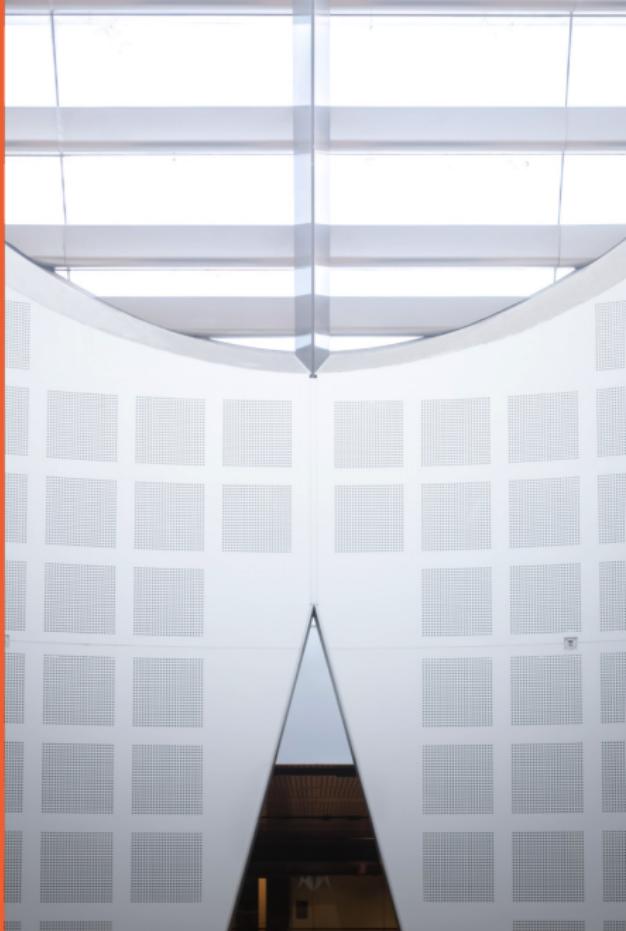


COMP2823

Introduction

Julián Mestre
School of Computer Science



Welcome to COMP2823

This class is your formal introduction to algorithms and data structures. Although you have been programming for a while and have been using algorithms and data structures, this class will lay the **foundations** for you to come up with your own algorithms and data structures and to be confident of their correctness and time complexity.

Welcome to COMP2823

This class is your formal introduction to algorithms and data structures. Although you have been programming for a while and have been using algorithms and data structures, this class will lay the **foundations** for you to come up with your own algorithms and data structures and to be confident of their correctness and time complexity.

After this class, getting your code to work will seem less like magic and more like science.



Data Structures and Algorithms (Adv)

This UoS comes in two flavors:

- COMP2123: Normal stream
- COMP2823: Advanced stream

What's the difference?

- COMP2823 has a separate lecture covering additional material
- COMP2823 assignments and final exam will be harder as they will test this additional material

Overview

Timetable:

- Main lecture: Tuesday 2p:00am-noon, PNR LT 2

Textbook (recommended but not mandatory):

- Algorithm Design and Applications by Goodrich and Tamassia

Systems:

- Canvas: Quizzes, and lecture recordings
- Ed: Discussion, slides, and assignments (programming)
- Gradescope: Assignments (written)

Assessments

Quizzes (worth 10%):

- 10 short equal-weight quizzes
- Open after lecture (starting Week 2), due before next lecture

Assignments (worth 30%):

- 6 short equal-weight assignments due on Tuesday
- either written (Gradescope) or programming (Ed)
- late submissions cost 20% of available marks per day

Final exam (worth 60%):

- closed book but allowed to bring one A4 sheet of paper
- 40% barrier

Late Submission Example

If you have not been granted special consideration (through the University):

- If your work would have scored 60% and is 1 hour late, you get 40%
- If your work would have scored 70% and is 28 hour late, you get 30%

Keep in mind:

Submission sites can become very slow near deadlines

Special Consideration

If your performance on assessments is affected by illness or misadventure:

Follow proper procedures

- Have a professional practitioner sign the USyd form
- Submit your application online, upload supporting documents
- Deadline is 3 days after assessment is due
- [https://sydney.edu.au/students/
special-consideration.html](https://sydney.edu.au/students/special-consideration.html)

There is a similar process for other reasons for special consideration, such as religious observance, military service, representative sports

Tutorials

We will post in Ed a tutorial sheet with exercises covering that week's material.

To get the most out of the tutorial, try to solve as many problems as you can before the tutorial. Your tutor is there to help you get unstuck, not to lecture.

At the end of the week, we will post solutions to selected exercises.

Communication

Email is an inefficient way to communicate in large classes such as ours. Unless yours is a personal issue, do not send us email.

If you have questions about the lecture materials, homework assignments, or any logistics related to the class, please use the Ed discussion forum so that others can benefit from the answers.

Finally, if you spot a question that you know the answer for, please free to answer. It helps your classmates but more importantly, writing your thoughts down helps you crystalize your understanding.

About the lecturer

Since we will be working hard together, it would be nice to get to know each other a bit better. Even though I cannot really ask you all to introduce yourselves, I can still introduce myself:

- From Argentina, but lived in America, Germany, and Australia
- Joined Uni Sydney in 2010 and taught Algorithms several times
- I do research on combinatorial optimization
- Since 2017 I've been splitting my time between the University and Facebook Research
- I spend most of my free time with my family
- I enjoy the outdoors (gardening, running, hiking, and sailing)

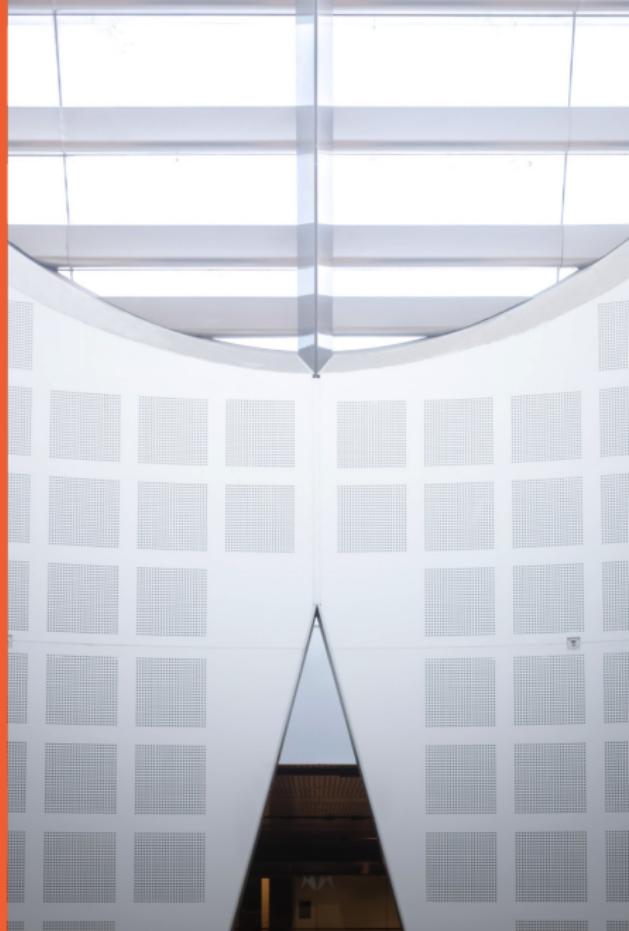
COMP2823

Algorithm analysis (GT 1.1.5-1.1.6, 1.3)

Julián Mestre
School of Computer Science



THE UNIVERSITY OF
SYDNEY



Three abstractions

Computational problem:

- defines a computational task
- specifies what the input is and what the output should be

Algorithm:

- a step-by-step recipe to go from input to output
- different from implementation

Correctness and complexity analysis:

- a formal proof that the algorithm solves the problem
- analytical bound on the resources it uses

Example computational problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], \dots, A[n - 1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i + 1] + \cdots + A[j]$$

Naive algorithm

High level description:

- Iterate over every pair $0 \leq i < j < n$.
- For each compute $A[i] + A[i + 1] + \dots + A[j]$
- Return the pair with the maximum value

Naive algorithm

```
1 def opt_strategy(A):
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     for i in [0:n]:
7         for j in [i+1:n]:
8             aux = sum(A[x] for x in [i:j+1])
9             if aux > curr_val:
10                 curr_val, curr_ans = aux, (i, j)
11
12 return curr_ans
```

Pre-processing algorithm

We can evaluate $A[i] + A[i + 1] + \cdots + A[j]$ faster if we do some pre-processing.

- Pre-compute $B[i] = A[0] + A[1] + \cdots + A[i - 1]$ using

$$B[i] = \begin{cases} 0 & \text{if } i = 0 \\ B[i - 1] + A[i - 1] & \text{if } i > 0 \end{cases} \quad (1)$$

- Iterate over every pair $0 \leq i < j < n$.
- For each compute
$$B[j + 1] - B[i] = A[i] + A[i + 1] + \cdots + A[j]$$
- Return the pair with the maximum value

Pre-processing algorithm

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]
11     for i in [0:n]:
12        for j in [i+1:n]:
13            aux = B[j + 1] - B[i]
14            if aux > curr_val:
15                curr_val, curr_ans = aux, (i, j)
16
17    return curr_ans
```

Efficiency

Definition (first attempt)

An algorithm is efficient if it runs quickly on real input instances

Not a good definition because it is not easy to evaluate:

- instances considered
- implementation details
- hardware it runs on

Our definition should be implementation and context independent:

- count number of “steps”
- bound the algorithm’s worst-case performance

Efficiency

Definition (second attempt)

An algorithm is efficient if it achieves qualitatively better worst-case performance than a brute-force approach

Not a good definition because it is subjective:

- brute-force approach is ill-defined
- qualitatively better is ill-defined

Our definition should be objective:

- not tied to a straw man baseline
- independently agreed upon

Efficiency

Definition

An algorithm is efficient if it runs in polynomial time; that is, on any instance of size n , it performs no more than $p(n)$ steps for some polynomial $p(x) = a_dx^d + \cdots + a_1x + a_0$.

This gives us information about the worst-case behavior of the algorithm, which is useful for making predictions and comparing different algorithms.

Asymptotic growth analysis

Definition

We say that $T(n) = O(f(n))$ if
there exists $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$

Definition

We say that $T(n) = \Omega(f(n))$ if
there exists $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$

Definition

We say that $T(n) = \Theta(f(n))$ if
 $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of size n .

If $T(n) = \Theta(p(n))$ where p is a polynomial of degree d , then doubling the size of the input should roughly increase the running time by a factor of 2^d :

$$\lim_{n \rightarrow \infty} \frac{T(2n)}{T(n)} = \lim_{n \rightarrow \infty} \frac{p(2n)}{p(n)} = \lim_{n \rightarrow \infty} \frac{(2n)^d}{n^d} = 2^d.$$

Asymptotic growth analysis gives us a tool for focusing on the term of $T(n)$ that dominates the running time

Examples of asymptotic growth

Polynomial

$O(n^c)$, consider efficient since most algorithms have small c

Logarithmic

$O(\log n)$, typical for search algorithms like Binary Search

Exponential

$O(2^n)$, typical for brute force algorithms exploring all possible combinations of elements

Comparison of running times

size	n	$n \log n$	n^2	n^3	2^n	$n!$
10	< 1s	< 1s	< 1s	<1s	<1s	3s
50	< 1s	< 1s	< 1s	<1s	17m	-
100	< 1s	< 1s	< 1s	1s	35y	-
1,000	< 1s	< 1s	1s	15m	-	-
10,000	< 1s	< 1s	2s	11d	-	-
100,000	< 1s	1s	2h	31y	-	-
1,000,000	1s	10s	4d	-	-	-

Properties of asymptotic growth

Transitivity:

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$

Sums of functions:

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$
- If $f = \Omega(h)$ then $f + g = \Omega(h)$

Asymptotic analysis is a powerful tool that allows us to ignore unimportant details and focus on what's important.

Survey of common running times

Let $T(n)$ be the running time of our algorithm.

We say that $T(n)$ is ... if ...

logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

Recall stock trading problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], \dots, A[n - 1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i + 1] + \cdots + A[j]$$

Naive algorithm analysis

```
1 def opt_strategy(A):
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     for i in [0:n]:
7         for j in [i+1:n]:
8             aux = sum(A[x] for x in [i:j+1])
9             if aux > curr_val:
10                 curr_val, curr_ans = aux, (i, j)
11
12     return curr_ans
```

Naive algorithm analysis

```
1 def opt_strategy(A):
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)    } O(1)
5     n = len(A)
6     for i in [0:n]:
7         for j in [i+1:n]:
8             aux = sum(A[x] for x in [i:j+1])
9             if aux > curr_val:
10                 curr_val, curr_ans = aux, (i, j)
11
12     return curr_ans
```

Naive algorithm analysis

```
1  def opt_strategy(A):
2      # find (i, j) maximizing A[i] + ... + A[j]
3
4      curr_val, curr_ans = 0, (None, None)    } O(1)
5      n = len(A)
6      for i in [0:n]:
7          for j in [i+1:n]:
8              aux = sum(A[x] for x in [i:j+1])
9              if aux > curr_val:
10                  curr_val, curr_ans = aux, (i, j)
11
12  return curr_ans                         } O(n^2)
```

Naive algorithm analysis

```
1  def opt_strategy(A):
2      # find (i, j) maximizing A[i] + ... + A[j]
3
4      curr_val, curr_ans = 0, (None, None)    } O(1)
5      n = len(A)
6      for i in [0:n]:
7          for j in [i+1:n]:
8              aux = sum(A[x] for x in [i:j+1])   } O(j - i)
9              if aux > curr_val:
10                  curr_val, curr_ans = aux, (i, j)
11
12  return curr_ans                         } O(1)
```

Naive algorithm analysis

```
1  def opt_strategy(A):
2      # find (i, j) maximizing A[i] + ... + A[j]
3
4      curr_val, curr_ans = 0, (None, None)    } O(1)
5      n = len(A)
6      for i in [0:n]:
7          for j in [i+1:n]:
8              aux = sum(A[x] for x in [i:j+1])   } O(j - i)
9              if aux > curr_val:
10                  curr_val, curr_ans = aux, (i, j) } O(j - i)
11
12      return curr_ans                      } O(1)
```

Naive algorithm analysis

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n .

$$T(n) = O(1) + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(j - i)$$

Naive algorithm analysis

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(j - i) \\ &= O(1) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} O(n) \\ &= O(1) + \sum_{i=0}^{n-1} O(n^2) \\ &= O(1) + O(n^3) \\ &= O(n^3) \end{aligned}$$

See GT 1.2 for a refresher if needed.

Aren't we being too pessimistic?

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n .

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Omega(j - i)$$

Aren't we being too pessimistic?

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Omega(j - i) \\ &= \sum_{i=0}^{n/4} \sum_{j=3n/4}^{n-1} \Omega(j - i) \\ &= \sum_{i=0}^{n/4} \sum_{j=3n/4}^{n-1} \Omega(n/2) \\ &= \Omega(n^3) \end{aligned}$$

Aren't we being too pessimistic?

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Omega(j - i) \\ &= \sum_{i=0}^{n/4} \sum_{j=3n/4}^{n-1} \Omega(j - i) \\ &= \sum_{i=0}^{n/4} \sum_{j=3n/4}^{n-1} \Omega(n/2) \\ &= \Omega(n^3) \end{aligned}$$

Thus, the running time of Naive is $\Theta(n^3)$.

Pre-processing algorithm analysis

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]
11        for i in [0:n]:
12            for j in [i+1:n]:
13                aux = B[j + 1] - B[i]
14                if aux > curr_val:
15                    curr_val, curr_ans = aux, (i, j)
16
17    return curr_ans
```

Pre-processing algorithm analysis

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]                      } O(1)
11        for i in [0:n]:
12            for j in [i+1:n]:
13                aux = B[j + 1] - B[i]
14                if aux > curr_val:
15                    curr_val, curr_ans = aux, (i, j)
16
17    return curr_ans
```

Pre-processing algorithm analysis

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]                      } O(1)
11        for i in [0:n]:
12            for j in [i+1:n]:
13                aux = B[j + 1] - B[i]
14                if aux > curr_val:
15                    curr_val, curr_ans = aux, (i, j)
16
17    return curr_ans
```

Pre-processing algorithm analysis

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]                      } O(1)
11        for i in [0:n]:
12            for j in [i+1:n]:
13                aux = B[j + 1] - B[i]
14                if aux > curr_val:                  } O(1)
15                    curr_val, curr_ans = aux, (i, j)
16
17    return curr_ans
```

Pre-processing algorithm analysis

```
1 def opt_strategy(A)
2     # find (i, j) maximizing A[i] + ... + A[j]
3
4     curr_val, curr_ans = 0, (None, None)
5     n = len(A)
6     # computer B[i] = A[0] + .. + A[i-1]
7     B = new array[n + 1]
8     B[0] = 0
9     for i in [0:n]:
10        B[i+1] = B[i] + A[i]           } O(1)      } O(n)
11     for i in [0:n]:
12         for j in [i+1:n]:
13             aux = B[j + 1] - B[i]    } O(1)      } O(n2)
14             if aux > curr_val:
15                 curr_val, curr_ans = aux, (i, j)
16
17 return curr_ans
```

Pre-processing algorithm analysis

```
1  def opt_strategy(A)
2      # find (i, j) maximizing A[i] + ... + A[j]
3
4      curr_val, curr_ans = 0, (None, None)
5      n = len(A)
6      # computer B[i] = A[0] + .. + A[i-1]
7      B = new array[n + 1]                      } O(1) or O(n)?
8      B[0] = 0
9      for i in [0:n]:
10         B[i+1] = B[i] + A[i]                  } O(1)   } O(n)
11         for i in [0:n]:
12             for j in [i+1:n]:
13                 aux = B[j + 1] - B[i]
14                 if aux > curr_val:
15                     curr_val, curr_ans = aux, (i, j) } O(1)   } O(n2)
16
17     return curr_ans
```

Pre-processing algorithm analysis

Let $T(n)$ be the running time of the Pre-processing Algorithm on an instance of size n .

$$T(n) = O(1) + \sum_{i=0}^n O(1) + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(1)$$

Pre-processing algorithm analysis

Let $T(n)$ be the running time of the Pre-processing Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^n O(1) + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(1) \\ &= O(1) + \sum_{i=0}^n O(1) + \sum_{i=0}^{n-1} \sum_{j=1}^{n-1} O(1) \\ &= O(1) + O(n) + O(n^2) \\ &= O(n^2) \end{aligned}$$

A major improvement from the $O(n^3)$ time Naive algorithm!

Re-use computation

For a fixed i , we can evaluate $\max_{i < j < n} B[j + 1] - B[i]$ faster if we re-use some of computation already done.

- Pre-compute $C[i] = \max_{i < j \leq n} B[j]$ using

$$C[i] = \begin{cases} B[n] & \text{if } i = n \\ \max(B[i], C[i + 1]) & \text{if } i < n \end{cases} \quad (2)$$

- Iterate over every $0 < i < n$.
- For each compute $C[i] - B[i] = \max_{i < j < n} A[i] + \cdots + A[j]$
- Return maximum value

Re-use computation algorithm

```
1 def opt_strategy(A)
2     # --- snip ---
3
4     # computer C[i] = max B[j] for i < j ≤n
5     C = new array[n+1]
6     C[n] = B[n]
7     for i in [0:n:-1]:
8         C[i] = max(B[i], C[i+1])
9     for i in [0:n]:
10        aux = C[i] - B[i]
11        if aux > curr_val:
12            curr_val = aux
13
14    return curr_val
```

Re-use computation algorithm

```
1 def opt_strategy(A)
2     # --- snip ---
3
4     # computer C[i] = max B[j] for i < j ≤n
5     C = new array[n+1]
6     C[n] = B[n]
7     for i in [0:n:-1]:
8         C[i] = max(B[i], C[i+1])
9     for i in [0:n]:
10        aux = C[i] - B[i]
11        if aux > curr_val:
12            curr_val = aux
13
14 return curr_val
```

But what about finding the indices?

Re-use computation algorithm analysis

Let $T(n)$ be the running time of the Pre-processing Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^n O(1) + \sum_{i=0}^n O(1) \\ &= O(1) + O(n) + O(n) \\ &= O(n) \end{aligned}$$

Re-use computation algorithm analysis

Let $T(n)$ be the running time of the Pre-processing Algorithm on an instance of size n .

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^n O(1) + \sum_{i=0}^n O(1) \\ &= O(1) + O(n) + O(n) \\ &= O(n) \end{aligned}$$

A massive improvement from the $O(n^3)$ time Naive algorithm!!!!

Recap

Asymptotic time complexity gives us some information about the expected behavior of the algorithm. It is useful for making predictions and comparing different algorithms.

Why do we make a distinction between problem, algorithm, implementation and analysis?

- somebody can design a better algorithm for a given problem
- somebody can come up with better implementation
- somebody can come up with better analysis

A note on style

For your assessments, you will have to design and analyze an algorithm for a given problem. This always consists of three steps:

- Describe your algorithm: A high level description in **English**, optionally followed by pseudocode. Never submit code!
- Prove its correctness: A formal proof that the algorithm does what it's supposed to do.
- Analyze its time complexity: A formal proof that the algorithm runs in the time you claim it does.

Try to model your own solution after the solution published for the tutorial sheets. You are encouraged to use LaTeX.

A note on pseudocode style

We will be using in this class follows closely the Python syntax:

- Arrays: we use zero-based indexing
- Slices: `[i:j:k]` is equivalent to Python's `range(i, j, k)`.
- References: Every non-basic data type is passed by reference

But we will deviate when writing things in plain English leads to easier to understand code.

This week

Tutorial sheet 1:

- posted on Tue 26 Feb
- make sure you work out a few problems before the tutorial, especially the warm-up

Assignment 1:

- posted on Wed 27 Feb
- due on Tue 5 Mar

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 2a: Lists

[GT 2.2]

Dr. Julian Mestre

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Abstract Data Types (ADT)

Type defined in terms of its data items and associated operations, **not its implementation**.

ADTs are supported by many languages, including Python.

Abstract Data Types (ADT)

Type defined in terms of its data items and associated operations, **not its implementation**.

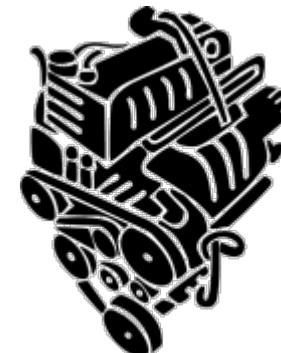
Simple example: Driving a car



interface



implementation



Benefits of ADT approach

- Code is easier to understand if different functionalities are separated into different places.
- Many different systems can use the same library, so only code tricky manipulations once, rather than in every system.
- There can be choices of implementations with different performance tradeoffs, and the client doesn't need to be rewritten extensively to change which implementation it uses.
- Clients using the ADT need to learn the interface only once

Example: Reservation system

- We have a theatre with 500 named seats, e.g., “N31”
- What kind of data should be stored?
 - Seats names
 - Seats reserved or available.
 - If reserved, name of the person who reserved the seat.
- Operations needed?



Example: Reservation system

- Operations needed?
 - `capacity_available()` : number of available seats (integer)
 - `capacity_sold()` : number of seats with reservations
 - `customer(x)` : name of customer who bought seat x
 - `release(x)` : make seat x available (ticket returned)
 - `reserve(x, y)` : customer y buys ticket for seat x
 - `add(x)` : install new seat whose id is x
 - `get_available()`: access available seats



ADT challenges

- Specify how to deal with the boundary cases
 - what to do if `reserve(x, y)` is invoked when `x` is already occupied?
 - what other cases can you think of?
- Do we need a new ADT? Could we use an existing one, perhaps by renaming the operations and tweaking the error-handling?
 - “Adapter” design pattern (see SOFT2201)
 - Could this example be mapped to an ADT you already know?

Abstract data types and Data structures

An **abstract data type (ADT)** is a specification of the desired behaviour from the point of view of the user of the data.

A **data structure** is a concrete representation of data, and this is from the point of view of an implementer, not a user.

Distinction is subtle but similar to the difference between a computational problems and an algorithm.

ADT in programming (Python)

- ADT is given as an *abstract base class* (*abc*)
- An abc declares methods (with their names and signatures) usually without providing code and we can't construct instances
- A data structure implementation is a class that inherits from the abc, provides code for all the required methods (and perhaps others) and has a constructor
- Client code can have variables that are instances of the data structure class and can call methods on these variables

Pseudocode

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

Pseudocode Style

Conditional branching:

```
if [condition] then  
    [true branch]  
else  
    [false branch]
```

Looping:

```
while [condition] do  
    [while loop body]  
  
for x in container do  
    [for loop body]
```

Function definition:

```
def [func name]([args])  
    [func body]  
    return [value]
```

Slices:

```
for x in array[a:b] do  
for i in [a:b] do
```

[a:b]
denotes indices
[a, a+1, ..., b-1]
just like Python slices

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Greatest Common Divisor - Java

```
public static int GCD(int a, int b) {  
    if(b == 0){  
        return a;  
    }  
    return GCD(b, a % b);  
}
```

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Greatest Common Divisor - Python

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of **instructions** that does not have to use specific syntax.

Greatest Common Divisor - Pseudocode

```
def gcd(a, b)
    # input: integers a and b
    # output: greatest common divisor of a and b
    if b == 0 then
        return a
    else
        return gcd(b, remainder of a / b)
```

Index-Based Lists (List ADT)

An index-based list supports the following operations:

size() (int) number of elements in the store

isEmpty() (boolean) whether or not the store is empty

get(i) return element at index i

set(i, e) replace element at index i with element e,
and return element that was replaced

insert(i, e) insert element e at index i existing elements with
index $\geq i$ are shifted up

remove(i) remove and return the element at index i existing
elements with index $\geq i$ are shifted down

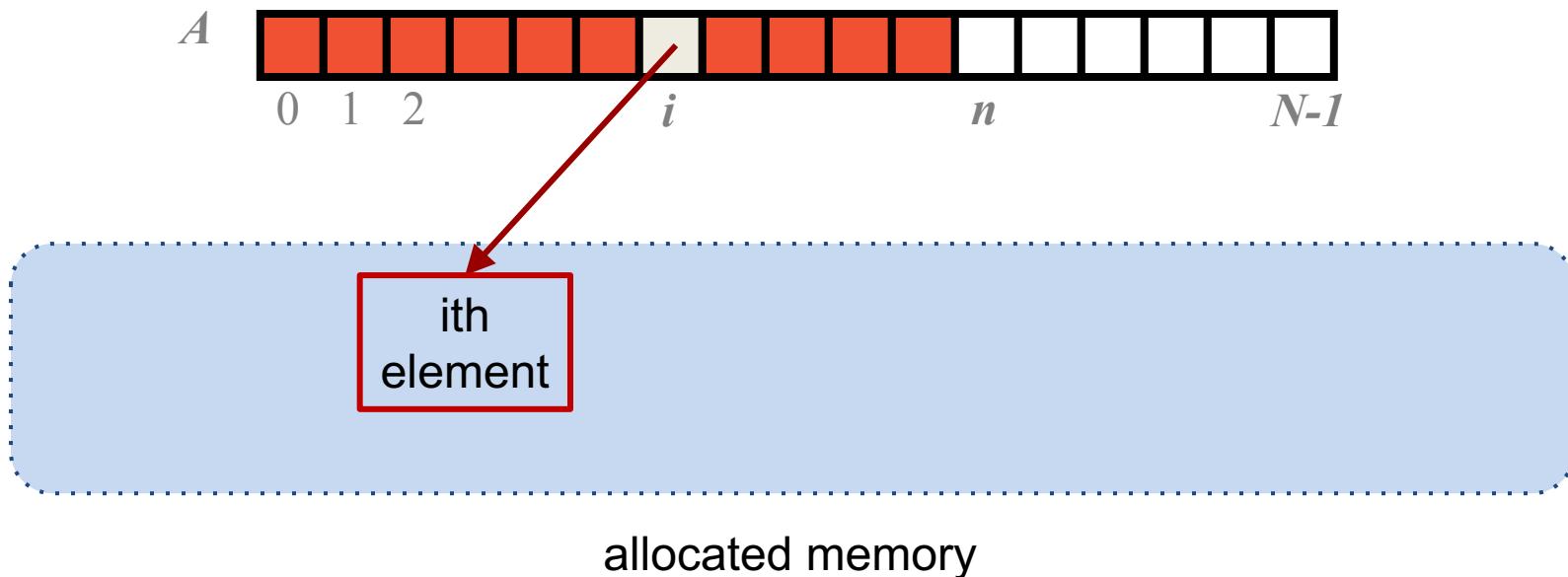
Example

A sequence of List operations:

Method	Returned value	List content
insert(0,A)	-	[A]
insert(0,B)	-	[B, A]
get(1)	A	[B, A]
set(2,C)	“error”	[B, A]
insert(2,C)	-	[B, A, C]
insert(4,D)	“error”	[B, A, C]
remove(1)	A	[B, C]
insert(1,D)	-	[B, D, C]
insert(1,E)	-	[B, E, D, C]
get(4)	“error”	[B, E, D, C]
insert(4,F)	-	[B, E, D, C, F]
set(2,G)	D	[B, E, G, C, F]

Array-based Lists

An option for implementing the list ADT is to use an array A , where $A[i]$ stores (a reference to) the element with index i .
If array has size N then we can represent lists of size $n < N$

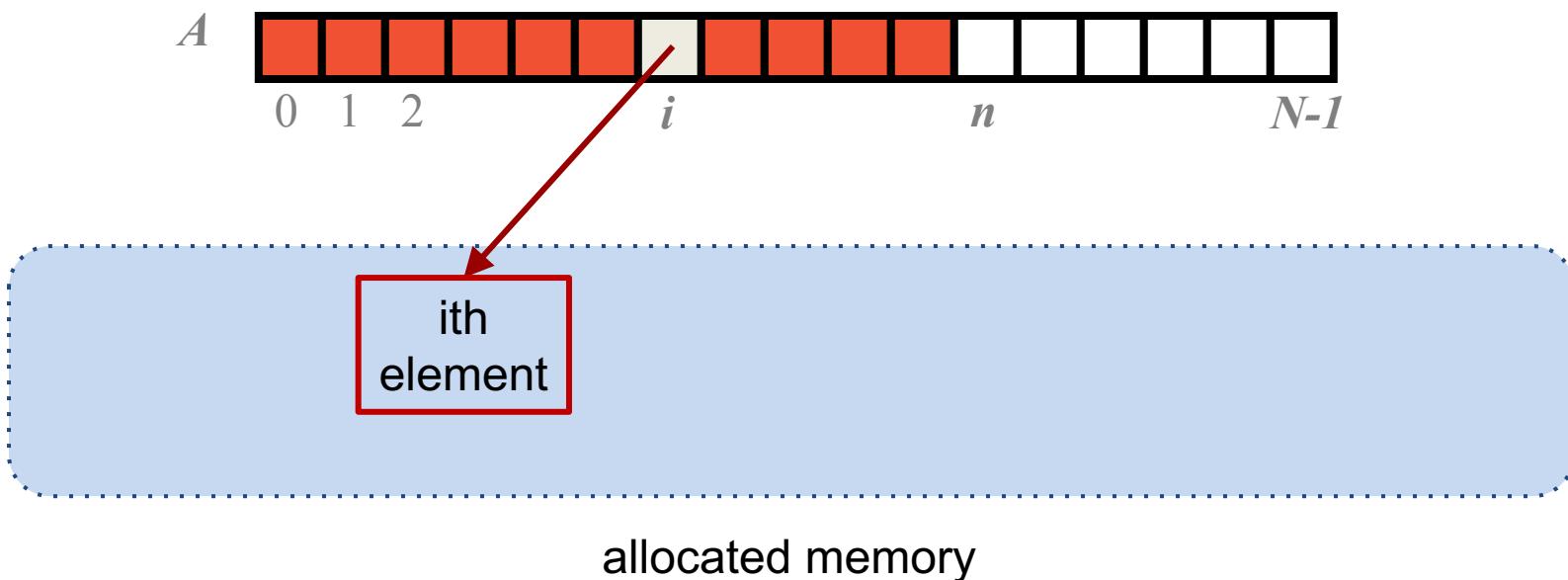


Array-based Lists: get(i)

The `get(i)` and `set(i, e)` methods are easy to implement by accessing `A[i]`

Must check that i is a legitimate index ($0 \leq i < n$)

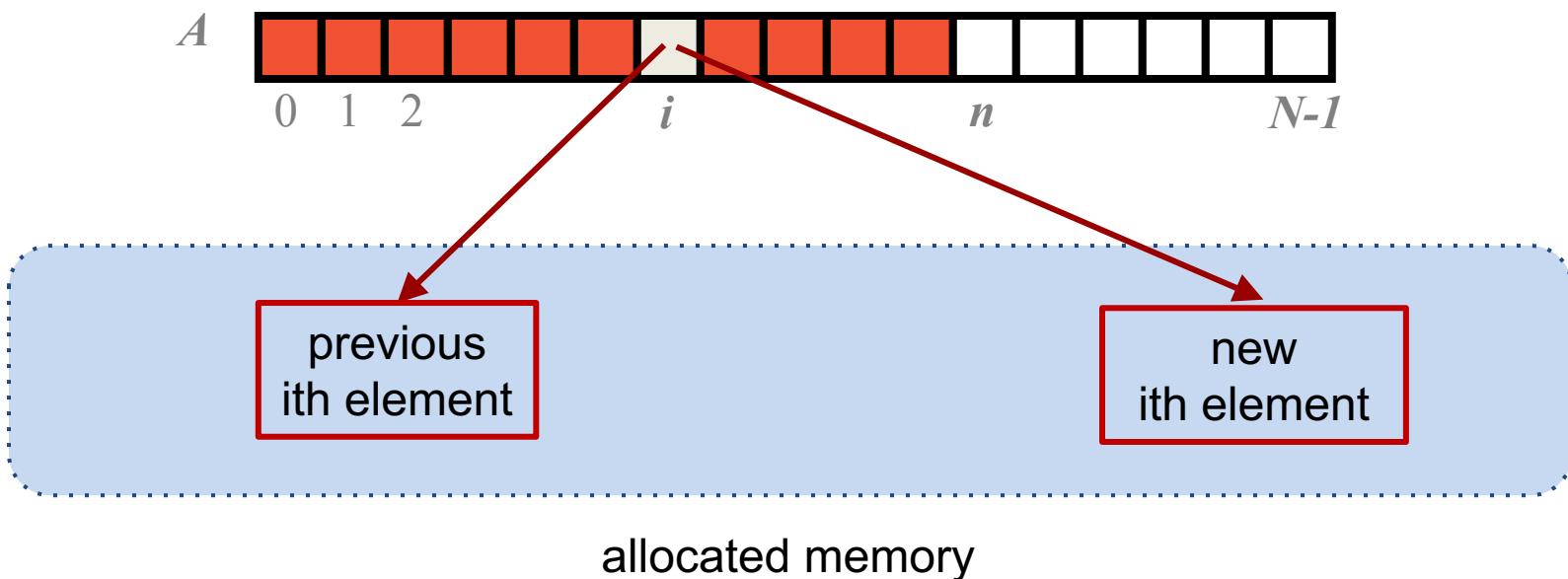
Both operations can be carried out in constant time
(a.k.a. $\mathcal{O}(1)$ time), independent of the size of the array



Array-based Lists: set(i,e)

The `get(i)` and `set(i, e)` methods are easy to implement by accessing $A[i]$

Must check that i is a legitimate index ($0 \leq i < n$)



Pseudo-code for get

```
def get(i):
    # input: index i
    # output: ith element in list
    if i < 0 or i >= n then
        return "index out of bound"
    else
        return A[i]
```

Time complexity of this operation is $O(1)$ time, independent of the size of the array (N) or the represented list (n)

Pseudo-code for set

```
def set(i, e):
    # input: index i and value e
    # do: update ith element in list to e
    if i < 0 or i >= n then
        return "index out of bound"
    A[i] = e
```

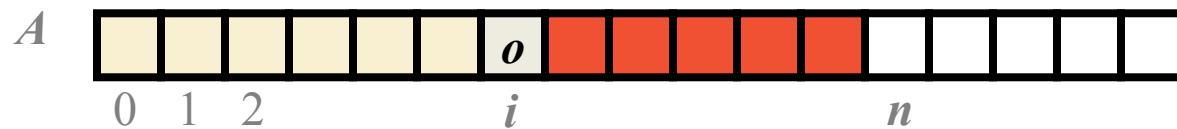
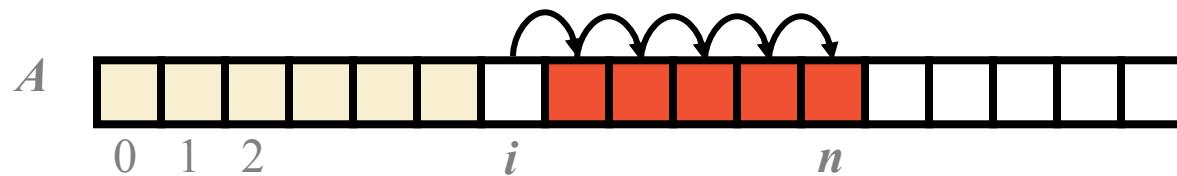
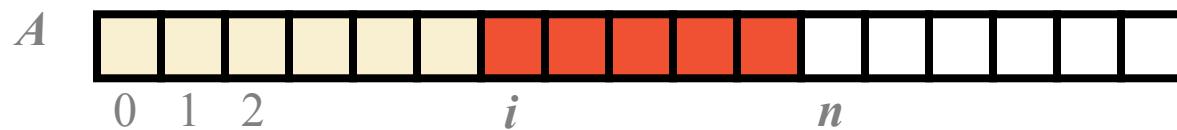
Time complexity of operation is $O(1)$ time, independent of the size of the array (N) or the represented list (n)

Array-based Lists: add(i,e)

In an operation `insert(i, e)`, we must make room for the new element by shifting forward elements $A[i], \dots, A[n - 1]$

Must check that there is space ($n < N$)

What is the most time consuming scenario?



Pseudo-code for insertion

```
def insert(i, e):
    if i < 0 or i > n
        return "index out of bound"
    if n == N then
        return "array is full"

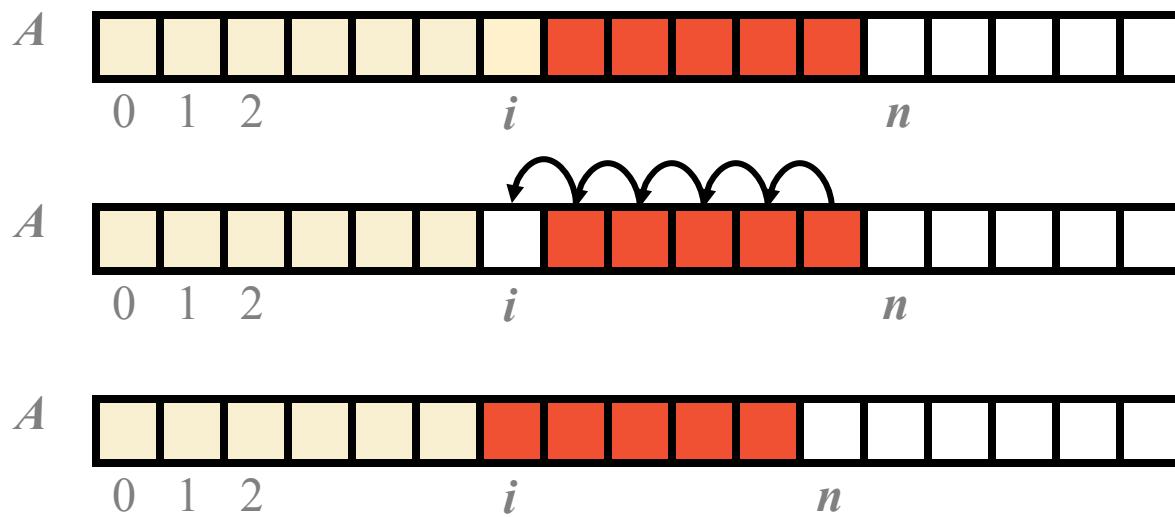
    for j in [n:i:-1] do
        A[j] = A[j-1]
    A[i] = e
    n = n + 1
```

Time complexity is $O(n)$ in the worst case

Array-based Lists: remove(i)

In an operation $\text{remove}(i)$, we need to fill the hole left at position i by shifting backward elements $A[i + 1], \dots, A[n - 1]$

Must check that i is a legitimate index ($0 \leq i < n$)



Pseudo-code for removal

```
def remove(i):
    if i < 0 or i >= n
        return "index out of bound"
    e = A[i]
    if i < n-1
        for j in [i:n-1] do
            A[j] ← A[j+1]
    n = n - 1
    return e
```

Time complexity is $O(n)$ in the worst case

Summary of (static) array-based Lists

Limitations:

- can represent lists up to the capacity of the array (n vs N)

Space complexity:

- space used is $O(N)$, whereas we would like it to be $O(n)$

Time complexity:

- both **get** and **set** take $O(1)$ time
- both **add** and **remove** take $O(n)$ time in the worst case

Dynamic array

Instead of using an array of fixed length, we can get a larger array each time we run out of space, and copy existing element there

This approach is called a **dynamic array** because the size of the array changes dynamically throughout the execution

Python's list type are implemented this way

Copying elements to new array makes the add operation sometimes much slower, but worst-case time is still $O(n)$

We can also shrink array if we want to be memory efficient, but we need to be careful how often we change array

Summary of dynamic array-based Lists

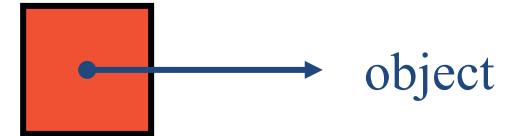
Space complexity:

- space used is $O(n)$

Time complexity:

- both **get** and **set** take $O(1)$ time
- both **add** and **remove** take $O(n)$ time in the worst case

Positional Lists



ADT for a list where we store elements at “positions”

Position models the abstract notion of place where a single object is stored within a container data structure.

Unlike index, this keeps referring to the same entry even after insertion/deletion happens elsewhere in the collection.

Position offers just one method:

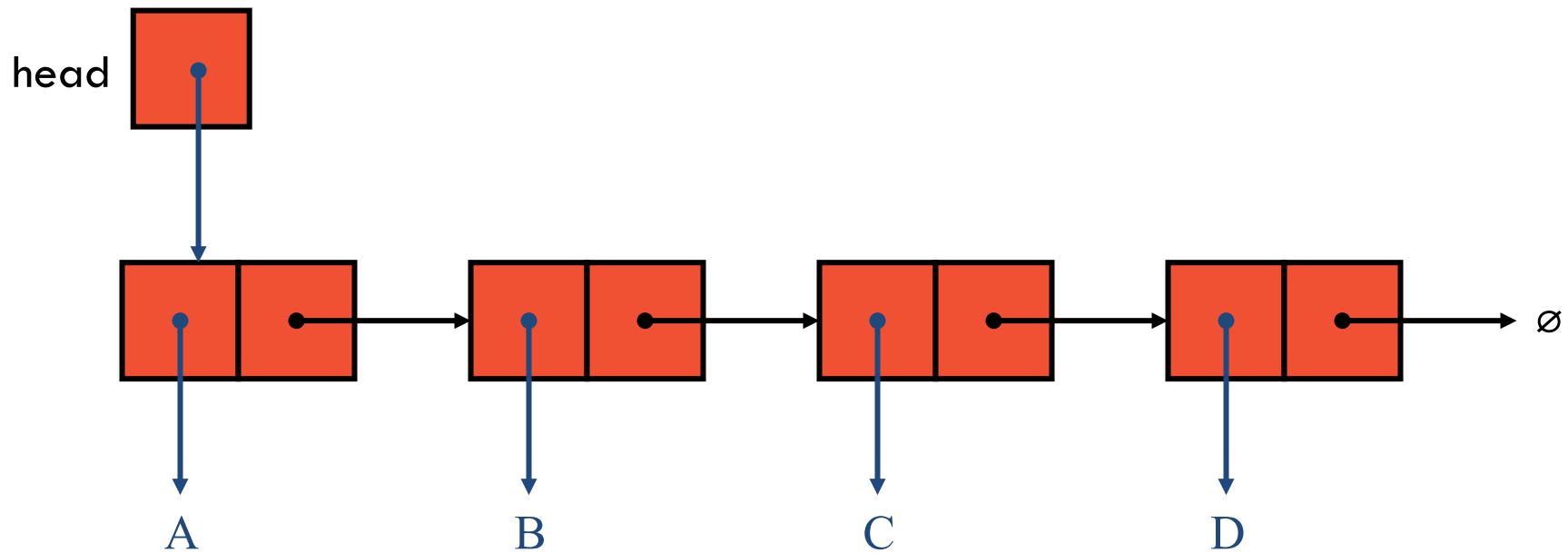
`element()` : return the element stored at the position instance

Positional Lists - Operations

size()	(int) number of elements in the store
isEmpty()	(boolean) whether or not the store is empty
first()	return position of first element (null if empty)
last()	return position of last element (null if empty)
before(p)	return position immediately before p (null if p is first)
after(p)	return position immediately after p (null if p last)
insertBefore(p, e)	insert e in front of the element at position p
insertAfter(p, e)	insert e following the element at position p
remove(p)	remove and return the element at position p

Singly Linked List

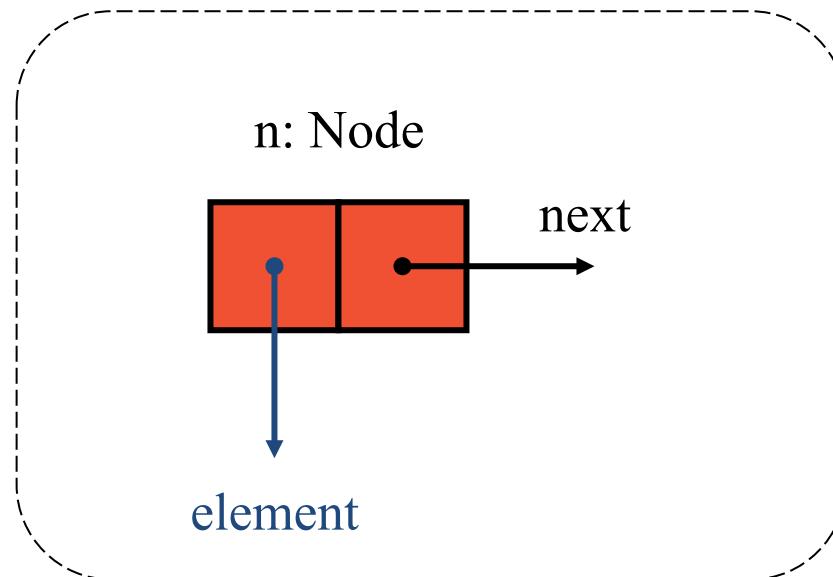
- A concrete data structure
- A sequence of **Nodes**, each with a reference to the next node
- List captured by reference (head) to the first **Node**



Node implements Position

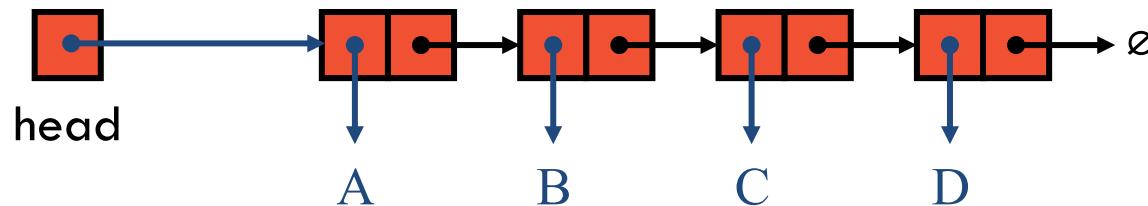
Each **Node** in a singly linked List stores

- its element, and
- a link to the next node.



Advice on working with linked structures

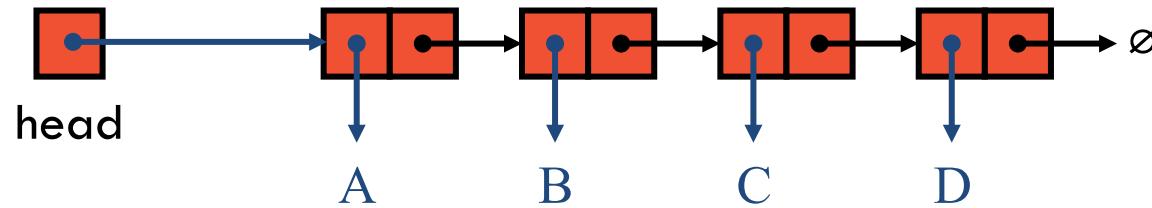
- Draw the diagram showing the state.
- Show a location where you place carefully each of the instance variables (including references to nodes).
- Be careful to step through dotted accesses e.g. **p.next.next**
- Be careful about assignments to fields e.g.
p.next = q or **p.next.next = r**



first()

first() : return **position** of first element (null if empty)

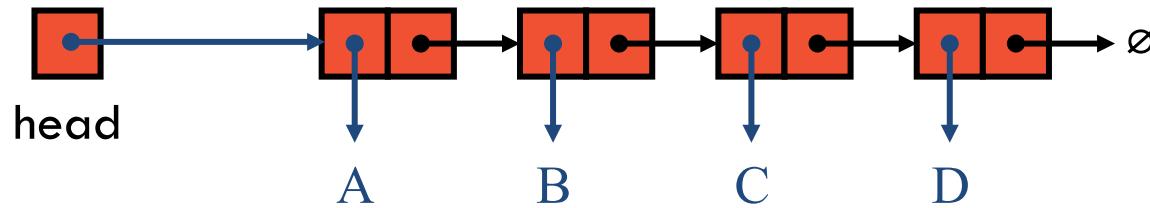
return?



first()

first() : return position of first element (null if empty)

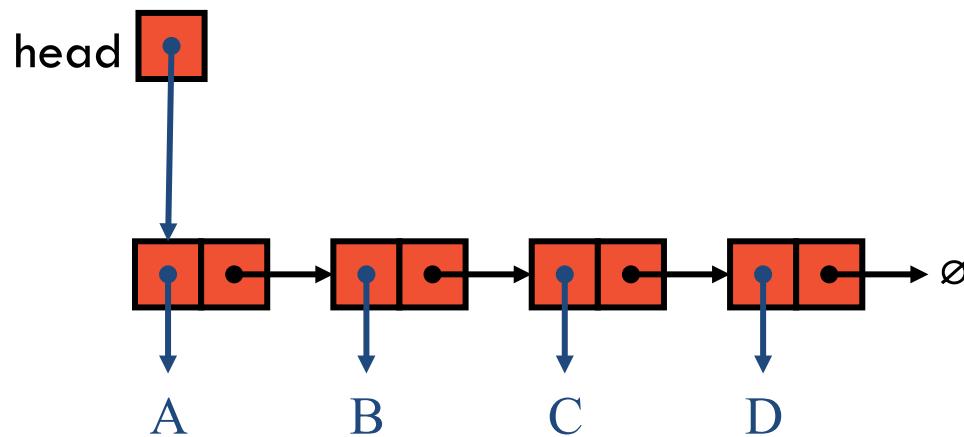
return head



Time complexity?

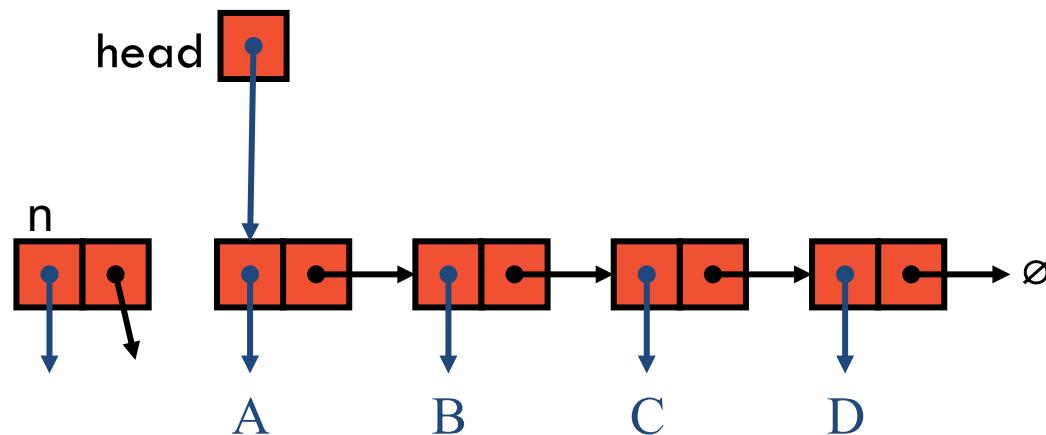
insertFirst(e)

1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n



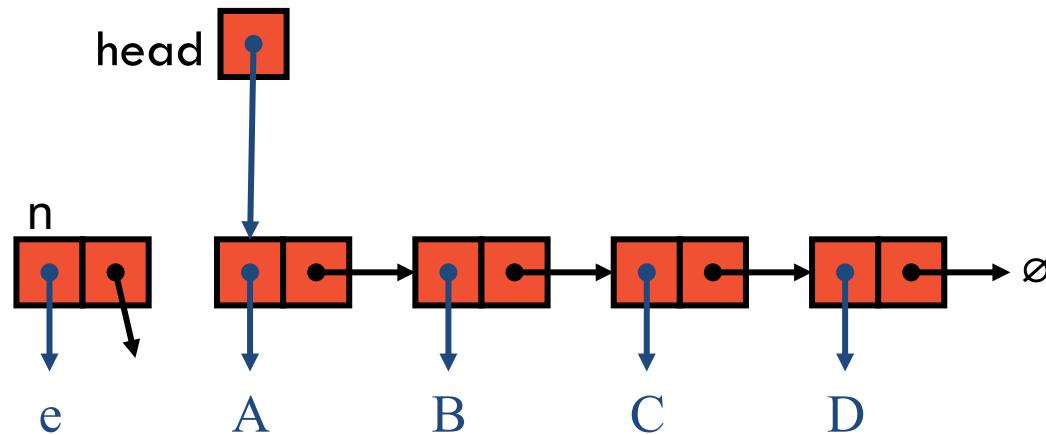
insertFirst(e)

1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n



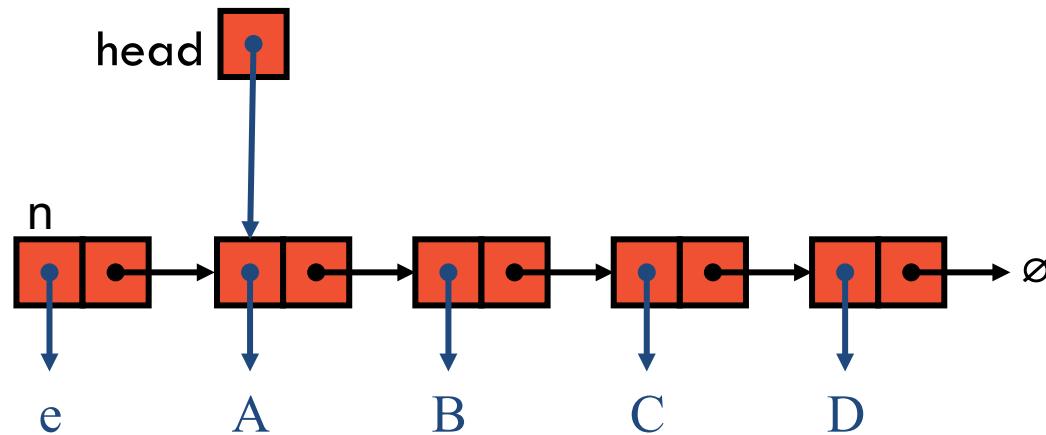
insertFirst(e)

1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n



insertFirst(e)

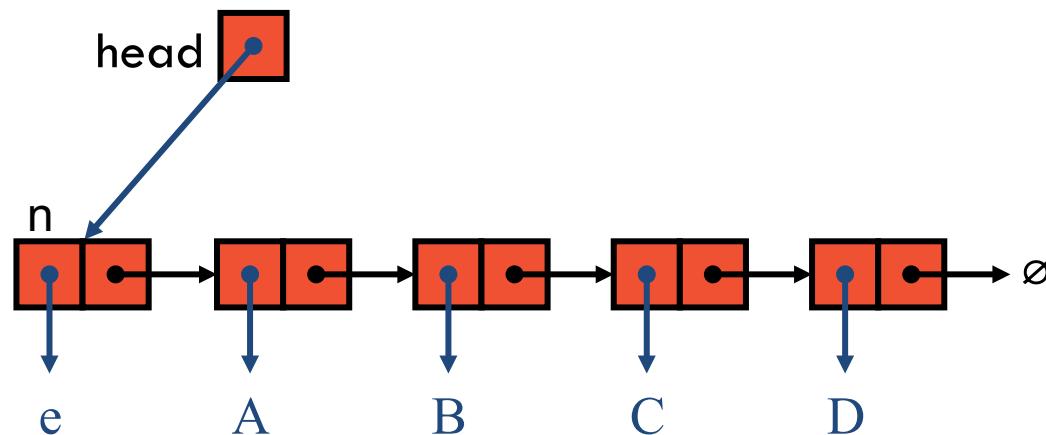
1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n



insertFirst(e)

1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n

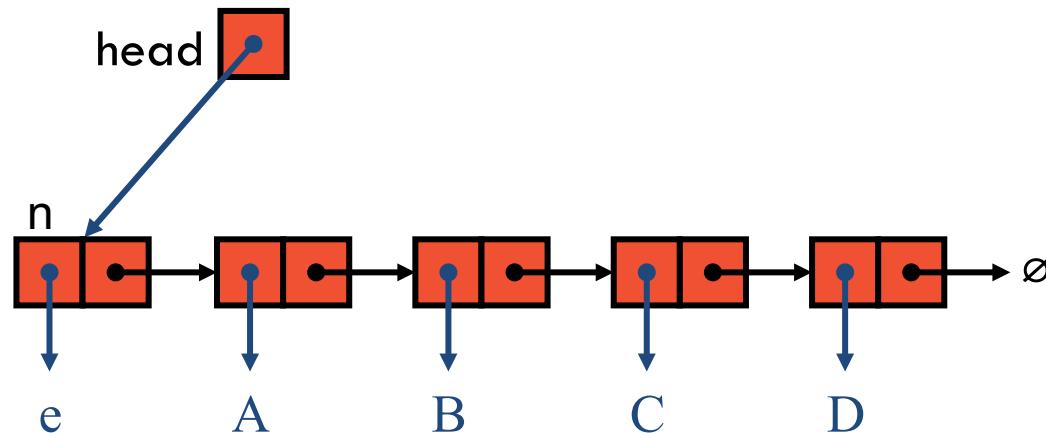
What is the time complexity?



insertFirst(e)

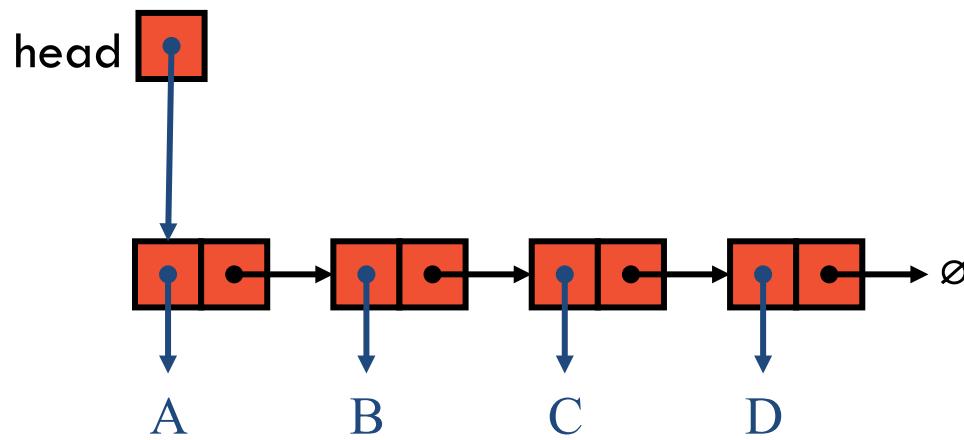
1. Instantiate a new node n
2. Set e as element of n
3. Set $n.next$ to point to (old) head
4. Update list's head to point to n

What is the time complexity? $O(1)$



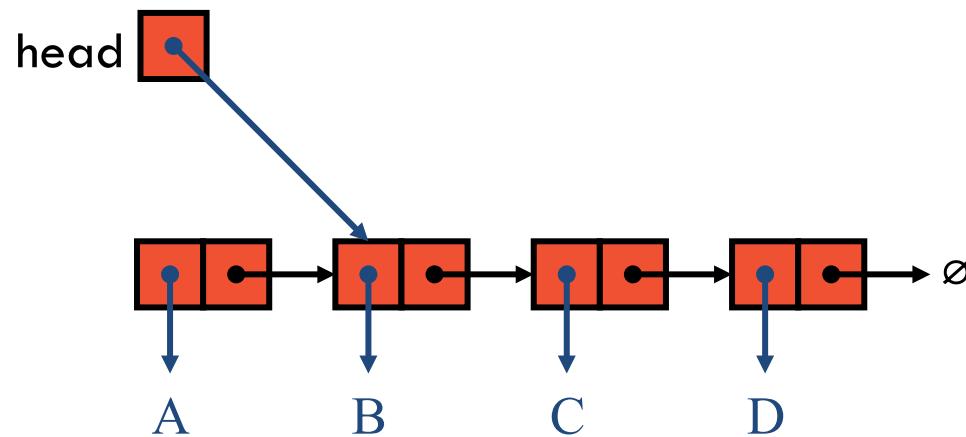
removeFirst()

1. Update head to point to next node
2. Delete the former first node



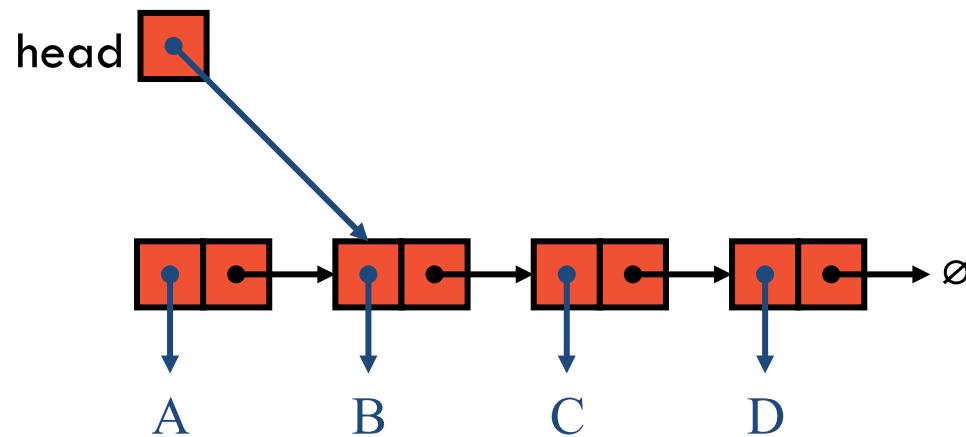
removeFirst()

1. Update head to point to next node
2. Delete the former first node



removeFirst()

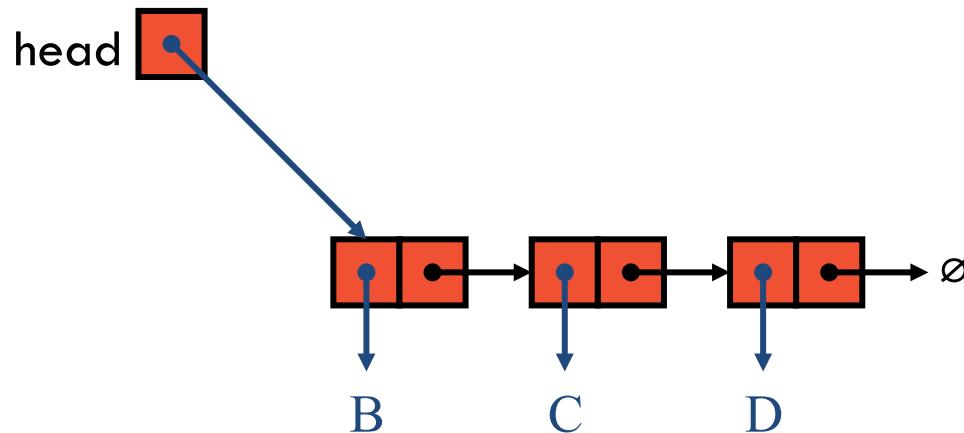
1. Update head to point to next node
2. Delete the former first node



removeFirst()

1. Update head to point to next node
2. Delete the former first node

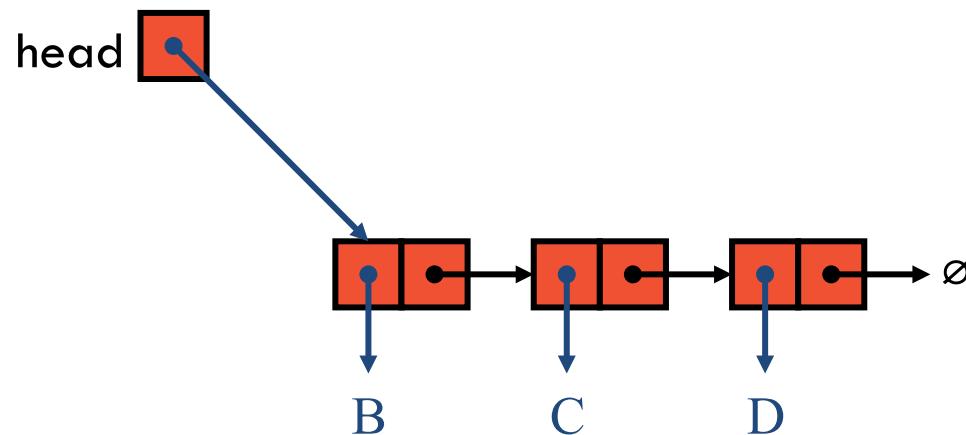
Time complexity?



removeFirst()

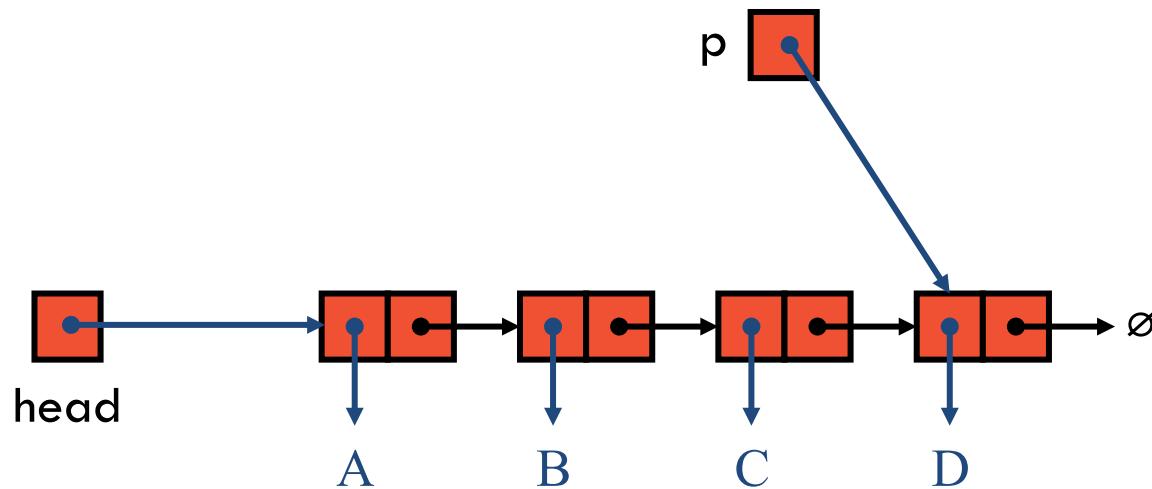
1. Update head to point to next node
2. Delete the former first node

Time complexity? $O(1)$



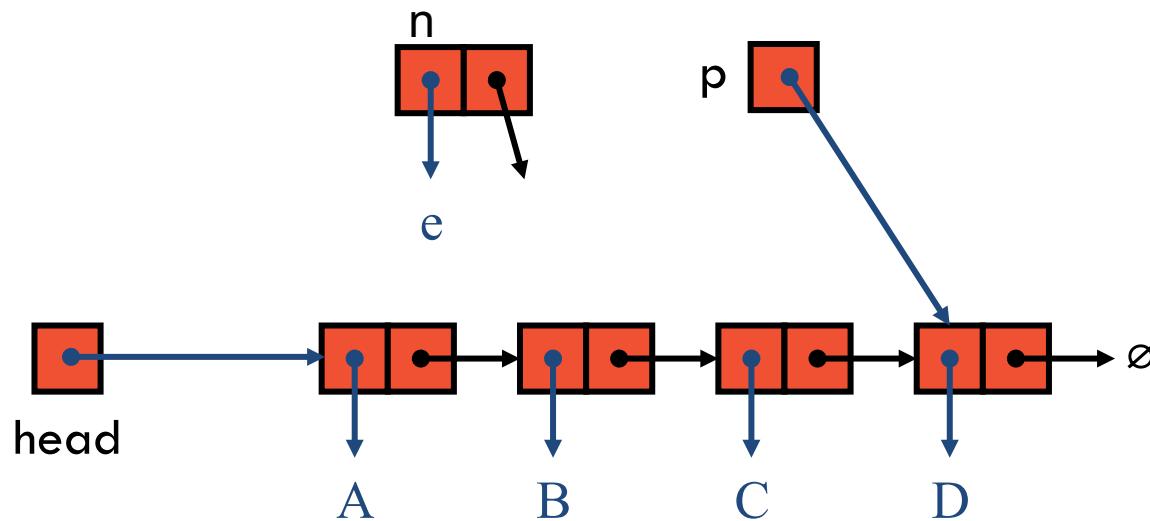
insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p



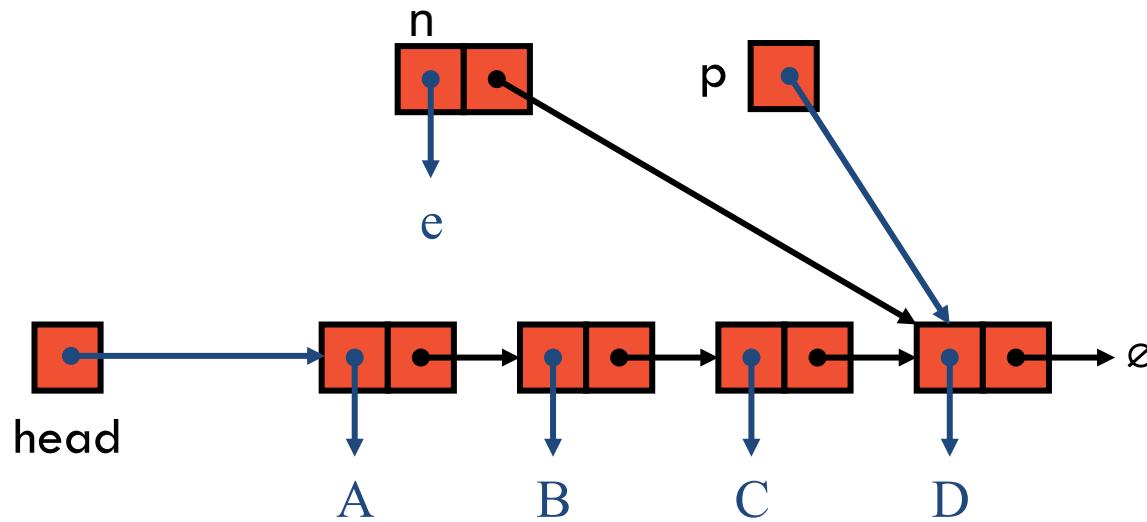
insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p



insertBefore(p,e)

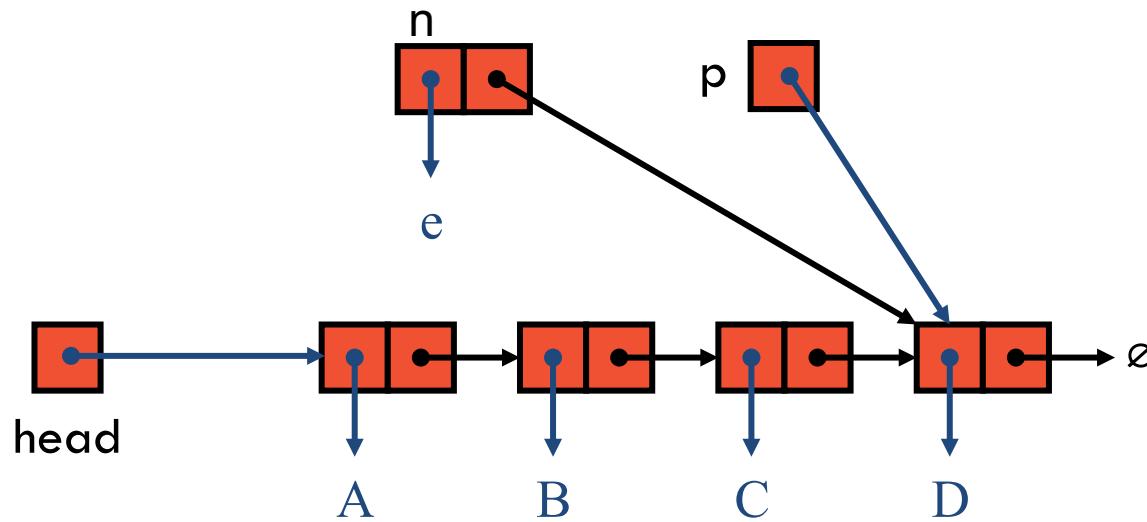
insertBefore(p,e) : insert e in front of the element at position p



What's the next step?

insertBefore(p,e)

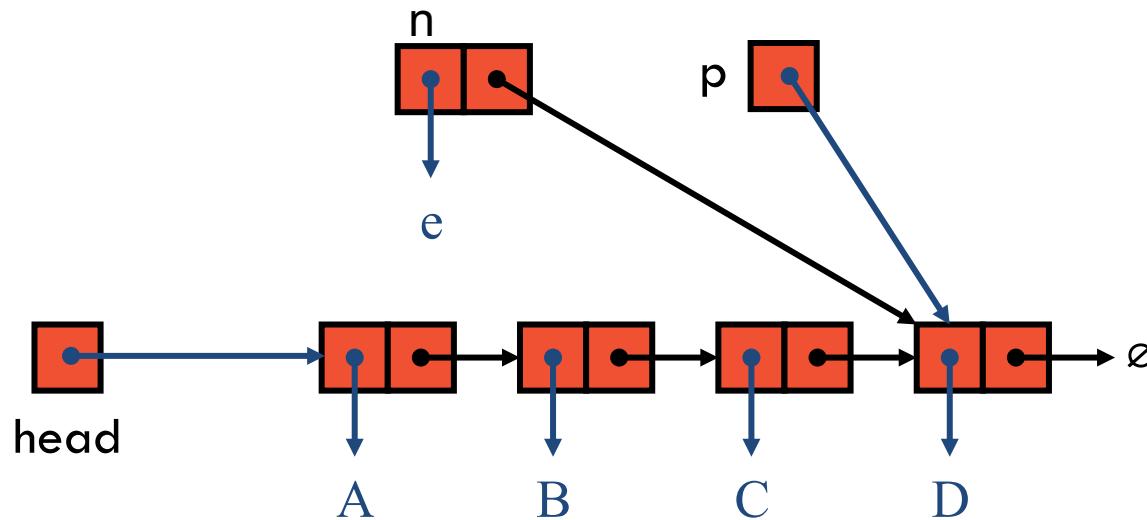
insertBefore(p,e) : insert e in front of the element at position p



What's the next step? Find the predecessor of n. How?

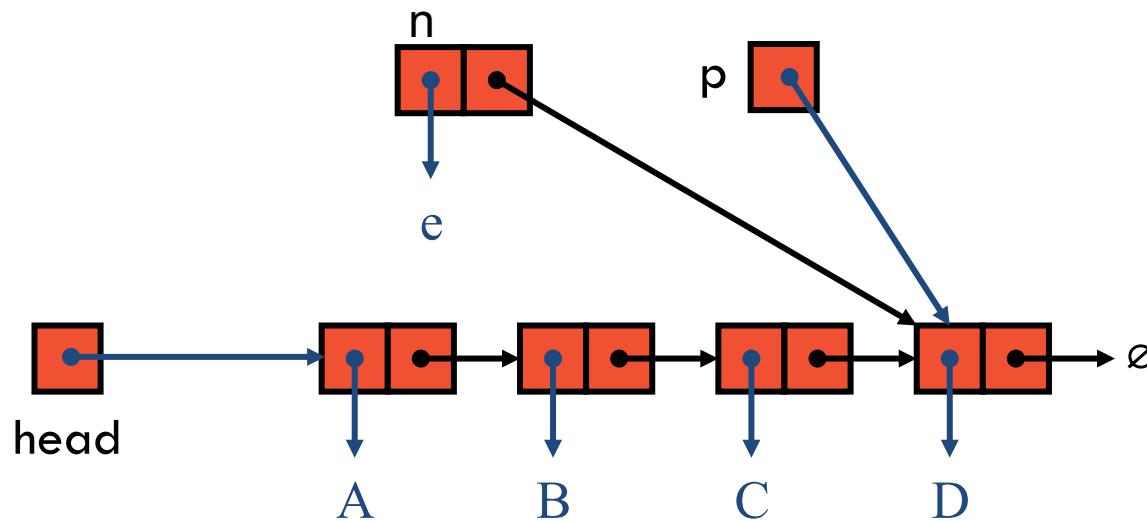
insertBefore(p,e)

To find the predecessor of p we need to follow the links from the “head”. Time complexity?



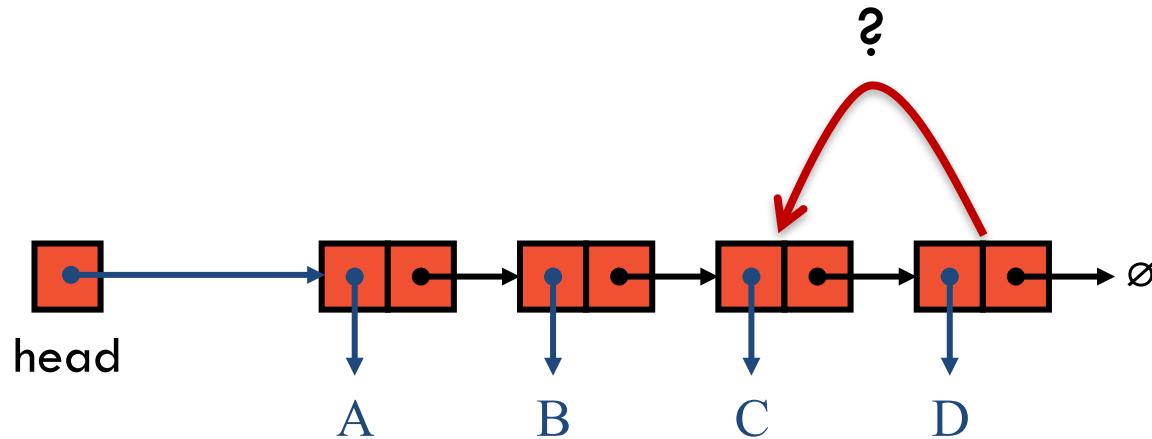
insertBefore(p,e)

To find the predecessor of p we need to follow the links from the “head”. Time complexity: $O(n)$



insertBefore(p,e)

There is no constant-time way to find the predecessor of a node in a Singly Linked List.

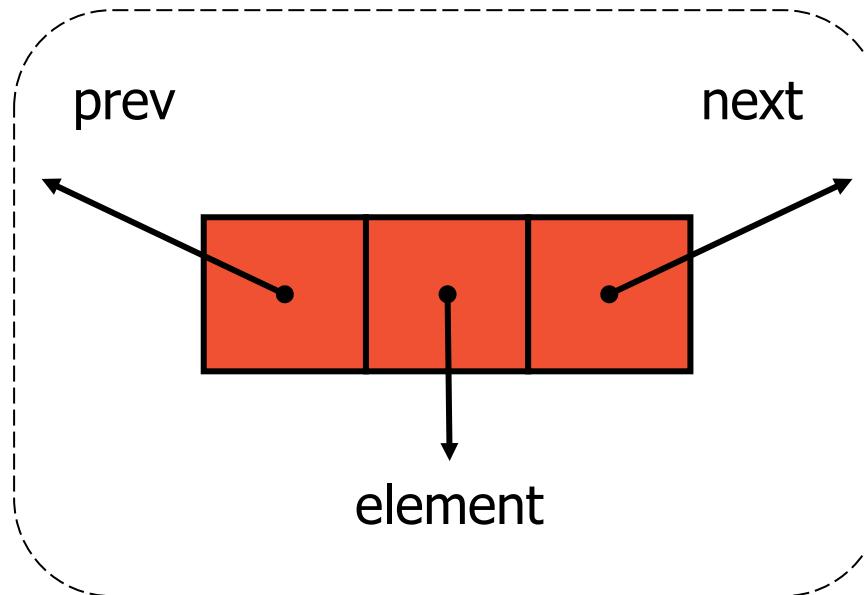


Another attempt

A very natural way to implement a positional list is with a doubly-linked list, so that it is easy/quick to find the position before.

Each Node in a Doubly Linked List stores

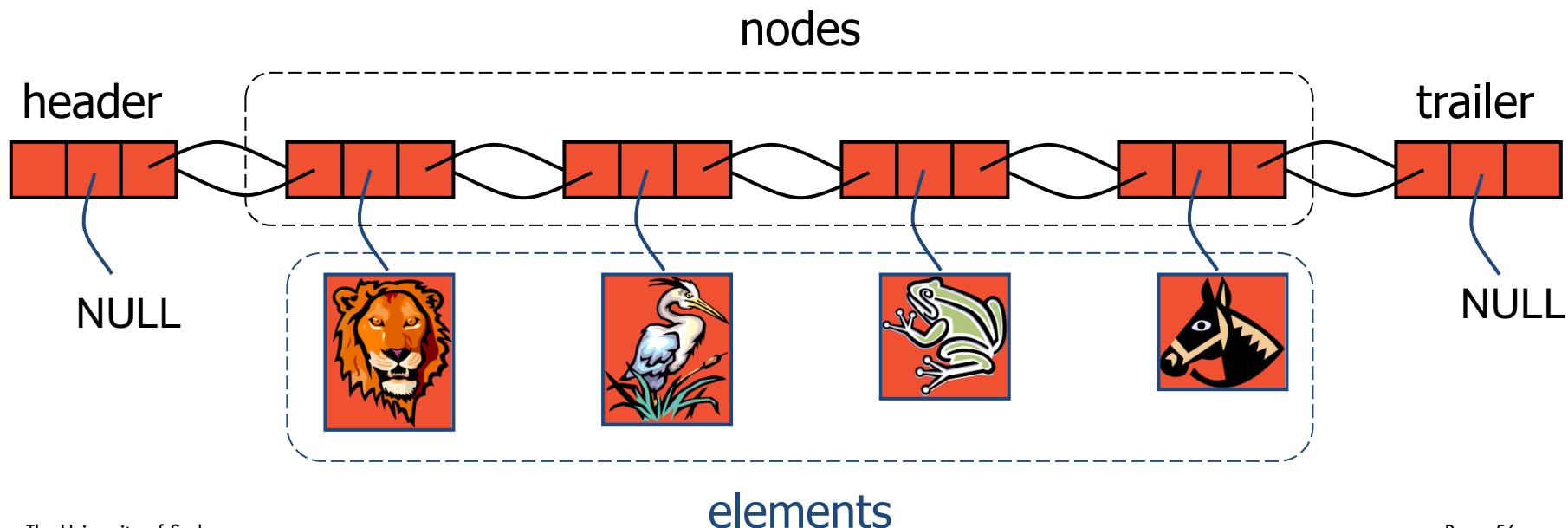
- its element, and
- a link to the previous and next nodes.



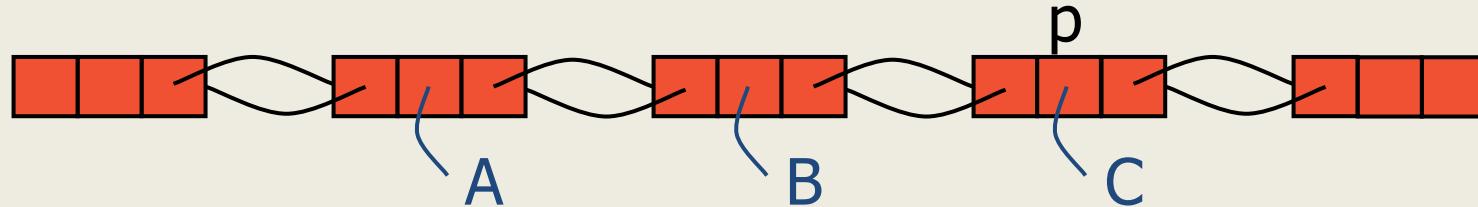
Doubly Linked Lists

A concrete data structure

- A sequence of Nodes, each with reference to prev and to next
- List captured by references to its **Sentinel Nodes**

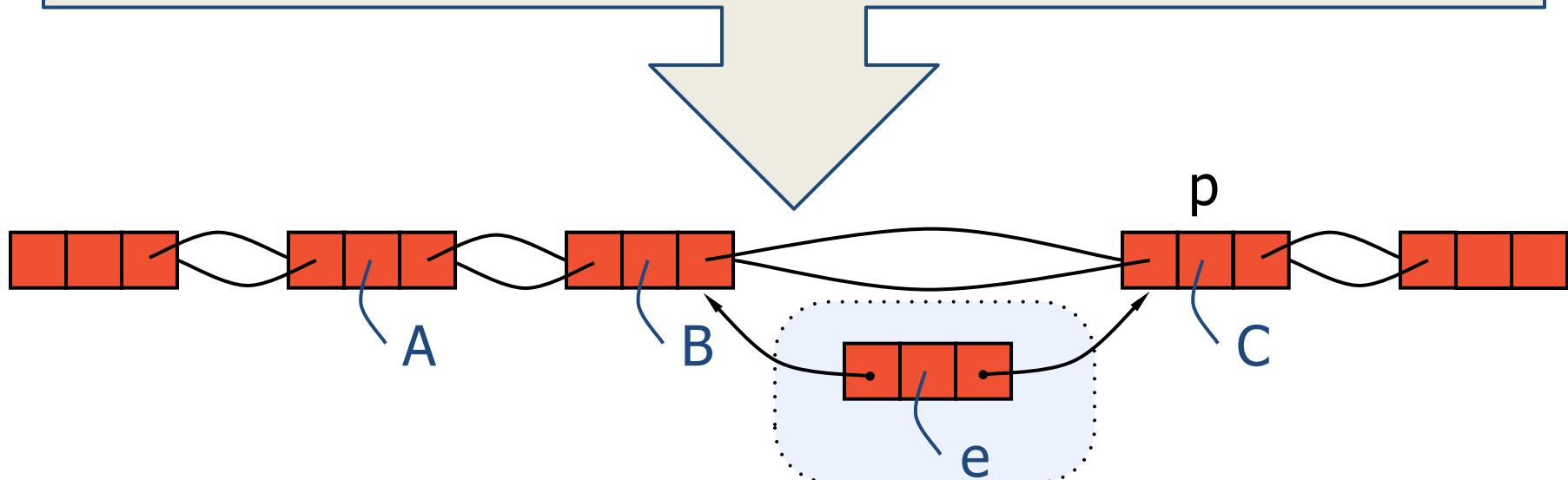


insertBefore(p,e) – step 1

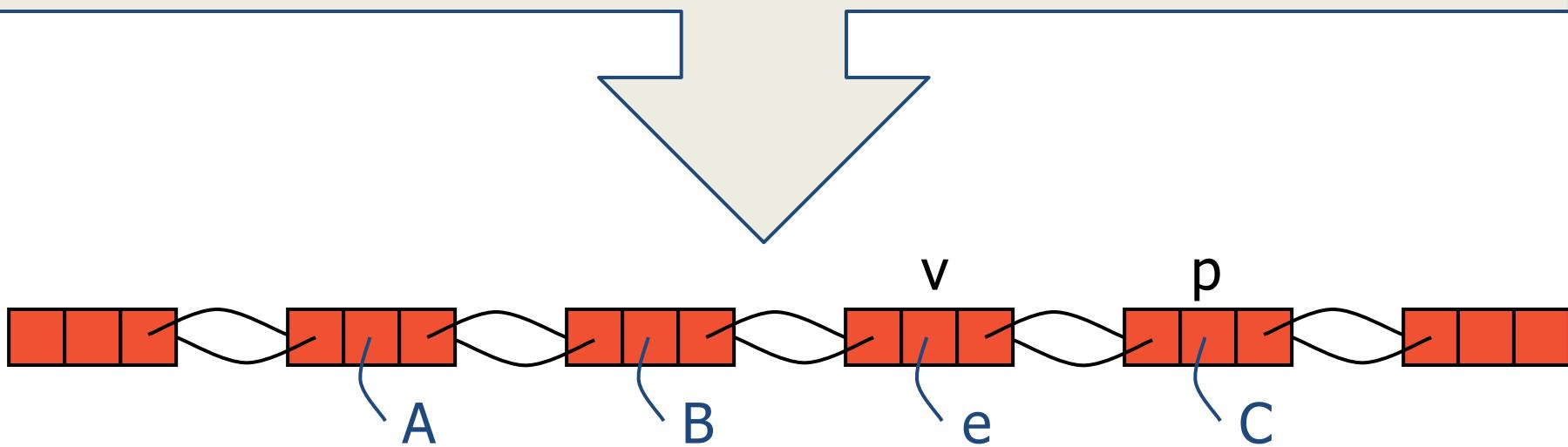
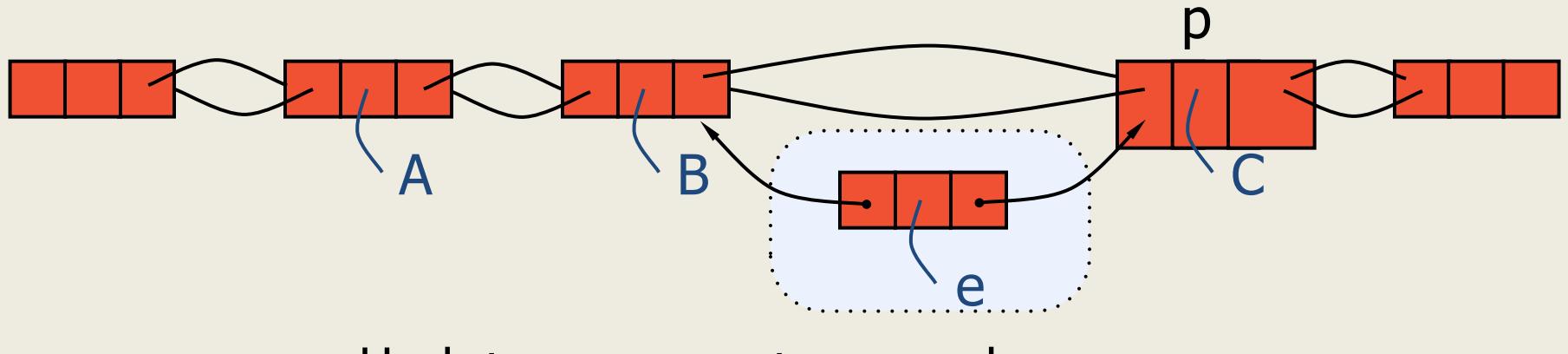


Instantiate new Node v with element set to e.

Update v.previous to point to p.previous and v.next to point to p.

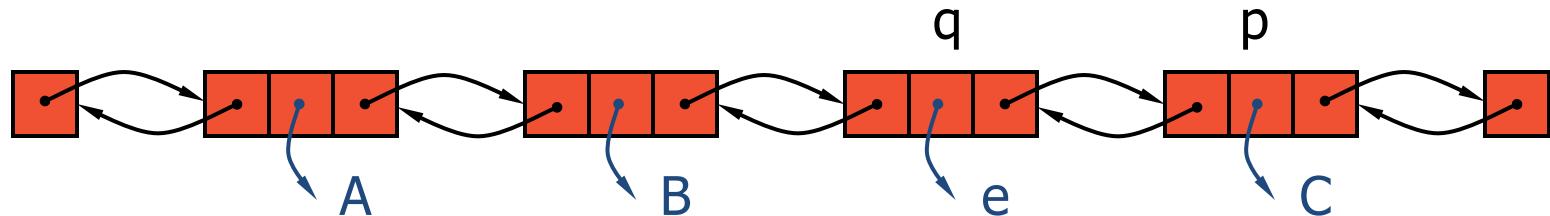
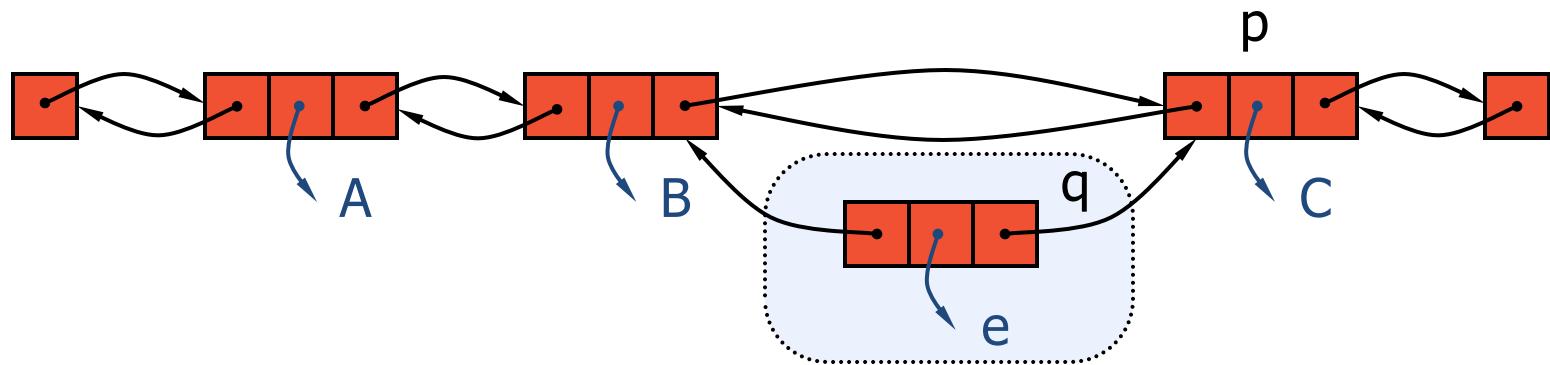
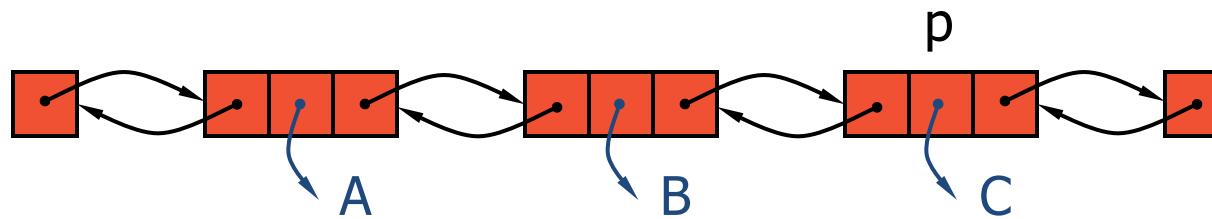


insertBefore(p,e) – step 2



insertBefore(p,e)

- Insert a new node with element e between p and its predecessor.



Pseudo-code

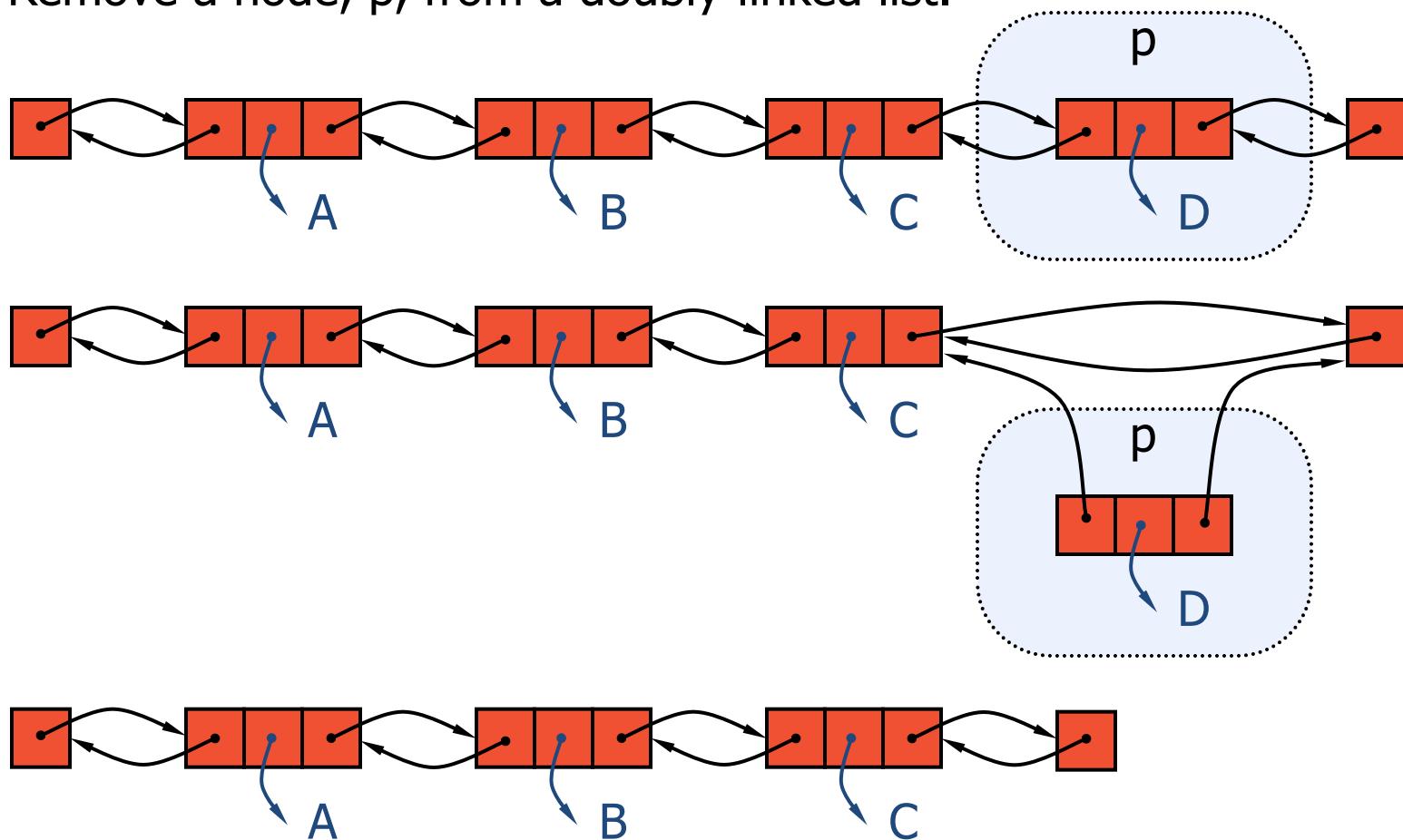
```
def insert_before(pos, elem):
    // insert elem before pos
    // assuming it is a legal pos

    new_node ← create a new node
    new_node.element ← elem
    new_node.prev ← pos.prev
    new_node.next ← pos
    pos.prev.next ← new_node
    pos.prev ← new_node

    return new_node
```

remove(p)

- Remove a node, p, from a doubly-linked list.



Pseudo-code

```
def remove(pos):
    // remove pos from the list
    // assuming it is a legal pos

    pos.prev.next ← pos.next
    pos.next.prev ← pos.prev

    return pos.element
```

Performance

A linked list can perform all of the accessor and update operations for a positional list in constant time.

Space complexity is $O(n)$

Time complexity is $O(1)$ for all operations

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insert_before(p, e)	$O(1)$
insert_after(p, e)	$O(1)$
remove(p)	$O(1)$
size()	$O(1)$
is_empty()	$O(1)$

Array or Linked List implementation?

Linked List

- good match to positional ADT
- efficient insertion and deletion
- simpler behaviour as collection grows
- modifications can be made as collection iterated over
- space not wasted by list not having maximum capacity

Arrays

- good match to index-based ADT
- caching makes traversal fast in practice
- no extra memory needed to store pointers
- allow random access (retrieve element by index)

Iterators

Abstracts the process of stepping through a collection of elements one at a time by extending the concept of position

Implemented by maintaining a cursor to the “current” element

Two notions of iterator:

- snapshot freezes the contents of the data structure
(unpredictable behaviour if we modify the collection)
- dynamic follows changes to the data structure
(behaviour changes predictably)

Iterators in Python

`iter(obj)` returns an iterator of the object collection

To make a class iterable define the method `__iter__(self)`

The method `__iter__()` returns an object having a `next()` method

Calling `next()` returns the next object and advances the cursor or raises `StopIteration()`

Iterators in Python

```
for x in collection:  
    [process x]
```

Is equivalent to:

```
it = x.__iter__()  
try:  
    while True:  
        x = it.next()  
        [process x]  
except StopIteration:  
    pass
```

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 2b: stacks and queues [GT 2.1]

Dr. Julian Mestre

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Stacks and queues

These ADTs are restricted forms of List, where insertions and removals happen only in particular locations:

- stacks follow last-in-first-out (LIFO)
- queues follows first-in-first-out (FIFO)

So why should we care about a less general ADT?

- operations names are part of computing culture
- numerous applications
- simpler/more efficient implementations than Lists

Stack ADT



Main stack operations:

- **push(e)**: inserts an element, e
- **pop()**: removes and returns the last inserted element

Auxiliary stack operations:

- **top()**: returns the last inserted element without removing it
- **size()**: returns the number of elements stored
- **isEmpty()**: indicates whether no elements are stored

Stack Example

operation	returns	stack
push(5)	-	[5]
push(3)	-	[5, 3]
size()	2	[5, 3]
pop()	3	[5]
isEmpty()	False	[5]
pop()	5	[]
isEmpty()	True	[]
push(7)	-	[7]
push(9)	-	[7, 9]
top()	9	[7, 9]
push(4)	-	[7, 9, 4]
pop()	4	[7, 9]

Stack Applications

Direct applications

- Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor
- Chain of method calls in a language supporting recursion
- Context-free grammars

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Method Stacks

The runtime environment keeps track of the chain of active methods with a stack, thus allowing **recursion**

When a method is called, the system pushes on the stack a frame containing

- Local variables and return value
- Program counter

When a method ends, we pop its frame and pass control to the method on top

```
main() {  
    int i = 5;  
    foo(i);  
}
```

```
foo(int j) {  
    int k;  
    k = j+1;  
    bar(k);  
}
```

```
bar(int m) {  
    ...  
}
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Parentheses Matching

Each "(", "{", or "[" must be paired with a matching ")" , "}" , or "]"

- correct: ()(()){([()])}
- correct: ((()(())){([()])}))
- incorrect:)(()){([()])}
- incorrect: {[])}
- incorrect: (

Scan input string from left to right:

- If we see an opening character, push it to a stack
- If we see a closing character, pop character on stack and check that they match

Evaluating Arithmetic Expressions

{not examinable}

$$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$$

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right

Example: $(x - y) + z$ rather than $x - (y + z)$

Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations.

Evaluating Arithmetic Expressions

{not examinable}

Slide by Matt Stallmann
included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use \$ as special “end of input” token with lowest precedence

Algorithm `doOp()`

```
x ← valStk.pop();  
y ← valStk.pop();  
op ← opStk.pop();  
valStk.push( y op x )
```

Algorithm `repeatOps(refOp)`:

```
while ( valStk.size() > 1 ∧  
prec(refOp) ≤ prec(opStk.top())  
doOp()
```

Algorithm `EvalExp()`

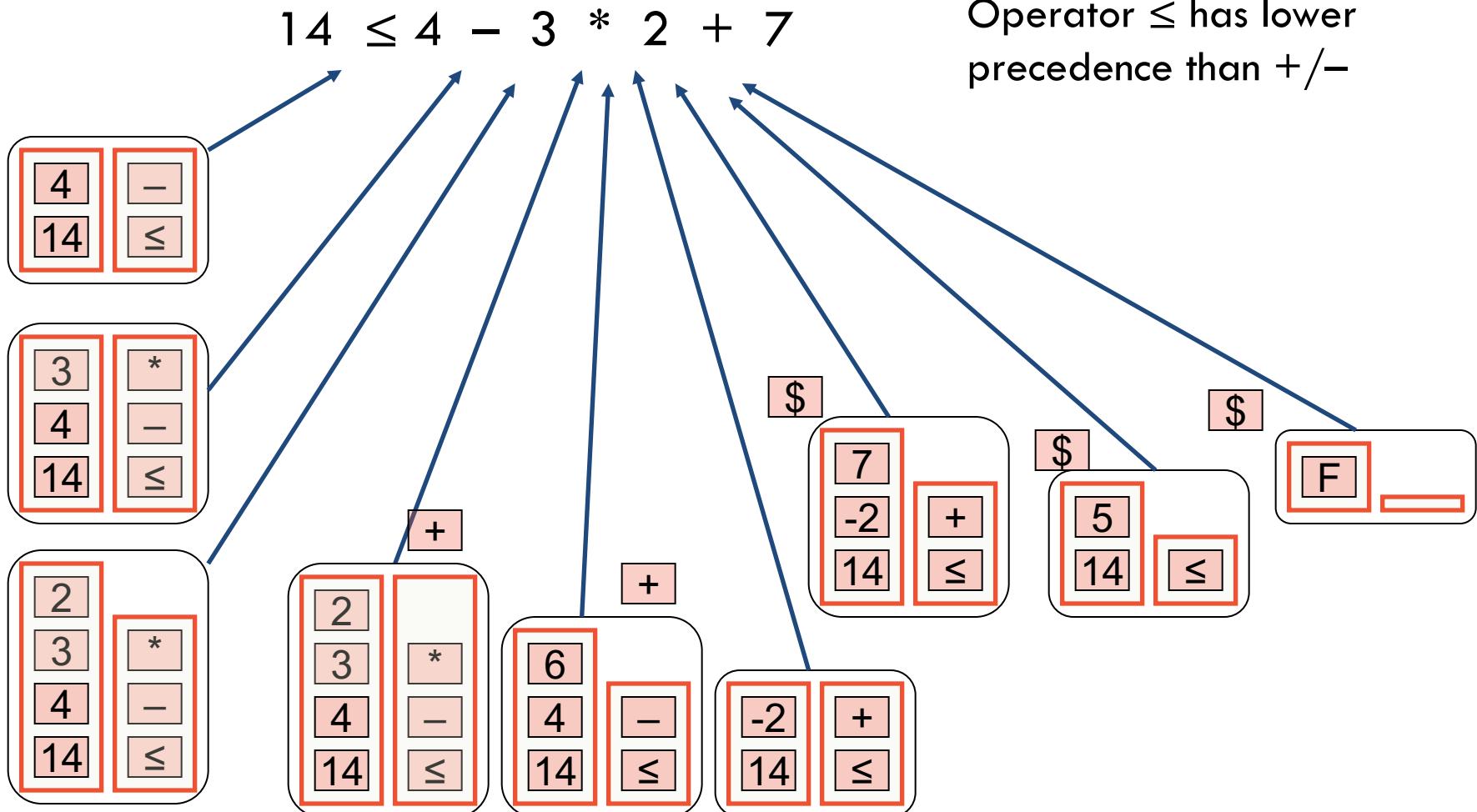
Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z  
if isNumber(z) then  
    valStk.push(z)  
else  
    repeatOps(z);  
    opStk.push(z)  
repeatOps($);  
return valStk.top()
```

Evaluating Arithmetic Expressions {not examinable}

Slide by Matt Stallmann
included with permission.



Stack implementation based on arrays

A simple way of implementing the Stack ADT uses an array:

- Array has capacity N
- Add elements from left to right
- A variable t keeps track of the index of the top element



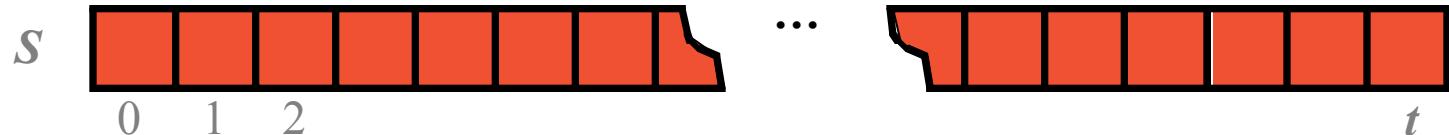
```
def size()
    return t + 1

def pop()
    if isEmpty()
        return null
    else
        t = t - 1
    return S[t + 1]
```

Stack implementation based on arrays

- The array storing the stack elements may become full
- A push operation will then either grow the array or signal a “stack overflow” error.

