

Algorithmics

Lecture 1

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Schedule

Lecturer: Oana Andrei

- contact details on Moodle (F123), office hours Tuesday 12:30–13:30
- use Padlet to ask anonymous questions (check link on Moodle)
- use MS Teams to ask non-anonymous questions (check link on Moodle)
 - please try and take part in discussions on Teams and Padlet

Teaching sessions:

- 10:00 – 12:00
- usually, 1 hour lecture followed by 1 hour tutorial or lab except
 - tutorial exercises catch up on first Friday 7 March both hours
 - use that time to prepare for the class test
 - class test on Monday 17 March (see slide on assessment)
 - 2h lab sessions on Wednesday 19 March & Wednesday 26 March
 - 2h lectures on Tuesday 18 March
 - 2h tutorial on Thursday 20 March
- revision session(s) before exam

Set up

On Moodle you will find

- slides in pdf format
- tutorial sheets
- model solutions to tutorial exercises after/during the tutorial session
- suggestions for optional supplementary online resources

Attendance recording on Moodle GA General Information 2024–25

- with password for each day (as usual)

Assessment

- Moodle quizzes 5%: every week
- in-class test 15% : Monday 17 March
 - open books on Moodle with uploading your answers in a Word file
 - bring your laptop along
 - expected to take 1 h30
 - covers content from weeks 1 and 2
- assessed exercise 20%:
 - to be released on Tuesday 18 March
 - submission deadline on Sunday 6 April
 - the only programming-based assessment
- exam 60%:
 - date to be confirmed – April/May 2024
 - same setup as for semester 1 exams

**Use of Generative AI tools are not allowed
for any piece of summative assessment!**

Assessment

In-class test (15%) on Monday 17 March

- based on strings & text algorithms and graphs algorithms sections of the course
- “pen-and-paper” type questions

Quizzes (5%)

- quiz questions organized per sections
- available to answer **by end of Friday each week** when the respective section is covered
- best **15** marks for the quizzes will be aggregated to give a band
- affected questions will be voided (email me and do **not submit** good cause claims for quizzes)
- individual answers

Tutorials and labs

Tutorials

- start with tracing the execution of algorithms from lectures:
 - work in pairs for tracing algorithms
- attempt the rest of tutorial exercises on your own
- mostly pen-and-paper
- model solution provided and we review them in class

Labs

- first lab session on Wed 19 March to prepare you for the AE
- second lab session on Wed 26 March is for working on the AE
- ask us questions during tutorial sessions as well
- two graduate teaching assistants for both tutorials and labs:
 - Peace Ayegba p.ayegba.1@research.gla.ac.uk
 - Laura Larios-Jones l.larios-jones.1@research.gla.ac.uk

Independent study is essential

Don't expect to understand all during the lecture/contact hours!

- independent engagement with the content, tutorial exercises
- for each one hour of formal contact, you are expected to spend 2 hours of study
- you need to put some effort in solving tutorial exercises before checking the model solutions (learning how to solve, not how to read solutions!)

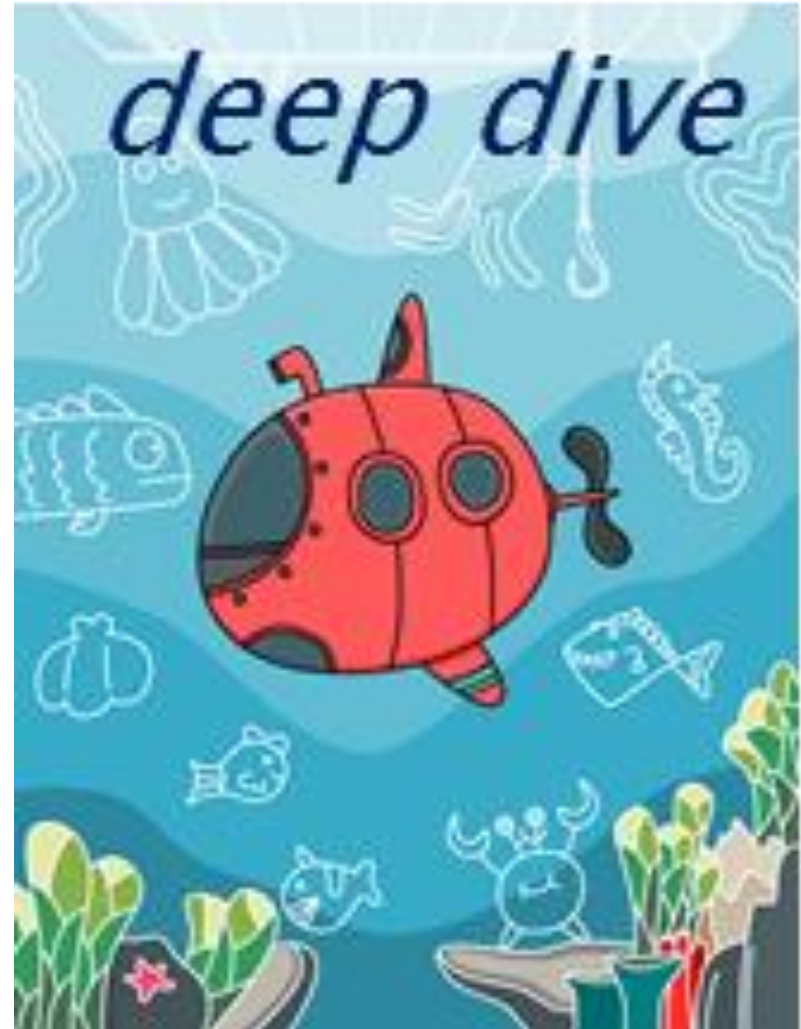
For example:

- review each concept covered in lectures, keep up with the pace
 - [optional] check supplementary references, e.g., books, Wikipedia, Youtube, online resources suggested on Moodle
 - e.g., <https://brilliant.org/computer-science/?subtopic=algorithms>
- follow the execution of the algorithm as presented in the slides
- trace each algorithm using the example input from the slides on your own then trace each algorithm using a new input

Deep dive topics – not examinable

Similar to what you have seen in
PA

Marked by the submarine
drawing



Who am I?

Lecturer since 2020

- Modelling Reactive Systems H/M
- GA Algorithmics
- GA Practical Algorithms (20%) in 2023–24
- Student project supervisor (L4, L5, MSc CS+/IT+, WPA3, WPA4/WPS, PGR)
- Course coordinator WPA4 and WPS in 2024–25 (and WPA3 in 2023–24)

Research and scholarship of learning and teaching

- 20+ years of research in formal methods (Romania, France, UK)
 - techniques for modelling complex systems as abstract mathematical entities amenable to a rigorous analysis of their properties
- 4+ years scholarship on learning and teaching
 - computer science curriculum, competency-based learning, and widening access to Higher Education in Computer Science & Software Engineering

Deputy Director of GA since Sept 2023, currently advisor for L4

MEd in Academic Practice student (!?)

Outline of course

Section 0: Quick recap on algorithm analysis – individual study

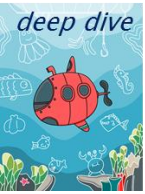
Section 1: Sorting algorithms

Section 2: Strings and text algorithms

Section 3: Graphs and graph algorithms

Section 4: An introduction to NP completeness

Section 5: A (very) brief introduction to computability



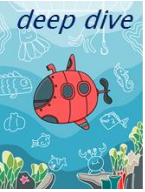
Why algorithmics? What is algorithmics?

Algorithms and data structures – see the PA course

Algorithmics:

- focuses on the design, analysis, and implementation of algorithms
- encompasses the theoretical and practical aspects of algorithms
 - **computational complexity** (such as time and memory space)
 - **correctness** (the algorithm produces the intended result for all valid inputs)
- aims to develop algorithms that are both efficient (minimal resource usage) and effective (correct and practical for the problem at hand), drawing from mathematical and logical principles to rigorously define and analyse algorithmic processes

* –ics (suffix): denoting arts or sciences, branches of study or action



Getting more epistemological

Computer science vs. computing science

- Why not submarine science? Or dishwasher science?

