

# Algorithms

---

Lent Term 2024/25

Dr John Fawcett

[jkf21@cam.ac.uk](mailto:jkf21@cam.ac.uk)

Computer Science Tripos, Part IA

👍 With acknowledgement and thanks to Peter Rugg for his many helpful comments, suggestions and corrections on earlier versions of these notes!

# Practicalities - Timetable

---

Short form: MWF10

Longer form: Mondays, Wednesdays, Fridays at 10am, whole term (24L)

Where: Arts School Lecture Theatre A, New Museums Site

--27 Jan --3 Feb --10 Feb --17 Feb --24 Feb --3 Mar --10 Mar--17 Mar

10:00-11:00

N C F--F F F--F F F--F F F--F F F--F F F--F F F--F F JKF Algorithms

# Practicalities - Algorithms 1 and 2, assessment

---

24 lectures are split into 12 + 12 for Algorithms 1 + Algorithms 2.

Assessed via...

- Two 'ticks'
- One exam question from a choice of two on Algorithms 1 material (CST Paper 1)
- One exam question from a choice of two on Algorithms 2 material (CST Paper 1)

## **Section D**

*Attempt 1 question*

7 Algorithms 1

8 Algorithms 1

## **Section E**

*Attempt 1 question*

9 Algorithms 2

10 Algorithms 2

# Practicalities - Resources

---

## Course webpages:

<https://www.cl.cam.ac.uk/teaching/2425/Algorithm1/>  
<https://www.cl.cam.ac.uk/teaching/2425/Algorithm2/>

## (Strongly) Recommended textbooks:

- “**CLRS**”: Thomas H. **C**ormen, Charles E. **L**eiserson, Ronald L. **R**ivest and Clifford **S**tein Introduction to Algorithms, Fourth Edition ISBN 9780262046305 Published: April 5, 2022 (*3rd edition also good!*)
- Sedgewick, R., Wayne, K. (2011). Algorithms. Addison-Wesley. ISBN 978-0-321-57351-3

# Advice

---

1. The Algorithms courses (1 and 2) are about the principles of algorithms, not the examples.

Every computer scientist needs to be able to write a program that sorts a list of numbers but our discussion of sorting algorithms will make a bigger point.

2. Write code. An algorithm is a set of instructions that *a/ways* achieve the objective. Until you have coded it and tested it (or formally proved it...) you will not be sure that you have covered every case.

The gist is often obvious; the subtle details and edge cases make the algorithm.

# Acknowledgements

---

Thanks to Peter Rugg, Amir Fazel, Victoria Mankin and many others who pointed out mistakes and offered corrections to earlier versions.

# Algorithms 1

---

## Section 1: Sorting

# What is an algorithm?

---

- CLRS: “Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.”
- Algorithms solve problems.
- What’s a problem?



# Problems and Problem Instances

---

- We distinguish problems and problem instances.
- **Problems** have a specified input and output. These need to be precise. e.g.  
Input: a sequence of numbers  $a_1, a_2, a_3, \dots a_n$   
Output: a reordering of the input sequence,  $b_1, b_2, b_3, \dots b_n$  such that  
 $b_1 \leq b_2 \leq b_3 \leq \dots \leq b_n$
- A **problem instance** is a specific set of inputs (meeting whatever constraints the problem imposes)  
Instance: 9, 102, 10, -7, 64, 18
- Notice that a problem instance has a specific size whereas a problem can describe the requirement for any size input and output.

# Correctness

---

- We say that an algorithm is **correct** if, for every input instance, the algorithm terminates (halts) with the correct output.
- Correct algorithms **solve** their problem.
- An incorrect algorithm might not halt for some input instances.
- An incorrect algorithm might give the wrong answer for some input instances.
  
- In this course, we are only interested in correct algorithms.
- The Part II course, Randomised Algorithms, considers a special branch of incorrect algorithms.

# Notation [1]

---

This course uses the same notation as CLRS:

- An array,  $A[1..n]$  has length  $n$  and its indices are numbered  $1..n$ .
- $A[1]$  is the first item **<< BEWARE! A common source of bugs!**
- We use  $A.length$  to refer to the number of items in an array.
  - This is familiar to Java and Python programmers
  - In C and C++, an array does not have a “.length” property and so we must provide the array and its length as two arguments to any function that needs access to the length.

```
void sort(int[] A) {
```

```
    ...  
}
```

Java/Python style.  
Passing an array argument gives  
access to the length

```
void sort(int *A, uint32_t len) {
```

```
    ...  
}
```

C/C++ style.  
Length is provided separately.

## Notation [2]

---

Also in common with CLRS, we write our algorithms in **pseudocode**.

Pseudocode is a hypothetical programming language, usually an imperative programming language, to abstract away from distracting syntactic details.

Block-structured, fixed-form syntax (white space / indenting matters, like Python).

At various points in the course we will note that our pseudocode is especially helpful or concise!

We must always be careful not to imagine that the impossible can be achieved by inventing pseudocode steps that our “real” (target) computer architecture does not support. (See Computation Theory, Complexity Theory and Quantum Computing.)

## Notation [3]

---

In our pseudocode, parameters are passed by value and objects (including arrays) are passed as pointers.

We follow Java conventions for short-circuiting boolean operators:  $A|B$  and  $A\&B$  always evaluate both  $A$  and  $B$  whereas  $A||B$  and  $A\&\&B$  only evaluate  $B$  if the value of  $A$  has not already determined the final value.

Loop induction variables hold their final value after the loop.

# Sorting

---

Problem statement:

- **Input:** a sequence of numbers  $a_1, a_2, a_3, \dots a_n$  (not necessarily distinct)
- **Output:** a reordering of the input sequence,  $b_1, b_2, b_3, \dots b_n$  such that  $b_1 \leq b_2 \leq b_3 \leq \dots \leq b_n$

The items to be sorted are often called **keys** and might have attached **payloads** (also known as **values**). We sort by the keys and move the payloads around with the keys so they end up sorted by their keys. Example: sorting an array of person records by passport number has `key=passport_number`, `payload=person_record`.

# INSERTION-SORT(A)

---

```
for j=2 to A.length  
    → Key = A[j]  
      i = j - 1  
      while i>0 && A[i]>Key  
          → → A[i+1] = A[i]  
              i = i - 1  
      A[i+1] = Key
```

The white space indicates which statements are inside the **for** loop and the **while** loop. This is what it means to have a fixed-form syntax.

# How does INSERTION-SORT(A) work? [1]

---

Earlier problem instance: 9, 102, 10, -7, 64, 18

A=[9, 102, 10, -7, 64, 18]

Considering a prefix of length 1, key 9 is correctly sorted in the array [9].

Begin the FOR loop, j=2:

Key = 102

i = 1

WHILE 1>0 && 9>102 – no iterations

A[2] = 102

```
for j=2 to A.length
    Key = A[j]
    i = j - 1
    while i>0 && A[i]>Key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = Key
```



# How does INSERTION-SORT(A) work? [2]

---

A=[9, 102, 10, -7, 64, 18]

Next iteration of the FOR loop, j=3:

Key = 10

i = 2

WHILE 2>0 && 102>10

A[3] = 102

i = 1

WHILE 1>0 && 9>102 – false, stop looping

A[2] = 10

```
for j=2 to A.length
    Key = A[j]
    i = j - 1
    while i>0 && A[i]>Key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = Key
```

# How does INSERTION-SORT(A) work? [3]

---

A=[9, 10, 102, -7, 64, 18]



The part we have done.    The part we have yet to do.

```
for j=2 to A.length
    Key = A[j]
    i = j - 1
    while i>0 && A[i]>Key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = Key
```

Exercise for the reader: single-step through the remaining steps to convince yourself that this does correctly sort the input sequence.

# INSERTION-SORT(A)

## Proving correctness

```
1  for j=2 to A.length
2      Key = A[j]
3      i = j - 1
4      while i>0 && A[i]>Key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = Key
```

A: “Is your algorithm correct?”

B: “Yes.”

A: “Can you prove it?”

B: “Erm...”



💡 Hint! It is helpful to number the lines of your algorithms so it's easy to refer to them, including in supervisions and exams!

# How can we prove correctness?

---

Many algorithms contain a loop, just like Insertion Sort.

We can often prove the correctness of a loop by talking about a mathematical or logical statement (a **property**) that involves the program variables (and sometimes meta-variables) and that has three characteristics:

1. **Initialisation**: it is true before the loop begins
2. **Maintenance**: each iteration of the loop changes the values of the variables in a way that preserves the truth of the statement (i.e. a **loop invariant**)
3. **Termination**: when the loop terminates, the condition for termination combined with the mathematical statement, proves correctness.

# Properties

---

- It is easy to come up with something that is true at the start, e.g.  $1=1$ .
- Does each iteration of the FOR loop preserve the property that  $1=1$ ? Yes.
- Is this useful at loop termination? No.

Why not? Because it didn't help us prove correctness. The “Termination” condition is the important one: being true at the end of the algorithm needs to prove correctness. This is also why the property needs to refer to the variables used in the program!

# Proving the correctness of INSERTION-SORT(A)

---

Consider this property, P:

$P \stackrel{\text{def}}{=} \textit{At the start of each iteration of the FOR loop, the subarray } A[1..j-1] \textit{ contains the elements originally in positions } 1..j-1, \textit{ but in sorted order.}$

To be perfectly precise, we ought to define what “at the start of each iteration of the FOR loop” means.

- It means after assigning the next value to variable  $j$  but before executing the first line of the loop body.

```
1  for j=2 to A.length
2      Key = A[j]
3      i = j - 1
4      while i>0 && A[i]>Key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = Key
```

# Proving the initialisation of INSERTION-SORT(A)

---

*P<sub>def</sub> At the start of each iteration of the FOR loop, the subarray  $A[1..j-1]$  contains the elements originally in positions  $1..j-1$ , but in sorted order.*

Before the first iteration, we have  $j=2$  and  $A=[9, 102, 10, -7, 64, 18]$ .

The subarray  $A[1..j-1]$  is  $[9]$  and that does indeed contain the same keys as the original subarray from  $1..j-1$ , and they are in sorted order.

Initialisation: 

```
1  for j=2 to A.length
2      Key = A[j]
3      i = j - 1
4      while i>0 && A[i]>Key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = Key
```

# Proving the maintenance of INSERTION-SORT(A)

---

*P<sub>def</sub> At the start of each iteration of the FOR loop, the subarray  $A[1..j-1]$  contains the elements originally in positions  $1..j-1$ , but in sorted order.*

The FOR loop works by moving each key  $A[j-1]$ ,  $A[j-2]$ , ... one place to the right until it finds a key that should not be moved to create space to insert  $A[j]$  in the correct place in sorted order.

Then it inserts  $A[j]$  in the place it found.

⚠ Check that this works if zero items are moved and if every item from  $1..j-1$  is moved.

Maintenance: 

```
1  for j=2 to A.length
2      Key = A[j]
3      i = j - 1
4      while i>0 && A[i]>Key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = Key
```



# Proving the termination (case) of INSERTION-SORT(A)

*P<sub>def</sub> At the start of each iteration of the FOR loop, the subarray A[1..j-1] contains the elements originally in positions 1..j-1, but in sorted order.*

The FOR loop finishes when  $j = n+1$  (where  $n$  is  $A.length$ ) because  $j = j+1$  is how the loop advances. Inserting  $j = n+1$  into P, the subarray  $A[1..n]$  (where  $n$  is  $A.length$ ) contains the original elements in sorted order.

Termination: 

Correctness of INSERTION-SORT(A): 

Nuance: I have proved *partial correctness*.

```
1  for j=2 to A.length
2      Key = A[j]
3      i = j - 1
4      while i>0 && A[i]>Key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = Key
```

# Partial and Total Correctness

---

**Partial Correctness:** if we terminate at all, then we do so with the correct answer.

**Total Correctness:** partial correctness plus assurance that we will terminate on all input instances.

Is Insertion Sort also totally correct?

In this case, the FOR loop terminates since it counts upwards from 2 to the array length, which is finite, so we always reach the stopping condition.

The WHILE loop terminates in every iteration of the FOR loop because it counts down from  $(j-1) > 0$  (because  $j > 1$ ) while  $i > 0$ , with  $i = i - 1$ . Well-founded induction.

# [Non-Examinable] Link to Discrete Maths

---

Later this term you will meet sets that are inductively defined by axioms and rules.

We can define a set of correct algorithms as those inductively defined by rules such as this rule from Hoare Logic:

$$\frac{\{P\} \ B \ \{P\} \qquad P \ \& \ !C \Rightarrow Q}{\{P\} \ \mathbf{WHILE} \ C \ \mathbf{DO} \ B \ \{Q\}}$$

P, Q are the pre-condition and post-condition. C is a boolean-valued expression in the programming language, and B is the body of the loop.

# Algorithm Analysis

---

Analysis is about predicting resource requirements for input instances that we have not yet run the algorithm on.

- Memory requirements (space complexity)
- CPU time (time complexity)
- Disk operations

In order to express the resource requirements as a function of the input size, we need to measure the size of an input instance.

The measure of size needs to be “succinct”.

# Measuring instance size

---

How large was our instance  $A=[9, 102, 10, -7, 64, 18]$ ?

Possible answers:

- 1 – because there is just one array?
- 6 – because there are 6 numbers in the array?
- 24 – because  $6 \times 4$ -byte integers is 24 bytes? Or 48 bytes on a 64-bit CPU.
- 28 (or 56) – because there is also the “.length” attribute to consider?
- Should we count bits, not bytes?

# Succinct size

---

The correct measure to use depends on the steps that our algorithm performs.

We are going to derive formulae measuring the time and space complexities as functions of the input size.

We must be careful not to 'hide' large amounts of work by measuring size in a poor way because that would give misleading claims about the cost of our algorithms.

Let's look at my six suggestions in a little more detail...

# Size=1

---

This is almost surely an inappropriate measure for INSERTION-SORT(A). It performs non-zero work on each item in the input array so, without looking any further into what it does, the time taken must depend on the length of the array.

The time taken by the algorithm below could be stated as a function of the number of arrays provided: it is a constant-time function.

```
1  void printNumberOfArrays (A)
2      if (A == null) print "No arrays"
3      else print "One array"
```

# Size=A.length

---

This is correct in all every-day implementations, and in our pseudocode. Why any caveats?

What about arrays that are so long that our programming language's integer data type cannot hold large enough values for the “length” attribute? If we use arbitrary precision arithmetic (arithmetic where numbers are themselves arrays of digits) then “+1” is not a constant time operation because a carry can ripple along the array of digits and the number of CPU cycles required depends on the size of the value we are incrementing.

Similarly if the values in the array are arbitrary precision: > takes variable time!



# Size=number of bytes or bits

---

Using the size in bytes (with or without counting array length) correctly accounts for arrays of arbitrary-precision values.

Using the size in bits is helpful when an algorithm operates at the bit-level, e.g. an algorithm that multiplies two  $n$ -bit numbers, or the various types of adder you study in Digital Electronics.

## What about $\text{Size} = 2^{A.\text{length}}$ ?

---

This is not appropriate because it (massively) overestimates the size of the instance we were asked to work on. If we think that an input instance is much larger than it really is, then terrible algorithms could look very efficient!



The concept of a *succinct* measure of the size of an input will be defined formally in Part IB Complexity Theory. For Algorithms 1, an intuition is sufficient and in Algorithms 2, when we look at some more exotic data structures, we will be precise about how we want our input to be represented such that its size is clear.

## A little more on being precise...

---

We must also be clear about what our memory can contain and the cost of the basic steps in our pseudocode language. You would need to change this if you use fancy hardware, a quantum computer, magic, etc.

- Memory 'cells', such as the elements in an array, can only hold one item.
- Indexing an array item takes one time unit (for read or write):  $A[i]$  takes 1 unit
- Arithmetic operations take 1 time unit
- Comparisons take 1 time unit
- Assignments to variables take 1 time unit

The **running time** of an algorithm is the number of primitive operations performed.

# Cost of INSERTION-SORT(A)

---

1	<b>for</b> j=2 <b>to</b> A.length	Cost a, executed $n-1 + 1 = n$ times
2	Key = A[j]	Cost b, executed $n-1$ times
3	i = j - 1	Cost c, executed $n-1$ times
4	<b>while</b> i>0 && A[i]>Key	Cost d, executed $\sum_{j=2}^n t_j$ times
5	A[i+1] = A[i]	Cost e, executed $\sum_{j=2}^n (t_j-1)$ times
6	i = i - 1	Cost f, executed $\sum_{j=2}^n (t_j-1)$ times
7	A[i+1] = Key	Cost g, executed $n-1$ times



Remember that FOR and WHILE loops evaluate the loop condition on the occasion that stops the loop (i.e. after the last iteration to determine that no more iterations are required).

# Cost of INSERTION-SORT(A)

$$T(n) = an + (b+c+g)(n-1) + d \sum_{j=2}^n t_j \\ + (e+f) \sum_{j=2}^n (t_j-1)$$

where  $t_x$  is number of iterations of the WHILE loop that are performed when  $j = x-1$ .

⚠ Notice that  $t_j$  depends on the input data!

💡 We said that  $a=b=c=d=e=f=g=1$  time unit but any *constant* amount of time would not change the functional form.

Cost a, executed  $n-1 + 1 = n$  times

Cost b, executed  $n-1$  times

Cost c, executed  $n-1$  times

Cost d, executed  $\sum_{j=2}^n t_j$  times

Cost e, executed  $\sum_{j=2}^n (t_j-1)$  times

Cost f, executed  $\sum_{j=2}^n (t_j-1)$  times

Cost g, executed  $n-1$  times

💡 The running time,  $T(n)$ , on inputs of size  $n$  is the cost of each line multiplied by the number of times the line is executed.

## Best case cost of INSERTION-SORT(A)

---

$$T(n) = an + (b+c+g)(n-1) + d \sum_{j=2}^n t_j + (e+f) \sum_{j=2}^n (t_j-1)$$

If the input data is in sorted order, no iterations of the WHILE loop are ever performed  $\Rightarrow t_j = 1$

$$T(n) = pn + q \text{ for constants } p \text{ and } q$$

Linear time complexity,  $O(n)$ , in the base case.

# Worst case cost of INSERTION-SORT(A)

---

$$T(n) = an + (b+c+g)(n-1) + d \sum_{j=2}^n t_j + (e+f) \sum_{j=2}^n (t_j-1)$$

If the input data is in reverse sorted order, each WHILE loop performs  $j-1$  iterations  
 $\Rightarrow t_j = j$  (remembering that the WHILE loop does a comparison after the last iteration to determine that no more should occur).

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j = n(n+1)/2 - 1 \quad \text{Sum of an arithmetic sequence - 1 (to exclude 1<sup>st</sup> term)}$$

$$\sum_{j=2}^n (t_j-1) = \sum_{j=2}^n (j-1) = n(n-1)/2 \quad \text{Sum of an arithmetic sequence}$$

$T(n) = pn^2 + qn + r$  for *constants*  $p$ ,  $q$  and  $r$ . Quadratic time complexity,  $O(n^2)$ .

# Average case cost of INSERTION-SORT(A)

---

On average, one half of the keys in the subarray  $A[1..j-1]$  will be less than  $A[j]$ . This also gives two arithmetic progressions and a  $T(n)$  that is  $O(n^2)$ .



The average case is often the same as the worst case, as happens for Insertion Sort.



# Which case do we want?

---

Usually (and if unspecified) we want the **worst case** running time and/or memory consumption, because:

1. It gives an upper bound: if we provision sufficient resources we know that the computation will succeed.
2. The worst case might occur fairly often (e.g. humans might not use a computer to perform an “easy case” that they could do quicker in their heads).
3. The average case is often the same order of magnitude as the worst case and for such algorithms there is no interesting differentiation to be made between average case and worst case.

# Order of Growth

---

We often care most about how rapidly the expected running time grows with the size of the input. Prof. Robert Sedgewick (Princeton) captures this nicely: “Good algorithms are better than supercomputers” and “Great algorithms are better than good ones.” His book motivates this:

	Insertion Sort $O(N^2)$			MergeSort $O(N \lg N)$			QuickSort $O(N \lg N)$ [expected]		
	Thousand	Million	Billion	Thousand	Million	Billion	Thousand	Million	Billion
Home PC	instant	2.8 hours	317 years	instant	1 second	18 minutes	instant	0.6 seconds	12 minutes
Super-computer	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

# Order of Growth: $\Theta(g(n))$

---

$\Theta(g(n))$  is the set of functions,  $f(n)$ , such that there exist positive constants  $c_1$ ,  $c_2$  and  $n_0$  such that  $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$  for all  $n \geq n_0$ .

$g(n)$  is an **asymptotically tight bound** for  $f(n)$ . This means, within a constant (multiplicative) factor.

Example: if the true cost is  $10.3n^2 + 6.1n - 0.4$ , we can write  $\Theta(n^2)$  but not  $\Theta(n^4)$  or  $\Theta(n)$ .

# Order of Growth: $O(g(n))$

---

$O(g(n))$  is the set of functions,  $f(n)$ , such that there exist positive constants  $c$  and  $n_0$  such that  $0 \leq f(n) \leq c g(n)$  for all  $n \geq n_0$ .

$g(n)$  is an **asymptotic upper bound** for  $f(n)$ .

Notice that  $\Theta(g(n)) \subseteq O(g(n))$

Example:

If the true cost is  $10.3 n^2 + 6.1n - 0.4$ , we can write  $O(n^2)$  and  $O(n^4)$  but not  $O(n)$ .

# Order of Growth: $\Omega(g(n))$

---

$\Omega(g(n))$  is the set of functions,  $f(n)$ , such that there exist positive constants  $c$  and  $n_0$  such that  $0 \leq c g(n) \leq f(n)$  for all  $n \geq n_0$ .

$g(n)$  is an **asymptotic lower bound** for  $f(n)$ .

Notice that  $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$

Example:

If the true cost is  $10.3 n^2 + 6.1n - 0.4$ , we can write  $\Omega(n^2)$  and  $\Omega(n)$  but not  $\Omega(n^4)$ .

# Order of Growth: $o(g(n))$ and $\omega(g(n))$

---

Little-o and little-omega denote asymptotically non-tight versions of their ‘big’ counterparts.  $O(n^2)$  is an *asymptotically tight* bound for  $3n^2$  but  $O(n^3)$  is not.  $o(g(n))$  is non-tight.

$o(g(n))$  is the set of functions,  $f(n)$ , such that for any positive constant  $c$ , there exists a positive constant  $n_0$  such that  $0 \leq f(n) < c g(n)$  for all  $n \geq n_0$ .

$\omega(g(n))$  is the set of functions,  $f(n)$ , such that for any positive constant  $c$ , there exists a positive constant  $n_0$  such that  $0 \leq c g(n) < f(n)$  for all  $n \geq n_0$ .

Examples:  $3n^2$  is in  $O(n^2)$  and  $\Omega(n^2)$  but neither in  $o(n^2)$  nor  $\omega(n^2)$ .

# Useful properties

---

**Transitivity:**  $\Theta$ ,  $O$ ,  $\Omega$ ,  $o$ ,  $\omega$  are transitive

$$\text{e.g. } f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$$

**Reflexive:**  $\Theta$ ,  $O$ ,  $\Omega$  are reflexive

$$\text{e.g. } f(n) \in O(f(n))$$

**Symmetric:**  $\Theta$  is symmetric

$$\text{e.g. } f(n) \in \Theta(g(n)) \text{ if and only if } g(n) \in \Theta(f(n))$$

# Other sorting techniques

---

Insertion sort is an **incremental algorithm**: having already built a sorted version of the subarray  $A[1..j-1]$ , it inserts  $A[j]$  into the correct place to build  $A[1..j]$ .

Another technique is **divide and conquer**:

1. **Divide** the original problem into two or more *smaller* instances of the *same* problem.
2. **Conquer** the subproblems by calling the same function recursively on each of them in turn.
3. **Combine** the solutions to the subproblems to build the solution to the original problem.



# Divide and Conquer sorting: MERGE-SORT(A, p, r)

---

```
1  if p < r
2      q = floor((p+r)/2)      // Divide
3      MERGE-SORT(A, p, q)      // } Conquer the two
4      MERGE-SORT(A, q+1, r)    // } subproblems
5      MERGE(A, p, q, r)        // Combine (next slide)
```

Claim: this sorts the array, A, from index p to index r, inclusive at both ends.