```
def push(e)
    if t == N - 1 then
        return "stack overflow"
    else
        t = t + 1
        S[t] = e
```



Stack implementation based on arrays

Performance

- The space used is $O(N)$
- Each operation runs in time $O(1)$

Qualifications

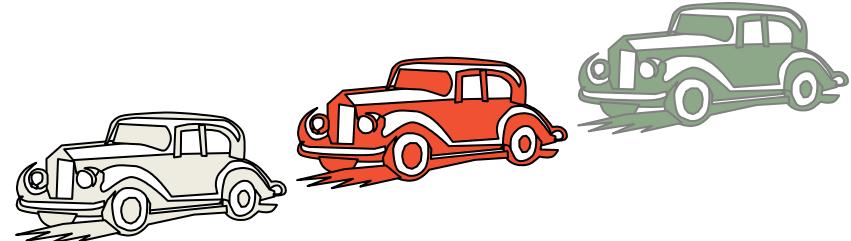
- Trying to push a new element into a full stack causes an implementation-specific exception or
- Pushing an item on a full stack causes the underlying array to double in size to make room for new element

What about a Linked List based Stack?

- No arbitrary capacity limit
- Top of stack is front of list: insertion and deletion in constant time

Stack method	Singly Linked list method
size() isEmpty() push(e) pop() top()	list.size() list.isEmpty() list.addFirst(e) list.removeFirst() list.first ()

Queue ADT



Main queue operations:

- **enqueue(e)**: inserts an element, e, at the end of the queue
- **dequeue()**: removes and returns element at the front of the queue

Auxiliary queue operations:

- **first()**: returns the element at the front without removing it
- **size()**: returns the number of elements stored
- **isEmpty()**: indicates whether no elements are stored

Boundary cases:

- Attempting the execution of dequeue or first on an empty queue signals an error or returns null

Queue Example

Operation		Output	Q
enqueue(5)	—	(5)	
enqueue(3)	—	(5, 3)	
dequeue()	5	(3)	
enqueue(7)	—	(3, 7)	
dequeue()	3	(7)	
first()	7	(7)	
dequeue()	7	()	
dequeue()	<i>null</i>	()	
isEmpty()	<i>true</i>	()	
enqueue(9)	—	(9)	
enqueue(7)	—	(9, 7)	
size()	2	(9, 7)	
enqueue(3)	—	(9, 7, 3)	
enqueue(5)	—	(9, 7, 3, 5)	
dequeue()	9	(7, 3, 5)	

Queue applications

Buffering packets in streams, e.g., video or audio

Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

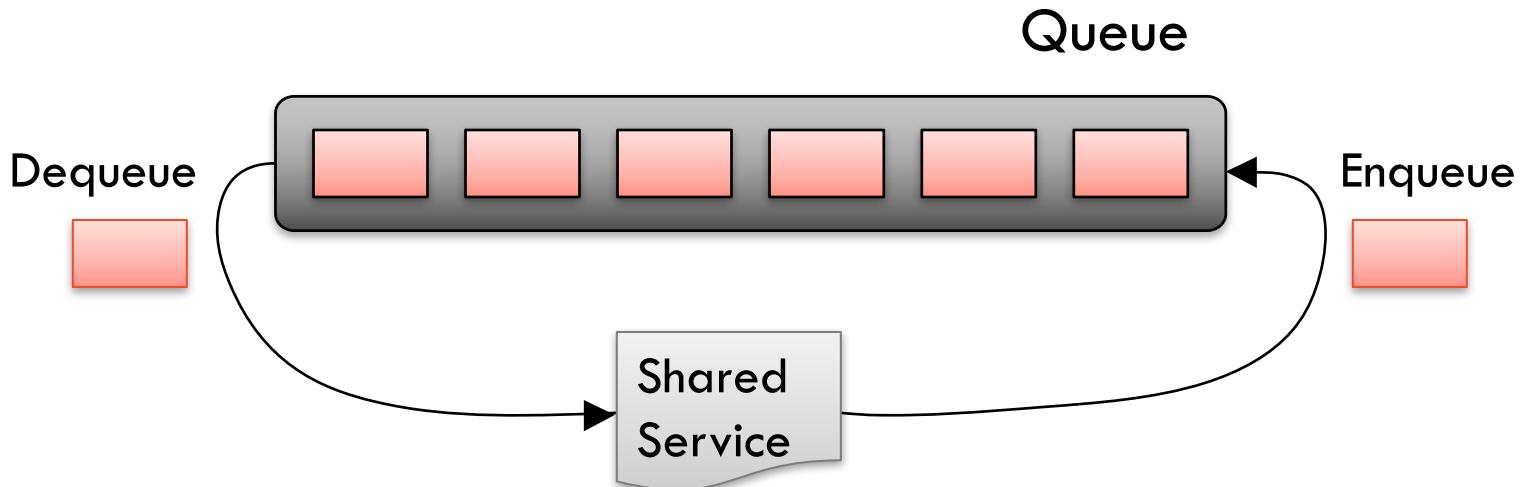
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Queue application: Round Robin Schedulers

Implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

1. `e = Q.dequeue()`
2. Service element e
3. `Q.enqueue(e)`



Queue implementation based on arrays

Use an array of size **N** in a circular fashion

Two variables keep track of the front and size

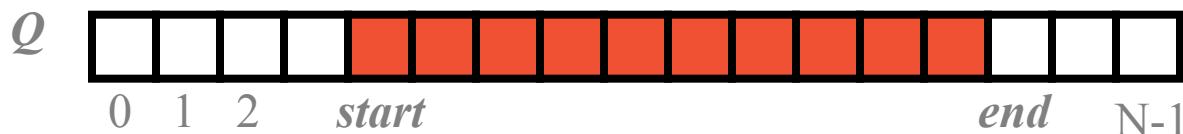
start : index of the front element

end : index past the last element

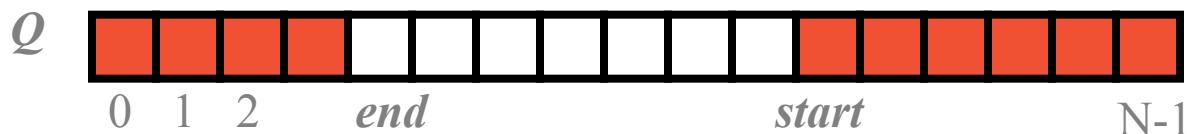
size : number of stored elements

These are related as follows $\text{end} = (\text{start} + \text{size}) \bmod N$, so we only need two, `start` and `size`

normal configuration



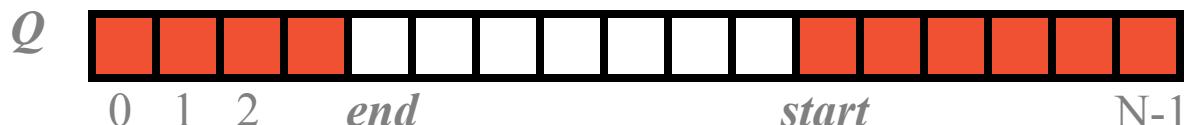
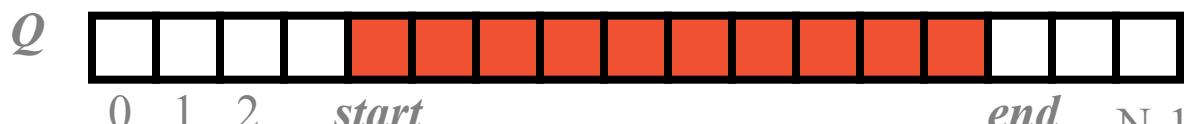
wrapped-around configuration



Queue Operations

- We use the modulo operator (remainder of division)

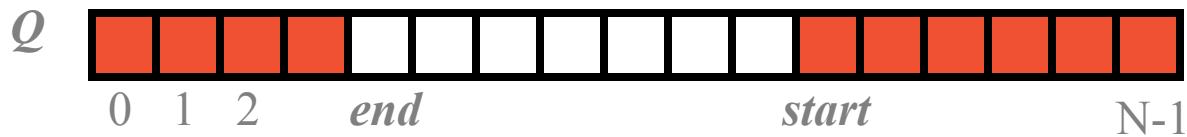
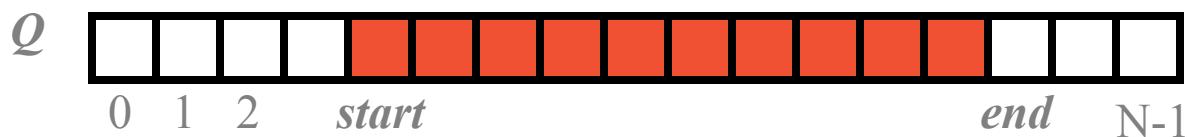
```
def size()  
    return s  
  
def isEmpty()  
    return (s == 0)
```



Queue Operations: Enqueue

Return an error if the array is full. Alternatively, we could grow the underlying array as dynamic arrays do

```
def enqueue(e)
    if size == N then
        return "queue full"
    else
        last = (first + size) mod N
        Q[last] = e
        size = size + 1
```

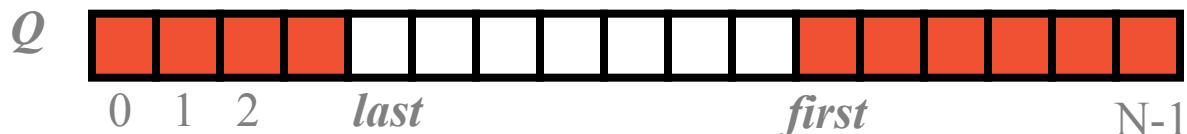
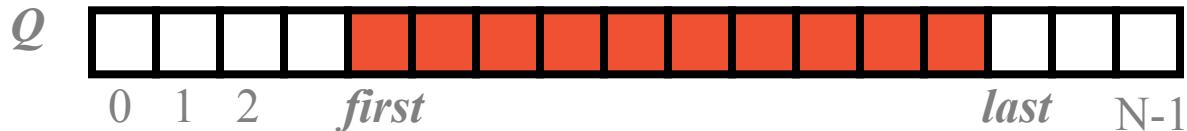


Queue Operations: Dequeue

Note that operation dequeue returns error if the queue is empty

One could alternatively signal an error

```
def dequeue()
    if isEmpty() then
        return "queue empty"
    else
        e = Q[first]
        first = (first + 1) mod N
        size = (size - 1)
        return e
```



Linked list Queue implementation

Linked-list implementation of the ADT Queue

- Overcomes size limitations
- But:
 - size per element slightly bigger
 - more complicated, extra care needed for cases where queue is empty before enqueue and after dequeue
 - need to keep pointers to both the first and the last nodes in the list if not then additions will not be $O(1)$

Double-ended queues: Deques

- A linear structure that allows insertions and deletions at both ends

Method	Time
size, isEmpty	$O(1)$
getFirst, getLast	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Table 5.4: Performance of a deque realized by a doubly linked list.

Double-ended queue operations

The deque abstract data type is richer than both the stack and the queue ADTs. The fundamental methods of the deque ADT are as follows:

`addFirst(e)`: Insert a new element *e* at the head of the deque.

`addLast(e)`: Insert a new element *e* at the tail of the deque.

`removeFirst()`: Remove and return the first element of the deque; an error occurs if the deque is empty.

`removeLast()`: Remove and return the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque ADT may also include the following support methods:

`getFirst()`: Return the first element of the deque; an error occurs if the deque is empty.

`getLast()`: Return the last element of the deque; an error occurs if the deque is empty.

`size()`: Return the number of elements of the deque.

`isEmpty()`: Determine if the deque is empty.

Stack implementation based on dynamic arrays

```
def push(e)
    # if we run out of space, double size of S
    if t == N - 1 then
        aux = new array[2*N]
        for j in [0:N] do
            aux[j] = S[j]
        S = aux
        N = 2*N
    t = t + 1
    S[t] = e
```

Amortized analysis

Back to the array-based implementation of lists. Every time we run out of space we double the underlying array.

Recall that worst-case analysis of push takes $O(n)$ time since we may need to double the size of underlying array

Let $T(i)$ be the complexity of i th operation

$$T(i) = \begin{cases} O(i) & \text{if } i = 2^k \\ O(1) & \text{if } i \neq 2^k \end{cases}$$

Amortized analysis

Starting from the empty list, the total complexity is linear:

Let $T(i)$ be the complexity of i -th operation

$$\begin{aligned}\sum_{i < n} T(i) &= \sum_{i \neq 2^k} O(1) + \sum_{k < \log n} O(2^k) \\ &= O(n) + O(2^{\log n}) \\ &= O(n)\end{aligned}$$

So even though a single operation can take $O(n)$, we can amortize (average) that cost among n operations.

Amortized analysis definition

We say that a sequence of n operation has $\mathcal{O}(f(n))$ amortized time complexity if in the worst-case the total amount of work done by the n operations is no more than $\mathcal{O}(n f(n))$

For the dynamic array implementation of stack. If we double the size of the array then append takes $\mathcal{O}(1)$ amortized time.

Improved space usage

Current solution is wasteful because if we pop too many items the underlying array may be much larger than the current size of the stack

What if we halve the size of S every time $t < N / 2$?

What if we halve the size of S every time $t < N / 4$?

This week

Tutorial Sheets 2: Available on Ed

Quiz 1: Available on Canvas

Assignment 1: Available on Ed and Gradescope

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 3: Trees

[GT 2.3]

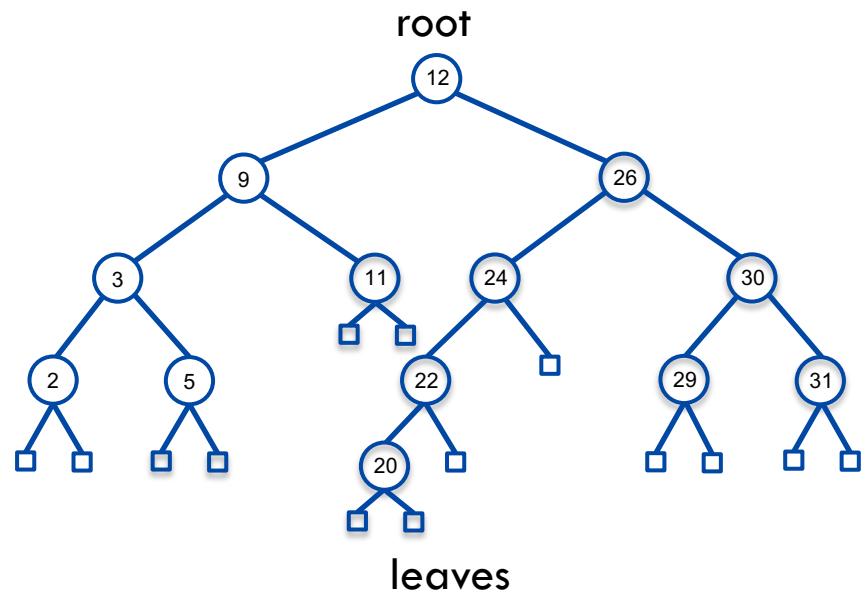
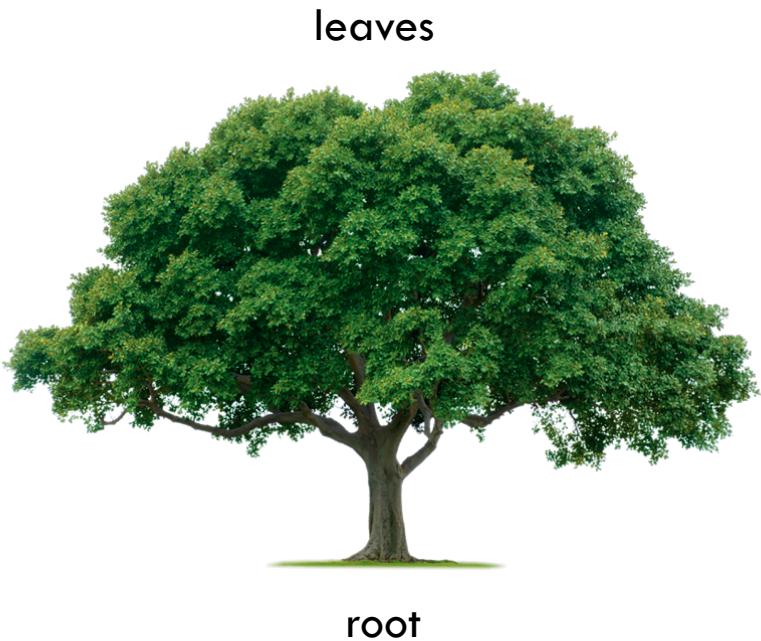
Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*

Agenda: Trees

- Definition and terminology
- Applications
- Tree ADT
- Tree traversal algorithms
- Binary trees
- Implementing trees
- Recursive code on trees
- Binary search trees
- B-trees

Trees

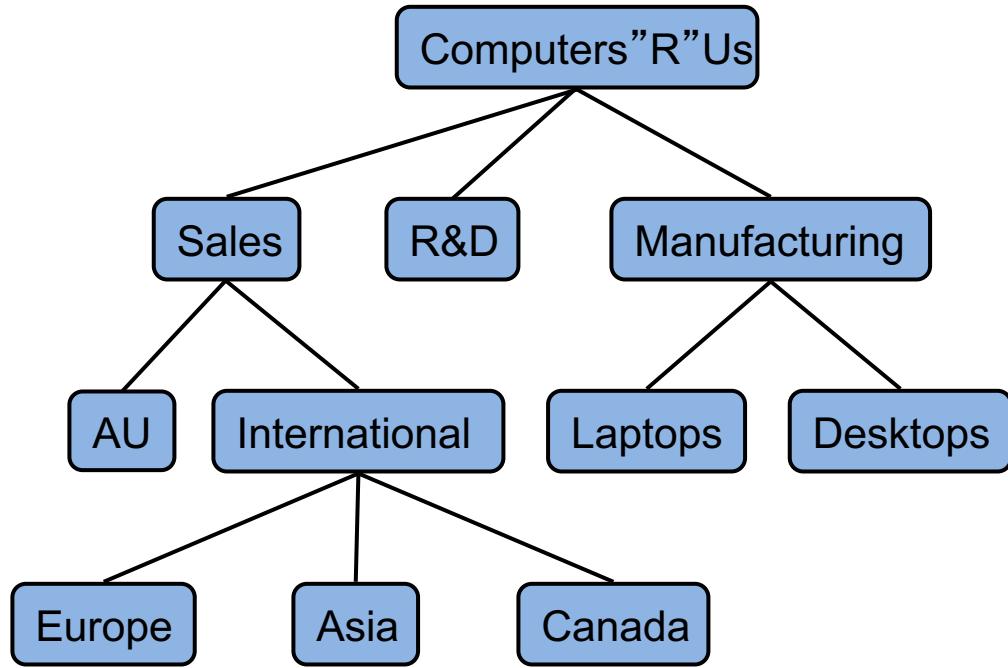


What is a Tree

In computer science, a tree is an abstract model of a hierarchical structure

A tree consists of nodes with a parent-child relation

- if u is parent of v , then v is a child of u
- a node has at most **one** parent in a tree
- a node can have zero, one or more children



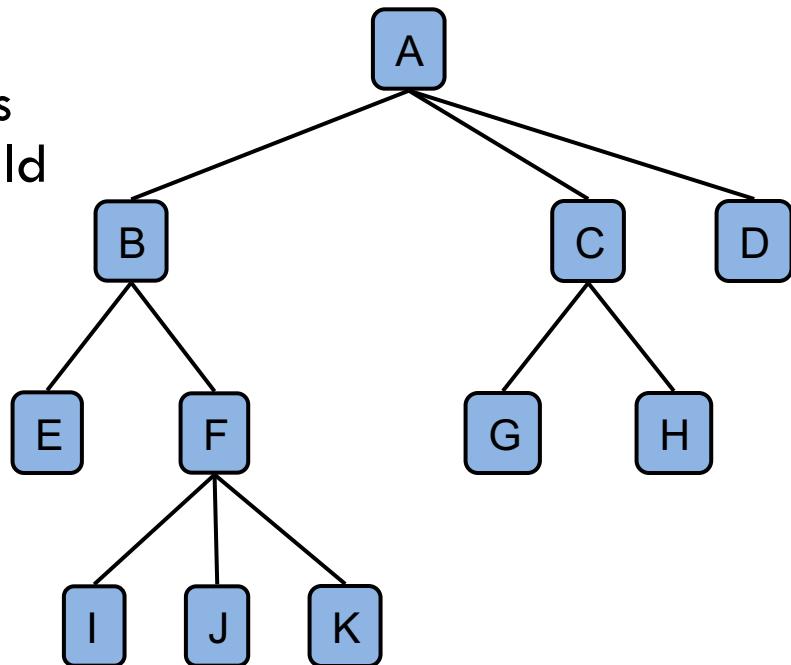
Applications:

- Organization charts
- File systems
- Phrase structure

Formal definition

A **tree T** is made up of a set of **nodes** endowed with **parent-child** relationship with following properties:

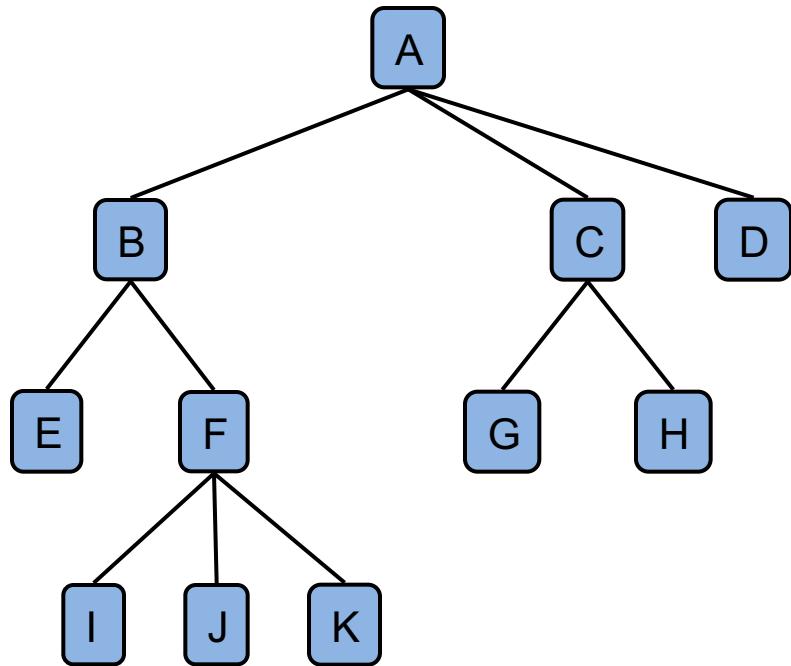
- If **T** is non-empty, it has a special node called the **root** that has no parent
- Every node **v** of **T** other than the root has a unique **parent**
- Following the parent relation always leads to the root (i.e., the parent-child relation does not have “cycles”)



Tree Terminology

Depending on where they are in the tree, we classify nodes into:

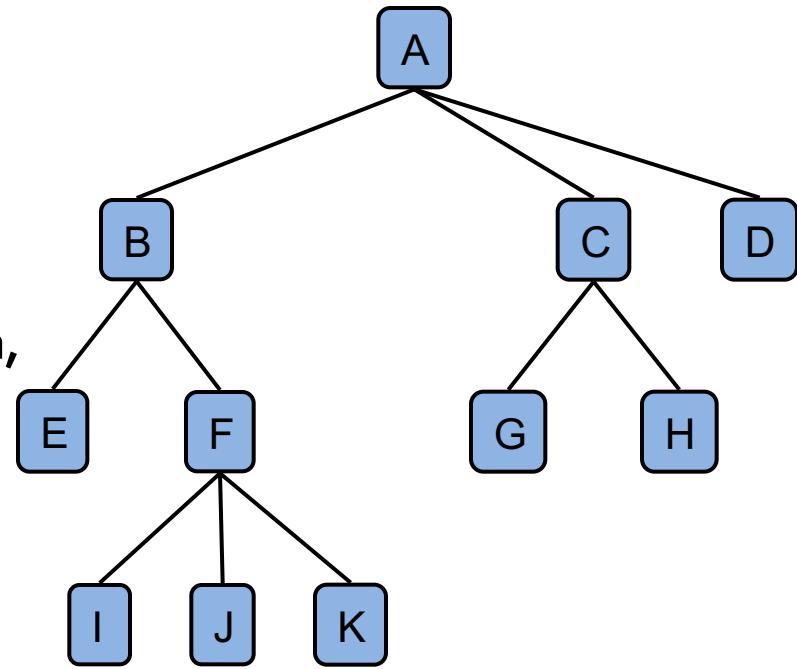
- **Root:** node without parent (e.g., A)
- **Internal node:** node with at least one child (e.g., A, B, C, F)
- **External/leaf node:** node without children (e.g., E, I, J, K, G, H, D)



Tree Terminology

We can extend the parent-child relation to capture indirect relations:

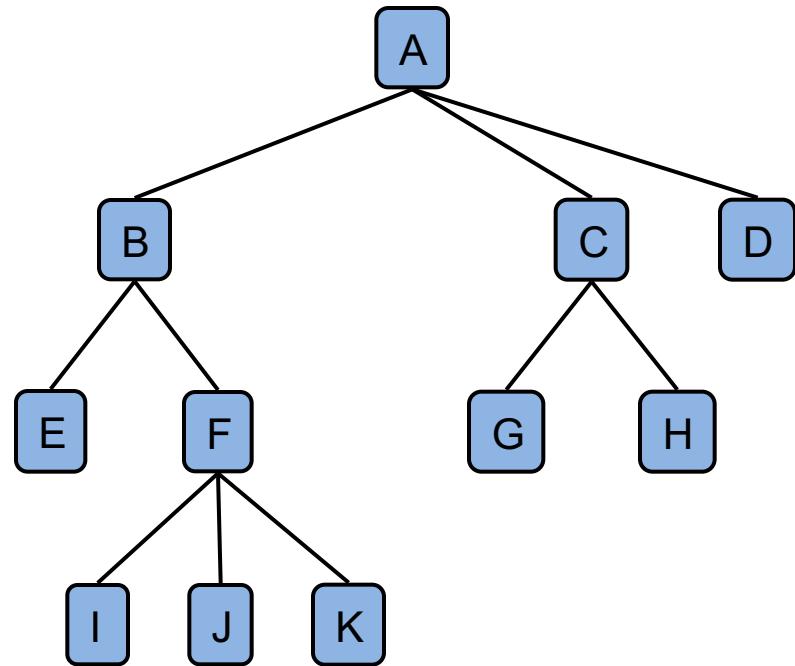
- **Ancestors:** parent, grandparent, great-grandparent, etc. (e.g., ancestors of F are A, B)
- **Descendants:** children, grandchildren, great-grandchildren, etc. (e.g., descendants of B are E, F, I, J, K)
- Two nodes with the same parent are **siblings** (e.g., B and D)



Tree Terminology

More fine-grained location concepts:

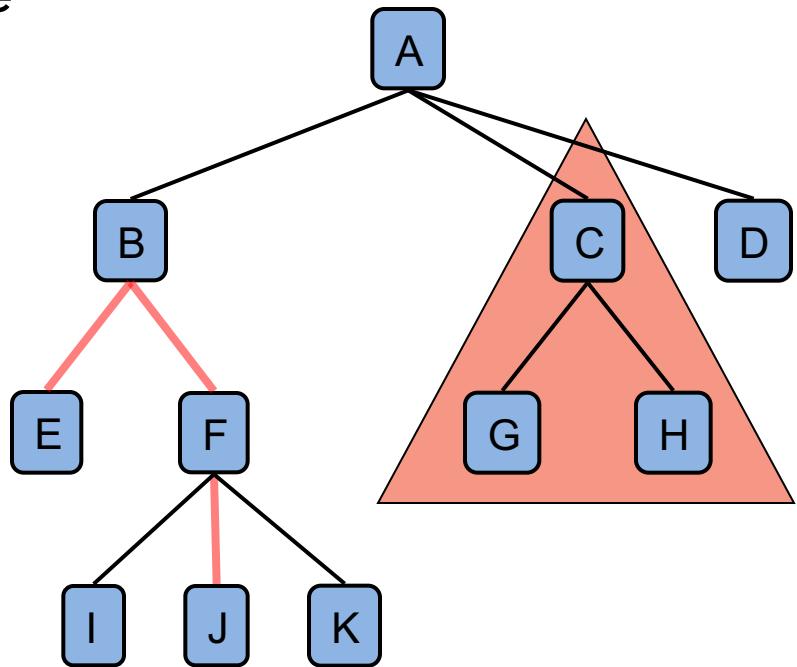
- **Depth of a node:** number of ancestors not including itself
(e.g., $\text{depth}(F) = 2$)
- **Level:** set of nodes with given depth
(e.g., $\{E, F, G, H\}$ are level 2)
- **Height of a tree:** maximum depth
(e.g., 3)



Tree Terminology

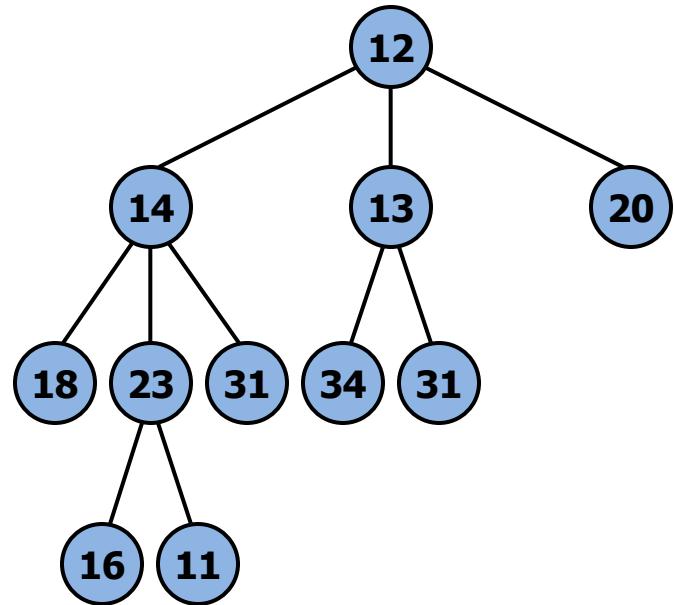
Substructures of a tree:

- **Subtree:** tree made up of some node and its descendants. (e.g., subtree rooted at C is {C, G, H})
- **Edge:** pair of nodes (u, v) such that one is the parent of the other
- **Path:** sequence of nodes such that 2 consecutive nodes in the sequence have an edge (e.g., $\langle E, B, F, J \rangle$).



Examples

- Node 14 has depth ... 1
- The tree has height ... 3
- Subtree rooted at node 14 has height ... 2
- Any subtree of a leaf has height ... 0
- The root has depth ... 0



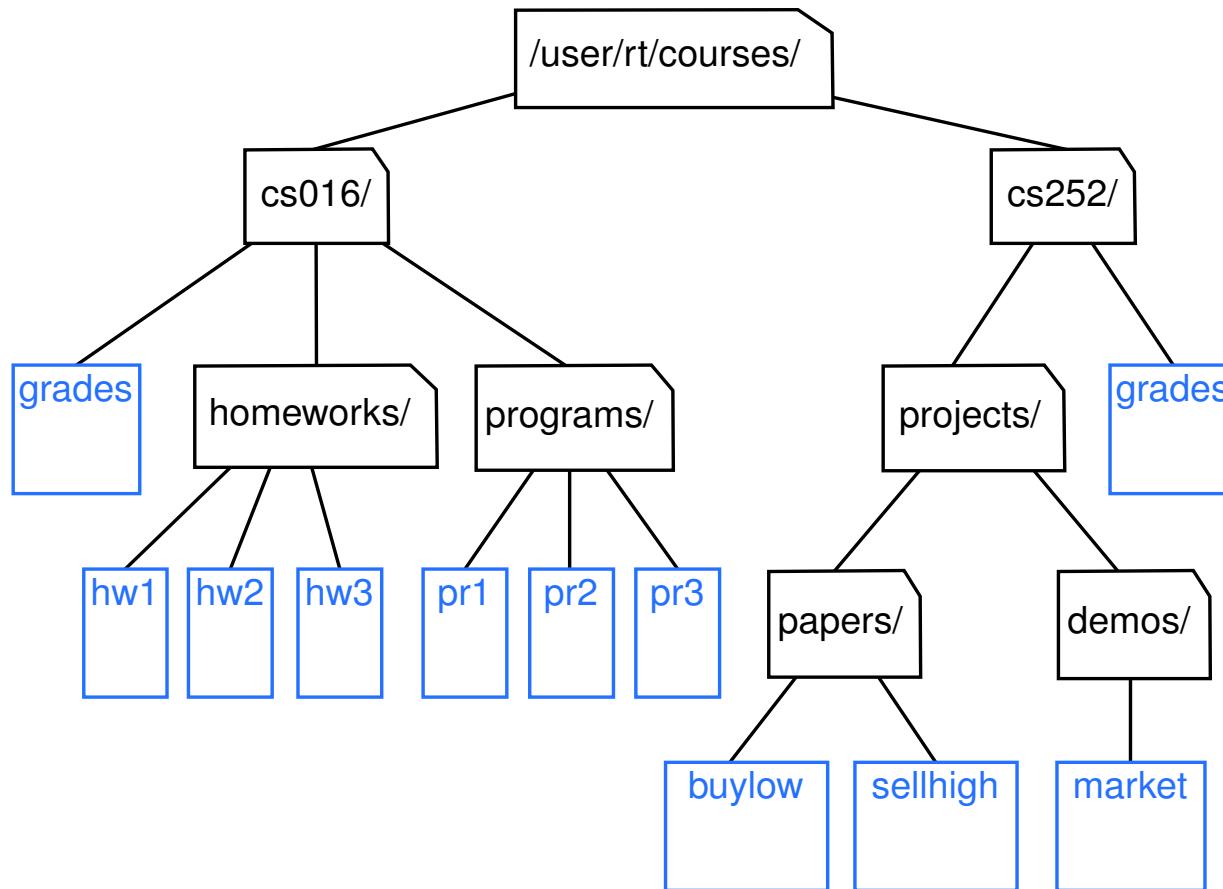
Ordered Trees

Sometimes order of siblings matter

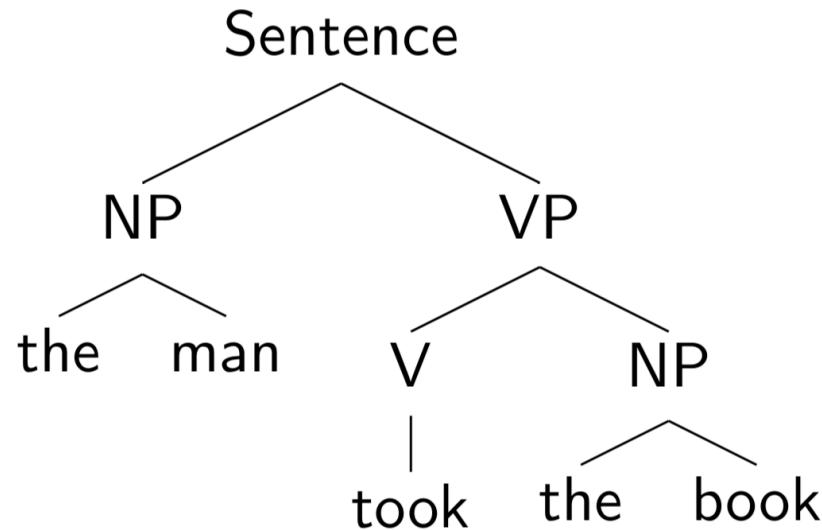
In an **ordered tree** there is a prescribed order for each node's children

In a diagram this ordering is usually represented by the left to right arrangement of the nodes

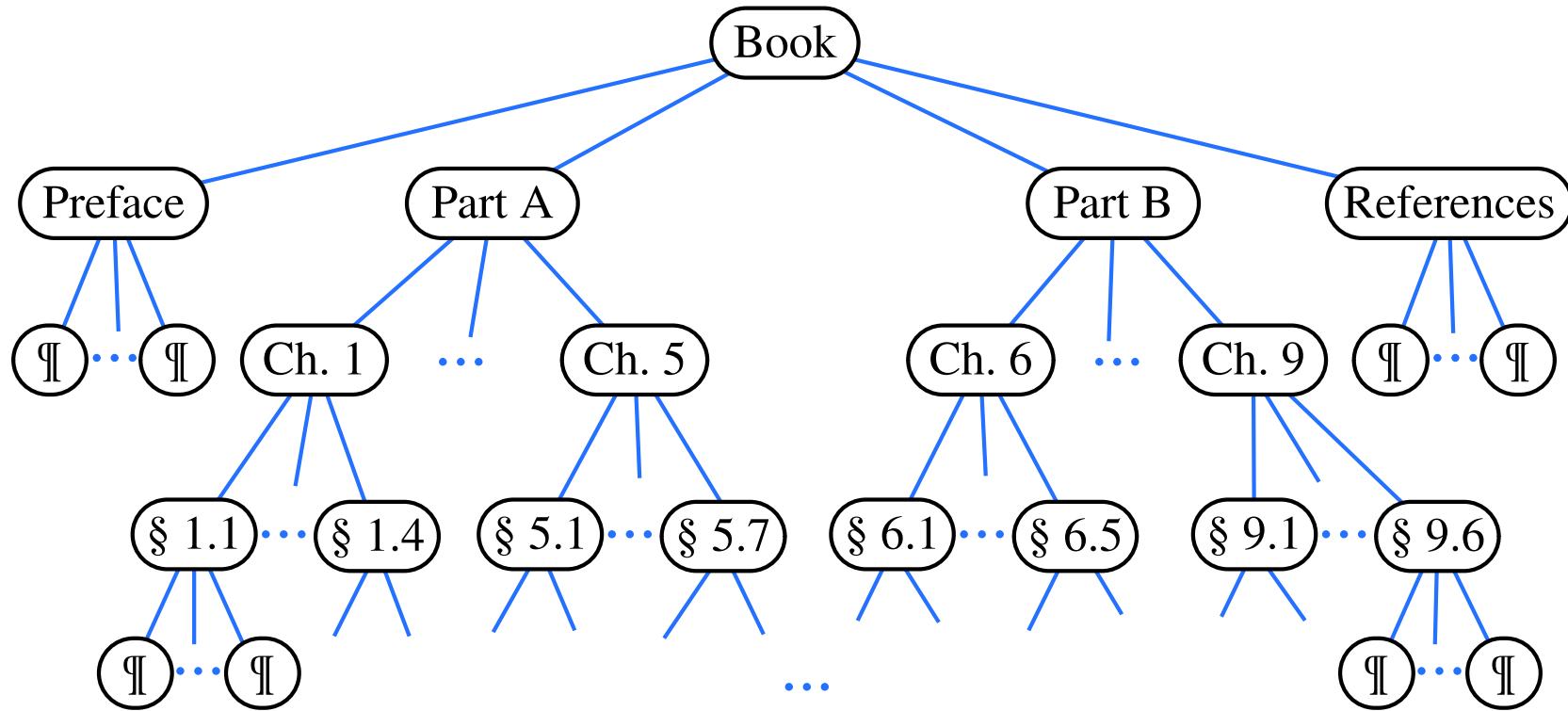
Application: OS file structure



Application: Phrase structure tree



Application: Document structure



Tree ADT

- Position as Node abstraction
 - Generic methods:
 - integer `size()`
 - boolean `isEmpty()`
 - Iterator `iterator()`
 - Iterable `positions()`
 - Access methods:
 - Position `root()`
 - Position `parent(p)`
 - Iterable `children(p)`
 - Integer `numChildren(p)`
- ▶ Query methods:
- ▶ boolean `isInternal(p)`
 - ▶ boolean `isExternal(p)`
 - ▶ boolean `isRoot(p)`
- ▶ Additional update methods may be defined by data structures implementing the Tree ADT

Node object

Node object implementation typically has the following attributes:

- value: the value associated with this Node
- children: set or list of children of this Node
- parent: (optional) the parent of this Node.

```
def is_internal(p)
    # test if p is internal
    return not p.children.is_empty()
```

```
def is_root(p)
    # test if p is root in self
    return p.parent == nil
```

Traversing trees

A **traversal** visits the nodes of a tree in a systematic manner

When traversing a simpler structure like a list there is one natural traversal strategy (forward or backwards)

Trees are more complex and admit more than one natural way:

- pre-order
- post-order
- in-order (for binary trees)

Preorder Traversal

To do a preorder traversal starting at a given node, we visit the node before visiting its descendants

```
def pre_order(v)
    visit(v)
    for each child w of v
        pre_order(w)
```

If tree is ordered visit the child subtrees in the prescribed order

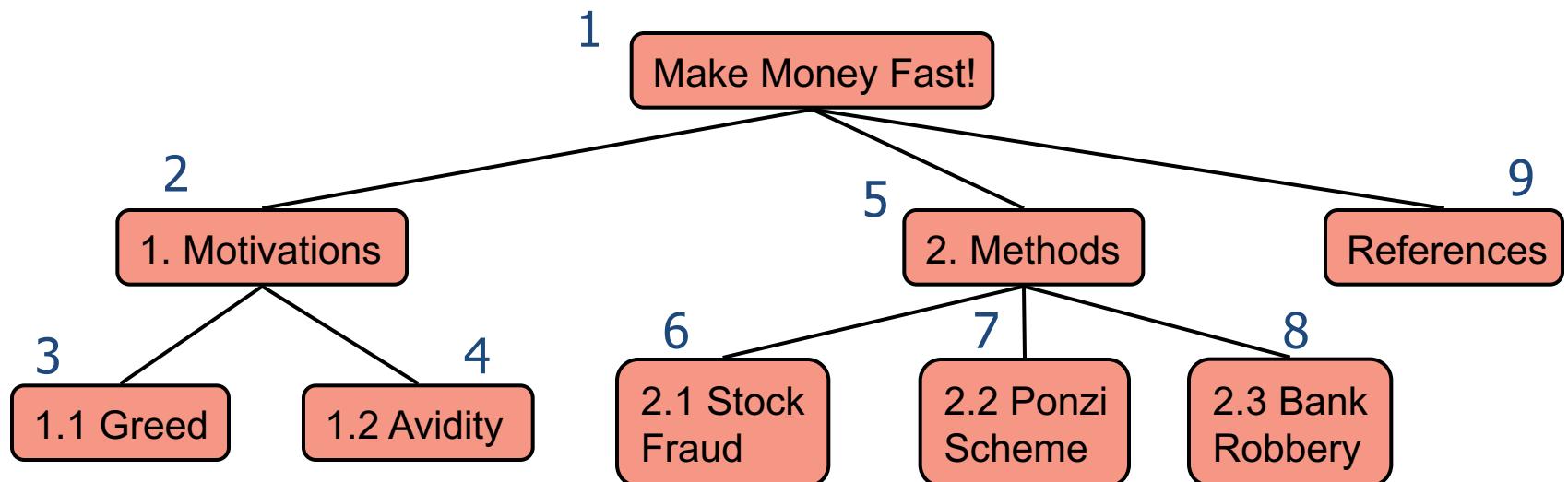
Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

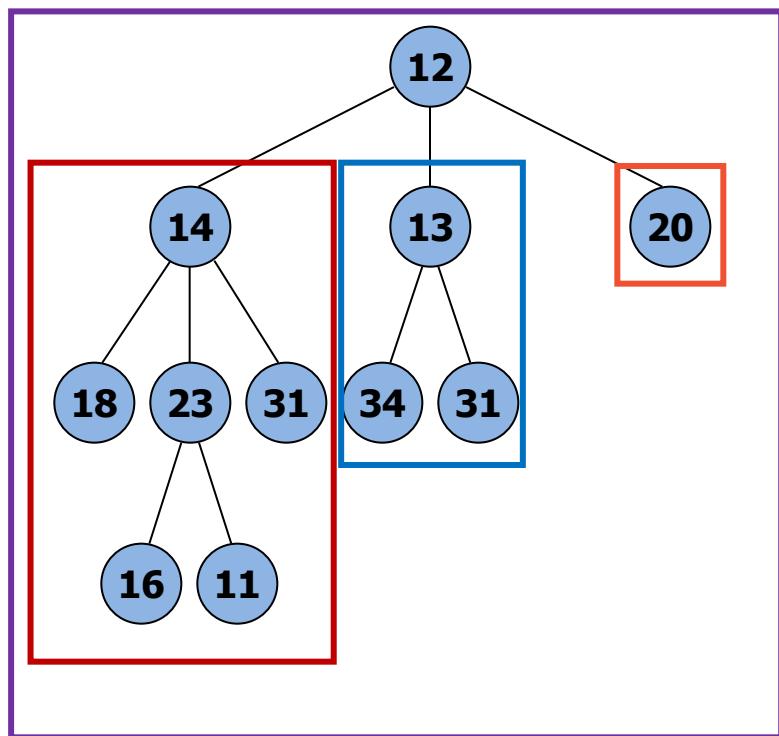
Preorder Traversal Example

Nodes are numbered in the order they are visited when we call `pre_order()` at the root

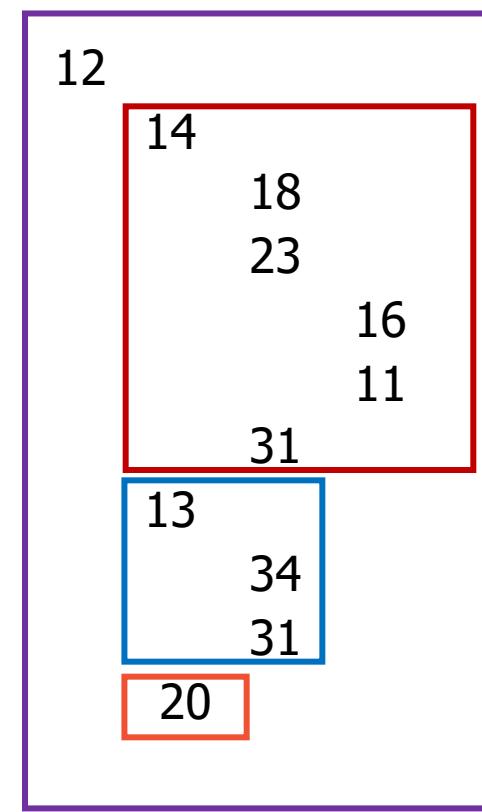
```
def pre_order(v)
    visit(v)
    for each child w of v
        pre_order(w)
```



Preorder Traversal Example



visit
order



Preorder
traversal
of subtree

Postorder Traversal

To do a postorder traversal starting at a given node, we visit the node after its descendants

If tree is ordered visit the child subtrees in the prescribed order

```
def post_order(v)
    for each child w of v do
        post_order (w)
    visit(v)
```

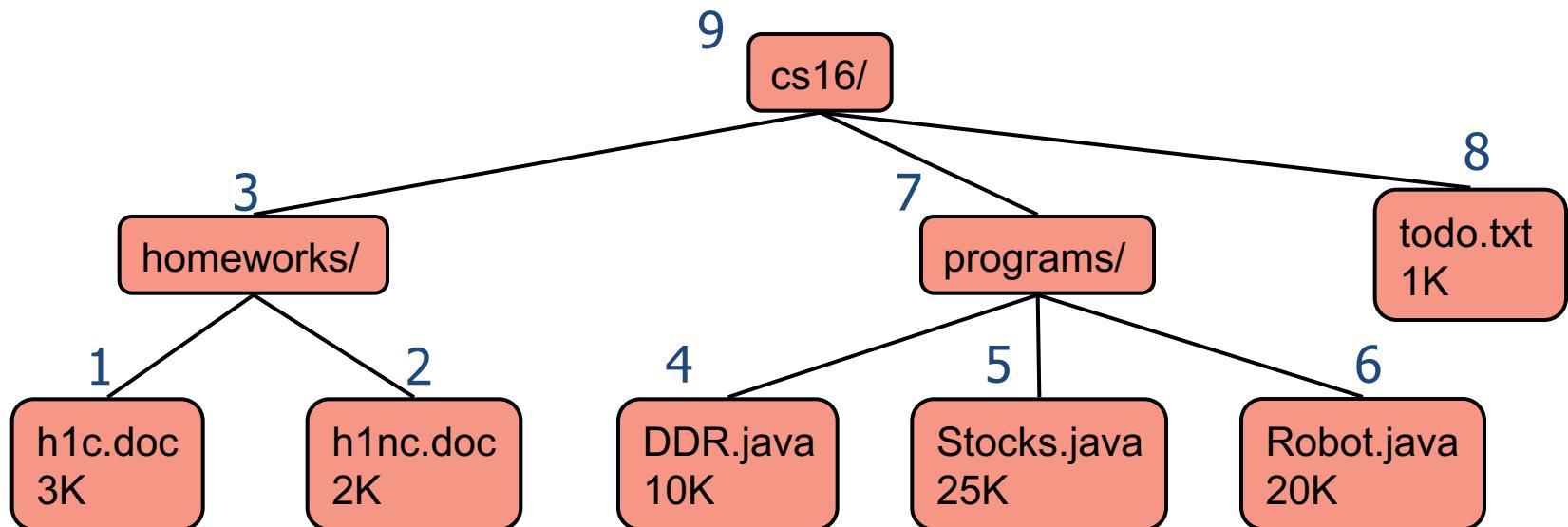
Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

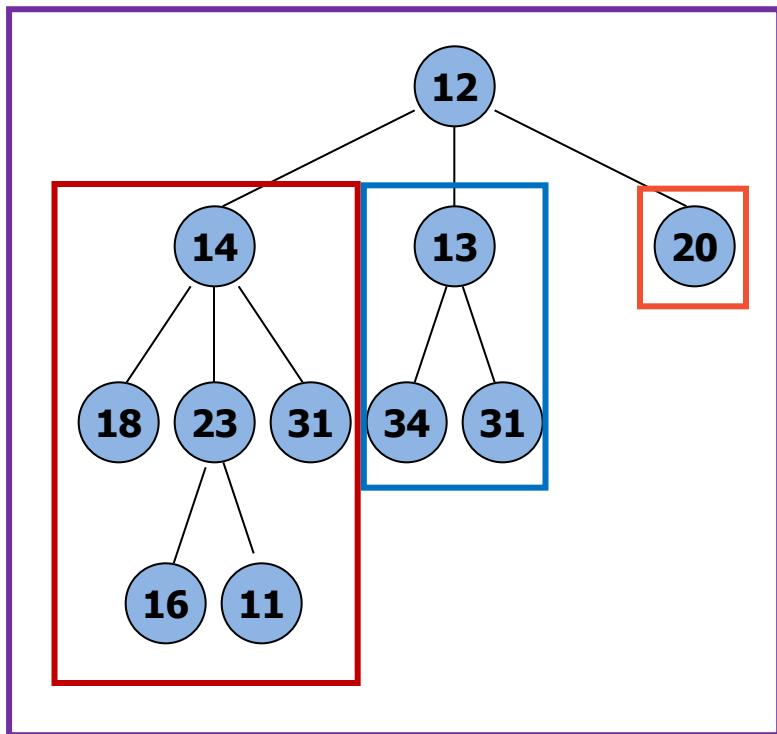
Postorder Traversal

Nodes are numbered in the order they are visited when we call `post_order()` at the root

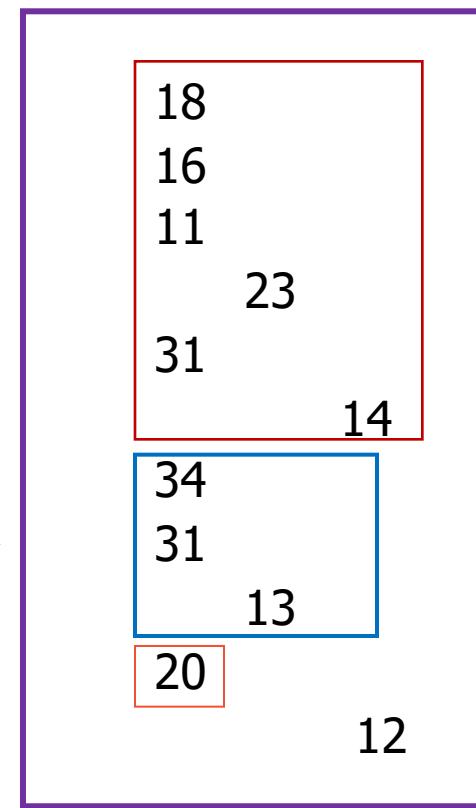
```
def post_order(v)
    for each child w of v do
        post_order (w)
    visit(v)
```



Traversing in postorder



visit
order



Postorder
traversal
of subtree

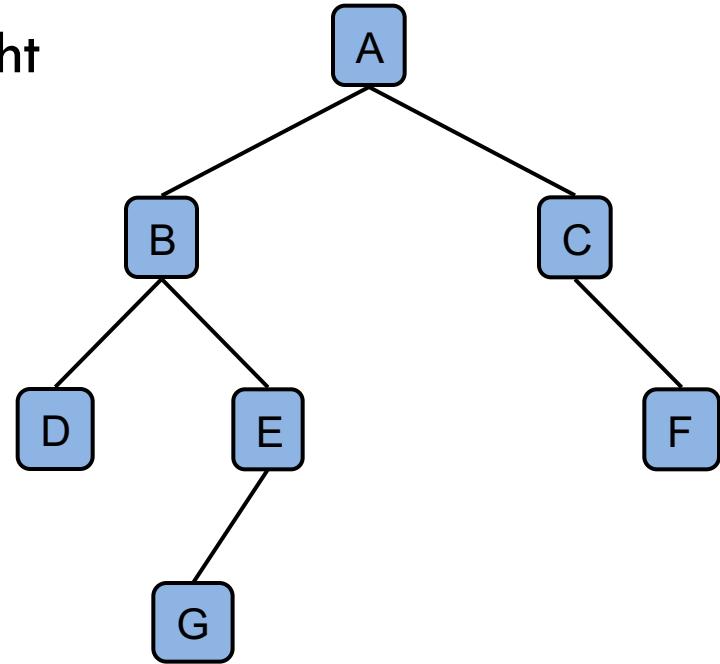
Binary Trees

A **binary tree** is an ordered tree with the following properties:

- Each internal node has at most two children
- Each child node is labeled as a **left child** or a **right child**
- Child ordering is left followed by right

The right/left subtree is the subtree root at the right/left child.

We say the tree is **proper** if every internal node has two children

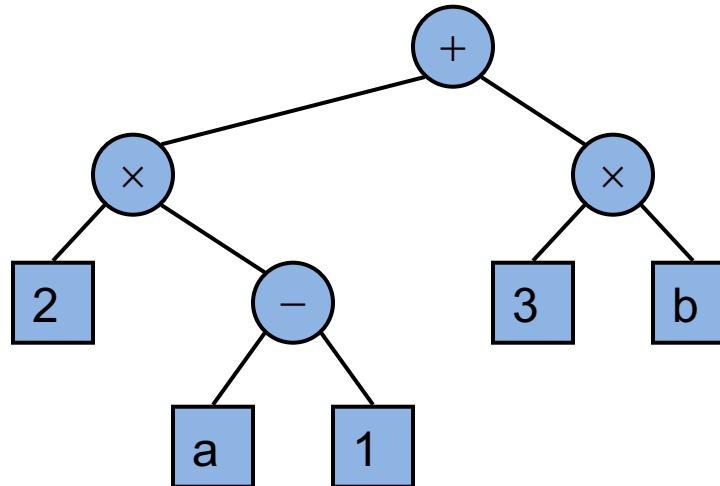


Binary tree application: Arithmetic expression tree

Binary tree associated with an arithmetic expression

- internal nodes: operators
- external nodes: operands

Example: Arithmetic expression tree for $(2 \times (a - 1) + (3 \times b))$

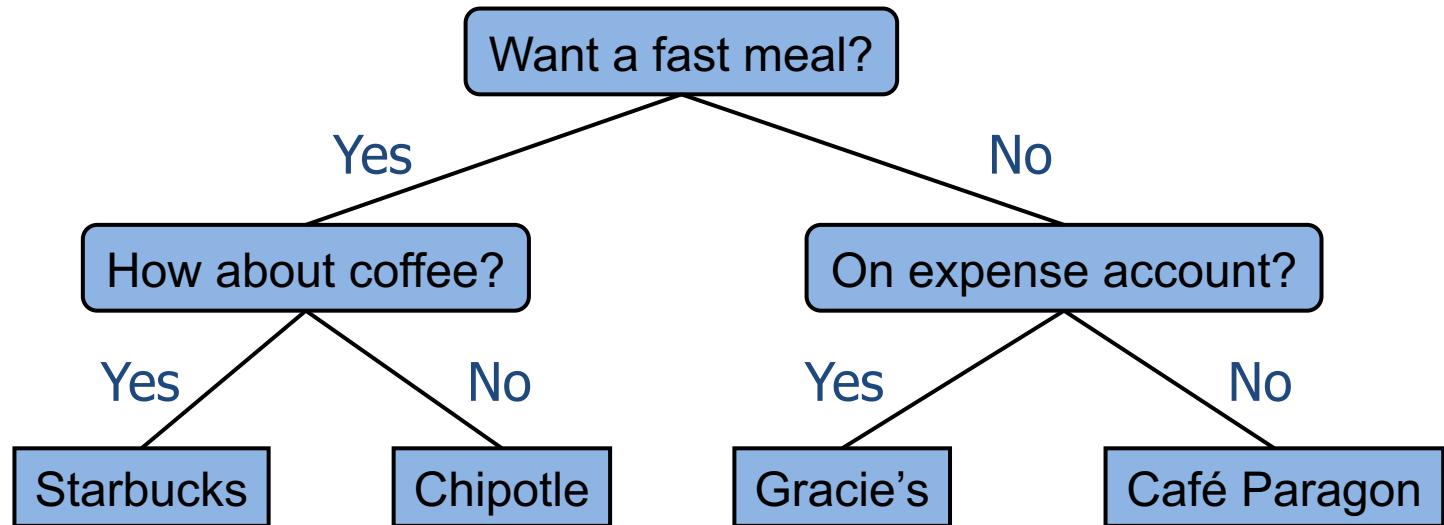


Binary tree application: Decision trees

Tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions

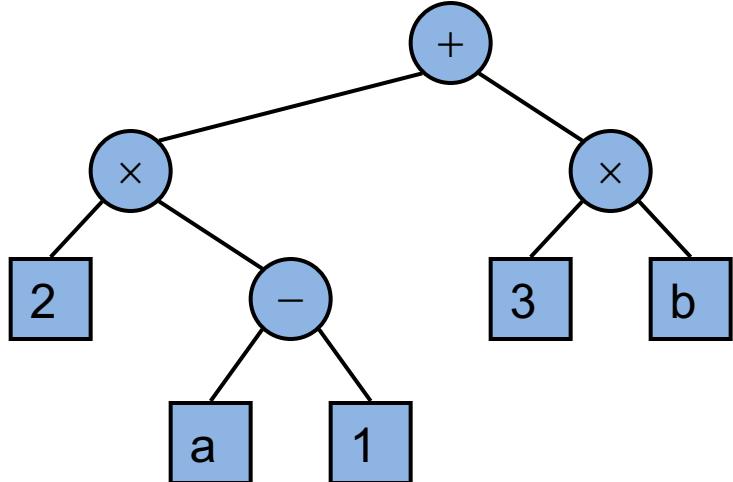
Example: dining decision



Print Arithmetic Expressions

Extended inorder traversal:

- print operand or operator when visiting node
- print "(" before left subtree
- print ")" after right subtree



```
def print_expr(v)
    if v.left ≠ nil then
        print("(")
        print_ expr(v.left)
    print(v.element)
    if v.right ≠ nil then
        print_ expr(v.right)
    print ")")
```

$$((2 \times (a - 1)) + (3 \times b))$$

Binary Tree Operations

- A **binary tree** extends the Tree operations, i.e., it inherits all the methods of a tree.
- Update methods may be defined by data structures implementing the binary tree
- Additional methods:
 - position **leftChild(p)**
 - position **rightChild(p)**
 - position **sibling(p)**

return null when there is no left, right, or sibling of p, respectively

Node object for binary trees

Node object implementation typically has the following attributes:

- value: the value associated with this Node
- left: left child of this Node
- right: right child of this Node
- parent: (optional) the parent of this Node

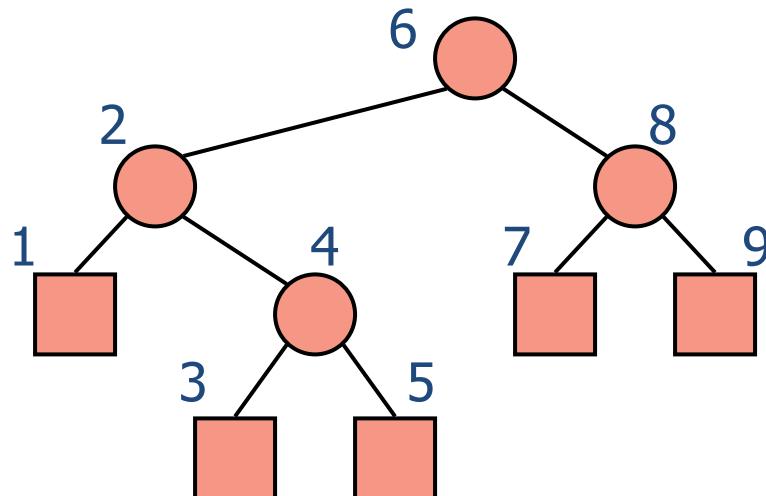
```
def is_internal(p)
    # test if p is internal
    return p.left != nil or p.right != nil
```

Inorder Traversal

To do an inorder traversal starting at a given node, the node is visited after its left subtree but before its right subtree

Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

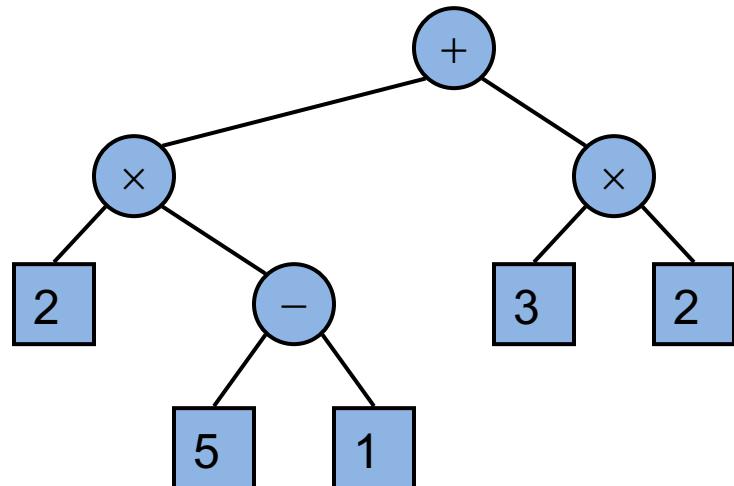


```
def in_order(v)
    if v.left ≠ null then
        in_order(v.left)
    visit(v)
    if v.right ≠ null then
        in_order(v.right)
```

Evaluate Arithmetic Expressions

Extended postorder traversal

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



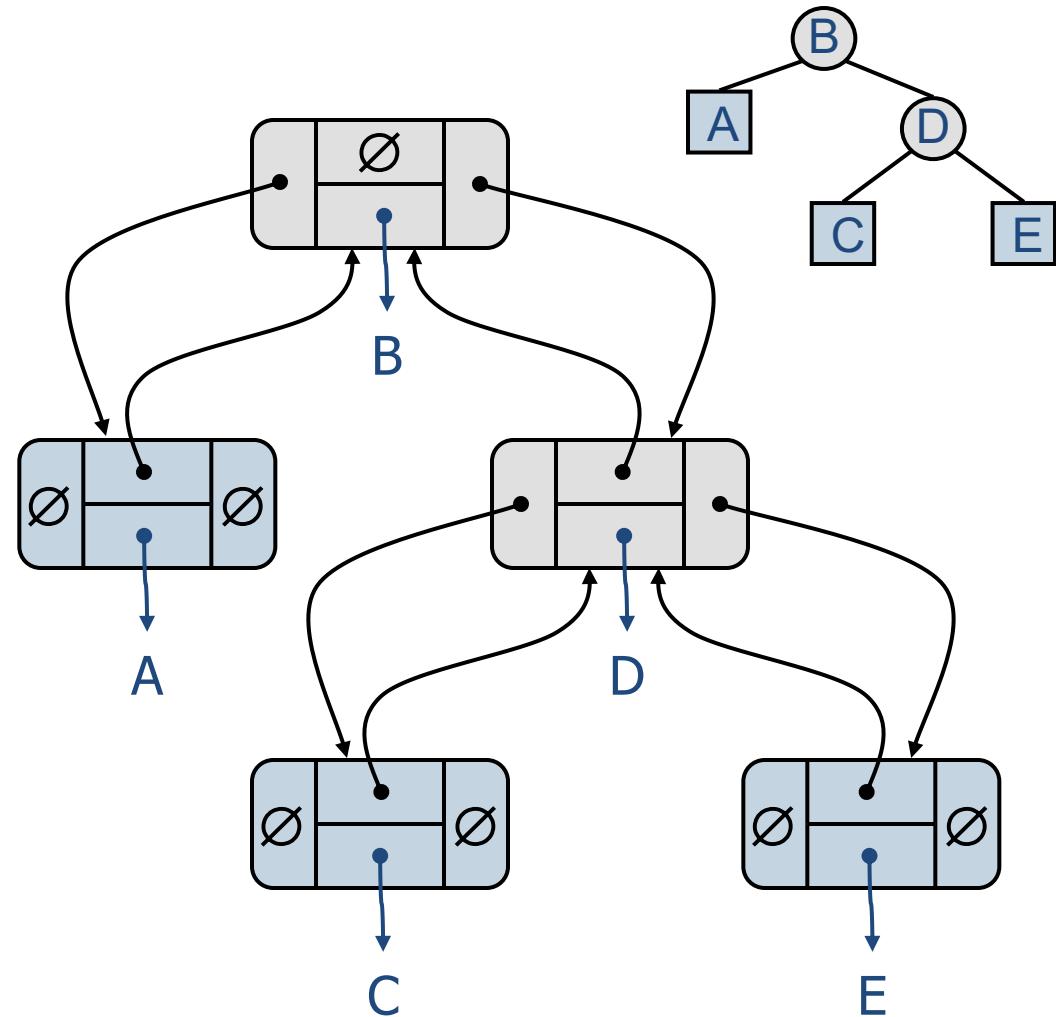
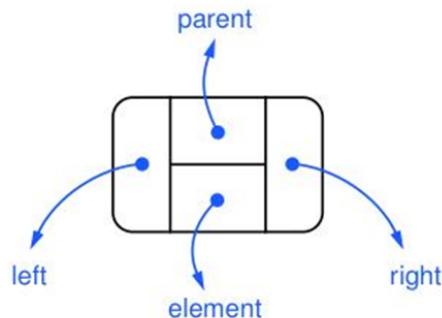
```
def eval_expr(v)
    if v.is_external() then
        return v.element
    else
        x ← eval_expr(v.left)
        y ← eval_expr(v.right)
        ⊕ ← v.element
        return x ⊕ y
```

Linked Structure for Binary Trees

A node is represented by an object storing

- Element
- Parent node
- Left child node
- Right child node

Node objects implement the Position ADT

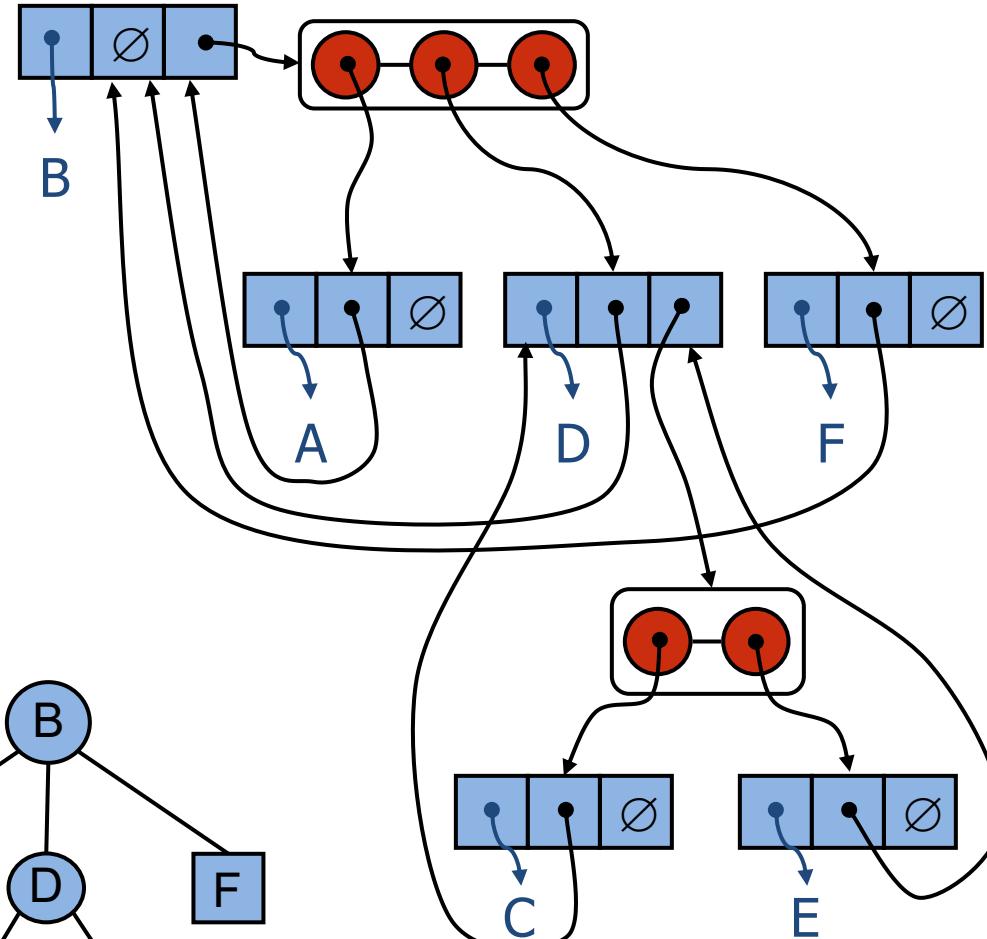
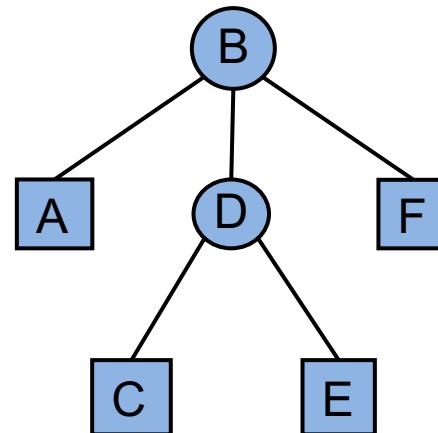
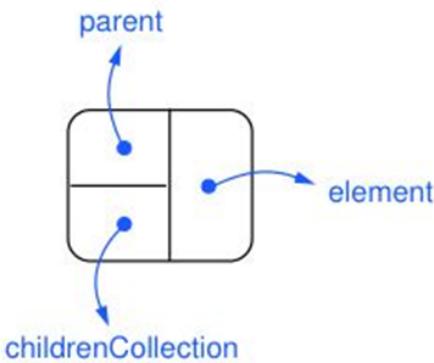


Linked Structure for General Trees

A node is represented by an object storing

- Element
- Parent node
- Sequence of children

Node objects implement the Position ADT



Examples of recursive code on trees

Calculating depth

```
def depth(v)
    if v.parent == nil then
        return 0
    else
        return depth(v.parent) + 1
```

Calculating height

```
def height(v)
    if v.isExternal() then
        return 0
    else
        h = 0
        for each child w of v do
            h = max(h, height(w))
        return h + 1
```

Complexity analysis of recursive algorithms on trees

Sometimes, the method may call itself on all children

- In worst case, do a call on every node
- If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the number of nodes

Sometimes, the method calls itself on at most one child

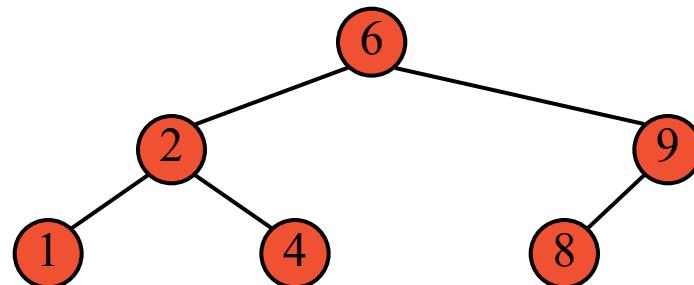
- In worst case, do one call at each level of the tree
- If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the height of the tree

Binary Search Tree

So far we've been focused on the structure of the tree. The real usefulness of trees hinges on the values we store at each element and how these values are laid out.

BST is a data structure for storing values that can be sorted. These values are laid out so that an in-order traversal of the BST visits the values in sorted order.

Can search for elements and insert/delete operations run in $O(\log n)$ time provided the tree is “balanced”. More on that next week!



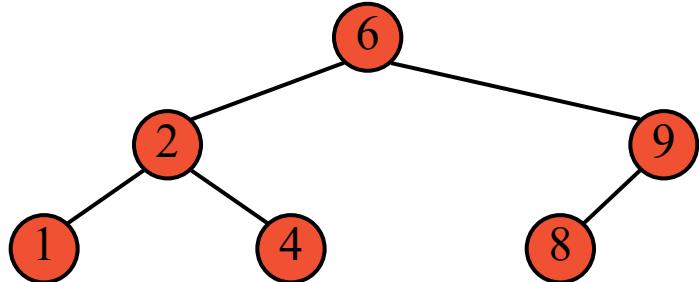
Binary Search Trees (BST)

A **binary search tree** is a binary tree storing keys (or key-value pairs) satisfying the following BST property

For any node v in the tree and
any node u in the left subtree of v and
any node w in the right subtree of v ,

$$\text{key}(u) < \text{key}(v) < \text{key}(w)$$

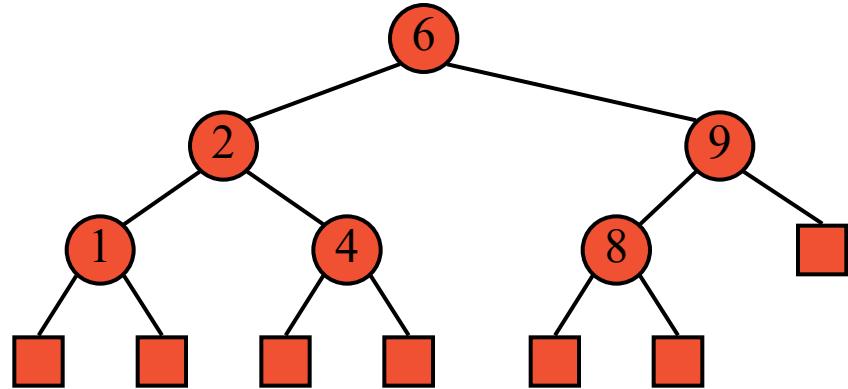
Note that an inorder traversal
of a binary search tree visits the keys
in increasing order.



BST Implementation

To simplify the presentation of our algorithms, we only store keys (or key-value pairs) at **internal** nodes

External nodes do not store items (and with careful coding, can be omitted, using null to refer to such)

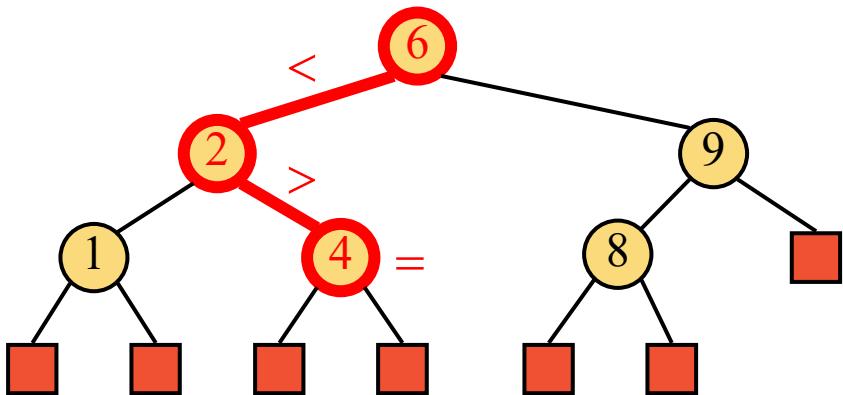


Searching with a Binary Search Tree

To search for a key k , we trace a downward path starting at the root

To decide whether to go left or right, we compare the current node with k

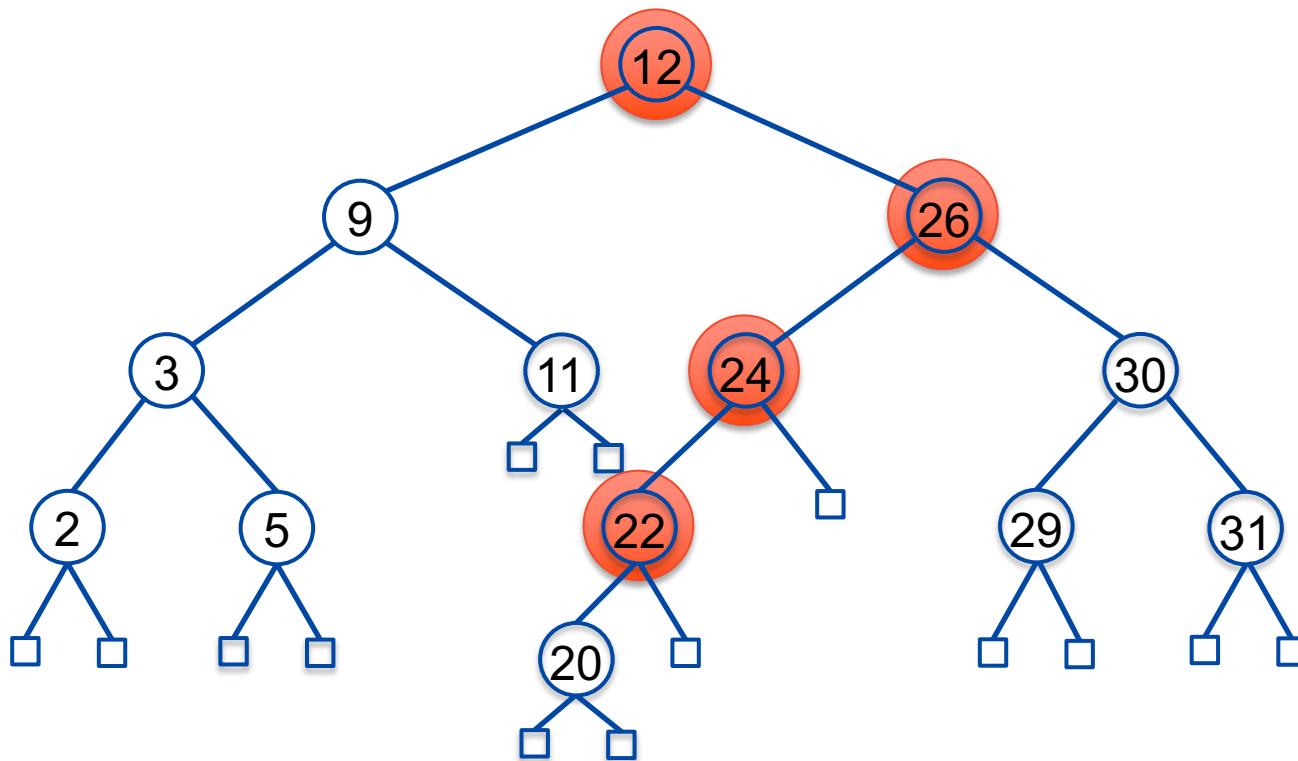
If we reach an external node, this means that the key is not in the data structure



```
def search(k, v)
    if v.isExternal() then
        # unsuccessful search
        return v
    if k = key(v) then
        # successful search
        return v
    else if k < key(v) then
        # recurse on left subtree
        return search(k, v.left)
    else
        # that is k > key(v)
        # recurse on right subtree
        return search(k, v.right)
```

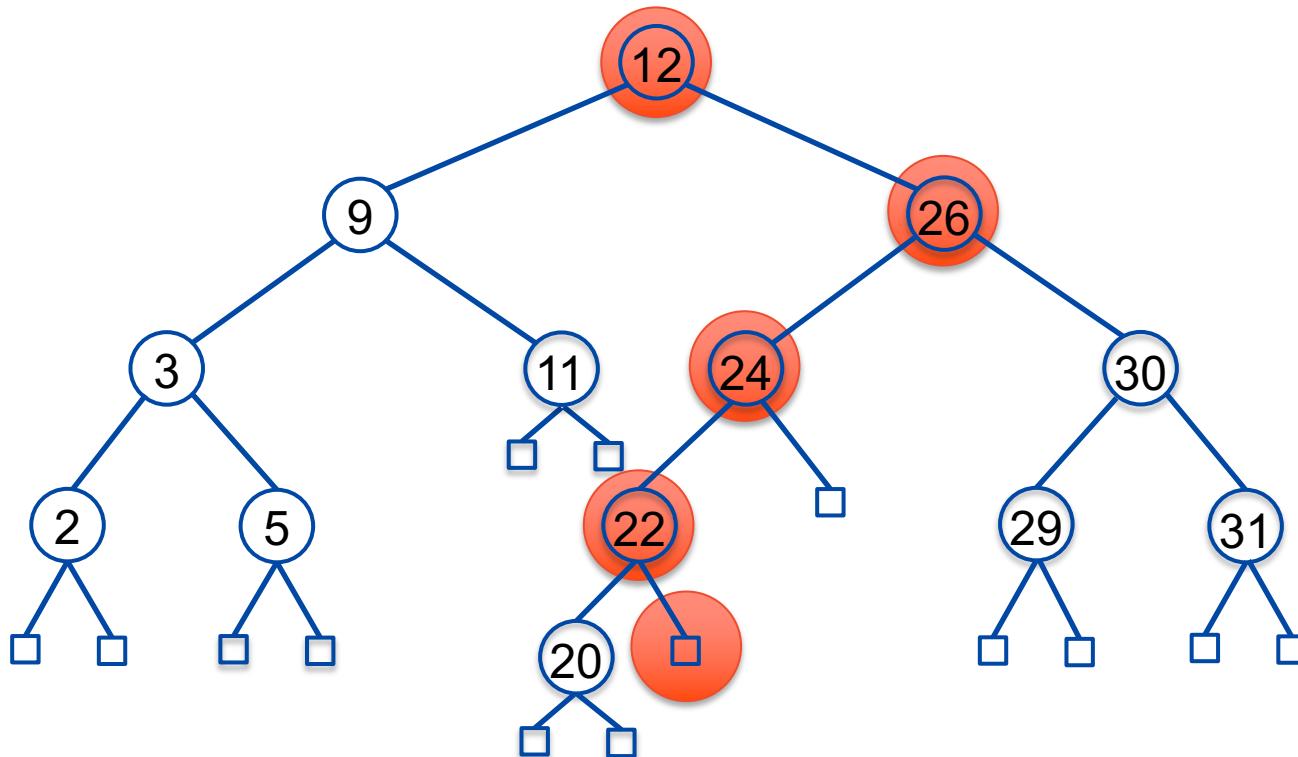
Example: Find 22

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Example: Find 23

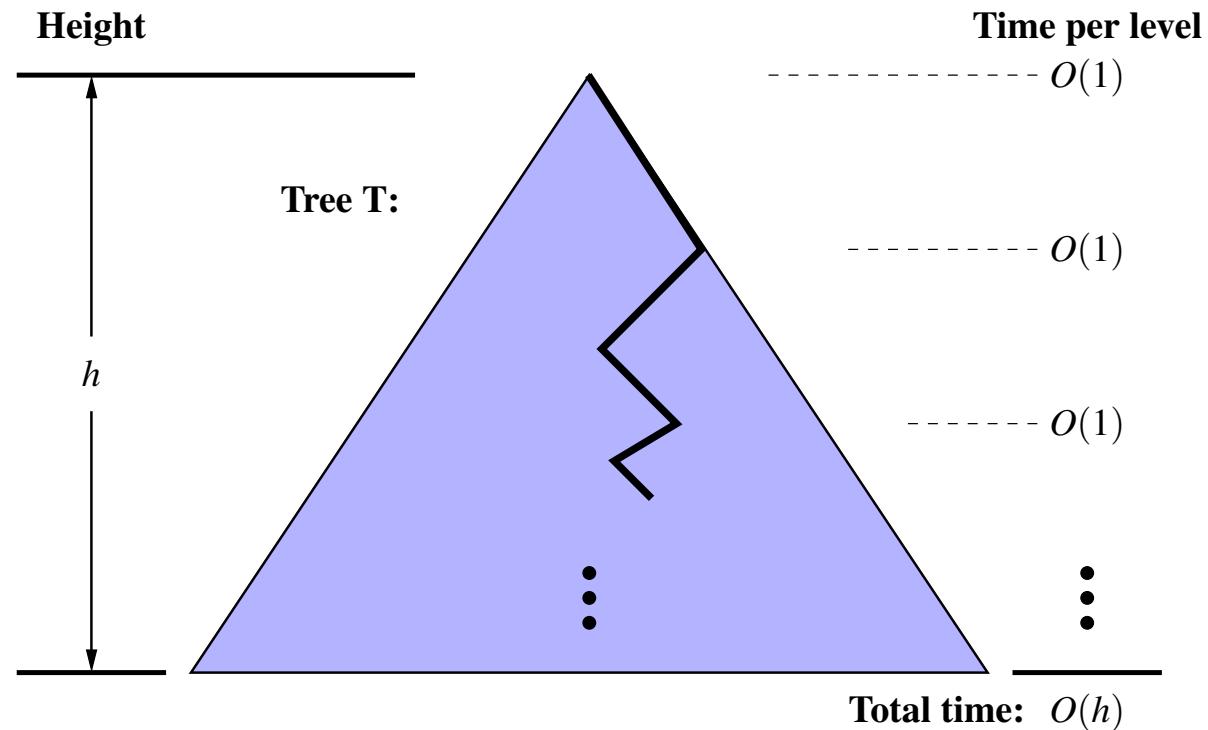
$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Analysis of Binary Tree Searching

Runs in $O(h)$ time, where h is the height of the tree.

Next week we will see how to balance a BST so that $h = O(\log n)$ even if we insert and delete nodes.



B-trees

Database systems use a generalization of BST to index data and allow fast lookups. Because data is stored in external memory we need a different strategy.

Two key features of external memory complicate the task:

- External memory is orders of magnitude slower than internal memory
- External memory is transferred in blocks of size B (order of 10^3 bytes)

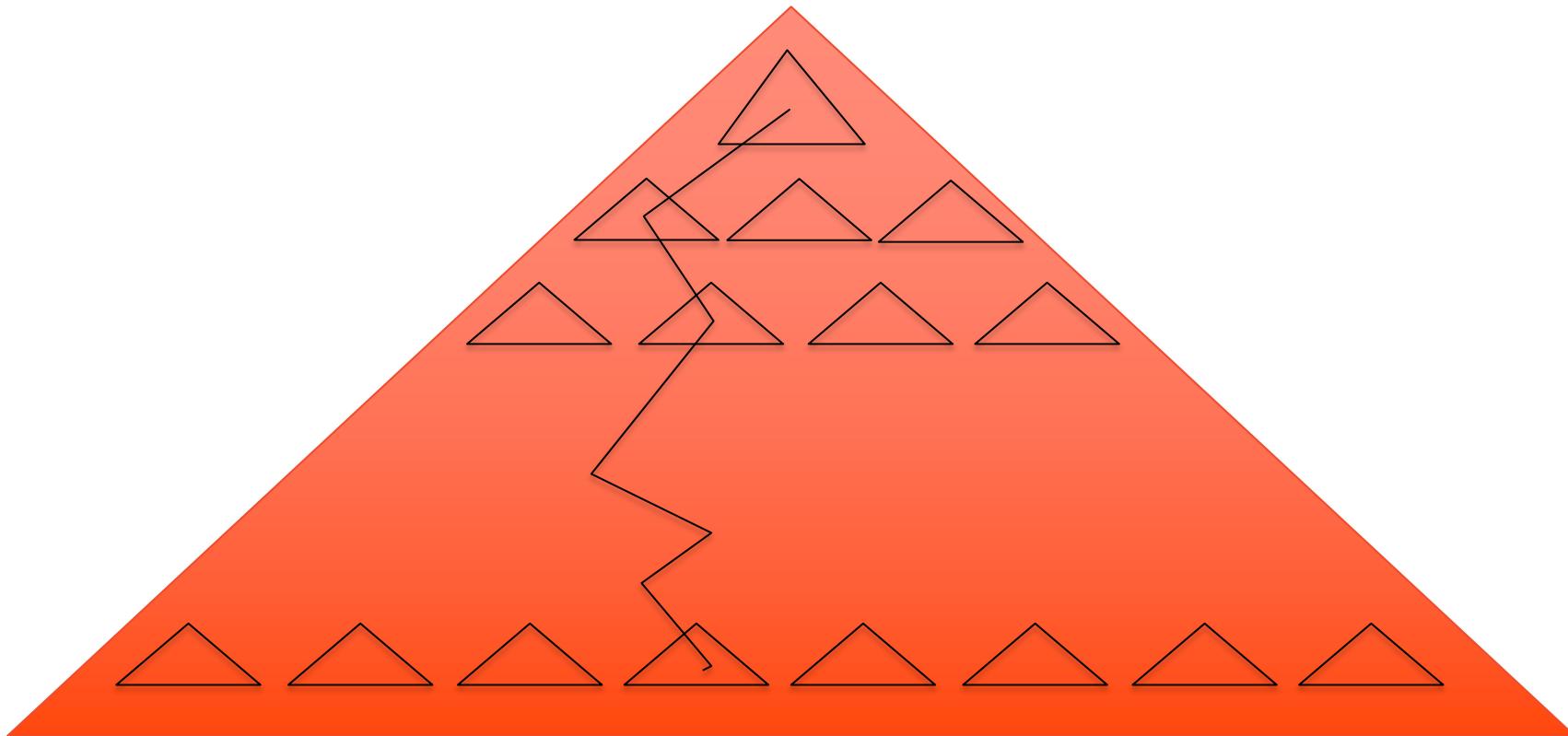
Our goal is to minimize the number of transfers (a.k.a. I/O operation)

If we implemented BST as is in external memory we need $\log_2 n$ I/Os even if the tree is perfectly balanced. When n is very large, and the throughput of queries is high this becomes too slow, so databases systems resort to a different data structure.

High level idea behind B-trees

Group tree into chunks of size **B** and layout each chunk in its own external memory block.

Number of I/Os equal number of chunks we need to fetch

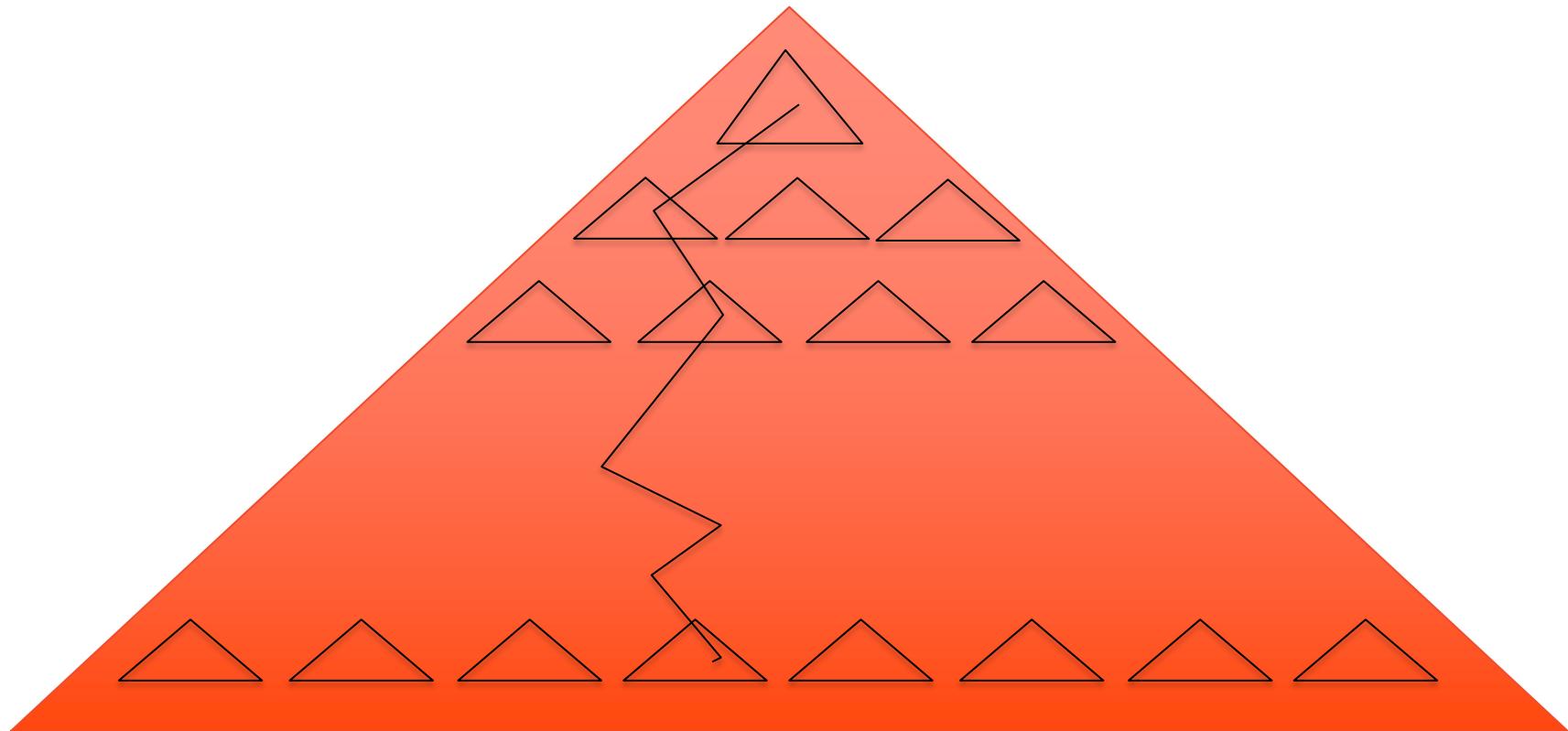


Performance of B-trees

Each chunks has close to B nodes, so each chunk has height close to $\log_2 B$

Thus, number of I/Os is close to $\log_2 n / \log_2 B$ or equivalently $\log_B n$

Recall that $B \approx 10^3$ so $\log_2 B \approx 10$

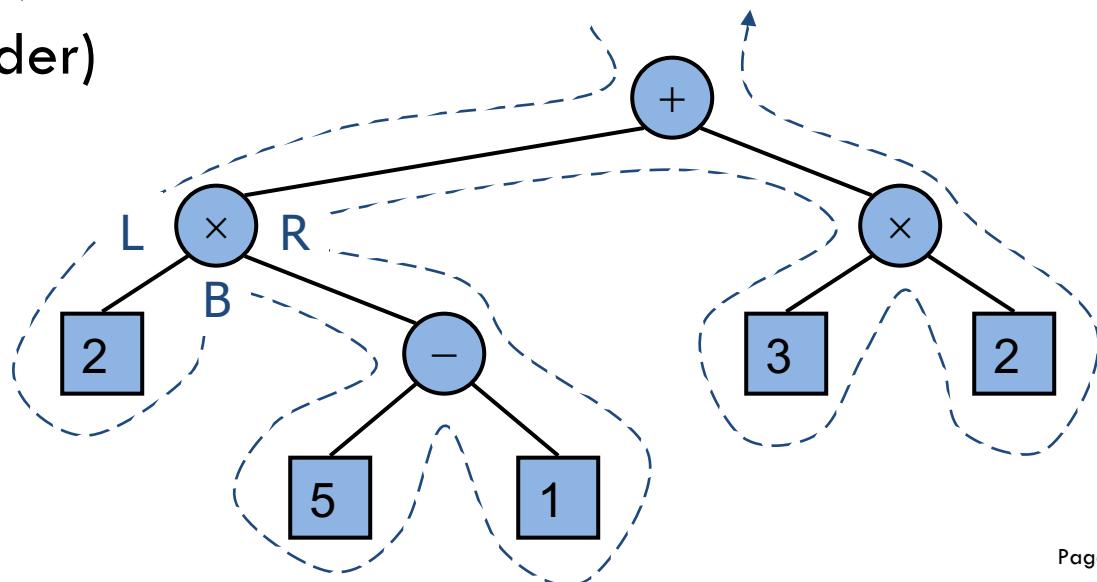


Euler Tour Traversal

Generic traversal of a binary tree. Includes as special cases the preorder, postorder and inorder traversals

Walk around the tree and visit each node three times:

- on the left (preorder)
- from below (inorder)
- on the right (postorder)



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Binary Search Trees [GT 3.1-2] [GT 4.2]

Dr. Julian Mestre
School of Information Technologies

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



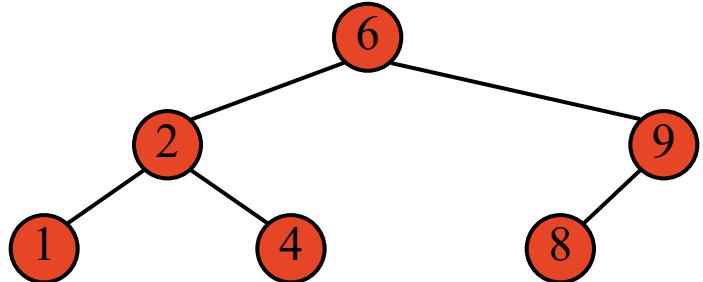
Binary Search Trees (BST)

A **binary search tree** is a binary tree storing keys (or key-value pairs) satisfying the following BST property

For any node v in the tree and
any node u in the left subtree of v and
any node w in the right subtree of v ,

$$\text{key}(u) < \text{key}(v) < \text{key}(w)$$

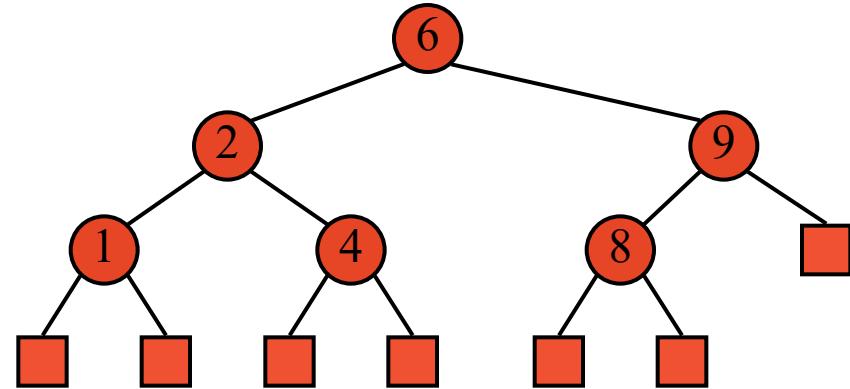
Note that an inorder traversal
of a binary search tree visits the keys
in increasing order.



BST Implementation

To simplify the presentation of our algorithms, we only store keys (or key-value pairs) at **internal** nodes

External nodes do not store items (and with careful coding, can be omitted, using null to refer to such)

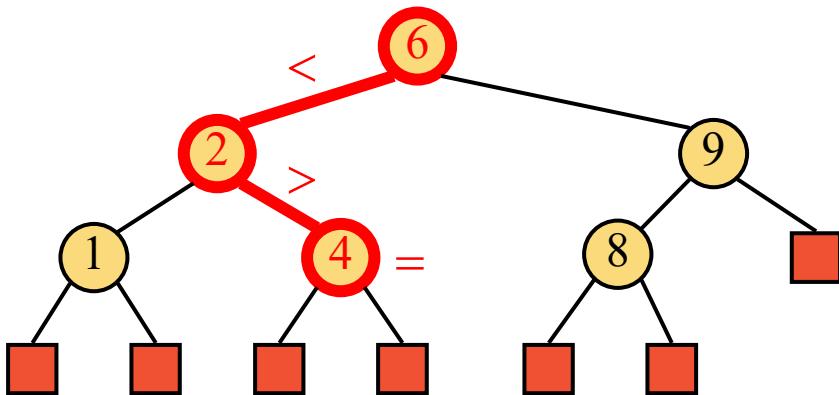


Searching with a Binary Search Tree

To search for a key k , we trace a downward path starting at the root

To decide whether to go left or right, we compare the key of the current node v with k

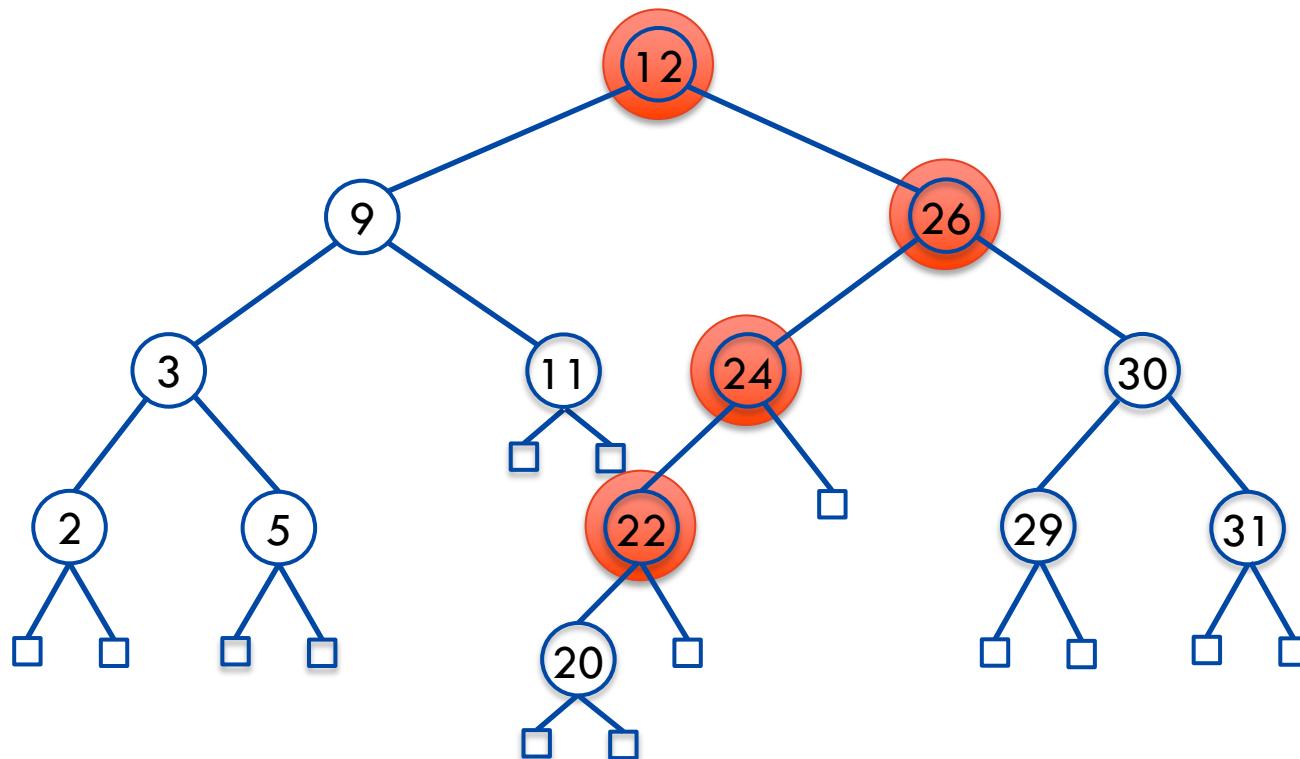
If we reach an external node, this means that the key is not in the data structure



