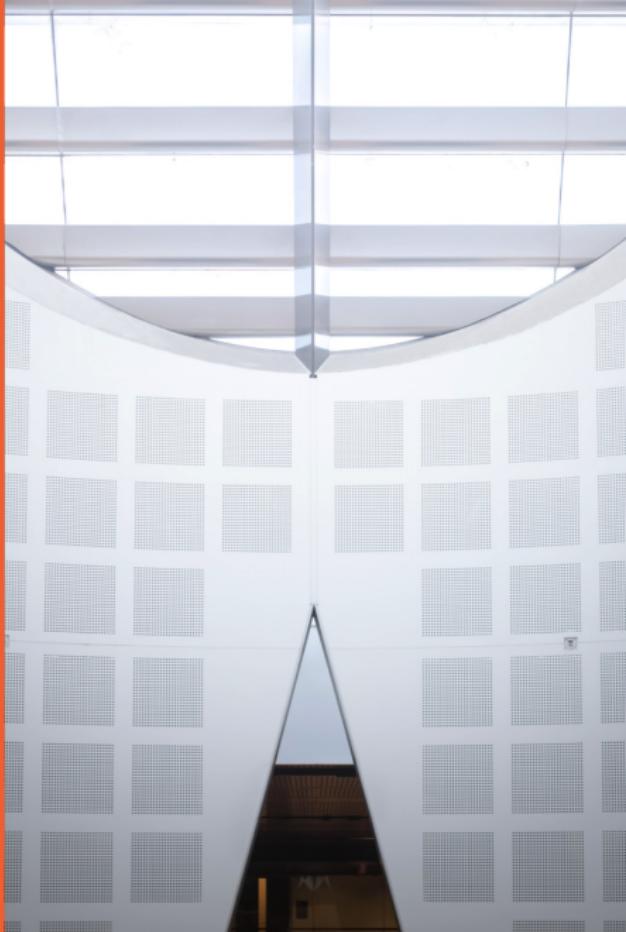


Data Structures and Algorithms

Algorithm analysis

[GT 1.1.5-1.1.6, 1.3]

Presented by
André van Renssen
School of Computer Science



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
(what your solution does)
- different from implementation

Correctness and complexity analysis:

- a formal proof that the algorithm solves the problem
(why is what your solution does correct)
- analytical bound on the resources it uses

Pseudocode

Control flow

- `if ... then ... [else ...]`
- `while ... do ...`
- `repeat ... until ...`
- `for ... do ...`
- Indentation replaces braces

Method call

- `method (arg [, arg...])`

Return value

- `return expression`

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Algorithm:

We go through all elements of the array in order and keep track of the largest element found so far (initially $-\infty$). So for each position i , we check if the value stored at $A[i]$ is larger than our current maximum, and if so we update the maximum. After scanning through the array, return the maximum we found.

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Optional pseudocode:

```
max ← −∞  
for  $i \leftarrow 0$  to  $n - 1$  do  
    if  $A[i] > max$  then  
        max ←  $A[i]$   
return max
```

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Correctness:

We maintain the following invariant: after the k -th iteration, \max stores the maximum of the first k elements.

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Correctness:

We maintain the following invariant: after the k -th iteration, \max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, \max is $-\infty$, which is the maximum of the first 0 elements.

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Correctness:

We maintain the following invariant: after the k -th iteration, \max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, \max is $-\infty$, which is the maximum of the first 0 elements.

Assume the invariant holds for the first k iterations, we show that it holds after the $(k + 1)$ -th iteration. In that iteration we compare \max to $A[k]$ and update \max if $A[k]$ is larger. Hence, afterwards \max is the maximum of the first $k + 1$ elements.

Small example

Computational problem:

We are given an array A of integers and we need to return the maximum.

Correctness:

We maintain the following invariant: after the k -th iteration, \max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, \max is $-\infty$, which is the maximum of the first 0 elements.

Assume the invariant holds for the first k iterations, we show that it holds after the $(k + 1)$ -th iteration. In that iteration we compare \max to $A[k]$ and update \max if $A[k]$ is larger. Hence, afterwards \max is the maximum of the first $k + 1$ elements.

The invariant implies that after n iterations, \max contains the maximum of the first n elements, i.e., it's the maximum of A .

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

Naive algorithm

High level description:

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

Naive with preprocessing

```
curr_val, curr_ans ← 0, (None, None)
for i ← 0 to n − 1 do
    for j ← i to n − 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }
        s ← 0
        for k ← i to j do
            s ← s + A[k]
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)
return curr_ans
```

Naive with preprocessing

```
curr_val, curr_ans ← 0, (None, None)
for i ← 0 to n − 1 do
    for j ← i to n − 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }
        s ← 0
        for k ← i to j do
            s ← s + A[k]
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)
return curr_ans
```

Why recompute s every time?

Naive with preprocessing

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

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

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

Naive with preprocessing

```
curr_val, curr_ans ← 0, (None, None)
B ← new array of size n + 1
B[0] ← 0
for i ← 1 to n + 1 do
    B[i] ← B[i − 1] + A[i − 1]
for i ← 0 to n − 1 do
    for j ← i to n − 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }
        s ← B[j + 1] − B[i]
        {Compare to current maximum}
        if  $s > curr\_val$  then
            curr_val, curr_ans ← s, (i, j)
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 implementation 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 strawman baseline
- independently agreed upon

Efficiency

Definition

An algorithm is efficient if it runs in polynomial time; that is, on an 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 some information about the expected behavior of the algorithm and is useful for making predictions and comparing different algorithms.

Asymptotic growth analysis

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

Problem: figuring out $T(n)$ exactly might be really hard! Also, the fine-grained details are not necessarily that relevant.

Example: $T(n) = 4n^2 + 4n + 5$, or $T(n) = 5n^2 - 2n + 100$.

Which one is best? Do the constants matter (recall, one “step” might take a slightly different time based on implementation or architecture details)?

Asymptotic growth analysis

Insight: in both examples, the worst-case number of steps $T(n)$ grew *quadratically* with n .

If n is multiplied by 2, then we expect $T(n)$ to be multiplied by 4.

More generally: if $T(n)$ 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 .

Asymptotic growth analysis gives us a tool for focusing on the terms that make up $T(n)$, which **dominate** the running time.

Asymptotic growth analysis

Recap: We want to analyse $T(n)$, the worst-case number of steps of our algorithm on an instance of size n .

But figuring out $T(n)$ exactly might be very hard (impossible), and also is often “too much information”.

We instead do asymptotic growth analysis (Big-Oh notation), which provides a coarser but sufficient way to summarise how $T(n)$ behaves when n increases.

Asymptotic growth analysis

Definition

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

Definition

We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ 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

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ 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

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $\Omega(n^2)$ and $\Omega(n)$, but not $\Omega(n^3)$.

Definition

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

Asymptotic growth analysis

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $\Omega(n^2)$ and $\Omega(n)$, but not $\Omega(n^3)$.

Definition

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

$$T(n) = 32n^2 + 17n + 32$$

$T(n)$ is $\Theta(n^2)$, but not $\Theta(n)$ or $\Theta(n^3)$.

Asymptotic growth analysis

tl;dr: think of those as

- $T(n) = O(f(n))$: $T(n)$ is “smaller” than $f(n)$ (up to a constant factor)
- $T(n) = \Omega(f(n))$: $T(n)$ is “bigger” than $f(n)$ (up to a constant factor)
- $T(n) = \Theta(f(n))$: $T(n)$ is “equal” to $f(n)$ (up to constants factors)

Important: Asymptotic growth analysis ($O(\cdot), \Omega(\cdot), \Theta(\cdot)$) is just a mathematical tool to compare functions when the input n grows. We are using this tool to analyse the worst-case behaviour of our algorithms. But asymptotic growth analysis and worst-case analysis are not the same thing (they just go hand in hand)!

Examples of asymptotic growth

Polynomial

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

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

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

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

Logarithmic

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

Exponential

Examples of asymptotic growth

Polynomial

$O(n^c)$, considered 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 $f = O(g)$ 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 ...

constant	$T(n) = \Theta(1)$
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	$T(n) = \Theta(c^n)$

What operations take $O(1)$ time?

Constant time:

Running time does not depend on the size of the input.

- Assignments ($a \leftarrow 42$)
- Comparisons ($=, <, >$)
- Boolean operations (and, or, not)
- Basic mathematical operations ($+, -, *, /$)
- Constant sized combinations of the above ($a \leftarrow (2 * b + c) / 4$)

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

```
curr_val, curr_ans ← 0, (None, None)
for i ← 0 to n − 1 do
    for j ← i to n − 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }
        s ← 0
        for k ← i to j do
            s ← s + A[k]
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)
return curr_ans
```

Naive algorithm

```
curr_val, curr_ans ← 0, (None, None)          O(1)
for i ← 0 to n - 1 do
    for j ← i to n - 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }
        s ← 0
        } O(1)
        for k ← i to j do
            s ← s + A[k]
            } O(1)
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)
        } O(1)
return curr_ans                                O(1)
```

Naive algorithm

```
curr_val, curr_ans ← 0, (None, None)                                O(1)
for i ← 0 to n - 1 do
    for j ← i to n - 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }                      }
        s ← 0                                                               } O(1)
        for k ← i to j do
            s ← s + A[k]                                              } O(1)      } O(j - i)
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)   } O(1)
    return curr_ans                                                        O(1)
```

Naive algorithm

```
curr_val, curr_ans ← 0, (None, None) O(1)
for i ← 0 to n - 1 do
    for j ← i to n - 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ } O(1)
        s ← 0
        for k ← i to j do
            s ← s + A[k]
        } O(1)
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)
        } O(1)
    } O(j - i)
} O(n-1)
return curr_ans O(1)
```

Naive algorithm

```
curr_val, curr_ans ← 0, (None, None)                                O(1)
for i ← 0 to n - 1 do
    for j ← i to n - 1 do
        {Compute  $A[i] + A[i + 1] + \dots + A[j]$ }           } O(1)
        s ← 0
        for k ← i to j do
            s ← s + A[k]                               } O(1)
        {Compare to current maximum}
        if s > curr_val then
            curr_val, curr_ans ← s, (i, j)   } O(1)
    return curr_ans
```

$\left. \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right\} O(j-i)$ $\left. \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right\} \sum_{j=i}^{n-1}$ $\left. \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right\} \sum_{i=0}^{n-1}$ $\left. \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right\} O(1)$

Naive algorithm

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

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

Naive algorithm

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

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

See GT 1.2 for a refresher if needed.

Naive with preprocessing

curr_val, curr_ans $\leftarrow 0, (None, None)$

B \leftarrow new array of size $n + 1$

B[0] $\leftarrow 0$

for *i* $\leftarrow 1$ to $n + 1$ do

B[*i*] \leftarrow *B*[*i* - 1] + *A*[*i* - 1]

for *i* $\leftarrow 0$ to $n - 1$ do

 for *j* $\leftarrow i$ to $n - 1$ do

 {Compute $A[i] + A[i + 1] + \dots + A[j]$ }

s \leftarrow *B*[*j* + 1] - *B*[*i*]

 {Compare to current maximum}

 if *s* > *curr_val* then

curr_val, curr_ans $\leftarrow s, (i, j)$

return *curr_ans*

Naive with preprocessing

curr_val, curr_ans $\leftarrow 0, (None, None)$

B \leftarrow new array of size $n + 1$

B[0] $\leftarrow 0$

for *i* $\leftarrow 1$ to $n + 1$ do

B[*i*] \leftarrow *B*[*i* - 1] + *A*[*i* - 1]

for *i* $\leftarrow 0$ to $n - 1$ do

 for *j* $\leftarrow i$ to $n - 1$ do

 {Compute $A[i] + A[i + 1] + \dots + A[j]$ }

s \leftarrow *B*[*j* + 1] - *B*[*i*]

 {Compare to current maximum}

 if *s* > *curr_val* then

curr_val, curr_ans $\leftarrow s, (i, j)$

return *curr_ans*

Improvement: $O(n^3) \rightarrow O(n^2)$

Recap

Asymptotic growth analysis gives us some information about the worst-case 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 analyse 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.
- Analyse 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

What we will be using in this class closely follows 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.

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 2: Lists

[GT 2.1-2.2]

Dr. André van Renssen

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 issues are separated into different places.
- Client can be considered at a higher, more abstract, level.
- Many different systems can use the same library, so only code tricky manipulations once, rather than in every client 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.

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

Index-Based Lists (List ADT)

An index-based list (usually) 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

add(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

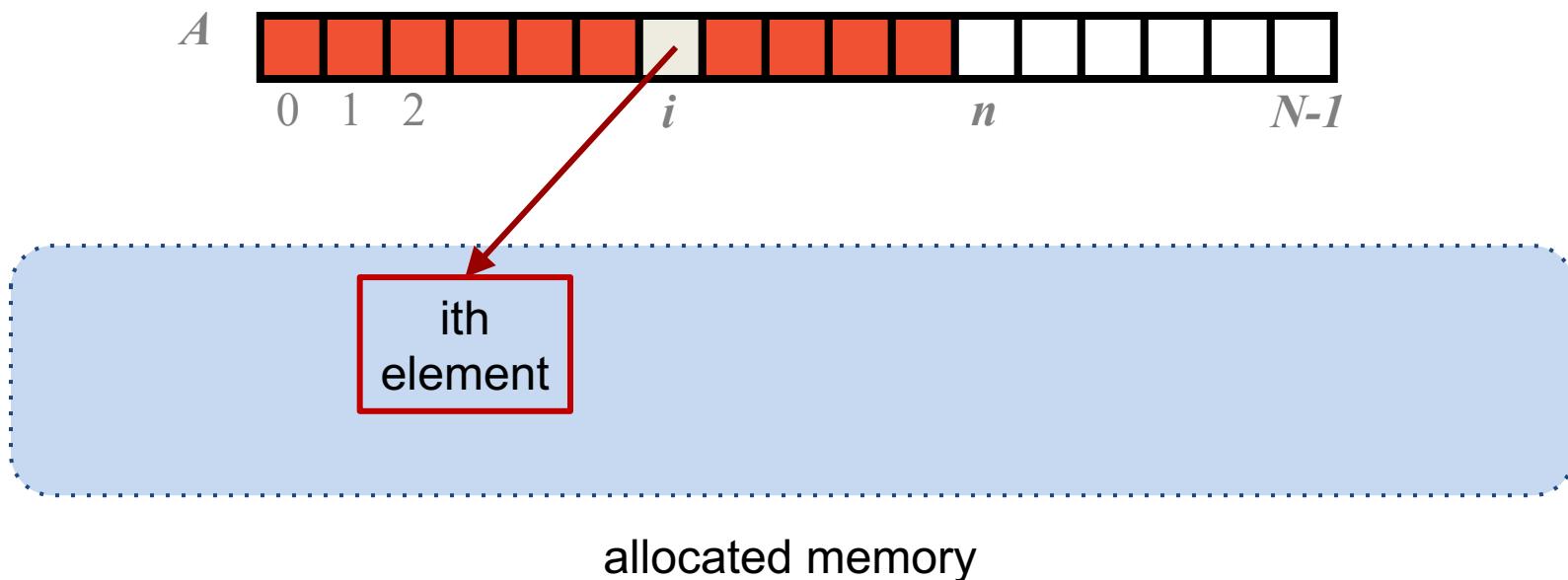
Example

A sequence of List operations:

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

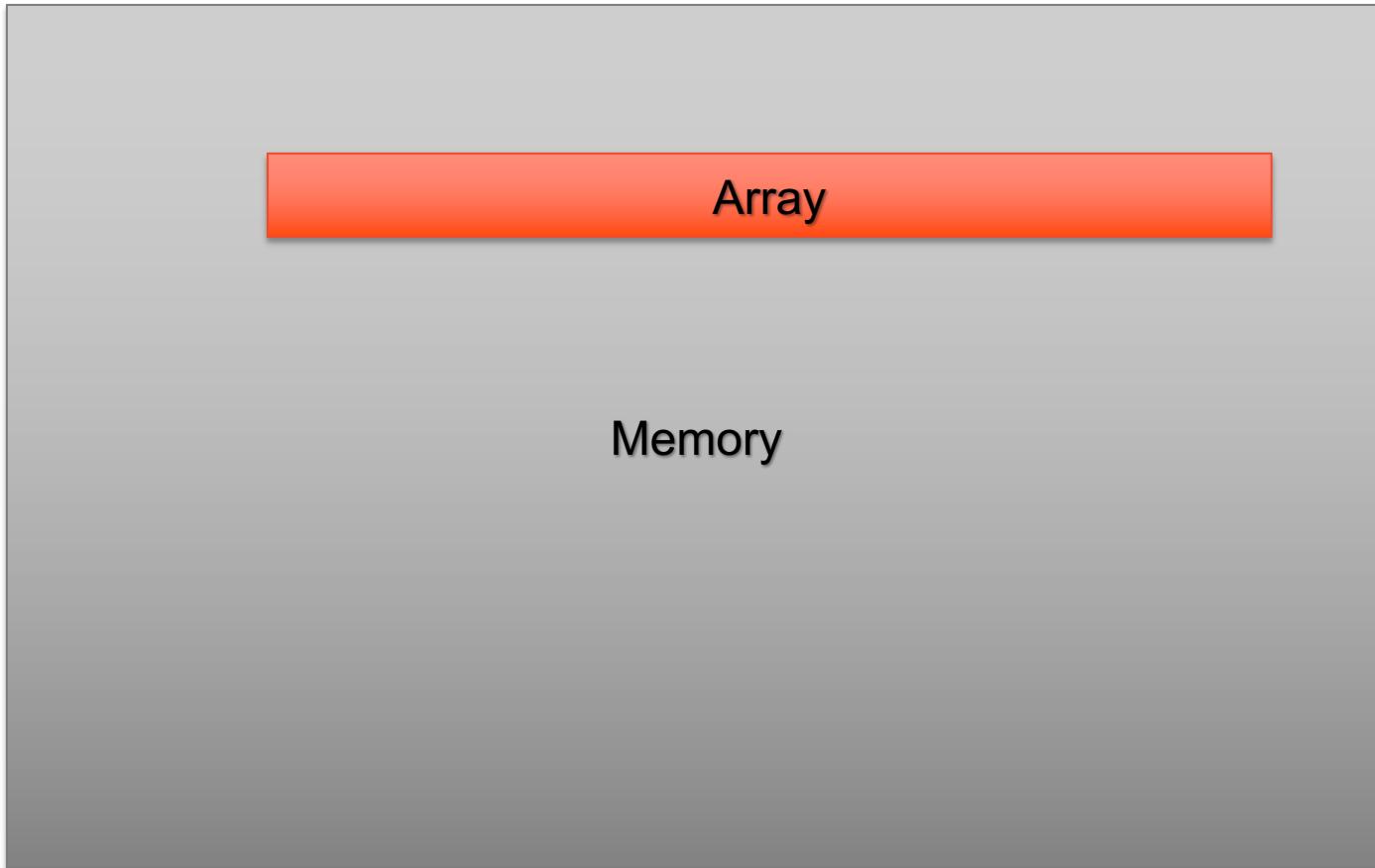
Array-based Lists

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 $n \leq N$



Array-based Lists

How is an array stored?

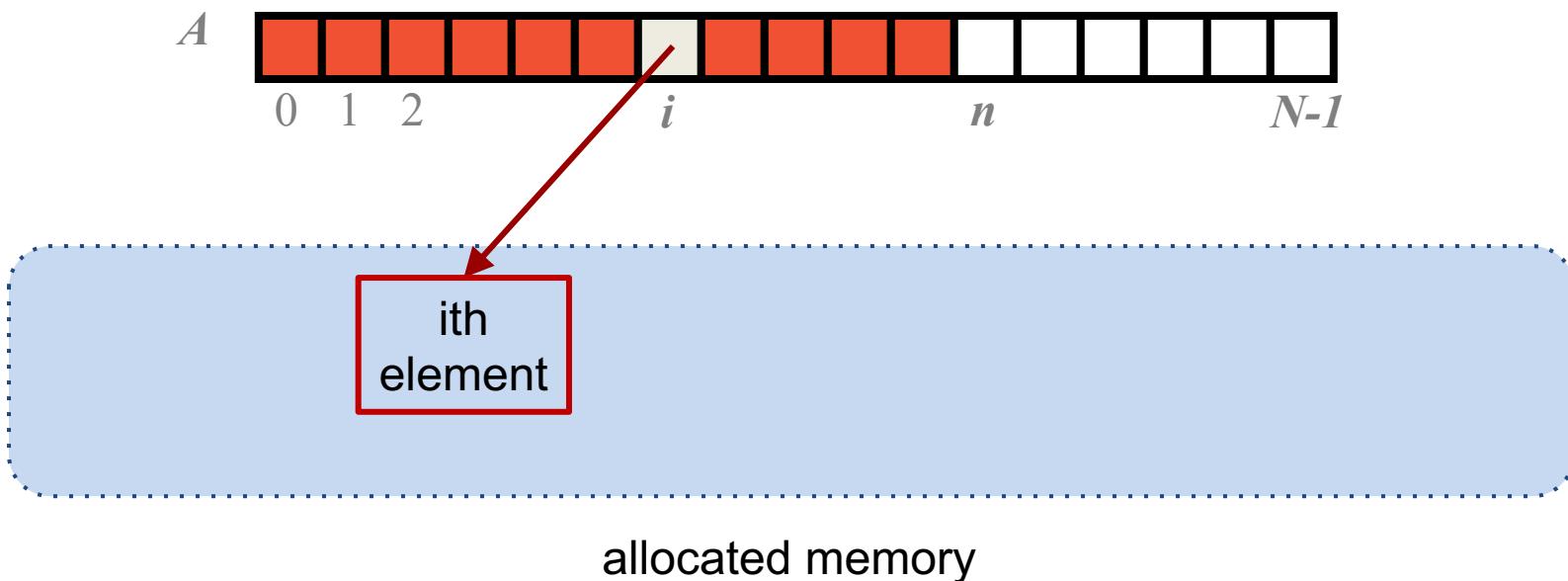


Array-based Lists: get(i)

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

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

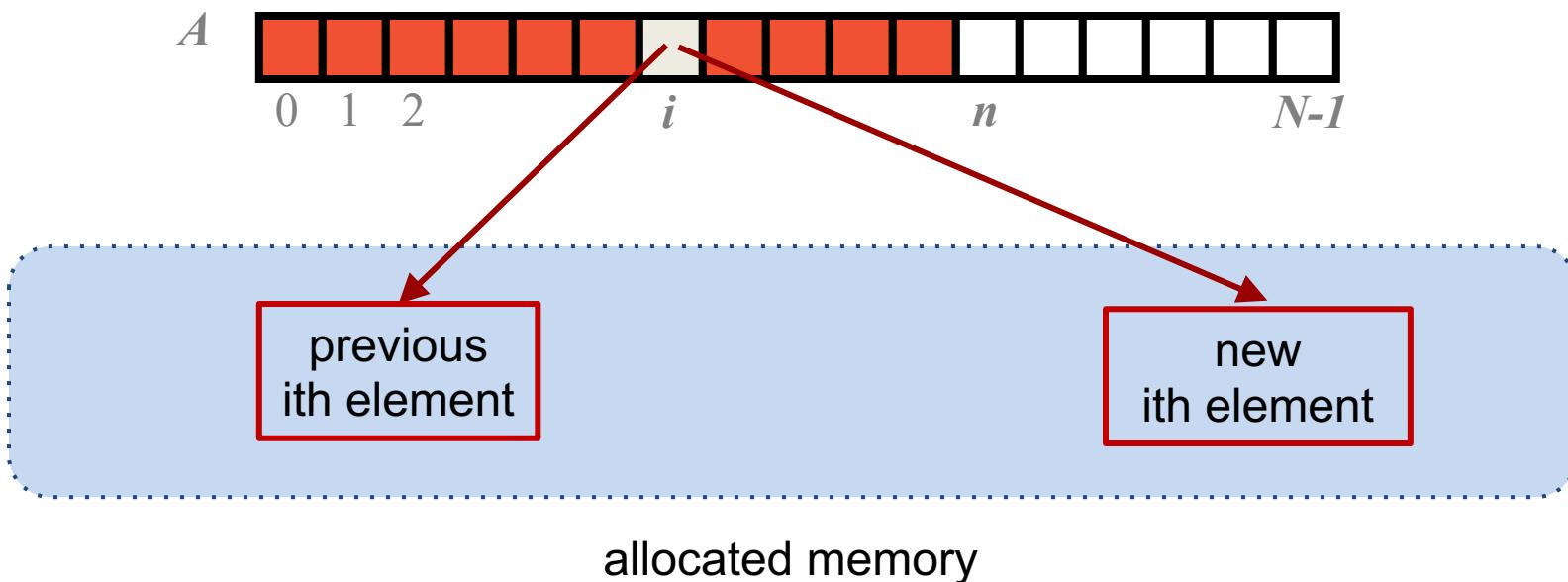
Both operations can be carried out in constant time
(a.k.a. $\mathcal{O}(1)$ time), independent of the size of the array



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"
    result ← A[i]
    A[i] ← e
    return result
```

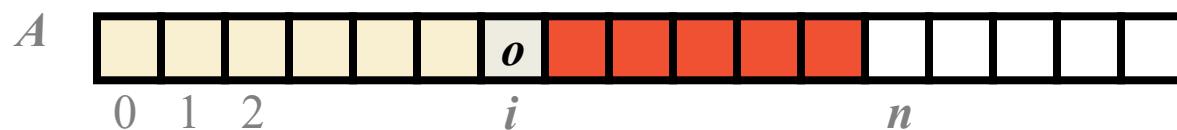
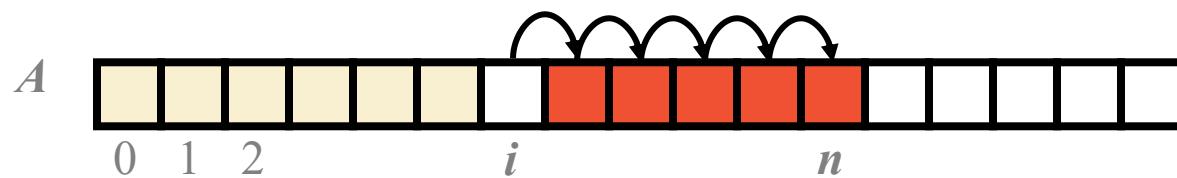
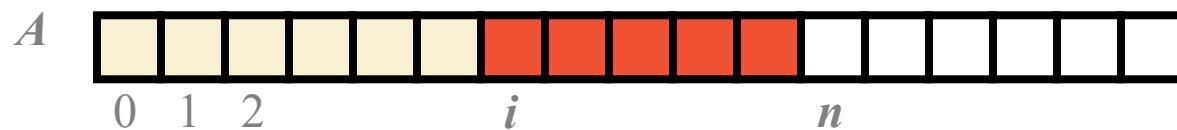
Time complexity of operation is $O(1)$ time, independent of the size of the array (N) or the represented list (n)

Array-based Lists: add(i, e)

In an operation $\text{add}(i, e)$, we must make room for the new element by shifting forward $n - i$ 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

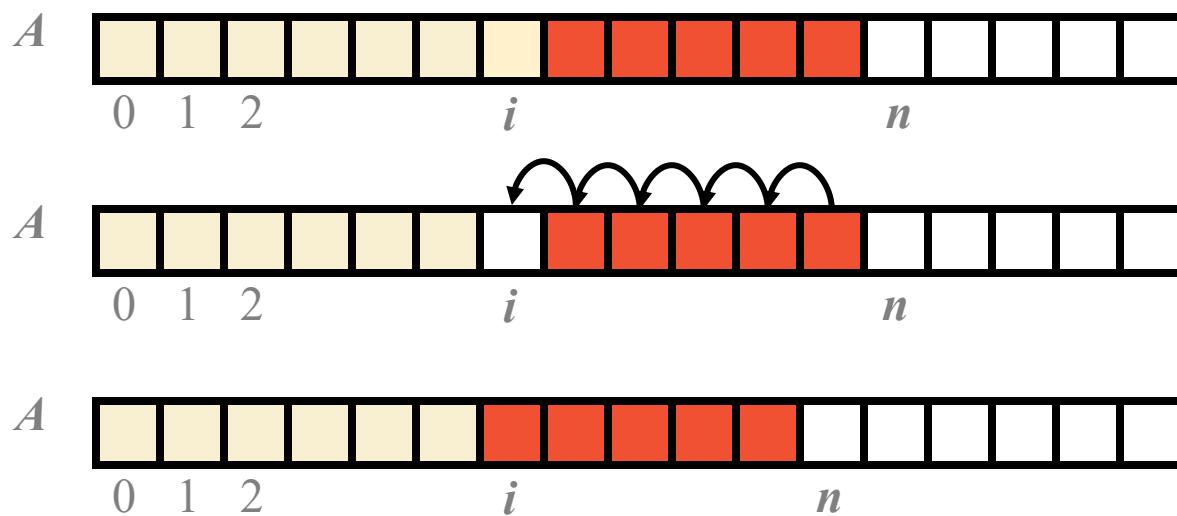
```
def add(i, e):
    if n = N then
        return "array is full"
    if i < n then
        for j in [n-1, n-2, ... , i] do
            A[j + 1] ← A[j]
        A[i] ← e
    n ← n + 1
```

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 $n - i - 1$ 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, i+1, ... , n-2] 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)

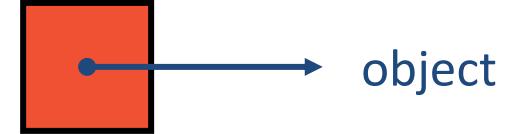
Space complexity:

- 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

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:

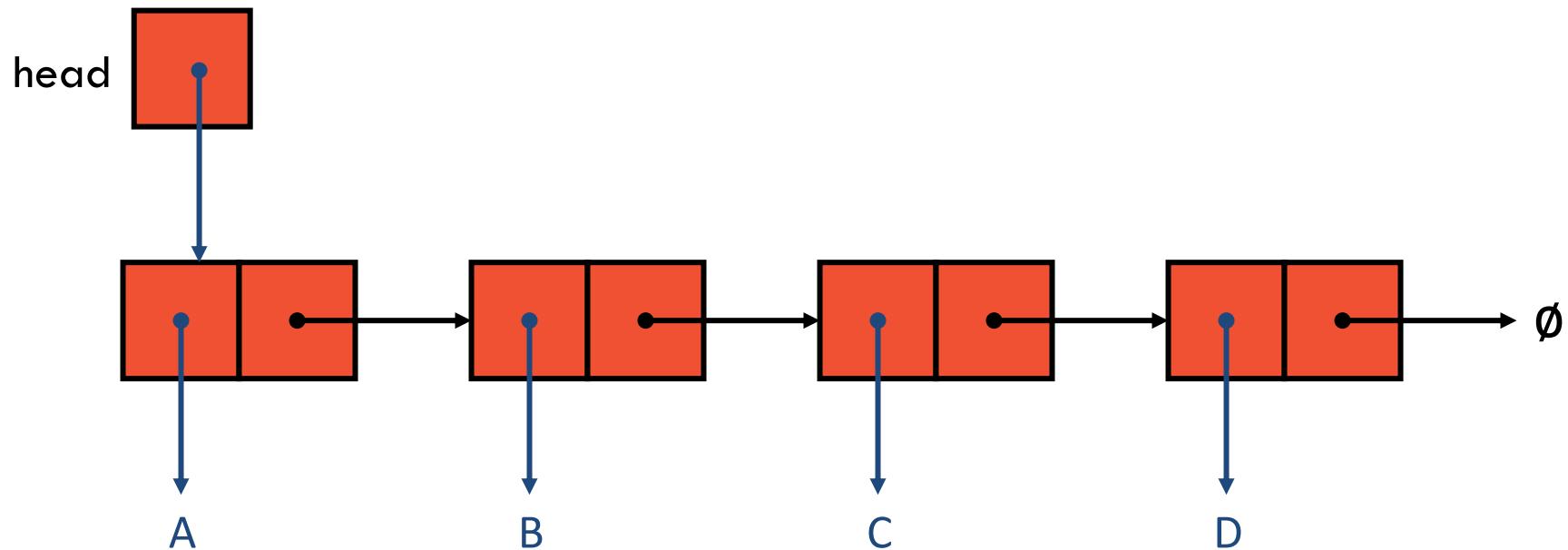
`element()` : return the element stored at the position instance

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

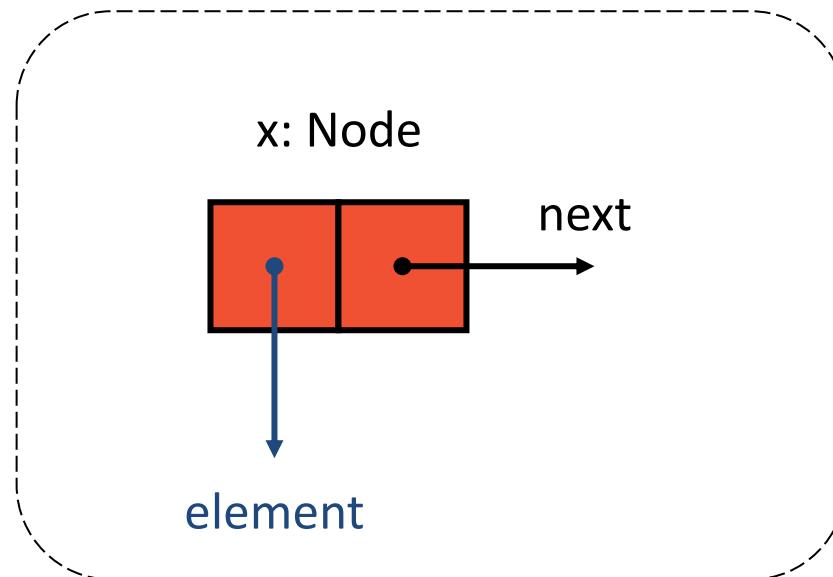
- A concrete data structure
- A sequence of **Nodes**, each with a reference to the next node
- List captured by reference (head) to the first **Node**



Node implements Position

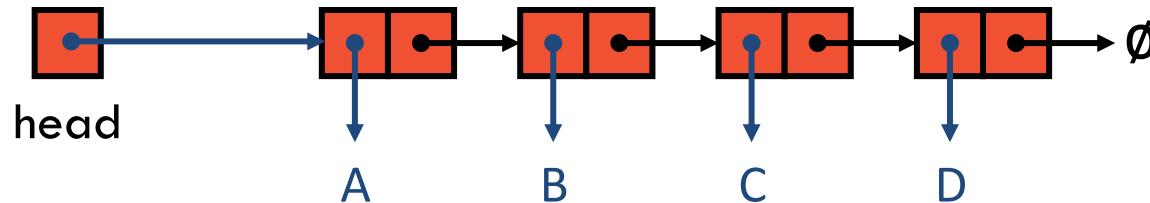
Each **Node** in a singly linked List stores

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



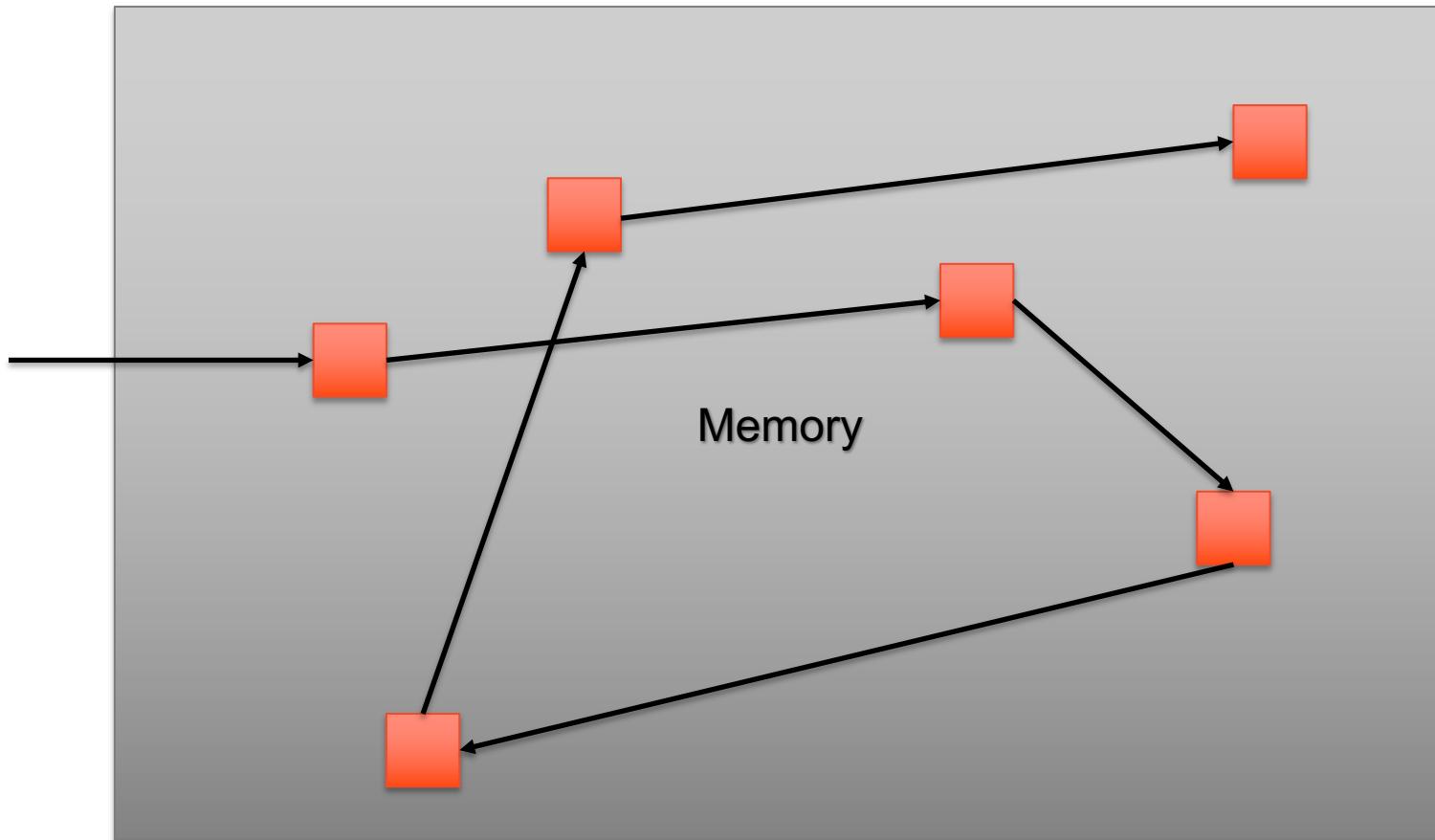
Advice on working with linked structures

- 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**



Linked Lists

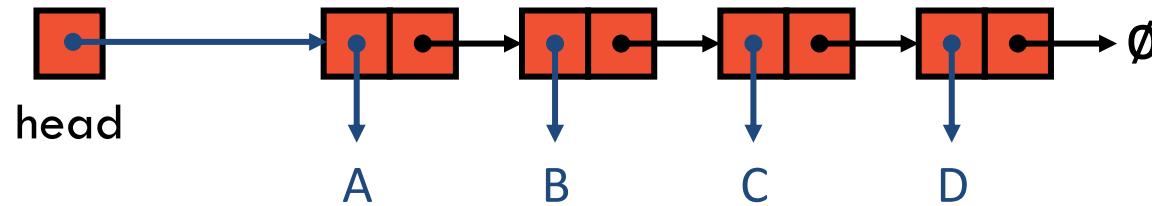
How are linked lists stored?



first()

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

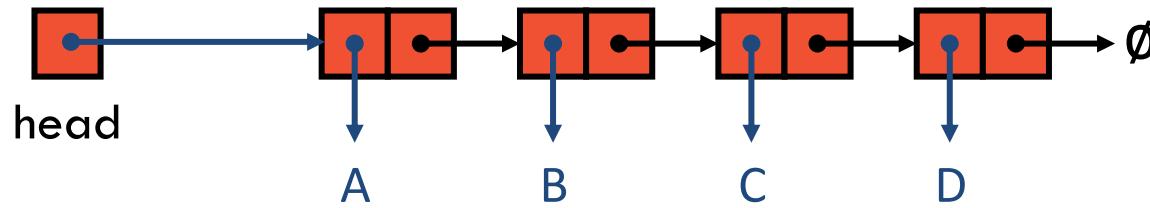
return?



first()

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

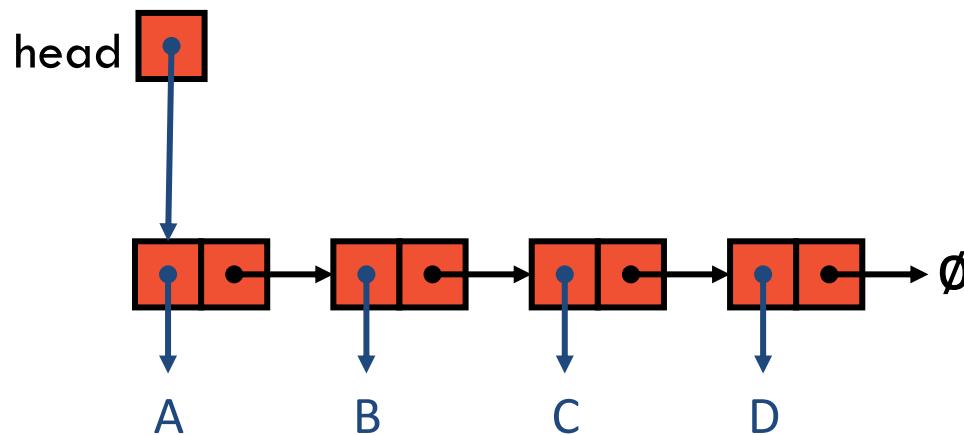
return head



Time complexity?

