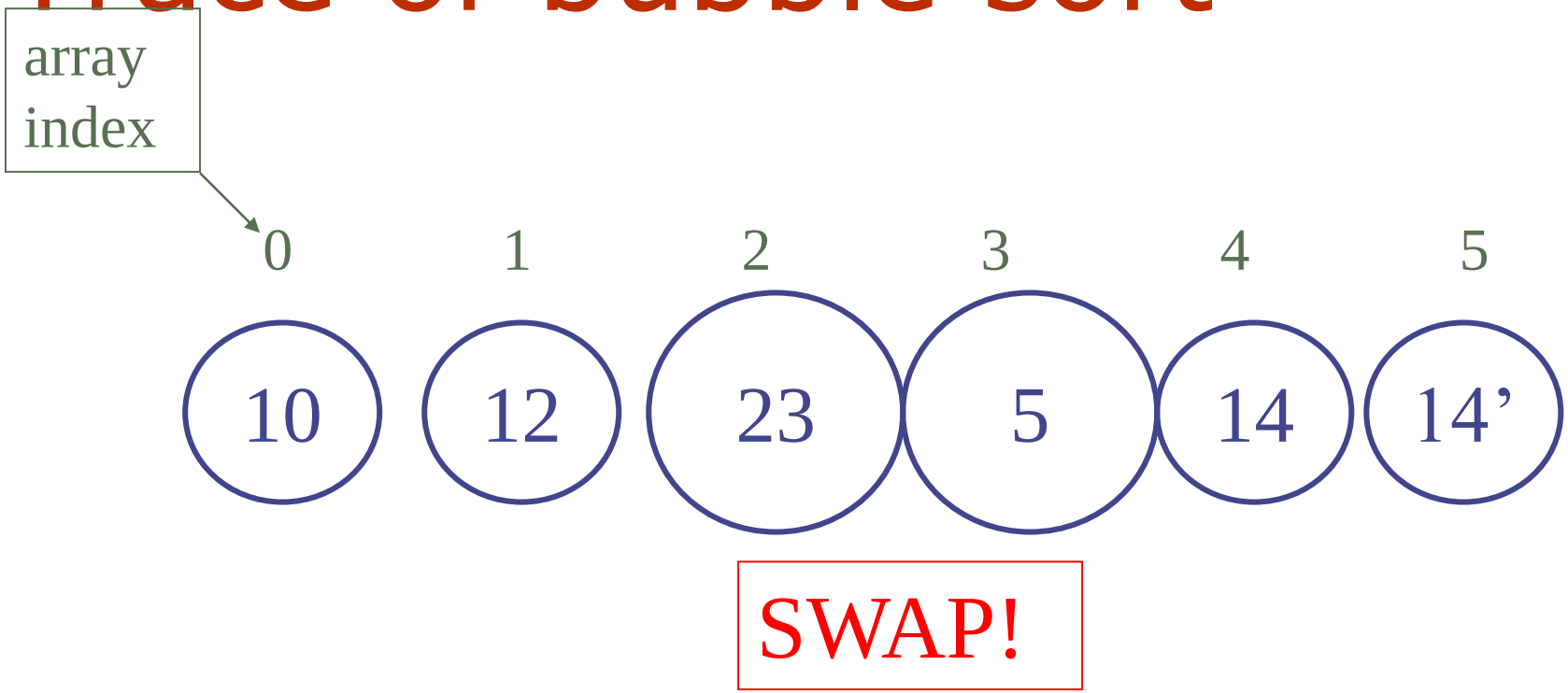
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

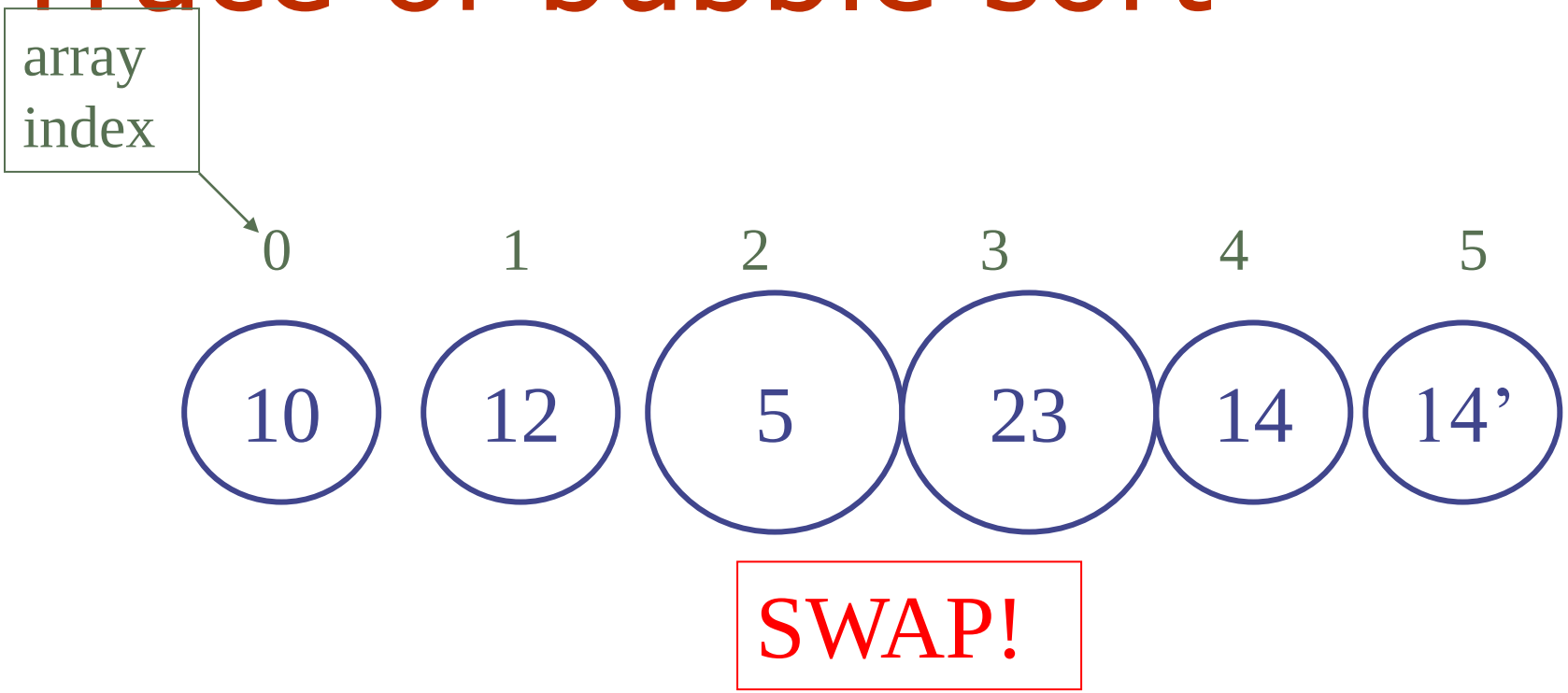
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

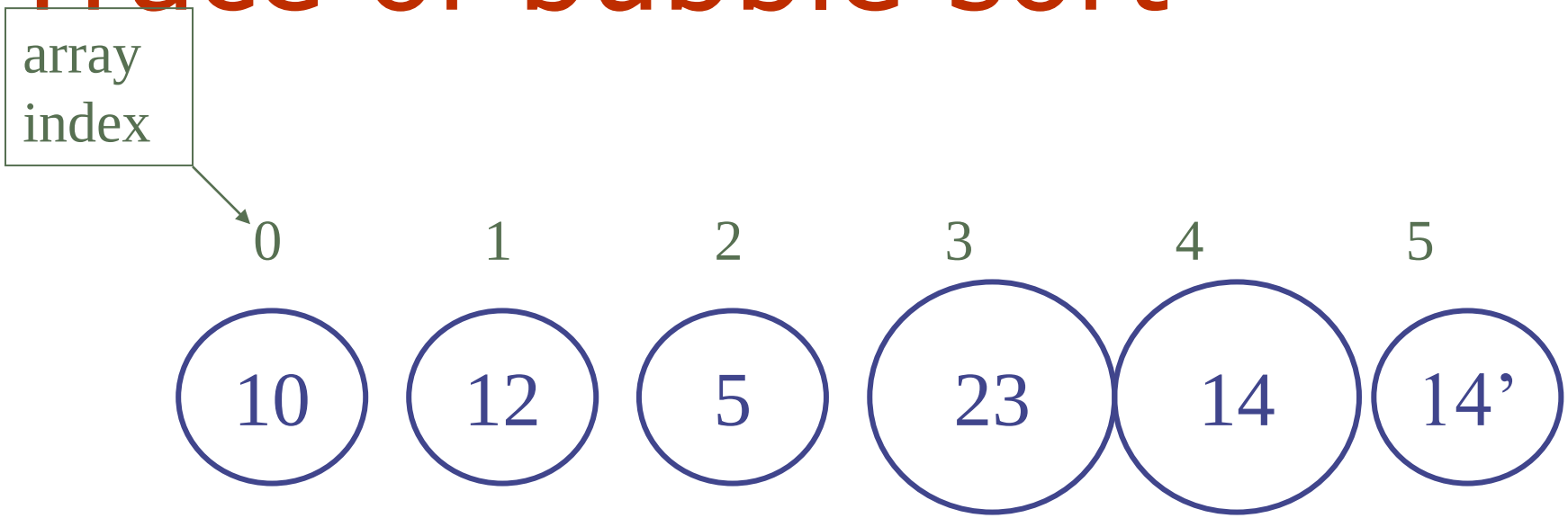
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

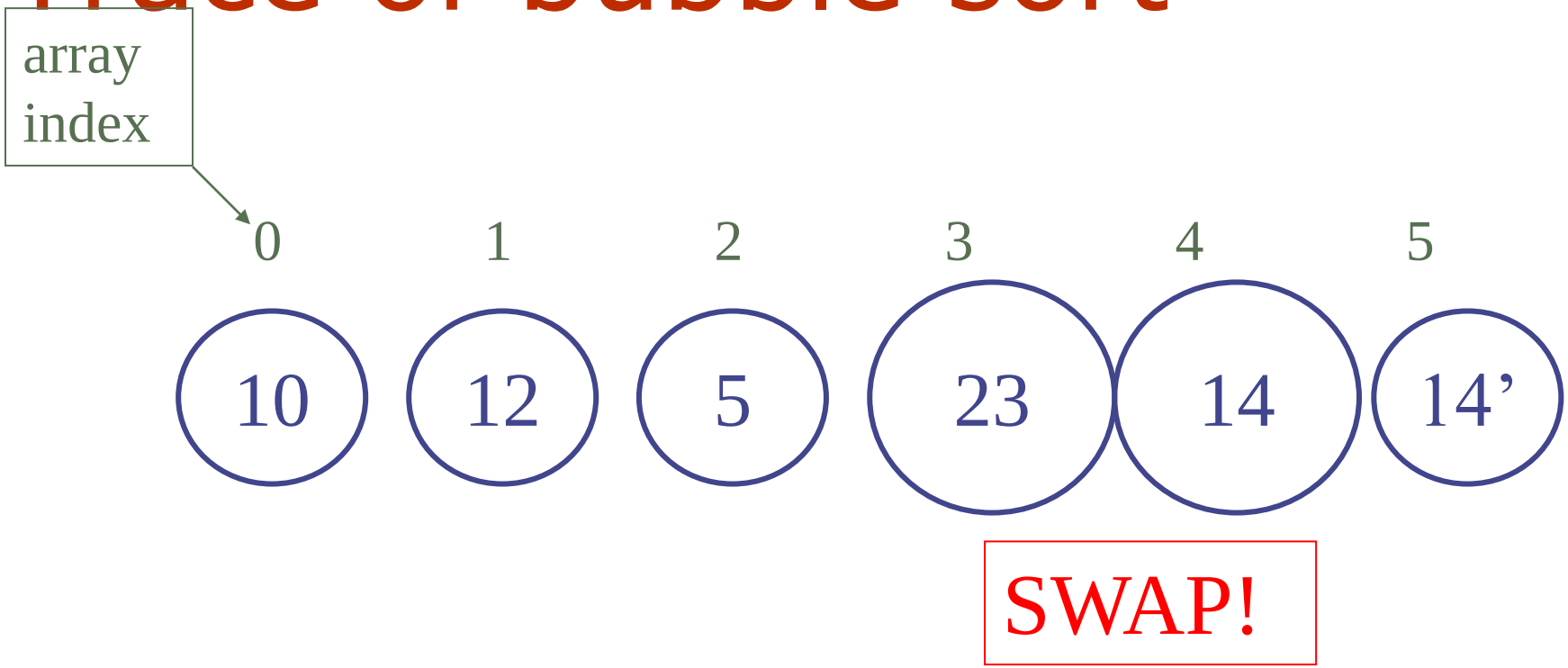
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

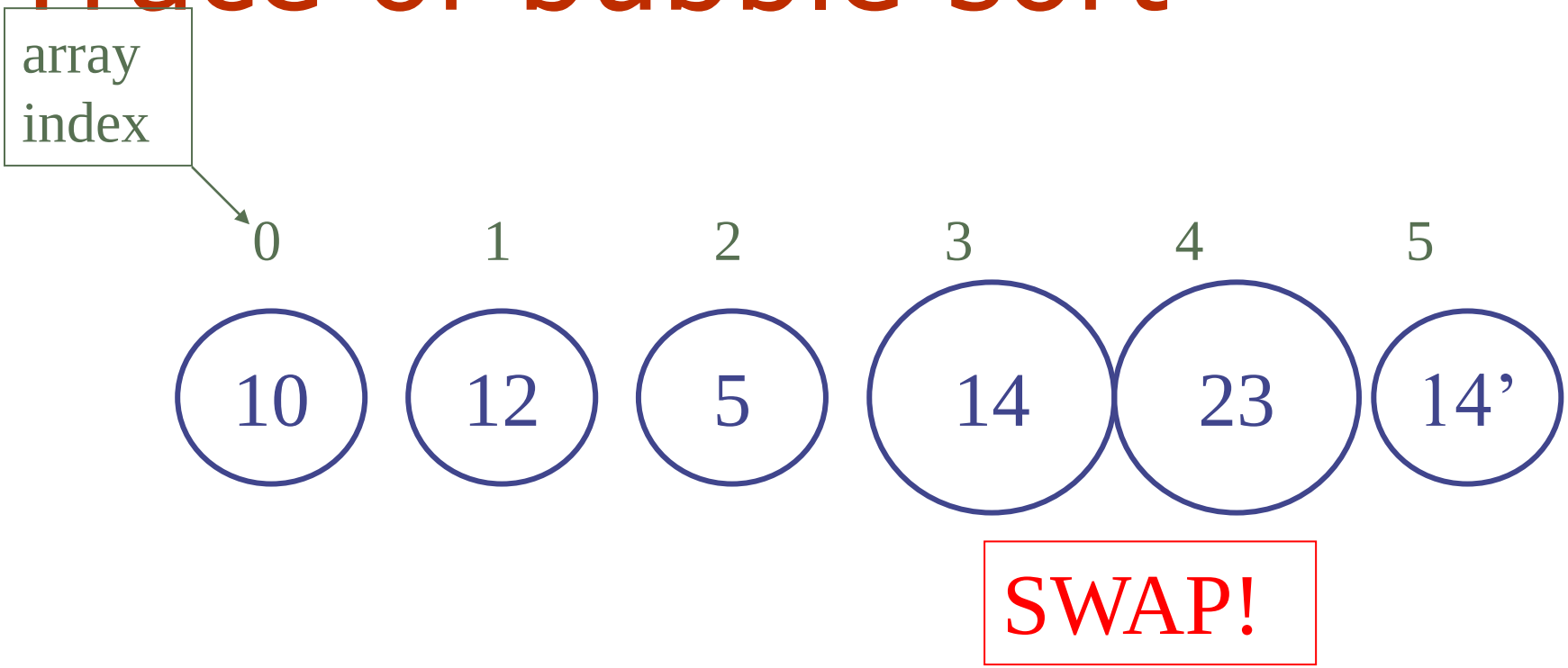
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

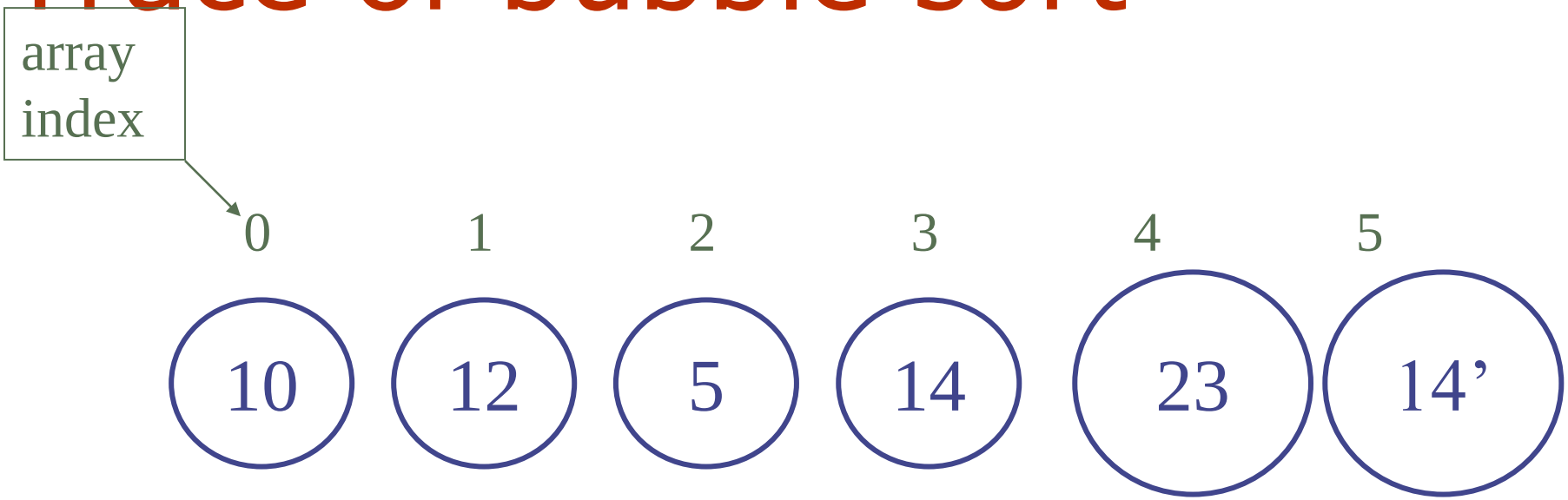
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

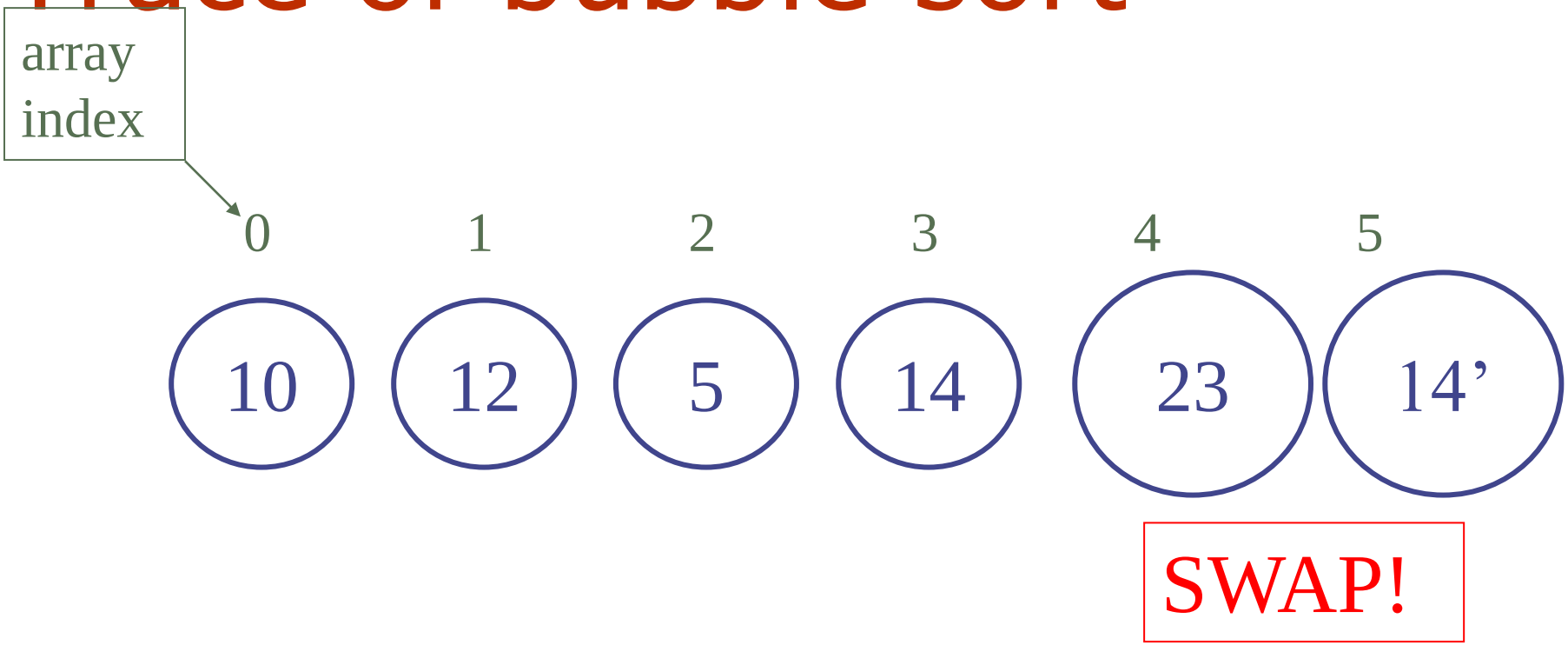
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

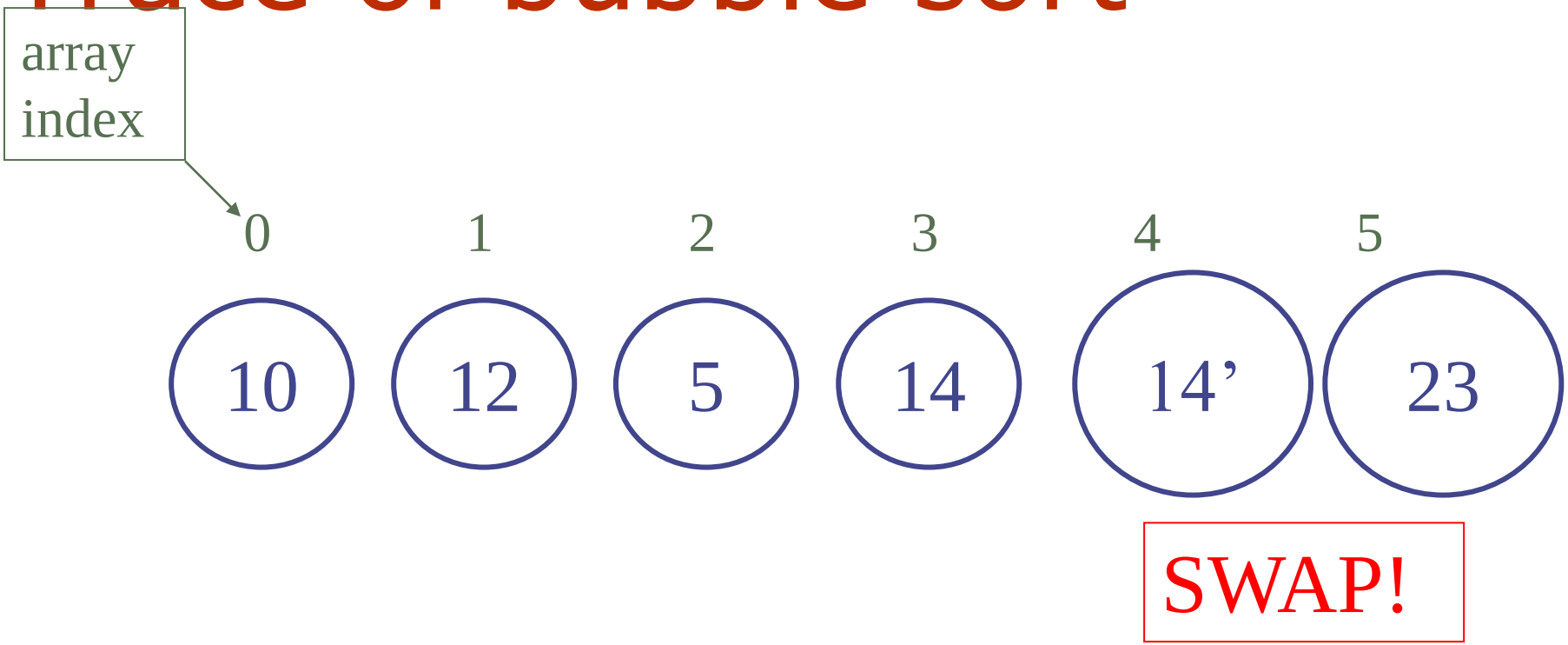
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

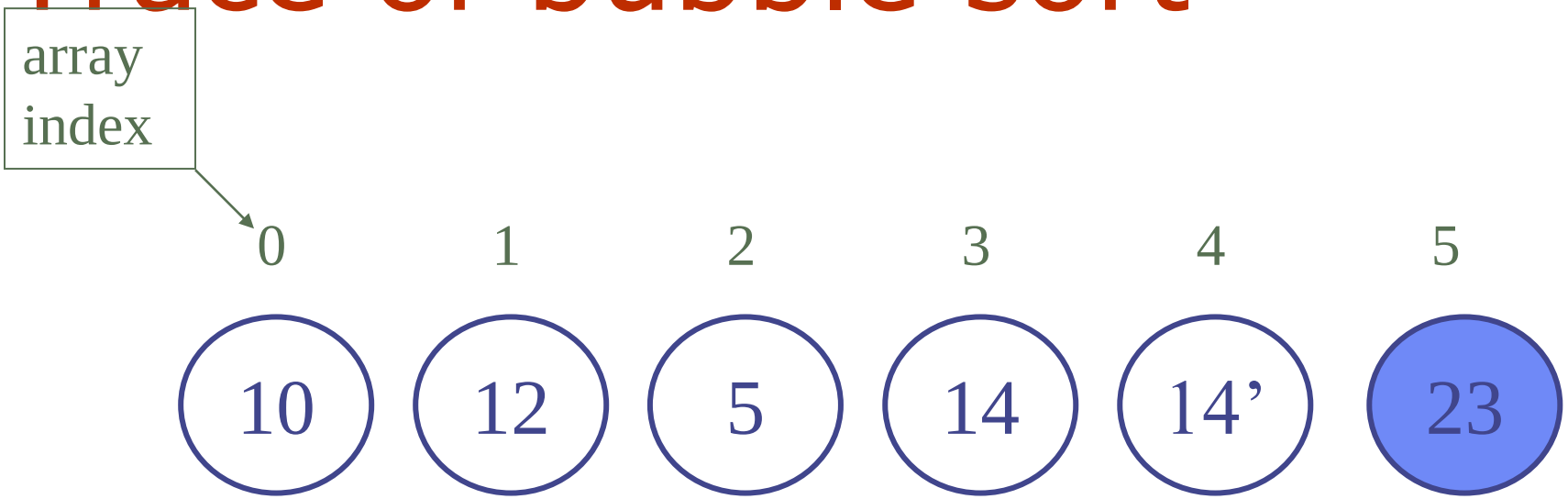
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

Trace of bubble sort



i = 5, first iteration of the outer loop

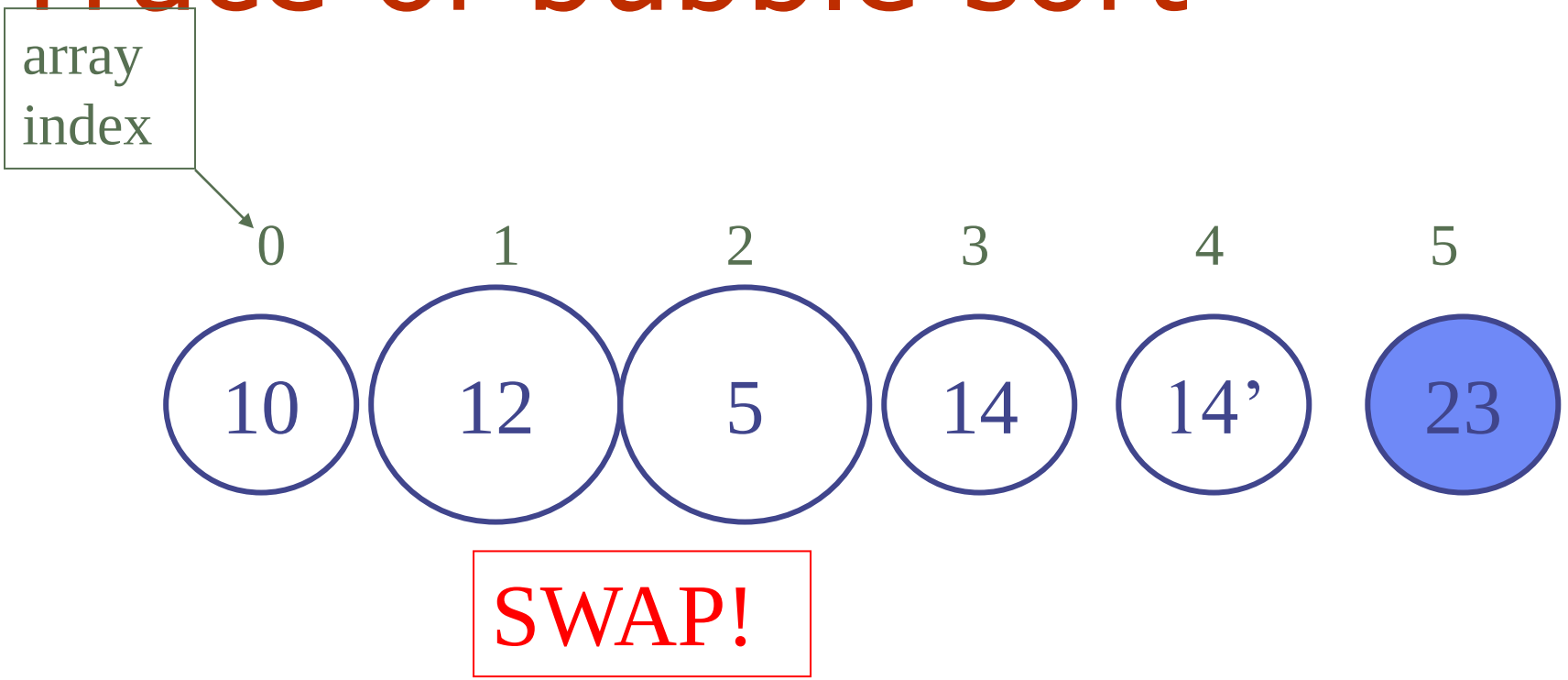
inner loop finished; largest element in position 5, positions 0-4 unsorted

Trace of bubble sort

i = 4, second iteration of the outer loop

j = 0, comparing arr[0] with arr[1]

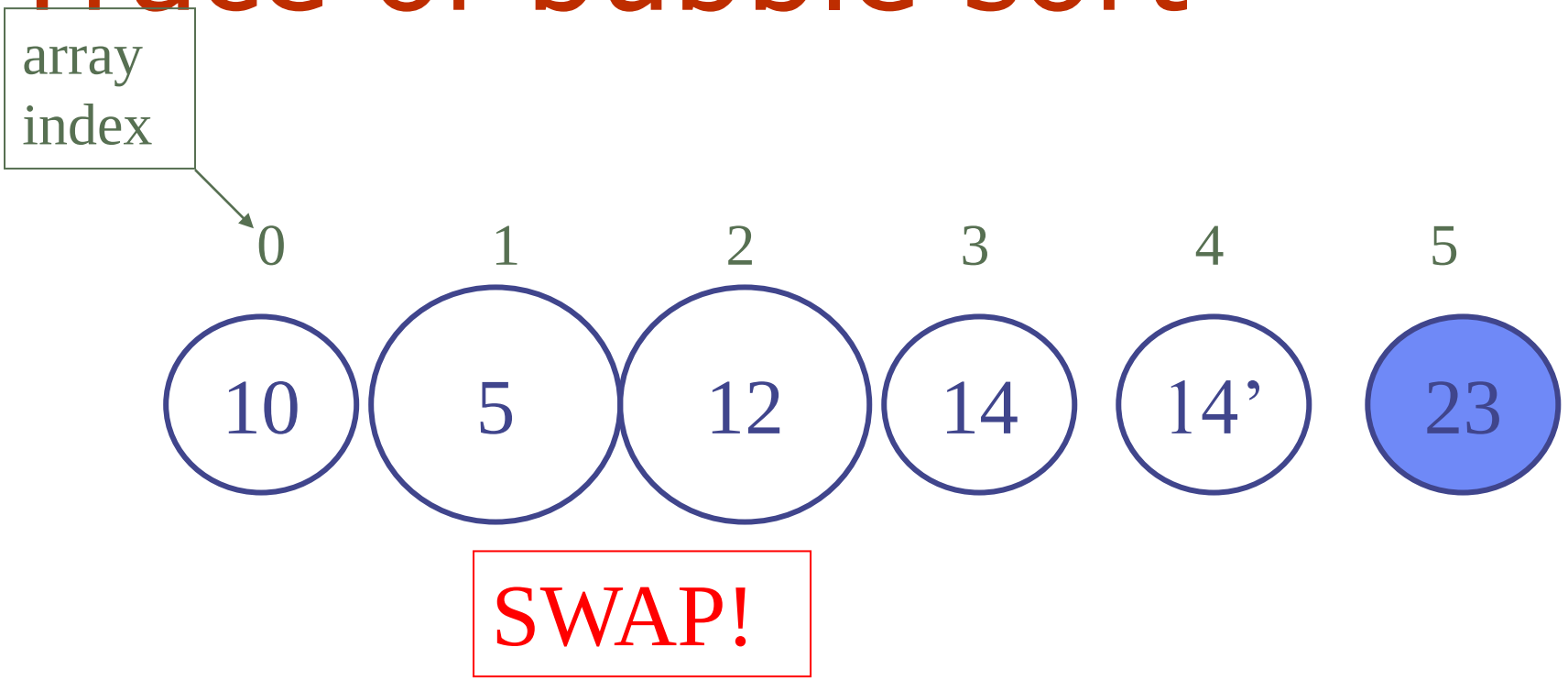
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 1, comparing arr[1] with arr[2]

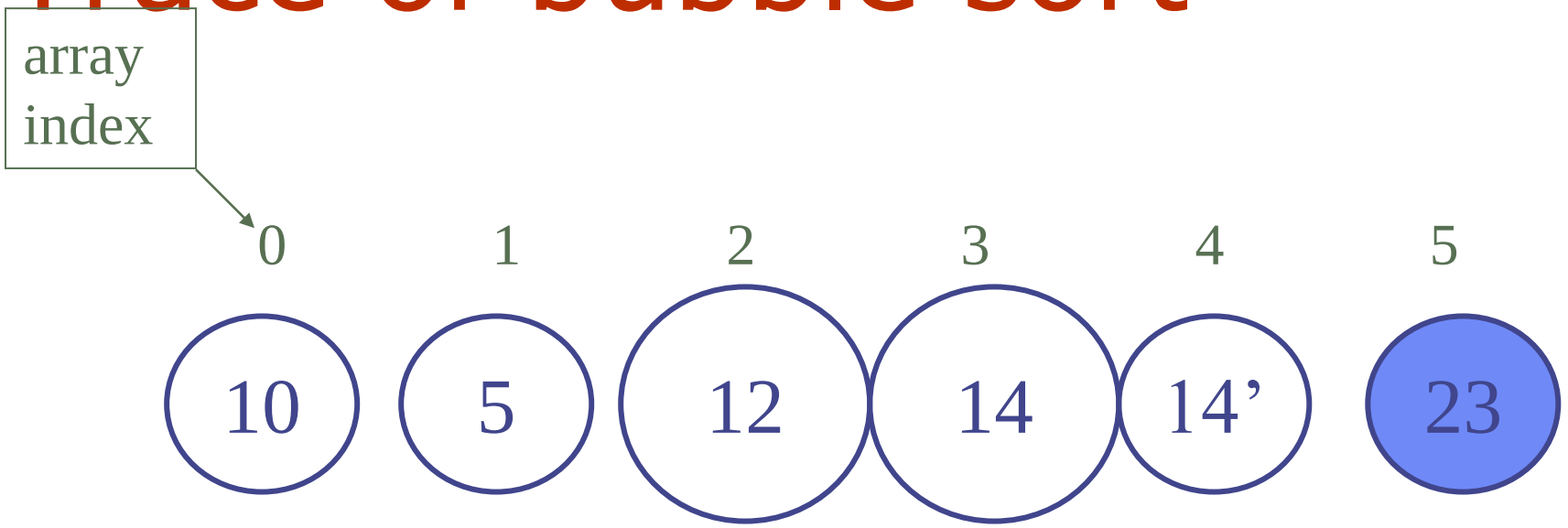
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 1, comparing arr[1] with arr[2]

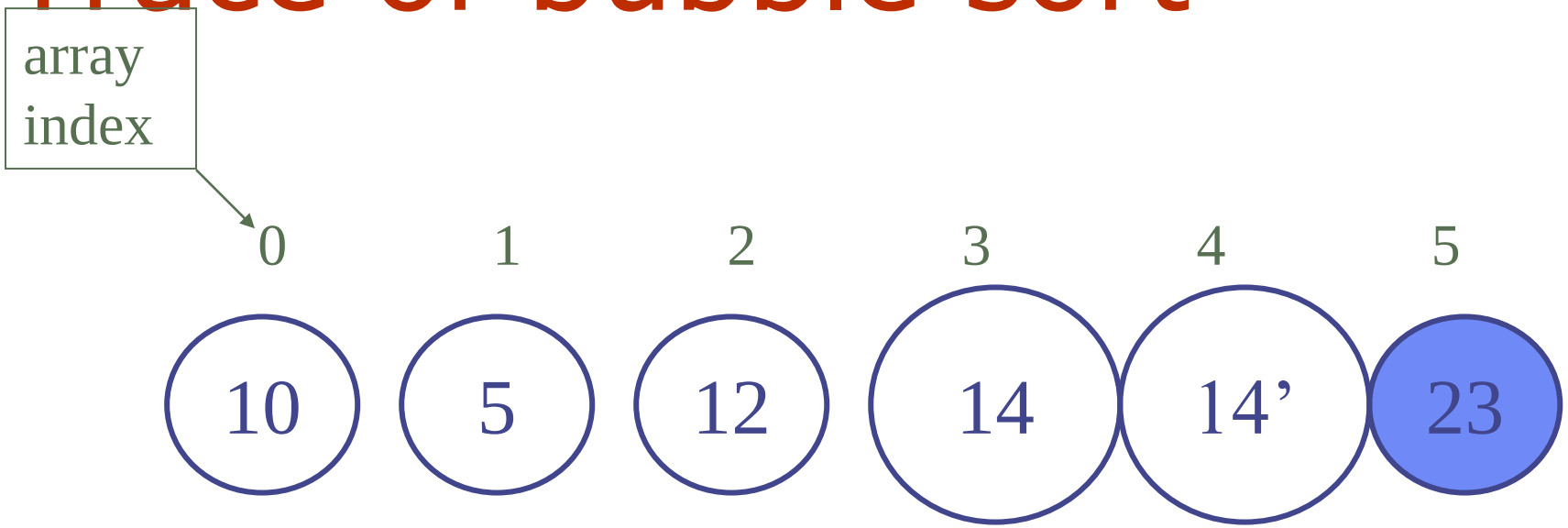
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 2, comparing arr[2] with arr[3]

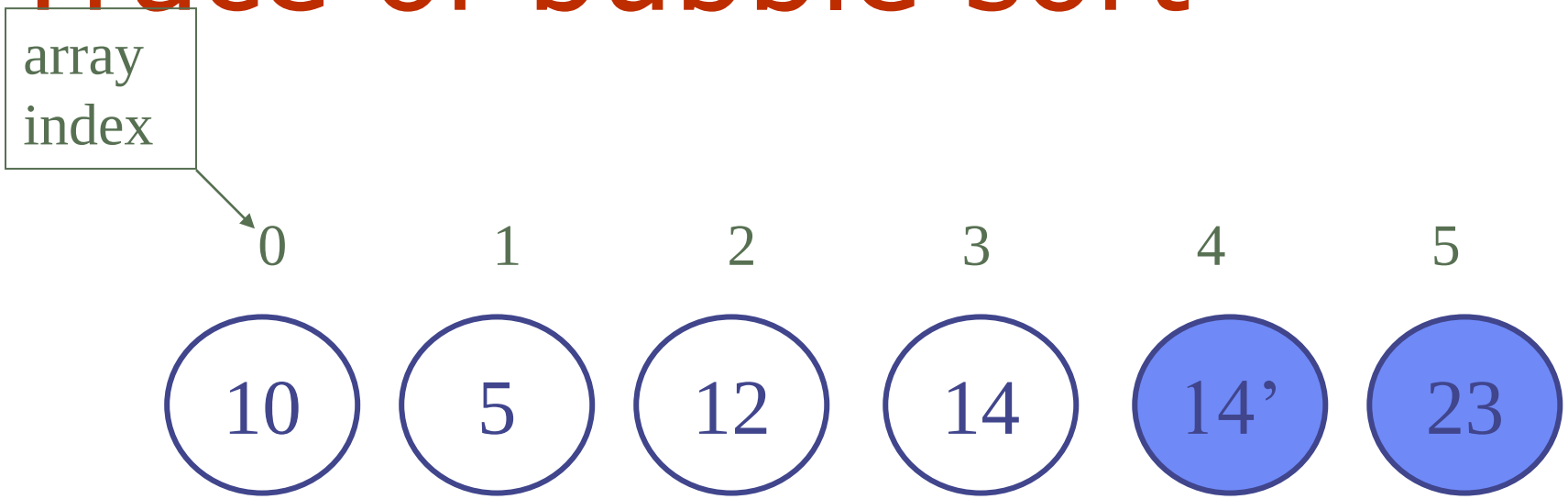
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 3, comparing arr[3] with arr[4]

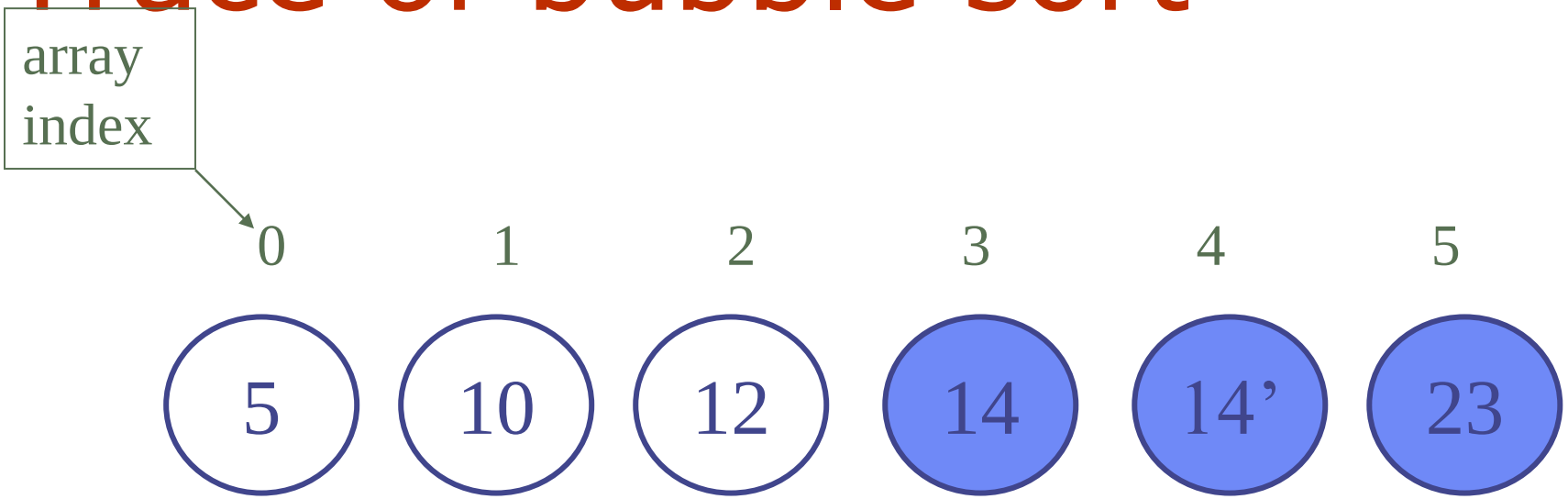
Trace of bubble sort



i = 4, second iteration of the outer loop

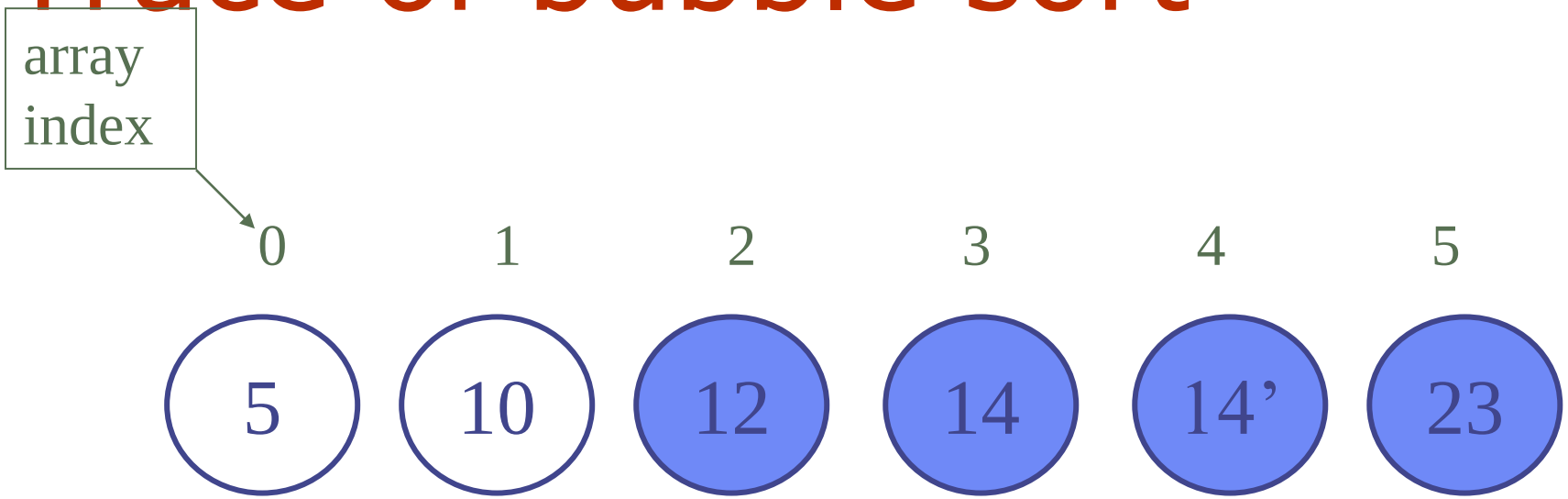
inner loop finished, second largest element in position 4, positions 0-3 unsorted

Trace of bubble sort



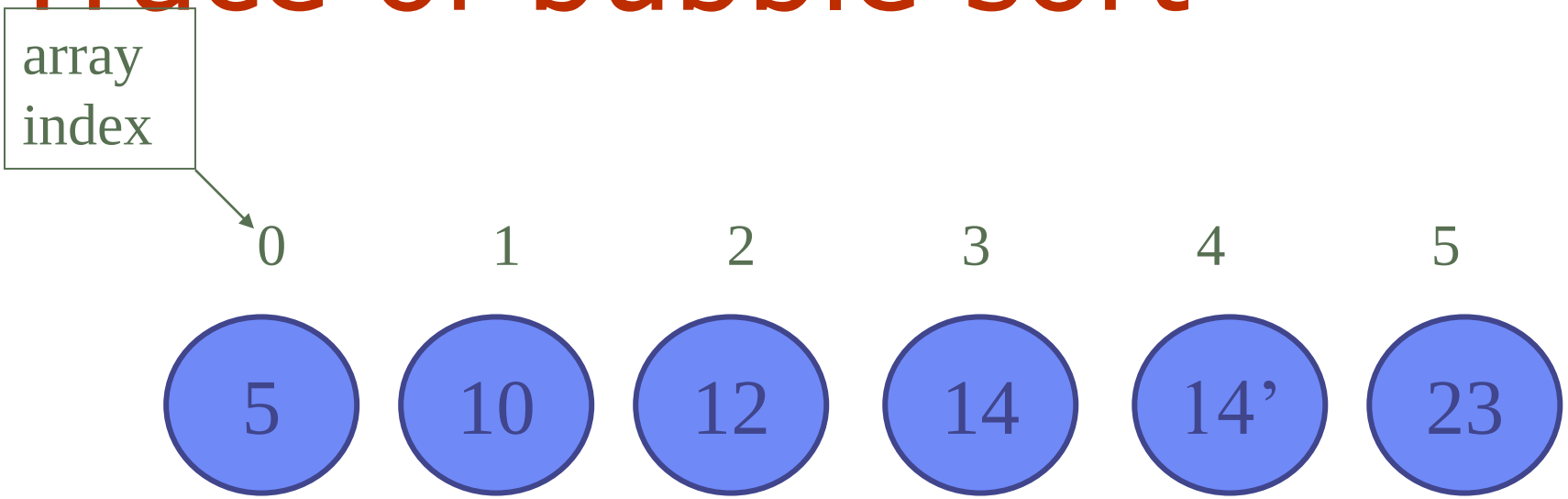
After third iteration…

Trace of bubble sort



After fourth iteration…

Trace of bubble sort



After fifth iteration…

Note: 14 and 14’ are in same relative order as they started, hence:

bubble sort is “stable”

Sorting “Stability”

- If sorting int[] then it does not really matter if the entries are swapped
- So why care about stability at all?

- NOTE: “stable” does NOT mean “the code does not break” !!
  - This is a common error!

Sorting "Stability"

- Often we sorting objects according to some comparison function: compare(o1, o2) returns negative, zero, or positive for "less than", "equal", or "greater than"
- compare(o1,o2)==0 means objects o1 and o2 are
  - equal with respect to the desired ordering
  - but not necessarily that they have the same contents
    - i.e. are not identical !

- E.g. object is a row of a spreadsheet and compare uses just one specified column - many different rows can be equal, but not identical

Sorting "Stability"

- If sorting a spreadsheet, then might sort by one column then another.
- Do not want the sorting to unnecessarily change the order of the rows, as this can be annoying and confusing.
- *"Sort by column A, followed by a stable sort on column B"*
  means that still will have a secondary sort on column A

- **EXERCISE (offline): Experiment with Excel to see if it does stable sorts or not.**

Complexity of bubble sort

- For an array of size n, in the worst case: 1st passage through the inner loop: n-1 comparisons and n-1 swaps
- ...
- (n-1)st passage through the inner loop: one comparison and one swap.
- All together: t ((n-1) + (n-2) + ... + 1), where t is the time required to do one comparison, one swap, check the inner loop condition and increment j.
- We also spend constant time k declaring i,j,temp and initialising i. Outer loop is executed n-1 times, suppose the cost of checking the loop condition and decrementing i is t_1.

Complexity of bubble sort

$t\ ((n-1) + (n-2) + \ldots + 1) + k + t_1(n-1)$

$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

["arithmetic sum" see self-study material]

so our function equals

$$t\ n*(n-1)/2 + k + t_1(n-1) =$$
$$\tfrac{1}{2}\ t\ (n^2-n) + t_1\ (n-1) + k$$

(worst-case) complexity $O(n^2)$.

Proof: Complexity of bubble sort (study offline)

Need to find n_0 and c, such that for all $n \geq n_0$, $\frac{1}{2} t (n^2-n) + t_1(n-1) + k \leq c * n^2$

$\frac{1}{2} t n^2 - \frac{1}{2} t n + t_1 n - t_1 + k \leq$

$\frac{1}{2} t n^2 + t_1 n + k \leq$

$t n^2 + t_1 n^2 + k n^2$ (if $n \geq 1$)

Take $c = t + t_1 + k$ and $n_0 = 1$.

Selection Sort: Basic Idea

- Similar to bubble sort;
- On each scan:
  - instead of always try to move the “greatest element so far” immediately, we just remember its location and move it at end of scan

Why one want to do this!?

Exercise: Why delay swaps?

Answer:

- Suppose that the entries are large then a swap operation might be quite expensive
- So might want to reduce the number of swaps by directly moving entries to “the right place”

Selection sort

```
void  selectionSort(int arr[]){
  int i, j, temp, pos_greatest;
  for( i = arr.length-1; i > 0; i--){
     pos_greatest = 0;
     for(j = 0; j <= i; j++){
       if( arr[j] >= arr[pos_greatest]){
          pos_greatest =  j;
     }//end inner for loop
     if ( i != pos_greatest ) {
       temp = arr[i];
       arr[i] = arr[pos_greatest];
        arr[pos_greatest] = temp; }
  }//end outer for loop
}//end selection sort
```

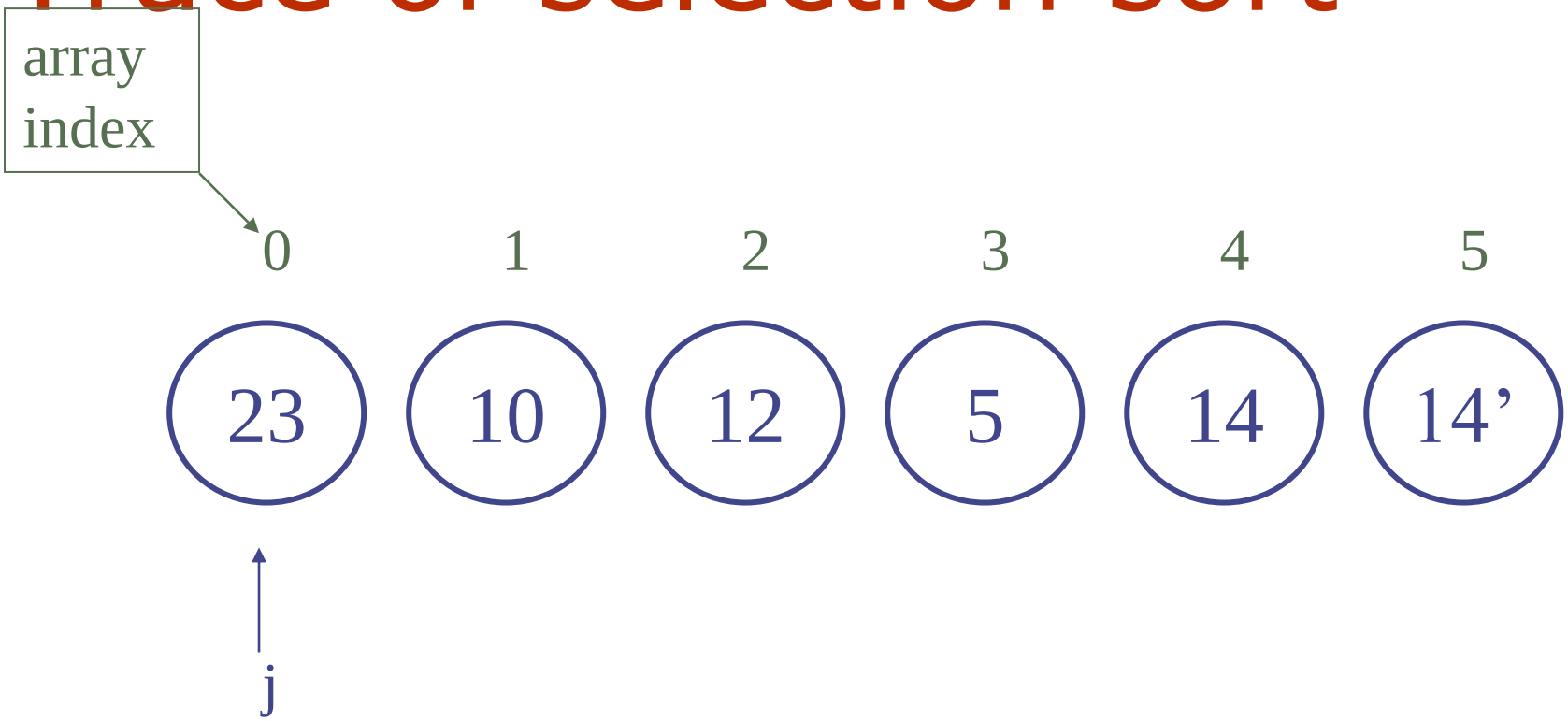
compare the current element to the largest seen so far; if it is larger, remember its index