```
def search(k, v)
    if v.isExternal() then
        # unsuccessful search
        return v
    if k = key(v) then
        # successful search
        return v
    else if k < key(v) then
        # recurse on left subtree
        return search(k, v.left)
    else
        # that is k > key(v)
        # recurse on right subtree
        return search(k, v.right)
```

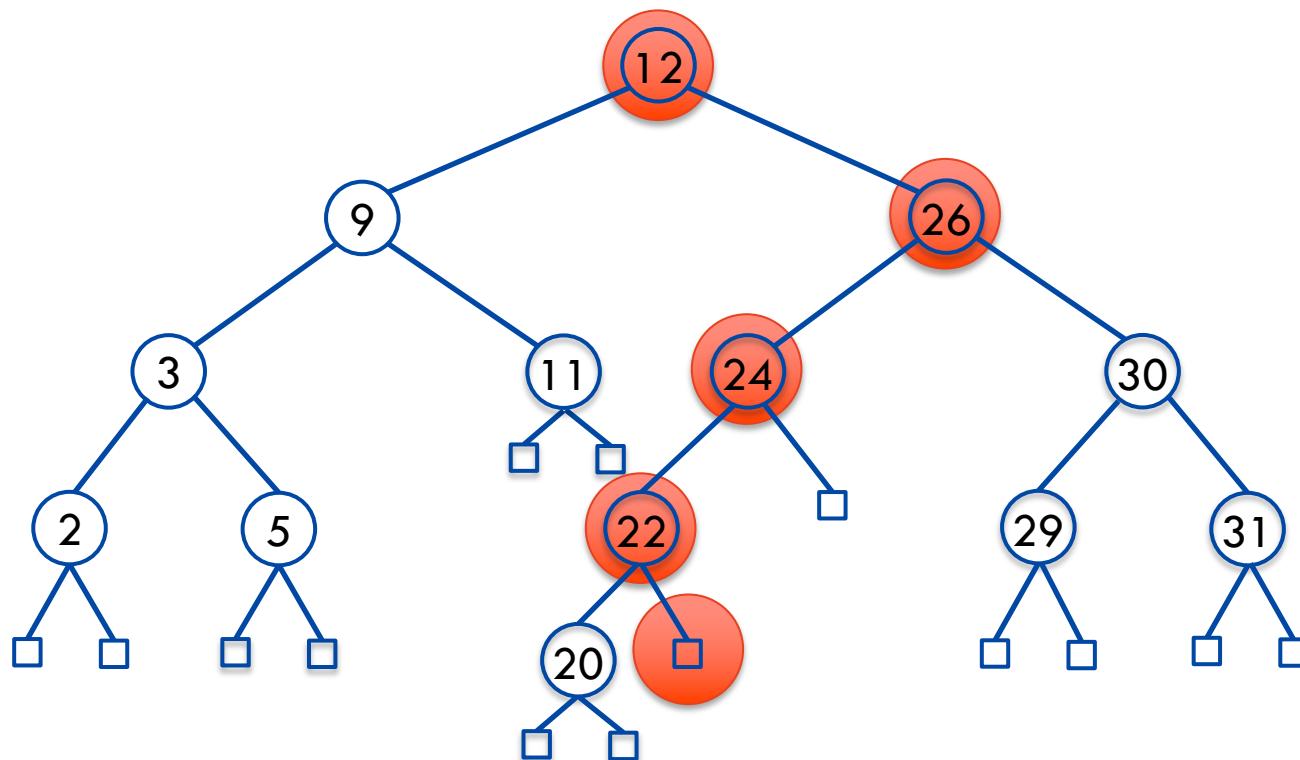
Example: Find 22

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Example: Find 23

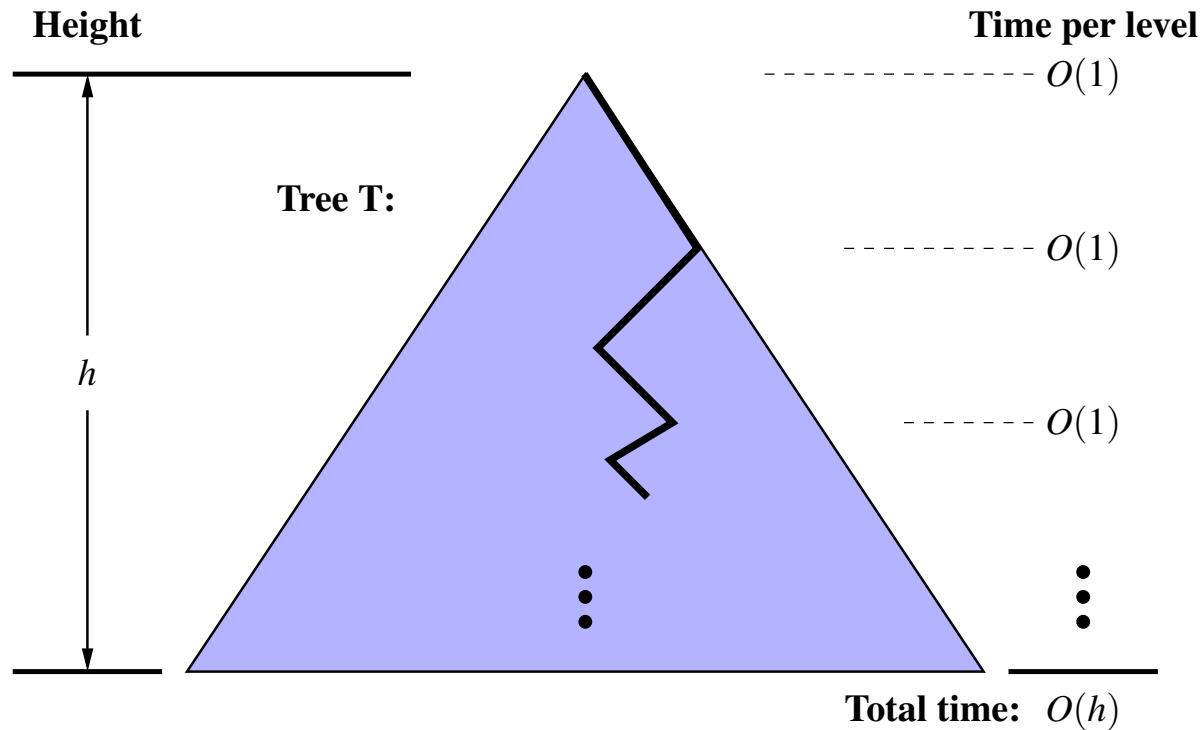
$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Analysis of Binary Tree Searching

Runs in $O(h)$ time, where h is the height of the tree

- ▶ worst case is $h = n - 1$
- ▶ best case is $h < \log_2 n$

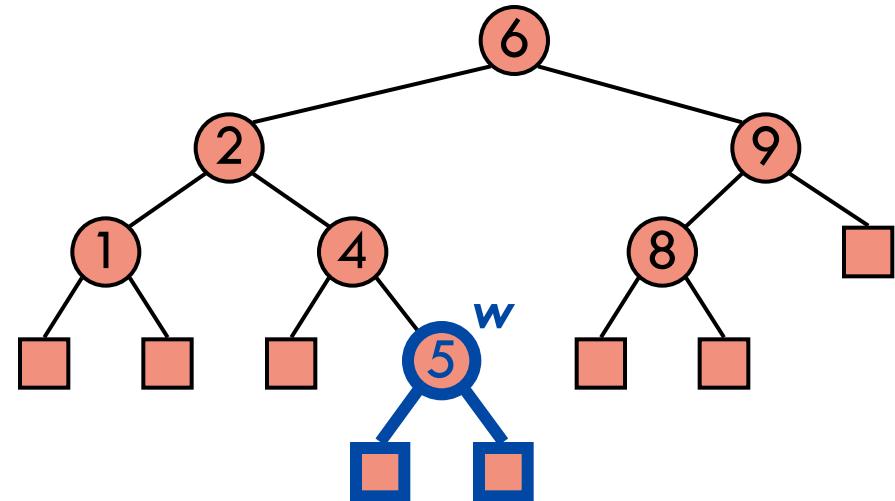
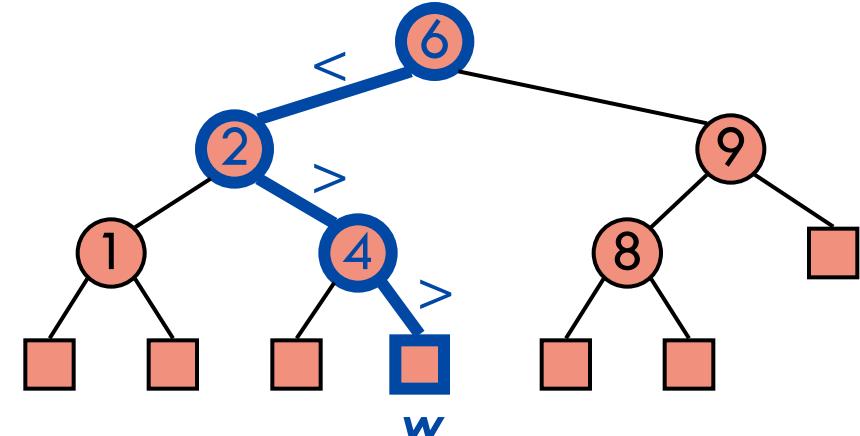


Insertion

To perform operation $\text{put}(k, o)$, we search for key k (using search)

If k is found in the tree, replace the corresponding value by o

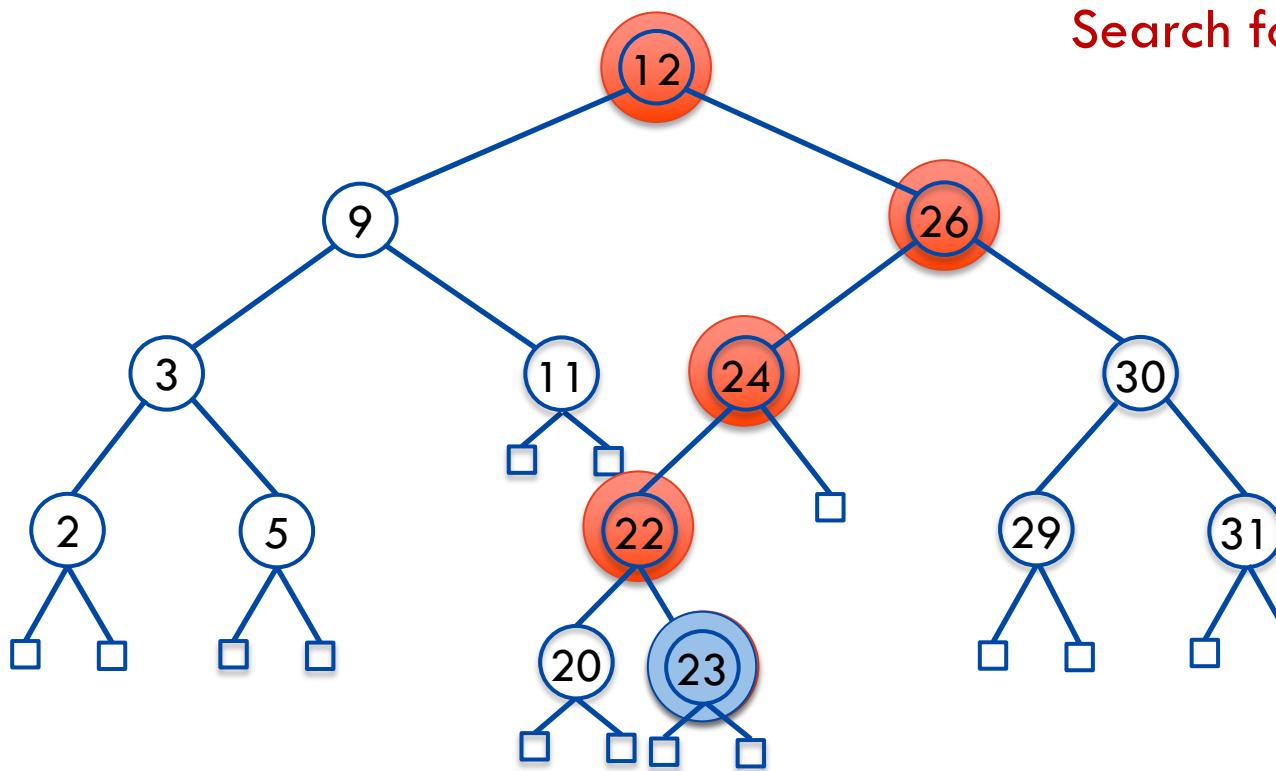
If k is not found, let w be the external node reached by the search. We replace w with an internal node holding (k, o)



Example: Insert 23

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Search for 23



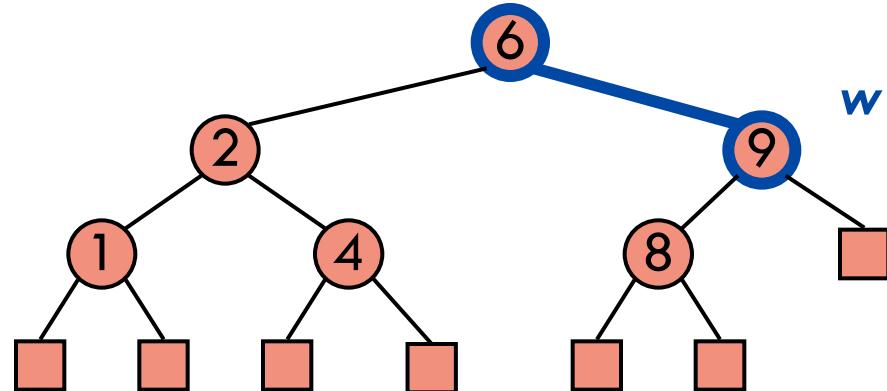
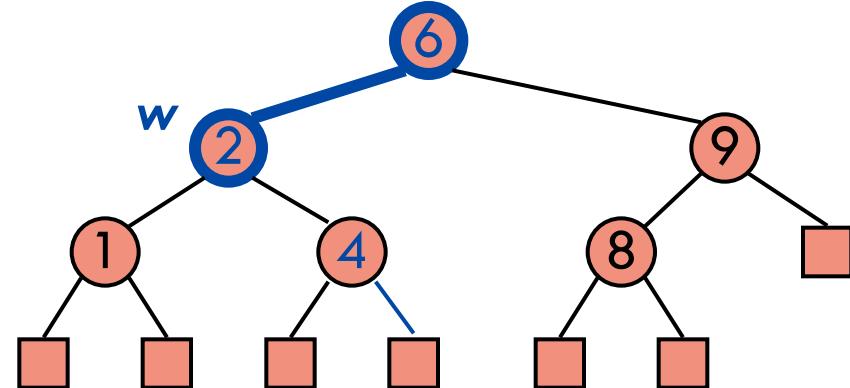
Delete

To perform operation `remove(k)`, we search for key k (using search) to find the node w holding k

We distinguish between two cases

- w has one external child
- w has two internal children

If k is not in the tree we can either throw an exception or do nothing depending on the ADT specs

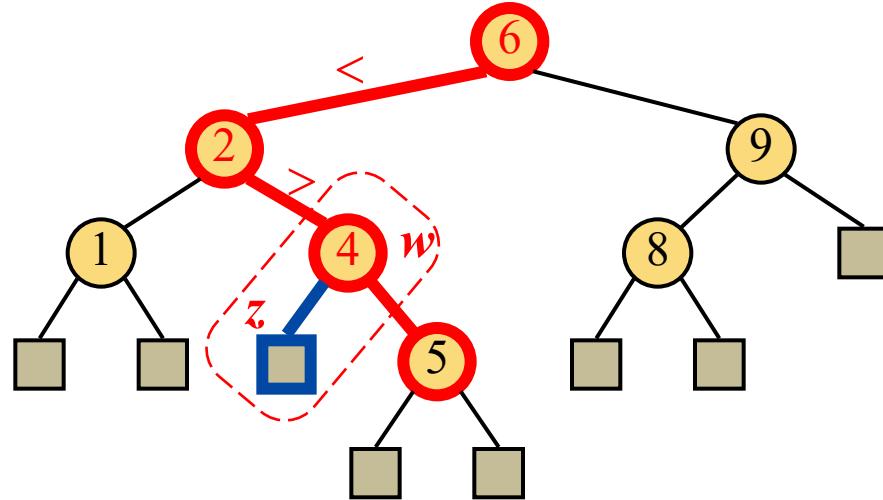


Deletion Case 1

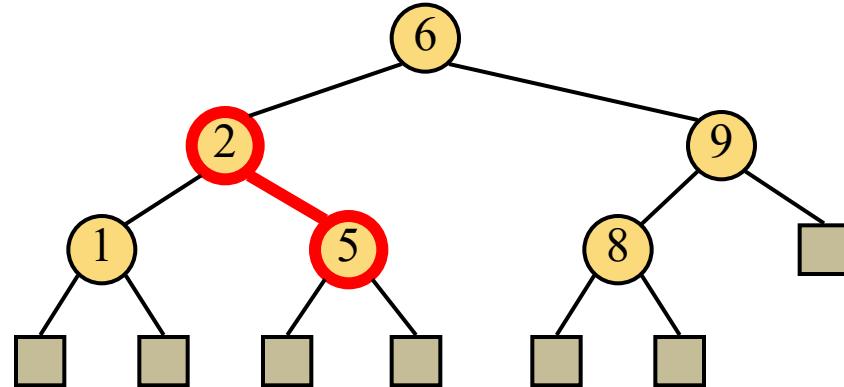
Suppose that the node w we want to remove has an external child, which we call z .

To remove w we

- remove w and z from the tree
- promote the other child of w to take w 's place



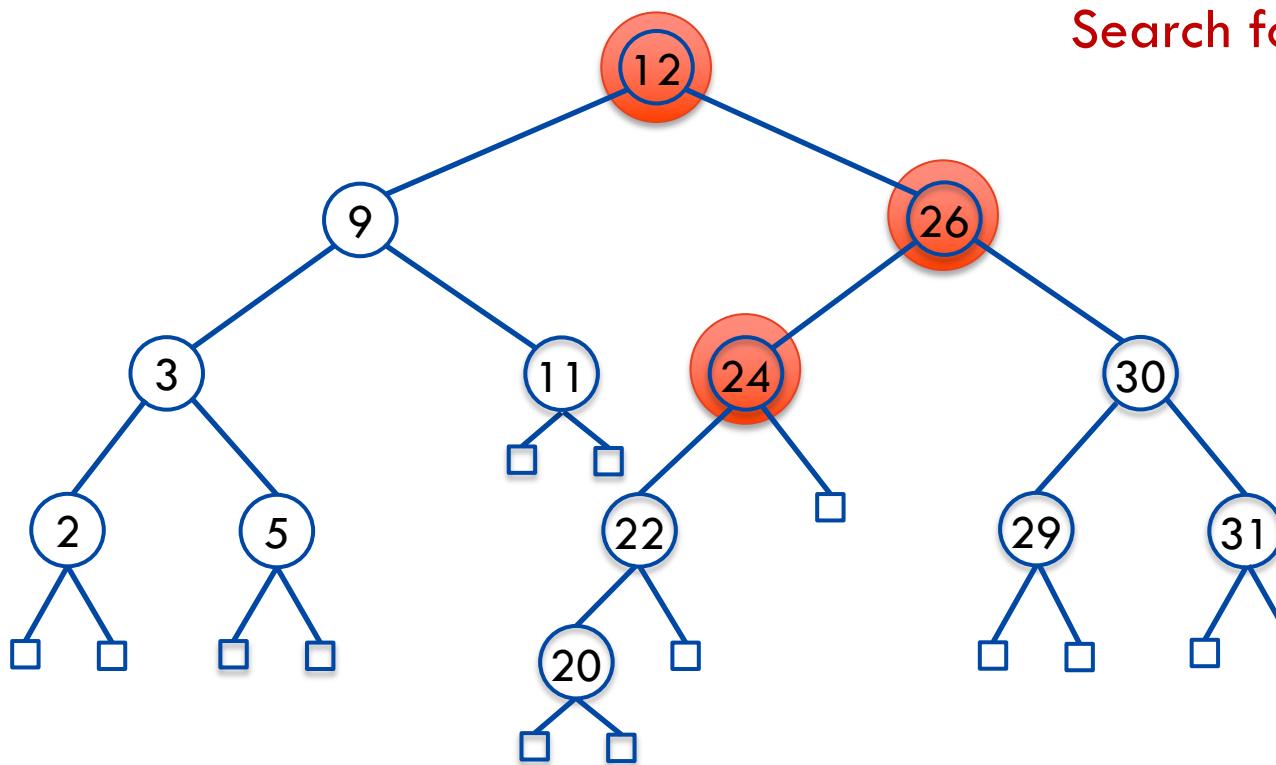
This preserves the BST property



Example: Delete 24

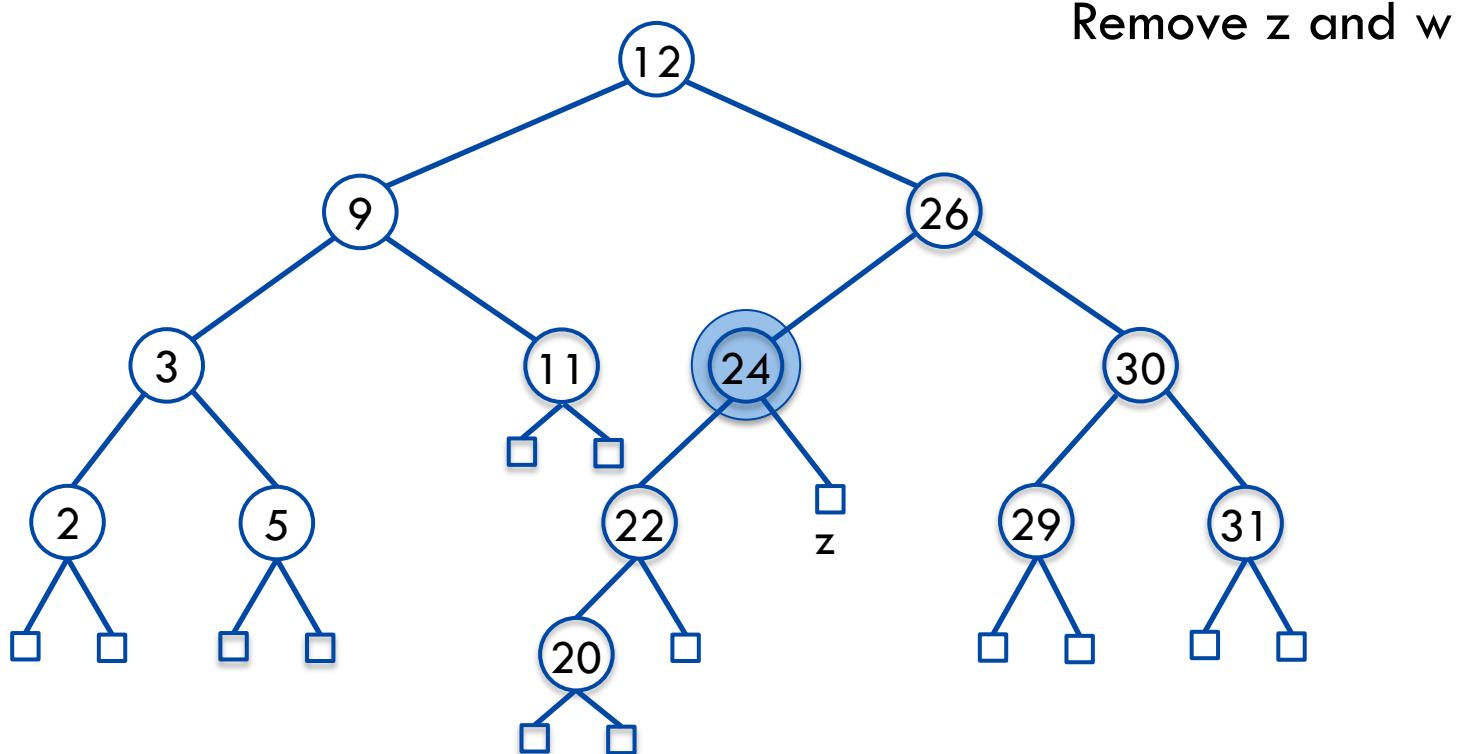
$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$

Search for 24



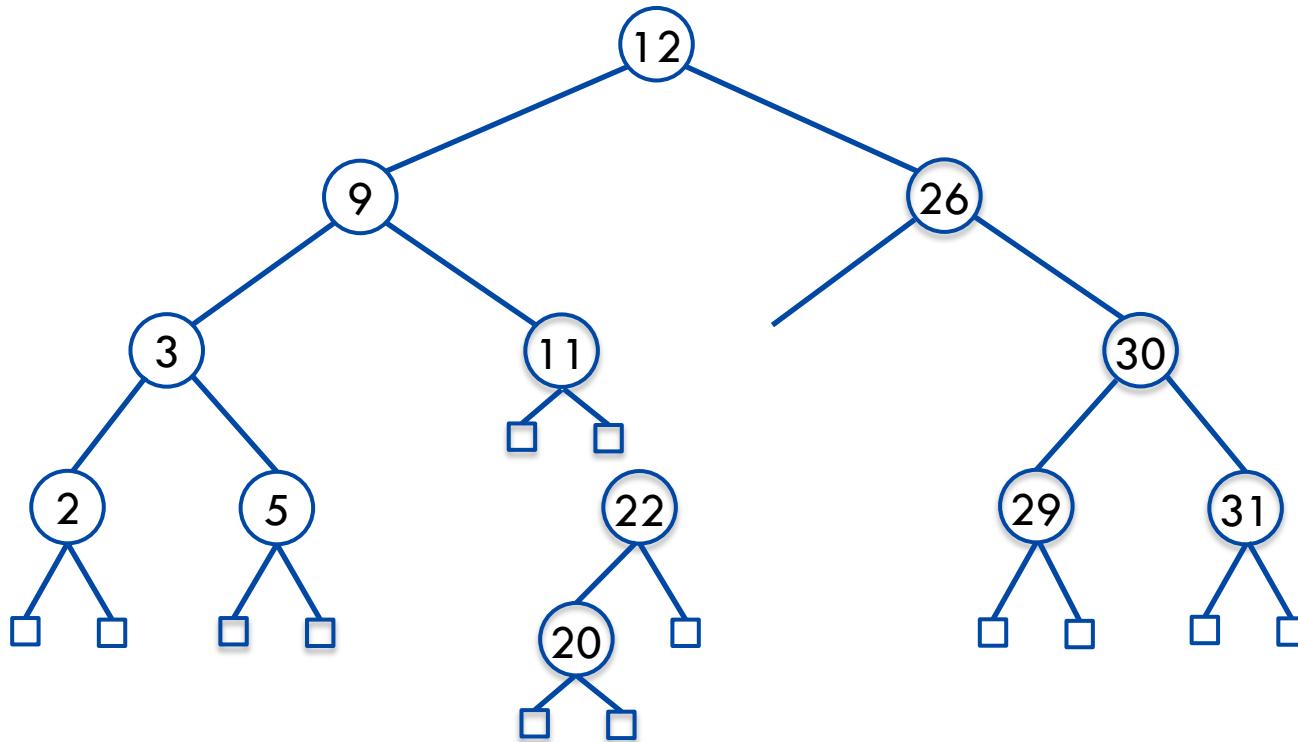
Example: Delete 24

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



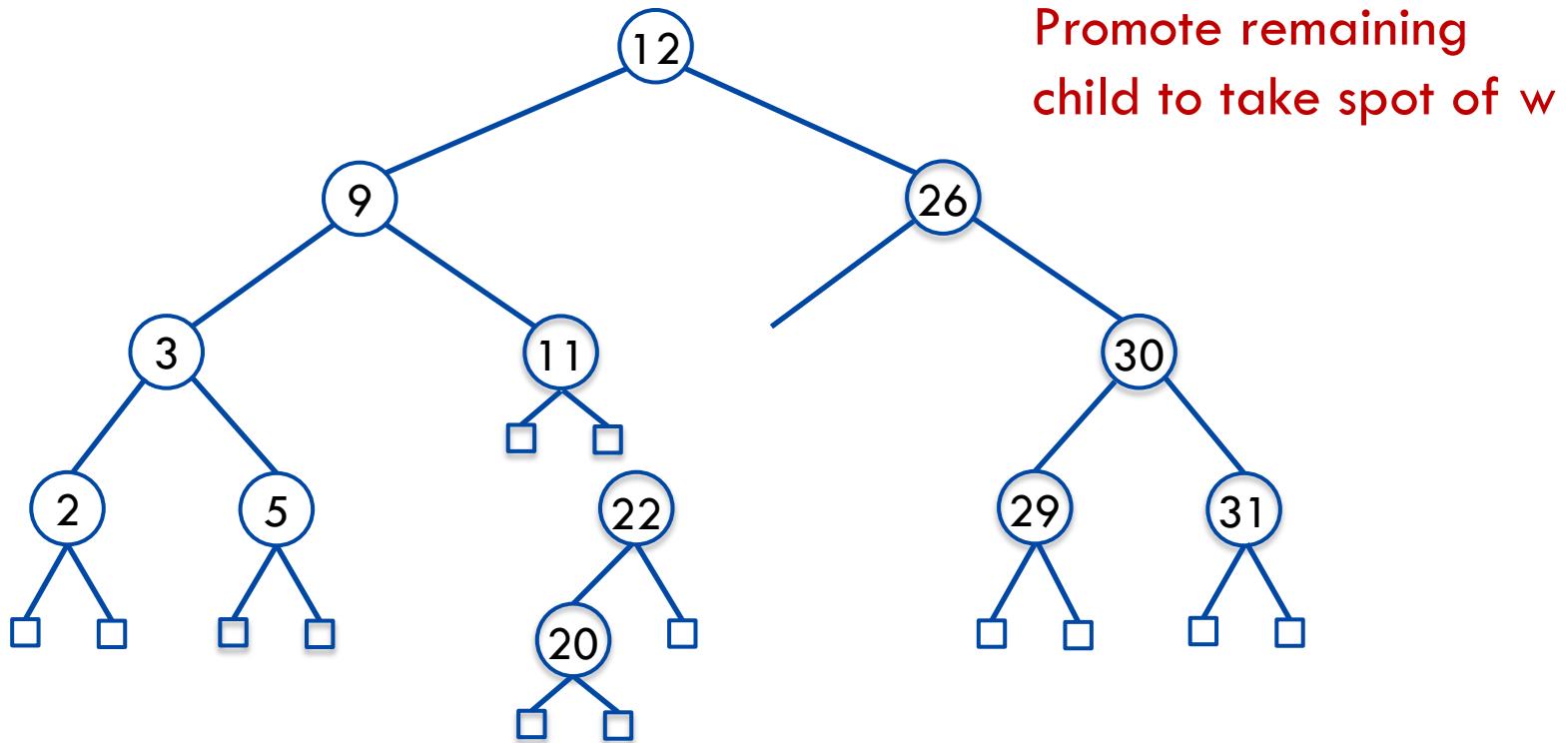
Example: Delete 24

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Example: Delete 24

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



Promote remaining
child to take spot of w

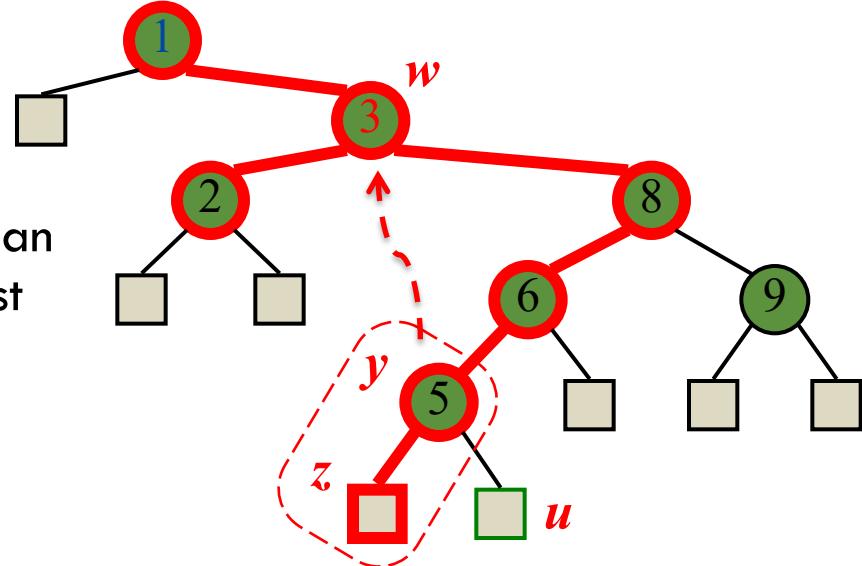
Deletion : Case 2

Suppose that the node w we want to remove has two internal children.

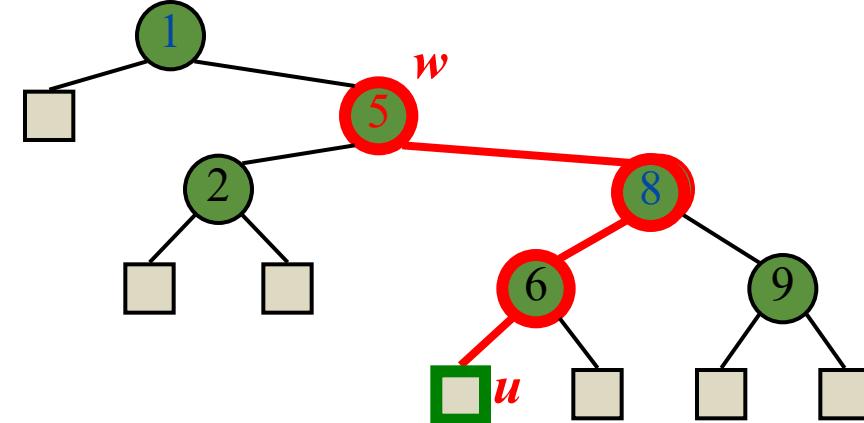
To remove w we

- find the internal node y following w in an inorder traversal (i.e., y has the smallest key among the right subtree under w)
- we copy the entry from y into node w
- we remove node y and its left child z , which must be external, using previous case

Example: remove(3)



This preserves the BST property



Deletion algorithm

```
def remove(k)
    w ← search(k, root)
    if w.isExternal() then
        # key not found
        return null
    else if w has at least one external child external z then
        remove z
        promote the other child of w to take w's place
        remove w
    else
        # y is leftmost internal node in the right subtree of w
        y ← immediate successor of w
        replace contents of w with entry from y
        remove(y)
```

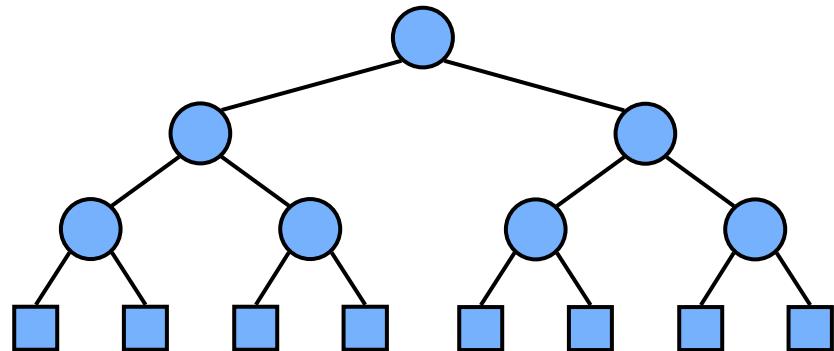
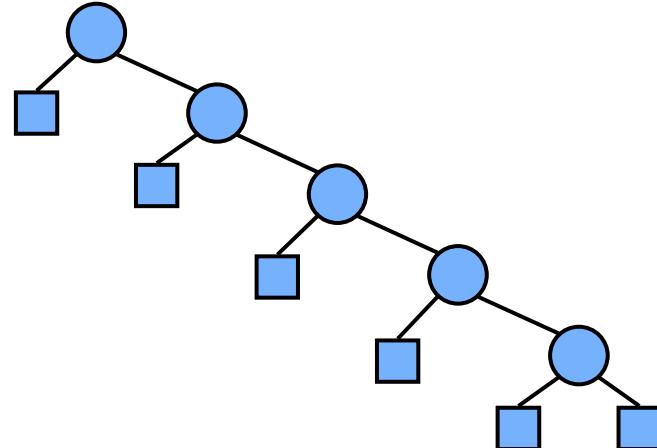
Complexity

Consider a map with n items implemented by means of a binary search tree of height h :

- the space used is $O(n)$
- get, put and remove take $O(h)$ time

The height h can be n in the worst case and $\log n$ in the best case.

Therefore the best one can hope is that tree operations take $O(\log n)$ time but in general we can only guarantee $O(n)$. But the former can be achieved with better insertion routines.



Duplicate key values in BST

Our definition says that keys are in strict increasing order

$$\text{key(left descendant)} < \text{key(node)} < \text{key(right descendant)}$$

This means that with this definition duplicate key values are not allowed (as needed when implementing Map)

However, it is possible to change it to allow duplicates. But that means additional complexity in the BST implementation:

- Allowing left descendants to be equal to the parent

$$\text{key(left descendant)} \leq \text{key(node)} < \text{key(right descendant)}$$

- Using a list to store duplicates

Range Queries

A range query is defined by two values k_1 and k_2 . We are to find all keys k stored in T such that $k_1 \leq k \leq k_2$

E.g., find all cars on eBay priced between 10K and 15K.

The algorithm is a restricted version of inorder traversal. When at node v :

- if $\text{key}(v) < k_1$: Recursively search right subtree
- if $k_1 \leq \text{key}(v) \leq k_2$: Recursively search left subtree, add v to range output, search right subtree
- if $k_2 < \text{key}(v)$: Recursively search left subtree

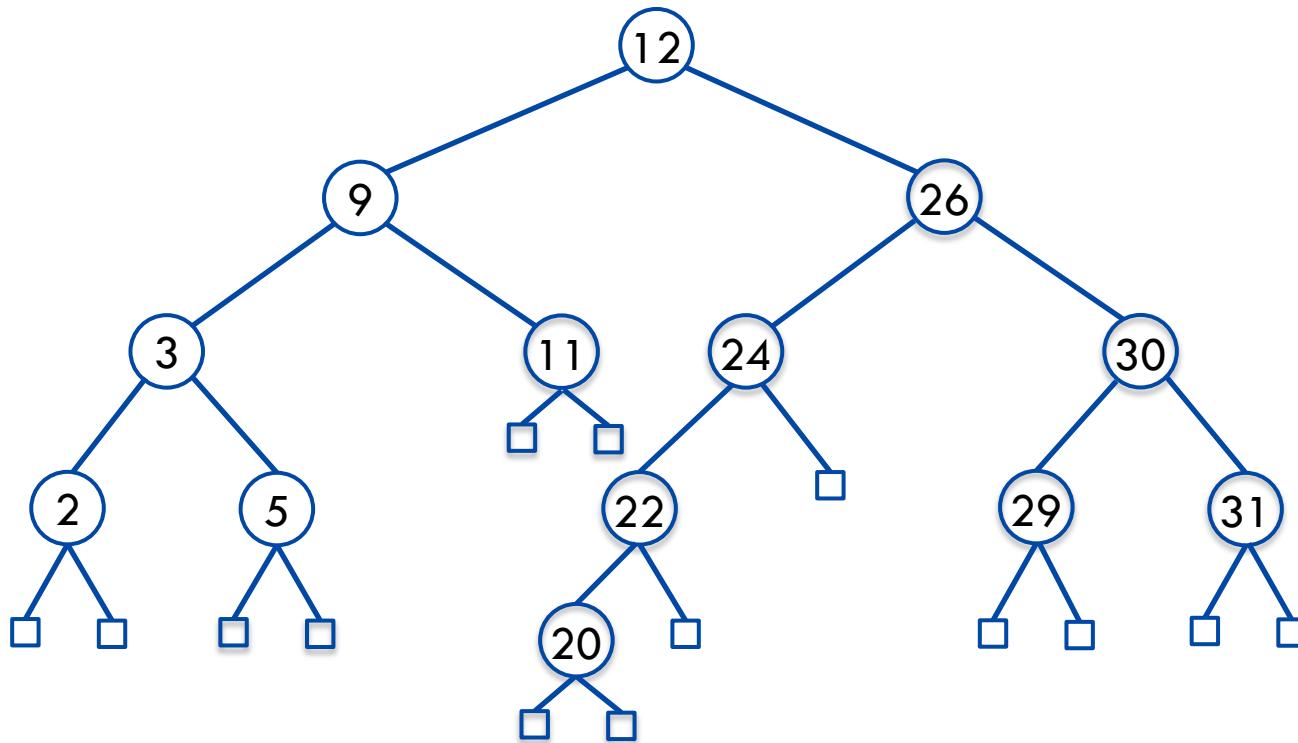
Pseudo-code

```
def range_search(T, k1, k2)
    output ← []
    range(T.root, k1, k2)
```

```
def range(v, k1, k2)
    if v is external then
        return null
    if key(v) > k2 then
        range(v.left, k1, k2)
    else if key(v) < k1 then
        range(v.right, k1, k2)
    else
        range(v.left, k1, k2)
        output.append(v)
        range(v.right, k1, k2)
```

Range queries

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$

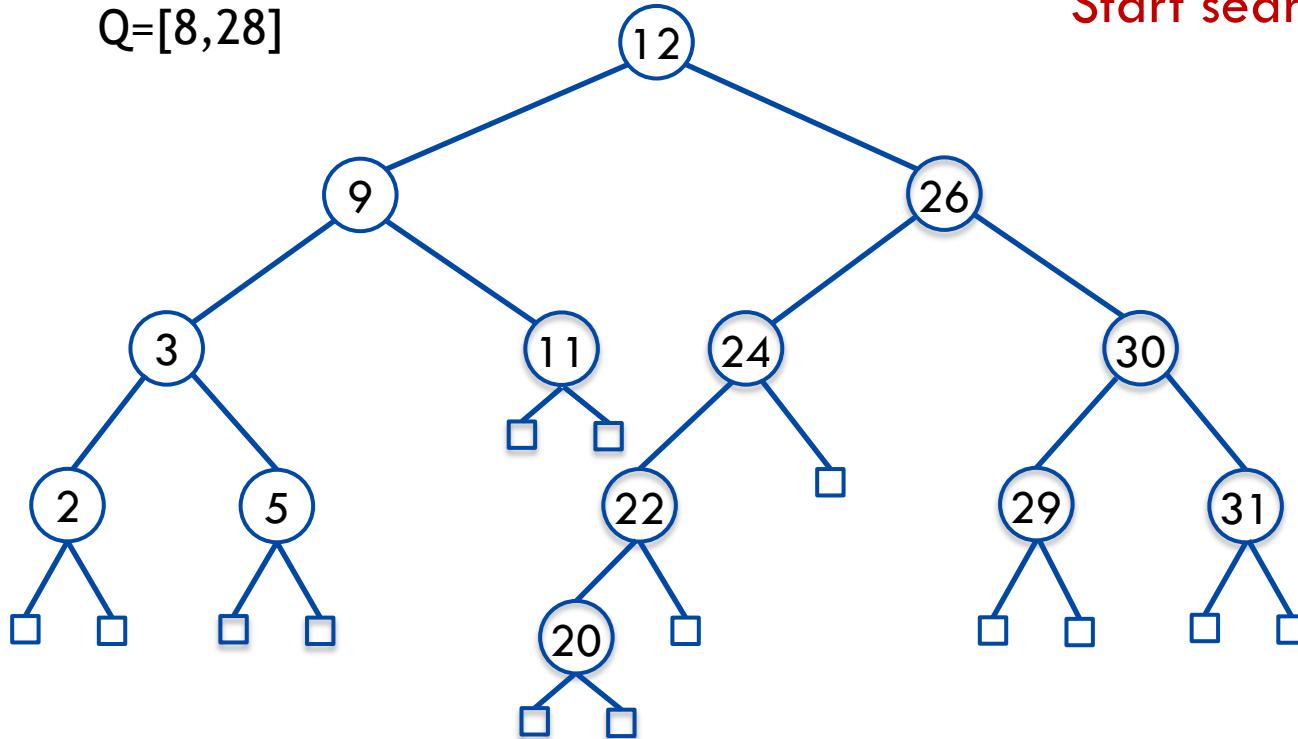


Range queries

$$S=\{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$$

$$Q=[8,28]$$

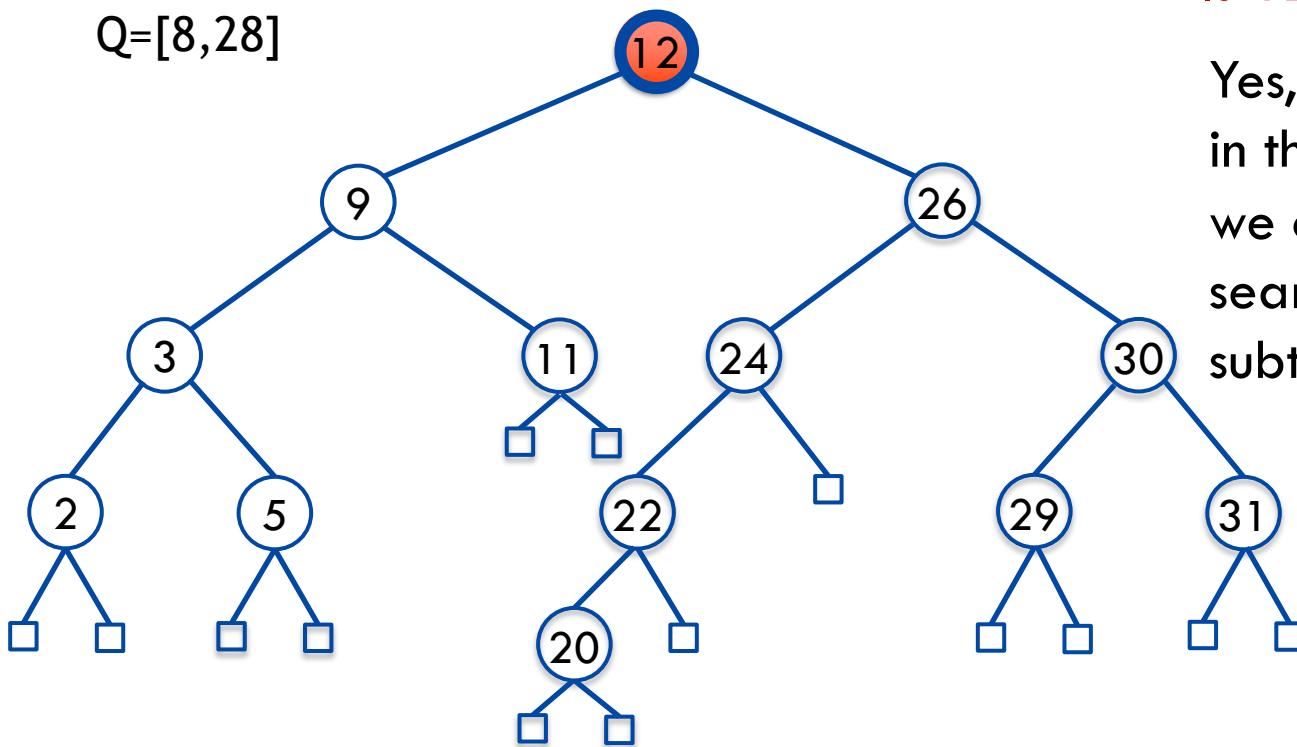
Start search



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

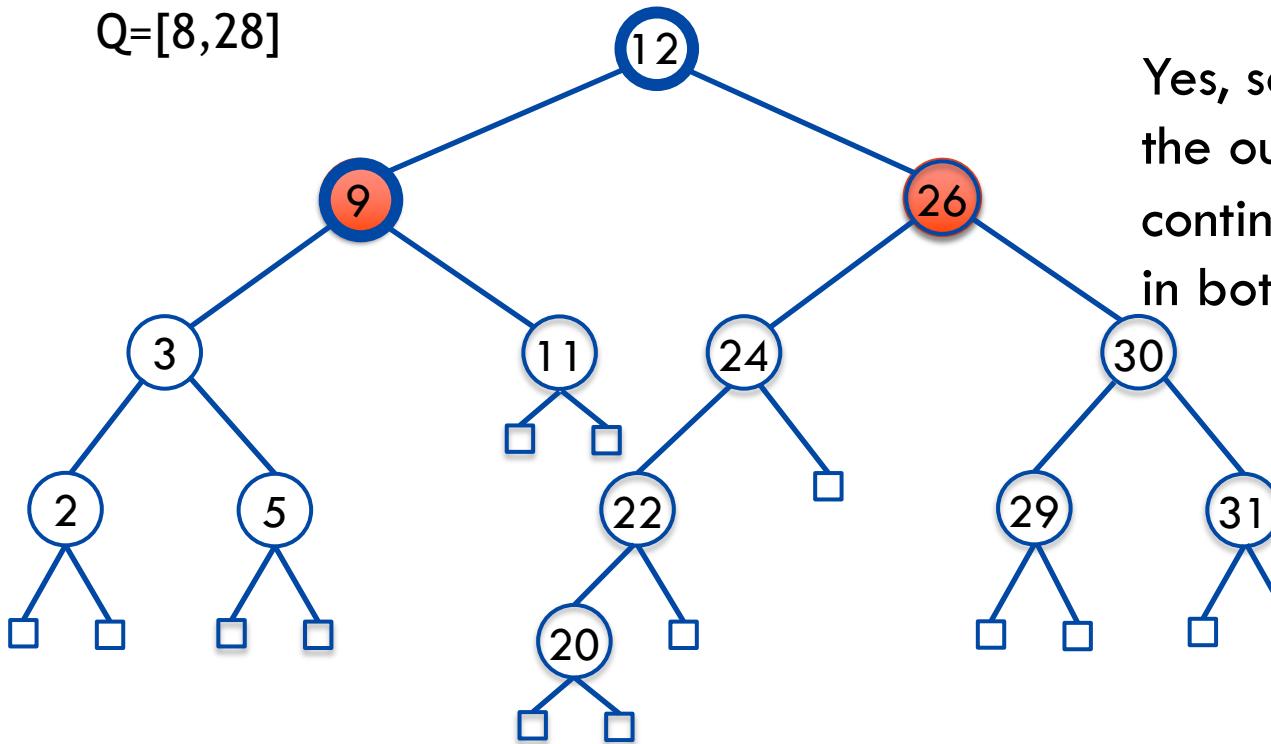
$Q = [8, 28]$



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



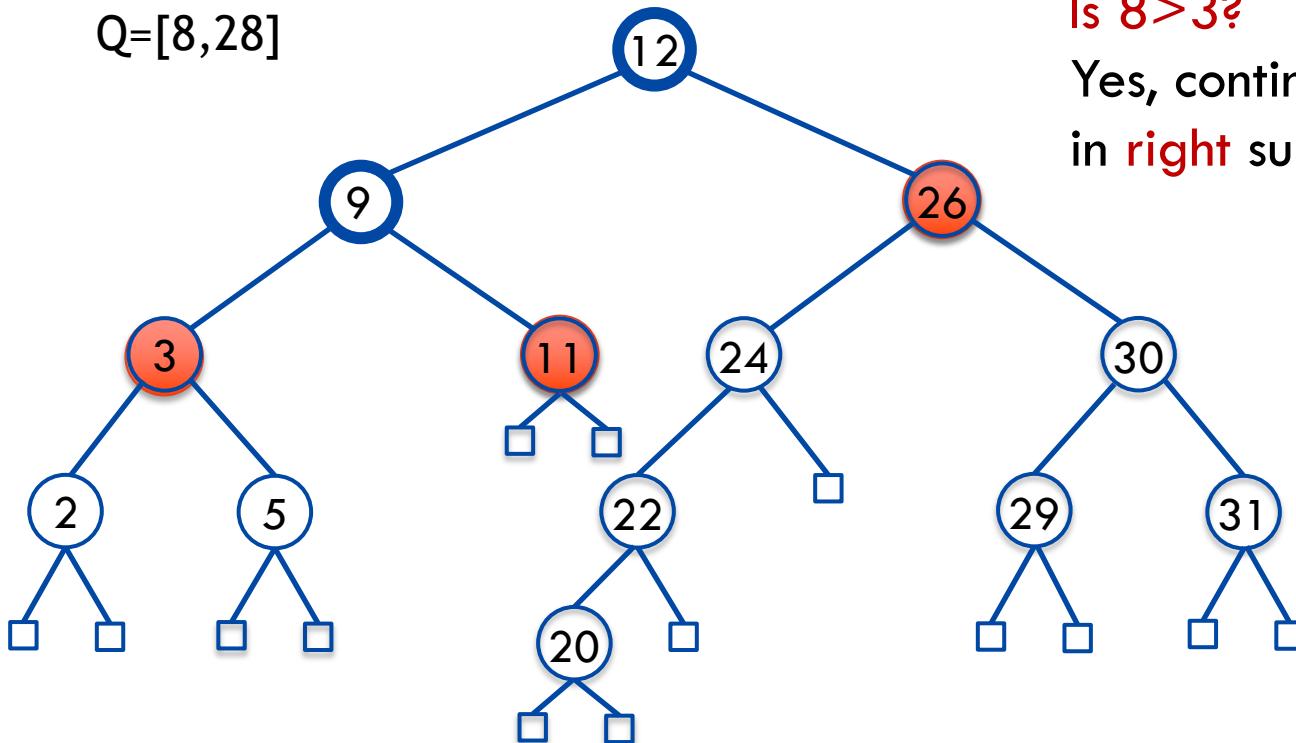
Is 9 in Q?

Yes, so 9 will be in the output and we continue the search in both subtrees

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

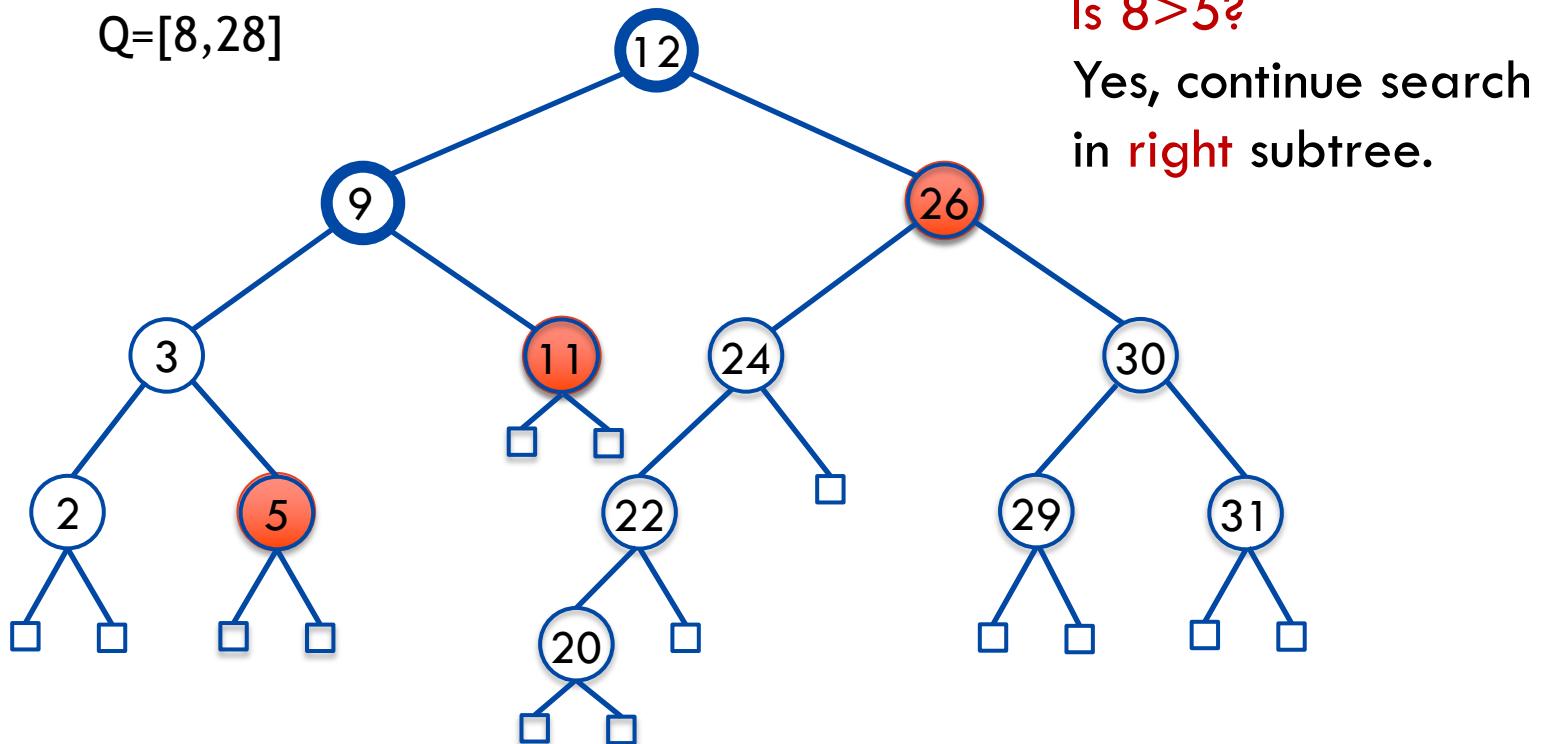


Is $8 > 3$?
Yes, continue search
in **right** subtree.

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

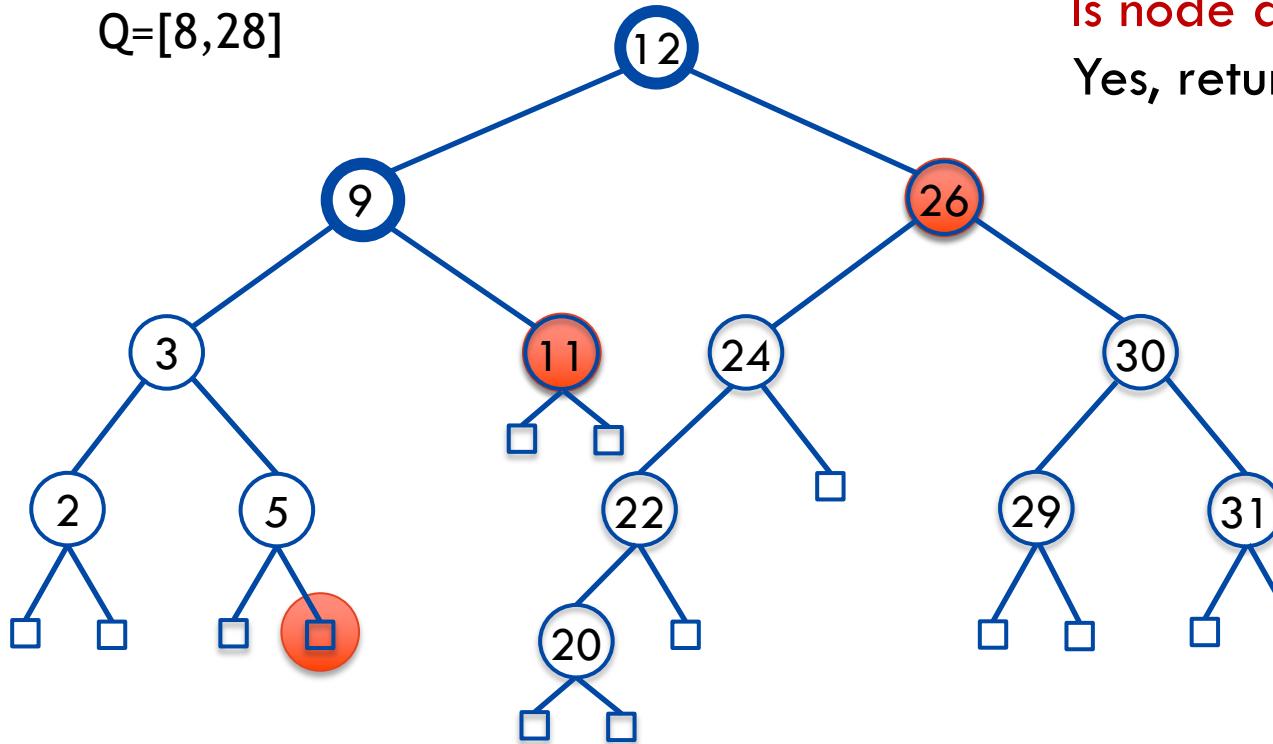


Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

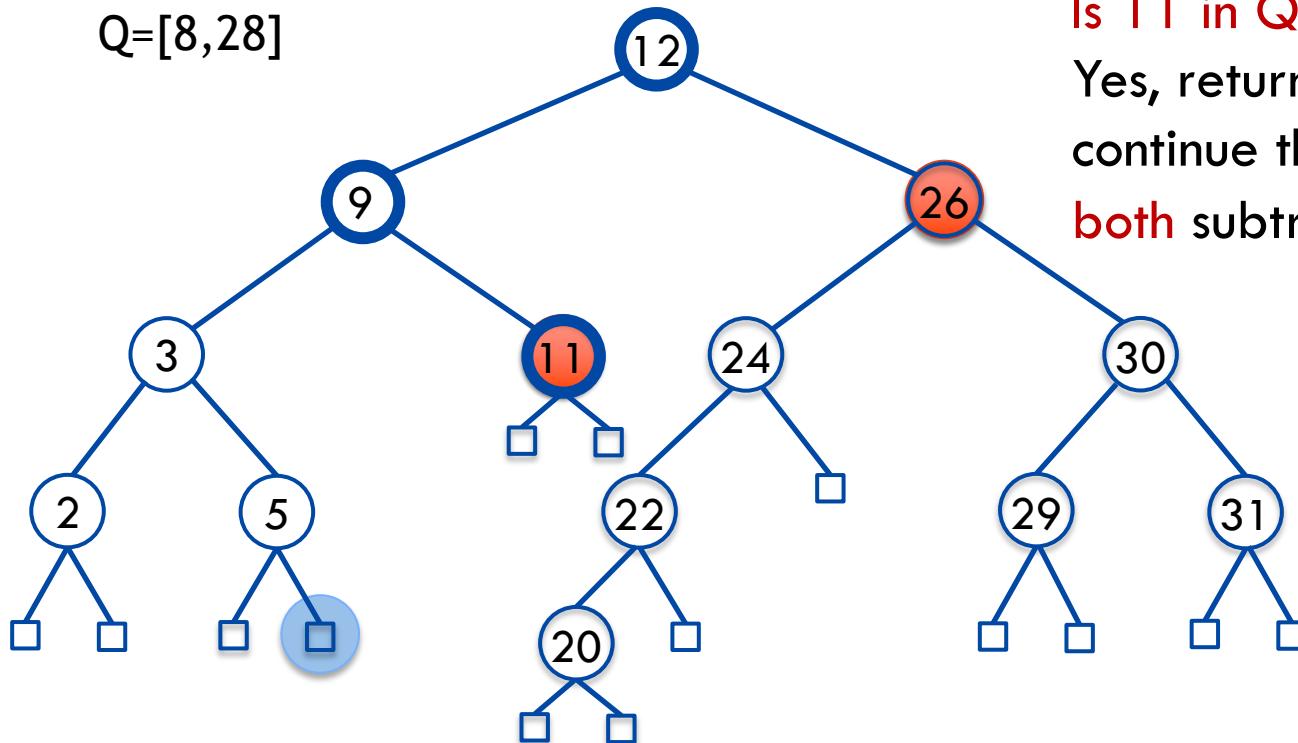
Is node a leaf?
Yes, return \emptyset .



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



Is 11 in Q?

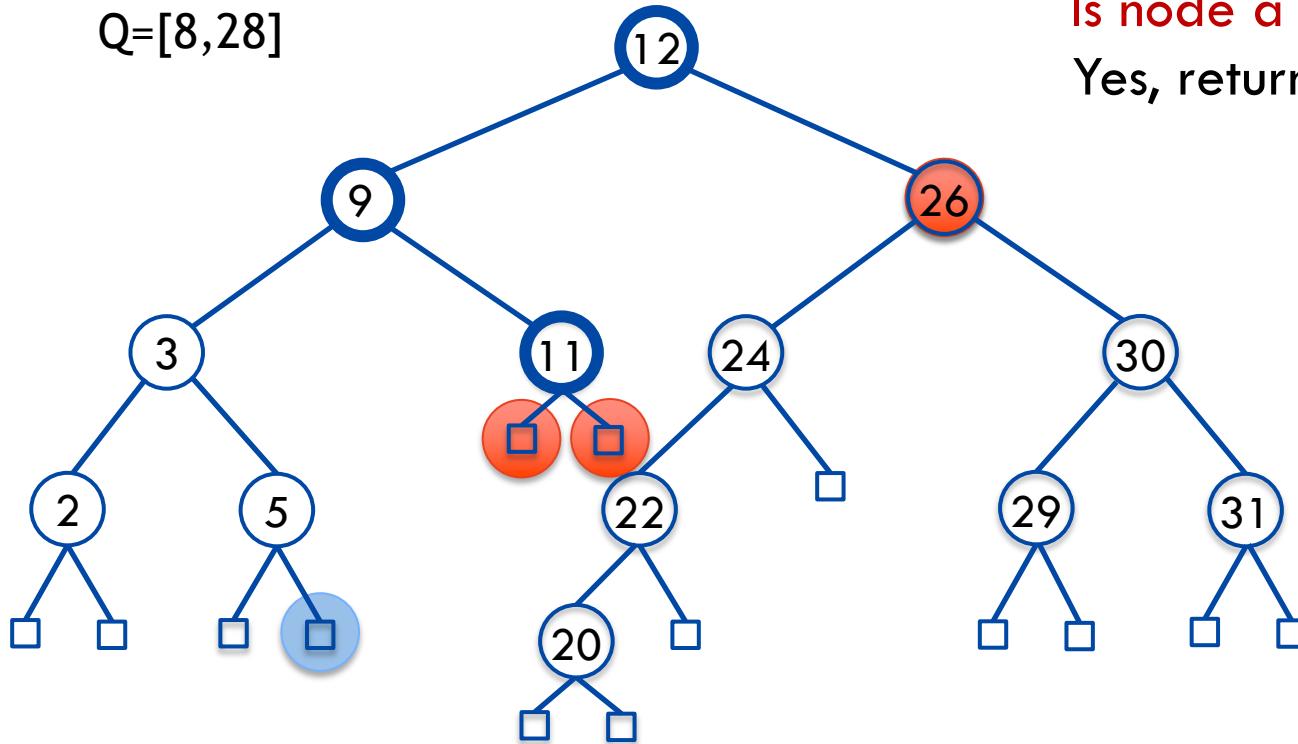
Yes, return 11 and continue the search in both subtrees.

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

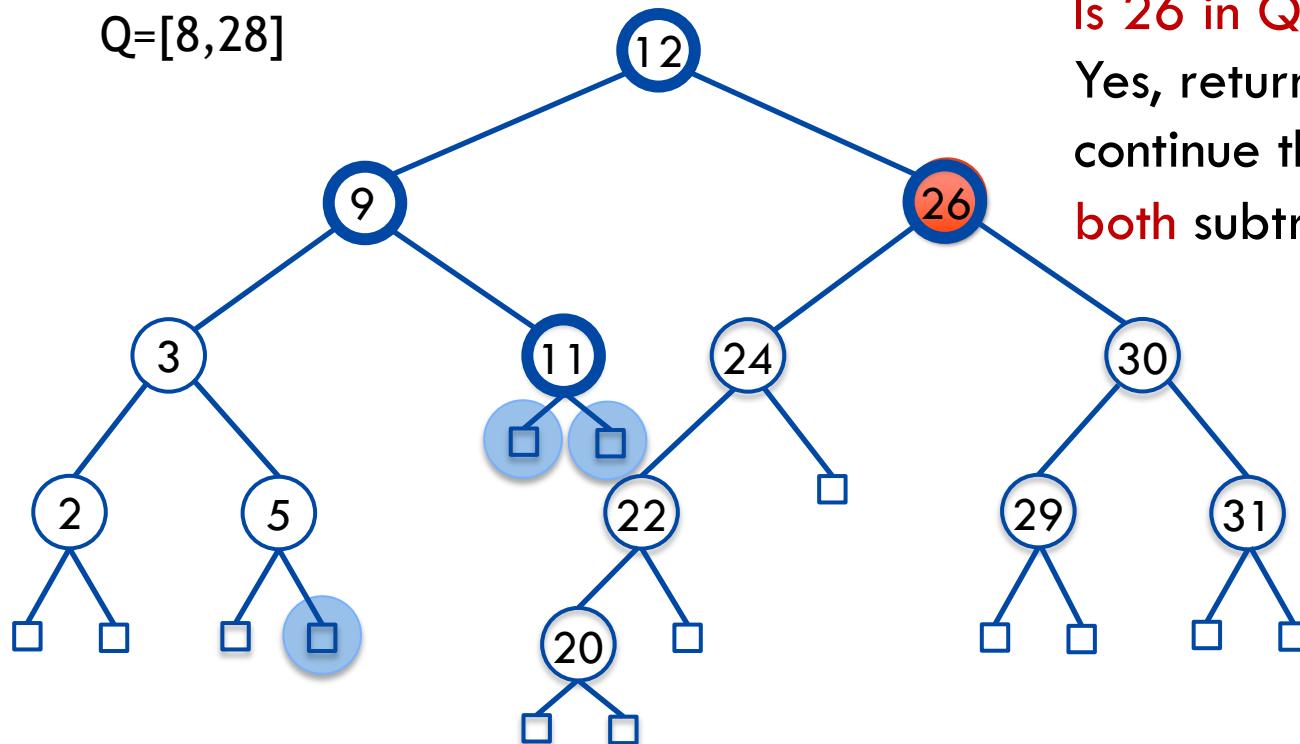
Is node a leaf ($\times 2$)?
Yes, return \emptyset .



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



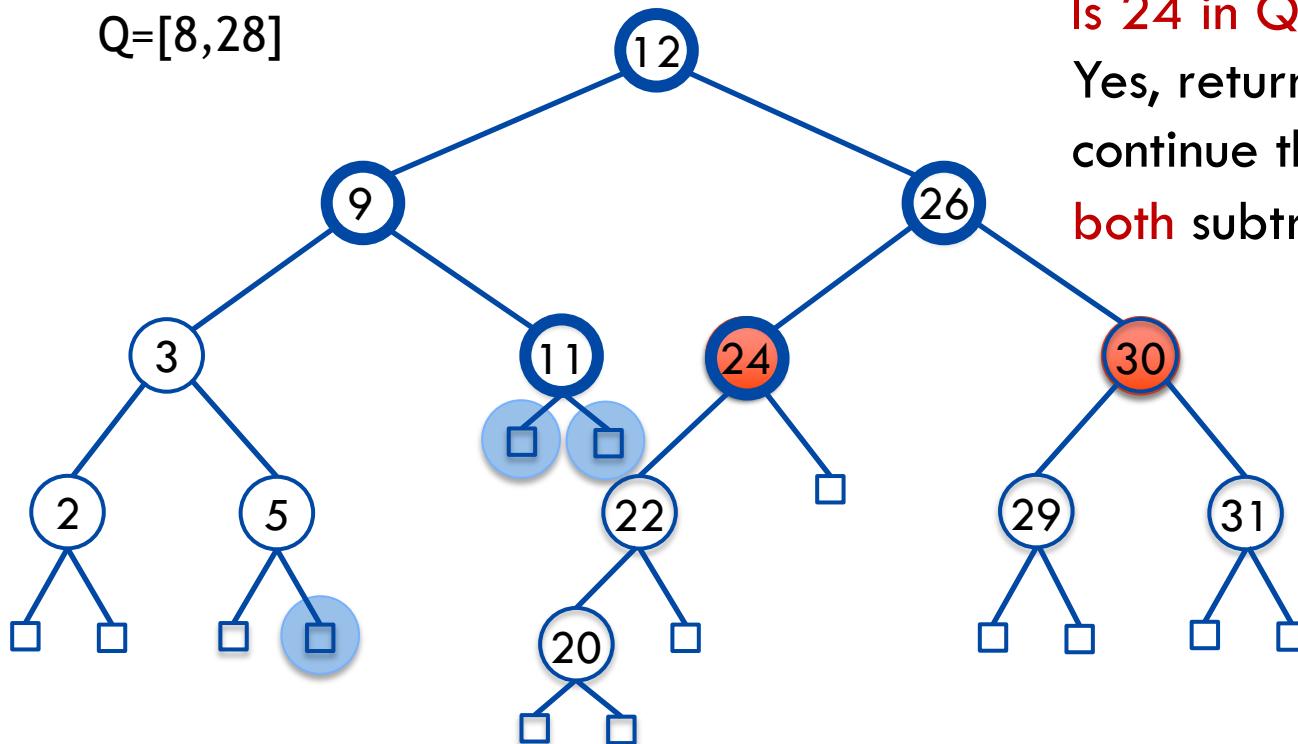
Is 26 in Q?

Yes, return 26 and continue the search in both subtrees.

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



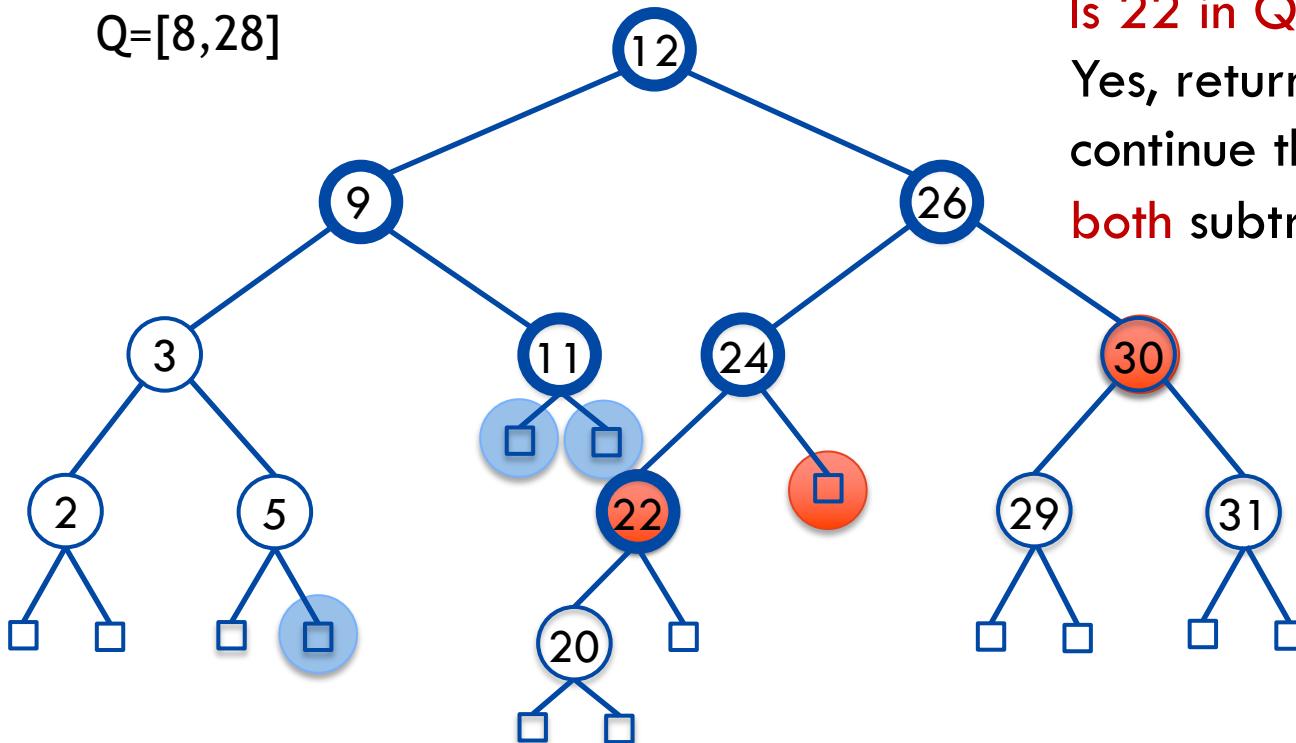
Is 24 in Q?

Yes, return 24 and
continue the search in
both subtrees.

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

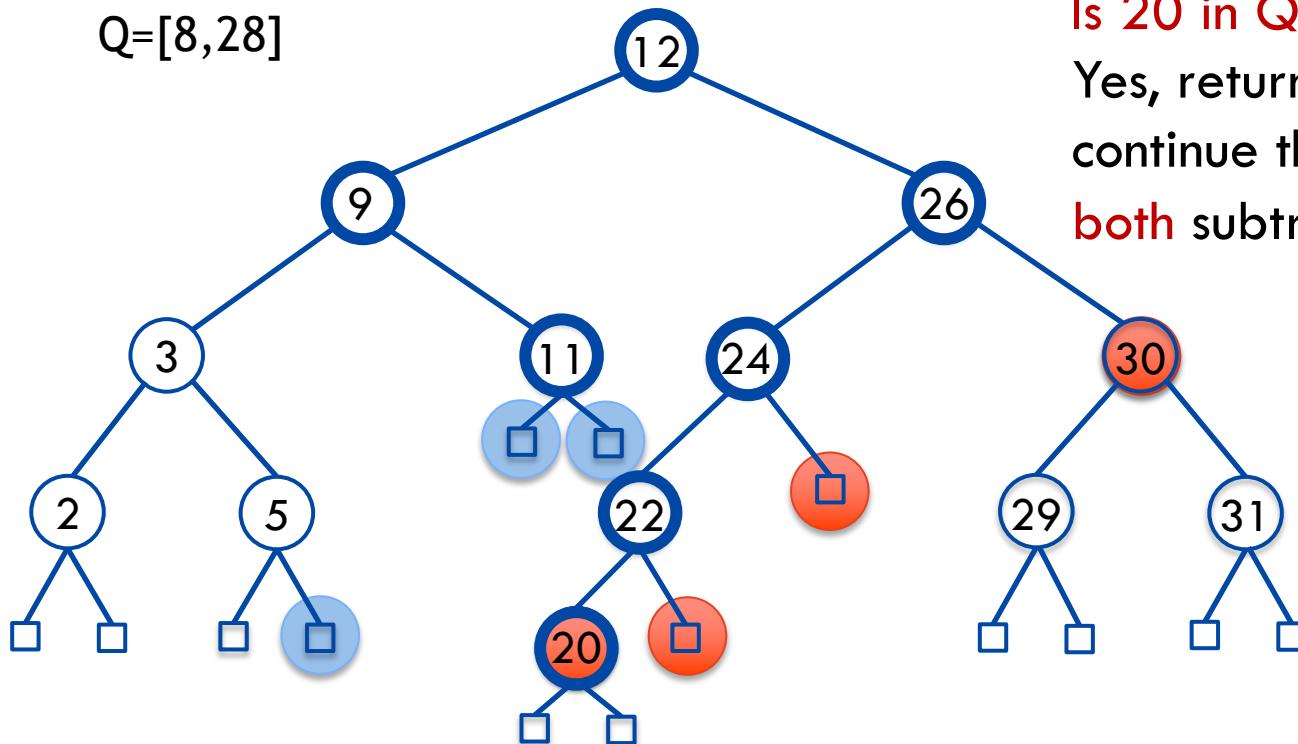
$Q = [8, 28]$



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



Is 20 in Q?

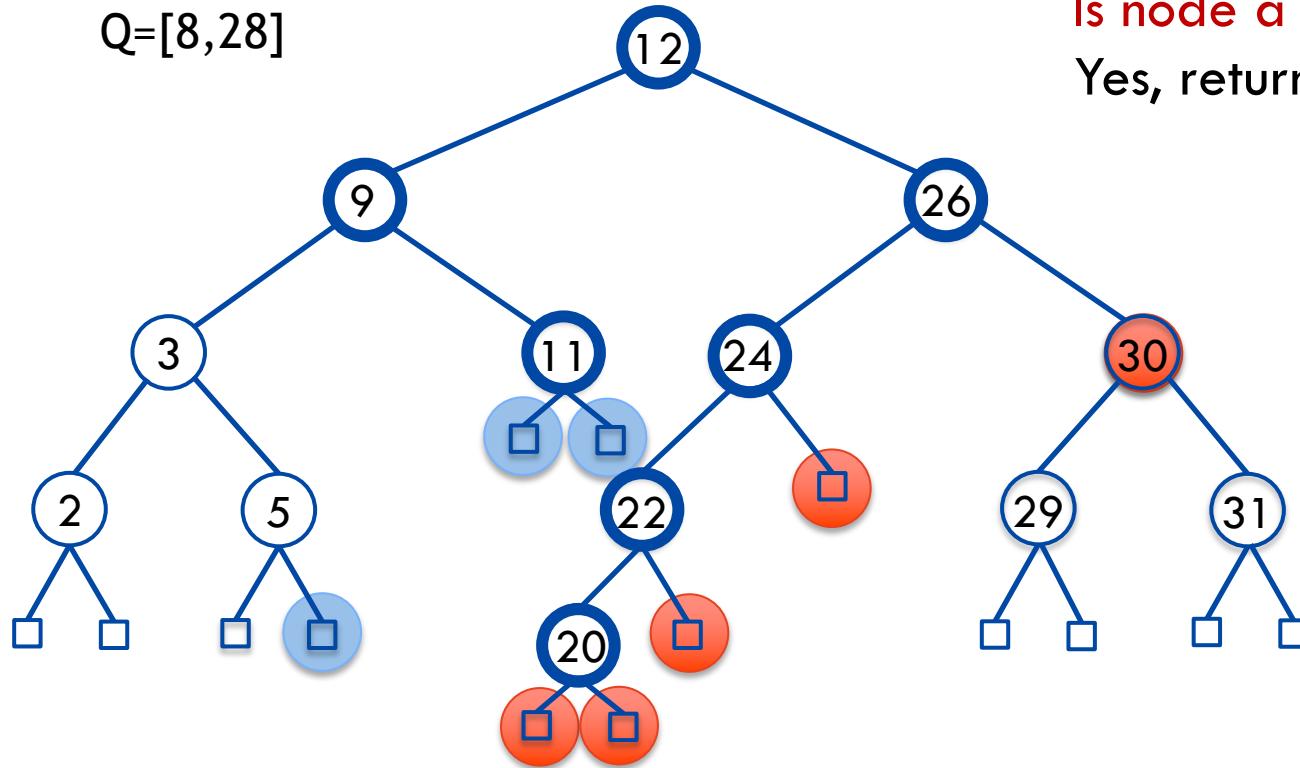
Yes, return 20 and
continue the search in
both subtrees.

Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

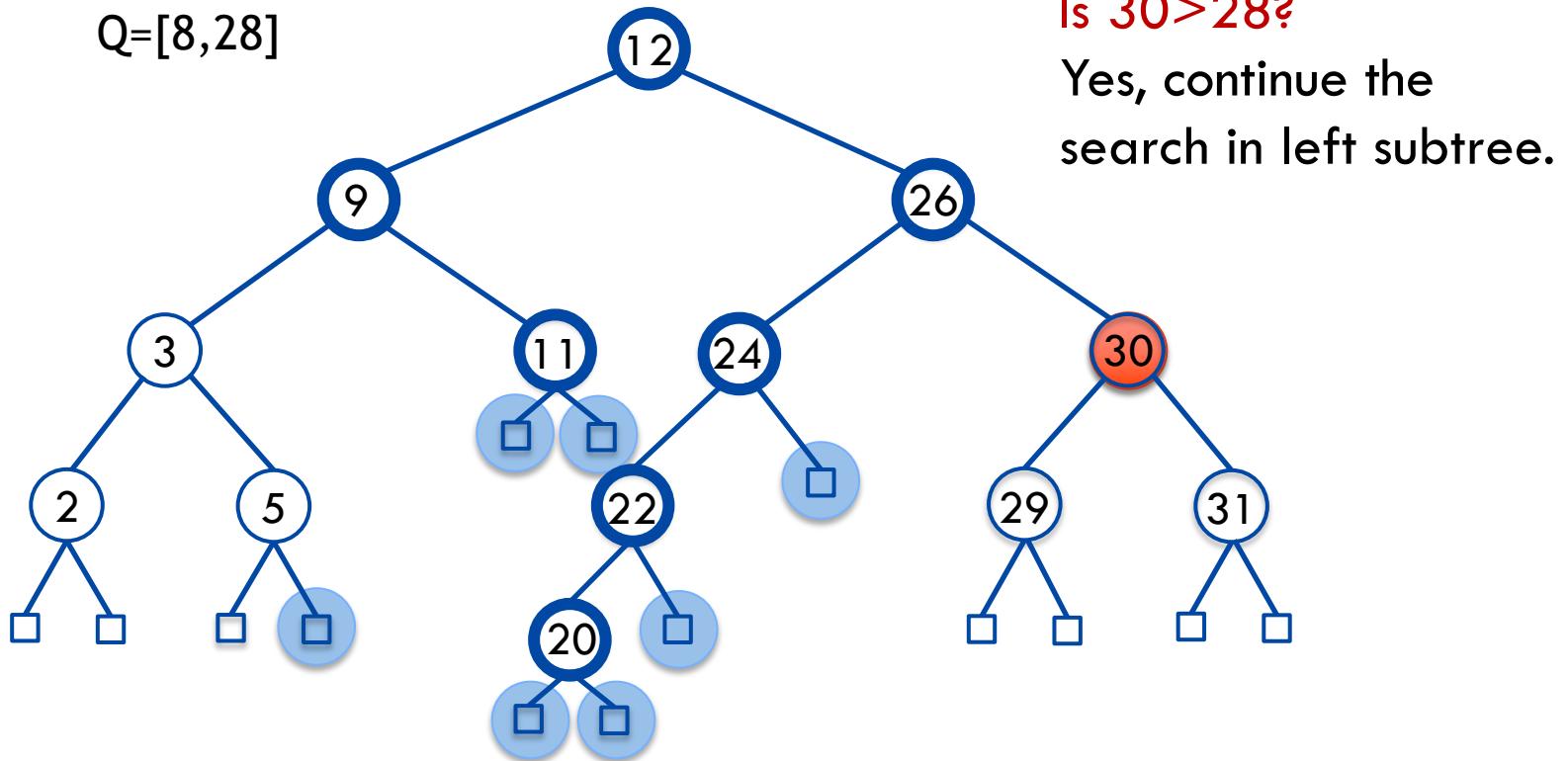
Is node a leaf ($\times 4$)?
Yes, return \emptyset .



Range queries

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$

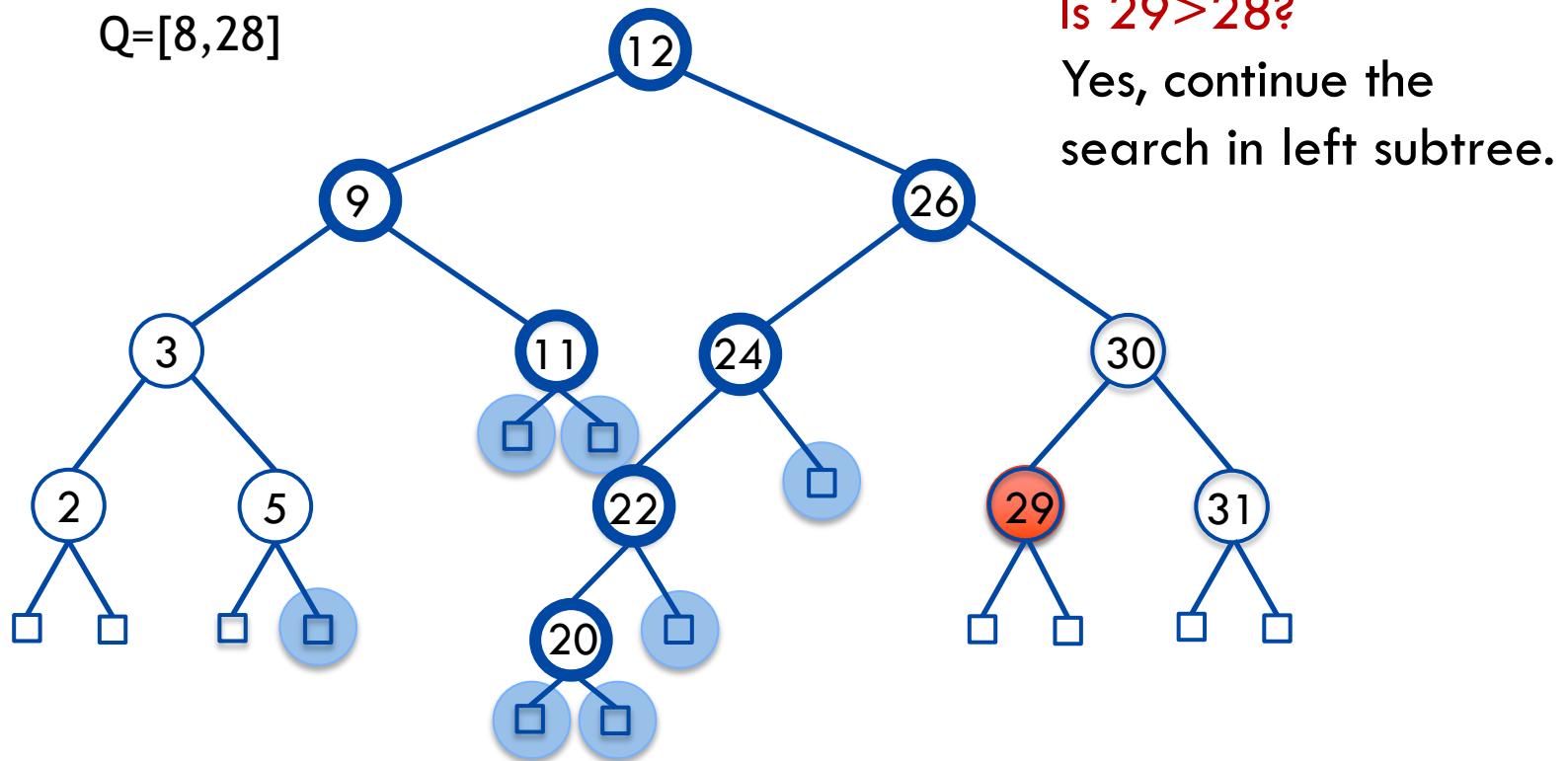
$Q=[8,28]$



Range queries

$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$

$Q=[8,28]$

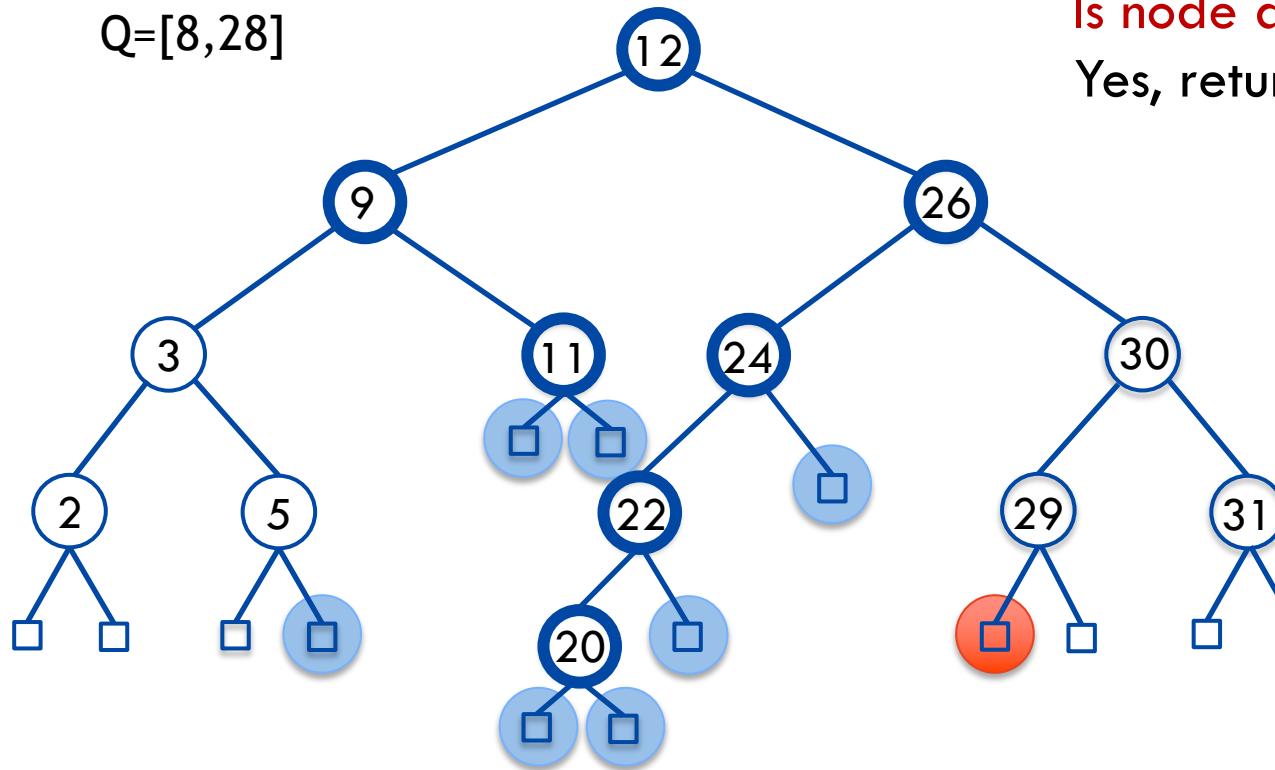


Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$

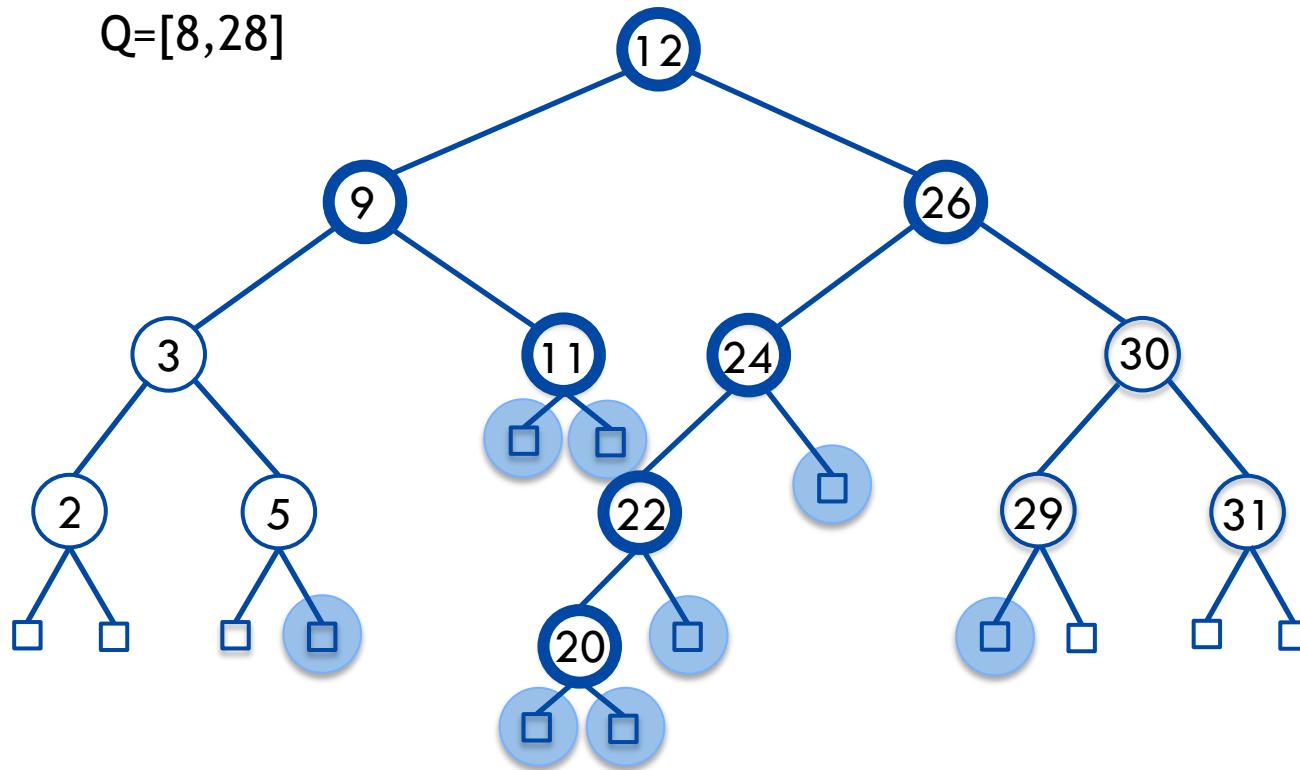
Is node a leaf?
Yes, return \emptyset .



Range queries

$S = \{2, 3, 5, 9, 11, 12, 20, 22, 24, 26, 29, 30, 31\}$

$Q = [8, 28]$



Performance

Let P_1 and P_2 be the binary search paths to k_1 and k_2

We say a node v is a:

- boundary node if v in P_1 or P_2
- inside node if $\text{key}(v)$ in $[k_1, k_2]$ but not in P_1 or P_2
- outside node if $\text{key}(v)$ not in $[k_1, k_2]$ but not in P_1 or P_2

The algorithm only visits boundary and inside nodes and

- $|\text{inside nodes}| \leq |\text{output}|$
- $|\text{boundary node}| \leq 2 * \text{tree height}$

Therefore, since we only spend $O(1)$ time per node we visit. The total running time of range search is $O(|\text{output}| + \text{tree height})$

Maintaining a balanced BST

We have seen operations on BSTs that take $O(\text{height})$ time to run. Unfortunately, the standard insertion implementation can lead to a tree with height $n-1$ (e.g., if we insert in sorted order)

In the rest of today's lecture we will cover much more sophisticated algorithms that maintain a BST with height $O(\log n)$ at all times by rebalancing the tree with simple local transformations

This directly translates into $O(\log n)$ performance for searching

Rank-balanced Trees

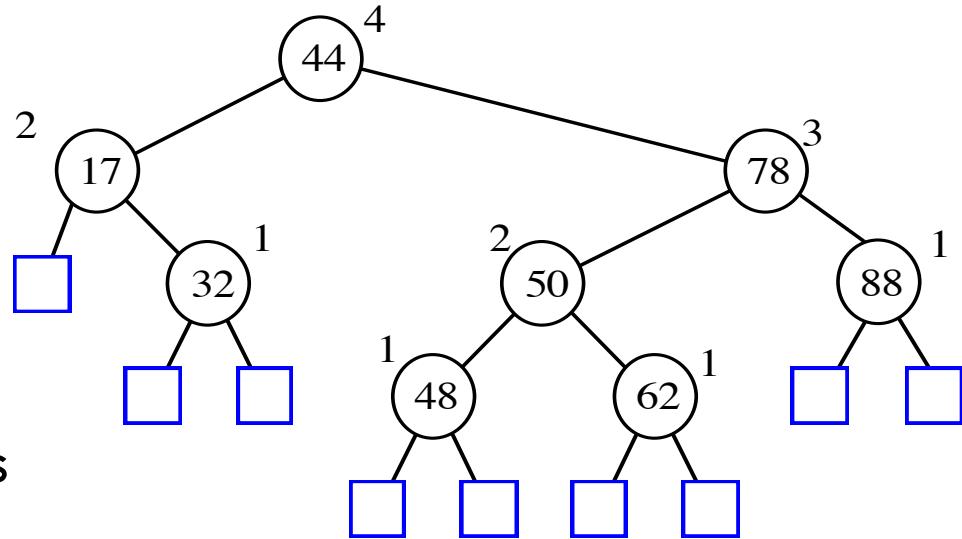
A family of balanced BST implementations that use the idea of keeping a “rank” for every node, where $r(v)$ acts as a proxy measure of the size of the subtree rooted at v

Rank-balanced trees aim to reduce the discrepancy between the ranks of the left and right subtrees:

- AVL Trees (now)
- Red-Black Trees (book)

AVL Tree Definition

AVL trees are rank-balanced trees, where $r(v)$ is its height of the subtree rooted at v



Balance constraint: The ranks of the two children of every internal node differ by at most 1.

Height of an AVL Tree

Fact: The height of an AVL tree storing n keys is at most $O(\log n)$.

Proof (by induction):

- Let $N(h)$ be the minimum number of keys of an AVL tree of height h .
- We easily see that $N(1) = 1$ and $N(2) = 2$
- Clearly $N(h) > N(h-1)$ for any $h \geq 2$
- For $h > 2$, the smallest AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and another of height at least $h-2$:

$$N(h) \geq 1 + N(h-1) + N(h-2) > 2 N(h-2)$$

- By induction we can show that for h even

$$N(h) \geq 2^{h/2}$$

- Taking logarithms: $h < 2 \log_2 N(h)$
- Thus the height of an AVL tree on n nodes is $O(\log n)$

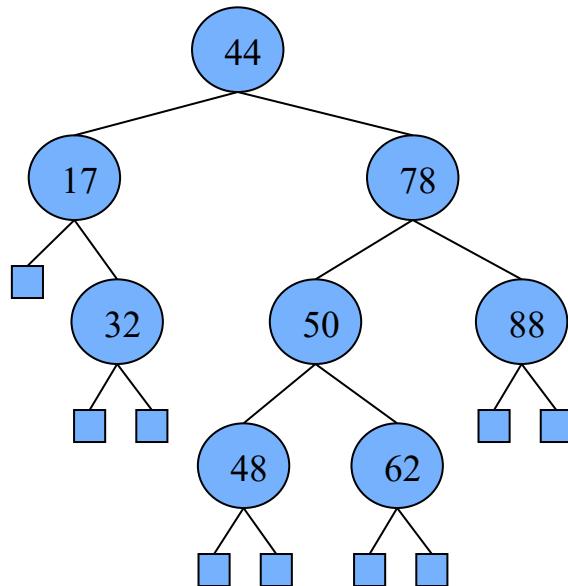
Insertion in AVL trees

Suppose we are to insert a key k into our tree:

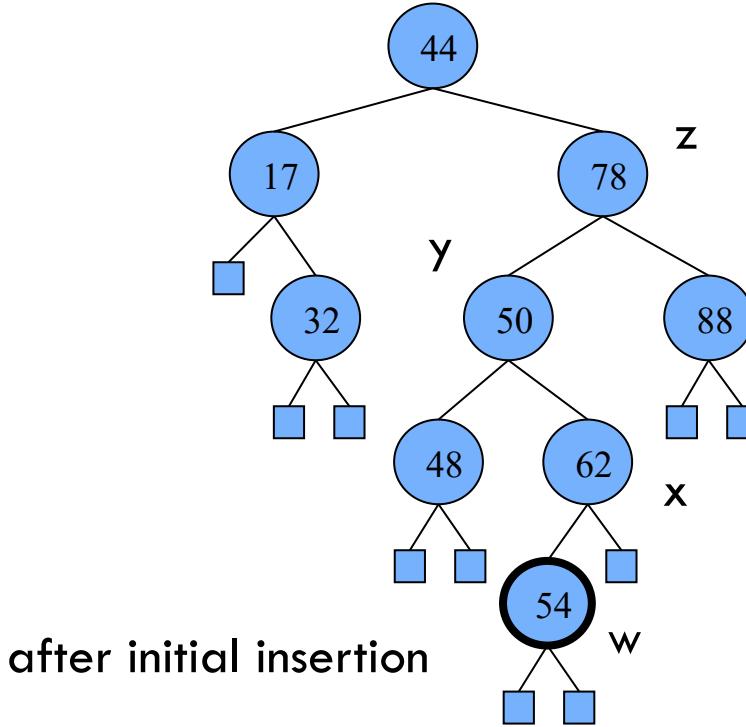
1. If k is in the tree, search for k ends at node holding k
There is nothing to do so tree structure does not change
2. If k is not in the tree, search for k ends at external node w .
Make this be a new internal node containing key k
3. The new tree has BST property, but it may not have AVL balance property at some ancestor of w since
 - some ancestors of w may have increased their height by 1
 - every node that is not an ancestor of w hasn't changed its height
4. We use rotations to re-arrange tree to re-establish AVL property, while keeping BST property

Re-establishing AVL property

- Let w be location of newly inserted node
- Let z be *lowest* ancestor of w , whose children heights differ by 2
- Let y be the child of z that is ancestor of w (taller child of z)
- Let x be child of y that is ancestor of w

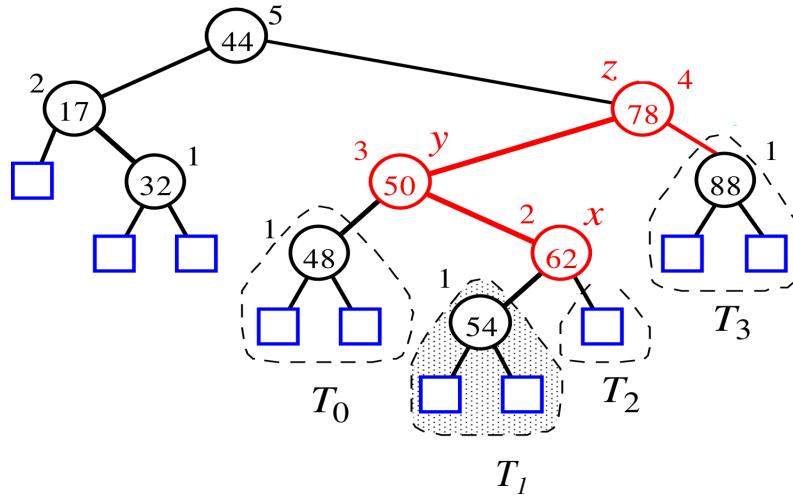


before inserting 54



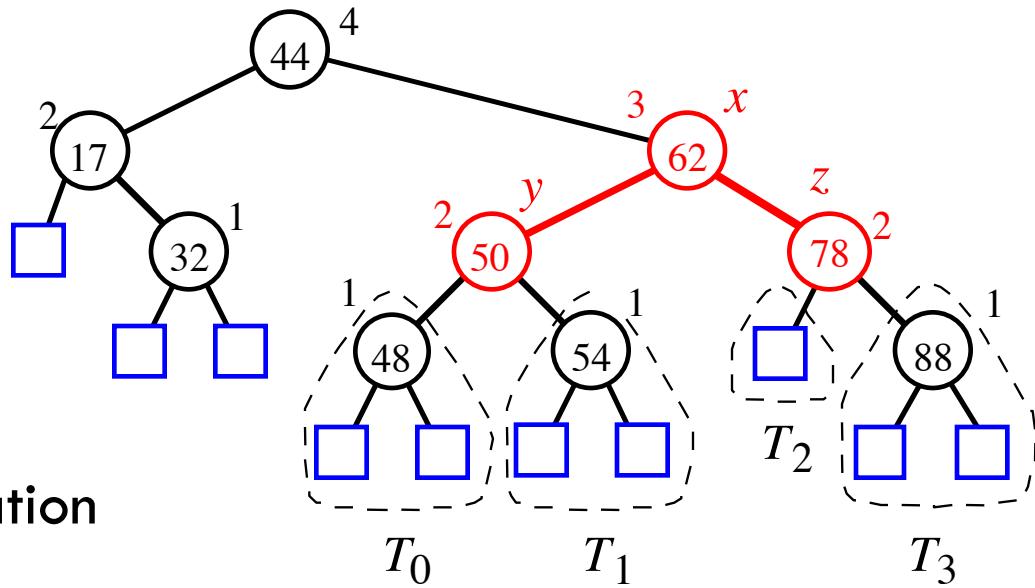
after initial insertion

Re-establishing AVL property



If tree does not have
AVL property, do a trinode
restructure at x, y, z

It can be argued that tree
has AVL property after operation



Augmenting BST with a height attribute

But how do we know the height of each node? If we had to compute this from scratch it would take $O(n)$ time

Therefore, we need to have this pre-computed and update the height value after each insertion and rebalancing operation:

- After we create a node w , we should set its height to be 1, and then update the height of its ancestors.
- After we rotate (z, y, x) we should update their height and that of their ancestors.

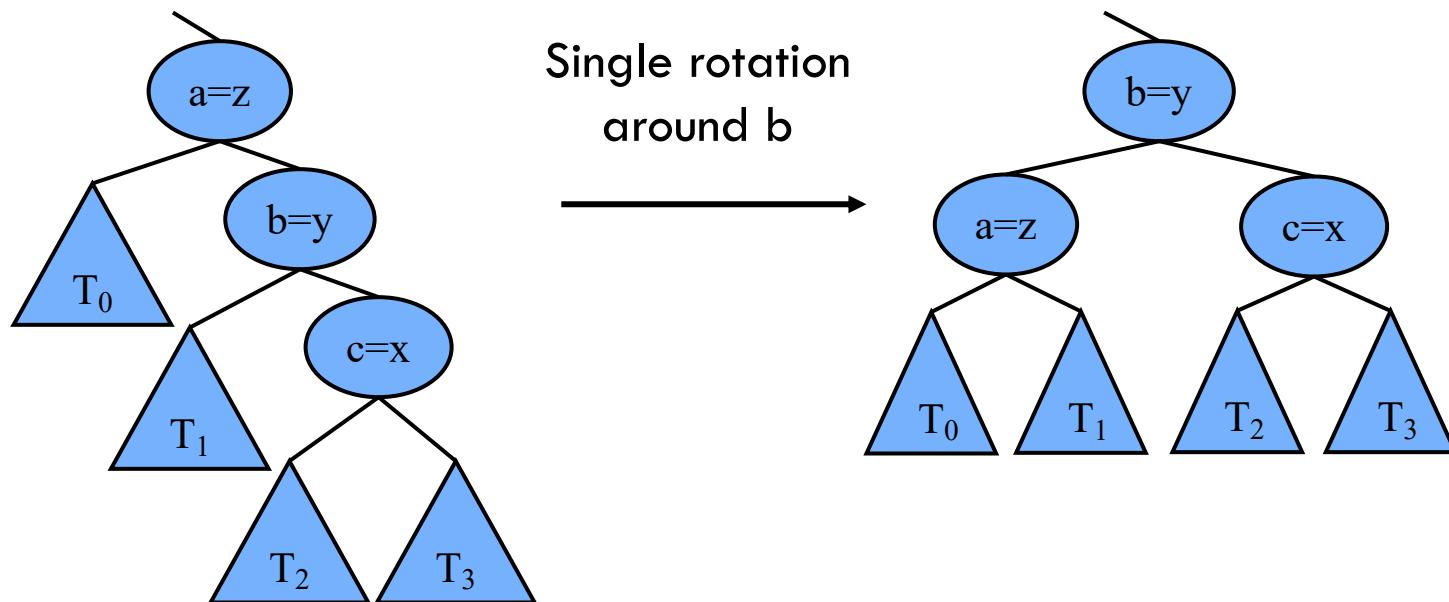
Thus, we can maintain the height only using $O(h)$ work per insert

Improving Balance: Trinode Restructuring

Let x, y, z be nodes such that x is a child of y and y is a child of z .

Let a, b, c be the inorder listing of x, y, z

Perform the rotations so as to make b the topmost node of the three.

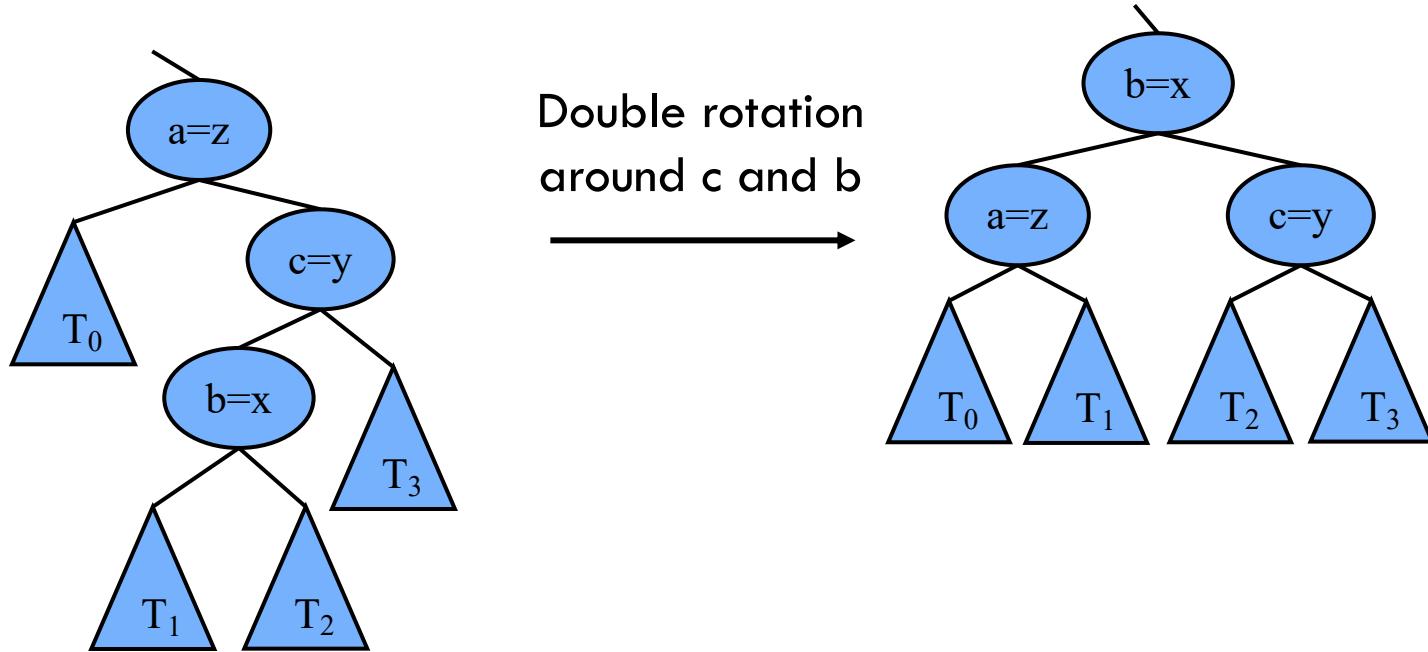


Improving Balance: Trinode Restructuring

Let x, y, z be nodes such that x is a child of y and y is a child of z .

Let a, b, c be the inorder listing of x, y, z

Perform the rotations so as to make b the topmost node of the three.



Pseudo-code

The algorithm for doing a trinode restructuring, which is used, possibly repeatedly, to restore balance after an insertion or deletion.

def restructure(*x*):

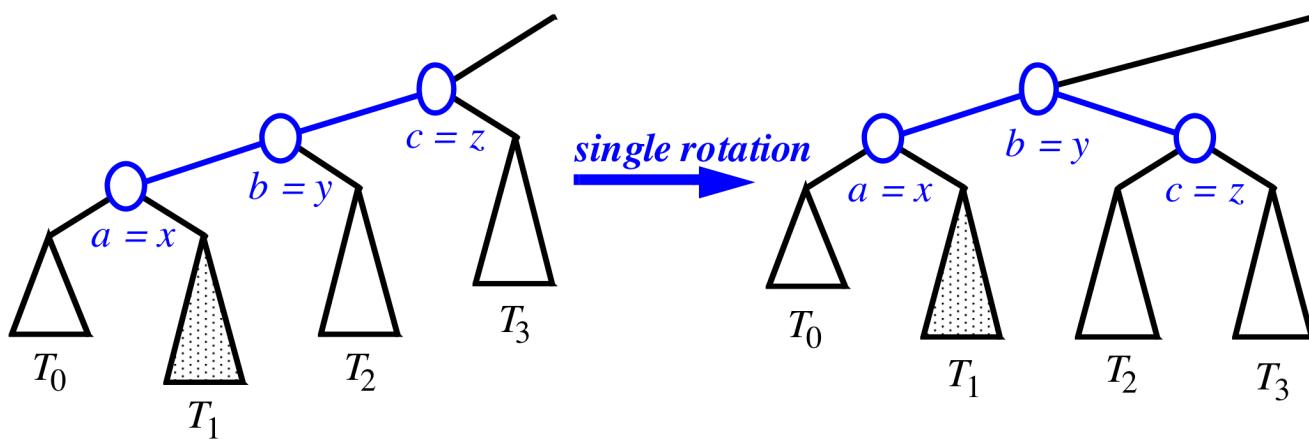
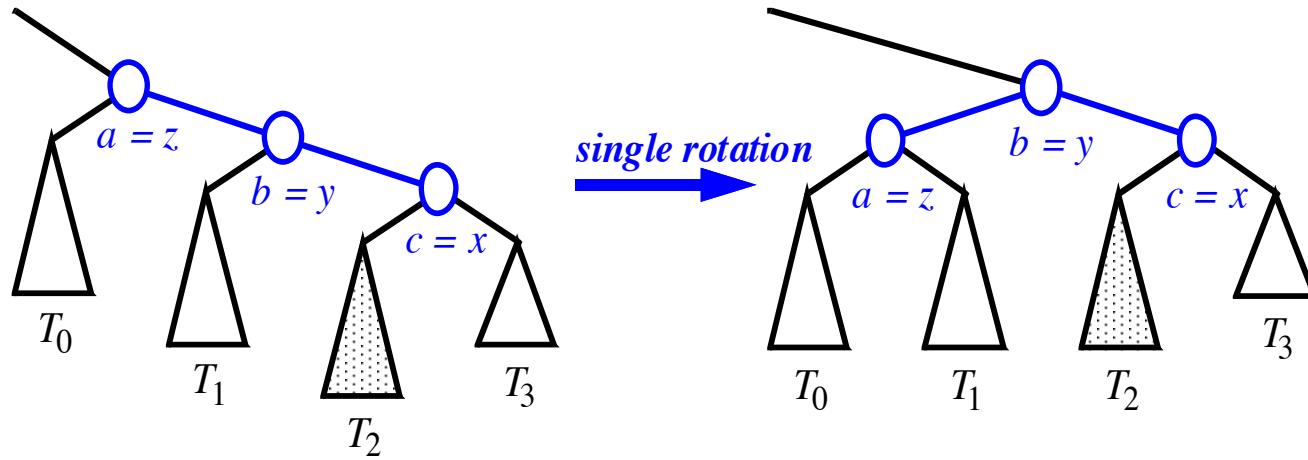
input A node *x* of a binary search tree *T* that has both a parent *y* and a grandparent *z*

output Tree *T* after a trinode restructuring (which corresponds to a single or double rotation) involving nodes *x*, *y*, and *z*

1. Let (a, b, c) be the left-to-right (inorder) listing of the nodes *x*, *y*, and *z*, and let (T_0, T_1, T_2, T_3) be the left-to-right (inorder) listing of the four subtrees of *x*, *y*, and *z* that are not rooted at *x*, *y*, and *z*.
2. Replace the subtree with a new subtree rooted at *b*.
3. Let *a* be the left child of *b* and let T_0 and T_1 be the left and right subtrees of *a*
4. Let *c* be the left child of *b* and let T_2 and T_3 be the left and right subtrees of *c*
5. Recalculate the heights of *a*, *b*, and *c* from the corresponding values stored at their children
6. **return** *b*

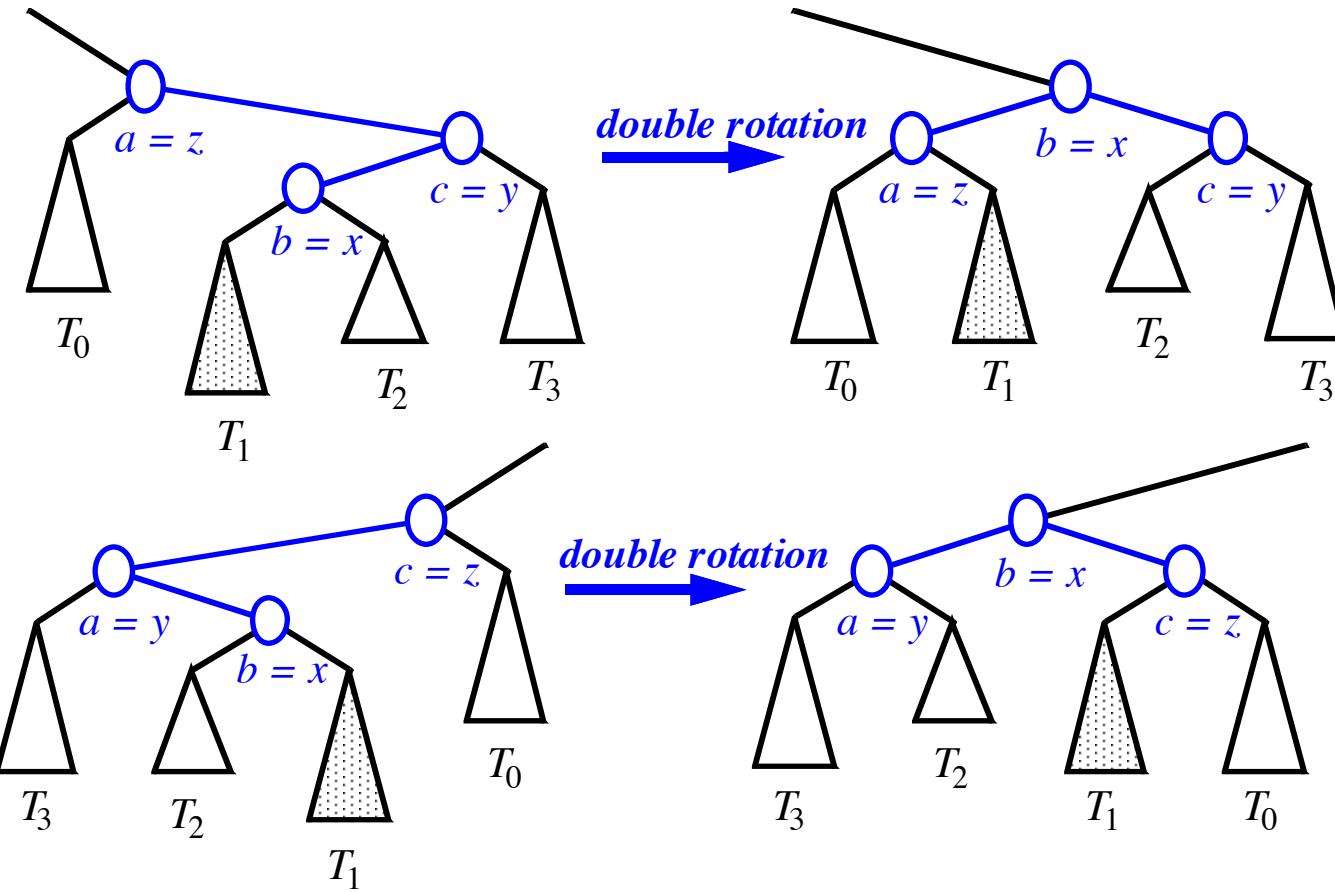
Trinode Restructuring (when done by Single Rotation)

Single Rotations:



Trinode Restructuring (when done by Double Rotation)

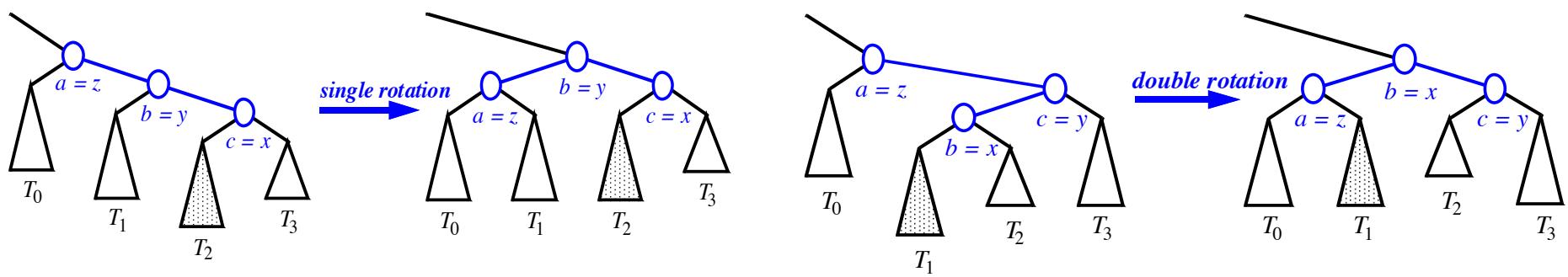
Double rotations:



Performance

Assume we are given a reference to the node x where we are performing a trinode restructure and that the binary search tree is represented using nodes and pointers to parent, left and right children

A single or double rotation takes $O(1)$ time, because it involves updating $O(1)$ pointers.



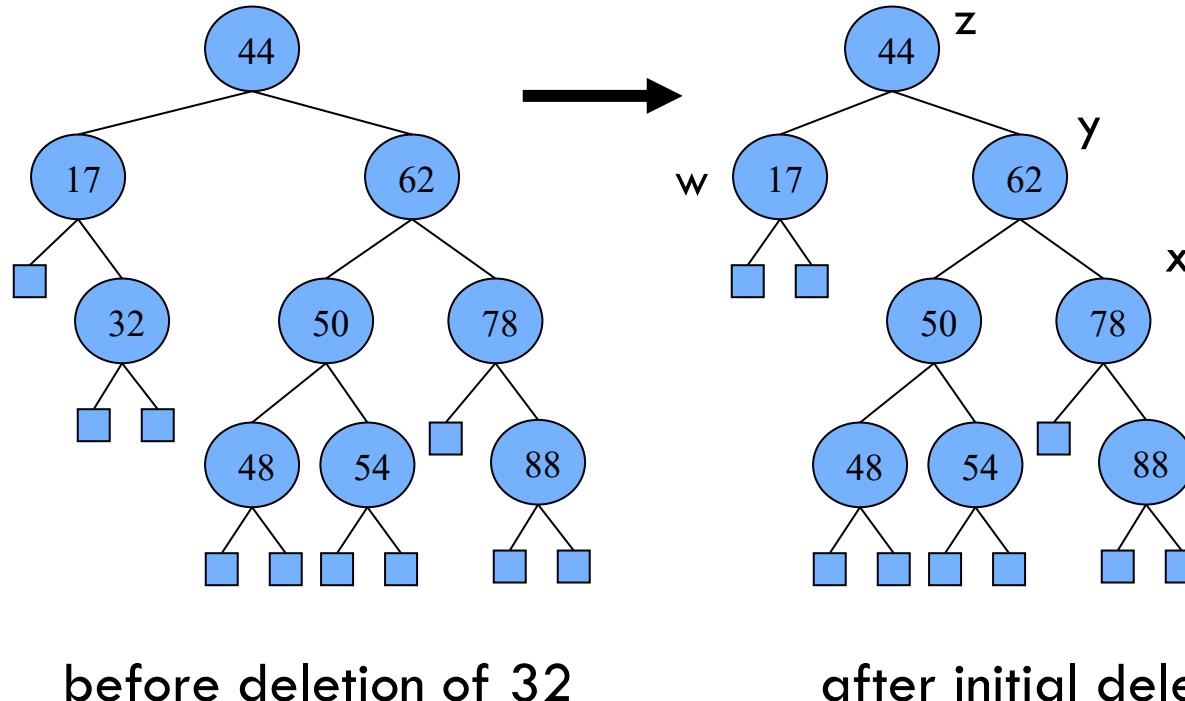
Removal in AVL trees

Suppose we are to remove a key k from our tree:

1. If k is not in the tree, search for k ends at external node
There is nothing to do so tree structure does not change
2. If k is in the tree, search for k performs usual BST removal
leading to removing a node with an external child and
promoting its other child, which we call w
3. The new tree has BST property, but it may not have AVL
balance property at some ancestor of w since
 - some ancestors of w may have decreased their height by 1
 - every node that is not an ancestor of w hasn't changed its heights
4. We use rotations to rearrange tree and re-establish AVL
property, while keeping BST property

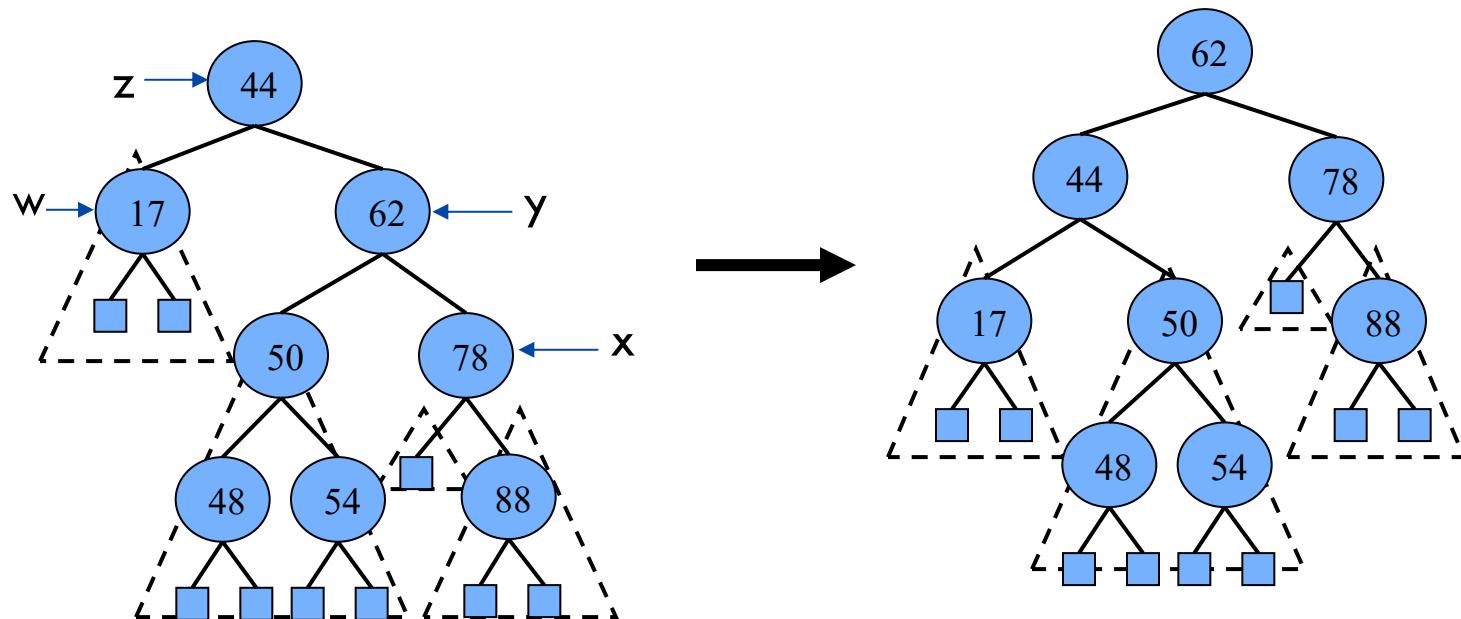
Re-establishing AVL property

- Let w be the parent of deleted node
- Let z be *lowest* ancestor of w , whose children heights differ by 2
- Let y be the child of z with larger height (y is not an ancestor of w)
- Let x be child of y with larger height



Re-establishing AVL property

- If tree does not have AVL property, do a trinode restructure at x, y, z
- This restores the AVL property at z but it may upset the balance of another node higher up in the tree, we must continue checking for balance until the root of T is reached



AVL Tree Performance

Suppose we have an AVL tree storing n items then

- The data structure uses $O(n)$ space
- Height of the tree $O(\log n)$
- Searching takes $O(\log n)$ time
- Insertion takes $O(\log n)$ time
- Removal takes $O(\log n)$ time

Today we just saw a sketch of how insertions and removals are performed. Working out all the details behind these operations is too heavy for the lecture, but I hope you got a flavor for what they involve and I encourage you to read the details on your own.

The Map ADT

- **get(k):** if the map M has an entry with key k , return its associated value
- **put(k, v):** if key k is not in M , then insert (k, v) into the map M ; else, replace the existing value associated to k with v
- **remove(k):** if the map M has an entry with key k , remove it
- **size(), isEmpty()**
- **entrySet():** return an iterable collection of the entries in M
- **keySet():** return an iterable collection of the keys in M
- **values():** return an iterable collection of the values in M

Example

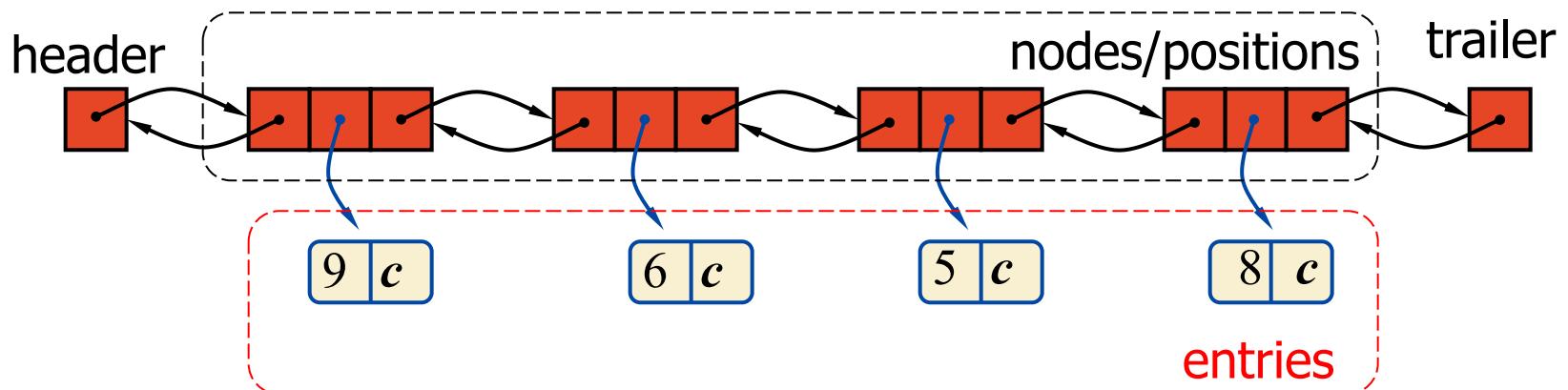
Operation	Output	Map
isEmpty()	true	\emptyset
put(5,A)	null	(5,A)
put(7,B)	null	(5,A),(7,B)
put(2,C)	null	(5,A),(7,B),(2,C)
put(8,D)	null	(5,A),(7,B),(2,C),(8,D)
put(2,E)	C	(5,A),(7,B),(2,E),(8,D)
get(7)	B	(5,A),(7,B),(2,E),(8,D)
get(4)	null	(5,A),(7,B),(2,E),(8,D)
get(2)	E	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
remove(5)	A	(7,B),(2,E),(8,D)
remove(2)	E	(7,B),(8,D)
get(2)	null	(7,B),(8,D)
isEmpty()	false	(7,B),(8,D)

List-Based (unsorted) Map

We can implement a map using an unsorted list of key-item pairs

To do a get or a put we may have to traverse the whole list, so those operations take $O(n)$ time.

Only feasible if map is very small or if we put things at the end and do not need to perform many gets (i.e., system log)



Sorted map ADT (extra methods)

firstEntry() returns the entry with smallest key; if map is empty, returns null

lastEntry() returns the entry with largest key; if map is empty, returns null

ceilingEntry(k) returns the entry with least key that is greater than or equal to k (or null, if no such entry exists)

floorEntry(k) returns the entry with greatest key that is less than or equal to k (or null, if no such entry exists)

lowerEntry(k) returns the entry with greatest key that is strictly less than k (or null, if no such entry exists)

higherEntry(k) returns the entry with least key that is strictly greater than k (or null, if no such entry exists)

subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2

Tree-Based (sorted) Map

We can implement a sorted map using a binary search tree, where each node stores a key-item pair

To do a get or a put we search for the key in the tree, so these operations take $O(h)$ time, which can be $O(\log n)$ if the tree is balanced.

Only feasible if for sorted maps, so there must exist a total ordering on the keys.

Index-based searching

Being able to do key lookups is very useful, but sometimes we need to access the i th key in the tree:

- **select(i)** returns the i th smallest key (assume no duplicate keys)

We could store the keys in a list. This solution needs to spend $O(n)$ time for selecting and for searching

We could store (i, k_i) in a tree where k_i is the i th key we want to store. This solution can do $O(\log n)$ lookups and selection but takes $O(n)$ time to do insertions and deletion.

There is a solution that can do all operations in $O(\log n)$ but requires us to augment the BST data structure.

Index-based searching

For each node u let us define $\text{size}(u)$ to be the size of the subtree rooted at u . The left subtree of u holds all the keys that are smaller than $\text{key}(u)$ subtree, so the rank of u in its subtree is $\text{size}(u.\text{left}) + 1$