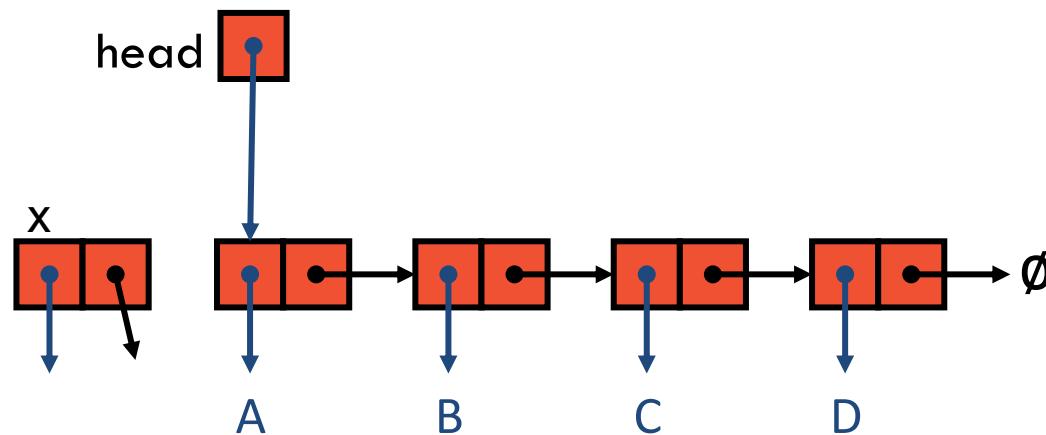
insertFirst(e)

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



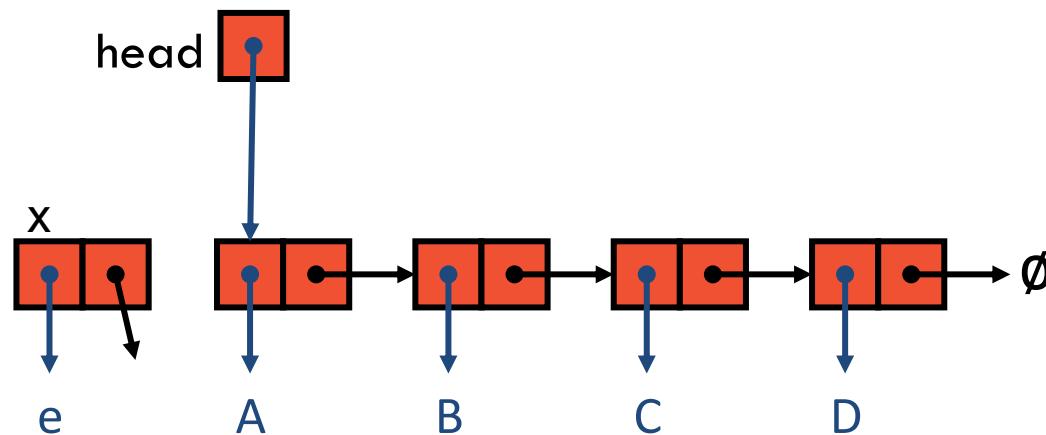
insertFirst(e)

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



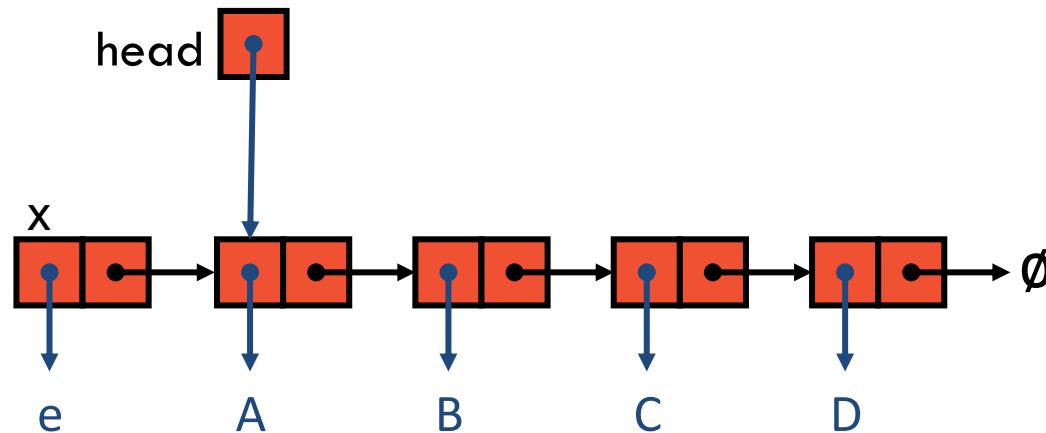
insertFirst(e)

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



insertFirst(e)

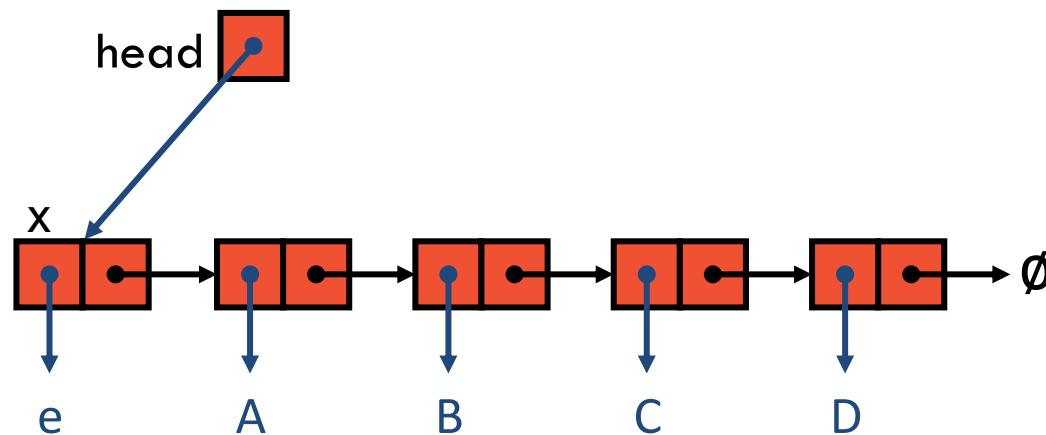
1. Instantiate a new node x
2. Set e as element of x
3. Set $x.next$ to point to (old) head
4. Update list's head to point to x



insertFirst(e)

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

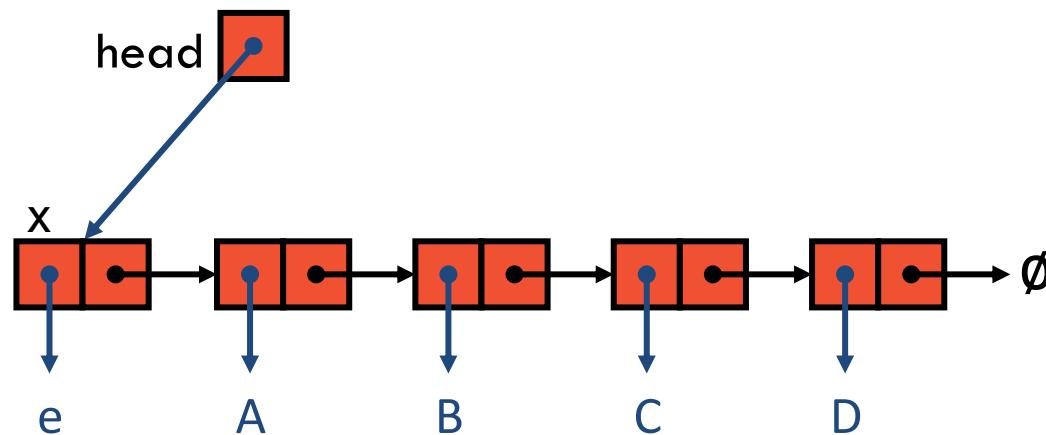
What is the time complexity?



insertFirst(e)

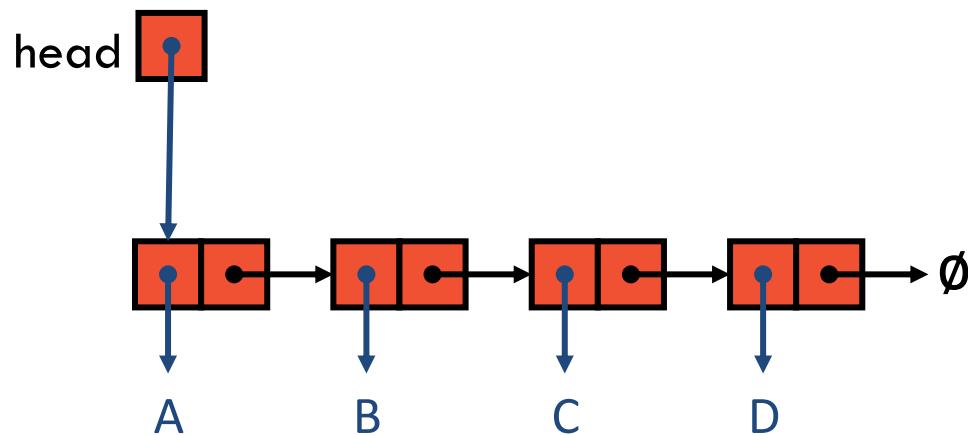
1. Instantiate a new node x
2. Set e as element of x
3. Set $x.next$ to point to (old) head
4. Update list's head to point to x

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



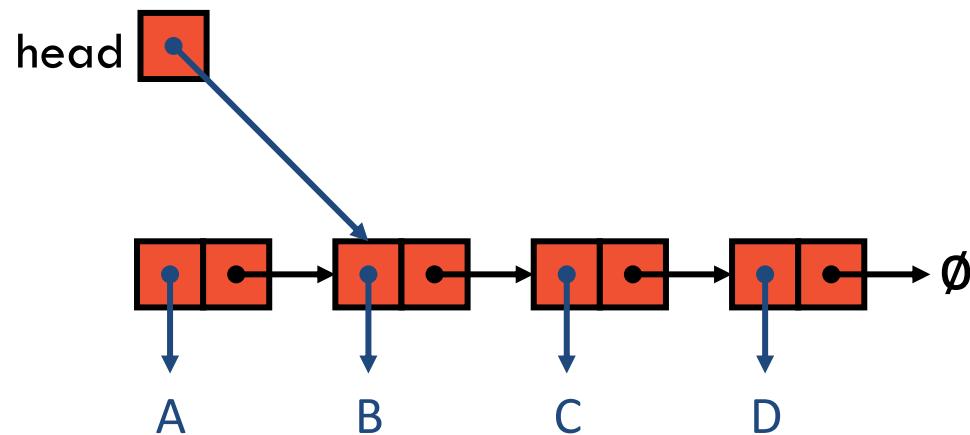
removeFirst()

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



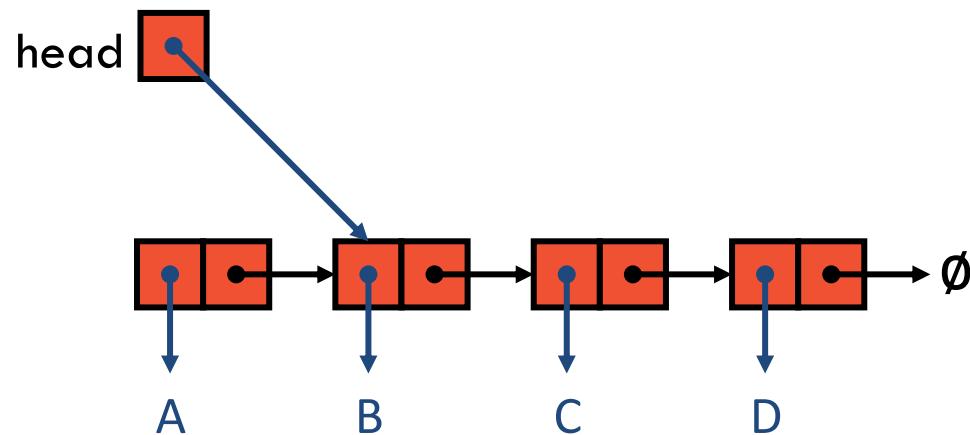
removeFirst()

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



removeFirst()

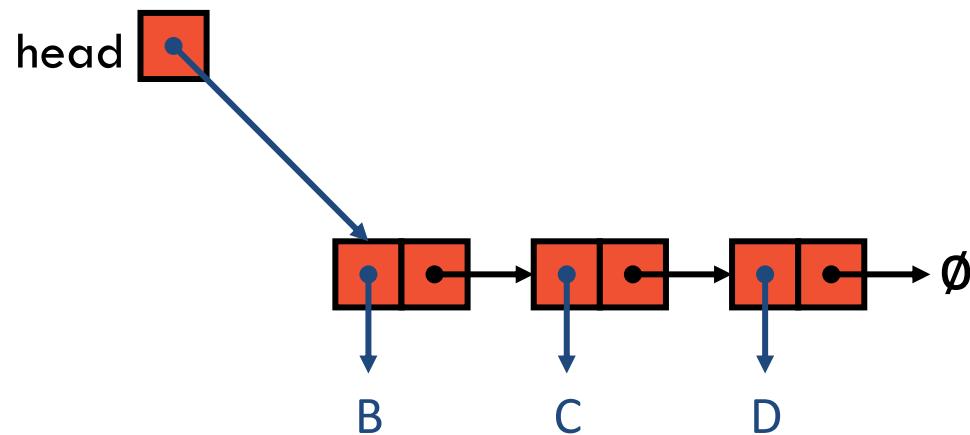
1. Update head to point to next node
2. Delete the former first node



removeFirst()

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

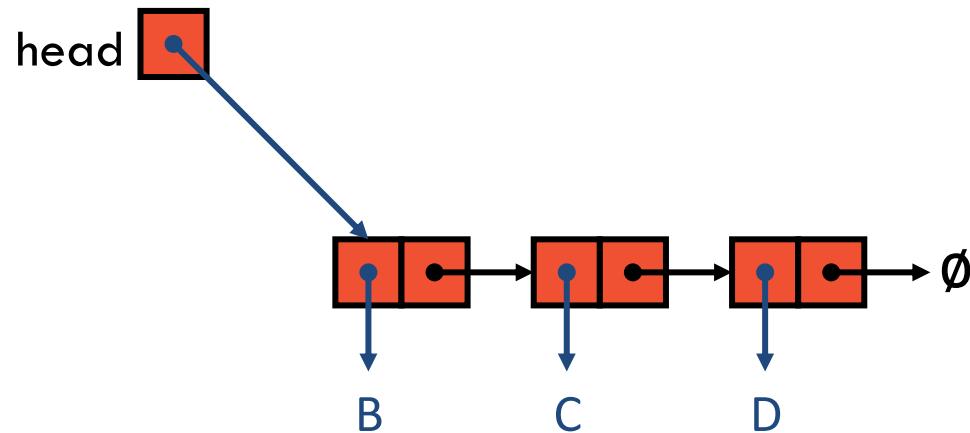
Time complexity?



removeFirst()

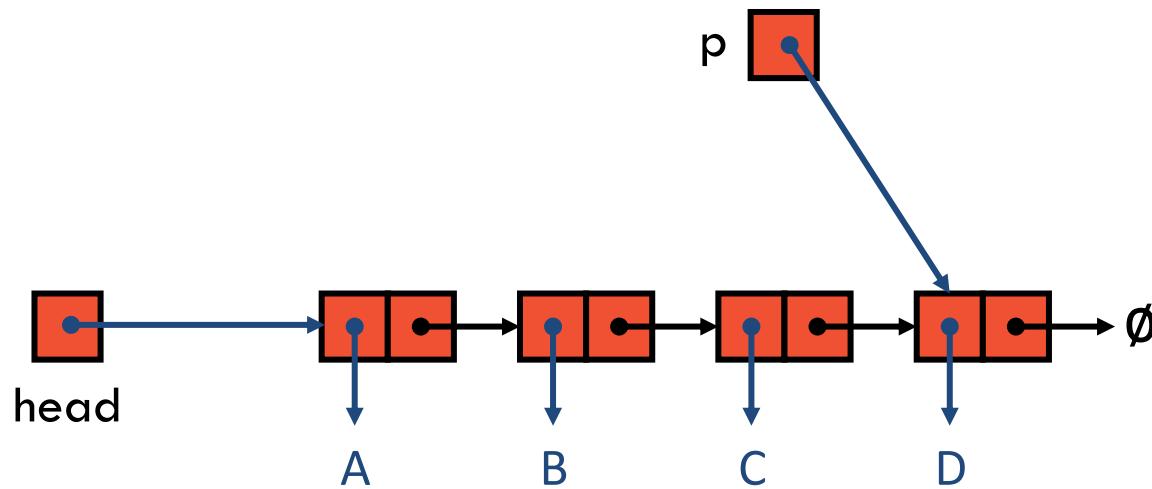
1. Update head to point to next node
2. Delete the former first node

Time complexity? $O(1)$



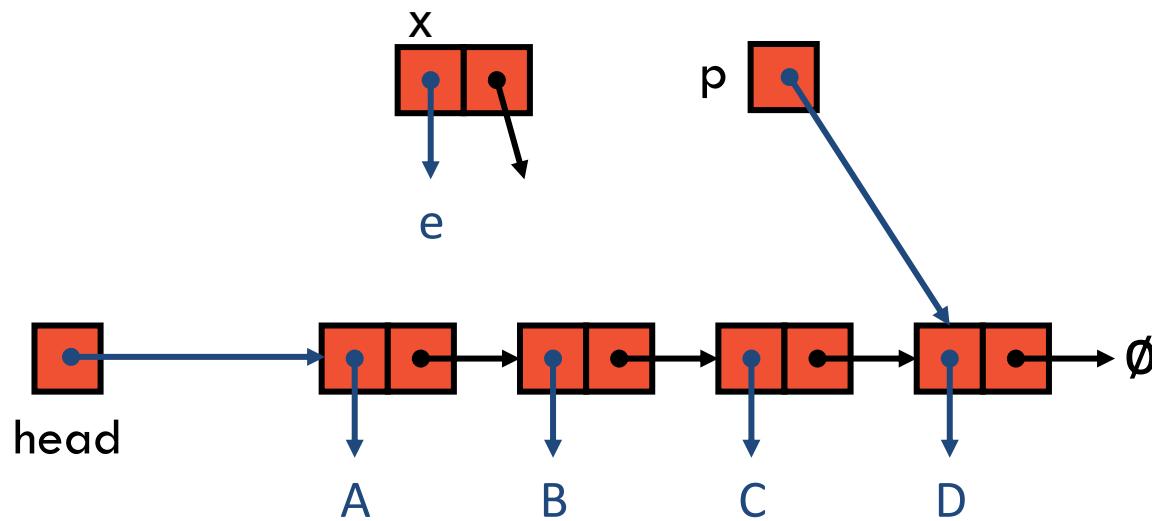
insertBefore(p,e)

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



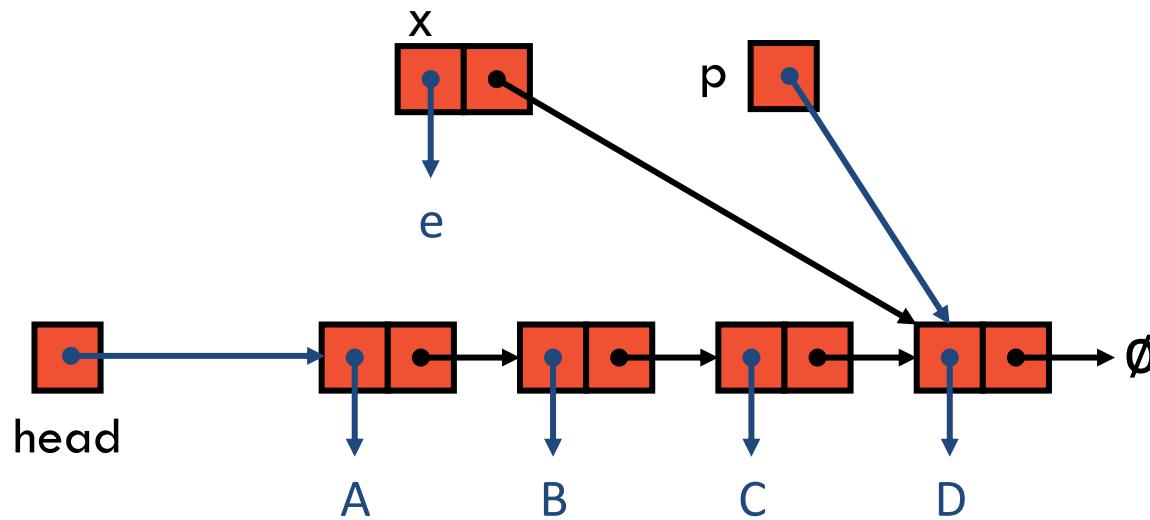
insertBefore(p,e)

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



insertBefore(p,e)

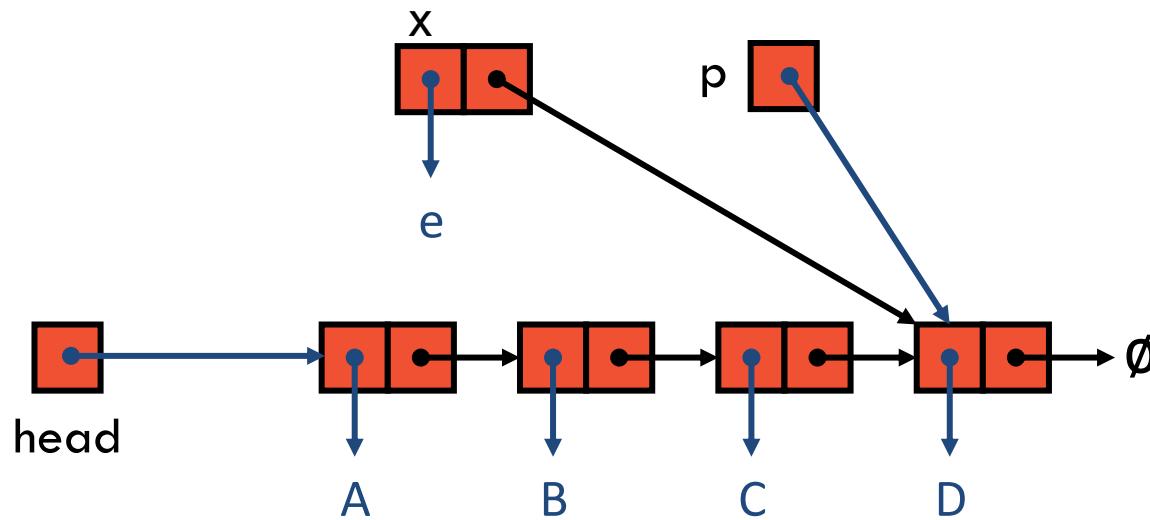
insertBefore(p,e) : insert e in front of the element at position p



What's the next step?

insertBefore(p,e)

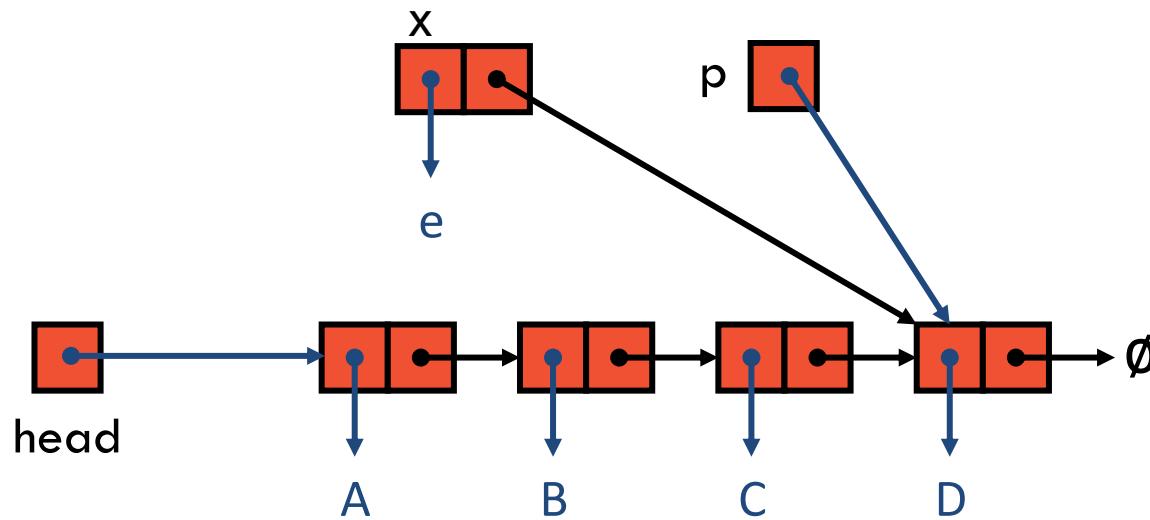
insertBefore(p,e) : insert e in front of the element at position p



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

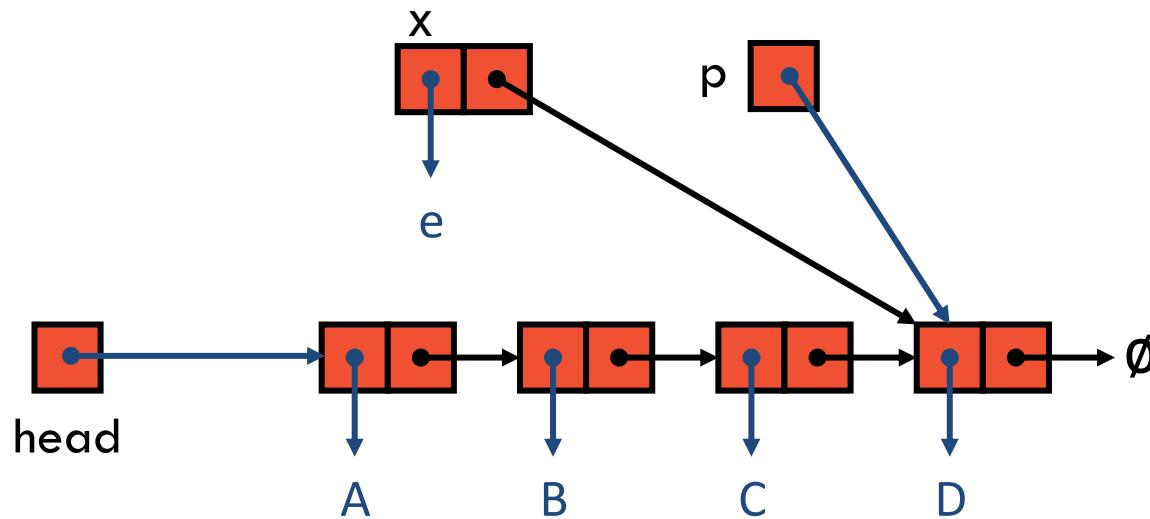
insertBefore(p,e)

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



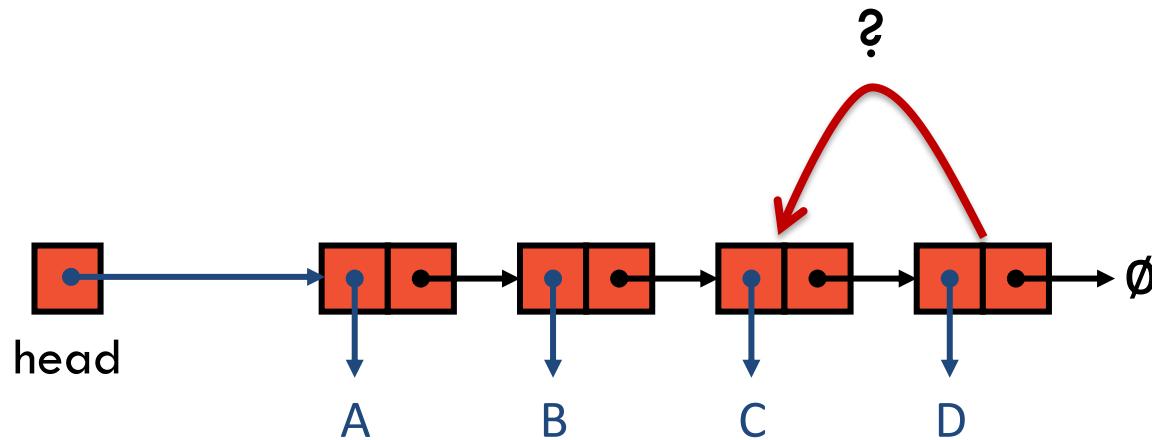
insertBefore(p,e)

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.

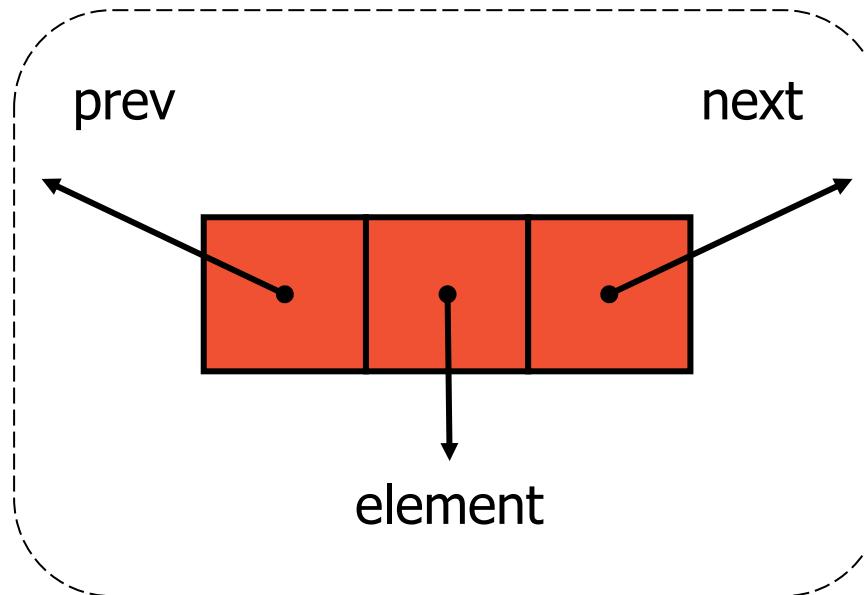


Another attempt

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

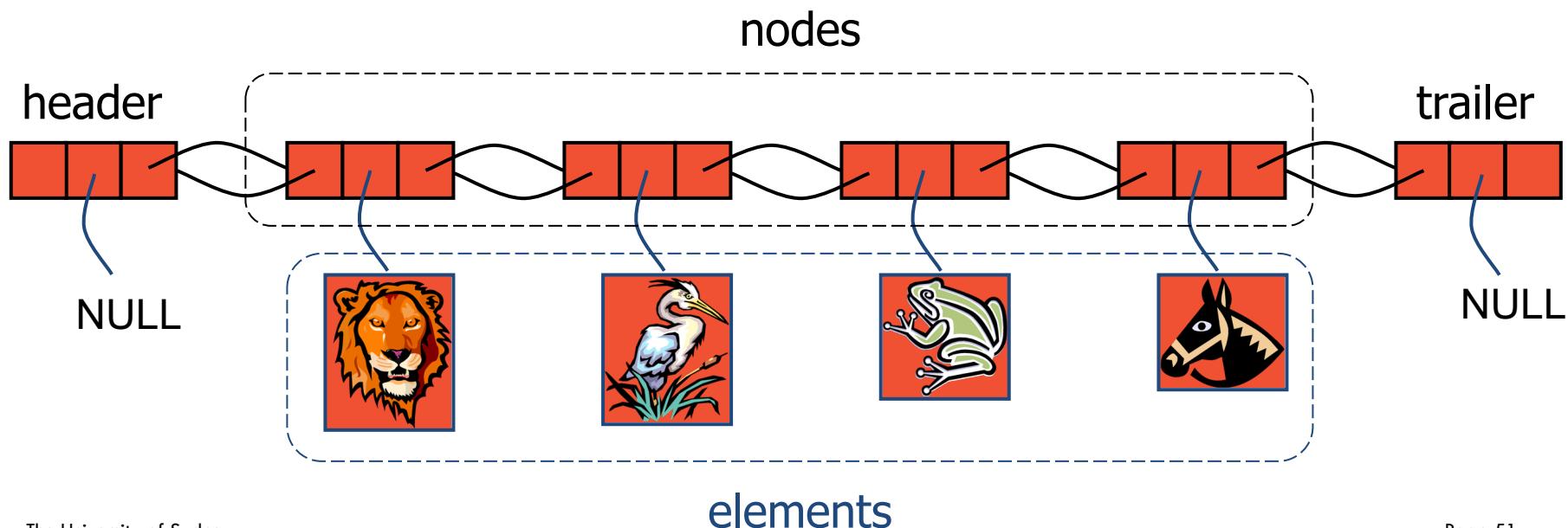
- its element, and
- a link to the previous and next nodes.



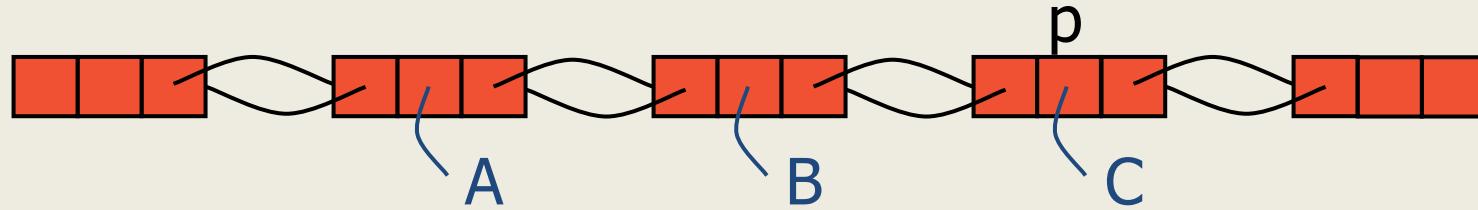
Doubly Linked Lists

A concrete data structure

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

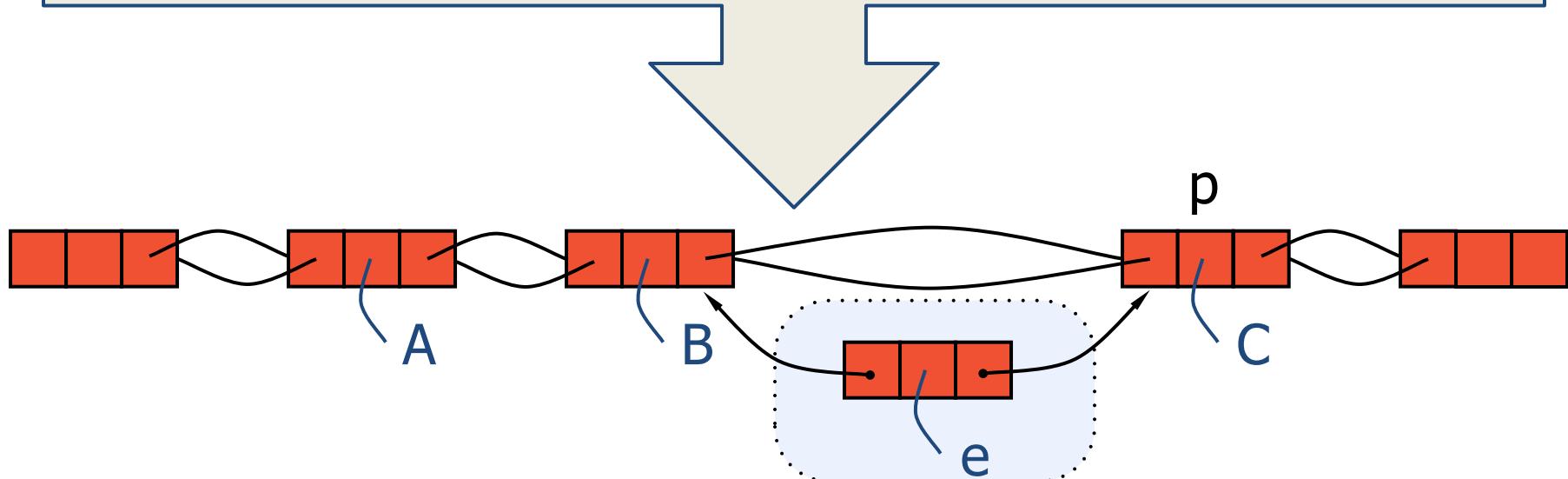


insertBefore(p,e) – step 1

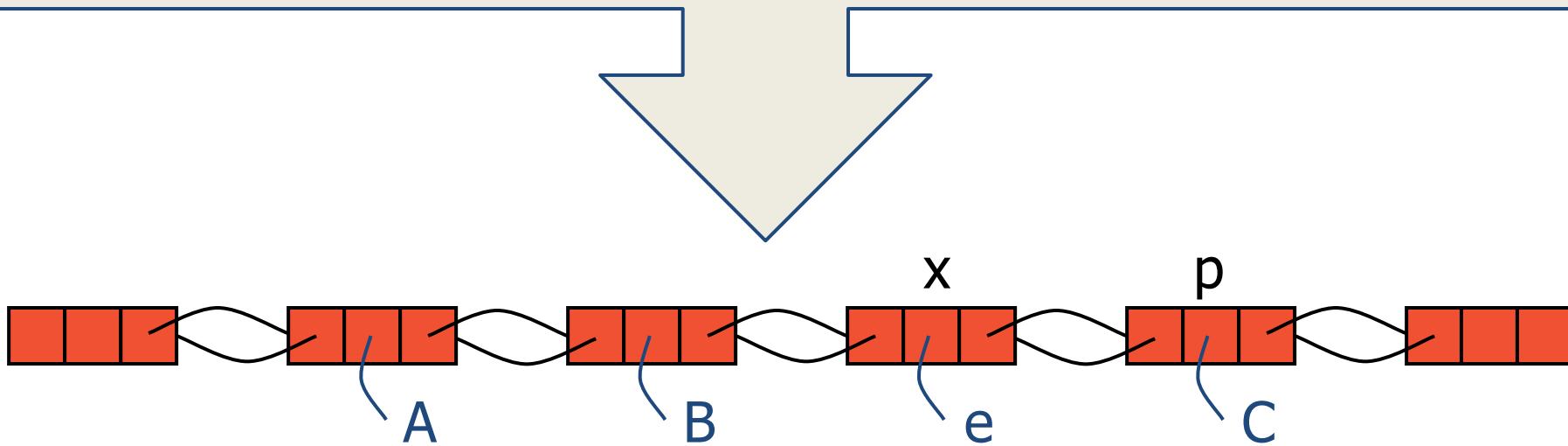
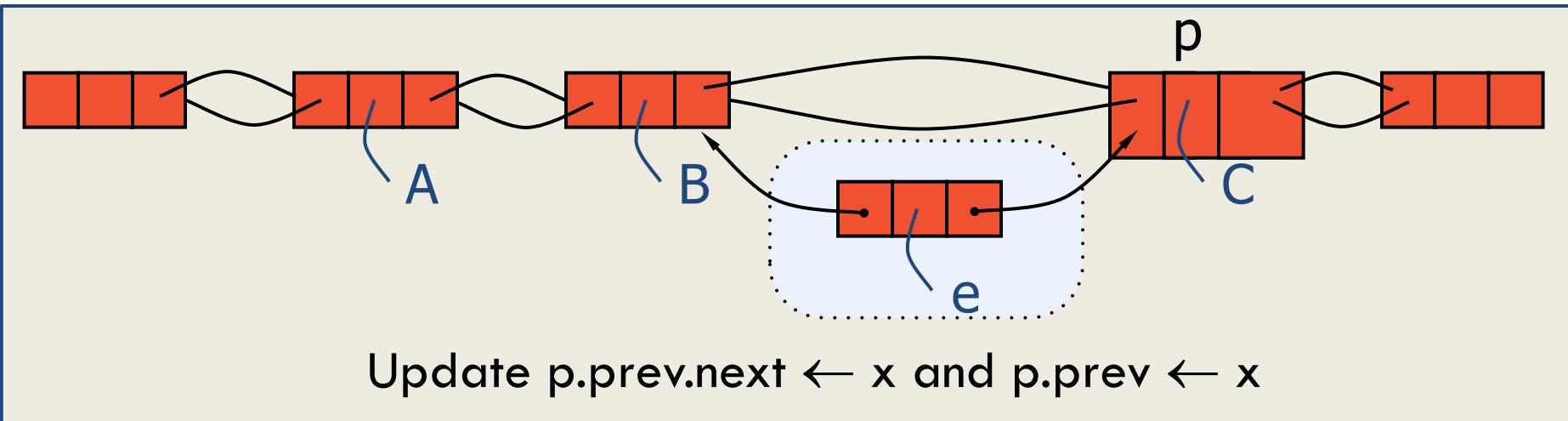


Instantiate new Node x with element set to e.

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

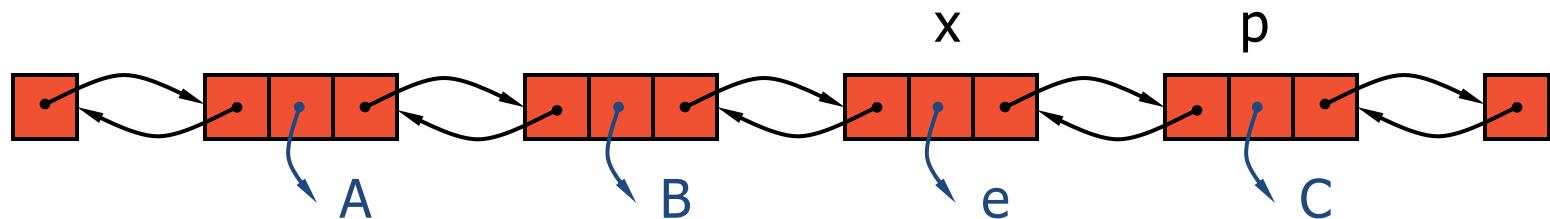
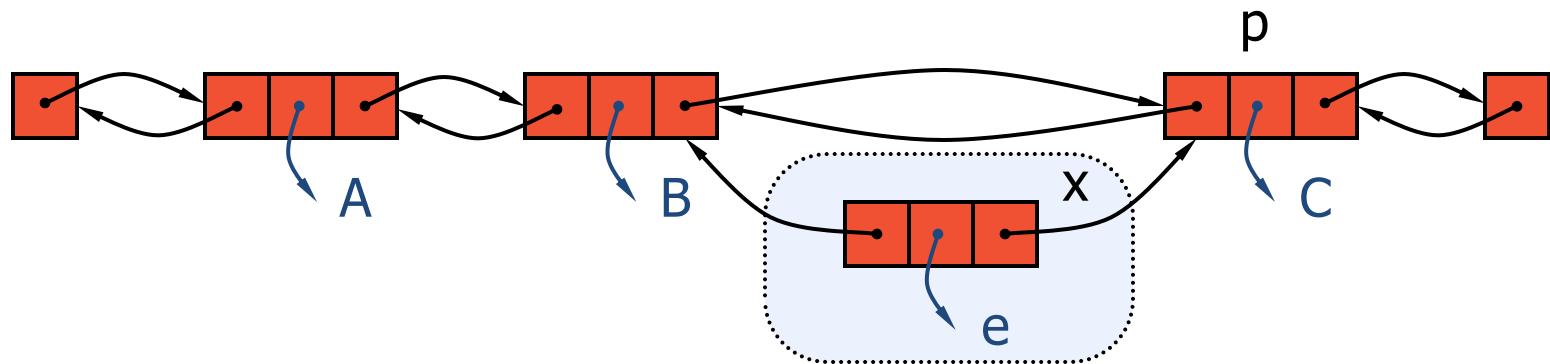
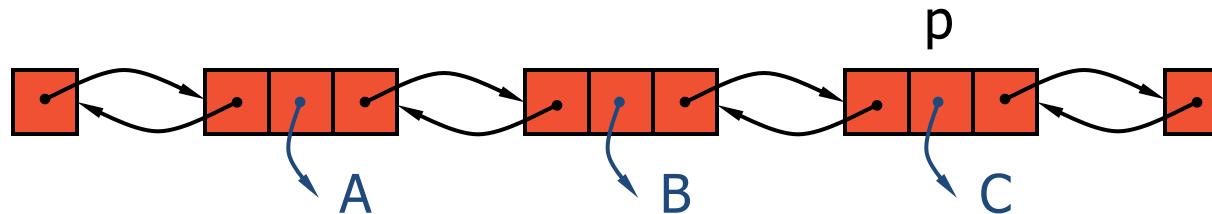


insertBefore(p,e) – step 2



insertBefore(p,e)

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



Pseudo-code

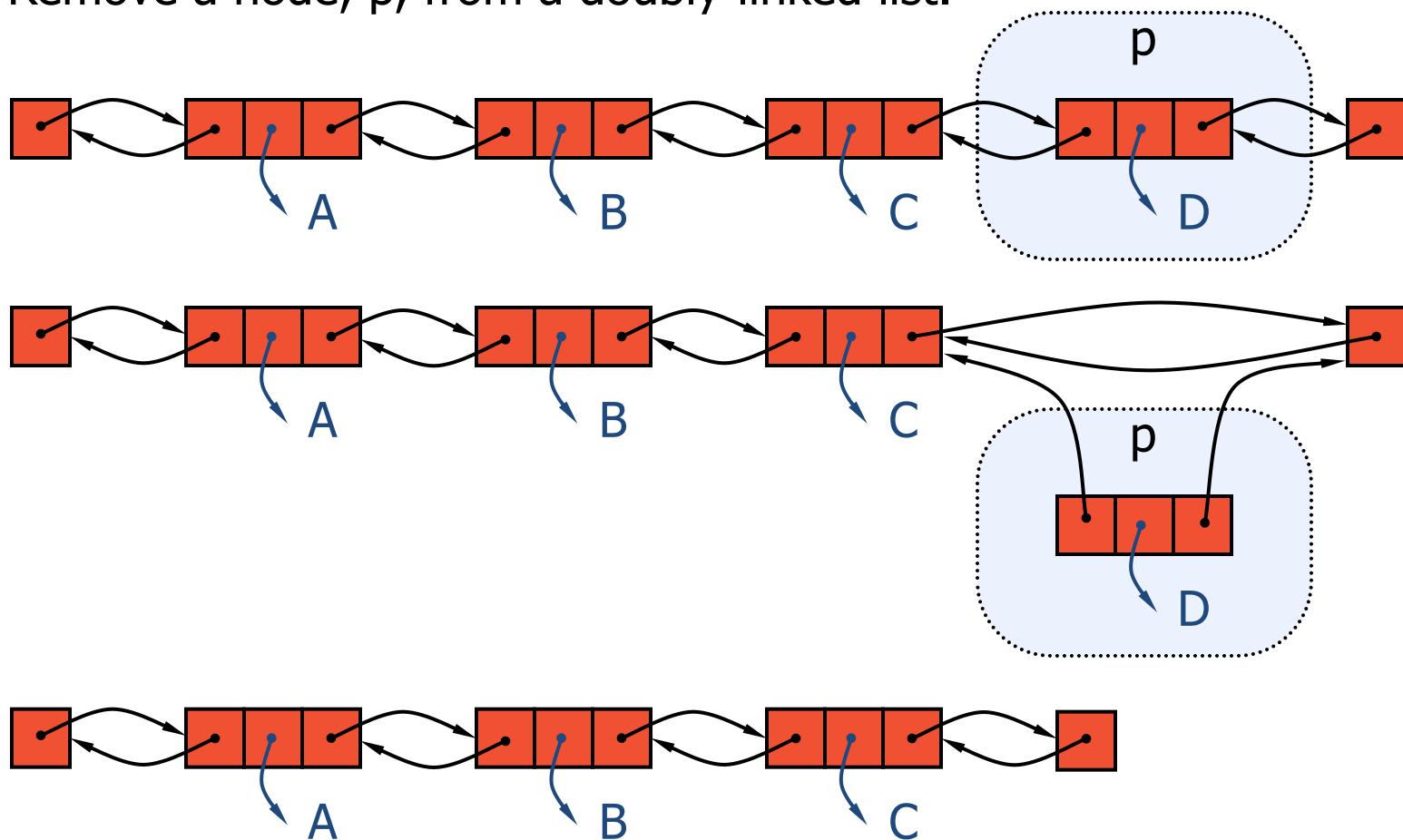
```
def insert_before(pos, elem):
    // insert elem before pos
    // assuming it is a legal pos

    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 (doubly) 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 obj:  
    // process x
```