swap the largest element to the end of range, if not already there

Trace of selection sort

i = 5, first iteration of the outer loop

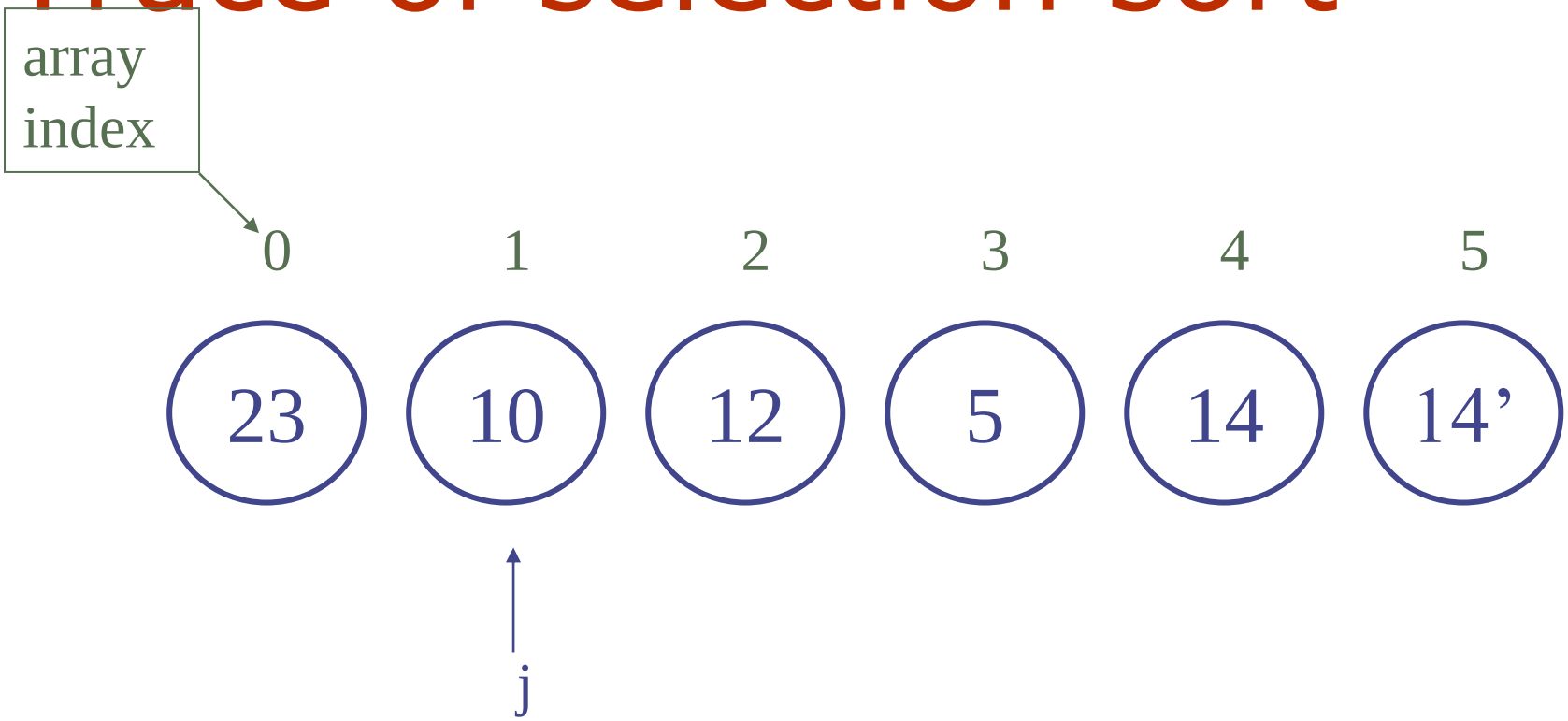
Trace of selection sort



i = 5, first iteration of the outer loop

j = 0, pos_greatest = 0

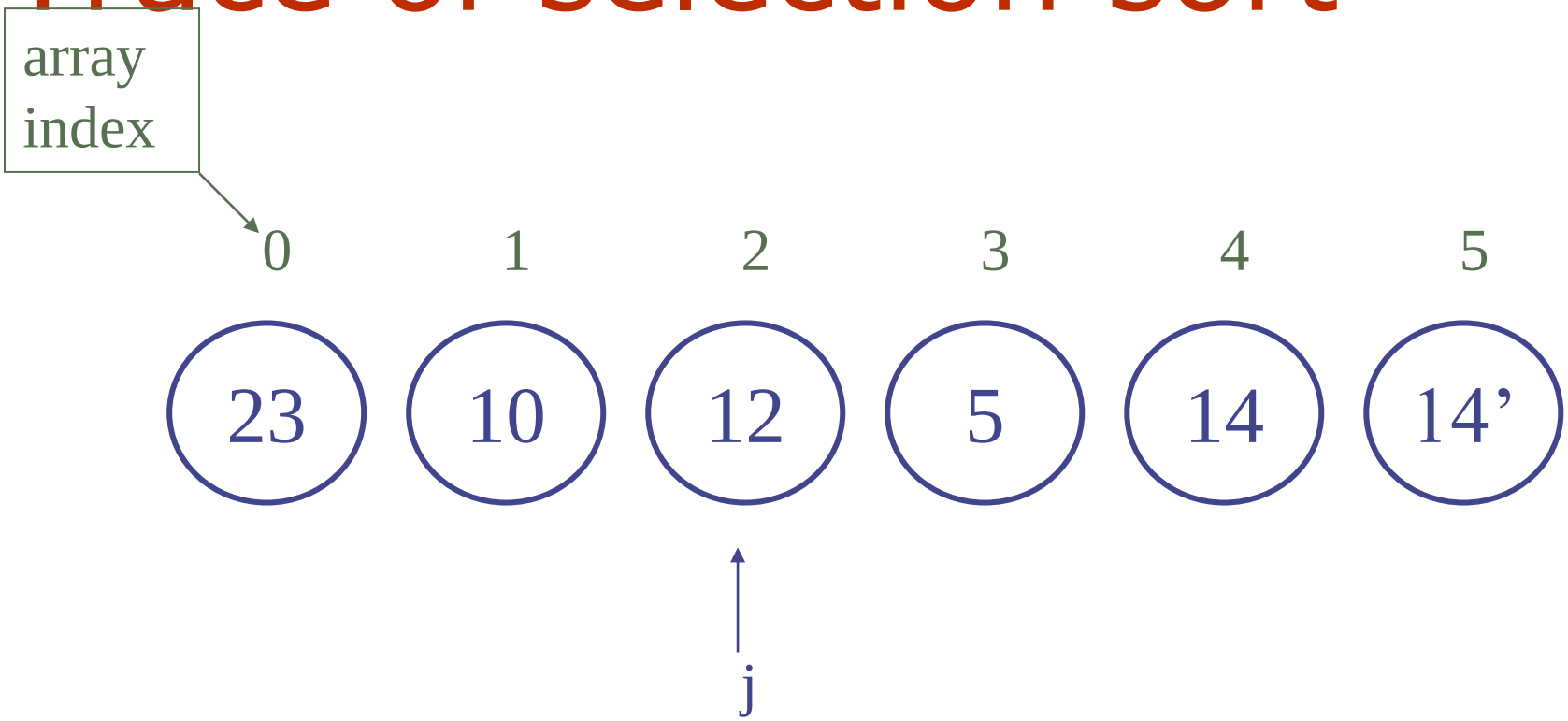
Trace of selection sort



i = 5, first iteration of the outer loop

j = 1, pos_greatest = 0

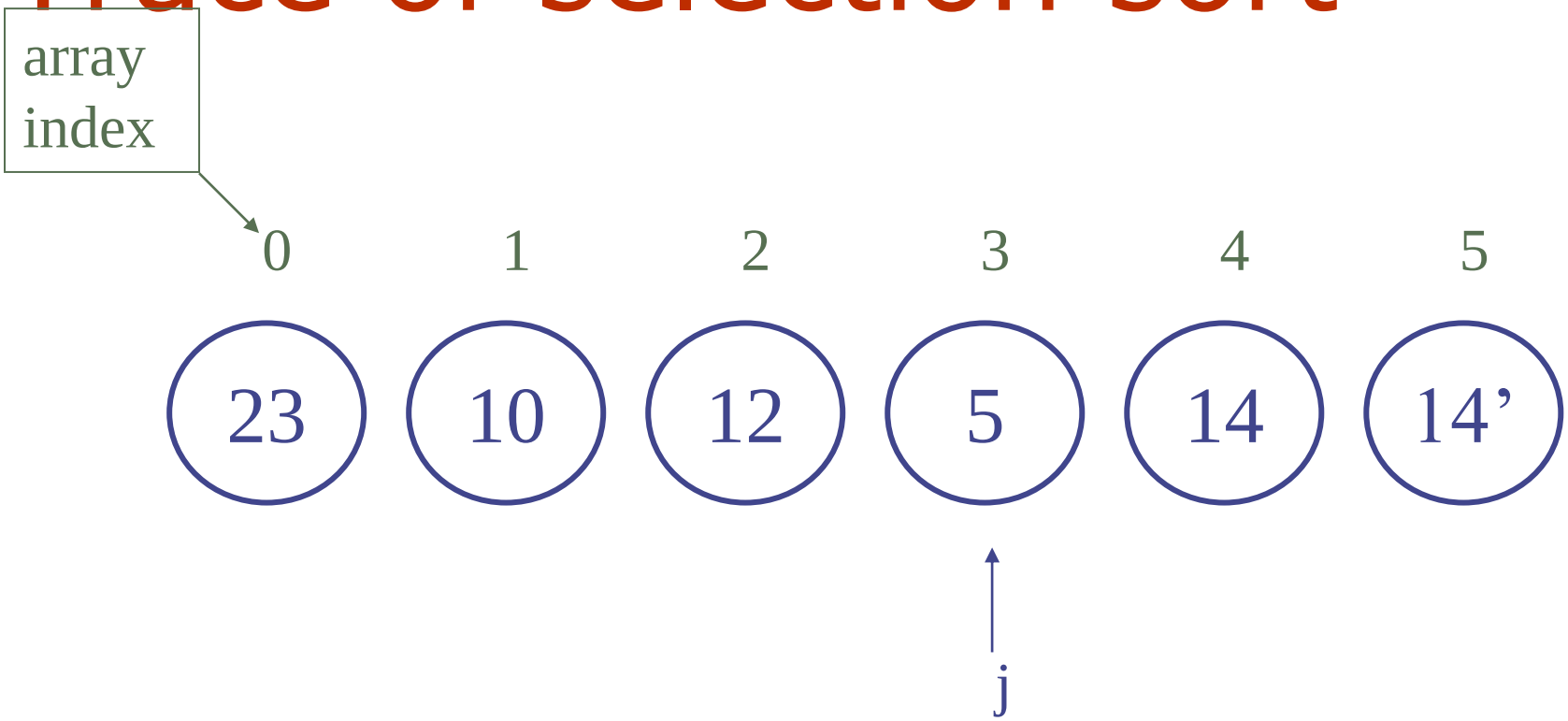
Trace of selection sort



i = 5, first iteration of the outer loop

j = 2, pos_greatest = 0

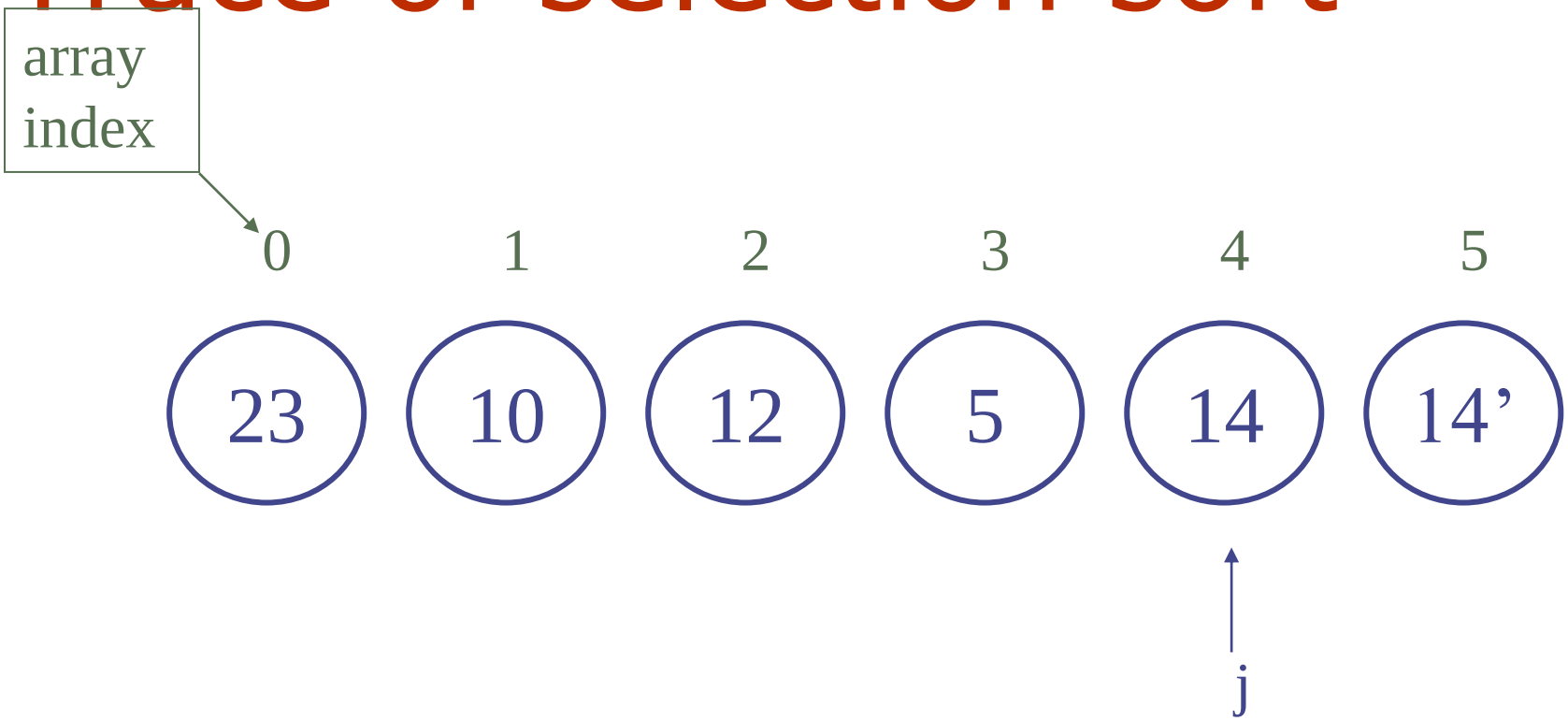
Trace of selection sort



i = 5, first iteration of the outer loop

j = 3, pos_greatest = 0

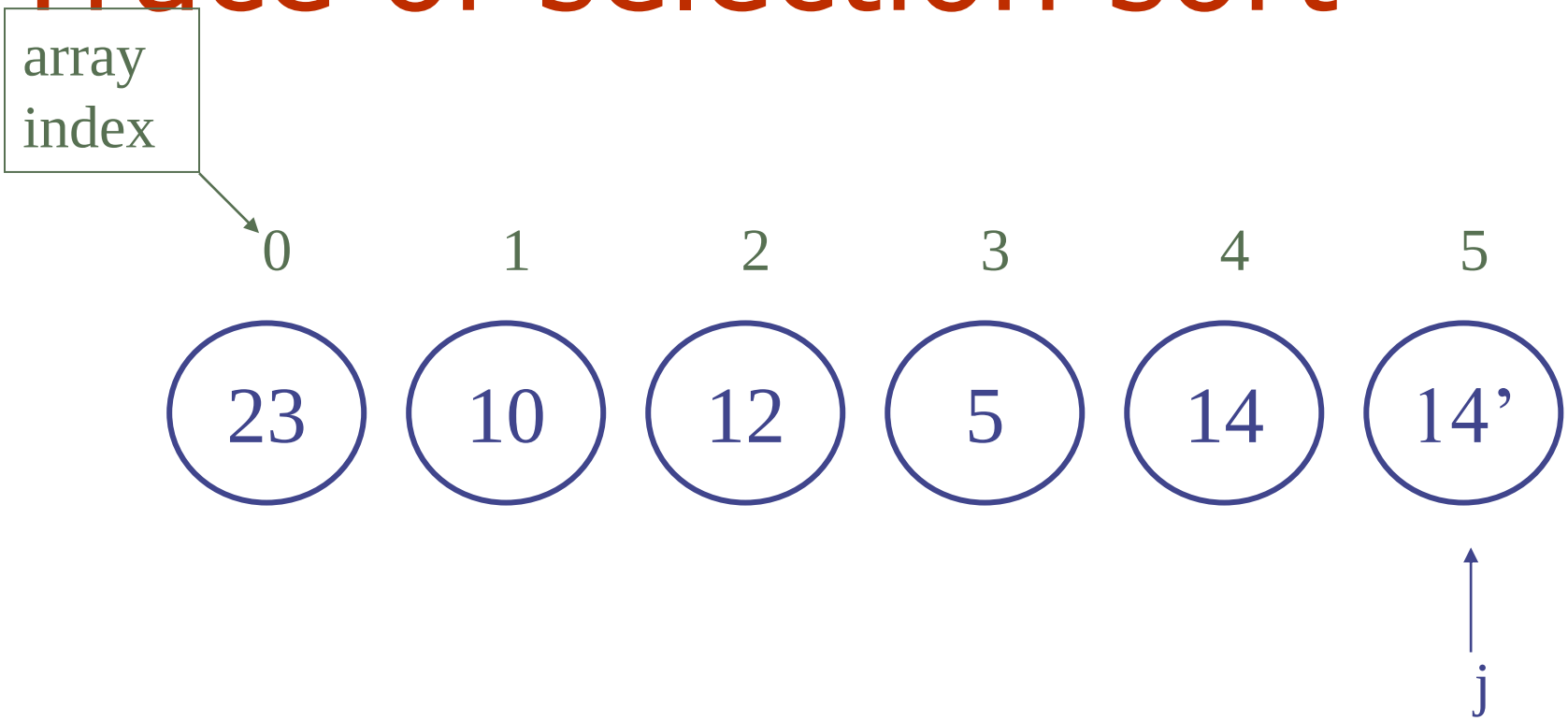
Trace of selection sort



i = 5, first iteration of the outer loop

j = 4, pos_greatest = 0

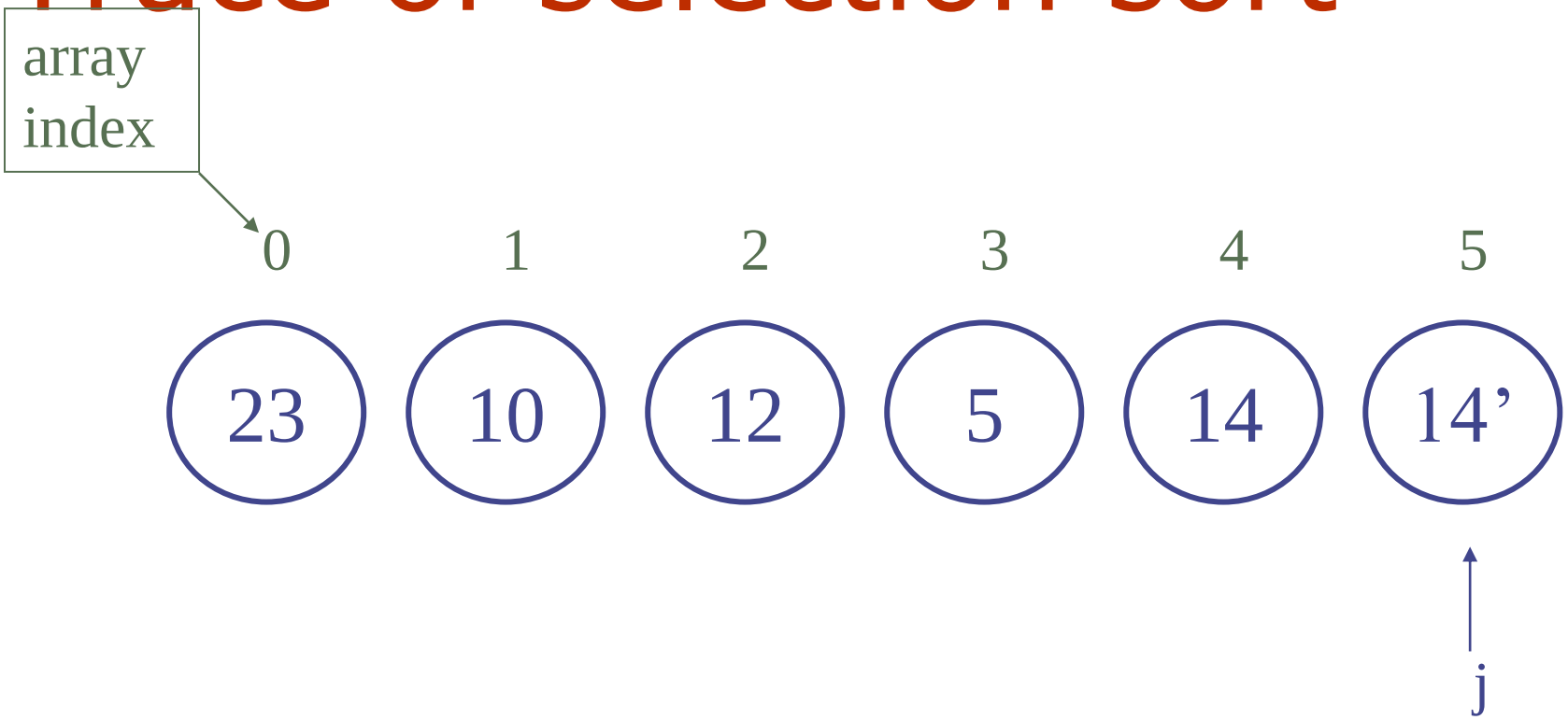
Trace of selection sort



i = 5, first iteration of the outer loop

j = 5, pos_greatest = 0

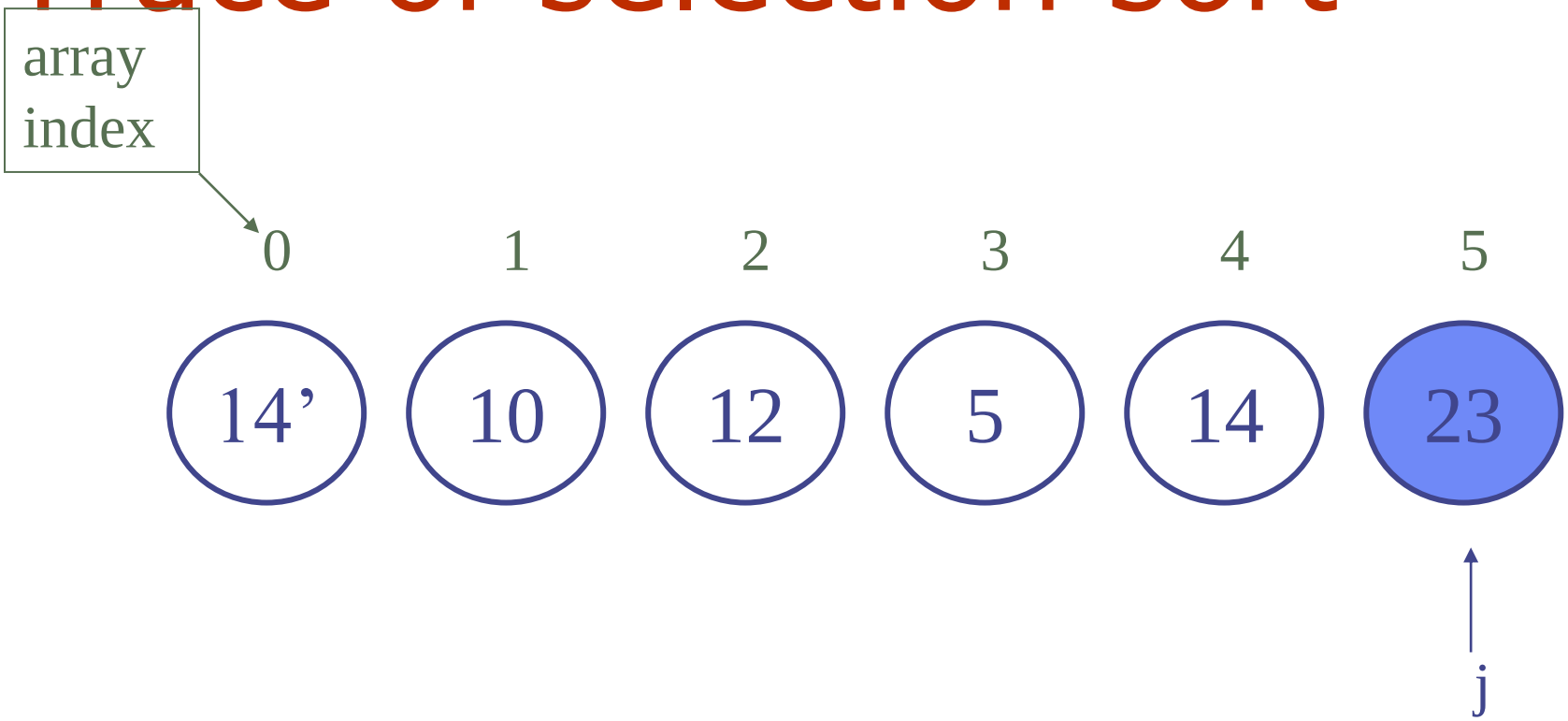
Trace of selection sort



i = 5, first iteration of the outer loop

swap element at pos_greatest=0 to 5

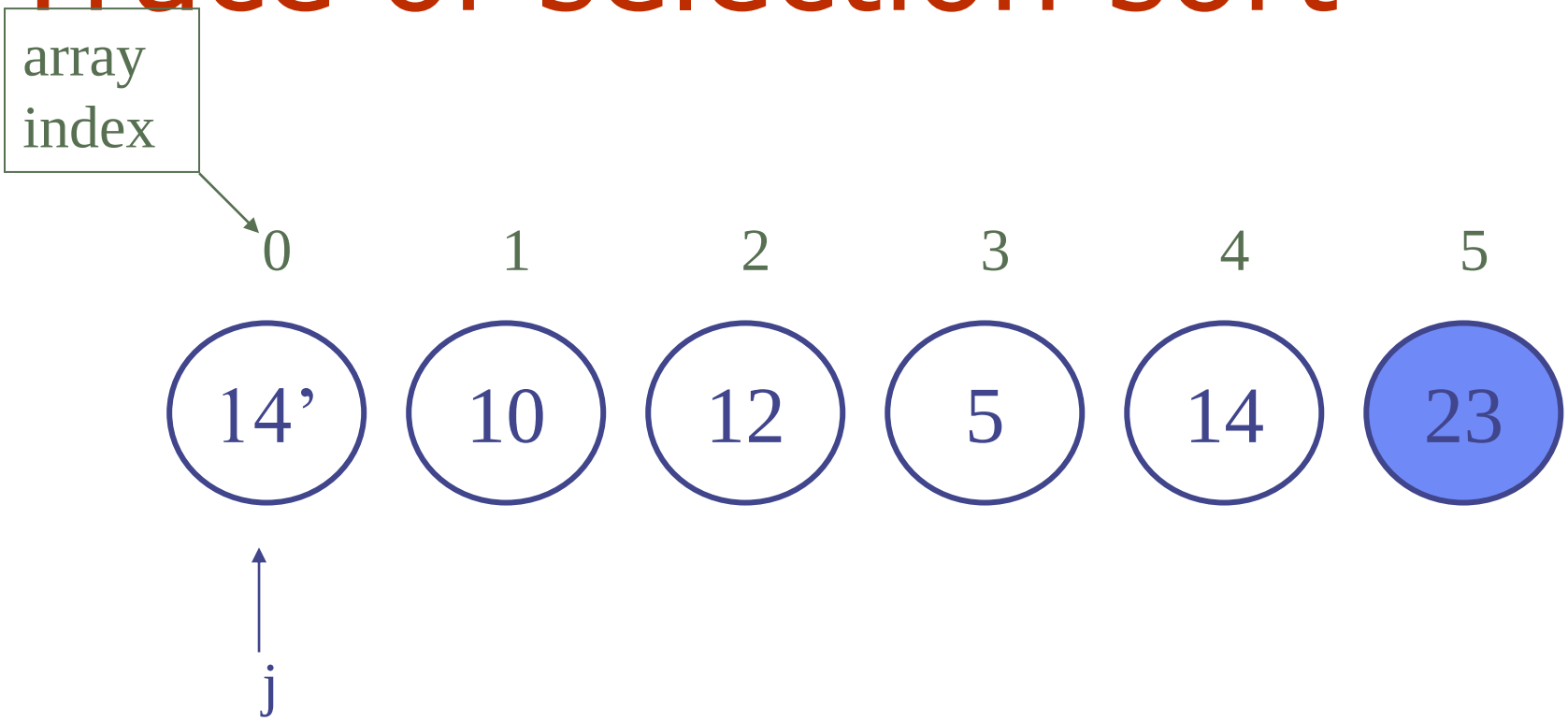
Trace of selection sort



i = 5, first iteration of the outer loop

swap element at pos_greatest=0 to 5

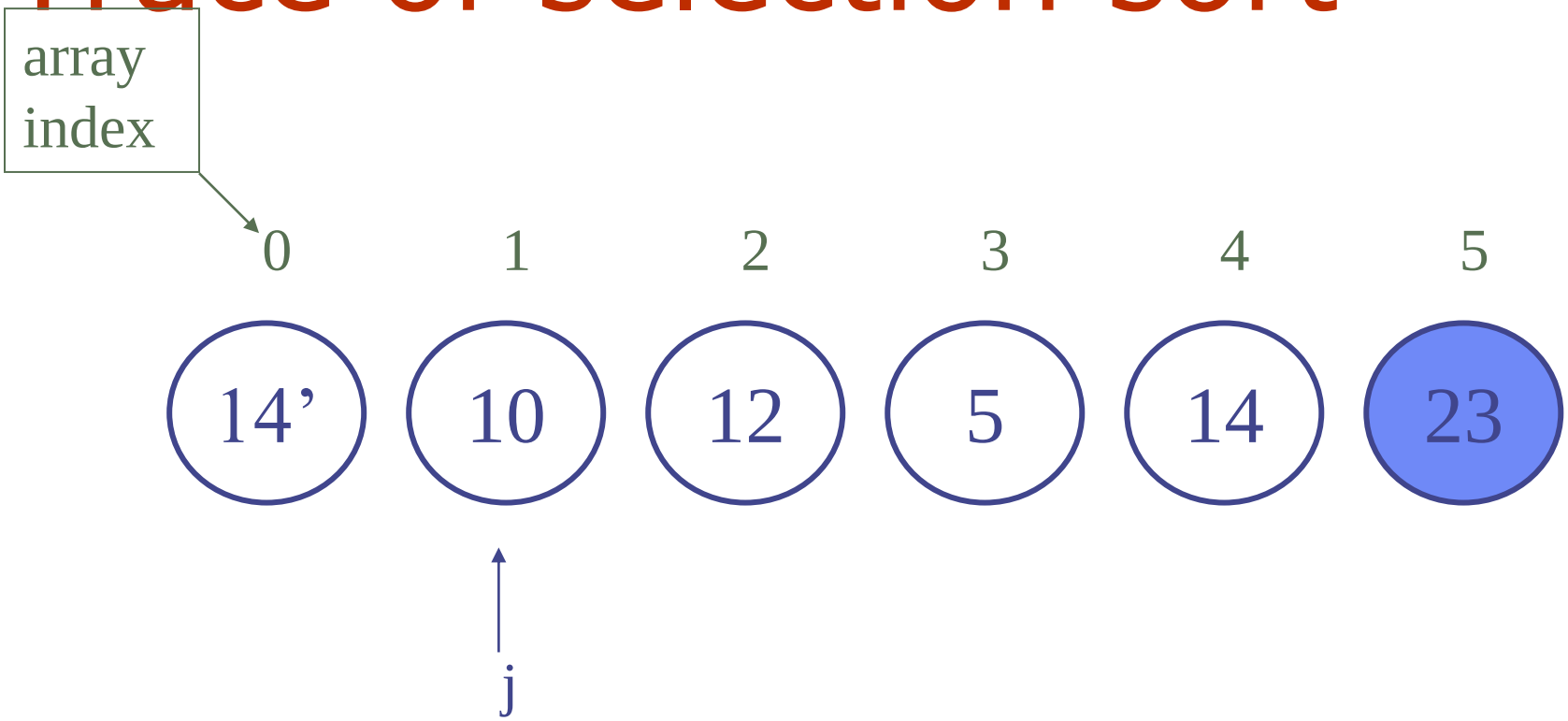
Trace of selection sort



i = 4, second iteration of the outer loop

j = 0, pos_greatest = 0

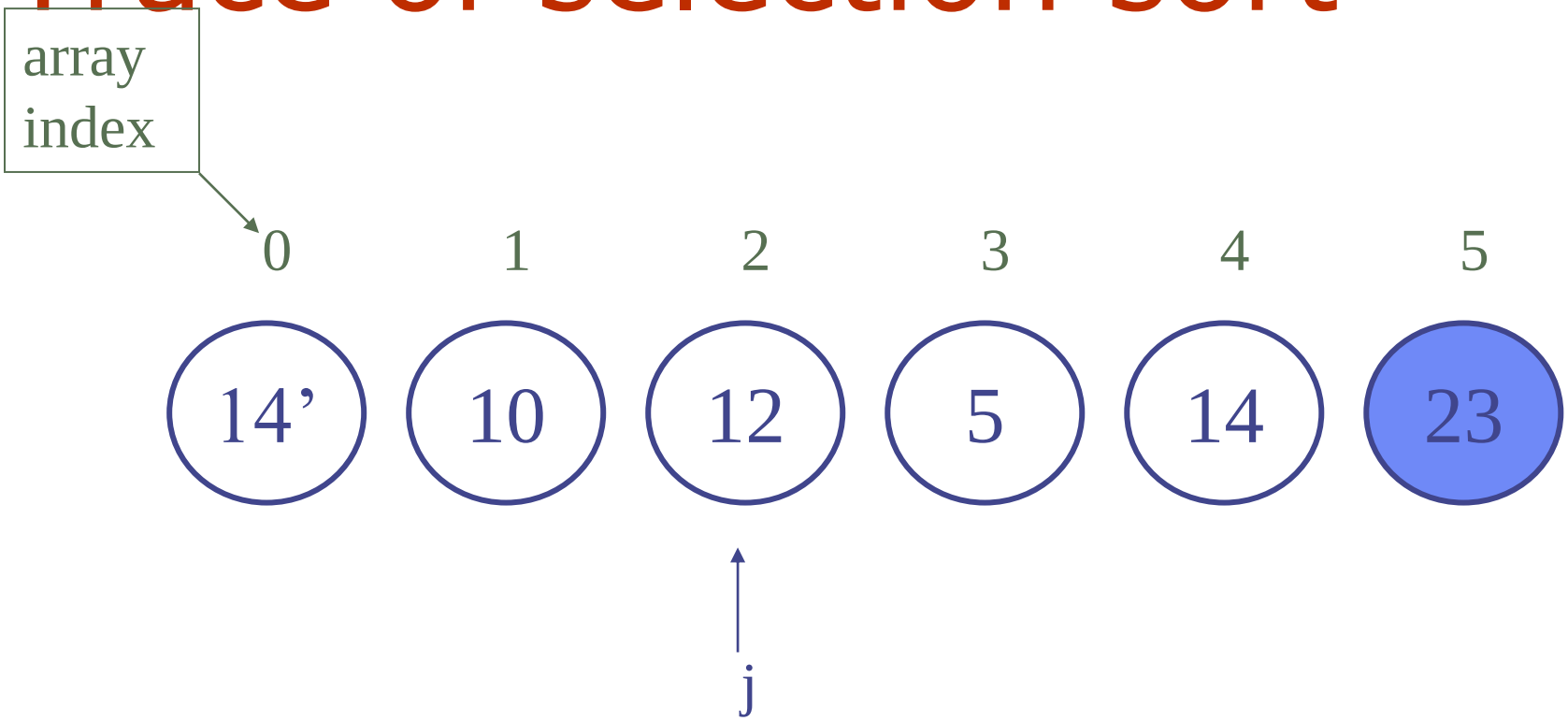
Trace of selection sort



i = 4, second iteration of the outer loop

j = 1, pos_greatest = 0

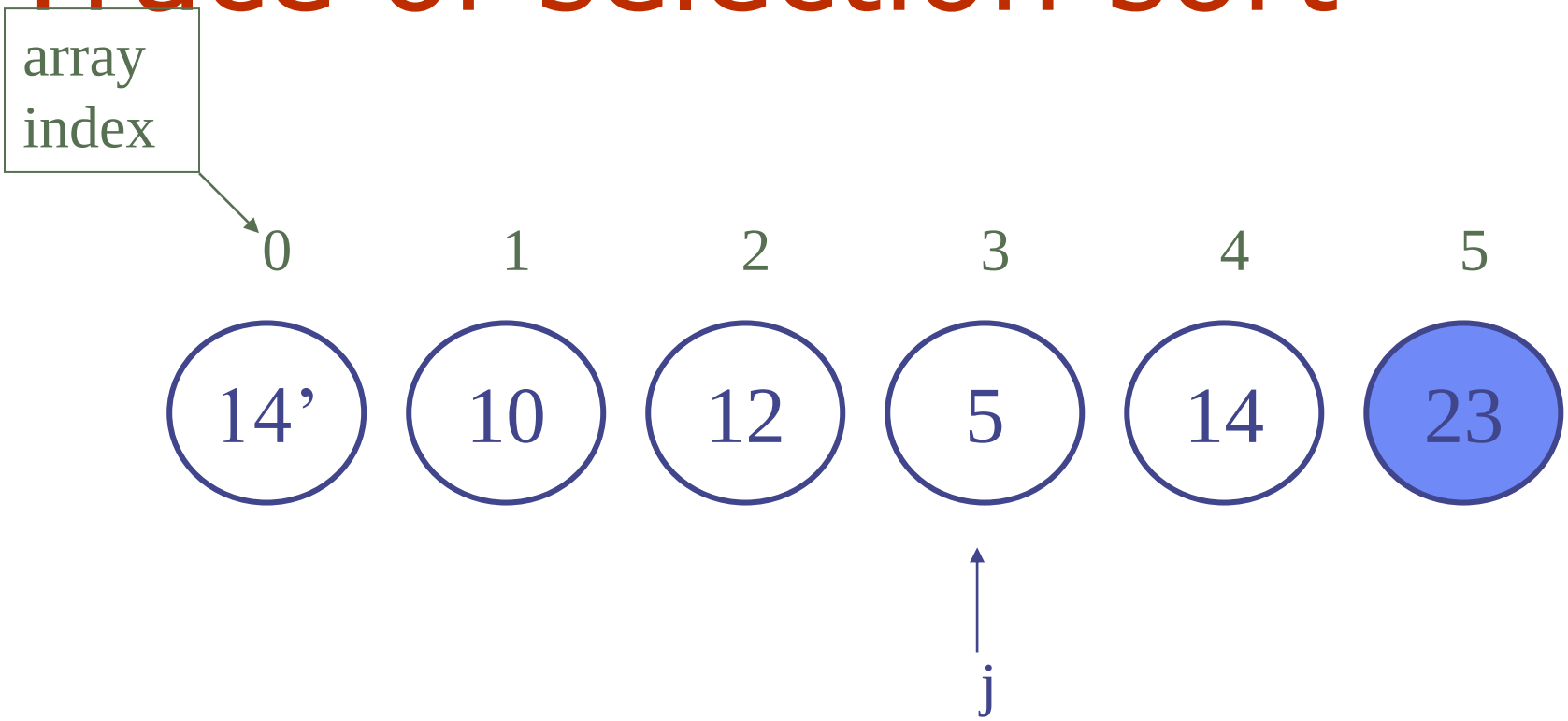
Trace of selection sort



i = 4, second iteration of the outer loop

j = 2, pos_greatest = 0

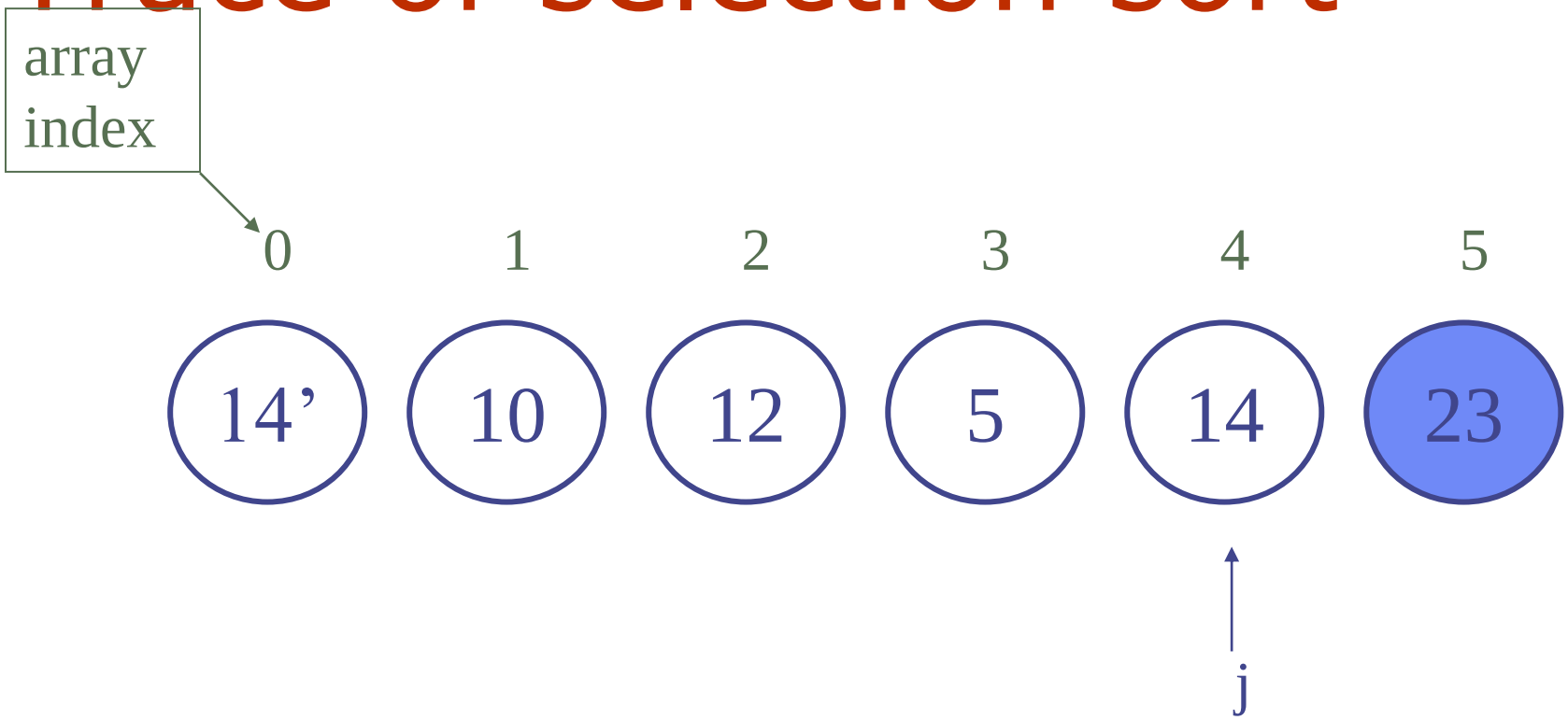
Trace of selection sort



i = 4, second iteration of the outer loop

j = 3, pos_greatest = 0

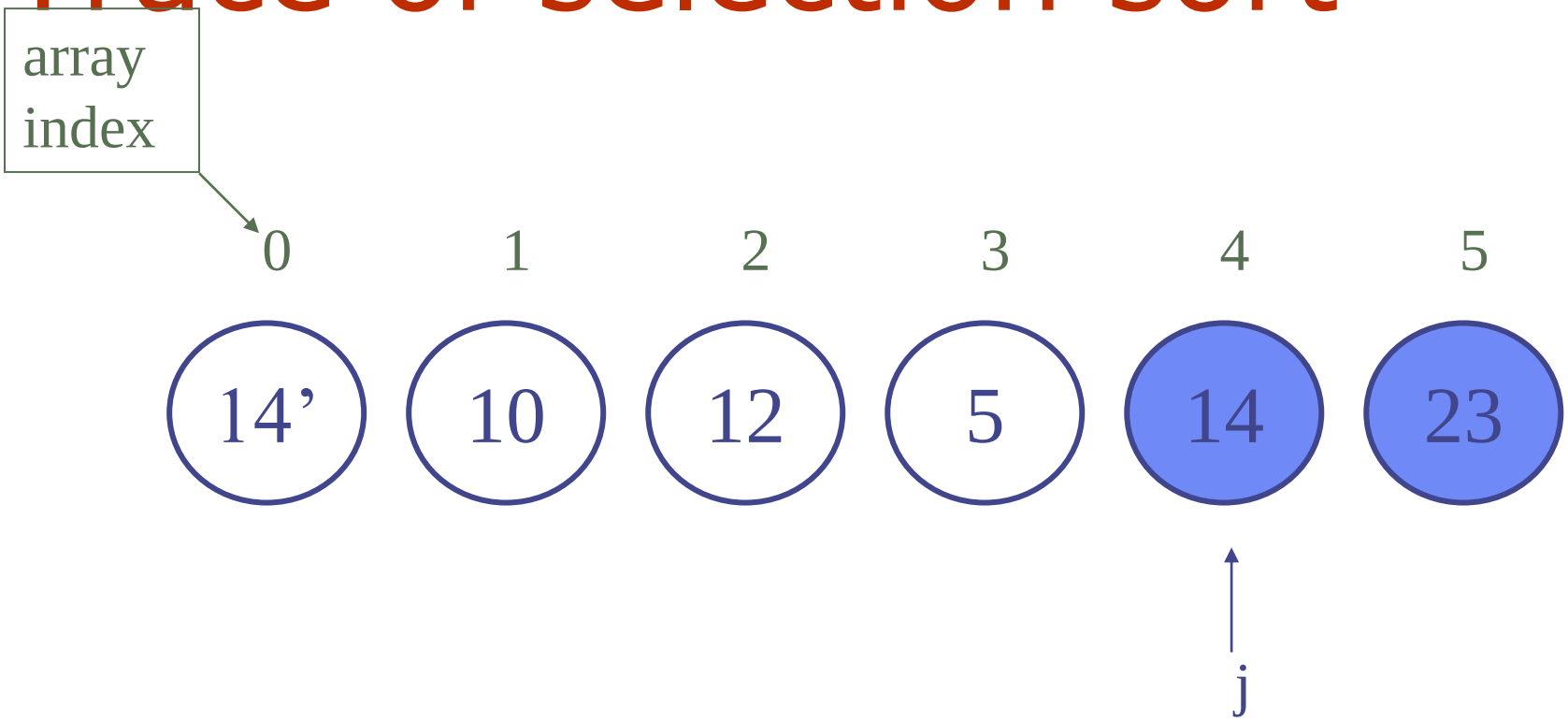
Trace of selection sort



i = 4, second iteration of the outer loop

j = 4, pos_greatest = 4 (because the “>=” in the comparison)

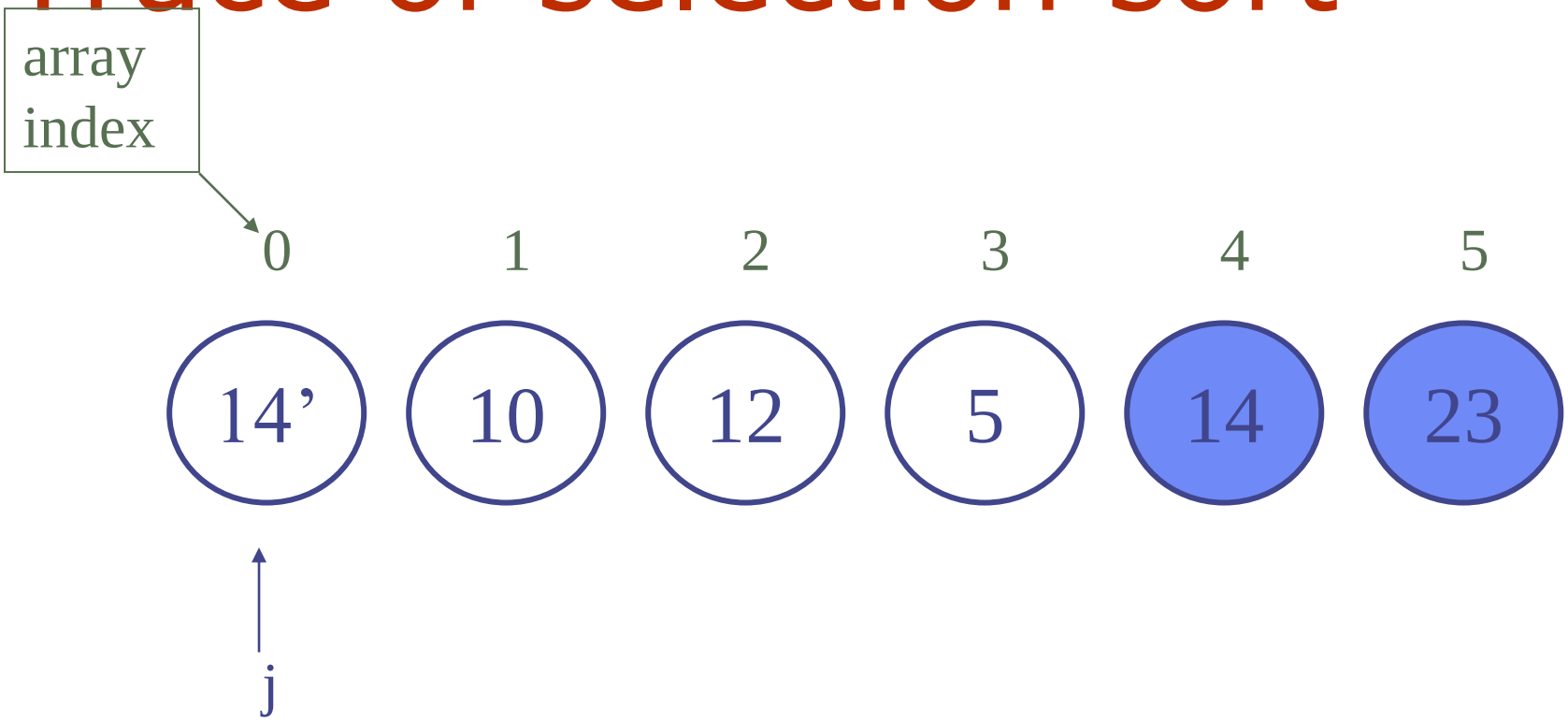
Trace of selection sort



i = 4, second iteration of the outer loop

No need to swap elements

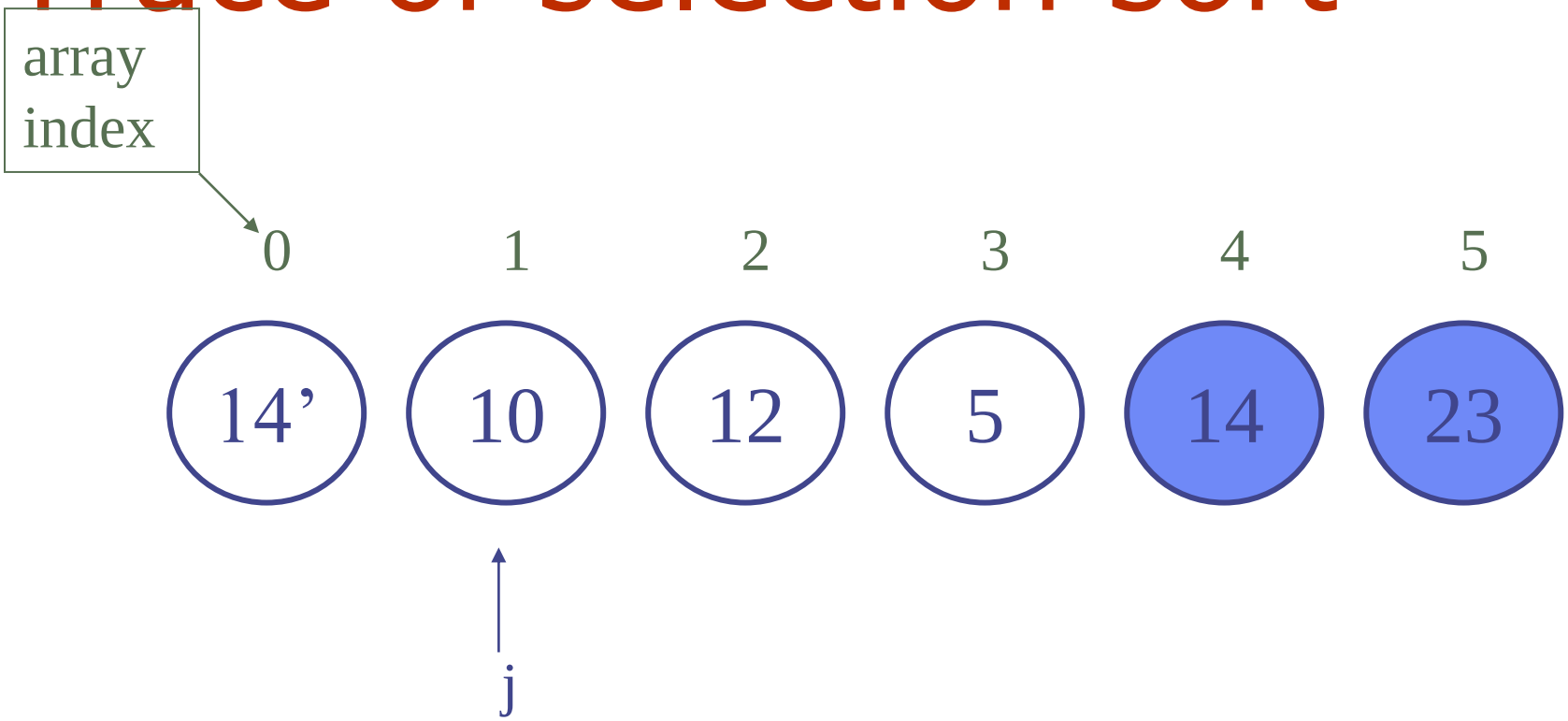
Trace of selection sort



i = 3, third iteration of the outer loop

j = 0, pos_greatest = 0

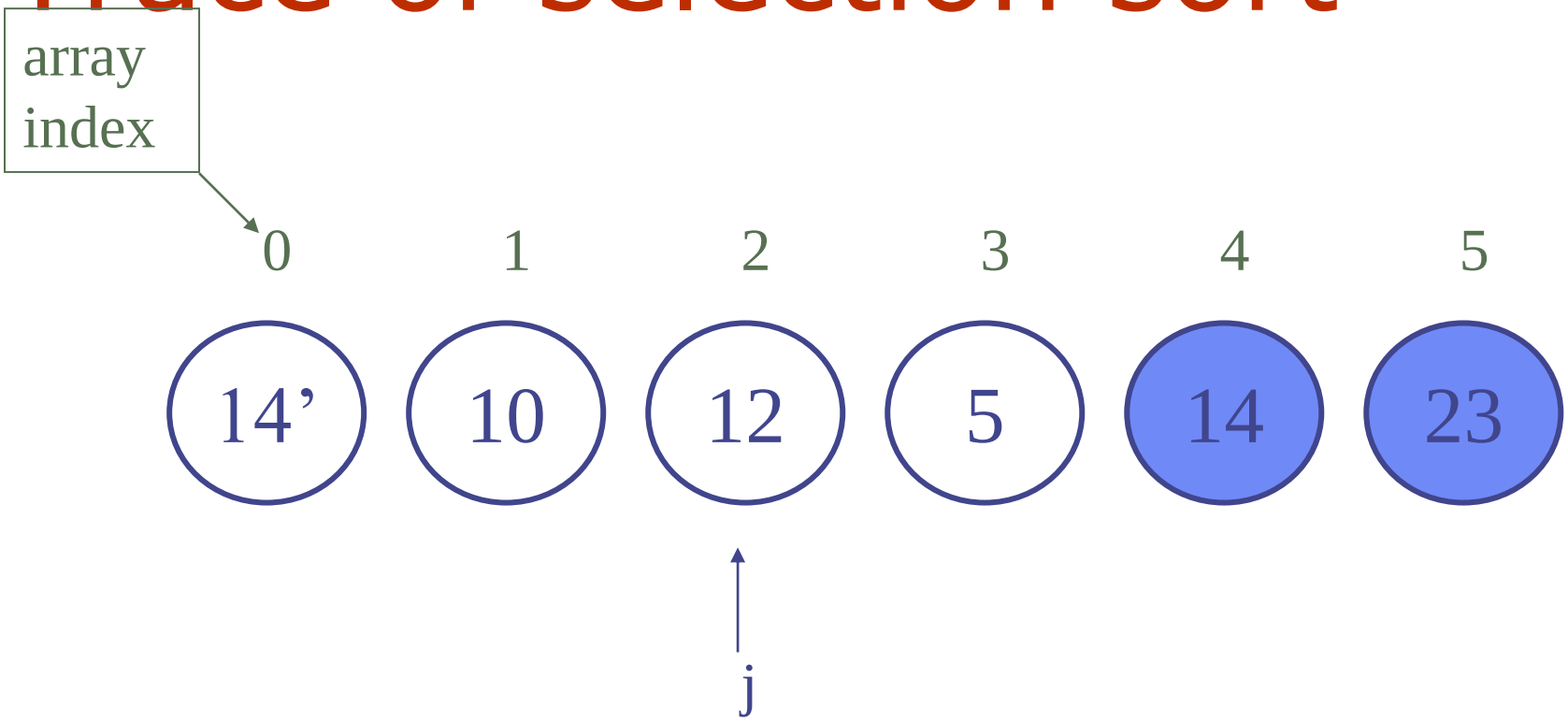
Trace of selection sort



i = 3, third iteration of the outer loop

j = 1, pos_greatest = 0

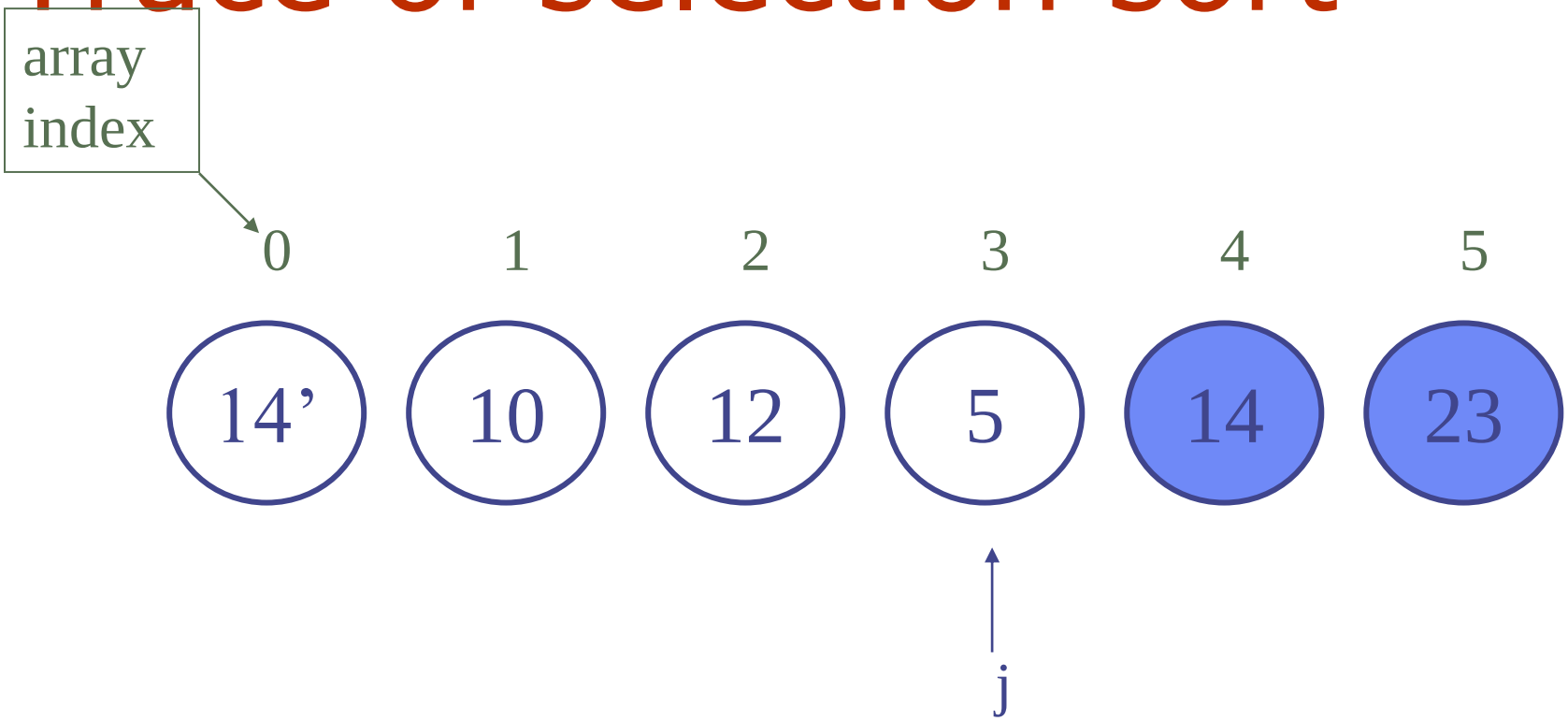
Trace of selection sort



i = 3, third iteration of the outer loop

j = 2, pos_greatest = 0

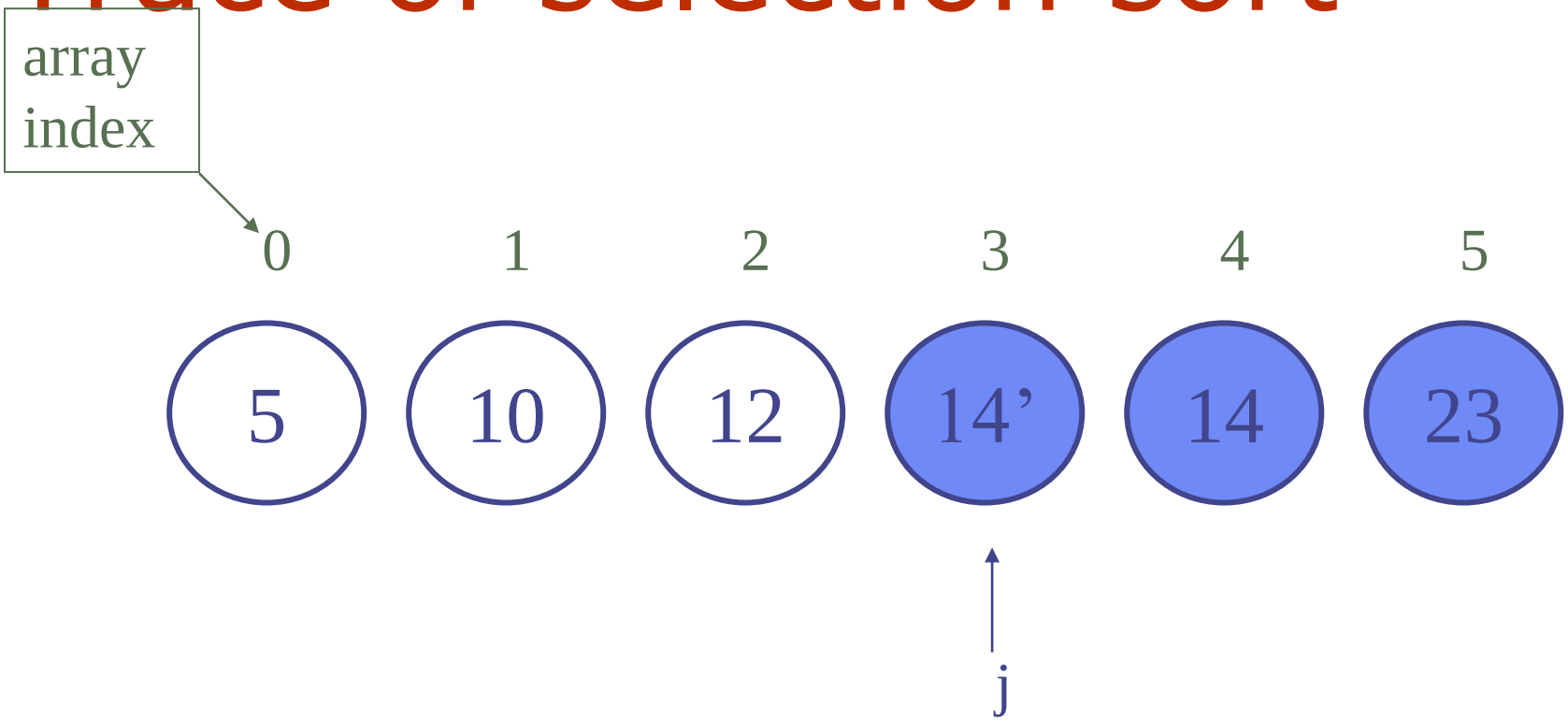
Trace of selection sort



i = 3, third iteration of the outer loop

j = 3, pos_greatest = 0

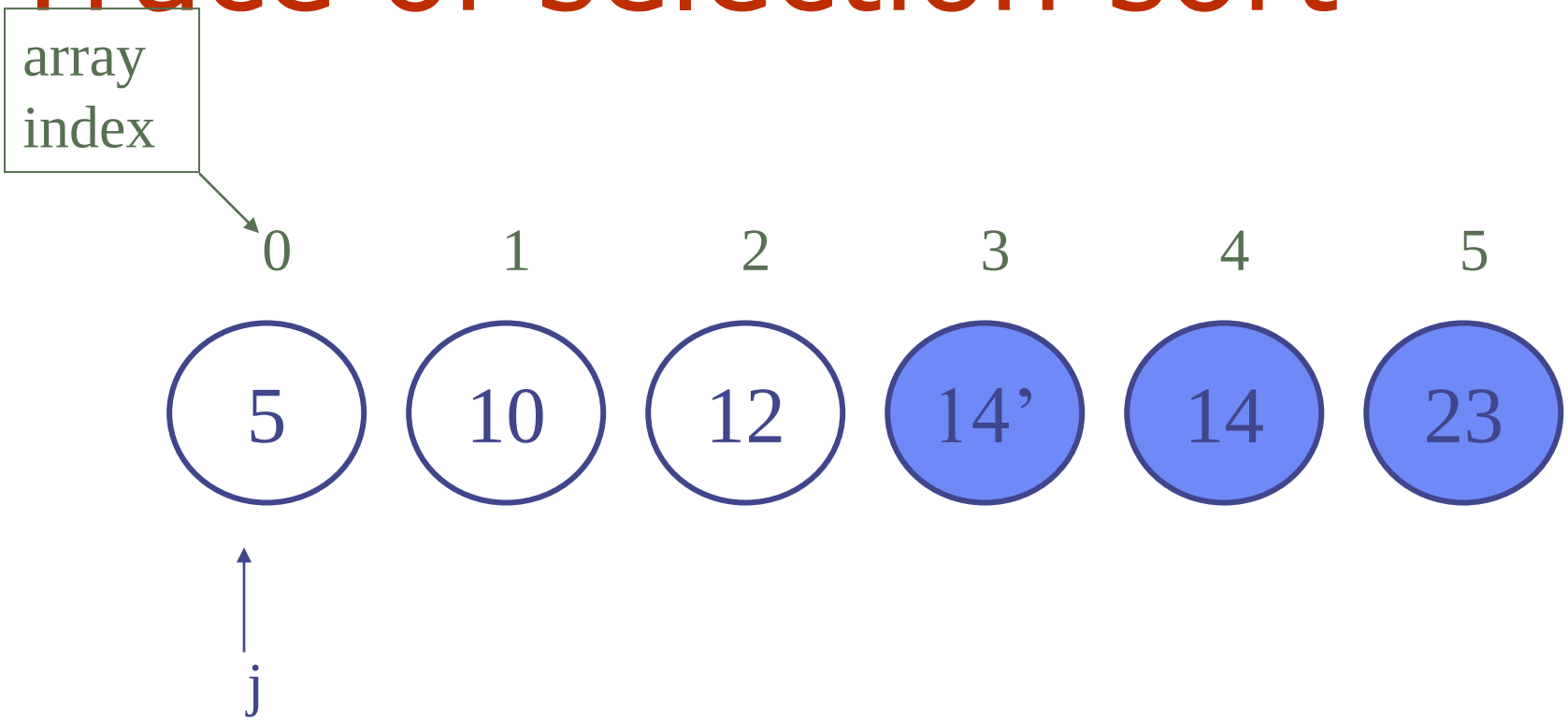
Trace of selection sort



i = 3, third iteration of the outer loop

swap elements at pos_greatest and 3

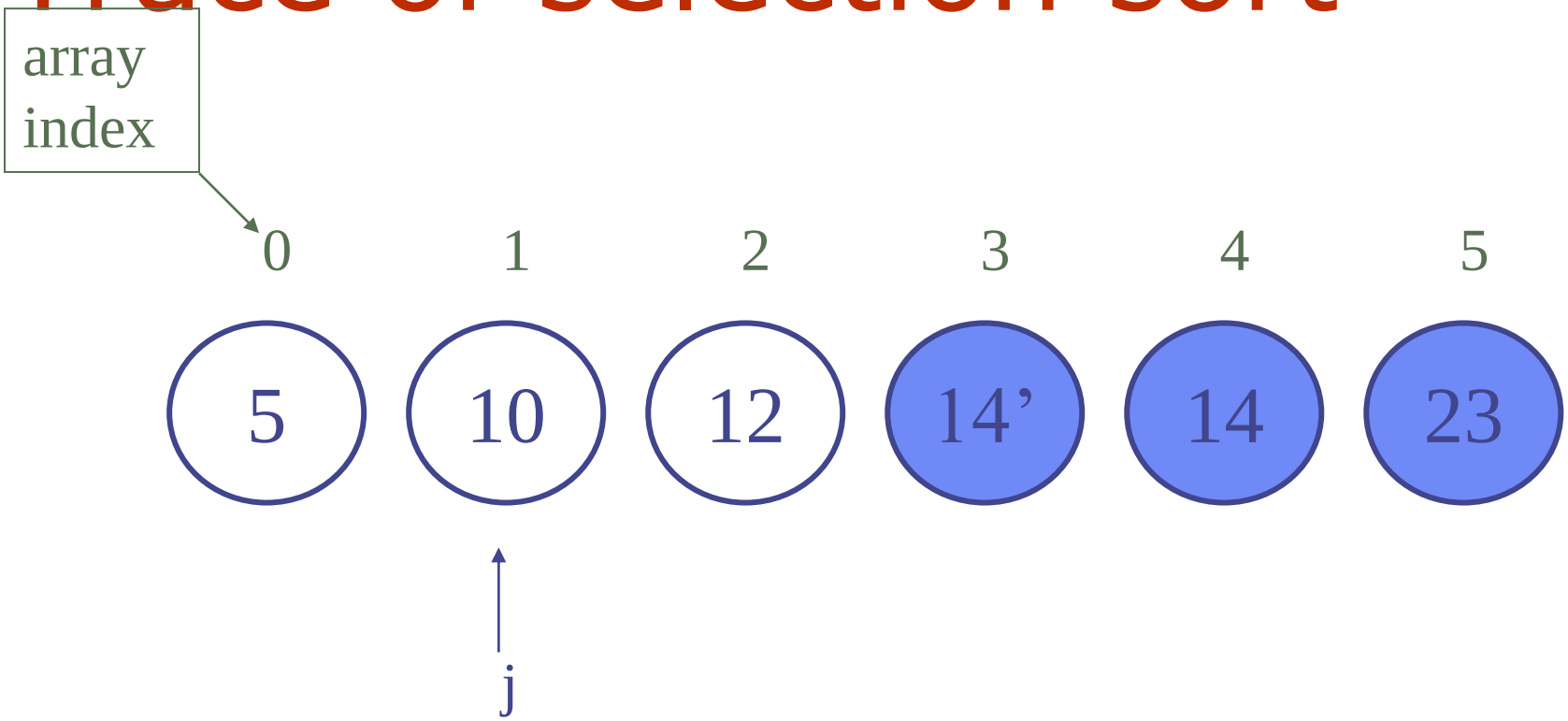
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 0, pos_greatest = 0

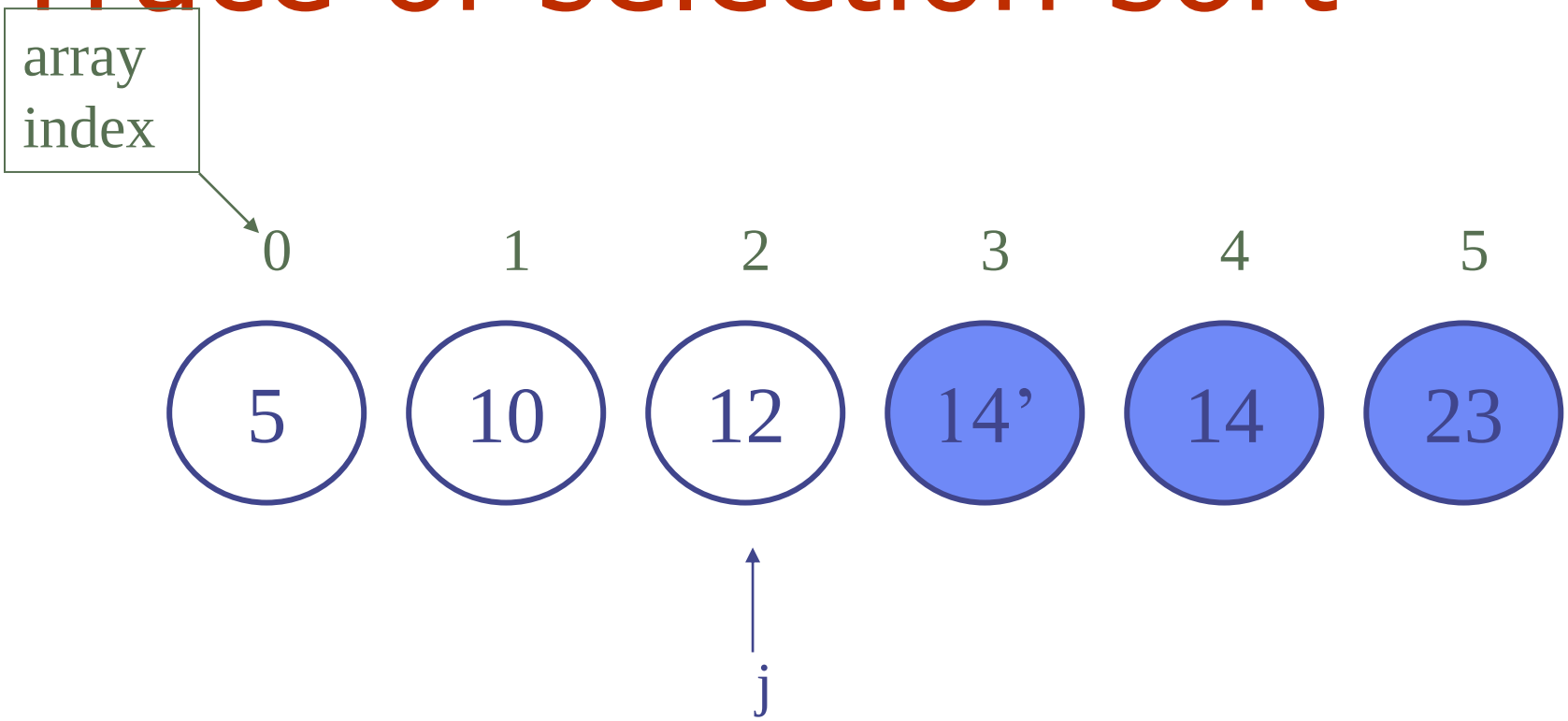
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 1, pos_greatest = 1 (changed!)

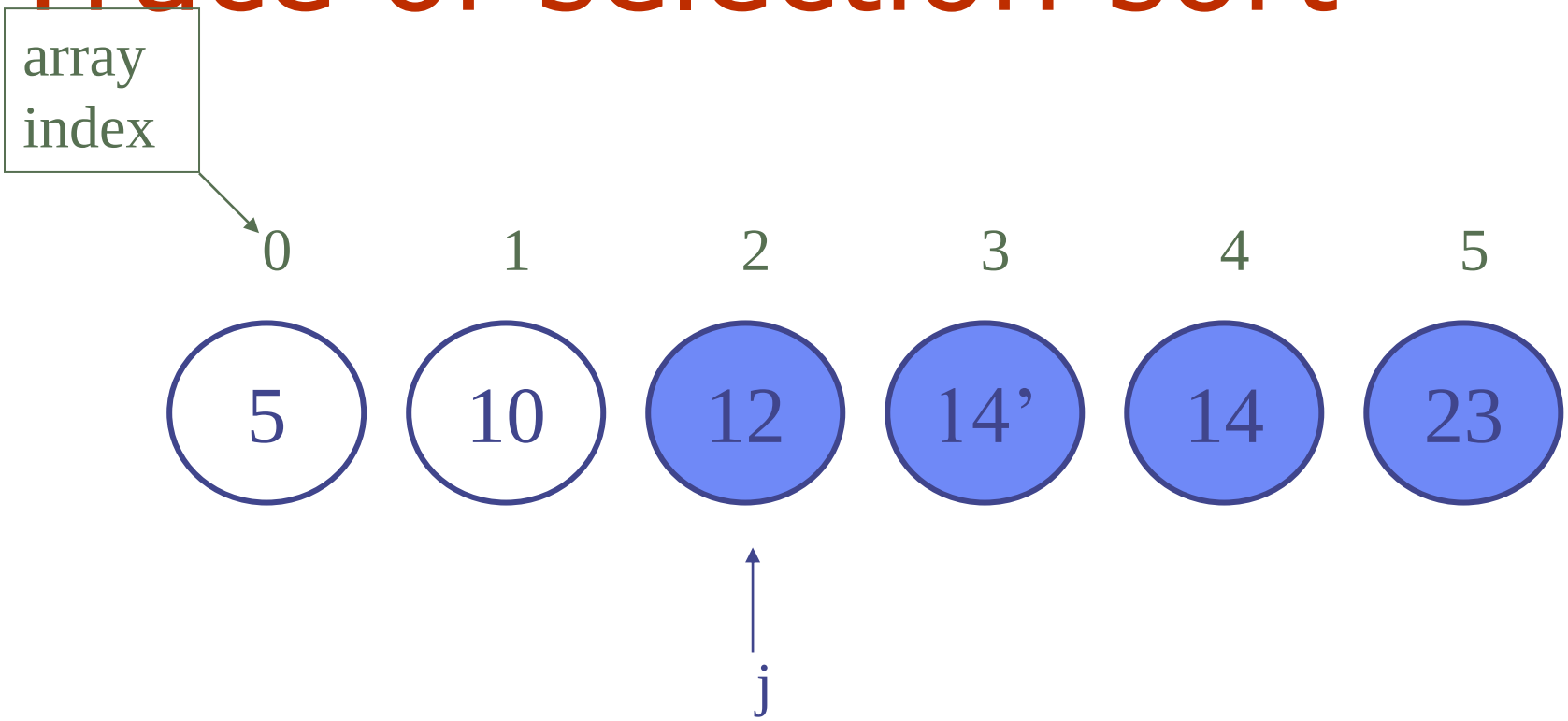
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 2, pos_greatest = 2 (changed again!)

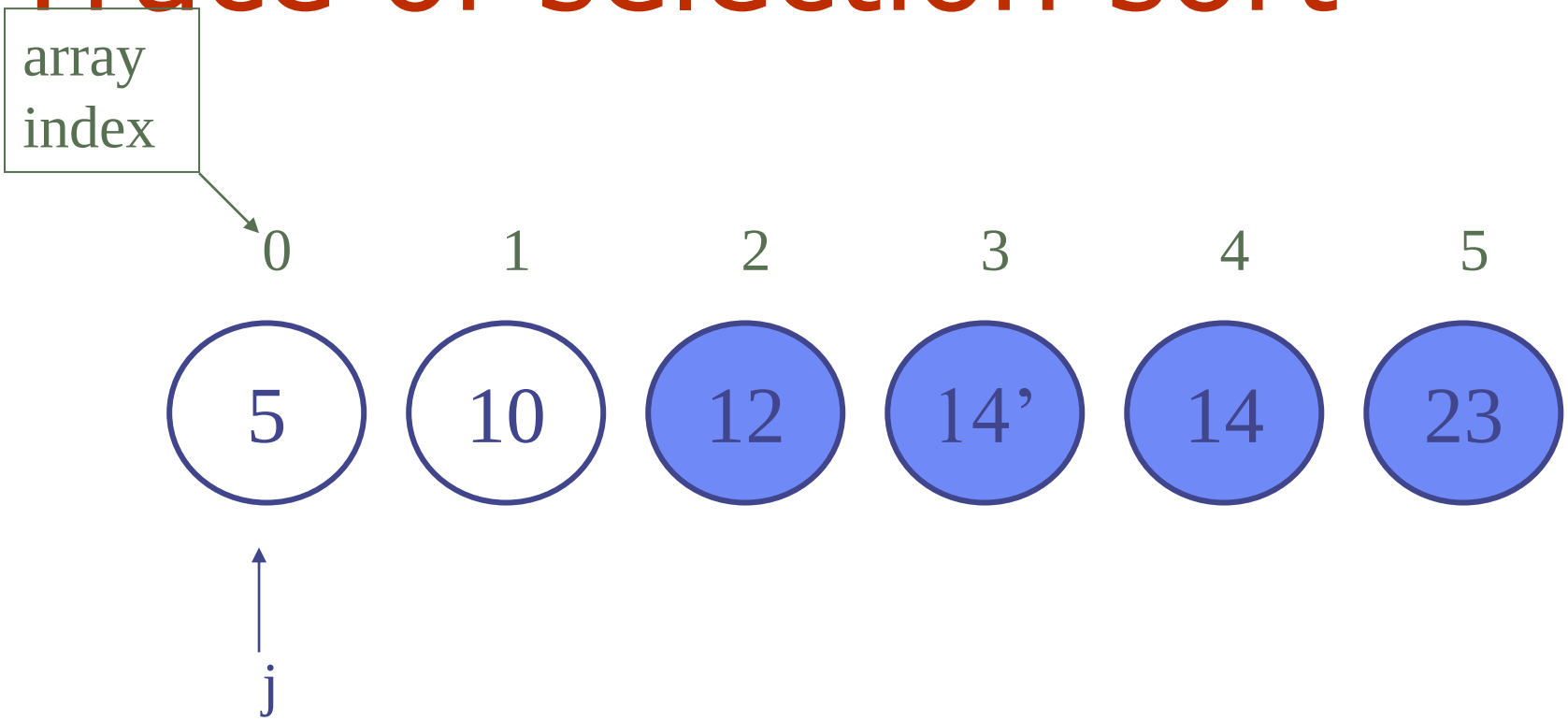
Trace of selection sort



i = 2, fourth iteration of the outer loop

swap elements at pos_greatest and 2 (element 12 with itself…)

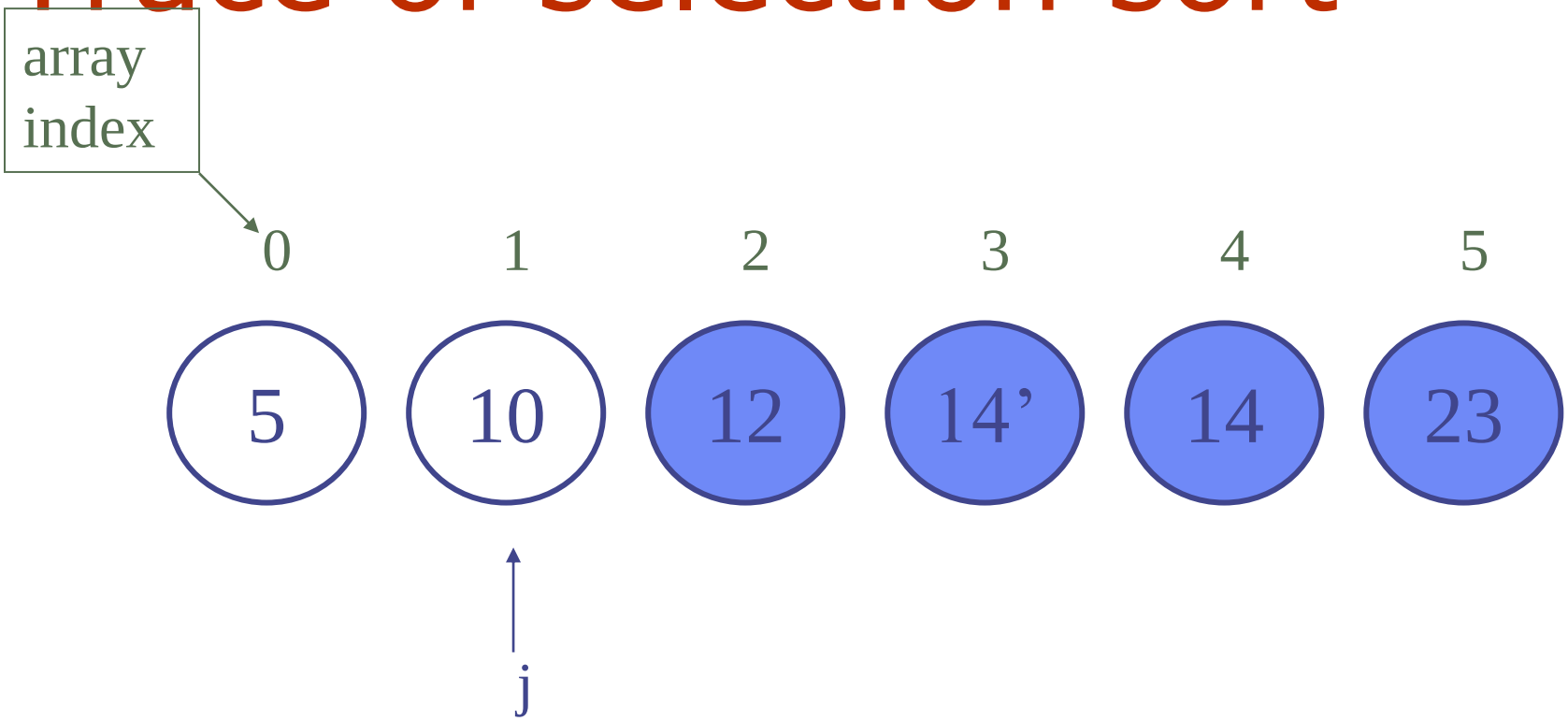
Trace of selection sort



i = 1, fifth iteration of the outer loop

j = 0, pos_greatest = 0

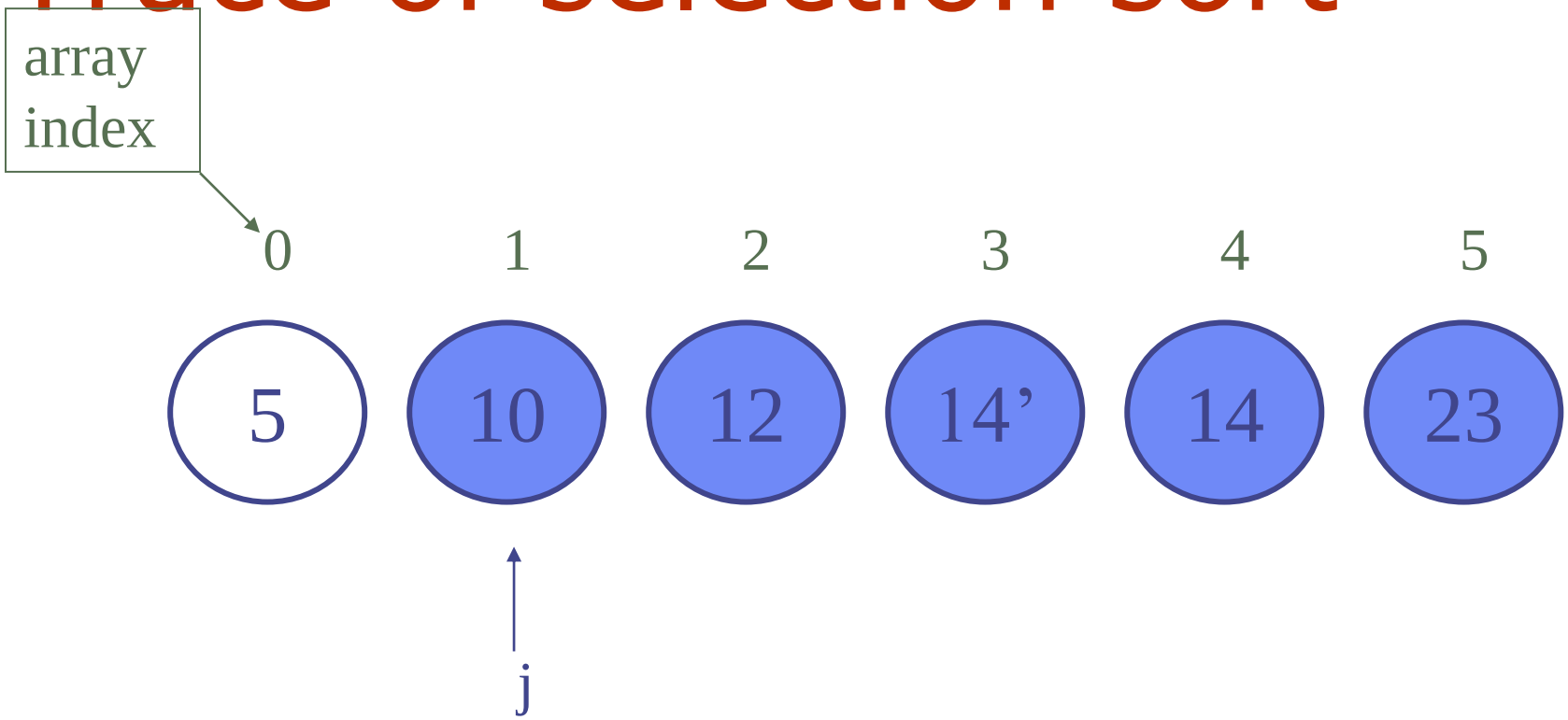
Trace of selection sort



i = 1, fifth iteration of the outer loop

j = 1, pos_greatest = 1 (changed)

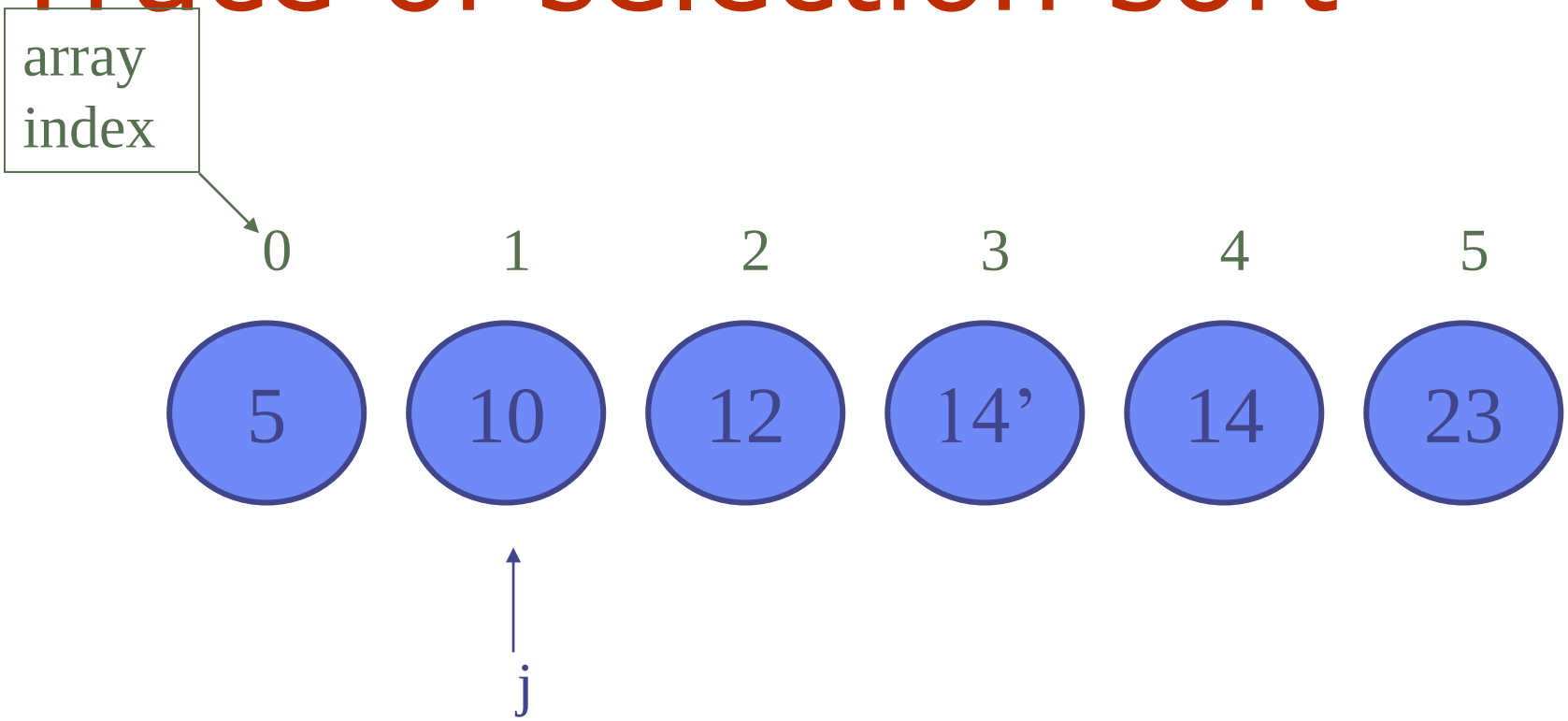
Trace of selection sort



i = 1, fifth iteration of the outer loop

swap element at pos_greatest with element at position 1 (10 with itself)

Trace of selection sort



i = 1, fifth iteration of the outer loop

NOTE: 14 and 14’ swapped ordering

This implementation of selection sort is not stable

<u>Exercise</u> (offline): can you make it stable or not?