```
def select(x, i):
    input A node x of a BST
        integer i : 1 <= i <= size(x) position in  $T_x$  to select
    output ith element in  $T_x$  ( $i==1$  is first key)

    if i <= size(x.left) then
        return select(x.left, i)
    else if i == size(x.left) + 1 then
        return x.key
    else
        return select(x.right, i - size(x.left) - 1)
```

Announcements

- Due to covid-19 outbreak we will move to online delivery:
 - You can attend lectures via zoom or watch lecture recordings
 - You will be able to attend tutorials via zoom (later today)
- Assignment 1 was due last week. We aim to have it marked before Assignment 3 is out
- Assignment 2 is out. It's due March 26. Please read the instructions in Ed.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

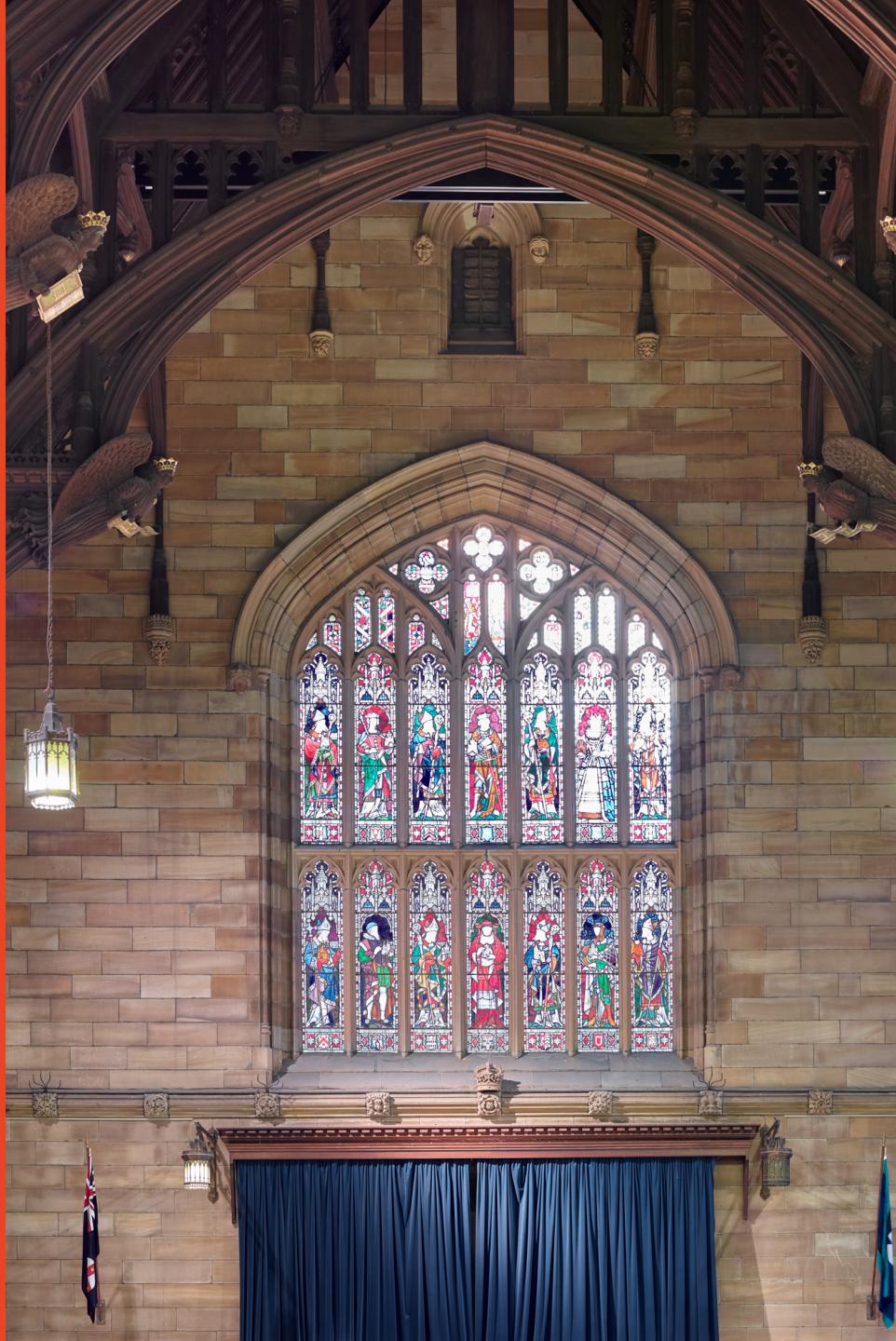
Lecture 5: Priority Queues [GT 5]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



Priority Queue ADT

Special type of ADT map to store a collection of key-value items where we can only remove smallest key:

- `insert(k, v)`: insert item with key **k** and value **v**
- `remove_min()`: remove and return the item with smallest key
- `min()`: return item with smallest key
- `size()`: return how many items are stored
- `is_empty()`: test if queue is empty

We can also have a max version of this min version, but we cannot use both versions at once.

Example

A sequence of priority queue methods:

Method	Return value	Priority queue
insert(5,A)		{(5,A)}
insert(9,C)		{(5,A),(9,C)}
insert(3,B)		{(3,B),(5,A),(9,C)}
min()	(3,B)	{(3,B),(5,A),(9,C)}
remove_min()	(3,B)	{(5,A),(9,C)}
insert(7,D)		{(5,A),(7,D),(9,C)}
remove_min()	(5,A)	{(7,D),(9,C)}
remove_min()	(7,D)	{(9,C)}
remove_min()	(9,C)	{}
is_empty()	true	{}

Application: Stock Matching Engines

At the heart of modern stock trading systems are highly reliable systems known as **matching engines**, which match the stock trades of buyers and sellers.

Buyers post bids to buy a number of shares of a given stock at or below a specified price

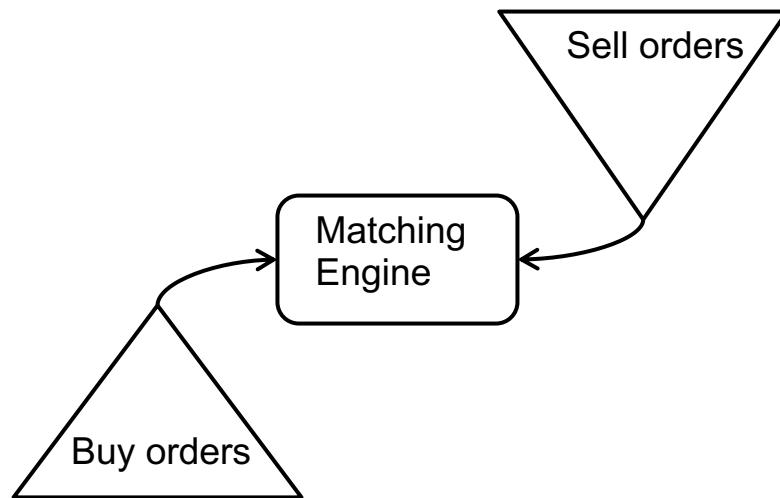
Sellers post offers (asks) to sell a number of shares of a given stock at or above a specified price.

STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Application: Stock Matching Engines

Buy and sell orders are organized according to a **price-time priority**, where price has highest priority and time is used to break ties

When a new order is entered, the matching engine determines if a trade can be immediately executed and if so, then it performs the appropriate matches according to price-time priority.



STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Application: Stock Matching Engines

A matching engine can be implemented with two **priority queues**, one for buy orders and one for sell orders.

This data structure performs element removals based on priorities assigned to elements when they are inserted.

```
while True:  
    bid = buy_orders.remove_max()  
    ask = sell_orders.remove_min()  
    if bid.price ≥ ask.price then  
        carry out trade (bid, ask)  
    else  
        buy_orders.insert(bid)  
        sell_orders.insert(ask)
```

STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Sequence-based Priority Queue

Unsorted list implementation



Sorted list implementation



- **insert** in $O(1)$ time since we can insert the item at the beginning or end of the sequence
- **remove_min** and **min** in $O(n)$ time since we have to traverse the entire list to find the smallest key

- **insert** in $O(n)$ time since we have to find the place where to insert the item
- **remove_min** and **min** in $O(1)$ time since the smallest key is at the beginning

Method	Unsorted List	Sorted List
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Priority Queue Sorting

We can use a priority queue to sort a list of keys:

1. iteratively insert keys into an empty priority queue
2. iteratively `remove_min` to get the keys in sorted order

Complexity analysis:

- `n` insert operations
- `n` `remove_min` operations

Either sequence-based implementation take $O(n^2)$

Method	Unsorted List	Sorted List
<code>size, isEmpty</code>	$O(1)$	$O(1)$
<code>insert</code>	$O(1)$	$O(n)$
<code>min, removeMin</code>	$O(n)$	$O(1)$

```
def priority_queue_sorting(A):
    pq = new priority queue
    n = size(A)
    for i in [0:n] do
        pq.insert(A[i])
    output = empty list
    for i in [0:n] do
        A[i] = pq.remove_min()
```

Selection-Sort

Variant of pq-sort using unsorted sequence implementation:

1. inserting elements with n insert operations takes $O(n)$ time
2. removing elements with n remove_min operations takes $O(n^2)$

Can be done in place
(no need for extra space)

Top level loop invariant:

- $A[0:i]$ is sorted
- $A[i:n]$ is the priority queue
and all $\geq A[i-1]$

```
def selection_sort(A):
    n = size(A)
    for i in [0:n] do
        # find s >= i minimizing A[s]
        s = i
        for j in [i+1:n] do
            if A[j] < A[s] then
                s = j
        # swap A[i] and A[s]
        A[i], A[s] = A[s], A[i]
```

Selection-Sort Example

i	A	s
0	7, 4, 8, <u>2</u> , 5, 3, 9	3
1	2, <u>4</u> , 8, 7, 5, <u>3</u> , 9	5
2	2, 3, <u>8</u> , 7, 5, <u>4</u> , 9	5
3	2, 3, 4, <u>7</u> , <u>5</u> , 8, 9	4
4	2, 3, 4, 5, <u>7</u> , 8, 9	4
5	2, 3, 4, 5, 7, <u>8</u> , 9	5
6	2, 3, 4, 5, 7, 8, <u>9</u>	6

```
def selection_sort(A):  
    n = size(A)  
    for i in [0:n] do  
        # find s ≥ i minimizing A[s]  
        s = i  
        for j in [i+1:n] do  
            if A[j] < A[s] then  
                s = j  
        # swap A[i] and A[s]  
        A[i], A[s] = A[s], A[i]
```

Insertion-Sort

Variant of pq-sort using sorted sequence implementation:

1. inserting elements with n insert operations takes $O(n^2)$ time
2. removing elements with n remove_min operations takes $O(n)$

Can be done in place
(no need for extra space)

Top level loop invariant:

- $A[0:i]$ is the priority queue (and thus sorted)
- $A[i:n]$ is yet-to-be-inserted

```
def insertion_sort(A):
    n = size(A)
    for i in [1:n] do
        x = A[i]
        # move forward entries > x
        j = i
        while j > 0 and x < A[j-1] do
            A[j] = A[j-1]
            j = j - 1
        # if j>0 ⇒ x ≥ A[j-1]
        # if j<i ⇒ x < A[j+1]
        A[j] = x
```

Insertion-Sort Example

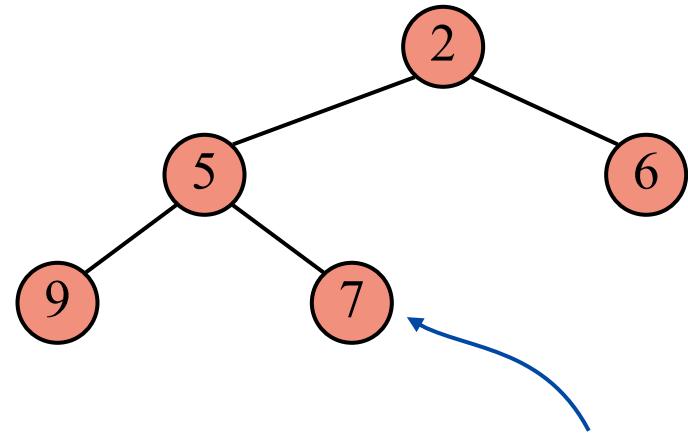
i	A	i
1	7, <u>4</u> , 8, 2, 5, 3, 9	0
2	4, 7, <u>8</u> , 2, 5, 3, 9	2
3	<u>4</u> , 7, 8, <u>2</u> , 5, 3, 9	0
4	2, 4, <u>7</u> , 8, <u>5</u> , 3, 9	2
5	2, <u>4</u> , 5, 7, 8, <u>3</u> , 9	1
6	2, 3, 4, 5, 7, 8, <u>9</u>	6

```
def insertion_sort(A):  
    n = size(A)  
    for i in [1:n] do  
        x = A[i]  
        # move forward entries > x  
        j = i  
        while j > 0 and x < A[j-1] do  
            A[j] = A[j-1]  
            j = j - 1  
        # if j>0 ⇒ x ≥ A[j-1]  
        # if j<i ⇒ x < A[j+1]  
        A[j] = x
```

Heap data structure (min-heap)

A **heap** is a binary tree storing (key, value) items at its nodes, satisfying the following property:

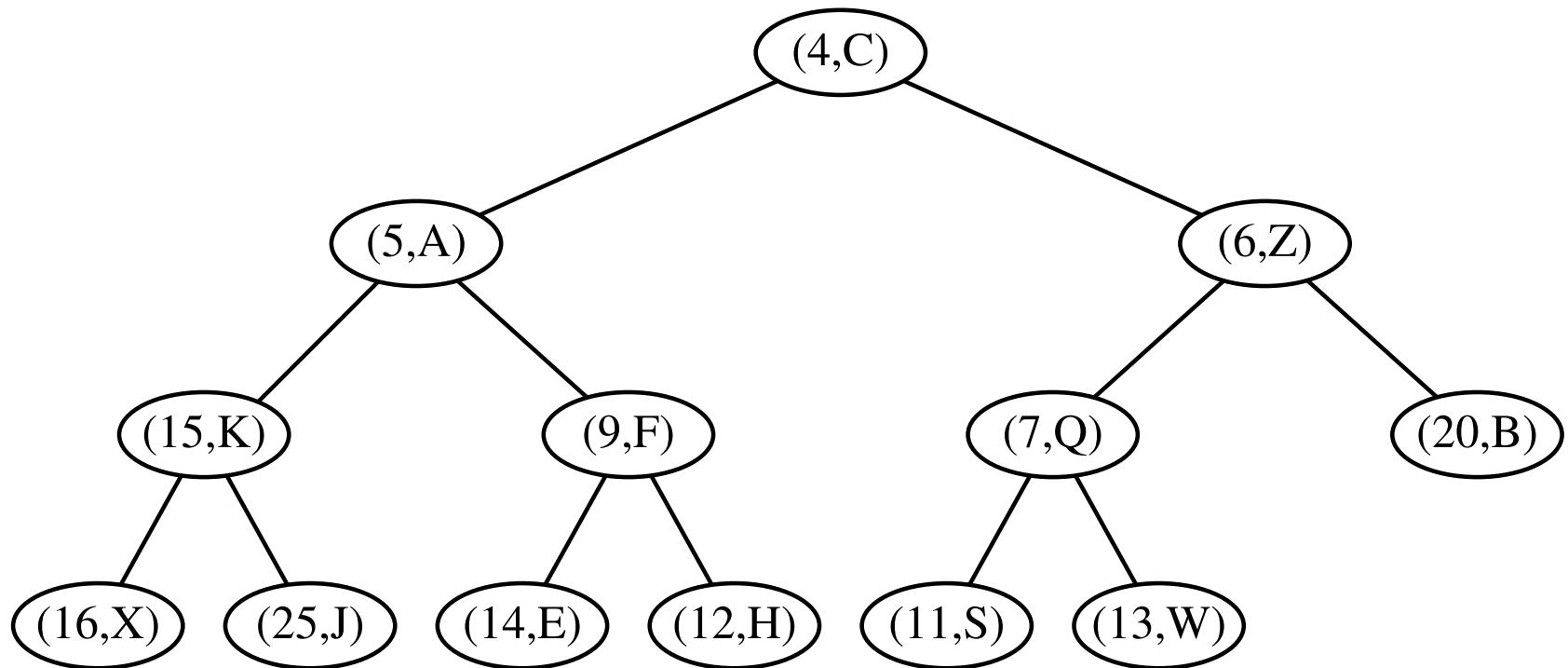
1. **Heap-Order:** for every node $m \neq \text{root}$,
 $\text{key}(m) \geq \text{key}(\text{parent}(m))$



2. **Complete Binary Tree:** let h be the height
 - every level $i < h$ is full (i.e., there are 2^i nodes)
 - remaining nodes take leftmost positions of level h

The **last node** is the rightmost node of maximum depth

Example

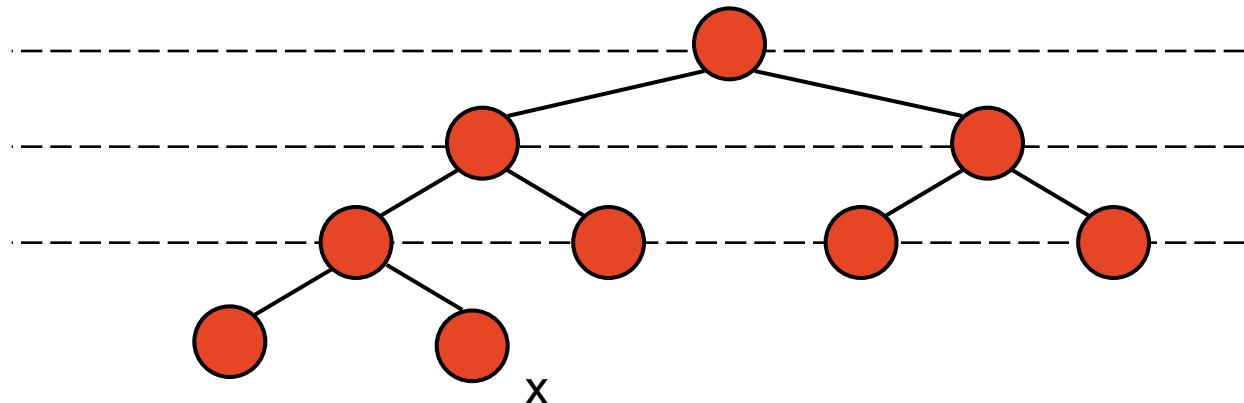


Minimum of a Heap

Fact: The root always holds the smallest key in the heap

Proof:

- Suppose the minimum key is at some internal node x
- Because of the heap property, as we move up the tree, the keys can only get smaller (assuming repeats, otherwise contradiction)
- If x is not the root, then its parent must also hold a smallest key
- Keep going until we reach the root

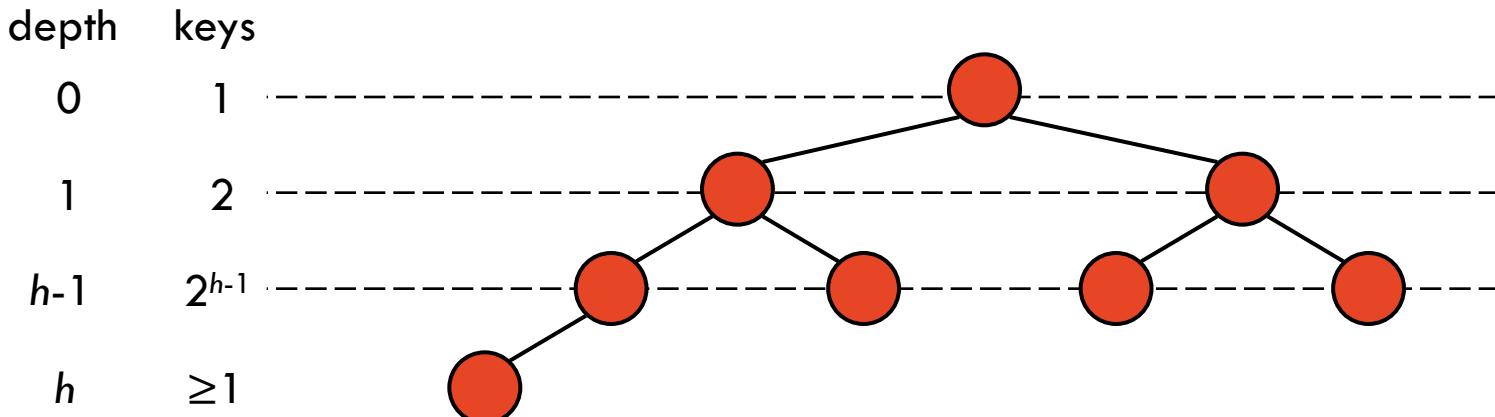


Height of a Heap

Fact: A heap storing n keys has height $\log n$

Proof:

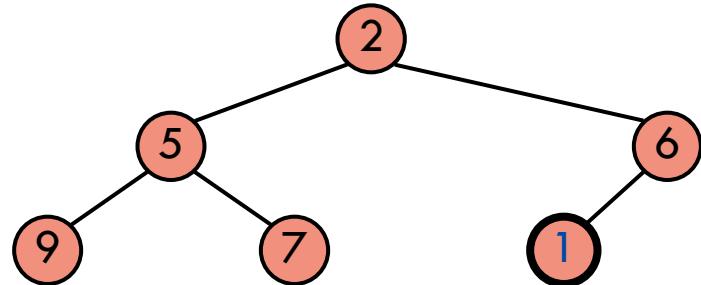
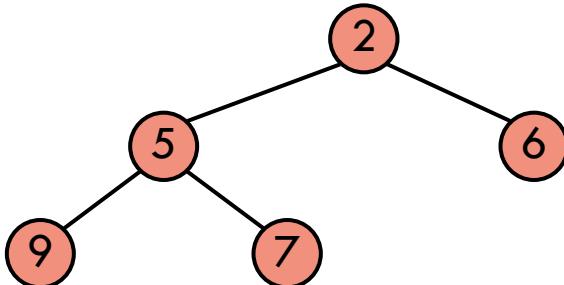
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h - 1$ and at least one key at depth h , we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, applying \log_2 on both sides, $\log_2 n \geq h$



Insertion into a Heap

- Create a new node with given key
- Find location for new node
- Restore the heap-order property

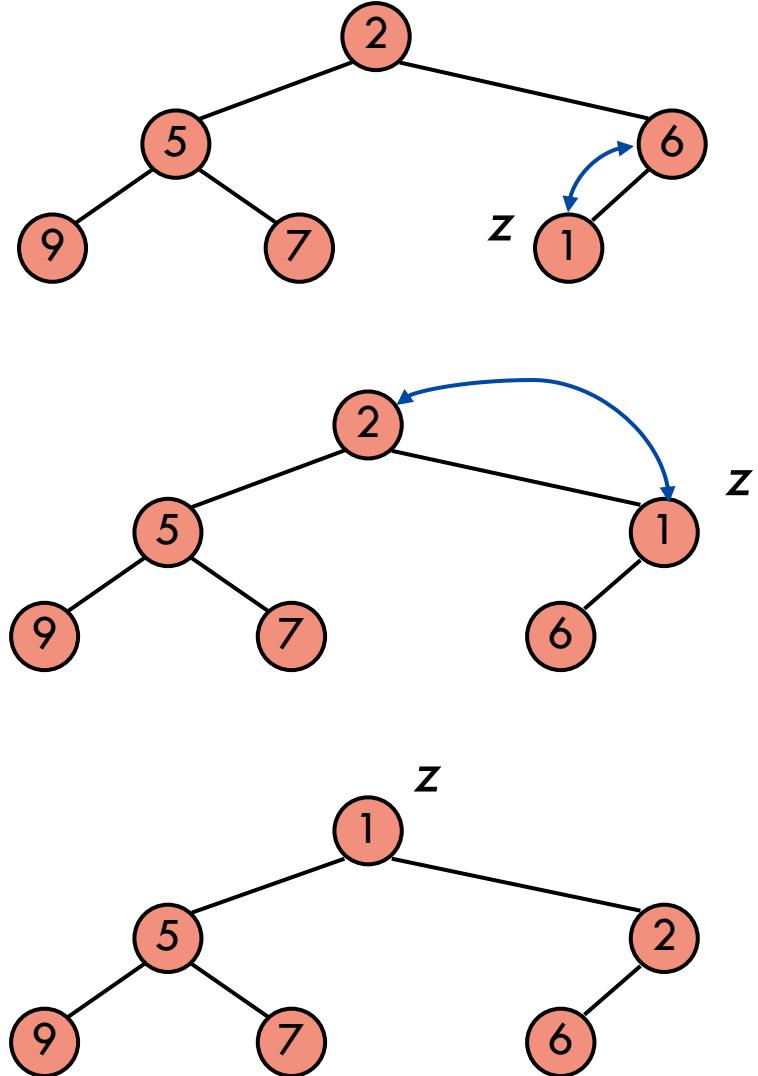
insert(1)



Upheap

Restore heap-order property by swapping keys along upward path from insertion point

```
def up_heap(z):
    while z ≠ root and
        key(parent(z)) > key(z) do
        swap key of z and parent(z)
        z = parent(z)
```



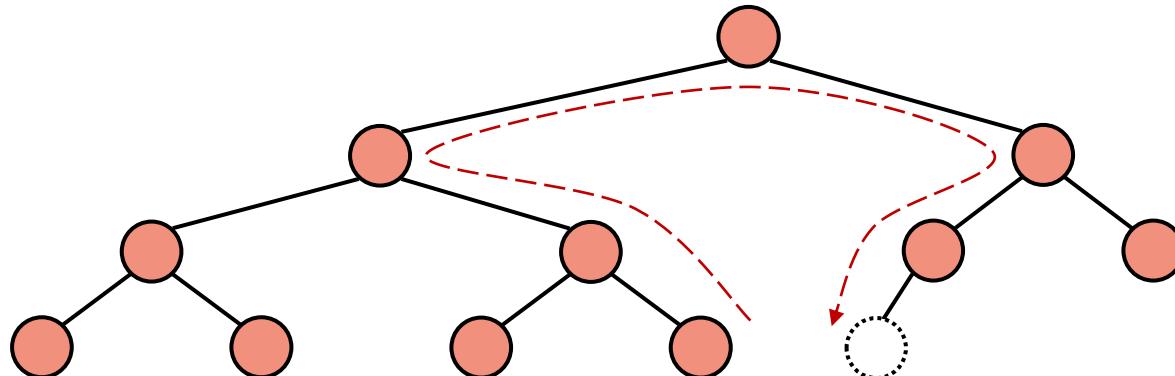
Correctness: after swapping the subtree rooted at **z** has the property

Complexity: **O(log n)** time because the height of the heap is **log n**

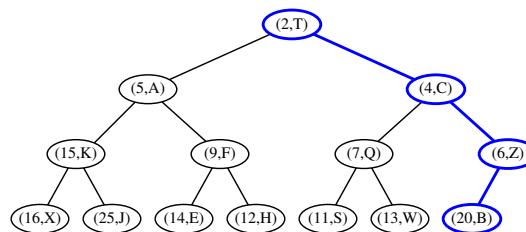
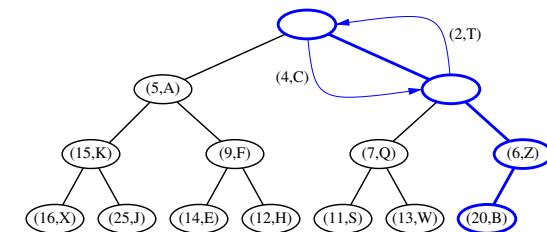
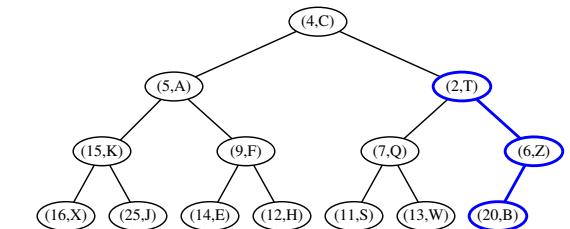
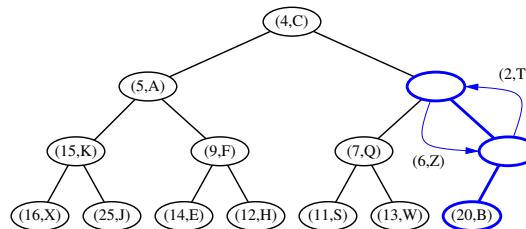
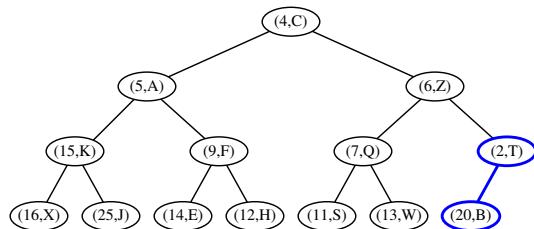
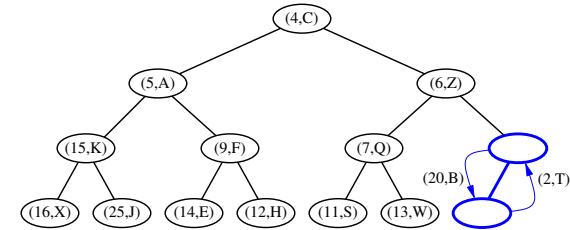
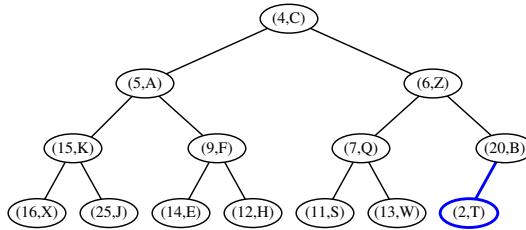
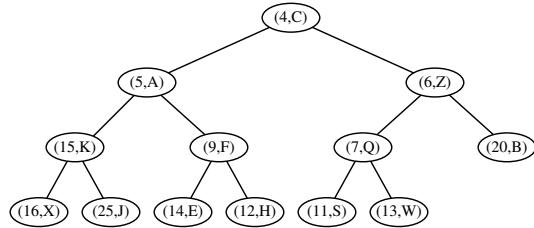
Finding the position for insertion

- start from the last node
- go up until a left child or the root is reached
- if we reach the root then need to open a new level
- otherwise, go to the sibling (right child of parent)
- go down left until a leaf is reached

Complexity of this search is $O(\log n)$ because the height is $\log n$.
Thus, overall complexity of insertion is $O(\log n)$ time



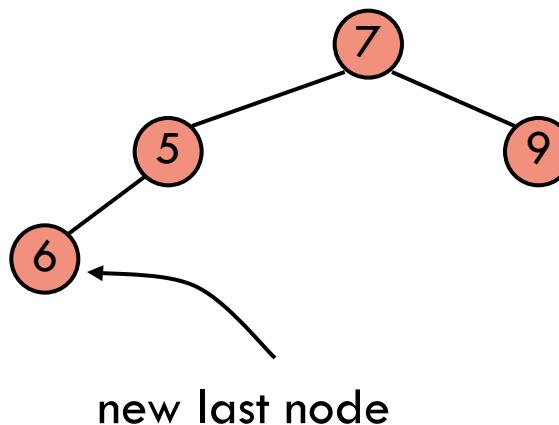
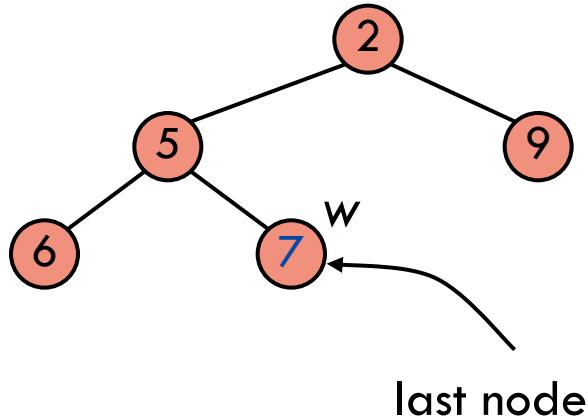
Example insertion



Removal from a Heap

- Replace the root key with the key of the last node w
- Delete w
- Restore the heap-order property

`remove_min()`



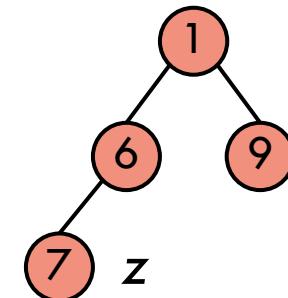
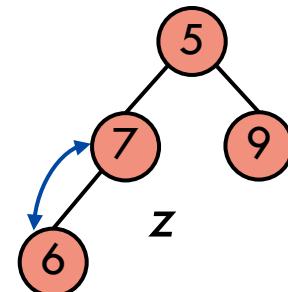
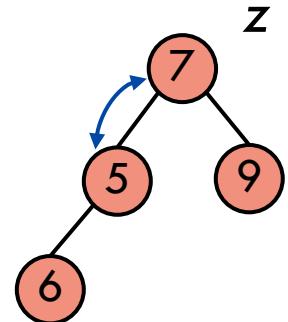
Downheap

Restore heap-order property by swapping keys along downward path from the root

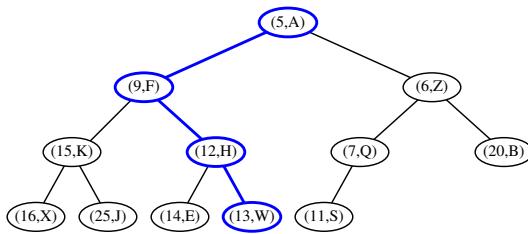
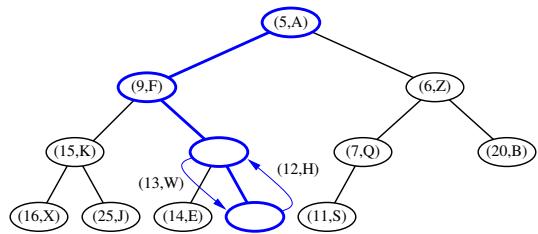
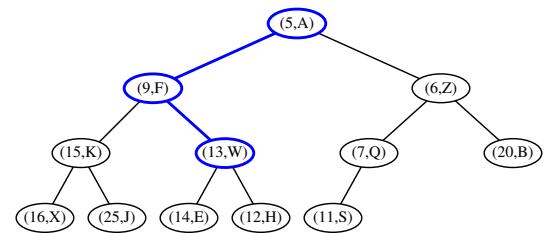
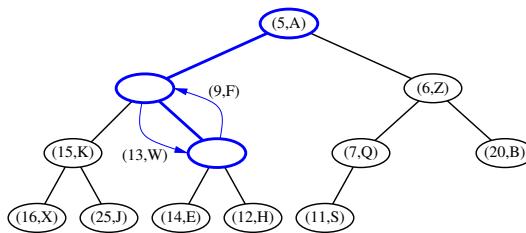
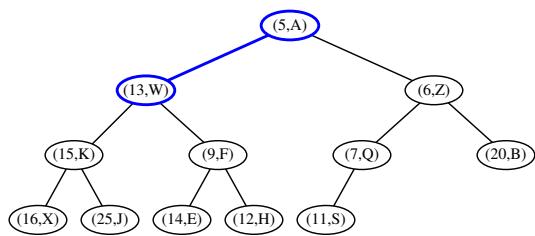
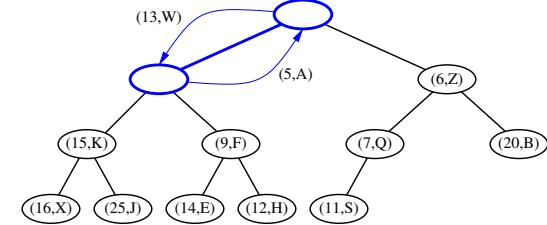
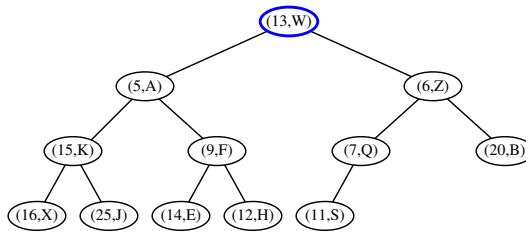
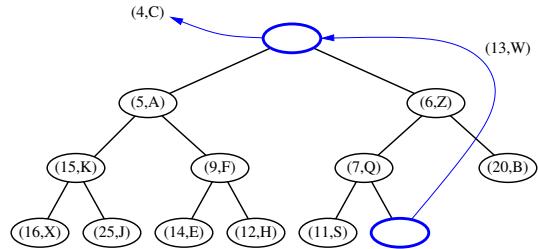
```
def down_heap(z):
    while z has child with
        key(child) < key(z) do
            x = child of z with smallest key
            swap keys of x and z
            z = x
```

Correctness: after swap **z** heap-order property is restored up to level of **z**

Complexity: **O(log n)** time because the height of the heap is **log n**



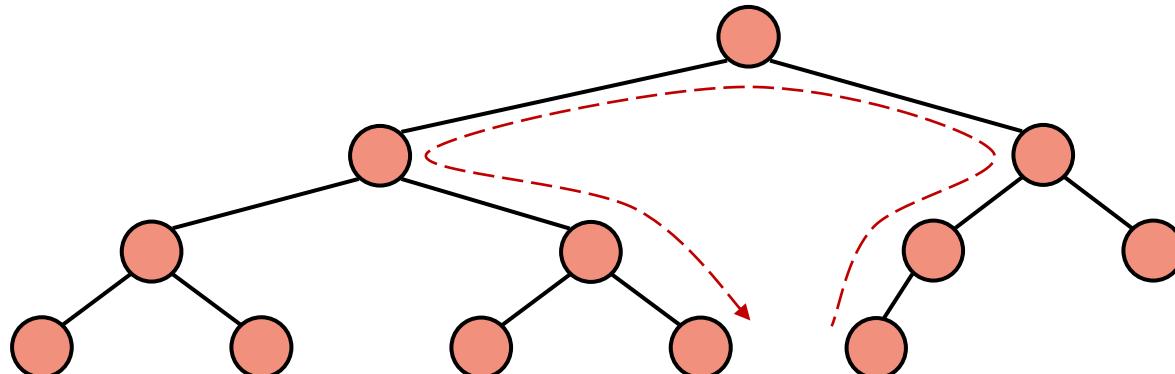
Example removal



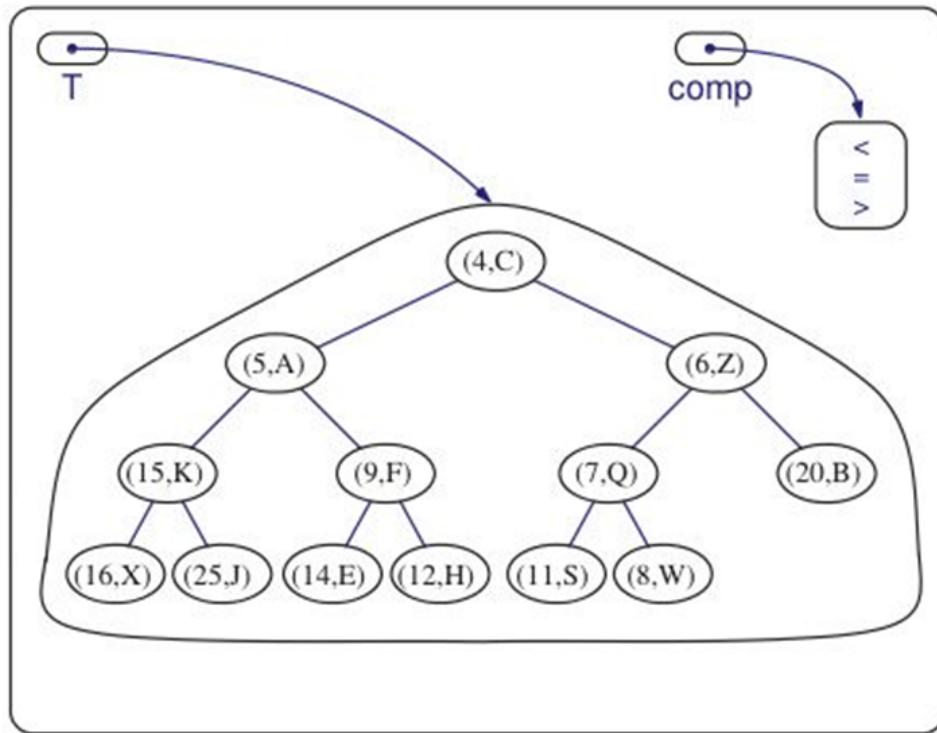
Finding next last node after deletion

- start from the (old) last node
- go up until a right child or the root is reached
- if we reach the root then need to close a level
- otherwise, go to the sibling (left child of parent)
- go down right until a leaf is reached

Complexity of this search is $O(\log n)$ because the height is $\log n$.
Thus, overall complexity of insertion is $O(\log n)$ time



Heap-based implementation of a priority queue



Operation	Time
size, isEmpty	$O(1)$
min,	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Heap-Sort

Consider a priority queue with n items implemented with a heap:

- the space used is $O(n)$
- methods `insert` and `remove_min` take $O(\log n)$

Recall that priority-queue sorting uses:

- n `insert` ops
- n `remove_min` ops

Heap-sort is the version of priority-queue sorting that implements the priority queue with a heap. It runs in $O(n \log n)$ time.

Heap-in-array implementation

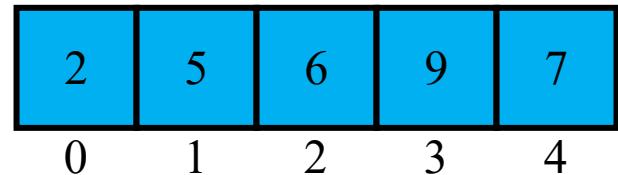
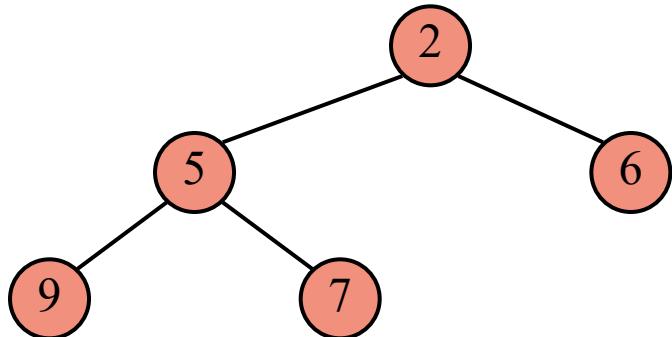
We can represent a heap with n keys by means of an array of length n

Special nodes:

- root is at 0
- last node is at $n-1$

For the node at index i :

- the left child is at index $2i+1$
- the right child is at index $2i+2$
- Parent is at index $\lfloor (i-1)/2 \rfloor$



Refinements and Generalization

Heap-sort can be arranged to work in place using part of the array for the output and part for the priority queue

A heap on n keys can be constructed in $O(n)$ time. But the n `remove_min` still take $O(n \log n)$ time

Sometimes it is useful to support a few more operations:

- `remove(e)`: Remove item e from the priority queue
- `replace_key(e, k)`: update key of item e with k
- `replace_value(e, v)`: update value of item e with v

Summary: Priority queue implementations

Method	Unsorted List	Sorted List	Heap
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$

Implementing a Priority Queue

Entries: An object that keeps track of the associations between keys and values

Comparators: A function or an interface to compare entry objects

compare(a, b): returns an integer i such that

- $i < 0$ if $a < b$,
- $i = 0$ if $a = b$
- $i > 0$ if $a > b$

Warning: do not assume that $\text{compare}(a,b)$ is always $-1, 0, 1$

Stock Application Revisited



Online trading system where orders are stored in two priority queues (one for sell orders and one for buy orders) as (p, t, s) entries:

- The key is (p, t) , the price of the order p and the time t such that we first sort by p and break ties with t
- The value is s , the number of shares the order is for

How do we implement the following:

- What should we do when a new order is placed?
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Building a Priority Queue in one go

Sometimes we have all the keys upfront. If we insert them one at a time, this can take $O(n \log n)$ time. However, there is a faster way to build the priority queue in this case.

```
def heapify (A):
    # turn A into a binary heap in place
    n = size(A)
    for i in [n-1:0:-1] do
        down_heap(A, i)
```

If we let $h(i)$ be the height of the node corresponding to $A[i]$ then $\text{down_heap}(A, i)$ can take $O(h(i))$ time.

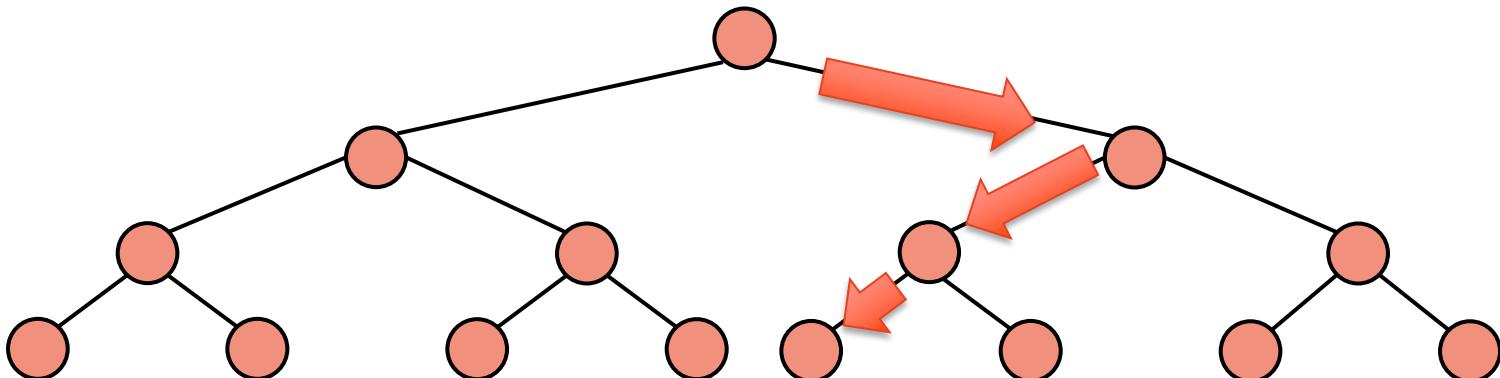
Thus, the running time of the algorithm is $O(\sum_i h(i))$

Building a Priority Queue in one go

Claim: The running time of the algorithm is $O(\sum_i h(i)) = O(n)$

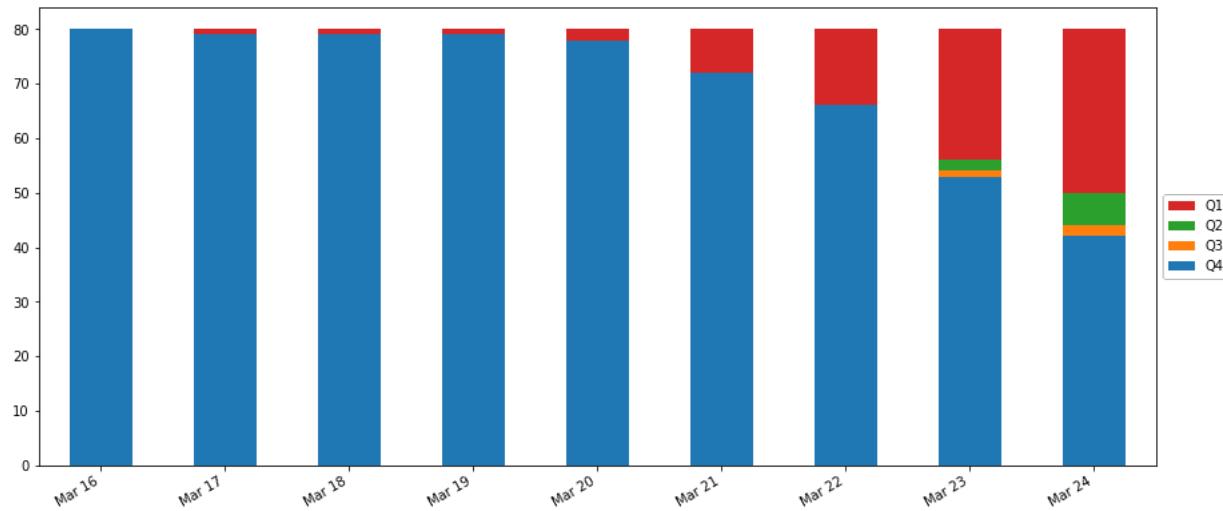
For each node i in the tree construct a path of length $h(i)$ by starting at node i , going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint



Assignment 2

- We are using git for submissions
- Clone repo, commit and push and edit **now** so that you don't encounter technical problems at the last minute.



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Data structures and Algorithms

Lecture 6: Hash tables [GT 6.1-4]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



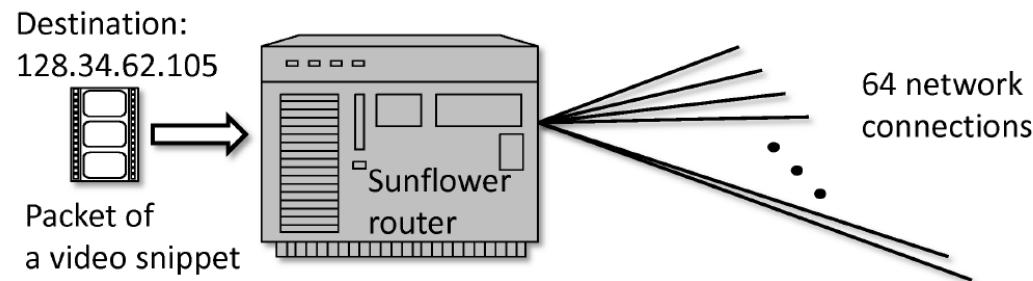
Application: Network Routers

Network routers process multiple streams of packets at high speed. To process a packet with destination k and data payload x , a router must determine which outgoing link to send the packet along

Such a system needs to support:

- destination-based lookups, i.e., $\text{get}(k)$ operations that return the outgoing link for destination k
- updates to the routing table, i.e., $\text{put}(k, c)$ operations, where c is the new outgoing link for destination k .

Ideally, we would like to achieve $O(1)$ time for both operations.



Recall: Maps

A map models a searchable collection of key-value pairs (a.k.a., items or entries)

The main operations of a map are for searching, inserting, and deleting items

At most one item per key is allowed



Recall: Map Operations

- **get(k)**: if the map M has an entry with key k , return its associated value; else, return null
- **put(k, v)**: insert entry (k, v) into the map M ; if key k is not already in M , then return null; else, return old value associated with k
- **remove(k)**: if the map M has an entry with key k , remove it from M and return its associated value; else, return null
- **size()**
- **is_empty()**

Map Operations (extended)

- `entries()`: return an iterable collection of the entries in M
- `keys()`: return an iterable collection of the keys in M
- `values()`: return an iterable collection of the values in M

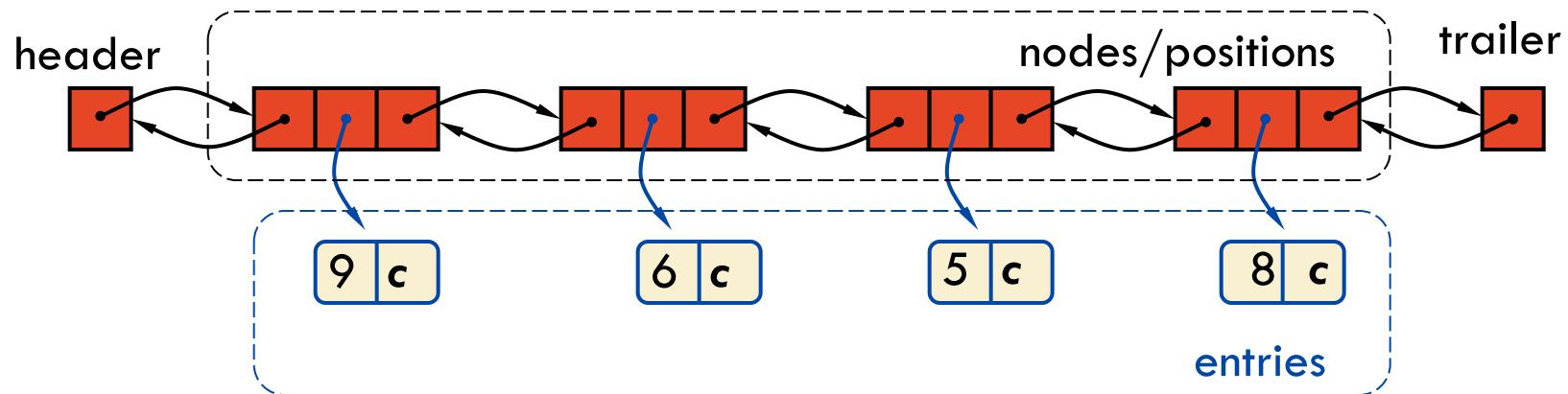
Example

Operation	Output	Map
		\emptyset
is_empty()	true	
put(5,A)	null	(5,A)
put(7,B)	null	(5,A), (7,B)
put(2,C)	null	(5,A), (7,B), (2,C)
put(8,D)	null	(5,A), (7,B), (2,C), (8,D)
put(2,E)	C	(5,A), (7,B), (2,E), (8,D)
get(7)	B	(5,A), (7,B), (2,E), (8,D)
get(4)	null	(5,A), (7,B), (2,E), (8,D)
get(2)	E	(5,A), (7,B), (2,E), (8,D)
size()	4	(5,A), (7,B), (2,E), (8,D)
remove(5)	A	(7,B), (2,E), (8,D)
remove(2)	E	(7,B), (8,D)
get(2)	null	(7,B), (8,D)
is_empty()	false	(7,B), (8,D)

A Simple List-Based Map

We can implement a map using an unsorted list

- Store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Performance of a List-Based Map

Performance:

- **put** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
- **get** and **remove** take $O(n)$ time since in the worst case we must traverse the entire sequence to look for an item with the given key

The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rare (e.g., historical record of logins to a workstation)

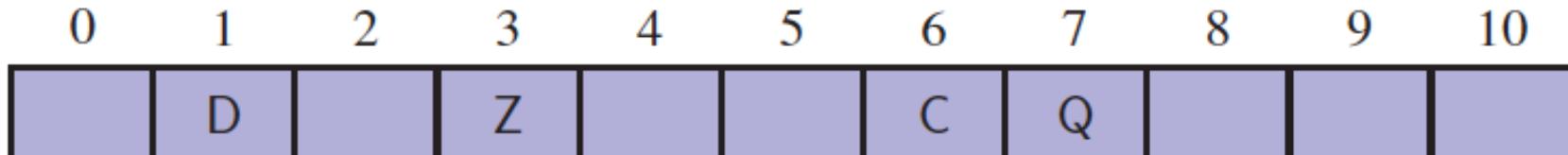
Simple Implementation with restricted keys

Maps support the abstraction of using keys as addresses to get items

Consider a restricted setting in which a map with n items with keys in a range from 0 to $N - 1$, for some $N \geq n$.

- Implement with an array of size N
- Key can be index so entries can be located directly
- $O(1)$ operations (get, put, remove)

Drawback is that usually $N \gg n$, e.g. StudentID is 9 digits, so a Map with StudentID key can be stored in array of 10,000,000,000 entries (way more than the number of students).



Evaluation of this structure

Really good worst-case runtime

Often, bad space utilization when key set is sparse in the space of possible keys, as in StudentID example

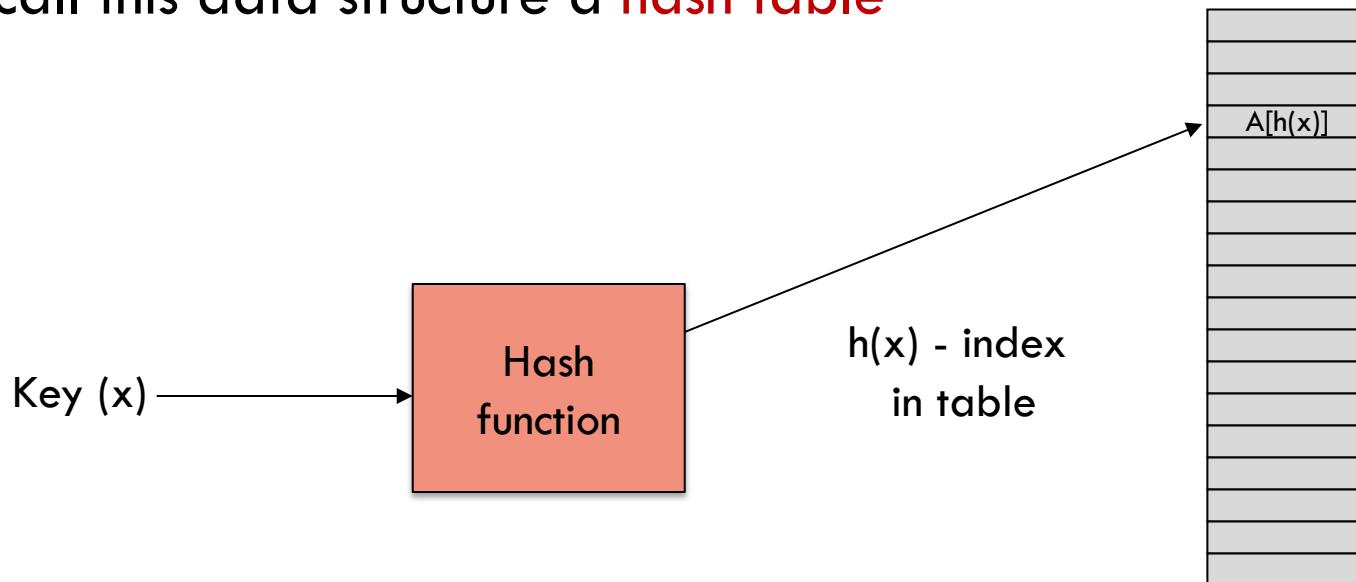
Unable to handle more general keys like strings

Hash Functions and Hash Tables

To get around these issues, we use a **hash function h** to map keys to corresponding indices in an array A .

- h is a mathematical function (always gives same answer for any particular x)
- h is fairly efficient to compute

We call this data structure a **hash table**





Hash Functions and Hash Tables

A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$.

- **Example:** $h(x) = x \bmod N$ is a hash function for integer keys
- The integer $h(x)$ is called the **hash value** of key x

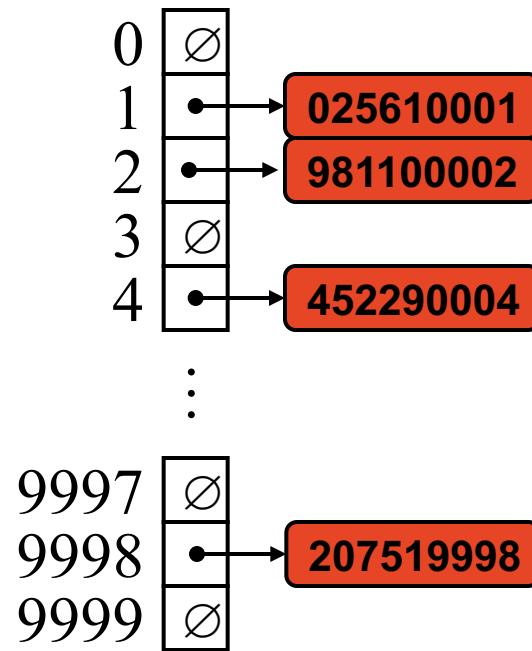
A **hash table** for a given key type K consists of

- Hash function $h: K \rightarrow [0, N-1]$
- Array (called table) of size N
- Ideally, item (x, o) is stored at $A[h(x)]$

Example

We design a hash table for a map storing entries as SIDs (student ids, a nine-digit positive integer).

Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$



Choice of Hash functions

Choosing a good hash function is not straightforward (to be discussed)

For our examples, we usually use very simple (and not good) choices that can be calculated by hand

- e.g. for unbounded integer key in array of size 11, we might use remainder mod 11 as hash function
- e.g. for String key, in array of size 10 we might do an example where $h(S) = (\text{position in alphabet of first character of } S) \bmod 10$, so $h(\text{"Mary"}) = 3$ since M is 13-th character in alphabet

Arithmetic modulo N

$x \bmod N$ is mathematical notation for remainder

- If $x = c \cdot N + r$ with $0 \leq r < N$ then $r = x \bmod N$
- Also $r = x - N \cdot \lfloor x/N \rfloor$
- So numbers wrap-around when working mod N
 - $35 \bmod 10 = 5$
 - $20 \bmod 10 = 0$
- Java/Python operator ($x \% N$)

Hash Functions

Many types of keys to start from : integers, floating point numbers, strings, or arbitrary objects (for example a whole binary search tree)

A hash function h is usually the composition of two functions:

- Hash code:

$h_1 : \text{keys} \rightarrow \text{integers}$

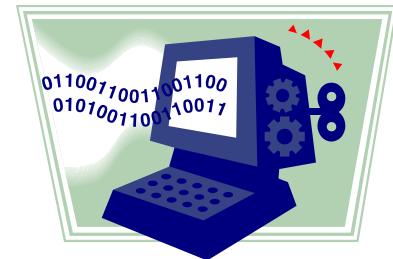
- Compression function:

$h_2 : \text{integers} \rightarrow [0, N - 1]$

$$h(x) = h_2(h_1(x))$$

The goal of the hash function is to “disperse” the keys in an apparently random way. In general we want to avoid having many items being hashed to the same location.

Common Hash Codes



Designing a good hash code is a bit of an artform.

There are two general approaches that one can take:

- view the key k as a tuple of integers (x_1, x_2, \dots, x_d) with each being an integer in the range $[0, M-1]$ for some M
- view the key k as (possibly very large) nonnegative integer

Examples:

- strings, image
- IP address, account number

Summing components

Used for keys $k = (x_1, x_2, \dots, x_d)$. There are many options:

- $h(k) = \sum_i x_i$
- $h(k) = \sum_i x_i \bmod p$ where p is a prime
- $h(k) = \bigoplus_i x_i \bmod p$

May cause problems because these hash code are invariant under permutations of the key tuple.

Example: “mate”, “meat”, “tame”, “team” all map to same code

Summing components

Used on keys $k = (x_1, x_2, \dots, x_d)$. For a given value of a we define

$$h(k) = x_1 a^{d-1} + x_2 a^{d-2} + \dots + x_{d-1} a + x_d$$

Now two permutations of the same tuple need to collide

Some observations:

- can be evaluated with Horner's algorithm in $O(d)$ time
- arithmetic ops usually done modulo a prime to avoid overflow
- value of a is chosen empirically to avoid collisions

Modular division

Used on keys k that are positive integers

$$h(k) = k \bmod N$$

for some prime number N

Fact: If keys are randomly uniformly distributed in $[0, M]$ where $M \gg N$ then the probability that two keys collide is $1/N$

Alas, keys are usually not randomly distributed.

Universal hash functions

Let H be a family of hash functions $[0, M] \rightarrow [0, N-1]$. We say that H is 2-universal if picking h uniformly at random (UAR) from H yields that for any two keys i and j :

$$\Pr[h(i) = h(j)] \leq 1/N$$

Fact: Let h be a function chosen UAR from a 2-universal family then the expected number of collision for a given key k in a set S of n keys is n/N

$$E[|\{j \in S : h(i) = h(j)\}|] = E[\sum_{j \in S} \Pr[h(i) = h(j)]] = |S|/N = n/N$$

Random Linear Hash Function

Used on keys k that are positive integers

$$h(k) = ((a k + b) \bmod p) \bmod N$$

for some prime number p , and a and b are chosen uniformly at random from $[0, p-1]$ with $a \neq 0$

Fact: If the keys are in the range $[0, M]$ and $p > M$ then the probability that two keys collide is $1/N$

[See Theorem 6.3 in GT for a proof]

Collision Handling



Collisions occur when two or more elements are hashed to the same location in our array

A good hash function makes collisions rare

However, when collisions do happen we need to have a method for dealing with them:

- Separate chaining
- Linear probing
- Cuckoo hashing

Separate Chaining

Let each cell in the table point to a linked list holding the entries that map there

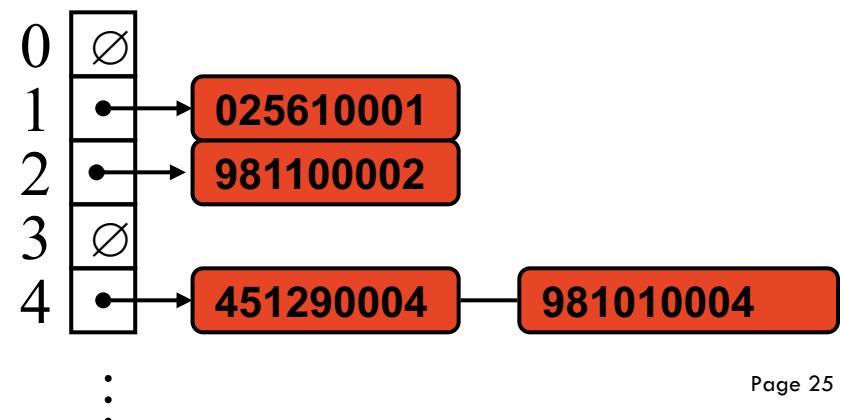
Get, put, and remove operations are delegated to the appropriate list

Separate chaining is simple, but requires additional memory outside the table

```
def get(k):  
    return A[h(k)].get(k)
```

```
def put(k, v):  
    A[h(k)].put(k, v)
```

```
def remove(k):  
    return A[h(k)].remove(k)
```



Performance of Separate Chaining

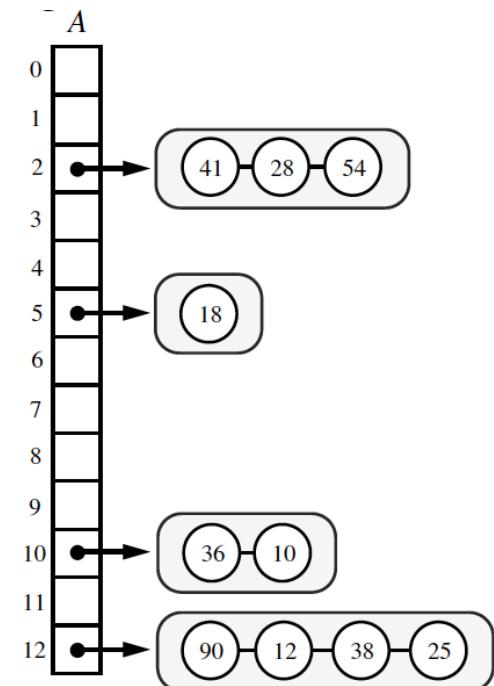
Assume that our hash function, maps n keys to independent uniform random values in the range $[0, N-1]$.

Let X be a random variable representing the number of items that map to a bucket in the array A , then $E(X) = n/N$

The parameter n/N , is called the **load factor** of the hash table, usually written as α .

The expected time for hash table operations is $O(1+\alpha)$ when collisions are handled with separate chaining.

But the worst case time is $O(n)$, which happens when all the items collide into a single chain



Open addressing using Linear Probing

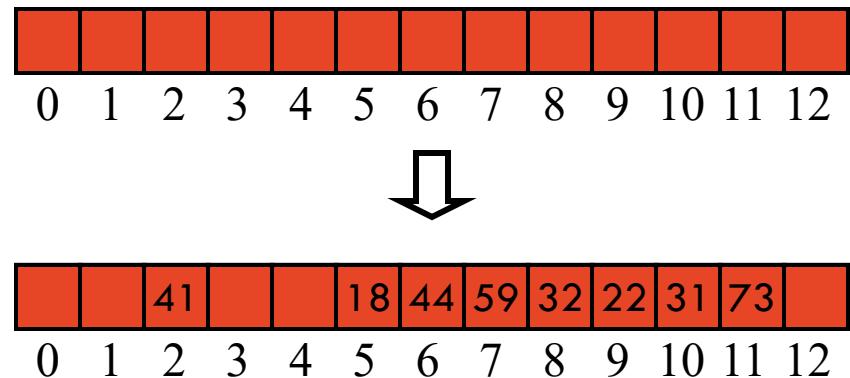
Open addressing: the colliding item is placed in a different cell of the table

Linear probing: handles collisions by placing the colliding item in the next (circularly) available cell

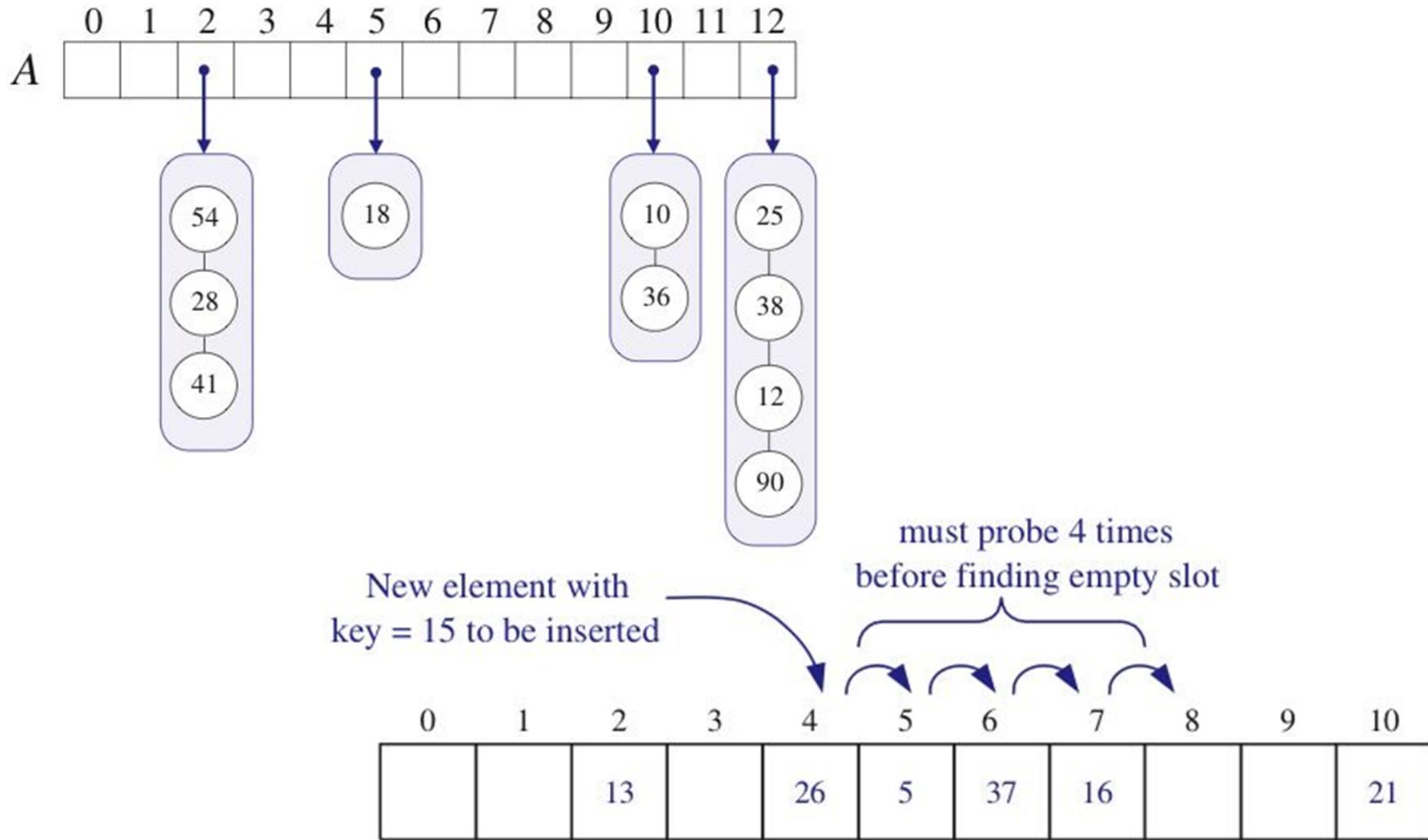
- Each table cell inspected is referred to as a probe
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Example with $h(x) = x \bmod 13$. Suppose we sequentially insert:

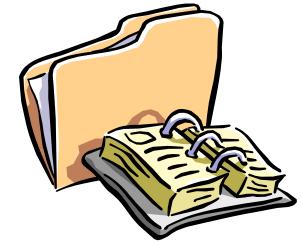
18 [5], 41 [2], 22 [9],
44 [5], 59 [7], 32 [6],
31 [5], 73 [11]



Chaining versus probing



Search with Linear Probing



How to implement `get(k)` in a hash table with linear probing:

- Start at cell $h(k)$
- Probe consecutive locations until one of the following occurs
 - An item with key k is found, or
 - An empty cell is found, or
 - N cells have been probed

```
def get(k):  
    i = h(k)  
    p = 0  
    repeat  
        c = A[i]  
        if c == ∅ then  
            return null  
        else if c.get_key()=k  
            return c.get_value()  
        else  
            i = (i + 1) mod N  
            p = p + 1  
    until p == N  
    return null
```

Updates with Linear Probing

To handle insertions and deletions, we introduce a special object, called *DEFUNCT*, which replaces deleted elements

- **get(k):** must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell
 - **remove(k):** search for the entry as in get(k). If found, replace it with the special item *DEFUNCT* and return element o
 - **put(k, o):** search for the entry as in get(k), but we also remember the index j of the first cell we find that has DEFUNCT or empty.
 - If we find key k , we replace the value there with o and return the previous value.
 - If we don't find k , we store (k, o) in cell with index j
- Throw exception if table is full

Performance of Linear Probing

In the worst case, get, put, and remove take $O(n)$ time.

Fact: Assuming hash values are uniformly randomly distributed, expected number of probes for each get and put is $1/(1-\alpha)$ where $\alpha = n/N$ is the load factor of the hash table.

Thus, if the load factor is a constant < 1 then the expected running time for get and put operations is $O(1)$

In practice, hashing is very fast provided the load factor is not close to 100%, but removals complicate the implementation and degrade the performance.

Hash tables implementations

Recall that load factor of a hash table is defined as $\alpha = n/N$

Experiments and theory suggest that α should be kept not too high:

- Java's HashSet uses chaining with $\alpha < 0.75$ and switches from a linked list to a binary search tree if bucket gets too large
- Python's dict uses open addressing with $\alpha < 0.66$

When a hash table reaches its load factor, the table is replaced with a larger table (e.g., twice the size) and the elements are hashed over to the new table

Cuckoo hashing

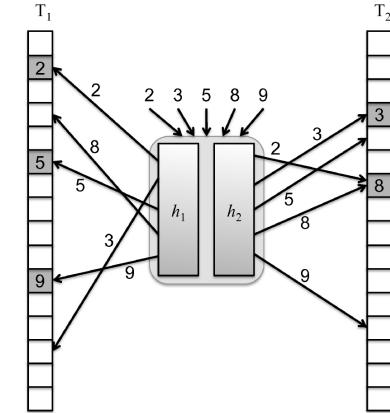
Main problem with the methods we've seen so far is that operations take $O(n)$ time in the worst-case time.

Cuckoo hashing achieves worst-case $O(1)$ time for lookups and removals, and expected $O(1)$ time for insertions.

In practice Cuckoo hashing is 20-30% slower than linear probing but is still often used due to its worst case guarantees on lookups and its ability to handle removal more gracefully.

The Power of Two Choices

Use two lookup, T_1 and T_2 , each of size N



Use two hash functions, h_1 and h_2 , for T_1 and T_2 respectively

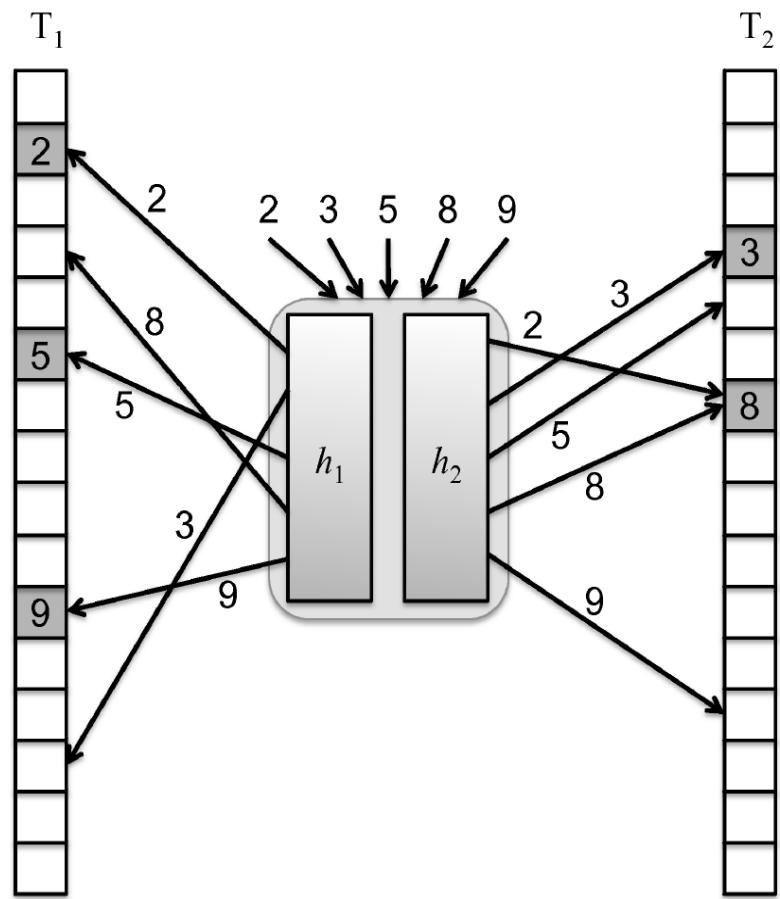
For an item with key k , there are only **two possible places** where we are allowed to store the item: $T_1[h_1(k)]$ or $T_2[h_2(k)]$

This restriction, simplifies lookup dramatically, while still allowing worst-case expected $O(1)$ running time for put and remove.

An Example of Cuckoo Hashing

Each key in the set $S = \{2, 3, 5, 8, 9\}$ has two possible locations it can go, one in the table T_1 and one in the table T_2 .

Note that 2 and 8 collide in T_2 , but that is okay, since there is no collision for 2 in its alternative location in T_1 .



Pseudo-code for get and remove

```
def get(k):
    if T1[h1(k)] != null and T1[h1(k)].key == k then
        return T1[h1(k)].value
    if T2[h2(k)] != null and T2[h2(k)].key == k then
        return T2[h2(k)].value
    return null
```