The term "Computing Science" might be preferred in some academic or professional contexts

- emphasis on the scientific and theoretical aspects of computing as a discipline
- beyond just the physical computers themselves
- focus on the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application

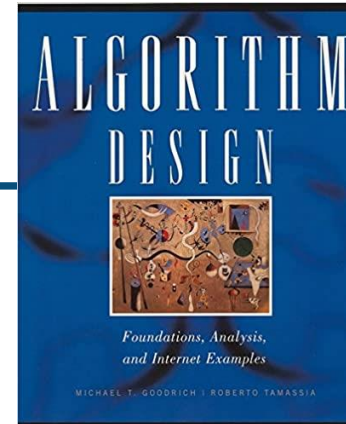
Informatics is a broader and more inclusive term

- the study, design, and application of computer and information systems
- integrates aspects from CS, information science, and cognitive science

Resources

Lecture handouts on moodle

- examples from lectures also on moodle



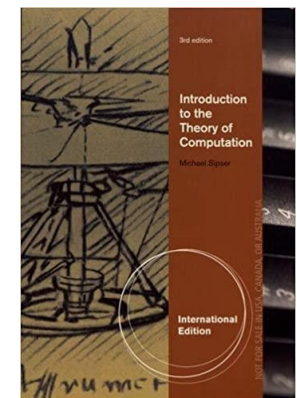
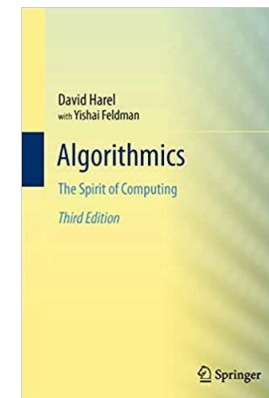
Course texts on moodle (recommended, not required)

- M.T. Goodrich & R. Tamassia. **Algorithm Design: Foundations, Analysis, and Internet Examples**. Wiley, 2002
- D. Harel & Y. Feldman. **Algorithmics: the Spirit of Computing**. Addison Wesley, 2004 (also earlier 1992 edition by D. Harel)
- M. Sipser. **Introduction to the Theory of Computation**. Course Technology, 2006

Tutorial exercises on moodle

- model solutions files after tutorial session

Lab material on moodle



Background

Basic knowledge of Java from HTLANL

- Java and a Java-style pseudocode is used to describe algorithms
- some algorithms are partially or fully implemented in Java

From Practical Algorithms

- fundamental concepts of **graphs**
- terminology and notation of **sets**, **relations** and **functions**
- basic mathematical **reasoning** and **proof**
- concept of an abstract data type (**ADT**)
- common **data structures** (e.g., lists, arrays and binary trees)
- basics of **algorithm analysis** (e.g., time complexity and big-O notation)
- common **searching** algorithms (e.g., linear search and binary search)
- common **sorting** algorithms (e.g., insertion sort and merge sort)

Outline of course

Section 0: Quick recap on algorithm analysis – individual study

- see slides on Moodle
- check also slides on some basic data structures

Section 1: Sorting algorithms

Section 2: Strings and text algorithms

Section 3: Graphs and graph algorithms

Section 4: An introduction to NP completeness

Section 5: A (very) brief introduction to computability

Section 1 – Sorting

Very quick recap

Comparison based–sorting

Radix sorting

Tries data structure

Sorting – Very quick recap

Naïve sorting algorithms: $O(n^2)$ in the worst/average case

- Selection-sort, Insertion-sort, Bubble-sort

Clever sorting algorithms: $O(n \log n)$ in the worst/average case

- Merge-sort, Heap-sort

The fastest sorting algorithm in practice is **Quicksort**

- $O(n \log n)$ on average
- but no better than $O(n^2)$ in the worst case (unless a clever variant is used)

Question: can we come up with a sorting algorithm that is better than $O(n \log n)$ in the worst case?

- for example, a $O(n)$ algorithm
- i.e., we are looking for a lower bound on comparison-based sorting

Section 1 – Sorting

Quick recap

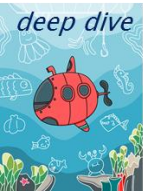
Comparison based–sorting

Radix sorting

Tries data structure

Comparison-based sorting lower bound

Claim: no sorting algorithm that is based on pairwise comparison of values can be better than $O(n \log n)$

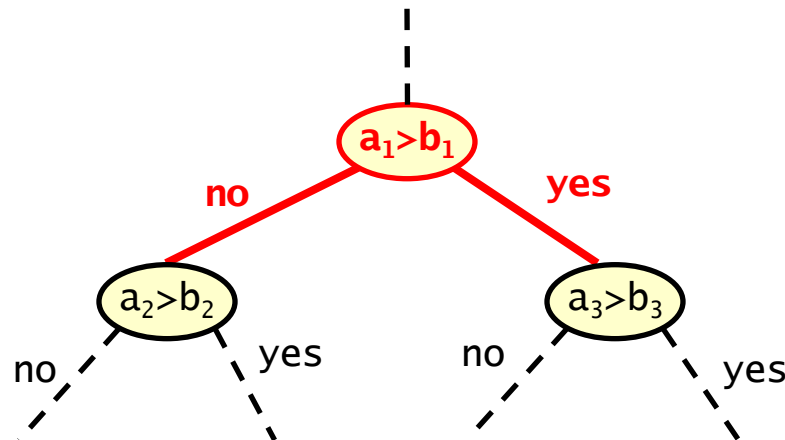


Comparison-based sorting lower bound

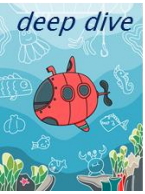
Claim: no sorting algorithm that is based on pairwise comparison of values can be better than $O(n \log n)$

Proof: describe the algorithm by a **decision tree** (binary tree)

- each node represents a comparison between two elements
- path branches left or right depending on the outcome of the comparison



We want to derive a lower bound on the number of comparisons.

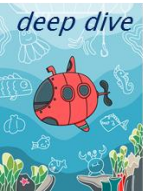


Comparison-based sorting lower bound

Claim: no sorting algorithm that is based on pairwise comparison of values can be better than $O(n \log n)$

Proof: describe the algorithm by a **decision tree** (binary tree)

- each internal node represents a comparison between two elements
- path branches left or right depending on the outcome of the comparison
- an execution of the algorithm is a path from the root to a leaf node
- the number of leaf nodes in the tree must be at least the number of ‘outcomes’ of the algorithm
 - let S be the sequence of elements to sort
 - we associate with each leaf node v the set of permutations of S that cause the sorting algorithm to end up in v
 - each leaf node represents the sequence of comparisons for at most one permutation of S

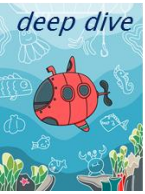


Comparison-based sorting lower bound

Claim: no sorting algorithm that is based on pairwise comparison of values can be better than $O(n \log n)$

Proof: describe the algorithm by a **decision tree** (binary tree)

- each internal node represents a comparison between two elements
- path branches left or right depending on the outcome of the comparison
- an execution of the algorithm is a path from the root to a leaf node
- the number of leaf nodes in the tree must be at least the number of ‘outcomes’ of the algorithm
- therefore, number of leaf nodes equals the possible orderings of n items
- that is there are least $n!$ leaf nodes (remember permutations from PA)



Comparison-based sorting lower bound

We have shown the decision tree has at least $n!$ leaf nodes

The worst-case complexity of the algorithm is no better than $O(h)$

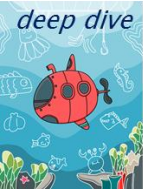
- where h is the height of the tree
- an execution is a path from the root node to a leaf node
- we perform an operation on each branch node so h operations in the worst case

A decision tree is a binary tree (two branches ‘yes’ and ‘no’)

and hence the number of leaf nodes is less than or equal to $2^{h+1}-1$

- a binary tree of height h has at most $2^{h+1}-1$ nodes (proof by induction)

Combining these properties it follows that $n! \leq 2^{h+1}-1 \leq 2^{h+1}$



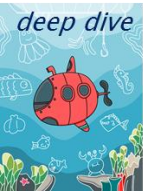
Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

$$h+1 \geq \log_2(n!)$$



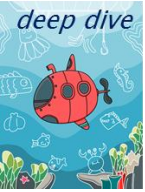
Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

$$\begin{aligned} h+1 &\geq \log_2(n!) \\ &> \log_2(n/2)^{n/2} && (\text{since } n! > (n/2)^{n/2}) \end{aligned}$$



Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

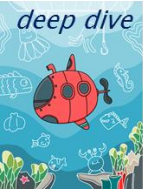
$$h+1 \geq \log_2(n!)$$

$$> \log_2(n/2)^{n/2}$$

$$= (n/2) \log_2(n/2)$$

$$\text{(since } n! > (n/2)^{n/2} \text{)}$$

$$\text{(since } \log a^b = b \log a \text{)}$$



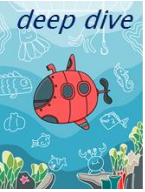
Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

$$\begin{aligned} h+1 &\geq \log_2(n!) \\ &> \log_2(n/2)^{n/2} && \text{(since } n! > (n/2)^{n/2} \text{)} \\ &= (n/2) \log_2(n/2) && \text{(since } \log a^b = b \log a \text{)} \\ &= (n/2) \log_2 n - (n/2) \log_2 2 && \text{(since } \log a/b = \log a - \log b \text{)} \end{aligned}$$



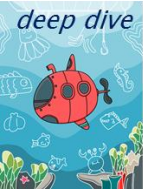
Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

$$\begin{aligned} h+1 &\geq \log_2(n!) \\ &> \log_2(n/2)^{n/2} && \text{(since } n! > (n/2)^{n/2} \text{)} \\ &= (n/2) \log_2(n/2) && \text{(since } \log a^b = b \log a \text{)} \\ &= (n/2) \log_2 n - (n/2) \log_2 2 && \text{(since } \log a/b = \log a - \log b \text{)} \\ &= (n/2) \log_2 n - n/2 && \text{(since } \log_a a = 1 \text{)} \end{aligned}$$



Comparison-based sorting lower bound

We have shown: complexity is no better than $O(h)$ and $2^{h+1} \geq n!$

- h is the height of the decision tree
- n is the number of items to be sorted

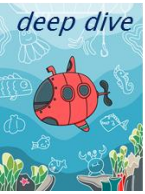
Taking \log_2 of both sides of $2^{h+1} \geq n!$ yields:

$$\begin{aligned} h+1 &\geq \log_2(n!) \\ &> \log_2(n/2)^{n/2} && \text{(since } n! > (n/2)^{n/2} \text{)} \\ &= (n/2) \log_2(n/2) && \text{(since } \log a^b = b \log a \text{)} \\ &= (n/2) \log_2 n - (n/2) \log_2 2 && \text{(since } \log a/b = \log a - \log b \text{)} \\ &= (n/2) \log_2 n - n/2 && \text{(since } \log_a a = 1 \text{)} \end{aligned}$$

Giving a complexity of at least $O(n \log n)$ as required

Comparison-based searching

We can also show that no comparison-based algorithm can guarantee to **search** a sequence of n elements for a particular value in better than **$O(\log n)$** time.



Comparison-based searching

We can also show that no comparison-based algorithm can guarantee to **search** a sequence of n elements for a particular value in better than **$O(\log n)$** time.

Similar proof:

- any comparison algorithm can be represented by a **decision tree** (binary tree)
- the branch nodes represent the comparisons performed during the search
- the leaf nodes represent the outcome of the search

Section 1 – Sorting

Quick Recap

Comparison based–sorting

Radix sorting

Tries data structure

Radix sorting

No sorting algorithm that is based on pairwise comparisons can be better than $O(n \log n)$ in the worst case

- therefore, to improve on this worst case bound, we have to devise a method based on something other than comparisons

Radix sort uses a different approach to achieve an $O(n)$ complexity

- the algorithm exploits the structure of the items being sorted, so may be less versatile
- in practice, it is faster than $O(n \log n)$ algorithms only for very large n

Radix sorting – motivation

Efficient data processing and organisation

- **Context:** Developing a system for processing a massive database of user-generated content (e.g., videos) tagged with numerical identifiers
- **Challenge:** Sorting these items efficiently to support features like chronological timelines, content aggregation, and personalised feeds
 - speed and efficiency in sorting vast volumes of data are critical for real-time processing and user satisfaction
 - traditional sorting algorithms (e.g., quicksort, merge sort) may not meet performance requirements under peak load
- **Solution:** use Radix Sort for its ability to sort data in linear time
 - crucial for maintaining a responsive and scalable system
 - particularly effective for large datasets with uniform key distribution, making it ideal for big data applications

Radix sorting

Assume items to sort can be treated as bit-sequences of length m

- let b be a chosen factor of m (b divides m)
- b and m are constants for any particular instance
- choosing b :
 - $b=1$ leads to m iterations and 2 buckets
 - $b=m$ leads to 1 iteration and 2^m buckets

Radix sorting – Algorithm

Each item has bit positions labelled $0, 1, \dots, m-1$

- bit 0 being the least significant

$\text{item} = 0010100100110001$

Radix sorting – Algorithm

Each item has bit positions labelled **0, 1, ..., m-1**

- bit **0** being the least significant (i.e. the right-most)

item = 001010010011000**1**



least significant (labelled 0)

Radix sorting – Algorithm

Each item has bit positions labelled $0, 1, \dots, m-1$

- bit 0 being the least significant (i.e. the right-most)

The algorithm uses m/b iterations

- in each iteration the items are distributed into 2^b buckets
- a bucket is just a list
- the buckets are labelled $0, 1, \dots, 2^b-1$ (or, equivalently, $\overbrace{00\dots0}^{\text{length } b}$ to $\overbrace{11\dots1}^{\text{length } b}$)
- during the i^{th} iteration an item is placed in the bucket corresponding to the integer represented by the bits in positions $b \times i - 1, \dots, b \times (i-1)$
 - e.g. for $b=4$ and $i=2$

item = 0010100100110001

Radix sorting – Algorithm

Each item has bit positions labelled $0, 1, \dots, m-1$

- bit 0 being the least significant (i.e., the right-most)

The algorithm uses m/b iterations

- in each iteration the items are distributed into 2^b buckets
- a bucket is just a list
- the buckets are labelled $0, 1, \dots, 2^b-1$ (or, equivalently, $\overbrace{00\dots0}^{\text{length } b}$ to $\overbrace{11\dots1}^{\text{length } b}$)
- during the i^{th} iteration an item is placed in the bucket corresponding to the integer represented by the bits in positions $b \times i - 1, \dots, b \times (i-1)$
 - e.g. for $b=4$ and $i=2$, consider bits in position $7, \dots, 4$

item = 0010100100110001

- 0011 represents the integer 3
- so item is placed in the bucket labelled 3 (or, equivalently, 0011)

Radix sorting – Algorithm

Each item has bit positions labelled $0, 1, \dots, m-1$

- bit 0 being the least significant (i.e. the right-most)

The algorithm uses m/b iterations

- in each iteration the items are distributed into 2^b buckets
- a bucket is just a list
- the buckets are labelled $0, 1, \dots, 2^b-1$ (or, equivalently, $\overbrace{00\dots0}^{\text{length } b}$ to $\overbrace{11\dots1}^{\text{length } b}$)
- during the i^{th} iteration an item is placed in the bucket corresponding to the integer represented by the bits in positions $b \times i - 1, \dots, b \times (i-1)$
- at the end of an iteration the buckets are concatenated to give a new sequence which will be used as the starting point of the next iteration

Radix sorting – Example

Suppose we want to sort the following sequence with Radix sort

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings are given by

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

- items have bit positions 0, ..., 5, hence $m=6$
- b must be a factor of m , so say we choose $b=2$

This means in Radix sort we have:

- $2^b=2^2=4$ buckets labelled 0, 1, 2, 3 (or equivalently 00, 01, 10, 11)
and $m/b = 3$ iterations are required

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

An iteration of radix (after substituting **b** with **2**)

- items are distributed into **4 buckets** (a bucket is just a list)
- during the **i^{th}** iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions **$2 \times i - 1, \dots, 2 \times (i - 1)$**
- buckets are concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into **4 buckets** (a bucket is just a list)
- during the **1st** iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions **$2 \times 1 - 1, \dots, 2 \times (1 - 1)$**
- buckets are concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets are concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets are concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00:

bucket 01:

bucket 10:

bucket 11:

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence:

Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,...,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

1st iteration:

bucket 00: 60

bucket 01: 5

bucket 10: 18 26 2

bucket 11: 15 43 27

new sequence: 60 5 18 26 2 15 43 27

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions $2 \times 2 - 1, \dots, 2 \times (2 - 1)$
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3, ..., 2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:

bucket 00:

bucket 01:

bucket 10:

bucket 11:

new sequence:

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	

Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27
----	---	----	----	---	----	----	----

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,...,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

2nd iteration:	
bucket 00:	18 2
bucket 01:	5
bucket 10:	26 43 27
bucket 11:	60 15
new sequence:	18 2 5 26 43 27 60 15

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions $2 \times 3 - 1, \dots, 2 \times (3 - 1)$
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5, ..., 4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,...,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:

bucket 00:

bucket 01:

bucket 10:

bucket 11:

sorted sequence:

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,...,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:	
bucket 00:	2 5 15
bucket 01:	18 26 27
bucket 10:	43
bucket 11:	60
sorted sequence:	

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,...,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:	
bucket 00:	2 5 15
bucket 01:	18 26 27
bucket 10:	43
bucket 11:	60
sorted sequence:	

Radix sorting – Example

New sequence

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,...,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:	
bucket 00:	2 5 15
bucket 01:	18 26 27
bucket 10:	43
bucket 11:	60
sorted sequence:	

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,...,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:	
bucket 00:	2 5 15
bucket 01:	18 26 27
bucket 10:	43
bucket 11:	60
sorted sequence:	

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5, ..., 4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:	
bucket 00:	2 5 15
bucket 01:	18 26 27
bucket 10:	43
bucket 11:	60
sorted sequence:	

Radix sorting – Example

New sequence:

18	2	5	26	43	27	60	15
----	---	---	----	----	----	----	----

Binary encodings:

18 = 010010	2 = 000010	5 = 000101	26 = 011010
43 = 101011	27 = 011011	60 = 111100	15 = 001111

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5, ..., 4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

3rd iteration:

bucket 00: 2 5 15

bucket 01: 18 26 27

bucket 10: 43

bucket 11: 60

sorted sequence: 2 5 15 18 26 27 43 60

Radix sorting – Q1 in Tutorial 1

Sequence:

15	43	5	27	60	18	26	2
----	----	---	----	----	----	----	---

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

Trace the radix sorting algorithm for $b=3$

- items are distributed into 2^3 buckets
- during the i^{th} iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions $2 \times i - 1, \dots, 2 \times (i - 1)$
- buckets are concatenated at the end of an iteration to give input sequence for the next iteration

Radix sorting – Pseudocode

```
// assume we have the following method which returns the value  
// represented by the b bits of x when starting at position pos  
private int bits(Item x, int b, int pos)  
  
// suppose that:  
//   a is the sequence to be sorted  
//   m is the number of bits in each item of the sequence a  
//   b is the 'block length' of radix sort  
  
int numIterations = m/b; // number of iterations required for sorting  
int numBuckets = (int) Math.pow(2, b); // number of buckets  
  
// represent sequence a to be sorted as an ArrayList of Items  
ArrayList<Item> a = new ArrayList<Item>();  
  
// represent the buckets as an array of ArrayLists  
ArrayList<Item>[] buckets = new ArrayList[numBuckets];  
for (int i=0; i<numBuckets; i++) buckets[i] = new ArrayList<Item>();
```

Radix sorting – Pseudocode

```
for (int i=1; i<=numIterations; i++){  
  
    // clear the buckets  
    for (int j=0; j<numBuckets; j++) buckets[j].clear();  
  
    // distribute the items (in order from the sequence a)  
    for (Item x : a){  
        // find the value of the b bits starting from position (i-1)*b in x  
        int k = bits(x, b, (i-1)*b); // find the correct bucket for item x  
        buckets[k].add(x); // add item to this bucket  
    }  
  
    a.clear(); // clear the sequence  
  
    // concatenate the buckets (in sequence) to form the new sequence  
    for (j=0; j<numBuckets; j++) a.addAll(buckets[j]);  
  
}
```


Radix sorting – Correctness

Let x and y be two items in a sequence to be ordered, with $x < y$

- need to show that x precedes y in the final sequence

Suppose j is the last iteration for which relevant bits of x and y differ

- since $x < y$ and j is the last iteration that x and y differ
the relevant bits of x must be smaller than those of y
- therefore x goes into an ‘earlier’ bucket than y
and hence x precedes y in the sequence after this iteration
- since j is the last iteration where bits differ:
 - * in all later iterations x and y go in the same bucket
 - * so their relative order is unchanged

Radix sorting – Complexity

Number of iterations is m/b and number of buckets is 2^b

During each of the m/b iterations

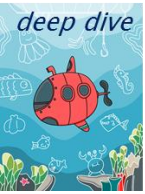
- the sequence is scanned and items are allocated buckets: $O(n)$ time
- buckets are concatenated: $O(2^b)$ time

Therefore the overall complexity is $O(m/b \cdot (n + 2^b))$

- this is $O(n)$, since m and b are constants

Time–space trade–off

- the larger the value of b , the smaller the multiplicative constant (m/b) in the complexity function and so the faster the algorithm will become
- however an array of size 2^b is required for the buckets
therefore increasing b will increase the space requirements



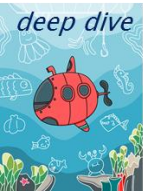
Radix sorting – Some history

Used for tabulating machines with punch cards – Herman Hollerith

- in 1890 US census: 63M subjects, 3 months instead of 10 years
- the Hollerith electric tabulating machine – PhD at Columbia University
- in 1896 created the Tabulating Machine Company
- in 1911 merging into CTR Company, renamed **IBM** in 1924
- LSD algorithm used for sorting punched cards in the 1920s (least significant digit), useful in accounting numbers (same length)

The Hollerith machine formed the basis of IBM's 1944 Automatic Sequence Controlled Calculator

- an electromechanical computer nicknamed as 'Babbage's dream come true'
- superseded by ENIAC (Electronic Numerical Integrator and Computer) – the first programmable, electronic, general-purpose digital computer; also Turing-complete



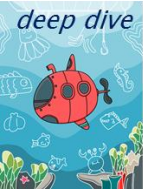
Radix sorting – LSD vs. MSD

Least significant digit (LSD)

- the right–most bit
- the Radix algorithm covered in the previous slides
- preserves the original/relative order of elements/records with equal keys (stable sorting)

Most significant digit (MSD)

- the left–most bit
- does not necessarily preserve the original order of duplicate elements
- useful for sorting elements in lexicographical order (strings or fixed–length integer representation)
 - for variable length integers, it would sort the natural numbers from 1 to 10 as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9



Hollerith 1890 Census Tabulator



The image shows [Herman Hollerith](#)'s 1890 tabulating machine (image from IBM; [CLICK HERE for a color photo](#)). The results of a tabulation are displayed on the clock-like dials. A sorter is on the right. On the tabletop below the dials are a Pantographic card punch (explained below) on left and the card reading station ("press") on the right, in which metal pins pass through the holes, making contact with little wells of mercury, completing an electrical circuit. All of these devices are fed manually, one card at a time, but the tabulator and sorter are electrically coupled.

<http://www.columbia.edu/cu/computinghistory/census-tabulator.html>

Section 1 – Sorting

Quick Recap

Comparison based–sorting

Radix sorting

Tries data structure

Tries (retrieval)