Is equivalent to:

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

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

```
def main()
    i = 5;
    foo(i);
```

```
def foo(j)
    k = j+1;
    bar(k);
```

```
def bar(m)
```

...

```
bar
PC = 1
m = 6
```

```
foo
PC = 2
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

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

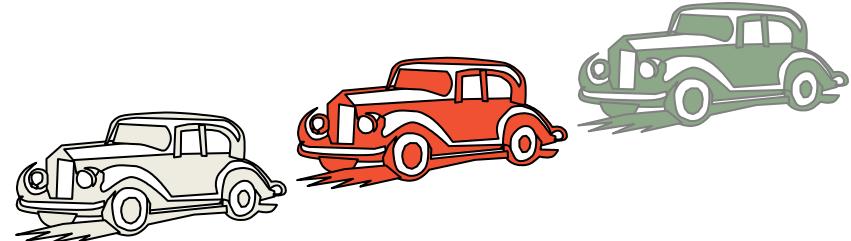
Performance

- 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, which implies each operation runs in $O(1)$ amortized time (still $O(n)$ worst-case).

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	Queue
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

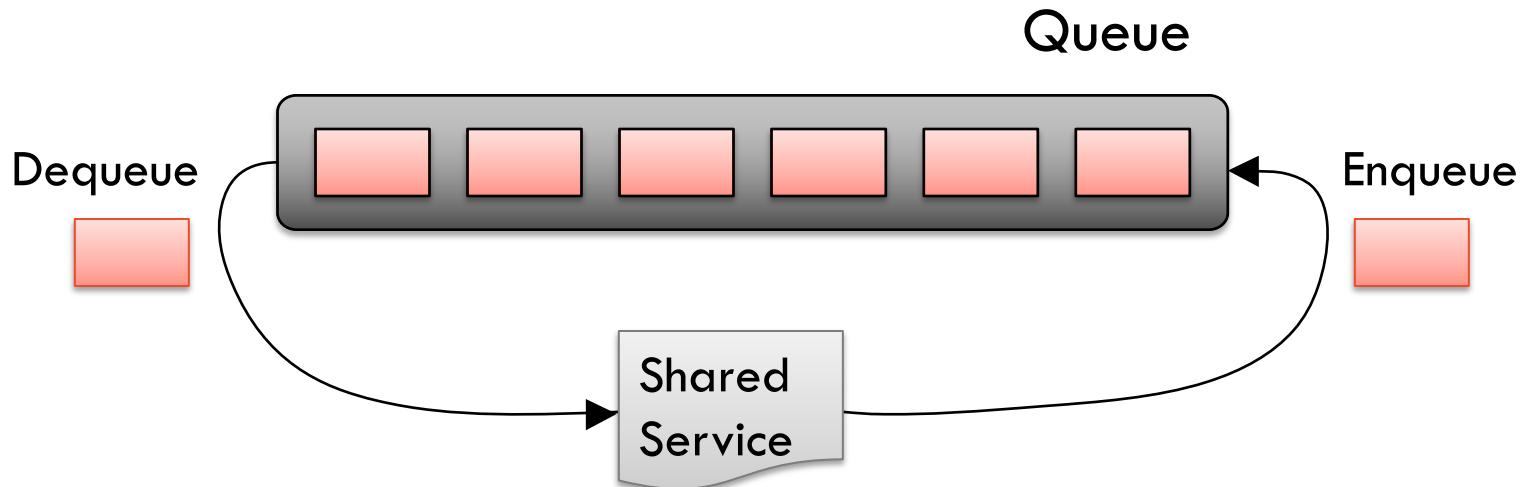
Indirect applications

- 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. $e \leftarrow Q.\text{dequeue}()$
2. Service element e
3. $Q.\text{enqueue}(e)$



Queue implementation based on arrays

Use an array of size N in a circular fashion

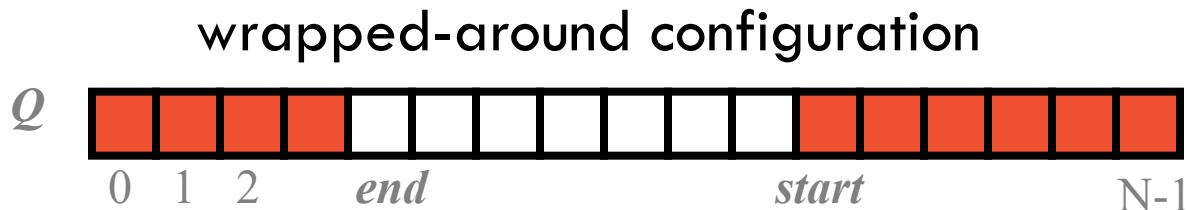
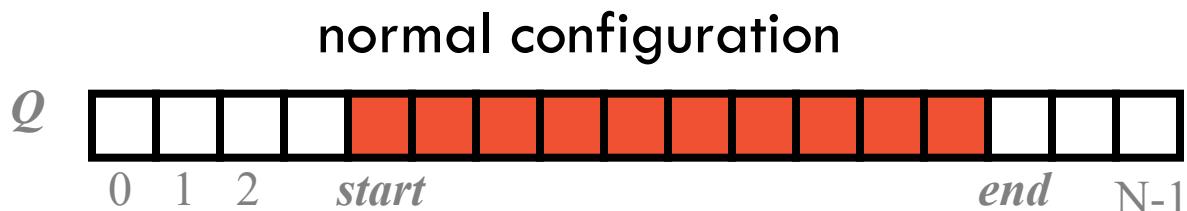
Two variables keep track of the front and size

start : index of the front element

end : index past the last element

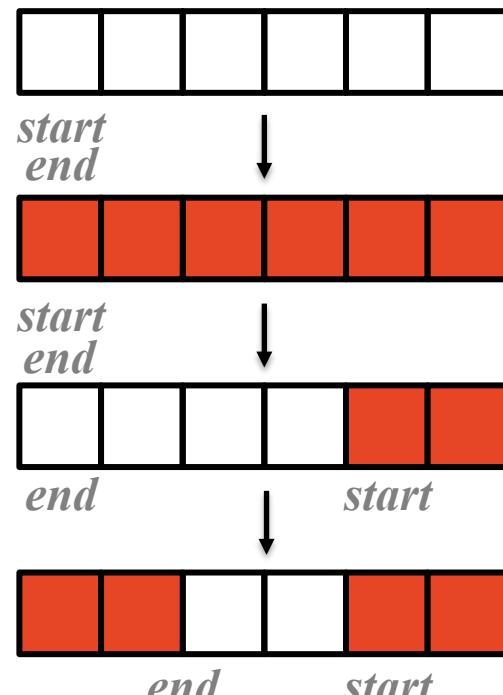
size : number of stored elements

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



How to get in a wrapped-around configuration

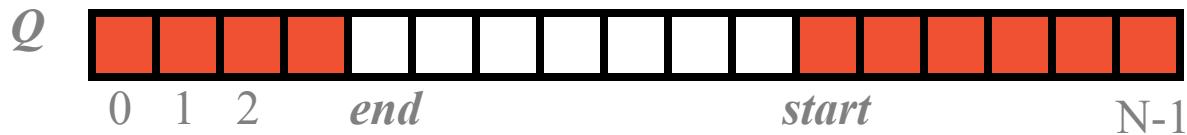
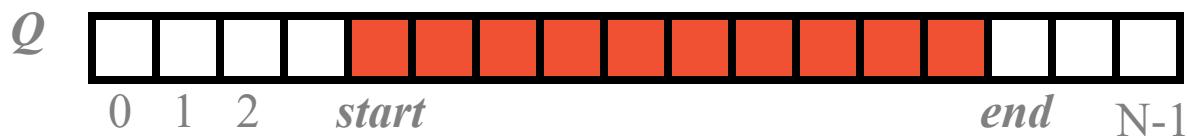
- Enqueue N elements
- Dequeue $k < N$ elements
- Enqueue $k' < k$ elements



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
        end ← (start + size) mod N
        Q[end] ← e
        size ← size + 1
```

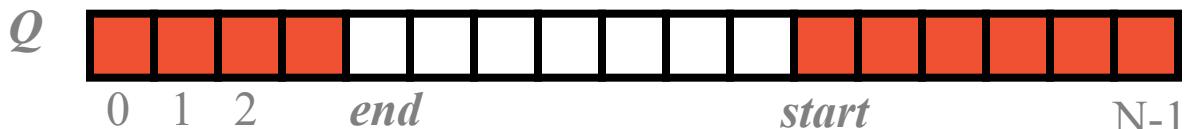
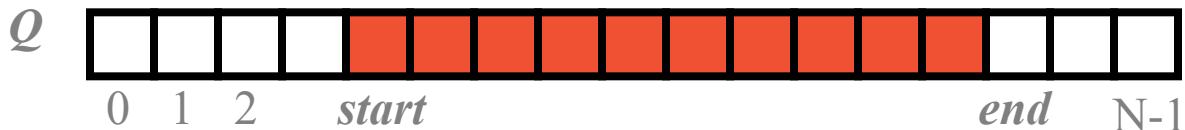


Queue Operations: Dequeue

Note that operation dequeue returns error if the queue is empty

One could alternatively return null

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



Queue implementation based on arrays

Performance

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

This week

Tutorial Sheet 1: Available on Ed

Quiz 1: Available on Canvas at the end of this lecture

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.

Data structures and Algorithms

Lecture 3: Trees [GT 2.3]

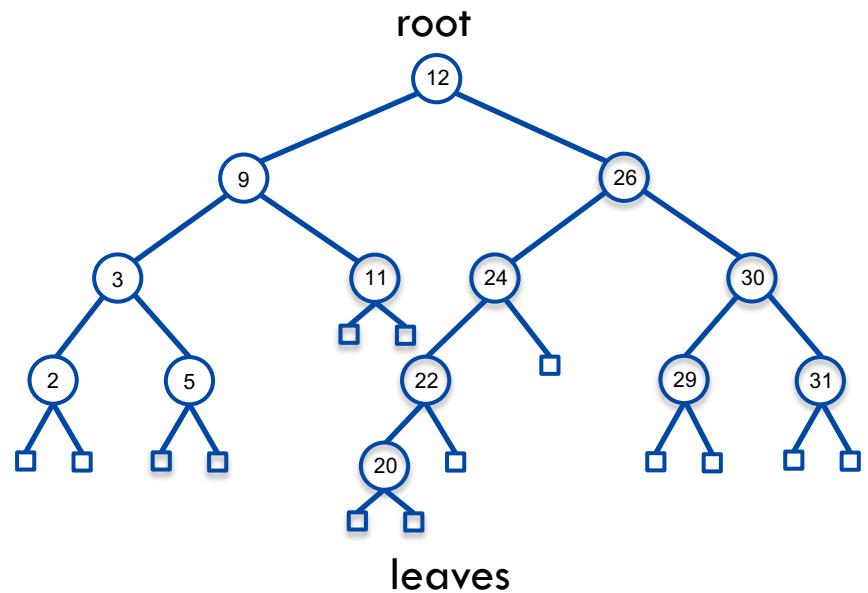
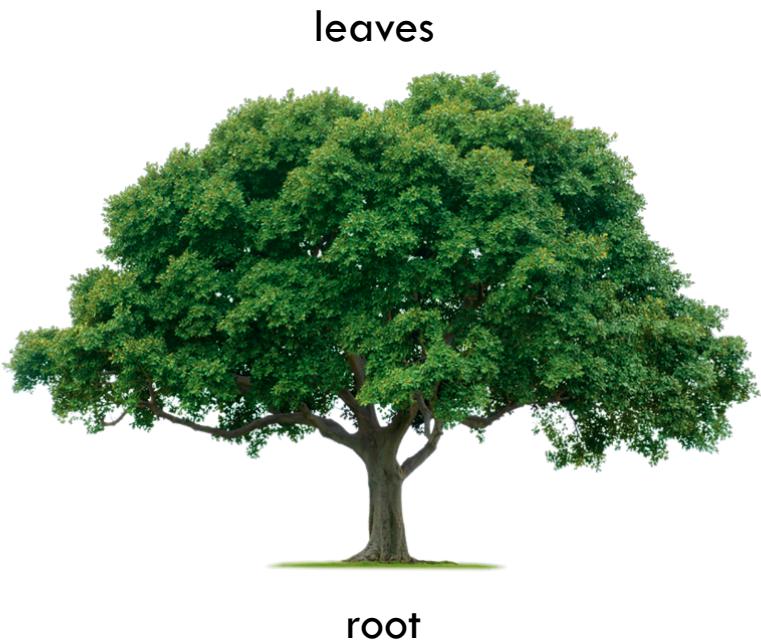
Dr. André van Renssen
School of Computer Science

*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

Trees

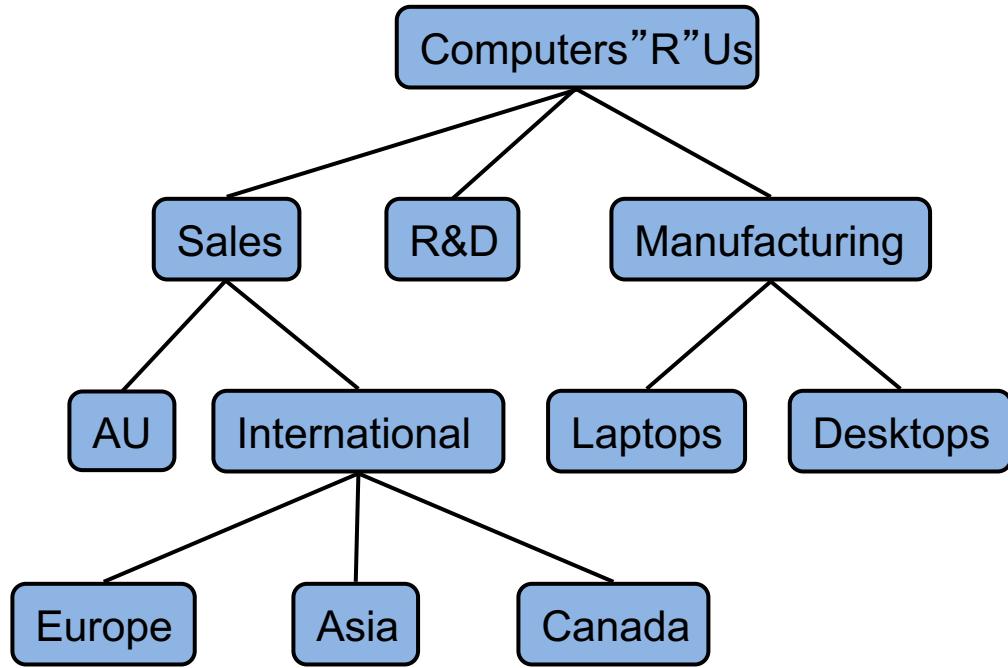


What is a Tree

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



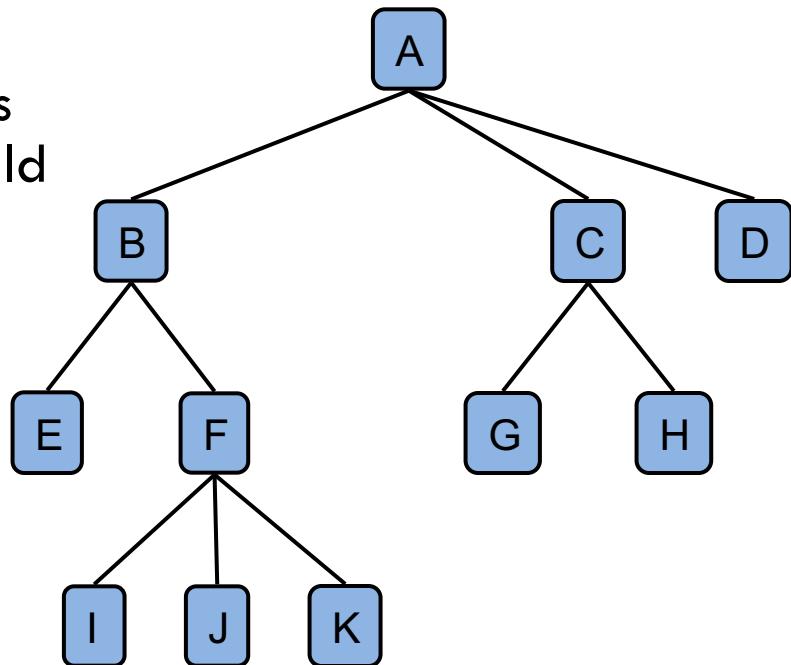
Applications:

- 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:

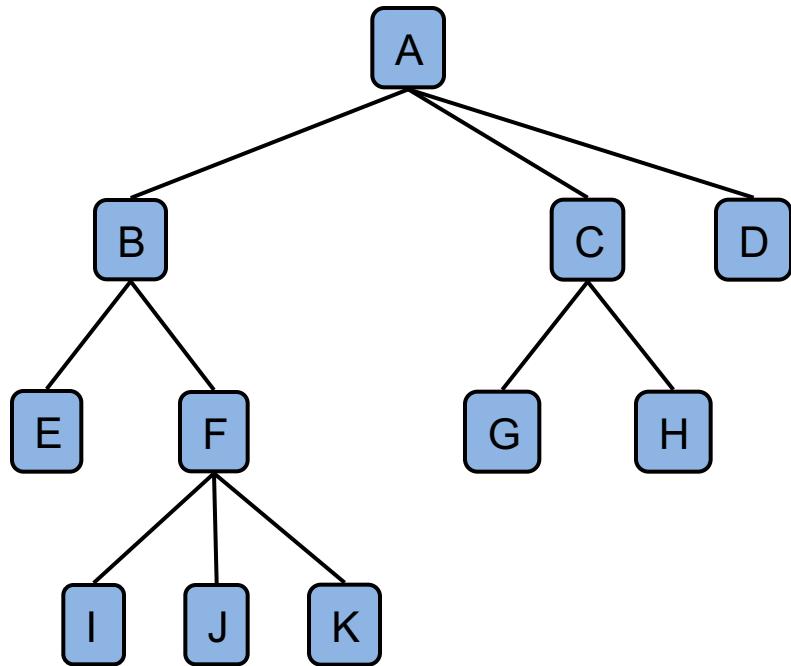
- If **T** is non-empty, it has a special node called the **root** that has no parent
- Every node **v** of **T** other than the root has a unique **parent**
- Following the parent relation always leads to the root (i.e., the parent-child relation does not have “cycles”)



Tree Terminology

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

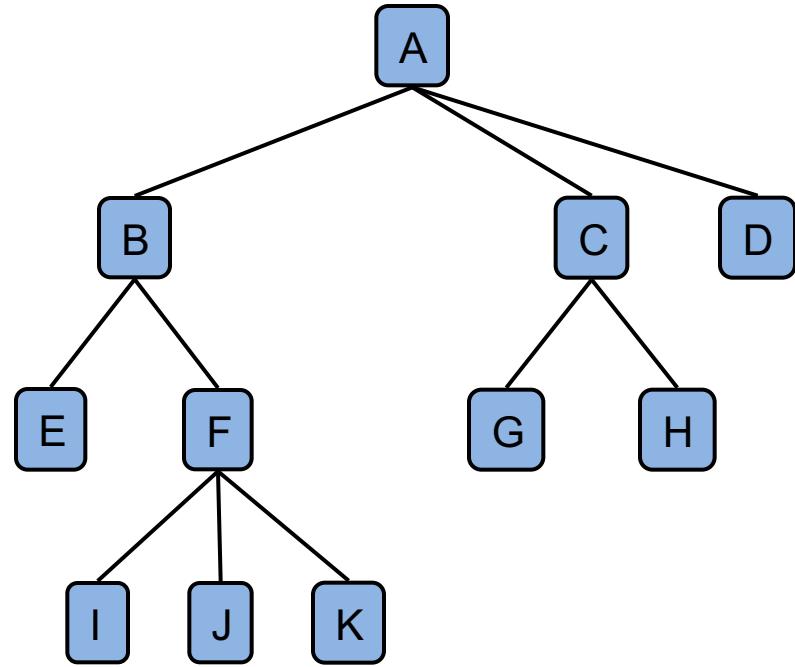
- **Root:** node without parent (e.g., A)
- **Internal node:** node with at least one child (e.g., A, B, C, F)
- **External/leaf node:** node without children (e.g., E, I, J, K, G, H, D)



Tree Terminology

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

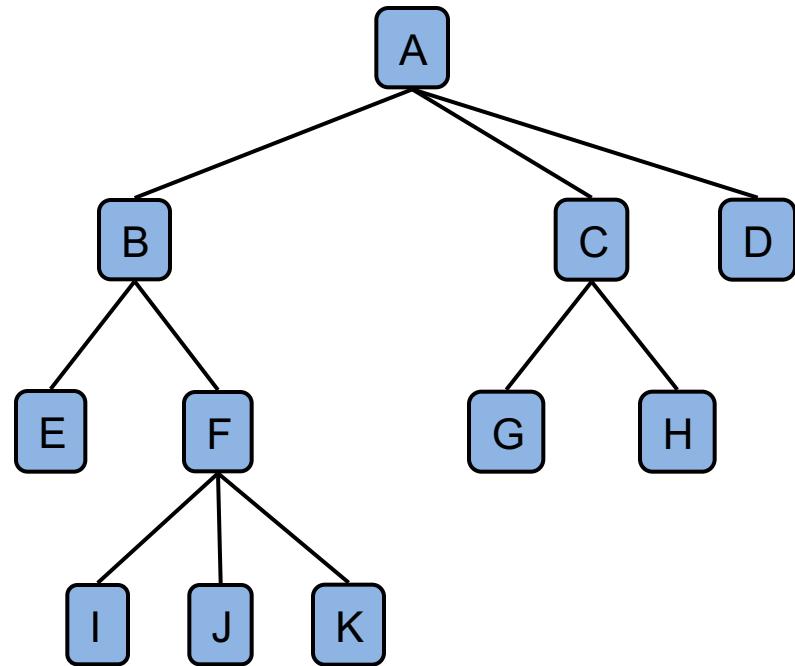
- **Ancestors:** parent, grandparent, great-grandparent, etc. (e.g., ancestors of F are A, B)
- **Descendants:** child, grandchild, great-grandchild, etc. (e.g., descendants of B are E, F, I, J, K)
- Two nodes with the same parent are **siblings** (e.g., B and D)



Tree Terminology

More fine-grained location concepts:

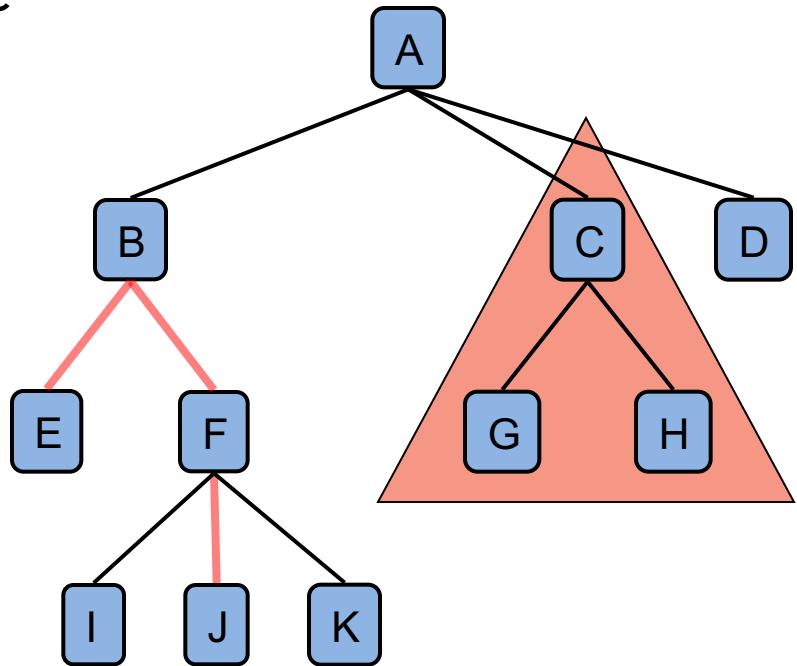
- **Depth of a node:** number of ancestors not including itself
(e.g., $\text{depth}(F) = 2$)
- **Level:** set of nodes with given depth
(e.g., $\{E, F, G, H\}$ are level 2)
- **Height of a tree:** maximum depth
(e.g., 3)



Tree Terminology

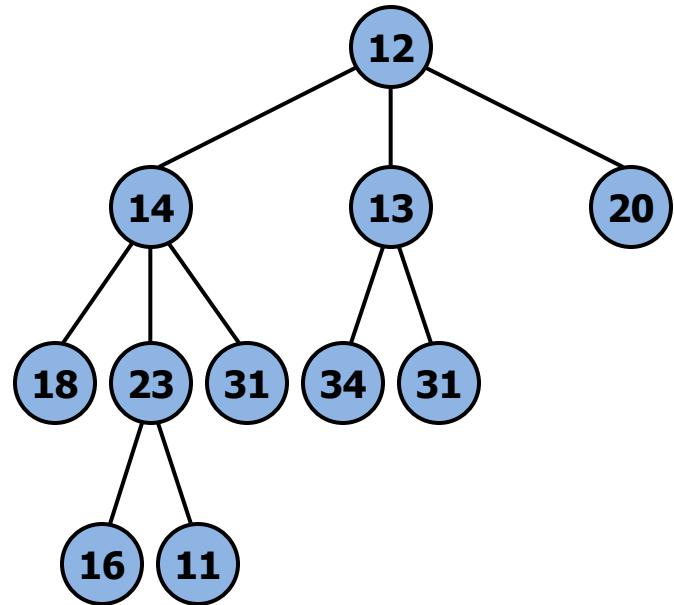
Substructures of a tree:

- **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., $\langle E, B, F, J \rangle$).



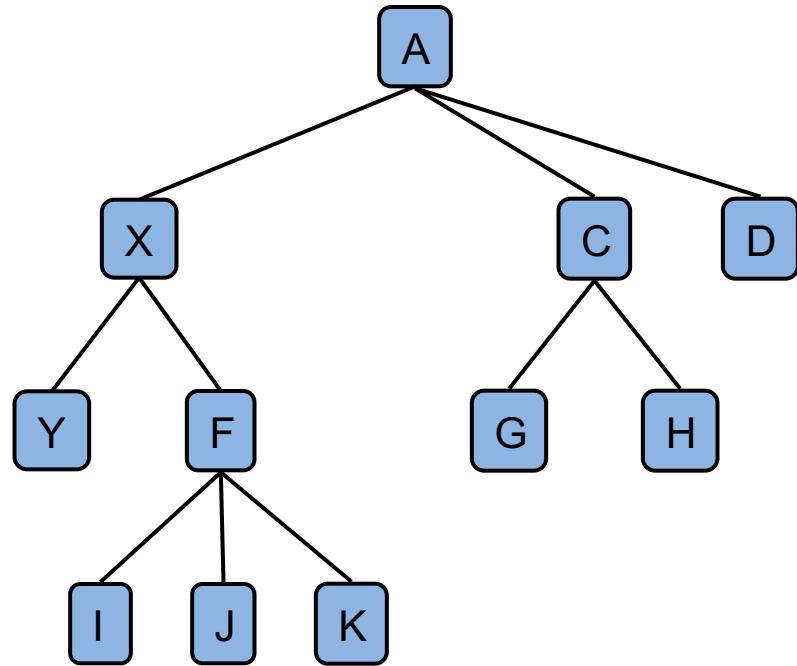
Examples

- 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



Tree facts

- If node X is an ancestor of node Y, then Y is a descendant of X.
- Ancestor/descendant relations are transitive
- Every node is a descendant of the root
- There may be nodes where neither is an ancestor of the other
- Every pair of nodes has at least one common ancestor.
- The lowest common ancestor (LCA) of x and y is a node z such that z is the ancestor of x and y and no descendant of z has that property



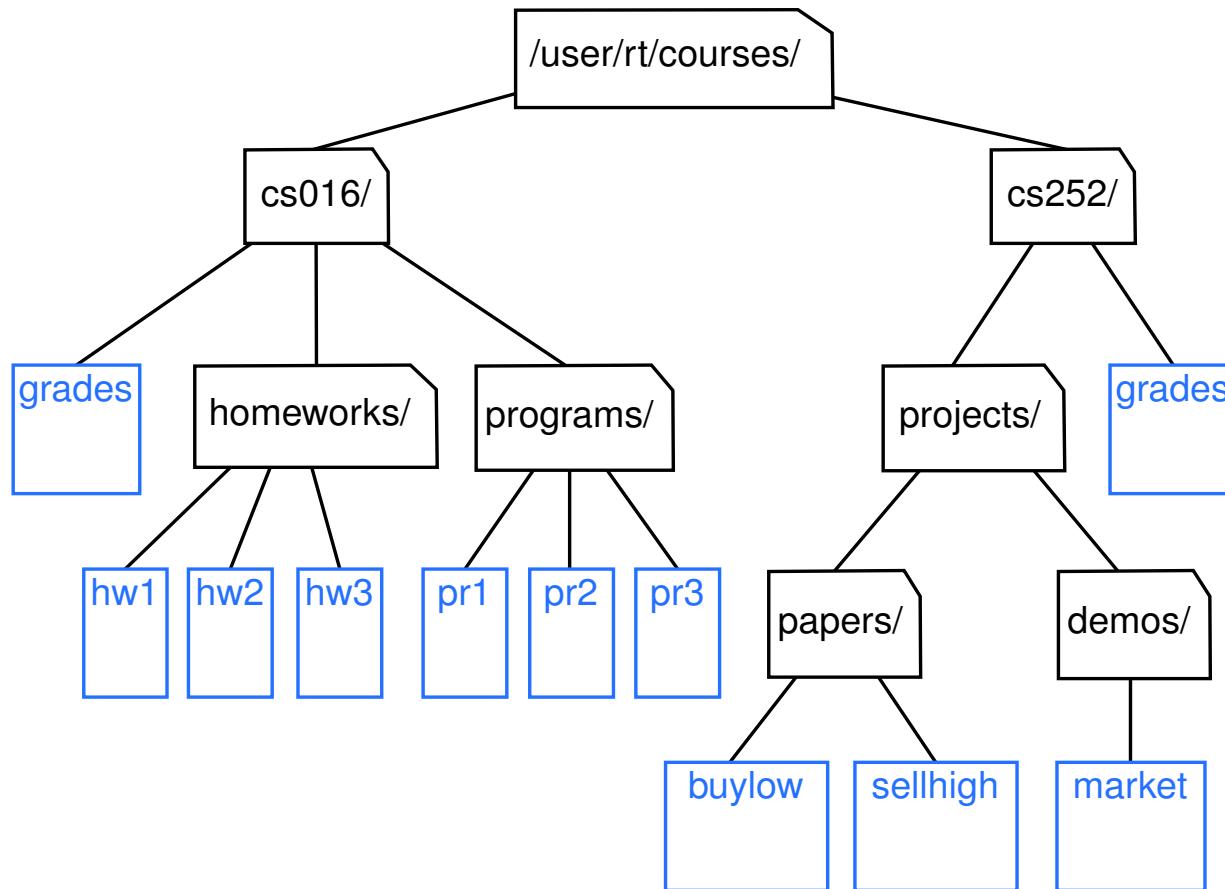
Ordered Trees

Sometimes order of siblings matter

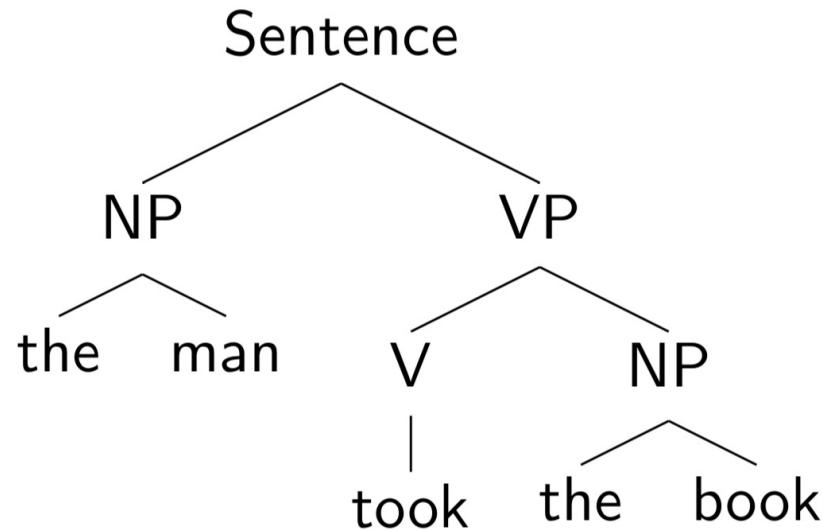
In an **ordered tree** there is a prescribed order for each node's children

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

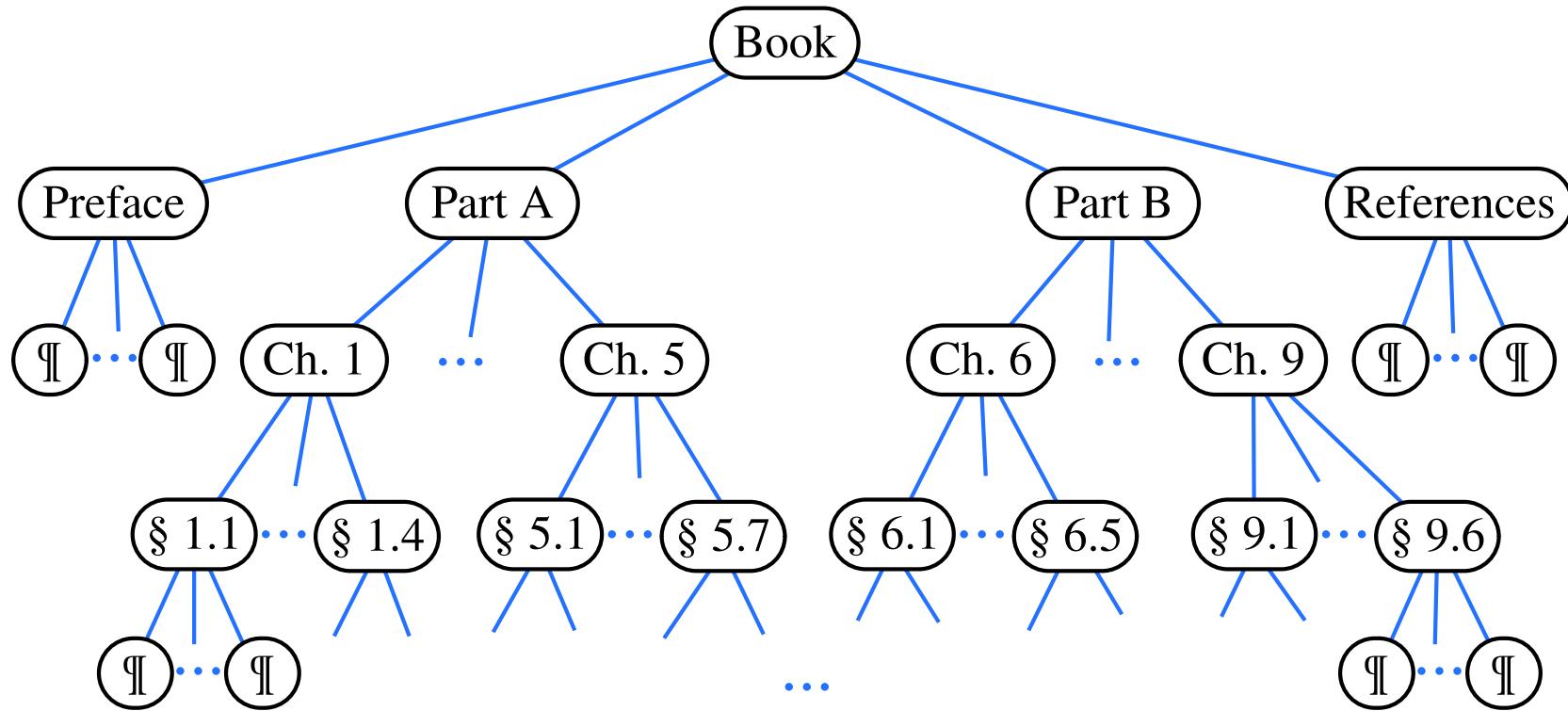
Application: OS file structure



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_external(v)
    # test if v is a leaf
    return v.children.is_empty()
```

```
def is_root(v)
    # test if v is the root
    return v.parent = null
```

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

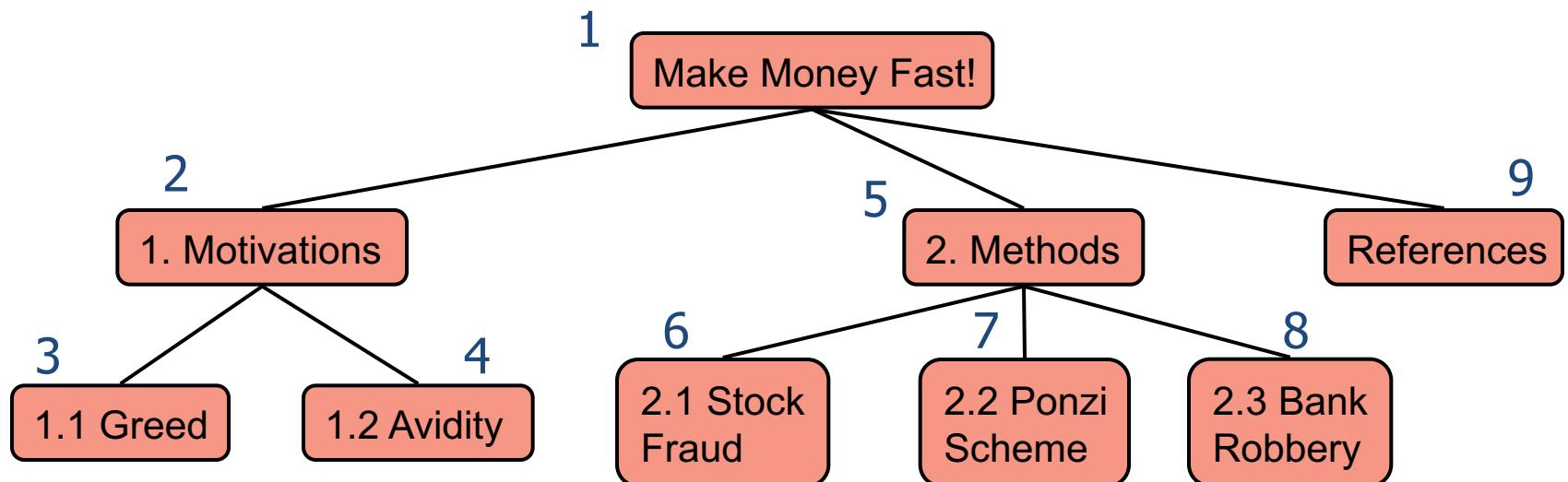
Visit does some work on the node:

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

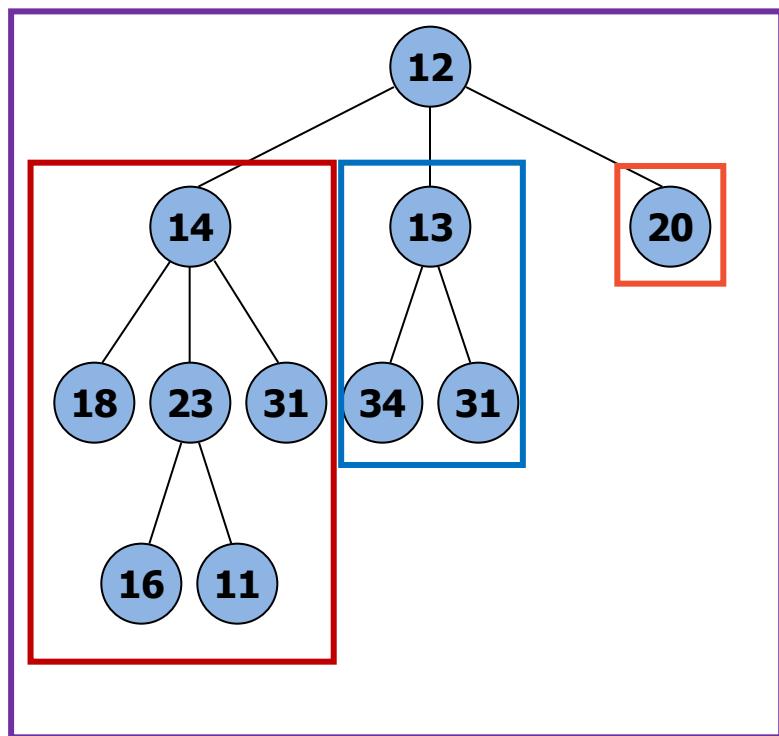
Preorder Traversal Example

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

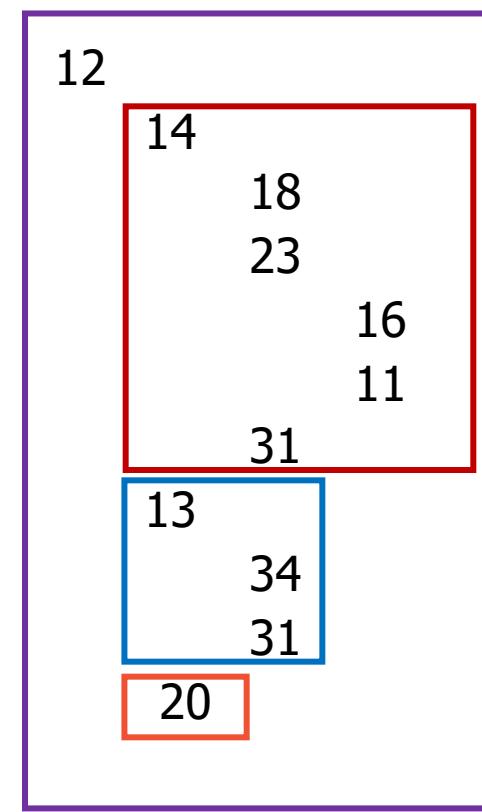
```
def pre_order(v)
    visit(v)
    for each child w of v
        pre_order(w)
```



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
        post_order(w)
    visit(v)
```

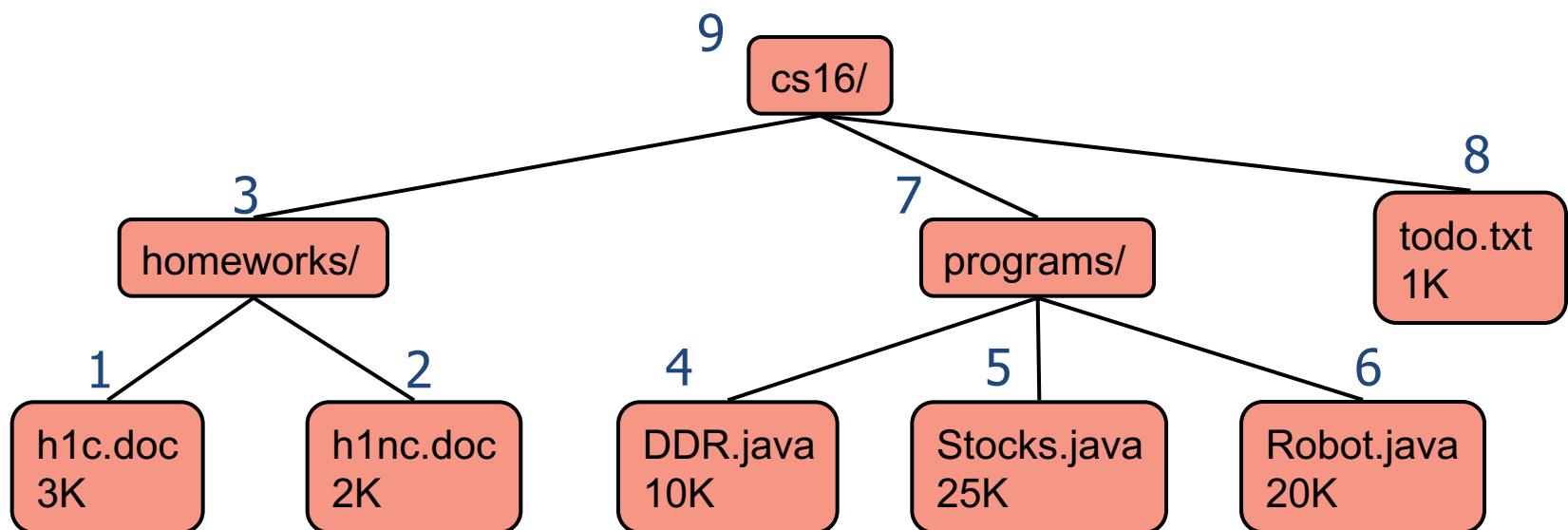
Visit does some work on the node:

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

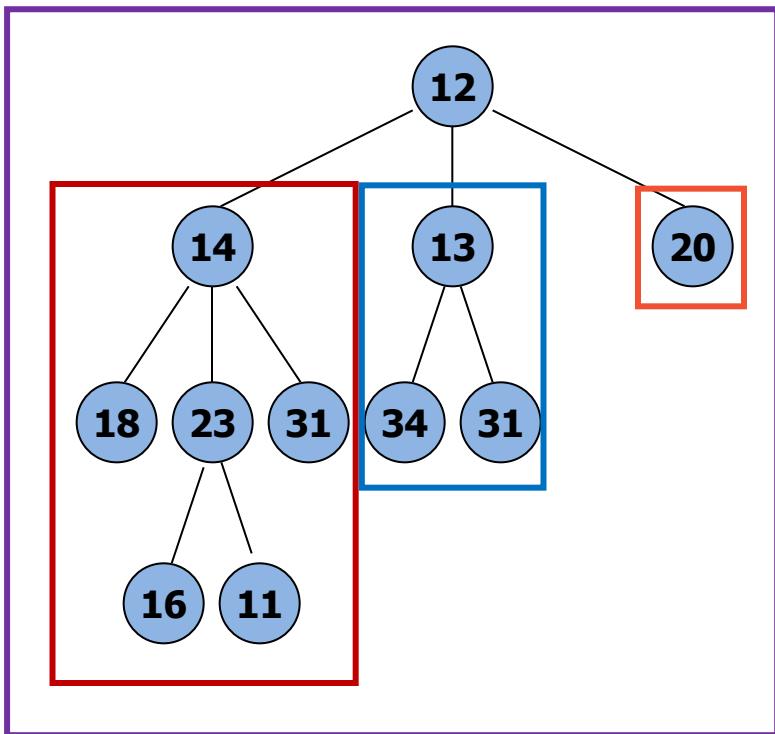
Postorder Traversal

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

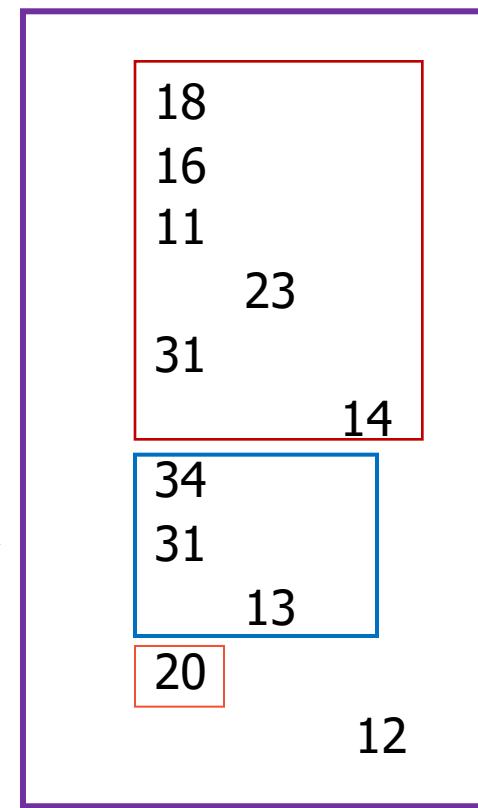
```
def post_order(v)
    for each child w of v
        post_order(w)
    visit(v)
```



Traversing in postorder



visit
order



Postorder
traversal
of subtree

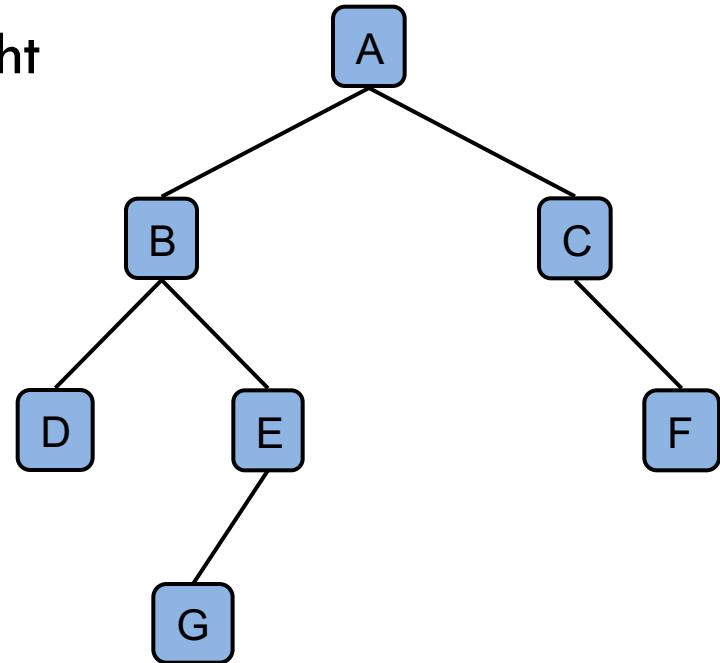
Binary Trees

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

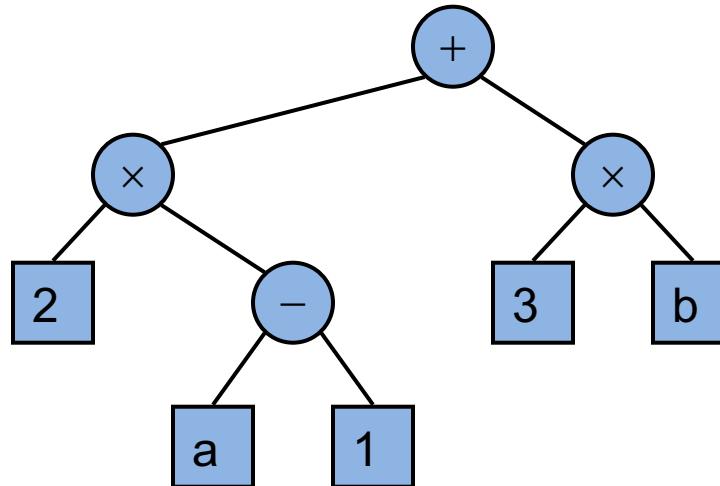


Binary tree application: Arithmetic expression tree

Binary tree associated with an arithmetic expression

- internal nodes: operators
- external nodes: operands

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

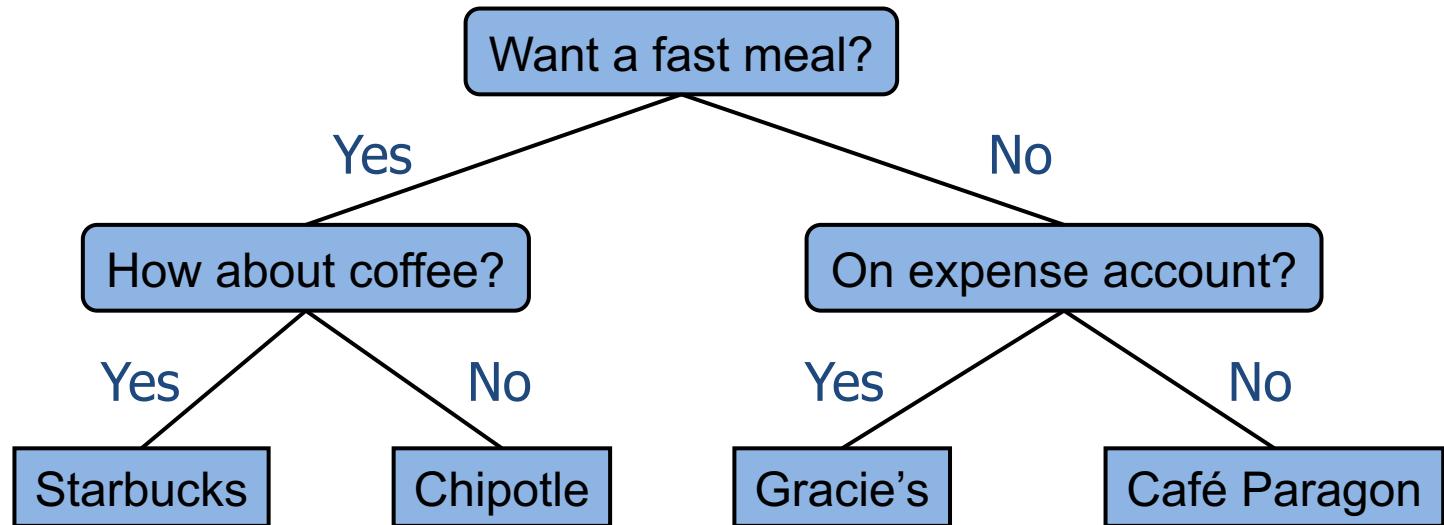


Binary tree application: Decision trees

Tree associated with a decision process

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

Example: dining decision



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

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

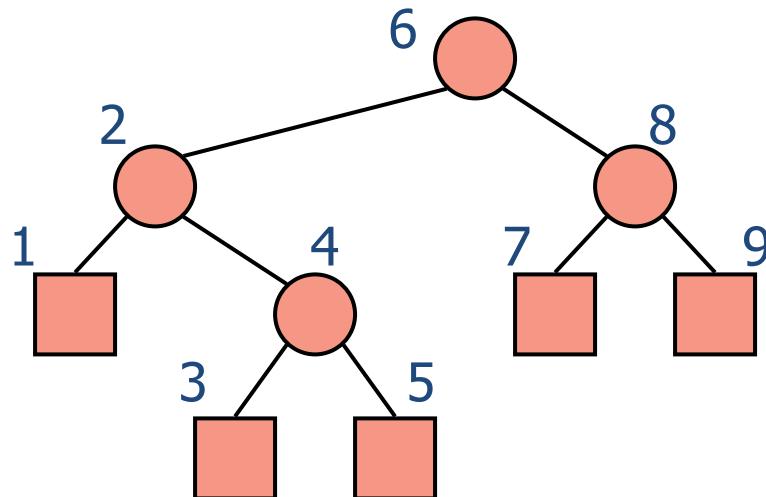
```
def is_external(v)
    # test if v is a leaf
    return v.left = null and v.right = null
```

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

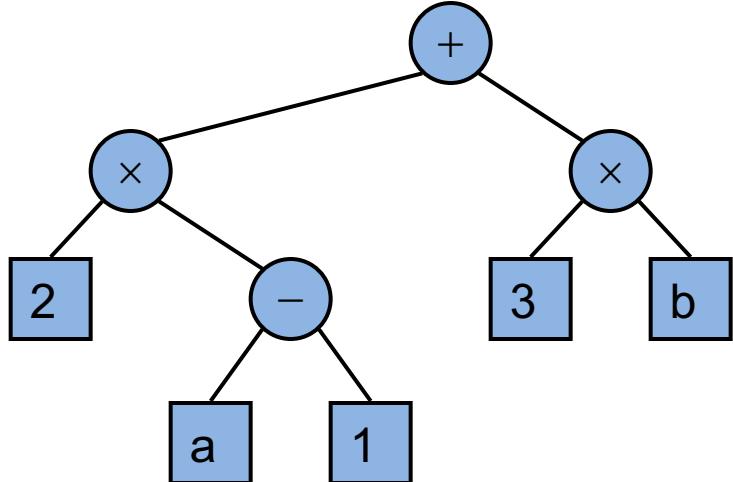


```
def in_order(v)
    if v.left ≠ null then
        in_order(v.left)
    visit(v)
    if v.right ≠ null then
        in_order(v.right)
```

Print Arithmetic Expressions

Extended inorder traversal:

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



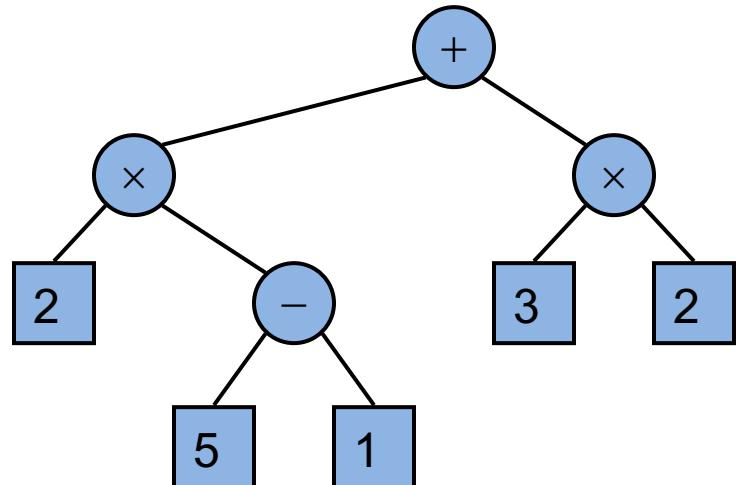
```
def print_expr(v)
    if v.left ≠ null then
        print("(")
        print_ expr(v.left)
    print(v.element)
    if v.right ≠ null then
        print_expr(v.right)
    print (")")
```

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

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



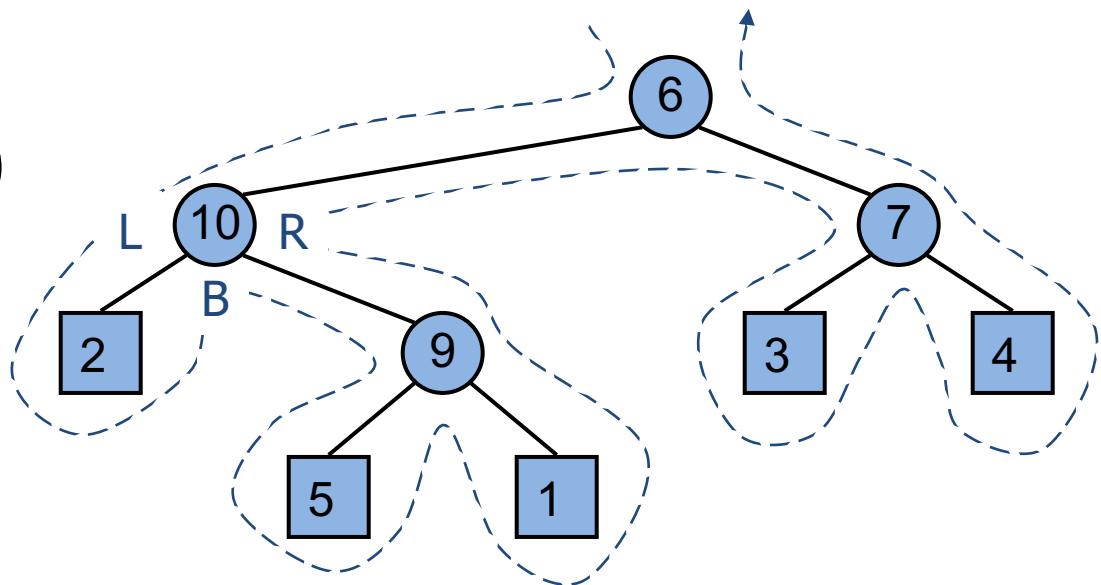
```
def eval_expr(v)
    if v.is_external() then
        return v.element
    else
        x ← eval_expr(v.left)
        y ← eval_expr(v.right)
        ⊕ ← v.element
        return x ⊕ y
```

Euler Tour Traversal

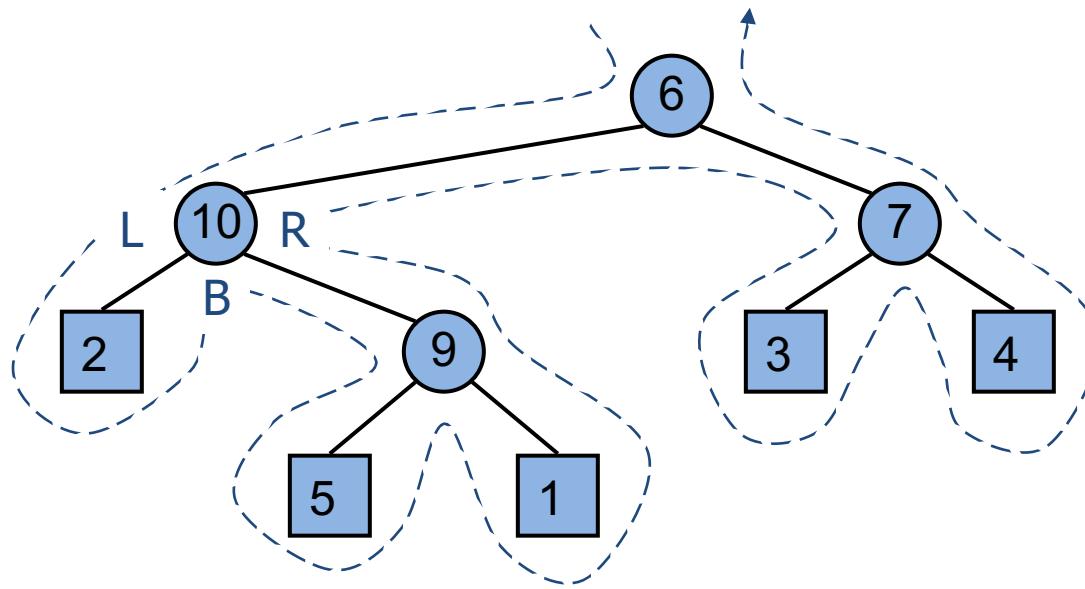
Generic traversal of a binary tree. Includes as special cases the preorder, postorder and inorder traversals

Walk around the tree, keeping the tree on your left, and visit each node three times:

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



Euler Tour Traversal



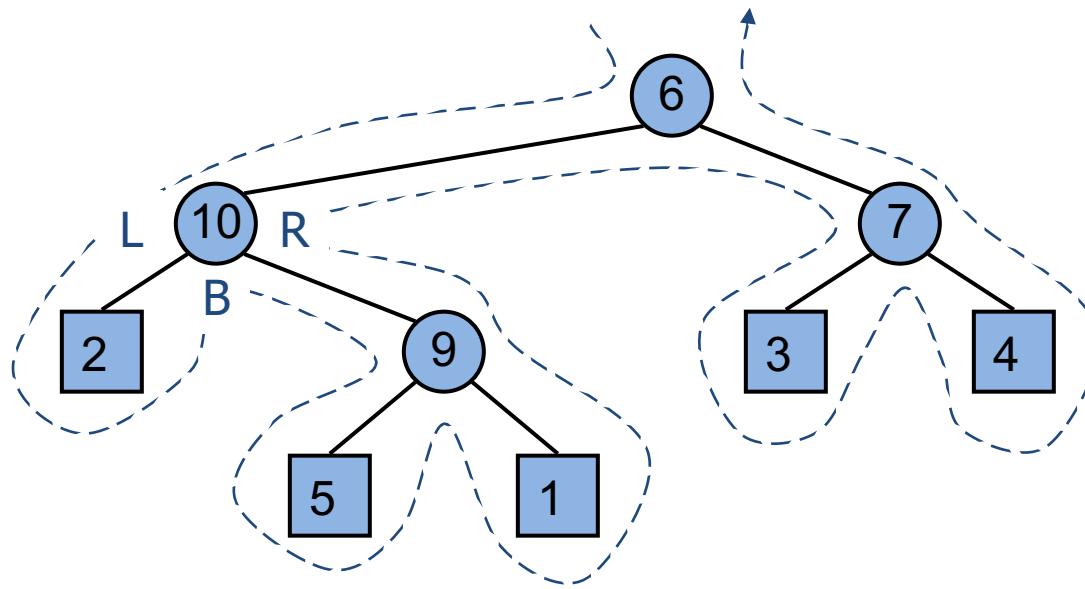
6,10,2,2,2,10,9,5,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,4,7,6

Preorder (first visit):

Inorder (second visit):

Postorder (third visit):

Euler Tour Traversal



6,10,2,2,2,10,9,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,4,7,6

Preorder (first visit): 6, 10, 2, 9, 5, 1, 7, 3, 4

Inorder (second visit): 2, 10, 5, 9, 1, 6, 3, 7, 4

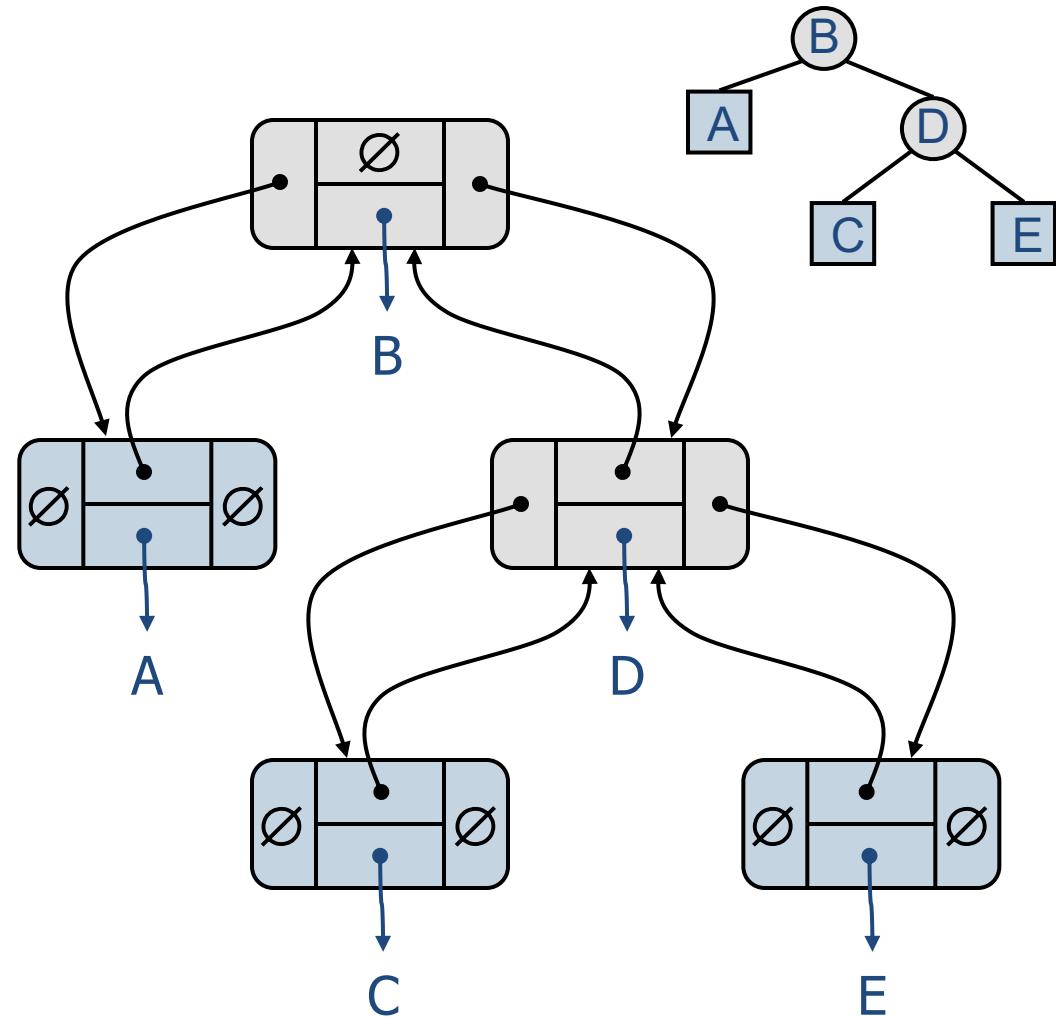
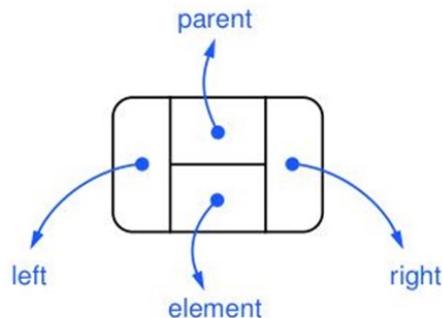
Postorder (third visit): 2, 5, 1, 9, 10, 3, 4, 7, 6

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

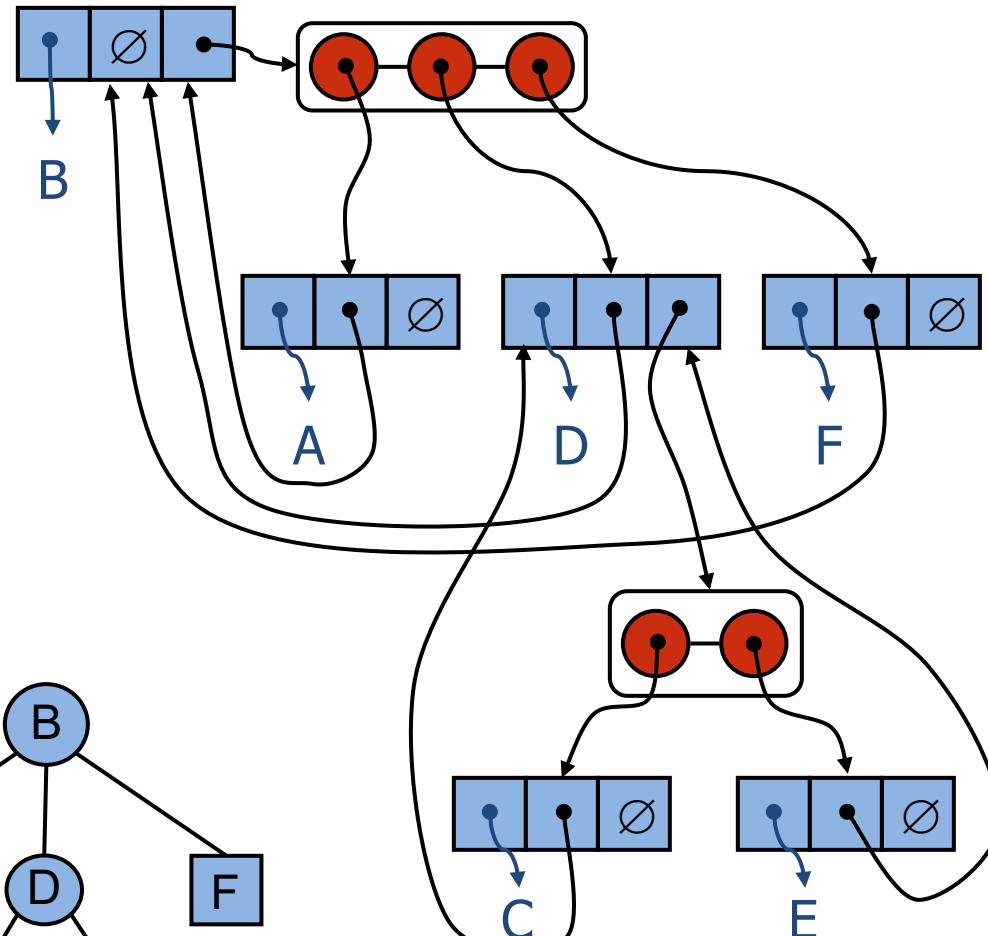
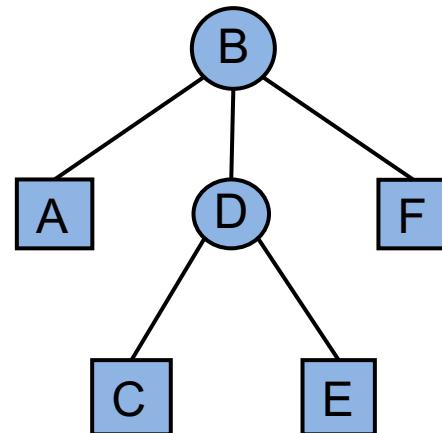
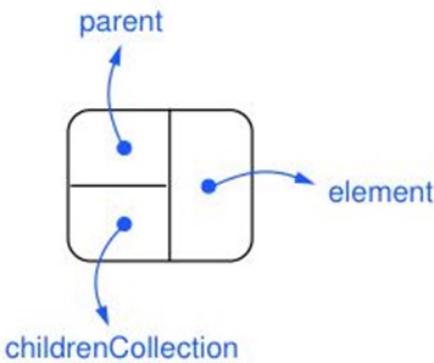


Linked Structure for General Trees

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 = null 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
            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

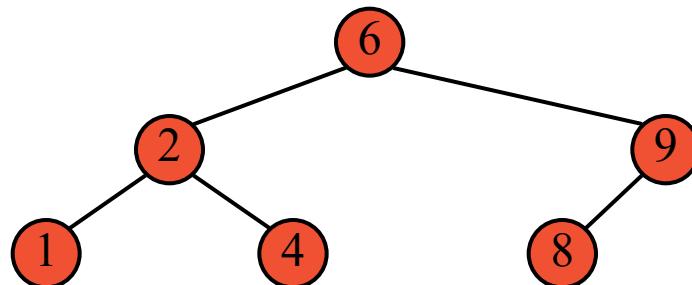
- In worst case, do one call at each level of the tree
- If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the height of the tree

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!



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

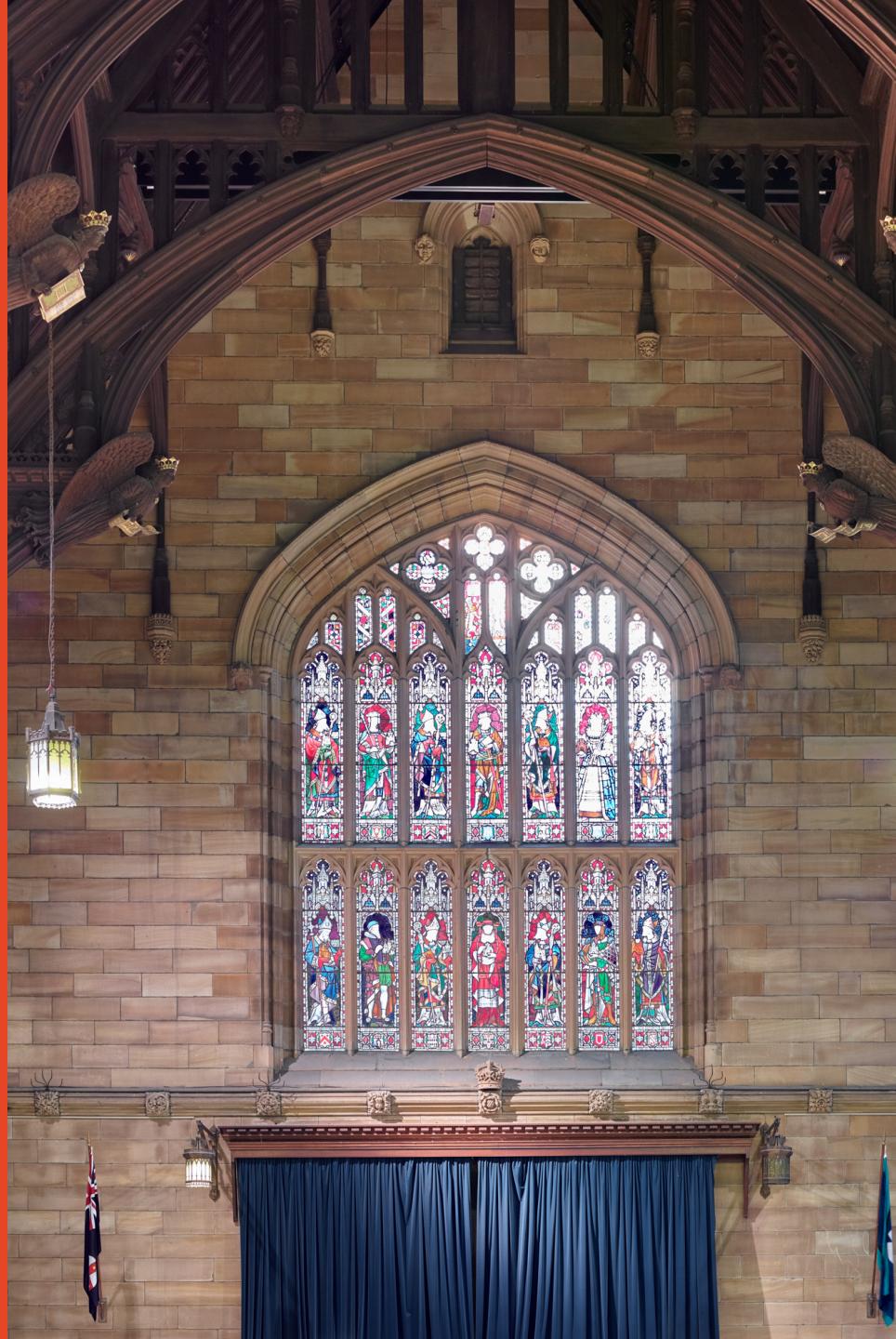
Binary Search Trees
[GT 3.1-2] [GT 4.2]

Dr. André van Renssen
School of Computer Science

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



THE UNIVERSITY OF
SYDNEY



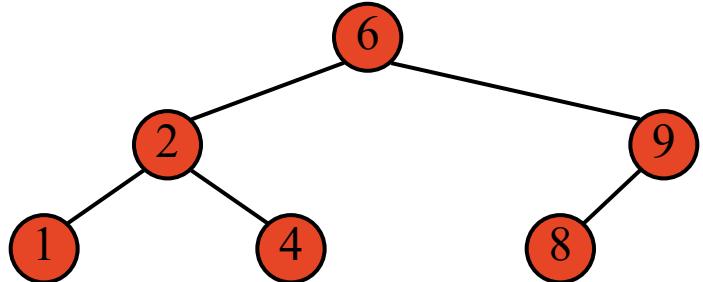
Binary Search Trees (BST)

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 ,

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

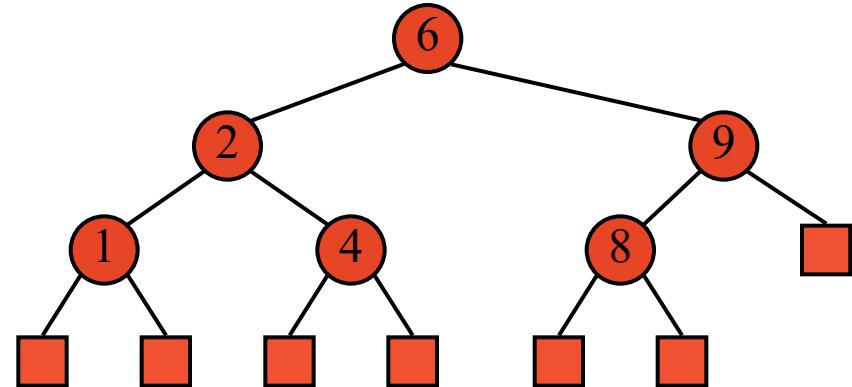
Note that an inorder traversal
of a binary search tree visits the keys
in increasing order.



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)

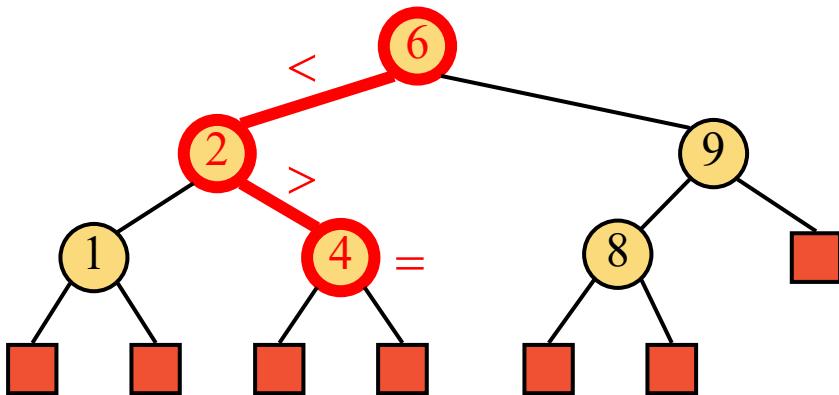


Searching with a Binary Search Tree

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

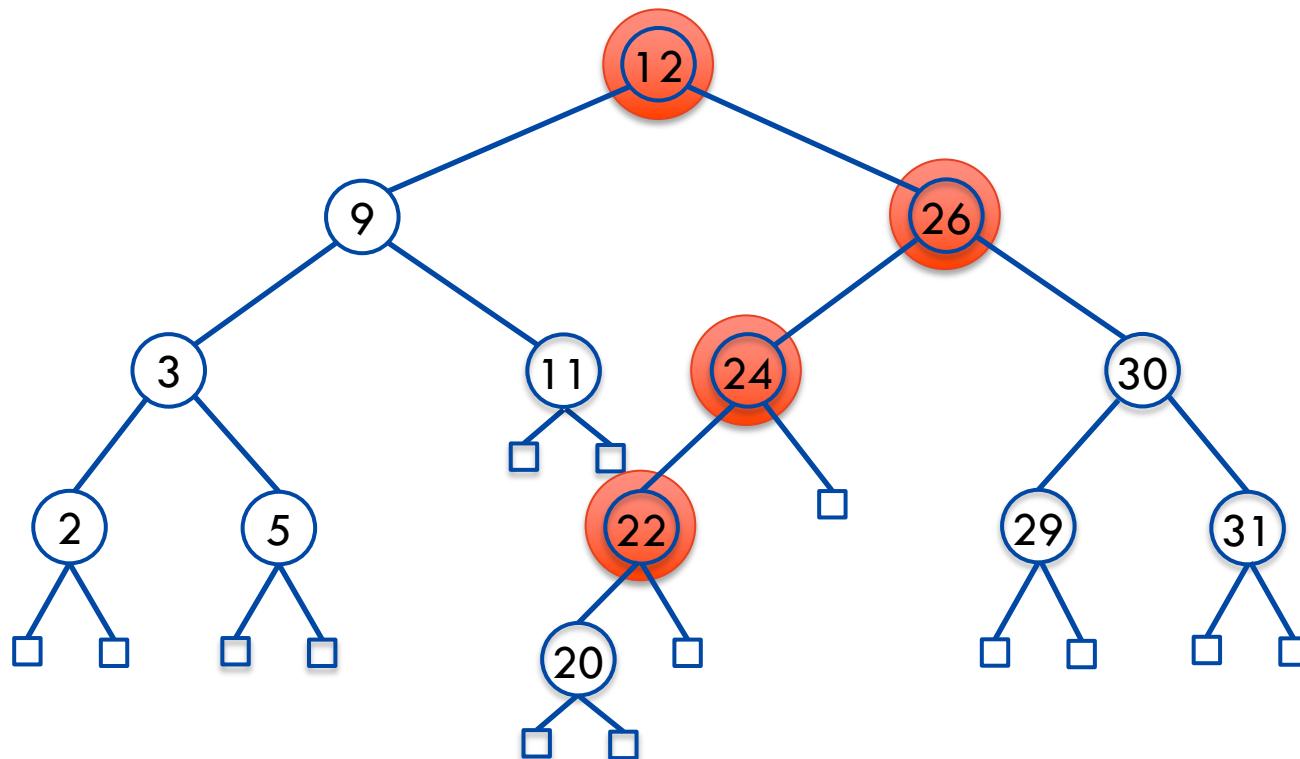
If we reach an external node, this means that the key is not in the data structure



```
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)
```

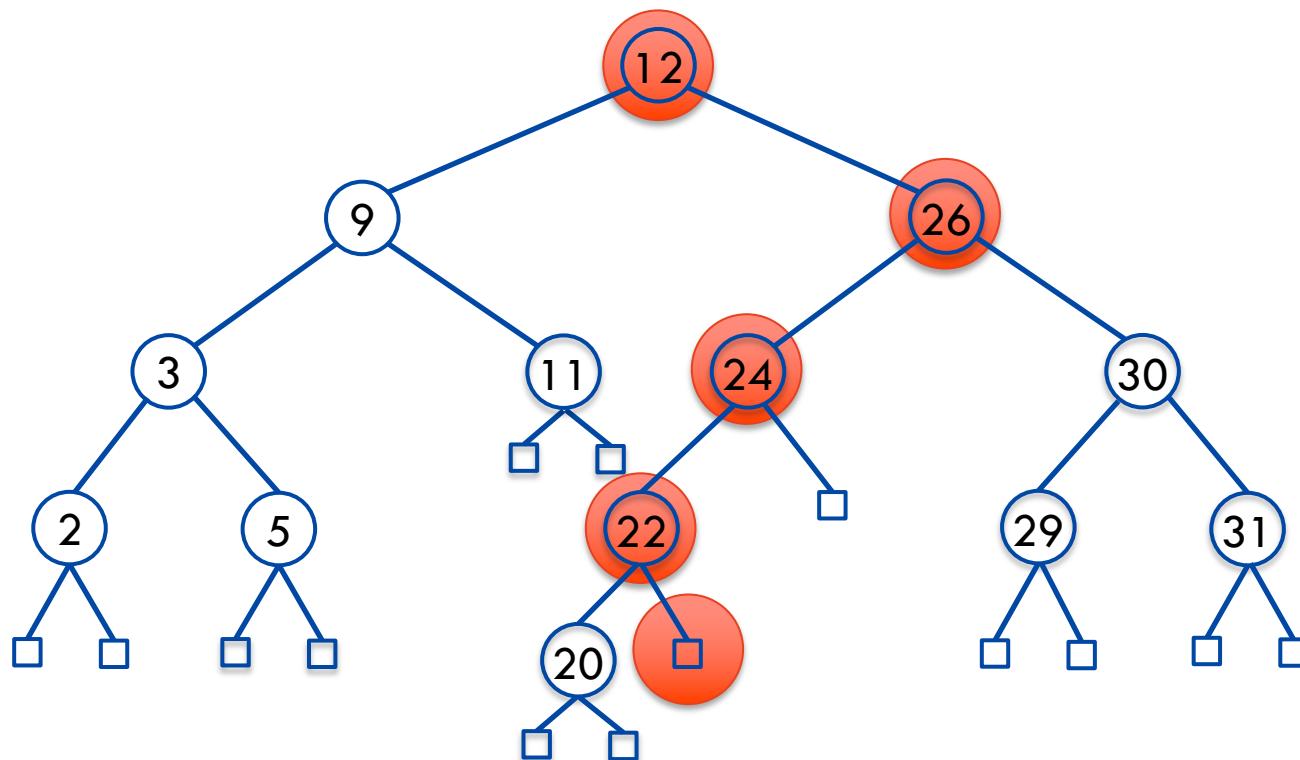
Example: Find 22

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



Example: Find 23

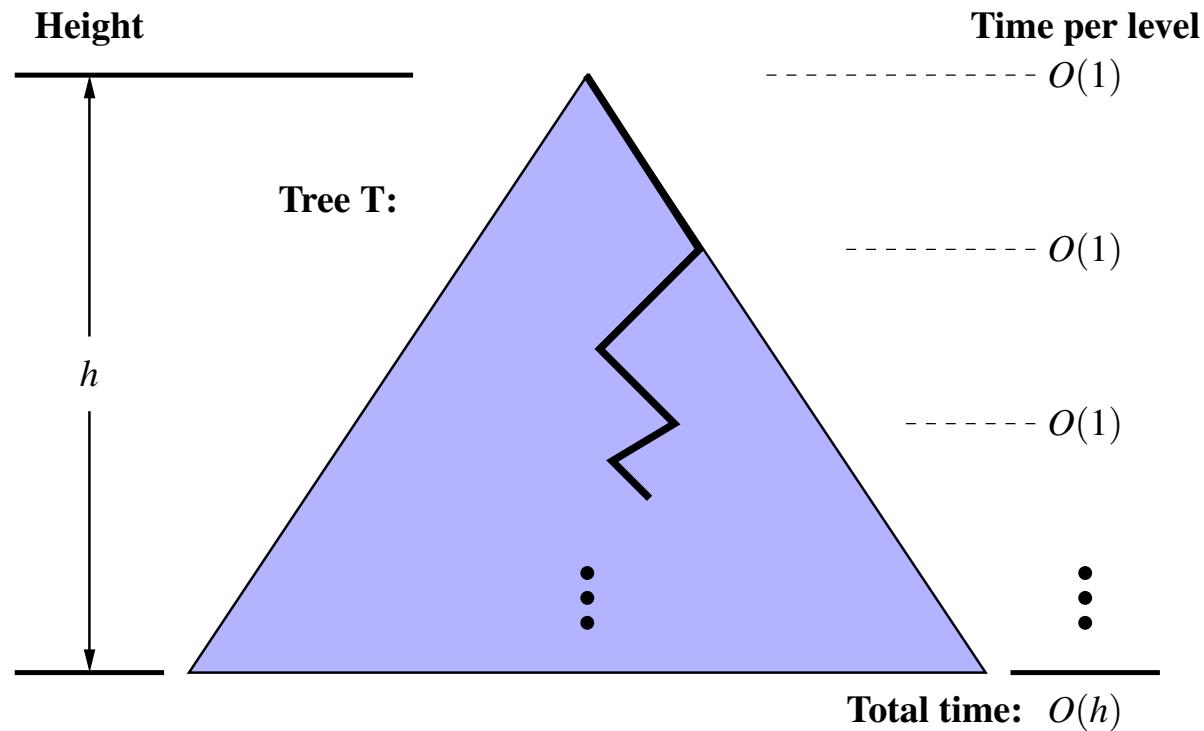
$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$