```
def remove(k):
    temp = null
    if T1[h1(k)] != null and T1[h1(k)].key == k then
        temp = T1[h1(k)].value
        T1[h1(k)] = null
    if T2[h2(k)] != null and T2[h2(k)].key == k then
        temp = T2[h2(k)].value
        T2[h2(k)] = null
    return temp
```

Both are simple and
run in $O(1)$ time

High level idea behind put

If a collision occurs in the insertion operation in the cuckoo hashing scheme, then we evict the previous item in that cell and insert the new one in its place.

This forces the evicted item to go to its alternate location in the other table and be inserted there, which may repeat the eviction process with another item, and so on.

Eventually, we either find an empty cell and stop or we repeat a previous eviction, which indicates an eviction cycle.

If we discover an eviction cycle, then we bail out or rehash all the items into larger tables

Intuition for the Name

The name “cuckoo hashing” comes from the way the $\text{put}(k, v)$ operation is performed in this scheme, because it mimics the breeding habits of the Common Cuckoo bird.

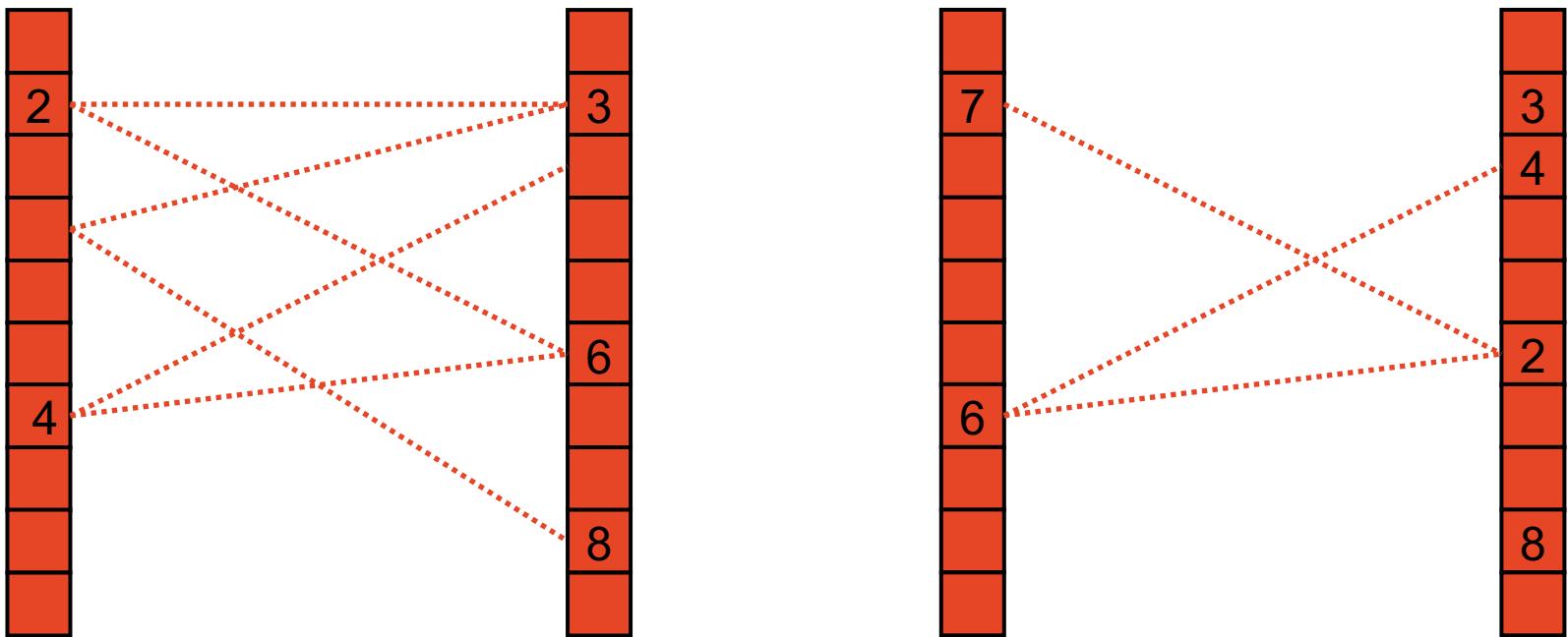
The Common Cuckoo is a brood parasite—it lays its egg in the nest of another bird after first evicting an egg out of that nest.



Catesby, Cockoo of Carolina. The bird is. 18th Century color illustration. Early American bird print. Catesby. Scan of 2 d images in the public domain believed to be free to use without restriction in the US.

Example Eviction Sequence

`put(7)` generated an eviction sequence of length 3:



Pseudo-code for put

```
def put(k, v):
    # try to fit item into T1
    if T1[h1(k)] != null and T1[h1(k)].key == k then
        T1[h1(k)] = (k, v)
        return
    # try to fit item into T2
    if T2[h2(k)] != null and T2[h2(k)].key == k then
        T2[h2(k)] = (k, v)
        return
    # start eviction sequence
    i = 1
repeat
    if Ti[hi(k)] == null then
        Ti[hi(k)] = (k, v)
        return
    temp = Ti[hi(k)]
    Ti[hi(k)] = (k, v)
    (k, v) = temp
    i = 1 if i==2 else 2
until a cycle occurs
rehash elements
```

How to detect eviction cycles

Use a counter to keep track of the number of evictions. If we iterate enough times we are guaranteed to have a cycle.

Keep an additional flag for each entry. Every time we evict an entry, we flag it. After a successful put, we need to unflag the entries flagged.

The details of these strategies are not complicated and are left as an exercise for the tutorials.

Performance of Cuckoo Hashing

One can show that “long eviction sequences” happen with very low probability.

Fact: Assuming hash values are uniformly randomly distributed, expected work of n put operations is $O(n)$ provided $N > 2n$

Fact: Cuckoo hashing achieves worst-case $O(1)$ time for lookups and removals

[See Section 6.4 in GT for a proof]

Another ADT: Set

A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.

Elements of a set are like keys of a map, but without any auxiliary values.

Set ADT

`add(e)`: Adds the element *e* to *S* (if not already present).

`remove(e)`: Removes the element *e* from *S* (if it is present).

`contains(e)`: Returns whether *e* is an element of *S*.

`iterator()`: Returns an iterator of the elements of *S*.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets *S* and *T*:

$$S \cup T = \{e: e \text{ is in } S \text{ or } e \text{ is in } T\},$$

$$S \cap T = \{e: e \text{ is in } S \text{ and } e \text{ is in } T\},$$

$$S - T = \{e: e \text{ is in } S \text{ and } e \text{ is not in } T\}.$$

`addAll(T)`: Updates *S* to also include all elements of set *T*, effectively replacing *S* by $S \cup T$.

`retainAll(T)`: Updates *S* so that it only keeps those elements that are also elements of set *T*, effectively replacing *S* by $S \cap T$.

`removeAll(T)`: Updates *S* by removing any of its elements that also occur in set *T*, effectively replacing *S* by $S - T$.

Set implemented via Map

- Use a Map to store the keys, and ignore the value.
- Allows `contains(k)` to be answered by `get(k)`
- Similarly for add and remove
- Using HashMap for Map, gives main Set operations that usually can be performed in $O(1)$ time.

MultiSet

- Like a Set, but allows duplicates
 - also called a Bag
 - operation **count(e)** says how many occurrences of e in collection
 - **remove(e)** removes ONE occurrence (provided e is in the collection already)
- Implement by Map where the element is the key, and the associated value is the number of occurrences

Practice vs Theory

In practice hash tables implementation are usually fast and people use them **as though** put, get, and remove take $O(1)$ time

The analyses we covered in lecture assume uniformly random hash values, which are not possible to implement in practice. Removing these assumptions is an active area of research well beyond the scope of this class

In theory we do not know of an implementation of hash tables that can perform put, get, and remove $O(1)$ time in the worst case.

So you cannot use such a data structure in your assignments.

Theory of Hashing

There is rich Theory of hashing beyond the basic hashing schemes we covered in class:

- Quadratic probing
- Double hashing
- Perfect hashing
- Universal hash families that are k-wise independent
- Cuckoo hashing with a stash
- Pseudorandom generators
- Cryptographic hash functions
- etc

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Graphs [GT 13.1-3]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



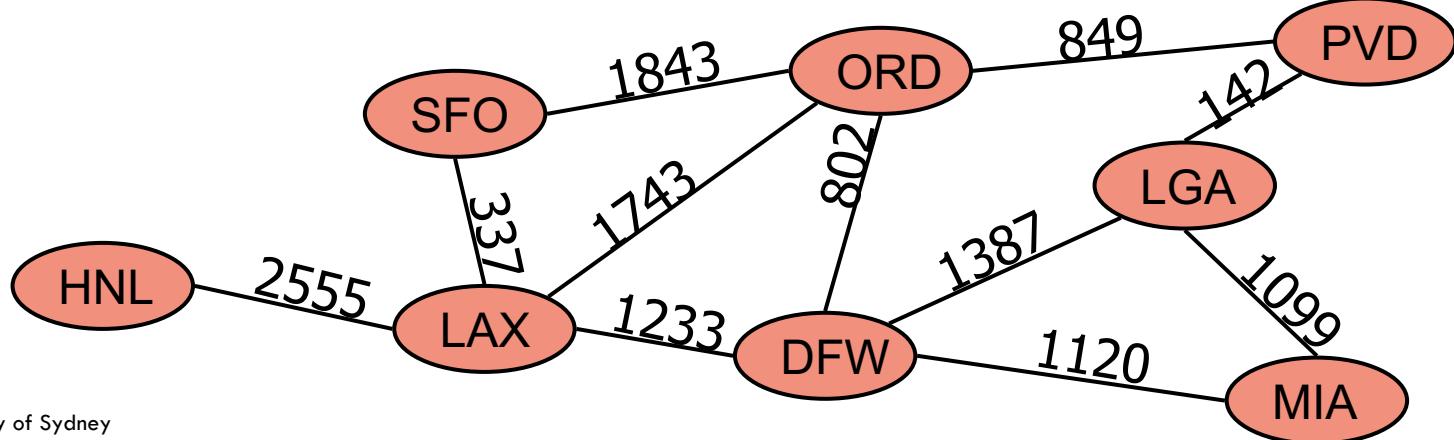
Graphs

A graph **G** is a pair (V, E) , where

- **V** is a set of nodes, called **vertices**
- **E** is a collection of pairs of vertices, called **edges**

Example:

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

Directed edge

- ordered pair of vertices (u, v)
- u is the origin/tail
- v is the destination/head
- e.g., a flight



Undirected edge

- unordered pair of vertices (u, v)
- e.g., a two-way road

Applications

Electronic circuits

- Printed circuit board
- Integrated circuit

Transportation networks

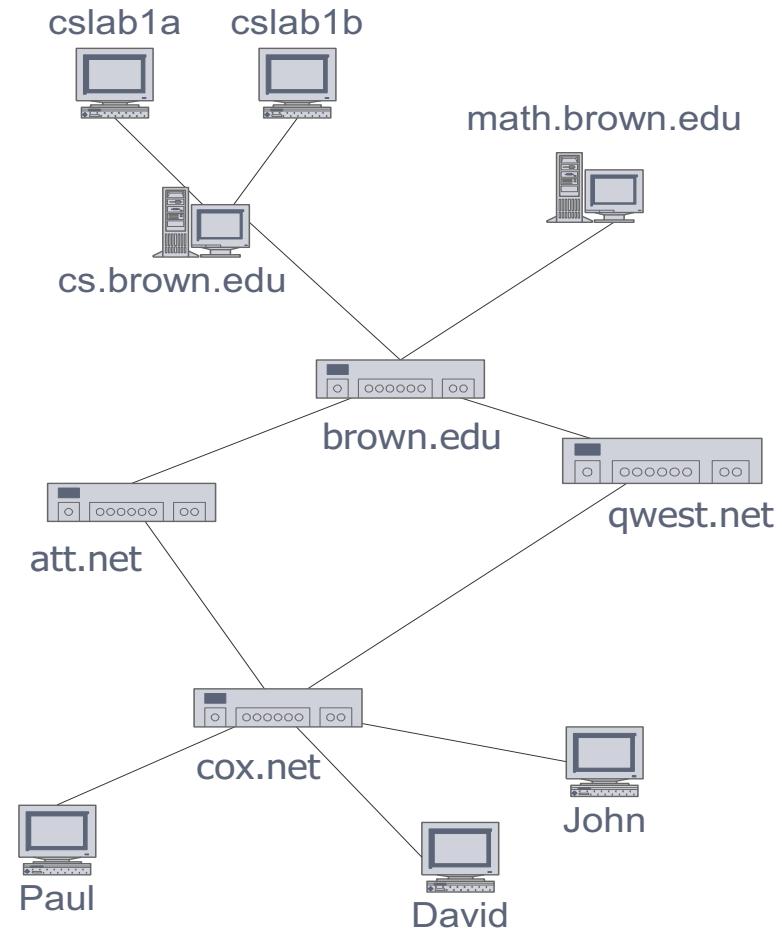
- Highway network
- Flight network

Computer networks

- Internet
- Web

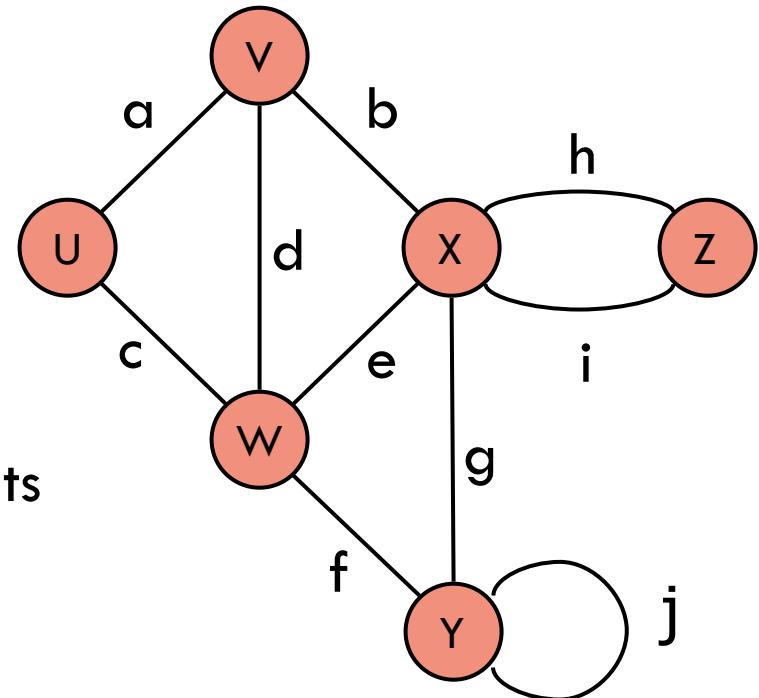
Modeling

- Entity-relationship diagram
- Gantt precedence constraints



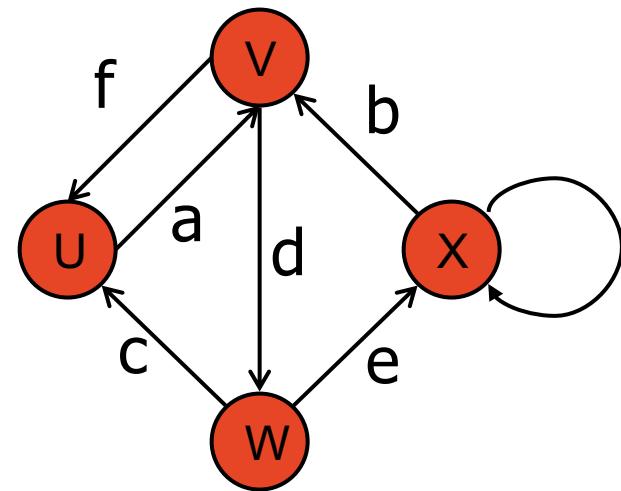
Terminology (Undirected graphs)

- Edges connect **endpoints**
e.g., W and Y for edge f
- Edges are **incident** on endpoints
e.g., a, d, and b are incident on V
- **Adjacent** vertices are connected
e.g., U and V are adjacent
- **Degree** is # of edges on a vertex
e.g., X has degree 5
- **Parallel edges** share same endpoints
e.g., h and i are parallel
- **Self-loop** have only one endpoint
e.g., j is a self-loop
- **Simple** graphs have no parallel or self-loops



Terminology (Directed graphs)

- Edges go from **tail** to **head**
e.g., W is the tail of c and U its head
- **Out-degree** is # of edges out of a vertex
e.g., W has out-degree 2
- **In-degree** is # of edges into a vertex
e.g., W has out-degree 1
- **Parallel edges** share tail and head
e.g., no parallel edge on the right
- **Self-loop** have same head and tail
e.g., X has a self-loop
- **Simple** directed graphs have no parallel or self-loops, but are allowed to have anti-parallel loops like f and a



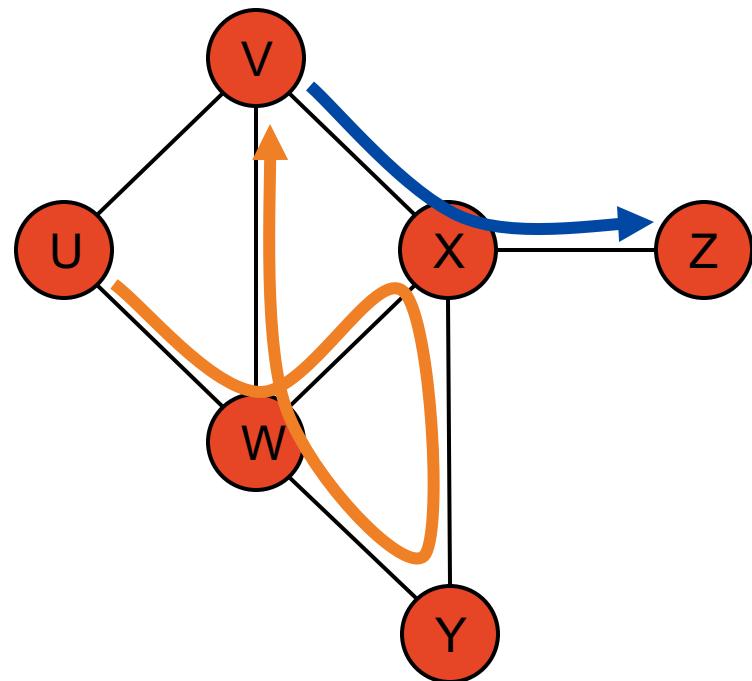
Terminology

A **path** is a sequence of vertices such that every pair of consecutive vertices are connected with an edge.

A simple path is one where all vertices are distinct

Examples

- (V, X, Z) is a simple path
- (U, W, X, Y, W, V) is a path that is not simple



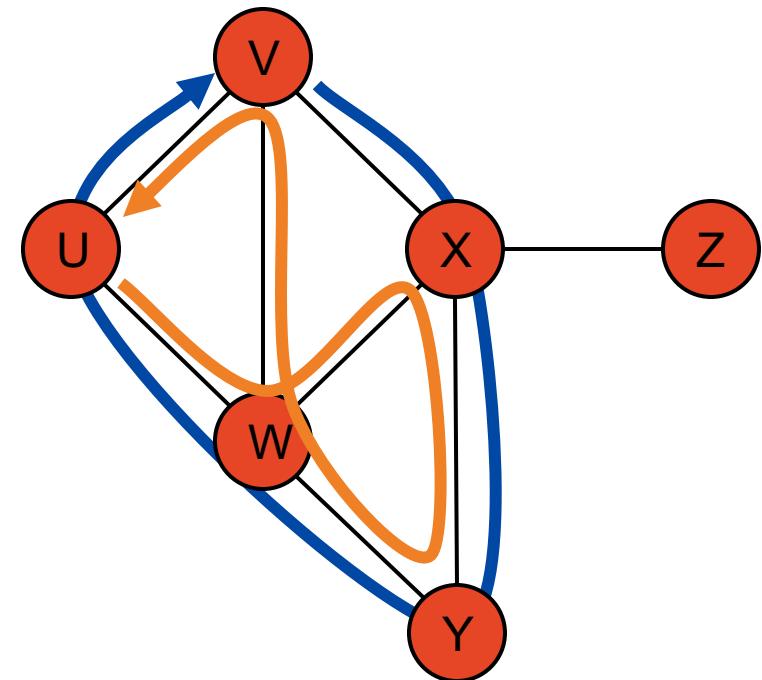
Terminology

A **cycle** is defined by a path that starts and ends at the same vertex

A **simple cycle** is one where all vertices are distinct

Examples

- (V, X, Y, W, U) is a simple cycle
- (U, W, X, Y, W, V) is a cycle that is not simple



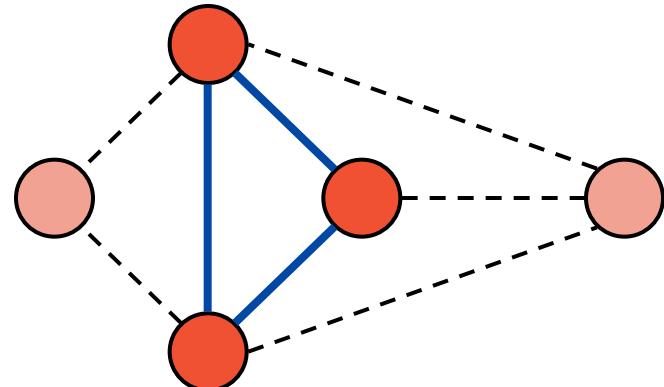
An **acyclic graph** has no cycles

Subgraphs

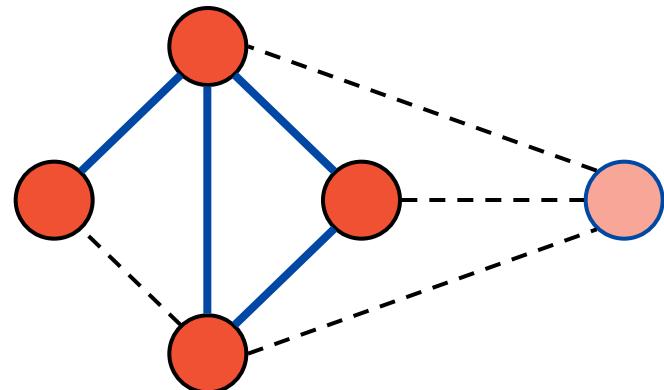
Let $G=(V, E)$ be a graph. We say $S=(U, F)$ is a subgraph of G if $U \subseteq V$ and $F \subseteq E$

A subset $U \subseteq V$ induces a graph $G[U] = (U, E[U])$ where $E[U]$ are the edges in E with endpoints in U

A subset $F \subseteq E$ induces a graph $G[F] = (V[F], F)$ where $V[F]$ are the endpoints of edges in F



Subgraph induced by red vertices

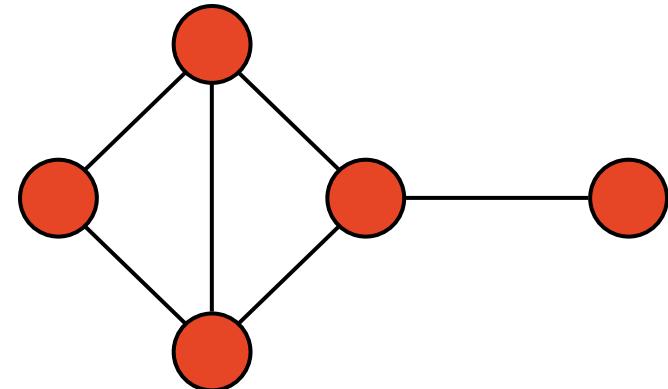


Subgraph induced by blue edges

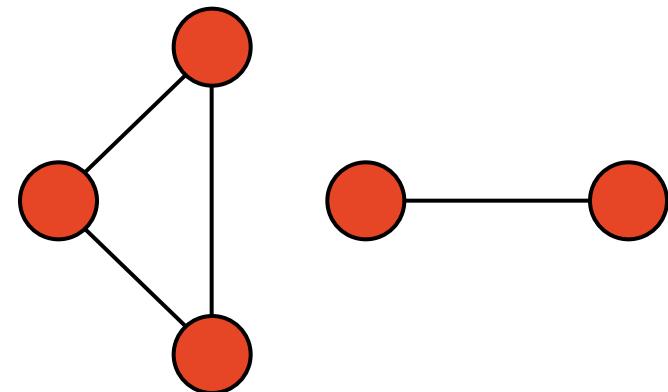
Connectivity

A graph $G=(V, E)$ is connected if there is a path between every pair of vertices in V

A connected component of a graph G is a maximal connected subgraph of G



Connected graph



Graph with two connected components

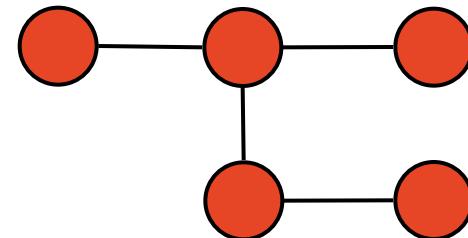
Trees and Forests

An unrooted tree T is a graph such that

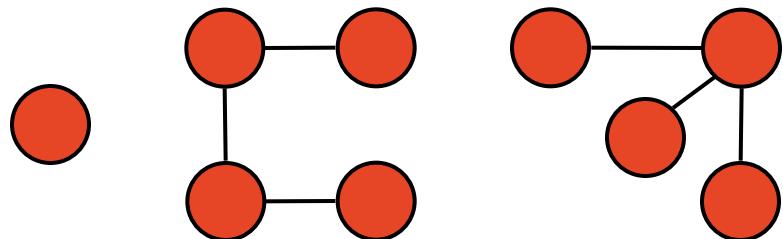
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree



Forest

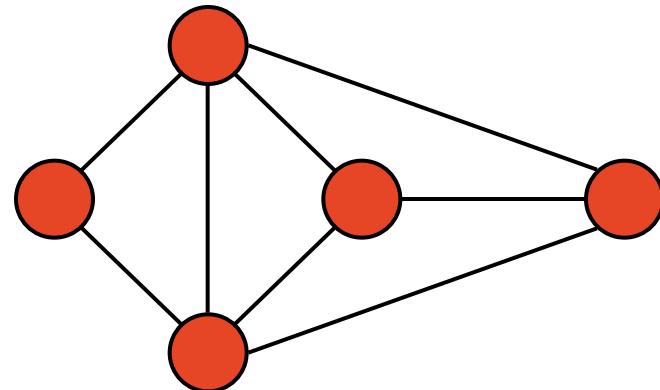
Spanning Trees and Forests

A spanning tree is a connected subgraph on the same vertex set

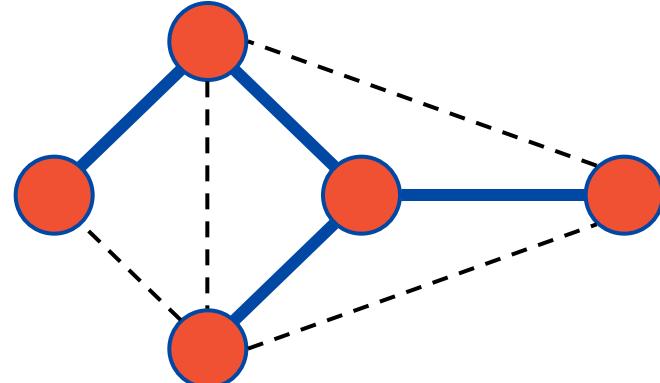
A spanning tree is not unique unless the graph is a tree

Spanning trees have applications to the design of communication networks

A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Properties

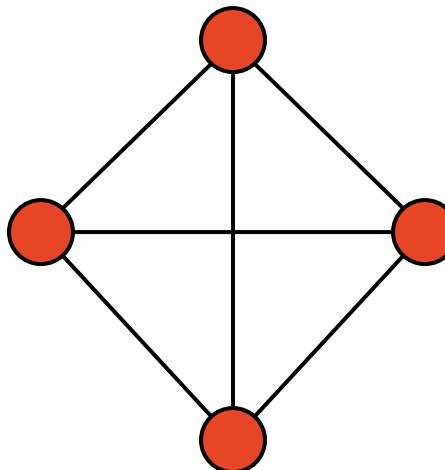
Fact: $\sum_{v \in V} \deg(v) = 2m$

Fact: In a simple undirected graph $m \leq n(n - 1)/2$

Fact: In a simple directed graph $m \leq n(n - 1)$

Notation

n	number of vertices
m	number of edges
Δ	maximum degree



Example: K_4

$$n = 4$$

$$m = 6$$

$$\max \deg = 3$$

Graph ADT

We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.

A Vertex stores an associated object (e.g., an airport code) that is retrieved with a getElement() method.

An Edge stores an associated object (e.g., a flight number, travel distance) that is retrieved with a getElement() method.

Directed Graph ADT

Undirected
Graph
alternatives

degree(v) ←

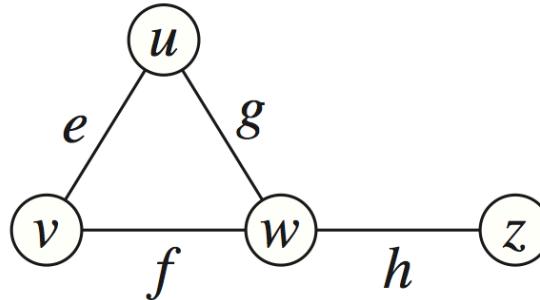
incidentEdges(v) ←

- numVertices()**: Returns the number of vertices of the graph.
- vertices()**: Returns an iteration of all the vertices of the graph.
- numEdges()**: Returns the number of edges of the graph.
- edges()**: Returns an iteration of all the edges of the graph.
- getEdge(u, v)**: Returns the edge from vertex u to vertex v , if one exists; otherwise return null. For an undirected graph, there is no difference between $\text{getEdge}(u, v)$ and $\text{getEdge}(v, u)$.
- endVertices(e)**: Returns an array containing the two endpoint vertices of edge e . If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e)**: For edge e incident to vertex v , returns the other vertex of the edge; an error occurs if e is not incident to v .
- outDegree(v)**: Returns the number of outgoing edges from vertex v .
- inDegree(v)**: Returns the number of incoming edges to vertex v . For an undirected graph, this returns the same value as does $\text{outDegree}(v)$.
- outgoingEdges(v)**: Returns an iteration of all outgoing edges from vertex v .
- incomingEdges(v)**: Returns an iteration of all incoming edges to vertex v . For an undirected graph, this returns the same collection as does $\text{outgoingEdges}(v)$.
- insertVertex(x)**: Creates and returns a new Vertex storing element x .
- insertEdge(u, v, x)**: Creates and returns a new Edge from vertex u to vertex v , storing element x ; an error occurs if there already exists an edge from u to v .
- removeVertex(v)**: Removes vertex v and all its incident edges from the graph.
- removeEdge(e)**: Removes edge e from the graph.

Edge List Structure

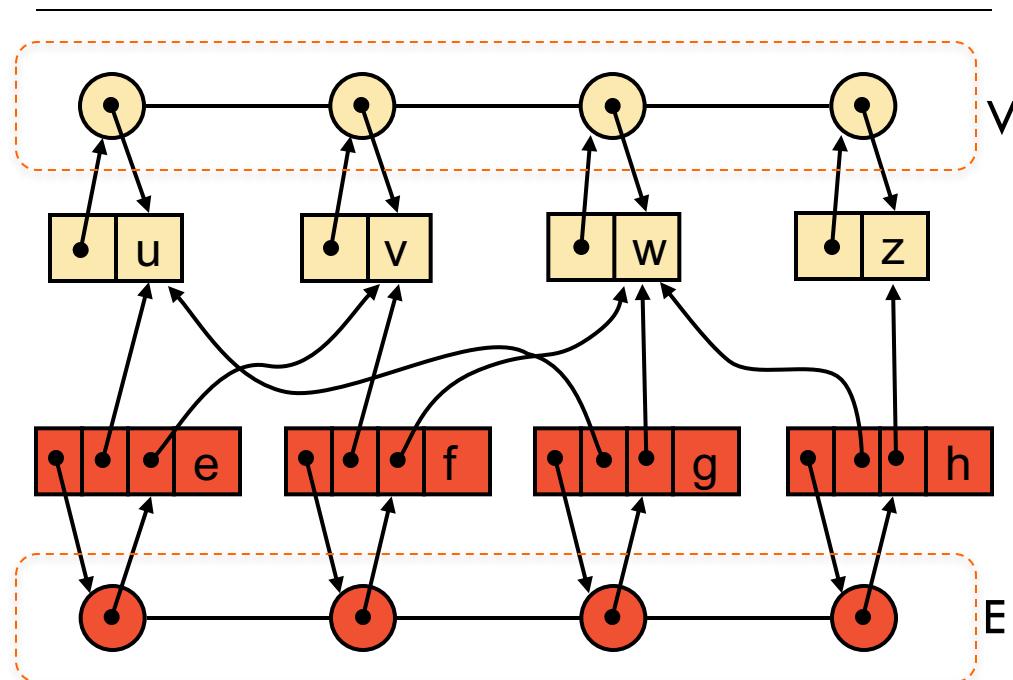
Vertex sequence holds

- sequence of vertices
- vertex objects keeps track of its position in the sequence



Edge sequence

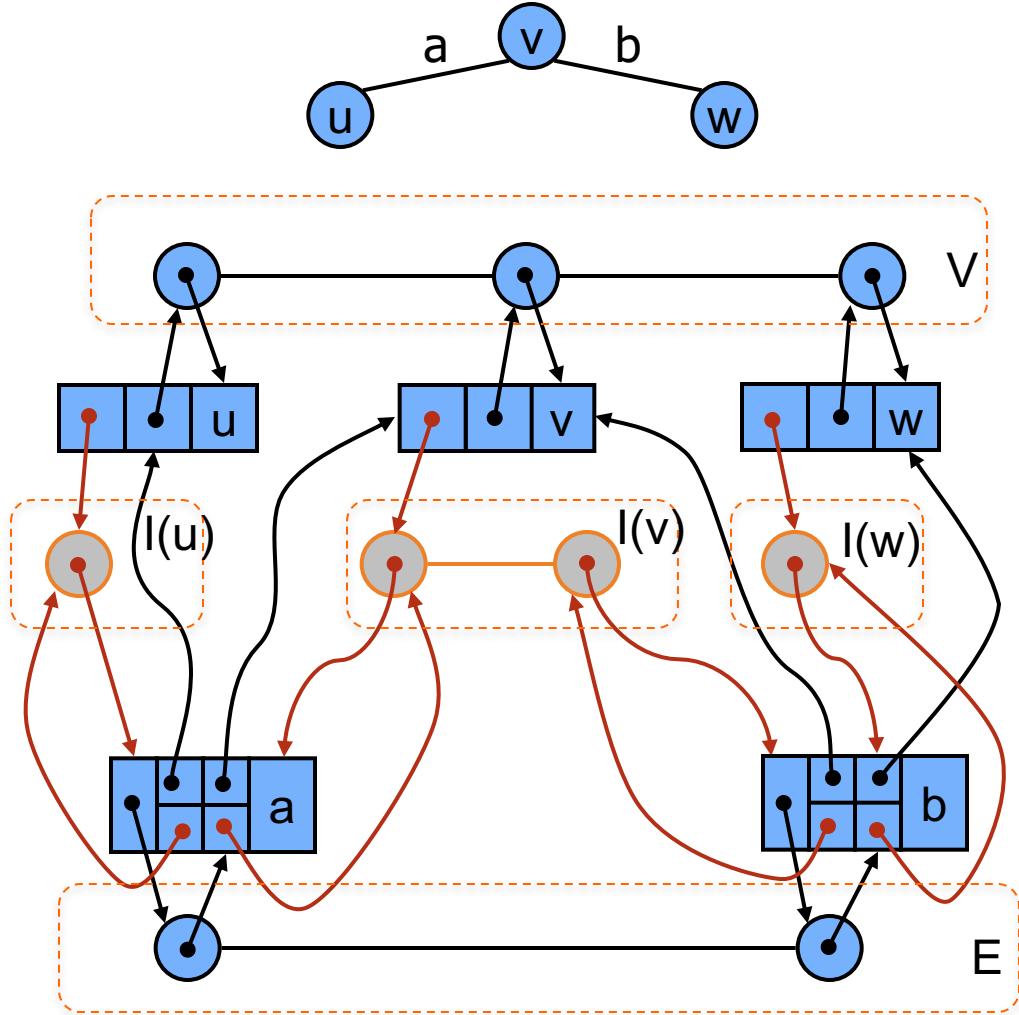
- sequence edges
- edge object keeps track of its position in the sequence
- Edge object points to the two vertices it connects



Adjacency List

Additionally each vertex keeps a sequence of edges incident on it

Edge objects keep reference to their position in the incidence sequence of its endpoints

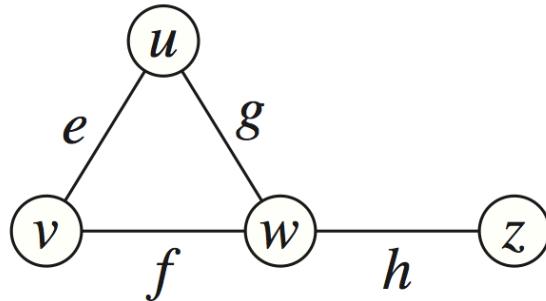


Adjacency Matrix Structure

Vertex array induces an index from 0 to n-1 for each vertex

2D-array adjacency matrix

- Reference to edge object for adjacent vertices
- Null for nonadjacent vertices



	0	1	2	3
$u \rightarrow$	0	e	g	
$v \rightarrow$	e		f	
$w \rightarrow$	g	f		h
$z \rightarrow$			h	

Asymptotic performance

<ul style="list-style-type: none"> ■ n vertices, m edges ■ no parallel edges ■ no self-loops 	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
<code>incidentEdges(v)</code>	$O(m)$	$O(\deg(v))$	$O(n)$
<code>getEdge(u, v)</code>	$O(m)$	$O(\min(\deg(u), \deg(v)))$	$O(1)$
<code>insertVertex(x)</code>	$O(1)$	$O(1)$	$O(n^2)$
<code>insertEdge(u, v, x)</code>	$O(1)$	$O(1)$	$O(1)$
<code>removeVertex(v)</code>	$O(m)$	$O(\deg(v))$	$O(n^2)$
<code>removeEdge(e)</code>	$O(1)$	$O(1)$	$O(1)$

Graph traversals

A fundamental kind of algorithmic operation that we might wish to perform on a graph is **traversing the edges and the vertices** of that graph.

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

For example, a **web crawler**, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.

A traversal is efficient if it visits all the vertices and edges in linear time: **$O(n+m)$** where **n**=number of vertices, **m**=number of edges.

Graph traversal techniques

A systematic and structured way of visiting all the vertices and all the edges of a graph

Two main strategies:

- Depth first search
- Breadth first search

Given adjacency list representation of the graph with n vertices and m edges both traversal run in $O(n + m)$ time

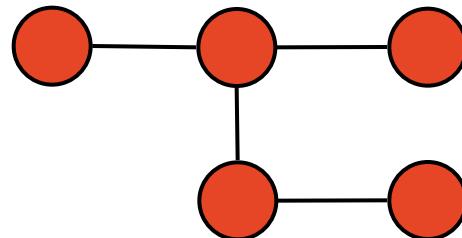
Reminder: Trees and Forests

An unrooted tree T is a graph such that

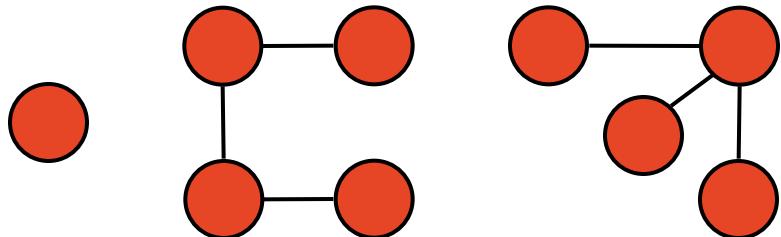
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree

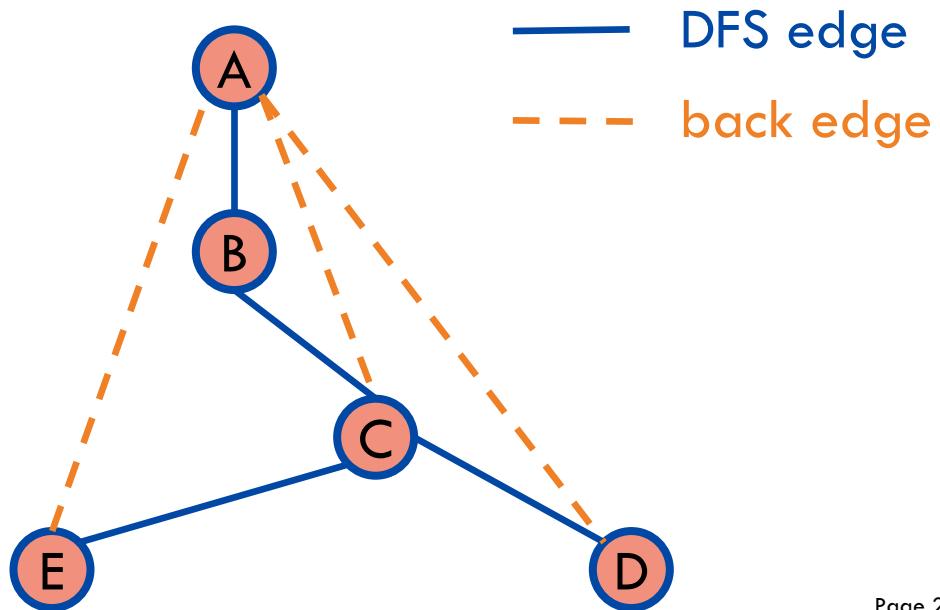
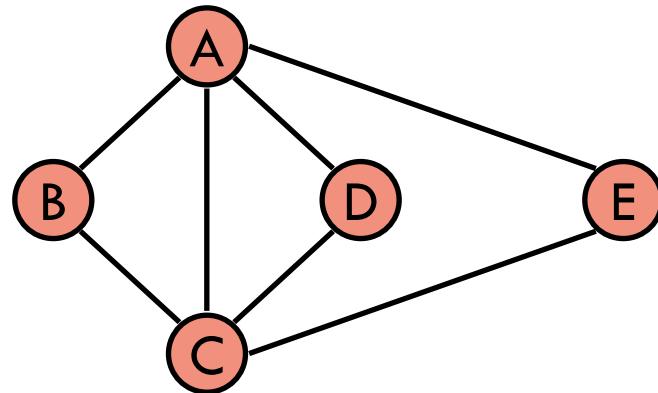


Forest

Depth-First Search (DFS) for undirected graphs

This strategy tries to follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if “stuck”

If an edge is used to discover a new vertex, we call it a DFS edge, otherwise we call it a back edge



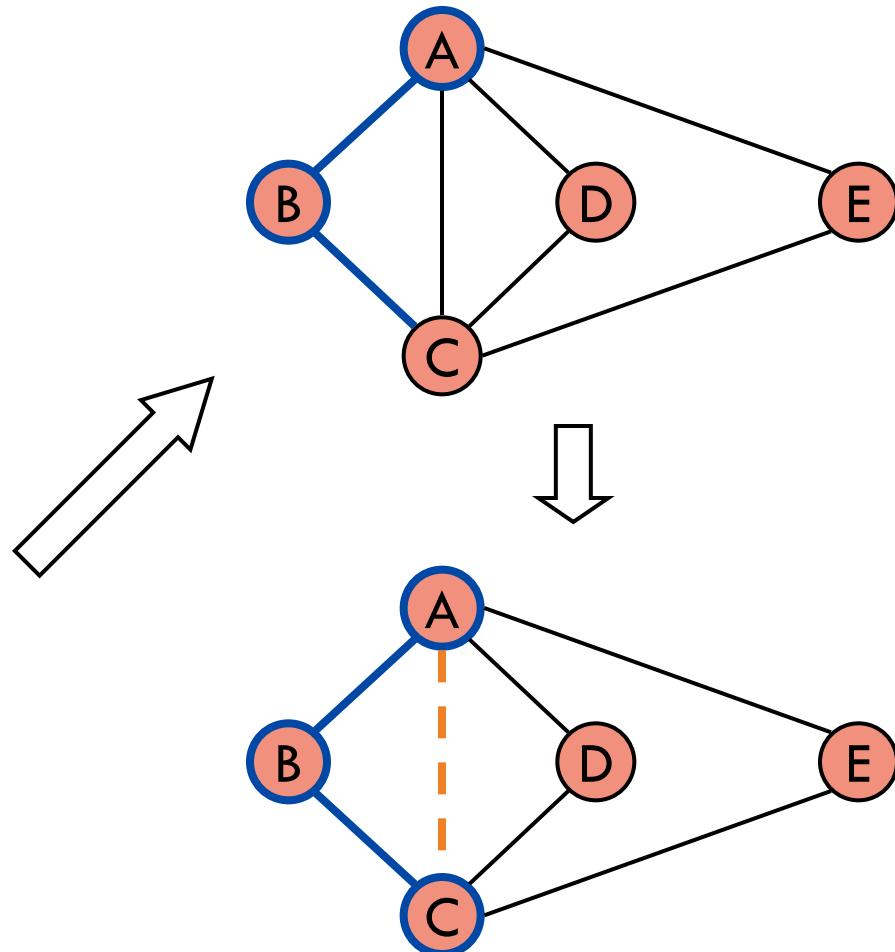
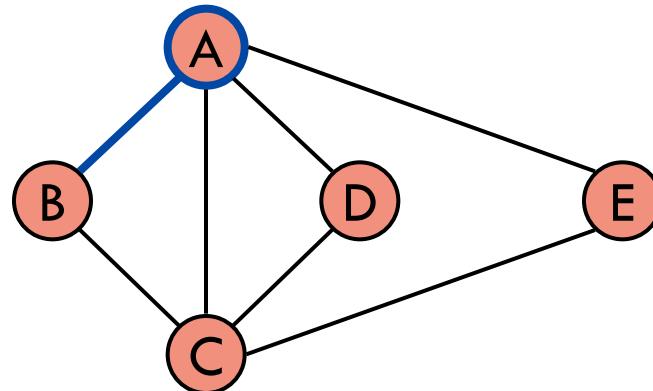
DFS pseudocode

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices(): do  
        visited[u] = False  
        parent[u] = None  
  
    # visit vertices  
    for u in G.vertices(): do  
        if not visited[u]: then  
            DFS_visit(u)  
  
    return parent
```

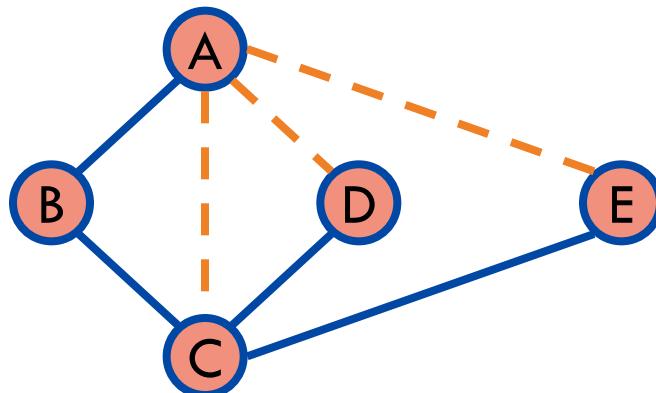
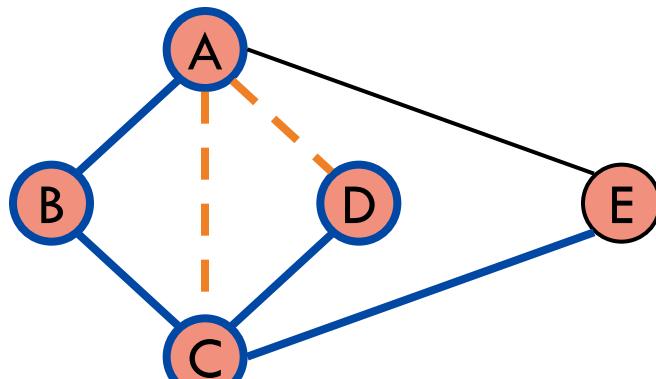
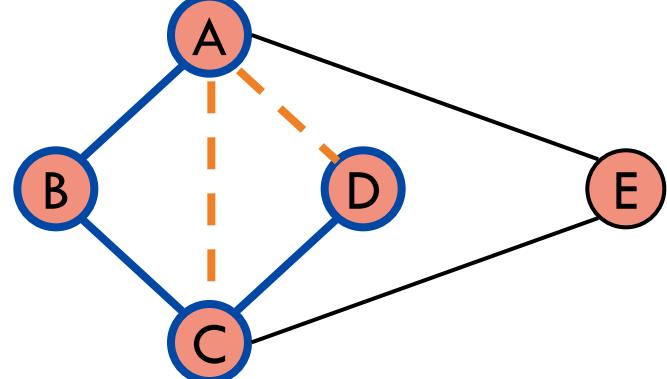
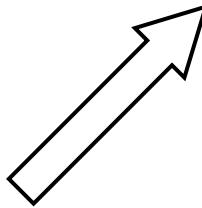
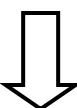
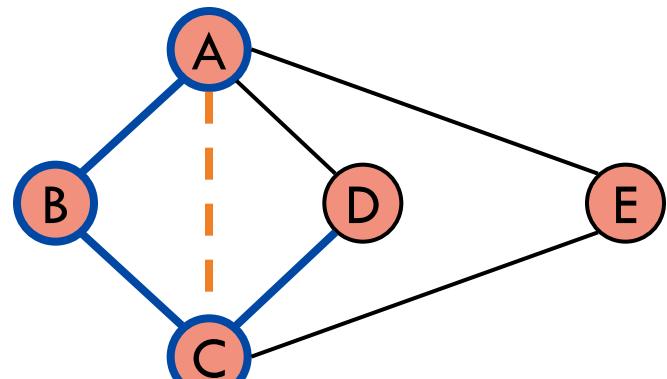
```
def DFS_visit(u):  
  
    visited[u] = True  
  
    # visit neighbors of u  
    for v in G.incident(u): do  
        if not visited[v]: then  
            parent[v] = u  
            DFS_visit(v)
```

Example

- unexplored vertex
- visited vertex
- unexplored edge
- DFS edge
- - - back edge



Example (cont.)



DFS main function performance

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices():  
        visited[u] = False  
        parent[u] = None  
  
    # visit vertices  
    for u in G.vertices():  
        if not visited[u]:  
            DFS_visit(u)  
  
    return parent
```

Assuming adjacency list representation

$O(n)$ time

$O(n)$ time not counting work done in `DFS_visit`

DFS_visit performance

Assuming adjacency list representation

$O(\deg(u))$ time not counting work done in recursive calls to DFS_visit

Thus, overall time is

$$O(\sum_u \deg(u)) = O(m)$$

```
def DFS_visit(u):
    visited[u] = True
    # visit neighbors of u
    for v in G.incident(u):
        if not visited[v]:
            parent[v] = u
            DFS_visit(v)
```

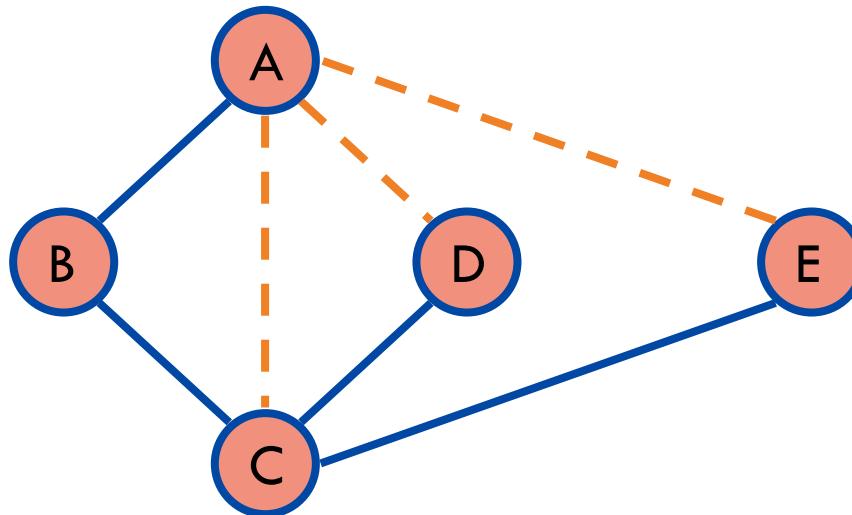
Properties of DFS

Let C_v be the connected component of v in our graph G

Fact: $\text{DFS_visit}(v)$ visits all vertices in C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \text{ in } C_v \}$ form a spanning tree of C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \text{ in } V \}$ form a forest of G



DFS Applications

DFS can be used to solve other graph problems in $O(n + m)$ time:

- Find a path between two given vertices, if any
- Find a cycle in the graph
- Test whether a graph is connected
- Compute connected components of a graph
- Compute spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- testing bi-connectivity
- finding cut edges
- finding cut vertices

Identifying cut edges

In a connected graph $G=(V, E)$, we say that an edge (u, v) in E is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected

The cut edge problem is to identify all cut edges

Trivial $O(m^2)$ time algorithm: For each edge (u, v) in E , remove (u, v) and check if DFS is still connected, put back (u, v)

Better $O(nm)$ time algorithm: Only test edges in a DFS tree of G

Identifying cut edges in $O(n+m)$ time

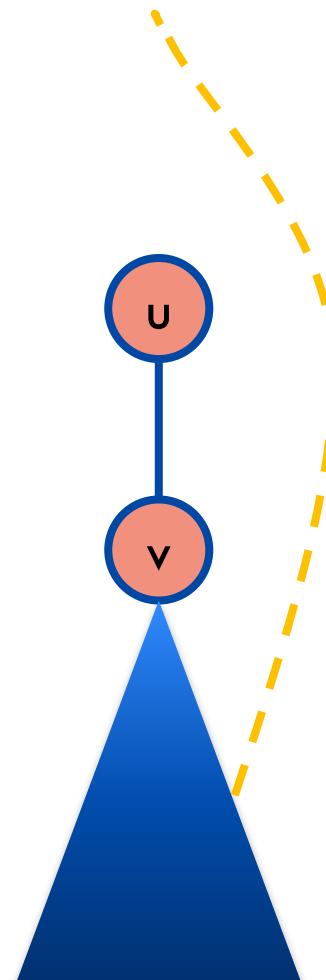
Compute a DFS tree of the input graph $G=(V, E)$

For every u in V , compute $\text{level}[u]$, its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this $\text{down_and_up}[v]$

Fact: A DFS edge (u, v) where $u = \text{parent}[v]$ is not a cut edge if and only if $\text{down_and_up}[v] \leq \text{level}[u]$

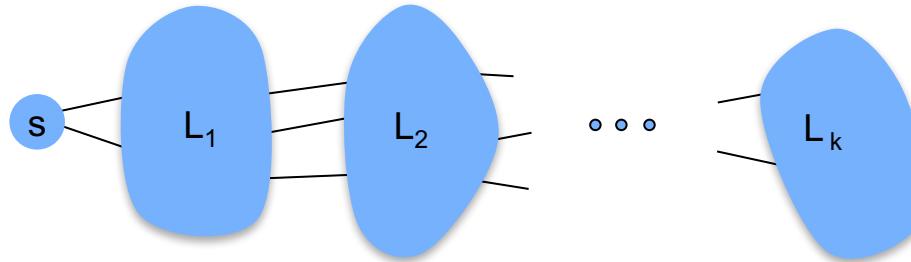
Basis of an $O(n+m)$ time algorithm for finding cut edges



Breadth-First Search (BFS)

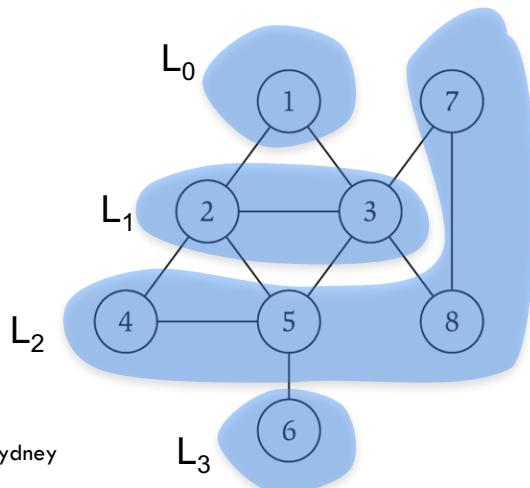
This strategy tries to visit all vertices at distance k from a start vertex s before visiting vertices at distance $k + 1$:

- $L_0 = \{s\}$
- $L_1 = \text{vertices one hop away from } s$
- $L_2 = \text{vertices two hops away from } s \text{ but no closer}$
- :
⋮
- $L_k = \text{vertices } k \text{ hops away from } s \text{ but no closer}$



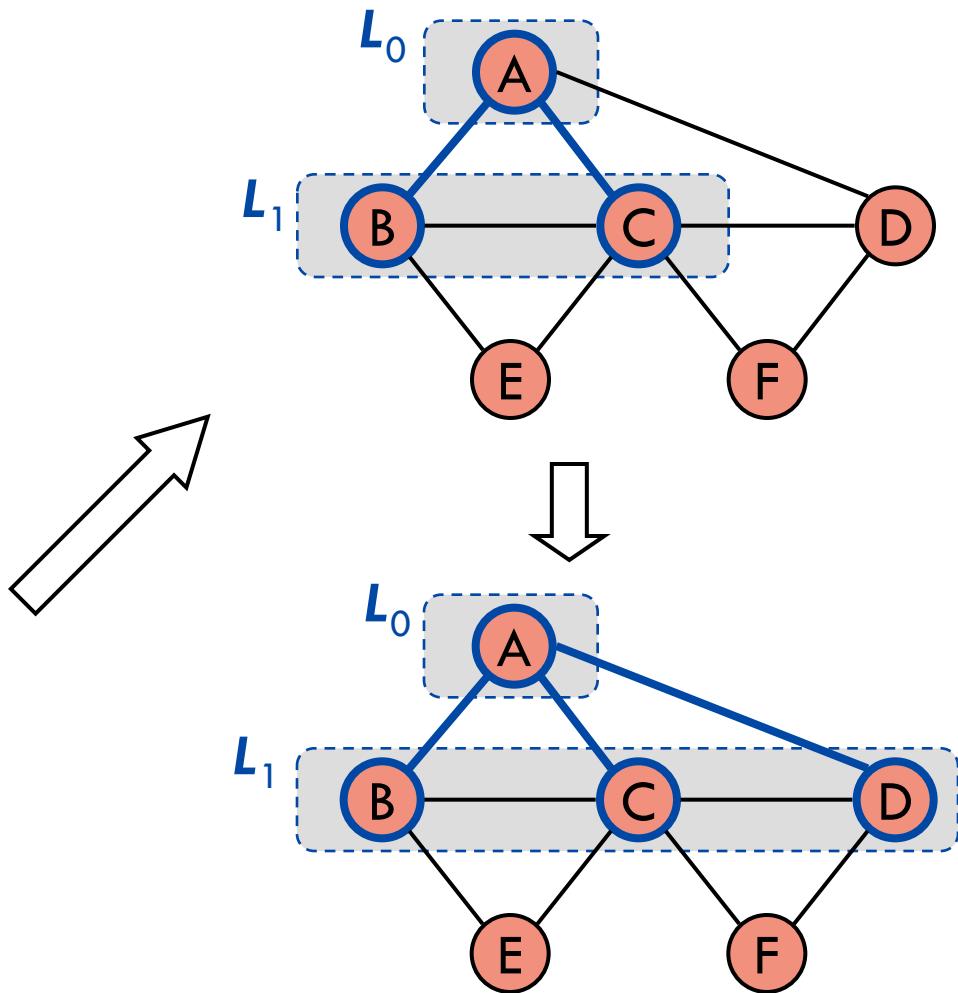
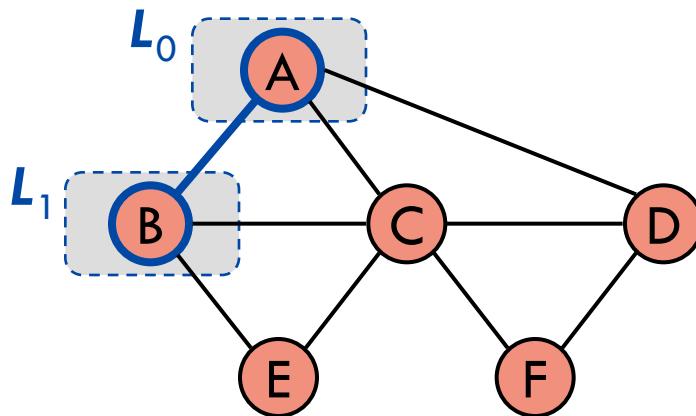
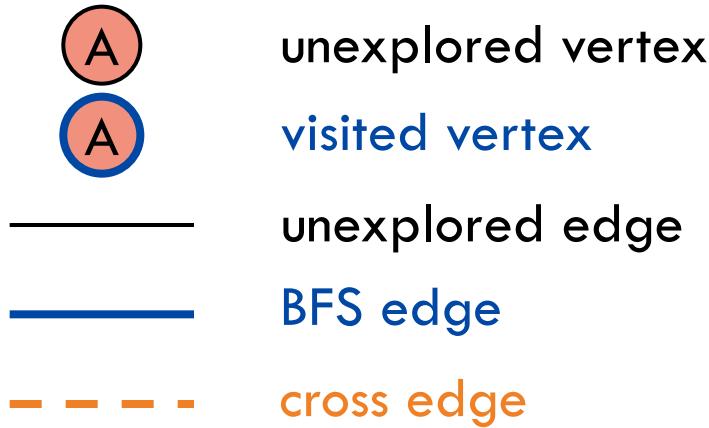
BFS

```
def BFS(G,s):  
  
    # set things up for BFS  
    for u in G.vertices():  
        seen[u] = False  
        parent[u] = None  
  
    seen[s] = True  
    layers = []  
    current = [s]  
    next = []
```

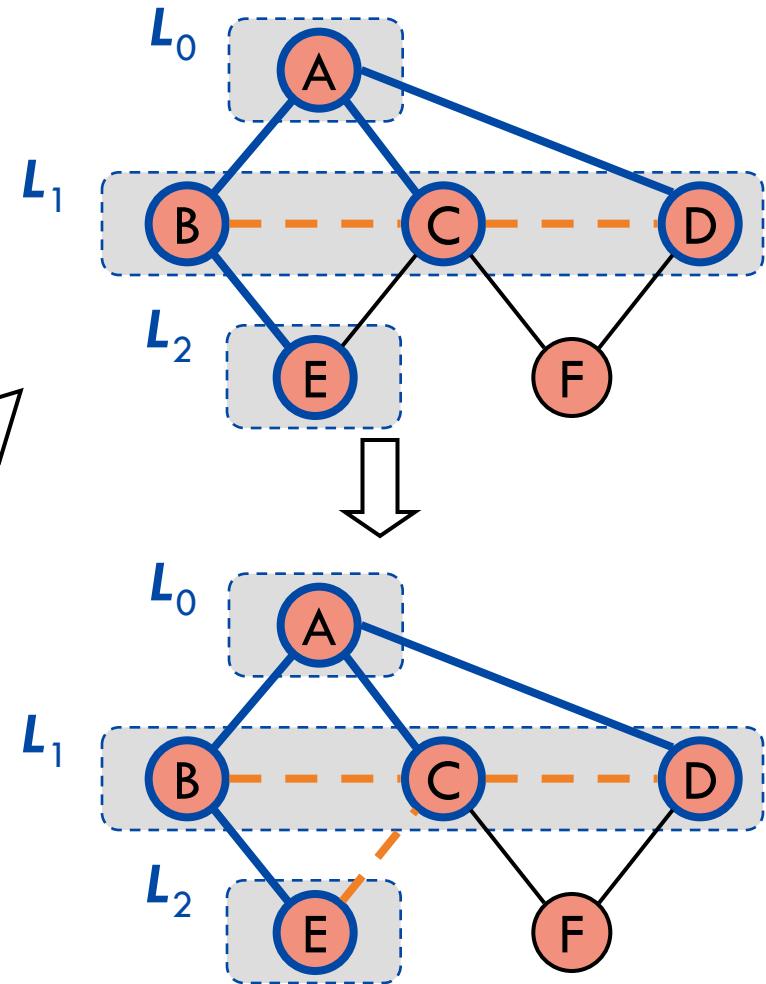
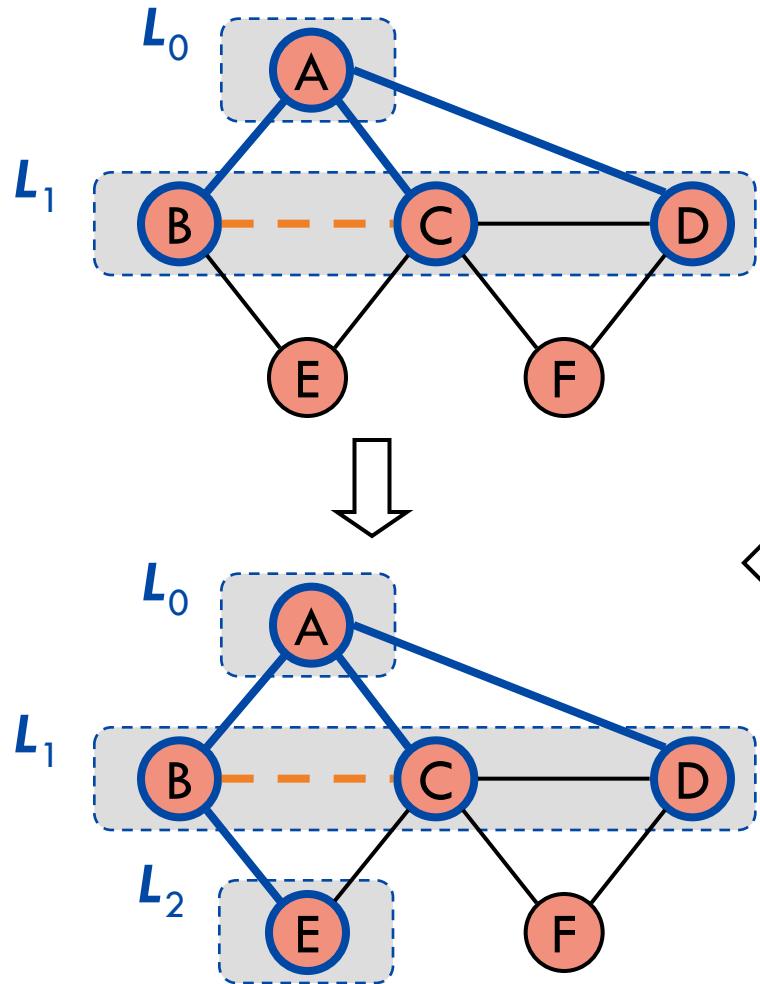


```
# process current layer  
while not current.is_empty():  
    layers.append(current)  
    # iterate over current layer  
    for u in current:  
        for v in G.incident(u):  
            if not seen[v]:  
                next.append(v)  
                seen[v] = True  
                parent[v] = u  
  
    # update current & next layers  
    current = next  
    next = []  
  
return layers, parent
```

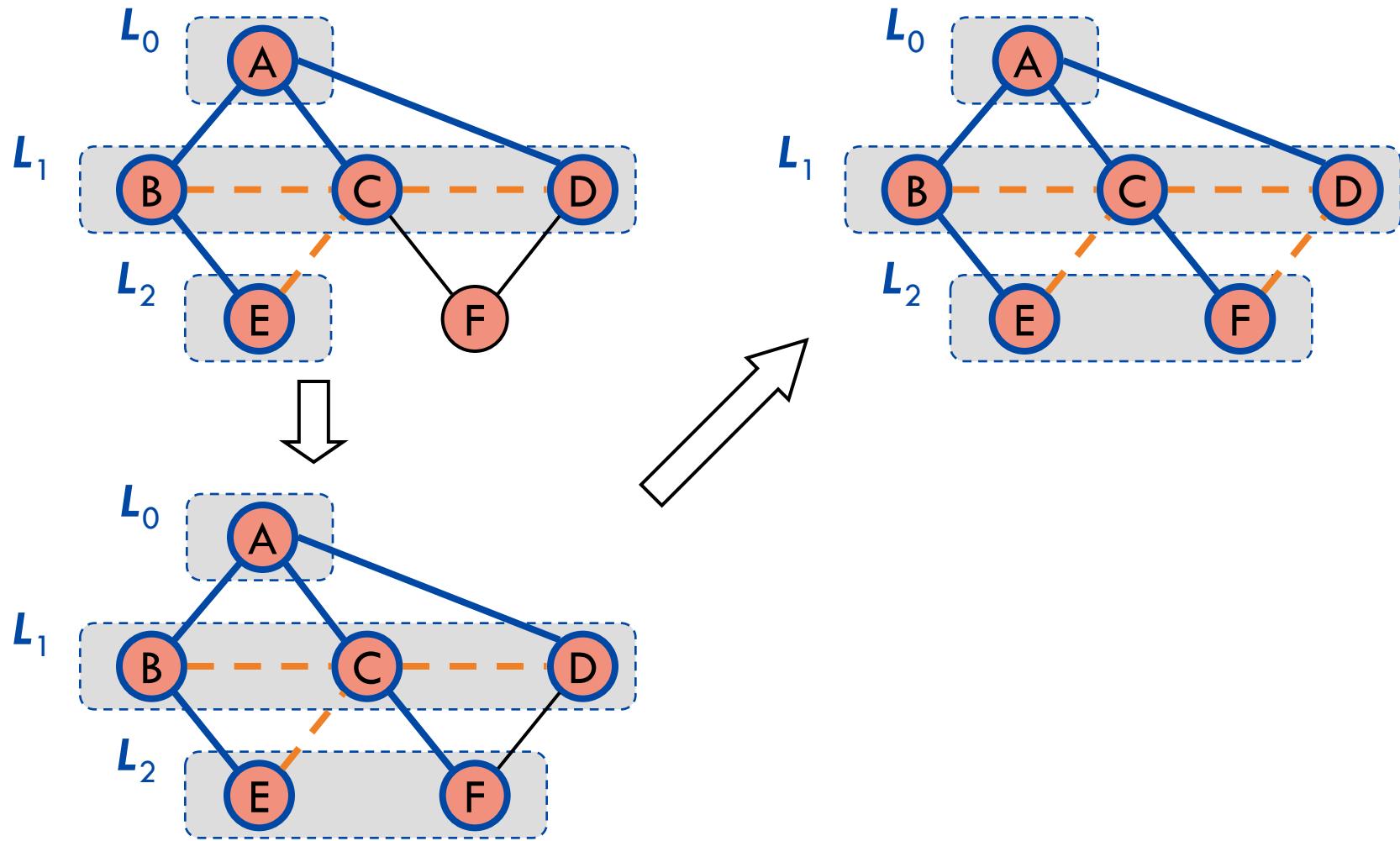
Example



Example (cont.)



Example (cont.)



Properties

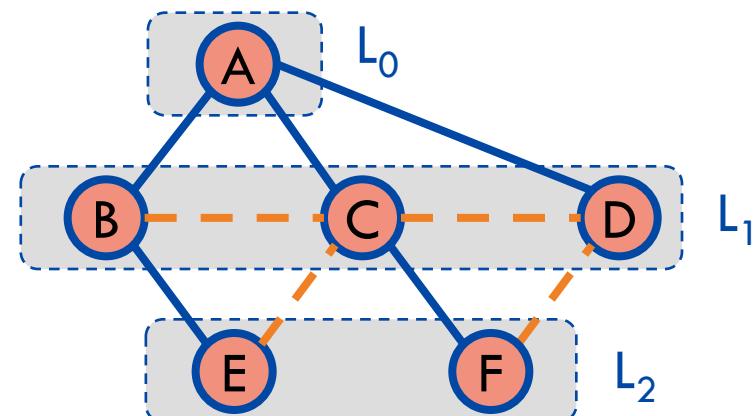
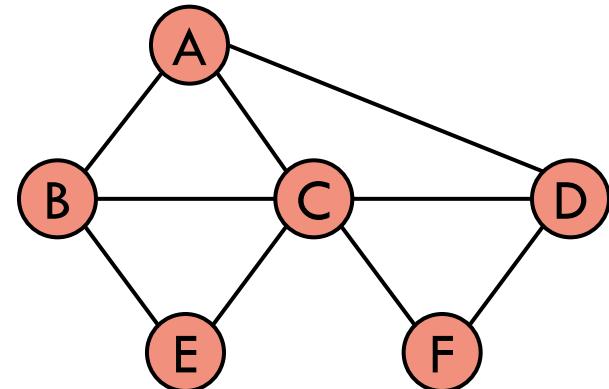
Let C_v be the connected component of v in our graph G

Fact: $\text{BFS}(G, v)$ visits all vertices in C_s

Fact: Edges $\{ (u, \text{parent}[u]): u \in C_s \}$ form a spanning tree T_s of C_s

Fact: For each $v \in L_i$ there is a path in T_s from s to v with i edges

Fact: For each $v \in L_i$ any path in G from s to v has at least i edges



BFS performance

```
def BFS(G, s):  
  
    # set things up for BFS  
    for u in G.vertices() do  
        seen[u] = False  
        parent[u] = None  
  
    seen[s] = True  
    layers = []  
    current = [s]  
    next = []  
  
    # process current layer  
    while not current.is_empty() do  
        layers.append(current)  
        # iterate over current layer  
        for u in current do  
            for v in G.incident(u) do  
                if not seen[v] then  
                    next.append(v)  
                    seen[v] = True  
                    parent[v] = u  
  
    # update curr and next layers  
    current = next  
    next = []  
  
    return layers
```

$\mathcal{O}(n)$ time

$\mathcal{O}(\sum_u \deg(u)) = \mathcal{O}(m)$ time

BFS performance

Fact: Assuming adjacency list representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n+m)$ time

Fact: Assuming adjacency matrix representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n^2)$ time

The additional attributes about the vertices (seen and parent) can be associated directly via Vertex class or we can use an external map data structure

BFS Applications

BFS can be used to solve other graph problems in $O(n + m)$ time:

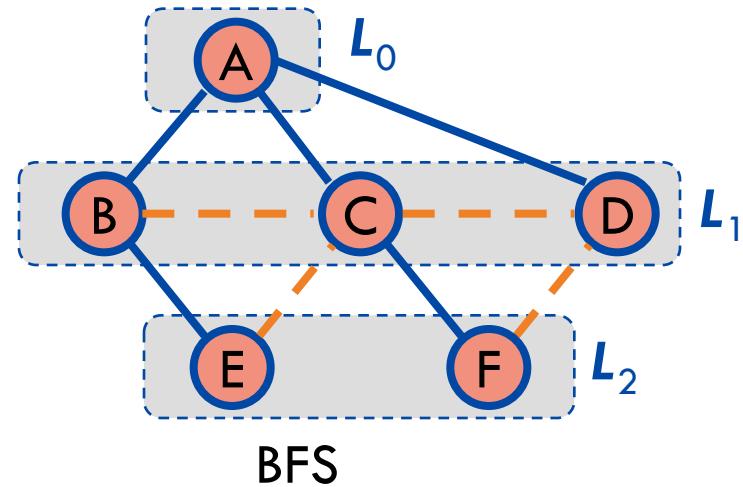
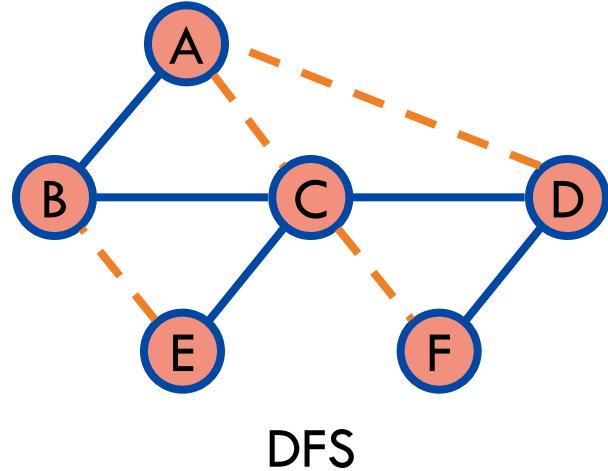
- Find a shortest path between two given vertices
- Find a cycle in the graph
- Test whether a graph is connected
- Compute a spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- Testing if graph is bipartite

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	

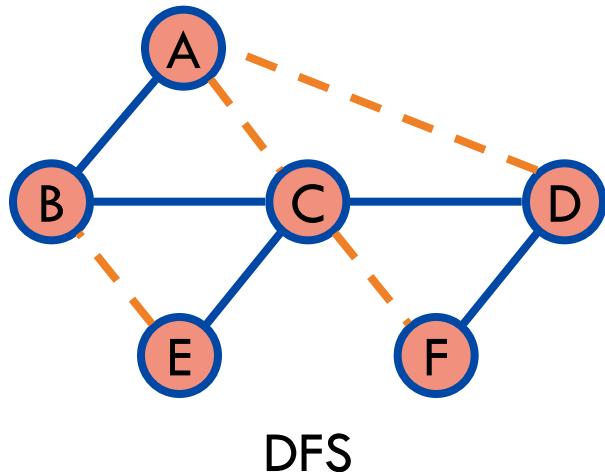


DFS vs. BFS (cont.)

Non-tree DFS edge (v, w)

w is an ancestor of v
in the DFS tree

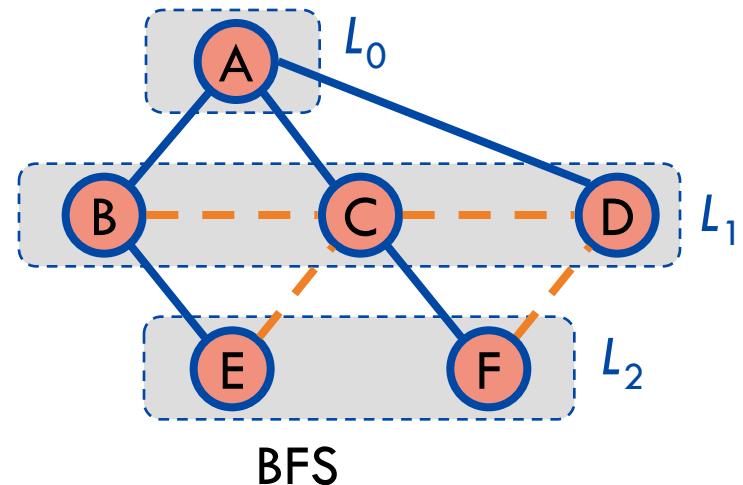
Called back edges



Non-tree BFS edge (v, w)

w is in the same level as v or
in the next level

Called cross edges



Directed Graphs

Both DFS and BFS can be adapted to run in directed graphs:
When visiting vertex **u**, iterate over edges out of **u**

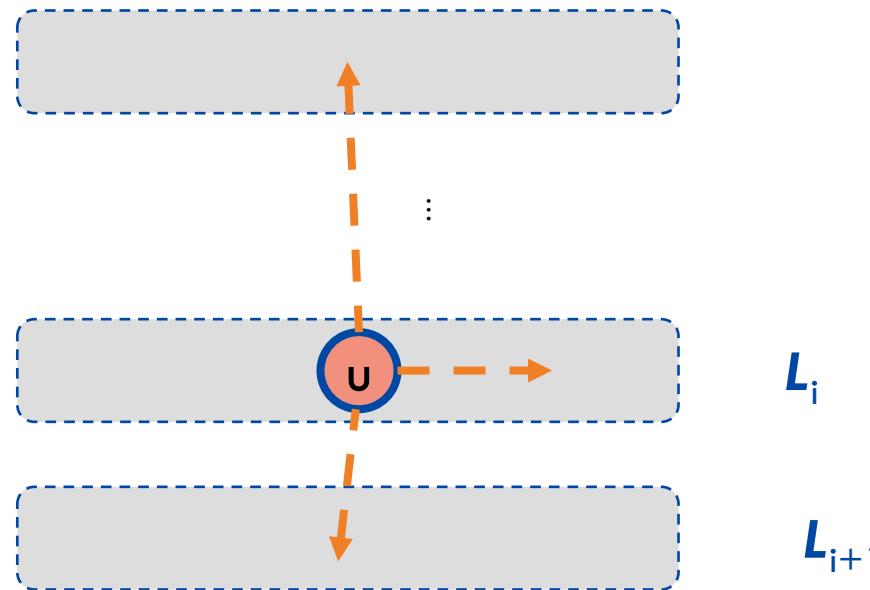
Both algorithm run in $O(n + m)$ time.

However, there are differences on the type of non-tree edges we can have.

BFS on Directed Graphs

Let (u, v) be a non-tree edges (i.e., we did not use the edge to discover v while visiting u)

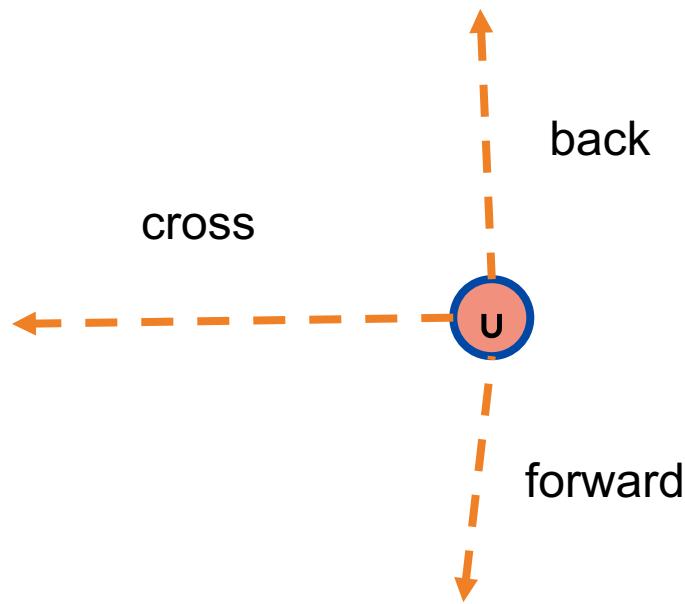
Fact: If u and v belong to layers L_i and L_j respectively, then $j \leq i+1$



DFS on Directed Graphs

Let (u, v) be a non-tree edges (i.e., we did not use the edge to discover v while visiting u)

Fact: The edge either goes up (**back**) or down the tree (**forward**) or side-ways to the left (**cross**)



Topological Sort

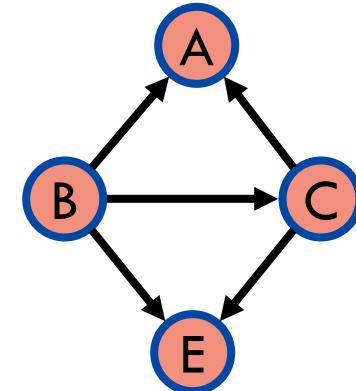
Directed Acyclic Graphs (DAG) have no directed cycles

Every DAG can be topologically sorted: Vertices can be laid out from left to right in such all edges go left to right as well.

For example, consider the graph on the right.

It admits two topological sorts:

- [B, C, A, E]
- [B, C, E, A]



DFS based topological sort

```
def topo(G):  
  
    run DFS on G  
    for (u, v) in G.edges() do  
        if (u, v) is a back edge then  
            return "G is not acyclic"  
  
    ans = [ u for u in G.nodes()]  
    sort ans in opposite order of DFS-visit call finish  
  
    return ans
```

For the correctness,
suppose **u** comes after **v** in **ans**
but $(u, v) \in E$



DFS based topological sort running time

```
def topo(G):  
  
    run DFS on G  
    for (u, v) in G.edges() do  
        if (u, v) is a back edge then  
            return "G is not acyclic"  
  
    ans = [ u for u in G.nodes()]  
    sort ans in opposite order of DFS-visit finish  
  
    return ans
```

To implement the algorithm efficiently, we can augment DFS-visit to perform the back edge check and build ans on the fly without increasing the time complexity of $O(n + m)$

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 8: Shortest Paths and Minimum Spanning Trees [GT 14.1-2, 15.1-3]

Dr. Julian Mestre
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.



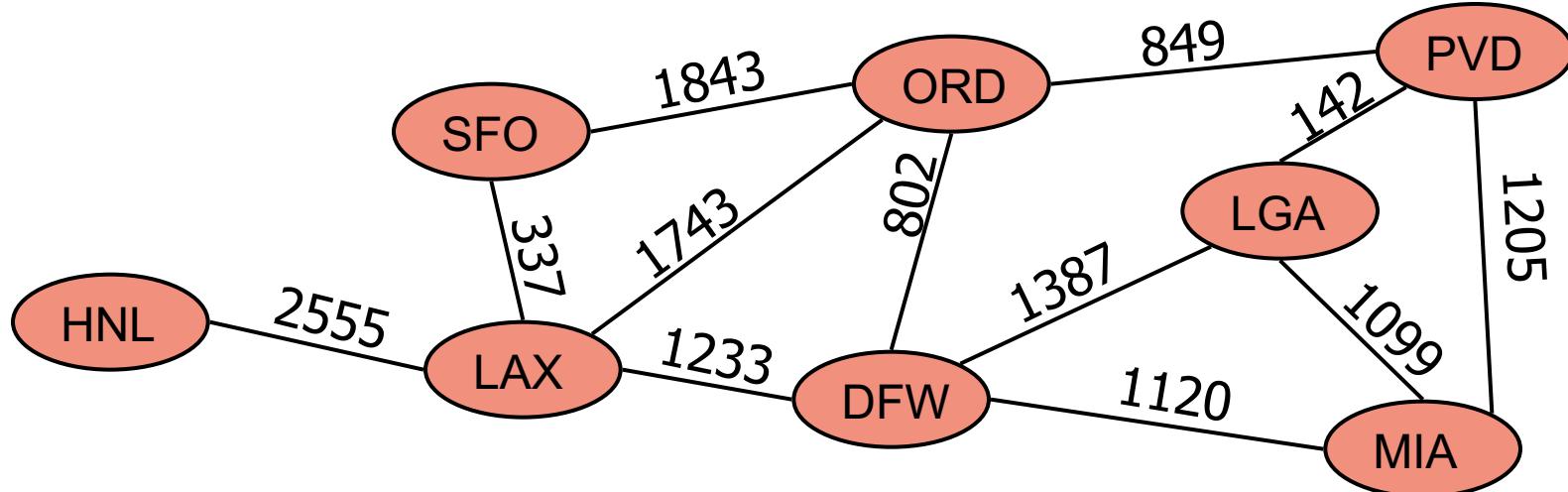
THE UNIVERSITY OF
SYDNEY



Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.

Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Greedy algorithms

A class of algorithms where we build a solution one step at a time making locally optimal choices at each stage in the hope of finding a global optimum solution

Some of the most elegant algorithms and the simplest to implement, but often among the hardest to design and analyze

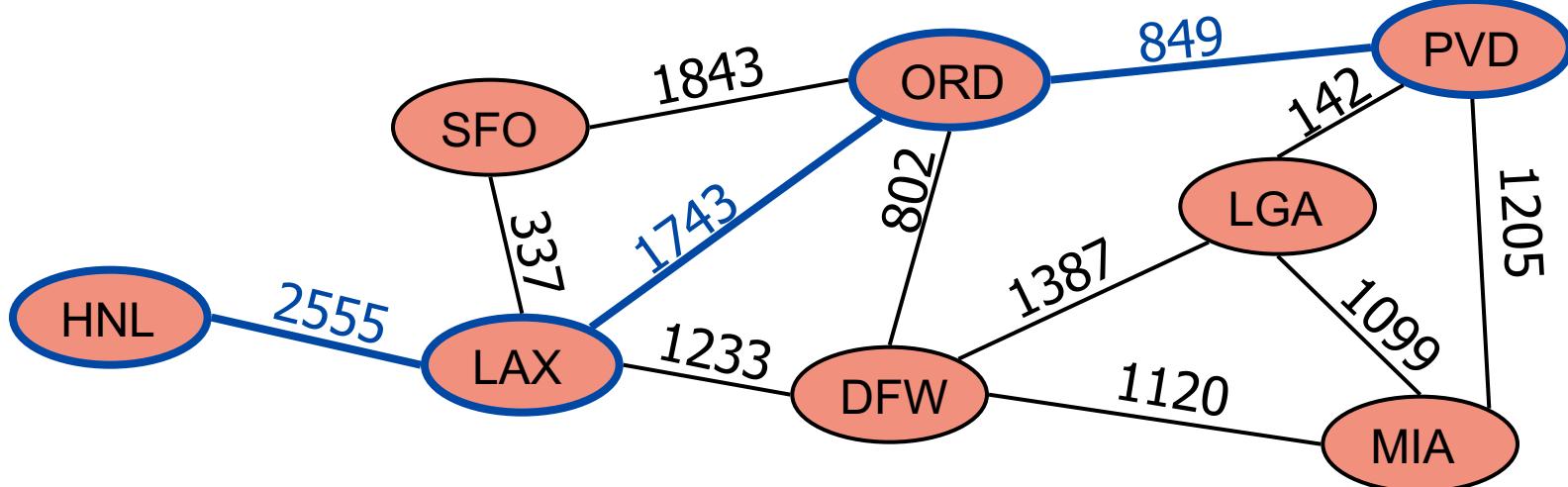
Even when they are not optimal in theory, greedy algorithms can be the basis of a very good heuristic.

Shortest Paths

Given an edge weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v , where the weight of a path is the sum of the weights of its edges).

Applications: Internet packet routing, flight reservations and driving directions.

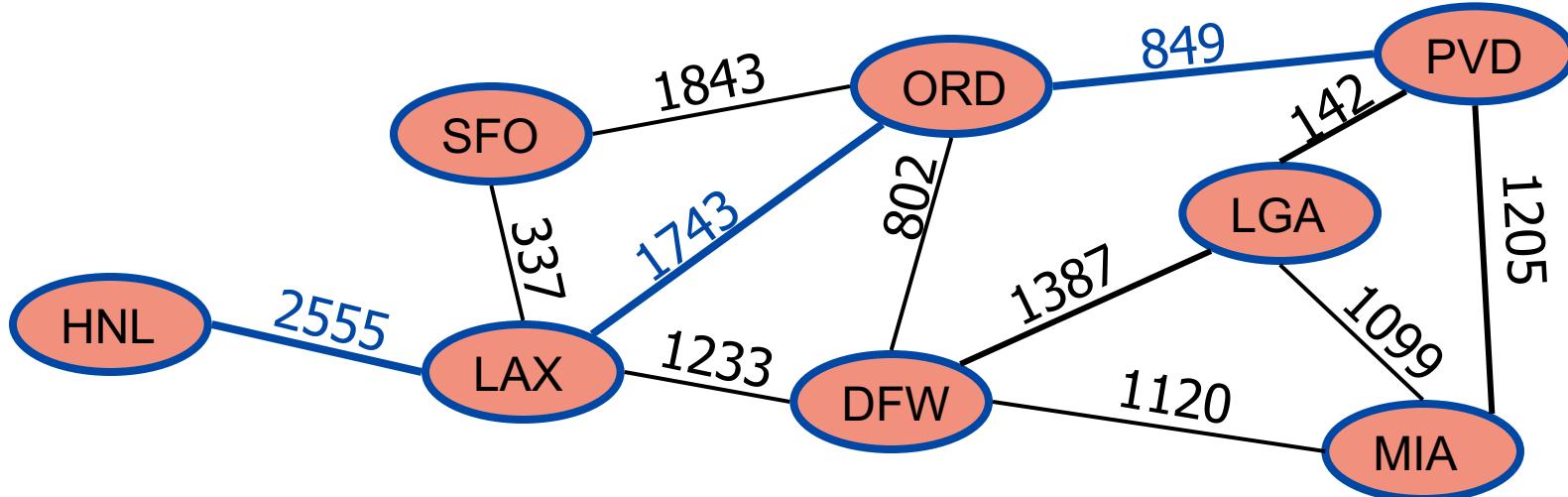
Example: Shortest path between Providence (PVD) and Honolulu (HNL)



Shortest Path Properties

Property: A subpath of a shortest path is itself a shortest path

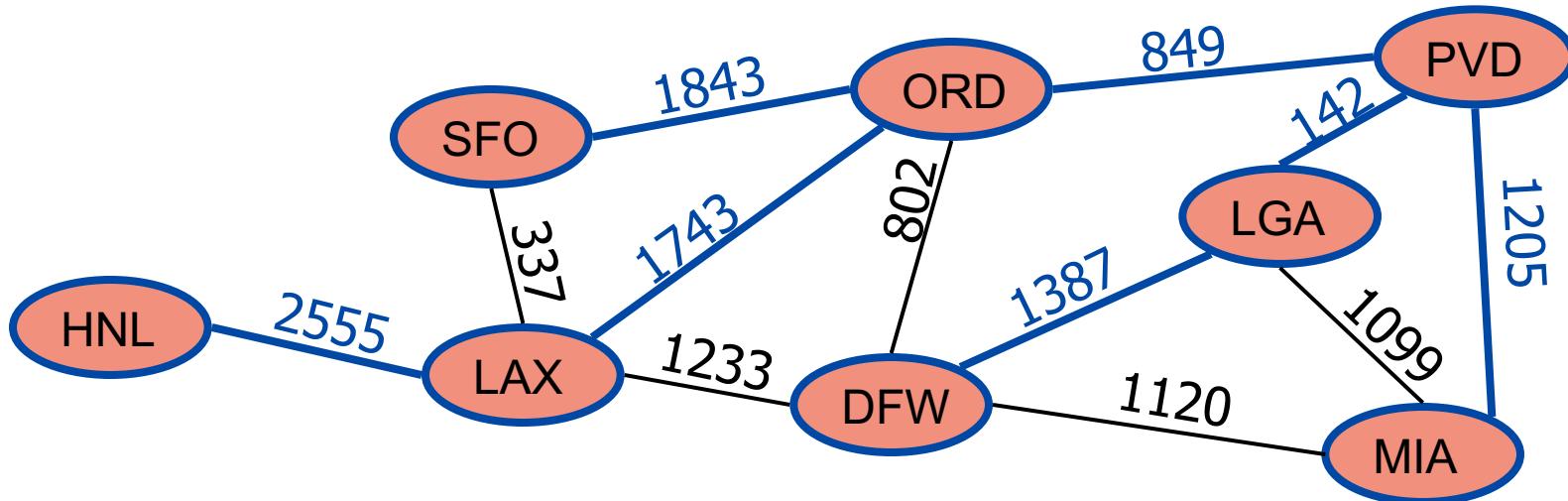
Example: Shortest path from Providence (PVD) to Honolulu (HNL)
also contains a shortest path from Providence (PVD) to
Los Angeles (LAX)



Shortest Path Properties

Property: There is a tree of shortest paths from a start vertex to all the other vertices (shortest path tree).

Example: Tree of shortest paths from Providence (PVD)



Dijkstra's Algorithm

Input:

- Graph $G = (V, E)$
- Edges weights $w : E \rightarrow \mathbb{R}_+$
- Start vertex s

Output:

- Distance from s to all v in V
- Shortest path tree rooted at s

Assumptions:

- G is connected and undirected
- edge weights are nonnegative

High level idea:

- Maintain a distance estimate
 $D[v] \geq \text{dist}_w(s, v)$ for all v in V
- Keep track of a subset S of V s.t.
 $D[v] = \text{dist}_w(s, v)$ for all v in S

Initially:

- $D[s] = 0$
- $D[v] = \infty$ for all v in $V - S$

In each iteration we:

- add to S vertex u in $V \setminus S$ with smallest $D[u]$
- update D -values for vertices adjacent to u

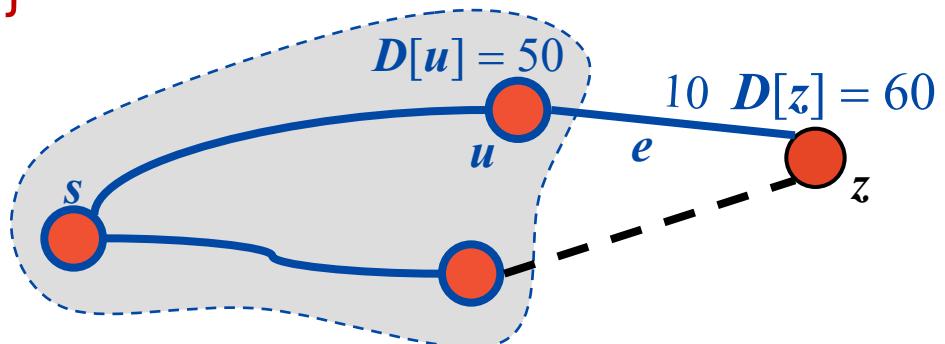
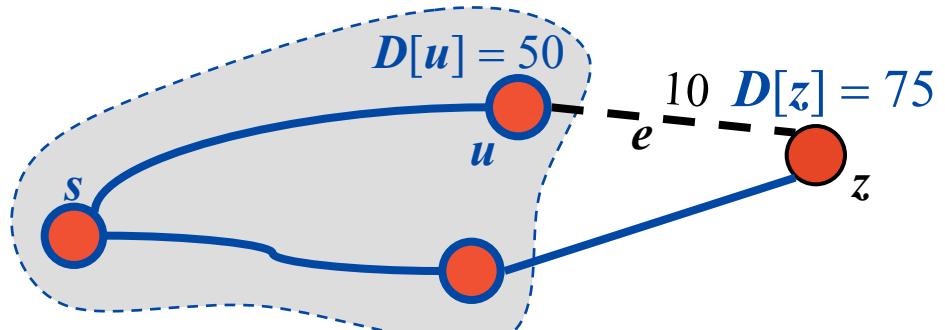
Edge Relaxation

Consider edge $e = (u, z)$ such that:

- u is the last vertex added to S
- z is not in S

The relaxation of edge (u, z) updates $D[z]$ as follows:

$$D[z] \leftarrow \min\{D[z], D[u] + w(u, z)\}$$



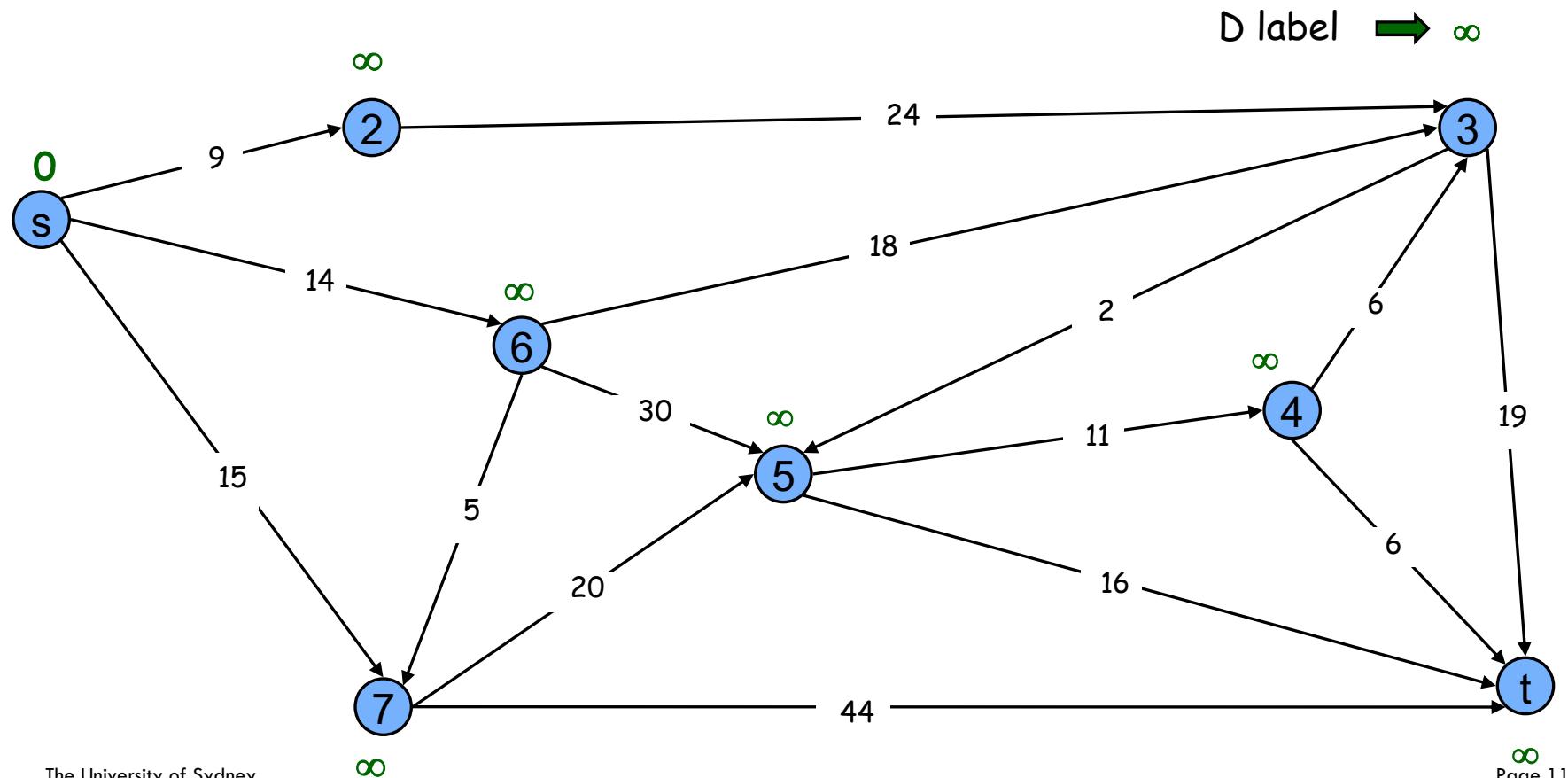
Dijkstra's Algorithm pseudocode

```
def Dijkstra(G, w, s):  
  
    # initialize algorithm  
    for v in V do  
        D[v] = infinity  
        parent[v] = nil  
    D[s] = 0  
    Q = new priority queue for { (v, D[v]) : v in V }  
  