## MERGE(A, p, q, r)

---

```
1  let n1 = q - p + 1
2  let n2 = r - q
3  let L = new Array(1..n1+1)
4  let R = new Array(1..n2+1)
5  L[1..n1] = A[p..p+n1-1]
6  L[n1+1] =  $\infty$ 
7  R[1..n2] = A[q+1..q+n2]
8  R[n2+1] =  $\infty$ 
9  i = j = 0
10 for k = p to r
11     if L[i]  $\leq$  R[j]
12         A[k] = L[i]
13         i = i + 1
14     else
15         A[k] = R[j]
16         j = j + 1
```

# Effect of MERGE-SORT(A, p, r) on memory

---

$A = [9, 10, 102, -7, 64, 18, \infty, \infty]$  // Size is a power of 2

MERGE-SORT(A, 1, 8):

$q=4$

because  $\text{floor}((1+8)/2)=4$

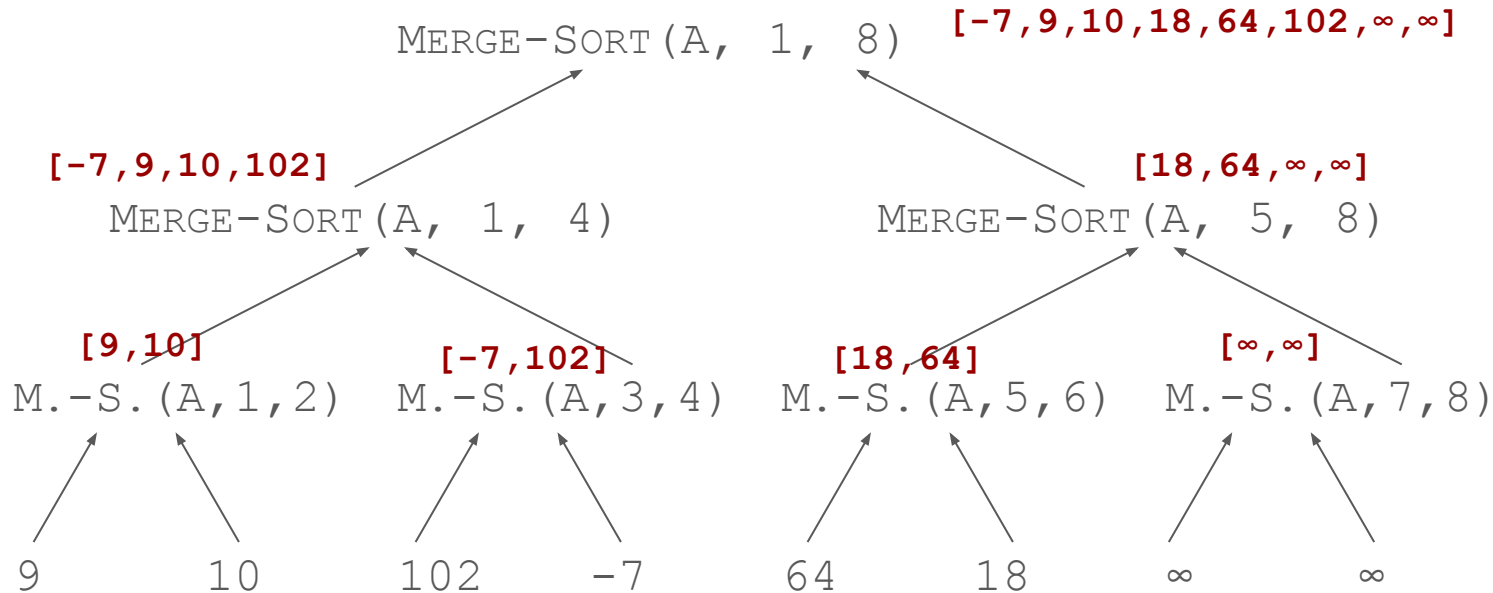
MERGE-SORT(A, 1, 4)  $\rightarrow A = [-7, 9, 10, 102, 64, 18, \infty, \infty]$

MERGE-SORT(A, 5, 8)  $\rightarrow A = [-7, 9, 10, 102, 18, 64, \infty, \infty]$

MERGE(A, 1, 5, 8)  $\rightarrow A = [-7, 9, 10, 18, 64, 102, \infty, \infty]$

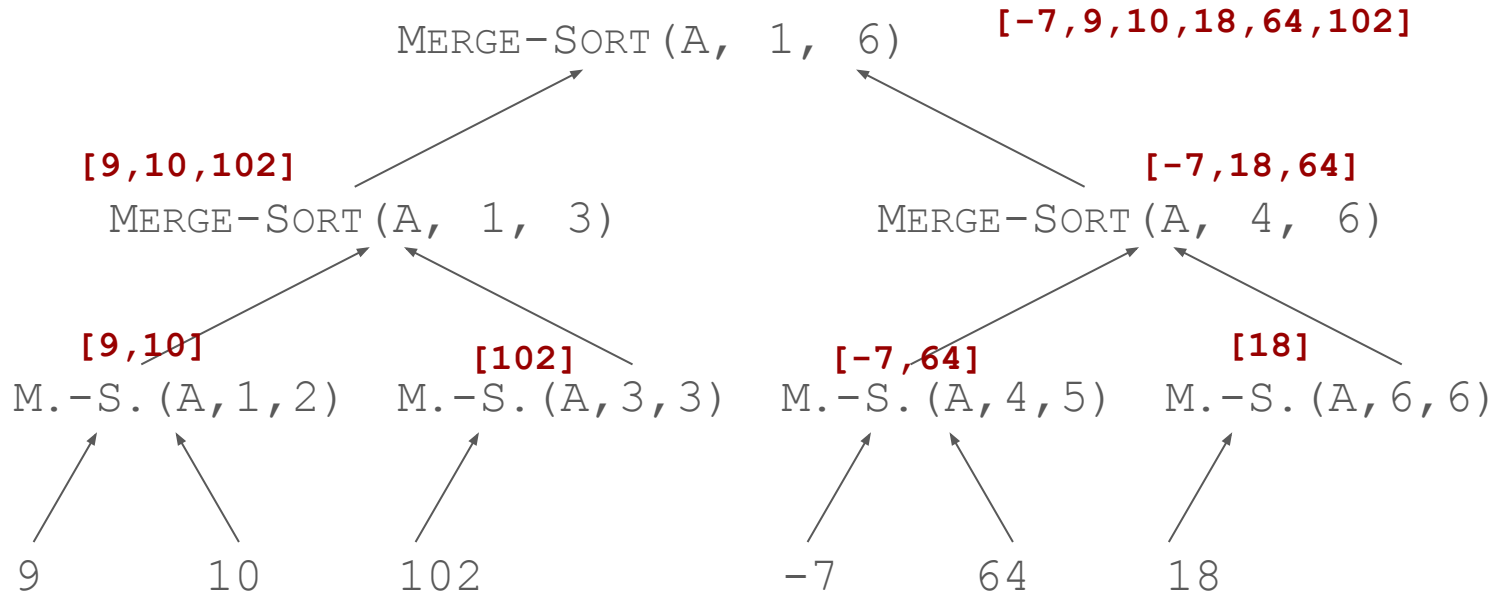
# Effect of MERGE-SORT(A, p, r) on memory

A = [9, 10, 102, -7, 64, 18,  $\infty$ ,  $\infty$ ]



# We don't need the padding!

A = [9, 10, 102, -7, 64, 18]



# Analysis of MERGE-SORT(A, p, r)

---

The measure of input size,  $n$ , is the length of the region to be sorted, i.e.  $r - p + 1$ .

E.g. the previous example sorted 1..8 and  $8-1+1 = 8$  keys to be sorted

Let  $T(n)$  be the cost of solving a problem of size  $n$  using MERGE-SORT(A, p, r).

When  $p=r$ , the algorithm stops immediately:  $T(1) = 1$

When  $p < r$ , the algorithm...

- Calculates  $q$ , which is  $\Theta(1)$  work
- Calls itself on two problems of size at most  $n/2$ :  $2 \times T(n/2)$  work
- Calls MERGE(A, p, q, r): let's call the cost of that  $M(A, p, q, r)$ .

# Analysis of MERGE(A, p, q, r)

---

Let  $n = r - p + 1$  i.e. the total length of the two subarrays to be merged

MERGE(A, p, q, r)...

- Creates and fills two arrays of total length  $n+2$  →  $\Theta(n)$  total cost
- Loops from  $k=p$  to  $r$  ( $n$  iterations)
  - Assigns into an array position – constant cost →  $\Theta(1)$
  - Adds one to either  $i$  or  $j$  – constant cost →  $\Theta(1)$  →  $\Theta(n)$  total from  $n$  iterations

Total cost of MERGE(A, p, q, r),  $M(A, p, q, r) \in \Theta(n)$

i.e.  $M(A, p, q, r) = k(r - p + 1)$  for some constant  $k > 0$ , for all  $n \geq n_0$

# Full Analysis of MERGE-SORT(A, p, r)

---

Remember  $n = r - p + 1$ .

$$T(1) = 1$$

$T(n) = \Theta(1) \text{ work} + 2 \times T(n/2) \text{ work} + M(A, p, q, r)$	// informal concept!
$= \Theta(1) \text{ work} + 2 \times T(n/2) \text{ work} + k_2 (r - p + 1)$	// mixed notation!
$= \Theta(1) \text{ work} + 2 \times T(n/2) \text{ work} + k_2 n$	// mixed notation!
$= k_1 + 2 T(n/2) + k_2 n$	// real maths!



# Closed form solution [1]

---

We want the **closed form solution**: this means not defined in terms of itself through direct or indirect recursive calls to the  $T()$  function.

Recursive form:

$$T(1) = 1$$

$$T(n) = k_1 + 2 T(n/2) + k_2 n$$

## Closed form solution [2]

---

One approach is to substitute into itself and spot a pattern.

$$\begin{aligned}T(n) &= k_1 + k_2 n + 2 T(n/2) \\&= k_1 + k_2 n + 2 (k_1 + k_2 n/2 + 2 T(n/4)) \\&= k_1 + k_2 n + 2 (k_1 + k_2 n/2 + 2 [ k_1 + k_2 n/4 + 2 T(n/8) ] ) \\&= \dots \\&= k_1 (1 + 2 + 4 + \dots) \quad // \log_2 n \text{ terms} \\&\quad + k_2 n (1 + 1 + 1 + \dots) \quad // \log_2 n \text{ terms} \\&\quad + 2^{\log n} T(1) \\&= k_1 (n - 1) + k_2 n \log_2 n + n \in \Theta(n \log_2 n) = \Theta(n \log n)\end{aligned}$$

Derivation on the next slide

Notice that the base of logs is irrelevant because a change of base is a multiplicative scale factor!

## Closed form solution [3]

---

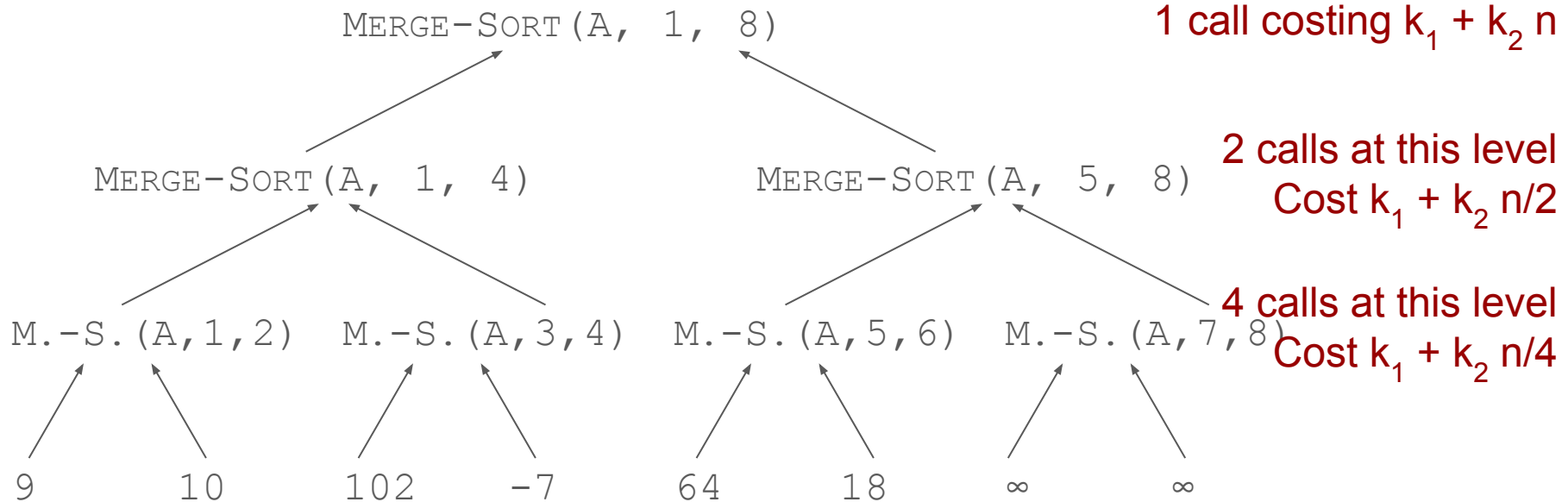
Derivation of  $k_1 (1 + 2 + 4 + \dots)$  //  $\log_2 n$  terms  
 $= k_1 (n - 1)$

This is a geometric series with  $a=1$ ,  $r=2$ ,  $n'=\log_2 n$ .

Sum first  $n$  terms  $= a(1-r^{n'})/(1-r) = 1 (1 - 2^{\log_2 n})/(1-2) = (1 - n)/-1 = n-1$ .

# Another way to find the cost of MERGE-SORT(A, p, r) [1]

Here is the call tree again:



## Another way to find the cost of MERGE-SORT(A, p, r) [2]

---

The  $k_2$  terms are asymptotically dominant over the  $k_1$  terms.

The  $k_2$  terms are  $k_2 n + 2 k_2 n/2 + 4 k_2 n/4 + \dots = k_2 n \times \text{NUM\_LEVELS}$

The tree has  $\log_2 n + 1$  levels (since  $\log_2 1 = 0$ ).

$\Rightarrow$  Total cost is  $\Theta(n \log n)$ .

# Notes on those solutions

---

1. You will see some authors using this notation where we inserted constants  $k_i$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Mathematically, this is unhygienic because we're adding a cost and a set but it's obvious why some people might prefer this crime against type correctness to the clutter of our constants!

2. We assumed that the array length was a power of 2. What if it's not?

We get  $T(n) = T(\text{ceil}(n/2)) + T(\text{floor}(n/2)) + k_1 + k_2 n$ , with the same solution.

**!** Derivations often assume that  $T(n) = 1$  for small enough  $n$ , and this is true for most algorithms. Boundary conditions are often neglected but remember to check that this applies in your own proofs!

# The Master Theorem

---

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the non-negative integers by the recurrence

$$T(1) = 1 \quad \text{and} \quad T(n) = a T(n/b) + f(n)$$

where we interpret  $n/b$  to mean either  $\text{floor}(n/b)$  or  $\text{ceil}(n/b)$ . Then  $T(n)$  has the following asymptotic bounds:

1. If  $f(n) \in O(n^{(-\varepsilon + \log_b a)})$  for some constant  $\varepsilon > 0$ , then  $T(n) \in \Theta(n^{\log_b a})$
2. If  $f(n) \in \Theta(n^{\log_b a})$ , then  $T(n) \in \Theta(n^{\log_b a} \cdot \lg n)$
3. If  $f(n) \in \Omega(n^{(\varepsilon + \log_b a)})$  for some constant  $\varepsilon > 0$ , and if  $a f(n/b) \leq c f(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) \in \Theta(f(n))$

# Understanding The Master Theorem: case 1

---

1. If  $f(n) \in O(n^{(-\varepsilon + \log_b a)})$  for some constant  $\varepsilon > 0$ , then  $T(n) \in \Theta(n^{\log_b a})$

$f(n)$  is bounded from above by  $n^{\log_b a}$ , i.e.  $n^{\log_b a} > f(n)$  beyond some point  $n_0$ , then  $n^{\log_b a}$  dominates and the solution is  $\Theta(n^{\log_b a})$ .

BUT what about  $\varepsilon$ ? That is requiring that  $n^{\log_b a}$  dominates  $f(n)$  by a polynomial factor of at least  $n^\varepsilon$  for some  $\varepsilon > 0$ .

Notice this leaves a gap between cases 1 and 2: there are situations where The Master Theorem cannot solve the recurrence equation. Fortunately, these crop up fairly rarely in practice.



## Understanding The Master Theorem: case 3

---

3. If  $f(n) \in \Omega(n^{\epsilon + \log_b a})$  for some constant  $\epsilon > 0$ , and if  $a f(n/b) \leq c f(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) \in \Theta(f(n))$

$f(n)$  is bounded from below by  $n^{\log_b a}$ , i.e.  $n^{\log_b a} < f(n)$  beyond some point  $n_0$ , then  $f(n)$  dominates and the solution is  $\Theta(f(n))$ .

Again,  $\epsilon$  requires that  $f(n)$  dominates  $n^{\log_b a}$  by a polynomial factor of at least  $n^\epsilon$  for some  $\epsilon > 0$  AND we have the “regularity” condition that  $a f(n/b) \leq c f(n)$ .

Notice this leaves another gap between cases 2 and 3 when The Master Theorem cannot solve the recurrence equation. The “regularity” condition is usually met.

# Understanding The Master Theorem: case 2

---

2. If  $f(n) \in \Theta(n^{\log_b a})$ , then  $T(n) \in \Theta(n^{\log_b a} \cdot \lg n)$

When  $f(n)$  and  $n^{\log_b a}$  are the same size, we multiply by a logarithmic factor so the solution is  $\Theta(f(n) \cdot \lg n) = \Theta(n^{\log_b a} \cdot \lg n)$ .

# The Master Theorem: MERGE-SORT(A, p, r)

---

Our recurrence relation was  $T(n) = 2 T(n/2) + k_2 n + k_1$

Applying The Master Theorem, we have  $a=2$ ,  $b=2$ , and  $f(n) = k_2 n + k_1 \in \Theta(n)$ .

Calculate  $n^{\log_b a} = n^{\log_2 2} = n^1 = n$ .

Case 2 applies because  $f(n) \in \Theta(n^{\log_b a}) = \Theta(n)$

We read off the solution:  $T(n) \in \Theta(n \lg n)$

... and cite The Master Theorem to explain our working.

Remember that  $\Theta(n \lg n) = \Theta(n \log_{10} n)$  (or any other base) as multiplicative scale factors can be ignored.

## Other examples of The Master Theorem [1]

---

$$T(1) = 1 \quad \text{and} \quad T(n) = 9 T(n/3) + n$$

We have  $a=9$ ,  $b=3$ ,  $f(n)=n$  and we calculate  $n^{\log_b a} \in \Theta(n^2)$ .

We must find some  $\varepsilon > 0$  such that  $f(n) \in O(n^{(-\varepsilon + \log_3 9)})$ ,  
i.e. that  $n \in O(n^{(-\varepsilon + \log_3 9)})$ .

$\varepsilon=0.1$  suffices:  $\log_3 9 = 2$  so  $n \in O(n^{1.9})$  and case 1 applies.

The solution is  $T(n) \in O(n^2)$ , by The Master Theorem.

## Other examples of The Master Theorem [2]

---

$$T(1) = 1 \quad \text{and} \quad T(n) = 3 T(n/4) + n \lg n$$

We have  $a=3$ ,  $b=4$ ,  $f(n)=n \lg n$  and we calculate  $n^{\log_b a} \in \Theta(n^{0.793})$ .

$f(n) \in \Omega(n^{(\varepsilon + \log_4 3)})$  when  $\varepsilon \approx 0.2$  so case 3 will apply if we can prove the “regularity” condition. For sufficiently large  $n$ , we have

$$a f(n/b) = 3 (n/4) \lg(n/4) \leq (3/4) n \lg n = c f(n) \text{ for } c = 3/4.$$

So case 3 does apply and the solution is  $T(n) \in O(n \lg n)$ , by The Master Theorem.

## Other examples of The Master Theorem [3]

---

$$T(1) = 1 \quad \text{and} \quad T(n) = 2 T(n/2) + n \lg n$$

We have  $a=2$ ,  $b=2$ ,  $f(n)=n \lg n$  and we calculate  $n^{\log_b a} \in \Theta(n)$ .

$f(n)$  is asymptotically greater than  $\Theta(n)$  so we try case 3 again.

We need a constant  $\varepsilon > 0$  such that  $f(n) \in \Omega(n^{\varepsilon + \log_b a})$  but  $f(n)/n^{\log_b a} = \lg n$  and this is asymptotically less than  $n^\varepsilon$  for any positive constant  $\varepsilon$ . Here,  $f(n)$  does dominate  $n^{\log_b a}$  but not by a polynomial factor (only by a logarithmic factor).

This case falls between cases 2 and 3, and cannot be solved by The Master Theorem.

# Exercises

---

It turns out that matrix multiplication can be solved using Strassen's Algorithm with this recurrence equation:

$$T(1) = 1 \quad \text{and} \quad T(n) = 7 T(n/2) + k n^2$$

Show that the closed form solution is  $T(n) \in \Theta(n^{\lg 7})$ .

Find the closed form solution of the recurrence equation

$$T(1) = 1 \quad \text{and} \quad T(n) = 2 T(n/4) + \sqrt{n} \quad (\text{CLRS 4.5-1 (b)})$$

## QUICKSORT(A, p, r)

```
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q-1)
4      QUICKSORT(A, q+1, r)
```

Initial call:

```
QUICKSORT(A, 1, A.length)
```

## PARTITION(A, p, r)

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```



# Intuition behind QUICKSORT(A, p, r)

---

- If the subarray to be sorted is length 0 or 1 then no action is required.
- Otherwise, pick any element as the **pivot** and rearrange the items in the subarray such that it looks like this:

[items < pivot value ... **pivot** ... items > pivot value]

*Notice that the pivot is definitely in the correct place!*

- If your array might contain duplicates, values equal to the pivot can be placed on either side arbitrarily.
- Recurse on the subarray to the left of the pivot (excluding the pivot itself)
- Recurse on the subarray to the right of the pivot (excluding the pivot itself)

# Intuition behind PARTITION(A, p, r)

---

'x' is the pivot value. It is chosen to be the last item in the subarray (index r).

<- region to be sorted ->

... 8    2    1    5    2    4    ...

p

r

x=4

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Intuition behind PARTITION(A, p, r)

We walk through the subarray maintaining 'i' as the right end of the items currently known to be less than the pivot. If we find an item that should be to the left of the pivot, we increase 'i' and put it there.

<- region to be sorted ->

... 8 2 1 5 2 4 ...

i      p,j                                  r

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

Key: known less than pivot, Pivot, known greater than pivot, not processed yet

## Intuition behind PARTITION(A, p, r)

$8 \leq 4$  is false so we move on:

<- region to be sorted ->

... 8 2 1 5 2 4 ...

i p j r

```

1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1

```

# Intuition behind PARTITION(A, p, r)

---

$2 \leq 4$  is true so we run the IF:

<- region to be sorted ->

... **2** **8** **1** **5** **2** **4** ...  
p,i j r

Then we move on again:

... **2** **8** **1** **5** **2** **4** ...  
p,i j r

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Intuition behind PARTITION(A, p, r)

---

$1 \leq 4$  is true so we run the IF:

<- region to be sorted ->

... **2** **1** **8** **5** **2** **4** ...  
p i j r

Then we move on again:

... **2** **1** **8** **5** **2** **4** ...  
p i j r

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Intuition behind PARTITION(A, p, r)

---

$5 \leq 4$  is false so we move on:

<- region to be sorted ->

... **2** **1** **8** **5** **2** **4** ...  
p i j r

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Intuition behind PARTITION(A, p, r)

---

$2 \leq 4$  is true so we run the IF:

<- region to be sorted ->

... **2** **1** **2** **5** **8** **4** ...  
p i j r

Then we move on again:

... **2** **1** **2** **5** **8** **4** ...  
p i j,r

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```



# Intuition behind PARTITION(A, p, r)

---

Loop ends; swap the pivot into place:

<- region to be sorted ->

... **2** **1** **2** **4** **8** **5** ...  
p i j,r

Return  $i+1 \Rightarrow$  the pivot position!

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Intuition behind PARTITION(A, p, r)

---

Notice that 5 & 8 swapped order several times. We never compared these two sort keys and we have no preference for which order they end up in, only that both are to the right of the pivot.

... **2** **1** **2** **4** **8** **5** ...

```
1  x = A[r]
2  i = p - 1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i+1], A[r])
8  return i + 1
```

# Algorithm paradigm

---

This is a **divide and conquer** algorithm.

**Divide:** partition the subarray,  $A$ , to be sorted into three regions  $[L \ p \ G]$  where  $p$  is any key chosen arbitrarily from  $A$ ;  $L$  contains all the keys from  $A$  that are less than or equal to  $p$  (but not the pivot itself), in any order;  $G$  contains all the keys from  $A$  that are greater than  $p$ , in any order.

**Conquer:** recurse on the two subarrays  $L$  and  $G$

**Combine:** no-op!

The algorithm is correct provided `PARTITION()` works: let's prove that it does.

# Requirement for PARTITION(A, p, r)

---

Given that  $p..r$  defines a valid subarray within  $A$ , PARTITION( $A$ ,  $p$ ,  $r$ ) must...