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 $h \leq \log_2 n$

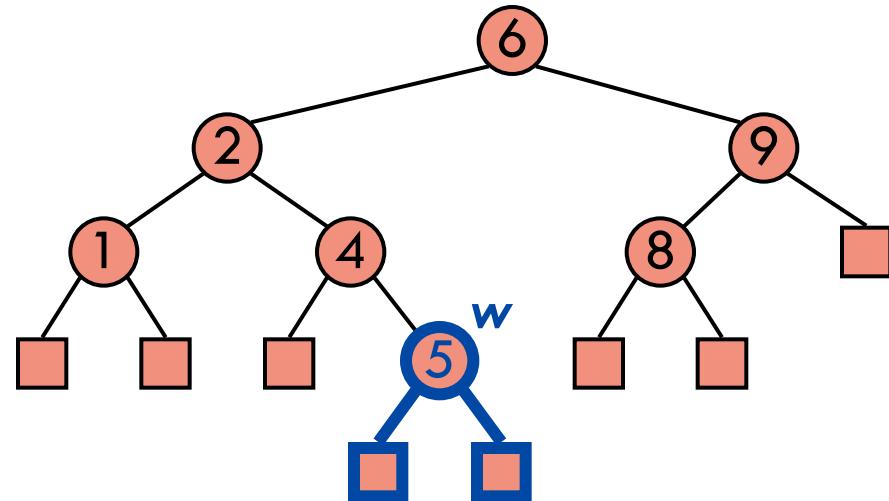
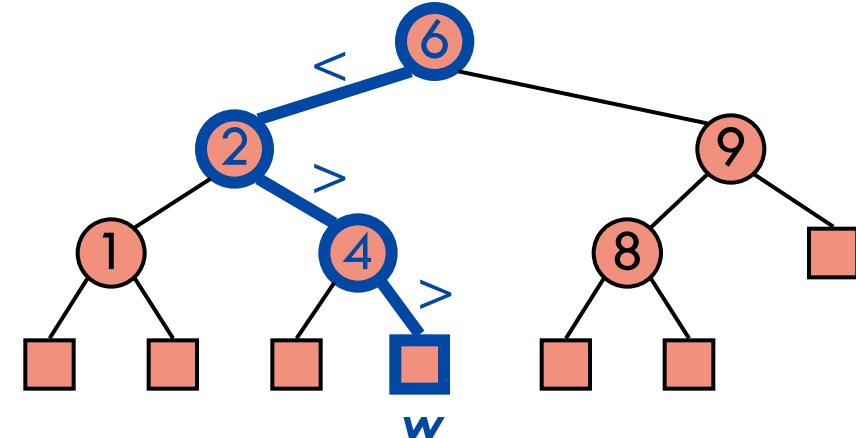


Insertion

To perform operation **put(k, o)**, we search for key k (using search)

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

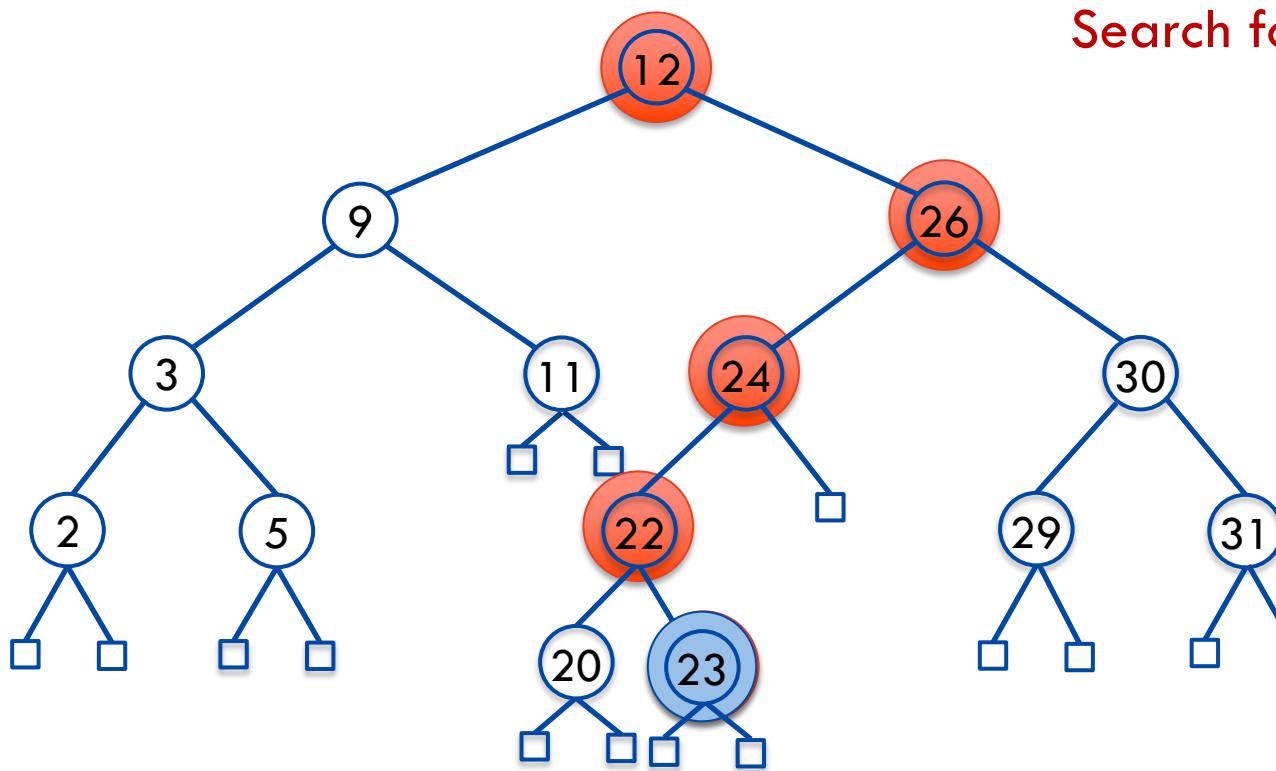
If k is not found, let w be the external node reached by the search. We replace w with an internal node holding (k, o)



Example: Insert 23

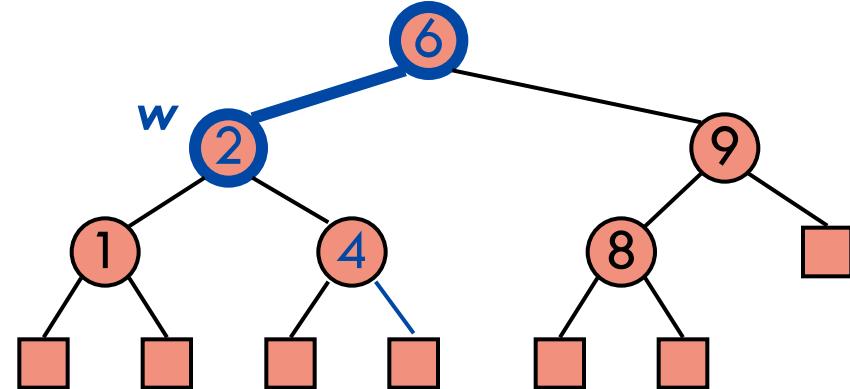
S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Search for 23



Delete

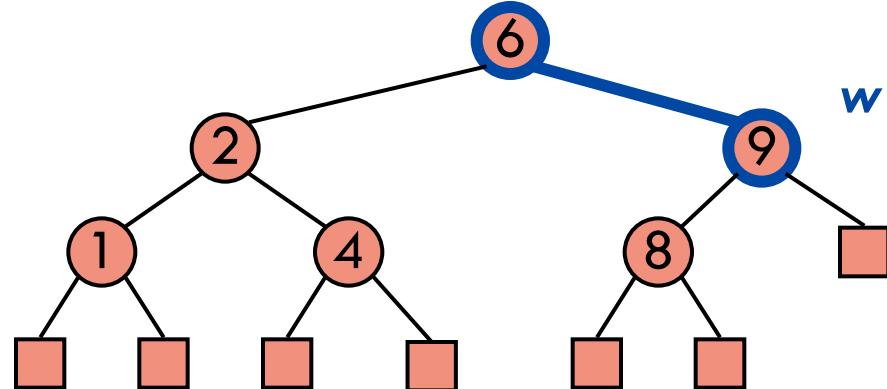
To perform operation `remove(k)`, we search for key k (using search) to find the node w holding k



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

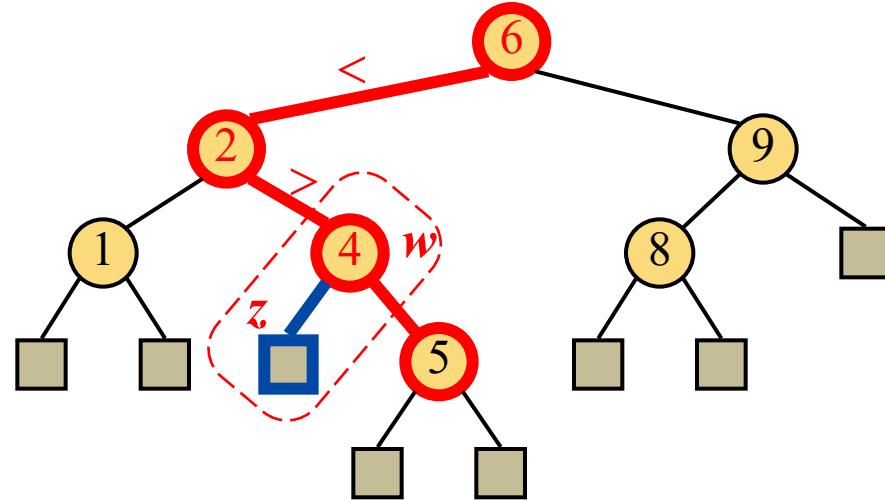


Deletion Case 1

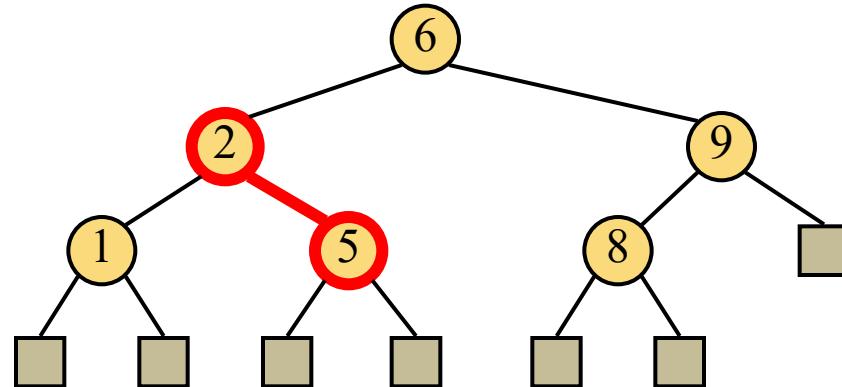
Suppose that the node w we want to remove has an external child, which we call z .

To remove w we

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



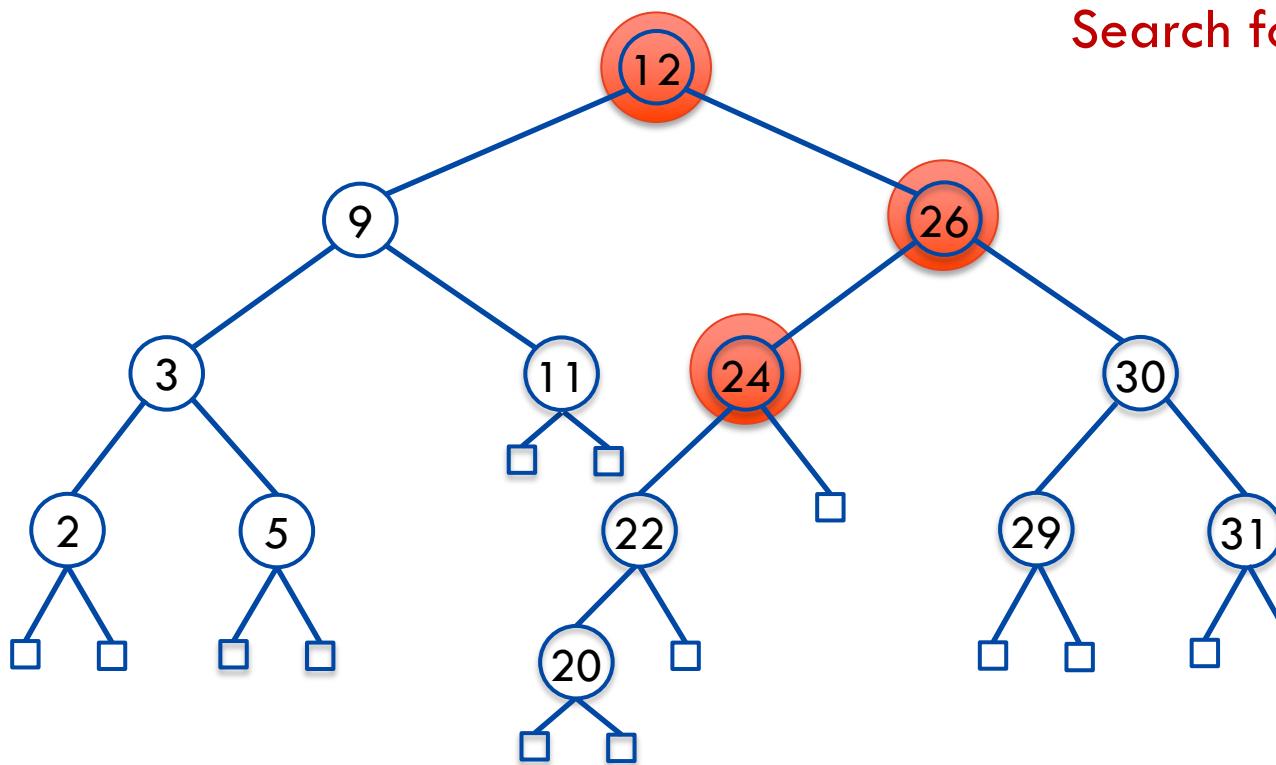
This preserves the BST property



Example: Delete 24

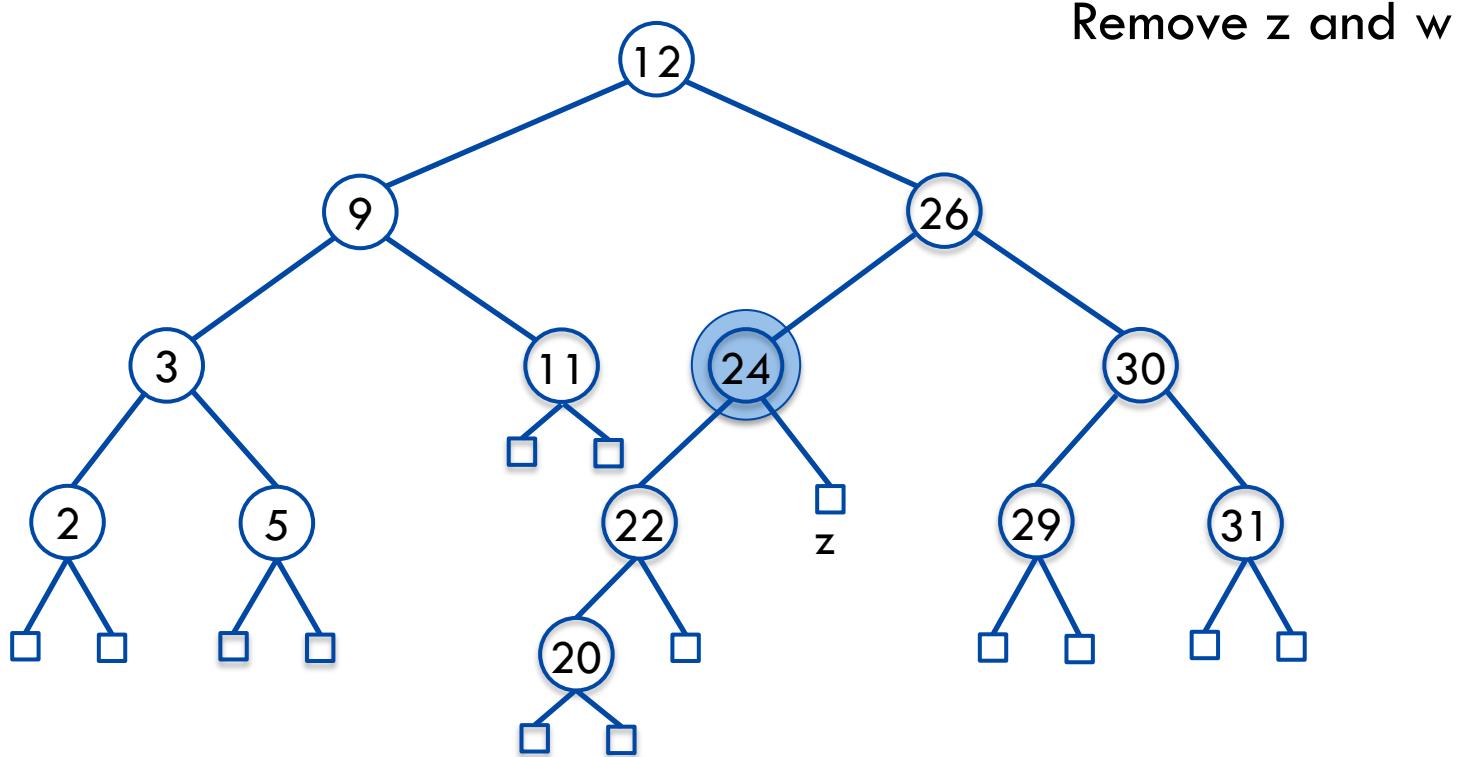
$S=\{2,3,5,9,11,12,20,22,24,26,29,30,31\}$

Search for 24



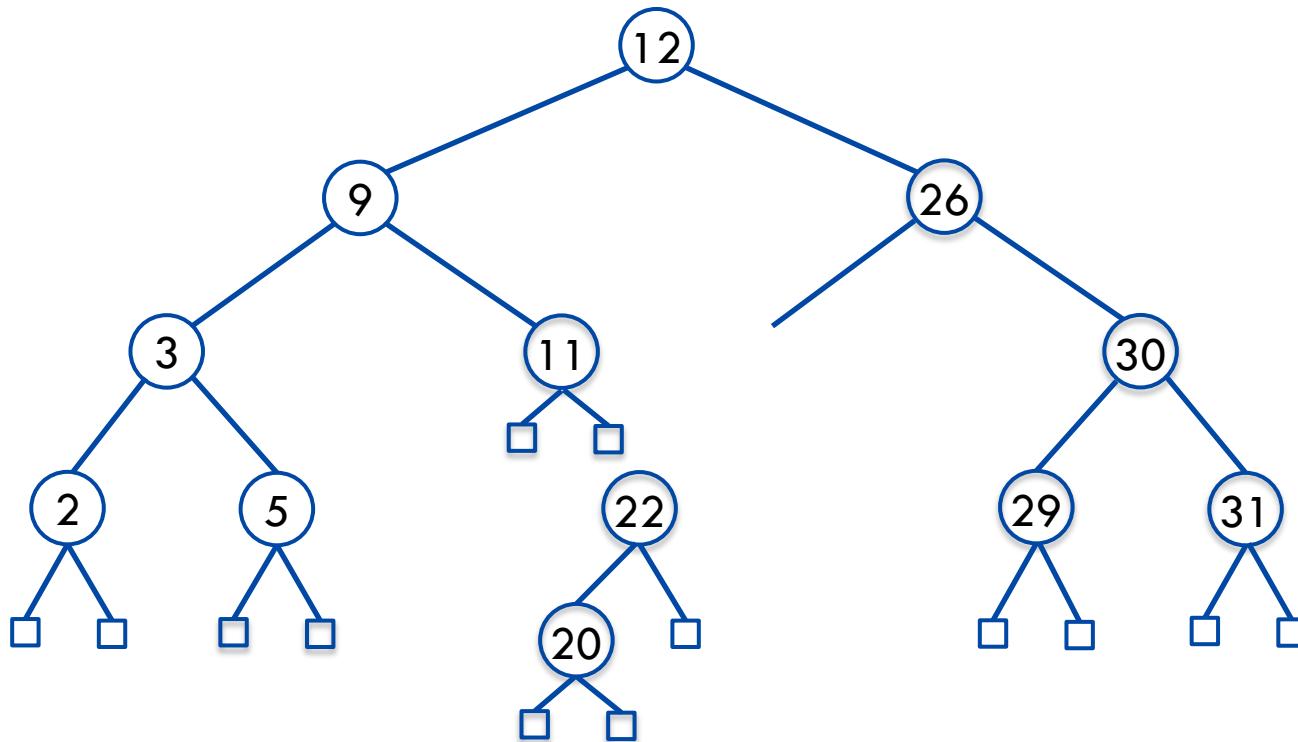
Example: Delete 24

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



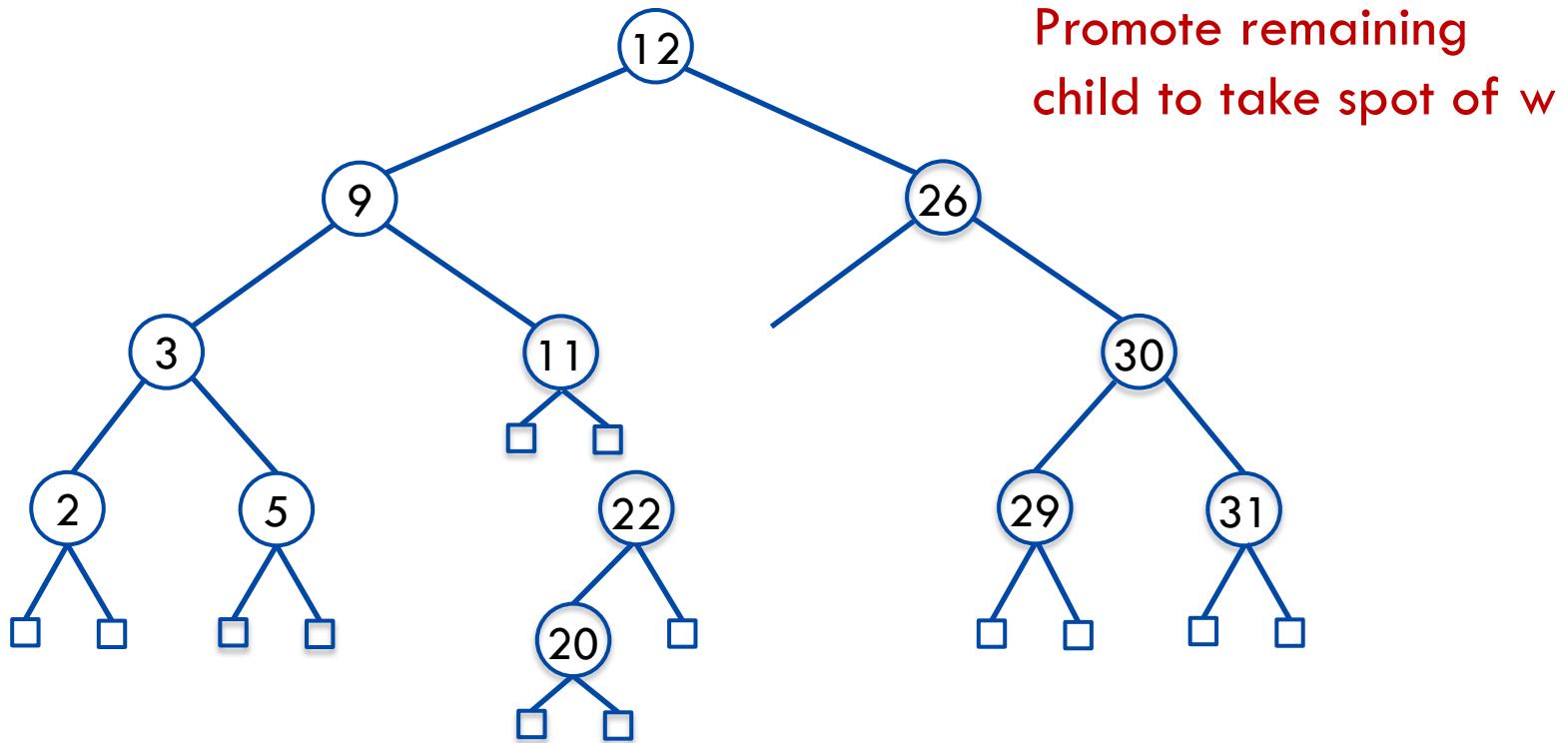
Example: Delete 24

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



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

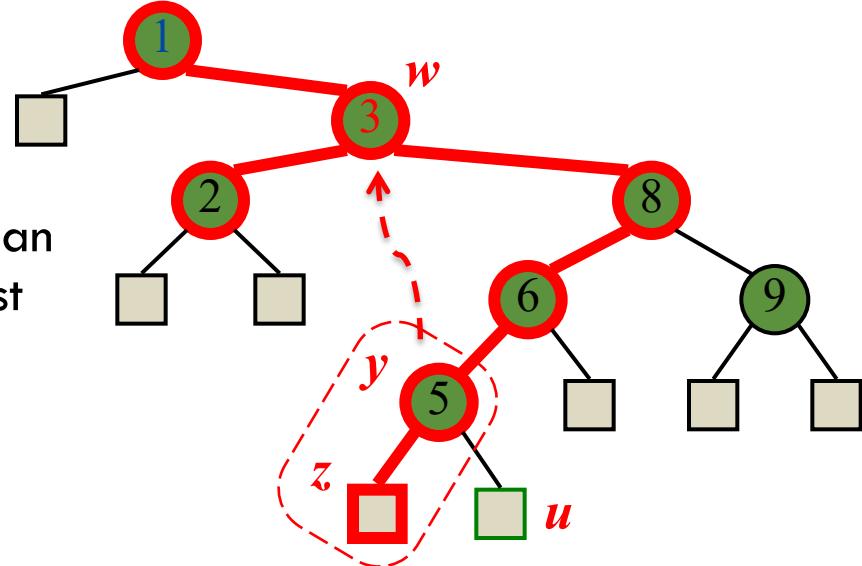
Deletion : Case 2

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

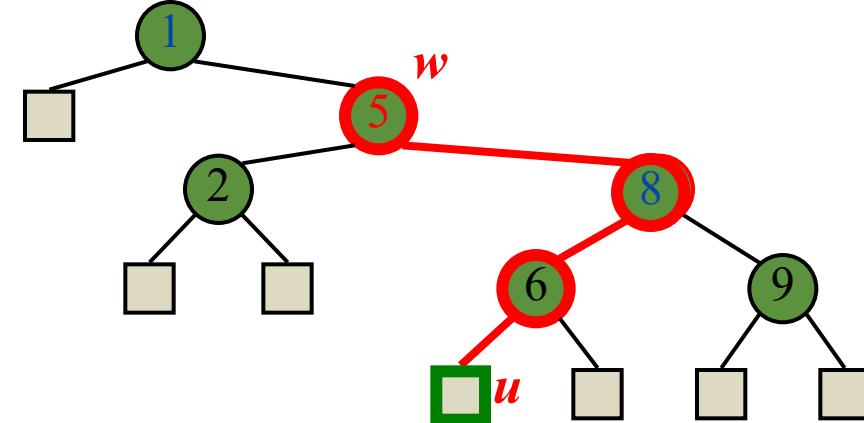
To remove w we

- 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 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 as above
```

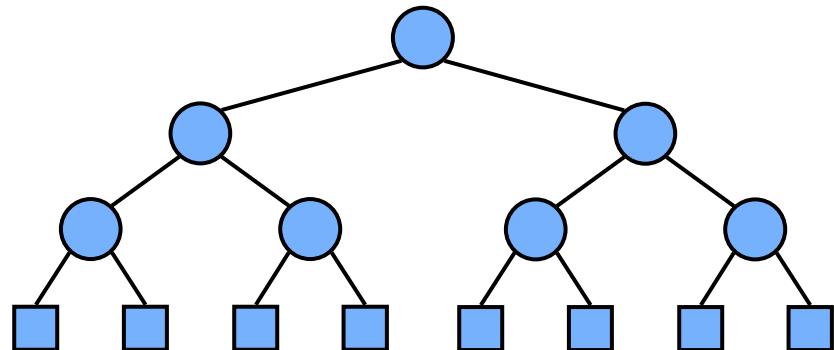
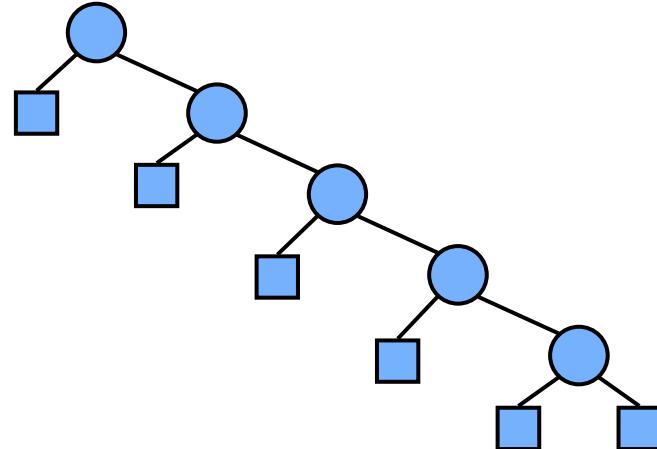
Complexity

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

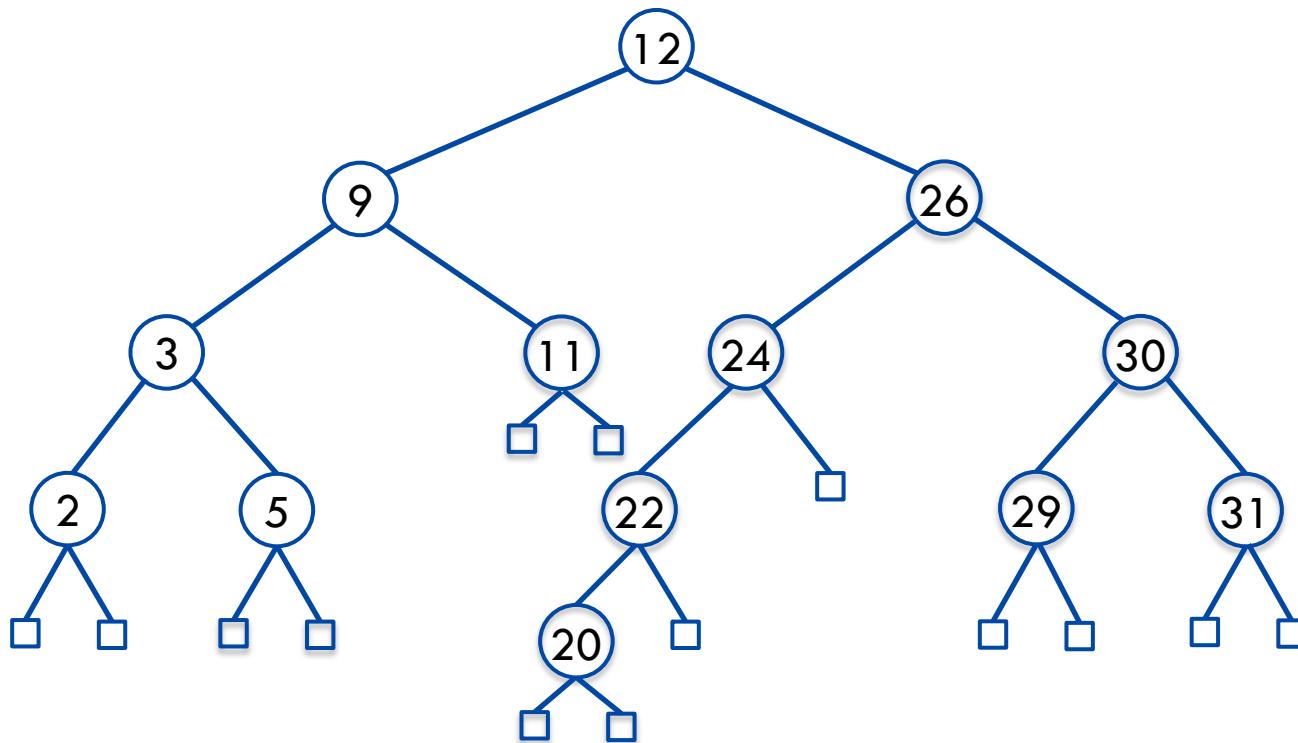
Pseudo-code

```
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

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

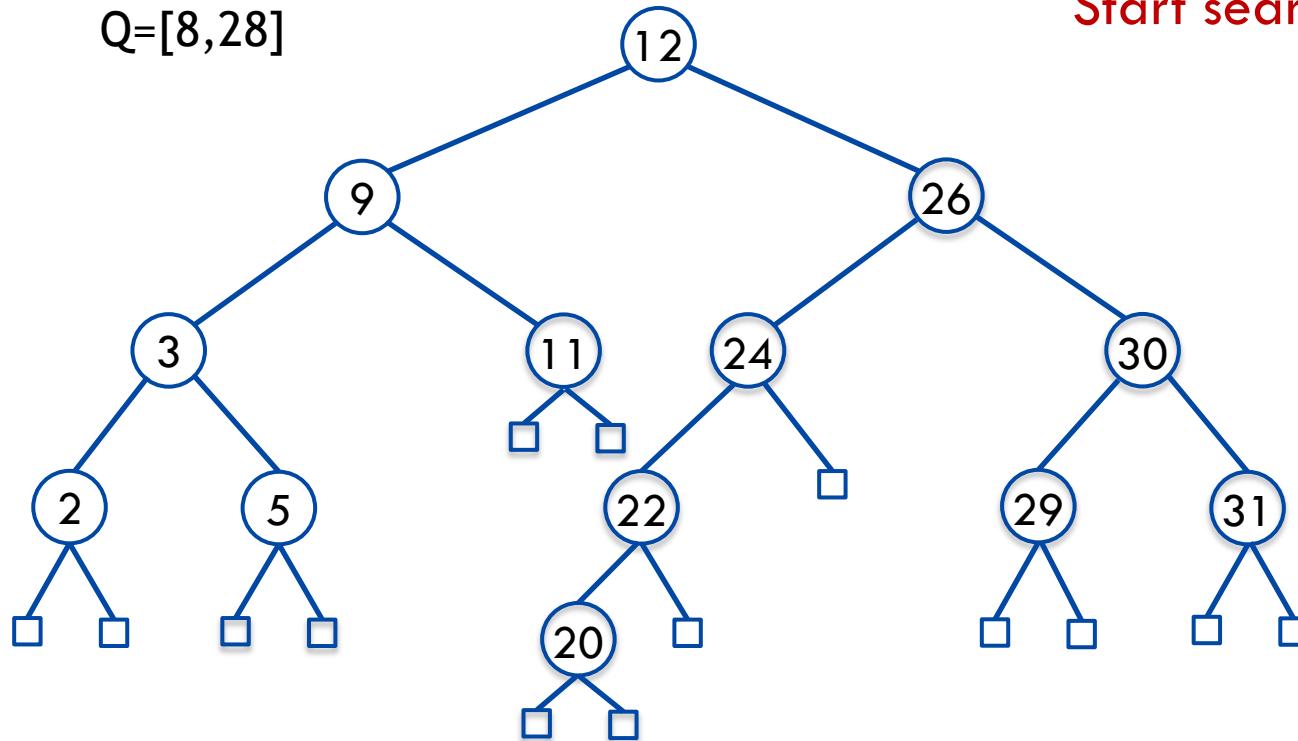


Range queries

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

$Q = [8, 28]$

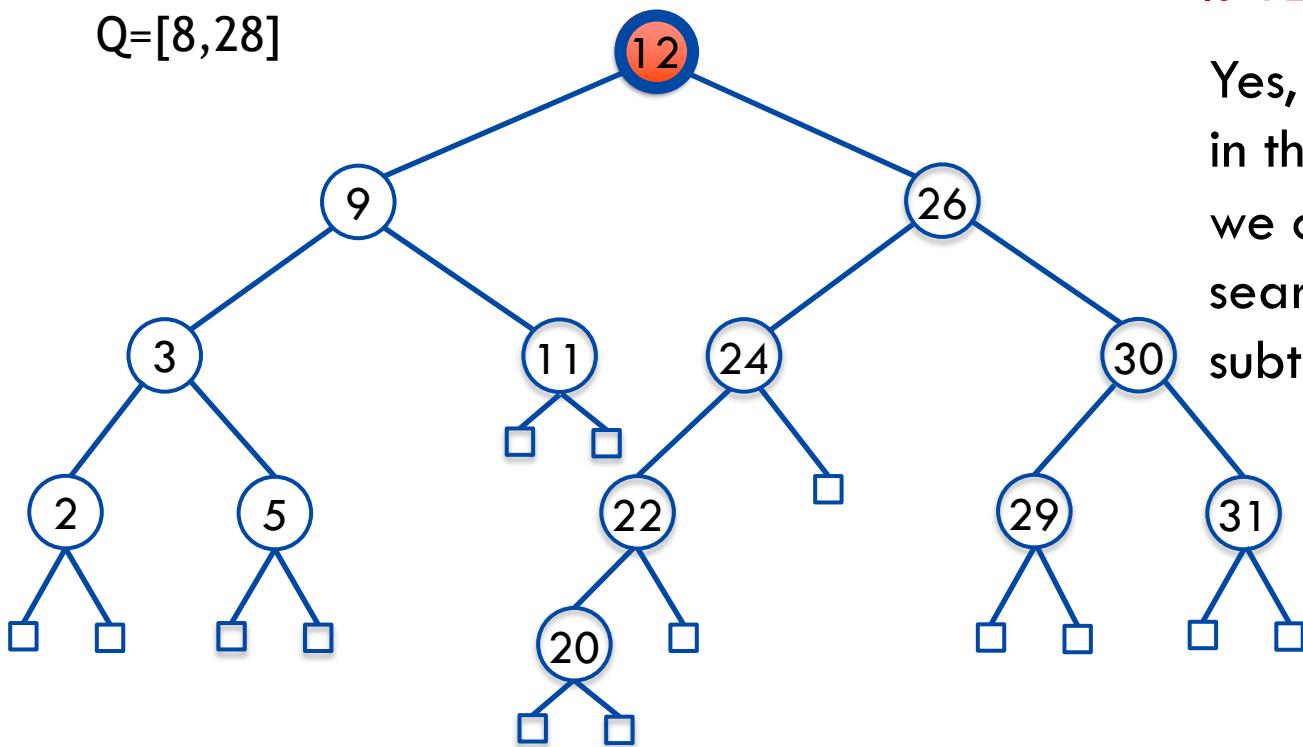
Start search



Range queries

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

$Q = [8, 28]$



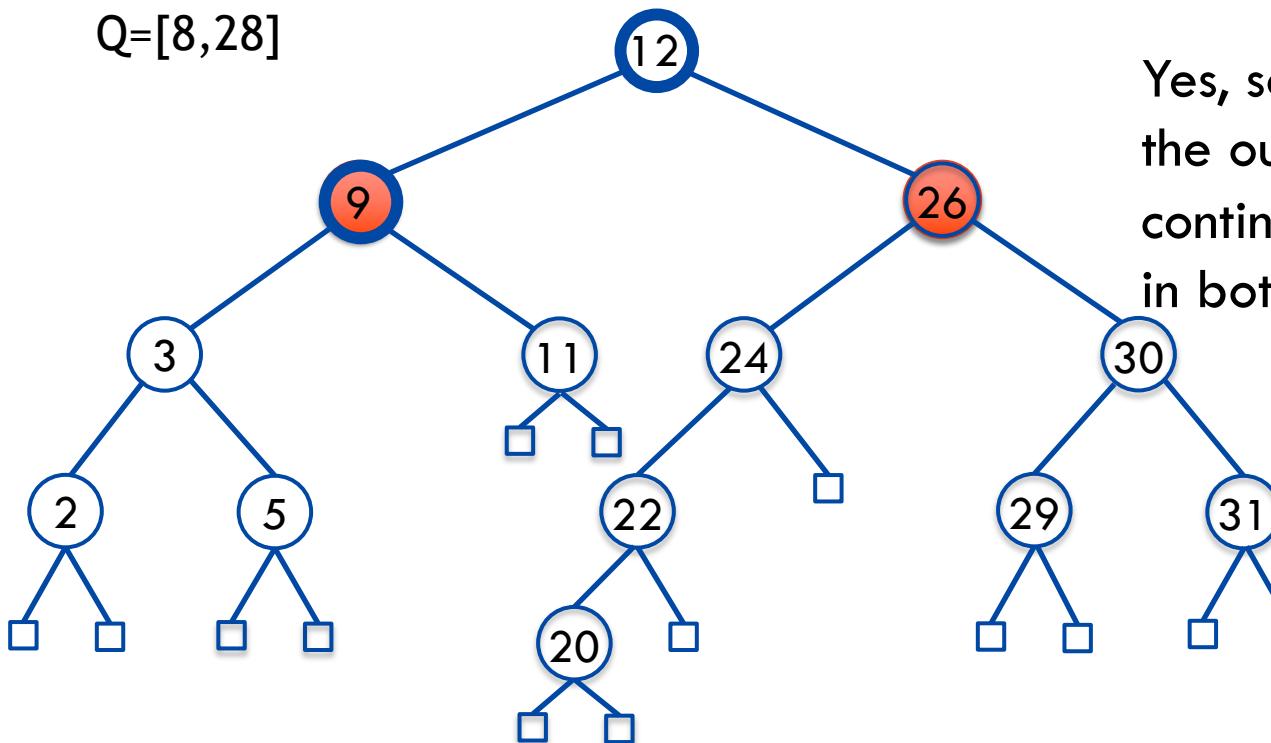
Is 12 in Q?

Yes, so 12 will be in the output and we continue the search in both subtrees

Range queries

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

$Q = [8, 28]$



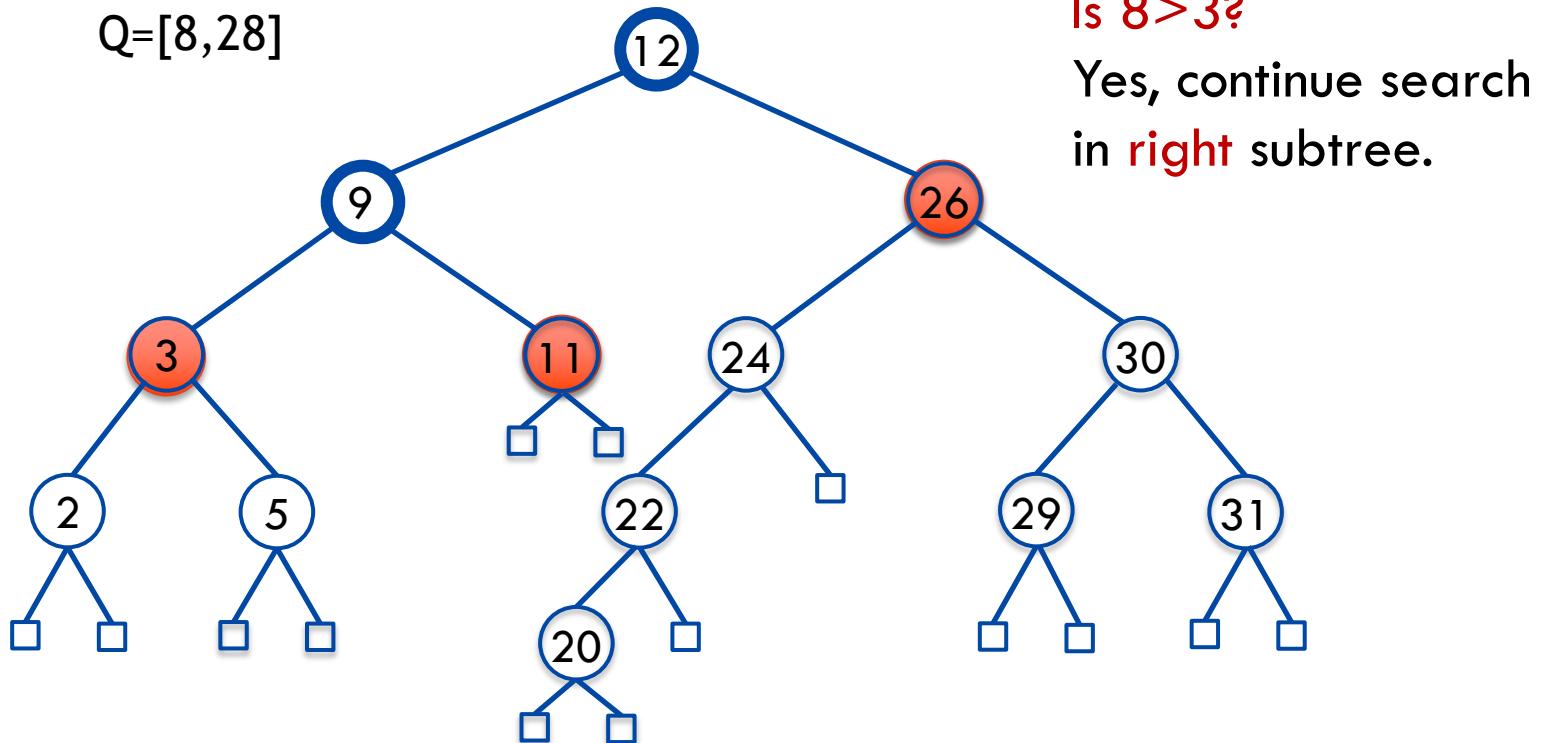
Is 9 in Q?