Complexity of selection sort

Compared to bubble sort:

- Same number of iterations
- Same number of comparisons in the worst case
- fewer swaps
  (one for each outer loop = n-1)
- Hence, also $O(n^2)$

Insertion Sort: Basic Idea

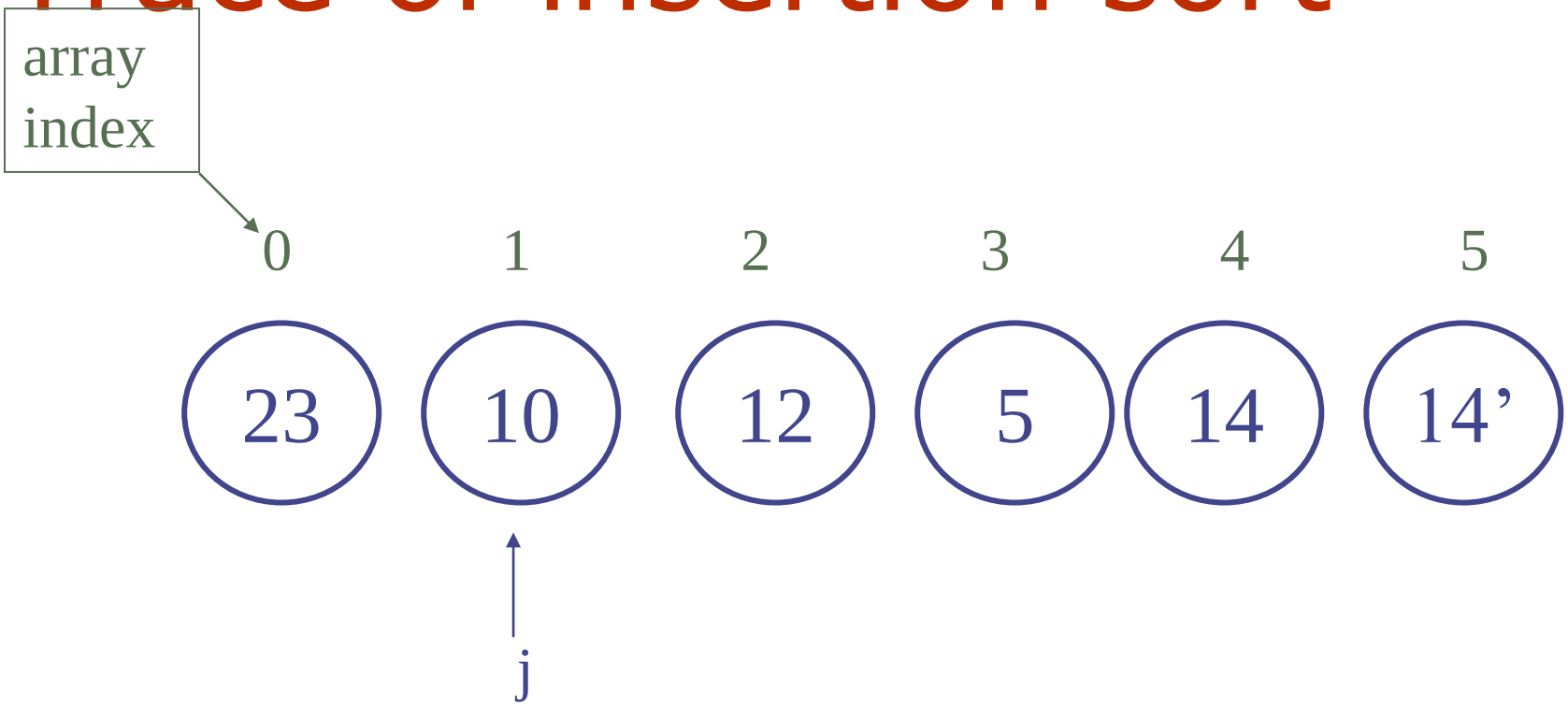
- **Keep the front of the list sorted**, and as we move through the back, elements we insert them into the correct place in the front

Insertion sort

```
void insertionSort(int arr[]){
  for(int j=1; j < arr.length; j++){
    int temp = arr[j];
    int i = j; // range 0 to j-1 is sorted
    while(i > 0 && arr[i-1] > temp){
      arr[i] = arr[i-1];
      i--;
    }
    arr[i] = temp;
  } // end outer for loop
}  // end insertion sort
```

Find a place to insert temp in the sorted range; as you are looking, shift elements in the sorted range to the right

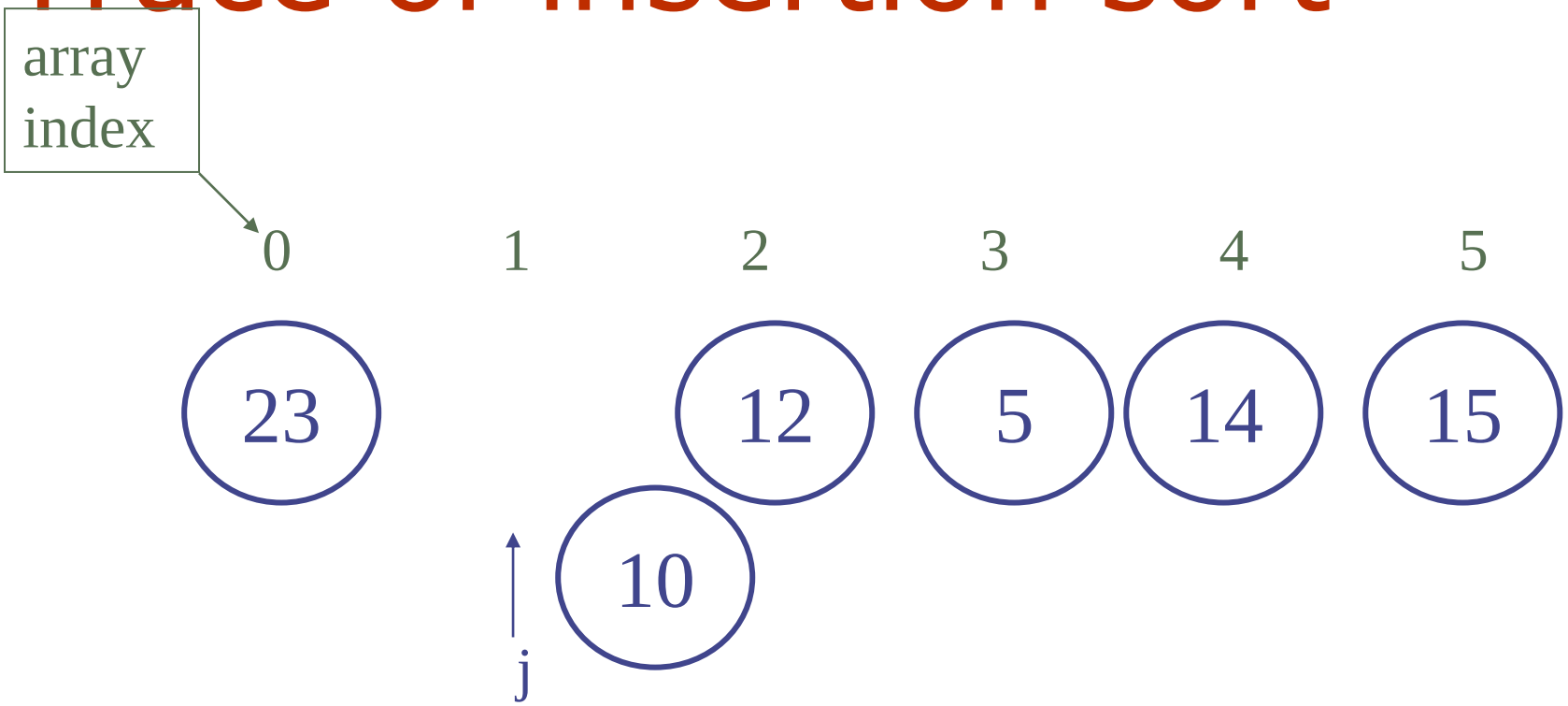
Trace of insertion sort



j = 1, first iteration of the outer loop

temp = 10; i = 1; arr[i-1] >= 10

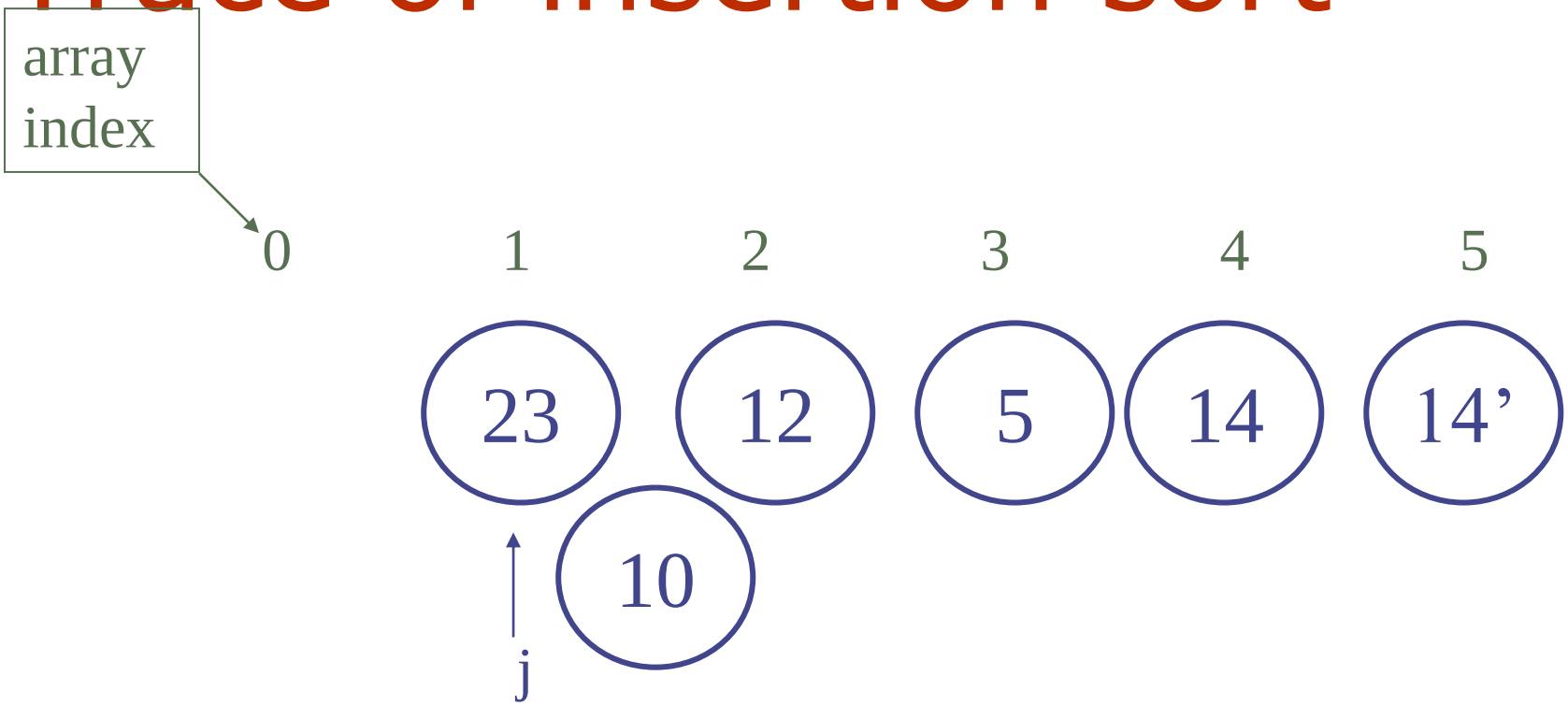
Trace of insertion sort



j = 1, first iteration of the outer loop

temp = 10; i = 1; arr[i-1] >= 10

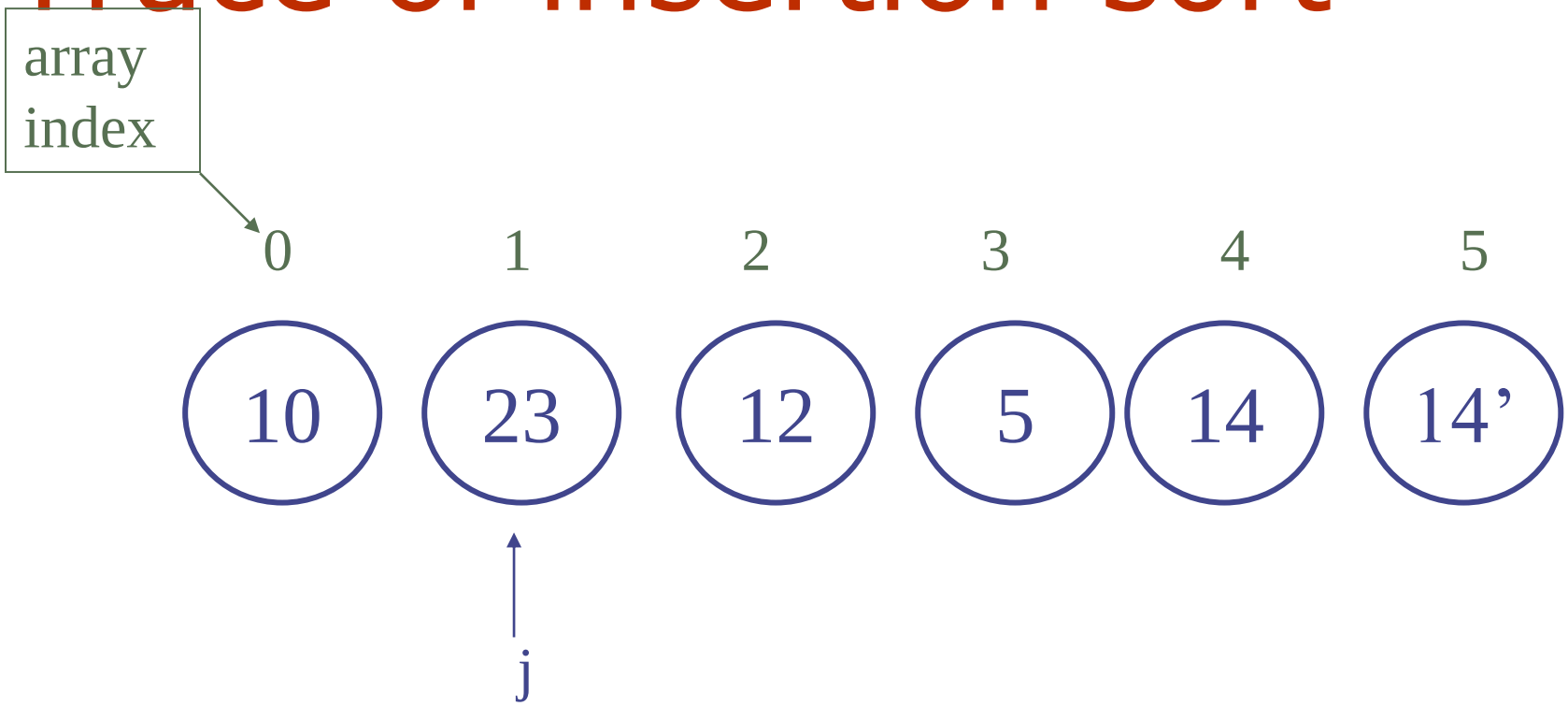
Trace of insertion sort



j = 1, first iteration of the outer loop

arr[i] = arr[i-1]

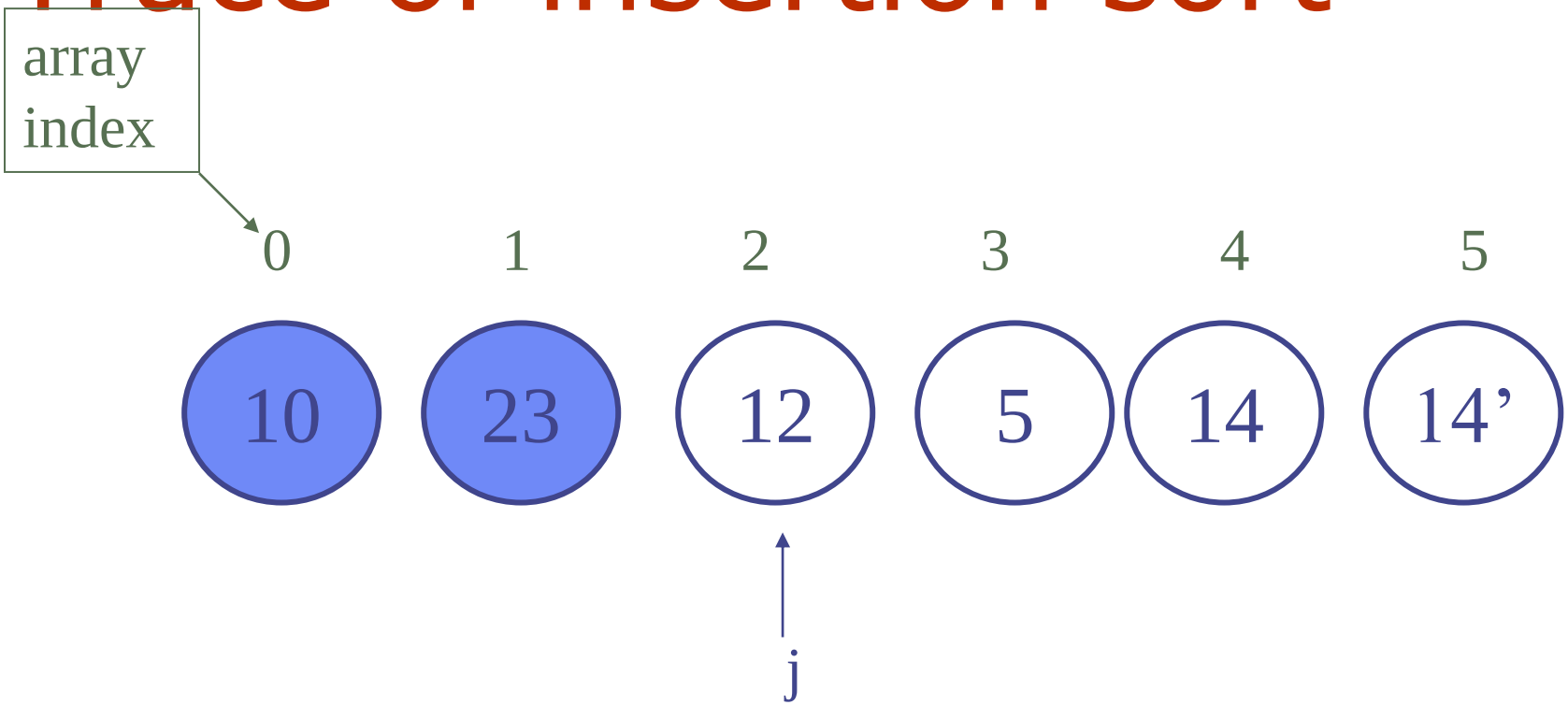
Trace of insertion sort



j = 1, first iteration of the outer loop

arr[i] = temp

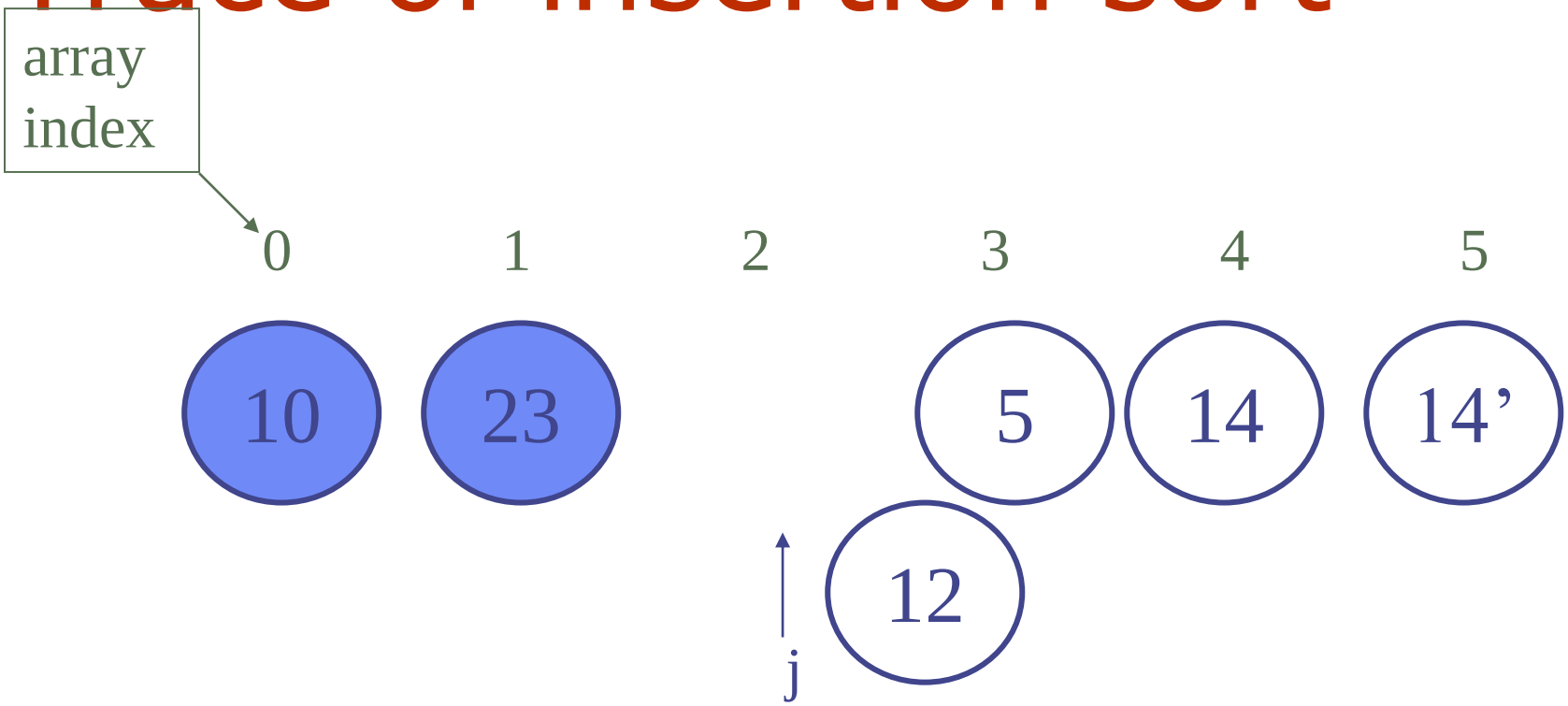
Trace of insertion sort



j = 2, second iteration of the outer loop

temp = 12; arr[i-1] >= temp

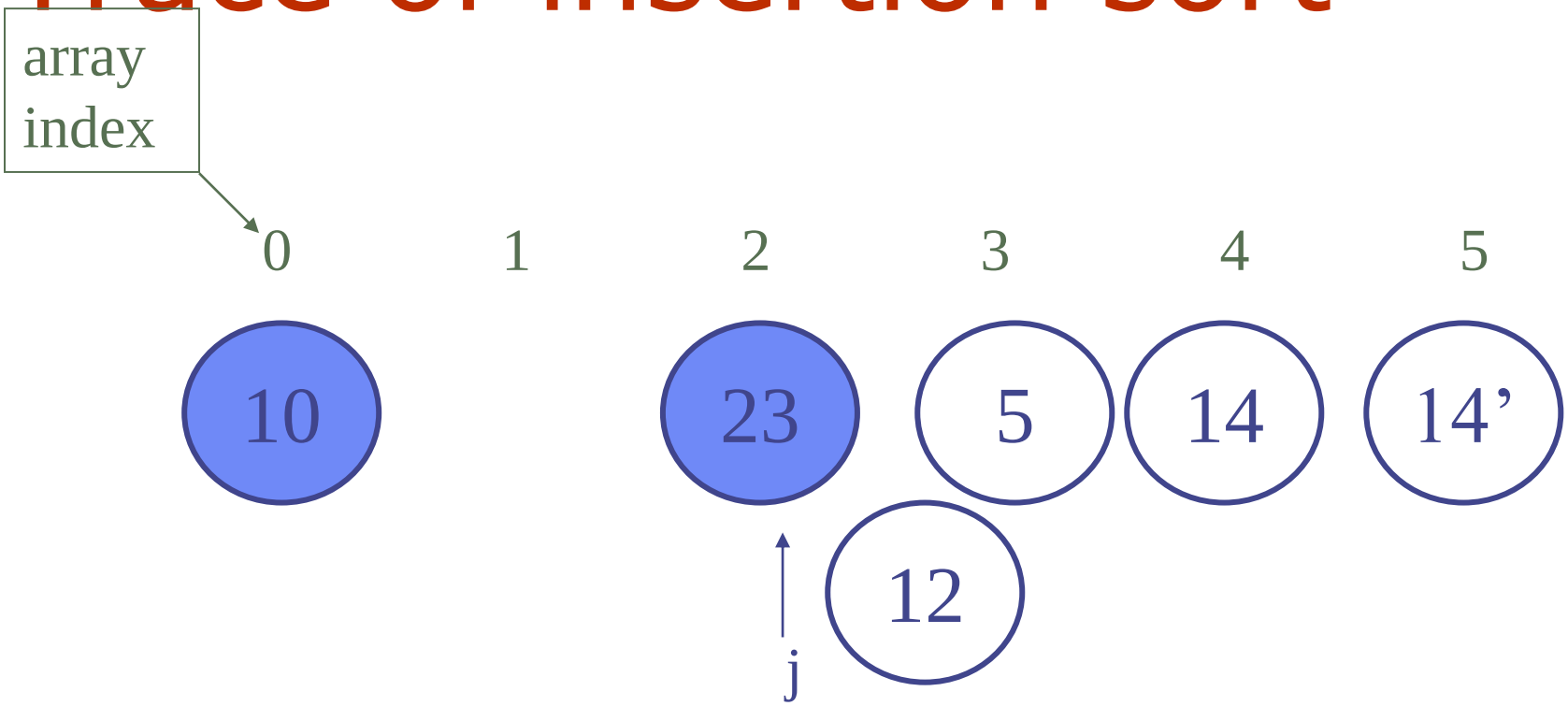
Trace of insertion sort



j = 2, second iteration of the outer loop

temp = 12; arr[i-1] >= temp

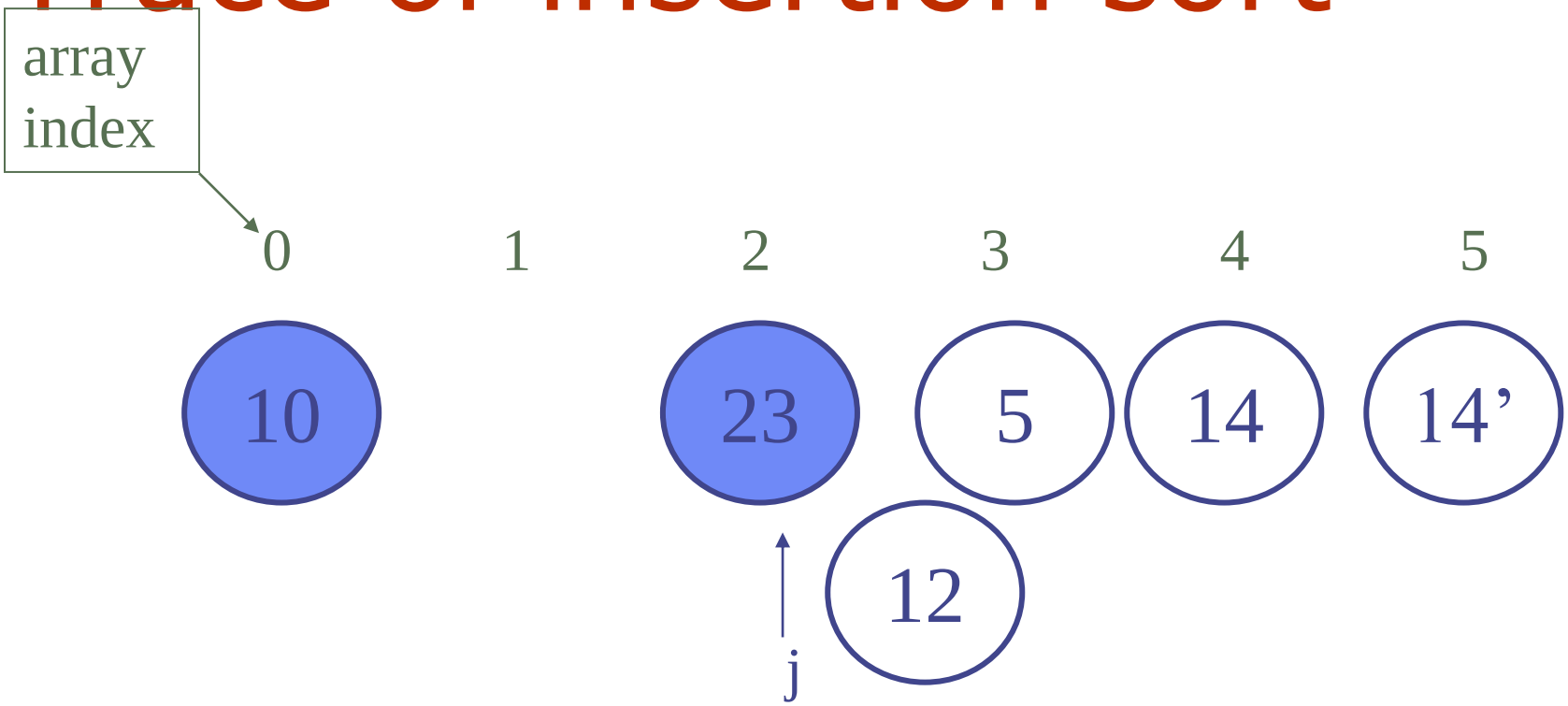
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] = arr[i]

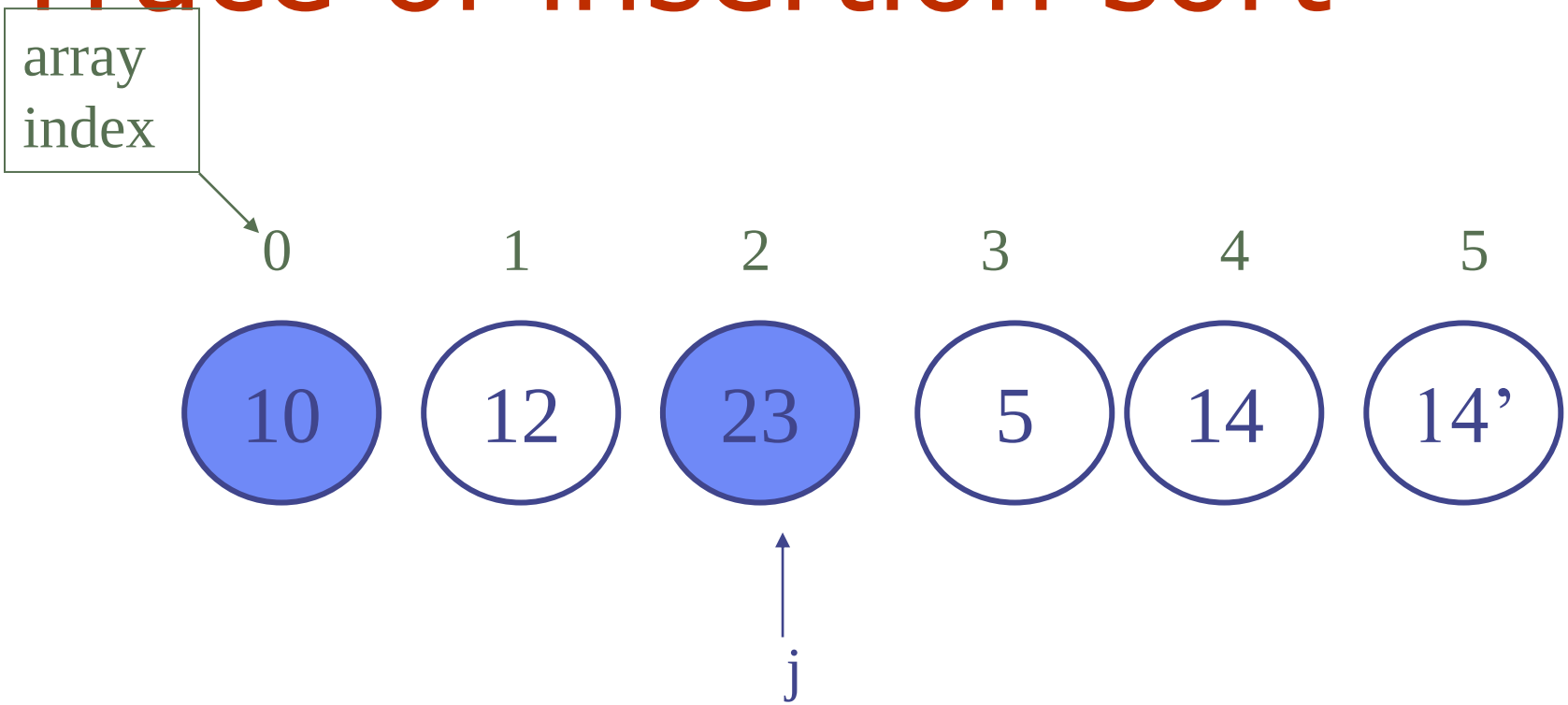
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] < temp

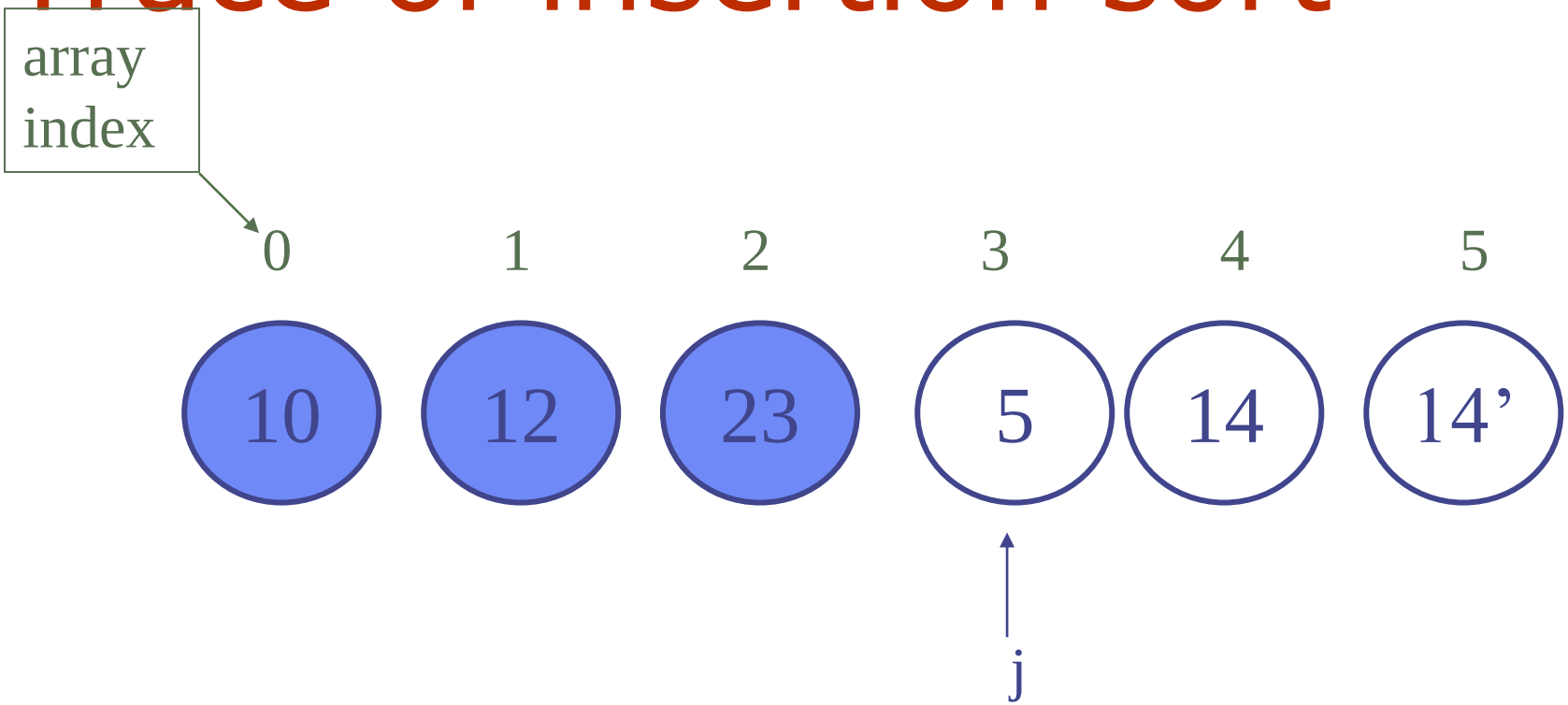
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] = temp

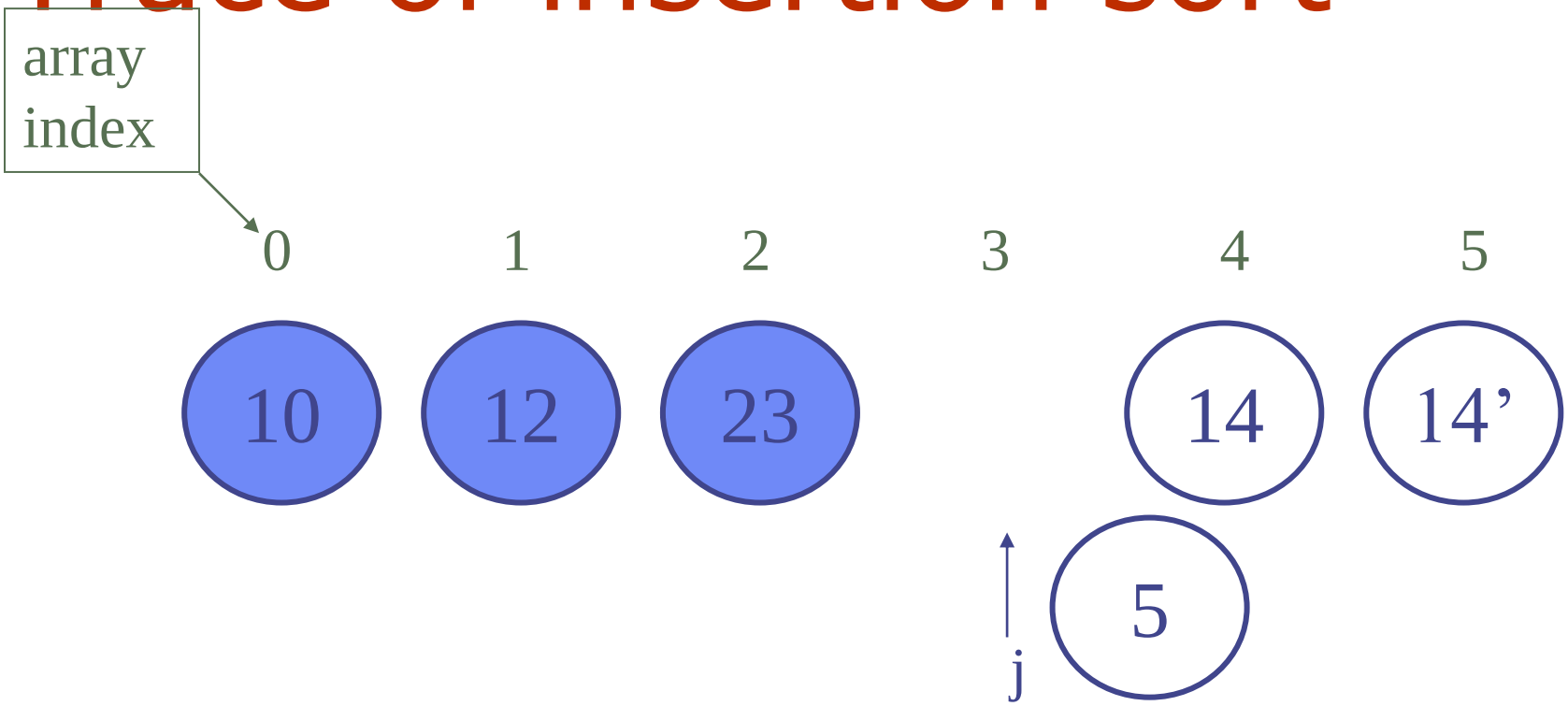
Trace of insertion sort



j = 3, third iteration of the outer loop

temp = 5

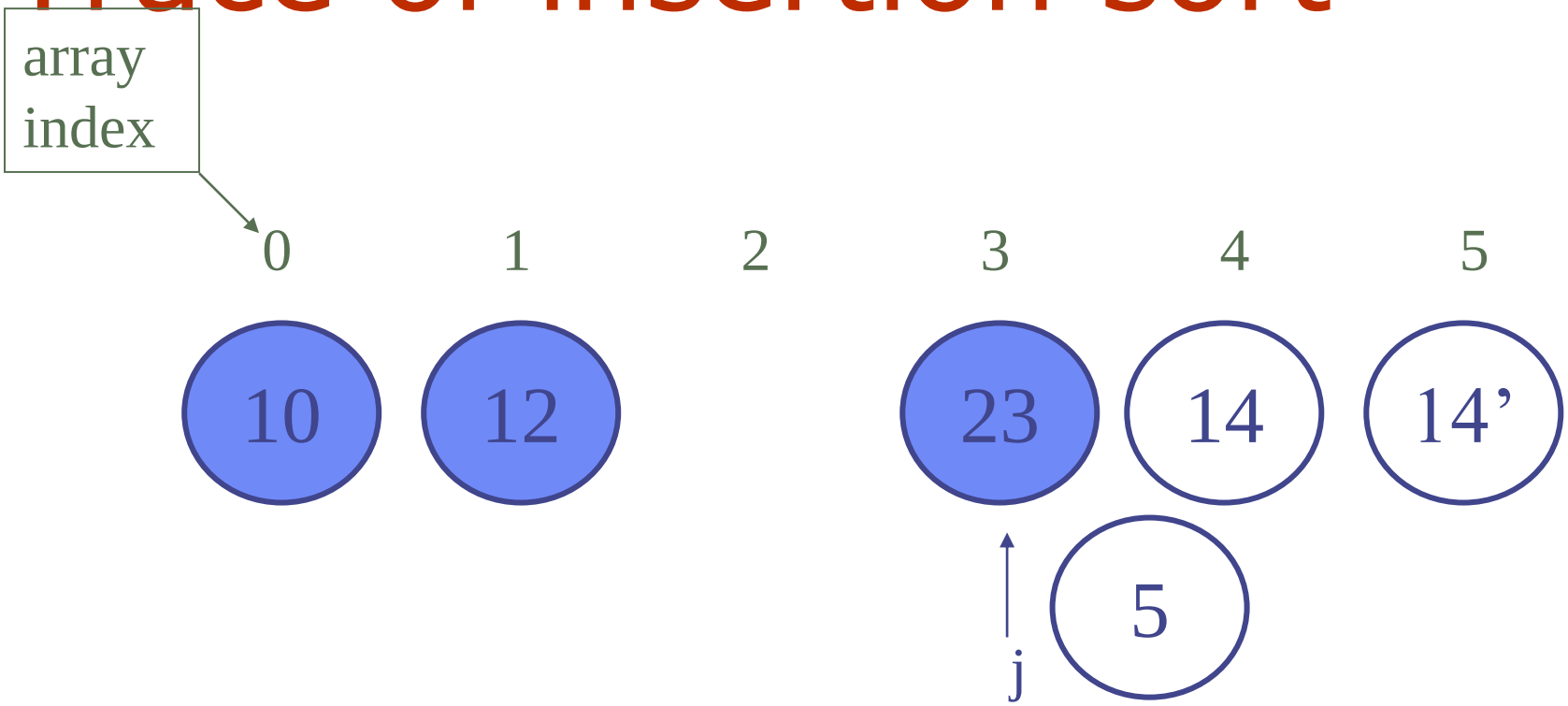
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

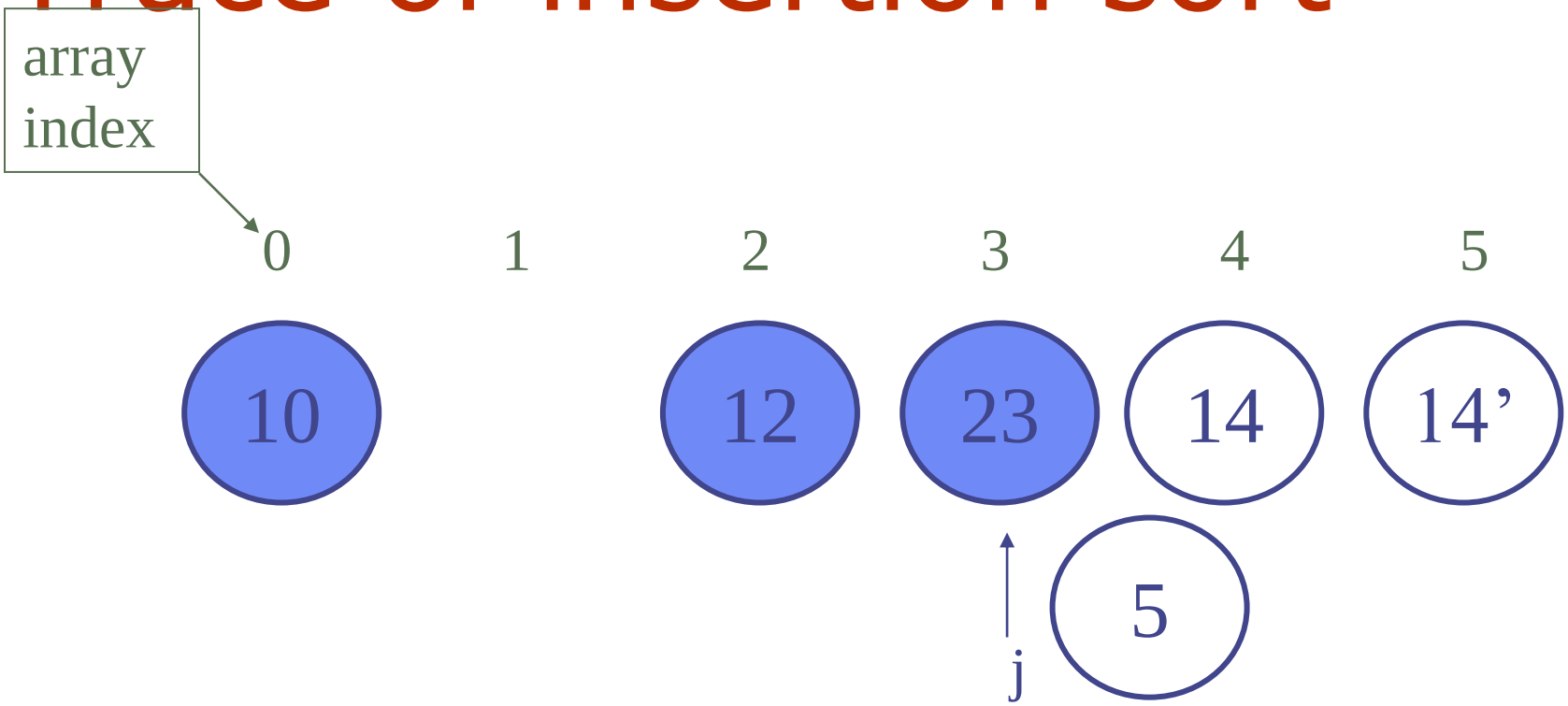
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

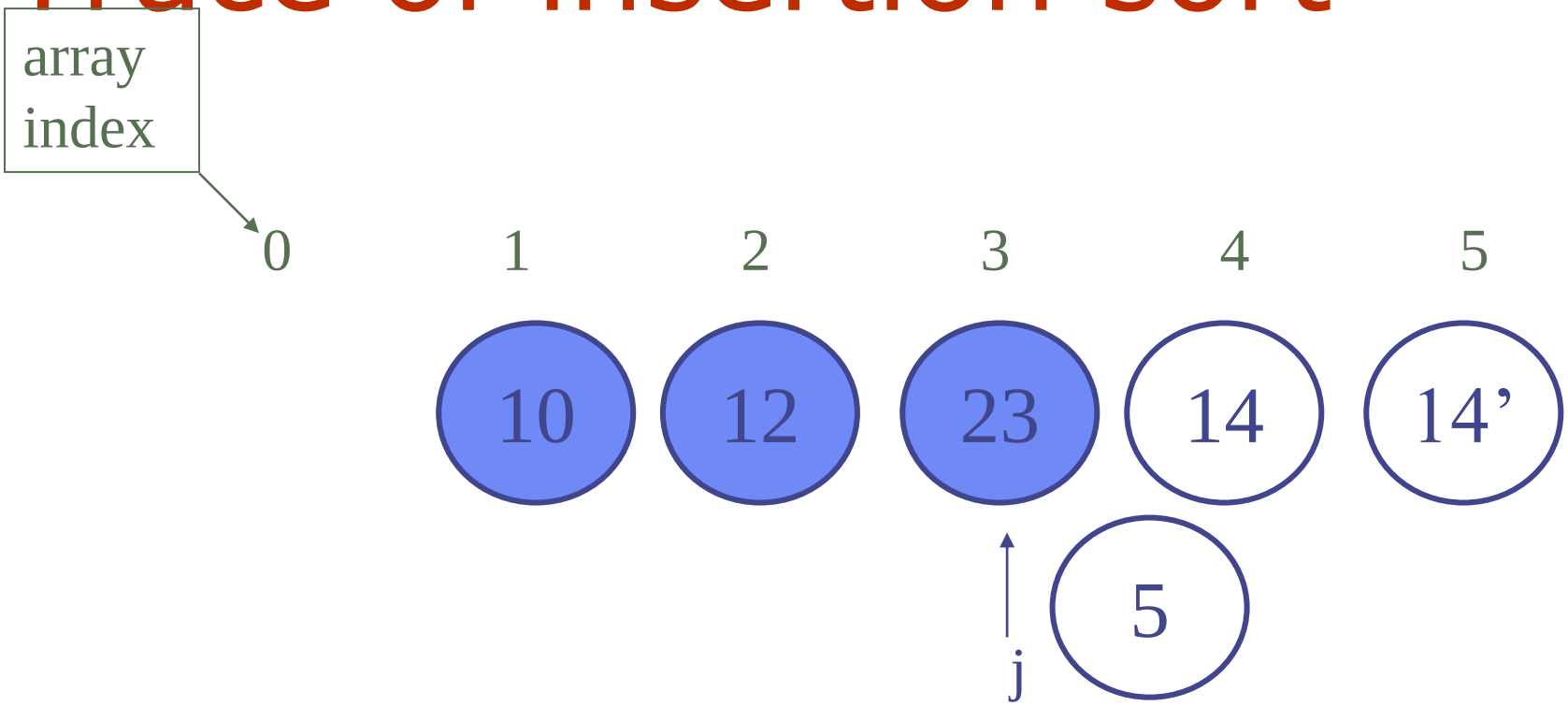
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

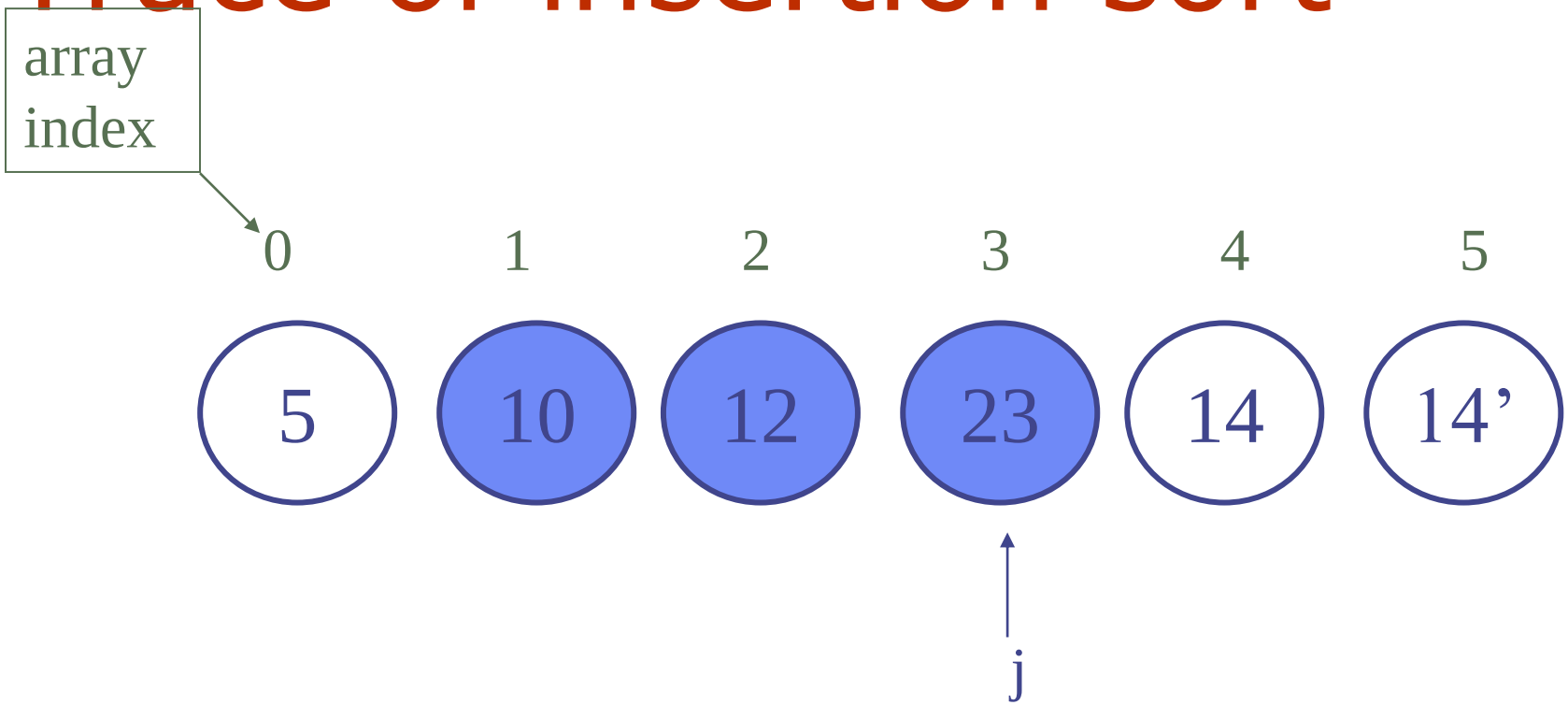
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

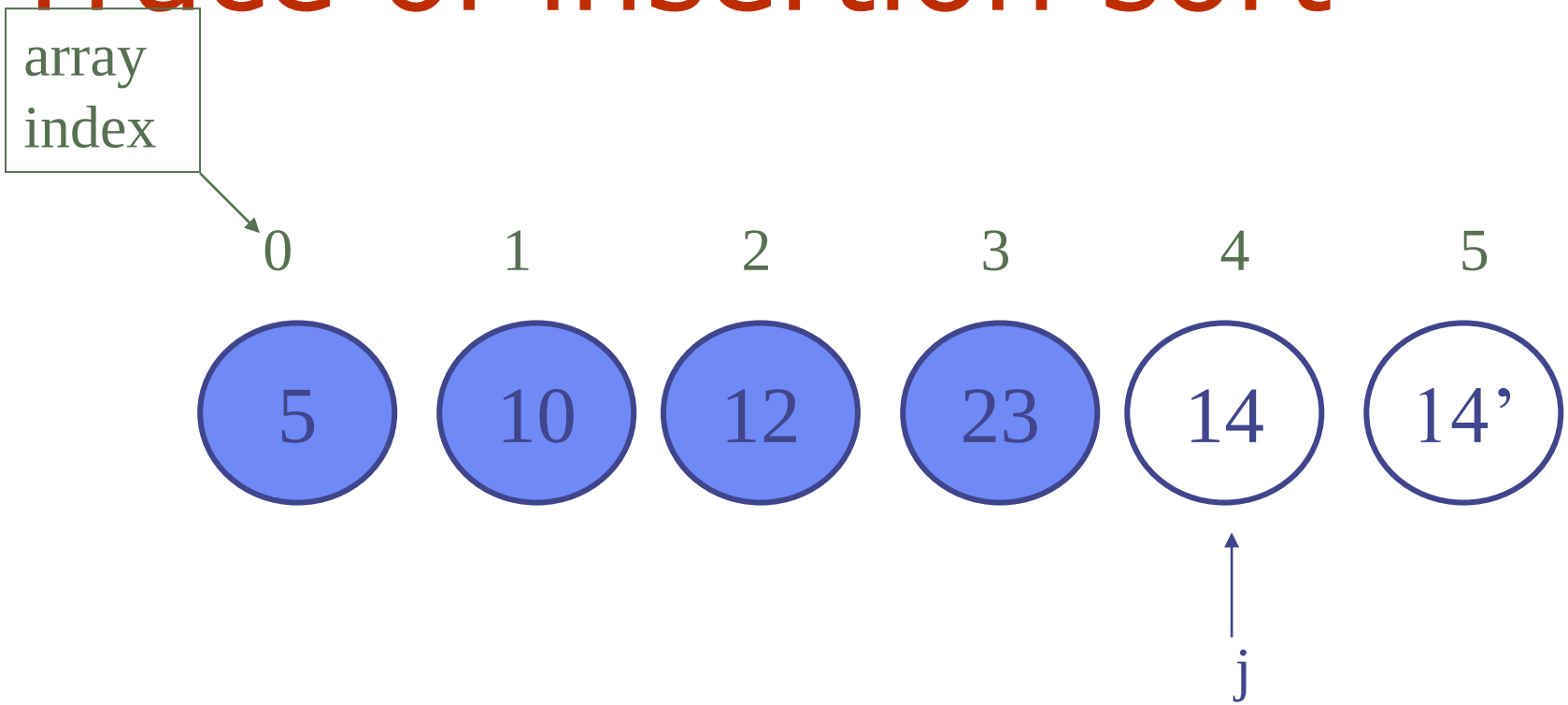
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] = temp

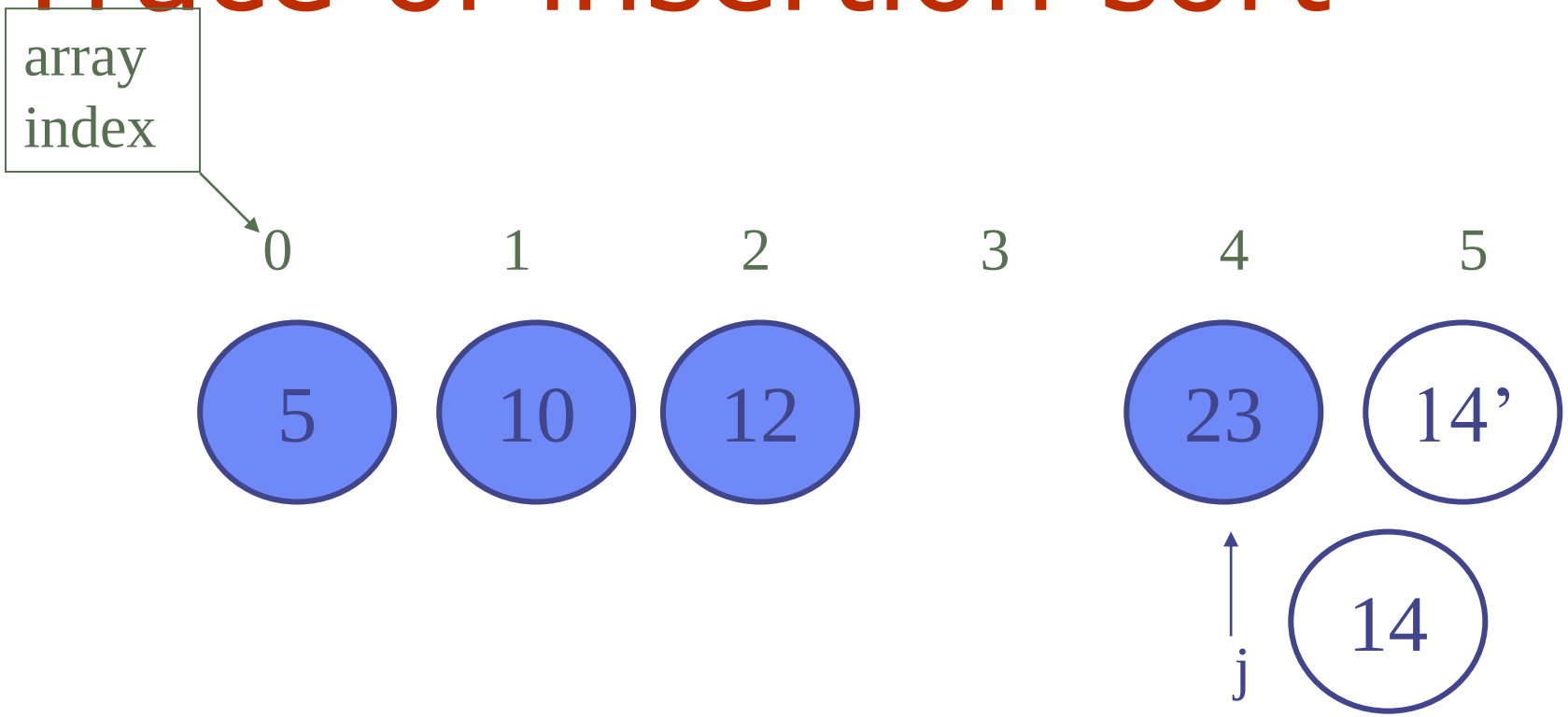
Trace of insertion sort



j = 4, fourth iteration of the outer loop

temp = 14

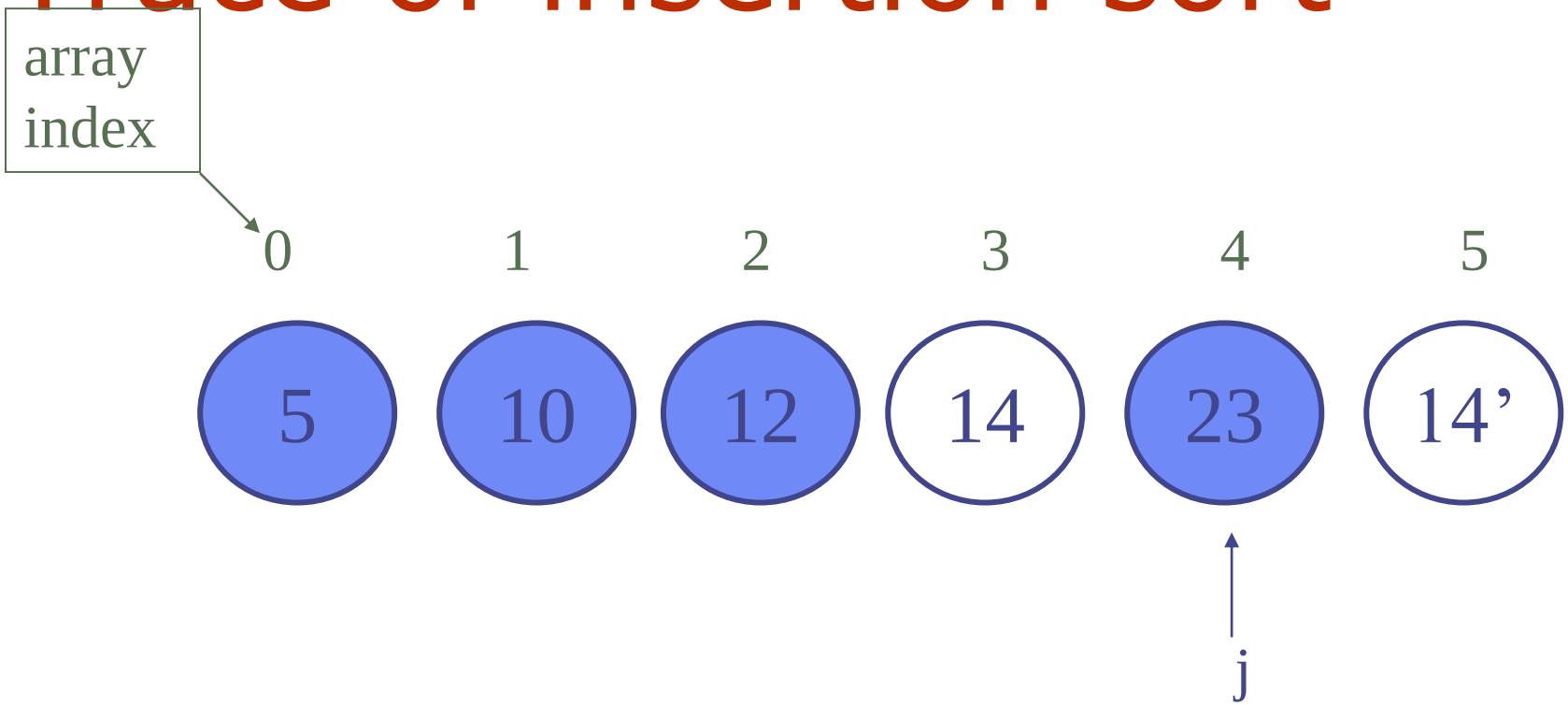
Trace of insertion sort



j = 4, fourth iteration of the outer loop

arr[i-1] >= temp

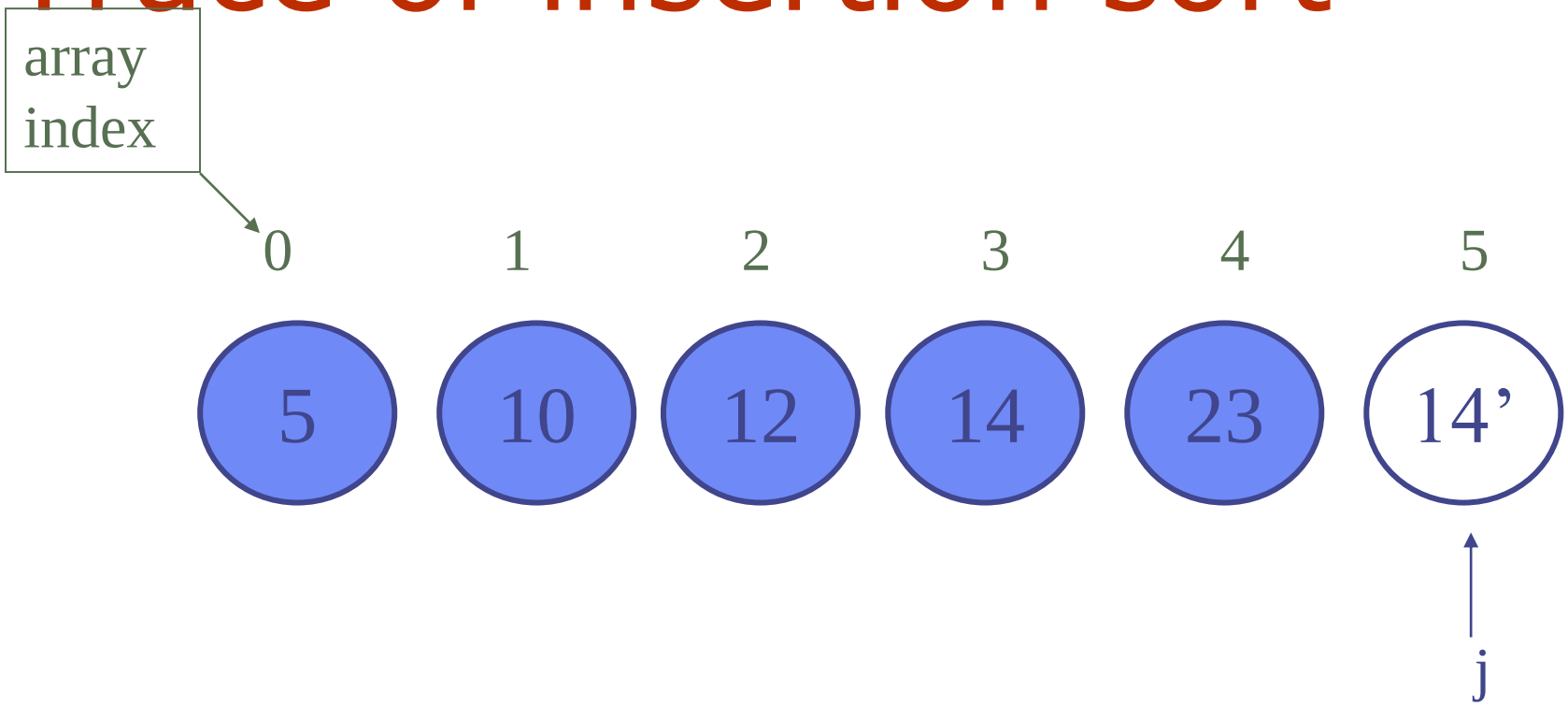
Trace of insertion sort



j = 4, fourth iteration of the outer loop

arr[i-1] = temp

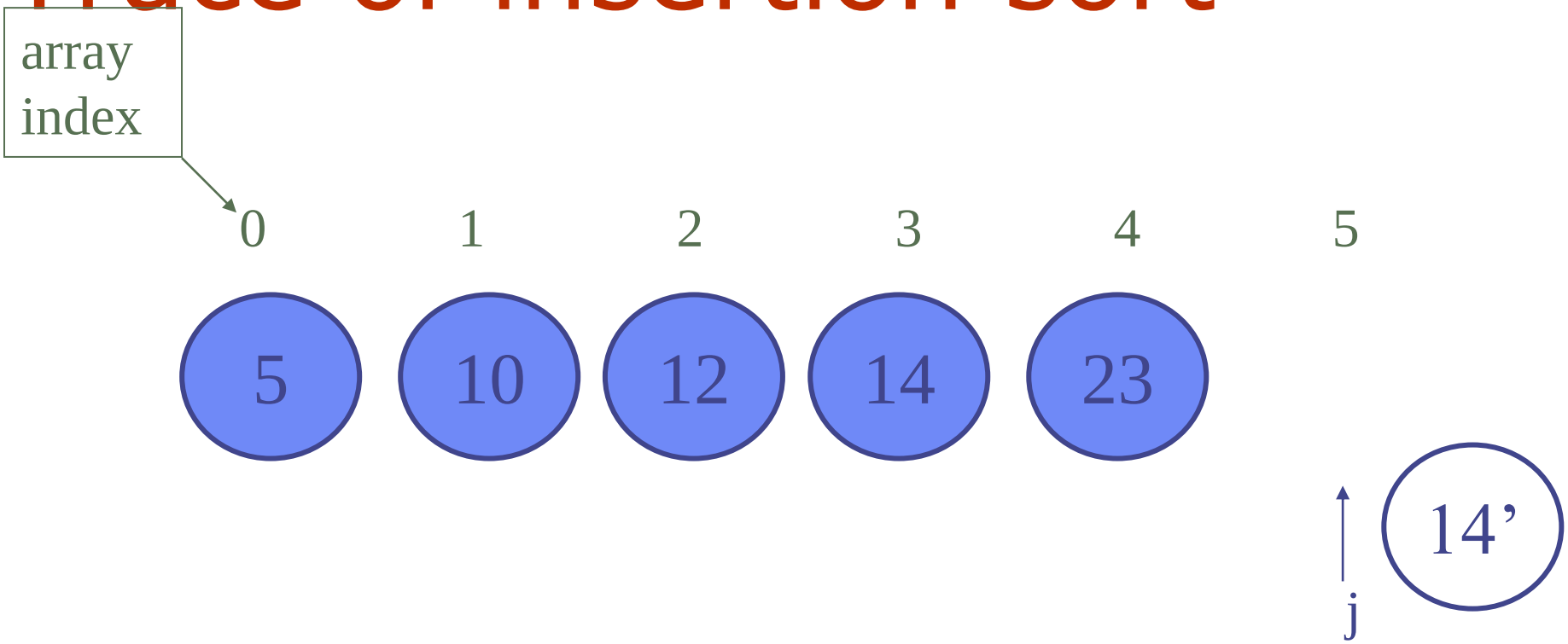
Trace of insertion sort



j = 5, fifth iteration of the outer loop

temp = 15

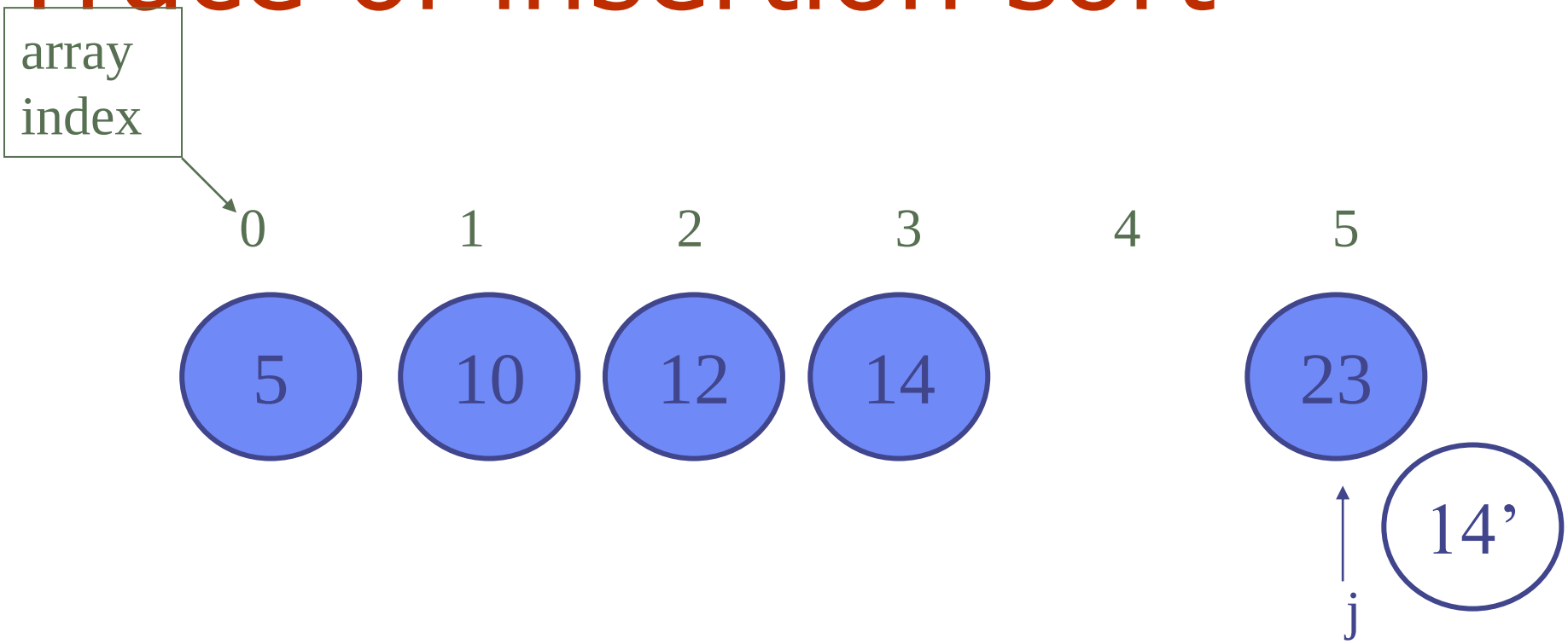
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] >= temp

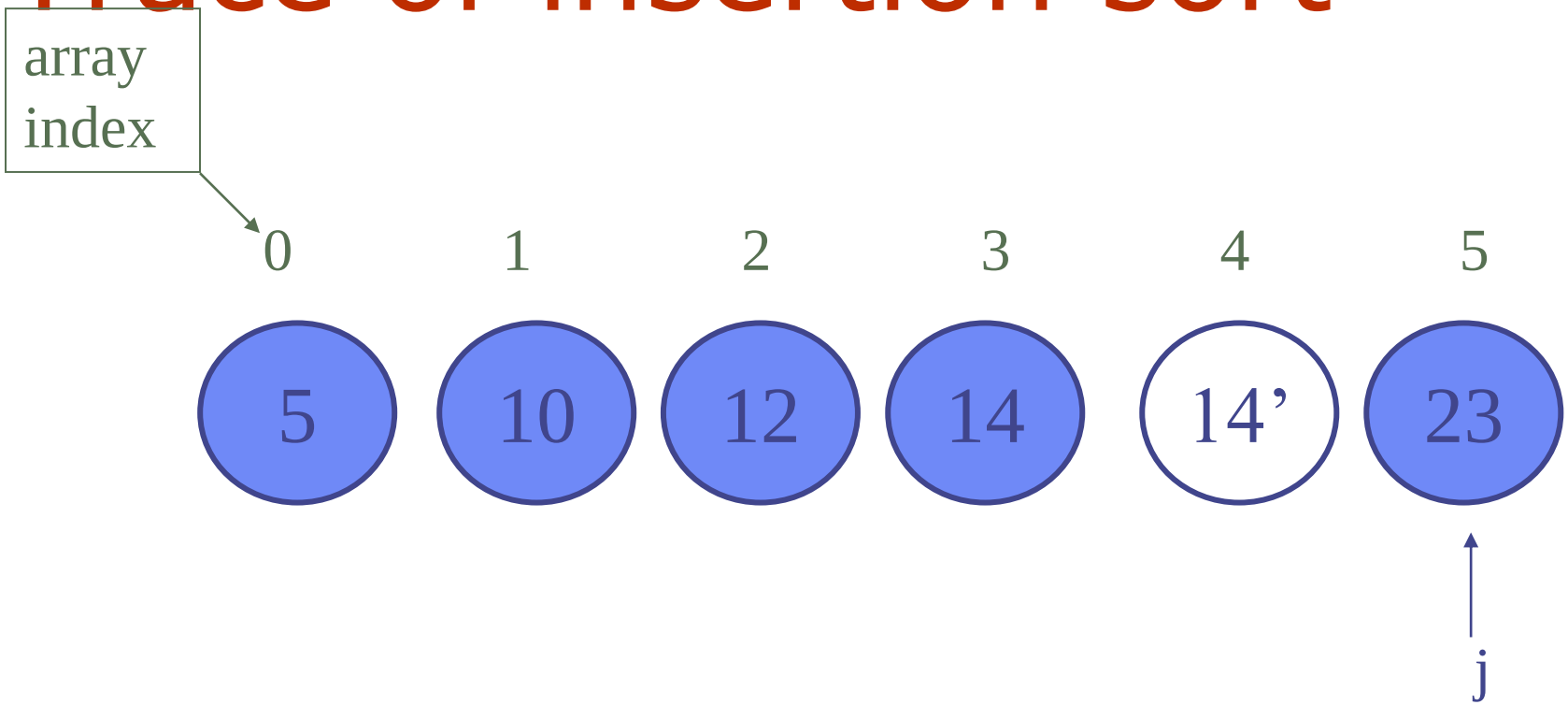
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] >= temp

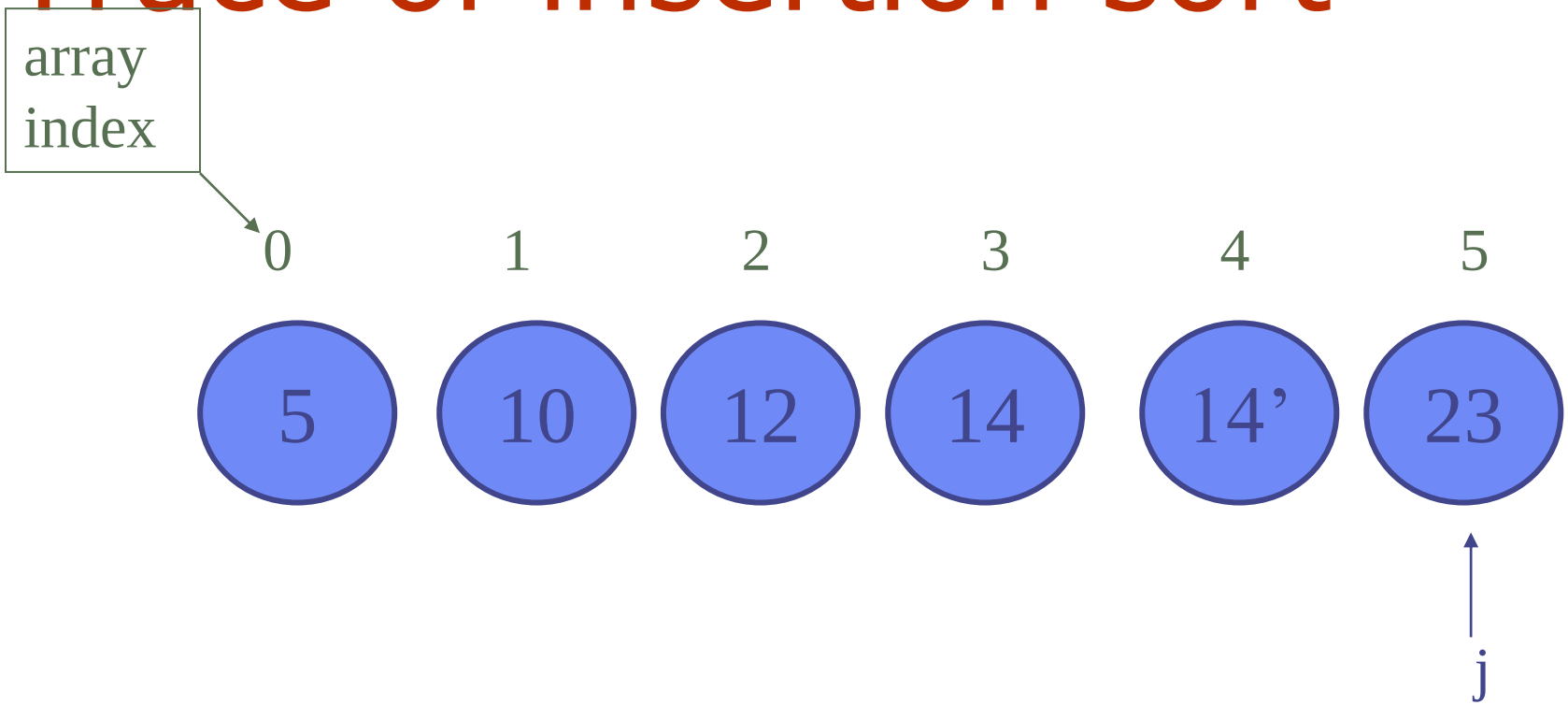
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] = temp

Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] = temp

Note: the sort was stable

Complexity of insertion sort

- In the worst case, has to make n(n-1)/2 comparisons and shifts to the right
- also $O(n^2)$ worst case complexity
- Best case: array already sorted
  - Backwards walk of inner loop stops immediately; no shifts.
  - Becomes O(n)

“Adaptive Sort”

- This is a (bizarre) name for what asking what happens to the complexity when the lists are already “nearly sorted”.
- In many applications lists might already be close to being sorted.
  - E.g. maybe they were from a list that was sorted and then some corrections were made.
  - It is then natural to ask, for each algorithm, whether the efficiency improves.
  - As a start one might consider what happens when the input list is actually already sorted?

“Adaptive Sort”

- Exercise (offline)
- For each of the 3 simple sorting methods
  - Consider how will they will perform if the data is already sorted, or nearly sorted
  - Consider how they might be modified to take advantage.
    - E.g. Suppose that each entry is guaranteed to be no more then one place from its final proper position – then can the algorithms take account of this?

Sorting on Lists

- So far have worked on arrays.

- Will these algorithms also work well on linked lists?
  - What about singly- vs. doubly-linked ?

Bubble sort of lists?

- Is bubble sort also workable for linked lists?
- Bubble sort is just as efficient (or rather inefficient) on linked lists.
  - We can easily bubble sort even for a singly linked list.

Bubble sort of singly-linked lists

- Assume we have a class Node with fields: element of type E and next of type Node.
  - (Strictly speaking getter/setter methods would be better, but this is just for the sake of brevity…)
- The List class just has head field.
- Which way are we going to traverse?
  - There is only one way we can traverse! From the head.
- What should we keep track of?
  - A "border" between the unsorted beginning part of the list, and the sorted end of the list

Bubble sort of a linked list

```
Node border = null; // first node in the sorted part
while (border != head) {
   Node current = head; // start from the first node
   while (current.next != border) {
      if (current.element > current.next.element) {
         E element v = current.element;
         current.element = current.next.element;
         current.next.element = v;
      }
      current = current.next;
   }
   border = current; // the sorted part increases by one
}
```

swap with the next node if elements out of order

Bubble sort of a linked list

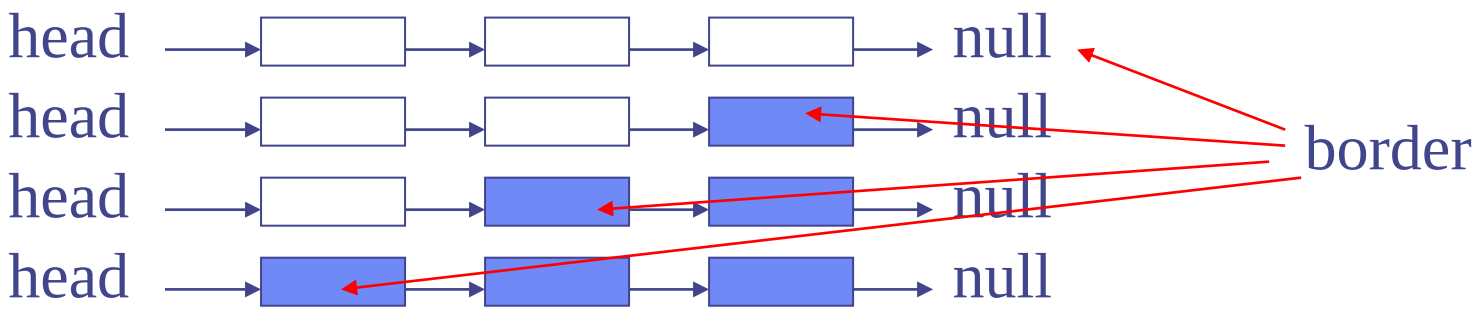
In the previous slide

```
E element v = current.element;
current.element = current.next.element;
current.next.element = v;
```

- Does a direct swap of the contents. This is fine if the element is small.
- This is usually the case as it will be a primitive data type or an object reference.
- One very rarely swaps the actual contents of objects, but usually swap the references to them.
- If necessary, then one could instead the references/pointers that give the structure of the list – that is, swap the way that the nodes are built into the list by rearranging the ‘next’ values.

Complexity of bubble sort on lists

- Same complexity as for arrays $O(n^2)$:
  - First time we iterate until we see a null (swapping elements)
  - Second time we iterate until we see the last node;
  - … each time the border is one link closer to the head of the list
  - until border == head.