1. Rearrange the elements of  $A[p..r]$  into two (possibly empty) subarrays  $A[p..q-1]$  and  $A[q+1..r]$  such that
  - a. Each element of  $A[p..q-1]$  is less than or equal to  $A[q]$
  - b.  $A[q]$  is less than or equal to each element of  $A[q+1..r]$
2. Return  $q$ .

# Proof of correctness for PARTITION(A, p, r) [1]

---

PARTITION is based on a loop so we look for a loop invariant property. Based on our walk-through of PARTITION, we might capture its behaviour with this property:

Let  $P \stackrel{\text{def}}{=}$  At the beginning of each iteration of the FOR loop, for any array index  $k$ :

1. If  $p \leq k \leq i$ , then  $A[k] \leq x$ . // Region known to contain  $\leq$  pivot
2. If  $i+1 \leq k \leq j-1$ , then  $A[k] > x$ . // Region known to contain  $>$  pivot  
// No claim about the unprocessed region
3. If  $k=r$ , then  $A[k] = x$ . // The pivot has not been lost/corrupted!

We check initialisation, maintenance, termination...

## Proof of correctness for PARTITION(A, p, r) [2]

---

Let  $P \stackrel{\text{def}}{=} \text{At the beginning of each iteration of the FOR loop, for any array index } k:$

1. If  $p \leq k \leq i$ , then  $A[k] \leq x$ . // Region known to contain  $\leq$  pivot
2. If  $i+1 \leq k \leq j-1$ , then  $A[k] > x$ . // Region known to contain  $>$  pivot  
// No claim about the unprocessed region
3. If  $k=r$ , then  $A[k] = x$ . // The pivot has not been lost/corrupted!

At initialisation,  $i=p-1$  so #1 is vacuously true because no  $k$  are in range. The same applies to #2 and line 1 set the pivot ( $x$ ) to the last item so #3 is true.

$\Rightarrow P$  holds (is true) when PARTITION() begins.

## Proof of correctness for PARTITION(A, p, r) [3]

---

Let  $P \stackrel{\text{def}}{=} \text{At the beginning of each iteration of the FOR loop, for any array index } k:$

1. If  $p \leq k \leq i$ , then  $A[k] \leq x$ . // Region known to contain  $\leq$  pivot
2. If  $i+1 \leq k \leq j-1$ , then  $A[k] > x$ . // Region known to contain  $>$  pivot  
// No claim about the unprocessed region
3. If  $k=r$ , then  $A[k] = x$ . // The pivot has not been lost/corrupted!

For the maintenance step, we need a case split on whether we enter the IF.

We don't enter the IF when  $A[j] > x$ . The only change is to increment  $j$ . #1 and #3 are unchanged so continue to hold. #2 holds as the item newly in range is  $> x$ .

# Proof of correctness for PARTITION(A, p, r) [4]

---

Let  $P \stackrel{\text{def}}{=} \text{At the beginning of each iteration of the FOR loop, for any array index } k:$

1. If  $p \leq k \leq i$ , then  $A[k] \leq x$ . // Region known to contain  $\leq$  pivot
2. If  $i+1 \leq k \leq j-1$ , then  $A[k] > x$ . // Region known to contain  $>$  pivot  
// No claim about the unprocessed region
3. If  $k=r$ , then  $A[k] = x$ . // The pivot has not been lost/corrupted!

If we do enter the IF then  $A[j] \leq x$  and we swap  $A[i]$  and  $A[j]$ . The swap ensures #1 still holds even though  $i$  has been incremented. #2 still holds because the item in  $A[j-1]$  was previously known to satisfy #2. #3 is unchanged.

$\Rightarrow$  the Maintenance step preserves property  $P$ .



# Proof of correctness for PARTITION(A, p, r) [5]

---

Let  $P \stackrel{\text{def}}{=} \text{At the beginning of each iteration of the FOR loop, for any array index } k:$

1. If  $p \leq k \leq i$ , then  $A[k] \leq x$ . // Region known to contain  $\leq$  pivot
2. If  $i+1 \leq k \leq j-1$ , then  $A[k] > x$ . // Region known to contain  $>$  pivot  
// No claim about the unprocessed region
3. If  $k=r$ , then  $A[k] = x$ . // The pivot has not been lost/corrupted!

When the loop terminates,  $j=r$  (so the unprocessed region is empty). All values up to and including position  $i$  were  $\leq x$  and positions  $i+1..r-1$  were  $> x$  at the end of the previous iteration so when we do the final swap of  $A[i+1]$  and  $A[r]$ , we achieve the post-condition for the PARTITION procedure.

We actually over-achieved the post-condition: all our items to the right of the pivot are *strictly* greater than the pivot but are only required to be  $\geq$  pivot.

## Other ways to express Quicksort, e.g. in OCaml

---

```
let rec qs = function
  | [] -> []
  | pivot::rest -> (List.filter ((>=) pivot) rest |> qs)
                    @ pivot ::
                      (List.filter ((<) pivot) rest |> qs);;

qs [2; 5; 1; 3; 8; 2; 4];;

: int list = [1; 2; 2; 3; 4; 5; 8]
```

*fun x -> pivot >= x*

*Reverse apply*

## Little Demo [1]

---

```
let gen_cmp () =  
  let c = ref 0 in  
  let read_c () = !c in  
  let cmp (a : int) b = (c := !c+1; a <= b) in  
  (cmp, read_c)
```

```
let rec qs cmp = function  
  | [] -> []  
  | x::xs -> ((List.filter (cmp x) xs) |> qs cmp)  
             @ x ::  
             ((List.filter (fun y -> not(cmp x y)) xs) |> qs cmp)
```

## Little Demo [2]

---

```
let perm l =
  let rec perm c = function
    | [] -> [[]]
    | _ when c = 0 -> []
    |  $\bar{x}::xs$  -> (perm (List.length xs) xs
                  |> List.map (fun l ->  $x::l$ ))
                @ perm (c-1) (xs@[x]))
  in perm (List.length l) l

let stimulus = [1;2;3;4;5] in
List.map (
  fun l ->
    let (c,r) = gen_cmp() in
    ( qs c l ; r() $\sqrt{2}$  )
) (perm stimulus);;
```

# Other ways to express Quicksort, e.g. with Hoare-Partition

---

HOARE-PARTITION(A, p, r)

1     $x = A[p]$

2     $i = p - 1$

3     $j = r + 1$

4    **while** true

5        **repeat**  $j = j - 1$

6        **until**  $A[j] \leq x$

7

**repeat**  $i = i + 1$

8

**until**  $A[i] \geq x$

9

**if**  $i < j$

10

          swap( $A[i]$ ,  $A[j]$ )

11

**else**

12

**return**  $j$

# Performance of QUICKSORT(A, p, r)

---

- We usually take the number of key-comparisons as the unit of cost.
  - These occur within PARTITION()
- Although integer “ $\leq$ ” is cheap on most CPUs, substantially more CPU work might be required to compare values of another data type, e.g. strings, so comparisons can quickly dominate the cost.
- We don’t count the cost of adding 1 to integers (“ $i = i + 1$ ”, nor in the machine code implementation of the FOR loop) because there is a fixed number of CPU instructions *per comparison* so counting them would be equivalent to increasing a multiplicative constant factor.
- It all depends on how PARTITION() splits the subarray...

# Best-case performance of QUICKSORT(A, p, r) [1]

---

The best, worst and average cases are different for QUICKSORT.

Best case:

- Every time we pick a pivot, PARTITION happens to split range p..r “in half”.
- More precisely, partitioning  $n$  items will yield one subproblem of size  $\text{floor}(n/2)$  and another of size  $\text{ceil}(n/2)-1$  (remember the pivot is excluded from both).
- PARTITION sweeps through the subarray performing one comparison per key

$$T(1) = 1$$

$$T(n) = 2 T(n/2) + \textcircled{k n}$$

PARTITION(A, p, r)  $\in \Theta(n)$   
where  $n = r - p + 1$

Where did the floor and ceil go?!

## Best-case performance of QUICKSORT(A, p, r) [2]

---

$$T(1) = 1$$

$$T(n) = 2 T(n/2) + k n$$

Applying The Master Theorem,  $a=2$ ,  $b=2$ ,  $f(n) = kn$ . Calculate  $n^{\log_b a} = n$ .

Noting that  $f(n) \in \Theta(n^{\log_b a}) = \Theta(n)$ , case 2 applies and we read off the solution:

$$T(n) \in \Theta(n \lg n)$$



# Ratio-splitting performance of QUICKSORT(A, p, r) [1]

---

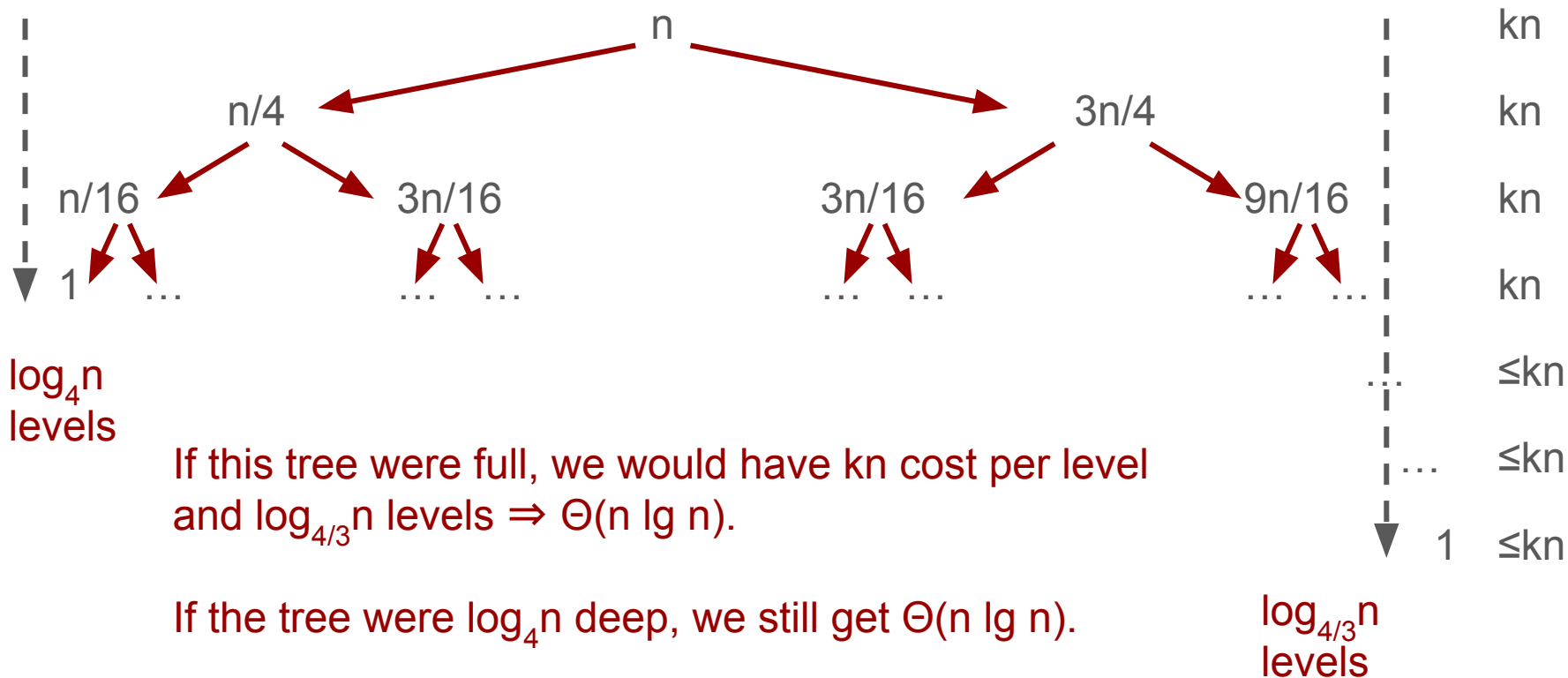
In fact, we can see that The Master Theorem would be satisfied if we split in any ratio, even 1% – 99%! Consider a 25% – 75% split at each level.

$$T(1) = 1$$

$$T(n) = T(n/4) + T(3n/4) + k n$$

Now let's draw the recursive calls and calculate the total cost.

# Ratio-splitting performance of QUICKSORT(A, p, r) [2]



## Ratio-splitting performance of QUICKSORT(A, p, r) [3]

---

If this tree were full and had the deepest depth right across, we would have  $kn$  cost per level and  $\log_{4/3} n$  levels  $\Rightarrow \Theta(n \lg n)$ .

If the tree were full and had the shallowest depth right across, we would have  $kn$  cost per level and  $\log_4 n$  deep  $\Rightarrow \Theta(n \lg n)$ .

It is easy to generalise and observe that any *ratio* split between the two sides will result in  $\Theta(n \lg n)$  performance.

This is instructive: it tells us to aim for a ratio split when we design divide and conquer algorithms because those are likely to be performant.

# Worst-case performance of QUICKSORT(A, p, r) [1]

---

Worst case:

- Every time we pick a pivot, PARTITION happens to split 'n' items into zero on one side, the pivot, and n-1 on the other side.
- In other words, the pivot is either the largest or smallest key in the subarray.
- (In)Famously, one such situation is when the input is already sorted.

$$T(1) = 1$$

$$T(n) = T(n-1) + T(0) + k n = T(n-1) + k n$$

## Worst-case performance of QUICKSORT(A, p, r) [2]

---

$$T(1) = 1$$

$$T(n) = T(n-1) + k n$$

We solve using the substitution method: substitute the definition of  $T(n)$  into itself and spot the pattern.

$$T(n) = T(n-1) + kn = (T(n-2) + k(n-1)) + kn = ((T(n-3) + k(n-2)) + k(n-1)) + kn = \dots$$

The number of times we can subtract 1 from  $n$  until we hit the base case is  $n-1$  so we have an arithmetic progression of  $n$  terms, first term 1 (the base case), and sum being  $kn(n+1)/2 \in \Theta(n^2)$ .

## Constant-splitting performance of QUICKSORT(A, p, r)

---

It doesn't have to be "0 and n-1"! Splitting  $n$  keys into  $[x, 1 \text{ pivot}, (n-x-1)]$  for any constant,  $x$ , gives the same outcome.

$$T(1) = 1$$

$$T(n) = T(n-x-1) + T(x) + k n$$

Whatever value 'x' is,  $T(x)$  is some constant so disappears into the constant factor. The arithmetic progression has a step size of 'x' instead of 1, but the sum is still in  $\Theta(n^2)$ . This is also instructive: avoid divide-and-conquer algorithms that split off a constant size subproblem from the rest!

# Order Statistics

---

The  $i^{\text{th}}$  **order statistic** is the  $i^{\text{th}}$  smallest value in a set of  $n$  elements (NB: 'set' implies there are no duplicates). Finding it is known as the **selection problem**:

**Input:** a set,  $A$ , of  $n$  (distinct) numbers and an integer,  $i$  such that  $1 \leq i \leq n$ .

**Output:** the element  $x \in A$  that is larger than exactly  $i-1$  other elements of  $A$ .

One obvious way to get the  $i^{\text{th}}$  order statistic is to sort the sequence and read off the value in the  $i^{\text{th}}$  position. We know this can be achieved in guaranteed  $\Theta(n \lg n)$  time using MERGE-SORT, dominated by the sorting step.

Can we do better?

# Minimum and Maximum

---

The minimum is the first order statistic and the maximum is the  $n^{\text{th}}$ . A simple linear scan can find either (or both) in  $\Theta(n)$  time, performing  $n-1$  comparisons.

MINIMUM (A)

```
1  min = A[1]
2  for i=2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

MAXIMUM (A)

```
1  max = A[1]
2  for i=2 to A.length
3      if max < A[i]
4          max = A[i]
5  return max
```

**Challenge:** find *both* min and max using at most  $3 \text{ floor}(n/2)$  comparisons.



## QUICKSELECT(A, p, r, i)

---

```
1  if p == r
2      return A[p]                // region size is one
3  q = PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k
6      return A[q]                // pivot is in position i
7  else if i < k return QUICKSELECT(A, p, q-1, i)
8  else return QUICKSELECT(A, q+1, r, i-k)
```

# How does QUICKSELECT( $A, p, r, i$ ) work?

---

- Similar to QUICKSORT() except we recurse only on the side containing the  $i^{\text{th}}$  element
- Notice the stopping conditions:
  - If the subarray has size one then the only element has to be the one we want
  - If, by luck, the pivot ends up in position  $i$ , we can return it because the pivot is in the correct place in sorted order.
- Notice two further optimisations we could make:
  - If the pivot ended up in position  $i-1$ , we could return MINIMUM() on  $A[q+1..r]$
  - If the pivot ended up in position  $i+1$ , we could return MAXIMUM() on  $A[p..q-1]$
  - Each of these would perform  $x-1$  comparisons on the  $x$  (many) items in the subarray and guarantee to return a result; that beats continuing QUICKSELECT(), which would perform  $x-1$  comparisons to partition about the next pivot and only terminate if, by luck, the pivot is the  $i^{\text{th}}$ .

## Worst-case cost of QUICKSELECT(A, p, r, i)

---

In the worst case, PARTITION() might split any subarray into 0 and  $n-1$  elements each time. We recurse on the non-zero partition.

$$T(1) = 1$$

$$T(n) = T(n-1) + k n$$

Using the substitution method, we find that the worst case cost  $\in \Theta(n^2)$ .

# Improving QUICKSORT and QUICKSELECT

---

There are several standard improvements discussed in algorithms literature.

1. Randomise the input data before starting
2. Take out all values equal to the pivot
3. Pick the pivot randomly
4. Median-of-Three pivot
5. Median-of-Medians pivot

# Randomise the input data before starting

---

Perform  $O(n)$ -many random swaps to 'scramble' the input array. This can incur substantial extra cost to generate good-quality random numbers but makes all input permutations (roughly) equally likely: good if pathological inputs are likely!

If the input data was already equally likely to be in any permutation, this does not make the worst case any less likely to occur, nor make it any less expensive when it does: for every worst case avoided, another becomes a worst case as a result of permuting an initially scrambled input order into a worst-case ordering.

This does make it hard or impossible to generate an input for QUICKSELECT that will hit the worst case. This might be an important security consideration.

 For QUICKSORT, it's still trivial to provide input that hits the worst case: make every value in the array be the same so every pivot is a worst case!

# Take out all values equal to the pivot

---

When partitioning, split into three regions with the middle region containing all the values equal to the pivot.

This is especially effective if the input might be a long array containing a small number of distinct values (i.e. lots of duplicates).

This also turns the “all values are the same” worst case into a best case so it can be used in combination with the previous technique.