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

Range queries

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

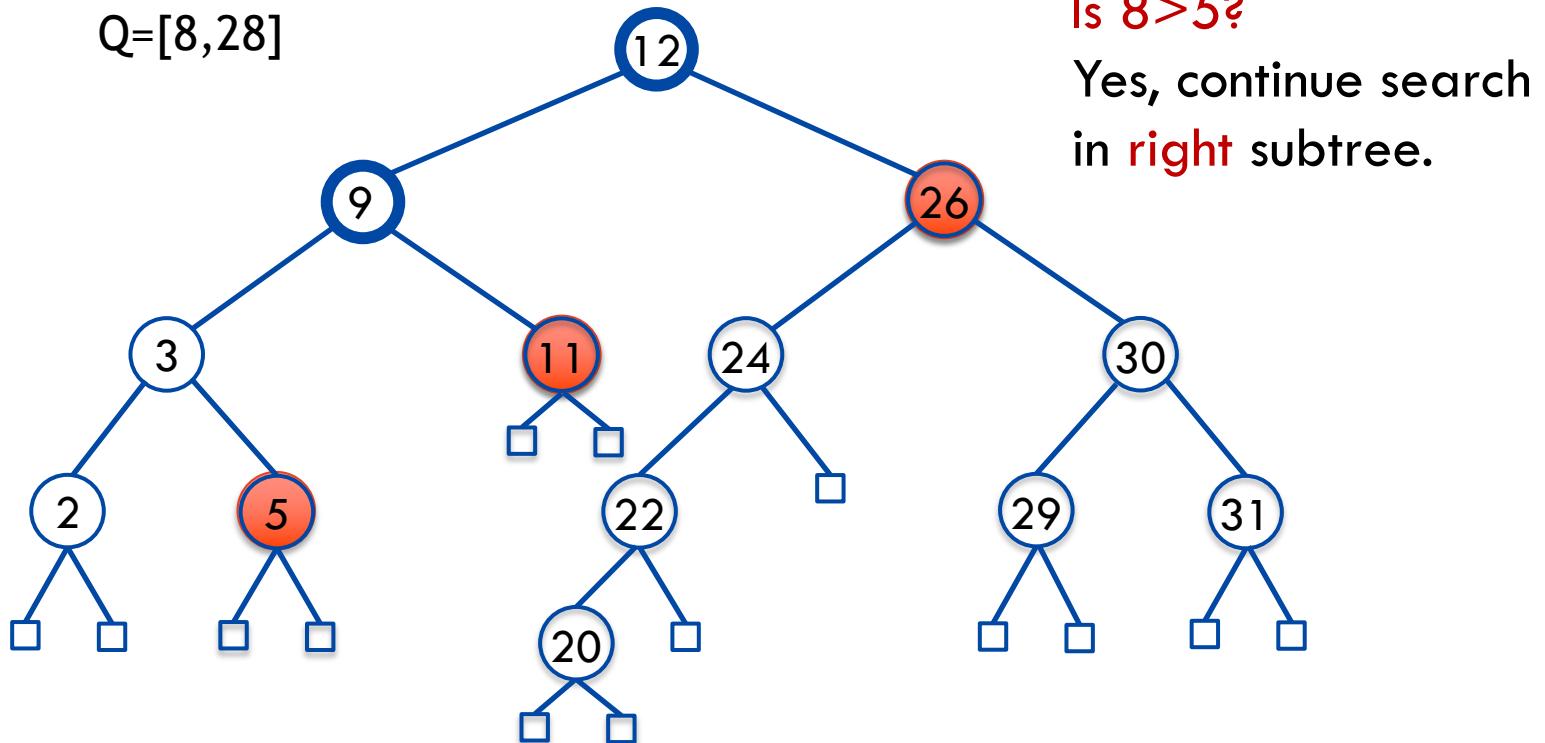
$Q = [8, 28]$



Range queries

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

$Q = [8, 28]$

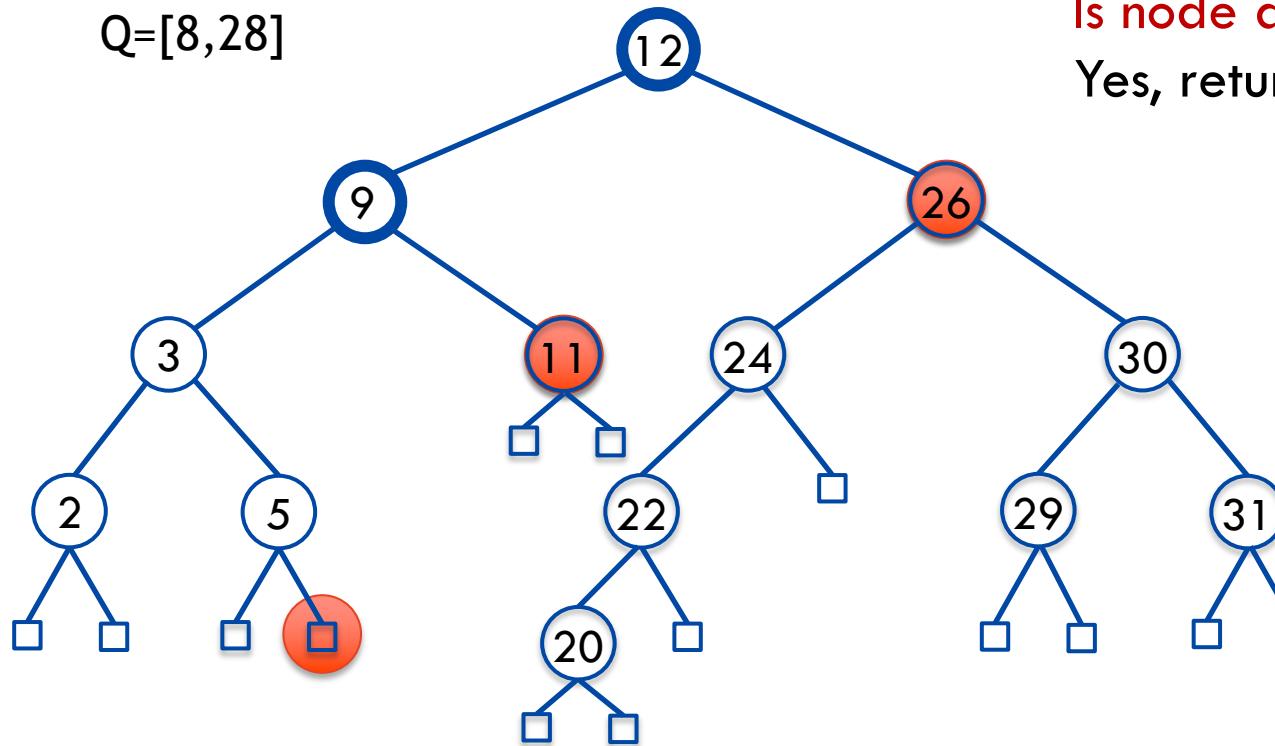


Range queries

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

$Q = [8, 28]$

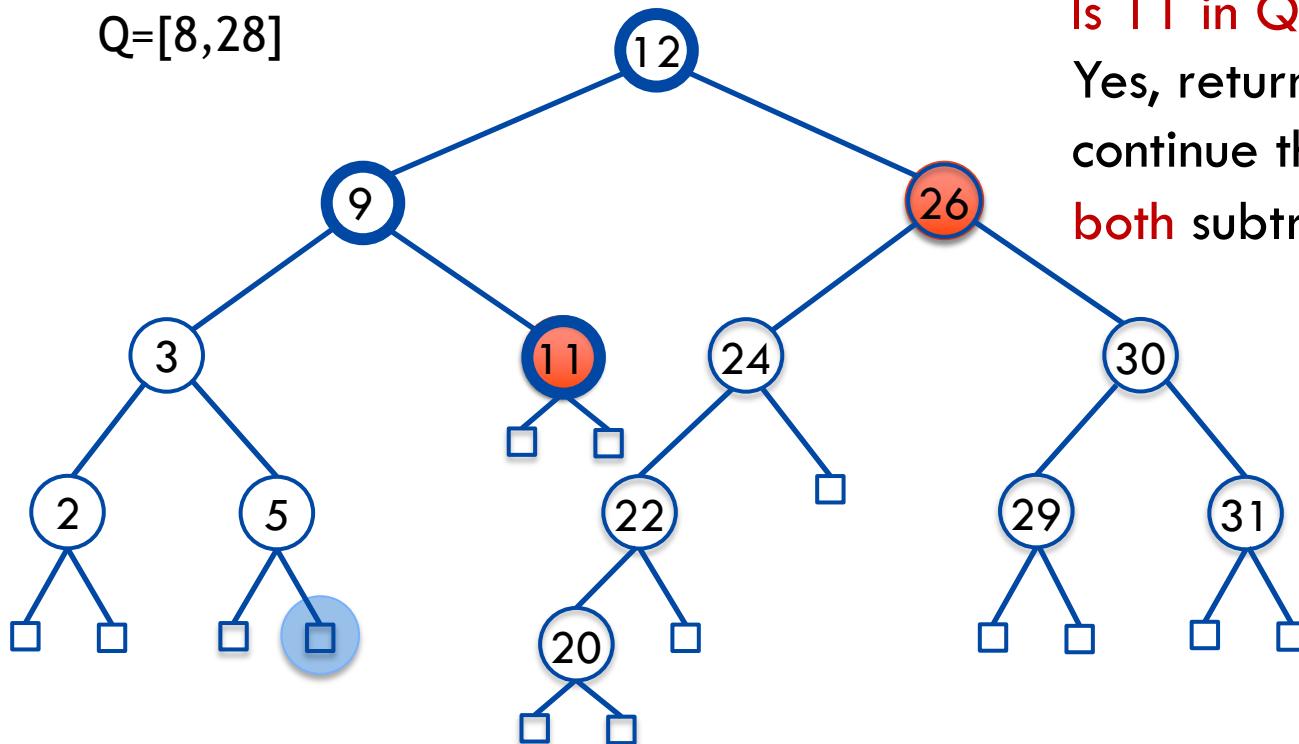
Is node a leaf?
Yes, return \emptyset .



Range queries

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

$Q = [8, 28]$



Is 11 in Q?

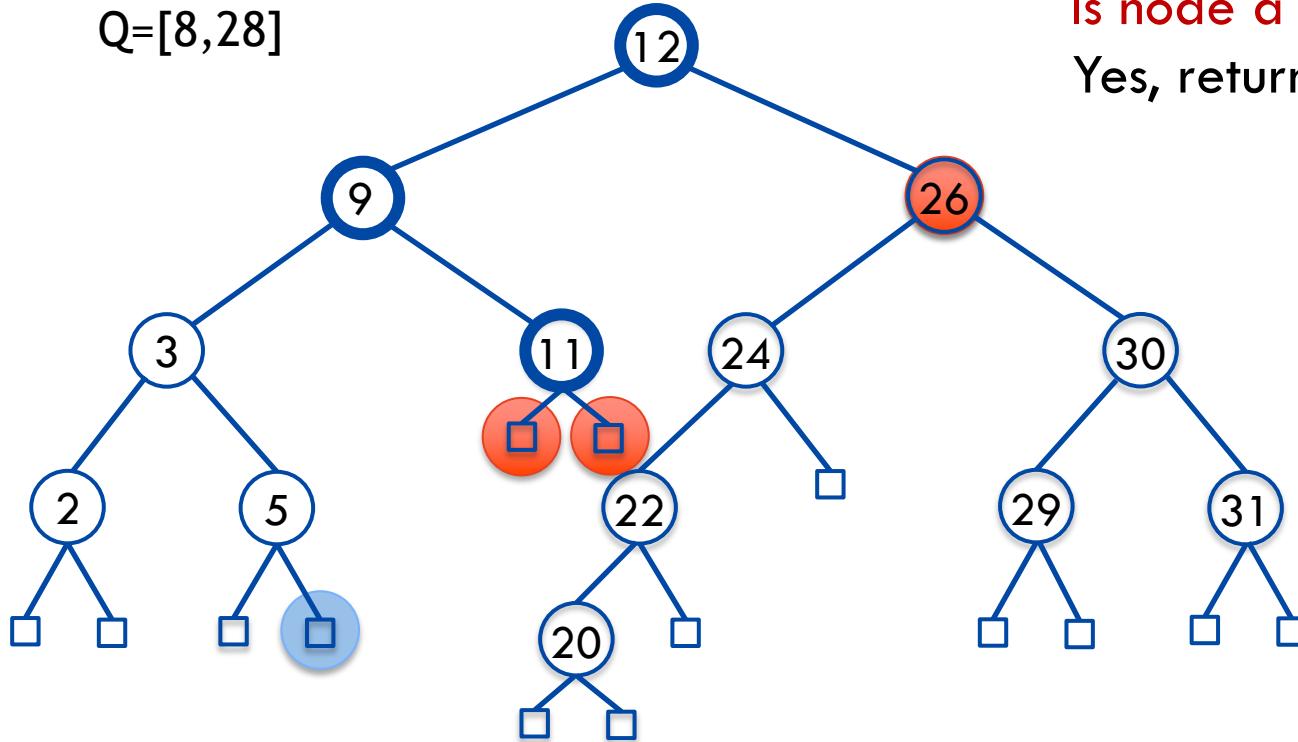
Yes, return 11 and
continue the search in
both subtrees.

Range queries

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

$Q = [8, 28]$

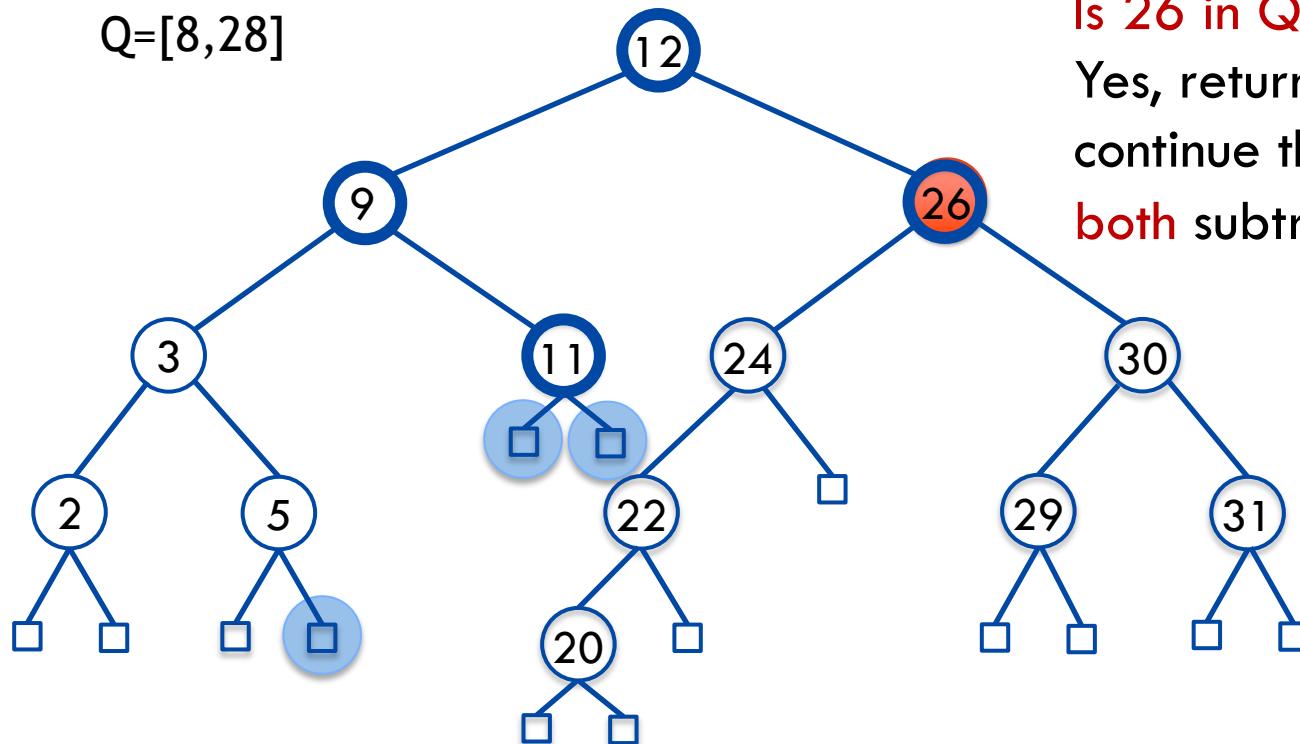
Is node a leaf ($\times 2$)?
Yes, return \emptyset .



Range queries

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

$Q = [8, 28]$



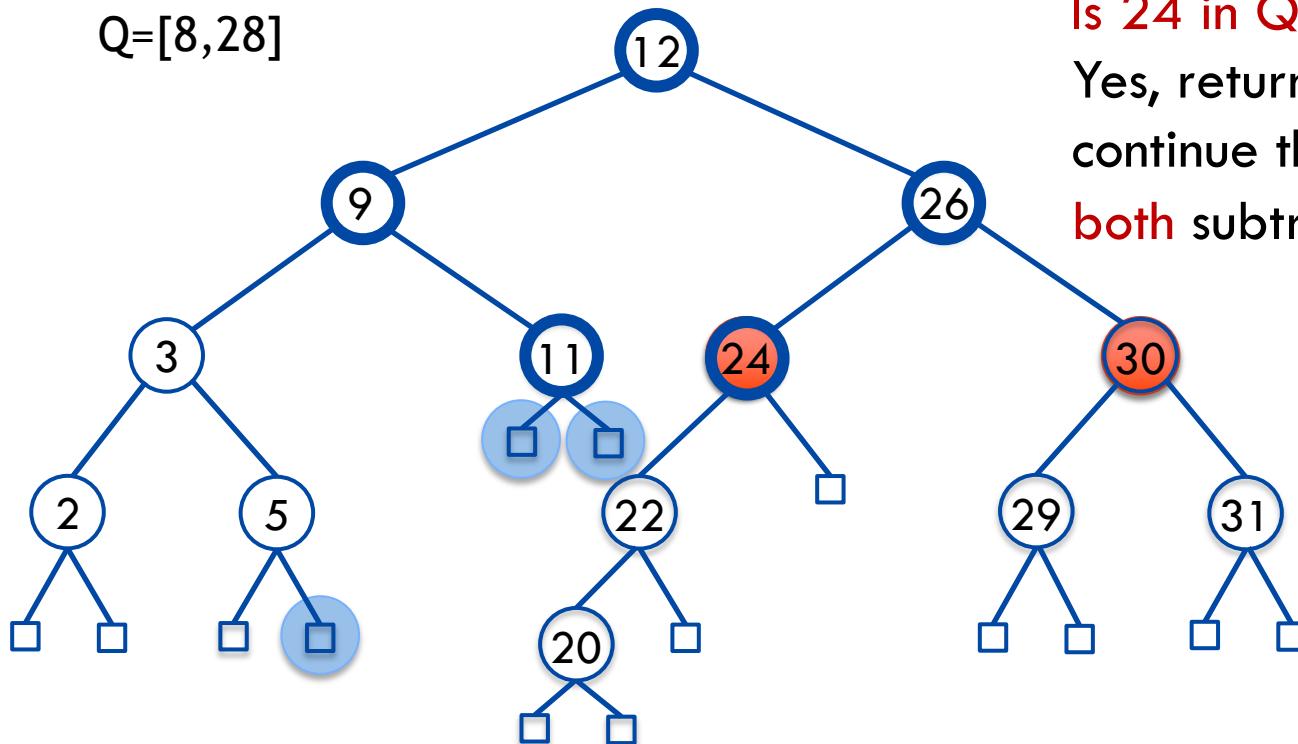
Is 26 in Q?

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

Range queries

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

$Q = [8, 28]$



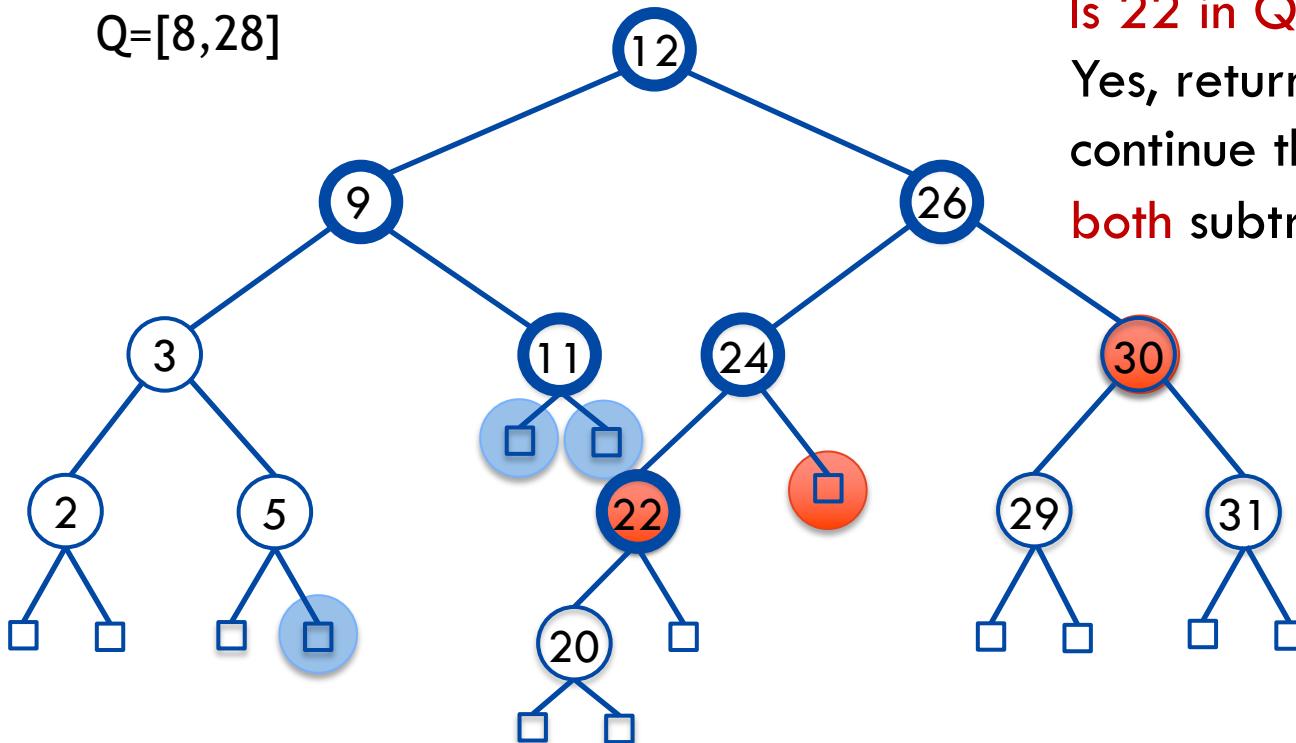
Is 24 in Q?

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

Range queries

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

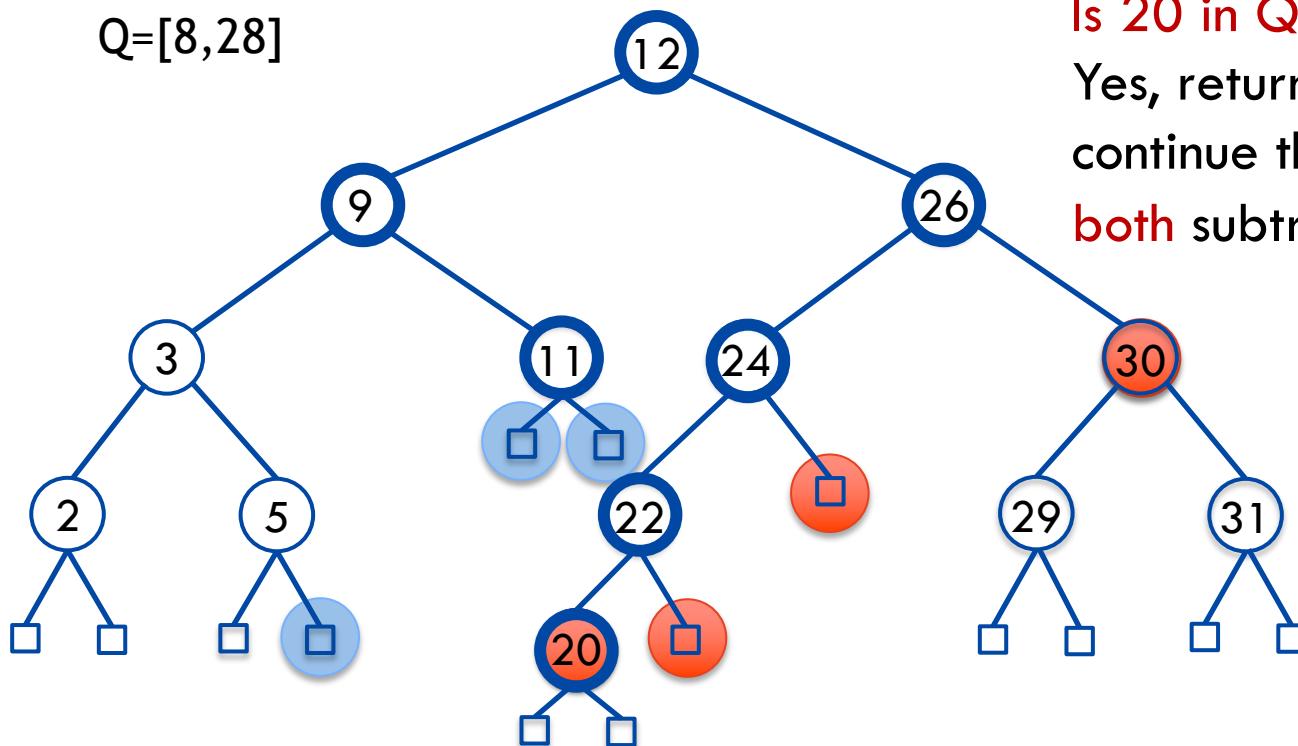
$Q = [8, 28]$



Range queries

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

$Q = [8, 28]$



Is 20 in Q?

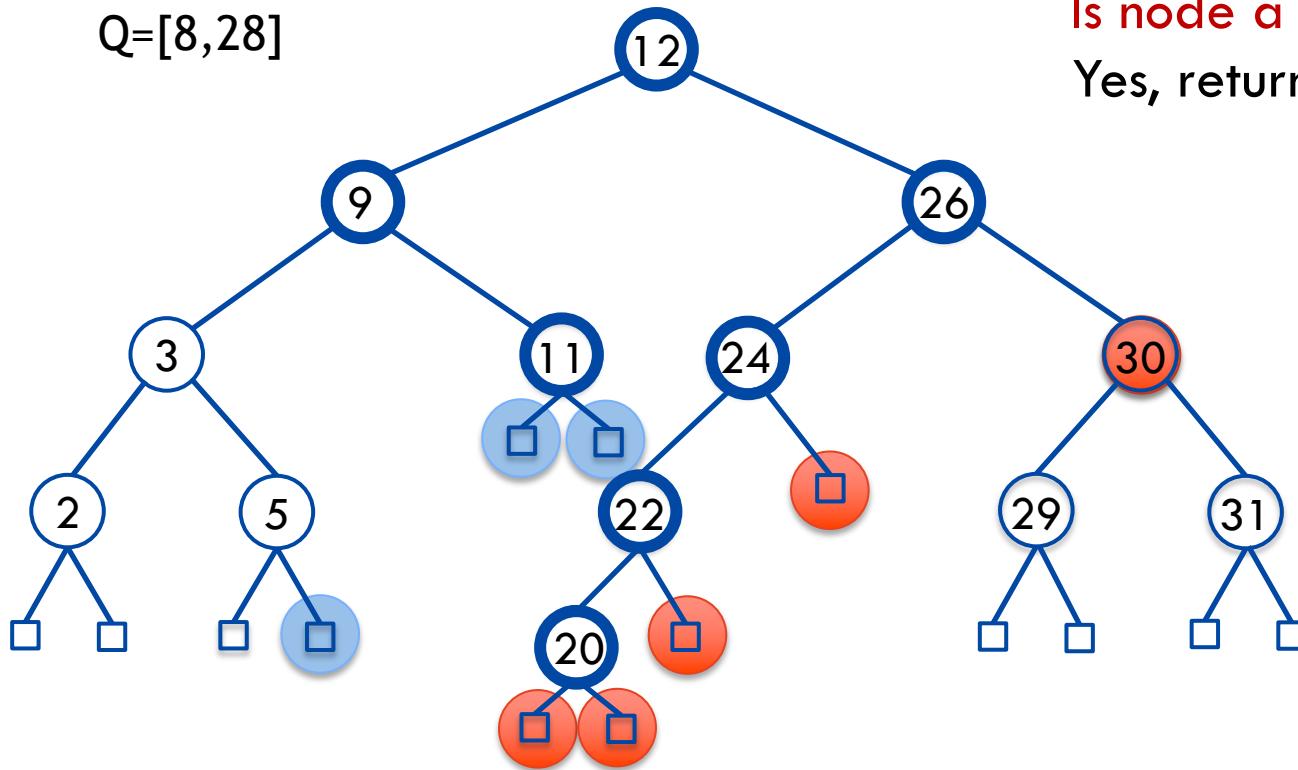
Yes, return 20 and continue the search in both subtrees.

Range queries

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

$Q = [8, 28]$

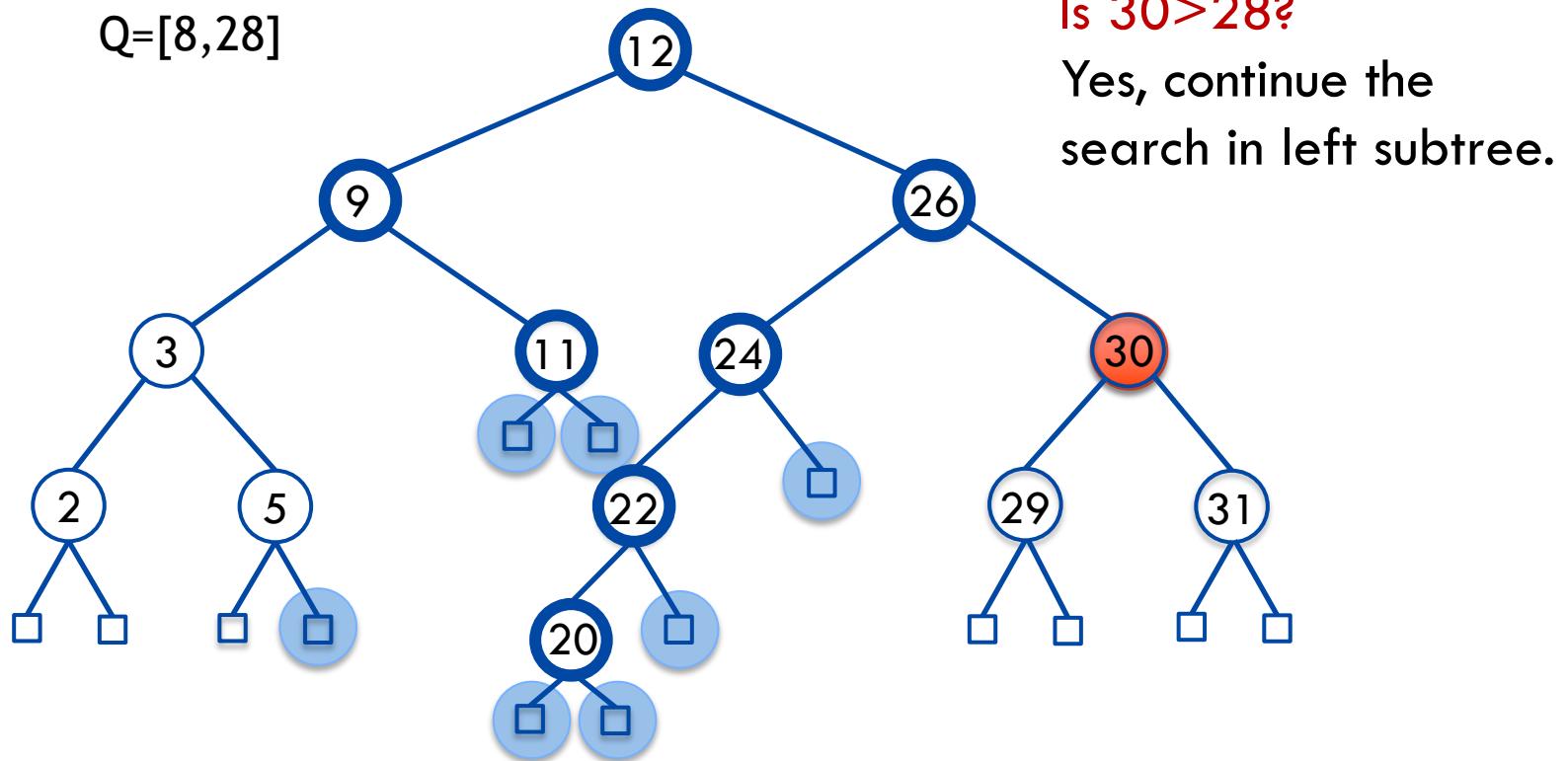
Is node a leaf ($\times 4$)?
Yes, return \emptyset .



Range queries

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

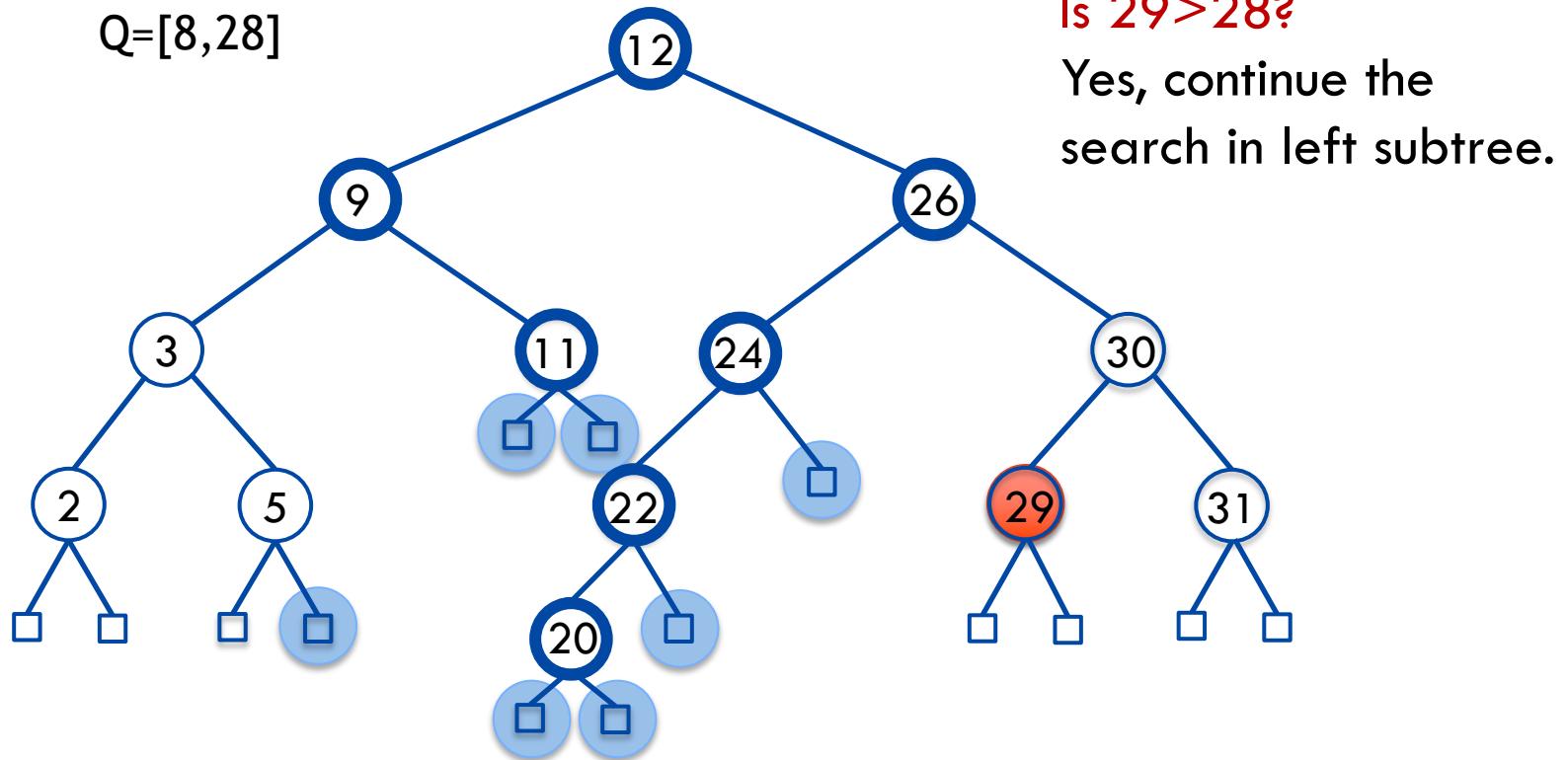
$Q = [8, 28]$



Range queries

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

$Q=[8,28]$

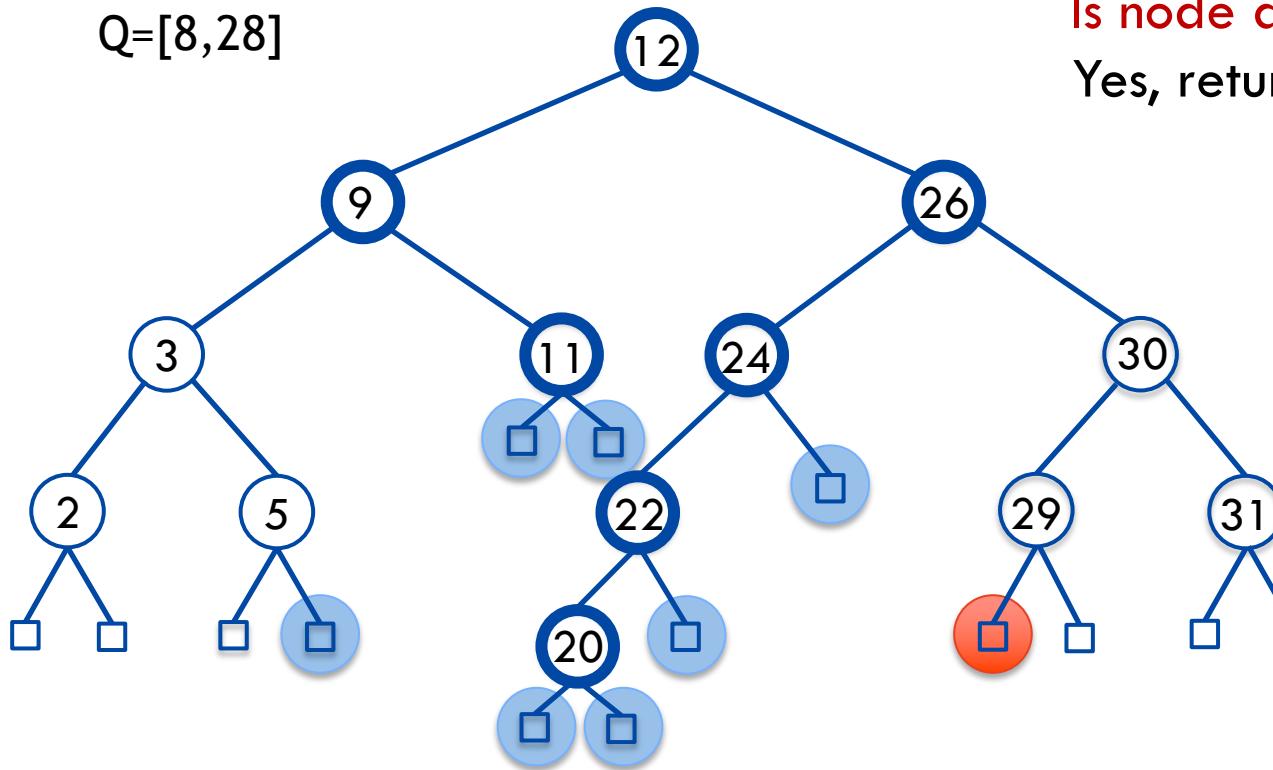


Range queries

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

$$Q=[8,28]$$

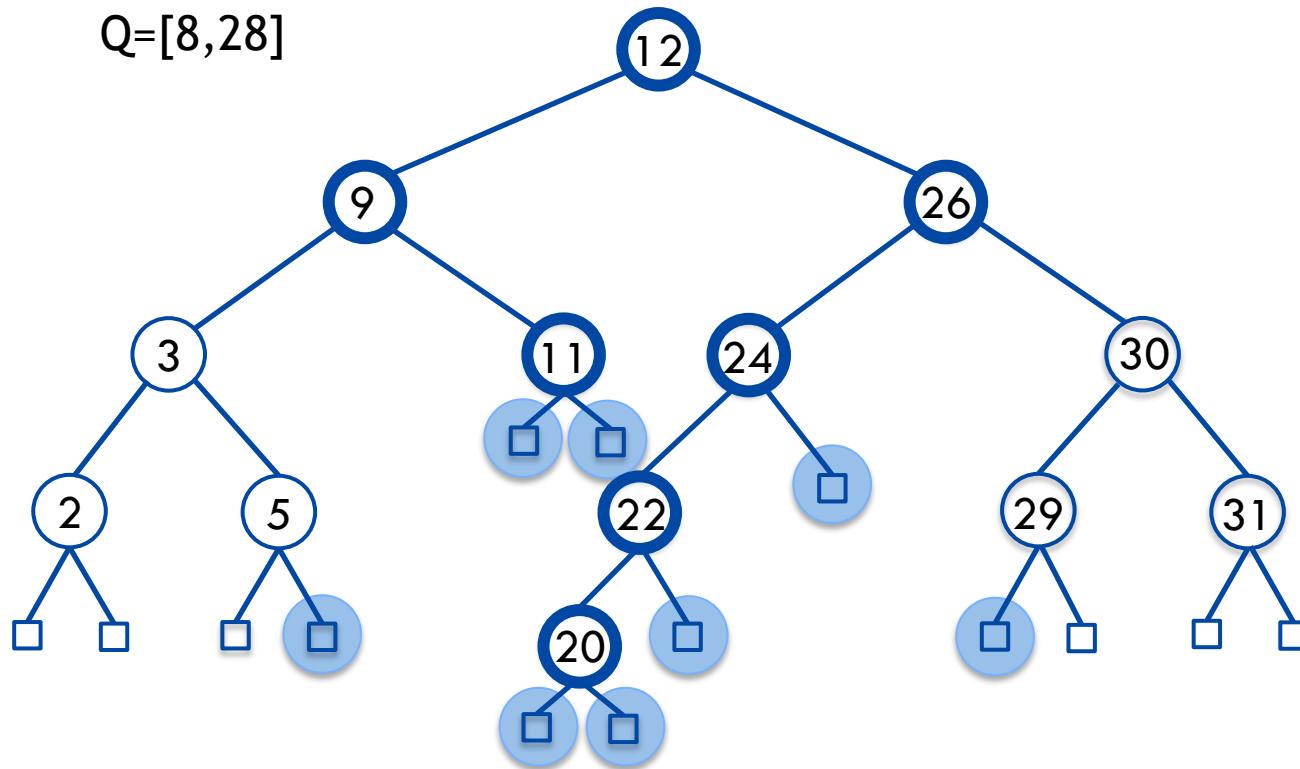
Is node a leaf?
Yes, return \emptyset .



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

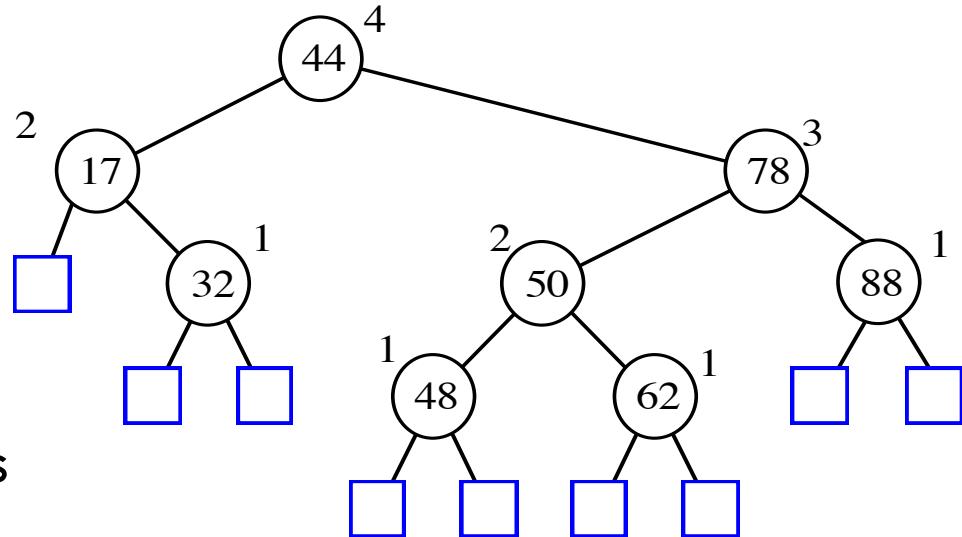
A family of balanced BST implementations that use the idea of keeping a “rank” for every node, where $r(v)$ acts as a proxy measure of the size of the subtree rooted at v

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

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

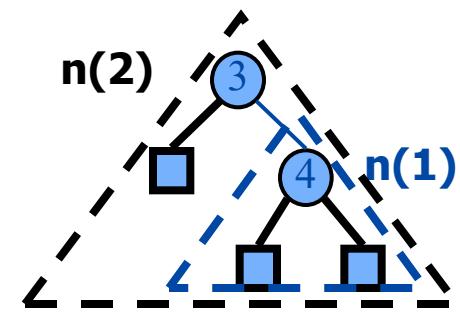
AVL Tree Definition

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 $O(\log n)$.