Binary search trees are comparison-based data structures

Tries are to binary trees as **Radixsort** is to comparison-based sorting

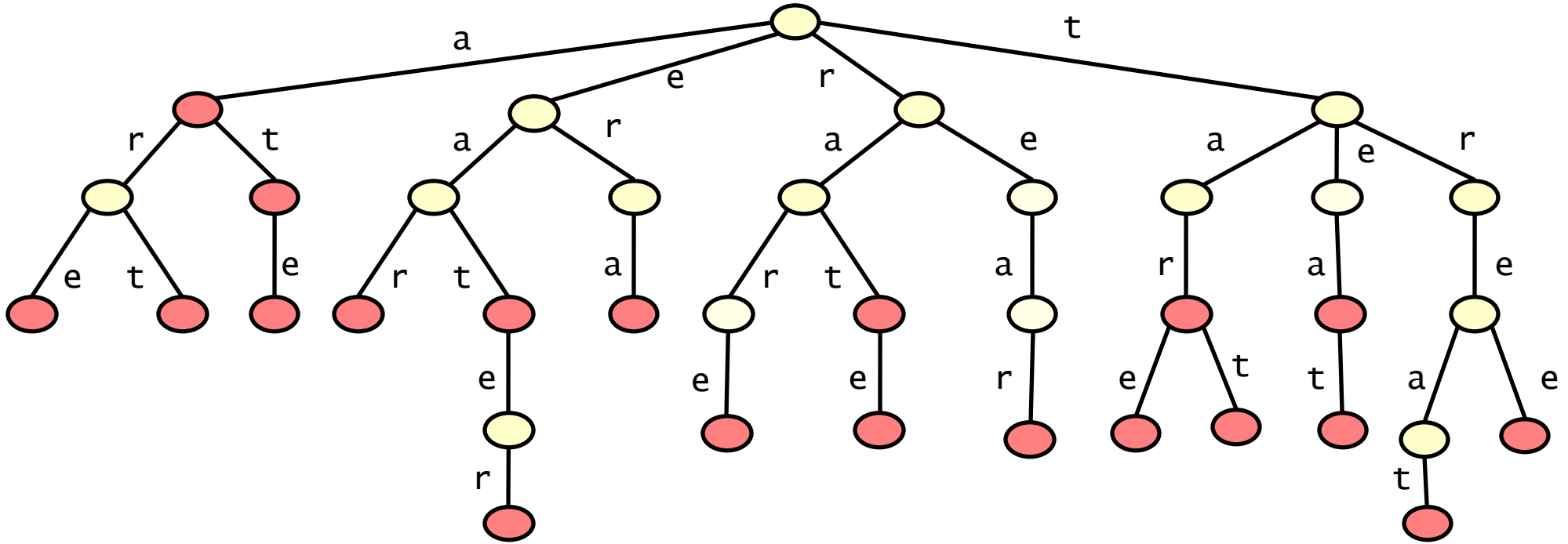
- stored items have a key value that is interpreted as a **sequence** of bits, or characters, ...
- there is a **multiway branch** at each node where each branch has an associated symbol and no two siblings have the same symbol
- the branch taken at level **i** during a search, is determined by the **i^{th}** element of the key value (**i^{th}** bit, **i^{th}** character, ...)
- tracing a path from the root to a node spells out the key value of the item



Example: use a trie to store items with a key value that is a **string**

- the words in a dictionary

Tries – Examples

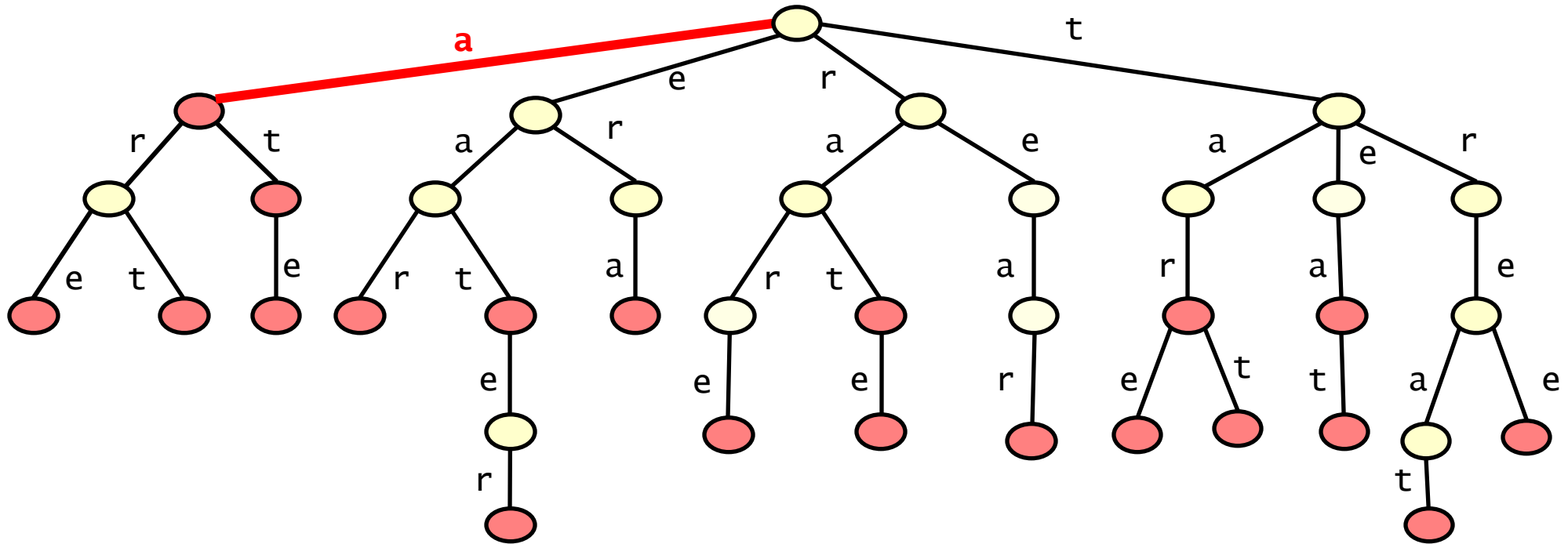
An example **trie** containing words from a **4** letter alphabet





- **Two kinds of nodes**
 -  nodes representing words
 -  internal/intermediate nodes

Tries – Examples

An example **trie** containing words from a **4** letter alphabet

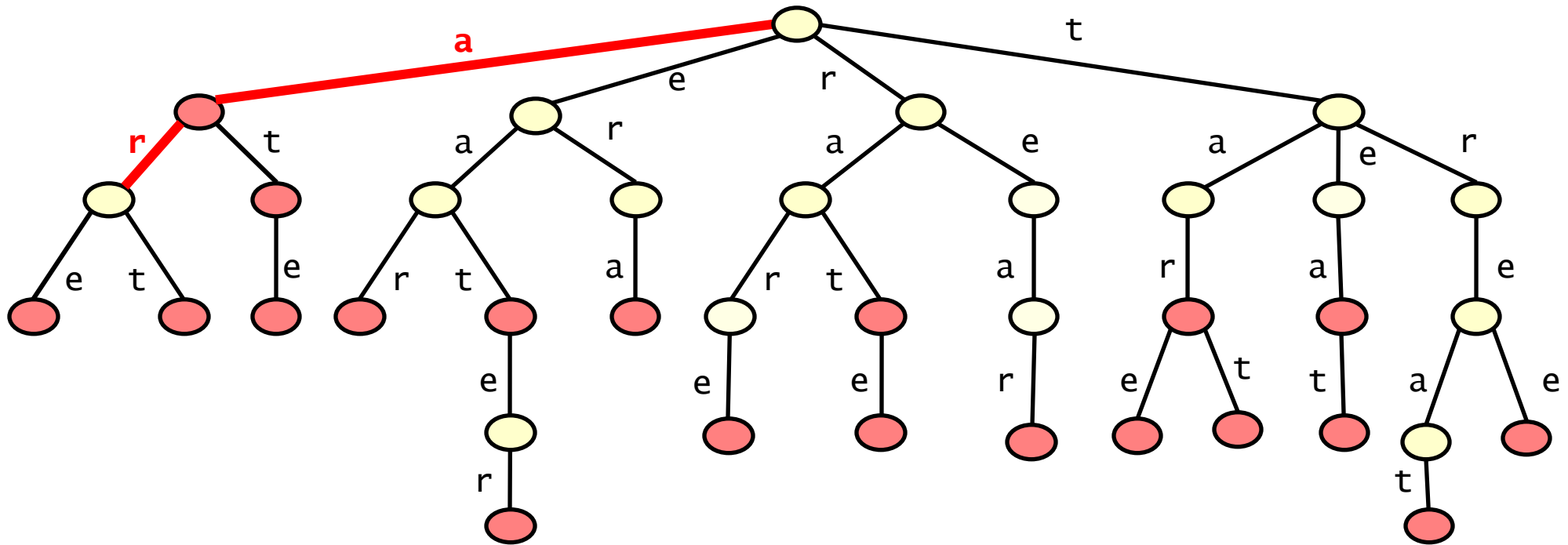


- **Two kinds of nodes**
 -  nodes representing words
 -  internal/intermediate nodes