```
let rec qs = function
| [] -> []
| pivot::rest -> (List.filter ((>) pivot) rest |> qs)
  @ (List.filter (|=) pivot) rest
  @ (List.filter ((<) pivot) rest |> qs);;
```

## Pick the pivot randomly

---

Instead of using the last key in the subarray as the next pivot (or the first in the OCaml implementation), pick a pivot at random.

This also incurs the cost of generating good quality random numbers but only needs as many as the number of levels in the recursion.

If some input orderings are more likely than others, and the more likely ones happen to lead to worst case performance, then, like randomising the input, this can reduce the probability of hitting a worst case.

# Median-of-Three pivot

---

If the subarray has length 1, we return. For length 2, we compare and swap if necessary.

Otherwise, we pick three items, systematically or randomly, and use the median as the pivot.

- If the items are distinct (as is always the case for QUICKSELECT), we have reduced the likelihood of hitting the worst case from  $\sim 2\text{-in-}n$  to  $\sim 2\text{-in-}n^2$ .
- Because there is always at least one item larger than the pivot and at least one smaller, we guarantee to take two items (that and the pivot) out of contention in each split, so we halve the number of splits.



# Median-of-Medians pivot [1]

---

Earlier, we learnt to aim for a *ratio split* in divide-and-conquer algorithms. The median-of-three idea only guaranteed at least 1 key on either side of the pivot: a *constant-and-the-rest* type of split.

To get a ratio split, we change PARTITION() to work as follows.

1. Consider the input subarray to be groups of 5 keys (the last may be smaller)
2. Find the median of each group of 5 by insertion sort and taking the 3<sup>rd</sup>
3. Find the median of those medians using QUICKSELECT and use it as the pivot

This helps QUICKSELECT (and QUICKSORT on distinct items). Let's see why...

# Definition of 'Median'

---

For an odd number,  $n$ , of distinct elements, the **median** is in position  $(n+1)/2$  when they are in sorted order. Equivalently,  $(n-1)/2$  elements are smaller than it.

For an even number,  $n$ , of distinct elements, the **lower median** and **upper median** are the two on either side of the halfway point, when they are in sorted order.

In both cases, the lower median is in position  $\text{floor}((n+1)/2)$  and the upper median is in position  $\text{ceil}((n+1)/2)$ .

In algorithms literature, the unqualified term 'median' refers to the lower median.



In many branches of mathematics, the median of an evenly-sized collection is the average of the lower and upper medians. Not so in algorithms!

## Median-of-Medians pivot [2]

---

The final pivot is the median of the  $\text{ceil}(n/5)$  medians.

Half of the medians (ceiling of one half if there's an even number of medians as we use the *lower median*) must be greater than pivot:  $\text{floor}(\frac{1}{2} \text{ceil}(n/5))$  keys.

For each median greater than the pivot, two of the five in its group are even greater (except the last group, which might be fewer than 5; and the group from which the pivot came when there's an odd number of medians).

So  $3(\text{ceil}(\frac{1}{2} \text{ceil}(n/5)) - 2) \geq 3n/10 - 6$  keys are definitely greater than the pivot.

Even the worst case is  $7n/10 + 6$  on one side and  $3n/10 - 6$  on the other.

# Median-of-Medians pivot [3]

---

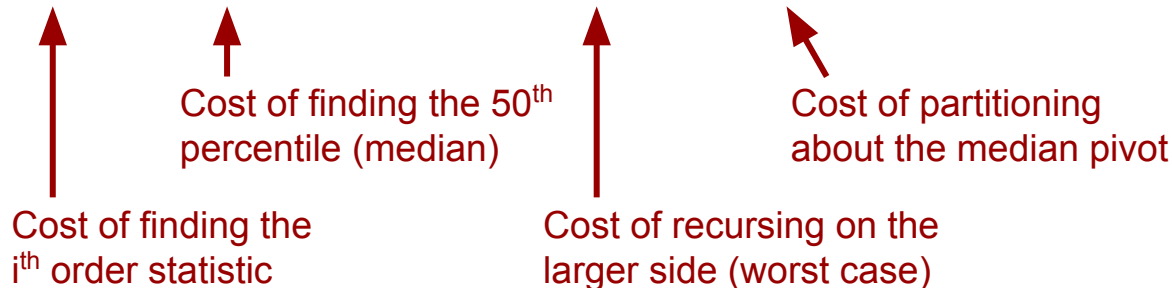
Finding the median of 5 numbers is a constant-time operation:  $\Theta(1)$ .

Suppose the cost of any problem with fewer than 140 elements is constant.  
(Bear with me...)

The worst-case recurrence for QUICKSELECT() would become...

$$T(n) = k_1 \quad \text{if } n < 140$$

$$T(n) = T(\text{ceil}(n/5)) + T(7n/10 + 6) + k n \quad \text{otherwise}$$



# Another way to solve recurrence equations

---

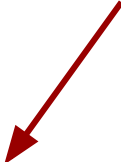
**Guess!** Then substitute your guess to see that it checks out.

Example:

$$T(1) = 1$$

$$T(n) = T(n/2) + kn^2$$

This is what we need to verify:  
this is what our guess means so  
if it's true, our guess is correct!



**Guess** that  $T(n) \in O(n^2)$ , i.e. for all  $n > n_0$ ,  $T(n) \leq cn^2$ , for some constant  $c > 0$ .

**Verify:**  $T(n) = T(n/2) + kn^2 \leq cn^2/4 + kn^2$ . This is  $\leq cn^2$  provided  $k \leq 3c/4$ , so there is a constant,  $c > 0$  satisfying this (specifically any value at least  $4k/3$ ).

## Median-of-Medians pivot [4]

---

We guess that the answer is  $\Theta(n)$ , i.e. the cost is  $cn$  for some real constant,  $c > 0$ . Substituting our guess into  $T(n) = T(\text{ceil}(n/5)) + T(7n/10 + 6) + k n$  yields...

$$\begin{aligned} T(n) &\leq c \text{ceil}(n/5) + c(7n/10 + 6) + kn \\ &\leq cn/5 + c + 7cn/10 + 6c + kn \\ &= 9cn/10 + 7c + kn \\ &= cn + (-cn/10 + 7c + kn) \end{aligned}$$

Our guess is correct if this is at most  $cn$ , which is when  $(-cn/10 + 7c + kn) \leq 0$ . Rearranging gives  $c \geq 10k(n/(n-70))$ . Since  $n \geq 140$ ,  $n/(n-70) \leq 2$  so any  $c \geq 20k$  validates our guess that  $T(n) \in \Theta(n)$ .



The cost of finding the  $i^{\text{th}}$  order statistic with QUICKSELECT is independent of  $i$ . This is not true of all strategies!

# QUICKSORT using Median-of-Medians?

---

Now we can find the  $i^{\text{th}}$  of  $n$  distinct elements in  $\Theta(n)$  time, we can rework the  $\text{PARTITION}(A, p, r)$  step of  $\text{QUICKSORT}(A, p, r)$  to use  $\text{QUICKSELECT}()$  to choose the best pivot (the median) in each step. This still only costs us  $\Theta(n)$ !

Provided the values to be sorted are distinct, the **worst case** cost for  $\text{QUICKSORT}(A, p, r)$  is given by the modified recurrence equation:

$$T(1) = 1$$

$$T(n) = 2 T(n/2) + k n$$

Guaranteed even split since all pivots are medians

PARTITION() is still  $\Theta(n)$

$$T(n) \in \Theta(n \lg n) \quad (\text{We solved this recurrence relation for MERGE-SORT.})$$

# HEAPSORT(A) [1]

---

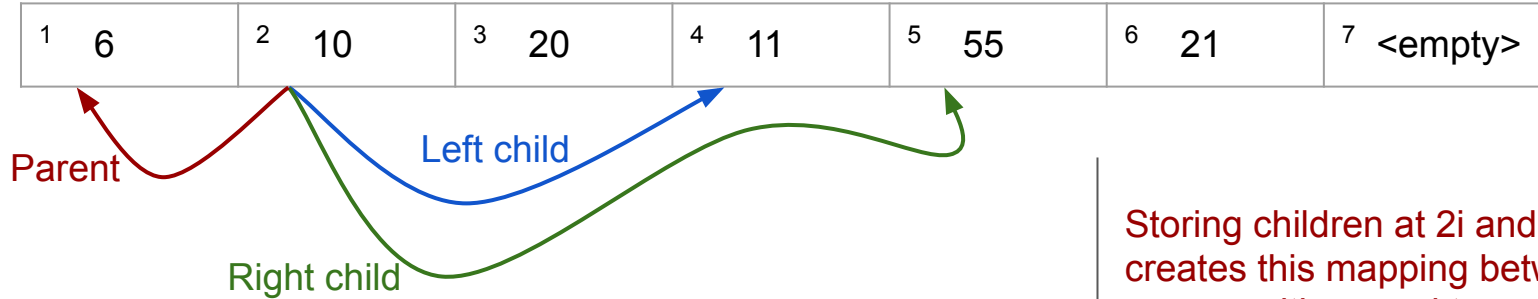
The (binary) heap is a data structure. It is sometimes helpful to think about a heap as a binary tree. However, to achieve the headline asymptotic costs, we need to store a heap as an array.

Two defining properties of a MIN-HEAP / MAX-HEAP:

1. **Structural property:** considered as a tree, a heap is a full tree, except possibly for the lowest level which is filled from left to right.
2. **Ordering property:** in a min-heap, every node holds a lesser-or-equal key than its child(ren); in a max-heap, nodes have greater-or-equal keys than their child(ren).



# HEAPSORT(A) [2]

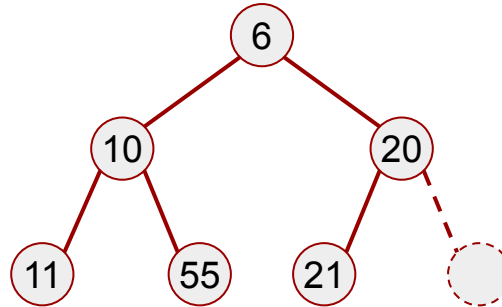


A.length = 7

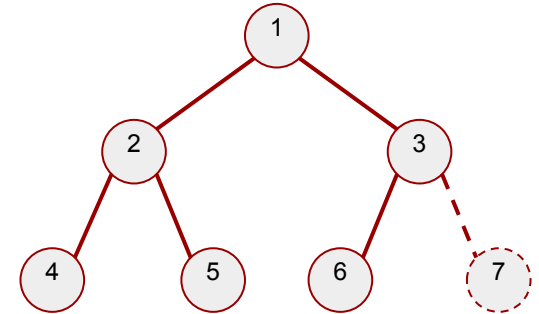
A.heap\_size = 6

This is a MIN-HEAP.

The smallest value in the collection, 6, is in the root.



Storing children at  $2i$  and  $2i+1$  creates this mapping between array positions and tree positions.



# HEAPSORT(A) [3]

---

## Array thinking

The root of a heap is always stored in  $A[1]$ .

The left child of the node in array position  $i$  is in position  $2i$ ; the right child is in  $2i+1$ .

The parent of the node in array position  $i$  is in position  $\text{floor}(i/2)$ .

A child that doesn't exist, is identified by  $2i$  or  $2i+1$  being 'off the end' of the `heap_size`.

We can tell when a node has no parent because  $\text{floor}(i/2) = 0$  (only the root has no parent).

## Tree thinking

The root is pointed to by an external variable.

Each node contains pointers to its left and right children.

Each node contains a pointer to its parent node.

When a child doesn't exist, we have a null value in the parent's left or right child pointer.

When a parent doesn't exist, we have a null value in the node's parent pointer.

# HEAPSORT(A) [4]

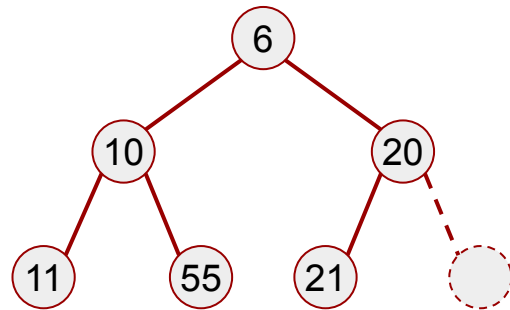
---

Heaps are **semi-structures**.

Semi-structures are cheaper to build than fully-structured data structures.

Here, the semi-structuring is about the partial sort order:

- The smallest item is only in one place: the root
- The second smallest is in one of two places (one of root's children)
- The third smallest is in one of three places (root's other child, or either child of the second smallest)



# HEAPSORT(A) [5]

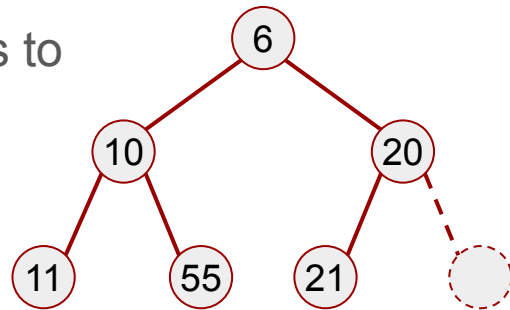
---

Semi-structures are clever because they support operations to change the heap cheaply, and maintain all the properties of the semi-structure.

Max-Heap operations:

- MAX-FULL-HEAPIFY in  $O(n)$  time, MAX-REHEAPIFY in  $O(\lg n)$  time
- MAX-PEEK in  $O(1)$  time
- MAX-INSERT, MAX-EXTRACT, INCREASE-KEY in  $O(\lg n)$  time << Priority Queue ADT!

Symmetrically, but with DECREASE-KEY, for Min-Heaps.

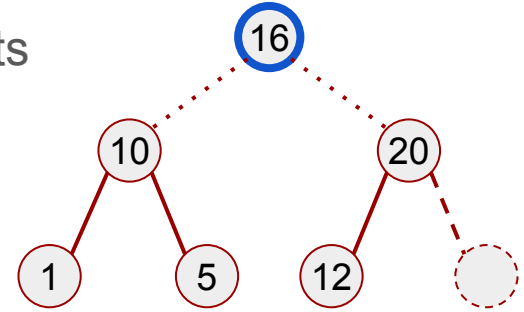


# MAX-REHEAPIFY(A, i) [1]

---

MAX-REHEAPIFY assumes that everything below node  $i$ , in its left and right children (if they exist), are valid MAX-HEAPS.

Its purpose is to build a single, large MAX-HEAP out of two, existing MAX-HEAPS and one extra key.



# MAX-REHEAPIFY(A, i) [2]

Compare the new key to the roots of the two existing heaps.

If the new key is the largest, we're done.

If not, swap with the larger of the two sub-heap roots and recurse on the node you swapped with.

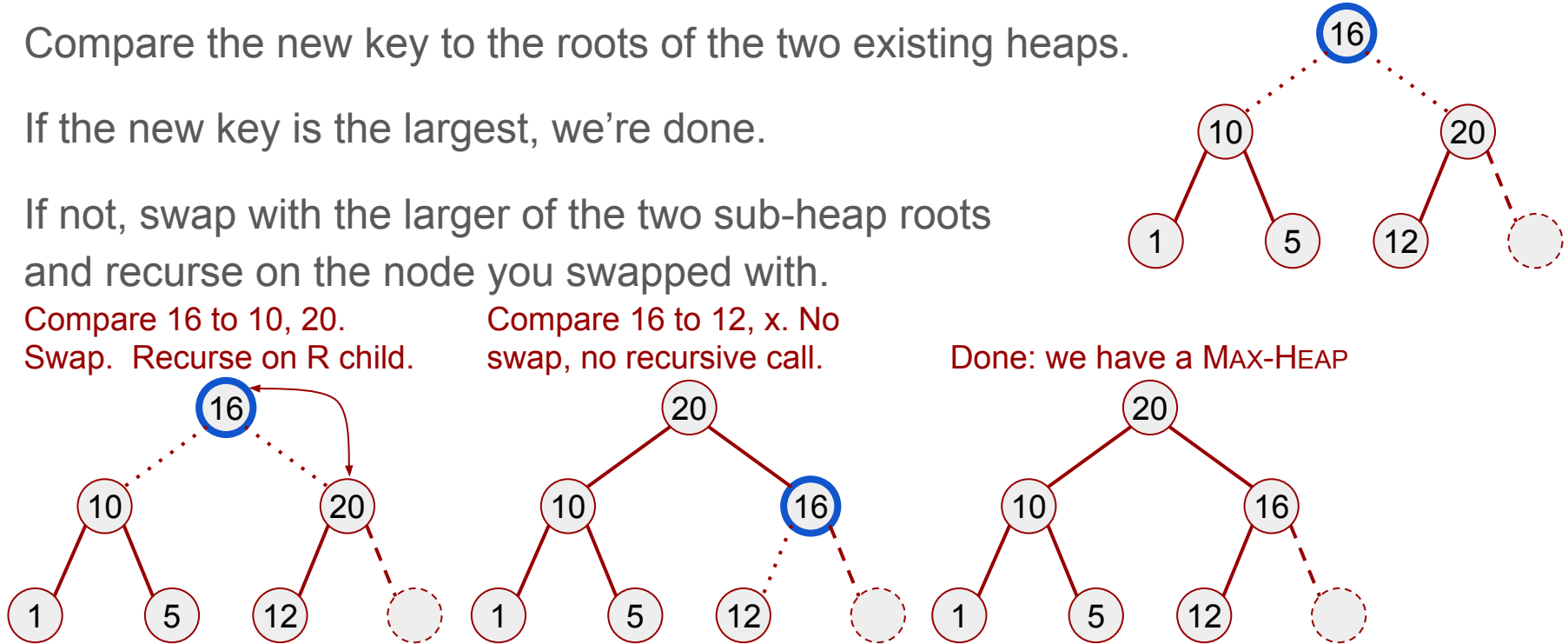
Compare 16 to 10, 20.

Swap. Recurse on R child.

Compare 16 to 12, x. No

swap, no recursive call.

Done: we have a MAX-HEAP



## MAX-REHEAPIFY(A, i) [3]

---

```
1  l = 2i
2  r = 2i + 1
3  largest = (l ≤ A.heap_size && A[l] > A[i]) ? l : i
4  if (r ≤ A.heap_size && A[r] > A[largest]) largest = r
5  if largest != i
6      swap(A[i], A[largest])
7      MAX-REHEAPIFY(A, largest)
```

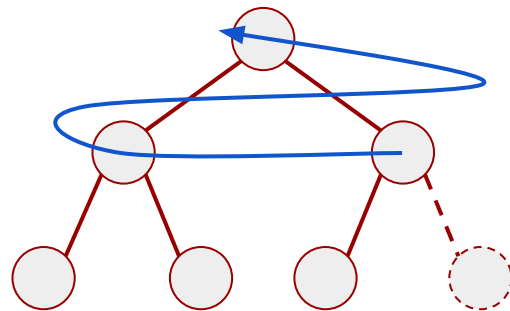
# MAX-FULL-HEAPIFY(A) [1]

---

MAX-FULL-HEAPIFY notes that the bottom-level leaves are valid MAX-HEAPS.

It calls MAX-REHEAPIFY on the last node that has at least one child, then works its way back up to the root.

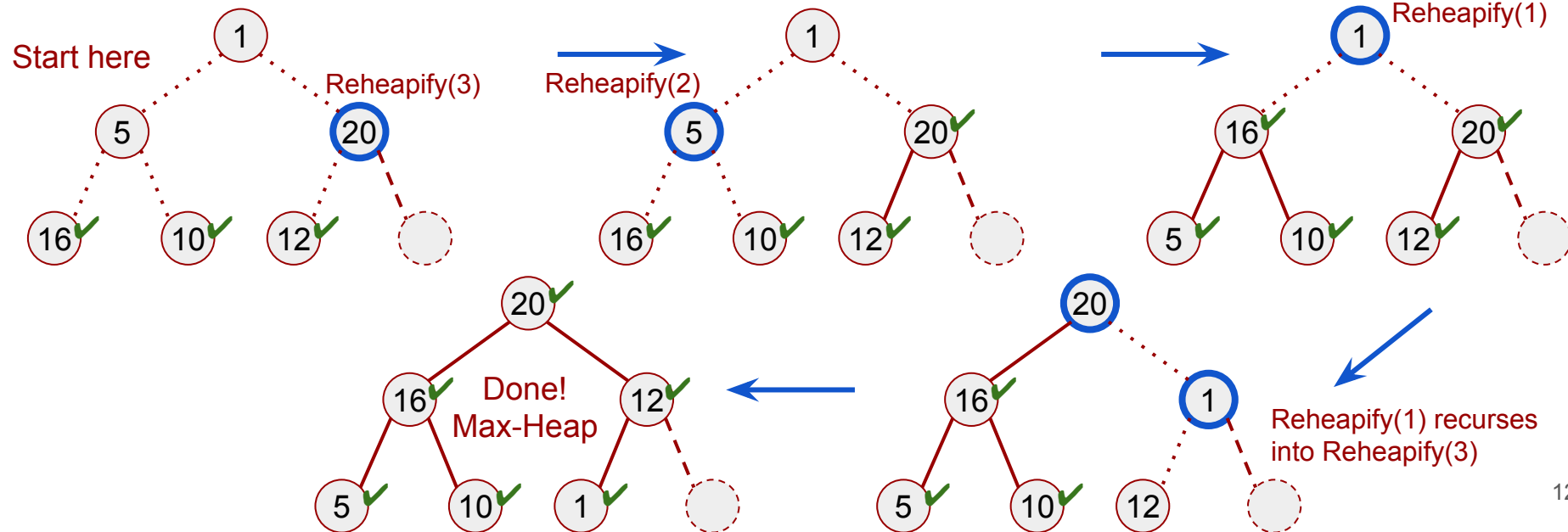
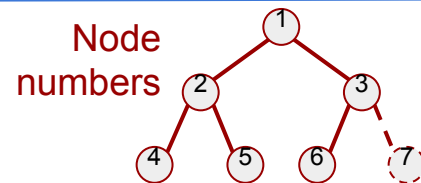
This is easy to do with the array representation and turns any array into a valid MAX-HEAP.





# MAX-FULL-HEAPIFY(A) [2]

```
1  A.heap_size = A.length
2  for i = floor(A.length/2) downto 1
3      MAX-REHEAPIFY(A, i)
```



## MAX-EXTRACT(A)

---

The largest key in a max-heap is the root. To extract the max, we cannot simply remove it because that would violate the defining structural property of a heap.

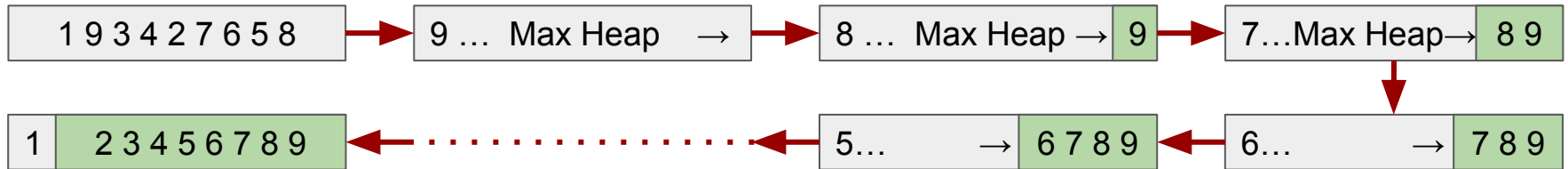
The only key we can remove without violating the structural property is the rightmost key on the bottom row.

We swap the root with the rightmost key from the bottom row and consider the heap to occupy one fewer array position ( $A.\text{heap\_size} = A.\text{heap\_size} - 1$ ).

The only 'damage' we have done to the structure is at the root so we call `MAX-REHEAPIFY(A, 1)` to fix-up the root node. Cost is  $O(\lg n)$  (proof: tree height).

# HEAPSORT(A) [6]

```
1  MAX-FULL-HEAPIFY (A)
2  for i = A.length downto 2
3      swap (A[1], A[i])
4      A.heap_size = A.heap_size - 1
5      MAX-REHEAPIFY (A, 1)
```



# Cost of MAX-FULL-HEAPIFY(A) [1]

---

## Best case

- The initial array order is compatible with the heap ordering property.
- MAX-FULL-HEAPIFY considers about  $n/2$  keys, performs 2 comparisons per key, but no swaps and no recursive calls.
- This is clearly  $\Theta(n)$ .

## Worst case

- Occurs when every comparison results in a swap and every recursive call also results in a swap and another recursive call.

# Cost of MAX-FULL-HEAPIFY(A) [2]

	ITEMS	COMPARISONS	TOT	WORK
X	1	6	1	x 6
X X	2	4	2	x 4
X X X X	4	2	4	x 2
X X X X X X X X	8	0	8	x 0

Let  $L = 1 + \text{floor}(\log_2 n)$  be the number of levels in the tree. The sum of the TOT WORK column is

$$\sum_{i=0}^{L-1} 2^{L-1-i} \times 2i = \dots = \dots \in \Theta(n)$$

<< Supervision exercise!

$i=0$  is the leaf level.

# Cost of HEAPSORT(A)

---

We call MAX-FULL-HEAPIFY, which costs  $\Theta(n)$ .

Then,  $n-1$  times, we perform a swap costing  $\Theta(1)$  and we call MAX-REHEAPIFY on the diminished heap. MAX-REHEAPIFY costs  $O(\lg n)$  when called on a heap\_size of  $n$  keys. Total cost is:

$$\begin{aligned} T(n) &= k_1 n + k_2 \lceil \lg(n) \rceil + k_2 \lceil \lg(n-1) \rceil + k_2 \lceil \lg(n-2) \rceil + \dots + 1 \\ &\leq k_1 n + k_2 \lg(n) + k_2 \lg(n-1) + k_2 \lg(n-2) + \dots + 1 + k_2 n \\ &= 1 + (k_1 + k_2)n + k_2 \lg(n!) \\ &\leq 1 + (k_1 + k_2)n + k_2(n \lg n - n) && \text{(Stirling's Approximation)} \\ &\in O(n \lg n) \end{aligned}$$

# Sorting in Linear Time

---

Our previous sorting methods worked for any range and distribution of input data, and achieved a sorted order by comparing elements against each other. These are collectively known as **comparison sorts** and it can be shown that the minimum number of comparisons required to sort  $n$  keys is  $\Omega(n \lg n)$ .

If we know something about the input data, we can often do better.

1. Counting sort:  $n$  inputs in the range  $[0..k]$  where  $k \in O(n)$
2. Radix sort: sorting  $d$ -digit numbers
3. Bucket sort: sorting data that is uniformly distributed over  $[0,1)$

# COUNTING-SORT(A, B, k)

```
1  let C = new Array[0..k]
2  for i = 0 to k
3      C[i] = 0
```

} Initialise counts to 0

```
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  for i = 1 to k
7      C[i] = C[i] + C[i-1]
```

Update C to store where instances of i should start in the output

Count instances of each value in A

```
8  for j = A.length downto 1
9      B[C[A[j]]] = A[j]
10     C[A[j]] = C[A[j]] - 1
```

Populate the output array: xx00x1xxx222xx3...  
by filling in the values 0..k from the right.

A: input data

B: array into which to write the output

k: top limit of the range of values

C[i] holds the number of instances of value i in the input array, A



## Cost of COUNTING-SORT(A, B, k)

---

Initialising the C array takes  $\Theta(k)$  time (lines 1,2,3).

Counting items in the A array takes  $\Theta(n)$  time, where  $n=A.length$  (lines 4,5).

Converting the count of key  $i$  to the index of the last instance of  $i$  in the output takes  $\Theta(k)$  time (lines 6,7).

Populating the output takes  $\Theta(n)$  time (lines 8,9,10).

Overall  $\Theta(k+n)$ .

# RADIX-SORT(A, d)

---

```
1  for i = 1 to d
2      sort array A on digit i with any stable sort
```

Note: digit 1 is the least significant digit.

# Stable sorts

---

A stable sort is one that guarantees to preserve the order of inputs when their sort keys are equal.

This is useful if you want a **secondary sort** key, e.g. sort exam results by mark but if two people have the same mark then list them in alphabetical order.

- Sort by name first – any kind of sort will do
- Now use a stable sort to re-sort by mark

When we re-sort by mark, the order of people with equal marks will be preserved, i.e. still in alphabetical order – as required.

# RADIX-SORT(A, d) in operation

---

123	241	123	123
277	451	241	148
149	123	148	149
148	277	149	241
241	148	451	277
451	149	277	451

The three numbers in bold are (one example of) relying on stable sorting. All three have 1 in their most significant place so the third stage of sorting has no preference for which goes first. It is important that we defer the decision to whatever the second sort did: 123 before 148 and 149. In turn, the second stage has no preference for which '4' comes first and defers that decision to whatever the first sorting stage did: 8<9 so 148 came before 149.

## Cost of RADIX-SORT(A, d)

---

To sort  $n$  numbers of  $d$  digits each, where each digit can take on one of  $k$  different values ( $0..k-1$ ),  $\text{RADIX-SORT}(A, d) \in \Theta(d(n+k))$ .

Each stable sort is applied to  $n$  keys using a key-range of  $0..k-1$ . COUNTING-SORT is the obvious way to achieve that, taking  $\Theta(n+k)$  time each.

# BUCKET-SORT(A)

---