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

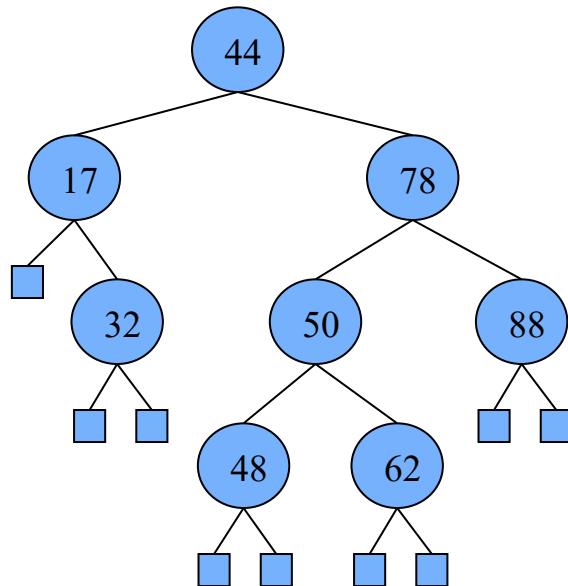
Insertion in AVL trees

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

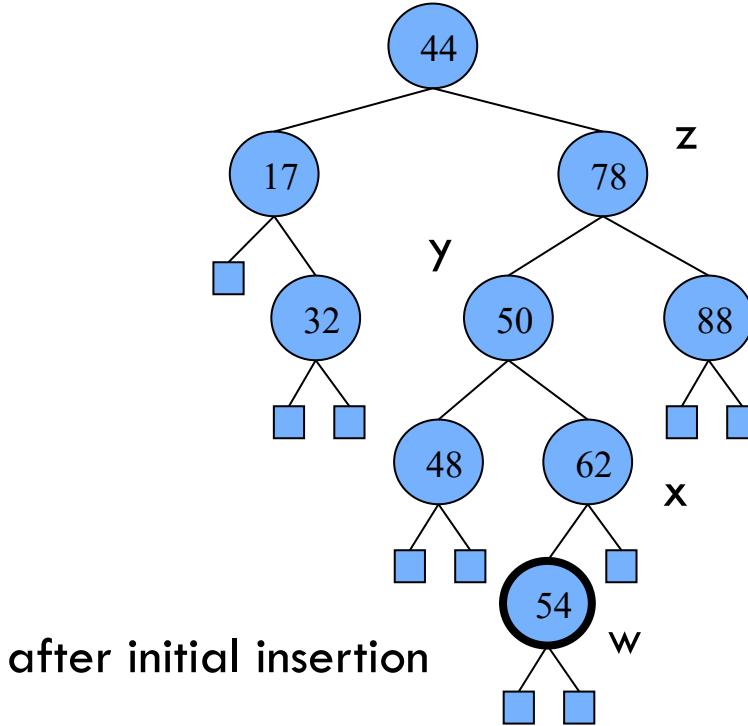
1. If k is in the tree, search for k ends at node holding k
There is nothing to do so tree structure does not change
2. If k is not in the tree, search for k ends at external node w .
Make this be a new internal node containing key k
3. The new tree has BST property, but it may not have AVL balance property 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

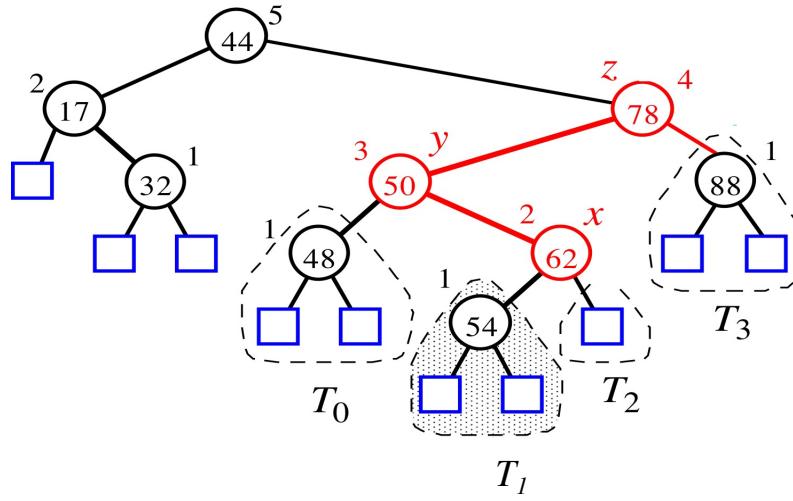


before inserting 54



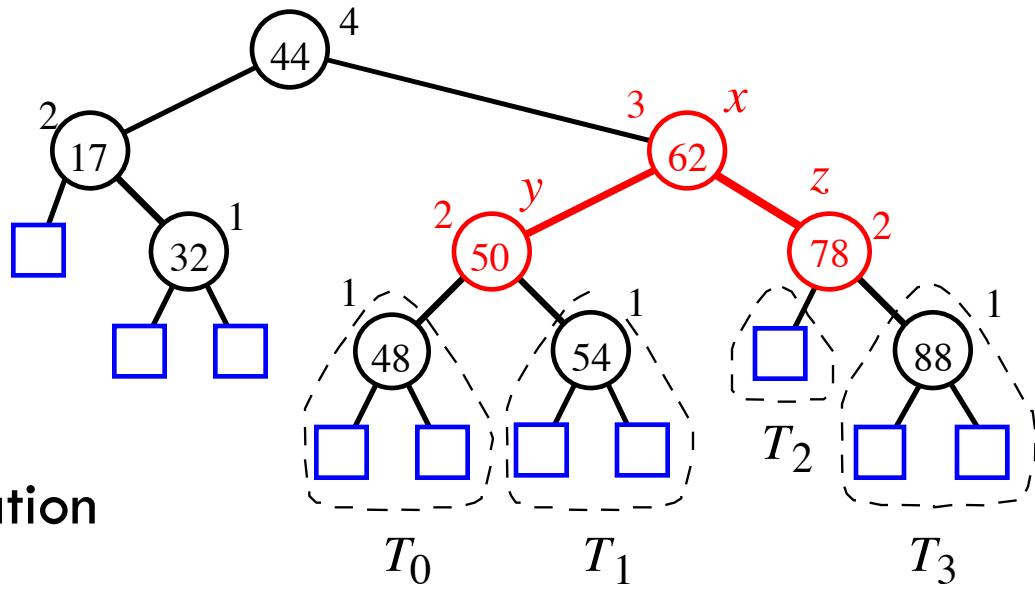
after initial insertion

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.

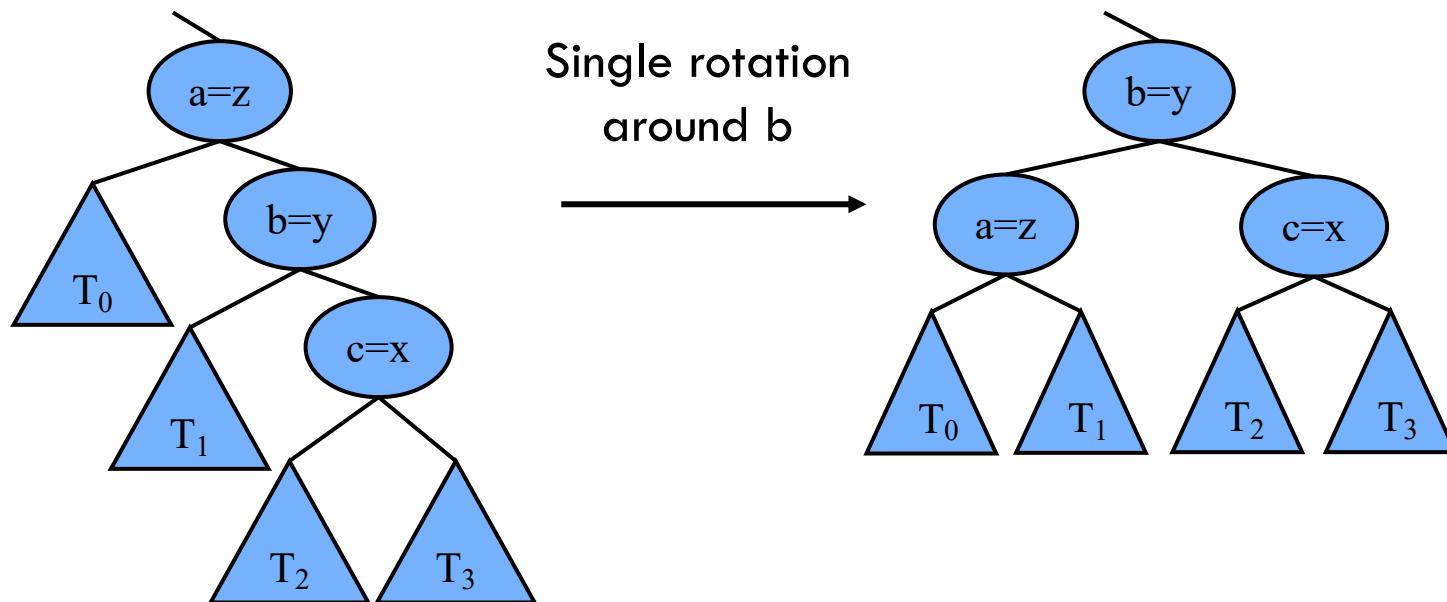
Thus, we can maintain the height only using $O(h)$ work per insert

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.

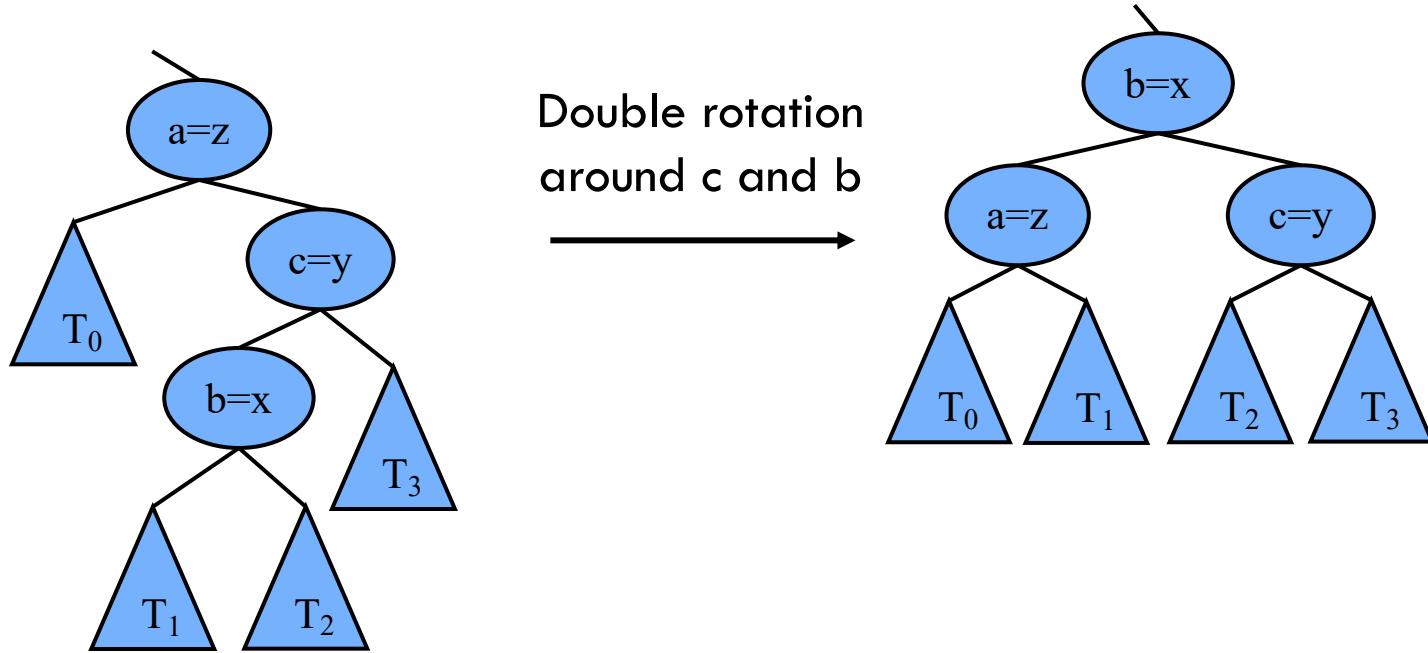


Improving Balance: Trinode Restructuring

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*):

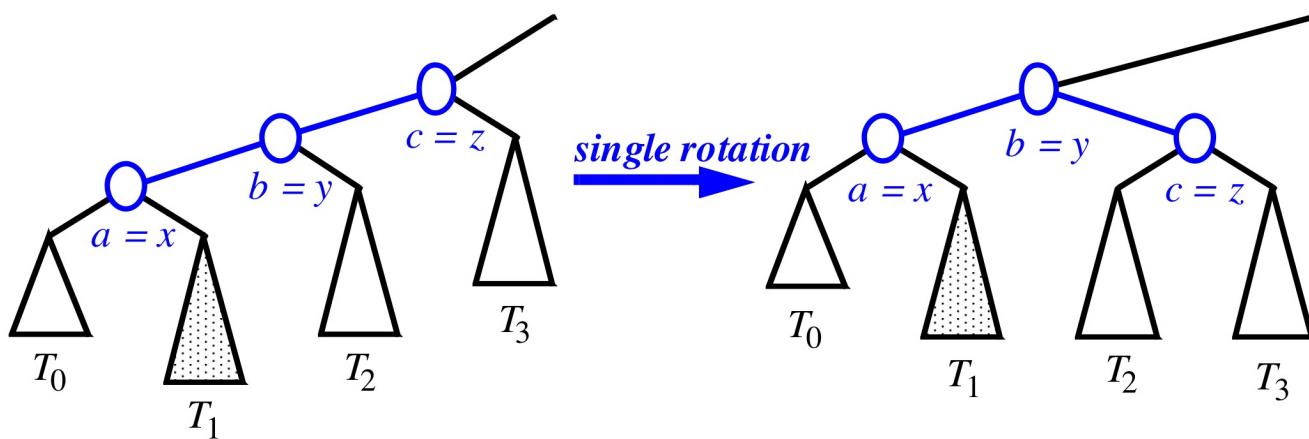
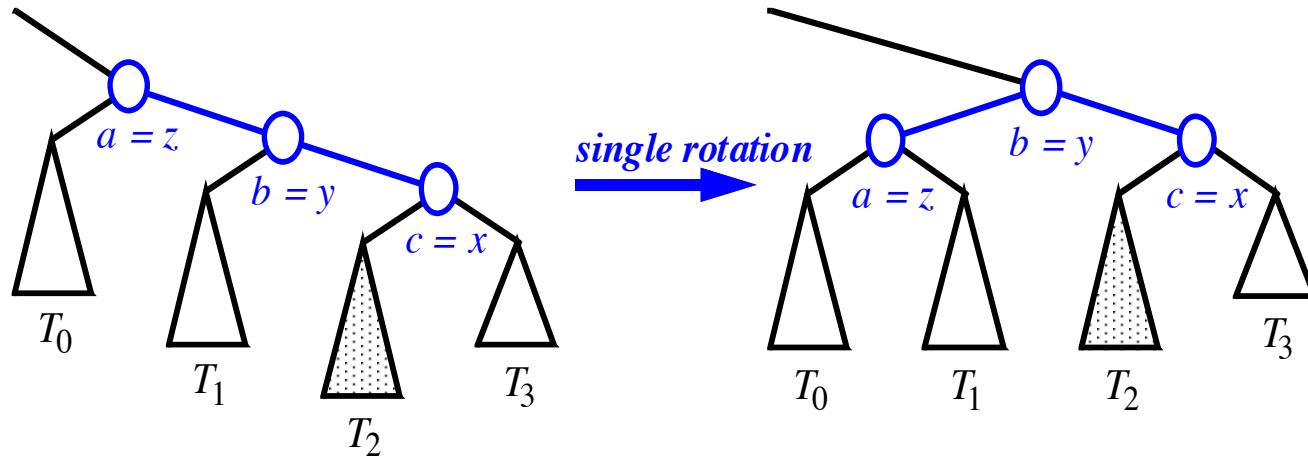
input A node *x* of a binary search tree *T* that has both a parent *y* and a grandparent *z*

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 right 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*

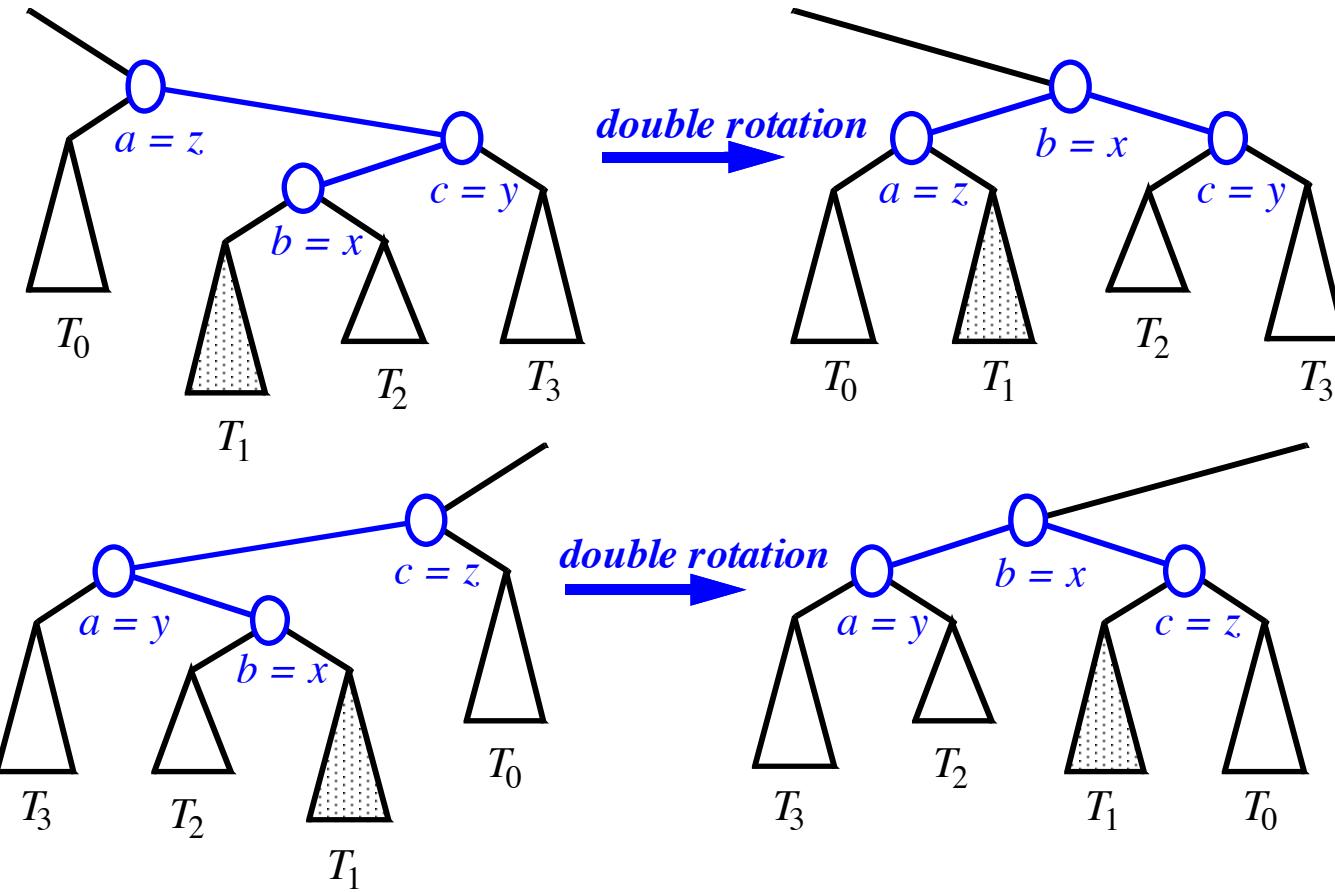
Trinode Restructuring (when done by Single Rotation)

Single Rotations:



Trinode Restructuring (when done by Double Rotation)

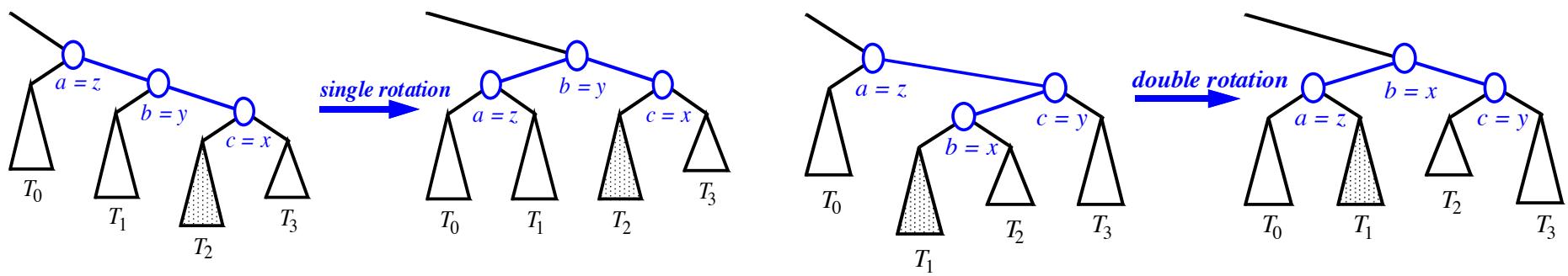
Double rotations:



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.



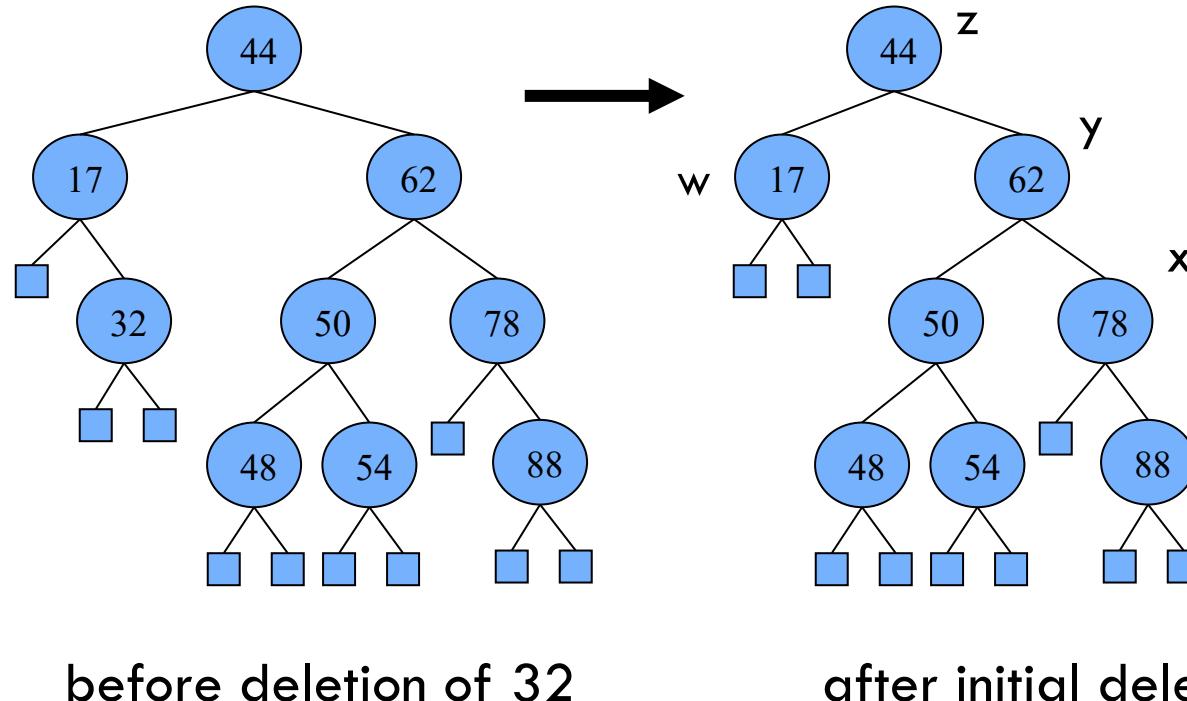
Removal in AVL trees

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

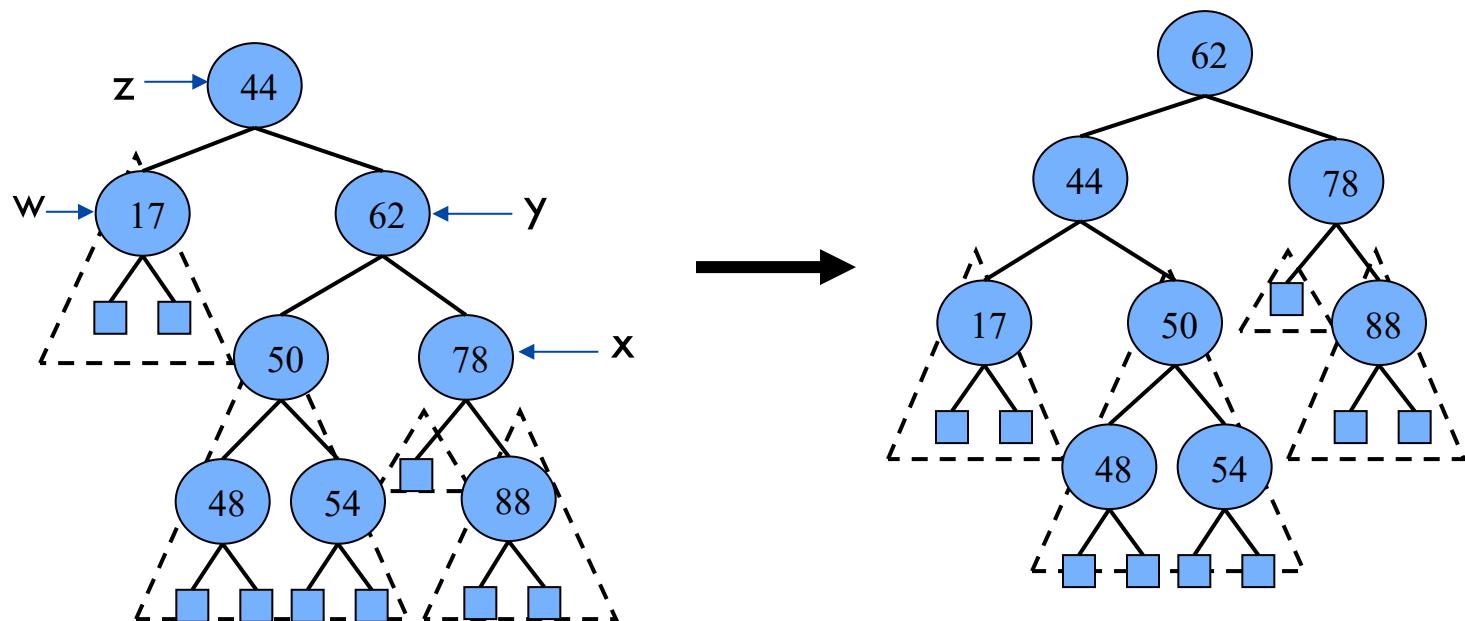
Re-establishing AVL property

- 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 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 entail 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)

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)

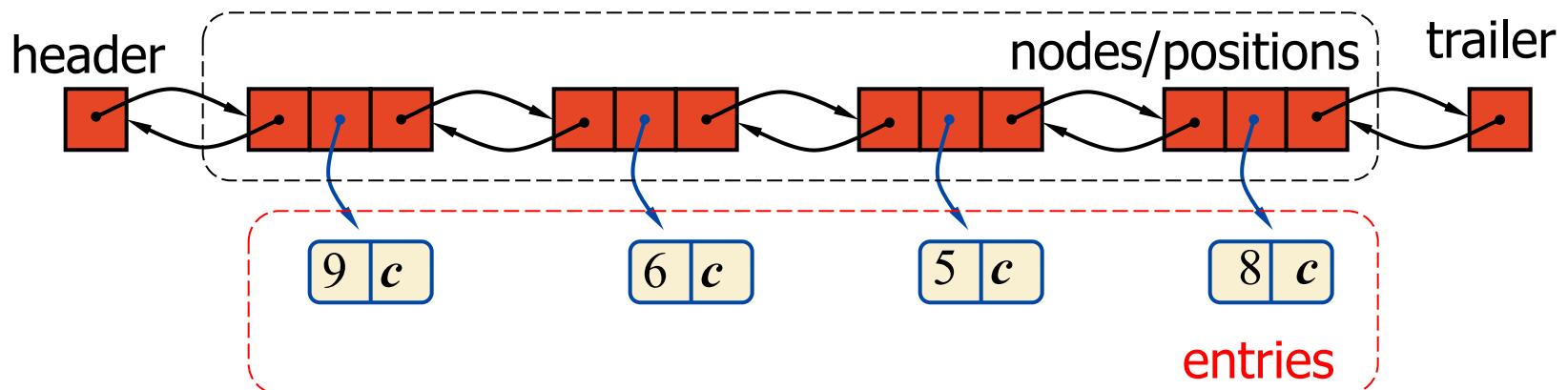
subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2

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)



Tree-Based (sorted) Map

We can implement a sorted map using an AVL 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 there is a total ordering on the keys.

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 5: Priority Queues [GT 5]

Dr. André van Renssen
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

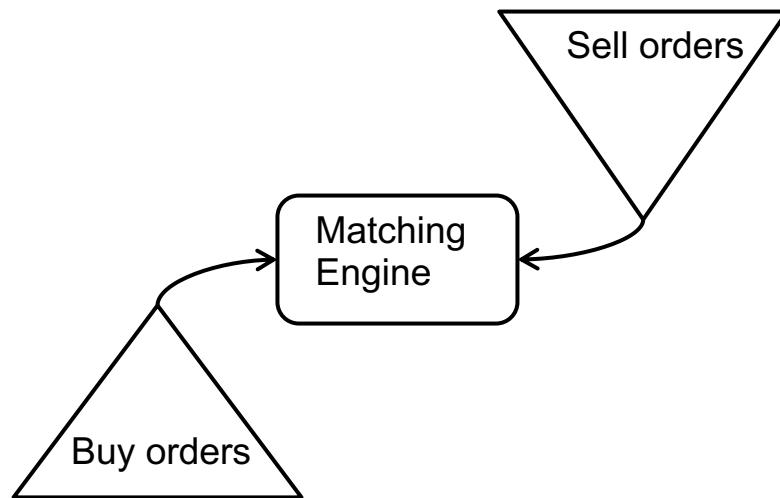
Sellers post offers (asks) to sell a number of shares of a given stock at or above a specified price.

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)
```

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 $O(n^2)$

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

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

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, 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, 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

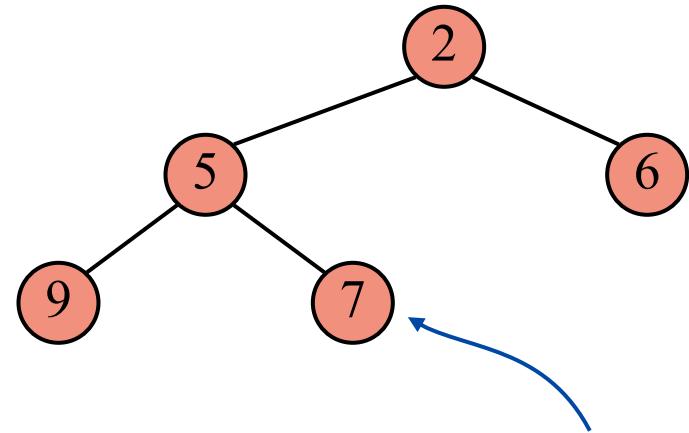
i	A	i
1	7, <u>4</u> , 8, 2, 5, 3, 9	0
2	4, 7, <u>8</u> , 2, 5, 3, 9	2
3	<u>4</u> , 7, 8, <u>2</u> , 5, 3, 9	0
4	2, 4, <u>7</u> , 8, <u>5</u> , 3, 9	2
5	2, <u>4</u> , 5, 7, 8, <u>3</u> , 9	1
6	2, 3, 4, 5, 7, 8, <u>9</u>	6

```
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 properties:

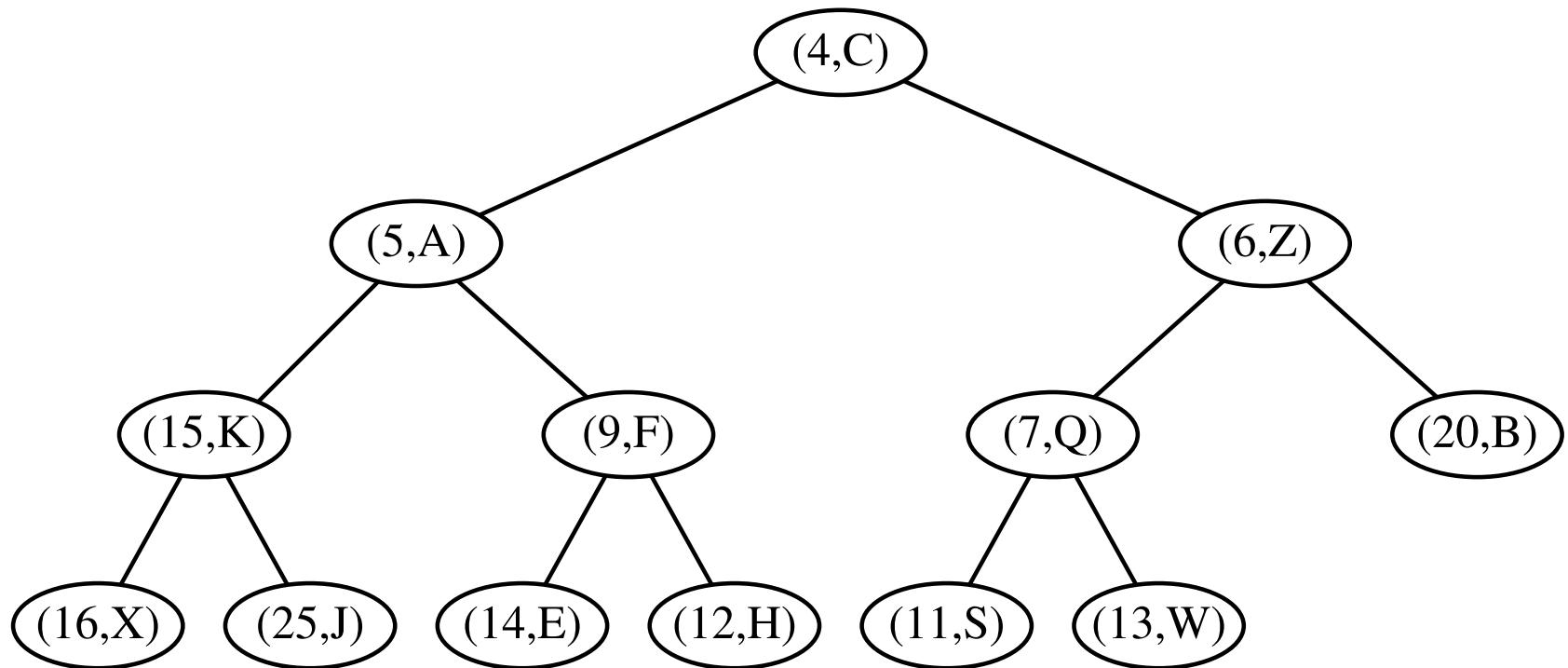
1. **Heap-Order:** for every node $m \neq \text{root}$,
 $\text{key}(m) \geq \text{key}(\text{parent}(m))$



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

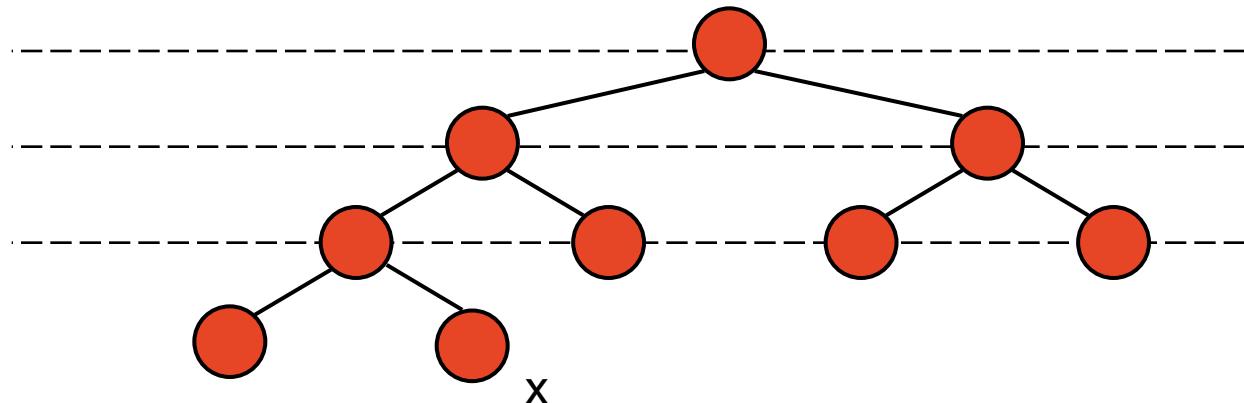


Minimum of a Heap

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

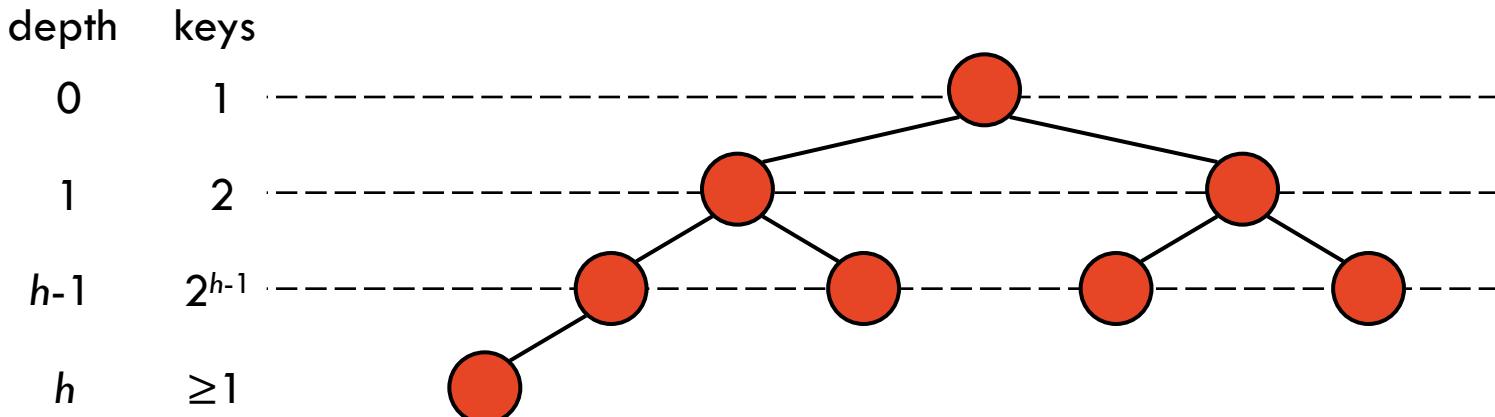


Height of a Heap

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

Proof:

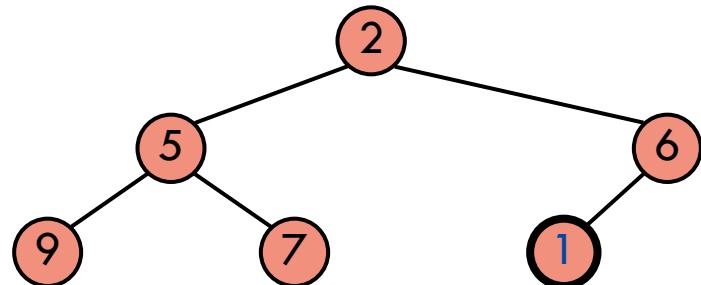
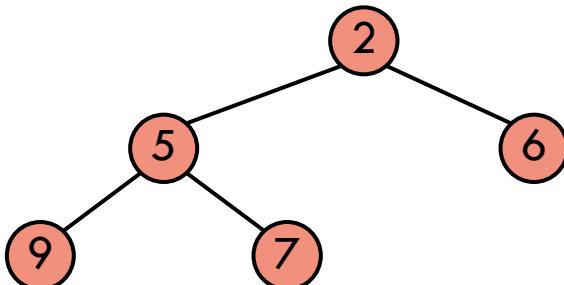
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h - 1$ and at least one key at depth h , we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, applying \log_2 on both sides, $\log_2 n \geq h$



Insertion into a Heap

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

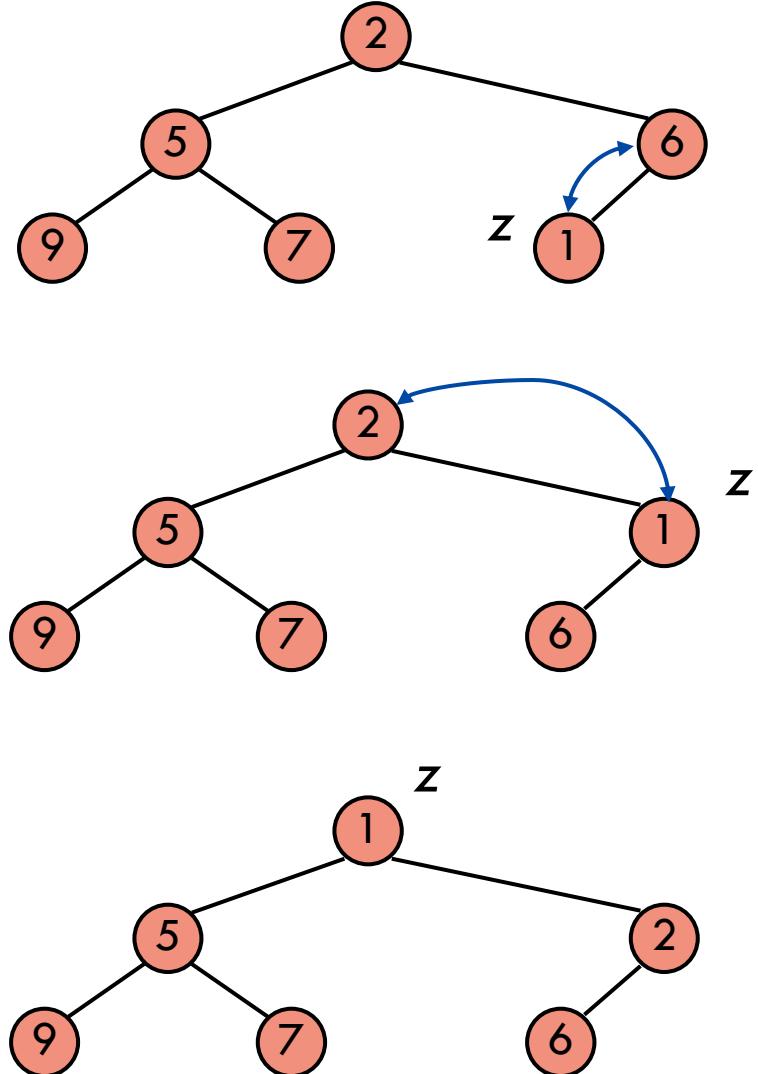
insert(1)