    # iteratively add vertices to S  
    while Q is not empty do  
        u = Q.remove_min()  
        for z in G.neighbors(u) do  
            if D[u] + w[u, z] < D[z] then  
                D[z] = D[u] + w[u, z]  
                Q.update_priority(z, D[z])  
                parent[z] = u  
    return D, parent
```

Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$

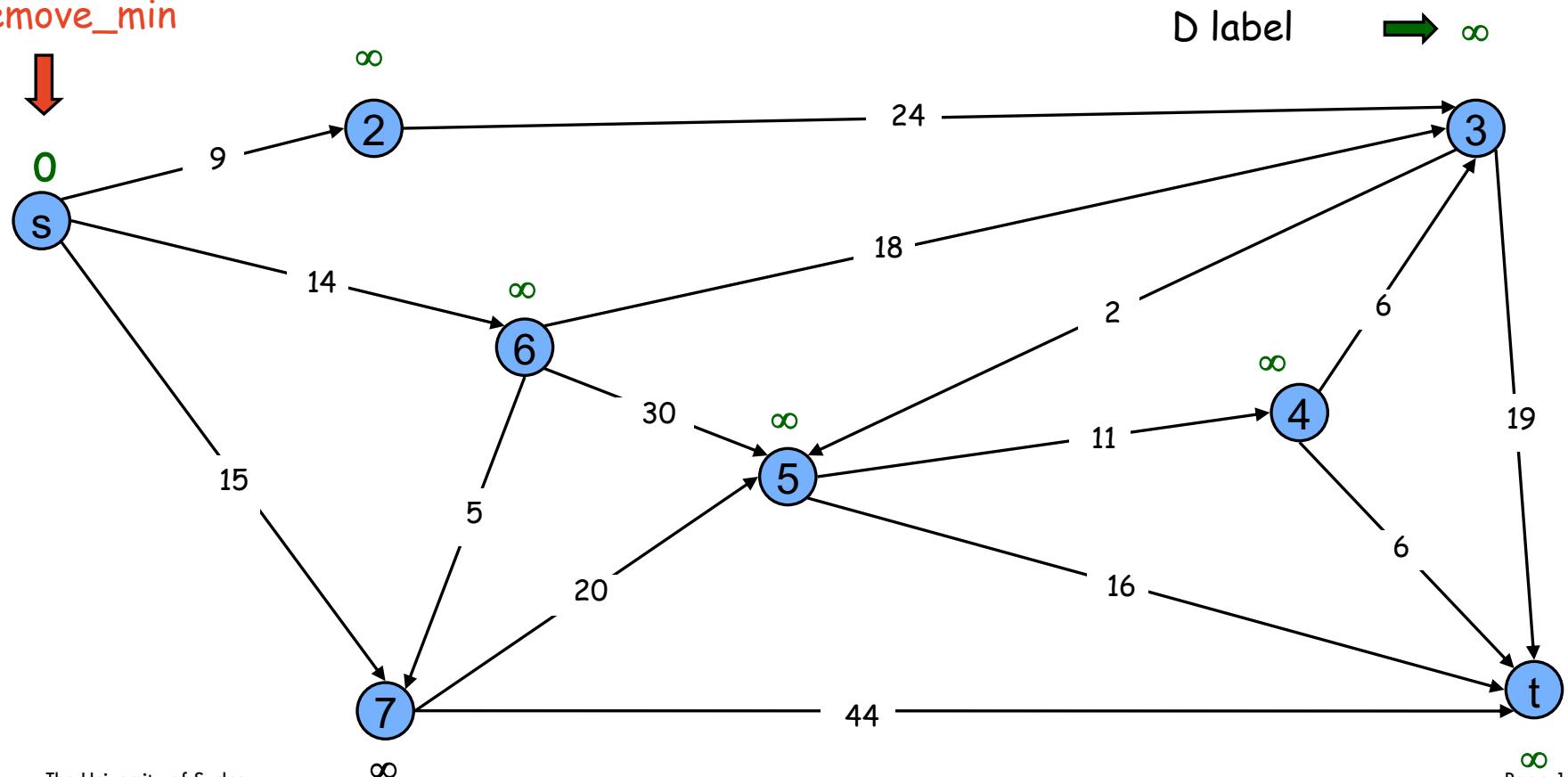


Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, t \}$

remove_min

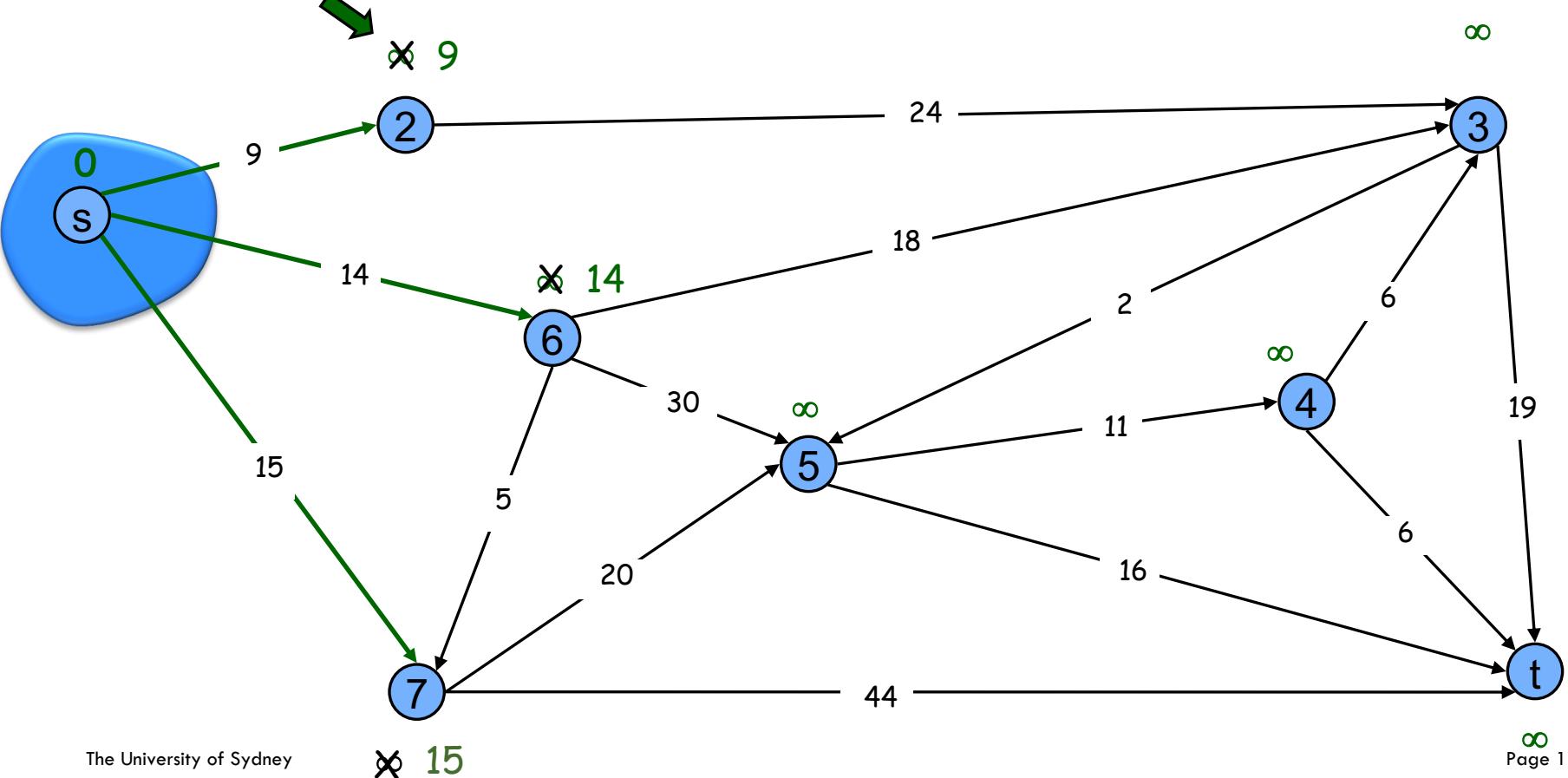


Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$

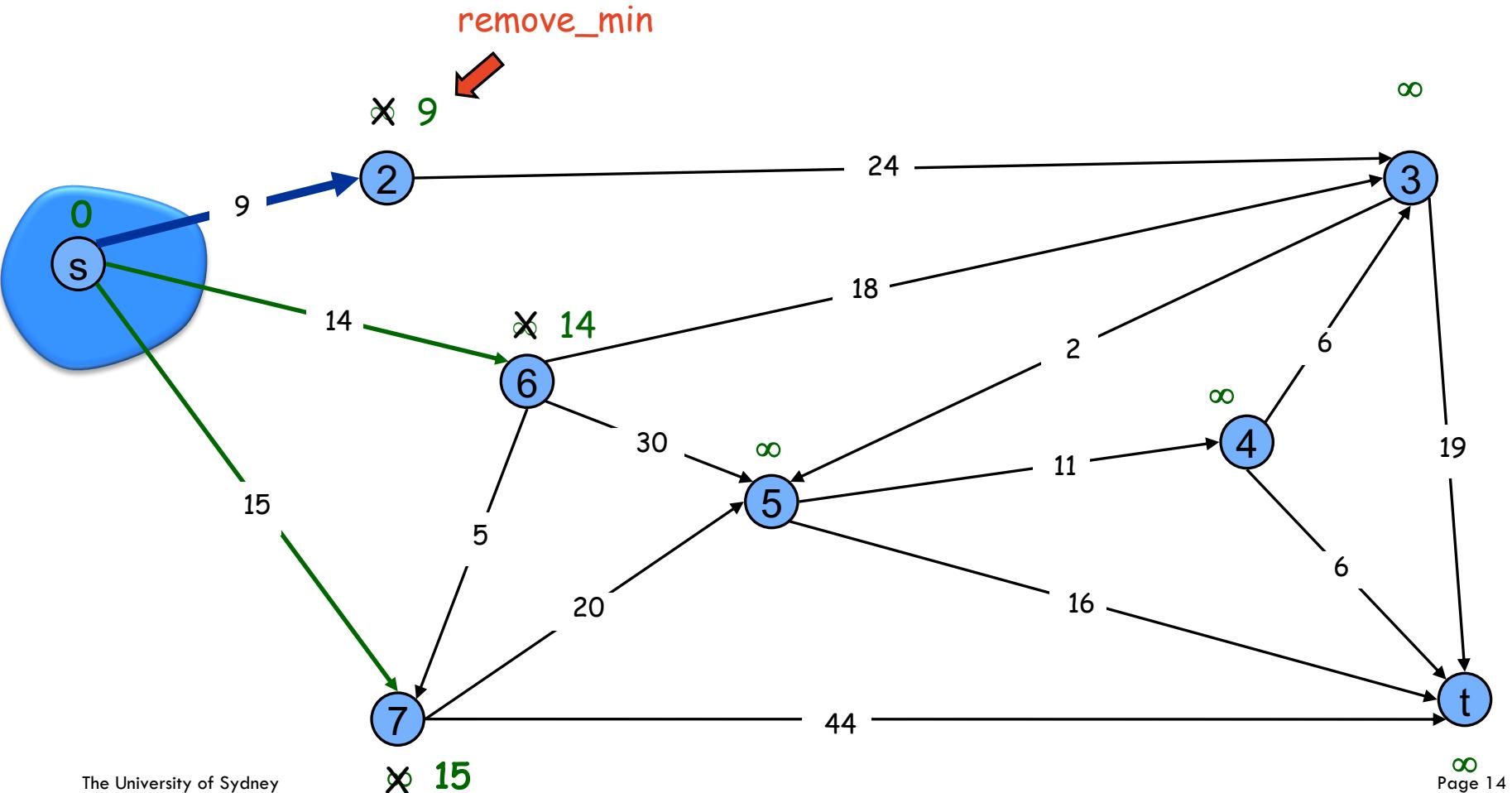
decrease key



Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

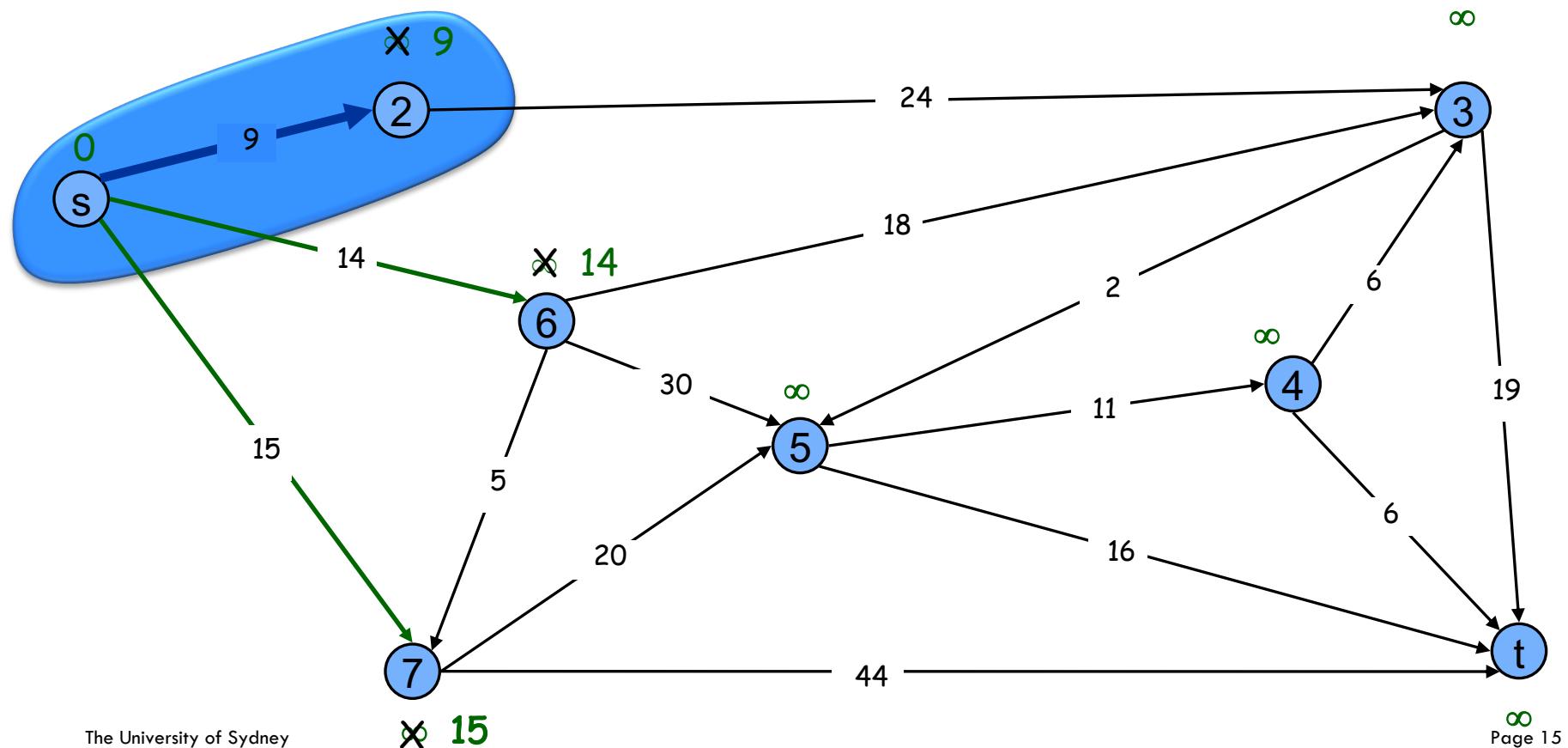
$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2 \}$

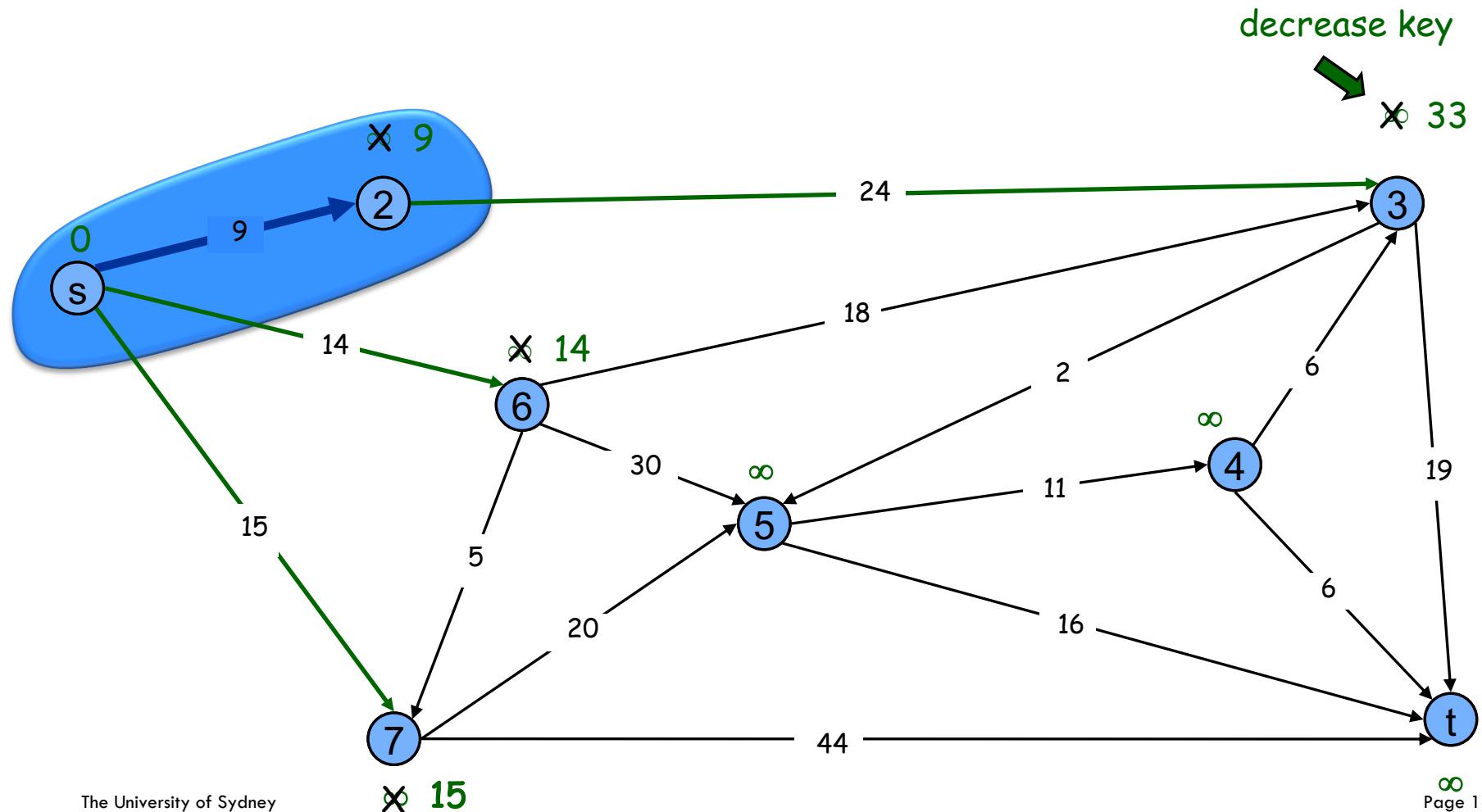
$PQ = \{ 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2 \}$$

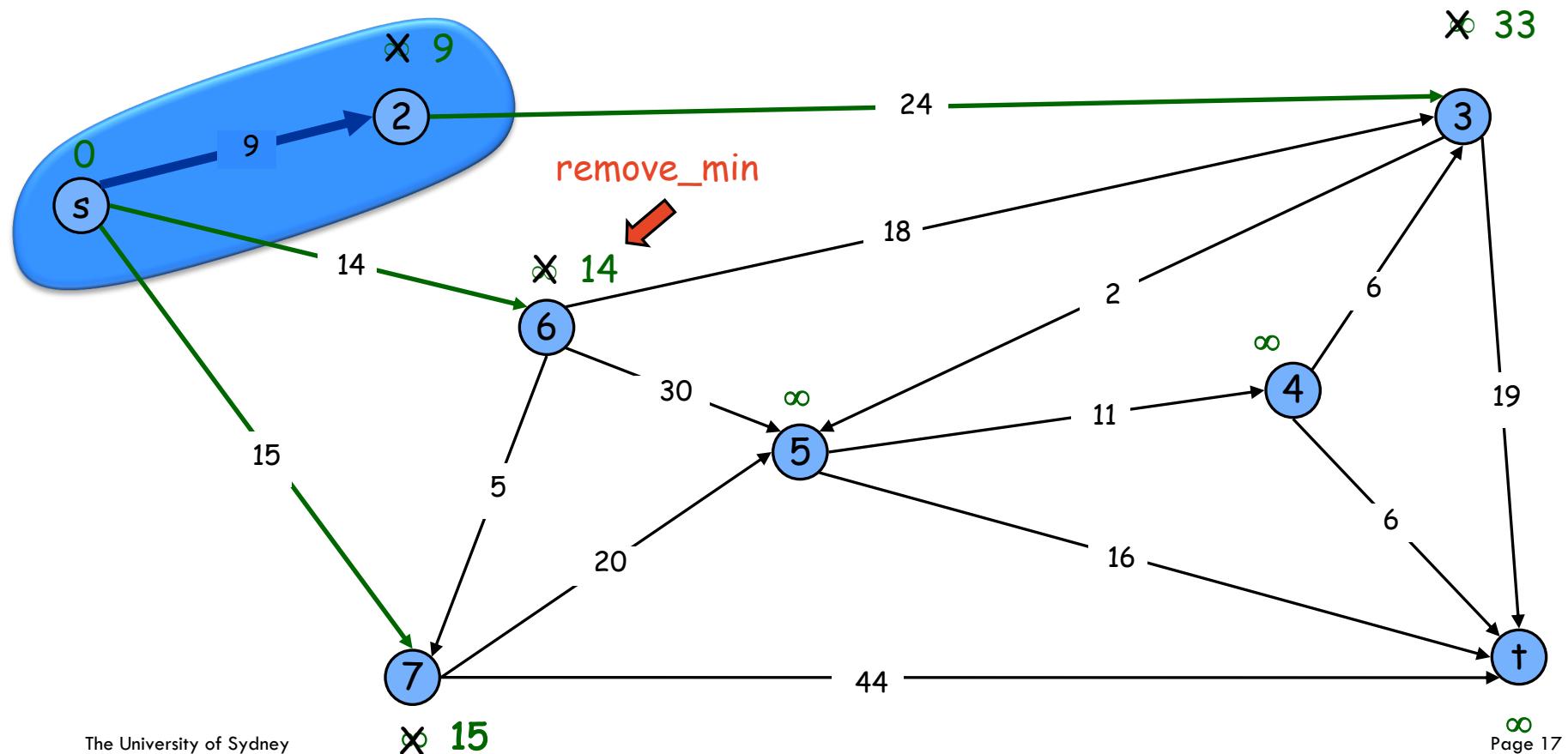
$$PQ = \{ 3, 4, 5, 6, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2 \}$$

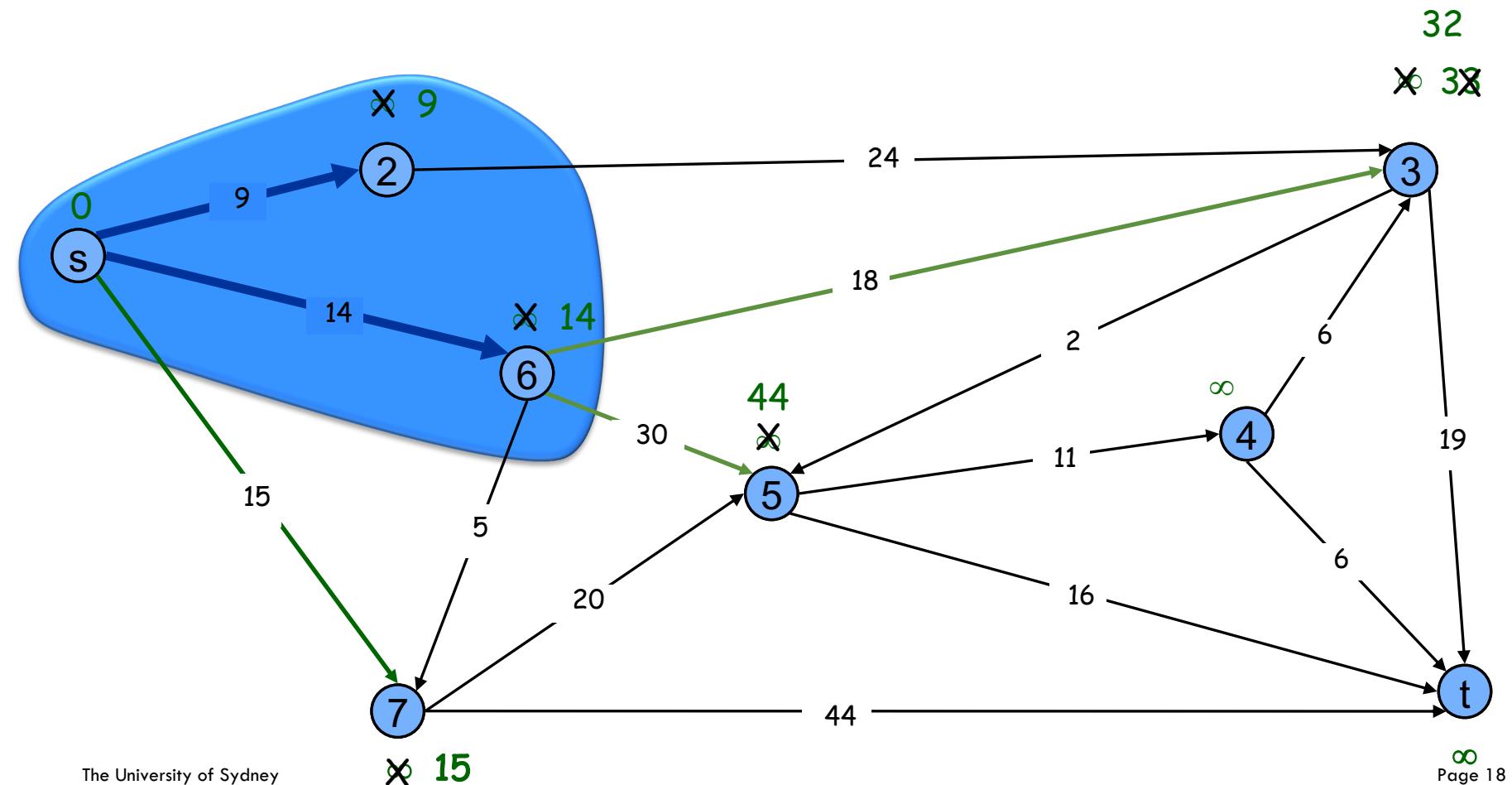
$$PQ = \{ 3, 4, 5, 6, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2, 6 \}$$

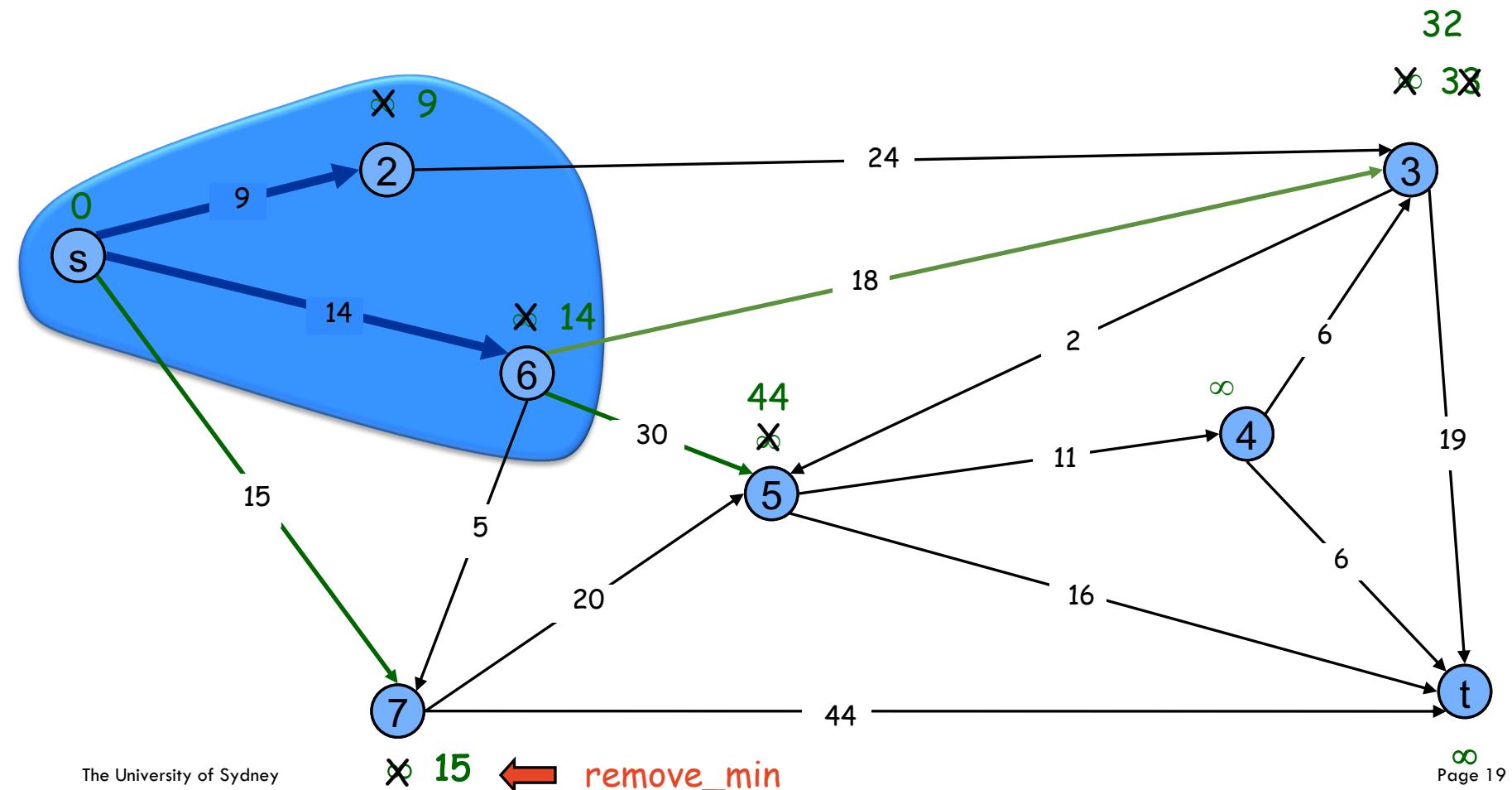
$$PQ = \{ 3, 4, 5, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2, 6 \}$$

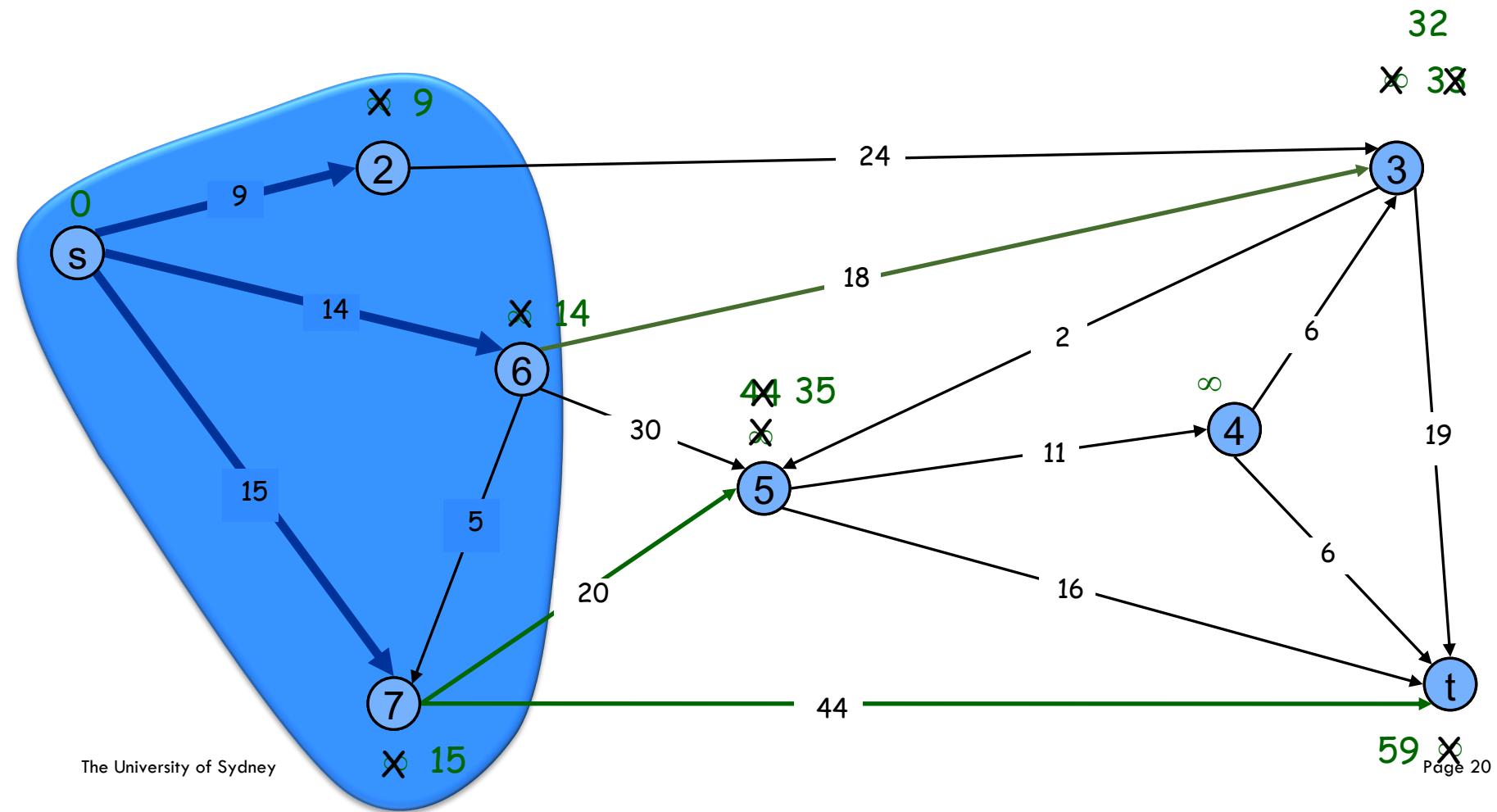
$$PQ = \{ 3, 4, 5, 7, t \}$$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 6, 7 \}$

$PQ = \{ 3, 4, 5, t \}$



Dijkstra's Shortest Path Algorithm

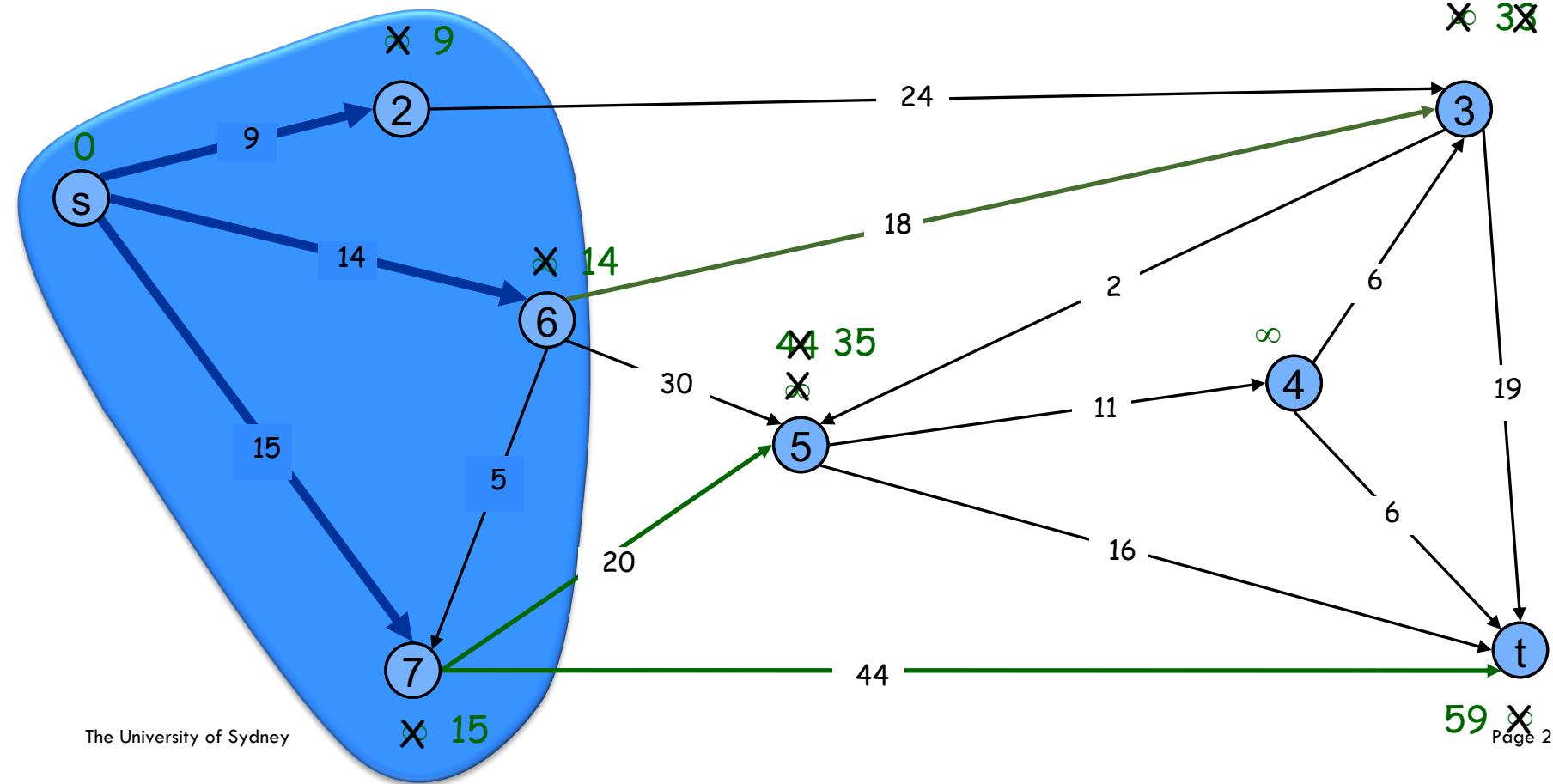
$S = \{ s, 2, 6, 7 \}$

$PQ = \{ 3, 4, 5, t \}$

remove_min

32

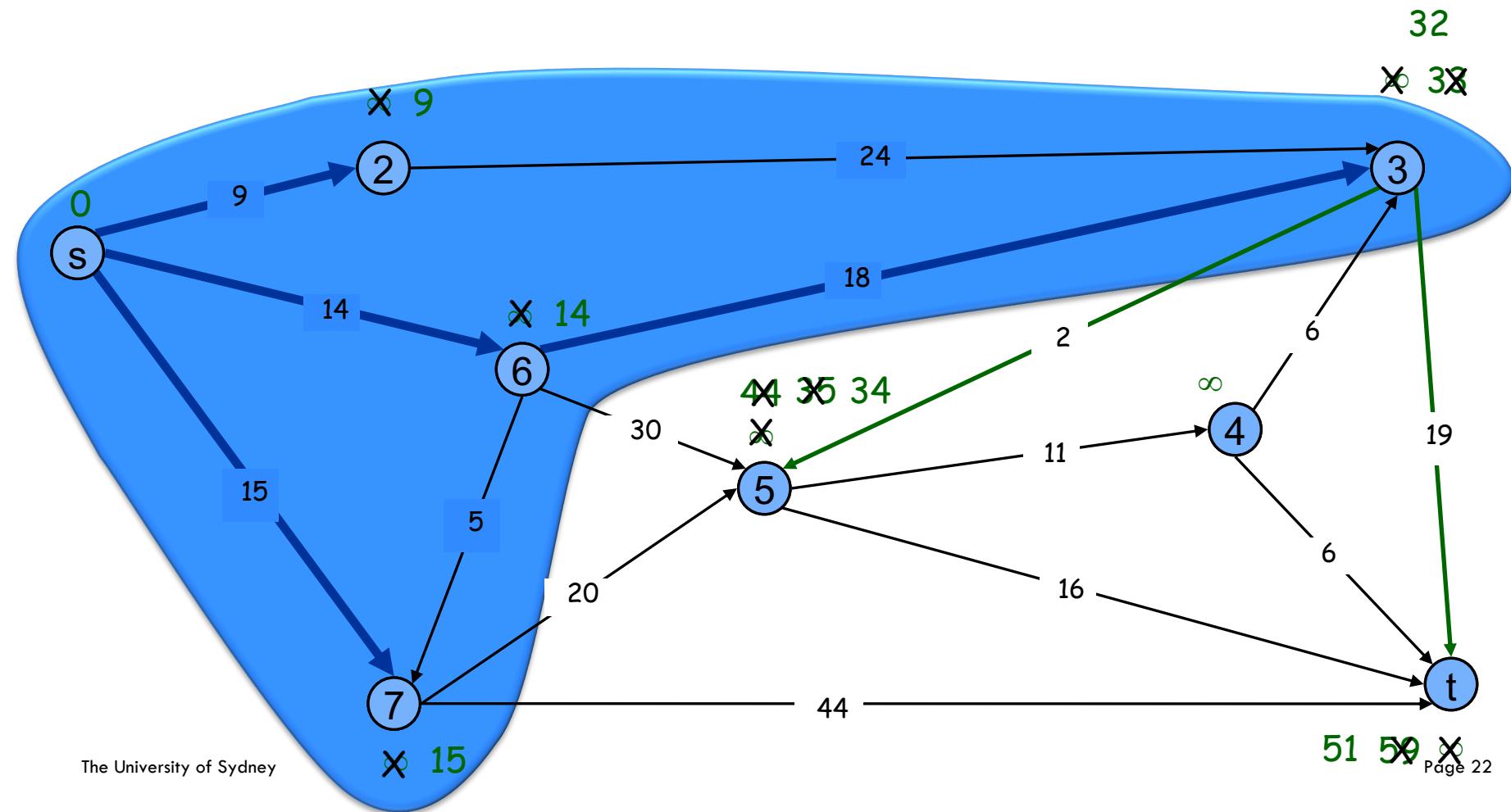
38



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 6, 7 \}$

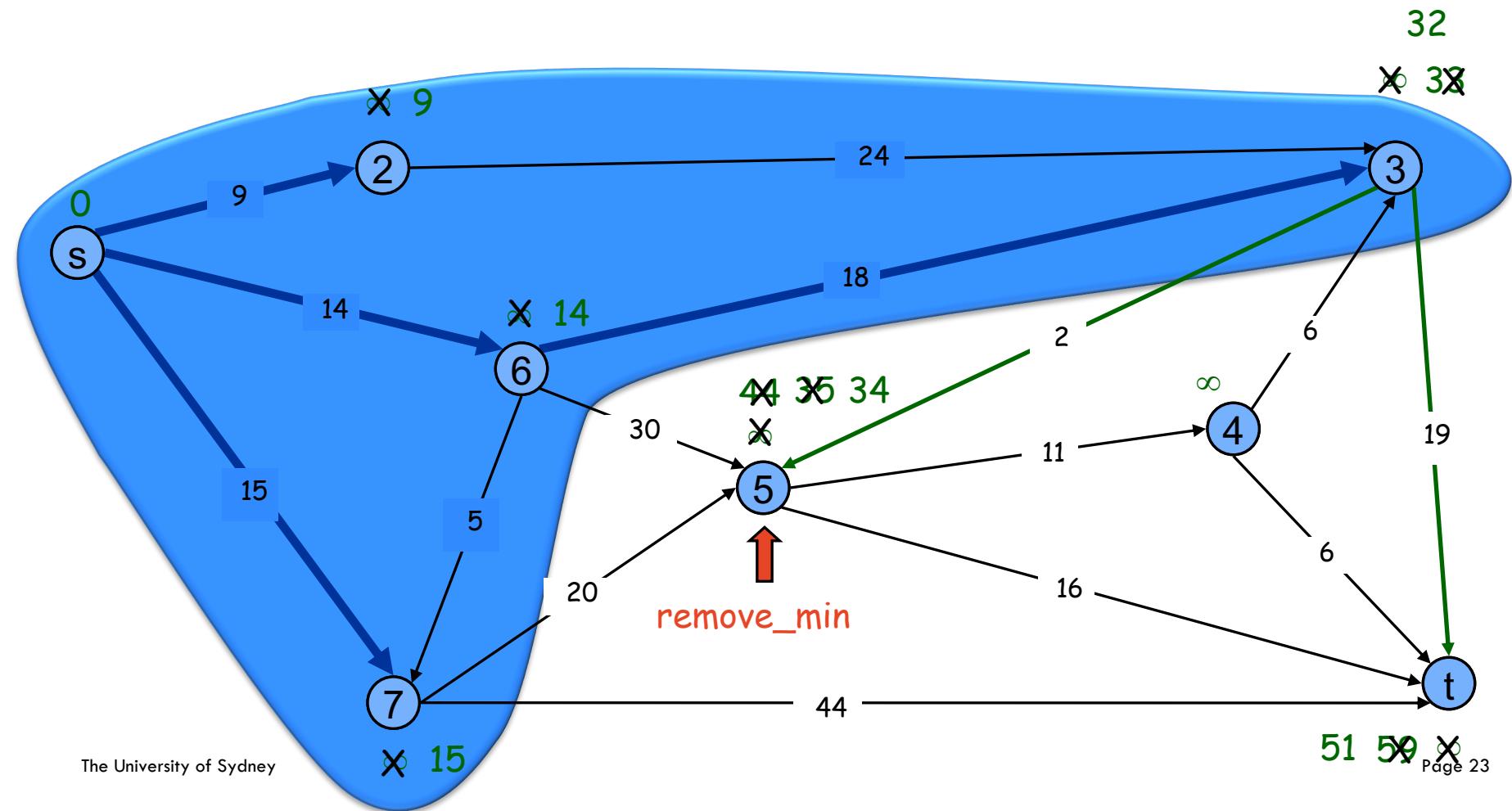
$PQ = \{ 4, 5, t \}$



Dijkstra's Shortest Path Algorithm

$$S = \{ s, 2, 3, 6, 7 \}$$

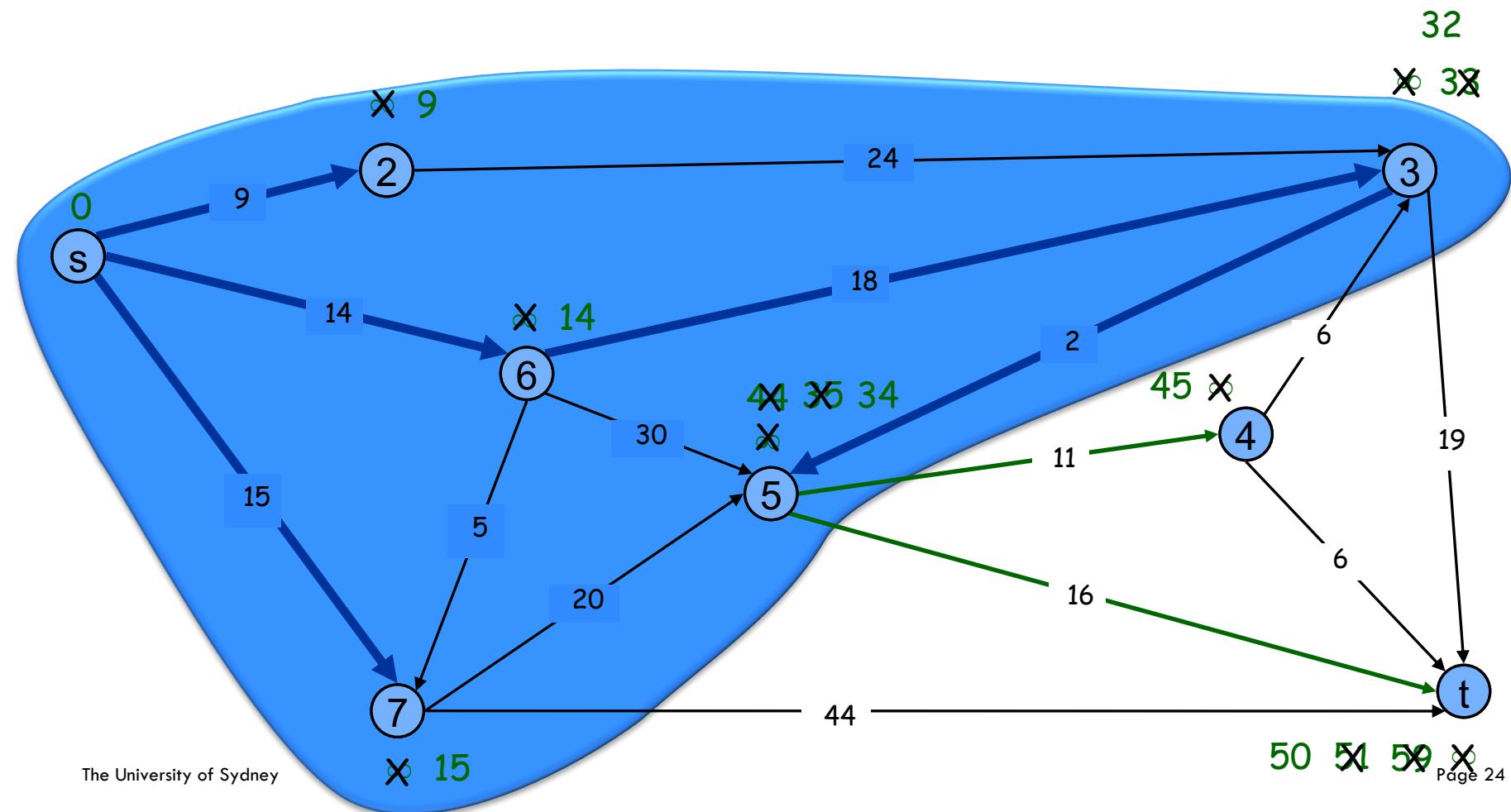
$$PQ = \{ 4, 5, t \}$$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 5, 6, 7 \}$

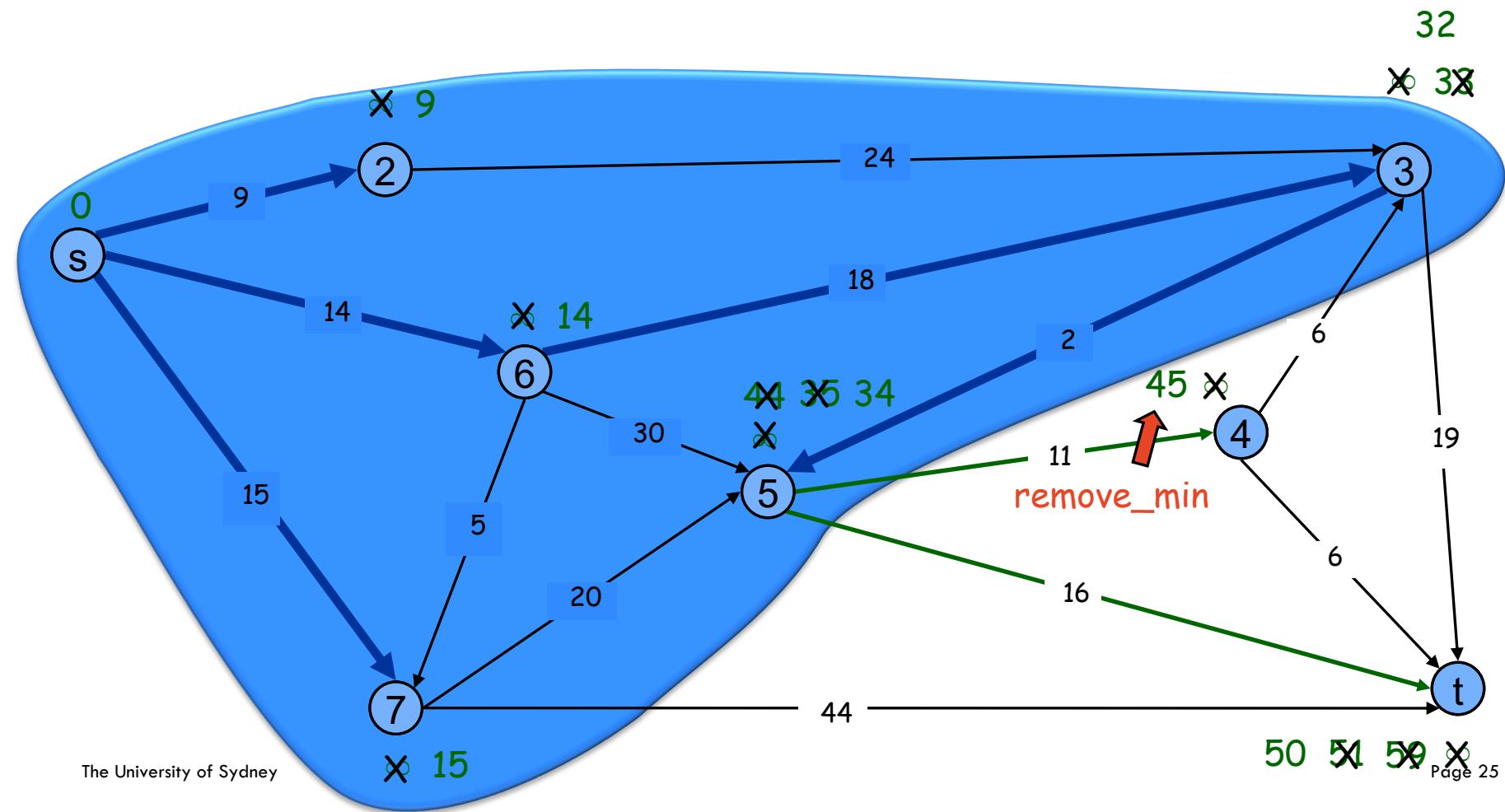
$PQ = \{ 4, t \}$



Dijkstra's Shortest Path Algorithm

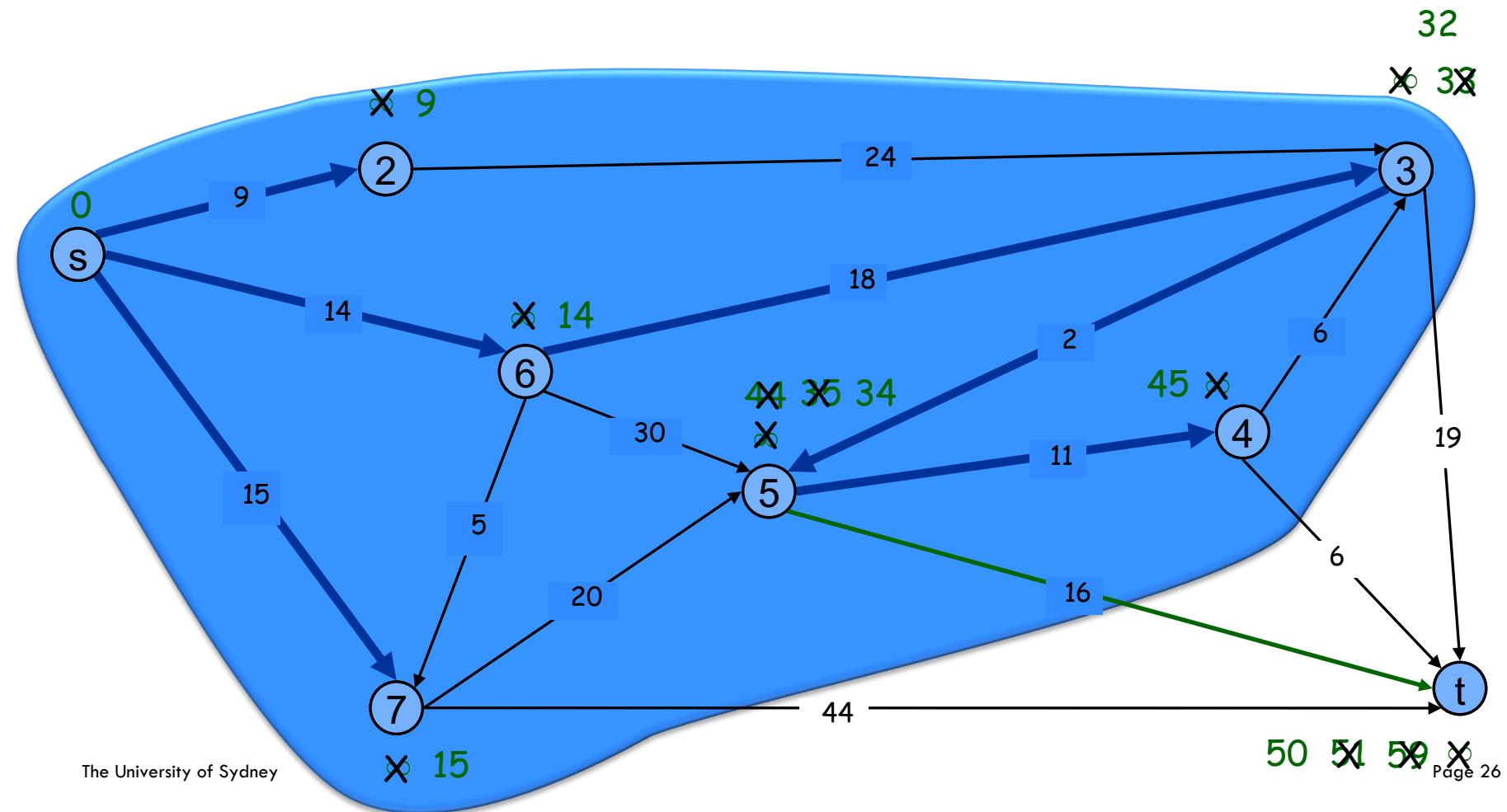
$$S = \{ s, 2, 3, 5, 6, 7 \}$$

$$PQ = \{ 4, t \}$$



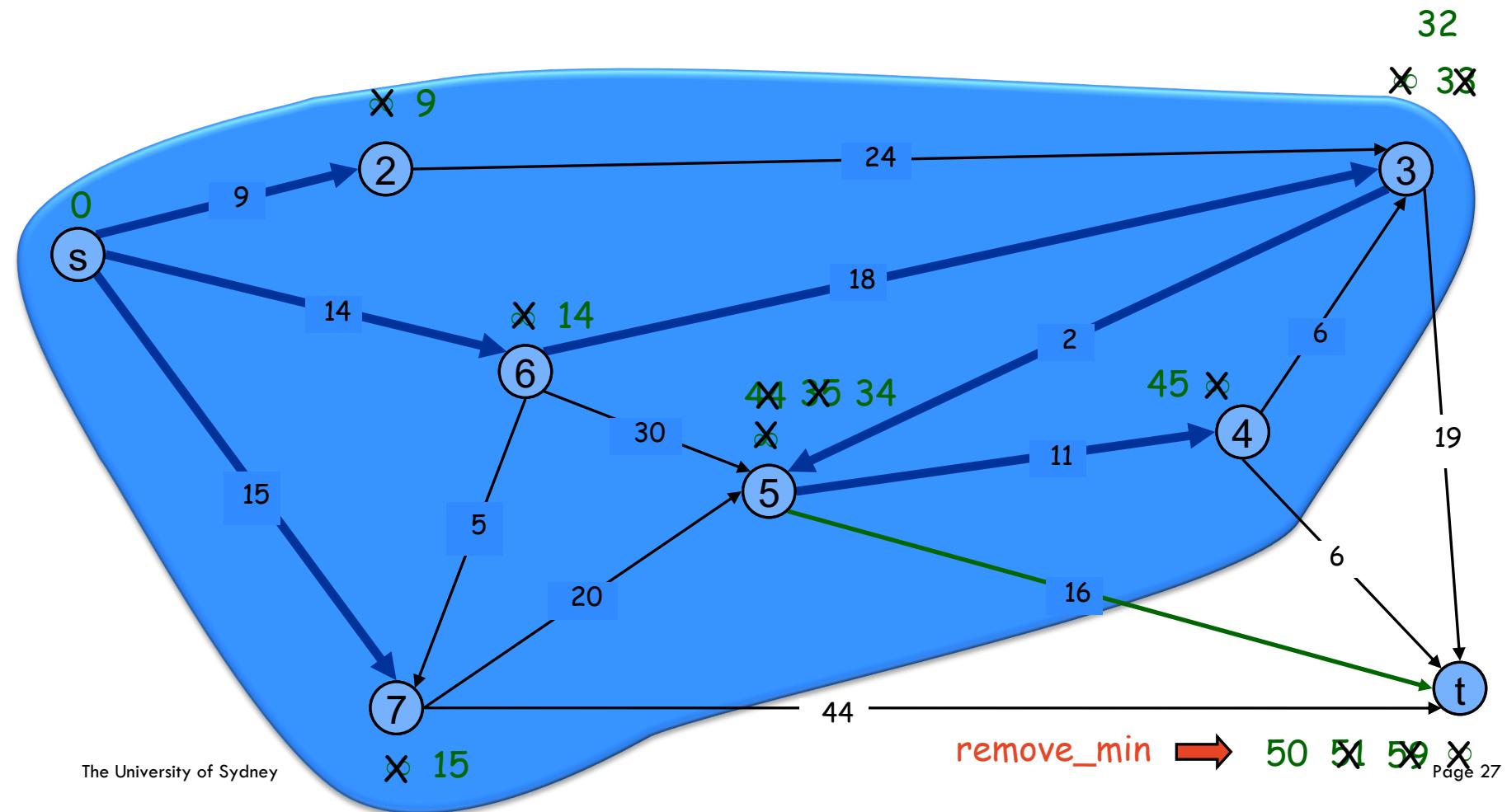
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7 \}$
 $PQ = \{ t \}$



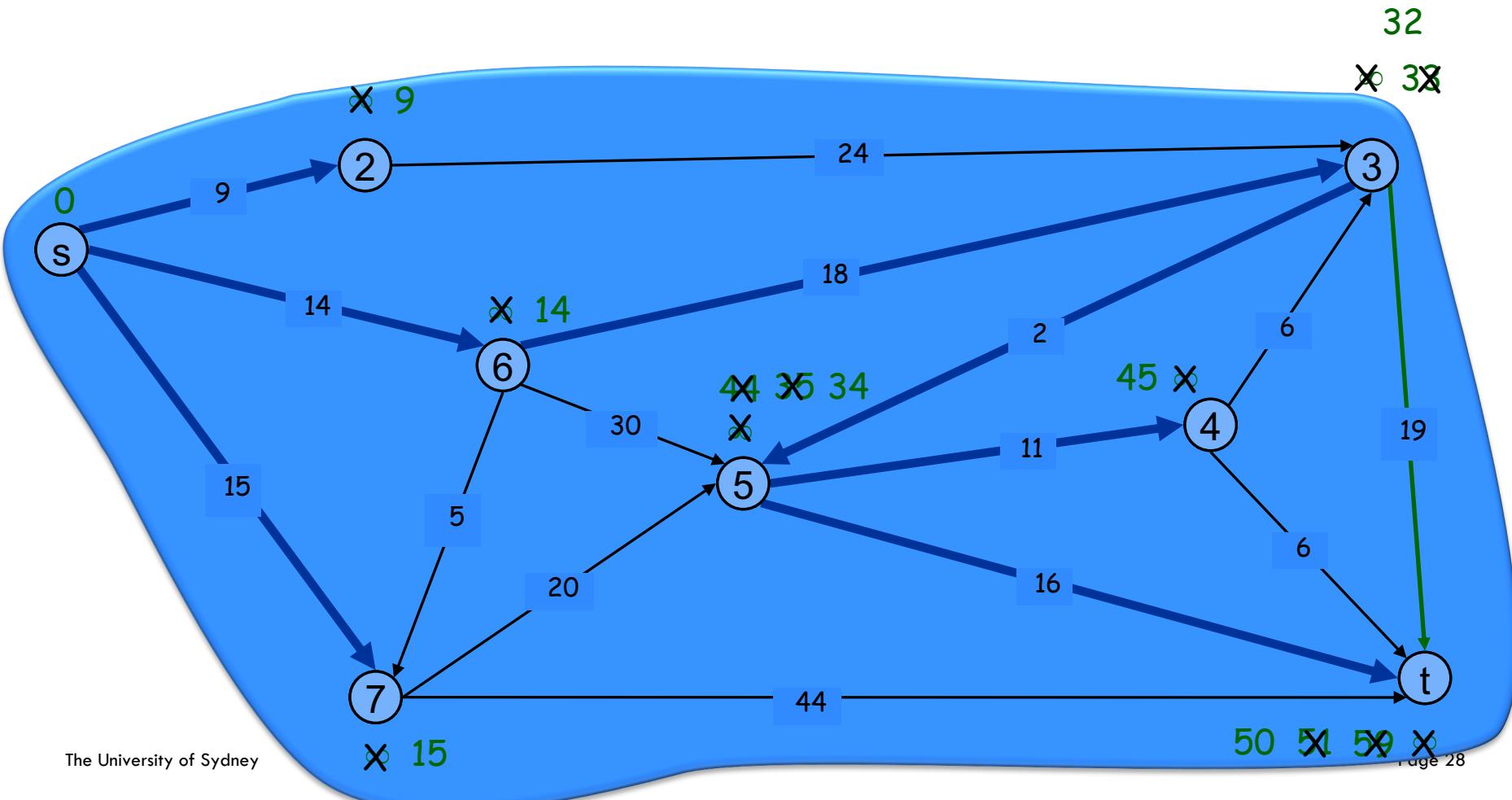
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7 \}$
 $PQ = \{ t \}$



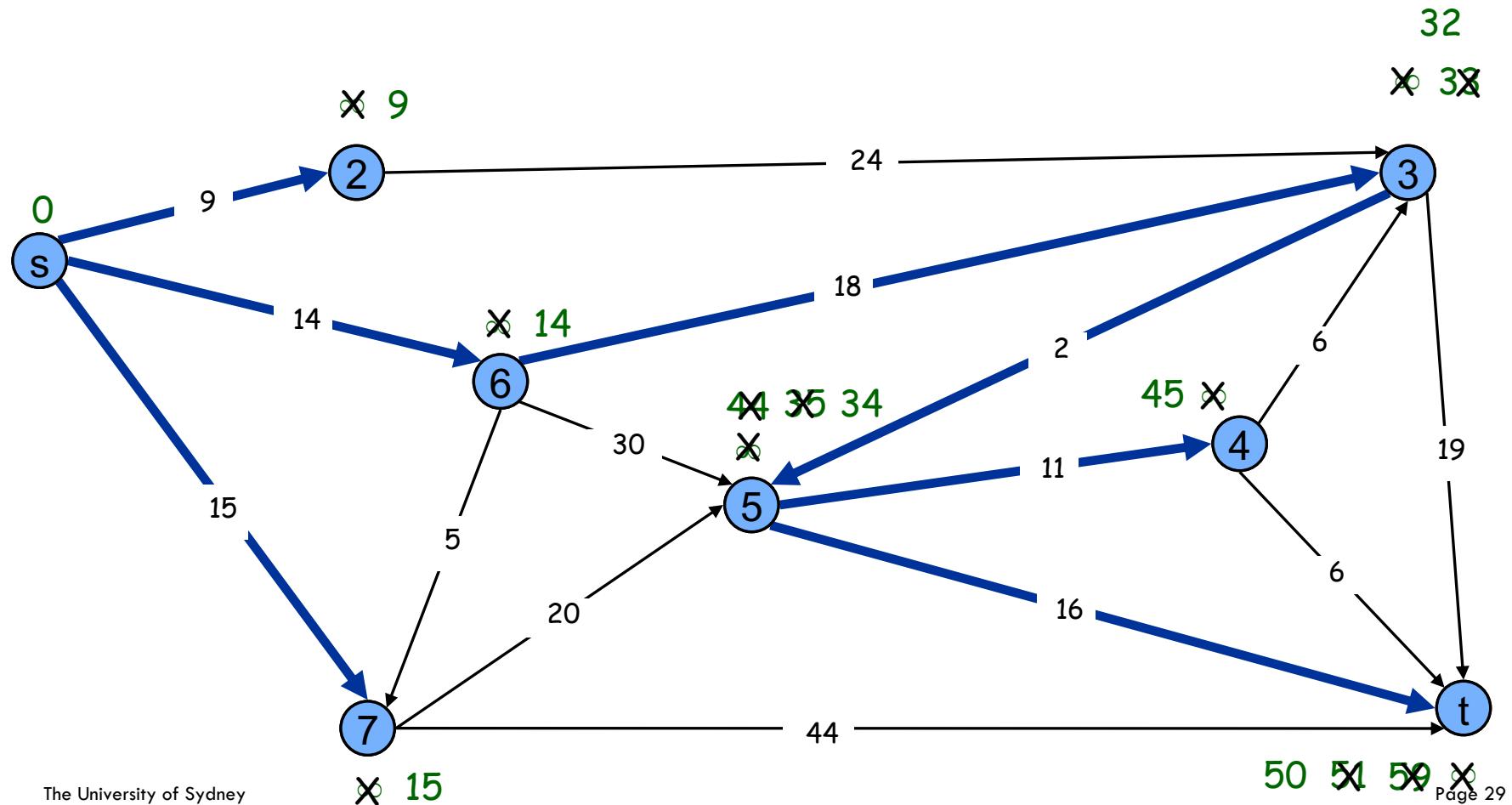
Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7, t \}$
 $PQ = \{ \}$



Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7, t \}$
 $PQ = \{ \}$



Dijkstra complexity analysis except PQ ops

```
def Dijkstra(G, w, s):
```

```
# initialize algorithm
```

```
for v in V do
    D[v] = infinity
    parent[v] = nil
D[s] = 0
```

O(n)

```
Q = new priority queue for { (v, D[v]) : v in V }
```

```
# iteratively add vertices to S
```

```
while Q is not empty do
```

```
    u = Q.remove_min()
```

```
    for z in G.neighbors(u) do
```

```
        if D[u] + w[u, z] < D[z] then
```

```
            D[z] = D[u] + w[u,z]
```

```
            Q.update_priority(z, D[z])
```

```
            parent[z] = u
```

```
return D, parent
```

O(deg(u)) for each $u \in V$
plus update_priority work

Dijkstra's Algorithm complexity analysis

Assuming the graph is connected (so $m \geq n-1$), the algorithm spends $\mathcal{O}(m)$ time on everything except PQ operations

Priority queue operation counts:

- insert: n
- decrease_key: m
- remove_min: n

Fact: Using a heap for PQ, Dijkstra runs in $\mathcal{O}(m \log n)$ time

Fibonacci heaps is a PQ that can carry out decrease key in $\mathcal{O}(1)$ amortized time. Using that instead we get $\mathcal{O}(m + n \log n)$ time.

Dijkstra's Algorithm Correctness

Invariant: For each $u \in S = V \setminus Q$, we have $D[u] = \text{dist}_w(s, u)$

Proof: (by induction on $|S|$)

Base case: $|S| = 1$ is trivial since $D[s] = 0$

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

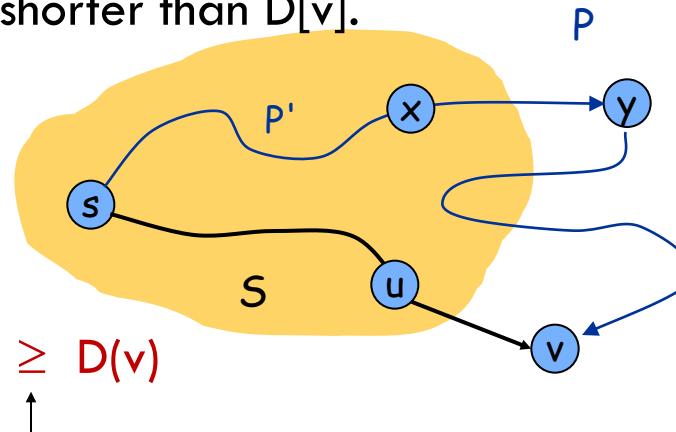
- Let v be next node added to S and $u = \text{parent}[v]$
- The shortest $s-u$ path plus (u, v) is an $s-v$ path of length $D[v]$
- Consider any $s-v$ path P . We'll see that it's no shorter than $D[v]$.
- Let $x-y$ be the first edge in P that leaves S , and let P' be the subpath to x .
- P is already too long as soon as it leaves S :

$$w(P) \geq w(P') + w(x, y) \geq D(x) + w(x, y) \geq D(y) \geq D(v)$$

inductive
hypothesis

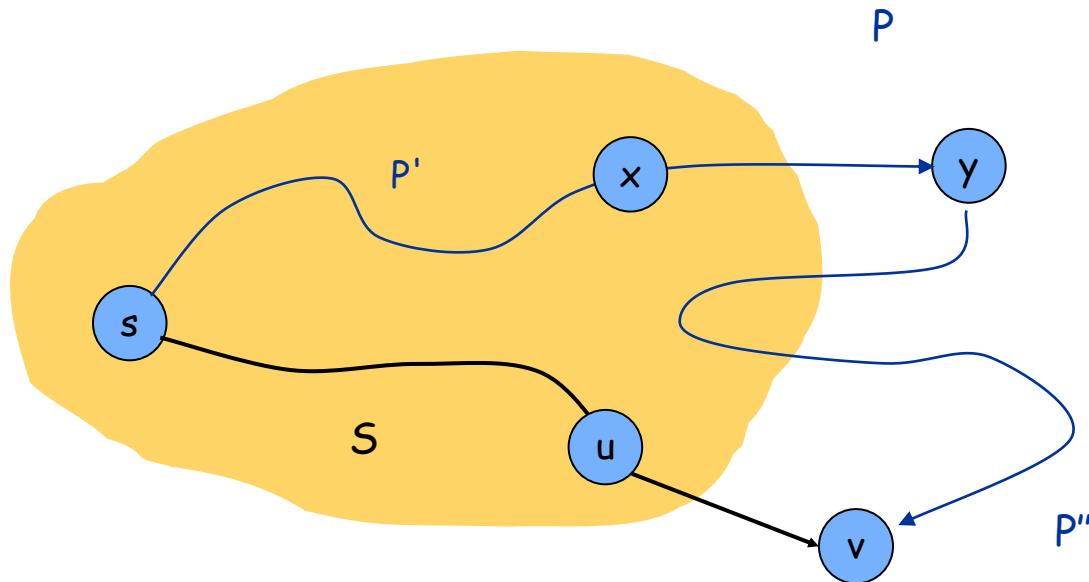
Def of
 $D(y)$

Dijkstra chose v
instead of y



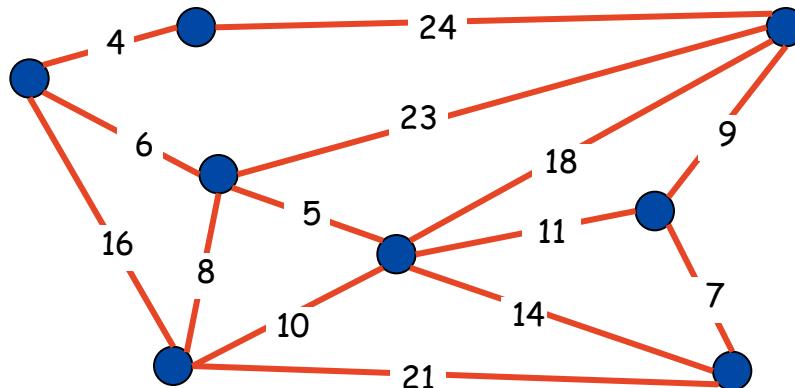
Warning: Dijkstra may not work for negative-weight edges

In the proof of correctness, even if $D[v]$ is the smallest label, it may be that $\text{dist}_w(s, v) < D[v]$ if $w(P'') < 0$

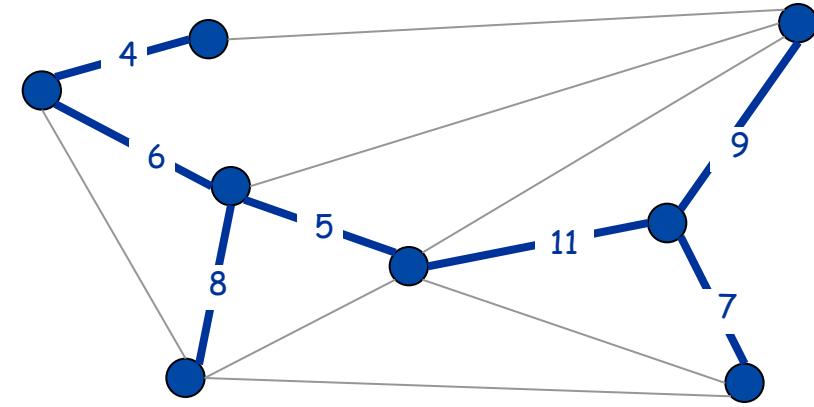


Minimum Spanning Tree

Given a connected graph $G = (V, E)$ with real-valued edge weights c_e , an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



T with $\sum_{e \in T} c_e = 50$

Applications

MST is fundamental problem with diverse applications.

Network design: Telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems: traveling salesperson problem, Steiner tree

Indirect applications.

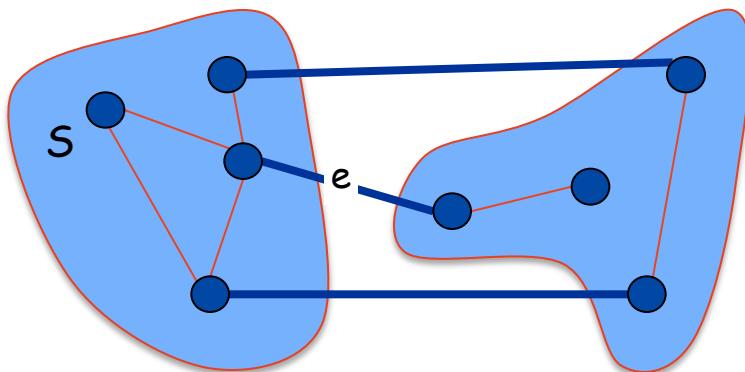
- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- ...

MST properties

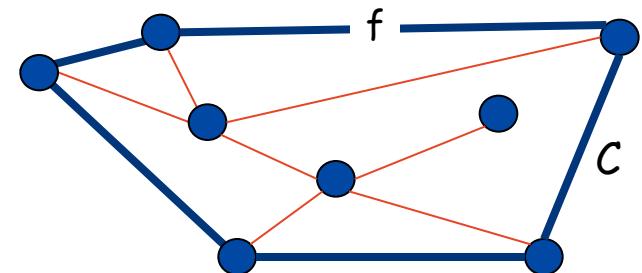
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C . Then the MST does not contain f .



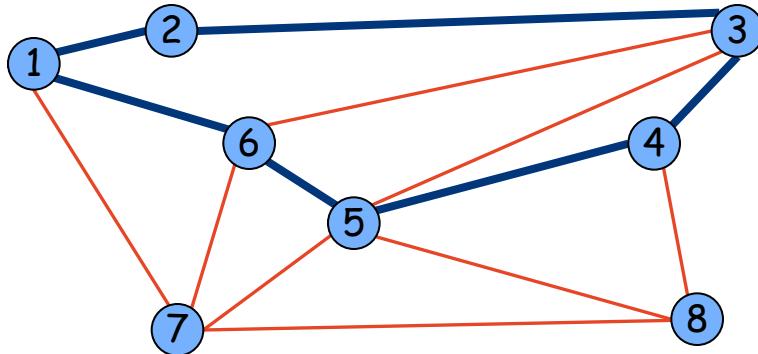
e is in the MST



f is not in the MST

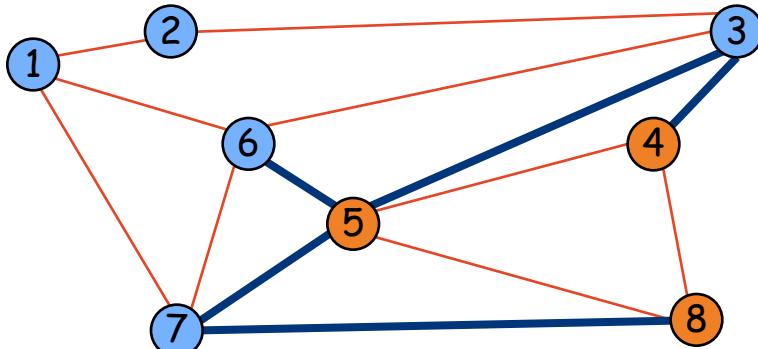
Cycles and Cuts

Cycle. Set of edges of the form $a-b, b-c, c-d, \dots, y-z, z-a$.



$$\text{Cycle } C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$$

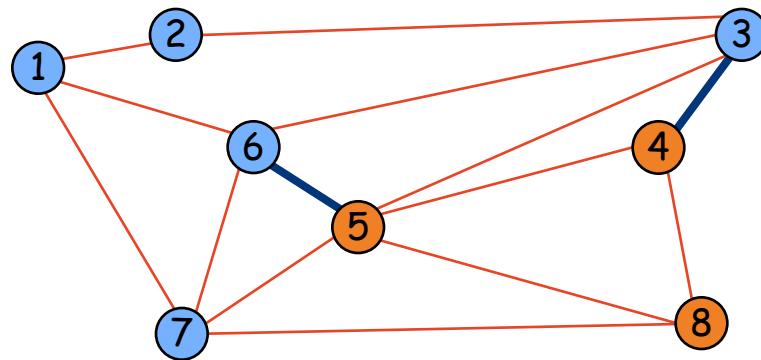
Cutset. A cut is a subset of nodes S . The corresponding cutset D is the subset of edges with exactly one endpoint in S .



$$\begin{aligned} \text{Cut } S &= \{4, 5, 8\} \\ \text{Cutset } D &= 5-6, 5-7, 3-4, 3-5, 7-8 \end{aligned}$$

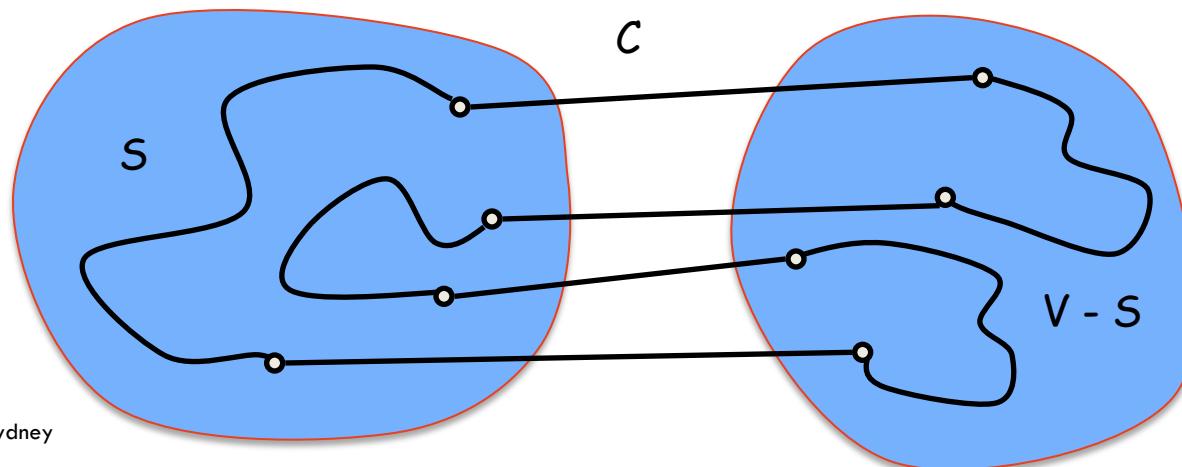
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection = 3-4, 5-6

Proof:



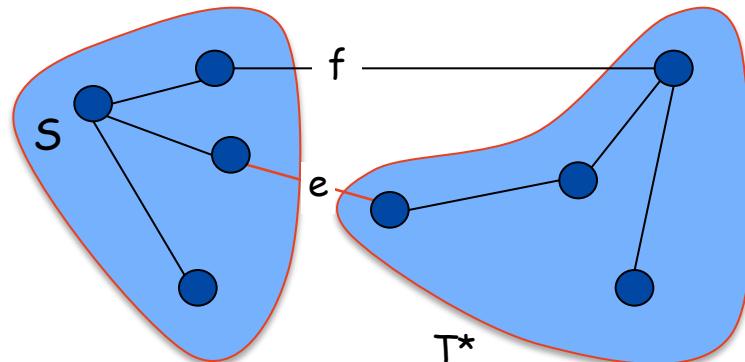
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

Proof: (exchange argument)

- Let T^* be MST and suppose e does not belong to T^*
- Adding e to T^* creates a cycle C in T^*
- Edge e is both in the cycle C and in the cutset D corresponding to $S \Rightarrow$ there exists another edge, say f , that is in both C and D .
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- A contradiction, so e must belong in T^*



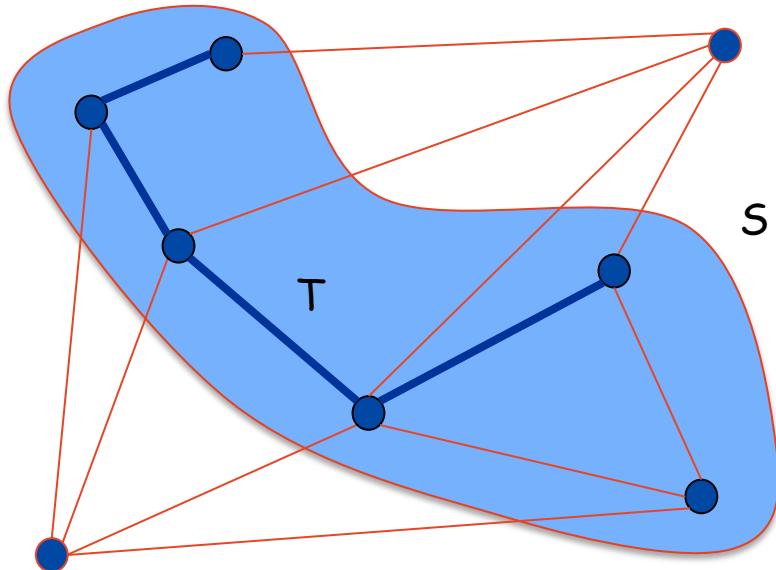
Prim's Algorithm

```
def prim(G, c):
    u = arbitrary vertex in V
    S = { u }
    T = ∅
    while |S| < |V| do
        (u, v) = min cost edge s.t. u in S and v not in S
        add (u, v) to T
        add v to S
    return T
```

Prim's Algorithm: Correctness

```
def prim(G, c):
    u = arbitrary vertex in V
    S = { u }
    T = ∅
    while |S| < |V| do
        (u, v) = min cost edge s.t. u in S and v not in S
        add (u, v) to T
        add v to S
    return T
```

Every time we add
an edge we follow
cut property!



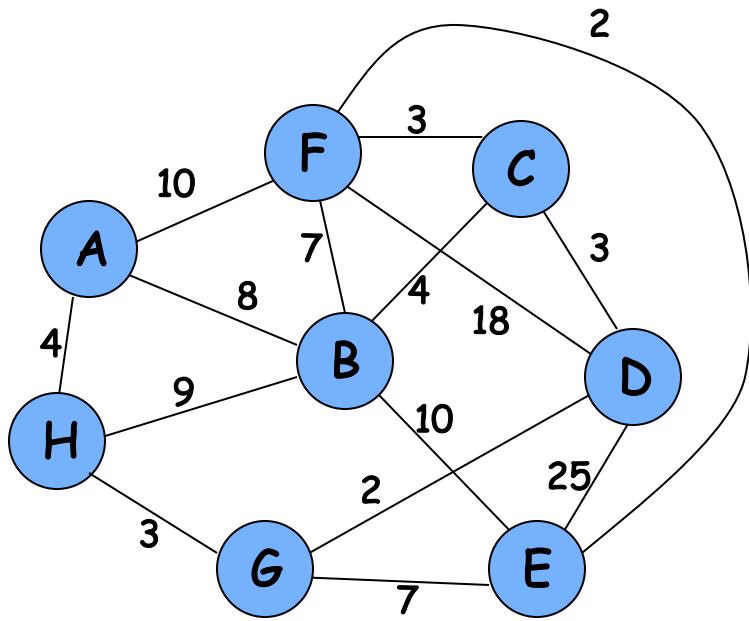
Implementation: Prim's Algorithm

```
def prim(G, c) {  
    for v in V do  
        d[v] = infinity  
        parent[v] = nil  
    u = arbitrary vertex in V  
    d[u] = 0  
    Q = new PQ with items { (v, d[v]) for v in V }  
    S =  $\emptyset$   
  
    while Q is not empty do  
        u = delete min element from Q  
        add u to S  
        for (u, v) incident to u do  
            if v  $\notin$  S and  $c_e < d[v]$  then  
                parent[v] = u  
                decrease priority d[v] to  $c_e$   
return parent
```

Main idea: for every $v \in V \setminus S$ we keep

- $d[v] = \text{distance to closest neighbor in } S$
- $\text{parent}[v] = \text{closest neighbor in } S$

Walk-Through



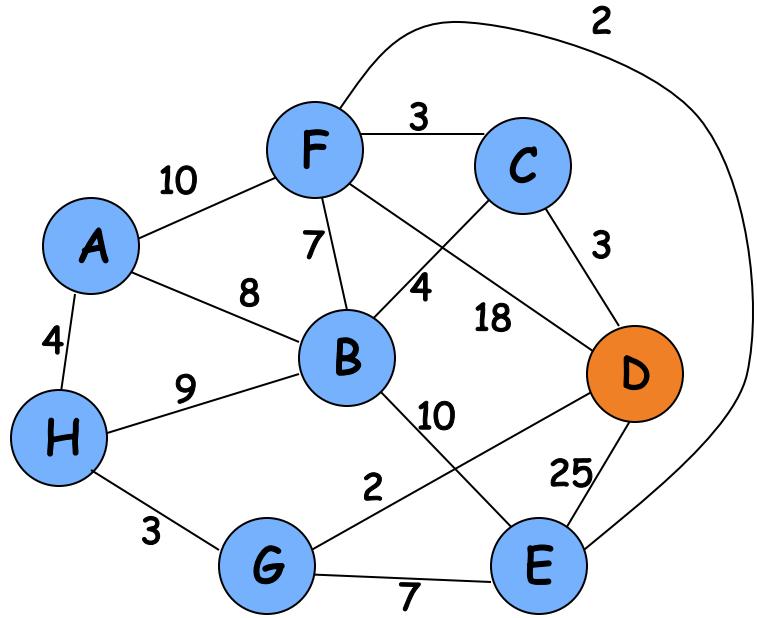
Initialize array

	S	d_v	p_v
A	F	∞	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-
F	F	∞	-
G	F	∞	-
H	F	∞	-

Set
 S

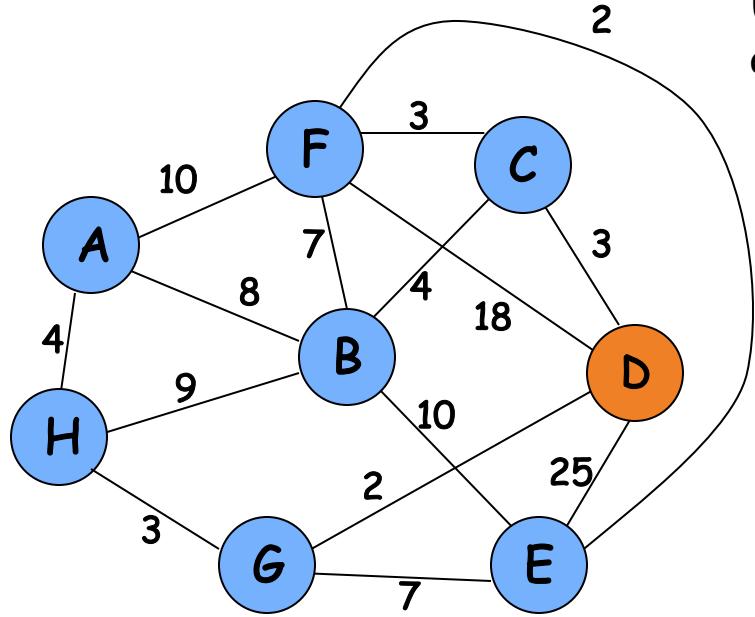
Min distance
to S

Closest
vertex in S



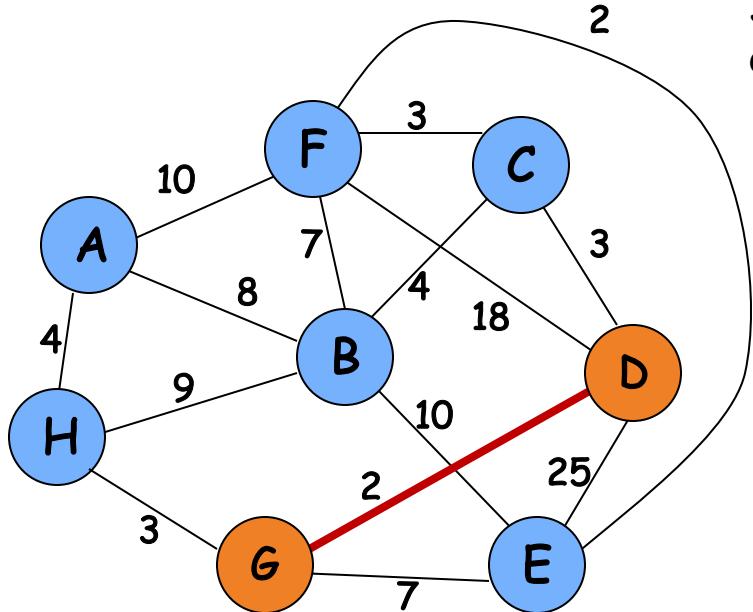
Start with any node, say D

	S	d_v	p_v
A			
B			
C			
D	T	0	-
E			
F			
G			
H			



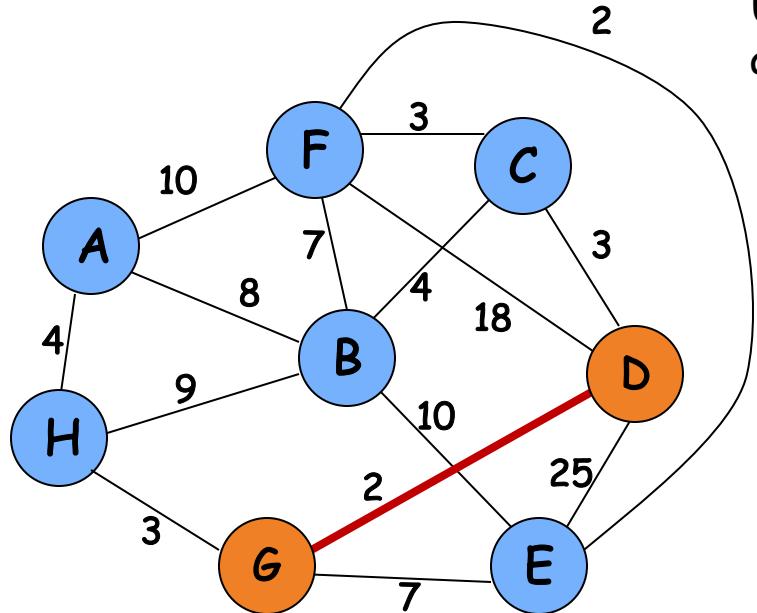
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G		2	D
H			



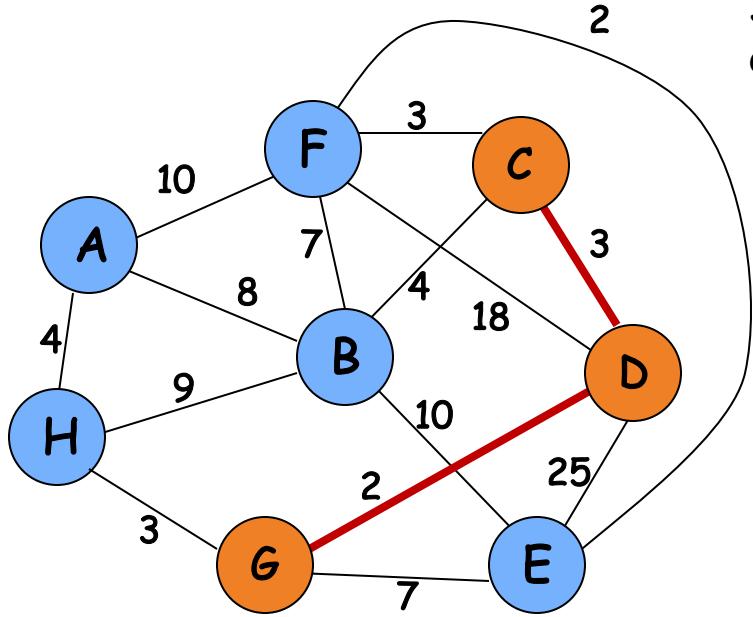
Select node with minimum distance

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G	T	2	D
H			



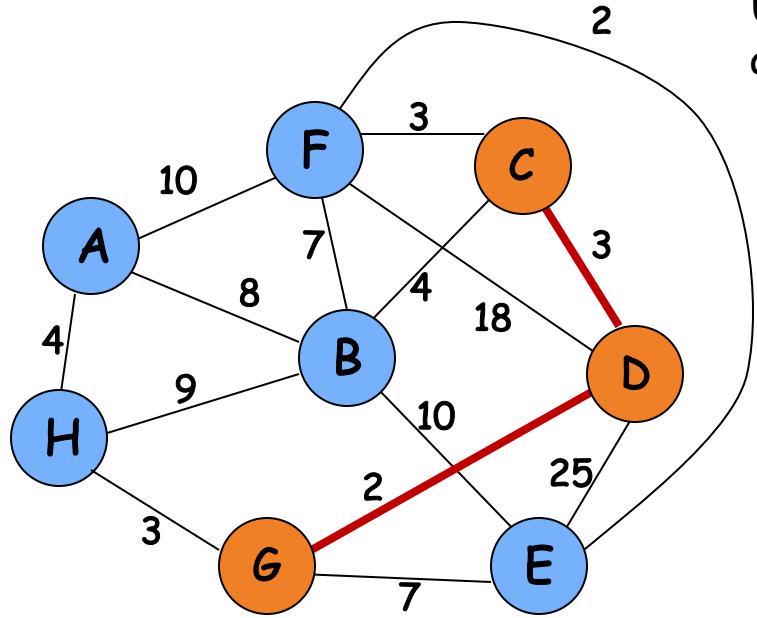
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G



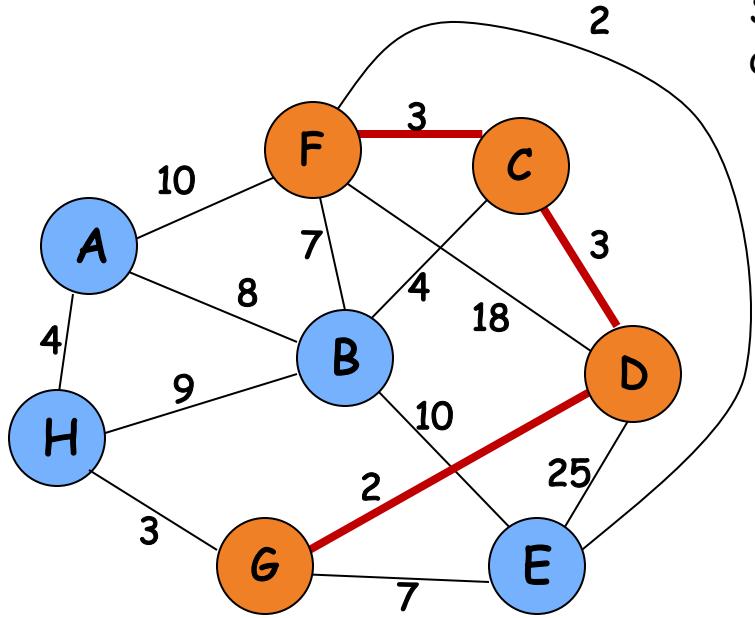
Select node with minimum distance

	S	d_v	p_v
A			
B			
C	T	3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G



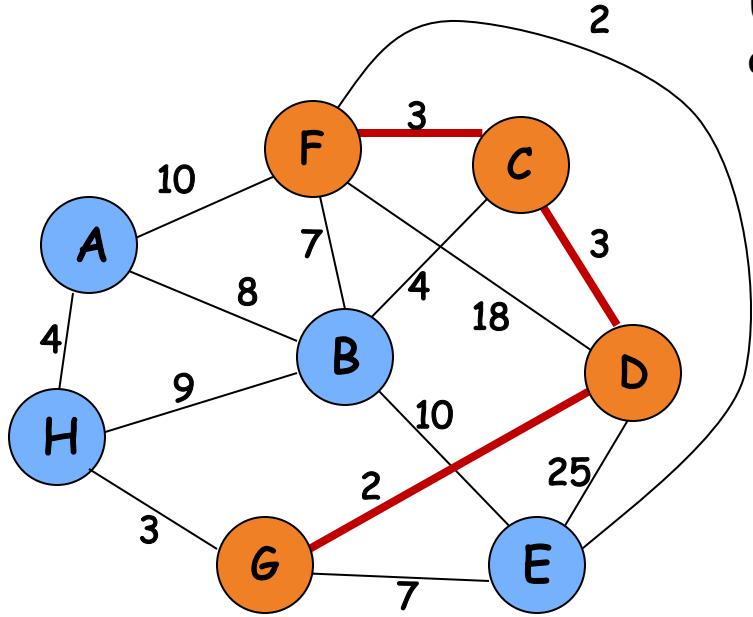
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F		3	C
G	T	2	D
H		3	G



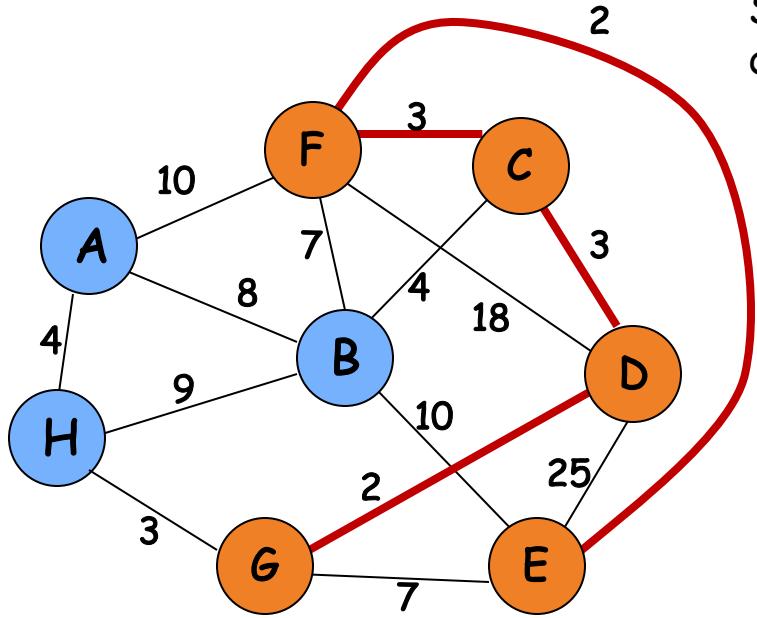
Select node with minimum distance

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F	T	3	C
G	T	2	D
H		3	G



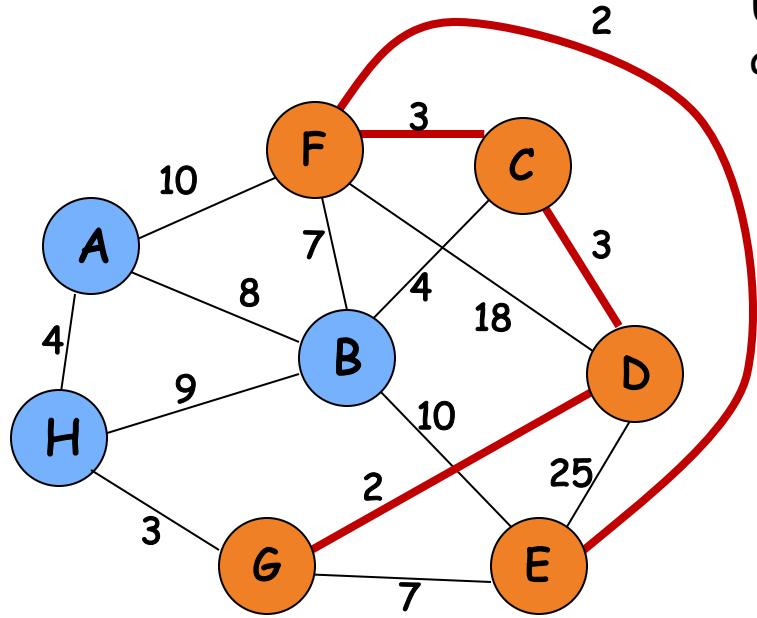
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

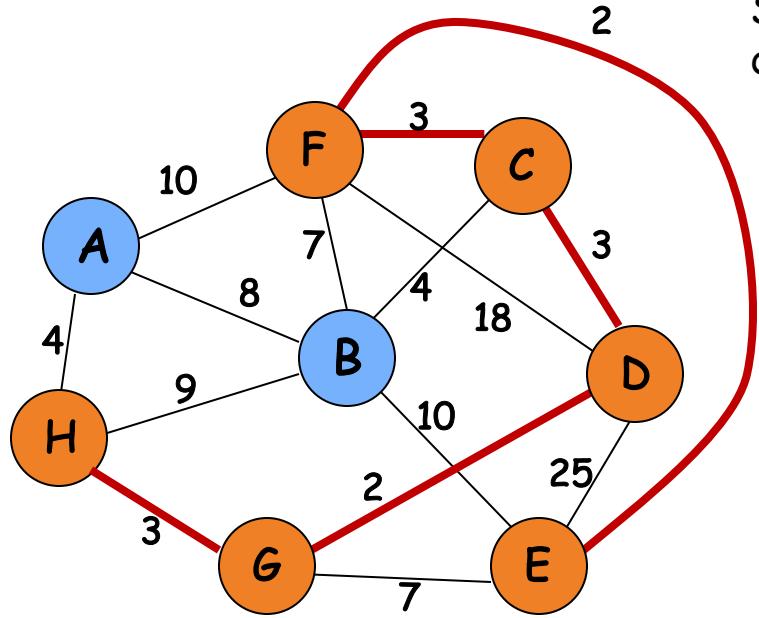
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

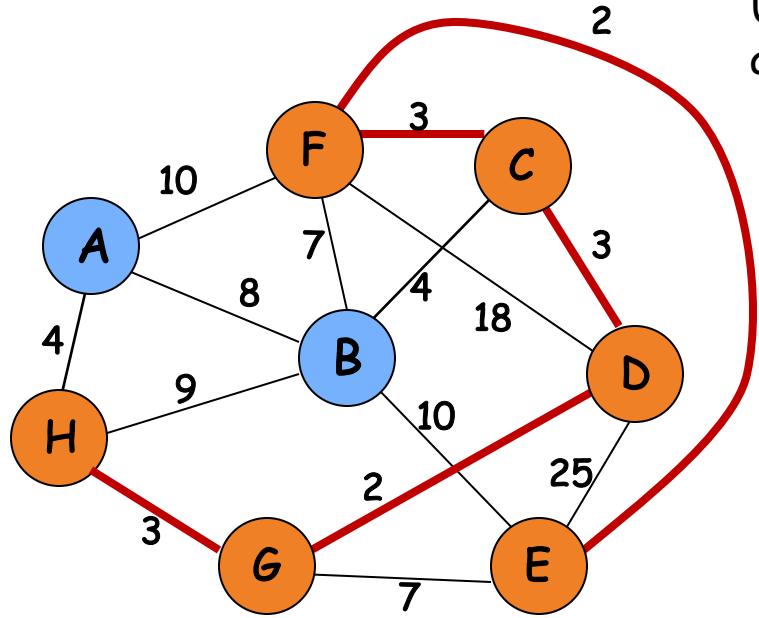
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



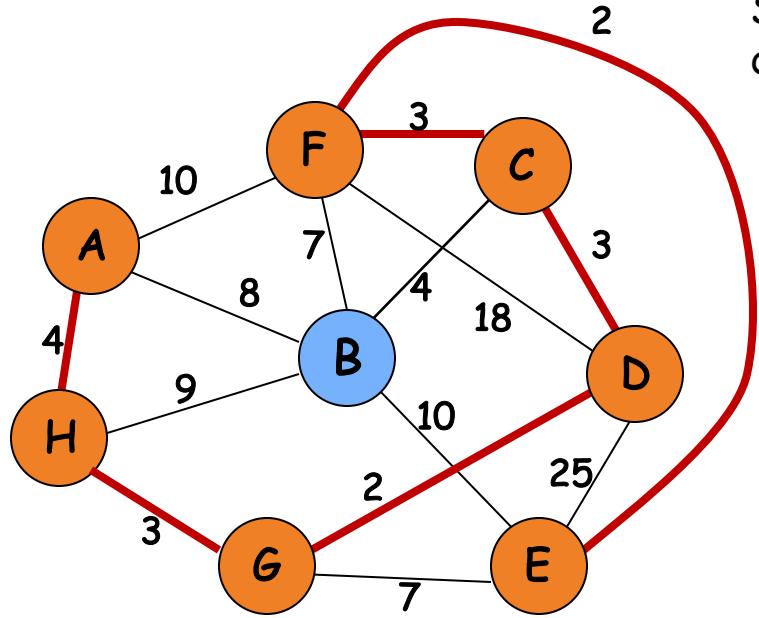
Select node with minimum distance

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



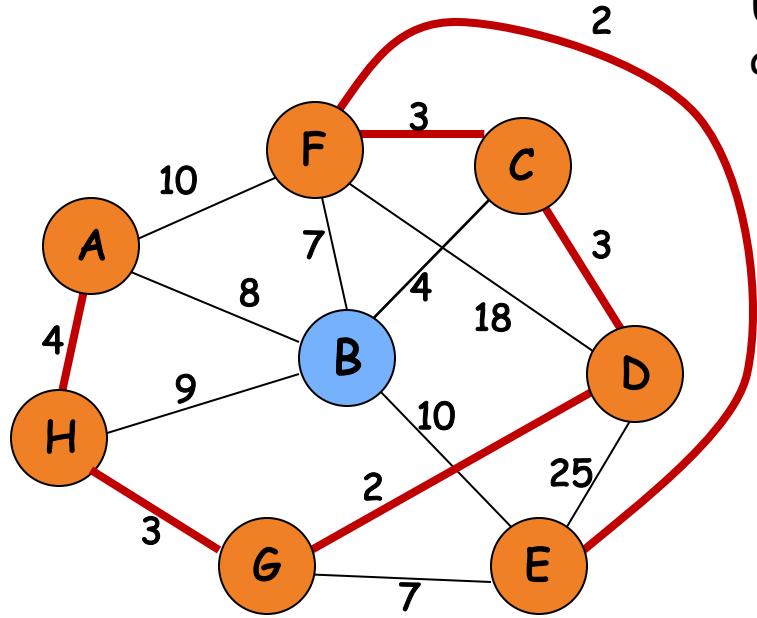
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

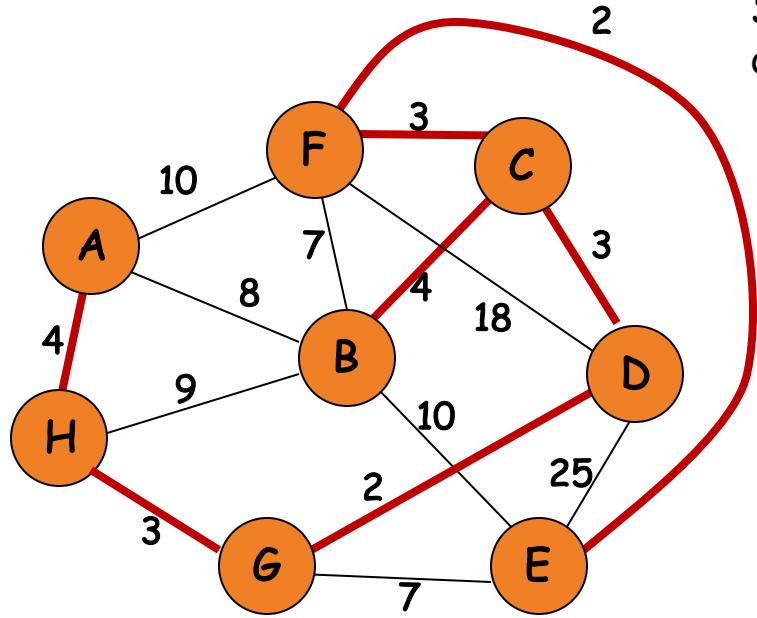
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

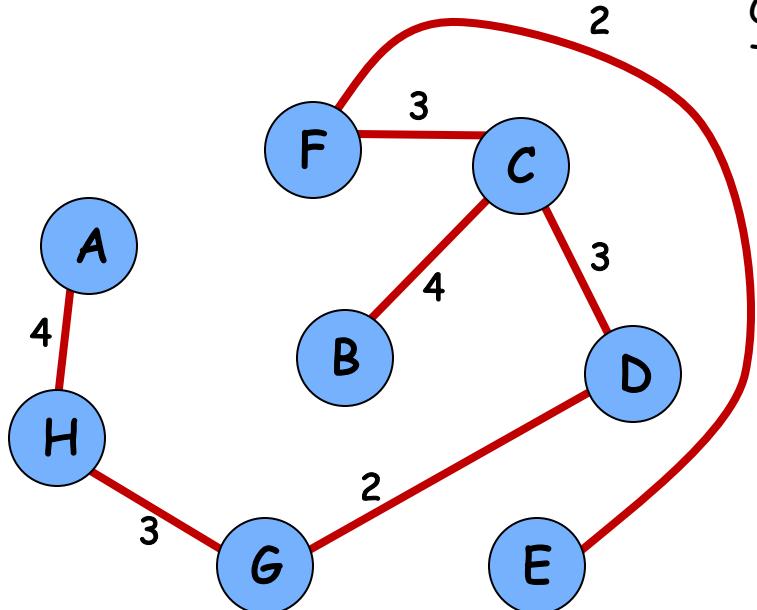
	S	d_v	p_v
A	S	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\sum d_v = 21$

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done!

Prim's Algorithm complexity