```
1  let n = A.length, B = new Array[0..n-1]
2  for i = 0 to n-1
3      B[i] = empty_list
4  for i = 1 to n
5      insert A[i] into list B[floor(n*A[i])]
6  for i = 0 to n-1
7      INSERTION-SORT(B[i])
8  concatenate B[0], B[1], .. B[n-1] (in that order)
```

## Cost of BUCKET-SORT(A)

---

All the steps are obviously linear in  $n$  except the calls to INSERTION-SORT, which needs a closer look.

We know that INSERTION-SORT takes  $O(n^2)$  time on inputs of length  $n$ . We have  $n$  buckets with  $n_i$  keys in each bucket  $0 \leq i \leq n$ .

$$T(n) = \Theta(n) + \sum_0^{n-1} O(n_i^2)$$

*Unfortunately, you cover the maths required to solve this in Easter Term!*

...and it turns out that  $T(n) = \Theta(n) + n O(2-1/n) = \Theta(n)$ .

# Summary of Algorithms 1 so far [1]

---

Methods to solve recurrence relations:

- Guess and verify
- Substitute and spot pattern, including the tree method to help spot patterns
- The Master Theorem

Algorithm designs:

- Incremental
- Divide and Conquer
- More to come in the next part of Algorithms 1



# Summary of Algorithms 1 so far [2]

---

Growth orders:

- $f(n) \in o(g(n))$   $f(n)$  has strictly less rapid growth than  $g(n)$
- $f(n) \in O(g(n))$   $f(n)$ 's growth is upper-bounded by  $g(n)$ : *“at most as fast as”*
- $f(n) \in \Theta(g(n))$   $f(n)$  grows within a constant factor at the same rate as  $g(n)$
- $f(n) \in \Omega(g(n))$   $f(n)$ 's growth is lower-bounded by  $g(n)$ : *“at least as fast as”*
- $f(n) \in \omega(g(n))$   $f(n)$  has strictly more rapid growth than  $g(n)$

# Summary of Algorithms 1 so far [3]

Technique	Worst case	Average case	Stable	In-place	Notes
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓	Tight loop, fast for small n
MERGE-SORT	$\Theta(n \lg n)$	$\Theta(n \lg n)$	✓		Bottom-up Merge-Sort is in-place
HEAP-SORT	$O(n \lg n)$			✓	
QUICKSORT	$\Theta(n^2)$	$\Theta(n \lg n)$ expected		✓	
QUICKSORT (MoM)	$\Theta(n \lg n)$	$\Theta(n \lg n)$		✓	
COUNTING-SORT	$\Theta(k + n)$	$\Theta(k + n)$	✓		
RADIX-SORT	$\Theta(d(n + k))$	$\Theta(d(n + k))$	✓		
BUCKET-SORT	$\Theta(n^2)$	$\Theta(n)$ average	✓		

# Algorithms 1

---

## Section 2: Strategies for Algorithm Design

# Dynamic Programming

---

Divide-and-Conquer split a problem into subproblems that did not overlap.

Dynamic Programming is useful when subproblems do overlap.

Note: 'Programming' does not refer to what we call 'coding' today! 'Programming' has another, historical meaning, referring to methods structured in some way around a table that is progressively filled in.

# Problems amenable to Dynamic Programming

---

**Optimal substructure** problems, usually minimising or maximising something.

Example: our problem is to minimise the total cost of a sequence that achieves some goal. We may perform operations A, B and C, with costs 1, 2 and 3 respectively. We consider three subproblems to achieve three slightly reduced goals (those where the only remaining step is A, B or C), and the optimal cost is  $\text{MIN}(\text{SUB}_A + 1, \text{SUB}_B + 2, \text{SUB}_C + 3)$ . Optimal substructure means that the solution we want uses the optimal solution to one of  $\text{SUB}_A$ ,  $\text{SUB}_B$ ,  $\text{SUB}_C$ .

Often there are many, equally good solutions and we seek any one: we want *an* optimal solution, not *the* optimal solution.

# Four steps of Dynamic Programming

---

We need to identify and exploit the optimal substructure.

Typically four steps:

1. Characterise the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically bottom-up
4. If required, construct an optimal solution from the computed information

# Approaches to Dynamic Programming

---

## Top-Down

Start with the problem you want to solve, divide into subproblems and keep going until you reach base cases.

Requires stack space for the recursive call tree.

Only solve subproblems that are required for the original problem.

Avoid solving the same subproblem twice by **memoising** results in a table.

## Bottom-Up

Start with the base case(s) and solve every problem that combines them in one step, putting results into the **memo table** as you go.

Now solve two-step problems and add those to the table. Carry on until the desired problem is encountered and solved.

No recursive stack space required.

Solves subproblems that might not be useful.

# Problems with Dynamic Programming Solutions

---

**Longest Common Subsequence:** a subsequence is a given sequence with zero or more elements removed (not necessarily consecutively). LCS is the problem to find the longest subsequence present in both of two input sequences. This is very common in bioinformatics (See the Part II Bioinformatics course).

**Matrix Multiplication Chains:** minimise the distinct scalar multiplications required to multiply a chain of matrices (optimally exploit associativity).

**Unweighted Shortest Path:** find a path  $u \rightsquigarrow v$  consisting of the fewest edges.

**Virtual Machine Hosting Problem (VMHP):** (rod-cutting, thinly veiled...)



# Example of Dynamic Programming: VMHP

---

This is a modern take on the “rod-cutting problem”.

You have a server with  $n$  CPU cores and wish to subdivide it into one or more virtual machines. A table (example below) provides the value of virtual machines with varying numbers of CPU cores up to  $n$  (‘value’ might be the amount customers are prepared to pay for each size of virtual machine). Our task is to maximise the value that can be achieved by subdividing our  $n$  CPU cores into virtual machines.

Cores	1	2	3	4	5	6	7	8	9	10
Value	1	6	7	9	9	10	11	16	18	25

# Where to start?

---

If you cannot see where to start, working through a small example is often enough to understand the choice your algorithm needs to make.

E.g. if we have  $n=4$  CPU cores, we can divide in 8 ways:

4	(1 VM with 4 cores)	1,1,2
1,3	(1x 1-core VM + 1x 3-core VM)	1,2,1
2,2	(...)	2,1,1
3,1		1,1,1,1

# Value of these 8 solutions

---

Cores	1	2	3	4	5	6	7	8	9	10
Value	1	6	7	9	9	10	11	16	18	25

4 9

1,1,2 1+1+6=8

1,3 1+7=8

1,2,1 1+6+1=8

**2,2 6+6=12 << WINNER!**

2,1,1 6+1+1=8

3,1 7+1=8

1,1,1,1 1+1+1+1=4

## Recursive top-down: VMIFY(v, n)

v: table of values

```
1  if n == 0
2      return 0
3  q = -1
4  for i = 1 to n
5      q = max(q, v[i] + VMIFY(v, n-i))
6  return q
```

💡 For our table of values, we can never 'get stuck' in a subdivision but if that were to be possible then the -1 would be used (e.g. if VMs of size 3 were not possible).

Given n remaining cores, we consider calving off a VM with size i, for each valid i, and finding the optimal way to use the remaining n-i cores. The optimal way to use n cores is the max value of using 1+rest, 2+rest, 3+rest, ...

## Recursive top-down: VMIFY(v, n) [2]

---

It is not hard to see that the running time of VMIFY(v, n) is exponential in n:

$$T(1) = 1 \quad \text{and} \quad T(n) = 1 + \sum_{i=0}^{n-1} T(i) \quad \Rightarrow T(n) \in O(2^n)$$

This means we can only practically solve the problem for CPUs with a small number of cores. This is not so useful.

It is slow because sub-problems are solved over and over again. E.g. VMIFY(v, 2) is solved as part of VMIFY(v, 4) with  $i=2$  and as part of VMIFY(v, 3) with  $i=1$ .

Dynamic Programming to the rescue!

## Recursive top-down: VMIFY( $v$ , $n$ ) [3]

---

We introduce a memo table in which we write down the answer for every ' $n$ ' we have evaluated. We initialise the table to  $-\infty$  so we can distinguish values that we have worked out from those we have not (in this problem, values are always non-negative).

## MEMO-VMIFY( $v, n$ )

```
1  let m[0..n] = new Array
2  for i = 0 to n
3      m[i] =  $-\infty$ 
4  return MVMIFY-AUX( $v, n, m$ )
```



For our table of values, we can never 'get stuck' in a subdivision but if that were to be possible (e.g. if VMs of size 3 were not possible) then the -1 would be used.

## MVMIFY-AUX( $v, n, m$ )

```
1  if m[n]  $\geq -1$ 
2      return m[n]
3  if n == 0
4      q = 0
5  else
6      q = -1
7      for i = 1 to n
8          q = max(q,
9              v[i] + MVMIFY-AUX( $v, n-i, m$ ))
10 return q
```

## Bottom-Up Memoising VMIFY( $v$ , $n$ )

---

We can also demonstrate the bottom-up approach with our VMHP example.

If we work out the optimum value obtainable from 1 CPU, then 2 CPUs, etc. then, when we come to larger values of  $n$ , all the smaller problems that it might ever wish to refer to have already been solved.

We can do this with a FOR loop that increases from the smallest problems to the value  $n$  we are interested in.

Notice that our table,  $v$ , ensures that all smaller problems will eventually be used to solve larger problems. This is not guaranteed in general so we may waste work.



# BOTTOM-UP-VMIFY(v, n)

---

```
1  let m[0..n] = new Array
2  m[0] = 0
3  for j = 1 to n
4      q = -1
5      for i = 1 to j
6          q = max(q, v[i] + m[j-i])
7      m[j] = q
8  return m[n]
```



All smaller problems have already been solved and memoised by the time we need them to solve larger problems.



# Summary of Dynamic Programming

---

- Useful for optimisation problems
- Requirements:
  - Optimal substructure
  - Overlapping subproblems
- Memoise previous results to avoid repeated recomputation
- Top-Down and Bottom-Up approaches

# Greedy Algorithms

---

Divide-and-Conquer split a problem into subproblems that did not overlap.

Dynamic Programming is useful when subproblems do overlap but we have to evaluate many/all options to identify an optimal solution.

Greedy Algorithms are useful when we can choose between the subproblems without having to evaluate all of them, usually based on a static analysis of the problem that feeds into the algorithm design, making it more efficient.

# Robbing the cake shop [1]

---

Consider two problems.

1. A cake shop sells pre-made cakes, packaged into cardboard boxes of certain sizes. Different types of cake have different price tags. You have a bag with dimensions  $w \times h \times d$  in which to carry away your swag. Which boxed cakes should you steal in order to maximise the value of cakes you obtain?
2. The shop also sells flour, sugar and other cake-making ingredients. Each ingredient is sold and priced by gramme and has a certain density (grammes per unit volume). You have the same  $w \times h \times d$  bag available. How much of each ingredient should you steal to maximise your take-away value?

# Robbing the cake shop [2]

---

Problem 2 is easy to solve with a greedy approach.

- For each ingredient, multiply its price-per-gramme by its density to obtain its price-per-volume.
- Sort (descending) by price-per-volume
- Start filling the bag with the top-ranked ingredient, stopping only when the bag is full or no more of that ingredient remains, whichever happens first.
- If there is still space in the bag, continue taking the second most valuable ingredient (by volume).
- Repeat until the bag is full or none of any ingredient remains.



This works because we never regret leaving some volume of ingredients behind because the same volume of our bag was taken up by a more valuable alternative, due to the sort.

# Robbing the cake shop [3]

---

Problem 1 is notoriously difficult and does not admit a greedy solution.

We could use a dynamic programming technique to solve it.

## Remember this?

---

Cores	1	2	3	4	5	6	7	8	9	10
Value	1	6	7	9	9	10	11	16	18	25

We studied  $n=4$  in the previous lecture. At the first step, we could make a virtual machine with 1, 2, 3 or 4 cores, earning 1, 6, 7 or 9 points of value. A greedy approach might choose a 4-core machine to maximise the value (9 points) at this choice point (the locally optimal choice).

We know this is not optimal overall because the winning combination was 2+2. This problem also does not admit a greedy solution.

# What do Greedy Algorithms need?

---

Work out whether the problem admits a greedy solution.

1. Look at your task as an optimisation problem in which we can select one move as being the locally best (greedy) option  $\Rightarrow$  one subproblem remains.
2. Prove that there is always an optimal solution to the original problem (remembering there could be more than one) if we eagerly commit to the first move being the greedily chosen one.
3. Prove the optimal substructure problem, that is, if we combine the greedy choice with an optimal solution to the subproblem that remains, we get an optimal solution to the original problem.



# Famous problems with Greedy Solutions

---

**Minimum spanning tree:** we will encounter this in Algorithms 2

**Huffman coding:** a data compression/decompression system, widely used, including in fax machines

**Matroid problems:** finding a maximum-weight independent subset in a weighted matroid  $\Rightarrow$  see CLRS chapter 16 section 4.

**Calculating change** using coins of particular denominations

Various **scheduling problems**, e.g. minimising average CPU completion time.  
Let's consider one such now.

# The Activity Selection Problem

---

Given a set of activities that wish to use a shared resource, **find a maximum-size subset of *compatible* activities.**

Two activities are compatible if they do not wish to use the shared resource at the same time. Activities run in the interval  $[s_i, f_i)$ , i.e. another can begin at the instant a previous activity finishes.

Activity, $i$	1	2	3	4
Start, $s_i$	08:00	09:00	10:00	10:45
Finish, $f_i$	08:30	11:30	11:00	12:00

# Solving The Activity Selection Problem [1]

---

Sort the activities by finish time.

Step 1: look at our task as an optimisation problem. We must maximise the cardinality of the set of activities we choose.

We start with the whole day available; denote this  $S(00:00, 23:59)$  and let its value be a maximum size set of activities that can be scheduled within those times.

Activity, $i$	1	3	2	4
Start, $s_i$	08:00	10:00	09:00	10:45
Finish, $f_i$	08:30	11:00	11:30	12:00

# Solving The Activity Selection Problem [2]

---

If we are solving  $S(i, k)$  with some set  $A$  of activities available to choose from, we pick an activity,  $a_j \in A$  and note the number of activities this would yield is

$$1 + |S(i, s_{a_j})| + |S(f_{a_j}, k)| \quad \text{... means set cardinality}$$

where the two recursive calls have a filtered subset of  $A$  containing only those activities that are compatible with  $a_j$  (i.e. do not overlap in time with  $a_j$ ).

$$S(i, k) = \text{ARGMAX}_j \{a_j\} \cup S(i, s_{a_j}) \cup S(f_{a_j}, k) \quad \Leftarrow \text{use dynamic programming!}$$

Activity, $i$	1	3	2	4
Start, $s_i$	08:00	10:00	09:00	10:45
Finish, $f_i$	08:30	11:00	11:30	12:00

💡 If this doesn't make sense, you're probably trying to maximise the in-use hours of the room, not the number of activities scheduled!

## Solving The Activity Selection Problem Greedily

It turns out that we do not need to try each activity in  $A$  in turn and take the max.

If we pick the activity with the earliest finish time, we leave the greatest amount of time for other activities to use the resource.

We never regret choosing this activity because any other would have finished later and could only have reduced the total number of activities we can schedule.

⇒ Solutions in this example are  $\{1, 3\}$ ,  $\{1, 2\}$ ,  $\{1, 4\}$ , all optimal (size 2).

Activity, $i$	1	3	2	4
Start, $s_i$	08:00	10:00	09:00	10:45
Finish, $f_i$	08:30	11:00	11:30	12:00

# Summary of Greedy Algorithms

---

- Not all problems can be solved greedily
- Greedy algorithms offer efficient solutions where they are possible
- Greedy algorithms solve optimisation problems where...
  - The locally optimal choice definitely does still allow an optimal solution to be reached
  - The combination of the locally optimal choice and an optimal solution to the subproblem that results from making the locally optimal first move, is optimal overall.
- Often, you need to think about what the 'greedy' choice is
  - E.g. choosing the earliest finishing time to allow the greatest cardinality set of activities overall

# Summary of Algorithms 1 so far [updated]

---

Algorithm designs:

- Incremental
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms

Coming next: Data Structures

# Algorithms 1

---

## Section 3: Data Structures



# Pointers

---

As our algorithms use integers, strings, arrays, and other objects, so our computer memory fills up. Each item begins at some point in the memory, known as the object's **base address**.

Some programming languages allow us to access base addresses and to store them in variables of **pointer** type.

Pointers allow our programs to have some dynamic behaviour: we can write our code to process a value, which value is not known until runtime.

**NIL** is a reserved pointer value that does not refer to any object.

# Stacks

---

Stacks are last-in first-out (LIFO) data structures.

The insert operation is called `PUSH(item)`.

The delete operation is called `POP()` (you cannot choose which item to delete).

To build a stack with a fixed maximum capacity, we can use an array. Stacks with unbounded capacity can be built using a linked list, or arrays provided our code is written to copy items to a larger array if necessary.

Both operations have worst case running time in  $O(1)$  for linked lists (and *amortised*  $O(1)$  for arrays, as we will define and prove in Algorithms 2!).

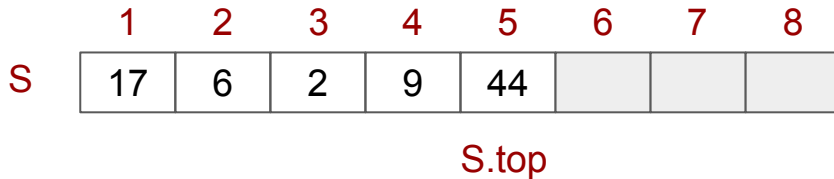
# Stacks in arrays

---

Maintain a variable, `top`, recording which array index is the top of the stack, either

1. The index of the item that is the current top of stack (or 0 to signify empty);  
OR
2. The index of the first blank space above the top (1 signifies empty).

With option 1, PUSH must increment `top` before storing into the array, and POP must decrement `top` after reading out of the array. Option 2 is the opposite.



This stack uses option 1.

# (Trivial) Stack Algorithms

---

STACK-EMPTY (S)

```
1  return S.top == 0
```

STACK-PUSH (S, i)

```
1  if S.top == S.length
```

```
2      error("full")
```

```
3  else
```

```
4      S.top = S.top + 1
```

```
5      S[S.top] = i
```

STACK-POP (S)

```
1  if STACK-EMPTY (S)
```

```
2      error("empty")
```

```
3  else
```

```
4      S.top = S.top - 1
```

```
5      return S[S.top+1]
```

# Stack Coding Challenge

---

Design a stack to store integers that supports three operations:

- `PUSH(S, i)`: as before, this must push `i` onto the top of the stack
- `POP(S, i)`: as before, this must return the top of the stack and remove it
- `AVERAGE(S)`: this must return the average of the values in the stack

All three must run in  $O(1)$  time (worst case). *Hint:  $O(1)$  extra space is all you need!*



Remember that the operations must ‘work’ when invoked in any order!

Can you add `MIN(S)`, also running in  $O(1)$  time, which returns the minimum of the elements currently in the stack?

# Queues

---

Queues are first-in first-out (FIFO) data structures.

The insert operation is called `ENQUEUE(item)`.

The delete operation is called `DEQUEUE()` (you cannot choose which item to delete).

To build a queue with a fixed maximum capacity, we can use an array. Queues with unbounded capacity can be built using a linked list, or arrays provided our code is written to copy items to a larger array if necessary.

Both operations have worst case running time in  $O(1)$ .

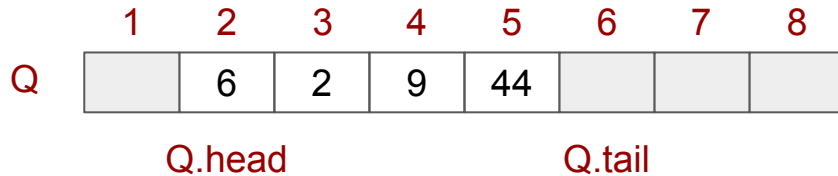
# Queues in arrays [1]

---

Maintain two variables, head and tail: enqueue at the tail, dequeue from the head.

1. Tail holds the index of the last item;  
OR
2. Tail holds the first blank space after the last item.

With option 1, ENQUEUE must increment tail before storing into the array. Option 2 increments tail before storing into the array.



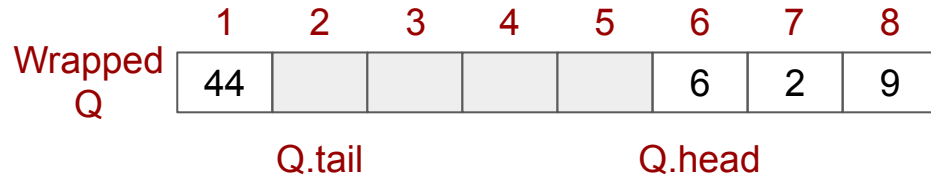
This queue uses option 2.



This is also known as a circular buffer. You will see these in Operating Systems (Kernel Pipes).

# Queues in arrays [2]

When tail and head reach the end, wrap back around to the start.

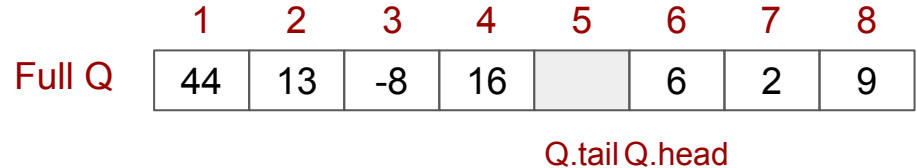


💡 In programming languages with 0-based arrays, we can say that  $\text{head} == (\text{tail} + 1) \% \text{Q.length}$  signifies a full queue.

Initially  $\text{head} = \text{tail} = 1$ . The queue is empty when  $\text{head} == \text{tail}$ .

The queue is “full” when

- $\text{head} == \text{tail} + 1$ ; OR
- $\text{head} == 1 \ \&\& \ \text{tail} == \text{Q.length}$



💡 With only  $\text{Q.length}$  different values that head and tail can take, and given that we need to represent empty (0 items), there are only enough values to distinguish 0, 1, 2, ...  $\text{Q.length}-1$  items in the queue.



# (Trivial) Queue Algorithms

---

QUEUE-EMPTY (Q)

```
1  return Q.head == Q.tail
```

QUEUE-ENQUEUE (Q, i)

```
1  if QUEUE-FULL (Q)
```

```
2      error ("full")
```

```
3  else
```

```
4      Q[Q.tail] = i
```

```
5      Q.tail = inc(Q.tail)
```

QUEUE-DEQUEUE (Q)

```
1  if QUEUE-EMPTY (Q)
```

```
2      error ("empty")
```

```
3  else
```

```
4      i = Q[Q.head]
```

```
5      Q.head = inc(Q.head)
```

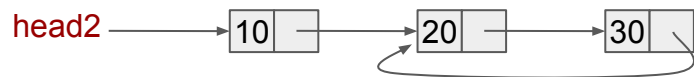
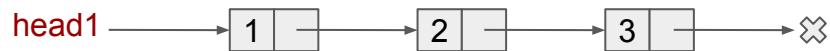
```
6      return i
```

```
inc(x) = (x == Q.length) ? 1 : x+1
```

# Linked Lists (Single Linking)

A singly linked list uses an external pointer-typed variable (conventionally called 'head') that refers to the first list cell, or is a NIL value if the list is empty. It is sometimes helpful to keep an additional pointer to the last cell in the list.

List cells are records (tuples) consisting of the data we wish to store and a pointer to the next cell (conventionally called 'next').