path represents the word: **a**

Tries – Examples

An example **trie** containing words from a **4** letter alphabet



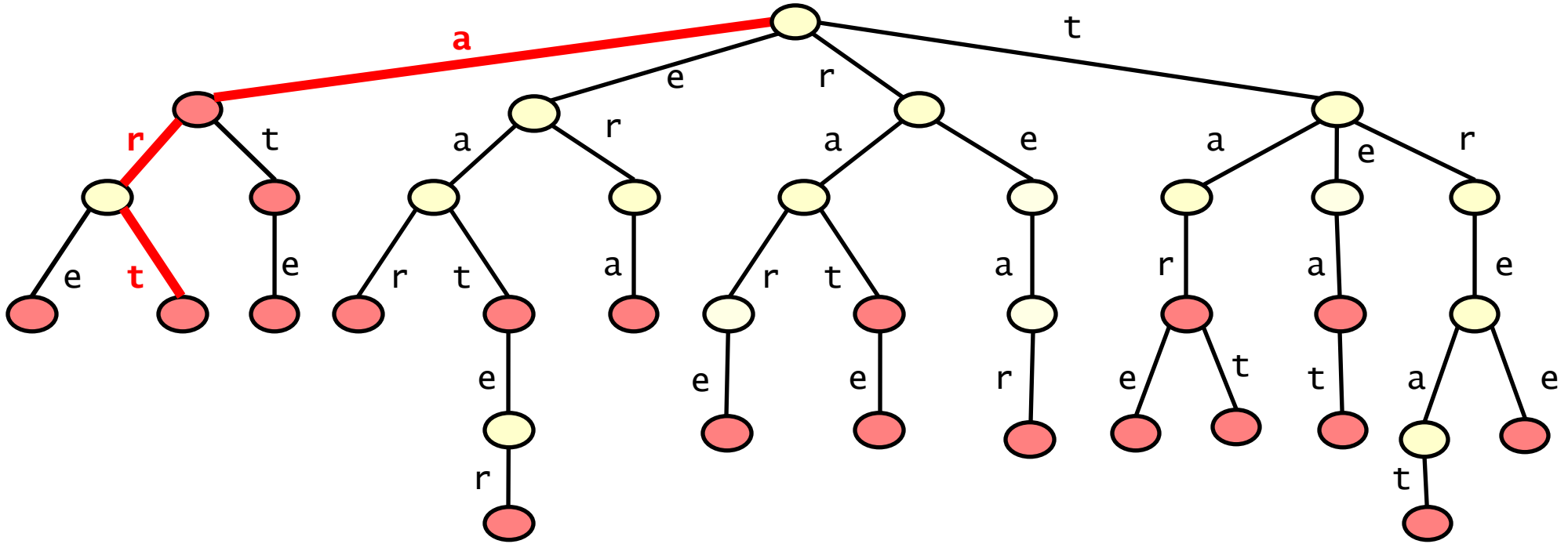
- **Two kinds of nodes**



-  nodes representing words
-  internal/intermediate nodes

path represents the string: **ar**
(not a word)

Tries – Examples

An example **trie** containing words from a **4** letter alphabet

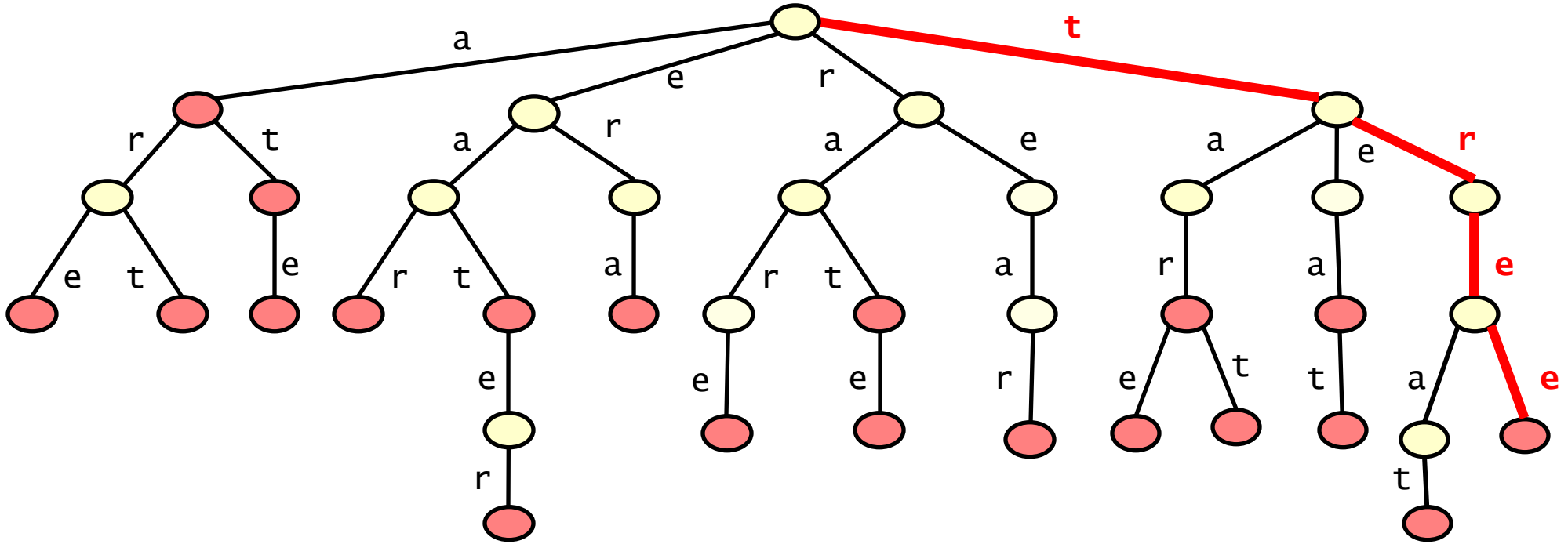




- **Two kinds of nodes**
 -  nodes representing words
 -  internal/intermediate nodes

path represents the word: **art**

Tries – Examples

An example **trie** containing words from a **4** letter alphabet



- **Two kinds of nodes**
 -  nodes representing words
 -  internal/intermediate nodes

path represents the word: **tree**

Tries – Search algorithm (pseudo code)

```
// searching for a word w in a trie t
Node crtNode = root of t; // current node (start at root)
int crtPos = 0; // current position in word w (start at beginning)

while (true) {
    if (crtNode has a child childNode labelled w.charAt(crtPos)) {
        // can match the character of word in the current position
        if (crtPos == w.length()-1) { // end of word
            if (childNode is an 'intermediate' node) return "absent";
            else return "present";
        }
        else { // not at end of word
            crtNode = childNode; // move to child node
            crtPos++; // move to next character of word
        }
    }
    else return "absent"; // cannot match current character
}
```

Tries – Insertion algorithm (pseudo code)

```
// inserting a word w in a trie t
Node crtNode = root of t; // current node (start at root)

for (int i=0; i < w.length(); i++){ // go through characters of word w
    if (crtNode has no child childNode labelled w.charAt(i)){
        // need to add new node
        create such a child childNode;
        mark childNode as intermediate;
    }
    crtNode = childNode; // move to child node
}
mark crtNode as representing a word;
```

Tries – Algorithms

Deletion of a string from a trie

- exercise – Q4 in Tutorial 1

Complexity of trie operations

- (almost) independent of the number of items
- essentially linear in the string length