Selection sort on linked lists

- Implementation similar to bubble sort; also $O(n^2)$
- Instead of pos_greatest, have a variable Node largest which keeps the reference of the node with the largest element we have seen so far
- Swap elements once in every iteration through the unsorted part of the list:

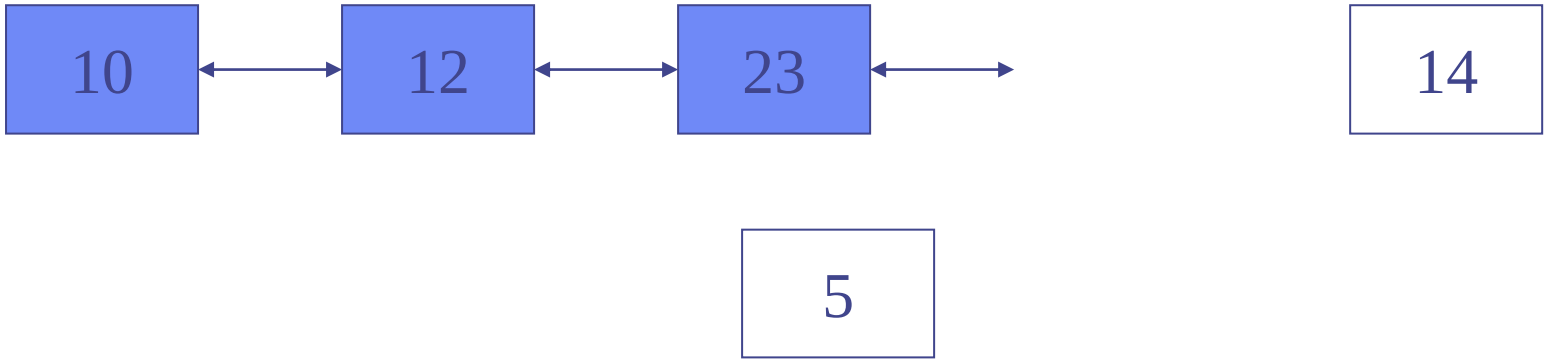
E element v = current.element; current.element =largest.element; largest.element= v;

Insertion sort on linked lists

- This is a suitable sorting method (only) for **doubly** linked lists – as need to walk backwards
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):

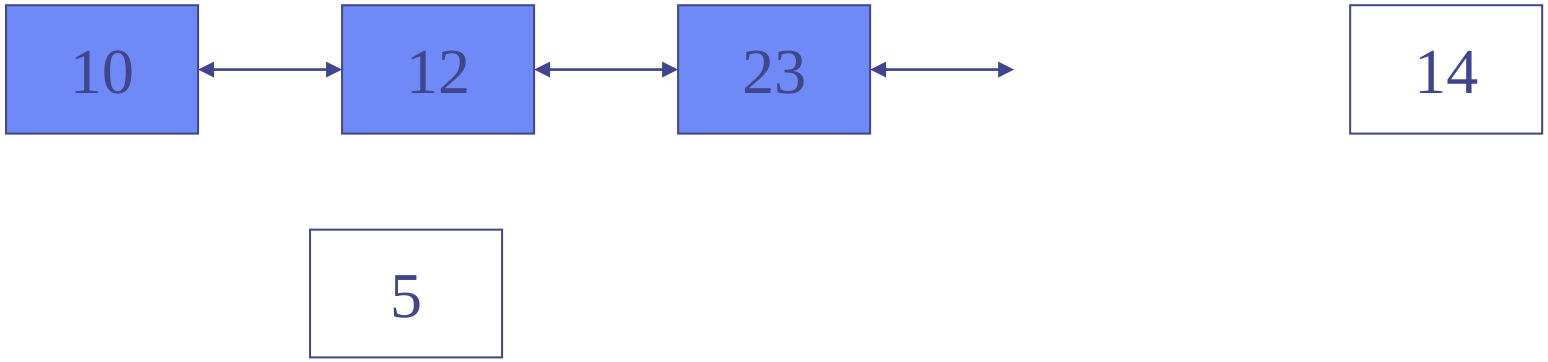
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



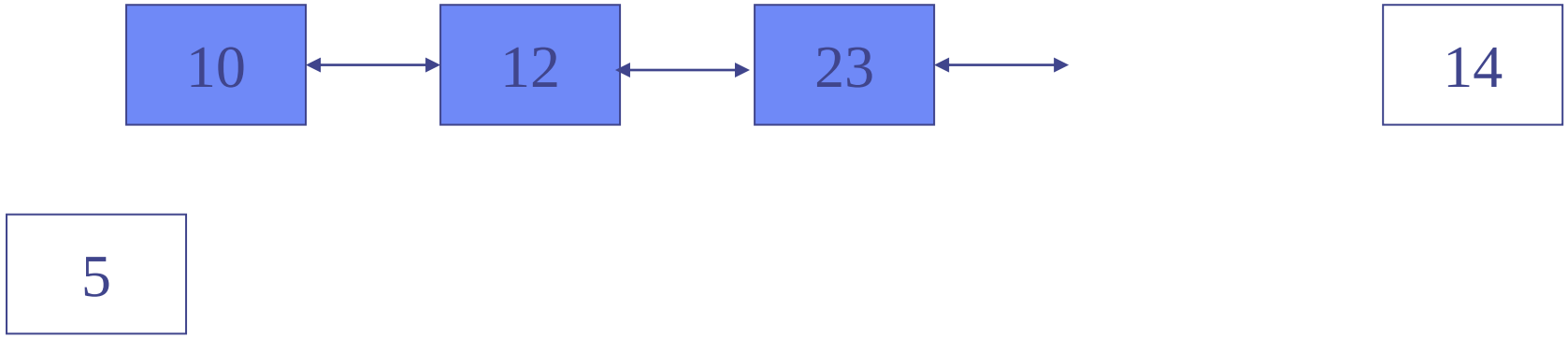
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



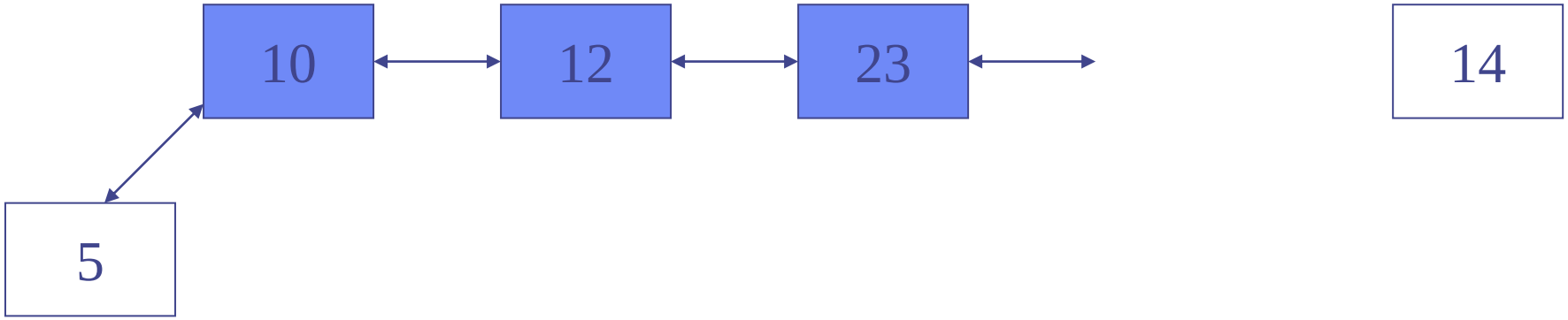
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



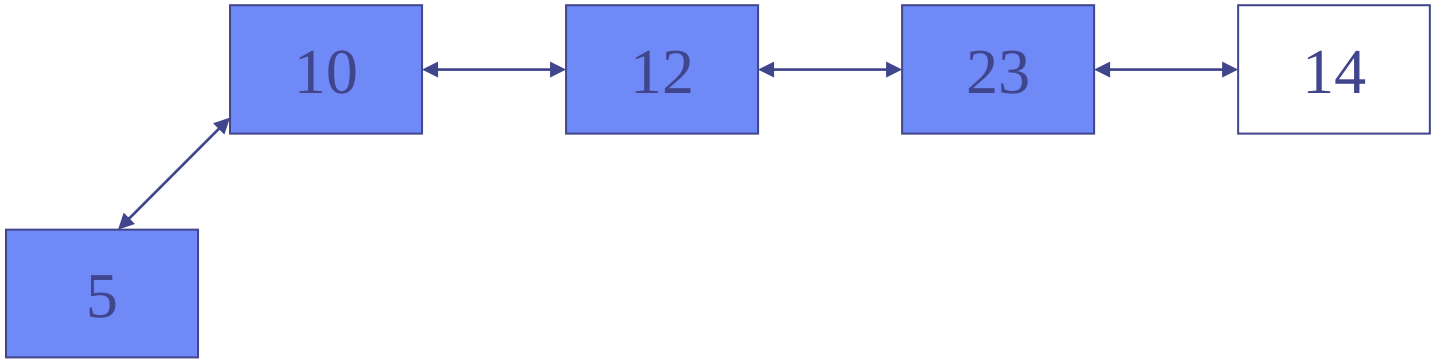
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



Exercise (offline)

- Implement all the methods.
- Good practice at coding ☺

Expectations

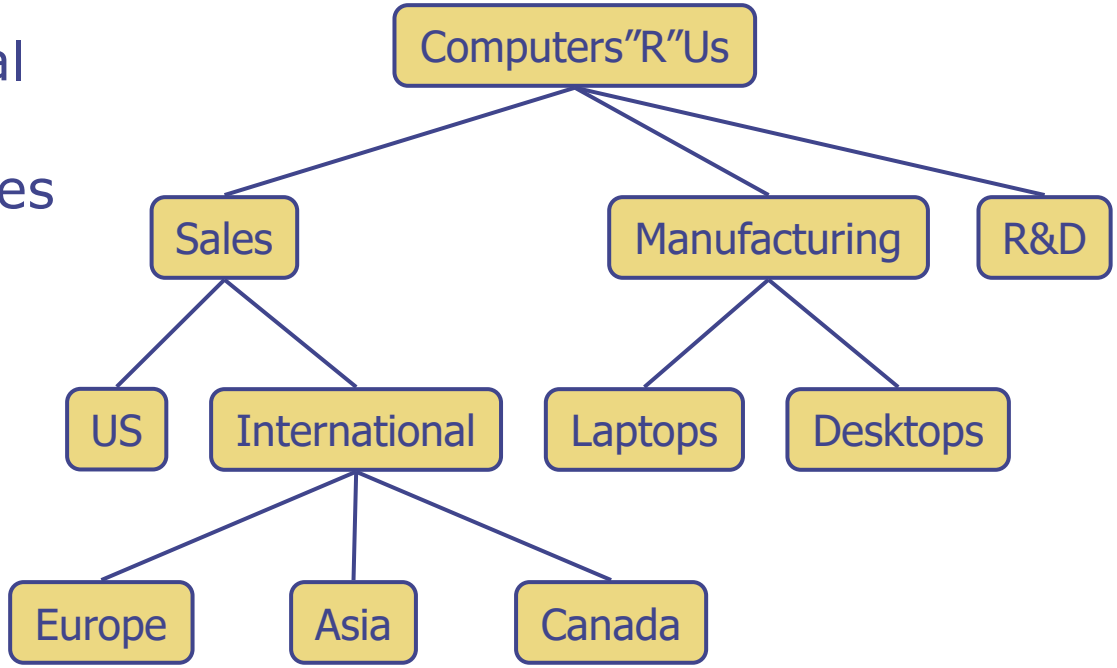
- Know these simple sorting algorithms
  - be able to implement them
  - recognise them
  - be able to analyse their complexity
    - And roughly what happens to the complexity if the input is already (nearly) sorted
- Understand the meaning of "stable" in the context of sorting
- Understand the idea of "adaptive sort" and performance on lists that are already partially sorted.

COMP2054-ADE
Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Trees : Terminology, Traversals, Representations, and Properties

What is a (Rooted) Tree

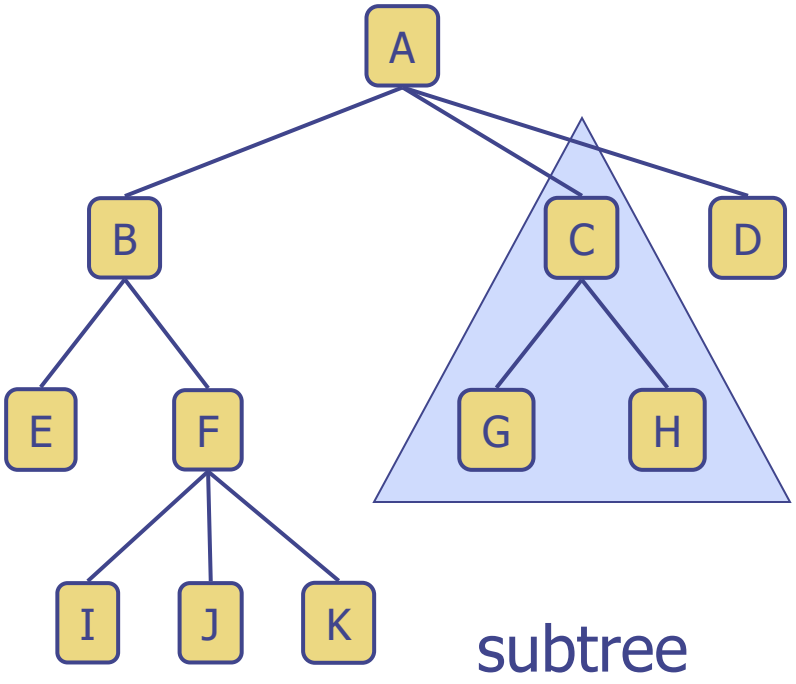
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation (at most one parent!)
- Applications:
  - Organization charts
  - File systems
  - Programming environments
- **As data structures**
  - **Heaps**
  - **Search trees**



Tree Terminology

- **Root**: node without parent (A)
- **Internal** node: node with at least one child (A, B, C, F)
- **External** node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Depth** of a node: number of ancestors (not counting itself)
- **Height** of a tree: maximum depth of any node = length of longest path from root to a leaf
  - Height of tree on right = 3
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.

- **Subtree**: tree consisting of a node and its descendants



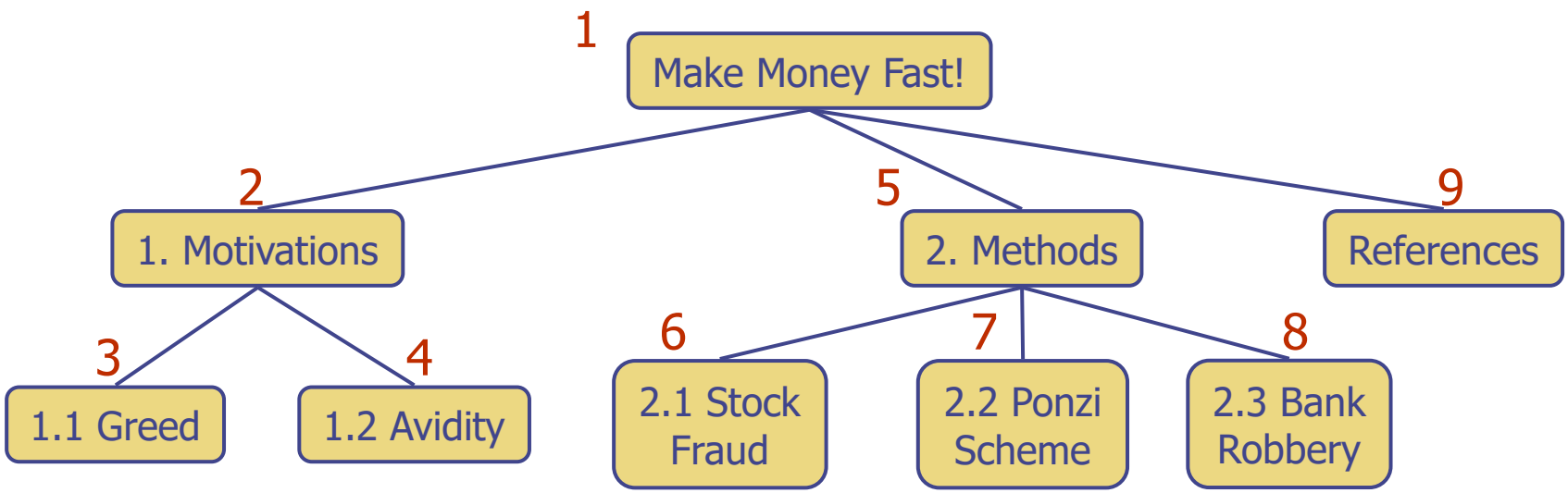
Traversals

- Given a data structure, a common task is to traverse all elements
  - visit each element precisely once
  - visit in some systematic and meaningful order
  - Note “visit” means “process the contents” but does not include just “passing through using the links”
- For an array, or linked list, the natural way is a forwards scan
  - For trees there are more options:

Preorder Traversal

- In a preorder traversal, a node is visited **before** its descendants
- Application: print a structured document

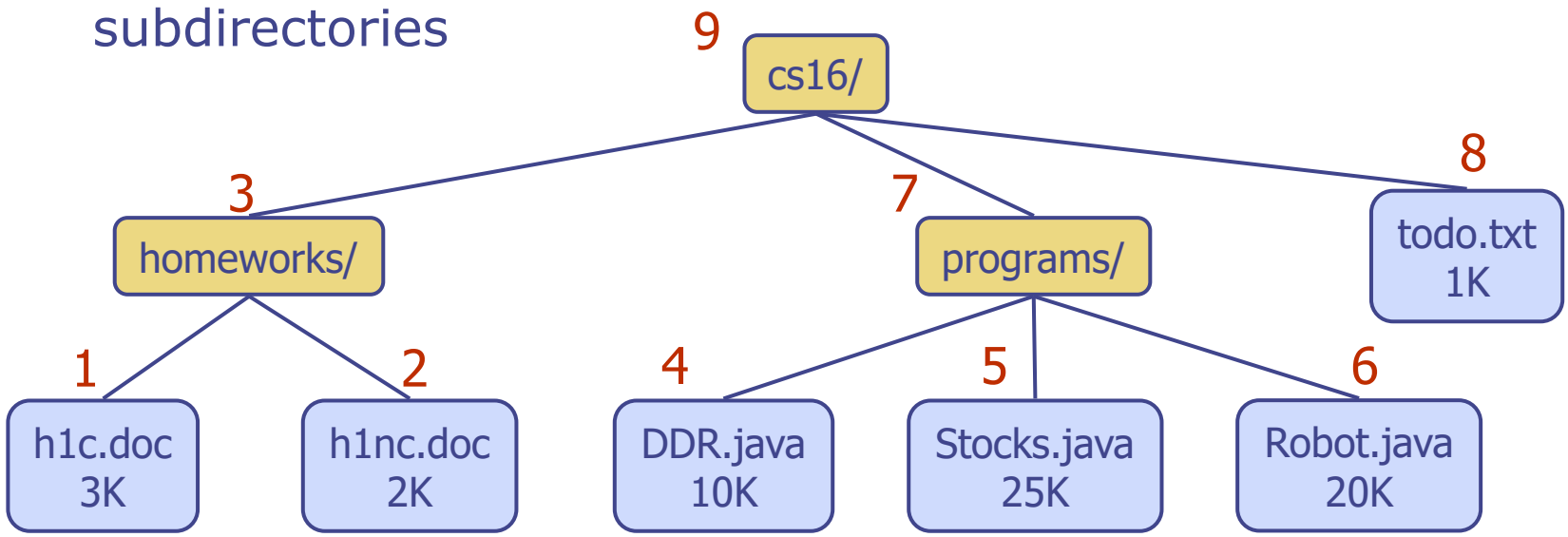
Algorithm ***preOrder***(***v***)
  visit(***v***)
  for each child ***w*** of ***v***
    preorder (***w***)



Postorder Traversal

- In a postorder traversal, a node is visited **after** its descendants
- Application: compute space used by files in a directory and its subdirectories

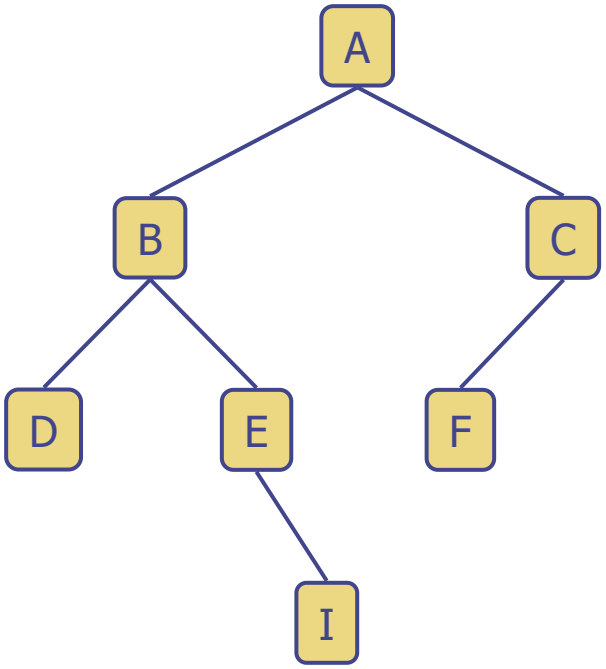
Algorithm ***postOrder(v)***
  for each child ***w*** of ***v***
    postOrder (***w***)
  visit(v)



Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair - though one might be "missing"
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of "children", each of which is missing (a null) or is the root of a binary tree

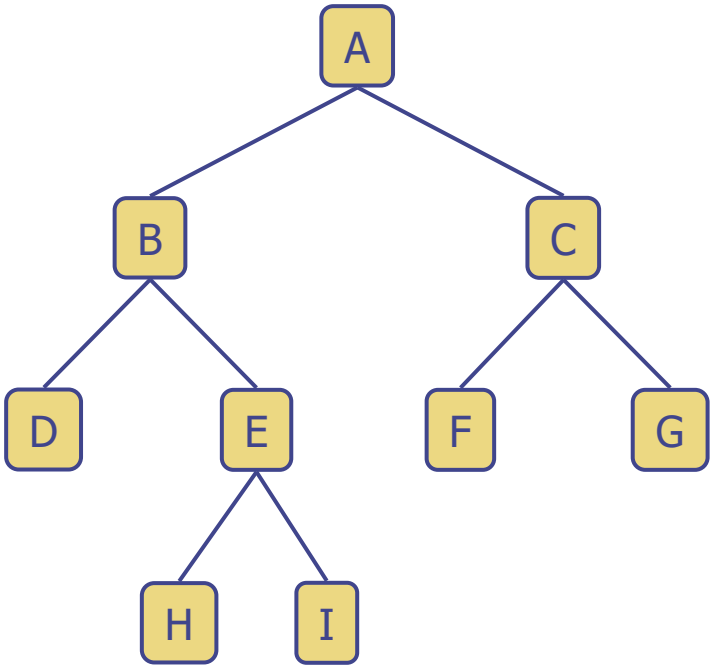
- Applications:
  - searching



Proper Binary Trees

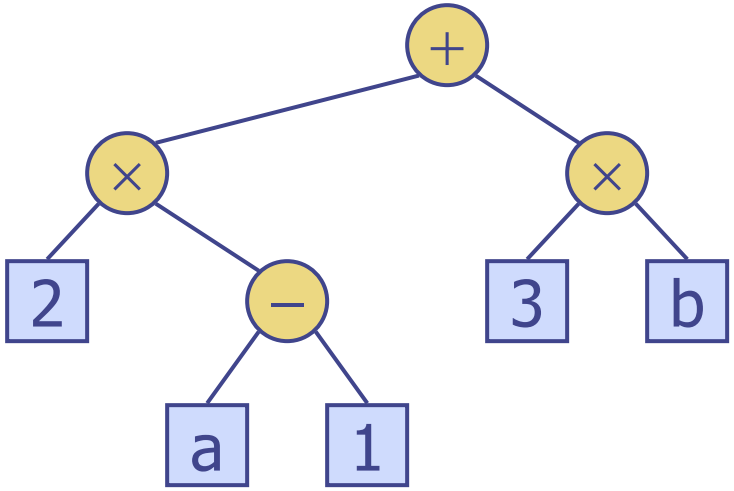
- A proper binary tree is a tree with the following properties:
  - Each internal node has either two children or no children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a root of a binary tree

- Applications:
  - arithmetic expressions
  - decision processes



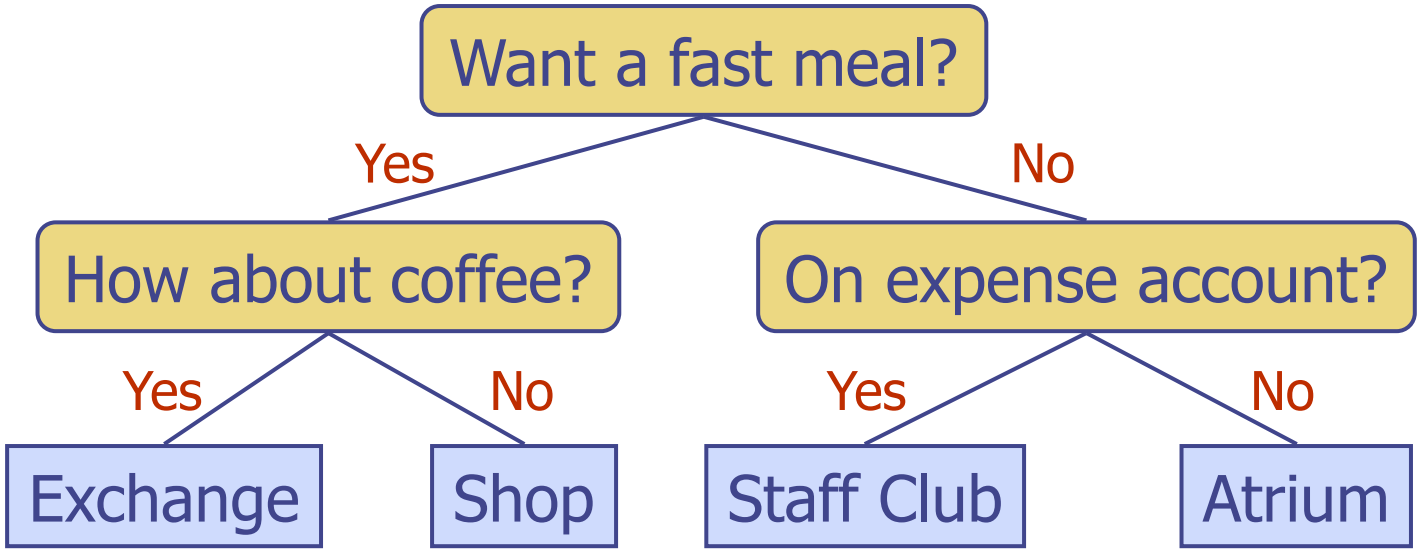
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: (binary) operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

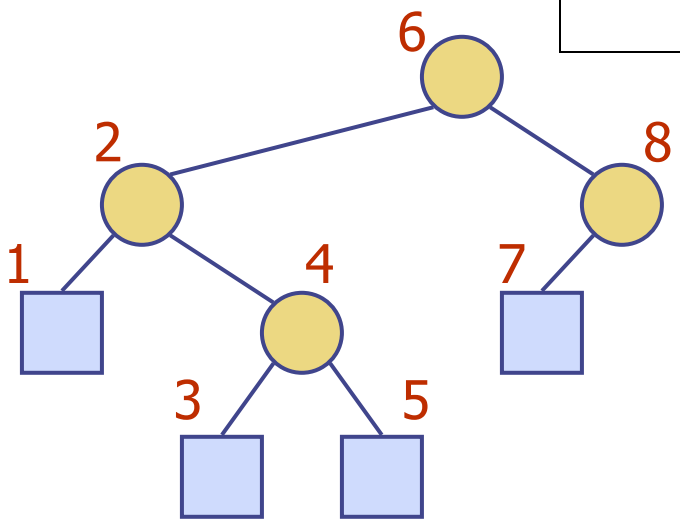
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
    - hence a proper tree
  - external nodes: decisions
- Example: dining decision



Inorder Traversal

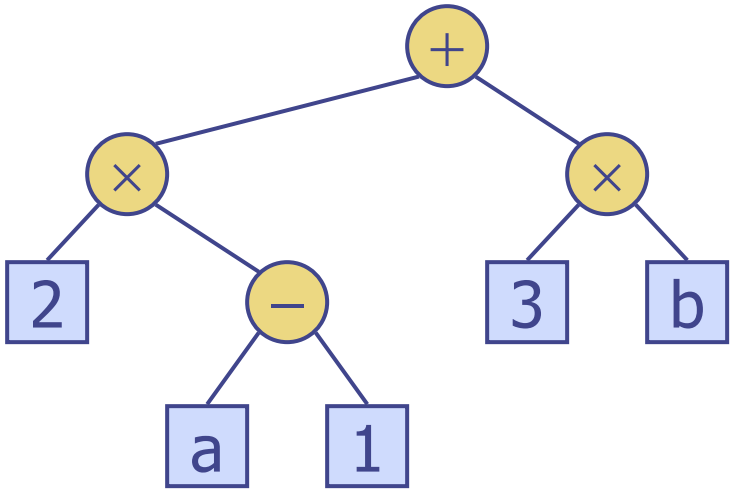
- **In an inorder traversal a node is visited after its left subtree and before its right subtree**
- Application: draw a binary tree by (x,y) coords:
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(*v*)
  if *hasLeft* (*v*)
    inOrder (*left* (*v*))
  visit(*v*)
  if *hasRight* (*v*)
    inOrder (*right* (*v*))



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



Algorithm *printExpression*(*v*)

  if *hasLeft* (*v*)

    print("(")

    printExpression (*left*(*v*))

  print(*v.element* ())

  if *hasRight* (*v*)

    printExpression(*right*(*v*))

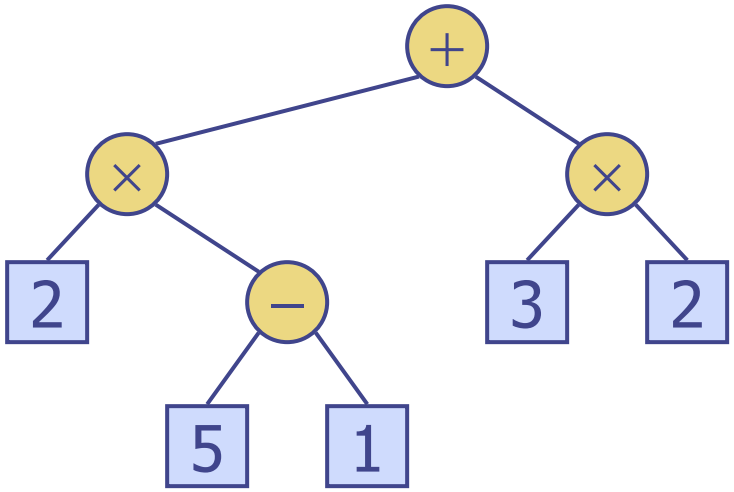
    print (")")

((2 × (a – 1)) + (3 × b))

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal:
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

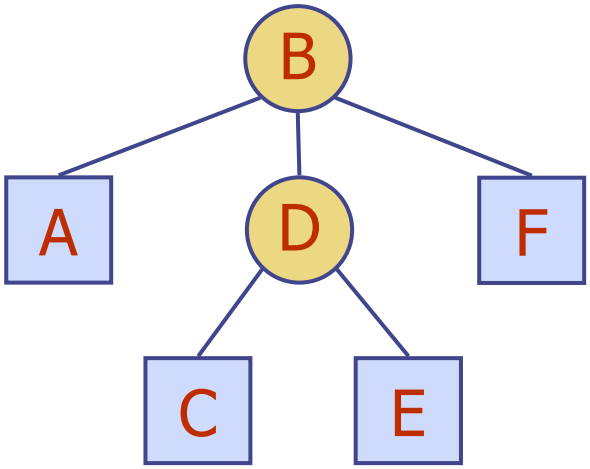
```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(leftChild (v))
        y ← evalExpr(rightChild (v))
        ◊ ← operator stored at v
        return x ◊ y
```

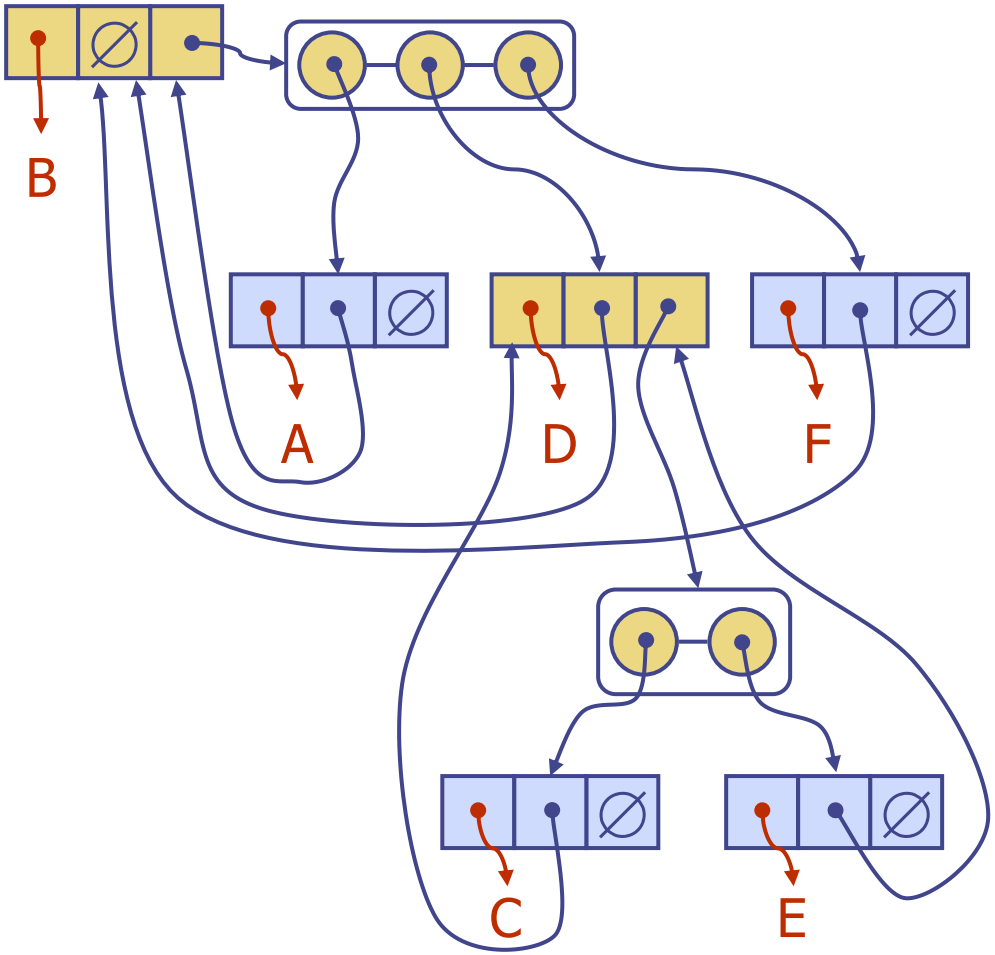


- Exercise: what is the value?
- Exercise: Which traversal ?
- Post- In- or Pre- ?

Linked Structure for Trees

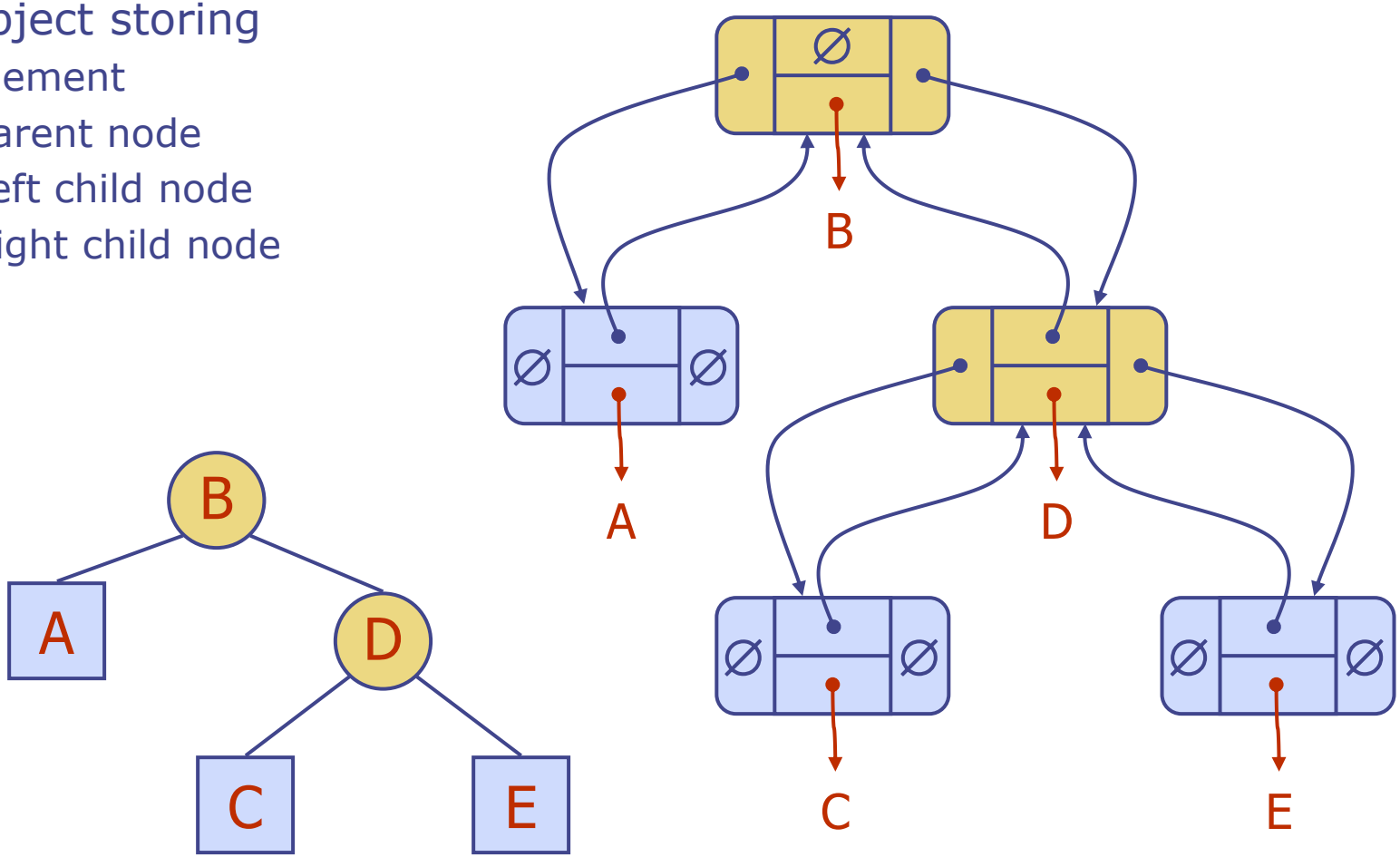
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes





Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node



Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- **An ADT specifies:**
  - **Data stored**
  - **Operations on the data**
  - **Error conditions associated with operations**
- **An ADT does <u>not</u> specify the implementation itself - hence "abstract"**

Abstract Data Types (ADTs)

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Concrete Data Types (CDTs)

- The actual date structure that we use
  - Possibly consists of Arrays or similar
- An ADT might be implemented using different choices for the CDT
  - The choice of CDT will not be apparent from the interface: “data hiding” “encapsulation” – e.g. see ‘Object Oriented Methods’
  - The choice of CDT will affect the runtime and space usage – and so is a major topic of this module

ADT & Efficiency

- Often the ADT comes with efficiency requirements expressed in big-Oh notation, e.g.
  - "cancel(order) must be O(1)"
  - "sell(order) must be O(log(|orders|))"
- However, such requirements do not automatically force a particular CDT.
  - The underlying implementation is still not specified
- This is typical of many "library functions"
- Note that such efficiency specifications rely on using the big-Oh family.

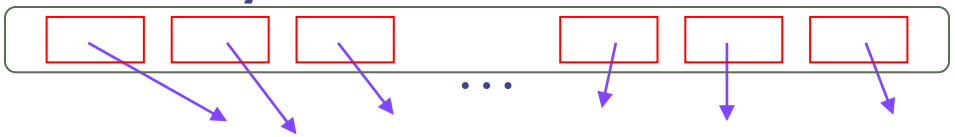
Tree ADT

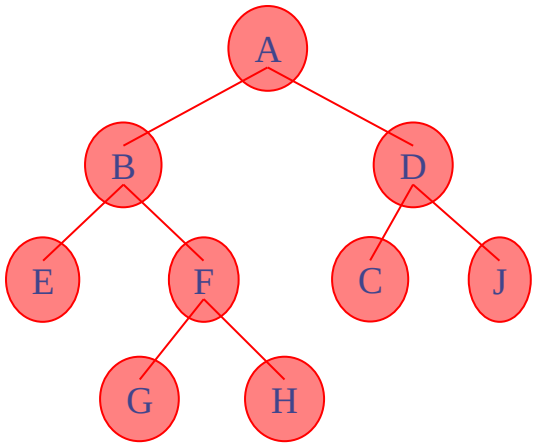
- We can use "positions", p, to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

BUT the CDT can be quite different !!

Array-Based Representation of Binary Trees

- nodes are stored in an array



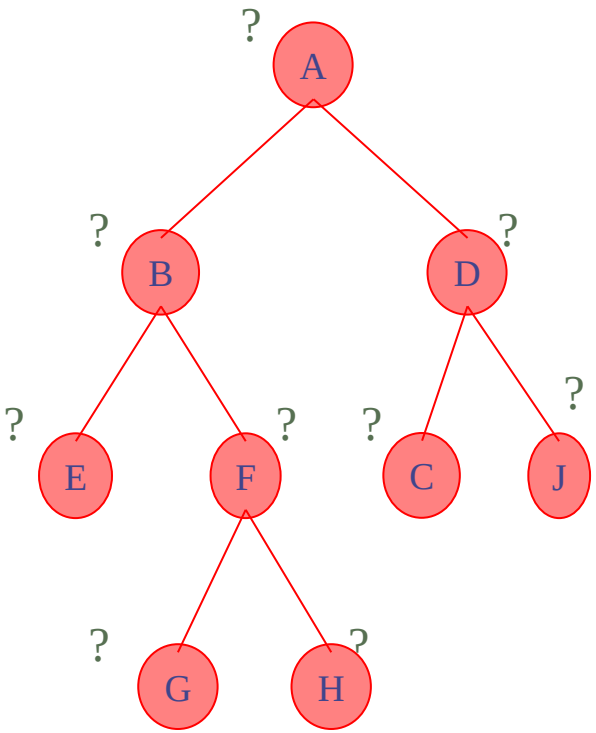


- let rank(node), or index in the array, be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Array-Based Representation of Binary Trees

- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

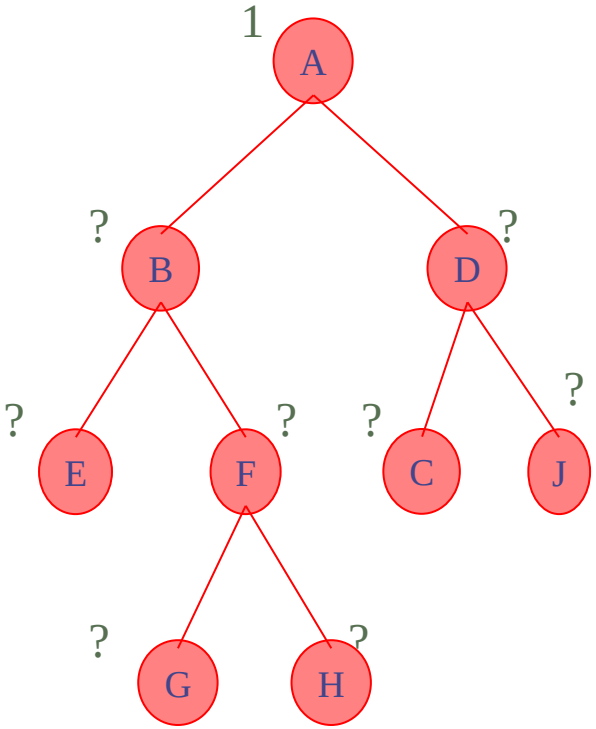


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Array-Based Representation of Binary Trees

- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

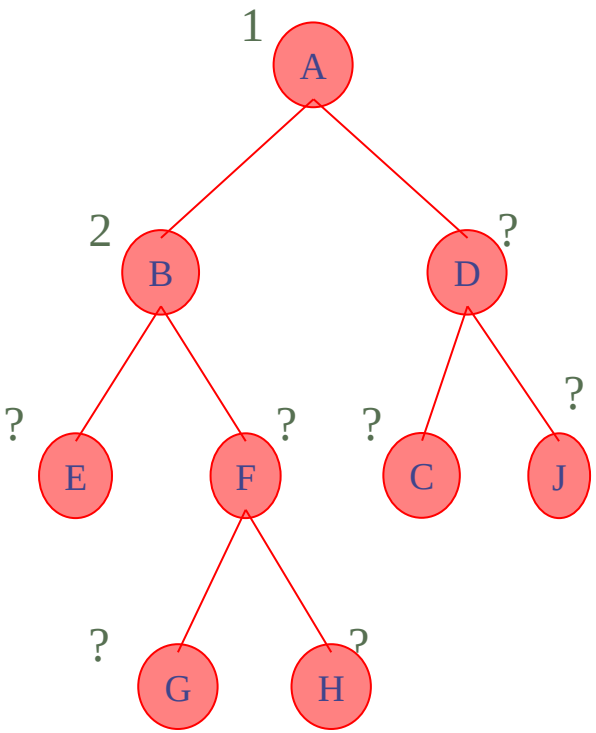


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A															

Array-Based Representation of Binary Trees

- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

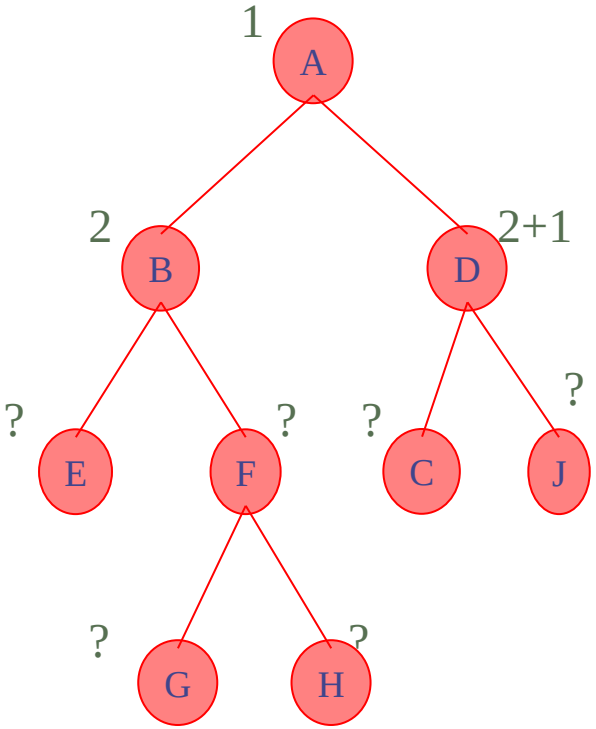


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B														

Array-Based Representation of Binary Trees

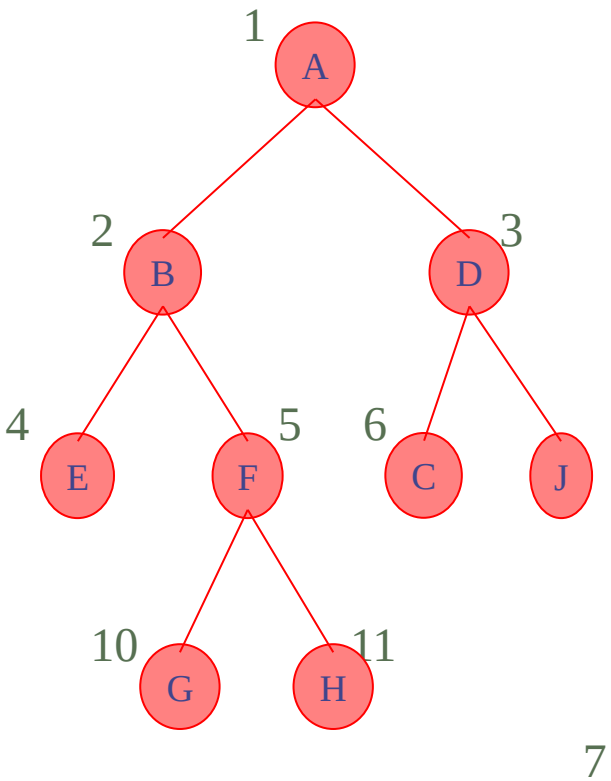
- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B	D													

Array-Based Representation of Binary Trees



- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B	D	E	F	C	J			G	H					

Implemention

- Remember that if think of the rank, r(n) of node n, as a binary number then
  - "times 2" is a left shift "<<1"
- r(n) = r(par(n))<<1 + 0 for left
- r(n) = r(par(n))<<1 + 1 for right
- E.g. r(par(n)) = 101 gives children at
  - "101"+"0" = 1010
  - "101"+"1" = 1011
- Going to the parent is a right shift
- Hence, implementations of this can be very fast – used in binary heaps later.

Exercise (offline)

- Why is this representation correct?
  - I.e. does the mapping between array satisfy needed uniqueness properties?
  - Is it true that each element of the array corresponds to a unique node of the tree?
  - E.g. If I claim that it is incorrect, then how would you convince me otherwise?
- Hint: from the previous slide think of the rank written as binary number
  - Realise that it describes the “L” vs. “R” decisions on going from the root.
  - Relate to each number having a unique binary representation

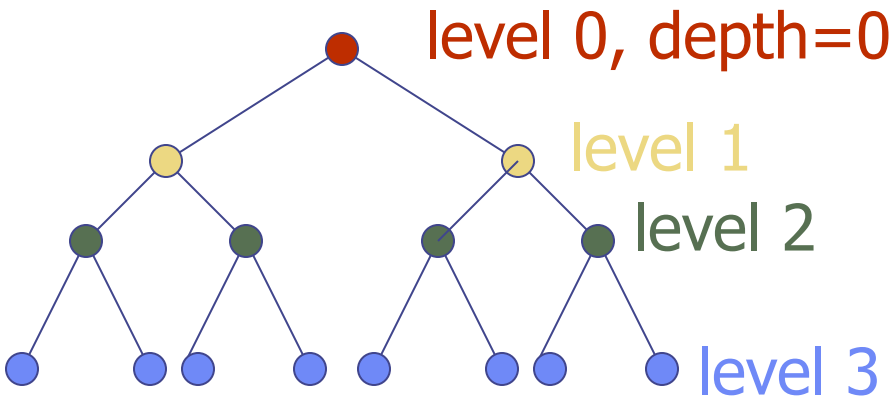
Misc. Comments

- Advantages of the tree-as-array structure:
  - Saves space as do not have to store the pointers – they are replaced by fast computations
  - The storage can be more compact – "better memory locality" and this can be good because of cache and memory hierarchies – when an array element is accessed then other entries can be pulled into the cache, and so access becomes faster.

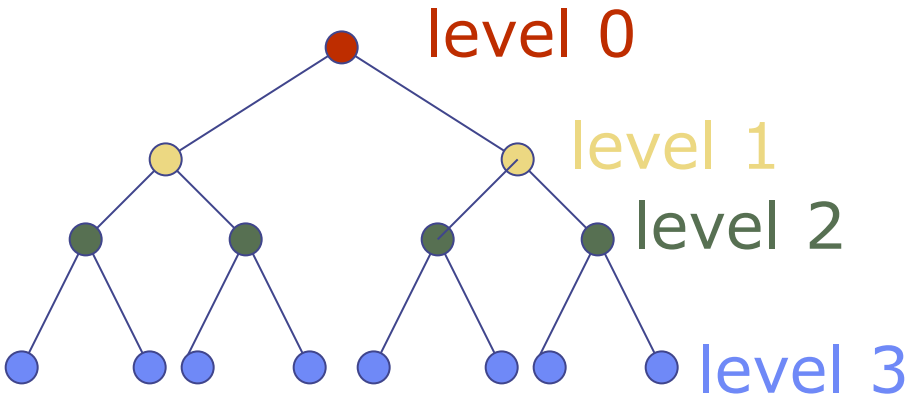
Properties of perfect binary trees

- A binary tree is said to be "**proper**" (a.k.a. "**full**") if every internal node has exactly 2 children
- It is "**perfect**" if it is proper and all leaves are at the same depth; hence all levels (depths) are full.

Perfect binary tree of height 3:



Properties of perfect binary trees



d	at d	at d or less
0	1	1
1	2	3
2	4	7
3	8	15

Counting suggests numbers of nodes are:

- 2^d at level d
- $2^{d+1}-1$ at level d or less

Can formally prove using induction

Numbers of nodes are exponential in the depth

Height (h) is logarithmic in size (n)

- This is a very important property of perfect binary trees
  - Exercise: is this true for all trees?
- Tree algorithms often work by going down the tree level by level – following a path from root to a leaf
  - Their running time depends on the number of levels
  - hence are O(height)
- But we usually only know the number of nodes, n, and so need to convert height, h, to a function of n
- Let us prove that for perfect binary trees

$$\boldsymbol{h} = \log_2 (\boldsymbol{n} + 1) - 1$$

where n in the number of nodes. Or (same thing):

number of levels = $\log_2 (\boldsymbol{n} + 1)$.

How many nodes at level k

- First, it is useful to find out how many nodes are at a certain level in perfect binary tree
- Let us count levels from 0. This way level k contains nodes which have depth k.

How many nodes at level k

- Claim: level (depth) k contains 2^k nodes.
- Proof: by induction on k.
  - (basis of induction) if k = 0, the claim is true:

  $2^0 = 1$, and we only have one node (root) at level 0.
  - (inductive step): suppose the claim is true for k-1: level k-1 contains 2^{k-1} nodes.
  - We need to prove that then the claim holds for k: level k holds 2^k nodes.
  - Since each node at level k-1 has 2 children, there are twice as many nodes at level k.
  - So, level k contains $2 * 2^{k-1} = 2^k$ nodes. QED

How many nodes in a tree of height h?

Theorem: A perfect binary tree of height h contains $2^{h+1} - 1$ nodes.

Proof: by induction on h

- (basis of induction): h=0. The tree contains $2^1 - 1 = 1$ node.
- (inductive step): assume a tree of height h-1 contains $2^h - 1$ nodes. A tree of depth h has one more level (h) which contains 2^h nodes. The total number of nodes in the tree of height h is: $2^h - 1 + 2^h = 2 * 2^h - 1 = 2^{h+1} - 1$. QED

What is the height of a (perfect) binary tree of size n (with n nodes)?

We know that $n = 2^{h+1} - 1$.

So, $2^{h+1} = n + 1$.

$h + 1 = \log_2(n+1)$

$h = \log_2(n+1) - 1$.

So, the height of the tree is logarithmic in the size of the tree.

The size of the tree is exponential in the height (number of levels) of the tree.

What is the height of an arbitrary binary tree of size n (with n nodes)?

- If the tree is perfect, then it has height that is logarithmic in the size of the tree: $\Theta(log(n))$
- If it is imperfect, then for the same n it must have at least this height: $\Omega(\log(n))$
- However, consider a simple "chain" – basically a linked list
  - each non-leaf node has just one child. It is still a binary tree – just a special case.
  - It has height n-1 and this is "obviously" maximal height
  - Hence, trees have height O(n).
- **Hence, for a general binary tree on n nodes, the height is $\Omega(log(n))$ and $O(n)$**

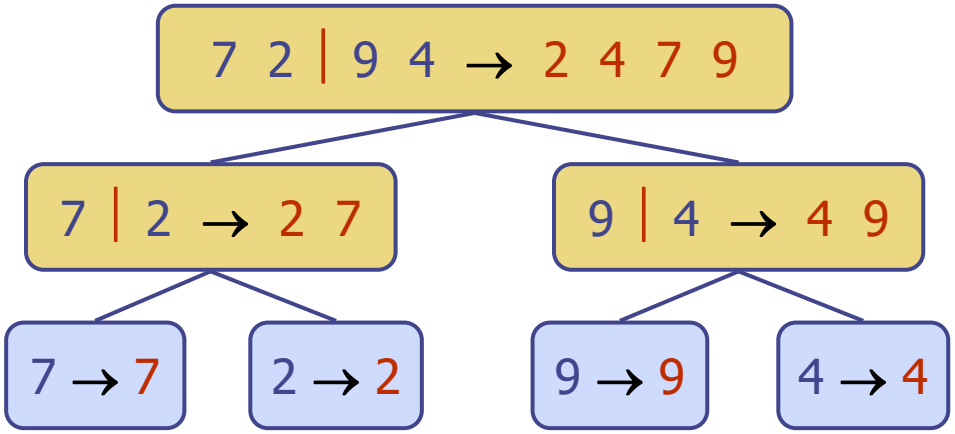
Minimum Expectations

- Definitions associated with trees
- **Post- Pre- and In-order traversal and their usages**
- Implementation methods
  - nodes
  - array based
- Binary Trees – meaning of proper, perfect
- **Sizes and heights of binary trees**

COMP2054-ADE

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Mergesort



Divide-and-Conquer

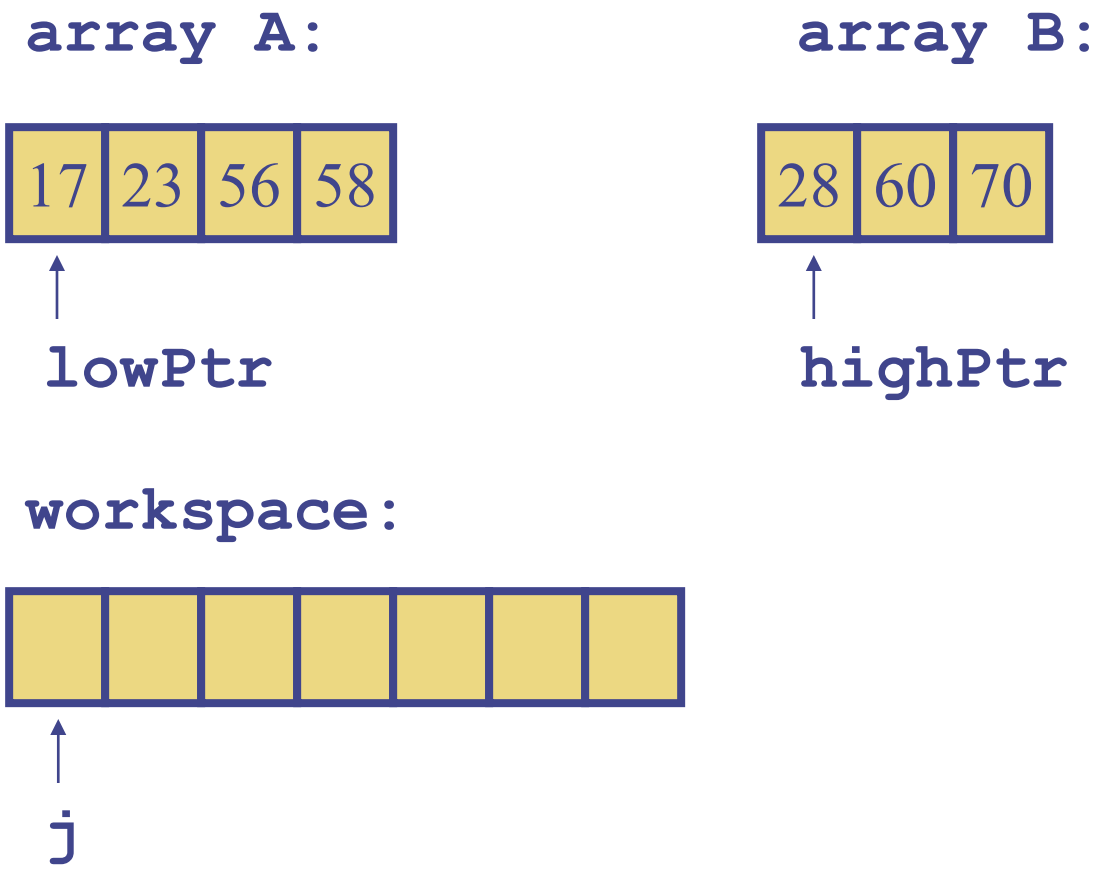
- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1 (or "small enough to be done directly")

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

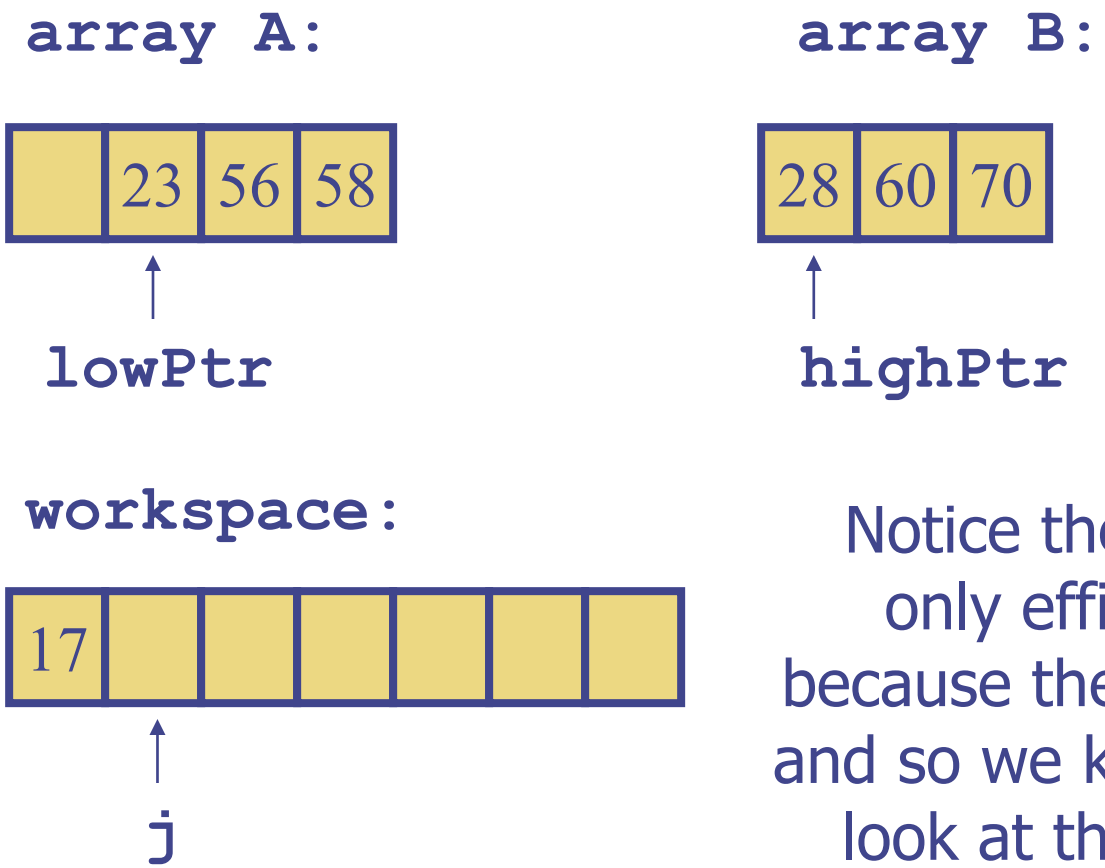
Merge-Sort

- Merge-sort on an input sequence (array/list) S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence
- First questions:
  Is the merge easy?
  What is the big-Oh of the merge?

Merging sorted arrays

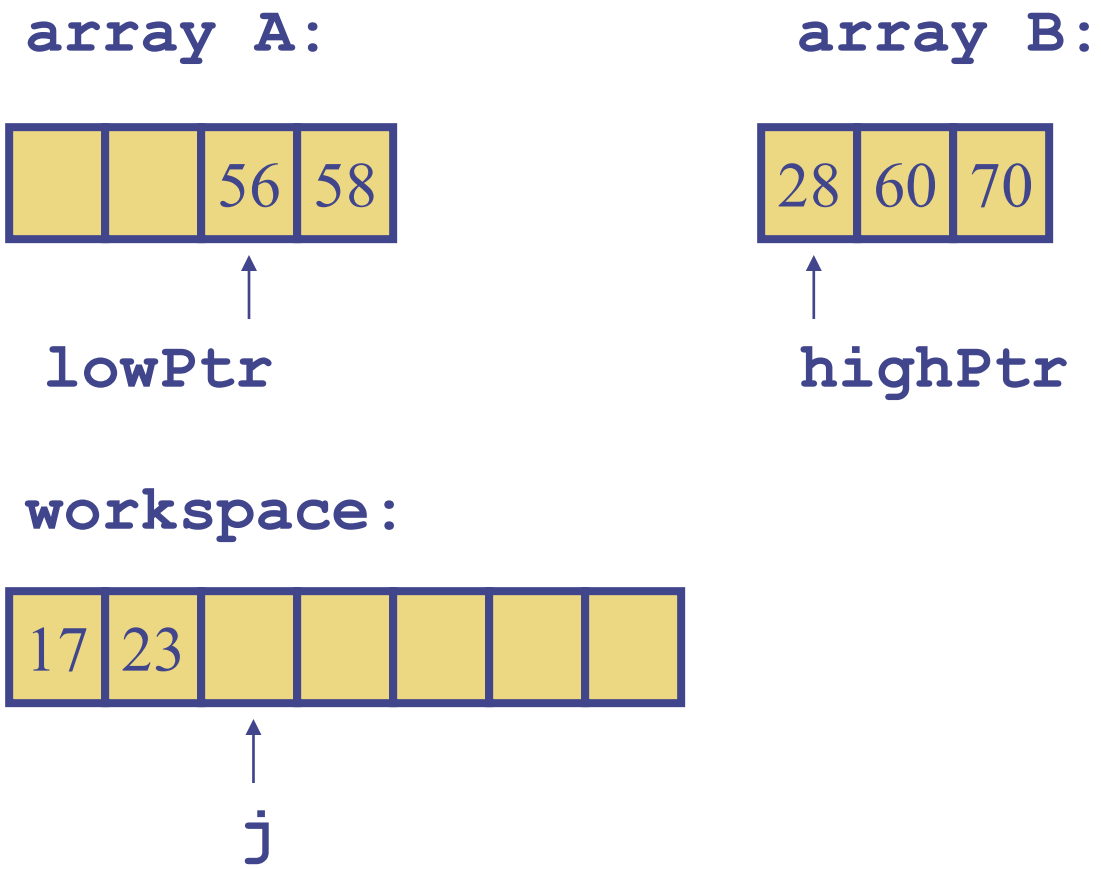


Merging sorted arrays

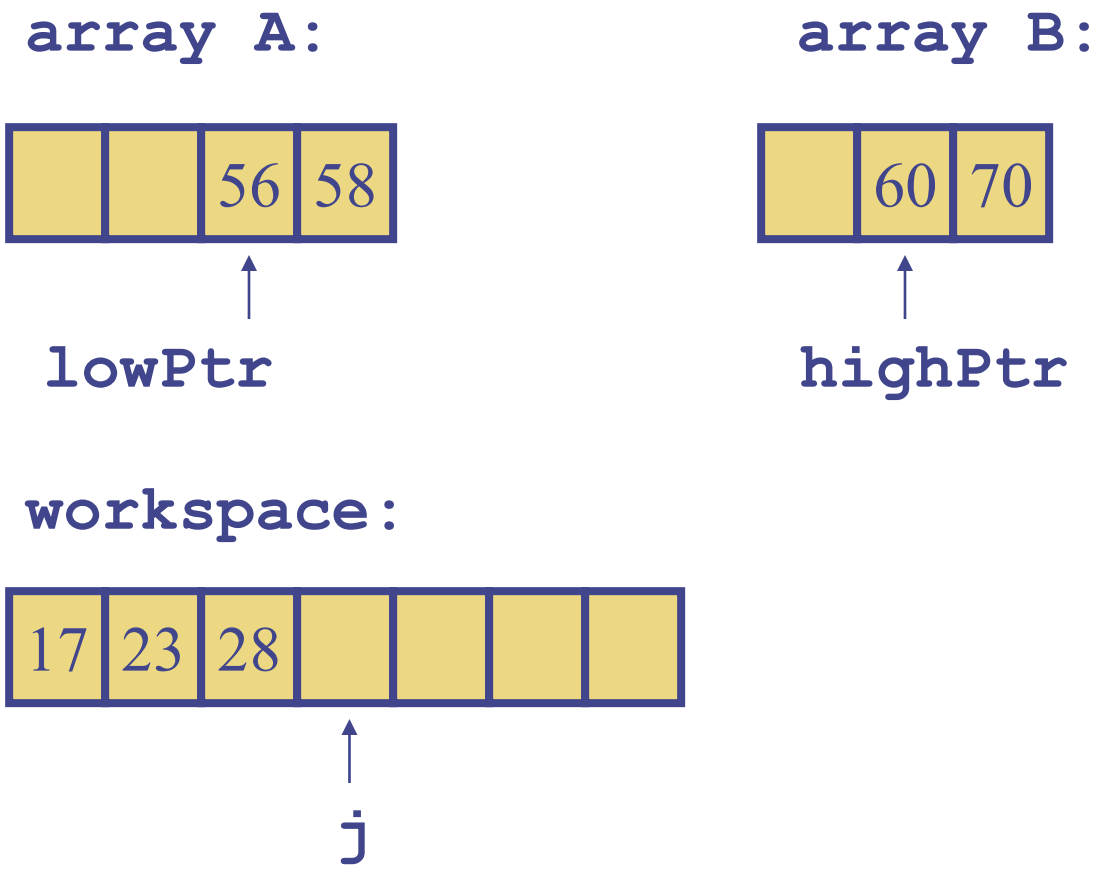


Notice the choice of 17 is only efficient and easy because the inputs are sorted and so we know we only need look at the first element!!

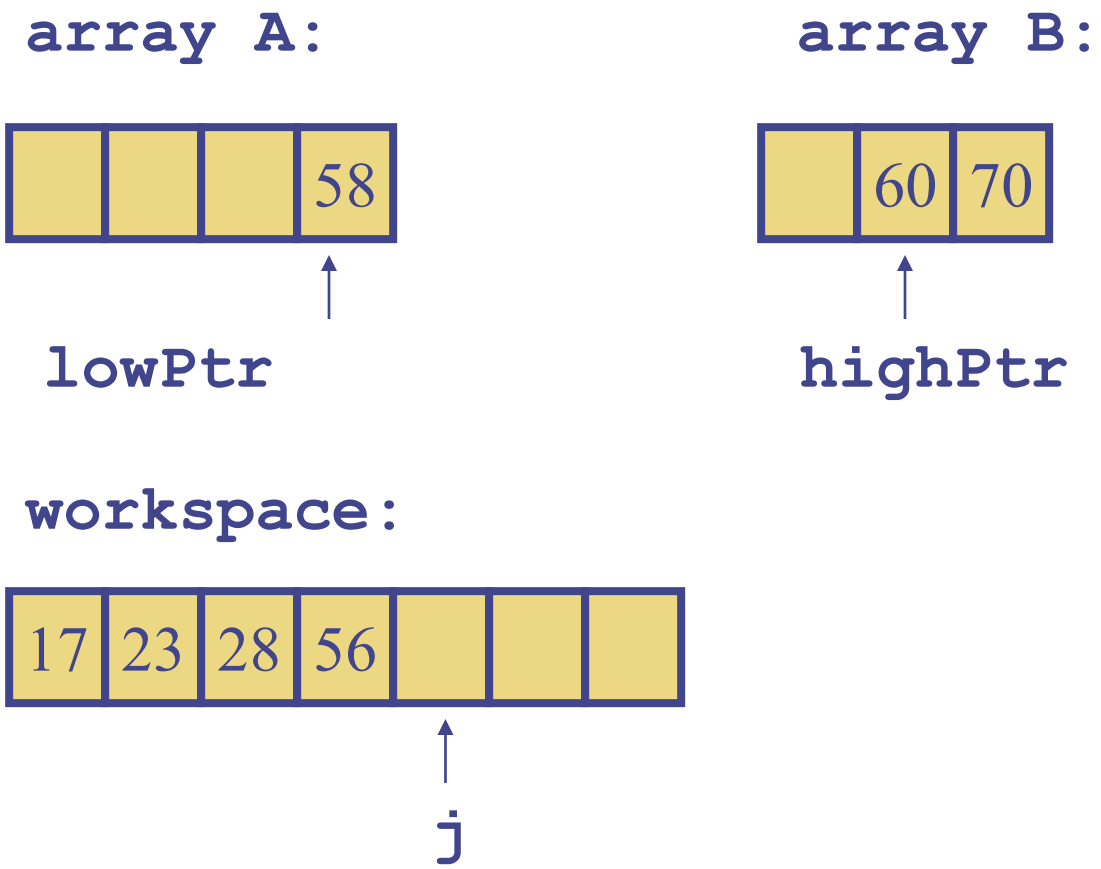
Merging sorted arrays



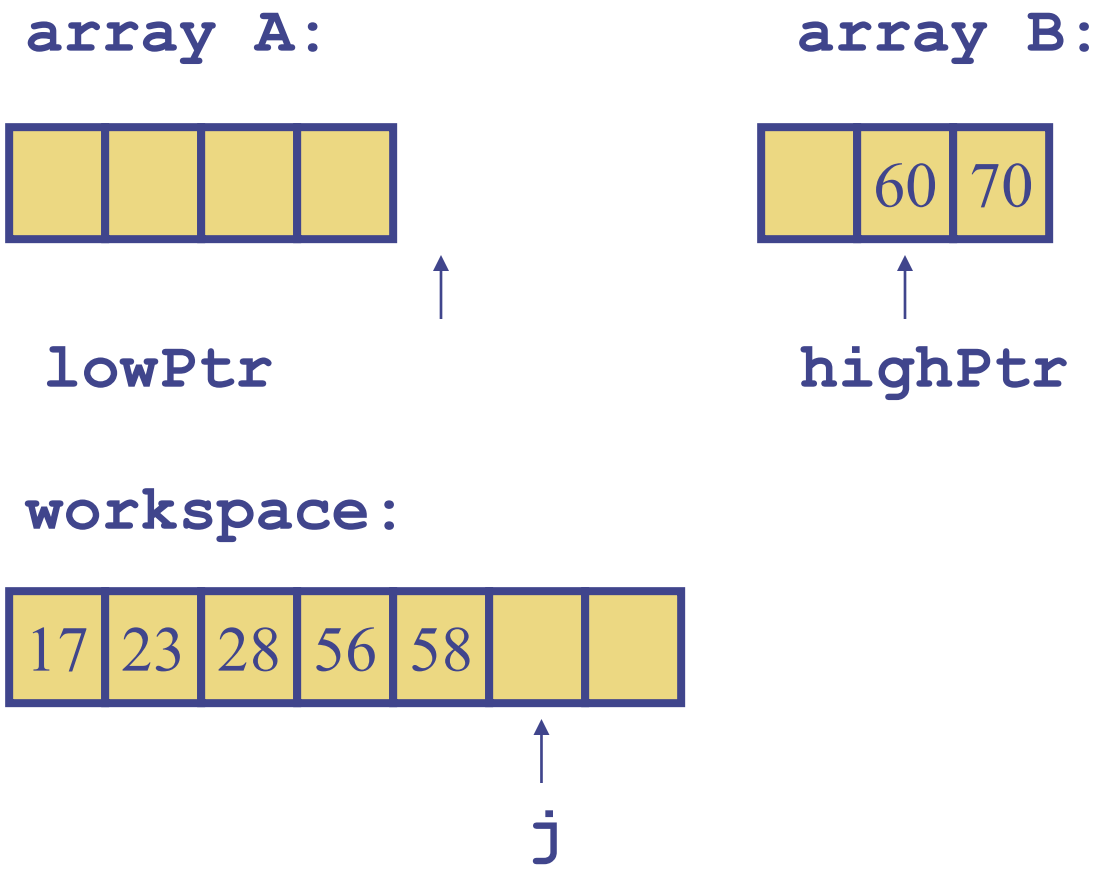
Merging sorted arrays



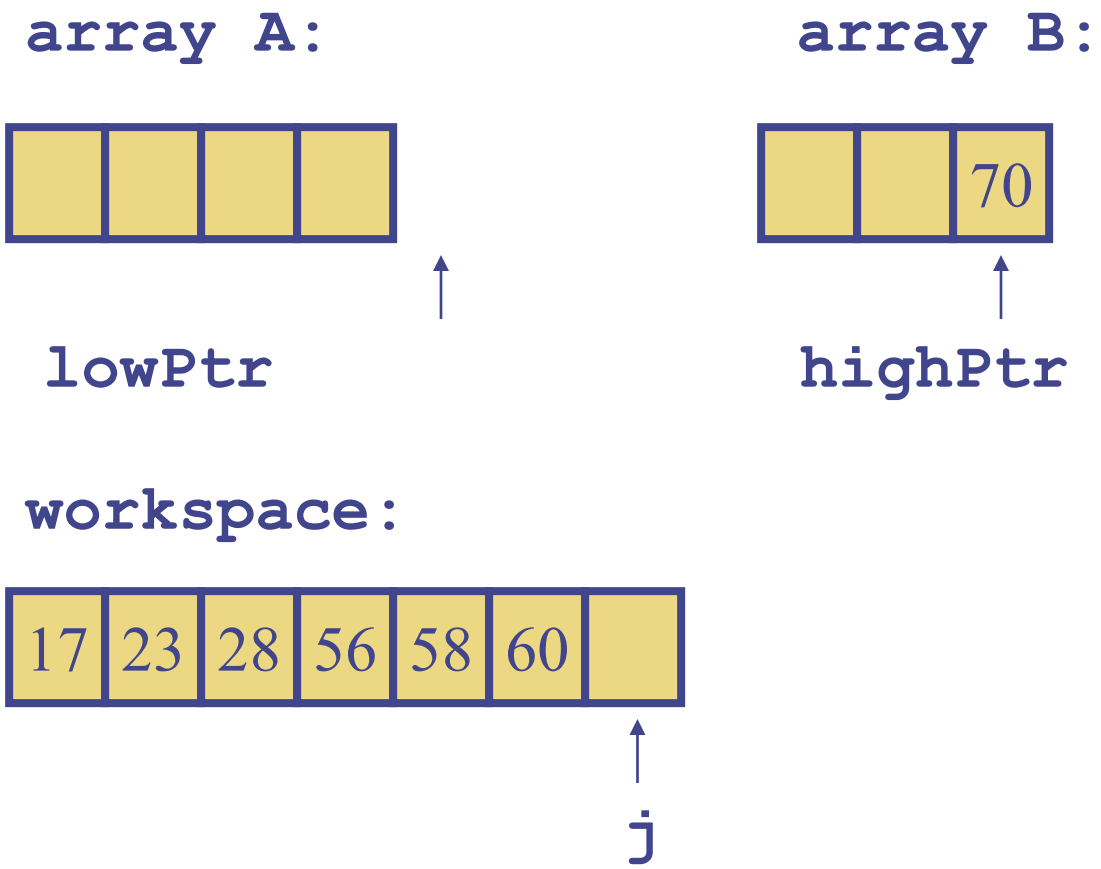
Merging sorted arrays



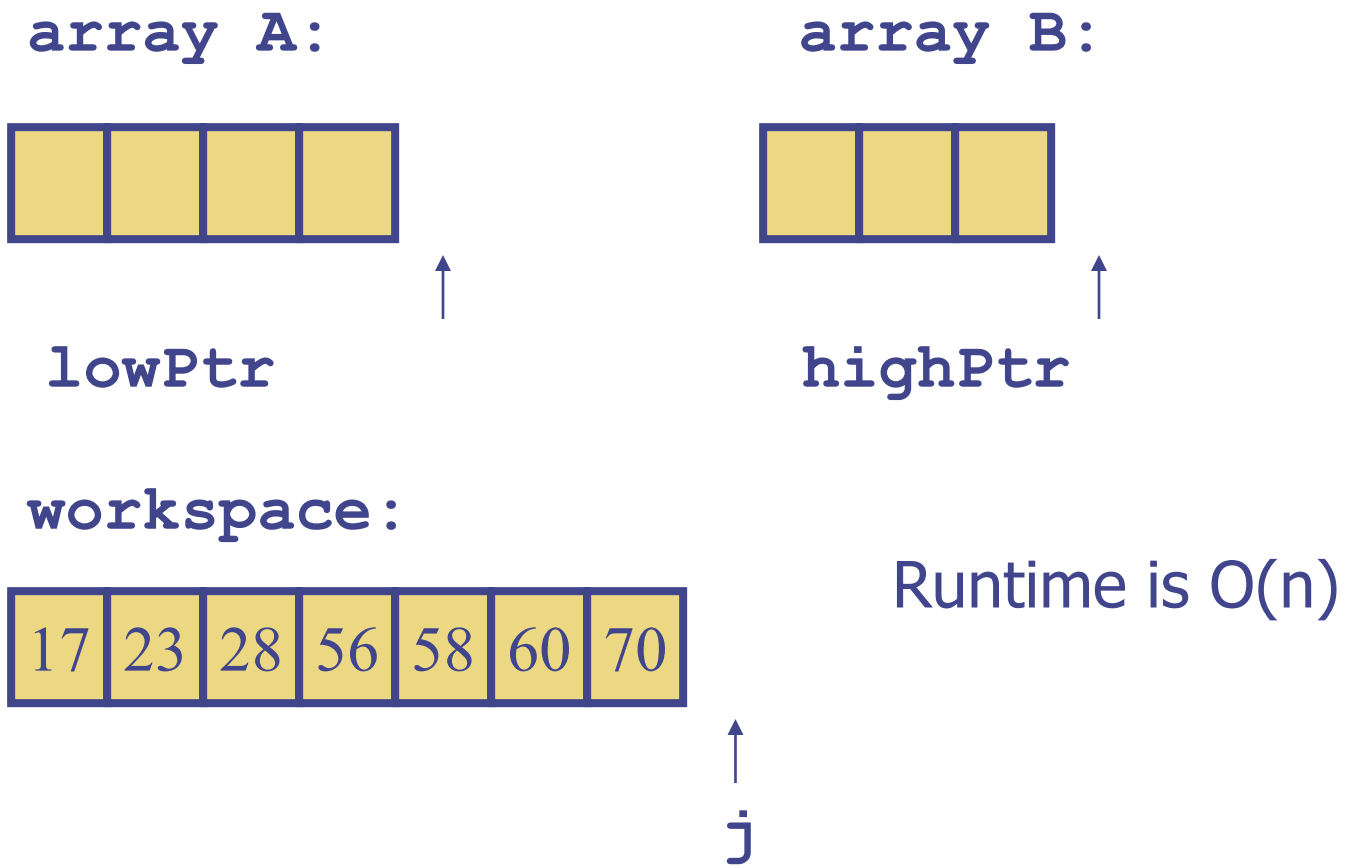
Merging sorted arrays



Merging sorted arrays



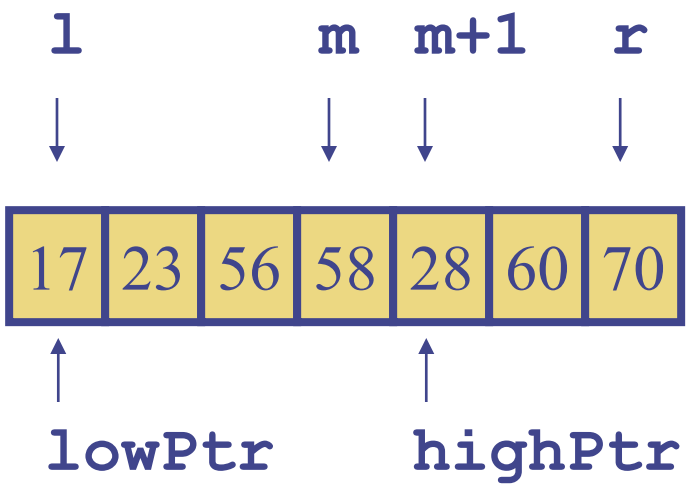
Merging sorted arrays



Runtime is O(n)

Merging halves of an array

- Pass boundaries of sub-arrays to the algorithm instead of new arrays, and merge into a "workspace":
- After merge, copy the workspace back to the original array



Implementation

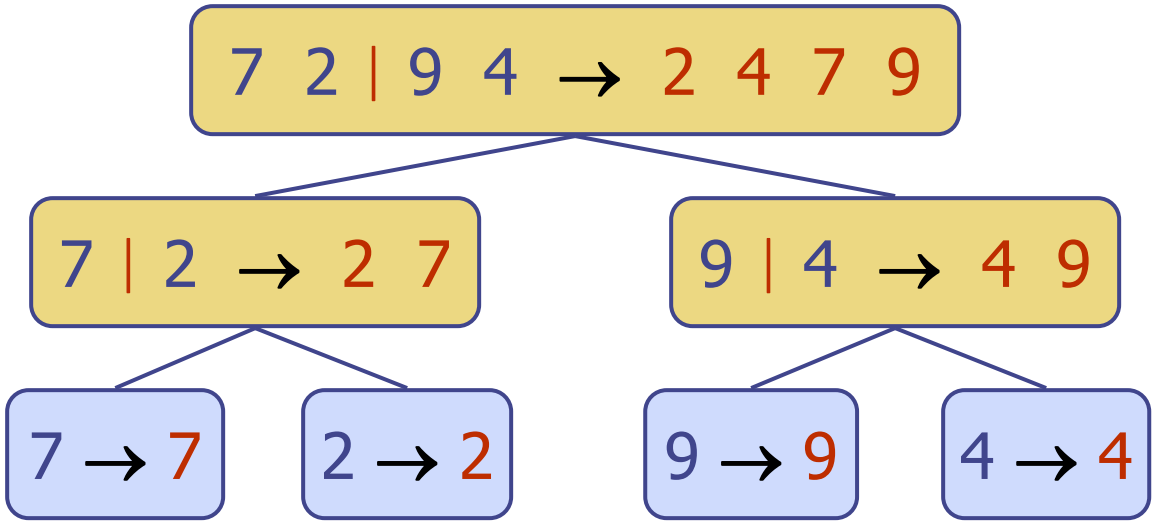
```
public static void recMergeSort
                    (int[] arr, int[] workSpace, int l, int r) {
    if (l == r) {
        return;
    } else {
        int m = (l+r) / 2;
        recMergeSort(arr, workSpace, l, m);
        recMergeSort(arr, workSpace, m+1, r);
        merge(arr, workSpace, l, m+1, r);
    }
}
```

Initial call is with l=0 and r to be end of the array

Merge-Sort Tree

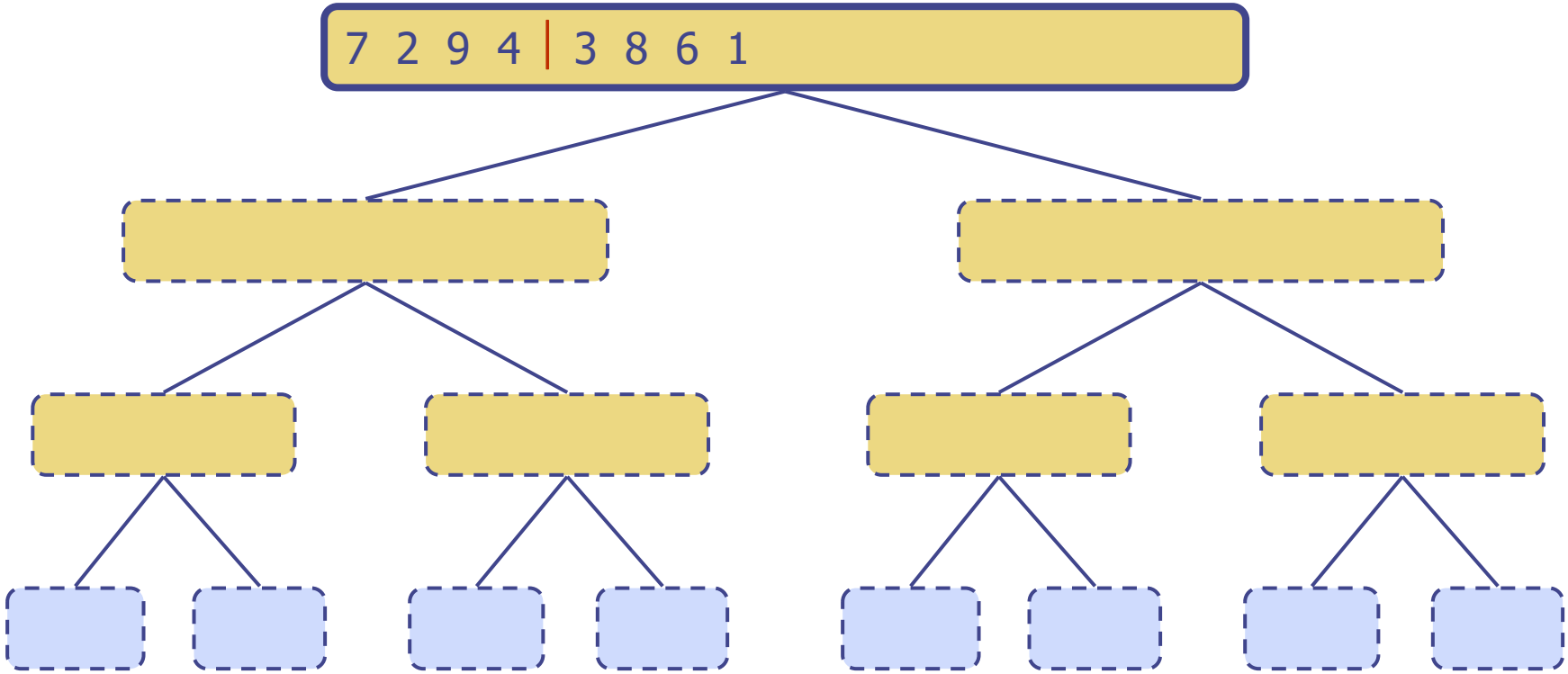
Not actually implemented this way! A `node' here is conceptual not real

- An execution of merge-sort is **depicted** by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



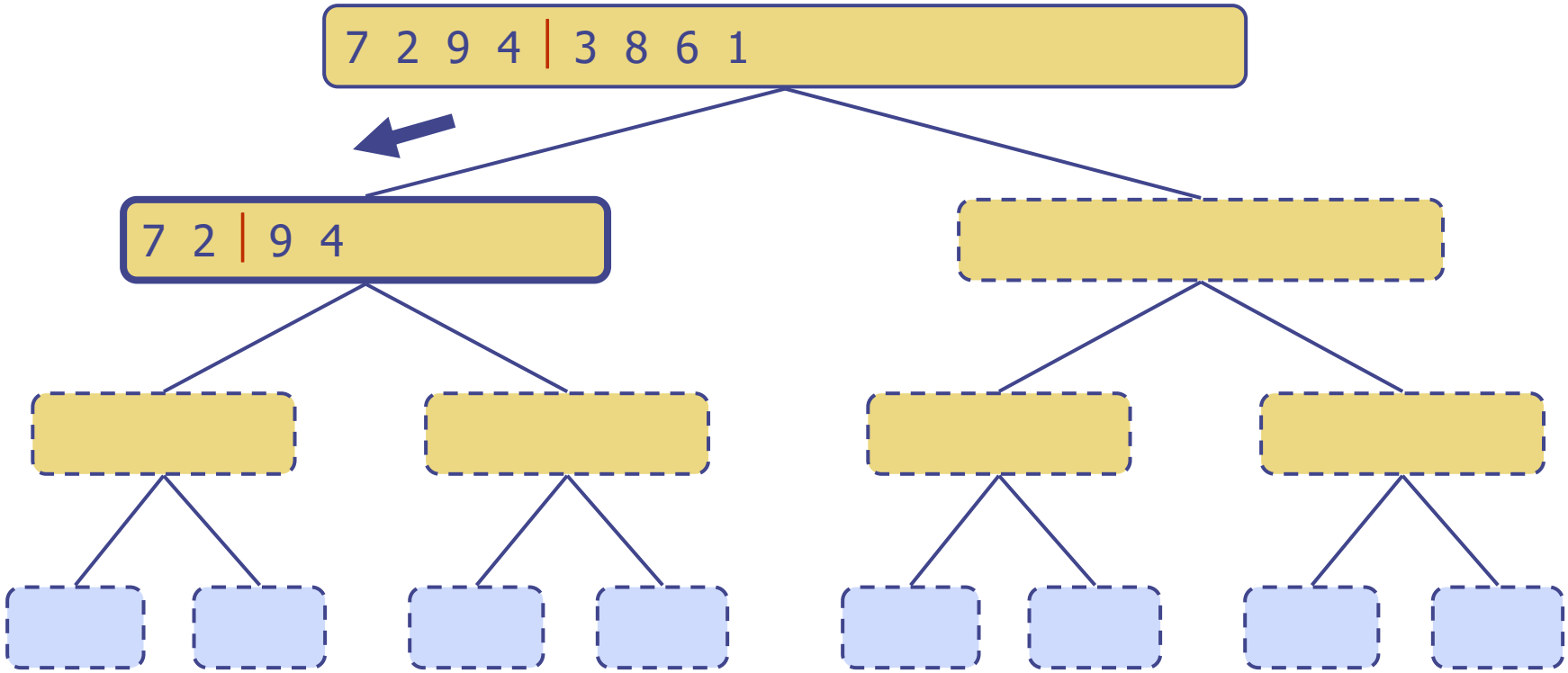
Execution Example

- Partition



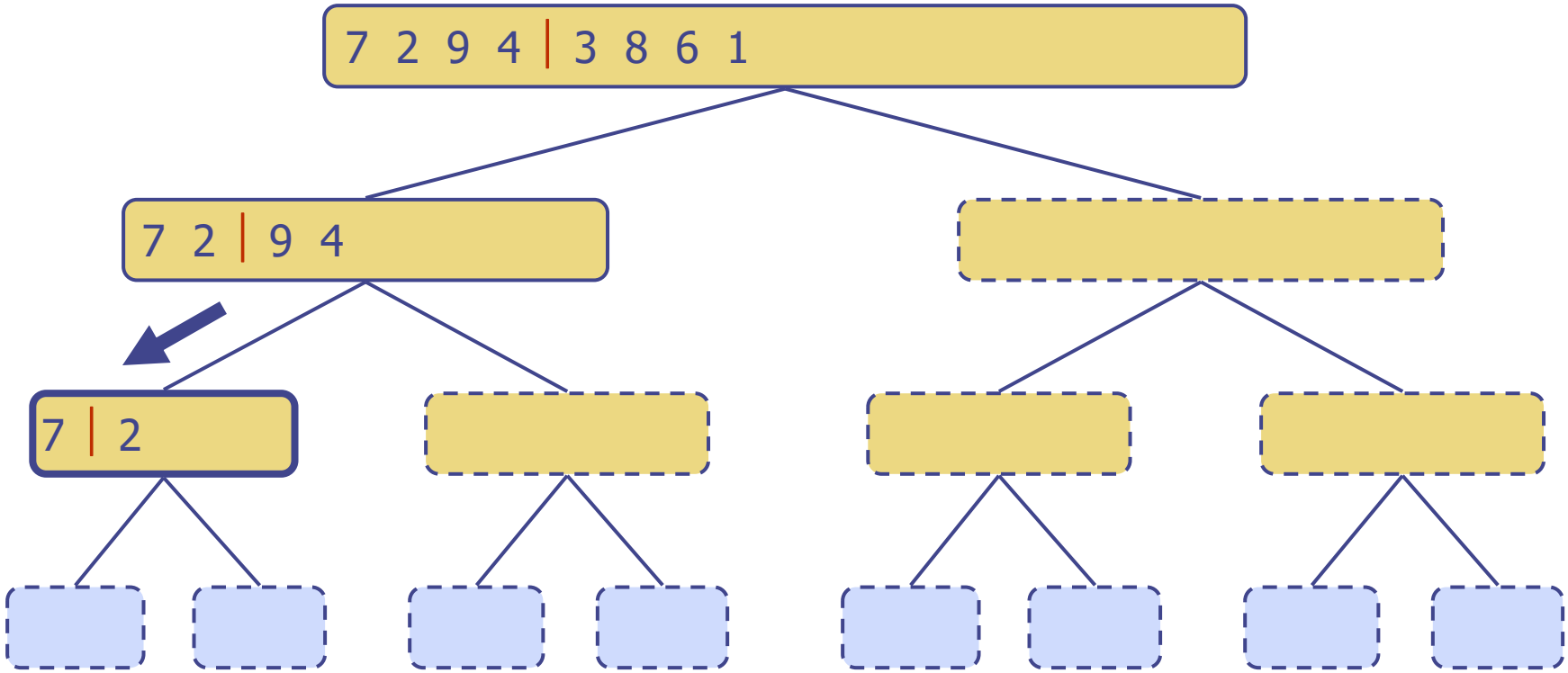
Execution Example (cont.)