```
let tmp = [76;37;94];;
let head3 = [5;6;7] @ tmp;;
let head4 = [-3;8;12] @ tmp;;
```

# Traversing an acyclic List

---

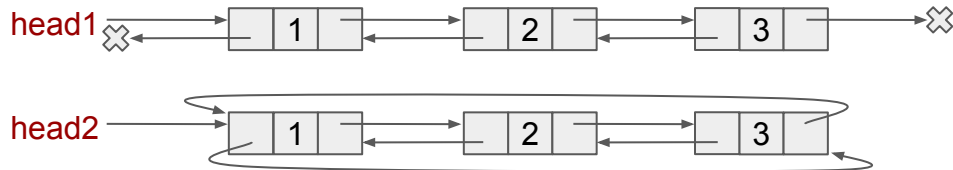
LIST-SEARCH (L, k)

```
1  c = L.head
2  while c != NIL && c.key != k
3      c = c.next
4  return c;
```

# Linked Lists (Double Linking)

A doubly linked list uses an external pointer-typed variable (conventionally called 'head') that refers to the first list cell, or is a NIL value if the list is empty. It is sometimes helpful to keep an additional pointer to the last cell in the list.

List cells are records (tuples) consisting of the data we wish to store and 2 pointers to the next and previous cells (conventionally called 'next' and 'prev').



Doubly linked cyclic lists cannot be lollypop-shape.



We cannot have converging doubly linked lists.

## DLL-INSERT-HEAD(L, i)

---

```
1  x = new DLLCell
2  x.key = i
3  x.next = L.head
4  x.prev = NIL
5  if L.head != NIL
6      L.head.prev = x
```



Henceforth, we shall abbreviate lines 1–4:  
`x = new DLLCell(prev=NIL, key=i,  
next=L.head)`

## DLL-INSERT-AFTER(L, i, k)

---

```
1  t = L.head
2  while t != NIL
3      if t.key == k
4          x = new DLLCell(prev=t, key=i, next=t.next)
5          if (t.next != NIL)
6              t.next.prev = x
7              t.next = x
8          return
9      t = t.next
```

# DLL-DELETE-HEAD(L)

---

```
1  if L.head != NIL
2      L.head = L.head.next
3      if (L.head != NIL)
4          L.head.prev = NIL
```

## DLL-DELETE-KEY(L, k)

---

```
1  t = L.head
2  while t != NIL && t.key != k
3      t = t.next
4  if t != NIL
5      if (t.prev == NIL) L.head = t.next
6      else t.prev.next = t.next
6      if (t.next != NIL)
7          t.next.prev = t.prev
```



# Implementing the Heap: Doug Lea's malloc algorithm

---

A program's memory (the “virtual address space” – when you encounter that in Operating Systems) contains the machine code, the stack, and a very large area known as the **heap** (and often a few other regions).

We wish to allocate and deallocate objects within the heap, in any order. Notice that stacks and queues solved the same problem but, in both cases, the job was made easier by the known order of allocation and deallocation.

# Implementing the Heap: challenges of Malloc

---

We might allocate a large number of small objects on the heap, or a small number of large objects. Where should we keep the metadata about each object?

Arrays are not appropriate because we would either run out of slots or waste lots of memory on vastly more slots than we require (filesystems typically do this).

We need to be able to add space for more metadata as we add more objects.

Doug Lea taught us to how solve the problem of tracking free and busy regions in the general case, using a doubly linked list.

# The Heap

---

The big idea is to represent free and busy **chunks**, in the order they are found in memory, in a linked list.

To allocate, we search the list for a free chunk that is at least big enough and we split it into the amount we want and the remaining free space. The first is marked as busy and the second is free.

To deallocate, we mark a busy chunk as free then merge it with either or both neighbours if they are also free. This serves to coagulate free space so we can allocate single, large objects in the space that was once used in separate pieces.

# The free/busy list

The doubly linked list cells are interleaved with the free/busy chunks in the heap.

The data stored in each list cell is a single free/busy bit. A **sentinel** node sits at the end.

Initially, the heap is represented as a single 'free' chunk.

heap (base address)



# First MALLOC call

```
p = malloc(1000)
```

This should set `p` to the base address of 1000 bytes of free space on the heap, and update the list to mark these bytes as busy. If there is not enough space anywhere in the heap, return NIL.



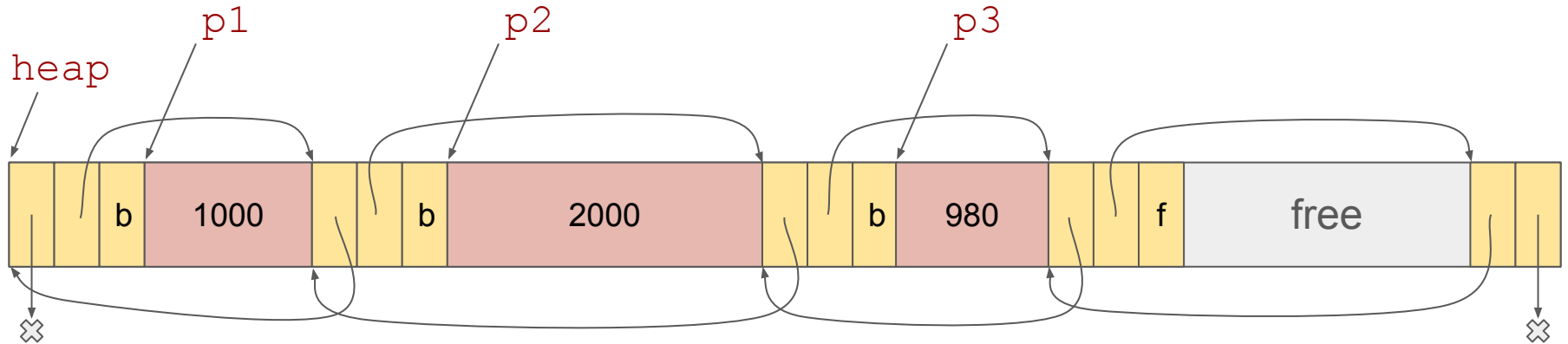
# Several MALLOC calls later...

---

```
p1 = malloc(1000)
```

```
p2 = malloc(2000)
```

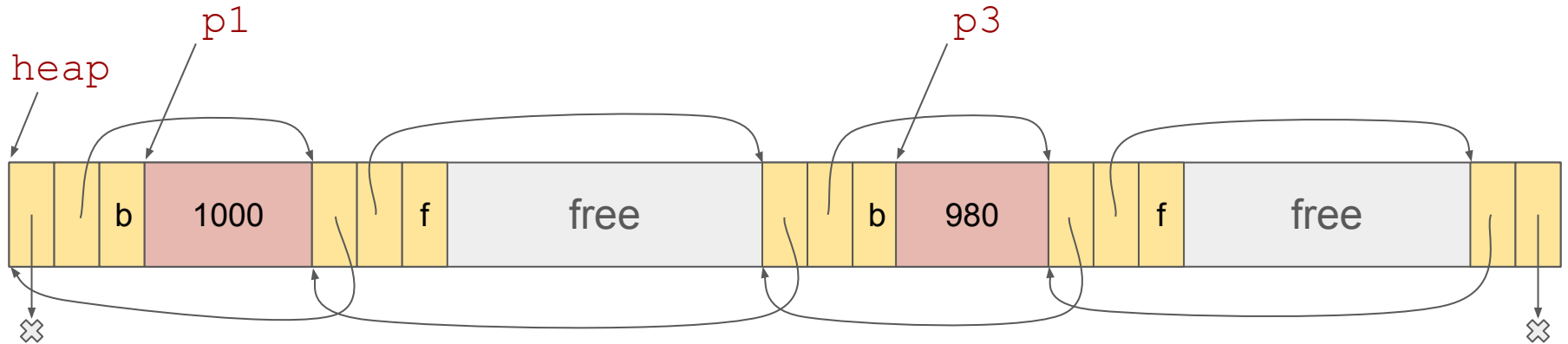
```
p3 = malloc(980)
```



# Freeing up memory: FREE(p) [1]

`free(p2)`

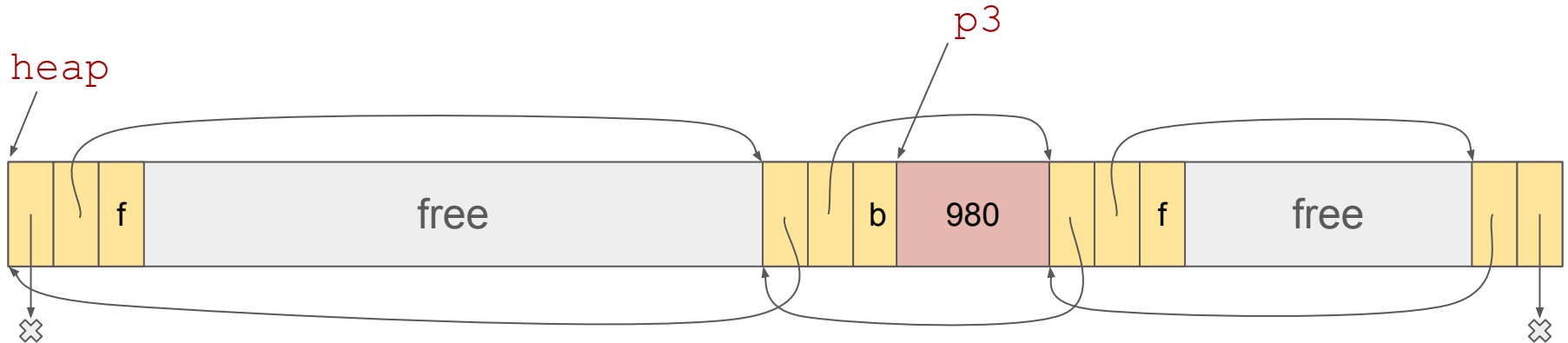
We provide the pointer `p2`. `FREE` subtracts the size of a `DLLCell` to get the base address of the `DLLCell` immediately before it. It marks this chunk as free.



# Freeing up memory: FREE(p) [2]

`free(p1)` FREE also checks whether the previous and next chunks (if they exist) are free. Goal: **never have 2 consecutive free chunks**.

- prev is free, next is busy → delete this DLLCell from the list
- prev is busy, next is free → delete next DLLCell from the list
- prev and next are both free → delete this DLLCell and next DLLCell





# Space Optimisation

---

Many implementations round up requests for memory to the next multiple of 4 bytes. This is more efficient for many CPUs and memory chips to handle than when there is a remainder modulo-4.

Now we know that each pointer will point to a multiple of 4, we know that the least significant 2 bits of each pointer must be zeros.

We store the free/busy bit in one of them! This avoids the need for an extra variable in the linked list nodes and reduces the overhead of tracking memory in this way. Before we use the value as a pointer, we bitwise AND with  $\sim 3$  to zero out the bottom bits.



3 is 00..011 in binary.  $\sim 3$  is the bitwise inverse: 11..100. ANDing with this zeros the 2 least sig bits.

# [Non Examinable] Mark and Sweep Garbage Collection

---

Doug Lea's malloc algorithm underpins the C, C++, C#, OCaml, Python, Java, ... memory allocators on Linux, Windows and MacOS. It is (arguably) the most executed algorithm since programmable computers were invented!

Remembering to call FREE (and not use use memory after calling FREE!) is a notorious source of bugs.

Garbage collection starts with our global variables, stack-allocated variables (and thread local storage). If any points to a heap location, we mark that location as in-use, and follow any pointers that might exist in the data stored there, recursively marking every reachable location. Now call FREE on unmarked locations.

# Summary of fundamental data structures

---

We looked into the details of ...

- Pointers
- Stacks, stacks with additional operations (average, minimum/maximum)
- Queues
- Singly linked lists, cyclic singly linked lists
- Doubly linked lists, cyclic doubly linked lists
- Use in Doug Lea's malloc algorithm

# Rooted Trees

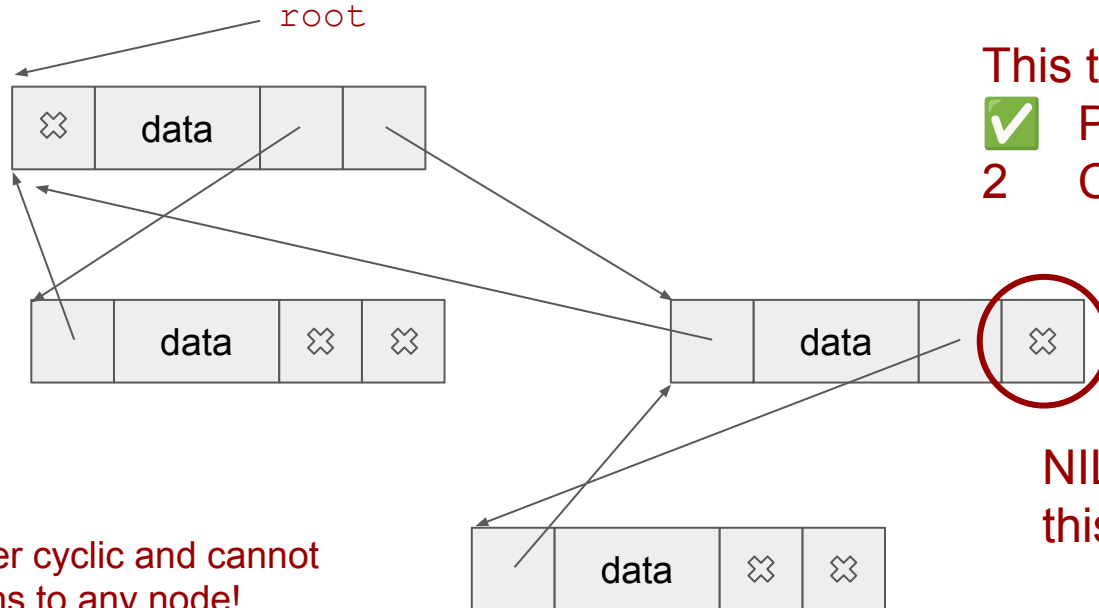
---

A **rooted tree** is a data structure with a single entry point: the root. Some types of tree insist that all operations begin at the root; others allow programs to keep pointers directly to nodes within the tree structure, allowing their algorithms to start from those places (in addition to the root).

The simplest type of rooted tree is the binary tree...

# Binary Trees

In each node, a binary tree holds one data item, pointers to two children, and optionally a pointer to the parent node (if your algorithm needs it).



This tree has...

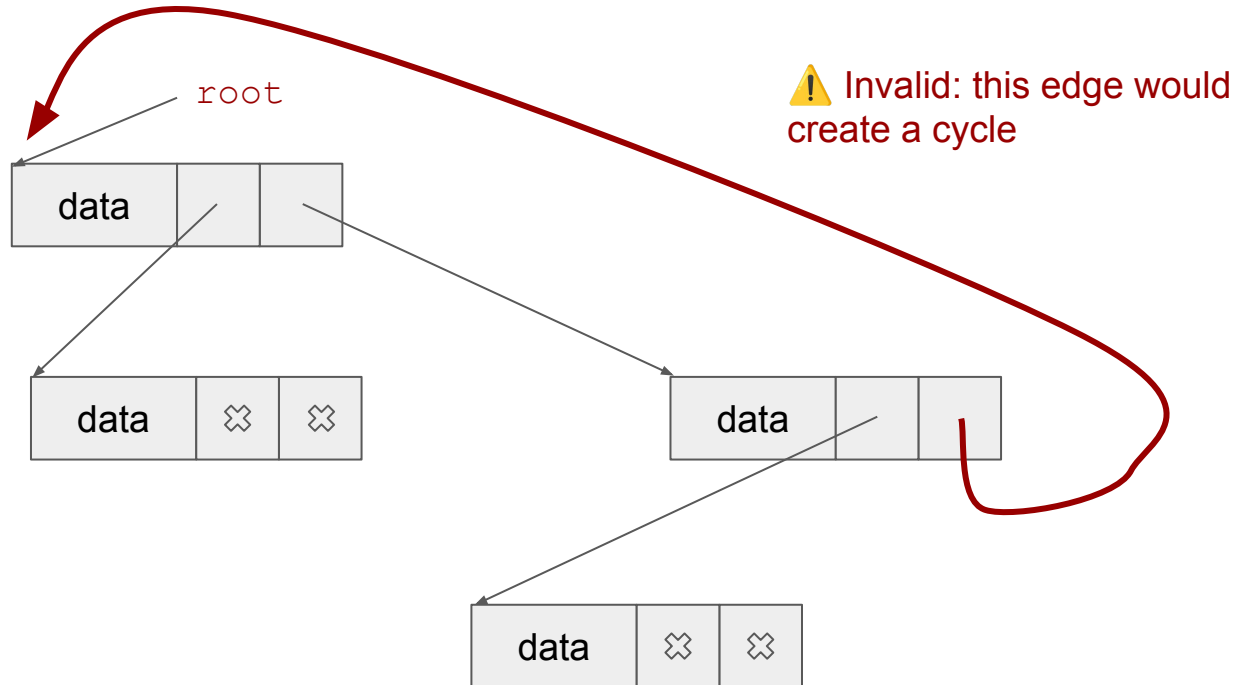
✓ Parent pointers  
2 Children

NIL if no child in  
this position (**leaf**)

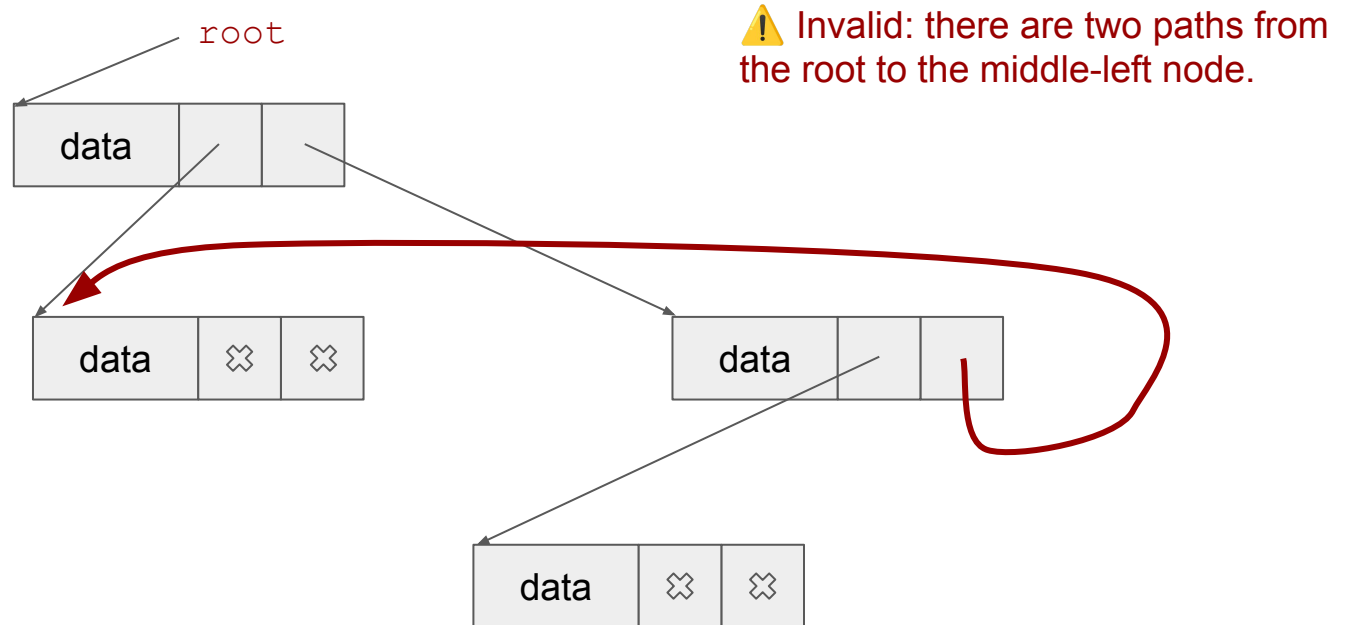
⚠ Trees are never cyclic and cannot  
have multiple paths to any node!

# A Non-Tree [1]

---

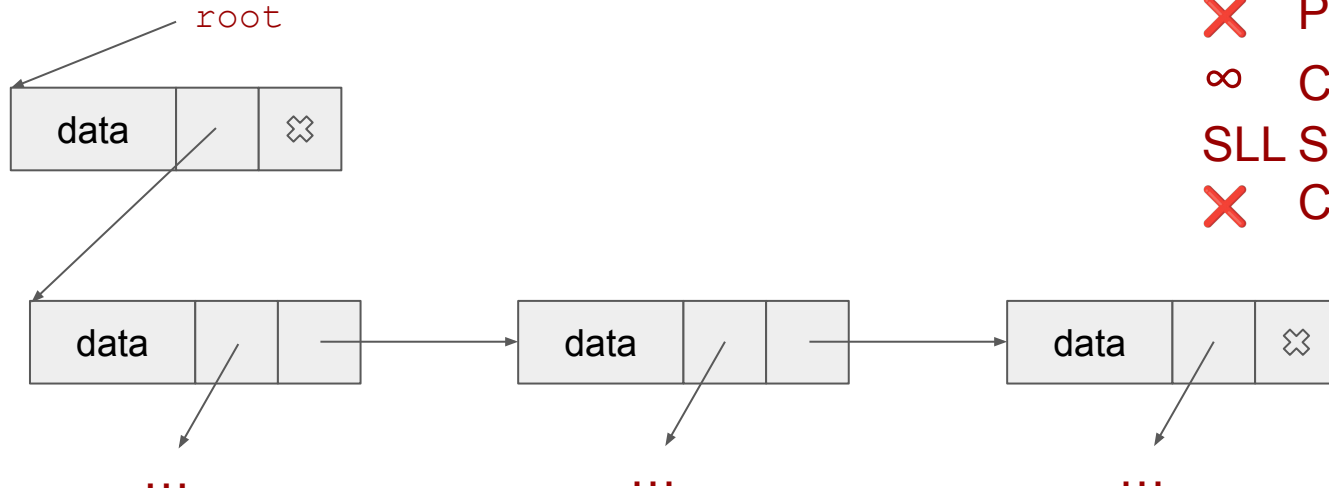


## A Non-Tree [2]



# Unbounded branching

Sometimes we need to add and remove children from our tree nodes while an algorithm runs. We cannot have a fixed number of child pointers and are forced to use a list of children: single or double, cyclic or not, as required. We can either use parent pointers, or not, as required.



This tree has...

✗ Parent pointers

$\infty$  Children

SLL Sibling list

✗ Cyclic siblings



# Binary Search Trees (BSTs)

---

BSTs are a specific use of binary trees (i.e. each node has at most two children).

- The data is a (key, payload) tuple.
- Subject to the **binary search tree property**:
  - the key in node  $i$  is (strictly) greater than all the keys in its left subtree; and
  - the key in node  $i$  is (strictly) less than all the keys in its right subtree.

Any kind of “search tree” does not allow duplicate keys. This is necessary to ensure good asymptotic running times.

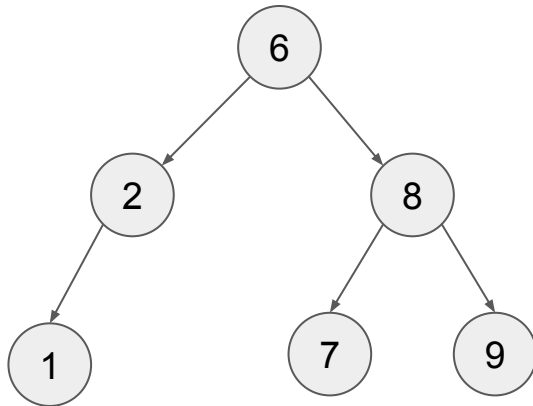
BSTs support INSERT, DELETE, SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM – all in  $O(\lg n)$  time.

# Example BST

---

We usually draw BSTs like this.

Each node still needs two child pointers but we reduce clutter by understanding that they will be present without drawing them.



## BST-SEARCH(p, k): search k, start at node pointed to by p

---

```
1  if p == NIL
2      return NIL
3  if p.key == k
4      return p
5  if k < p.key
6      return BST-SEARCH(p.left, k)
7  else
8      return BST-SEARCH(p.right, k)
```



You might return p.payload if you just want the payload rather than a pointer to the node containing the search key.

## BST-SEARCH(p, k): iteratively

---

```
1  while p != NIL && p.key != k
2      if k < p.key
3          p = p.left
4      else
5          p = p.right
6  return p
```

## BST-MINIMUM(p)

```
1  if p == NIL
2      return NIL
3  while p.left != NIL
4      p = p.left
5  return p
```

## BST-MAXIMUM(p)

```
1  if p == NIL
2      return NIL
3  while p.right != NIL
4      p = p.right
5  return p
```

## BST-PREDECESSOR(p)

```
1  if p.left != NIL
2      return BST-MAXIMUM(p.left)
3  y = p.parent
4  while y != NIL &&
           p == y.left
5      p = y
6      y = y.parent
7  return y
```

## BST-SUCCESSOR(p)

```
1  if p.right != NIL
2      return BST-MINIMUM(p.right)
3  y = p.parent
4  while y != NIL &&
           p == y.right
5      p = y
6      y = y.parent
7  return y
```

# INSERT(bst, k, v)

---