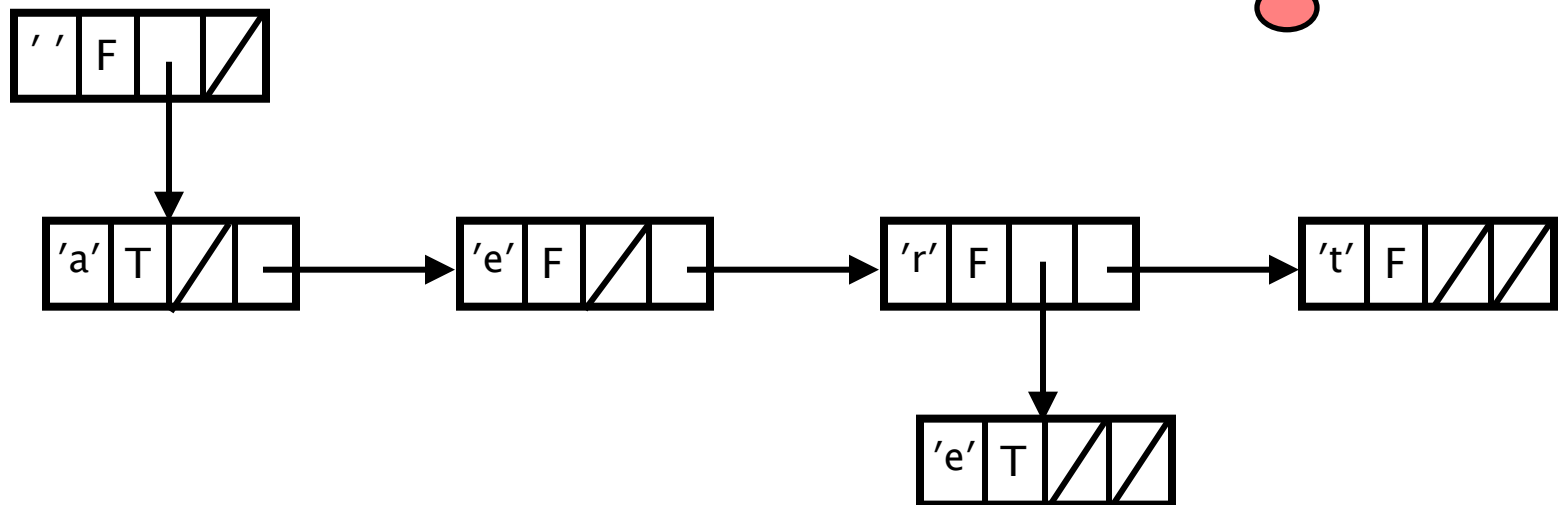
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



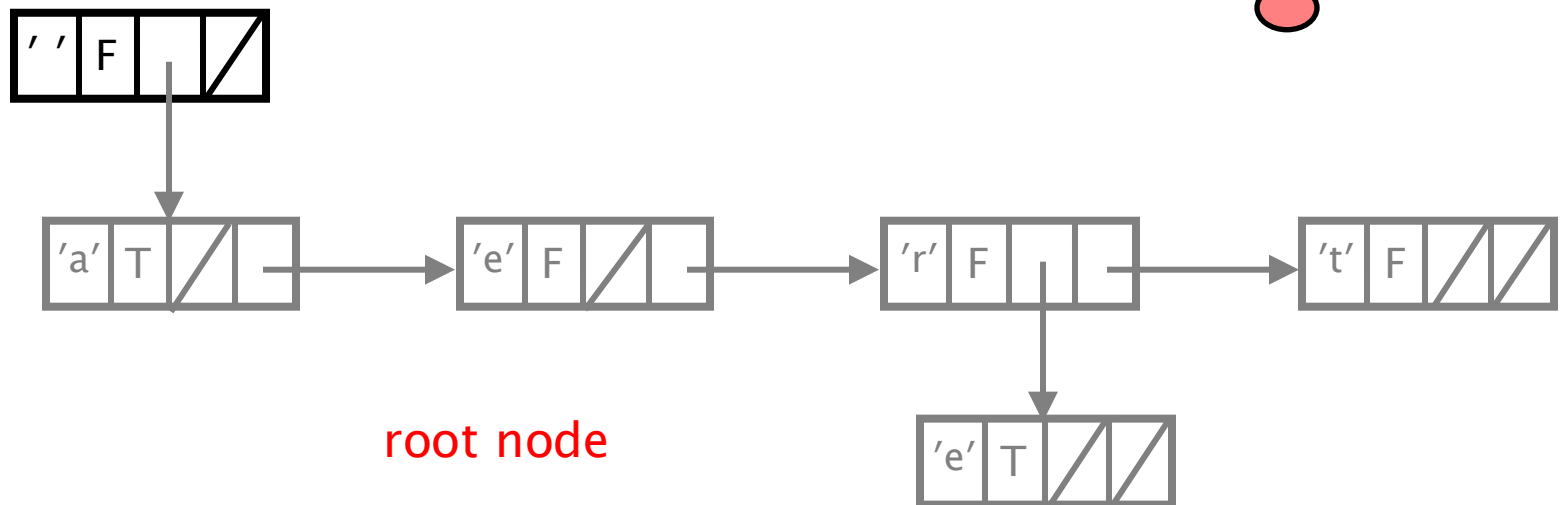
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



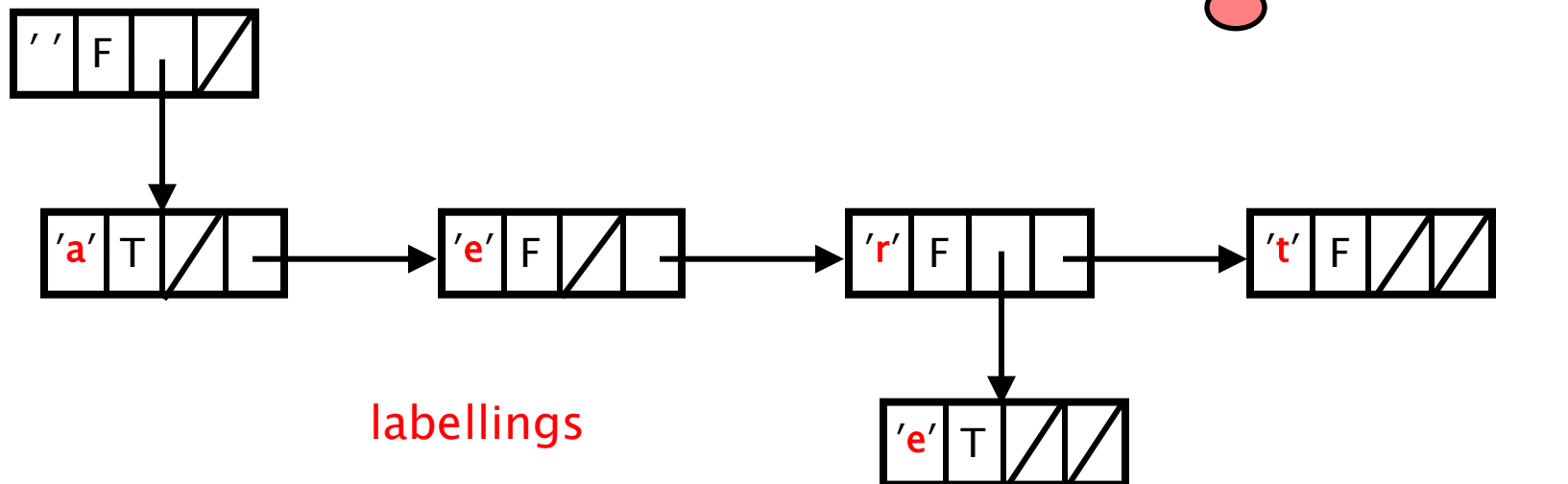
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



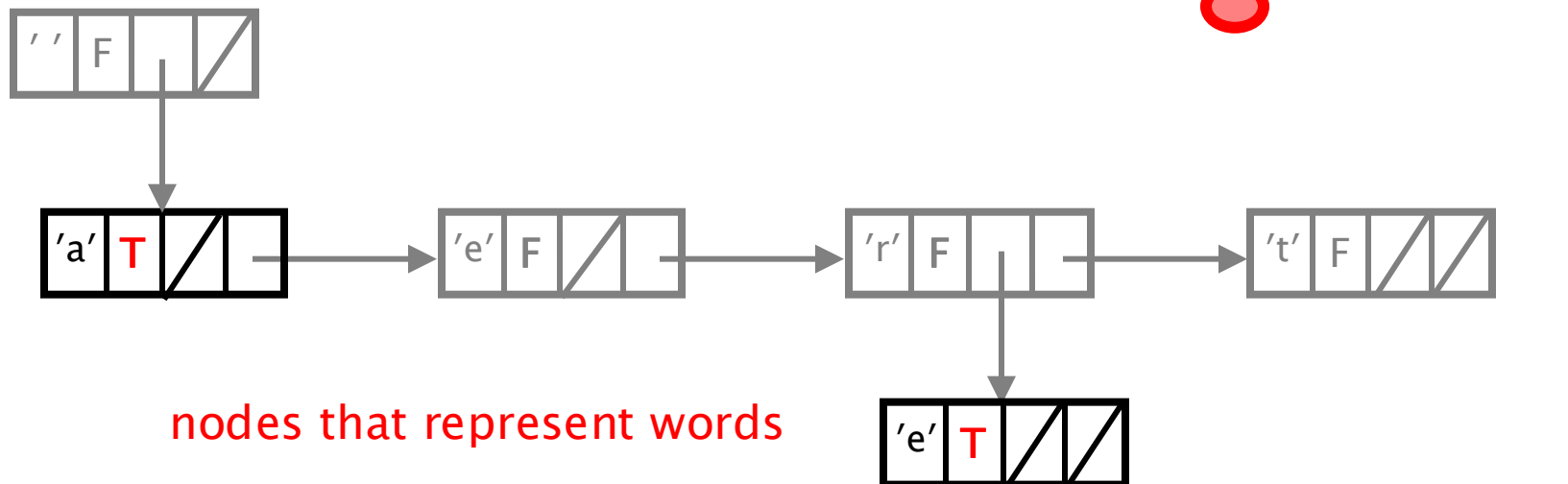
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