- Recursive call, partition



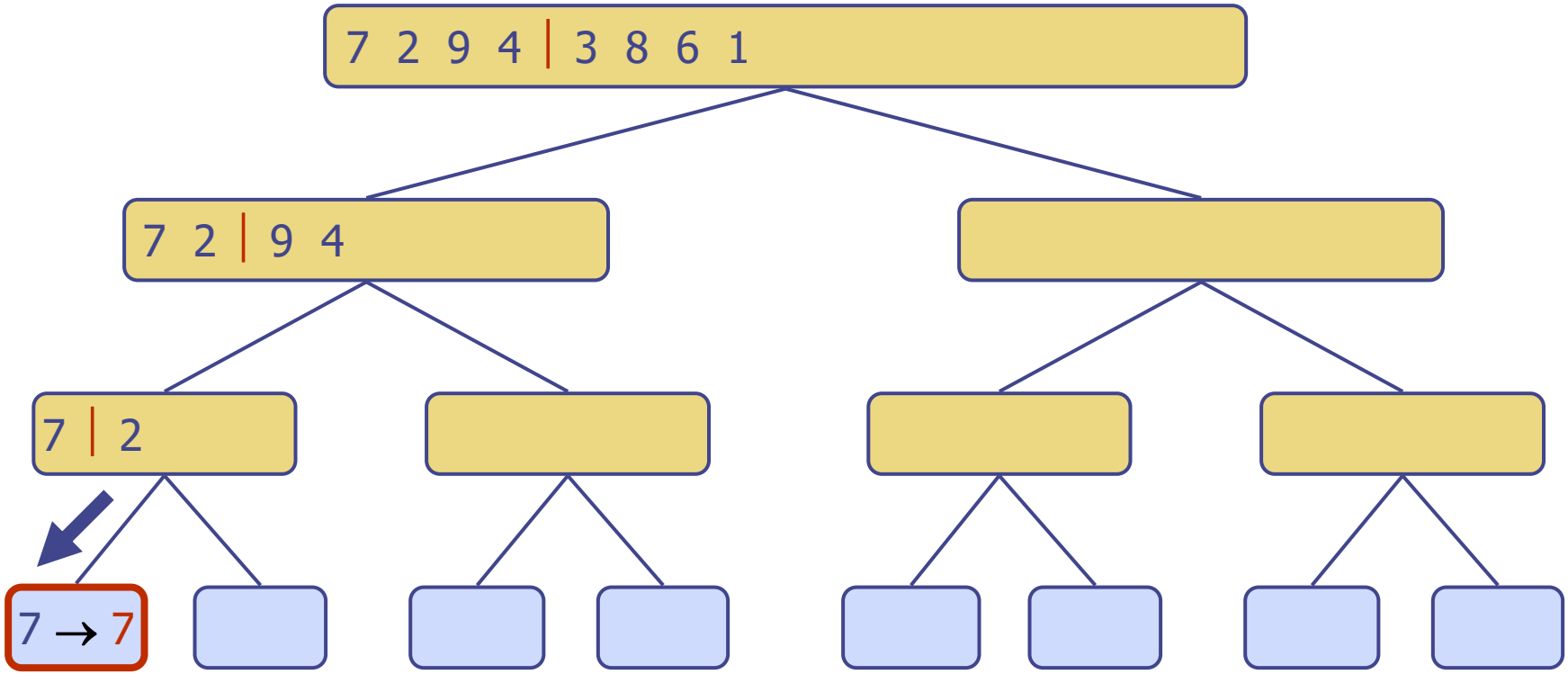
Execution Example (cont.)

- Recursive call, partition



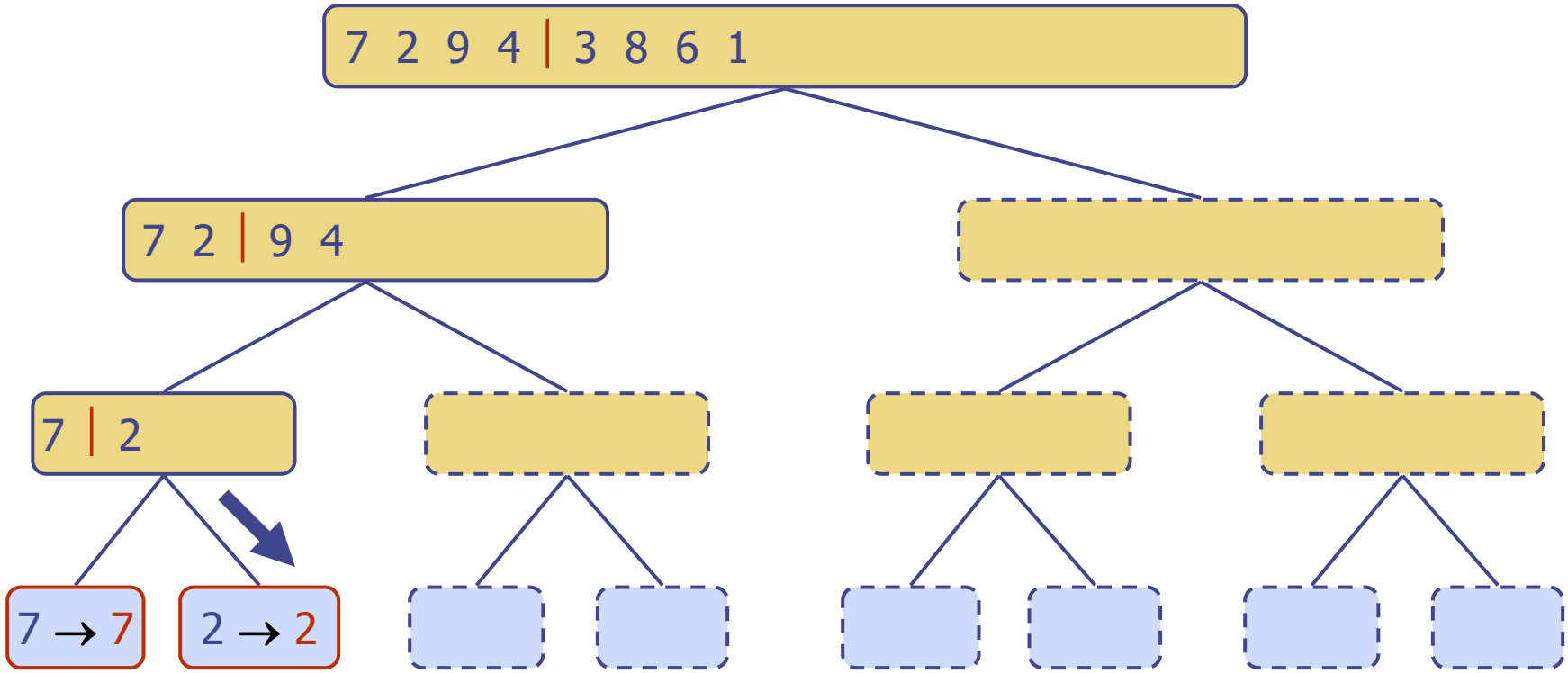
Execution Example (cont.)

- Recursive call, base case



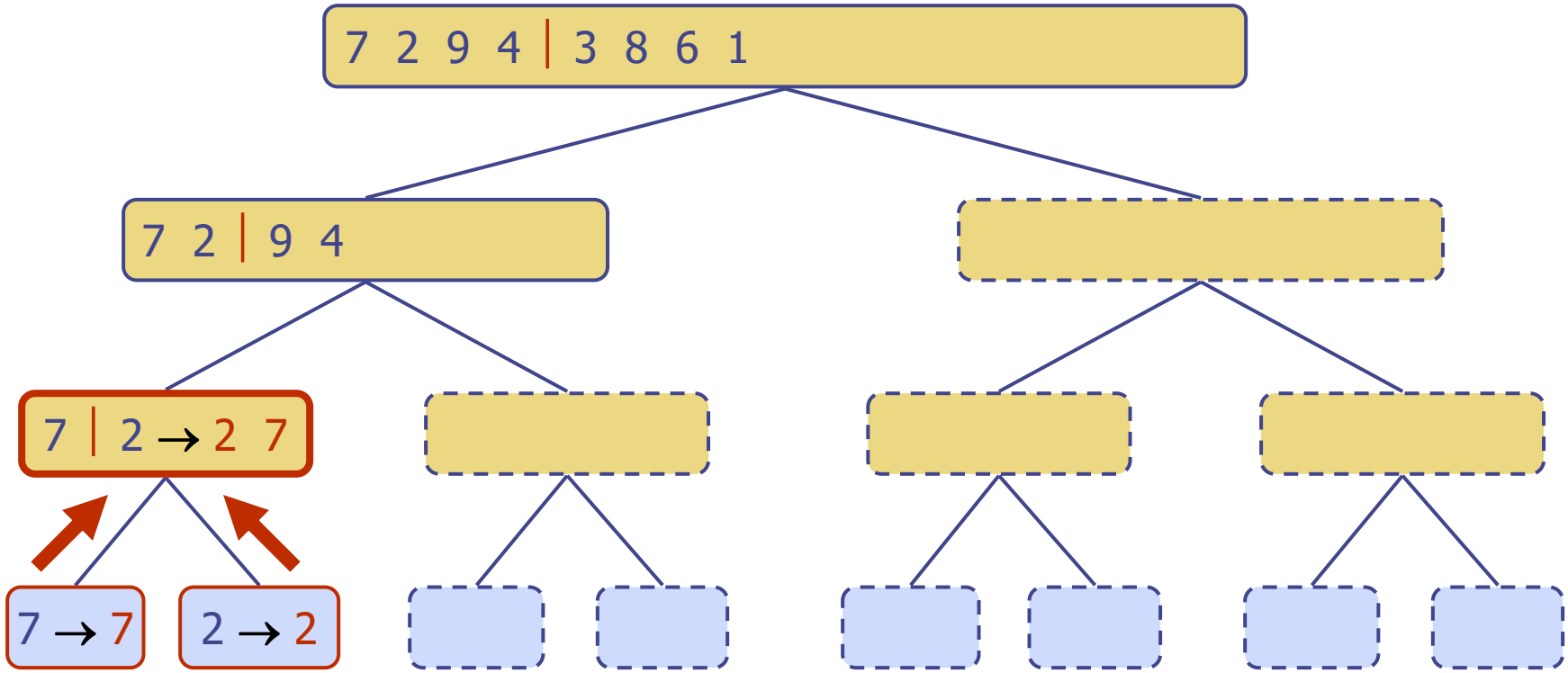
Execution Example (cont.)

- Recursive call, base case



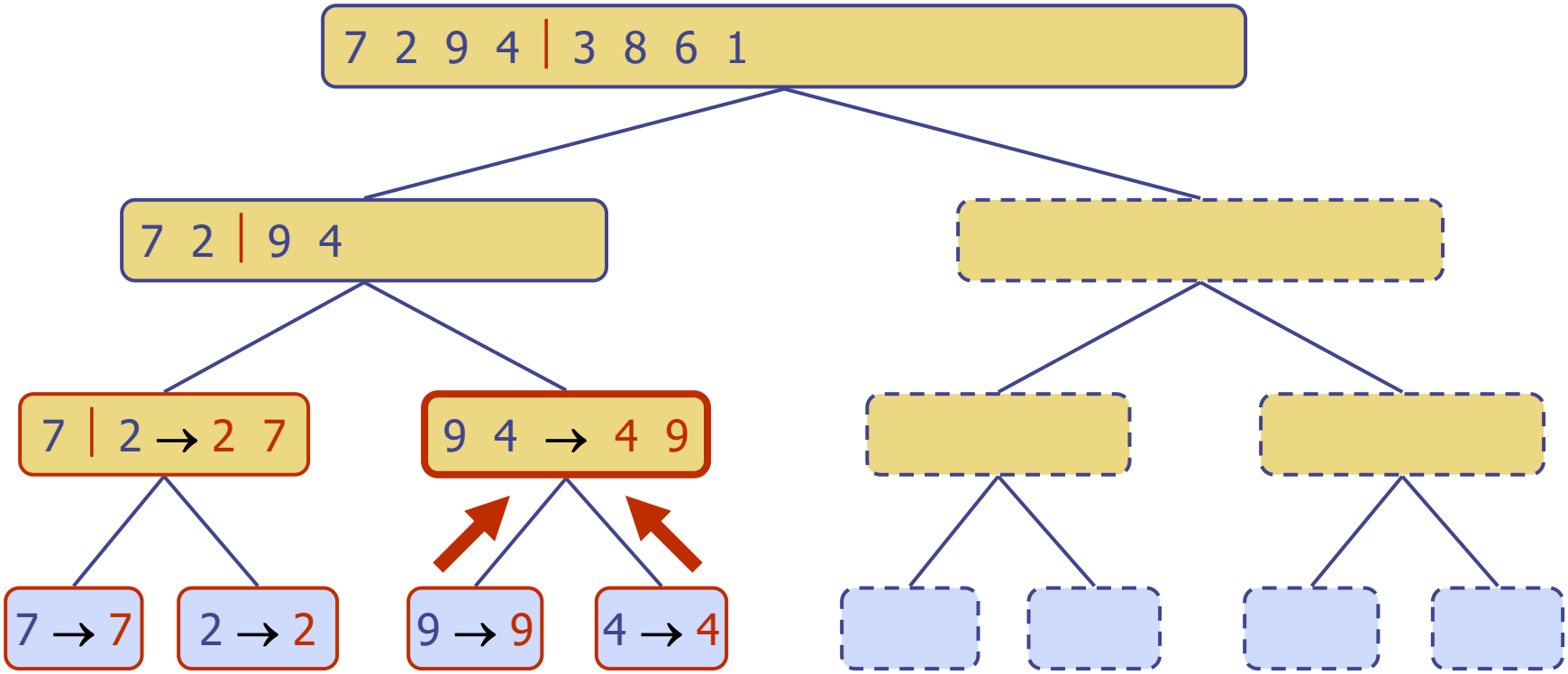
Execution Example (cont.)

- Merge



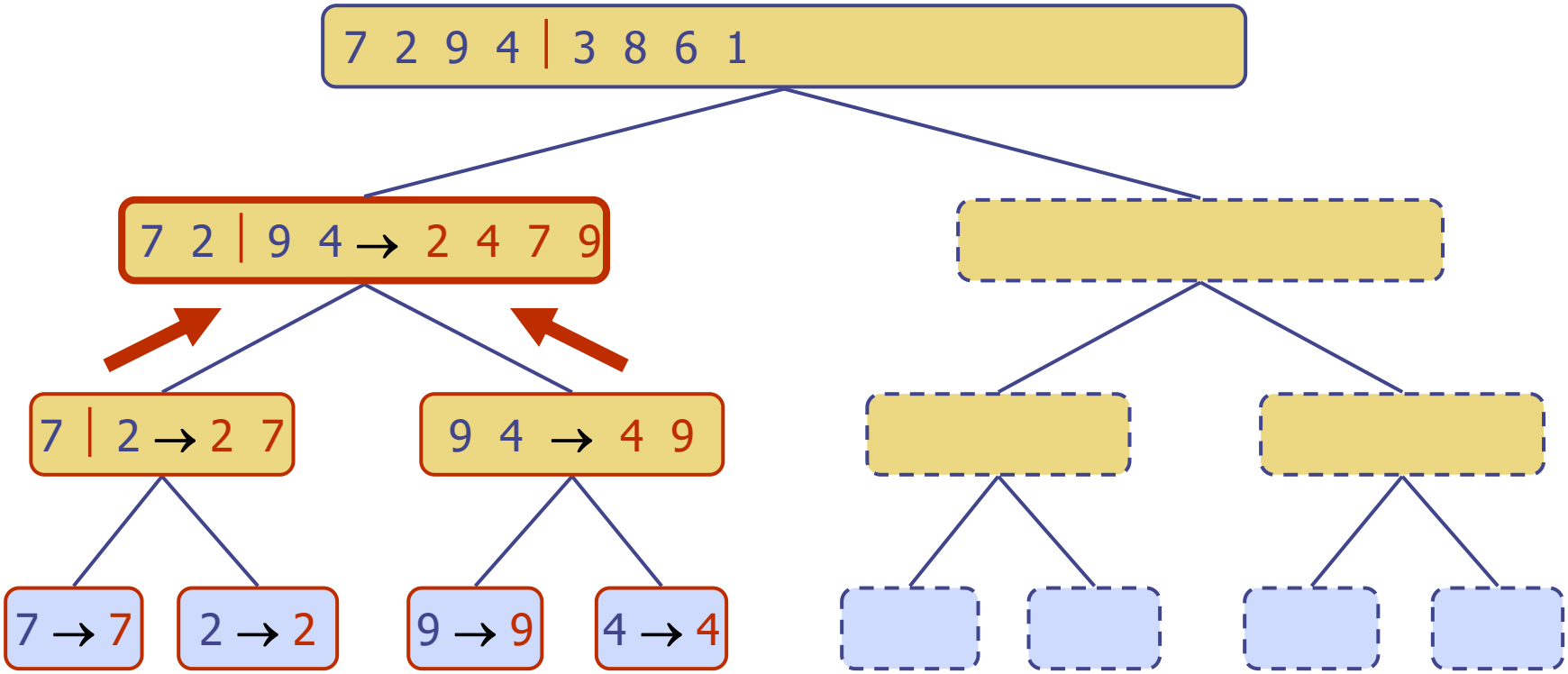
Execution Example (cont.)

- Recursive call, …, base case, merge



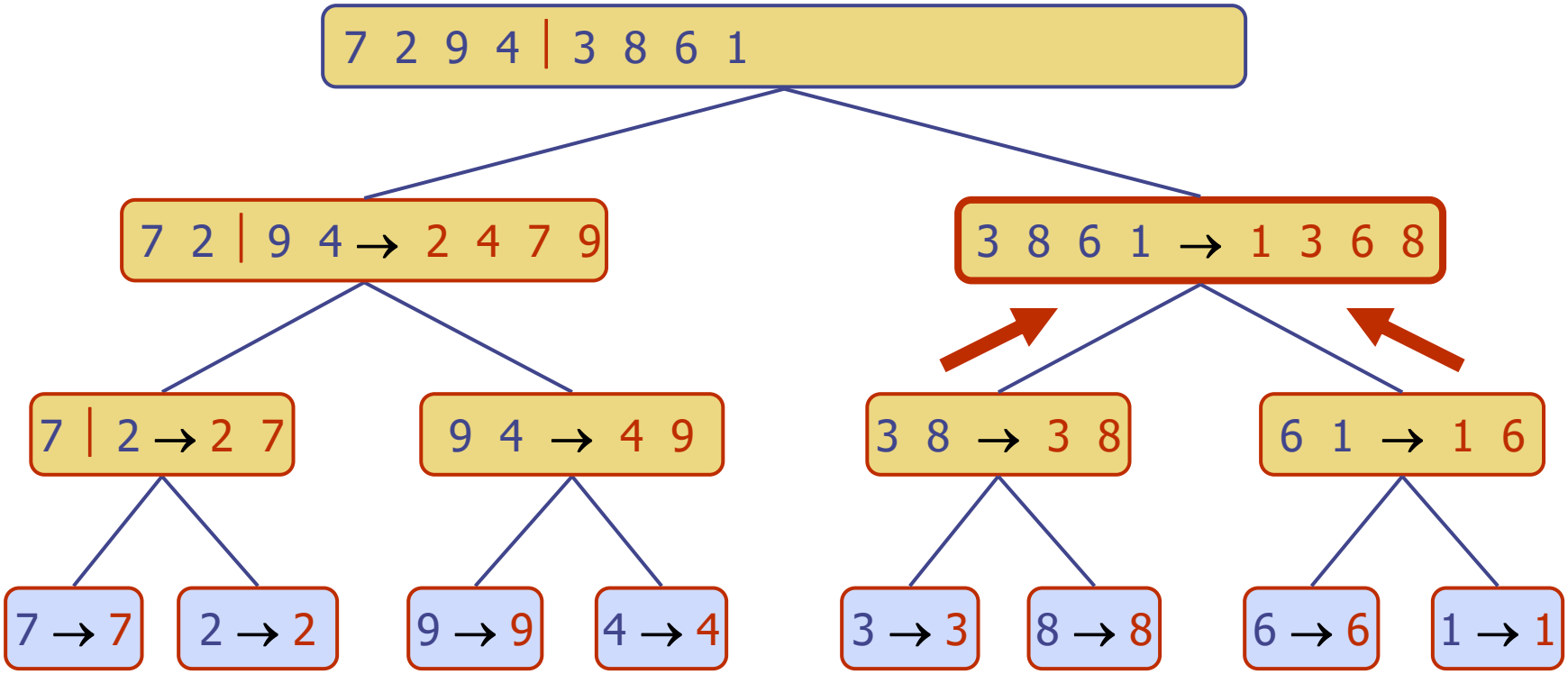
Execution Example (cont.)

- Merge



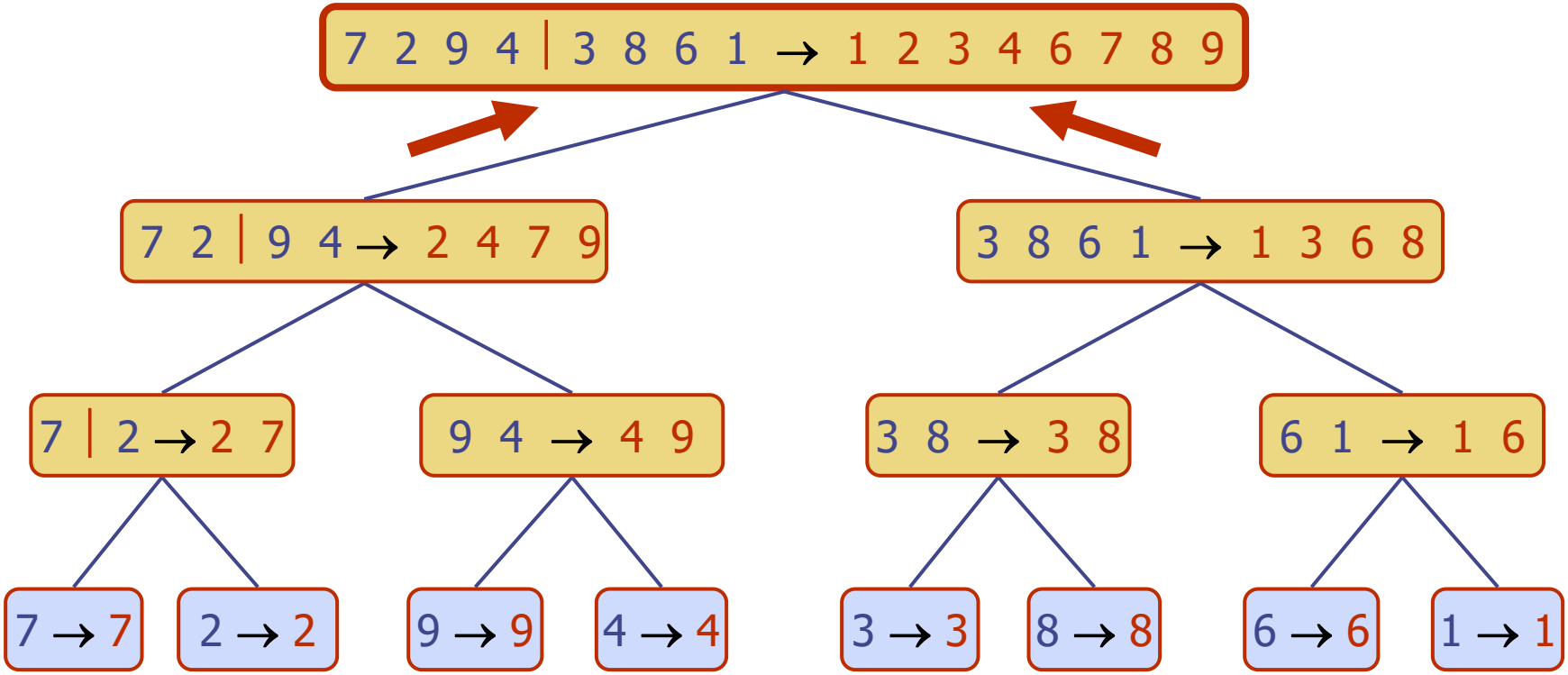
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

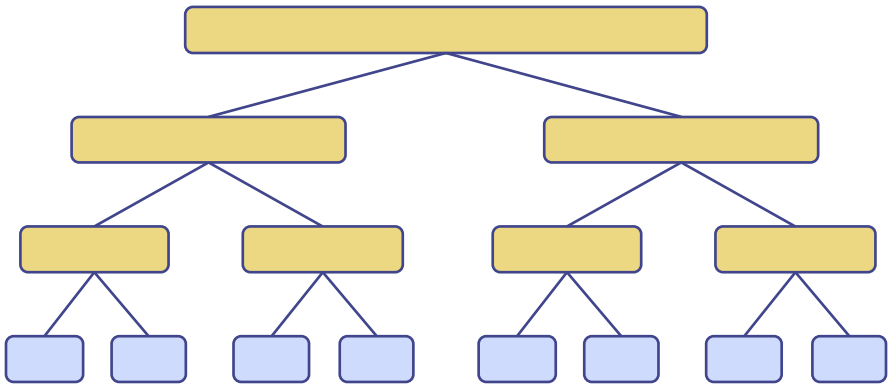
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at all the nodes at depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
  - the numbers all occur and are all "used" at each depth
    - So, each depth uses O(n) work
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Using merge sort

- Fast sorting method for arrays
- Good for sorting data in external memory – because works with adjacent indices in the array (data access is sequential)
  - It accesses data in a sequential manner (suitable for sorting data on a disk)
- Not so good with lists: relies on constant time access to the middle of the sequence

Questions to ask about sorting algorithms

- Big-Oh complexity (both time and space)?
  - Best case inputs? Worst case inputs?
- Extra workspace needed?
  Or is it `in-place'?
- Stable sorts?
- "Dynamic sorting" – how well does it do if the data is already "nearly sorted"
- Data access patterns?
  - Sequential? Random Access?
- Relevant and appropriate assertions

Aim to understand these issues for various sorting algorithms.

Minimum Expectations

- Know the Mergesort algorithm and how it works on examples
- Know and be able to justify/prove the big-Oh family of behaviours of Mergesort

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE
Recurrence Relations

Recurrence Relations

- A recurrence relation is a recursively-defined function
  - But, generally, applied to the case when the function is some measure of resources …
  - and we might only want the big-Oh family properties of the solution
- Suppose that the runtime of a program is T(n), then a recurrence relation will express T(n) in terms of its values at other (smaller) values of n.
  - (By "runtime" is usually meant something 'abstract' such as counting of operations. We do not care about difficulties of true timing in nanoseconds.)

Example: Merge Sort

- Suppose the runtime of merge-sort of an array of n integers is T(n). Then

$$T(n) = 2\,T(n/2) + b + a\,n$$

  - "2 T(n/2)" is due to having to sort the two sub-arrays each of size n/2
  - "b" is the cost of doing the split
  - "a n" is the cost of doing the merge (and any copying to/from the workspace, etc.)

Example: Merge Sort

- Suppose the runtime of merge-sort of an array of n integers is T(n).
  - Gave the recursive case
  - We also need a base-case.
  - We can take

$$T(1) = 1$$

  - As just need to check the array length is 1.
  - If make it some other number then we could just rescale the results for T(n) to match. This convention is just for simplicity and convenience.

Example: Merge Sort

- Suppose the runtime of merge-sort of an array of n integers is T(n).
  - Note that we simplified: if n is odd then we ought to have
    - "T(n/2) + T(n/2+1) + …
  - E.g. at n=9 T(9) = T(4) + T(5) + …
  - However, (generally), ignore such details, as they (generally) make no difference to the final statements of big-Oh behaviour.

Example: Merge Sort

- How would we solve

$$T(n) = 2\,T(n/2) + b + a\,n$$

- We will do some special cases:

Example 1:

- How would we solve

$$T(n) = 2\,T(n/2) \quad \text{with } T(1)=1$$

(If watching offline, pause and try)

Example 1:

- How would we solve
  $T(n) = 2\,T(n/2)$ with $T(1)=1$
- Given $T(1)=1$, what else can we evaluate?
  - $T(2)$ or $T(1/2)$
    - but want to solve for larger n, not smaller fractions, hence:
  - $T(2) = 2\,T(1) = 2$, and then can get
  - $T(4) = 2\,T(4/2) = 2\,T(2) = 4$
  - $T(8) = 2\,T(8/2) = 2\,T(4) = 8$
  - Etc.
- It seems a good guess $T(2^k) = 2^k$
- But how do we prove it in general?
  - Induction!

Example 1 (cont):

- How would we solve
  $T(n) = 2\,T(n/2)$ with $T(1)=1$
- Claim: forall k. $T(2^k) = 2^k$
- Proof by induction:
  - Base case: true at k=0. Recall: $2^0 = 1$
  - Step case. Suppose true at k (hypothesis), then need to show is true at k+1:
    $T(\,2^{(k+1)}\,) = 2\,T(2^{k+1}/2)$ (using the recurrence)
    $= 2\,T(2^k)$ (simplification)
    $= 2\;2^k$ (using the hypothesis claim)
    $= 2^{(k+1)}$ QED. As matches the claim at k+1

Make sure you understand the structure of this proof!

Example 1 (cont):

- How would we solve

  $T(n) = 2\,T(n/2)$ with $T(1)=1$
- Showed $T(2^k) = 2^k$ for all k in N, that is,

  $T(n) = n$ forall n in {1,2,4,8,16…}
- What about other values of n ? E.g. what is T(3)?
- Depends what one wants!
  - Usually (in this module) just want the growth rate
    - So we can just be imprecise with, T(n)=n for all n, and so then is $\Theta(n)$
  - Might need to refine the recurrence relation, use ceiling and floors to get integers.
    - Messy! But would be the same scaling answer, so do not (usually) bother

Example 2:

- How would we solve

$$T(n) = 2\,T(n/2) + b \qquad \text{with } T(1)=1$$

(Pause and try)

Example 2:

- How would we solve
  $T(n) = 2\,T(n/2) + b$ with $T(1)=1$
- We know $T(1)=1$, hence
  - $T(2) = 2\,T(1) + b = 2 + b$
  - $T(4) = 2\,T(4/2) + b = 2\,(2 + b) + b = 4 + (2+1)b$
  - $T(8) = 2\,(4 + (2+1)b) + b = 8 + (4+2+1)\,b$
- It seems a good guess
  $T(2^k) = 2^k + (2^{(k-1)}+\ldots+1)b$
  $= 2^k + (2^k - 1)\,b$

So $T(n) = n+(n-1)\,b = (1+b)n-b$ for n in $\{1,2,4,8\ldots\}$

Still $\Theta(n)$

Example 2: (cont)

- How would we solve

  $$T(n) = 2\,T(n/2) + b \qquad \text{with } T(1)=1$$

- Claim: $T(2^k) = 2^k + (2^k - 1)\,b$
- Proof by induction:
  - Base case: $k=0$, $T(1) = 1 + (1-1)*b = 1$
  - Step case: assume true at k
  - $T(2^{k+1}) = 2\,T(2^k) + b$

    $$\begin{aligned} &= 2\,(2^k + (2^k - 1)\,b\,) + b \\ &= 2^{k+1} + (2^{k+1} - 2 + 1)\,b \\ &= 2^{k+1} + (2^{k+1} - 1)\,b \end{aligned}$$

  QED.

Example 3:

- How would we solve

$$T(n) = 2\,T(n/2) + a\,n \qquad \text{with } T(1)=1$$

(Pause and try, even if you saw the answer before!)

Example 3:

- How would we solve
  $T(n) = 2\,T(n/2) + a\,n$ with $T(1)=1$
- We know $T(1)=1$, hence try with $n = 2^k$
  - $k=1$: $T(2) = 2\,T(1) + 2\,a = 2 + 2\,a$
  - $k=2$: $T(4) = 2\,T(4/2) + 4\,a = 2\,(2 + 2\,a) + 4\,a$
    $= 4 + 2 * 4\,a$
  - $k=3$: $T(8) = 2\,(4 + 8\,a) + 8\,a = 8 + 3 * 8\,a$
  - $k=4$: $T(16) = 2\,(8 + 3 * 8\,a) + 16\,a = 16 + 4*16\,a$
- It seems a good guess
  $T(2^k) = 2^k + k\,2^k\,a = 2^k\,(1 + k\,a)$

So $T(n) = n + \log_2(n)\,n\,a$ for $n = \{1,2,4,8\ldots\}$

Now $\Theta(n \log n)$: what we expect of merge-sort !!

Example 3: (cont)

- How would we solve

  $T(n) = 2\,T(n/2) + a\,n \qquad \text{with } T(1)=1$

- Claim: $T(2^k) = 2^k + k\,2^k\,a = 2^k\,(1 + k\,a)$
- Base case: $k=0 \quad T(1) = 1 + 0 * 1 * a = 1$
- Step case: assume true at k
  - $T(2^{k+1}) = 2\,T(2^k) + 2^{k+1}\,a$

    $= 2\,(\,2^k + k\,2^k\,a\,) + 2^{k+1}\,a$

    $= 2^{k+1} + k\,2^{k+1}\,a + 2^{k+1}\,a$

    $= 2^{k+1} + (k+1)\,2^{k+1}\,a$

  QED

Example 4:

- How would we solve

  $T(n) = 4\,T(n/2)$ with $T(1)=1$

Example 4:

- How would we solve
  $T(n) = 4\,T(n/2)$ with $T(1)=1$
- We know $T(1)=1$, hence
  - $k=1$: $T(2) = 4\,T(1) = 4 = 2 * 2$
  - $k=2$: $T(4) = 4\,T(4/2) = 4 * 4 = 16$
  - $k=3$: $T(8) = 4\,(16) = 64 = 8 * 8$
  - $k=4$: $T(16) = 4\,(8*8) = (2*8) * (2*8) = 16 * 16$
- It seems a good guess
  $T(2^k) = (\,2^k\,)^2$

So $T(n) = n^2$ for n in {1,2,4,8…}

Hence $\Theta(n^2)$

Example 4:

- How would we solve
  $T(n) = 4\,T(n/2)$ with $T(1)=1$
- Claim $T(2^k) = (\,2^k\,)^2$
- Proof by induction:
  - Base case: $k=1$ $T(1) = 1^2 = 1$
  - Step case: assume true at k.
    - $T(2^{k+1}) = 4\,T(\,2^k\,) = 2 * 2 * 2^k * 2^k = (2^{k+1})^2$ QED.

Exercise (offline): Ensure you understand the details

Example 5:

- How would we solve

$$T(n) = 4\,T(n/2) + d\,n \qquad \text{with } T(1)=1$$

Example 5:

- How would we solve

  $T(n) = 4\,T(n/2) + d\,n$ with $T(1)=1$

- We know $T(1)=1$, hence
  - $k=1$: $T(2) = 4\,T(1) + 2\,d = 4 + 2\,d = 2^2 + 2 * 1 * d$
  - $k=2$: $T(4) = 4\,T(4/2) + 4\,d = 4\,(4 + 2\,d) + 4\,d$
    $= 16 + 12\,d = 4^2 + 4*3*d$
  - $k=3$: $T(8) = 4\,(16 + 12\,d) + 8\,d = 8^2 + 8*7\,d$
- It seems a good guess that

  $T(n) = n^2 + n(n-1)\,d$

  Exercise: proof it by induction.
- So $T(n)$ is $\Theta(n^2)$. "Value of d does not matter"

Example 6:

- $T(n) = T(n/2) + d \qquad T(1) = 1$
  - E.g. From binary search of a sorted array

Example 6:

- $T(n) = T(n/2) + d \qquad T(1) = 1$
  - E.g. From binary search of a sorted array
    - k=1: $T(2) = T(1) + d = 1 + d$
    - k=2: $T(4) = T(2) + d = 1 + 2\,d$
    - k=3: $T(8) = T(4) + d = 1 + 3\,d$
    - k=4: $T(16) = T(8) + d = 1 + 4\,d$
  - Guess $T(2^k) = 1 + k\,d$
  - Hence: $T(n) = 1 + d \log_2(n)$
  - Exercise (offline): prove by induction.
  - That is, T(n) is $\Theta(\log n)$, as expected for binary search

Simple sorting?

- Bubble sort etc. do not naturally generate recurrence relations as they are not naturally recursive.
  - But could be phrased that way
  - Bubble sort
  - $T(n) = T(n-1) + d\ n$
    - $d\ n$ for a pass of the outer loop
    - $T(n-1)$ for the remaining passes – which now only need to process n-1 numbers.

Example 7:

- $T(n) = T(n - 1) + d\,n$ $\qquad T(1) = 1$
  - (Bubble sort, etc.)
  - $T(2) = T(1) + 2\,d = 1 + 2\,d$
  - $T(3) = (1 + 2\,d) + 3\,d = 1 + (2 + 3)\,d$
  - $T(4) = (1 + (2+3)d) + 4d = 1 + (2+3+4)d$
- Guess $T(n) = 1 + (2+\ldots+n)\,d$
  $= 1 + (\,n(n+1)/2 - 1\,)\,d$
- Exercise (offline): prove by induction
- Observe it is $\Theta(n^2)$ as expected.

Solving Recurrence

- General pattern
  1. Starting from the base case, use the recurrence to work out many cases, by directly substituting and working upwards in values of n
  2. Inspect the results, look for a pattern and make a hypothesis for the general results
  3. Attempt to prove the hypothesis – typically using some form of induction

  Often then extract the large n behavior using big-Oh family

  Can be long, tedious, and error-prone, but many cases are covered by a general rule with the name of "Master theorem" (next lecture)

Expectations

- Be able to extract recurrence relations from algorithms
  - Typically, used for recursive algorithms and especially "divide and conquer"

- Be able to explicitly solve (fairly simple) cases
  - Apply the recursion formula for sequence of (small) n
  - Guess pattern
  - Prove using induction
  - (You should generate multiple examples yourself and practice at solving them, and doing the induction proofs)

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE
The “Master Theorem”

Master Theorem (MT)

Consider recurrence relations of the form

$$T(n) = a\,T(n/b) + f(n)$$

- Designed for "divide and conquer" in which problems are divided into 'a' instances of a problem of size n/b.
- Aim is to be able to quickly express the "big-Oh family" behavior of T(n) for various cases of the values of a and b, and the scaling behavior of f(n).

It does not cover all cases, but does cover many useful cases.

Master Theorem

- No proof needed in this module.
  - Though you are expected to have some understanding of its structure, and why it is reasonable
- Need to learn it, and how to apply it!
- Suggest to generate and try many examples

Motivations

- Consider the special case that $f(n) = 0$
- $T(n) = a\,T(n/b)$ with $T(1) = 1$
  - $T(b) = a$
  - $T(b^2) = a^2$
  - $T(b^3) = a^3$
- So $T(b^k) = a^k$
- Exercise (offline): prove by induction

Motivations

- Consider the special case that $f(n) = 0$
  - $T(n) = a\,T(n/b)$ with $T(1) = 1$
  - Gives $T(b^k) = a^k$
- Put $n = b^k$ then
  - (for clarity in the superscripts write "$\log_b$" as "log_b")
  - $a^k = (\, b^{\log_b(a)} \,)^k$
    - Note: used the identity that $a = b^{\log_b(a)}$
    - Then we have (swapping the order of the exponents)
  - $a^k = (\, b^k \,)^{\log_b(a)}$
  - $a^k = (\, n \,)^{\log_b(a)}$
- So we finally get $T(n) = n^{\log_b(a)}$

Motivations

- Consider the special case that $f(n) = 0$
  - $T(n) = a\,T(n/b)$ with $T(1) = 1$
  - Gives $T(n) = n^{\log_b(a)}$
- But now suppose $f(n) = n^c$ for some c
- We can ask which term dominates
  - the recurrence or the f(n)?
- So need to compare the values of c and $\log_b(a)$

Master Theorem (MT): Cases

T(n) = a T(n/b) + f(n)

Results are split into 3 cases, according to comparing the growth rate of f(n) to n^$(\log_b(a))$

- Case 1: f(n) "grows slower". Recurrence term dominates. "Solution ignores f"
- Case 2: f(n) grows same – up to log factors – "mix of recurrence with a,b, and also the f(n) term"
- Case 3: f(n) grows faster. "Solution ignores recurrence terms and a,b"

MT: Case 1

$T(n) = a\,T(n/b) + f(n)$

$f(n)$ is $O(n^c)$ with $c < \log_b a$

Note: it is "<" not "<=" and it is a "big-Oh"

Then $T(n)$ is $\Theta(n\text{^}(\log_b a))$

That is, $T(b^k)$ grows as $b^{k \log_b a} = a^k$,

as expected from earlier

MT: Case 1: Example

$T(n) = 2\,T(n/2) + d$

$a = 2$, $b=2$ so $\log_b(a) = \log_2(2) = 1$

$f(n)$ is $O(1)$ which is $O(n^c)$ with $c = 0$
and so we have that $c < \log_b(a)$

Hence MT gives that $T(n)$ is $\Theta(n)$

MT: Case 2

T(n) = a T(n/b) + f(n)

if

f(n) is $\Theta(n^c(\log n)\hat{}k)$

with c = $\log_b$ a and $k \geq 0$

(Note: it is "c =" and Big-Theta)

Then T(n) is $\Theta(n^c\,(\log n)\hat{}(k+1)\,)$

Note the growth depends on both the recurrence, a,b, and also depends on f via k.

MT: Case 2: Example

$T(n) = 2\,T(n/2) + 3\,n + 5$

$f(n)$ is $\Theta(\, n\,(\log n)^k\,)$

with

$c = \log_2 2 = 1,$

and

$k=0$

Then $T(n)$ is $\Theta(n \log n)$

(Same as merge sort of previous lecture)

MT: Case 3

T(n) = a T(n/b) + f(n)

f(n) is $\Omega(n^c)$ with c > $\log_b$ a

Notice: it is "c > .." and big-Omega!

And f(n) satisfies the "regularity condition"

a f(n/b) <= k f(n) for some k < 1

Then T(n) is $\Theta(f(n))$.

Growth is dominated by f(n) and so a,b of the recurrence are not used.

MT: Case 3 : Example

$T(n) = 2\,T(n/2) + n^2$

$f(n)$ is $\Omega(n^c)$ with $c = 2 > \log_b a = \log_2 2 = 1$

Also, $f(n)$ satisfies the "regularity condition"

$2\,f(n/2) = 2\,(n/2)^2 <= k\,f(n)$ with $k=1/2$

Then $T(n)$ is $\Theta(n^2)$

Regularity Condition (case 3)

- a f(n/b) <= k f(n) for some k < 1
- Suppose f(n) = d n^c then we need
  - a d (n/b)^c <= k d n^c for some k < 1
  - a / b^c <= k for some k < 1
  - a / b^c < 1
  - a < b^c
  - Now take log_b of both sides
  - Need log_b(a) < c
  - Which is already satisfied for case 3 to apply.
  - So is not a new condition in this case (needed for more complex cases)
- (Note: this is included for completeness; usually not relevant)

MT Example

- $T(n) = 4\,T(n/2) + d\,n$ with $T(1)=1$
- $a=4$, $b=2$
- so $\log_b a = \log_2 4 = \log_2 2^2 = 2$

- Note that $f(n)$ is $O(n^c)$ with $c=1$
  - Hence $c < 2$, and is big-Oh
- Hence is case 1.

- Hence is $\Theta(\,n^2\,)$
  - Matches the exact solution in previous lecture.

Expectations

- Know and understand the Master Theorem (MT)
  - Be able to apply it to examples
  - (Do not need to be able to prove the MT itself!)

- May well be asked to solve a recurrence relation exactly, and then also to solve it using the Master Theorem

COMP2054-ADE

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Quicksort

Motivations

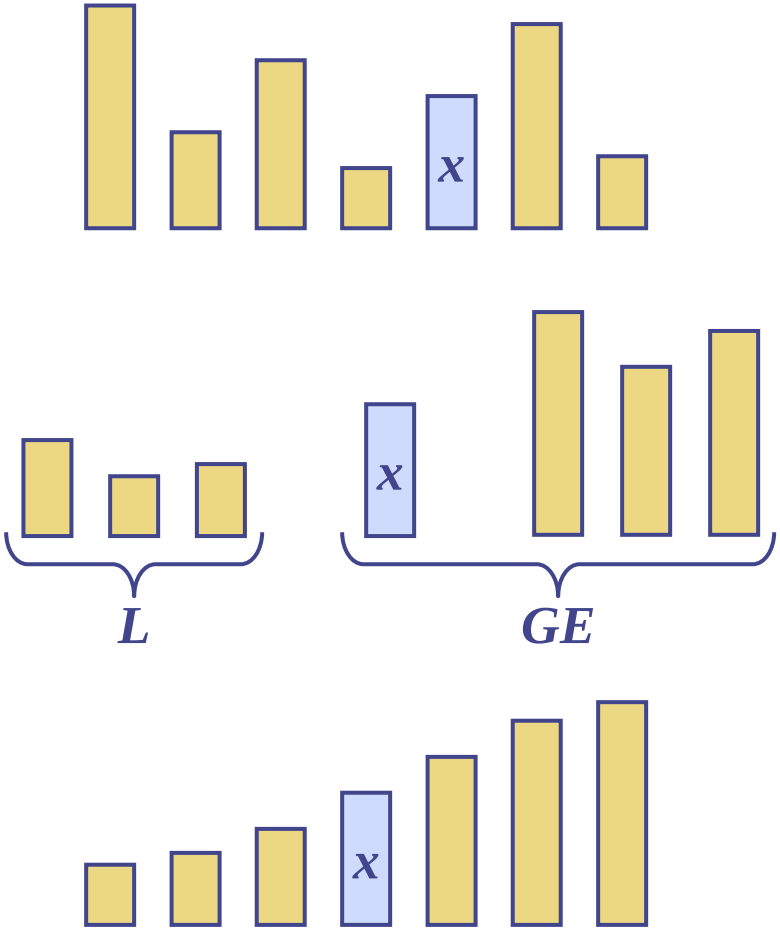
- In merge sort the `divide' is simple, and the `merge' (relatively) complicated
- Can we swap this and make the 'merge' simple?
  - Answer: make the `divide' more complicated so that the merge becomes `concatenate'
- Analogy: sort a pack of cards by
  1. divide into 'red' and 'black' cards
  2. divide by suit (red into hearts and diamonds,…)
  3. divide by value …

When is `merge' simple?

- When the lists A and B are sorted and known to be in disjoint ordered ranges
  - all of elements of A are smaller than all those of B
- If A and B are stored as consecutive sub-arrays, then merge needs no work at all:
  - Just "forget the boundary"

Quick-Sort

- Quick-sort is a (randomized) sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick an element x (called pivot) and partition S into
    - L: elements less than x
      - *Have to be careful it is not empty*
    - GE: elements greater than or equal to x
    - *Pivot is often picked as a random element*
  - Recur: sort L and GE
  - Conquer: join L, GE



Partition of lists (using extra workspace)

- Suppose store L, EG as separate structures (e.g. as arrays, vectors or lists)
- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning (or end) of the sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

“In-place” or “extra workspace”?

- For sorting algorithms (and algorithms in general) an important issue can be how much extra working space they need besides the space to store the input
- “**In-place**” means they only a “little” extra space (e.g. O(1)) is used to store data elements.
  - The input array is also used for output, and only need a few temporary variables
  - Exercise: check that bubble-sort is “in-place”
  - Previous “merge” used extra O(n) array (can be made in-place, but messy and so we ignore this option)

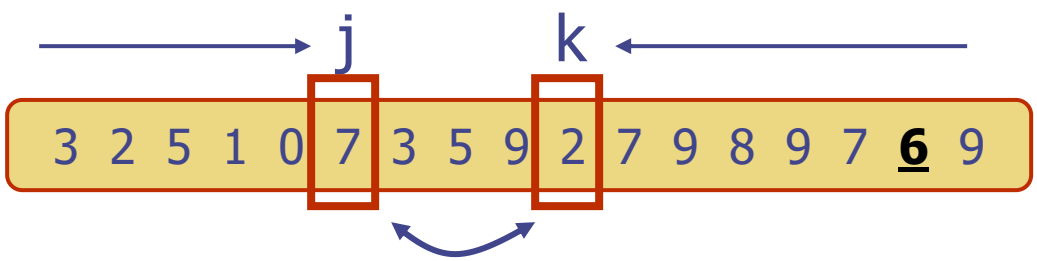
Partitioning arrays "in-place"

- Perform the partition using two indices to do a "2-way split" of S into L and E+G.

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross (or meet, until j >= k):
  - Scan j to the right until finding an element ≥ pivot.
  - Scan k to the left until finding an element < pivot.
  - Swap elements at indices j and k



The scans are not done in 'lock step' but independently work inwards. Do some examples and make sure you understand how and why this works!!

Partitioning

- Partitioning is actually more subtle
- Have to make sure that we make progress – specifically that neither of the partitions are empty
  - GE always contains the pivot, so is okay.
  - But L could be empty
    - So the pivot should not be a minimum element
- Or simpler is to do a "3-way" split:
  - check and move the pivot to the left and do a 3-way split into L, {pivot}, E+G
  - And so always ensure a one copy of the pivot (there could be many) is moved to the middle.

Exercise (offline)

- Write Java code to do the partition and check that it works on some examples
  - (It is only 10-20 lines of code, but will greatly help clarify the algorithm.)
- Investigate and explore some versions of how to do it effectively and efficiently

Quicksort Overall Implementation

With the previous (2-way) split:

```
public static void recQuickSort(int[] arr, int left, int right) {
   if (right - left <= 0) return;
   else {
      int border = partition(arr, left, right); // “crossing position”
      recQuickSort(arr, left, border);
      recQuickSort(arr, border+1, right);
   }
}
```

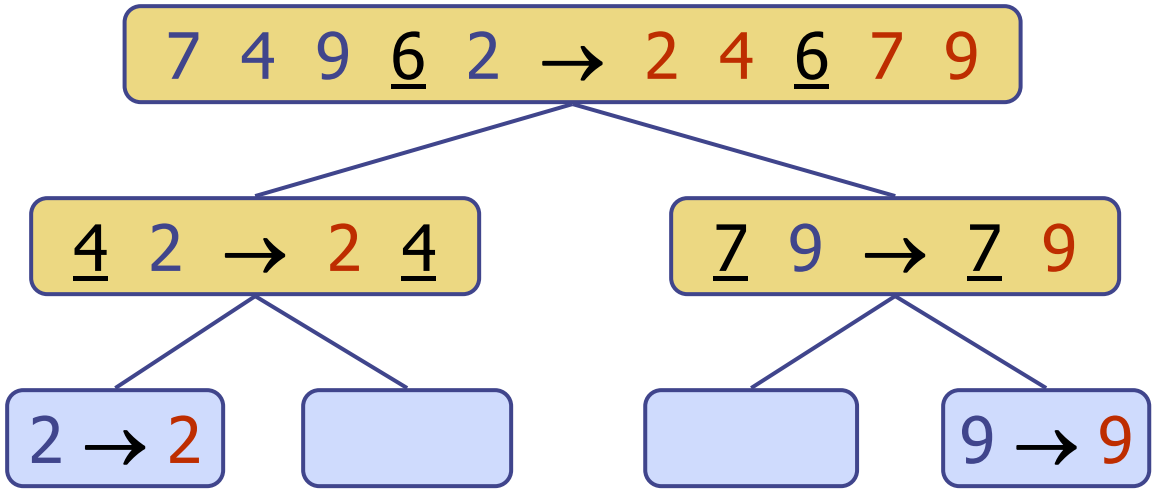
Quicksort Overall Implementation

With a 3-way split (assuming just one copy of the pivot):

```
public static void recQuickSort(int[] arr, int left, int right) {
  if (right - left <= 0) return;
  else {
    int border = partition(arr, left, right); // pivot position
    recQuickSort(arr, left, border-1);
    recQuickSort(arr, border+1, right);
  }
}
```

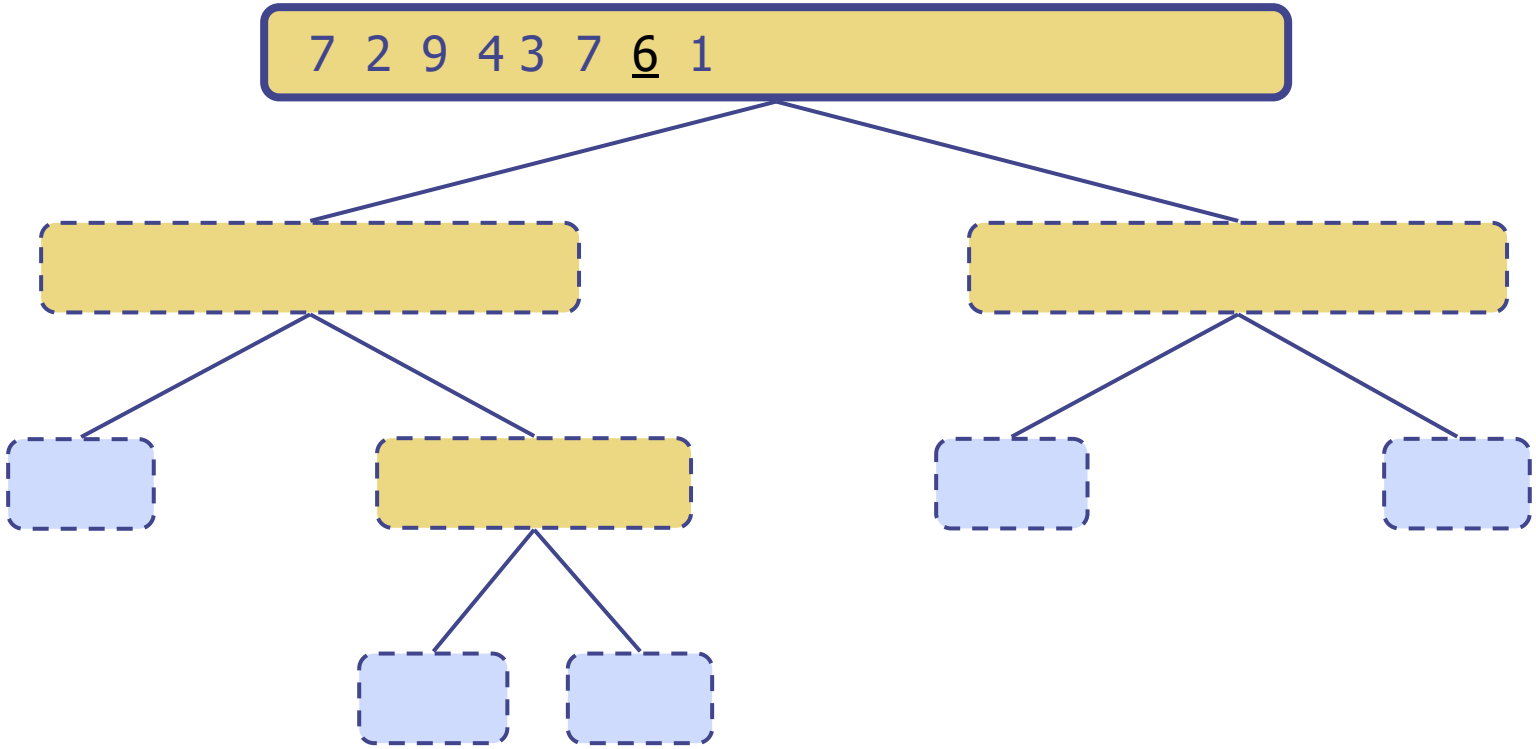
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1
  - Example shows 3-way split (one copy of the pivot is `removed' on each partition).



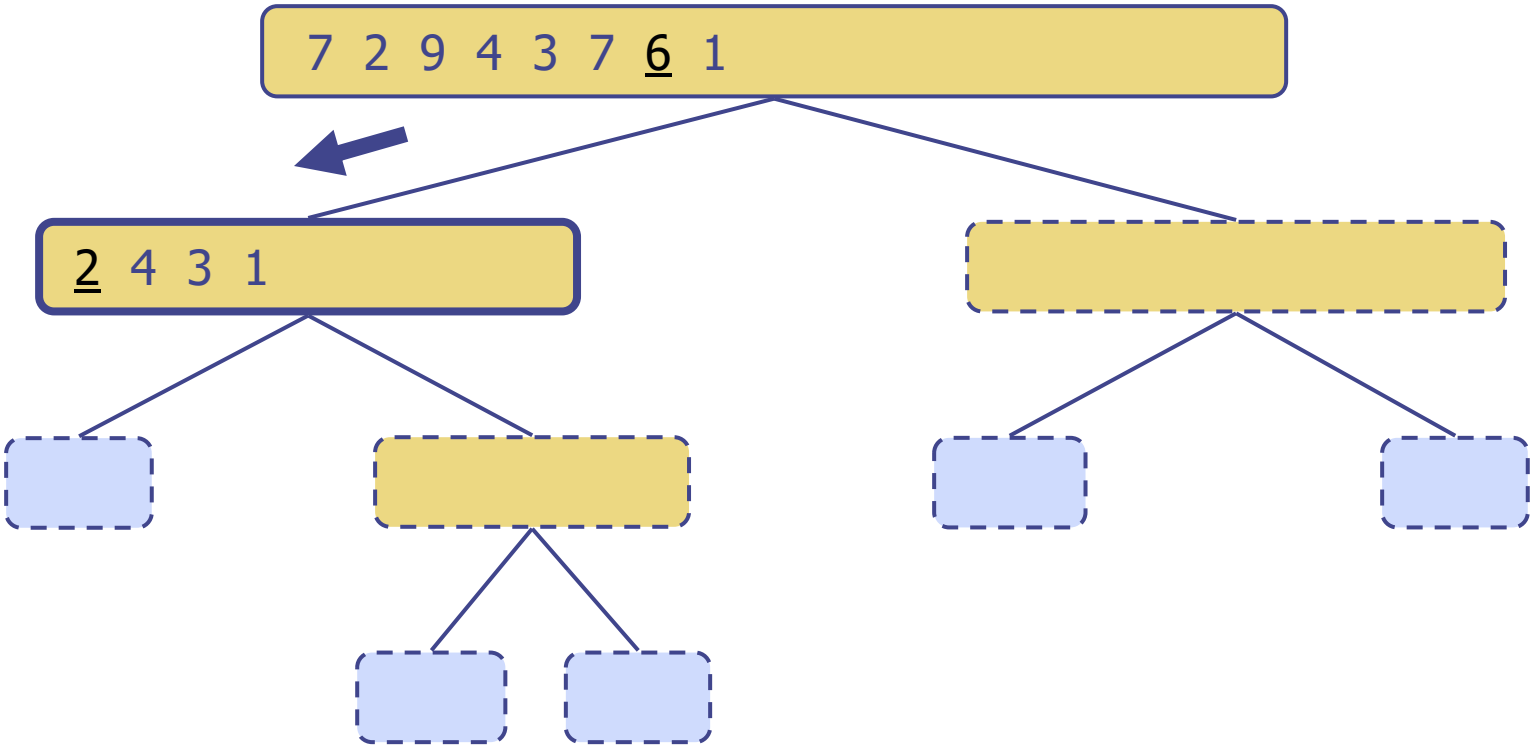
Execution Example

- Pivot selection



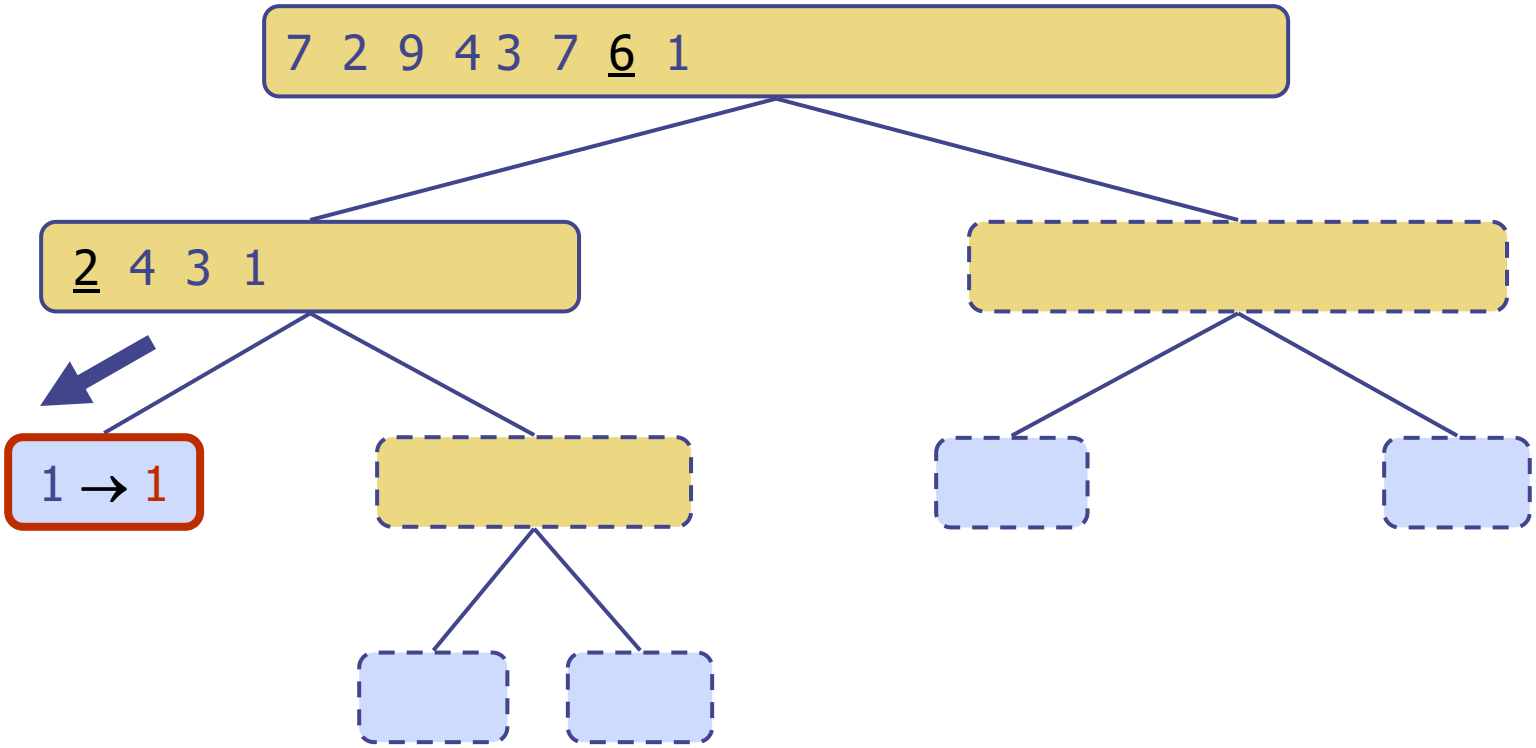
Execution Example (cont.)

- Partition, recursive call, pivot selection



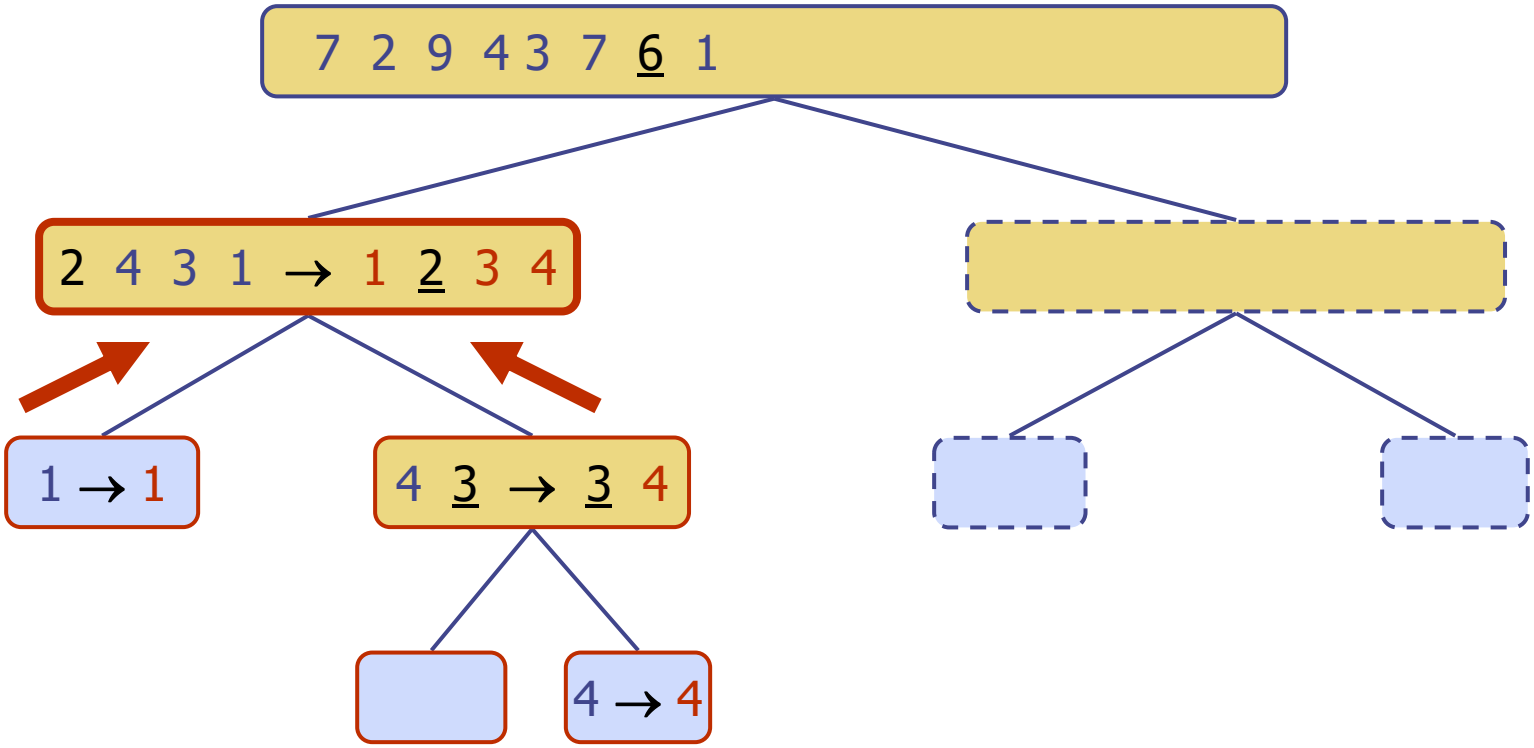
Execution Example (cont.)

- Partition, recursive call, base case



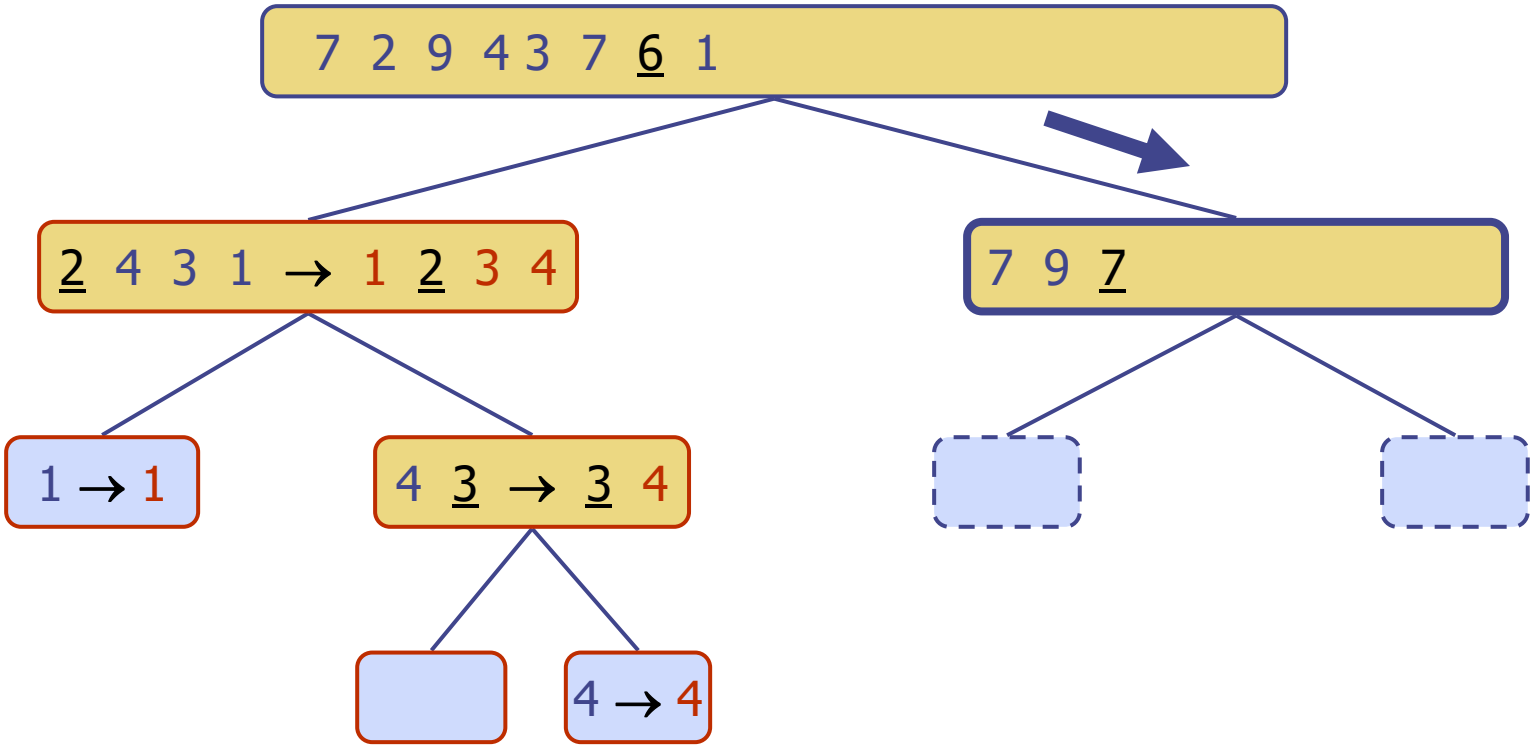
Execution Example (cont.)