```
def prim(G, c) {  
    for v in V do  
        d[v] = infinity  
        parent[v] = nil  
    u = arbitrary vertex in V  
    d[u] = 0  
    Q = new PQ with items { (v, d[v]) for v in V }  
    S = ∅  
  
    while Q is not empty do  
        u = delete min element from Q  
        S = S ∪ { u }  
        for (u, v) incident to u do  
            if v ∉ S and c_e < d[v] then  
                parent[v] = u  
                decrease priority d[v] to c_e  
return parent
```

Similar analysis to Dijkstra's algorithm:

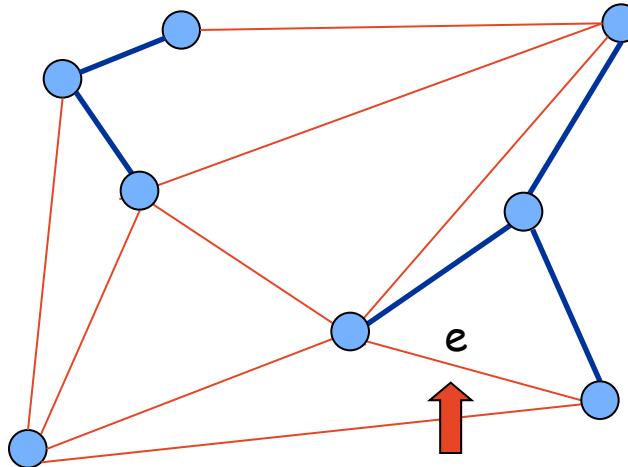
- $O(m \log n)$ using a heap
- $O(m + n \log n)$ using Fibonacci heap

Kruskal's Algorithm

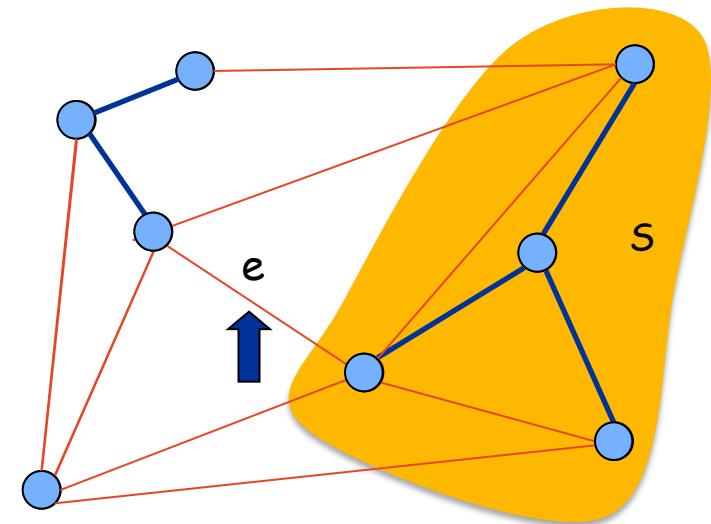
Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert $e = (u, v)$ into T according to cut property where $S = \text{set of nodes in } u\text{'s connected component}$.

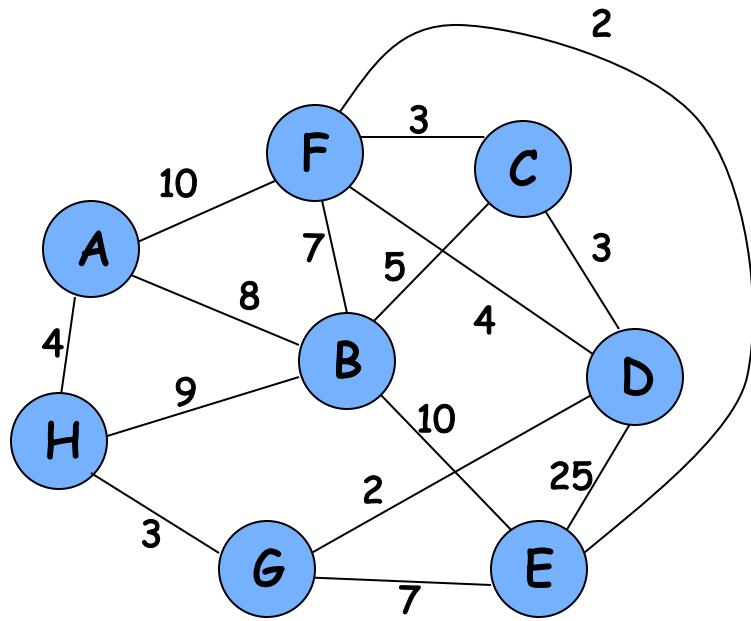


Case 1

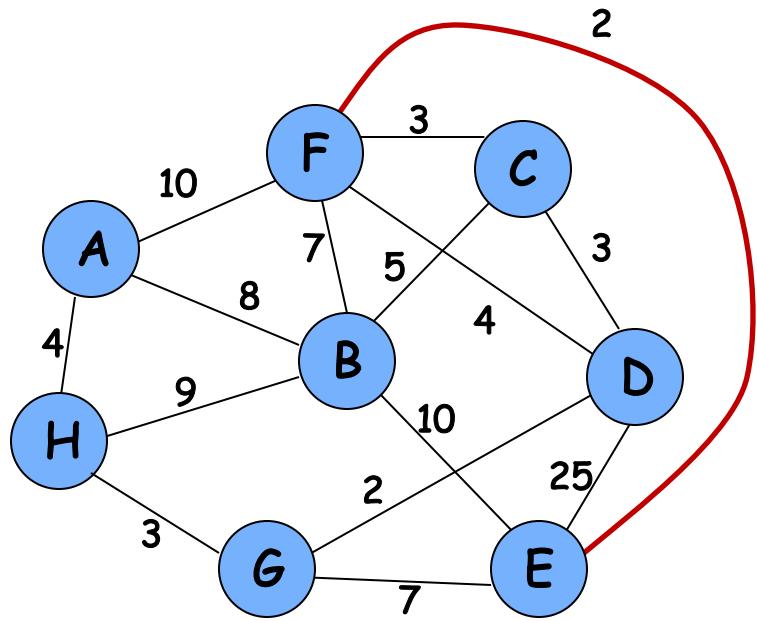


Case 2

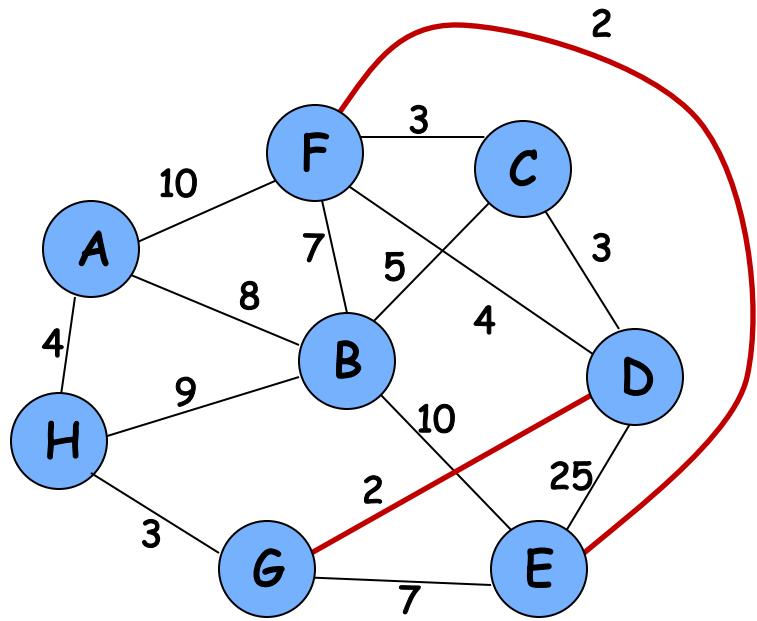
Walk-Through



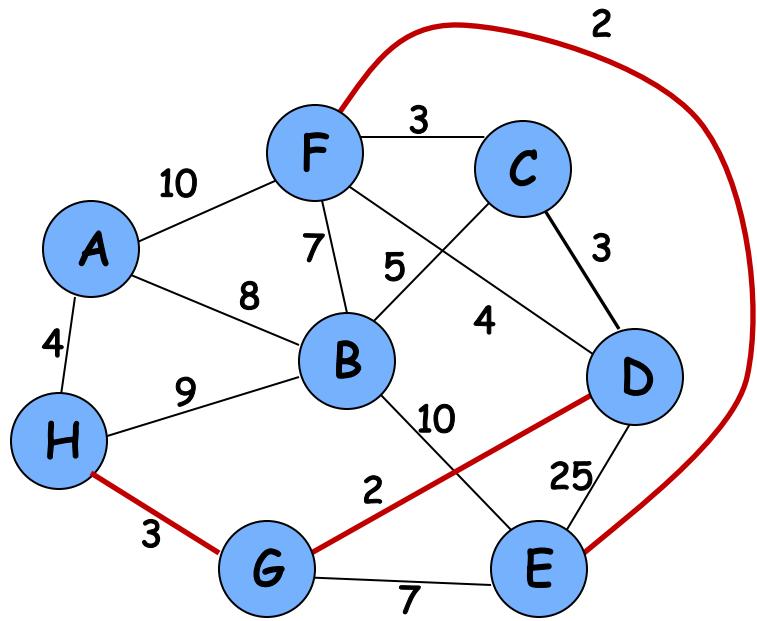
Walk-Through



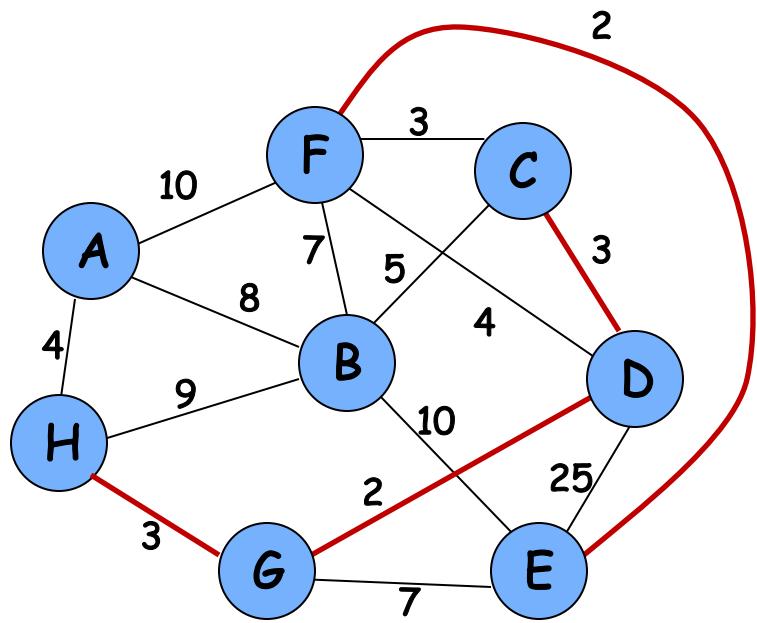
Walk-Through



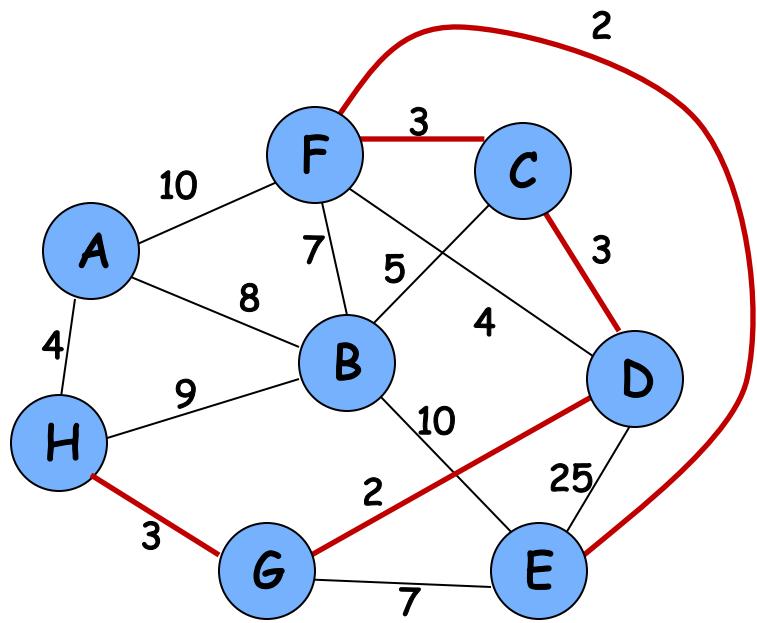
Walk-Through



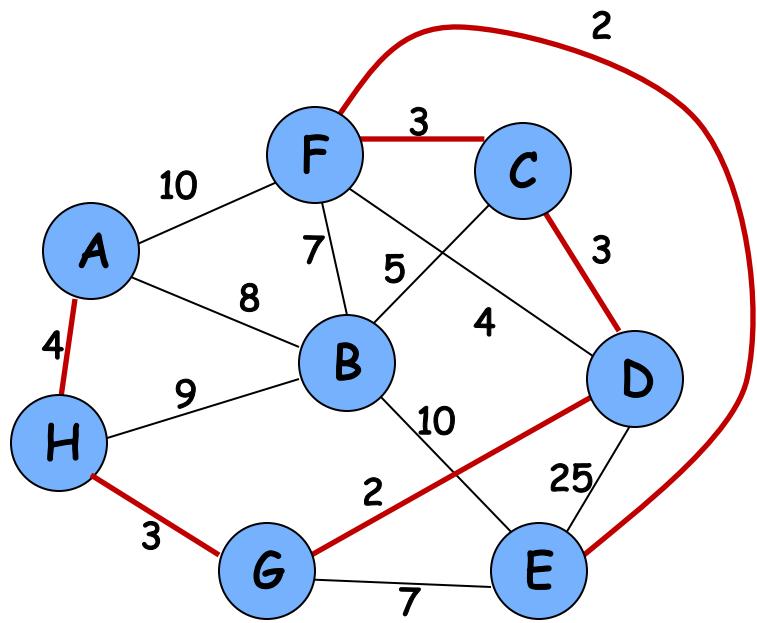
Walk-Through



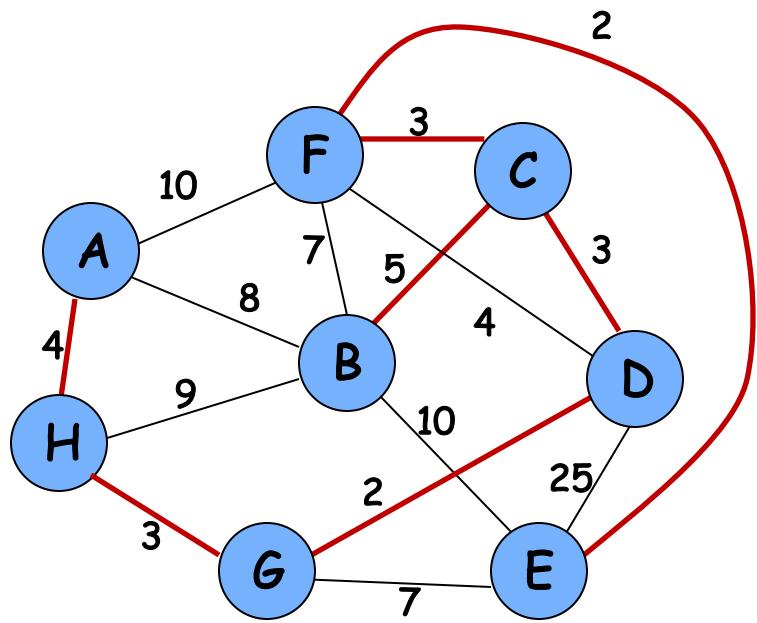
Walk-Through



Walk-Through



Walk-Through



Kruskal's Algorithm: Time complexity

Sorting edges takes $O(m \log m)$ time

We need to be able to test if adding a new edge creates a cycle, in which case we skip the edge

One option is to run DFS in each iteration to see if the number of connect components stays the same. This leads to $O(m n)$ time for the main loop

Can we do better?

Yes, keep track of the connected components with a data structure

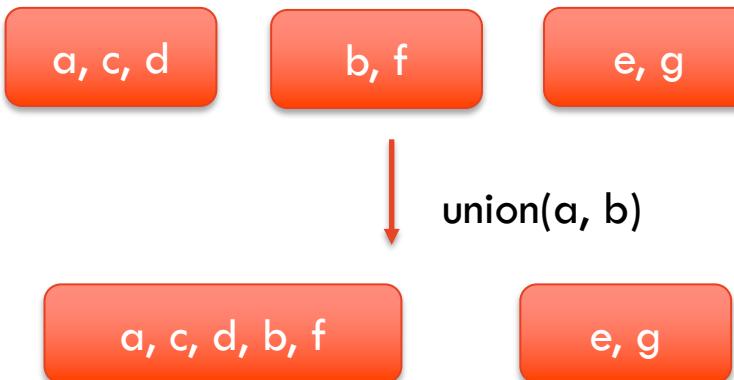
Union Find ADT

Data structure defined on a ground set of elements A

Used to keep track of an evolving partition of A

Supported operations:

- `make_sets(A)` : makes $|A|$ singleton sets with elements in A
- `find(a)` : returns an id for the set element a belongs to
- `union(a,b)` : union the sets elements a and b belong to



Kruskal's algorithm implementation

```
def Kruskal(G,c):  
  
    sort E in increasing c-value  
    answer = []  
    comp = make_sets(V)  
    for (u,v) in E do  
        if comp.find(u) != comp.find(v) then  
            answer.append( (u,v) )  
            comp.union(u, v)  
    return answer
```

Union find operations:

- `make_sets(A)` : one call with $|A| = |V|$
- `find(a)` : $2m$ calls
- `union(a,b)` : $n-1$ calls

Simple union-find implementation

Sets are represented with lists. And we keep an array mapping elements to the set they belongs to

- `make_sets(A)` creates and initialized the array
- `find(u)` is a simple lookup in the array
- `union(u,v)` adds elements in u's set to v's set

Time complexity:

- `make_sets(A)` takes $O(n)$ time, where $n = |A|$
- `find(u)` takes $O(1)$ time
- `union(u,v)` take $O(n)$ time

Kruskal's algorithm would run in $O(n^2)$ time

Better union-find implementation

Keep track of cardinality of each set. When taking the union of two sets change the smallest.

That way the union of two sets A and B take $O(\min(|A|, |B|))$

This way an element can change sets at most $O(\log n)$ time. So a sequence of n union operations takes at most $O(n \log n)$ time.

With this implementation, Kruskal's algorithm would run in $O(m \log n)$ time

Even better union-find implementation!

Keep track of sets using trees: For every node u in the graph we keep $\text{parent}[u]$

Taking the union of two roots u and v we set $\text{parent}[u] = v$ or $\text{parent}[v] = u$, depending who has the largest tree.

Perform $\text{find}(u)$ by following $\text{parent}[u]$ until reaching root r of tree.
Set $\text{parent}[v] = r$ for every node found along the way

It can be shown that a sequence of n union and $m > n$ find operations takes at most $O(m \alpha(n))$ time, where $\alpha(n)$ is a **very slow** growing function called the *Inverse Ackerman function*.

With this implementation, Kruskal's algorithm would run in $O(m \alpha(n))$ time after the edge weights are sorted.

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

For example, assuming all costs are integral, if we add i/n^2 to each edge e_i , then any MST under the perturbed weights is still an MST under the original weights.

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 9: The greedy method [GT 10]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



Greedy algorithms

Greedy algorithms can be some of the simplest algorithms to implement and design, but they're often among the hardest algorithms to prove correctness.

Today we will apply greedy strategies to three very different problems:

- Fractional knapsack
- Task scheduling
- Text compression

The Fractional Knapsack Problem



Given: A set S of n items, with each item i having

- b_i : a positive benefit
- w_i : a positive weight

Goal: Choose items with maximum total benefit of weight at most W .

Let x_i denote the amount we take of item i

Objective: maximize $\sum_{i \in S} b_i(x_i / w_i)$ [maximize benefit]

Constraint: $\sum_{i \in S} x_i \leq W$ [total weight is bounded]

$0 \leq x_i \leq w_i$ [individual weight is bounded]

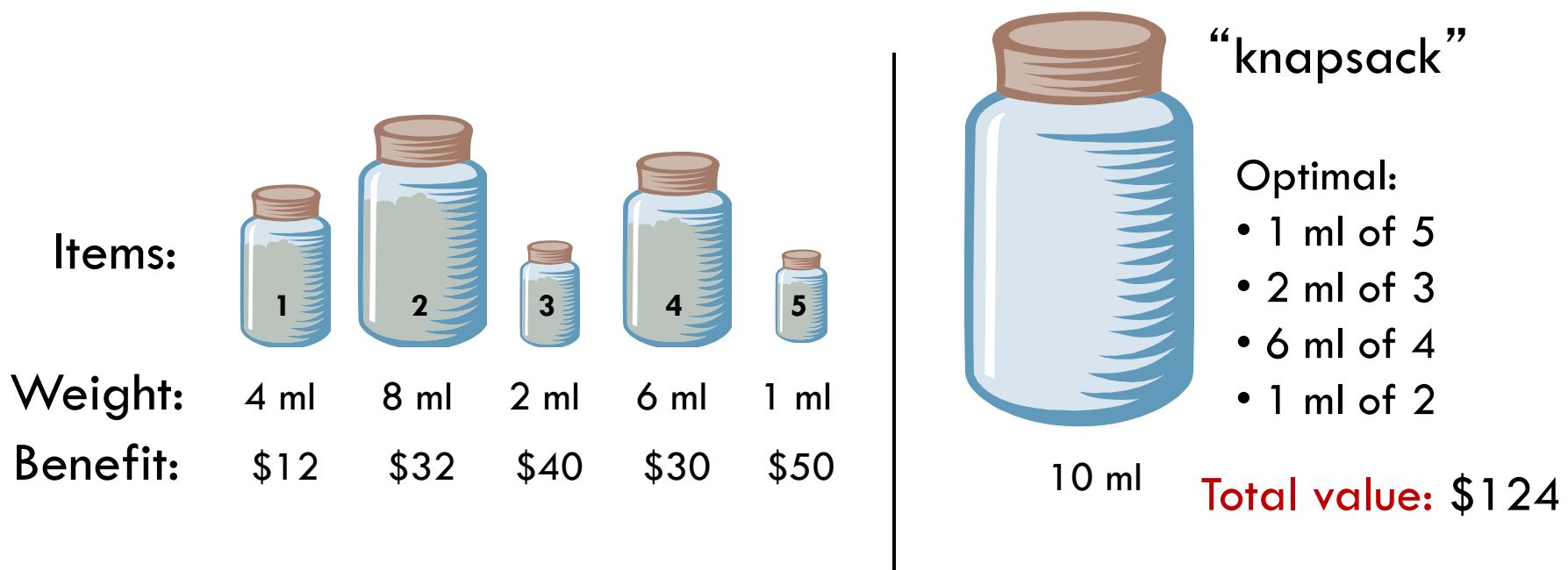
Example



Given: A set S of n items, with each item i having

- b_i - a positive benefit
- w_i - a positive weight

Goal: Choose items with maximum total benefit of weight at most W .



The Fractional Knapsack Algorithm



Initial configuration: no
items chosen

Each step: identify the
“best” item available and
add as much as possible
(all of it if you can) to the
knapsack

What defines “**best**” choice
of item to add next?

```
def fractional_knapsack(b, w, W):  
  
    # initialization  
    x = array of size |b| of zeros  
    curr = 0  
  
    # iteratively make greedy choice  
    while curr < W do  
        i = “best” item not yet chosen  
        x[i] = min(w[i], W - curr)  
        curr = curr + x[i]  
    return x
```

Different strategies.



A greedy choice: Keep taking as much as possible of the “**best**” item, where best means:

[highest benefit]: Select items with highest benefit.

[smallest weight]: Select items with smallest weight.

[benefit/weight]: Select items with highest benefit to weight ratio.

Each of these defines a different greedy strategy for this problem.

What's “best”?



Greedy choice: Keep taking the “best” item.

[highest benefit]: Select items with highest benefit.

1 ml of 5 → \$50

2 ml of 3 → \$40

7 ml of 2 → \$28

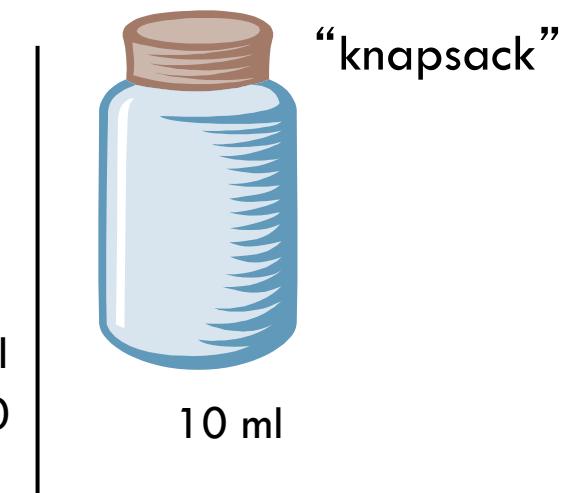
Total value: \$118

Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50



What's “best”?



Greedy choice: Keep taking the “best” item.

[smallest weight]: Select items with smallest weight.

1 ml of 5 → \$50

2 ml of 3 → \$40

4 ml of 1 → \$12

3 ml of 4 → \$15

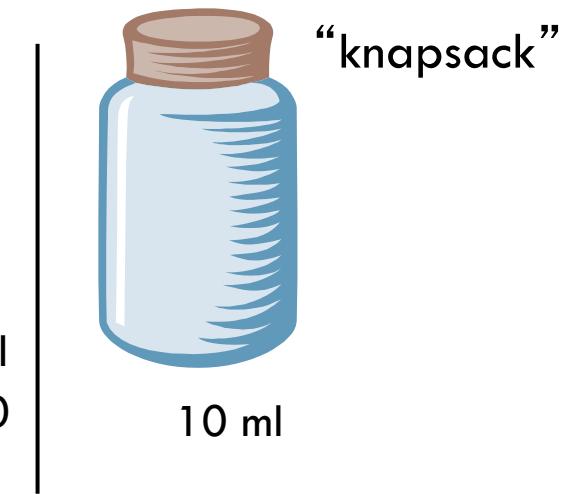
Total value: \$117

Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50



What's “best”?



Greedy choice: Keep taking the “best” item.

[benefit/weight]: Select items with highest benefit to weight ratio.

1 ml of 5 → \$50

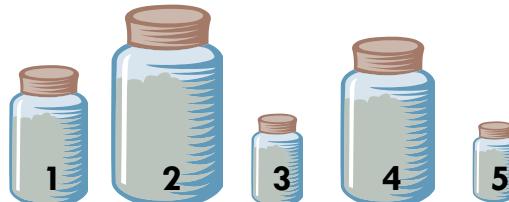
2 ml of 3 → \$40

6 ml of 4 → \$30

1 ml of 2 → \$4

Total value: \$124

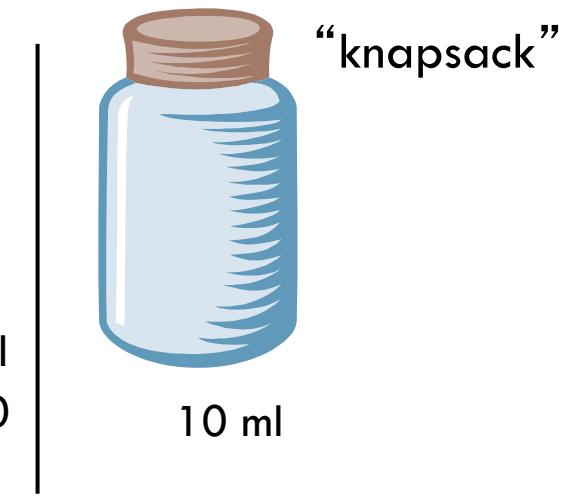
Items:



Weight: 4 ml 8 ml 2 ml 6 ml 1 ml

Benefit: \$12 \$32 \$40 \$30 \$50

Benefit/ml: 3 4 20 5 50



The Fractional Knapsack Algorithm: Correctness

Theorem: The greedy strategy of picking item with highest benefit to weight ratio computes an optimal solution.

Proof (sketch):

- Use an exchange argument
- Assume for simplicity that all ratios are different $b_i/w_i \neq b_k/w_k$
- Consider some feasible solution x different than the greedy one
- There must be items i and k s.t. $x_i < w_i$, $x_k > 0$ and $b_i/w_i > b_k/w_k$
- If we replace some k with some of i , we get a better solution
- How much? $\min\{w_i - x_i, x_k\}$
- Thus, there is no better solution than the greedy one

The Fractional Knapsack Algorithm: Complexity

Sort items by their benefit-to-weight values, and then process them in this order.

Require $O(n \log n)$ time to sort the items and then $O(n)$ time to process them in the for-loop.

```
def fractional_knapsack(b, w, W):
    # initialization
    x = array of size |b| of zeros
    curr = 0

    # iteratively do greedy choice
    for i in descending b[i]/w[i] order do
        x[i] = min(w[i], W - curr)
        curr = curr + x[i]
    return x
```

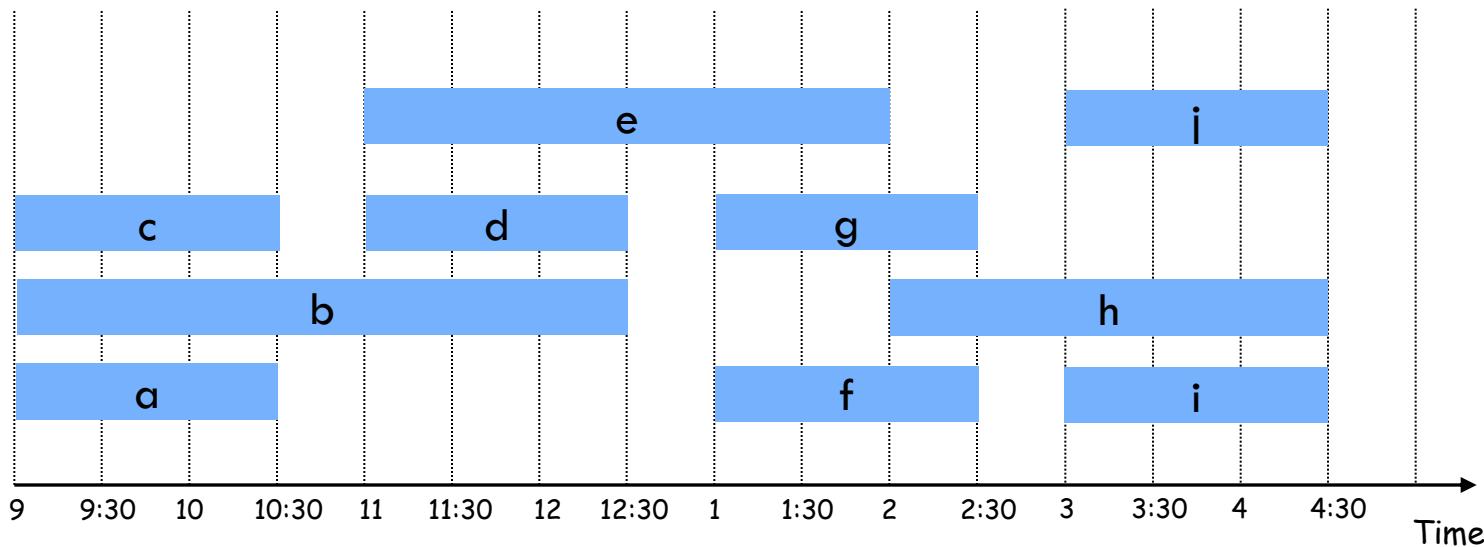
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i .

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses 4 classrooms to schedule 10 lectures.



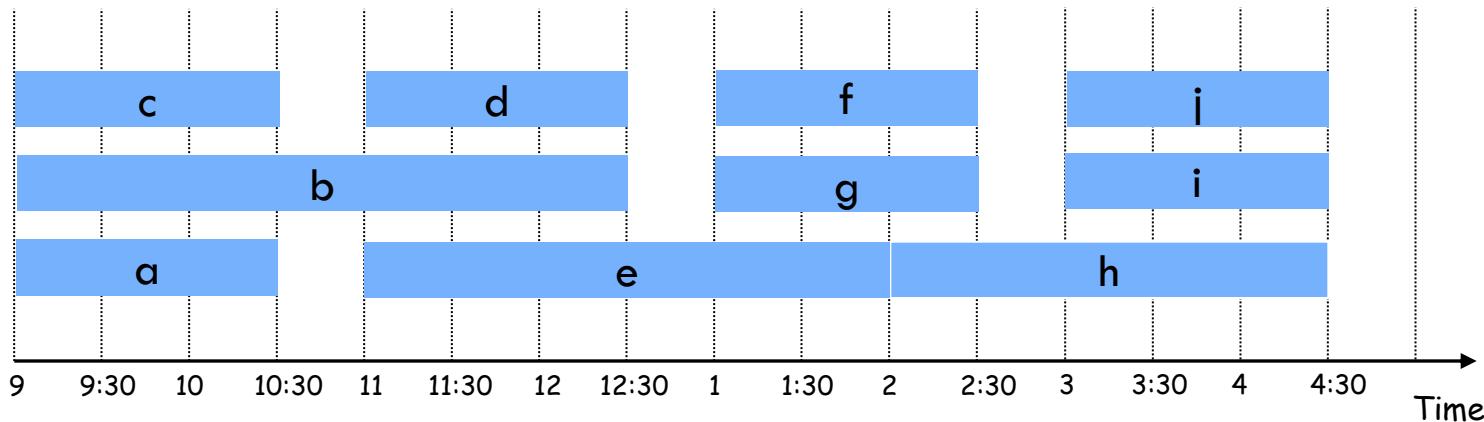
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i .

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses only 3!



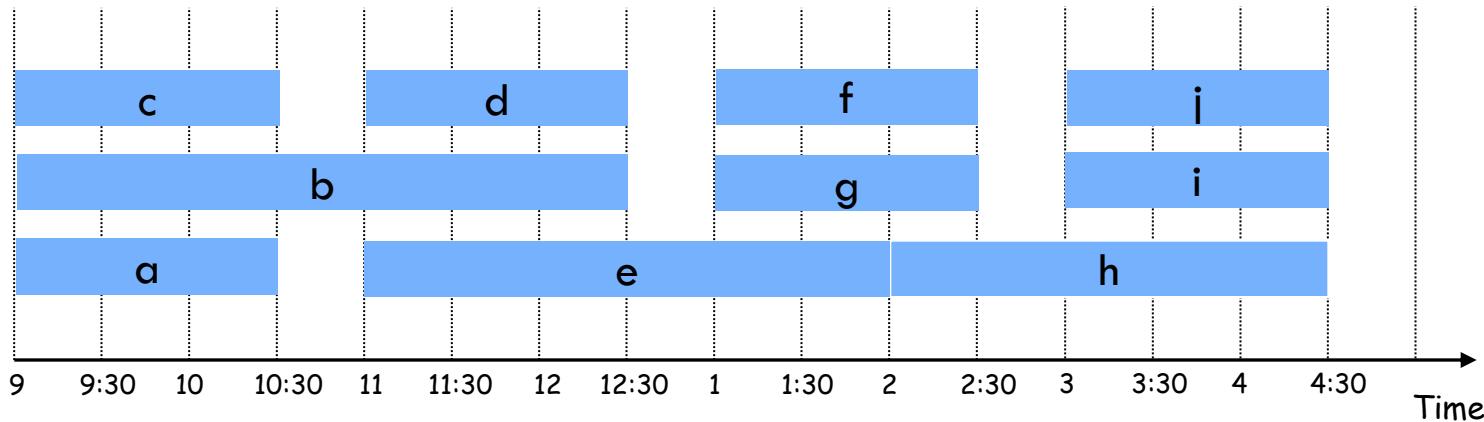
Interval Partitioning: Lower bound

Definition: The **depth** of a set of open intervals is the maximum number that contain any given time.

Observation: Number of classrooms needed \geq depth. Why?

Example: Depth of schedule below is 3 [a, b, c all contain 9:30]
 \Rightarrow schedule below is optimal.

Question: Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):
    # initialization
    sort intervals in increasing starting time order
    d = 0      # number of allocated classrooms

    # iteratively do greedy choice
    for i in increasing starting time order do
        if lecture i is compatible with some classroom k then
            schedule lecture i in classroom  $1 \leq k \leq d$ 
        else
            allocate a new classroom  $d+1$ 
            schedule lecture i in classroom  $d+1$ 
            d = d+1
    return d
```

Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):
    # initialization
    sort intervals in increasing starting time order
    d = 0      # number of allocated classrooms

    # iteratively do greedy choice
    for i in increasing starting time order do
        if lecture i is compatible with some classroom k then
            schedule lecture i in classroom  $1 \leq k \leq d$ 
        else
            allocate a new classroom  $d+1$ 
            schedule lecture i in classroom  $d+1$ 
            d = d+1
    return d
```

Implementation: $O(n \log n)$.

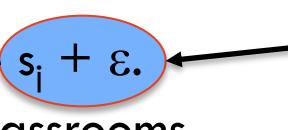
- For each classroom k , maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation: Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem: Greedy algorithm is optimal.

Proof:

- $d = \text{number of classrooms that the greedy algorithm allocates.}$
- Classroom d is opened because we needed to schedule a job, say i , that is incompatible with all $d-1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i .
- Thus, we have d lectures overlapping at time $s_i + \varepsilon$. 
just
after
time s_i
- Key observation \Rightarrow all schedules use $\geq d$ classrooms.

[Greedy algorithm stays “ahead”]

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y
(saves memory and/or bandwidth)

Input:

WWWWWWWWWWWWWWWWBWWWWWWWWWWWWWWWWBWWWWWWWW
WW

Run length encoding (very simple approach):

12W1B12W3B24W1B14W

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y
(saves memory and/or bandwidth)

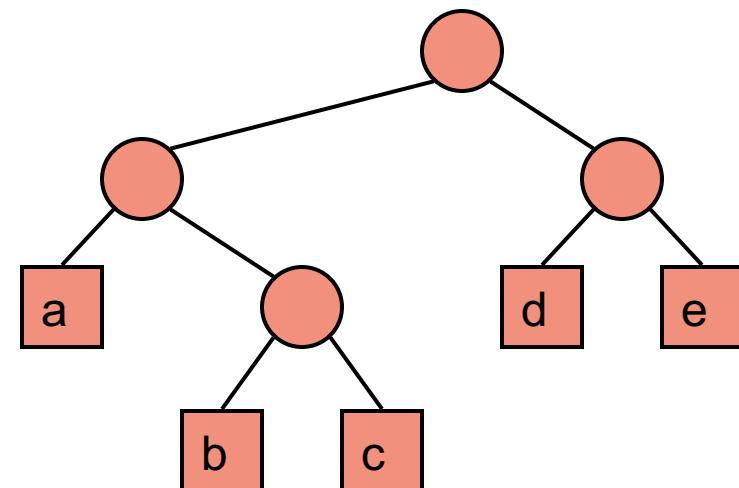
A better approach: **Huffman encoding**

- Let C be the set of characters in X
- Compute frequency $f(c)$ for each character c in C
- Encode high-frequency characters with short code words
- No code word is a prefix for another code
- Use an optimal encoding tree to determine the code words

Encoding Tree Example

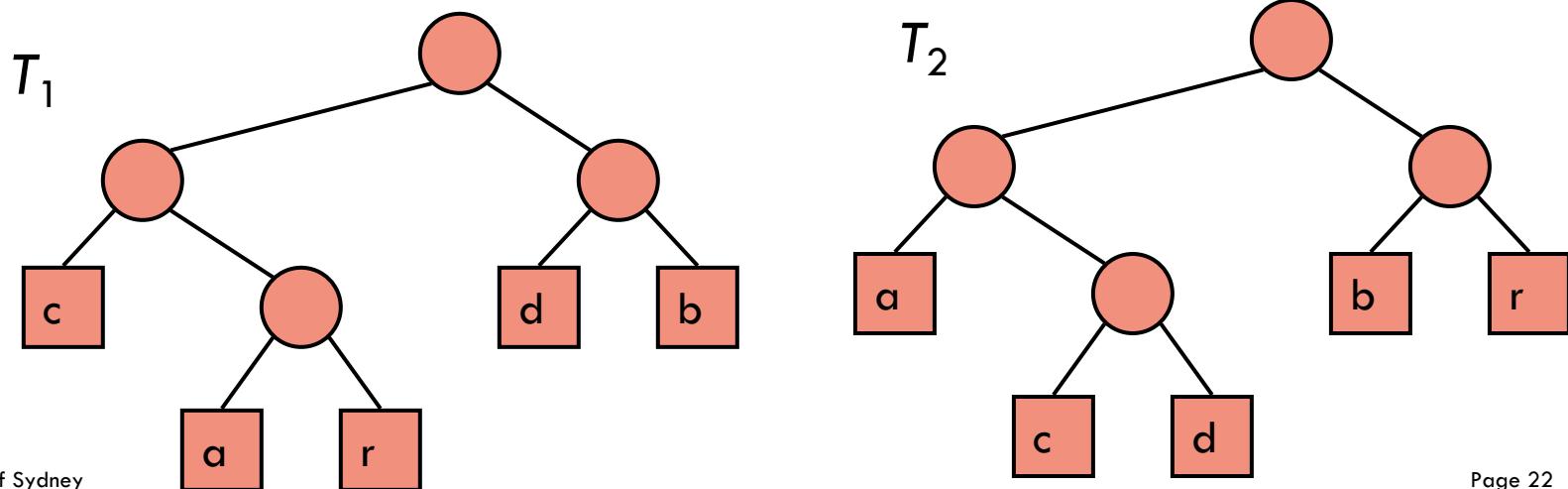
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
 - Each external node stores a character
 - The code-word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X , we want to find a prefix code for the characters of X that yields a small encoding for X
 - Frequent characters should have short code-words
 - Rare characters should have long code-words
- Example
 - $X = \text{abracadabra}$
 - T_1 encodes X into 29 bits
 - T_2 encodes X into 24 bits



Huffman's Algorithm

Given a string X , Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of X

It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X

The algorithm builds the encoding tree from the bottom up, merging trees as it goes along, using a priority queue to guide the process

End result minimizes bits needed to encode X :

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$

Huffman's Algorithm

```
def huffman(C, f):  
  
    # initialize priority queue  
    Q = empty priority queue  
    for c in C do  
        T = single-node binary tree storing c  
        Q.insert(f[c], T)  
  
    # merge trees while at least two trees  
    while Q.size() > 1 do  
        f1, T1 = Q.remove_min()  
        f2, T2 = Q.remove_min()  
        T = new binary tree with T1/T2 as left/right subtrees  
        f = f1 + f2  
        Q.insert(f, T)  
  
    # return last tree  
    f, T = Q.remove_min()  
    return T
```

Example

$X = \text{abracadabra}$

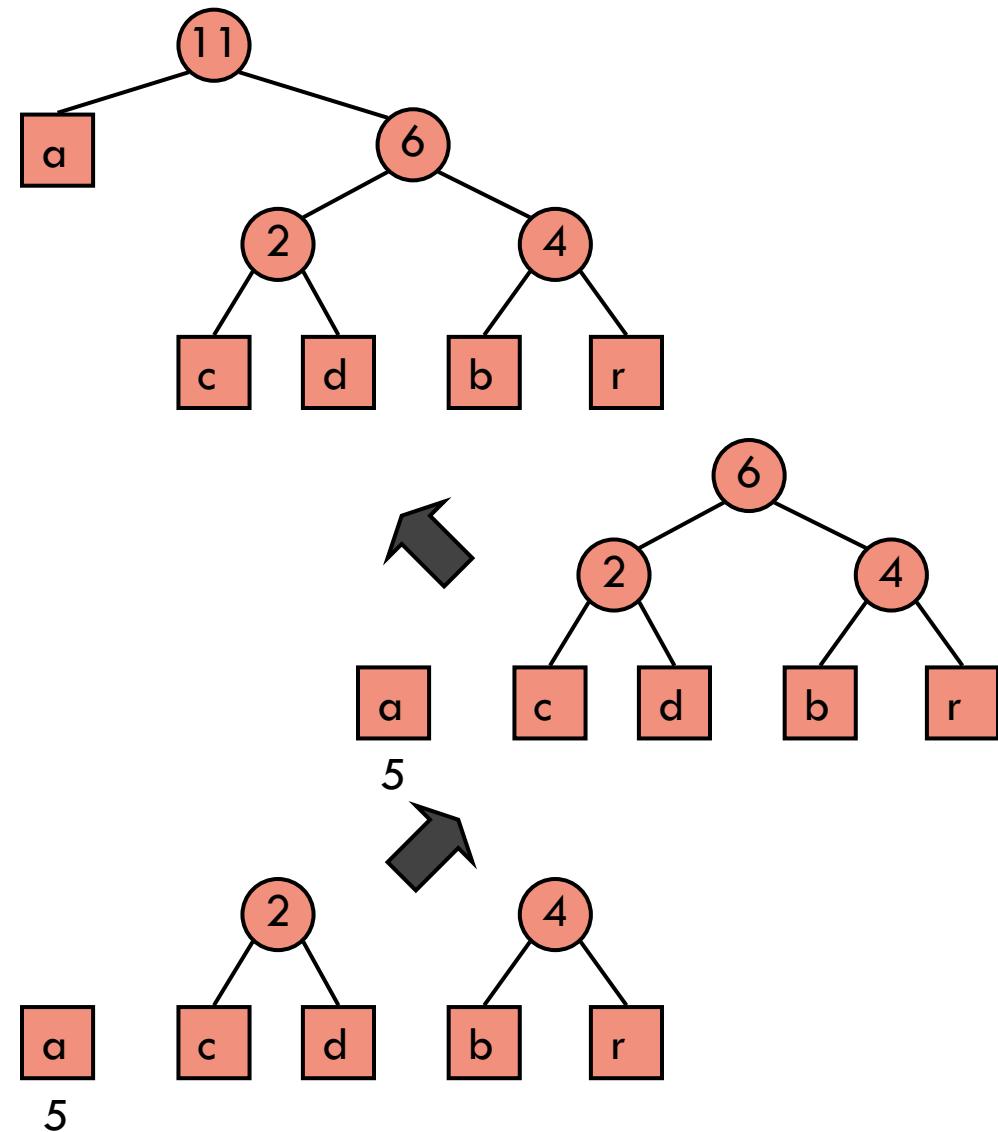
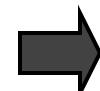
Frequencies

a	b	c	d	r
5	2	1	1	2

a	b	c	d	r
5	2	1	1	2



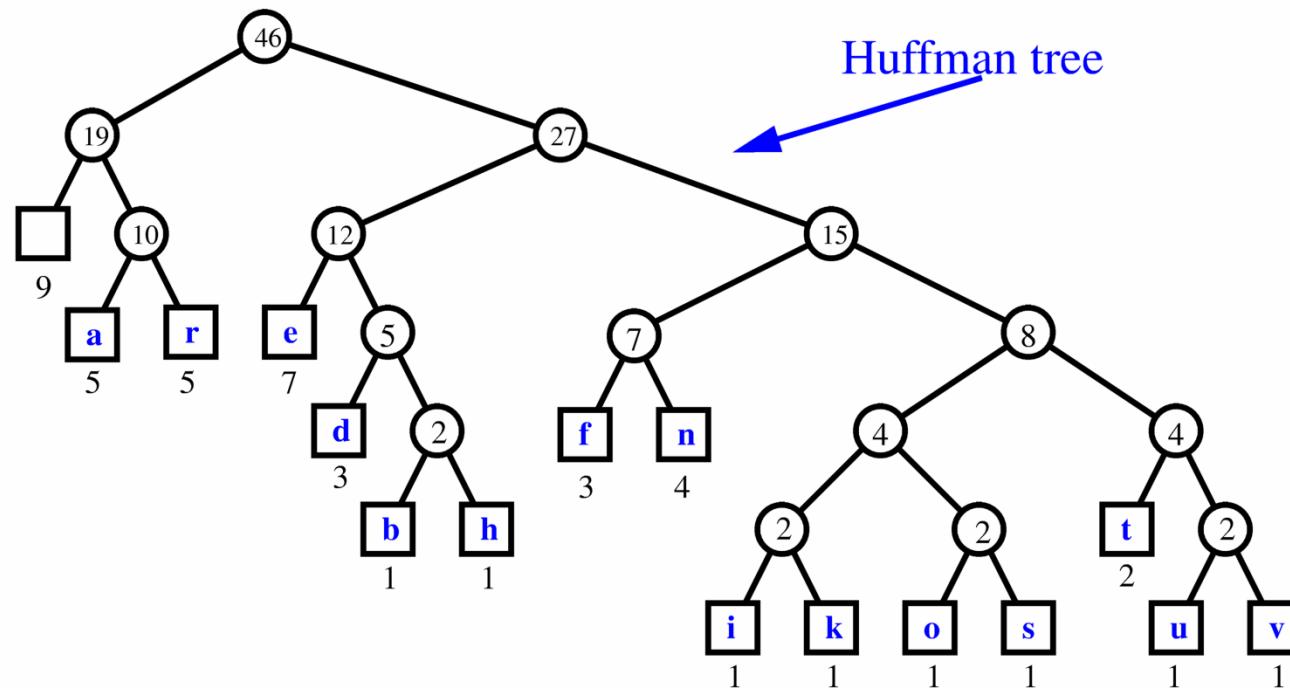
a	b	c	d	r
5	2	1	1	2



Extended Huffman Tree Example

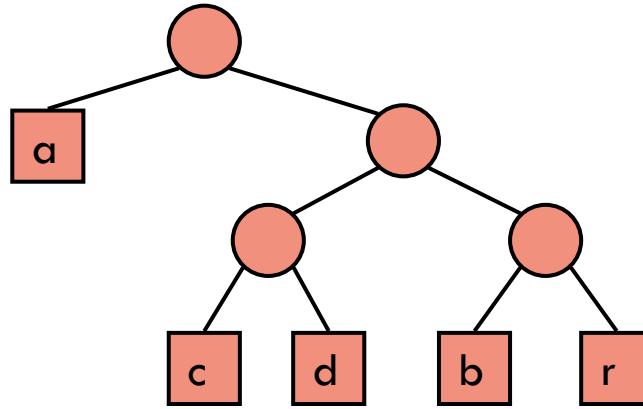
String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v	
Frequency		9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Huffman's Algorithm Correctness

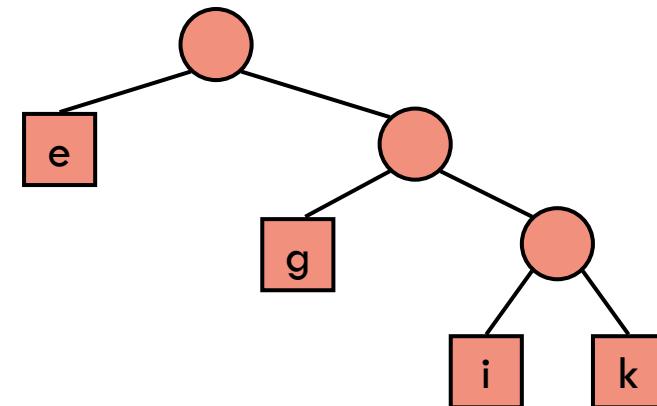
Obs: Every tree encoding has a pair of leaves that are *siblings*.



Huffman's Algorithm Correctness

Obs: In an optimal tree encoding T for any a and b in C , if $\text{depth}_T(a) < \text{depth}_T(b)$ then $f(a) \geq f(b)$.

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$

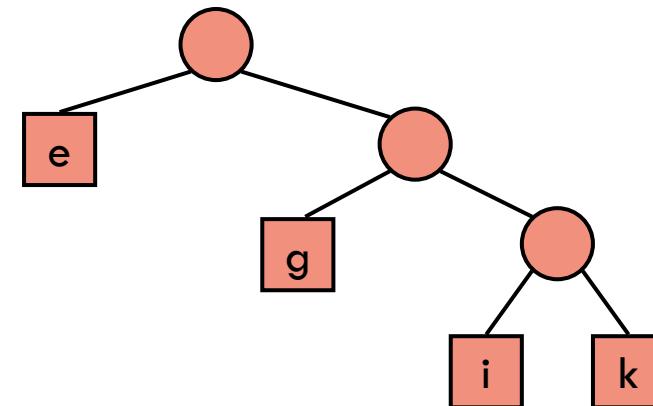


For example, if $f(e) < f(g)$
then swapping them leads
to shorter encoding

Huffman's Algorithm Correctness

Obs: There is an optimal tree encoding T where the two sibling leaves furthest from the root have lowest frequency.

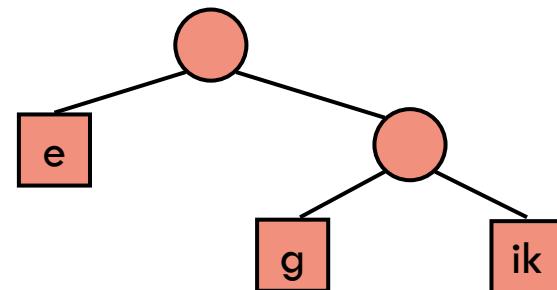
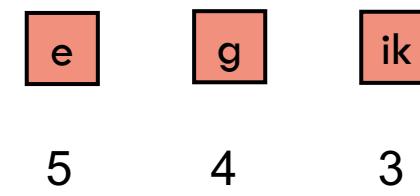
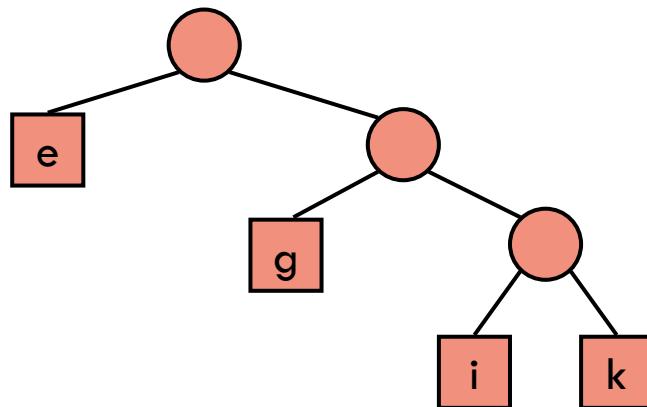
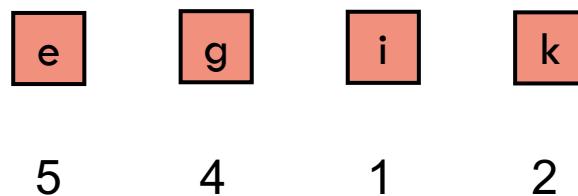
$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c)$$



For example, characters **i** and **k** have lowest frequency

Huffman's Algorithm Correctness

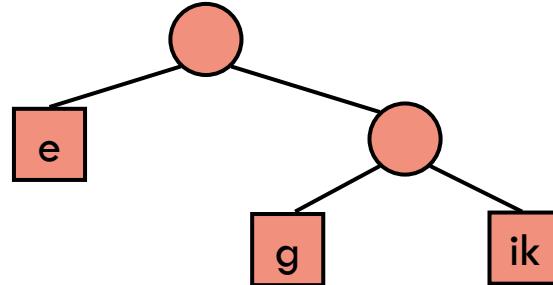
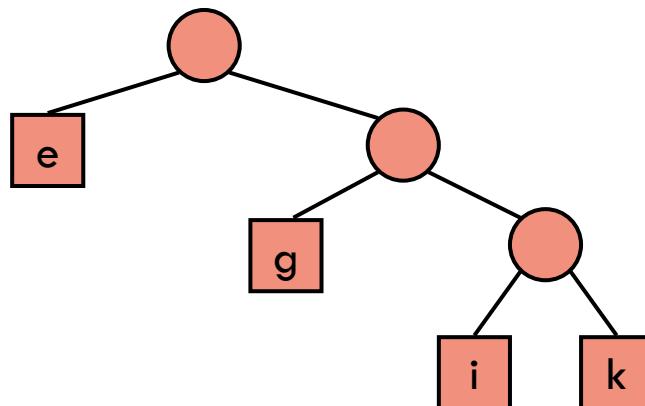
Obs: If we combine the two lowest frequency characters to get a new instance (C', f') , an optimal tree encoding T' for (C', f') can be expanded to get an optimal tree encoding T for (C, f)



Huffman's Algorithm Correctness

Obs: If we combine the two lowest frequency characters to get a new instance (C', f') , an optimal tree encoding T' for (C', f') can be expanded to get an optimal tree encoding T for (C, f)

$$\begin{aligned} \sum_{c \text{ in } C} f(c) * \text{depth}_T(c) - \sum_{c \text{ in } C'} f'(c) * \text{depth}_{T'}(c) \\ = f(i) * \text{depth}_T(i) + f(k) * \text{depth}_T(k) - f'(ik) * \text{depth}_{T'}(ik) \\ = f(i) + f(k) \end{aligned}$$



Huffman's Algorithm Correctness

Thm: Huffman's algorithm computes a minimum length tree encoding of (C, f)

Proof (by induction):

- If $|C| = 1$ then the encoding is trivially optimal
- If $|C| > 1$ then let (C', f') be the contracted instance
- By inductive hypothesis, the tree encoding T' constructed for (C', f') is optimal
- Recall that

$$\sum_{c \text{ in } C} f(c) * \text{depth}_T(c) = \sum_{c \text{ in } C'} f'(c) * \text{depth}_{T'}(c) + f(i) + f(k)$$

thus, the tree T is optimal for (C, f)

Huffman's Algorithm

```
def huffman(C, f):  
  
    # initialize priority queue  
    Q = empty priority queue  
    for c in C do  
        T = single-node binary tree storing c  
        Q.insert(f[c], T)  
  
    # merge trees while at least two trees  
    while Q.size() > 1 do  
        f1, T1 = Q.remove_min()  
        f2, T2 = Q.remove_min()  
        T = new binary tree with T1/T2 as left/right subtrees  
        f = f1 + f2  
        Q.insert(f, T)  
  
    # return last tree  
    f, T = Q.remove_min()  
    return T
```

Time complexity is dominated by PQ ops, which using heap take $O(|C| \log |C|)$ time

Greedy algorithms recap

Greedy heuristics are easy to design but they are not always optimal. And when they are, small modifications of the problem can render them suboptimal:

- 0-1 knapsack is hard
- if tasks/lectures have special needs the problem is hard
- if we use non-binary encodings, Huffman does not work

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

Lecture 10: Randomized Algorithms [GT 19.1 and 19.6]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Reasons for using randomization:

- Sampling data from a large population or dataset
- Avoid pathological worst-case examples
- Avoid predictable outcomes
- Allow for simpler algorithms

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Today:

- Generating random permutations
- Treaps
- Skip lists

Generating random permutations

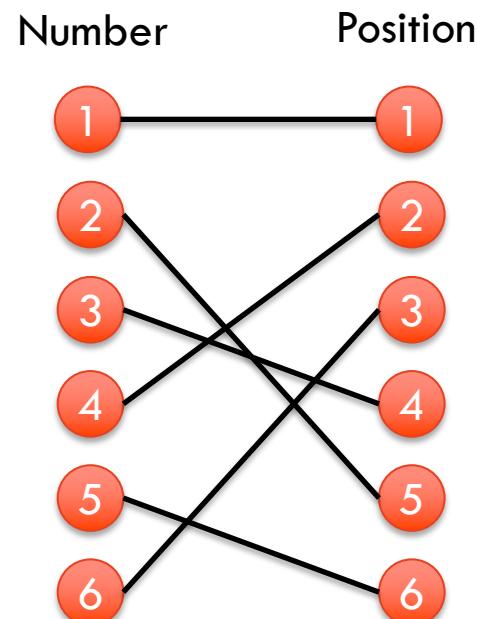
Input: An integer n .

Output: A permutation of $\{1, \dots, n\}$ chosen uniformly at random, i.e., every permutation has the same probability of being generated.

Example:

$n = 6$

$<1,4,6,3,2,5>$



Generating random permutations

What are random permutations used for?

- Many algorithms whose input is an array perform better in practice after randomly permuting the input (for example, QuickSort).
- Can be used to sample k elements without knowing k in advance by picking the next element in the permuted order when needed.
- Can be used to assign scarce resources.
- Can be a building block for more complex randomized algorithms.

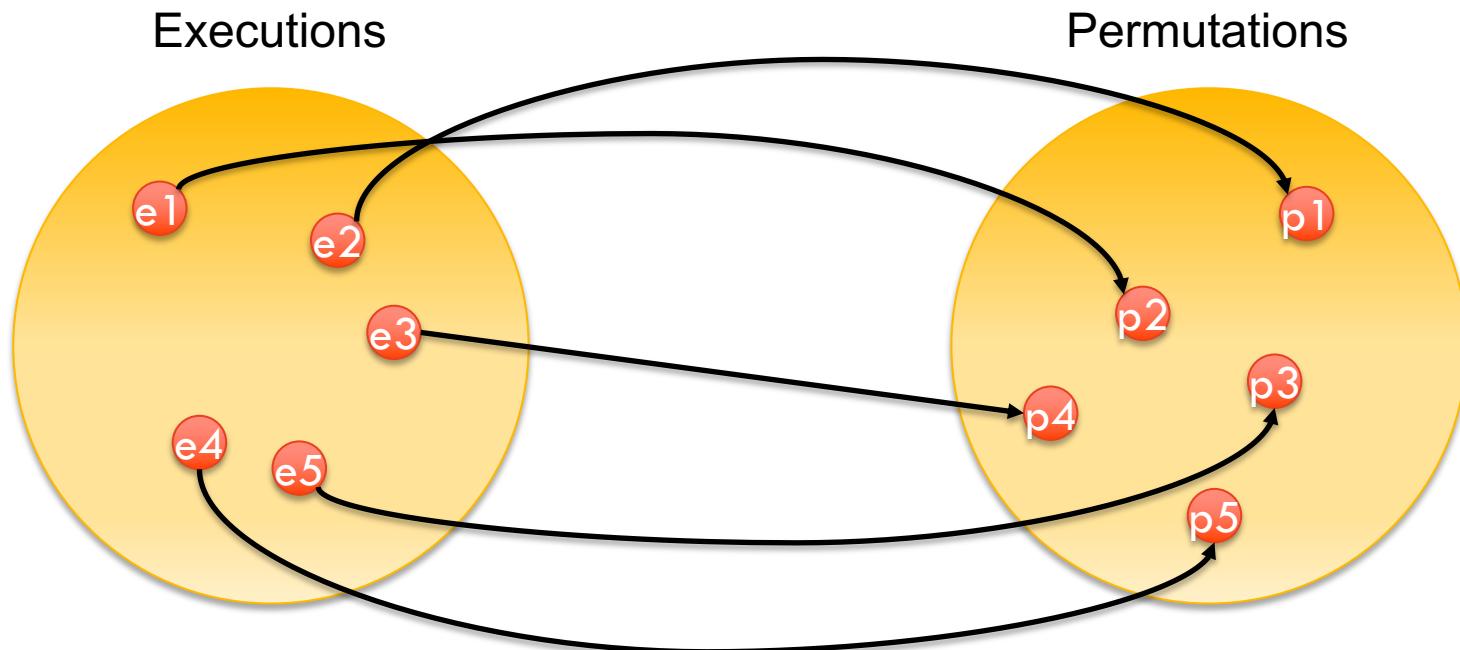
First (incorrect) attempt

```
def permute(A):  
  
    # permute A in place  
    n = length of array A  
  
    for i in [0:n] do  
        # swap A[i] with random position  
        j = pick_uniformly_at_random([0:n])  
        A[i], A[j] = A[j], A[i]  
  
    return A
```

Note that since j is picked at random, different executions lead to different outcomes

So, why is this incorrect?

For all permutations to be equally likely, we want that every permutation is generated by the same number of possible executions.



First (incorrect) attempt: Analysis

Number of executions:

$$n * n * n * \dots * n = n^n$$


n times

Number of permutations:

$$1 * 2 * 3 * \dots * n = n!$$

n^n isn't divisible by $n!$

Example:

$$n = 3$$

$$n^n = 27$$

$$n! = 6$$

27 isn't a multiple of 6, so some permutations are more likely than others.

```
def permute(A):  
    # permute A in place  
    n = length of array A  
  
    for i in [0:n] do  
        # swap A[i] with random position  
        j = pick_uniformly_at_random([0:n])  
        A[i], A[j] = A[j], A[i]  
  
    return A
```

Second attempt

```
def FisherYates(A):  
  
    # permute A in place  
    n = length of array A  
  
    for i in [0:n] do  
        # swap A[i] with random position  
        j = pick_uniformly_at_random([i:n])  
        A[i], A[j] = A[j], A[i]  
  
    return A
```

Note that since j is picked at random, different executions lead to different outcomes

Second attempt: Analysis

Number of executions:

$$1 * 2 * 3 * \dots * n = n!$$

Number of permutations:

$$1 * 2 * 3 * \dots * n = n!$$

Observation: Every execution leads to a different permutation.

```
def FisherYates(A):
    # permute A in place
    n = length of array A

    for i in [0:n] do
        # swap A[i] with random position
        j = pick_uniformly_at_random([i:n])
        A[i], A[j] = A[j], A[i]

    return A
```

Example: To generate $<3,2,4,1>$ starting from $<1,2,3,4>$

$<1,2,3,4> \rightarrow <3,2,1,4>$, $i=0$ and $j=2$

$<3,2,1,4> \rightarrow <3,2,1,4>$, $i=1$ and $j=1$

$<3,2,1,4> \rightarrow <3,2,4,1>$, $i=2$ and $j=3$

$<3,2,4,1> \rightarrow <3,2,4,1>$, $i=3$ and $j=3$

Second attempt: Analysis

Theorem:

The Fisher-Yates algorithm generates a permutation uniformly at random.

Proof:

- Every execution of the algorithm happens with probability $1/n!$.
- Each execution generates a different permutation.
- Hence, the probability that a specific permutation is generated is $1/n!$, for all possible permutations of $\langle 1, 2, \dots, n \rangle$.

Alternatives to Balanced Binary Search Trees

Alternative to AVL trees:

- Treaps
- Skip lists

So, why are we looking at another different way of doing this?

- More simple data structures built in a randomized way
- No need for rebalancing like in AVL trees
- put and get in $O(\log n)$ worst-case time with high probability

Applications:

- Various database systems use it
- Concurrent/parallel computing environments

The Map ADT (recap)

- **get(k)**: if the map M has an entry with key k , return its associated value
- **put(k, v)**: if key k is not in M , then insert (k, v) into the map M ; else, replace the existing value associated to k with v
- **remove(k)**: if the map M has an entry with key k , remove it
- **size(), isEmpty()**
- **entrySet()**: return an iterable collection of the entries in M
- **keySet()**: return an iterable collection of the keys in M
- **values()**: return an iterable collection of the values in M

Treap

Given a collection $\{(v_0, p_0), \dots, (v_{n-1}, p_{n-1})\}$ a Treap is a binary tree T holding these items such that:

1. If we look at the v -values T is binary search tree
2. If we look at the p -values T is heap

Theorem:

Given values $\{v_0, \dots, v_{n-1}\}$ if we pick p_i UAR from $[0, 1]$ then a Treap for $\{(v_0, p_0), \dots, (v_{n-1}, p_{n-1})\}$ has height $O(\log n)$ with high probability

Building a Treap

Theorem:

Given a collection $\{(v_0, p_0), \dots, (v_{n-1}, p_{n-1})\}$ sort by increasing value we build a Treap using a recursive approach in $O(n \text{ height})$ time

```
def build_treap(items):  
  
    n = len(items)  
    i = index in [0:n] minimizing items[i][1]  
    left = build_treap(items[0:i])  
    right = build_treap(items[i+1:n])  
  
    return Node(items[i], left, right)
```

We could reduce the space complexity to $O(n)$ by representing the subproblems implicitly with a pair of indices.

Building a Map

Theorem:

Given values $\{v_0, \dots, v_{n-1}\}$ we can build in $O(n \log n)$ a binary search tree with height $O(\log n)$ (with high probability) using the `build_treap` function by picking priorities at random.

```
def build_map(values):
    sort(values)
    priorities = list of n random reals in [0, 1]
    items = zip(values, priorities)

    return build_treap(items)
```

But we know how to do this already by putting the median element at the root!

Building a Treap incrementally

Theorem:

Given a treap T for $\{(v_0, p_0), \dots, (v_{n-1}, p_{n-1})\}$ we insert a new item in $O(\text{height})$ time

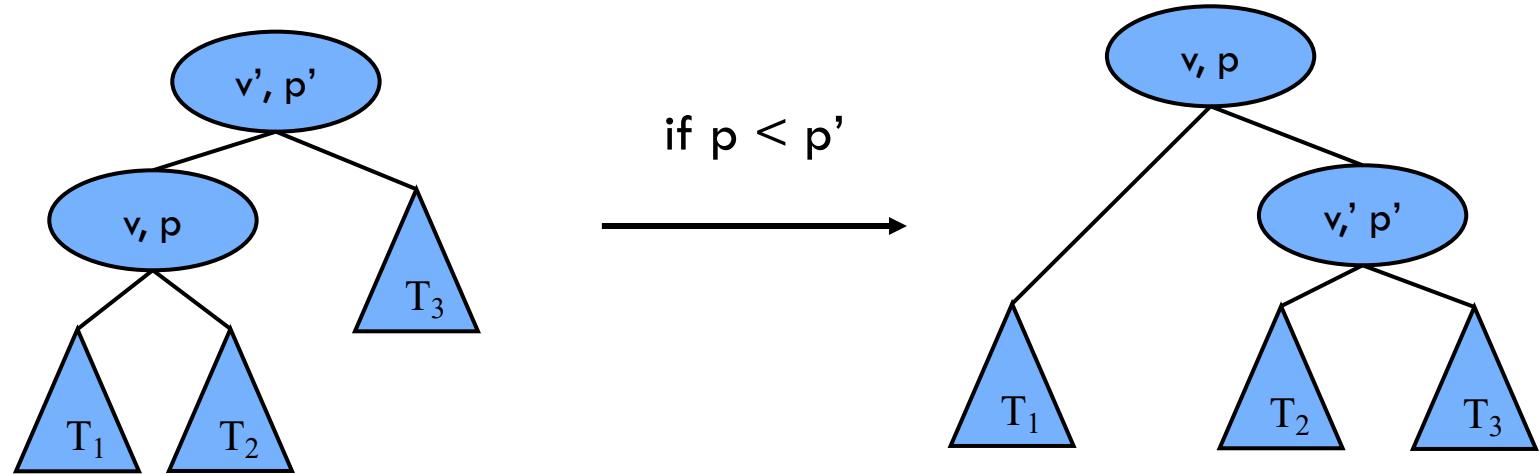
```
def insert_map(T, value):  
  
    priority = random real in [0, 1]  
    T.insert_treap( (value, priority) )
```

Theorem:

We can incrementally build a binary search tree in $O(\log n)$ time per insertion using the `insert_item` function by picking priorities at random.

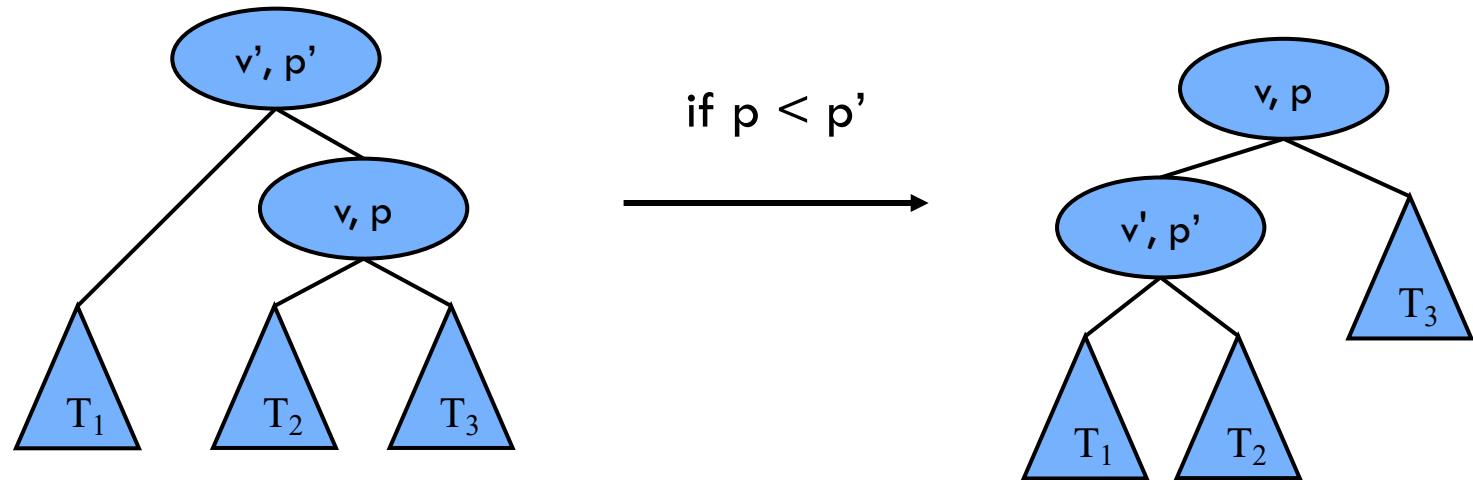
High level idea behind Treap increment

1. insert (v, p) using the standard BST insert
2. use modified bubble-up routine to fix heap property



High level idea behind Treap increment

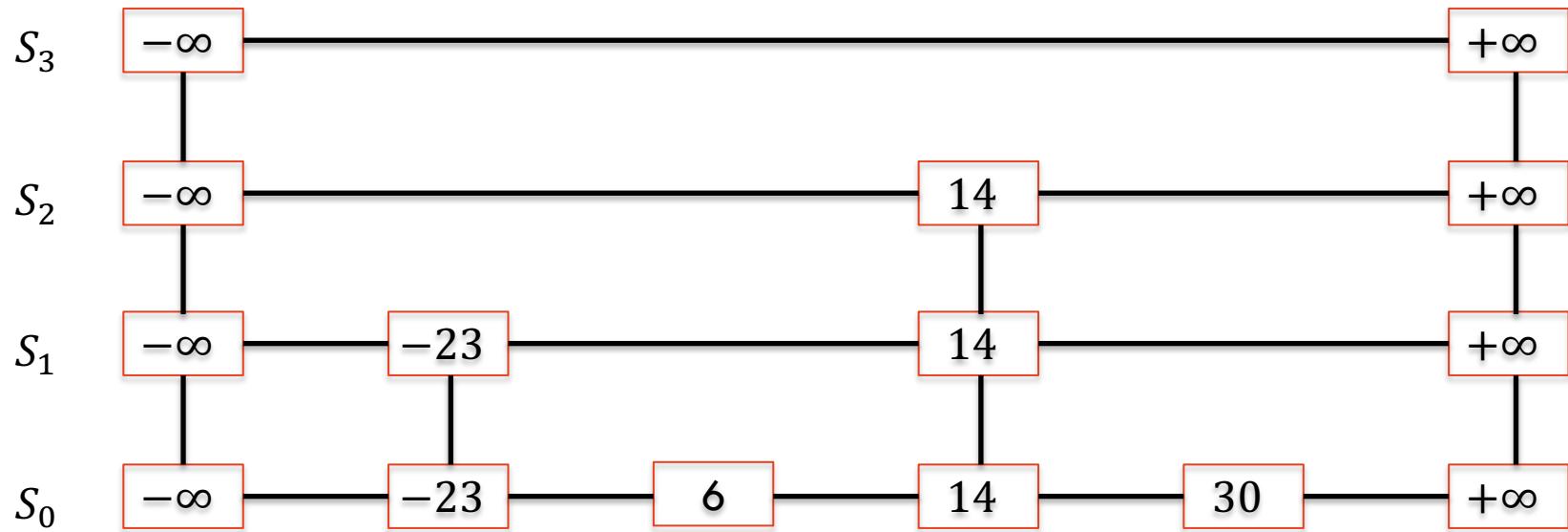
1. insert (v, p) using the standard BST insert
2. use modified bubble-up routine to fix heap property



Skip lists

Leveled structure, where every level is a subset of the one below it.

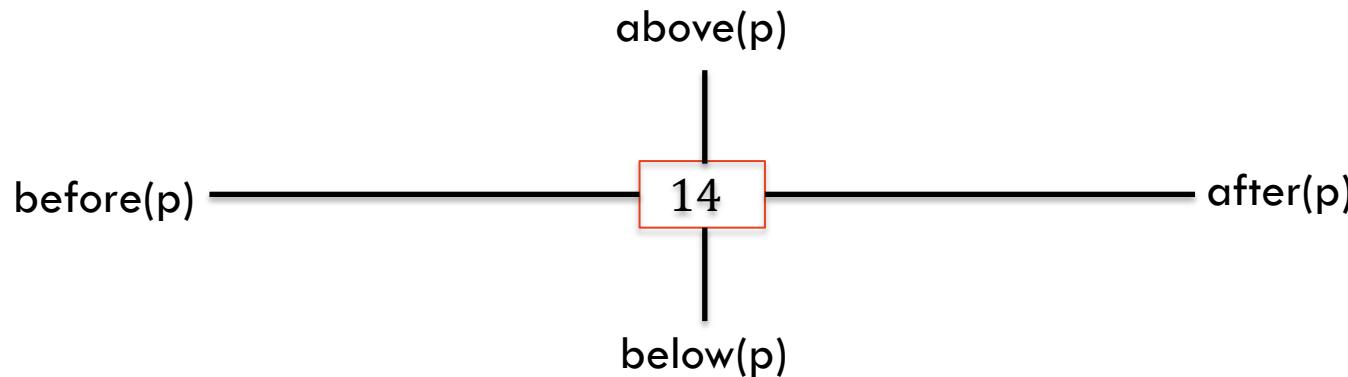
Next level's elements determined by coin flips.



Skip lists

A node p has pointer to:

- $\text{after}(p)$: Node following p on same level.
- $\text{before}(p)$: Node preceding p in the same level.
- $\text{above}(p)$: Node above p in the same tower.
- $\text{below}(p)$: Node below p in the same tower.



Search

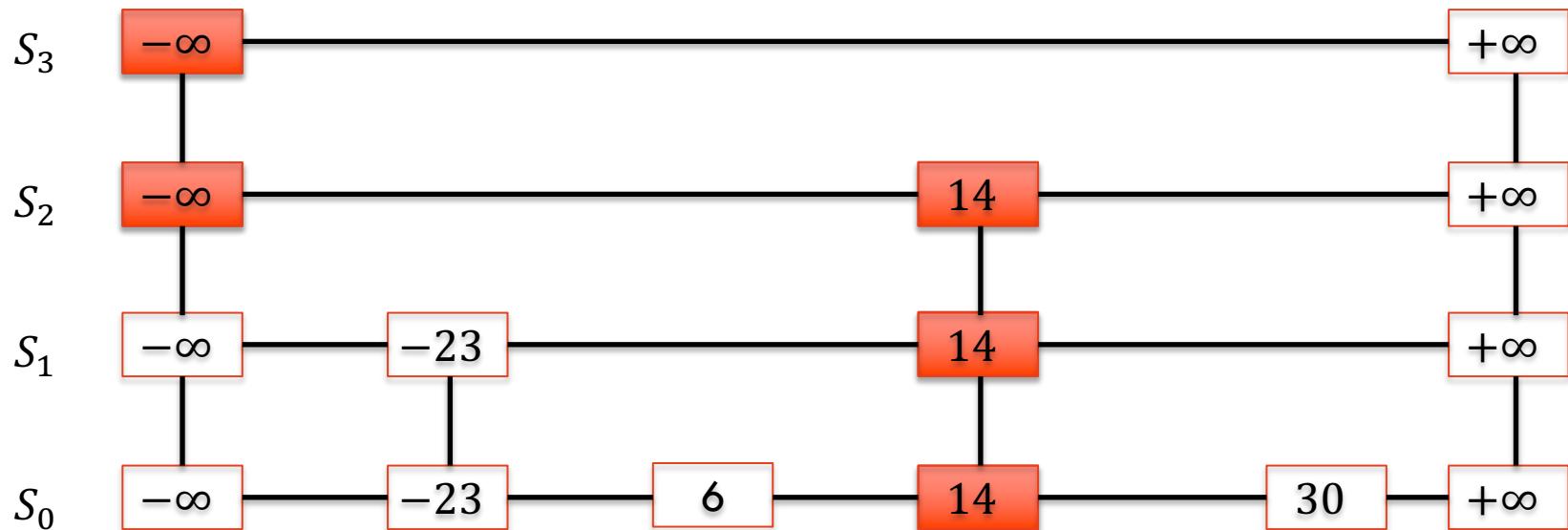
```
def search(p, k):
    while below(p) != null do
        p = below(p)
        while key(after(p)) ≤ k do
            p = after(p)
    return p
```



Example: `search(topleft node, 30)`

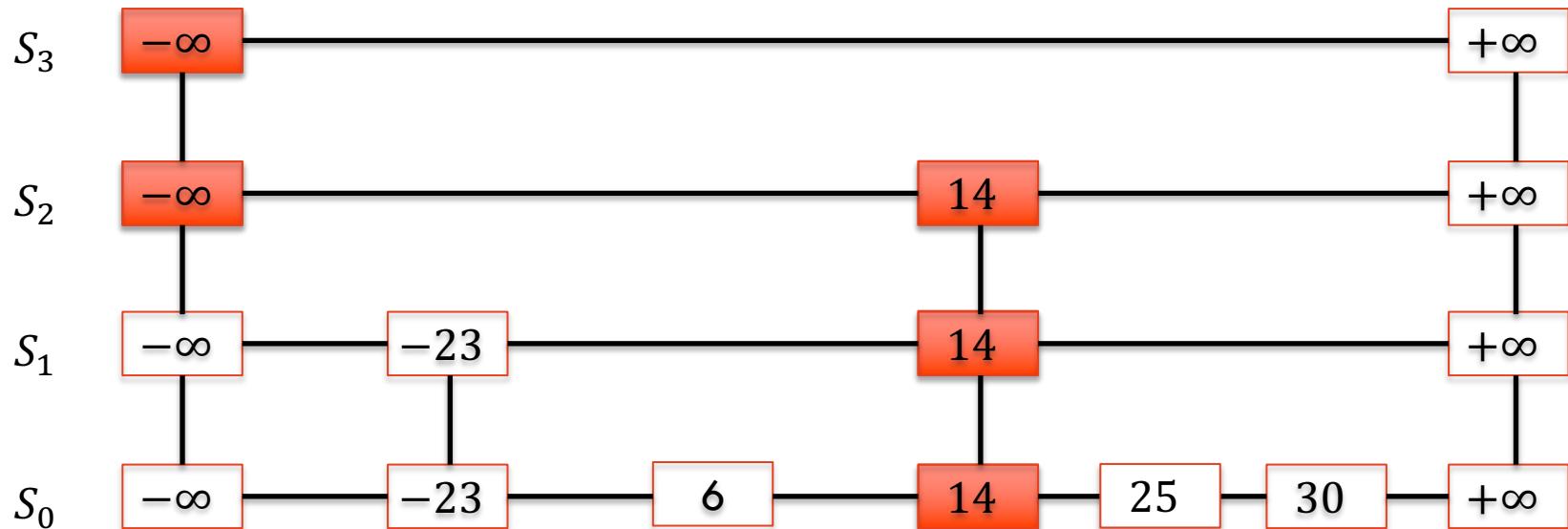
Insertion

```
def insert(p,k):
    p = search(p,k)
    q = insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) == null do
            p = before(p)
        p = above(p)
    q = insertAfterAbove(p,q,k)
```



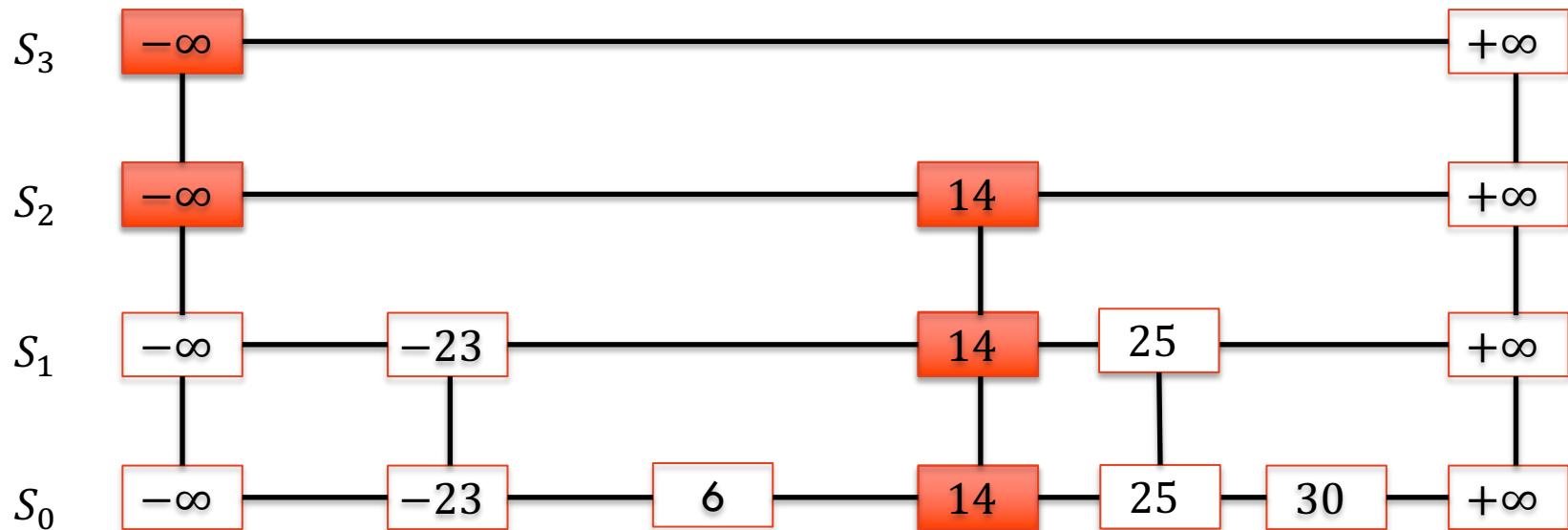
Insertion

```
def insert(p,k):
    p = search(p,k)
    q = insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) == null do
            p = before(p)
        p = above(p)
    q = insertAfterAbove(p,q,k)
```



Insertion

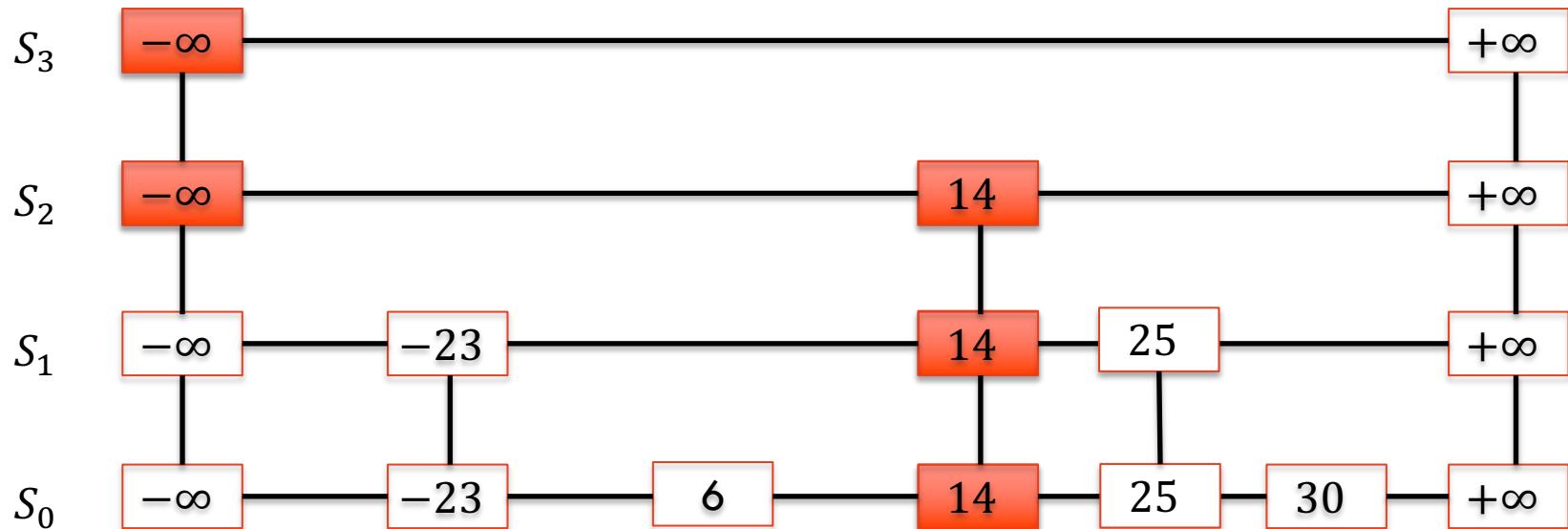
```
def insert(p,k):
    p = search(p,k)
    q = insertAfterAbove(p,null,k)
    while coin flip is heads do
        while above(p) == null do
            p = before(p)
        p = above(p)
    q = insertAfterAbove(p,q,k)
```



Example: `insert(topleft node, 25)`

Removal

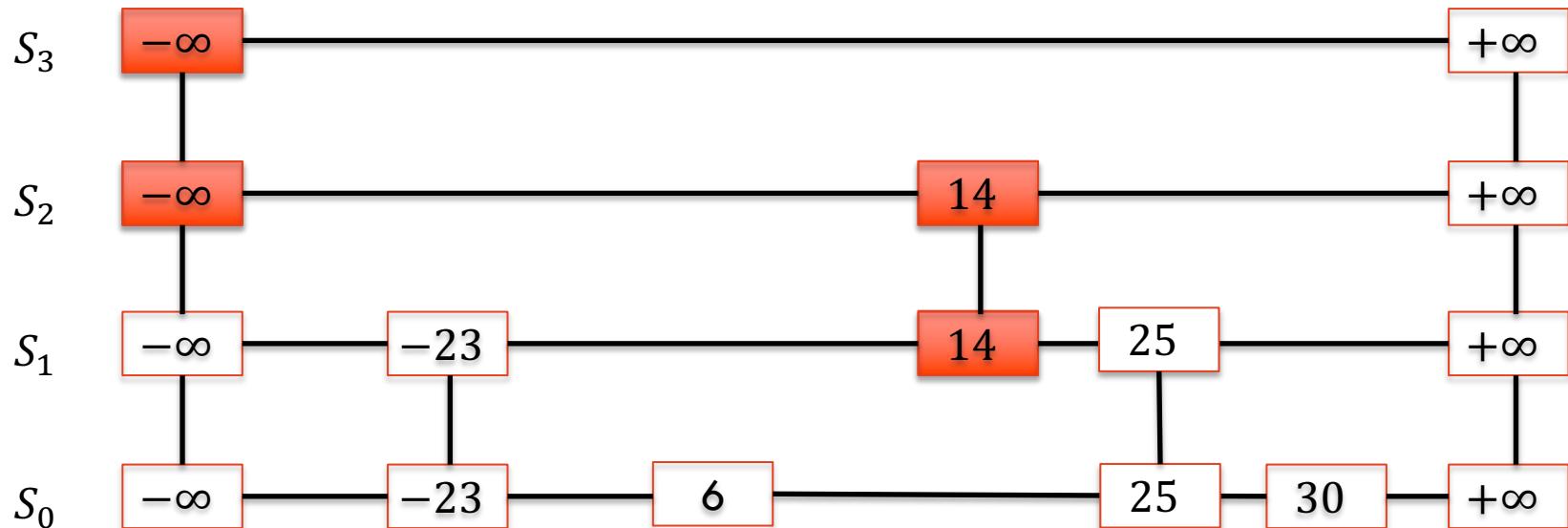
```
def remove(p, k):
    p = search(p, k)
    if key(p) != k then
        return null
    repeat
        remove p
        p = above(p)
    until above(p) == null
```



Example: `remove(topleft node, 14)`

Removal

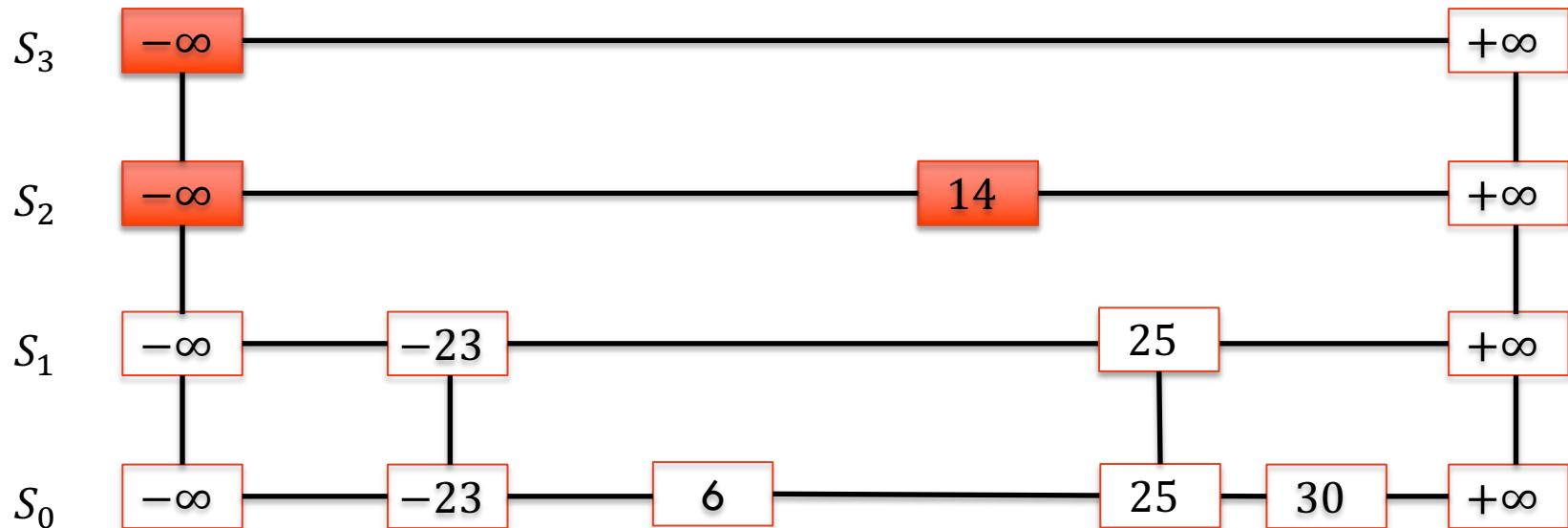
```
def remove(p, k):
    p = search(p, k)
    if key(p) != k then
        return null
    repeat
        remove p
        p = above(p)
    until above(p) == null
```



Example: `remove(topleft node, 14)`

Removal

```
def remove(p, k):
    p = search(p, k)
    if key(p) != k then
        return null
    repeat
        remove p
        p = above(p)
    until above(p) == null
```



Example: `remove(topleft node, 14)`

Removal

```
def remove(p, k):
    p = search(p, k)
    if key(p) != k then
        return null
    repeat
        remove p
        p = above(p)
    until above(p) == null
```



Example: `remove(topleft node, 14)`

Skip lists: Top layer

Keep a pointer to the topleft node.

Choices for the top layer:

- Keep at a fixed level, say $\max\{10, 3[\log(n)]\}$
 - Insertion needs to take this into account
- Variable level
 - Continue insertion until coin comes up tails
 - No modification required
 - Probability that this gives more than $O(\log n)$ levels is very low

Skip lists: Analysis

Theorem:

The expected height of a skip list is $O(\log n)$.

Proof:

- The probability that an element is present at height i is $1/2^i$.
 - I.e., the probability that the coin comes up heads i times.
- The probability that level i has at least one item is at most $n/2^i$.
- The probability that skip list has height h is probability that level h has at least one element.
- So, probability that skip list to have height larger than $c \log n$ is at most

$$\frac{n}{2^{c \log n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

- So, probability that skip list to have height $O(\log n)$ is at least

$$1 - \frac{1}{n^{c-1}}$$

Skip lists: Search Analysis

Theorem:

The expected search time of a skip list is $O(\log n)$.

Proof:

- Searching consists of horizontal and vertical steps.
- There are h vertical steps, so $O(\log n)$ with high probability.
- To have a horizontal step on level i , the next node can't be on level $i+1$.
- The probability of this is $1/2$.
- This means that the expected number of horizontal steps per level is 2.
- So we expect to spend $O(1)$ time per level.
- Expected search time: $O(\log n)$ time with high probability.

Insertion and deletion take expected $O(\log n)$ time using similar analysis.

Skip lists: Space Analysis

Theorem:

The expected space used by a skip list is $O(n)$.

Proof:

- Space per node: $O(1)$
- Expected number of nodes at level i is $n/2^i$.
- Thus expected number of nodes is

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

Skip lists: Summary

Expected space: $O(n)$

Expected search/insert/delete time: $O(\log n)$

Works very well in practice and doesn't require any complicated rebalancing operations.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

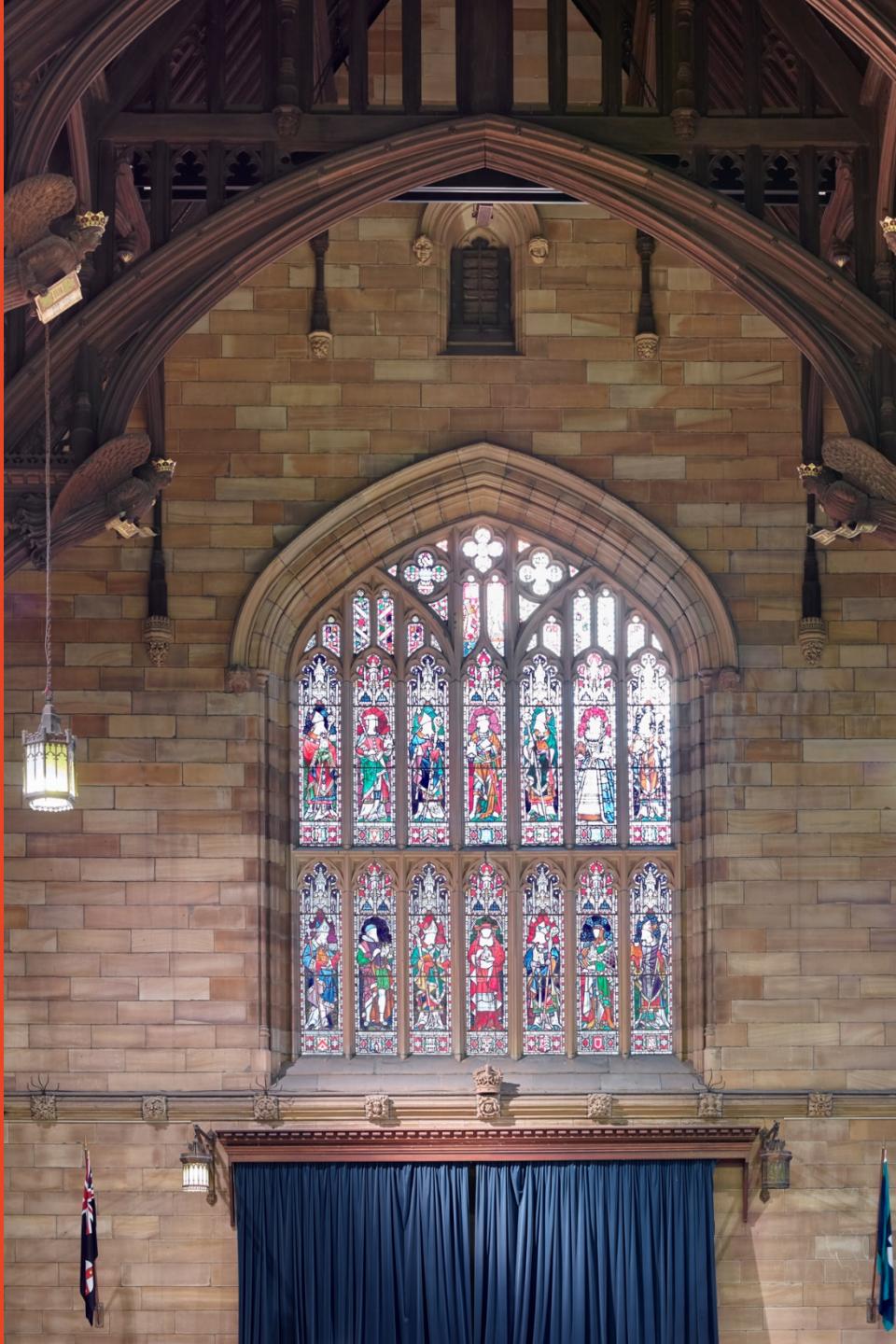
Do not remove this notice.

COMP2823

Lecture 11: Divide and Conquer [GT 3.1 and 8]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



Divide and Conquer

Divide and Conquer algorithms can normally be broken into these three parts:

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur** Recursively solve each part [each sub-problem].
3. **Conquer** Combine the solutions of each part into the overall solution.

Searching Sorted Array

Given A sorted sequence S of n numbers a_0, a_1, \dots, a_{n-1} stored in an array A[0, 1, ..., n - 1].

Problem Given a number x, is x in S?

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Searching: Naïve Approach

Problem Given a number x , is x in S ?

Idea Check every element in turn to see if it is equal to x .