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)
```



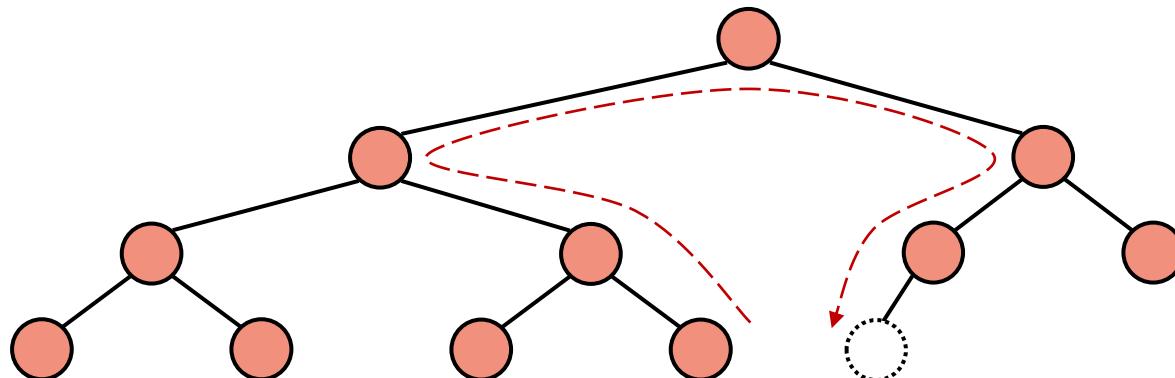
Correctness: after swapping the subtree rooted at **z** has the property

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

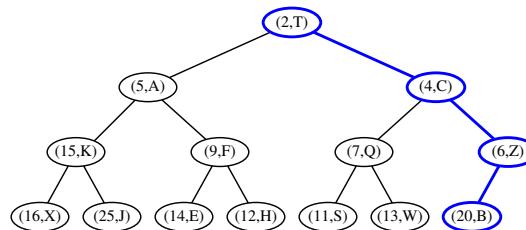
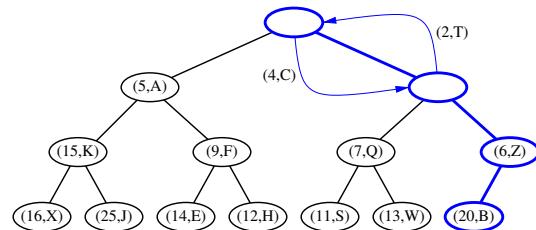
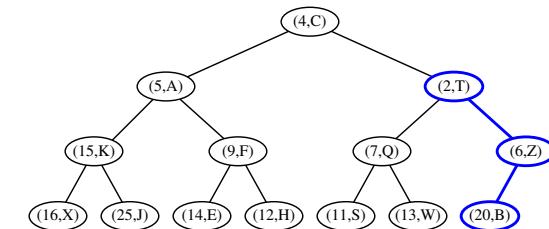
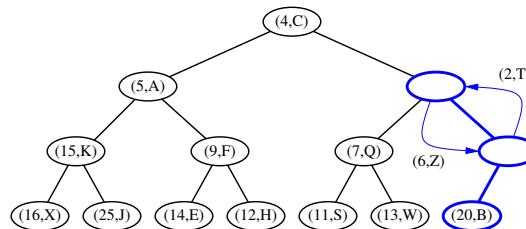
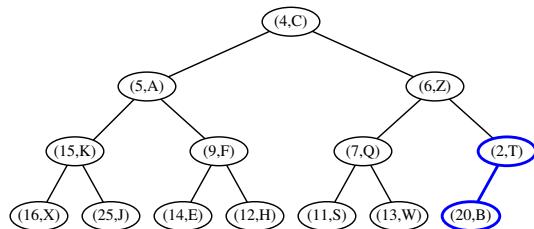
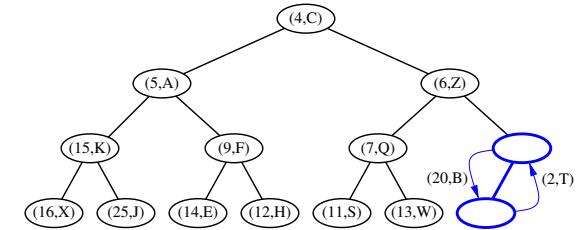
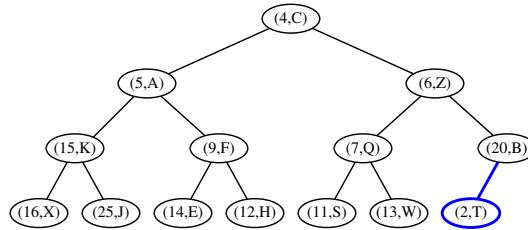
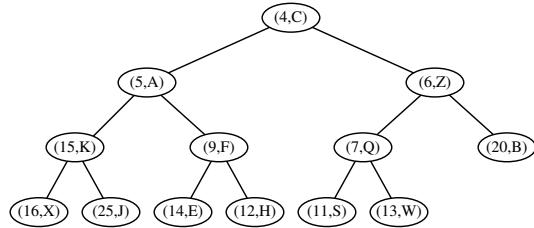
Finding the position for insertion

- 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



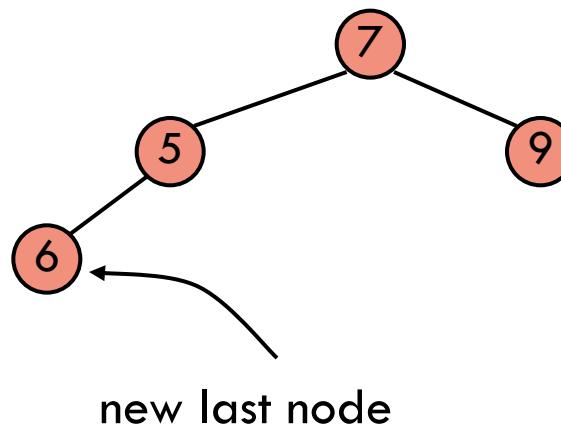
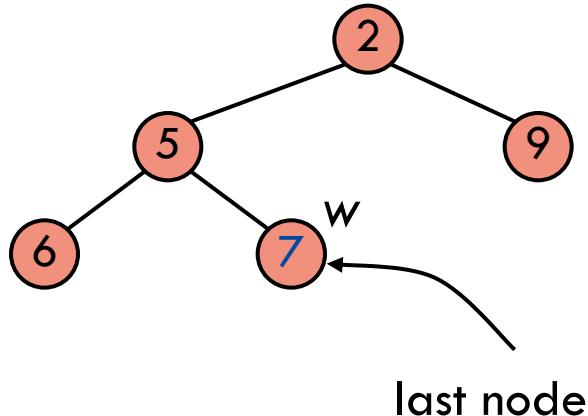
Example insertion



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()`



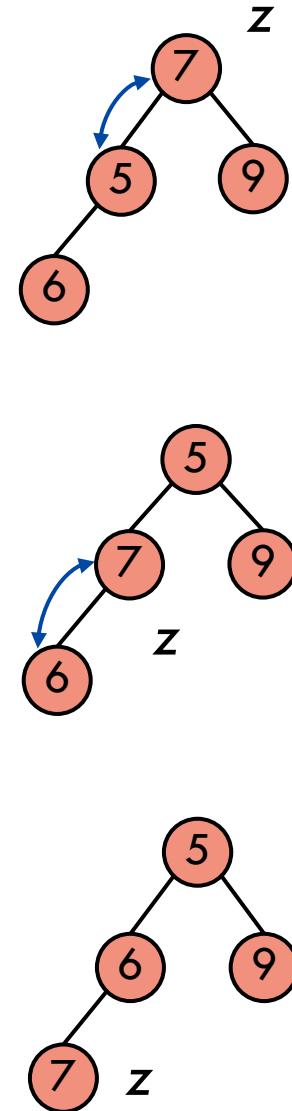
Downheap

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

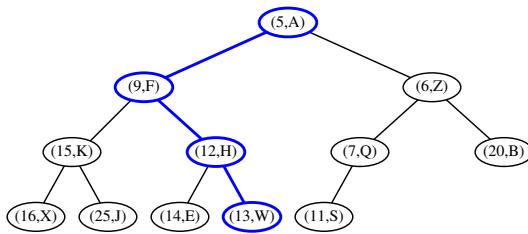
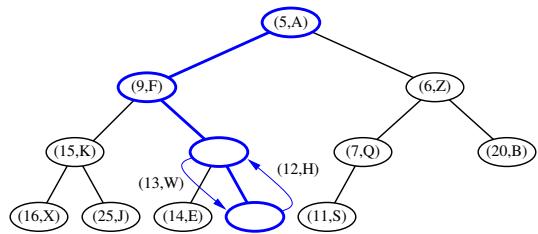
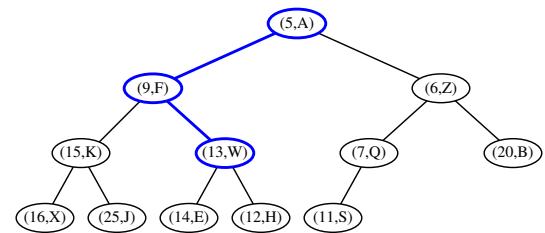
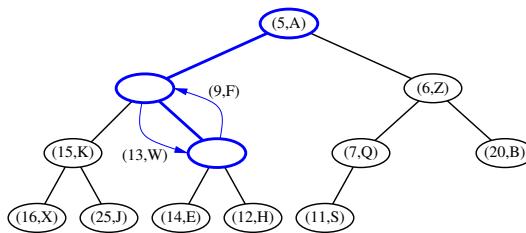
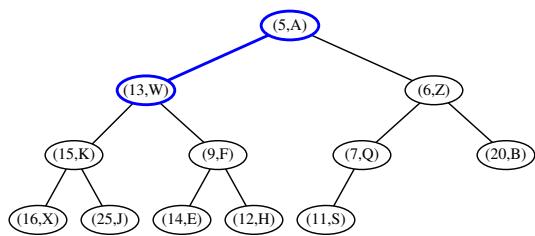
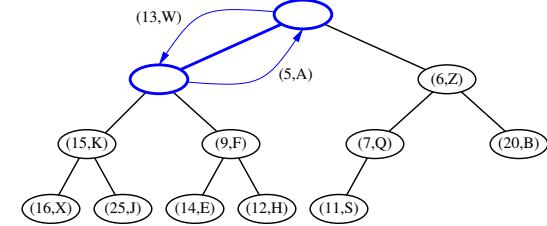
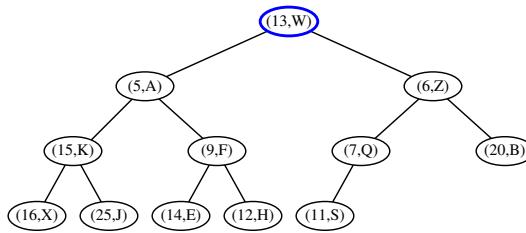
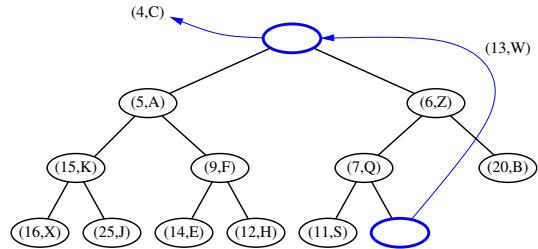
```
def down_heap(z):
    while z has child with
        key(child) < key(z) do
            x ← child of z with smallest key
            swap keys of x and z
            z ← x
```

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**



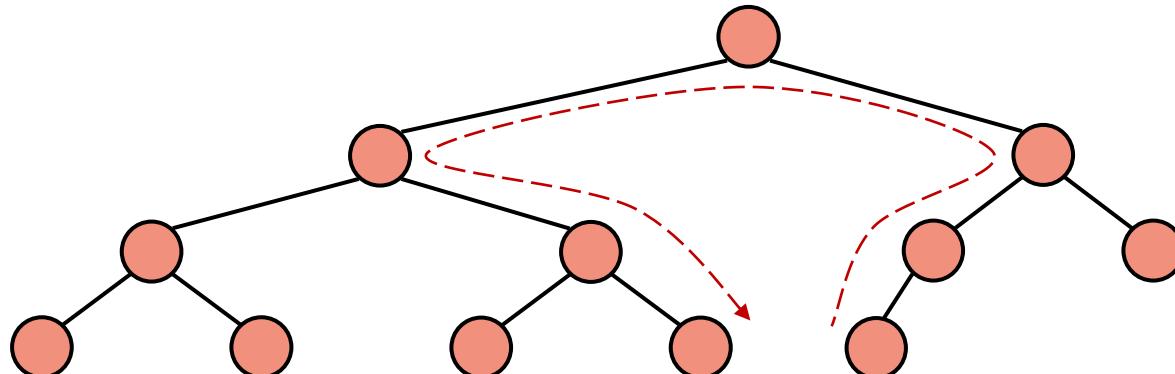
Example removal



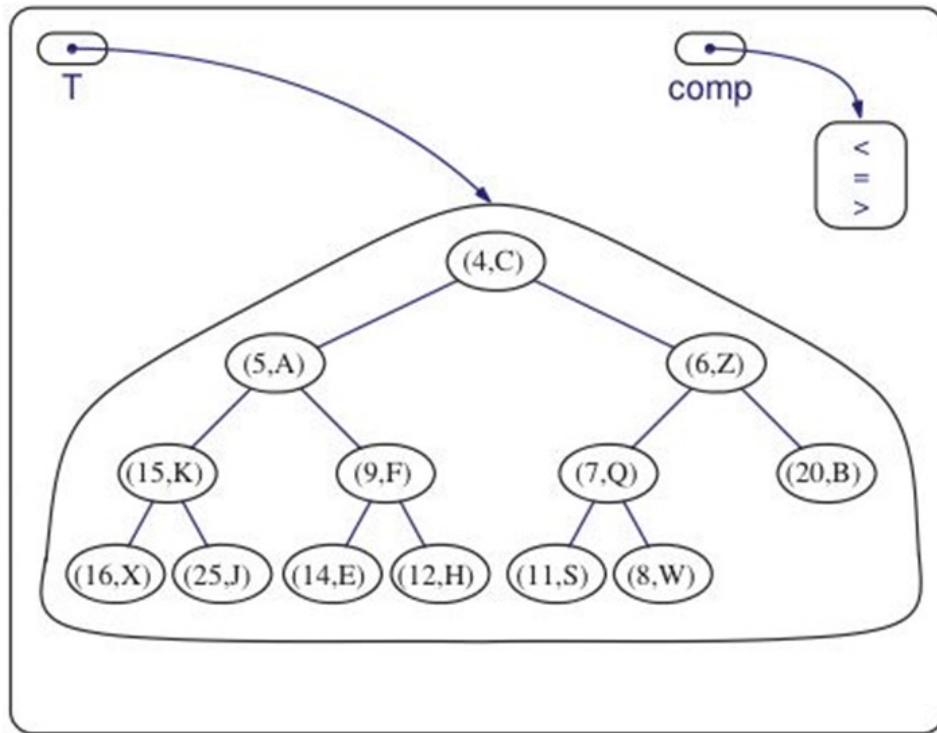
Finding next last node after deletion

- 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 deletion 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

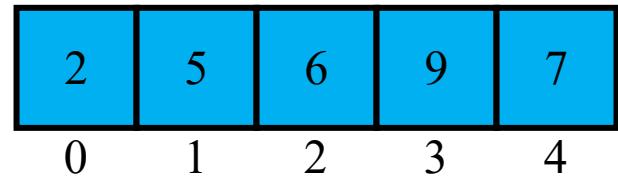
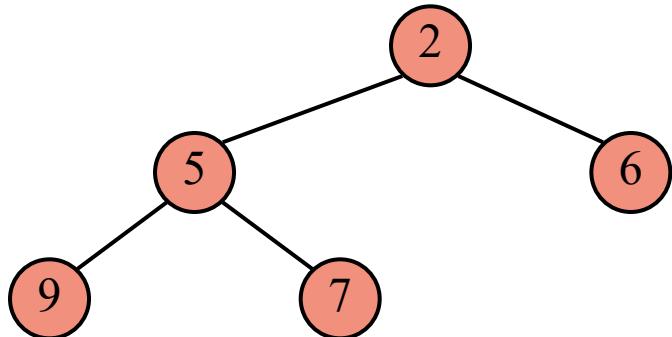
We can represent a heap with n keys by means of an array of length n

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 (all given a pointer to e):

- `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?

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

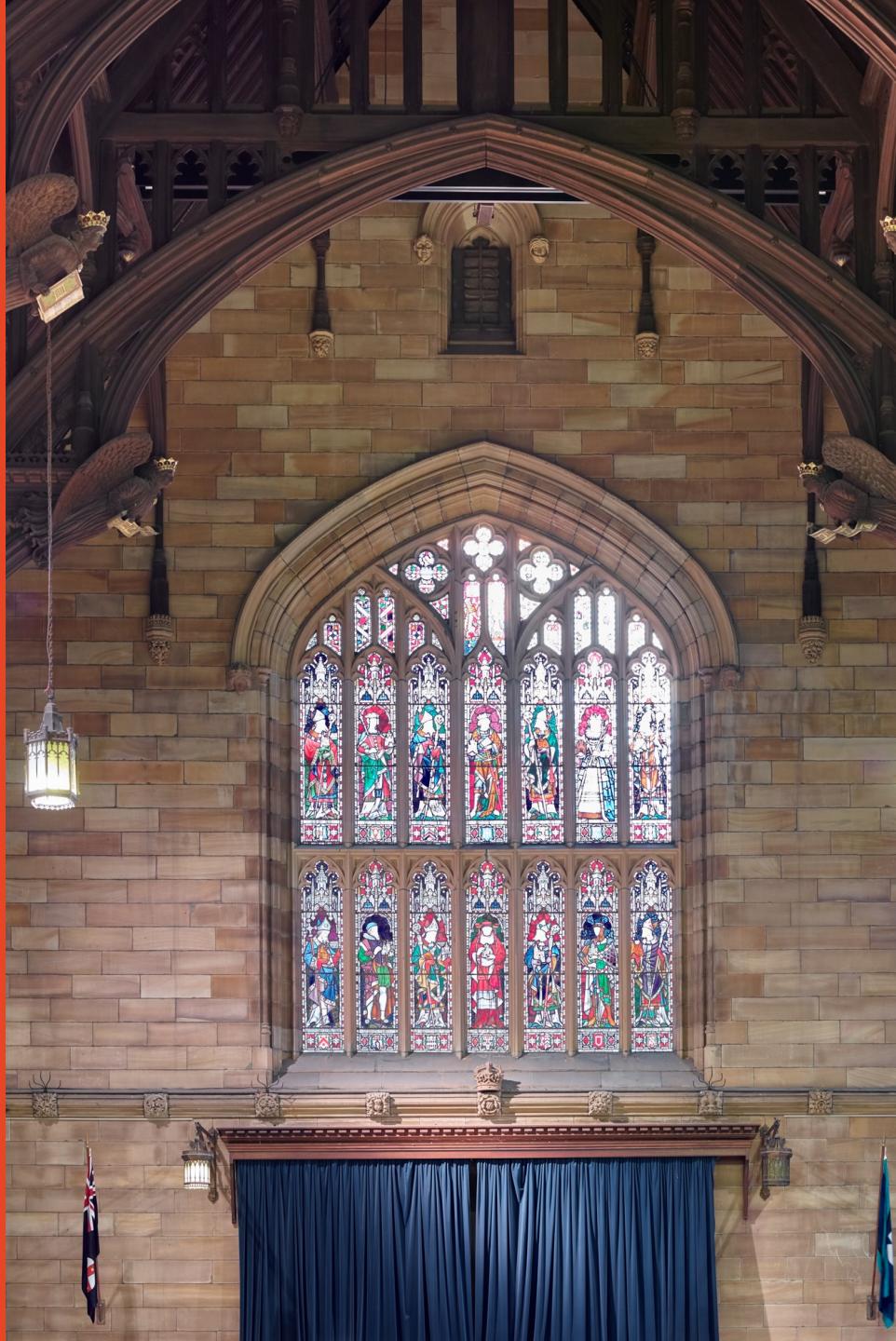
Lecture 6: Hash tables [GT 6.1-4]

Dr. André van Renssen
School of Computer Science

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



THE UNIVERSITY OF
SYDNEY



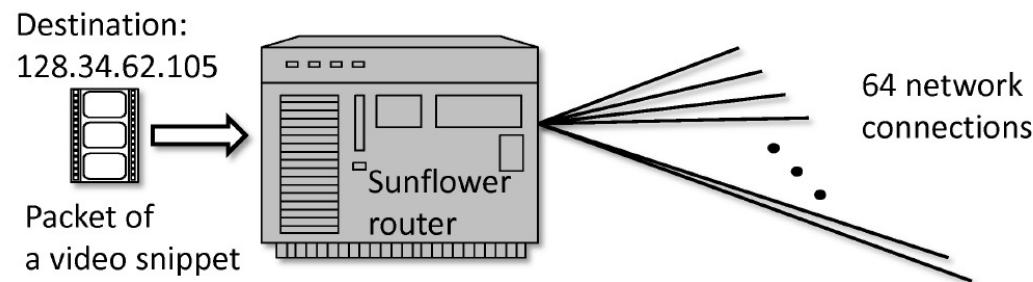
Application: Network Routers

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

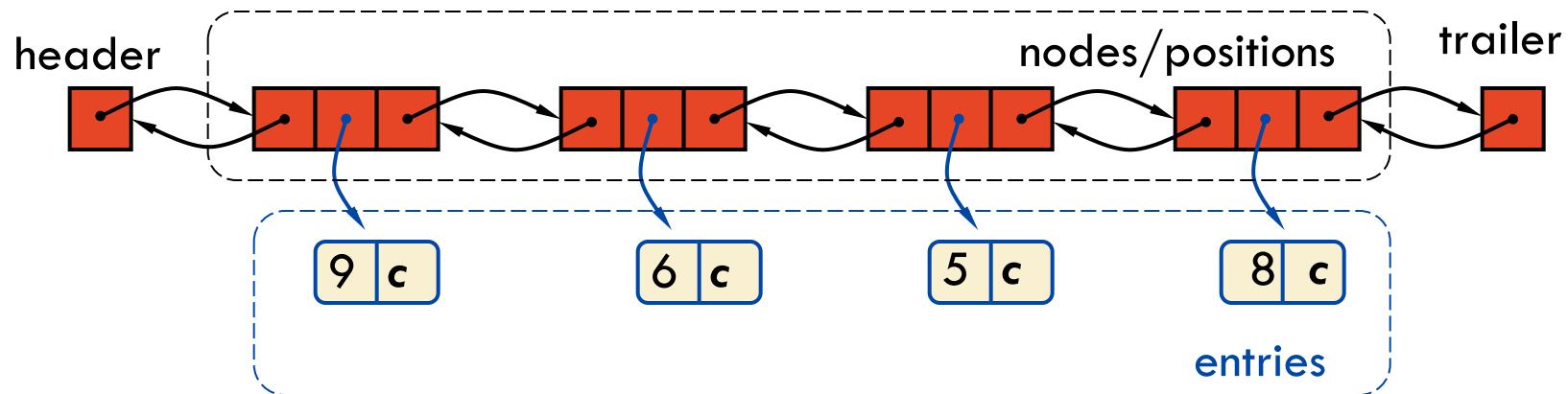
Example

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 if the key doesn't exist yet since we can insert the new item at the beginning or at the end of the sequence
- **put, 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)

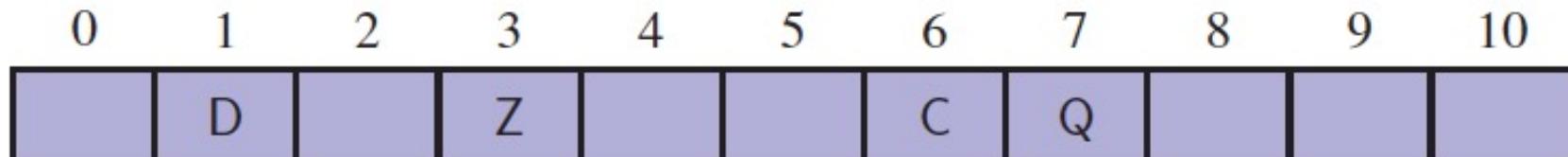
Simple Implementation with restricted keys

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 1,000,000,000 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

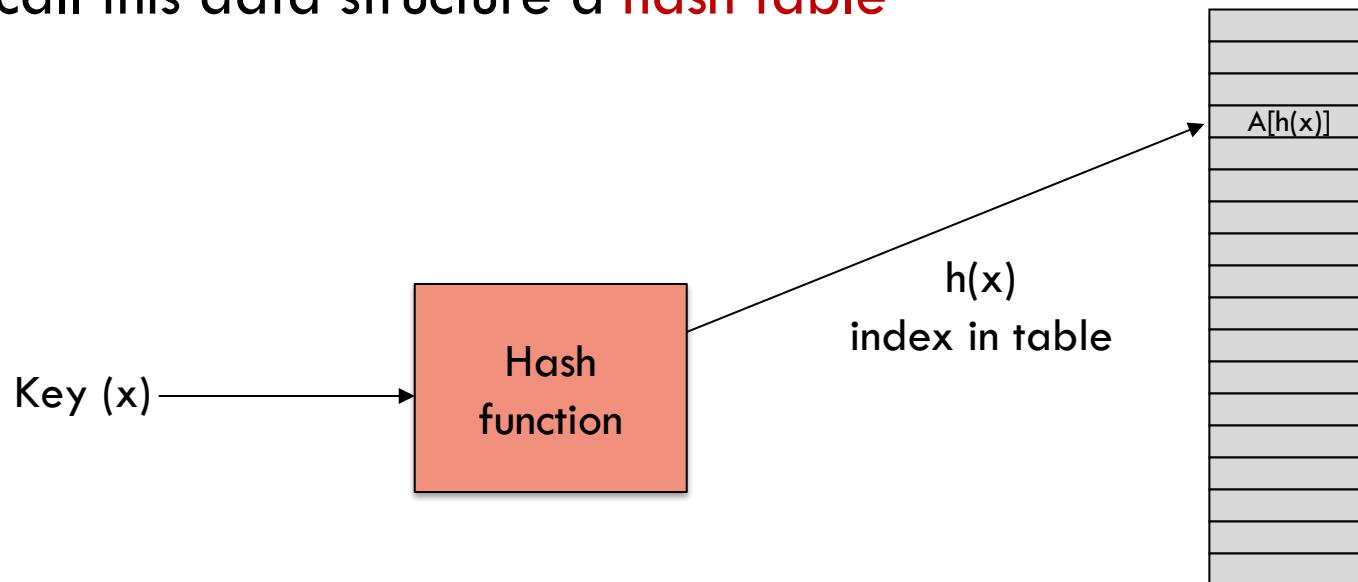
Unable to handle more general keys like strings

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

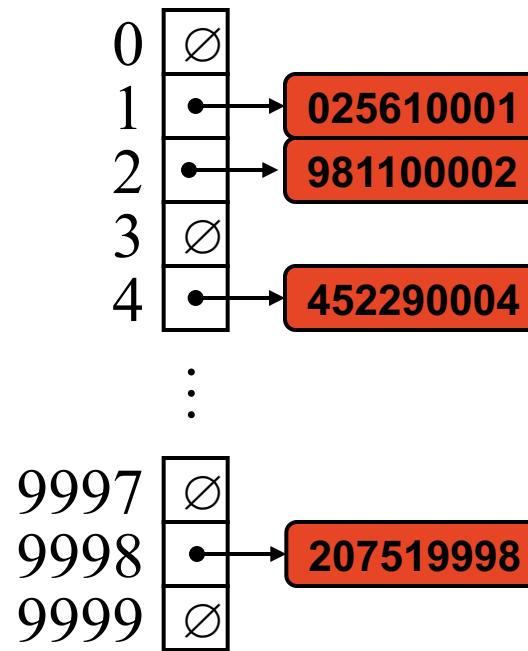
A **hash table** for a given key type K consists of

- 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
 - $35 \bmod 10 = 5$
 - $20 \bmod 10 = 0$
- 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:

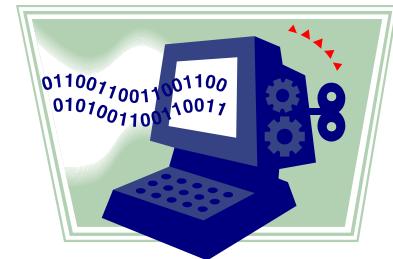
$h_2 : \text{integers} \rightarrow [0, N - 1]$

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

The diagram illustrates the composition of two functions. It shows a red arrow pointing from the input x to the function $h_1(x)$. A second red arrow points from $h_1(x)$ to the final result $h(x) = h_2(h_1(x))$.

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 codes 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 don't 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

Suppose that $[0, M]$ is the range of our keys and we need a hash function with range in $[0, N-1]$

Let H be a family of such hash functions. We say that H is 2-universal if picking h uniformly at random (UAR) from H yields

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

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 of n keys is at most 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 $[1, p-1]$ 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

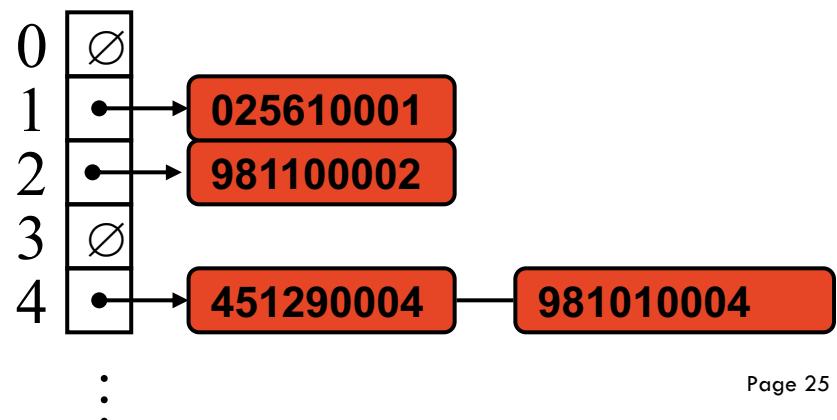
Get, put, and remove operations are delegated to the appropriate list, where put needs to search through the list to replace the existing value of the key, if present

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

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

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

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



Performance of Separate Chaining

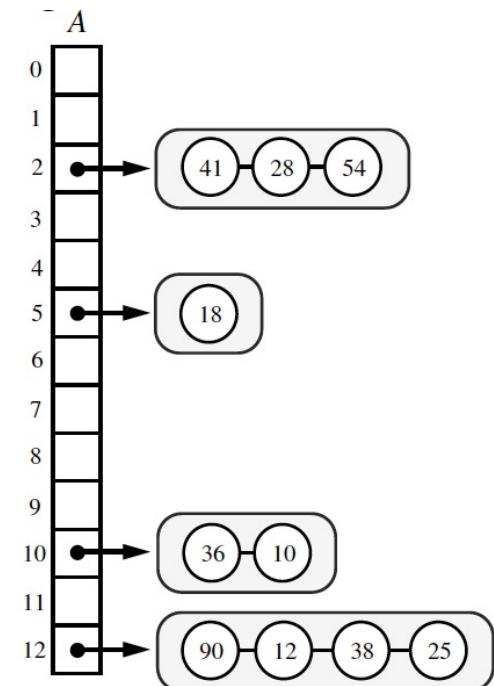
Assume that our hash function, maps n keys to independent uniform random values in the range $[0, N-1]$.

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

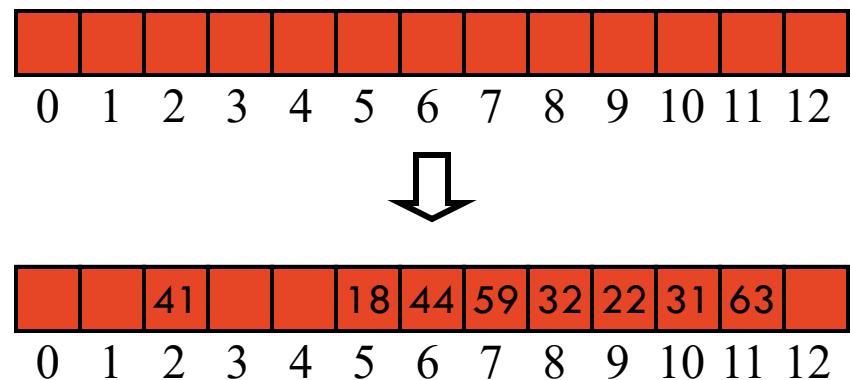
Open addressing: the colliding item is placed in a different cell of the table

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

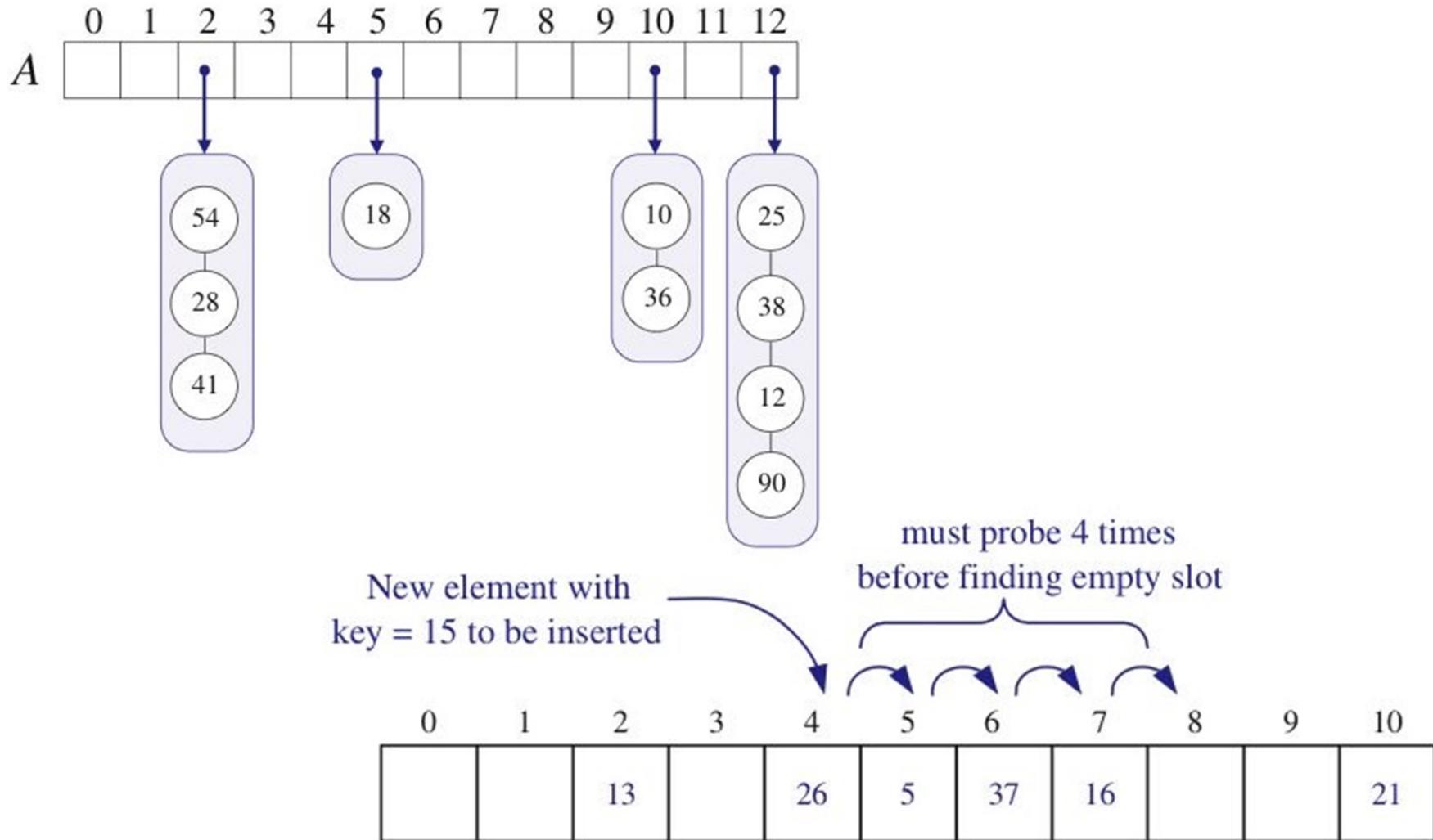
- Each table cell inspected is referred to as a probe
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

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

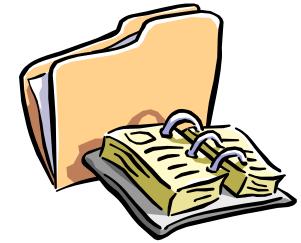
18 [5], 41 [2], 22 [9],
44 [5], 59 [7], 32 [6],
31 [5], 63 [11]



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 then  
            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, to tell them apart from empty cells

- **get(k)**: must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell
 - **put(k,v)**: 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.
 - **remove(k)**: search for the entry as in get(k). If found, replace it with the special item *DEFUNCT* and return element v
- 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, v) 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 the 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 the load factor of a hash table reaches the given bound, 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

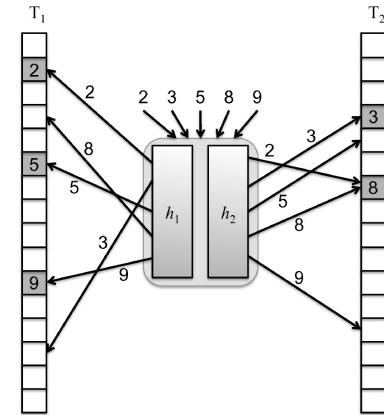
Main problem with the methods we've seen so far is that operations take $O(n)$ time in the worst-case.

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 hash tables, 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

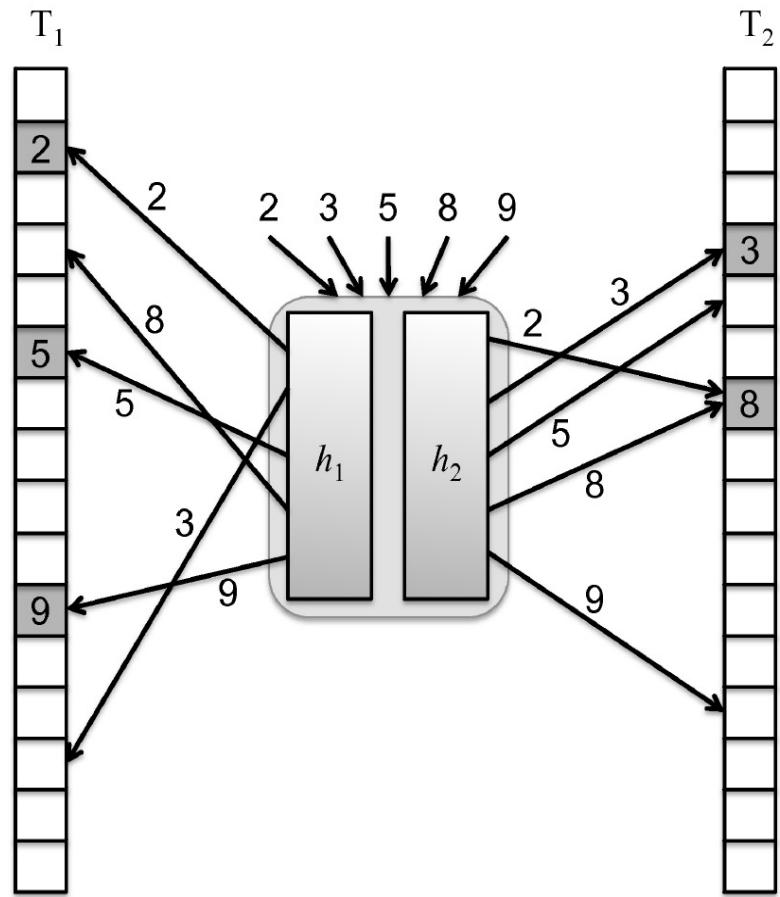
For an item with key k , there are only **two possible places** where we are allowed to store the item: $T_1[h_1(k)]$ or $T_2[h_2(k)]$

This restriction, simplifies lookup dramatically, while still allowing worst-case $O(1)$ running time for get 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 $\mathcal{O}(1)$ 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.

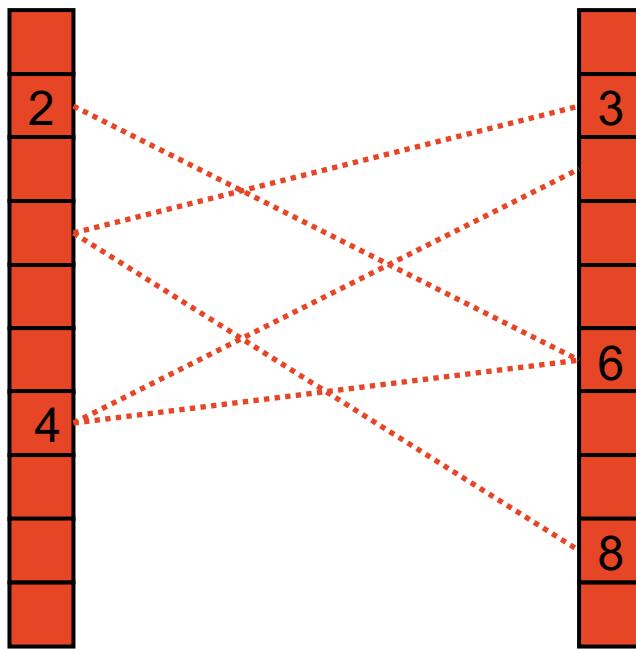
The Common Cuckoo is a brood parasite—it lays its egg in the nest of another bird after first evicting an egg out of that nest.



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.

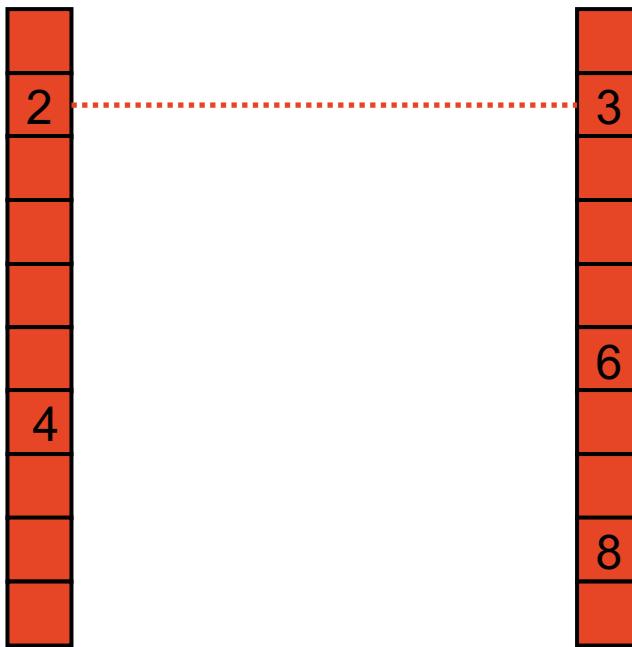
Example Eviction Sequence

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



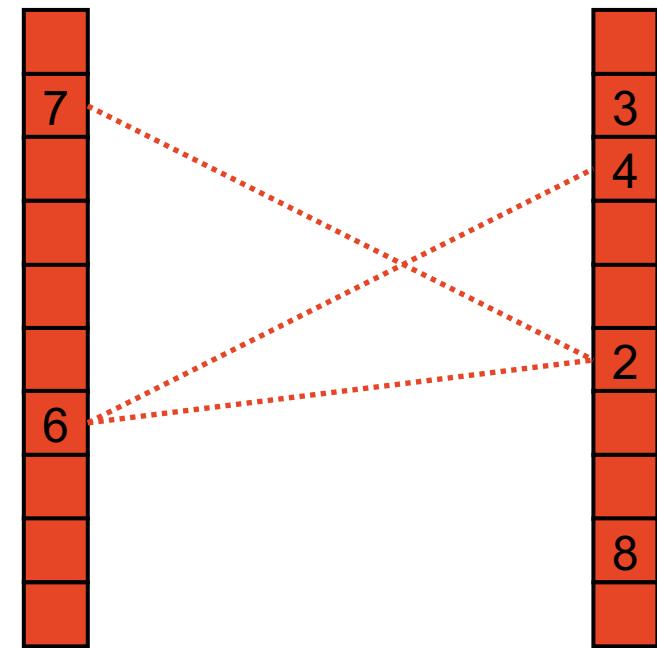
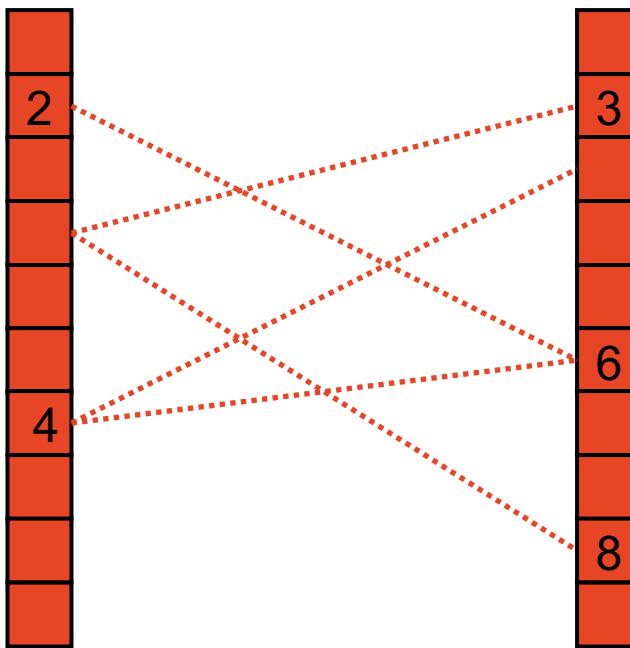
Example Eviction Sequence

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



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 evict 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.