- Recursive call, …, base case, join



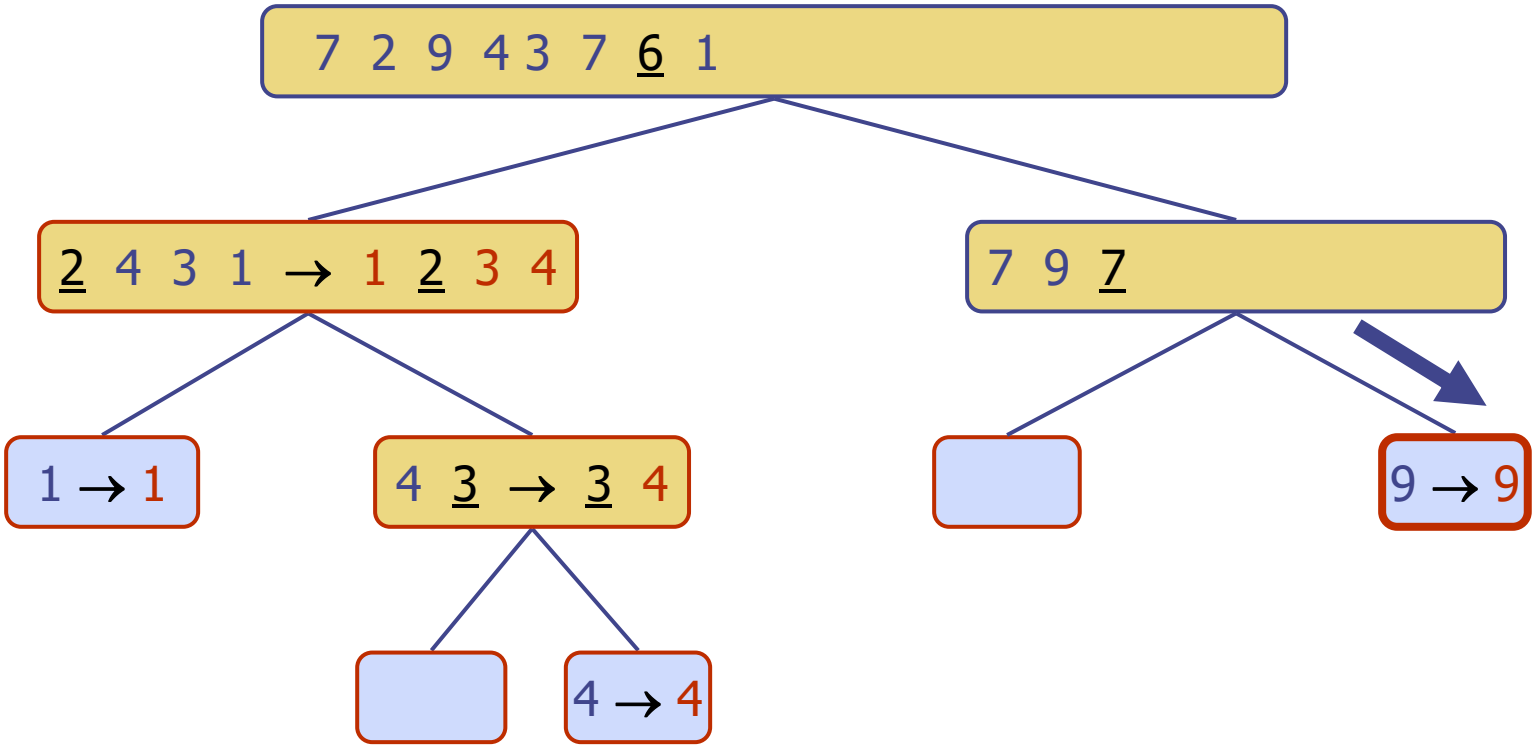
Execution Example (cont.)

- Recursive call, pivot selection



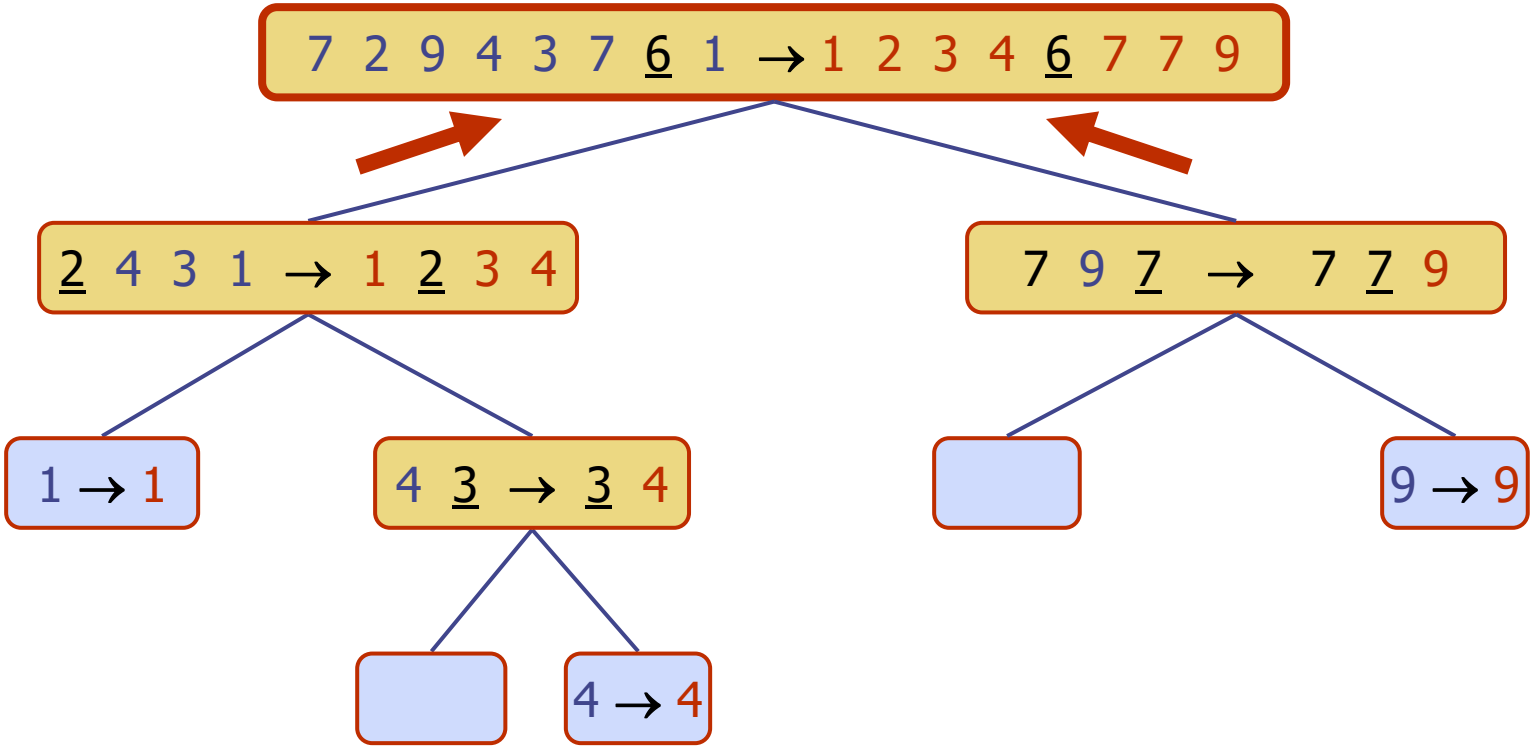
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



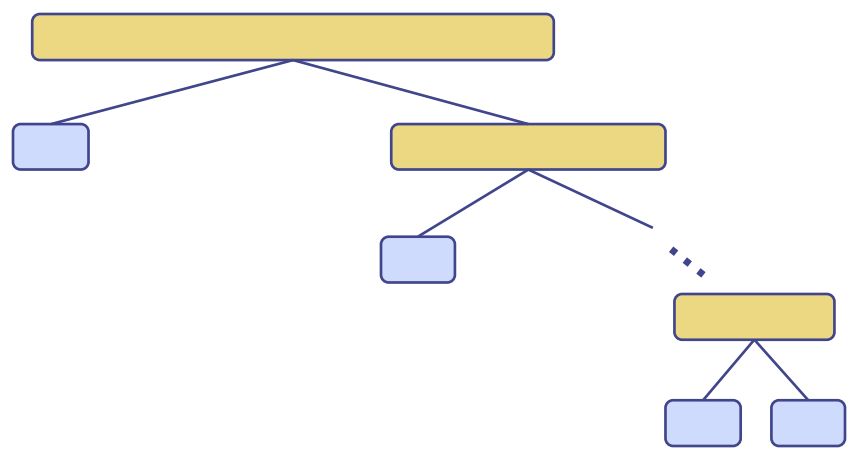
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and E+G has size $n-1$ and the other has size 1
  - (3-way split, so the pivot is always "removed", so that some progress is made)
- The running time is proportional to the sum

$$n + (n-1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Best-case Running Time

- The best case for quick-sort occurs when the pivot is the median element
- The L and G parts are equal – the sequence is split in halves, like in merge sort
- Thus, the best-case running time of quick-sort is $O(n \log n)$

Average-case Running Time

- The average case for quick-sort: half of the times, the pivot is roughly in the middle
- Thus, the average-case running time of quick-sort is $O(n \log n)$ again
  - Detailed proof in textbook
- Basic idea: suppose that the pivot is always in the middle third
  - The both L and E+G are size at least n/3, and at most 2n/3 = n / (3/2)
  - So height of the call tree is the number of times we can divide n by (3/2)

Motivations for quicksort

- Why do we select a pivot? I.e. what advantages might quicksort ever have over mergesort?
  - Because it can be done "in-place"
    - Uses a small amount of workspace
    - Because the "merge" step is now a lot easier!!
  - The "split" is more complicated, and the merge "much" easier – but turns out that the quick-sort split is easier to do in-place than the merge-sort merge

Minimum Expectations

- For both merge- and quicksort:
  - know the algorithm and how it works on examples
  - know and be able to justify/prove their big-Oh behaviours

COMP2054-ADE

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Comparison Based Sorting

Lower Bound on Efficiency

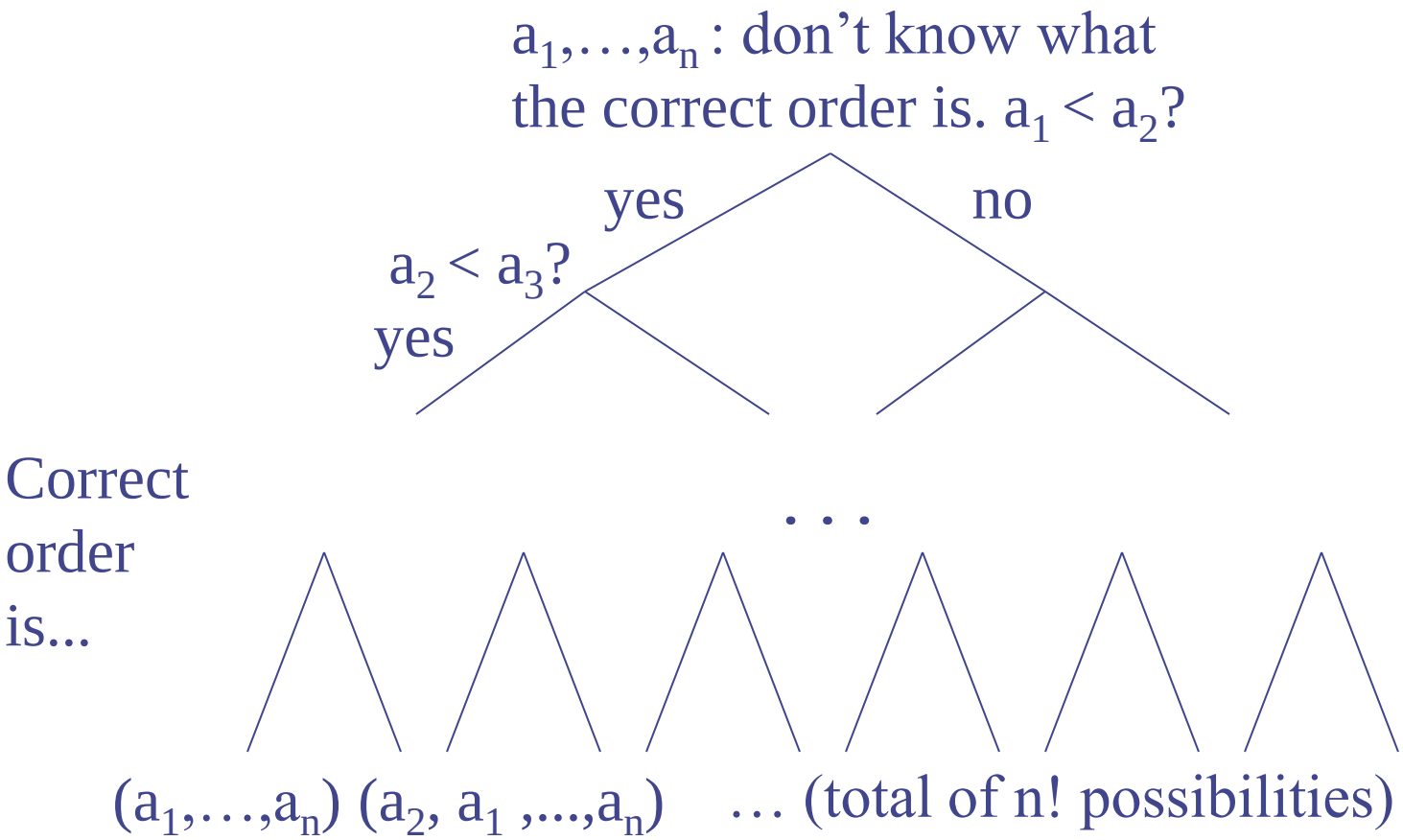
Comparison sorting

- A sorting algorithm is a comparison sorting algorithm if it uses comparisons between elements in the sequence to determine in which order to place them
- Examples of comparison sorts: bubble sort, selection sort, insertion sort, heap sort, merge sort, quicksort.
- Example of a sort that is ***not*** comparison-based: bucket sort
  - Runs in O(n), but relies on knowing the range of values in the sequence (e.g. "integers between 1 and 1000").

Lower bound for comparison sort

- We can model sorting which depends on comparisons between elements as a binary decision tree.
- At each node, a comparison between two elements is made; there are two possible outcomes and we find out a bit more about the correct order of items in the array.
- Finally arrive at full information about the correct order of the items in the array.

Comparison sorting



How many comparisons?

- If a binary tree has n! leaves, than the minimal number of levels (assuming the tree is perfect) is ($\log_2$ n!) +1.
- This shows that O(n log n) sorting algorithms are essentially optimal
  - $\log_2 n!$ is not equal to n $\log_2 n$, but has the same growth rate
- **Comparison-based sorting cannot do better than O(n log n)**
  - Technically, it uses Stirling's approximation https://en.wikipedia.org/wiki/Stirling%27s_approximation that

    $\log_2(n!) = n \log_2(n)$ + "smaller terms"

    - Note: you should know the this approximation, but (obviously) do not need to know the proof.

Questions to ask about sorting algorithms

- Big-Oh complexity (both time and space)?
  - Best case inputs? Worst case inputs?
- Extra workspace needed?
  Or is it `in-place'?
- Stable sorts?
- "Dynamic sorting" – how well does it do if the data is already "nearly sorted"
- Data access patterns?
  - Sequential? Random Access?
- Relevant and appropriate assertions

Aim to understand these issues for various sorting algorithms.

Minimum Expectations

- Know the meaning of ‘comparison-based sorting’ and the resulting O(n log n) lower bound on complexity.

COMP2054-ADE

The Vector Data Type, and Amortised Analysis

Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Vector ADT

- The "Vector" is an Abstract Data Type, ADT, corresponding to generalising the notion of the "Array" (Concrete Data Type, CDT)

- Key idea:
  - The "index" of an entry in an array can be thought of as the "number of elements preceding it"
  - E.g. in an array A, then A[2] has two elements, A[0], A[1] that precede it
  - In these lectures it is then called "rank"
  - The notion of "rank" can be more general than the idea of index:
    - Not necessarily implemented as "C-style with pointers" A[i]=*(A+i)

The Vector ADT

- The Vector ADT is based on the array CDT, and stores a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its **rank, i.e. the number of elements preceding it**
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)
- Main vector operations:
  - object elemAtRank(integer r): returns the element at rank r without removing it
  - object replaceAtRank(integer r, object o): replace the element at rank with o and return the old element
  - insertAtRank(integer r, object o): insert a new element o to have rank r
  - object removeAtRank(integer r): removes and returns the element at rank r
- Additional operations size() and isEmpty()

Vector as a Stack

- A common usage of a Vector is as a "Stack"
  - add and remove elements at the end
  - i.e. maximum rank values:
  - push(o) - insertAtRank(size(), object o) – hence the new element is after all the existing ones
    pop() - removeAtRank(size()) – hence the element removed is after all the existing ones

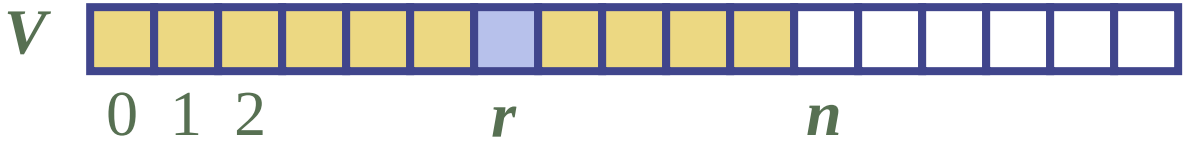
- This is very useful if reading elements from a file, and not knowing how many there will be

Applications of Vectors

- There is not an automatic limit on the storage size
  - unlike arrays of a fixed size
- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for many algorithms
  - Components of other data structures

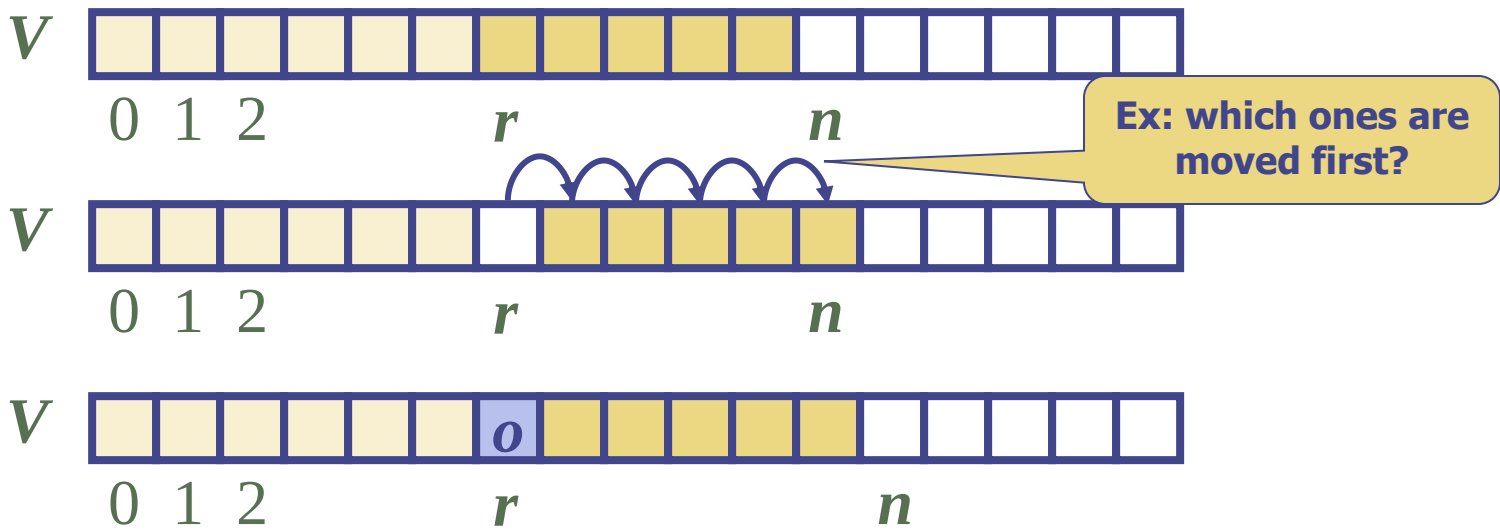
Array-based Vector

- Use an array V of size N as the CDT
- A variable n keeps track of the size of the vector (number of elements currently stored)
- Operation $elemAtRank(r)$ is implemented in $O(1)$ time by simply returning $V[r]$



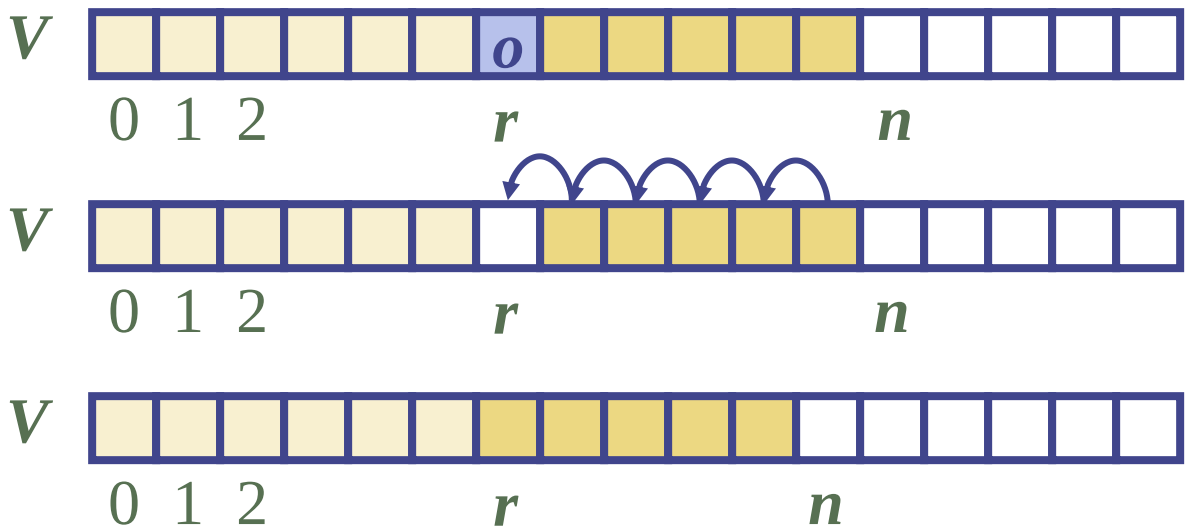
Insertion

- In operation $insertAtRank(r, o)$, we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \ldots, V[n-1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Deletion

- In operation ***removeAtRank***(r), we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r+1]$, …, $V[n-1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Performance

- In the array-based implementation of a Vector
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***elemAtRank*** and ***replaceAtRank*** run in $O(1)$ time
  - ***insertAtRank*** and ***removeAtRank*** run in $O(n)$ time
  - ***push*** runs in O(1) time, as do not need to move elements
    - **unless need to resize the array**
  - ***pop*** runs in O(1) time
- In an ***insertAtRank*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growable Array-based Vector

- In a push (insertAtRank(n)) operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
  - But the resizing has a high cost – O(n) as need to copy at all n elements
- How large should the new array be?
  - **incremental strategy**: increase size by a constant c
  - **doubling strategy**: double the size

```
Algorithm push(o)
  if n = V.length -1
then
    A ← new array of
          size …
    for i ← 0 to t do
      A[i] ← V[i]
    V    ← A
    n    ← n + 1
    V[t] ← o
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
  - "push" means "add an element at the end" – treat it as a stack
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized** time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

Meaning of “Amortize”

- See http://www.thefreedictionary.com/amortize or similar if you are not familiar with the word.
- It refers to writing off, or paying off, debts over a period of time.
- Similar to the way a mortgage for a house is paid back over many years, as opposed to needing to pay all in one go.

Remarks on Amortised Analysis

- Suppose some individual operation (such as 'push') takes time T in the worst-case
- Suppose do a **sequence** of operations:
  - Suppose s such operations take total time T_s
  - Then sT is an upper-bound for the total time T_s
  - But, such an upper-bound might not ever occur.
- The time T_s might well be o(sT) even in the worst-case
  - the average time per operation, T_s /s can be the most relevant quantity in practice
  - E.g. if 'push'ing a long sequence of elements into a Vector – e.g. when reading from a file

Question (pause and try):

- Why is amortised analysis different from the average case analysis?

Question:

- Why is amortised analysis different from the average case analysis?
- Answer:
  - "Amortised": (long) **real sequence** of dependent operations
  - vs.
  - "Average": **Set** of (possibly independent) operations

Describing amortised costs

Note the usual big-Oh family is still used to describe amortised analysis

- Recall Big-oh is just describing functions
  - It is not limited to "worst case of an algorithm"!
- We have different measures of the runtime cost:
  - Worst-case "cost per operation of a sequence"

  not just
  - "Worst case of a single operation"

Incremental: Example

- Take c=3, with start capacity of 3, then a sequence of pushes might have costs for each push as follows:

1,1,1,3+1,1,1,6+1,1,1,9+1,1,1,12+1,1,1,15+1,1,1,18+1,...

- A constant fraction, 1/3, of the pushes have cost O(n).
- Average, per push operation, is O(n).
- Much worse than O(1) cost without resizing!

Example: In detail

- [- , - , -] n=0 Starting point
  - empty, but capacity=3
  - "-" means "not yet used"
- Suppose do a sequence of "push(0)", we get the sequence of costs (primitive operations):
  1. [0 , - , -] n=1, **cost=1**
  2. [0 , 0 , -] n=2, **cost=1**
  3. [0 , 0 , 0] n=3, **cost=1**
  4. [0 , 0 , 0, 0 , - , -] n=4, **cost=3+1**
     - cost= "3 for the copy" + "1 for the push(0)"
  5. [0 , 0 , 0, 0 , 0 , -] n=5, **cost=1**
  6. [0 , 0 , 0, 0 , 0 , 0] n=6, **cost=1**
  7. [0 , 0 , 0, 0 , 0 , 0 , 0 , - , -] n=7, **cost=6+1**
     - cost= "6 for the copy" + "1 for the push(0)"
  8. etc, etc...

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- Each "replace" costs the current size
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of a push operation is $O(n)$
- This is bad as the normal cost of a push is O(1).

Doubling: Example

- With start capacity of 3, then a sequence of pushes starting from an empty vector, might have costs for each push in turn of

1,1,1,3+1,1,1,6+1,1,1,1,1,1,12+1,1,1,1,1,1,1,...

- The 3,6,12,… are the costs for the resizing
- The fraction of pushes having cost O(n) drops with n.
- What is the average?

Doubling: Example

- With start capacity of 2, then a sequence of pushes might have costs

1,1,2+1,1,4+1,1,1,1,8+1,1,1,1,1,1,1,1,1,1,...,16+1,…

- For every push of cost O(n) we will be able to do another O(n) pushes of cost O(1) before having to resize again.
- The O(n) cost on resizing can be 'amortised' (spread) over n other O(1) operations
- Gives an average of O(1) per operation.

Doubling: Example

- With start capacity of 2, then a sequence of pushes might have costs

  1,1,2+1,1,4+1,1,1,1, ...

- The cost of the doubling can be "spread" over the later operations and so might be counted as

  1,1,1+1,1+1,1+1,1+1,1+1,1+1,...

where the red is a cost that has been moved.

Analogy: save £1/day "for a rainy day"

- This 'view' makes it clearer that the net effect will just be a (rough) doubling of the original costs

Doubling Strategy Analysis

- Question: for ***n*** 'pushes' how many times is the array grown?
- For simplicity assume n is a power of 2
  - We replace the array $\boldsymbol{k} = \log_2 \boldsymbol{n}$ times

- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of ***n*** push operations is proportional to

$$\boldsymbol{n} + 1 + 2 + 4 + 8 + \ldots + 2^{\boldsymbol{k-1}}$$

Examples

- n=1 #doublings=0
  - log2 n = 0
- n=2 #doublings=1
  - log2 n = 1
- n=4 #doublings=2
  - log2 n = 2

Example: n=4

'|' is 'End of stack' marker

- start [| _]
  - capacity=1, size=0
- push(A) [A |]
- push(B) [A B |]
  - needed: 1 double – 1 copies
- push(C) [A B C | _]
  - needed: 1 double – 2 copies
- push(D) [A B C D |]

Exercise (offline): Do this is full detail; as done earlier for the incremental strategy.

Recall: Geometric sums

Want to find S

$$S = 1 + 2 + 2^2 + 2^3 + \ldots + 2^{\boldsymbol{k}}$$

Standard trick:

$$2S = 2 + 2^2 + 2^3 + \ldots + 2^k + 2^{k+1}$$

So

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

Hence $S = 2^{k+1} - 1$

I.e. "the next term minus one"

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^{k-1} =$$

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

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

- Amortized time of a single push operation is $O(1)$
- That is, no worse than if all the needed memory was pre-assigned!
  - (Big-Oh hides the constant factor '2' extra cost.)

Offline Exercises/Discussion

- When might you still want to use incremental increase rather than doubling?

- Why is it doubling rather than tripling? Or increasing by some other constant factor?

  - Try redoing the analysis with a arbitrary growth factor *b*
  - *As ever: implement it all!*

C/C++ comments

- Doubling vector size might be made even more efficient if use realloc rather than malloc or new
  - Internally it can (often not always) just extend the space allocated to the array, and so avoid the need for a copy
  - It can use “memcpy” which is direct copy rather than via individuals

Expectations

- The “Vector” data structure
- The big-Oh costs of various single operations
- The amortised cost of a sequence of operations
- The amortised complexities of different strategies for resizing of the underlying array

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE

Priority Queues & Heaps

ADT and CDT

Terminology:

- Abstract Data Type (ADT)
  - Only concerned with specifying the interface
  - “Behaviour as seen from the outside”
  - Specifies the methods provided - and possibly requirements on their complexity, e.g. “must run in O(n)”
- Concrete Data Type (CDT)
  - Some structure and algorithm used for an actual implementation
  - “Behaviour as seen from the inside”

ADT and CDT utility

Standard object-oriented reasons:

- Separation of the "specification" from the "implementation details"
- Allows different CDTs to be explored for the same ADT
- Quickly swap or improve CDTs without users of the ADT having to change their code

- **Watch out for: difference between**
  - **names used for ADTs**
  - **names used for associated CDTs**

Priority Queue ADT

- A priority queue stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v)
    inserts an entry with key k and value v
  - removeMin()
    removes and returns the entry with smallest key
  - Note: unlike Map (seen later), there is no requirement of a method find(k).
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby passengers
  - Auctions
  - Stock market
  - Printer queues

Portions of slides are © 2004 Goodrich, Tamassia

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined (between all different pairs)
  - Just need a "compare" operation between the keys and that behaves like <= or < as needed

- Two distinct entries in a priority queue can have the same key
  - i.e. that compare as being equal

Simple Implementations, CDT, of a Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the head or tail of the list
  - removeMin and min take $O(n)$ time since we have to traverse the entire list to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the position to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the head

Exercise (offline): is singly-linked sufficient? Or is doubly-linked needed?

Portions of slides are © 2004 Goodrich, Tamassia

General Remark

- "Some operations are O(1) and others O(n), so the average is not too bad" ??
  - NO NO NO!!!
- The average could still be O(n)
- Suppose, that
  - 99% take time 1
  - 1% take time n

  The average (n+ 99)/100 is still O(n)
- This only improves if the worst-case operations are 'rare', otherwise the expensive operations still dominate
  - E.g. see the study of amortised complexity. If the resizing is "rare" then it does not dominate the cost.

Aim:

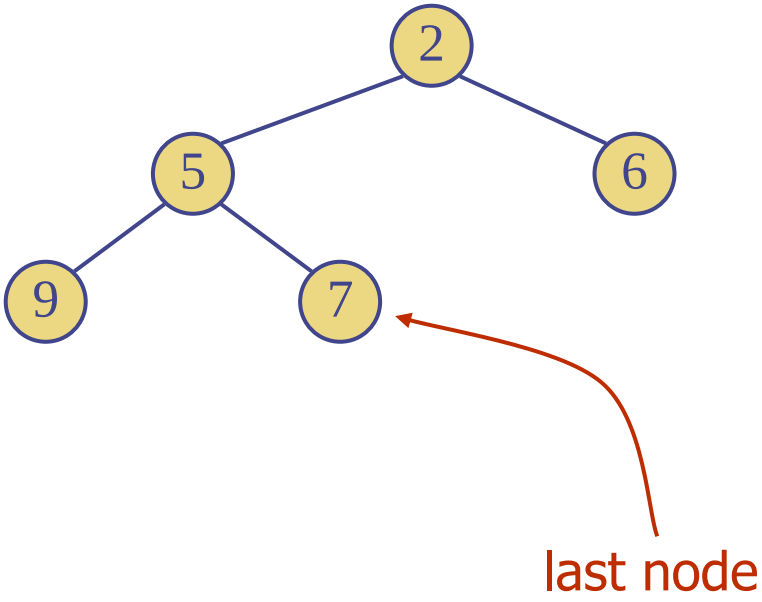
- Can we find an implementation of a PQ such that **all** operations are better than O(n)?
  - I.e. want all operations o(n).
  - E.g. aim for them all to be O(log n)

- We can do this by implementing a Priority Queue using a kind of binary tree called a 'heap':

Heaps

- A heap is a binary tree storing key-value pairs at its nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$, the nodes are to the left of any 'missing nodes'
      - in this case the children of 6 are 'missing'

- The last node of a heap is the rightmost node of depth h

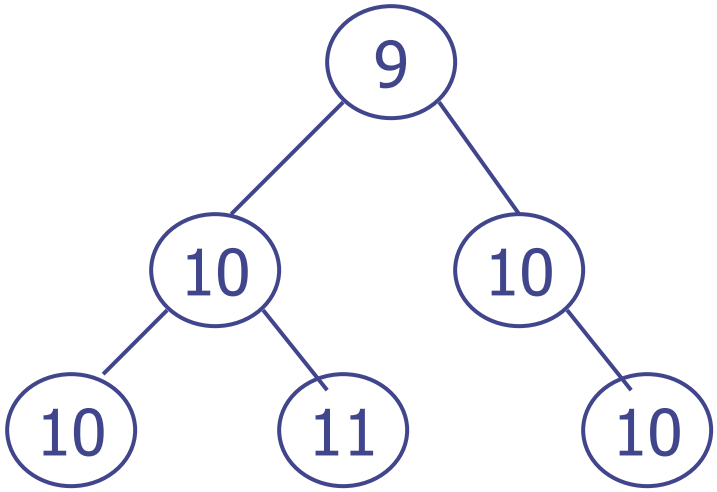


Portions of slides are © 2004 Goodrich, Tamassia

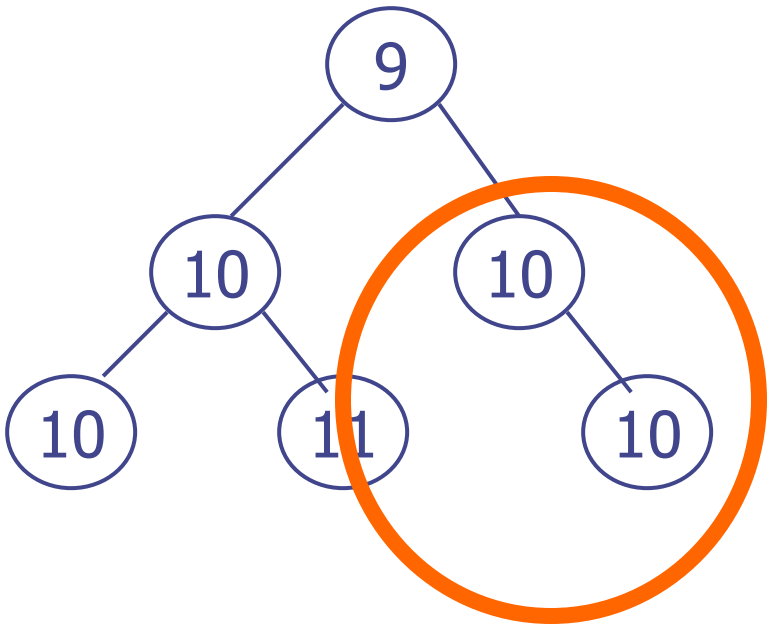
Note on terminology:

- The previous definition of “heap” is a standard one for a “Binary Heap”
  - It is used/assumed in this module.
- Other notions of “heap” do exist
  - they keep the numeric ordering of parent relative to children
  - but can have different conditions for the “shape”
    - E.g. “Binomial” and “Fibonacci” heaps

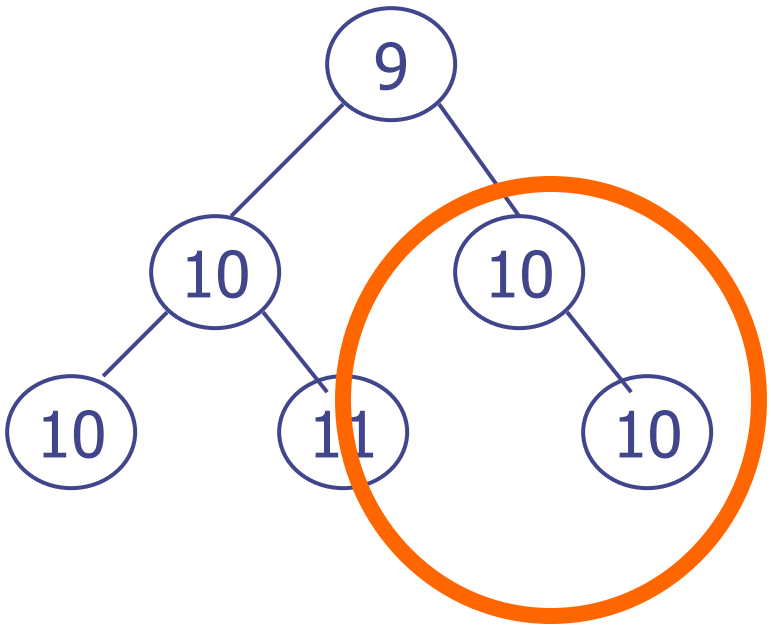
Exercise: is this a heap?



Exercise: is this a heap?



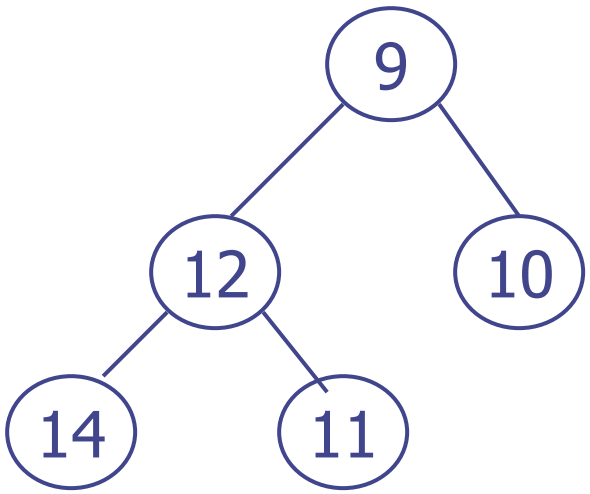
Exercise: is this a heap?



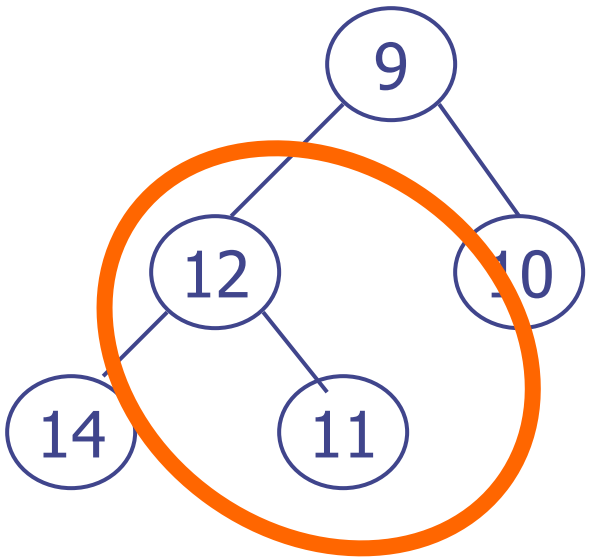
Answer is no, because the shape of the tree is wrong:

- The nodes on the bottom row are not "as leftwards as they can be"
- E.g. If a node has just one child it must be a left child.

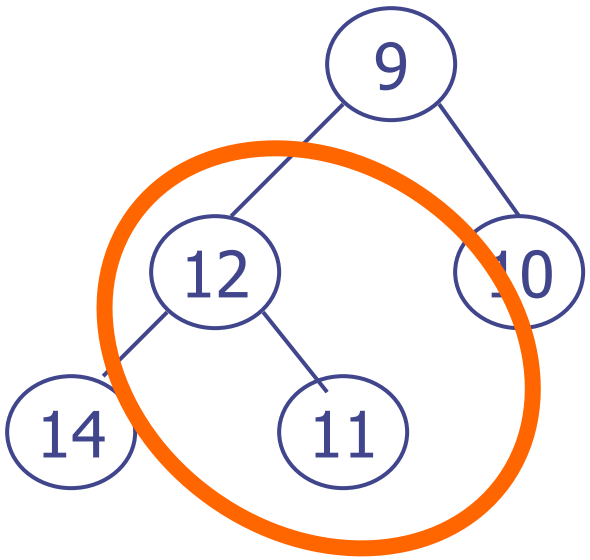
Exercise: is this a heap?



Exercise: is this a heap?



Exercise: is this a heap?



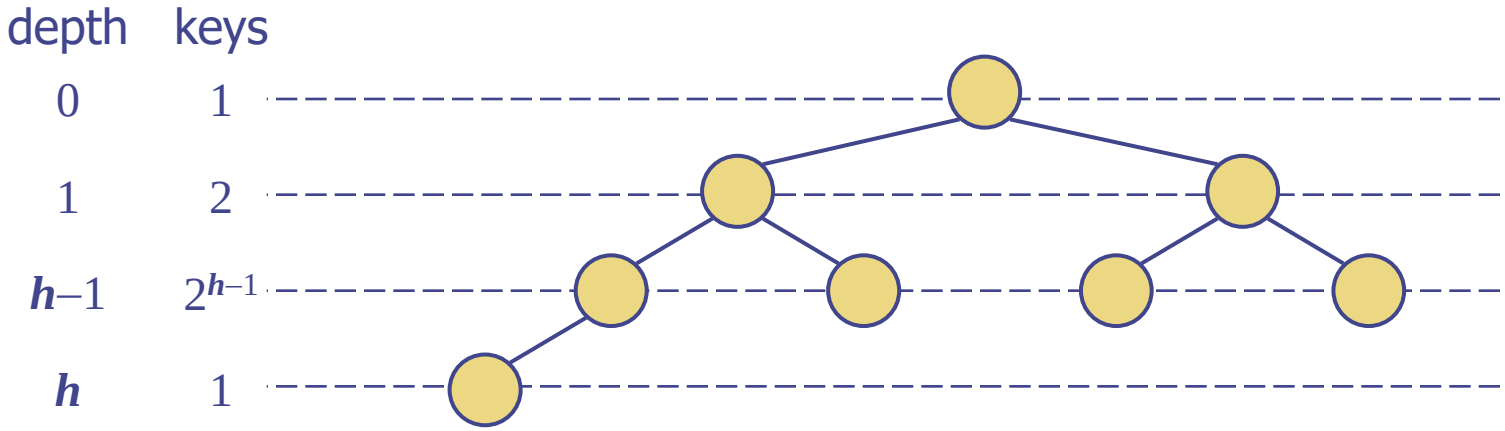
The answer is no:

- Because the numeric conditions are not satisfied
- In a heap that supports removal of a minimum element, then the children of a node can never be smaller.

Height of a Heap

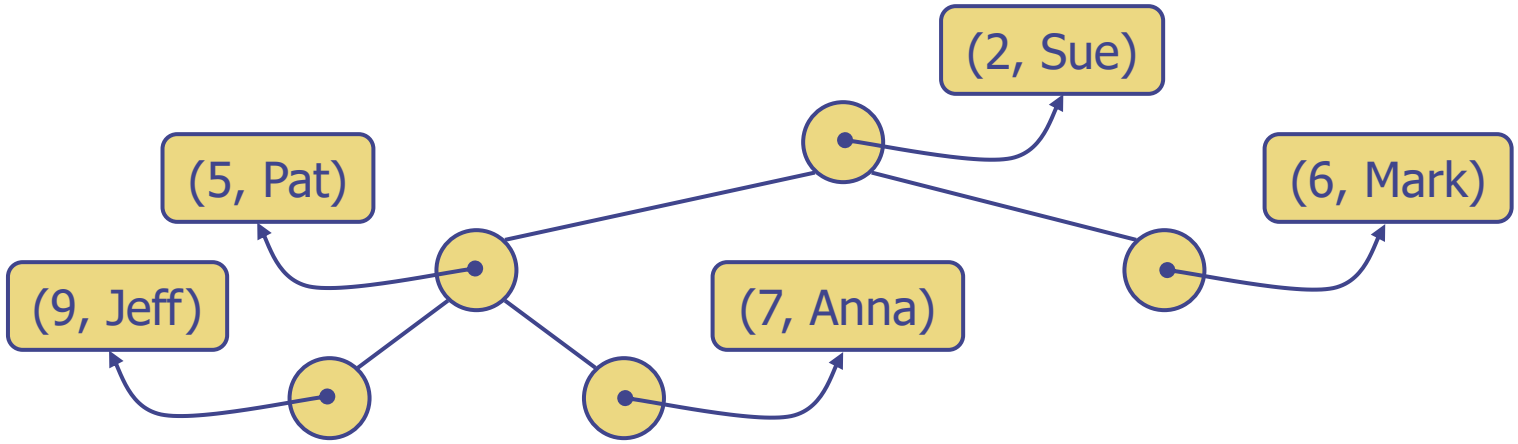
- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: This uses just the complete binary tree property
  - Let h be the height of a heap storing n key
  - The perfect binary tree of height h-1 has $2^h - 1$ nodes
  - Our tree has at least one more, thus, $n \geq 2^h$, i.e., $h \leq \log n$



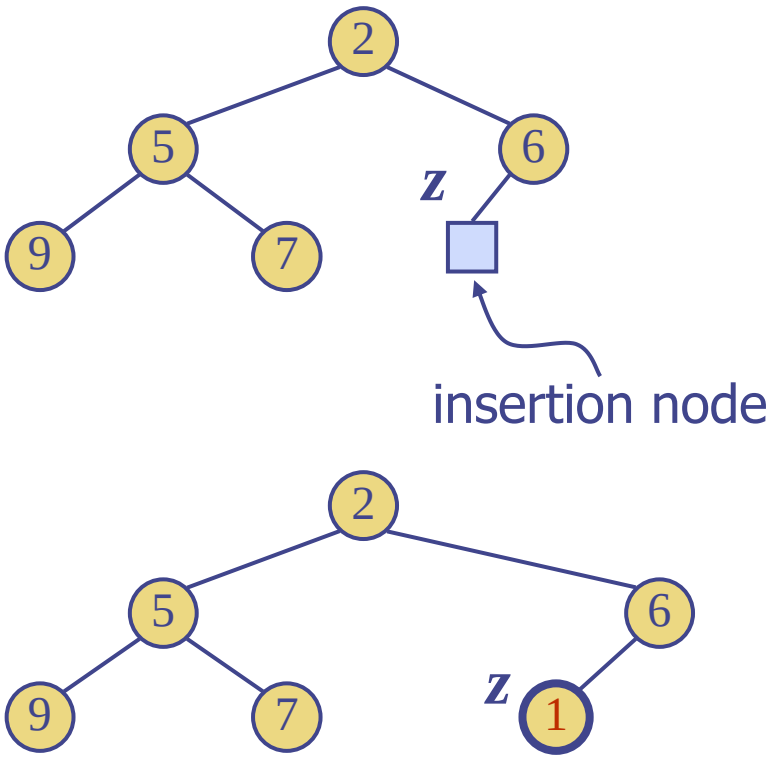
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each node
- We keep track of the position of the last node



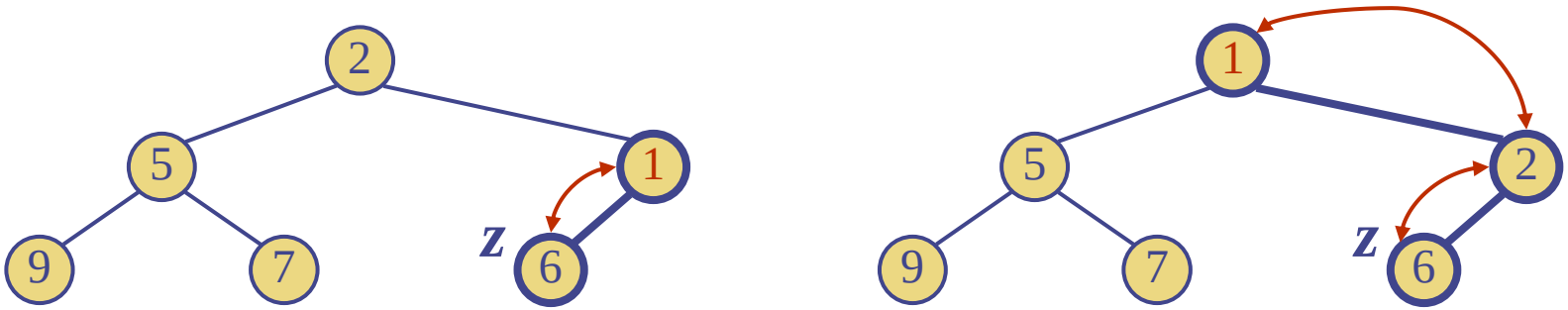
Insertion into a Heap

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



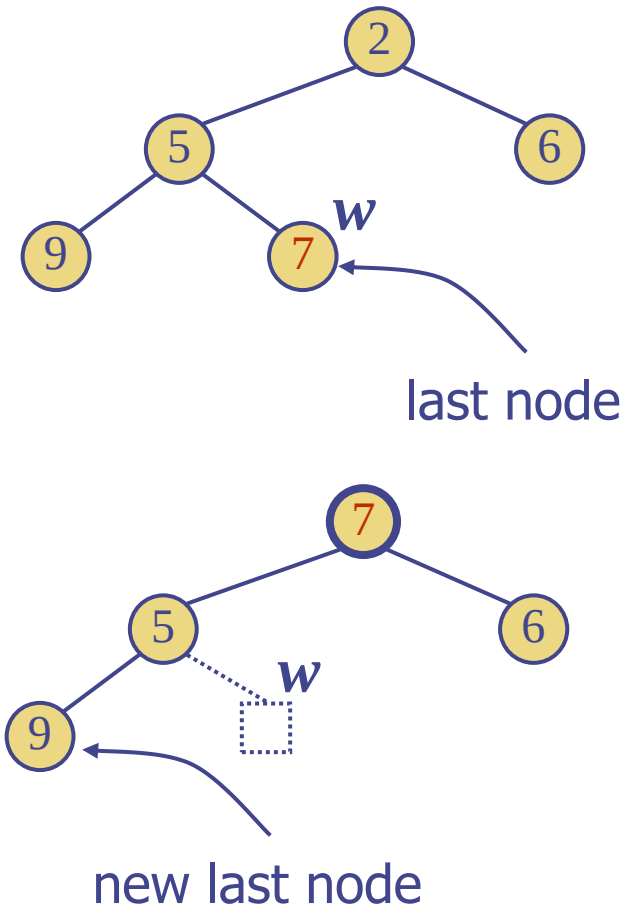
Exercises

- Convince yourself that the upheap method is correct
  - i.e. that it always results in the heap properties being satisfied after it is completed
  - It is typical in data structure algorithms that required properties are temporarily broken, but then restored

- Remark:
  - You should always strive to convince yourself of the correctness for all algorithms given to you
  - If nothing else, it encourages deeper thinking about the algorithm, and this makes it easier to
    - 'fill in the gaps' if forget parts of them ☺
    - Extend the ideas to other cases

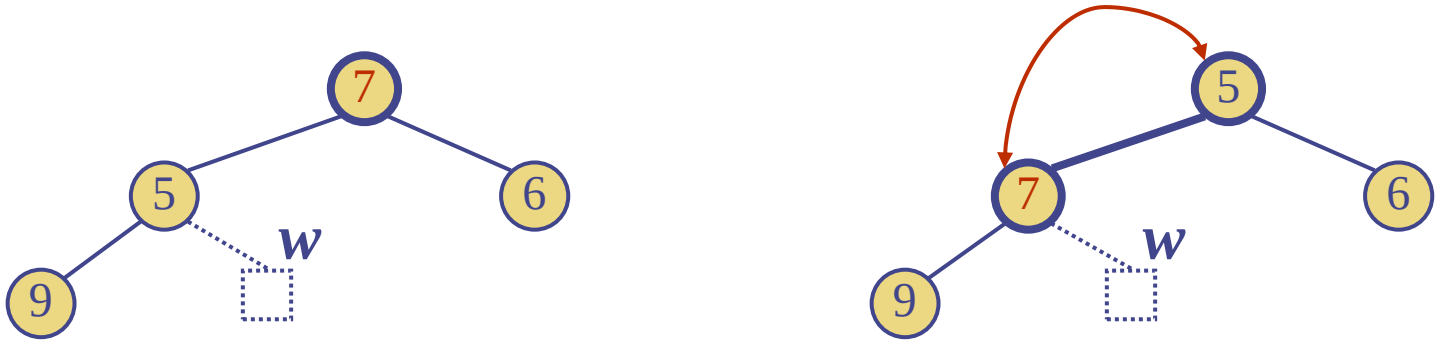
Removal from a Heap

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



Downheap

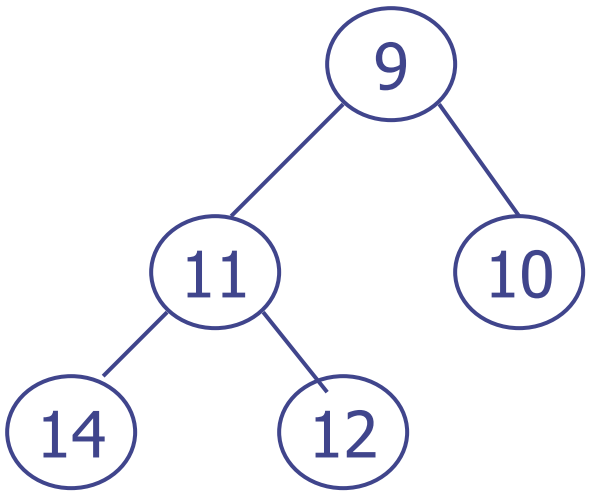
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a particular downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



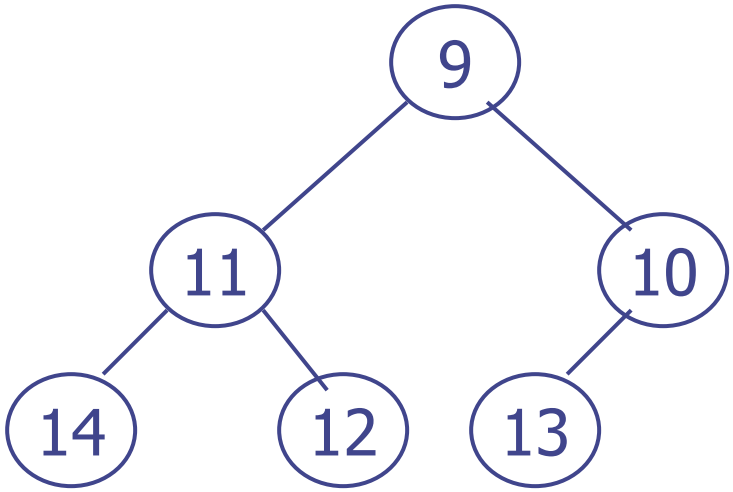
Exercise (offline)

- Be sure that you understand why the upheap and downheap operations are correct.
- In particular, why do we only need to fix nodes on the path from the "adjusted nodes" back up to the root node?
  - I.e. Why do we not need to scan the entire tree?
  - Otherwise, the operations would not be O(h)=O(log(n)) but would be O(n)
    - (that is they would be exponentially slower)

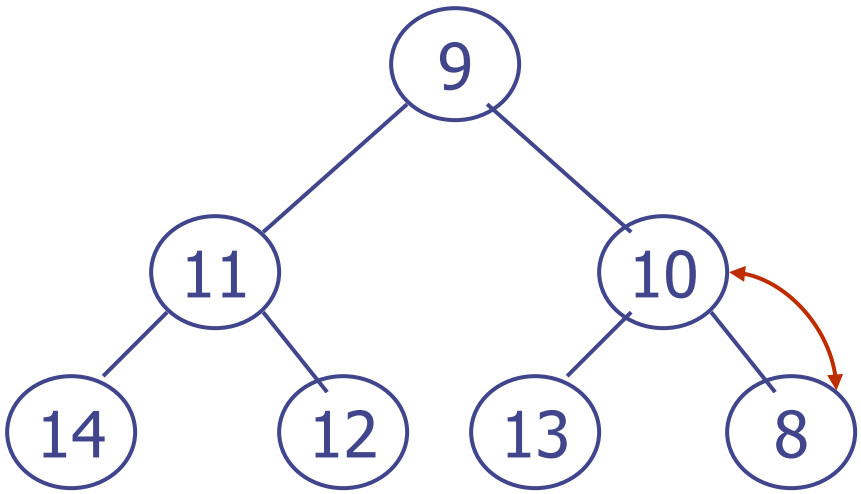
Exercise: insert 13, 8, 7



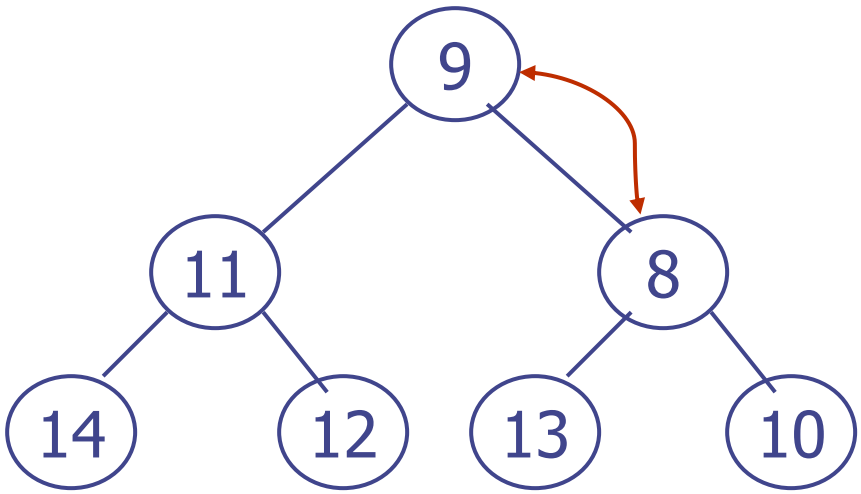
Exercise: insert 13



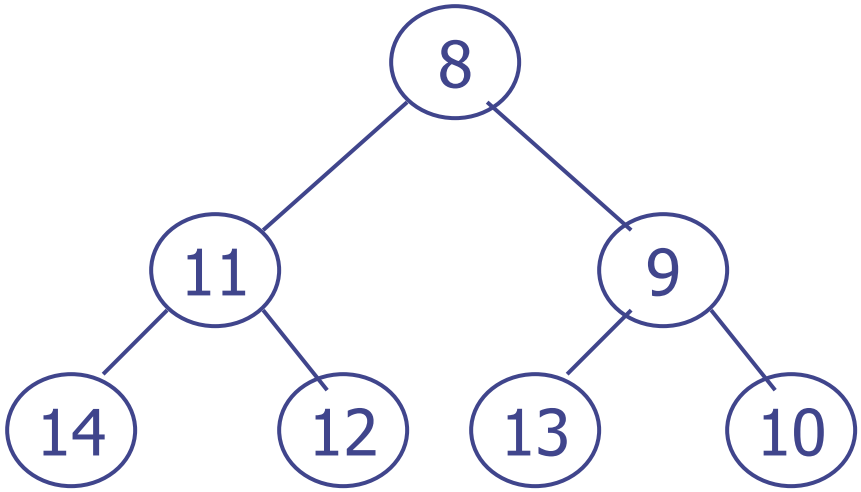
Exercise: insert 8



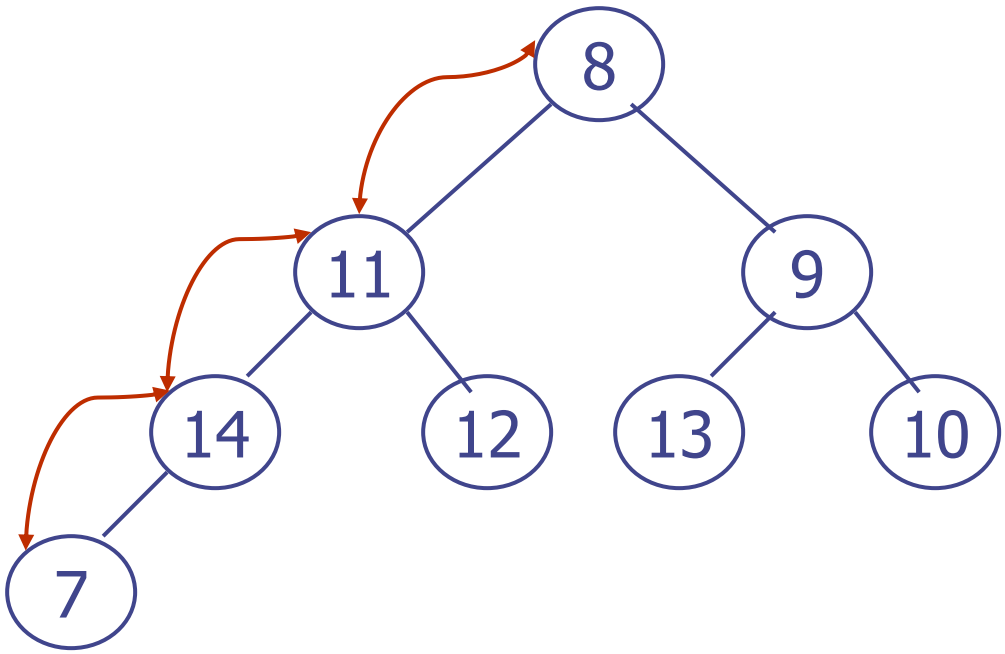
Exercise: insert 8



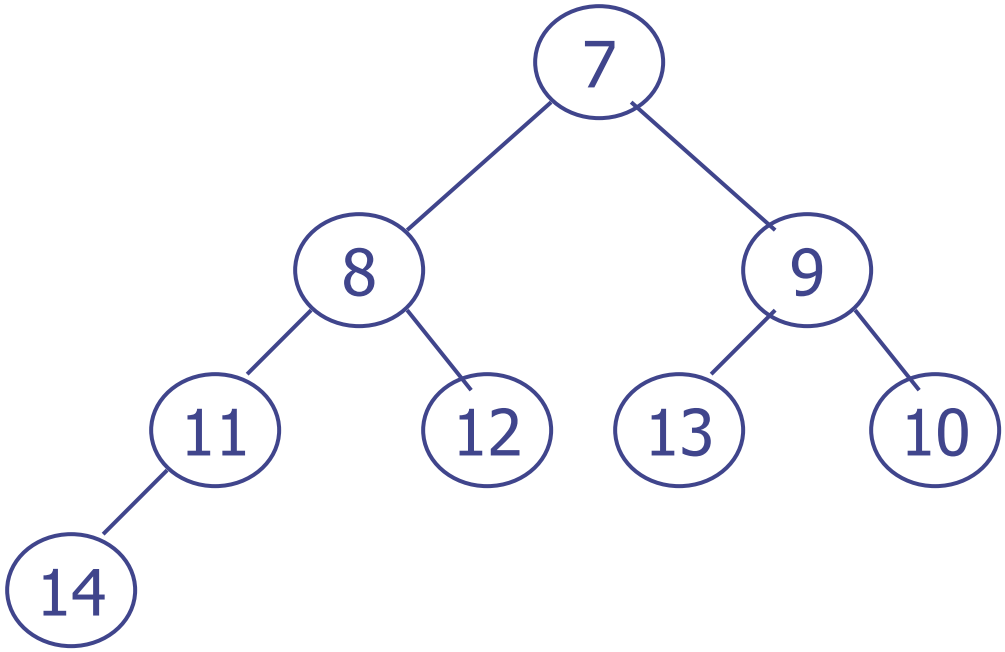
Exercise: insert 8



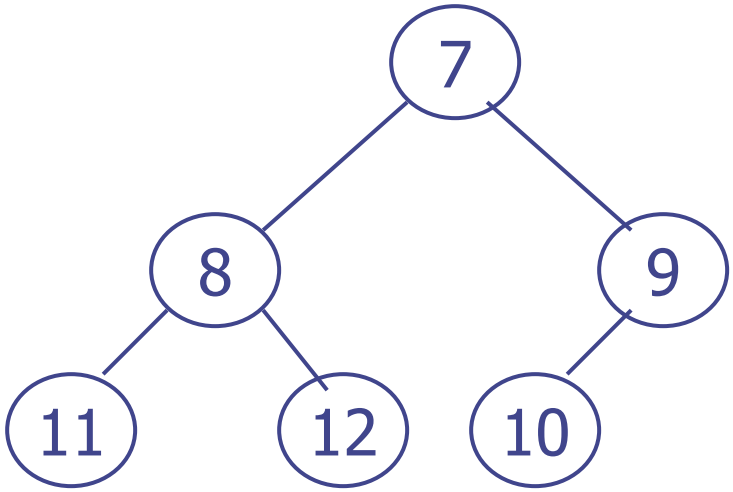
Exercise: insert 7



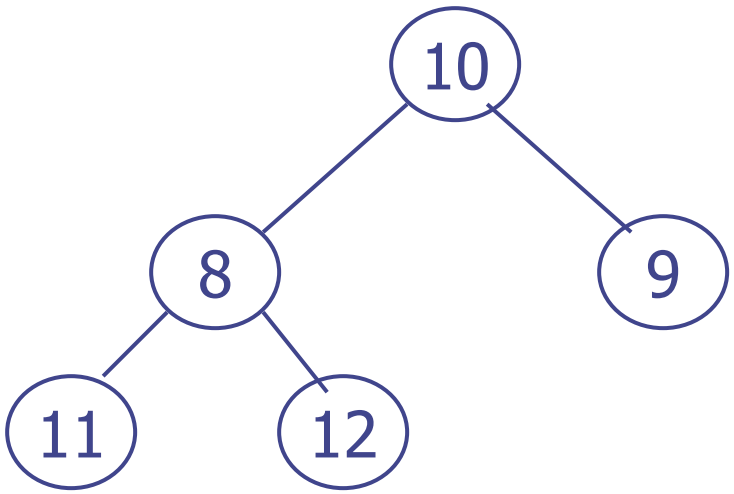
Exercise: insert 7



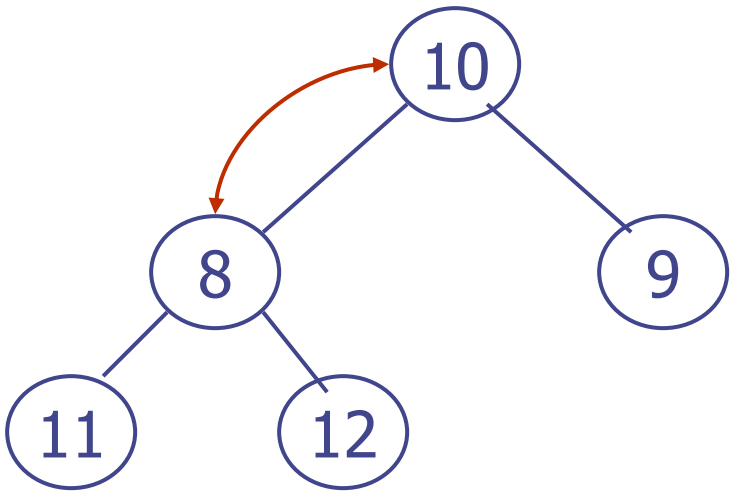
Exercise: remove 7



Exercise: remove 7



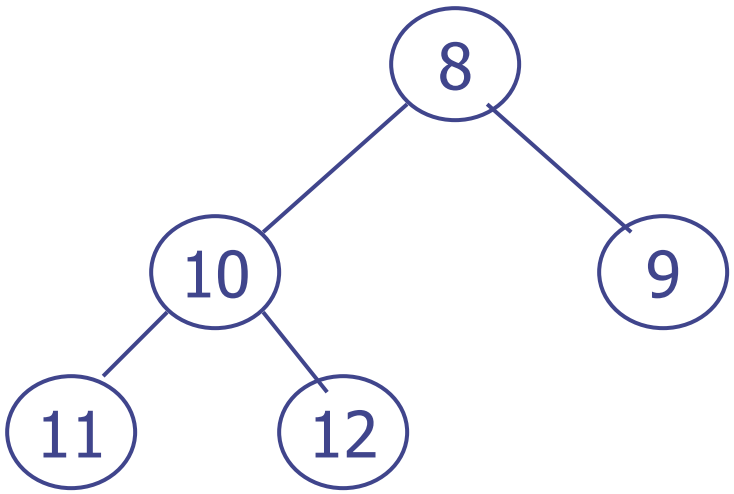
Exercise: remove 7



Always swap with the smallest child!

Exercises (offline):
Why is this needed?
Why is it always correct?
E.g. Try some other rule, such as
"swap with the left child if it is smaller" (BAD RULE)
and find an example where the bad rule fails.

Exercise: remove 7



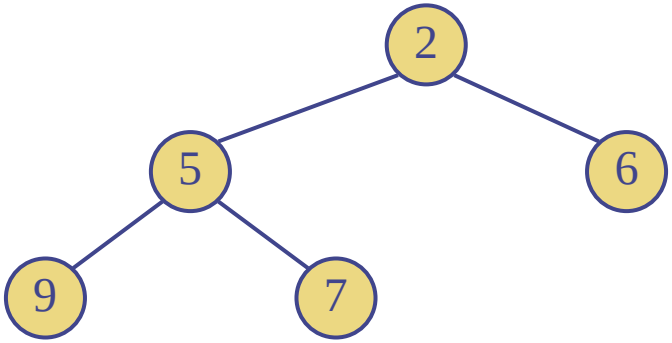
Final result is a valid version of the data structure

Always check this!!

(In code one could have an assertion to check it, and that is activated in DEBUG mode.)

Array Heap Implementation

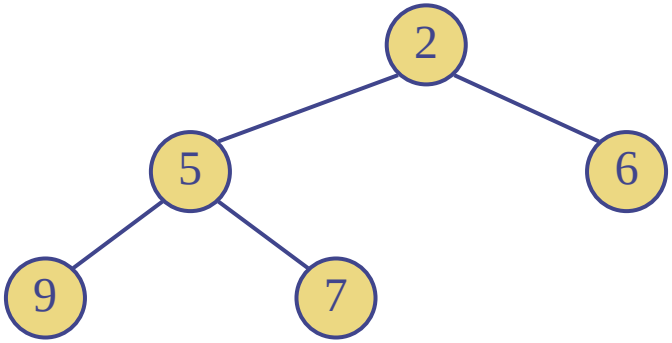
- (See the "trees" lecture on representing binary trees within an array)
- We can represent a heap with n keys by means of an Array (or Vector) of length $n+1$
- Links between nodes are not explicitly stored, instead:
- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i+1$
  - the parent is at index $i/2$
- The cell at index 0 is not used
  - Makes the equations easier/faster
- Notice that there are no "gaps" when storing a heap



	2	5	6	9	7
0	1	2	3	4	5

Array Heap Implementation

- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i+1$
  - the parent is at index $i/2$
- Operation insert corresponds to inserting at index $n+1$
- Operation removeMin corresponds to moving index n to index 1
- Up- and down-heap operations just swap appropriate elements within the array
- Together with the lack of gaps, this makes the implementation very efficient, and this is the standard way to implement a Heap



	2	5	6	9	7
0	1	2	3	4	5

Implementing Priority Queue with a Heap

- To create a priority queue, initialise a heap
- To insert in the priority queue, insert in the heap
- To get the value with the minimal key, ask for the value

  of the root of the heap
- To dequeue the highest priority item, remove the root and return the value stored there.

Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Heap-Sort 2

The algorithm has two stages:

1. Insert all elements into the heap one by one.
   This takes n insertions with O(log n) each for a total of O(n log n)
2. Remove all elements one by one, using removeMin(), hence obtaining them in sorted order.
   This takes n removals with O(log n) each for a total of O(n log n).

Hence, the overall cost is O(n log n)

Conclusion

- Priority Queue ADT can be implemented using an unsorted list, a sorted list, or a heap.
  - In the first two cases, one of the methods requires O(n) time.
  - For the heap implementation, all methods run in O(log n).

IMPORTANT

Do NOT confuse

"Binary Tree implementations of heaps"
(this lecture – used for PQ ADT)

with

"Binary Search Trees"
(other lectures – used for Map ADT)

Minimal Expectations

- Thoroughly know the definitions
  - Be able to reason about them and explain their motivations, not only be able to reproduce them
- Know the Binary tree implementation
  - **Do NOT confuse "Binary Tree implementations of heaps" with "Binary Search Trees" (later lectures)**
- Know
  - The insertion/deletion procedures
  - Array-based implementations

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE

Map ADT : using lists

Abstract Data Type: Maps

- A map models a collection of key-value entries that is searchable `by the key'
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
  - (may be allowed in a 'multi-map')

Common Applications of Maps

- Address book:
  - key=name, value=address
- Student-record database
  - key=student-ID, value=name, marks, etc.
- Dictionaries:
  - key=word, value=‘meaning’
- Symbol table:
  - key=symbol, value=‘something internally useful’
  - standard component of compilers, and any program that has to read data from a file

Also known as “associative arrays” (perl, php) because they act like an array in which the index need not be an integer. Maps are very useful in many coding tasks

Portions of slides are © 2004 Goodrich, Tamassia

The Map ADT over pairs <K,V>

Map ADT methods:

- V get(K k): if the map M has an entry with key k, return its associated value; else, return null
- V put(K k, V v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- V remove(K k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- int size(), boolean isEmpty()
- {K} keys(): return an iterable collection of the keys in M
- {V} values(): return an iterable collection of the values in M
- {<K,V>} entries(): return an iterable collection of the entries in M

Portions of slides are © 2004 Goodrich, Tamassia

VERY IMPORTANT

Be careful not to confuse different usages of binary trees:

- **Priority Queue &**
  "Binary Tree implementations of heaps" (in other lectures)

with

- **MAP &**
  "Binary Search Trees" (in later lecture)

Both insert entries based on a key.
The crucial difference is the way entries are accessed:

- MAP: access **any** key
- PQ: access only the **minimum** key

A Simple List-Based Map

- It is straightforward to implement a map using a list – either singly or doubly-linked.
- We also store separate size counter, n, so that "getSize()" is O(1), and maintain it on adding or removing elements

The get(k) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value
3. Else return null (or false)

Worst case: O(n)

The put(k,v) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value, and overwrite with the new value v
3. Else insert <k,v> as a node at the end of the list

Note: we cannot just put the new entry at the start of the list, because we need to check that the key k is not already stored !

Worst case: O(n)

The remove(k) Algorithm

Simply

1. Walk along the list looking for key k
2. If find it then return the currently associated value and remove the node.
3. Else return null/false

Worst case: O(n)

List-Based Map

- Overall: the operations are O(n) because of needing to traverse the list.

- The unsorted list implementation is suitable only for maps of small size

- We could use a sorted list, but still O(n)
  - It does not help as we need to be able to access any k, not just the minimum

Remark: Prototyping ADTs

- List version of a Map is
  "simple but inefficient"
- Quite common that ADTs can be implemented very easily using lists & arrays.
  - Easier to be correct
  - Even though slow, they can be useful
    - allow the rest of the project to proceed without waiting for the efficient version
    - can be used for a regression test of a "faster but trickier" later version
    - e.g. use in a "shadow mode" where the "slow-obviously-correct" version is used together with the "fast-possibly-buggy" version and the results checked against each other.

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE

Map ADT and hashtables

The Map ADT over <K,V>

Map ADT methods:

- V get(K k): if the map M has an entry with key k, return its associated value; else, return null
- V put(K k, V v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- V remove(K k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- int size(), boolean isEmpty()
- {K} keys(): return an iterable collection of the keys in M
- {V} values(): return an iterable collection of the values in M
- {<K,V>} entries(): return an iterable collection of the entries in M

Portions of slides are © 2004 Goodrich, Tamassia

Hash Tables

- Hash tables are a concrete data structure which is suitable for implementing maps.

- **Basic idea: convert each key into an index into a (big) array.**

- Look-up of keys and insertion and deletion in a hash table usually runs in O(1) time.
  - Not guaranteed, and design of the table needs to be done carefully if want the access to be "reliably O(1)"

Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
  - Example:
    $h(k) = k \bmod N$
    is a hash function for integer keys
- The integer $h(k)$ is called the hash value of key k

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, v) at index $i = h(k)$

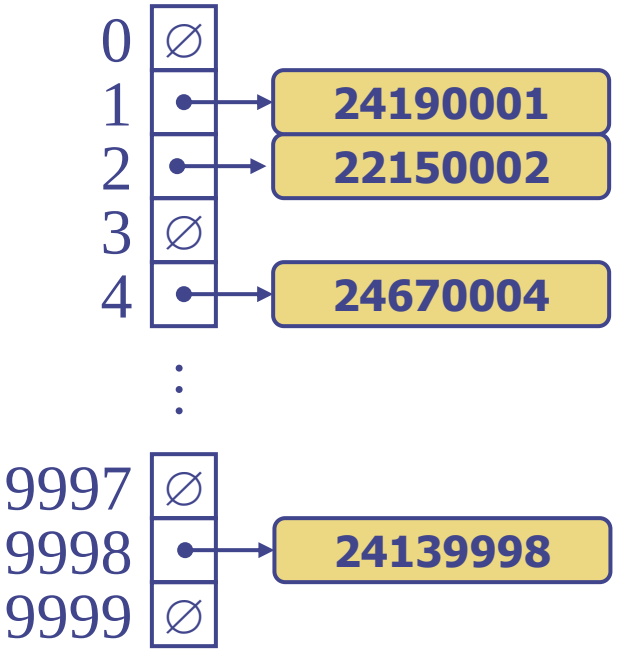
Example

- We design a hash table for a map storing entries as (SID, Name), where SID (student identity number) is an eight-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function

  $h(x) = \text{last four digits of } x$

  $= \text{“}x \bmod 10000\text{”}$

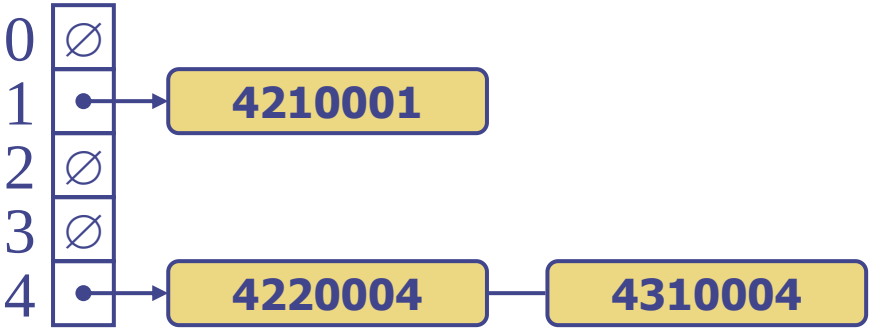
  (details depends if SID is stored as an int or a string)



Portions of slides are © 2004 Goodrich, Tamassia

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- A lot of the theory and practice of hashing consists of devising better ways to avoid or handle collisions



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  $\boldsymbol{h}_1$: keys $\rightarrow$ integers

  Compression function:

  $\boldsymbol{h}_2$: integers $\rightarrow [0, \boldsymbol{N} - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

- The goal of the hash function is to "disperse" the keys in an "apparently random" way

Hash functions: “dispersal”?

- Re: The goal of the hash function is to “disperse” the keys in an “apparently random” way
- Questions:
  - Why disperse?
  - Why random?

Hash functions: “dispersal”?

- Why disperse?
  - to reduce numbers of collisions
- Why random?
  - random means ‘no pattern’
  - if there is an obvious pattern then the incoming data might have a matching pattern that leads to many collisions
  - “sometimes ‘no pattern’ is the only safe pattern” (e.g. rock-paper-scissors game)

Hash Codes [Not assessed]

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.) [Not assessed]

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots \ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Or, representing a string as a number on base z
- Compare with base 10:

$365 = 3*10^2 + 6*10^1 + 5*10^0$

- Base 27 (26 characters + blank):

$cab = 3*27^2 + 1*27^1 + 2*27^0$

where

a = 1, b = 2, c = 3 and z = 26

Portions of slides are © 2004 Goodrich, Tamassia

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime

  (hash codes will tend to spread better)

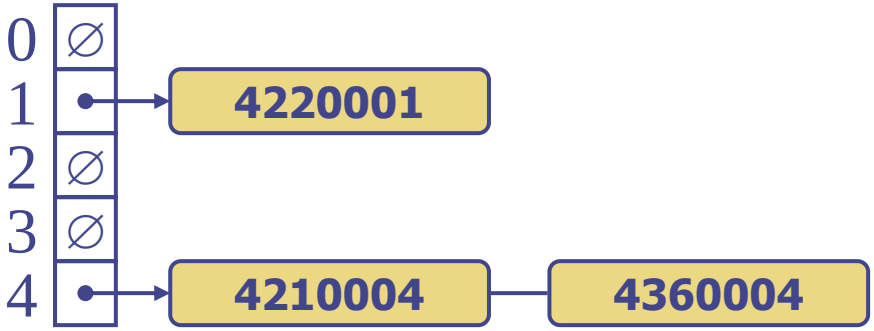
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers
  - such that

    $a \bmod N \neq 0$

    - Otherwise, every integer would map to the same value b

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to (e.g.) a linked list of entries that map there
  - Note: In practice, should use a more efficient Map; e.g. a Binary Search Tree (BST), (see later lectures)



Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):

Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map

return $A[h(k)]$.get(k)

// Simply delegates the "get" to the list-based map at $A[h(k)]$

Map Methods with Separate Chaining used for Collisions

Algorithm put(k,v):

Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**

$t \leftarrow A[h(k)]$.put(k,v)

// Simply delegates the put to the list-based map at $A[h(k)]$

if t = **null then** {k is a new key}

$n \leftarrow n + 1$

return t

Map Methods with Separate Chaining used for Collisions

Algorithm remove(k):

Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map

$t \leftarrow A[h(k)]$.remove(k)

// Simply delegates the remove to the list-based map at $A[h(k)]$

if $t \neq$ **null then** {k was found}

$n \leftarrow n - 1$

return t

Access is still O(n), but usually the relevant n is much smaller, because it usually builds many small lists, e.g. length n/N on average.
So this method might be okay, sometimes.

Separate Chaining

- Separate chaining is simple and fast, but requires additional memory outside the table.
- When memory is critical then we try harder to remain within the existing memory:

Open addressing

- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell (variant: cell + c where c is a constant)
  - "Circular array" – once get to the right-hand end, then just start again at the beginning of the array
- Each table cell inspected is referred to as a "probe"
- Disadvantage: Colliding items lump together, causing future collisions to cause a longer sequence of probes

Open addressing : Example

- Example: $h(x) = x \bmod 13$, c=1
  - Insert keys 18, 41, 22, 44, 59, 32, 5 in this order
  - 18,41,22 have no collisions, giving

		41			18				22			

- But 44mod13=5 collides with 18mod13=5, so use index 5+1

		41			18	44			22			

- Now 59mod13=7 has no collisions

		41			18	44	59		22			

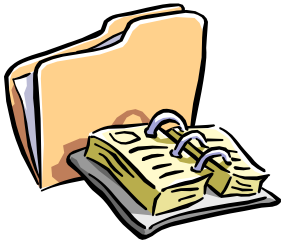
- Now 32mod13=6 collides with the 44, and so walk past the 44 and 59 to find an empty space

		41			18	44	59	32	22			

- Now 5mod13=5 has a long walk to find an empty space:

		41			18	44	59	32	22	5		

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return null
```

Open addressing : Example

- Example: $h(x) = x \bmod 13$, c=1
  - Insert keys 18, 41, 22, 44, 59, 32, 5 in this order
  - Gave

		41			**18**	**44**	**59**	**32**	**22**	**5**		

- Suppose we now remove(22).
  It seems we should just simply obtain:

		41			**18**	**44**	**59**	**32**		**5**		

  - Do you see a problem?
  - Suppose we try to do get(5)
  - get(5) would fail to find the 5
  - as the scan stops at the empty cell where 22 used to be

Exercise

- How do you safely remove an element x?
- One answer:
  - (not the best answer, but simplest):
  - Find x using `get' and set the entry back to blank, i.e. null or empty (which sometimes write as '#')
  - Fix the sequence on its right-hand-side
    - WHY!?: If any entry on the right used linear probing then it might no longer be discoverable by 'get' because it will stop at the blank!!!
    - Fix: Move such entries, e.g. by removing them and then re-inserting them all.
  - **EXERCISE (offline)** Figure out the details and write pseudo-code for this and do examples. (Ask in labs/tutorials if needed!)