```
1  x = new BSTCell(k,v, NIL,NIL)
2  if (bst.root==NIL) bst.root = x
3  else
4      p = bst.root
5      while true
6          if k < p.key
7              if p.left == NIL
8                  p.left = x
9                  return
10             else
11                 p = p.left
12             else if k == p.key
13                 p.payload = v
14                 return
15             else
16                 if p.right == NIL
17                     p.right = x
18                     return
19                 else
20                     p = p.right
```

# DELETE(bst, k)

---

Steps:

1. Find the node,  $d$ , containing  $k$ . If NIL then return as  $k$  is not present.
2. If  $d$  has no children, remove it from its parent.
3. If  $d$  has one child, make  $d$ 's parent point to  $d$ 's child instead of  $d$ .
4. If  $d$  has two children then replace  $d$ 's (key, payload) with that of  $d$ 's predecessor, then delete the predecessor node (which can have at most one child).

Implement this algorithm and prove that what remains is definitely a valid BST.



# Summary of Rooted and Binary Search Trees

---

- Varieties of rooted trees: with/out parent pointers, fixed/varying child count.
- Search trees don't allow duplicate keys!
- BSTs support a rich set of operations with expected runtime  $O(\lg n)$
- Simple logic in the operations – simple to code and small constant factors
- But  $O(n)$  worst case performance might leave algorithms using BSTs vulnerable to very bad worst case performance (exponentially worse than the average running time, of course:  $n$  is exponential in  $\lg n$ ).
- Memory overhead of 2 pointers per (key,value) pair might be significant (3 points if you need parent pointers).

# Balanced Trees: B-Trees


---

BSTs achieved average-case  $O(\lg n)$  performance but the worst case is  $O(n)$  (when every tree node has exactly one child).

To achieve  $O(\lg n)$  in the worst case, we need to ensure that our trees remain balanced, regardless of the order in which keys are inserted or deleted.

This is especially important for data structures held on disk. The costs change!

Follow Pointer  $\rightarrow$  disk operation  $\rightarrow$  2,000,000 CPU cycles  
CPU operations still use 1 cycle each!

 E.g. the file system  
on your hard disk!

We need to minimise the number of pointers we need to follow!

# B-Trees<T>

---

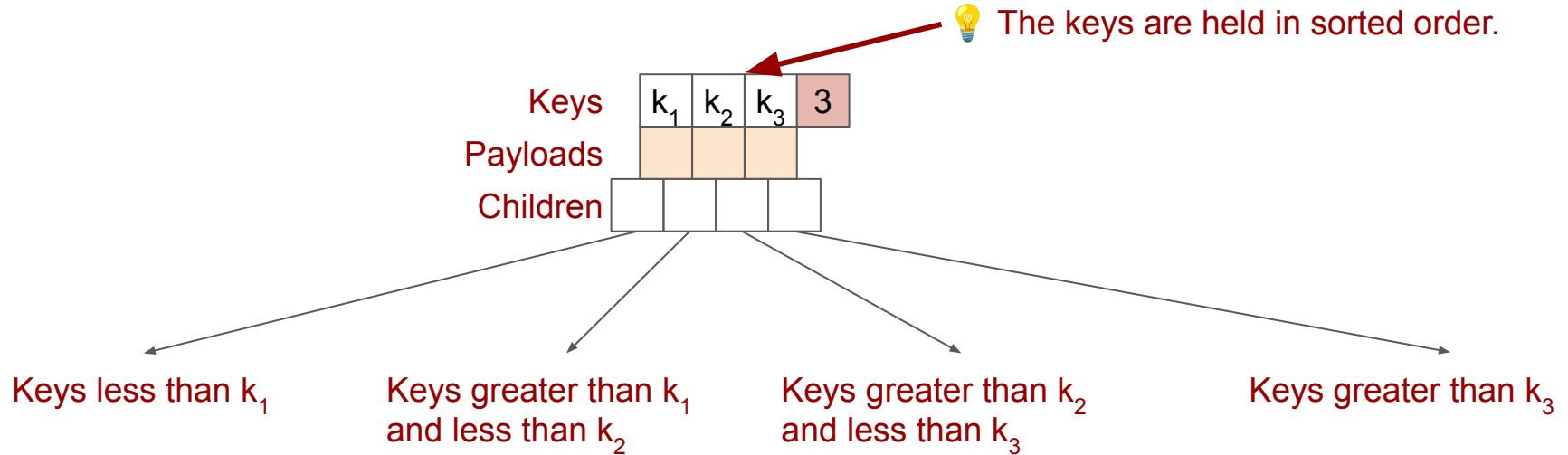
B-Trees are made of leaf nodes and internal nodes. Leaf nodes hold no keys or payloads. Internal nodes hold varying numbers of keys and payloads.

A B-Tree of **minimum degree  $T$** , has five **defining characteristics**:

- Internal nodes must hold at least  $T-1$  keys and payloads (except the root)
- Internal nodes can hold at most  $2T-1$  keys and payloads (including the root)
- A node with  $t$  keys must have  $t+1$  children
- Keyless leaves all exist at the same depth below the root
- The keys in any internal node divide the ranges of keys in their children (generalising the binary search tree property).

# B-Tree keys and children

The children contain keys between the separator keys in their parent.

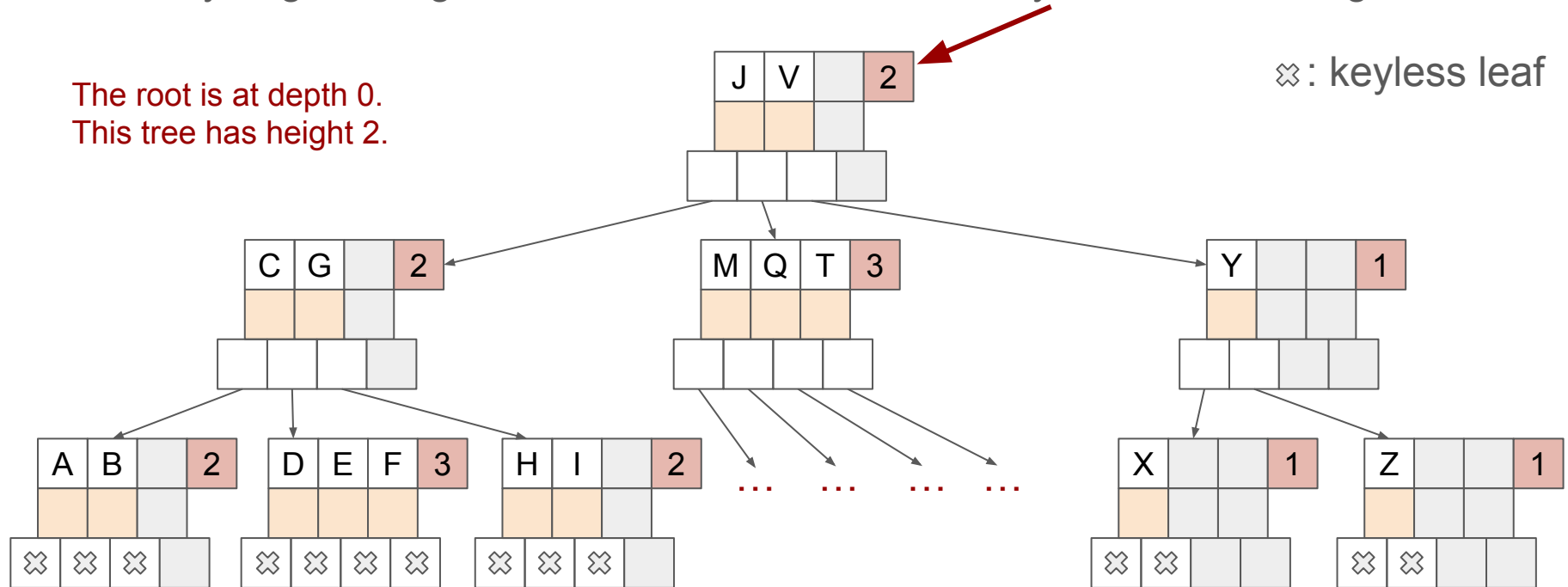


# Example B-Tree<2>

$T=2$  so internal nodes contain 1–3 keys and 2–4 children. Internal nodes contain an array large enough for the max, and its current key-count in an integer.

The root is at depth 0.  
This tree has height 2.

⊗: keyless leaf



# Height of a B-Tree<T> with N keys

---

Because tree algorithms depend on the height of a tree, we want an **upper bound** on the height of a B-Tree<T> with N keys.

The root must have at least 1 key. In subsequent levels, there must be at least  $T-1$  keys ( $T$  children).

At least 1 key at the root level. At least  $2(T-1)$  keys at the first level. At least  $2T(T-1)$  keys at the second level.  $2T^2(T-1)$  at the third level, etc.

Summing this geometric progression and rearranging, we find that the number of levels required to hold N keys is at most  $\log_T((N+1)/2) \Rightarrow$  we expect  $O(\lg N)$  time!

# Pointers to nodes

---

It is NOT permitted to maintain pointers to any internal nodes (or leaf nodes).

The only pointer into a B-Tree is the root pointer. All algorithms start at the root.

Why?

Algorithms are going to move keys around to maintain the balance. If you kept a pointer to an internal node, perhaps believing it to contain a particular key, it might not contain that key by the time you come to use the pointer! Inserting or deleting other, unrelated keys, even far away in the key space, can restructure the tree!

## BT-SEARCH(n, k): top-level call uses n=root

---

```
1  if n == BT-LEAF
2      return NIL                                // k is not present
2  i = 1
3  while i ≤ n.keycount && k > n.key[i]
4      i = i + 1
5  if i ≤ n.keycount && k == n.key[i]
6      return n.payload[i]                        // or return pointer to node n
7  else return BT-SEARCH(n.child[i], k)
```

💡 We might use binary search instead:  $O(\lg T)$  rather than  $O(T)$  search time, but both are  $O(1)$ . However, this makes search  $O(\lg n)$  even if you make  $T$  a function of  $n$ .



# BT-INSERT( $n, k, p$ )

---

- Start at the root. If  $\text{root} == \text{BT-Leaf}$ , set  $\text{root} = \text{new BT-Node}(k, p)$  and return.
- Walk down the tree, searching for key  $k$ .
- If  $k$  is found, replace the payload and return.
- If you reach the **bottom level** of internal nodes (children are keyless leaves) then insert into this node, increasing this node's keycount by 1.
- BUT, if that bottom node is already full ( $2T-1$  keys already) then split it about its median key into two nodes of  $T-1$  keys + the median. Insert the median into the parent node, conveniently allowing the parent to take second half as another child.
- If the parent is full, recurse back up. If the root splits, update the root pointer!



B-Trees only increase in height when the root splits, preserving the height-balance property!

# Improved BT-INSERT( $n, k, p$ )

---

- Start at the root. If  $\text{root} == \text{BT-Leaf}$ , set  $\text{root} = \text{new BT-Node}(k, p)$  and return.
- Walk down the tree, searching for key  $k$ . If a full node is encountered, split it.
- If  $k$  is found, replace the payload and return.
- If you reach the **bottom level** of internal nodes (children are keyless leaves) then insert into this node, increasing this node's keycount by 1.
- BUT, if that bottom node is already full ( $2T-1$  keys already) then split it about its median key into two nodes of  $T-1$  keys + the median. Insert the median into the parent node, conveniently allowing the parent to take second half as another child.
- The parent cannot be full (would have been split), avoiding recursion back up.



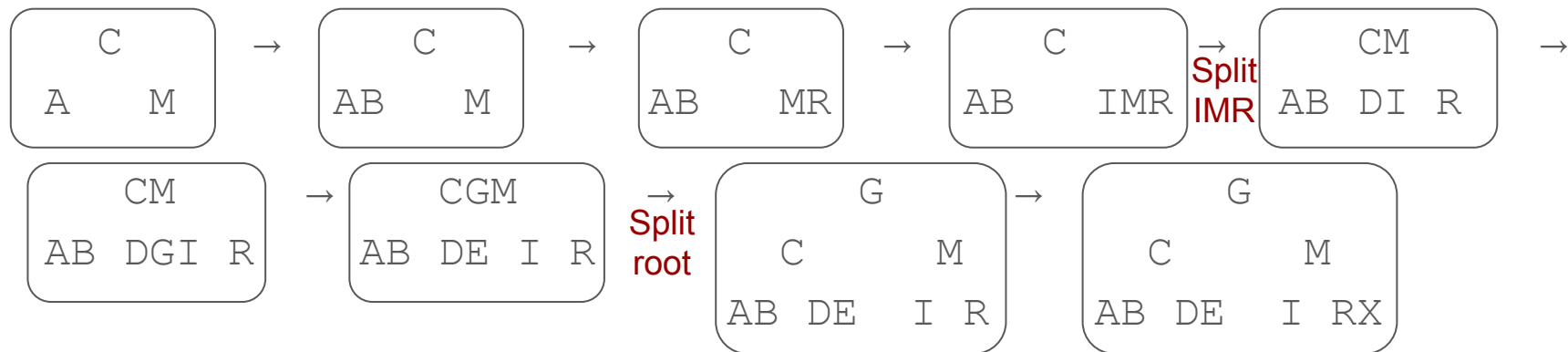
Remembering that the nodes are on disk, this avoids seeking to nodes we have been to before.

# Examples of BT-INSERT

- Insert CAMBRIDGEX into an initially empty B-Tree of minimum degree 2.

<leaf> → C → AC → ACM → ...

When we insert B, we find the root is full and split it. Splits happen at D, E, and X.



# BT-DELETE( $n, k$ ) [1]

---

- Start at the root.
- Walk down the tree, searching for key  $k$ .
- If  $k$  is not found, then return.
- If  $k$  is the only key in the root, replace root with a BT-Leaf node and return.
- If  $k$  is found but **not** at the bottom level, swap  $k$  with its predecessor (which must be at the bottom level).
  - At this point, the key to be deleted is at the bottom level. –
- If the node is not minimum size, remove  $k$  from the node and decrease  $n.keycount$ .



Successor is equally acceptable to predecessor. Both must exist and be bottom-level keys.

## BT-DELETE( $n, k$ ) [2]

---

- If the node containing  $k$  is minimum size but its left or right sibling (if exists) is not minimum size, then redistribute one/more keys from the sibling into this node (incl. separator key in the parent), then delete  $k$  and decrease keycount.
- If both siblings are minimum size or don't exist (we're the first/last child) then merge this node ( $T-1$  keys) with the sibling ( $T-1$ ) keys and 'steal' the separator key from the parent. This makes a full node ( $2T-1$  keys), from which we delete  $k$  and decrease keycount.
- If the parent is minimum size and cannot give us a key, recursively redistribute or merge it with its siblings first.
- If the root's last key is stolen by its merging children, update the root pointer!

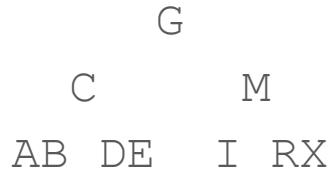


Remembering that the nodes are on disk, this avoids seeking to nodes we have been to before.

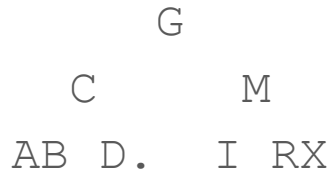
# Examples of BT-DELETE [1]

---

Delete E from this B-Tree<2>:



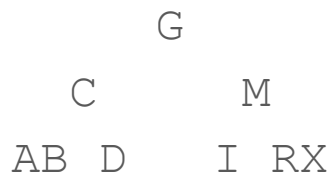
Start at the root. Find E at the bottom level. Node is not minimum size. Delete E.



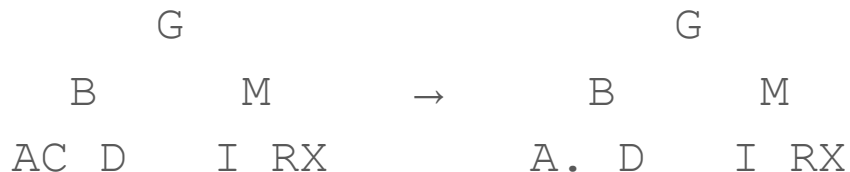
## Examples of BT-DELETE [2]

---

Delete C from this B-Tree<2>:



Start at the root. Find C not at the bottom level. Swap with predecessor, B.



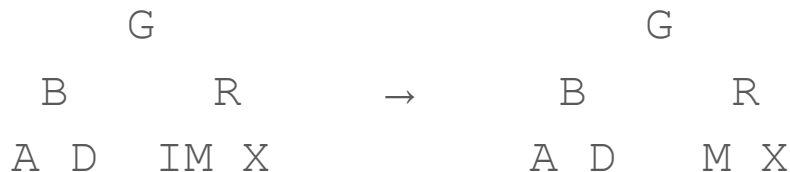
💡 There are no keys between k and its predecessor so swapping them only “messes up” the ordering for those two, and it is resolved when we delete k!

# Examples of BT-DELETE [3]

Delete I from this B-Tree<2>:



Start at the root. Find I at the bottom level. Redistribute, then delete.



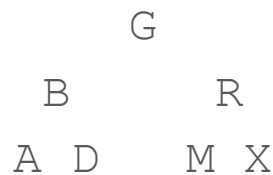
💡 We only need to move 1 key but for a B-Tree with larger T we would balance the size of the two nodes to reduce the chance of needing to split/merge again soon.



# Examples of BT-DELETE [4]

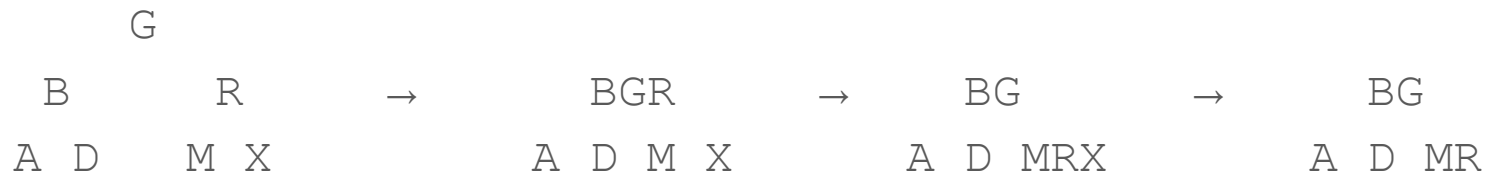
---

Delete X from this B-Tree<2>:



💡 Some authors suggest performing these operations in another order, e.g. merge MRX, temporarily leaving 'R' with too few keys. This will fail if your algorithm is interrupted (e.g. power cut), leaving a corrupted tree on the disk. Proving correctness is also harder because existing proofs do not apply when the “tree” is not valid!

Start at the root. Find X at the bottom level. Merge parent, merge then delete.



# Recommended exercises

---

1. Past exam question: y2007 p11 q9 ([link](#))
2. Prove that, if BT-INSERT is executed on a valid B-Tree with minimum degree  $T$  then what results also satisfies the 5 defining characteristics of a B-Tree.
3. Prove that, if BT-DELETE is executed on a valid B-Tree with minimum degree  $T$  then what results also satisfies the 5 defining characteristics of a B-Tree.
4. We improved BT-INSERT by preemptively splitting on the way down but did not recommend preemptively merging on the way down for BT-DELETE. Did we miss a trick?
5. Show that there is no way to guarantee  $O(\lg n)$  performance if duplicate keys are permitted.

# Red-Black Trees (RBTs)

---

Red-Black trees are binary search trees where each node has an additional colour attribute.

The colour is used by modified INSERT and DELETE algorithms to maintain **approximate balance**.

Approximate balance means that the longest path from the root to a leaf is no more than twice as long as the shortest such path.

# Red-Black Tree Properties

---

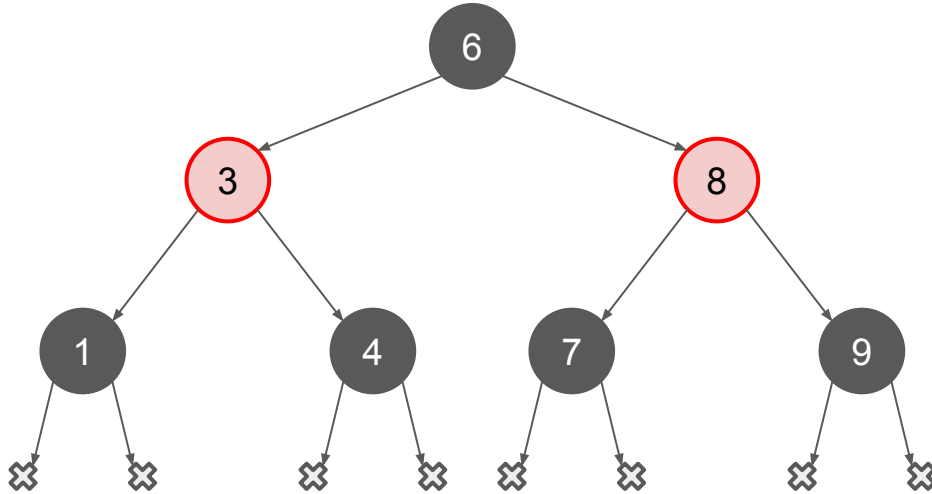
A red-black tree is a binary search tree that satisfies the following additional constraints:

1. Every node is either red or black.
2. The root is black.
3. The leaves are black and contain no keys or payloads.
4. Both children of a red node are black.
5. For each node, all simple paths to descendant leaves contain the same number of black nodes.

#5 is often phrased as “the tree is **black-height balanced**”.

# Black-Height of a node

---



BH = 2

BH = 2

BH = 1

BH = 0

- DO NOT count the node you start from.
- DO count black nodes below.
- DO count the black leaves.

# Range of tree sizes

---

Suppose a black node,  $x$ , has black height  $bh(x) = y$ . What are the minimum and maximum numbers of internal nodes (non-leaves) in the subtree rooted at  $x$ ?

**Minimum:** all the nodes are black. The tree must be a complete binary tree, by RBT Property 5. Since the black height includes the leaves but not node  $x$ , the number of internal nodes is the size of tree excluding the leaf level:  $2^{bh(x)} - 1$ .

**Maximum:** every black node has 2 red children. The subtree rooted at  $x$  has internal nodes for  $2 bh(x)$  levels, plus leaves below that. The number of internal nodes is  $2^{2 bh(x)} - 1$ .



You can prove either by induction: leaves are height 0 and formula is correct; inductive case combines two subtrees (which are definitely smaller) and 1 extra node.

# Logarithmic running time

---

**Lemma:** A red-black tree with  $n$  internal nodes (not counting leaves) has at most  $2 \lg(n+1)$  levels.

This is the basis of proofs that operations on red-black trees will have  $O(\lg n)$  *worst* case running time.

Let's look at how to prove the lemma.

# Proof of lemma

---

Consider an arbitrary black node,  $x$ . The subtree rooted at  $x$  has at least  $2^{bh(x)} - 1$  internal nodes.

Let the height of the (whole) tree be  $h$ . At least half of the nodes on a path of length  $h$  from the root to a leaf are black (RBT Property 4). The black height of the root must be at least  $h/2$ .

The number of nodes,  $n$ , in a tree with black height  $h/2$ :  $n \geq 2^{h/2} - 1$ .

Hence  $n + 1 \geq 2^{h/2}$ , and so  $\lg(n+1) \geq h/2$ , and finally  $h \leq 2\lg(n+1)$ , as required.



# Isomorphism with B-Tree<2> (2-3-4 Trees)

---

A B-Tree with minimum degree  $T=2$ , also known as a 2-3-4 Tree, is made from:

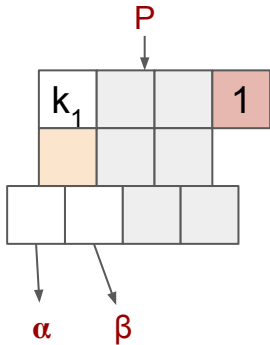
- Leaves
- Internal nodes with 1 key / 2 children
- Internal nodes with 2 keys / 3 children
- Internal nodes with 3 keys / 4 children

We can map these data structures *and the algorithms that operate on them* to red-black trees. We can convert in either direction, perform some operations, convert back, and obtain the same as if we had performed the operations on the original.

# Isomorphic twins [1]

## 2-3-4 Tree

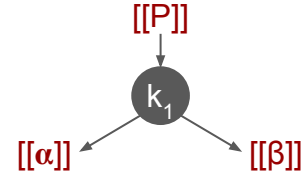
⊗ BT-Leaf



Internal node with 1 key  
and 2 children

## Red-Black Tree

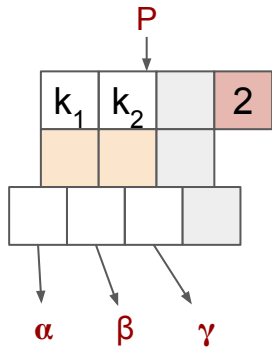
⊗ Leaf (Black)



A single black node  
Parent is the isomorphic  
mapping of  $P$ . Children are the  
mappings of  $\alpha$  and  $\beta$ .

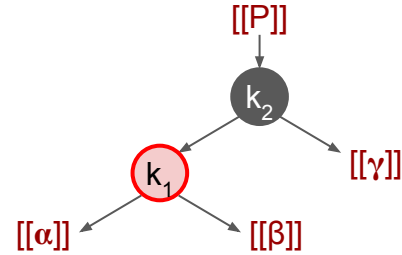
# Isomorphic twins [2]

## 2-3-4 Tree

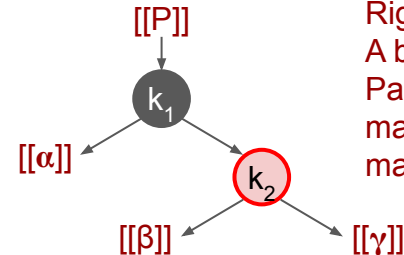


Internal node with 2 keys  
and 3 children

## Red-Black Tree



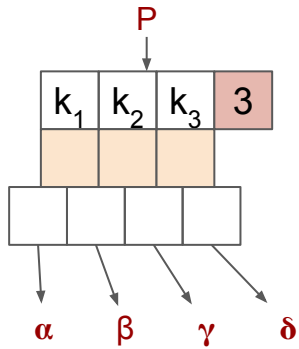
Left-handed isomer.  
A black node with one red child.  
Parent is the isomorphic  
mapping of  $P$ . Children are the  
mappings of  $\alpha$ ,  $\beta$  and  $\gamma$ .



Right-handed isomer.  
A black node with one red child.  
Parent is the isomorphic  
mapping of  $P$ . Children are the  
mappings of  $\alpha$ ,  $\beta$  and  $\gamma$ .

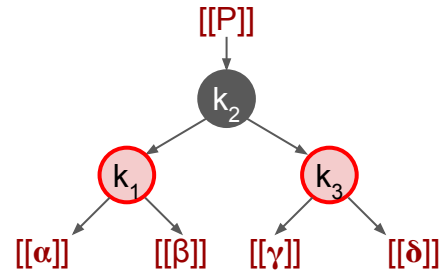
# Isomorphic twins [3]

## 2-3-4 Tree



Internal node with 3 keys  
and 4 children

## Red-Black Tree



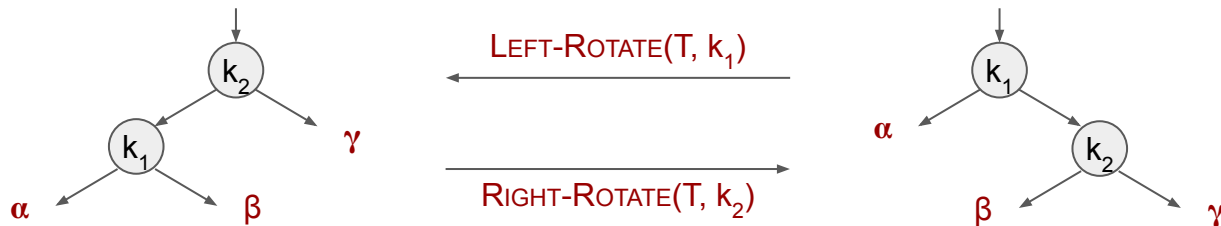
A black node with two red  
children.

# Tree rotations

---

A tree rotation is a local restructuring that preserves the global ordering property.

Tree-rotations can be applied to BSTs and red-black trees (with additional restrictions on node colours and rules to recolour the nodes correctly).



# Exercises

---

- Figure out when rotations can be applied to red-black tree clusters (e.g. does the top node have to be black? Can one be red and one black? ...)
- Figure out the rules for recolouring nodes when rotations are applied to red-black trees.
- Convert BT-INSERT and BT-DELETE to RB-INSERT and RB-DELETE.
- Prove that RB-INSERT and RB-DELETE have  $O(\lg n)$  running time.



Answers are in CLRS chapter 13.

# Priority Queues

---

A max priority queue provides INSERT(PQ, k), MAXIMUM(PQ), EXTRACT-MAX(PQ), and INCREASE-KEY(PQ, p, k) operations.

Max PQs are useful for schedulers to select the most important job to run next.

A min priority queue provides INSERT(PQ, k), MINIMUM(PQ), EXTRACT-MIN(PQ), and DECREASE-KEY(PQ, p, k) operations.

Min PQs are useful for ordering work items to run in a particular order.

# Implementation using a heap

---

Using a max-heap or min-heap, which we saw in heapsort, we can build a Max-PQ or Min-PQ.

Let's implement a Min-PQ using a min-heap.

Peeking at the minimum, without extracting it, is as simple as looking at the root of the min-heap:

PQ-MINIMUM(pq)