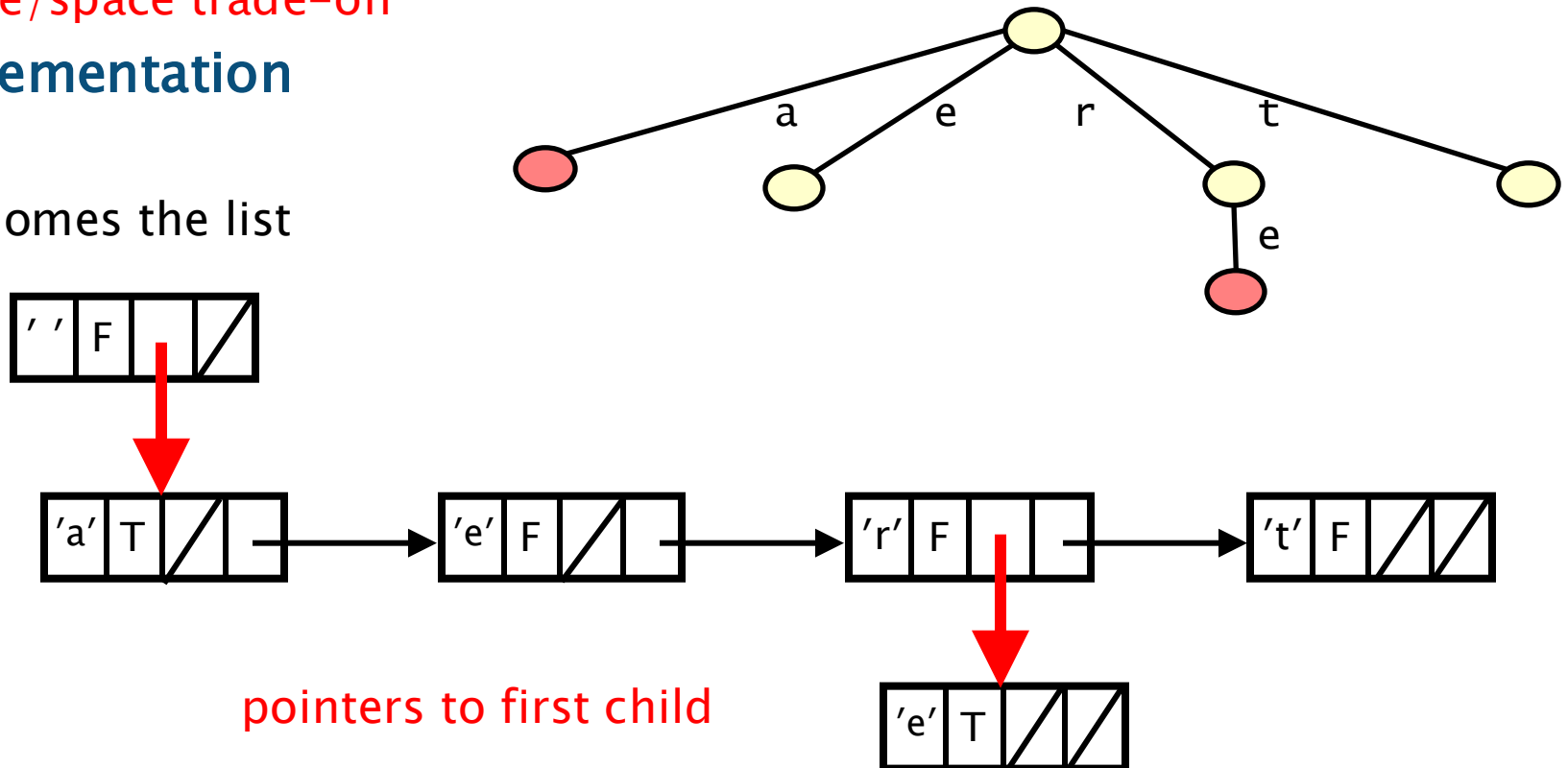
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



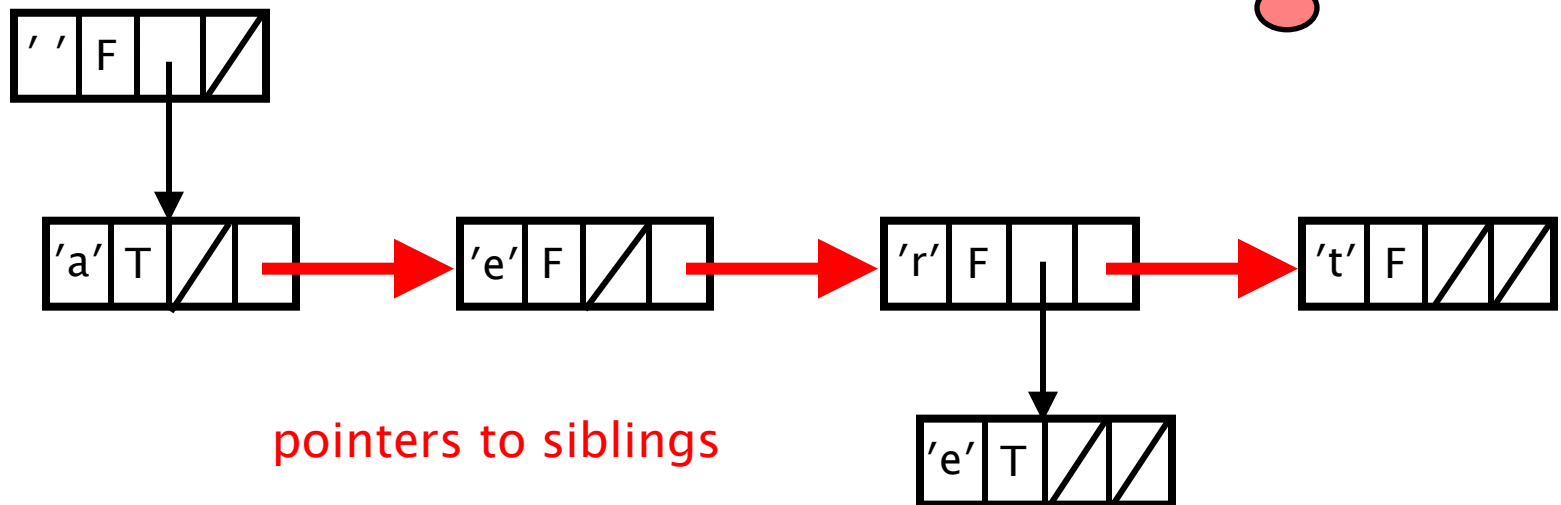
Tries – Implementation

Various possible implementations

- using an **array** (of pointers to represent the children of each node)
- using a linked **lists** (to represent the children of each node)
- **time/space trade-off**

List implementation

- trie
- becomes the list



Tries – Some applications

Autocomplete systems

- such as those found in search engines, text editors, or IDEs, where quick suggestions for partial inputs are necessary

IP Routing in networking

- tries are used for IP routing to efficiently find the best matching prefix for an IP address, facilitating quick data routing

Spell checking and correction

- quickly find words that closely match the given input, enhancing user experience in digital communication platforms

Search engine optimisation

- in indexing and prefix-based search queries

DNA sequencing in bioinformatics

- for handling genomic sequences, allowing for efficient searching and matching of DNA sequences

Outline of course – in the next lecture

Section 0: Quick recap on algorithm analysis

Section 1: Sorting algorithms

Section 2: Strings and text algorithms

Section 3: Graphs and graph algorithms

Section 4: An introduction to NP completeness

Section 5: A (very) brief introduction to computability