Exercise

How do you safely remove an element x? Another solution:

- "**Lazy deletion**": don't mark the entry as a blank, but as a 'deleted' and fix the entries later. E.g. see http://opendatastructures.org/versions/edition-0.1e/ods-java/5_2_LinearHashTable_Linear_.html

- E.g. in the find, skip over a 'deleted' entry rather than stopping

(See Tutorials)

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

  $$(h(k) + j\, d(k)) \bmod N$$

  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- Linear probing is just $d(k)=1$
- The table size N must be a prime to allow probing of all the cells

- Common choice for the secondary hash function:

  $$d(k) = q - (k \bmod q)$$

  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

  $$1, 2, \ldots, q$$

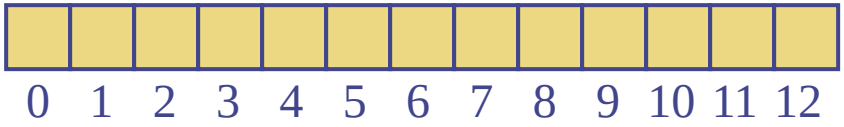
Remarks

- "The table size ***N*** must be a prime to allow probing of all the cells"
- E.g. consider d(k) = 4
  - With N=12 then the only positions scanned are 4,8,0,4, …
    - So we miss many cells that might have space
  - With N=11 then the positions are 4,8,1,5,9,2, …
- With a prime N, then eventually all table positions will be probed

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

- Exercise: do the details yourself, and see tutorials

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- In Java, maximal load factor is 0.75 (75%) – after that, rehashed
  - as for Vector, it may be good to "roughly double" the table size each rehash
    - pick a new (prime) close to twice the current size
- The expected running time of all the map ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%

Re-Hashing

When the table gets too full then "re-hash":

Create a new larger table and new hash function.

- Need to (eventually) transfer all the entries from the old table to the new one
- If do so immediately, then
  - one can amortise the cost over many entries (as for Vector) and so get an average cost of O(1) again
  - but the worst case might be O(n) when the table is rehashed, and this might be bad for a real time system
  - Option:
    do not transfer all entries "in one go" but do "a few at a time"
  - Keep both tables until the transfer is complete; but only do insertions into the new table.

Exercise (offline): consider this in more detail, and read the relevant part of the text book (Section 9.2.7 Load factors and Reshashing) or the wiki page http://en.wikipedia.org/wiki/Hash_table#Dynamic_resizing

Applications of Hashing

- Direct applications of hash tables:
  - small databases
  - compilers
  - browser caches – the weird and wonderful filenames in the browser cache folder are hashcodes of something?

- Hash tables as an auxiliary data structure in a program:
  - Look-up table: if you want to check whether some object has been seen before, for example in a graph or list traversal, keep a hashtable of (object,"seen before") pairs, where the key is the reference to the object, and the value is some arbitrary marker.

Comparison of HashMap and PQ

- HashMap does not use the ordering of keys
  - E.g. does not implement min()
  - In a hash table it would need a scan of all the keys in the table, so O(n) (or worse)
- PQ does not allow direct access to a key
  - E.g. there is no easy way to do get(k)
  - In a (standard) heap we would have to walk through all the entries

Minimum Expectations

- Map ADT, and its usage
- basic concepts of hash tables
- hash codes and compression functions
- Options to handle collisions
  - Separate chaining, linear probing, double hashing
  - How to insert, find/get, remove for all these systems

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054 ADE
Binary Search Trees

Note to help with understanding: "Global vs. Local Perspectives"

(intended to help with coding/understanding algorithms)

- Global view of tree (or any other data structure)
  - can look at entire tree in one go
  - human seeing a picture of a (small) tree
- Local view of tree
  - what your code sees: "code perspective"
  - code generally only sees a local portion of a tree and must work with that
- Vital coding skill: develop ability to see the local view, "code perspective", and not only the global
  - at each code line need to know which data is immediately accessible
  - Avoid trying to code in a way that cannot be done locally
  - Note the local view still needs to work towards a global goal

Portions of slides are © 2004 Goodrich, Tamassia

Note intended to help with your coding: "Global vs. Local Perspectives" Analogies

- Your Data Structure:
  - A real tree
- Your code:
  - "A short-sighted ant crawling over the tree"

Note intended to help with your coding: "Global vs. Local Perspectives" Analogies

- Your Data Structure:
  - A big underground cave system
- Your code:
  - You!
  - … with a weak flashlight
  - … and no overall map
  - … but a goal of which cavern to reach

Motivations

- Suppose you have an array A (of ints) and want to search for a particular int k
- If the array is unsorted then no choice but to scan all elements, hence O(n)
- If it is sorted then we can do a lot better. How?

Motivations

- Binary Search for an element within a sorted array:

```
Algorithm BinarySearch(int[] A, int k, int L, int R)
// searches for k in the range A[L] … A[R] of the sorted array A
// (note the standard technique of working with a sub-array)
    if R < L
        return false
    m = floor( (L + R) / 2  ) // an approximate midpoint
    if k == A[m]
        return true
    else if k < A[m]
        return BinarySearch(A, k, L, m-1)
    else  //  k > A[m]
        return BinarySearch(A, k, m+1, R)
```


Portions of slides are © 2004 Goodrich, Tamassia

Motivations

- Binary Search for an element within a sorted array is fast
  - The array to be searched is halved at each iteration
  - Hence O(log n) (Next slide proves this).
- It only works because of the step of
  "knowing whether to go left or right"
- But arrays suffer from being slow, O(n), to insert new elements
  - because need to shift O(n) elements to make room for them

- **Search trees attempt to fix the inefficiency of insertion whilst keeping good O(log n) properties of binary search**
  - **So, we need a tree where we "know which way to go":**

Binary search complexity

- Using recurrence relations
  - $T(n) = T(n/2) + 1$
    - "n/2" because only have to search ½ the array after recursion
    - "1" as the cost of doing the comparison and deciding whether to go left or right
  - Can do exactly (exercise!)

- Use Master Theorem (revise lectures if needed):
  - $a=1$, $b=2$, $\log_b(a) = \log_2(1) = 0$
  - So $c = \log_b(a) = 0$
  - Have $f(n) = 1$ which is $\Theta(n^0)$
  - So is Case 2: and then is n^0 but "get an extra log"
  - Hence, $\Theta(\log(n))$

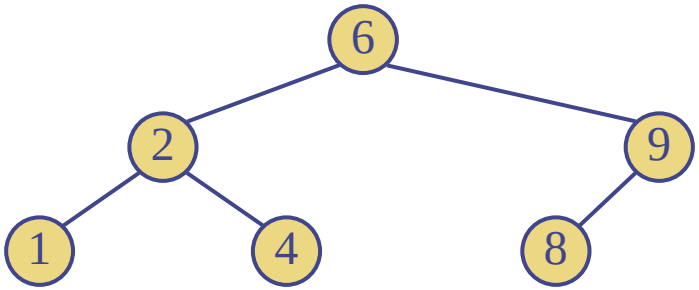
** Binary Search Trees **

- A binary search tree is a binary tree storing key-value entries at its internal nodes and satisfying the following "search tree" property:
  - Let u, v, and w be any three nodes such that u is in the **left subtree** of v and w is in the **right subtree** of v. Then we must have

    $key(u) \leq key(v) \leq key(w)$

    or, as we will assume there are no duplicate keys:

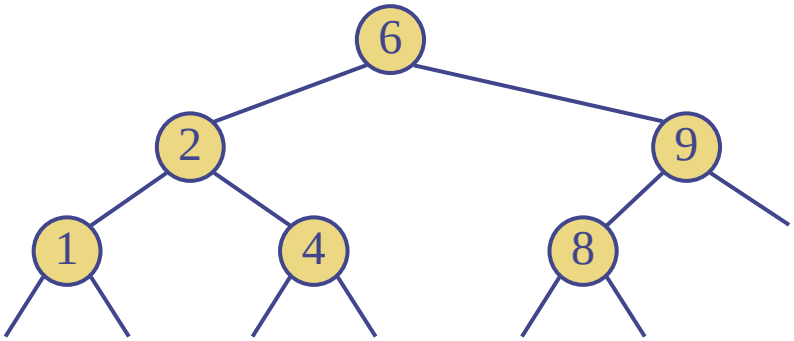
    $key(u) < key(v) < key(w)$



Portions of slides are © 2004 Goodrich, Tamassia

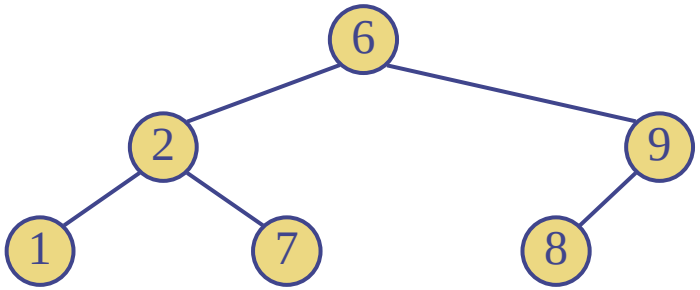
Binary Search Trees

- “”Null” children
  - Sometimes, just for for clarity, we explicitly show the “null pointers” that are for the case of a non-existent child



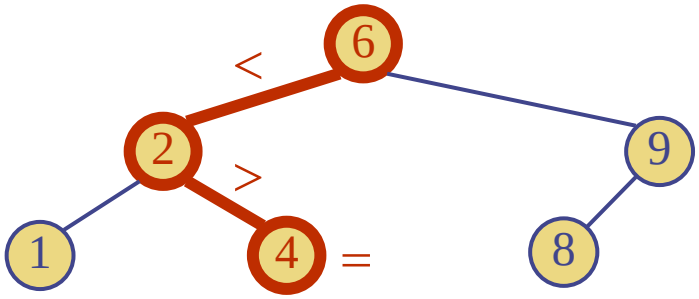
Not a Binary Search Tree

- VERY IMPORTANT:
- The following is **not** a Binary Search Tree
  - Even though the immediate children of every node look correct, we need that **every** node in the left subtree of 6 is less than 6, and this is failed by the node '7'



Search

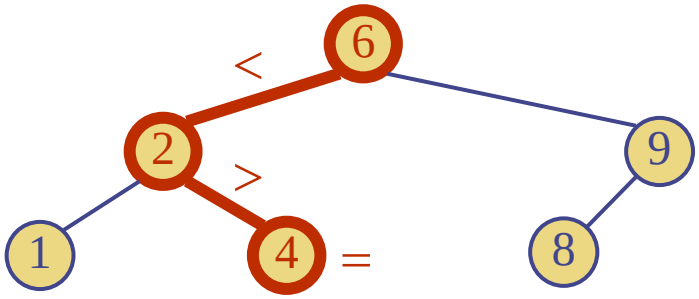
- To search for key k, trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return null



Search

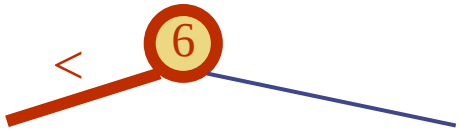
```
Algorithm Node TreeSearch(Key k, Node n)
    if n.isExternal () // or, "if n == null"
        return null
    if k < n.key()
        return TreeSearch(k, n.left())
    else if k = n.key()
        return n
    else // k > n.key()
        return TreeSearch(k, n.right())
```

- Example: find(4):
  - Call TreeSearch(4,root)
  - Note: standard trick of using an auxiliary function that keeps more of the "state"
- Compare this with the binary search of an array



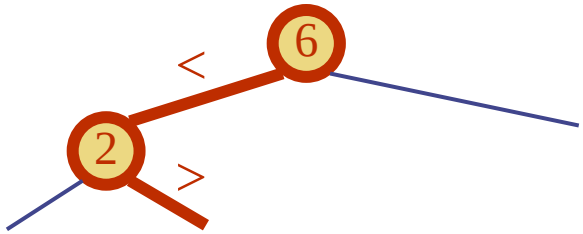
Search from "code perspective"

- Example: find(4):
- 4 < 6 so need to go left



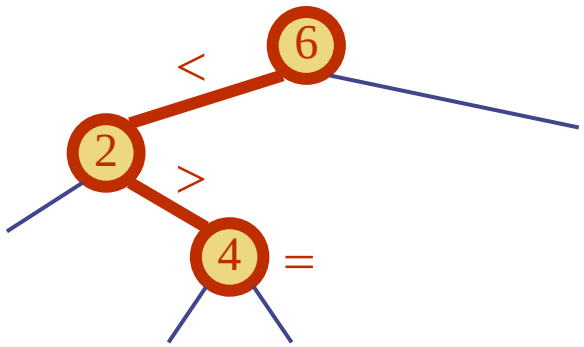
Search from "code perspective"

- Example: find(4):
- 4 > 2 so need to go right



Search from "code perspective"

- Example: find(4):
- Found 4, but note that we only saw/visited a small part of the entire tree



- It only follows a path from the root to some node, and so is clearly O(h) where h is the height of the tree.

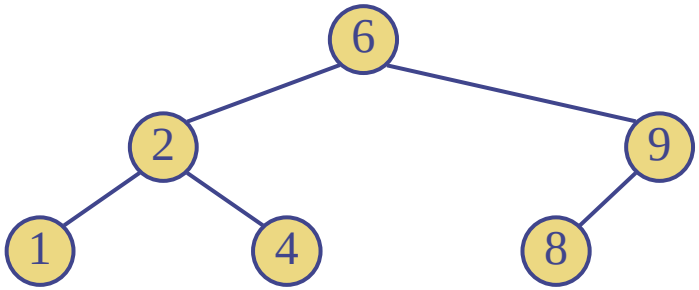
Recursive vs. Iterative

- Generally: Recursive programs are easier to implement, but less efficient
  - because of the overhead of a function call
- For best efficiency, will need to convert to an iterative program

  "while (test) { … }"
- Exercises (offline): convert to iterative
  - binary search of an array
  - search in a binary search tree

Fundamental Property of Search Tree

- Exercise: what does an inorder traversal of the following search tree produce?



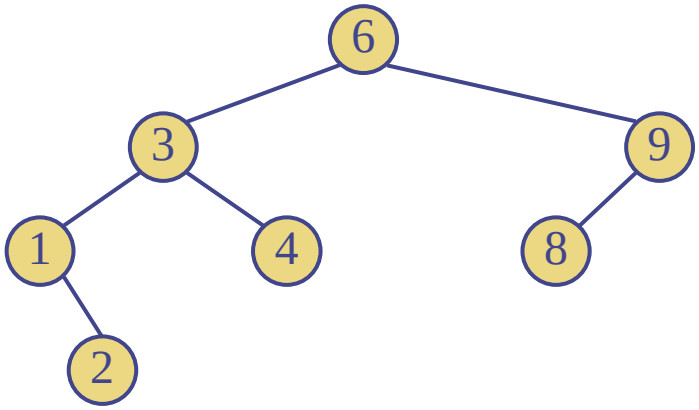
Fundamental Property of Search Tree

Directly from the definition:

- An inorder traversal of a (binary) search trees visits the keys in increasing order

Fundamental Property of Search Tree

- Note that to access the minimum key, we just need to 'always go left'



Insertion

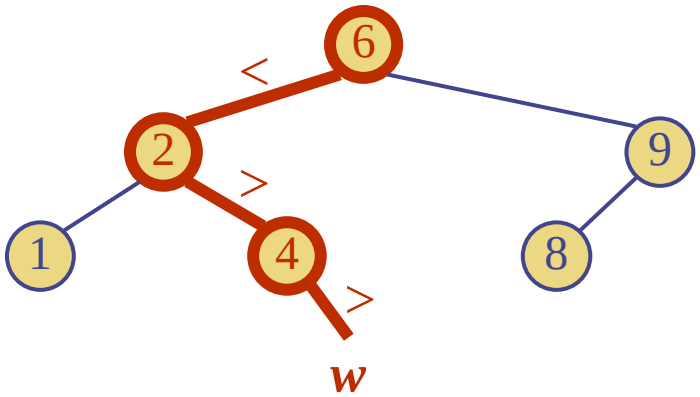
- Exercise: how to do insertion, insert(k, v) ?

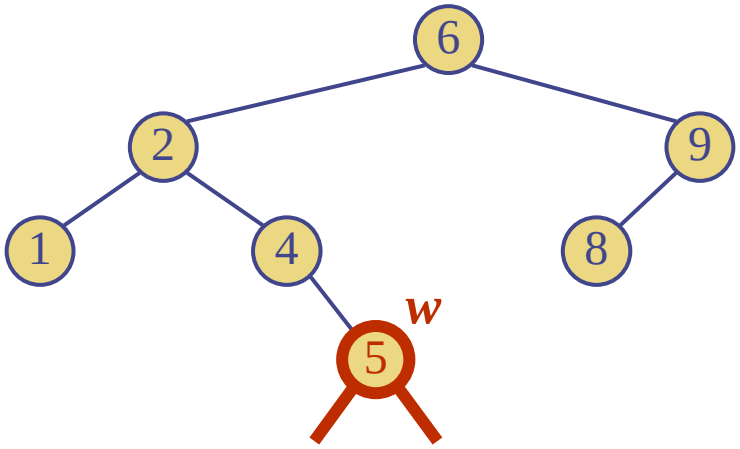
- Have to insert k where a get(k) would find it!
- So natural that insert(k,v) starts with get(k)

- We search for key k (using TreeSearch)
- If k is already in the tree then just replace the value
- Otherwise, k is not already in the tree, and let w be the leaf reached by the search
  - We "insert k at node w and expand w into an internal node"
- Again, only follows a path from the root and so is O(h)

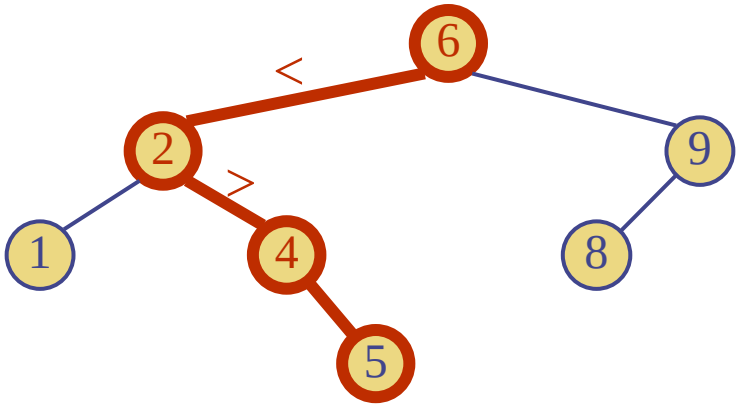
Insertion

- Example: insert 5
- We search for key k=5 (using TreeSearch)
- Let w be "missing child position" reached by the search
- We insert a real node, with key k, at that location w
- Exercise: check that this is correct! Does it preserve the search tree property?





Deletion

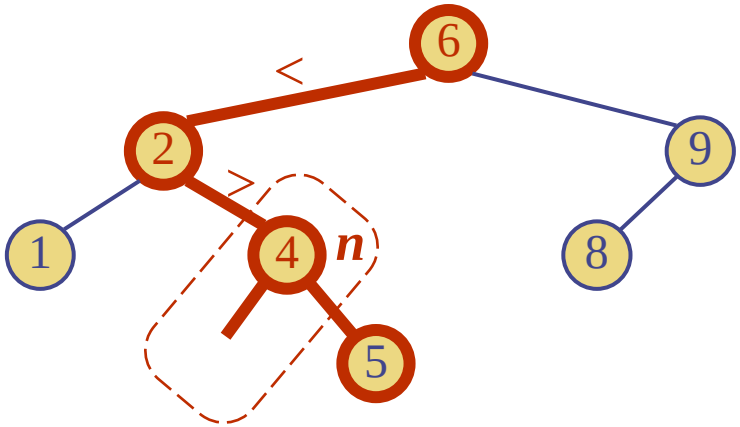


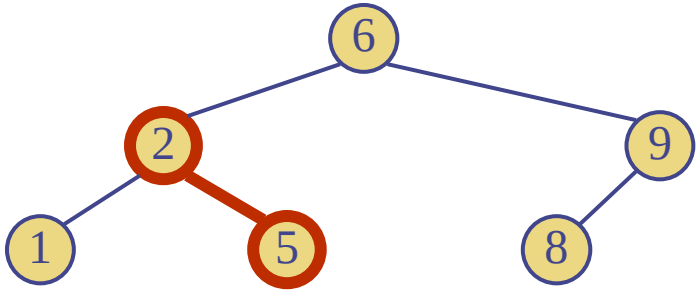
- How can we perform operation remove(k) ?
- As usual we start by trying to find(k)
- Four cases: (think of the externals as null – not as real children)

1. k is not present, e.g. remove(7)
   nothing to do
   otherwise k is stored in some node n
2. n has no children, e.g. remove(5),
   (straightforward, left as an exercise)
3. n has one child, e.g. remove(4)
4. n has two children, e.g. remove(2)

Deletion – with one child

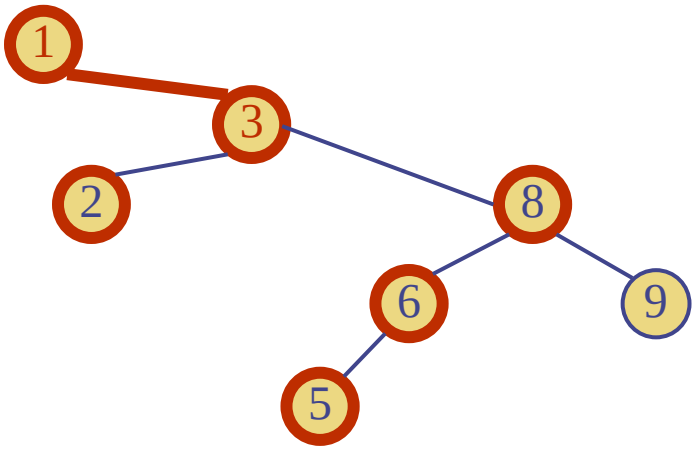
- Example: remove 4
- To perform operation remove(*4*), we search for key ***4.***
- Let ***n*** be the node storing ***4.***
- Node ***n*** has a null left child, and a real child 5
- We remove ***n*** from the tree and connect 5 back to the parent of ***n***
- **Note that this still works if n is a right child and has a left child. (Exercise).**





Deletion – with two children

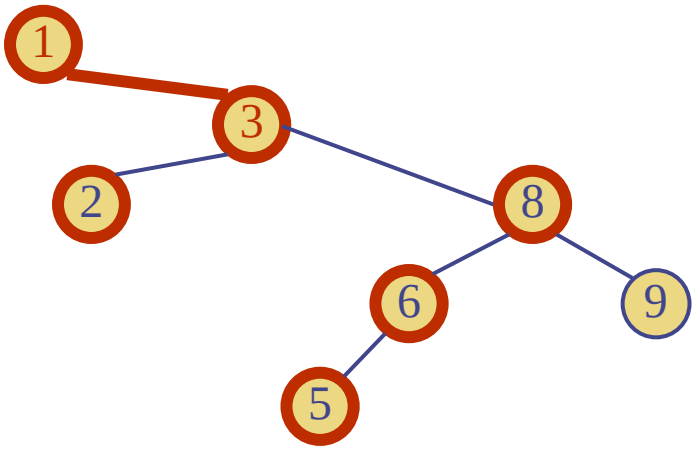
- Example: remove 3
- **What can we do and still keep the property of being a search tree?**



- "Search tree" means an "inorder traversal" visits the keys "in order"
- As a sorted list, we would have [1,2,3,5,6,8,9]
- If we want to remove '3' then a way to do this with minimal change to the rest of the list is to copy a key k' that is adjacent to '3' on top of '3' and then delete that key k'
  - Options in this case are k' being 2 or 5. We will focus on the nextKey, that is, '5'

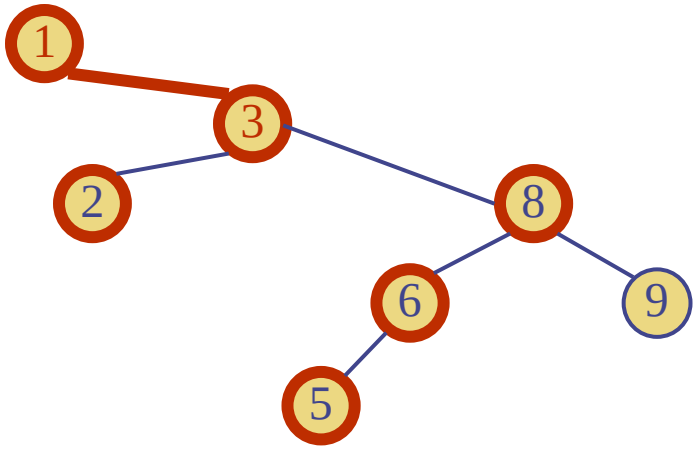
Deletion – with two children

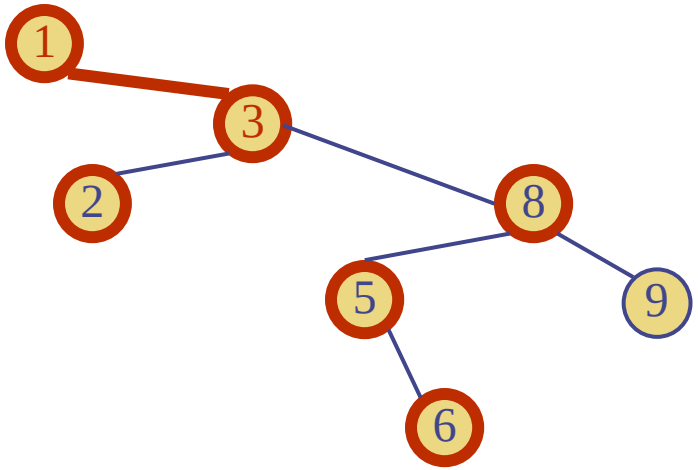
- Example: remove 3
- **What can we do and still keep the property of being a search tree?**



- "Search tree" means an "inorder traversal" visits the keys "in order"
- If these were a sorted list then "adjacent in the sequence" nodes 2 and 5 could be involved in the removal of node 3
- Can the next key after 3 ever have any left children?
- Can the next key after 3 ever have any children?

Deletion – with two children





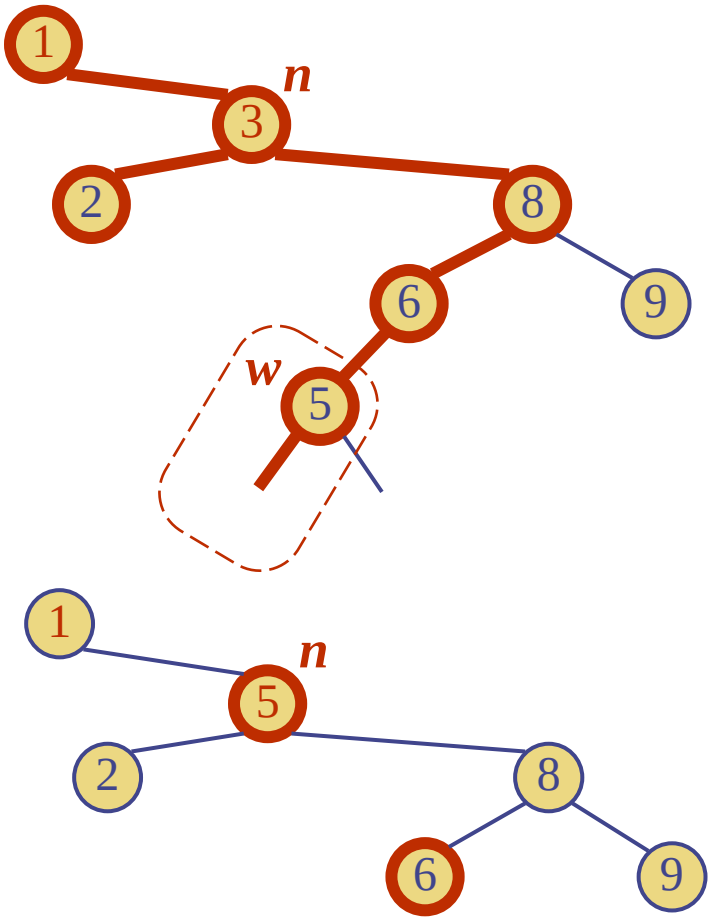
- Can the next (using inorder) key after 3 ever have any children?
- YES. The right hand tree is also a possible search tree with same keys
- To access the nextKey, we need to go right, and then recursively-go-left
- IMPORTANT: that that the next key from 3 cannot have a left child: Proof by contradiction. If it did then the left child would not be "nextKey"

Deletion – with two children

Example: remove 3

The key node n has two internal children

1. we find the internal node w that follows n in an inorder traversal
2. we copy $key(w)$ into node n
3. we remove node w by means of same procedure as before for "one child"



Deletion – with two children

General procedure:

- Consider the case where the key k to be removed is stored at a node n whose children are both internal
  - we find the internal node w that follows n in an inorder traversal
  - we copy $key(w)$ into node n
  - we remove node w and its left child z (which must be a leaf) by using the same procedure as discussed earlier for "remove node with one child"

- Exercises (offline):
  - convince yourself this works properly even when w has a right child
  - Check that it is still O(h)

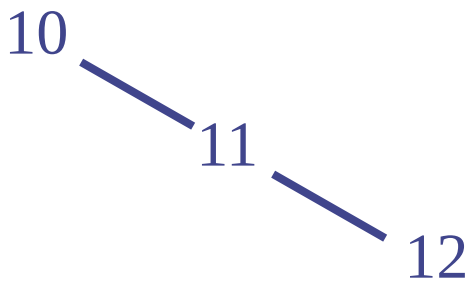
Exercises (offline)

In the "2 children deletion", we had "we find the internal node that follows n in an inorder traversal"

1. Could it also have been "precedes" instead of "follows"?
2. Might making a good choice of "follows" of "precedes" have some advantages?

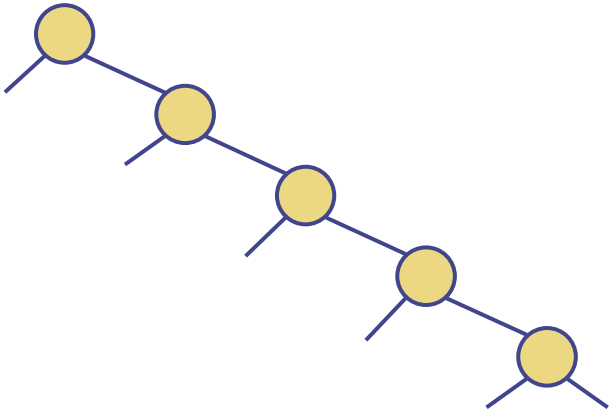
Balanced Trees

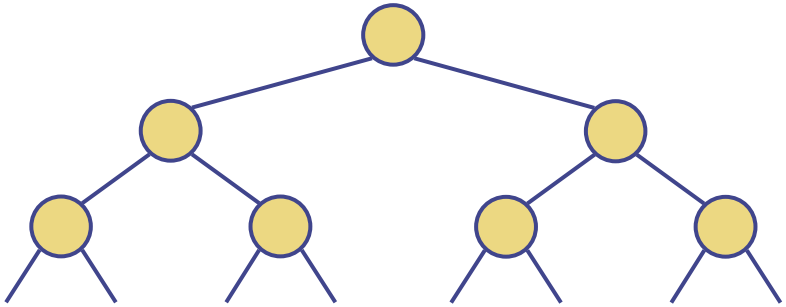
- Binary search trees: if all levels filled, then search, insertion and deletion are O(log N).
  - As they are all O(height)
- However, performance may deteriorate to linear if nodes are inserted in order:



Performance

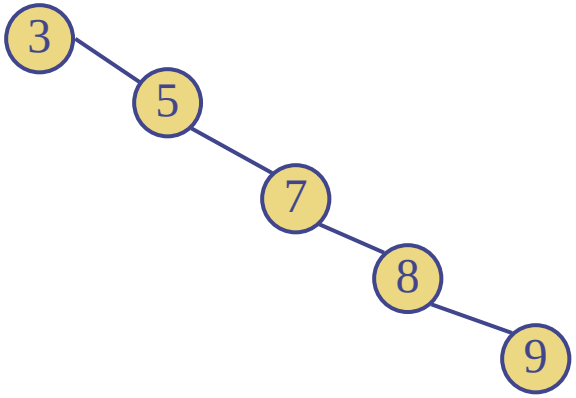
- Consider a binary search tree of height h with n items
  - the space used is $O(n)$
  - methods find, insert and remove take $O(h)$ time
  - EXERCISE: carefully study the methods and check that this is true
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



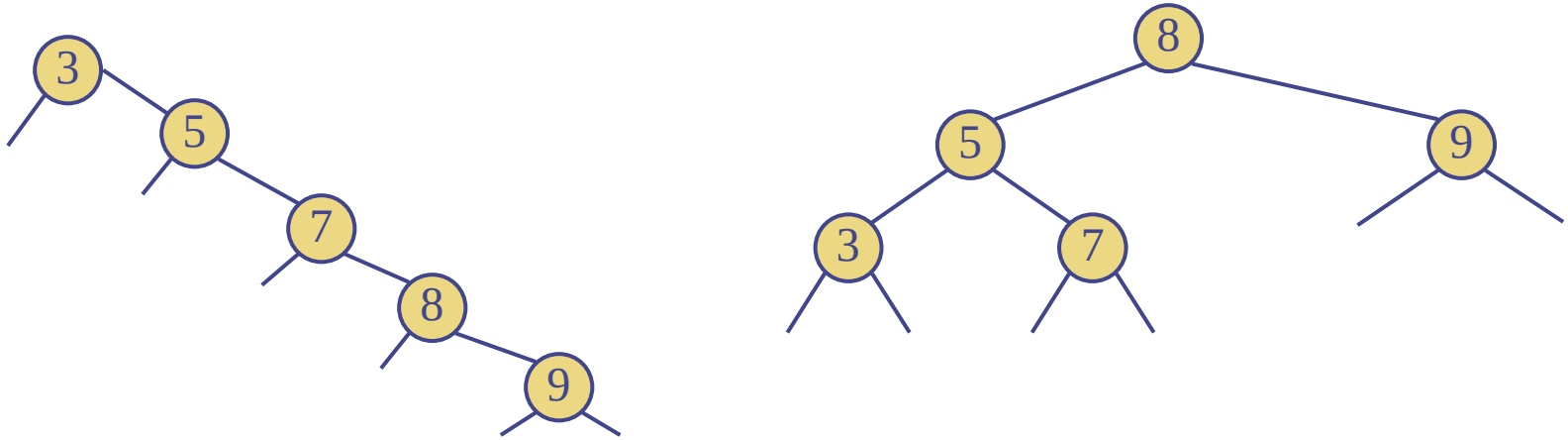


Issues in "Self-Balancing"

- Suppose you have a very imbalanced search tree – is there always a corresponding balanced search tree?
- Exercise (online): find a balanced tree for



Example



- For any number of nodes, there is (obviously?) a balanced tree
- Could balance by a "total rebuild"
  - just placing the keys into a new tree in the correct order

Solution: Self-Balancing

Goal of "Self-Balancing":

- Constantly re-structure the trees:
- Keep the trees height balanced so that the height is logarithmic in the size
- Performance always logarithmic.

Issues in Self-Balancing

- Suppose you have a very imbalanced search tree, there are always corresponding balanced search trees
- Could make trees balanced using a "total rebuild"
  - But would require O(n), and so very inefficient compared to the desired O(log n)
- Re-balancing needs to be O(log n) or O(height)
- Suggests re-balancing needs to just look at the path to some recently changed node, not the entire tree
- A priori, it is not at all obvious that this is possible!
- In a later lecture we show how to use "rotations" to adjust the tree whilst preserving the ordering

Minimal Expectations

- The definitions of Map ADT, BST, etc
- How to use binary search trees
- The complexity of operations as function of nodes *n* and height *h*

COMP2054 ADE
Brute force, D&C, heuristics and "Dynamic Programming"

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

General Methods

There are various general methods ("paradigms") for finding solutions to problems.

Common ones include:

- Brute force – "generate and test"
- Divide-and-conquer
- Heuristics
- Dynamic Programming

“Brute Force”

- This is roughly “generate and test”
  - Generate all potential solutions
  - Test for which ones are actual solutions
- Example: we could do “sorting” by
  - Generate all possible permutations
  - Test to see which one is correctly ordered
    - Extremely inefficient, as is O(n!)
- Can be useful in some (small) cases
  - E.g. Due to the simplicity

Divide and Conquer

- Recursively, break the problem into smaller pieces, solve them, and put them back together
  - Merge-sort and Quicksort were classic examples
  - [Not assessed, just an FYI] "Fast Fourier Transform" is an algorithm heavily used in signal processing and engineering
    - In a list of the "top 10 algorithms" https://ieeexplore.ieee.org/document/814652
    - uses "divide-and-conquer" to reduce O(n^2) to be O(n log n)

Heuristics

- "Heuristic" = "rule of thumb"
  - Generally, meant to mean something that gives better decisions, than the naïve methods, but still not necessarily optimal
- Two common types (the term is over-loaded)
  - Decisions within a procedure that gives exact/optimal answers, but are designed to make it go faster (usually)
  - Decisions within a procedure that might not give optimal answers, but are designed to give good answers that are impractical to obtain otherwise

“Heuristics in exact methods”

- These are general methods that works in an algorithm that does give exact or optimal answers
  - But need the heuristics to decrease the (average/typical) runtime
  - Examples:
    - “Admissible heuristic” in A* search (from first year AI) – decreases the search time compared to plain search
    - “pick a random pivot” in quicksort

“Heuristics in inexact methods”

- These are general methods that (generally) are not be guaranteed to give the best possible answers, but that can give good answers quickly
  - Used on problems when the exact methods are too slow, e.g.
    - Timetabling and scheduling
    - Many design problems
  - It is a vast research area, e.g.
    - genetic algorithms
    - metaheuristics (simulated annealing, tabu search, etc, etc)
    - approximate greedy methods.
    - (See the AIM module COMP2001)

Greedy algorithms

- A common “heuristic” is to be “greedy”
- Take the decision that looks best in the short term – without looking ahead

Greedy algorithm: optimal

- Sometimes greedy algorithms can still give optimal answers.
- E.g. Prim's algorithm (later lecture) for constructing a Minimal Spanning Tree is a ***greedy algorithm***:
  - It just adds the shortest edge without worrying about the overall structure, without looking ahead.
  - It makes a locally optimal choice at each step.
  - But it turns out that this is sufficient for the final answer to be optimal

Greedy algorithm: non-optimal

- Usually greedy algorithms cannot guarantee to give optimal answers
  - but often still give (nearly) optimal answers in practice
- Example: "Change-giving":
  - Problem: given a collection of coins (a multi-set, that allows repeated elements), and a desired target for the change. Supply the change in as few coins as possible:

“Min-Coins-Change-Giving” 1

- INSTANCE:
  - Given a set S of coins and their values x[]
    S = { x[1] , x[2], … x[n] }
    (coins can be repeated, so S is actually a multi-set}
  - A target K (the total value to be returned)

- TASK:
  - Find the set, a subset of S, with the minimum number of coins but whose total value, total sum, is **exactly** K.
  - That is, supply the change in as few coins as possible.
  - (Or show it is not possible with the given coins.)

“Min-Coins-Change-Giving” 2

Examples

- S = { 50, 20 10 } K = 15
  - There is no solution – “I have to go and get more change” ☺
- S = { 50, 20 10 } K = 10
  - There is a solution with 1 coin {10}
- S = { 50, 20 10 } K = 0
  - There is solution with 0 coins {}
    - Might seem bizarre to consider this – but it is an important special case!
- S = { 50, 20, 10, 10, 5, 2 } K = 15
  - There is a solution with 2 coins {10, 5 }
  - but no solution with just 1 coin

"Min-Coins-Change-Giving" 3

- INSTANCE: Given a set S of coins and their values x[]
  S = { x[1] , x[2], … x[n] }, and a target K
- TASK: Find the set with the minimum number of coins and whose total value is **exactly** K.

- "Obviously" this can be solved by enumerating all possible subsets of S, selecting those that sum to K, and picking a subset with the fewest elements
  - But with n coins there are 2^n subsets and so this "generate-and-test" naïve algorithm is exponential in the worst case.
  - Can we do better? Firstly, consider greedy strategies:

Change-Giving Greedy algorithm?

- Greedy strategy:
- Iterate the process of:
  - Pick the largest coin which is still available and does not cause to exceed the target
- Often this will work fine, e.g.
- Coins = {50,50,20,20,10,5,2,2,1,1}. "Change": 73
  - Greedily pick 50, leaving change = 23
  - Greedily pick 20, leaving change = 3
  - Greedily pick 2, leaving change = 1
  - Greedily pick 1, leaving change = 0
- Answer: 50+20+2+1 = 73

Change-Giving Greedy algorithm? 2

- Sometimes it fails, e.g.

- Coins = {5,2,2,2,2}. Change= 8
  - the greedy choice of the largest coin '5' is a 'fatal mistake' – as it not part of any solution

- Coins = {50, 50, 20, 20, 20, 2, 2, 2, 2, 2} Change= 60
  - Greedy method picks { 50, 2, 2, 2, 2, 2 } - 6 coins
  - But can do it with { 20, 20, 20 } – 3 coins

Dynamic Programming (DP)

- DP is a general method that can be suitable when the optimal solutions satisfy a "decomposition property"
  - (Ignore the choice of the name – "dynamic" is not a very helpful jargon.)
- The general idea is roughly:
  - Splitting an optimal solution into sub-solutions corresponds to splitting the problem into sub-problems and the sub-solutions are optimal for the sub-problems
  - So optimal solutions can be built out of optimal solutions of (smaller) sub-problems
  - Hence:
    "solve small sub-problems first, then build up towards the full solution."
  - (Difference from divide-and-conquer is that in DP the sub-problems can overlap.)

"DP" for Change-giving/Subset Sum

- Firstly, consider just giving exact change and not worrying about the number of coins. The problem is better known as
- "Subset-Sum":
  - Given (multi-)set S of positive integers x[i] and a target K
    Is there a subset of S that sums to exactly K?

- (Note: looks innocuous, but can be hard.)

DP for Subset Sum 1

- Algorithm: Consider the numbers one at a time keeping track of "which subset sums are possible so far".

- Main data structure:

  - Boolean Array, Y, for [0,…,K]
    - Y[m] = true iff some subset has been found that sums to m

DP for Subset Sum 2

- The simple underlying idea is to suppose we have found all the subset-sums for x[0],…x[i-1] and then want to also add the effect of x[i]
  - **if some subset summed to m, then with the inclusion x[i] we can also find a subset that sums to m+x[i]**
  - Note this works:
    upwards from small sets with small sums, moving to larger sets with larger sums

** DP for Subset Sum **

```
Input: x[0],…,x[n-1] and K
Initialise all Y[m] = false  for m=1,…K
Y[0] = true;   // As can always provide no change

for (i=0 ; i<n ; i++) { // consider effect of x[i]
      for (m=K-x[i] ; m>=0 ; m--) {  // Exercise: Why "scan down"?
            if (Y[m]==true) {
                  // m was achievable with x[0]…x[i-1]
                  // hence now also m+x[i] is achievable
                  if (m+x[i] == K ) return success
                  if (m+x[i]   < K ) Y[ m+x[i] ] = true
      }
    }
 }
```

Complexity 1

- Outer loop has to consider all the coins
  - hence O(n)
- Inner loop scans the entire array Y,
  - hence O(K)

- Overall is O(n K)

- Much better than O(2^n) ?

Complexity 2

- Overall is O(nK)
  - However, "K" has the "hidden exponential" if it is represented in binary:
  - The relevant input size is the number of bits B that are needed to represent, B=O(log(K))
- The complexity in terms of the size of the binary input is $O(n\ 2^B)$
  - Is called "pseudo-polynomial".

[Aside] Change-giving problem

- "Exercise": find a fast algorithm for the general version of this problem, when expressed in binary, or show one does not exist.
  - Fast means "polynomial in the number of **binary** digits needed to write down the problem"
- Note:
  - See COMP3001 – Computability - lectures on "**P**" and "**NP**".
  - Change-giving is a version of "SUBSET SUM" and "**NP**-hard".
  - Not meant as a real exercise for COMP2009 !!! ☺
    - see "millennium prize problems" there is a $1m prize for solving this "exercise" ☺, or showing that no solution exists
    - It is (essentially) the "**P** vs **NP** problem"
    - No one has solved it in many decades of effort
    - No-one has even found a sub-exponential algorithm
    - Just an FYI: No need to know for this module!

Min-Coins version

- Previous just asked if it is possible to do the change
- But want to minimise the coins
- Hence, need to not just track
  - "is a given target possible?"
- But also the minimum number of coins that are needed

DP for Min-Change-Giving 1

- Algorithm: As before, inspect the coins one at a time keeping track of the best answers obtained with the coins inspected so far
- Main data structure:
  - Integer Array, Y, for [0,…,K]
    - Y[m] = -1 if have not found any sum for m as yet
    - Y[m] = c >= 0 means that have found that can achieve the sum m with c coins.
- Aim: when the algorithm finishes then Y[K] will be the minimum number of coins
  - "Side-effect": All the values of Y[m] m < K, will also be the minimum number for a value of m.

DP for Min-Change-Giving 2

- The simple underlying idea is to suppose we have found all the best answers for the coins x[0],…x[i-1] and then want to also add the effect of one more coin x[i]
  - **if some set summed to m, then with the inclusion x[i] we can also find a subset that sums to m+x[i]**
    - **and with one more coin than was recorded as possible for m**
    - If a set of coins had already been found then take the one that gives the minimum.

DP for Min-Change-Giving (schematic)

Input: x[0],…,x[n-1] and K

Initialise: Y[0] = 0, // as can give a change of 0, with 0 coins
and Y[m] = -1 for m > 0

```
for (i=0 ; i<n ; i++) { // consider effect of x[i]
    for (m=K-x[i] ; m>=0 ; m--) { // scan array
        if (Y[m] >= 0 ) {
            // value m was achievable with x[0]…x[i-1] using Y[m] coins,
            // so,  m+x[i] is now achievable with Y[m]+1 coins
            // but might already have found a better answer
            // stored as Y[m + x[i] ] so then take the best
            if (Y[m + x[i] ] == -1 )
                Y[ m + x[i] ] = Y[m]+1
            else
                Y[ m + x[i] ] = min(  Y[m + x[i] ]  , Y[m]+1  )
        }
    }
}
```

Worked example

Input: x[] = {5,2,2,2,1} and K=6

```
Initialise: Y[0] = 0 and Y[m] = -1 for m > 0
for (i=0 ; i<n ; i++) { // consider effect of x[i]
        // state of Y here for each k is given below
       for (m=K-1 ; m>=0 ; m--) { // scan array
                    if ( Y[m] >= 0 ) {
                                Y[m + x[i] ) = min(  Y[m + x[i] ]  , Y[m]+1  )
                    }
        }
}
```

k=0 Y[] = [0,-1,-1,-1,-1,-1,-1] Y[0]=0 for change {}

k=1 Y[] = [0,-1,-1,-1,-1,1,-1] Y[5]=1 for change {5}

k=2 Y[] = [0,-1,1,-1,-1,1,-1] Y[2]=1 for change {2}

k=3 Y[] = [0,-1,1,-1,2,1,-1] Y[4]=2 for change {2,2}

k=4 Y[] = [0,-1,1,-1,2,1,3] Y[6]=3 for change {2,2,2}

k=5 Y[] = [0,-1,1,-1,2,1,2] Y[6]=min(3,1+1)=2 for change {5,1}

Finished: so optimal answer is 2 coins.

[Not assessed] Why does this work?

- Point of DP is that it exploits the case when there is a good decomposition of the problem
  - Doing with optimal of N+M means the val(N) and val(M) are separately done optimally
- Such properties of the solutions are usually expressed by "Bellman Equations"
  - https://en.wikipedia.org/wiki/Bellman_equation
  - DP is a way to exploit these equations and structures.
  - When it works it can give surprisingly good algorithms for problems that otherwise are very difficult

Comments

- The structure of the algorithm for the change giving has many applications
  - We will see a similar structure in the later lectures for finding shortest paths in graphs
  - Many “advanced but highly effective” algorithms can use dynamic programming methods

Minimum Expectations

- Have a broad understanding of the different classes of algorithms and be able to give examples of their applications

- Understand the change-giving problem and the associated DP-style algorithm

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054 ADE
Minimum Spanning Trees

Spanning Tree

- Input: connected, undirected graph
- Output: a tree which connects all vertices in the graph using only the edges present in the graph

Minimum Spanning Tree

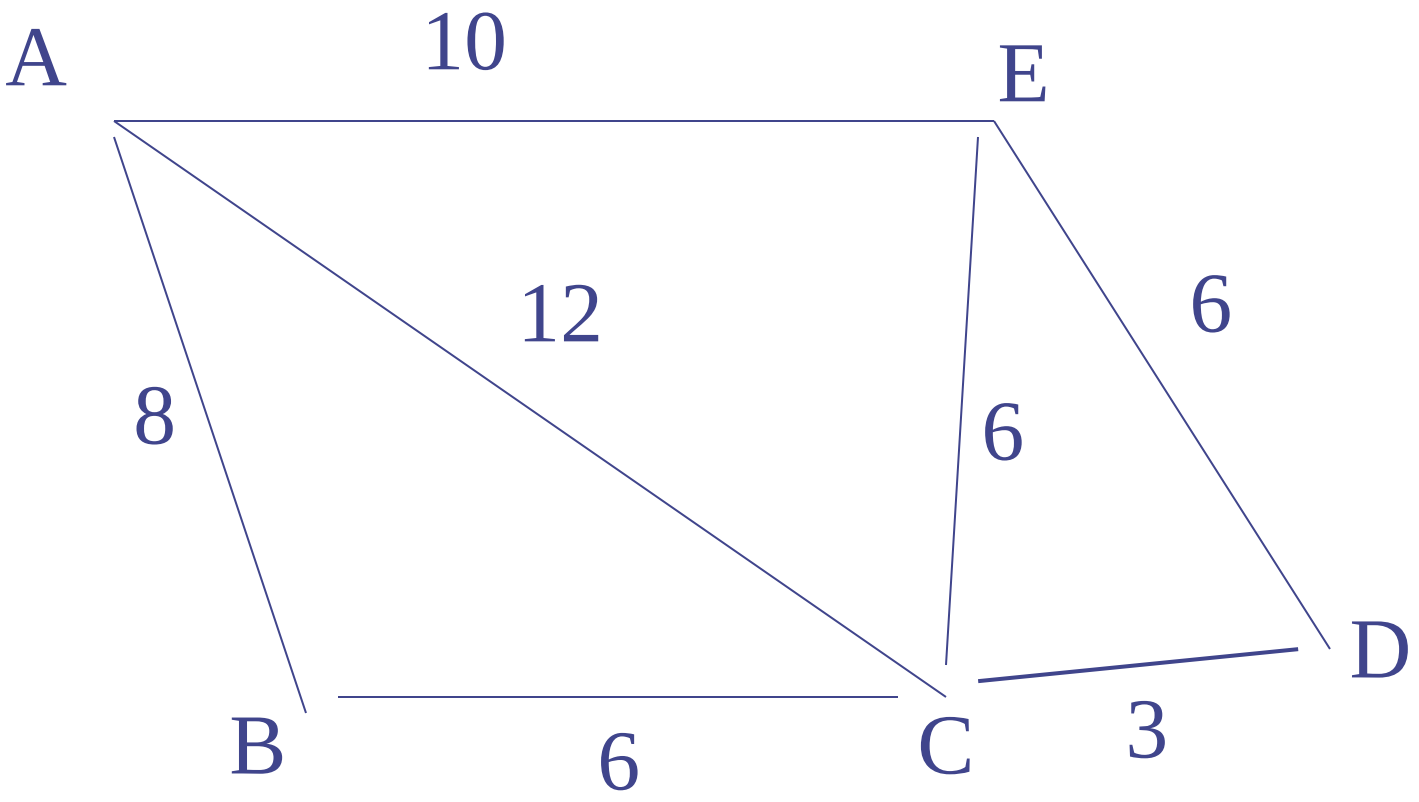
- Input: connected, undirected, weighted graph
- Output: a spanning tree
  - (connects all vertices in the graph using only the edges present in the graph)
  - and is minimum in the sense that the sum of weights of the edges is the smallest possible for any spanning tree

MST Usages

- Gas and water pipelines, optic fibres networks
  - Simply need to build a network to reach every house but minimizing total cost of construction
    - Nodes are supply/delivery locations
    - Edges are "pipes" or "cables" that we can build, and the weight is their cost
    - Aim is to build the cheapest system that meets the supply requirements
      - (In real-world might want to modify to allow redundancy in case of breakages)

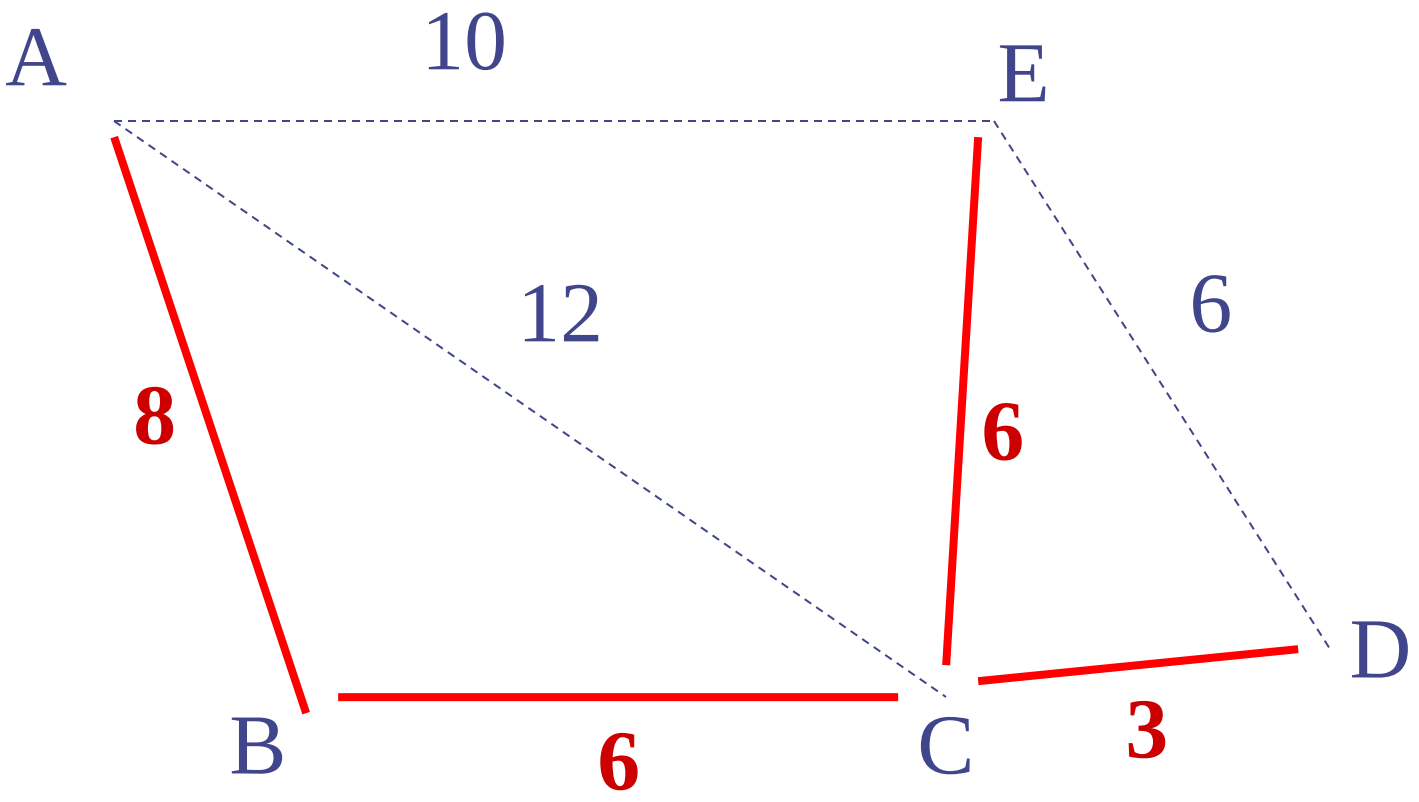
- As part of other graph algorithms
  - E.g. a "Christofides algorithm" to approximately solve the TSP exploits constructing an MST and then converting to a tour https://en.wikipedia.org/wiki/Christofides_algorithm#Algorithm

Example: graph



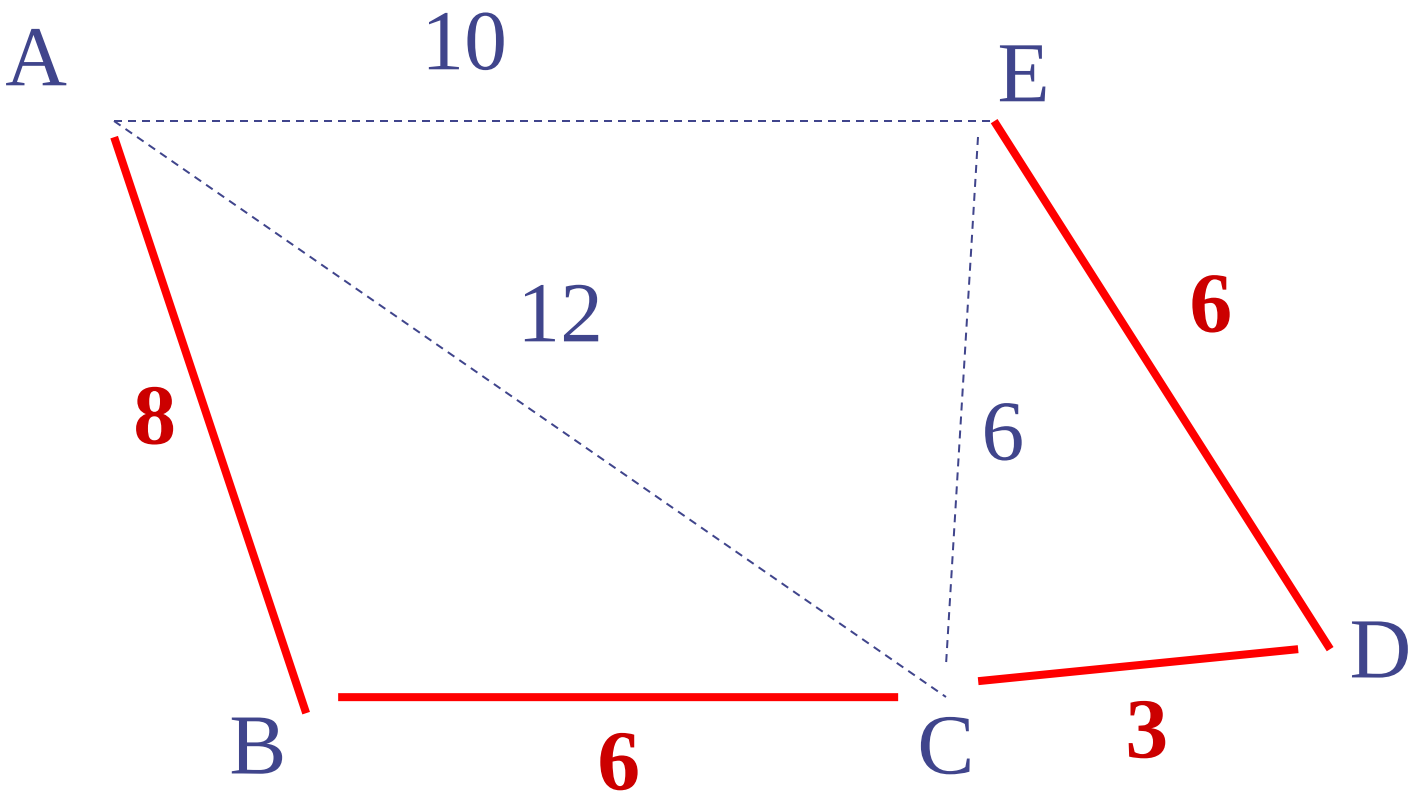
Portions of slides are © 2004 Goodrich, Tamassia

Example: a MST (cost 23)



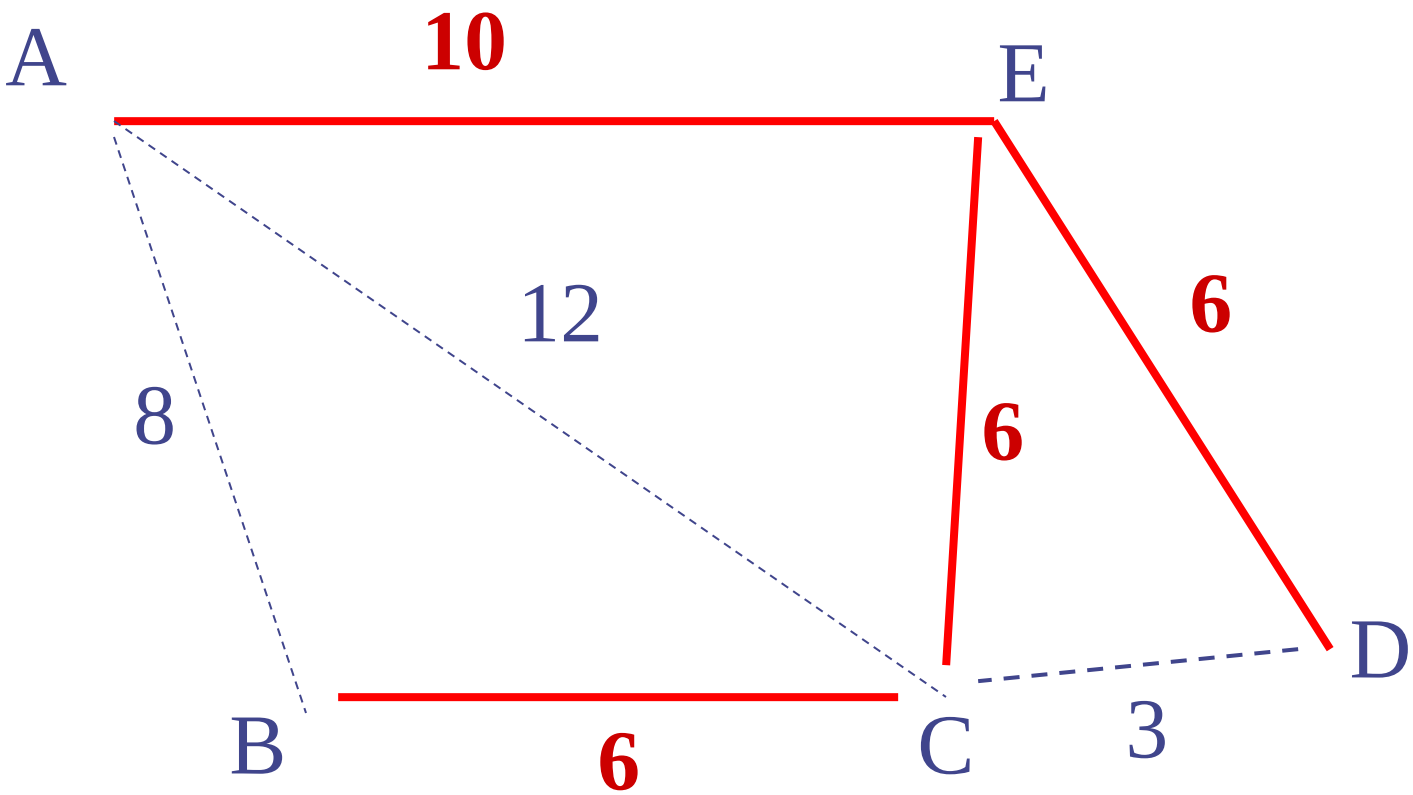
Note: This MST is NOT a path !!!!

Example: another MST (cost 23)



Note: An MST can (sometimes) be a path, but it is still not necessarily the shortest path between the endpoints

Example: not MST (cost 28)



Note: The above is a “spanning tree”, but it is not “minimum”

Why MST is a tree

- We really want a minimum spanning sub-graph
  - that is, a subset of the edges that is connected and that contains every node
- (Assuming all weights are non-negative)
  If the graph has a cycle then we can remove an edge of the cycle, and the graph will still be connected, and will have a smaller weight
- If a graph is connected and acyclic then it is a tree

An MST is (generally) a TREE

- Do not confuse a minimum **TREE** with a "minimum" (shortest) **PATH**
  - Finding the shortest path that goes through all the nodes is a different problem (roughly "TSP" / "Hamiltonian cycle") from the MST (and much harder)
  - It is also different from shortest path between two nodes
- (Many have people confused these on many exams).

Usages?