```
1  if (pq.heap_size == 0) error("Empty Priority Queue")
2  return pq[1]
```



## PQ-EXTRACT-MIN(pq)

---

```
1  if (pq.heap_size == 0) error("Empty Priority Queue")
2  min = pq[1]
3  pq[1] = pq[pq.heap_size]
4  pq.heap_size = pq.heap_size - 1
5  MIN-REHEAPIFY(pq, 1)
6  return min
```

## PQ-DECREASE-KEY(pq, idx, k)

---

```
1  if (pq[idx] < k) error("Key already smaller than " + k)
2  pq[idx] = k
3  while idx > 1 && pq[PARENT(idx)] > pq[idx]
4      swap(pq[idx], pq[PARENT(idx)])
5      idx = PARENT(idx)
```

## PQ-INSERT(pq, k)

---

```
1  pq.heap_size = pq.heap_size + 1
2  pq[pq.heap_size] =  $\infty$ 
3  PQ-DECREASE-KEY(pq, pq.heap_size, k)
```

# Implementation using a red-black tree

---

We can implement the priority queue ADT using a red-black tree. Let's do that!

Peeking at the minimum, without extracting it, is as simple as getting the Minimum of the red-black tree:

PQ-MINIMUM(pq)

```
1  if (pq.root == NIL) error("Empty Priority Queue")
2  x = pq.root
3  while (x.left != NIL) x = x.left
4  return x.key
```

## PQ-EXTRACT-MIN(pq)

---

```
1  min = PQ-MINIMUM(pq)      // Could be combined to avoid two
2  RB-DELETE(pq, min)        // walks down the tree.
3  return min
```

## PQ-DECREASE-KEY(pq, oldk, k)

- 1 RB-DELETE (pq, oldk)
- 2 RB-INSERT (pq, k)

## PQ-INSERT(pq, k)

- 1 RB-INSERT (pq, k)

# Cost of operations

---

Linux Kernel task scheduler

	Heap Implementation	Red-Black Tree Implementation
PQ-MINIMUM	$O(1)$	$O(\lg n)$
PQ-EXTRACT-MINIMUM	$O(\lg n)$	$O(\lg n)$
PQ-DECREASE-KEY	$O(\lg n)$	$O(\lg n)$
PQ-INSERT	$O(\lg n)$	$O(\lg n)$



In Algorithms 2, we will achieve  $O(1)$ ,  $O(\lg n)$ ,  $O(1)$ ,  $O(1)$  – albeit with a large constant factor.

# Summary of Priority Queues

---

- We have seen two ways to implement the priority queue ADT.
  - We will see another in Algorithms 2
- Implementing a priority queue was little more than using a heap or red-black tree.



# Hash Tables

---

Hash Tables implement the dictionary interface operations:

- Insert
- Search
- Delete

Under certain (reasonable) assumptions, hash tables provide  $O(1)$  performance for all three operations.

 Hash tables number slots from 0, not from 1. This lecture uses the notation  $T[..]$  with 0-based indexing to distinguish from  $A[..]$  with 1-based indexing elsewhere in the course.

# Use cases for hash tables

---

**Sparse arrays:** the keys are integers but only a very small number of keys will ever be in use at once. E.g. the keys might be 128-bit integers (UUIDs): even if we want to store 1 bit to record which UUIDs we have seen, **direct addressing** (as used by arrays) with  $2^{128}$  slots would use  $4 \times 10^{28}$  GB of RAM, which is infeasible with today's technology. **Hashed addressing** is a feasible alternative.

**Non-integer keys:** if the keys are not integers (e.g. are strings), the direct addressing scheme of an array cannot be used.

# Direct addressing

---

CPUs can LOAD data from memory and STORE data to memory but those two machine code instructions need the memory address (in bytes) to access.

The data type of array elements is fixed and known in advance. This tells us the size,  $X$ , in bytes of each entry.

Direct addressing translates programming language syntax such as “ $T[i]$ ” into a CPU LOAD or STORE instruction, like this:

**LOAD**  $X$  bytes **from** (*beginning* +  $i * size$ ) **into** register  $\langle r \rangle$

This is for zero-based hash tables; it would be  $(i-1)*size$  for 1-based arrays.

# Hashed addressing

---

To support another data type,  $t$ , as the key we use a **hash function**,  $h: t \rightarrow \text{int}$ .  
E.g. to support  $T[\text{"Cambridge"}]$  we would need a  $\text{string} \rightarrow \text{int}$  hash function.

The input to  $h$  is called the **hash key** and the output is called the **hash value**.

A naïve interpreter or compiler might convert  $T[\text{"Cambridge"}] = 42$  into:

```
idx = h_STRING("Cambridge")  
T[idx] = 42
```

...and then into a LOAD or STORE instruction using *beginning + idx \* size*.


idx is short for 'index' and is the table index that "Cambridge" maps to.

# Problems with hashed addressing [1]

---

**Problem 1: Range.** The integer output (hash value) of the hash function is used as a table index. However, we often wish to change the size of our hash tables as our data grows and do not wish to have a great many different hash functions, one for each size of table.

The solution is to require that, taken across all possible inputs, a hash function must output a *uniformly distributed* unsigned integer in some range, e.g.  $[0..2^{32}-1]$ .

We use range reduction to convert to a table index, perhaps as remainder after division by the table size. E.g. if  $h(\text{"Cambridge"}) = 34$ , and the table size is 10, we use index  $34 \text{ MOD } 10 = 34 \% 10 = 4$ .  “%” is MOD in many prog.langs.

 This will not (quite) give a uniform distribution of table indices!

## Problems with hashed addressing [2]

---

**Problem 2: Performance.** It is extremely difficult to take arbitrary input data and yield a uniformly distributed output, especially when you know nothing about the inputs that a particular program might use as hash keys.

Real-world hash functions are very complex, slow-running functions.

E.g. the input might be DNS hostnames or words in a natural language spelled with ASCII characters so some byte values never appear in the input!

E.g. if it's Australian, British or US English, the relative frequencies of “S” and “Z” will differ... and other languages have totally different letter distributions.



Hash tables are asymptotically fast but that might hide huge constant factors!

## Problems with hashed addressing [2]


---

**Problem 3: Collisions.** If we are mapping strings of any length into 32-bit integers, there are necessarily collisions, by the pigeonhole principle: we have more possible hash keys than possible hash values so at least one hash value must be used at least twice – a collision.

```
T["Cambridge"] = 1209
```

```
T["Oxford"] = 1096
```

```
Print T["Cambridge"] → runs to give 1096 ?!?!?
```

 **This shows what must NOT happen in our implementations of a hash table!**

The hash values collided and the Oxford data over-wrote the Cambridge data. Resolving this is central to using hash tables well in our algorithms.

# Problems with hashed addressing [3]

---

**Problem 4: Entropy.** If our keys are short or very similar, there is not enough entropy to map them to different places in the table.

E.g. similar keys might be mapped to a smaller number of distinct hash values:  
ROT, TOT, POT, DOT, GOT, HOT, JOT, LOT, COT, BOT, NOT.

E.g. if the keys are single letters of the alphabet, there are only 26 possible inputs to a hash function, hence at most 26 different outputs in the best case. That will not be a uniform distribution across the range  $[0..2^{32}-1]$  !!

If you're going to use low-entropy inputs, choose your hash function wisely!



See CLRS chapter 11 for ideas.



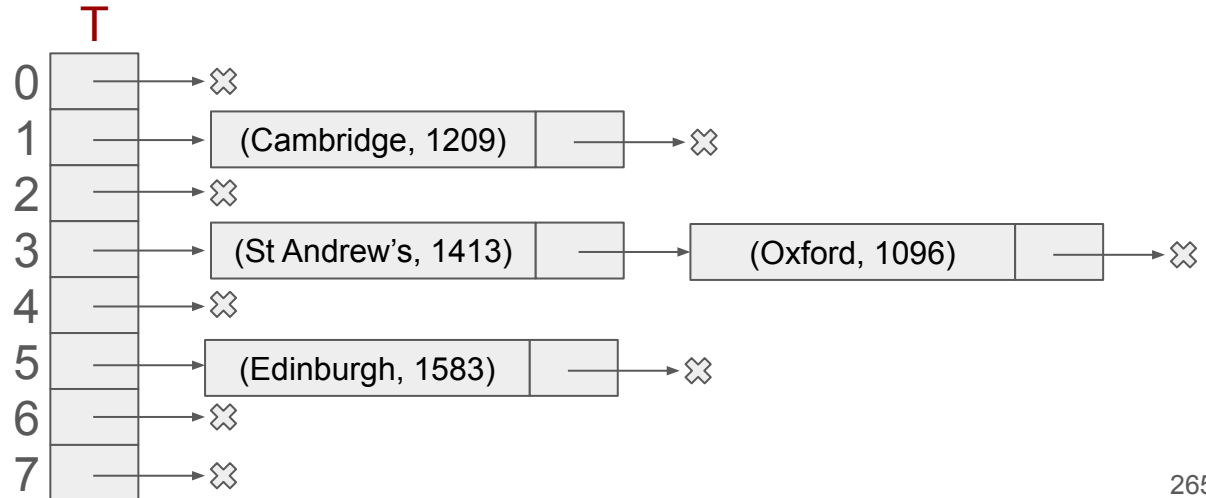
# Collision resolution by Chaining

The entries in the hash table, T, are pointers to linked lists, initially NIL pointers.

List cells are (Key, Payload, Ptr) tuples, where Ptr is the pointer to the next cell.

There are still lots of variants!

INSERT("Oxford", 1096)  
 $h(\text{"Oxford"}) \rightarrow 3$



# Chaining Variant 1

---

INSERT( $T$ , key, payload): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, replace the payload; if no match, append a new cell on the end of the list.

SEARCH( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, return the payload; if no match, return NIL.

DELETE( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, remove cell from list. NO-OP if not found.

💡 Insert means “ensure future searches this payload under this key”.

💡 Delete means “ensure future searches do not find this key”.

# Performance of Chaining Variant 1 [1]

---

INSERT( $T$ , key, payload): if the hash function is good, the keys will be spread evenly across the hash table. We expect each list to hold  $\alpha = n/T.size$ , where  $n$  is the number of live keys stored in the table.  $\alpha$  is called the **load factor**.

Provided we resize when  $n \approx T.size$ , the expected runtime for insert is  $O(1)$ , in general  $O(1+\alpha)$ . The worst case time for insert is  $\Theta(n)$ , if the hash function is poor, or the data is pathological or malicious.

If we insert keys randomly, the average time to insert a key that is already present (known as **rebinding** the key to a new payload) is one half of the time to insert a new key – because existing keys are found, on average, half way along the list.

# Performance of Chaining Variant 1 [2]

---

SEARCH( $T$ , key): searching has expected cost  $O(1+\alpha)$ , worse case  $\Theta(n)$ , for the same reasons as INSERT.

Searching for keys that are present takes, on average, half the time to search for a key that is not present. This is not a good variant to use as a **negative cache** (holding a list of 'bad' items that you need to check against but will rarely match).

DELETE( $T$ , key): same as SEARCH, including that attempts to delete keys are not present takes twice as long, on average, as deletes for keys that are present.

## Chaining Variant 2: keep lists sorted by key

---

INSERT( $T$ , key, payload): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, replace the payload; at the end or if the next key is greater than the insert key, insert into the list here (INSERTION-SORT).

SEARCH( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, return the payload; stop if next key  $>$  search key.

DELETE( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for an existing cell with the same key. If found, remove cell from list. Stop if next key  $>$  search key.

💡 Insert means “ensure future searches this payload under this key”.

💡 Delete means “ensure future searches do not find this key”.

⚠️ This additionally requires that the keys are comparable: we are sorting keys (not hash values, since in any list, all keys have the same hash value!).

## Performance of Chaining Variant 2

---

INSERT( $T$ , key, payload): expected  $O(1+\alpha)$ ; worst  $\Theta(n)$ . Rebind and fresh inserts take the *same* time, on average, since both stop searching on average half way down the list when a larger key is encountered.

SEARCH( $T$ , key): same as INSERT.

DELETE( $T$ , key): same as INSERT.

## Chaining Variant 3: push on the head

---

INSERT( $T$ , key, payload): let  $k=h(\text{key})$ . Insert a new list cell on the head of list  $T[k]$ . The list may contain the same key repeatedly.

SEARCH( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for the first cell with the same key. If found, return the payload; at end of list, return NIL.

DELETE( $T$ , key): let  $k=h(\text{key})$ . Insert a new list cell on the head of list  $T[k]$  with the key to be deleted and NIL as the payload.

💡 Insert means “ensure future searches this payload under this key”.

💡 Delete means “ensure future searches do not find this key”.

# Performance of Chaining Variant 3

---

INSERT( $T$ , key, payload) and DELETE( $T$ , key):  $\Theta(1)$ , always.

SEARCH( $T$ , key):  $O(I+D)$ , where  $I$  and  $D$  are the number of times we have called INSERT and DELETE respectively. Grows over time: best for read-only tables!

⚠️ SEARCH  $\in O(n)$  is not a useful statement because 'n' would need to be the length of the list in slot  $T[k]$  but there is no way that any *user* of the hash table could know that value. That makes the statement meaningless to them!

💡 We must express running times as functions of values that an external user can reasonably know, such as the number of times they have invoked the operations.



## Chaining Variant 4: push on the head, delete from list

---

INSERT( $T$ , key, payload): let  $k=h(\text{key})$ . Insert a new list cell on the head of list  $T[k]$ . The list may contain the same key repeatedly.

SEARCH( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for the first cell with the same key. If found, return the payload; at end of list, return NIL.

DELETE( $T$ , key): let  $k=h(\text{key})$ . Walk down the list at  $T[k]$  looking for the first cell with the same key. If found, remove it from the list. NO-OP if not found.

💡 Insert means “ensure future searches this payload under this key”.

💡 Delete means “undo the most recent insert for this key”.

# Performance of Chaining Variant 4

---

INSERT( $T$ , key, payload):  $\Theta(1)$ , always.

SEARCH( $T$ , key):  $O(I)$ , where  $I$  is the number of times we have called INSERT. If we change DELETE to indicate whether it found the key to be deleted, we can say  $O(I-D)$ , where  $D$  is the number of times we have called DELETE.

DELETE( $T$ , key): same as SEARCH.

Notice that when we delete, the previous binding for that key “comes back to life”. The hash table now offers stack-like behaviour for each key.

# Collision resolution by Open Addressing

---

Keep (key, payload) tuples in the slots of table, T. Initialise all table slots to NIL.

If your slot is full, use another one ... but how to pick a different slot?

INSERT("Oxford", 1096)  
 $h(\text{"Oxford"}) \rightarrow 3$

	T
0	×
1	Cambridge, 1209
2	×
3	St Andrew's, 1413
4	Oxford, 1096
5	Edinburgh, 1583
6	×
7	×

# Open Addressing

---

INSERT( $T$ , key, payload): let  $k=h(\text{key})$ . If  $T[k] == \text{NIL}$  then insert in position  $k$ , otherwise use the **probe sequence** to find another empty slot.

**Linear Probing** is a probe sequence that tries slot  $(k+1)\%T.\text{size}$ ,  $(k+2)\%T.\text{size}$ , ... until a NIL slot is found; or if we wrap back around to slot  $k$ , fail as the table is full.

SEARCH( $T$ , key): start from slot  $k=h(\text{key})$  and use the same probe sequence if key is not found straight away. If the probe sequence reaches a NIL slot or gets back to slot  $k$ , return NIL (not found).

# Linear Probing

---

Open addressing reduces the **selectivity** of the hash function: when we search for a key, we end up having to compare the search key to keys that do not even have the same hash value. This is as if the chains in the previous method had got jumbled up! Linear probing is more vulnerable than other probe sequences.

The build-up of long probe sequences that do not even contain keys with the same hash value is called **primary clustering**.

Using a step size of 2 (or any other value) will not help. That just changes where the clustering appears, and might reduce the number of values that can be stored if the step size is not coprime with the table size.



**Selectivity** is how small is the subset of keys present in  $T$  that we must compare using key equality.

# Quadratic Probing

---

Instead of  $k, k+1, k+2, \dots$  we use this sequence for the  $i^{\text{th}}$  probe position:

$\text{probe}(T, \text{key}, i) = (h(\text{key}) + a i + b i^2) \% T.\text{size}$  for constants  $a, b$

This hits every position  $0..T.\text{size}-1$  in some order for  $i=0..T.\text{size}-1$  provided at least one of  $a, b$  is non-zero, and  $a, b$  are strictly less than  $T.\text{size}$ , which is prime.

Crucially, the order is different for each  $h(\text{key})$ , which solves primary clustering.

E.g.  $a=1, b=1$ : Starting at 4: 4, 6, 10, 16, ... Starting at 3: 3, 5, 9, 15, ... Starting at 2: 2, 4, 9, 14

Quadratic probing is prone to **secondary clustering** (a lesser problem): keys that hash to the same value will collide with each other at every probe position.

# Double Hashing

---

Double hashing solves primary clustering and secondary clustering.

$$\text{probe}(T, \text{key}, i) = (h_1(\text{key}) + i h_2(\text{key})) \% T.\text{size}$$

Now keys that collide under  $h_1$  almost surely do not also collide under  $h_2$  so their probe sequences step apart.

To avoid pathological values of  $h_2$  (e.g. 0, or any multiple of table size), we can use  $h_2(\text{key}) = (h'_2(\text{key}) \% (T.\text{size} - 1)) + 1$

Notice the additional computation cost of using a second hash function.

# Deleting with Open Addressing

You cannot simply remove keys: that would break the probe sequence!

Leave a marker behind: markers are **full and non-matching** when SEARCHing (never what you seek but don't stop the search, unlike a NIL slot) and are **empty** when INSERTing (can be overwritten, like a NIL slot).

0	×
1	Cambridge, 1209
2	×
3	St Andrew's, 1413
4	Oxford, 1096
5	Edinburgh, 1583
6	×
7	×

DELETE("St Andrew's")  
 $h(\text{"St Andrew's"}) \rightarrow 3$

0	×
1	Cambridge, 1209
2	×
3	Ⓢ
4	Oxford, 1096
5	Edinburgh, 1583
6	×
7	×

SEARCH("Oxford")  
 $h(\text{"Oxford"}) \rightarrow 3$   
Returns 1096. ✓



# Resizing with Open Addressing

---

To avoid overloading the table, maintain counters tracking:

- The number of keys,  $n_k$
- The number of markers,  $n_m$

**if**  $n_k + n_m > \text{THRESHOLD}$  **then**

**if**  $n_k \gg n_m$  **then** rehash into a larger table

**else if**  $n_k \approx n_m$  **then** rehash into a table of same size (remove markers)

**else** rehash into a smaller table

$\text{THRESHOLD}$  might be  $0.5 \times T.\text{size}$



Some implementations use an array of primes to use as larger/smaller table sizes.

# Software Engineering considerations

---

Hash tables can be difficult to test because the technical challenge is in the non-functional requirements of a hash function.

**Functional requirements:** must have type `String->int` – easily checked

**Non-functional requirements:** must output a (reasonably) uniform spread of hash values when given all the keys that my program is going to use it with.

If the hash function always returned 0, or the resize logic was broken (especially for chaining), your unit tests might not notice. Your users will when they enter their much larger data set! Hash tables are banned in some industry sectors.

# Summary of Hash Tables

---

- Expected  $O(1+\alpha)$  time for many variants, which can be  $O(1)$  if we manage the load factor carefully by resizing when the table is too full for the hash selectivity to remain performant.
- Collisions need to be managed carefully.
- Don't overlook the time required to run the hash function. It could dominate the expected runtime of a hash table.
- Software engineering considerations
- Need to perform equality checks on keys (during collision resolution), which might also be slow.

# Revision Guide / Summary of Algorithms 1 [1]

---

Methods to solve recurrence relations:

- Guess and verify
- Substitute and spot pattern, including the tree method to help spot patterns
- The Master Theorem

# Revision Guide / Summary of Algorithms 1 [2]

---

Growth orders:

- $f(n) \in o(g(n))$   $f(n)$  has strictly less rapid growth than  $g(n)$
- $f(n) \in O(g(n))$   $f(n)$ 's growth is upper-bounded by  $g(n)$ : *“at most as fast as”*
- $f(n) \in \Theta(g(n))$   $f(n)$  grows within a constant factor at the same rate as  $g(n)$
- $f(n) \in \Omega(g(n))$   $f(n)$ 's growth is lower-bounded by  $g(n)$ : *“at least as fast as”*
- $f(n) \in \omega(g(n))$   $f(n)$  has strictly more rapid growth than  $g(n)$

# Revision Guide / Summary of Algorithms 1 [3]

---

Algorithm designs:

- Incremental
- Divide and Conquer: non-overlapping subproblems
- Semi-structures
- Dynamic Programming
- Greedy Algorithms

# Revision Guide / Summary of Algorithms 1 [4]

---

## Dynamic Programming

- Useful for optimisation problems
- Requirements:
  - Optimal substructure
  - Overlapping subproblems
- Memoise previous results to avoid repeated recomputation
- Top-Down and Bottom-Up approaches

# Revision Guide / Summary of Algorithms 1 [5]

---

## Greedy Algorithms

- Not all problems can be solved greedily
- Greedy algorithms offer efficient solutions where they are possible
- Greedy algorithms solve optimisation problems where...
  - The locally optimal choice definitely does still allow an optimal solution to be reached
  - The combination of the locally optimal choice and an optimal solution to the subproblem that results from making the locally optimal first move, is optimal overall.
- Often, you need to think about what the 'greedy' choice is
  - E.g. choosing the earliest finishing time to allow the greatest cardinality set of activities overall



# Revision Guide / Summary of Algorithms 1 [6]

---

We looked into the details of several data structures...

- Pointers
- Stacks, stacks with additional operations (average, minimum/maximum)
- Queues
- Singly linked lists, cyclic singly linked lists
- Doubly linked lists, cyclic doubly linked lists
- Use in Doug Lea's malloc algorithm
- Priority Queues
- Hash Tables: chaining, open addressing, primary and secondary clustering

# Revision Guide / Summary of Algorithms 1 [7]

---

## Rooted Trees

- Varieties of rooted trees: with/out parent pointers, fixed/varying child count.
- Search trees don't allow duplicate keys!
- BSTs support a rich set of operations with expected runtime  $O(\lg n)$
- Simple logic in the operations – simple to code and small constant factors
- But  $O(n)$  worst case performance might leave algorithms using BSTs vulnerable to very bad worst case performance
- Memory overhead of 2 pointers per (key,value) pair might be significant (3 points if you need parent pointers).
- B-Trees, 2-3-4 Trees, Red-Black Trees, and their isomorphisms

# Revision Guide / Summary of Algorithms 1 [8]

Technique	Worst case	Average case	Stable	In-place	Notes
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓	Tight loop, fast for small n
MERGE-SORT	$\Theta(n \lg n)$	$\Theta(n \lg n)$	✓		Bottom-up Merge-Sort is in-place
HEAP-SORT	$O(n \lg n)$			✓	
QUICKSORT	$\Theta(n^2)$	$\Theta(n \lg n)$ expected		✓	
QUICKSORT (MoM)	$\Theta(n \lg n)$	$\Theta(n \lg n)$		✓	
COUNTING-SORT	$\Theta(k + n)$	$\Theta(k + n)$	✓		
RADIX-SORT	$\Theta(d(n + k))$	$\Theta(d(n + k))$	✓		
BUCKET-SORT	$\Theta(n^2)$	$\Theta(n)$ average	✓		