```
for e in S do
    if e equals x then
        return "Yes"
return "No"
```



Found an element equal to x in S

There was no element equal to x in S

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Running Time $O(n)$

Binary Search in sorted A[0 to n-1]

1. If the array is empty, then return “No”
2. Compare x to the middle element, namely $A[\lfloor n/2 \rfloor]$
3. If this middle element is x , then return “Yes”
4. When the middle element is not x : if $A[\lfloor n/2 \rfloor] > x$, then recursively search $A[0:\lfloor n/2 \rfloor]$
5. if $A[\lfloor n/2 \rfloor] < x$, then recursively search $A[\lfloor n/2 \rfloor + 1:n]$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Heads up: pseudocode textbook uses indexing from 1 to n , not 0 to $n-1$

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

A[6]

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22	25	37	39	50	55	80
---	---	---	---	----	----	----	----	----	----	----	----

$$A[6] = 25 > 5 = x$$

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22
---	---	---	---	----	----

A[3]

25	37	39	50	55	80
---------------	----	----	----	----	----

Binary Search

- Example, search for $x=5$

0	2	5	7	12	22
---	---	---	---	----	----

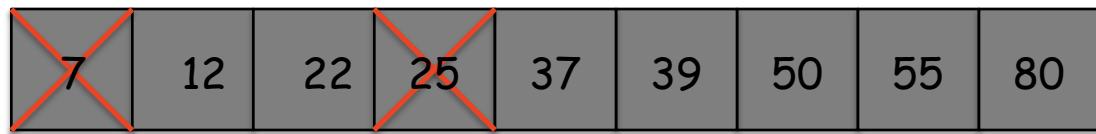
25	37	39	50	55	80
---------------	----	----	----	----	----

$$A[3] = 7 > 5 = x$$

Binary Search

- Example, search for $x=5$

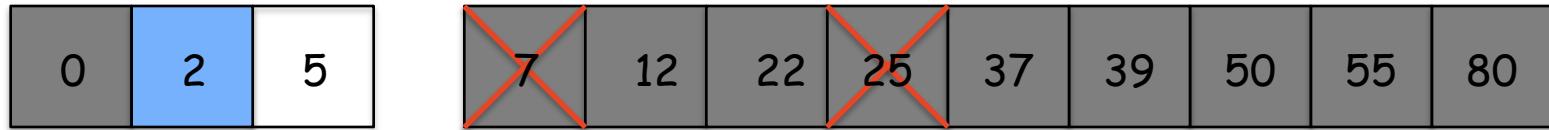
0	2	5
---	---	---



$A[1]$

Binary Search

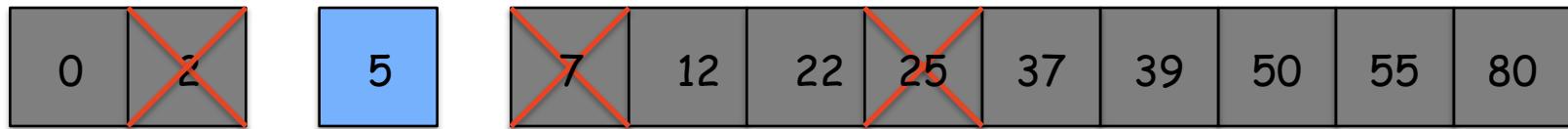
- Example, search for $x=5$



$$A[1] = 2 < 5 = x$$

Binary Search

- Example, search for $x=5$



$A[2]$

Binary search correctness

Invariant: If x is in A before the divide step, then x is in A after the divide step

- if $A[\lfloor n/2 \rfloor] > x$, then x must be in $A[0: \lfloor n/2 \rfloor]$
- if $A[\lfloor n/2 \rfloor] < x$, then x must be in $A[\lfloor n/2 \rfloor + 1:n]$

Every divide step leads to a smaller array.

Thus, if x in A , we will eventually inspect its position due to the invariant and return “Yes”.

Thus, if x in not in A , then eventually we reach the empty array and return “No”.

Recurrence formula

An easy way to analyze the time complexity of a divide-and-conquer algorithm is to define and solve a recurrence

Let $T(n)$ be the running time of the algorithm, we need to find out:

- Divide step cost in terms of n
- Recur step(s) cost in terms of $T(\text{smaller values})$
- Conquer step cost in terms of n

Together with information about the base case, we can set up a recurrence for $T(n)$ and then solve it.

Binary search on an array complexity analysis

Divide step (find middle and compare to x) takes $O(1)$

Recur step (solve left or right subproblem) takes $T(n/2)$

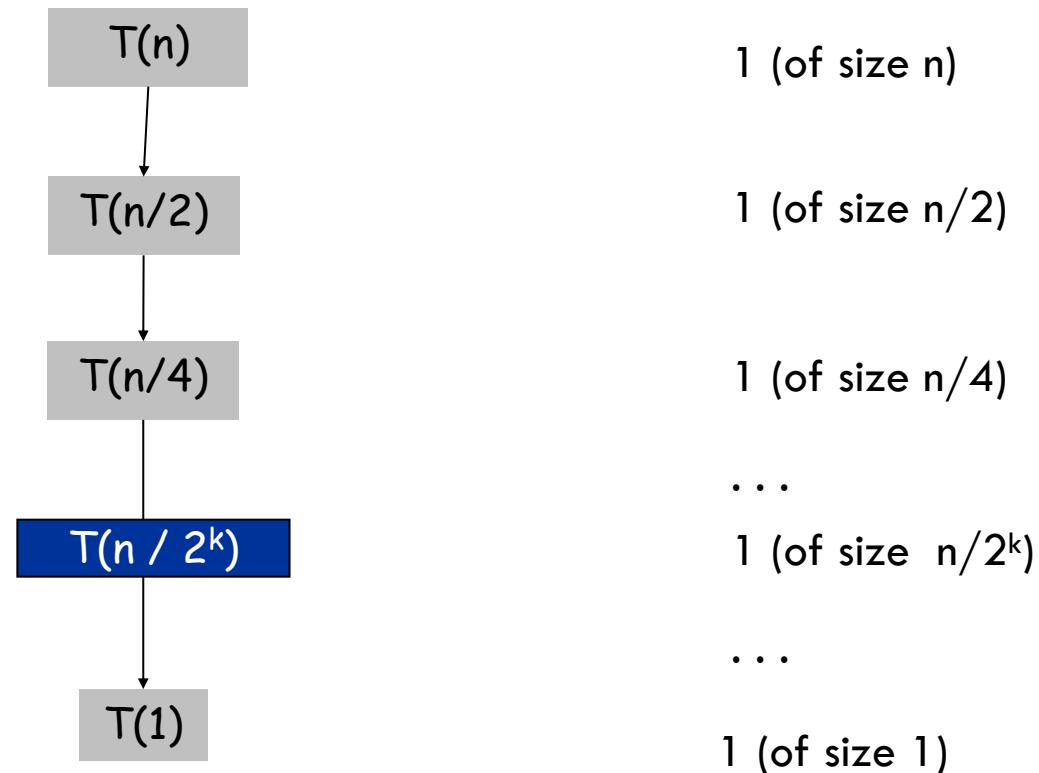
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

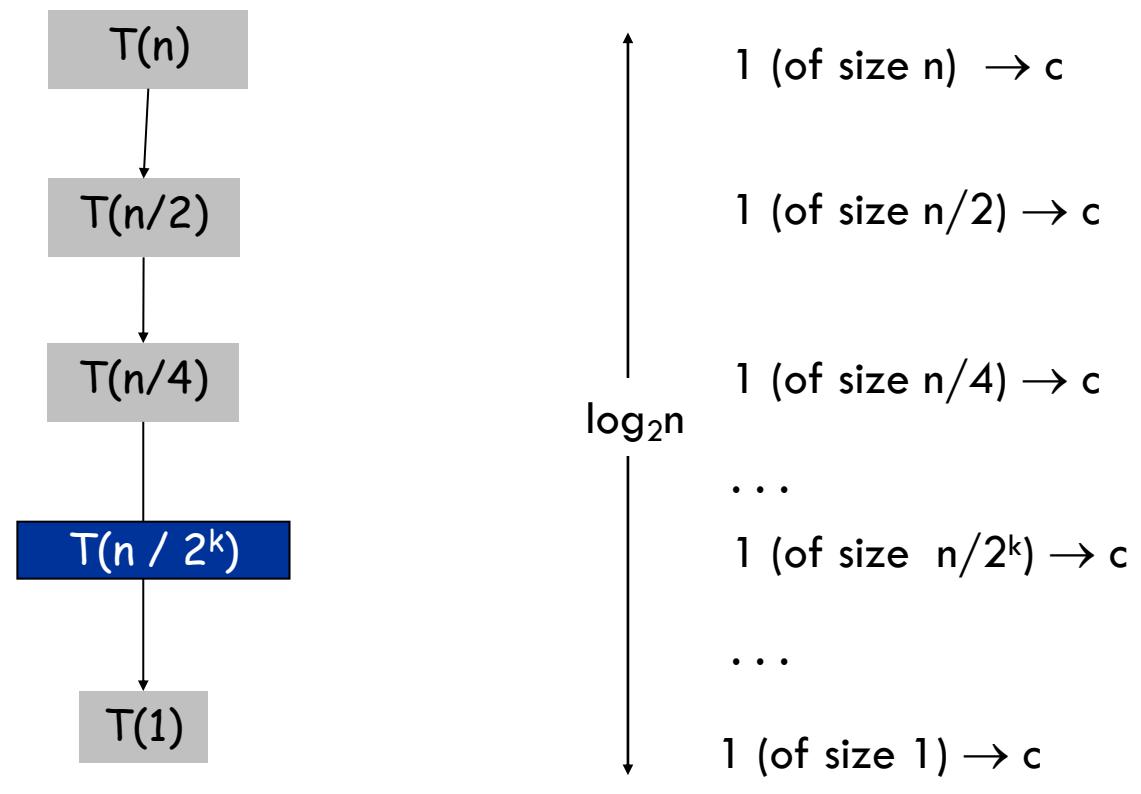
$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(\log n)$, since we can only halve the input $O(\log n)$ times before reaching a base case

Proof by unrolling



Proof by unrolling



Binary search on a linked list complexity analysis

Divide step (find middle and compare to x) takes $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

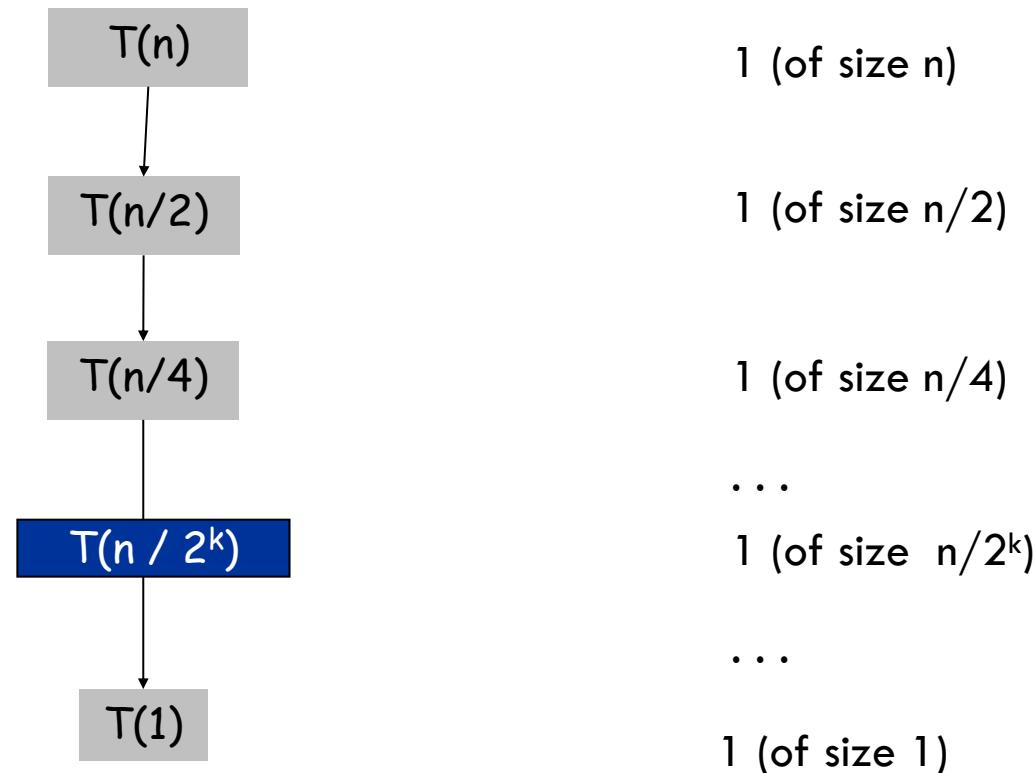
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

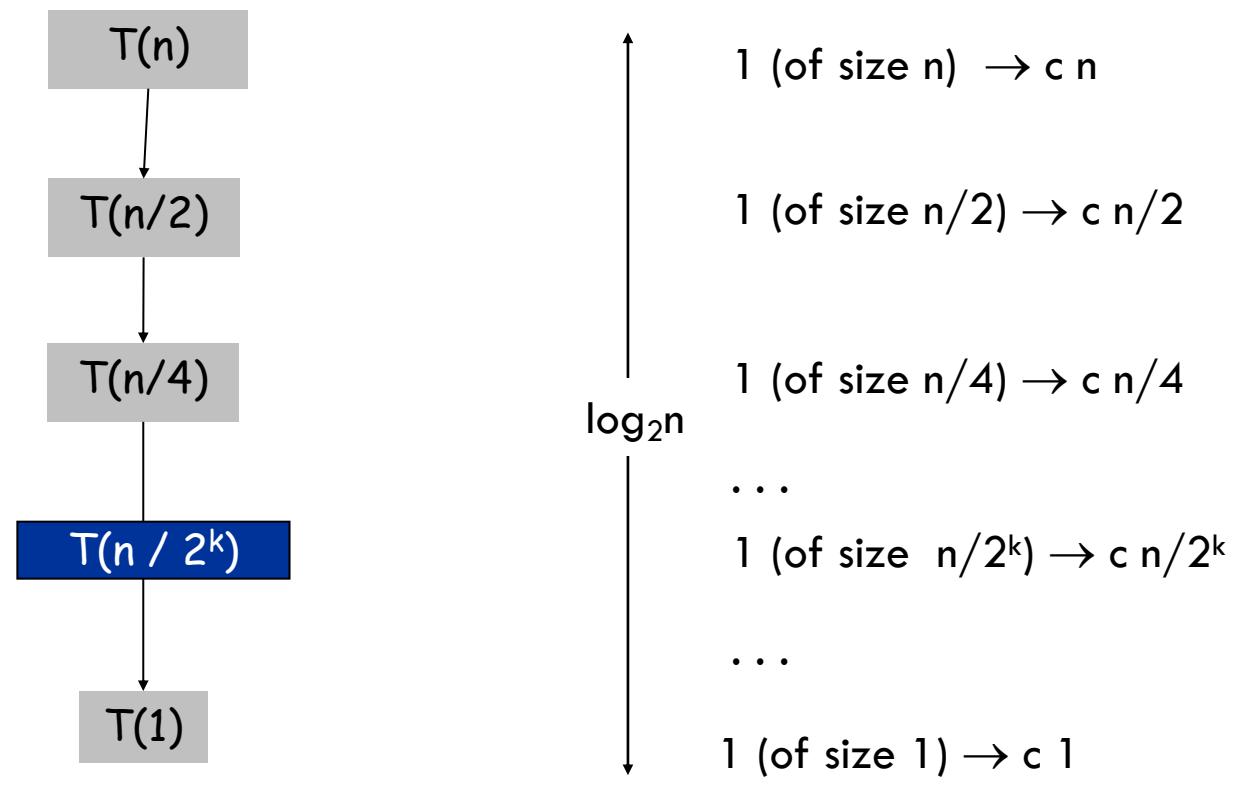
$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n)$, since to access the next index we end up with $n/2 + n/4 + n/8 + \dots$

Proof by unrolling



Proof by unrolling



Merge-Sort

1. **Divide** the array into two halves.
2. **Recur** recursively sort each half.
3. **Conquer** two sorted halves to make a single sorted array.

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

Divide

1	5	12	16	19	6	7	13	20	23
---	---	----	----	----	---	---	----	----	----

Recur

1	5	6	7	12	13	16	19	20	23
---	---	---	---	----	----	----	----	----	----

Conquer

Merge-Sort pseudocode

```
def merge_sort(S):
    # base case
    if |S| < 2 then
        return S

    # divide
    mid = |S|/2
    left = S[:mid]      # doesn't include S[mid]
    right = S[mid:]     # includes S[mid]

    # recur
    sorted_left ← merge_sort(left)
    sorted_right ← merge_sort(right)

    # conquer
    return merge(sorted_left, sorted_right)
```

How?

Merge

Input Two sorted lists.

Output A new merged sorted list.

To merge, we use:

- $O(n)$ comparisons.
- An array to store our results.



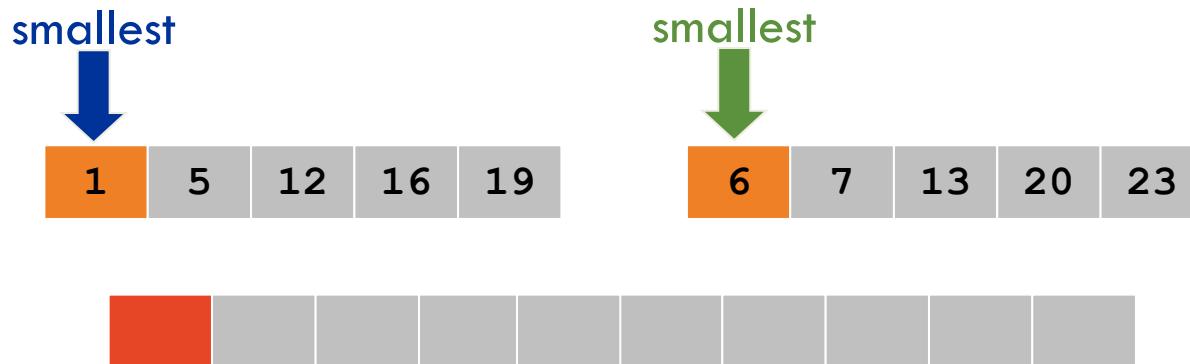
Result:



Merge

Merge Algorithm

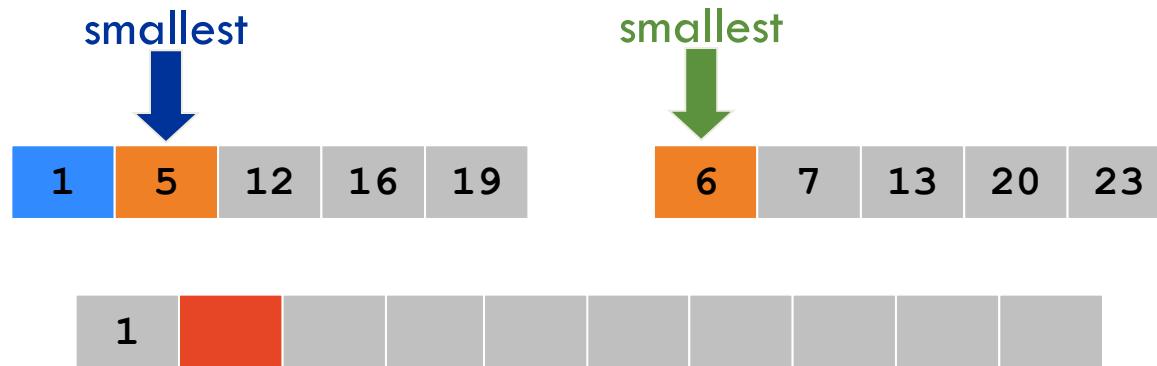
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

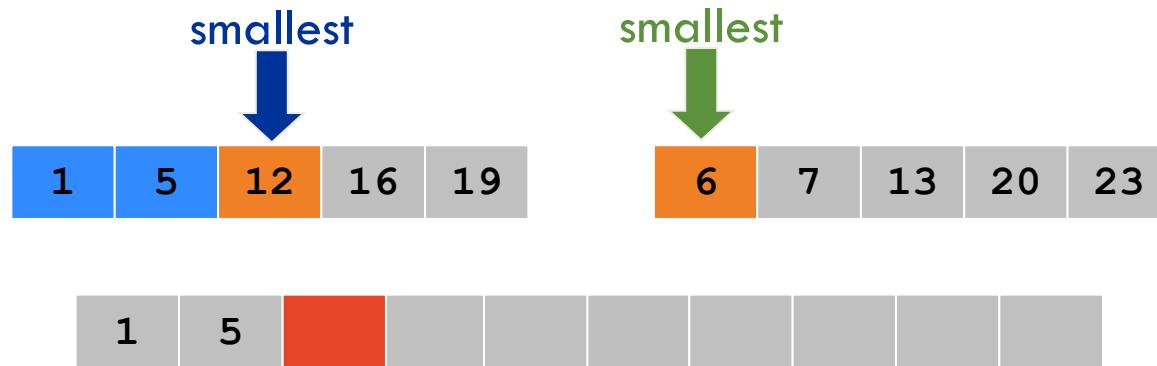
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

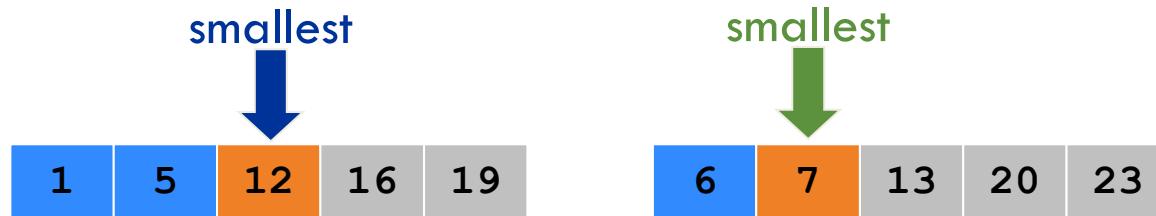
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



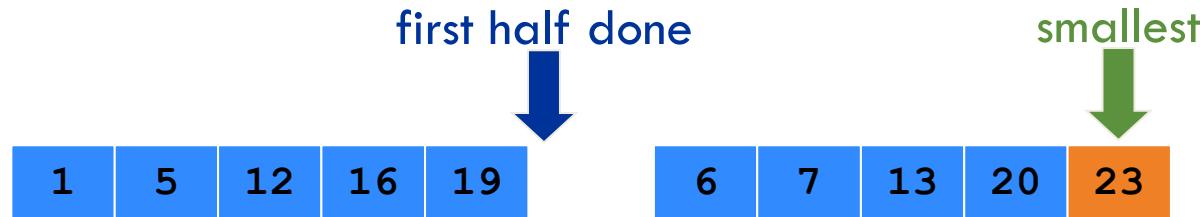
Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Result:



Merge: Implementation

```
def merge(L, R):
    result = array of length (|L| + |R|)
    l, r = 0, 0
    while l + r < |result| do
        index = l + r
        if r ≥ |R| or (l < |L| and L[l] < R[r]) then
            result[index] = L[l]
            l = l + 1
        else
            result[index] = R[r]
            r = r + 1
    return result
```

Merge-Sort

1. **Divide** array into two halves.
2. **Conquer** Recursively sort each half.
3. **Merge** Merge two sorted halves to make a sorted whole.

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

1	12	5	16	19	7	23	6	13	20
---	----	---	----	----	---	----	---	----	----

divide

1	5	12	16	19	6	7	13	20	23
---	---	----	----	----	---	---	----	----	----

recur

1	5	6	7	12	13	16	19	20	23
---	---	---	---	----	----	----	----	----	----

conquer

Merge sort complexity analysis

Divide step (find middle and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $2 T(n/2)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

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

This solves to $T(n) = O(n \log n)$

Solving recurrences by unrolling

General strategy:

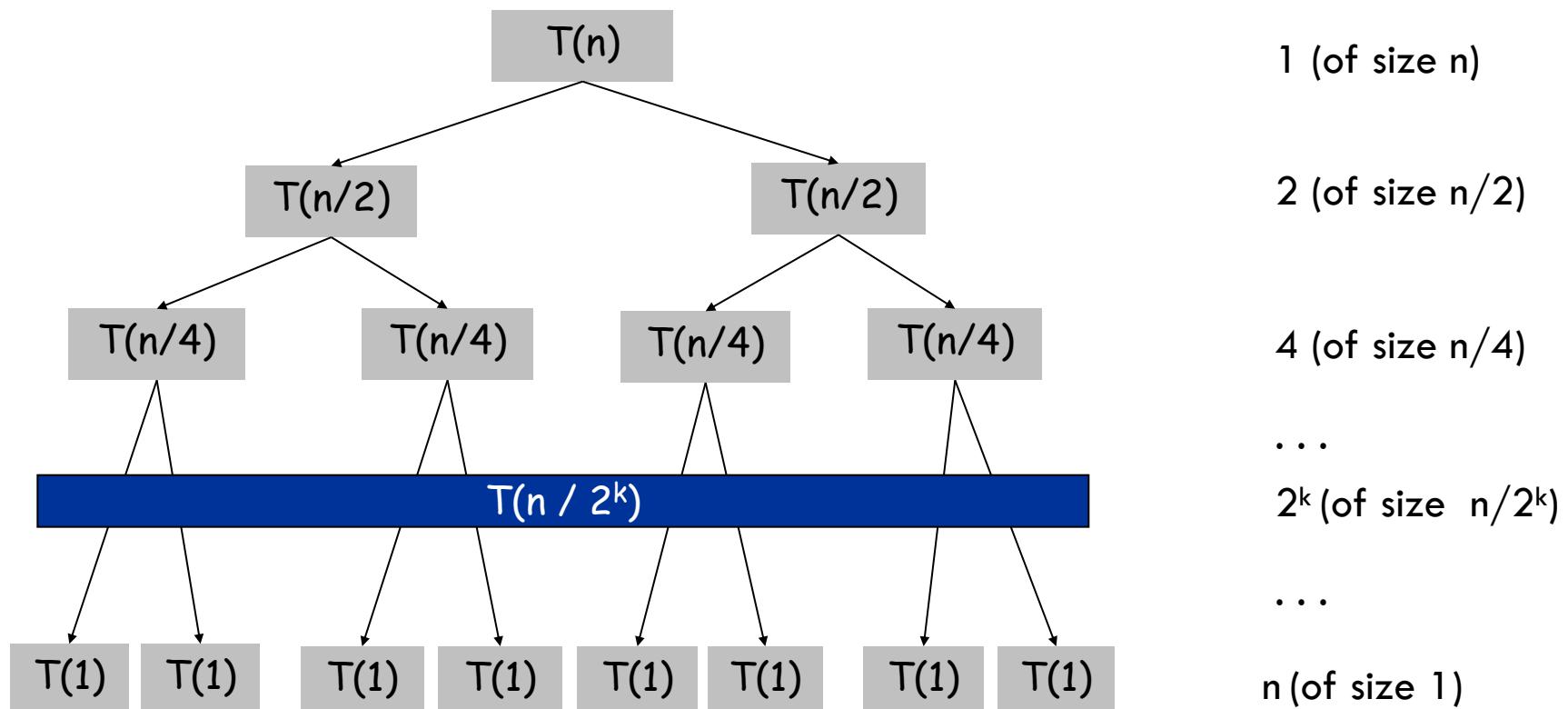
- Analyze first few levels
- Identify the pattern for a generic level
- Sum up over all levels

To verify the solution, we can substitute guess into the recurrence and prove it formally using induction if needed

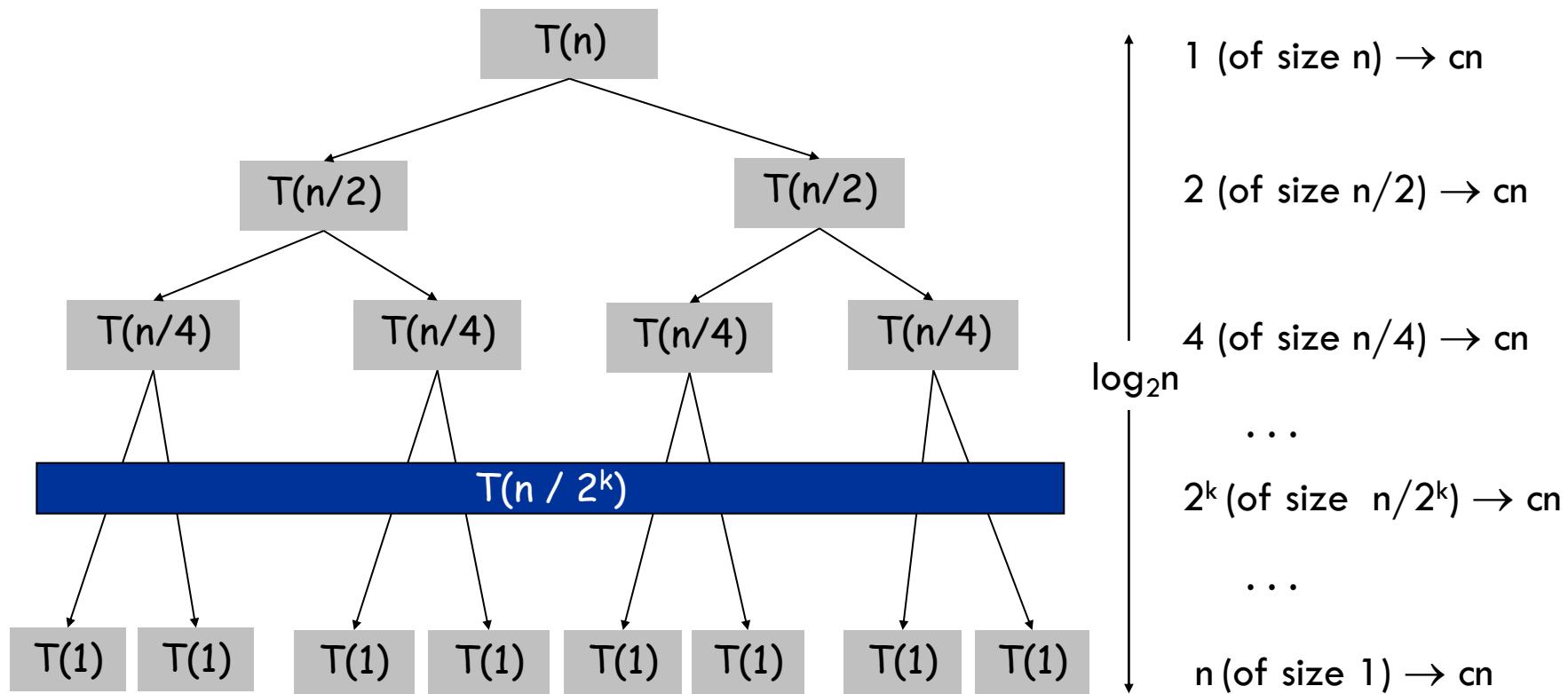
For Merge sort this method yields $T(n) = O(n \log n)$

There is a “Master theorem” (see textbook) that can handle most recurrences of interest, but unrolling is enough for our purposes

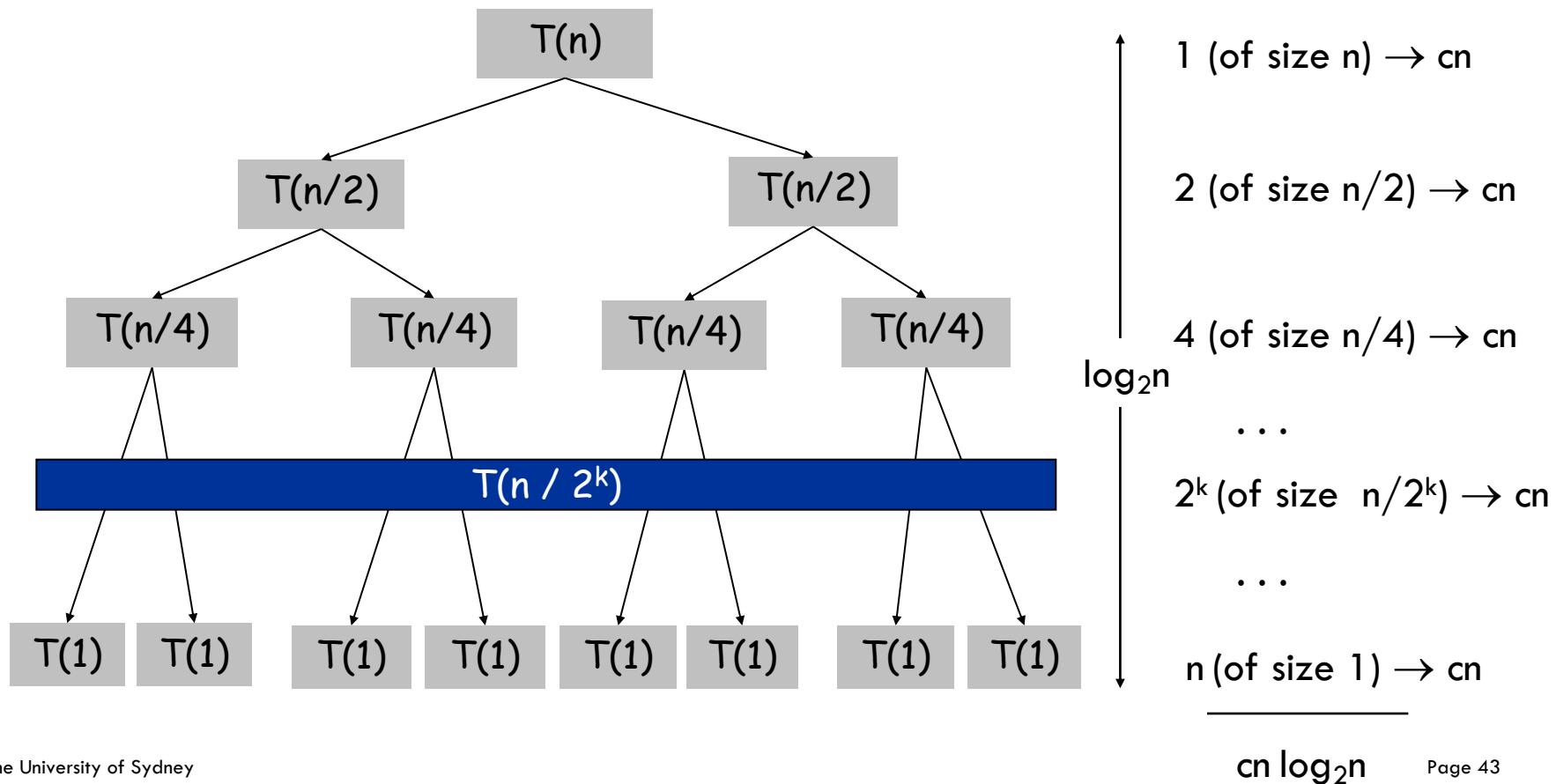
Proof by unrolling



Proof by unrolling



Proof by unrolling



Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2 T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2 T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2 T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

What if n is not even?

Technically speaking we should be solving

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

If $n=2^k$, we would get the neat $T(n) = 2T(n/2) + O(n)$. But this is not always possible for more complicated recurrences.

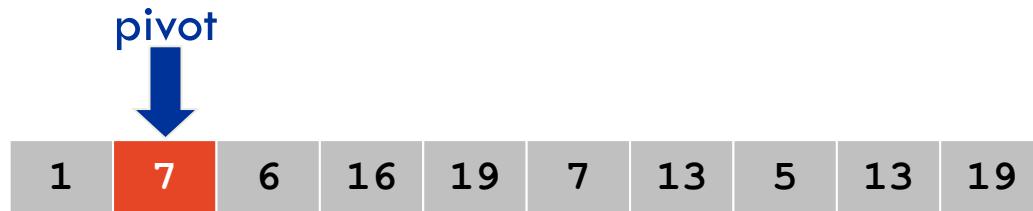
For those cases, if we want to be formal, we have two options:

1. Solve the simpler recurrence to get a guess of a solution and then prove the solution by induction on the real recurrence
2. Define an auxiliary recurrence $S(n) = T(n + c)$ for some constant c . Show that $S(n) \leq 2 S(n) + O(n)$

But you don't need to worry about all that in your proofs here.

Quick sort

1. **Divide** Choose a random element from the list as the **pivot**
Partition the elements into 3 lists:
(i) less than, (ii) equal to and (iii) greater than the **pivot**
2. **Recur** Recursively sort the **less than** and **greater than** lists
3. **Conquer** Join the sorted 3 lists together



Quick sort complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n_L) + T(n_R)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n \log n)$

Quick sort analysis

✓ $T(n) = E \left[\begin{array}{l} \# \text{ of comparisons by QS} \\ \text{on array of size } n \end{array} \right]$

$$\underline{T(n)} = n + \sum_{j=1}^{\frac{n}{2}} \frac{1}{n} [T(j-1) + T(n-j)]$$

✓ $T(n) = n + \sum_{j=1}^{\frac{n}{2}} \frac{2}{n} T(j)$

✓ $n T(n) = n^2 + 2 \sum_{j=1}^{n-1} T(j)$

$$\checkmark n T(n) = n^2 + 2 \sum_{j=1}^{n-1} T(j)$$

$$\checkmark - (n-1) T(n-1) = (n-1)^2 + 2 \sum_{j=1}^{n-2} T(j)$$

$$\checkmark \underline{n T(n)} - \underline{(n-1) T(n-1)} = \underline{2n-1} + \underline{2 T(n-1)}$$

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

$$\frac{T(n)}{n+1} = \frac{2n-1}{n(n+1)} + \frac{T(n-1)}{n}$$

$$\checkmark \left| \frac{T(n)}{n+1} < \frac{2}{n+1} + \frac{T(n-1)}{n} \right|$$

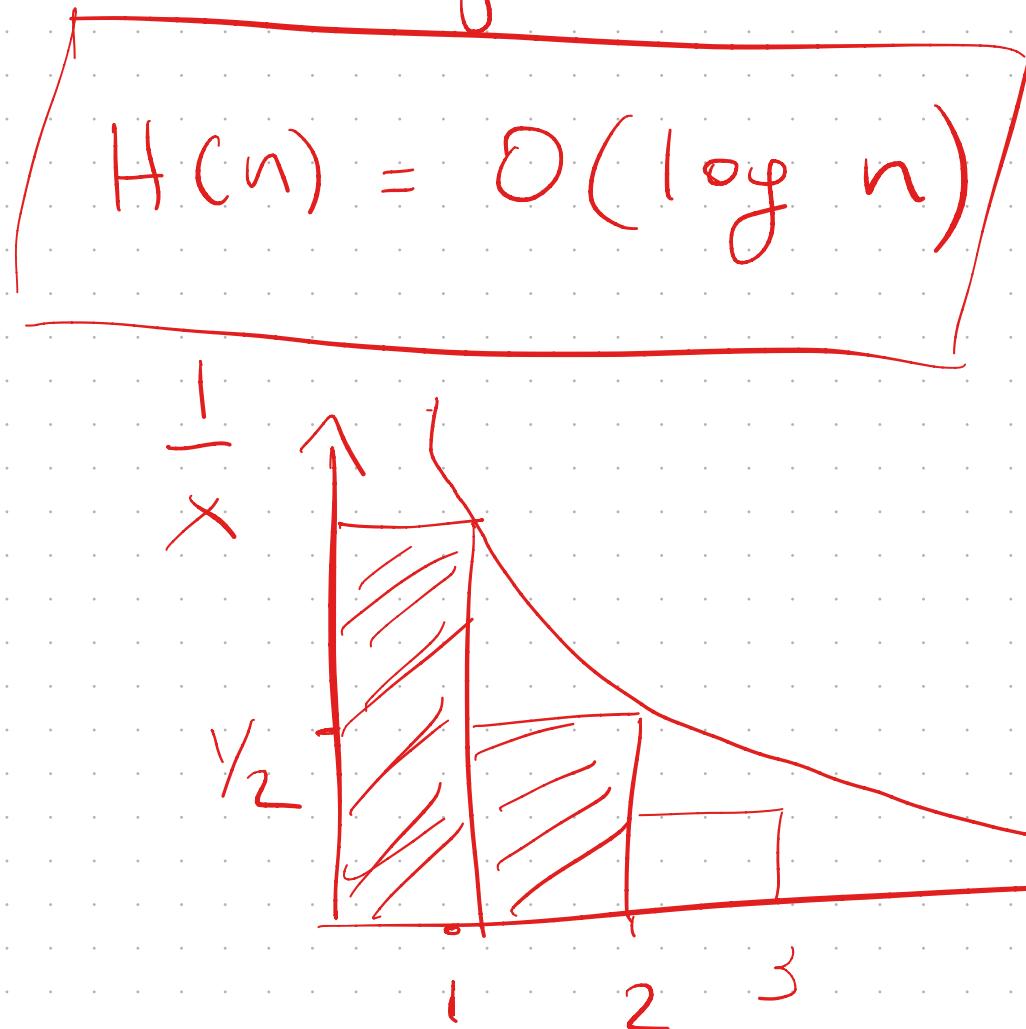
$$\begin{aligned}
 \frac{T(n)}{n+1} &\leq \frac{2}{n+1} + \frac{T(n-1)}{n} \\
 &< \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \dots + \frac{2}{2}
 \end{aligned}$$

$$\Rightarrow T(n) \leq 2 \underbrace{(n+1)}_{\text{H}(n+1)} \left[\frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n} \right]$$

$$T(n) = O(n \log n)$$

$$H(n) = \sum_{j=1}^n \frac{1}{j}$$

Harmonic
number



$$\int_1^{n+1} \frac{1}{x} dx = \ln x \Big|_1^{n+1}$$

Interlude: Comparison sorting lower bound

So far we've seen many sorting algorithms. Some run in $O(n^2)$ time while others run in $O(n \log n)$ time.

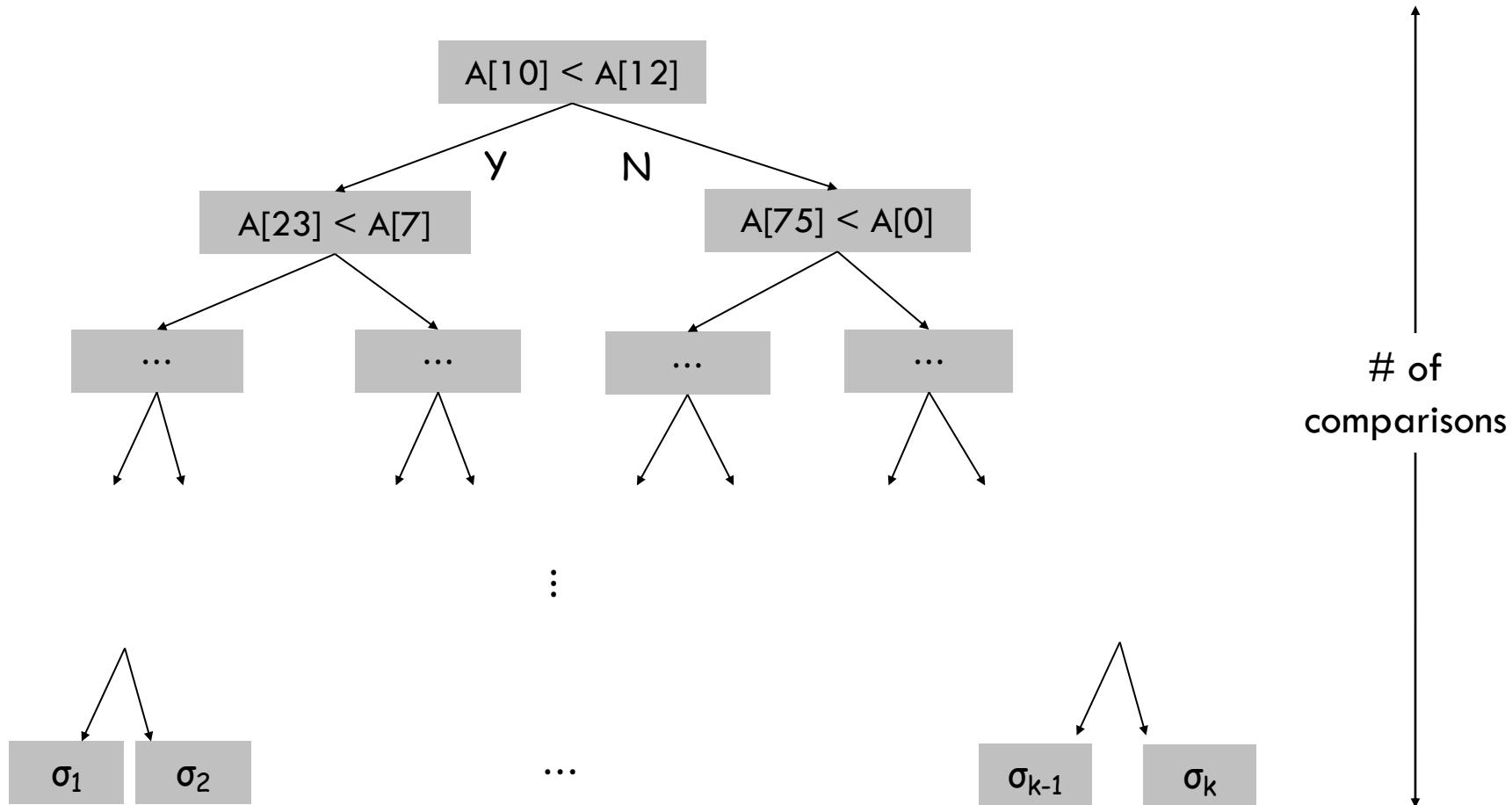
These algorithms work by performing pair-wise comparisons between elements of the sequence we are trying to sort

Such algorithms can be viewed as a decision tree where:

- each internal node compares two indices of the input array
- each external node corresponds to a permutation of $\{1, \dots, n\}$

The height of the decision tree is a lower bound on the running time of the algorithm, since it only counts number of comparisons

Decision tree



The output of a leaf is $A[\sigma(1)], A[\sigma(2)], \dots, A[\sigma(n)]$

Interlude: Comparison sorting lower bound

Fact: Comparison-based sorting algorithms take $\Omega(n \log n)$ time

Proof:

The decision tree associated with a comparison-based sorting algorithm is binary and has at least $n!$ external nodes. Thus the height is $\log n!$ which is $\Omega(n \log n)$

$$\begin{aligned}\log n! &= \log (n * (n-1) * \dots * 1) \\&= \log n + \log(n-1) + \dots + \log 1 \\&> n/2 * (\log n/2) \\&= n/2 * (\log n - 1) \\&= \Omega(n \log n)\end{aligned}$$

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

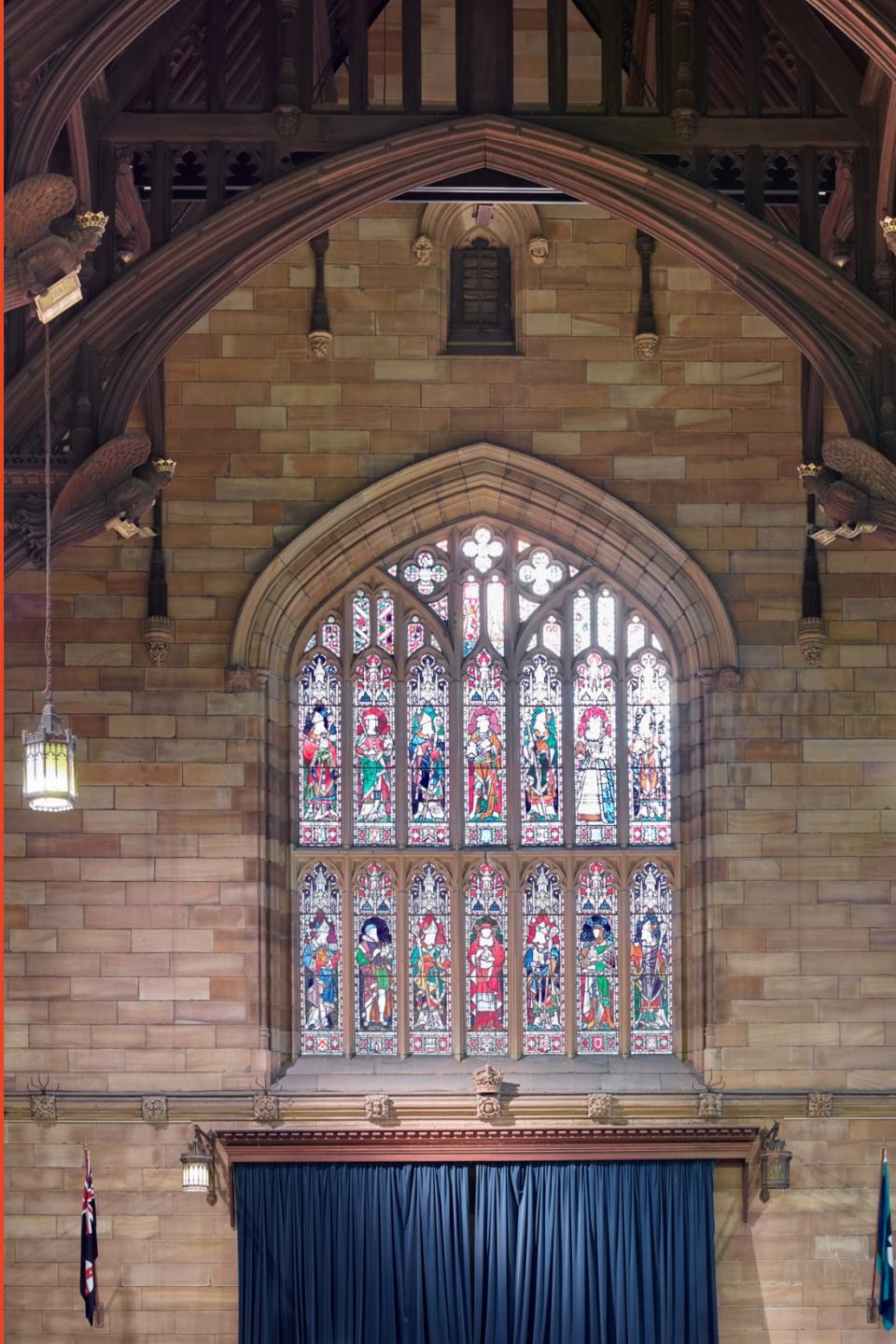
Do not remove this notice.

COMP2823

Lecture 12: Divide and Conquer [GT 11 and 9]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



Divide and Conquer

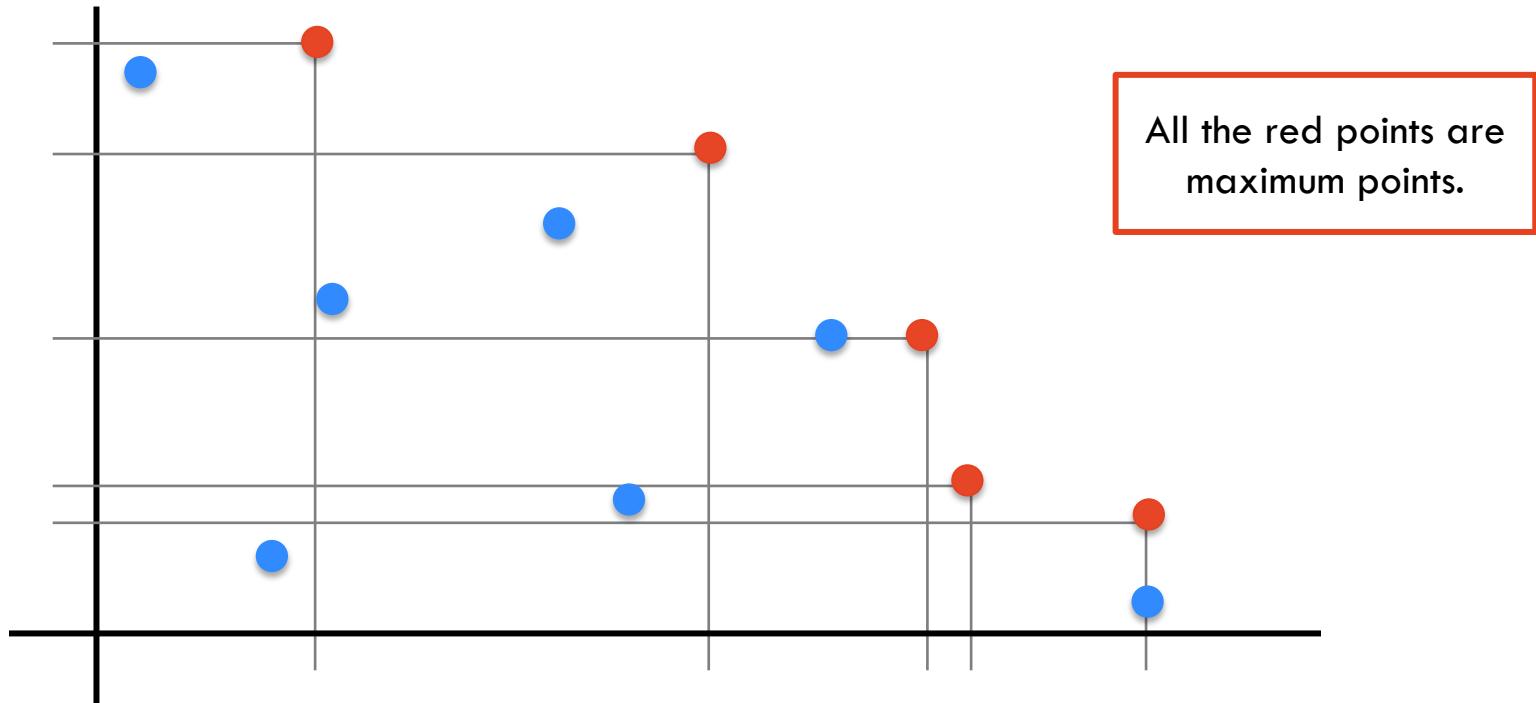
Divide and Conquer algorithms can normally be broken into these three parts:

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur** Recursively solve each part [each sub-problem].
3. **Conquer** Combine the solutions of each part into the overall solution.

Maxima-Set (Pareto frontier)

Definition A point is maximum in a set if all other points in the set have either a smaller x- or smaller y-coordinate.

Problem Given a set S of n distinct points in the plane (2D), find the set of all maximum points.



Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S .

To check if point p is a maximum point in S :

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```



There is a point q that dominates p

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S.

To check if point p is a maximum point in S:

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

Naïve algorithm to find the maxima-set of S:

```
maximaSet = empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S .

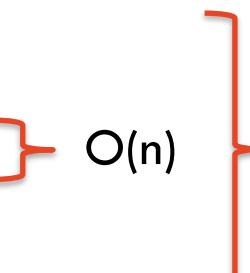
To check if point p is a maximum point in S :

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```



Naïve algorithm to find the maxima-set of S :

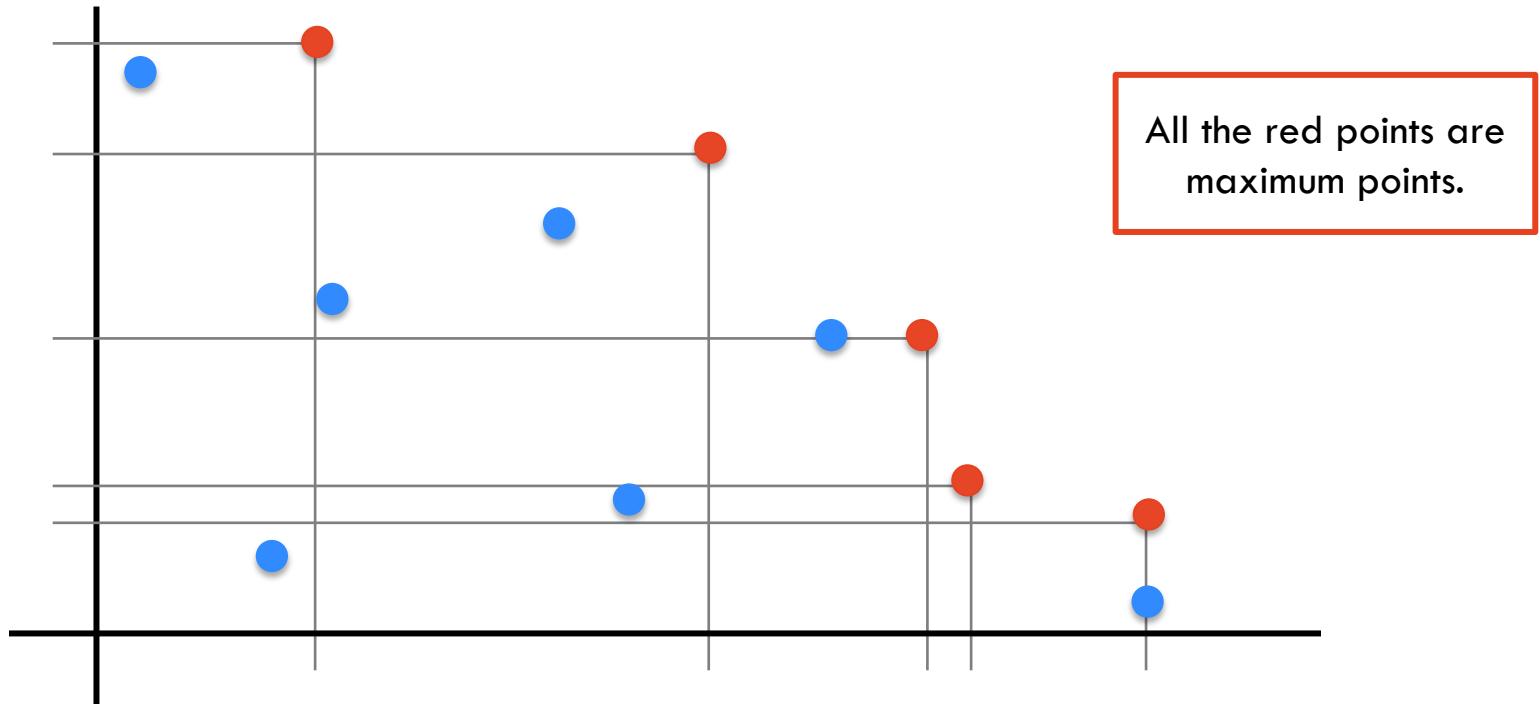
```
maximaSet = empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```



Maxima-Set

Definition A point is maximum in a set if all other points in the set have either a smaller x or smaller y coordinate.

Problem Given a set S of n distinct points in the plane (2D), find the set of all maximum points.



Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.

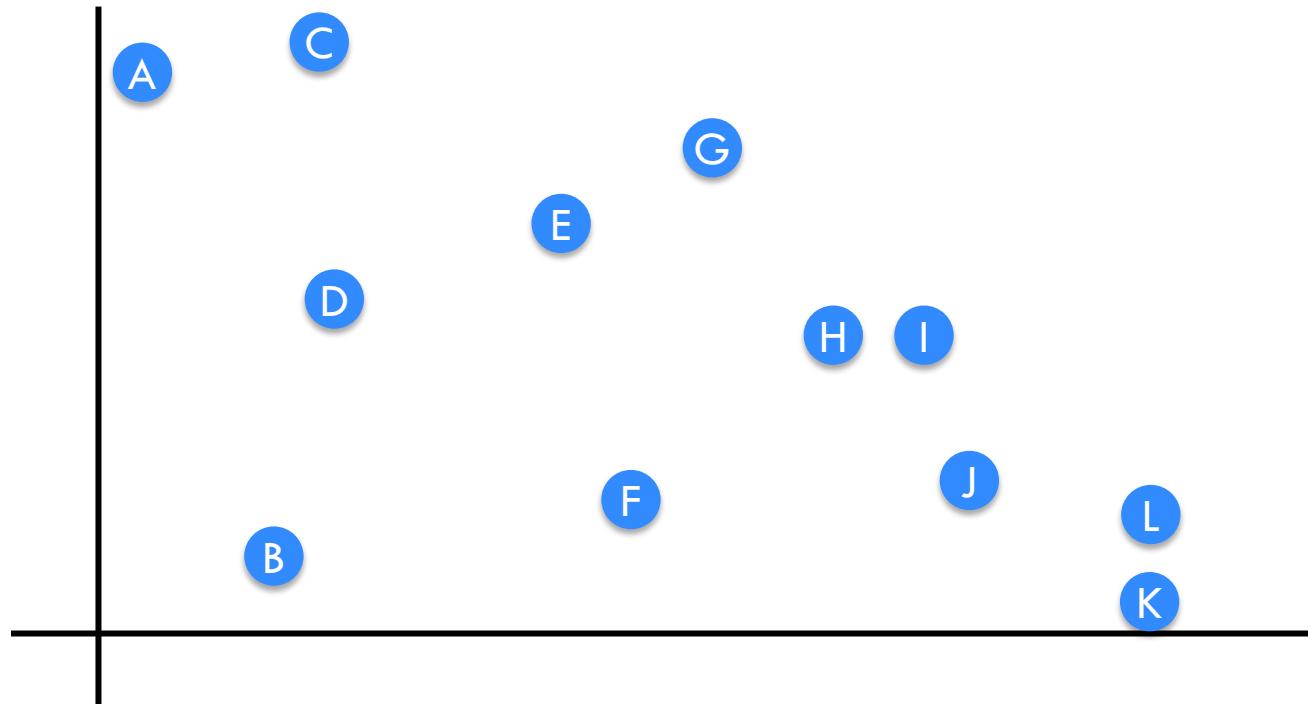
Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute the MS of the union of Left and Right MS

Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

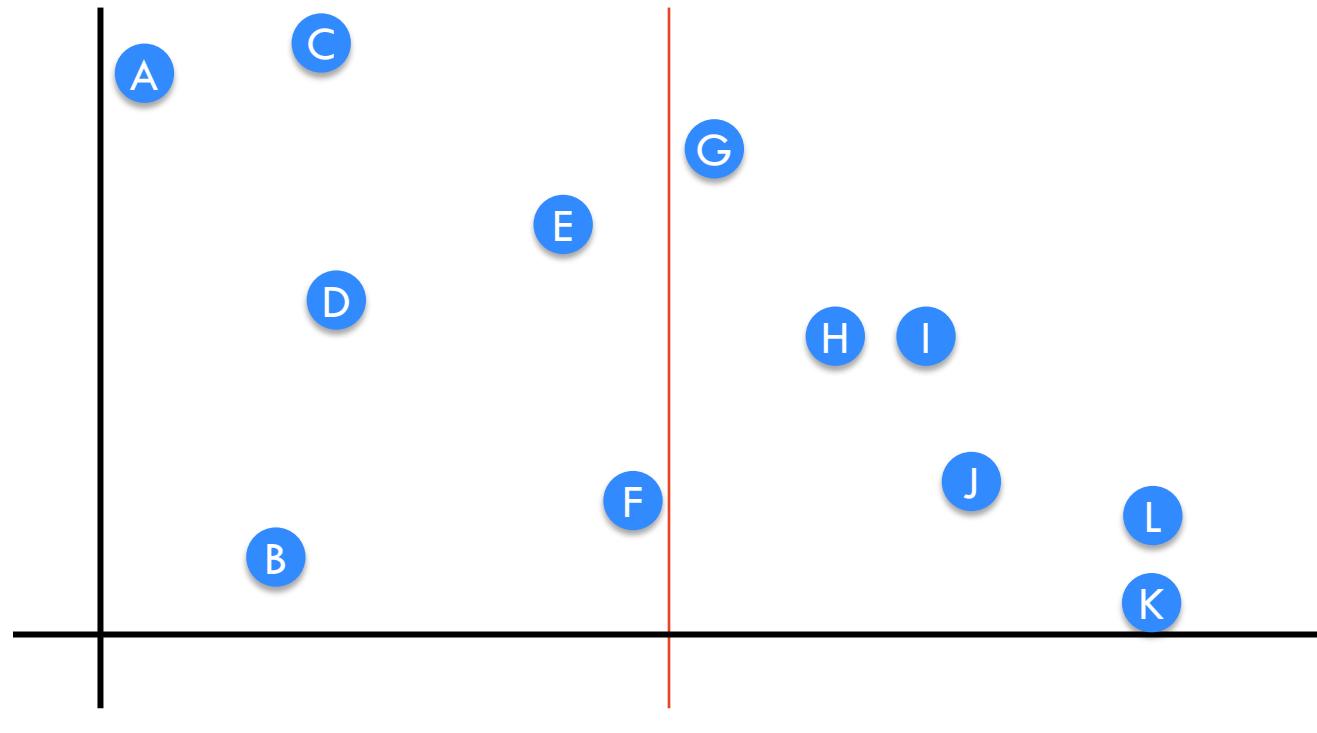


Sorted Points

A	B	C	D	E	F	G	H	I	J	K	L
---	---	---	---	---	---	---	---	---	---	---	---

Maxima-Set

Divide array into two halves.

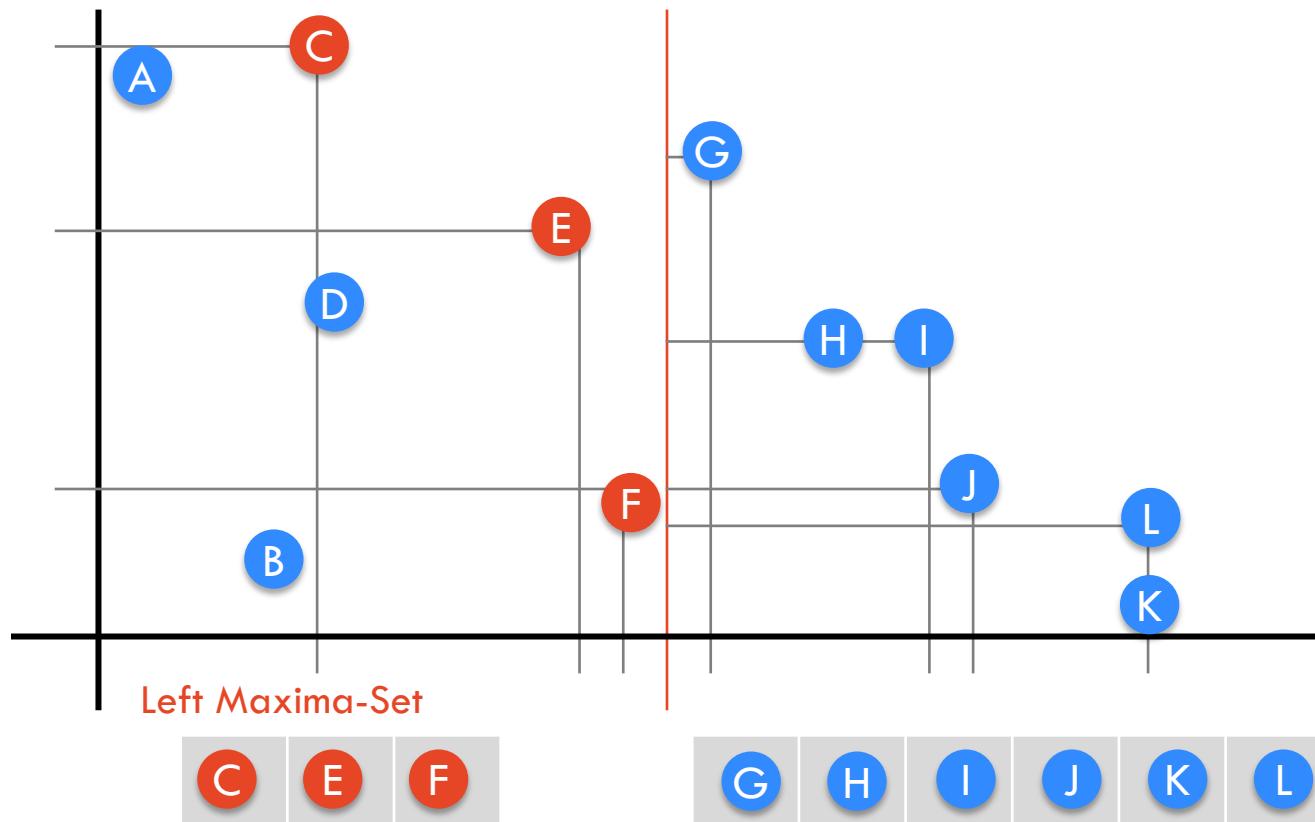


Sorted Points



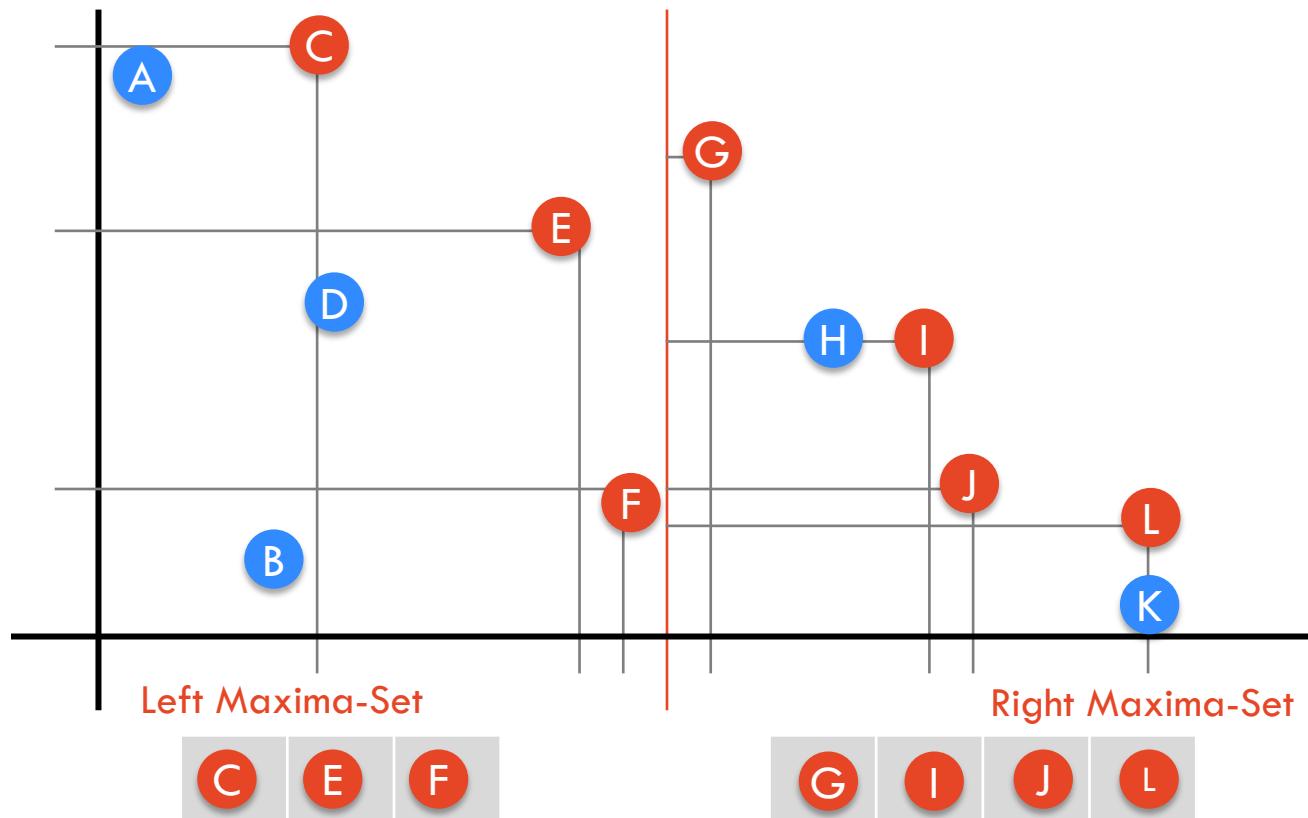
Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

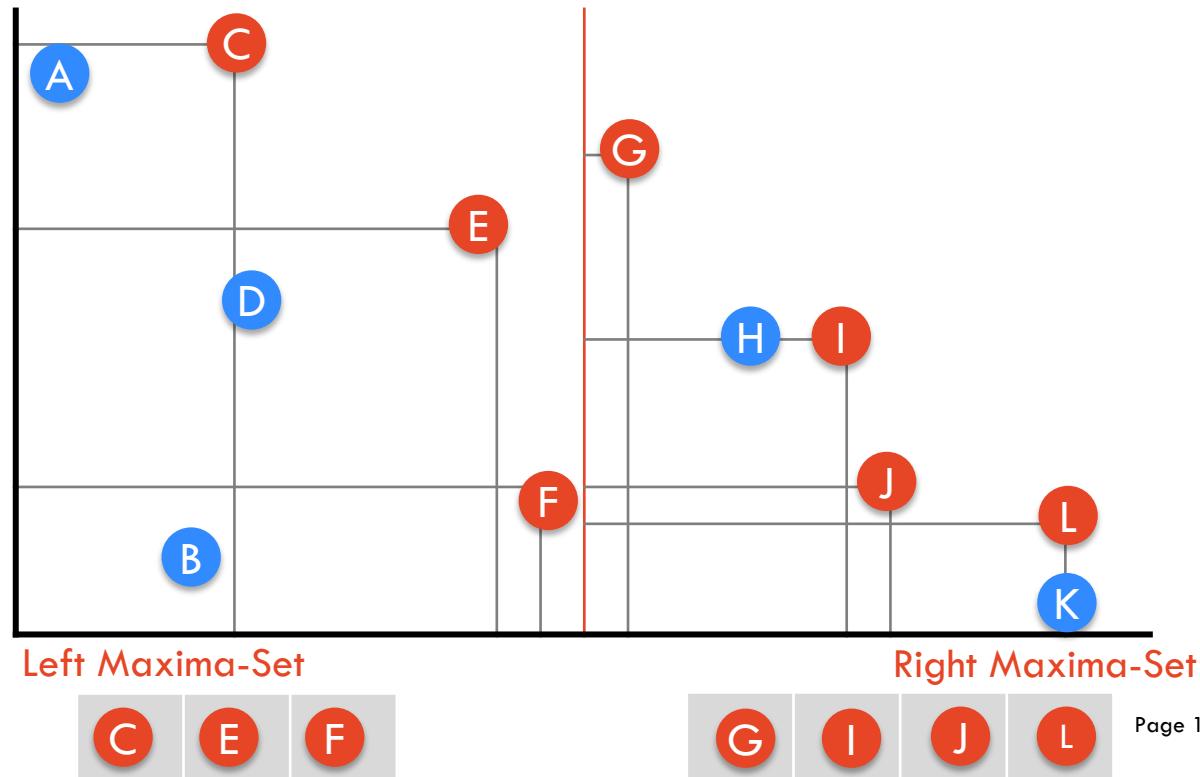
Conquer

1. Find the highest point p in the Right MS

$$p = G$$

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p



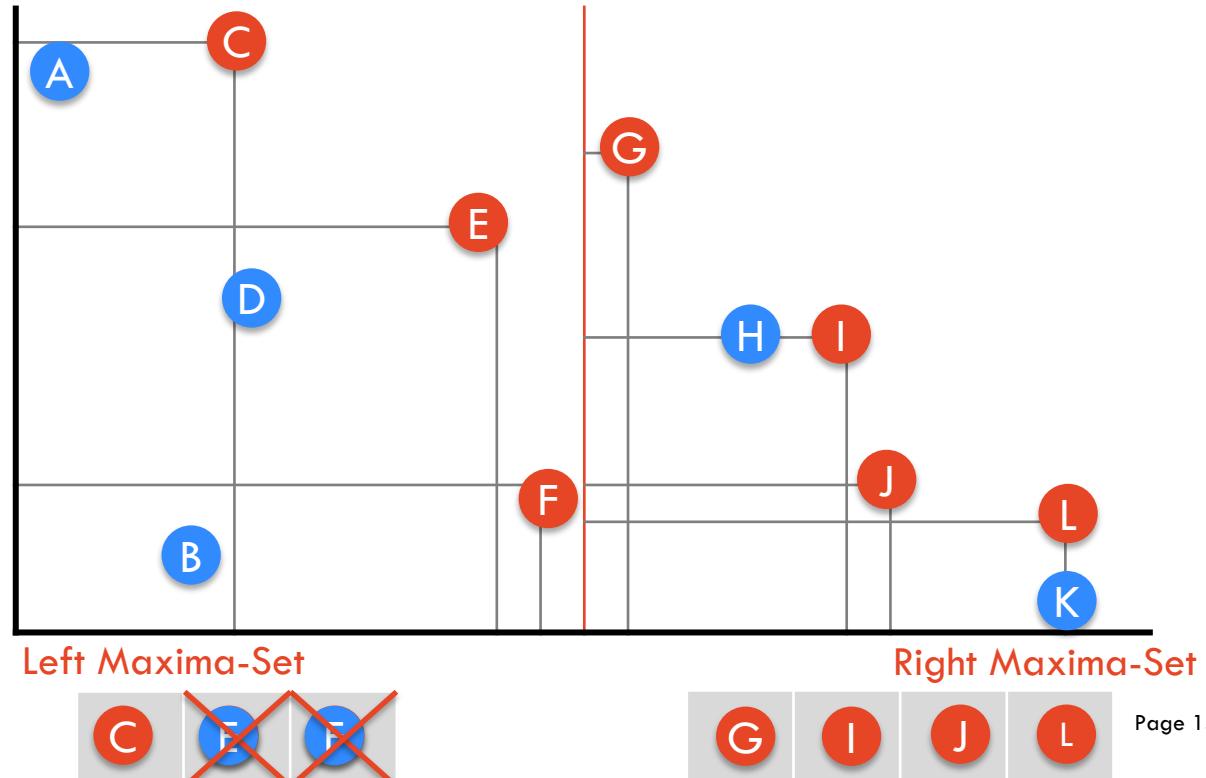
Maxima-Set

Conquer

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$$p = G$$

Merged Maxima-Set



Maxima-Set

Base case a single point.

The MS of a single point is the point itself.



Maxima-Set: Analysis

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

$O(n \log n)$

Divide sorted array into two halves.

$O(n)$

Recur recursively find the MS of each half.

$2T(n/2)$

Conquer compute the MS of the union of Left and Right MS

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$O(n)$

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Overall Running Time: pre-processing + $T(n) = O(n \log n)$

Maxima-Set: Correctness

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once.
Break ties in x by sorting by increasing y coordinate.

Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute MS of union of Left/Right MS

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
If $q.y > p.y$, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p

Integer multiplication

Given two n-digit integers x and y

Problem compute the product $x \cdot y$

While this seems like recreational mathematics, it does have real applications: Public key encryption is based on manipulating integers with thousands of bits.

Integer multiplication: Naïve approach

Given two n -digit integers x and y

Problem compute the product $x \cdot y$

Suppose we wanted to do it by hand. We assume that two digits can be multiplied or added in constant time

In primary school we all learn an algorithm for this problem that performs $\Theta(n^2)$ operations

Integer multiplication: Divide and conquer

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

Then $x \cdot y = x_1 \cdot y_1 \cdot 2^n + x_1 \cdot y_0 \cdot 2^{n/2} + x_0 \cdot y_1 \cdot 2^{n/2} + x_0 \cdot y_0$

We can compute the product of two n -digit numbers by making 4 recursive calls on $n/2$ -digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
    // x and y are positive integers represented in binary
    if x == 0 or y == 0 then return 0
    if x == 1 then return y
    if y == 1 then return x

    // recursive case
    let  $x_1$  and  $x_0$  be such that  $x = x_1 \cdot 2^{n/2} + x_0$ 
    let  $y_1$  and  $y_0$  be such that  $y = y_1 \cdot 2^{n/2} + y_0$ 

    return multiply( $x_1$ ,  $y_1$ )  $\cdot 2^n$  +
           (multiply( $x_1$ ,  $y_0$ ) + multiply( $x_0$ ,  $y_1$ ))  $\cdot 2^{n/2}$  +
           multiply( $x_0$ ,  $y_0$ )
```

Integer multiplication: Correctness

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

Then $x \cdot y = x_1 \cdot y_1 \cdot 2^n + x_1 \cdot y_0 \cdot 2^{n/2} + x_0 \cdot y_1 \cdot 2^{n/2} + x_0 \cdot y_0$

Straight forward application of induction to prove
that $\text{multiply}(x, y) = x \cdot y$

Integer multiplication: Complexity analysis

Recall $x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0$

Divide step (produce halves) takes $O(n)$

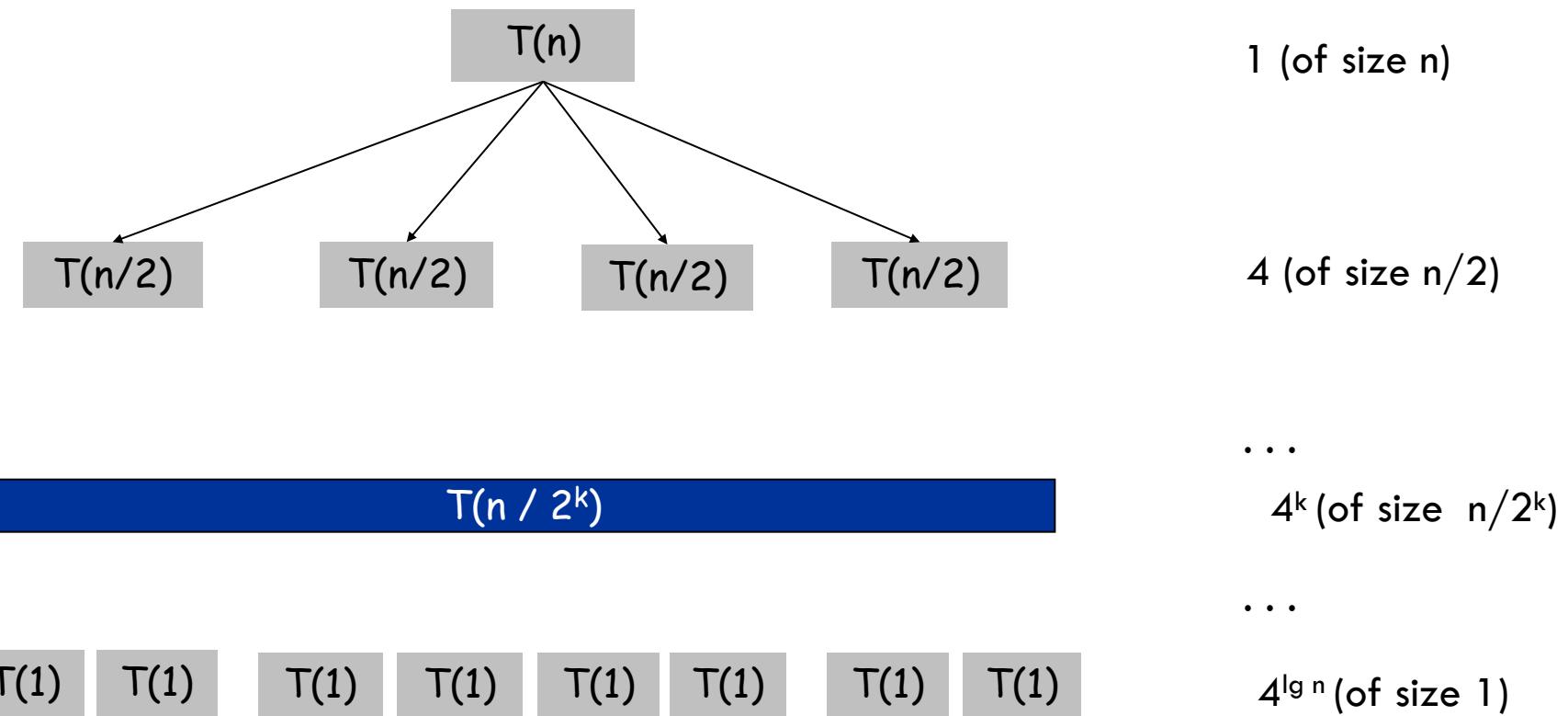
Recur step (solve subproblems) takes $4 T(n/2)$

Conquer step (add up results) takes $O(n)$

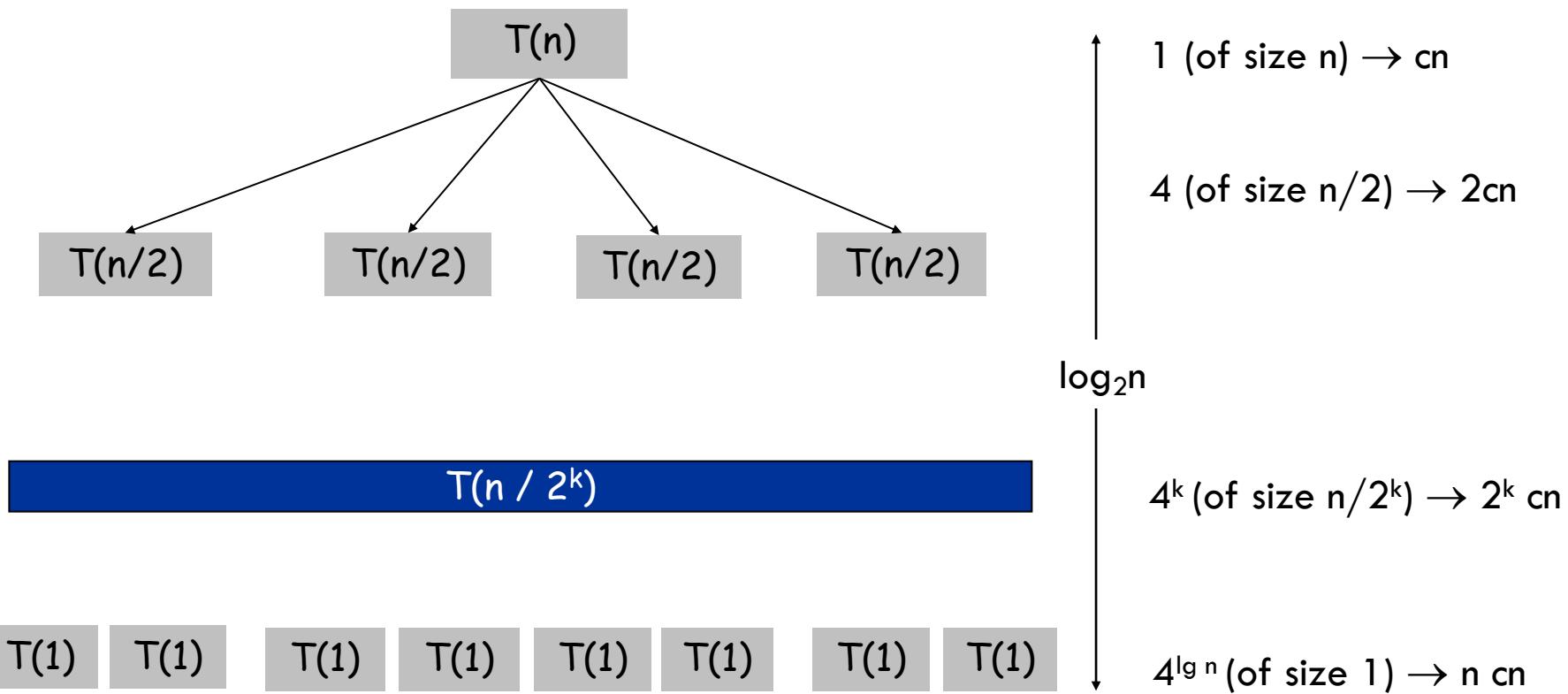
$$T(n) = \begin{cases} 4 T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^2)$. No better than naïve!!!

Proof by unrolling



Proof by unrolling



Integer multiplication: Divide and conquer v2.0

Let $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$

$$\begin{aligned}x \cdot y &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \\(x_1 + x_0)(y_1 + y_0) &= x_1 \cdot y_1 + x_1 \cdot y_0 + x_0 \cdot y_1 + x_0 \cdot y_0\end{aligned}$$

We can compute the product of two n -digit numbers by making 3 recursive calls on $n/2$ -digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
    // base case
    :
    // recursive case
    let  $x_1$  and  $x_0$  be such that  $x = x_1 \cdot 2^{n/2} + x_0$ 
    let  $y_1$  and  $y_0$  be such that  $y = y_1 \cdot 2^{n/2} + y_0$ 

    first_term = multiply( $x_1$ ,  $y_1$ )
    last_term = multiply( $x_0$ ,  $y_0$ )
    other_term = multiply( $x_1 + x_0$ ,  $y_1 + y_0$ )

    return first_term  $\cdot 2^n$  +
        (other_term - first_term - last_term)  $\cdot 2^{n/2}$  +
        last_term
```

Integer multiplication: Complexity analysis

Divide step (produce halves) takes $O(n)$

Recur step (solve subproblems) takes $3 T(n/2)$

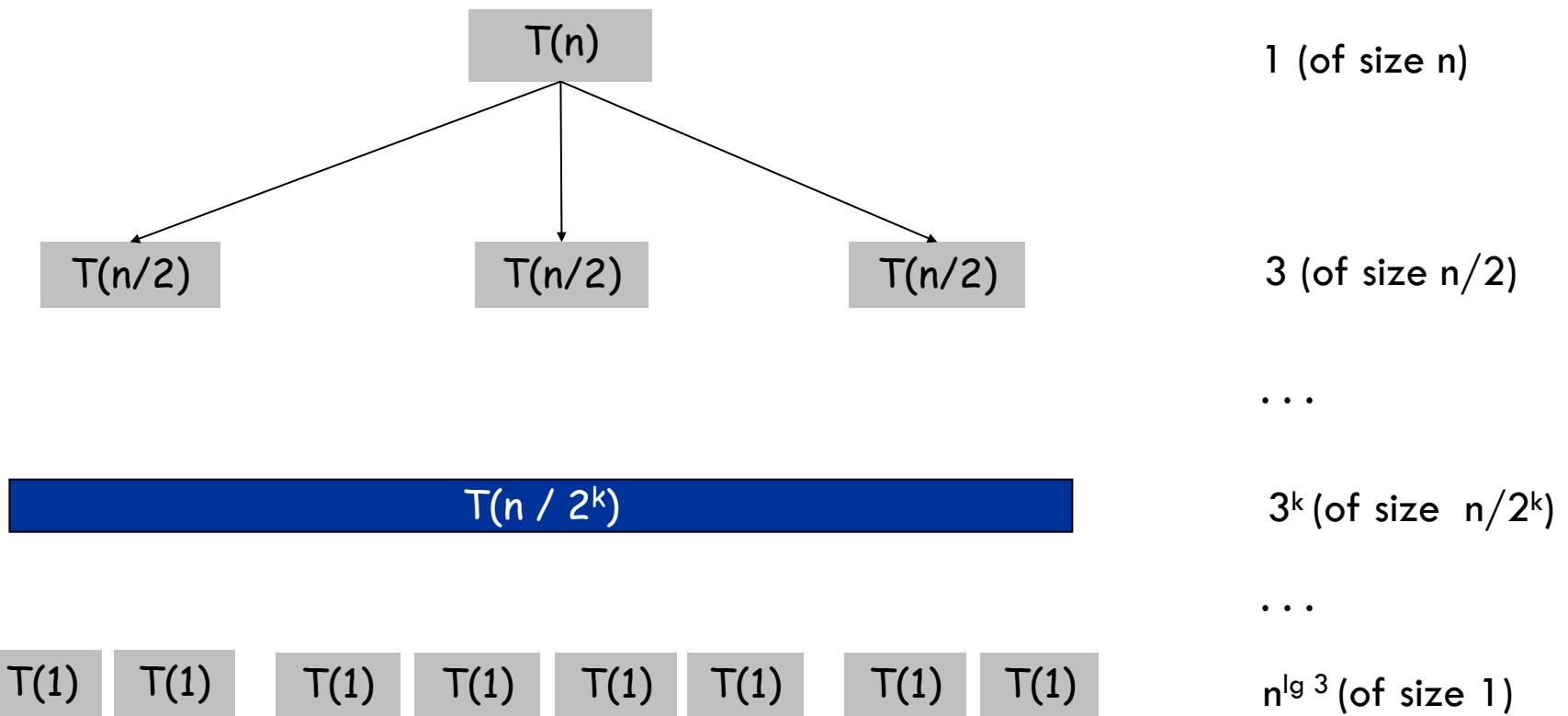
Conquer step (add up results) takes $O(n)$

$$T(n) = \begin{cases} 3 T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

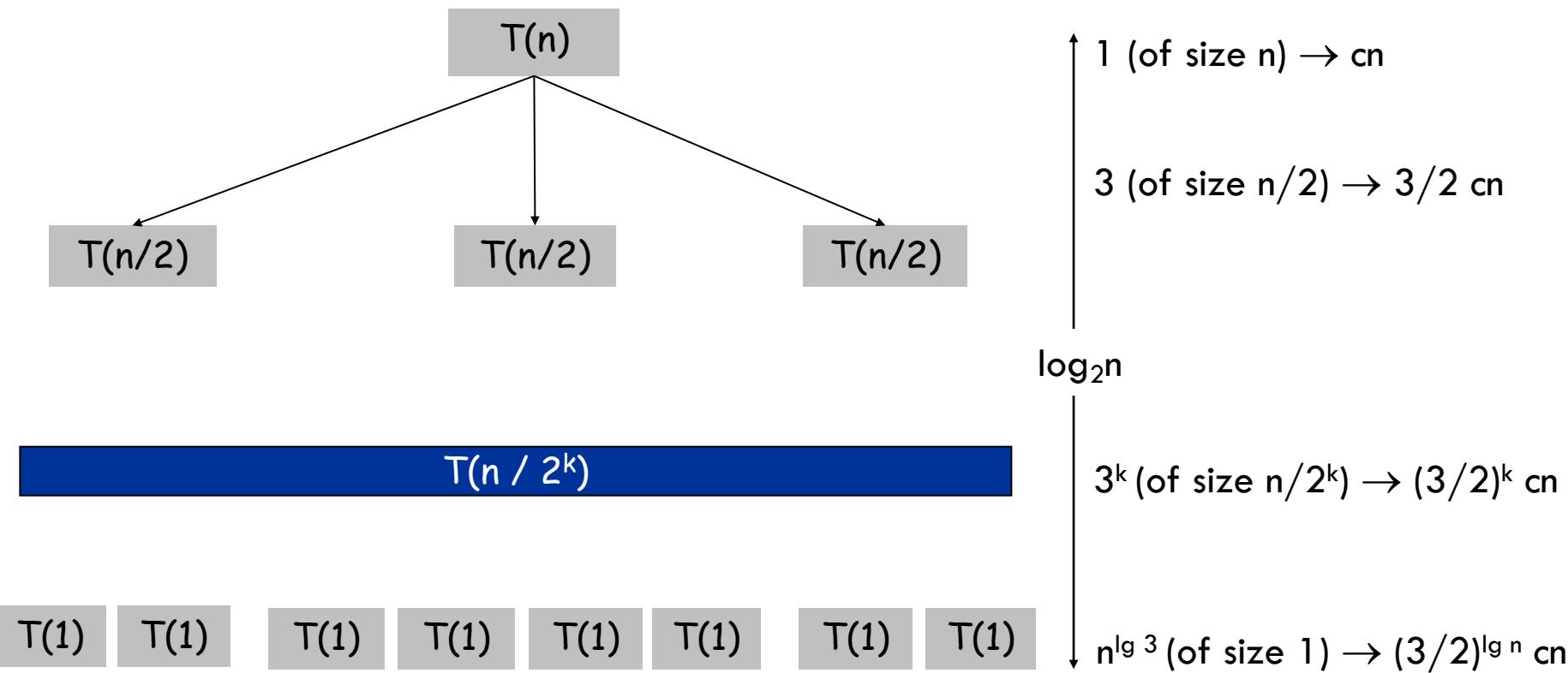
This solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$

Better than naïve!!!

Proof by unrolling



Proof by unrolling



Geometric series facts

Let r be a positive real and k a positive integer then

$$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$$

Consequently if $r > 1$ then

$$1 + r + r^2 + \dots + r^k < r^{k+1} / (r-1)$$

and if $r < 1$ then

$$1 + r + r^2 + \dots + r^k < 1 / (1-r)$$

Logarithms facts

Base exchange rule:

$$\log_a x = (\log_b x) / (\log_b a)$$

Product rule:

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

Power rule:

$$\log_a x^b = b \log_a x$$

Master Theorem

Let $f(n)$ and $T(n)$ be defined as follows:

$$T(n) = \begin{cases} a T(n/b) + f(n) & \text{for } n > 1 \\ c & \text{for } n < d \end{cases}$$

Depending on a , b and $f(n)$ the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a f(n/b) \leq \delta f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,

Note: You should be able to solve all recurrences in this class using unrolling, but if you are comfortable using the Master Theorem, go for it.

Selection

Given an unsorted array A holding n numbers and an integer k ,
find the k th smallest number in A

Trivial solution: Sort the elements and return k th element

Can we do better than $O(n \log n)$?

Yes, with divide and conquer!

First attempt

1. **Divide** find the median ($\lfloor n/2 \rfloor$ th element for simplicity) and split array on the halves, \leq and $>$ than the median
2. **Recur** if $k \leq \lfloor n/2 \rfloor$ find k th element on smaller half
if $k > \lfloor n/2 \rfloor$ find $(k - \lfloor n/2 \rfloor)$ th element on larger half
3. **Conquer** return value of the recursive call



$k = 6$
 $n = 10$



13
Conquer

Selection time complexity

Divide step (find median and split) takes at least $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

Conquer step (return recursive result) takes $O(1)$

If we could compute the median in $O(n)$ time then:

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

This solves to $T(n) = O(n)$ but only if we can solve the median problem, which is in fact a special case of selection with $k=[n/2]$

Second attempt: Approximating the median

We don't need the exact median. Suppose we could find in $O(n)$ time an element x in A such that

$$|A| / 3 \leq \text{rank}(A, x) \leq 2 |A| / 3$$

Then we get the recurrence

$$T(n) = \begin{cases} T(2n/3) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

Which again solves to $T(n) = O(n)$

To approximate the median we can use a recursive call!

Median of 3 medians

Consider the following procedure

- Partition A into $|A| / 3$ groups of 3
- For each group find the median
- Let x be the median of the medians

We claim that x has the desired property

$$|A| / 3 \leq \text{rank}(A, x) \leq 2|A| / 3$$

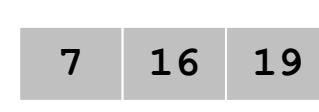
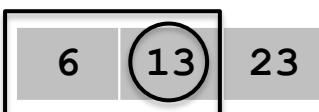
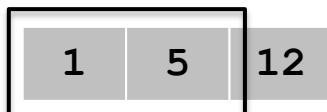
Half of the groups have a median that is smaller/larger than x , and each group has two elements smaller/larger than x , thus

$$\begin{aligned}\# \text{ elements smaller than } x &> 2(|A| / 6) = |A| / 3 \\ \# \text{ elements greater than } x &> 2(|A| / 6) = |A| / 3\end{aligned}$$

Median of 3 medians

Let x be the median of the medians, then

$$|A| / 3 \leq \text{rank}(A, x) \leq 2|A| / 3$$



elements smaller than $x > 2(|A| / 6) = |A| / 3$
elements greater than $x > 2(|A| / 6) = |A| / 3$

Median of 3 median time complexity

We don't need the exact median. With a recursive call on $n/3$ elements, we can find x in A such that

$$|A|/3 < \text{rank}(A, x) < 2|A|/3$$

Then we get the recurrence

$$T(n) = T(2n/3) + T(n/3) + O(n)$$

Which solves to $T(n) = O(n \log n)$

No better than sorting!

Median of 5 medians

We don't need the exact median. With a recursive call on $n/5$ elements, we can find x in A such that

$$3|A|/10 < \text{rank}(A, x) < 7|A|/10$$

Then we get the recurrence

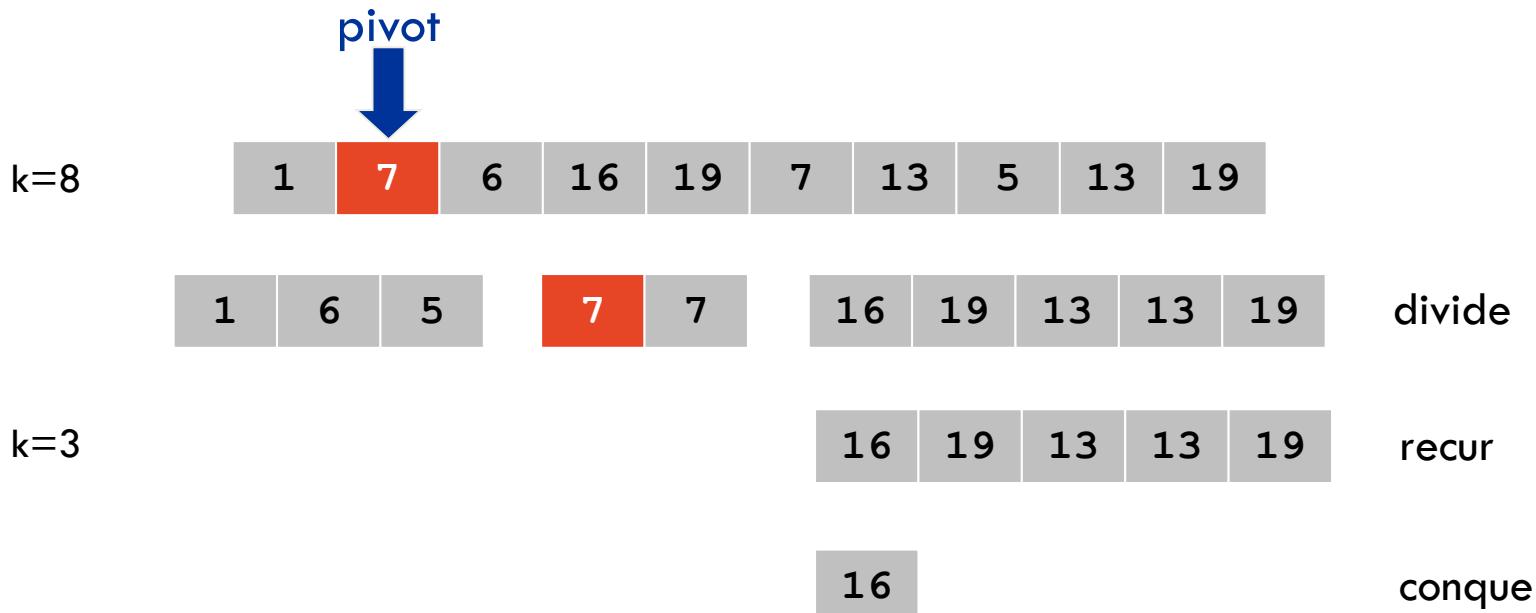
$$T(n) = T(7n/10) + T(n/5) + O(n)$$

Which solves to $T(n) = O(n)$

Asymptotically faster than sorting!

Quick selection

1. **Divide** Choose a random element from the list as the **pivot**
Partition the elements into 3 lists:
(i) less than, (ii) equal to and (iii) greater than the **pivot**
2. **Recur** Recursively select right element from correct list
3. **Conquer** Return solution to recursive problem



Quick selection complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n')$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n')] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n)$

(details available on the textbook but not examinable)

Announcements

Research internships:

<https://canvas.sydney.edu.au/courses/2806/pages/faculty-of-engineering-research-internship-program>

Apply by May 24:

<https://form.jotform.co/91317624817864>

Feedback (Unit of Study Survey). Submit by June 8:

<https://student-surveys.sydney.edu.au/students/>

Chance to win 64gb Apple iPad Air, an Apple Watch and JB HiFi Gift Cards

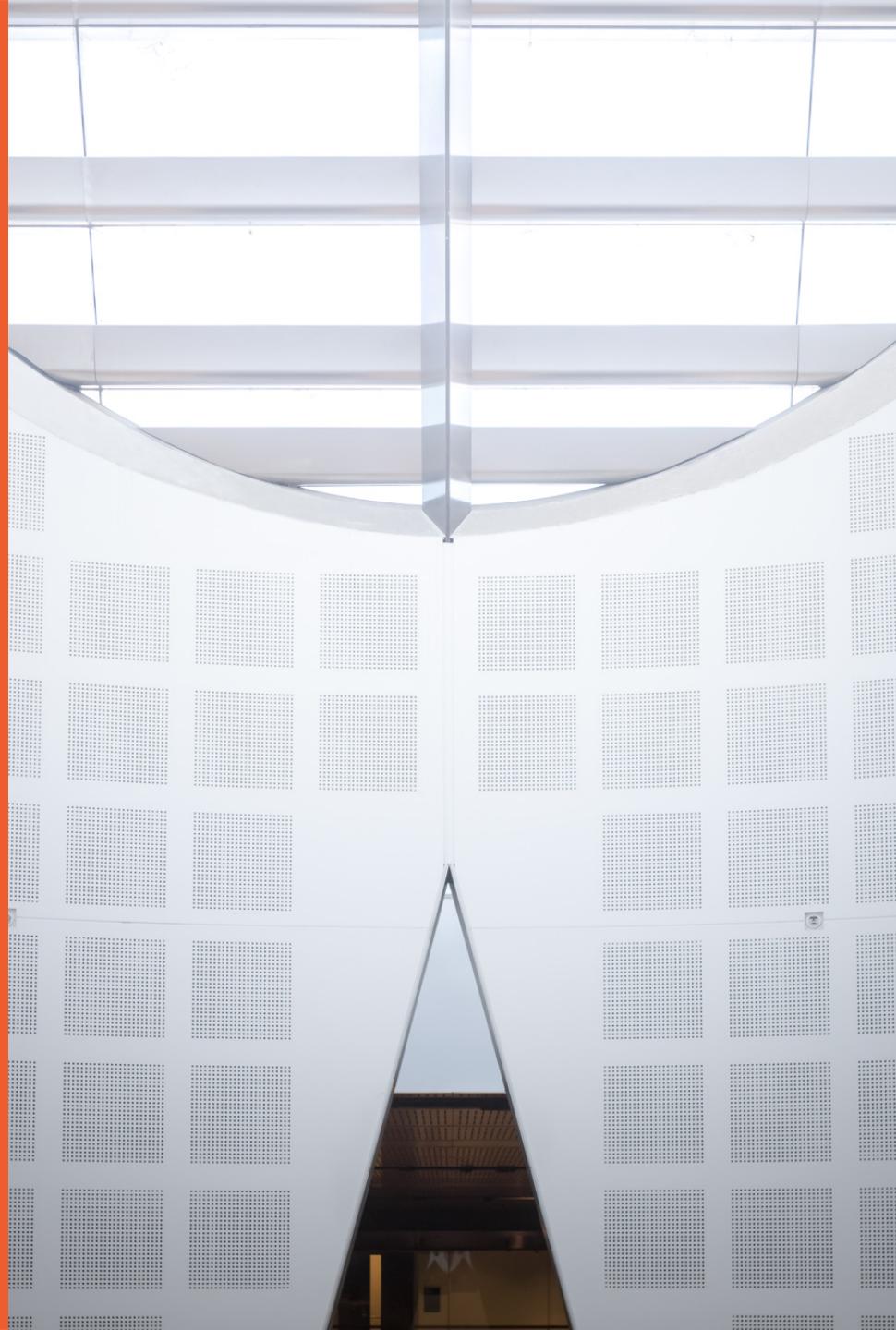
COMP2823

Recap and Exam Review

Dr. Julian Mestre
School of Computer Science



THE UNIVERSITY OF
SYDNEY



Quick announcements

Fill out online Unit of Study Survey

- <https://student-surveys.sydney.edu.au/students/>
- Use the free text to help us make this better for next years students.
“Pay it forward”

Examples of changes based on previous year feedback:

- consolidated the assignments from 10 to 6
- created a w0 introductory tutorial sheet to review inductive proofs
- split the advanced cohort (COMP2823) from the regular cohort (COMP2123), which allowed us to differentiate the units a bit more
- structured tutorial sheets into warm-up, problem solving and advanced problem solving
- made quizzes a bit more challenging

Assignment

Posted assignment 3 and 4 marks in canvas. Please check that everything looks alright.

Posted solutions to assignment 5. Marking is underway.

Week 13 Tutorial

There won't be a tutorial sheet this week, but you can use this last tutorial to ask questions about any material covered during the semester

Week 13 Quiz

Quiz 10 will be about the final. Unlike previous quizzes you will be able to attempt it multiple times.

It's available until the start of the exam.

Looking back

Lecture slides

Week 1 - analysis	PDF
Week 1 - introduction	PDF
Week 1 - running time demo	IPYNB
Week 2 - list append demo	IPYNB
Week 2 - lists	PDF
Week 2 - stacks and queues	PDF
Week 3 - trees	PDF
Week 4 - binary search tree	PDF
Week 5 - priority queue	PDF
Week 6 - hashing	PDF
Week 7 - graphs	PDF
Week 8 - shortest path and mst	PDF
Week 9 - coding interview demo	IPYNB
Week 9 - greedy method	PDF
Week 10 - randomization	PDF
Week 11 - divide_and_conquer	PDF
Week 12 - divide_and_conquer_ii	PDF

We covered a
lot of ground!

Core concept 1: Abstraction layers

Abstract Data Type



Data Structure



Computer code

Problem definition



Algorithm



Computer code

Core concept 2: Algorithm analysis

A principled framework for evaluating algorithms:

- measuring performance of resource use
- proving correctness

These should inform your design and implementation choices

Learning outcomes

1. Proficiency in organising, presenting and discussing professional ideas [...]
2. Using mathematical methods to evaluate the performance of an algorithm.
3. Using notation of big-O to represent asymptotic growth of cost functions.
4. Understanding of commonly used data structures, including lists, stacks, queues, priority queues, search trees, hash tables, and graphs. This covers the way information is represented in each structure, algorithms for manipulating the structure, and analysis of asymptotic complexity of the operations.
5. Understanding of basic algorithms related to data structures, such as algorithms for sorting, tree traversals, and graph traversals.
6. Ability to write code that recursively performs an operation on a data structure.
7. Experience designing an algorithmic solution to a problem, coding it, and analysing its complexity.
8. Ability to apply basic algorithmic techniques (e.g. divide-and-conquer, greedy) to given design tasks.

Beyond this unit of study

SCS offers many algorithmic units:

- **COMP2022** Models of Computation (S2)
- **COMP3027** Algorithm Design (S1)
- **COMP3530** Discrete Optimization (S2, not in 2020)
- **COMP5045** Computational Geometry (S1)

Sydney Algorithms and Computation Theory:

- weekly seminar on Algorithms research
- do a research project with us
- we are always looking for motivated honors students

What is examinable?

Everything from the lectures, the referenced sections of the textbooks, the tutorials, the quizzes, the assignments. Exceptions to this rule:

- when explicitly labeled as non-examinable.
- probabilistic analysis of randomized algorithms

In general though, if it happened during this unit, you are expected to know about it!

Focus on the things we put most emphasis on, as seen in tutorials and assignments

Final Exam Structure

2 hours writing plus 10 minutes reading

5 questions worth in total 100 points

Worth 60% of overall COMP2823 grade

Final exam has usual a 40% barrier

Do's and Don'ts

Type your answers and submit it as an assignment in **Canvas**

Handwritten/scanned answers will not be accepted.

Can use one double-sided A4 sheet summary

- Made by yourself
- Handwritten or typed
- Don't submit this

Start your submission with your student ID

- Don't include your name

Problem 1

20 points

Analysis of given algorithms

Easy problem. Make sure you nail it!

Problem 2

20 points

Analysis of given algorithms

Easy problem. Make sure you nail it!

Problem 3

20 points

Design or modify an ADT

Medium difficulty problem.

Remember to:

- Describe your data structure
- Prove correctness
- Analyze complexity

Problem 4

20 points

Design an algorithm for a “new” problem

Medium difficulty problem.

Remember to:

- Describe your algorithm
- Prove correctness
- Analyze complexity

Problem 5

20 points

Design an algorithm for a completely new problem

Hard problem.

Remember to:

- Describe your algorithm
- Prove correctness
- Analyze complexity

Problem 3-5

Check if you're supposed to use a specific technique:

- “design a greedy algorithm”
- “design a divide and conquer algorithm”

Let the running time requirement guide you:

- If we ask $O(1)$ time, this limits your options considerably
- If we ask $O(n)$ time, you can't sort the input

Exam technique

Read all questions to see which ones you can answer quickly

Plan how you will allocate time (wisely)

Start with easy problems and move to harder ones

Write clearly and efficiently

- Start with outline/bullet points, then expand if you have time
- No need for fancy style or overly formal

Figures take a lot of time to draw

- Describing in text is faster (a graph is a bunch of vertices and edges)
- No scanned drawings allowed

Pragmatic Advice

- Practice submitting a file in Canvas!
- Be alone in your room to avoid distractions
- Let housemates know when your exam is to avoid distractions
- Bring water
- Have clothing in layers
- Start your submission with your student ID
- Do not write your name on the exam (marking is anonymous)
- Breathe
- Relax
- Do not contact teaching staff during the exam

Good Luck!!!