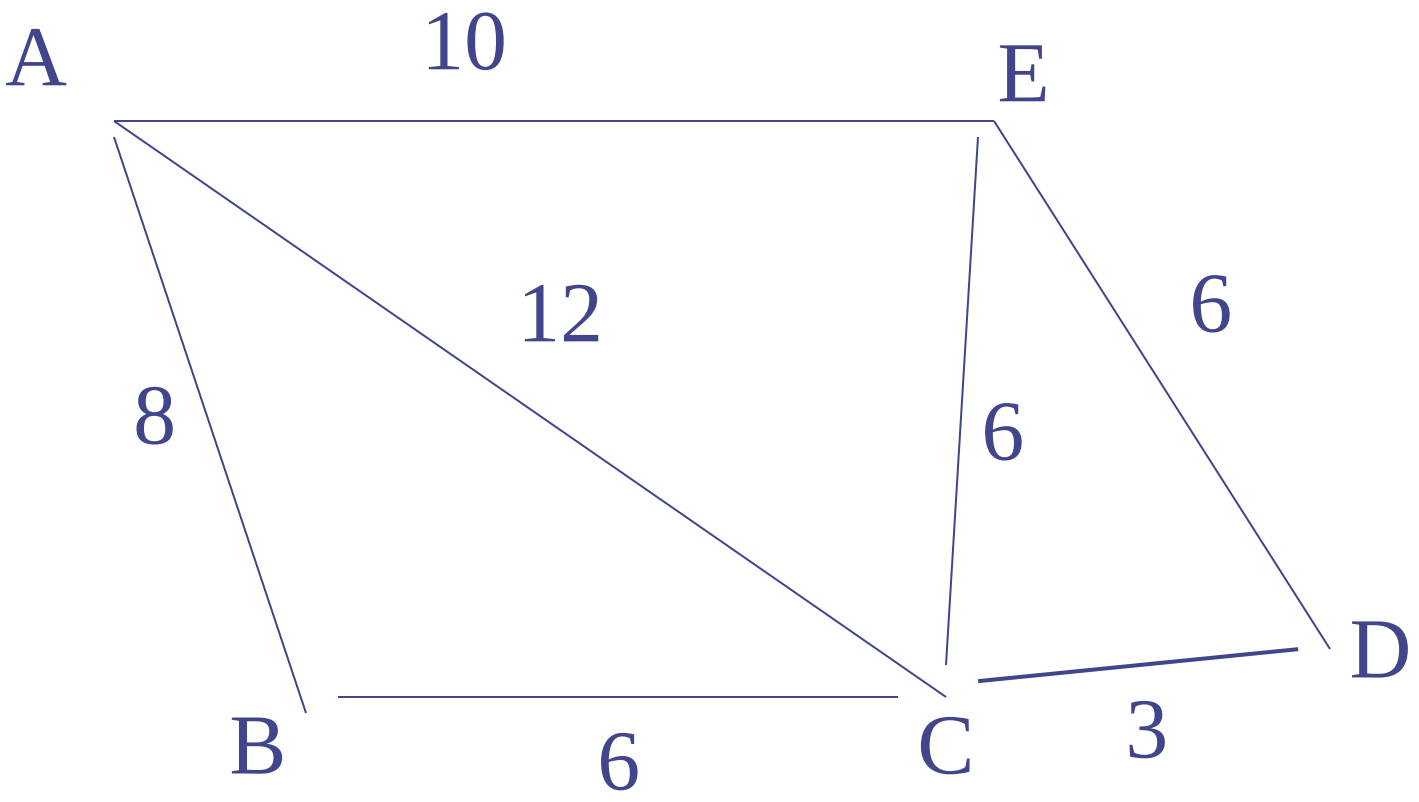
- Gas/water (main) pipelines
  - Connect every house
  - Minimise the total length of all the pipes
  - Follow roads
- Other network problems
- As part of some other algorithm
  - "first find an MST, and then …"

Prim's algorithm

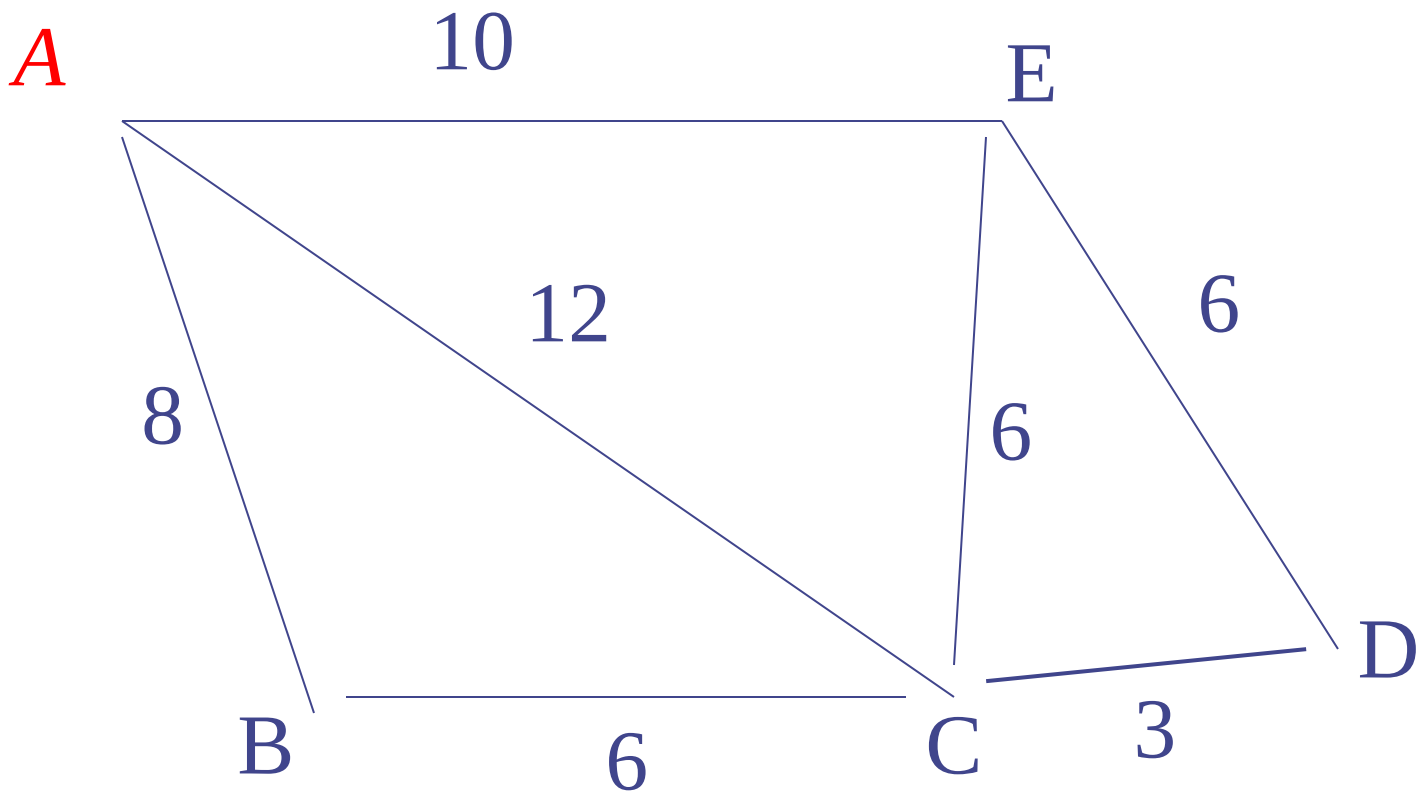
To construct an MST:

- Start by picking any vertex M
- Choose the shortest edge from M to any other vertex N
- Add edge (M,N) to the MST
- Loop:
  - Continue to add at every step a shortest edge from a vertex in MST to a vertex outside, until all vertices are in MST
  - (If there are multiple shortest edges, then can take any arbitrary one)

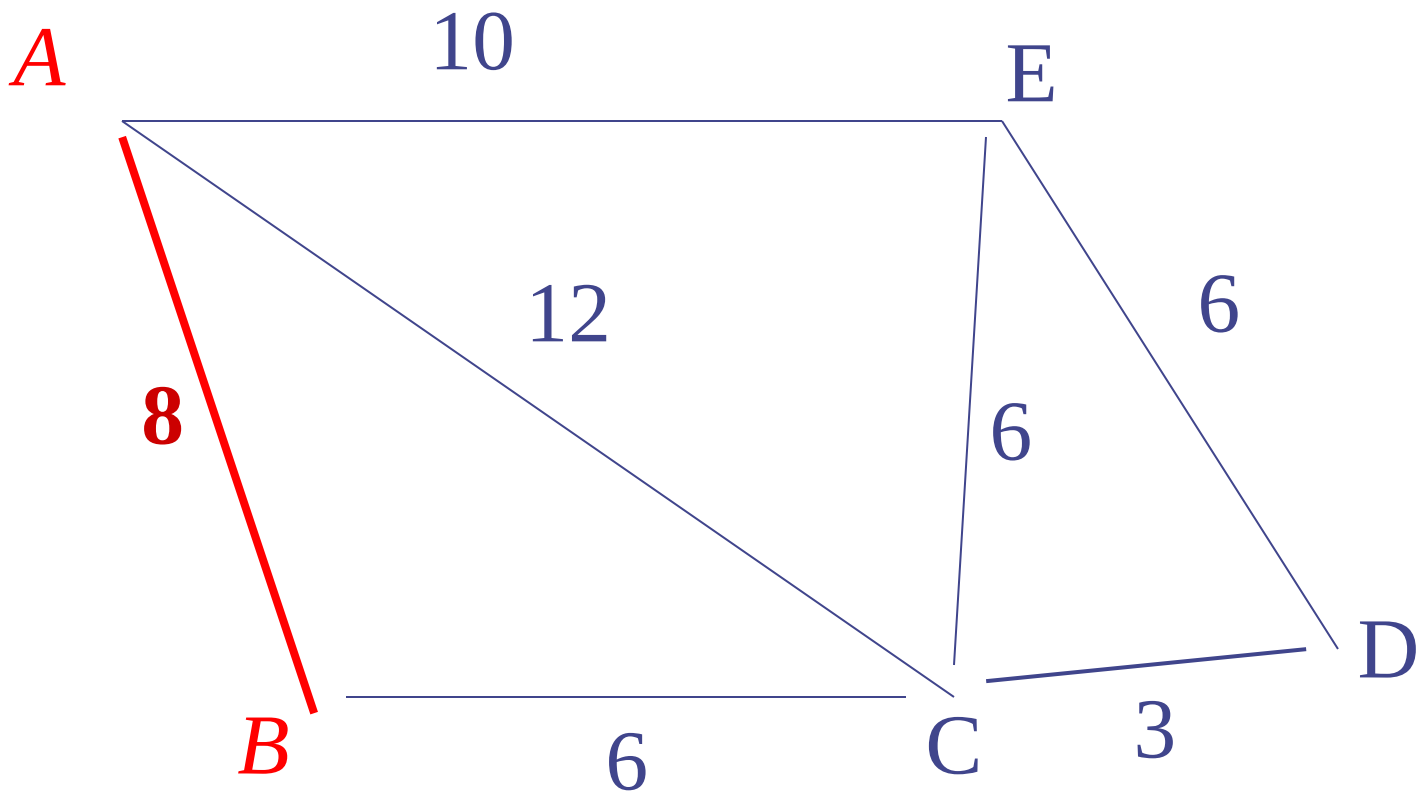
Example



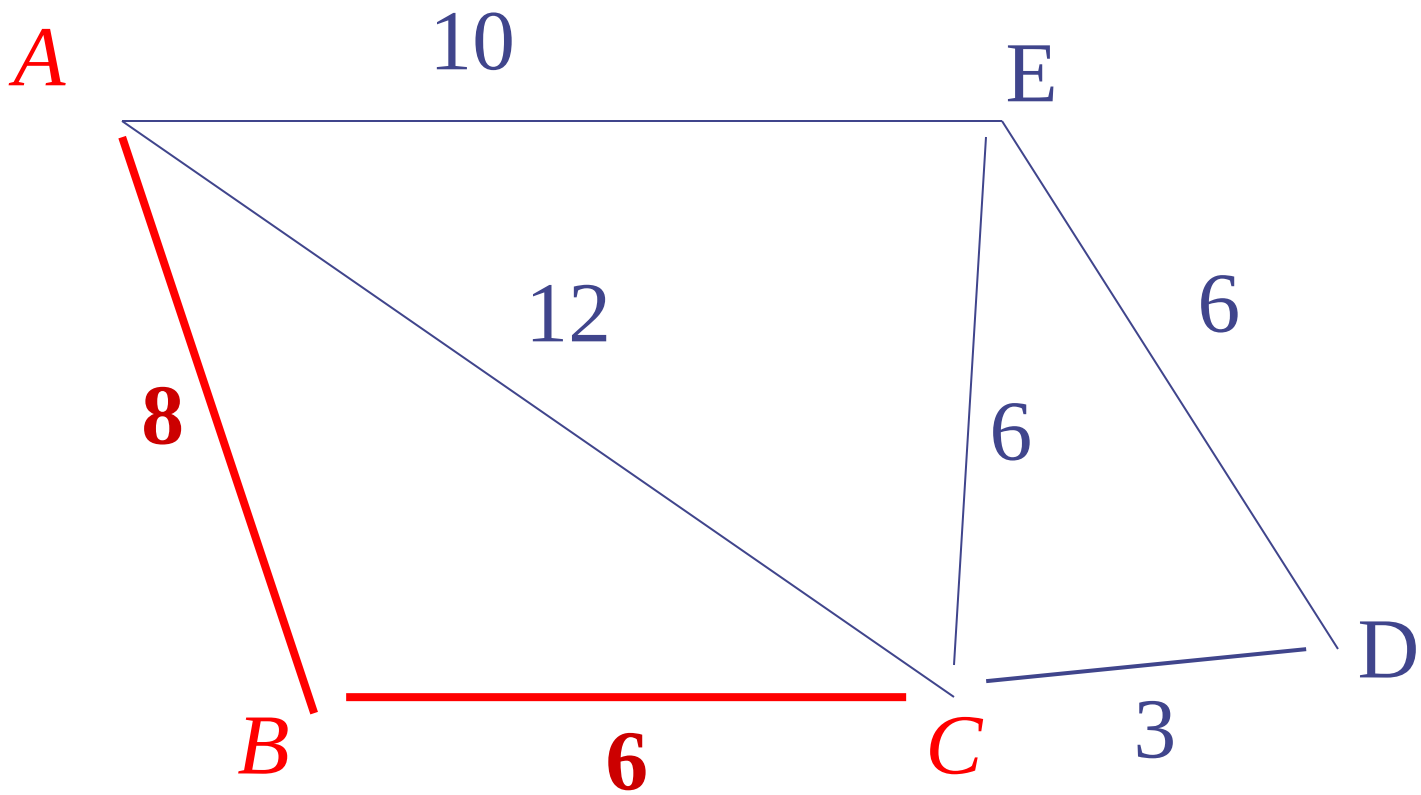
Example



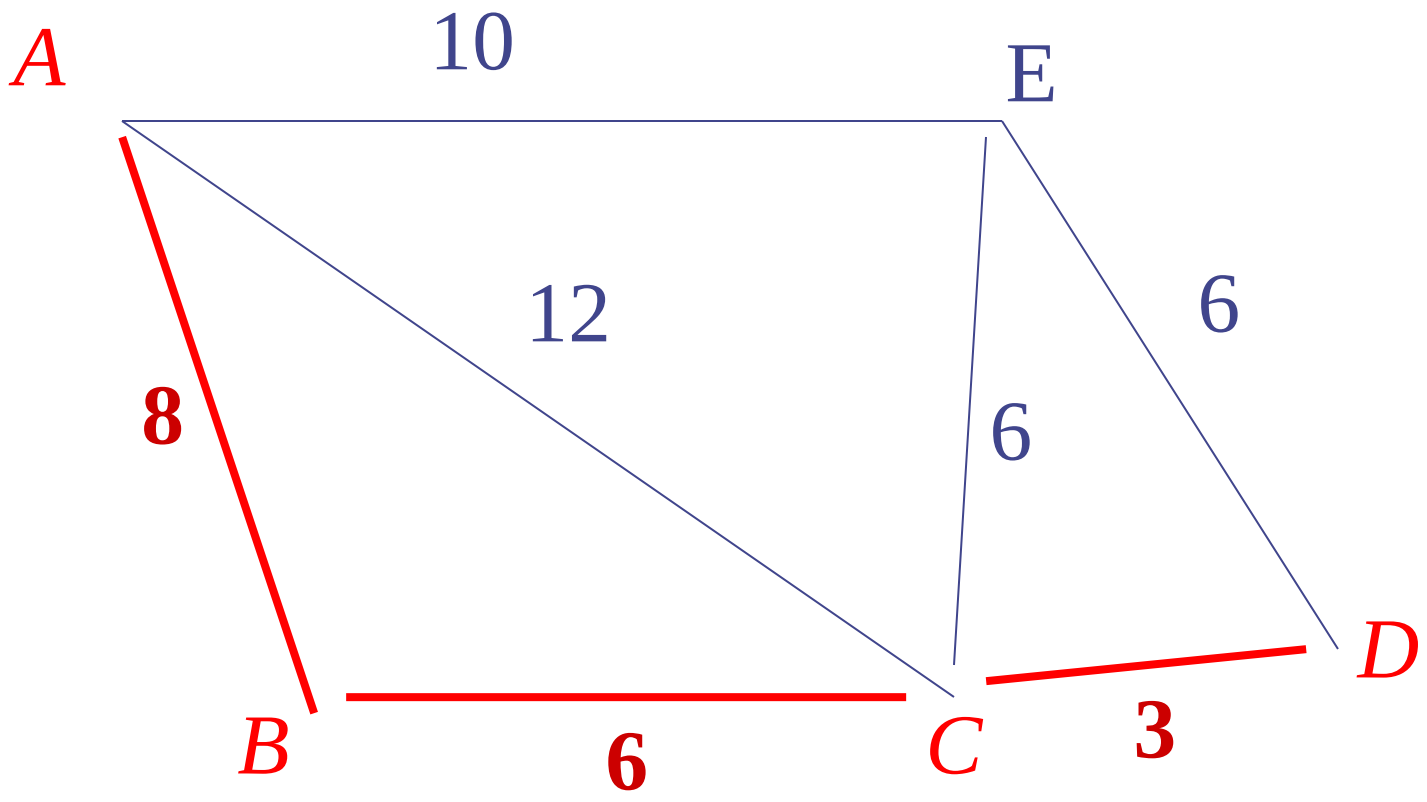
Example



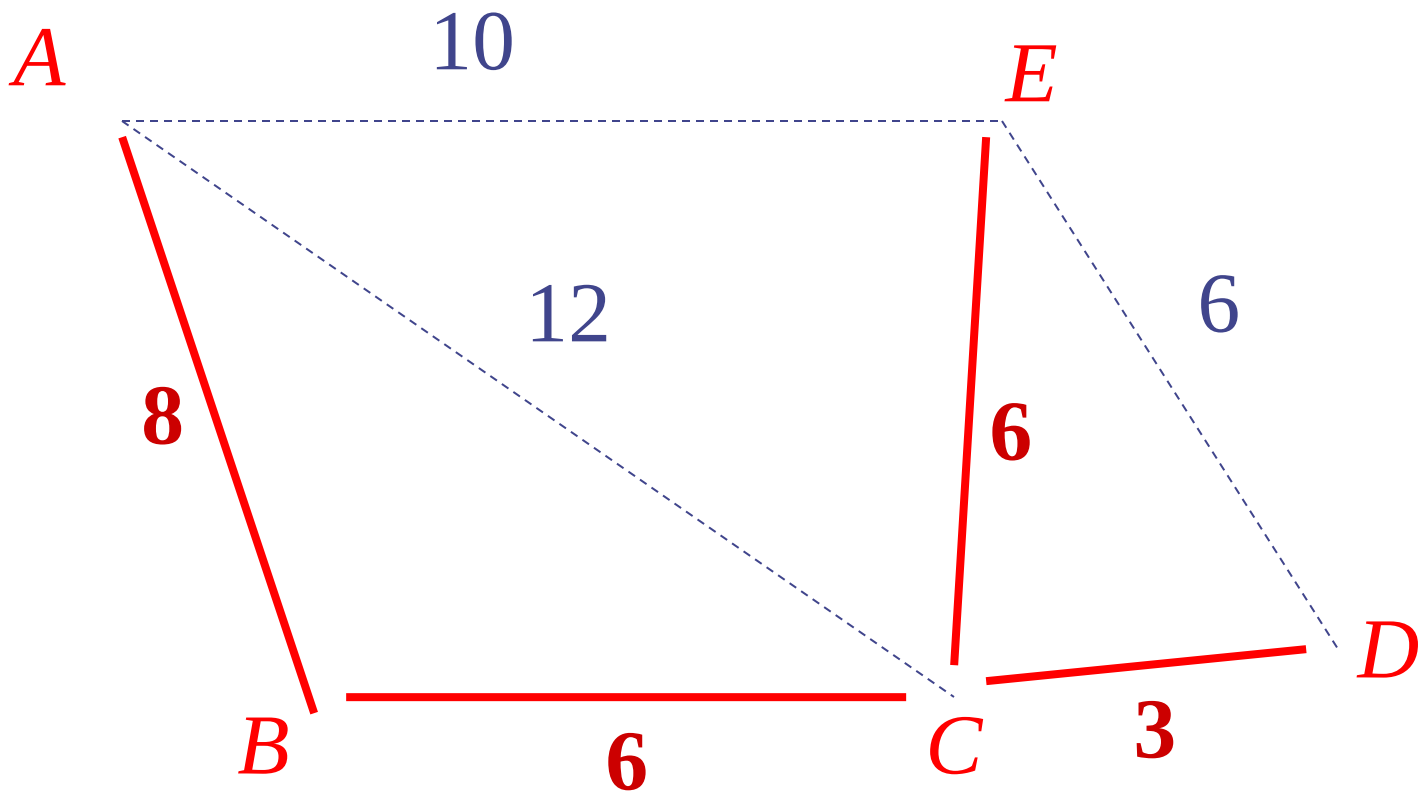
Example



Example



Example



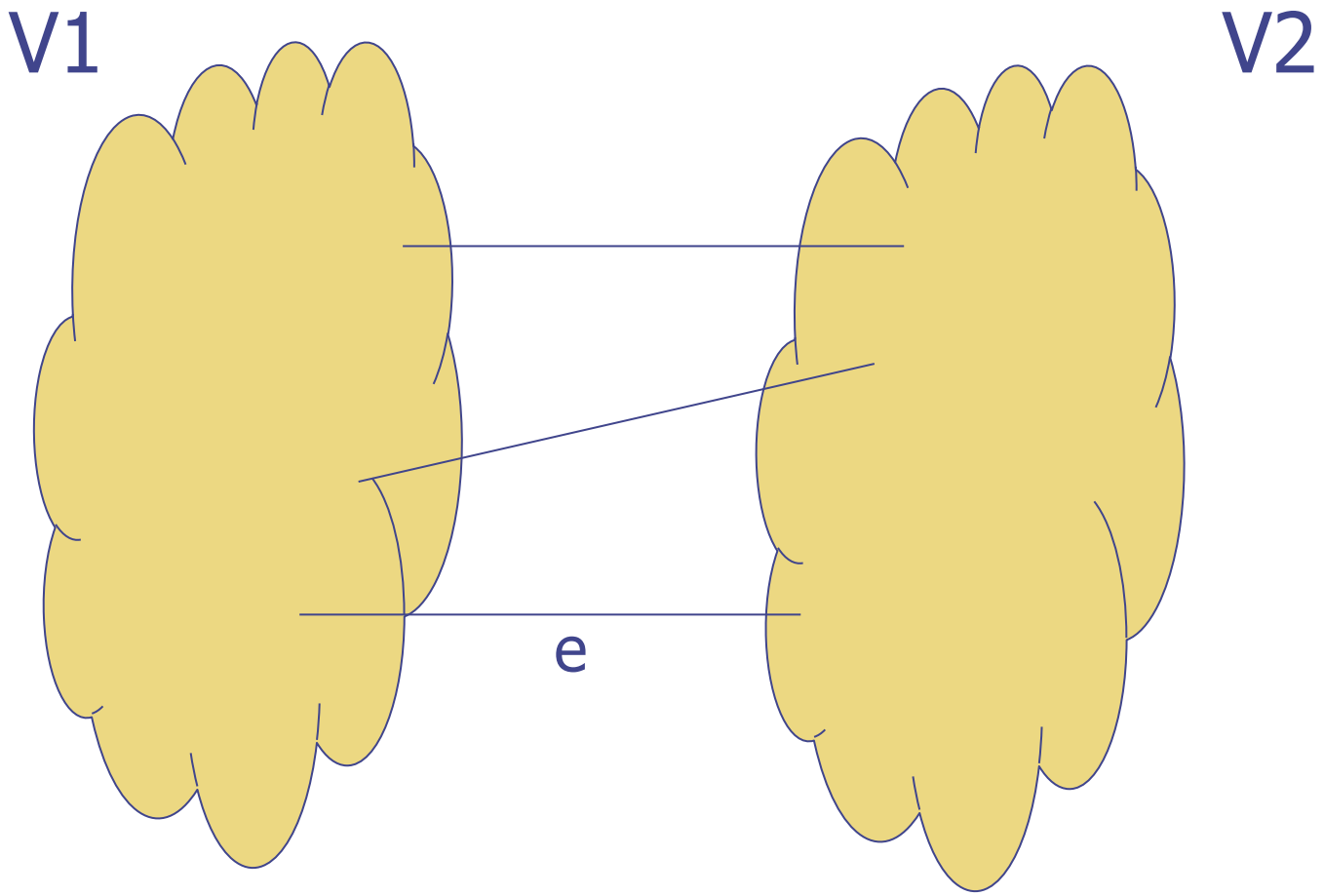
Why is this optimal!?

- [Goodrich/Tamassia Textbook]. Proposition 13.25 (Section 13.7)

- "Let G be a weighted connected graph, and
  - let V1 and V2 be a partition of the vertices of G into two disjoint non-empty sets.
  - Furthermore, let e be an edge with minimum weight from among those with one endpoint in V1 and the other in V2.
  - There is an MST that has e as one of its edges."

Portions of slides are © 2004 Goodrich, Tamassia

Why is this optimal!?

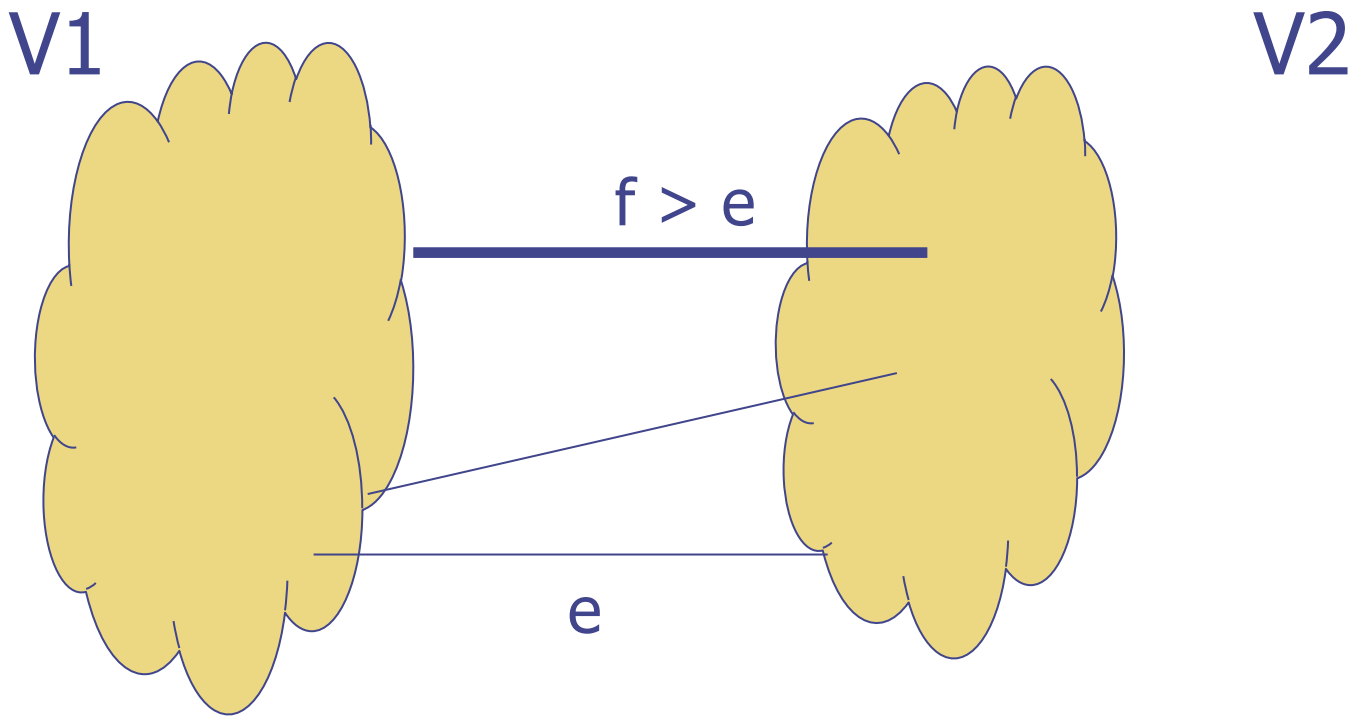


Portions of slides are © 2004 Goodrich, Tamassia

Justification of Prop. 13.25

- Argument by contradiction.
- Suppose that some minimum spanning tree T that is better than all trees containing e.
- Then can add edge e to T and remove some other edge between V1 and V2 and obtain a better MST

Justification of Prop. 13.25



Remove f and replace with e :

- Still gives a spanning tree.
- Gives a better spanning tree.

Prop 13.25 and Prims

- At each stage:
  - V1 = vertices within the current MST
  - V2 = “the rest” (vertices not in the MST)
  - The algorithm adds a minimum weight edge between V1 and V2, and so this edge must be part of some MST
  - Hence, the construction cannot make a “fatal mistake” – at no point can it add an edge not part of an MST

Greedy algorithm

- Prim's algorithm for constructing a Minimal Spanning Tree is a ***greedy algorithm***:
  - it just adds a minimum weight edge
  - without worrying about the overall structure, without looking ahead.
  - It makes a locally optimal choice at each step.
- However, it still gives a globally optimal answer
  - This is fairly unusual
  - Usually greedy methods give answers that are "good but possibly sub-optimal" (e.g. change-giving)

Other MST Algorithms (not assessed)

- Kruskal
  https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

- Reverse-delete
  https://en.wikipedia.org/wiki/Reverse-delete_algorithm

Minimal Expectations

- Clearly know, understand and be able to use
  - Definition of an MST
  - Algorithm to create one
  - Why it gives an optimal (minimum weight) spanning tree

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054 ADE

Binary Search Trees: Balance and Rotations

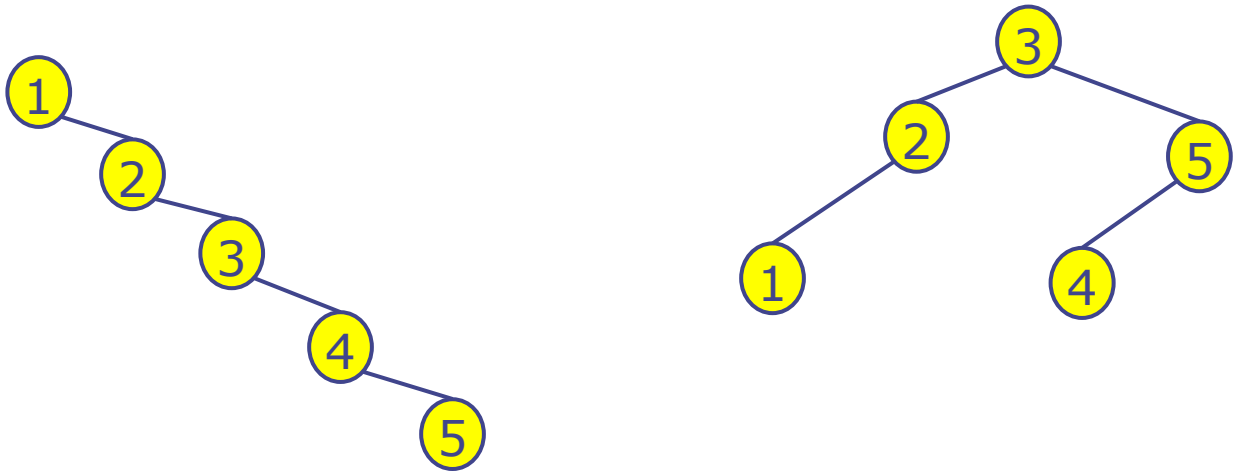
Recap: Binary Search Trees

- A binary search tree (BST) is a binary tree storing key-value entries at its nodes and satisfying the "search tree" property:
  - For any node (not only the root) then
  - Nodes in the **left subtree** have (strictly) **smaller** keys
  - Nodes in the **right subtree** have (strictly) **larger** keys

Hence:

- An in-order traversal of a BST visits the keys in increasing order

The BST Imbalance problem

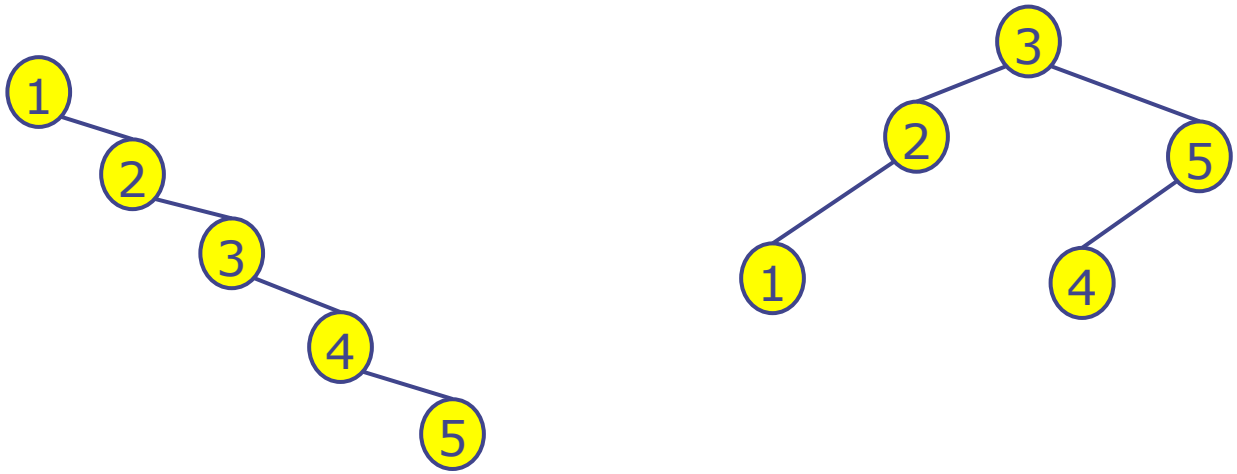


- Both trees are valid BSTs with the same content
- Both have in-order traversal: 1,2,3,4,5. But …
  - The left has height 4
  - The right has height 2
- This matters because the vital BST algorithms are O(height)
- The left tree has a larger height because it is imbalanced: the heights of the left and right subtrees are very different
- A tree is said to be "balanced" if the heights of left and right subtrees of any node are (close to) equal, and so the height is O(log n)

Recall: Issues in Self-Balancing

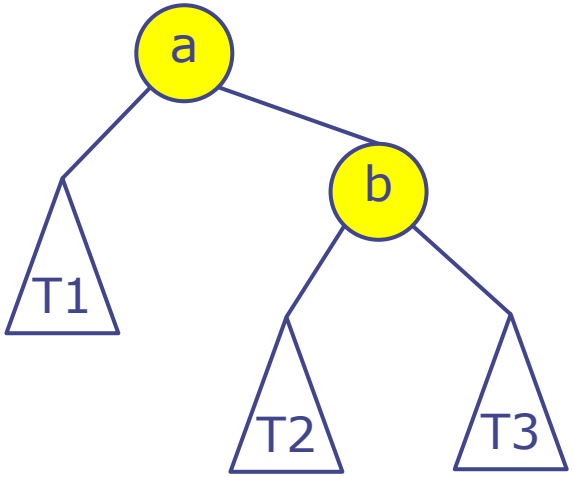
- Suppose you have a very imbalanced search tree, there are always corresponding balanced search trees
- Could make trees balanced using a "total rebuild"
  - But would require O(n), and so very inefficient compared to the desired O(log n)
- Re-balancing needs to be O(log n) or O(height)
- Suggests re-balancing needs to just look at the path to some recently changed node, not the entire tree
- A priori, it is not at all obvious that this is possible!

The BST Imbalance problem



- For any number of nodes, there is (obviously?) a balanced tree
  - Could balance by a "total rebuild"
  - just placing the keys into a new tree in the correct order
  - But total rebuilds would require O(n) time – doing them would destroy the O(log n) efficiency advantages of a BST.
- AIM: Do "small local rebuilds" during insert/delete operations, to maintain the balance, and so retain the O(log n) cost
- This lecture: What "small local rebuilds can we do"?

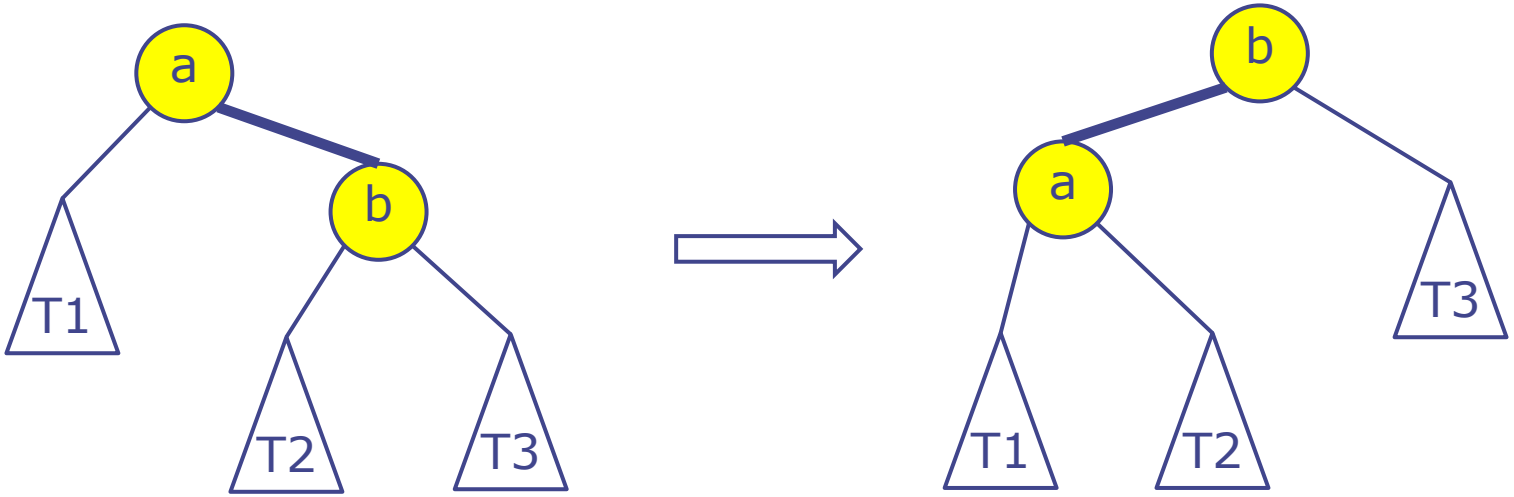
Example of a BST, T



- Where subtrees T1, T2, T3 are BSTs themselves
- Height h(T) = max(1 + h(T1), 2 + h(T2), 2 + h(T3))
- In-order traversal, "In" :

  In(T) = In(T1), a, In(T2), b, In(T3)
- Can we use the same nodes and subtrees to build a different BST?
  - But must have the same In-order Traversal !
  - Too costly to do anything internally to T1,T2,T3 themselves !

Example of a "rotation"

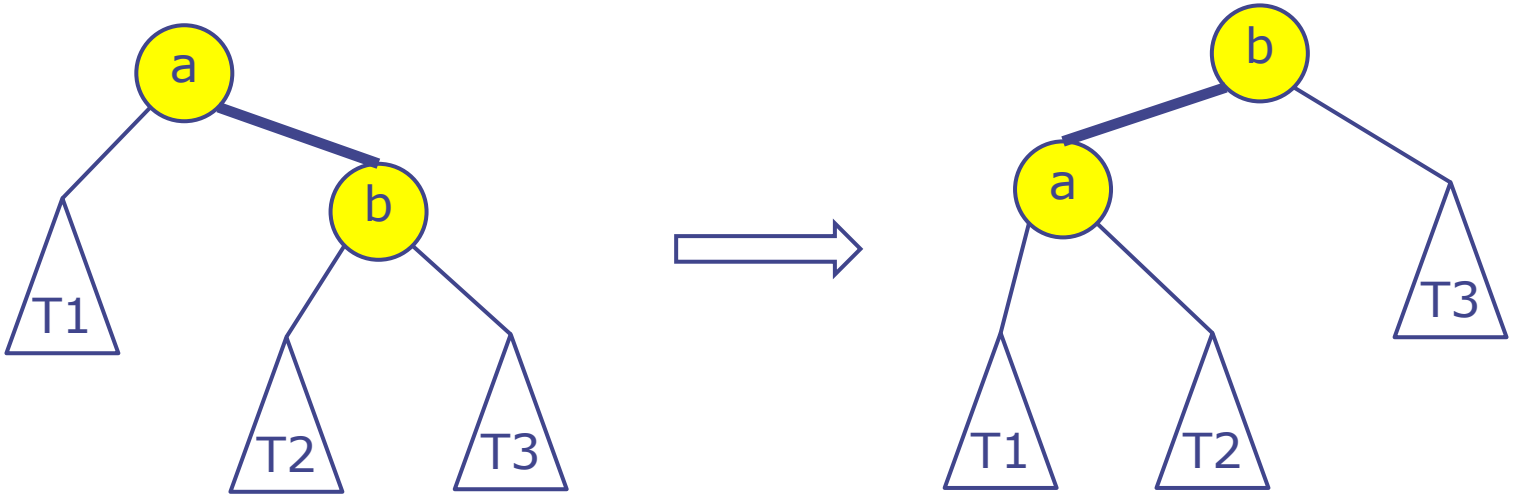


- Both trees have in-order traversal:

  In(T1), a, In(T2), b, In(T3)

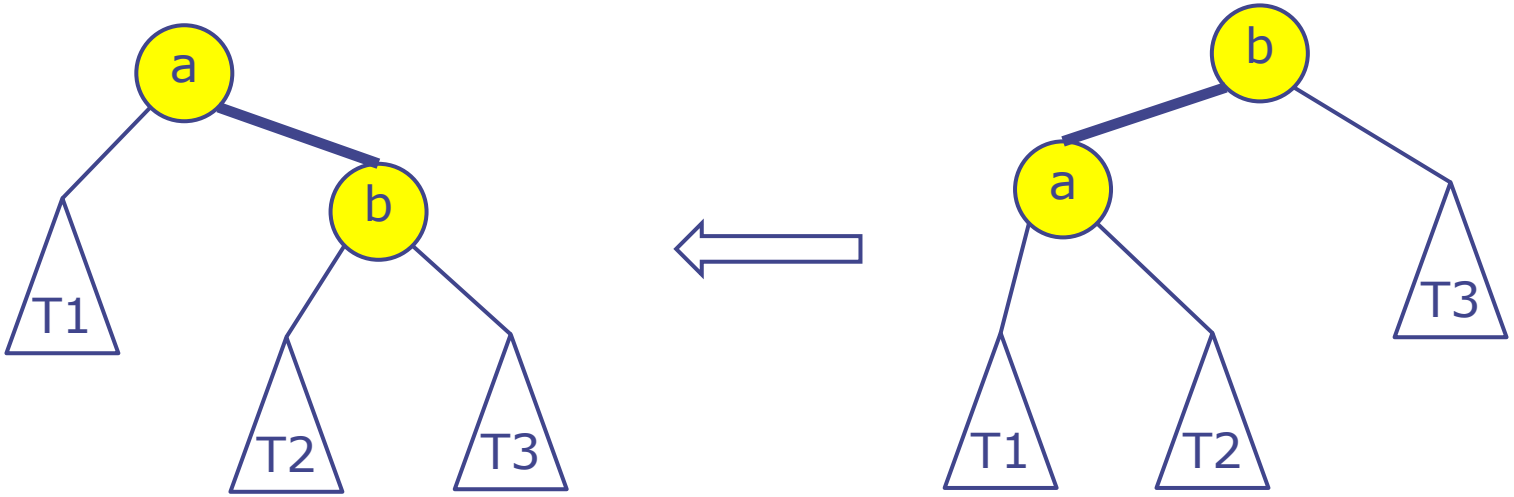
- Depths:
  - Node a moved down by one
  - Node b moved up by one
  - Sub-tree T1 moved down by one
  - Sub-tree T3 moved up by one

Example of a "rotation" 2



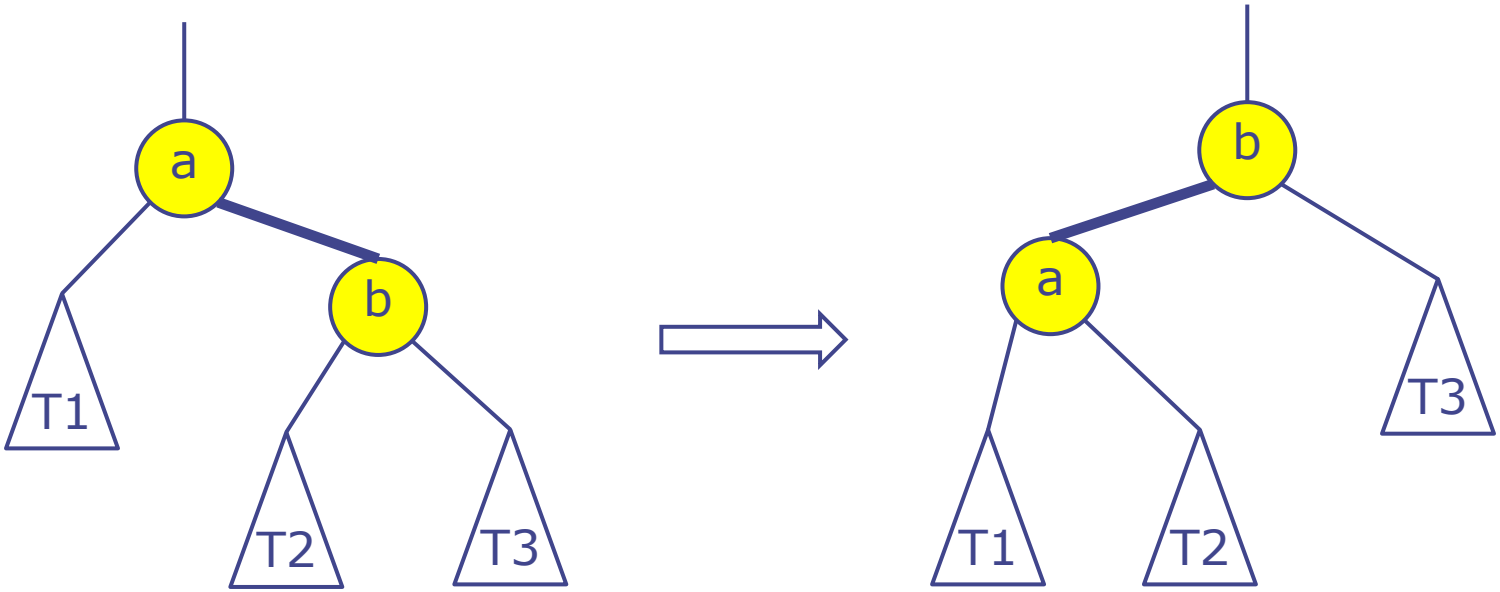
- Called a (single) "rotation" as the edge a-b rotated
- No clear convention as to whether it is a "left rotation" or a "right rotation"
  - Wiki calls it a "left rotation" around node 'a'
  - Clearer to just identify the edge, as the rotation is then fixed
- The root of the tree changed from 'a' to 'b'

Example of a "rotation" 3



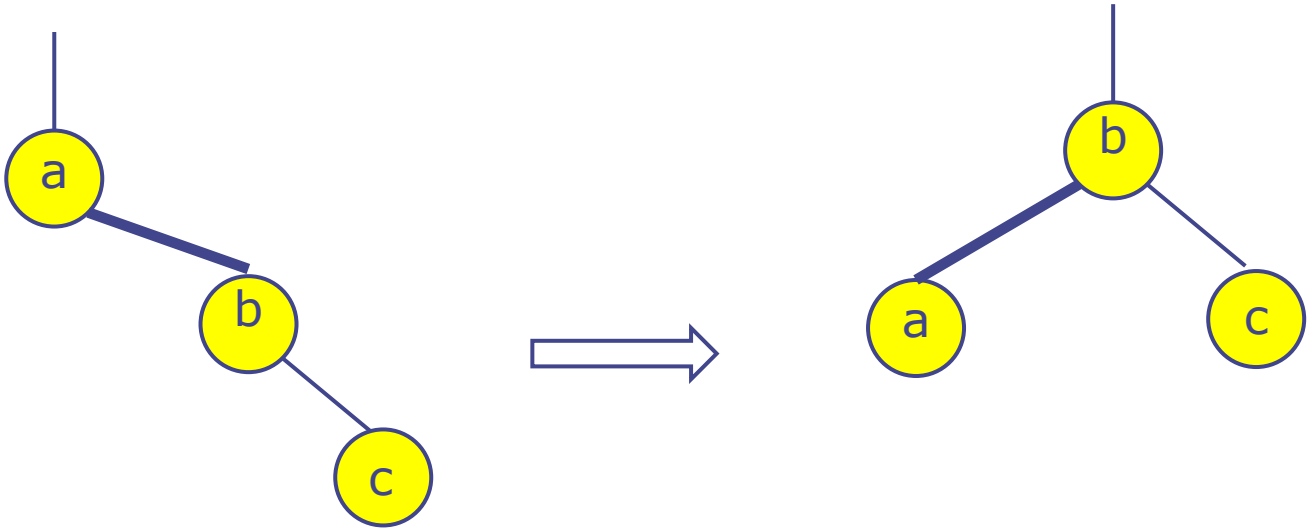
- Unsurprisingly, we can also do the reverse process
  - With the reverse effects on the depths on the sub-trees:
  - 'a' and T1 rise up,
  - 'b' and T3 sink down

Example of a "rotation" 4



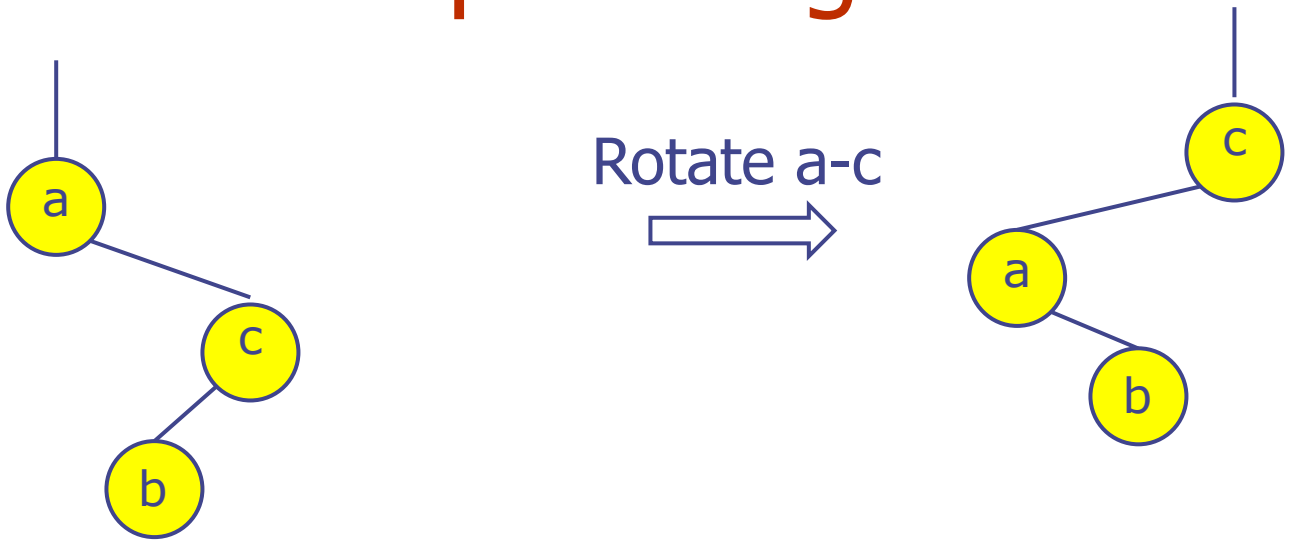
- The tree can also be a sub-tree of a BST
- The parent of a is moved to be the parent of b
- Once we have decided to rotate a specific edge then we only need to look at a fixed number of nodes and edges.
  - Hence, **a single rotation only has O(1) in cost**.

Simple Example – single rotation



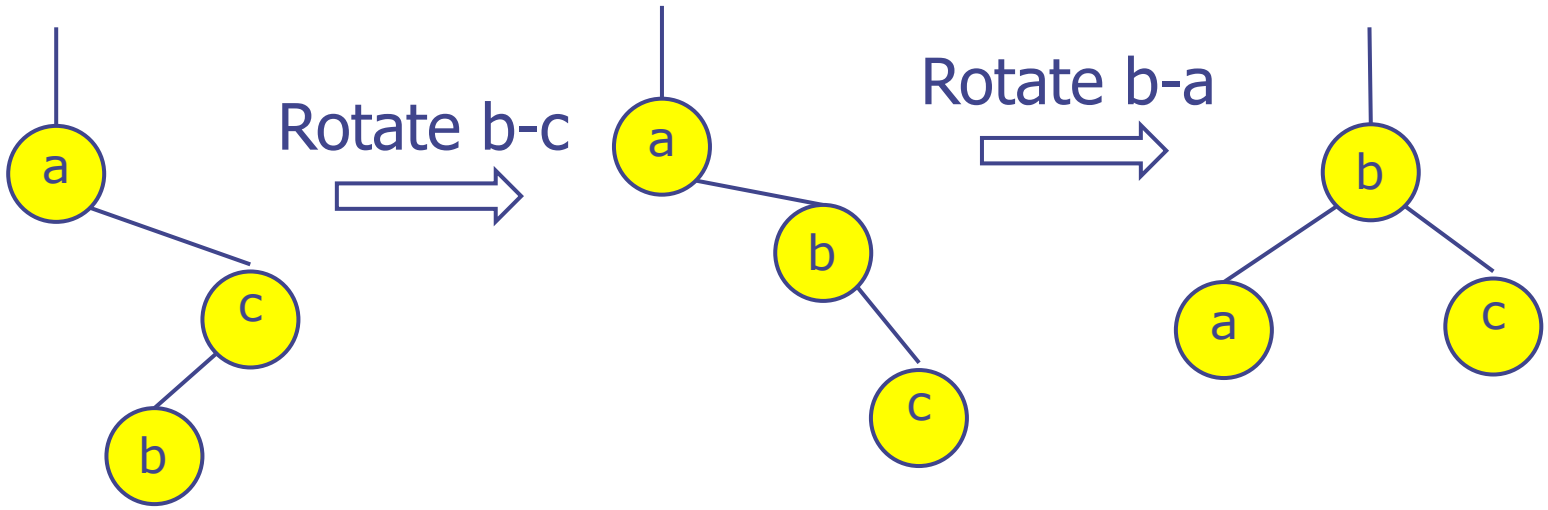
- Worst case: a chain, with a < b < c
- Rotation on the edge a-b
  - Pushes node a down, **pulls node b up** to make it the new root
  - Balances the tree
  - reduces the height from 2 to 1
- "Raising b" was the good choice because it is the median value and so should be the root

Harder Example: Single rotation fails



- "Oscillating Chain", with a < b < c
- Rotation on the edge a-c
  - Pushes node a down, pulls node c up to make it the new root
  - No effect on the height
  - Still have "oscillating chain"
  - NO USEFUL IMPROVEMENT

Specific Example: Double works



- Rotation on the edge b-c
  - Pushes node c down, pulls node b.
  - No immediate effect on the height, but now have a "straight chain"
- Then can rotate on edge a-b
  - Pulls b up again, to make it the root
  - Height is now reduced
- The median node is 'b' and so it should be at the root – so do rotations to "pull it up" (like 'up-heap') – depth of b needs to drop from 2 to 0
  - Each rotation can only move 'b' up by 1, hence need two rotations
- Coordinated "Double rotation" is sometimes needed !

Using rotations

- The goal is then to control the usage of rotations to reduce the overall height of the tree
  - Single rotations sometimes help
  - Sometimes need "double rotations" – a "coordinated pair of single rotations"
- In small examples, we can do this "by inspection"
- In general, need to be able to select rotations algorithmically

Solution: Self-Balancing

Goal of "Self-Balancing":

- Constantly do a "local restructuring" of the trees:
- Keep the trees height balanced so that the height is logarithmic in the size
- Only do rotations along the paths encountered during the BST operations
- Performance always logarithmic.

AVL Trees [[Not assessed]]

- AVL (Adelson-Velskii & Landis) trees are binary search trees where nodes also have additional information: the difference in depth between their right and left subtrees (*balance factor*). First example of balanced trees.
- For each node, the "balance factor" of that node is
  height(right subtree) – height(left subtree)
- In an AVL tree the balance of **every** node is allowed to be **only** 0,1 or -1.
- AVL trees do dynamic self-balancing in O(log n) time

Top down and bottom up insertion [[Not assessed]]

- Top down insertion algorithms make changes to the tree (necessary to keep the tree balanced) as they search for the place to insert the item. They make one pass through the tree.
- Bottom up: first insert the item, and then work back through the tree making changes. Less efficient because make two passes through the tree.

Examples [Not assessed]

- Bottom up: AVL trees
  - Details not required this session!
- Top down: 2-4 trees & red-black trees
  - Details not required this session!

But should know that they are designed to help keep the trees balanced in order that the height stays small.
They guarantee O(log n) – unlike Hashmaps

Minimal Expectations

- The definitions of (single) rotations
- Examples of their usage to reduce the height, in some small(ish) trees
- Manually selecting the rotations

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054 ADE
Shortest Paths:
Floyd-Warshall (FW)

All-Pairs Shortest Paths

- Suppose that wanted to find the SP between **all** pairs of start and end nodes
  - Dijkstra's algorithm finds all shortest path costs from one given starting node
  - [Not assessed] The complexity of Dijkstra is O(|V| * (|V| * log(|V|) +|E|))
  - We could run Dijkstra from every node
  - Would then be a factor of O(|V|) worse than Dijkstra, e.g. O(|V| * (|V| * log(|V|) +|E|))
- But may be better to run a specific algorithm
  - We will do the "Floyd-Warshall" (FW) algorithm

FW All-Pairs SPs

- The basic method has similarities to the methods for "change giving" in an earlier lecture
  - "Build best answers for a set of coins, and then add the effects of coins one at a time"
- "Build the optimal answers using a subset of the nodes. Then add the effects of other nodes one at a time"

FW All-Pairs SPs: data structure

Main data structure definition:

d(i,j,k) =

shortest distance between nodes i and j,

but using only the nodes {1,…,k} as potential allowed intermediary points

- E.g. d(2,5,3) =
  "shortest distance from n2 to n5 using only {n1,n2,n3} as potential intermediate points'."
  - Usage of all or any of these intermediate points is not forced, but other points, such as n4 …, cannot be used

FW: Initialisation of data structure

Initialisation:

d(i,j,0) = best distance between nodes i and j, but not using any intermediate nodes,

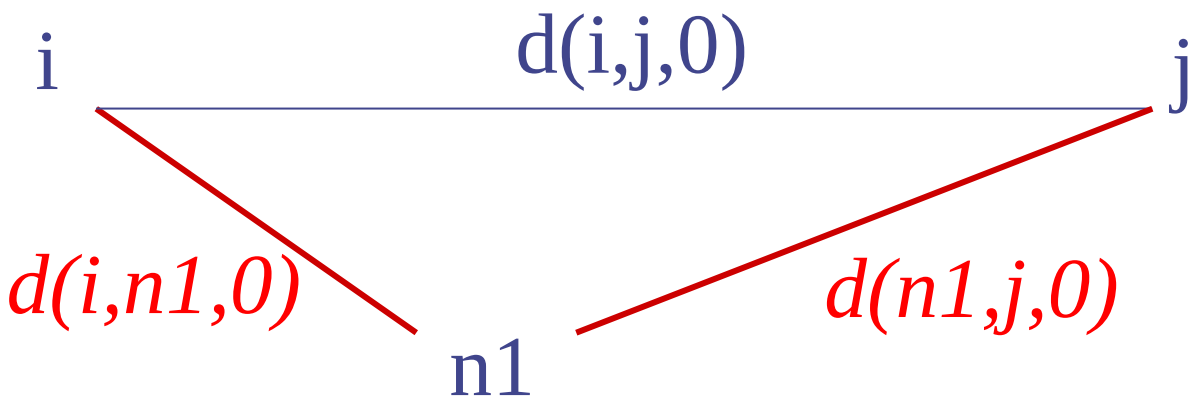
so only using a single edge, Hence

d(i,j,0) = w(i,j) if there is an edge i to j
= Inf otherwise

where Inf is 'infinity', and could be implemented as a special value, or a number large enough to act as infinity (bigger than all other numbers encountered), etc.

FW All-Pairs SPs

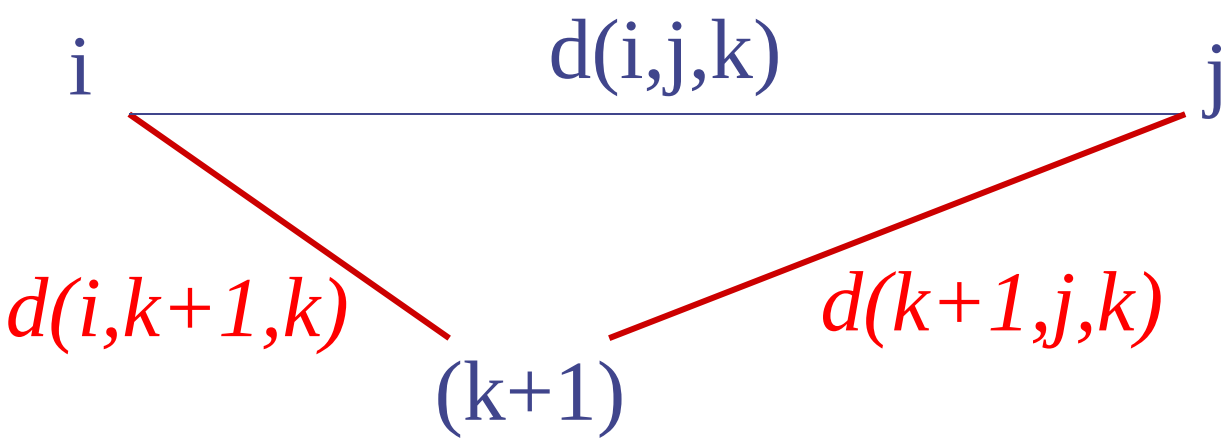
- Now suppose that we add the node 'n1' to the set of nodes that can be intermediates, i.e. consider k = 1
- Best path is now the best of "either direct, or via n1."
- d(i,j,1) = min (d(i,j,0), w(i,n1) + w(n1,j))
  = min (d(i,j,0), d(i,n1,0) + d(n1,j,0))



FW All-Pairs SPs

- Now suppose that we add the new node "(k+1)" to the set of "via nodes" that can be intermediates, but have already considered k of them
- Best path is now either direct using only the k 'via nodes' already accounted for, or else also via node 'k+1' (and using the previous k via's)

$$d(i,j,k+1) = \min (d(i,j,k),\ \ d(i,k+1,k) + d(k+1,j,k))$$



FW equations

$d(i,j,0) = w(i,j)$ if there is an edge i to j

$\quad\quad\quad = \text{Inf}$ otherwise

$$d(i,j,k+1) = \min\left(d(i,j,k),\ d(i,k+1,k) + d(k+1,j,k) \right)$$

Note:

- The rhs depends only on d(-,-,k)
- The lhs gives d(-,-,k+1)
- So solve by "start at k=0 and iteratively increase k"

(These are an example of "Belmann Equations" and can be solved using DP.)

FW equations – Comment

k is a dummy variable, and so we could equally well write

$$d(i,j,k) = \min\left(d(i,j,k-1),\ d(i,k,k-1) + d(k,j,k-1) \right)$$

Of course, in practice, when implementing, then have to be more careful about the range of k. So that it correctly captures the idea of

- Start with no "via nodes" allowed
- Add "via nodes" one at a time

Self-edges?

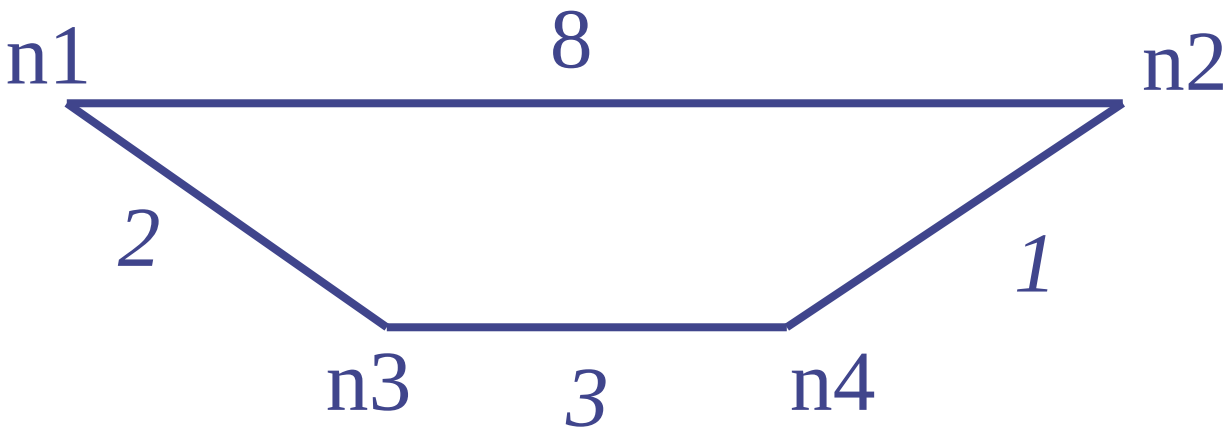
- The FW format does not in itself make any assumptions about d(i,i)
  - i.e. whether "self-edges" are always assumed to exist.

- However, we will assume that

$$d(\ i,\ i\ ) = 0 \quad \text{forall } i$$

  - that is, travelling from a node A to itself is possible at cost 0.
  - This is for simplicity, and also makes it compatible with Dijkstra which (typically) has the same assumption

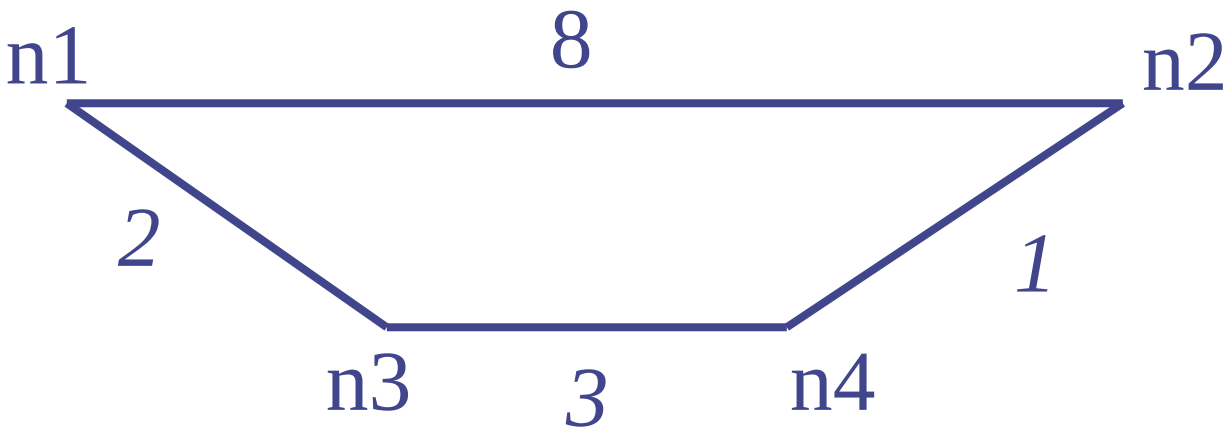
FW example : d(i,j,0)



- Initial d(i,j,0)
  - direct edges only
- Edges are undirected, hence matrix is symmetric

d(i,j,0)	n1	n2	n3	n4
n1	0	8	2	Inf
n2	8	0	Inf	1
n3	2	Inf	0	3
n4	Inf	1	3	0

FW example : d(i,j,1), partial



Set of intermediates = { n1 }

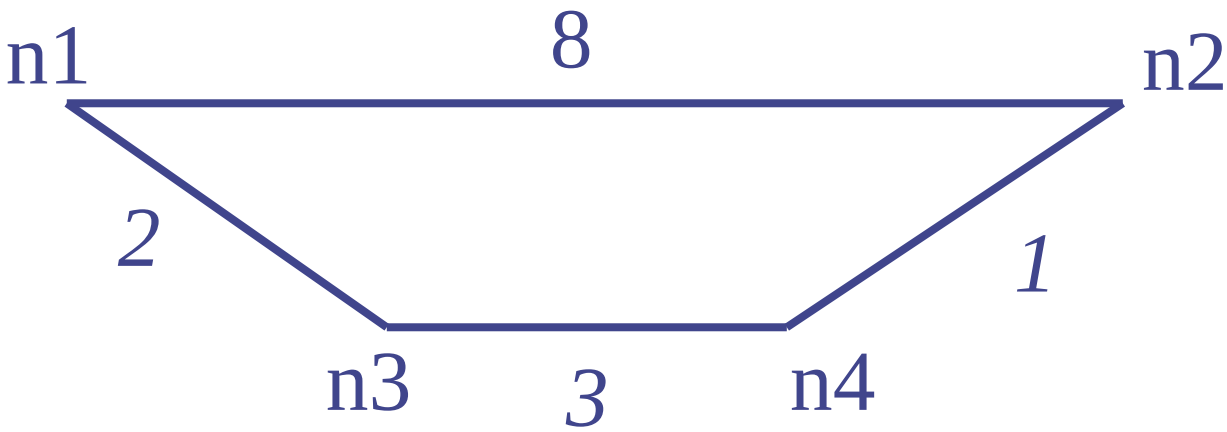
Example:

d(n3,n2,1)
= min(Inf, d(n3,n1,0) + d(n1,n2,0))
= min(Inf, 2+8) = 10

(We can "see" the optimal will be 3+1 via n4, but we have not considered n4 yet)

d(i,j,1)	n1	n2	n3	n4
n1	0	8	2	Inf
n2	8	0	~~Inf~~ **10**	1
n3	2	~~Inf~~ **10**	0	3
n4	Inf	1	3	0

FW example : d(i,j,1)



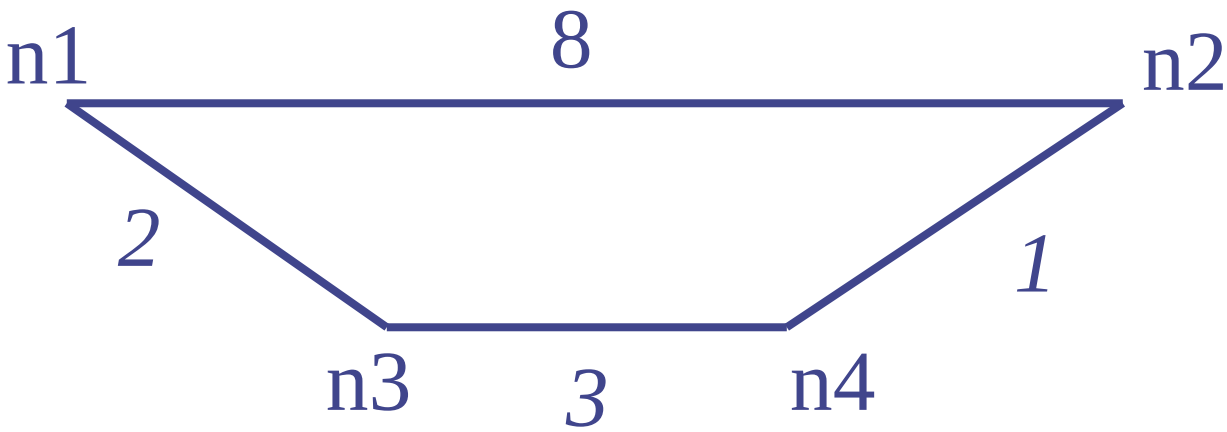
Set of intermediates = { n1 }

Filling in the other entries – check that no other entries change

The matrix is symmetric so will now just give the lower-triangle.

d(i,j,1)	n1	n2	n3	n4
n1	0			
n2	8	0		
n3	2	10	0	
n4	Inf	1	3	0

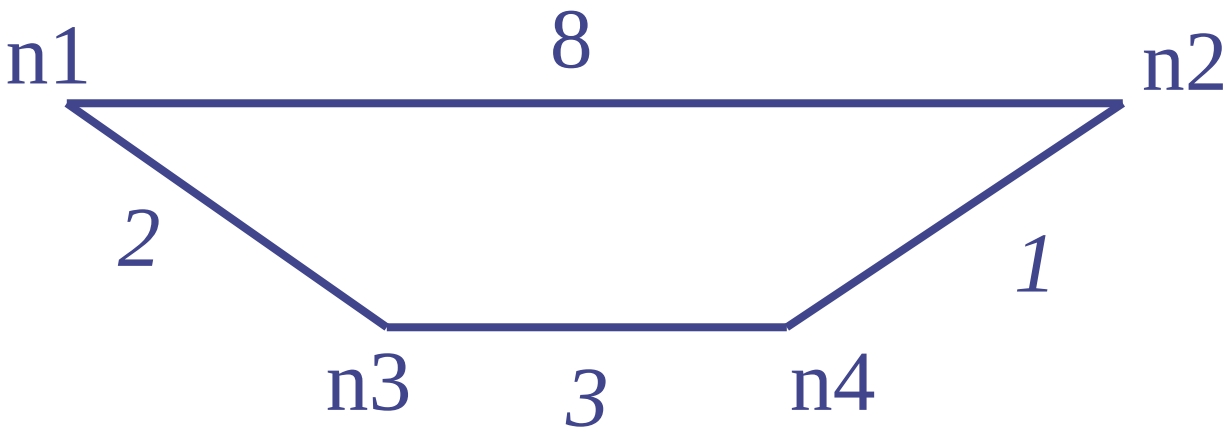
FW example : d(i,j,2)



Set of intermediates = { n1, n2 }
Node n2 is new, so e.g.
d(n4,n1,2)
= min(Inf, d(n4,n2,1) + d(n2,n1,1))
= min(Inf, 1+8) = 9

d(i,j,2)	n1	n2	n3	n4
n1	0			
n2	8	0		
n3	2	10	0	
n4	~~Inf~~ 9	1	3	0

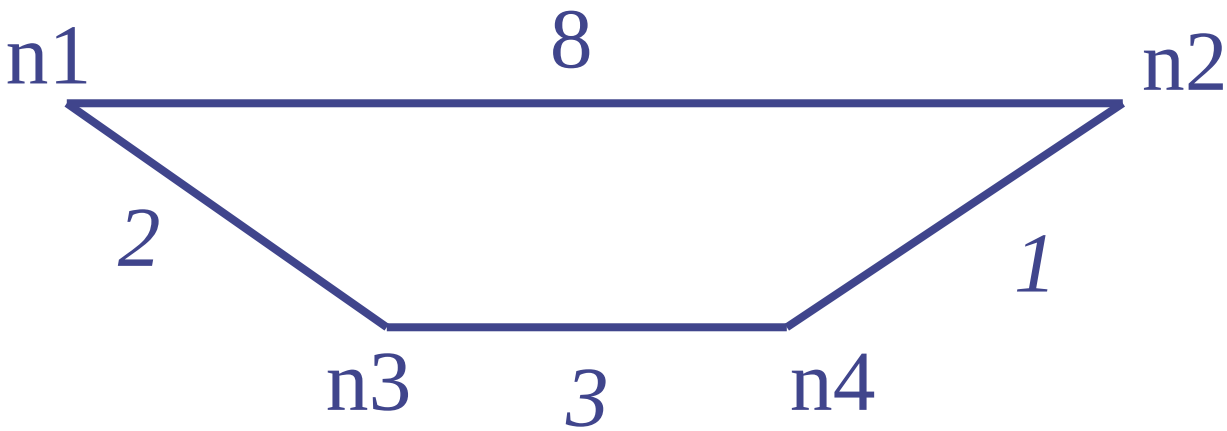
FW example : d(i,j,3)



Set of intermediates = { n1, n2, n3 }
Node n3 is new, so e.g.
d(n4,n1,3)
= min(9, d(n4,n3,2) + d(n3,n1,2))
= min(9, 3+2) = 5

d(i,j,3)	n1	n2	n3	n4
n1	0			
n2	8	0		
n3	2	10	0	
n4	~~9~~ 5	1	3	0

FW example : d(i,j,4)



Set of intermediates = {n1,n2,n3,n4}
Node n4 is new, so e.g.
d(n1,n2,4)
= min(8, d(n1,n4,3) + d(n4,n2,3))
= min(8, 5+1) = 6

d(i,j,4)	n1	n2	n3	n4
n1	0			
n2	~~8~~ 6	0		
n3	2	~~10~~ 4	0	
n4	5	1	3	0

Finished as all intermediates are now accounted for

FW code & complexity

The main loop after initialisation is:

```
foreach k = 1, …     (so "foreach n_k ∈ V")
 foreach i ∈ V
  foreach j ∈ V
     d(i,j,k+1) = min( d(i,j,k), d(i,k+1,k) + d(k+1,j,k) )
```

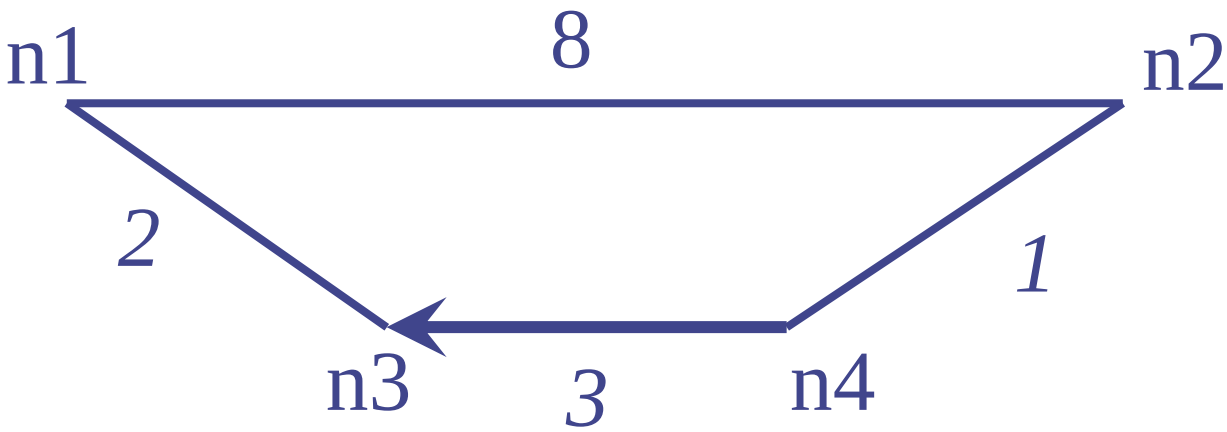
Note that is it is vital that 'k' is the outer loop.
Have 3 nested loops, of ranges |V|. **Hence is O($|V|^3$)**

(If the graph is sparse, then |E| << $|V|^2$, so then "all-starts Dijkstra" may be better. "<<" means "much less than".)

FW on digraphs

- FW also works the same on directed graphs
  - The initial matrix d(i,j,0) need not be symmetric, but then the remaining calculations use exactly the same formulas

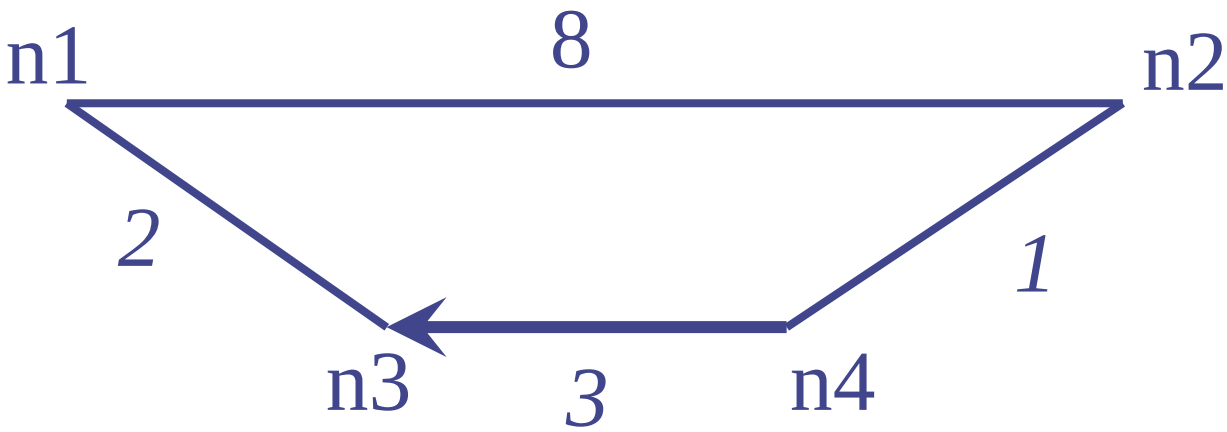
FW digraph example : d(i,j,0)



- Initial d(i,j,0)
  - direct edges only
  - edges with no arrows are bidirectional
- Matrix is no longer symmetric

d(i,j,0)	n1	n2	n3	n4
n1	0	8	2	Inf
n2	8	0	Inf	1
n3	2	Inf	0	**Inf**
n4	Inf	1	3	0

FW example : d(i,j,1), partial



Set of intermediates = { n1 }

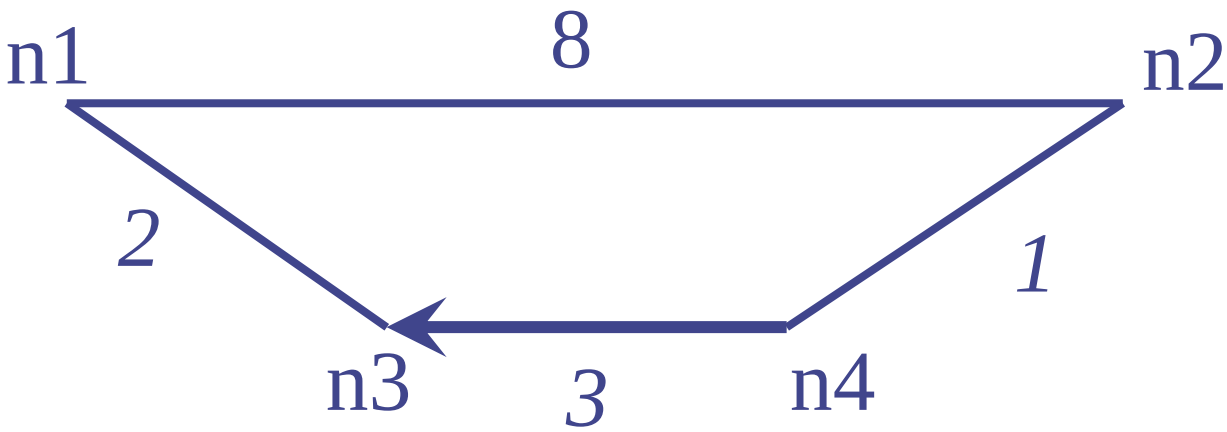
Example:

d(n2,n3,1)
= min(Inf, d(n2,n1,0) + d(n1,n3,0))
= min(Inf, 8+2) = 10

(We can "see" the optimal will be 3+1 via n4, but we have not considered n4 yet)

d(i,j,1)	n1	n2	n3	n4
n1	0	8	2	Inf
n2	8	0	~~Inf~~ **10**	1
n3	2	~~Inf~~ **10**	0	Inf
n4	Inf	1	3	0

FW example : d(i,j,1)

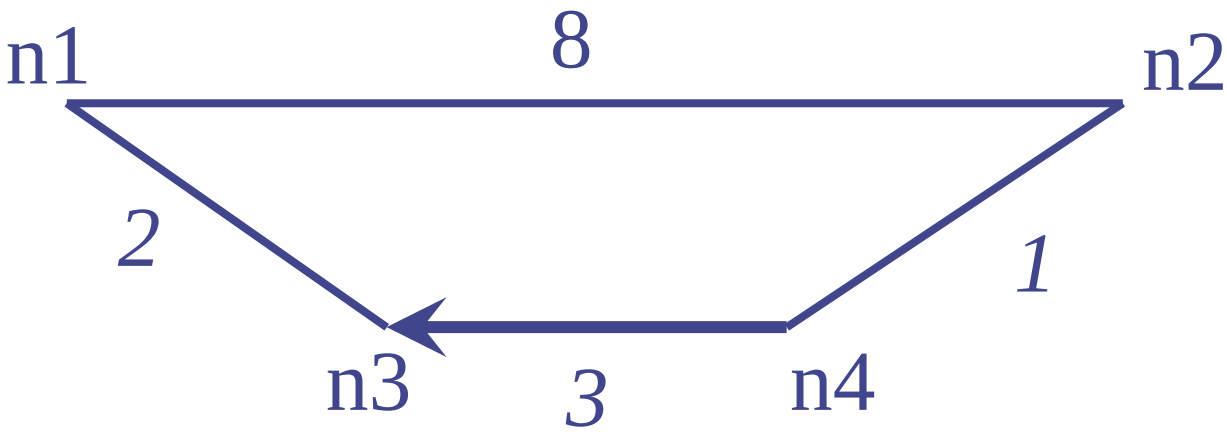


Set of intermediates = { n1 }

Filling in the other entries – check that no other entries change

d(i,j,1)	n1	n2	n3	n4
n1	0	8	2	Inf
n2	8	0	10	1
n3	2	10	0	Inf
n4	Inf	1	3	0

FW example : d(i,j,2)



Set of intermediates = { n1, n2 }

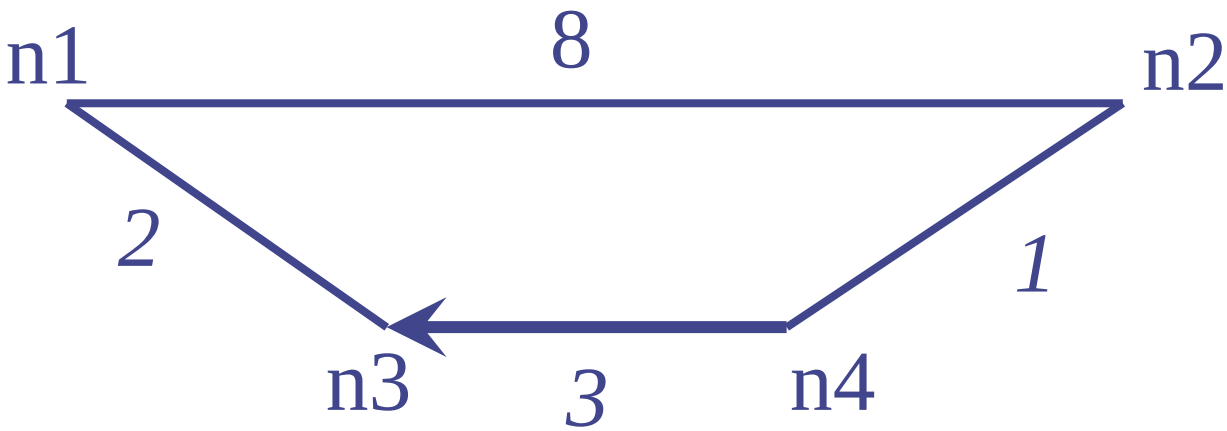
Node n2 is new, so e.g.

d(n4,n1,2)
= min(Inf, d(n4,n2,1) + d(n2,n1,1))
= min(Inf, 1+8) = 9 etc

d(n3,n4,2)
= min(Inf, d(n3,n2,1) + d(n2,n4,1))
= min(Inf, 10+1) = 11

d(i,j,2)	n1	n2	n3	n4
n1	0	8	2	~~Inf~~ 9
n2	8	0	10	1
n3	2	10	0	~~Inf~~ 11
n4	~~Inf~~ 9	1	3	0

FW example : d(i,j,3)

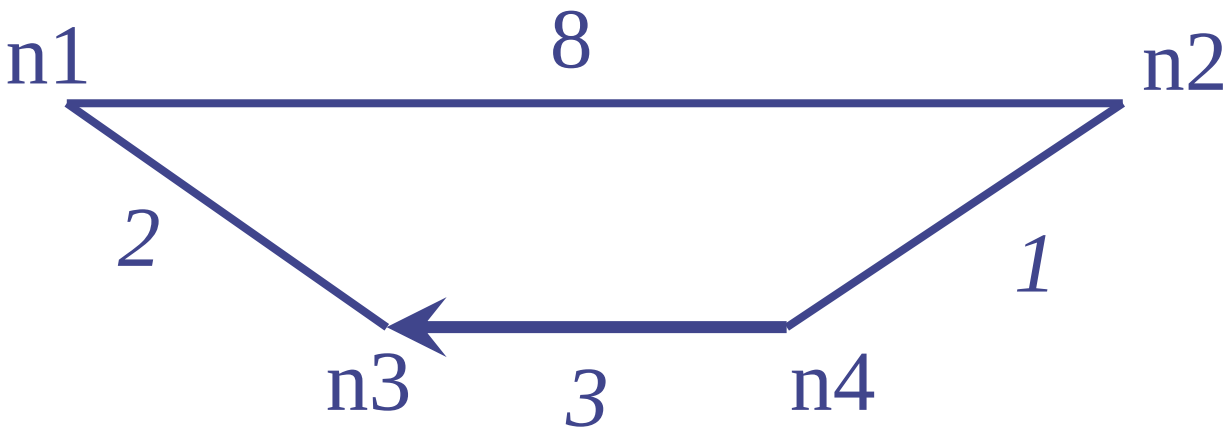


Set of intermediates = { n1, n2, n3 }
Node n3 is new, so e.g.
d(n4,n1,3)
= min(9, d(n4,n3,2) + d(n3,n1,2))
= min(9, 3+2) = 5

but
d(n1,n4,3)
= min(9, d(n1,n3,2) + d(n3,n4,2))
= min(9, 2+Inf) = 9

d(i,j,3)	n1	n2	n3	n4
n1	0	8	2	**9**
n2	8	0	10	1
n3	2	10	0	11
n4	~~9~~ **5**	1	3	0

FW example : d(i,j,4)



Set of intermediates = {n1,n2,n3,n4}
Node n4 is new, so e.g.
d(n2,n1,4)
= min(8, d(n2,n4,3) + d(n4,n1,3))
= min(8, 1 + 5) = 6

d(i,j,4)	n1	n2	n3	n4
n1	0	8	2	9
n2	**~~8~~ 6**	0	**~~10~~ 4**	1
n3	2	10	0	11
n4	5	1	3	0

Finished as all intermediates are now accounted for

FW with negative edges

- FW even works if some (directed) edge weights are negative
  - BUT it is essential that there are no cycles of total negative weight
  - Otherwise simply repeatedly following around the negative cycle may reduce lengths to be as negative as desired, so there is no shortest path

- Offline Exercise: repeat the previous example with d(n4,n3) = -1 (instead of +3)
  - The cycle n4-n3-n1-n2-n4 has weight -1+2+8+1=10 which is allowed

FW: Recall - key idea

Uses that SP does satisfy a nice decomposition of

- If P(A,B) is a shortest path, and goes via M then P(A,M) is optimal for A to M and P(M B) is optimal for M to B
- Hence uses a version of “dynamic programming”

Exercise

- You are **highly** recommended to
  - create some small to medium graphs (directed and undirected) and work through both (Dijkstra and FW) algorithms
  - repeat working examples until you **understand** them fully and can do it 'by hand' quickly and easily
    - Both algorithms are classics, and similar ideas appear in many other algorithms

Minimum Expectations

- Know and understand definition of all-pairs shortest path, and Floyd-Warshall algorithm
- Be able to apply FW, by hand, to small graphs